第 3 章

栈与队列

提纲

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

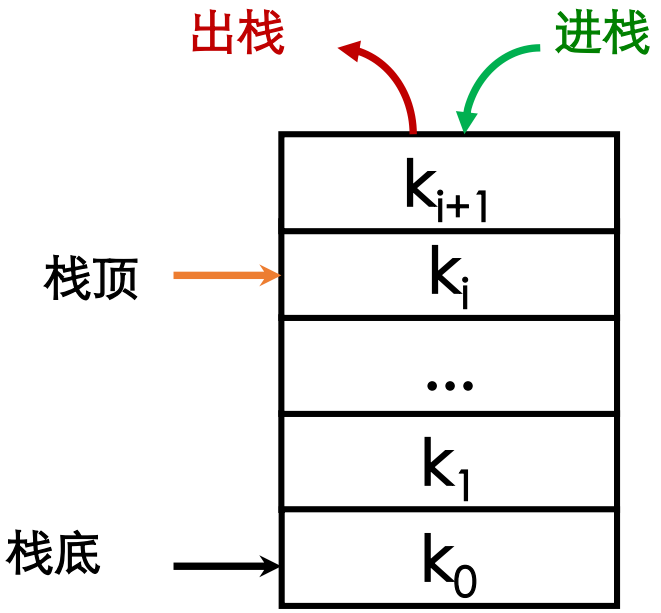
- **栈（Stack）**
  - 运算只在表的**一端**进行
- **队列（Queue）**
  - 运算只在表的**两端**进行

栈

- 后进先出（**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
  - 进制转换
  - 括号语句判断
  - 表达式求值
  - 函数调用
  - 深度优先搜索



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*| *ai*∈ElemSet， *i*=1,2,…*n*, *n* > 0 } 或*Φ*，即空表；ElemSet为元素集合。

数据关系：

{<*ai*, *ai*+1>|*ai*, *ai*+1∈ElemSet， *i*=0,1,…*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。

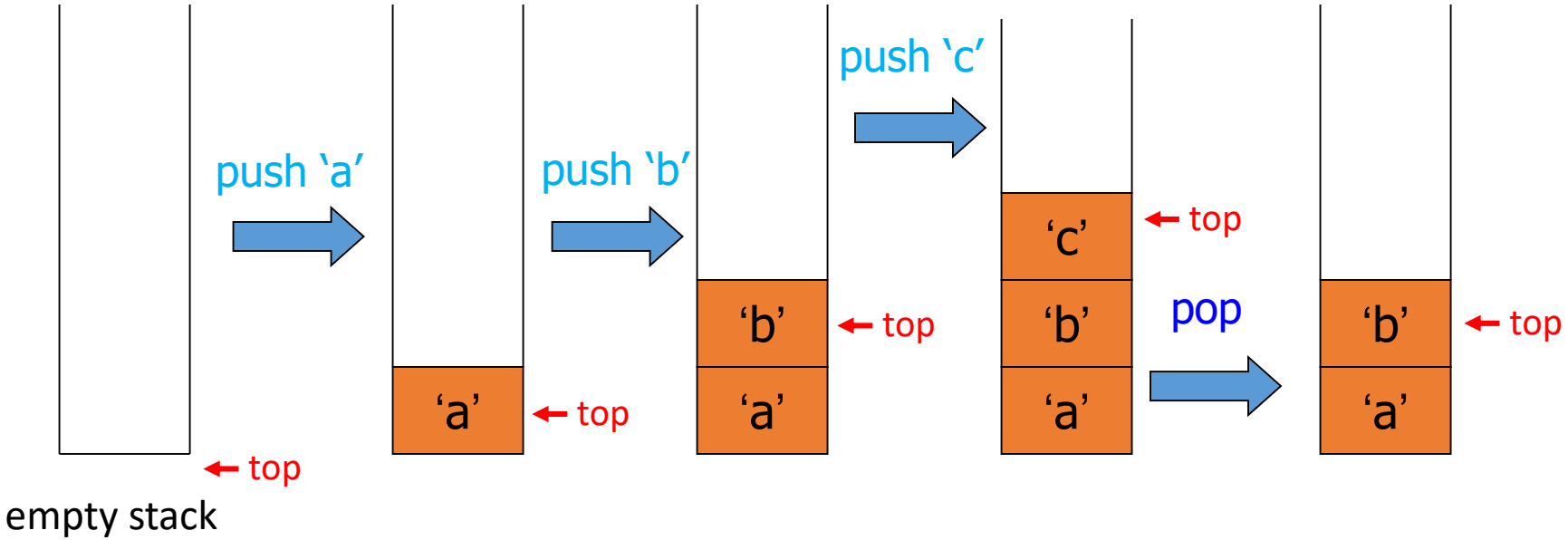
Top (*stack*)：　返回栈*stack*的栈顶结点，栈顶结点不变。

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*。

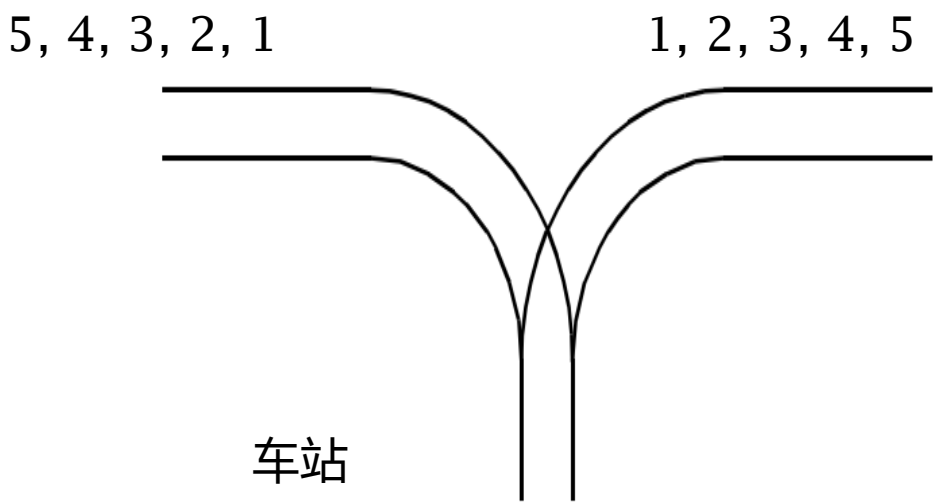
}

栈的示例



示例：火车进出栈问题

- 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?

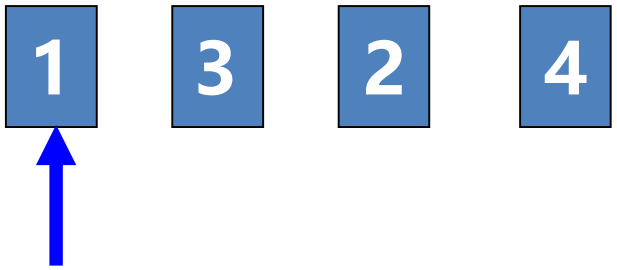


思考

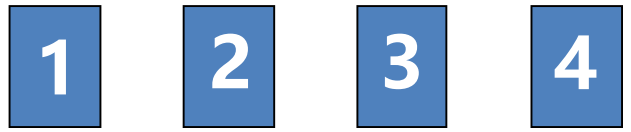
- 若入栈顺序为1,2,3,4的话，则出栈的顺序可以有哪些?
  - 1234
  - 1243
  - 1324
  - 1342
  - 1423
  - 1432
  - 2134
  - 2143
  - ……

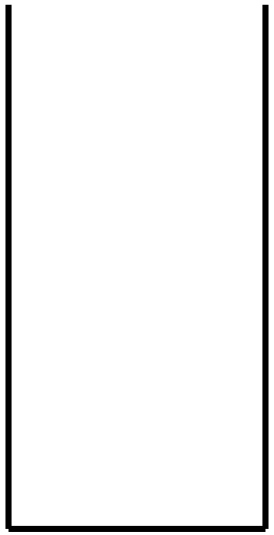
示例：火车进出栈问题

出栈顺序



入栈顺序





- **入栈**：栈为空或者栈顶元素与出栈元素不同
- **出栈**：栈顶元素与出栈元素一致

思考

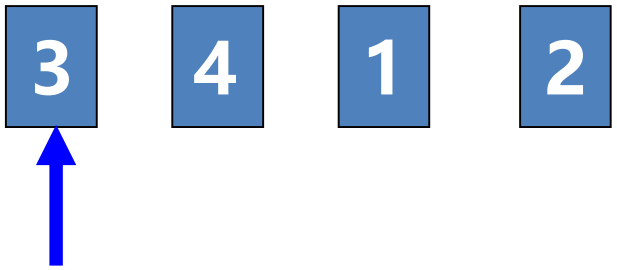
- 若入栈顺序为1,2,3,4的话，则出栈的顺序可以有哪些?
  - 1234
  - 1243
  - 1324
  - 1342
  - 1423
  - 1432
  - 2134
  - 2143
  - ……

- 存在无效的出栈顺序，如：3412

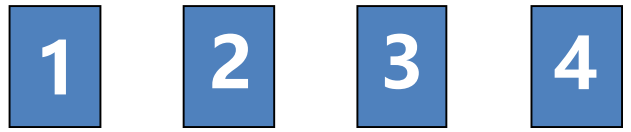
示例：火车进出栈问题

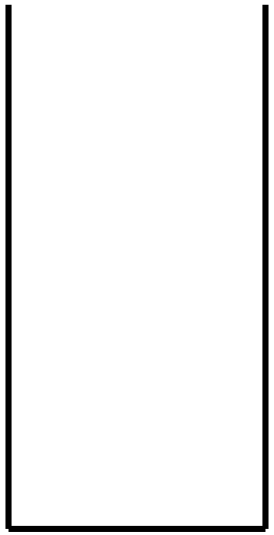
- 利用 合法的重构 发现 冲突

出栈顺序



入栈顺序





栈顶元素不同且
无可入栈元素

多选题 1分

设置

出栈顺序：1, 2, 3, 4，则下面哪些序列是无效的出栈顺序：

A 4321

B 1423

C 3124

D 2143

提交

思考

- **经典面试题**：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?

假设入栈序列为：$1, 2, \ldots, K, \ldots, N$

（1）所有数值均可成为最后出栈的元素

（2）**问**：若$K(1 \le K \le N)$最后出栈，有多少种出栈序列?

- K最后出栈，表明K入栈前，$1, 2, \ldots, K-1$已入栈并全部出栈
- K最后出栈，表明K入栈后至出栈前，$K+1$，$K+2$，$\ldots, N$ 已入栈并全部出栈

独立过程
单独计数

（3）设f(n)表示长度n序列的出栈序列数目，则K最后出栈情况下的出栈序列数:

$$f(K-1) * f(N-K)$$

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?

假设入栈序列为：$1, 2, \ldots, K, \ldots, N$

- 出栈序列总数：

$$f(N) = \sum_{K=1}^{N} f(K-1) * f(N-K)\ , \qquad f(0) = 1$$

$$f(N) = \frac{1}{N+1} C_{2N}^{N}$$

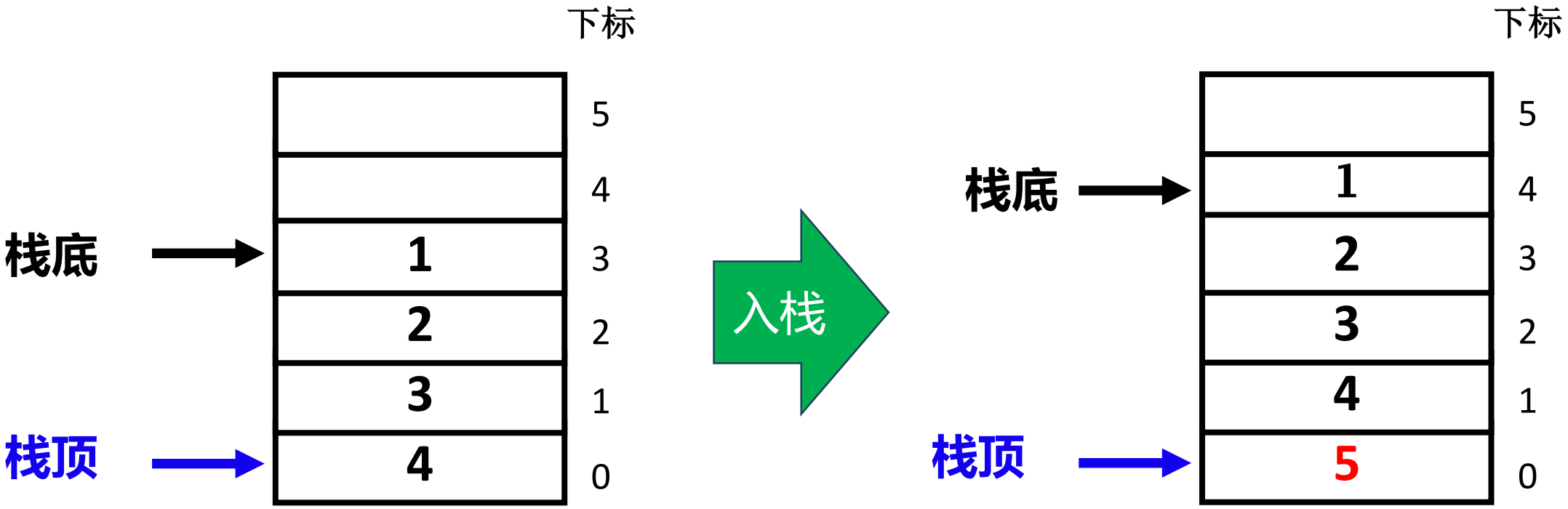
栈的实现方式

- 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

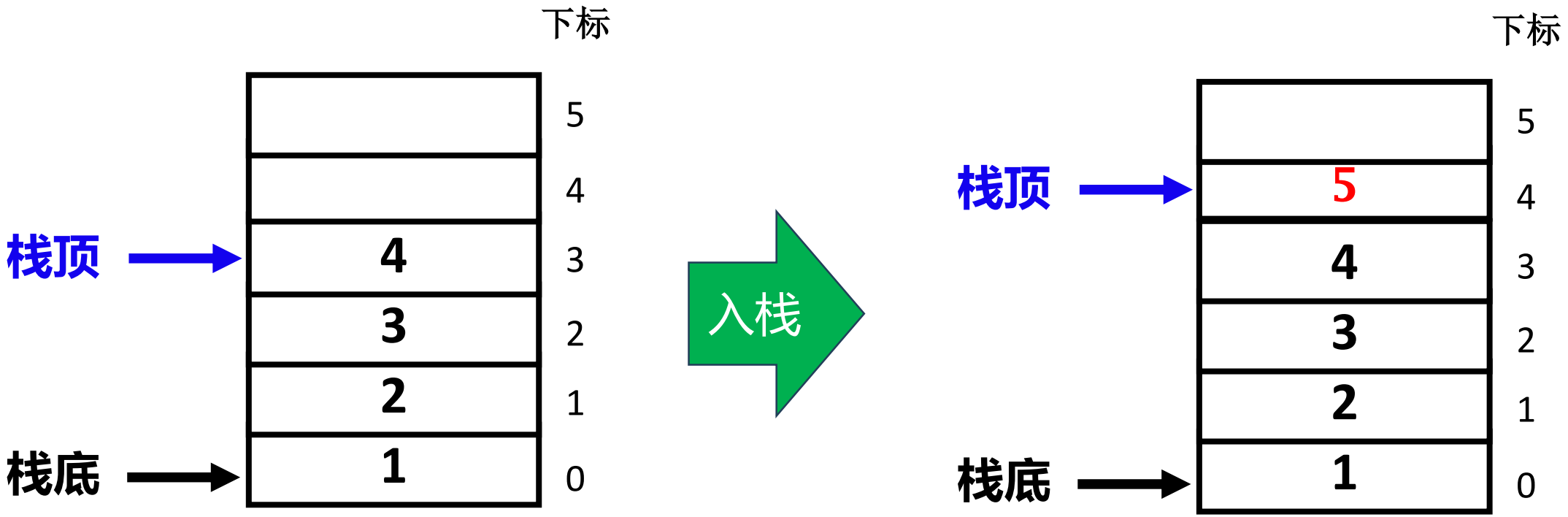
栈的实现方式

- 顺序栈的栈顶设置方式1：固定在顺序表前端



栈的实现方式

- 顺序栈的栈顶设置方式2: 指向顺序表最后一个元素



单选题 1分

设置

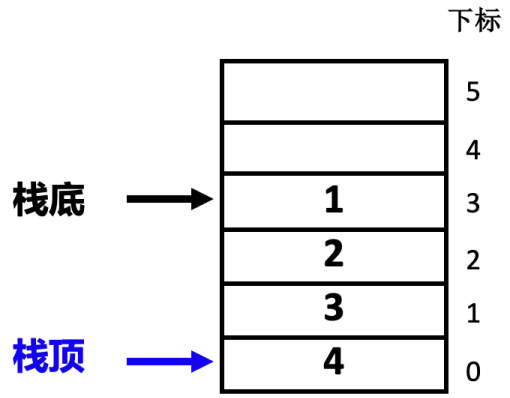
两种方式的入栈（出栈）时间分别是：(n表示栈中元素数量)

A O(1) 和 O(1)

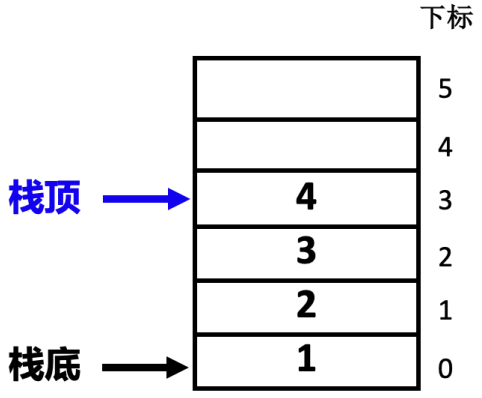
B O(n) 和 O(n)

C O(n) 和 O(1)

D O(1) 和 O(n)



方式1



方式2

提交

顺序栈

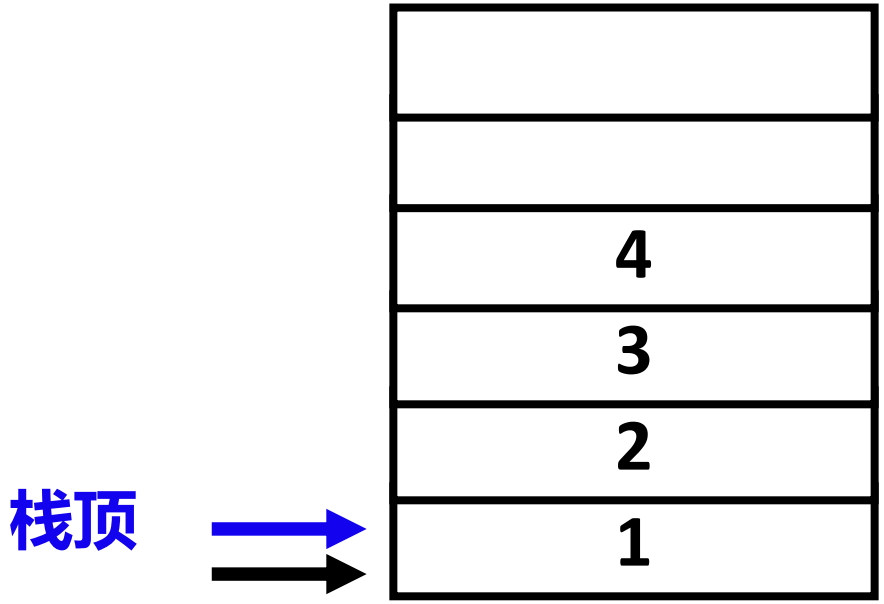
代码：顺序实现的栈的初始化 InitStack (*stack*, kSize)

输入：栈*stack*和正整数kSize
输出：一个大小为kSize的顺序栈

1. *stack.capacity* ← kSize
2. *stack.top* ← -1
3. *stack.data* ← **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 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 IsFull(*stack*)=**true then**

2. | 栈满，退出

3. else

4. | *stack.top* ← *stack.top* + 1

5. | *stack.data* [*stack.top*] ← *x*

6. end

出栈操作

算法：顺序栈的取顶操作 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.data[stack.top]
5.  end
```

读栈操作

算法：顺序栈的出栈操作 Pop (*stack*)

输入：栈*stack*

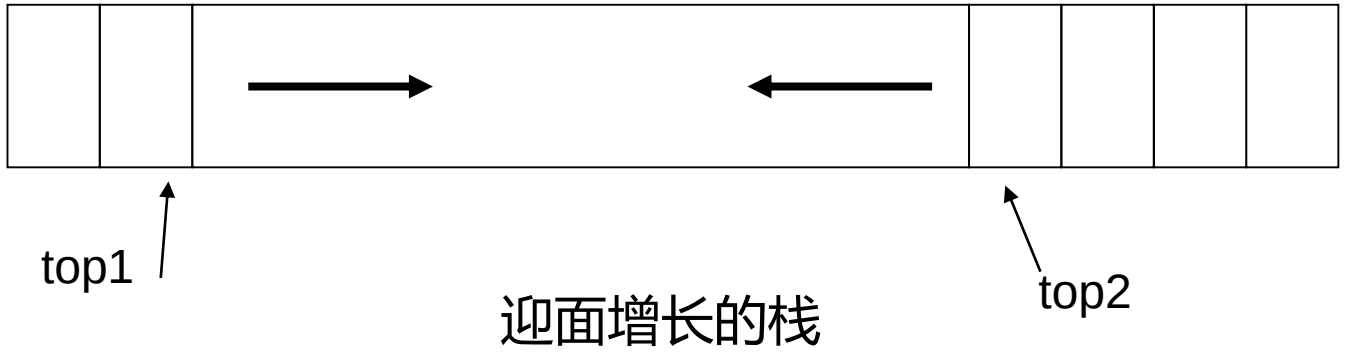
输出：删除栈顶元素后的顺序栈；若栈空，则退出

1. **if** IsEmpty(*stack*)=**true then**
2. | 栈空，退出
3. **else**
4. | *stack.top* ← *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?

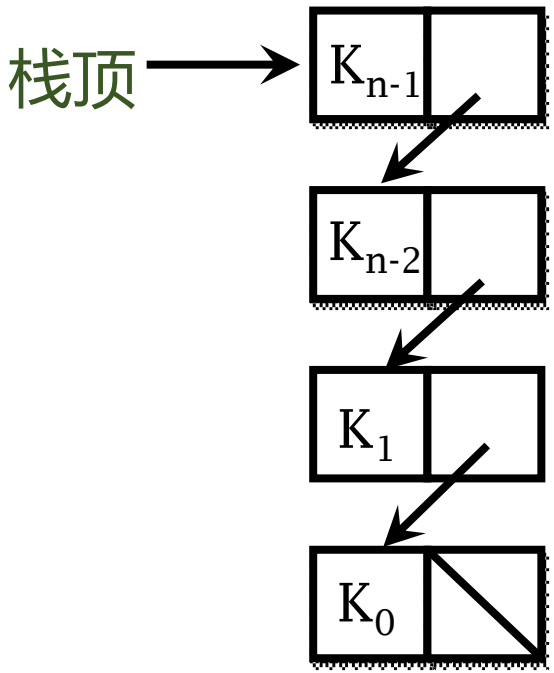
底部相连的栈



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* ← 0
2. *stack.top* ← NIL

压栈操作

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

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 ← stack.top
5. | stack.top ← stack.top.next
6. | delete temp
7. | stack.size ← 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

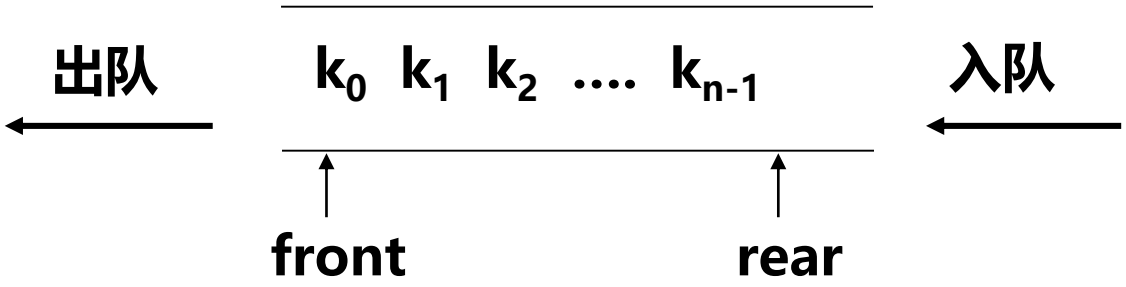
- 空间效率
  - ✓ 顺序栈结构紧凑，但须事先确定长度
  - ✓ 链式栈长度可变，增加结构性开销

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（FirstInFirstOut）**
  - 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*∈ElemSet，*i*=1,2,…*n*, *n* > 0 } 或*Φ*，即空表；ElemSet为元素集合。

数据关系：

{<*ai*, *ai*+1>|*ai*, *ai*+1∈ElemSet，*i*=0,1,…*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

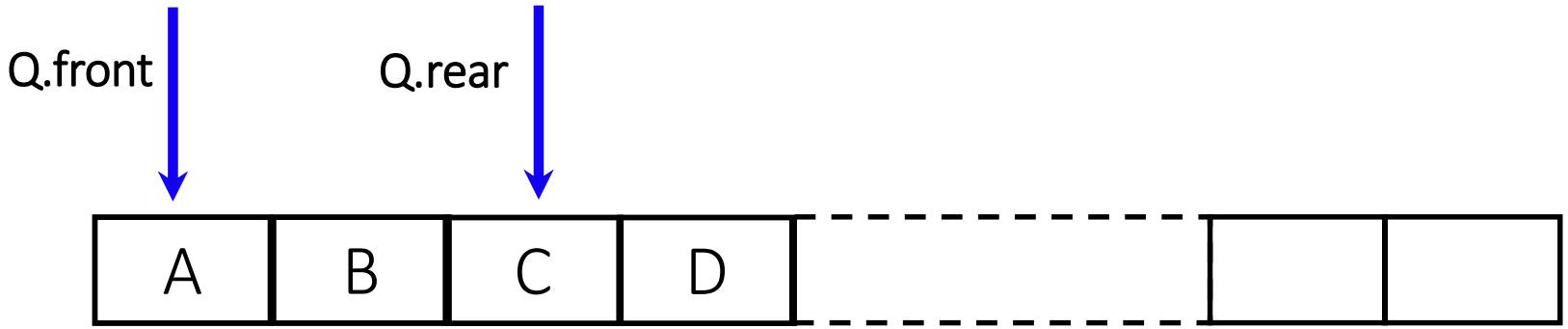
- 上溢
  - 当队列满时，入队操作所出现的现象
- 下溢
  - 当队列空时，出队操作所出现的现象
- 假溢出
  - 顺序队列可能出现的一种现象：当队尾指针达到最大值（rear = mSize）时，再做入队运算产生溢出，但此时队列前端可能尚有空闲位置

顺序队列

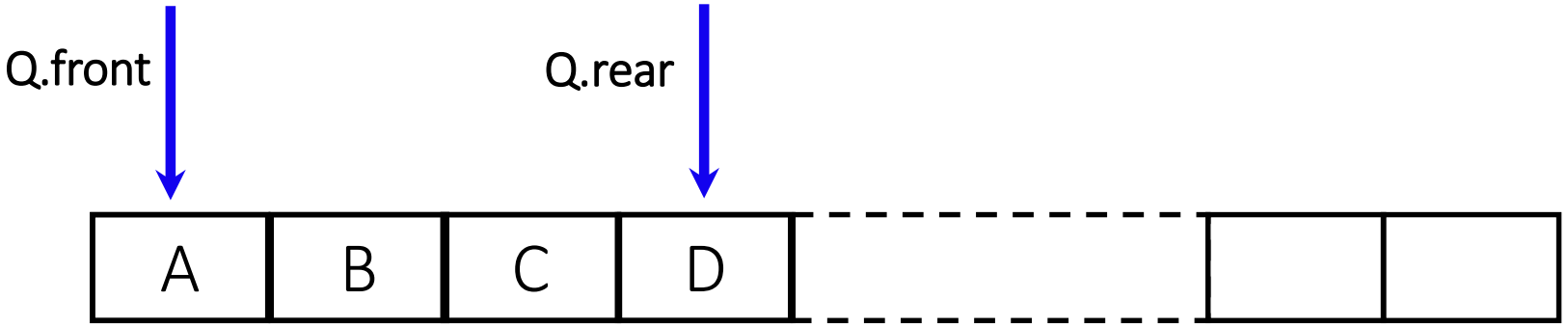
- 使用顺序表来实现队列
  - 用数组存储队列元素，用两个变量分别指向队列的队头（前端）和队尾（尾端）
    - Q.front
    - Q.rear

顺序队列的维护

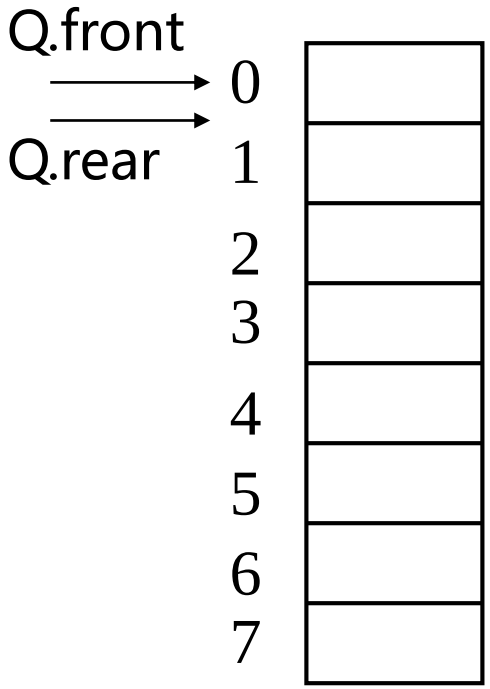
- 尾指针Q.rear 的处理
  - ✓ 实指

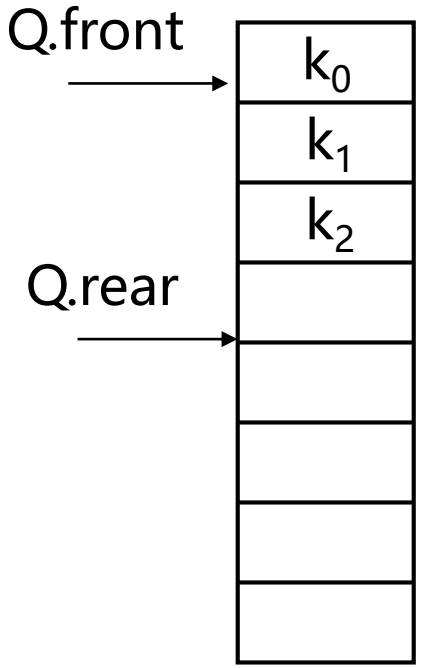


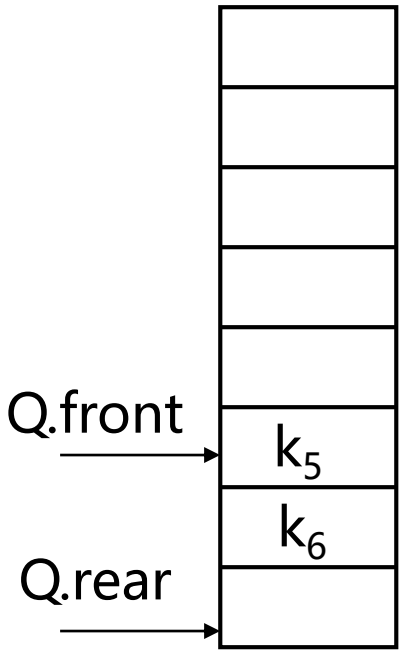
  - ✓ 虚指



队列示意：普通





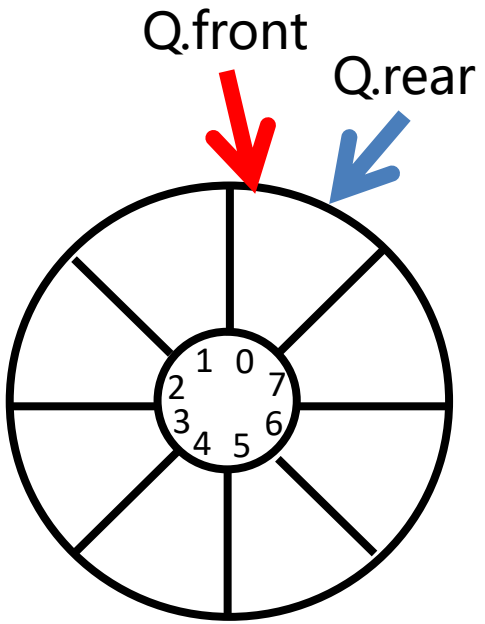


队列空

再进队一个元素如何?

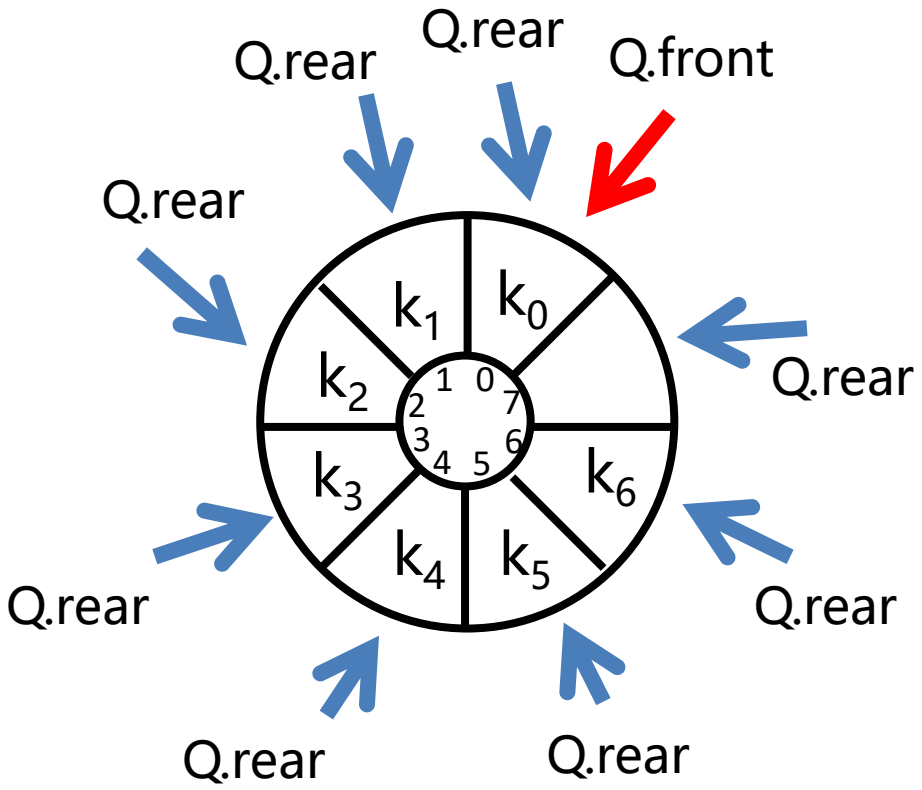
队列示意：环形

- 虚指针



队列空：

Q.rear == Q.front



EnQueue(k_0)

EnQueue(k_1)

EnQueue(k_2)

EnQueue(k_3)

EnQueue(k_4)

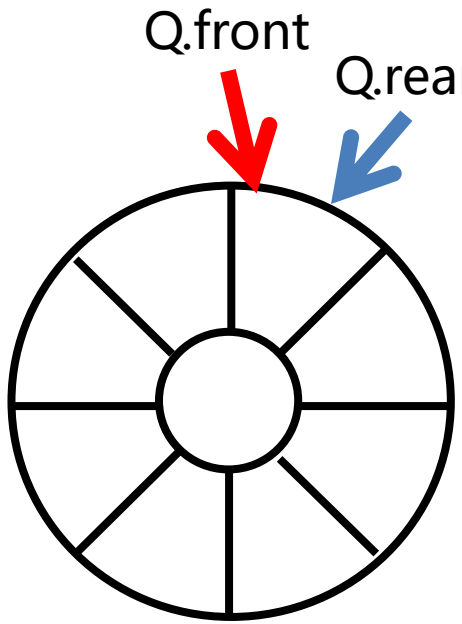
EnQueue(k_5)

EnQueue(k_6)

队列满：

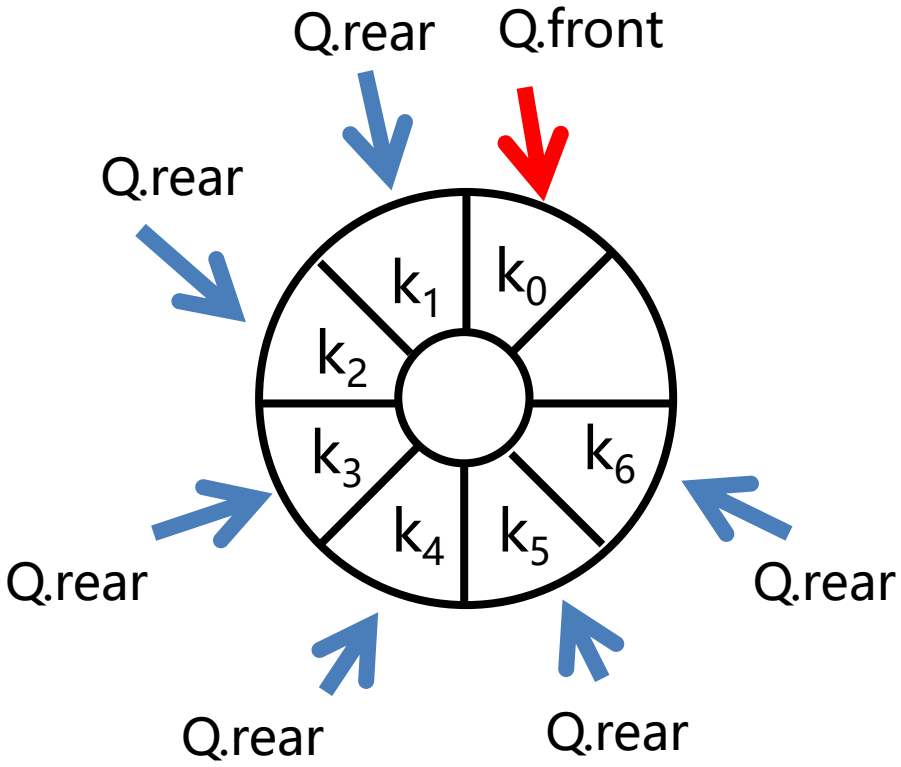
(Q.rear+1) mod M == Q.front （M表示顺序表长度）

队列示意：环形



队列空：

Q.rear == Q.front

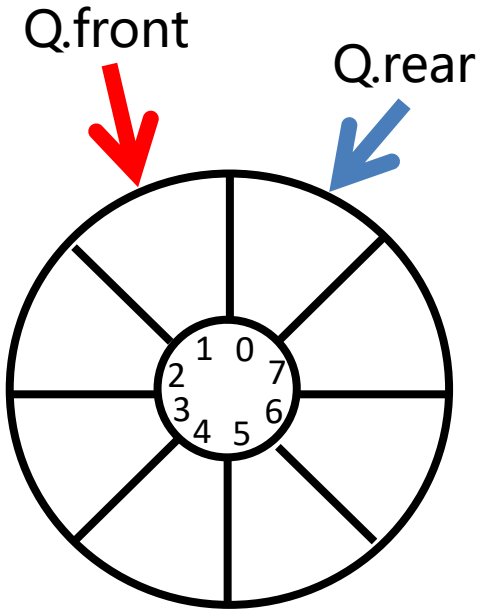


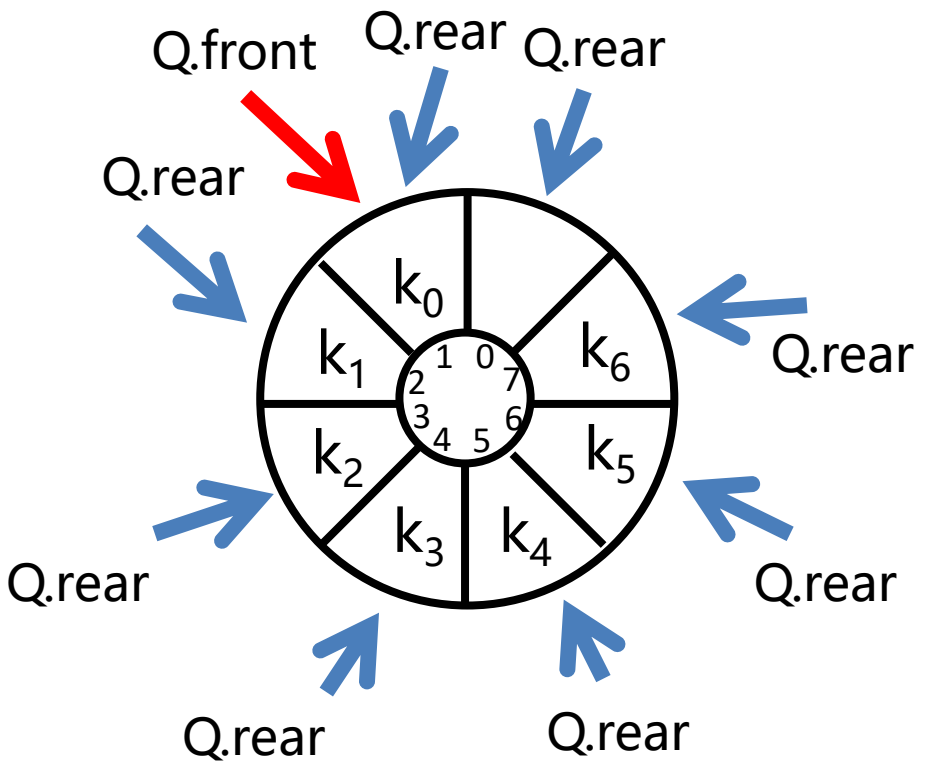
队列满：

(Q.rear+1) mod M == Q.front

队列示意：环形

- 实指针





EnQueue(k_0)

EnQueue(k_1)

EnQueue(k_2)

EnQueue(k_3)

EnQueue(k_4)

EnQueue(k_5)

EnQueue(k_6)

队列空：

(Q.rear+1) mod M == Q.front

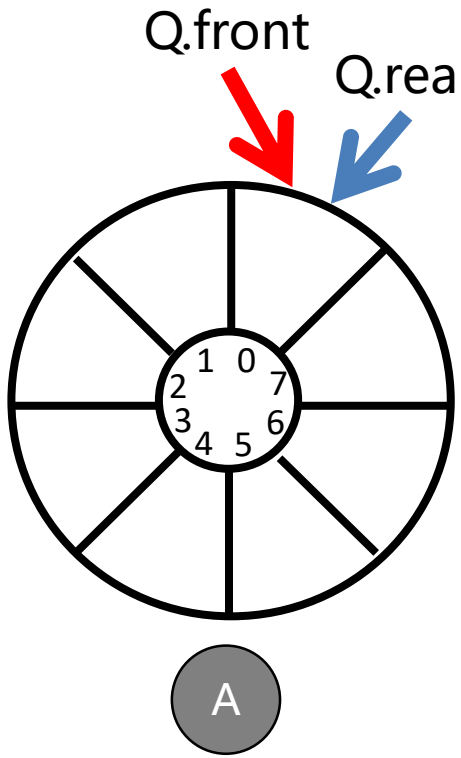
队列满：

(Q.rear+2) mod M == Q.front （M表示顺序表长度）

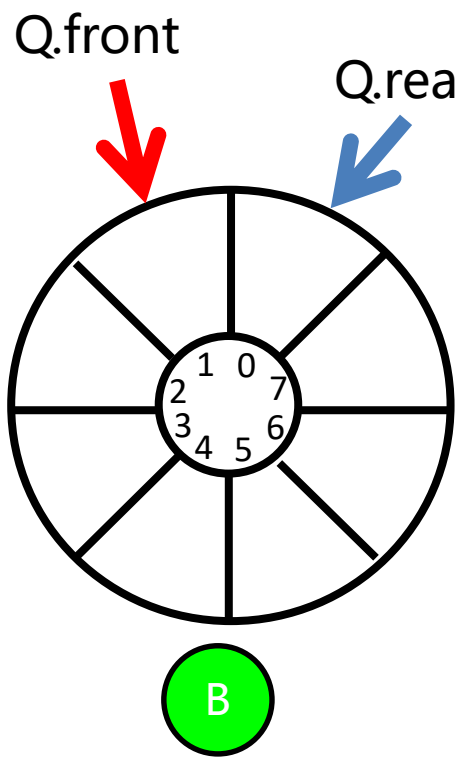
单选题 1分

设置

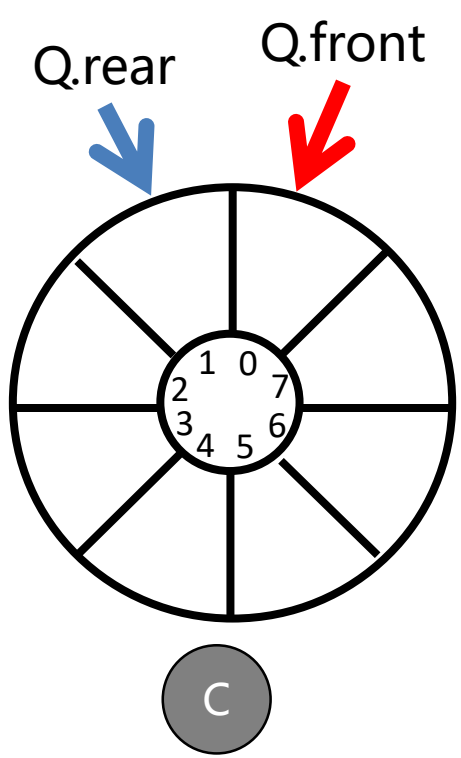
如果用**实指针**，空栈如何表示?



A



B



C

提交

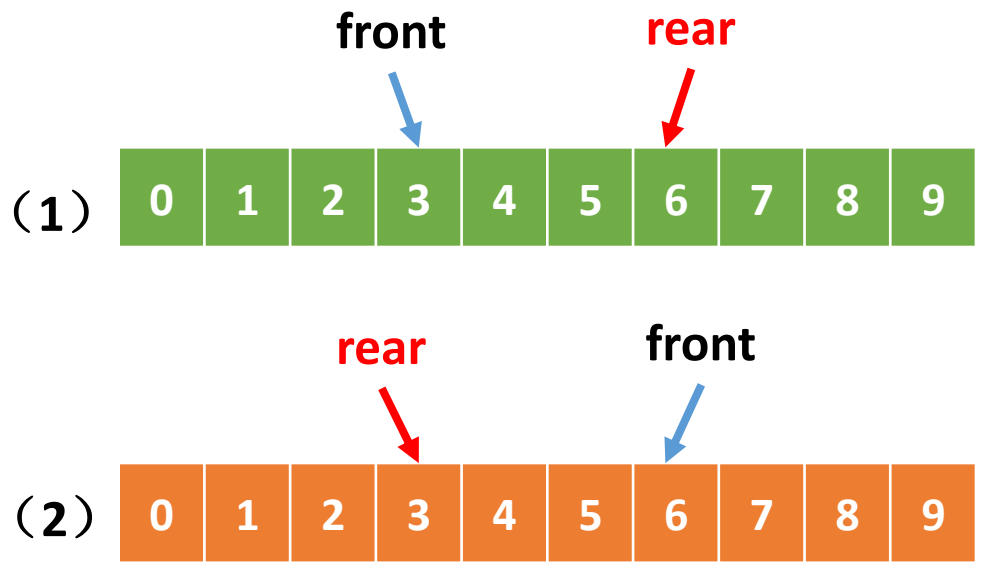
思考

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?

单选题 1分

设置

循环队列使用实指针，两个队列中各自存放有多少个元素？



A 3和7

B 3和8

C 4和6

D 4和8

提交

单选题 1分

设置

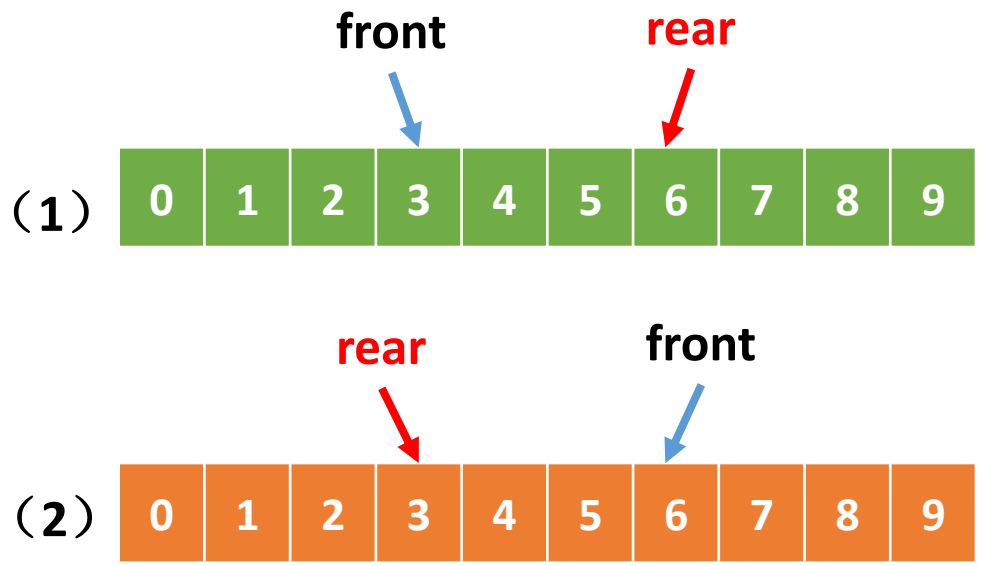
循环队列使用虚指针，两个队列中各自存放有多少个元素？

A 3和7

B 3和8

C 4和6

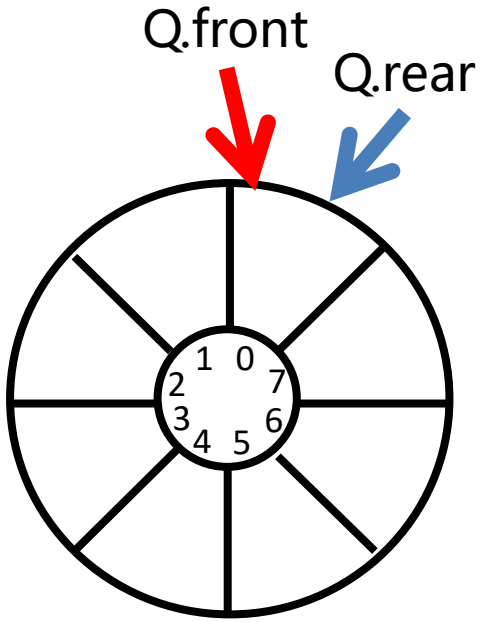
D 4和8



提交

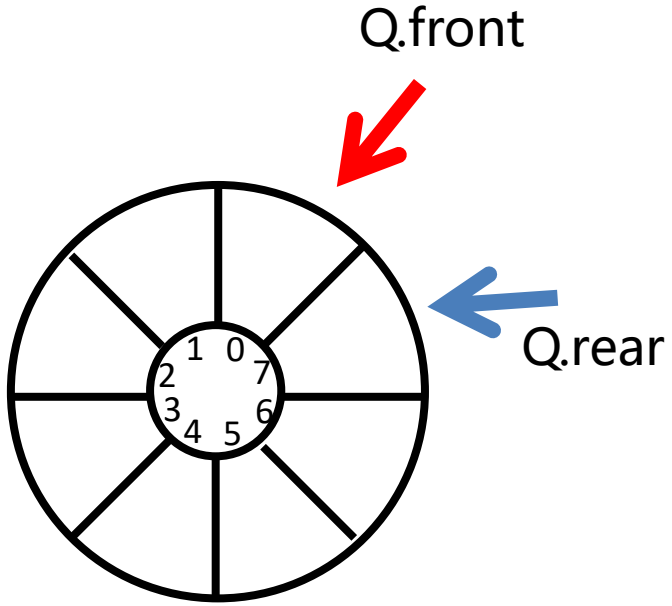
队列示意：环形

- 虚指针



- **队列空**：

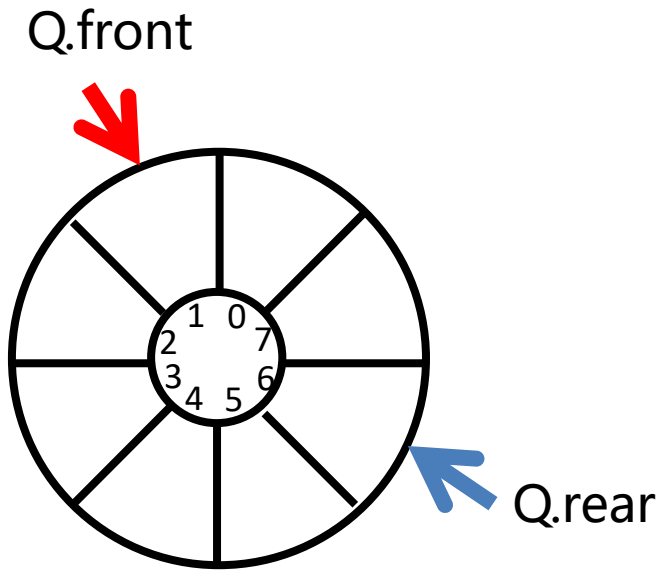
Q.rear == Q.front



- **队列满**：

(Q.rear+1) mod M == Q.front

（M表示顺序表长度）

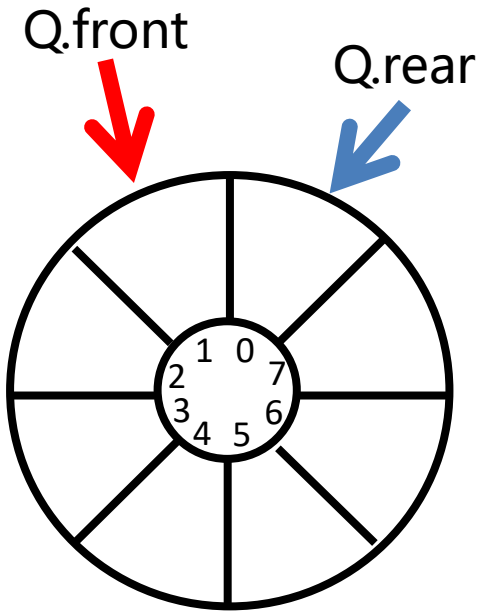


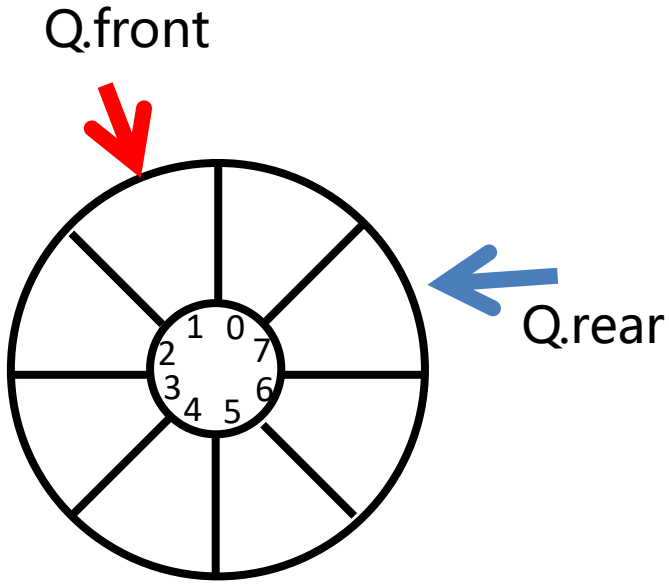
- **队列中的元素数量**：

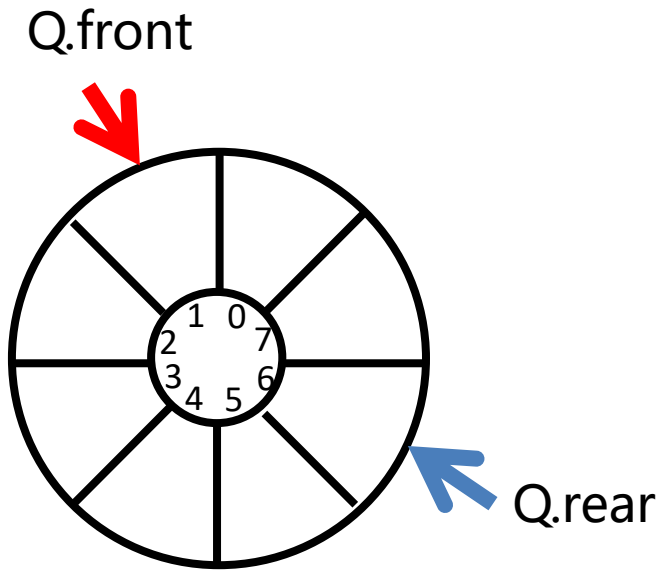
(M+Q.rear-Q.front) mod M

队列示意：环形

- 实指针







- **队列空**：
(Q.rear+1) mod M == Q.front

- **队列满**：
(Q.rear+2) mod M == Q.front
（M表示顺序表长度）

- **队列中的元素数量**：
(M+Q.rear-Q.front+1) mod M

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
5.
```

顺序队列的实现

算法3-7：顺序队列的入队操作 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 IsFull(queue)=true then
2. | 队列满，退出
3. else
4. | queue.data[queue.rear] ← x
5. | queue.rear ← (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 GetFront(*queue*)

输入：队列*queue*
输出：队列的头元素；若队列空，则输出NIL

```
1. if IsEmpty(queue)=true then
2. | return NIL
3. else
4. | return queue.data[queue.front]
5. end
```

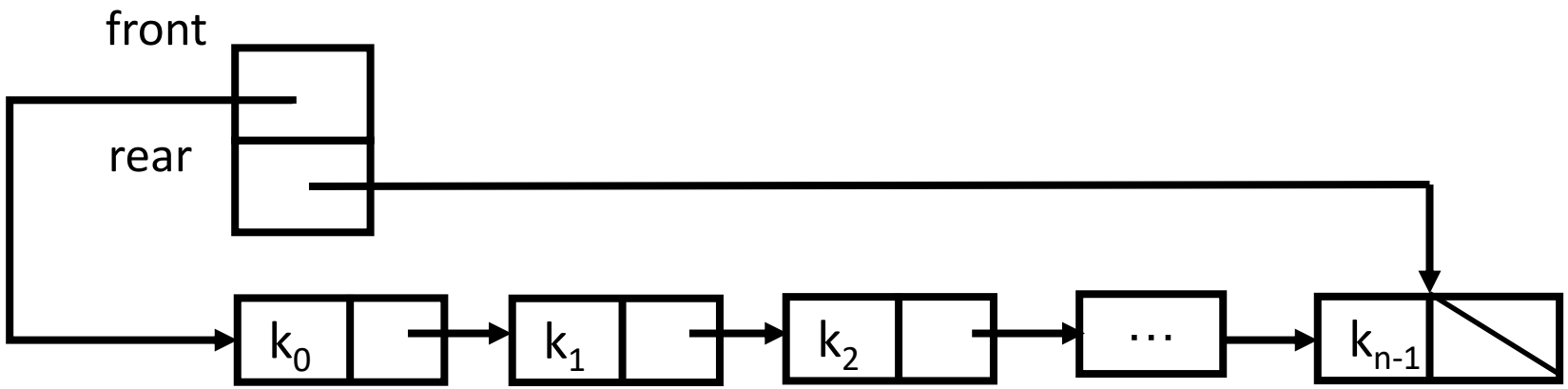
算法：顺序队列的出队操作 DeQueue(*queue*)

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 IsEmpty(queue)=true then
2. | 队列空，退出
3. else
4. | queue.front ← (queue.front +1) % queue.capacity
5. end
```

链式队列

- 单链表队列
- 链接指针的方向是从队队头到队尾



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 ← 0
2. queue.front ← NIL
3. queue.rear ← NIL

链式队列的入队

算法：链式队列的入队操作 EnQueue(*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.  new_node ← new QueueNode
2.  new_node.data ← x
3.  new_node.next ← NIL
4.  if IsEmpty(queue)=true then //特殊处理插入空队列的情况
5.  | queue.rear ← new_node
6.  | queue.front ← new_node
7.  else
8.  | queue.rear.next ← new_node
9.  | queue.rear ← queue.rear.next
10. end
11. queue.size ← 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 GetFront(*queue*)

输入：队列*queue*
输出：队首元素；若队列空，则输出NIL

```
1. if IsEmpty(queue)=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 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
```

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

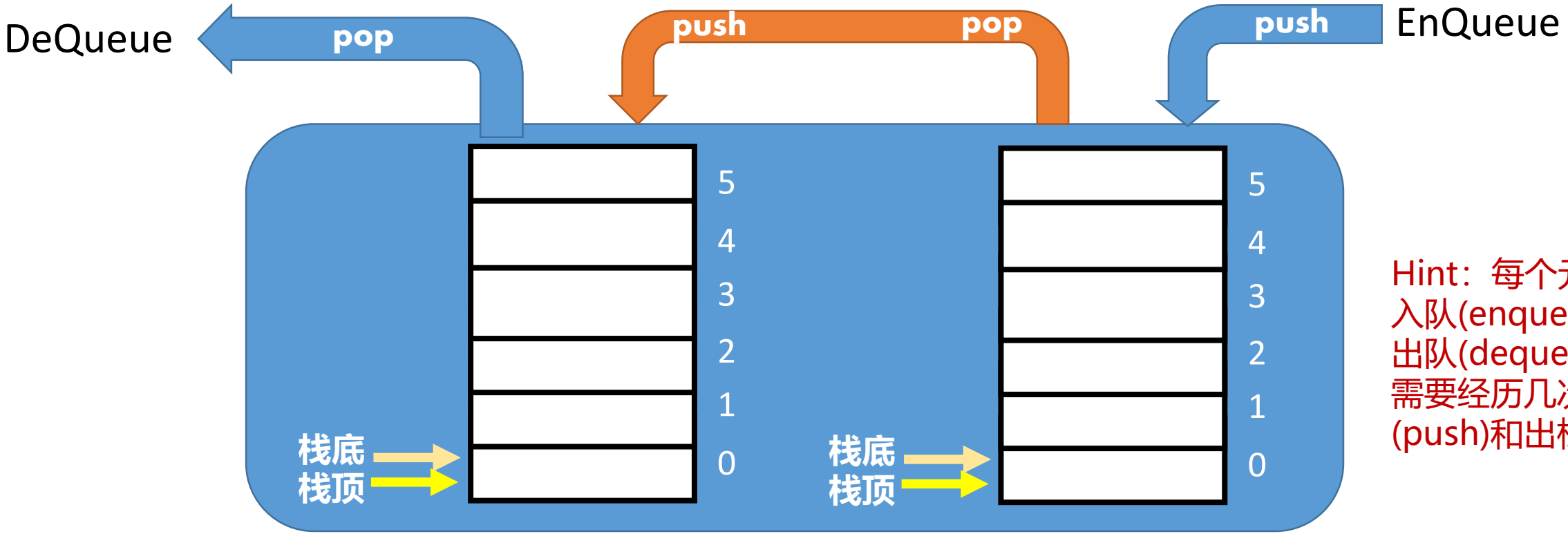
- 是否可以用栈来模拟队列？若可以的话，需要几个栈，如何来模拟？

- 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。

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？

  - 元素什么时候从左边栈移到右边栈？如何移？
  - 队列的容量是？
  - 模拟的时间复杂度？



Hint：每个元素从入队(enqueue)到出队(dequeue)，需要经历几次入栈(push)和出栈(pop)？

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

括号语句的判定

- **括号语句**：由括号字符: (,), [,], {, }按如下规则递归生成的字符串
  (1) 空字符串是括号语句
  (2) 如果A是括号语句，则(A), [A]以及{A}是括号语句
  (3) 如果A和B是括号语句，则AB是括号语句

  - ✓ 括号语句：(), [], {}, ([]), ()[]{}, ([{}])
  - ✓ 非括号语句：((), ([)], {{]], ((([[))])]

- **括号语句的判定问题**：给出一个由括号字符组成的字符串，判断其是否是括号语句

括号语句的判定

- 设$s_1 s_2 \ldots s_i s_{i+1} \ldots s_{j-1} s_j \ldots s_n$是括号语句，满足以下性质

  - s_1是左括号，s_n是右括号
  - 如果s_i $(1 \le i < n)$是左括号，存在唯一的$j \in [i+1, n]$，s_j 是与s_i 匹配的右括号
  - 如果s_j $(1 < j \le n)$是右括号，存在唯一的$i \in [1, j-1]$，s_i 是与s_j 匹配的左括号
  - 如果s_i与s_j相互匹配，则子串$s_{i+1} \ldots s_{j-1}$是括号语句

用**栈**实现括号语句判定

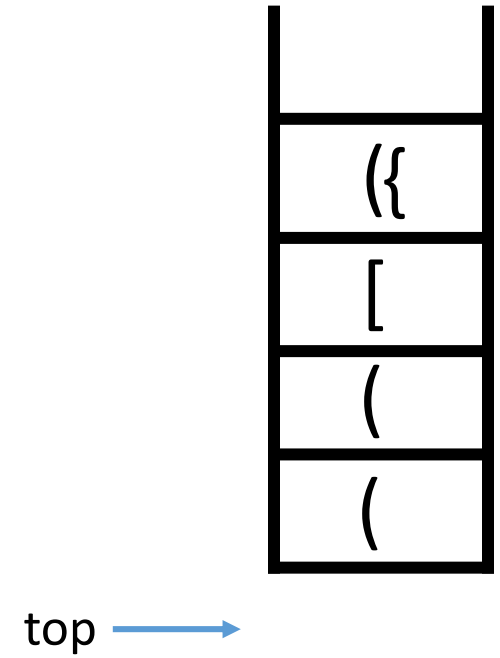
括号语句的判定

```
1.  InitStack(stack)                    //初始化栈，存放左括号字符
2.  n ← Length(str)                     //字符串str的长度，字符串由括号字符组成
3.  k ← 0
4.  valid ← true                        //判定结果的初始值
5.  while valid 且 k < n do
6.  |   if str[k] = `(` 或 str[k] = `[` 或 str[k] = `{` then  //第k个字符是左括号
7.  |   |   Push(stack, str[k])                                //入栈
8.  |   else                  //str[k]是右括号
9.  |   |   if IsEmpty(stack) = false 且 isMatch(Top(stack), str[k]) = true then
10. |   |   |   Pop(stack)          //与栈顶的左括号类型一致，弹出左括号
11. |   |   else                //栈顶括号与str[k]不匹配
12. |   |   |   valid ← false
13. |   |   end
14. |   end
15. |   k ← k + 1
16. end
17. if valid = true 且 IsEmpty(stack) = false then  //还有未匹配的左括号
18. |  valid ← false
19. end
20. DestroyStack(stack)
21. return valid
```

- 时间复杂度：O(n)
- 空间复杂度：O(n)

括号语句的判定示例

(([( ) { }]))



单选题 1分

设置

使用栈判定字符串 (()[(()())](())) 是否括号语句，栈的最小容量是

A 2

B 3

C 4

D 5

提交

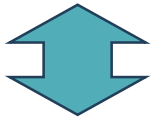
括号语句的判定

- **思考**：如果字符串仅含小括号（和），括号语句的判定有无高效算法？

  - ✓ 括号语句：(), (()), ()()(), (()(()))()(())
  - ✓ 非括号语句：(,)(), (((), (())())

- 仅含（和）的括号语句基本性质：

  - 字符串的所有前缀中，左括号数量不少于右括号数量
  - 整个字符串中的左括号与右括号数量相等



充分必要条件（思考）

字符串是括号语句

思考：如何检测左括号与右括号的数量差？

括号语句的判定

```
1. n ← Length(str)              //字符串str的长度，字符串由小括号字符组成
2. pref_sum ← 0                 //前缀和的初始值
3. for k ← 0 to n-1 do
4.   | if str[k] = `(` then  //第k个字符是左括号
5.   |   | pref_sum ← pref_sum + 1          //前缀和加1
6.   | else                          //str[k]是右括号
7.   |   | pref_sum ← pref_sum - 1          //前缀和减1
8.   | end
9.   | if pref_sum < 0 then   //前缀和小于0，即右括号比左括号多
10.  |   | return false          //非括号语句
11.  | end
12. end
13. if pref_sum > 0 then  //str中左括号比右括号多
14. | return false
15. end
16. return true
```

- 时间复杂度：O(n)
- 空间复杂度：O(1)

表达式计算

- 表达式定义
  - ✓ 基本符号集：{0，1，…，9，+，-，*，/，（，）}
  - ✓ 语法成分集：{<表达式>，<项>，<因子>，<常数>，<数字>}
  - ✓ 语法公式集

- 中缀表达式

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

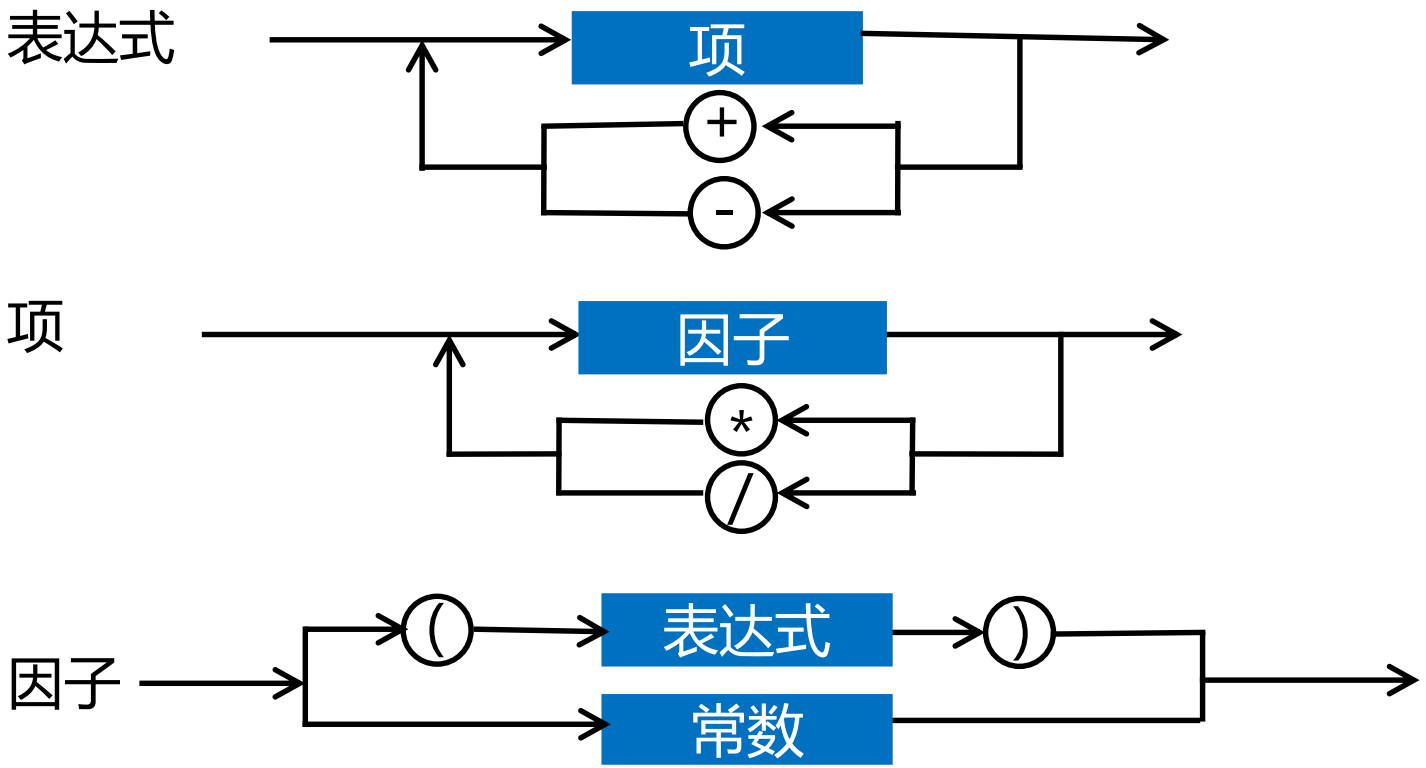
- 后缀表达式

  23 34 45 * 5 6 + 7 + / +

中缀表达式的语法公式

<表达式> ::= <项> + <项>
| <项> - <项>
| <项>
<项> ::= <因子> * <因子>
| <因子> / <因子>
| <因子>
<因子> ::= <常数>
| (<表达式>)
<常数> ::= <数字>
| <数字> <常数>
<数字> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

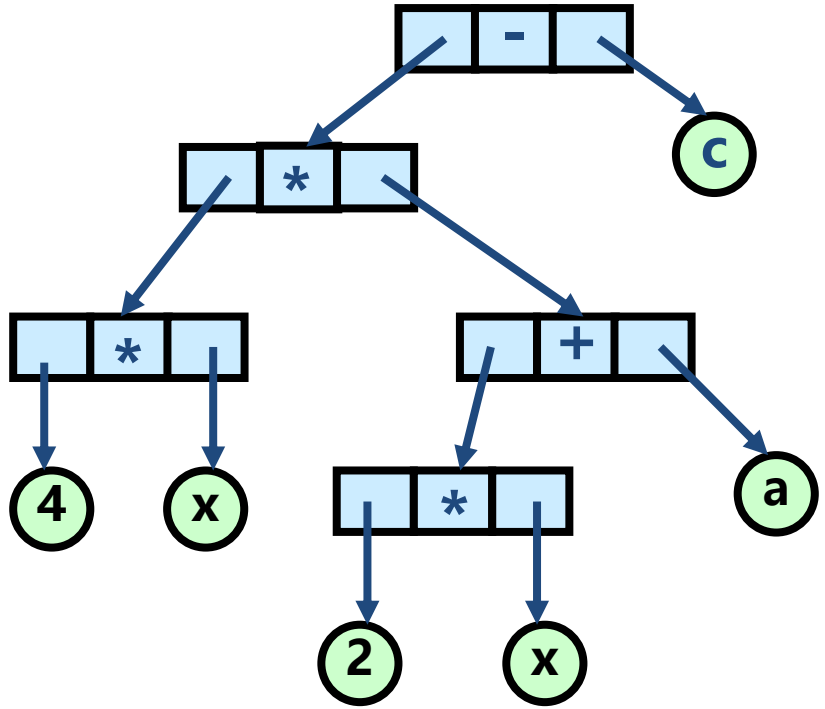
中缀表达式的递归图示



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

<表达式> ::= <项> <项> +
| <项> <项> -
| <项>

<项> ::= <因子> <因子> *
| <因子> <因子> /
| <因子>

<因子> ::= <常数>

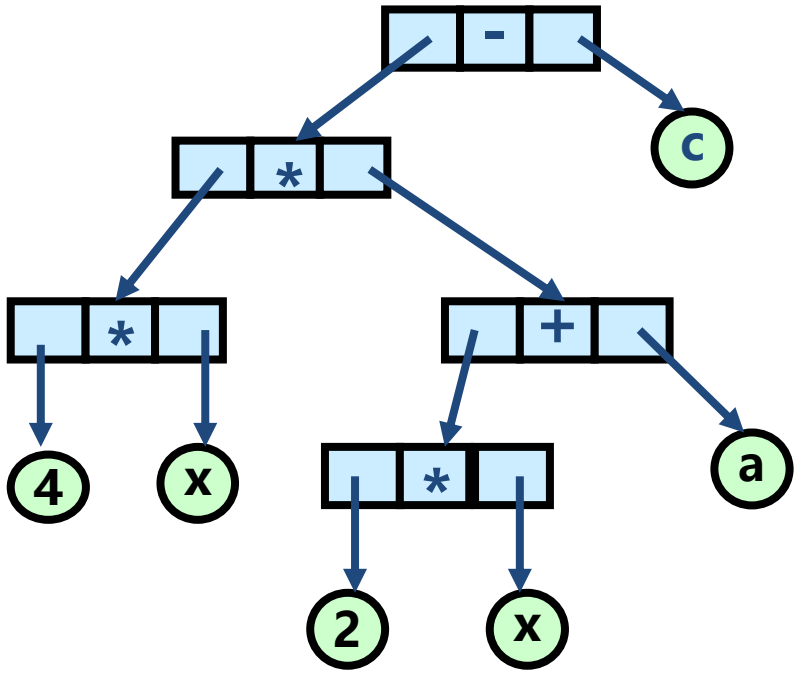
<常数> ::= <数字>
| <数字> <常数>

<数字> ::= 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 + / + =?

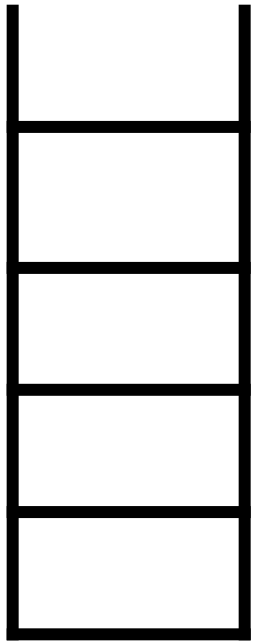
计算特点?

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



	1

计算 **结果**

后缀表达式求值算法

算法3-13：后缀表达式求值 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* ← GetToken(*expr*) // 从表达式中取出一个元素
2. while *token* ≠ 表达式结尾 **do**
3. | **if** IsOperand(*token*) **then** // 如果该元素是操作数
4. | | Push(*stack*, *token*) // 则入栈
5. | **else** // 如果该元素是操作符
6. | | *operand*2 ← Top(*stack*)
7. | | Pop(*stack*)
8. | | *operand*1 ← Top(*stack*)
9. | | Pop(*stack*)
10. | | *result* ← Calculate(*operand*1, *token*, *operand*2) //计算*operand*1 *token* *operand*2的值
11. | | Push(*stack*, *result*) // 计算结果入栈
12. | **end**
13. | *token* ← GetToken(*expr*) //从表达式中取下一个元素
14. end
15. *result* ← Top(*stack*)
16. Pop(*stack*)
17. if IsEmpty(*stack*) = **false then** //表达式不规范
18. | *result* ← ErrorCode
19. end
20. 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 ?

中缀表达式的**异**：不同运算符的优先级有差异，括号的使用可改变运算符的优先次序!

运算符的计算次序与符号在表达式中出现的先后顺序可能无关（非线性）

用 同 化 异 ?

后缀表达式的**同**：所有运算符的优先级相同，没有括号!

运算符的计算次序与符号在表达式中出现的先后顺序一致 （线性）

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；

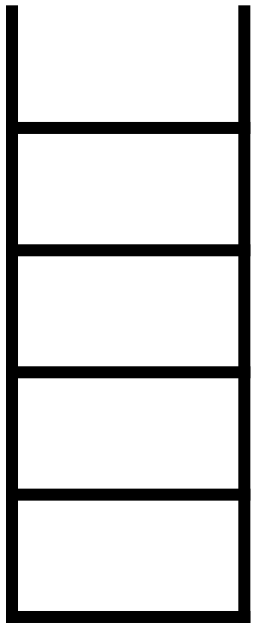
(5) 读完全部输入符号，若栈内仍有元素，将其全部依次弹出，置于后缀表达式序列的尾部。若弹出元素中有开括号出现，则说明括号不匹配，做异常处理，清栈退出

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

单选题 1分

设置

与中缀表达式 1+2/(3+4*5)等价的后缀表达式是

A 1 2 + 3 4 + 5 * /

B 1 2 3 / 4 5 * + + /

C 1 2 3 4 + 5 * / +

D 1 2 3 4 5 * + / +

提交

单选题 1分

设置

与后缀表达式 ab+c*ab+e/-等价的中缀表达式是

A a+b*c-a+b/e

B (a+b)*c/(a+b-e)

C (a+b)*c-(a+b)/e

D ((a+b)-(a+b)/e)*c

提交

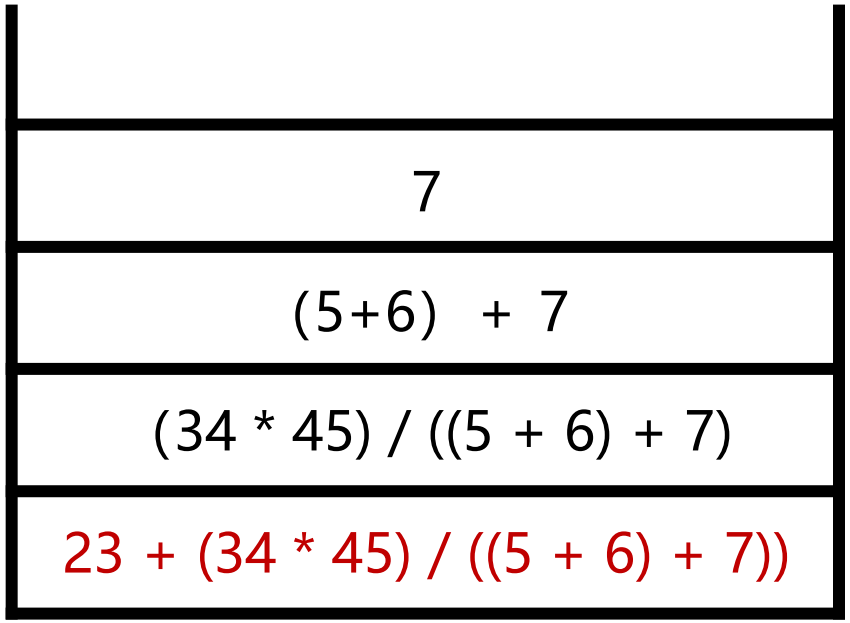
*后缀表达式 to 中缀表达式

待处理后缀表达式

23 34 45 * 5 6 + 7 + / +

栈状态的变化

存放项(term)的栈



思考：时间和空间复杂度?

生成的中缀表达式
(含冗余括号)

中缀表达式求值

- 中缀表达式 → 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：

$$\text{factiorial}(n)=\begin{cases}1,\text{if } n\le 1\\ n\times\text{factiorial}(n-1),\text{if } n>1\end{cases}$$

- 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?

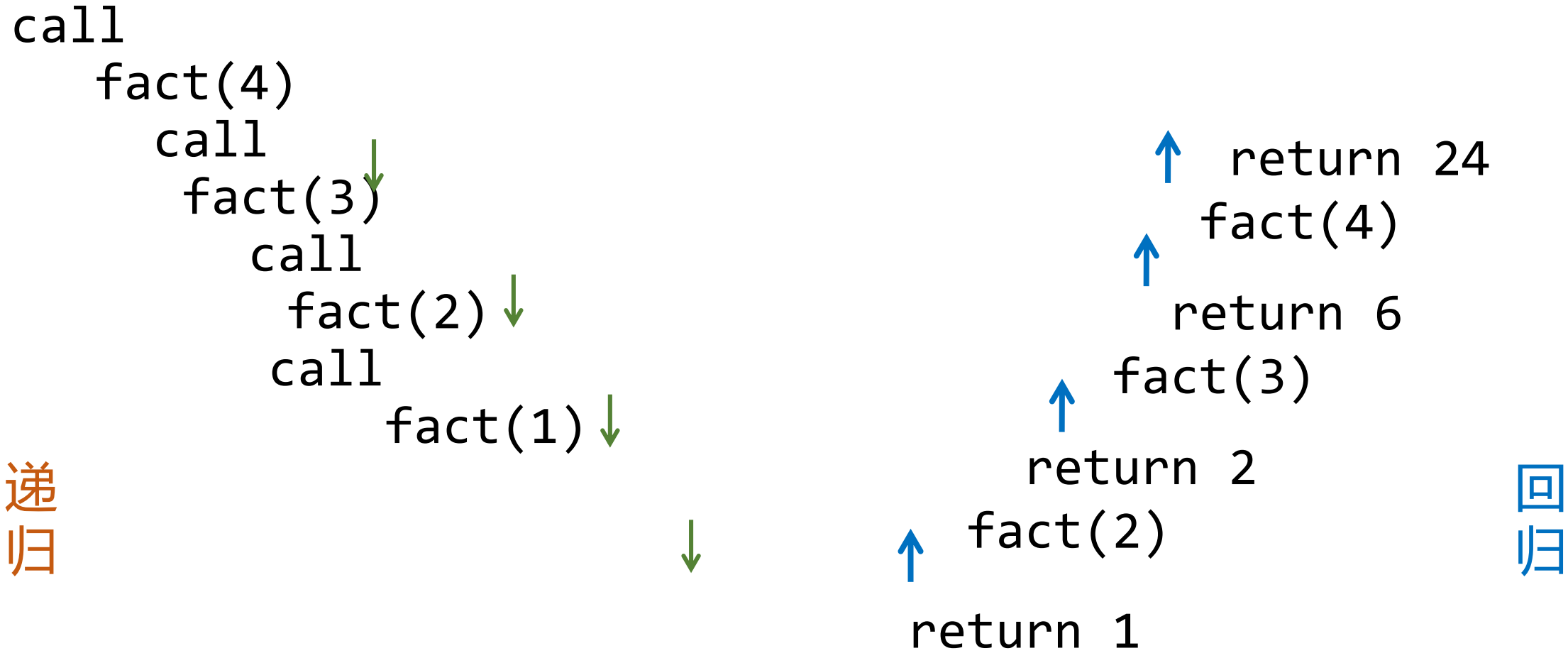
递归示例： 阶乘函数的程序实现

算法：阶乘的递归实现 Factorial(n)

输入：整数$n \geq 0$
输出：整数n的阶乘

```
1. if  n ≤ 0 then
2. |  return 1
3. else
4. |  return n×Factorial(n-1)
5. end
```

函数执行过程图解：4!

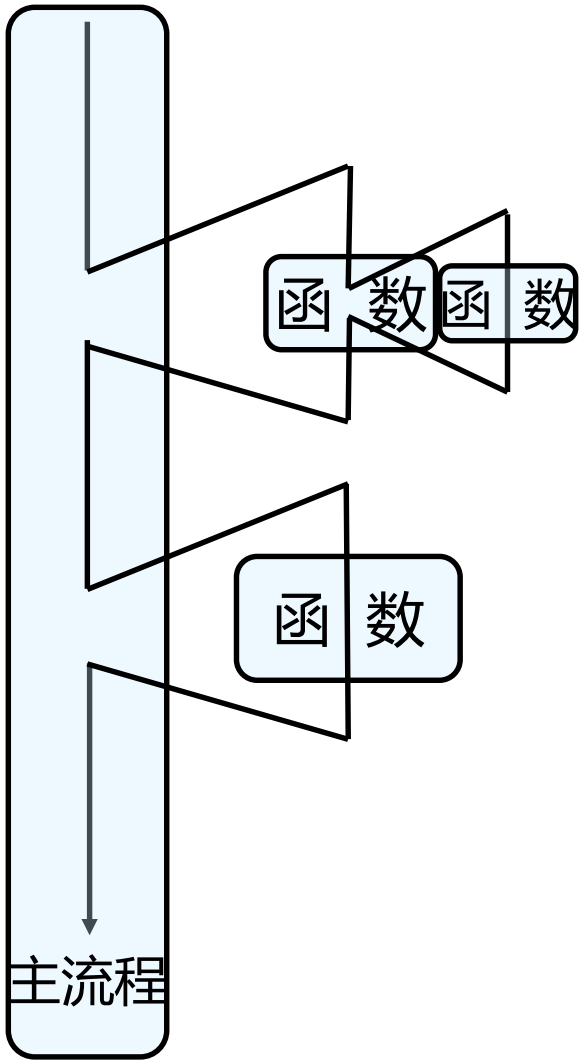

call
fact(4)
call
fact(3)
call
fact(2)
call
fact(1)
递归
return 24
fact(4)
return 6
fact(3)
return 2
fact(2)
return 1
回归

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

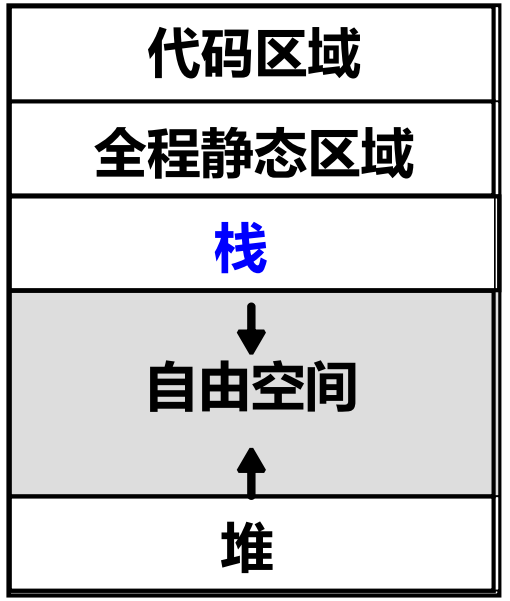
c++程序在执行时，将内存大方向划分为4个区域

代码区：存放函数体的二进制代码，有操作系统进行管理；

全局区：存放全局变量和静态变量以及常量；

栈区：由编译器自动分配释放，存放函数的参数值、局部变量等；

堆区：由程序员分配和释放，若程序员不释放，程序结束时由操作系统回收。



内存四区意义：

不同区域存放的数据，赋予不同的生命周期，给我们更大的灵活编程。

```cpp
// 全局变量属于全局区，由操作系统管理释放
int g_a = 1;
int g_b = 2;
int main(void)
{
    cout << "g_a 的地址为: \t"<< int(&g_a) << endl;
    cout << "g_b 的地址为: \t" << int(&g_b) << endl;
    // 创建普通的局部变量,属于栈区
    int a = 10;
    int b = 20;
    cout << "a 的地址为: \t" << int(&a) << endl;
    cout << "b 的地址为: \t" << int(&b) << endl;
    // 创建静态变量，属于全局区
    static int s_a = 40;
    static int s_b = 50;
    cout << "s_a 的地址为: \t" << int(&s_a) << endl;
    cout << "s_b 的地址为: \t" << int(&s_b) << endl;
    // 程序员自己创建变量，属于堆区
    int* d_a = new int(10);
    int* d_b = new int(20);
    cout << "d_a 的地址为: \t" << int(d_a) << endl;
    cout << "d_b 的地址为: \t" << int(d_b) << endl;
}
```

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

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

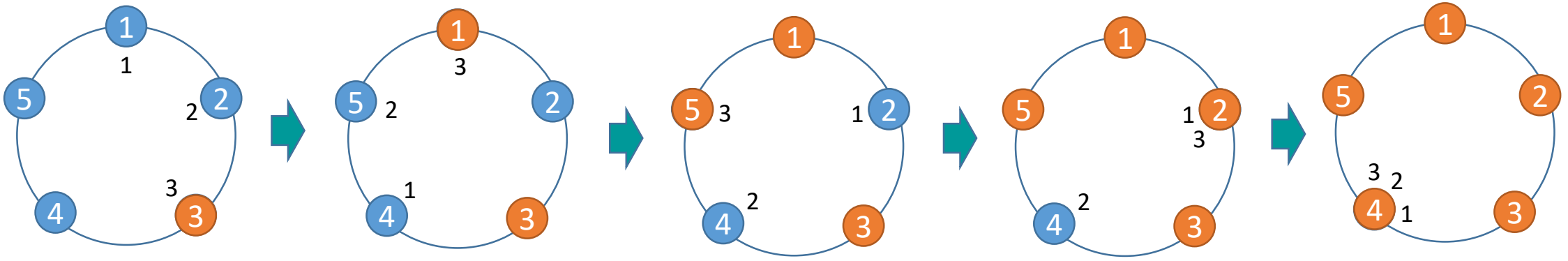
- 满足先来先服务特性的应用均可采用队列作为其数据组织方式或中间数据结构
- **调度或缓冲**
  - 消息缓冲器
  - 邮件缓冲器
  - 数据缓冲
  - 计算机的硬件设备间通信也需要队列作为数据缓冲
  - 操作系统的资源管理
- **宽度优先搜索**

队列应用：约瑟夫环问题

- 问题描述：

编号为1, 2, …, n的n个人围成一圈，从编号1的人开始报数，数到m的人出圈，然后下一个人重新开始报数，直到所有人出圈为止，求依次出圈的人的编号。

n=5, m=3



出圈顺序：3->1->5->2->4

队列应用：约瑟夫环问题

- 问题描述：

编号为1, 2, …, n的n个人围成一圈，从编号1的人开始报数，数到m的人出圈，然后下一个人重新开始报数，直到所有人出圈为止，求依次出圈的人的编号。

- 算法1：直接模拟

关键数据结构：循环链表、FIFO队列、并查集等

- 算法2：动态规划+数学分析（mod运算）

队列应用：约瑟夫环问题

- 问题描述：

编号为1, 2, …, n的n个人围成一圈，从编号1的人开始报数，数到m的人出圈，然后下一个人重新开始报数，直到所有人出圈为止，求依次出圈的人的编号。

- 算法1：直接模拟

```
1. InitQueue(queue)              //队列的初始化
2. for k ← 1 to n do
3. |   EnQueue(k)               //数字1--n依次入队
4. end
5. while Length(queue) > 0 do  //循环至队列为空
6. |  for i ← 1 to m-1 do     //重复m-1次出队后入队
7. |  |   data ← GetFront(queue)
8. |  |   DeQueue(queue)
9. |  |   EnQueue(data)
10.|  end
11.|  data ← GetFront(queue)
12.|  DeQueue(queue)  //去除第m次出队元素
13.|  print data                  //输出第m次出队元素
14.end
```

- 时间复杂度：O(nm)
- 空间复杂度：O(n)

队列应用：约瑟夫环问题

- 问题的来源：

1）在罗马人占领乔塔帕特后，39 个犹太人与Josephus及他的朋友躲到一个洞中，39个犹太人决定宁愿死也不要被敌人抓到，于是决定了一个自杀方式，41个人排成一个圆圈，由第1个人开始报数，每报数到第3人该人就必须自杀，然后再由下一个重新报数，直到所有人都自杀身亡为止。然而Josephus和他的朋友并不想遵从，开始时要站在什么地方才能避免被处决（即Josephus和他盆友能留到最后）？

2） 17世纪的法国数学家加斯帕在《数目的游戏问题》中讲了这样一个故事：15个教徒和15 个非教徒在深海上遇险，必须将一半的人投入海中，其余的人才能幸免于难，于是想了一个办法：30个人围成一圆圈，从第一个人开始依次报数，每数到第九个人就将他扔入大海，如此循环进行直到仅余15个人为止。问怎样排法，才能使每次投入大海的都是非教徒。

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

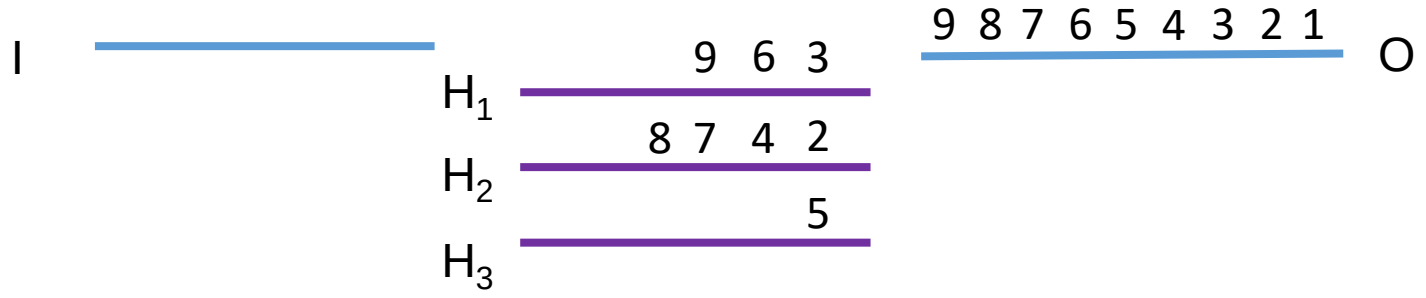
示例： 9节车厢的货车，使用3个容量为3的缓冲轨
初始状态： 9节车厢均在入轨排队
最终状态：9节车厢按递增序排在出轨 上

队列应用：车厢重排

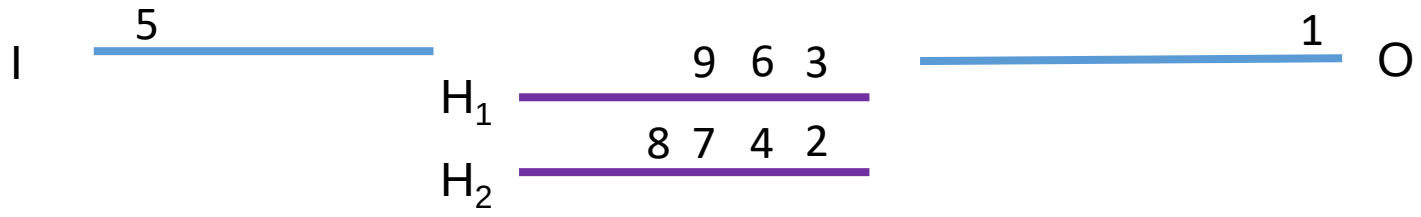
示例： 9节车厢的货车，使用3个容量为3的缓冲轨

初始状态： 9节车厢均在入轨排队

最终状态：9节车厢按递增序排在出轨 上



思考题：队列H3可否去掉?



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
最终状态：9节车厢按递增序排在出轨 上

*思考题：重排长度为n的车厢至少需要多少个缓冲轨（队列）？

*答案：缓冲轨数量的最小值 等于 输入序列中包含的最长递增子序列的数目减1！！

例如上述序列包含3条最长递增子序列：5 8 1 7 4 2 9 6 3

队列应用：车厢重排

算法：车厢重排 TrainCarriageScheduling(*in_track*, *out_track*, *n*, *k*)
输入：入轨的车厢序列*in_track*；车厢数量*n*≥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 ←1 to k do
2. |  InitQueue(buffer[i])
3. end
4. InitQueue(out_track)
5. next_out ← 1              //下一个待出轨车厢编号，从1还是计数
6. for i ←1 to n do
7. |  if in_track[i]=next_out then     //入轨中的第i个车厢正是待出轨车厢
8. |  |  EnQueue(out_track, i)         //直接移到出轨
9. |  |  next_out ← next_out + 1
10. |  else                           //不是待出轨车厢
11. |  |  for j←1 to k do             // 考察每一缓冲轨队列
12. |  |  |  front_crg ← GetFront(buffer[j]) // 查看队列j 的头元素
13. |  |  |  if front_crg≠NIL且 in_track[front_crg]=next_out then   //队头元素是待出轨元素
14. |  |  |  |  EnQueue(out_track, front_crg)
15. |  |  |  |  DeQueue(buffer[j])
16. |  |  |  |  next_out ← 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 ← 0
22. | | | max_buffer ← -1
23. | | | for j←1 to k do                    // 考察每一缓冲轨队列的队尾
24. | | | | rear_crg ← GetRear(buffer[j])
25. | | | | if rear_crg≠NIL且in_track[i]>in_track[rear_crg] then   //队非空且队尾元素小于入轨第i个元素
26. | | | | | if in_track[rear_crg] > max_rear then
27. | | | | | | max_rear ← in_track[rear_crg]  // 最大队尾元素值
28. | | | | | | max_buffer ← j          // 最大队尾元素所在的队列编号
29. | | | | | end
30. | | | | end
31. | | | end
32. | | | if max_buffer ≠ -1 then
33. | | | | EnQueue(buffer[max_buffer], i)
34. | | | else
35. | | | | for j←1 to k do
36. | | | | | if IsEmpty(buffer[j]) = true then
37. | | | | | | break
38. | | | | | end
39. | | | | end
```

队列应用：车厢重排

```
40. | | | | if  j ≤ 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      //将缓冲队列中的元素按序移到出轨
50. | for j←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 ←1 to k do
60. | DestroyQueue(buffer[i])
61. end
```

- 时间复杂度：O(nk)
- 空间复杂度：O(nk)

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

2< 8，直接 入栈

7> 2，弹出2，此时7 < 8，入栈

3 < 7，直接入栈

10> 3 及栈中所有元素，依此出栈后，10入栈

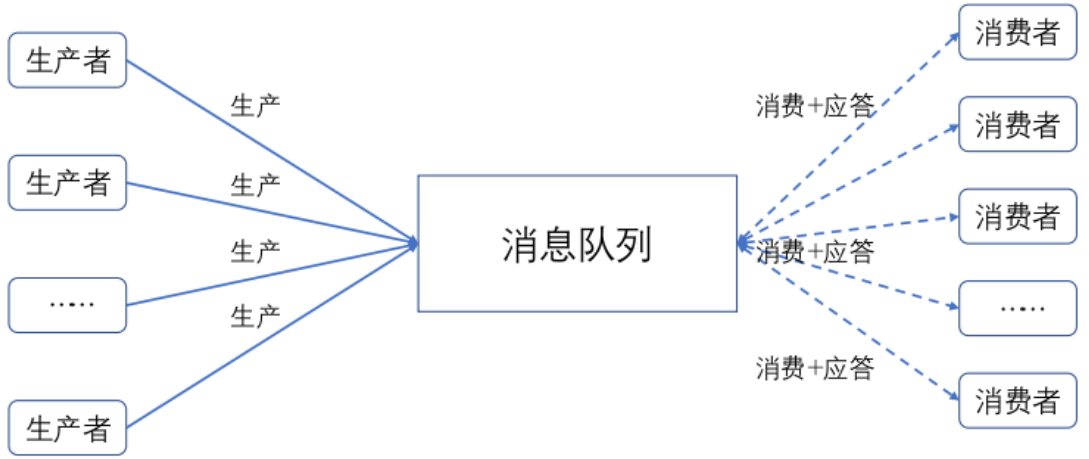
单调队列

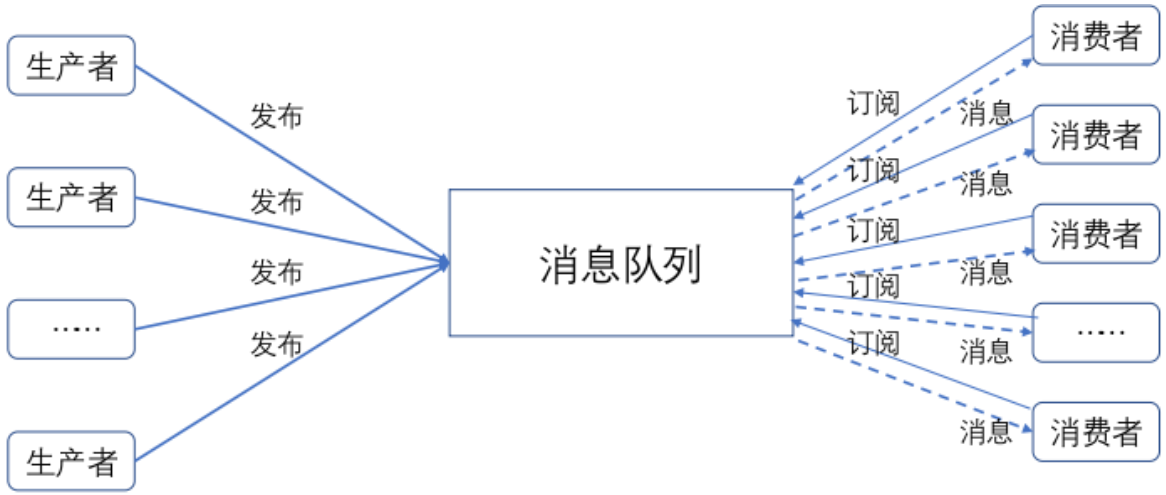
- 一种元素具有严格单调性的队列，满足两个约束：
  1. 队列元素从头到尾的严格单调性。
  2. 队列元素先进先出，当然队首元素比尾部元素要先进。
- 根据元素的递增或递减可组织为
  - ✓ 单调递增队列
  - ✓ 单调递减队列
- 单调队列的主要操作基本与队列相同，区别在于单调队列的元素入队时，需要单调性检查，通过删除队列中不满足单调性的元素来保持队列的单调性

- 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