第三章 线性表、堆栈和队列

- 3.1 线性表的定义和基本操作
- 3.2 线性表的顺序存储结构
- 3.3 线性表的链接存储结构
- 3.4 复杂性分析
- 3.5 堆栈
- 3.6 队列

两类操作受限的线性表，应用广泛

3.5 堆 栈

3.5.1 堆栈的定义和主要操作

3.5.2 顺序栈

3.5.3 链式栈

3.5.4 顺序栈与链式栈的比较

3.5.5 堆栈的应用

1、堆栈的定义

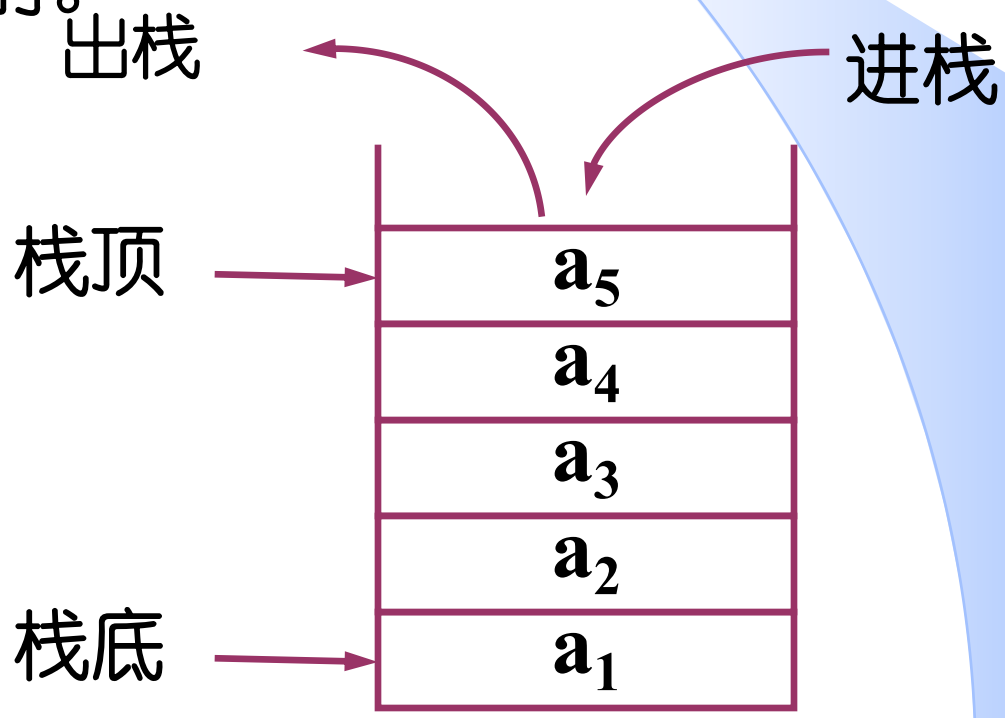
堆栈(简称栈)：栈是插入和删除只能在其同一端进行的线性表，并按后进先出的原则进行操作。

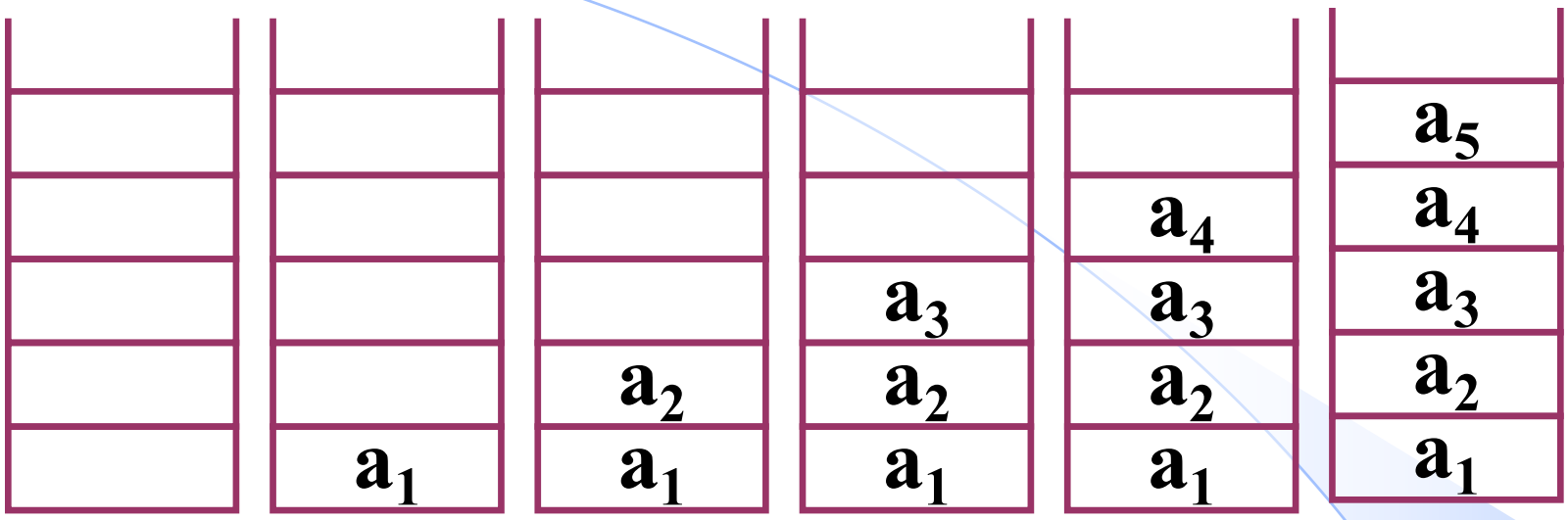
栈顶：进行插入、删除操作的一端；

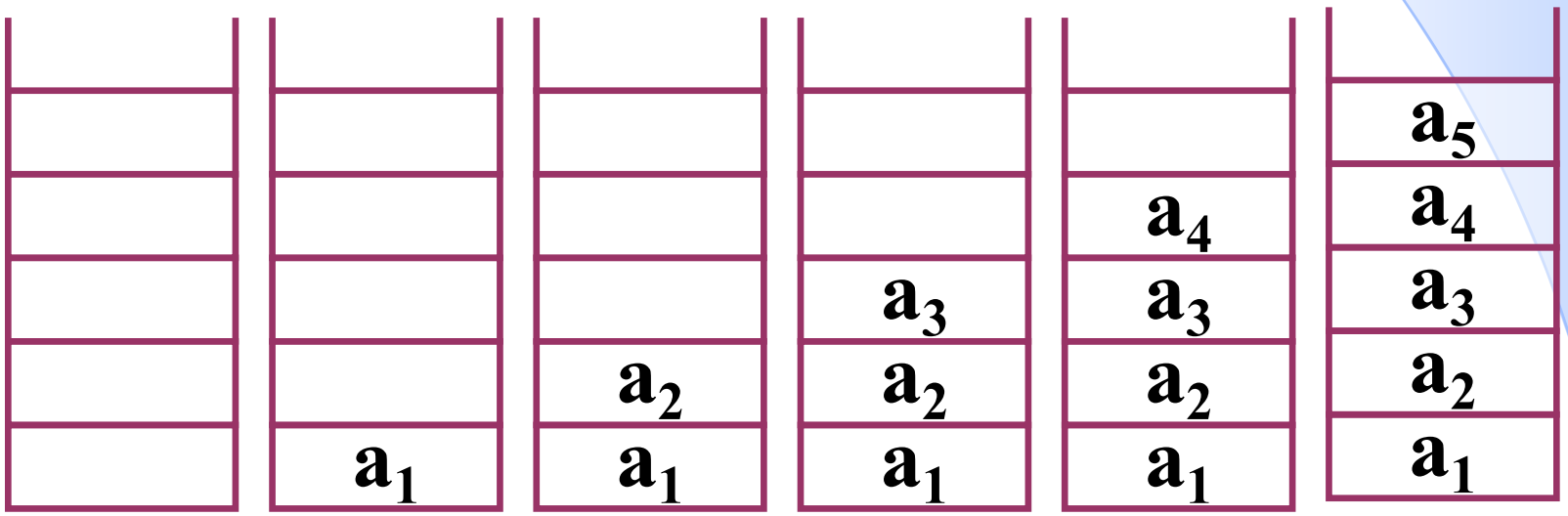
栈底：另一端；

空栈：表中没有元素时。

[例] 线性表 $(a_1, a_2, \ldots, a_5)$，进栈出栈情况




a1
a2
a1
a3
a2
a1
a4
a3
a2
a1
a5
a4
a3
a2
a1


a1
a2
a1
a3
a2
a1
a4
a3
a2
a1
a5
a4
a3
a2
a1

- **栈的后进先出性：**可以对输入序列部分或全局求逆；凡符合后进先出性，都可应用栈，如十进制数与其它数制的转换、递归的实现、算数表达式求值等问题。
- **栈的封闭性：**插入和删除都只能在栈顶进行，除了栈顶元素外，其他元素不会被改变。因而，栈的封闭性非常好，使用起来很安全。

2、堆栈的基本操作

- 插入操作也称为进栈或入栈；
- 删除操作也称为出栈、退栈或弹出堆栈；
- 栈也称为后进先出表。

（1）栈初始化Create(S)

（2）进栈 Push

（3）出栈 Pop

（4）读取栈顶元素 Peek

（5）判断栈空 IsEmpty

（6）判断栈满 IsFull

（7）置空栈 Clear

3.5 堆 栈

3.5.1 堆栈的定义和主要操作

3.5.2 顺序栈

3.5.3 链式栈

3.5.4 顺序栈与链式栈的比较

3.5.5 堆栈的应用

堆栈的顺序存储

使用数组存放栈元素，栈的规模必须小于或等于数组的规模，当栈的规模等于数组的规模时，就不能再向栈中插入元素。

存放堆栈元素的数组：

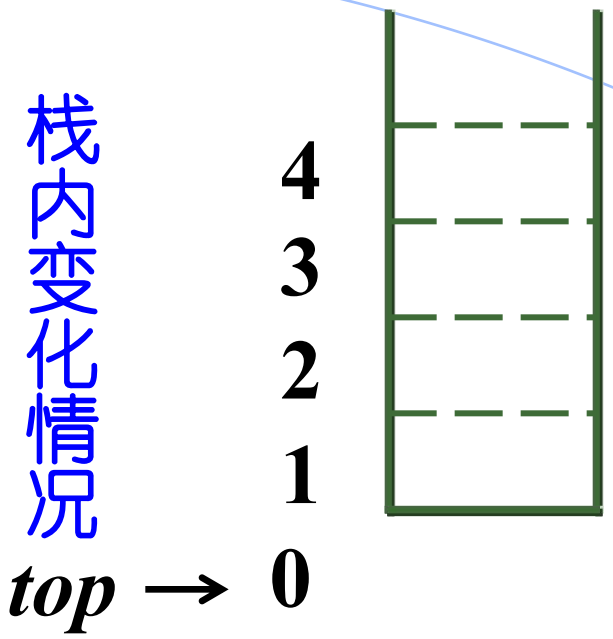
T A[size];

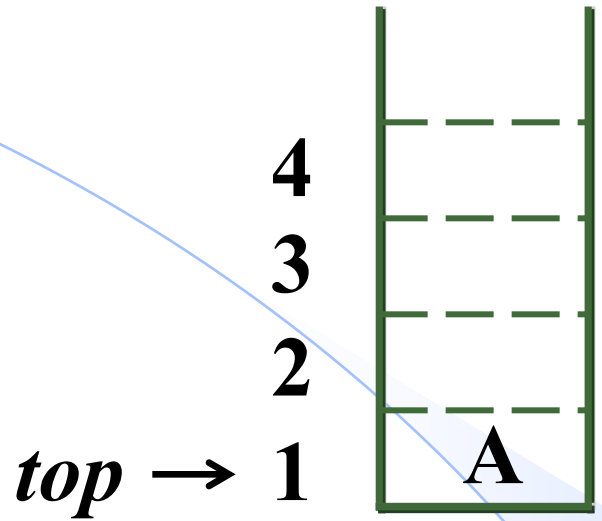
栈顶所在数组元素的下标：*top*;

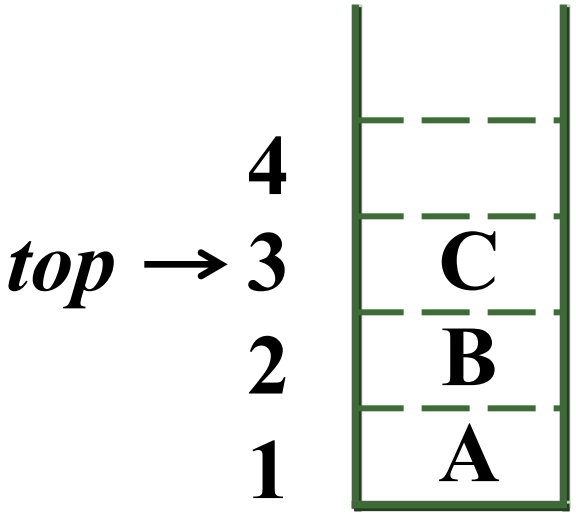
堆栈空： *top* = 0

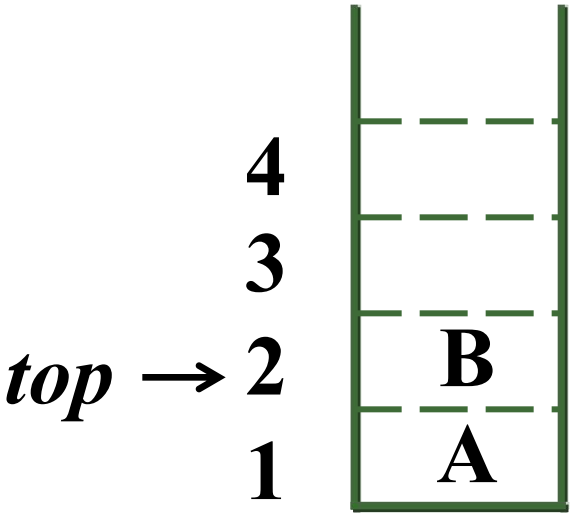
堆栈满： *top* = size

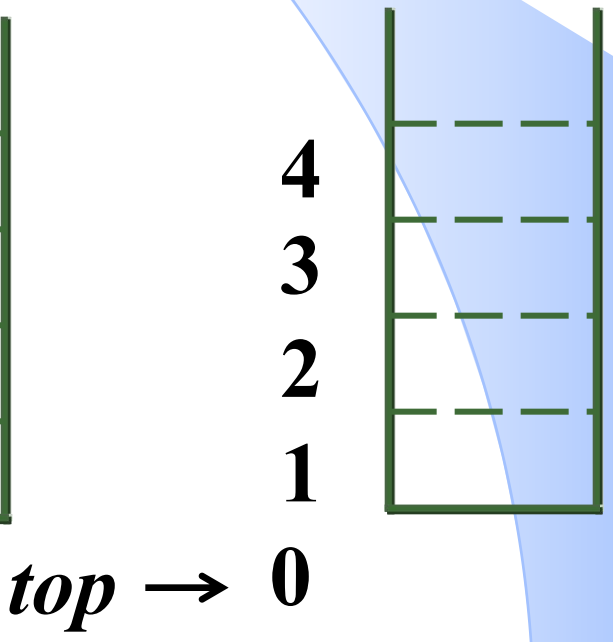
栈内变化情况











算法 **Push** (A, *item*. A) // 向顺序栈A中压入一个元素*item*

P1. [栈满？]

IF *top*=size THEN (

PRINT“栈满无法压入”.

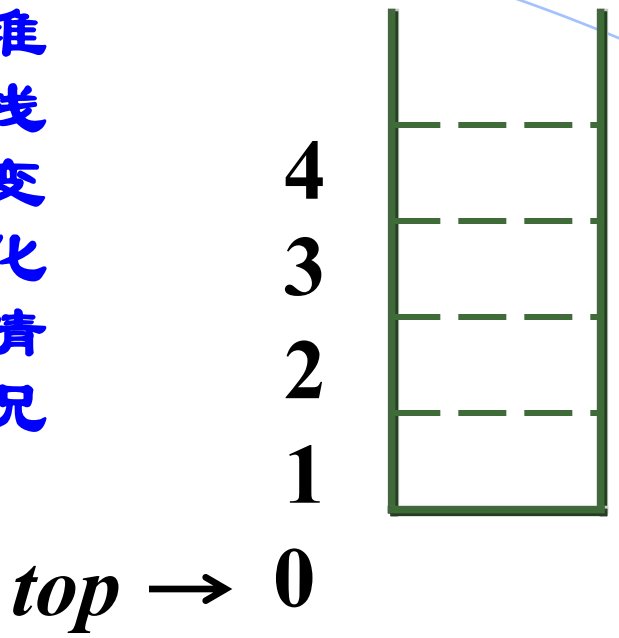
RETURN.)

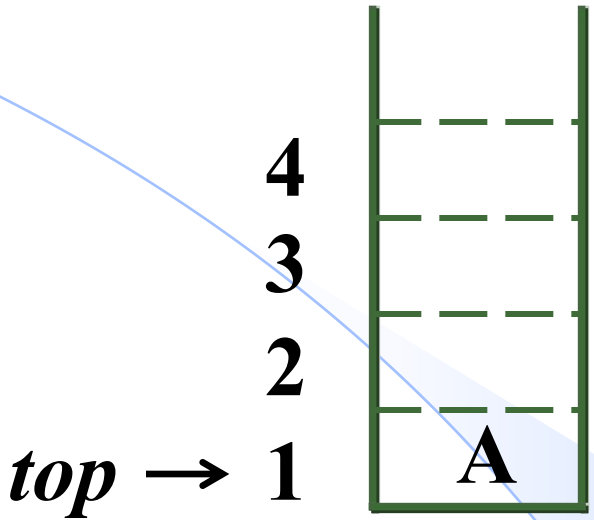
P2. [入栈]

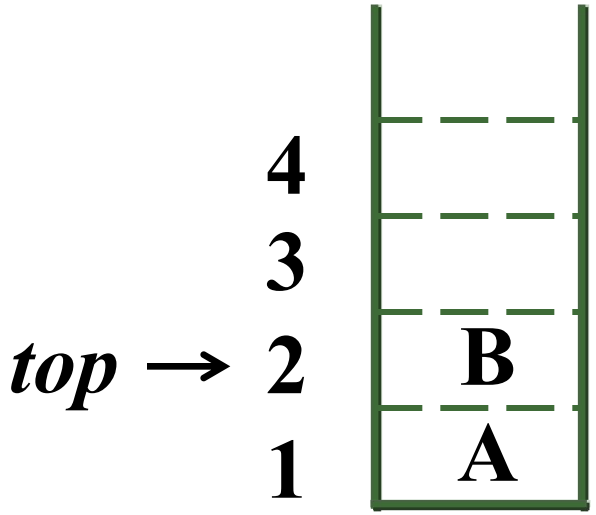
top←*top*+1. // 更新栈顶元素的下标

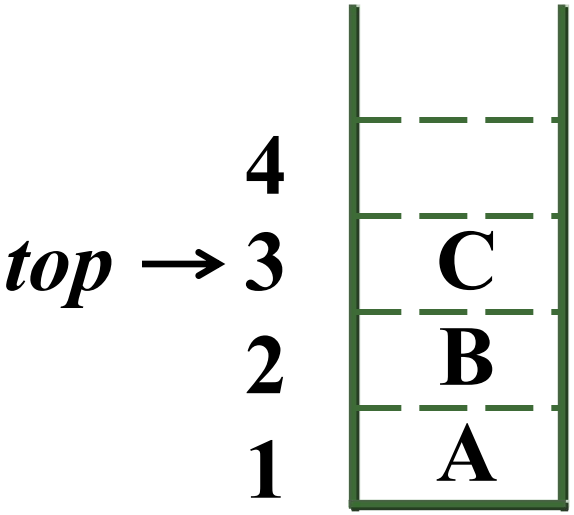
A[*top*]←*item*. // 压入新栈顶元素

堆栈变化情况









算法 **Pop** (A. *item*) /* 从顺序栈A中弹出栈顶元素，并存放在变量*item*中*/

P1. [栈空?]

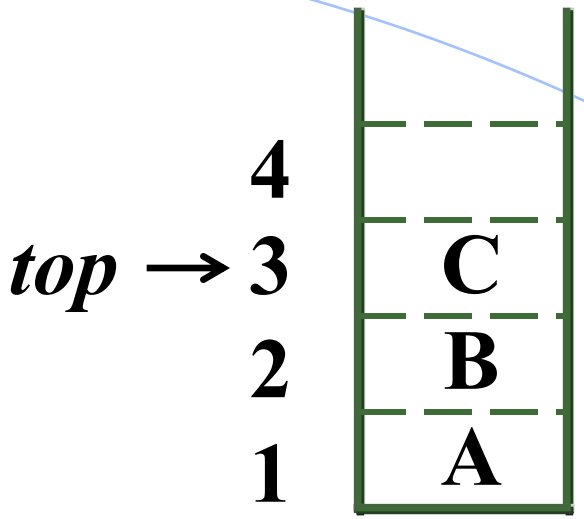
IF *top* = 0 THEN (PRINT “栈空无法弹出”.
RETURN.)

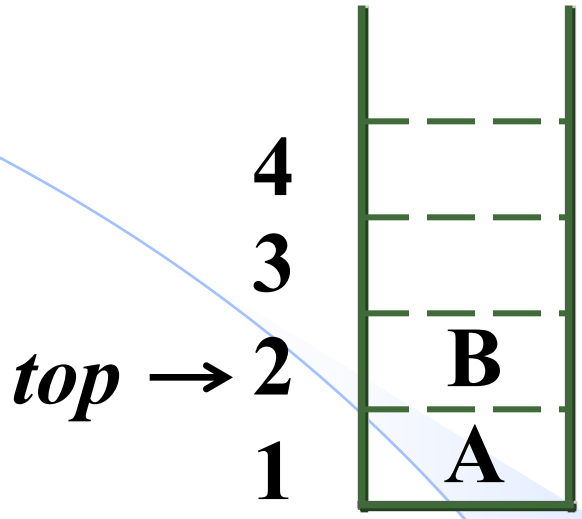
P2. [出栈]

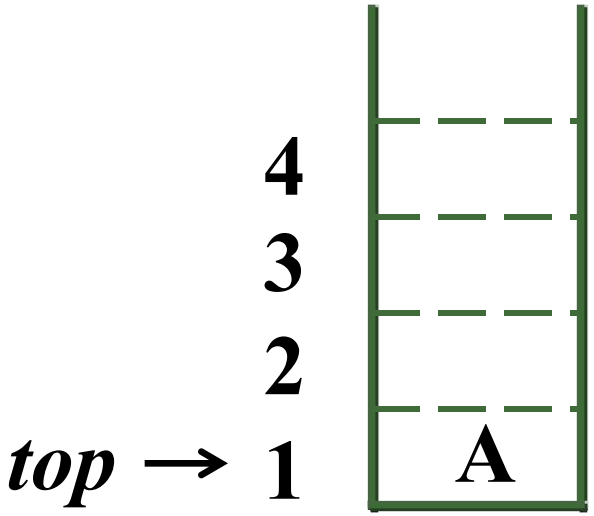
item←A[*top*]. // 保存栈顶元素值

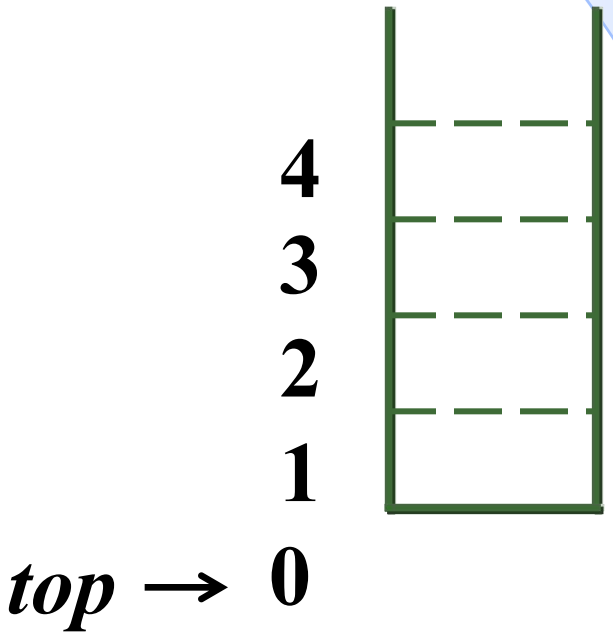
top←*top*–1. // 更新栈顶元素的下标

堆栈变化情况









算法 Peek (A. *item*) // 取栈A的栈顶元素存放在变量item中

P1. [栈空？]

IF *top* = 0 THEN (PRINT “栈空”. RETURN.)

P2. [读取栈顶元素]

item←A[*top*]. // 保存栈顶元素值

Peek与pop的区别是什么？

top ← *top* – 1

小结：

- 堆栈是一种操作受限制的线性表
- push 和 pop 操作只与栈顶有关
- 堆栈的特性：后进先出
- 堆栈的状态

  堆栈空： $top = 0$

  堆栈满： $top = \text{size}$

- 用数组实现栈（顺序栈）效率很高，但若同时使用多个栈，顺序栈将浪费大量空间。

3.5 堆 栈

3.5.1 堆栈的定义和主要操作

3.5.2 顺序栈

3.5.3 链式栈

3.5.4 顺序栈与链式栈的比较

3.5.5 堆栈的应用

3.5.3 堆栈的链式存储

- 用单链表实现堆栈要为每个栈元素分配一个额外的指针空间。
- 考虑：栈顶对应单链表的表头还是表尾？

  因栈主要操作（入栈、出栈、存取）的对象是栈顶元素，若栈顶对应表尾，则每次操作的时间复杂性为$O(n)$；若栈顶对应表头，则每次操作的时间复杂性为$O(1)$，显然，栈顶对应表头是合理的。
- 另外，链式栈中不需要哨位结点。

算法 **Push** (*item*. *top*) /* 向栈顶指针为*top*的链式栈中压入一个元素*item* */

P1. [创建新结点]

s⇐AVAIL. //为新结点申请空间

data(*s*) ←*item*. next(*s*) ←*top*. /* 新结点的数据域存放*item*，指针域存放原栈顶结点的地址信息*/

P2. [更新栈顶指针]

top←*s*. // 更新栈顶指针，令其指向新入栈结点

算法 Pop (. *item*) /* 从栈顶指针为*top*的链式栈中弹出栈顶元素，并存放在变量*item*中*/

P1. [栈空？]

IF *top*= NULL THEN (PRINT“栈空无法弹出”.
RETURN.)

P2. [出栈]

item←data(*top*). // 保存栈顶结点的字段值

q←next(*top*). // 令指针q指向次栈顶结点

AVAIL⇐*top*. // 释放栈顶结点的空间

top←*q*. // 更新栈顶指针，令其指向q所指结点

算法 **Peek** (***.item***) /* 将栈顶指针为*top*的链式栈的栈顶元素存放在变量*item*中*/

P1. [栈空？]

IF *top*=NULL THEN (PRINT“栈空”. RETURN.)

P2. [存取栈顶] /* 将栈顶结点的字段值保存在变量*item*中*/

item←data(*top*). ▌

算法 **Clear** (. *top*) // 将栈顶指针为*top*的链式栈清空

C1. [逐一出栈，直至栈空**]**

WHILE *top*≠**NULL DO**

(*q*←**next(***top***).** // 保存次栈顶结点的地址信息

AVAIL⇐*top*. // 释放栈顶结点的存储空间

top←*q*.) // 更新栈顶指针 ▌

3.5 堆 栈

3.5.1 堆栈的定义和主要操作

3.5.2 顺序栈

3.5.3 链式栈

3.5.4 顺序栈与链式栈的比较

3.5.5 堆栈的应用

顺序栈与链式栈的比较

- 在空间复杂性上，顺序栈必须初始就申请固定的空间，当栈不满时，必然造成空间的浪费；链式栈所需空间是根据需要随时申请的，其代价是为每个元素提供空间以存储其*next*指针域。
- 在时间复杂性上，对于针对栈顶的基本操作（压入、弹出和栈顶元素存取），顺序栈和链式栈的时间复杂性均为$O(1)$.
- 存取非栈顶元素，如需对非栈顶元素进行存取，用数组实现的顺序栈可以快速定位，其时间复杂性为$O(1)$，链式栈为$O(n)$。

多栈共享邻接空间

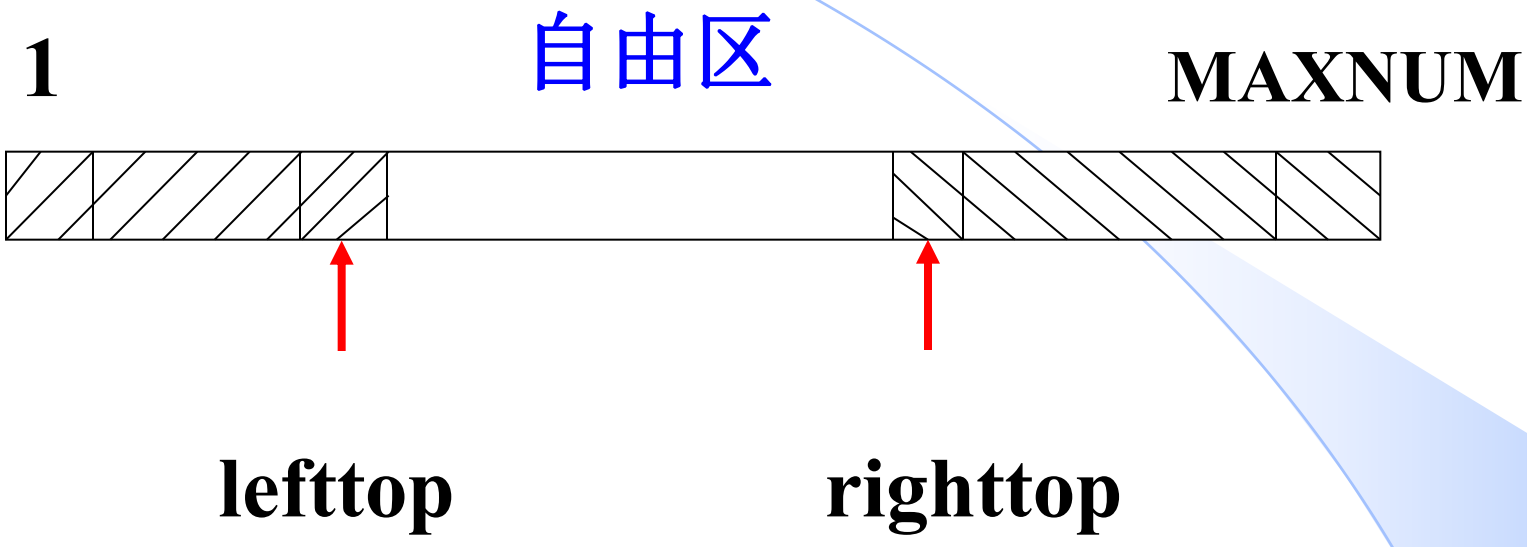
在计算机系统软件中，各种高级语言的编译系统都离不开栈的使用。

常常一个程序中要用到多个栈，为了不发生上溢错误，必须给每个栈预先分配一个足够大的存储空间，但实际中很难准确地估计。

另一方面，若每个栈都预分配过大的存储空间，势必会造成系统空间紧张。若让多个栈共用一个足够大的连续存储空间，则可利用栈的动态特性使它们的存储空间互补。这就是栈的共享邻接空间。

双向栈在一维数组中的实现

- 栈的共享中最常见的是两栈的共享。假设两个栈共享一维数组stack[MAXNUM]，则可以利用栈的“栈底位置不变，栈顶位置动态变化”的特性，两个栈底分别为1和MAXNUM， 而它们的栈顶都往中间方向延伸。因此，只要整个数组stack[MAXNUM]未被占满，无论哪个栈的入栈都不会发生上溢。



两个栈共享邻接空间如图所示，左栈入栈时，栈顶指针加1，右栈入栈时，栈顶指针减1。

3.5 堆 栈

3.5.1 堆栈的定义和主要操作

3.5.2 顺序栈

3.5.3 链式栈

3.5.4 顺序栈与链式栈的比较

3.5.5 堆栈的应用

堆栈的应用——括号匹配

- 高级语言程序设计中的各种括号应该匹配，开括号与相应的闭括号匹配。例如：“(”与“)”匹配、“[”与“]”匹配、“{”与“}”匹配等。
- 字符串{a=(b×c}+free( )]中的括号就没有匹配上，因为串中第一个闭括号“}”和最近的未匹配开括号“(”不匹配。

检查输入文本文件中的括号是否正确匹配，文件中的字符是按次序依次输入的。

- 如果输入的字符是开括号，应该将其存放起来，继续下一个字符的输入；
- 如果输入的是闭括号，应考察其与最近的未匹配开括号是否匹配，若匹配，则应将匹配的开括号从存放处删除。
- 每次用来和闭括号进行匹配的开括号都是最后输入的，这符合堆栈的后进先出性质，因此用堆栈来存放开括号是合理的。

算法思想

- 根据匹配规则：后遇到的开括号先匹配
- 采用栈存放开括号，将当前闭括号和栈顶开括号进行匹配
- 考虑匹配失败的情况

算法 MatchBrackets(string . B) /*假定一个字符串存储在数组string中，字符串长度为*length*。检查字符串中括号是否匹配*/

MB1. [检测一个关括号是否与栈顶开括号匹配？]

CREATE(S). /* 操作CREATE(*S*) 创建一个空堆栈*S* */

FOR i =1 TO *length* DO

(// 若检测到开括号，则将其压入栈

IF (string[*i*]= ‘{‘ OR string[*i*] = ‘[‘ OR string[*i*] = ‘(‘)

THEN S ⇐ string[*i*] . //将开括号压入堆栈S

// 若检测到关括号，则检测是否匹配

ELSE (IF (string[*i*]= ‘}‘ OR string[*i*]= ‘]‘ OR string[*i*]= ‘)‘)

THEN (IF IsEmpty (S)

THEN (PRINT“Unmatched closing bracket”.

B←false. RETURN.) //匹配不成功

```
        open_bracket ⇐ S  . //取出栈顶开括号
        IF NOT( ( open_bracket = '(' AND string[i] = ')' )
            OR ( open_bracket = '[' AND string[i] = ']' )
            OR ( open_bracket = '{' AND string[i] = '}' ) )
        THEN ( PRINT"Unmatched closing bracket".
                    B←false. RETURN.)  //匹配不成功
          )
      )
    )
// 若堆栈中仍有开括号，说明匹配失败，因已无关括号与之匹配
IF ( NOT IsEmpty (S) )
  THEN ( PRINT"Unmatched bracket".  B←false. RETURN. )
   ELSE ( PRINT"Matched bracket".  B←true. RETURN. )
```

堆栈的应用——将十进制数转换成其它进制数

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

先求得的余数在写出结果时最后输出，最后求出的余数最先输出，符合栈的后进先出性质，所以可用栈来实现数制转换。

十进制数转换为r进制数

1. 十进制整数 x 除以基数 r，所得整余数是 r 进制数 y 的最低位 y_1，压入堆栈；
2. 用 x除以 r 的整数商，除以 r，所得整余数是 y 的次低位 y_2，压入堆栈；
3. 依此类推，
4. 直到商为 0 ，所得整余数是 y 的最高位y_m，压入堆栈；

   此时，栈中的 m 个 r 进制数即为所得。

栈的应用—算术表达式求值

算术表达式求值是程序设计语言编译中的一个最基本问题。它的实现方法是栈的一个典型的应用实例。

表达式都是由操作数（operand）、运算符（operator）和界限符（delimiter）组成的。

其中操作数可以是常数，也可以是变量或常量的标识符；运算符是算术运算符（ + ， － ， × ， / ）；界限符为左右括号和标识表达式结束的结束符。

算术表达式的表示方法

1. 中缀表达式——运算符在操作数之间

如： **A × B / C**

运算规则：

(1) 先计算括号内，后计算括号外；

(2) 在无括号或同层括号内，先进行乘除运算，后进行加减运算，即乘除运算的优先级高于加减运算的优先级；

(3) 同一优先级运算，从左向右依次进行。

用计算机来处理中缀表达式比较复杂。一个中缀表达式中有多少个运算符，原则上就得对表达式扫描多少遍，才能完成计算。

在编译系统中，把中缀表达式转换成另外一种表示方法，即后缀表达式，然后对后缀表达式进行处理，后缀表达式也称为逆波兰式。

2. 后缀表达式

1929 年，由波兰逻辑学家（Lukasiewicz）提出。

[例] A × B / C;

A B × C / ;

定义：运算符紧跟在两个操作数之后的表达式称为后缀表达式。

优点：① 后缀表达式没有括号

② 不存在优先级的差别

③ 计算过程完全按照运算符出现的先后次序进行

中缀表达式	后缀表达式
a+b	a b+
a+b×c	a b c×+
a×b×c+c×d	a b × c × c d × +
(a+b)×((c-d)×e+f)	a b + c d – e × f + ×
A + (B – C / D) × E	ABCD/－E×+

[例] 计算后缀表达式ABC×/DE×+AC×－

- 后缀表达式求值的方法

① 从左到右读入后缀表达式，若读到的是操作数，将它压入堆栈。

② 若读到的是运算符，就从堆栈中连续弹出两个元素（操作数），进行相应的运算，并将结果压入栈中。

③ 读入结束符时，栈顶元素就是计算结果。

操作	后缀表达式	栈
——	ABC×/DE×+AC×－ =;	ABC
T_1=B×C	AT_1/DE×+AC×－ =;	AT_1
T_2 =A/T_1	T_2 DE×+AC×－ =;	T_2 DE
T_3 =D×E	T_2T_3 +AC×－ =;	T_2 T_3
T_4=T_2+T_3	T_4AC×－ =;	T_4AC
T_5 =A×C	T_4 T_5 －=;	T_4 T_5
T_6 =T_4－ T_5	=;	T_6

定义两个函数

APPLIED(P, *x* , *y*) ： 将运算符 **P** 作用于操作数 ***x*** 和 ***y*** 得到结果；

ISOPTOR(*x*): 若***x***为运算符时为真，否则为假；

后缀表达式求值的ADL算法：

```
算法EPE ( p, n )    // p存储长度为n的后缀表达式
   EPE1 [初始化]
        CREATS ( S ) .
   EPE2 [表达式求值]
        FOR i ← 1 TO n DO
              IF NOT ( ISOPTOR( P[ i ] ))
              THEN S ⇐ P[ i ]
              ELSE ( y ⇐ S . x ⇐ S.
                     S ⇐ APPLIED(P[ i ], x , y) )
```

中缀表达式转换成后缀表达式

首先设定一个运算符栈，用来保存扫描中缀表达式得到的暂不能放入后缀表达式中的运算符。

基本思想：从左到右依次读出中缀表达式中的各个符号（操作数或运算符），每读出一个符号后，根据如下运算规则进行处理：

1. 假如是操作数，将其放入后缀表达式中；

2. 如果是运算符，则：

（1）栈空：运算符放入栈中；

（2）栈不空：比较当前读到的运算符与栈顶运算符的优先级；

1）假如读出的运算符的优先级大于栈顶运算符的优先级，则将其压入运算符栈，读中缀表达式的下一个符号；

2）若栈顶运算符的优先级比读到的运算符的优先级高或二者相等，弹出栈顶运算符放入后缀表达式中，返回 2）继续比较；

3）遇到“（”，压入堆栈；

4）遇到“）”，把“（”上面的操作符依次弹出加到后缀表达式中，“（”出栈。

3. 假如读出的是表达式结束符“#”，栈中剩余的运算符依次出栈并写入到后缀表达式中，转换完成。

将(A+B)×((C-D)×E+F)转换成后缀表达式

AB+CD-E×F+×

(A+B)×((C-D)×E+F)

栈	输出序列
（	A
（＋	A B
	A B ＋
×((	A B ＋C
×((-	A B ＋CD
×(	A B ＋CD-
×(×	A B ＋CD- E
×(＋	A B ＋CD- E×
×(＋	A B ＋CD- E × F
×	A B ＋CD- E × F+
	A B ＋CD- E × F+ ×

第三章 线性表、堆栈和队列

- 3.1 线性表的定义和基本操作
- 3.2 线性表的顺序存储结构
- 3.3 线性表的链接存储结构
- 3.4 复杂性分析
- 3.5 堆栈
- 3.6 队列

3.6 队 列

3.6.1 队列的定义和主要操作

3.6.2 顺序队列

3.6.3 链式队列

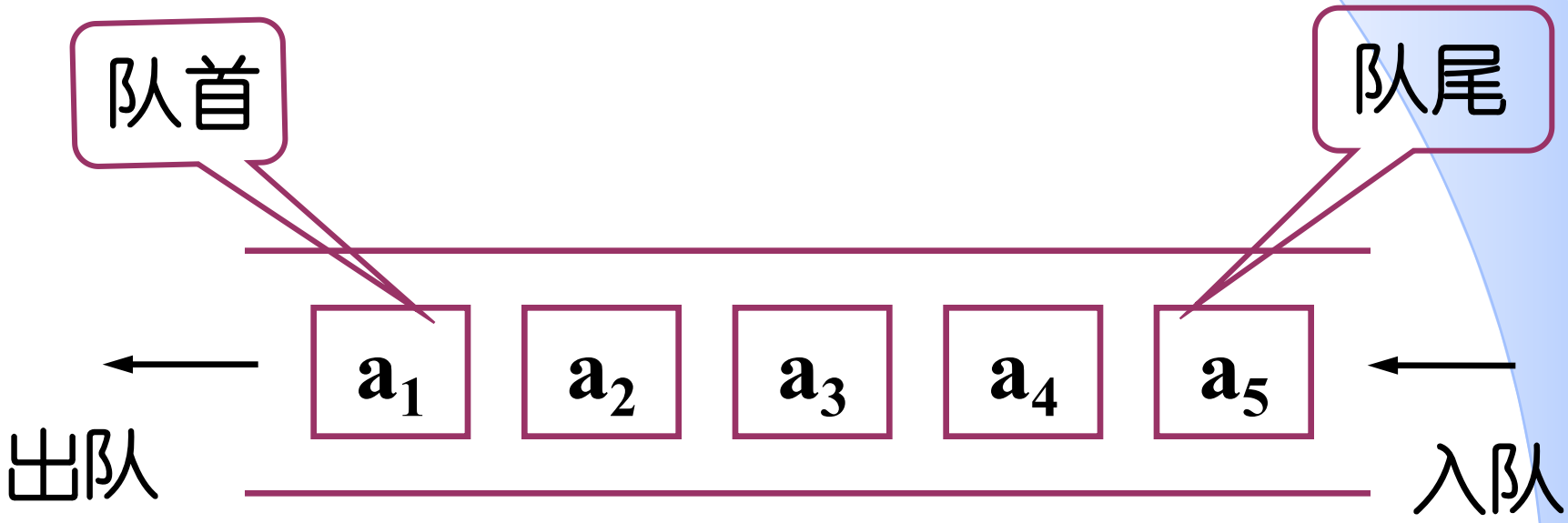
1、队列的定义

队列的定义：队列是插入操作在一端进行而删除操作在其另一端进行的线性表。按**先进先出**的原则进行操作。

能进行删除操作的一端称为**队首**（*front*）；

能进行插入操作的一端称为**队尾**（*rear*）；

没有元素的队列称为**空队列**。



- **队列的先进先出性：**可以对输入序列起到缓冲作用；凡符合先进先出性，都可应用队列，如操作系统中作业调度、图的广度优先搜索等问题。
- **队列的封闭性：**与栈类似，队列的封闭性也非常好，使用起来很安全。

2、队列的基本操作

（1）队列初始化

（2）入队（插入）

（3）出队（删除）

（4）读取队首元素

（5）判断队列是否为空

（6）确定队列中元素个数

（7）置空队列

3.6 队 列

3.6.1 队列的定义和主要操作

3.6.2 顺序队列

3.6.3 链式队列

队列的顺序存储

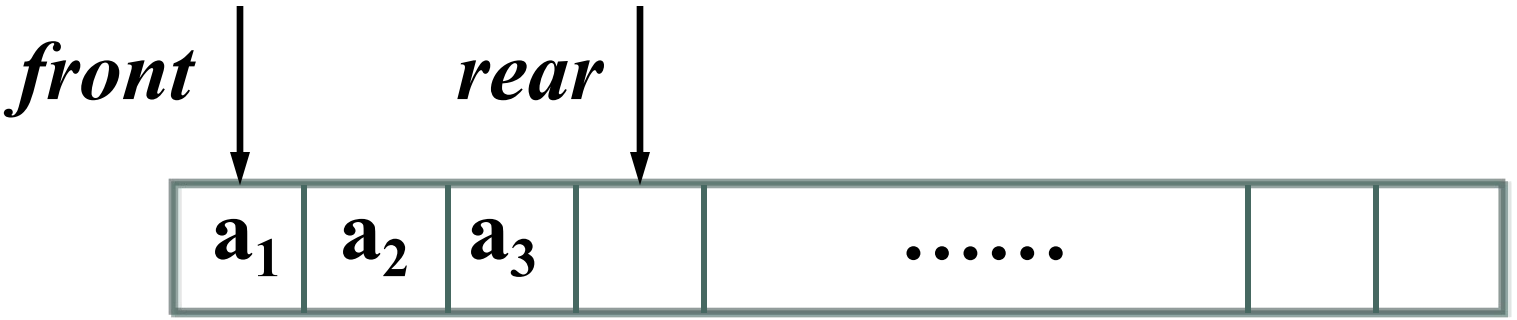
存放队列元素的数组：

T A[Size]

front 队首元素的数组下标

rear (要入队元素的下标) 队尾元素的下标加1

count 队列中元素的个数



[例] 等待处理的作业进队、出队情况

初始状态

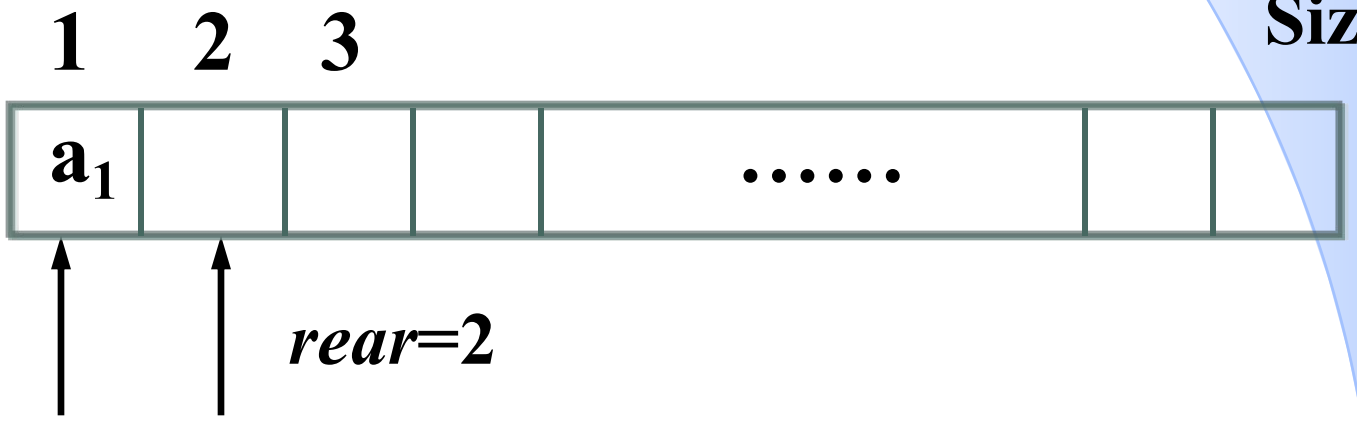
front=1

Count=0

rear=1

插入： *rear*=*rear*+1

a_1进队



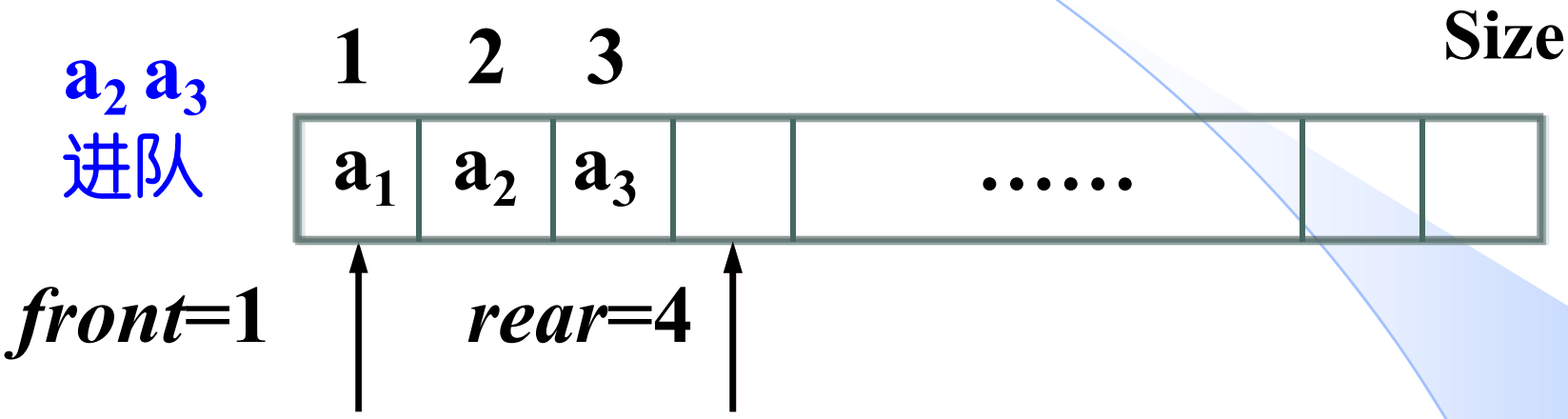
front=1

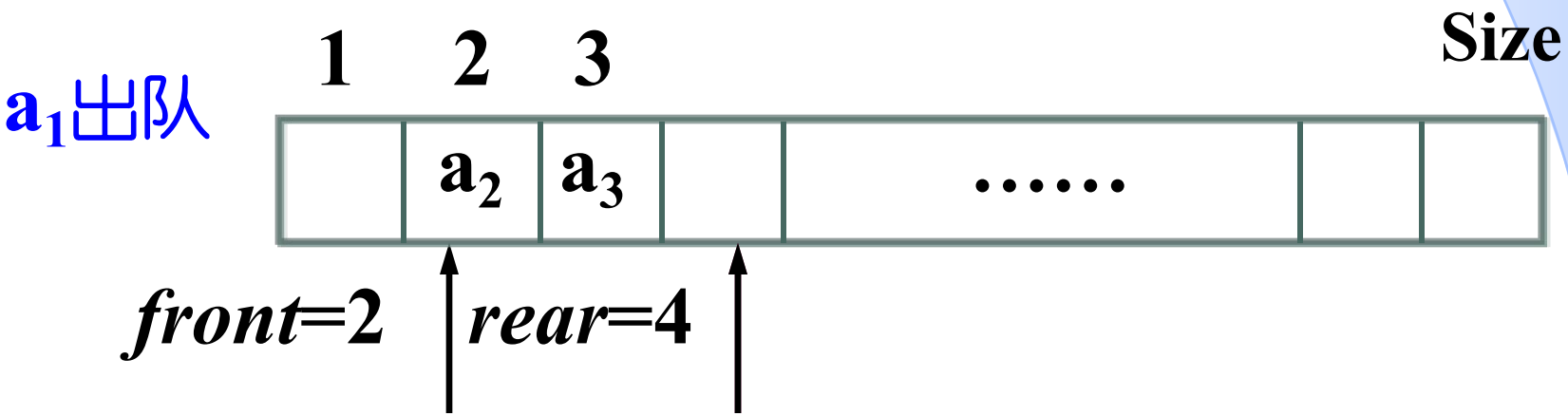
Count=1

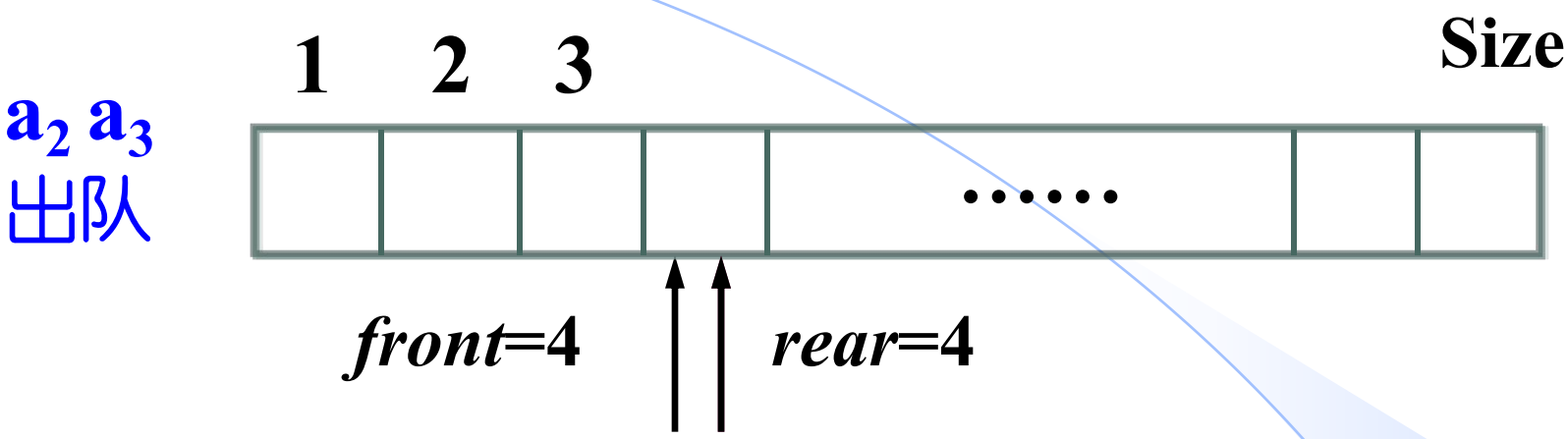
rear=2

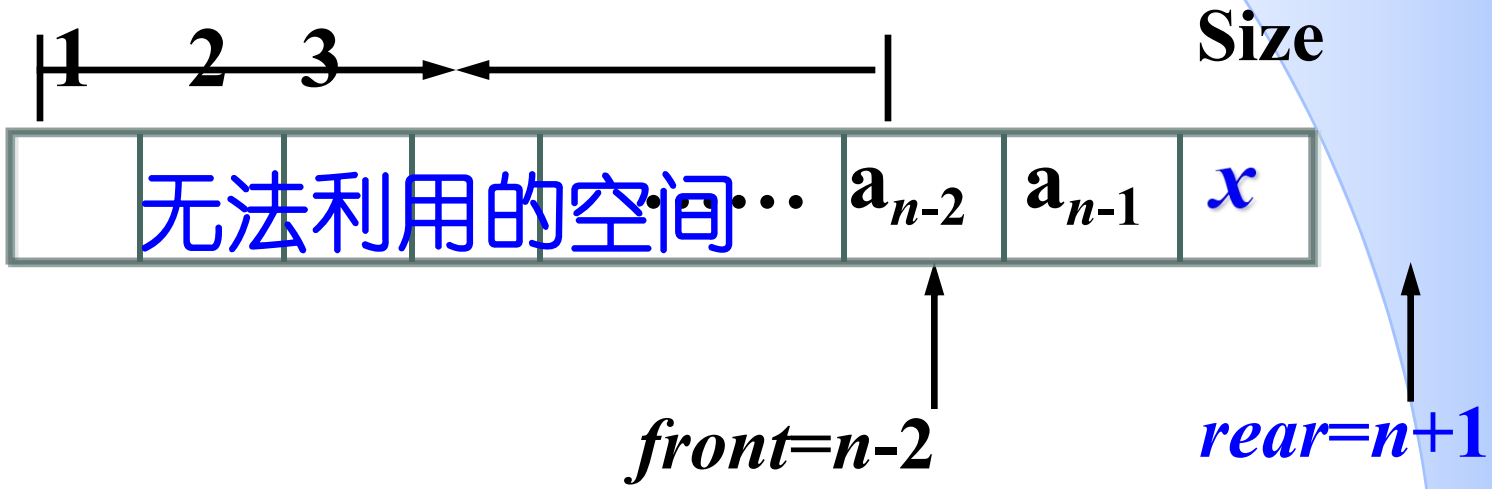
以出队操作为例，讨论出、入队方法：

出队方法1：令*front*=*front*+1

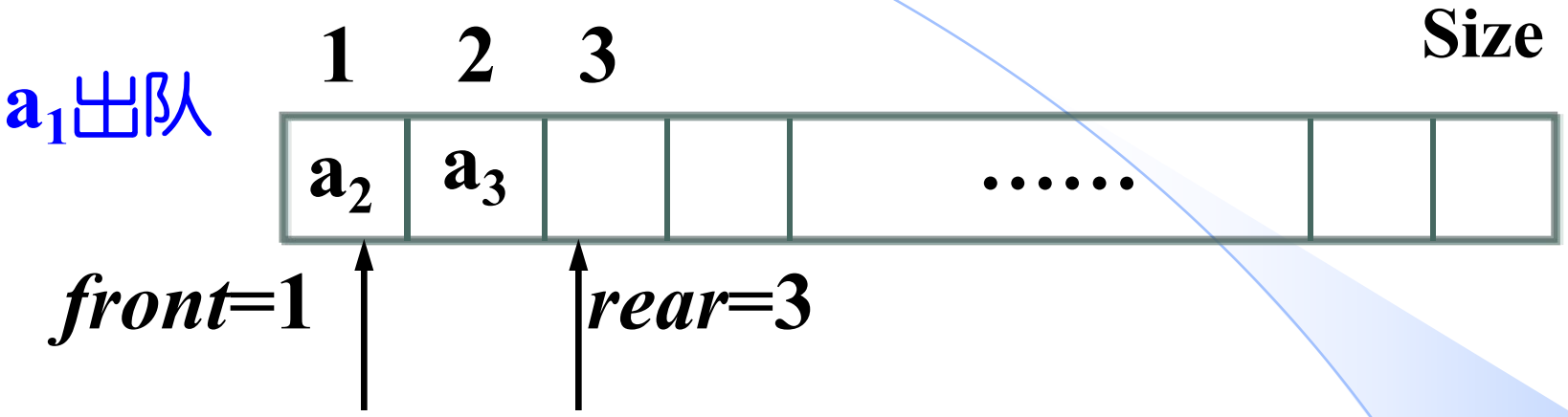


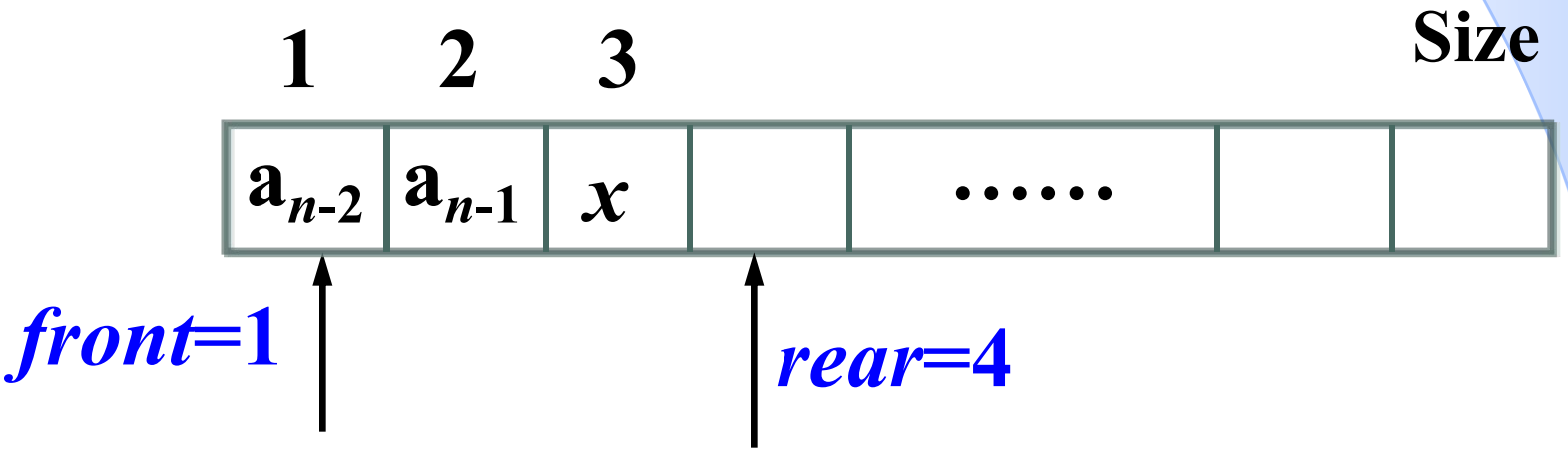



1 2 3
Size
a_2 a_3
出队
front=4
rear=4


1 2 3
Size
无法利用的空间
a_{n-2}
a_{n-1}
x
front=n-2
rear=n+1

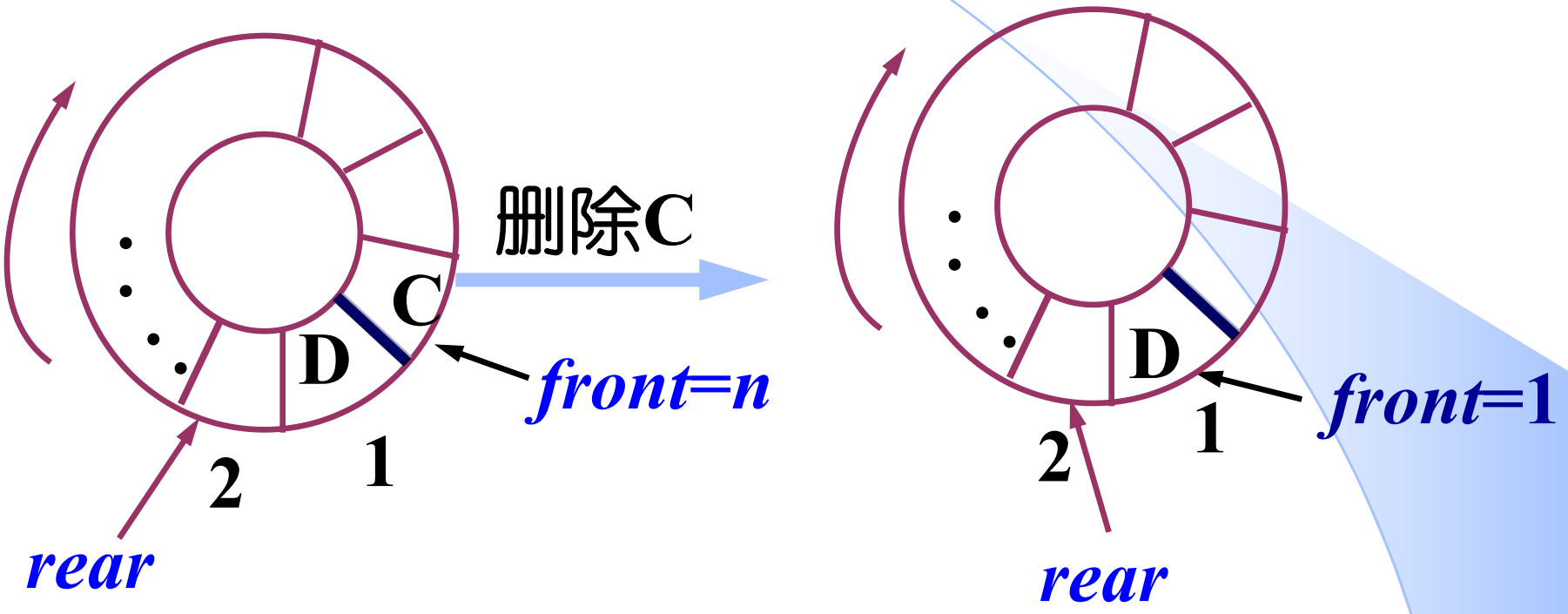
出队方法2：元素向前移动，*front*总等于1





删除队首元素的方法3：循环队列

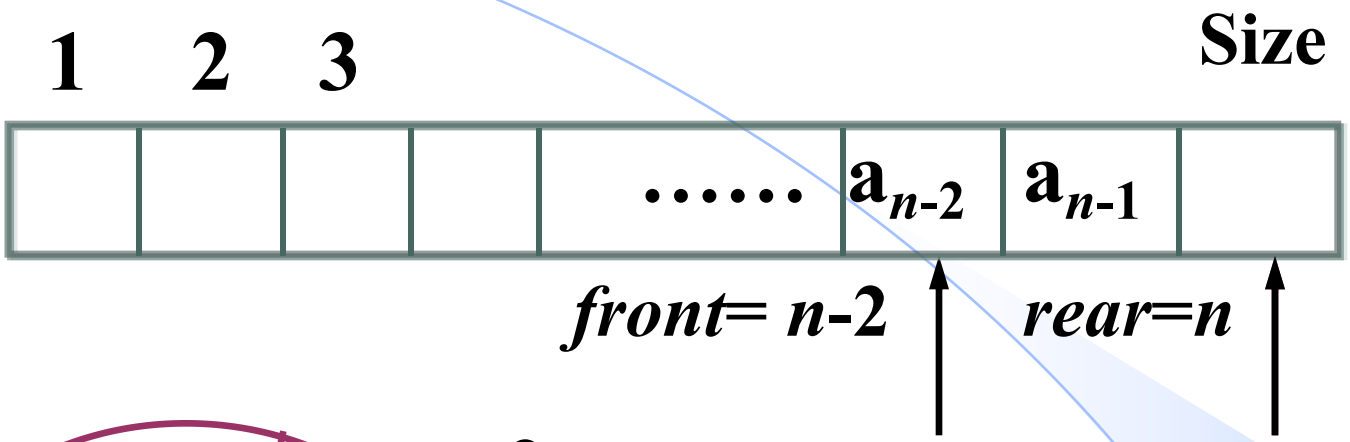
删除元素：*front*顺时针移动一位

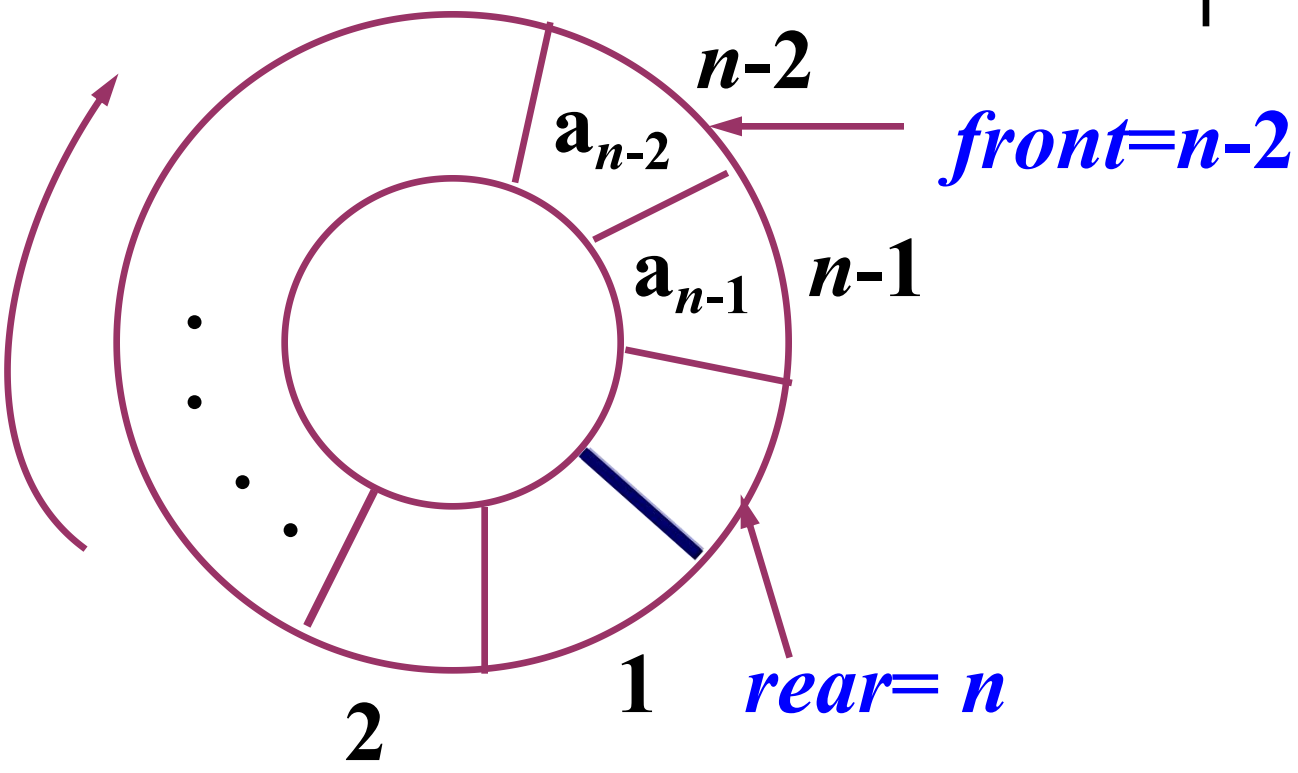


IF *front* <Size THEN　*front* ← *front* +1.

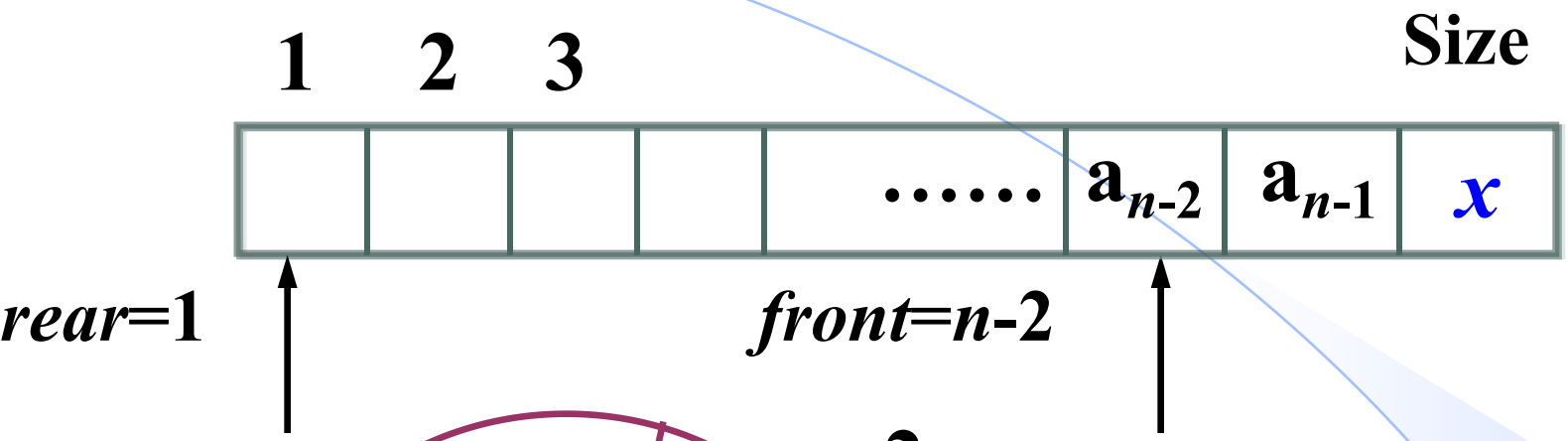
ELSE　*front* ← 1.

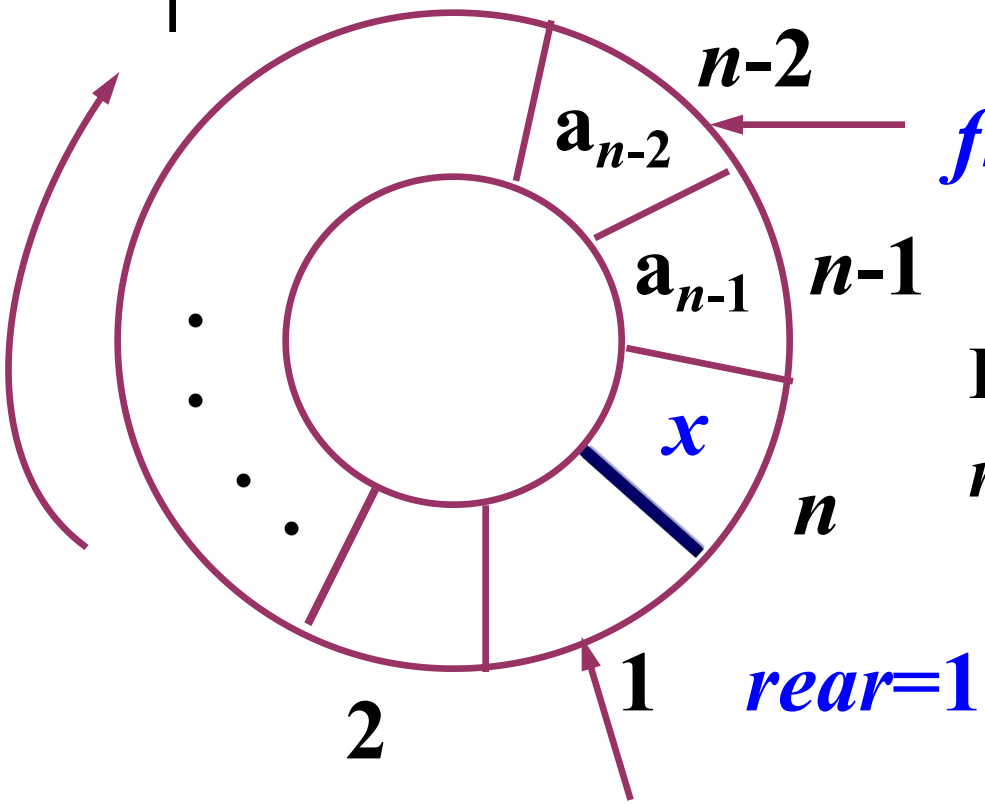
插入元素 *x*： *rear*顺时针移动一位





插入元素 *x*： *rear*顺时针移动一位





IF *rear* <Size THEN

rear ← *rear* +1.

ELSE *rear* ← 1.

采用环状模型来实现队列：

*front*指向队首位置，删除一个元素就将*front*顺时针移动一位；

- *rear*指向元素要插入的位置，插入一个元素就将rear顺时针移动一位；
- *count*存放队列中元素的个数，当*count*等于Size时，不可再向队列中插入元素。

队空：*count*=0

队满：*count*= Size

算法**QInsert** (A, *item*. A) // 将元素*item*插入队列A的队尾

QI1. [队列满？]

IF *count*=Size THEN (PRINT“队列已满无法插入”.
RETURN.)

QI2. [插入]

A[*rear*] ←*item*. // 将新元素插入队尾

QI3. [更新]

// 更新队尾下标

IF *rear*<Size THEN *rear* ← *rear* +1. ELSE *rear* ← 1.

// 更新队列长度

count←*count*+1.

算法**QDelete** (A, *item* . A)

// 删除队列A的队首元素，并将其元素值赋给变量*item*

QD1. [队列空？]

IF *count*=0 THEN (PRINT “队列空无法删除”.
RETURN.)

QD2. [出队]

item ←A[*front*]. // 将队首元素保存至*item*

QD3. [更新]

// 更新队首元素下标

IF *front*<Size THEN *front*←*front*+1. ELSE *front* ← 1.

// 更新队列长度

count←*count* –1. ▌

算法**QFront** (A. *item*)

// 读取队列A的队首元素值，并将其赋给变量*item*

QF1. [队列空？]

IF *count*=0 THEN (PRINT “队列空无法读取”.
RETURN.)

QF2. [存取]

item ←A[*front*]. ▌ // 将队首元素保存至*item*

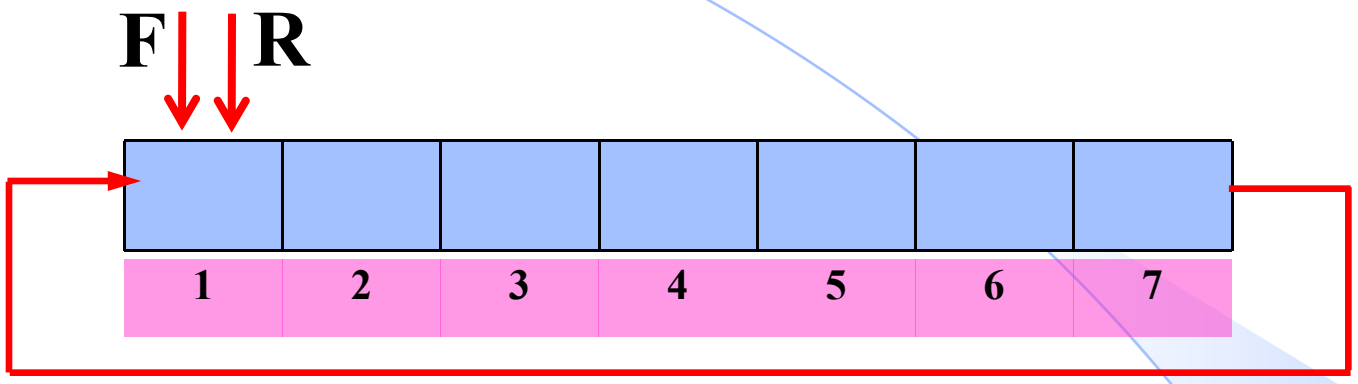
- 队列空的条件：

$$count = 0$$

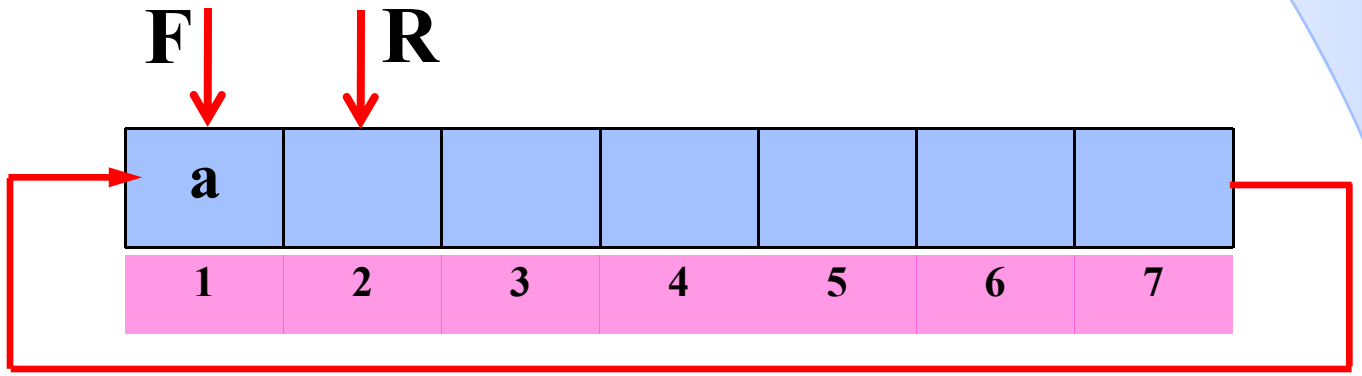
- 队列满的条件：

$$count = Size$$

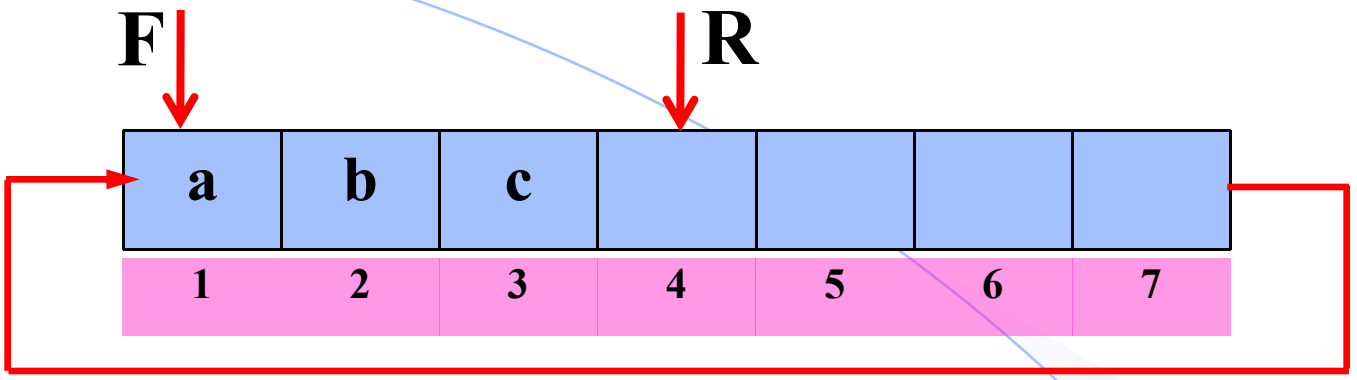
队列运行示意图



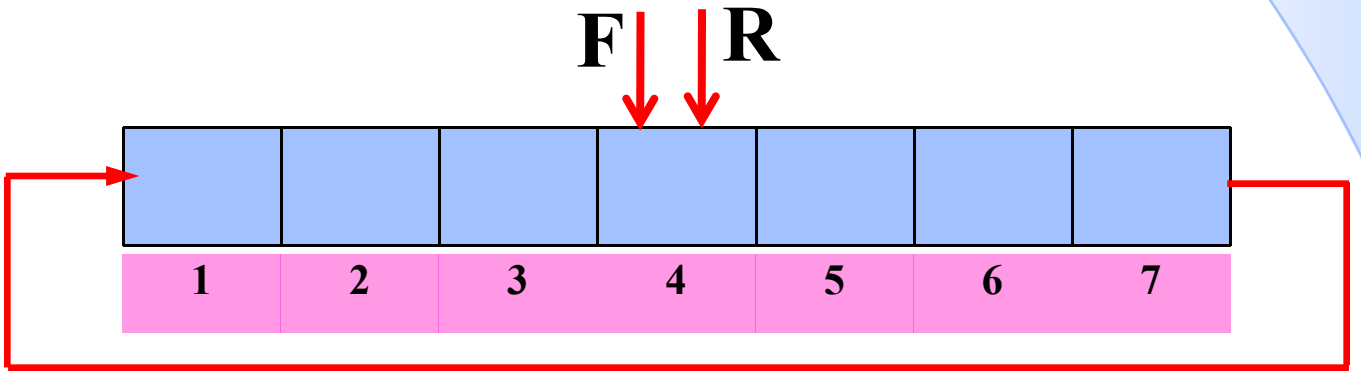
(a) 创建一个队列



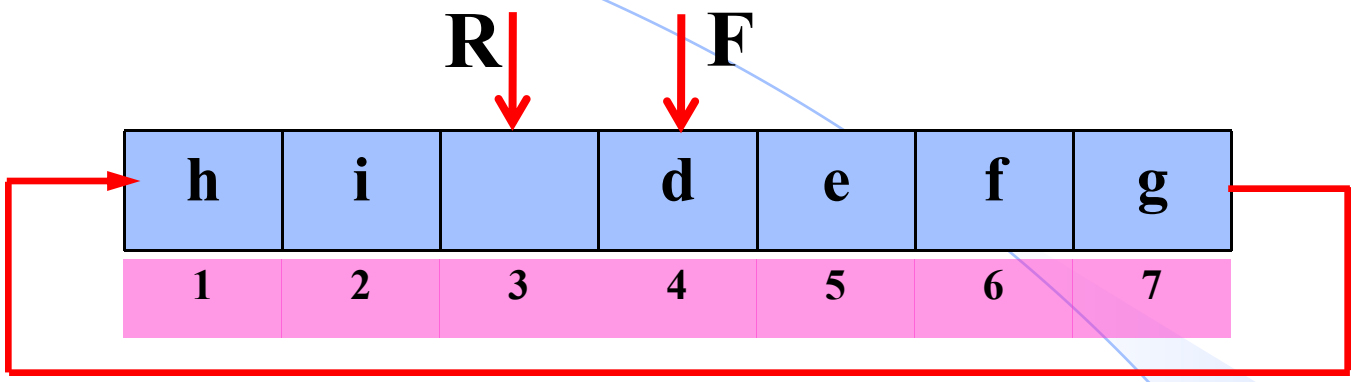
(b) 插入元素 a



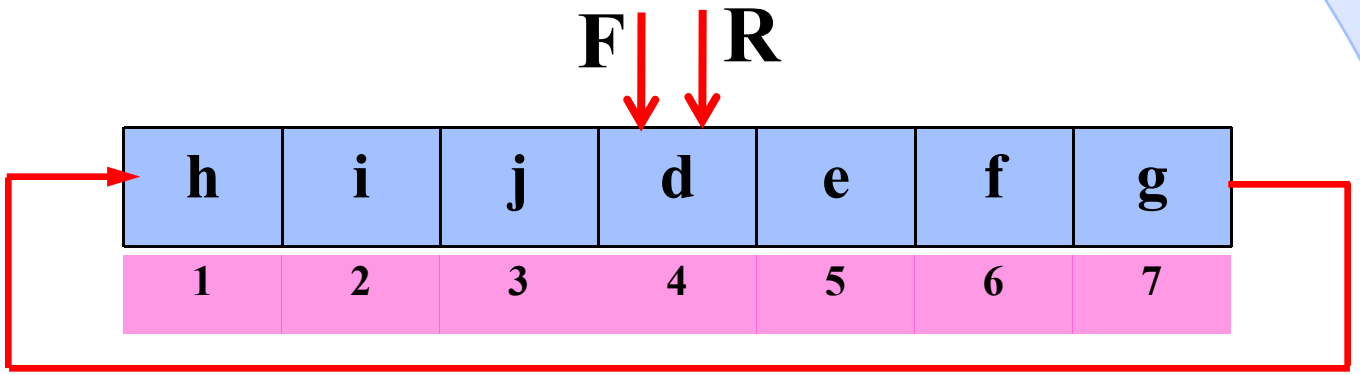
(c) 插入元素b、c



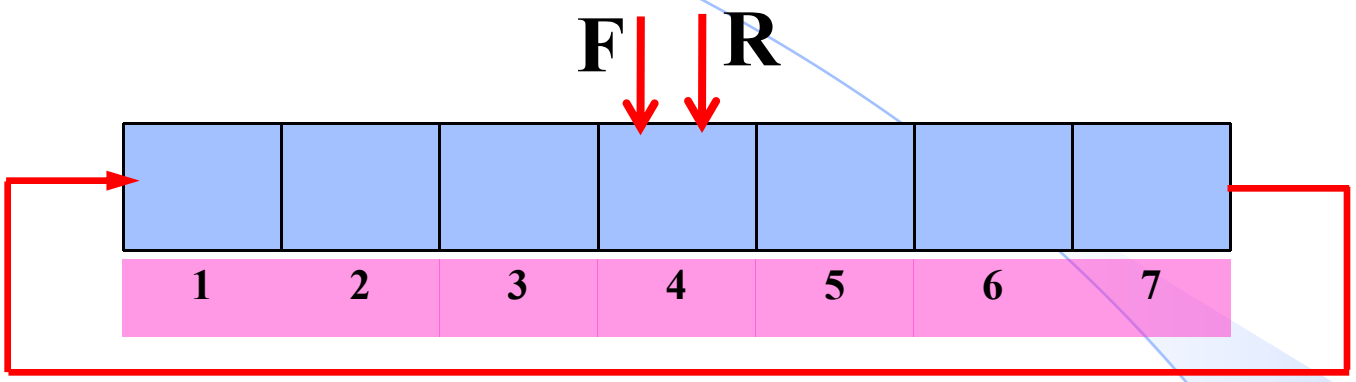
(d) 删除元素 a、b、c



(e) 插入元素d、e、f、g、h、i



(f) 插入元素 j



(g) 删除所有元素

3.6 队 列

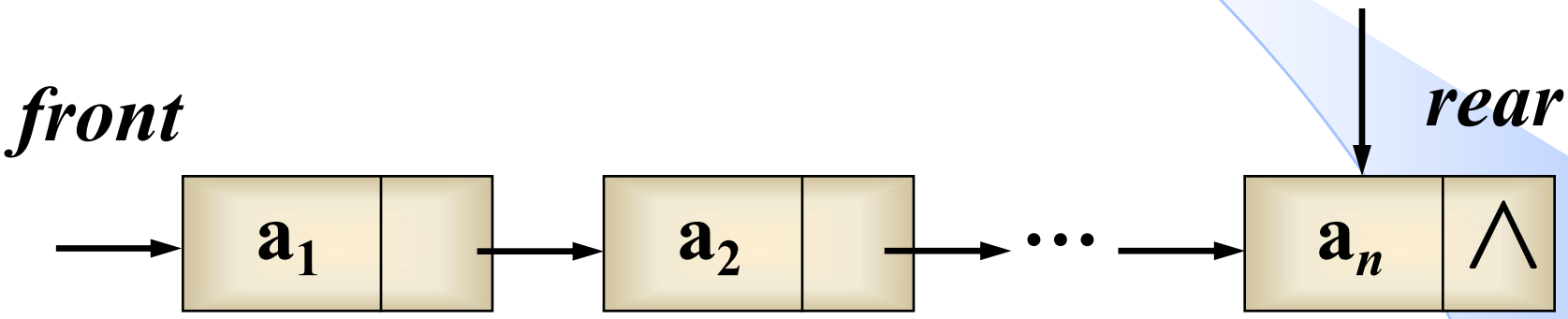
3.6.1 队列的定义和主要操作

3.6.2 顺序队列

3.6.3 链式队列

队列的链接存储

链式队列的结构： $(a_1, a_2, \ldots, a_n)$



算法**QInsert** (*item*. *front*, *rear*) // 将元素*item*插入队尾

QI1. [创建新结点]

s⇐ AVAIL. data(*s*)←*item*. next(*s*)←NULL. /* 为新结点申请空间，令其字段值为*item*，指针域为空*/

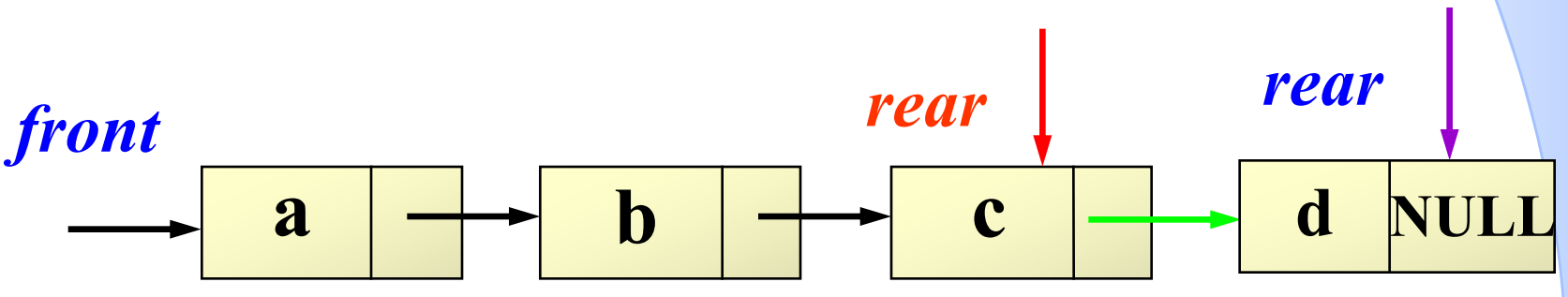
QI2. [插入]

IF *front*=NULL THEN *front*←*s*. //若队列空，令队首指针指向*s*

ELSE next(*rear*)←*s*. //若队列非空，令表尾结点的*next*指向*s*

QI3. [更新]

rear←*s*. ▌ // 更新表尾指针



算法**QDelete** (. *item*) // 删除队首结点并将其字段值存于*item*

QD1. [队列空？]

IF *front*=NULL THEN (PRINT “队列为空”. RETURN.)

QD2. [出队]

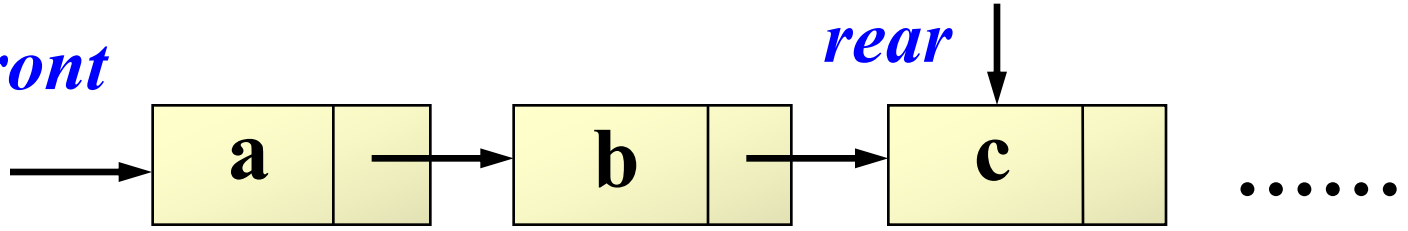
q←*front*. *item*←data(*q*). // 令指针*q*指向队首，并保存其字段值

front←next(*front*). // 令队首指针指向原队首结点之后继结点

AVAIL⇐*q*. // 释放原队首结点的存储空间

QD3. [出队后队列空？]

IF *front*=NULL THEN *rear*←NULL. /* 若删除队首结点后队列为空，则令队尾指针为空*/



算法**QDelete** (. *item*) // 删除队首结点并将其字段值存于*item*

QD1. [队列空？]

IF *front*=**NULL THEN (PRINT** “队列为空” **. RETURN.)**

QD2. [出队]

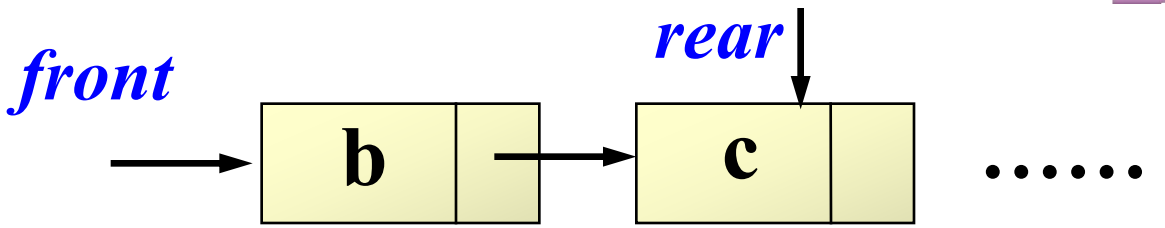
q←*front*. *item*←**data**(*q*). // 令指针*q*指向队首，并保存其字段值

front←**next**(*front*). // 令队首指针指向原队首结点之后继结点

AVAIL⇐*q*. // 释放原队首结点的存储空间

QD3. [出队后队列空？]

IF *front*=**NULL THEN** *rear*←**NULL.** /* 若删除队首结点后队列为空，则令队尾指针为空*/



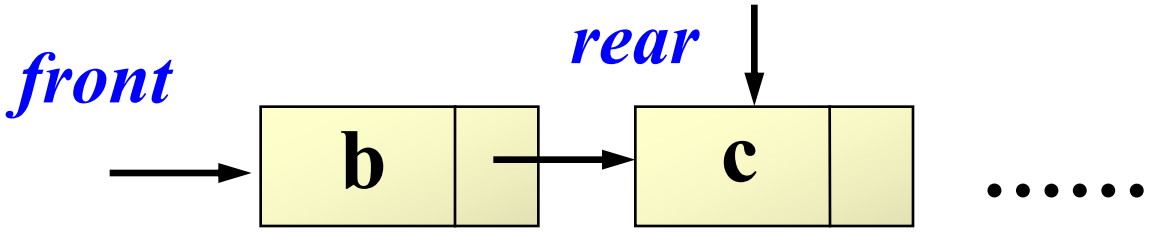
算法**QFront** (. *item*) // 读取队首元素值，并将其赋给变量*item*

QF1. [队列空？]

IF *front*=NULL THEN (PRINT “队列空无法读取”.
RETURN.)

QF2. [存取]

item ←data(*front*). ▌ // 将队首元素保存至*item*



双端队列：同样是一个运算受限的线性表，同样限制在表的端点处进行插入和删除运算，但是它允许插入和删除操作可以在表的任何一端进行。

循环队列和链式队列的比较

队列的应用：缓冲区、资源请求队列、优先级队列、日常排队。

堆栈和队列的共同点和区别？