

Algoritmi e Strutture di Dati

Capitolo 4 - Strutture di dati elementari

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

◆Struttura di dati

- ◆Dinamica e lineare

- ◆Contenente elementi generici (Item), potenzialmente anche duplicati

- ◆Ordine all'interno della sequenza è importante

◆Interfaccia

- ◆E' possibile aggiungere / togliere elementi, specificando la posizione

 - ◆ $S = s_1, s_2, \dots, s_n$

 - ◆L'elemento s_i è in posizione pos_i

 - ◆Esistono le posizioni (fittizie) pos_0, pos_{n+1}

- ◆E' possibile accedere *direttamente* ad alcuni elementi (testa / coda)

- ◆E' possibile accedere *sequenzialmente* a tutti gli altri elementi



◆ **Struttura dati “generale”:** *insieme dinamico*

- ◆ **Può crescere, contrarsi, cambiare contenuto**
- ◆ **Operazioni base: inserimento, cancellazione, verifica contenimento**
- ◆ **Il tipo di insieme (= struttura) dipende dalle operazioni**

◆ **Elementi**

- ◆ **Elemento: oggetto “puntato” da un riferimento/puntatore**
- ◆ **Composto da:**
 - ◆ **campo chiave di identificazione**
 - ◆ **dati satellite**
 - ◆ **campi che fanno riferimento ad altri elementi dell'insieme**



◆ Il dizionario rappresenta il concetto matematico di relazione univoca

◆ Relazione $R : D \rightarrow C$

◆ Insieme D è il dominio (elementi detti chiavi)

◆ Insieme C è il codominio (elementi detti valori)

◆ Associazione chiave-valore

◆ Operazioni ammesse:

◆ ottenere il valore associato ad una particolare chiave (se presente), o nil

◆ inserire una nuova associazione chiave- valore, cancellando eventuali associazioni precedenti;

◆ rimuovere un'associazione chiave-valore esistente



◆ Realizzazione di alcuni tipi di dato:

- ◆ Sequenza \leftrightarrow lista
- ◆ Dizionario \leftrightarrow lista
- ◆ Insieme \leftrightarrow lista

◆ Realizzazione alternativa

- ◆ Vettori
- ◆ Alberi Bilanciati
- ◆ Tabelle Hash

◆ La scelta della struttura di dati ha riflessi sull'efficienza e sulle operazioni ammesse



♦ Liste (List, Linked List)

- ♦ Una sequenza di nodi (record), contenenti dati arbitrari e 1-2 puntatori all'elemento successivo e/o precedente
- ♦ Contiguità nella lista \nRightarrow contiguità nella memoria

♦ Realizzazione possibili

- ♦ *Bidirezionale* / monodirezionale
- ♦ *Con sentinella* / senza sentinella
- ♦ *Circolare* / non circolare



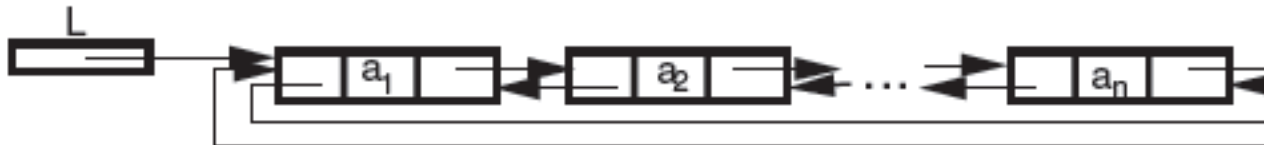
Linked List - esempi



monodirezionale



bidirezionale



bidirezionale circolare



monodirezionale con sentinella

Linked List – bidirezionale circolare con sentinella



List - circolare, bidirezionale, con sentinella

LIST

LIST *pred* % Predecessore

LIST *succ* % Successore

ITEM *value* % Elemento

LIST List()

```
LIST t ← new LIST
t.pred ← t
t.succ ← t
return t
```

boolean isEmpty()

```
return pred = succ = this
```

Pos head()

```
return succ
```

Pos tail()

```
return pred
```

Pos next(Pos *p*)

```
return p.succ
```

Pos prev(Pos *p*)

```
return p.pred
```

boolean finished(Pos *p*)

```
return (p = this)
```

ITEM read(Pos *p*)

```
return p.value
```

write(Pos *p*, ITEM *v*)

```
p.value ← v
```

Pos insert(Pos *p*, ITEM *v*)

```
LIST t ← List()
t.value ← v
t.pred ← p.pred
p.pred.succ ← t
t.succ ← p
p.pred ← t
return t;
```

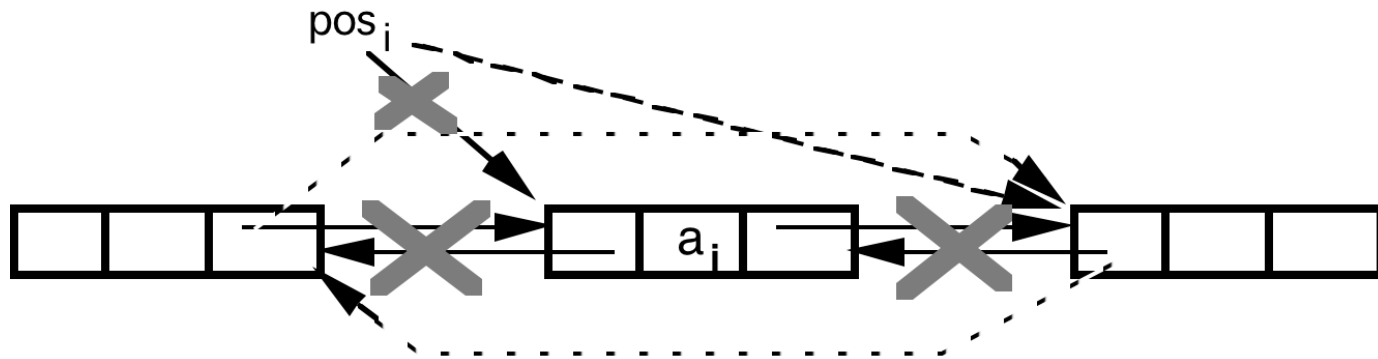
Pos remove(Pos *p*)

```
p.pred.succ ← p.succ
p.succ.pred ← p.pred
LIST t ← p.succ
delete p
return t;
```

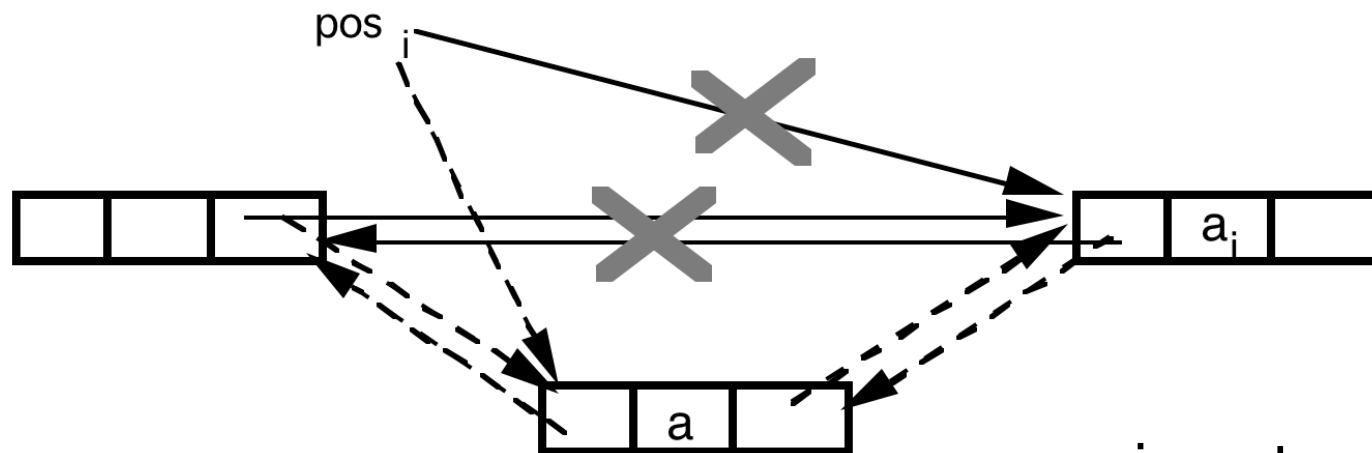
Nota:
LIST e POS
sono tipi
equivalenti



List – cancellazione e inserimento



remove



insert

Esempio List – Il catalogo di Don Giovanni

Cancellare dal *catalogo* tutte le duchesse e copiare in un'altra lista *L* tutte le spagnole:

```
donGiovanni(LIST catalogo)
  LIST L ← List()
  POS q ← catalogo.head()
  while not catalogo.finished(q) do
    ITEM v ← catalogo.read(q)
    if v.nazione = spagnola then L.insert(L.next(L.tail()), v)
    if v.rango = duchessa then
      | q ← catalogo.remove(q)
    else
      | q ← catalogo.next(q)
```

Complessità: vedremo in seguito



♦E' possibile realizzare una lista con vettori

- ♦Posizione \equiv indice nel vettore
- ♦Le operazioni `insert()` e `remove()` cosa fanno?
- ♦Vedremo successivamente il costo delle operazioni

♦Problema

- ♦Spesso non si conosce a priori quanta memoria serve per memorizzare la lista
- ♦Se ne alloca una certa quantità, per poi accorgersi che non è sufficiente.

♦Soluzione

- ♦Si alloca un vettore di dimensione maggiore, si ricopia il contenuto del vecchio vettore nel nuovo e si rilascia il vecchio vettore
- ♦Esempi: `java.util.Vector`, `java.util.ArrayList`



Stack

♦ Una pila (stack)

- ♦ è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato: *“quello che per meno tempo è rimasto nell'insieme”*
- ♦ politica *“last in, first out”* (LIFO)

♦ Operazioni previste

STACK

% Restituisce **true** se la pila è vuota

boolean isEmpty()

% Inserisce *v* in cima alla pila

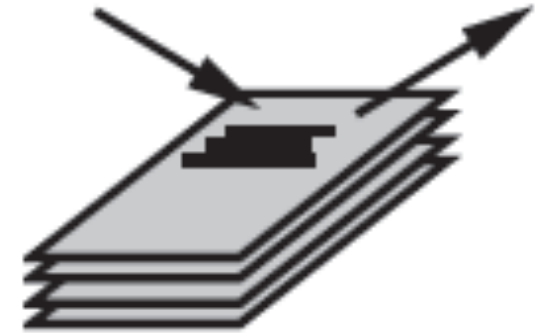
push (ITEM *v*)

% Estrae l'elemento in cima alla pila e lo restituisce al chiamante

ITEM pop()

% Legge l'elemento in cima alla pila

ITEM top()



pratiche burocratiche



tubetto di pastiglie



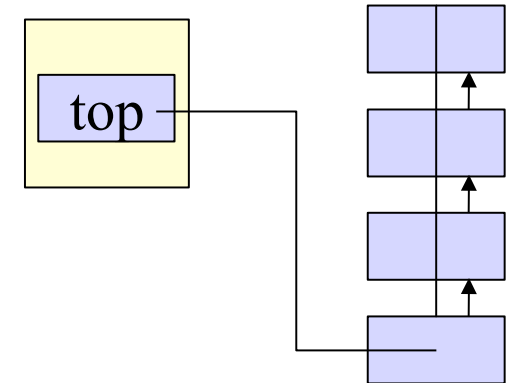
♦ Possibili utilizzi

- ♦ Nei linguaggi con procedure: gestione dei record di attivazione

- ♦ Nei linguaggi stack-oriented:

- ♦ Le operazioni elementari prendono uno-due operandi dallo stack e inseriscono il risultato nello stack

- ♦ Es: Postscript, Java bytecode



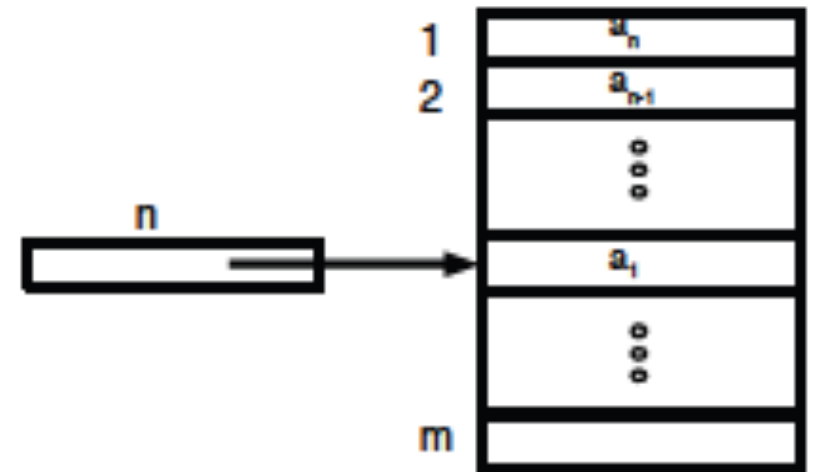
♦ Possibili implementazioni

- ♦ Tramite *liste bidirezionali*

- ♦ puntatore all'elemento top

- ♦ Tramite *vettore*

- ♦ dimensione limitata, overhead più basso



✦ Reverse Polish Notation (PNG), o notazione postfissa

✦ Espressioni aritmetiche in cui gli operatori seguono gli operandi

✦ Definita dalla grammatica: $\langle \text{expr} \rangle ::= \langle \text{numeral} \rangle \mid \langle \text{expr} \rangle \langle \text{expr} \rangle \langle \text{operator} \rangle$

✦ Esempi:

✦ $(7 + 3) \times 5$ si traduce in $7 \ 3 + 5 \times$

✦ $7 + (3 \times 5)$ si traduce in $7 \ 3 \ 5 \times +$

✦ Valutazione PNG, stack based (esempio: Java bytecode)

✦ push 7 7

✦ push 3 3 7

✦ push 5 5 3 7

✦ op. \times : pop, pop, push 15 7

✦ op. $+$: pop, pop, push 22



Stack - pseudocodice

STACK

ITEM[] A % Elementi
integer n % Cursore
integer m % Dim. max

STACK Stack(integer dim)

 STACK $t \leftarrow$ new STACK

$t.A \leftarrow$ new integer[1 ... dim]

$t.m \leftarrow dim$

$t.n \leftarrow 0$

 return t

ITEM top()

 precondition: $n > 0$

 return $A[n]$

boolean isEmpty()

 return $n = 0$

ITEM pop()

 precondition: $n > 0$

 ITEM $t \leftarrow A[n]$

$n \leftarrow n - 1$

 return t

push(ITEM v)

 precondition: $n < m$

$n \leftarrow n + 1$

$A[n] \leftarrow v$



♦ Una coda (queue)

- ♦ è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato: *“quello che per più tempo è rimasto nell'insieme”*
- ♦ politica ***“first in, first out”*** (***FIFO***)



♦ Operazioni previste

QUEUE

% Restituisce **true** se la coda è vuota

boolean isEmpty()

% Inserisce *v* in fondo alla coda

enqueue (ITEM *v*)

% Estrae l'elemento in testa alla coda e lo restituisce al chiamante

ITEM **dequeue**()

% Legge l'elemento in testa alla coda

ITEM **top**()

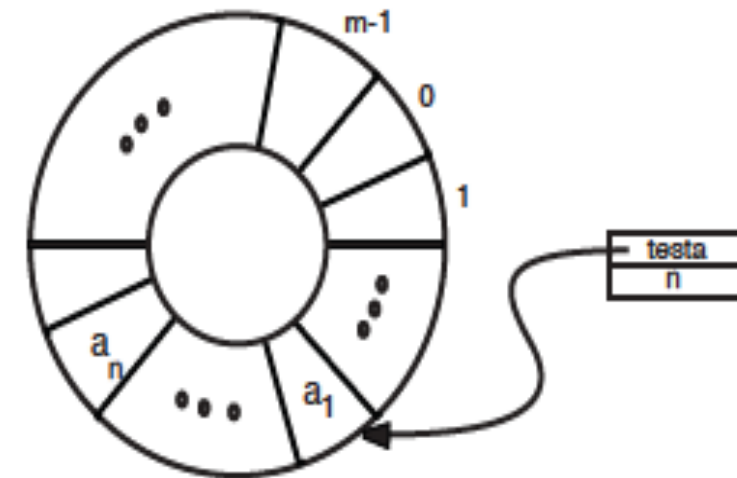
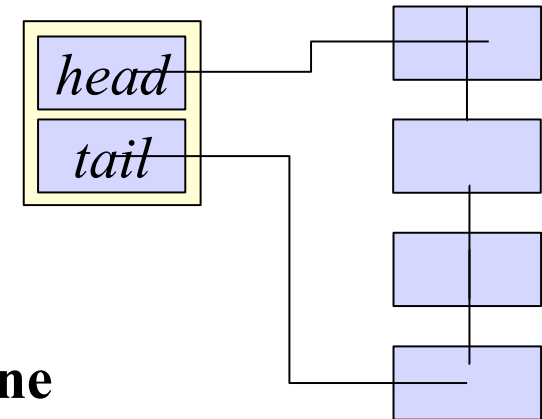


♦ Possibili utilizzi

- ♦ Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda
- ♦ La politica FIFO è fair

♦ Possibili implementazioni

- ♦ Tramite liste monodirezionali
 - ♦ puntatore *head* (inizio della coda), per estrazione
 - ♦ puntatore *tail* (fine della coda), per inserimento
- ♦ Tramite array circolari
 - ♦ dimensione limitata, overhead più basso

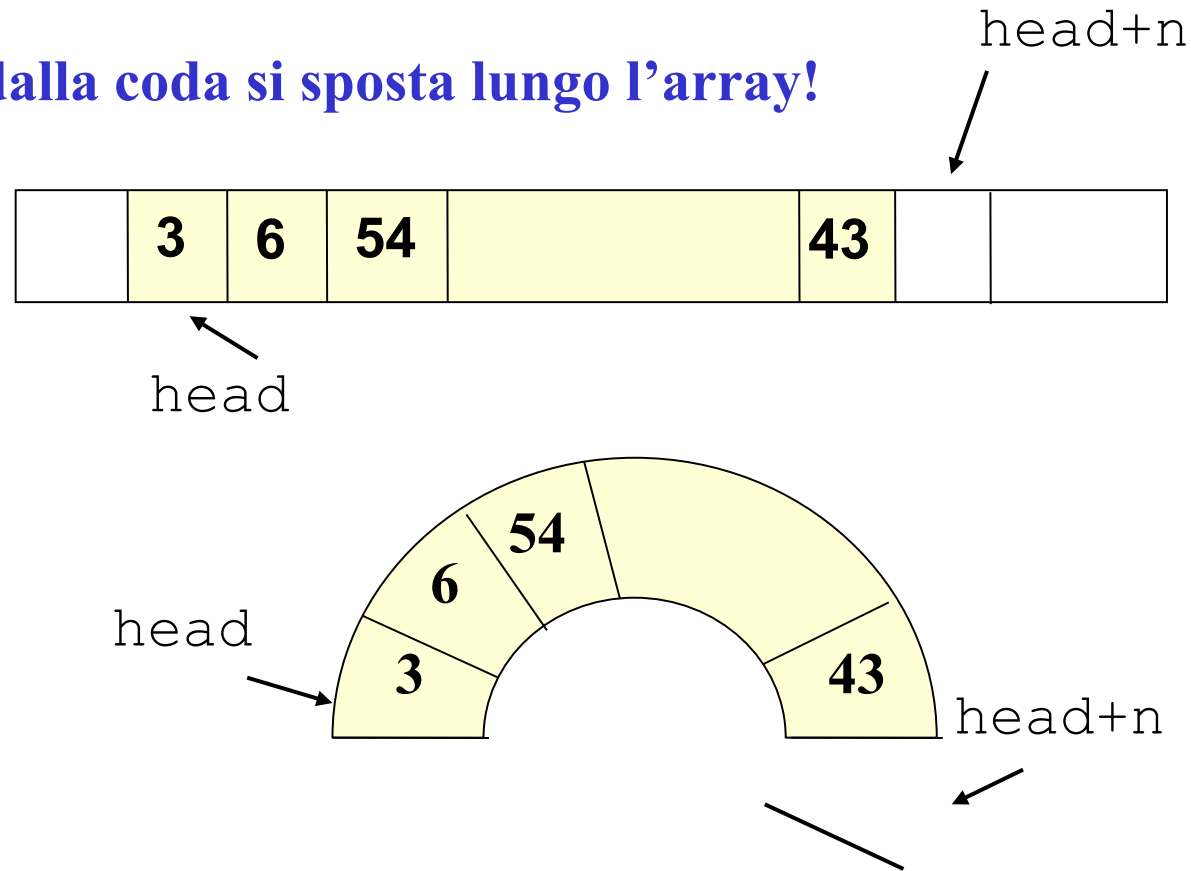


Coda: Realizzazione tramite vettore circolare

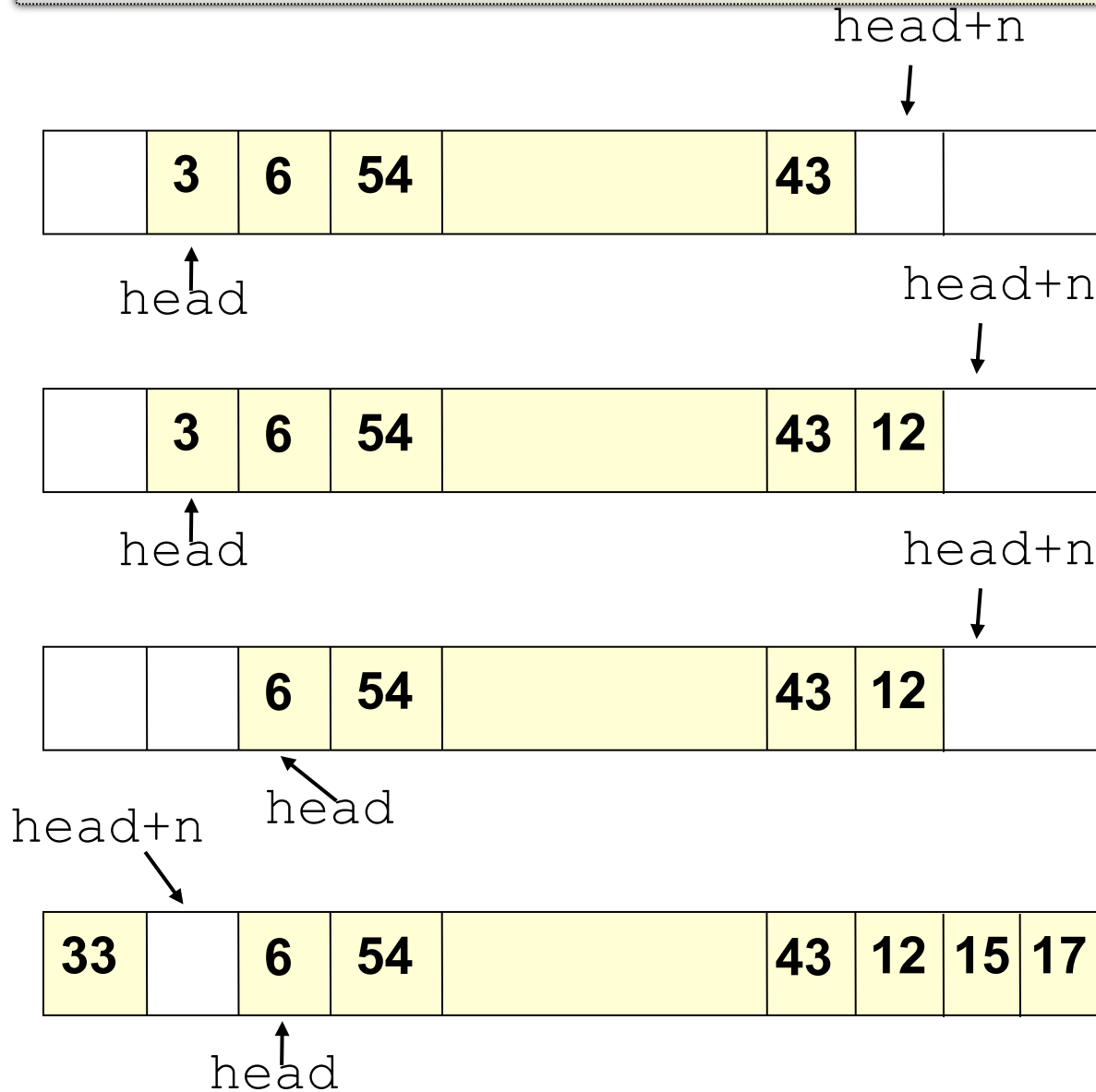
- ♦ La “finestra” dell’array occupata dalla coda si sposta lungo l’array!

- ♦ **Dettagli implementativi**

- ♦ L'array circolare può essere implementato con un'operazione di modulo
- ♦ Bisogna prestare attenzione ai problemi di overflow (buffer pieno)



Coda: Realizzazione tramite vettore circolare



`enqueue (12)`

`dequeue () → 3`

`enqueue (15, 17, 33)`



QUEUE

ITEM[] *A* % Elementi
integer *n* % Dim. attuale
integer *testa* % Testa
integer *m* % Dim. max

QUEUE Queue(integer *dim*)
 QUEUE *t* ← new QUEUE
 t.A ← new integer[0...*dim* - 1]
 t.m ← *dim*
 t.testa ← 0
 t.n ← 0
 return *t*

ITEM top()
 precondition: *n* > 0
 return *A[testa]*

boolean isEmpty()

 return *n* = 0

ITEM dequeue()

 precondition: *n* > 0

 ITEM *t* ← *A[testa]*

testa ← (*testa* + 1) mod *m*

n ← *n* - 1

 return *t*

enqueue(ITEM *v*)

 precondition: *n* < *m*

*A[(testa + *n*) mod *m*] ← *v**

n ← *n* + 1



Esercizi

Modificare lo pseudocodice relativo a `top()`, `isEmpty()`, `dequeue()` e `enqueue()` nella struttura `Queue` nel caso si abbia la seguente definizione di `Queue`:

`Item [] A` % Elementi
integer *n* % Dim. Attuale
integer *testa* % Testa
integer *m* % Dim. Massima

QUEUE `Queue(integer dim)`
 QUEUE *t* \leftarrow **new** **QUEUE**
 t.A \leftarrow **new integer** [$1 \dots dim$]
 t.m $\leftarrow dim$
 t.testa $\leftarrow 1$
 t.n $\leftarrow 0$
 return *t*



Esercizi

Modificare lo pseudocodice relativo a `top()`, `isEmpty()`, `dequeue()` e `enqueue()` nella struttura `Queue` nel caso si utilizzi un vettore circolare a due indici: uno per la testa e uno per la coda:

<code>Item [] A</code>	% Elementi
<code>integer <i>n</i></code>	% Dim. Attuale
<code>integer <i>testa</i></code>	% Testa
<code>integer <i>coda</i></code>	% Coda
<code>integer <i>m</i></code>	% Dim. Massima

```
QUEUE Queue(integer dim)  
    QUEUE t ← new QUEUE  
    t.A ← new integer [0... dim-1]  
    t.m ← dim  
    t.testa ← 0  
    t.coda ← 0  
    t.n ← 0  
    return t
```

