

# Algoritmi e Strutture di Dati

Tipi astratti di dato

(pile e code realizzate tramite array)

*m.patrignani*

# Nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

# Sommario

- Tipi astratti di dato
- Strutture di dati elementari
  - le pile
  - le code
- Realizzazione con array di queste strutture
- La complessità ammortizzata

# Tipo astratto di dato

- Un *tipo astratto di dato* (o ADT, abstract data type) è una descrizione di un tipo di dato indipendente dalla sua realizzazione in un linguaggio di programmazione
- Un tipo astratto di dato è costituito da:
  - i domini interessati
    - tra cui il dominio di interesse ed eventualmente altri domini di supporto
  - un insieme di costanti
  - una collezione di operazioni

# Il tipo astratto di dato pila

- Le pile (o stack) realizzano una strategia LIFO (last-in first-out)
- Tipo astratto: pila di interi
  - domini
    - il dominio di interesse è l'insieme delle pile  $P$  di interi
    - dominio di supporto: gli interi  $Z = \{0, 1, -1, 2, -2, \dots\}$
    - dominio di supporto: i booleani  $B = \{\text{true}, \text{false}\}$
  - costanti
    - la pila vuota

# Il tipo astratto di dato pila

- Tipo astratto: pila di interi
  - operazioni
    - verifica se una pila è vuota
$$\text{IS\_EMPTY}: P \rightarrow B$$
    - inserimento di un elemento nella pila
$$\text{PUSH}: P \times Z \rightarrow P$$
    - rimozione e restituzione dell'elemento affiorante della pila
$$\text{POP}: P \rightarrow P \times Z$$
  - operazioni aggiuntive
    - lettura dell'elemento affiorante della pila
$$\text{TOP}: P \rightarrow Z$$
    - svuotamento della pila
$$\text{EMPTY}: P \rightarrow P$$
    - numero degli elementi nella pila
$$\text{SIZE}: P \rightarrow Z$$

# Realizzazione di un tipo astratto di dato

- Un tipo astratto di dato può essere *realizzato* (o *implementato*) in uno specifico linguaggio di programmazione tramite la definizione di
  - tipi concreti o strutture dati nello specifico linguaggio di programmazione corrispondenti ai *domini* necessari
  - costrutti che consentono di codificare le *costanti*
  - funzioni che realizzano le *operazioni* previste
- Ovviamente uno stesso tipo astratto di dato può avere diverse realizzazioni con diverse proprietà

# Realizzazione di un ADT: osservazioni

- La definizione dei tipi concreti o delle strutture che corrispondono ai domini può comportare delle limitazioni
  - esempi di limitazioni
    - il dominio  $P$  delle pile viene ristretto alle pile di dimensione massima `maxsize`, nota a priori
    - il dominio  $Z$  degli interi viene realizzato tramite il tipo concreto `int` che ha un valore minimo `MININT` e massimo `MAXINT`
- Le costanti vengono spesso codificate tramite delle funzioni che ritornano la struttura corrispondente
  - esempio di costante:
    - la pila vuota viene realizzata tramite  
`CREATE-STACK(maxsize)`  
che ritorna il riferimento ad una pila vuota che potrà contenere al massimo `maxsize` elementi



# Funzioni che operano su pile

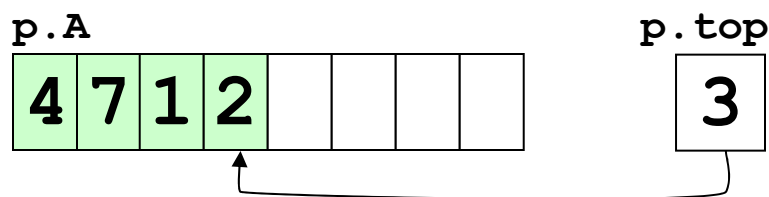
- `NEW_STACK(maxsize)`
  - ritorna il riferimento ad una pila vuota che potrà contenere al massimo `maxsize` elementi
- `IS_EMPTY(p)`
  - ritorna `true` se la pila è vuota, `false` altrimenti
- `PUSH(p, x)`
  - inserimento di un elemento nella pila
    - può dare un errore di “overflow” se l’implementazione prevede un numero massimo di elementi nella pila
- `POP(p)`
  - rimozione e restituzione dell’elemento affiorante della pila
    - dà un errore di “underflow” se la pila è vuota

# Altre funzioni su pile

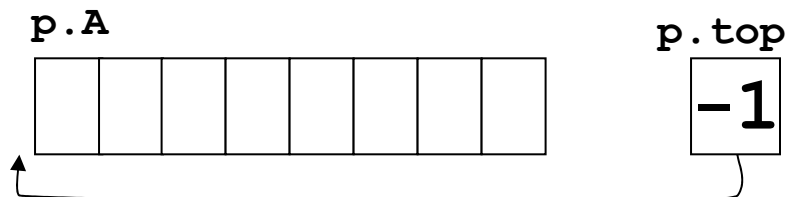
- $\text{TOP}(p)$ 
  - ritorna l'elemento affiorante senza rimuoverlo
    - può dare errore se la pila è vuota
- $\text{EMPTY}(p)$ 
  - svuota la pila
- $\text{SIZE}(p)$ 
  - ritorna il numero degli elementi in pila

# Realizzazione di una pila con array

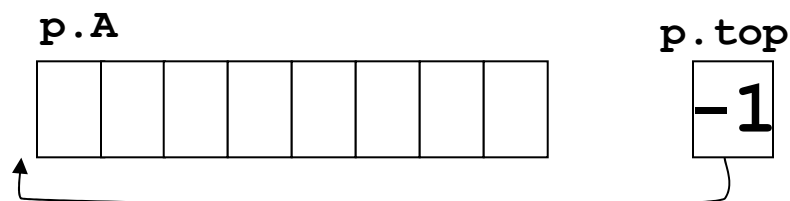
- Una pila  $p$  può essere realizzata tramite un oggetto contenente un array  $p.A$  e un intero  $p.top$  che specifica l'indice dell'elemento affiorante



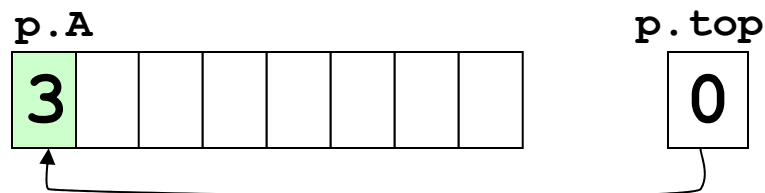
- Quando la pila è vuota  $p.top$  vale -1



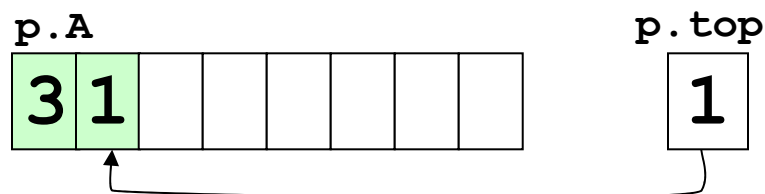
# Sequenza di operazioni su una pila



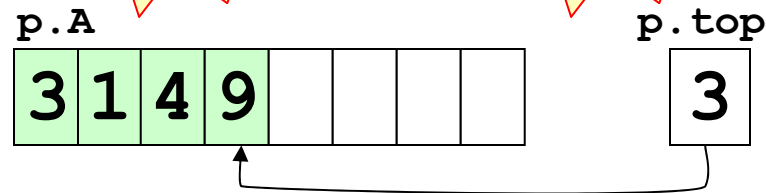
**PUSH(p,3)**



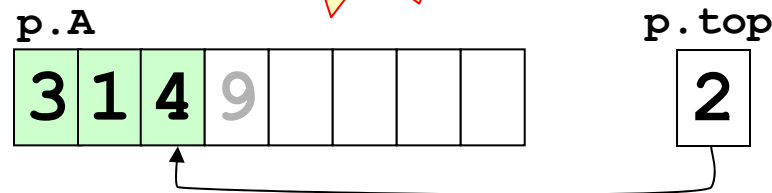
**PUSH(p,1)**



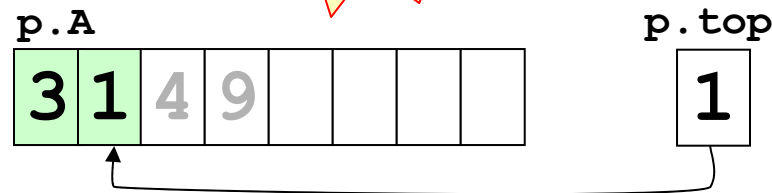
**PUSH(p,4)**      **PUSH(p,9)**



**POP(p)**



**POP(p)**



# Operazione NEW\_STACK

- Implementazione in pseudocodice della funzione di creazione

**NEW\_STACK(maxsize)**

1. ▷ creo un oggetto p con: p.A array di **maxsize** interi
2. ▷ p.top intero
3. p.top = -1
4. **return** p

- L'uso della funzione NEW\_STACK è il seguente

```
...  
p = NEW_STACK(10)    ▷ creo una pila con 10 posizioni  
...
```

# Operazioni PUSH e POP

- Implementazione delle funzioni di modifica

**PUSH(p, x)**

```
1. if p.top == p.A.length-1
2.     error("overflow")
3. else
4.     p.top = p.top + 1
5.     p.A[p.top] = x
```

**POP(p)**

```
1. if p.top == -1
2.     error("underflow")
3. else
4.     p.top = p.top - 1
5.     return p.A[p.top + 1]
```

# Altre operazioni sulle pile

**IS\_EMPTY (p)**

1. **return** p.top == -1      ▷ true se la pila è vuota

**EMPTY (p)**

1. p.top = -1      ▷ vuoto la pila

**TOP (p)**

1. **return** p.A[p.top]      ▷ l'elemento affiorante

**SIZE (p)**

1. **return** p.top + 1      ▷ il numero di elementi

- Il tempo di esecuzione di ogni operazione è  $\Theta(1)$

# Gestione telescopica della pila

- Strategia utilizzata per non avere limiti sulla dimensione della pila
- Si adotta una dimensione iniziale di default
  - può essere un numero fissato dal programmatore
    - per esempio 128 posizioni
  - può essere il numero di celle specificato dall'utente nella funzione `NEW_STACK`
- Quando viene eseguita una `PUSH` sulla pila piena si raddoppia la dimensione della pila corrente per poter inserire ulteriori elementi



# Nuova funzione PUSH

- Questa funzione PUSH raddoppia la dimensione dell'array `p.A` quando l'array è pieno

**PUSH (p, x)**

1. **if** `p.top == p.A.length-1`

2.     ▷ B nuovo array di `2*p.A.length` posizioni

3.     **for** `i = 0 to p.A.length-1`

4.         `B[i] = p.A[i]`     ▷ copio `p.A` dentro `B`

5.     `p.A = B`             ▷ sostituisco `B` a `p.A`

6.     `p.top = p.top + 1`   ▷ inserisco l'elemento `x`

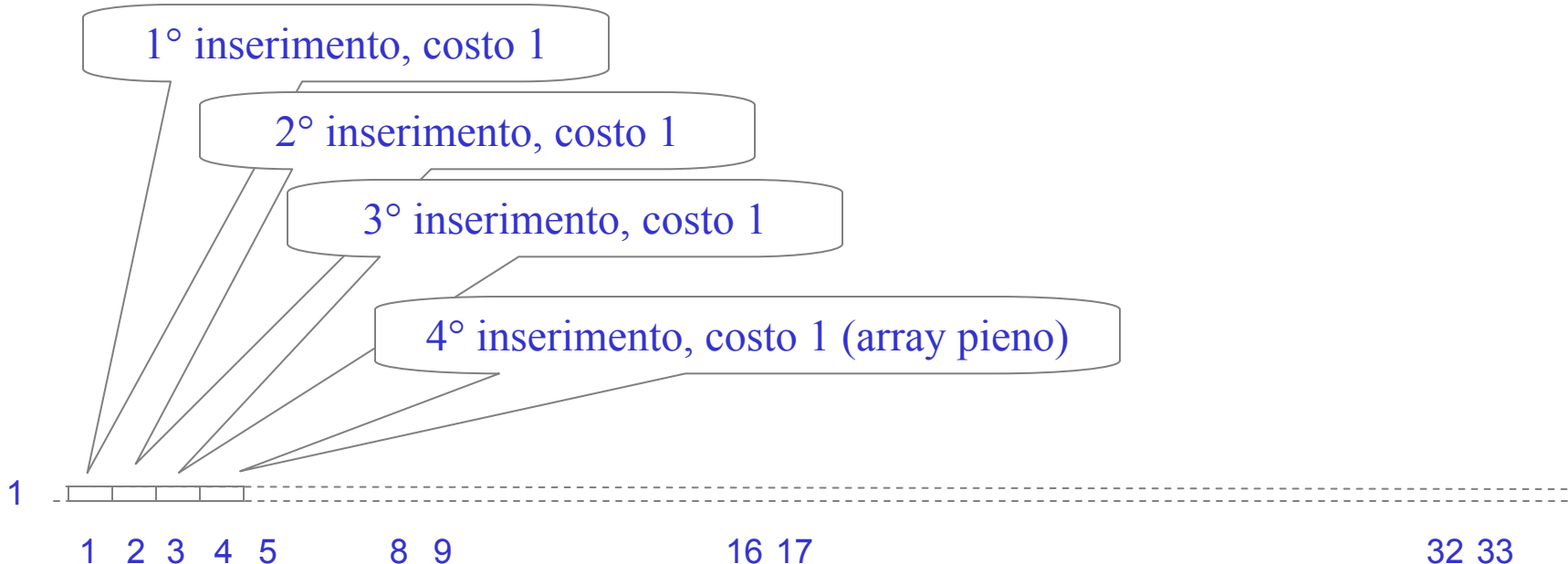
7.     `p.A[p.top] = x`

# Analisi ammortizzata

- Calcoliamo il costo degli inserimenti partendo da un array da 4 elementi

1	2	3	4
---	---	---	---

- Diagramma dei costi:



# Analisi ammortizzata

- Calcoliamo il costo degli inserimenti partendo da un array da 4 elementi



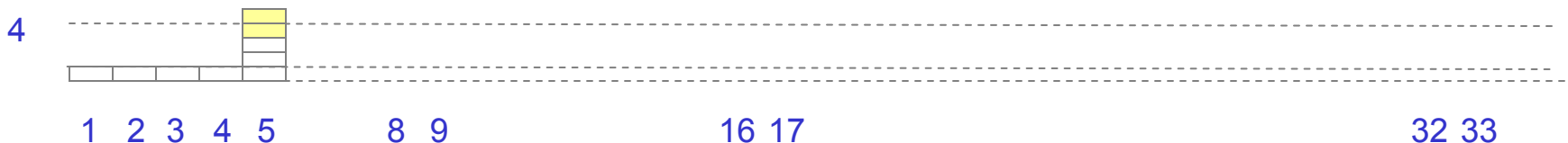
5

5° inserimento

- Diagramma dei costi:

raddoppio la dimensione dell'array e ci copio dentro i primi 4 elementi

5° inserimento, costo 5  
(4 per la copia, 1 per l'inserimento)

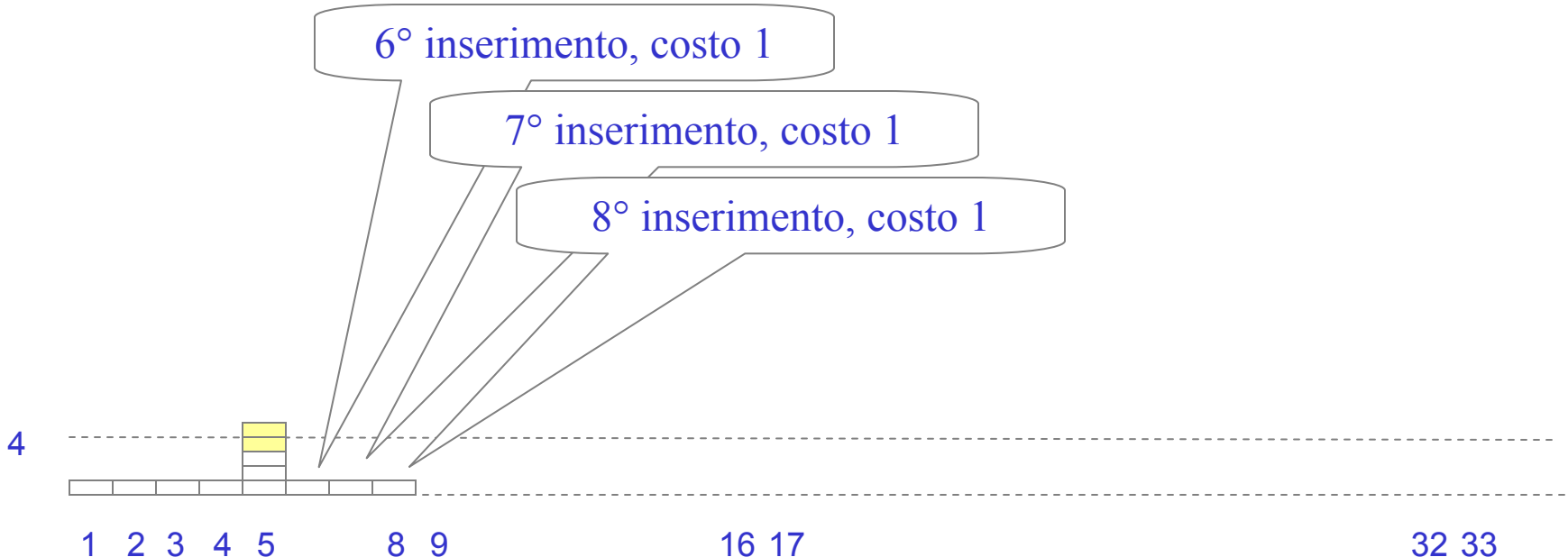


# Analisi ammortizzata

- Calcoliamo il costo degli inserimenti partendo da un array da 4 elementi

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---

- Diagramma dei costi:



# Analisi ammortizzata

- Calcoliamo il costo degli inserimenti partendo da un array da 4 elementi



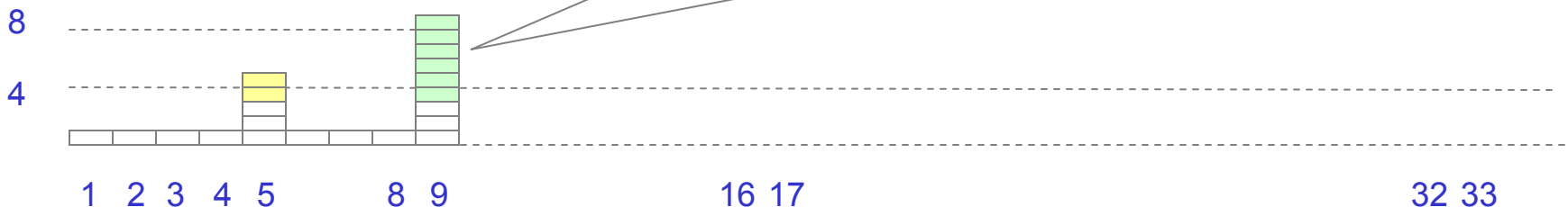
9

9° inserimento

- Diagramma dei costi:

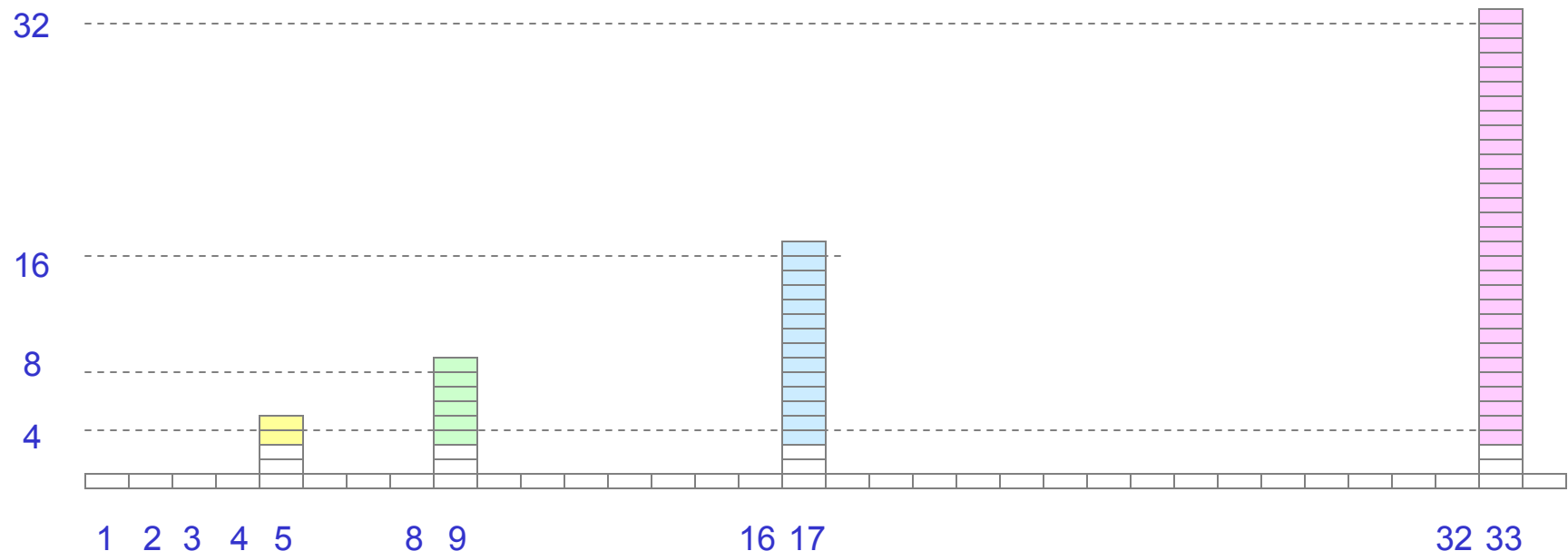
raddoppio la dimensione dell'array e ci copio dentro i primi 8 elementi

9° inserimento, costo 9  
(8 per la copia, 1 per l'inserimento)



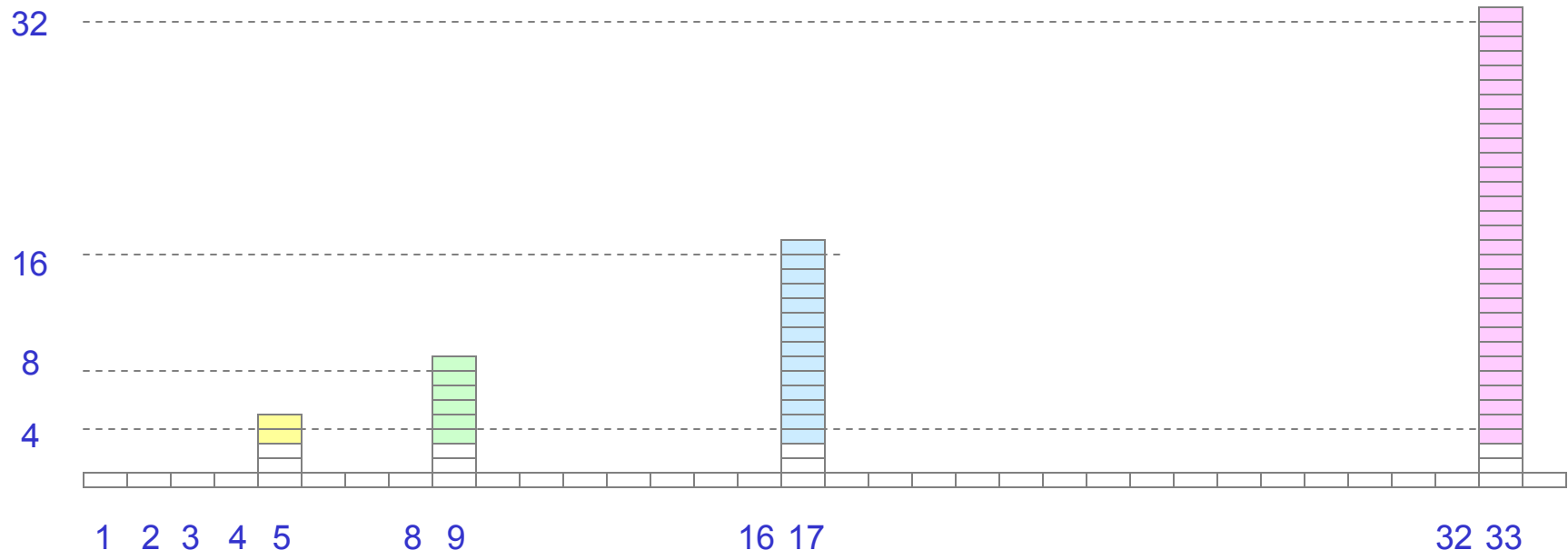
# Analisi ammortizzata

- Calcoliamo il costo degli inserimenti partendo da un array da 4 elementi
- Diagramma dei costi dopo molti inserimenti



# Analisi ammortizzata

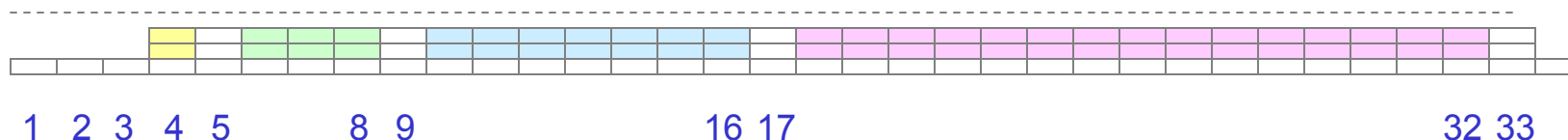
- Calcoliamo il costo degli inserimenti partendo da un array da 4 elementi
- Diagramma dei costi dopo molti inserimenti
- Distribuiamo i costi sugli inserimenti



# Complessità ammortizzata

- Alcuni inserimenti costano effettivamente  $\Theta(n)$ , dove  $n$  è il numero di elementi già contenuti nella pila
  - è corretto dire che il costo di un inserimento nel caso peggiore è  $\Theta(n)$
- Una sequenza di  $n$  inserimenti costa  $\Theta(n)$ 
  - si dice che la complessità “ammortizzata” di un inserimento è  $\Theta(1)$

4





# Uso delle realizzazioni di un ADT

- Dopo aver realizzato un ADT possiamo mettere l'implementazione a disposizione di altri programmatori specificando
  - le eventuali limitazioni della realizzazione
  - l'elenco delle operazioni supportate e la loro complessità asintotica o ammortizzata
- Esempio
  - si dispone di una realizzazione di una pila con le seguenti funzioni e complessità nel caso peggiore
    - `NEW_STACK(maxsize)`  $\Theta(1)$
    - `IS_EMPTY(p)`  $\Theta(1)$
    - `PUSH(p, x)`  $\Theta(1)$
    - `POP(p)`  $\Theta(1)$
    - `EMPTY(p)`  $\Theta(1)$

# Il tipo astratto di dato coda

- Le code (o queue) realizzano una strategia FIFO (first-in first-out)
- Tipo astratto: coda di interi
  - domini
    - il dominio di interesse è l'insieme delle code  $Q$  di interi
    - dominio di supporto: gli interi  $Z = \{0, 1, -1, 2, -2, \dots\}$
    - dominio di supporto: i booleani  $B = \{\text{true}, \text{false}\}$
  - costanti
    - la coda vuota

# Il tipo astratto di dato coda

- Tipo astratto: coda di interi

- operazioni

- verifica se una coda è vuota

$$\text{IS\_EMPTY: } Q \rightarrow B$$

- inserimento di un elemento nella coda

$$\text{ENQUEUE: } Q \times Z \rightarrow Q$$

- rimozione e restituzione dell'elemento più vecchio della coda

$$\text{DEQUEUE: } Q \rightarrow Q \times Z$$

- operazioni aggiuntive

- lettura dell'elemento più vecchio della coda

$$\text{FRONT: } Q \rightarrow Z$$

- svuotamento della coda

$$\text{EMPTY: } Q \rightarrow Q$$

- numero degli elementi nella coda

$$\text{SIZE: } Q \rightarrow Z$$

# Realizzazione dell'ADT coda

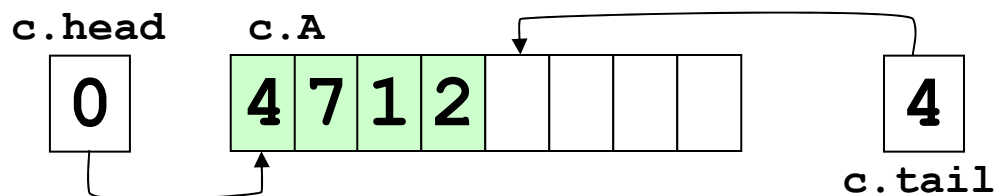
- `NEW_QUEUE(maxsize)`
  - ritorna un riferimento ad una coda vuota che può contenere al massimo `maxsize` interi
- `IS_EMPTY(c)`
  - ritorna `true` se la coda è vuota, altrimenti ritorna `false`
- `ENQUEUE(c, x)`
  - inserisce un elemento `x` nella coda
    - può dare un errore di “overflow” se l’implementazione prevede un numero massimo di elementi
- `DEQUEUE(c)`
  - rimuove l’elemento più vecchio della coda e lo restituisce
    - dà un errore di “underflow” se la coda è vuota

# Altre operazioni sulle code

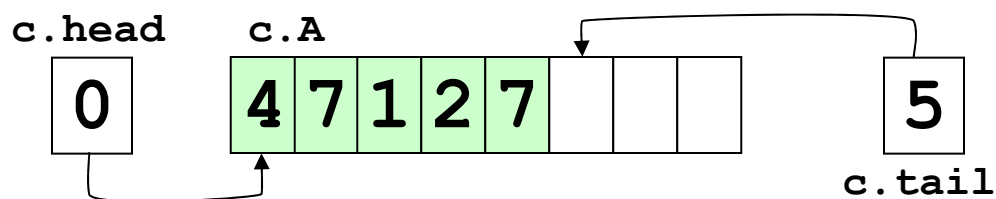
- FRONT (c)
  - ritorna l'elemento più vecchio senza rimuoverlo
    - può dare errore se la coda è vuota
- EMPTY (c)
  - svuota la coda
- SIZE (c)
  - ritorna il numero degli elementi in coda

# Realizzazione di una coda tramite array

- Una coda  $c$  può essere realizzata con un array  $c.A$  arricchito da due attributi  $c.head$  e  $c.tail$  che contengono gli indici dell'elemento più vecchio e della prima posizione utile

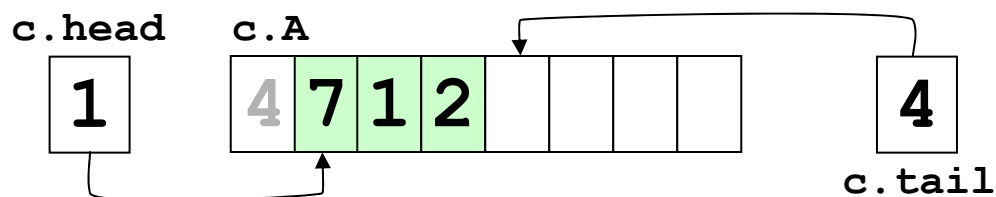
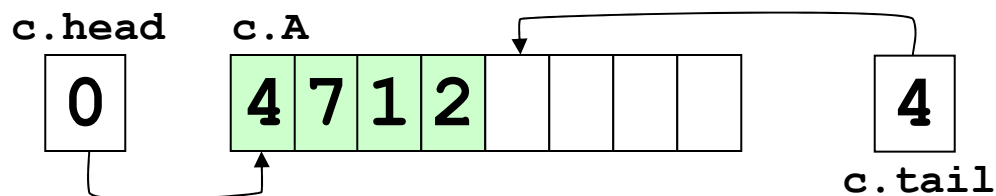


- Un nuovo elemento viene aggiunto da ENQUEUE nella posizione  $c.tail$  (che viene incrementato)

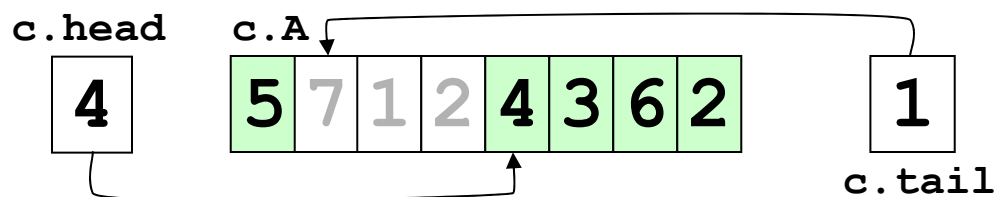


# Realizzazione di una coda tramite array

- L'elemento ritornato da DEQUEUE è quello in posizione `c.head` (che viene incrementato)

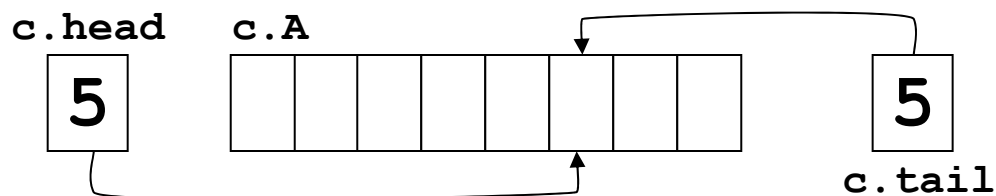


- L'array è gestito come una lista circolare

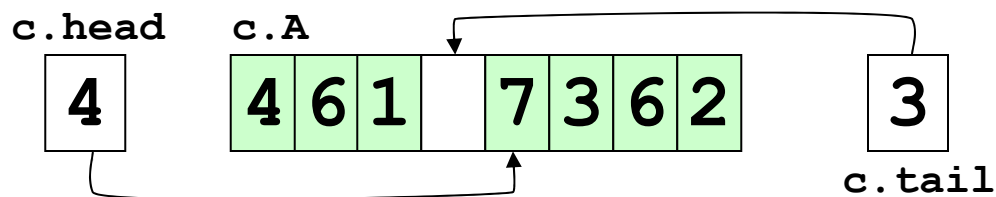


# Realizzazione di una coda tramite array

- La coda è vuota quando `c.head` e `c.tail` puntano alla stessa casella



- La coda non può avere più di  $n-1$  elementi
  - se avesse  $n$  elementi, che valore potremmo dare a `c.tail` senza creare equivoco con la coda vuota?





# Operazioni ENQUEUE e DEQUEUE

## ENQUEUE (c, x)

```
1.  if c.head==c.tail+1 or (c.tail==c.A.length-1 and c.head==0)
2.      error("overflow")
3.  else  c.A[c.tail] = x
4.      if c.tail == c.A.length-1
5.          c.tail = 0
6.      else  c.tail = c.tail + 1
```

## DEQUEUE (c)

```
1.  if c.head == c.tail
2.      error("underflow")
3.  else  x = c.A[c.head]
4.      if c.head == c.A.length-1
5.          c.head = 0
6.      else  c.head = c.head + 1
7.      return x
```

# Altre operazioni sulle code

**IS\_EMPTY(c)**

```
1. return c.head == c.tail
```

**EMPTY(c)**

```
1. c.head = c.tail = 0
```

**FRONT(c)**

```
1. if c.head == c.tail
2.     error("empty queue")
3. else
4.     return c.A[c.head]
```

- Il tempo di esecuzione di ogni operazione è  $\Theta(1)$

# Esercizi sulle code

1. Scrivi lo pseudocodice della procedura `NEW_QUEUE()` che restituisce il riferimento ad una coda vuota
2. Scrivi lo pseudocodice della procedura `SIZE(c)` che restituisce il numero di elementi in una coda
3. Scrivi lo pseudocodice della procedura `ENQUEUE (q, x)` che abbia complessità ammortizzata  $\Theta(1)$

# Esercizi su pile e code

4. Supponi di disporre di una realizzazione di una pila con le seguenti caratteristiche

- funzioni `NEW_STACK`, `IS_EMPTY` e `POP` di complessità  $\Theta(1)$
- funzione `PUSH` di complessità nel caso peggiore  $\Theta(n)$  e complessità ammortizzata  $\Theta(1)$

Descrivi come sia possibile implementare un coda utilizzando esclusivamente pile

- ti occorreranno due pile `p1` e `p2` da usare simultaneamente
- l'operazione `NEW_QUEUE` creerà le due pile `p1` e `p2`
- le operazioni `ENQUEUE` e `DEQUEUE` si tradurranno in opportune operazioni di `PUSH` e `POP` sulle due pile `p1` e `p2`

Discuti la complessità delle operazioni `ENQUEUE` e `DEQUEUE`

# Esercizi su pile e code

5. Supponi di disporre di una realizzazione di una coda con le seguenti caratteristiche
- funzioni `NEW_QUEUE`, `IS_EMPTY` e `DEQUEUE` di complessità  $\Theta(1)$
  - funzione `ENQUEUE` di complessità nel caso peggiore  $\Theta(n)$  e complessità ammortizzata  $\Theta(1)$

Descrivi come sia possibile implementare un pila utilizzando esclusivamente code

- ti occorreranno due code `q1` e `q2` da usare simultaneamente
- le operazioni `PUSH` e `POP` si tradurranno in opportune operazioni di `ENQUEUE` e `DEQUEUE` sulle due code `q1` e `q2`

Discuti la complessità delle operazioni `PUSH` e `POP`

# Esercizi su pile e code

6. Scrivi lo pseudocodice della procedura `CONTAINS (p, x)` che ritorna `true` se l'elemento `x` è contenuto nella pila `p` e ritorna `false` se `x` non è contenuto (lasciando la pila invariata)
7. Scrivi lo pseudocodice di una struttura dati in cui si possa inserire/rimuovere elementi sia in testa che in coda
  - deve avere contemporaneamente `PUSH`, `POP`, `DEQUEUE`, `ENQUEUE` (dove evidentemente `ENQUEUE` è uguale a `PUSH`)