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

俞勇、张铭、陈越、韩文弢

上海交通大学、北京大学、浙江大学、清华大学

第 3 章

栈与队列

赵海燕
北京大学

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

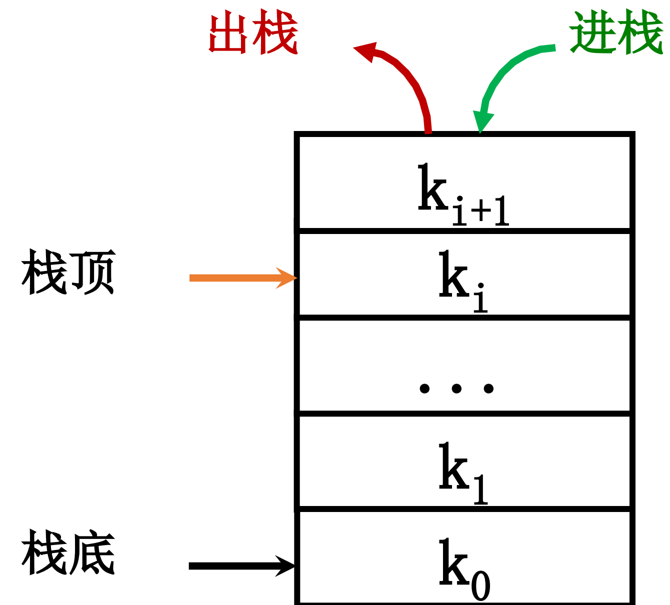
- **栈（Stack）**
  - 运算只在表的**一端**进行
- **队列（Queue）**
  - 运算只在表的**两端**进行

栈

- 后进先出（**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

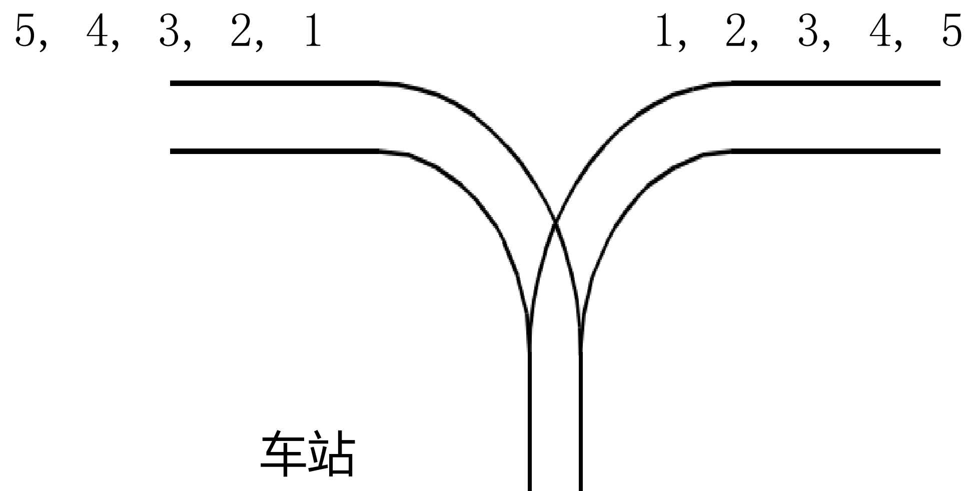
- 利用 合法的重构 发现 冲突

3 4 1 2

1 2 3 4

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

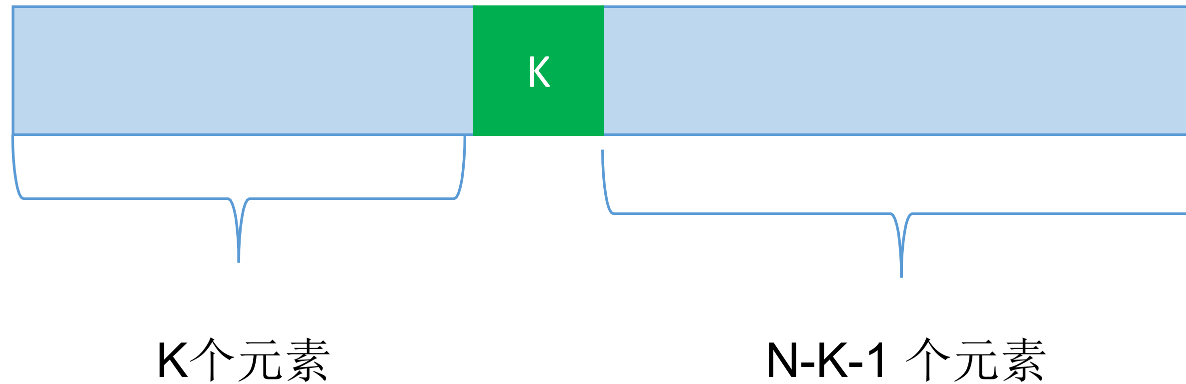
- 若入栈顺序为1, 2, 3, 4的话，则出栈的顺序可以有哪些?
  - 1234
  - 1243
  - 1324
  - 1342
  - 1423
  - 1432
  - 2134
  - 2143
  - ……

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？



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

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：

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

数据关系：

{<a_i, a_{i+1}>|a_i, a_{i+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

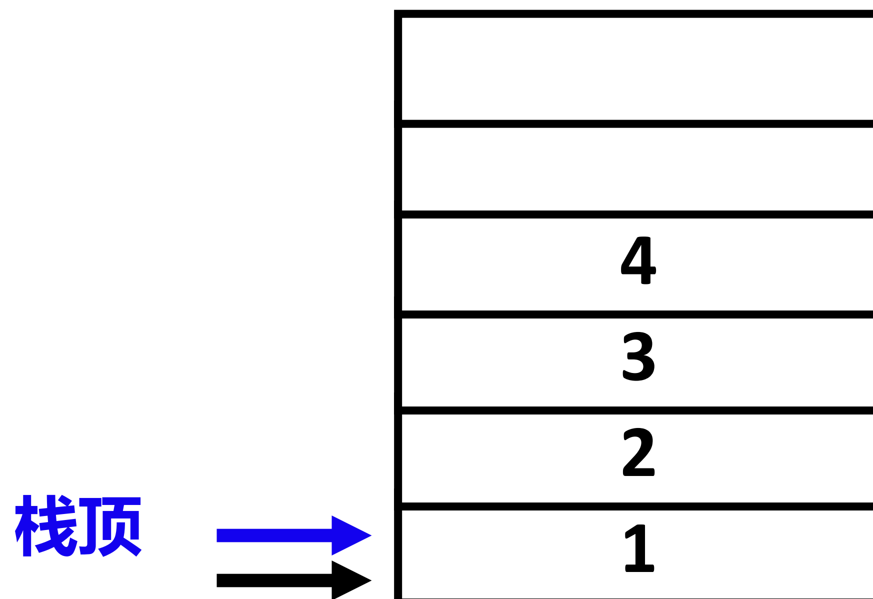
代码：顺序实现的栈的初始化 InitStack (*stack*, kSize)

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

1. *stack.capacity* ← 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. | $stack.top \leftarrow stack.top + 1$

5. | $stack.data[stack.top] \leftarrow x$

6. end

出栈操作

算法：顺序栈的取顶操作 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*)

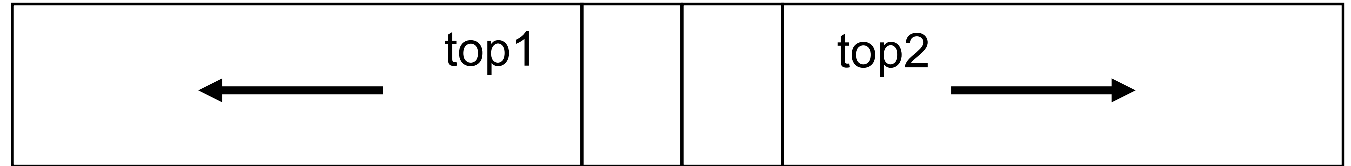
输入：栈*stack*

输出：删除栈顶元素后的顺序栈；若栈空，则退出

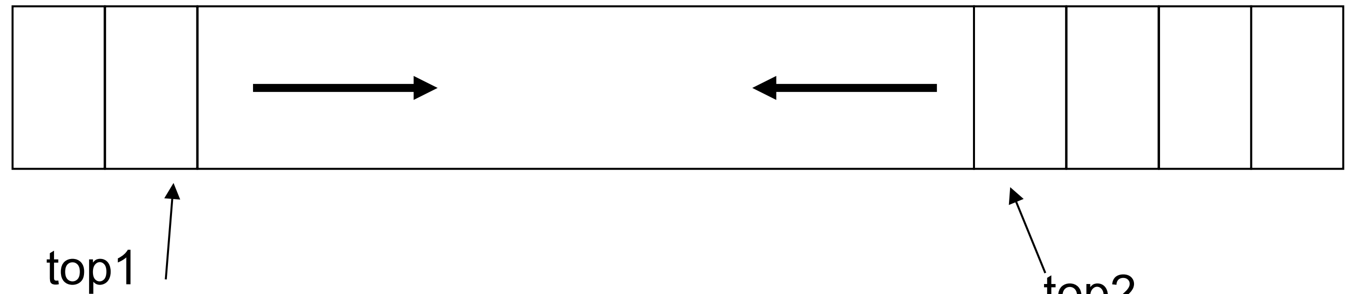
1. **if** IsEmpty(*stack*)=**true then**
2. | 栈空，退出
3. **else**
4. | $stack.top \leftarrow stack.top - 1$
5. **end**

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？



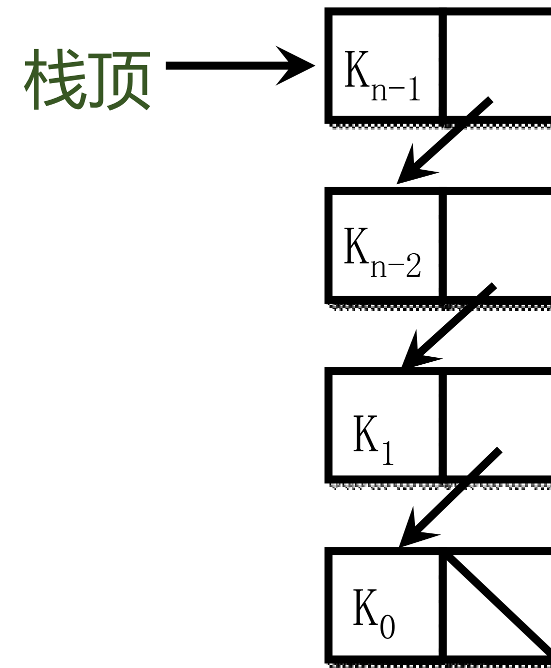
底部相连的栈



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* ← 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

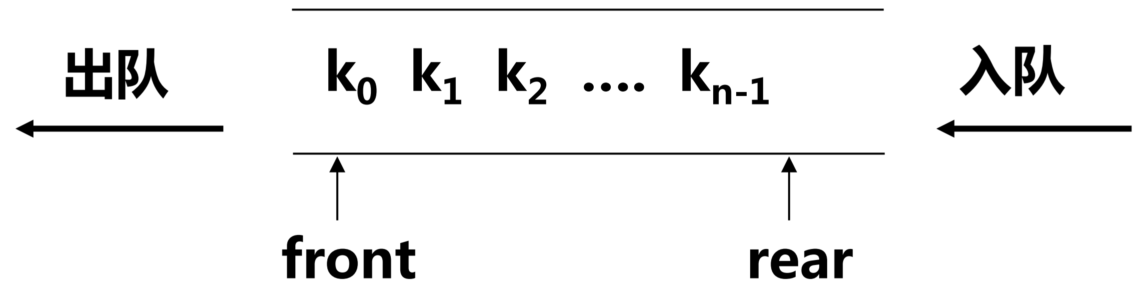
- 空间效率
  - ✓ 顺序栈结构紧凑，但须事先确定长度
  - ✓ 链式栈长度可变，增加结构性开销

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：

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

数据关系：

{<a_i, a_{i+1}>|a_i, a_{i+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

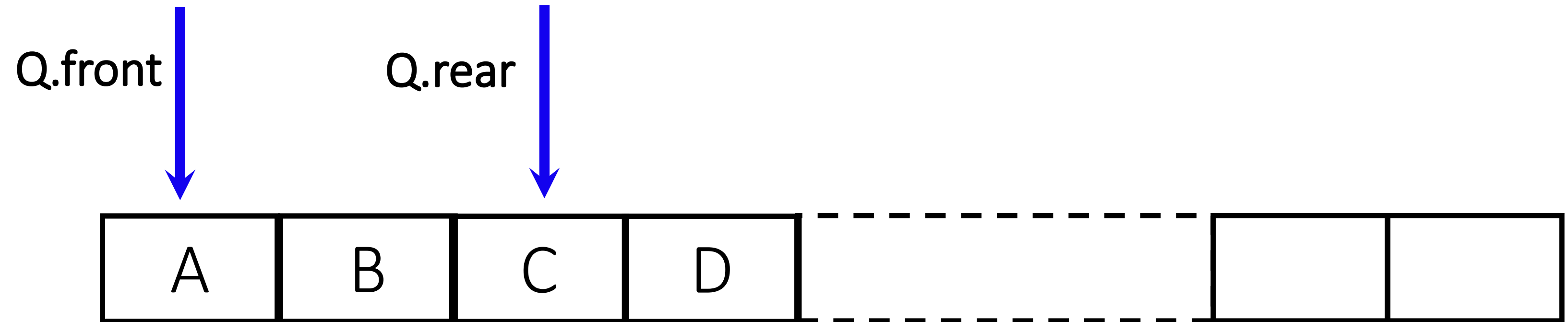
- 上溢
  - 当队列满时，入队操作所出现的现象
- 下溢
  - 当队列空时，出队操作所出现的现象
- 假溢出
  - 顺序队列可能出现的一种现象：当队尾指针达到最大值（rear = mSize）时，再做入队运算产生溢出，但此时队列前端可能尚有空闲位置

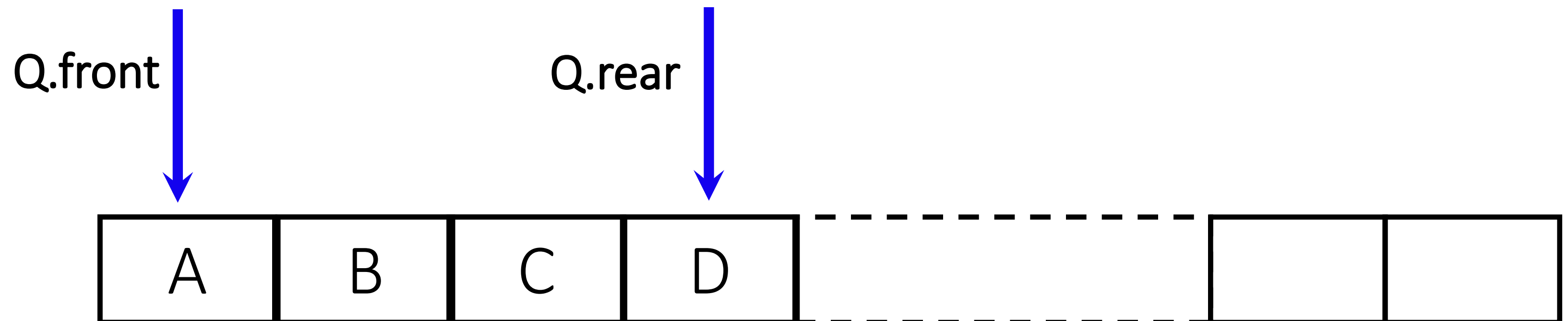
顺序队列

- 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

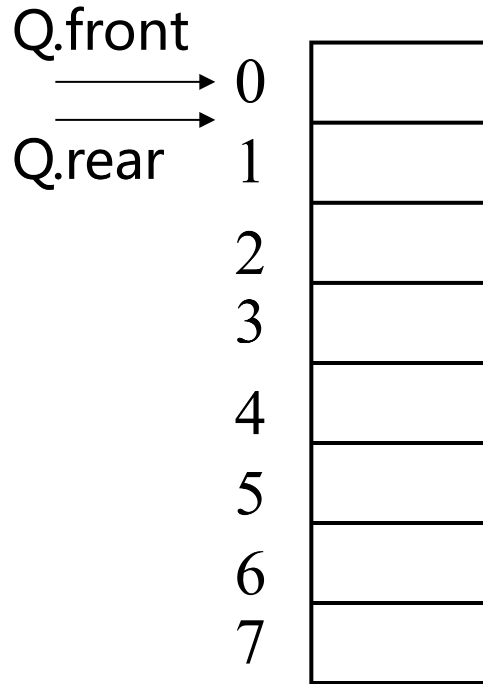
顺序队列的维护

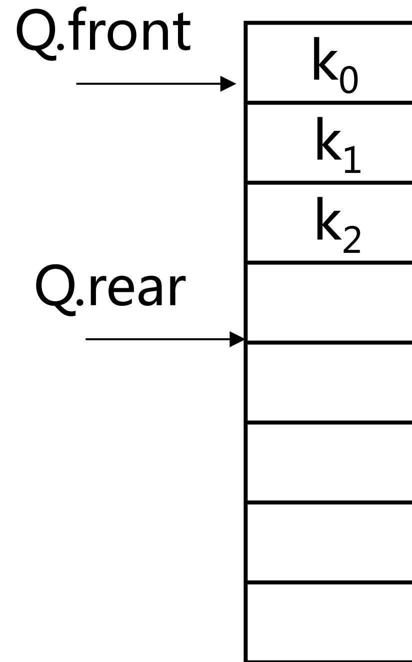
- 尾指针Q.rear 的处理
  - ✓ 实指
  - ✓ 虚指

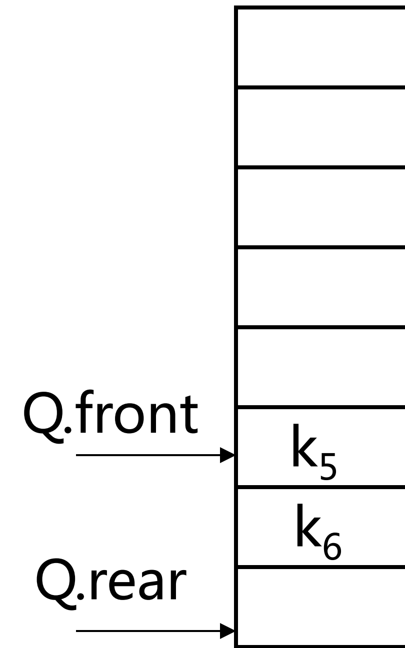




队列示意：普通



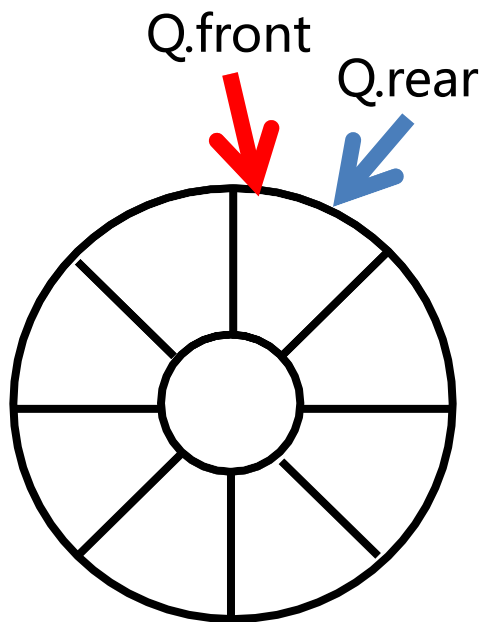


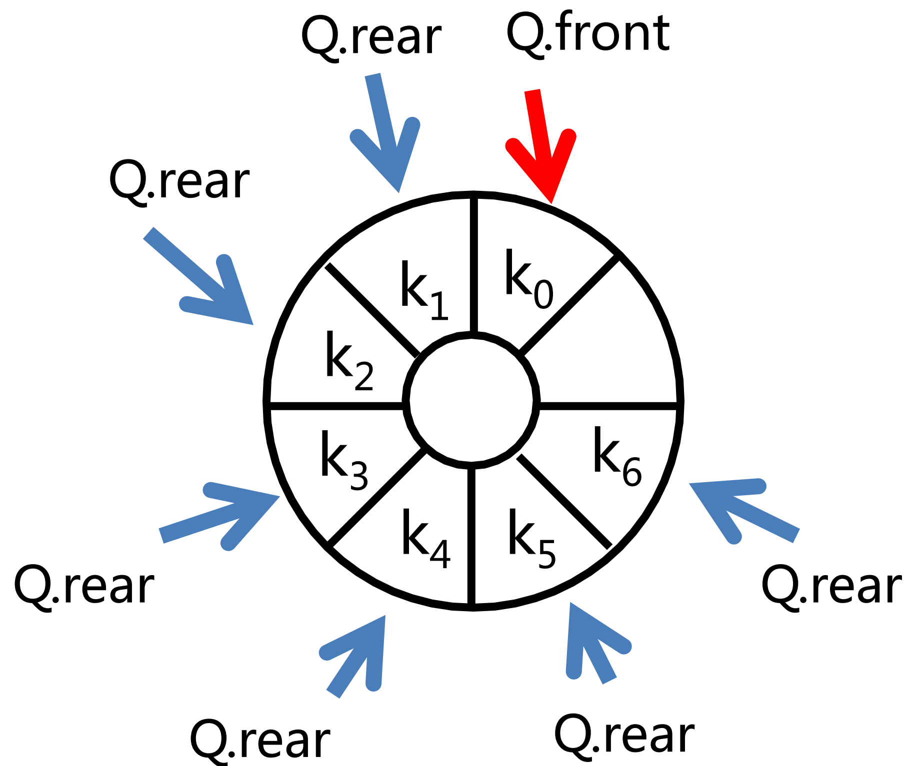


队列空

再进队一个元素如何？

队列示意：环形





队列空：

Q.rear == Q.front

队列满：

(Q.rear+1) mod 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
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

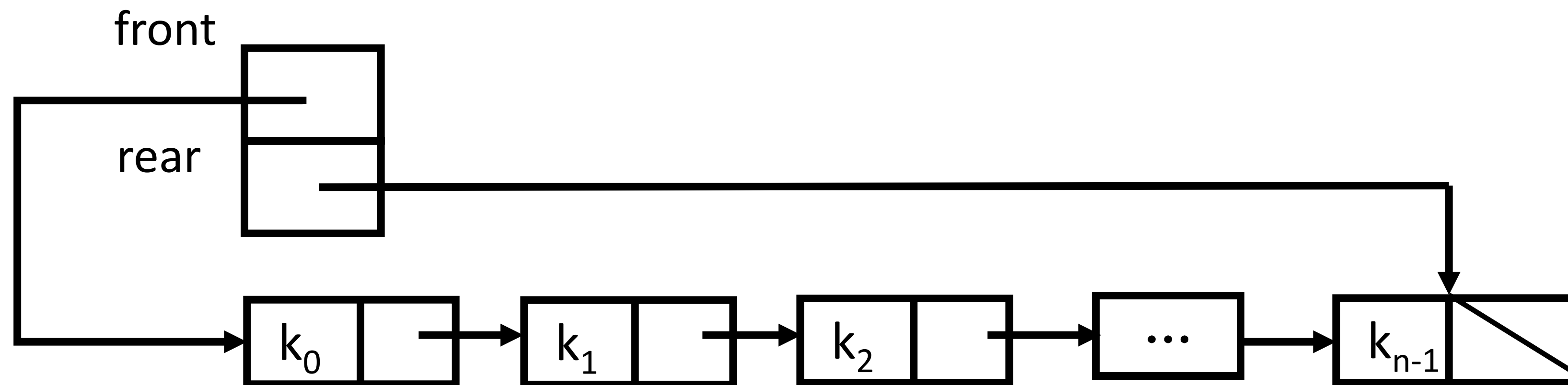
算法：顺序队列的出队操作 DeQueue(*queue*)

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

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：
https://d.book118.com/398034015055007001