# Algoritmi e Strutture di Dati I

Massimo Franceschet

# Pila (stack)

La struttura di dati **pila** rappresenta un insieme dinamico nel quale le operazioni di inserimento e cancellazione seguono una politica

**LIFO = Last In First Out**

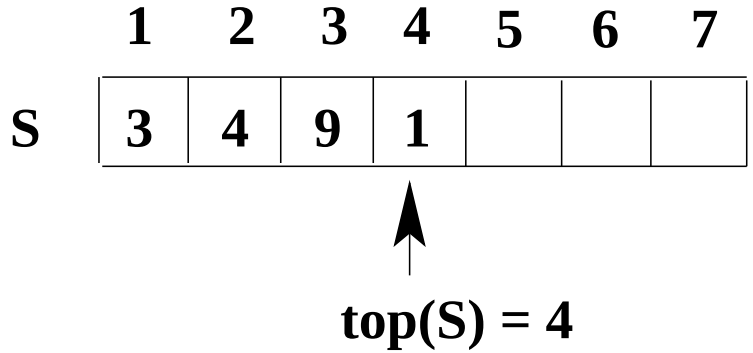
Ciò significa che l'oggetto cancellato è quello che è stato inserito **per ultimo**.

# Pila (stack)

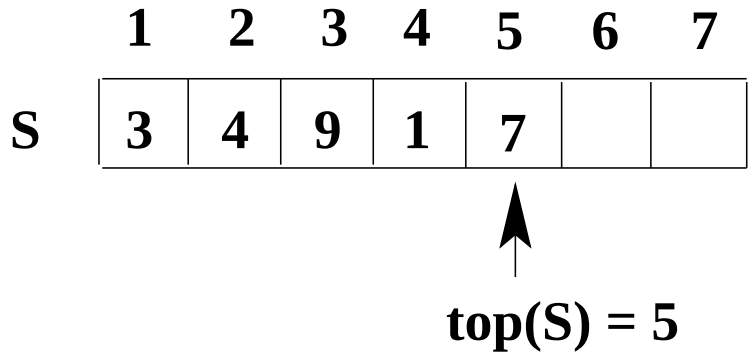
Una pila $S$ possiede un attributo $top(S)$ che punta alla cima della pila e le seguenti quattro operazioni:

- $StackEmpty(S)$ che controlla se la pila $S$ è vuota;
- $StackFull(S)$ che controlla se la pila $S$ è piena;
- $Push(S, x)$ che inserisce l'oggetto $x$ in cima alla pila $S$;
- $Pop(S)$ che rimuove l'oggetto che si trova in cima alla pila $S$;

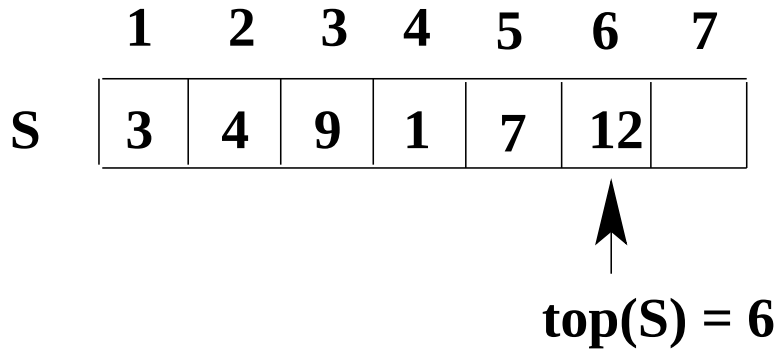
# Una pila $S$



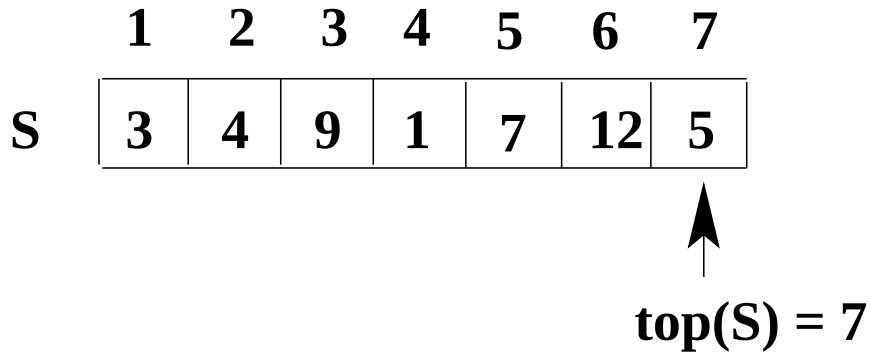
$Push(S, 7)$



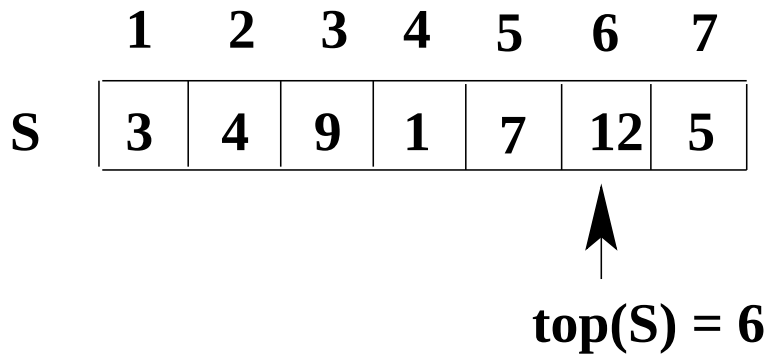
$Push(S, 12)$



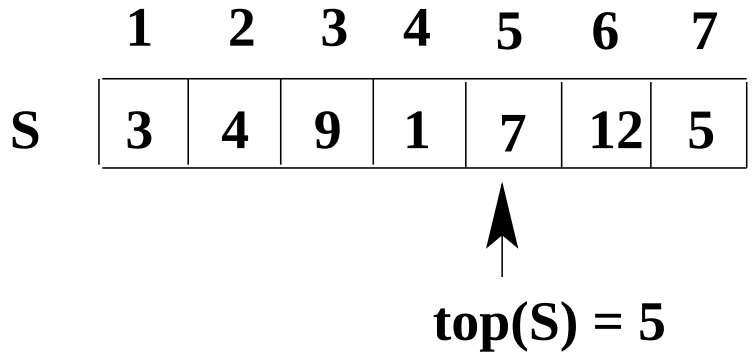
$Push(S, 5)$



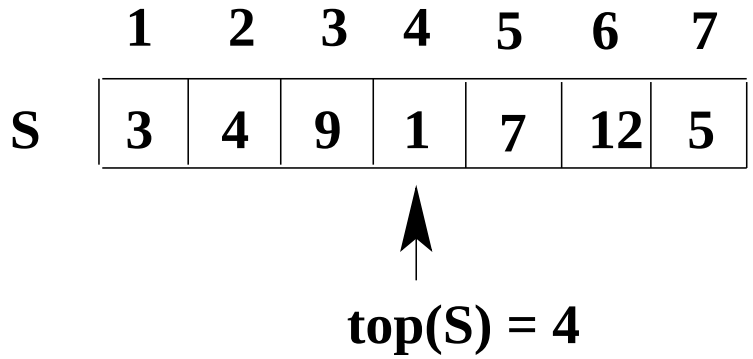
## $Pop(S)$



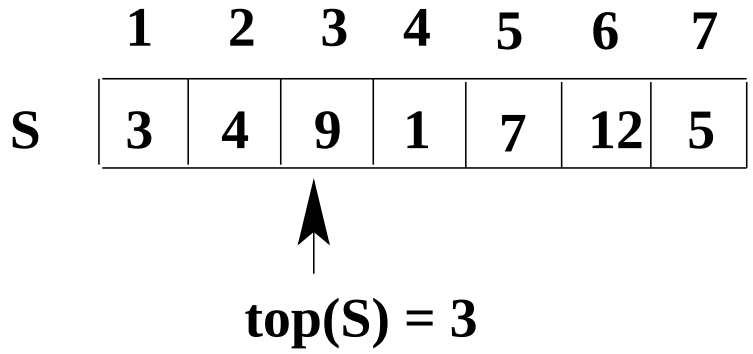
$Pop(S)$



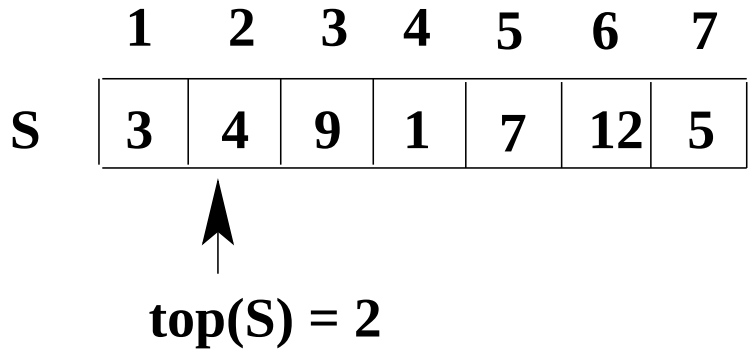
$Pop(S)$



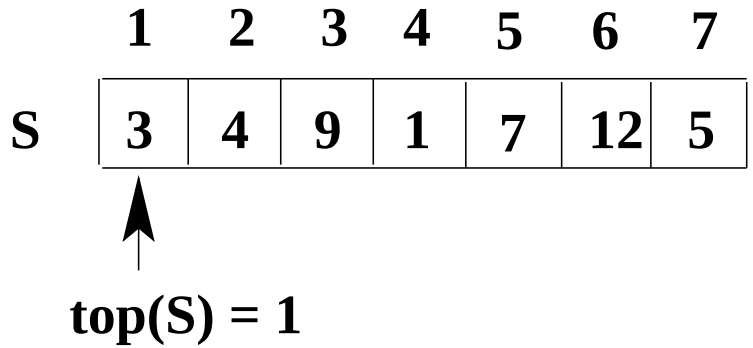
$Pop(S)$



$Pop(S)$



$Pop(S)$



$Pop(S)$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| **S** | **3** | **4** | **9** | **1** | **7** | **12** | **5** |

**top(S) = 0**

**StackEmpty($S$)**

```
1: if top(S) = 0 then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**StackFull($S$)**

```
1: if top(S) = length(S) then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**Push(S,x)**

```
1: if StackFull(S) then
2:     error OVERFLOW
3: else
4:     top(S) ← top(S) + 1
5:     S[top(S)] ← x
6: end if
```

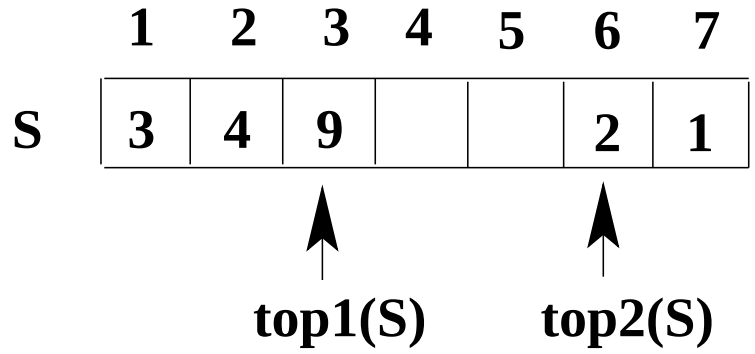
**Pop(S)**

```
1: if StackEmpty(S) then
2:     error UNDERFLOW
3: else
4:     x ← S[top[S]]
5:     top(S) ← top(S) − 1
6:     return x
7: end if
```

## Cosa fa?

**while not** $StackEmpty(R)$ **do**
  $Push(S, Pop(R))$
**end while**

**Esercizio** *Utilizzare un vettore di $n$ elementi per rappresentare due pile in maniera che nessuna delle due pile vada in overflow se il numero totale degli elementi delle due pile è inferiore a $n$.*



**StackEmpty1($S$)**

```
1: if top1(S) = 0 then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**StackEmpty2($S$)**

```
1: if top2(S) = length(S) + 1 then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**StackFull1**($S$)

```
1: if top1(S) = top2(S) − 1 then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**StackFull2**($S$)

```
1: if top1(S) = top2(S) − 1 then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**Push1(S,x)**

1: **if** $StackFull1(S)$ **then**
2: **error** OVERFLOW
3: **else**
4: $top1(S) \leftarrow top1(S) + 1$
5: $S[top1(S)] \leftarrow x$
6: **end if**

**Push2(S,x)**

1: **if** $StackFull2(S)$ **then**
2: **error** OVERFLOW
3: **else**
4: $top2(S) \leftarrow top2(S) - 1$
5: $S[top2(S)] \leftarrow x$
6: **end if**

**Pop1(S)**

1: **if** $StackEmpty1(S)$ **then**
2: **error** UNDERFLOW
3: **else**
4: $x \leftarrow S[top1[S]]$
5: $top1(S) \leftarrow top1(S) - 1$
6: **return** $x$
7: **end if**

**Pop2(S)**

1: **if** $StackEmpty2(S)$ **then**
2: **error** UNDERFLOW
3: **else**
4: $x \leftarrow S[top2[S]]$
5: $top2(S) \leftarrow top2(S) + 1$
6: **return** $x$
7: **end if**

# Coda (queue)

La struttura di dati **coda** rappresenta un insieme dinamico nel quale le operazioni di inserimento e cancellazione seguono una politica

**FIFO = First In First Out**

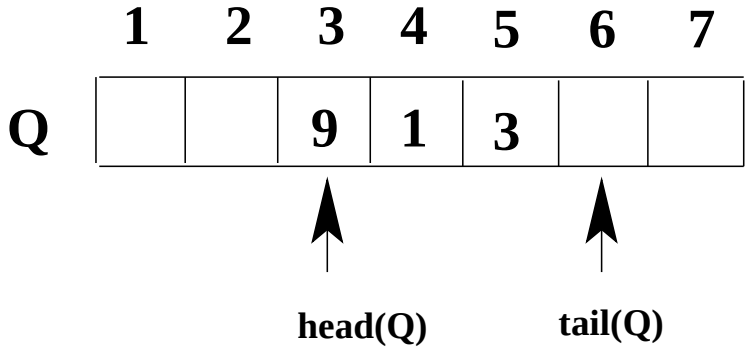
Ciò significa che l'oggetto cancellato è quello che è stato inserito **per primo**.

# Coda (queue)

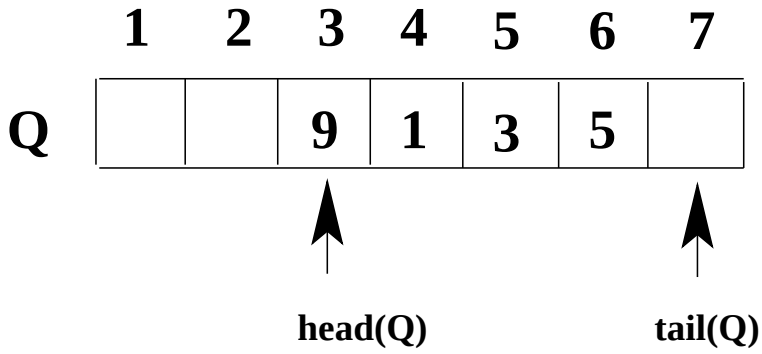
Una coda $Q$ possiede un attributo $head(Q)$ che contiene un puntatore all'**inizio** della coda (dove escono gli elementi), un attributo $tail(Q)$ che contiene un puntatore alla **fine** della coda (dove entrano gli elementi), e le seguenti quattro operazioni:

- $QueueEmpty(Q)$ che controlla se la coda $Q$ è vuota;
- $QueueFull(Q)$ che controlla se la coda $Q$ è piena;
- $Enqueue(Q, x)$ che inserisce l'oggetto $x$ alla fine della coda $Q$;
- $Dequeue(S)$ che rimuove l'oggetto all'inizio della coda $Q$.

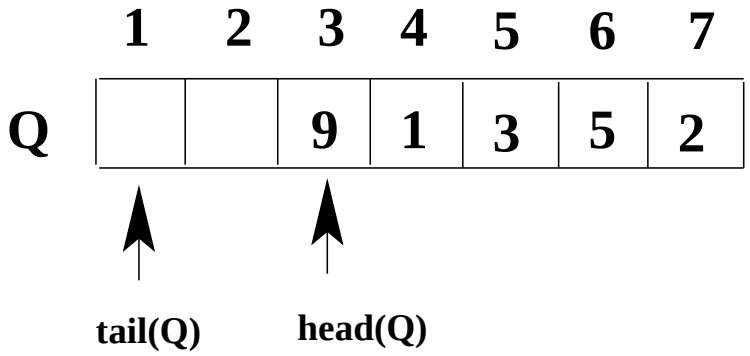
# Una coda $Q$



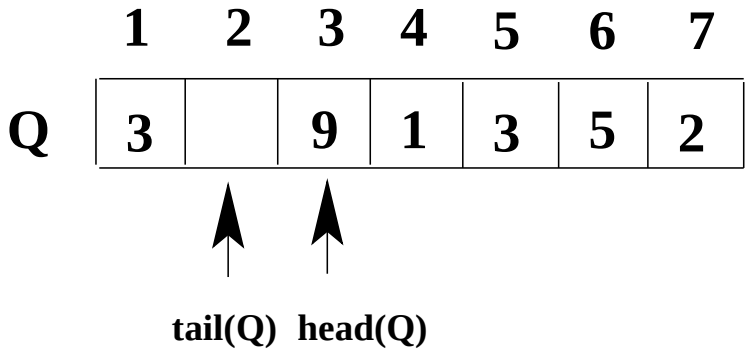
$Enqueue(Q, 5)$



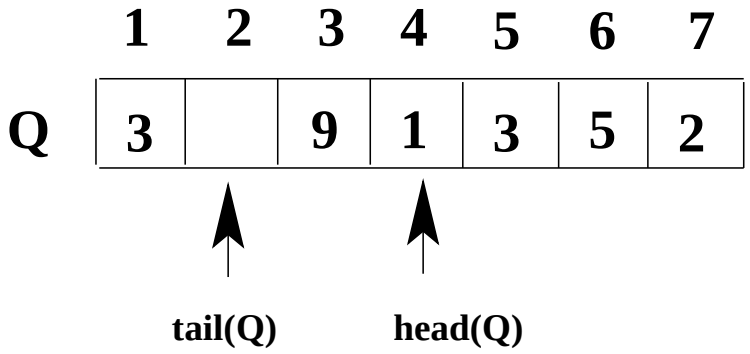
$Enqueue(Q, 2)$



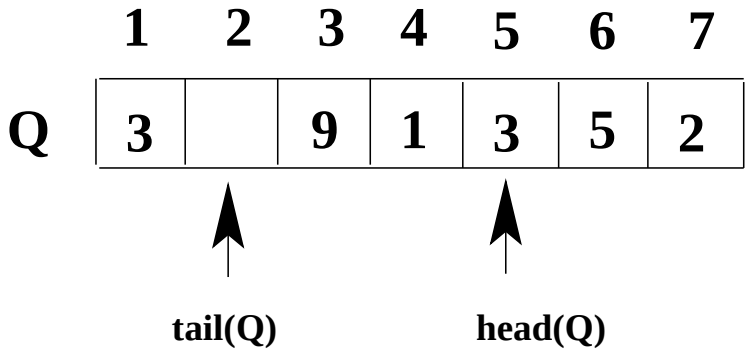
$Enqueue(Q, 3)$



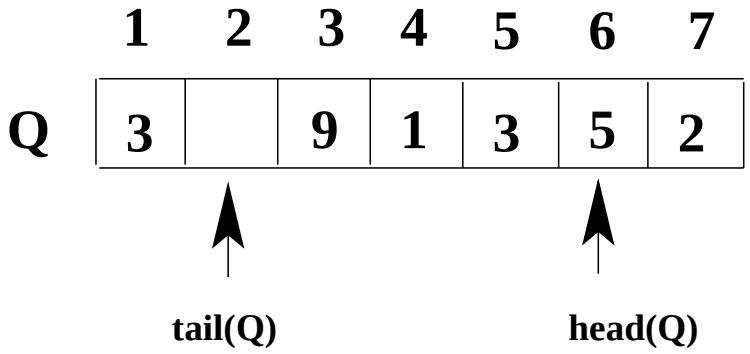
## $Dequeue(Q)$



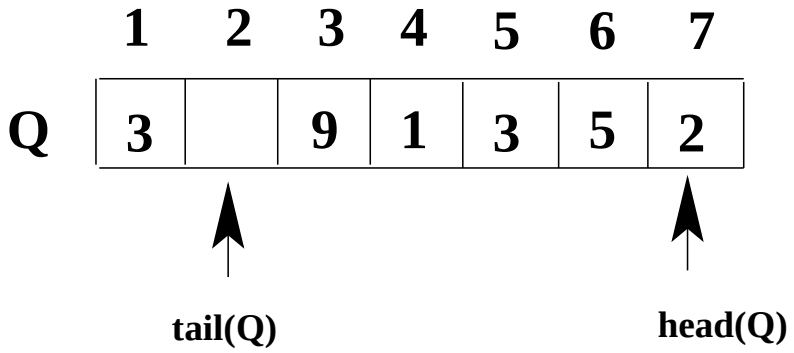
## $Dequeue(Q)$



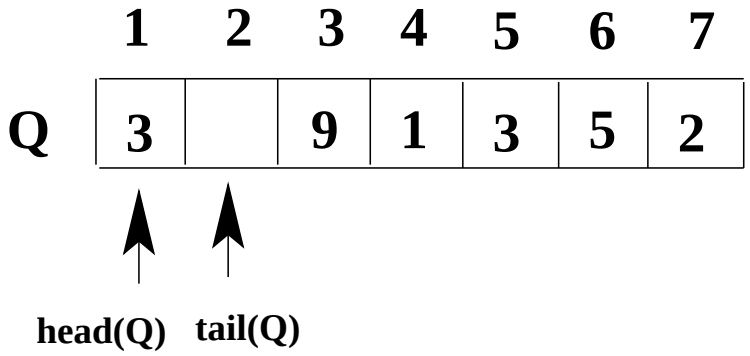
## $Dequeue(Q)$



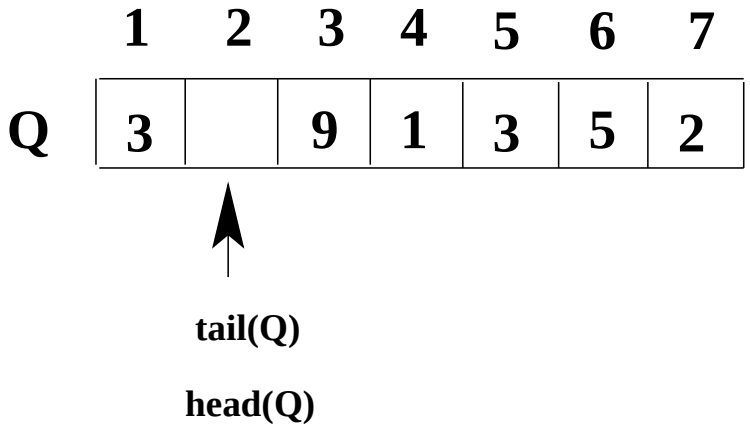
## $Dequeue(Q)$



## $Dequeue(Q)$



## $Dequeue(Q)$



**QueueEmpty($Q$)**

```
1: if head(Q) = tail(Q) then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**QueueFull($Q$)**

```
1: if ((tail(Q) + 1) mod length(Q)) = head(Q) then
2:     return TRUE
3: else
4:     return FALSE
5: end if
```

**Enqueue(Q,x)**

1: **if** $QueueFull(Q)$ **then**
2: **error** OVERFLOW
3: **else**
4: $Q[tail(Q)] \leftarrow x$
5: $tail(Q) \leftarrow (tail(Q) + 1)\ mod\ length(Q)$
6: **end if**

**Dequeue(Q)**

1: **if** $QueueEmpty(S)$ **then**
2: **error** UNDERFLOW
3: **else**
4: $x \leftarrow Q[head(Q)]$
5: $head(Q) \leftarrow (head(Q) + 1)\ mod\ length(Q)$
6: **return** x
7: **end if**

## Cosa fa?

**while not** $QueueEmpty(P)$ **do**
  $Enqueue(Q, Dequeue(P))$
**end while**

**Esercizio** *Sia $Q$ una coda. Si scriva una procedura $QueueLength(Q)$ che ritorna la lunghezza della coda $Q$.*

**Esercizio** *Sia $Q$ una coda. Si scriva una procedura $QueueLength(Q)$ che ritorna la lunghezza della coda $Q$.*

**QueueLength(Q)**

```
1: if head(Q) ≤ tail(Q) then
2:     return tail(Q) − head(Q)
3: else
4:     return length(Q) − head(Q) + tail(Q)
5: end if
```

**Esercizio** *Sia $Q$ una coda. Si scriva una procedura $QueueLength(Q)$ che ritorna la lunghezza della coda $Q$.*

**QueueLength(Q)**

1: **if** $head(Q) \leq tail(Q)$ **then**
2: **return** $tail(Q) - head(Q)$
3: **else**
4: **return** $length(Q) - head(Q) + tail(Q)$
5: **end if**

La complessità di $QueueLength(Q)$ è costante, cioè $\Theta(1)$.

**Esercizio** *Sia $Q$ una coda. Si scriva una procedura*

$$QueueReverse(Q)$$

*che inverte la coda $Q$ usando una pila e una procedura*

$$StackReverse(S)$$

*che inverte la pila $S$ usando una coda.*

**QueueReverse(Q)**

```
1: S ← ∅
2: while not QueueEmpty(Q) do
3:     Push(S, Dequeue(Q))
4: end while
5: while not StackEmpty(S) do
6:     Enqueue(Q, Pop(S))
7: end while
```

**StackReverse(S)**

```
1: Q ← emptyset
2: while not StackEmpty(S) do
3:     Enqueue(Q, Pop(S))
4: end while
5: while not QueueEmpty(Q) do
6:     Push(S, Dequeue(Q))
7: end while
```