

# Analisi Ammortizzata

Jocelyne Elias

<https://www.unibo.it/sitoweb/jocelyne.elias>

**Moreno Marzolla**

<https://www.moreno.marzolla.name/>

Dipartimento di Informatica—Scienza e Ingegneria (DISI)  
Università di Bologna

Copyright © Alberto Montresor, Università di Trento, Italy

<http://cricca.disi.unitn.it/montresor/teaching/asd/>

Copyright © 2021 Moreno Marzolla, Università di Bologna, Italy

<https://www.moreno.marzolla.name/teaching/ASD/>



*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.*

Ci stiamo iscrivendo alla palestra di Buff. Buff richiede una quota di iscrizione di 60 euro al mese, più 3 euro per ogni accesso alla palestra. Poiché siamo disciplinati, andiamo alla palestra di Buff ogni giorno per tutto il mese di novembre. Oltre alla quota mensile di 60 euro, paghiamo altri  $3 \times 30 = 90$  euro per gli accessi del mese di novembre. Anche se possiamo immaginare di suddividere le spese in una tariffa fissa di 60 euro a cui aggiungere 90 euro di spese giornaliere, possiamo anche considerarle in un altro modo. Complessivamente, pagheremo 150 euro in 30 giorni, ovvero una media di 5 euro al giorno. In questo modo, stiamo ammortizzando la tariffa mensile sui 30 giorni del mese, distribuendola 2 euro al giorno

La stessa cosa si può fare quando si analizzano i tempi di esecuzione. Nell'analisi armonizzata, facciamo una media del tempo richiesto per eseguire una sequenza di operazioni su una struttura dati rispetto al numero di tutte le operazioni eseguite.

- L'analisi armonizzata differisce dall'analisi nel caso medio perché non richiede la teoria di probabilità. Infatti un'analisi armonizzata valuta le prestazioni medie di ciascuna operazione nel caso peggiore

Ricordo che i costi assegnati durante l'analisi ammortizzata servono esclusivamente all'analisi, nel senso che non devono, e nemmeno dovrebbero apparire nel codice.

# Introduzione

- **Analisi ammortizzata**
  - Si considera il tempo richiesto nel caso peggiore per eseguire una sequenza di operazioni
- **Sequenza**
  - Operazioni costose e meno costose
  - Se le operazioni più costose sono poco frequenti, allora il loro costo può essere compensato ("ammortizzato") da quelle meno costose
- **Importante differenza**
  - **Analisi ammortizzata:**
    - **deterministica**, su operazioni multiple (una sequenza di operazioni), caso pessimo
  - **Analisi del caso medio:**
    - **basata su probabilità**, su singola operazione

# Metodi per l'analisi ammortizzata

- **Metodo dell'aggregazione**
  - Si calcola la complessità  $O(f(n))$  per eseguire  $n$  operazioni in sequenza nel caso pessimo
  - Il costo ammortizzato di una singola operazione è  $O(f(n)/n)$
- **Metodo degli accantonamenti** (o del contabile)
  - Alle operazioni vengono assegnati costi ammortizzati che possono essere maggiori/minori del loro costo effettivo
  - Si deve dimostrare che la somma dei costi ammortizzati è un limite superiore al costo effettivo
- **Metodo del potenziale**
  - Lo stato del sistema viene descritto tramite differenze di potenziale (*non lo vediamo*)

# Esempio 1: Pila con `multipop()`

- Una pila con le solite operazioni...
  - `push(x)` aggiunge `x` in cima alla pila
  - `pop()` rimuove l'elemento che sta in cima alla pila
  - `top()` restituisce l'elemento in cima alla pila senza rimuoverlo
  - `isEmpty()` *true* se la pila è vuota, *false* altrimenti
- ...più una nuova operazione `multipop(k)` che
  - o rimuove i `k` elementi in cima,
  - o svuota la pila se contiene meno di `k` elementi

```
algoritmo multipop(integer k)
    while (not isEmpty()) and (k > 0) do
        pop()
        k ← k - 1
    endwhile
```

# Esempio 1: analisi grossolana

- Se la pila contiene  $m$  elementi il ciclo *while* è iterato  $\min(m, k)$  volte e quindi `multipop(k)` ha complessità  $O(\min(m, k))$
- Consideriamo una sequenza di  $n$  operazioni eseguite a partire dalla pila vuota
  - mix di `push()`, `pop()`, `multipop()`
- L'operazione più costosa `multipop()` richiede tempo  $O(n)$  nel caso pessimo
- Moltiplicando per  $n$  otteniamo il limite superiore  $O(n^2)$  per il costo della sequenza di  $n$  operazioni
  - da cui il costo ammortizzato sarebbe  $O(n^2 / n) = O(n)$

# Metodo dell'aggregazione



- Considerazioni per un'analisi più accurata
  - Un elemento può essere tolto solo dopo essere stato inserito
  - Quindi il numero totale di `pop()` (comprese quelle nella `multi-pop()`) non può superare il numero totale di `push()`
  - Quindi il numero totale di `pop()` è sicuramente minore di  $n$
- **Nota** (importante per il metodo di aggregazione)
  - Questa proprietà è vera per qualunque sequenza di qualunque lunghezza  $n$
  - In altre parole, stiamo considerando il caso pessimo



# Metodo dell'aggregazione

- Metodo dell'aggregazione
  - Costo per eseguire i **pop()** all'interno di tutte le **multipop()**: minore di  $n$ , quindi  $O(n)$
  - Costo per eseguire le altre operazioni (**push**, **top**, **isEmpty**, ...), qualunque esse siano:  $O(n)$
  - Costo totale:  $O(n) + O(n) = O(n)$
- Costo ammortizzato:
  - $O(n) / n = O(1)$

→ Per sommare 1 (modulo  $2^k$ ) al valore del contatore, utilizziamo la procedura INCREMENT

## Esempio 2: contatore binario



- Contatore binario

- Array  $A[0..k-1]$  di bit
- La rappresentazione binaria di  $x$  ha il bit meno significativo in  $A[0]$  e quello più significativo in  $A[k-1]$

$$x = \sum_{i=0}^{k-1} 2^i \times A[i]$$

- Supponiamo che  $A$  venga usato per contare a partire da  $x = 0$  usando l'operazione di incremento

```
algoritmo incrementa(A[0..k-1])
  i ← 0
  while i < k and A[i] = 1 do
    A[i] ← 0
    i ← i + 1
  endwhile
  if i < k then
    A[i] ← 1
  endif
```

# Esempio 2: funzionamento

x	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Costo	Totale
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1
2	0	0	0	0	1	0	2	3
3	0	0	0	0	1	1	1	4
4	0	0	0	1	0	0	3	7
5	0	0	0	1	0	1	1	8
6	0	0	0	1	1	0	2	10
7	0	0	0	1	1	1	1	11
8	0	0	1	0	0	0	4	15
9	0	0	1	0	0	1	1	16
10	0	0	1	0	1	0	2	18
11	0	0	1	0	1	1	1	19
12	0	0	1	1	0	0	3	22
13	0	0	1	1	0	1	1	23
14	0	0	1	1	1	0	2	25
15	0	0	1	1	1	1	1	26
16	0	1	0	0	0	0	5	31

Analisi Ammortizzata

# Esempio 2

- Analisi “grossolana”
  - Una singola operazione di incremento richiede tempo  $O(k)$  nel caso pessimo (per un qualche  $k$ )
  - Limite superiore  $O(nk)$  per una sequenza di  $n$  incrementi
- Considerazioni per un'analisi più accurata
  - Il tempo necessario ad eseguire l'intera sequenza è proporzionale al numero di bit che vengono modificati
  - Quanti bit vengono modificati complessivamente?

```
algoritmo incrementa(A[0..k-1])  
  i ← 0  
  while i < k and A[i] = 1 do  
    A[i] ← 0  
    i ← i + 1  
  endwhile  
  if i < k then  
    A[i] ← 1  
  endif
```

# Esempio 2: funzionamento

x	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Costo	Totale
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	1	1	1
2	0	0	0	0	1	0	2	3
3	0	0	0	0	1	1	1	4
4	0	0	0	1	0	0	3	7
5	0	0	0	1	0	1	1	8
6	0	0	0	1	1	0	2	10
7	0	0	0	1	1	1	1	11
8	0	0	1	0	0	0	4	15
9	0	0	1	0	0	1	1	16
10	0	0	1	0	1	0	2	18
11	0	0	1	0	1	1	1	19
12	0	0	1	1	0	0	3	22
13	0	0	1	1	0	1	1	23
14	0	0	1	1	1	0	2	25
15	0	0	1	1	1	1	1	26
16	0	1	0	0	0	0	5	31

← k →

Analisi Ammortizzata

# Esempio 2

## Metodo dell'aggregazione

- Dalla simulazione si vede che:
  - $A[0]$  viene modificato ad ogni incremento del contatore,
  - $A[1]$  viene modificato ogni due incrementi,
  - $A[2]$  ogni 4 incrementi....
- $A[i]$  viene modificato ogni  $2^i$  incrementi
- Su una sequenza di  $n$  operazioni,  $A[i]$  viene modificato  $n/2^i$  volte

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = 2$$

- Quindi:
  - Costo aggregato:  $\sum_{i=0}^{k-1} \frac{n}{2^i} < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$
  - Costo ammortizzato:  $2n / n = 2 = \underline{\underline{O(1)}}$

# Metodo degli accantonamenti o del contabile

Costo di un operazione → costo ammortizzato

- Si assegna un **costo ammortizzato** ad ognuna delle operazioni possibili
  - Nel metodo dell'aggregazione abbiamo calcolato un **costo ammortizzato costante** (  $O(1)$  )

- Il **costo ammortizzato può essere diverso dal costo effettivo**

*Quando il costo ammortizzato di un'operazione supera il costo effettivo, si assegna la differenza a specifici oggetti nella struttura dati sottoforma di credito. Il credito potrà essere utilizzato successivamente per contribuire a pagare le operazioni il cui costo ammortizzato è minore del costo effettivo.*

- Le operazioni meno costose vengono caricate di un costo aggiuntivo detto **credito**
  - **costo ammortizzato** = **costo effettivo** + **credito prodotto**
- I crediti accumulati saranno usati per pagare le operazioni più costose
  - **costo ammortizzato** = **costo effettivo** – **credito consumato**

# Come assegnare costi ammortizzati?

- Ricordate che lo scopo è:
  - dimostrare che la somma dei costi ammortizzati  $\hat{c}_i$  è un limite superiore alla somma dei costi effettivi  $c_i$ :

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i$$

- Alcuni punti da ricordare
  - La dimostrazione deve essere valida per tutte le sequenze di input (caso pessimo)



# Esempio 1 (multipop)

## Metodo degli accantonamenti

Costi effettivi  $c$

<code>push()</code>	1
<code>pop()</code>	1
<code>multipop()</code>	$\min(k, m)$

Costi ammortizzati  $\hat{c}$

<code>Push()</code>	2 (1+1)
<code>Pop()</code>	0 (1-1)
<code>multipop()</code>	0 (1-1)

$\text{Push}() \rightarrow 1$  • paga il costo effettivo dell'inserimento  
1 è credito associato all'elemento appena inserito. Questo credito sarà usato per pagare la sua eventuale rimozione futura

### • Costi ammortizzati:

#### – `push()`:

- una unità per pagare il costo effettivo,
- una unità come credito associato all'elemento inserito

$\text{Pop}(), \text{multiPop}() \rightarrow$  Ogni credito rimosso ha già un credito di 1 (assegnato al momento del `push`), quindi il costo è già coperto  $\rightarrow 0$

#### – `pop()`, `multipop()`:

- usa l'unità di costo associata all'elemento da estrarre
- quindi hanno costo ammortizzato zero

# Esempio 1 (multipop)

## Metodo degli accantonamenti

- Dimostrazione:
  - qualunque sia la sequenza, ad ogni **pop()** corrisponde una **push()**
  - L'operazione **push()** ha pagato un credito per se stessa, e un credito per la **pop()** che eliminerà quell'elemento
  - il numero di elementi è non-negativo, quindi anche il credito è non-negativo
  - Caso peggiore: facciamo solo **push()**  $\Rightarrow n$  **push()**
    - il costo totale ammortizzato è  $2 \cdot n = O(n)$
  - Costo ammortizzato per singola operazione:  $O(n/n) = O(1)$

→ Perché push genera ogni volta 2 crediti

# Esempio 2 (contatore binario)

## Metodo degli accantonamenti

- Costo **effettivo** dell'operazione `increment()`:  $d$   
(dove  $d$  è il numero di bit che cambiano valore)
- Costo **ammortizzato** dell'operazione `increment()`:  $2$ 
  - 1 per cambio del bit da 0 a 1 (costo effettivo)
  - 1 per il futuro cambio dello stesso bit da 1 a 0
- Ne segue che:
  - in ogni istante, il credito è pari al numero di bit 1 attualmente presenti
- Costo totale ammortizzato:  $O(n)$

# Esempio 2 (contatore binario)

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

0	0	0	0	0	0	0	1
---	---	---	---	---	---	---	---

~~€~~  
€

0	0	0	0	0	0	1	0
---	---	---	---	---	---	---	---

~~€~~  
€ ~~€~~

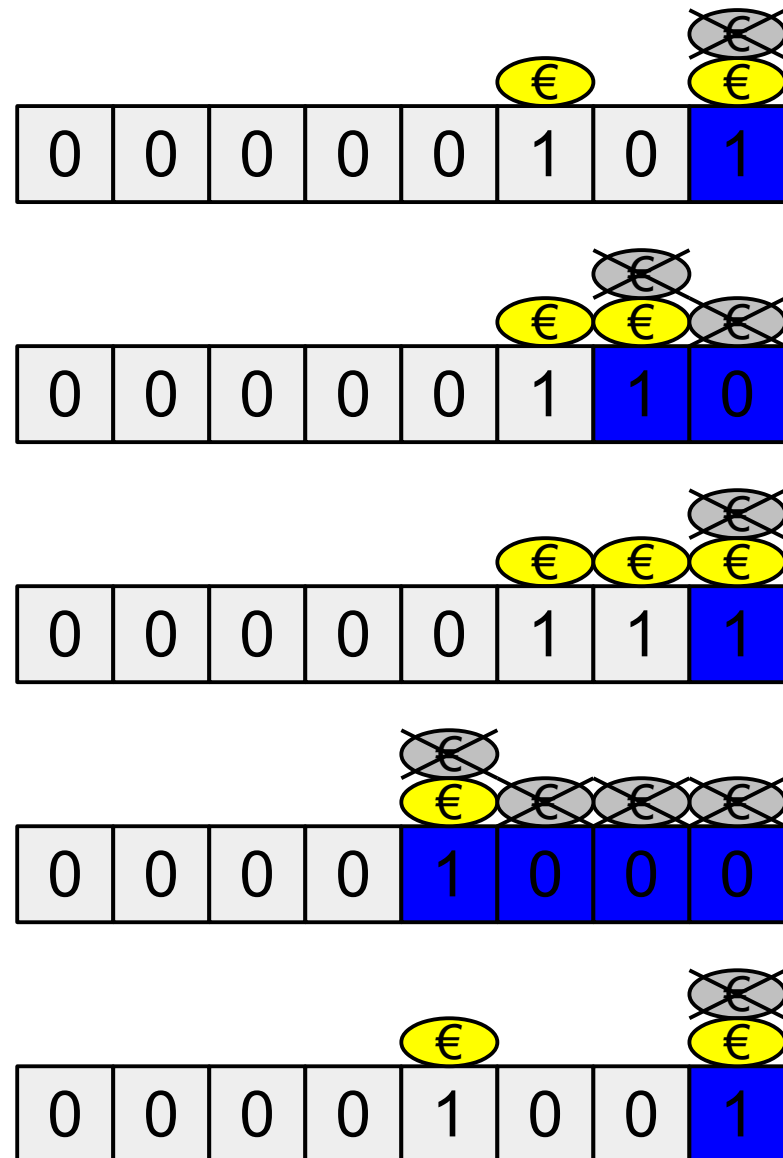
0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

~~€~~  
€ €


0	0	0	0	0	1	0	0
---	---	---	---	---	---	---	---

~~€~~  
€ ~~€~~ ~~€~~

# Esempio 2 (contatore binario)



# Array dinamici

- Un esempio più utile:
  - Spesso non si conosce a priori quanta memoria serve per memorizzare un array (tabella hash, heap, stack, ecc.)
  - Si alloca una certa quantità di memoria, per poi accorgersi che non basta
- Soluzione *→ soluzione manuale?*
  - Si alloca un buffer maggiore, si ricopia il contenuto del vecchio buffer nel nuovo e si rilascia il vecchio buffer
- Esempi
  - `java.util.Vector`, `java.util.ArrayList` 
- Vediamo un esempio con uno `stack` (pila)

# Interfaccia Pila

```
public interface Pila {  
    /**  
     * Verifica se la pila è vuota.  
     */  
    public boolean isEmpty();  
    /**  
     * Aggiunge l'elemento in cima  
     */  
    public void push(Object e);  
    /**  
     * Restituisce l'elemento in cima  
     */  
    public Object top();  
    /**  
     * Cancella l'elemento in cima  
     */  
    public Object pop();  
}
```

# Implementare una pila tramite array

```
public class PilaArray implements Pila
{
    private Object[] S = new Object[1];
    private int n = 0;

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Object e) { ... }
    public Object top() { ... }
    public Object pop() { ... }
}
```



# PilaArray: metodo top()

```
public Object top()
{
    if (this.isEmpty())
        throw new EccezioneStrutturaVuota("Pila vuota");
    return S[n - 1];
}
```

**Costo:  $O(1)$**

# PilaArray: metodo push()

```
public void push(Object e)
```

```
{
```

```
    if (n == S.length) {
```

```
        Object[] temp = new Object[2 * S.length];
```

```
        for (int i = 0; i < n; i++)
```

```
            temp[i] = S[i];
```

```
        S = temp;
```

```
    }
```

```
    S[n] = e;
```

```
    n = n + 1;
```

```
}
```

Raddoppio la  
dimensione della Pila

Il numero di  
elementi  $n$  è  
uguale alla  
capacità del  
vettore?

==>

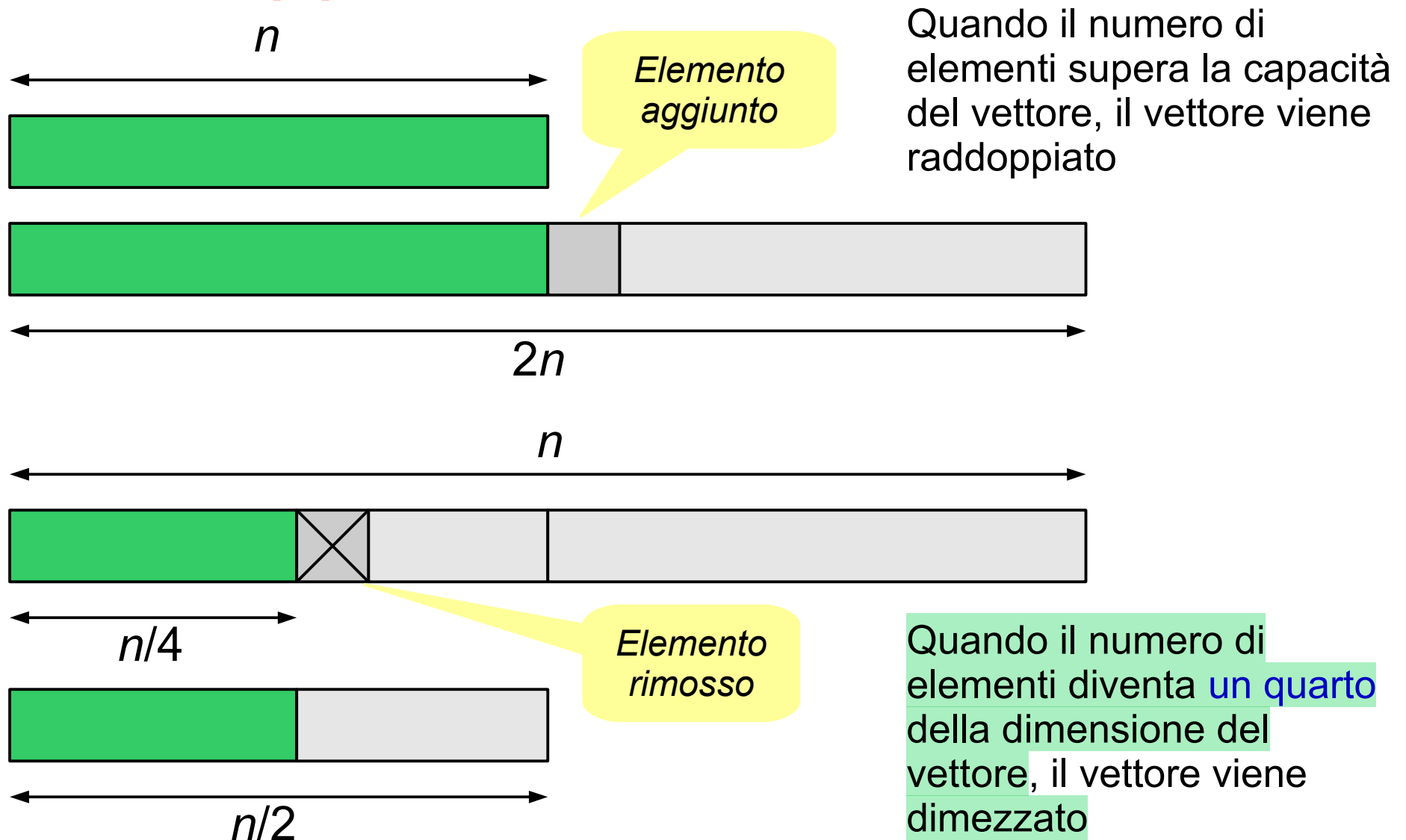
Raddoppio!

Caso  
Sfortunato  
(abbiamo  
ricuperto la  
Pila)

Caso "base"

Costo: ???  
 $O(1)/O(n)$

# Metodo del raddoppiamento/dimezzamento



# PilaArray: metodo pop ()

Prima rimuovo poi dimezzo, se necessario

```
public Object pop()
{
    if (this.isEmpty())
        throw new EccezioneStrutturaVuota("Pila vuota");
    n = n - 1;  $\rightarrow$  rimuovo
    Object e = S[n];
    if (n > 1 && n <= S.length / 4) {
        Object[] temp = new Object[S.length / 2];
        for (int i = 0; i < n; i++)
            temp[i] = S[i];
        S = temp;
    }
    return e;
}
```

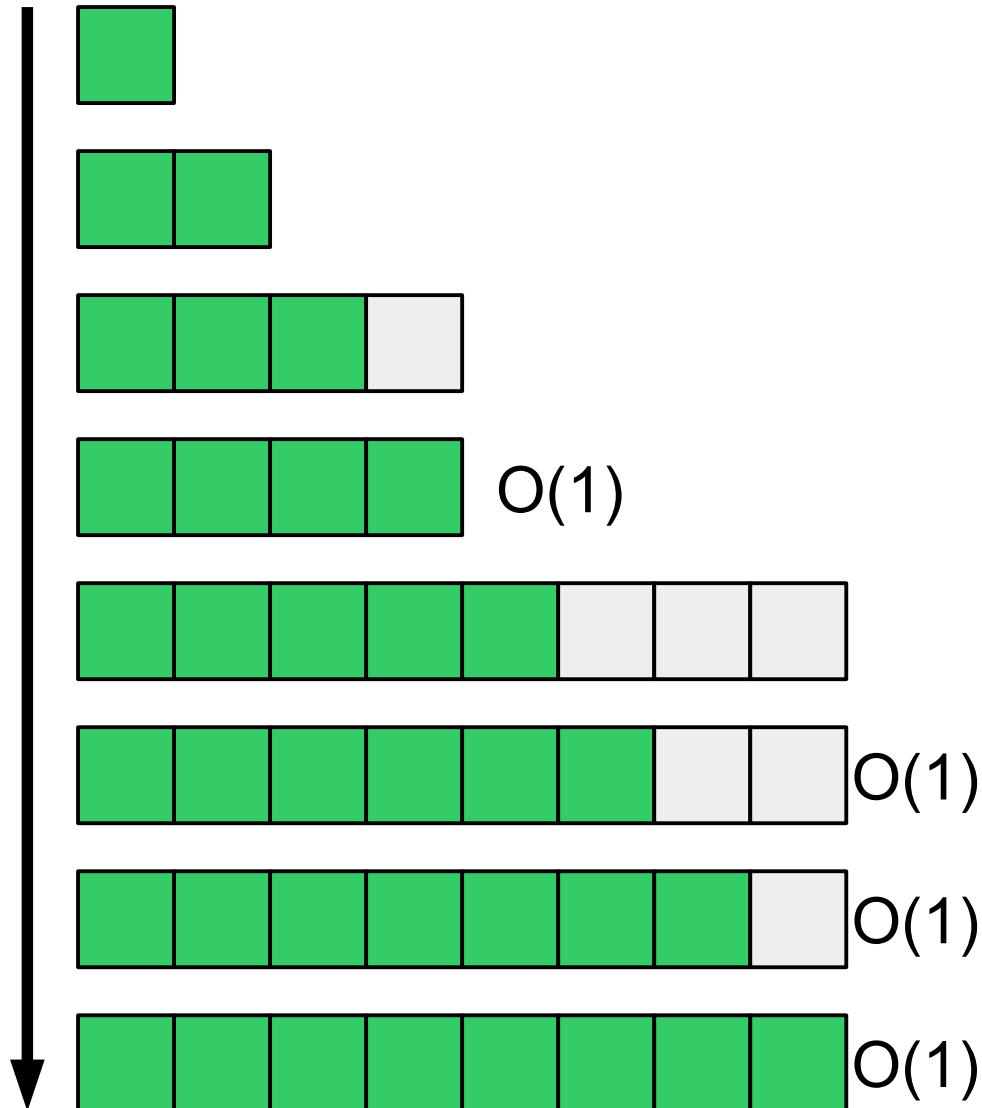
Dimezzo

Il numero di elementi  $n$  è inferiore o uguale a un quarto della **capacità** del vettore?  
 $\Rightarrow$   
Dimezzo!

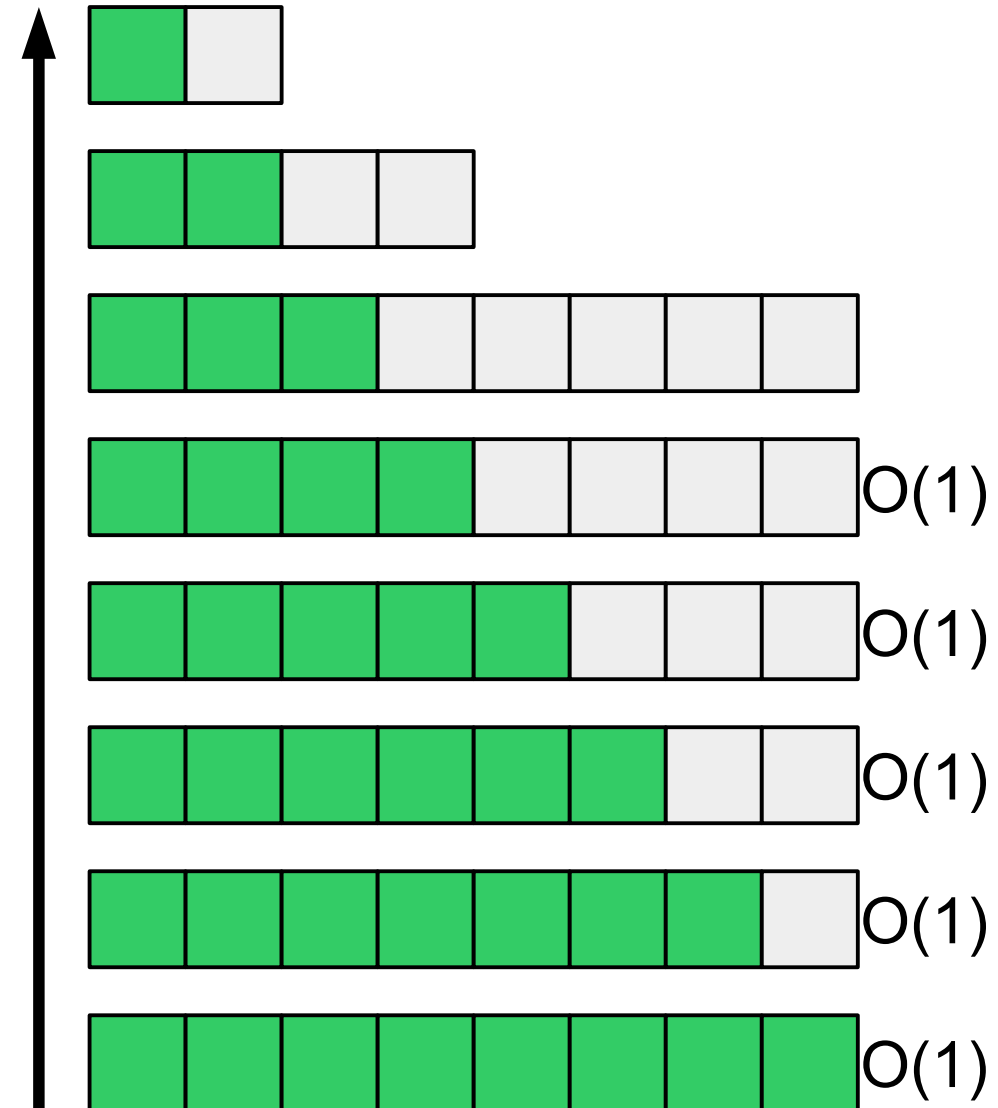
Costo: ???  
 $O(1)/O(n)$

# Analisi delle operazioni `push()` e `pop()`

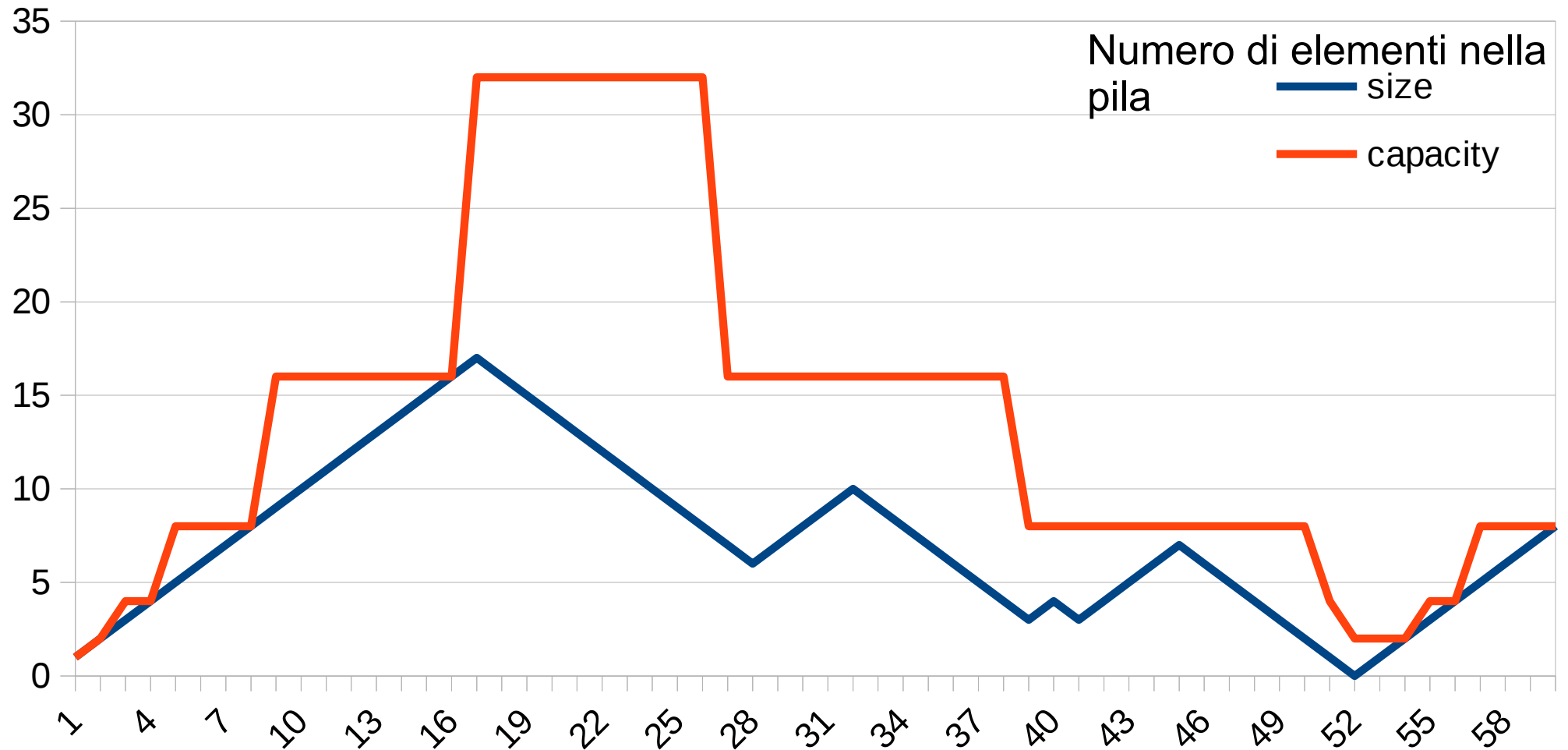
`push()`



`pop()`



## Raddoppiamento/dimezzamento



# Analisi delle operazioni **push ( )** e **pop ( )**

- Nel caso peggiore, entrambe sono  $O(n)$
- Nel caso migliore, entrambe sono  $O(1)$
- Partendo dallo stack vuoto, quanto costano  $n$  **push ( )** consecutive?
  - $1 + 2 + 4 + \dots + n/2^i = O(n)$
- Partendo da uno stack con  $n$  elementi, quanto costano  $n$  **pop ( )** consecutive?
  - $n/2 + n/4 + n/8 + \dots + 2 + 1 = O(n)$

*Partendo dallo stack vuoto, quanto costano  $n$  `push()` consecutive?*

Costo effettivo  $c_i$  di una operazione `push()`:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

$n$	costo
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Costo complessivo di  
 $n$  operazioni `push()`:

$$\begin{aligned} T(n) &= \sum_{i=1}^n c_i \\ &= n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \\ &= n + 2^{\lfloor \log n \rfloor + 1} - 1 \\ &\leq n + 2^{\log n + 1} - 1 \\ &= n + 2n - 1 = O(n) \end{aligned}$$

Costo Ammortizzato  
di una operazione  
`push()`:

$$T(n)/n = \frac{O(n)}{n} = O(1)$$

$2^{\log n + 1} = 2 * 2^{\log n} = 2n$  perché il log è in base 2.



**FINE**

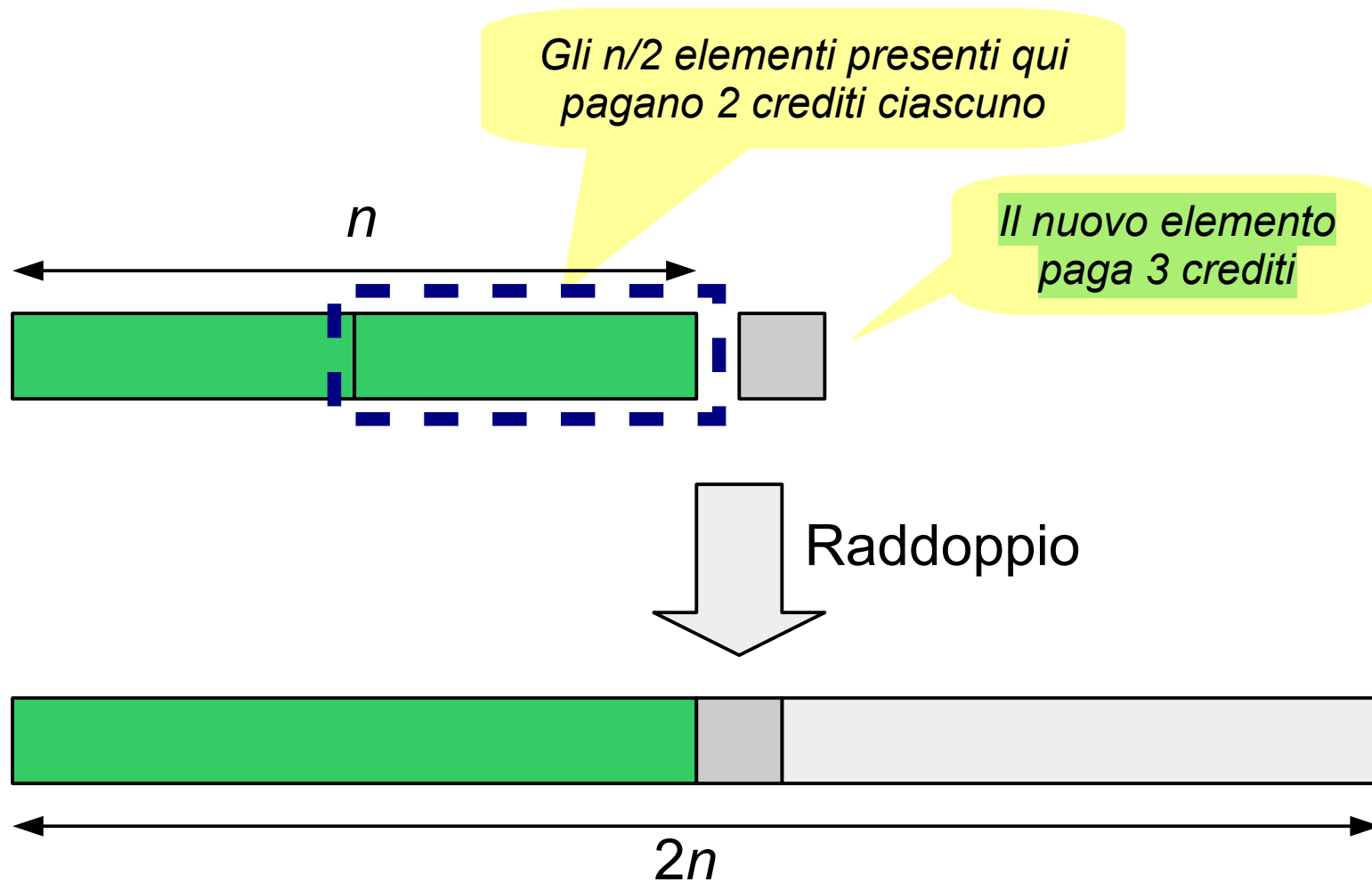
# Analisi ammortizzata: il metodo dei crediti

- Associamo a ciascun elemento della struttura dati un numero di crediti
  - Un credito può essere utilizzato per eseguire  $O(1)$  operazioni elementari
- Quando creo un elemento la prima volta, “pago” un certo numero di crediti
- Userò quei crediti per pagare ulteriori operazioni su quell'elemento, in futuro

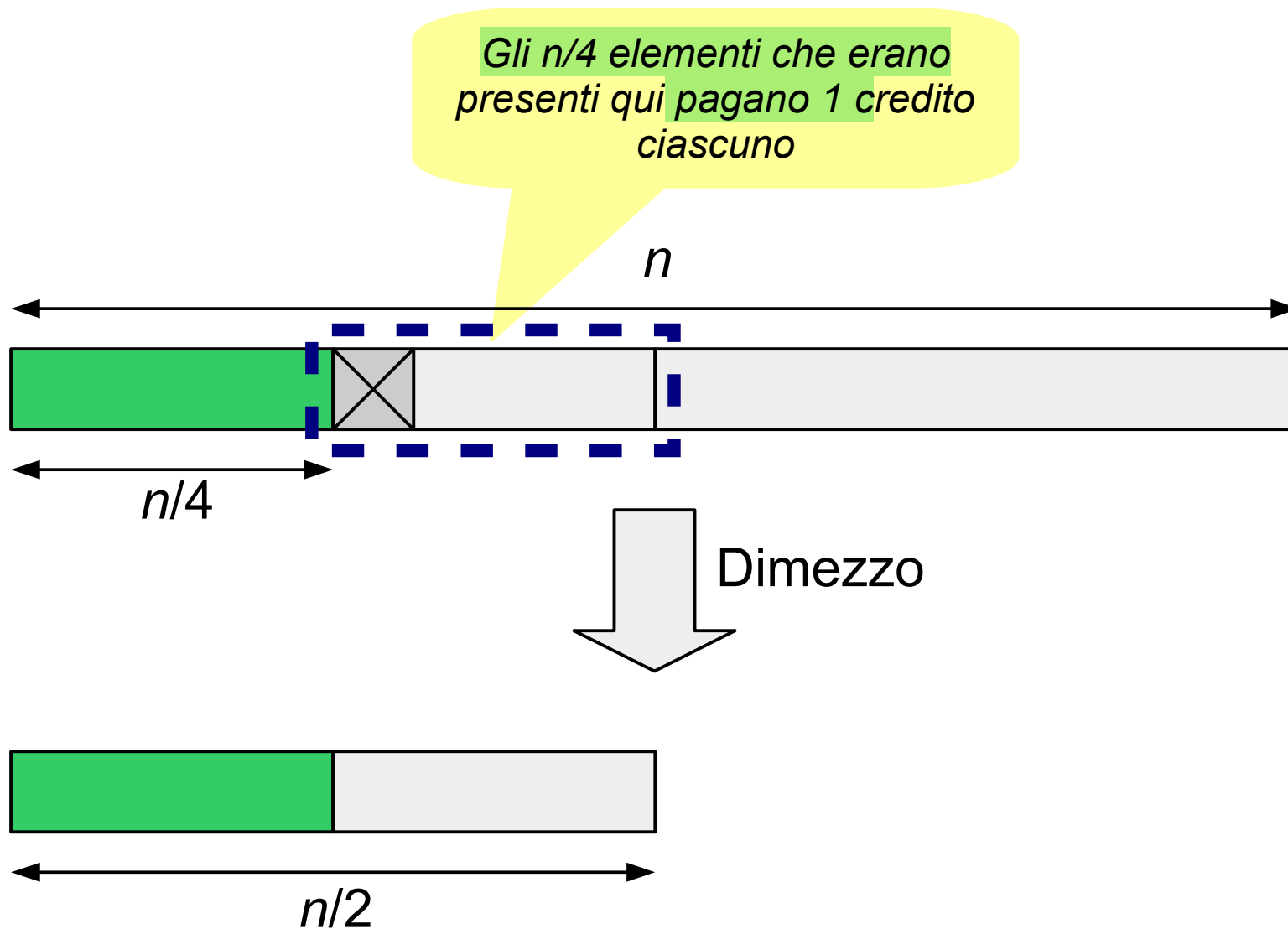
# Analisi ammortizzata delle operazioni `push()` e `pop()`

- L'inserimento di un elemento nello stack deposita 3 crediti sulla cella dell'array
- Quando devo raddoppiare
  - Sottraggo 2 crediti dalle celle nella seconda metà dell'array (prima del raddoppio);
  - Uso questi crediti per “pagare” la copia dei valori dall'array originale a quello “raddoppiato”
- Quando devo dimezzare
  - Sottraggo 1 credito dalle celle nel secondo quarto dell'array (prima del dimezzamento)
  - Uso questi crediti per “pagare” la copia

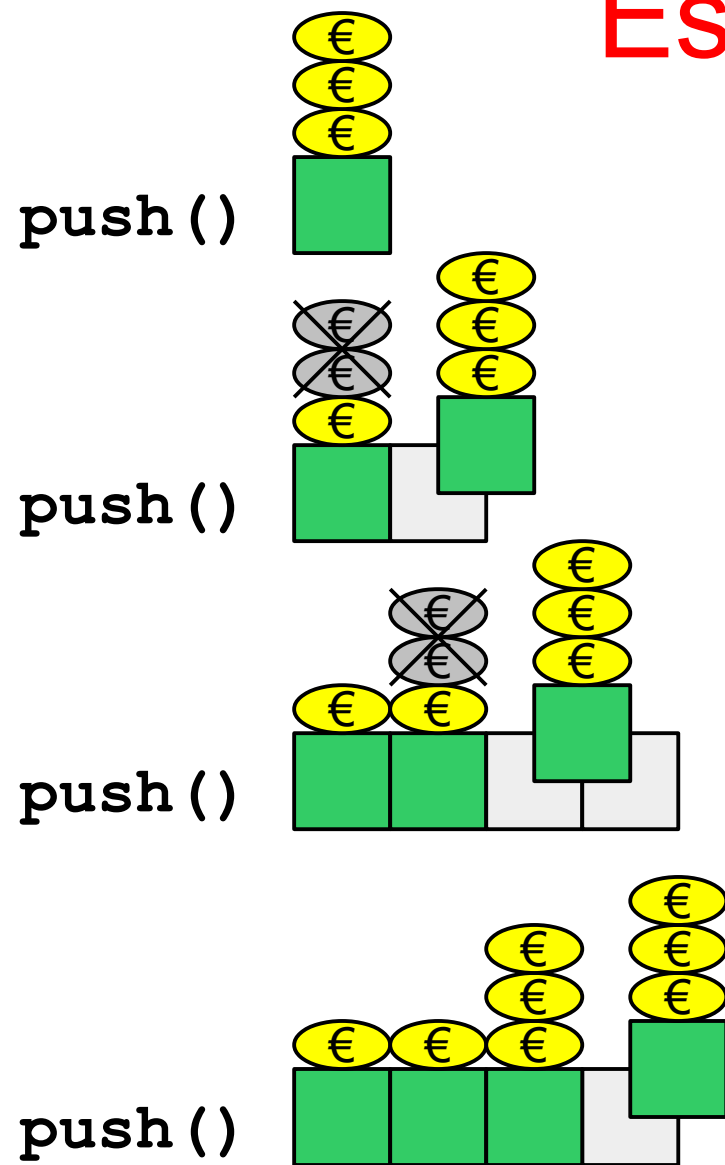
# Analisi ammortizzata



# Analisi ammortizzata

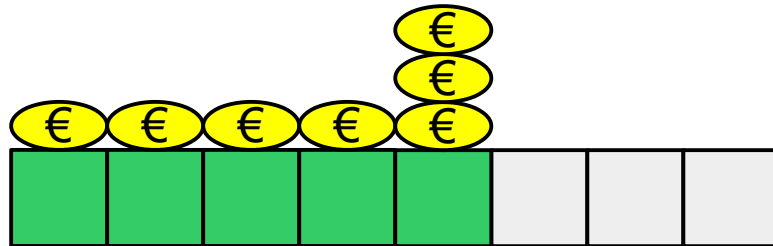
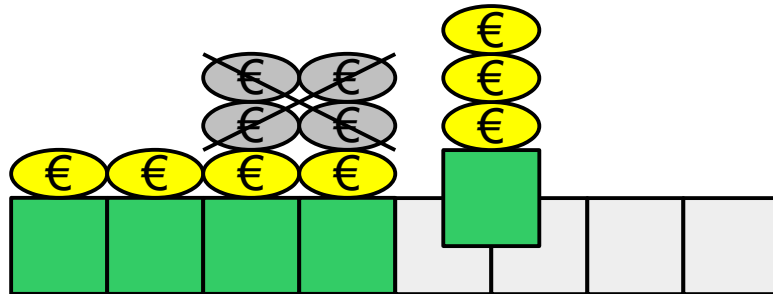


# Esempio

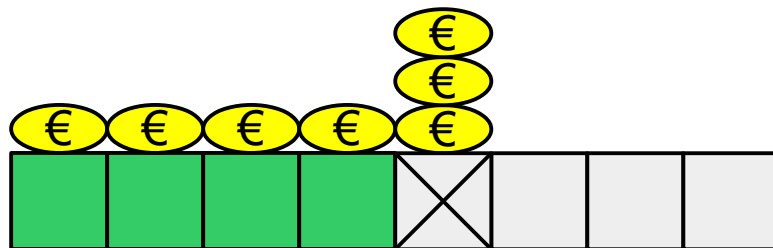


# Esempio (cont.)

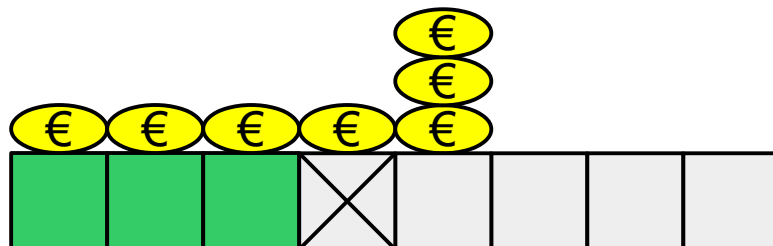
push ()



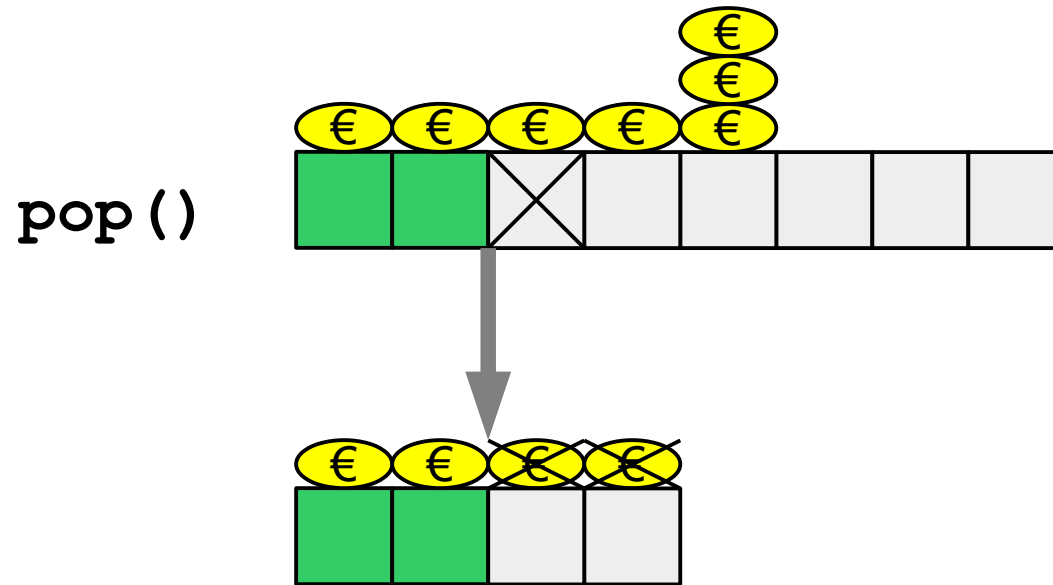
pop ()



pop ()



## Esempio (cont.)





# Analisi ammortizzata: il metodo dei crediti

- Quindi:
  - Nel caso peggiore, le operazioni possono avere costo  $O(n)$  se causano un raddoppio o un dimezzamento dell'array
- Ma:
  - Le operazioni "costose" sono rare e il loro costo può essere compensato da altre meno costose
- Il costo ammortizzato di **push ()** e **pop ()** su uno stack dinamico è  $O(1)$