UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

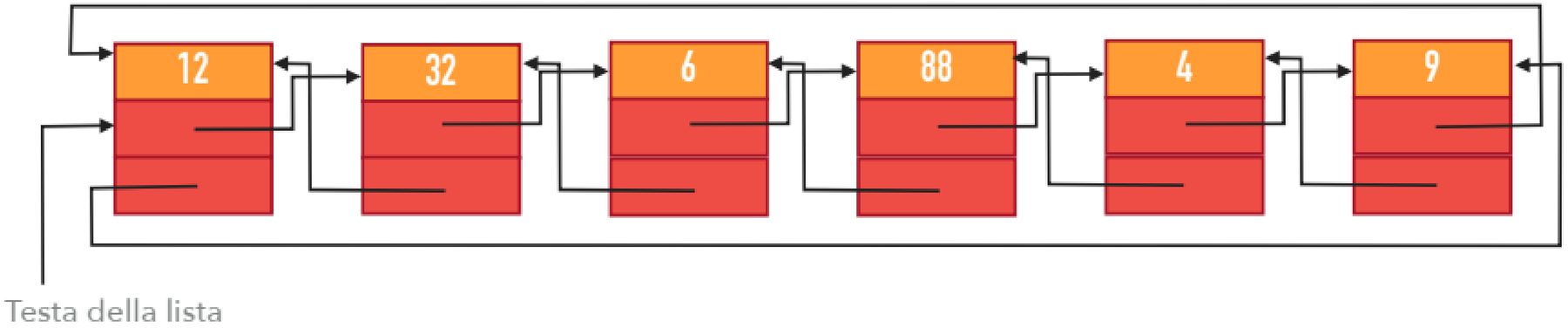
MODULO 2: Pila e coda (Stack & queue)

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 23 aprile 2025

RECAP



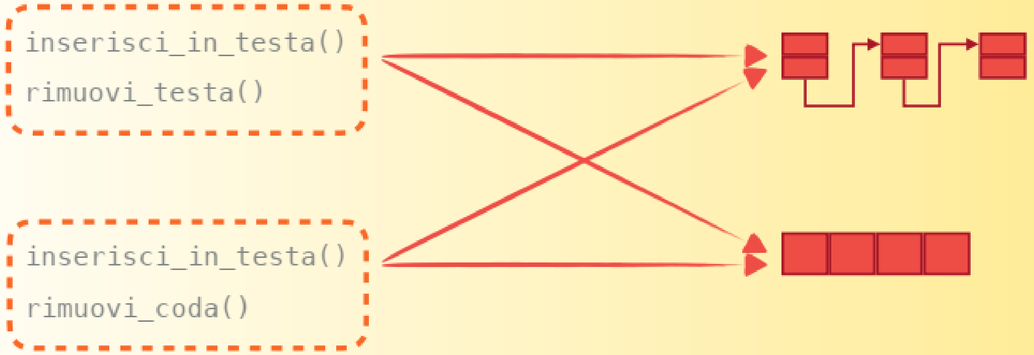
Lista concatenata doppia e circolare

OPERAZIONE	ARRAY	LISTA CONC. SINGOLA *(con tail)*	LISTA CONC. DOPPIA *(con tail)*
ACCEDERE AD UN ELEMENTO	$O(1)$	$O(n)$	$O(n)$
RICERCA	$O(n)$	$O(n)$	$O(n)$
INSERIMENTO	$O(n)$	$O(n)$ ma $O(1)$ se in testa	$O(n)$ ma $O(1)$ se in testa o in coda
RIMOZIONE	$O(n)$	$O(n)$ ma $O(1)$ se in testa o se abbiamo già il puntatore al nodo da cancellare	$O(n)$ ma $O(1)$ se in testa o in coda

RECAP

Definisce il comportamento

Specifica come sono organizzati i dati



Ma la distinzione non è sempre netta, è un continuo!

Stack (pila) di piatti

TIPO DI STRUTTURA	FIRST-IN-LAST-OUT
OPERAZIONI POSSIBILI	**Significato**
PUSH	Inserire un elemento nello stack
POP	Rimuovere un elemento dallo stack
EMPTY	Verificare se lo stack è vuoto
PEEK	Leggere il valore in cima allo stack

Reverse Polish Notation (RPN)

Notazione usata in alcune calcolatrici scientifiche

(2 + 3) × 4 diventa 2 3 + 4 ×

UNIVERSITÀ DEGLI STUDI DI TRIESTE

STACK: IMPLEMENTAZIONE

- Dato che uno stack definisce solo le operazioni che si possono effettuare dobbiamo decidere come implementare lo stack

- Vediamo due modi:
  - Stack implementato **tramite liste concatenate singole**
  - Stack implementato **tramite array**

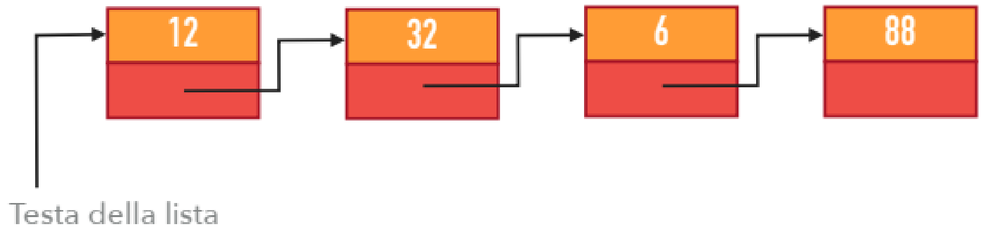
STACK: IMPLEMENTAZIONE CON LISTE CONCATENATE

- L’implementazione con liste concatenate richiede due operazioni:
  - Push: inserimento in testa alla lista
  - Pop: rimozione dalla testa della lista

- Altre operazioni sono comunque facili da implementare:
  - Peek: valore della testa della lista
  - Empty: controllo che la testa non sia None

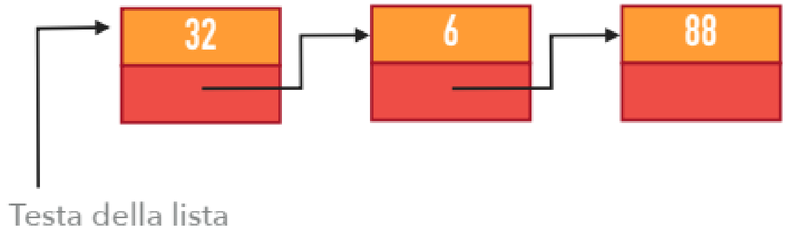
STACK: IMPLEMENTAZIONE CON LISTE CONCATENATE

```
pop()
push(3)
```



STACK: IMPLEMENTAZIONE CON LISTE CONCATENATE

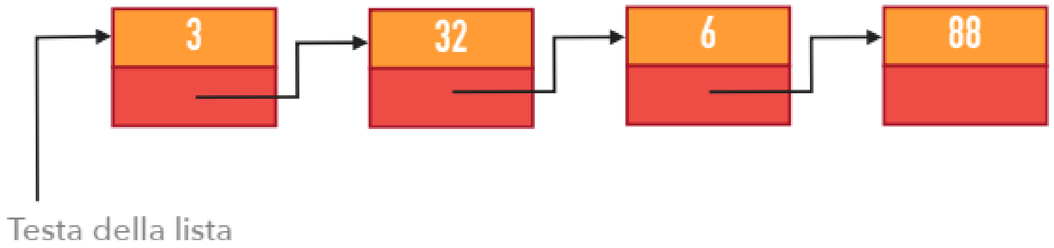
```
pop()
push(3)
```



Valore ritornato: 12

STACK: IMPLEMENTAZIONE CON LISTE CONCATENATE

```
pop()
push(3)
```



STACK: IMPLEMENTAZIONE CON LISTE CONCATENATE

- La **complessità** di inserimento e rimozione è pari a quella di inserimento in testa e rimozione del primo elemento

OPERAZIONE	ARRAY	LISTA CONC. SINGOLA *(con tail)*	LISTA CONC. DOPPIA *(con tail)*
ACCEDERE AD UN ELEMENTO	$O(1)$	$O(n)$	$O(n)$
RICERCA	$O(n)$	$O(n)$	$O(n)$
INSERIMENTO	$O(n)$	$O(n)$ ma $O(1)$ se in testa	$O(n)$ ma $O(1)$ se in testa o in coda
RIMOZIONE	$O(n)$	$O(n)$ ma $O(1)$ se in testa o se abbiamo già il puntatore al nodo da cancellare	$O(n)$ ma $O(1)$ se in testa o in coda

Push: *O*(1)

Pop: *O*(1)

- **Nota bene**: la complessità delle operazioni dipende dall'implementazione!
- Non possiamo dire quale sia il costo computazionale senza dire come sono state implementate!

IMPLEMENTAZIONE CON LISTE CONCATENATE: QUIZ

Perché non abbiamo usato una lista concatenata **doppia** per implementare lo stack?

A) Potevamo usarla, avrebbe migliorato la complessità

B) Potevamo usarla, ma non avrebbe migliorato la complessità

C) Non era utilizzabile

D) Era utilizzabile solo se circolare

STACK: IMPLEMENTAZIONE CON **ARRAY**

- Dato che un array non “cresce”, gli array possono **implementare facilmente solo stack di dimensione limitata**, ma *copiando* l’array nel caso si inserisca un elemento in un array “pieno” è possibile ottenere stack di dimensioni arbitrarie.

Spoiler Lab!!

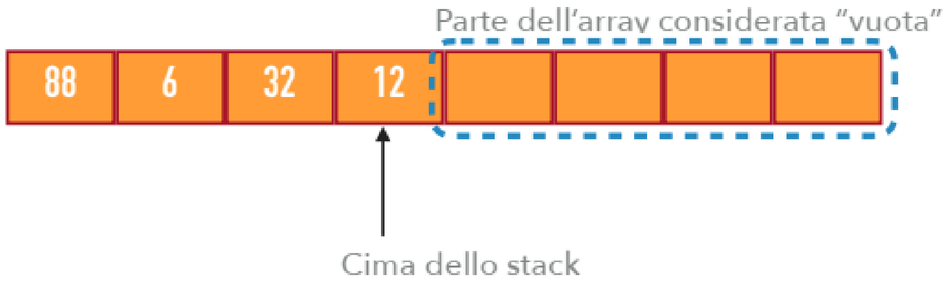
- Uno **stack di *n* elementi** occupa le posizioni da *0* a *n-1* dell’array con la testa dello stack in posizione *n-1*. L’**array ha dimensione K** ≥ n.

- **Inserire un elemento** significa copiarlo in posizione *n* e ricordarsi che la testa dello stack è ora in posizione *n*

- **La rimozione** è simile: ritorniamo l’elemento in posizione *n-1* ricordando che la testa dello stack è ora in posizione *n-2*

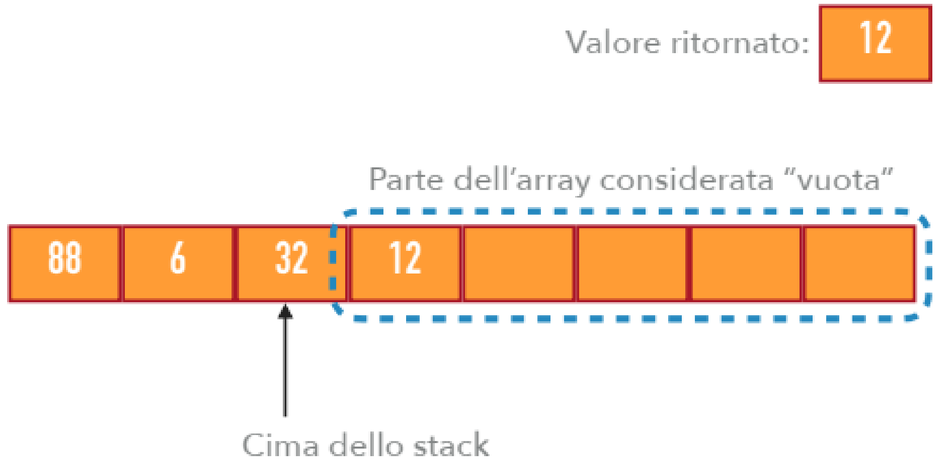
STACK: IMPLEMENTAZIONE CON ARRAY

```
pop()
push(3)
```



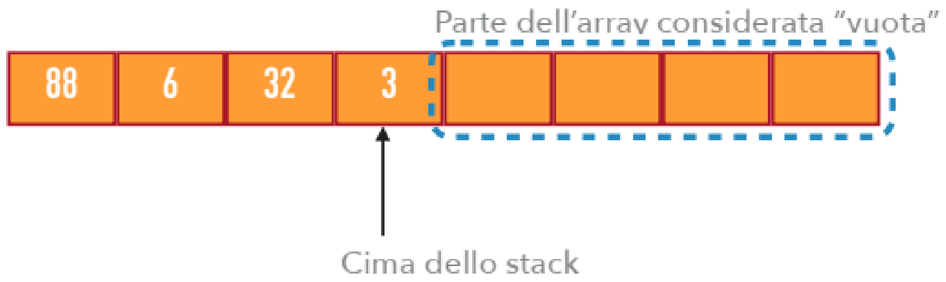
STACK: IMPLEMENTAZIONE CON ARRAY

```
pop()
push(3)
```



STACK: IMPLEMENTAZIONE CON ARRAY

```
pop()
push(3)
```



Se la cima dello stack fosse stata l'ultima posizione dell'array avremmo dovuto copiare tutti in un array più grande prima di inserire il nuovo elemento!

STACK: IMPLEMENTAZIONE CON ARRAY

- La complessità della rimozione è data dal decrementare una variabile e copiare un valore: ***O*(1)**
- L'inserimento, nel caso ci sia spazio disponibile è rapido richiedendo un numero costante di passi…
- … ma nel caso peggiore dobbiamo copiare l'intero array, ottenendo un tempo che è ***O*(*n*)**

CODE

- **Struttura dati astratta** con le seguenti operazioni:
  - **Enqueue**. Inserisce in elemento in fondo alla coda
  - **Dequeue**. Rimuove l'elemento nella coda che è stato inserito più indietro nel tempo

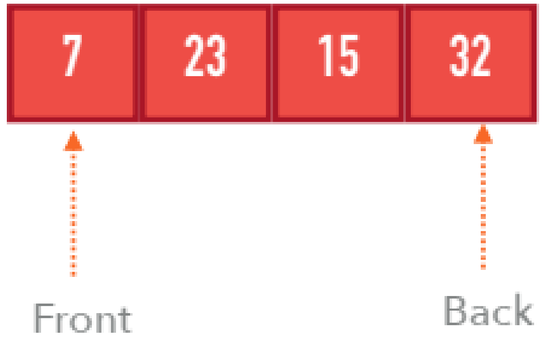
- Dette anche **FIFO** (**First in - First out**)

- Altre possibili operazioni:
  - **Empty**. Per chiedere se la coda è vuota

CODA: ESEMPIO (1/6)

Operazioni da eseguire:

```
enqueue(3)
dequeue()
enqueue(5)
dequeue()
dequeue()
```



Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseriti.

CODA: ESEMPIO (2/6)

Operazioni da eseguire:

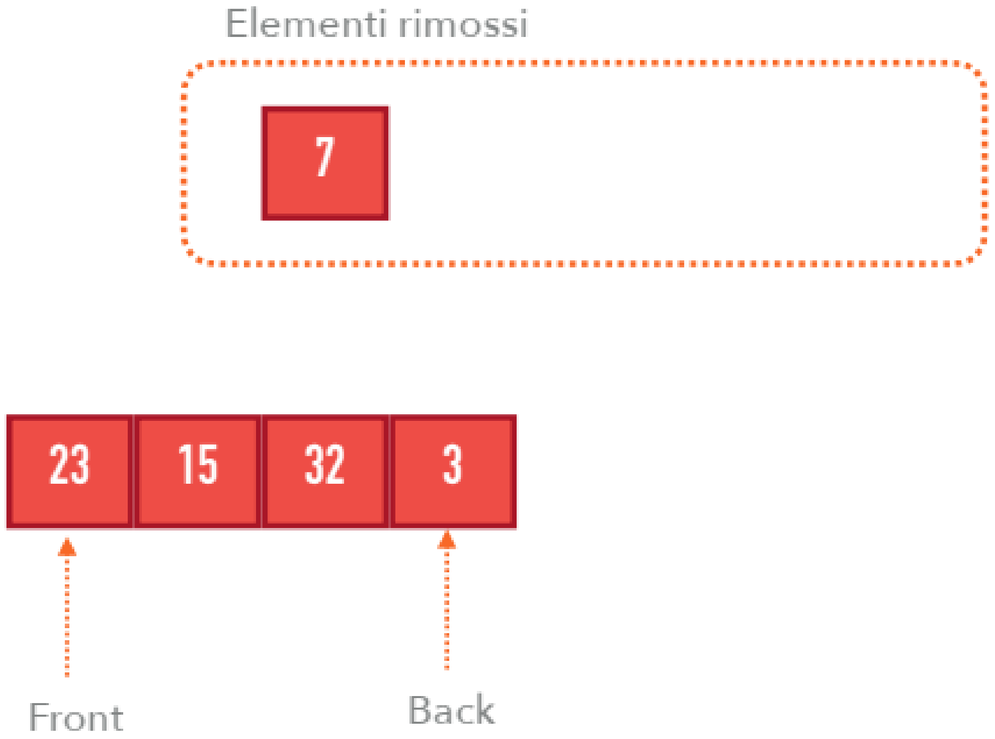
```
enqueue(3)
dequeue()
enqueue(5)
dequeue()
dequeue()
```

Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseritiav

CODA: ESEMPIO (3/6)

Operazioni da eseguire:

```
enqueue(3)
dequeue()
enqueue(5)
dequeue()
dequeue()
```

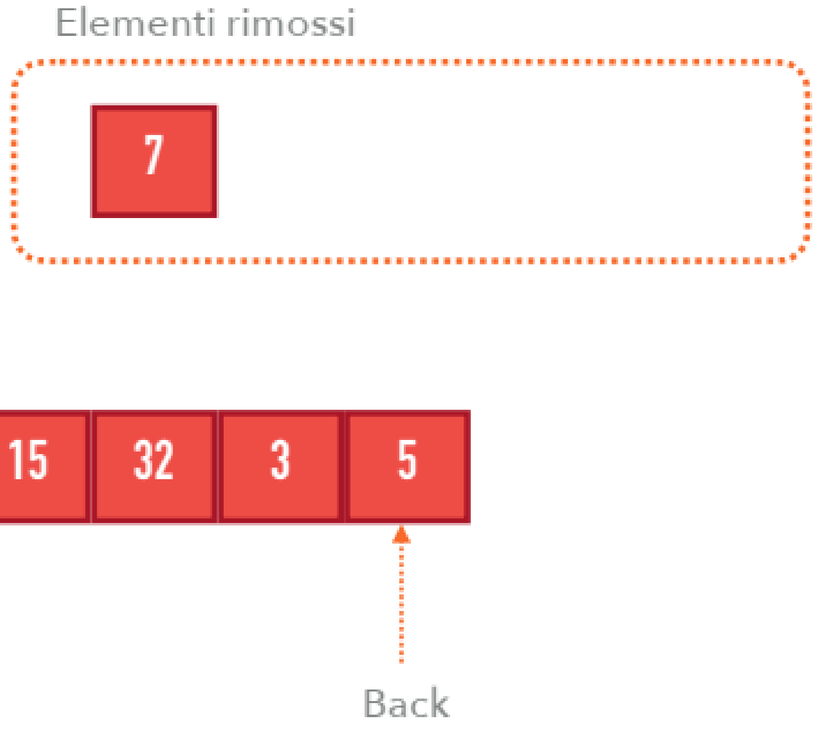


Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseritiav

CODA: ESEMPIO (4/6)

Operazioni da eseguire:

```
enqueue(3)
dequeue()
enqueue(5)
dequeue()
dequeue()
```

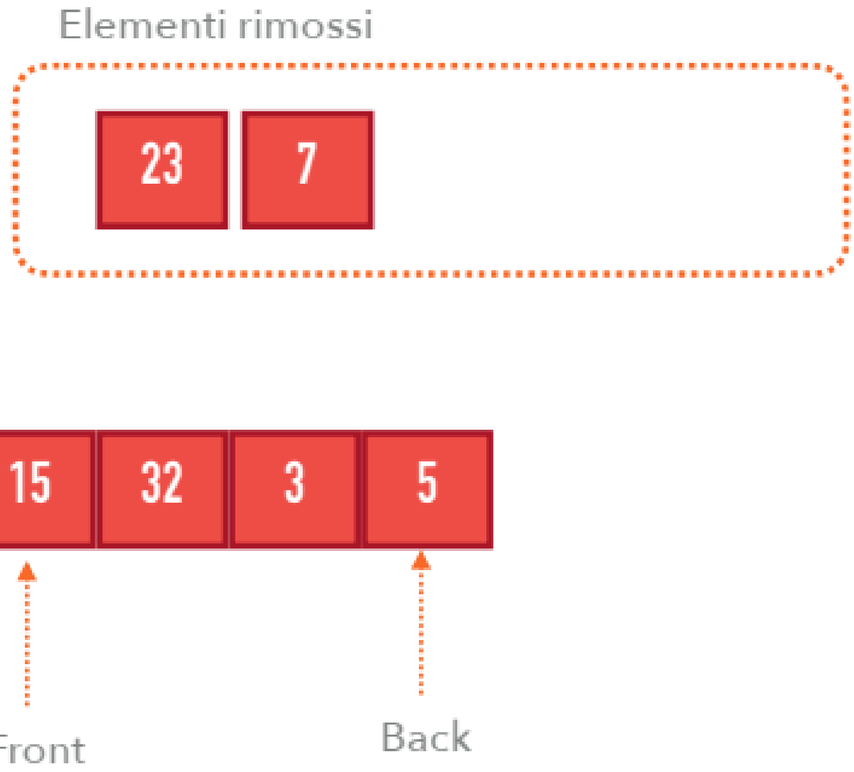


Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseritiav

CODA: ESEMPIO (5/6)

Operazioni da eseguire:

```
enqueue(3)
dequeue()
enqueue(5)
dequeue()
dequeue()
```

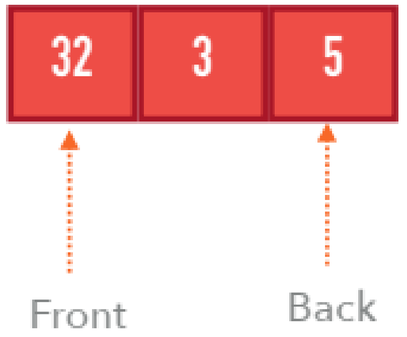


Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseritiav

CODA: ESEMPIO (6/6)

Operazioni da eseguire:

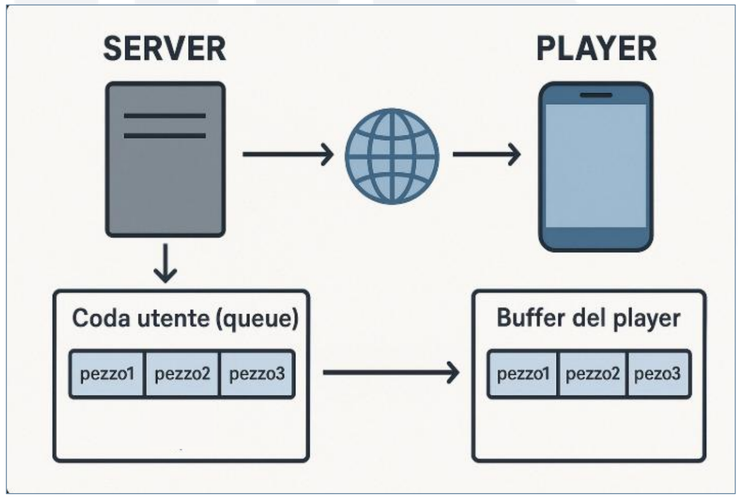
```
enqueue(3)
dequeue()
enqueue(5)
dequeue()
dequeue()
```



Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseritiav

CODE: UTILITA’

- **Gestione di dati in arrivo**: di solito vogliamo siano processati nello stesso ordine con cui li riceviamo
- All’interno di altri algoritmi: ricerca di un percorso in un grafo, etc.
- Coda dei processi da eseguire, ***streaming video***, etc.
- Forse una delle strutture più comuni nell’informatica



Il programma («player») riproduce il video (esempio: **VLC**, **YouTube app**, **Netflix app**, **Twitch player**...). **Il buffer** è **una piccola memoria temporanea dentro** il player.

A cosa serve il buffer?
Il **buffer** si riempie con **un po' di dati video** prima che il player inizi a mostrarli a schermo.
Serve per compensare:
• **Ritardi della rete** (lag)
• **Piccole interruzioni**
• **Variazioni della velocità di internet**

CODE: IMPLEMENTAZIONE

- Vediamo due modi:
  - Coda implementata **tramite liste concatenate singole**
  - Coda implementata **tramite array**

CODE: IMPLEMENTAZIONE CON LISTE CONCATENATE

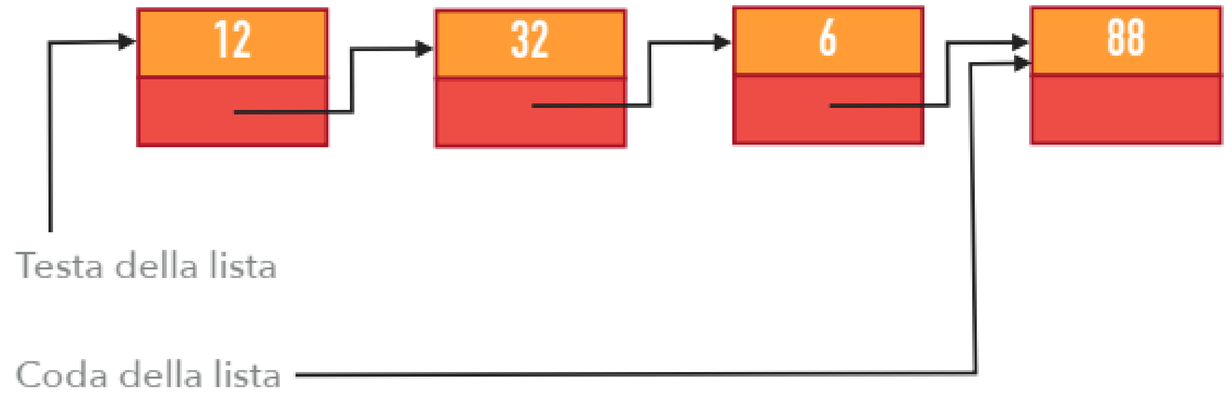
- Possiamo usare una lista concatenata singola in cui la coda **inizia alla testa della lista** e termina con l'ultimo elemento

- La rimozione del primo elemento è rapida…
- *…ma l'aggiunta in coda alla lista richiede di scorrere tutta la lista*

- Possiamo eliminare il problema in almeno due modi:
  - Lista con riferimento alla coda (***tail***)
  - **Lista circolare** con riferimento all'ultimo elemento (invece che al primo)

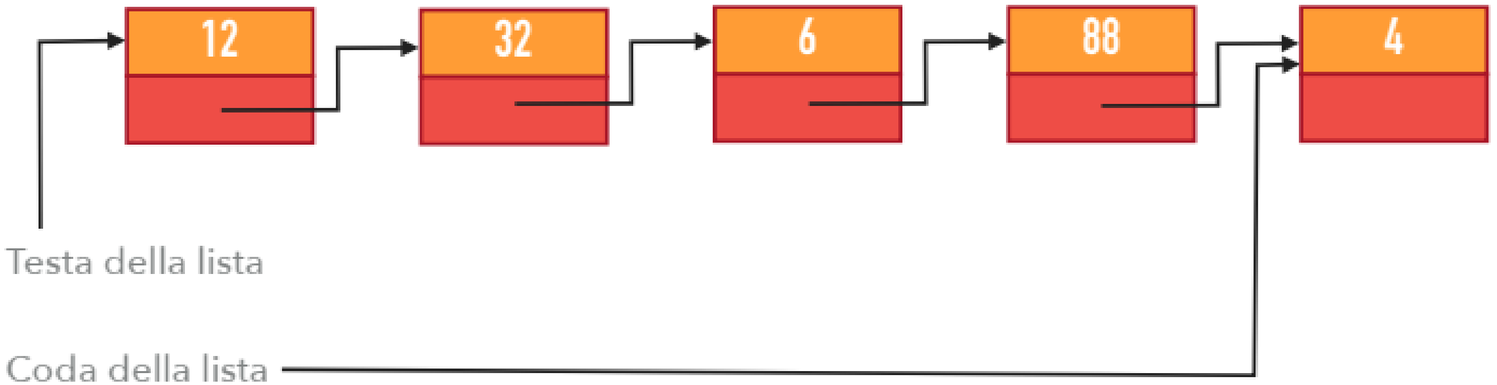
CODA: IMPLEMENTAZIONE CON LISTE CONCATENATE

```
enqueue(4)
dequeue()
```



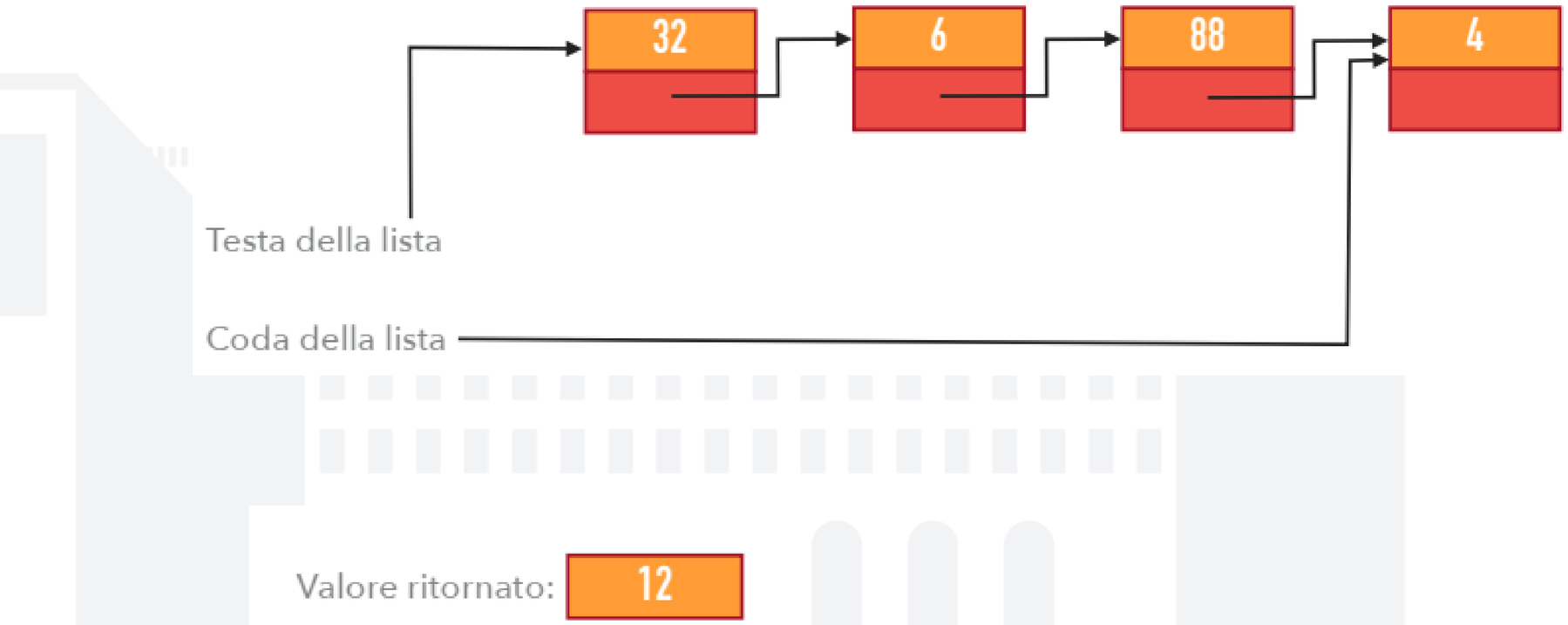
CODA: IMPLEMENTAZIONE CON LISTE CONCATENATE

```
enqueue(4)
dequeue()
```



CODA: IMPLEMENTAZIONE CON LISTE CONCATENATE

```
enqueue(4)
dequeue()
```



CODA: IMPLEMENTAZIONE CON LISTE CONCATENATE

- La **rimozione** in testa (“dequeue”) richiede tempo ***O*(1)**
- **L’aggiunta** in coda («enqueue») richiede anch’essa tempo ***O*(1)** ma *solo perché abbiamo un reference all’ultimo elemento* della lista
- Anche in questo caso potevamo usare una lista concatenata doppia (con un reference alla coda)…
- …ma non avremmo avuto vantaggi in termini di costo computazionale

CODA: IMPLEMENTAZIONE CON **ARRAY** (1/2)

- Possiamo utilizzare un array con *n* posizioni per memorizzare una coda in grado di contenere *n-1* elementi

- Si utilizza un **buffer o array circolare** che, finché non superiamo *n-1* elementi nella coda, permette di fare inserimenti e rimozioni in tempo costante

- Teniamo **due indici**:
  - dove inizia la coda
  - la prima posizione libera dopo la fine della coda

CODA: IMPLEMENTAZIONE CON ARRAY (2/2)

- Il primo indice denota dove si trova il primo elemento della coda
- Il secondo indice denota dove verrà inserito il prossimo elemento

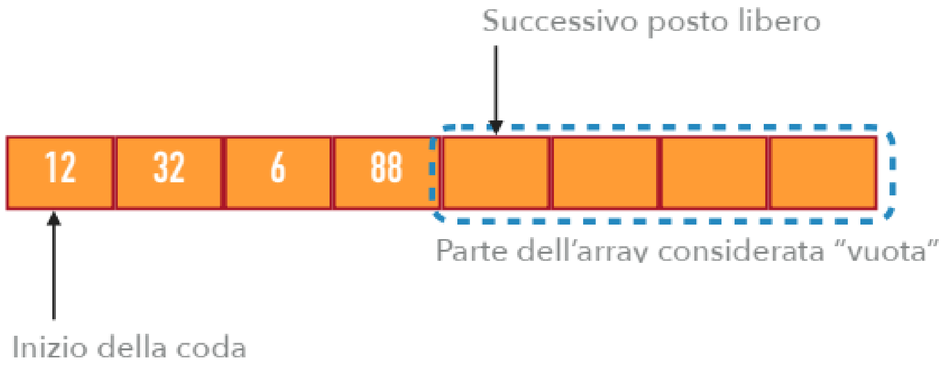
  Dequeue: spostamento in avanti del primo indice

  Enqueue: spostamento in avanti del secondo indice

- Gli indici sono considerati «modulo *n*» *(una volta che arrivano a n-1, si torna a 0)*

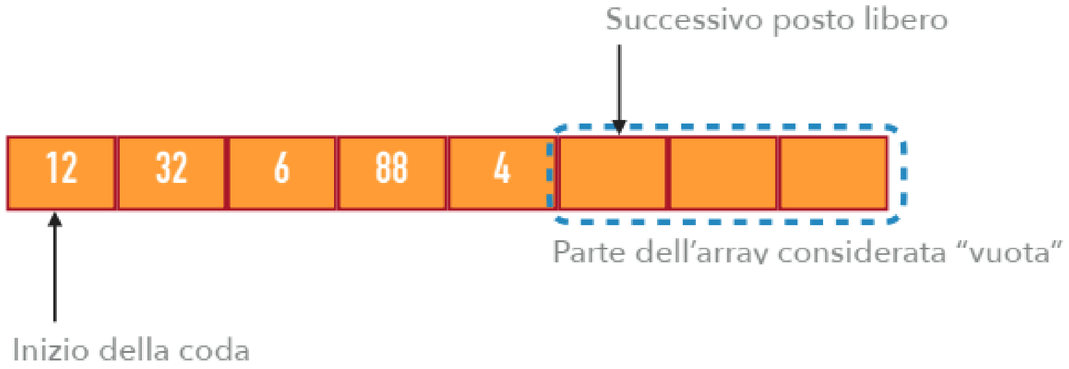
CODA: IMPLEMENTAZIONE CON ARRAY

```
enqueue(4)
dequeue()
```



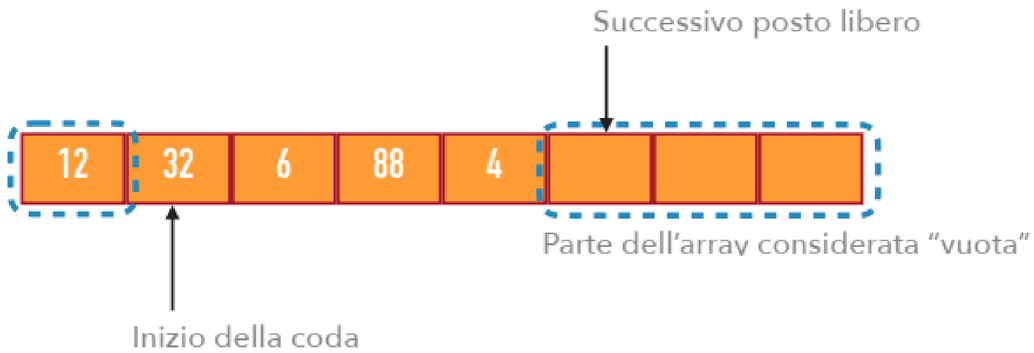
CODA: IMPLEMENTAZIONE CON ARRAY

```
enqueue(4)
dequeue()
```



CODA: IMPLEMENTAZIONE CON ARRAY

```
enqueue(4)
dequeue()
```

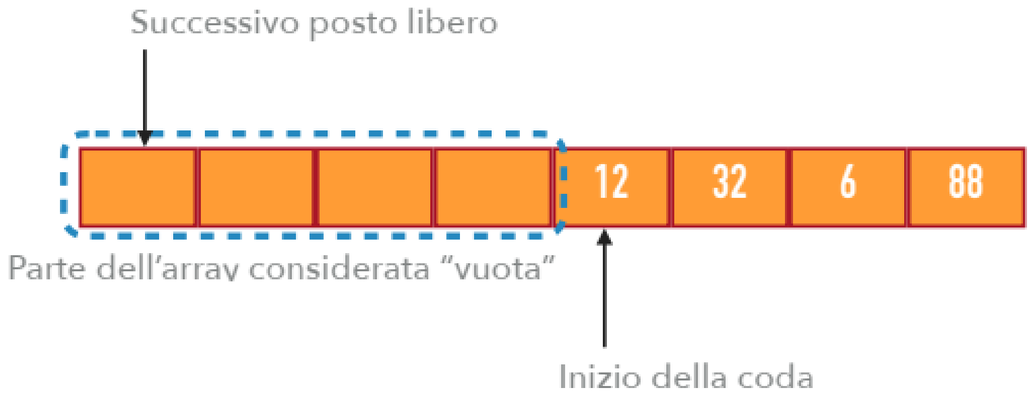


Valore ritornato: 12

CODA: IMPLEMENTAZIONE CON ARRAY **#2**

```
enqueue(4)
dequeue()
```

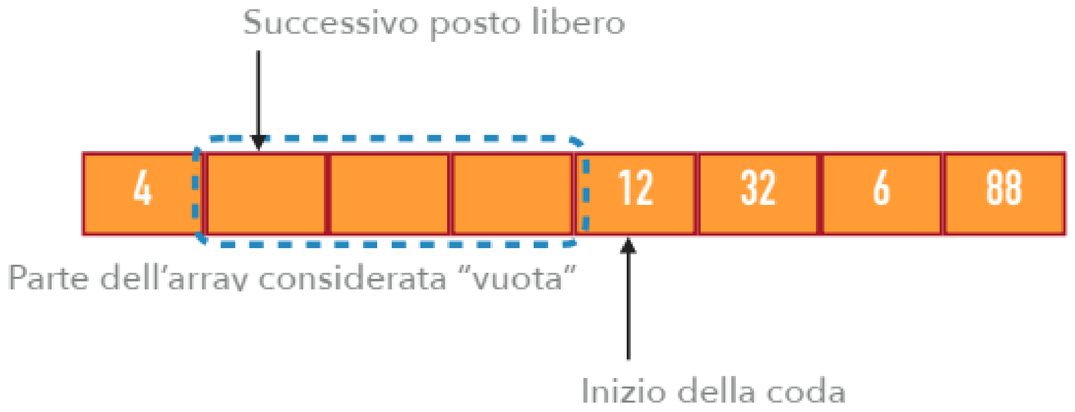
Le posizioni sono sempre considerate modulo *n*, quindi il successore della posizione *n* - 1 è la posizione 0



Questa array e quello dell'esempio precedente rappresentano la stessa coda!

CODA: IMPLEMENTAZIONE CON ARRAY #2

```
enqueue(4)
dequeue()
```

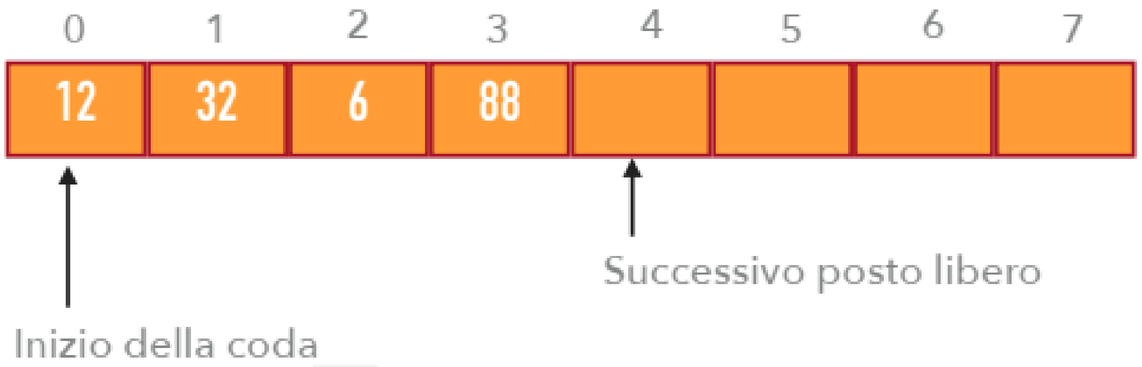


Gli elementi vengono rimossi nello stesso ordine con cui sono stati inseriti.

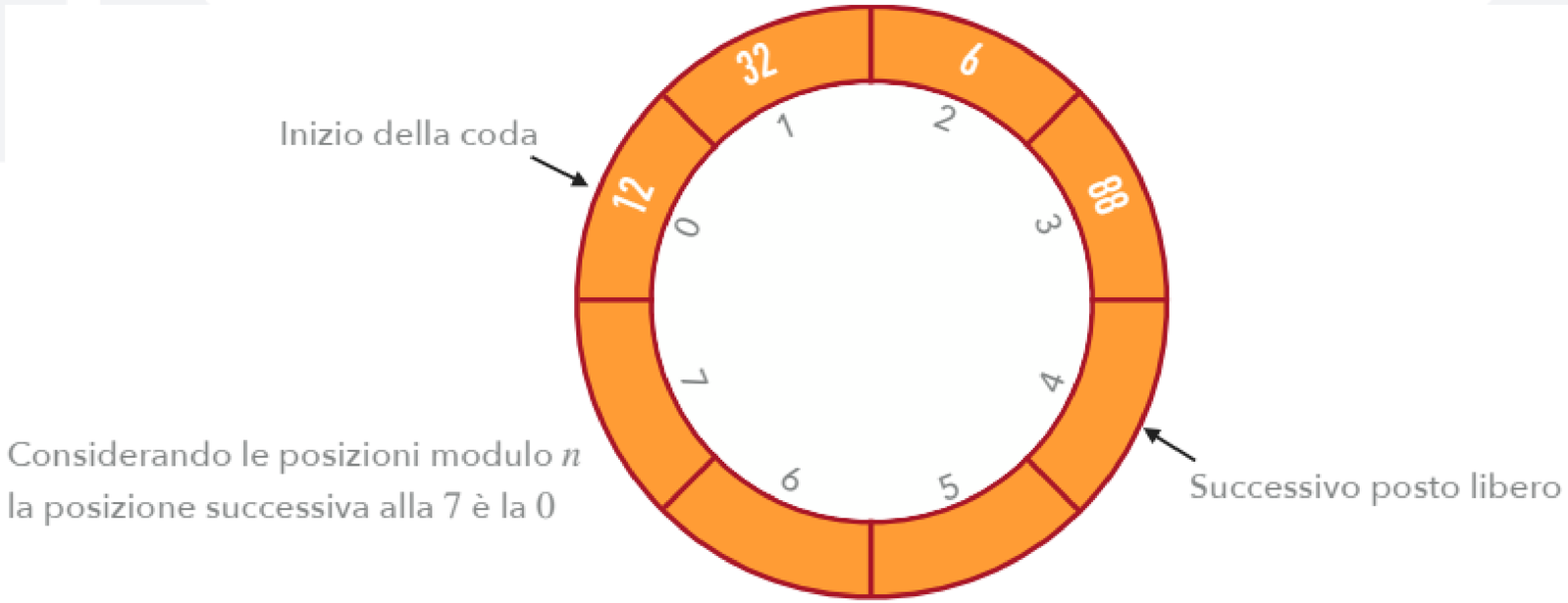
CODA: IMPLEMENTAZIONE CON ARRAY #2

```
enqueue(4)
dequeue()
```

CODA: IMPLEMENTAZIONE CON ARRAY



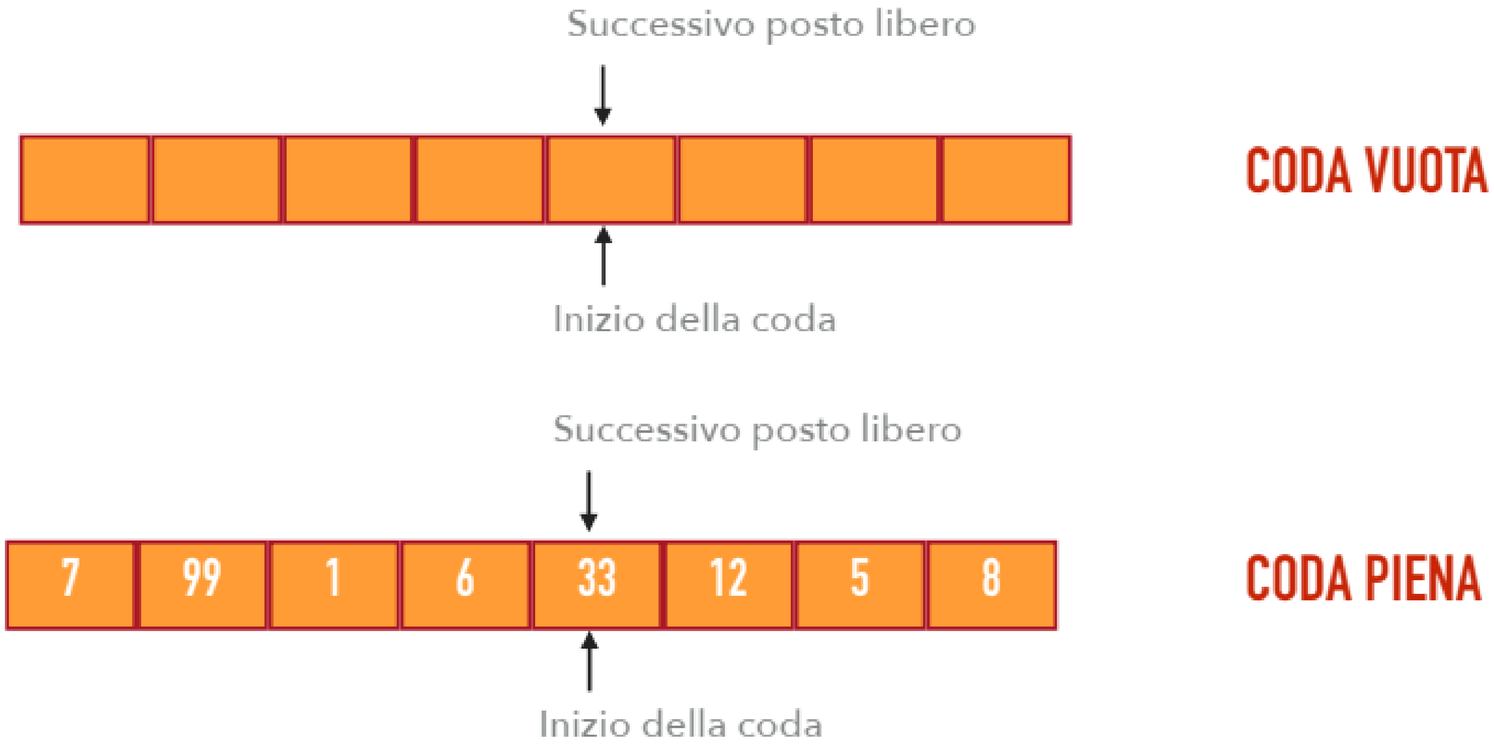
Due modi di vedere lo stesso array: in modo lineare e considerando le posizioni modulo *n*



UNIVERSITÀ DEGLI STUDI DI TRIESTE

IMPLEMENTAZIONE CON ARRAY: PERCHE' *n-1* ELEMENTI?

Se consentissimo di memorizzare *n* elementi (invece di *n-1*) non potremmo distinguere tra due casi:

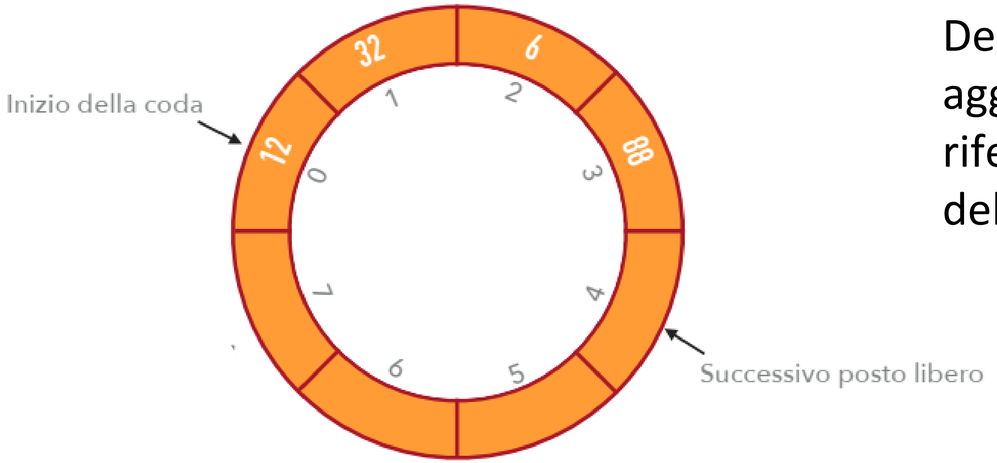


*Conflitto potenzialmente risolubile memorizzando informazioni aggiuntive

IMPLEMENTAZIONE CON ARRAY

- Fino a quando il numero di elementi nella coda rimane limitato ogni operazione di inserimento e rimozione richiede tempo costante **O(1)**

- Se non avessimo i due indici ma tenessimo sempre l'inizio della coda in posizione 0 dovremmo copiare l'array ad ogni operazione di *dequeue*!



Dequeue richiede aggiornamento del riferimento all'inizio della coda

Potremmo utilizzare degli stack per simulare una coda?

Possiamo utilizzare due stack:

- In uno effettueremo le operazioni di *enqueue*
- Nell'altro le operazioni di *dequeue*
- Dobbiamo — in qualche modo — spostare elementi da uno stack all'altro

CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)      B.push(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

CODA: IMPLEMENTAZIONE CON STACK

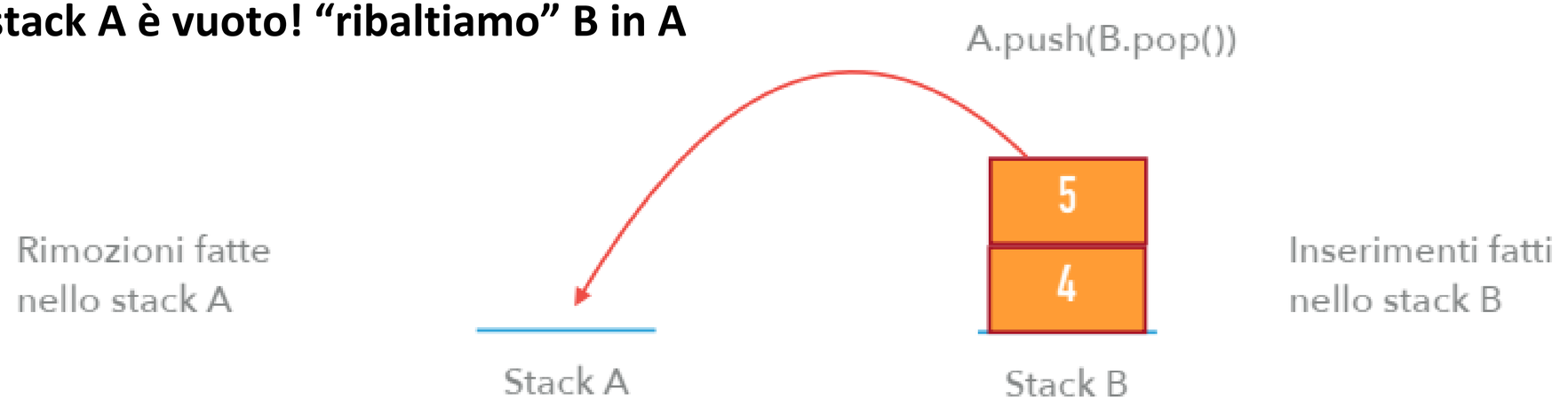
```
enqueue(4)
enqueue(5)    B.push(5)
dequeue()
enqueue(3)
dequeue()
```

CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

NOTA. Questo «ribaltamento» consiste in una serie di pop() dallo Stack B che sono operazioni native e potenzialmente molto veloci da eseguire.

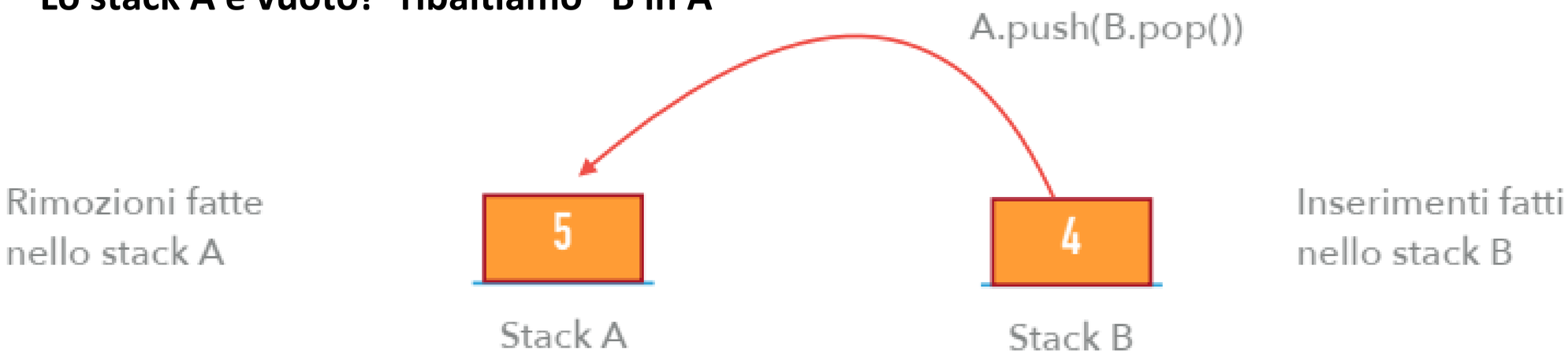
Lo stack A è vuoto! “ribaltiamo” B in A



CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

Lo stack A è vuoto! “ribaltiamo” B in A



CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

Ora il top dello stack A contiene

CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

Valore ritornato: 4

CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

`B.push(3)`

Rimozioni fatte nello stack A

Stack A

Stack B

Inserimenti fatti nello stack B

CODA: IMPLEMENTAZIONE CON STACK

```
enqueue(4)
enqueue(5)
dequeue()
enqueue(3)
dequeue()
```

Valore ritornato: 5

Se lo stack A non è vuoto ci basta fare una operazione di pop

CODA: IMPLEMENTAZIONE CON STACK

Qual è quindi la complessità per un'operazione di inserimento e rimozione di un elemento in una coda se la implementiamo con due stack?

- Inserimento (enqueue) richiede tempo costante **O(1)**

- Rimozione (dequeue) richiede tempo costante **O(1)** se lo stack A non è vuoto.

Altrimenti, devo prima «ribaltare» lo stack B in A. Caso peggiore: **O(n)**.

Nota. Se però si fanno **tante operazioni di enqueue** (inserimenti) ma **poche dequeue** (rimozioni), allora il costo O(n) sporadico del trasferimento Stack B → Stack A non pesa quasi niente.

DOUBLE-ENDED QUEUE («DEQUE»)

Ibrido di stack e coda: si possono **inserire e rimuovere da entrambe le estremità**:

- Inserire elementi all'inizio o alla fine
- Rimuovere elementi dall'inizio o dalla fine

Come negli altri casi può essere implementata come lista concatenata oppure con un array utilizzato come buffer circolare.

In questo caso, **potrebbe essere comoda una lista concatenata doppia**:

- ogni nodo punta sia avanti che indietro
- inserire e togliere all'inizio o alla fine diventa **O(1)**

Applicazioni: browser cronologia avanti/indietro, task scheduling (scegliere se processare subito o mettere in coda).

QUIZ FINALE

In una struttura dati inseriamo in ordine i valori:

4 5 9 3

Rimuovendoli li otteniamo in ordine:
3 9 5 4

Quale di queste strutture dati astratte potrebbe essere?

A) Coda

B) Stack

C) Reference

D) nessuna delle precedenti

Annunci

Molto probabile **SEMINARIO** martedì 20/05 h.14-16 (aula da definire)

Video su datapath (a complemento della lezione in jigsaw) a breve disponibile su Moodle. E' parte integrante del materiale del corso.

Per chi ha partecipato alla *jigsaw* e una volta visto il video, *qui* trovate un breve questionario (anonimo) di feedback dell'attività svolta.

QUESTIONARIO DI METÀ CORSO (anonimo)

Materiale per la lezione

- Cormen et al. **CAP. 3.10**

Lab: portatevi il PC!

Prossima lezione: 24 ***aprile, h.9:00****,* ***aula 4C***