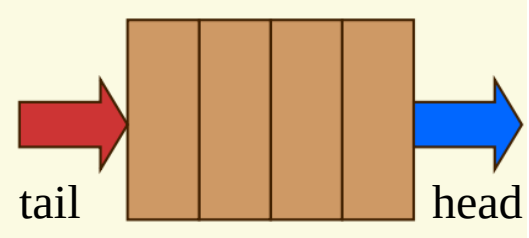
ADT *CODA (QUEUE)*

Il tipo di dati astratto *Coda (queue)*

- Una ***coda*** (*spesso chiamata anche* ***queue***) è una sequenza di elementi di un determinato tipo, in cui gli elementi si aggiungono da un lato (***tail***) e si tolgono dall'altro lato (***head***).



- Questo significa che la sequenza viene gestita con la modalità detta ***FIFO (First-in-first-out)*** cioè il primo elemento inserito nella sequenza sarà il primo ad essere eliminato.
- La coda è una struttura dati *lineare* a *dimensione variabile* in cui si può accedere direttamente solo alla testa (***head***) della lista.
- Non è possibile accedere ad un elemento diverso da ***head***, se non dopo aver eliminato tutti gli elementi che lo precedono (cioè quelli inseriti prima).

ADT *Queue:* Specifica sintattica

- **Tipo di riferimento**: queue
- **Tipi usati**: item, boolean

- **Operatori**
  - newQueue() → queue
  - emptyQueue(queue) → boolean
  - enqueue(item, queue) → queue
  - dequeue(queue) → item

ADT *Queue*: Specifica semantica

- **Tipo di riferimento queue**
  - queue è l'insieme delle sequenze $S=a1,a2,\ldots,an$ di tipo ***item***
  - L'insieme queue contiene inoltre un elemento ***nil*** che rappresenta la coda vuota (priva di elementi)

ADT *Queue*: Specifica semantica

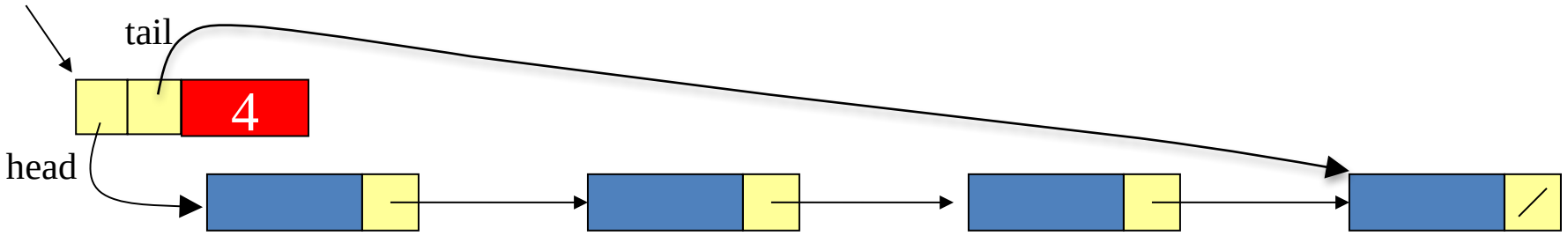
- **Operatori**
  - newQueue() → q
    - Post: q = nil
  - emptyQueue(q) → b
    - Post: se q=nil allora b = true altrimenti b = false
  - enqueue(e, q) → q'
    - Post: se q = nil allora q' = <e> altrimenti
      se q = $<a_1, a_2, \ldots a_n>$ con n > 0 allora q' = $<a_1, a_2, \ldots a_n, e>$
  - dequeue(q) → a
    - Pre: q = $<a_1, a_2, \ldots, a_{n-1}, a_n>$ n>0 (q ≠ nil)
    - Post: a = a_1 e l'elemento a_1 viene rimosso da q

Implementare il tipo astratto *Queue*

- Tra le possibili implementazioni, le più usate sono realizzate tramite:
  - **Lista concatenata**
  - **Array**

Implementazione della queue con liste collegate

- A differenza dello stack, per gestire la politica FIFO conviene avere accesso sia al primo elemento (estrazione) sia all'ultimo (inserimento)
- Il tipo queue è definito come un puntatore ad una struct che contiene
  - Un intero numelem che indica il numero di elementi della coda
  - Un puntatore head ad uno struct nodo
  - Un puntatore tail ad uno struct nodo



Implementazione di *Queue*: header file queue.h

```
// file queue.h

typedef struct c_queue *queue;

// prototipi

queue newQueue(void);

int emptyQueue(queue q);

item dequeue(queue q);

int enqueue(item val, queue q);
```

L'ADT queue è realizzato in modo da non dipendere dal tipo degli elementi contenuti.

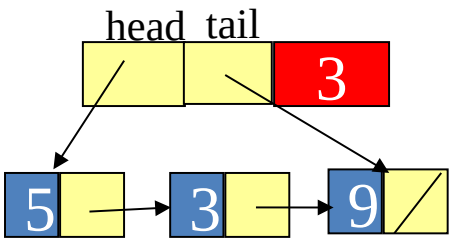
Utilizza il tipo generico item già visto in precedenza

dequeue toglie e restituisce l'elemento in testa alla coda

enqueue restituisce un intero che indica l'esito dell'operazione

file queue.c (versione con lista collegata)

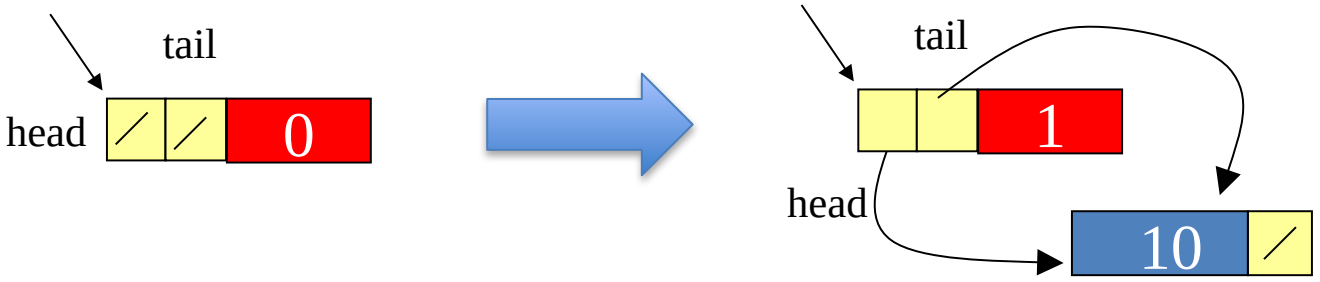
```
#include <stdio.h>
#include <stdlib.h>
#include "item.h"
#include "queue.h"

struct node {
    item value;
    struct node *next;
};

struct c_queue {
    struct node *head,*tail;
    int numel;
};
```



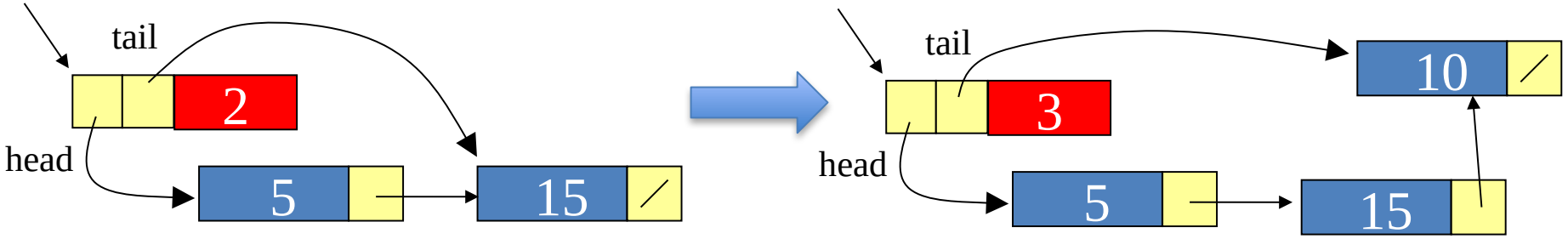

```
queue newQueue(void)
{
    struct c_queue *q;
    q = malloc (sizeof(struct c_queue));
    if (q == NULL)
        return NULL;

    q->numel = 0;
    q->head = NULL;
    q->tail = NULL;
    return q;
}

int emptyQueue(queue q)
{
    if (q==NULL)
        return -1;
    return q->numel == 0;
}
```

Inserire un item in coda

- Dobbiamo innanzitutto creare un nuovo nodo a cui dovrà puntare il puntatore tail
- Poi bisogna distinguere il caso in cui la coda di input è vuota e il caso in cui non è vuota
  - Coda vuota: il puntatore head dovrà puntare al nuovo nodo



  - Coda non vuota: il puntatore next dell'ultimo nodo dovrà puntare a nuovo



file queue.c (versione con lista collegata)

```
int enqueue(item val, queue q)
{
   if (q==NULL)
         return -1;

   struct node *nuovo;
   nuovo = malloc (sizeof(struct node));
   if (nuovo == NULL) return 0;

   nuovo->value = val;
   nuovo->next= NULL;

   if(q->head==NULL)
       q->head = nuovo;          //  caso coda vuota
   else
       q->tail->next= nuovo;  //  caso coda non vuota

   q->tail = nuovo;                 // tail punta al nuovo nodo
   (q->numel)++;                    // incrementare il numero di elementi
   return 1;
}
```

Rimuovere elemento da coda

- Bisogna prima salvare il puntatore al nodo da eliminare (quello puntato da head)
- Head dovrà quindi puntare al successivo
- A questo punto si può deallocare la memoria del nodo da rimuovere
- Se la coda aveva un solo elemento, ora è vuota, per cui bisogna porre anche il puntatore tail a NULL

file queue.c (versione con lista collegata)

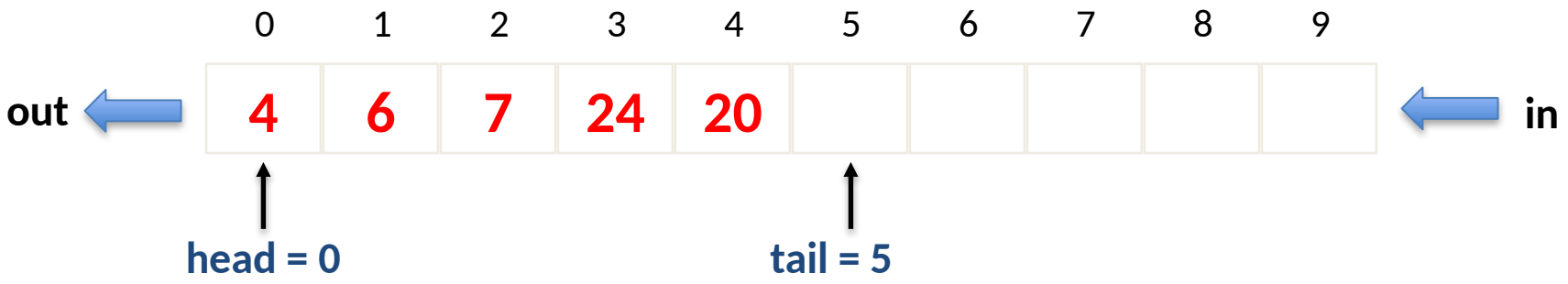
```
item dequeue(queue q)
{
    if (q==NULL) return NULLITEM;

    if (q->numel == 0) return NULLITEM;    //  coda vuota

    item result = q->head->value;   //  item da restituire

    struct node *temp = q->head;   //  nodo da rimuovere

    q->head = q->head->next;       // q->head avanza
    free(temp);                    // liberiamo memoria nodo da rimuovere

    if(q->head==NULL)   // se la coda conteneva un solo elemento
        q->tail=NULL;

    (q->numel)--;

    return result;
}
```

ADT *QUEUE (CODA)*

Altre implementazioni

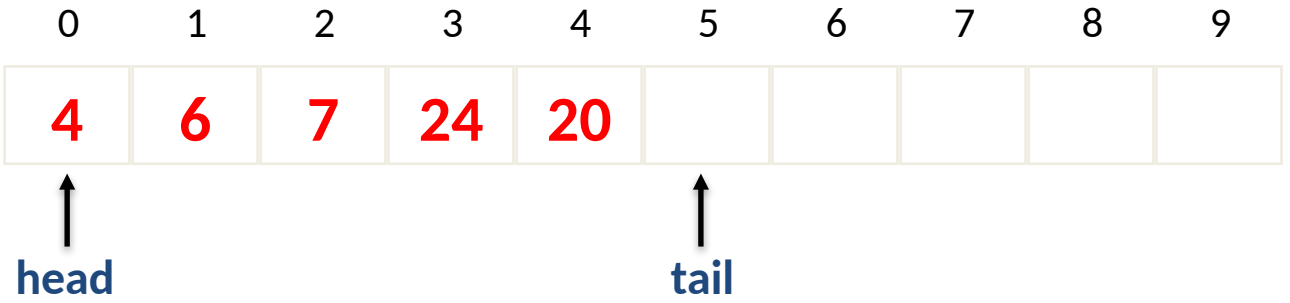
Implementazione semplice di queue con array

- La coda è implementata come un puntatore ad una struct c_queue che contiene due elementi:
  - Un array di MAXQUEUE elementi
  - Un intero che indica la posizione *head* della coda
  - Un intero che indica la posizione *tail* della coda
- Quando la coda si riempie, non è possibile eseguire l'operazione enqueue …

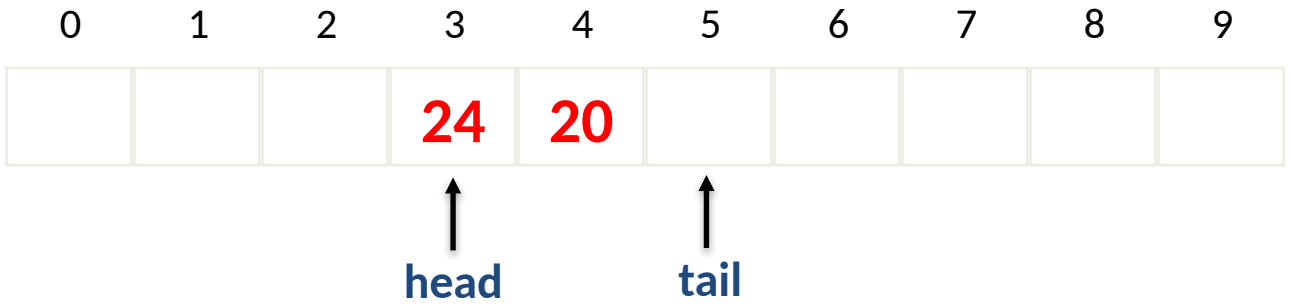


queue rappresentata con array

- Se l'array viene gestito normalmente, cioè mantenendo head <= tail, ci sono dei problemi…
- Esempio:



- Se rimuoviamo uno alla volta i primi tre elementi in coda otteniamo:



- Risultano disponibili solo le posizioni a destra di tail, ma sono libere anche quelle a sinistra di head

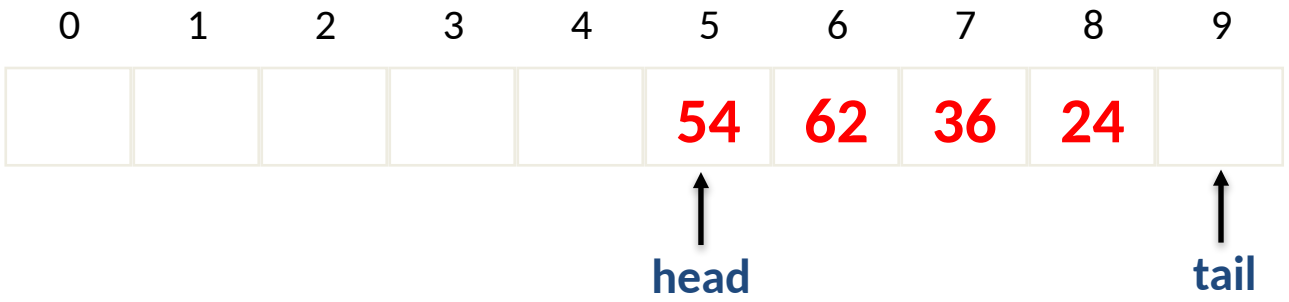
Queue rappresentata con array

- Prima soluzione: ad ogni rimozione si compatta l'array nelle posizioni iniziali, con uno shift degli elementi
  - **TROPPO COSTOSO!**

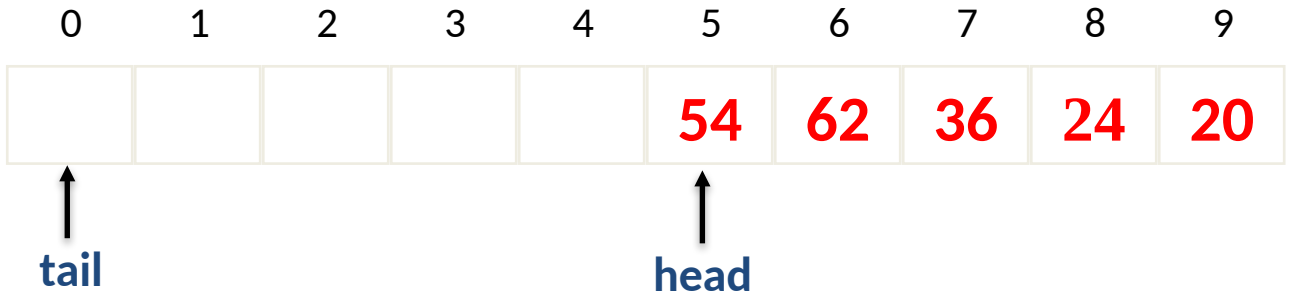
- Seconda soluzione: si gestisce l'array in modo circolare
  - In ogni istante, gli elementi della coda si trovano nel segmento head, head+1,… tail-1
  - … ma non necessariamente head <= tail
  - Infatti, dopo aver inserito in posizione N-1 (ultima posizione dell'array), se c'è ancora spazio in coda, si inseriscono ulteriori elementi a partire dalla posizione 0 (prima posizione dell'array)
  - In questo modo si riesce a garantire che ad ogni istante la coda abbia capacità massima di N-1 elementi

Queue rappresentata con array

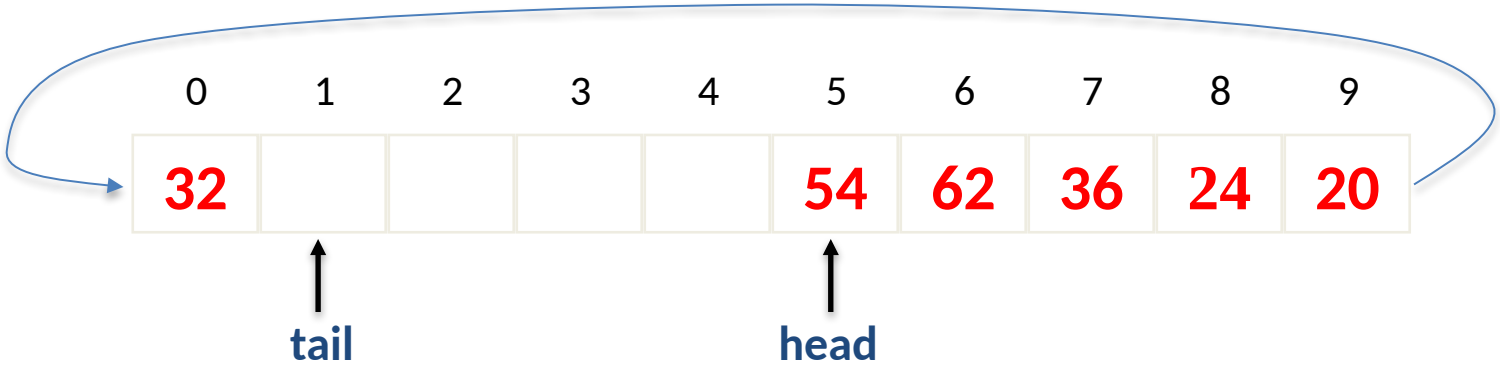
- Supponiamo di voler inserire 20 e 32 in questa coda



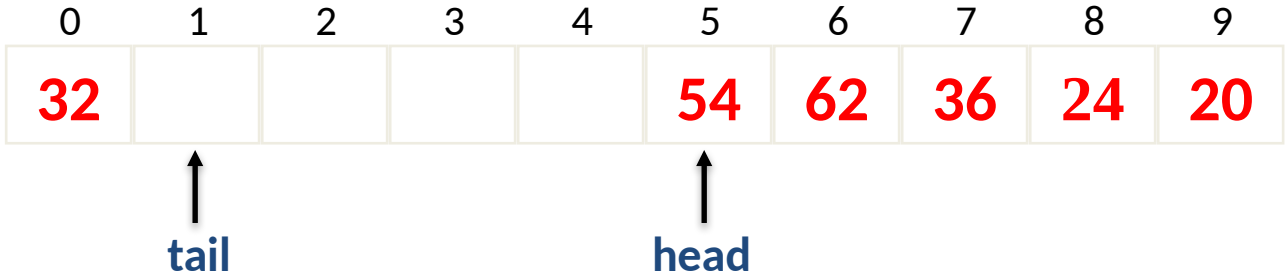
- Inseriamo 20 …



- Adesso inseriamo 32 …



Queue rappresentata con array



- Adesso tail < head, perché la posizione 0 segue la posizione N-1
- In questo ordine circolare il successore di p è (p + 1) % N
  - Ogni volta che si inserisce un elemento tail avanza: tail = (tail + 1) % N
  - Ogni volta che si rimuove un elemento head avanza: head = (head + 1) % N
- La coda è piena (e non si può eseguire enqueue) se il successore di tail in questo ordine circolare è head
  - (tail + 1) % n == head La condizione comporta una locazione vuota necessaria a distinguere la condizione di buffer vuoto da quella di pieno
- La coda ha un solo elemento se tail == (head +1)%N
  - eseguendo dequeue la coda si svuota
- Quando la coda è vuota, i valori di head e tail coincidono

Implementazione di *Queue* con array:

header file queue.h

```
// file queue.h

typedef struct c_queue *queue;

// prototipi

queue newQueue(void);

int emptyQueue(queue q);

item dequeue(queue q);

int enqueue(item val, queue q);
```

L'header file non cambia rispetto alla versione con le liste collegate

Gli operatori hanno la stessa interfaccia

file queue.c (versione con uso di array)

```
#include <stdio.h>
#include <stdlib.h>
#include “item.h”
#include “queue.h”

# define MAXQUEUE 100   // dimensione massima di default

struct c_queue {
     item *vet;
     int size;                // dimensione della coda
     int head, tail;

};

// Continua su prossime slide
```

file queue.c

```
queue newQueue(void)
{
    struct c_queue *q = malloc(sizeof(struct c_queue));

    if (q == NULL) return NULL;
    q->vet = malloc(MAXQUEUE * sizeof(item));

    if (q->vet == NULL)  {
            free (q);
            return NULL;
    }
    q->size = 0;        // dimensione massima di default
    q->head = 0;
    q->tail = 0;
    return q;
}

int emptyQueue(queue q)
{
     if(q==NULL) return -1;
     return (q->size == 0);
}
```

file queue.c

```
int enqueue(item val, queue q)
{
    if(q==NULL) return -1;

    if (q->size == MAXQUEUE-1)       //coda piena
            return 0;
    (q->size)++;
    q->vet[q->tail]=val;                    // inserisco in coda
    q->tail= (q->tail + 1)%MAXQUEUE;
    return 1;
}

item dequeue(queue q)
{
     if(q ==NULL || q->size == 0) return NULLITEM;

     item result = q->vet[q->head];          // item da restituire
     q->head = (q->head + 1) % MAXQUEUE;  // operatore % per gestione circolare
     return result;
}
```

Considerazioni

- Abbiamo visto due implementazioni diverse dell'ADT coda.
  - lista
    - **pro**: è una implementazione espandibile (unico limite è la capacità della memoria)
    - **contro**: la struttura è più complessa
  - vettore circolare
    - **pro**: gli elementi sono memorizzati in modo contiguo e la struttura è più semplice
    - **contro**: dimensione fissata, bisogna conoscere a priori il numero massimo di elementi che la coda deve contenere, parte dello spazio è inutilizzato
  - Note su vettore circolare
    - Potrei usare la realloc per ridimensionare la coda (come fatto per lo stack) e poter quindi sempre inserire elementi ?
    - Sì, ma devo considerare che l'ordine degli elementi della coda nell'array non necessariamente va dalla posizione 0 alla posizione n-1 …
    - Farlo come esercizio …