

INFORMAZIONI SUL CORSO
COSA È UN ALGORITMO
ESEMPI DI ALGORITMI

ALGORITMI E STRUTTURE DATI

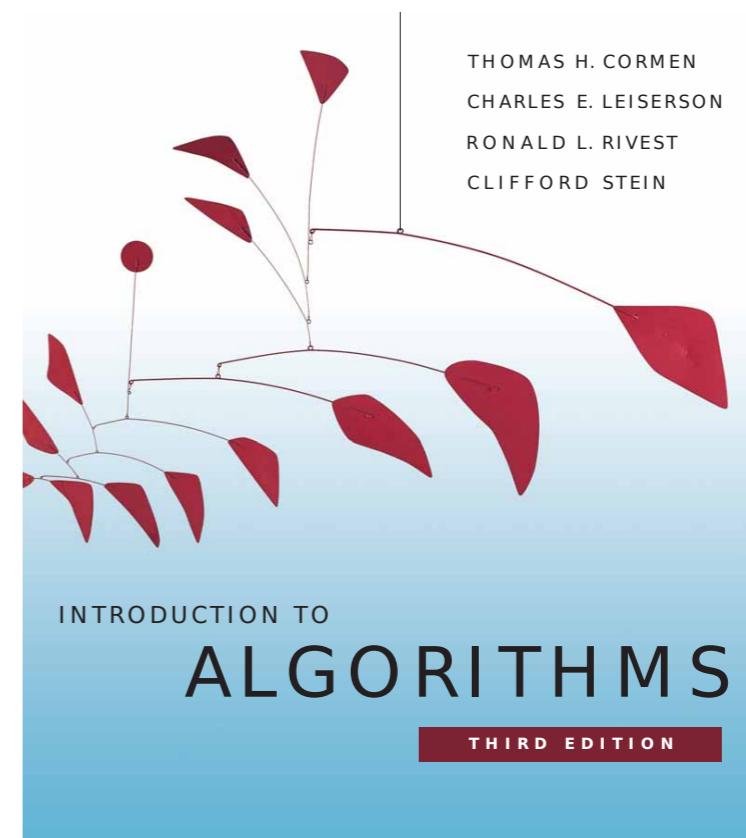
STRUTTURA DEL CORSO

- ▶ Introduzione alla progettazione di algoritmi (complessità, ricorsione, strutture dati)
- ▶ Ordinamento e selezione
- ▶ Strutture dati per l'indicizzazione (alberi binari di ricerca, hash)
- ▶ Strutture relazionali (grafo e algoritmi su grafi)

LIBRI E MATERIALE

- ▶ Cormen, Leiserson, Rivest, Stein
Introduction to Algorithms (Introduzione agli algoritmi)
3rd edition (o 4th edition).
MIT Press (McGraw Hill per l'edizione italiana)

(È un libro grosso, con tanti argomenti oltre a quelli che vedremo).



SOFTWARE E MATERIALE DEL CORSO

- ▶ Il materiale del corso (slide ed esempi di codice) sarà disponibile online sul sito e-learning del corso
- ▶ Come software si utilizzerà Python versione 3.6 o successive
(disponibile su <https://www.python.org>)
 - ▶ Solitamente preinstallato su macOS e Linux
 - ▶ È possibile usare un ambiente Python 3 online senza installazione su repl.it (<https://repl.it>)

ESAME E HOMEWORK

- ▶ L'esame avrà una parte scritta ed una orale. Lo scritto è obbligatorio, l'orale è obbligatorio se (a) il voto dopo lo scritto è 16 o 17 e non vi ritirate o (b) se il voto dopo lo scritto ed il bonus esercizi è > 29 . Tutti possono decidere di fare l'orale per incrementare il voto.
- ▶ **Scritto:** risoluzione di problemi tipo quelli visti durante il corso.
- ▶ **Orale:** domande sulla teoria e risoluzione di problemi, vale ± 5 punti.
- ▶ **Provette:** ci saranno due provette intermedie. Chi le supera entrambe può saltare lo scritto (vale la media delle provette come voto dello scritto).
- ▶ **Homework:** verranno dati settimanalmente alcuni esercizi per casa, sia problemi da risolvere che esercizi implementativi. Chi li consegna in tempo, in funzione del voto preso, avrà dei punti bonus allo scritto (massimo 3 punti, ma il passaggio da 29 a 30 costa due punti).
- ▶ La lode si può prendere solo con l'orale, di norma con un voto sullo scritto di almeno 27.

LET'S START!

Cos'è un algoritmo?

COSA È UN ALGORITMO? (DEFINIZIONE INFORMALE)

Una procedura computazione ben definita
che prende un valore o insieme di valori come **input**
e produce un valore o insieme di valori come **output**

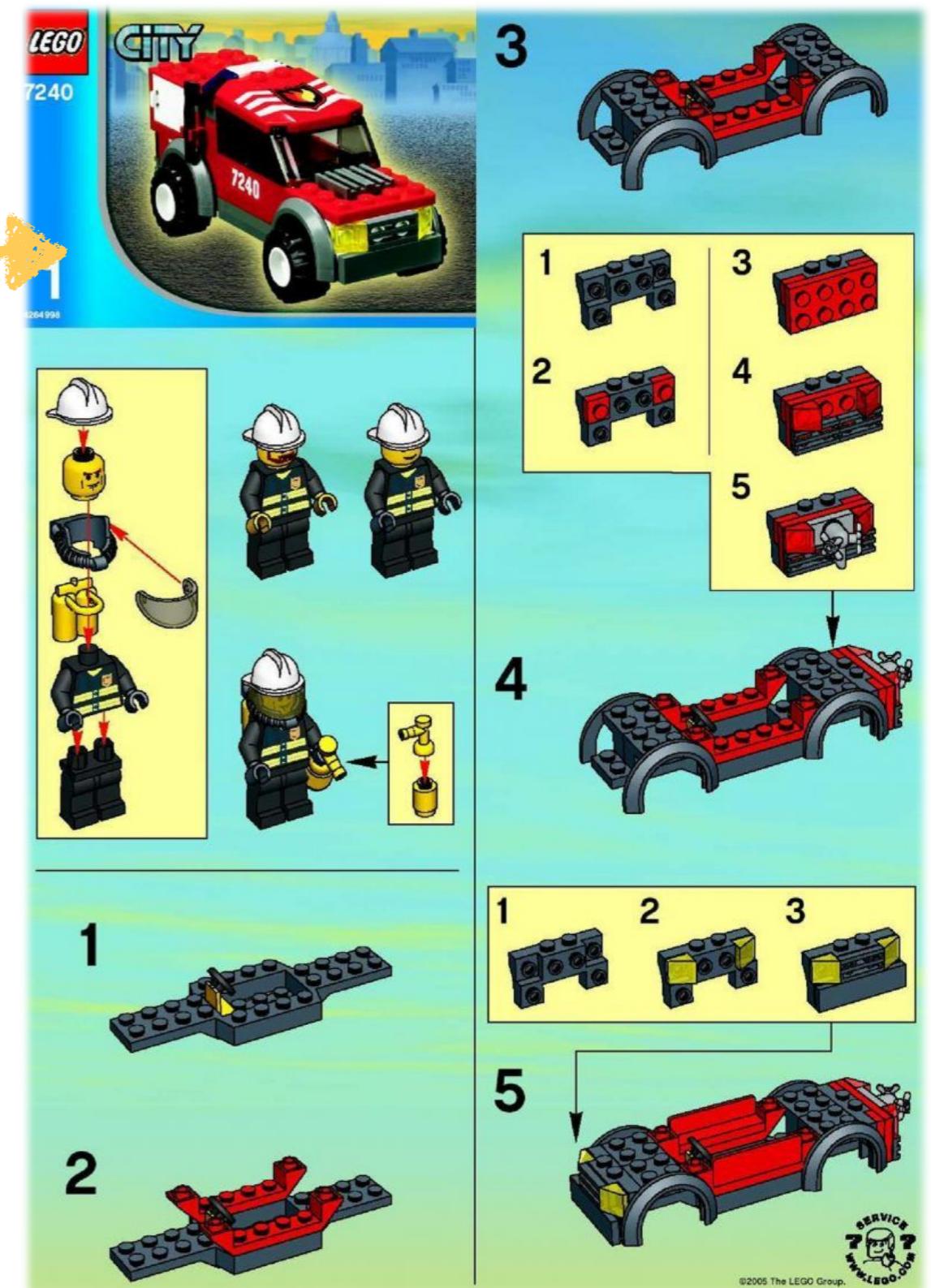
Un algoritmo è quindi un insieme di operazioni
che trasformano un input in un output

ALCUNI ESEMPI COMUNI

Input: un insieme ben definito di pezzi di lego

Output:

Le istruzioni contengono un insieme di passi non ambigui che trasformano l'input nell'output



ALCUNI ESEMPI COMUNI

Input: un insieme ben definito di pezzi di ingredienti

Output: tartufi al tiramisù

Anche in questo caso le istruzioni contengono un insieme di passi non ambigui (più o meno) che trasformano l'input nell'output

Preparazione

COME FARE I TARTUFI AL TIRAMISÙ



Per preparare i tartufini al tiramisù iniziamo ad inserire i Pavesini nel mixer [1] e a tritarli, fino ad ottenere una polvere fine. [2]



A parte lavoriamo il mascarpone con lo zucchero, aiutandoci con le fruste elettriche. [3]
Aggiungiamo la polvere di Pavesini ed amalgamiamo con una spatola a mano. [4]

ORDINARE UN MAZZO DI CARTE



Abbiamo un mazzo di carte che è stato mescolato

Vogliamo disporre le carte in ordine crescente

Siamo in grado di dare un **algoritmo**?

Iniziamo a formalizzare il problema

ESEMPIO: ALGORITMI DI ORDINAMENTO

ORDINARE UN MAZZO DI CARTE



Dati in ingresso (**input**)

Un mazzo di carte non necessariamente ordinate



Dati in uscita (**output**)

Una permutazione del mazzo di carte fornito in input
in cui tutte le carte sono ordinate in ordine crescente

ESEMPIO: ALGORITMI DI ORDINAMENTO

ORDINARE UN MAZZO DI CARTE



Dati in ingresso (**input**)

Un mazzo di carte non necessariamente ordinate

CI SERVE UNA PROCEDURA MECCANICA
CHE DALL'INPUT PRODUCA L'OUTPUT

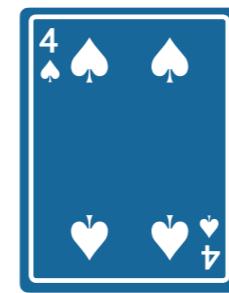
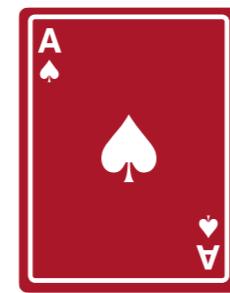
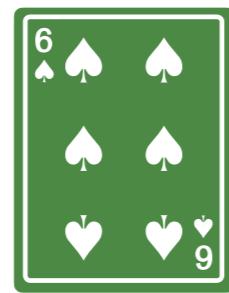
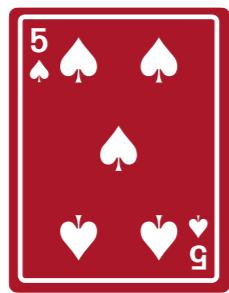
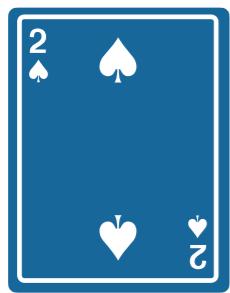


Dati in uscita (**output**)

Una permutazione del mazzo di carte fornito in input
in cui tutte le carte sono ordinate in ordine crescente

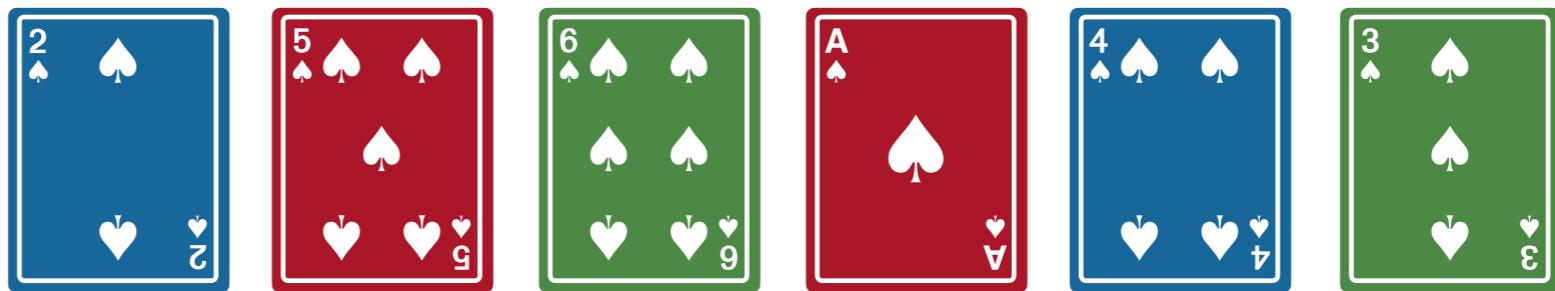
ESEMPIO: ALGORITMI DI ORDINAMENTO

IDEE?



ESEMPIO: ALGORITMI DI ORDINAMENTO

IDEA: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



- 1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
- 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate

→ 2. Finché ci rimangono carte non ordinate

2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X

2.2. Metti X in fondo alla lista di carte ordinate

2.3. Finché la carta che precede X ha valore maggiore del valore di X,
scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA

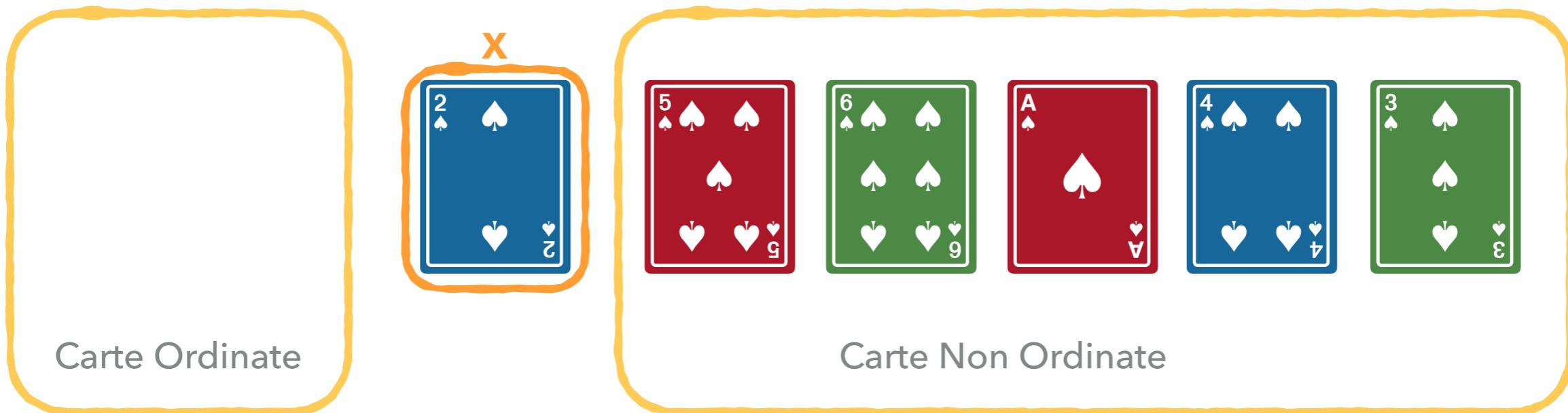


1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate

**CONTINUEREMO AD ESEGUIRE
QUESTE ISTRUZIONI
FINCHÉ LA CONDIZIONE
NON SARÀ SODDISFATTA**

- 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
- 2.2. Metti X in fondo alla lista di carte ordinate
- 2.3. Finché la carta che precede X ha valore maggiore del valore di X,
scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

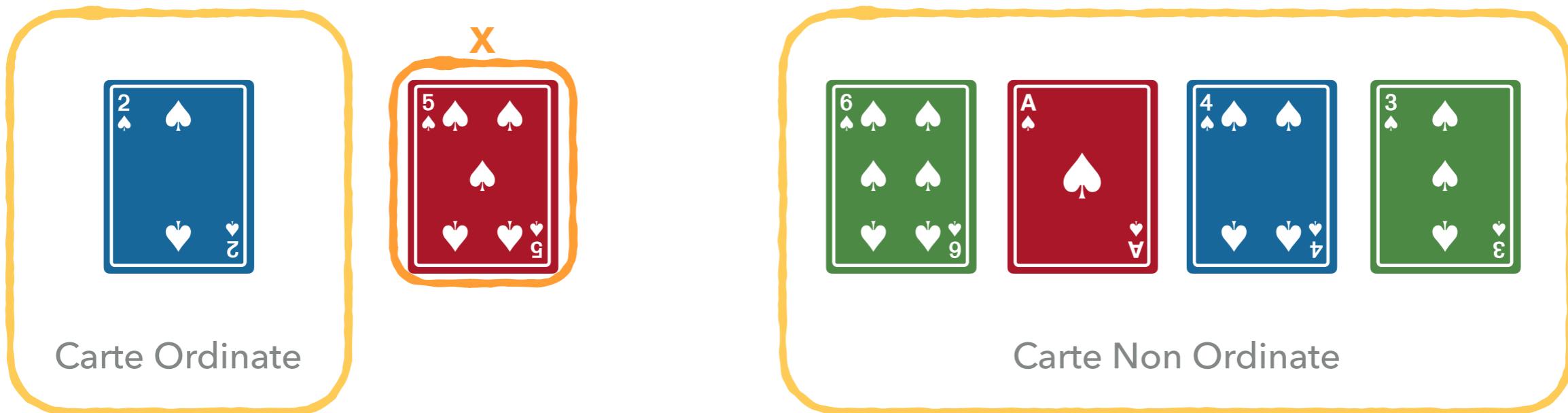
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

**NON DOBBIAMO FARE NIENTE
(NON CI SONO ALTRE CARTE GIÀ IN ORDINE)**

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



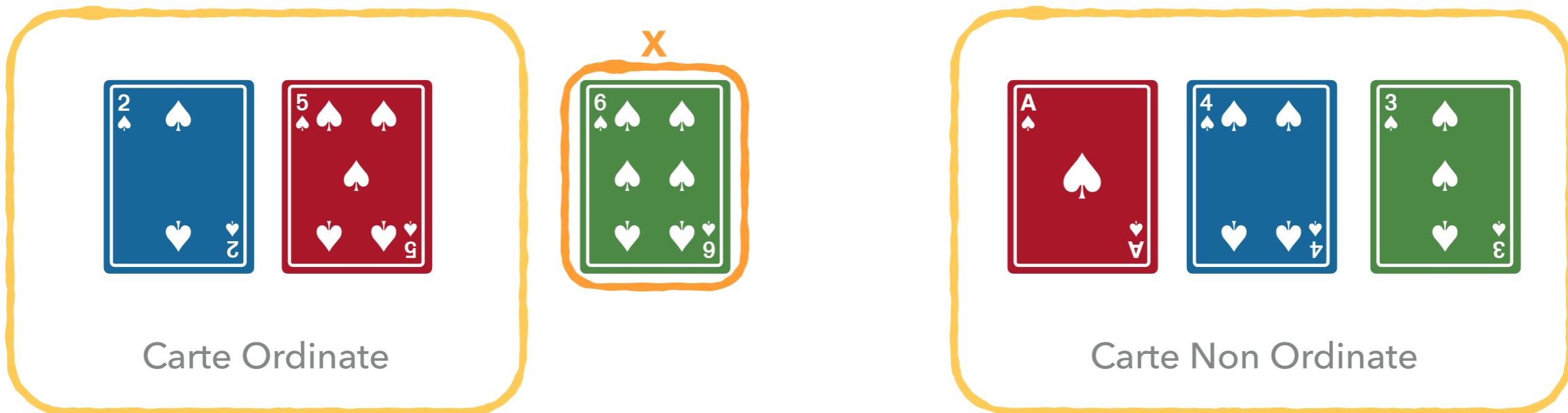
$2 > 5?$ NO



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

IL TEST FALLISCE
E QUINDI PROSEGUIREMO
CON L'ISTRUZIONE NUMERO...

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

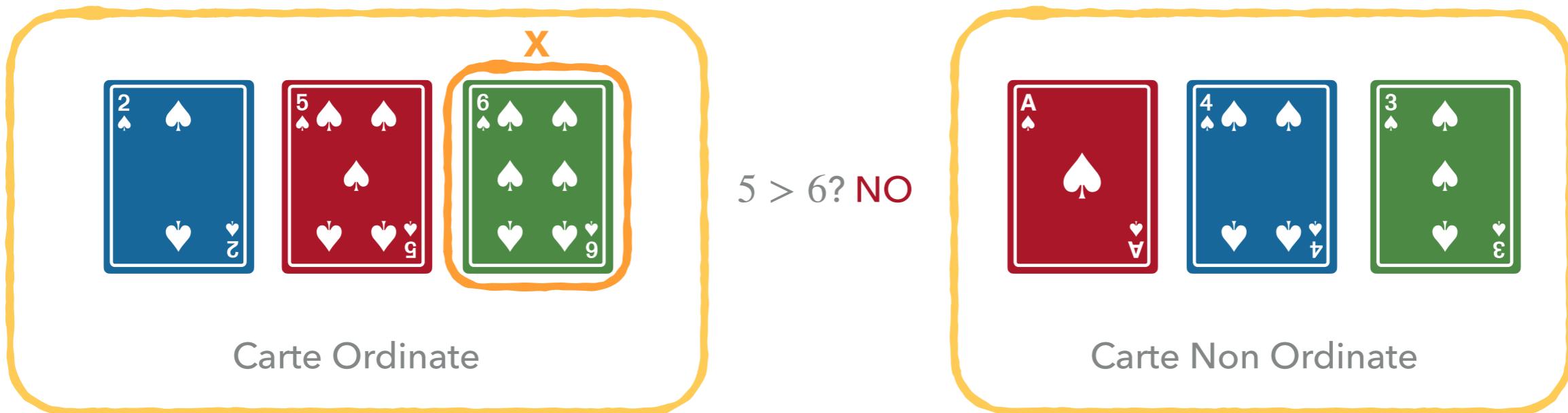
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

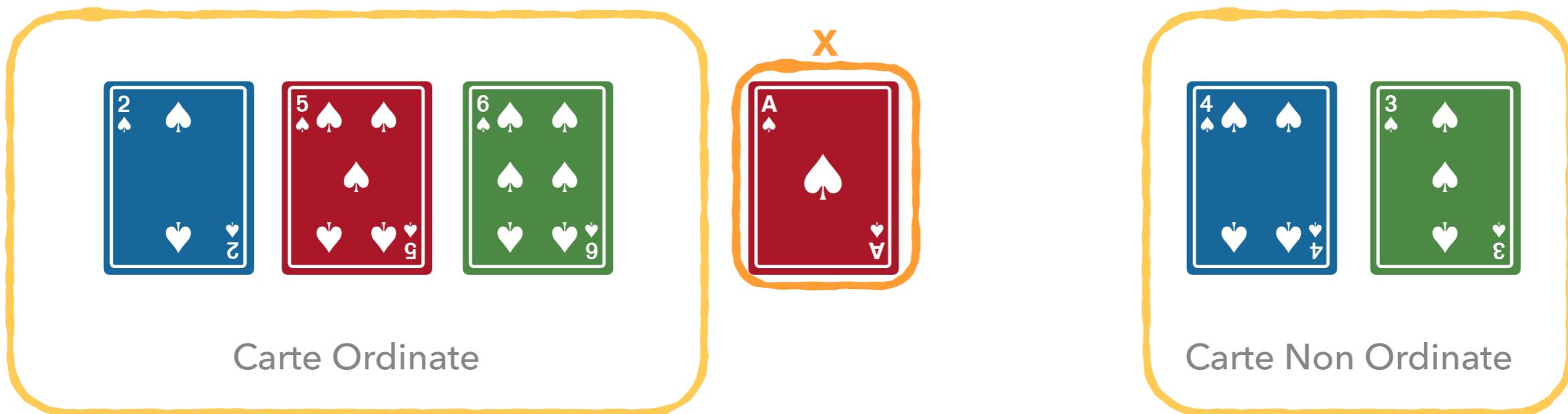
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

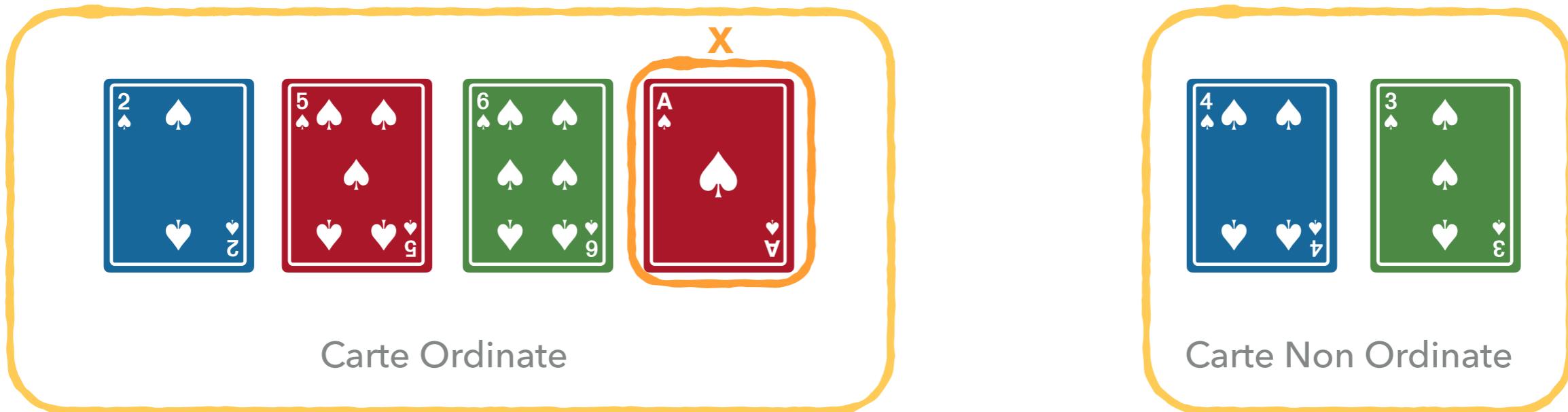
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

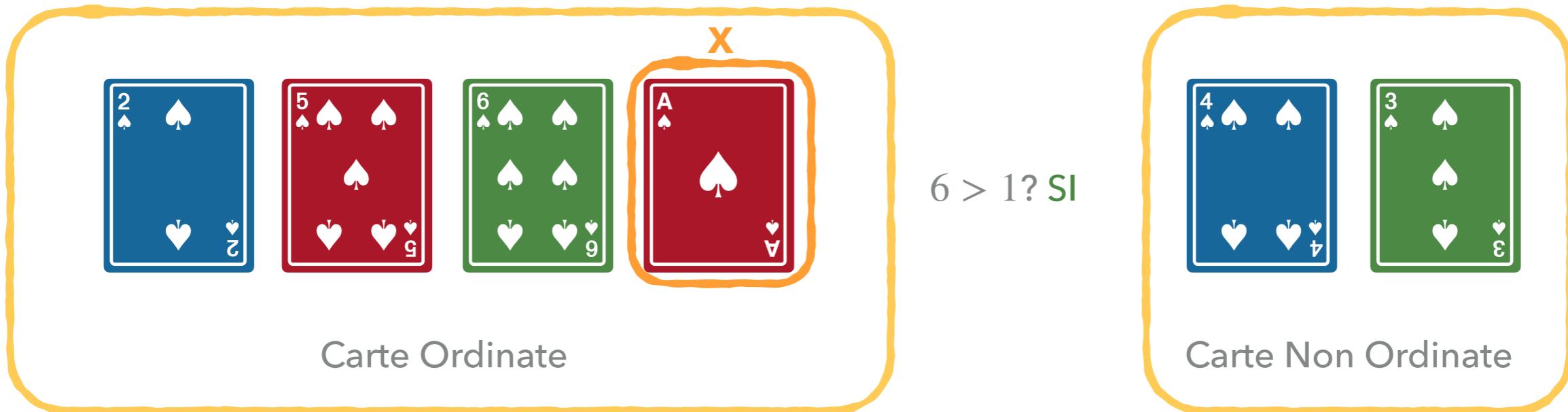
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

QUI INVECE IL TEST HA SUCCESSO
QUINDI DOBBIAMO ESEGUIRE
L'OPERAZIONE DI SCAMBIO

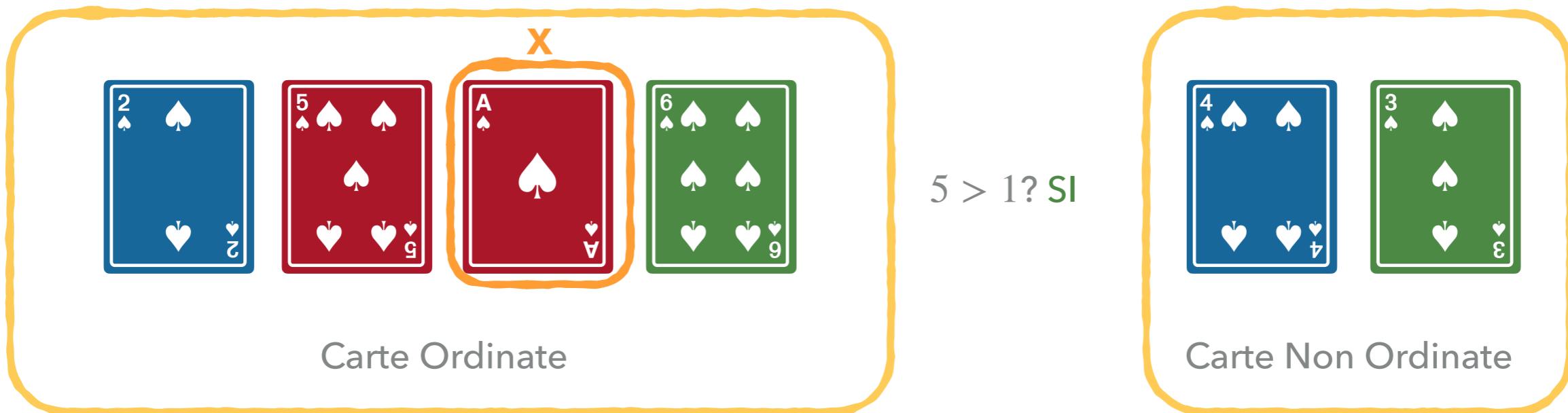
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

**ORA CHE DOBBIAMO FARE?
CONTINUIAMO CON L'ISTRUZIONE 2.3
O TORNIAMO ALLA 2.1?**

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA

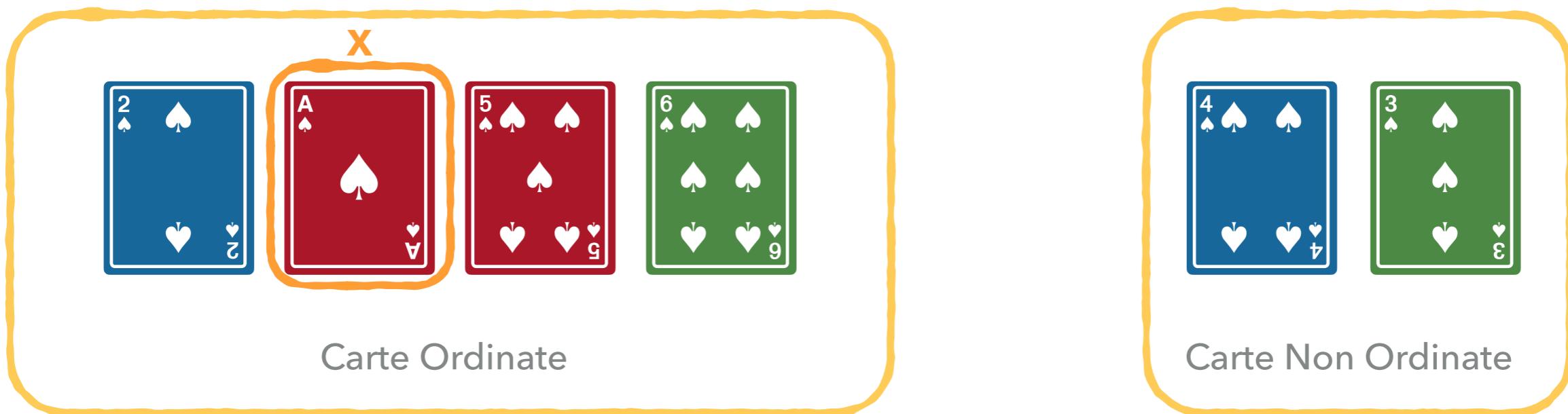


1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ORA CHE DOBBIAMO FARE?
CONTINUIAMO CON L'ISTRUZIONE 2.3
O TORNIAMO ALLA 2.1?

ESEMPIO: ALGORITMI DI ORDINAMENTO

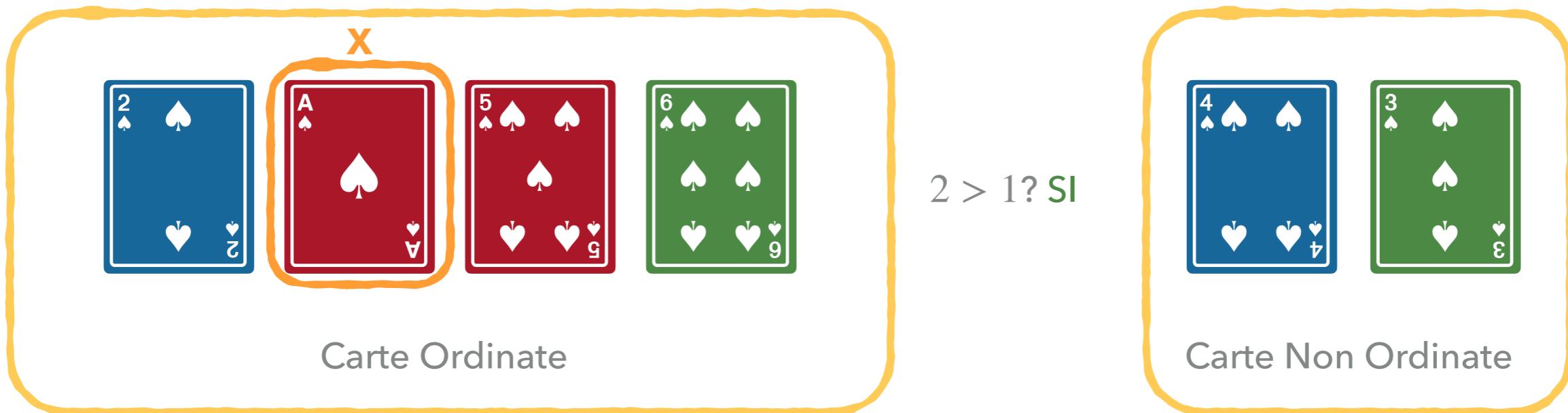
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

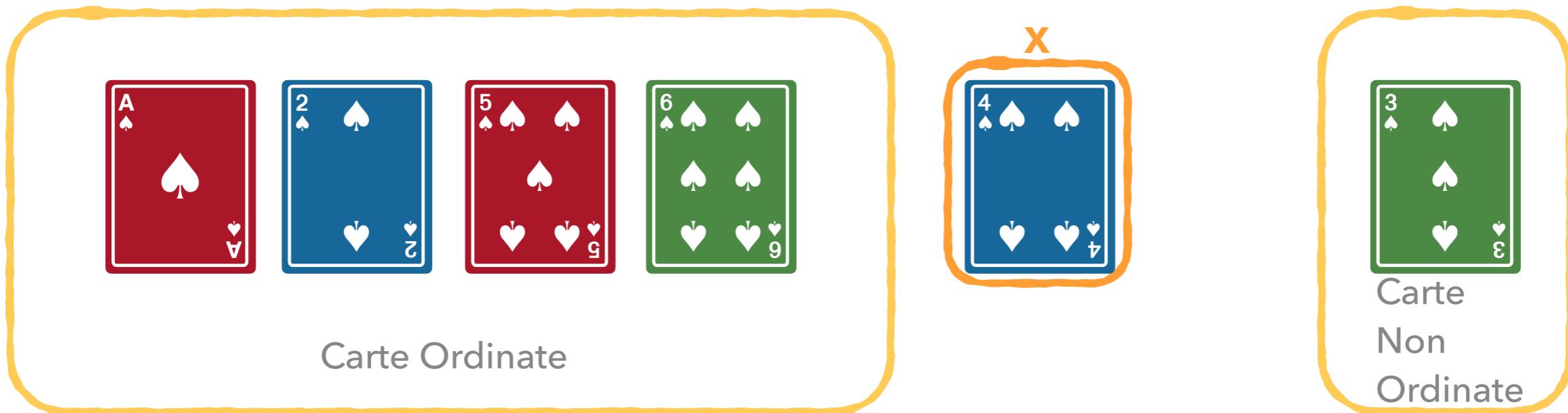
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

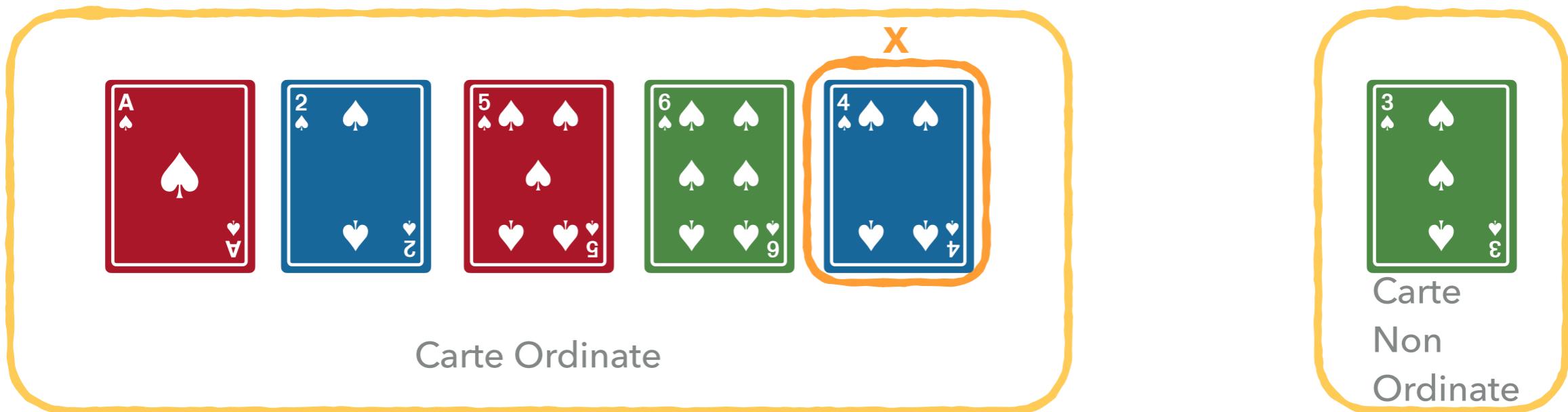
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

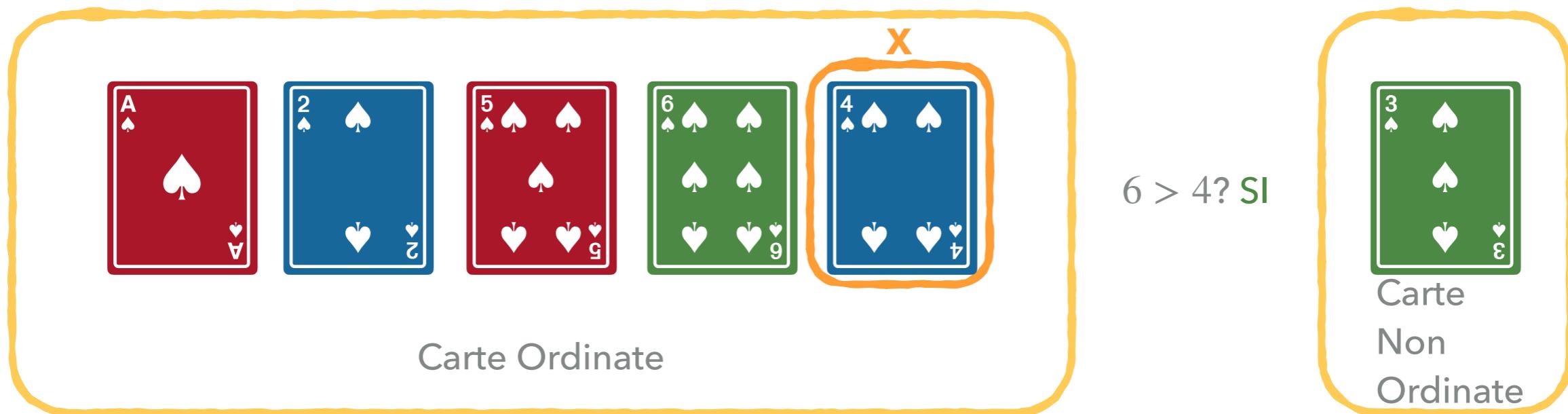
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate →
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

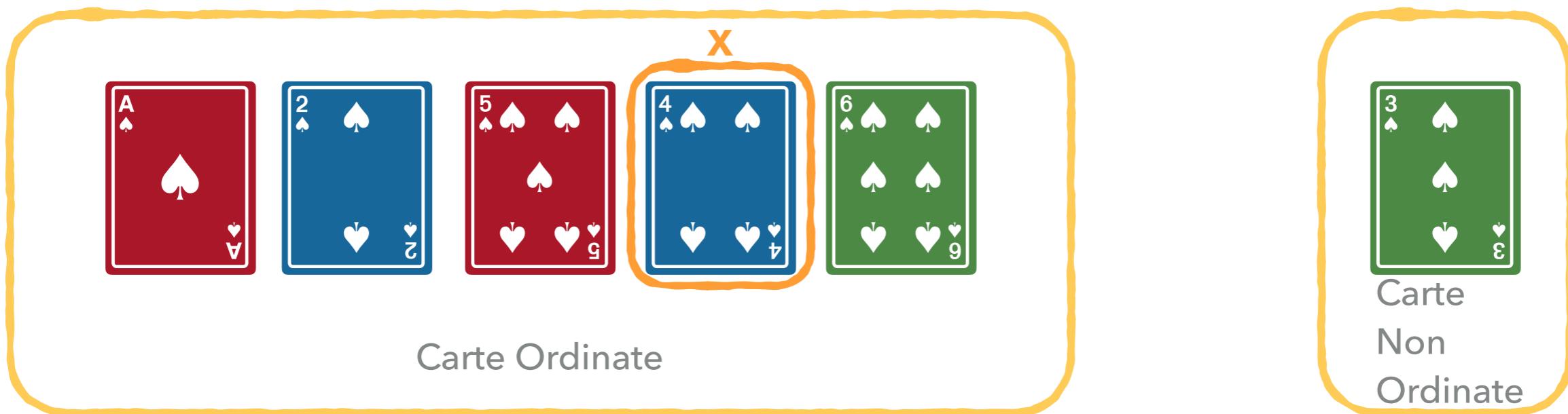
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

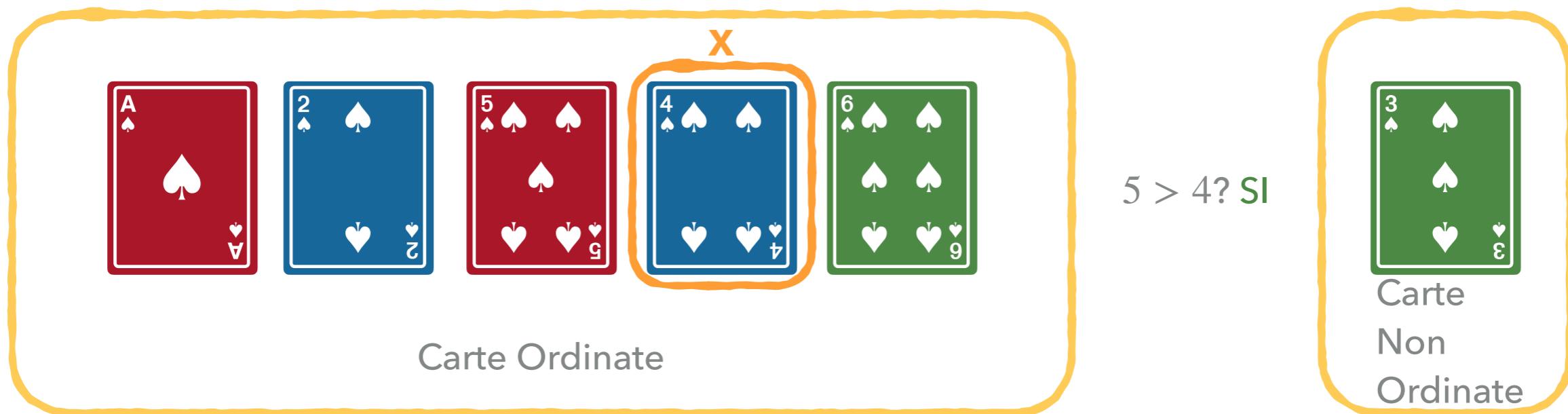
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

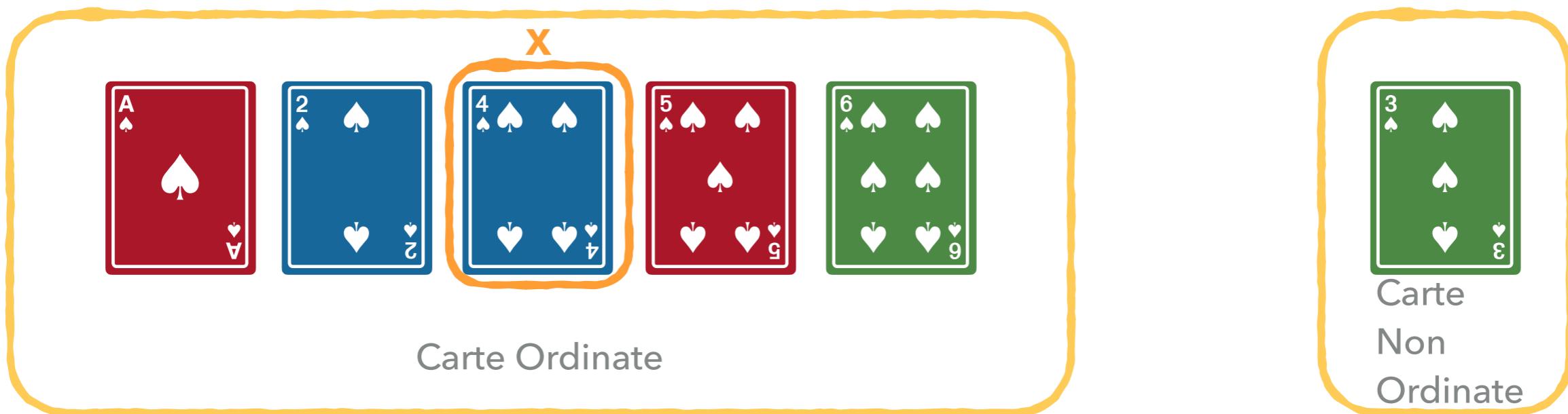
ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

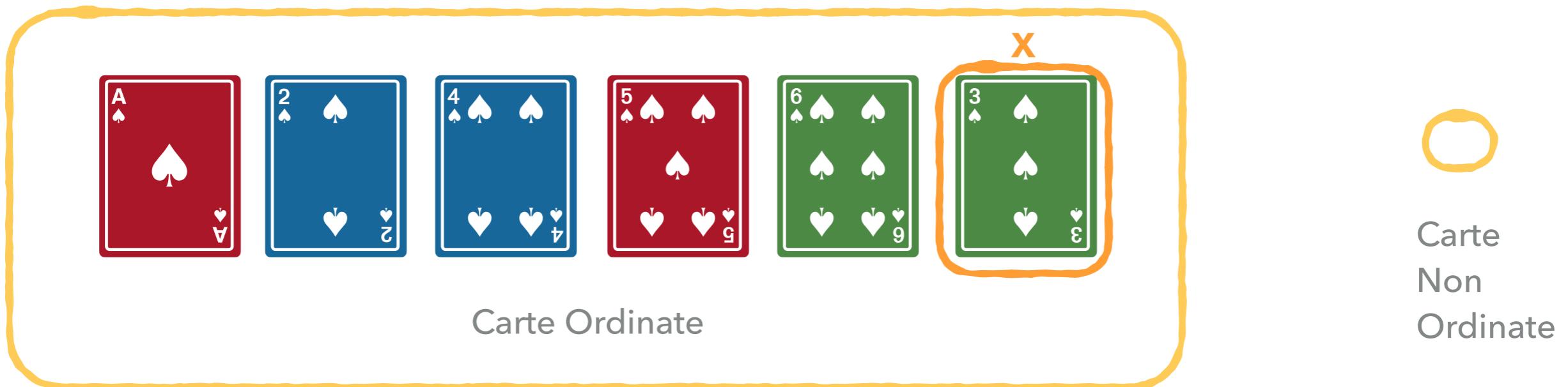
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

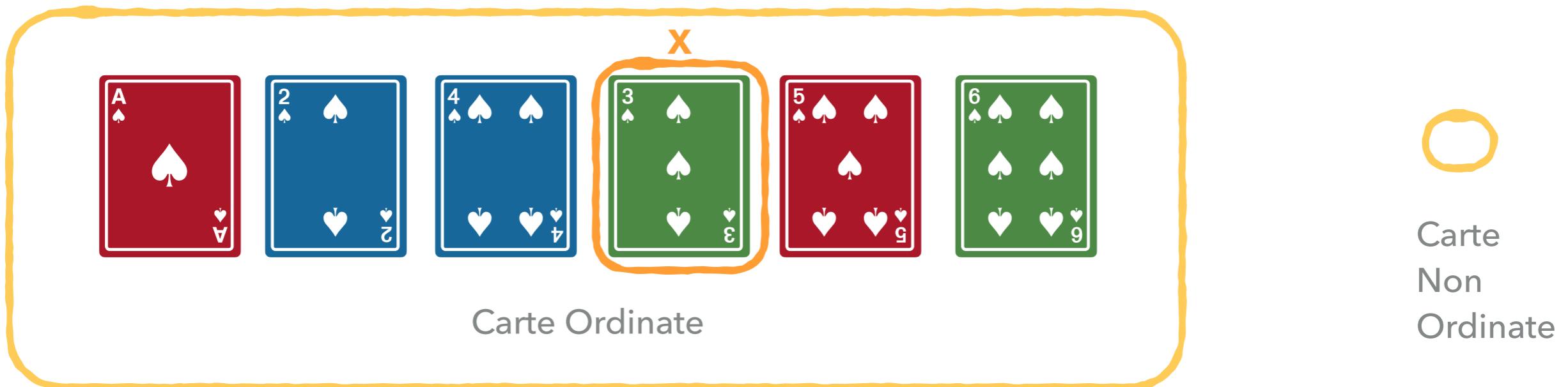
ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

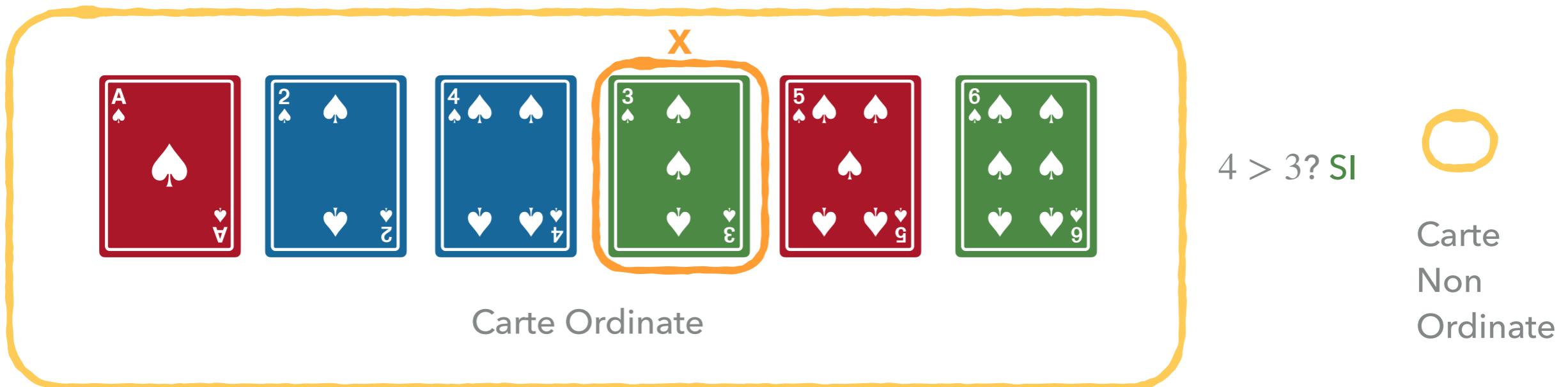
OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
- 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
- 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
- 2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ESEMPIO: ALGORITMI DI ORDINAMENTO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
- 2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



1. Inizialmente non si hanno carte ordinate
- 2. Finché ci rimangono carte non ordinate
 - 2.1. Prendi una carta dalla lista di carte non ordinata e chiamala X
 - 2.2. Metti X in fondo alla lista di carte ordinate
 - 2.3. Finché la carta che precede X ha valore maggiore del valore di X, scambia X con la carta che lo precede

ORA LA CONDIZIONE NON È PIÙ VERA
E ABBIAMO TERMINATO

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA



Siamo riusciti ad ottenere un mazzo di carte ordinato

Rimangono alcune domande:

- ▶ Correttezza: questo algoritmo funziona con ogni input?
- ▶ Efficienza: quanti passi ci mettiamo ad ordinare un mazzo di n carte al crescere di n ?

OPZIONE 2: INSERIMENTO DI UNA CARTA ALLA VOLTA

- ▶ Quante comparazioni dobbiamo fare nel caso peggiore?
- ▶ Inserire una carta nella lista delle carte ordinate richiede non più di n comparazioni
- ▶ Dobbiamo inserire n carte nella lista delle carte ordinate per ordinarle tutte
- ▶ Quindi il numero di comparazioni è limitato superiormente da n^2

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n, $m > n$

Output: il più grande intero più piccolo di m ed n che li divide entrambi.

Idea: provo tutti i possibili divisori, dal più grande al più piccolo....

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n, $m > n$

Output: il più grande intero più piccolo di m ed n che li divide entrambi.

Idea: provo tutti i possibili divisori, dal più grande al più piccolo....

```
def MCD(m,n):
    mcd = 0
    for i=1 to floor(sqrt(n)):
        if i | n:
            if n//i | m:
                return n//i
            else if i | m
                mcd = i
    return mcd
```

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n, $m > n$

Output: il più grande intero più piccolo di m ed n che li divide entrambi.

Idea: provo tutti i possibili divisori, dal più grande al più piccolo....

```
def MCD(m,n):
    mcd = 0
    for i=1 to floor(sqrt(n)):
        if i | n:
            if n//i | m:
                return n//i
        else if i | m
            mcd = i
    return mcd
```

Complessità

l'algoritmo fa nel caso peggiore \sqrt{n} iterazioni del ciclo for.

Ci piace questa complessità?

Proviamo con $n = 2^{10}$?

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n, $m > n$

Output: il più grande intero più piccolo di m ed n che li divide entrambi.

Idea: provo tutti i possibili divisori, dal più grande al più piccolo....

```
def MCD(m,n):
    mcd = 0
    for i=1 to floor(sqrt(n)):
        if i | n:
            if n//i | m:
                return n//i
        else if i | m
            mcd = i
    return mcd
```

Complessità

l'algoritmo fa nel caso peggiore \sqrt{n} iterazioni del ciclo for.

Ci piace questa complessità?

Proviamo con $n = 2^{10}$?
e con 2^{100} ?

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n, $m > n$

Output: il più grande intero p più piccolo di m ed n che li divide entrambi.

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n , $m > n$

Output: il più grande intero p più piccolo di m ed n che li divide entrambi.

Idea: ci penso un poco...

partiamo da $m = qn + r$:

se $r = 0$, allora $n \mid m$, e ho finito

se $r \neq 0$, siccome $p \mid m$ e $p \mid n$, allora $p \mid r$

quindi $\text{MCD}(m, n) = \text{MCD}(n, r)$

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n, $m > n$

Output: il più grande intero p più piccolo di m ed n che li divide entrambi.

Idea: ci penso un poco...

partiamo da $m = qn + r$:

se $r = 0$, allora $n \mid m$, e ho finito

se $r \neq 0$, siccome $p \mid m$ e $p \mid n$, allora $p \mid r$

quindi $\text{MCD}(m, n) = \text{MCD}(n, r)$

```
def Euclide(m,n):
```

```
    while n!=0:
```

```
        m, n = n, m mod n
```

```
    return m
```

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n , $m > n$

Output: il più grande intero p più piccolo di m ed n che li divide entrambi.

Idea: ci penso un poco...

partiamo da $m = qn + r$:

se $r = 0$, allora $n \mid m$, e ho finito

se $r \neq 0$, siccome $p \mid m$ e $p \mid n$, allora $p \mid r$

quindi $\text{MCD}(m, n) = \text{MCD}(n, r)$

def Euclide(m, n):

while $n \neq 0$:

$m, n = n, m \bmod n$

return m

Complessità:

Partiamo da $m = qn + r$

$m > n$, ergo $q \geq 1$ e $n > r$

Segue $m = qn + r \geq qr + r \geq r + r = 2r$

Quindi $r \leq m/2$.

Ergo in due passi di Euclide, m dimezza.

L'algoritmo termina dopo non più di $2 \log_2 m$ passi.

EFFICIENCY MATTERS: MASSIMO COMUNE DIVISORE

Input: due numeri interi m ed n , $m > n$

Output: il più grande intero p più piccolo di m ed n che li divide entrambi.

Idea: ci penso un poco...

partiamo da $m = qn + r$:

se $r = 0$, allora $n \mid m$, e ho finito

se $r \neq 0$, siccome $p \mid m$ e $p \mid n$, allora $p \mid r$

quindi $\text{MCD}(m, n) = \text{MCD}(n, r)$

def **Euclide**(m, n):

 while $n \neq 0$:

$m, n = n, m \bmod n$

 return m

Complessità:

Partiamo da $m = qn + r$

$m > n$, ergo $q \geq 1$ e $n > r$

Segue $m = qn + r \geq qr + r \geq r + r = 2r$

Quindi $r \leq m/2$.

Ergo in due passi di Euclide, m dimezza.

L'algoritmo termina dopo non più di $2 \log_2 m$ passi.

Compariamo la complessità quando $m = 2^{100}$ e $n = 2^{100-1}$

MODELLO DI COMPUTAZIONE
COMPLESSITÀ DEGLI ALGORITMI

ALGORITMI E STRUTTURE DATI

Algoritmi e Modello di Computazione

Cos'è un algoritmo?

Definition (Algoritmo)

È una sequenza di passi “*ben definiti*” per trasformare in un tempo finito un insieme di dati in input in un insieme di dati in output.

“*ben definiti*” dipende dal modello di calcolo

Cos'è un algoritmo?

Definition (Algoritmo)

È una sequenza di passi “*ben definiti*” per trasformare in un tempo finito un insieme di dati in input in un insieme di dati in output.

“*ben definiti*” dipende dal modello di calcolo

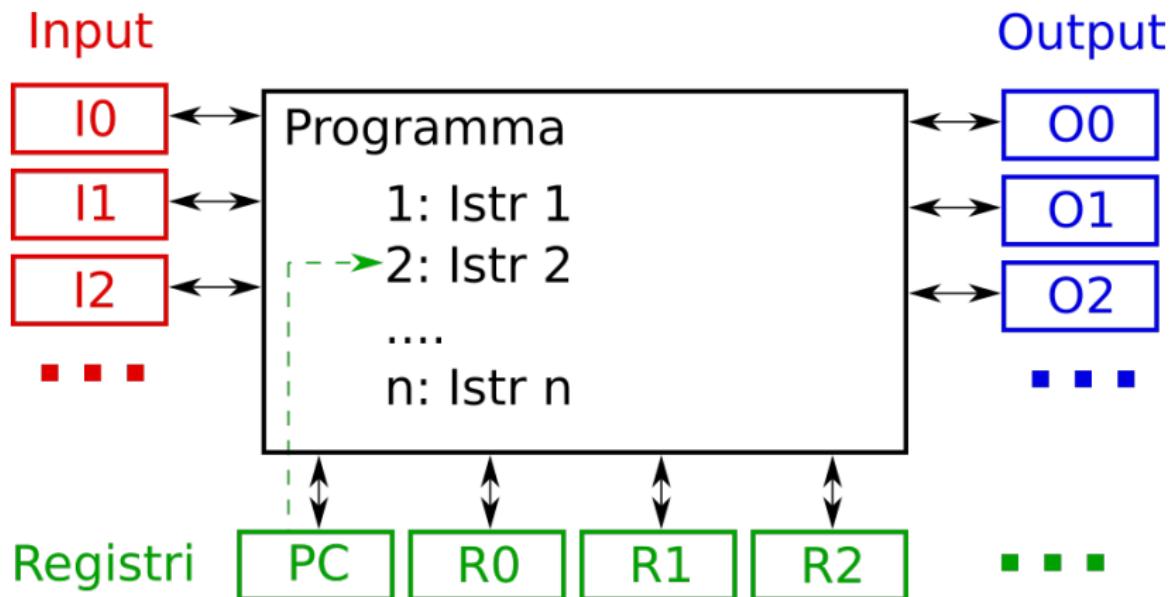
Definition (Modello di Calcolo)

È uno strumento formale per eseguire delle computazioni.

Random-Access Machine (RAM)

- ▶ memoria infinita (**registri**)
- ▶ ogni registro può contenere un qualsiasi numero naturale
- ▶ **input** (sola lettura) e **output** (sola scrittura)
- ▶ il programma è una sequenza di istruzioni
- ▶ il **Program Counter** è un registro che indica la prossima istruzione da eseguire
- ▶ ad ogni istruzione eseguita il **PC** viene automaticamente incrementato (se non cambiato altrimenti)

Random-Access Machine (RAM)



Istruzioni della RAM

Istruzioni di calcolo

- ▶ **CLR(r)** assegna al registro Rr il valore 0;
- ▶ **INC(r)** assegna al registro Rr il valore contenuto in Rr + 1;
- ▶ ***r** ottieni il valore contenuto nel registro r.

Es., Se $*5=8$, allora $**5$ è il contenuto di R8;

- ▶ **JE(*r,s,j)** se $*r=s$ assegna j a PC
- ▶ **HLT** termina l'esecuzione del programma

Cosa possiamo fare con questo numero limitato di istruzioni?

Assegnamento

Il seguente programma assegna il valore v a Rr .

```
1: CLR( r )
2: INC( r )
...
v+1: INC( r )
```

Possiamo estendere l'insieme delle istruzioni della RAM aggiungendo l'**assegnamento** (\leftarrow).

Es.,

```
1: Rr ← v
```

Relazione d'ordine

Verifica se $*s < *r$

- | | |
|----------------|-----------------|
| 1: JE(*r,*s,9) | 7: JE(*1,*r,11) |
| 2: R0 ← *r | 8: JE(0,0,4) |
| 3: R1 ← *s | 9: R0 ← 0 |
| 4: INC(0) | 10: JE(0,0,12) |
| 5: JE(*0,*s,9) | 11: R0 ← 1 |
| 6: INC(1) | 12: ... |

Il risultato è in R0

Possiamo aggiungere le **relazioni l'ordine** (\leq , $<$, $=$, $>$, e \geq).

Espresioni Booleane

Se intepretiamo 0 come FALSO

Negazione di *s

- 1: JE(*s, 0, 4)
- 2: R0 ← 0
- 3: JE(0, 0, 5)
- 4: R0 ← 1
- 5: ...

Disgiunzione logica Rr ∨ Rs

- 1: R0 ← 1
- 2: JE(*r, 1, 5)
- 3: JE(*s, 1, 5)
- 4: R0 ← 0
- 5: ...

Possiamo aggiungere la **logica Booleana** (\vee , \wedge , e \neg).

Es.,

$$1: Rr \leftarrow \neg(*r \vee *s)$$

Costrutti Condizionali e Cicli

If *s then A else B

```
1:    JE(*s, 0, c)
      A
      JE(0, 0, d)
c:    B
d:    ...
```

While *s do A

```
1:    JE(*s, 0, c+1)
      A
c:    JE(0, 0, 1)
c+1:   ...
```

Possiamo aggiungere i costrutti **if-then-else** e **while-do**.

Es.,

```
1:  if  ¬(*r ∨ *s)
2:    Rs ← c
3:  endif
```

```
1:  while  ¬(*r ∨ *s)
2:    Rs ← c
3:  endwhile
```

Somma e Sottrazione

Somma Rs a Rr

- 1: $R0 \leftarrow 0$
- 2: **while** $\neg(*0 == *s)$
- 3: INC(0)
- 4: INC(r)
- 5: **endwhile**

Il risultato è in Rr

Sottrae Rs da Rr

- 1: $R0 \leftarrow 0$
- 2: **while** $(*r > *s)$
- 3: INC(0)
- 4: INC(s)
- 5: **endif**

Il risultato è in R0

Possiamo aggiungere **somma** (+) e **sottrazione** (-).

Es.,

- 1: $Rr \leftarrow *r + *s$
- 2: $Rs \leftarrow *r - 5$

Moltiplicazione e Divisione

Moltiplica Rr per Rs

```
1: R0 ← 0  
2: R1 ← 0  
2: while (*s>*1)  
3:     R0 ← *0 + *r  
4:     R1 ← *1 + 1  
5: endwhile
```

Dividi Rr per Rs

```
1: R0 ← 0  
2: R1 ← 0  
2: while (*r≥*1+*s)  
3:     R1 ← *1 + *s  
4:     R0 ← *0 + 1  
5: endwhile  
6: R1 ← *r - *1
```

Possiamo aggiungere **moltiplicazione** (*) e **divisione** (/).

E.s.,

```
1: Rr ← *r * *s  
2: Rs ← *r / 5
```

La nostra RAM

In sostanza possiamo semplificare la RAM e assumere di avere:

- ▶ variabili (no tipi)
- ▶ array
- ▶ costanti intere e floating point
- ▶ funzioni algebriche: $+$, $-$, $/$, $*$, $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$
- ▶ puntatori
- ▶ istruzioni condizionali e cicli (while e for)
- ▶ procedure e ricorsione (provate a capire come)
- ▶ semplici funzioni “ragionevoli” come $|\cdot|$

... e rimuovere gli indirizzi nel codice

Un Semplice Algoritmo

Input: Un array A di numeri $\langle a_1, \dots, a_n \rangle$.

Output: Il massimo tra a_1, \dots, a_n .

```
def find_max(A):
    max_value ← A[1]
    for i ← 2..|A|:
        if A[i] > max_value:
            max_value ← A[i]
        endif
    endfor

    return max_value
enddef
```

La RAM NON è una Macchina Reale!!!

RAM rappresenta le macchine fisiche, ma non ha:

- ▶ la finitezza della memoria
- ▶ la finitezza della rappresentazione dei numeri
- ▶ le gerarchie della memoria

Complessità degli Algoritmi

Complessità degli Algoritmi

Complessità in termini di tempo: è la somma dei costi delle istruzioni da eseguire

Complessità in termini di spazio: è la somma dello spazio occupato

Qual'è il costo delle operazioni della RAM?

Complessità degli Algoritmi

Complessità in termini di tempo: è la somma dei costi delle istruzioni da eseguire (**dipende dal modello di calcolo**)

Complessità in termini di spazio: è la somma dello spazio occupato

Qual'è il costo delle operazioni della RAM?

Criterio di Costo Uniforme

Il costo in termini di tempo di:

- ▶ una operazione $+, -, *, /$ è 1
- ▶ un'operazione di riferimento in memoria è 1
- ▶ un'istruzione di controllo è 1
- ▶ un assegnamento è 1

Il costo in termini di spazio di un registro è 1

Criterio di Costo Uniforme: un Esempio

```
def test(n):
    Z ← 2                                // costo 1
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo 2
    endfor

    return Z
enddef
```

Il costo totale è $1 + (n + 1) * 1 + n * 2$

Criterio di Costo Uniforme: un Esempio

```
def test(n):
    Z ← 2                                // costo 1
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo 2
    endfor

    return Z
enddef
```

Il costo totale è $1 + (n + 1) * 1 + n * 2$ e alla fine $test(n) = 2^{2^n} \dots$

in tempo lineare l'output occupa spazio $2^n!!!$

Criterio di Costo Logaritmico

Il costo in termini di tempo di:

- ▶ una operazione $a \cdot b$ con $\cdot \in \{+, -, *, /\}$ è $\max(\log a, \log b)$
- ▶ un'operazione di riferimento in memoria è 1
- ▶ un'istruzione di controllo è 1
- ▶ un assegnamento è 1

Il costo in termini di spazio di un registro è log del valore memorizzato

Criterio di Costo Logaritmico: l'Esempio Precedente

```
def test(n):
    Z ← 2                                // costo log 2
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo log Z
    endfor

    return Z
enddef
```

Il costo totale è

$$\begin{aligned}\log 2 + n + 1 + \sum_{i=0}^{n-1} \log 2^{2^i} &= \log 2 + n + 1 + \sum_{i=0}^{n-1} 2^i \\ &= \log 2 + n + 1 + (2^n - 1)\end{aligned}$$

Costo Uniforme vs Costo Logaritmico

Il costo logaritmico modella algoritmi che lavorano con grandi numeri.

Se lo spazio occupato dai valori è limitato (come nei computer reali), il costo uniforme va benissimo.

Come Confrontare l'Efficienza?

Tempo di esecuzione?

Come Confrontare l'Efficienza?

Tempo di esecuzione? (per quale input?)

Gli algoritmi non sono programmi

Assumere costo uniforme/logaritmico non sembra essere realistico perché il tempo di esecuzione dipende da:

- ▶ l'insieme delle istruzioni della CPU
- ▶ Clock di CPU/Memoria/Bus
- ▶ il linguaggio e il compilatore usati
- ▶ come il sistema operativo gestisce la memoria
- ▶ ...

Come Confrontare l'Efficienza?

~~Cosa ne dite del tempo di esecuzione?~~

Qualche altra idea?

Come Confrontare l'Efficienza?

~~Cosa ne dite del tempo di esecuzione?~~

Qualche altra idea?

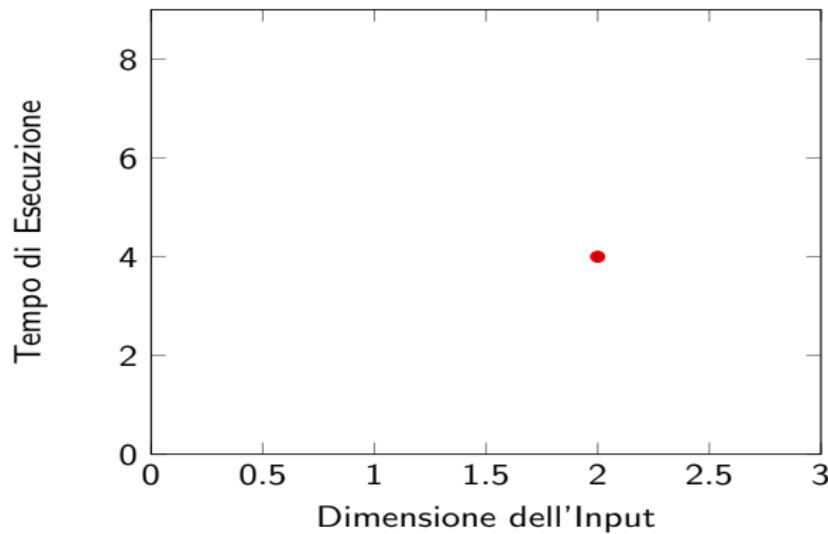
Scalabilità?

Definition (Scalabilità)

Efficienza di un sistema nel gestire crescite nella dimensione dell'input.

Crescita della Complessità

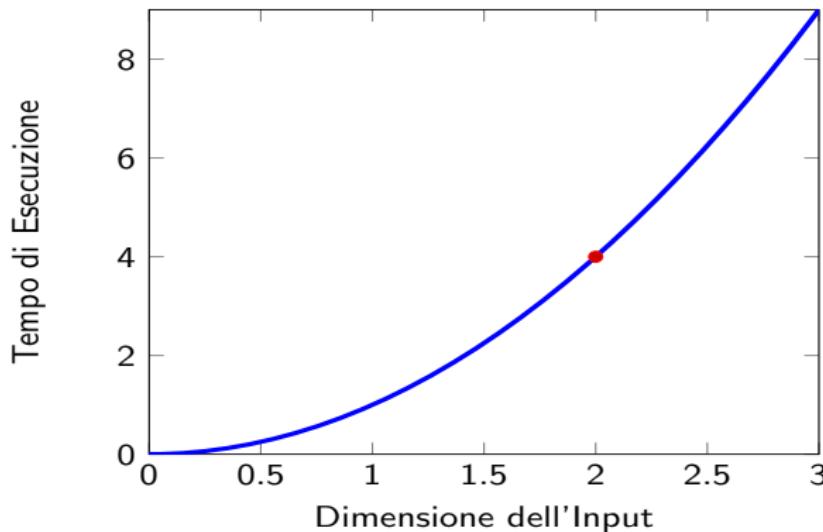
Non misuriamo il tempo di esecuzione **per un dato input**



Crescita della Complessità

Non misuriamo il tempo di esecuzione **per un dato input**

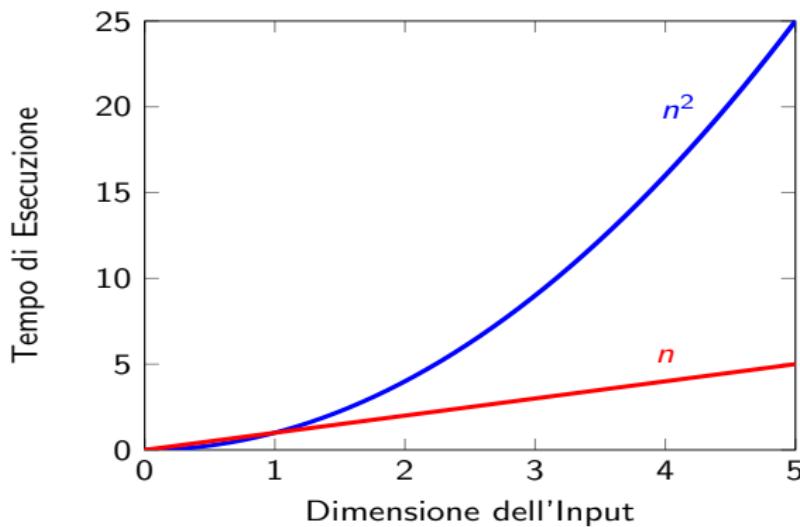
Stimiamo la relazione tra la dimensione dell'input e il tempo di esecuzione



Quiz sulla Complessità!

Quale crescita è preferibile tra:

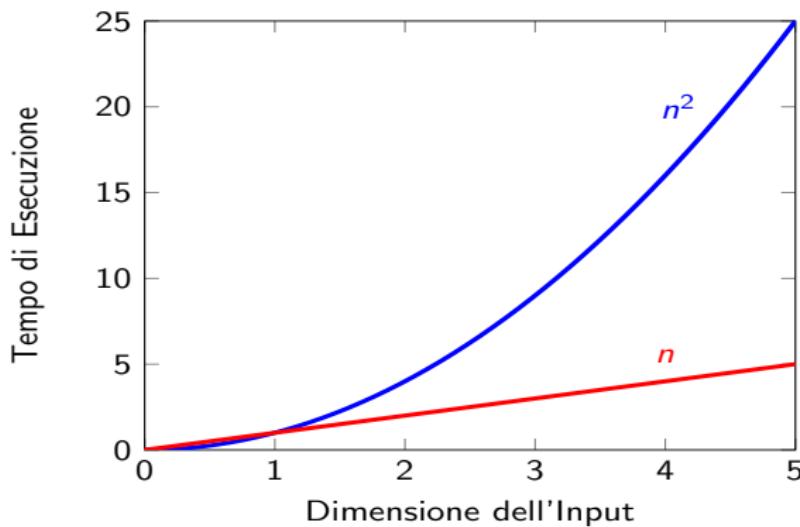
- ▶ n^2 e n ?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

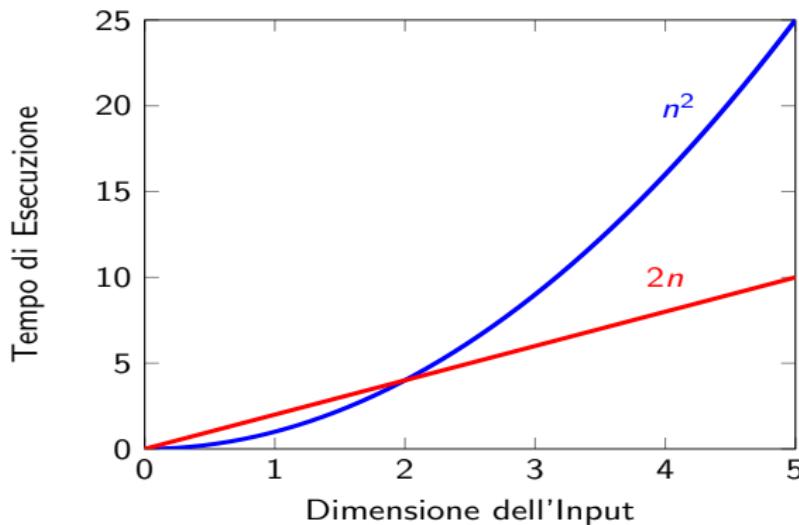
- ▶ n^2 e \underline{n} ?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

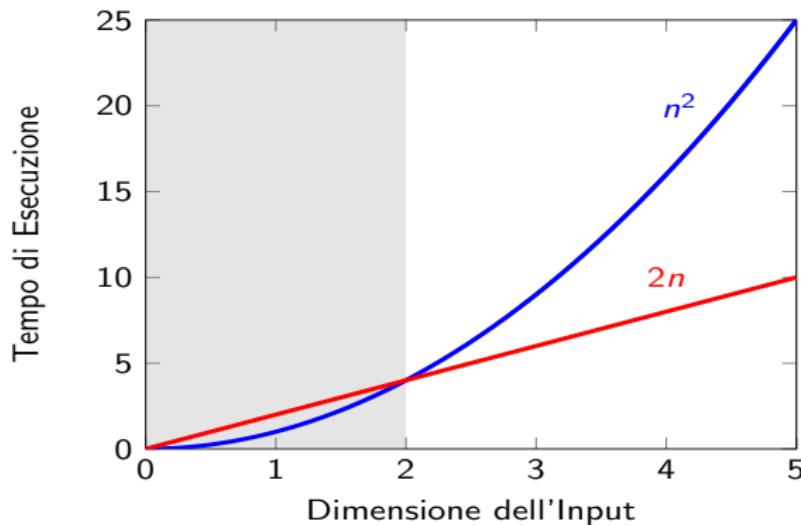
- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

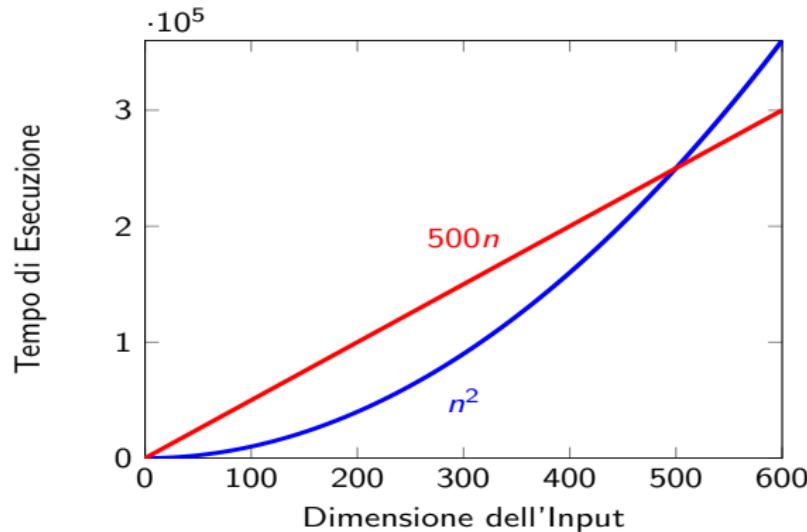
- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

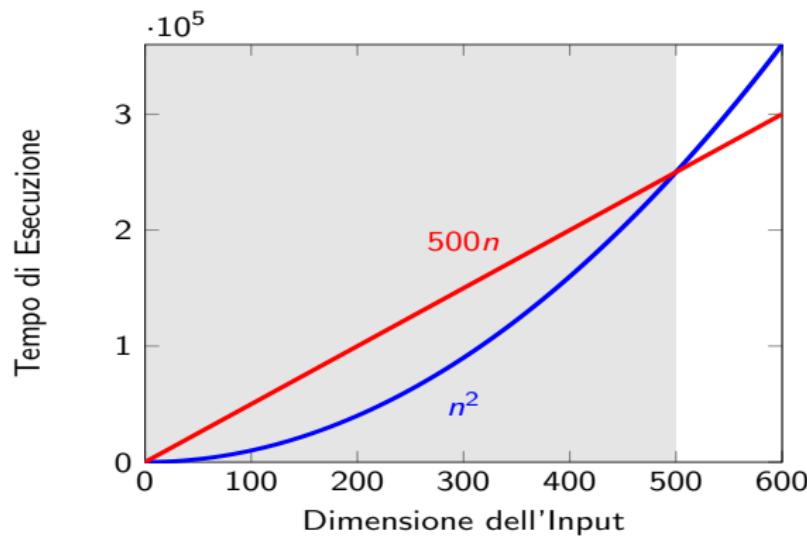
- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?
- ▶ n^2 e $500 * n$?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?
- ▶ n^2 e $500 * n$?



Complessità Asintotica

Le costanti non sono utili. Ci interessano i **comportamenti asintotici**.

La RAM e i criteri uniforme e logaritmico tornano a essere interessanti.

Possiamo astrarre il tempo di esecuzione della singola istruzione!!!

Complessità Asintotica

Le costanti non sono utili. Ci interessano i **comportamenti asintotici**.

La RAM e i criteri uniforme e logaritmico tornano a essere interessanti.

Possiamo astrarre il tempo di esecuzione della singola istruzione!!!

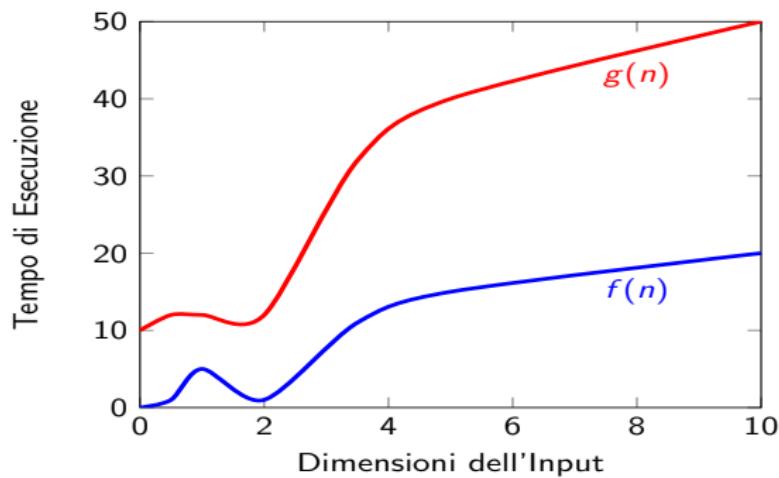
Come raggruppare tutte le funzioni che dal punto di vista asintotico si comportano nello stesso modo?

Consideriamo solo le funzioni a codominio in $\mathbb{R}_{>0}$

Notazione O-grande

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

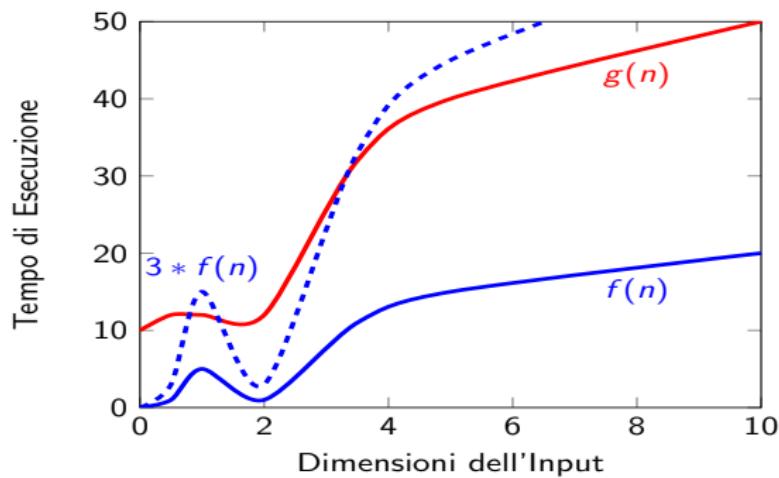
$g(n) \in O(f(n))$ se e solo se



Notazione O-grande

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

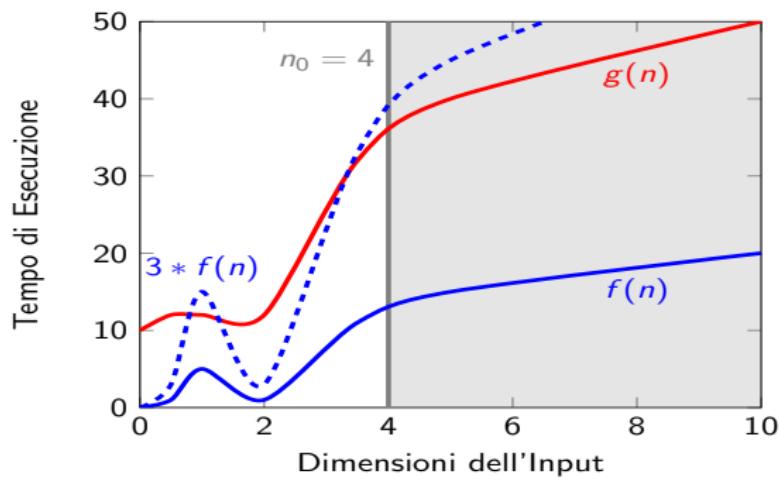
$g(n) \in O(f(n))$ se e solo se



Notazione O-grande

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

$g(n) \in O(f(n))$ se e solo se



Alcuni Utili Proprietà

Per ogni $c_1, c_2 \in \mathbb{R}_{\geq 1}$ e per ogni $k \in \mathbb{Z}$

- ▶ $f(n) \in O(f(n))$
- ▶ $O(f(n)) = O(c_1 * f(n) + k)$
- ▶ $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$ es. $n \in O(n^2)$
- ▶ se $h(n) \in O(f(n))$ e $h'(n) \in O(g(n))$, allora
 - ▶ $h(n) + h'(n) \in O(g(n) + f(n))$
 - ▶ $h(n) * h'(n) \in O(g(n) * f(n))$

Alcuni Utili Proprietà

Per ogni $c_1, c_2 \in \mathbb{R}_{\geq 1}$ e per ogni $k \in \mathbb{Z}$

- ▶ $f(n) \in O(f(n))$
- ▶ $O(f(n)) = O(c_1 * f(n) + k)$
- ▶ $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$ es. $n \in O(n^2)$
- ▶ se $h(n) \in O(f(n))$ e $h'(n) \in O(g(n))$, allora
 - ▶ $h(n) + h'(n) \in O(g(n) + f(n))$
 - ▶ $h(n) * h'(n) \in O(g(n) * f(n))$

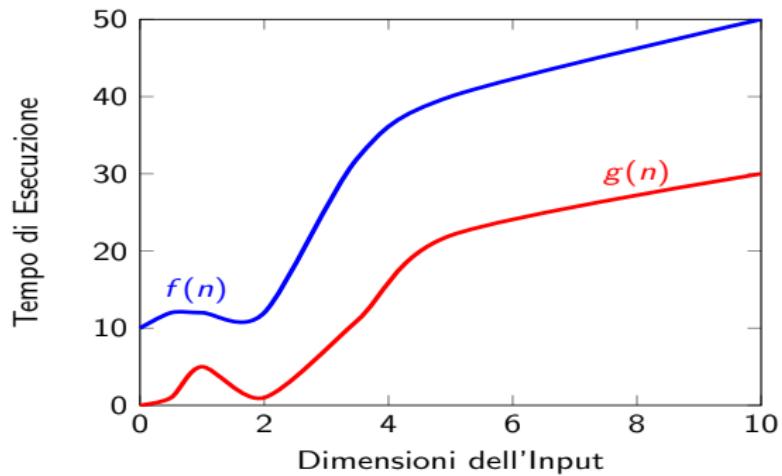
Abusando della notazione, potremmo scrivere:

- ▶ $g(n) + O(f(n))$ al posto di $O(g(n) + f(n))$
- ▶ $g(n) * O(f(n))$ intendendo $O(g(n) * f(n))$

Notazione Ω -grande

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

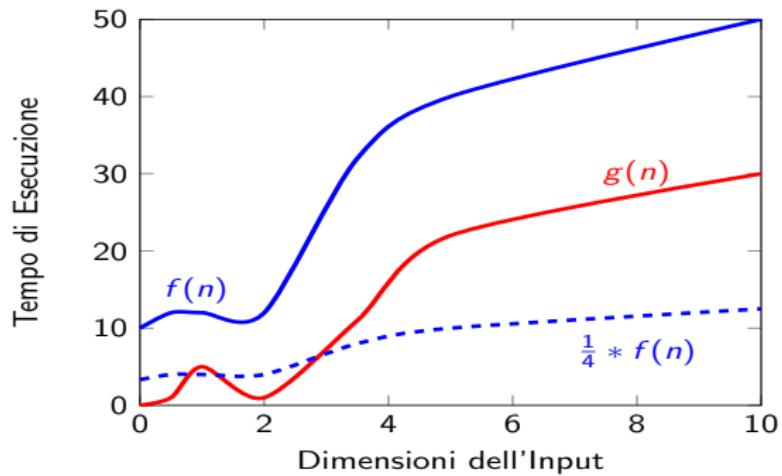
$g(n) \in \Omega(f(n))$ se e soltanto se



Notazione Ω -grande

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

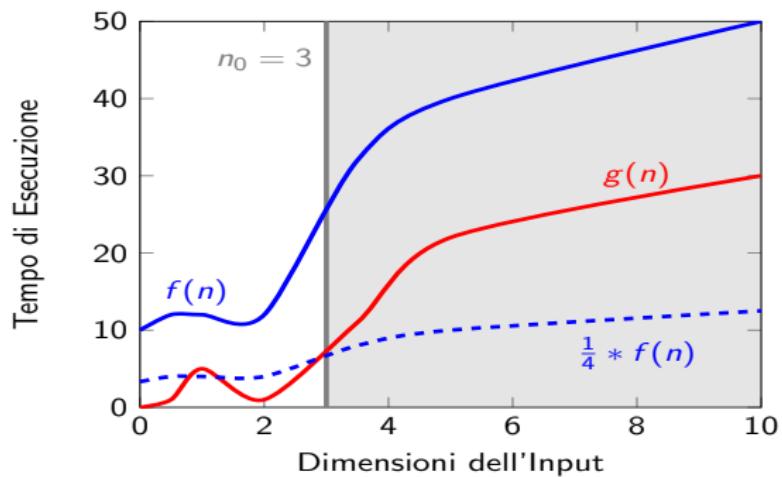
$g(n) \in \Omega(f(n))$ se e soltanto se



Notazione Ω -grande

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

$g(n) \in \Omega(f(n))$ se e soltanto se



Notazione Θ-theta

$$\Theta(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c_1, c_2 > 0 \exists n_0 > 0 \\ m \geq n_0 \implies c_1 * f(m) \leq g(m) \leq c_2 * f(m)\}$$

Theorem

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n))$$

Calcoliamo la Complessità di ...

Input: Un array A di numeri $\langle a_1, \dots, a_n \rangle$.

Output: Il massimo tra a_1, \dots, a_n .

```
1 def find_max(A):
2     max_value ← A[1]
3     for i ← 2..|A|:
4         if A[i] > max_value:
5             max_value ← A[i]
6         endif
7     endfor
8
9     return max_value
10 enddef
```

Criterio di Costo Logaritmico: l'Esempio Precedente

```
def test(n):
    Z ← 2                                // costo log 2
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo log Z
    endfor

    return Z
enddef
```

Il costo totale è

$$\begin{aligned} \log 2 + n + 1 + \sum_{i=0}^{n-1} \log 2^{2^i} &= \log 2 + n + 1 + \sum_{i=0}^{n-1} 2^i \\ &= \log 2 + n + 1 + (2^n - 1) \end{aligned}$$

MODELLO DI COMPUTAZIONE
COMPLESSITÀ DEGLI ALGORITMI

ALGORITMI E STRUTTURE DATI

Algoritmi e Modello di Computazione

Cos'è un algoritmo?

Definition (Algoritmo)

È una sequenza di passi “*ben definiti*” per trasformare in un tempo finito un insieme di dati in input in un insieme di dati in output.

“*ben definiti*” dipende dal modello di calcolo

Cos'è un algoritmo?

Definition (Algoritmo)

È una sequenza di passi “*ben definiti*” per trasformare in un tempo finito un insieme di dati in input in un insieme di dati in output.

“*ben definiti*” dipende dal modello di calcolo

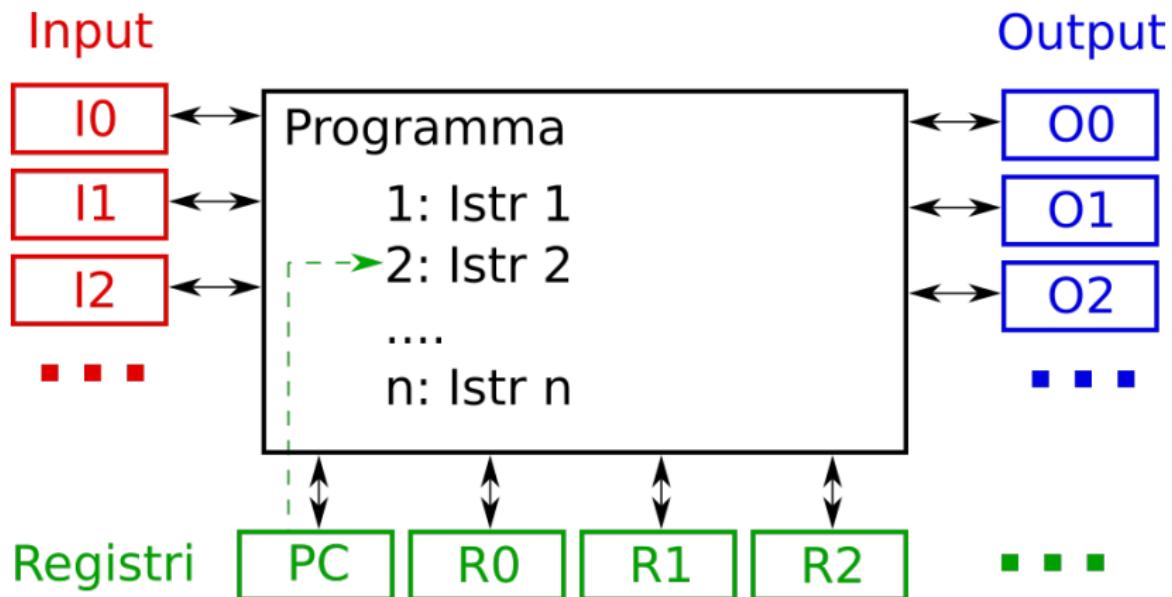
Definition (Modello di Calcolo)

È uno strumento formale per eseguire delle computazioni.

Random-Access Machine (RAM)

- ▶ memoria infinita (**registri**)
- ▶ ogni registro può contenere un qualsiasi numero naturale
- ▶ **input** (sola lettura) e **output** (sola scrittura)
- ▶ il programma è una sequenza di istruzioni
- ▶ il **Program Counter** è un registro che indica la prossima istruzione da eseguire
- ▶ ad ogni istruzione eseguita il **PC** viene automaticamente incrementato (se non cambiato altrimenti)

Random-Access Machine (RAM)



Istruzioni della RAM

Istruzioni di calcolo

- ▶ **CLR(r)** assegna al registro Rr il valore 0;
- ▶ **INC(r)** assegna al registro Rr il valore contenuto in Rr + 1;
- ▶ ***r** ottieni il valore contenuto nel registro r.

Es., Se $*5=8$, allora $**5$ è il contenuto di R8;

- ▶ **JE(*r,s,j)** se $*r=s$ assegna j a PC
- ▶ **HLT** termina l'esecuzione del programma

Cosa possiamo fare con questo numero limitato di istruzioni?

Assegnamento

Il seguente programma assegna il valore v a Rr .

```
1: CLR( r )
2: INC( r )
...
v+1: INC( r )
```

Possiamo estendere l'insieme delle istruzioni della RAM aggiungendo l'**assegnamento** (\leftarrow).

Es.,

```
1: Rr ← v
```

Relazione d'ordine

Verifica se $*s < *r$

- | | |
|----------------|-----------------|
| 1: JE(*r,*s,9) | 7: JE(*1,*r,11) |
| 2: R0 ← *r | 8: JE(0,0,4) |
| 3: R1 ← *s | 9: R0 ← 0 |
| 4: INC(0) | 10: JE(0,0,12) |
| 5: JE(*0,*s,9) | 11: R0 ← 1 |
| 6: INC(1) | 12: ... |

Il risultato è in R0

Possiamo aggiungere le **relazioni l'ordine** (\leq , $<$, $=$, $>$, e \geq).

Espresioni Booleane

Se intepretiamo 0 come FALSO

Negazione di *s

- 1: JE(*s, 0, 4)
- 2: R0 ← 0
- 3: JE(0, 0, 5)
- 4: R0 ← 1
- 5: ...

Disgiunzione logica Rr ∨ Rs

- 1: R0 ← 1
- 2: JE(*r, 1, 5)
- 3: JE(*s, 1, 5)
- 4: R0 ← 0
- 5: ...

Possiamo aggiungere la **logica Booleana** (\vee , \wedge , e \neg).

Es.,

$$1: Rr \leftarrow \neg(*r \vee *s)$$

Costrutti Condizionali e Cicli

If *s then A else B

```
1:    JE(*s, 0, c)
      A
      JE(0, 0, d)
c:    B
d:    ...
```

While *s do A

```
1:    JE(*s, 0, c+1)
      A
c:    JE(0, 0, 1)
c+1:   ...
```

Possiamo aggiungere i costrutti **if-then-else** e **while-do**.

Es.,

```
1:  if  ¬(*r ∨ *s)
2:    Rs ← c
3:  endif
```

```
1:  while  ¬(*r ∨ *s)
2:    Rs ← c
3:  endwhile
```

Somma e Sottrazione

Somma Rs a Rr

- 1: $R0 \leftarrow 0$
- 2: **while** $\neg(*0 == *s)$
- 3: INC(0)
- 4: INC(r)
- 5: **endwhile**

Il risultato è in Rr

Sottrae Rs da Rr

- 1: $R0 \leftarrow 0$
- 2: **while** $(*r > *s)$
- 3: INC(0)
- 4: INC(s)
- 5: **endif**

Il risultato è in R0

Possiamo aggiungere **somma** (+) e **sottrazione** (-).

Es.,

- 1: $Rr \leftarrow *r + *s$
- 2: $Rs \leftarrow *r - 5$

Moltiplicazione e Divisione

Moltiplica Rr per Rs

```
1: R0 ← 0  
2: R1 ← 0  
2: while (*s>*1)  
3:     R0 ← *0 + *r  
4:     R1 ← *1 + 1  
5: endwhile
```

Dividi Rr per Rs

```
1: R0 ← 0  
2: R1 ← 0  
2: while (*r≥*1+*s)  
3:     R1 ← *1 + *s  
4:     R0 ← *0 + 1  
5: endwhile  
6: R1 ← *r - *1
```

Possiamo aggiungere **moltiplicazione** (*) e **divisione** (/).

E.s.,

```
1: Rr ← *r * *s  
2: Rs ← *r / 5
```

La nostra RAM

In sostanza possiamo semplificare la RAM e assumere di avere:

- ▶ variabili (no tipi)
- ▶ array
- ▶ costanti intere e floating point
- ▶ funzioni algebriche: $+$, $-$, $/$, $*$, $\lfloor \cdot \rfloor$, $\lceil \cdot \rceil$
- ▶ puntatori
- ▶ istruzioni condizionali e cicli (while e for)
- ▶ procedure e ricorsione (provate a capire come)
- ▶ semplici funzioni “ragionevoli” come $|\cdot|$

... e rimuovere gli indirizzi nel codice

Un Semplice Algoritmo

Input: Un array A di numeri $\langle a_1, \dots, a_n \rangle$.

Output: Il massimo tra a_1, \dots, a_n .

```
def find_max(A):
    max_value ← A[1]
    for i ← 2..|A|:
        if A[i] > max_value:
            max_value ← A[i]
        endif
    endfor

    return max_value
enddef
```

La RAM NON è una Macchina Reale!!!

RAM rappresenta le macchine fisiche, ma non ha:

- ▶ la finitezza della memoria
- ▶ la finitezza della rappresentazione dei numeri
- ▶ le gerarchie della memoria

Complessità degli Algoritmi

Complessità degli Algoritmi

Complessità in termini di tempo: è la somma dei costi delle istruzioni da eseguire

Complessità in termini di spazio: è la somma dello spazio occupato

Qual'è il costo delle operazioni della RAM?

Complessità degli Algoritmi

Complessità in termini di tempo: è la somma dei costi delle istruzioni da eseguire (**dipende dal modello di calcolo**)

Complessità in termini di spazio: è la somma dello spazio occupato

Qual'è il costo delle operazioni della RAM?

Criterio di Costo Uniforme

Il costo in termini di tempo di:

- ▶ una operazione $+, -, *, /$ è 1
- ▶ un'operazione di riferimento in memoria è 1
- ▶ un'istruzione di controllo è 1
- ▶ un assegnamento è 1

Il costo in termini di spazio di un registro è 1

Criterio di Costo Uniforme: un Esempio

```
def test(n):
    Z ← 2                                // costo 1
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo 2
    endfor

    return Z
enddef
```

Il costo totale è $1 + (n + 1) * 1 + n * 2$

Criterio di Costo Uniforme: un Esempio

```
def test(n):
    Z ← 2                                // costo 1
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo 2
    endfor

    return Z
enddef
```

Il costo totale è $1 + (n + 1) * 1 + n * 2$ e alla fine $test(n) = 2^{2^n} \dots$

in tempo lineare l'output occupa spazio $2^n!!!$

Criterio di Costo Logaritmico

Il costo in termini di tempo di:

- ▶ una operazione $a \cdot b$ con $\cdot \in \{+, -, *, /\}$ è $\max(\log a, \log b)$
- ▶ un'operazione di riferimento in memoria è 1
- ▶ un'istruzione di controllo è 1
- ▶ un assegnamento è 1

Il costo in termini di spazio di un registro è log del valore memorizzato

Criterio di Costo Logaritmico: l'Esempio Precedente

```
def test(n):
    Z ← 2                                // costo log 2
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo log Z
    endfor

    return Z
enddef
```

Il costo totale è

$$\begin{aligned}\log 2 + n + 1 + \sum_{i=0}^{n-1} \log 2^{2^i} &= \log 2 + n + 1 + \sum_{i=0}^{n-1} 2^i \\ &= \log 2 + n + 1 + (2^n - 1)\end{aligned}$$

Costo Uniforme vs Costo Logaritmico

Il costo logaritmico modella algoritmi che lavorano con grandi numeri.

Se lo spazio occupato dai valori è limitato (come nei computer reali), il costo uniforme va benissimo.

Come Confrontare l'Efficienza?

Tempo di esecuzione?

Come Confrontare l'Efficienza?

Tempo di esecuzione? (per quale input?)

Gli algoritmi non sono programmi

Assumere costo uniforme/logaritmico non sembra essere realistico perché il tempo di esecuzione dipende da:

- ▶ l'insieme delle istruzioni della CPU
- ▶ Clock di CPU/Memoria/Bus
- ▶ il linguaggio e il compilatore usati
- ▶ come il sistema operativo gestisce la memoria
- ▶ ...

Come Confrontare l'Efficienza?

~~Cosa ne dite del tempo di esecuzione?~~

Qualche altra idea?

Come Confrontare l'Efficienza?

~~Cosa ne dite del tempo di esecuzione?~~

Qualche altra idea?

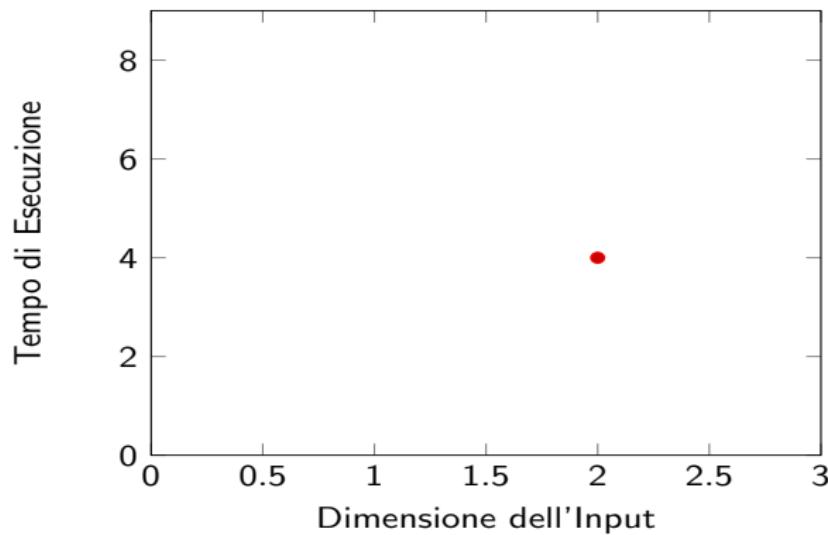
Scalabilità?

Definition (Scalabilità)

Efficienza di un sistema nel gestire crescite nella dimensione dell'input.

Crescita della Complessità

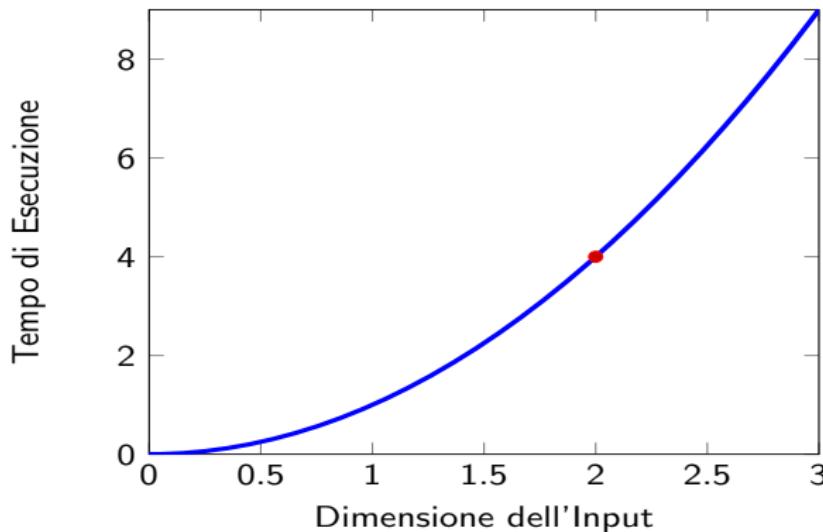
Non misuriamo il tempo di esecuzione **per un dato input**



Crescita della Complessità

Non misuriamo il tempo di esecuzione **per un dato input**

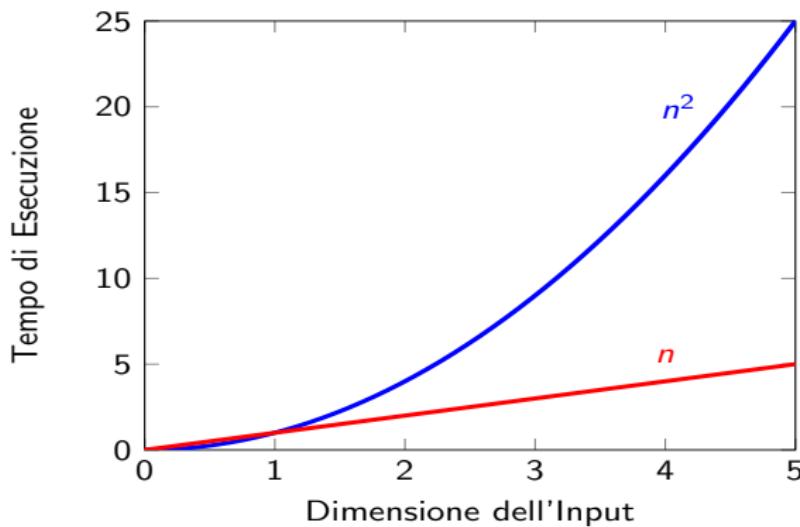
Stimiamo la relazione tra la dimensione dell'input e il tempo di esecuzione



Quiz sulla Complessità!

Quale crescita è preferibile tra:

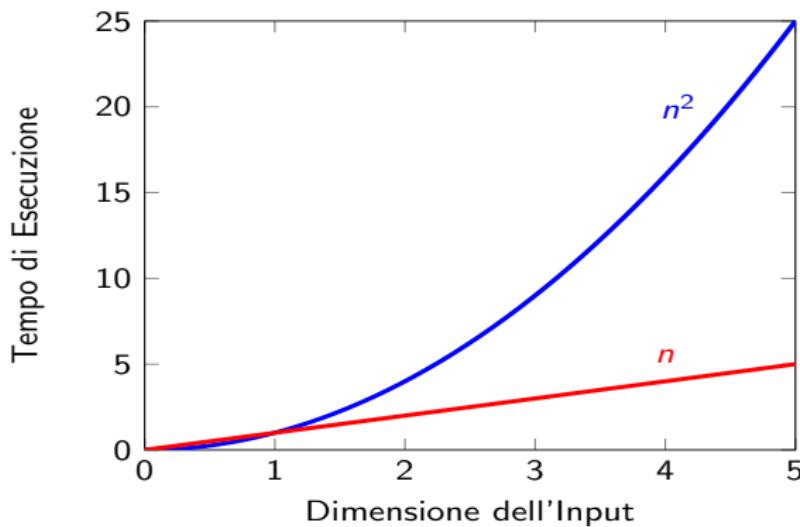
- ▶ n^2 e n ?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

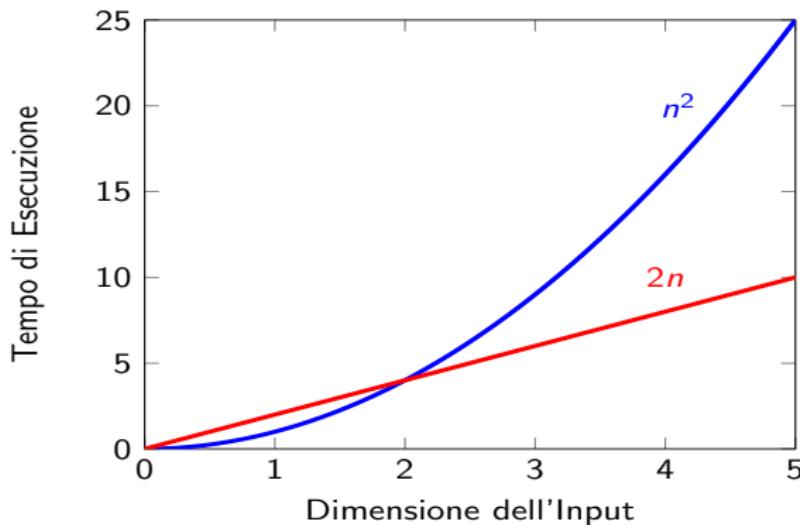
- ▶ n^2 e \underline{n} ?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

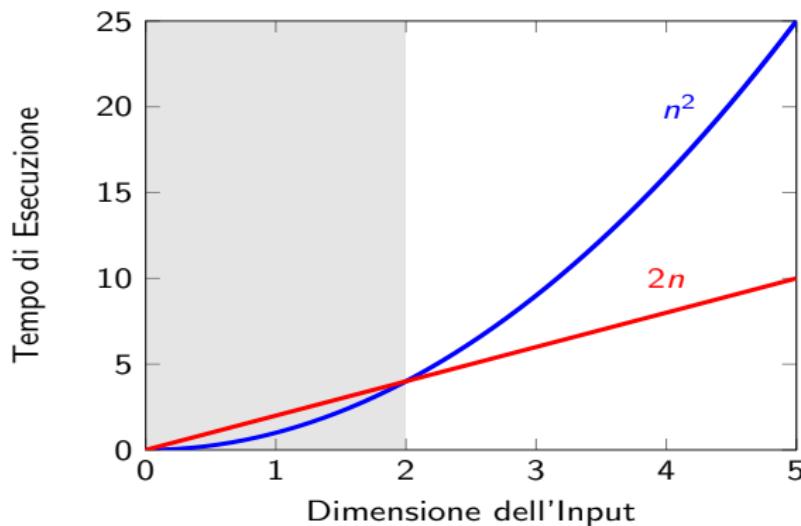
- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

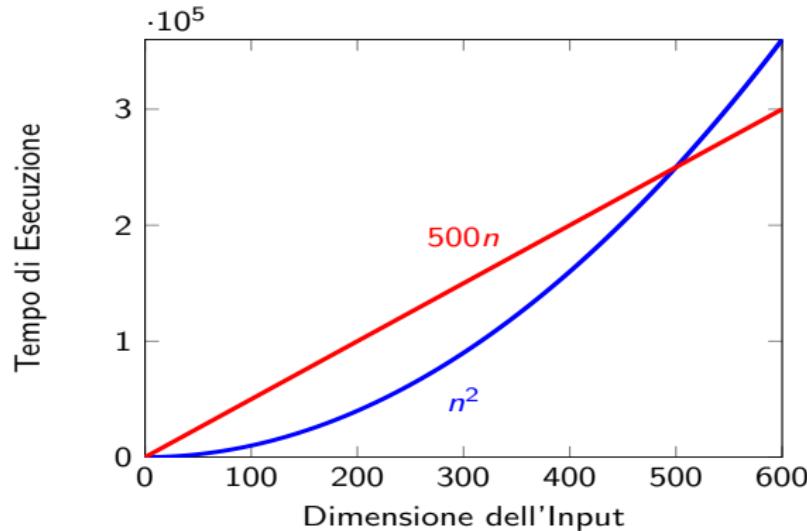
- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

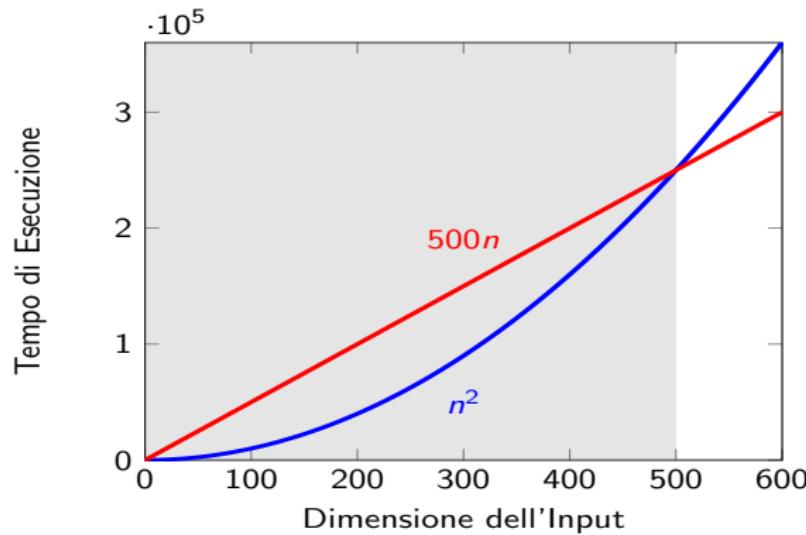
- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?
- ▶ n^2 e $500 * n$?



Quiz sulla Complessità!

Quale crescita è preferibile tra:

- ▶ n^2 e n ?
- ▶ n^2 e $2 * n$?
- ▶ n^2 e $500 * n$?



Complessità Asintotica

Le costanti non sono utili. Ci interessano i **comportamenti asintotici**.

La RAM e i criteri uniforme e logaritmico tornano a essere interessanti.

Possiamo astrarre il tempo di esecuzione della singola istruzione!!!

Complessità Asintotica

Le costanti non sono utili. Ci interessano i **comportamenti asintotici**.

La RAM e i criteri uniforme e logaritmico tornano a essere interessanti.

Possiamo astrarre il tempo di esecuzione della singola istruzione!!!

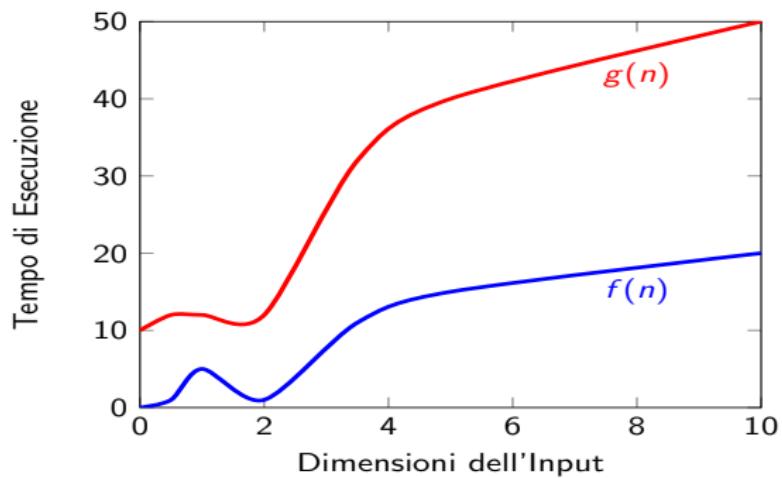
Come raggruppare tutte le funzioni che dal punto di vista asintotico si comportano nello stesso modo?

Consideriamo solo le funzioni a codominio in $\mathbb{R}_{>0}$

Notazione O-grande

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

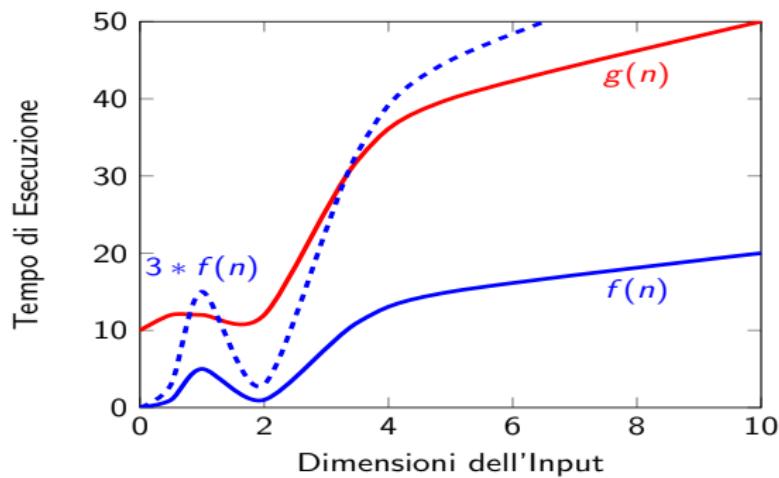
$g(n) \in O(f(n))$ se e solo se



Notazione O-grande

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

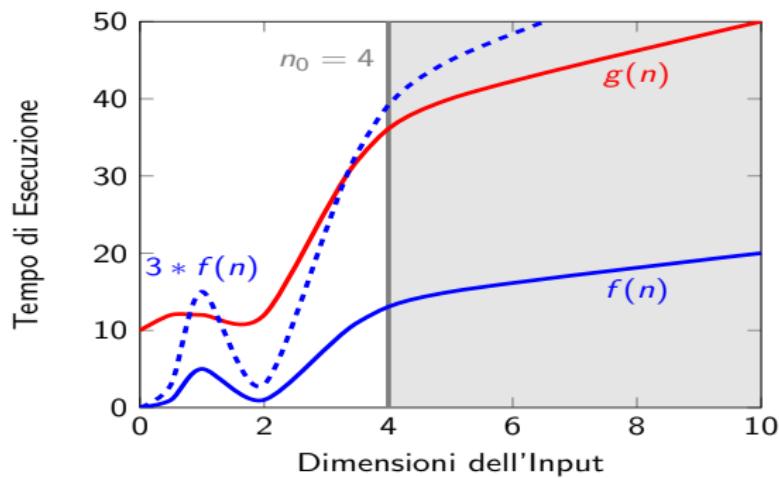
$g(n) \in O(f(n))$ se e solo se



Notazione O-grande

$$O(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies g(m) \leq c * f(m)\}$$

$g(n) \in O(f(n))$ se e solo se



Alcuni Utili Proprietà

Per ogni $c_1, c_2 \in \mathbb{R}_{\geq 1}$ e per ogni $k \in \mathbb{Z}$

- ▶ $f(n) \in O(f(n))$
- ▶ $O(f(n)) = O(c_1 * f(n) + k)$
- ▶ $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$ es. $n \in O(n^2)$
- ▶ se $h(n) \in O(f(n))$ e $h'(n) \in O(g(n))$, allora
 - ▶ $h(n) + h'(n) \in O(g(n) + f(n))$
 - ▶ $h(n) * h'(n) \in O(g(n) * f(n))$

Alcuni Utili Proprietà

Per ogni $c_1, c_2 \in \mathbb{R}_{\geq 1}$ e per ogni $k \in \mathbb{Z}$

- ▶ $f(n) \in O(f(n))$
- ▶ $O(f(n)) = O(c_1 * f(n) + k)$
- ▶ $O(f(n)^{c_1}) \subseteq O(f(n)^{c_1+c_2})$ es. $n \in O(n^2)$
- ▶ se $h(n) \in O(f(n))$ e $h'(n) \in O(g(n))$, allora
 - ▶ $h(n) + h'(n) \in O(g(n) + f(n))$
 - ▶ $h(n) * h'(n) \in O(g(n) * f(n))$

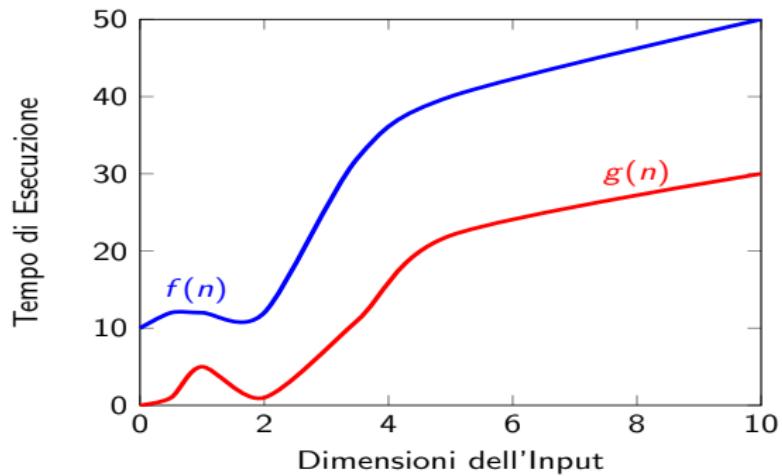
Abusando della notazione, potremmo scrivere:

- ▶ $g(n) + O(f(n))$ al posto di $O(g(n) + f(n))$
- ▶ $g(n) * O(f(n))$ intendendo $O(g(n) * f(n))$

Notazione Ω -grande

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

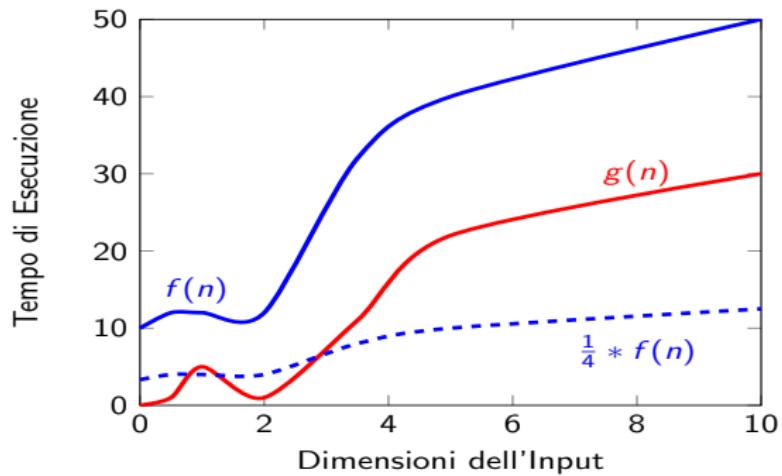
$g(n) \in \Omega(f(n))$ se e soltanto se



Notazione Ω -grande

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

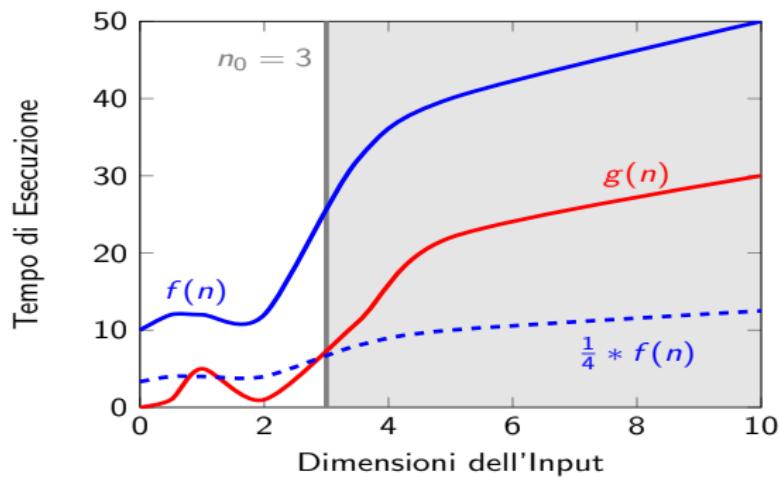
$g(n) \in \Omega(f(n))$ se e soltanto se



Notazione Ω -grande

$$\Omega(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c > 0 \exists n_0 > 0 \ m \geq n_0 \implies c * f(m) \leq g(m)\}$$

$g(n) \in \Omega(f(n))$ se e soltanto se



Notazione Θ-theta

$$\Theta(f(n)) \stackrel{\text{def}}{=} \{g(n) \mid \exists c_1, c_2 > 0 \exists n_0 > 0 \\ m \geq n_0 \implies c_1 * f(m) \leq g(m) \leq c_2 * f(m)\}$$

Theorem

$$f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \cap \Omega(g(n))$$

Calcoliamo la Complessità di ...

Input: Un array A di numeri $\langle a_1, \dots, a_n \rangle$.

Output: Il massimo tra a_1, \dots, a_n .

```
1 def find_max(A):
2     max_value ← A[1]
3     for i ← 2..|A|:
4         if A[i] > max_value:
5             max_value ← A[i]
6         endif
7     endfor
8
9     return max_value
10 enddef
```

Criterio di Costo Logaritmico: l'Esempio Precedente

```
def test(n):
    Z ← 2                                // costo log 2
    for i ← 1..n:                          // n+1 volte: costo 1
        Z ← Z * Z                         // n volte: costo log Z
    endfor

    return Z
enddef
```

Il costo totale è

$$\begin{aligned} \log 2 + n + 1 + \sum_{i=0}^{n-1} \log 2^{2^i} &= \log 2 + n + 1 + \sum_{i=0}^{n-1} 2^i \\ &= \log 2 + n + 1 + (2^n - 1) \end{aligned}$$

RECUPERO DEI DATI

ALGORITMI DI ORDINAMENTO

ALGORITMI E STRUTTURE DATI

Indicizzazione

Recupero dei Dati

$A = \langle a_1, \dots, a_n \rangle$ contiene alcuni dati, es. dati medici di pazienti

Ciascun elemento è associato a un **identificatore**, $A[i].id$, es.
codice fiscale

Come trovare i dati associati all'identificatore id_1 ?

La Soluzione Naïve

Scandiamo tutta la base di dati cercando un i per cui $A[i].id = id_1$

Qual'è la complessità asintotica?

La Soluzione Naïve

Scandiamo tutta la base di dati cercando un i per cui $A[i].id = id_1$

Qual'è la complessità asintotica? $O(n)$

Possiamo fare meglio?

Suggerimento: Come cerchiamo una pagina in un libro? E una parola nel dizionario? Perché?

Una Tecnica più Efficiente: La Ricerca Binaria

Se $A = \langle a_1, \dots, a_n \rangle$ è ordinato rispetto agli id...

(cioè, $i < j$ implica $A[i].id \leq A[j].id$)

Una Tecnica più Efficiente: La Ricerca Binaria

Se $A = \langle a_1, \dots, a_n \rangle$ è ordinato rispetto agli id...

(cioè, $i < j$ implica $A[i].id \leq A[j].id$)

Consideriamo il valore della mediana (cioè $A[n/2]$)

Una Tecnica più Efficiente: La Ricerca Binaria

Se $A = \langle a_1, \dots, a_n \rangle$ è ordinato rispetto agli id...

(cioè, $i < j$ implica $A[i].id \leq A[j].id$)

Consideriamo il valore della mediana (cioè $A[n/2]$)

se $A[n/2].id = id_1$

Trovato!

se $A[n/2].id > id_1$

Concentriamoci sulla I metà di A , i.e., $\langle a_1, \dots, a_{n/2-1} \rangle$

if $A[n/2].id < id_1$

Concentriamoci sulla II metà di A , i.e., $\langle a_{n/2+1}, \dots, a_n \rangle$

Ripetiamo finchè A non è vuoto

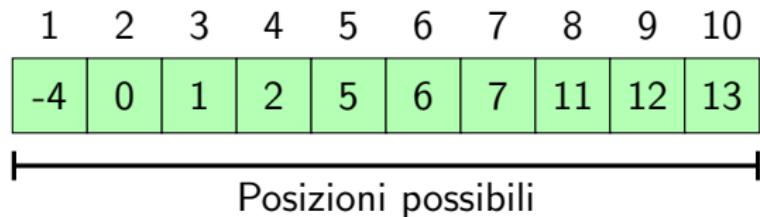
Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.

| | | | | | | | | | |
|----|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| -4 | 0 | 1 | 2 | 5 | 6 | 7 | 11 | 12 | 13 |

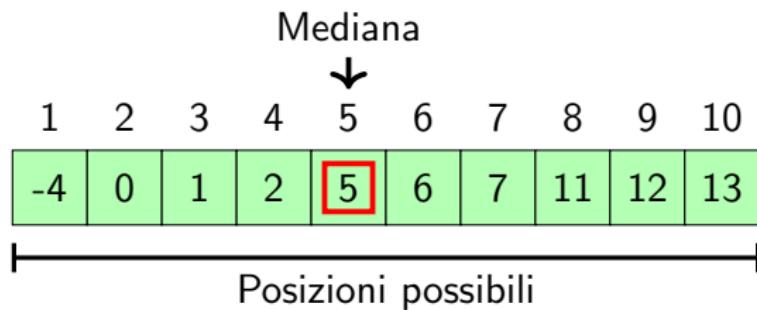
Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.

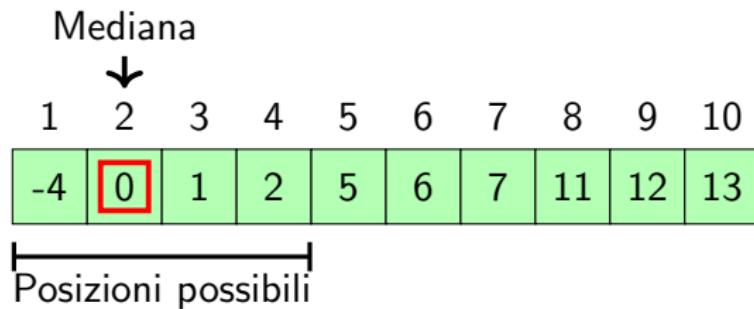
| | | | | | | | | | |
|----|---|---|---|---|---|---|----|----|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| -4 | 0 | 1 | 2 | 5 | 6 | 7 | 11 | 12 | 13 |

Posizioni possibili



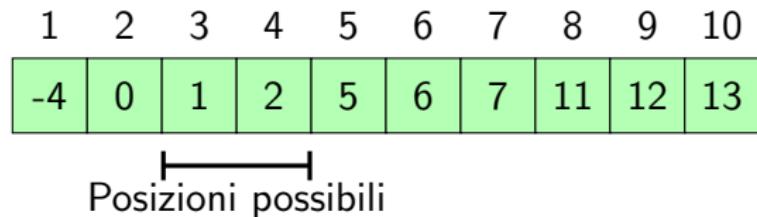
Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



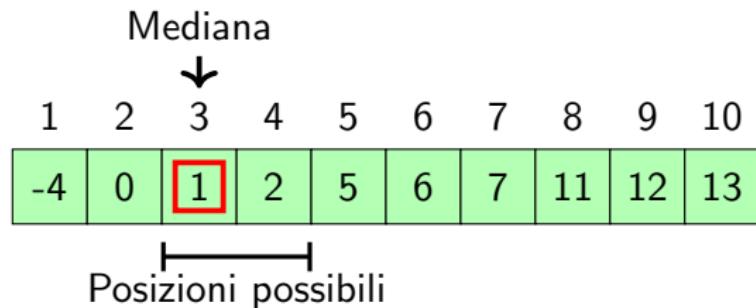
Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



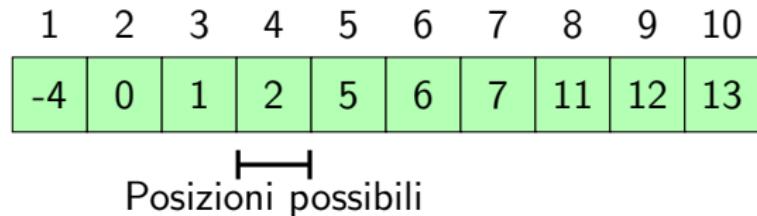
Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



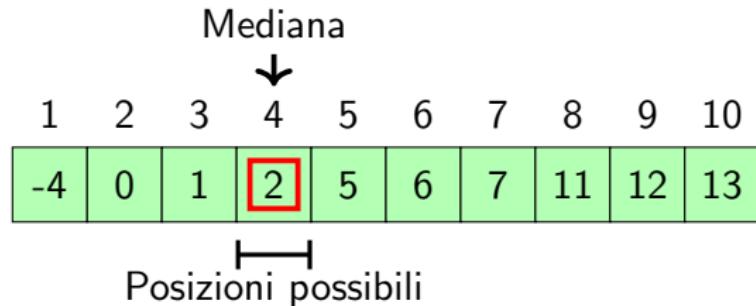
Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



Ricerca Dicotomica: Un Esempio

Cerchiamo 2 in $\langle -4, 0, 1, 2, 5, 6, 7, 11, 12, 13 \rangle$.



Trovato: $A[4] = 2$

Ricerca Dicotomica: Pseudo-Codice e Complessità

```
def di_find(A, a):
    (l, r) ← (1, |A|)
    while r ≥ l
        m ← (l+r)/2
        if A[m]=a
            return m
        endif
        if A[m]>a
            r ← m-1
        else
            l ← m+1
        endif
    endwhile
    return 0
enddef
```

$l - r$ si dimezza a ogni iterazione

Se $|A| = 2^n$, di_find termina in al più n iterazioni

Il blocco del while costa $\Theta(1)$

La complessità di di_find è

$O(\log n)$

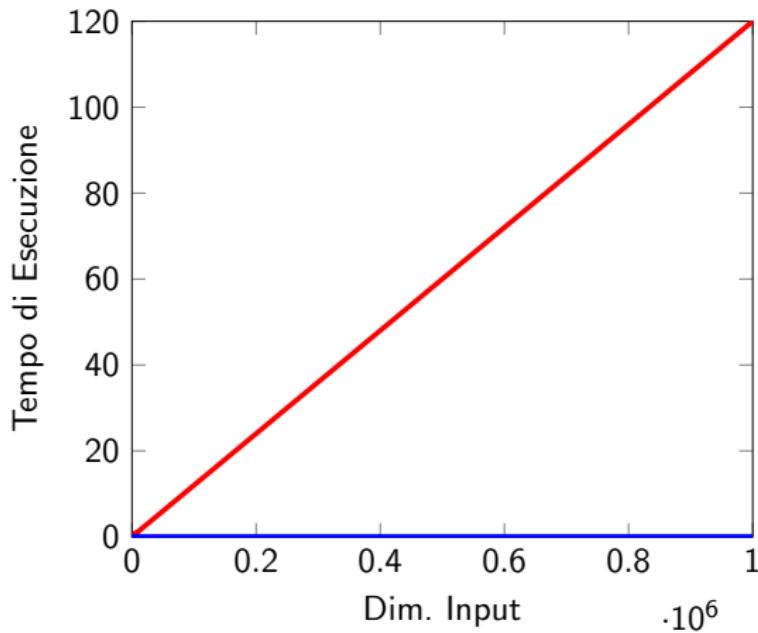
Ricerca Dicotomica vs Ricerca Lineare: un Test

Tempo di esecuzione per 1×10^5 ricerche casuali.

| Dim. Input | Ricerca Lineare | Ricerca Dicotomica |
|-----------------|------------------------|------------------------|
| 1×10^1 | 3.3×10^{-3} s | 3.2×10^{-3} s |
| 1×10^2 | 1.4×10^{-2} s | 4.3×10^{-3} s |
| 1×10^3 | 1.2×10^{-1} s | 5.9×10^{-3} s |
| 1×10^4 | 1.2 s | 7.8×10^{-3} s |
| 1×10^5 | 1.2×10^1 s | 8.7×10^{-3} s |
| 1×10^6 | 1.2×10^2 s | 1.2×10^{-2} s |

Ricerca Dicotomica vs Ricerca Lineare: un Test

Tempo di esecuzione per 1×10^5 ricerche casuali.



Ordinamento

Il Problema dell'Ordinamento

Input: Un array A di numeri

Output: L'array A ordinato, cioè, se $i < j$ allora $A[i] \leq A[j]$

Ese.,

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|----|---|---|----|---|----|
| 13 | 5 | 7 | 2 | -4 | 4 | 1 | 11 | 6 | 0 |

↓

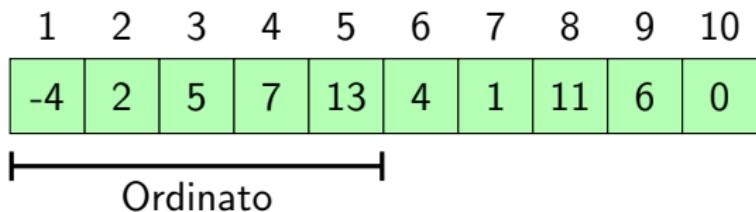
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|---|---|---|---|----|----|
| -4 | 0 | 1 | 2 | 4 | 5 | 6 | 7 | 11 | 13 |

Qualche suggerimento su un possibile algoritmo? Qual'è la complessità attesa?

Insertion Sort

Insertion Sort: Intuizione

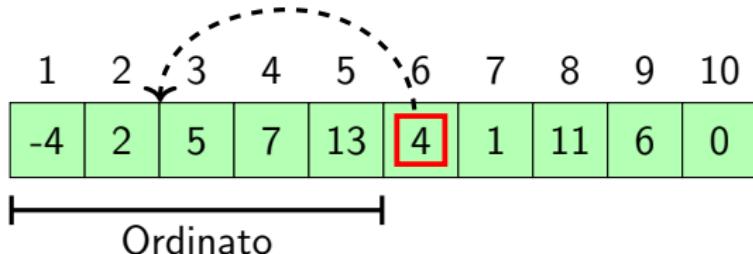
Se la prima parte dell'array è già ordinata



Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

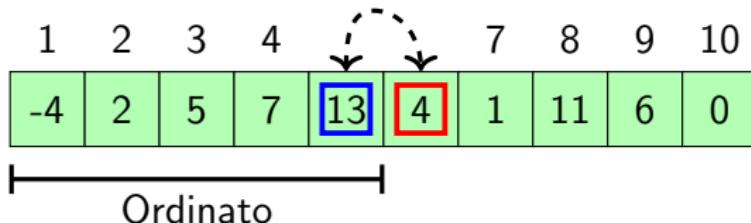


Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

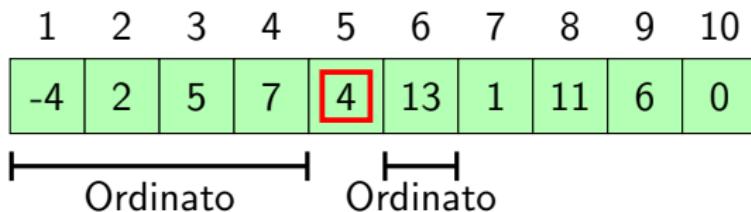


Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

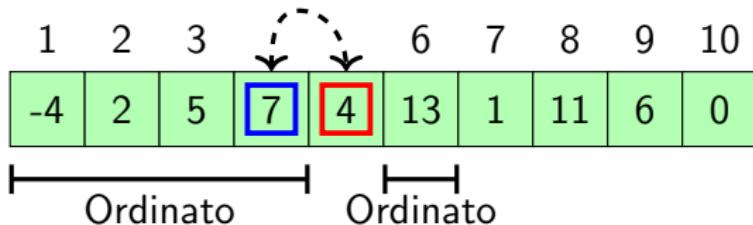


Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

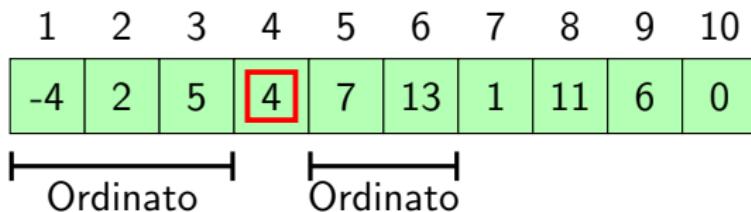


Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

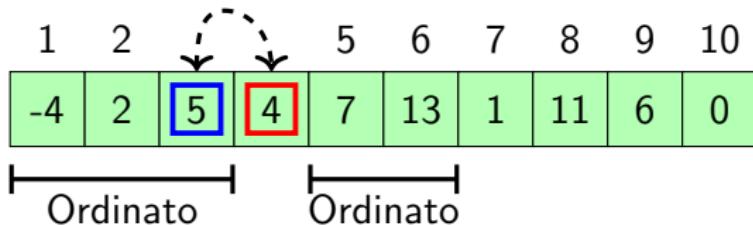


Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**



Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

finchè **p** (se esiste) è più grande di **v**

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|---|----|---|----|---|----|
| -4 | 2 | 4 | 5 | 7 | 13 | 1 | 11 | 6 | 0 |

Ordinato ≤ 4 Ordinato > 4

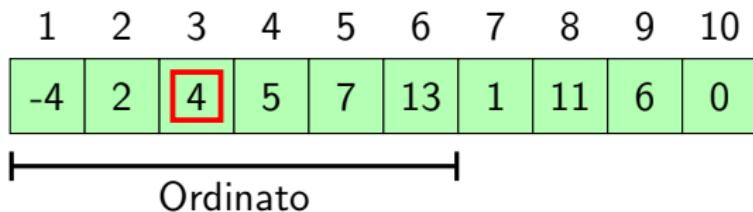
Insertion Sort: Intuizione

Se la prima parte dell'array è già ordinata

possiamo “allargarla” inserendo il valore successivo **v** nella posizione corretta

scambiando **v** con il valore precedente nell'array **p**

finchè **p** (se esiste) è più grande di **v**



Insertion Sort: Codice e Complessità

```
def insertion_sort(A):
    for i in 2..|A|:
        j ← i
        while (j > 1 and
               A[j] < A[j - 1]):
            swap(A, j - 1, j)
            j ← j - 1
    endwhile
endfor
enddef
```

Il blocco del while costa $\Theta(1)$

Viene iterato $O(i)$ ($\Omega(1)$) volte
per ogni $i \in [2, n]$

$$\sum_{i=2}^n O(i) * O(1) = O\left(\sum_{i=2}^n i\right) \\ = O(n^2)$$

$$\sum_{i=2}^n \Omega(1) * \Omega(1) = \Omega\left(\sum_{i=2}^n 1\right) \\ = \Omega(n)$$

Bolle, Selezioni e Varie

Ordinare Scegliendo il Massimo

Trova il massimo

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|----|---|---|----|---|----|
| 13 | 5 | 7 | 2 | -4 | 4 | 1 | 11 | 6 | 0 |

Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

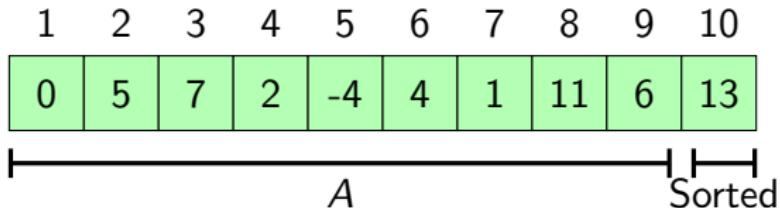
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|----|---|---|----|---|----|
| 0 | 5 | 7 | 2 | -4 | 4 | 1 | 11 | 6 | 13 |

Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se $|A| > 1$, ripeti sul frammento iniziale di A

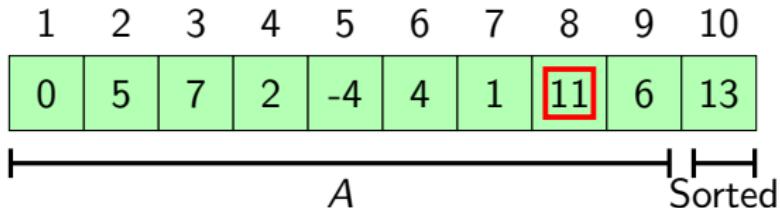


Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se $|A| > 1$, ripeti sul frammento iniziale di A

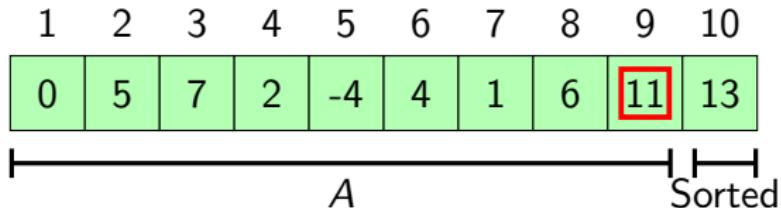


Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se $|A| > 1$, ripeti sul frammento iniziale di A

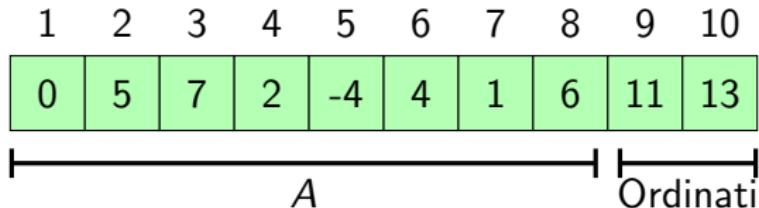


Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se $|A| > 1$, ripeti sul frammento iniziale di A

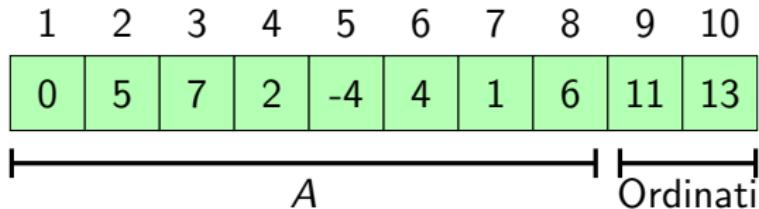


Ordinare Scegliendo il Massimo

Trova il massimo

Sposta il massimo alla fine dell'array

Se $|A| > 1$, ripeti sul frammento iniziale di A



La complessità è $\sum_{i=1}^{|A|} (T_{\max}(i) + \Theta(1))$

Come Trovare il Massimo?

Usando ...

- ▶ spostandolo verso destra con scambi \implies Bubble Sort

$$T(|A|) \in \sum_{i=1}^{|A|} (\Theta(i) + O(i)) = \Theta(|A|^2)$$

- ▶ ricerca lineare sulla porzione non ordinata \implies Selection Sort

$$T(|A|) \in \sum_{i=1}^{|A|} (\Theta(i) + \Theta(1)) = \Theta(|A|^2)$$

Bubble Sort: Pseudo-Codice e Complessità

```
def bubble_sort(A):
    for i in |A|..2:
        for j in 1..i-1:
            if A[j] > A[j+1]:
                swap(A, j, j+1)
            endif
        endfor
    endfor
enddef
```

il costrutto if costa $\Theta(1)$

Viene iterato $i - 1$ volte per
ogni $i \in [2, n]$

$$\sum_{i=2}^n \sum_{j=1}^{i-1} \Theta(1) = \Theta(n^2)$$

Selection Sort: Pseudo-Codice e Complessità

```
def selection_sort(A):
    for i in |A|..2:
        max_i ← i
        for j in 1..i-1:
            if A[j] > A[max_i]:
                max_i ← j
            endif
        endfor
        swap(A, max_i, i)
    endfor
enddef
```

il costrutto **if**, l'assegnamento e lo scambio costano $\Theta(1)$

il costrutto **if** viene iterato $i - 1$ volte per ogni $i \in [2, n]$

$$\sum_{i=2}^n \left(\Theta(1) + \sum_{j=1}^{i-1} \Theta(1) \right) = \Theta(n^2)$$

Merge Sort

Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|----|---|---|----|---|----|
| 13 | 5 | 7 | 2 | -4 | 4 | 1 | 11 | 6 | 0 |

Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un idea per ordinare un array:

1. ordino la prima metà

| | | | | | | | | | |
|----|---|---|---|----|---|---|----|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 13 | 5 | 7 | 2 | -4 | 4 | 1 | 11 | 6 | 0 |



Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un idea per ordinare un array:

1. ordino la prima metà

| | | | | | | | | | |
|----|---|---|---|----|---|---|----|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| -4 | 2 | 5 | 7 | 13 | 4 | 1 | 11 | 6 | 0 |



Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà
2. ordino la seconda metà

| | | | | | | | | | |
|----|---|---|---|----|---|---|----|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| -4 | 2 | 5 | 7 | 13 | 4 | 1 | 11 | 6 | 0 |



Divide-et-Impera

Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà
2. ordino la seconda metà

| | | | | | | | | | |
|----|---|---|---|----|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| -4 | 2 | 5 | 7 | 13 | 0 | 1 | 4 | 6 | 11 |



Divide-et-Impera

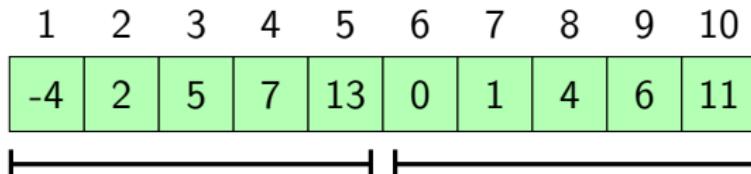
Il paradigma divide-et-impera è uno strumento tipico di progettazione degli algoritmi.

L'idea di fondo è di risolvere ricorsivamente istanze più semplici del problema e combinare queste soluzioni parziali in una soluzione del problema originale.

Se le istanze sono molto semplici, la soluzione è di solito banale.

Un'idea per ordinare un array:

1. ordino la prima metà
2. ordino la seconda metà
3. unisco in qualche modo le due metà



Come unisco (**merge**) due array ordinati?

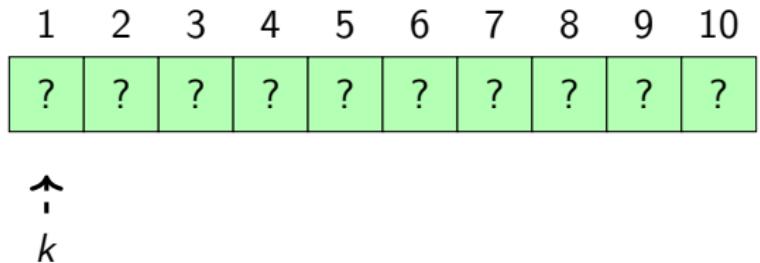
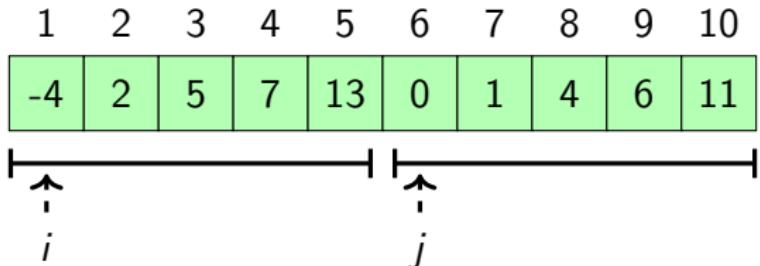
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|----|---|---|---|----|---|---|---|---|----|
| -4 | 2 | 5 | 7 | 13 | 0 | 1 | 4 | 6 | 11 |



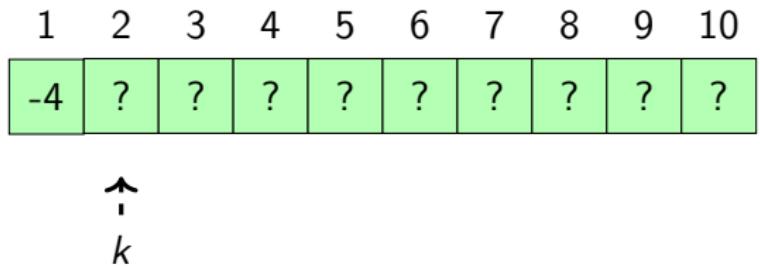
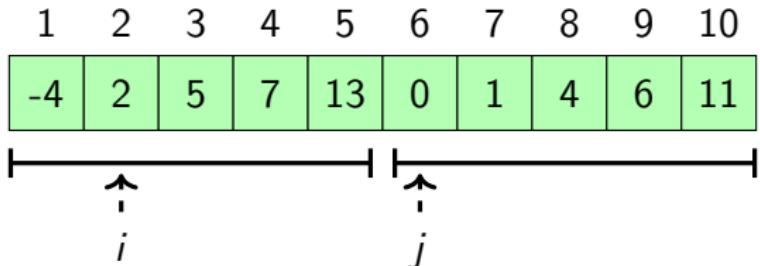
Come unisco (merge) due array ordinati?

| | | | | | | | | | |
|----|---|---|---|----|---|---|---|---|----|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| -4 | 2 | 5 | 7 | 13 | 0 | 1 | 4 | 6 | 11 |

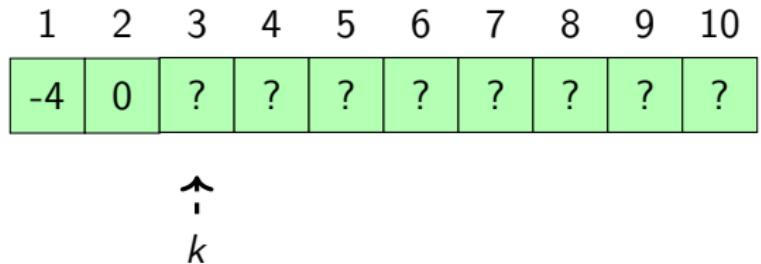
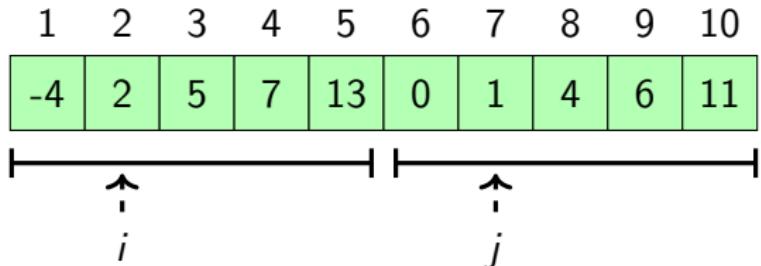
Come unisco (**merge**) due array ordinati?



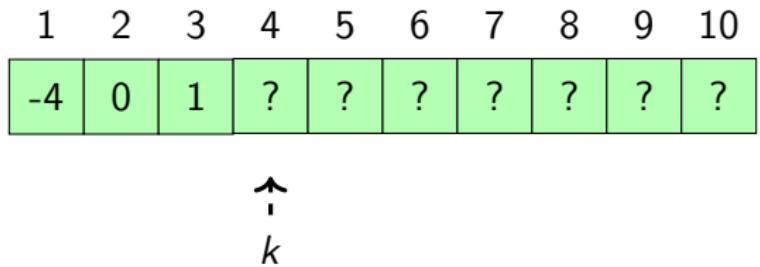
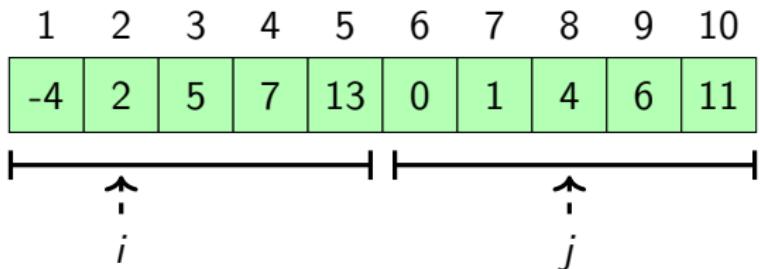
Come unisco (**merge**) due array ordinati?



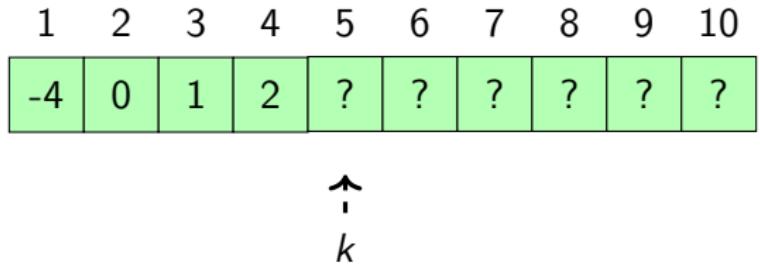
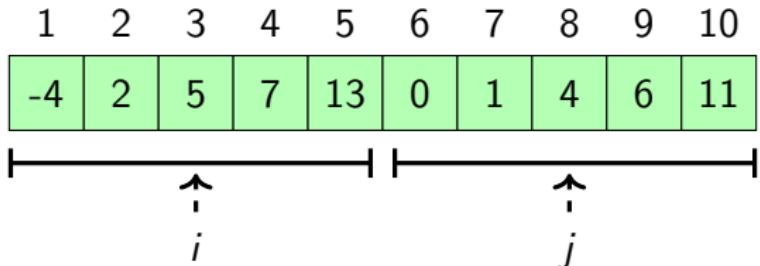
Come unisco (**merge**) due array ordinati?



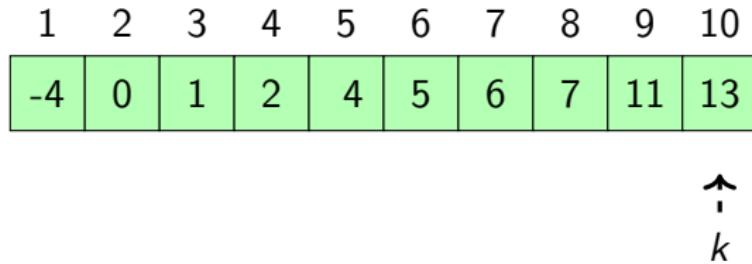
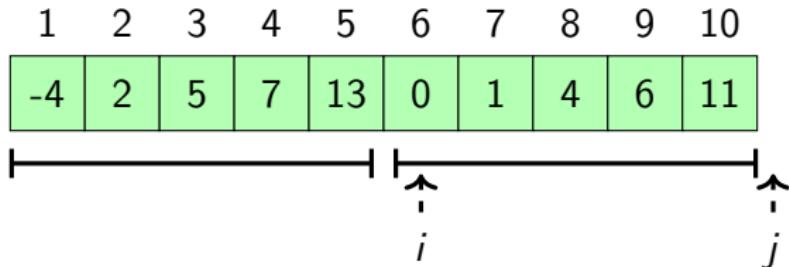
Come unisco (**merge**) due array ordinati?



Come unisco (**merge**) due array ordinati?



Come unisco (merge) due array ordinati?



Merge sort: Pseudo-Code

```
def MERGESORT(A, begin=1, end=|A|):
    if end>begin:
        median ← (begin+end)/2
        MERGESORT(A, begin , median)
        MERGESORT(A, median+1, end )
        MERGE(A, begin , median , end )
enddef
```

Merge: Pseudo-Code

```
def MERGE(A, begin , median , end ):  
    L ← A[begin:median]  
    R ← A[median+1:end]  
  
    i , j ← 1, 1  
  
    for k in range(begin , end ):  
        if (i > len(L) or  
            (j ≤ len(R) and R[j] ≤ L[i])):  
            A[k] ← R[j]  
            j ← j+1  
        else :  
            A[k] ← L[i]  
            i ← i+1  
enddef
```

La Complessità di Merge Sort

Merge prende tempo $\Theta(n)$

Ma mergesort contiene due chiamate ricorsive. Come facciamo a calcolare la complessità in questo caso?

RECUPERO DEI DATI

ALGORITMI DI ORDINAMENTO

ALGORITMI E STRUTTURE DATI

Algoritmi Ricorsivi e Complesità

Algoritmi Ricorsivi

Fino a ora abbiamo visto **algoritmi iterativi**

Possiamo calcolarne la complessità contando il numero di iterazioni.

Possiamo sfruttare la ricorsione scrivendo gli algoritmi?

Come calcolarne la complessità?

Un Algoritmo Ricorsivo . . . Familiare

```
def do_something(A, n=|A|):
    if n>1
        do_something(A, n-1)

    j ← n
    while (j>1 and A[j]<A[j-1]):

        swap(A, j-1, j)
        j←j-1
    endwhile
endif
enddef
```

Un Algoritmo Ricorsivo . . . Familiare

```
def do_something(A, n=|A|):
    if n>1
        do_something(A, n-1)

    j ← n
    while (j>1 and A[j]<A[j-1]):

        swap(A, j-1, j)
        j←j-1
    endwhile
endif
enddef
```

Non è familiare questo codice? È **insertion sort** ricorsivo!

Come Calcolarne la Complessità?

Usando le **equazioni ricorsive di complessità!!!**

Se $T(|A|)$ il tempo per risolvere il problema su A allora

Come Calcolarne la Complessità?

Usando le **equazioni ricorsive di complessità!!!**

Se $T(|A|)$ il tempo per risolvere il problema su A allora

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(n - 1) + \Theta(n) & \text{se } n > 1 \end{cases}$$

Per induzione, $T(|A|) \in \sum_{i=0}^{|A|} \Theta(i) = \Theta(|A|^2)$

Un Esempio Più Spinoso

```
def rec_find(A, v, l=1, r=|A|):
    if r < l
        return 0
    endif

    m ← (l+r)/2
    if A[m]=v
        return m
    endif
    if A[m]>v
        return rec_find(A, v, l, m-1)
    else
        return rec_find(A, v, m+1, r)
    endif
enddef
```

Scriviamo l'Equazione Ricorsiva...

Se $n = 1$ o $A[m] = v$ allora $T(n) \in \Theta(1)$. Altrimenti

$$T(n) \leq T(\lfloor n/2 \rfloor - 1) + \Theta(1)$$

Scriviamo l'Equazione Ricorsiva...

Se $n = 1$ o $A[m] = v$ allora $T(n) \in \Theta(1)$. Altrimenti

$$T(n) \leq T(\lfloor n/2 \rfloor - 1) + \Theta(1) \leq T(n/2) + \Theta(1)$$

Scriviamo l'Equazione Ricorsiva...

Se $n = 1$ o $A[m] = v$ allora $T(n) \in \Theta(1)$. Altrimenti

$$T(n) \leq T(\lfloor n/2 \rfloor - 1) + \Theta(1) \leq T(n/2) + \Theta(1)$$

Poniamo $m \stackrel{\text{def}}{=} \log_2 n$ e $P(m) \stackrel{\text{def}}{=} T(2^m)$

Scriviamo l'Equazione Ricorsiva...

Se $n = 1$ o $A[m] = v$ allora $T(n) \in \Theta(1)$. Altrimenti

$$T(n) \leq T(\lfloor n/2 \rfloor - 1) + \Theta(1) \leq T(n/2) + \Theta(1)$$

Poniamo $m \stackrel{\text{def}}{=} \log_2 n$ e $P(m) \stackrel{\text{def}}{=} T(2^m)$

$$\begin{aligned} P(m) &= T(2^m) \\ &\leq T(2^m/2) + \Theta(1) \\ &= T(2^{m-1}) + \Theta(1) = P(m-1) + \Theta(1) \end{aligned}$$

Quindi $P(m) \leq \sum_{i=0}^m \Theta(1) = \Theta(m)$, $P(m) \in O(m)$ e ...

$$T(|A|) = P(\log_2 |A|) = O(\log |A|)$$

Quando l'Equazione è Più Complicata?

Per esempio:

$$T(i) = \begin{cases} \Theta(1) & \text{se } i = 1 \\ 2 * T(i/2) + \Theta(i) & \text{se } i > 1 \end{cases}$$

Tre metodi:

- ▶ metodo di sostituzione
- ▶ l'albero di ricorsione
- ▶ il teorema dell'esperto

Metodo di Sostituzione

Il Metodo di Sostituzione

Due passi:

1. ipotizzo una complessità per $T(n)$
2. dimostro per induzione che esistono delle costanti che soddisfano l'ipotesi

Il Metodo di Sostituzione: Un Esempio

Es. $T(n) = 2 * T(n/2) + O(n)$

1. ipotizzo che $T(n) \in O(n \log n)$
2. assumo che $\exists c \ \forall m < n \ T(m) \leq c * m * \log_2 m$

Il Metodo di Sostituzione: Un Esempio

Es. $T(n) = 2 * T(n/2) + O(n)$

1. ipotizzo che $T(n) \in O(n \log n)$
2. assumo che $\exists c \forall m < n T(m) \leq c * m * \log_2 m$

$$\begin{aligned}T(n) &\leq 2 * T(n/2) + c' * n \\&\leq 2 * (c * n/2 * \log_2(n/2)) + c' * n \\&= c * n * \log_2(n/2) + c' * n \\&= c * n * \log_2(n) - c * n * \log_2(2) + c' * n\end{aligned}$$

Se scelgo $c' < c$, allora $T(n) \leq c * n * \log_2(n)$.

La costante c è la **stessa** per ogni n

Il Metodo di Sostituzione: Come NON fare

Esempio: $T(n) = 3 * T(n/2) + O(n)$

1. ipotizzo che $T(n) \in O(n * \log n)$
2. assumo che $\exists c \forall m < n T(m) \leq c * m * \log_2 m$

$$\begin{aligned} &\leq 3 * (c * n/2 * \log_2(n/2)) + c' * n \\ &= \frac{3}{2} * c * n * \log_2(n/2) + c' * n \\ &= \frac{3}{2} * c * n * \log_2(n) - c * n * \log_2(2) + c' * n \\ &\leq c * \frac{3}{2} * n * \log_2(n) \end{aligned}$$

Il Metodo di Sostituzione: Come NON fare

Es. $T(n) = 3 * T(n/2) + O(n)$

1. ipotizzo che $T(n) \in O(n * \log n)$
2. assumo che $\exists c \forall m < n T(m) \leq c * m * \log_2 m$

$$\begin{aligned} &\leq 3 * (c * n/2 * \log_2(n/2)) + c' * n \\ &= \frac{3}{2} * c * n * \log_2(n/2) + c' * n \\ &= \frac{3}{2} * c * n * \log_2(n) - c * n * \log_2(2) + c' * n \\ &\leq c * \frac{3}{2} * n * \log_2(n) \end{aligned}$$

La costante per $m < n$ era c , mentre per n è $\frac{3}{2} * c$

Albero di Ricorsione

Albero di Ricorsione

Costruisci un albero (grafo connesso aciclico) in cui ogni nodo:

- ▶ rappresenta una chiamata ricorsiva
- ▶ corrisponde al costo della chiamata senza i passi ricorsivi
- ▶ è collegato alle chiamate ricorsive che genera

Sommando il costo di tutti i nodi, otteniamo il costo totale.

Albero di Ricorsione: Un Esempio

Consideriamo l'equazione

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 8 * T(n/2) + \Theta(n^2) & \text{se } m > 1 \end{cases}$$

Albero di Ricorsione: Un Esempio

Consideriamo l'equazione

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 8 * T(n/2) + \Theta(n^2) & \text{se } m > 1 \end{cases}$$

Scegliamo una funzione in $\Theta(n^2)$ (es. $c * n^2$)

Albero di Ricorsione: Un Esempio

Consideriamo l'equazione

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 8 * T(n/2) + \Theta(n^2) & \text{se } m > 1 \end{cases}$$

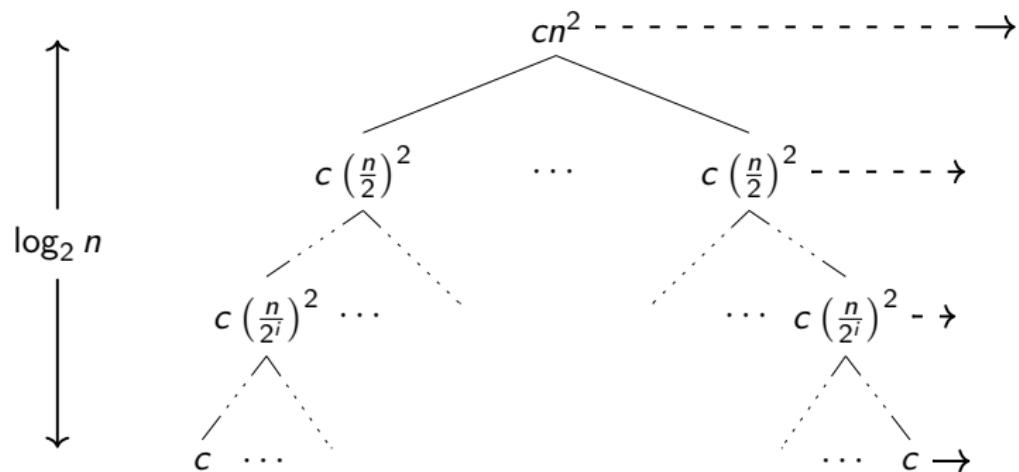
Scegliamo una funzione in $\Theta(n^2)$ (es. $c * n^2$)

Se m è la dimensione dell'input per una chiamata:

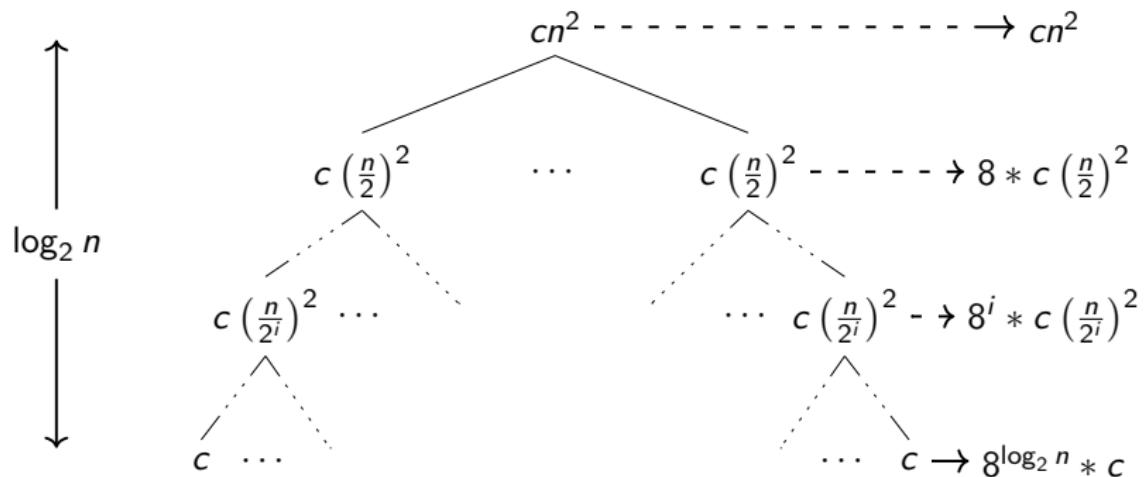
- ▶ il costo su una chiamata è cm^2
- ▶ ogni nodo dell'albero ha 8 figli (le chiamate ricorsive)
- ▶ m si dimezza a ogni chiamata ricorsiva

Albero di Ricorsione: Un Esempio (Cont'd)

Albero di Ricorsione: Un Esempio (Cont'd)

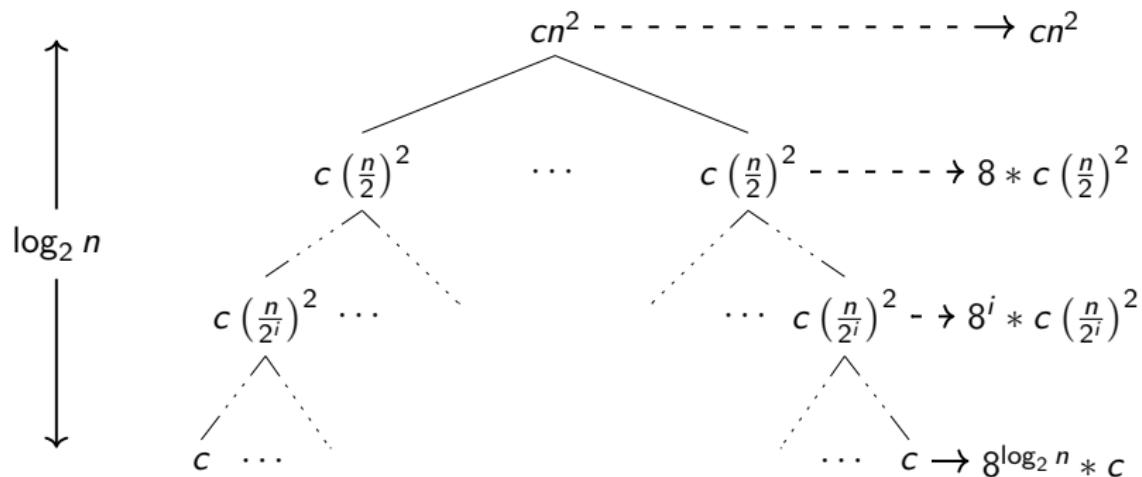


Albero di Ricorsione: Un Esempio (Cont'd)



$$T_M(n) = cn^2 \left(1 + 2 + \dots + 2^i + \dots + 2^{\log_2 n} \right)$$

Albero di Ricorsione: Un Esempio (Cont'd)



$$\begin{aligned}T_M(n) &= cn^2 \left(1 + 2 + \dots + 2^i + \dots + 2^{\log_2 n} \right) \\&= cn^2 \left(2^{1+\log_2 n} - 1 \right) = cn^2 (2n - 1) \in \Theta(n^3)\end{aligned}$$

Il Metodo dell'Esperto

Metodo dell'Esperto

Theorem (Teorema dell'Esperto)

Siano $a \geq 1$ e $b > 1$ due costanti, $f(n)$ una funzione ≥ 0 e

$$T(n) \stackrel{\text{def}}{=} \begin{cases} \Theta(1) & \text{se } n = 1 \\ a * T(n/b) + f(n) & \text{altrimenti} \end{cases}$$

Se, per qualche costante ϵ , $f(n)$ sta in

1. $O(n^{\log_b a - \epsilon})$, allora $T(n) \in \Theta(n^{\log_b a})$;
2. $\Theta(n^{\log_b a})$, allora $T(n) \in \Theta(n^{\log_b a} * \log n)$;
3. $\Omega(n^{\log_b a + \epsilon})$ e $a * f(n/b) \leq c * f(n)$ per qualche $c < 1$ e n sufficientemente grande, allora $T(n) \in \Theta(f(n))$.

Applicazioni

Consideriamo:

$T(n) = 9 * T(n/3) + O(n)$: $a = 9$, $b = 3$ e $f(n) \in O(n^{\log_3 9 - \epsilon})$)
con $\epsilon = 1$ quindi Caso 1 $\Rightarrow T(n) \in \Theta(n^2)$

Applicazioni

Consideriamo:

$$T(n) = 9 * T(n/3) + O(n): a = 9, b = 3 \text{ e } f(n) \in O(n^{\log_3 9 - \epsilon})$$

con $\epsilon = 1$ quindi Caso 1 $\Rightarrow T(n) \in \Theta(n^2)$

$$T(n) = T(2 * n/3) + 1: a = 1, b = 3/2 \text{ e}$$
$$f(n) \in \Theta(n^{\log_{3/2} 1}) = \Theta(1) \text{ quindi Caso 2} \Rightarrow T(n) \in \Theta(\log n)$$

Applicazioni

Consideriamo:

$$T(n) = 9 * T(n/3) + O(n): a = 9, b = 3 \text{ e } f(n) \in O(n^{\log_3 9 - \epsilon})$$

con $\epsilon = 1$ quindi Caso 1 $\Rightarrow T(n) \in \Theta(n^2)$

$$T(n) = T(2 * n/3) + 1: a = 1, b = 3/2 \text{ e}$$
$$f(n) \in \Theta(n^{\log_{3/2} 1}) = \Theta(1) \text{ quindi Caso 2} \Rightarrow T(n) \in \Theta(\log n)$$

$$T(n) = 3 * T(n/4) + n * \log n: a = 3, b = 4 \text{ e } f(n) \in \Omega(n^{\log_4 3 + \epsilon})$$

con $\epsilon \approx 0.2$ quindi Caso 3 $\Rightarrow T(n) \in \Theta(n * \log n)$

Applicazioni

Consideriamo:

$$T(n) = 2 * T(n/2) + n * \log n: a = 2, b = 2 \text{ e } f(n) \in \Omega(n^{\log_2 2}).$$

Sfortunatamente,

$$\lim_{n \rightarrow \infty} \frac{n * \log n}{n^{1+\epsilon}} = 0$$

per ogni $\epsilon > 0$ e **NON** si può applicare il teorema dell'esperto.

Dimostrazione

Lemma

Siano $a \geq 1$ e $b > 1$ due costanti, $f(n)$ una funzione ≥ 0 e

$$T(n) \stackrel{\text{def}}{=} \begin{cases} \Theta(1) & \text{se } n = 1 \\ a * T(n/b) + f(n) & \text{se } n = b^i \end{cases}$$

dove i è un naturale positivo. Allora

$$T(n) \in \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

Dim. tramite l'albero di ricorsione.

Dimostrazione (Cont'd)

Lemma

Siano $a \geq 1$ e $b > 1$ due costanti, $f(n)$ una funzione ≥ 0 e:

$$g(n) \stackrel{\text{def}}{=} \sum_{j=0}^{\log_b n - 1} a^j f(n/b^j)$$

1. se $f(n) \in O(n^{\log_b a - \epsilon})$, allora $g(n) \in O(n^{\log_b a})$;
2. se $f(n) \in \Theta(n^{\log_b a})$, allora $g(n) \in \Theta(n^{\log_b a} \log n)$;
3. se $a * f(n/b) \leq cf(n)$ per qualche $c < 1$ e n sufficientemente grande, allora $g(n) \in \Theta(f(n))$.

Dim. tramite sostituzione delle condizioni

ORDINAMENTO
HEAPSORT

ALGORITMI E STRUTTURE DATI

HEAPSORT: IDEA DI BASE

selection_sort(A):

```
for i in len(a)...2:  
    j = find_argmax(A,1,i)  
    swap(A,i,j)
```

Qual è il bottleneck computazionale di selection_sort?

Idea: uso una struttura dati che permetta di estrarre il massimo in tempo sublineare e la cui costruzione costi poco.



Una struttura dati in cui:

- Possiamo inserire elementi
- Possiamo estrarre il massimo tra tutti gli elementi inseriti

HEAPSORT: IDEA DI BASE

- ▶ Dobbiamo eseguire n inserimenti nella coda di priorità
- ▶ Dobbiamo eseguire n rimozioni dalla coda di priorità
- ▶ Quindi la complessità computazione dipende da:
 - ▶ Costo degli inserimenti
 - ▶ Costo delle rimozioni del massimo

PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

Un primo approccio è quello di creare una coda di priorità usando un array di n elementi e una variabile che ci dice il prossimo posto libero nell'array:



`indice_libero = 0`

L'inserimento aggiunge solo l'elemento nella prima posizione libera

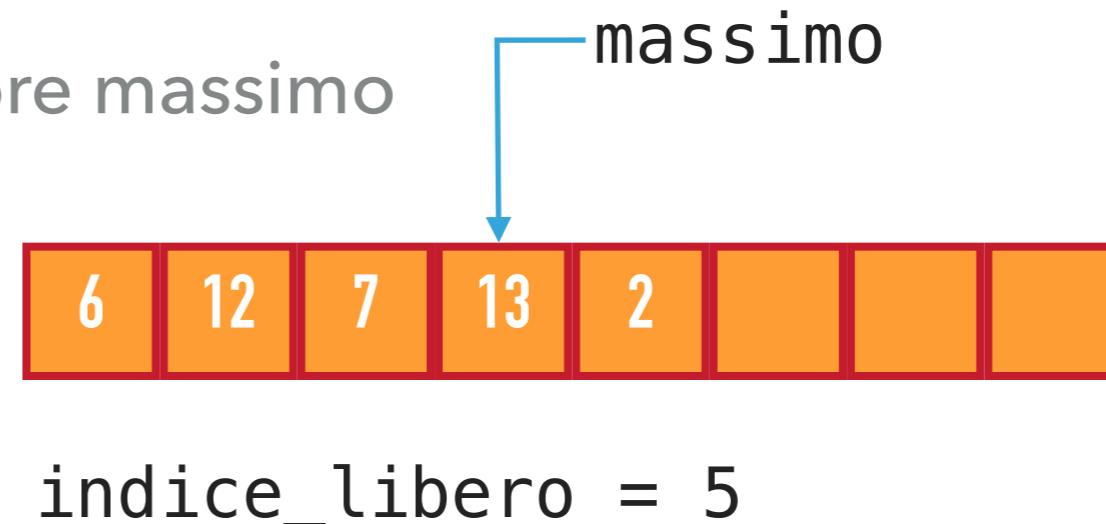


`indice_libero = 1`

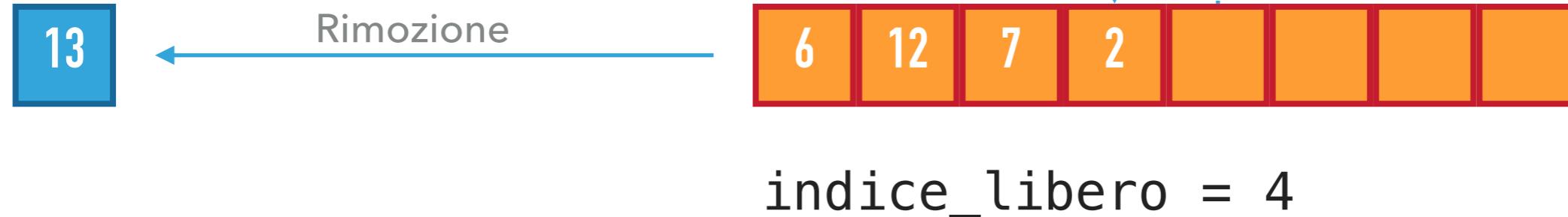
PRIMO APPROCCIO PER CREARE UNA CODA DI PRIORITÀ

La rimozione consiste in due fasi

1. Individuare l'elemento col valore massimo



2. Rimuovere l'elemento di valore massimo e rimpiazzarlo con l'ultimo elemento presente nell'array



ANALISI DELL'APPROCCIO NAIVE

- ▶ L'inserimento richiede di copiare un valore e modificare una variabile, questo richiede tempo costante: $O(1)$
- ▶ La rimozione richiede di trovare il massimo in un array di n elementi e nel caso peggiore richiede un numero lineare di passi: $O(n)$
- ▶ Per ordinare, ognuna di queste operazioni viene eseguita un numero lineare di volte, ottenendo quindi un costo totale che è *quadratico*: $O(n^2)$

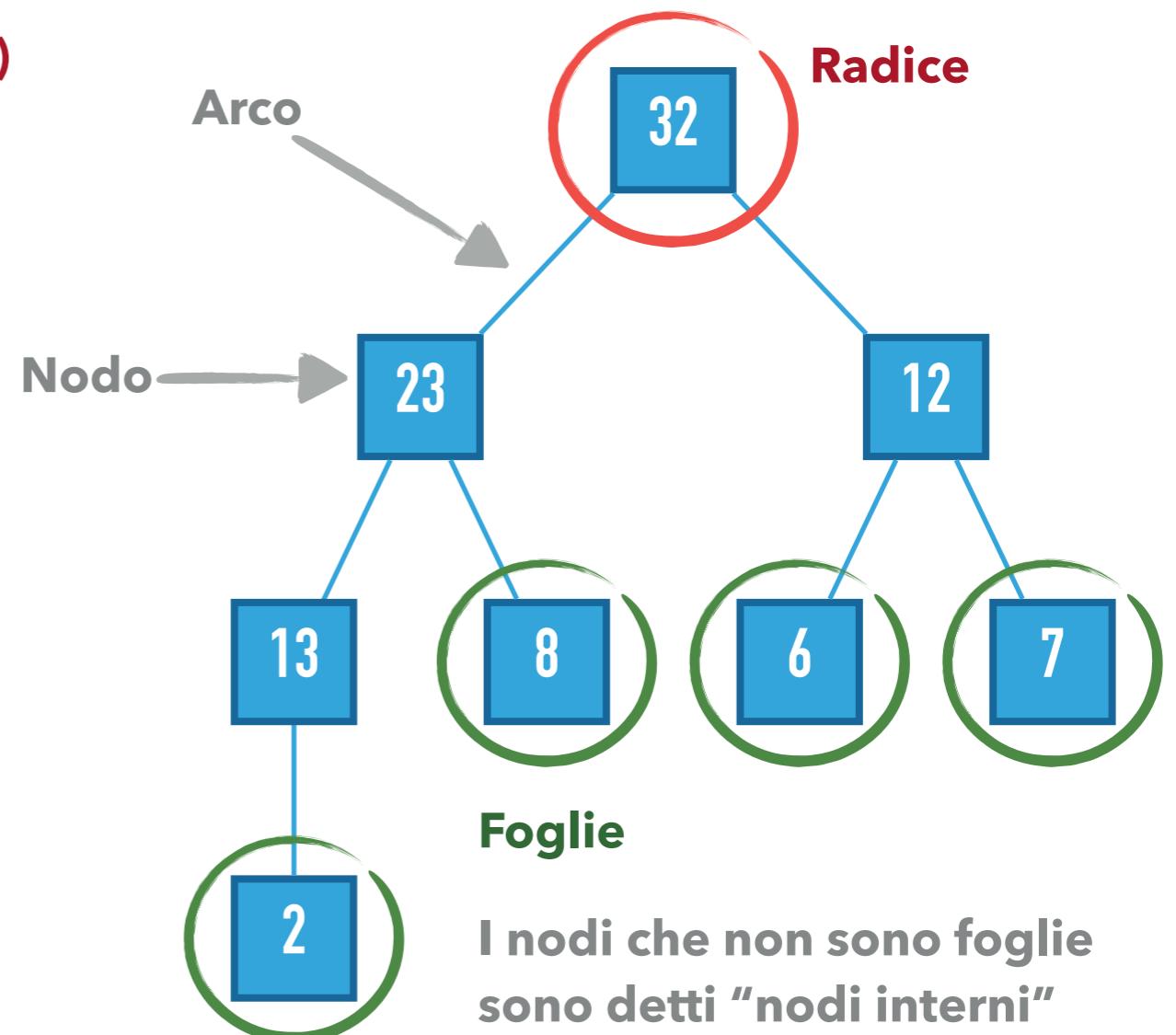
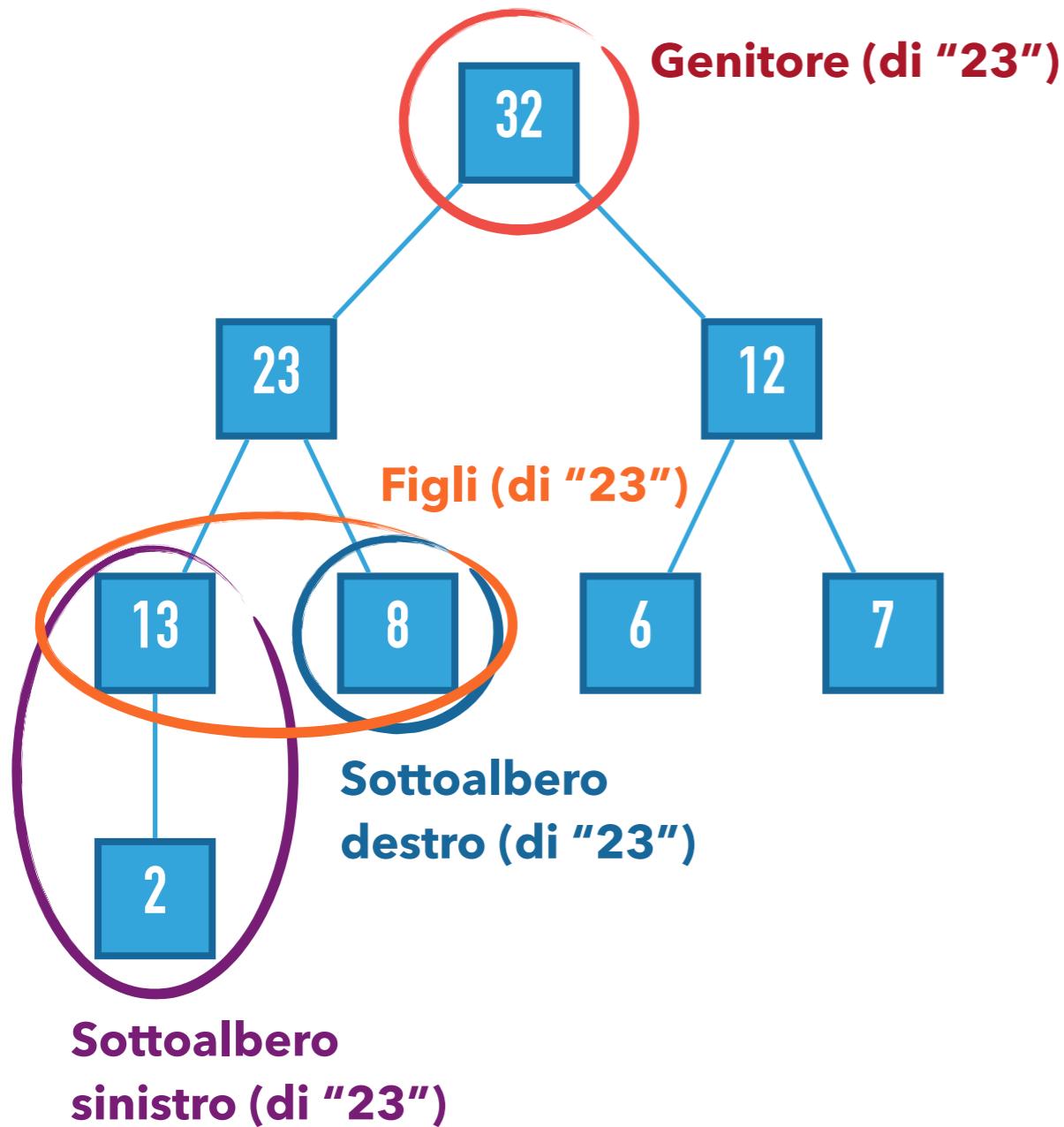
ANALISI DELL'APPROCCIO NAIVE

- ▶ Il costo di questo approccio è lo stesso di insertion sort, selection sort e bubble sort.
- ▶ Possiamo migliorare la situazione cambiando il modo di rappresentare la coda di priorità?
- ▶ Vediamo la struttura del *max-heap*, che consente di effettuare inserimenti e rimozioni in tempo $O(\log n)$

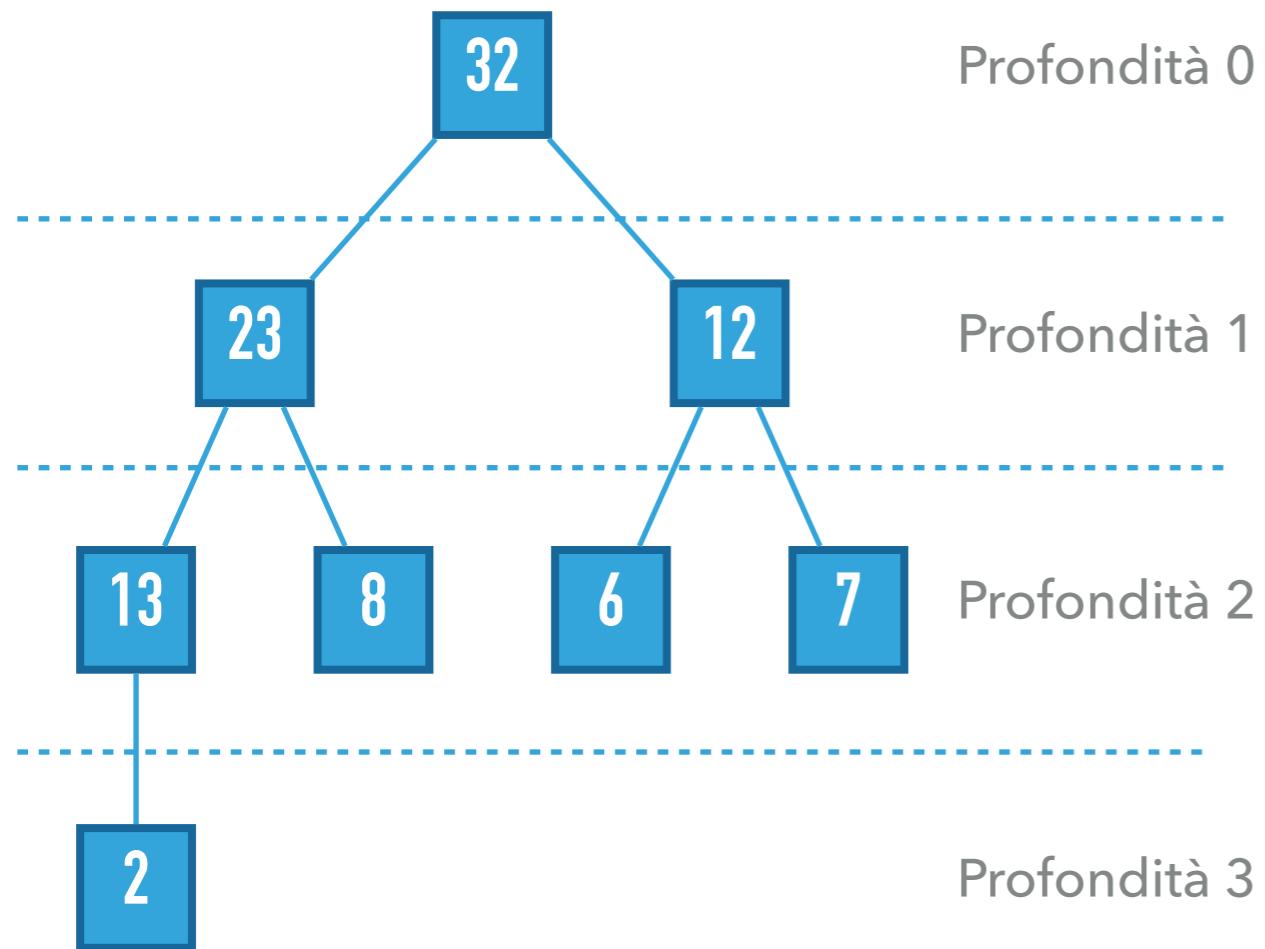
ALBERI BINARI

- ▶ Un **albero binario** è una struttura su un insieme finito di nodi tale per cui:
 - ▶ Non contiene nodi (albero vuoto), oppure
 - ▶ È una tripla composta da:
 - ▶ Un **nodo radice**
 - ▶ Un albero binario chiamato **sottoalbero sinistro**
 - ▶ Un albero binario chiamato **sottoalbero destro**

ALBERI BINARI



ALBERI BINARI



La **profondità** del nodo i è il numero di archi da percorrere a partire dalla radice per raggiungere il nodo i .

L'**altezza** di un albero è il massimo della profondità dei suoi nodi

Un albero di altezza h ha al più $2^{h+1} - 1$ nodi: let's prove it!

Un albero binario **quasi completo** ha tutti i livelli pieni eccetto l'ultimo, pieno da sx a dx.

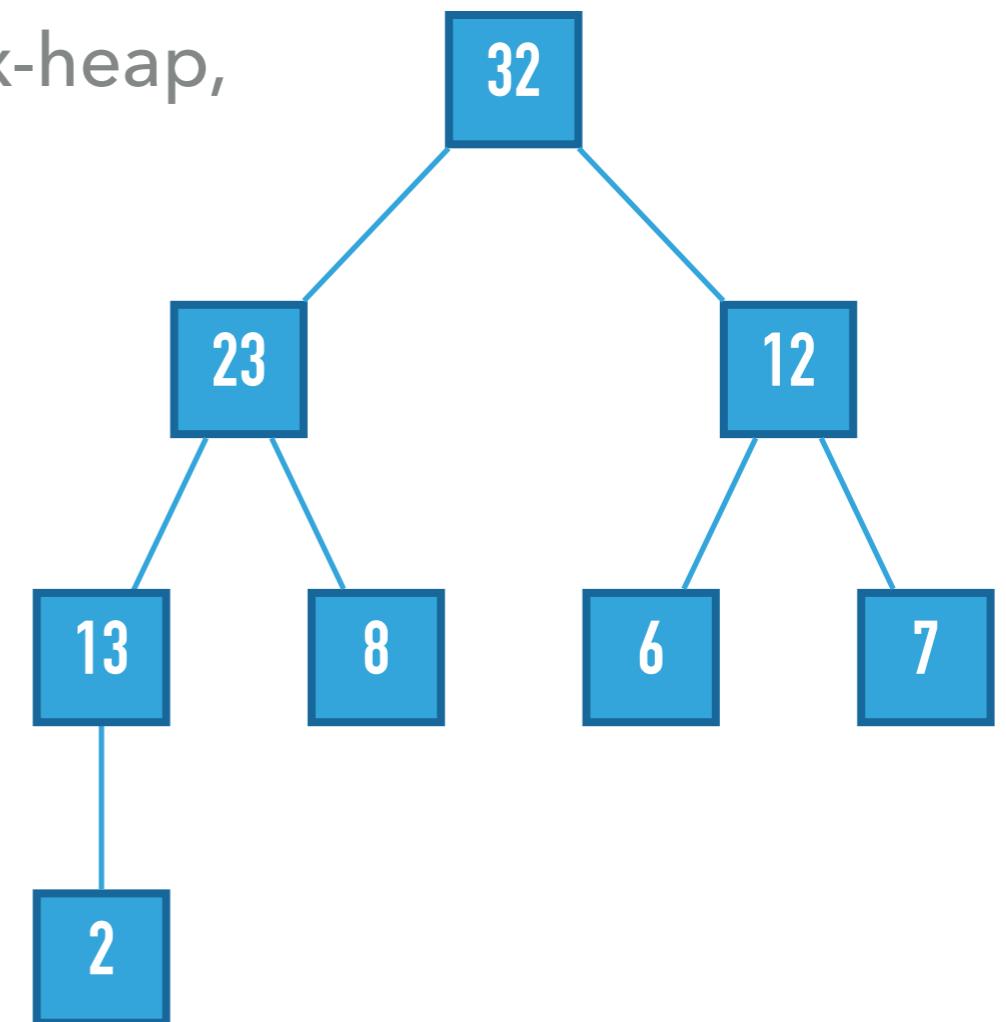
Un albero binario completo con n nodi ha $\left\lceil \frac{n}{2} \right\rceil$ foglie.

MAX-HEAP

Questo albero binario rappresenta un max-heap,
ovvero rispetta la proprietà di max-heap

Definizione (proprietà di max-heap)

Un albero binario ha la proprietà di *max-heap* (resp., *min-heap*) se per ogni nodo diverso dalla radice, il valore in esso contenuto è minore o uguale (resp. maggiore o uguale) al valore contenuto nel nodo genitore.



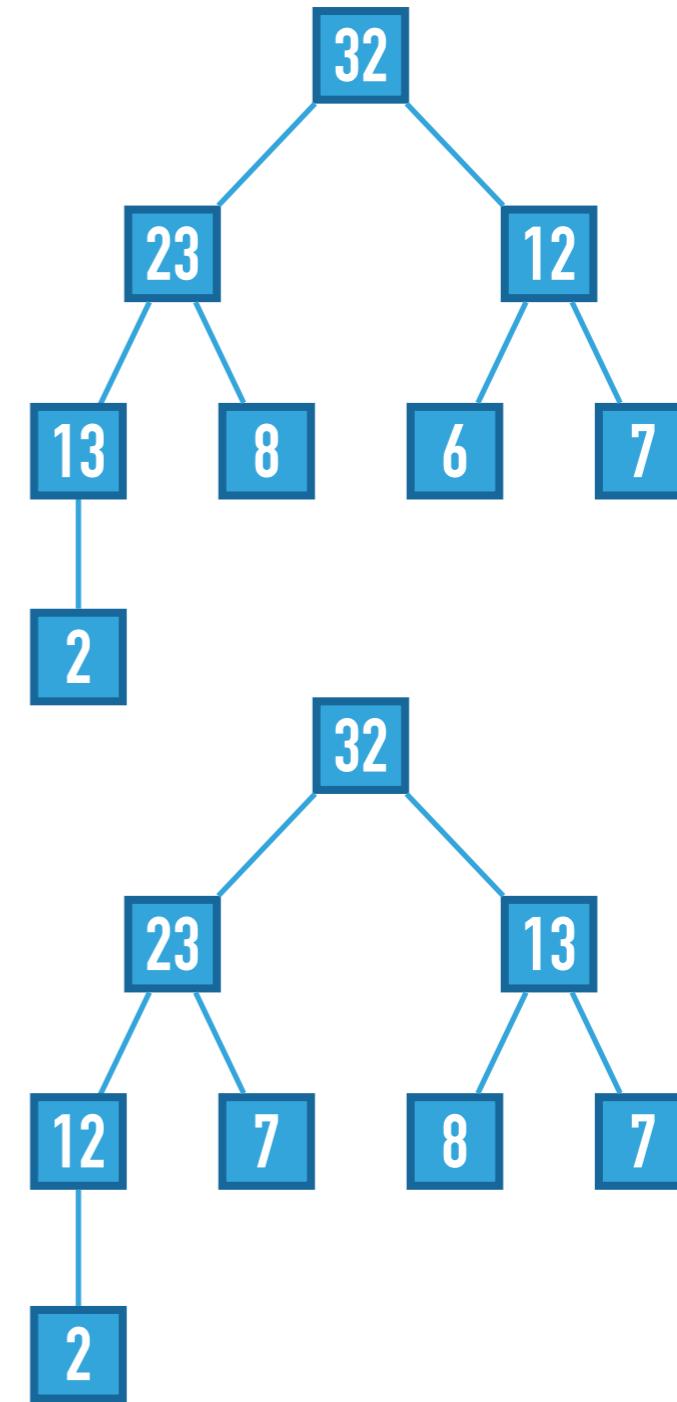
$$\text{key}(x) \geq \text{key}(\text{left}(x)) \text{ e } \text{key}(x) \geq \text{key}(\text{right}(x))$$

$$x.\text{key} \geq x.\text{left}.\text{key} \text{ e } x.\text{key} \geq x.\text{right}.\text{key}$$

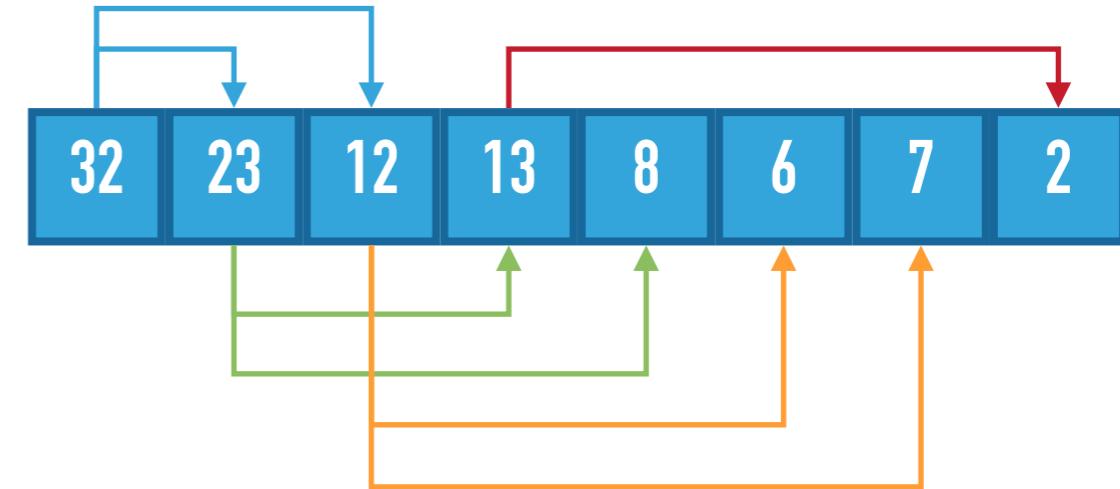
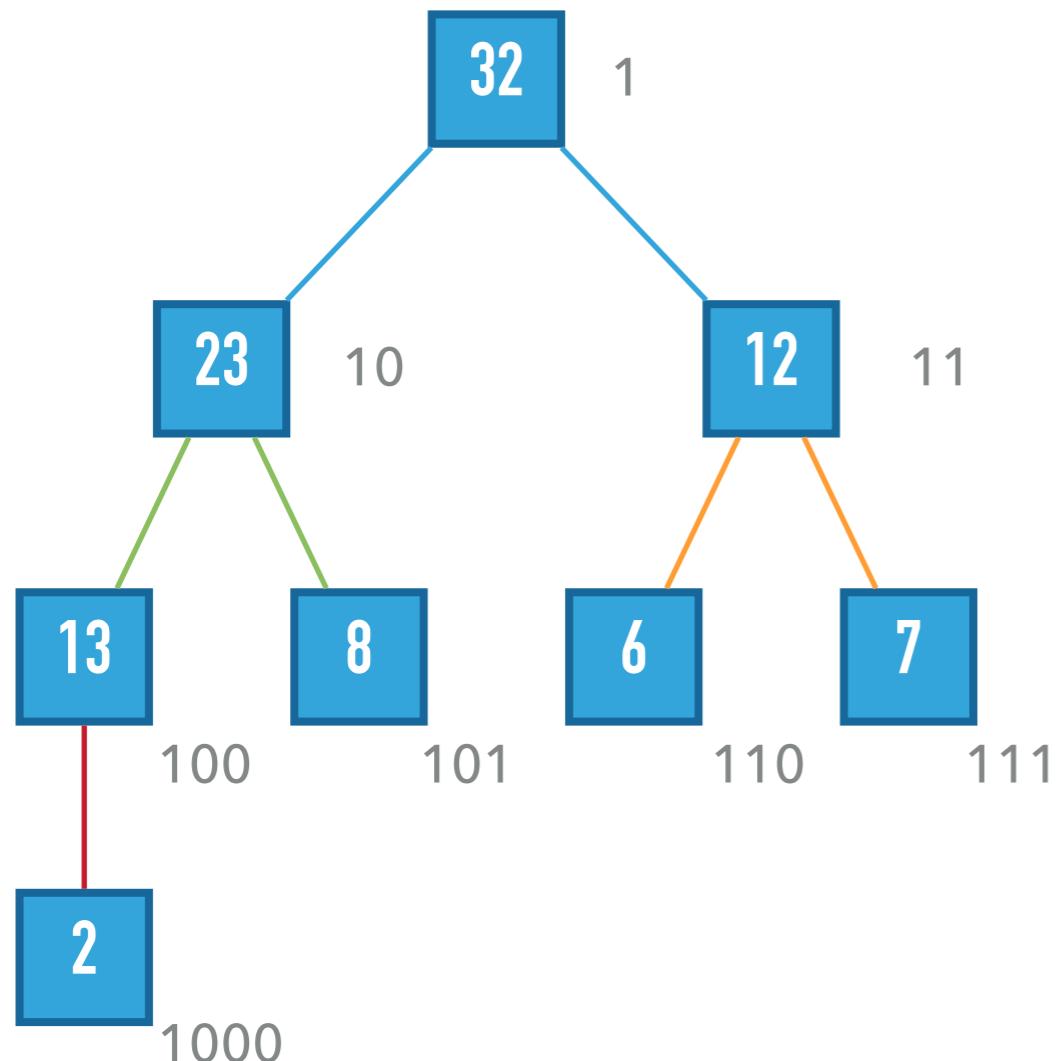
CONSEGUENZE DELLA PROPRIETÀ DI MAX-HEAP

- ▶ Il valore massimo del max-heap si trova sempre nella radice
- ▶ Per lo stesso insieme di valori possono esistere più alberi che rispettano la proprietà di max-heap

MA COME POSSO RAPPRESENTARE UN MAX-HEAP NEL CODICE?



ALBERI BINARI COME ARRAY



Usiamo un array per rappresentare un max-heap. Gli indici ci indicano chi sono i figli e il genitore di un nodo.

heap-size(A) ($\leq \text{length}(A)$)

Dato il nodo in posizione i dell'array:

$$\text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor$$

Genitore

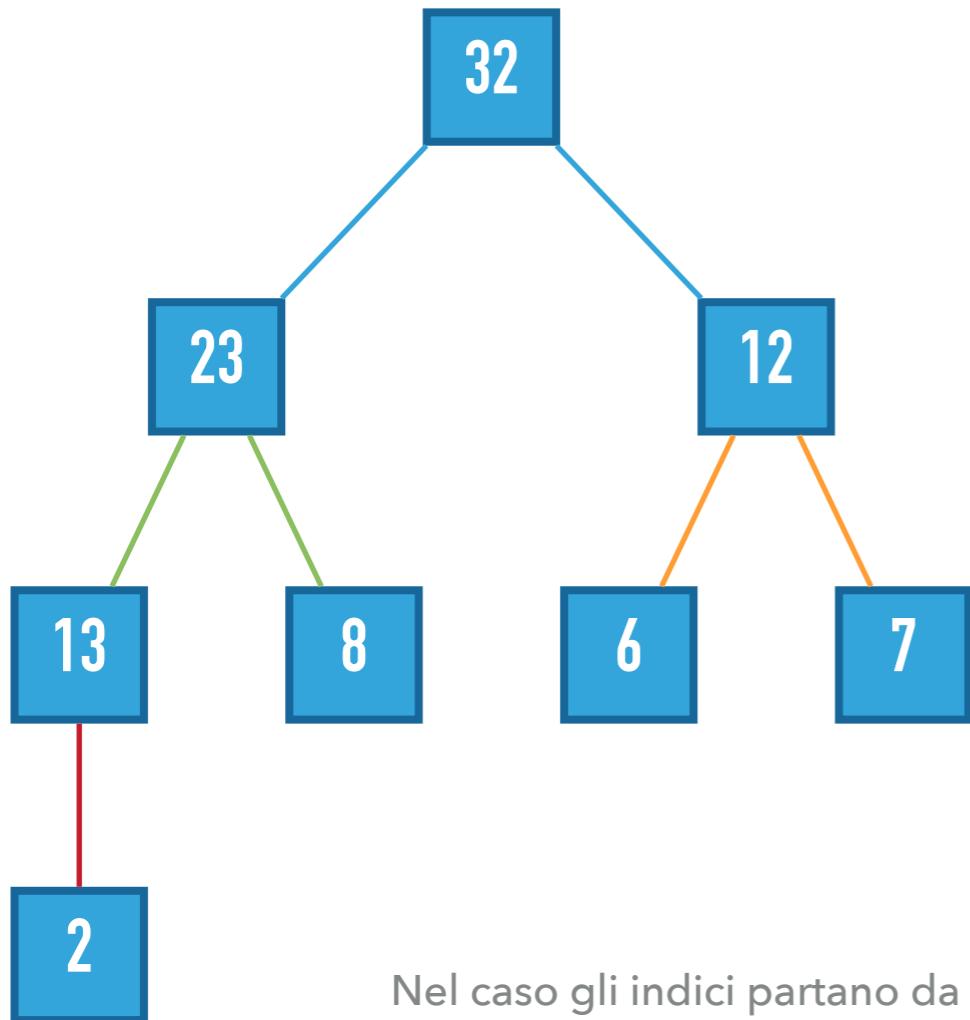
$$\text{left}(i) = 2i$$

Figlio sinistro

$$\text{right}(i) = 2i + 1$$

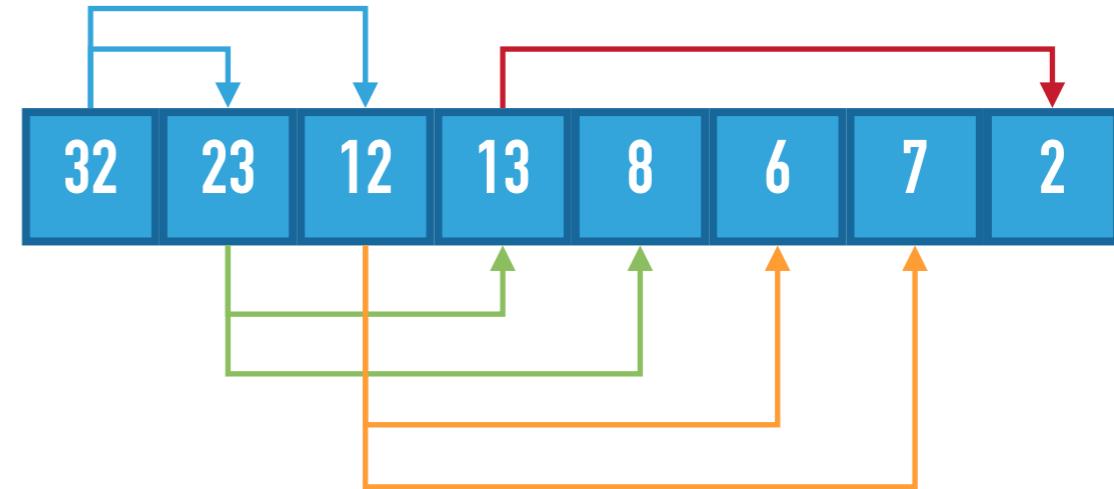
Figlio destro

ALBERI BINARI COME ARRAY



Nel caso gli indici partano da zero

Dato il nodo in posizione i dell'array:



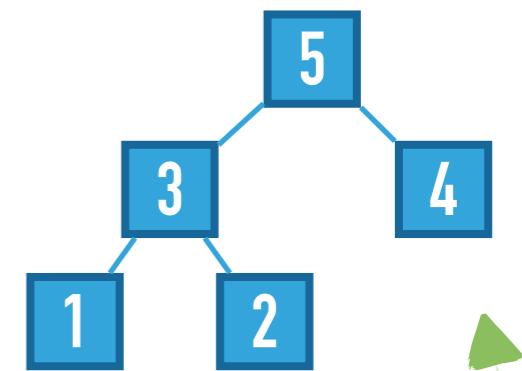
Usiamo un array per rappresentare un max-heap. Gli indici ci indicano chi sono i figli e il genitore di un nodo

$$\text{parent}(i) = \left\lfloor \frac{i-1}{2} \right\rfloor \quad \text{Genitore}$$

$$\text{left}(i) = 2i + 1 \quad \text{Figlio sinistro}$$

$$\text{right}(i) = 2i + 2 \quad \text{Figlio destro}$$

ARRAY COME MAX-HEAP: QUIZ



Quale dei seguenti array rappresenta un max-heap?

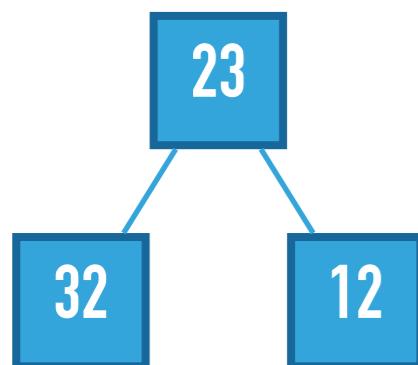


PROCEDURE DA DEFINIRE

- ▶ Inserimento nello heap
- ▶ Rimozione del massimo dall'heap
- ▶ Come procedure di support useremo:
 - ▶ Build-max-heap: per costruire un max-heap dato l'array da ordinare
 - ▶ Max-heapify: per ripristinare la proprietà di max-heap quando la radice non rispetta la proprietà di max-heap (ma tutti gli altri nodi la rispettano)

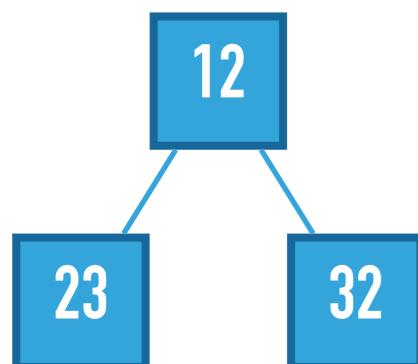
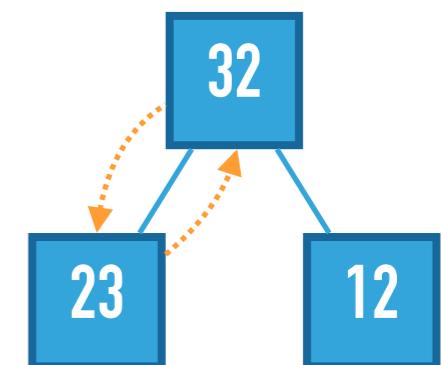
MAX-HEAPIFY

Vediamo su un albero di dimensione ridotta in che modi possiamo infrangere la proprietà di max-heap e come ripristinarla



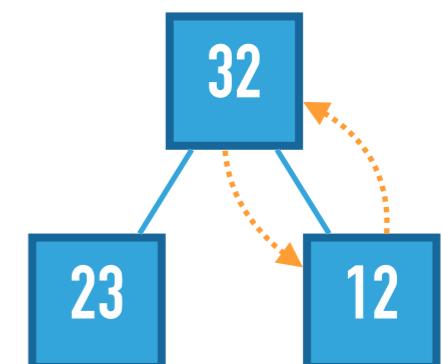
Il valore massimo è nel figlio di sinistra

Scambiamo il valore del figlio di sinistra
con quello contenuto nella radice



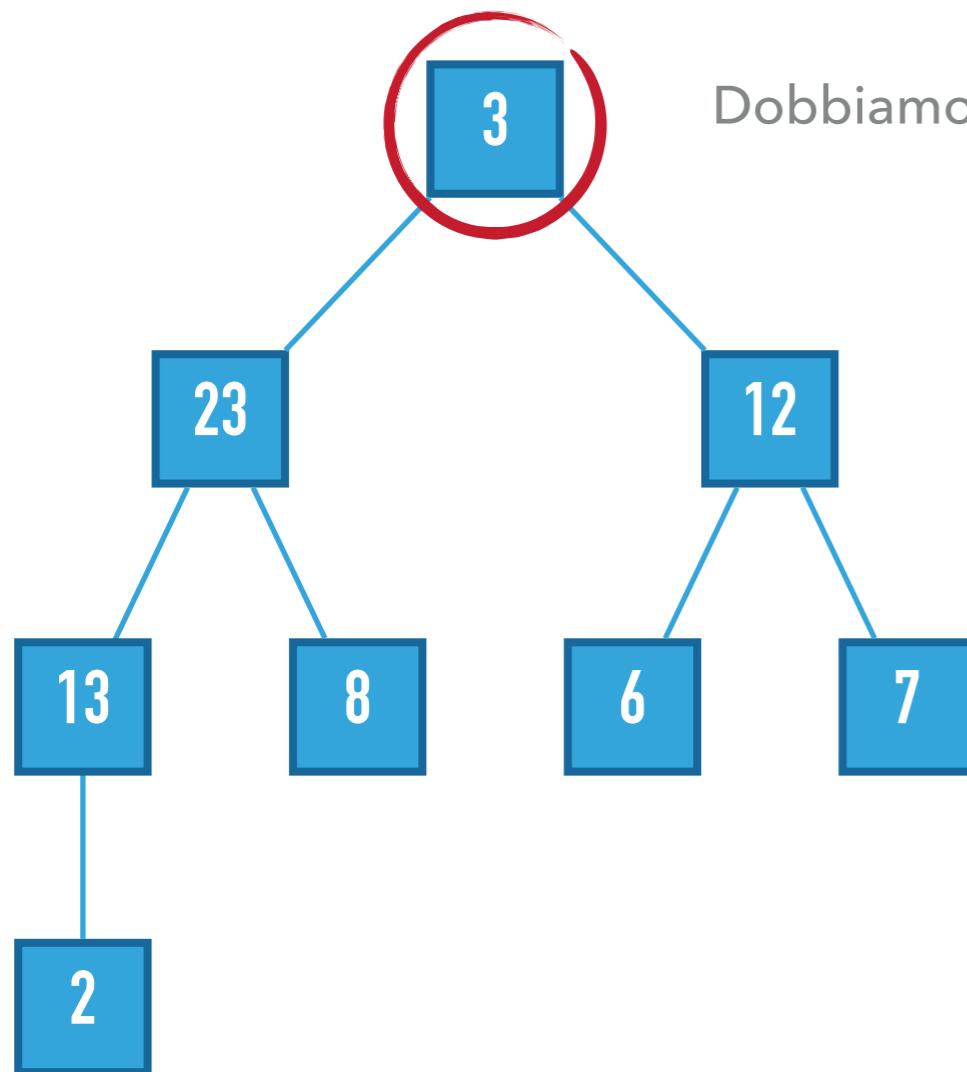
Il valore massimo è nel figlio di destra

Scambiamo il valore del figlio di destra
con quello contenuto nella radice



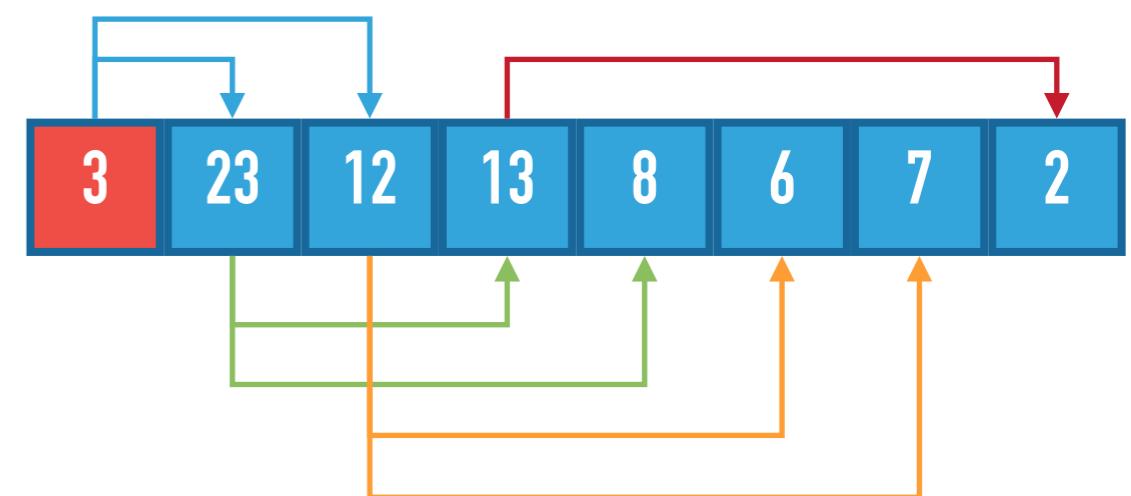
MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



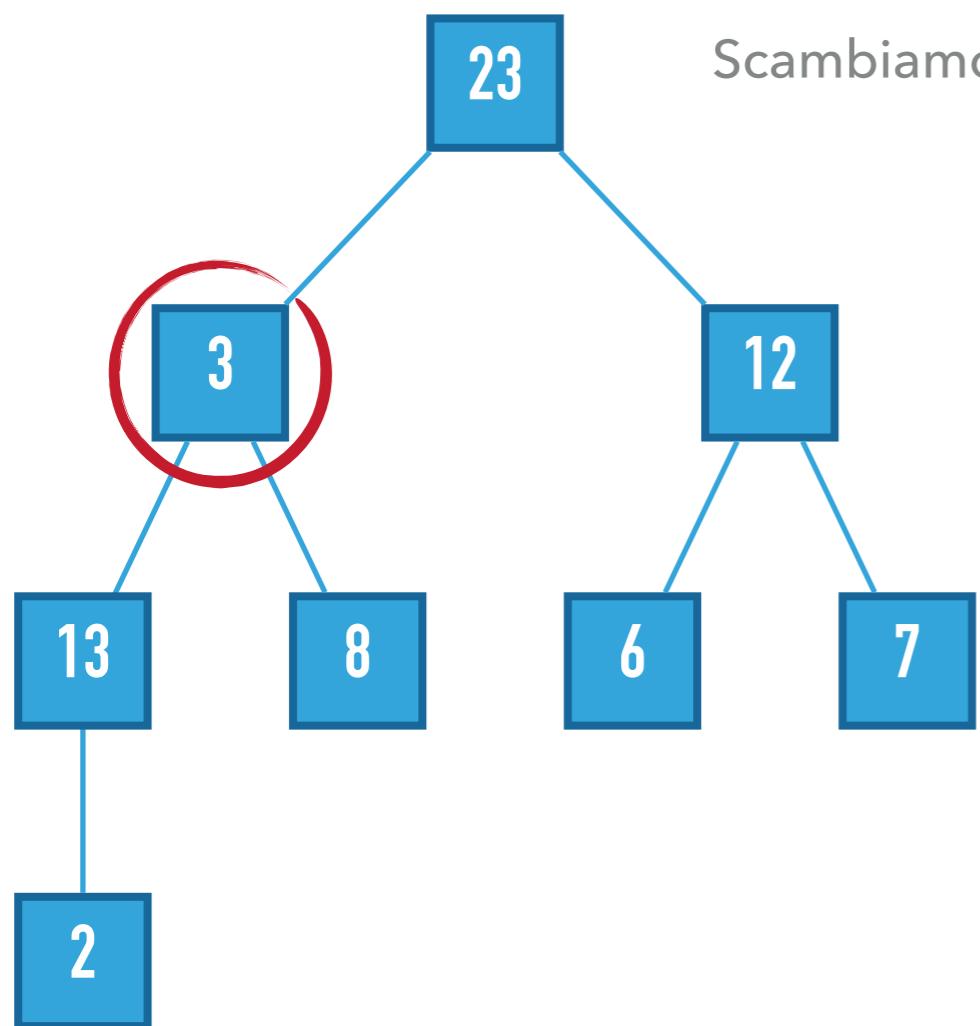
Dobbiamo posizionare "3" correttamente

Visualizzazione della rappresentazione come array del max-heap:



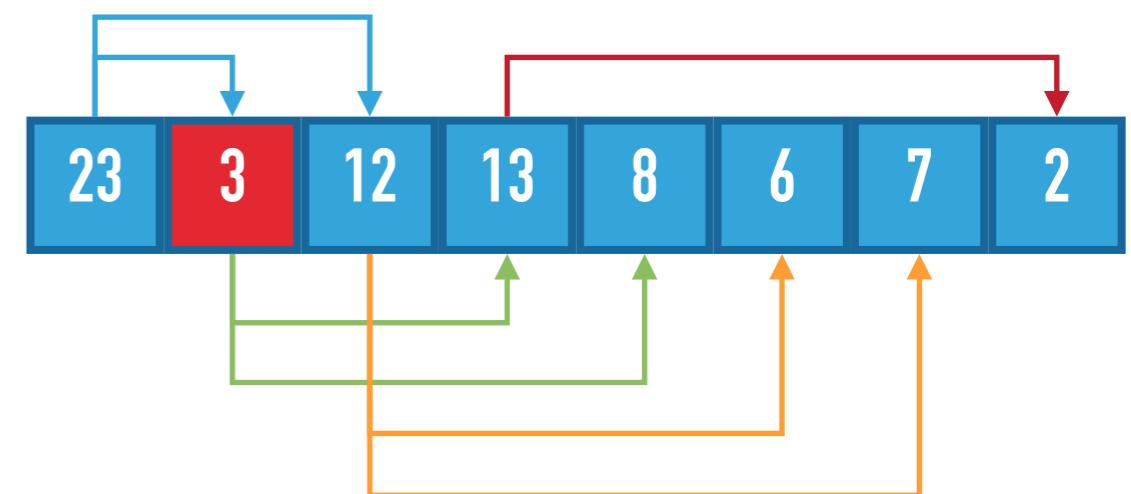
MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



Scambiamo con "23" perché è il maggiore

Visualizzazione della rappresentazione come array del max-heap:

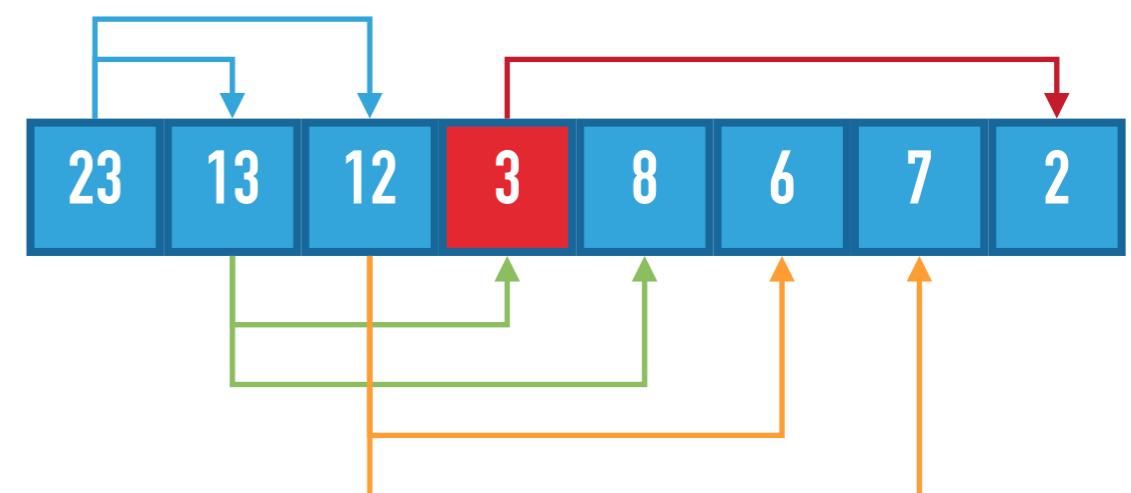


MAX-HEAPIFY

Per alberi di dimensione maggiore basta ripetere la procedura ricorsivamente finché la proprietà non è rispettata



Visualizzazione della rappresentazione come array del max-heap:



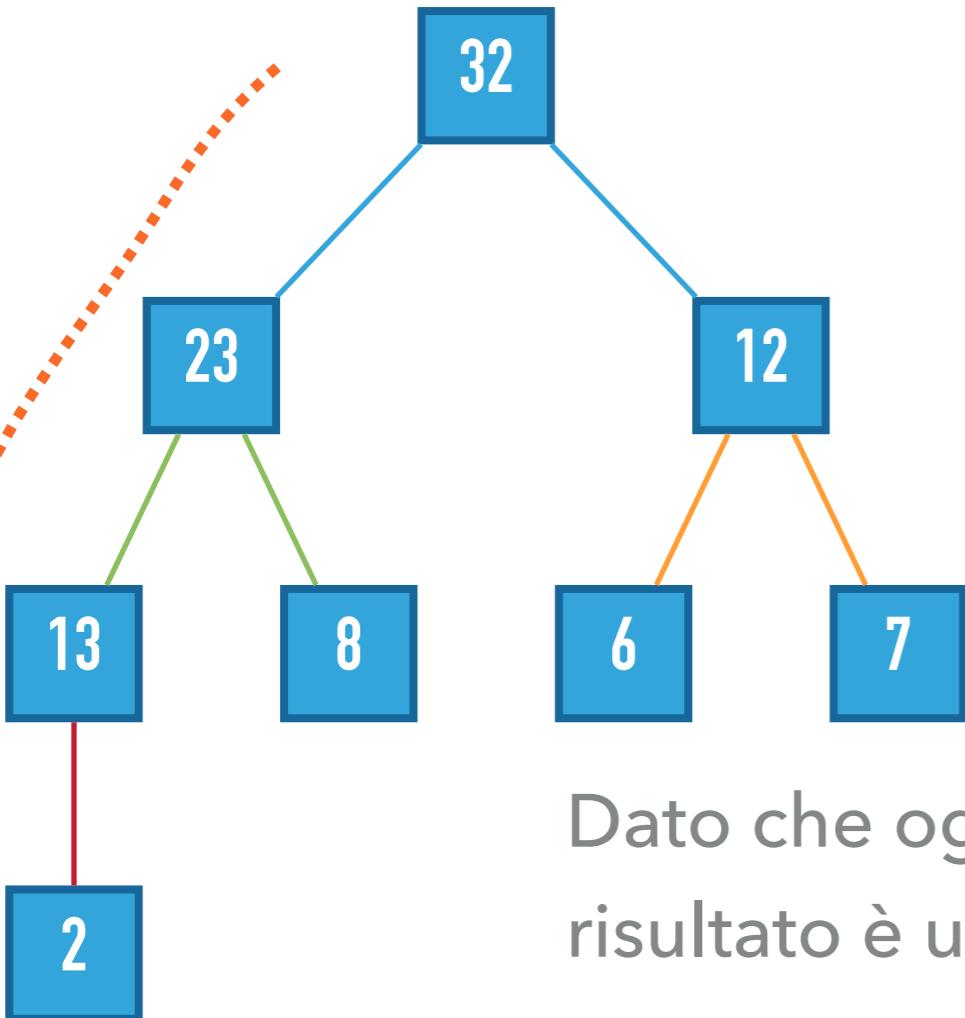
MAX-HEAPIFY: PSEUDOCODICE

- ▶ Parametri: A (array), i (posizione)
- ▶ L = left(i)
- ▶ R = right(i)
- ▶ largest = i # *inizialmente “i” è il nostro candidato*
- ▶ if L <heap-size(A) and A[L] > a[largest]
 - ▶ largest = L # *se il figlio sinistro è maggiore*
- ▶ if R <heap-size(A) and A[R] > a[largest]
 - ▶ largest = R # *se il figlio destro è maggiore*
- ▶ if largest ≠ i
 - ▶ swap(A[i],A[largest])# *scambiamo “i” e “largest”*
 - ▶ max-heapify(A, largest) # *chiamiamo ricorsivamente*

MAX-HEAPIFY: COMPLESSITÀ

- ▶ Le operazioni non ricorsive che facciamo ad ogni passo richiedono un tempo costante (si tratta di confronti e scambi): tempo $\Theta(1)$
- ▶ Ogni chiamata ricorsiva “elimina” un intero sottoalbero. Nel caso peggiore rimangono un numero di nodi che è pari a $2/3$ la dimensione originale. Perchè?
- ▶ L'equazione di ricorrenza è quindi: $T(n) \leq T(2n/3) + \Theta(1)$
- ▶ Per il teorema dell'esperto, $T(n) = O(\log n)$

MAX-HEAPIFY: COMPLESSITÀ



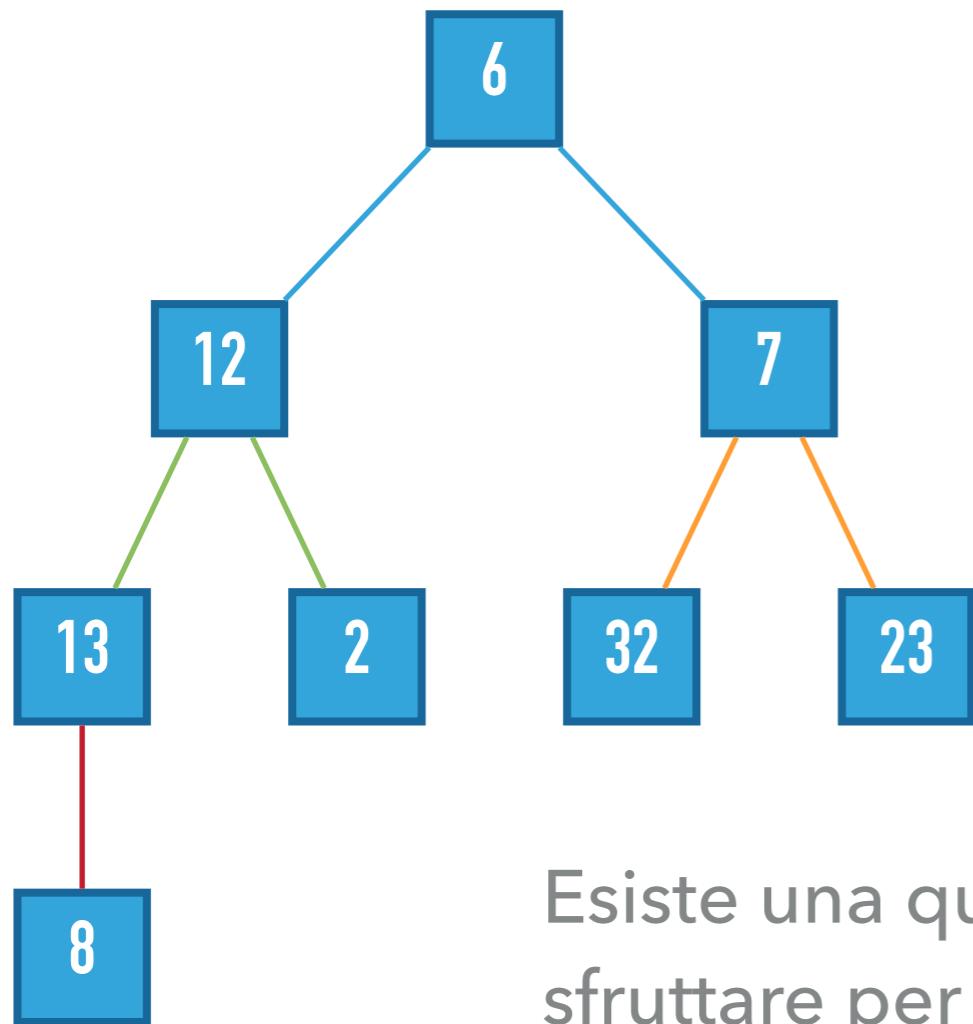
Un altro modo di vedere la complessità:
ad ogni chiamata ricorsiva scendiamo di
un livello nell'albero binario.

L'albero binario ha una profondità che è
logaritmica rispetto al numero di nodi

Dato che ogni “discesa” richiede tempo costante, il
risultato è una complessità di $O(\log n)$

In generale, per uno heap di altezza h , la procedura
max-heapify richiede tempo $O(h)$

BUILD-MAX-HEAP: IDEA

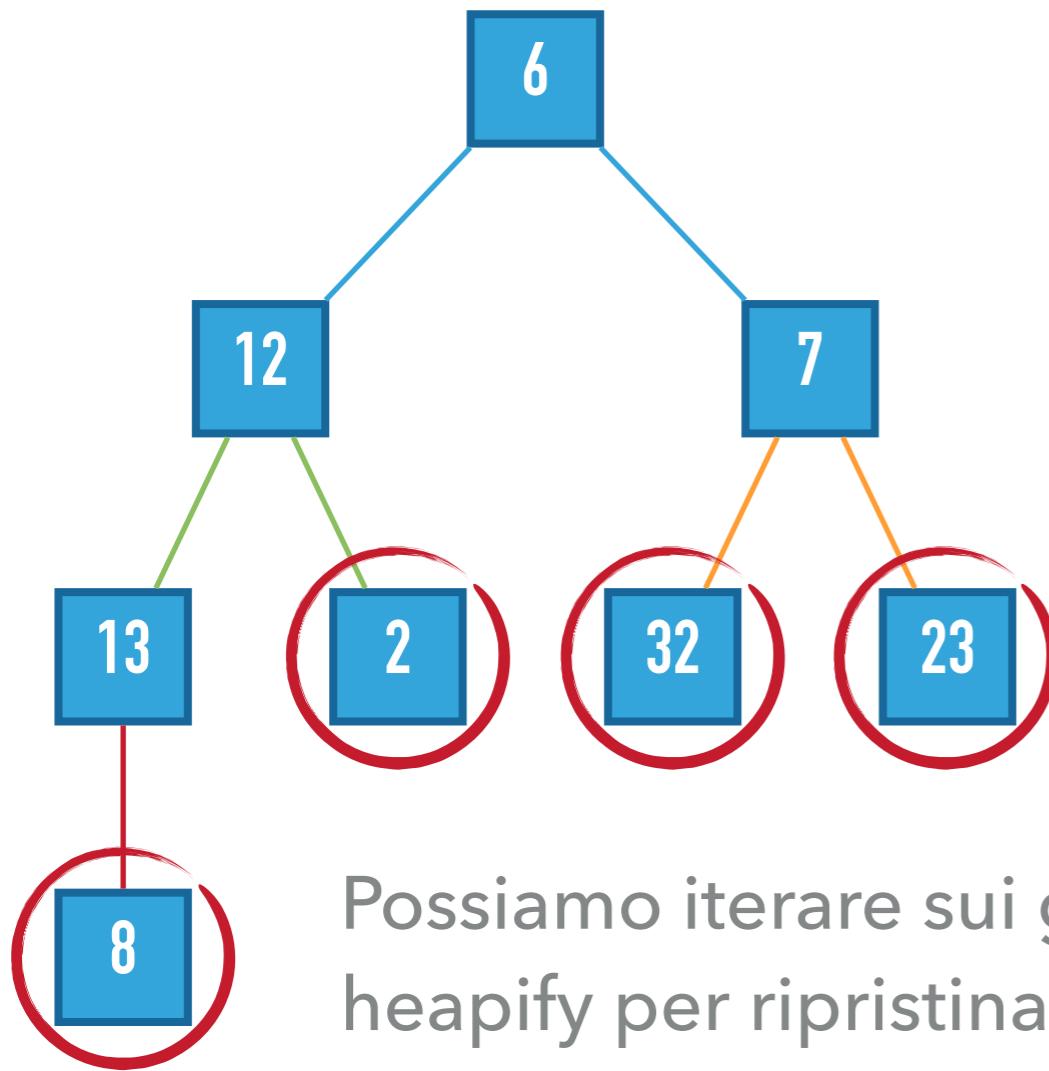


L'array con coi iniziamo non è un max-heap:

| | | | | | | | |
|---|----|---|----|---|----|----|---|
| 6 | 12 | 7 | 13 | 2 | 32 | 23 | 8 |
|---|----|---|----|---|----|----|---|

Esiste una qualche struttura che possiamo sfruttare per costruire un max-heap?

BUILD-MAX-HEAP: IDEA



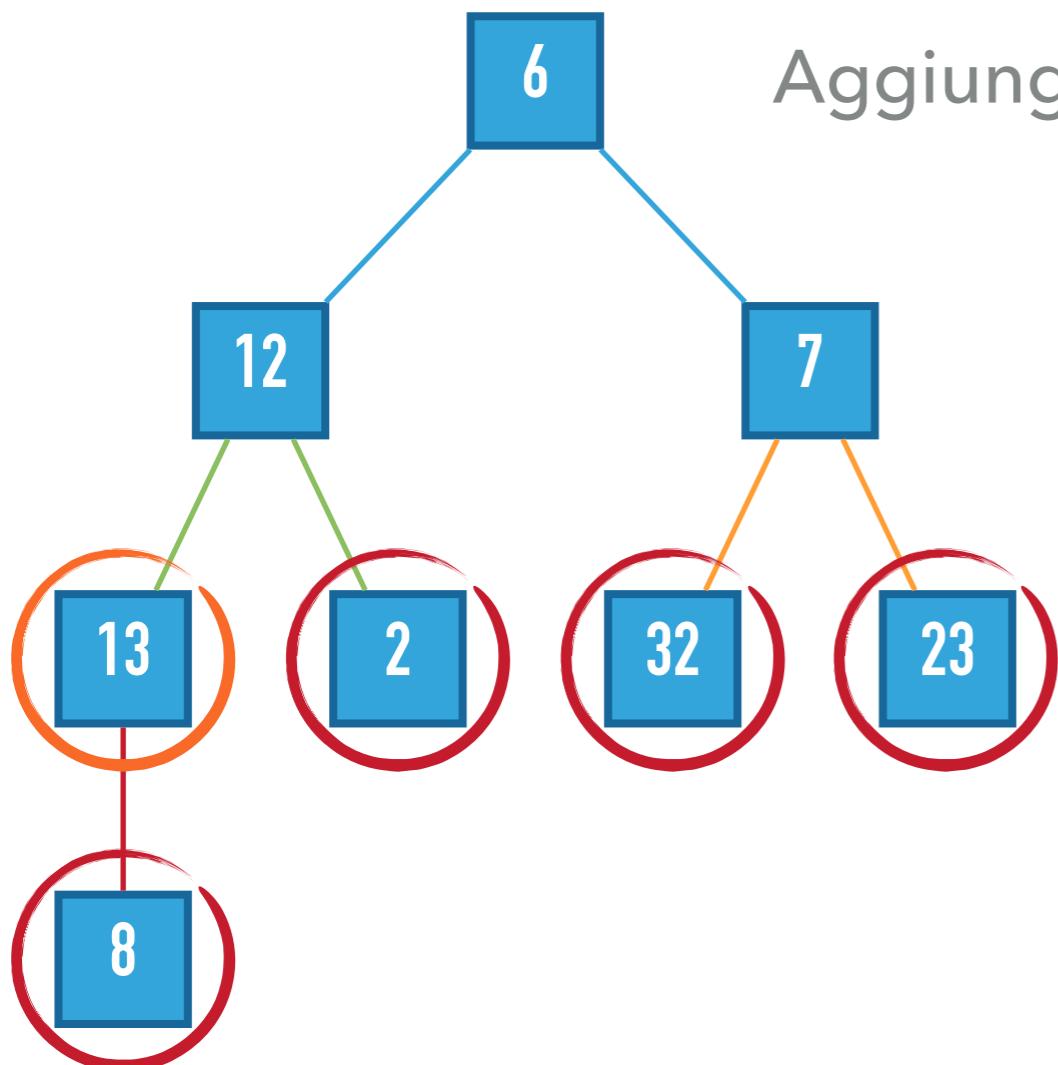
Tutte le foglie sono già max-heap!



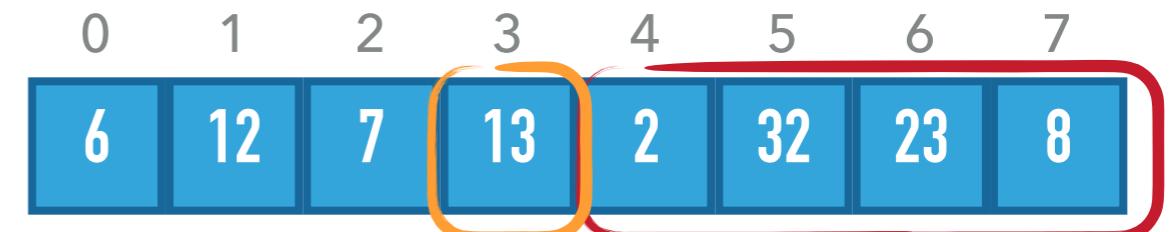
Le foglie sono nell'intervallo di indici da $[n/2]$ a $n - 1$

Possiamo iterare sui genitori delle foglie e chiamare max-heapify per ripristinare la proprietà di max-heap e proseguire sui genitori dei genitori e via così fino a quando non abbiamo creato un max-heap

BUILD-MAX-HEAP: IDEA

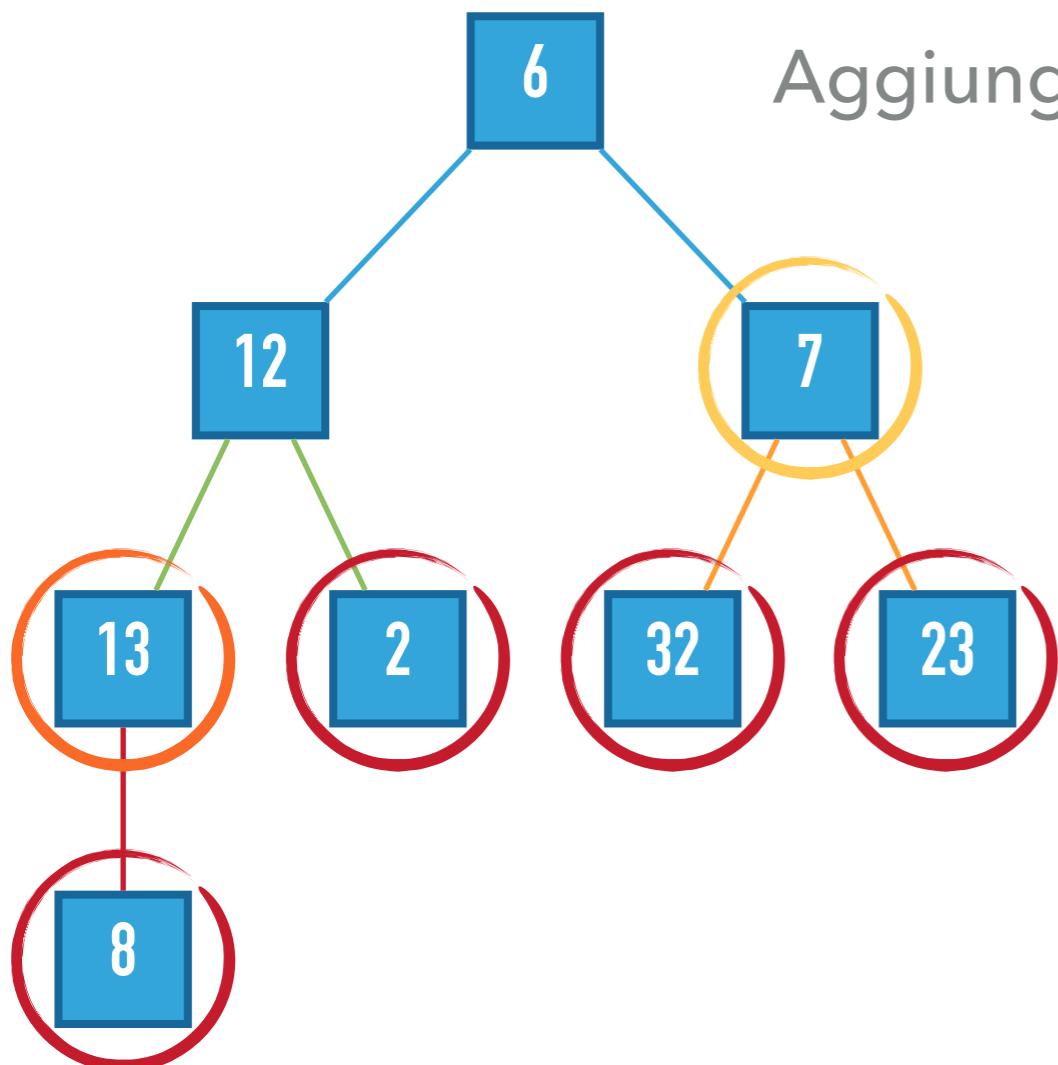


Aggiungiamo l'elemento in posizione $\lfloor n/2 \rfloor - 1$

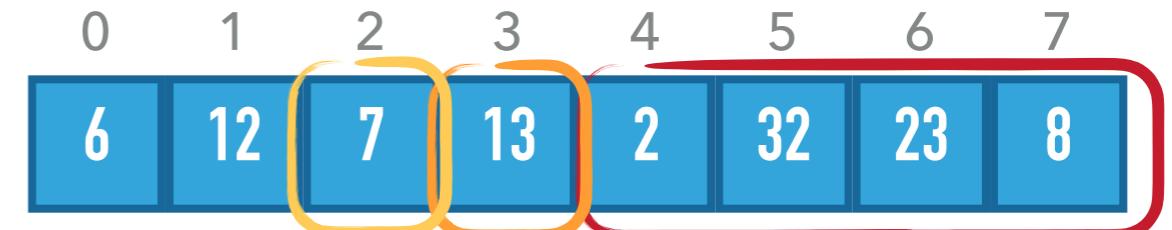


Chiamiamo max-heapify
con indice $\lfloor n/2 \rfloor - 1$

BUILD-MAX-HEAP: IDEA

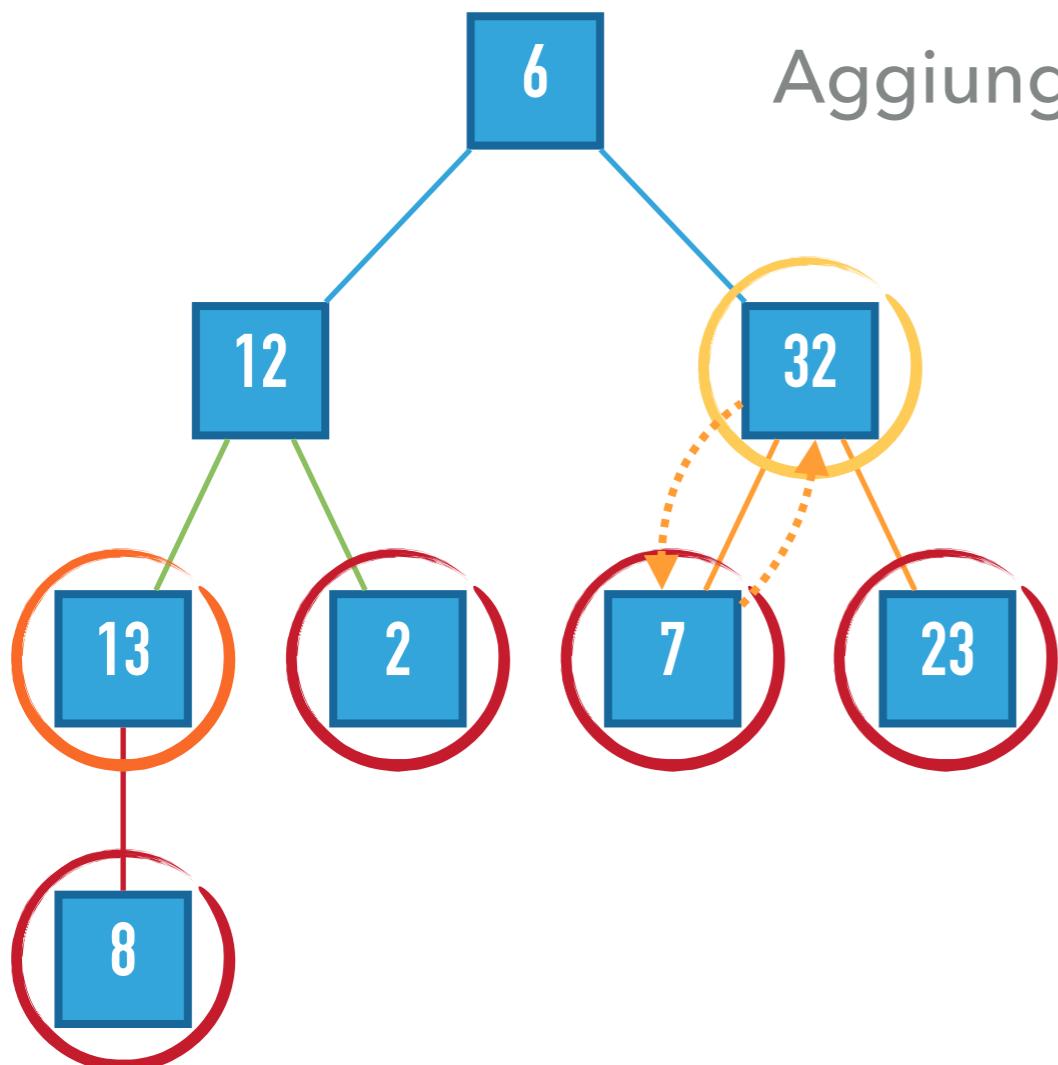


Aggiungiamo l'elemento in posizione $\lfloor n/2 \rfloor - 2$



Chiamiamo max-heapify
con indice $\lfloor n/2 \rfloor - 2$

BUILD-MAX-HEAP: IDEA

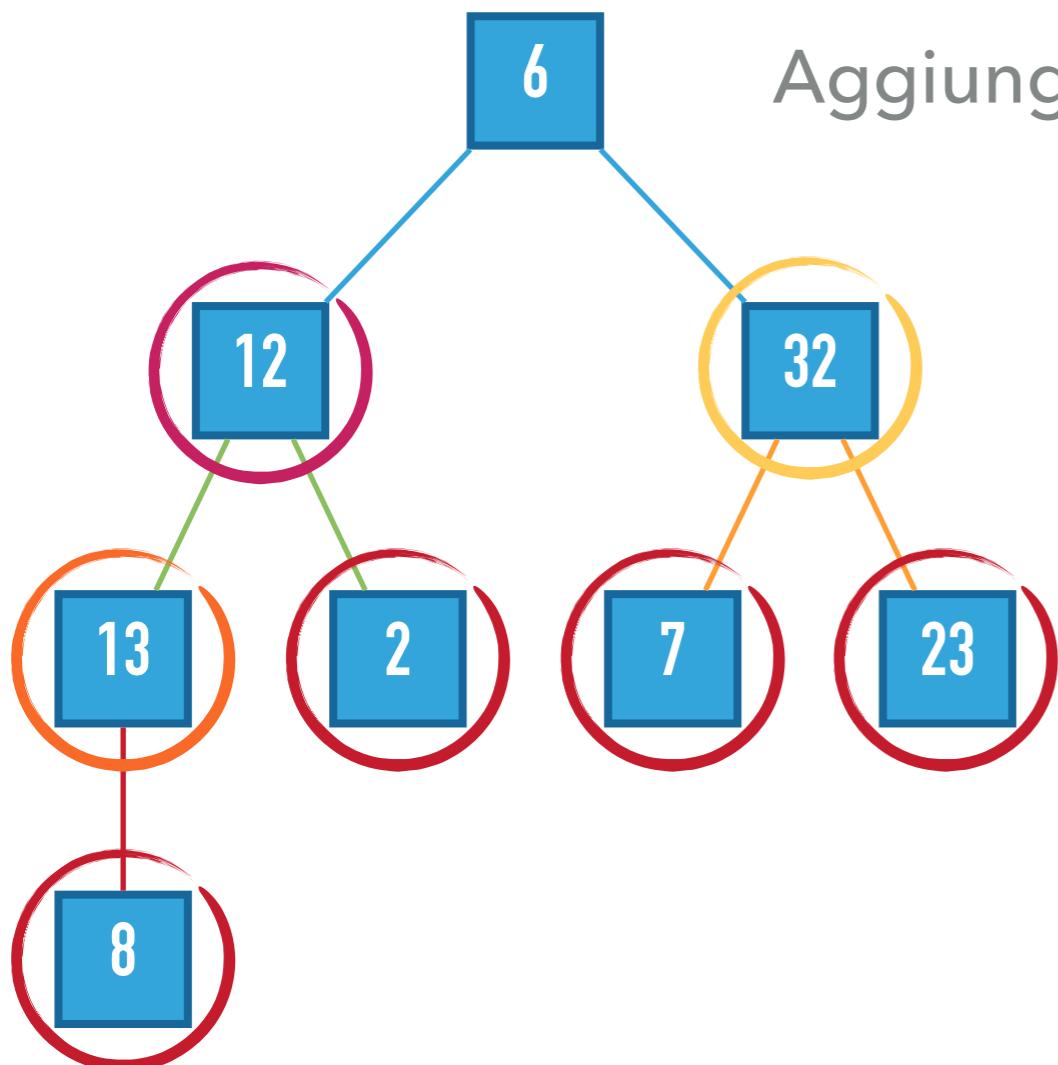


Aggiungiamo l'elemento in posizione $\lfloor n/2 \rfloor - 2$

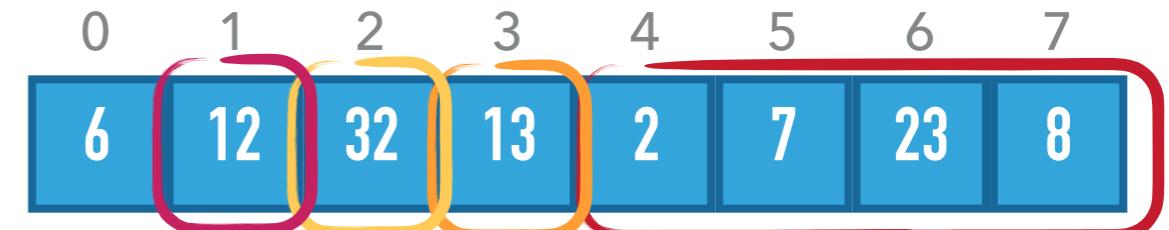


Chiamiamo max-heapify
con indice $\lfloor n/2 \rfloor - 2$

BUILD-MAX-HEAP: IDEA

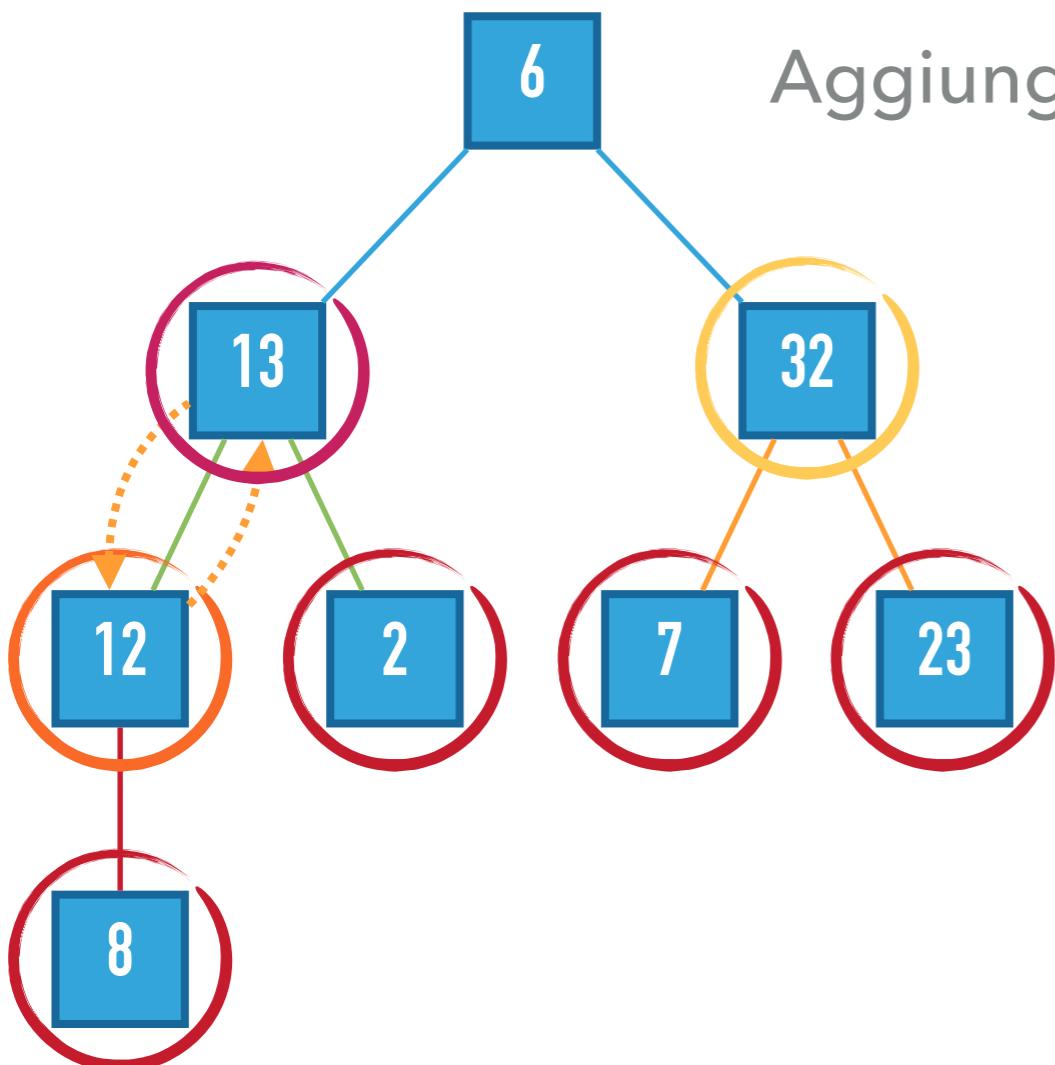


Aggiungiamo l'elemento in posizione 1

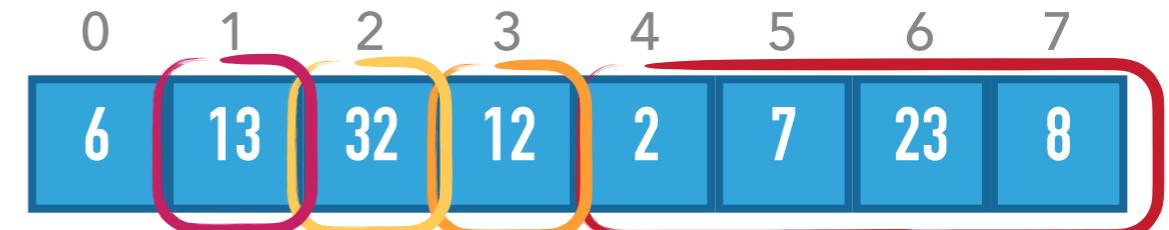


Chiamiamo max-heapify
con indice 1

BUILD-MAX-HEAP: IDEA

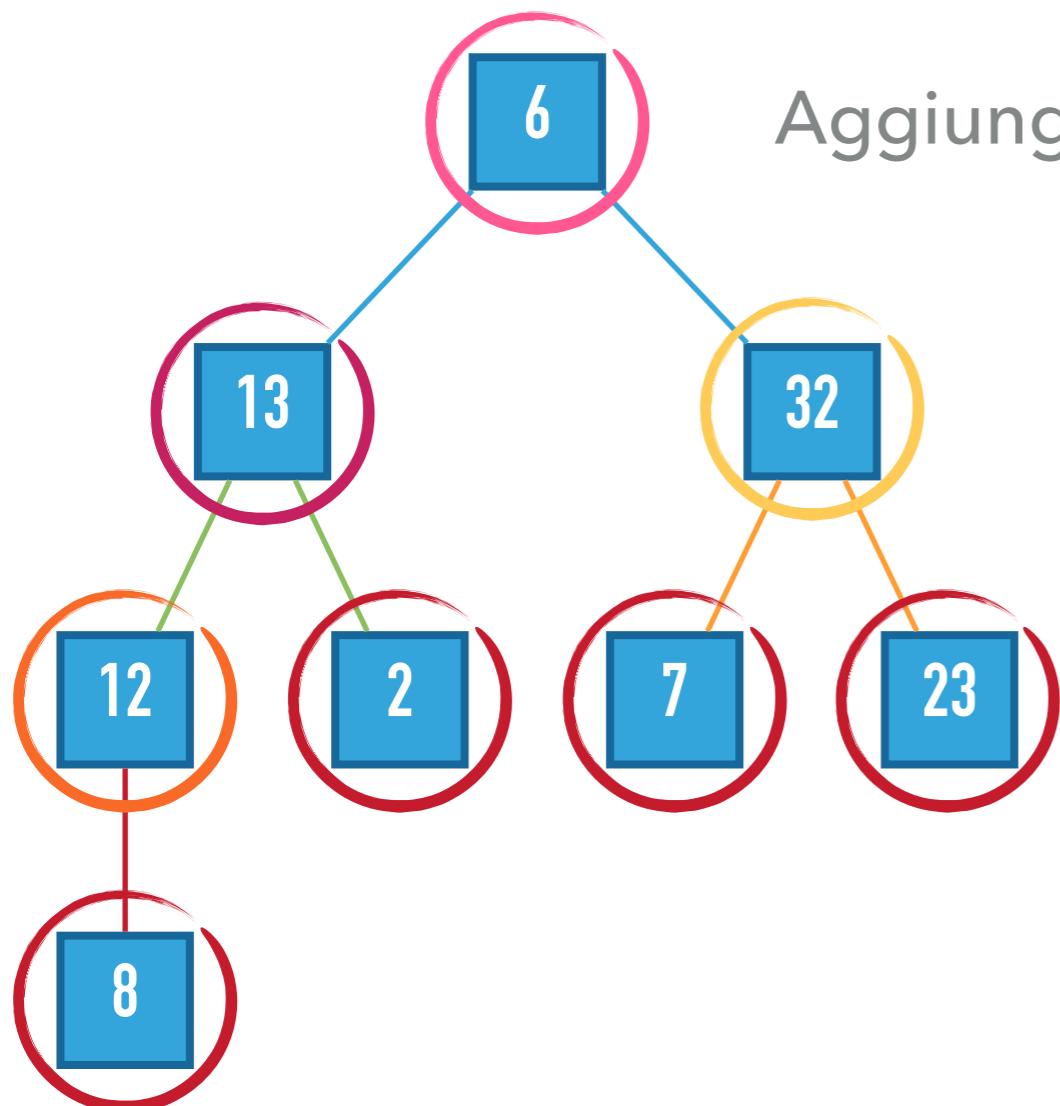


Aggiungiamo l'elemento in posizione 1



Chiamiamo max-heapify
con indice 1

BUILD-MAX-HEAP: IDEA

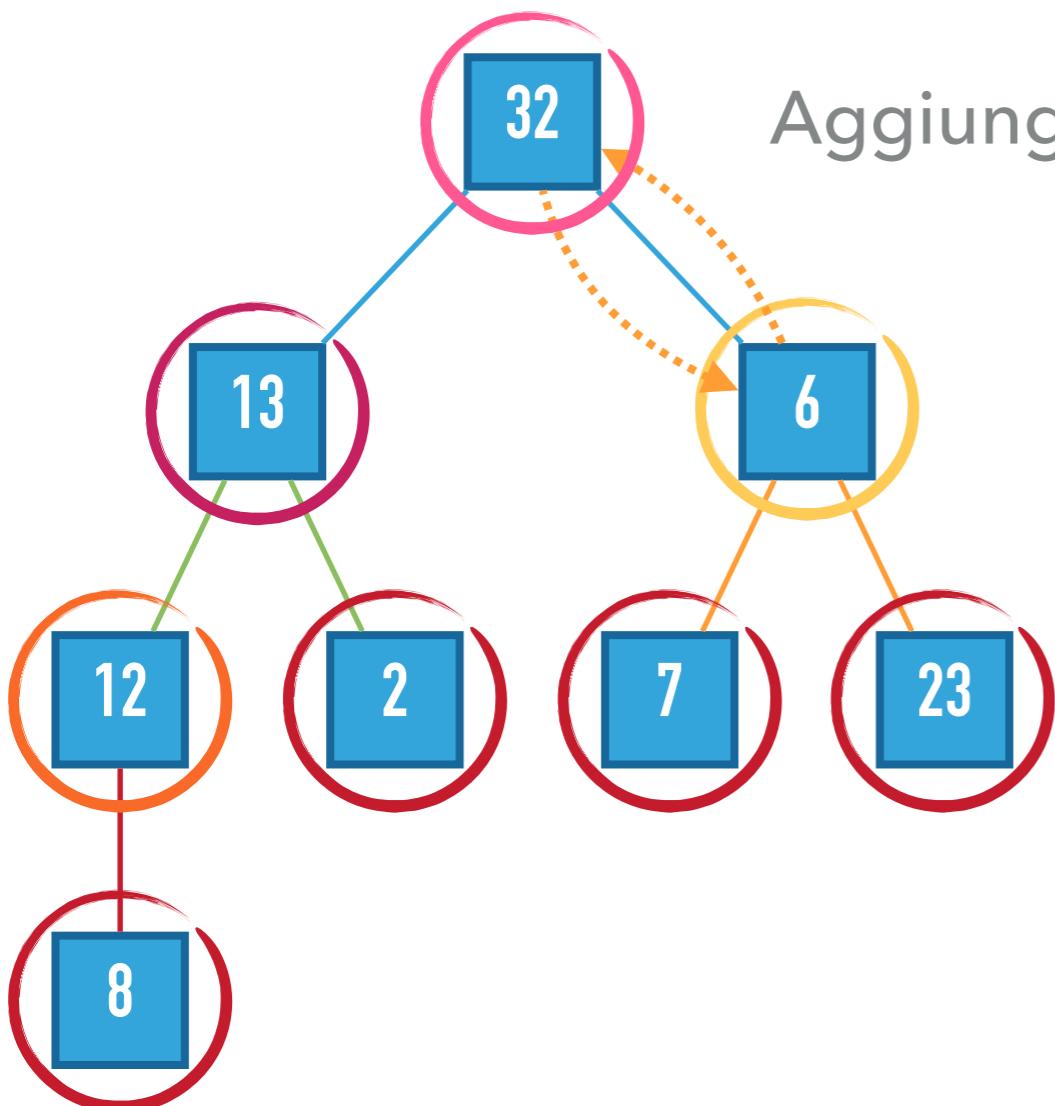


Aggiungiamo l'elemento in posizione 0

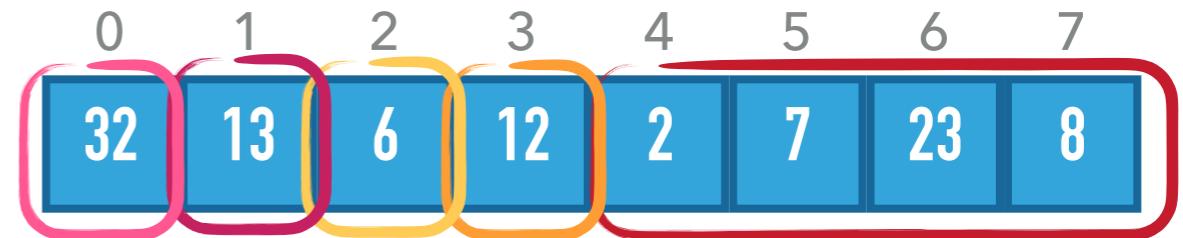


Chiamiamo max-heapify
con indice 0

BUILD-MAX-HEAP: IDEA

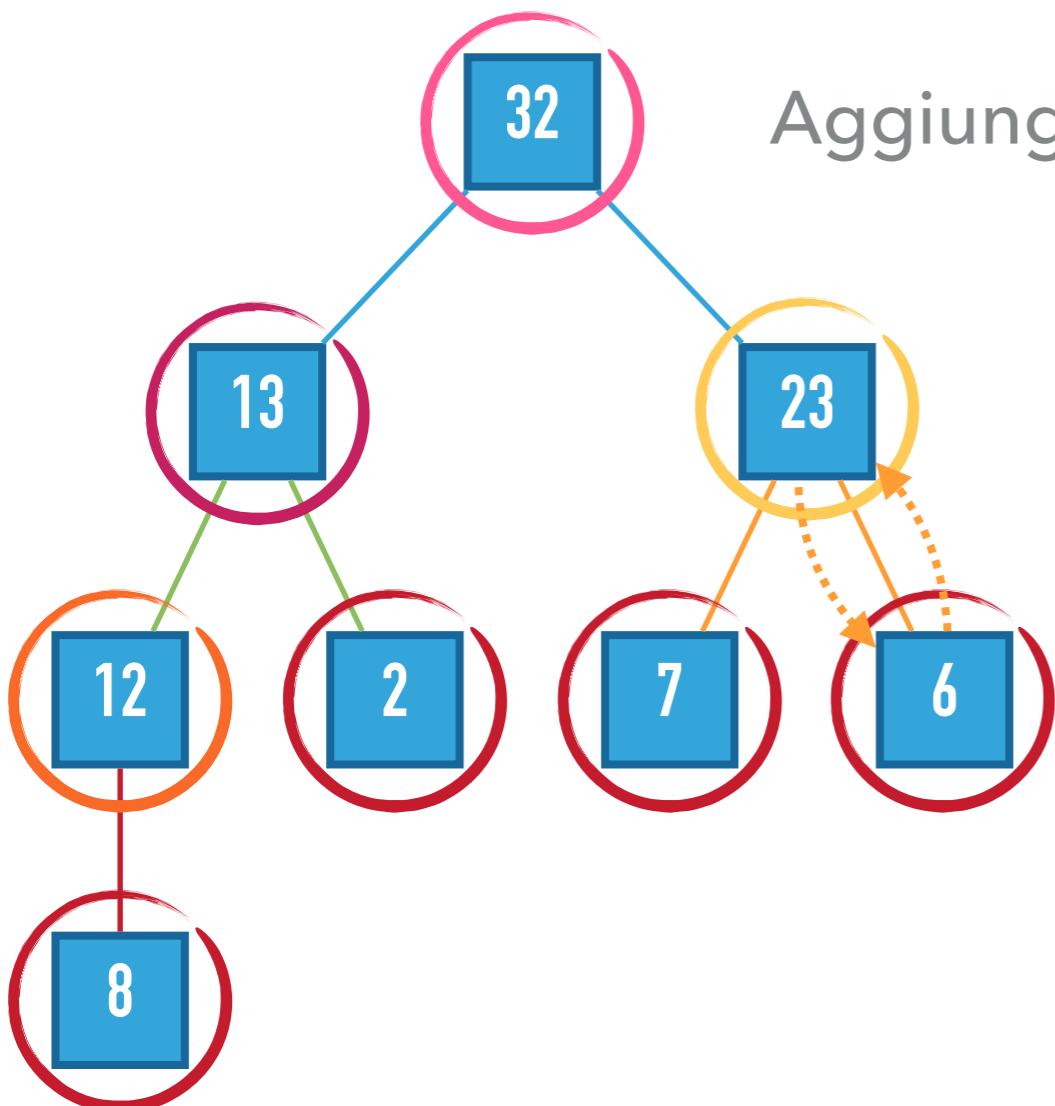


Aggiungiamo l'elemento in posizione 0



Chiamiamo max-heapify
con indice 0

BUILD-MAX-HEAP: IDEA



Aggiungiamo l'elemento in posizione 0



Chiamiamo max-heapify
con indice 0

BUILD-MAX-HEAP: PSEUDOCODICE

- ▶ Parametri: A (array)
- ▶ for i from $\lfloor n/2 \rfloor - 1$ down to 0 (o da $\lfloor n/2 \rfloor$ a 1 se numeriamo da 1)
 - ▶ max-heapify(A, i)
- ▶ Inizializzazione: tutti gli elementi da $\lfloor n/2 \rfloor$ a $n - 1$ sono, singolarmente, max-heap.
- ▶ Invariante: i figli dell'elemento i -esimo hanno indice maggiore di i , quindi sono già max-heap e dopo max-heapify anche $A[i:]$ sarà una heap
- ▶ Terminazione: arrivati a $i = 0$, tutti gli elementi sono radici di un max-heap, in particolare il primo elemento è l'intero array

TEMPO DI CALCOLO

Ogni chiamata a max-heapify richiede tempo $O(h)$

Ed eseguiamo un numero lineare di chiamate a max-heapify

Ma ognuna di queste chiamate dipende dal valore di h che **non** è costante:

Abbiamo un sotto-albero di altezza $\lceil \log_2 n \rceil$, due di altezza $\lceil \log_2 n \rceil - 1$...

Fino ad arrivare a $\lceil n/2 \rceil$ albero di altezza 0

In generale, abbiamo al più $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ nodi che sono radici di (sotto-)alberi di altezza h .

Let's prove it.

TEMPO DI CALCOLO

Quindi il tempo di richiesto è $\sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)$, ovvero $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right)$.

Sapendo che $\sum_{h=0}^{+\infty} \frac{h}{2^h} = \frac{1/2}{(1 - 1/2)^2} = 2$

Otteniamo $O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \left\lceil \frac{h}{2^{h+1}} \right\rceil\right) = O\left(n \sum_{h=0}^{+\infty} \frac{h}{2^{h+1}}\right) = O(n)$

Quindi la procedura max-heapify richiede tempo *lineare* rispetto alla dimensione dell'array.

HEAPSORT: DOVE SIAMO ARRIVATI

Array da ordinare



BUILD-MAX-HEAP

Inseriamo tutti gli elementi
nella coda di priorità

MAX-HEAP

CODA DI
PRIORITÀ

QUESTA PARTE CI MANCA

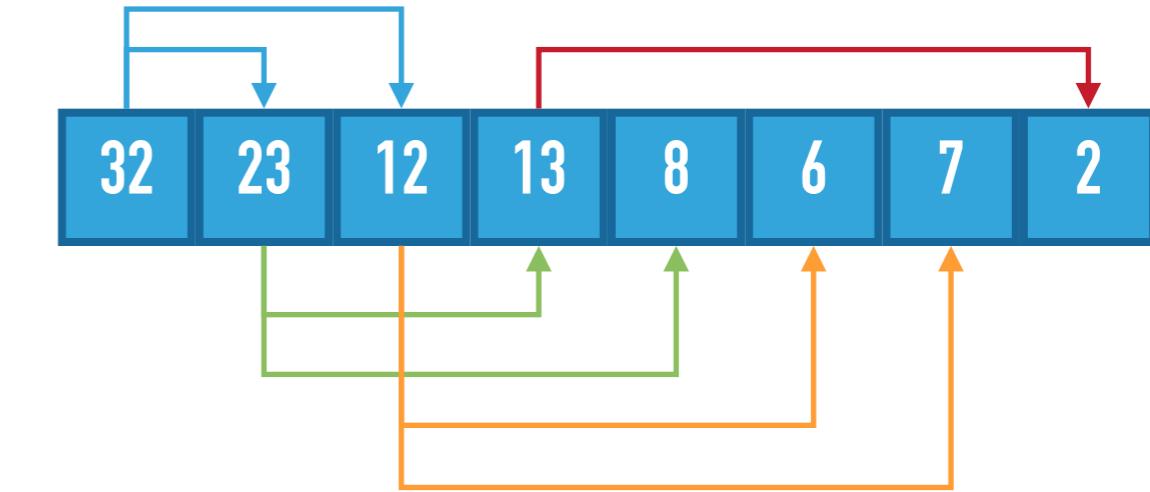
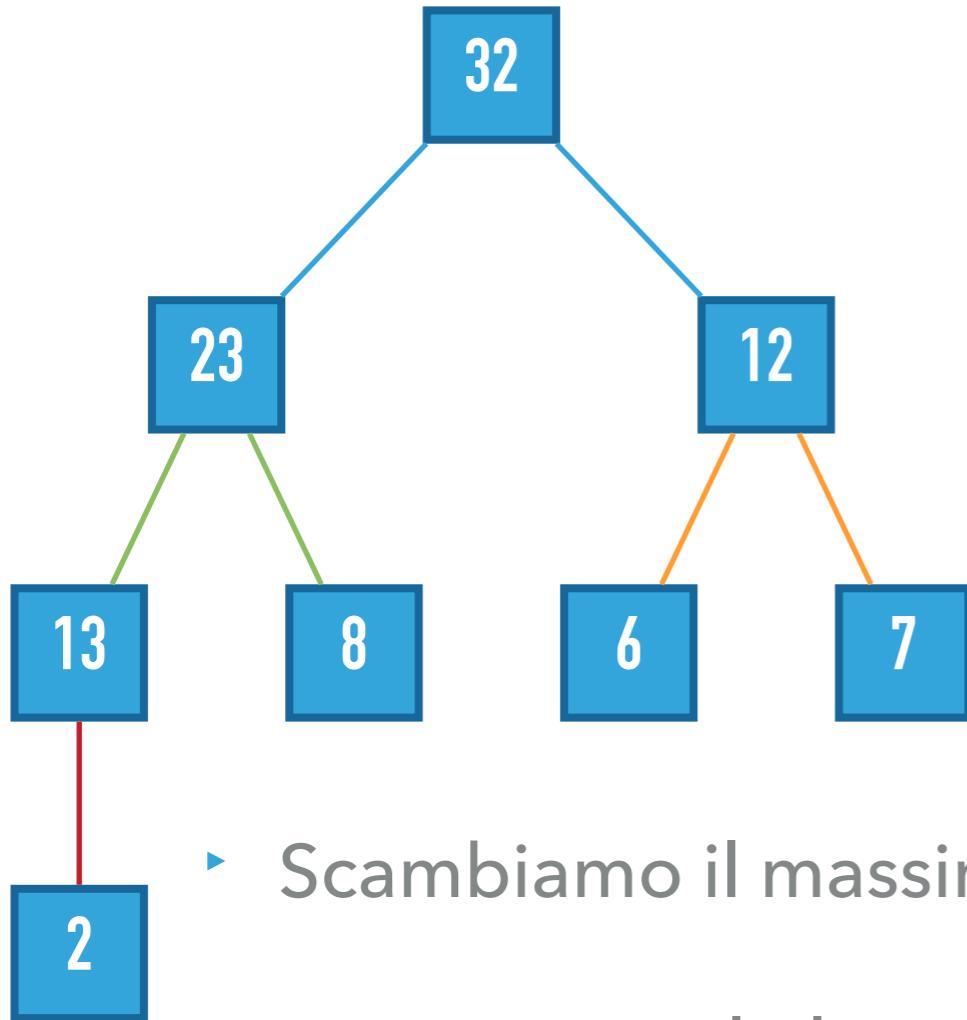
Estraiamo i valori uno ad uno, dal più
grande al più piccolo



Una struttura dati in cui:

- Possiamo inserire elementi
- Possiamo estrarre il massimo
tra tutti gli elementi inseriti

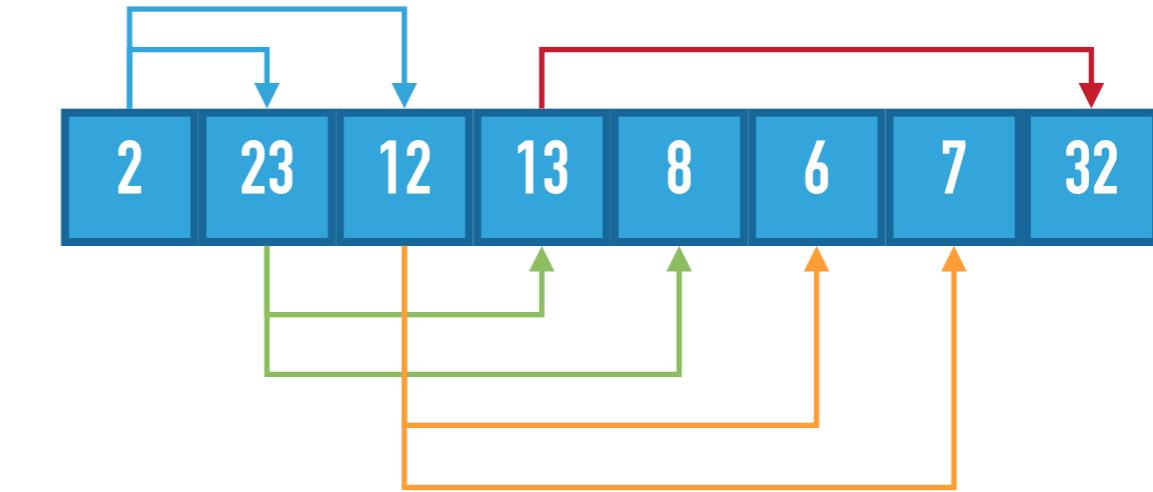
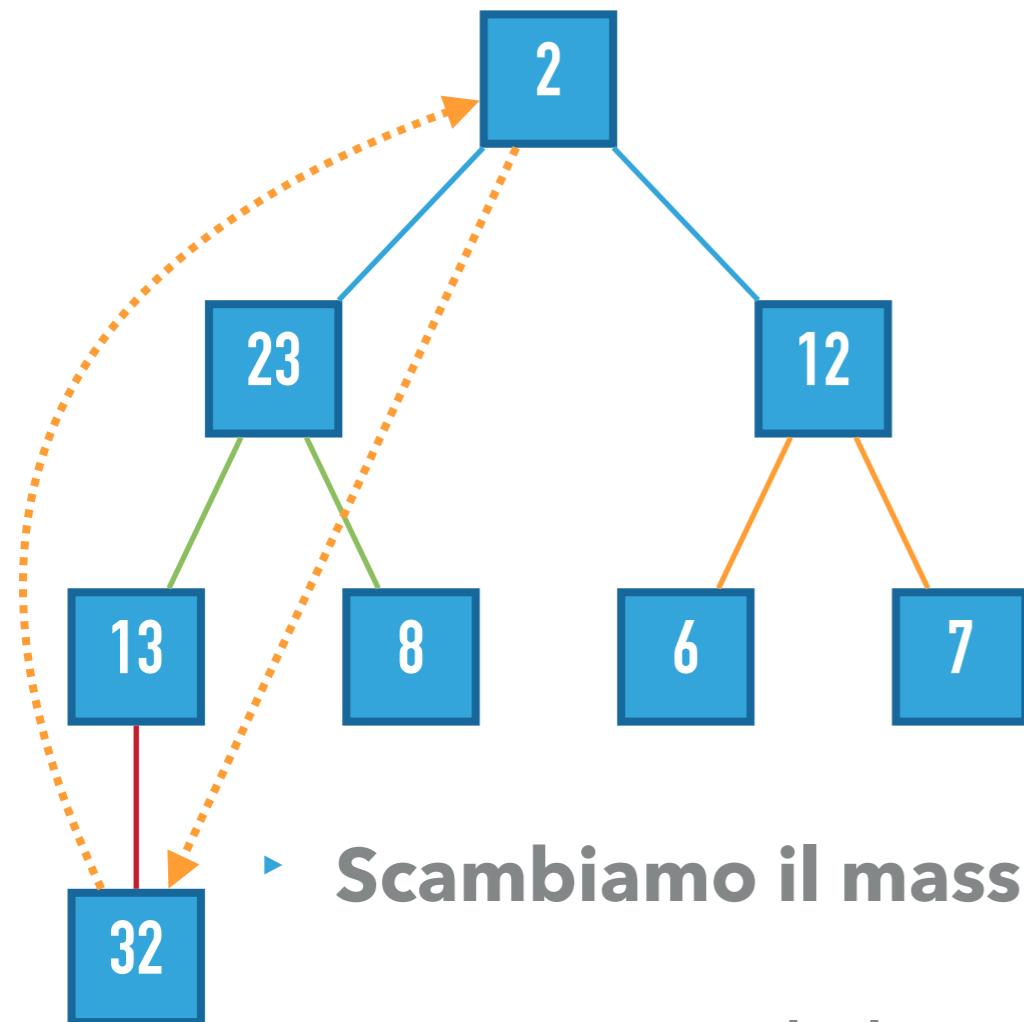
RIMOZIONE DEL MASSIMO



IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia (decrementando `heap_size`)
- ▶ Chiamiamo `max-heapify` per ristabilire la proprietà di max-heap

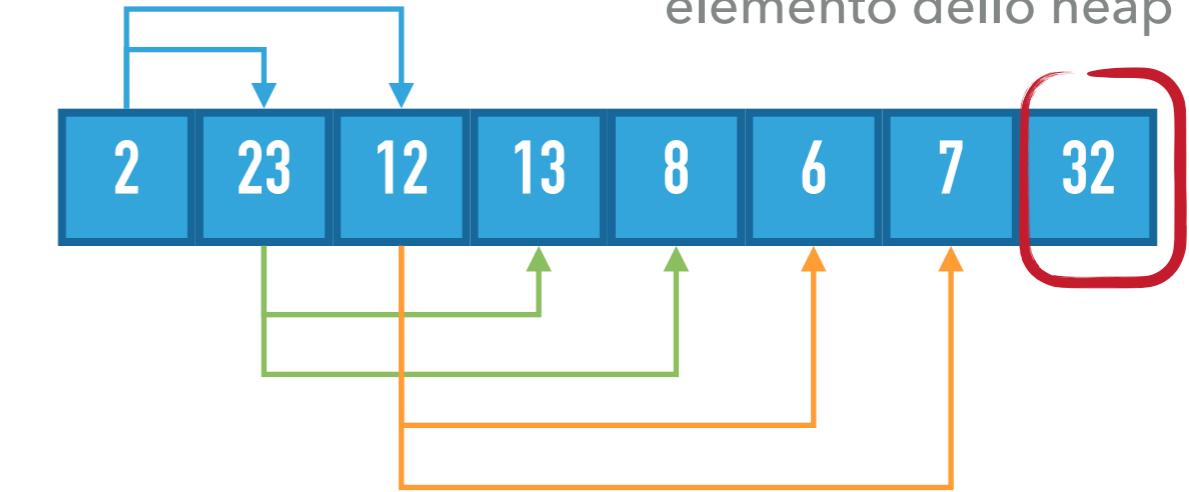
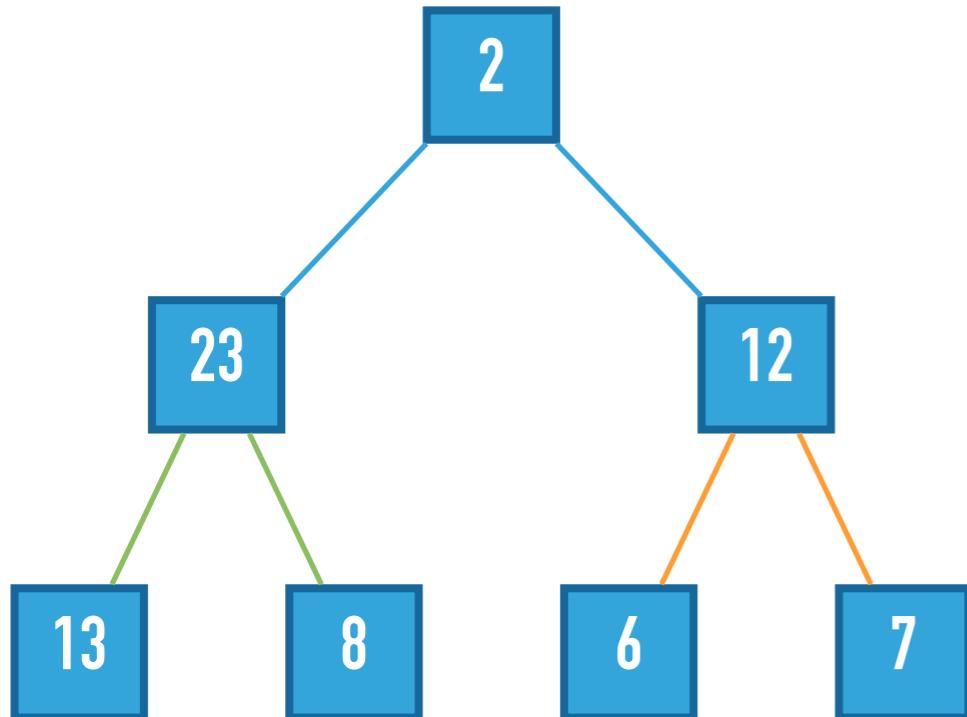
RIMOZIONE DEL MASSIMO



IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia
- ▶ Chiamiamo max-heapify per ristabilire la proprietà di max-heap

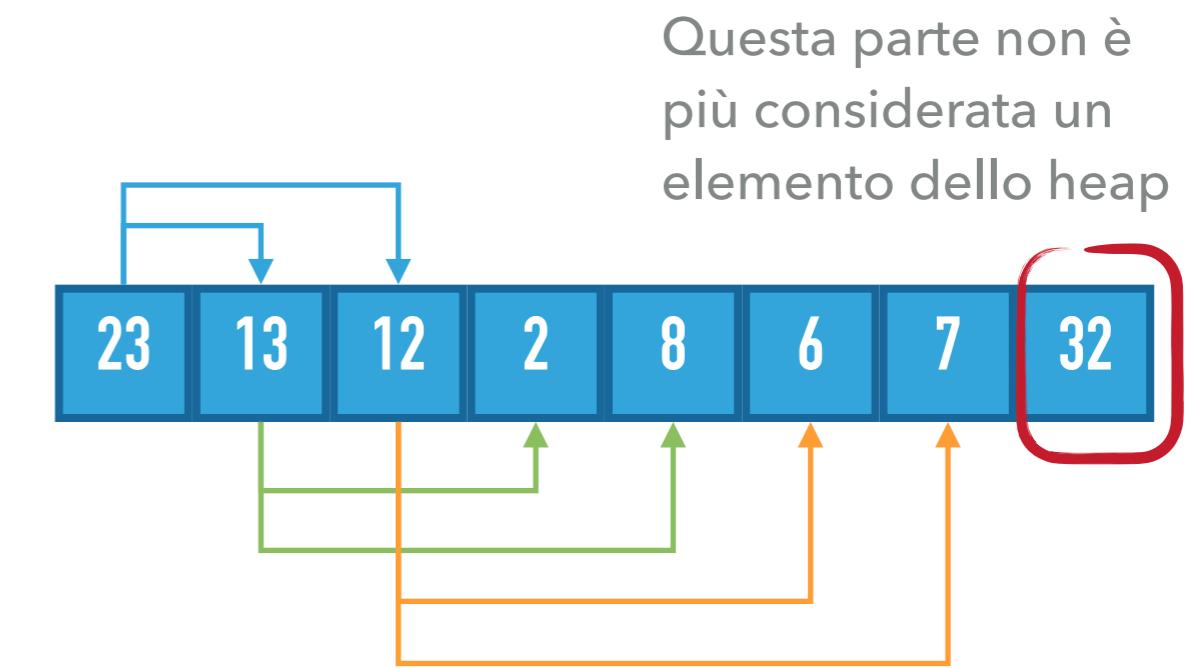
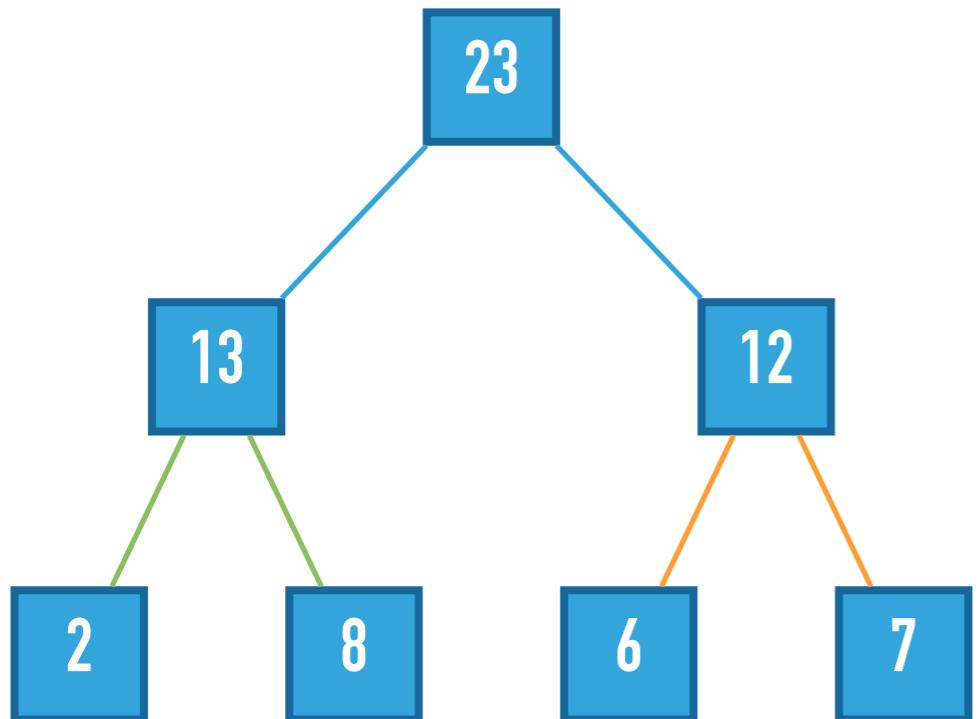
RIMOZIONE DEL MASSIMO



IDEA DI BASE

- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ **Rimuoviamo l'ultima foglia**
- ▶ Chiamiamo max-heapify per ristabilire la proprietà di max-heap

RIMOZIONE DEL MASSIMO



IDEA DI BASE

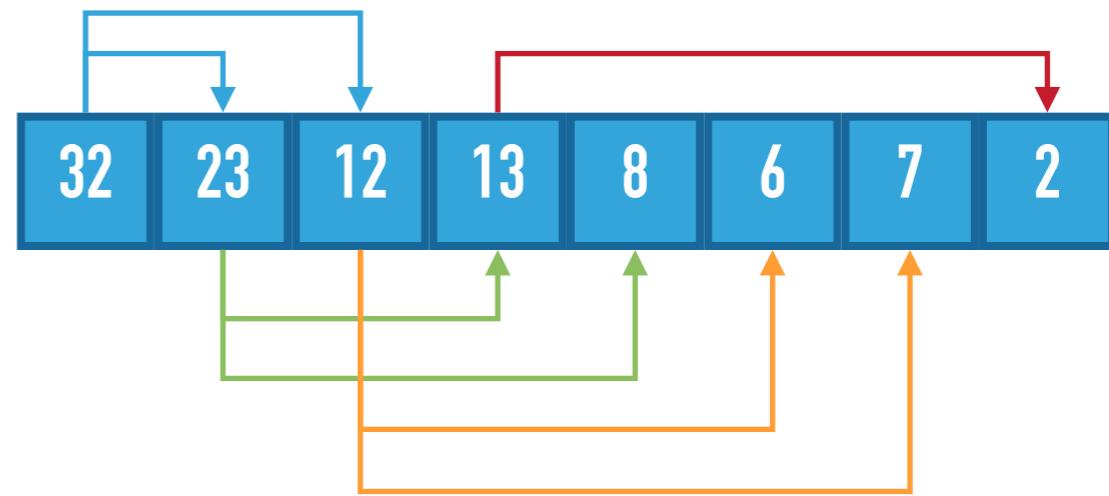
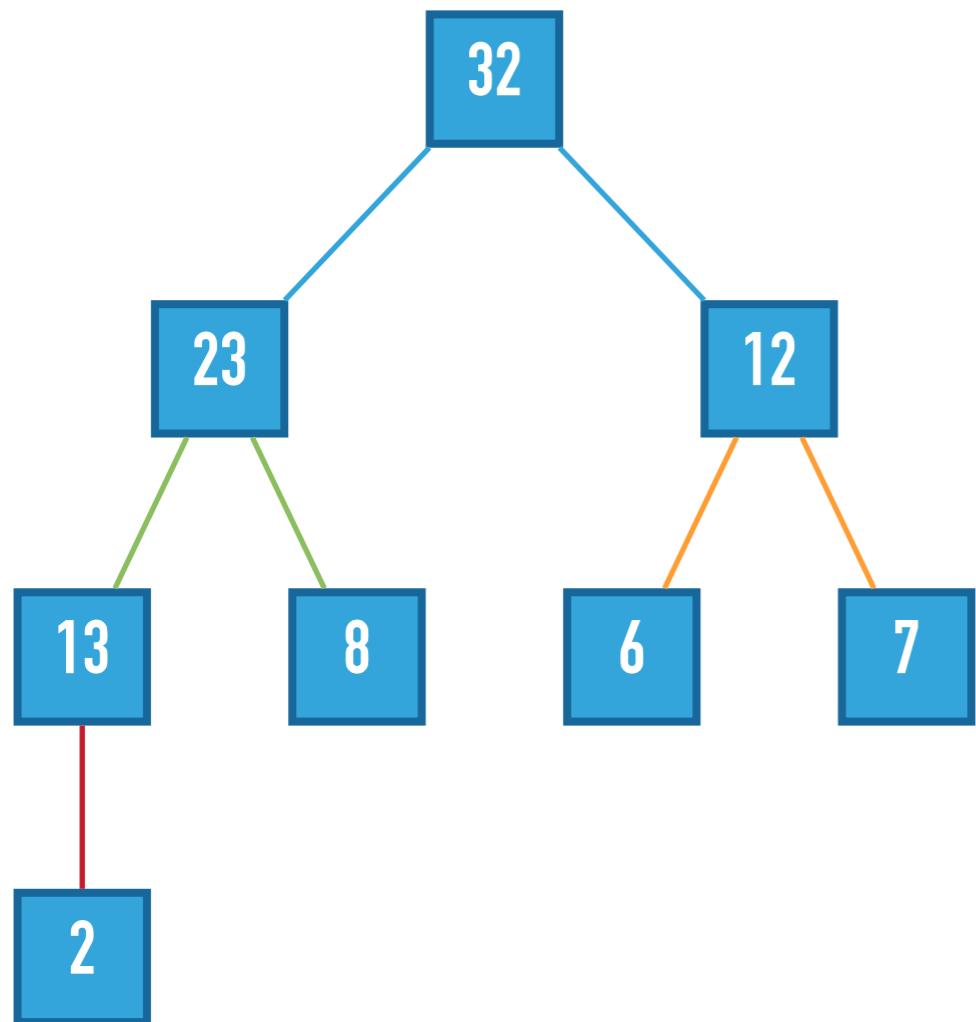
- ▶ Scambiamo il massimo con l'ultima foglia
- ▶ Rimuoviamo l'ultima foglia
- ▶ **Chiamiamo max-heapify per ristabilire la proprietà di max-heap**



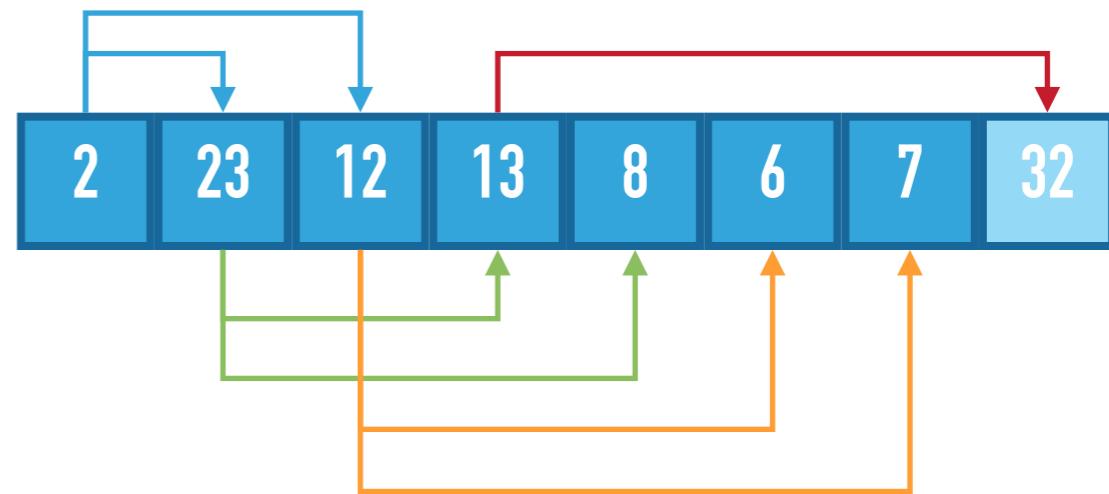
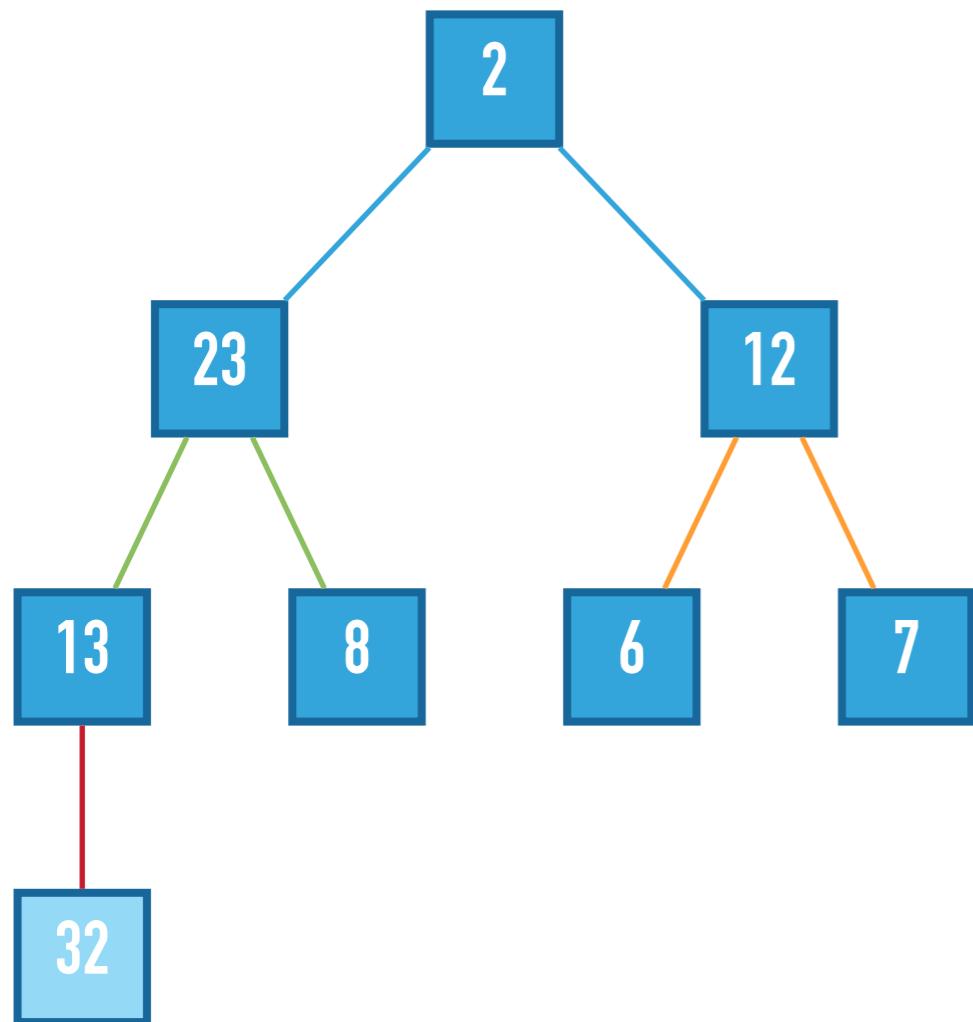
HEAPSORT: PSEUDOCODICE

- ▶ Parametri: A (array)
- ▶ build-max-heap(A) $\leftarrow O(n)$
- ▶ heap-size(A) = n
- ▶ for i from $n - 1$ down to 1
 - ▶ swap A[i] and A[0]
 - ▶ decrement heap-size(A) by 1
 - ▶ max-heapify(A, 0) $\leftarrow O(\log n)$
- ▶ Eseguiamo una operazione di costo $O(n)$ e un numero lineare di operazioni di costo $O(\log n)$
- ▶ L'algoritmo di heapsort ordina l'array in tempo $O(n \log n)$

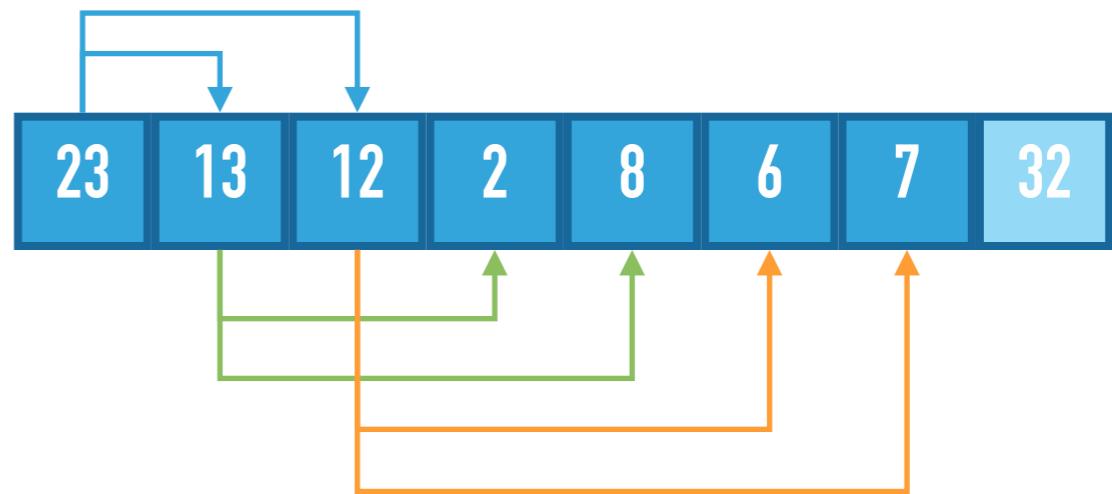
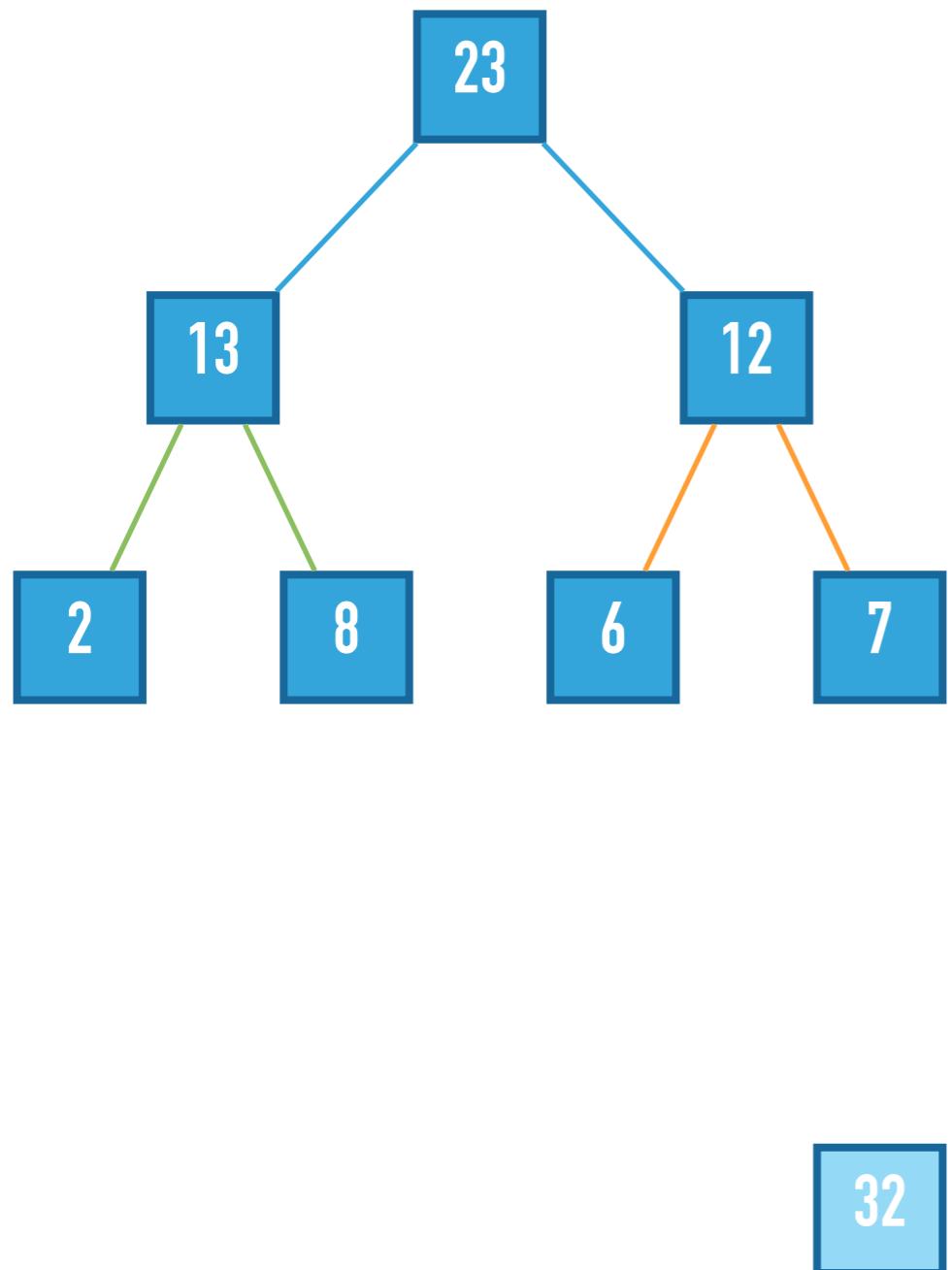
HEAPSORT: ESEMPIO



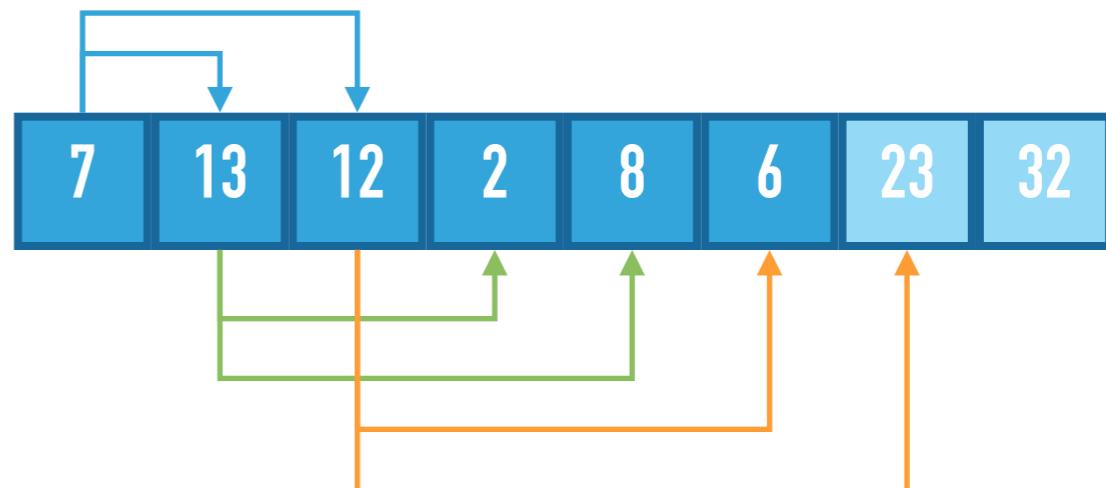
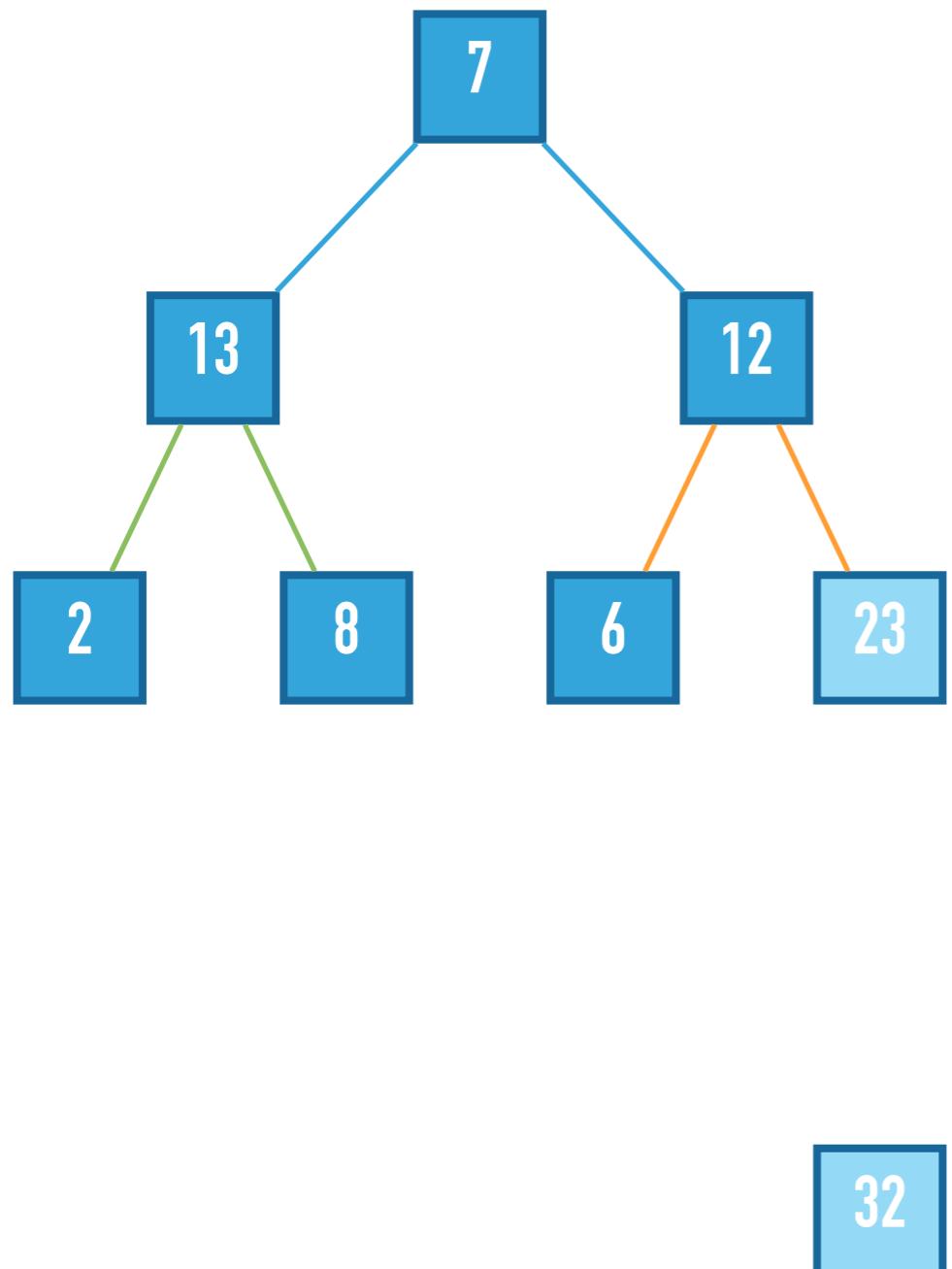
HEAPSORT: ESEMPIO



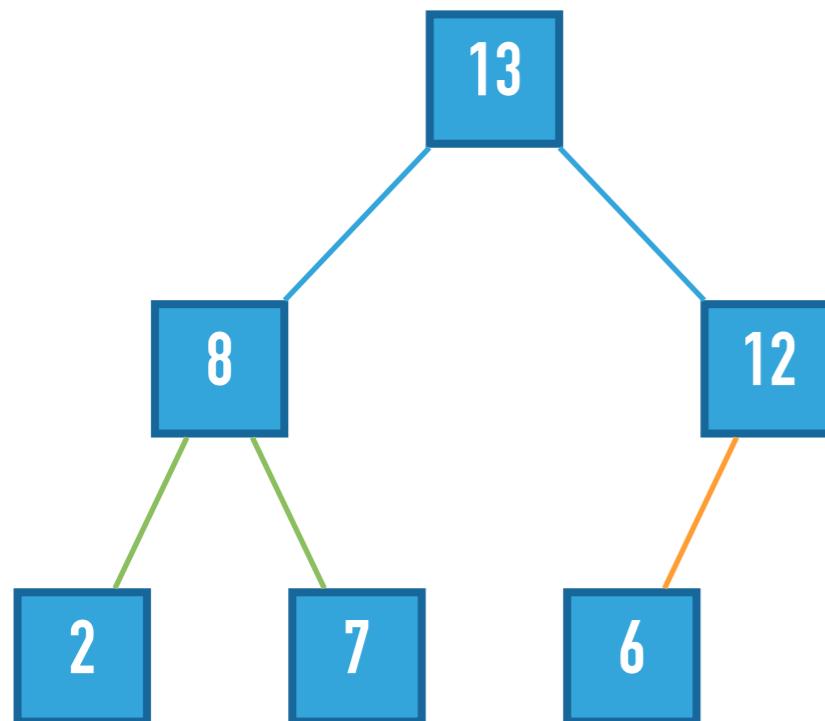
HEAPSORT: ESEMPIO



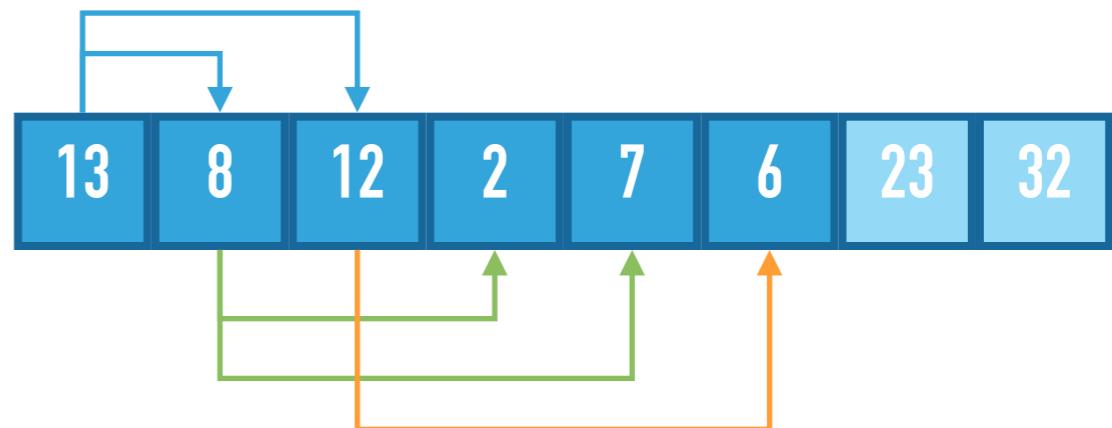
HEAPSORT: ESEMPIO



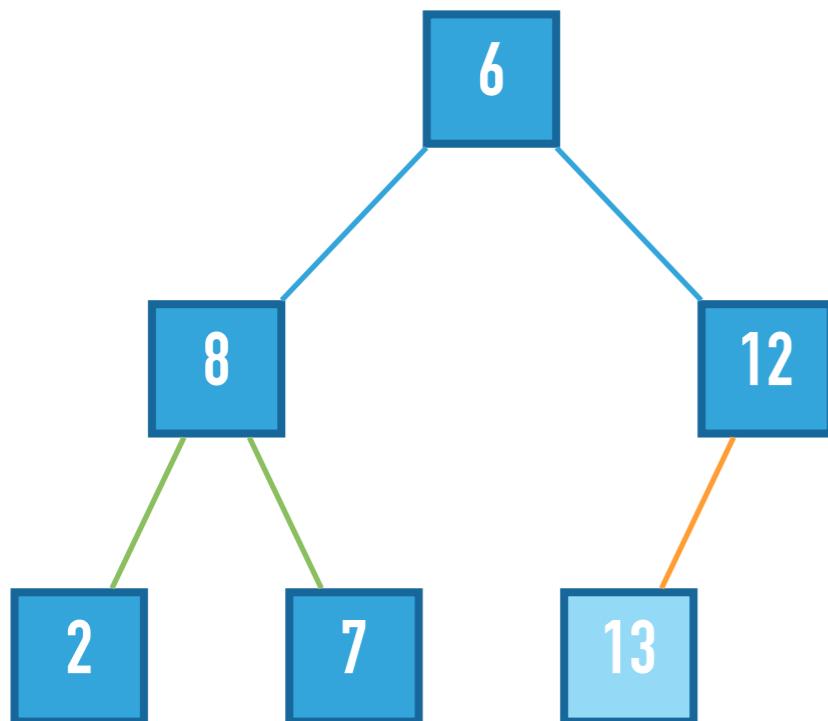
HEAPSORT: ESEMPIO



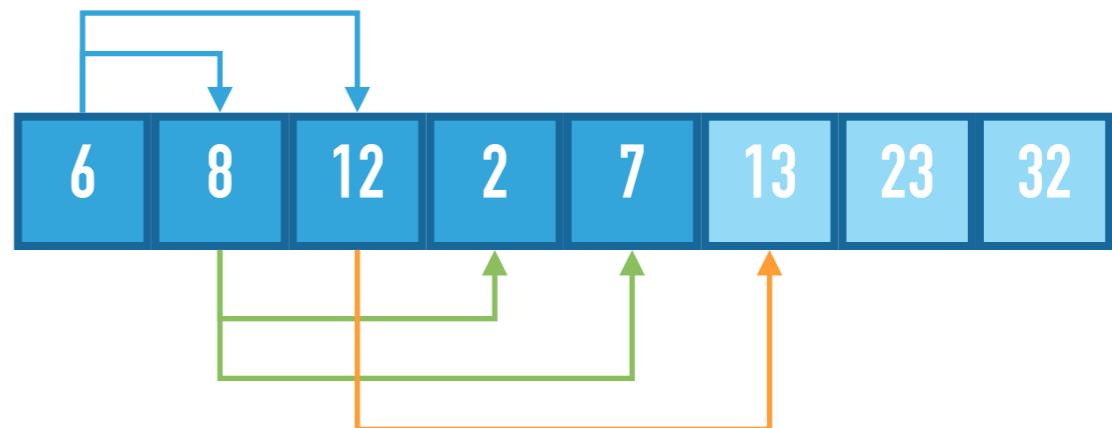
23 32



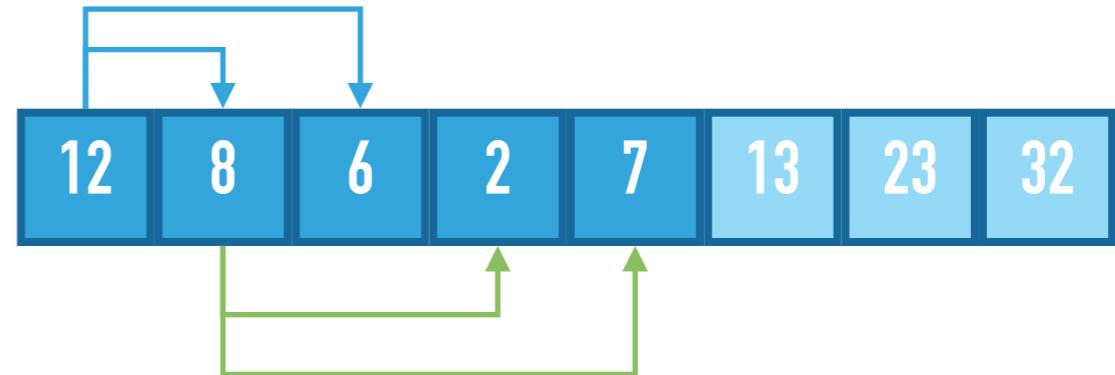
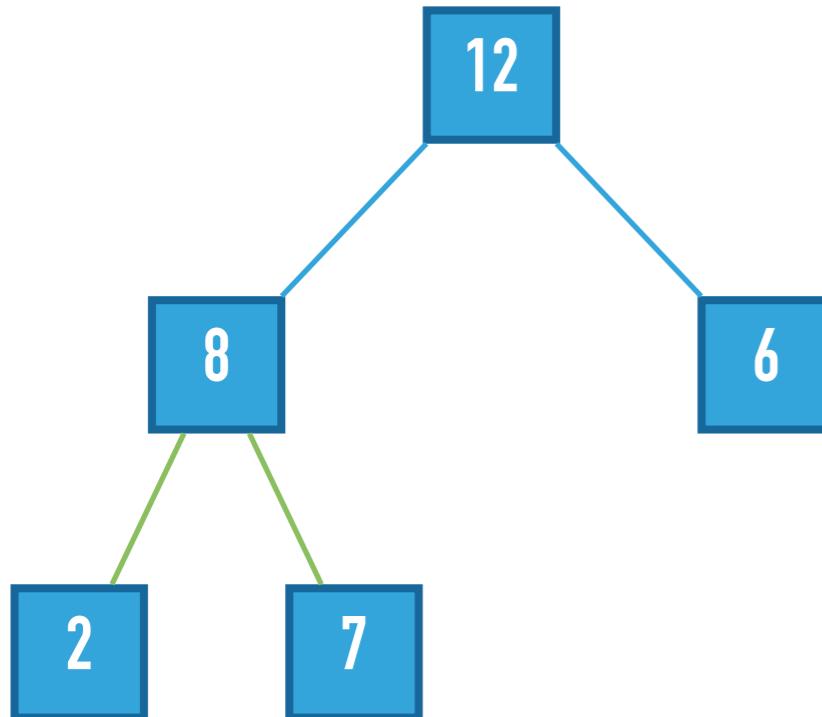
HEAPSORT: ESEMPIO



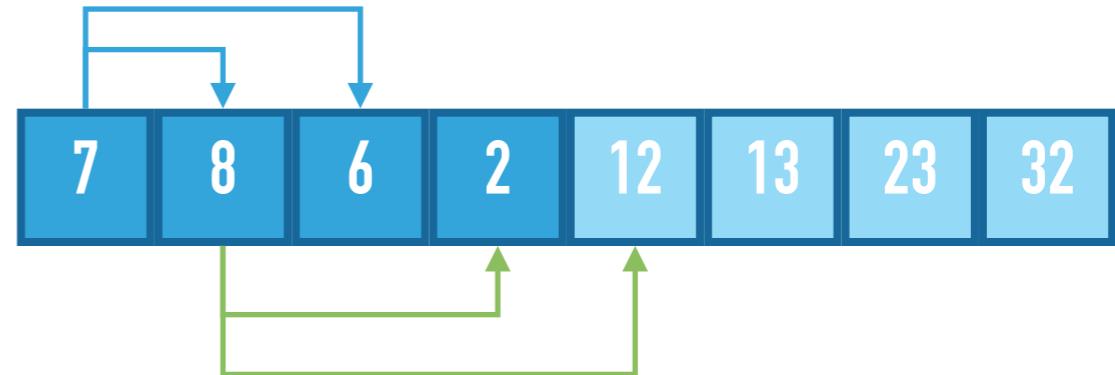
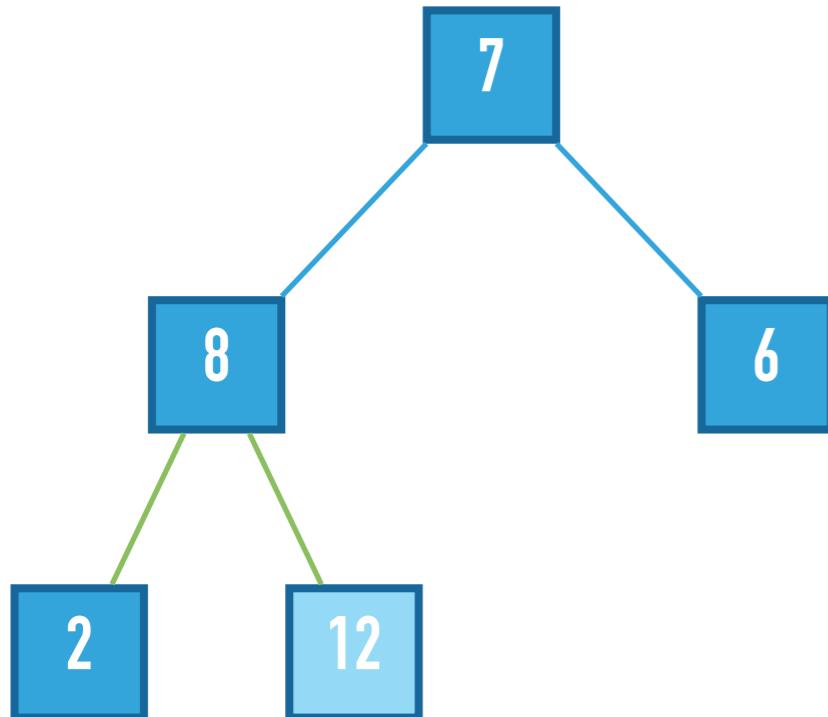
23 32



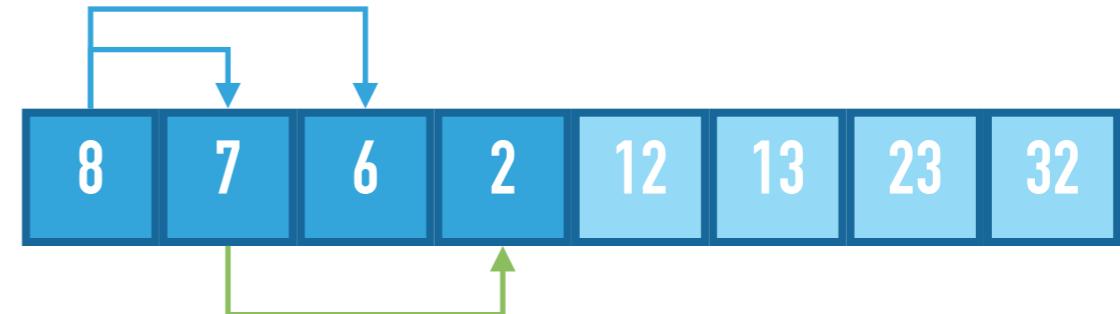
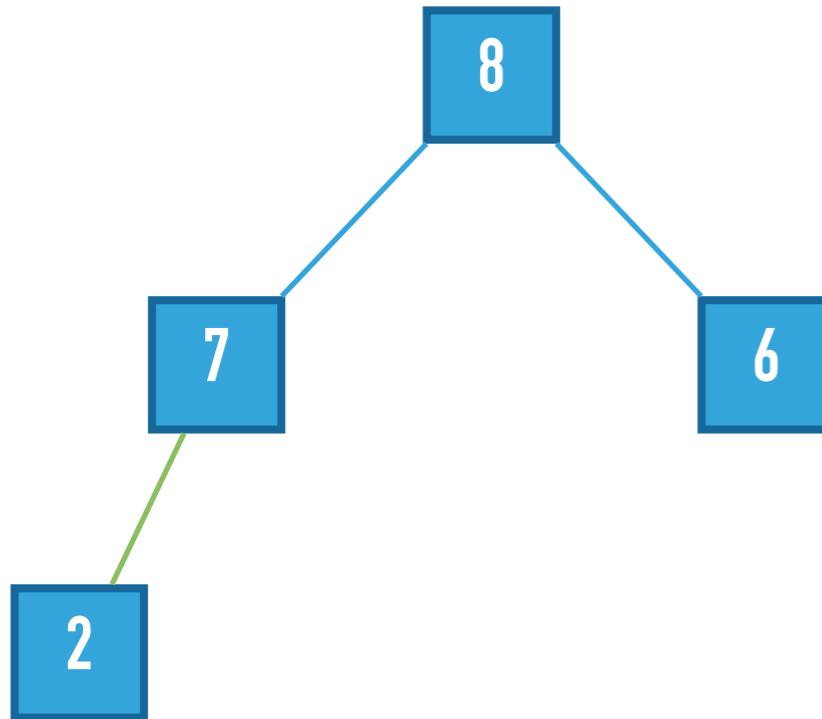
HEAPSORT: ESEMPIO



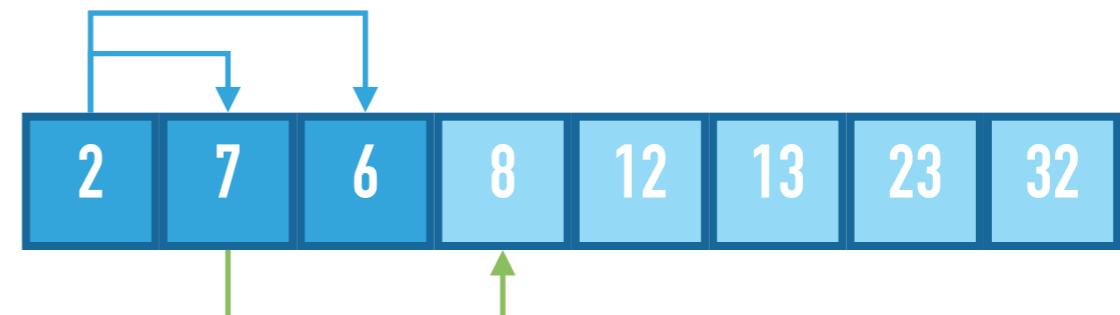
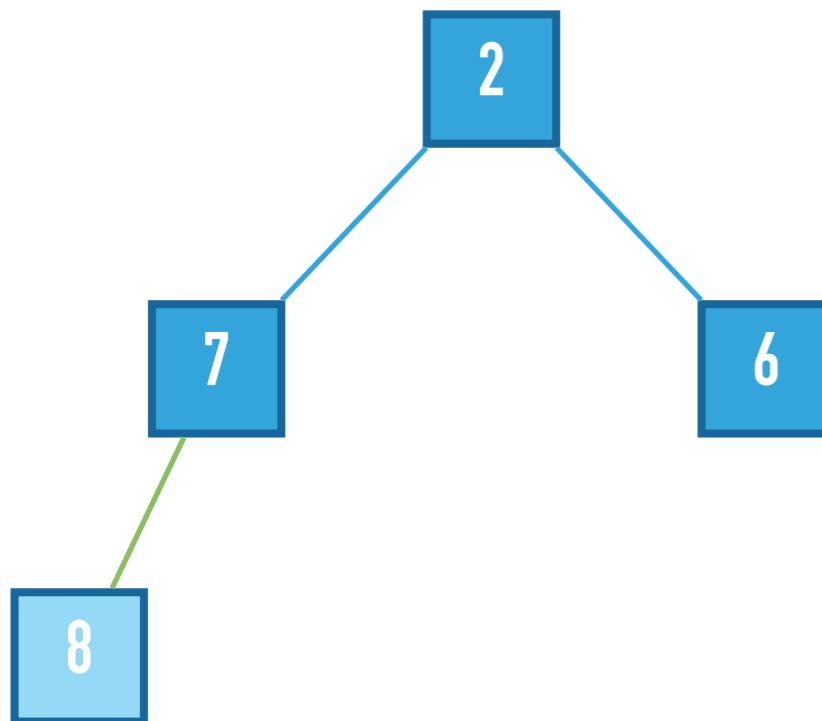
HEAPSORT: ESEMPIO



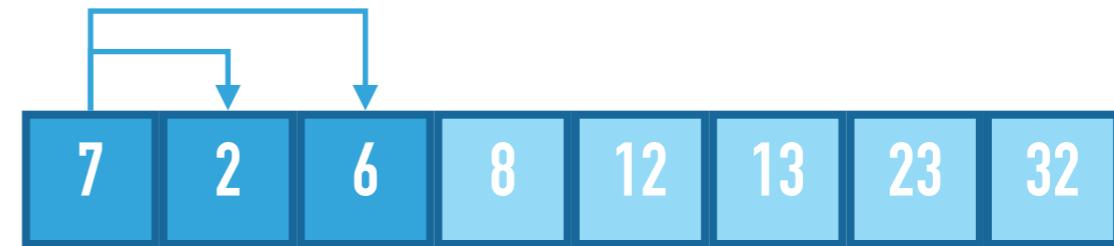
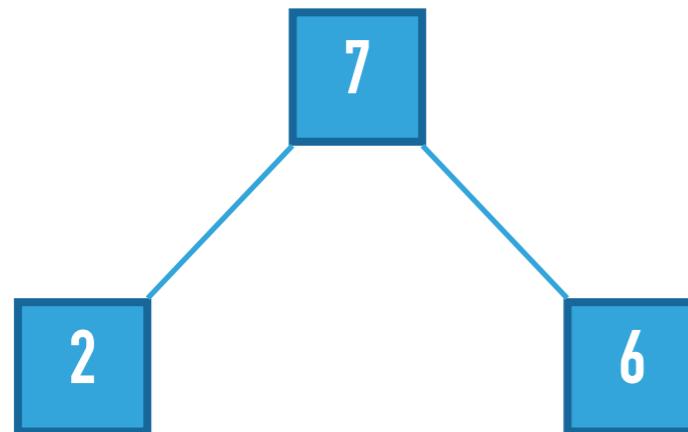
HEAPSORT: ESEMPIO



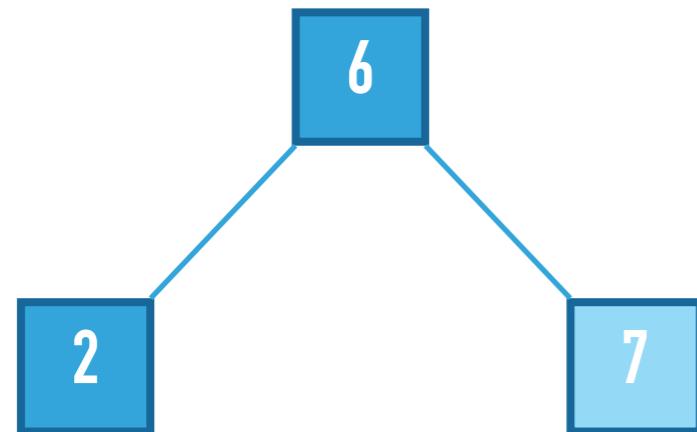
HEAPSORT: ESEMPIO



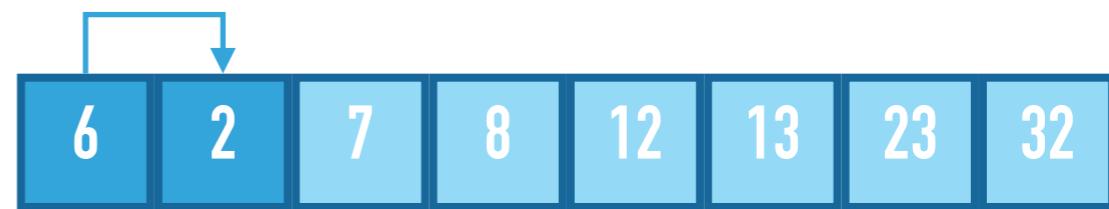
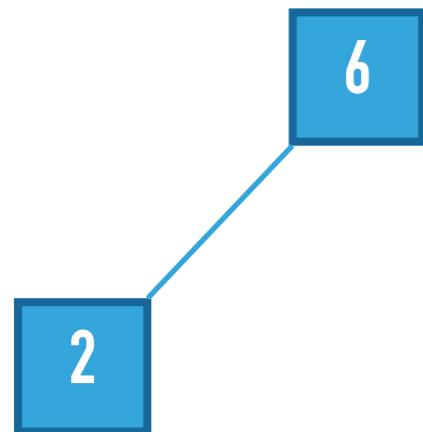
HEAPSORT: ESEMPIO



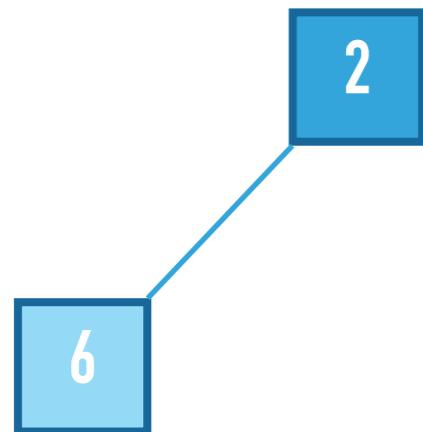
HEAPSORT: ESEMPIO



HEAPSORT: ESEMPIO



HEAPSORT: ESEMPIO



HEAPSORT: ESEMPIO

2

2 6 7 8 12 13 23 32

6 7 8 12 13 23 32

HEAPSORT: ESEMPIO



ORDINAMENTO
QUICKSORT

ALGORITMI E STRUTTURE DATI

QUICKSORT

- ▶ Abbiamo visto due algoritmi di ordinamento per comparazione che sono ottimali in termini di tempo: $\Theta(n \log n)$
- ▶ Ora vedremo un algoritmo che nel caso peggiore richiede tempo quadratico...
- ▶ ...ma nel caso medio richiede tempo $O(n \log n)$
- ▶ Domanda: perché potrebbe avere senso studiare questo algoritmo?

QUICKSORT: STORIA

- ▶ Ideato da Tony Hoare (vincitore del premio Turing nel 1980) nel 1959-60
- ▶ Quando ben implementato il Quicksort è, nella pratica, più veloce di mergesort e heapsort
- ▶ Questo nonostante abbia un caso peggiore quadratico...
- ▶ ... perché il caso **medio** è $O(n \log n)$



QUICKSORT: IDEA DI BASE

- ▶ Il quick sort è un algoritmo “divide et impera”
- ▶ L’idea di base è quella di scegliere in un array di n elementi un **pivot**, spostare gli elementi più piccoli prima del pivot e quelli più grandi dopo il pivot.
- ▶ Se applichiamo ricorsivamente lo stesso algoritmo ai due sotto-array risultanti (elementi minori e maggiori del pivot) otteniamo un array ordinato

PROCEDURA DI PARTIZIONAMENTO



Dobbiamo partizionare
l'array in base al pivot

Pivot



Il Pivot adesso è già nella
posizione corretta!

Tutti gli elementi minori lo precedono
e quelli maggiori lo seguono

Ora dobbiamo fare la stessa operazione sui due sottoarray

PROCEDURA DI PARTIZIONAMENTO

- ▶ Dobbiamo definire la procedura di partizionamento in modo efficiente
- ▶ Ne esistono diverse, noi vediamo lo schema di partizionamento di Hoare
- ▶ Idea di base: teniamo due indici:
 - ▶ i indica l'ultimo degli elementi minori del pivot
 - ▶ j viene utilizzato per scorrere l'array

PROCEDURA DI PARTIZIONAMENTO



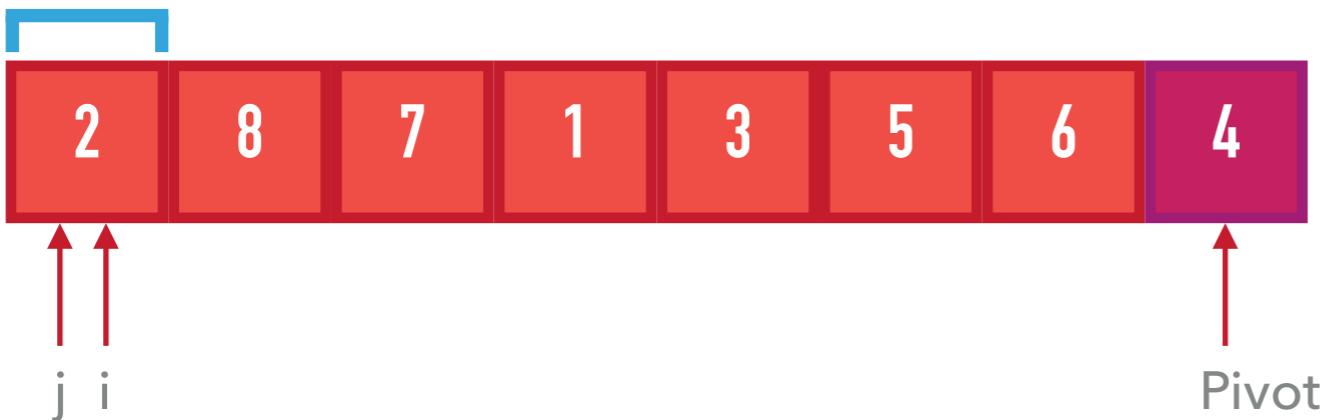
PROCEDURA DI PARTIZIONAMENTO



Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO

Elementi minori del pivot

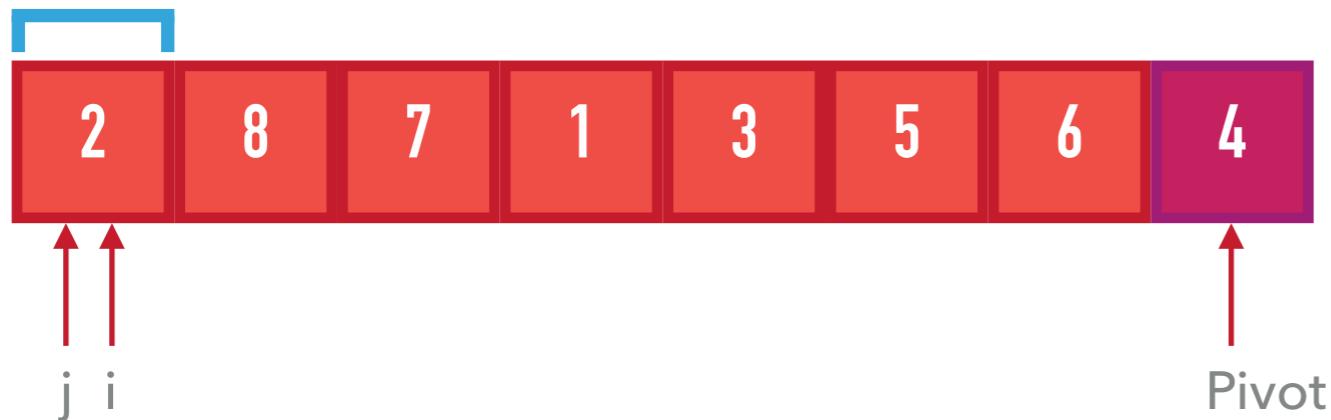


Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

In questo caso non cambia nulla (i è uguale a j)

PROCEDURA DI PARTIZIONAMENTO

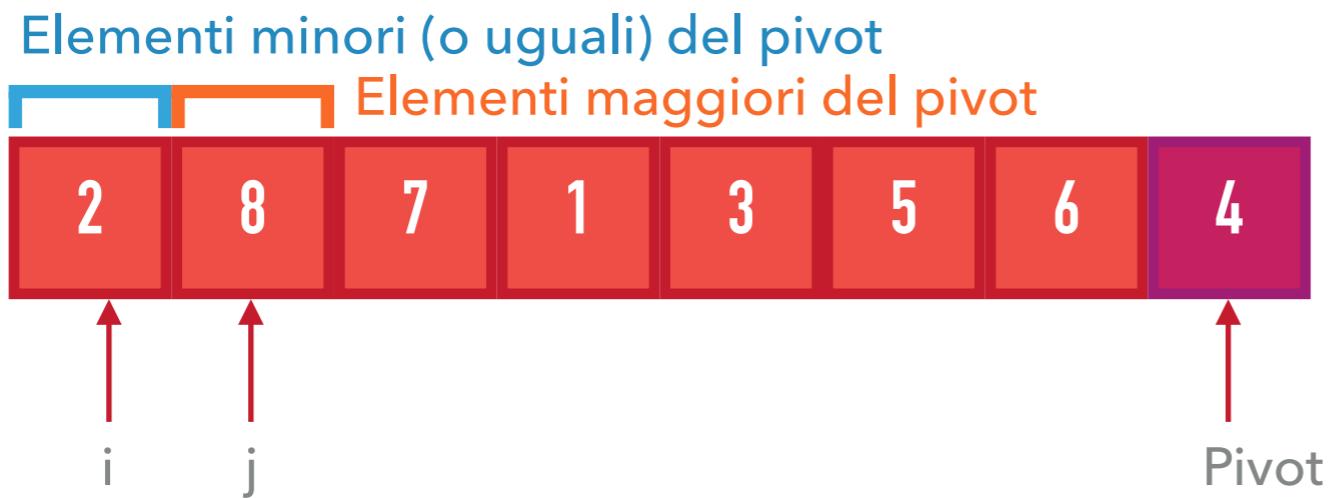
Elementi minori (o uguali) del pivot



Se $A[j]$ è minore (o uguale) del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

In questo caso non cambia nulla (i è uguale a j)

PROCEDURA DI PARTIZIONAMENTO



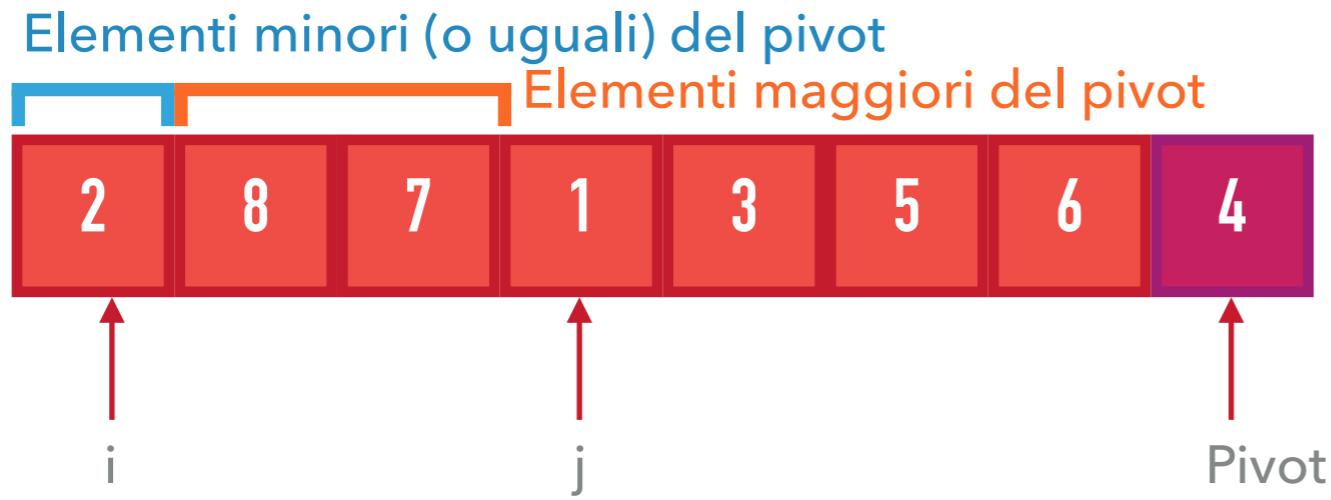
Se $A[j]$ è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



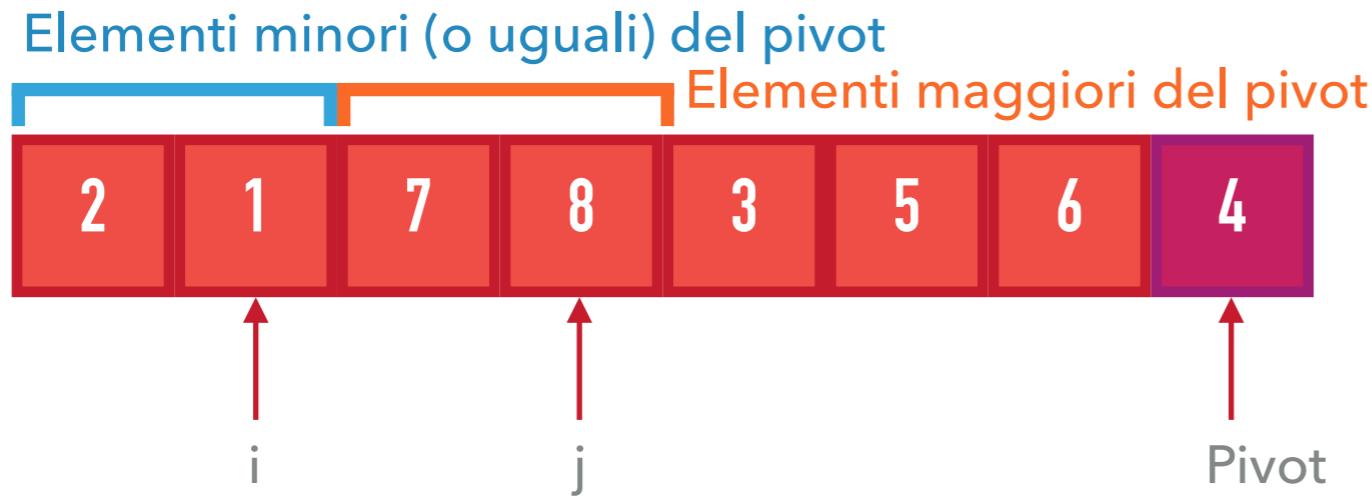
Se $A[j]$ è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



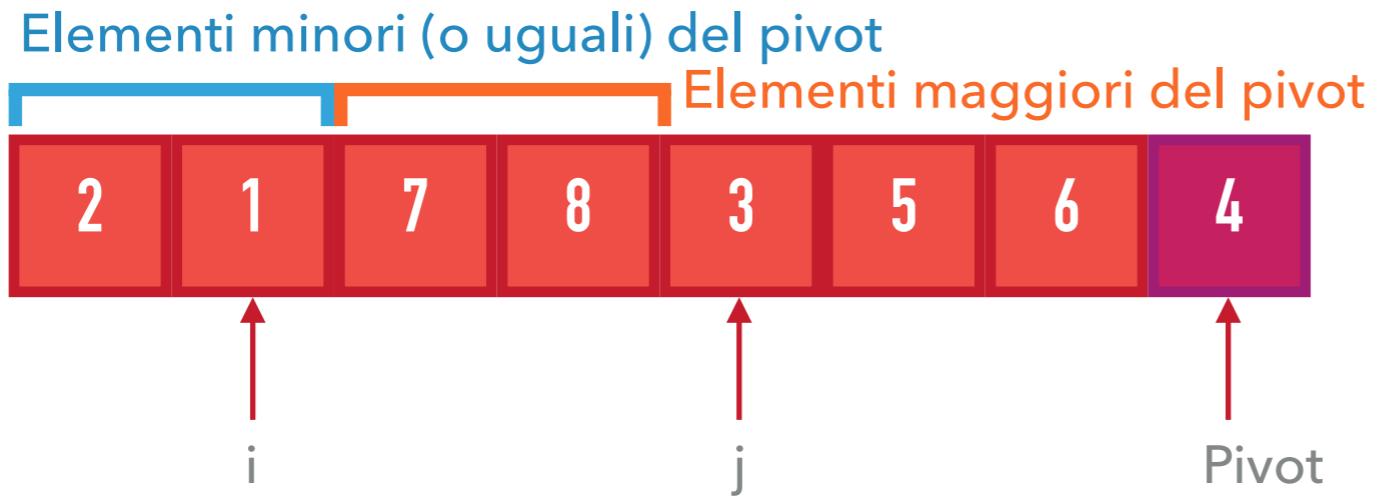
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



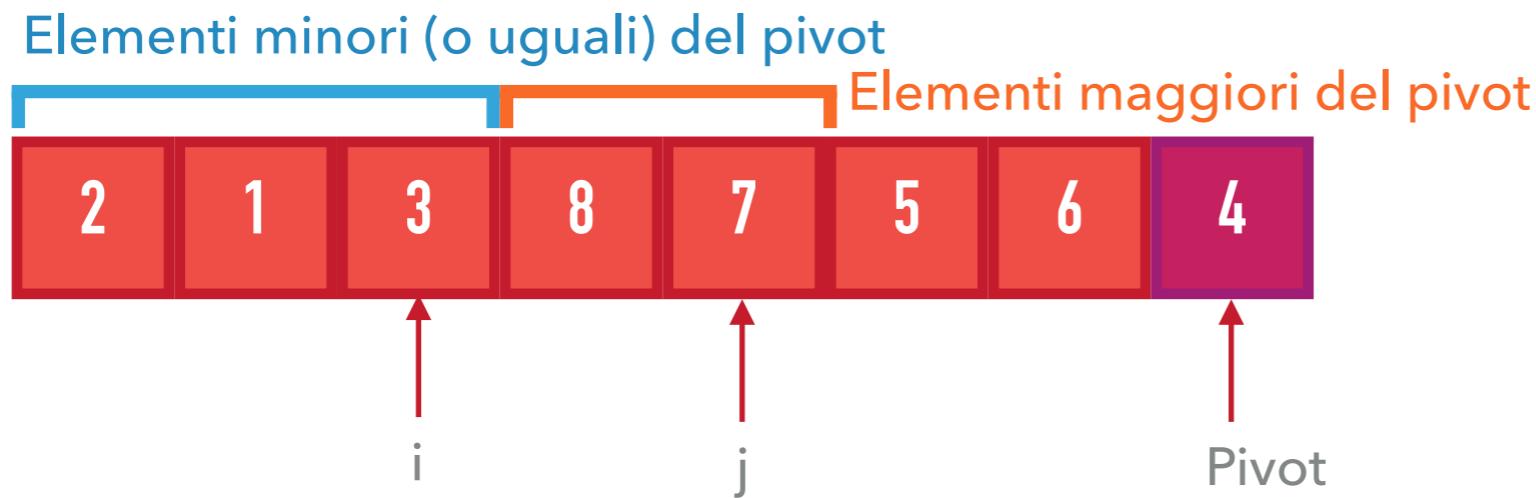
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



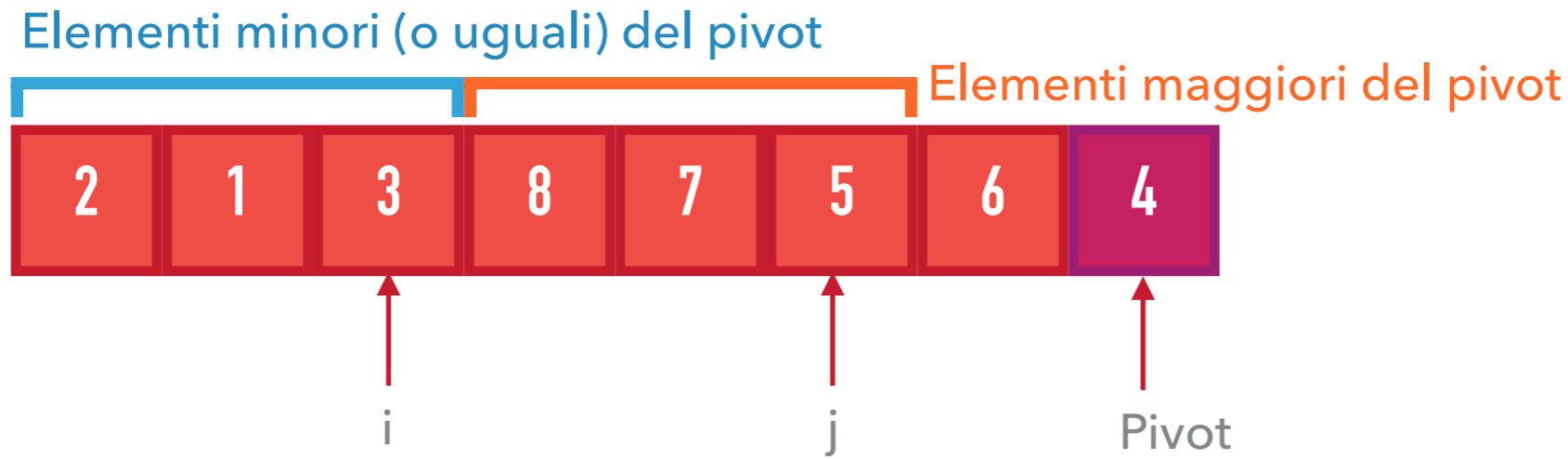
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



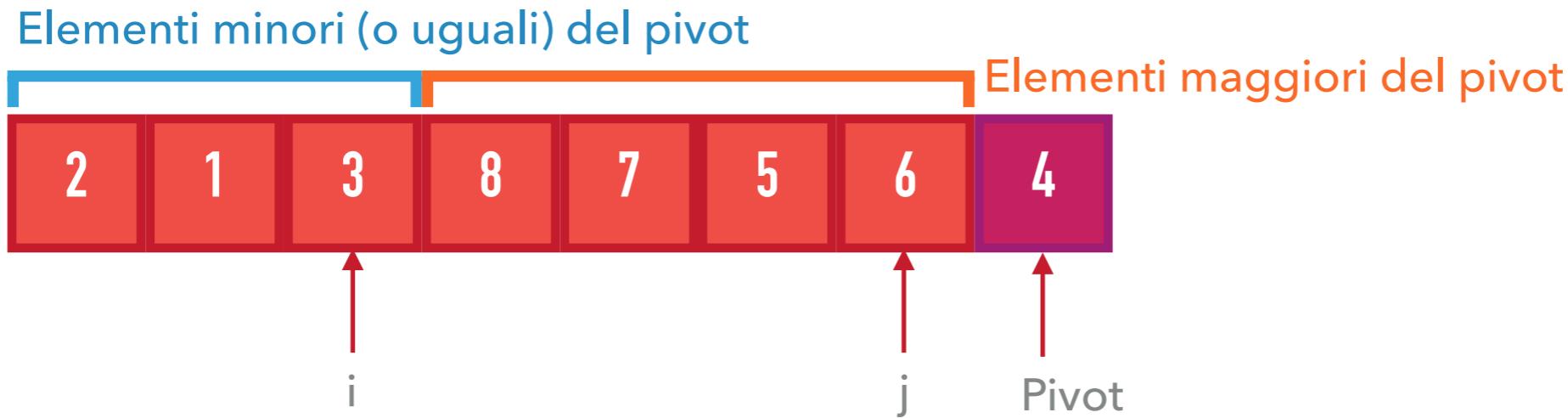
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



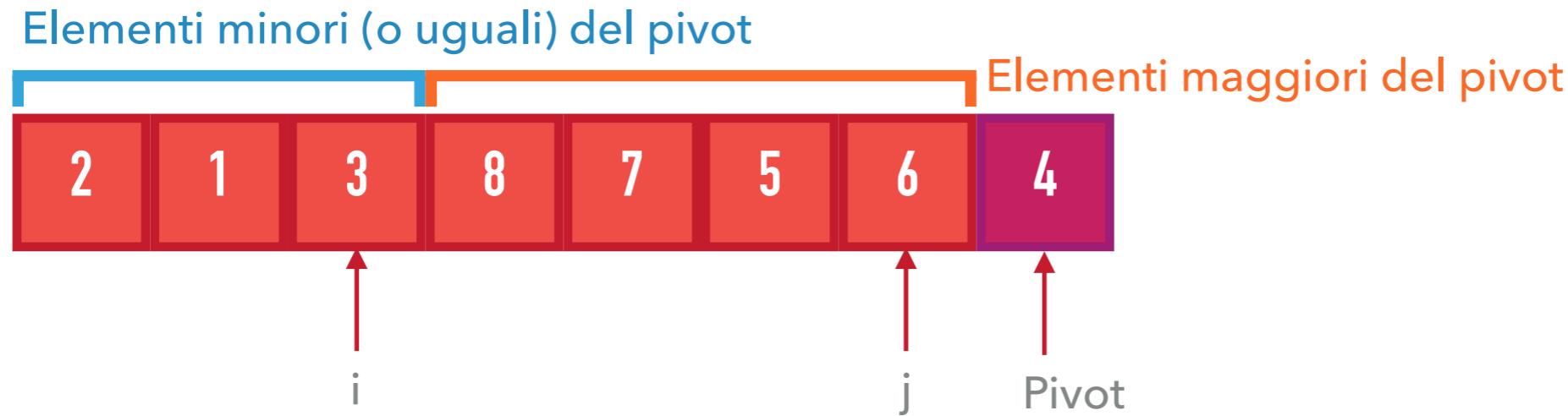
Se $A[j]$ è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



Se $A[j]$ è maggiore del pivot
non facciamo nulla

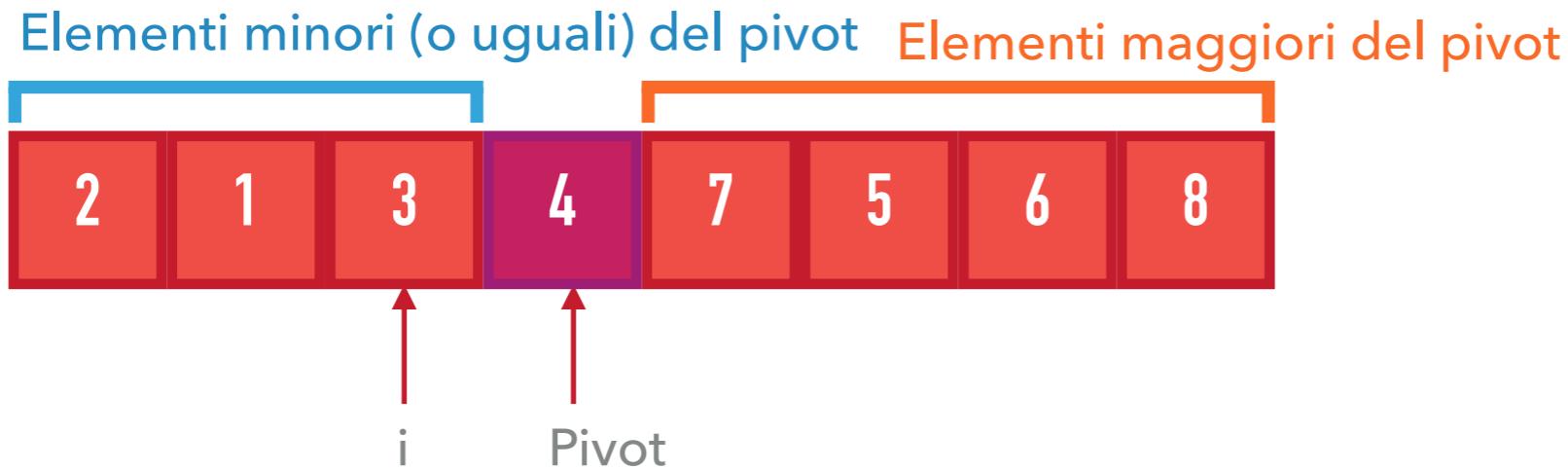
PROCEDURA DI PARTIZIONAMENTO



Abbiamo effettuato una scansione di tutto l'array e abbiamo raccolto tutti gli elementi minori o uguali del pivot negli indici $[0, i]$.

Dobbiamo solo posizionale il pivot tra le due partizioni

PROCEDURA DI PARTIZIONAMENTO



Ci basta scambiare l'elemento in posizione $i + 1$ (che è necessariamente maggiore del pivot o il pivot stesso) con il pivot

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
i = -1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
    if A[j] <= x # se troviamo un elemento minore del pivot...
        i = i + 1
        scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array
Scambia A[i+1] con A[n-1] # mettiamo il pivot nella sua posizione finale
return i+1
```

Ma a noi servirà partizionare segmenti arbitrari di un array, quindi possiamo usare un'altra versione della procedura di partizionamento che lo applica solo tra due indici

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)

i = p-1 # posizione iniziale degli elementi minori del pivot

for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot

if A[j] ≤ x # se troviamo un elemento minore del pivot...

 i = i + 1

 scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array

Scambia A[i+1] con A[r] # mettiamo il pivot nella sua posizione finale

return i+1

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

- ▶ Invariante (condizione che rimane vera ad ogni ciclo):
 - ▶ Gli elementi tra l'inizio dell'array e i sono tutti minori o uguali del pivot
 - ▶ Gli elementi tra $i + 1$ e j sono tutti maggiori del pivot
- ▶ Ogni iterazione continua a far rispettare questa condizione:
 - ▶ Se il nuovo elemento è maggiore del pivot viene mantenuto nella sua posizione, rispettando quindi la condizione
 - ▶ Se il nuovo elemento è minore o uguale del pivot, i viene incrementato l'elemento in posizione j viene scambiato con quello in posizione i (che, dato che i è stato incrementato, è maggiore del pivot)

PARTIZIONAMENTO: COMPLESSITÀ

- ▶ Il partizionamento viene fatto con una singola “passata” dell’array (il ciclo for esterno)
- ▶ Tutte le operazioni all’interno e all’esterno del ciclo for hanno un costo costante
- ▶ Ne segue che il partizionamento viene fatto in tempo $\Theta(n)$

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

```
if p ≥ r:  
    return  
q = partiziona(A, p, r) # indice del pivot dopo il partizionamento  
quicksort(A, p, q-1) # chiamata ricorsiva sugli elementi minori o uguali  
quicksort(A, q+1, r) # chiamata ricorsiva sugli elementi maggiori
```

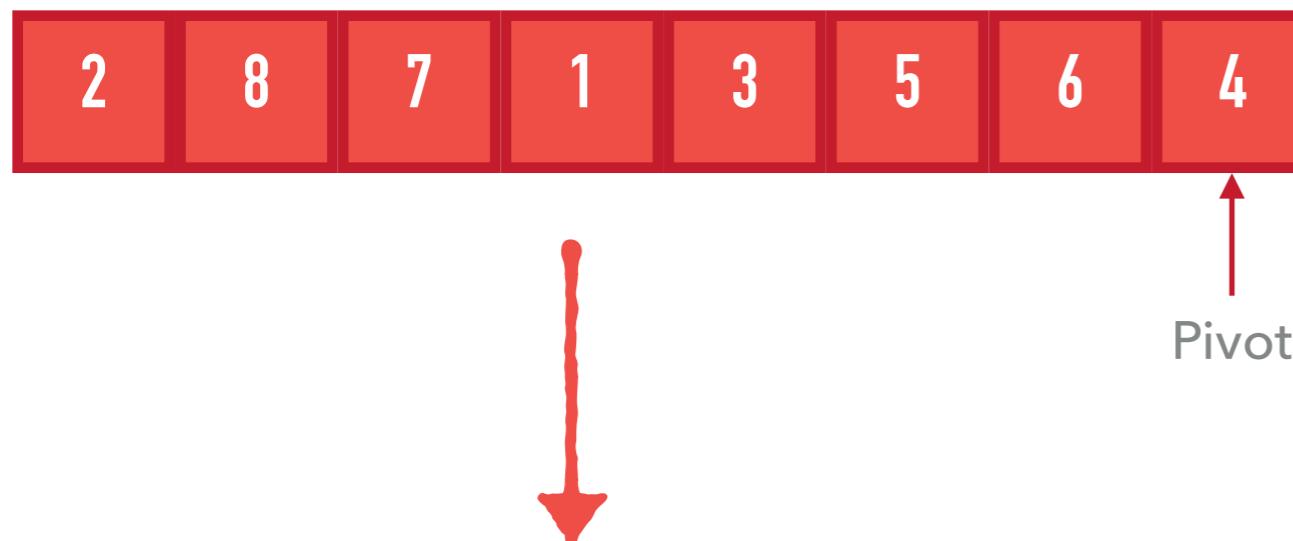
QUICKSORT: PERCHÉ FUNZIONA?

- ▶ Per un array di dimensione 0 o 1 il quicksort funziona per motivi banali
- ▶ Supponiamo di avere provato che il quicksort funziona per ogni dimensione minore di n , vediamo che funziona per la dimensione n
- ▶ Dopo la procedura di partizionamento il pivot è nella posizione corretta e otteniamo due sotto-array uno che precede il pivot con tutti gli elementi minori o uguali al pivot e uno con tutti gli elementi maggiori
- ▶ Richiamiamo quicksort sui due sotto-array di dimensione minore di n , che per ipotesi ci restituiranno gli array ordinati
- ▶ Otteniamo tutti gli elementi minori o uguali al pivot ordinati, seguiti dal pivot e da tutti gli elementi maggiori del pivot ordinati. Quindi l'array di n elementi è ordinato.

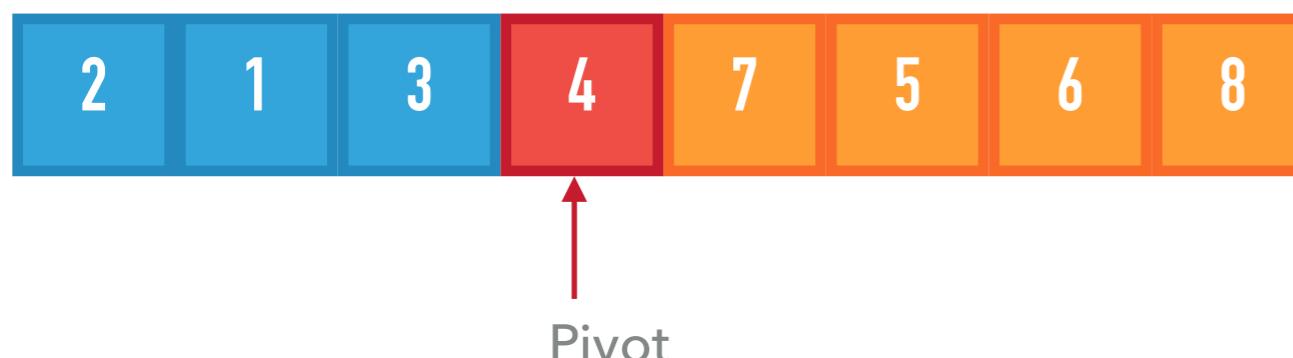
ANALISI DELLA COMPLESSITÀ

- ▶ L'analisi della complessità del quicksort è più delicata di mergesort e heapsort
- ▶ La dimensione degli array nelle chiamate ricorsive dipende dalla procedura di partizionamento
- ▶ La procedura di partizionamento dipende, a sua volta dai dati che abbiamo

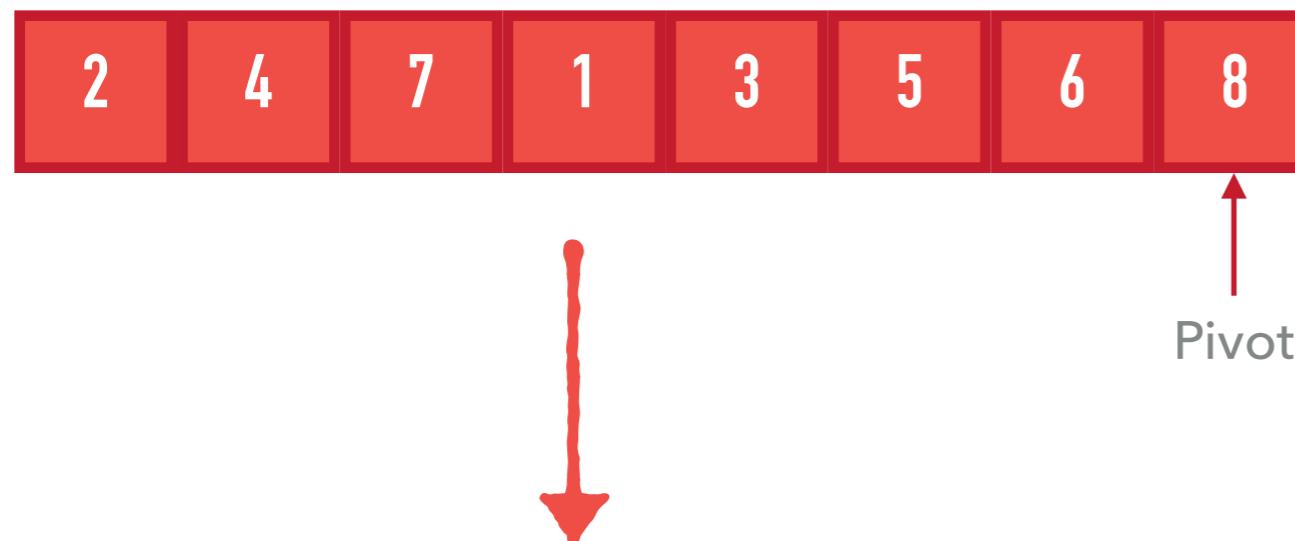
POSSIBILI PARTIZIONAMENTI



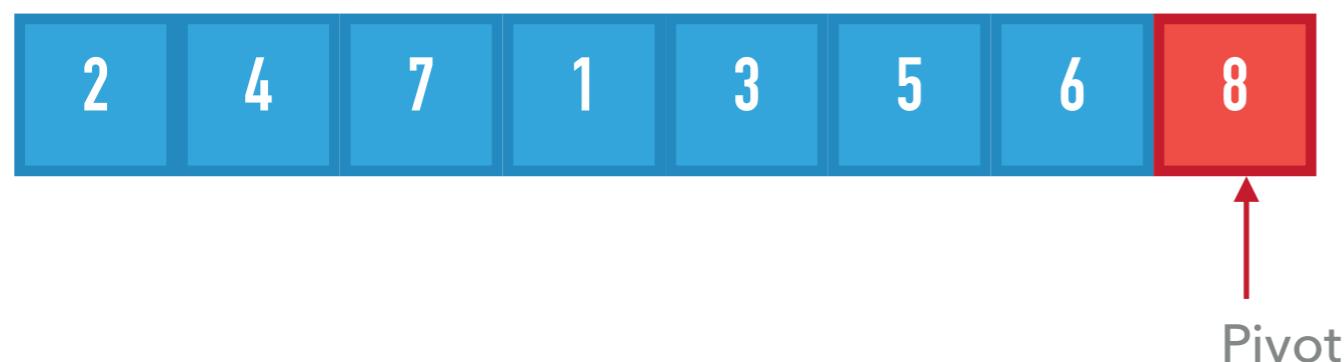
Un “buon” partizionamento:
i due sottoarray risultanti sono
ognuno circa la metà dell’array di
partenza



POSSIBILI PARTIZIONAMENTI



Un “cattivo” partizionamento:
uno dei sotto-array risultanti
contiene tutti gli elementi tranne
uno e l’altro è vuoto!



QUIZ

In **quali** dei seguenti casi il partizionamento è maggiormente sbilanciato?

1) [1, 2, 3, 4, 5]

2) [2, 6, 5, 3, 4]

3) [1, 2, 5, 4, 3]

4) [5, 4, 3, 2, 1]

“BUON PARTIZIONAMENTO”: ANALISI

- ▶ Assumiamo che ogni partizionamento crei due sottoarray di dimensioni approssimativamente $n/2$
- ▶ L'equazione di ricorrenza diventa quindi:
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
- ▶ Per il teorema dell'esperto il tempo di calcolo del quicksort è $\Theta(n \log n)$
- ▶ Ma questo risultato vale solo per un “buon” partizionamento

“CATTIVO PARTIZIONAMENTO”: ANALISI

- ▶ Assumiamo che ogni partizionamento crei un sottoarray vuoto ed uno di dimensione $n - 1$
- ▶ L'equazione di ricorrenza diventa quindi:
$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$$
- ▶ Se espandiamo $T(n)$ vediamo che abbiamo n “passi ricorsivi”, ognuno dei quali esegue un lavoro lineare rispetto alla dimensione dell'array da ordinare
- ▶ Come risultato otteniamo $\Theta(n^2)$

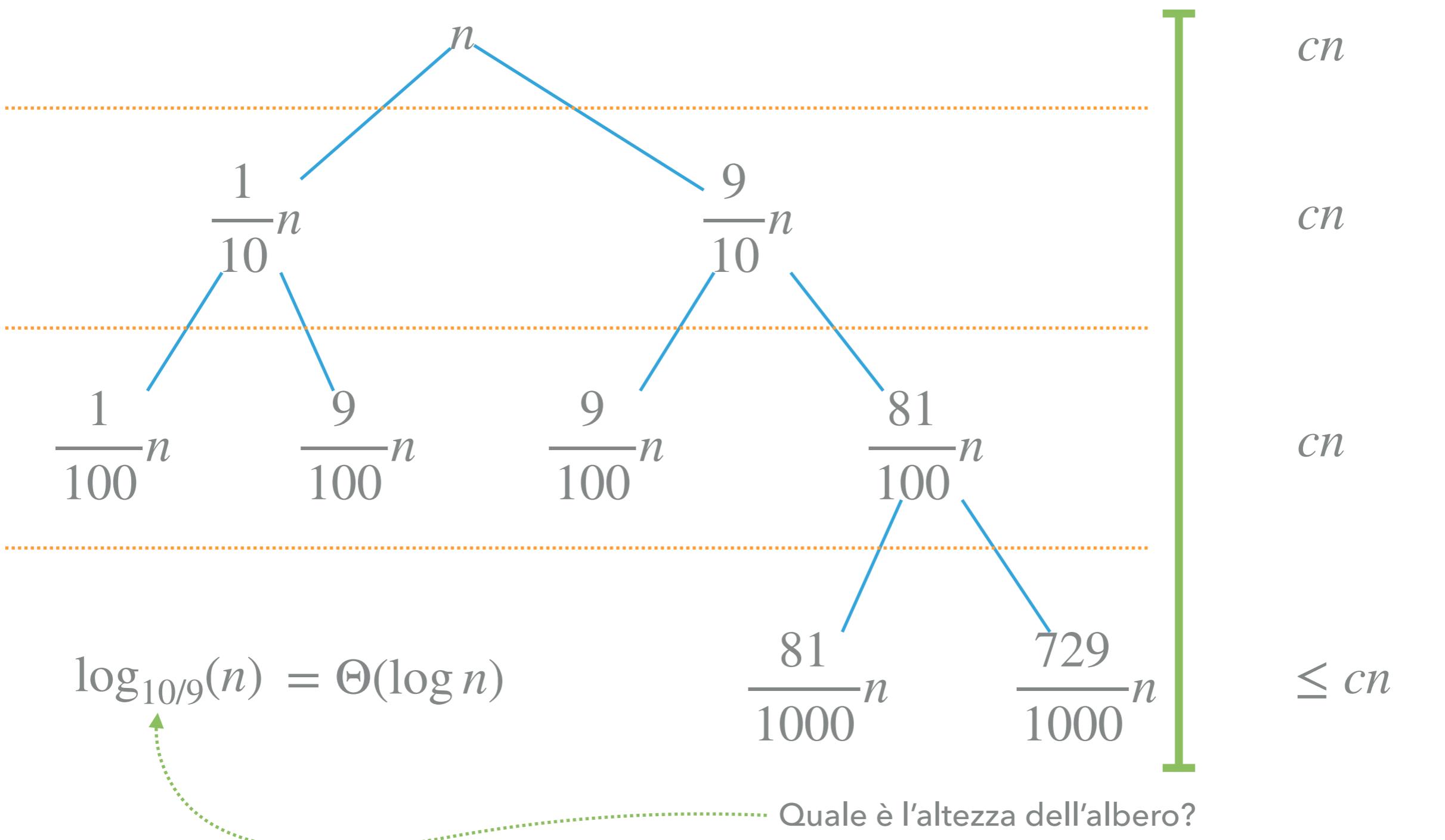
ANALISI DELLA COMPLESSITÀ

- ▶ Nel caso peggiore quindi otteniamo $\Theta(n^2)$
- ▶ Ma quanto è frequente il caso peggiore?
- ▶ Supponiamo che le nostre partizioni siano molto sbilanciate: la prima metà include $1/10$ dell'array e la seconda ne include $9/10$
- ▶ Lo stesso ragionamento vale anche per $1/100$ e $99/100$, etc.

ANALISI DELLA COMPLESSITÀ

- ▶ $T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$
- ▶ Esplicitiamo la costante nascosta c nell' $\Theta(n)$:
$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + cn$$
- ▶ Costruiamo l'albero delle chiamate ricorsive

ANALISI DELLA COMPLESSITÀ



ANALISI DELLA COMPLESSITÀ

- ▶ Stiamo eseguendo al più cn passi per ognuno degli $\Theta(\log n)$ livelli dell'albero
- ▶ Di conseguenza il tempo di esecuzione è ancora $\Theta(n \log n)$
- ▶ Cosa succede nel “caso medio”?
- ▶ È possibile provare che, se la scelta del pivot viene effettuata in modo casuale, il **tempo atteso** di esecuzione è $\Theta(n \log n)$

QUICKSORT: POSSIBILI VARIANTI

- ▶ Esistono molte varianti del quicksort per minimizzare il rischio di cadere nel caso peggiore
- ▶ Invece di scegliere l'ultimo elemento come pivot, viene scelto un elemento a caso che viene spostato in ultima posizione (Randomized-Quicksort)
- ▶ Vengono presi tre elementi e la mediana dei tre viene usata come pivot
- ▶ I libri “algorithms in C”, “algorithms in Java” di Robert Sedgewick trattano molti di questi miglioramenti

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Come sono distribuiti gli input? Consideriamo array A di lunghezza n , con elementi diversi tra loro, e assumiamo ogni permutazione sia equiprobabile.
- ▶ L'analisi che faremo vale uguale anche per randomized quicksort.
- ▶ Il **rango** di un elemento $a \in A$ è la sua posizione nell'array A ordinato.
- ▶ Indichiamo con z_i l'elemento di A di rango i , quindi $Z = [z_1, \dots, z_n]$ è l'array ordinato.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Cosa domina il costo di quicksort?
- ▶ Il costo è uguale al numero totale di confronti del tipo $A[j] \leq pivot$ in **tutte** le chiamate di partition. Chiamo X la **variabile aleatoria** che conta il numero totale di confronti in partition.
- ▶ Ogni coppia z_i, z_j di elementi verrà confrontata al più una volta.
Perchè?
 - ▶ perchè confronto solo con il pivot, che a fine partition non viene più toccato.
 - ▶ Sia $X_{i,j}$ una v.a. che vale 1 se confronto z_i con z_j e 0 altrimenti.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Possiamo scrivere:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}$$

- ▶ Ci interessa il numero atteso di confronti:

$$\mathbb{E}[X] = \mathbb{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{i,j}]$$

- ▶ Vale anche $\mathbb{E}[X_{i,j}] = \Pr\{X_{i,j} = 1\}$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Qual è la probabilità che z_i sia confrontato con z_j ?
- ▶ Sia $Z_{i,j} = [z_i, z_{i+1}, \dots, z_j]$, e sia z_k il primo elemento di $Z_{i,j}$ ad essere scelto come pivot.
- ▶ Se $z_k = z_i \circ z_k = z_j$, allora $X_{i,j} = 1$ altrimenti $X_{i,j} = 0$. Why?
- ▶ Se scelgo un $z_k, k \neq i, j$, allora z_i e z_j finiscono in due call ricorsive diverse, e non possono più essere confrontati.
- ▶ Per ipotesi sulla distribuzione:

$$\Pr\{z_k \text{ è il primo pivot in } Z_{i,j}\} = \frac{1}{j-i+1}$$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ $Pr\{X_{i,j} = 1\} = \frac{2}{j - i + 1}$
- ▶ $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} < \sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w}$ e
- ▶ $\sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$
- ▶ La complessità nel caso medio è $O(n \log n)$!

ORDINAMENTO
QUICKSORT

ALGORITMI E STRUTTURE DATI

QUICKSORT

QUICKSORT

QUICKSORT

- ▶ Abbiamo visto due algoritmi di ordinamento per comparazione che sono ottimali in termini di tempo: $\Theta(n \log n)$

QUICKSORT

- ▶ Abbiamo visto due algoritmi di ordinamento per comparazione che sono ottimali in termini di tempo: $\Theta(n \log n)$
- ▶ Ora vedremo un algoritmo che nel caso peggiore richiede tempo quadratico...
- ▶ ...ma nel caso medio richiede tempo $O(n \log n)$

QUICKSORT

- ▶ Abbiamo visto due algoritmi di ordinamento per comparazione che sono ottimali in termini di tempo: $\Theta(n \log n)$
- ▶ Ora vedremo un algoritmo che nel caso peggiore richiede tempo quadratico...
- ▶ ...ma nel caso medio richiede tempo $O(n \log n)$
- ▶ Domanda: perché potrebbe avere senso studiare questo algoritmo?

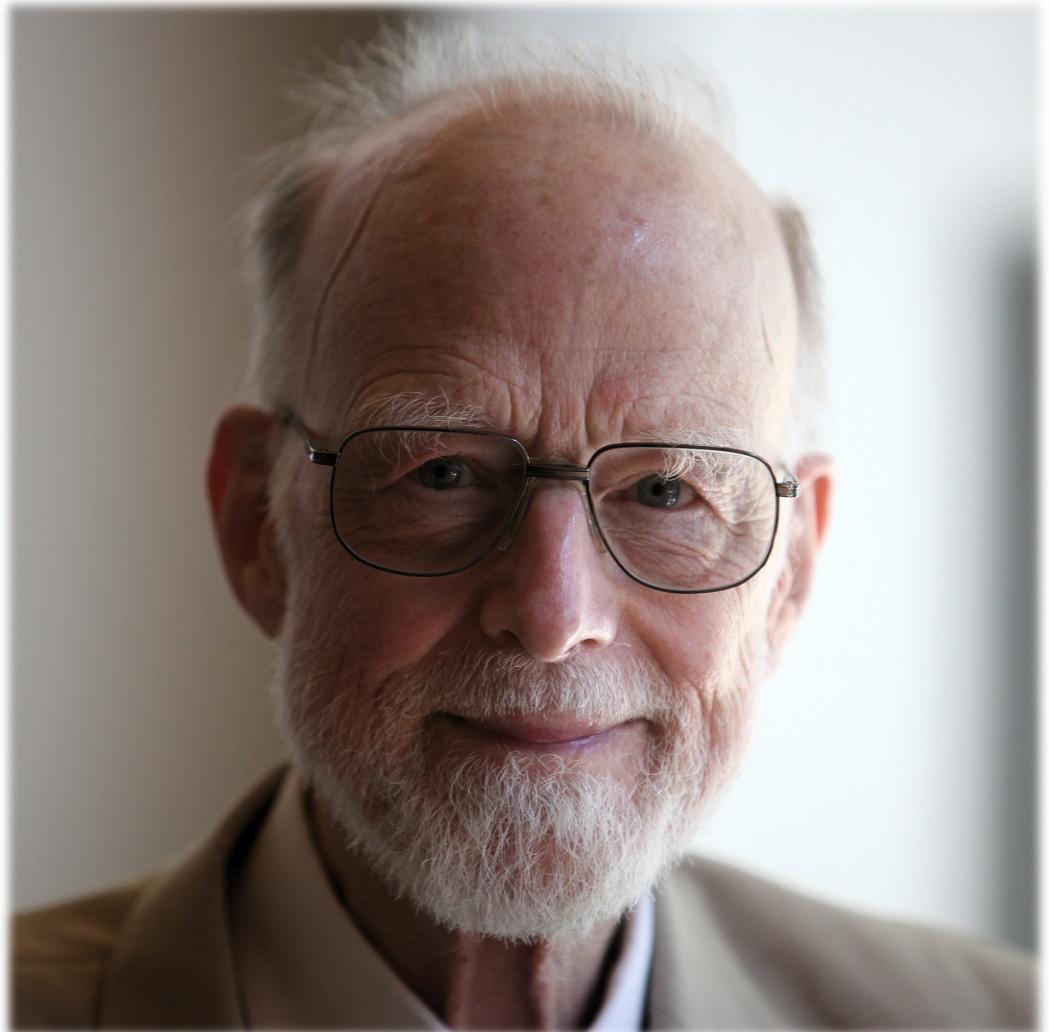
QUICKSORT

QUICKSORT: STORIA



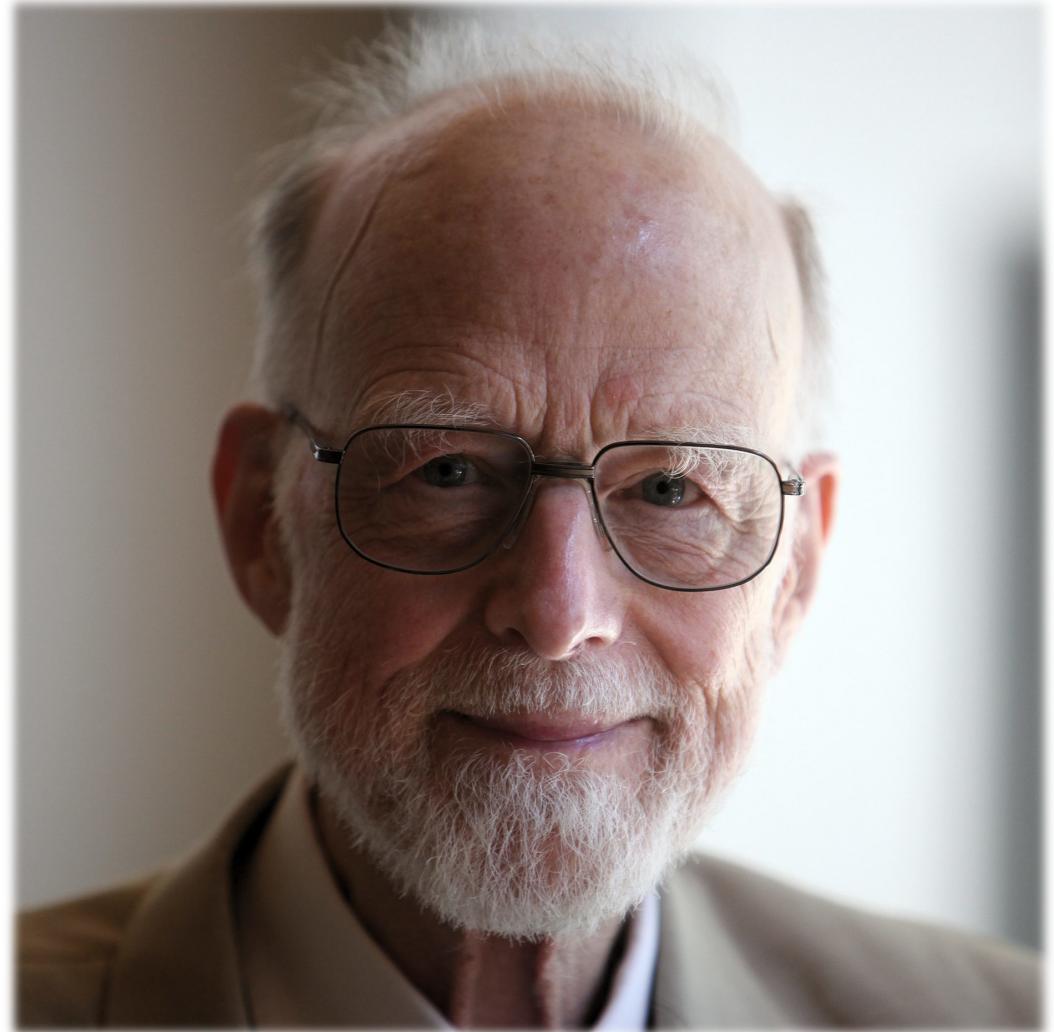
QUICKSORT: STORIA

- ▶ Ideato da Tony Hoare (vincitore del premio Turing nel 1980) nel 1959-60



QUICKSORT: STORIA

- ▶ Ideato da Tony Hoare (vincitore del premio Turing nel 1980) nel 1959-60
- ▶ Quando ben implementato il Quicksort è, nella pratica, più veloce di mergesort e heapsort
- ▶ Questo nonostante abbia un caso peggiore quadratico...



QUICKSORT: STORIA

- ▶ Ideato da Tony Hoare (vincitore del premio Turing nel 1980) nel 1959-60
- ▶ Quando ben implementato il Quicksort è, nella pratica, più veloce di mergesort e heapsort
- ▶ Questo nonostante abbia un caso peggiore quadratico...
- ▶ ... perché il caso **medio** è $O(n \log n)$



QUICKSORT: IDEA DI BASE

QUICKSORT: IDEA DI BASE

- ▶ Il quick sort è un algoritmo “divide et impera”

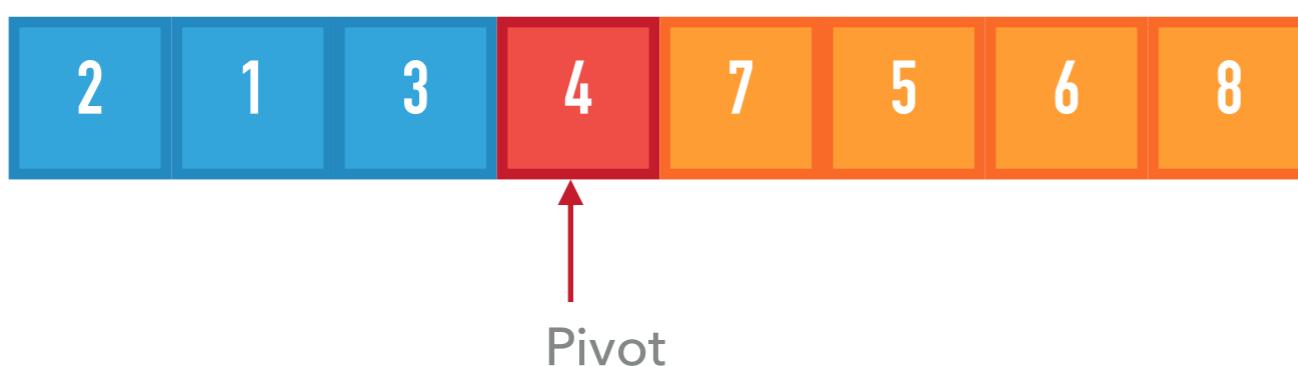
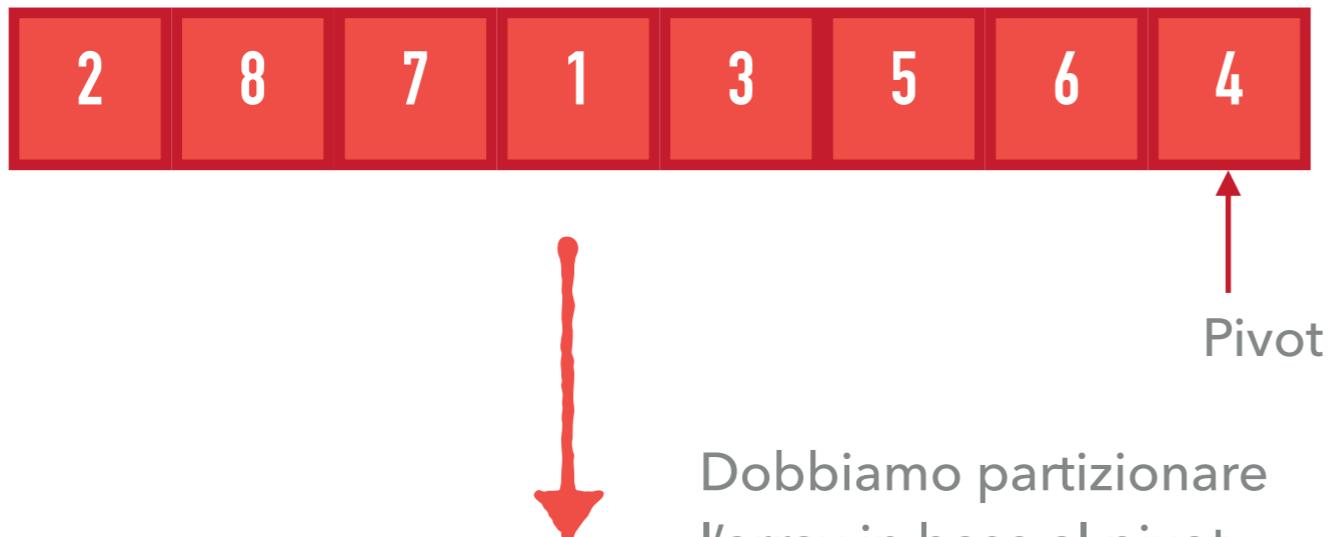
QUICKSORT: IDEA DI BASE

- ▶ Il quick sort è un algoritmo “divide et impera”
- ▶ L’idea di base è quella di scegliere in un array di n elementi un **pivot**, spostare gli elementi più piccoli prima del pivot e quelli più grandi dopo il pivot.

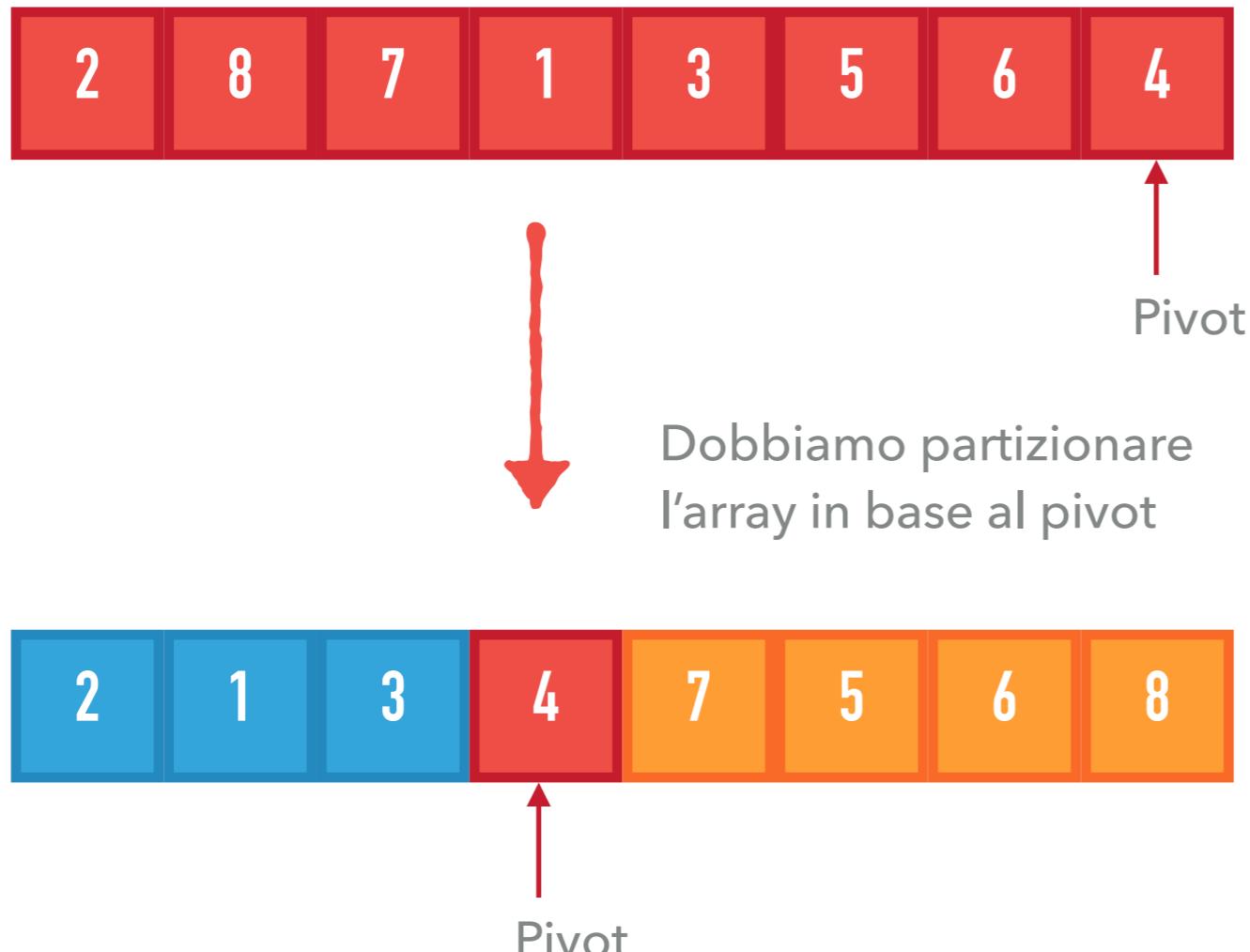
QUICKSORT: IDEA DI BASE

- ▶ Il quick sort è un algoritmo “divide et impera”
- ▶ L’idea di base è quella di scegliere in un array di n elementi un **pivot**, spostare gli elementi più piccoli prima del pivot e quelli più grandi dopo il pivot.
- ▶ Se applichiamo ricorsivamente lo stesso algoritmo ai due sotto-array risultanti (elementi minori e maggiori del pivot) otteniamo un array ordinato

PROCEDURA DI PARTIZIONAMENTO



PROCEDURA DI PARTIZIONAMENTO



Il Pivot adesso è già nella posizione corretta!

Tutti gli elementi minori lo precedono e quelli maggiori lo seguono

PROCEDURA DI PARTIZIONAMENTO



Dobbiamo partizionare
l'array in base al pivot

Pivot



Il Pivot adesso è già nella
posizione corretta!

Tutti gli elementi minori lo precedono
e quelli maggiori lo seguono

Ora dobbiamo fare la stessa operazione sui due sottoarray

PROCEDURA DI PARTIZIONAMENTO

PROCEDURA DI PARTIZIONAMENTO

- ▶ Dobbiamo definire la procedura di partizionamento in modo efficiente
- ▶ Ne esistono diverse, noi vediamo lo schema di partizionamento di Hoare

PROCEDURA DI PARTIZIONAMENTO

- ▶ Dobbiamo definire la procedura di partizionamento in modo efficiente
- ▶ Ne esistono diverse, noi vediamo lo schema di partizionamento di Hoare
- ▶ Idea di base: teniamo due indici:
 - ▶ i indica l'ultimo degli elementi minori del pivot
 - ▶ j viene utilizzato per scorrere l'array

PROCEDURA DI PARTIZIONAMENTO



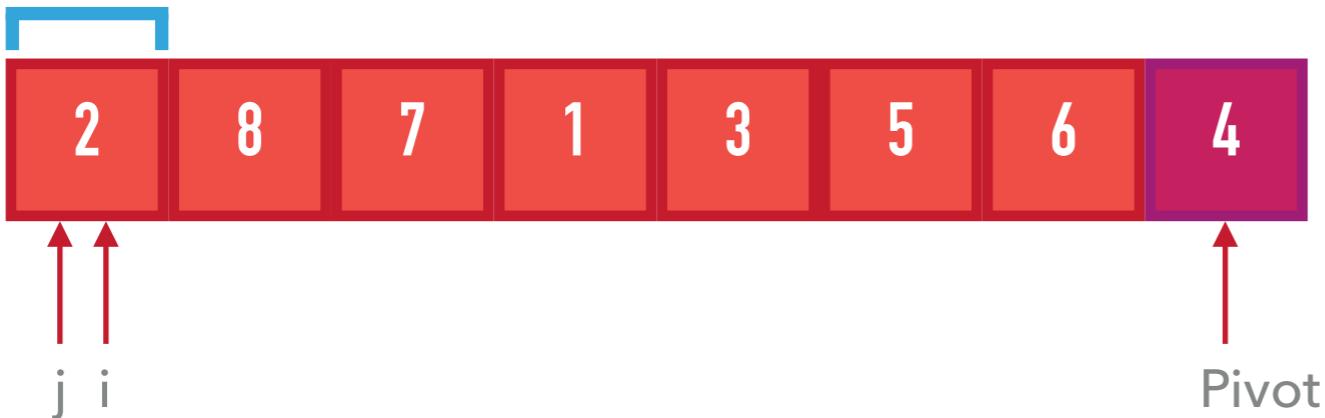
PROCEDURA DI PARTIZIONAMENTO



Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO

Elementi minori del pivot

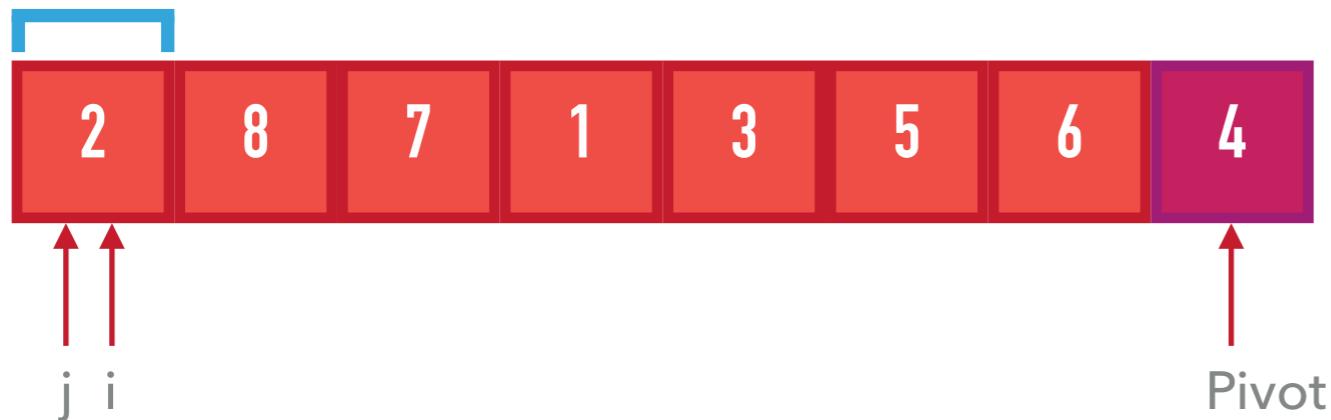


Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

In questo caso non cambia nulla (i è uguale a j)

PROCEDURA DI PARTIZIONAMENTO

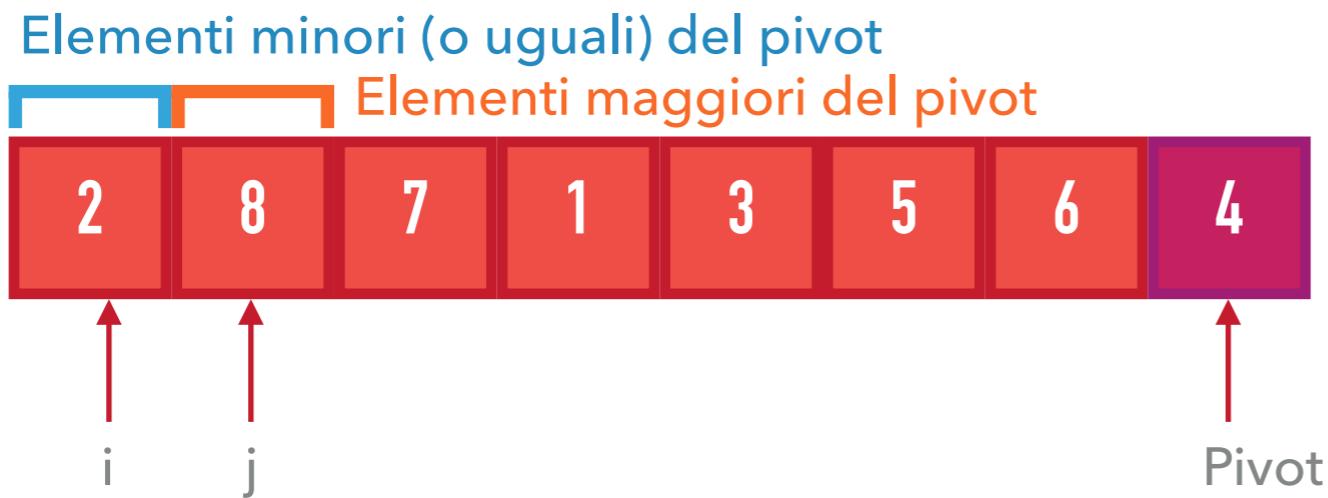
Elementi minori (o uguali) del pivot



Se $A[j]$ è minore (o uguale) del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

In questo caso non cambia nulla (i è uguale a j)

PROCEDURA DI PARTIZIONAMENTO



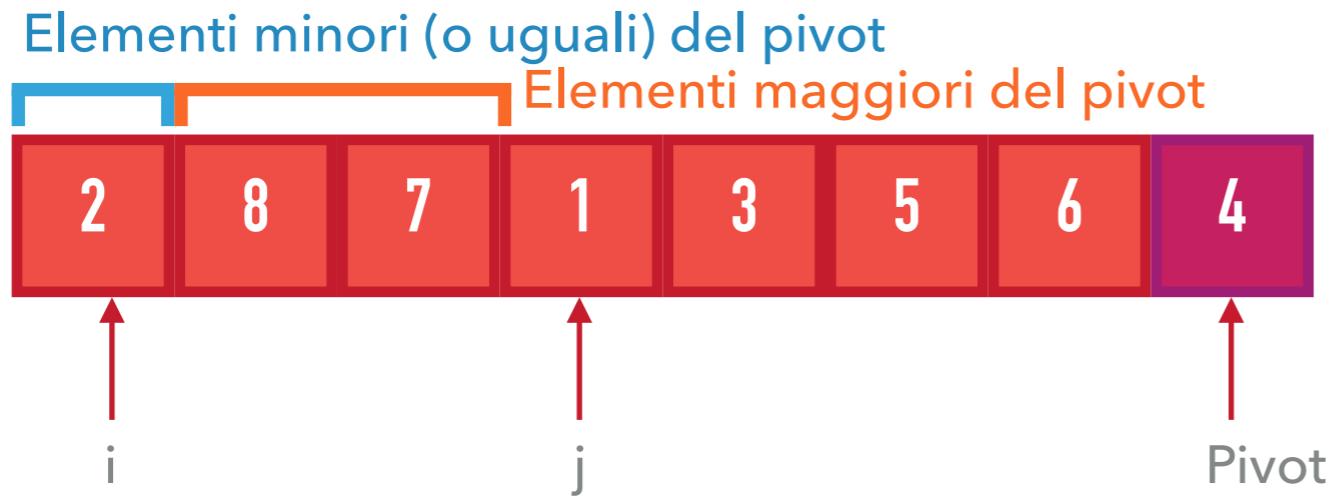
Se $A[j]$ è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



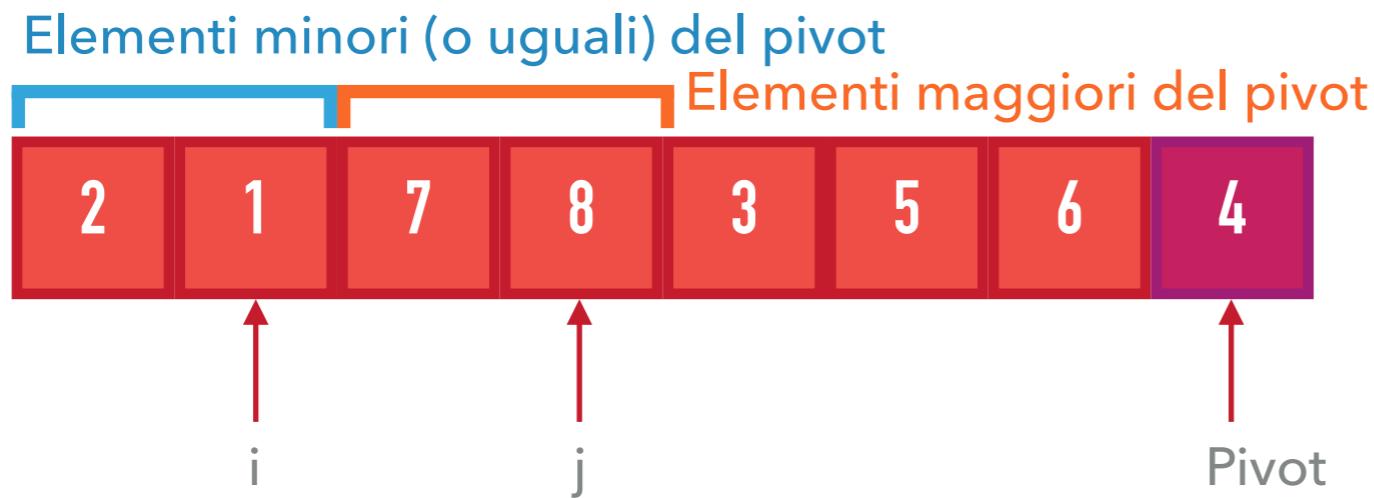
Se $A[j]$ è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



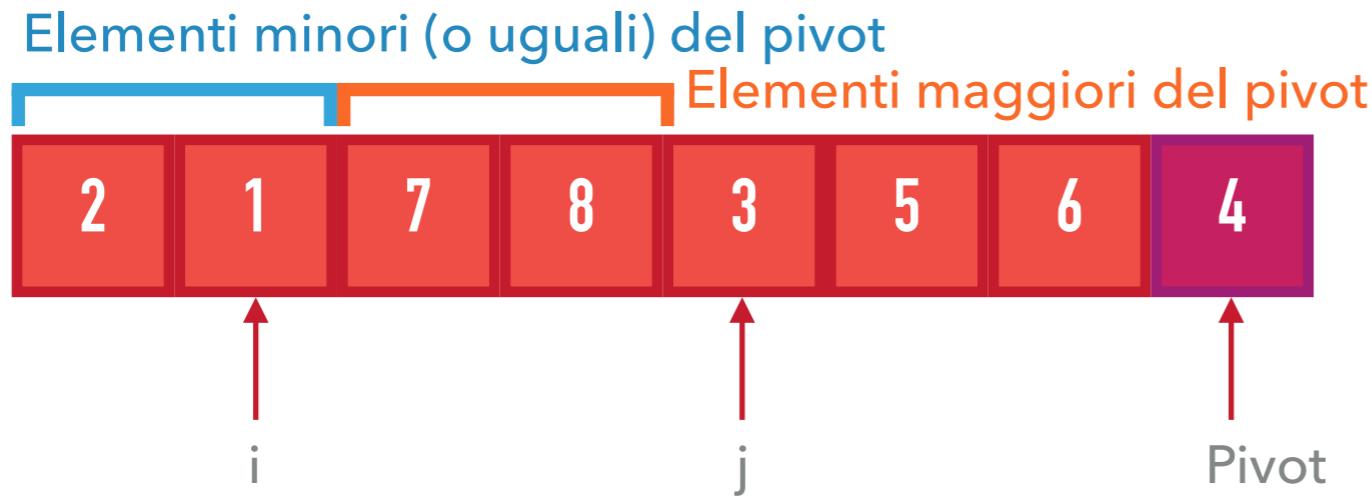
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



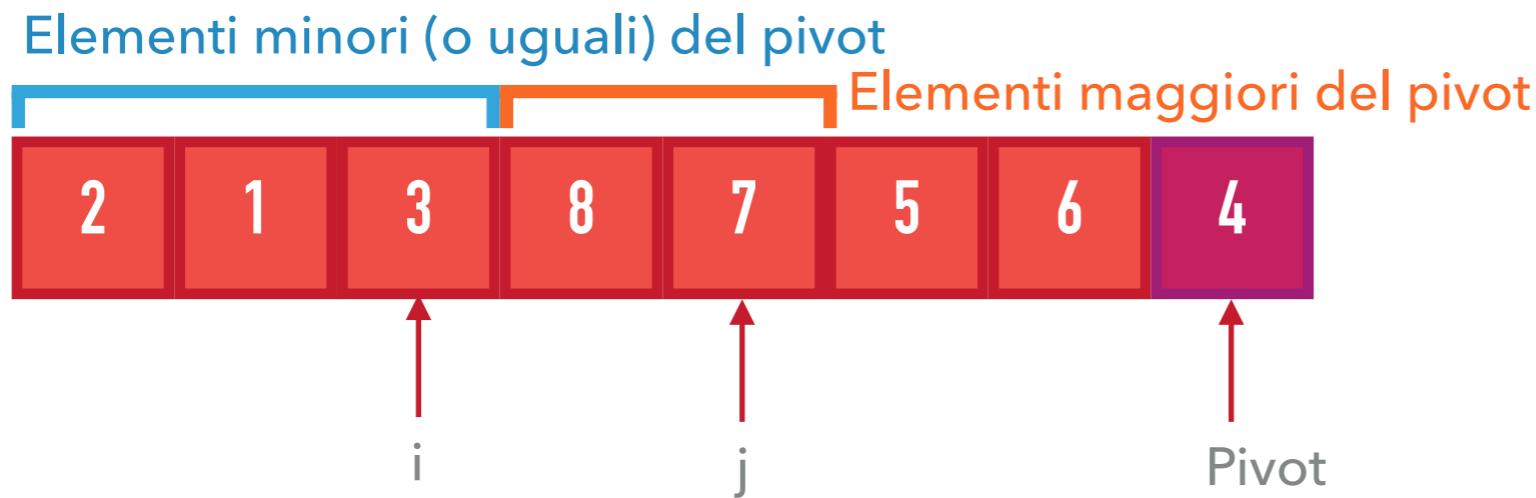
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



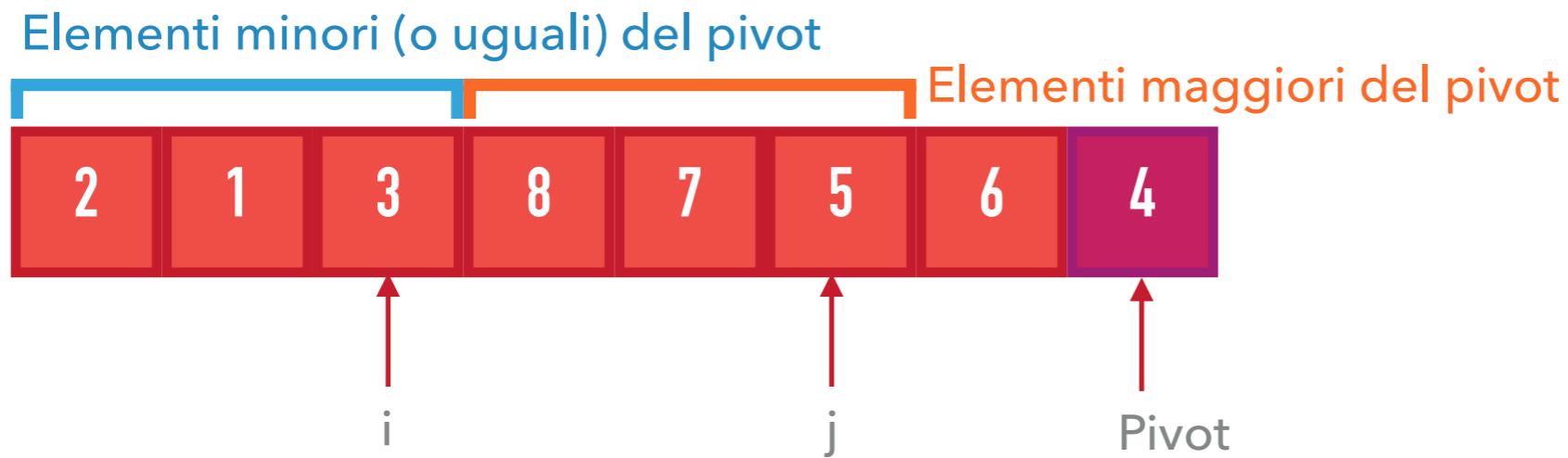
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



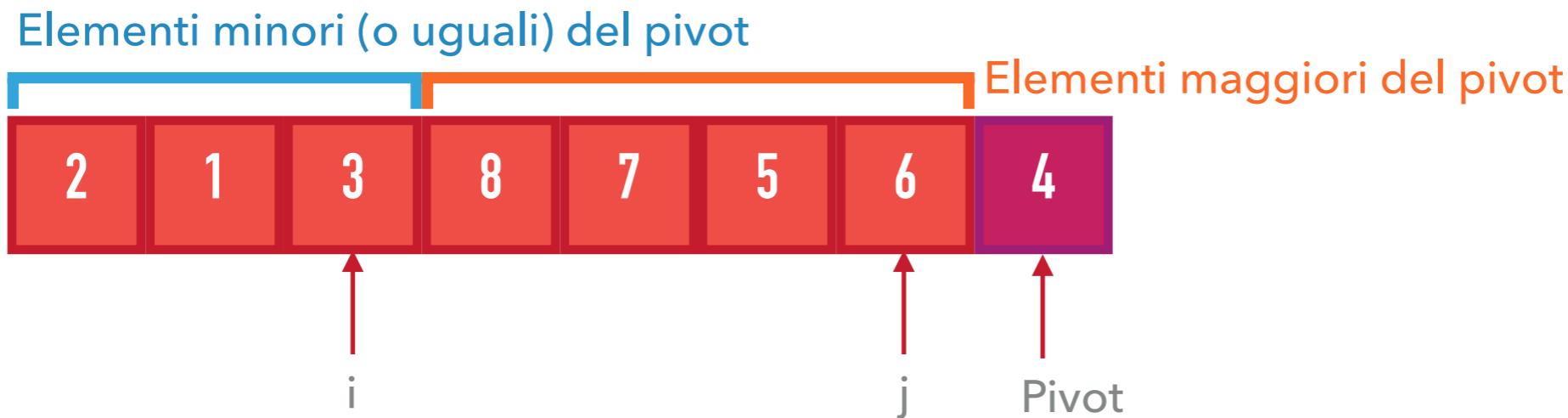
Se $A[j]$ è minore del pivot
incrementiamo i e scambiamo $A[i]$ e $A[j]$

PROCEDURA DI PARTIZIONAMENTO



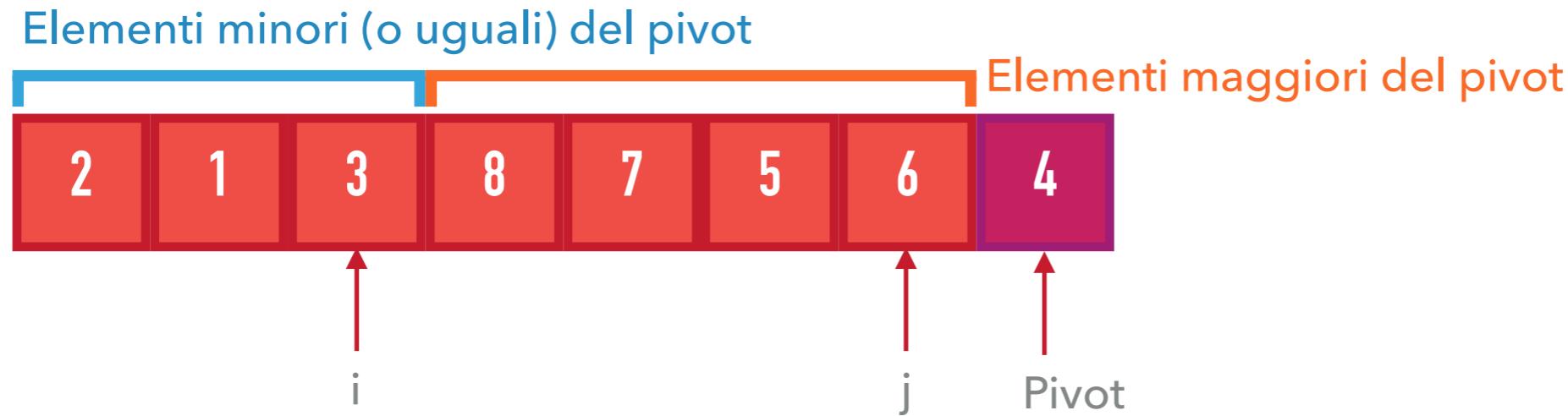
Se $A[j]$ è maggiore del pivot
non facciamo nulla

PROCEDURA DI PARTIZIONAMENTO



Se $A[j]$ è maggiore del pivot
non facciamo nulla

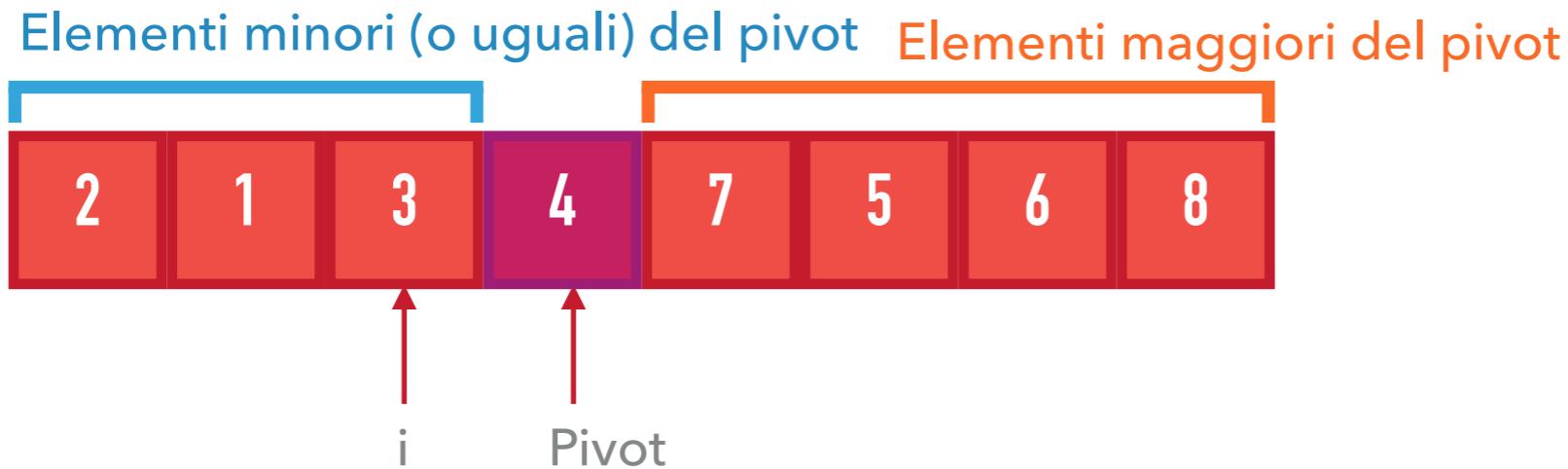
PROCEDURA DI PARTIZIONAMENTO



Abbiamo effettuato una scansione di tutto l'array e abbiamo raccolto tutti gli elementi minori o uguali del pivot negli indici $[0, i]$.

Dobbiamo solo posizionale il pivot tra le due partizioni

PROCEDURA DI PARTIZIONAMENTO



Ci basta scambiare l'elemento in posizione $i + 1$ (che è necessariamente maggiore del pivot o il pivot stesso) con il pivot

PARTIZIONAMENTO: PSEUDOCODICE

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

x = A[n-1] # il pivot è l'ultimo elemento dell'array

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
```

```
i = -1 # posizione iniziale degli elementi minori del pivot
```

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
```

```
i = -1 # posizione iniziale degli elementi minori del pivot
```

```
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
```

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
i = -1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
    if A[j] <= x # se troviamo un elemento minore del pivot...
```

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
i = -1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
    if A[j] <= x # se troviamo un elemento minore del pivot...
        i = i + 1
```

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
i = -1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
    if A[j] <= x # se troviamo un elemento minore del pivot...
        i = i + 1
        scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array
```

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
i = -1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
    if A[j] <= x # se troviamo un elemento minore del pivot...
        i = i + 1
        scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array
Scambia A[i+1] con A[n-1] # mettiamo il pivot nella sua posizione finale
return i+1
```

PARTIZIONAMENTO: PSEUDOCODICE

Parametri: A (un array)

```
x = A[n-1] # il pivot è l'ultimo elemento dell'array
i = -1 # posizione iniziale degli elementi minori del pivot
for j in range(0, n-1) # per tutti gli elementi dell'array tranne l'ultimo
    if A[j] <= x # se troviamo un elemento minore del pivot...
        i = i + 1
        scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array
Scambia A[i+1] con A[n-1] # mettiamo il pivot nella sua posizione finale
return i+1
```

Ma a noi servirà partizionare segmenti arbitrari di un array, quindi possiamo usare un'altra versione della procedura di partizionamento che lo applica solo tra due indici

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)

i = p-1 # posizione iniziale degli elementi minori del pivot

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)

i = p-1 # posizione iniziale degli elementi minori del pivot

for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

```
x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)
i = p-1 # posizione iniziale degli elementi minori del pivot
for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot
if A[j] <= x # se troviamo un elemento minore del pivot...
```

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

```
x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)
```

```
i = p-1 # posizione iniziale degli elementi minori del pivot
```

```
for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot
```

```
if A[j] <= x # se troviamo un elemento minore del pivot...
```

```
    i = i + 1
```

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)

i = p-1 # posizione iniziale degli elementi minori del pivot

for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot

if A[j] ≤ x # se troviamo un elemento minore del pivot...

 i = i + 1

 scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

```
x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)
```

```
i = p-1 # posizione iniziale degli elementi minori del pivot
```

```
for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot
```

```
if A[j] <= x # se troviamo un elemento minore del pivot...
```

```
    i = i + 1
```

```
        scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array
```

```
Scambia A[i+1] con A[r] # mettiamo il pivot nella sua posizione finale
```

PARTIZIONAMENTO: PSEUDOCODICE

Vediamo una procedura più generale che effettua il partizionamento tra gli indici p e r

Parametri: A (un array), p, r (indice di inizio e fine)

x = A[r] # il pivot è l'ultimo elemento dell'array (tra gli indici p e r)

i = p-1 # posizione iniziale degli elementi minori del pivot

for j in range(p,r) # per tutti gli elementi dell'array tranne il pivot

if A[j] <= x # se troviamo un elemento minore del pivot...

 i = i + 1

 scambia A[i] e A[j] # ...lo spostiamo nella prima parte dell'array

Scambia A[i+1] con A[r] # mettiamo il pivot nella sua posizione finale

return i+1

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

- ▶ Invariante (condizione che rimane vera ad ogni ciclo):
 - ▶ Gli elementi tra l'inizio dell'array e i sono tutti minori o uguali del pivot
 - ▶ Gli elementi tra $i + 1$ e j sono tutti maggiori del pivot

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

- ▶ Invariante (condizione che rimane vera ad ogni ciclo):
 - ▶ Gli elementi tra l'inizio dell'array e i sono tutti minori o uguali del pivot
 - ▶ Gli elementi tra $i + 1$ e j sono tutti maggiori del pivot
- ▶ Ogni iterazione continua a far rispettare questa condizione:

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

- ▶ Invariante (condizione che rimane vera ad ogni ciclo):
 - ▶ Gli elementi tra l'inizio dell'array e i sono tutti minori o uguali del pivot
 - ▶ Gli elementi tra $i + 1$ e j sono tutti maggiori del pivot
- ▶ Ogni iterazione continua a far rispettare questa condizione:
 - ▶ Se il nuovo elemento è maggiore del pivot viene mantenuto nella sua posizione, rispettando quindi la condizione

PARTIZIONAMENTO: PERCHÉ FUNZIONA?

- ▶ Invariante (condizione che rimane vera ad ogni ciclo):
 - ▶ Gli elementi tra l'inizio dell'array e i sono tutti minori o uguali del pivot
 - ▶ Gli elementi tra $i + 1$ e j sono tutti maggiori del pivot
- ▶ Ogni iterazione continua a far rispettare questa condizione:
 - ▶ Se il nuovo elemento è maggiore del pivot viene mantenuto nella sua posizione, rispettando quindi la condizione
 - ▶ Se il nuovo elemento è minore o uguale del pivot, i viene incrementato l'elemento in posizione j viene scambiato con quello in posizione i (che, dato che i è stato incrementato, è maggiore del pivot)

PARTIZIONAMENTO: COMPLESSITÀ

- ▶ Il partizionamento viene fatto con una singola “passata” dell’array (il ciclo for esterno)
- ▶ Tutte le operazioni all’interno e all’esterno del ciclo for hanno un costo costante
- ▶ Ne segue che il partizionamento viene fatto in tempo $\Theta(n)$

QUICKSORT: PSEUDOCODICE

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

if $p \geq r$:

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

if $p \geq r$:

 return

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

```
if p ≥ r:  
    return  
q = partiziona(A, p, r) # indice del pivot dopo il partizionamento
```

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

```
if p ≥ r:  
    return  
q = partiziona(A, p, r) # indice del pivot dopo il partizionamento  
quicksort(A, p, q-1) # chiamata ricorsiva sugli elementi minori o uguali
```

QUICKSORT: PSEUDOCODICE

Parametri: A (un array), p, r (indice di inizio e fine)

```
if p ≥ r:  
    return  
q = partiziona(A, p, r) # indice del pivot dopo il partizionamento  
quicksort(A, p, q-1) # chiamata ricorsiva sugli elementi minori o uguali  
quicksort(A, q+1, r) # chiamata ricorsiva sugli elementi maggiori
```

QUICKSORT: PERCHÉ FUNZIONA?

QUICKSORT: PERCHÉ FUNZIONA?

- ▶ Per un array di dimensione 0 o 1 il quicksort funziona per motivi banali

QUICKSORT: PERCHÉ FUNZIONA?

- ▶ Per un array di dimensione 0 o 1 il quicksort funziona per motivi banali
- ▶ Supponiamo di avere provato che il quicksort funziona per ogni dimensione minore di n , vediamo che funziona per la dimensione n

QUICKSORT: PERCHÉ FUNZIONA?

- ▶ Per un array di dimensione 0 o 1 il quicksort funziona per motivi banali
- ▶ Supponiamo di avere provato che il quicksort funziona per ogni dimensione minore di n , vediamo che funziona per la dimensione n
- ▶ Dopo la procedura di partizionamento il pivot è nella posizione corretta e otteniamo due sotto-array uno che precede il pivot con tutti gli elementi minori o uguali al pivot e uno con tutti gli elementi maggiori

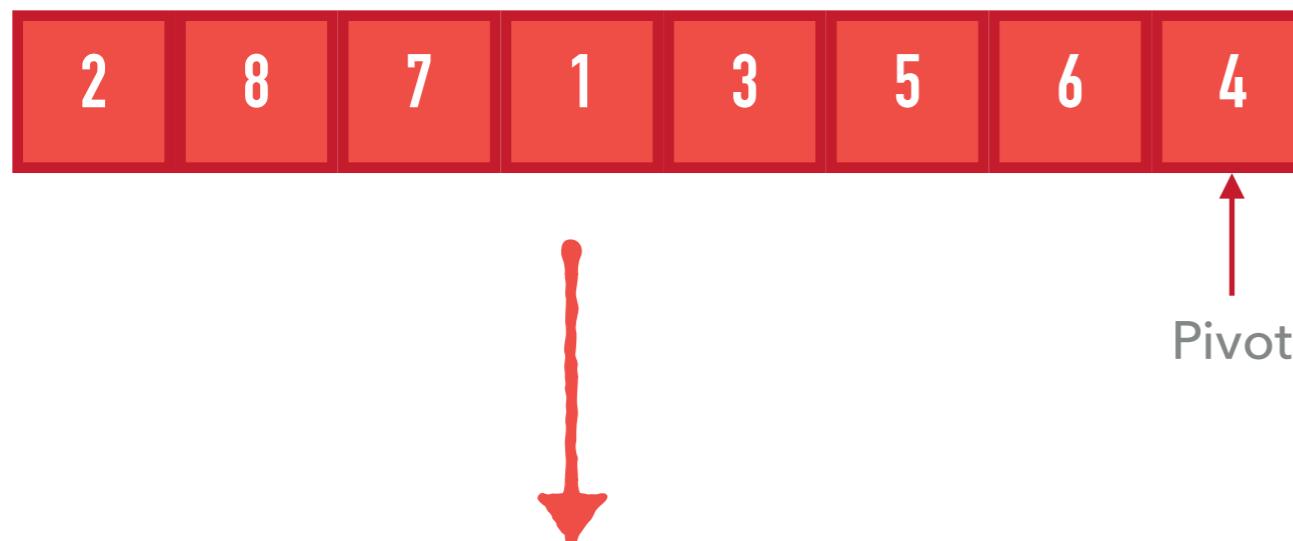
QUICKSORT: PERCHÉ FUNZIONA?

- ▶ Per un array di dimensione 0 o 1 il quicksort funziona per motivi banali
- ▶ Supponiamo di avere provato che il quicksort funziona per ogni dimensione minore di n , vediamo che funziona per la dimensione n
- ▶ Dopo la procedura di partizionamento il pivot è nella posizione corretta e otteniamo due sotto-array uno che precede il pivot con tutti gli elementi minori o uguali al pivot e uno con tutti gli elementi maggiori
- ▶ Richiamiamo quicksort sui due sotto-array di dimensione minore di n , che per ipotesi ci restituiranno gli array ordinati
- ▶ Otteniamo tutti gli elementi minori o uguali al pivot ordinati, seguiti dal pivot e da tutti gli elementi maggiori del pivot ordinati. Quindi l'array di n elementi è ordinato.

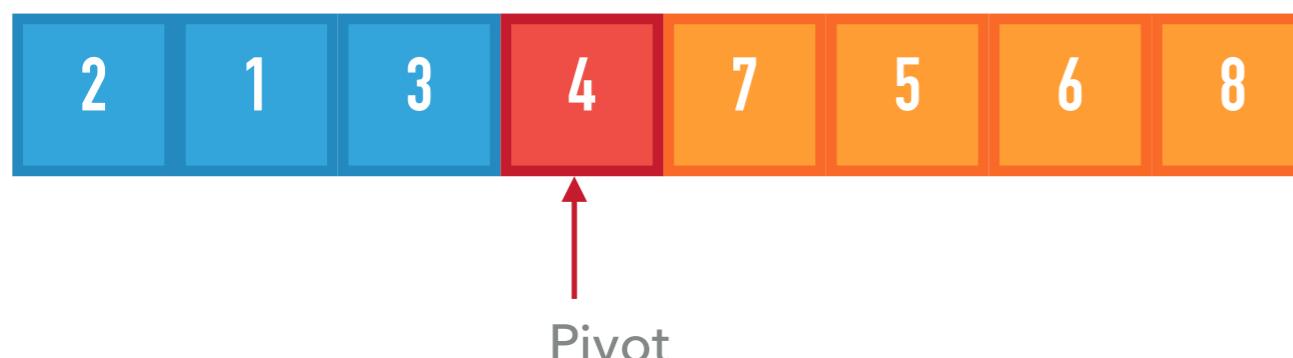
ANALISI DELLA COMPLESSITÀ

- ▶ L'analisi della complessità del quicksort è più delicata di mergesort e heapsort
- ▶ La dimensione degli array nelle chiamate ricorsive dipende dalla procedura di partizionamento
- ▶ La procedura di partizionamento dipende, a sua volta dai dati che abbiamo

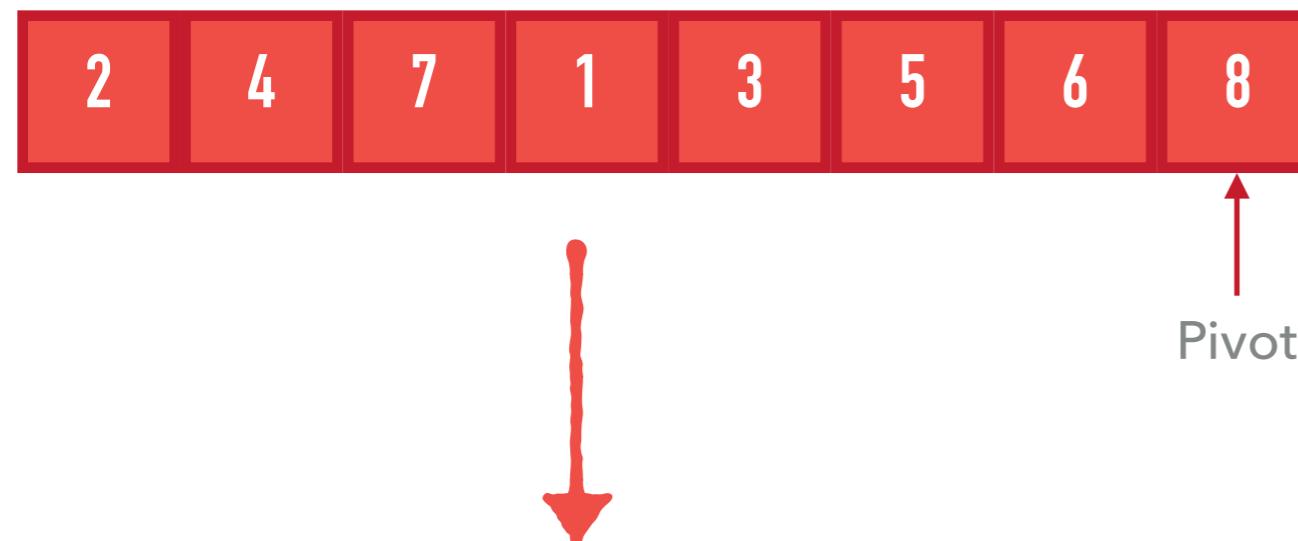
POSSIBILI PARTIZIONAMENTI



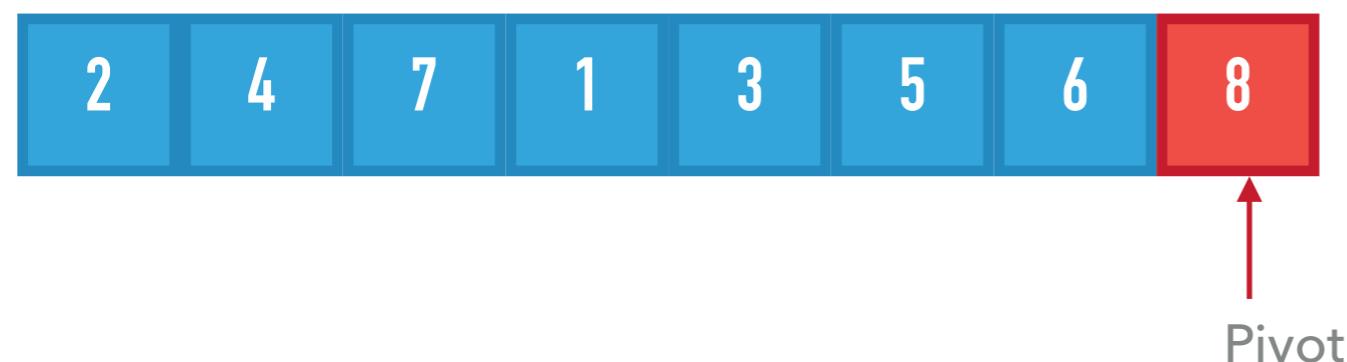
Un “buon” partizionamento:
i due sottoarray risultanti sono
ognuno circa la metà dell’array di
partenza



POSSIBILI PARTIZIONAMENTI



Un “cattivo” partizionamento:
uno dei sotto-array risultanti
contiene tutti gli elementi tranne
uno e l’altro è vuoto!



QUIZ

In **quali** dei seguenti casi il partizionamento è maggiormente sbilanciato?

1) [1, 2, 3, 4, 5]

2) [2, 6, 5, 3, 4]

3) [1, 2, 5, 4, 3]

4) [5, 4, 3, 2, 1]

QUIZ

In **quali** dei seguenti casi il partizionamento è maggiormente sbilanciato?

1) [1, 2, 3, 4, 5]

2) [2, 6, 5, 3, 4]

3) [1, 2, 5, 4, 3]

4) [5, 4, 3, 2, 1]

QUIZ

In **quali** dei seguenti casi il partizionamento è maggiormente sbilanciato?

1) [1, 2, 3, 4, 5]

2) [2, 6, 5, 3, 4]

3) [1, 2, 5, 4, 3]

4) [5, 4, 3, 2, 1]

“BUON PARTIZIONAMENTO”: ANALISI

“BUON PARTIZIONAMENTO”: ANALISI

- ▶ Assumiamo che ogni partizionamento crei due sottoarray di dimensioni approssimativamente $n/2$
- ▶ L'equazione di ricorrenza diventa quindi:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

“BUON PARTIZIONAMENTO”: ANALISI

- ▶ Assumiamo che ogni partizionamento crei due sottoarray di dimensioni approssimativamente $n/2$
- ▶ L'equazione di ricorrenza diventa quindi:
$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$
- ▶ Per il teorema dell'esperto il tempo di calcolo del quicksort è $\Theta(n \log n)$
- ▶ Ma questo risultato vale solo per un “buon” partizionamento

“CATTIVO PARTIZIONAMENTO”: ANALISI

“CATTIVO PARTIZIONAMENTO”: ANALISI

- ▶ Assumiamo che ogni partizionamento crei un sottoarray vuoto ed uno di dimensione $n - 1$
- ▶ L'equazione di ricorrenza diventa quindi:

$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$$

“CATTIVO PARTIZIONAMENTO”: ANALISI

- ▶ Assumiamo che ogni partizionamento crei un sottoarray vuoto ed uno di dimensione $n - 1$
- ▶ L'equazione di ricorrenza diventa quindi:
$$T(n) = T(n - 1) + T(0) + \Theta(n) = T(n - 1) + \Theta(n)$$
- ▶ Se espandiamo $T(n)$ vediamo che abbiamo n “passi ricorsivi”, ognuno dei quali esegue un lavoro lineare rispetto alla dimensione dell'array da ordinare
- ▶ Come risultato otteniamo $\Theta(n^2)$

ANALISI DELLA COMPLESSITÀ

ANALISI DELLA COMPLESSITÀ

- ▶ Nel caso peggiore quindi otteniamo $\Theta(n^2)$
- ▶ Ma quanto è frequente il caso peggiore?

ANALISI DELLA COMPLESSITÀ

- ▶ Nel caso peggiore quindi otteniamo $\Theta(n^2)$
- ▶ Ma quanto è frequente il caso peggiore?
- ▶ Supponiamo che le nostre partizioni siano molto sbilanciate: la prima metà include $1/10$ dell'array e la seconda ne include $9/10$
- ▶ Lo stesso ragionamento vale anche per $1/100$ e $99/100$, etc.

ANALISI DELLA COMPLESSITÀ

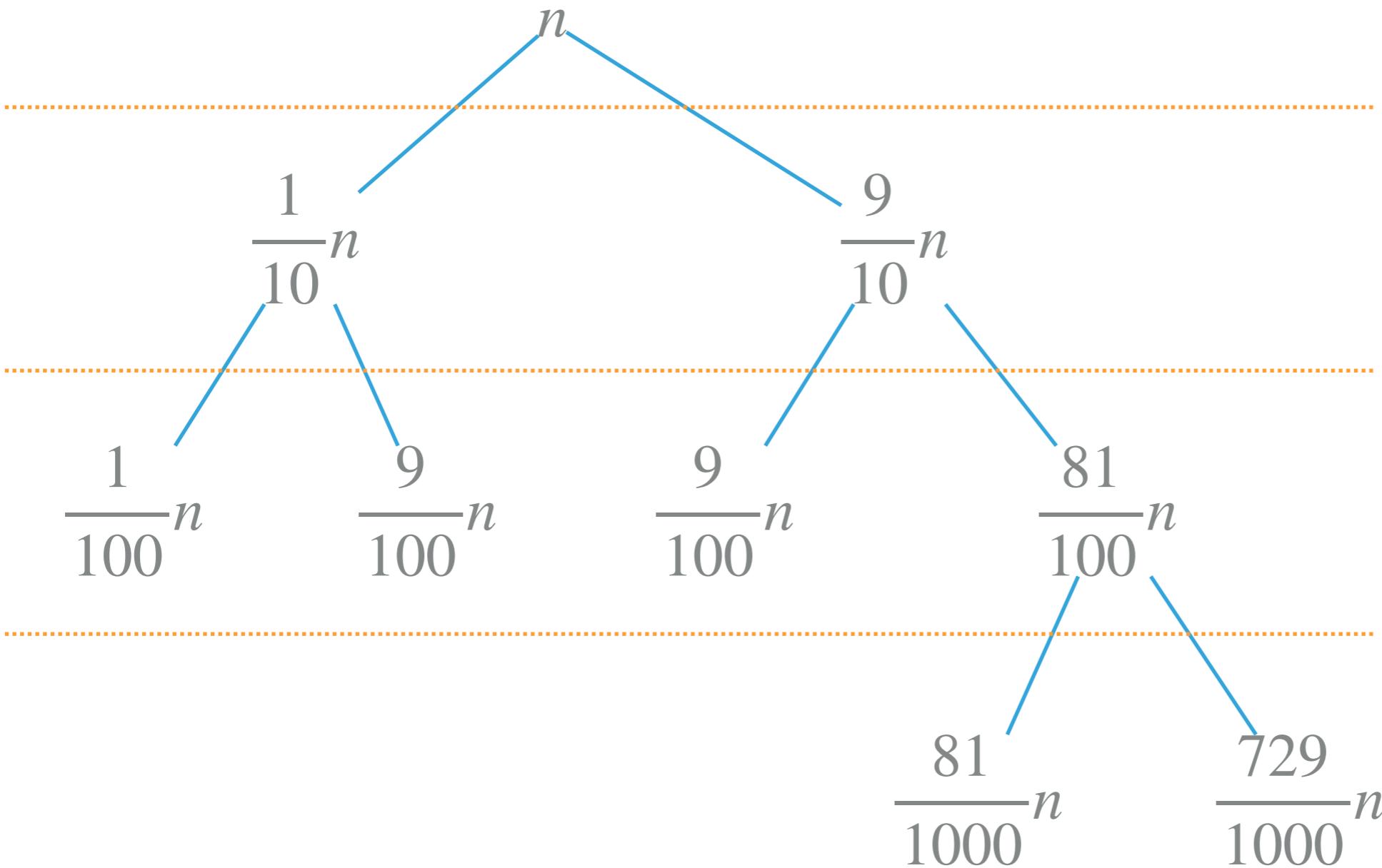
ANALISI DELLA COMPLESSITÀ

► $T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$

ANALISI DELLA COMPLESSITÀ

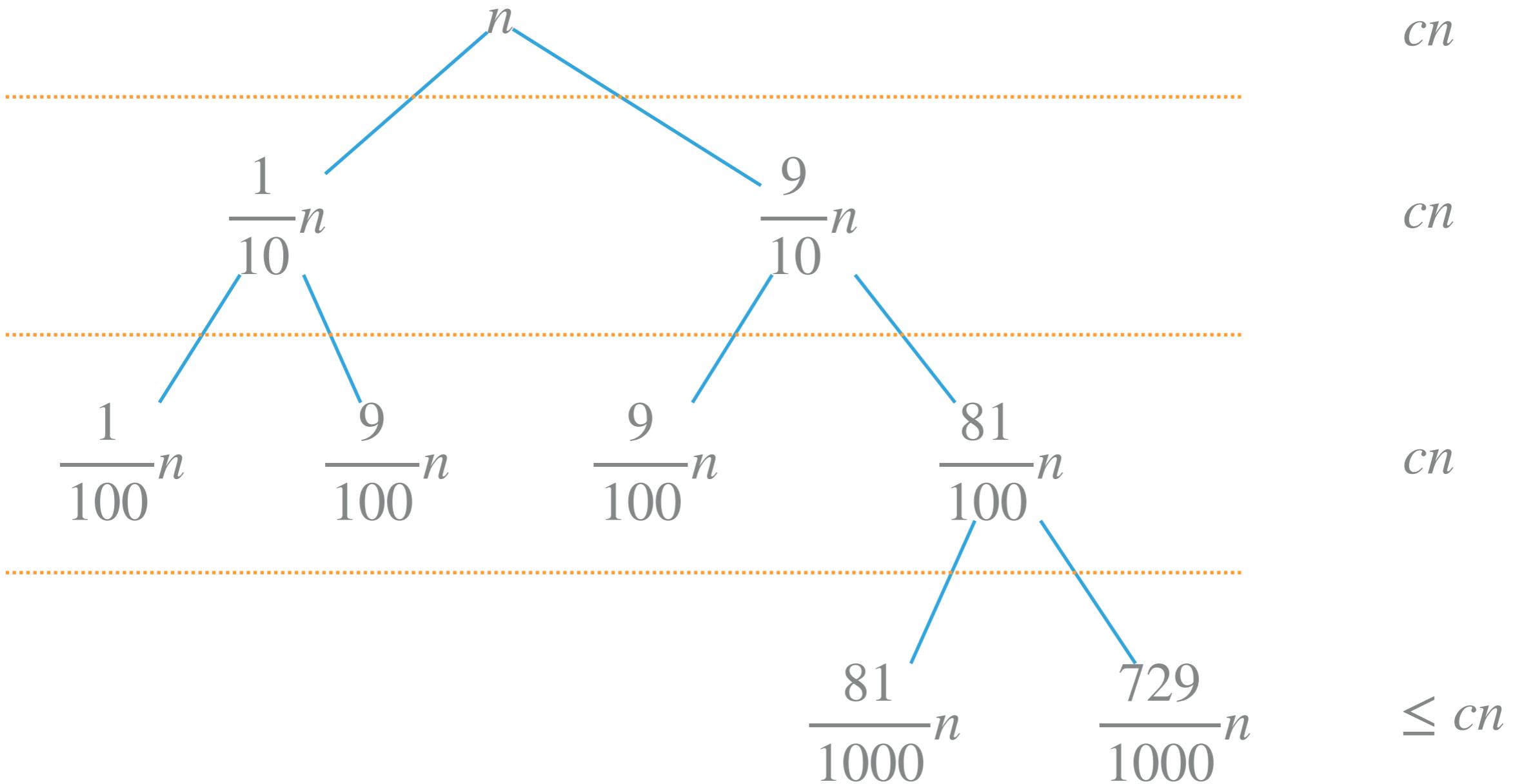
- ▶ $T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + \Theta(n)$
- ▶ Esplicitiamo la costante nascosta c nell' $\Theta(n)$:
$$T(n) = T\left(\frac{9}{10}n\right) + T\left(\frac{1}{10}n\right) + cn$$
- ▶ Costruiamo l'albero delle chiamate ricorsive

ANALISI DELLA COMPLESSITÀ

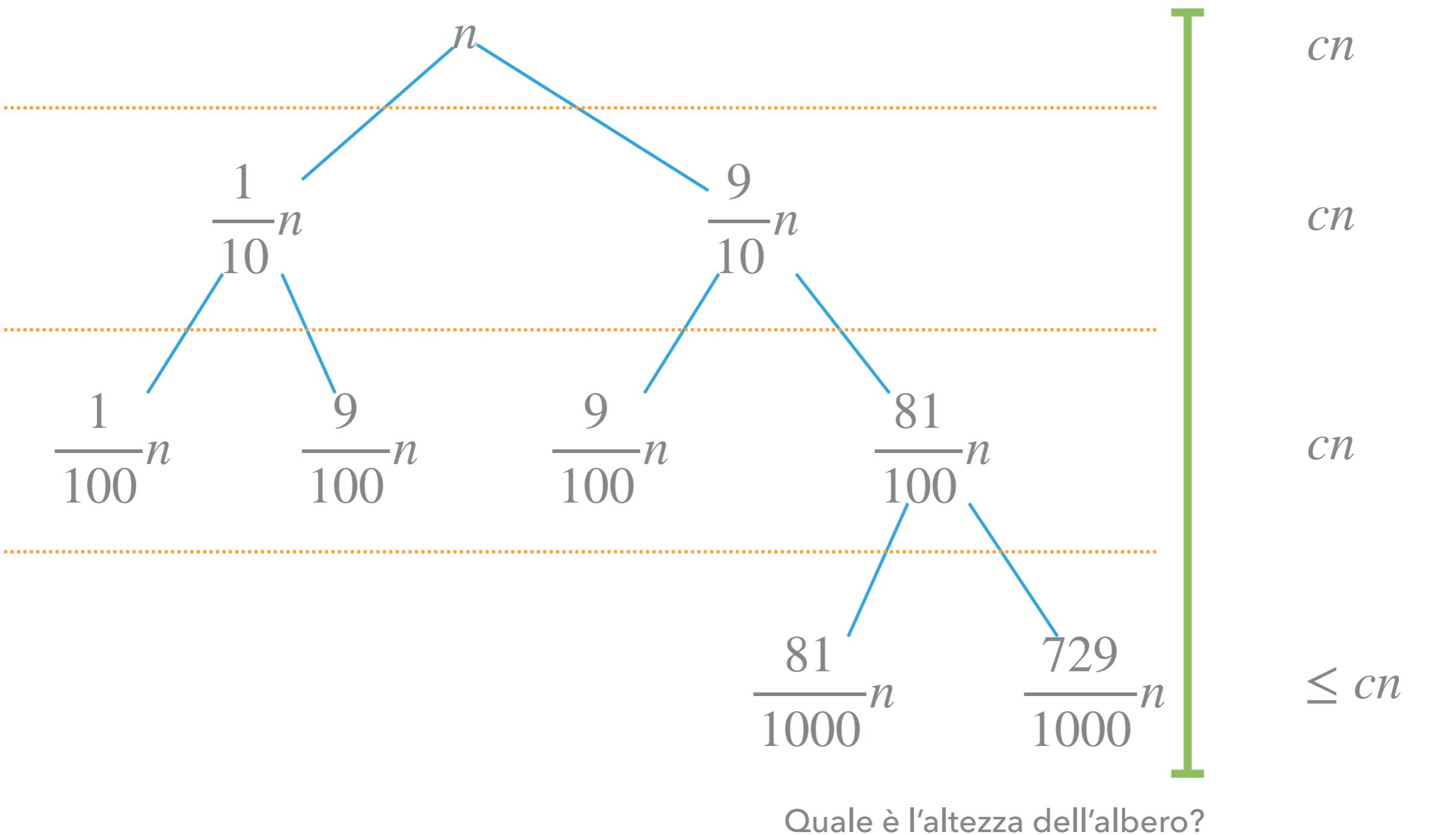


ANALISI DELLA COMPLESSITÀ

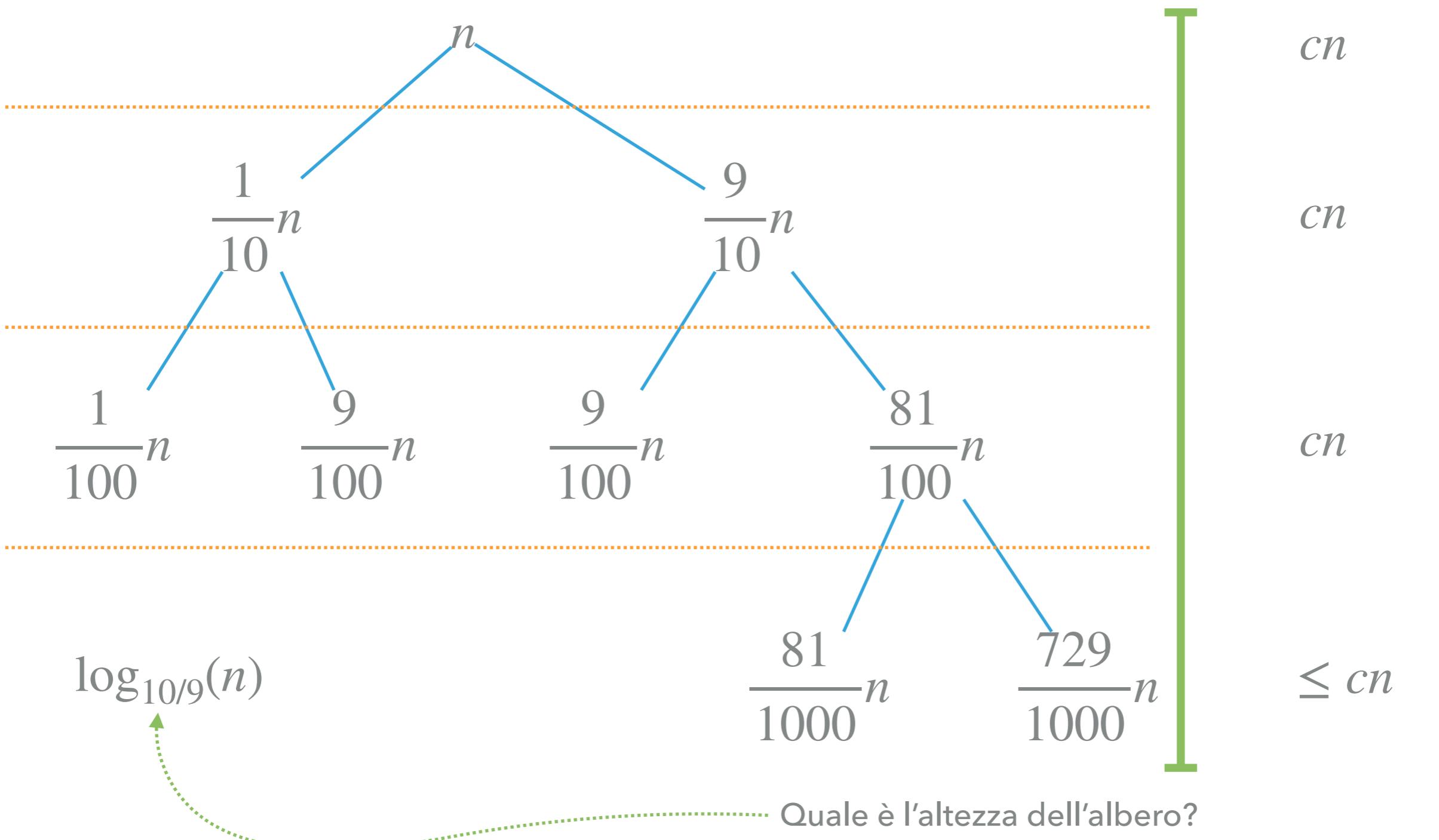
"costo" per livello



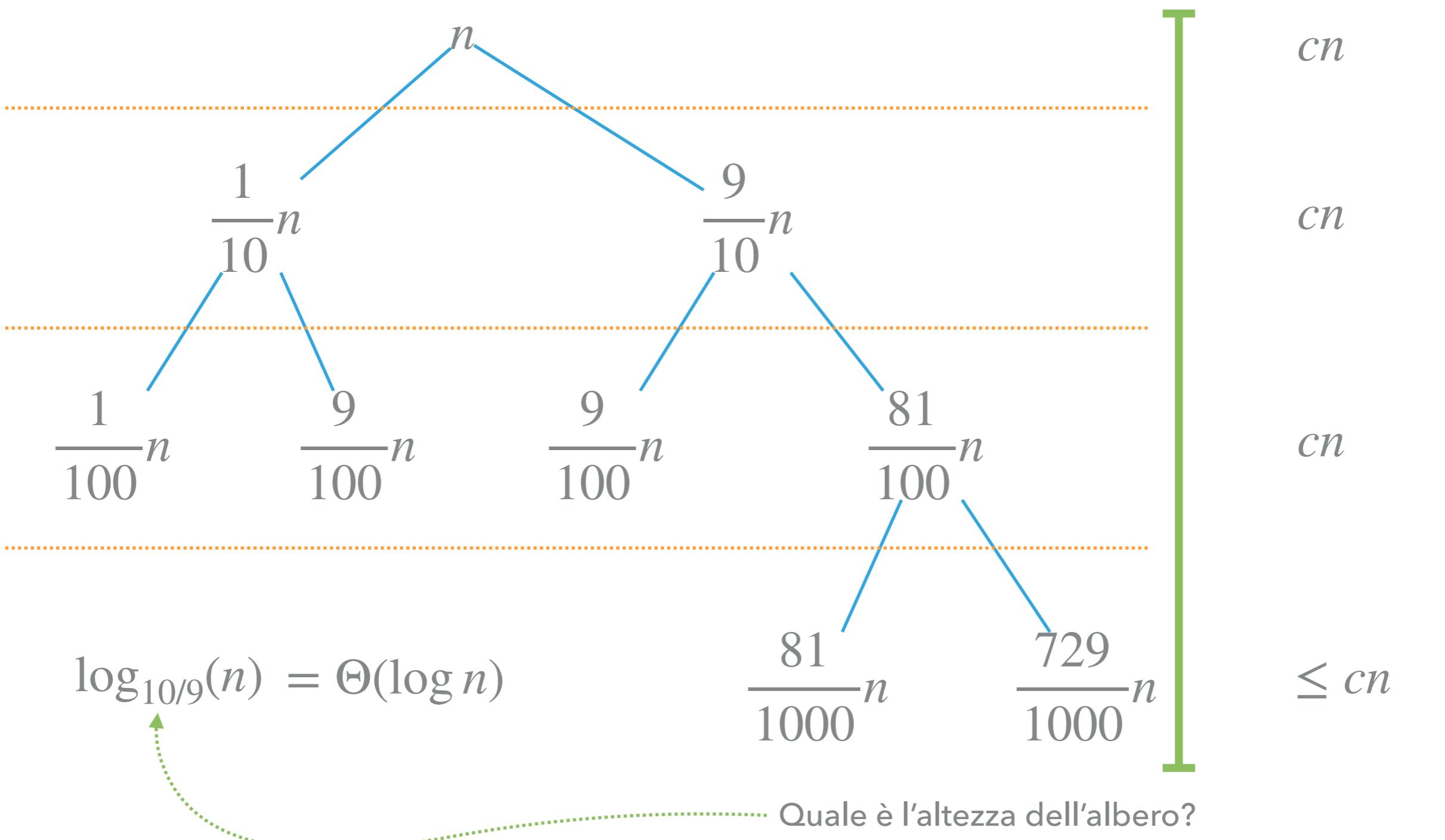
ANALISI DELLA COMPLESSITÀ



ANALISI DELLA COMPLESSITÀ



ANALISI DELLA COMPLESSITÀ



ANALISI DELLA COMPLESSITÀ

ANALISI DELLA COMPLESSITÀ

- ▶ Stiamo eseguendo al più cn passi per ognuno degli $\Theta(\log n)$ livelli dell'albero
- ▶ Di conseguenza il tempo di esecuzione è ancora $\Theta(n \log n)$

ANALISI DELLA COMPLESSITÀ

- ▶ Stiamo eseguendo al più cn passi per ognuno degli $\Theta(\log n)$ livelli dell'albero
- ▶ Di conseguenza il tempo di esecuzione è ancora $\Theta(n \log n)$
- ▶ Cosa succede nel “caso medio”?
- ▶ È possibile provare che, se la scelta del pivot viene effettuata in modo casuale, il **tempo atteso** di esecuzione è $\Theta(n \log n)$

QUICKSORT: POSSIBILI VARIANTI

QUICKSORT: POSSIBILI VARIANTI

- ▶ Esistono molte varianti del quicksort per minimizzare il rischio di cadere nel caso peggiore

QUICKSORT: POSSIBILI VARIANTI

- ▶ Esistono molte varianti del quicksort per minimizzare il rischio di cadere nel caso peggiore
- ▶ Invece di scegliere l'ultimo elemento come pivot, viene scelto un elemento a caso che viene spostato in ultima posizione (Randomized-Quicksort)

QUICKSORT: POSSIBILI VARIANTI

- ▶ Esistono molte varianti del quicksort per minimizzare il rischio di cadere nel caso peggiore
- ▶ Invece di scegliere l'ultimo elemento come pivot, viene scelto un elemento a caso che viene spostato in ultima posizione (Randomized-Quicksort)
- ▶ Vengono presi tre elementi e la mediana dei tre viene usata come pivot

QUICKSORT: POSSIBILI VARIANTI

- ▶ Esistono molte varianti del quicksort per minimizzare il rischio di cadere nel caso peggiore
- ▶ Invece di scegliere l'ultimo elemento come pivot, viene scelto un elemento a caso che viene spostato in ultima posizione (Randomized-Quicksort)
- ▶ Vengono presi tre elementi e la mediana dei tre viene usata come pivot
- ▶ I libri “algorithms in C”, “algorithms in Java” di Robert Sedgewick trattano molti di questi miglioramenti

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Come sono distribuiti gli input? Consideriamo array A di lunghezza n , con elementi diversi tra loro, e assumiamo ogni permutazione sia equiprobabile.
- ▶ L'analisi che faremo vale uguale anche per randomized quicksort.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Come sono distribuiti gli input? Consideriamo array A di lunghezza n , con elementi diversi tra loro, e assumiamo ogni permutazione sia equiprobabile.
- ▶ L'analisi che faremo vale uguale anche per randomized quicksort.
- ▶ Il **rango** di un elemento $a \in A$ è la sua posizione nell'array A ordinato.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Come sono distribuiti gli input? Consideriamo array A di lunghezza n , con elementi diversi tra loro, e assumiamo ogni permutazione sia equiprobabile.
- ▶ L'analisi che faremo vale uguale anche per randomized quicksort.
- ▶ Il **rango** di un elemento $a \in A$ è la sua posizione nell'array A ordinato.
- ▶ Indichiamo con z_i l'elemento di A di rango i , quindi $Z = [z_1, \dots, z_n]$ è l'array ordinato.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Cosa domina il costo di quicksort?

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Cosa domina il costo di quicksort?
- ▶ Il costo è uguale al numero totale di confronti del tipo $A[j] \leq pivot$ in **tutte** le chiamate di partition. Chiamo X la **variabile aleatoria** che conta il numero totale di confronti in partition.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Cosa domina il costo di quicksort?
- ▶ Il costo è uguale al numero totale di confronti del tipo $A[j] \leq pivot$ in **tutte** le chiamate di partition. Chiamo X la **variabile aleatoria** che conta il numero totale di confronti in partition.
- ▶ Ogni coppia z_i, z_j di elementi verrà confrontata al più una volta.
Perchè?

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Cosa domina il costo di quicksort?
- ▶ Il costo è uguale al numero totale di confronti del tipo $A[j] \leq pivot$ in **tutte** le chiamate di partition. Chiamo X la **variabile aleatoria** che conta il numero totale di confronti in partition.
- ▶ Ogni coppia z_i, z_j di elementi verrà confrontata al più una volta.
Perchè?
 - ▶ perchè confronto solo con il pivot, che a fine partition non viene più toccato.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Cosa domina il costo di quicksort?
- ▶ Il costo è uguale al numero totale di confronti del tipo $A[j] \leq pivot$ in **tutte** le chiamate di partition. Chiamo X la **variabile aleatoria** che conta il numero totale di confronti in partition.
- ▶ Ogni coppia z_i, z_j di elementi verrà confrontata al più una volta.
Perchè?
 - ▶ perchè confronto solo con il pivot, che a fine partition non viene più toccato.
 - ▶ Sia $X_{i,j}$ una v.a. che vale 1 se confronto z_i con z_j e 0 altrimenti.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Possiamo scrivere:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}$$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Possiamo scrivere:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}$$

- ▶ Ci interessa il numero atteso di confronti:

$$\mathbb{E}[X] = \mathbb{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{i,j}]$$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Possiamo scrivere:

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j}$$

- ▶ Ci interessa il numero atteso di confronti:

$$\mathbb{E}[X] = \mathbb{E} \left[\sum_{i=1}^{n-1} \sum_{j=i+1}^n X_{i,j} \right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \mathbb{E}[X_{i,j}]$$

- ▶ Vale anche $\mathbb{E}[X_{i,j}] = \Pr\{X_{i,j} = 1\}$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Qual è la probabilità che z_i sia confrontato con z_j ?

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Qual è la probabilità che z_i sia confrontato con z_j ?
- ▶ Sia $Z_{i,j} = [z_i, z_{i+1}, \dots, z_j]$, e sia z_k il primo elemento di $Z_{i,j}$ ad essere scelto come pivot.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Qual è la probabilità che z_i sia confrontato con z_j ?
- ▶ Sia $Z_{i,j} = [z_i, z_{i+1}, \dots, z_j]$, e sia z_k il primo elemento di $Z_{i,j}$ ad essere scelto come pivot.
- ▶ Se $z_k = z_i \circ z_k = z_j$, allora $X_{i,j} = 1$ altrimenti $X_{i,j} = 0$. Why?

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Qual è la probabilità che z_i sia confrontato con z_j ?
- ▶ Sia $Z_{i,j} = [z_i, z_{i+1}, \dots, z_j]$, e sia z_k il primo elemento di $Z_{i,j}$ ad essere scelto come pivot.
- ▶ Se $z_k = z_i \circ z_k = z_j$, allora $X_{i,j} = 1$ altrimenti $X_{i,j} = 0$. Why?
- ▶ Se scelgo un $z_k, k \neq i, j$, allora z_i e z_j finiscono in due call ricorsive diverse, e non possono più essere confrontati.

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ Qual è la probabilità che z_i sia confrontato con z_j ?
- ▶ Sia $Z_{i,j} = [z_i, z_{i+1}, \dots, z_j]$, e sia z_k il primo elemento di $Z_{i,j}$ ad essere scelto come pivot.
- ▶ Se $z_k = z_i \circ z_k = z_j$, allora $X_{i,j} = 1$ altrimenti $X_{i,j} = 0$. Why?
- ▶ Se scelgo un $z_k, k \neq i, j$, allora z_i e z_j finiscono in due call ricorsive diverse, e non possono più essere confrontati.
- ▶ Per ipotesi sulla distribuzione:

$$\Pr\{z_k \text{ è il primo pivot in } Z_{i,j}\} = \frac{1}{j-i+1}$$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

► $Pr\{X_{i,j} = 1\} = \frac{2}{j - i + 1}$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ $Pr\{X_{i,j} = 1\} = \frac{2}{j - i + 1}$
- ▶ $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} < \sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w}$ e

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ $Pr\{X_{i,j} = 1\} = \frac{2}{j - i + 1}$
- ▶ $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} < \sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w}$ e
- ▶ $\sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$

ANALISI DELLA COMPLESSITÀ NEL CASO MEDIO

- ▶ $Pr\{X_{i,j} = 1\} = \frac{2}{j - i + 1}$
- ▶ $\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{2}{j - i + 1} < \sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w}$ e
- ▶ $\sum_{i=1}^{n-1} \sum_{w=1}^n \frac{1}{w} = \sum_{i=1}^{n-1} O(\log n) = O(n \log n)$
- ▶ La complessità nel caso medio è $O(n \log n)$!

LOWER BOUND DELL'ORDINAMENTO PER CONFRONTI
ORDINAMENTO IN TEMPO LINEARE
COUNTING SORT
RADIX SORT

ALGORITMI E STRUTTURE DATI

LOWER BOUND ALL'ORDINAMENTO PER CONFRONTI

LIMITI DELL'ORDINAMENTO

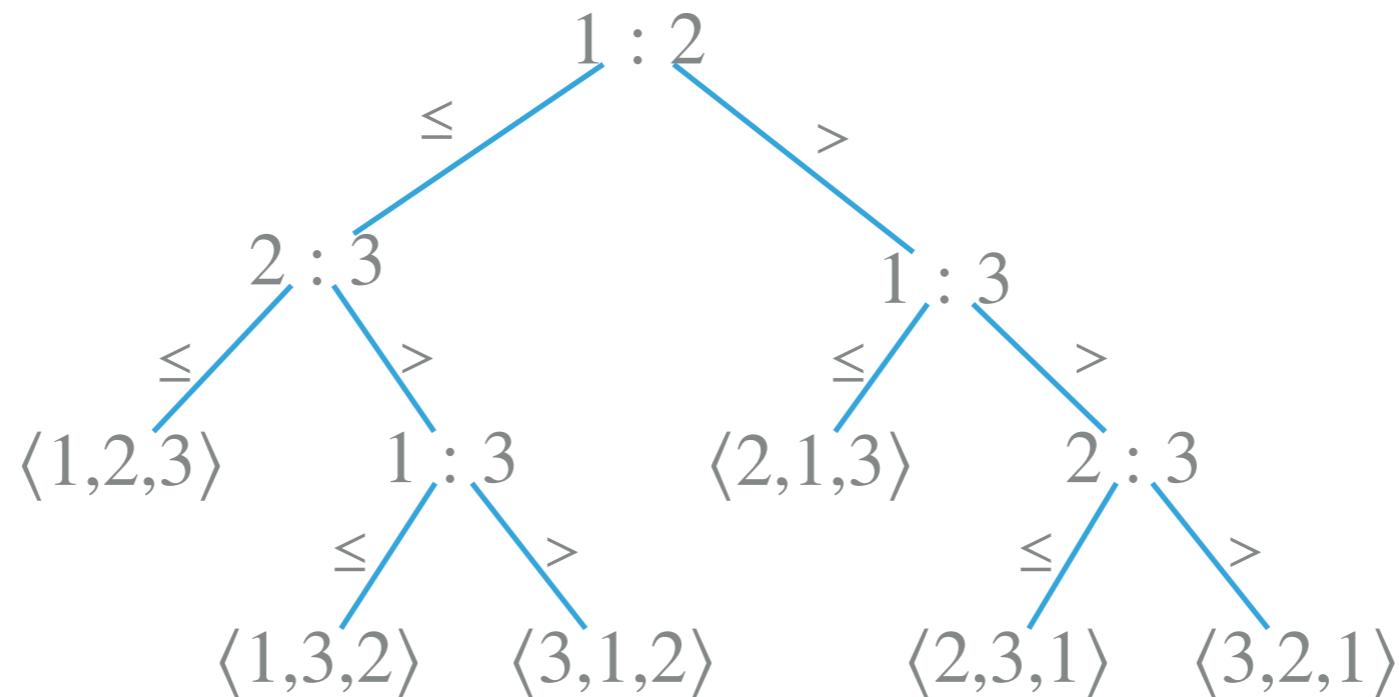
- ▶ Sia mergesort che heapsort richiedono tempo $O(n \log n)$
- ▶ Sappiamo di non poter andare al di sotto del tempo lineare: dobbiamo *almeno* leggere l'input
- ▶ Possiamo migliorare e avere, per esempio, un algoritmo di ordinamento che richiede tempo lineare?
- ▶ Mostriamo che gli algoritmi di ordinamento che usano la comparazione sono in $\Omega(n \log n)$

ALBERO DI DECISIONE

- ▶ Dato un array A di n elementi
- ▶ Consideriamo un albero binario in cui ogni nodo interno ha associata una comparazione $i : j$, che indica che compariamo l'elemento i -esimo con l'elemento j -esimo
- ▶ Ci spostiamo a sinistra se $A[i] \leq A[j]$ e a destra se $A[i] > A[j]$
- ▶ Ogni foglia è una permutazione π di $0 \dots n - 1$, gli indici di A

ALBERO DI DECISIONE

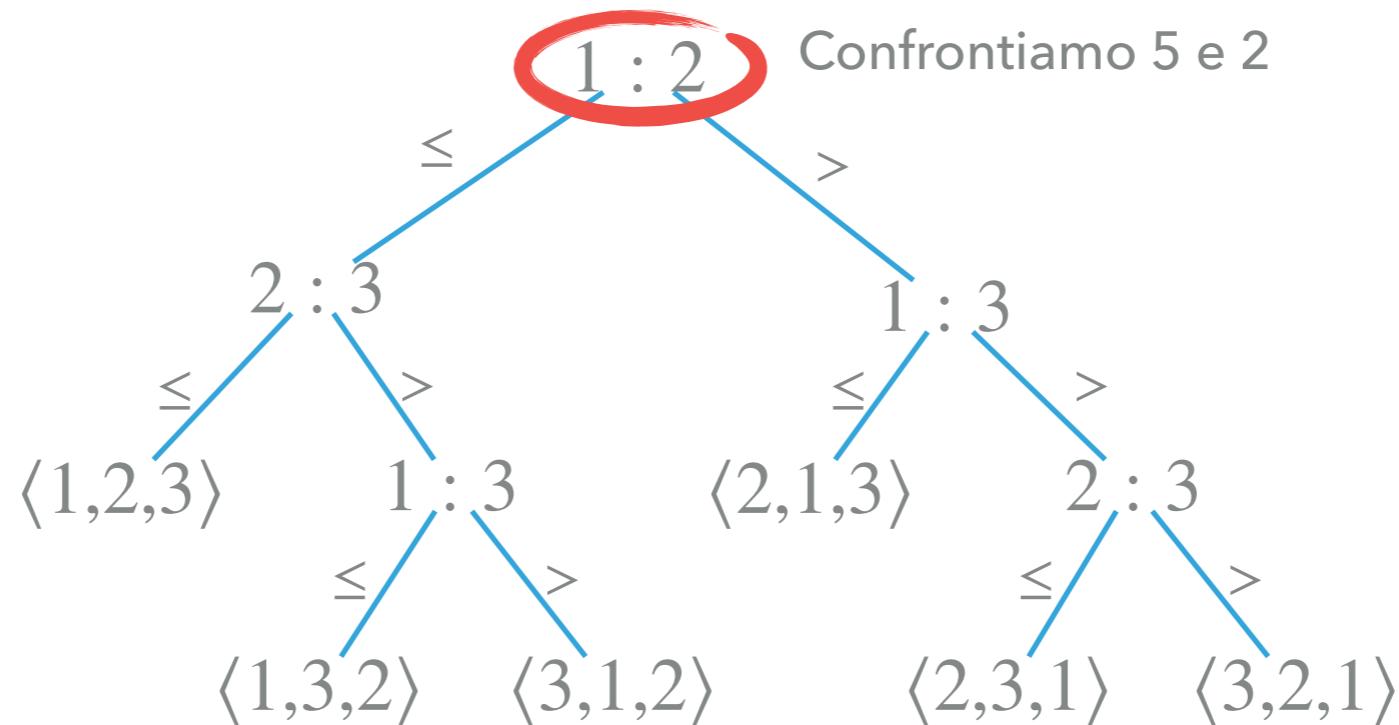
| | | |
|---|---|---|
| 5 | 2 | 4 |
|---|---|---|



In questo esempio gli indici dell'array partono da 1, come sul libro di testo

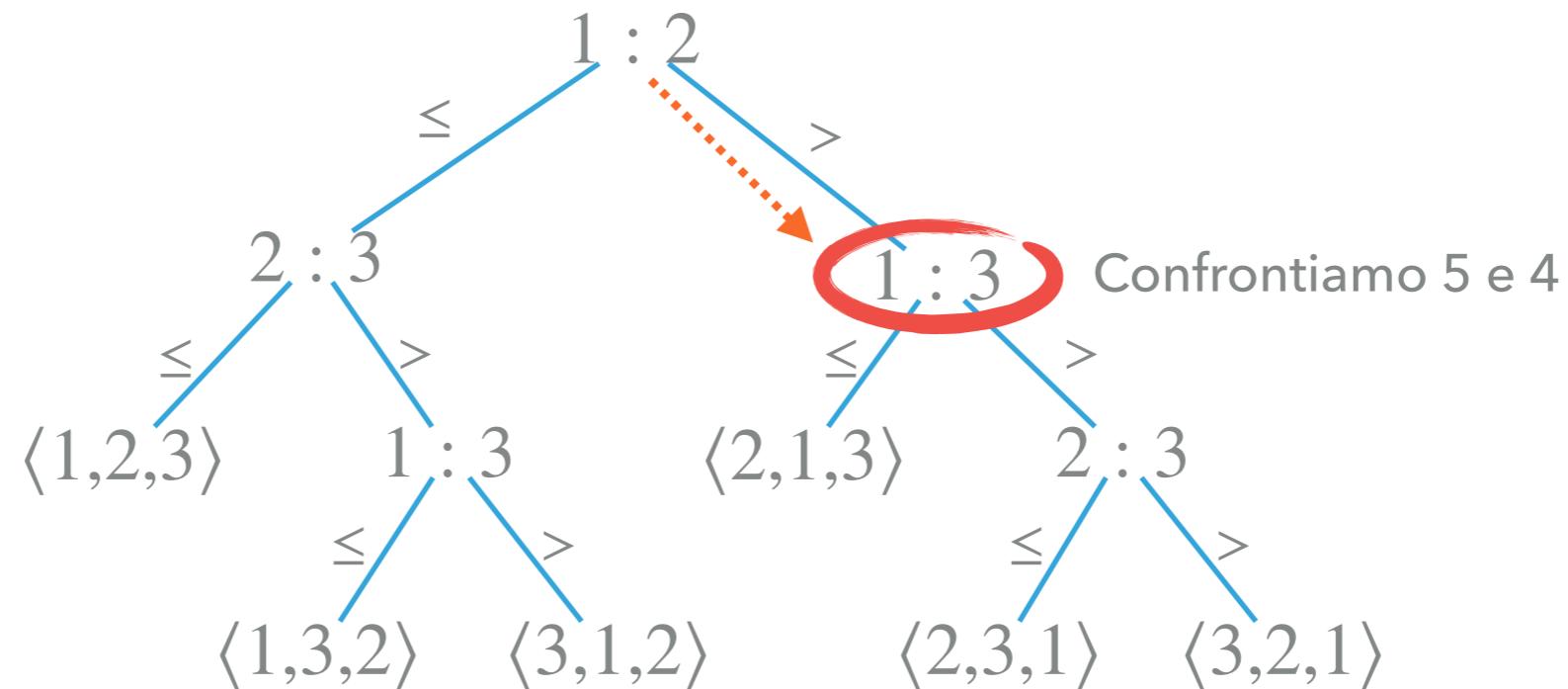
ALBERO DI DECISIONE

| | | |
|---|---|---|
| 5 | 2 | 4 |
|---|---|---|



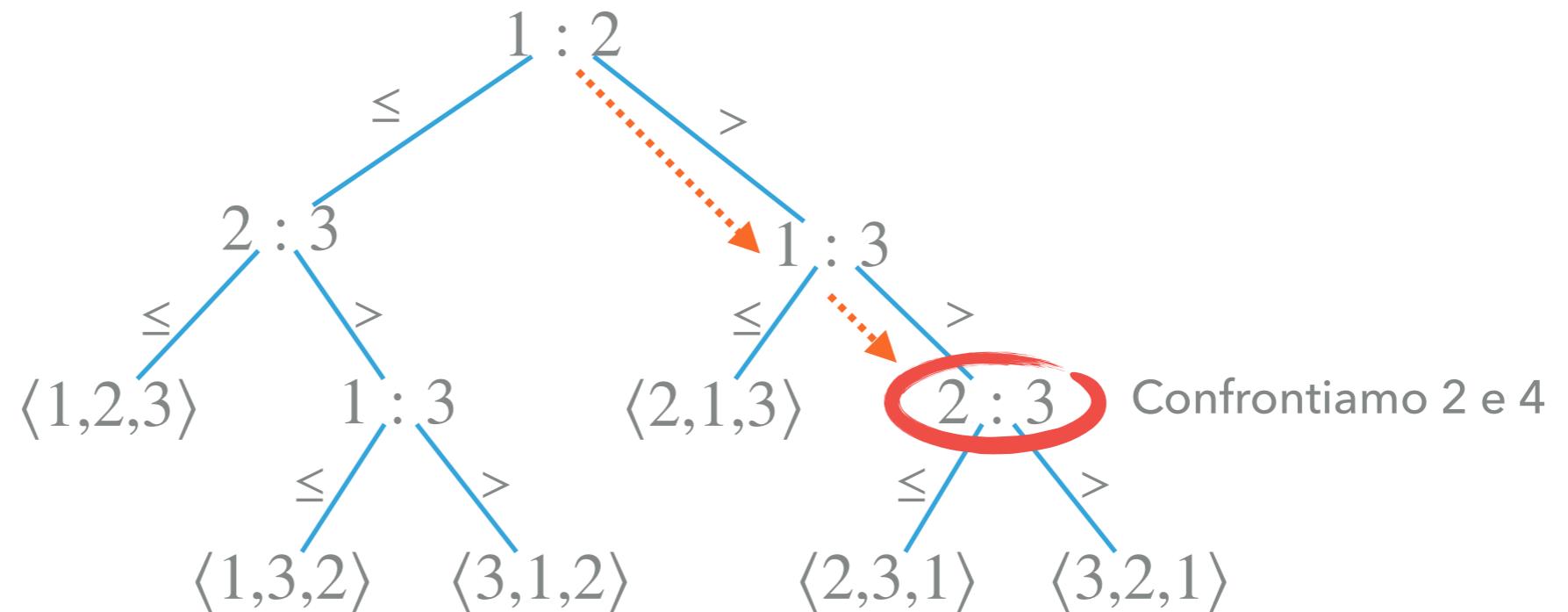
ALBERO DI DECISIONE

| | | |
|---|---|---|
| 5 | 2 | 4 |
|---|---|---|

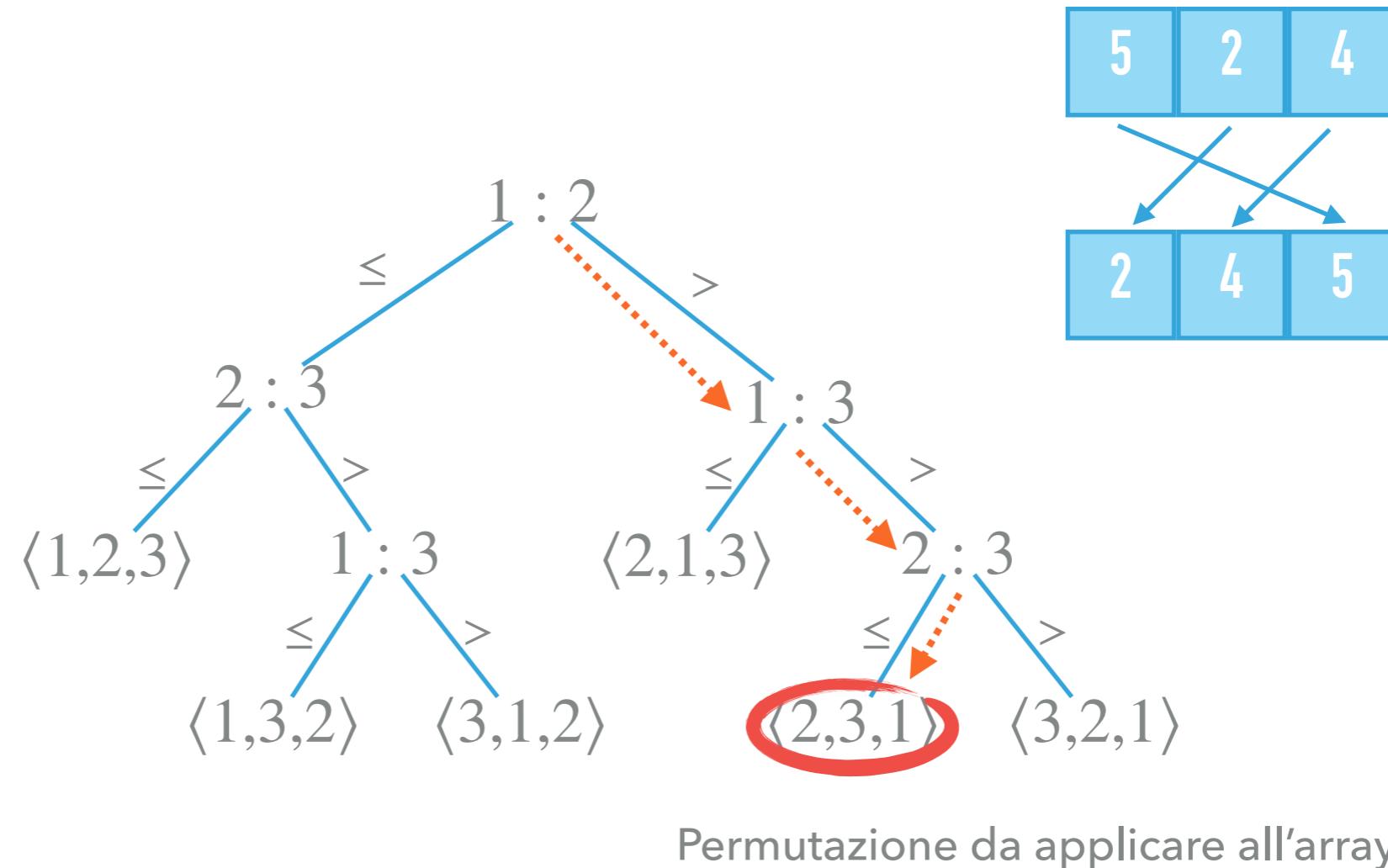


ALBERO DI DECISIONE

| | | |
|---|---|---|
| 5 | 2 | 4 |
|---|---|---|



ALBERO DI DECISIONE



ALBERO DI DECISIONE

- ▶ Dato un qualsiasi algoritmo di ordinamento che usa la comparazione di elementi dell'array per compiere le scelte, possiamo rappresentarlo come un albero di decisione
- ▶ Algoritmi diversi corrispondono ad alberi diversi
- ▶ Dato un array in input la sua esecuzione individuerà un percorso dalla radice ad una delle permutazioni nelle foglie

ALBERO DI DECISIONE

- ▶ Come conseguenza **ogni** permutazione deve essere presente nelle foglie!
- ▶ Le permutazioni di n elementi sono $n!$
- ▶ Quale è l'altezza minima h di un albero con $\ell \geq n!$ foglie?
- ▶ Questa altezza minima ci indica il numero di comparazioni minimo nel caso peggiore che siamo costretti a fare in un algoritmo di ordinamento basato sulla comparazione

ALBERO DI DECISIONE

- ▶ Ricordiamo che un albero di profondità h ha al più 2^h foglie
- ▶ Ne segue $n! \leq \ell \leq 2^h$
- ▶ Prendiamo il logaritmo: $h \geq \log_2(n!)$
- ▶ Mostriamo ora che $\log_2(n!)$ è $\Theta(n \log n)$

ALBERO DI DECISIONE

- ▶ Per approssimazione di Stirling:

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

- ▶ Prendendo il logaritmo dell'approssimazione passiamo da prodotti a somme:

$$\log_2 \sqrt{2\pi n} + \log_2 n^n + \log_2 e^{-n} + \log_2(1 + \Theta(n^{-1}))$$

- ▶ Il termine che domina asintoticamente è $\log_2 n^n = n \log_2 n$, quindi $\log n! = \Theta(n \log n)$

ALBERO DI DECISIONE

- ▶ Abbiamo quindi che $h = \Omega(n \log n)$
(non usiamo $\Theta(n \log n)$ perché h è \geq di $\log_2 n$!)
- ▶ **Teorema.** Ogni algoritmo di ordinamento basato sulla comparazione richiede almeno $\Omega(n \log n)$ comparazioni nel caso peggiore
- ▶ Come conseguenza sia heapsort che mergesort sono ottimali tra gli algoritmi di ordinamento basati sulla comparazione

ORDINAMENTO LINEARE

ORDINARE SENZA COMPARARE

- ▶ Abbiamo visto che ogni algoritmo di ordinamento basato sulla comparazione richiede tempo $\Omega(n \log n)$
- ▶ Il punto chiave è “basato sulla comparazione”
- ▶ Possiamo scrivere algoritmo di ordinamento che non comparano gli elementi tra di loro
- ▶ Questi algoritmi non sono totalmente generici, ma richiedono delle assunzioni aggiuntive

ORDINARE SENZA COMPARARE

- ▶ Che tipo di assunzioni sono?
- ▶ Sappiamo esattamente il range degli numeri da ordinare, $[0, k - 1]$ e $k = O(n)$. **Counting sort**
- ▶ Sappiamo il numero di cifre necessarie a rappresentare i numeri da ordinare. **Radix sort**
- ▶ I numeri da ordinare sono estratti da una distribuzione uniforme. **Bucket sort**

COUNTING SORT

COUNTING SORT

- ▶ Se sappiamo quali sono i valori possibili da ordinare, possiamo contare per ogni valori quanti sono quelli più piccoli e quindi stabilire la posizione finale
- ▶ Se, per esempio abbiamo i numeri da 0 a 10 e sappiamo che ci sono 5 numeri più piccoli di 7, sappiamo che se incontriamo 7 dovremo metterlo in sesta posizione
- ▶ Questo non richiede di confrontare direttamente i numeri

CONTEGGIO DEI VALORI

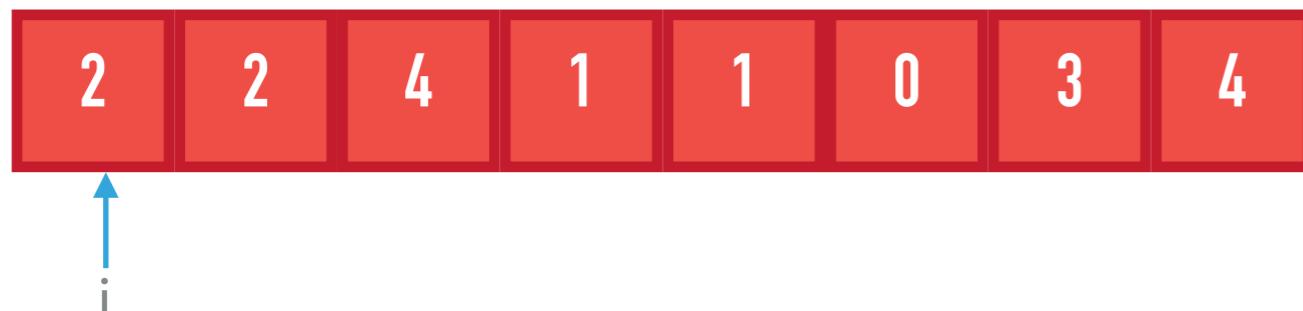


Supponiamo tutti i valori siano in [0,4]



Allotchiamo un array di $k=5$ elementi

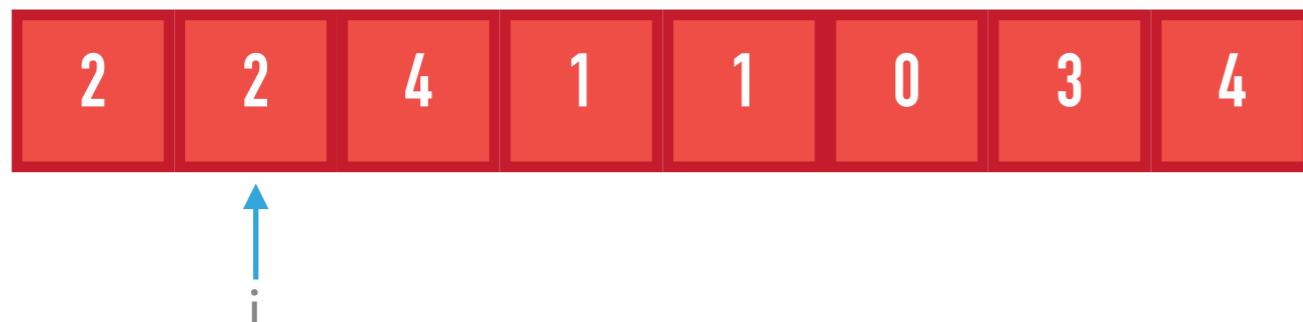
CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



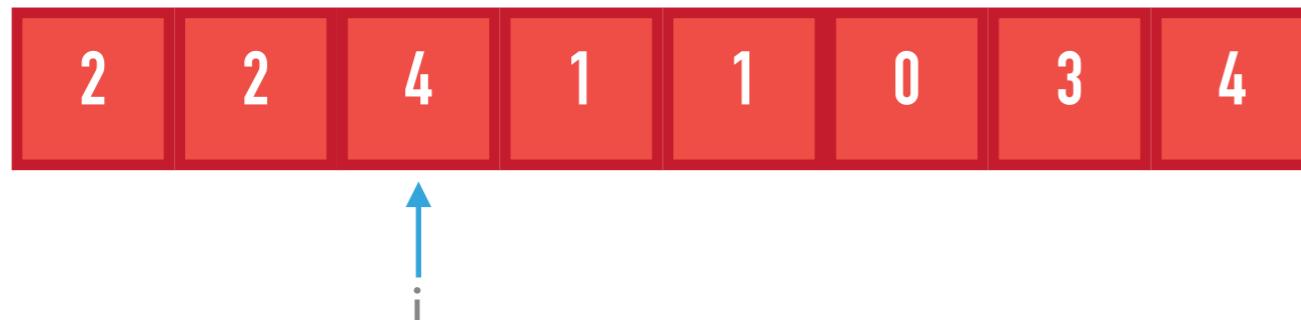
CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



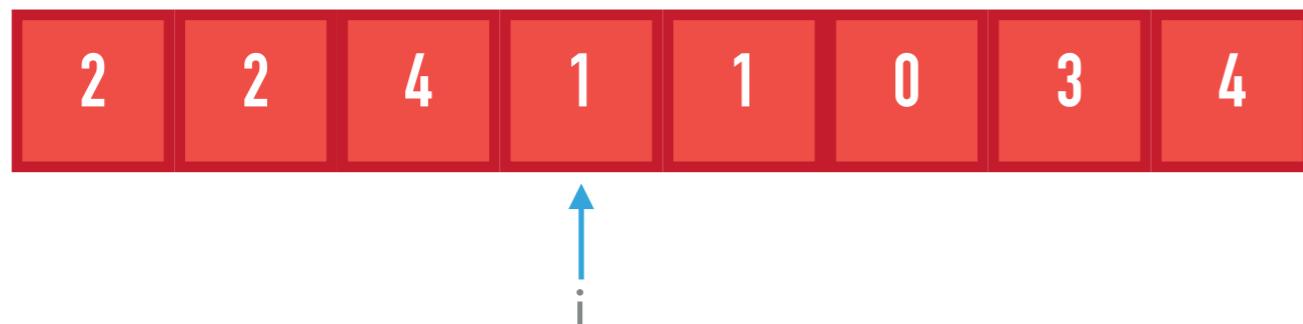
CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]



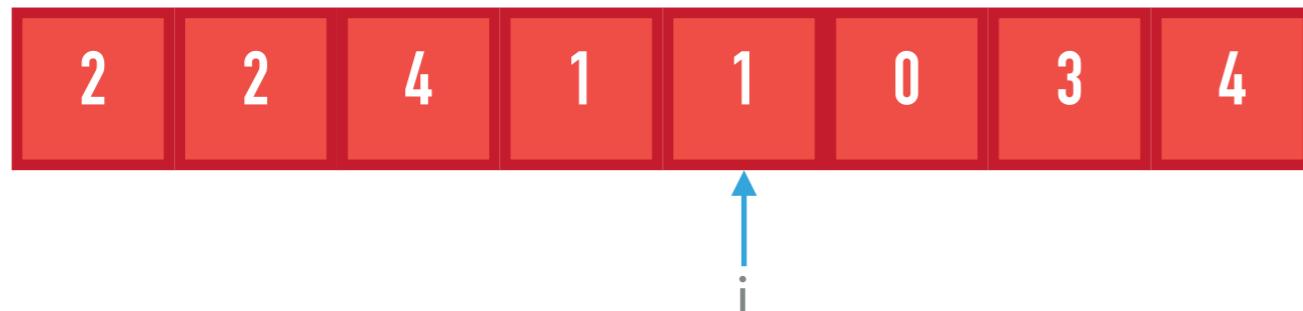
CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



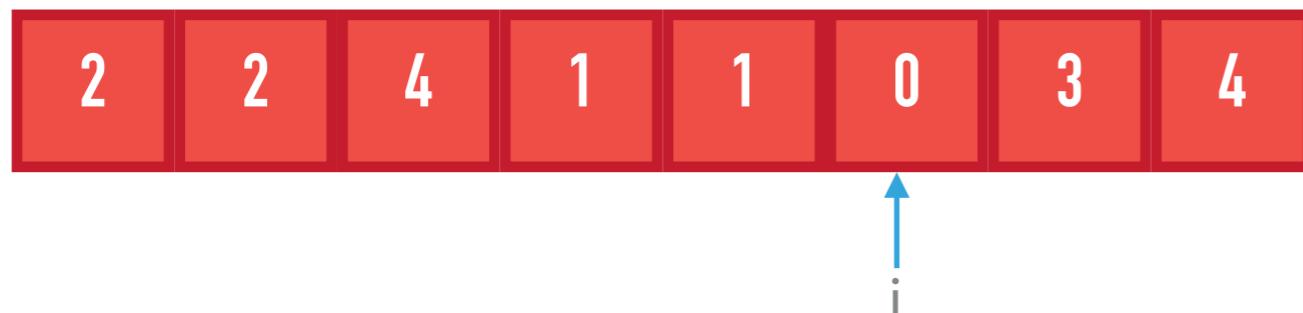
CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]



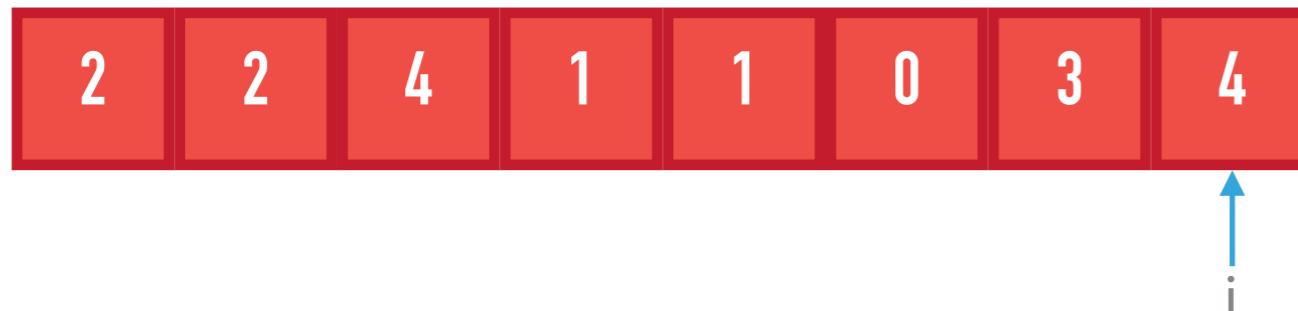
CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in $[0,4]$



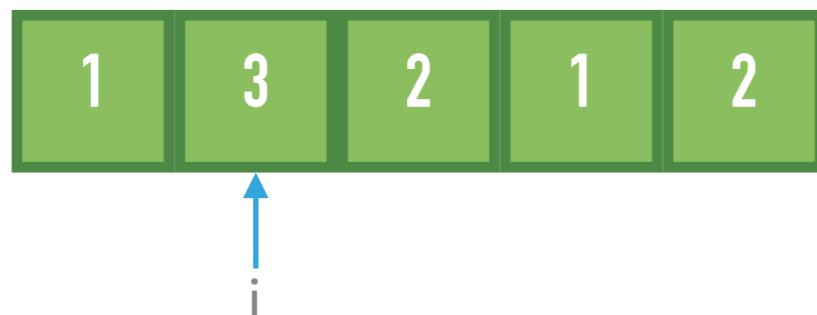
Ora abbiamo a disposizione per ogni valore in $[0,k]$ il numero di occorrenze

Vogliamo contare per ogni valore il numero di elementi **minori o uguali** a quel valore che sono presenti nell'array di partenza

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]

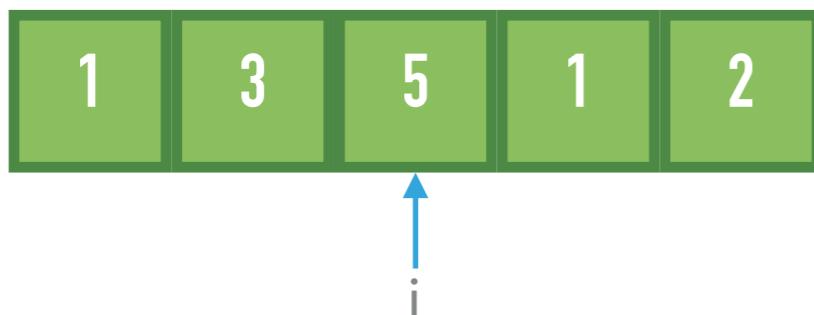


$$B[i] = B[i] + B[i-1]$$

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]



$$B[i] = B[i] + B[i-1]$$

CONTEGGIO DEI VALORI



Supponiamo tutti i valori siano in [0,4]



$$B[i] = B[i] + B[i-1]$$

CONTEGGIO DEI VALORI



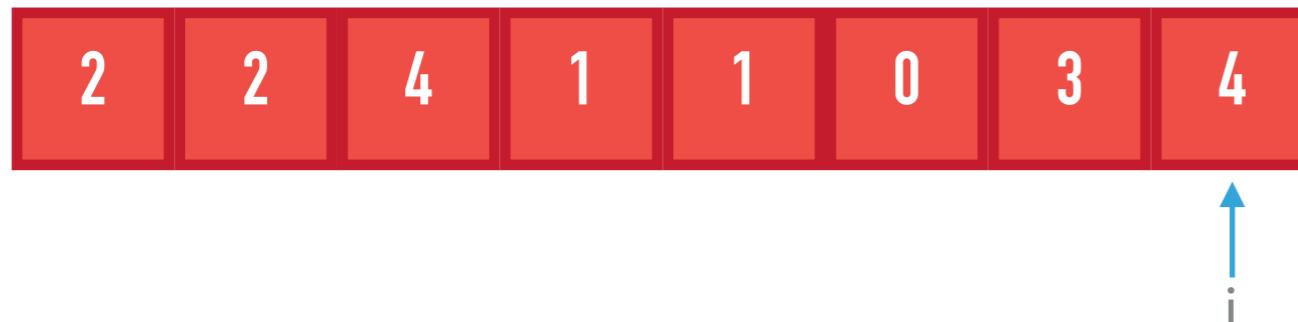
Supponiamo tutti i valori siano in [0,4]



$$B[i] = B[i] + B[i-1]$$

Ora abbiamo in $B[i]$ l'ultima posizione in cui il valore " i " deve apparire

ORDINAMENTO



**QUESTA OPERAZIONE
RICHIEDE UNA SPIEGAZIONE**

$C[B[A[i]]-1] = A[i]$

ORDINAMENTO



↑
i



Elemento da ordinare

$$C[B[A[i]]-1] = A[i]$$

Posizione in cui mettere l'elemento
(indicizzata dall'elemento stesso)

ORDINAMENTO



Supponiamo tutti i valori siano in [0,4]



$$B[A[i]] = B[A[i]] - 1$$



$$C[B[A[i]]-1] = A[i]$$

Scorriamo dal fondo

ORDINAMENTO



Supponiamo tutti i valori siano in [0,4]

Scorriamo dal fondo



$$B[A[i]] = B[A[i]] - 1$$



$$C[B[A[i]]-1] = A[i]$$

ORDINAMENTO



Supponiamo tutti i valori siano in $[0,4]$



Scorriamo dal fondo



$$B[A[i]] = B[A[i]] - 1$$

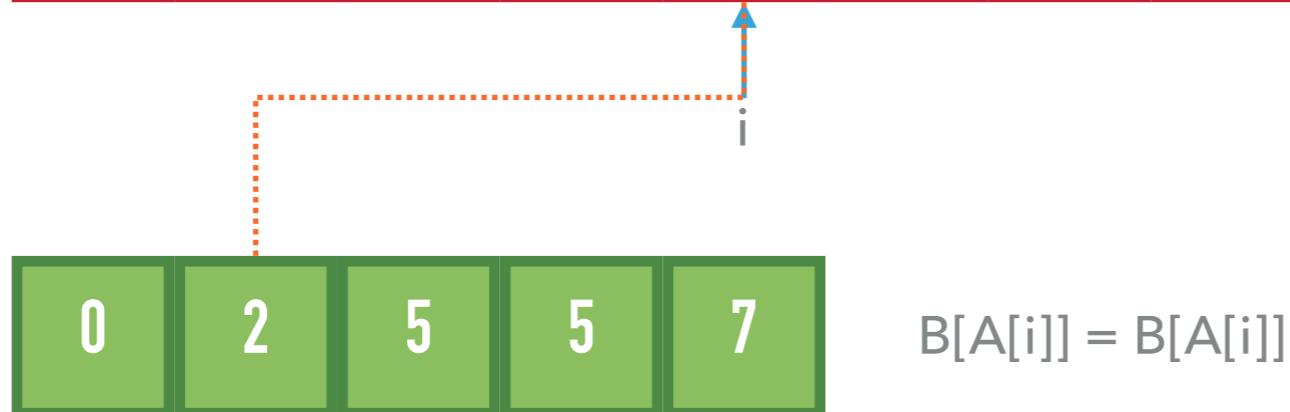


$$C[B[A[i]]-1] = A[i]$$

ORDINAMENTO

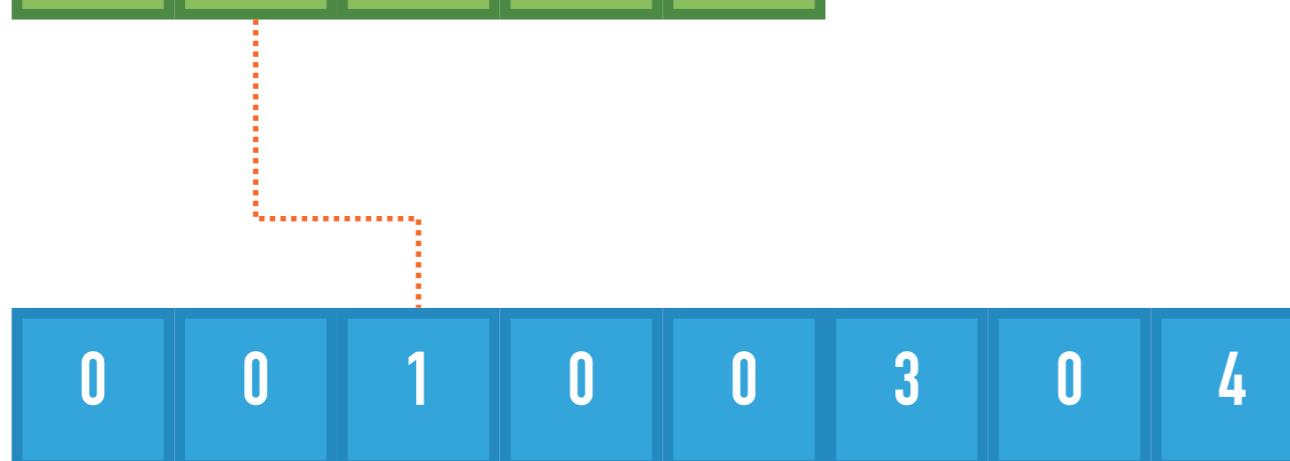


Supponiamo tutti i valori siano in [0,4]



Scorriamo dal fondo

$$B[A[i]] = B[A[i]] - 1$$

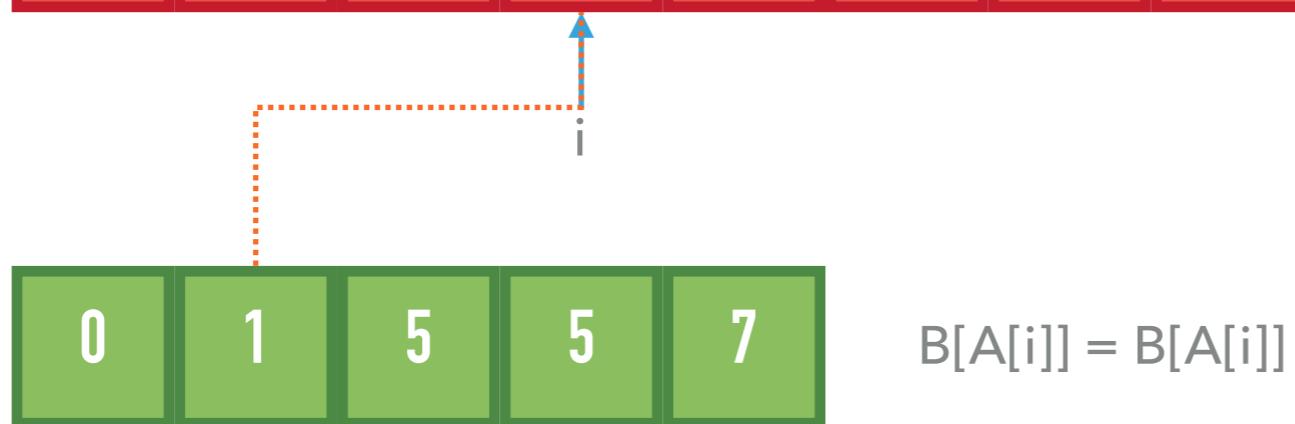


$$C[B[A[i]]-1] = A[i]$$

ORDINAMENTO

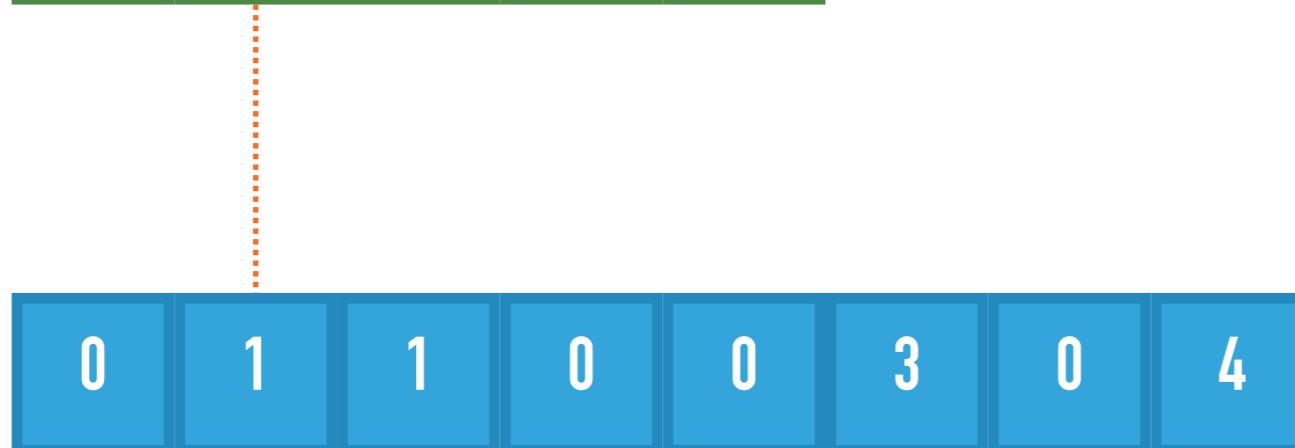


Supponiamo tutti i valori siano in [0,4]



Scorriamo dal fondo

$$B[A[i]] = B[A[i]] - 1$$



$$C[B[A[i]]-1] = A[i]$$

ORDINAMENTO



Supponiamo tutti i valori siano in [0,4]



Scorriamo dal fondo



$$B[A[i]] = B[A[i]] - 1$$

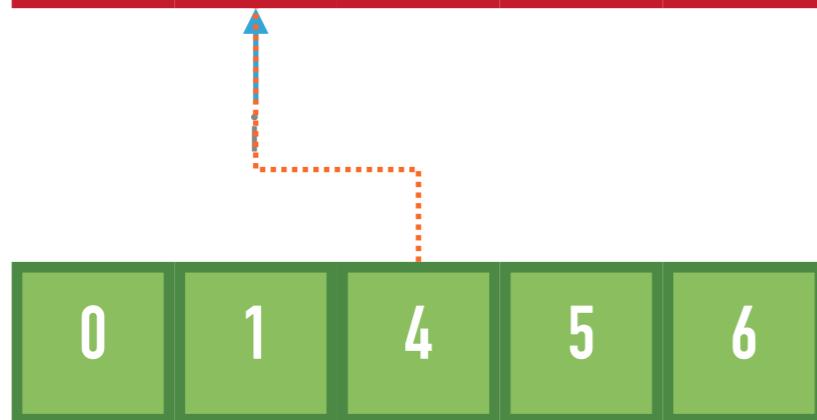


$$C[B[A[i]]-1] = A[i]$$

ORDINAMENTO

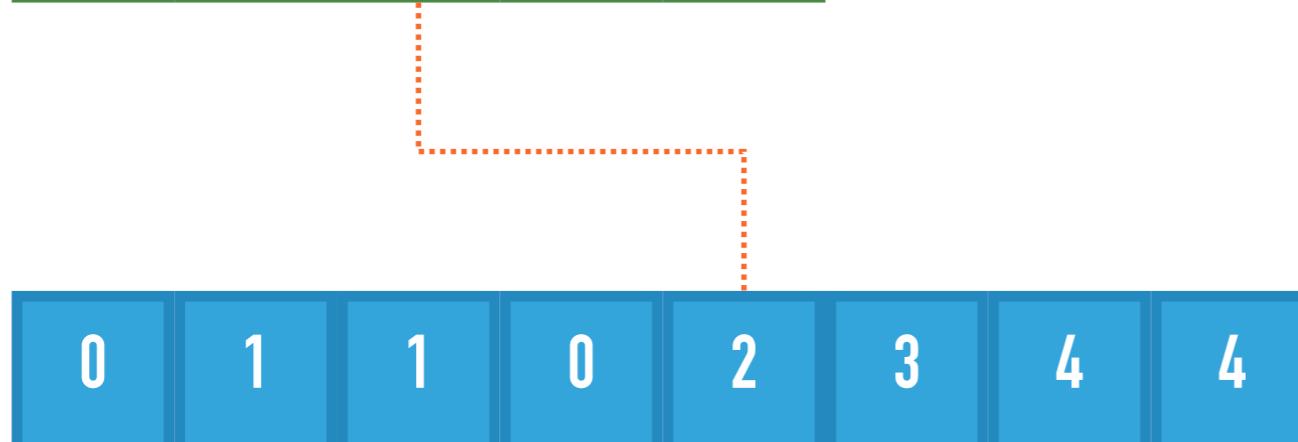


Supponiamo tutti i valori siano in [0,4]



Scorriamo dal fondo

$$B[A[i]] = B[A[i]] - 1$$



$$C[B[A[i]]-1] = A[i]$$

ORDINAMENTO



Supponiamo tutti i valori siano in [0,4]



$B[A[i]] = B[A[i]] - 1$



$C[B[A[i]]-1] = A[i]$



Scorriamo dal fondo



$B[A[i]] = B[A[i]] - 1$



$C[B[A[i]]-1] = A[i]$

COUNTING SORT: PSEUDOCODICE

Argomenti: A (array), k (valore massimo)

B = array di k elementi inizialmente zero

C = array di len(A) elementi

for i in range(0, len(A))

 B[A[i]] = B[A[i]] + 1 # incrementa il numero di valori A[i] trovati

for i in range(1,k)

 B[i] = B[i] + B[i-1] # in modo da avere in B[i] il numero di elementi $\leq i$

for i in range(len(A), -1, -1) # contiamo dalla fine all'inizio

 C[B[A[i]]-1] = A[i] # trasferiamo A[i] nella sua posizione in C

 B[A[i]] = B[A[i]] - 1

return C

COUNTING SORT: COMPLESSITÀ

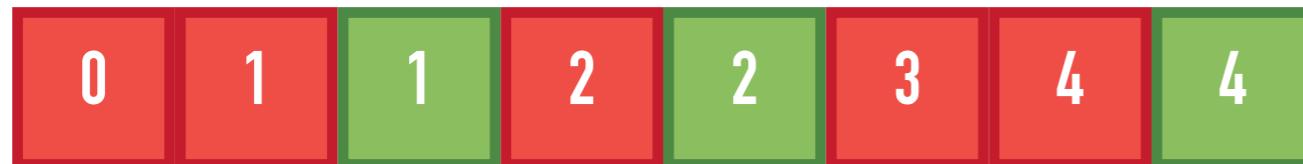
- ▶ Questa volta la complessità è funzione di **due** parametri: n (la dimensione dell'array) e k (il numero di valori distinti)
- ▶ Due cicli for che sono lunghi n cicli
- ▶ Un ciclo for che è lungo $k - 1$ cicli
- ▶ Risultato: $\Theta(n + k)$
- ▶ Se abbiamo che $k = O(n)$ otteniamo che l'algoritmo richiede tempo $\Theta(n)$

ORDINAMENTO STABILE

- ▶ Possiamo fare una distinzione aggiuntiva tra ordinamenti stabili e non stabili
- ▶ Un ordinamento stabile preserva l'ordine relativo di elementi con lo stesso valore



Ordiniamo solo guardando i numeri
ma le celle verdi vengono sempre dopo
le celle rosse con lo stesso valore



Un ordinamento stabile preserva
questa proprietà



Un ordinamento non stabile potrebbe
non preservarla

ORDINAMENTO STABILE

- ▶ In diversi casi la stabilità dipende da dettagli implementativi, in particolare da come sono trattati i valori identici
- ▶ In particolare a noi servirà avere il counting sort stabile
- ▶ La nostra implementazione lo è...
- ▶ ...perché inseriamo i valori a partire dal fondo
- ▶ Se avessi cambiato l'ordine di inserimento non sarebbe stato stabile

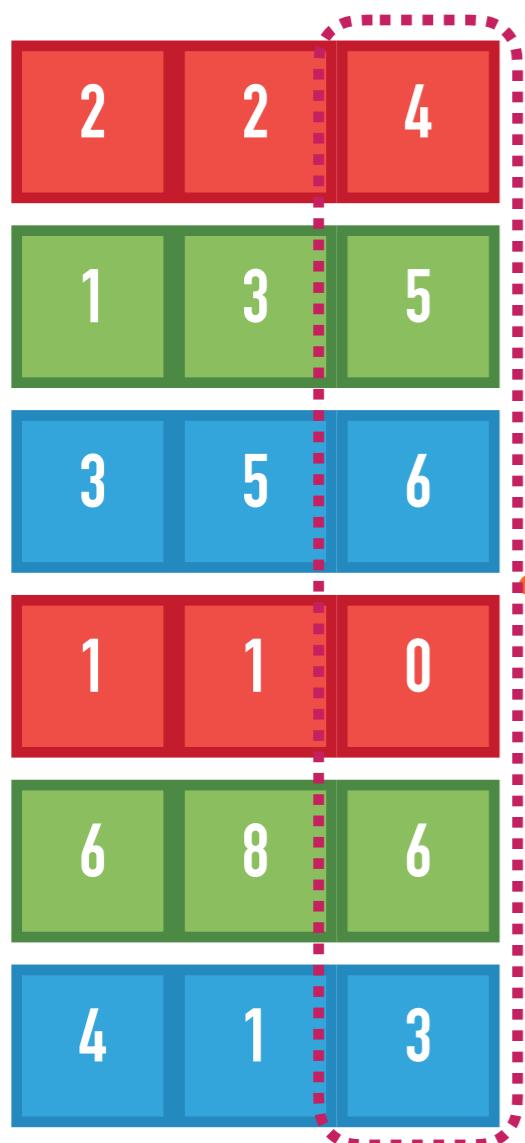
RADIX SORT

RADIX SORT

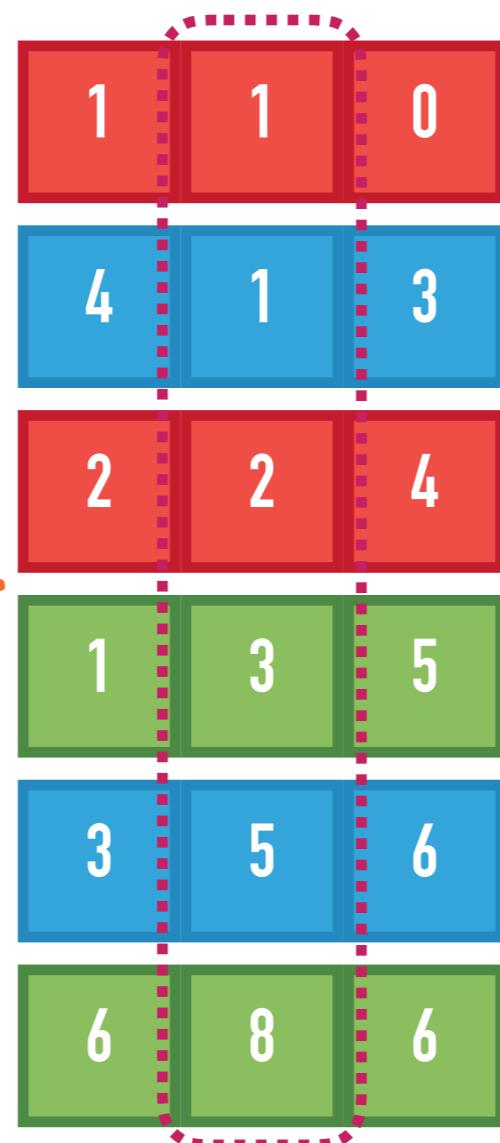
- ▶ Il radix sort richiede un ordinamento stabile
- ▶ È nato per ordinare le schede perforate, quindi le prime implementazioni di radix sort non erano con il codice, ma con strumenti meccanici!
- ▶ Idea di base: se abbiamo numeri di d cifre possiamo ordinarli una cifra alla volta usando un algoritmo di ordinamento stabile

ESEMPIO DI RADIX SORT

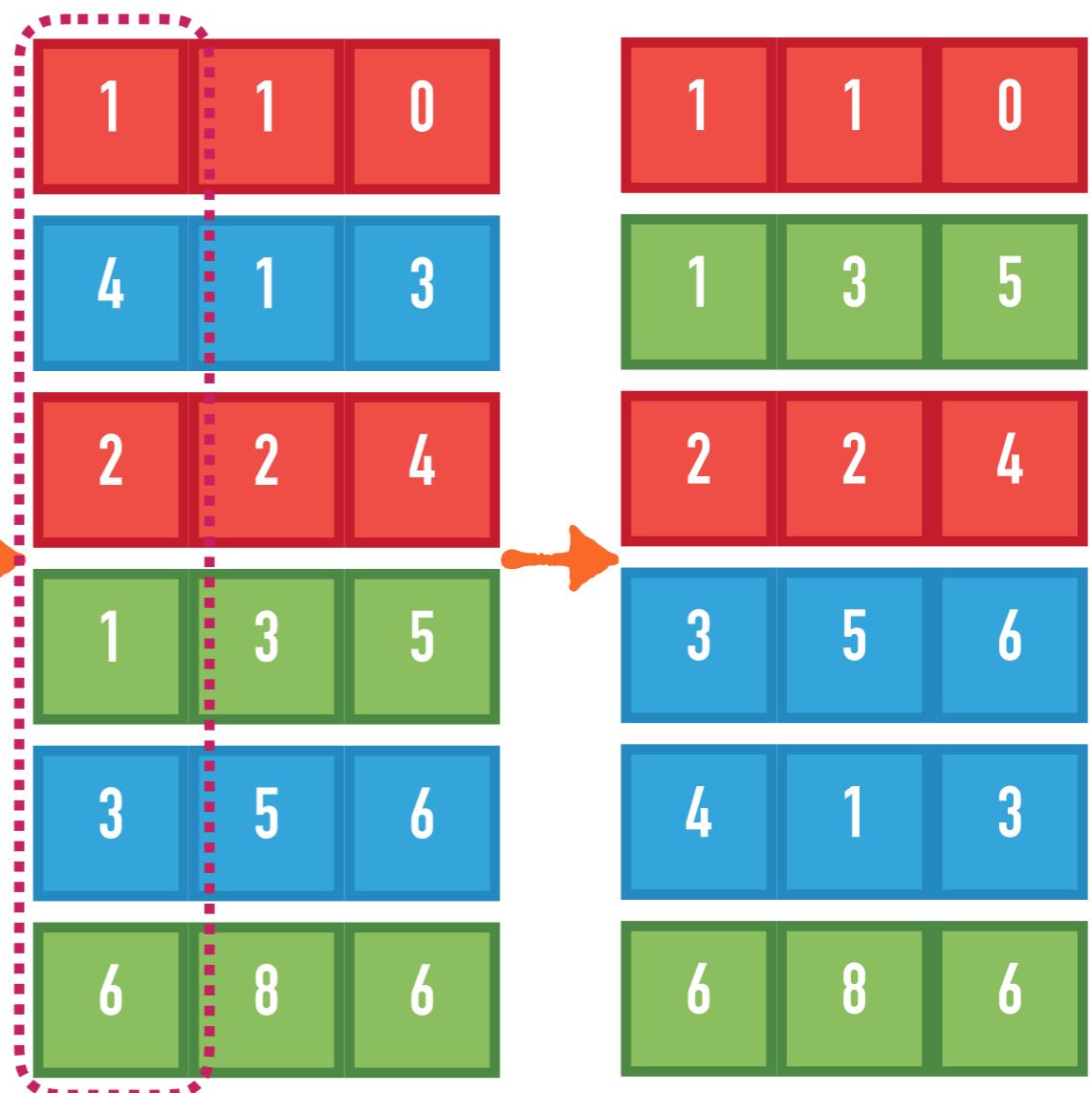
Ordiniamo rispetto
all'ultima cifra



Ordiniamo rispetto
alla penultima cifra



Ordiniamo rispetto
alla prima cifra



RADIX SORT: PSEUDOCODICE

Argomenti: A (array), k (cifre), d (numero di cifre)

```
for i in range(d-1,-1,-1) # dall'ultima alla prima cifra  
    ordina stabilmente rispetto alla cifra i-esima
```

- ▶ Dato che le possibili cifre sono limitate, la scelta dell'ordinamento stabile è solitamente il **counting sort**

RADIX SORT: PERCHÉ FUNZIONA?

- ▶ Dopo aver ordinato per la cifra meno significativa abbiamo tutti i numeri che terminano con 0 che precedono quelli che terminano con 1, etc.
- ▶ Dopo aver ordinato rispetto alla penultima cifra abbiamo tutti i numeri che terminano con 00 che precedono quelli che terminano con 01, ..., 10, 11, ..., 99
- ▶ In questo punto è importante che l'ordinamento sia stabile, altrimenti non è detto che sia preservato l'ordine relativo sull'ultima cifra!
- ▶ Finito di ordinare su tutte le d cifre abbiamo che gli elementi risultano ordinati

RADIX SORT: COMPLESSITÀ

- ▶ Questa volta abbiamo tre parametri: n , k e d
- ▶ Se usiamo counting sort all'interno del ciclo for, ogni ordinamento stabile rispetto ad una cifra ha costo $\Theta(n + k)$
- ▶ Dato che dobbiamo ripetere questa procedura per d cifre, la complessità temporale risultante è $\Theta(d(n + k))$
- ▶ Quando k è $O(n)$ – o addirittura costante, abbiamo $\Theta(dn)$
- ▶ Quindi dipende tutto da quante cifre abbiamo, per esempio se d è costante, radix sort richiede tempo $\Theta(n)$

BUCKET SORT

BUCKET SORT

- ▶ Bucket sort richiede che l'input sia distribuito uniformemente in un intervallo. Noi useremo $[0,1)$
- ▶ Sotto questa ipotesi il tempo medio richiesto da bucket sort è $\Theta(n)$
- ▶ L'idea è di usare, per un array di n elementi, n "secchi" distinti in cui inserire i valori
- ▶ Ordiniamo ciascuno dei "secchi" con insertion sort
- ▶ Concateniamo ciascuno dei "secchi" in ordine

ORDINAMENTO



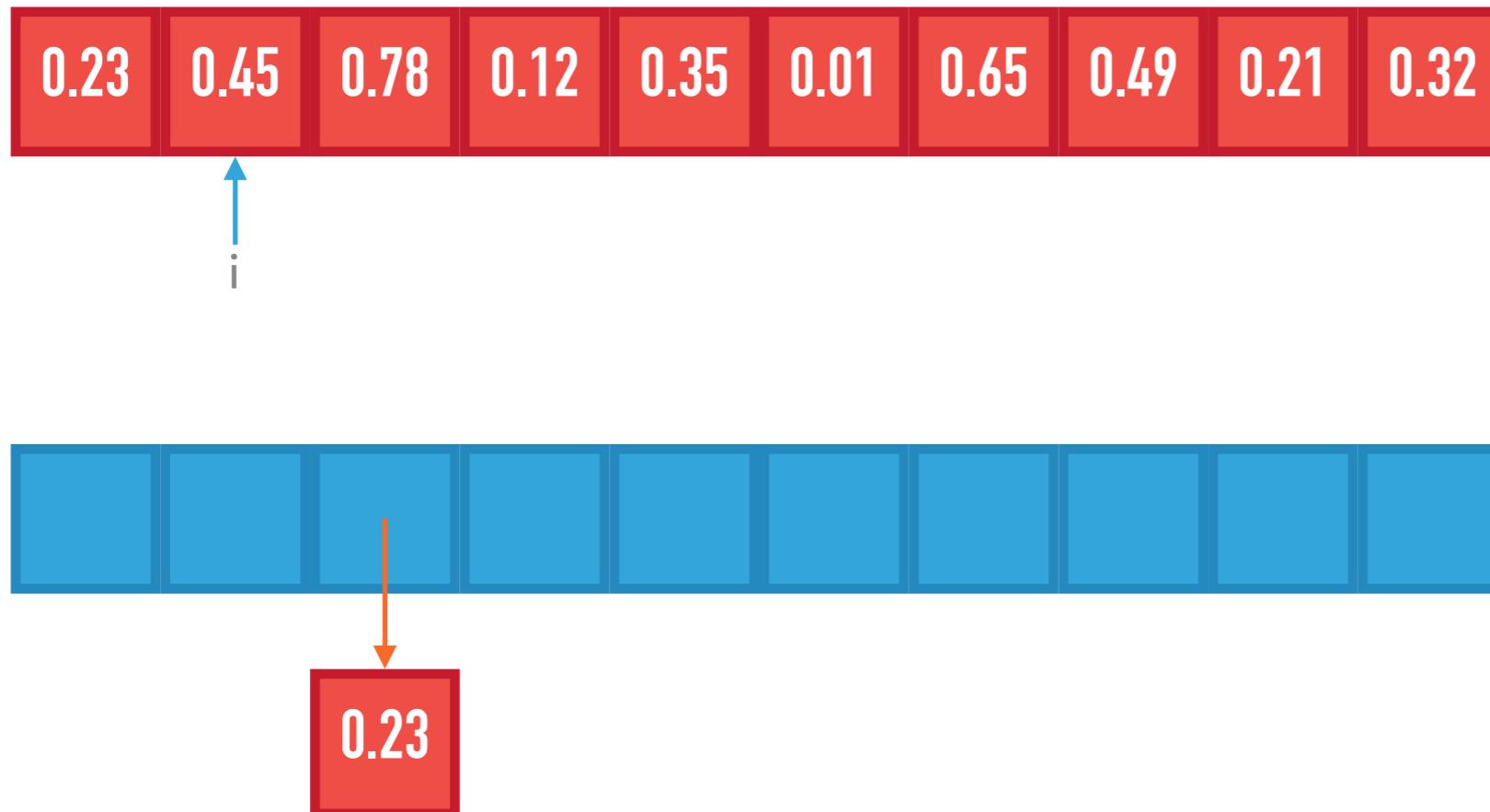
Conterrà i numeri tra 0 (incluso) e 0.1 (escluso)



Array di n liste vuote

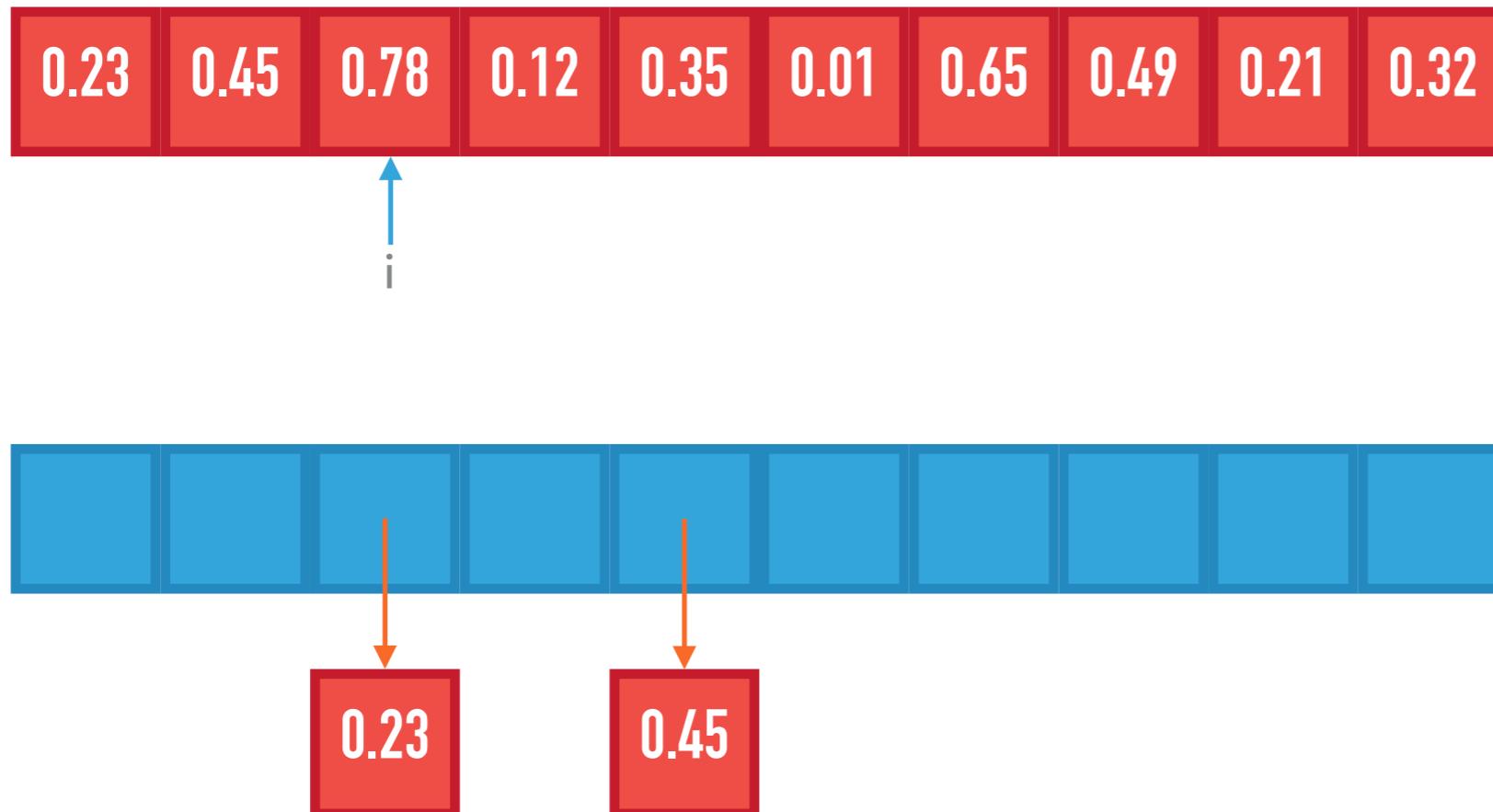
Conterrà i numeri tra 0.7 (incluso) e 0.8 (escluso)

ORDINAMENTO

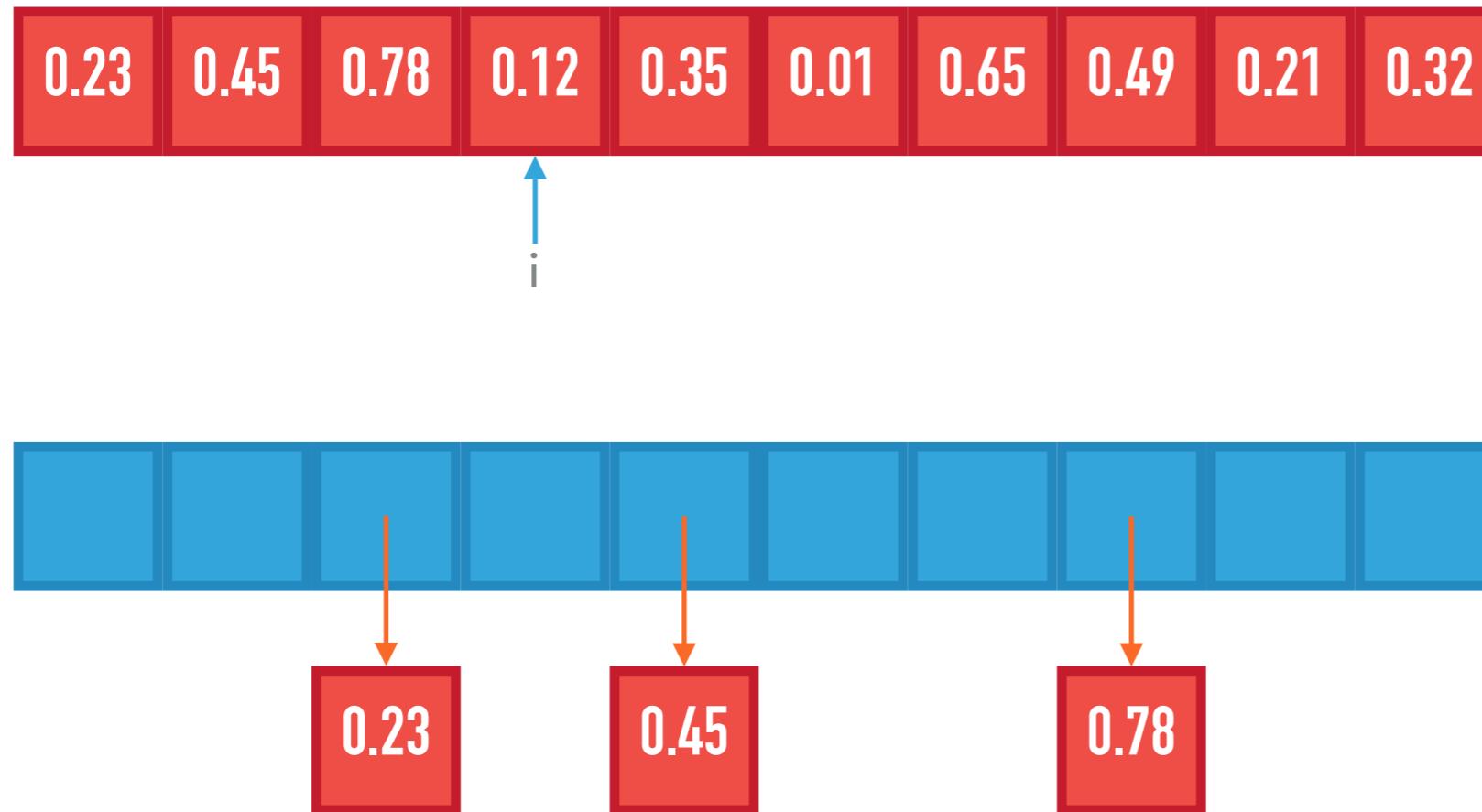


Array di n liste vuote

ORDINAMENTO

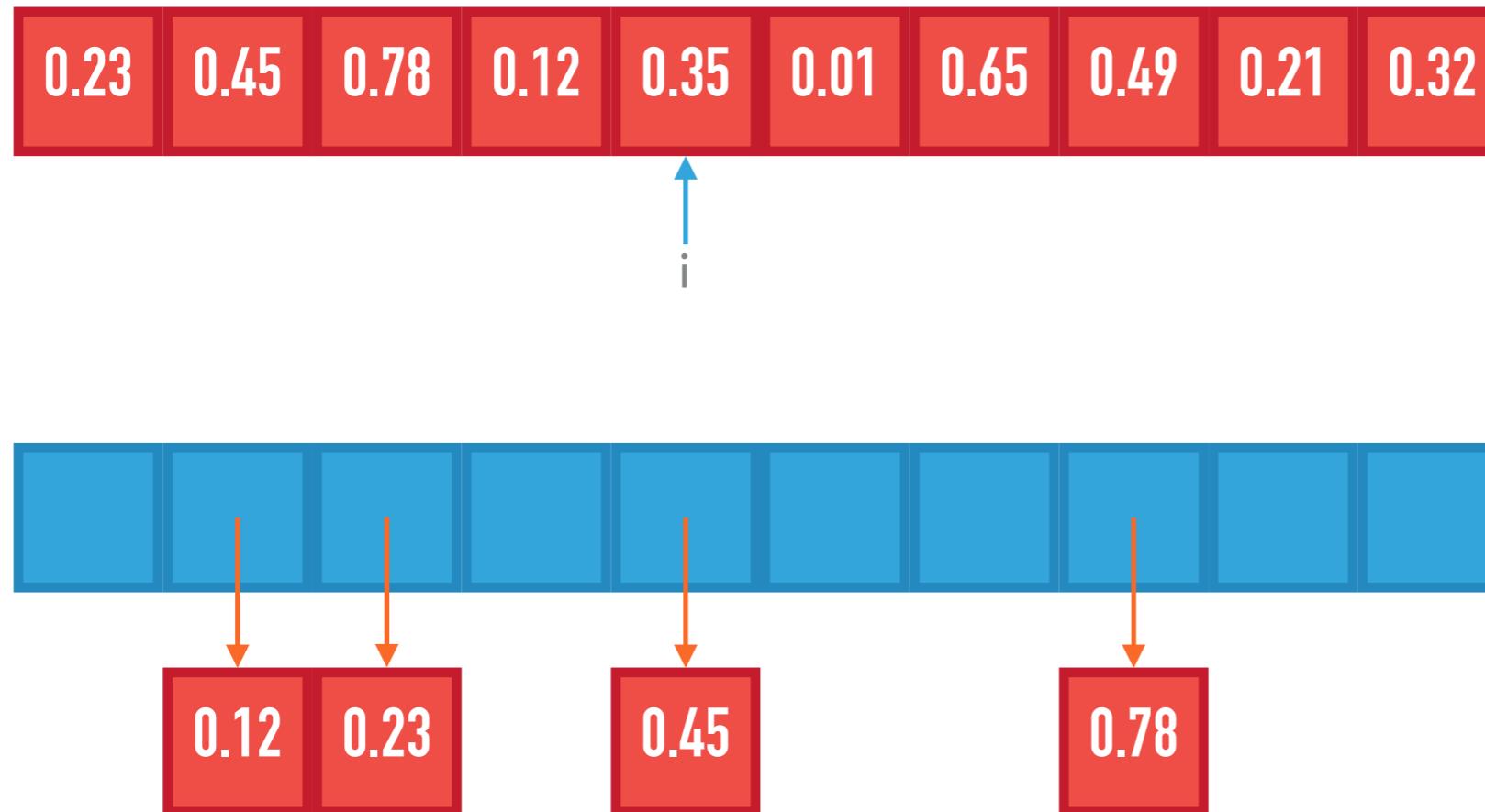


ORDINAMENTO



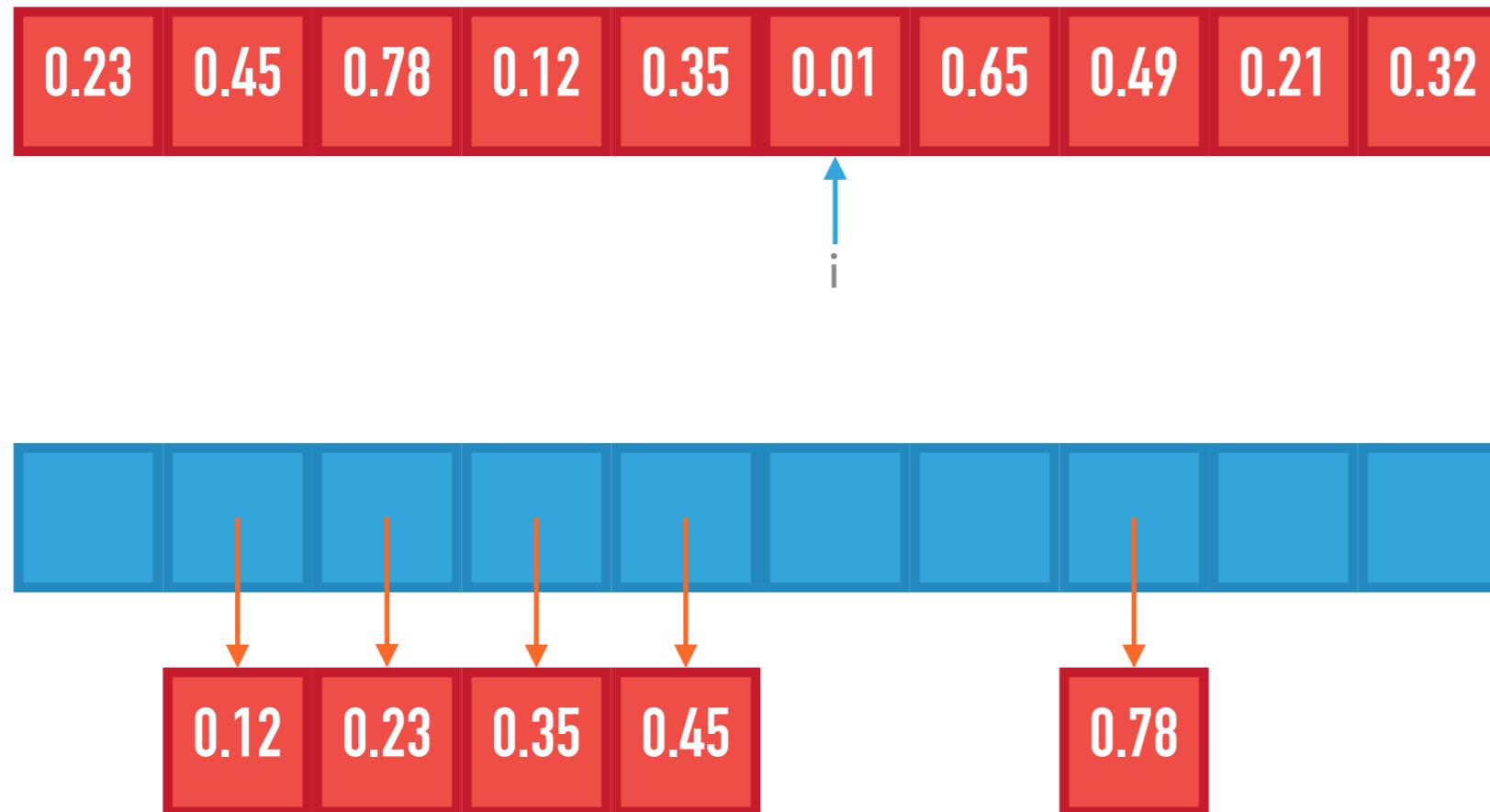
Array di n liste vuote

ORDINAMENTO



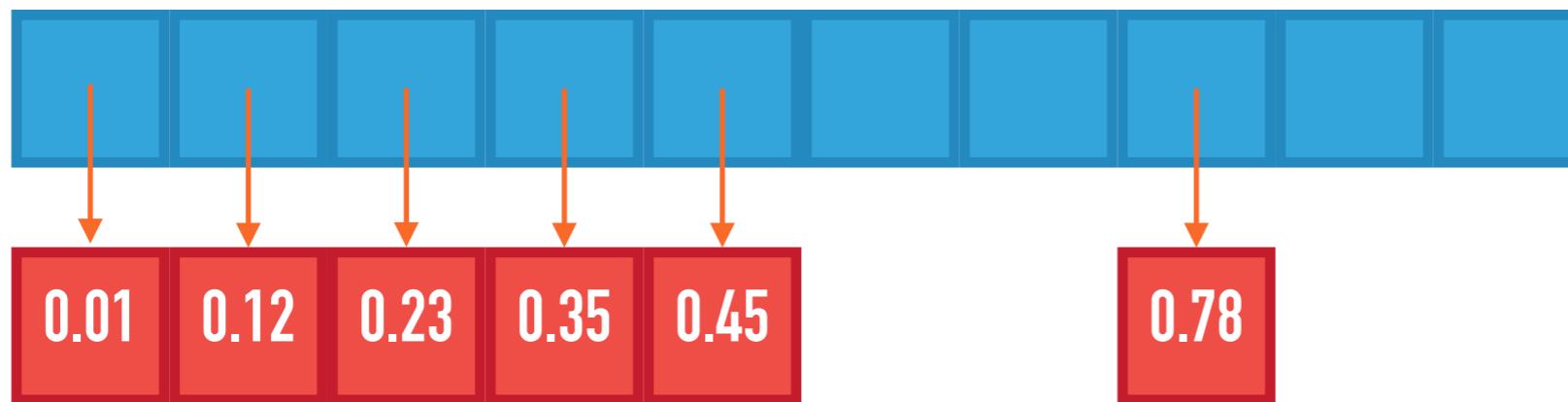
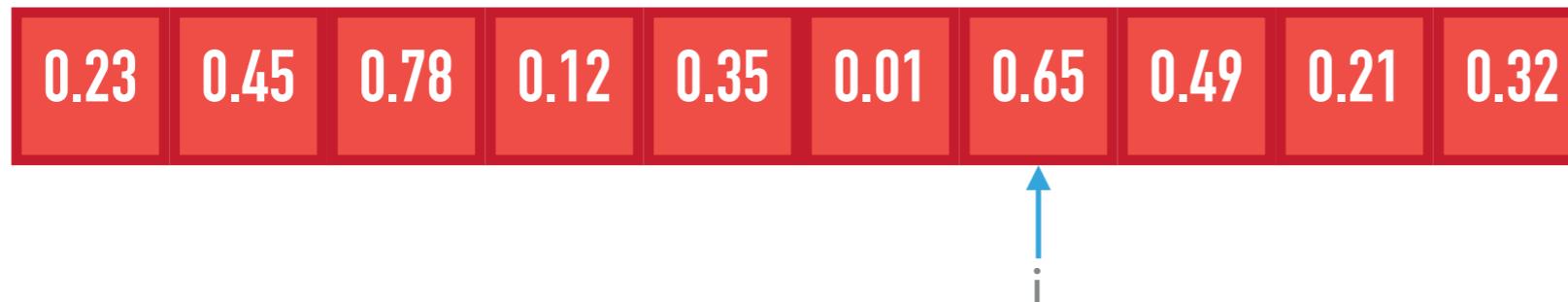
Array di n liste vuote

ORDINAMENTO



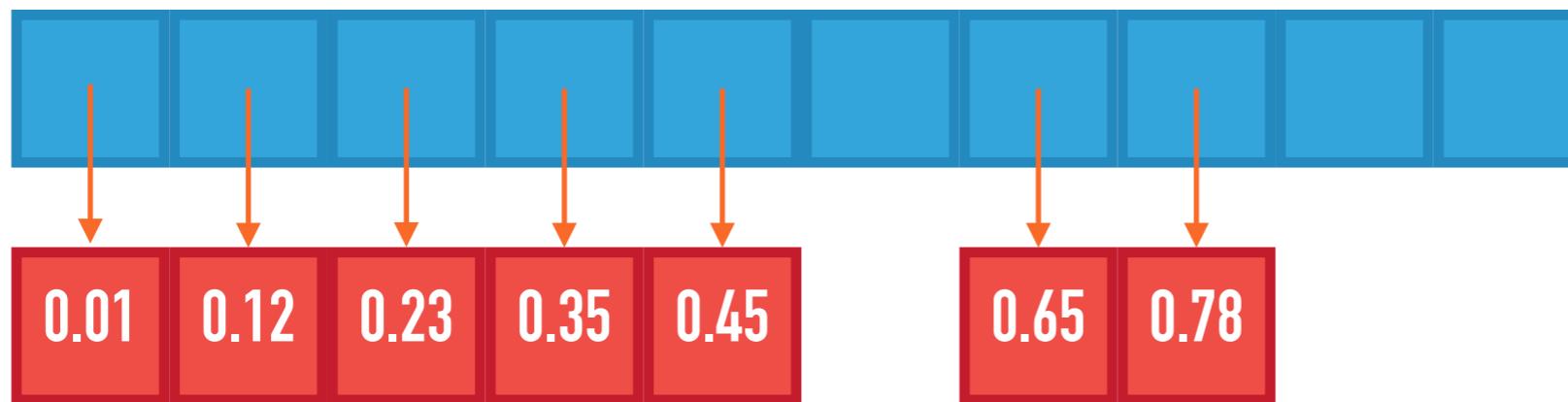
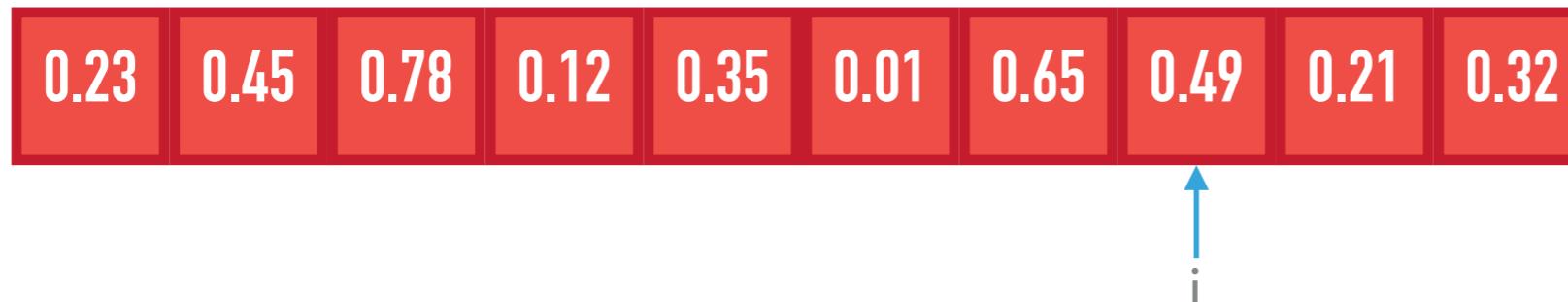
Array di n liste vuote

ORDINAMENTO



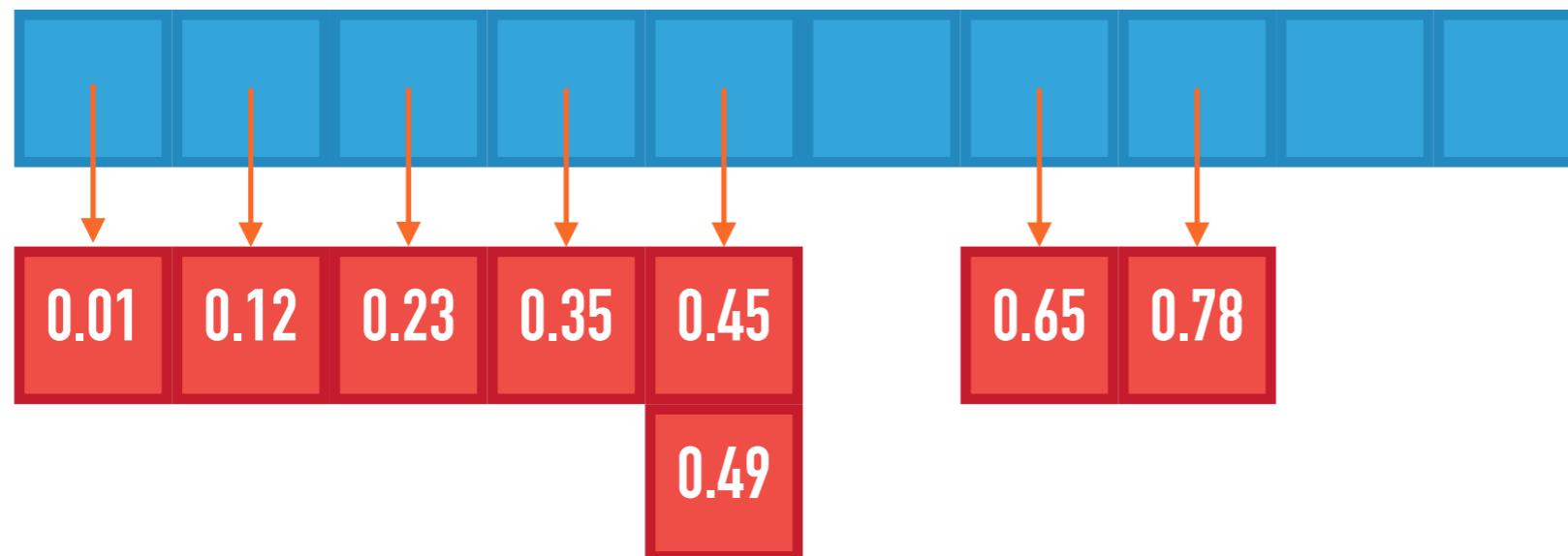
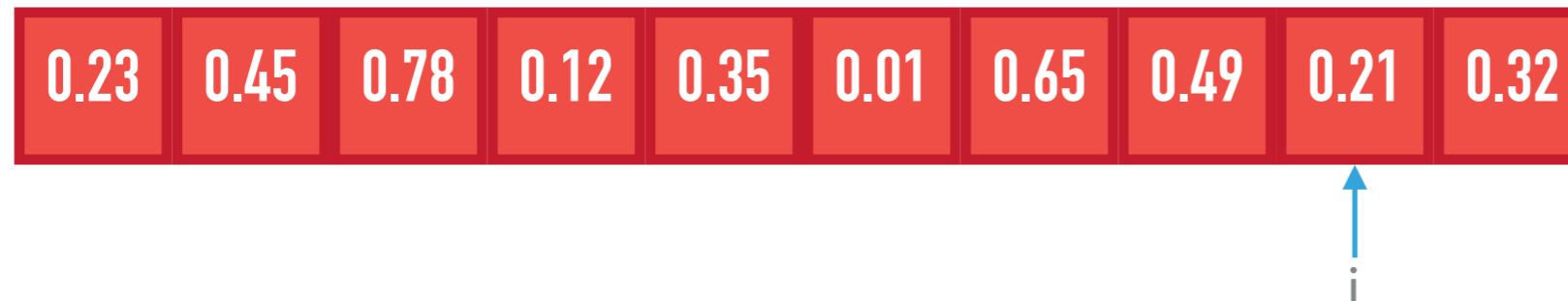
Array di n liste vuote

ORDINAMENTO



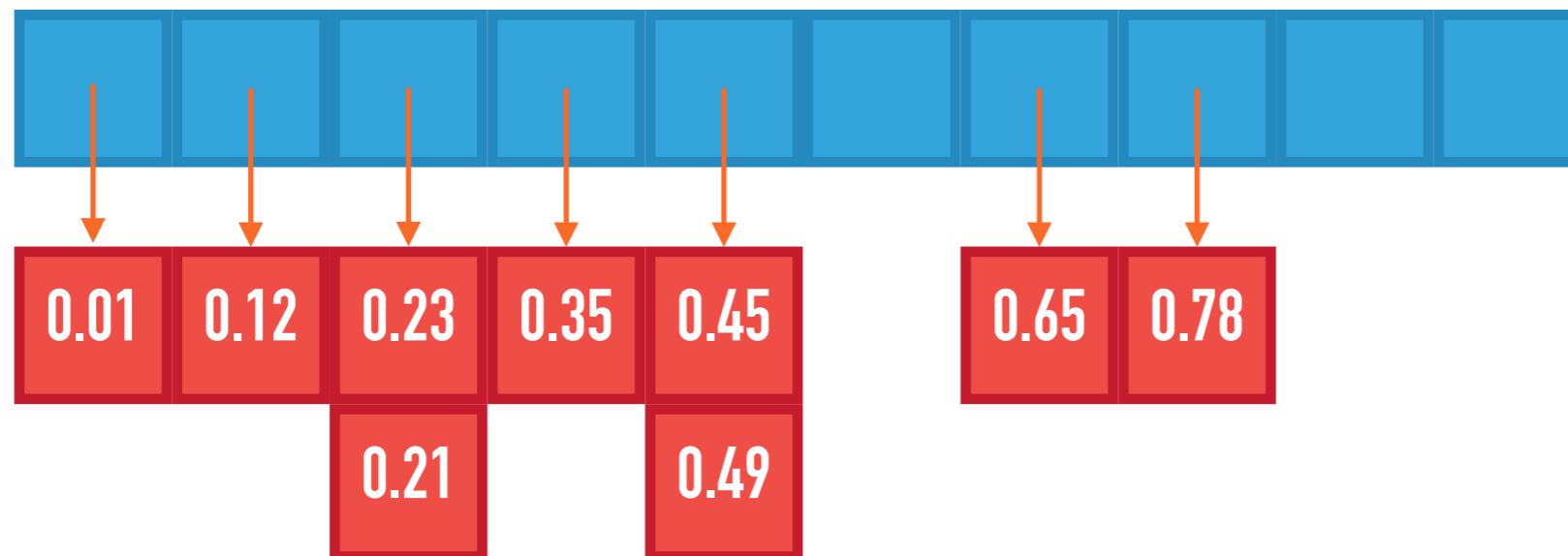
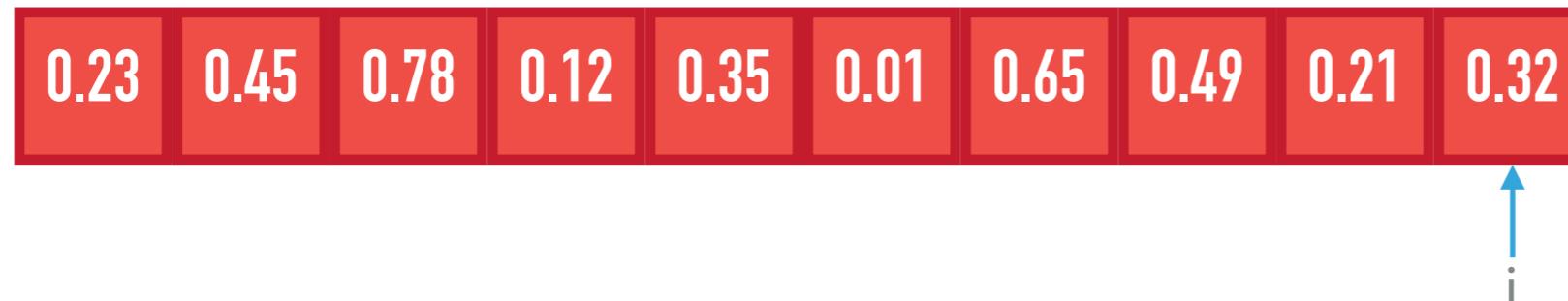
Array di n liste vuote

ORDINAMENTO



Array di n liste vuote

ORDINAMENTO



Array di n liste vuote

ORDINAMENTO



Ora dobbiamo ordinare i singoli bucket usando, per esempio, insertion sort



ORDINAMENTO



L'ordinamento è rapido perché nella maggior parte dei casi i bucket contengono pochi elementi



Ora passando i bucket da sinistra a destra e in ordine all'interno degli stessi otteniamo l'array ordinato

BUCKET SORT: PSEUDOCODICE

- Argomenti: A (array)
 - Alloca un array B di n liste vuote
 - for i in range(0 , len(A))
 - Aggiungi $A[i]$ a $B[\lfloor n \times A[i] \rfloor]$
 - for i in range(0 , len(B))
 - Ordina $B[i]$ con insertion sort
 - Concatena $B[0]$, $B[1]$, ... $B[n-1]$
-
- ▶ Senza andare troppo nei dettagli del tempo di calcolo, se la distribuzione è uniforme, il contenuto della maggior parte dei bucket sarà ridotto e quindi l'ordinamento rapido.
 - ▶ In particolare si trova che il tempo atteso è $\Theta(n)$

CALCOLO DEI PERCENTILI
CALCOLO DELLE MEDIANE
STIMA DELLA DENSITÀ

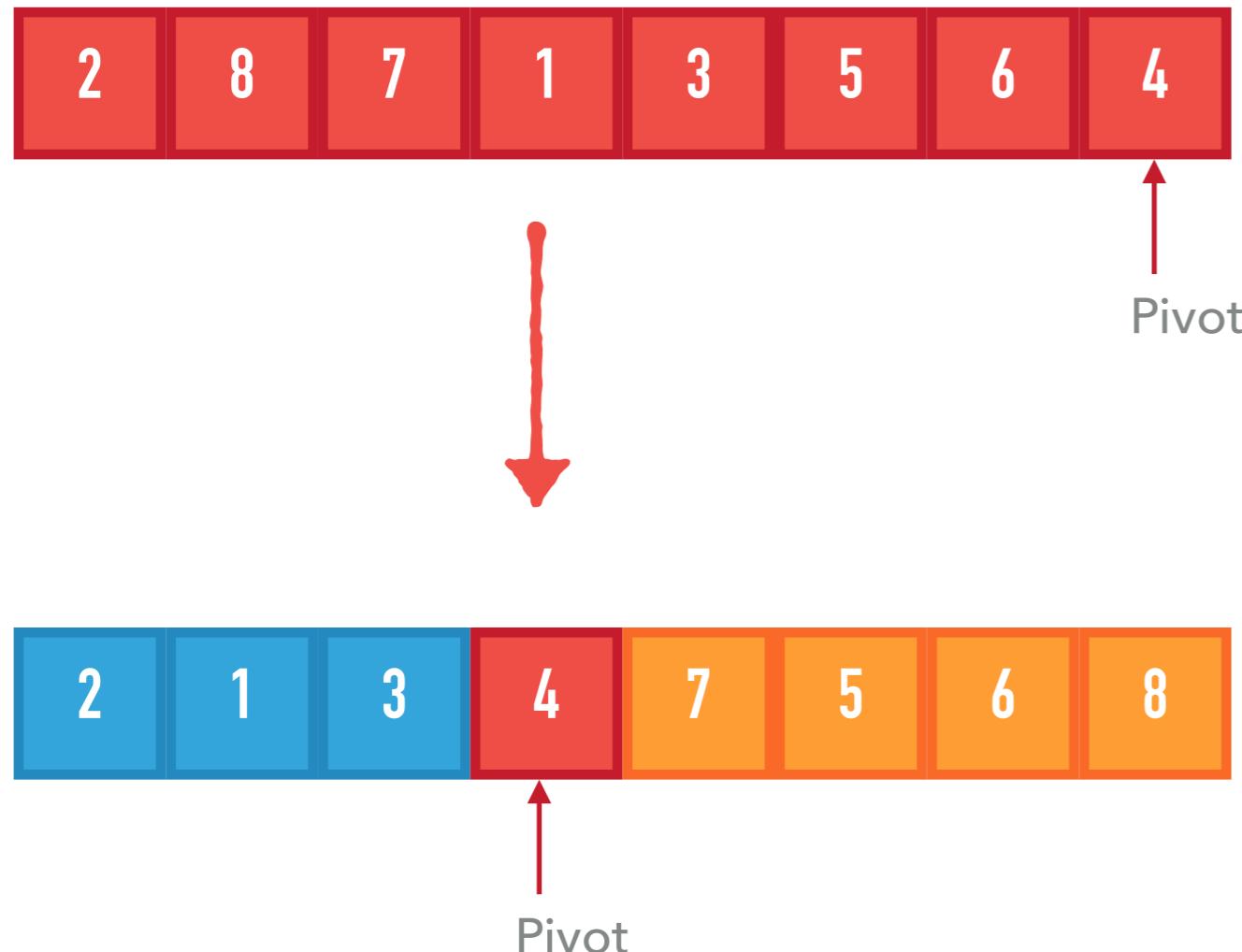
ALGORITMI E STRUTTURE DATI

PERCENTILI

PERCENTILI

- ▶ Dato un array A di numeri (o elementi totalmente ordinabili), il p-esimo percentile è quel numero $A[j]$ tale che $p\%$ dei rimanenti numeri dell'array sono minori di $A[j]$.
- ▶ Il 50-simo percentile è la **mediana**.
- ▶ Un modo per trovare il p-simo percentile è ordinare l'array in senso crescente e ritornare $A[j]$, con $j = \left\lceil \frac{p}{100} \text{len}(A) \right\rceil$
- ▶ Il costo di questo approccio è $O(n \log n)$.
- ▶ Trovare il massimo o il minimo costa $O(n)$. Esiste un metodo migliore per calcolare un singolo percentile?

QUICKSELECT



Supponiamo di cercare il 75simo percentile p.

Lanciamo Partition.
Dove si trova p rispetto al pivot?

QUICKSELECT

```
QuickSelect(A, l, r, p)
  if l = r return A[l]
  q = Partition(A, l, r)
  i = q-l+1 //p è relativo a r-l+1, q a len(A)
  if p=i return A[q]
  else if p < i return QuickSelect(A, l, q-1, p)
  else return QuickSelect(A, q, r, p-i)
```

Correttezza: per induzione

Ci sono tre casi da considerare rispetto alla relazione tra p e $i = q-l+1$:

- $p < i$: il percentile sta tra l e $q-1$
- $p = i$: il percentile è proprio $A[q]$
- $p > i$: il percentile sta tra $q+1$ e r .

QUICKSELECT - COMPLESSITÀ

```
QuickSelect(A, l, r, p)
    if l = r return A[l]
    q = Partition(A, l, r)
    i = q-l+1 //p è relativo a r-l+1, q a len(A)
    if p=i return A[q]
    else if p < i return QuickSelect(A, l, q-1, p)
    else return QuickSelect(A, q, r, p-i)
```

Caso peggiore

Pivot ritorna una partizione in due array di lunghezza 0 e n-1,
ed il percentile sta in quello più lungo:

$$T(n) = T(n-1) + O(n) \Rightarrow T(n) = O(n^2)$$

QUICKSELECT - COMPLESSITÀ MEDIA

Caso ottimo

Pivot ritorna una partiziona in due array di lunghezza $n/2$:

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(n) \Rightarrow T(n) = \Theta(n)$$

Caso medio

Assumiamo la variante randomized-select (pivot scelto a caso).

Partition ritorna due partizioni di lunghezza k e $n-k-1$ con probabilità $1/n$.

Assumiamo il percentile stia sempre in quella più lunga.

$$T(n) \leq \frac{1}{n} \left(\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} T(k) \right) + \Theta(n)$$

QUICKSELECT - COMPLESSITÀ MEDIA

Siamo ottimisti ed assumiamo che la complessità media vada come quella ottima. Dimostriamo per sostituzione che $T(n) \leq cn$ per qualche $c > 0$.

$$T(n) \leq \frac{1}{n} \left(\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} ck \right) + dn$$

$$\frac{1}{n} \left(\sum_{k=\lceil \frac{n}{2} \rceil}^{n-1} ck \right) + dn \leq \frac{c}{n} \left(\sum_{k=1}^{n-1} k \right) + dn = \frac{c}{2n} (n^2 - n) + dn = \frac{cn}{2} - \frac{c}{2} + dn$$

Segue $T(n) \leq cn$ per $c > 2d$

MEDIAN OF MEDIAN

Can one find a partition scheme that has worst case running time $O(n)$?

1. Divide the n elements of the input array into $\lfloor n/5 \rfloor$ groups of 5 elements each and at most one group made up of the remaining $n \bmod 5$ elements.
2. Find the median of each of the $\lceil n/5 \rceil$ groups by first insertion-sorting the elements of each group (of which there are at most 5) and then picking the median from the sorted list of group elements.
3. Use SELECT recursively to find the median x of the $\lceil n/5 \rceil$ medians found in step 2. (If there are an even number of medians, then by our convention, x is the lower median.)
4. Partition the input array around the median-of-medians x using the modified version of PARTITION. Let k be one more than the number of elements on the low side of the partition, so that x is the k th smallest element and there are $n - k$ elements on the high side of the partition.
5. If $i = k$, then return x . Otherwise, use SELECT recursively to find the i th smallest element on the low side if $i < k$, or the $(i - k)$ th smallest element on the high side if $i > k$.

Why this works?

Intuitively, the approximate median found is always between the 30th and the 70th percentile: half medians of groups and 2 other elements of these groups are smaller

MEDIAN OF MEDIAN

Esiste anche un algoritmo che ha complessità $\Theta(n)$ nel caso peggiore.

MoM_Select(A, l, r, p)

if $l = r$ **return** $A[l]$

$B = \text{array di lunghezza } m = \lceil (r - l + 1)/5 \rceil$, il cui elemento j
è la mediana del j -simo blocco di lunghezza 5 di $A[p:r]$

$_w = MoM_Select(B, 1, m, m/2)$

$\text{swap}(A[r], A[l+5w+2])$

$q = Partition(A, l, r)$

$i = q-l+1$ // p è relativo a $r-l+1$, q a $\text{len}(A)$

if $p=i$ **return** $A[q]$

else if $p < i$ **return** $MoM_Select(A, l, q-1, p)$

else return $MoM_Select(A, q, r, p-i)$

MEDIAN OF MEDIAN

Come costruisco B?

```
// B array di lunghezza  $m = \lceil (r - l + 1)/5 \rceil$ 
for j,i in enumerate(range(l,r,5))
    insertion_sort(A,j,j+5)
    B[i] = A[j+2]
```

B è un array di mediane di lunghezza $n/5$ e cerco ricorsivamente la sua mediana. Poi uso questa mediana di mediane come elemento di pivot nella procedura di **select**.

La correttezza è identica a quella di **quick_select** - posso ignorare il passo di cercare la mediana di mediane.

MEDIAN OF MEDIANS - COMPLESSITÀ

MoM_Select si chiama ricorsivamente due volte, la prima su una istanza di dimensione $n/5$, la seconda sul risultato di partition.

Intuizione: la mediana di mediane x è più grande (piccola) della metà delle mediane dei gruppi di 5 elementi di A , che a loro volta sono più grandi (piccole) di altri due elementi del gruppo.

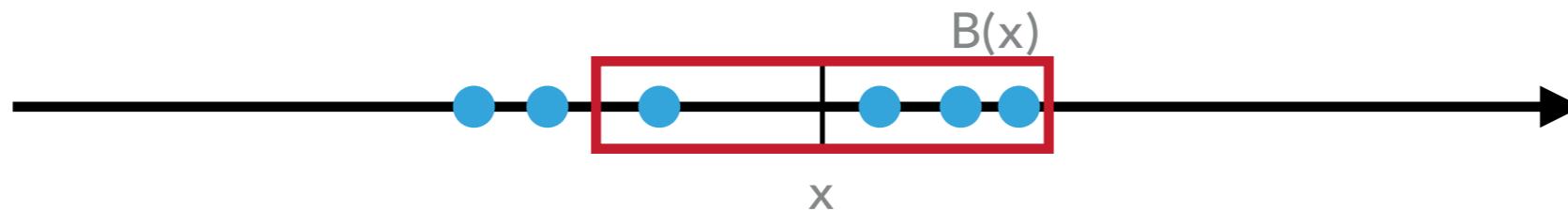
Quindi x è più grande (piccola) di $3 \cdot \frac{1}{2} \cdot \frac{n}{5} = \frac{3}{10}n$ elementi.

Sempre intuitivamente: $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$

Per sostituzione $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7n}{10}\right) + O(n)$ implica $T(n) = O(n)$

K-NN DENSITY ESTIMATION

Dati N osservazioni (in una dimensione) di una variabile continua, possiamo stimare la densità in x come $p(x) = K/NV(x)$, dove $V(x)$ è il volume della palla $B(x)=[x-a,x+a]$ centrata in x che contiene esattamente K punti, ovvero $V(x) = 2a$ in questo caso.



Come possiamo calcolare efficientemente $V(x)$, per ogni x ?

Come possiamo gestire lo scenario online, in cui nuovi punti arrivano di tanto in tanto?

TIPI DI DATO ASTRATTI E CONCRETI

ALGORITMI E STRUTTURE DATI

I Tipi di Dato

I Tipi di Dato

Sono “contenitori” di dati

Fissano le politiche di accesso

Es.

- ▶ First In First Out
- ▶ Last In First Out
- ▶ Ad accesso casuale (Random Access)

Tipi di Dato Astratti e Concreti

Dobbiamo distinguere tra:

- ▶ Tipi di Dato Astratti (ADT): modelli che specificano:
 - ▶ il dominio dei dati da memorizzare
 - ▶ le primitive di accesso
 - ▶ in alcuni casi, definiscono come vengono memorizzati i dati
- ▶ Tipi di Dato Concreti (CDT): implementano gli ADT e:
 - ▶ codificano ogni primitiva di accesso
 - ▶ possono fornire qualche funzionalità “fuori-specifica”

Tipi di Dato Astratti e Concreti

Dobbiamo distinguere tra:

- ▶ Tipi di Dato Astratti (ADT): modelli che specificano:
 - ▶ il dominio dei dati da memorizzare
 - ▶ le primitive di accesso
 - ▶ in alcuni casi, definiscono come vengono memorizzati i dati
- ▶ Tipi di Dato Concreti (CDT): implementano gli ADT e:
 - ▶ codificano ogni primitiva di accesso
 - ▶ possono fornire qualche funzionalità “fuori-specifica”

Due CDT possono implementare la stessa ADT in modi diversi

Array

Array

Consentono l'accesso ai dati memorizzati tramite **indici**

| | | | | | | | | | |
|----|---|---|---|---|---|---|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| -4 | 0 | 1 | 2 | 5 | 6 | 7 | 11 | 12 | 13 |

Array

Consentono l'accesso ai dati memorizzati tramite **indici**

| | | | | | | | | | |
|----|---|---|---|---|---|---|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| -4 | 0 | 1 | 2 | 5 | 6 | 7 | 11 | 12 | 13 |

Le primitive fornite sono:

- ▶ `create(n)` crea un array di dimensione n
- ▶ `get(i)` restituisce il valore in posizione i
- ▶ `set(i, v)` scrive il valore v in posizione i
- ▶ `size()` restituisce la dimensione dell'array

Array

Consentono l'accesso ai dati memorizzati tramite **indici**

| | | | | | | | | | |
|----|---|---|---|---|---|---|----|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| -4 | 0 | 1 | 2 | 5 | 6 | 7 | 11 | 12 | 13 |

Le primitive fornite sono:

- ▶ `create(n)` crea un array di dimensione n
- ▶ `get(i)` restituisce il valore in posizione i
- ▶ `set(i, v)` scrive il valore v in posizione i
- ▶ `size()` restituisce la dimensione dell'array

In Python sono implementati dalla classe `list` (!?!)!

CDT in Python: Array

```
>>> A = [2, -3, 0, 5]      # A is [2,-3,0,5]

>>> A[0]                  # the first index is 0
2

>>> A[-1]                 # the last index is -1
5                         # the 2nd last is -2, etc.

>>> A[1] = 7               # now A is [2,7,0,5]
>>> A
[2, 7, 0, 5]

>>> len(A)                # A stores 4 values
4
```

Liste

Liste

Sono sequenze di valori che forniscono le seguenti funzionalità

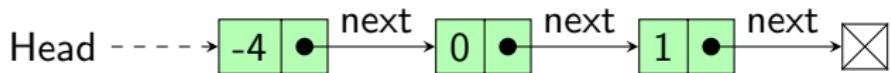
- ▶ `create()` crea una lista vuota
- ▶ `head()` restituisce il primo valore della lista
- ▶ `tail()` restituisce una “vista” alla lista priva del primo valore
- ▶ `prepend(v)` aggiunge v in testa alla lista
- ▶ `is_empty()` verifica se la lista è vuota

Liste Concatenate

ADT derivate dalle liste

Propongono l'organizzazione dei dati:

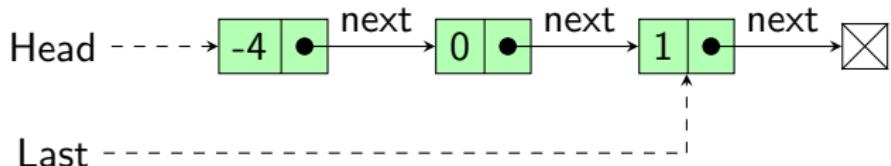
- ▶ ogni valore è contenuto in un **nodo**
- ▶ ogni nodo ha un riferimento al nodo successivo della lista



La lista può essere visitata dal primo elemento all'ultimo.

Liste Concatenate: Una Piccola Aggiunta

Può essere vantaggioso, mantenere un riferimento all'ultimo nodo



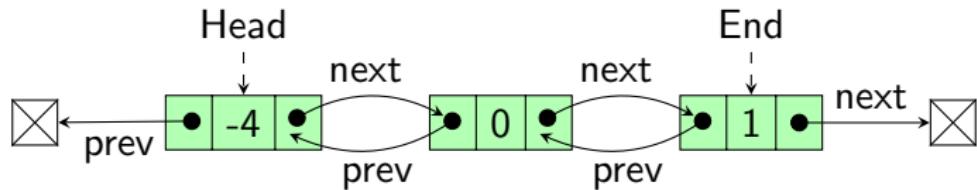
Senza dover scandire la lista, possiamo implementare

- ▶ `last()` restituisce il valore in fondo alla lista
- ▶ `append(v)` aggiunge v in fondo alla lista
- ▶ `extend(L)` accoda la lista L

Liste Doppiamente Concatenate

Sono ADT derivate dalle liste

I nodi hanno anche un riferimento al **predecessore**



Possono essere visitate anche dalla fine alla testa

Senza scandire la lista, implementano anche:

- ▶ `delete_last()` cancella il valore in ultima posizione

Liste vs Array

Le liste sono ADT **dinamiche**:

- ▶ il numero di valori che memorizzano può variare nel tempo
- ▶ non dobbiamo specificare il numero valori contenuti in fase di creazione

Liste vs Array

Le liste sono ADT **dinamiche**:

- ▶ il numero di valori che memorizzano può variare nel tempo
- ▶ non dobbiamo specificare il numero valori contenuti in fase di creazione

Gli array invece sono ADT **statiche**:

- ▶ hanno una dimensione fissata
- ▶ durante la creazione dobbiamo dire quanti oggetti contengono

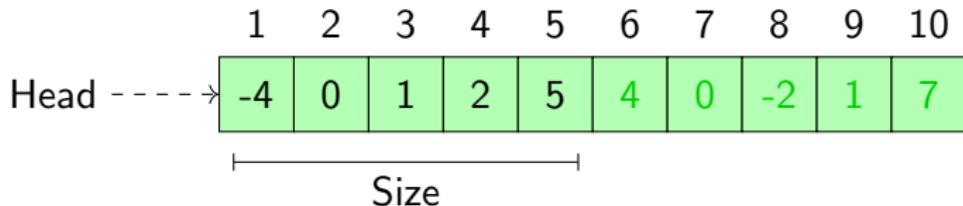
Ma consentono l'indicizzazione senza scansione che è comoda...

Possiamo “Simulare” la Liste con gli Array?

Dobbiamo conoscere la **dimensione massima** della lista

Poi potremmo ...

- ▶ memorizzare i valori in ordine dal primo indice
- ▶ tenere traccia della dimensione

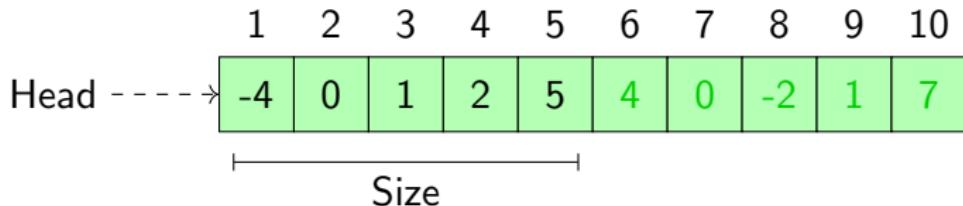


Possiamo “Simulare” la Liste con gli Array?

Dobbiamo conoscere la **dimensione massima** della lista

Poi potremmo ...

- ▶ memorizzare i valori in ordine dal primo indice
- ▶ tenere traccia della dimensione



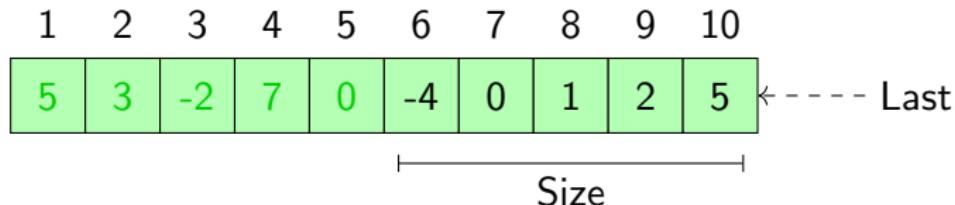
Ottimo per `append(v)` e `delete_last()`, ma... `tail()` e `prepend(v)`?

Possiamo “Simulare” la Liste con gli Array?

Dobbiamo conoscere la **dimensione massima** della lista

Poi potremmo ...

- ▶ memorizzare i valori in ordine dall'ultimo indice
- ▶ tenere traccia della dimensione

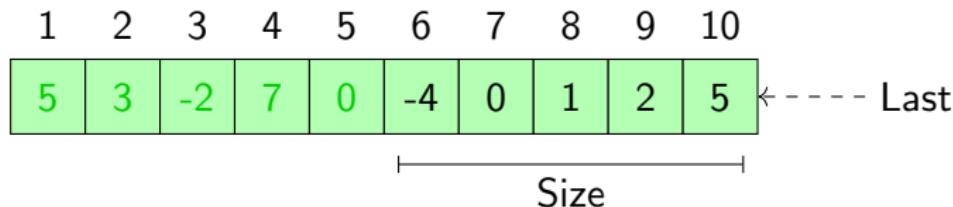


Possiamo “Simulare” la Liste con gli Array?

Dobbiamo conoscere la **dimensione massima** della lista

Poi potremmo ...

- ▶ memorizzare i valori in ordine dall'ultimo indice
- ▶ tenere traccia della dimensione



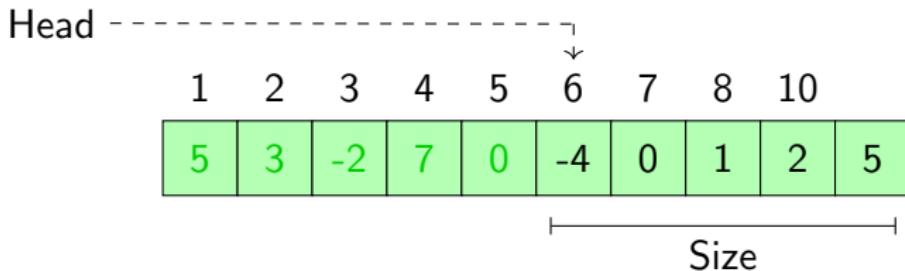
Ok per `tail()` e `prepend(v)`, ma ... `append(v)` e `delete_last()`?

Array Circolari

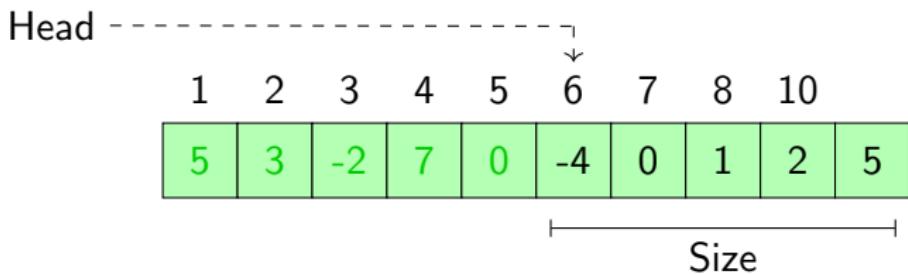
Gli Array Circolari

Rappresentano liste usando array memorizzando:

- ▶ la dimensione della lista
- ▶ l'indice del primo elemento della lista



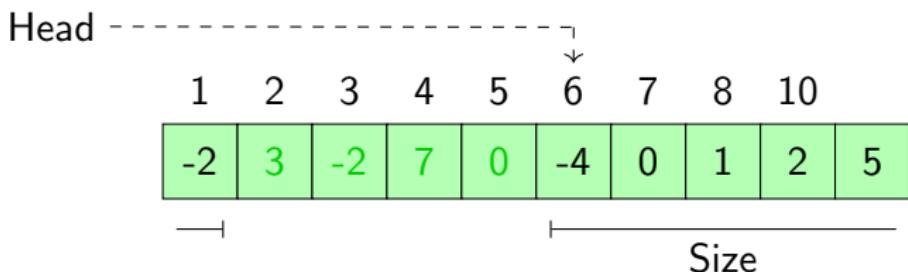
Gli Array Circolari



Per aggiungere -2 in fondo alla lista:

- ▶ inseriamo il valore in posizione $(\text{Head} + \text{Size}) \% \text{max_size} = 0$
- ▶ incrementiamo di 1 la dimensione della lista

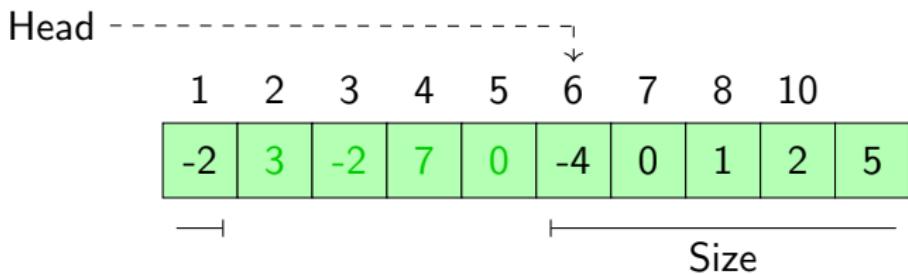
Gli Array Circolari



Per aggiungere -2 in fondo alla lista:

- ▶ inseriamo il valore in posizione $(\text{Head} + \text{Size}) \% \text{max_size} = 0$
- ▶ incrementiamo di 1 la dimensione della lista

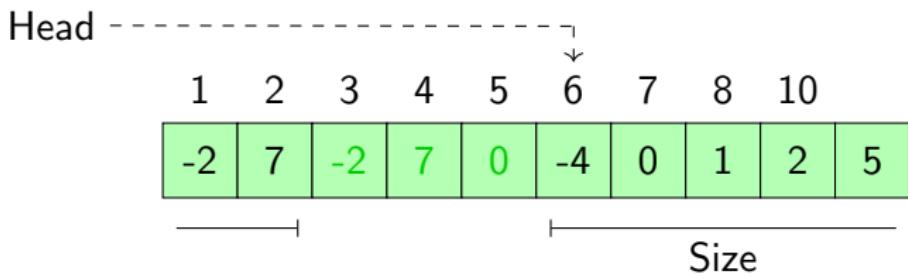
Gli Array Circolari



Per aggiungere 7 in fondo alla lista:

- ▶ inseriamo il valore in posizione $(\text{Head} + \text{Size}) \% \text{max_size} = 1$
- ▶ incrementiamo di 1 la dimensione della lista

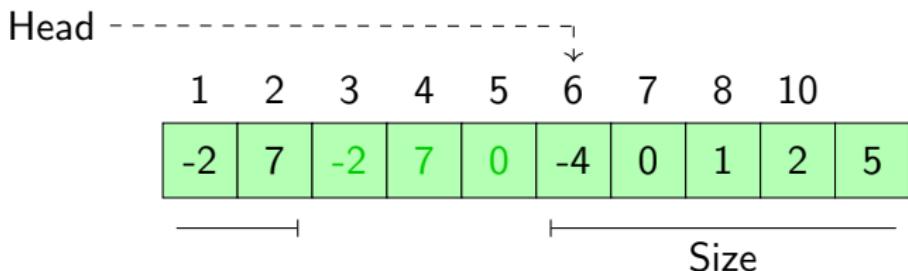
Gli Array Circolari



Per aggiungere 7 in fondo alla lista:

- ▶ inseriamo il valore in posizione $(\text{Head} + \text{Size}) \% \text{max_size} = 1$
- ▶ incrementiamo di 1 la dimensione della lista

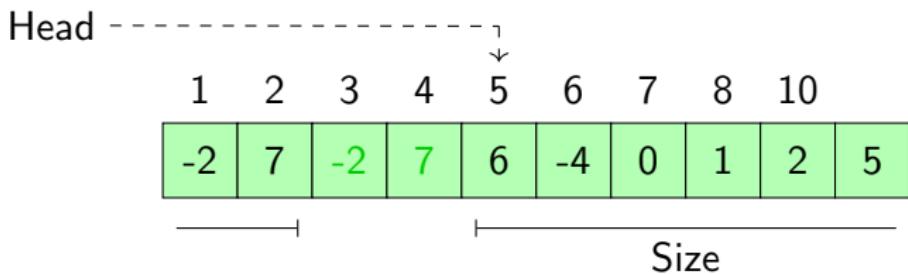
Gli Array Circolari



Per aggiungere 6 in testa alla lista:

- ▶ inseriamo il valore in posizione $(\text{Head}-1) \% \text{max_size} = 4$
- ▶ incrementiamo di 1 la dimensione della lista
- ▶ riassegnamo l'indice della testa

Gli Array Circolari



Per aggiungere 6 in testa alla lista:

- ▶ inseriamo il valore in posizione $(\text{Head}-1)\% \text{max_size} = 4$
- ▶ incrementiamo di 1 la dimensione della lista
- ▶ riassegnamo l'indice della testa

Pile

Pile (Stack)

ADT che usano la politica First In Last Out:

- ▶ `push(v)` inserisce il valore `v` in cima alla pila
- ▶ `pop()` rimuove il valore in cima alla pila e lo restituisce
- ▶ `top()` restituisce il valore in cima alla pila senza rimuoverlo dalla stessa
- ▶ `is_empty()` restituisce `true` se e solo se la pila è vuota

Pile (Stack)

Come implementare le code?

Pile (Stack)

Come implementare le pile? Con le liste !?!?!?

- ▶ Liste Concatenate
- ▶ Array (non servono nemmeno quelli circolari!?!?)

`push(v) = append(v)` e `pop() = last() + delete_last()`

Code

Code (Queue)

ADT che usano la politica First In First Out:

- ▶ `enqueue(v)` inserisce il valore v nella coda
- ▶ `dequeue()` rimuove dalla coda il valore che ci sta da più tempo
- ▶ `head()` restituisce il valore in testa alla coda senza rimuoverlo dalla stessa
- ▶ `is_empty()` restituisce true se e solo se la coda è vuota

Code (Queue)

Come implementare le code?

Code (Queue)

Come implementare le code? Con le liste !?!?!?

- ▶ Liste Doppiaamente Concatenate
- ▶ Array Circolari

`enqueue(v) = append(v)` e `dequeue() ≈ head() + tail()`

DIZIONARI
ALBERI BINARI

ALGORITMI E STRUTTURE DATI

DYNAMIC SET / DIZIONARI

- ▶ A volte le operazioni che vogliamo fare sono più complesse di quelle consentite da stack e code
- ▶ Supponiamo di avere un insieme di valori, o di coppie chiave e valore e di volerli organizzare in un insieme che però possiamo modificare dinamicamente.
- ▶ Esempi di coppie chiave valore (k, v):
studenti (chiave: matricola),
automobili (chiave: targa),
persone (chiave: codice fiscale)

DYNAMIC SET / DIZIONARI

- ▶ Le operazioni di base dei dizionari / insiemi dinamici sono:
 - ▶ **Inserimento** (*insert*). Inserisce un elemento nel dizionario
 - ▶ **Rimozione** (*delete*). Rimuove un elemento dal dizionario
 - ▶ **Ricerca** (*search*). Ritorna l'elemento nel dizionario con la chiave fornita come argomento (se esiste)

DYNAMIC SET / DIZIONARI

- ▶ Se l'insieme delle chiavi è totalmente ordinato possiamo definire le seguenti operazioni:
 - ▶ **Minimo.** Ritorna l'oggetto con la chiave di valore minimo
 - ▶ **Massimo.** Ritorna l'oggetto con la chiave di valore massimo
 - ▶ **Successore.** Dato un oggetto ritorna il successivo (rispetto al valore della chiave)
 - ▶ **Predecessore.** Dato un oggetto ritorna il successivo (rispetto al valore della chiave)

DYNAMIC SET / DIZIONARI

- ▶ Vedremo diversi modi di implementare in modo efficiente un dizionario che sono parte di due famiglie:
 - ▶ Alberi binari di ricerca
 - ▶ Tabelle hash
- ▶ Prima però vediamo perché le liste concatenante non sono la scelta migliore

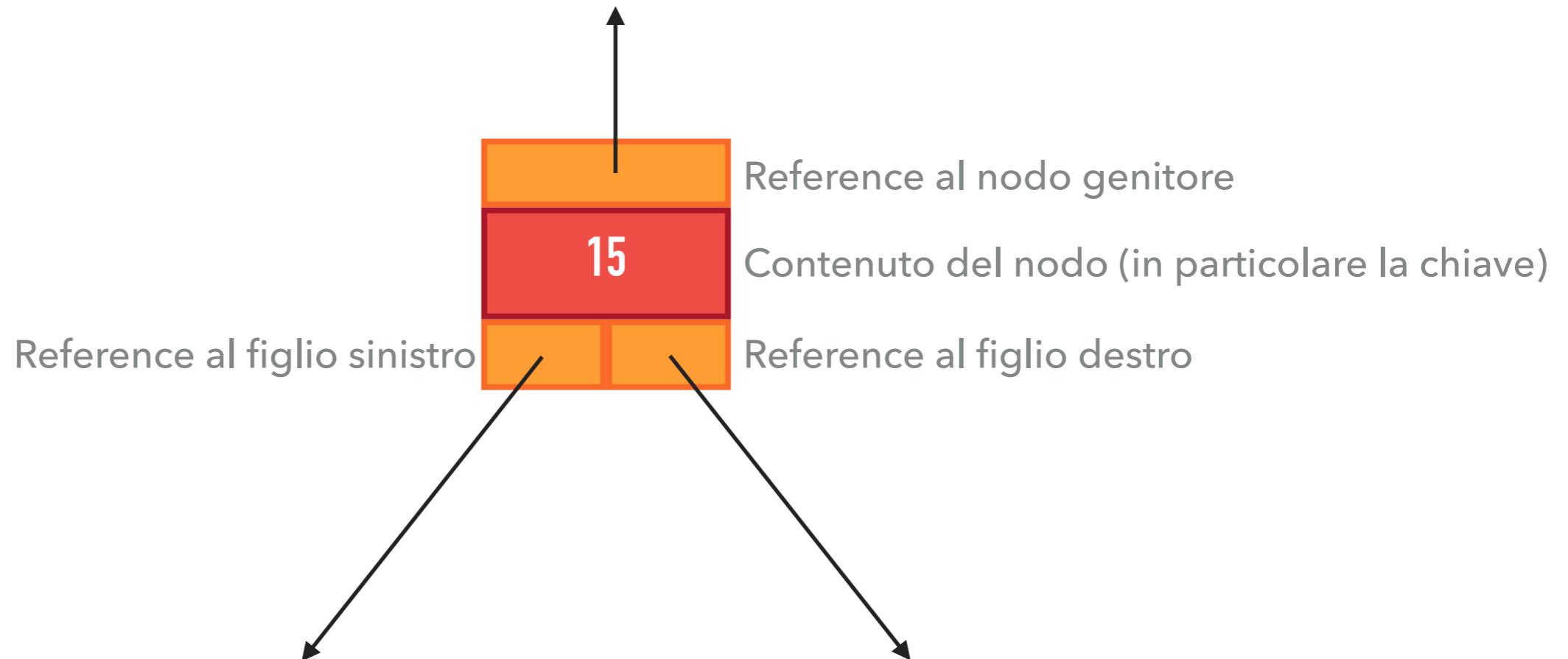
DYNAMIC SET / DIZIONARI CON LISTE CONCATENATE

- ▶ L'inserimento può essere effettuato rapidamente in testa o in coda (tempo costante)...
- ▶ ...ma in quel caso la ricerca richiede di scorrere tutta la lista (tempo lineare)
- ▶ Non possiamo migliorare la situazione tenendo la lista ordinata, perché non possiamo effettuare efficientemente la ricerca binaria (l'accesso a posizioni arbitrarie richiede tempo lineare)

ALBERI BINARI: RAPPRESENTAZIONE

- ▶ Abbiamo già visto gli alberi binari (heapsort)
- ▶ Per utilizzi “generici” potrebbe essere utile avere una rappresentazione che non si basa su tenere tutto all’interno di un array
- ▶ Rappresentiamo quindi ogni nodo come l’oggetto che deve essere contenuto (dato che noi interessa solo la chiave rappresenteremo solo quella) e un insieme di reference ai nodi figli e al nodo genitore

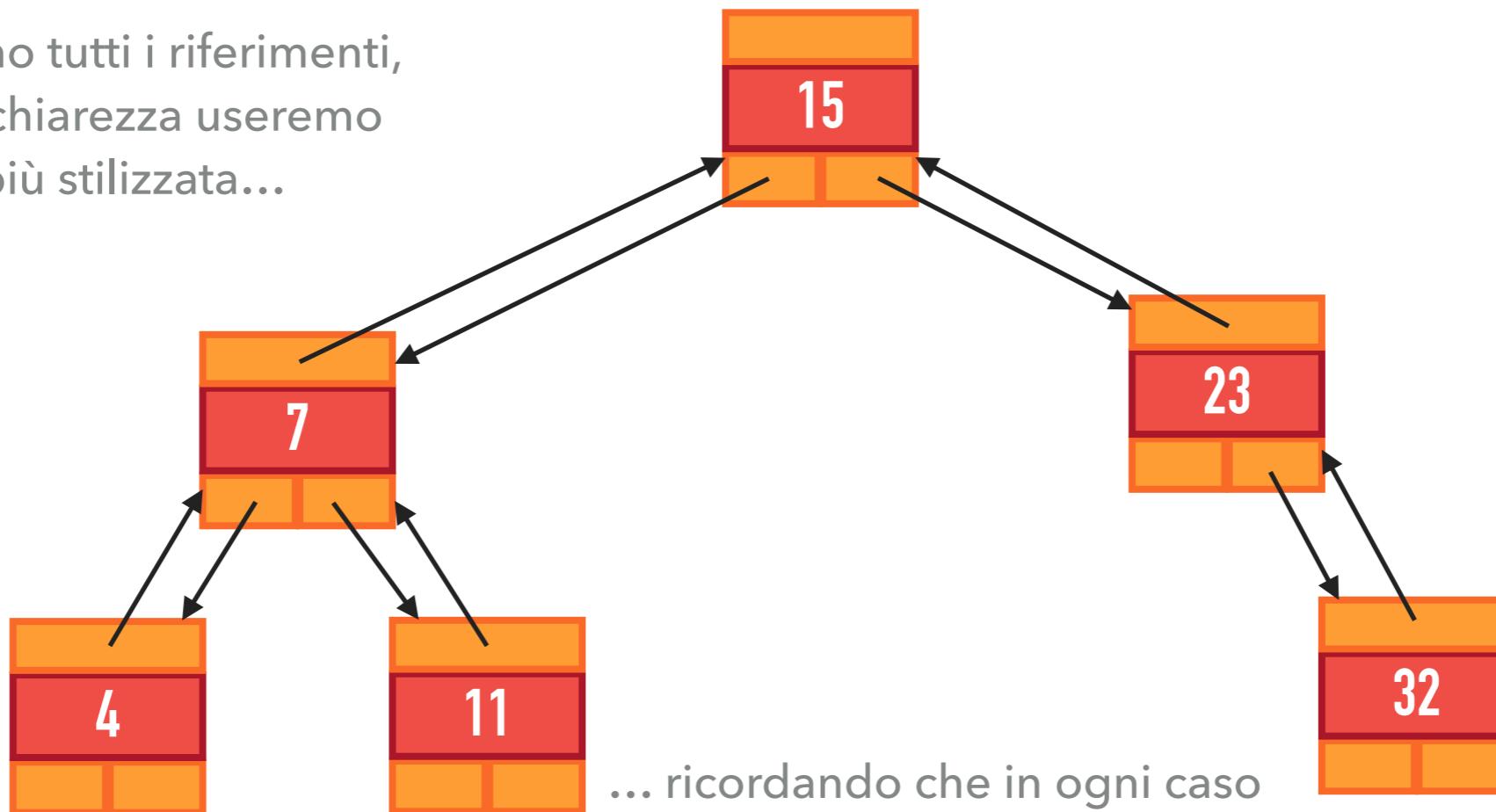
NODO DI UN ALBERO BINARIO



A seconda dell'implementazione e delle operazioni da svolgere potremmo non avere il riferimento al nodo genitore, avere un riferimento al nodo "fratello", etc.

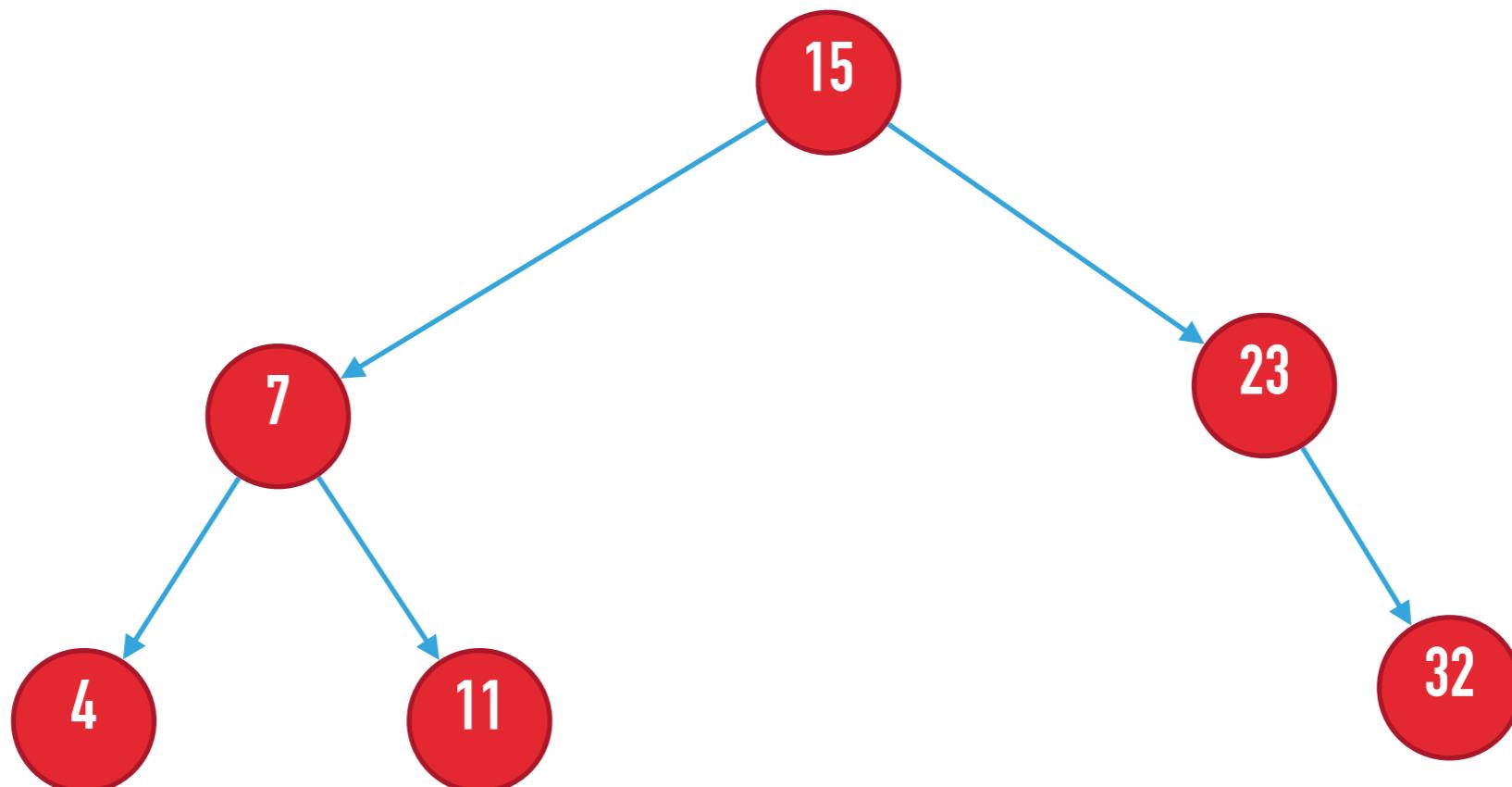
ALBERO BINARIO RAPPRESENTAZIONE GRAFICA

Qui visualizziamo tutti i riferimenti,
ma spesso per chiarezza useremo
una notazione più stilizzata...



... ricordando che in ogni caso
tutti questi reference continuano
ad esistere

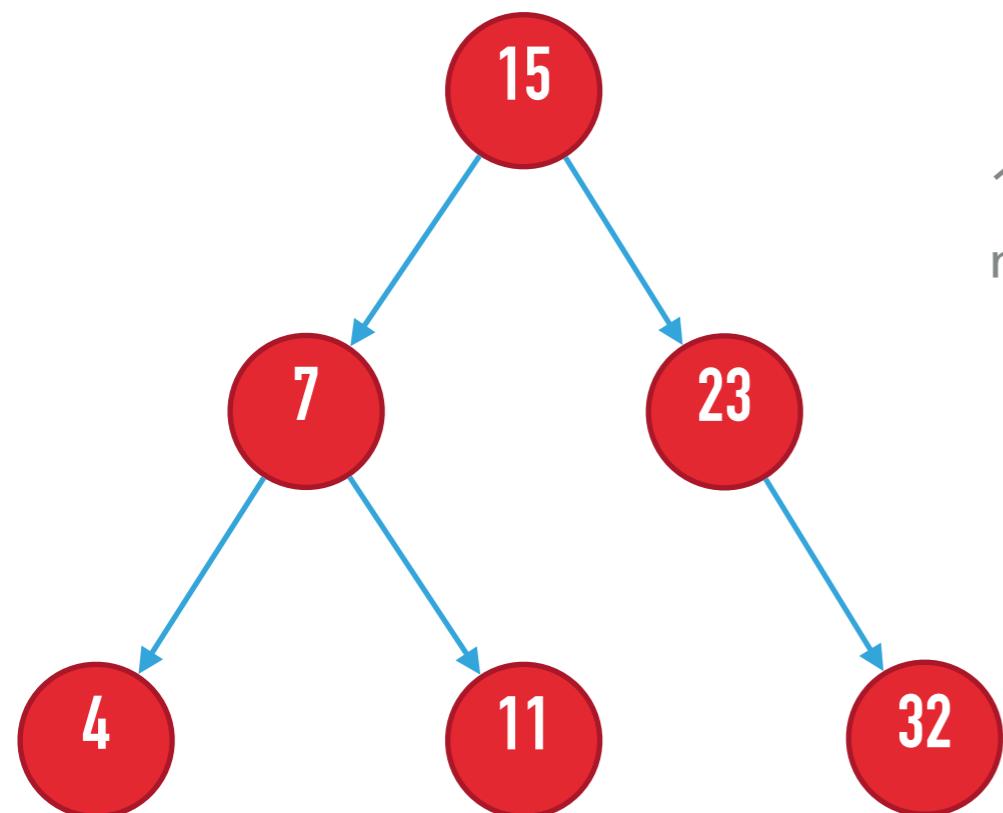
ALBERO BINARIO RAPPRESENTAZIONE GRAFICA



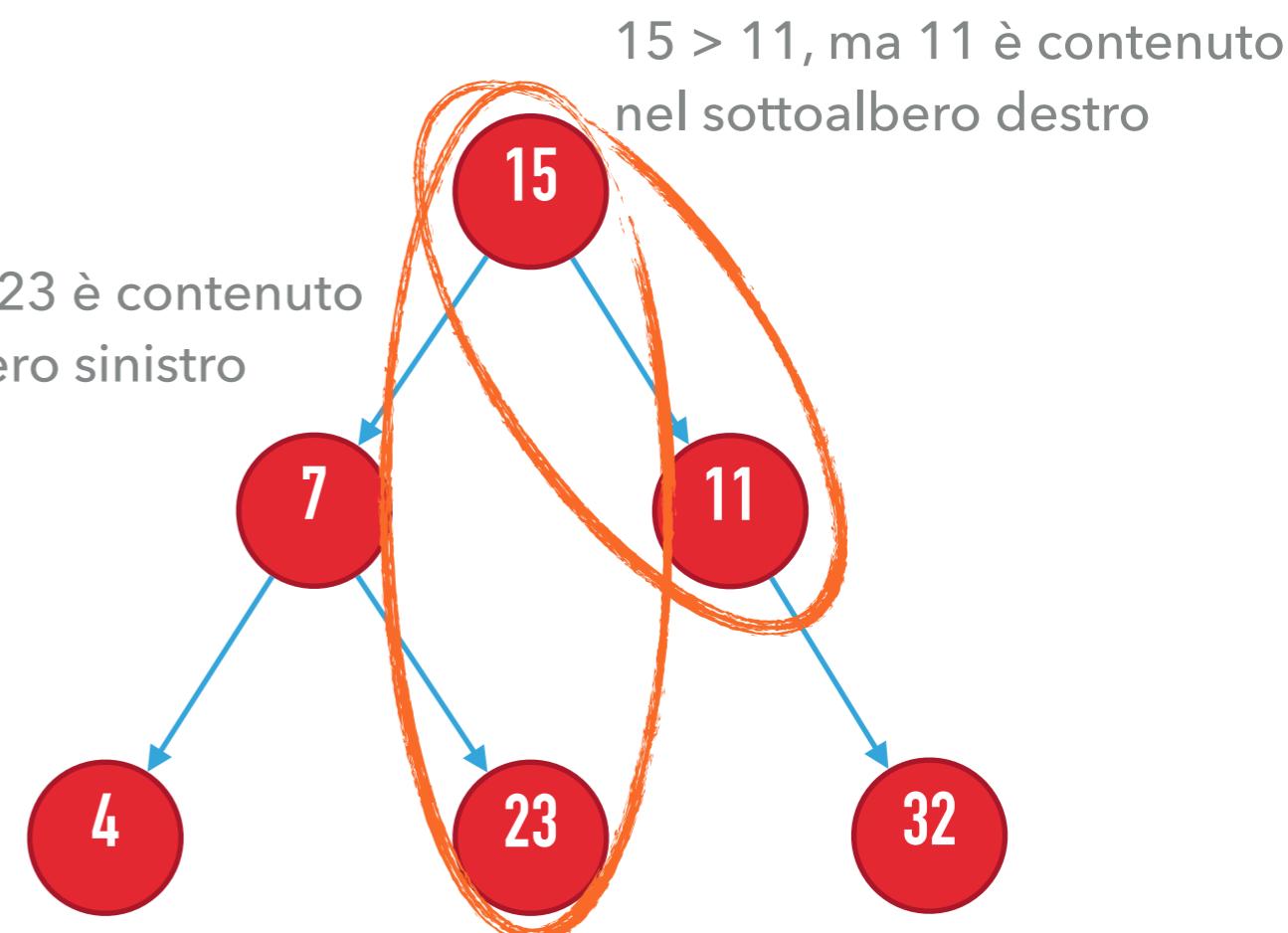
ALBERI BINARI DI RICERCA

- ▶ Un albero binario di ricerca (o binary search tree – BST) richiede chiavi che siano totalmente ordinate (e.g., interi)
- ▶ Ogni nodo dell'albero contiene una chiave
- ▶ Ogni nodo dell'albero ha la seguente proprietà:
 - ▶ Tutti i nodi nel sottoalbero sinistro hanno chiave *minore* della chiave nel nodo
 - ▶ Tutti i nodi del sottoalbero destro hanno chiave *maggiori* della chiave nel nodo

ALBERO BINARIO DI RICERCA



É un albero binario di ricerca



NON è un albero binario di ricerca

ALBERI BINARI: RICERCA

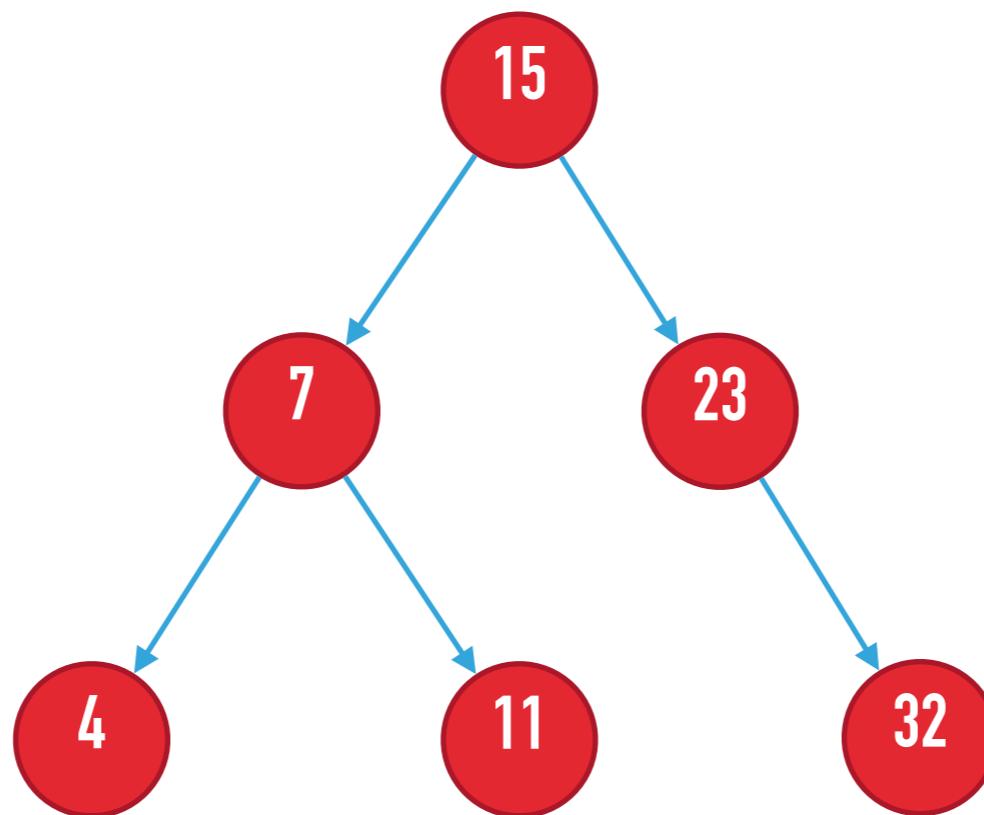
- ▶ Dato un albero binario la ricerca può essere suddivisa in quattro casi:
- ▶ Albero vuoto: l'elemento non è contenuto
- ▶ La chiave della radice è quella che stiamo cercando: abbiamo trovato l'elemento
- ▶ La chiave che stiamo cercando è **minore** della chiave nella radice: cerchiamo ricorsivamente nel sottoalbero di **sinistra**
- ▶ La chiave che stiamo cercando è **maggior**e della chiave nella radice: cerchiamo ricorsivamente nel sottoalbero di **destra**

ALBERI BINARI: PSEUDOCODICE DELLA RICERCA

```
▶ Parametri: Nodo radice, key
▶ if radice is None
    ▶ return None # l'albero è vuoto, quindi sicuramente la chiave non esiste
▶ if radice.key == key
    ▶ return radice # abbiamo trovato un nodo con la chiave che cercavamo
▶ if key < radice.key
    ▶ return ricerca(radice.left, key) # ricerca nel sottoalbero sinistro
▶ else # ovvero key > radice.key
    ▶ return ricerca(radice.right, key) # ricerca nel sottoalbero destro
```

ALBERO BINARIO: RICERCA

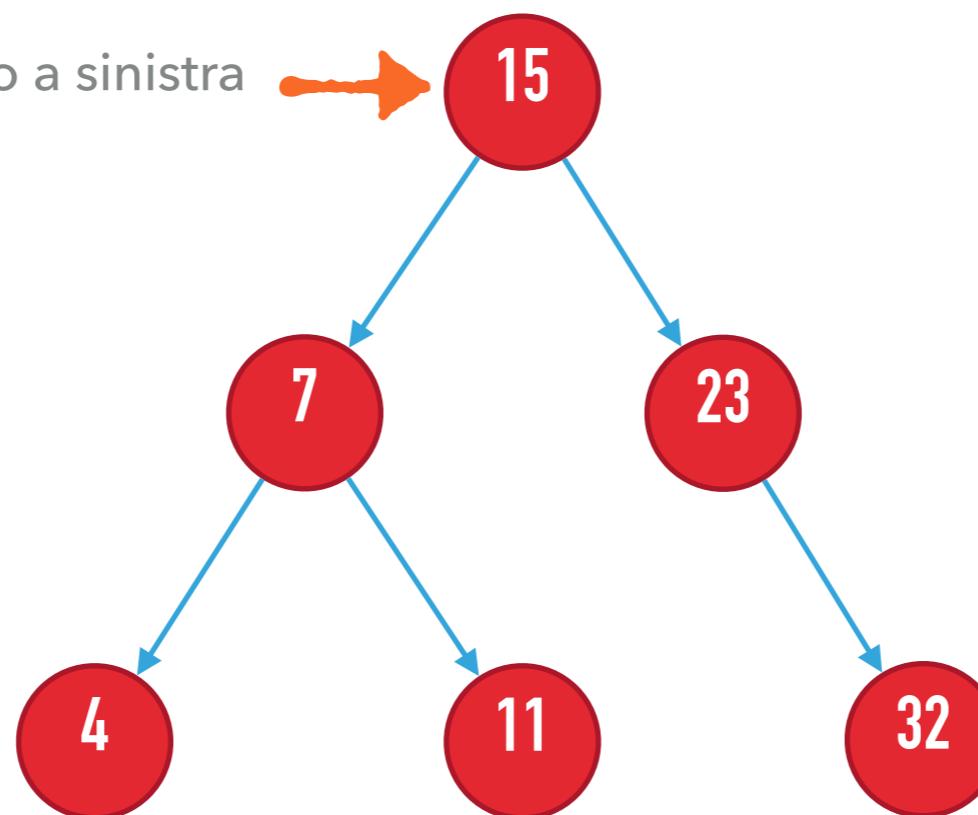
Supponiamo di voler trovare l'elemento di chiave 11



ALBERO BINARIO: RICERCA

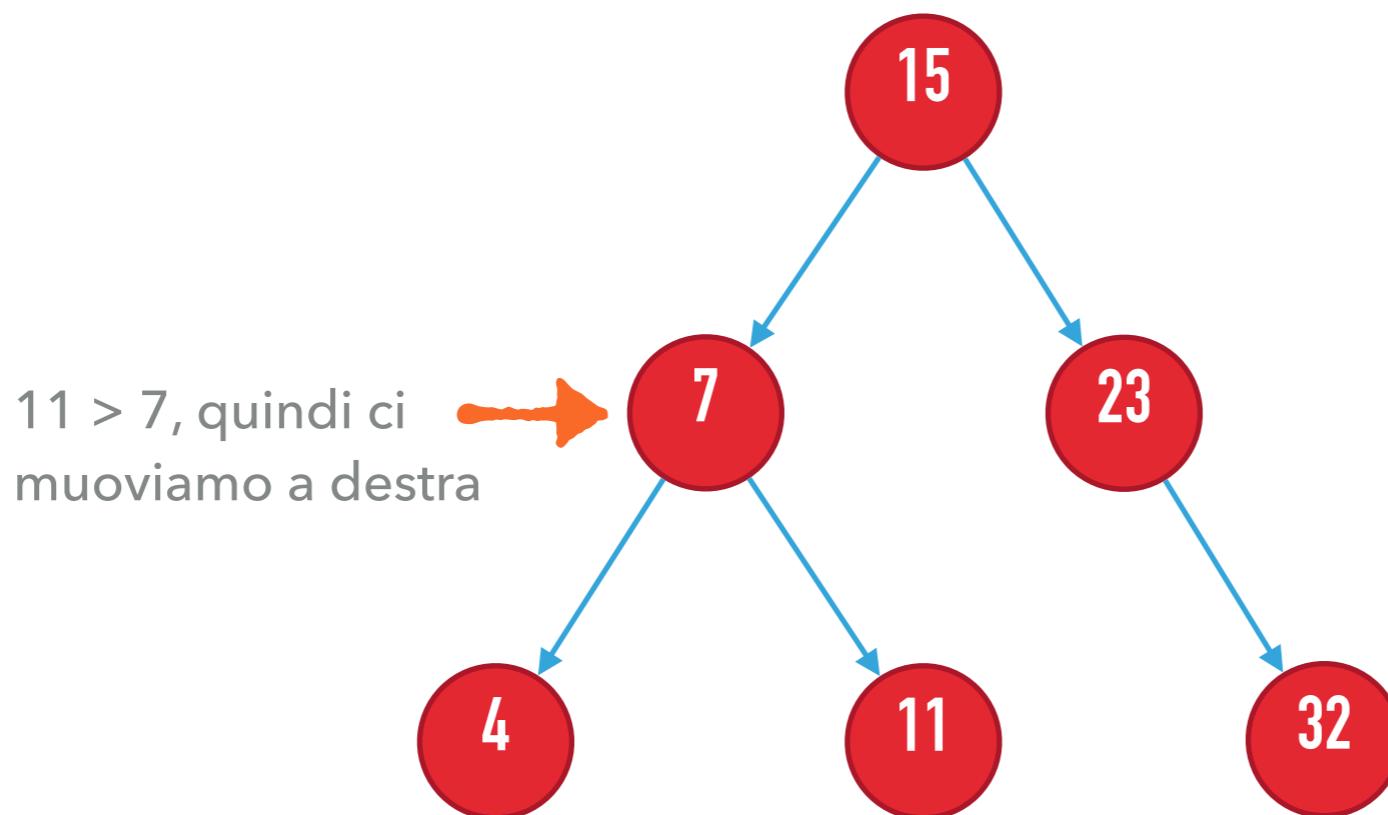
Supponiamo di voler trovare l'elemento di chiave 11

11 < 15, quindi ci muoviamo a sinistra



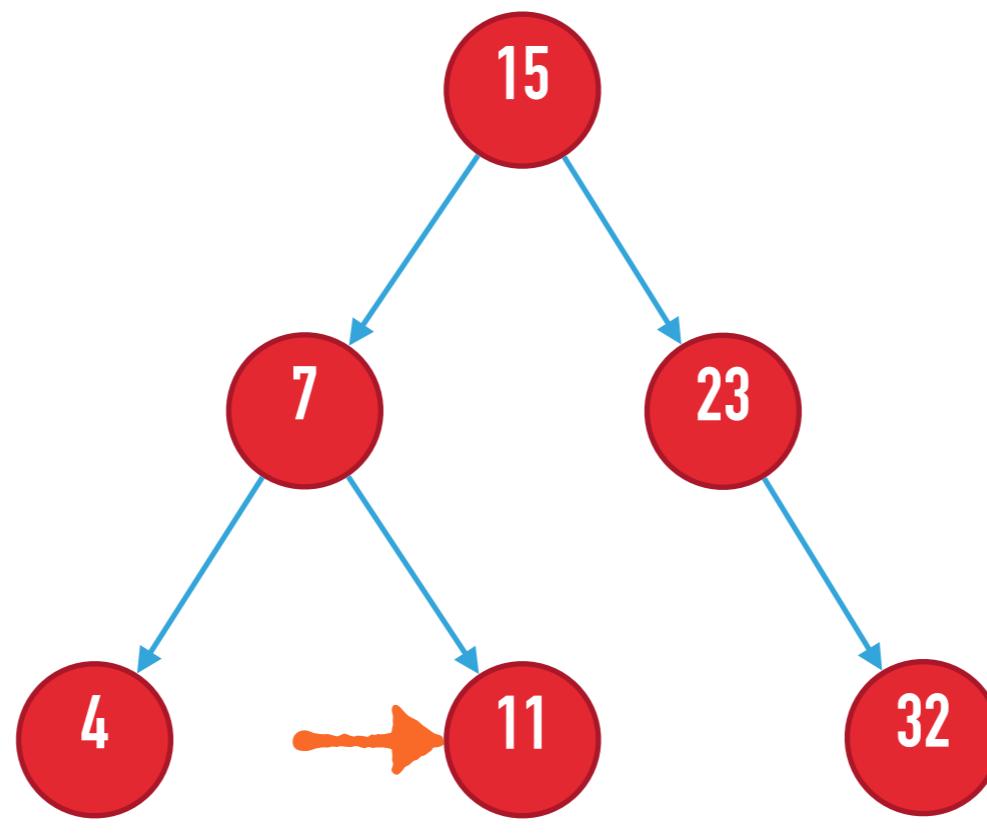
ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l'elemento di chiave 11



ALBERO BINARIO: RICERCA

Supponiamo di voler trovare l'elemento di chiave 11

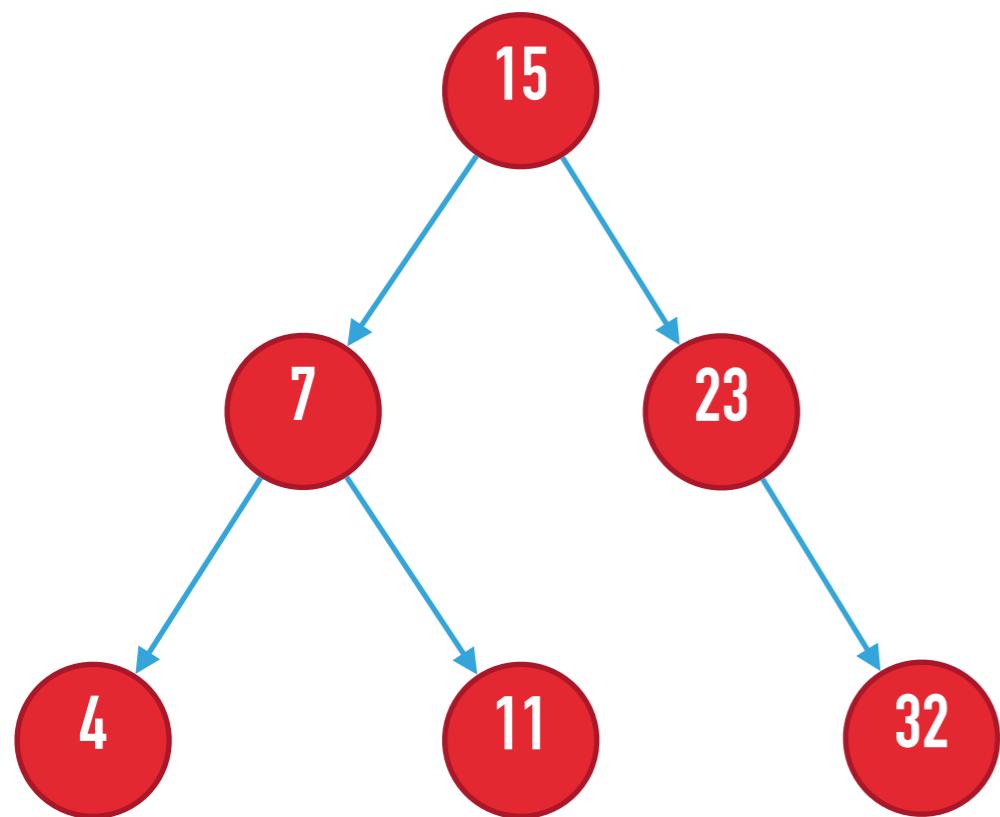


Trovato!

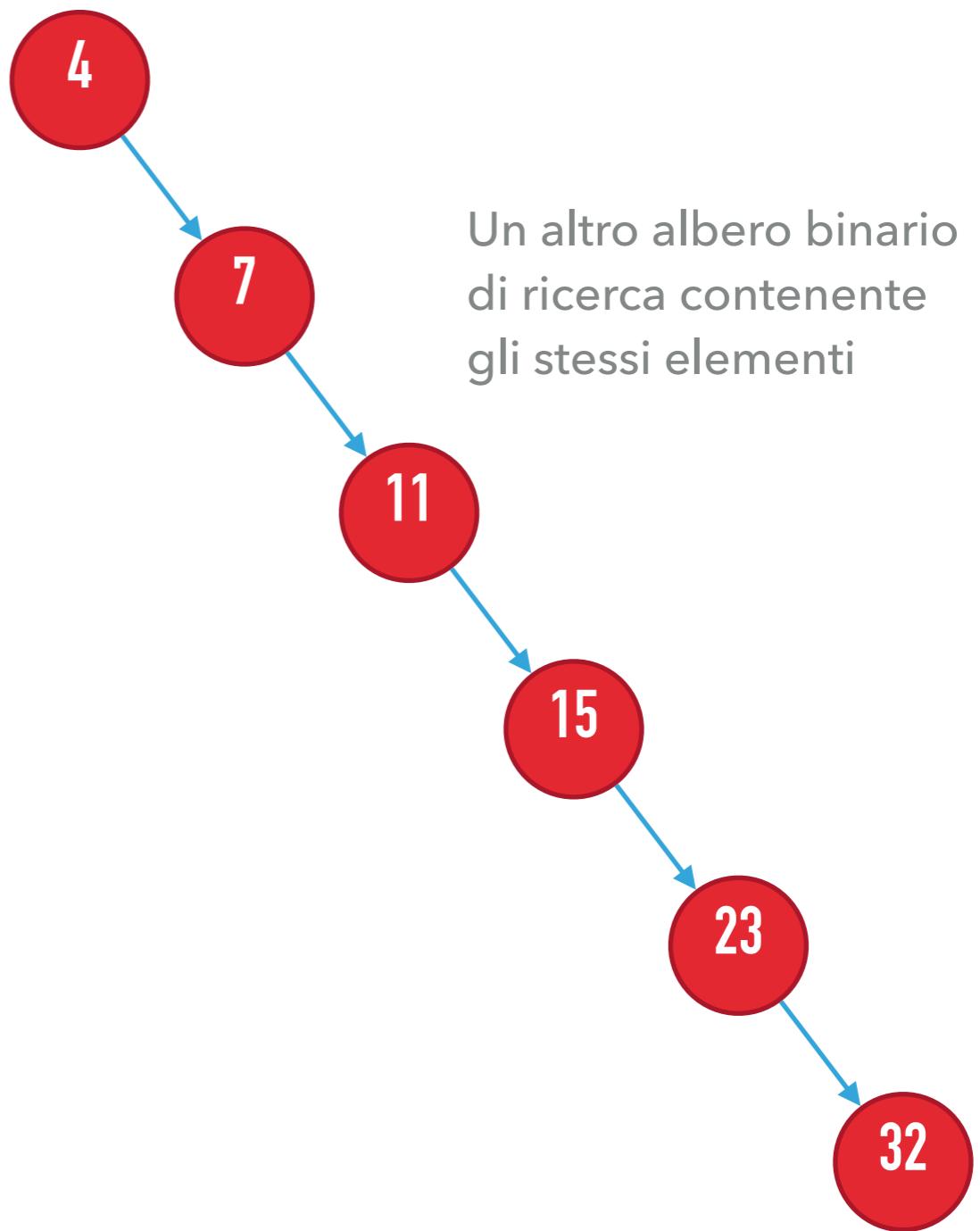
ALBERI BINARI: RICERCA

- ▶ Quale è la complessità della ricerca?
- ▶ Facciamo un numero costante di passi prima di ogni chiamata ricorsiva...
- ▶ ... E facciamo un numero di chiamate ricorsive che è limitato dall'altezza dell'albero (ad ogni chiamata ricorsiva scendiamo di un livello)
- ▶ Quindi $O(h)$ dove h è l'altezza dell'albero.
- ▶ Quindi $O(\log n)$?

ALBERO BINARIO DI RICERCA



Albero binario di ricerca



ALBERI BILANCIATI E SBILANCIATI

- ▶ Un albero binario può essere più o meno sbilanciato
- ▶ Nel caso migliore abbiamo un albero con la profondità minima necessaria a contenere tutti gli elementi, quindi la ricerca avviene in tempo $O(\log n)$
- ▶ Nel caso peggiore abbiamo qualcosa di simile ad una lista concatenata, la profondità dell'albero è lineare rispetto al numero di elementi e la ricerca richiede tempo $O(n)$
- ▶ Vedremo più avanti metodi per mantenere gli alberi bilanciati

VISITA IN PRE-ORDINE, IN-ORDINE E POST-ORDINE

- ▶ Gli alberi binari generalmente hanno diversi modi in cui possono enumerare gli elementi (visitandoli tutti)
- ▶ I tre modi principali differiscono solo del momento in cui viene visitato il nodo corrente:
- ▶ **Pre-ordine:** nodo corrente, sottoalbero sinistro, sottoalbero destro
- ▶ **In-ordine:** sottoalbero sinistro, nodo corrente, sottoalbero destro
- ▶ **Post-ordine:** sottoalbero sinistro, sottoalbero destro, nodo corrente

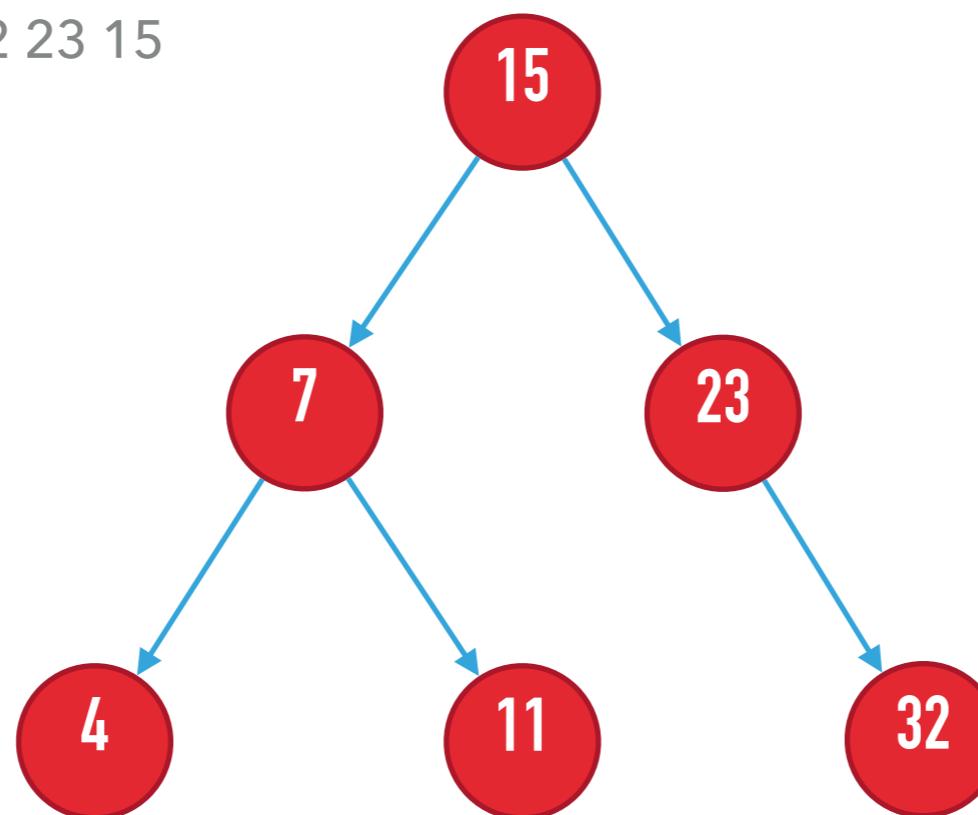
ALBERO BINARIO: VISITE

Visita in pre-ordine: 15 7 4 11 23 32

Visita in ordine: 4 7 11 15 23 32

Visita in post-ordine: 4 11 7 32 23 15

La visita in ordine visita sempre gli elementi in
ordine del valore della chiave



ALBERI BINARI: INSERIMENTO

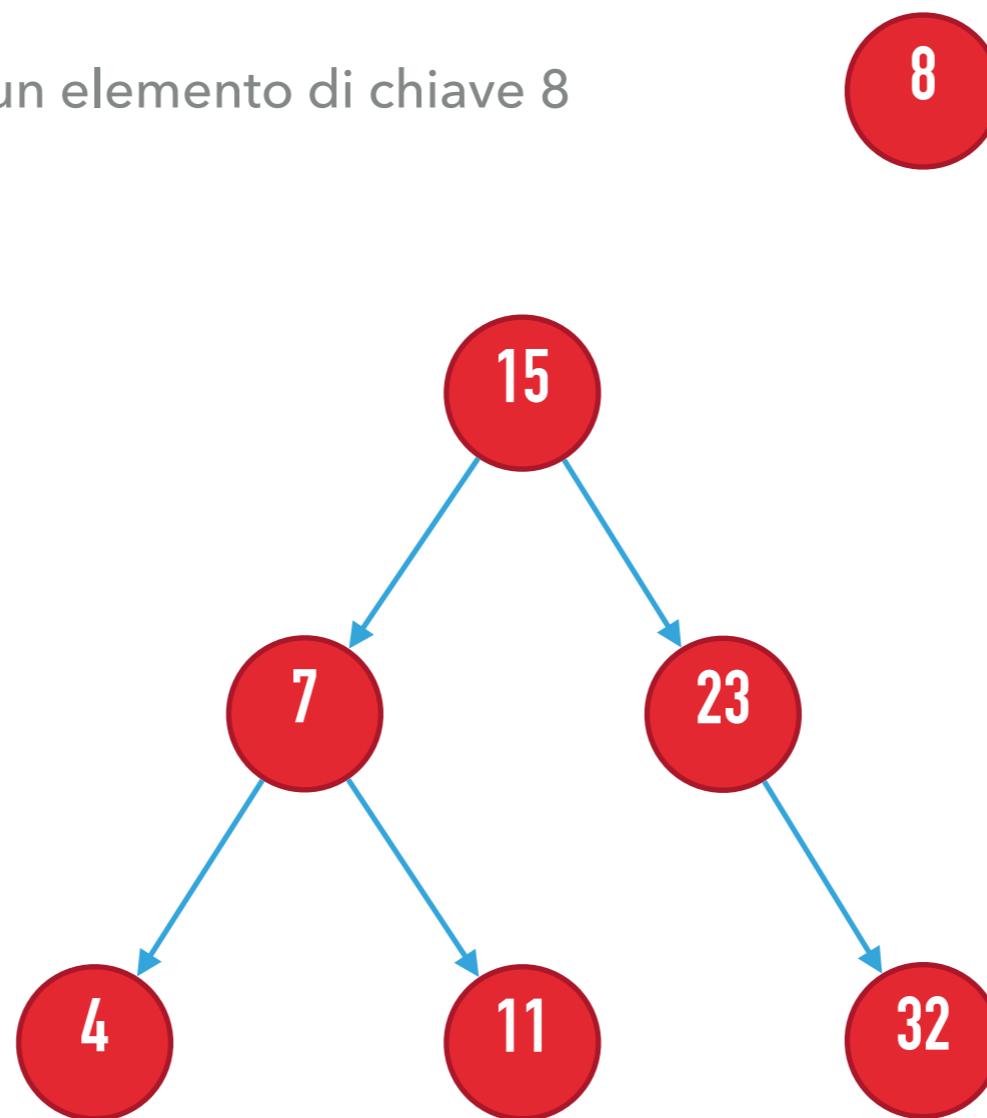
- ▶ L'inserimento avviene in modo simile alla ricerca
- ▶ Data una chiave k dobbiamo trovare un posto libero per inserire quella chiave **rispettando la proprietà dell'albero binario di ricerca**
- ▶ Confrontiamo k con la chiave nella radice:
 - ▶ Se k è maggiore, andrà inserita a destra della radice
 - ▶ Se k è minore, andrà inserita a sinistra della radice

ALBERI BINARI: INSERIMENTO

- ▶ Supponiamo di dover proseguire a sinistra (i.e., k minore del valore nella radice). Abbiamo due casi:
 - ▶ La radice non ha un figlio sinistro: possiamo direttamente inserire k come figlio sinistro della radice
 - ▶ La radice ha un figlio sinistro: chiamiamo ricorsivamente l'inserimento nell'albero di radice il figlio sinistro
- ▶ Simmetricamente se si prosegue a destra

ALBERO BINARIO: INSERIMENTO

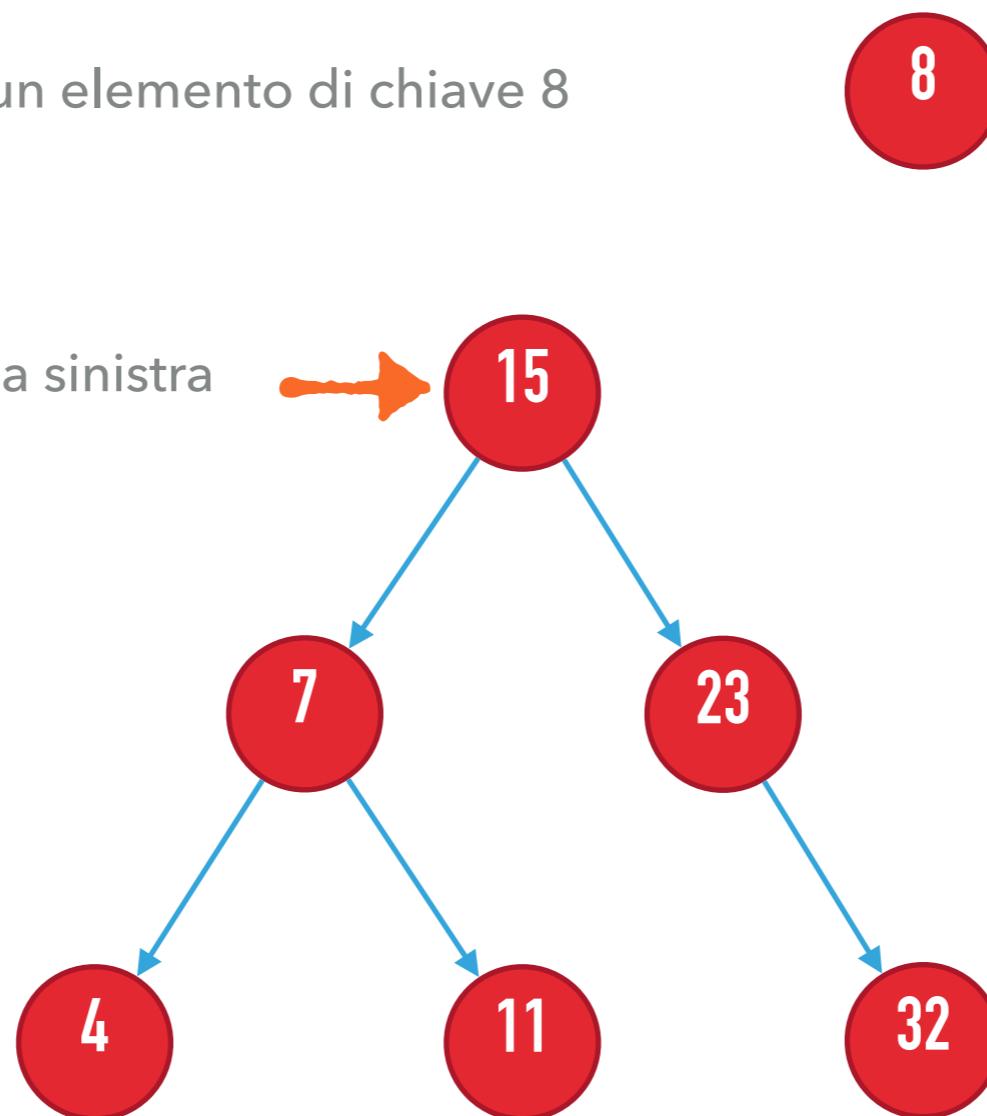
Supponiamo di voler inserire un elemento di chiave 8



ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8

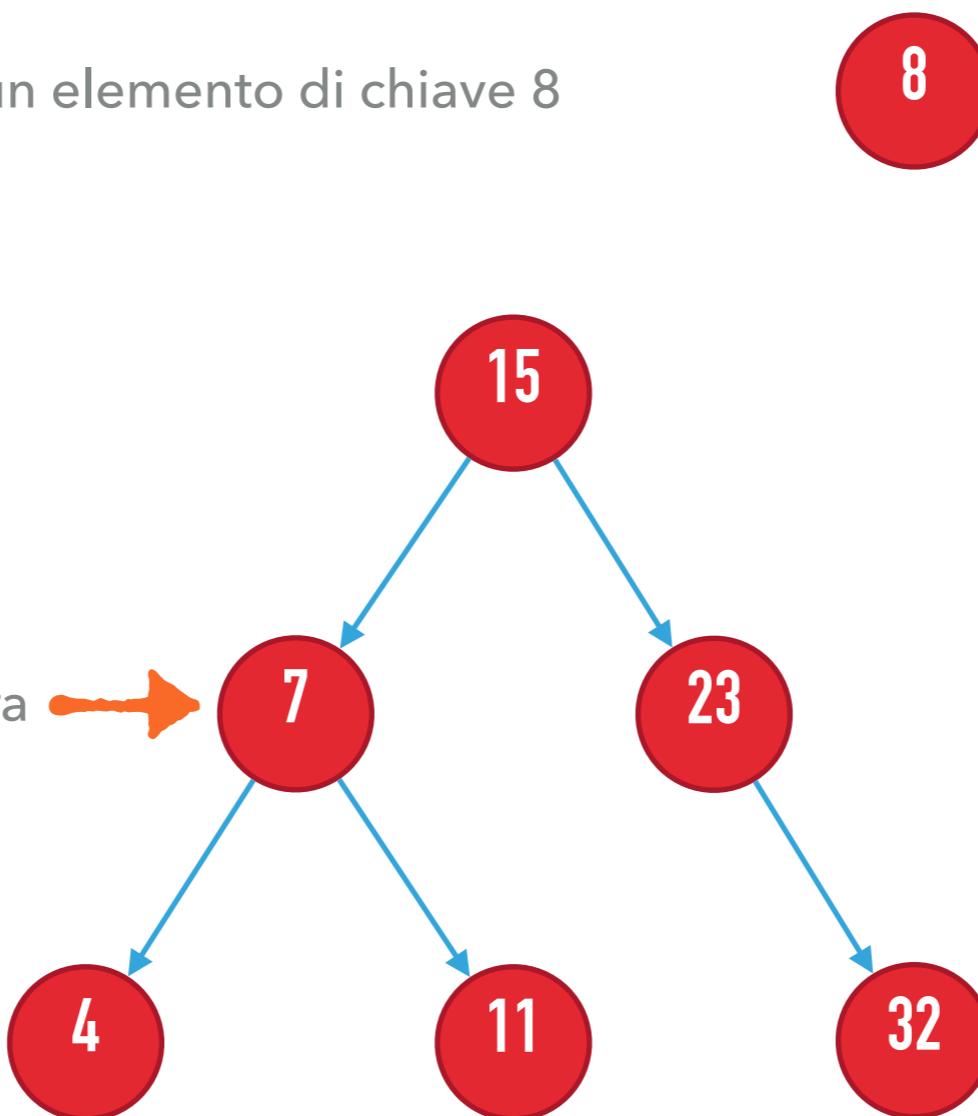
$8 < 15$, quindi ci muoviamo a sinistra



ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8

8 > 7, quindi ci muoviamo a destra



ALBERO BINARIO: INSERIMENTO

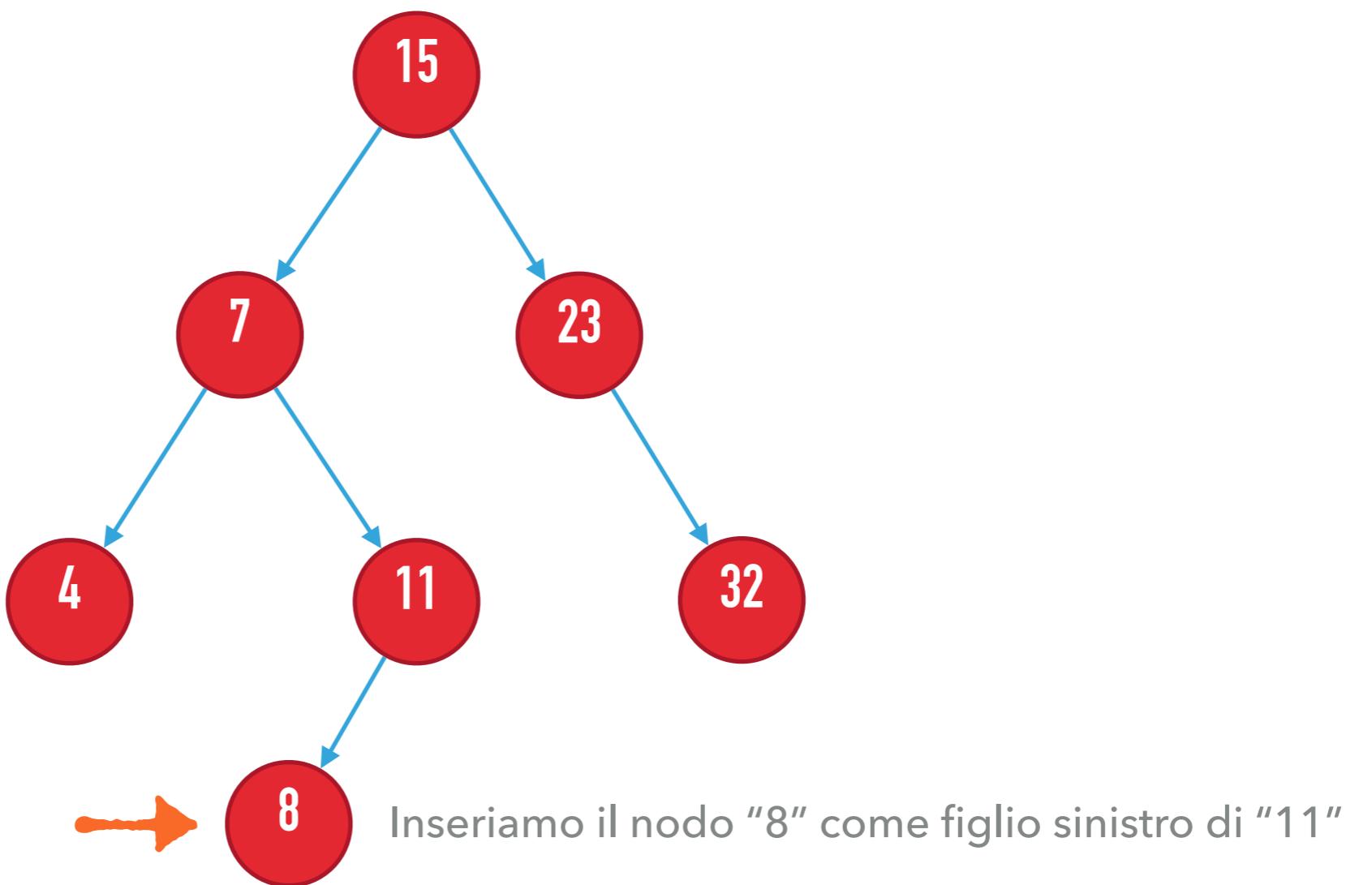
Supponiamo di voler inserire un elemento di chiave 8



$8 < 11$, ci dovremmo muovere a sinistra,
ma il nodo non ha un figlio sinistro

ALBERO BINARIO: INSERIMENTO

Supponiamo di voler inserire un elemento di chiave 8



ALBERI BINARI: PSEUDOCODICE DELL'INSERIMENTO

```
▶ Parametri: Nodo radice, key
▶ if radice is None
    ▶ return Nodo(key) # l'albero è vuoto ritorniamo un nuovo albero
▶ if key < radice.key
    ▶ if radice.left is None # figlio sinistro libero per l'inserimento
        ▶ radice.left = nuovo_nodo(key)
    ▶ else # chiamata ricorsiva sul sottoalbero sinistro
        ▶ inserisci(radice.left, key)
▶ else # ovvero key ≥ radice.key
    ▶ if radice.right is None # figlio destro libero per l'inserimento
        ▶ radice.right = nuovo_nodo(key)
    ▶ else # chiamata ricorsiva sul sottoalbero sinistro
        ▶ inserisci(radice.right, key)
```

ALBERI BINARI: INSERIMENTO

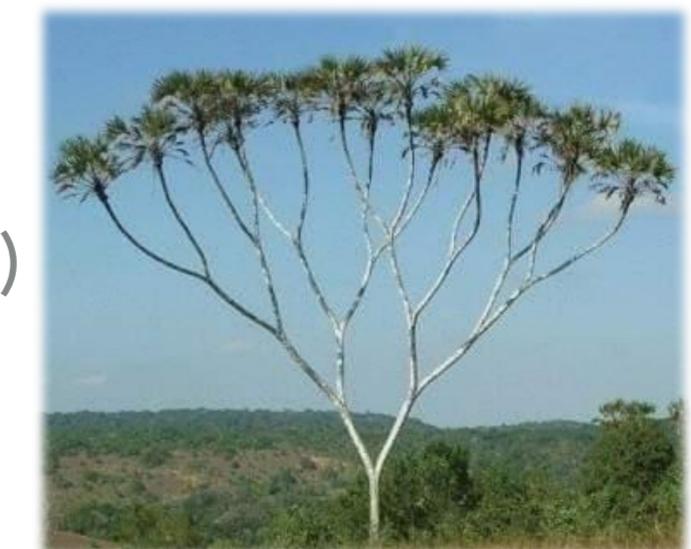
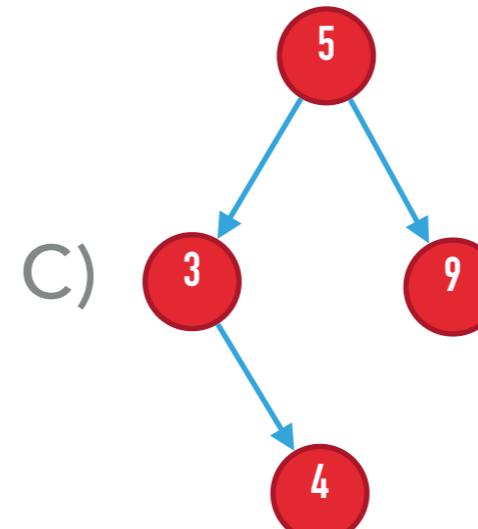
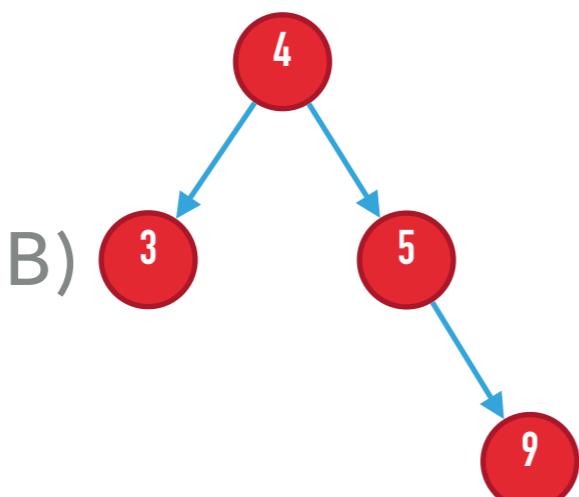
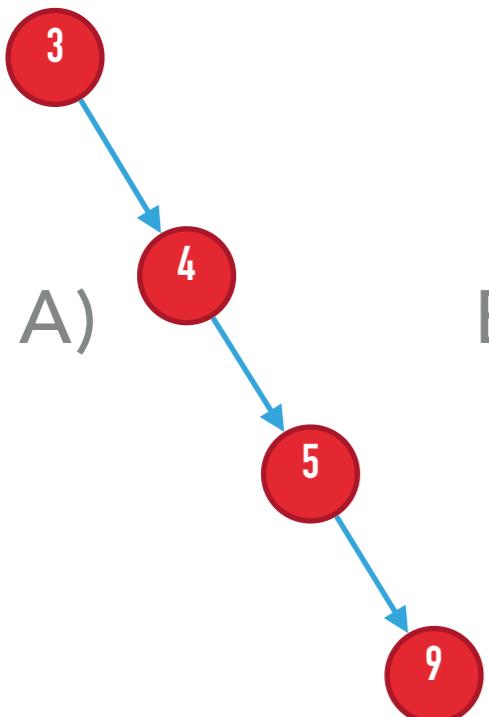
- ▶ Prima di effettuare una chiamata ricorsiva effettuiamo un numero costante di passi
- ▶ Ogni chiamata ricorsiva ci fa scendere di un livello nell'albero
- ▶ Quindi la complessità dell'inserimento dipende, come per la ricerca, dalla profondità dell'albero: $O(h)$

QUIZ: INSERIMENTO

Supponiamo di inserire in un albero inizialmente vuoto i seguenti valori nell'ordine in cui appaiono:

4 5 9 3

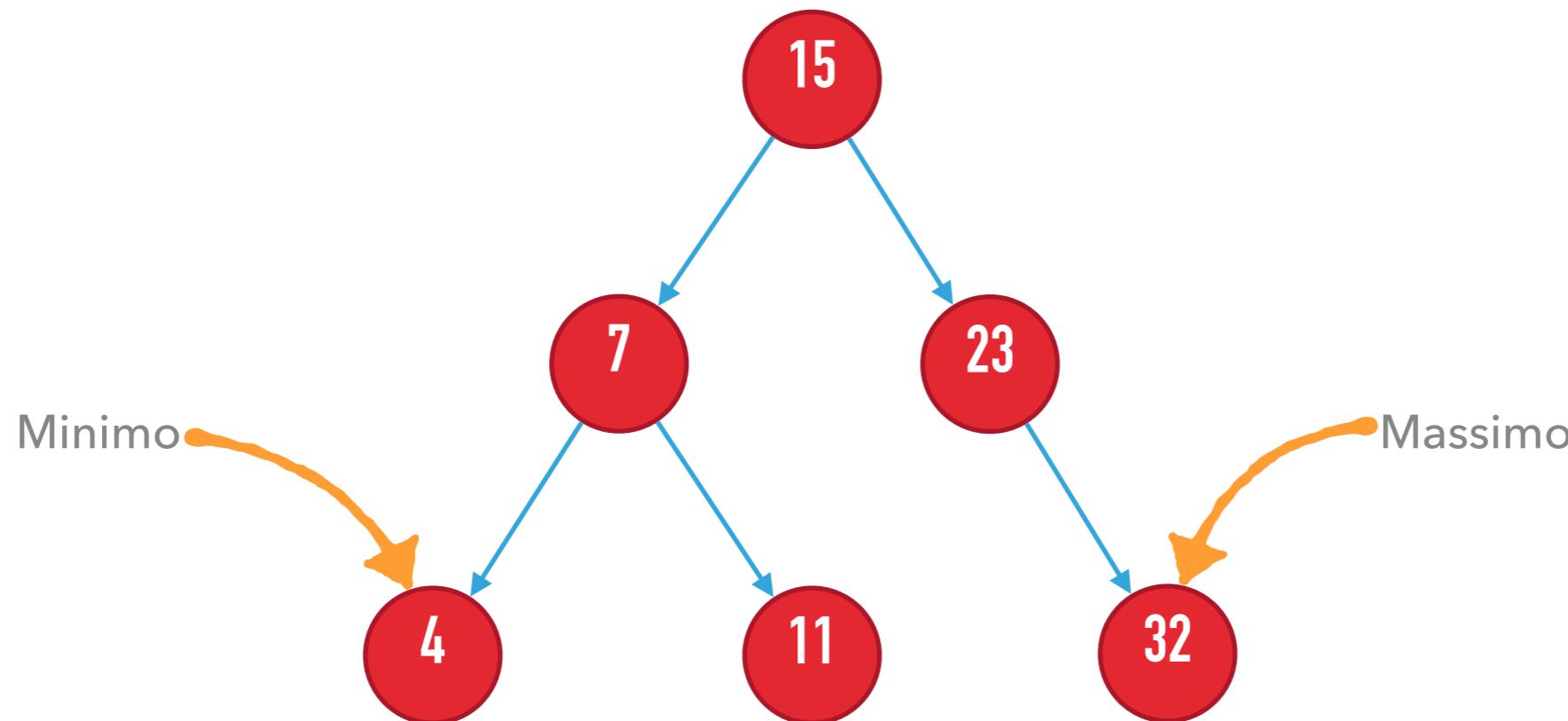
Quale di questi è l'albero risultante?



ALBERI BINARI: MASSIMO E MINIMO

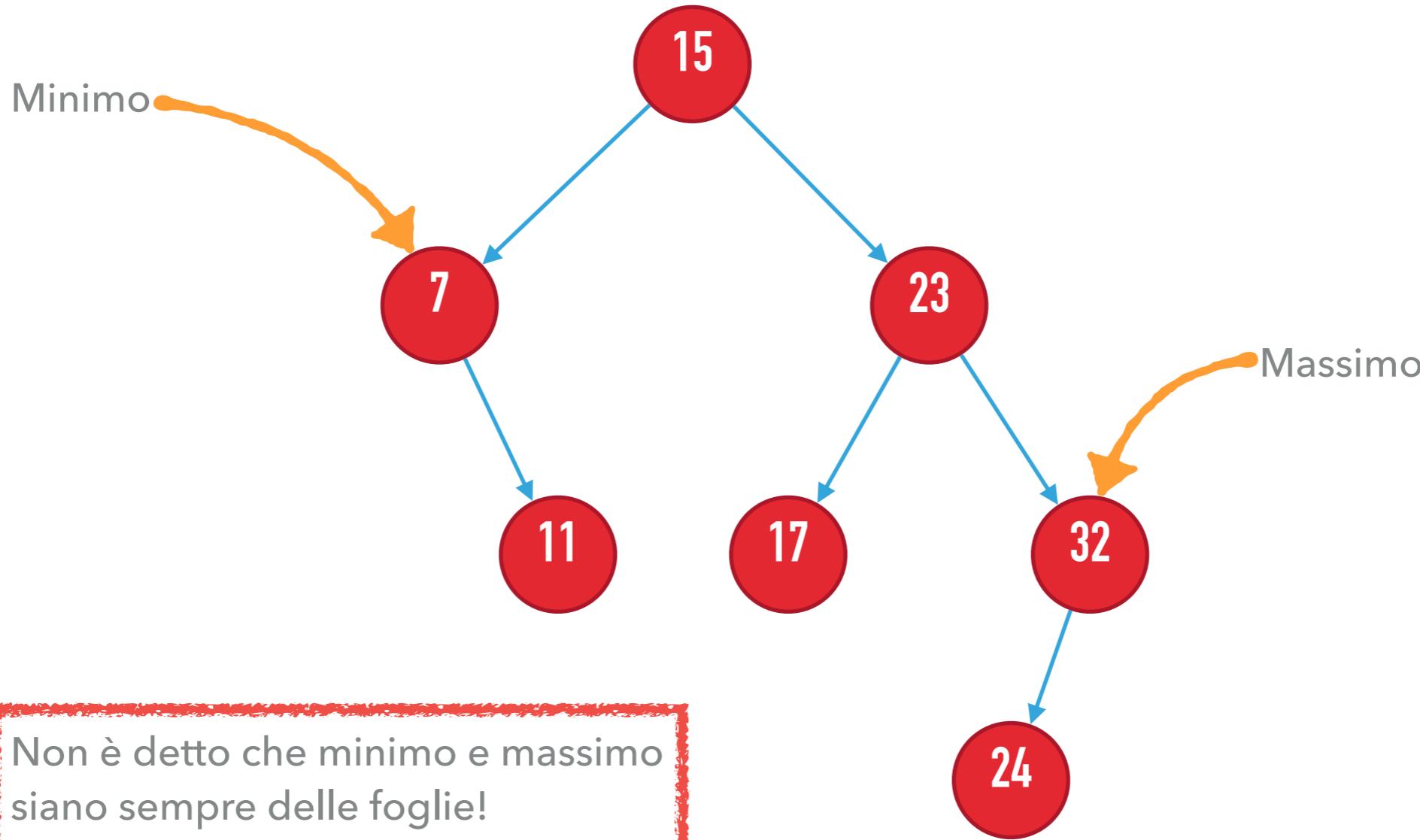
- ▶ Per definizione il massimo sarà il nodo “più a destra” nell’albero
- ▶ È sufficiente muoversi dalla radice verso destra fino a quando si incontra un nodo senza figlio destro: il valore che contiene è il massimo
- ▶ Simmetricamente per il minimo: è sufficiente continuare a spostarsi verso sinistra

ALBERO BINARIO DI RICERCA



Dato che dobbiamo “scendere” fino alle foglie, nel caso peggiore siamo comunque limitati dall’altezza dell’albero per trovare il minimo o il massimo

ALBERO BINARIO DI RICERCA

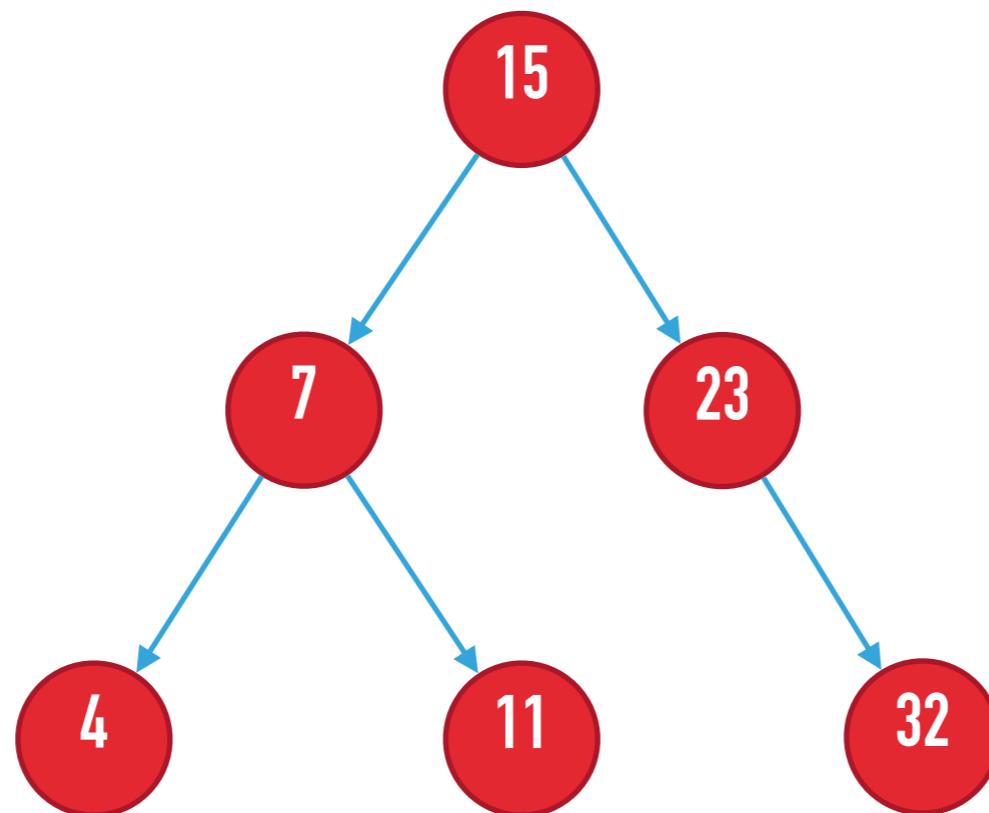


ALBERI BINARI: SUCCESSORE

- ▶ Vediamo come implementare il successore, il predecessore è simmetrico
- ▶ Caso semplice: il nodo (di chiave k) ha un figlio destro
 - ▶ Allora la chiave più piccola strettamente più grande di k è il minimo del sottoalbero avente radice il figlio destro
- ▶ Caso difficile: il nodo non ha figlio destro
 - ▶ Allora dobbiamo risalire nella catena di genitori

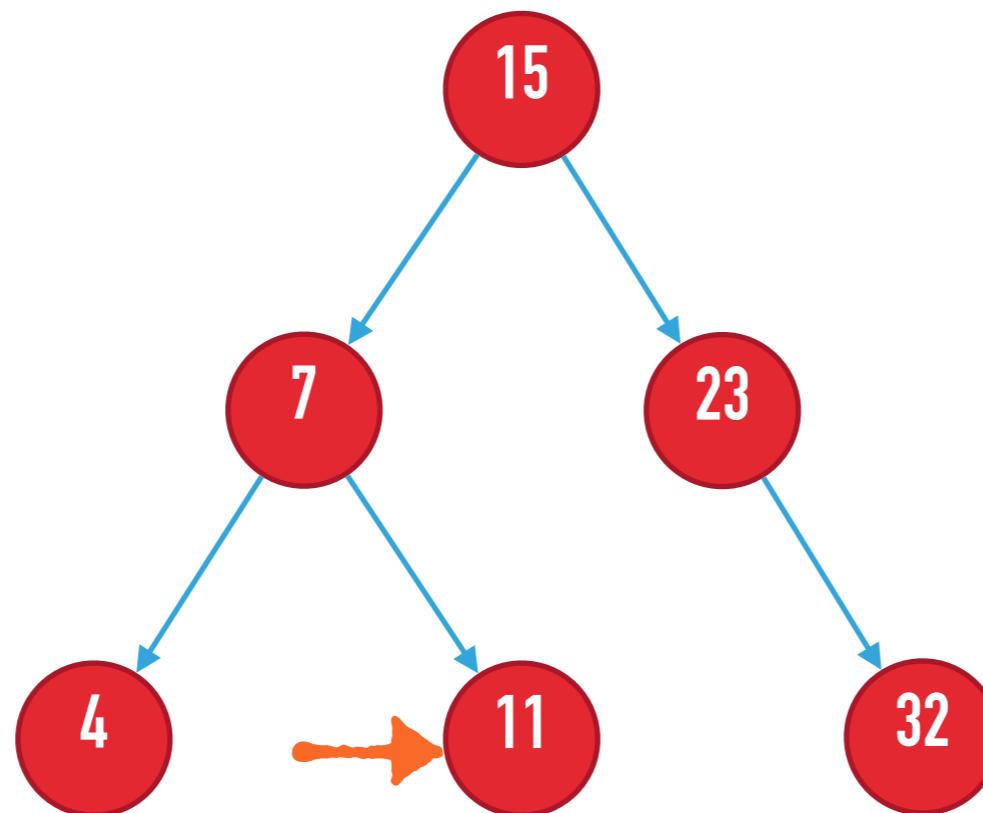
ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di "11"



ALBERO BINARIO: SUCCESSORE

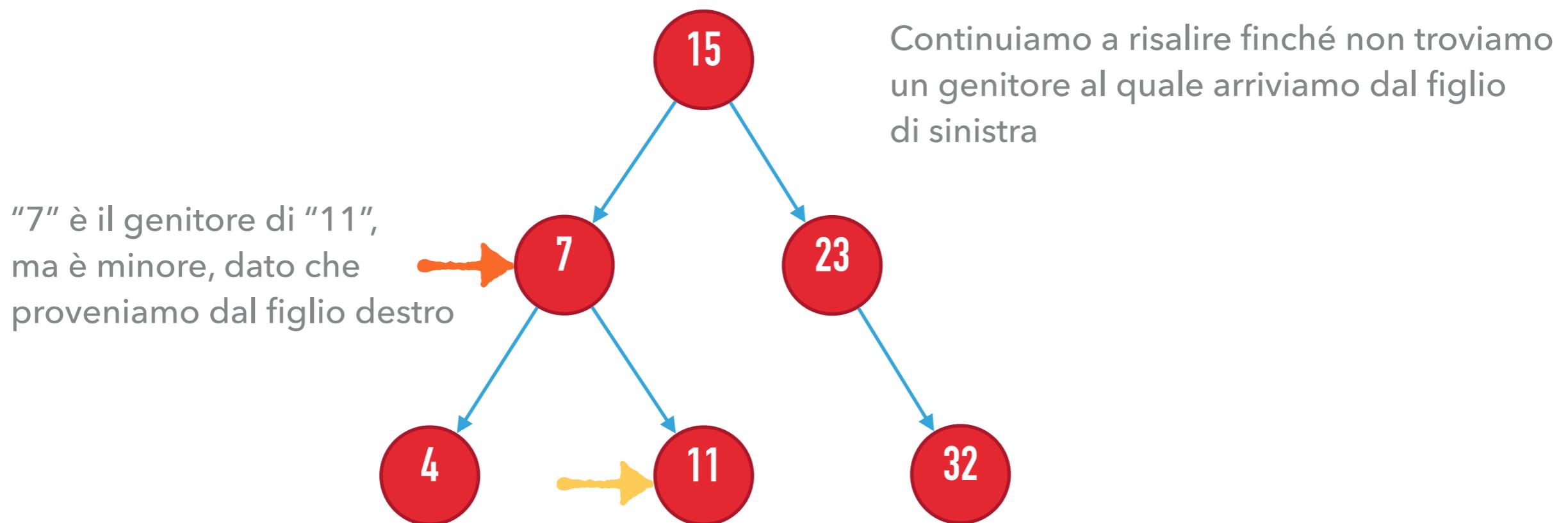
Supponiamo di voler trovare il successore di "11"



"11" non ha un figlio destro, quindi cerchiamo tra i genitori

ALBERO BINARIO: SUCCESSORE

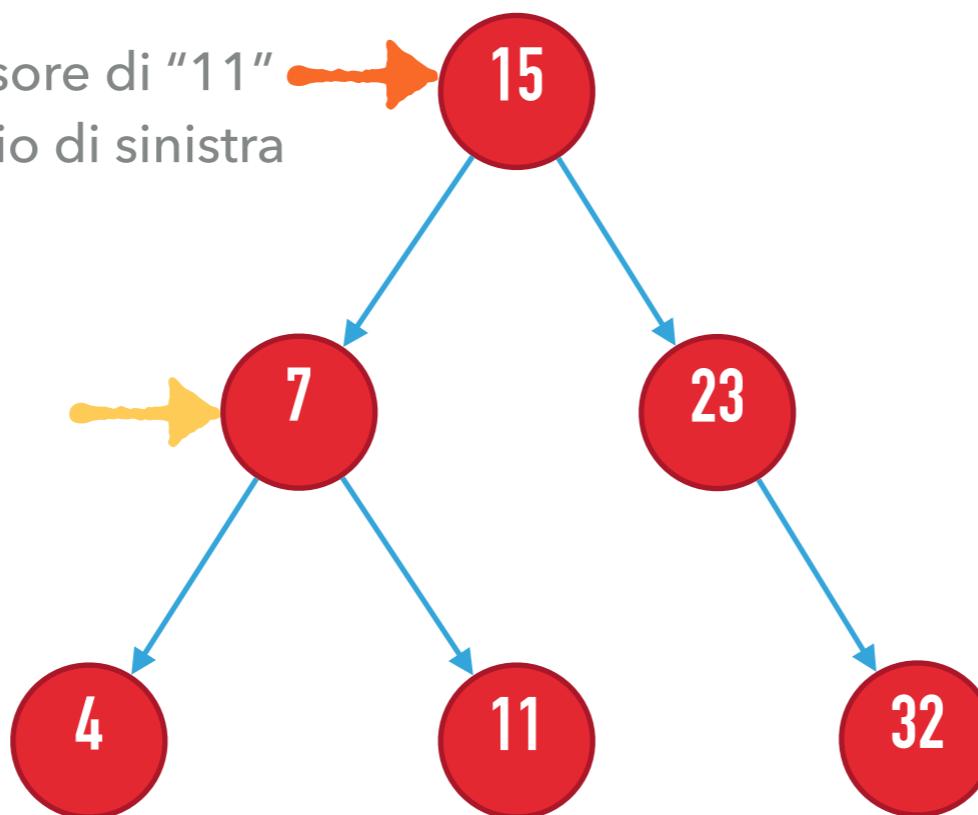
Supponiamo di voler trovare il successore di "11"



ALBERO BINARIO: SUCCESSORE

Supponiamo di voler trovare il successore di "11"

Abbiamo trovato il successore di "11" →
dato che arriviamo dal figlio di sinistra



ALBERI BINARI: PSEUDOCODICE DEL SUCCESSORE

- ▶ Parametri: Nodo
- ▶ if nodo.right is not None:
 - ▶ return minimo(nodo.right)
- ▶ x = nodo
- ▶ y = nodo.parent
- ▶ While y is not None and y.right is x
 - ▶ # finché esiste un genitore e proveniamo dal figlio di destra
 - ▶ x = x.parent # saliamo di un livello
 - ▶ y = y.parent
- ▶ return y

ALBERI BINARI: SUCCESSORE

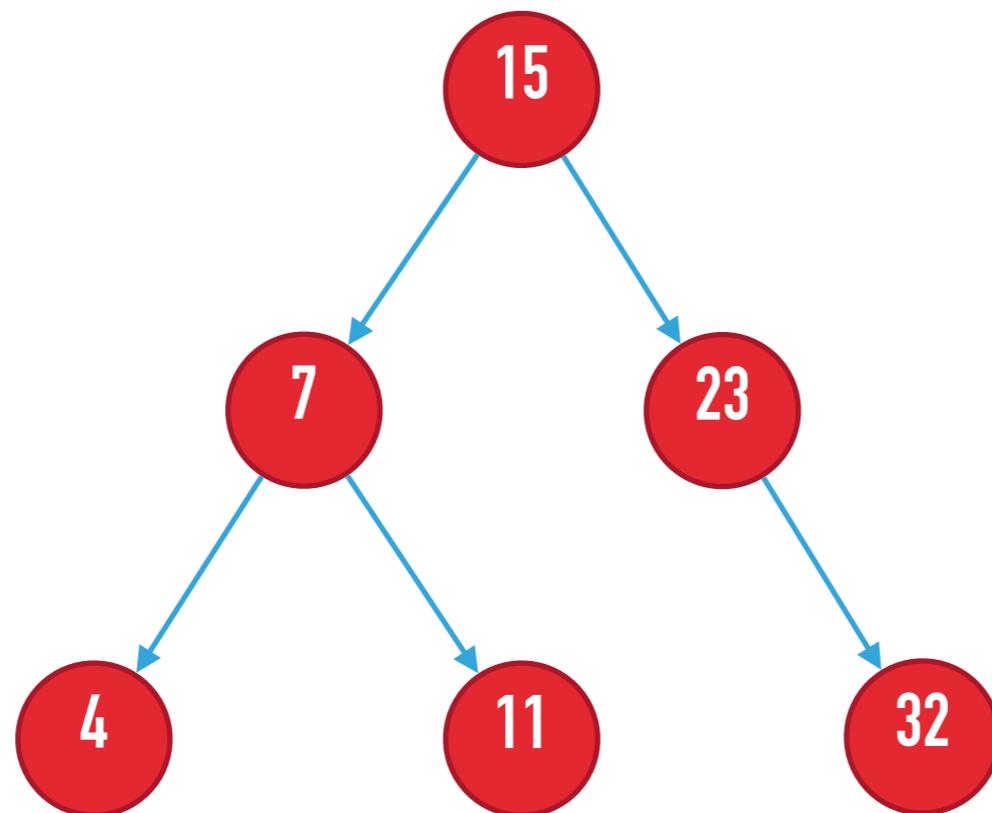
- ▶ La complessità di trovare il successore può essere divisa in due casi:
 - ▶ Uguale alla complessità di trovare il minimo se esiste un figlio destro
 - ▶ Se il figlio destro non esiste dobbiamo risalire l'albero... ma il numero di volte che risaliamo è comunque limitato dall'altezza dell'albero.
- ▶ Quindi trovare il successore richiede tempo $O(h)$

ALBERI BINARI: CANCELLAZIONE

- ▶ La rimozione di un elemento è l'operazione più complessa
- ▶ Abbiamo tre casi a seconda del numero di figli che ha un nodo:
 - ▶ Il nodo non ha figli: la rimozione è facile
 - ▶ Il nodo ha un solo figlio: facciamo prendere al figlio il posto del genitore
 - ▶ Il nodo ha due figli... questo è più complesso

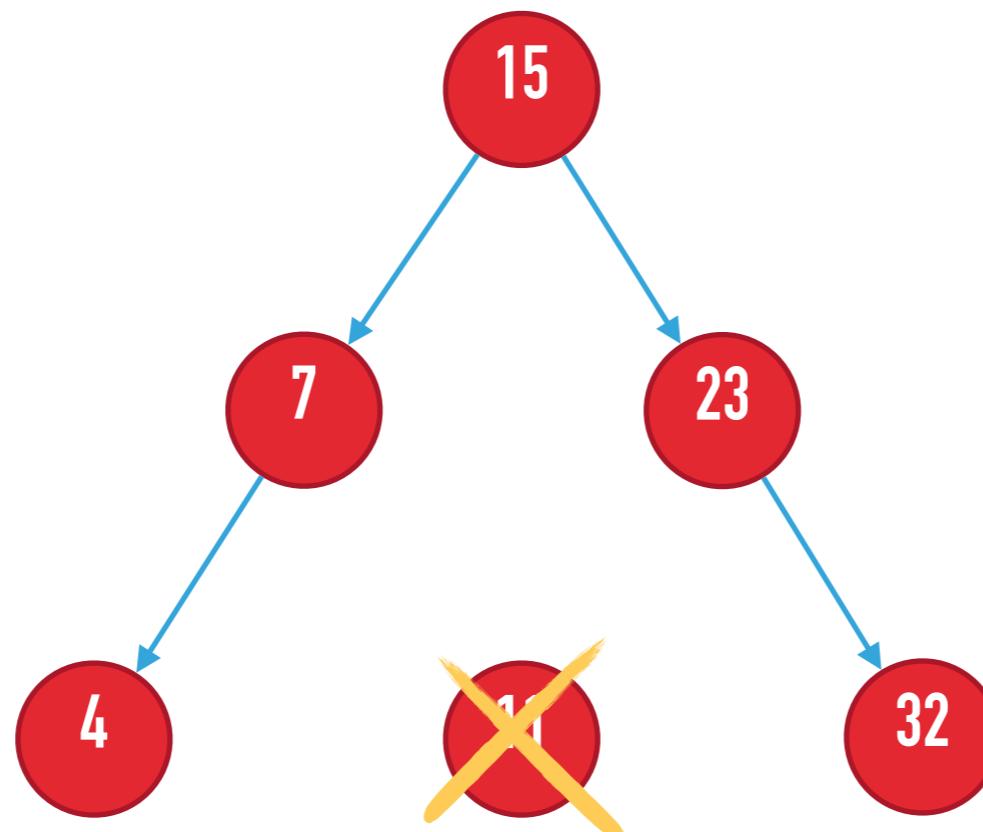
ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare "11"



ALBERO BINARIO: CANCELLAZIONE

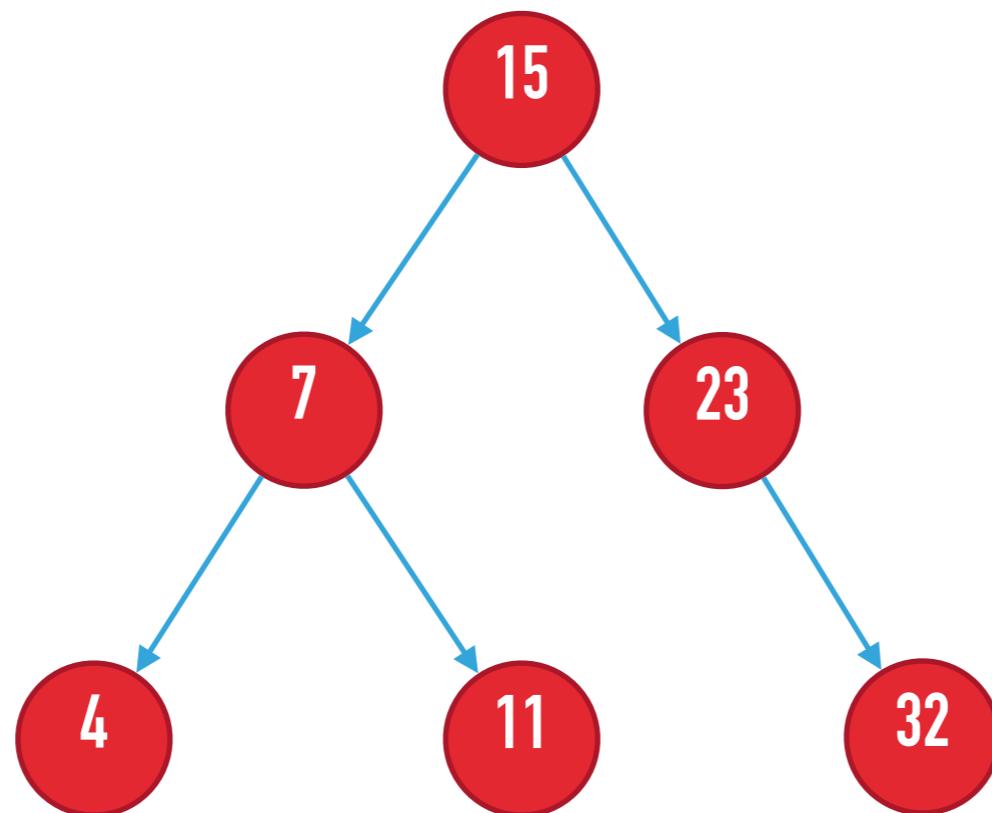
Supponiamo di voler cancellare "11"



"11" Non ha figli, quindi possiamo eliminarlo senza problemi

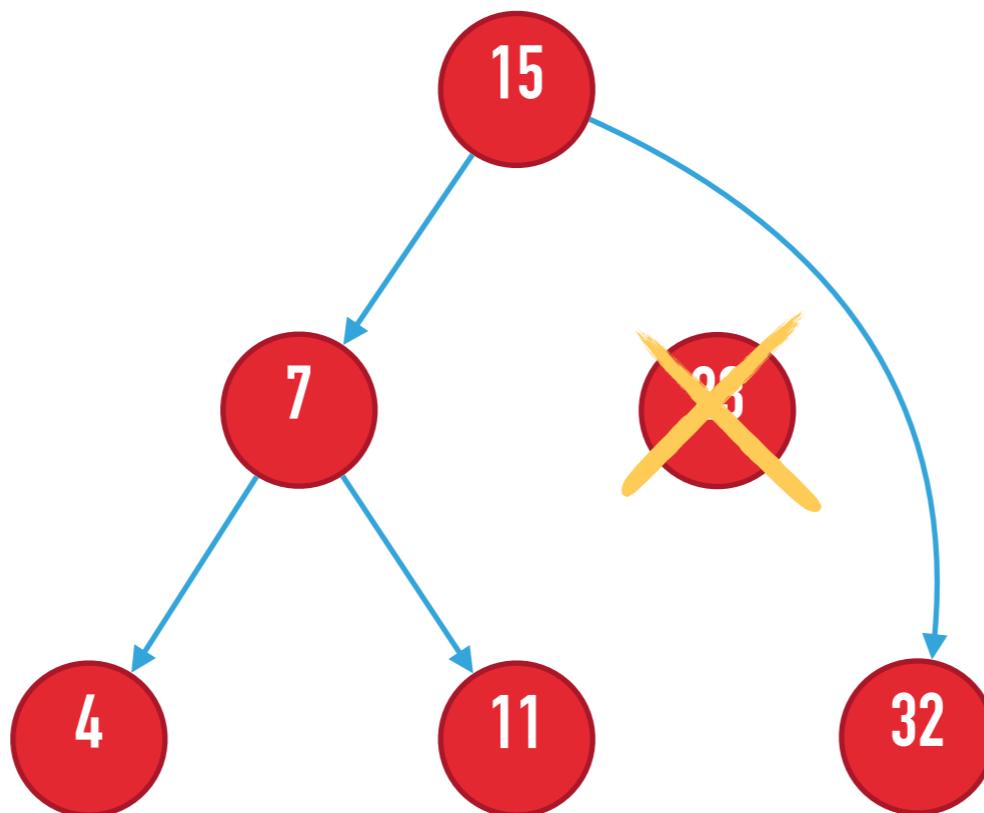
ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare "23"



ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare "23"



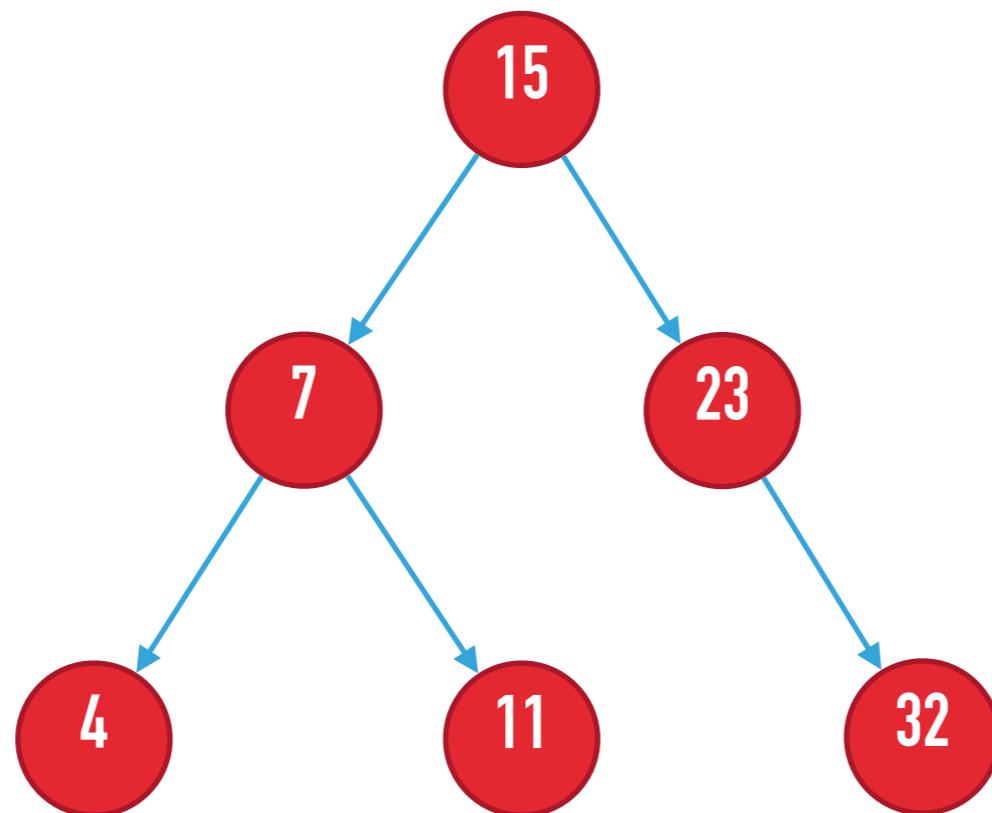
Avendo un solo figlio, possiamo rimpiazzare "23"
col suo sottoalbero destro

ALBERI BINARI: CANCELLAZIONE

- ▶ Per cancellare nel caso di due figli possiamo dividere la procedura in due passi:
- ▶ Mettiamo il successore del nodo da cancellare nel posto del nodo cancellato (il successore è necessariamente nel sottoalbero destro)
- ▶ Dato che il successore era in una posizione con solamente un figlio (è il minimo del sottoalbero, non può avere figlio sinistro), possiamo eliminarlo senza problemi (equivalentemente possiamo usare il predecessore)

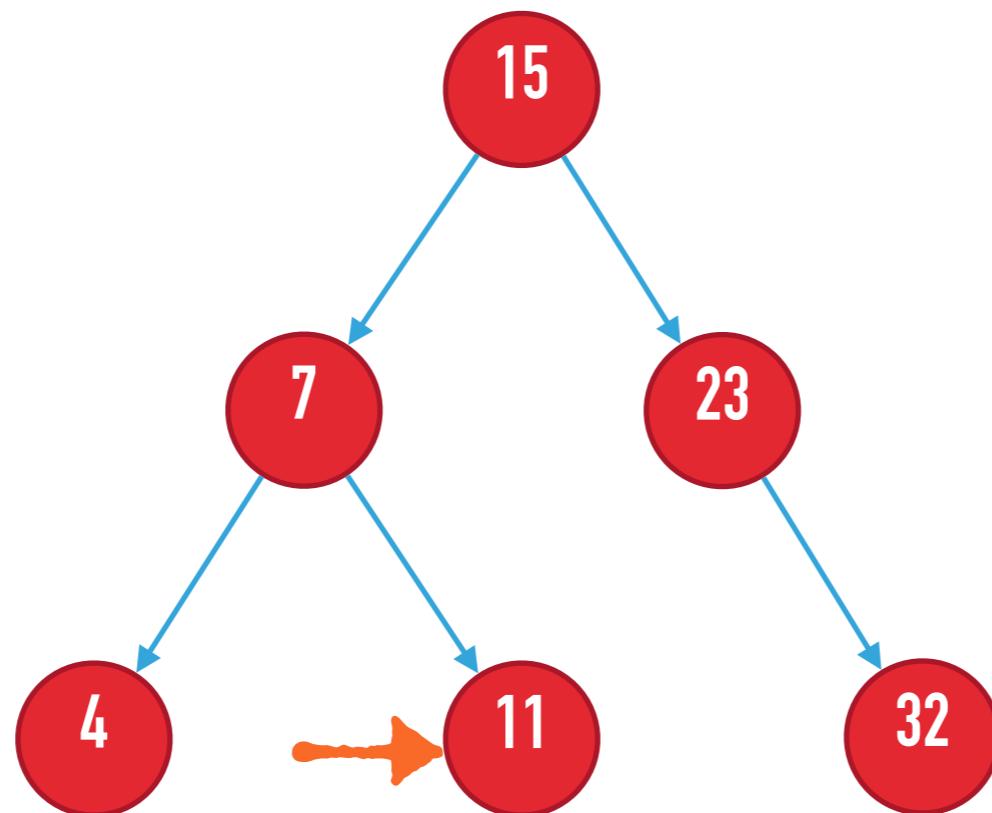
ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare "15"



ALBERO BINARIO: CANCELLAZIONE

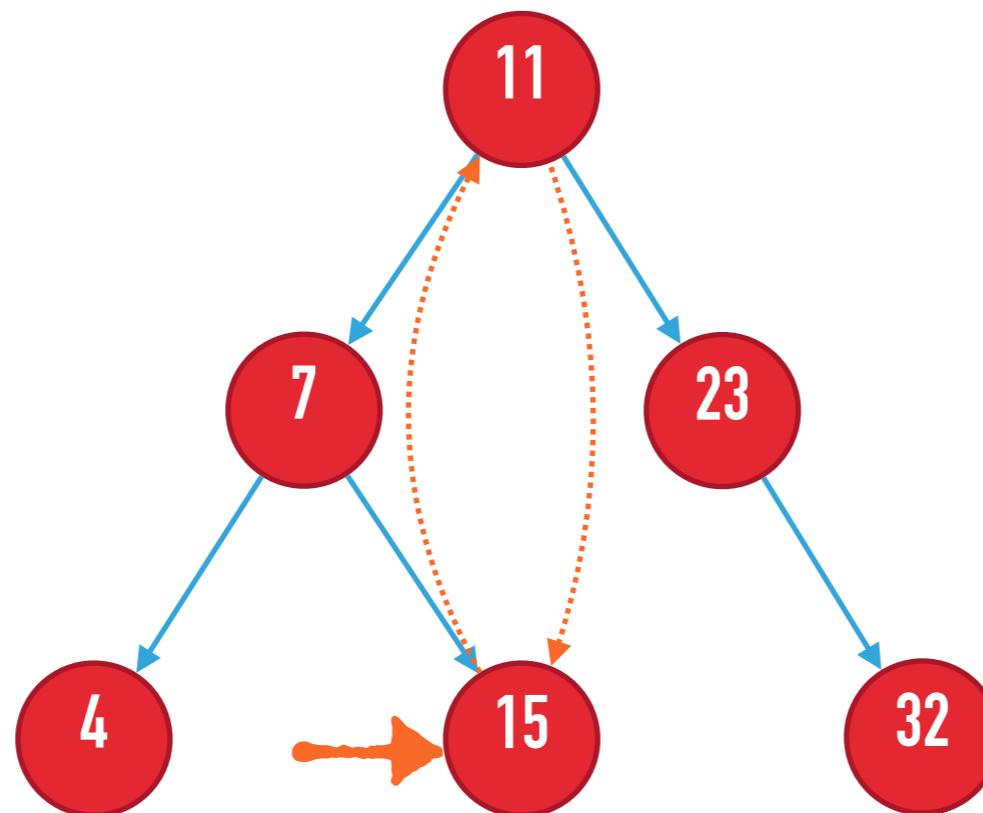
Supponiamo di voler cancellare “15”



Individuiamo il predecessore di “15”
(sarebbe identico usando il successore)

ALBERO BINARIO: CANCELLAZIONE

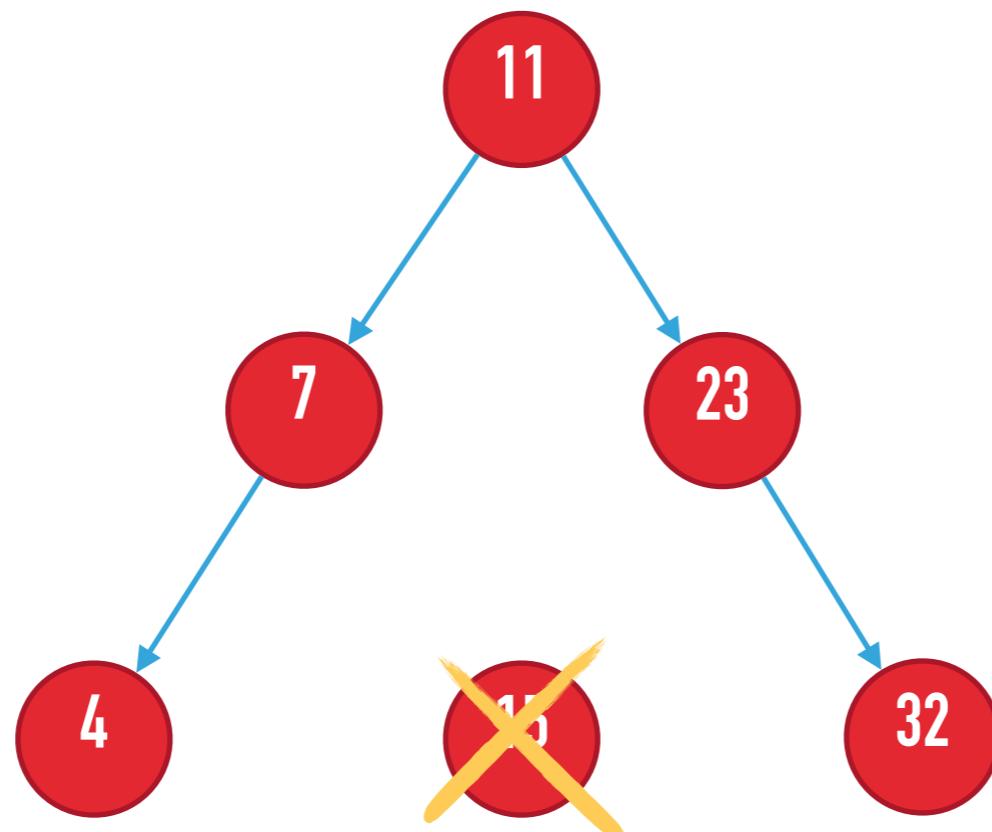
Supponiamo di voler cancellare “15”



Scambiamo di posto il nodo da cancellare ed il predecessore

ALBERO BINARIO: CANCELLAZIONE

Supponiamo di voler cancellare “15”



Eliminiamo il nodo, cosa possibile dato che ha zero o uno figli

ALBERI BINARI: CANCELLAZIONE

- ▶ La complessità della cancellazione dipende dal tipo di rimozione che dobbiamo fare:
 - ▶ Rimuovere un nodo con zero o un figlio richiede tempo costante
 - ▶ Rimuovere un nodo con due figli richiede di trovare il nodo successore, che richiede tempo $O(h)$
- ▶ Anche in questo caso la complessità dipende dall'altezza

ALBERI BINARI: DISCUSSIONE

- ▶ Tutte le operazioni viste dipendono dall'altezza dell'albero
- ▶ A seconda dell'ordine in cui vengono inseriti i dati è possibile ottenere grandi differenze in termini di altezza:
 - ▶ Alberi bilanciati a profondità $O(\log n)$
 - ▶ Ma nei casi più sbilanciati la profondità è $O(n)$
- ▶ Esistono varianti di alberi in grado di mantenere il bilanciamento (e.g., red-black trees, alberi AVL)

ALBERI SPLAY

ALGORITMI E STRUTTURE DATI

ALBERI SPLAY

- ▶ Gli alberi binari di ricerca possono avere il problema che, a seconda dell'ordine di inserimento, possono avere profondità lineare
- ▶ Come primo esempio di albero che si modifica vediamo gli alberi splay (splay tree)
- ▶ Ideati nel 1985 da Sleator e Tarjan
- ▶ Ci serviranno per introdurre il concetto di rotazione per modificare la forma degli alberi

ALBERI SPLAY

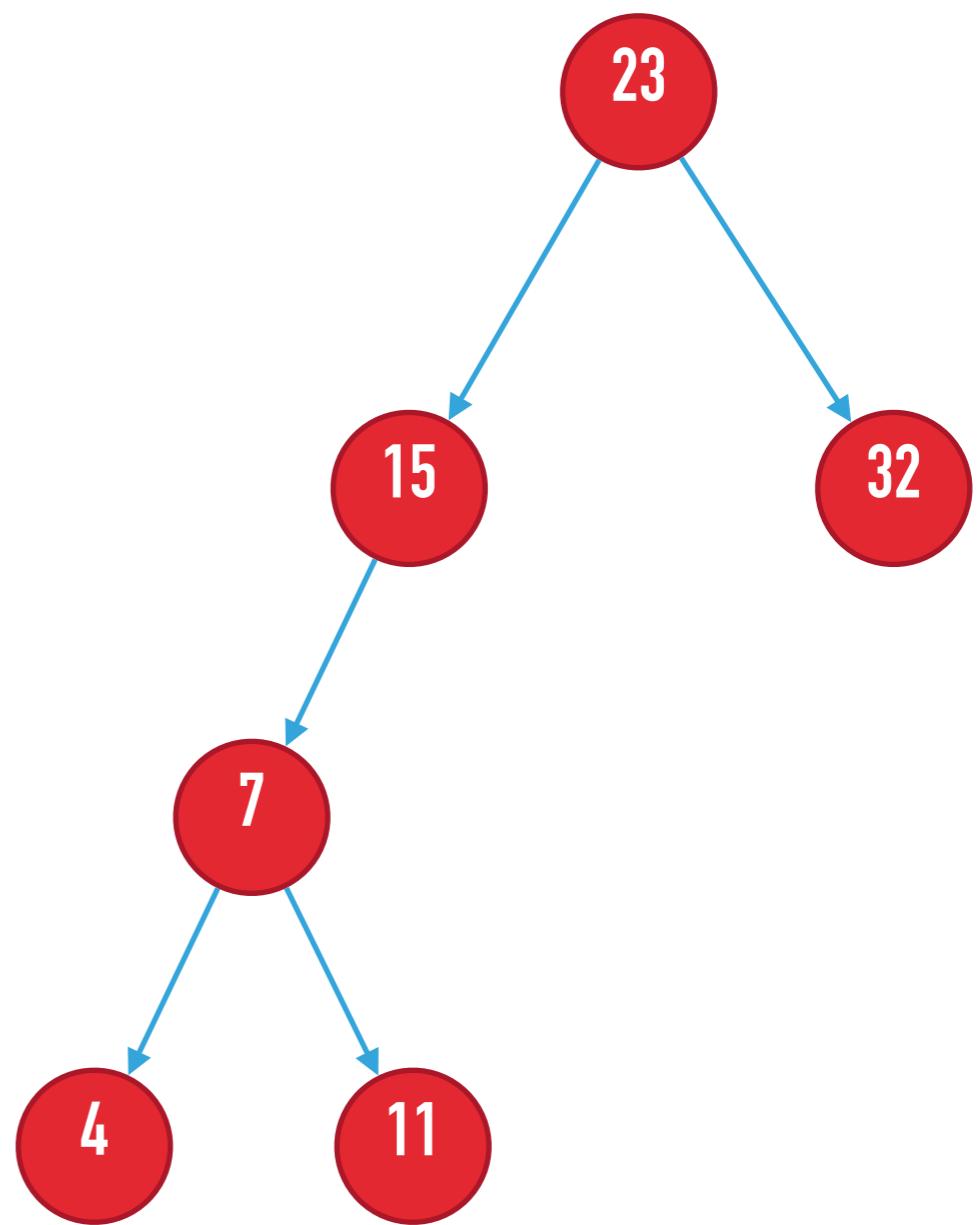
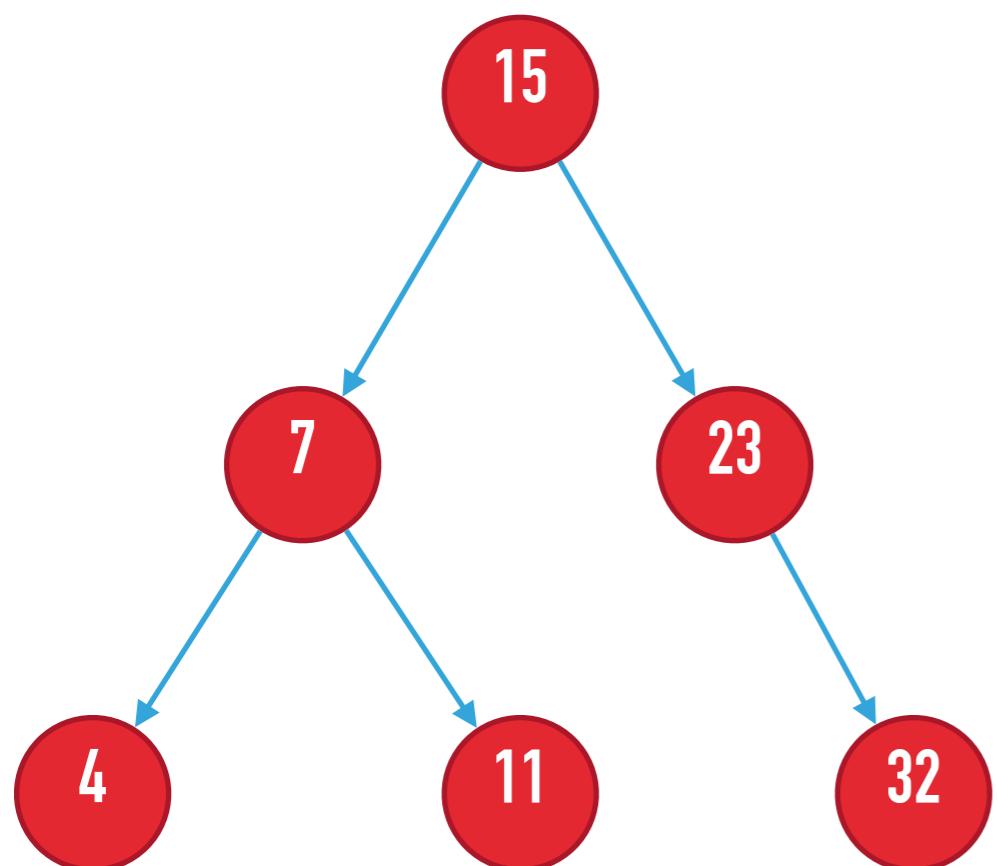
- ▶ L'idea di base è di spostare gli elementi che cerchiamo alla radice dell'albero
- ▶ L'effetto è che tutti gli elementi cercati di recente vengono a trovarsi vicino alla radice
- ▶ Questo non mantiene necessariamente l'albero bilanciato: il caso peggiore è ancora $O(n)$
- ▶ Ci sono però una serie di altri vantaggi (e.g., costo ammortizzato di una sequenza di operazioni)

ROTAZIONI

- ▶ Ruotare un nodo è l'operazione che ci permette di scambiare la posizione di un nodo e del suo figlio sinistro o destro
- ▶ Rotazione a sinistra di x : il nodo x viene sostituito dal suo figlio destro e ne diventa il figlio sinistro
- ▶ Rotazione a destra di x : il nodo x viene sostituito dal suo figlio sinistro e ne diventa il figlio destro

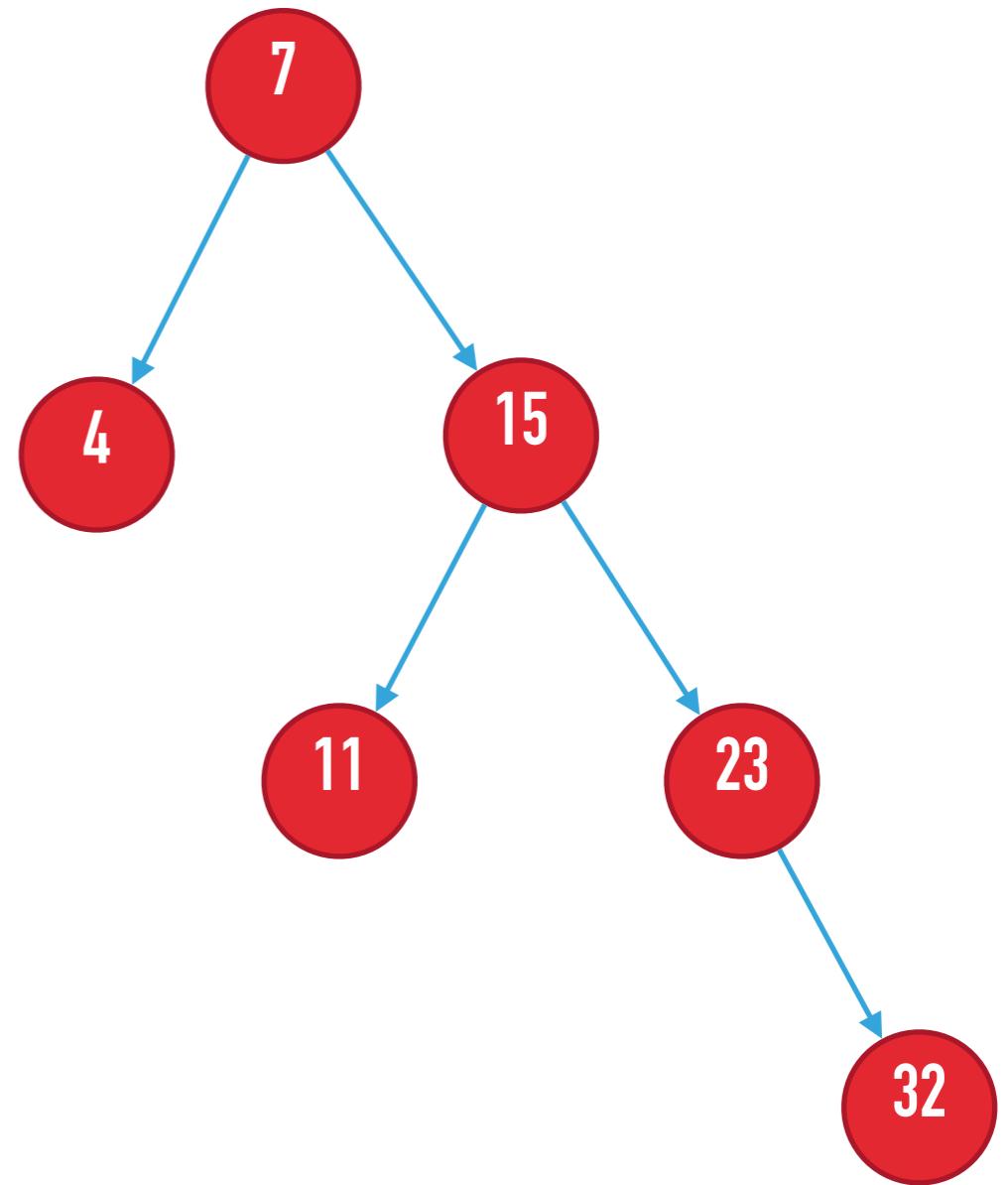
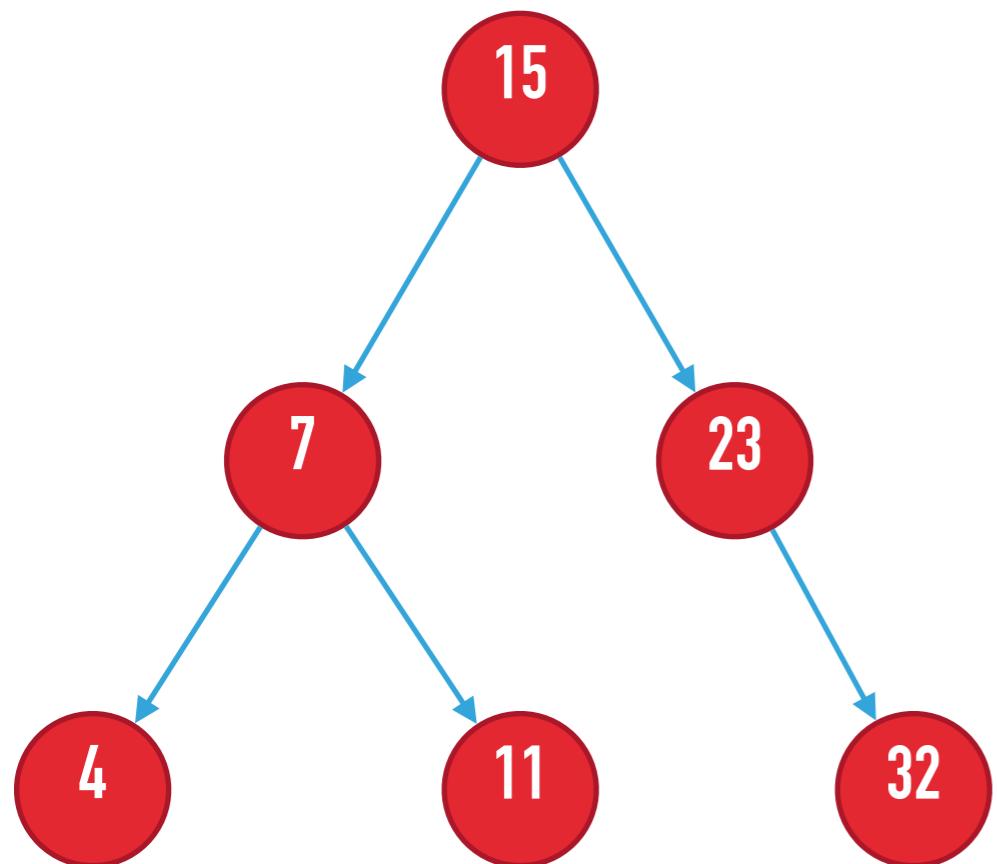
ROTAZIONE A SINISTRA

Vogliamo ruotare a sinistra “15”



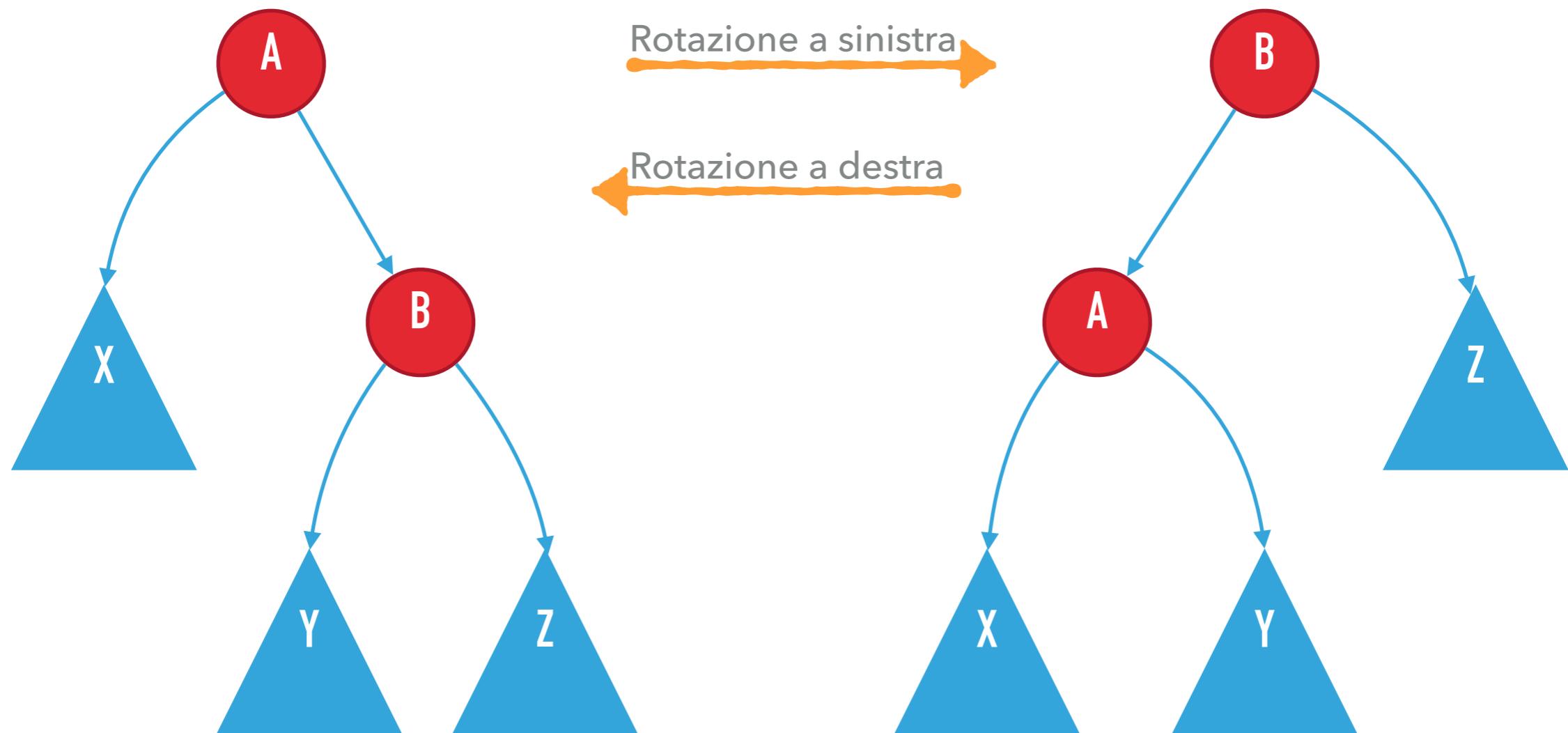
ROTAZIONE A DESTRA

Vogliamo ruotare a destra "15"



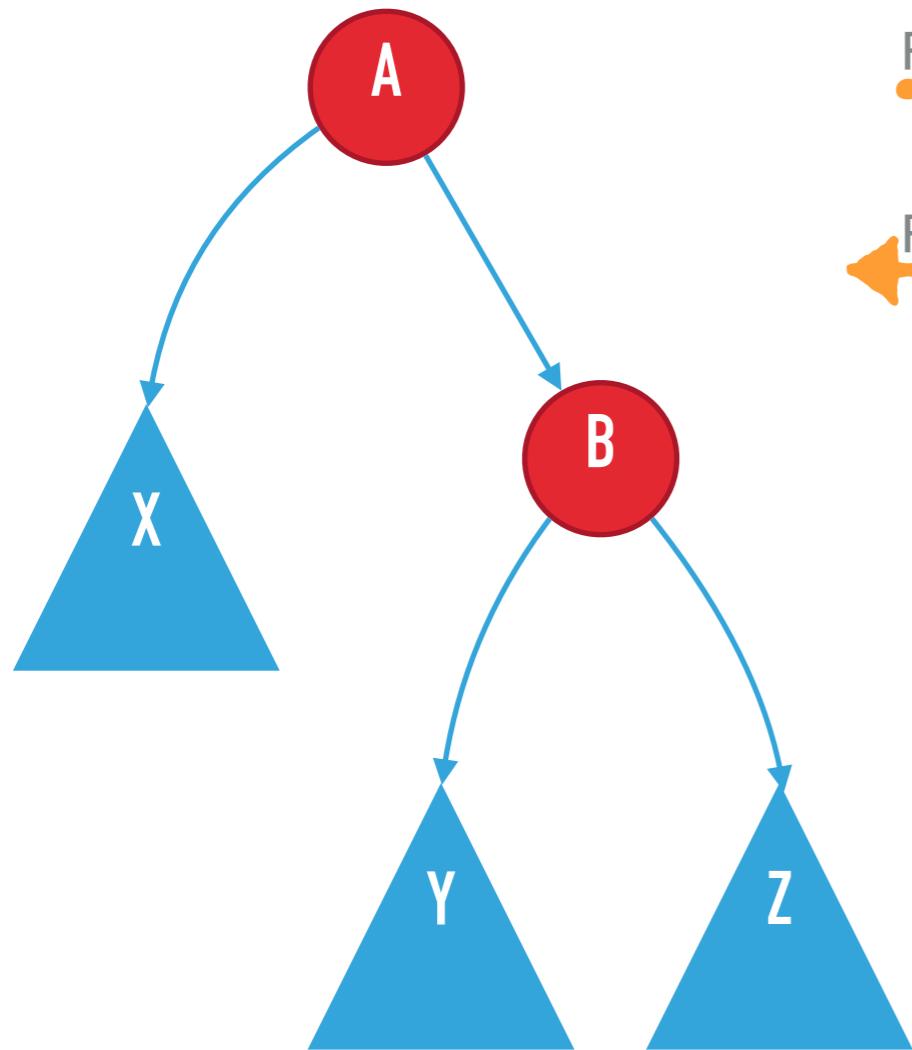
ROTAZIONE

Astraiamo e vediamo come la rotazione agisce su due nodi generici

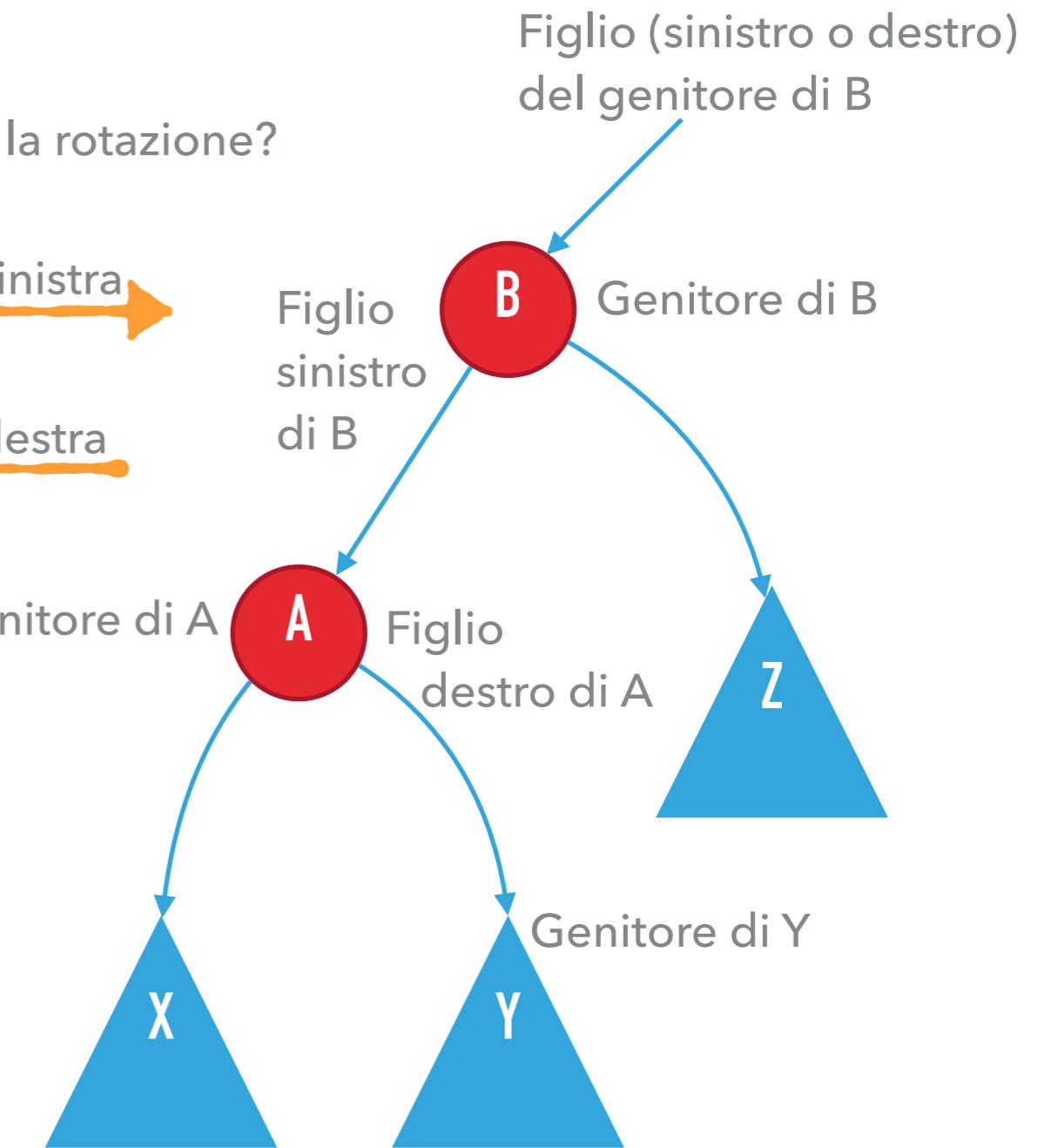


ROTAZIONE: COSA BISOGNA MODIFICARE

Cosa dobbiamo modificare nei vari nodi per effettuare la rotazione?



Rotazione a sinistra
Rotazione a destra



ROTAZIONI

- ▶ Parametri: nodo a
- ▶ b = a.right # nodo che prenderà il posto di a
- ▶ a.right = b.left
- ▶ if a.right is not None:
 - ▶ a.right.parent = a
- ▶ b.left = a
- ▶ # con queste operazioni abbiamo messo il figlio sinistro di b come figlio destro di a, dobbiamo ancora sistemare i genitori di a e b
- ▶ if a.parent is not None: # sistemiamo il genitore di a
 - ▶ if a.parent.left is a: # nel caso a fosse figlio sinistro
 - ▶ a.parent.left = b
 - ▶ else # nel caso a fosse figlio destro
 - ▶ a.parent.right = b
- ▶ b.parent = a.parent
- ▶ a.parent = b

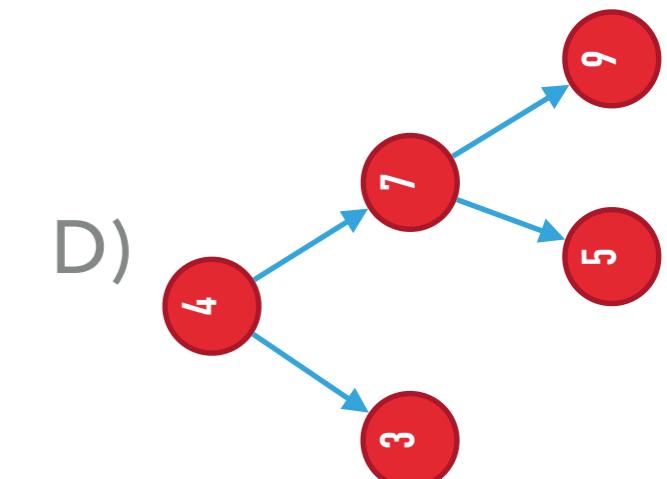
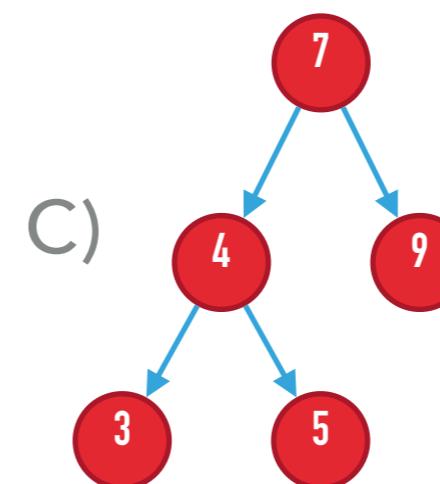
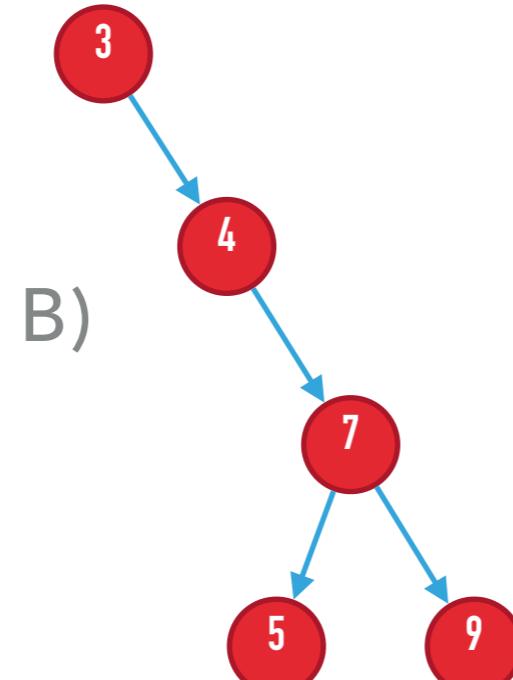
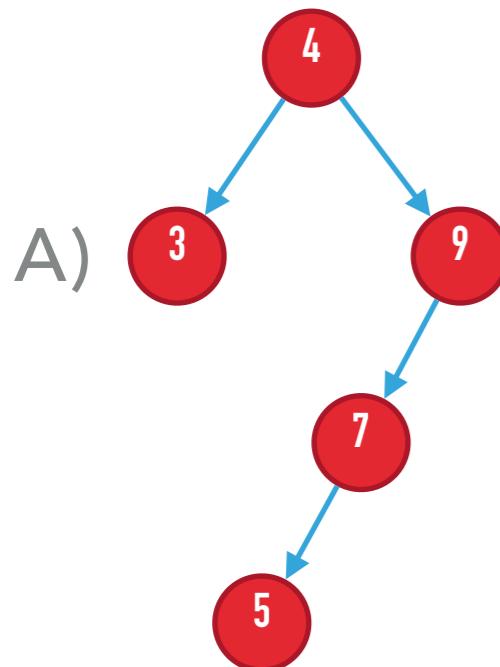
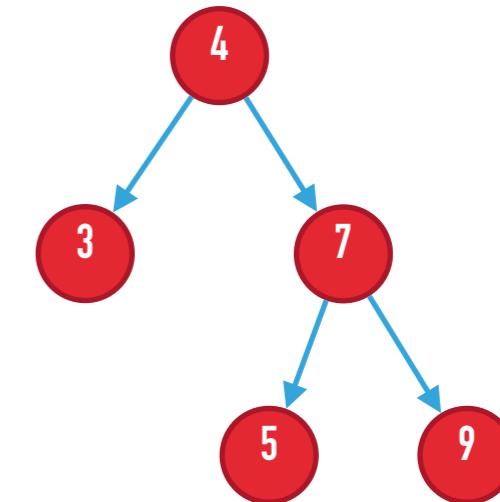
ROTAZIONI

- ▶ Ogni rotazione richiede tempo costante
- ▶ Ci permette di “muovere” un nodo lungo l’albero facendolo salire o scendere tramite una serie di rotazioni
- ▶ Negli alberi Splay useremo le rotazioni per far risalire un nodo fino a farlo diventare la radice
- ▶ In altri alberi che garantiscono il bilanciamento (come gli alberi AVL) le rotazioni sono usate per bilanciare dopo ogni inserimento o rimozione

QUIZ: ROTAZIONI

Supponiamo di avere il seguente albero binario:

Quale è il risultato di ruotare a sinistra “4”?

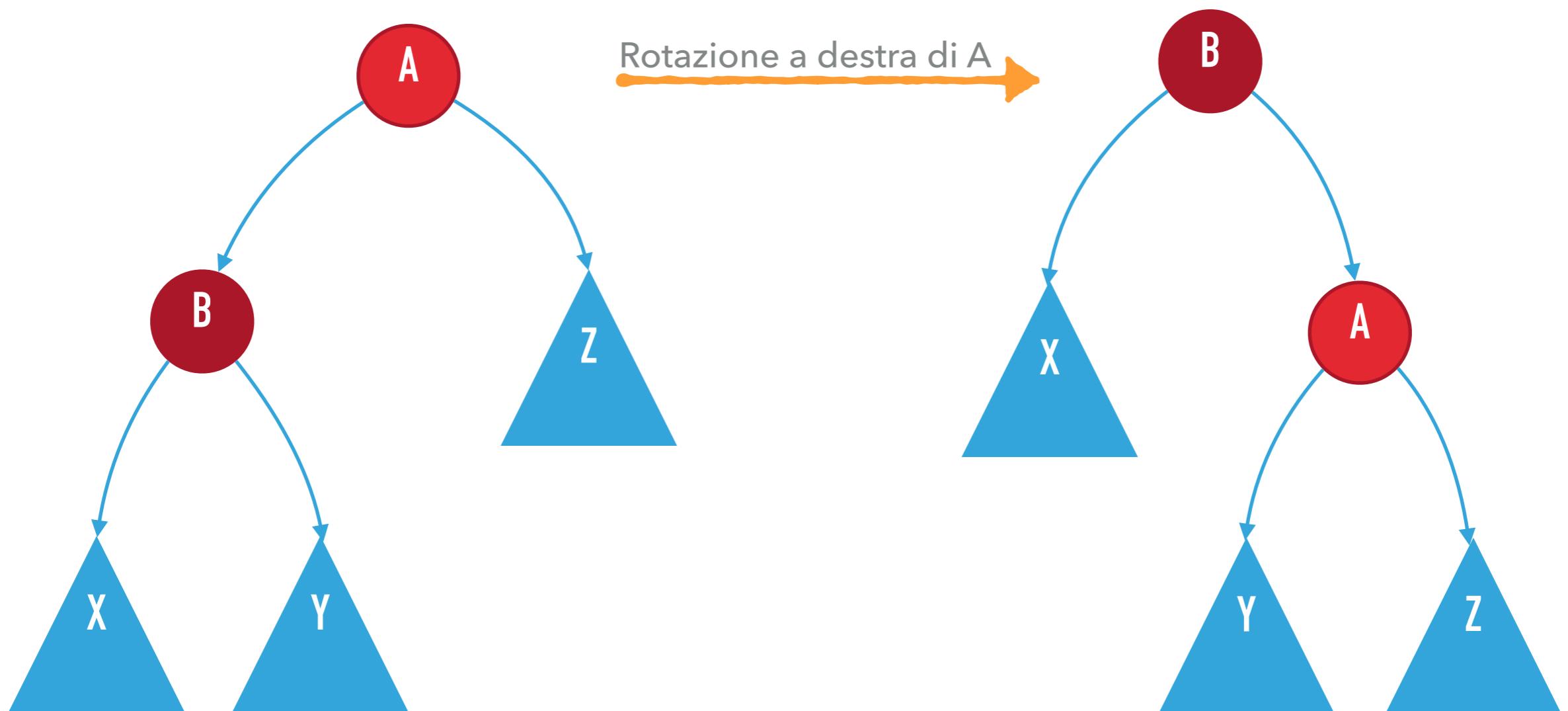


ALBERI SPLAY

- ▶ Ogni volta che cerchiamo un elemento nell'albero chiamiamo l'operazione di "splay" o "muovi alla radice" che sposta l'elemento cercato alla radice dell'albero
- ▶ Questo viene fatto con rotazioni secondo tre casi:
 - ▶ Zig.
 - ▶ Zig zig.
 - ▶ Zig zag.

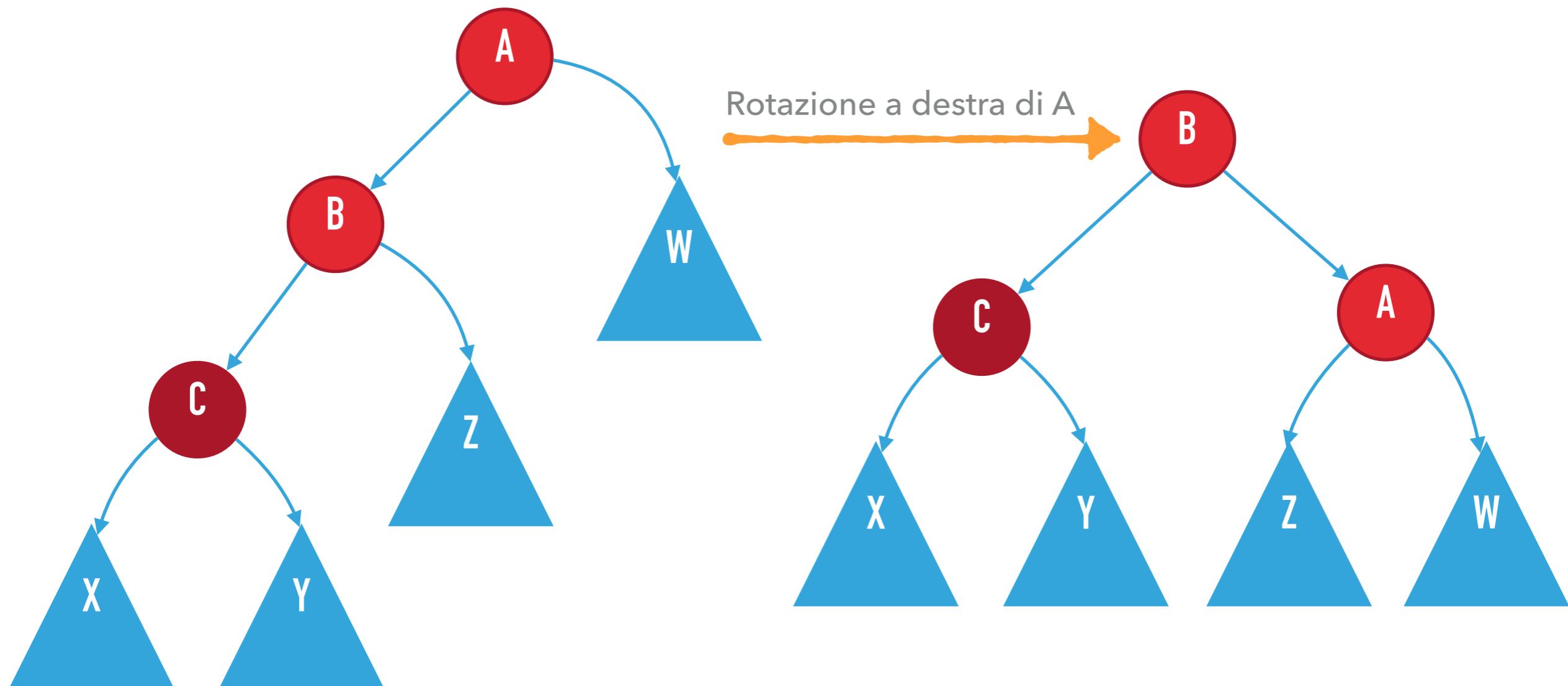
CASO ZIG

Il nodo da muovere è figlio sinistro della radice



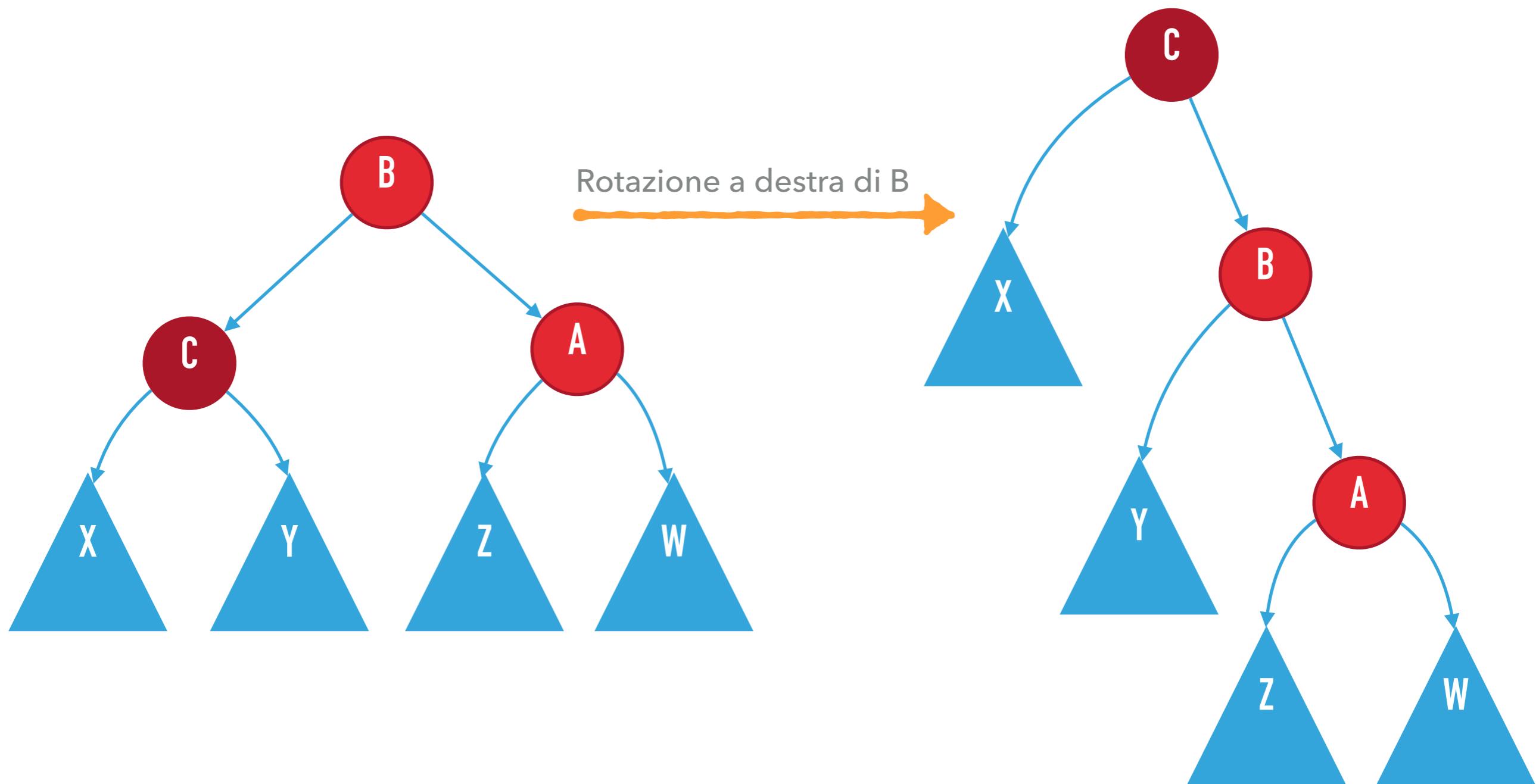
CASO ZIG ZIG

Il nodo da muovere è figlio sinistro di un nodo che è a sua volta figlio sinistro



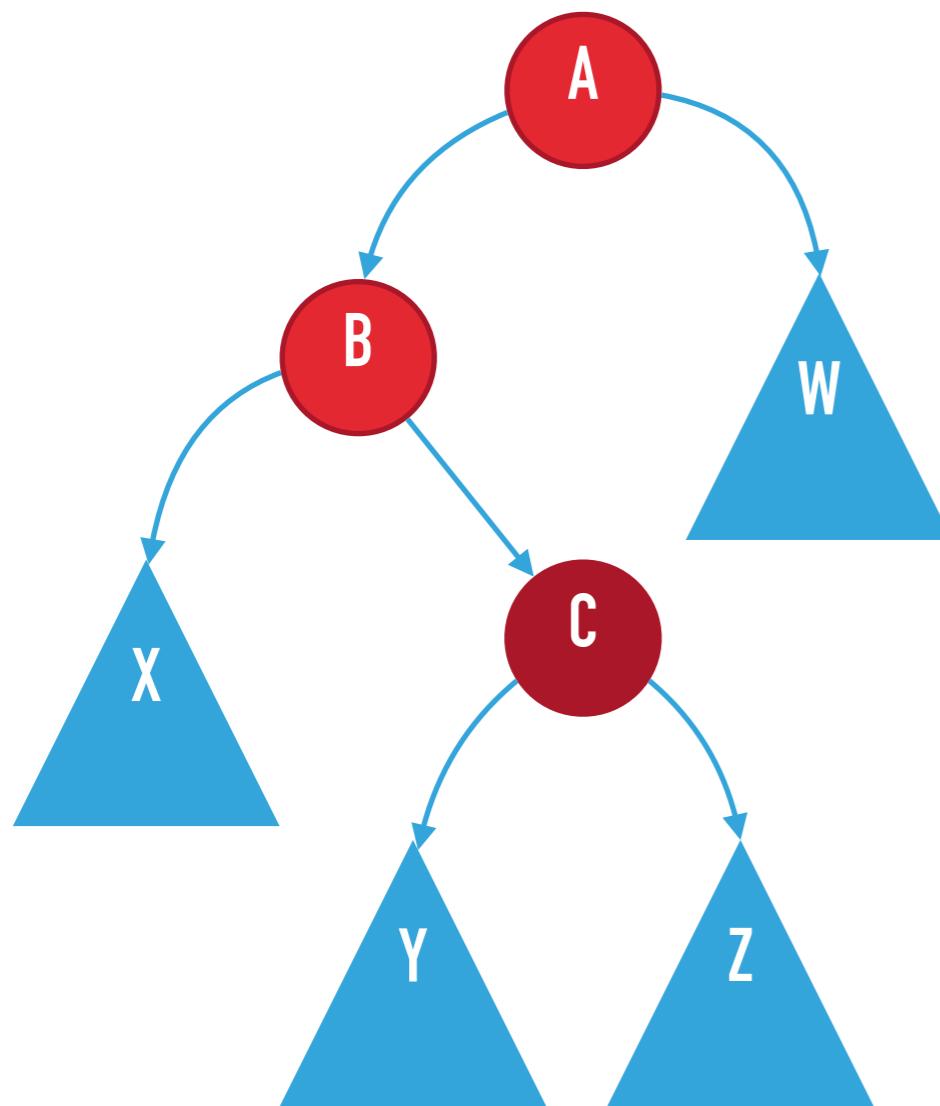
CASO ZIG ZIG

Il nodo da muovere è figlio sinistro di un nodo che è a sua volta figlio sinistro



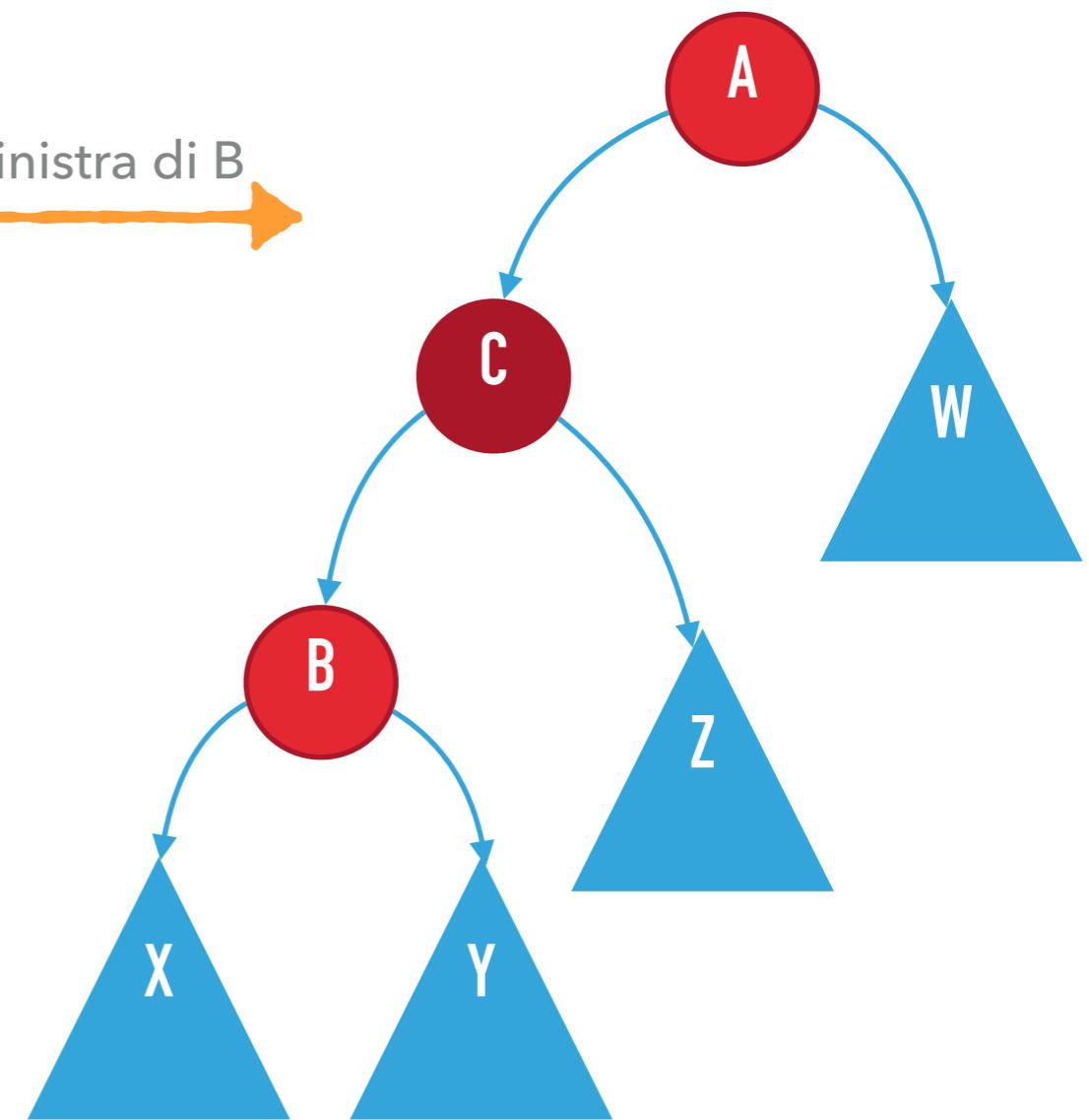
CASO ZIG ZAG

Il nodo da muovere è figlio destro di un nodo che è figlio sinistro



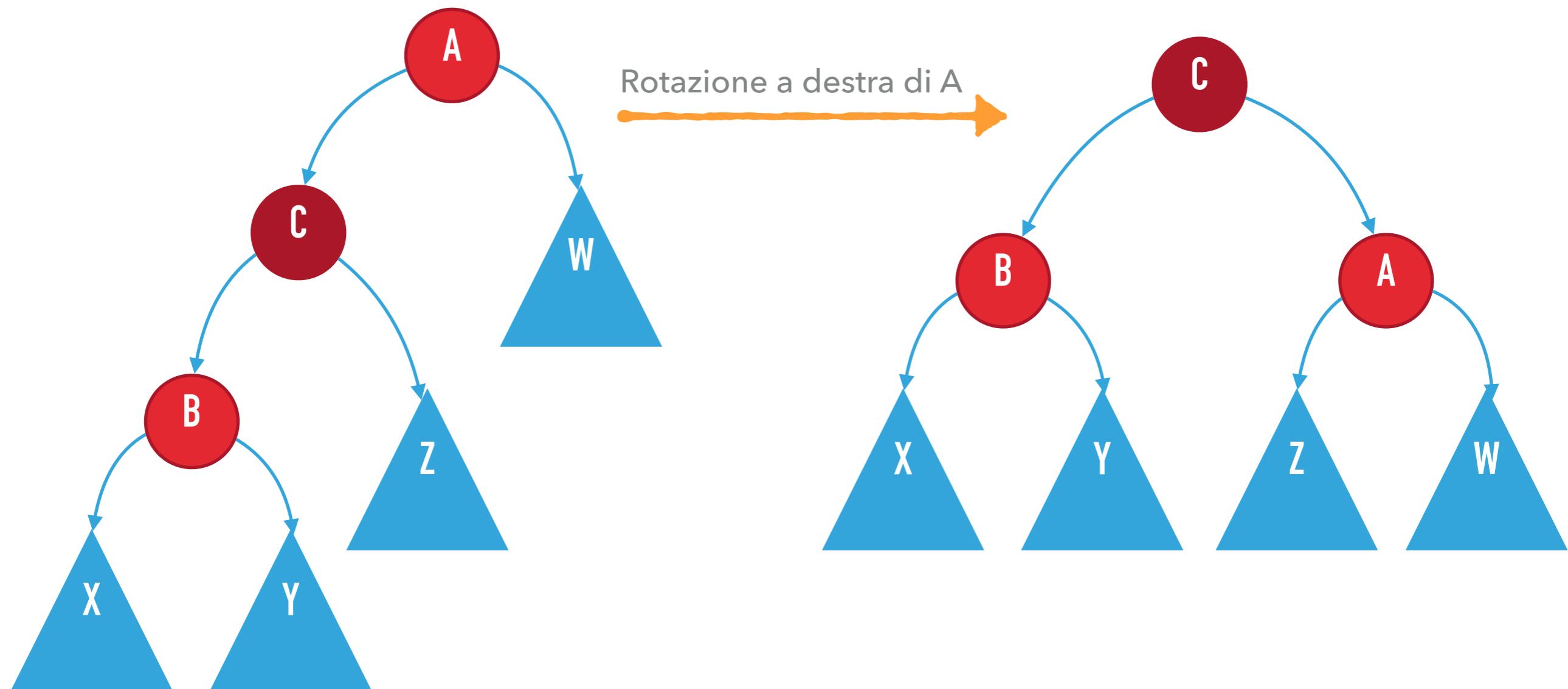
Rotazione a sinistra di B

An orange arrow points from the text "Rotazione a sinistra di B" towards the diagram, indicating a left rotation of node B.



CASO ZIG ZAG

Il nodo da muovere è figlio destro di un nodo che è figlio sinistro



ALBERI SPLAY

- ▶ Applicando uno tra questi tre casi (o i loro simmetrici) ad ogni passo possiamo spostare un nodo fino alla radice
- ▶ La struttura di base è:
 - ▶ Il nodo è alla radice? 
 - ▶ Sì: allora abbiamo completato
 - ▶ No: vedi in quali dei tre casi si è. Applicare le rotazioni.

ALBERI SPLAY

- ▶ Non vedremo una analisi accurata del tempo di calcolo delle operazioni di ricerca
- ▶ Esiste però il seguente teorema (**Balance Theorem**)
- ▶ Data una sequenza di m operazioni di ricerca in un albero splay di n elementi, il costo di effettuare la sequenza di operazioni è $O(m \log n + n \log n)$
- ▶ Questo significa che se facciamo almeno n operazioni il costo medio di ogni operazione è logaritmico

ALBERI AVL
ALBERI ROSSO NERI
DYNAMIC SELECT

ALGORITMI E STRUTTURE DATI

ALBERI AVL

ALBERI AVL

COSA SONO GLI ALBERI AVL

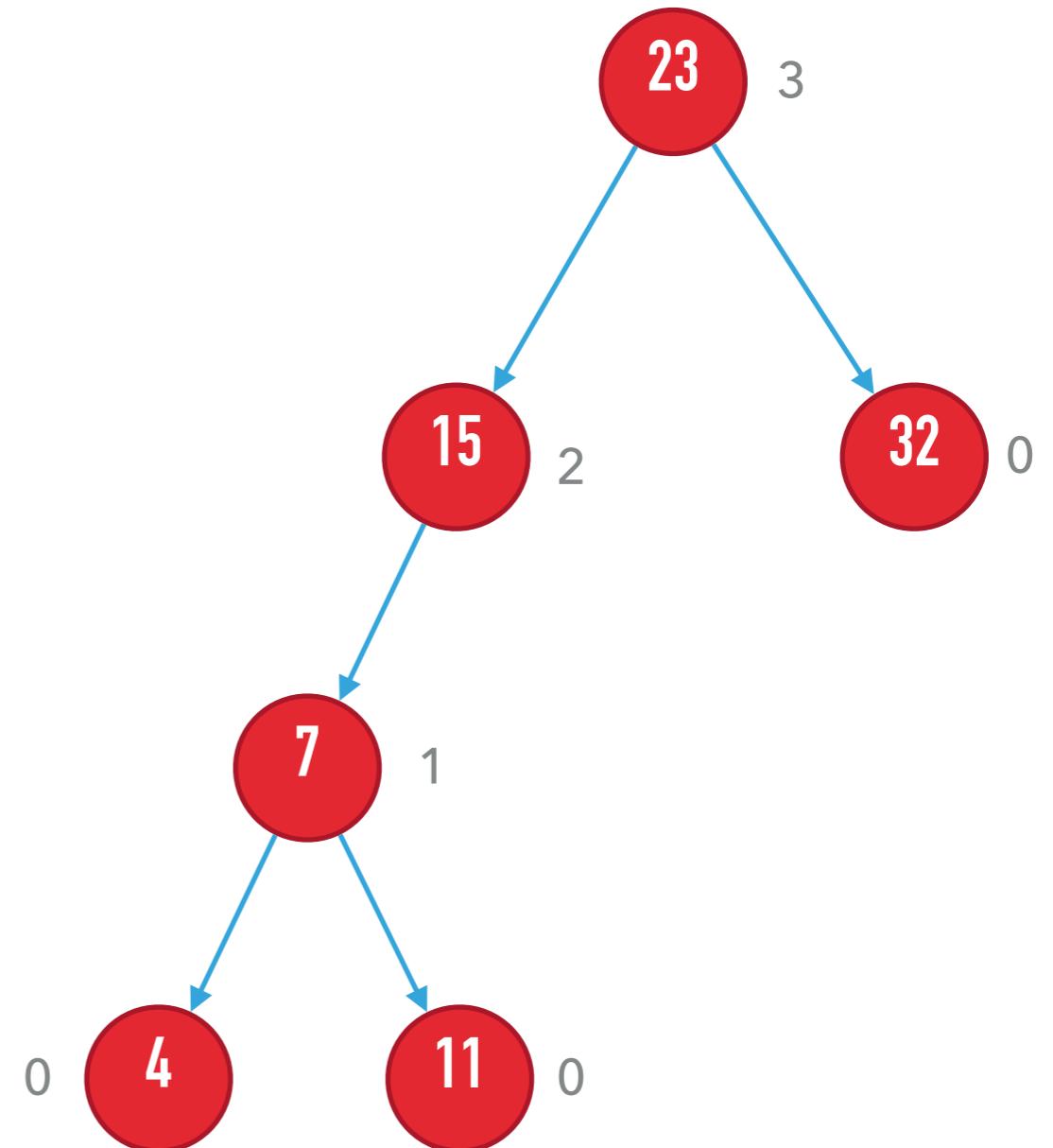
- ▶ AVL da Adelson-Velsky e Landis, cognomi dei due ideatori (anno 1962)
- ▶ Sono alberi binari di ricerca che sono sempre bilanciati
- ▶ Ove bilanciati non significa esattamente profondità $\log n$, ma $O(\log n)$
- ▶ Nel caso degli alberi AVL la costante del $\log n$ è bassa

DIFFERENZA DAGLI ALBERI SPLAY

- ▶ A differenza degli alberi splay, gli alberi AVL assicurano il bilanciamento, quindi le operazioni di ricerca richiedono sempre tempo $O(\log n)$
- ▶ Le operazioni di inserimento e rimozione modificano l'albero per mantenerlo bilanciato
- ▶ Le operazioni di ricerca non modificano l'albero (al contrario degli alberi splay)

ALTEZZA DI UN NODO

Definiamo come altezza di un nodo
il percorso più lungo da quel nodo a
una foglia

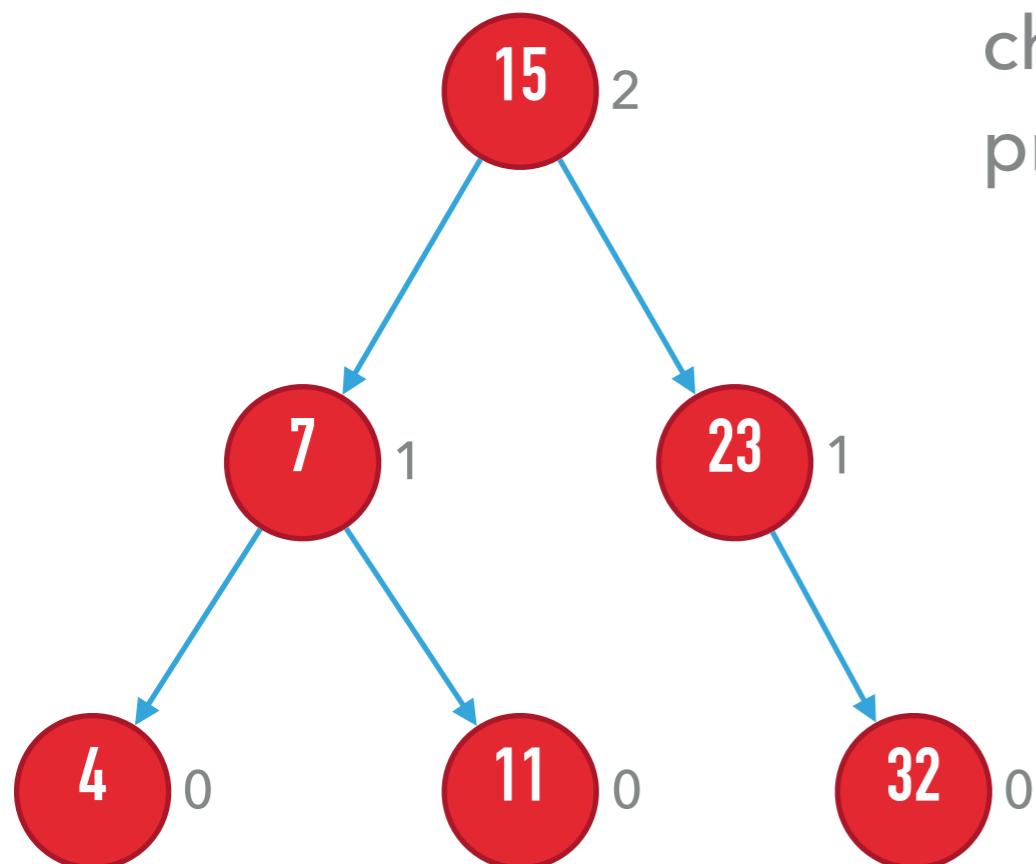


Possiamo facilmente calcolarla per ogni nodo come
 $1 + \max\{\text{altezza del figlio sinistro}, \text{altezza del figlio destro}\}$

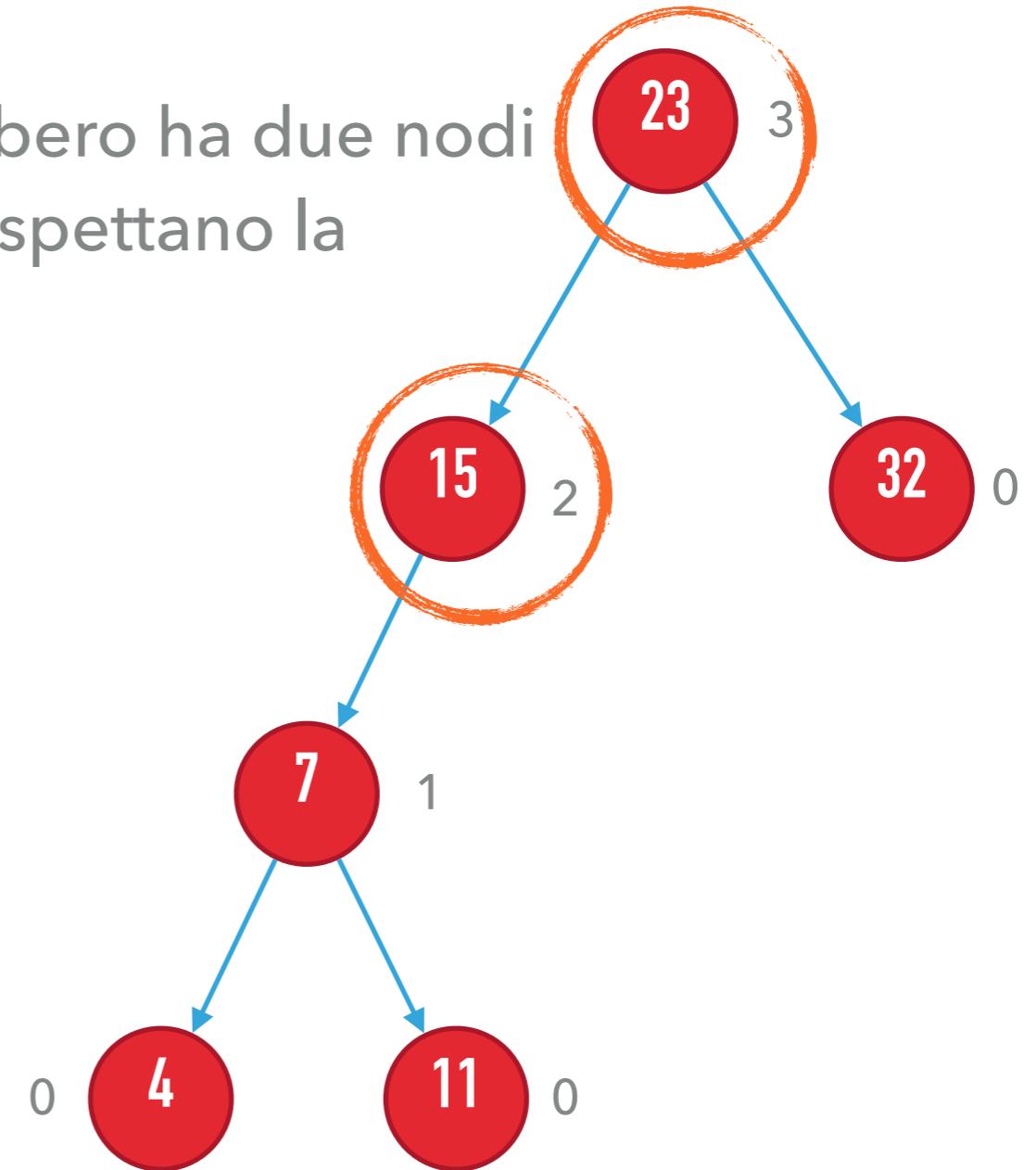
PROPRIETÀ DEGLI ALBERI AVL

- ▶ Gli alberi AVL hanno la proprietà che la differenza in altezza del figlio destro e del figlio sinistro di ogni nodo è al più 1 (in valore assoluto)
- ▶ In caso di figlio assente si assume altezza di -1

DIFFERENZA IN ALTEZZA

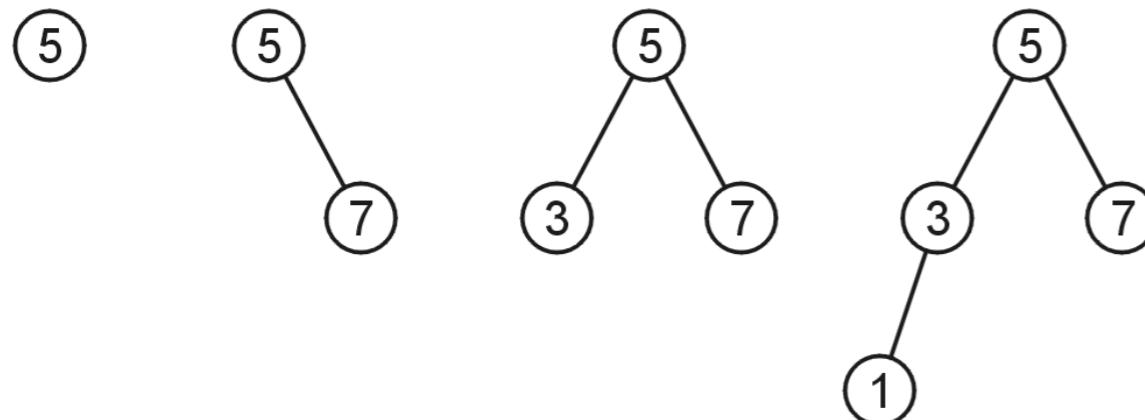


Questo albero ha due nodi
che non rispettano la
proprietà

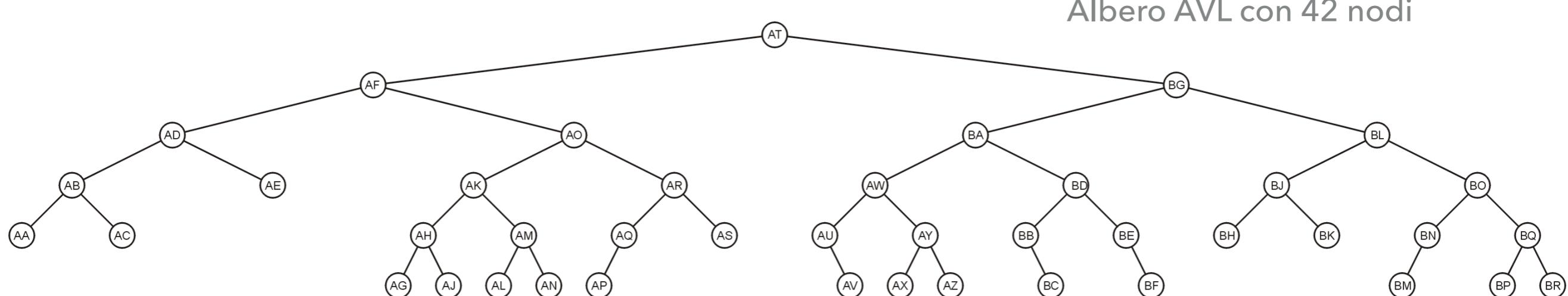


Questo albero rispetta la proprietà
(tutte le differenze sono zero o uno)

ESEMPI DI ALBERI AVL



Alberi AVL con 1, 2, 3 e 4 nodi



Albero AVL con 42 nodi

Sembra abbastanza "piatto", con un'altezza limitata.

Possiamo dimostrare che un albero AVL ha sempre un'altezza $O(\log n)$ quando contiene n nodi?

QUANTI NODI CONTIENE UN ALBERO AVL

- ▶ Invertiamo il problema di chiedere l'altezza dell'albero chiedendoci invece quale sia il numero minimo di nodi che un albero AVL di altezza h può contenere
- ▶ Indichiamo con N_h questo numero di nodi
- ▶ Dalla slide precedente otteniamo $N_0 = 1, N_1 = 2, N_2 = 4$

QUANTI NODI CONTIENE UN ALBERO AVL

- ▶ Quale è il caso peggiore per un albero AVL?
- ▶ Quando c'è uno sbilanciamento di 1 tra i due figli.
- ▶ Quindi possiamo definire N_h come una ricorrenza:
$$N_h = 1 + N_{h-1} + N_{h-2}$$
- ▶ Come possiamo ottenere un bound per il valore di N_h ?
Vediamo due modi

QUANTI NODI CONTIENE UN ALBERO AVL

- ▶ Dato che $N_{h-1} \geq N_{h-2}$ possiamo riscrivere:

$$N_h = 1 + N_{h-1} + N_{h-2} \geq 1 + 2N_{h-2} \geq 2N_{h-2}$$

- ▶ Espandendo la ricorrenza otteniamo che per arrivare al caso base (valore di h costante) dobbiamo espandere almeno $h/2$ volte:

$$N_h \geq 2N_{h-2} \geq 4N_{h-4} \geq 8N_{h-6} \geq \dots \geq 2^i N_{h-2i}$$

- ▶ Otteniamo quindi $N_h \geq 2^{h/2}$

QUANTI NODI CONTIENE UN ALBERO AVL

- ▶ Possiamo ora prendere il logaritmo in base 2 da entrambi i lati:
$$\log_2 N_h \geq \log_2 2^{h/2} = h/2$$
- ▶ Otteniamo quindi un bound sull'altezza: $h \leq 2 \log_2 N_h$
- ▶ Questo significa che per ogni numero n di nodi, anche se disposti nel caso peggiore saranno in un albero di altezza non più di $2 \log_2 n = O(\log n)$

QUANTI NODI CONTIENE UN ALBERO AVL

► Vediamo ora un metodo un poco più raffinato

$N_h = 1 + N_{h-1} + N_{h-2}$ assomiglia molto alla definizione dei numeri di Fibonacci:

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, \dots, F_k = F_{k-1} + F_{k-2}$$

Mostriamo per induzione che $N_h = F_{h+3} - 1$

Casi base: $N_0 = 1 = F_3 - 1$ e $N_1 = 2 = F_4 - 1$

QUANTI NODI CONTIENE UN ALBERO AVL

Passo induttivo:

$$\begin{aligned}N_h &= N_{h-1} + N_{h-2} + 1 \\&= F_{h+2} - 1 + F_{h+1} - 1 + 1 \\&= F_{h+2} + F_{h+1} - 1 \\&= F_{h+3} - 1\end{aligned}$$

Quindi possiamo usare i numeri di Fibonacci per ottenere un bound sull'altezza dell'albero.

QUANTI NODI CONTIENE UN ALBERO AVL

Sappiamo che

$$F_k = \frac{\phi^k - \psi^k}{\sqrt{5}} \text{ con } \phi = \frac{1 + \sqrt{5}}{2} \text{ e } \psi = \frac{1 - \sqrt{5}}{2}$$

In generale $F_k \geq \frac{\phi^k - 1}{\sqrt{5}}$ e quindi usando $N_h = F_{h+3} - 1$:

$$N_h \geq \frac{\phi^{h+3} - 1}{\sqrt{5}} - 1, \text{ ovvero } N_h \geq \frac{\phi^{h+3}}{\sqrt{5}} - \frac{1 + \sqrt{5}}{\sqrt{5}} \geq \frac{\phi^{h+3}}{\sqrt{5}} - 1.45$$

QUANTI NODI CONTIENE UN ALBERO AVL

Riordinando: $\sqrt{5}(N_h + 1.45) \geq \phi^{h+3}$

Prendiamo il logaritmo in base ϕ : $\log_\phi(\sqrt{5}(N_h + 1.45)) \geq h + 3$

Ovvero: $h \leq \log_\phi(\sqrt{5}(N_h + 1.45)) - 3$

Cambio di base del logaritmo: $h \leq \frac{\log_2(\sqrt{5}(N_h + 1.45))}{\log_2 \phi} - 3$

Riscriviamo come $h \leq \frac{1}{\log_2 \phi} \log(N_h + 2) + c$

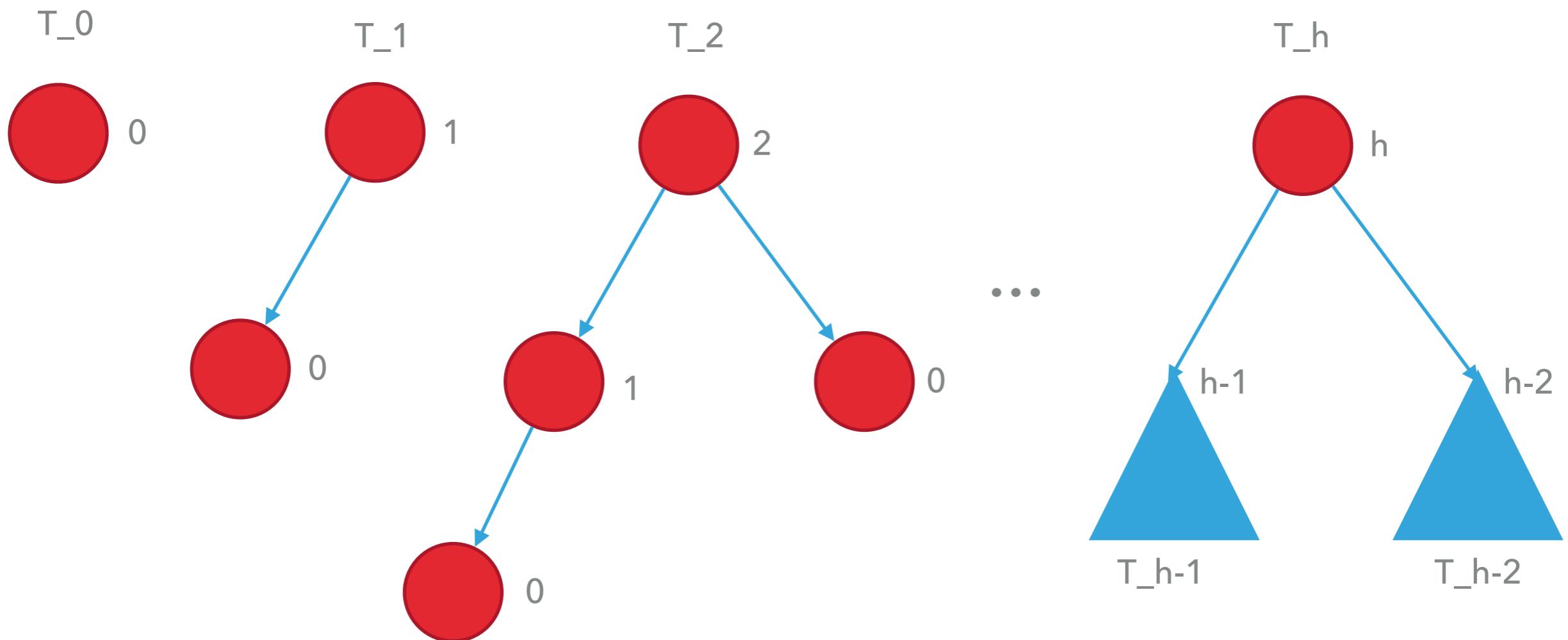
QUANTI NODI CONTIENE UN ALBERO AVL

Sapendo che $\frac{1}{\log_2 \phi} \leq 1.441$ si ha $h \leq 1.441 \log(N_h + 2) + c$

Questo significa che un albero con n nodi ha altezza $O(\log n)$

Possiamo anche dire di più: dato che libero binario più piccolo che contiene n nodi ha profondità $\lceil \log_2 n \rceil$, non siamo molto distanti da quel valore dato che il fattore moltiplicativo è circa 1.44.

ALBERI DI FIBONACCI



Sono gli alberi AVL con meno nodi.

COME GARANTIRE IL BILANCIAMENTO

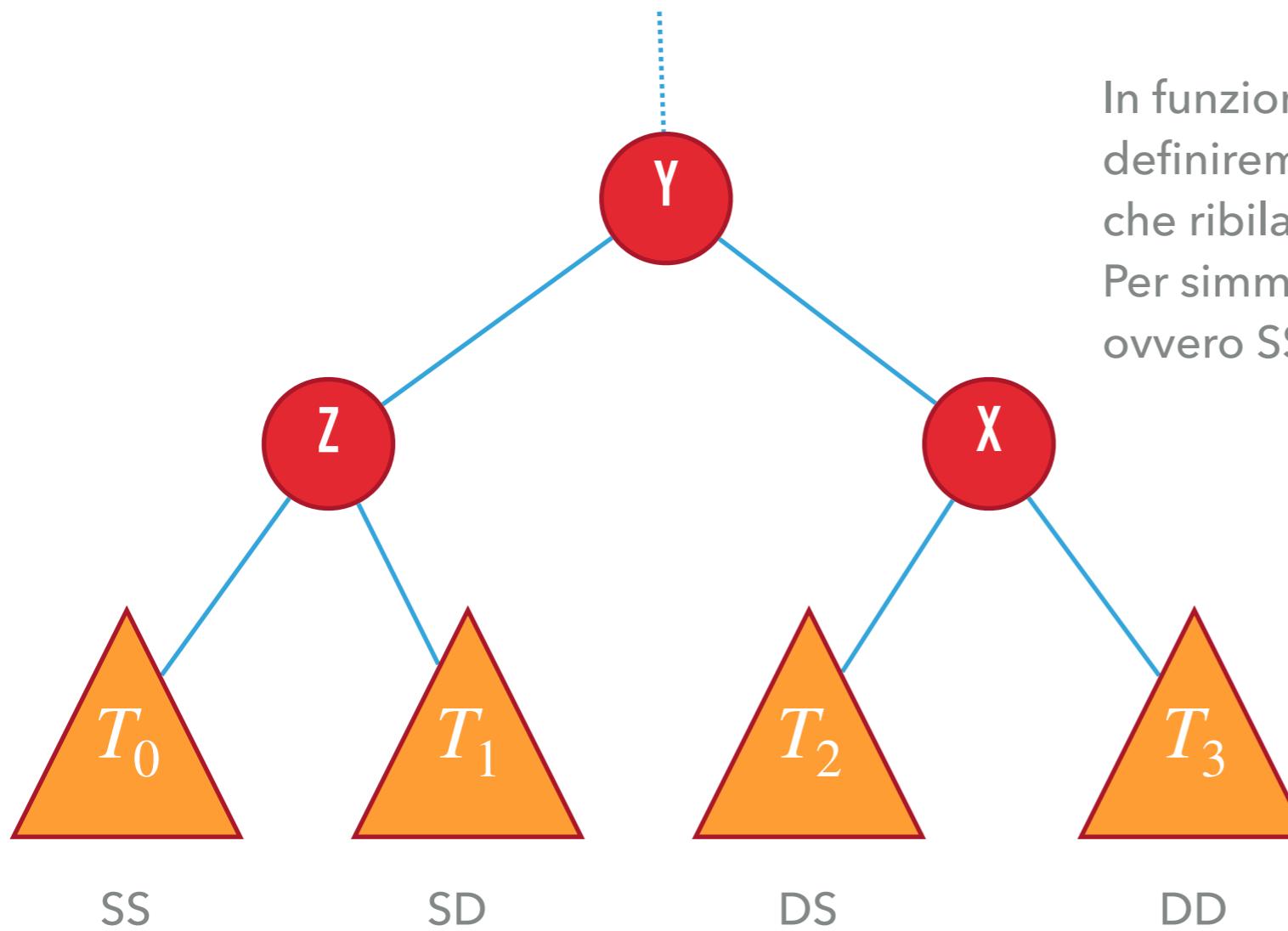
- ▶ Per garantire il bilanciamento dobbiamo memorizzare dell'informazione all'interno dei nodi.
- ▶ Due possibilità:
 - ▶ Altezza del nodo
 - ▶ Quale è lo sbilanciamento: { -1,0,1 }
 - ▶ Il valore rappresenta la differenza tra altezza del figlio destro e altezza del figlio sinistro

INSERIMENTO E CANCELLAZIONE

- ▶ L'inserimento può violare la proprietà AVL su più nodi nel percorso che va dalla radice al punto dove è stato inserito il nuovo nodo
- ▶ La cancellazione analogamente può violare la proprietà AVL nei nodi antenati del nodo cancellato
- ▶ Vediamo come ogni sbilanciamento può essere risolto tramite una sequenza di rotazioni

RIBILANCIAMENTO

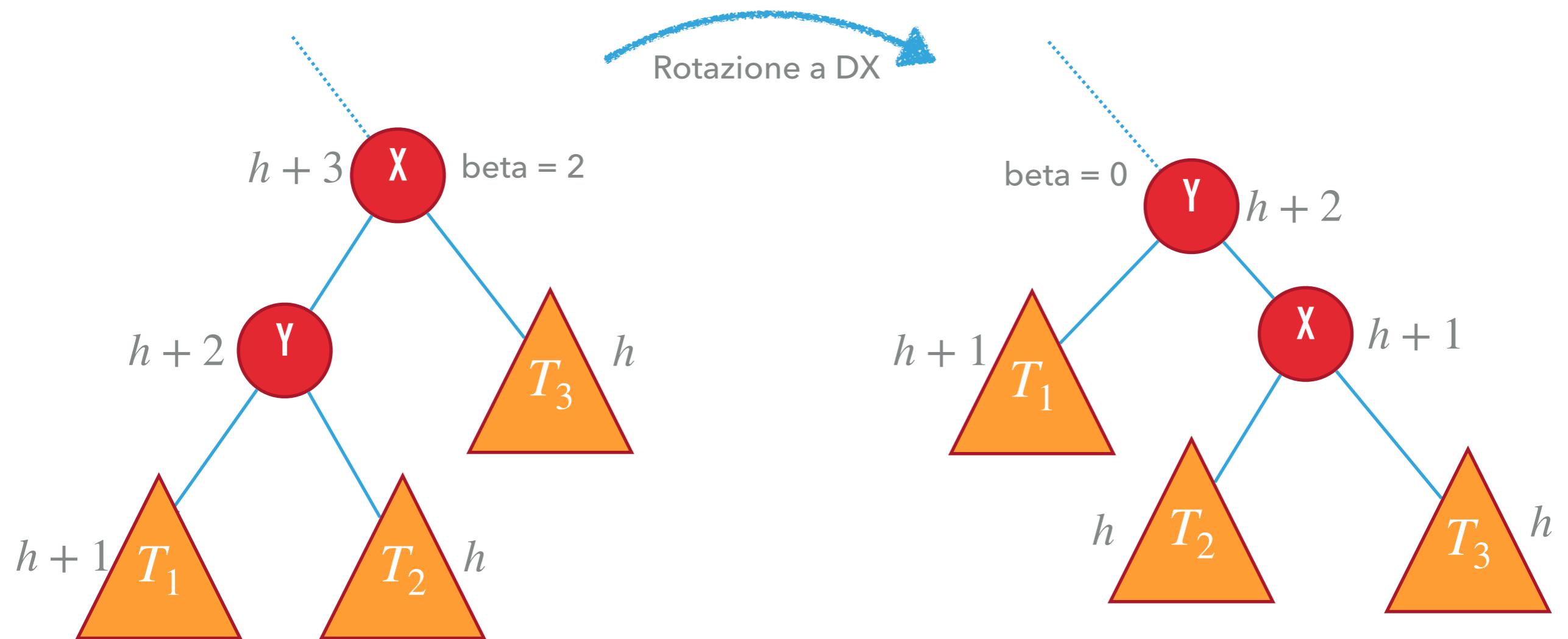
Se emerge uno sbilanciamento dopo un inserimento o una cancellazione di un nodo, allora il fattore di sbilanciamento sarà 2 o -2, e causato da un nodo in più (o in meno) in uno dei 4 sottoalberi T_0 T_1 T_2 T_3 .



In funzione di dove emerge lo sbilanciamento, definiremo una sequenza di rotazioni opportuna che ribilanci l'albero.
Per simmetria, tratteremo solo due dei 4 casi, ovvero SS e SD.

RIBILANCIAMENTO - CASO SS

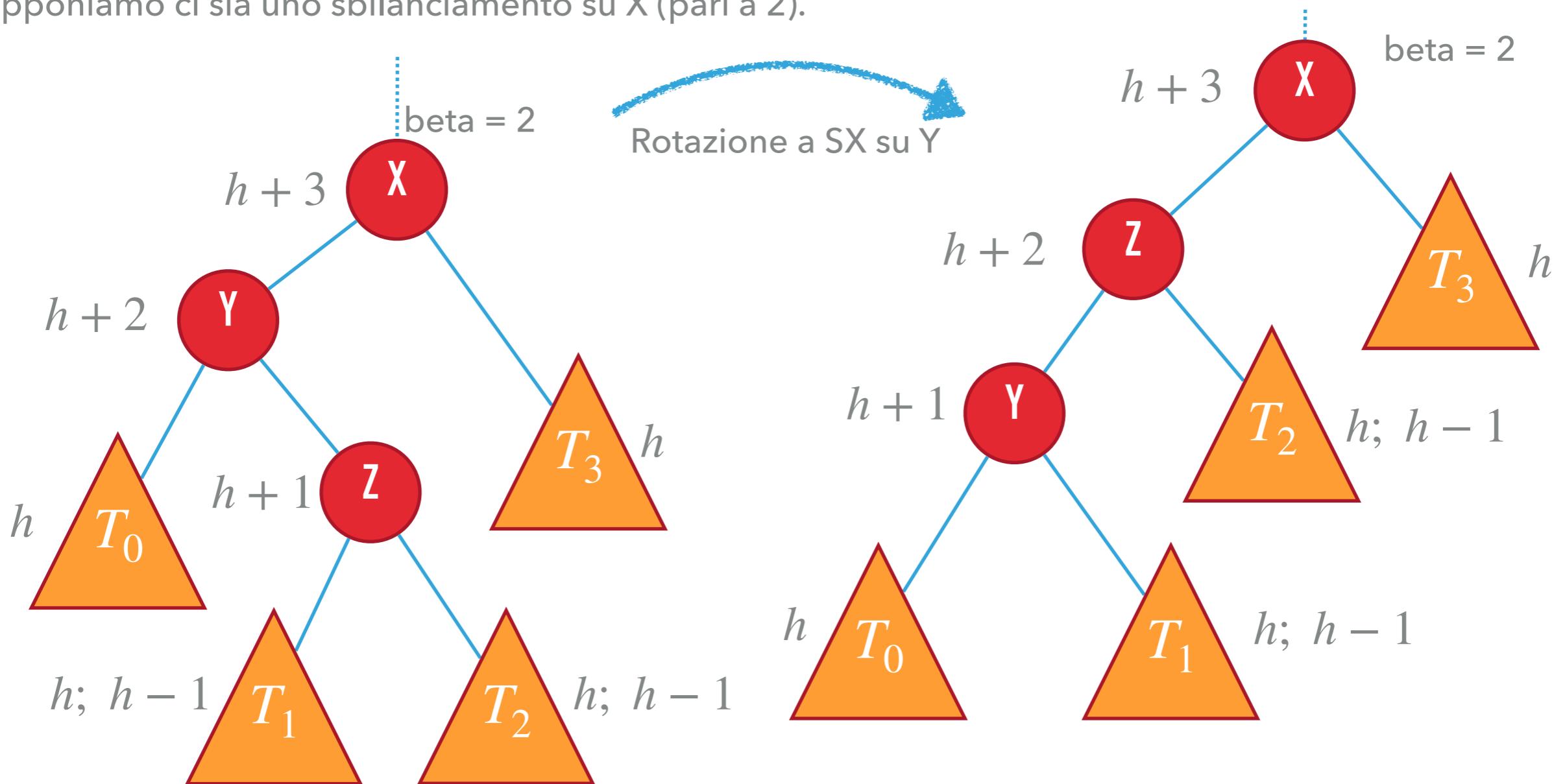
Supponiamo ci sia uno sbilanciamento su X (pari a 2) per la differenza tra le altezze di T1 e T3.



L'altezza del sottoalbero precedentemente radicato in X decresce di uno

RIBILANCIAMENTO - CASO SD

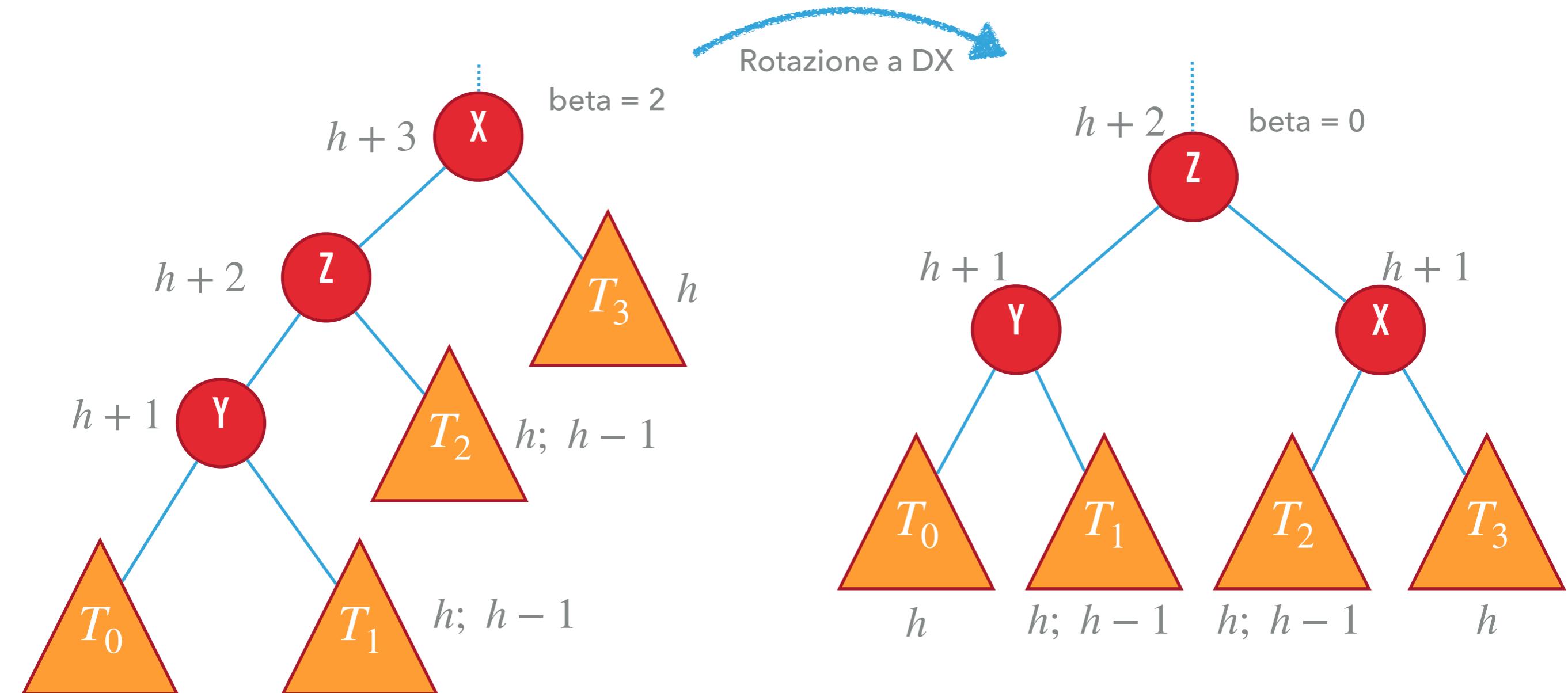
Supponiamo ci sia uno sbilanciamento su X (pari a 2).



Solo uno tra T_1 e T_2 ha altezza h (causa inserimento)

La prima rotazione non risolve lo sbilanciamento

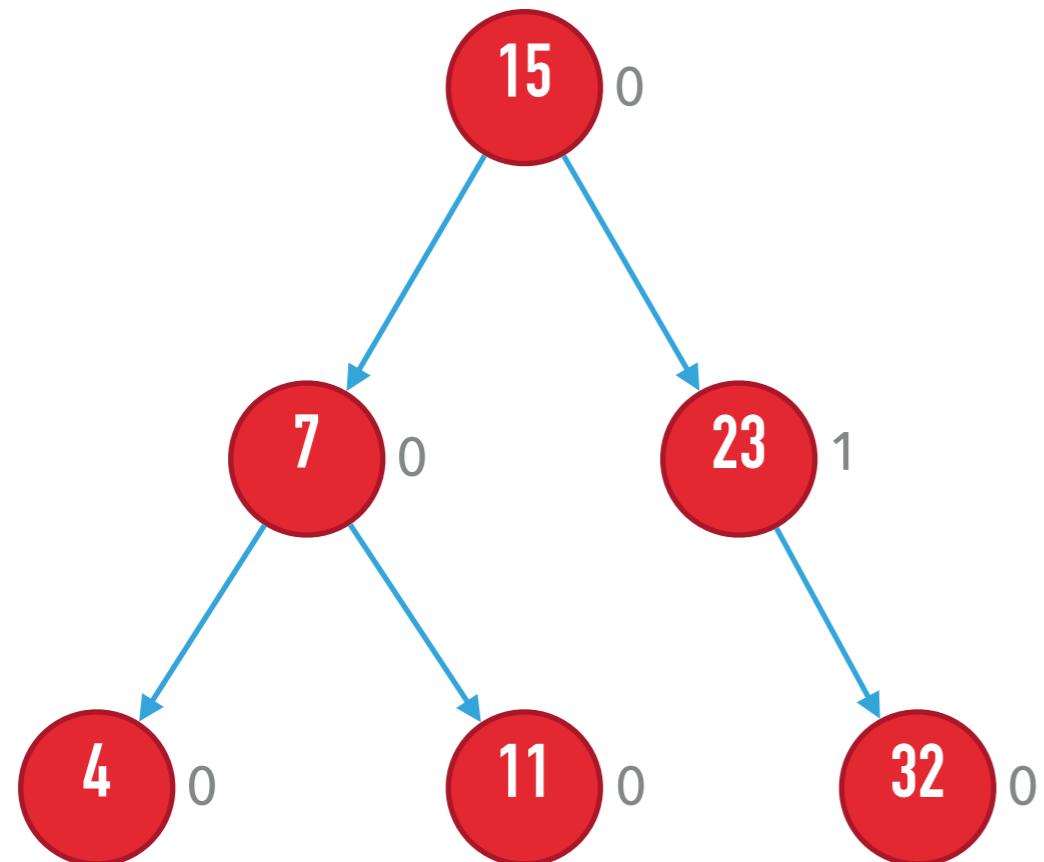
RIBILANCIAMENTO - CASO SD



L'altezza del sottoalbero precedentemente radicato in X decresce di uno

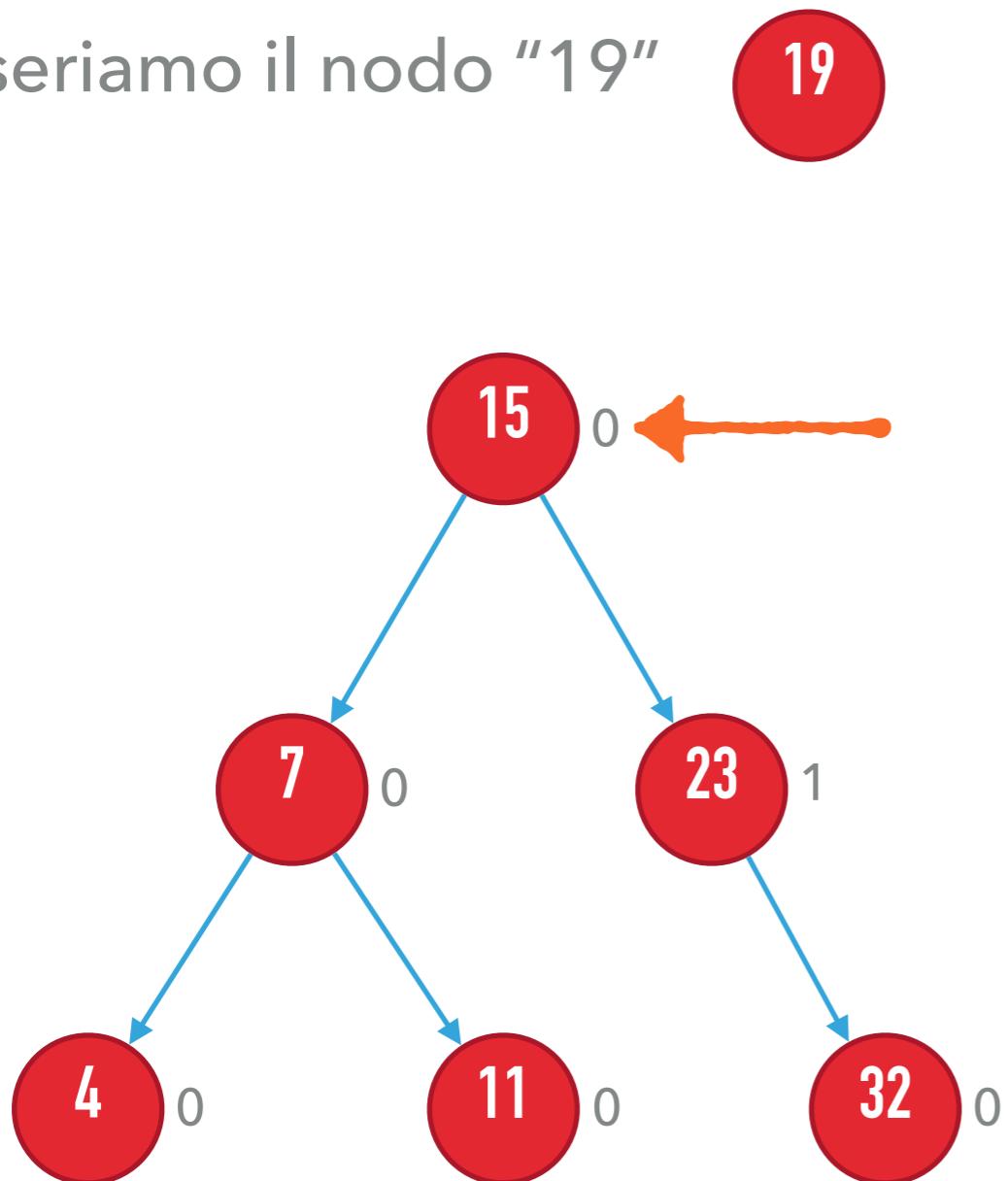
INSERIMENTO

Inseriamo il nodo "19"



INSERIMENTO

Inseriamo il nodo "19"



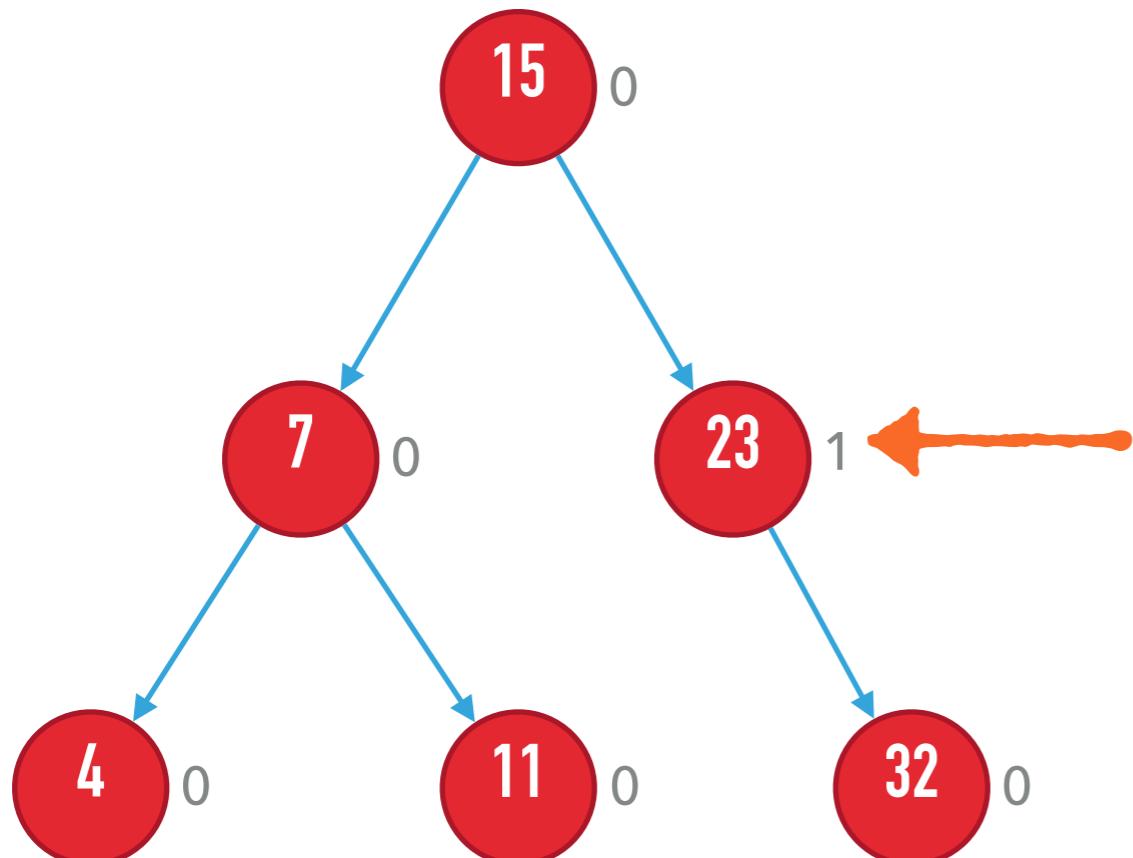
Stack dei nodi visitati

INSERIMENTO

Inseriamo il nodo "19"

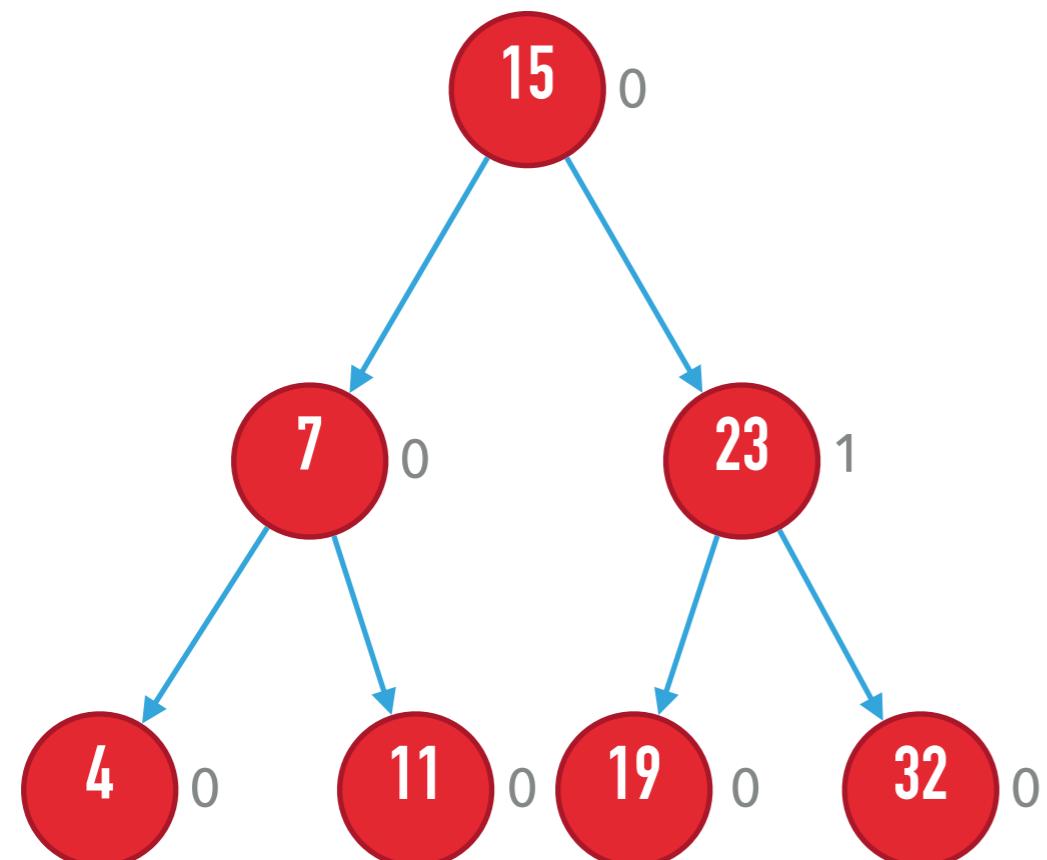


Stack dei nodi visitati



INSERIMENTO

Inseriamo il nodo "19"

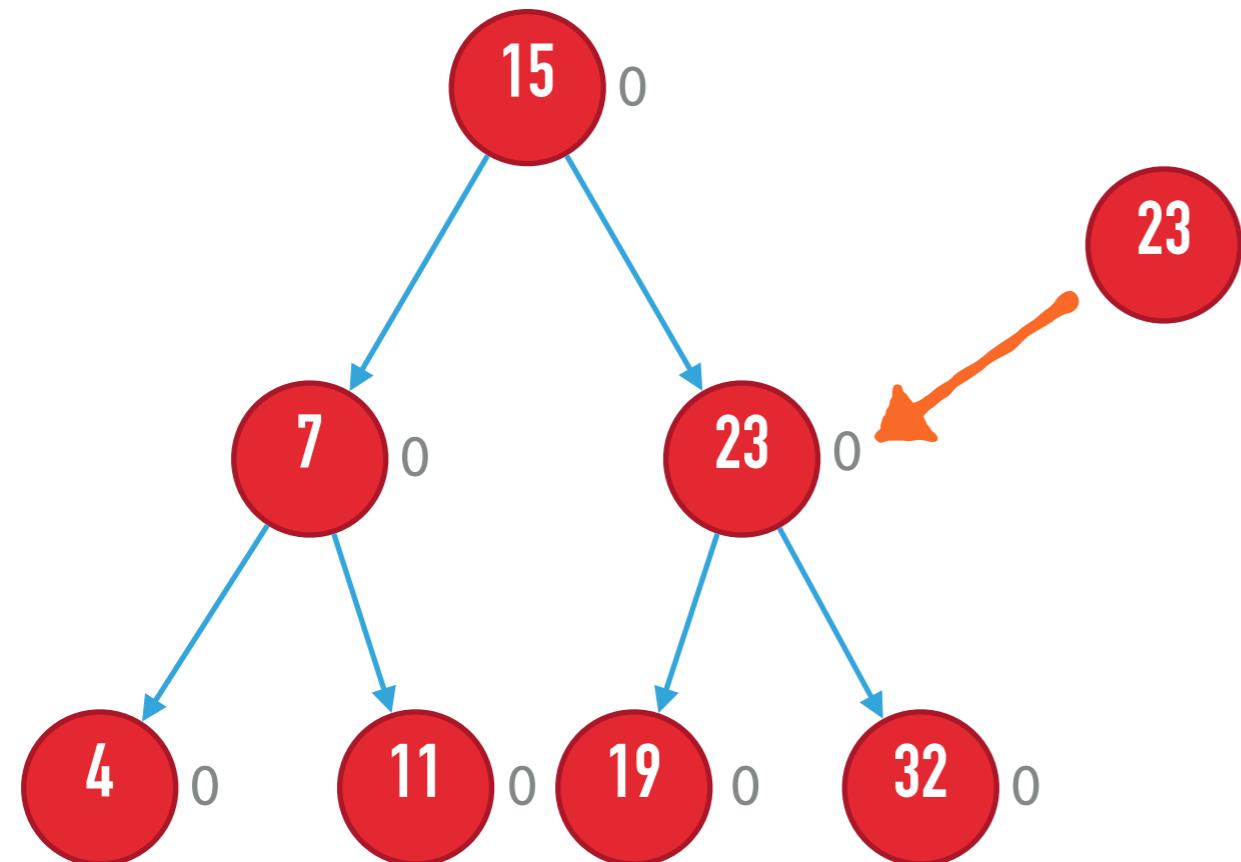


Stack dei nodi visitati

Abbiamo inserito il nodo, ora svuotiamo
lo stack e aggiorniamo il bilanciamento

INSERIMENTO

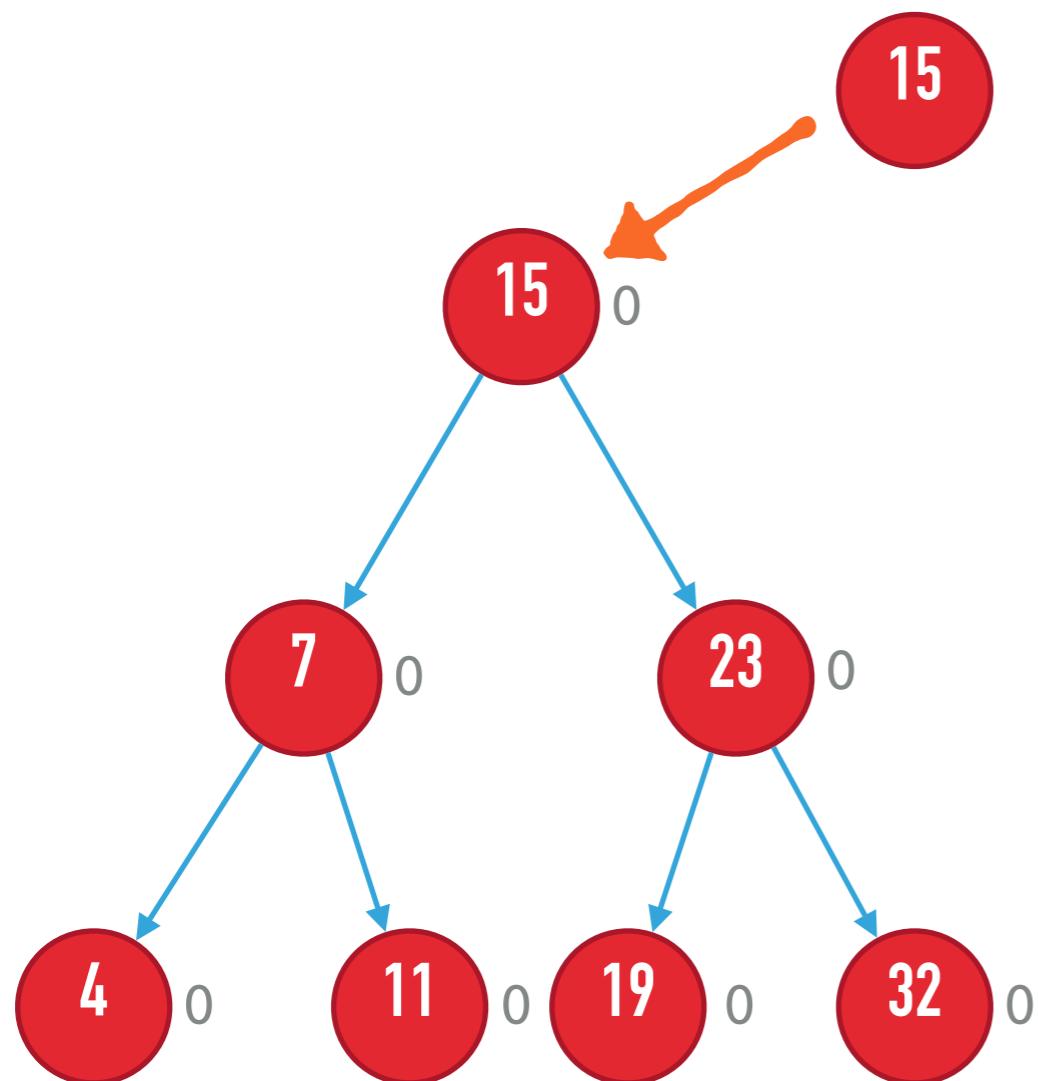
Inseriamo il nodo "19"



Stack dei nodi visitati

INSERIMENTO

Inseriamo il nodo "19"



Stack dei nodi visitati

INSERIMENTO

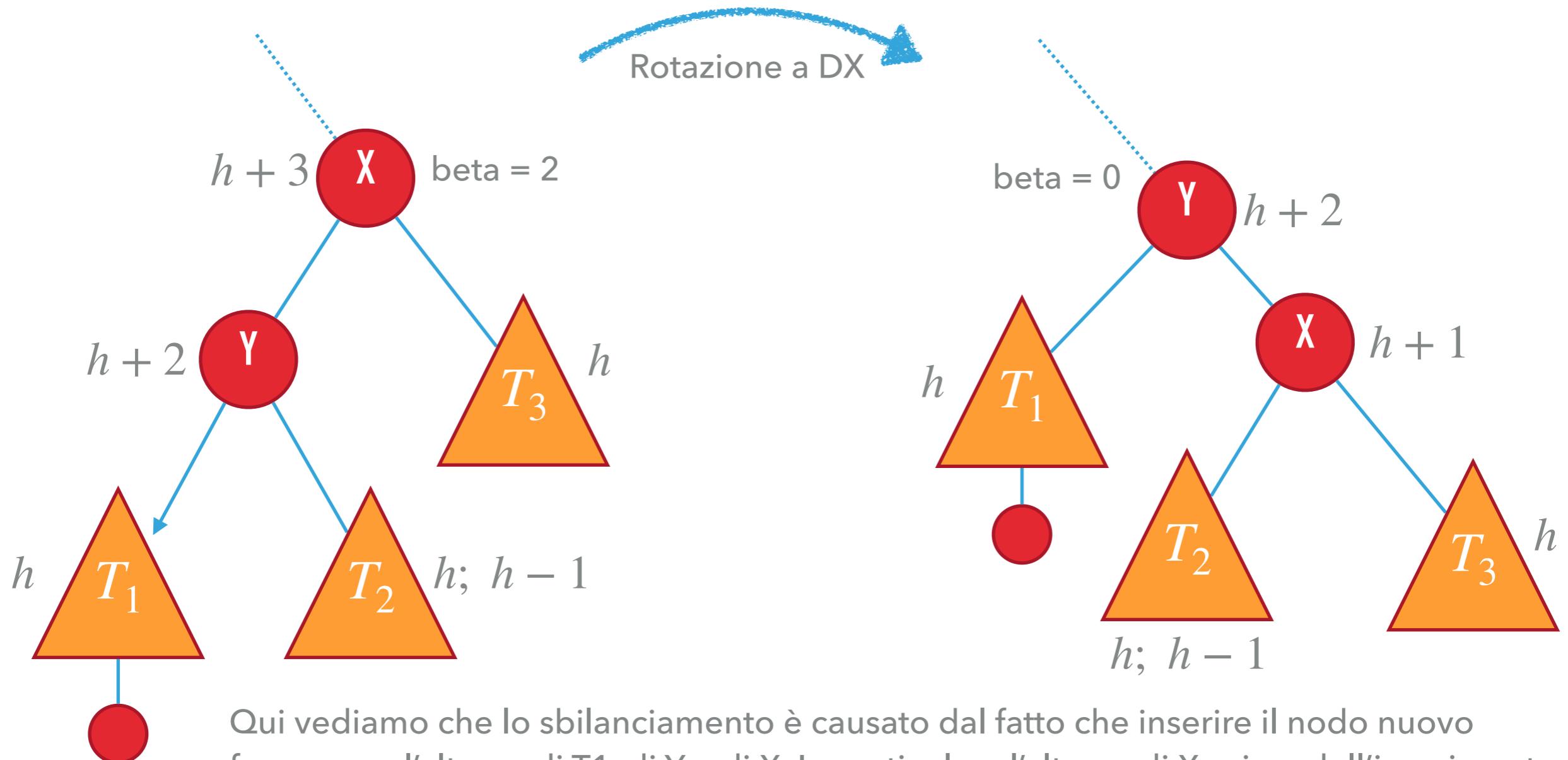
- ▶ Il caso migliore è quello in cui non otteniamo nessun valore +2 o -2 e aggiorniamo solamente il bilanciamento
- ▶ Notate come se c'è uno sbilanciamento, esso deve essere con un valore ± 2 , dato che l'aggiunta di un nodo modifica l'altezza di al più 1.
- ▶ Nel caso vi sia sbilanciamento cerchiamo il **primo** nodo – risalendo dal nodo appena inserito – che è sbilanciato - e applichiamo uno degli schemi di rotazione visti prima.

INSERIMENTO

- ▶ Ma come facciamo a provare che queste operazioni rendono un albero bilanciato?
- ▶ Non è direttamente intuitivo, lo proviamo per un caso (sinistra-sinistra), per gli altri casi è equivalente
- ▶ Associamo ad ogni nodo la sua altezza e vediamo come questa viene modificata dalle operazioni di rotazione

INSERIMENTO

Se effettuiamo la rotazione notiamo come la proprietà degli alberi AVL venga ripristinata



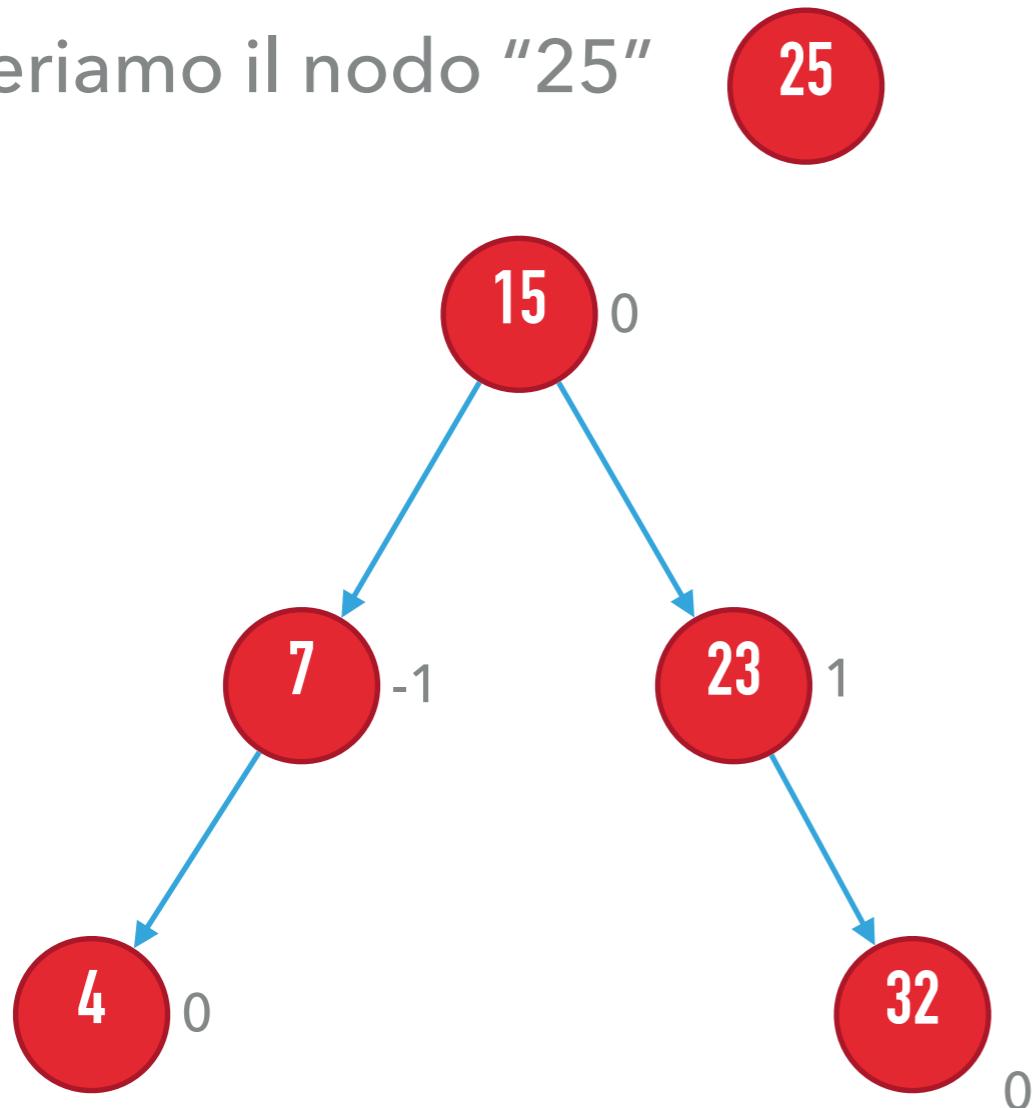
Qui vediamo che lo sbilanciamento è causato dal fatto che inserire il nodo nuovo fa crescere l'altezza di T_1 , di Y e di X . In particolare l'altezza di X prima dell'inserimento è $h+2$, e l'altezza del nodo Y , che sostituisce X , dopo la rotazione è di nuovo $h+2$. Non solo l'albero si ribilancia, ma anche ogni antenato di X torna ad essere bilanciato.

INSERIMENTO

- ▶ Abbiamo visto che per un caso le rotazioni ribilanciano il sottoalbero.
- ▶ Queste rotazioni possono creare problemi più in altro nell'albero?
- ▶ No, perché l'albero ottenuto dopo le rotazioni ha esattamente la stessa altezza dell'albero **prima** dell'inserimento
- ▶ Quindi se i nodi antenati erano bilanciati prima dell'inserimento lo rimangono anche dopo

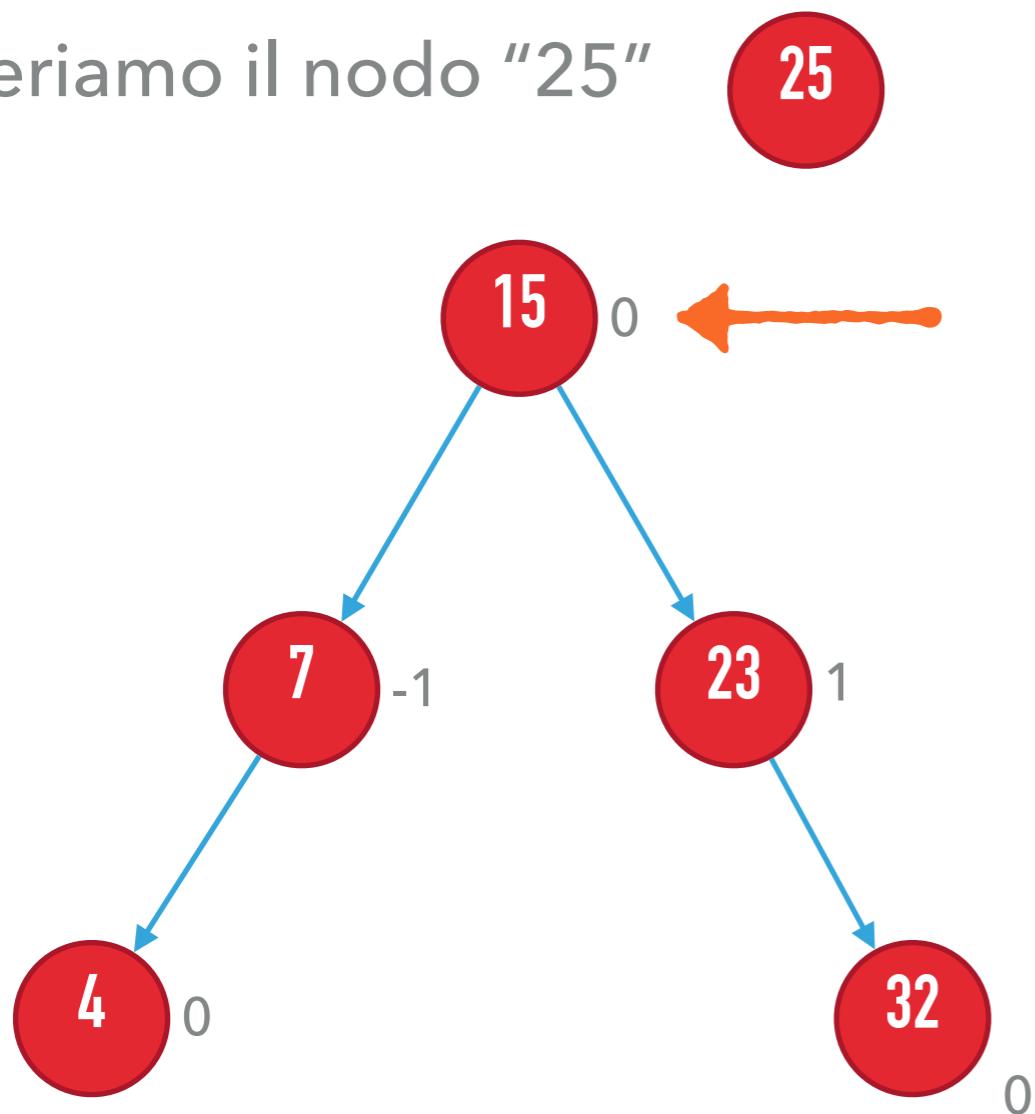
INSERIMENTO

Inseriamo il nodo "25"



INSERIMENTO

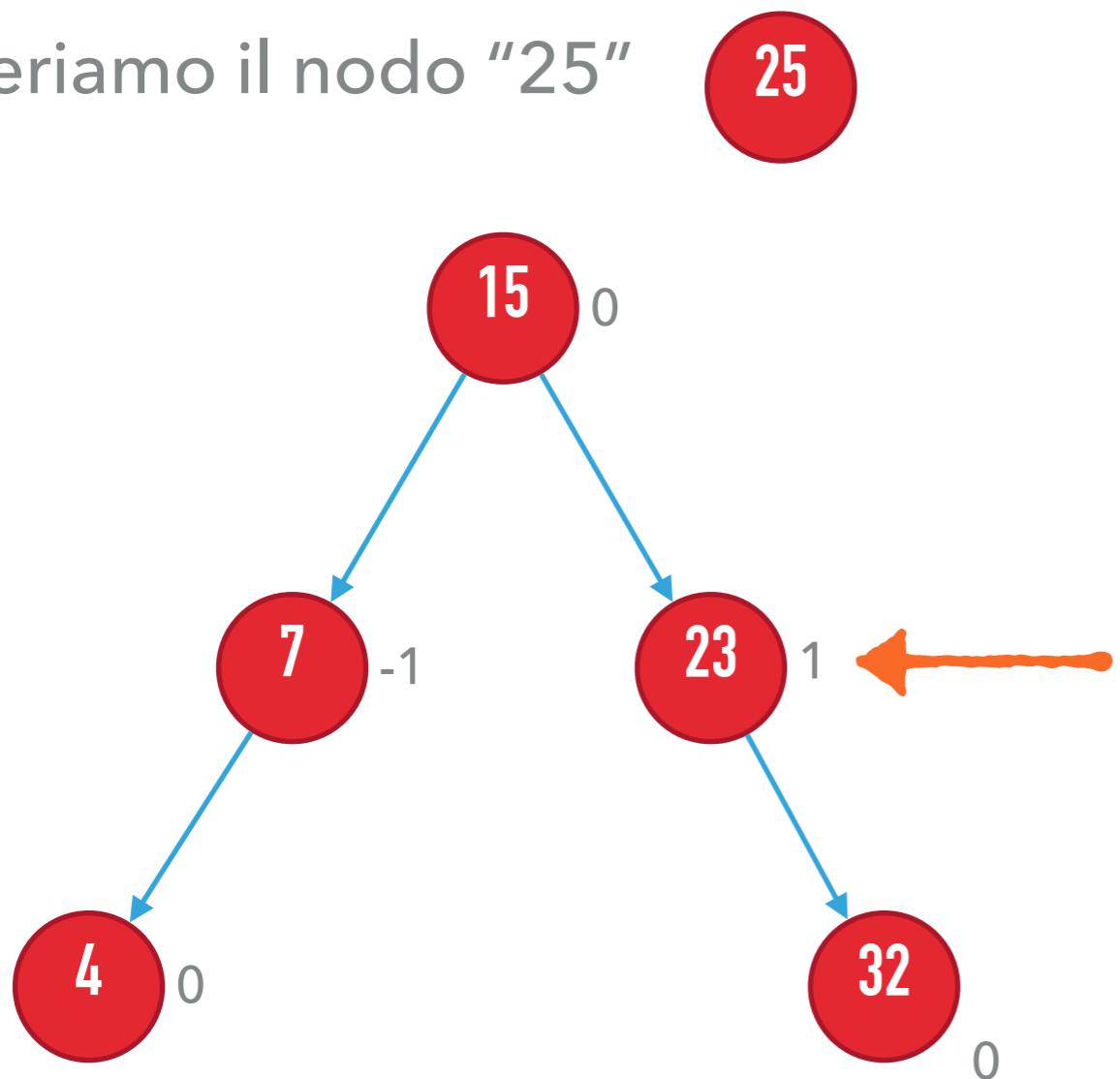
Inseriamo il nodo "25"



Stack dei nodi visitati

INSERIMENTO

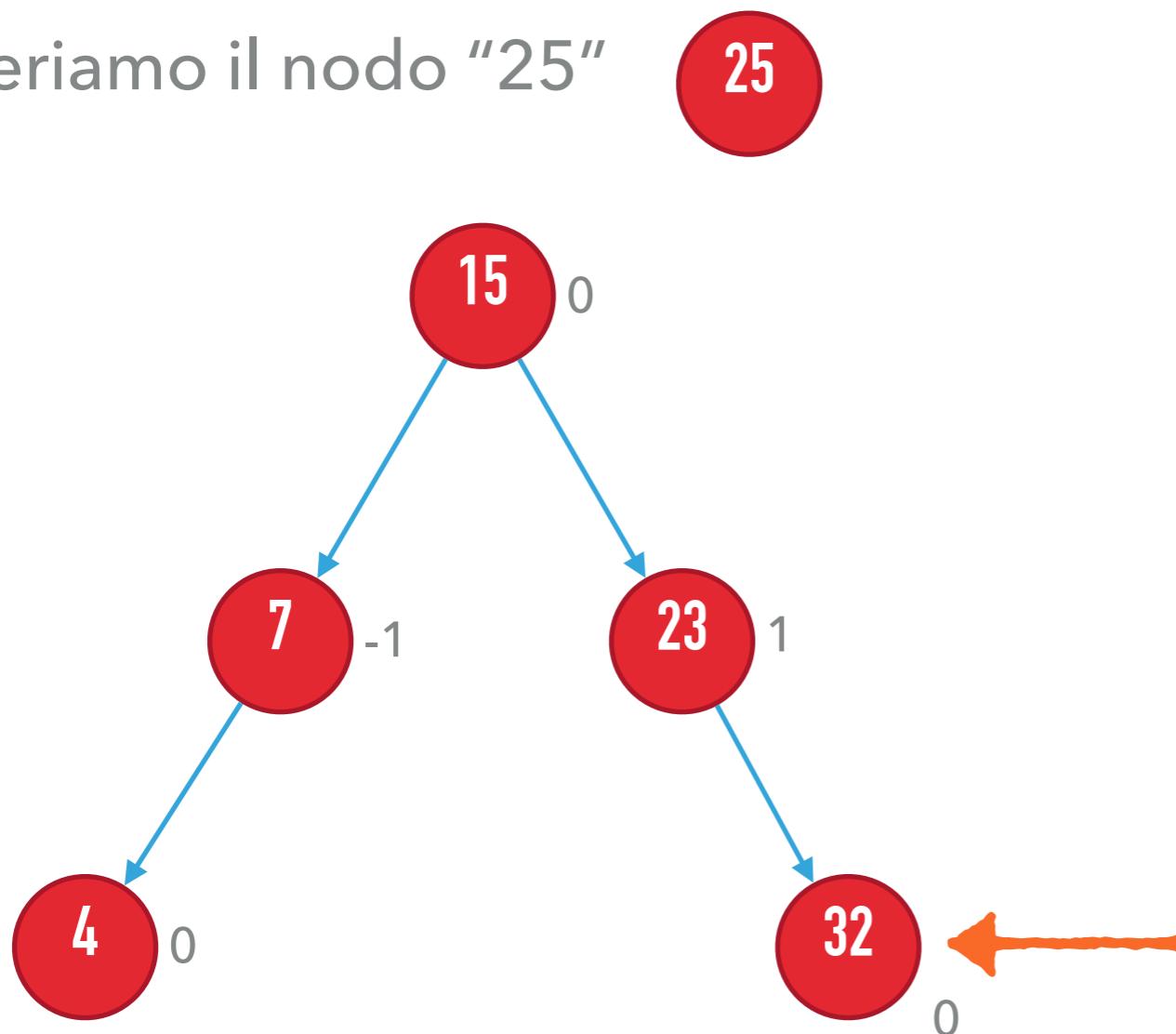
Inseriamo il nodo "25"



Stack dei nodi visitati

INSERIMENTO

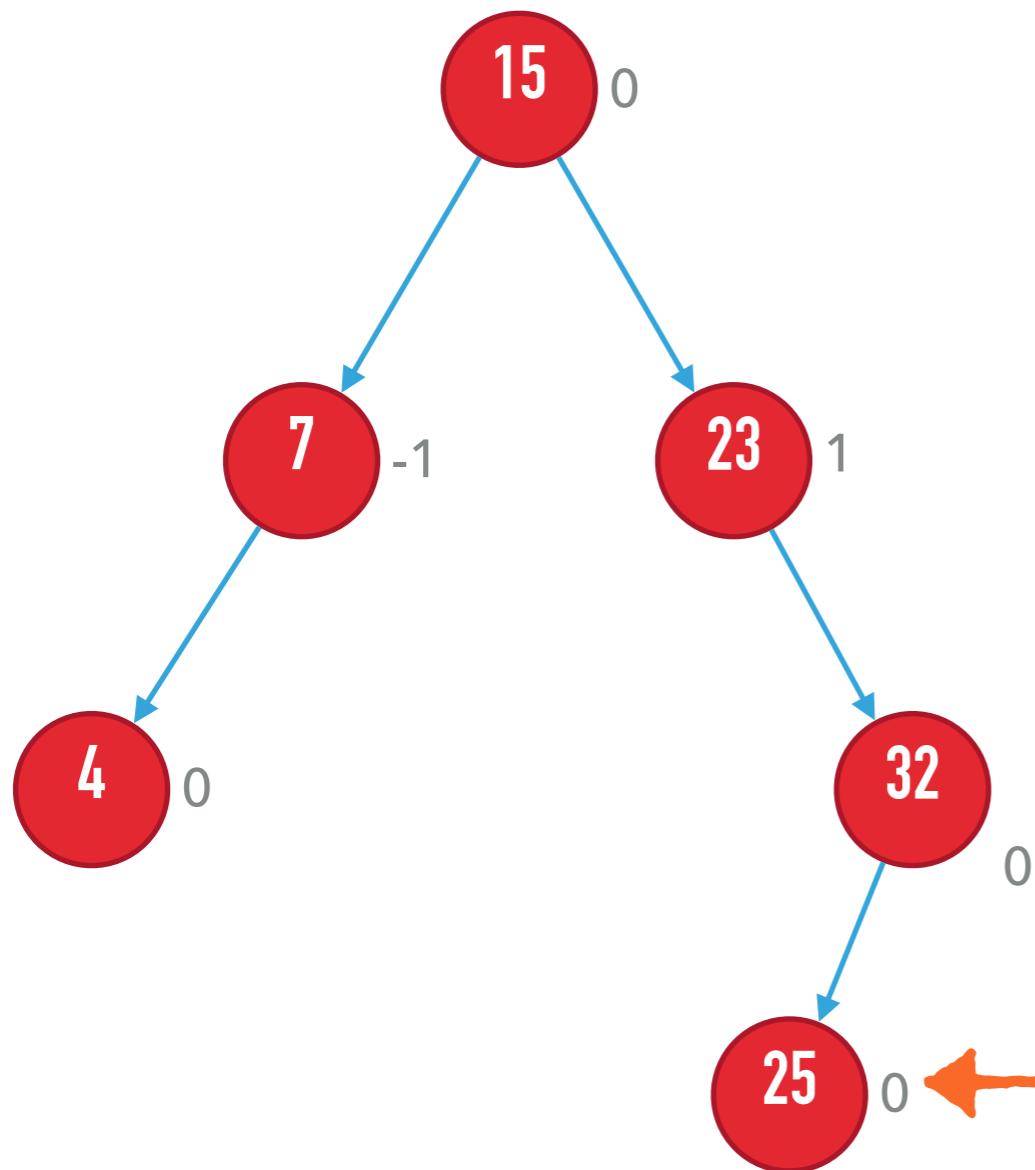
Inseriamo il nodo "25"



Stack dei nodi visitati

INSERIMENTO

Inseriamo il nodo "25"

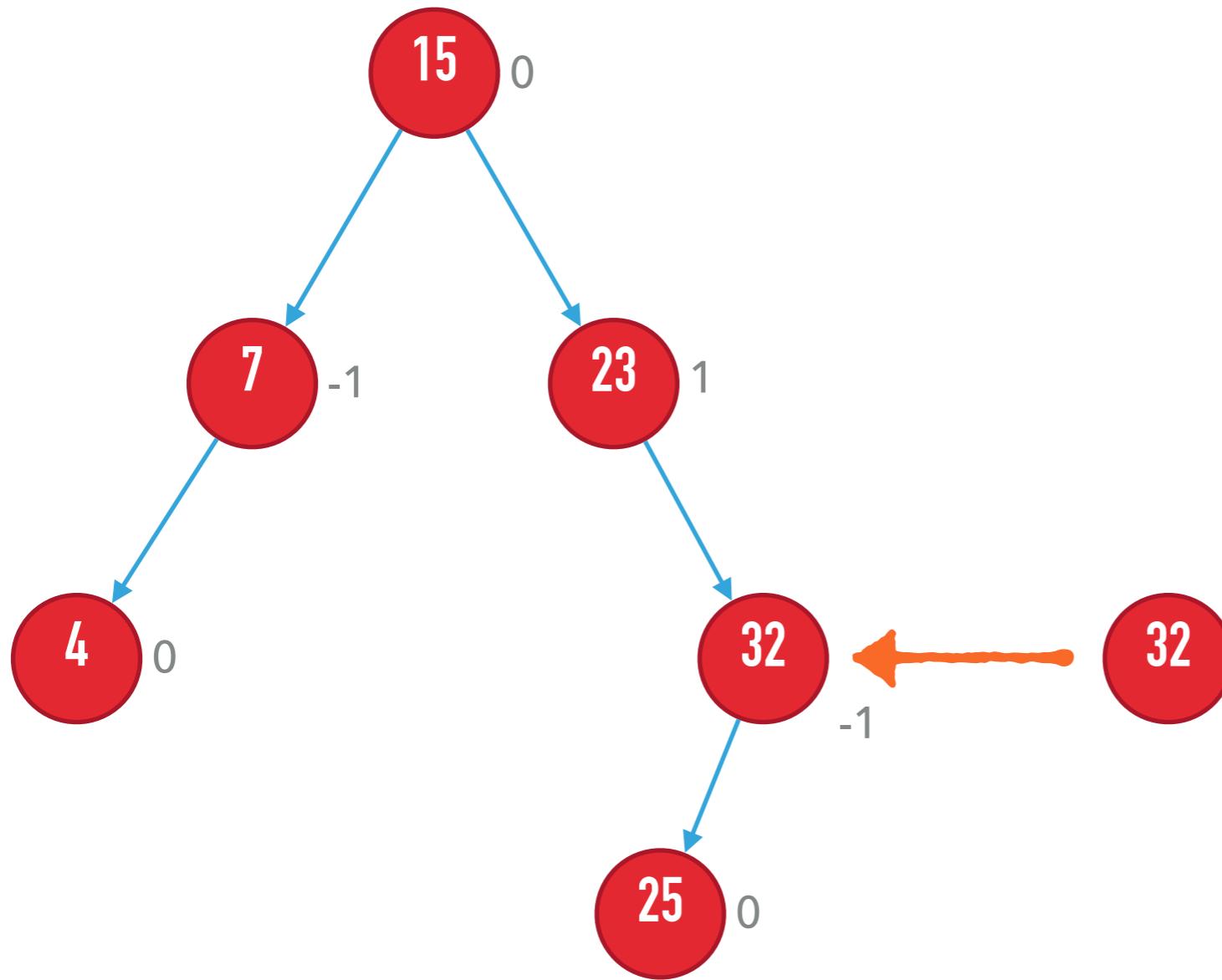


Stack dei nodi visitati

Inseriamo il nodo ed iniziamo ad aggiornare
lo sbilanciamento

INSERIMENTO

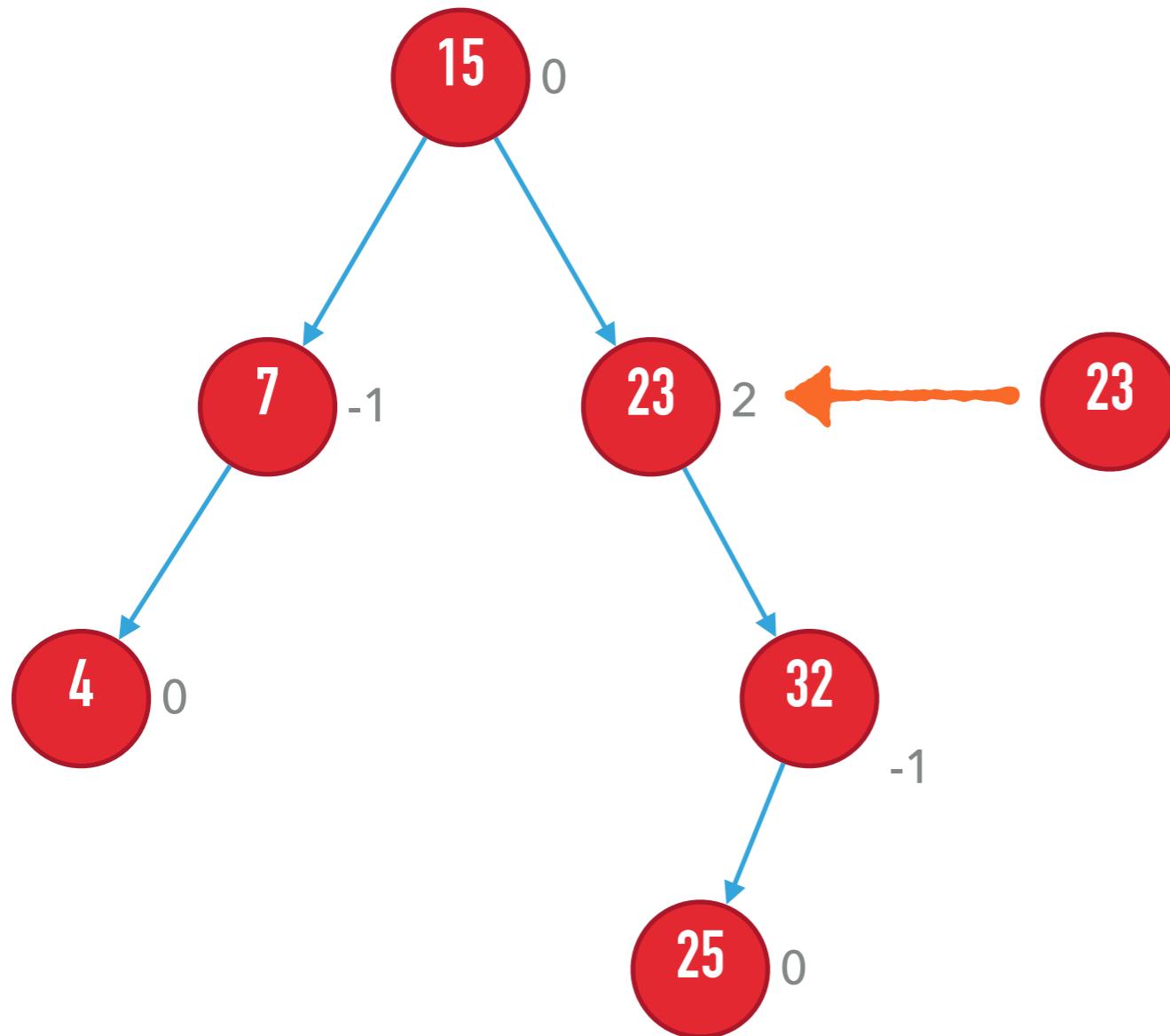
Inseriamo il nodo "25"



Stack dei nodi visitati

INSERIMENTO

Inseriamo il nodo "25"

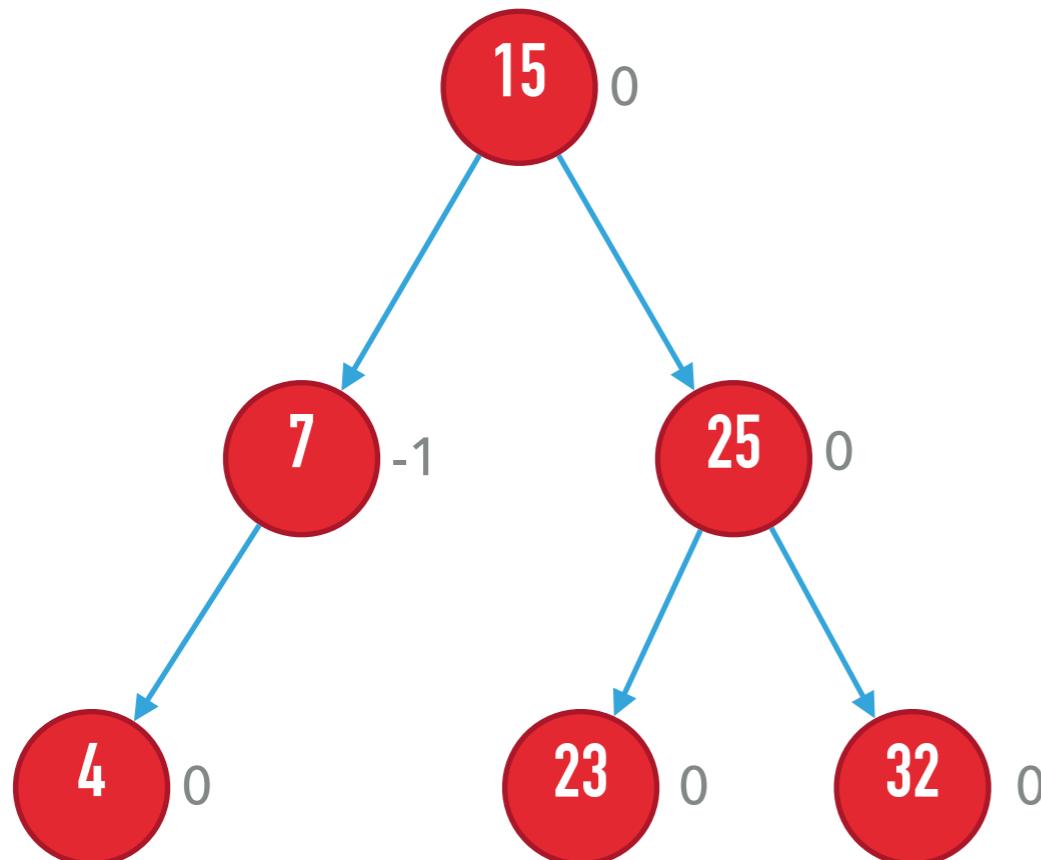


Stack dei nodi visitati

Abbiamo trovato il primo nodo sbilanciato: caso destra-sinistra

INSERIMENTO

Inseriamo il nodo "25"



Stack dei nodi visitati

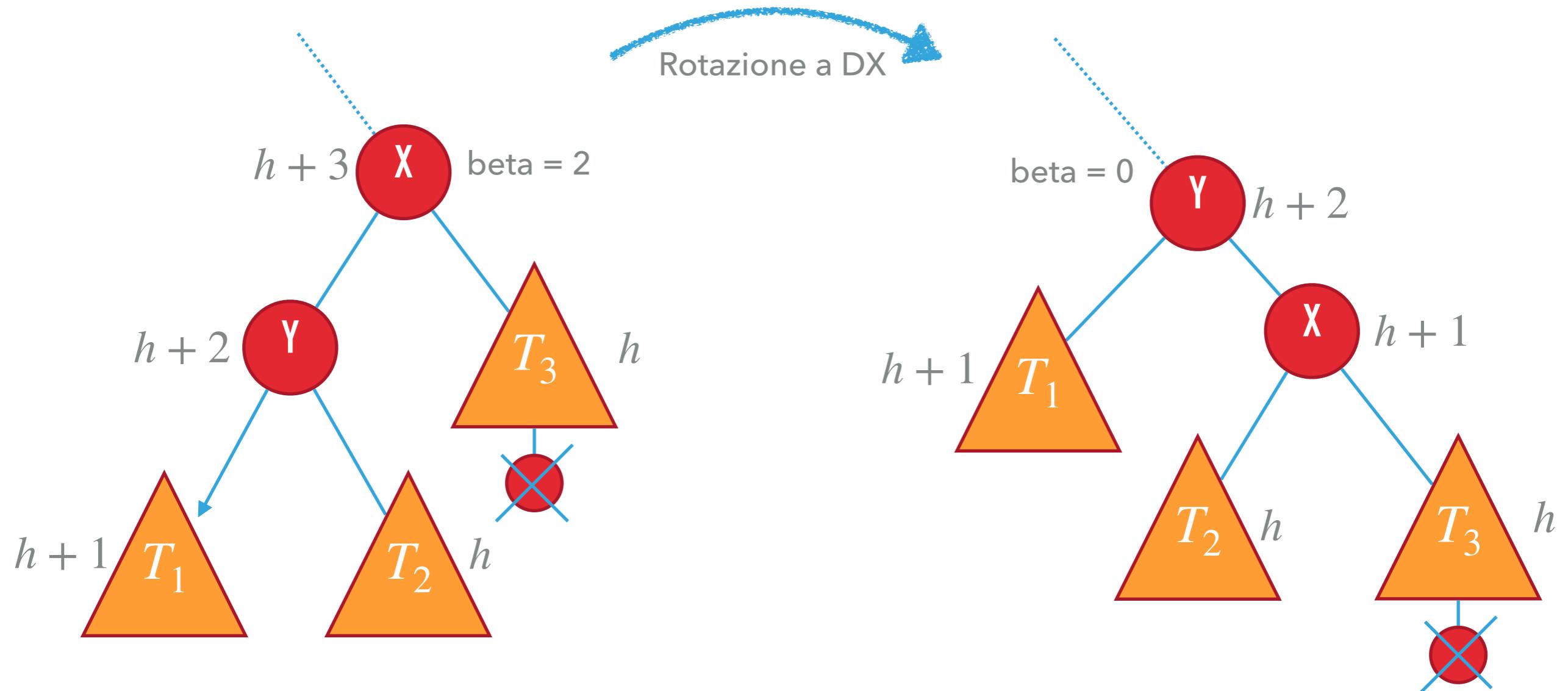
Abbiamo ribilanciato l'albero

CANCELLAZIONE

- ▶ La cancellazione avviene come per un Albero Binario di Ricerca, ma rimuovendo un nodo si può creare uno sbilanciamento.
- ▶ Utilizzando gli schemi di rotazioni visti prima possiamo ribilanciare.
- ▶ In questo caso però, l'altezza dopo il ribilanciamento descresce di una unità rispetto a prima della cancellazione.
- ▶ Quindi potremmo dover ribilanciare anche in tutti i nodi antenati. Quindi la cancellazione costa $O(h)$ passi.

CANCELLAZIONE

Se effettuiamo la rotazione notiamo come la proprietà degli alberi AVL venga ripristinata



Vediamo che l'albero torna ad essere bilanciato ma l'altezza decresce di uno.

ALBERI AVL

- ▶ Gli alberi AVL sono alberi con profondità $O(\log n)$
- ▶ Le operazioni di ricerca non cambiano rispetto agli alberi binari di ricerca normali
- ▶ Le operazioni di inserimento e rimozione rimangono $O(h)$, con h l'altezza dell'albero, quindi $O(\log n)$
- ▶ Richiedono però informazione aggiuntiva salvata nei nodi (come minimo 2 bit/nodo)

ALBERI ROSSO-NERI

ALBERI ROSSO-NERI

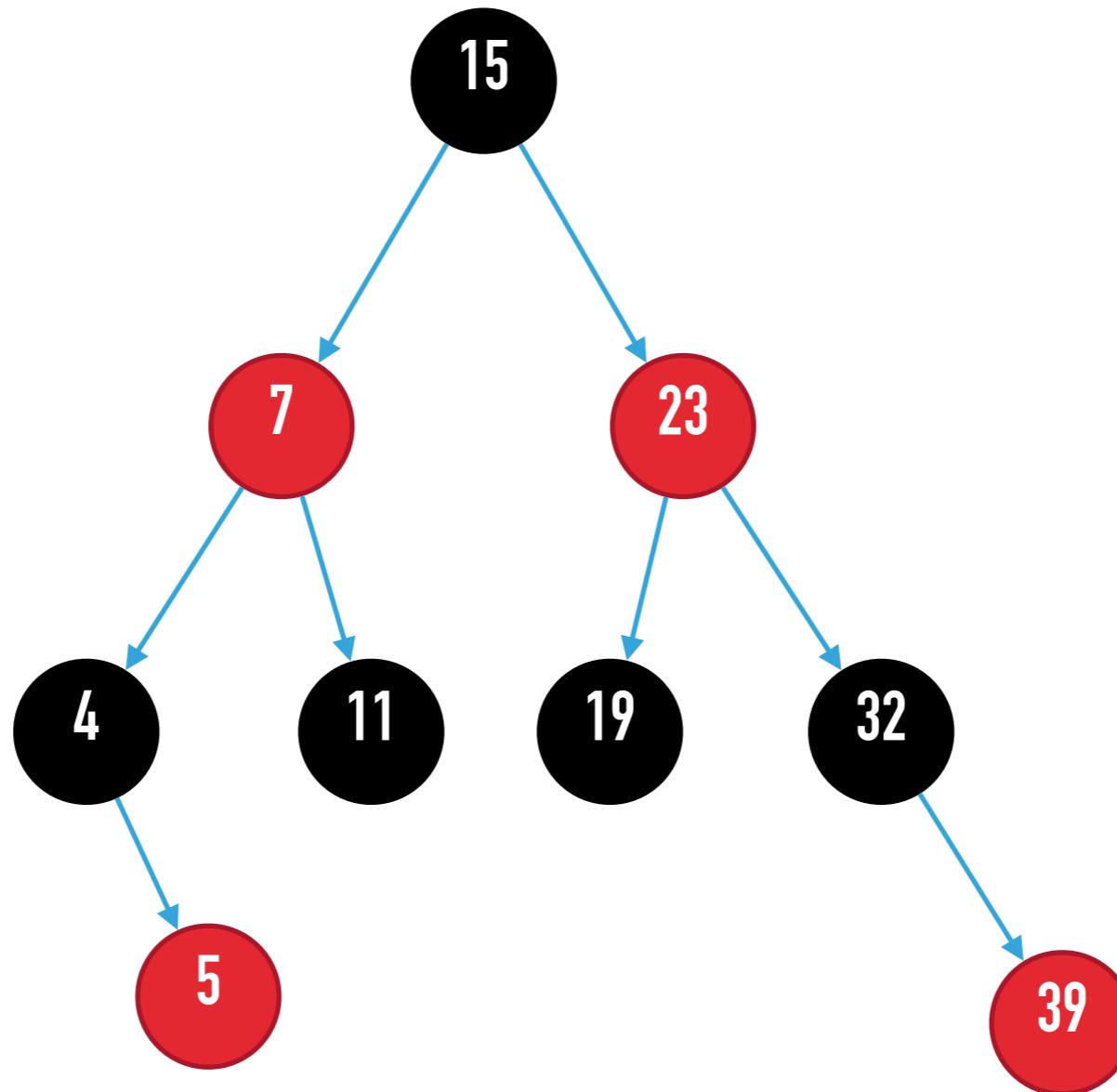
ALTRÉ TIPOLOGIE DI ALBERI BINARI DI RICERCA BILANCIATI

- ▶ Gli alberi AVL non sono l'unico tipo albero binario di ricerca bilanciato
- ▶ Un altro tipo di albero che si incontra spesso "in the wild" sono i **red-black tree** o alberi rosso-neri.
- ▶ Idea: ogni nodo contiene un bit di informazione aggiuntivo che dice se il nodo è colorato di **nero** o di **rosso**

PROPRIETÀ DEGLI ALBERI ROSSO-NERI

- ▶ Ogni nodo è colorato di **rosso** o di **nero**
- ▶ La radice deve essere colorata di **nero**
- ▶ I valori **None** si considerano colorati di **nero**
- ▶ I figli di un nodo **rosso** devono essere nodi **neri**
- ▶ Ogni percorso da un qualsiasi nodo ad un valore **None** nel sottoalbero del nodo (equiv. un percorso per arrivare alle foglie o ai nodi con reference a **None**) attraversa lo stesso numero di nodi **neri**

ALBERO ROSSO-NERO



Questo albero rispetta tutte le proprietà di albero rosso-nero

ALBERO ROSSO-NERO



Questo albero NON rispetta
tutte le proprietà di albero rosso-nero

ALBERI ROSSO-NERI: INSERIMENTO

- ▶ L'inserimento negli alberi rosso-neri inizia come per gli alberi binari di ricerca normali
- ▶ Il nodo da inserire viene colorato di rosso
- ▶ Se una proprietà degli alberi rosso-neri viene violata si eseguono una serie di ricolorazioni e rotazioni per ripristinarla (sempre in tempo $O(\log n)$)

ALBERI ROSSO-NERI: NODI CONTENUTI

- ▶ Mostriamo ora che un albero rosso-nero con n nodi ha altezza $O(\log n)$
- ▶ Dato che ogni nodo negli alberi rosso-neri ha lo stesso numero di nodi neri da lui stesso alle foglie e ai nodi con reference a None, possiamo definire la black height del nodo
- ▶ Dato un nodo x , definiamo $bh(x)$ il numero di nodi neri che si devono attraversare per arrivare da x a una qualsiasi foglia e nodo con reference a None

ALBERI ROSSO-NERI: NODI CONTENUTI

Mostriamo per induzione che ogni sottoalbero avente il nodo x come radice contiene almeno $2^{bh(x)} - 1$ nodi interni (i.e., foglie escluse).

La proprietà è vera se x è una foglia: $bh(x) = 0$, e contiene $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nodi interni.

Sia x un nodo con $bh(x) > 0$. Questo significa che ha due figli di altezza $bh(x)$ o $bh(x) - 1$ a seconda del colore.

ALBERI ROSSO-NERI: NODI CONTENUTI

Per ipotesi induttiva ciascuno dei sottoalberi ha almeno $2^{bh(x)-1} - 1$ nodi interni.

Quindi x ha almeno $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$, ovvero $2^{bh(x)} - 1$ nodi interni, che è quello che volevamo provare.

Consideriamo ora l'altezza h della radice r . Dato che la non possono esserci due nodi consecutivi rossi e che la radice è nera, essa ha $bh(r) \geq h/2$

ALBERI ROSSO-NERI: NODI CONTENUTI

Ma, per la proprietà precedente, l'intero albero può contenere al più $2^{bh(r)} - 1 = 2^{h/2} - 1$ nodi.

Da questo deriviamo

$$n \geq 2^{h/2} - 1$$

$$\log_2(n - 1) \geq \log_2(2^{h/2})$$

$$2 \log_2(n - 1) \geq h$$

E quindi $h \leq 2 \log_2(n - 1)$, quindi l'altezza dell'albero è al più due volte $\log_2(n - 1)$, quindi $O(\log n)$.

DYNAMIC SELECT

DYNAMIC SELECT

CALCOLARE I PERCENTILI IN UN INSIEME DINAMICO

Supponiamo di avere un albero binario di ricerca. Come facciamo a trovare il p-simo percentile, e.g. la mediana?

Partiamo dalla radice. Ci sono tre casi:

- a sx della radice r ci sono $m = \left\lfloor \frac{n}{2} \right\rfloor$ nodi: r è la mediana
- a sx della radice r ci sono $< m$ nodi: la mediana è a dx
- a sx della radice r ci sono $> m$ nodi: la mediana è a sx

AUMENTARE UNA STRUTTURA DATI

Per poter risolvere il problema di select efficientemente, devo quindi sapere quanti nodi contiene un sottoalbero.

Aggiungo ad ogni nodo x una variabile `size`, che conta il numero di nodi nel sottoalbero radicato in x . Vale

$$x.size = x.left.size + x.right.size + 1$$

Devo estendere le operazioni su un BST per mantenere aggiornato il campo `size`. In particolare per **INSERT** e **DELETE**.

INSERT PER DYNAMIC SELECT

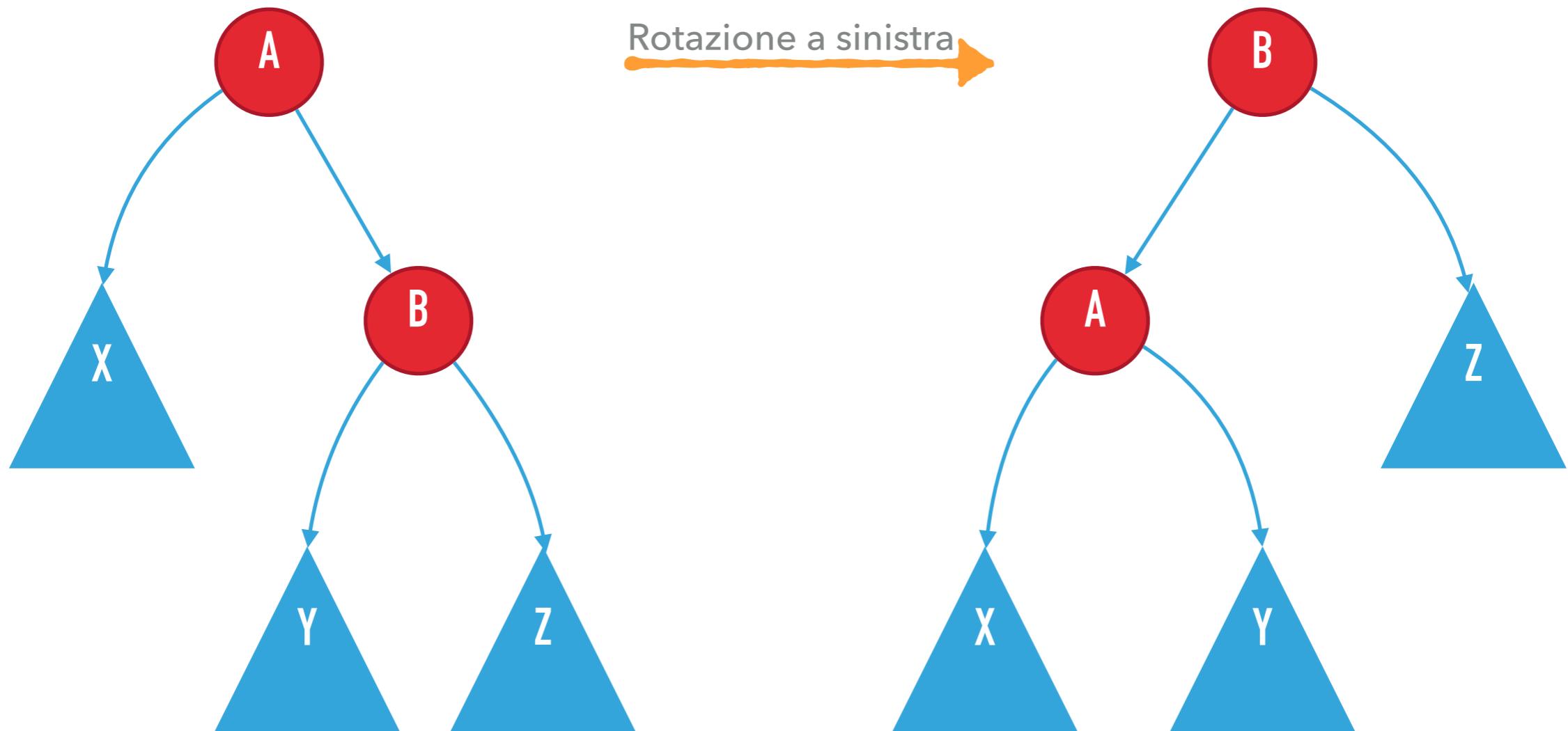
Inserisco il nodo normalmente come foglia. Quali nodi avranno un campo *size* diverso?

Risalgo dal nodo inserito alla radice dell'albero ed incremento *size* di 1 per ogni nodo.

Nel caso di un albero AVL, in cui posso fare delle rotazioni per bilanciare l'albero, devo modificare le rotazioni in modo che preservino il valore corretto di *size*.

ROTAZIONI PER DYNAMIC SELECT

Cambiano solo i valori di size in A e B



$$\text{size}(A) = \text{size}(A) - \text{size}(B.\text{right}) - 1$$

$$\text{size}(B) = \text{size}(B) + \text{size}(A.\text{left}) + 1$$

ROTAZIONE SINISTRA PER DYNAMIC SELECT

- ▶ Parametri: nodo a
- ▶ b = a.right # nodo che prenderà il posto di a
- ▶ **b.size = b.size + a.left.size + 1 # aggiorno size b**
- ▶ **a.size = a.size - b.right.size - 1 # aggiorno size a**
- ▶ a.right = b.left
- ▶ if a.right is not None:
 - ▶ a.right.parent = a
- ▶ b.left = a
- ▶ # con queste operazioni abbiamo messo il figlio sinistro di b come figlio destro di a, dobbiamo ancora sistemare i genitori di a e b
- ▶ if a.parent is not None: # sistemiamo il genitore di a
 - ▶ if a.parent.left is a: # nel caso a fosse figlio sinistro
 - ▶ a.parent.left = b
 - ▶ else # nel caso a fosse figlio destro
 - ▶ a.parent.right = b
- ▶ b.parent = a.parent
- ▶ a.parent = b

ANALISI AMMORTIZZATA
ALBERI SPLAY: ANALISI

ALGORITMI E STRUTTURE DATI

ANALISI AMMORTIZZATA

ANALISI AMMORTIZZATA

ANALISI DI SEQUENZE DI OPERAZIONI

- ▶ Possiamo valutare sequenze di operazioni invece di singole operazioni
- ▶ Data una sequenza di m operazioni su una struttura dati, valutiamo il loro costo computazionale $T(m)$
- ▶ Il costo ammortizzato di una singola operazione è quindi dato da $\frac{T(m)}{m}$
- ▶ Questo anche se la singola operazione ci può mettere di più!

ANALISI DI SEQUENZE DI OPERAZIONI

- ▶ In poche parole, se anche abbiamo alcune delle operazioni che sono costose, il loro costo è *ammortizzato* da molte operazioni poco costose
- ▶ Ci sono tre principali metodi per compiere una analisi ammortizzata del tempo di calcolo:
 - ▶ Metodo dell'aggregazione (*aggregate analysis*)
 - ▶ Metodo del banchiere (*accounting method*)
 - ▶ Metodo del potenziale (*potential method*)

FORMULAZIONE DEL PROBLEMA

Data una sequenza di m operazioni, vogliamo trovare il tempo $T(m)$ che queste m operazioni richiedono nel caso peggiore

Indicheremo con c_1, c_2, \dots, c_m il costo delle m operazioni

$$\text{Ne segue che } T(m) = \sum_{i=1}^m c_i$$

ALGORITMO D'ESEMPIO PER LO STUDIO

- ▶ Come algoritmo di esempi consideriamo uno stack implementato come array
- ▶ Quando l'array è pieno, la sua dimensione viene raddoppiata tramite copia
- ▶ Le operazioni che possiamo svolgere sono push e pop
- ▶ Qui valuteremo il costo ammortizzato di m operazioni di push (è lo worst case scenario)

ARRAY CON RADDOPPIO DI DIMENSIONE



push(3)



push(5)



push(2)



push(7)



push(9)

METODO DELL'AGGREGAZIONE

Significa semplicemente che si calcola direttamente che valore assume $T(m)$ e poi si divide per m per trovare il costo ammortizzato

Valutiamo il costo dell'inserimento in un array:

- ▶ Il costo di inserimento è 1 se non dobbiamo raddoppiare
- ▶ Altrimenti è pari al numero di elementi attualmente contenuti dell'array (dato che dobbiamo copiarli)

METODO DELL'AGGREGAZIONE

Quindi il costo è $c_i = \begin{cases} i & \text{se } i - 1 \text{ è una potenza di 2} \\ 1 & \text{altrimenti} \end{cases}$

Indichiamo con $d_i = c_i - 1$, ovvero d_i è il costo di copia degli elementi diversi da quello inserito

Ne segue che $d_i = \begin{cases} i - 1 & \text{se } i - 1 \text{ è una potenza di 2} \\ 0 & \text{altrimenti} \end{cases}$

METODO DELL'AGGREGAZIONE

La somma $\sum_{i=1}^m c_i$ può essere quindi spezzata in due parti:

$$T(m) = \sum_{i=1}^m 1 + \sum_{i=1}^m d_i$$

Che corrisponde a $T(m) = m + \sum_{i=1}^m d_i$

Ricordiamo che $d_i = 0$ quando $i - 1$ non è una potenza di 2

METODO DELL'AGGREGAZIONE

Maggioriamo la somma $\sum_{i=1}^m d_i$ con una somma di una quantità logaritmica di potenze di 2:

$$\sum_{i=1}^m d_i \leq \sum_{j=0}^k 2^j \text{ con } k = \lceil \log_2(m - 1) \rceil$$

La somma delle prime k potenze di 2 è $2^{k+1} - 1$

Ma $2^{k+1} - 1 = O(m)$

METODO DELL'AGGREGAZIONE

Quindi $T(m) = m + O(m) = O(m)$

Il costo ammortizzato di una singola operazione è quindi

costante: $\frac{O(m)}{m} = O(1)$

Conclusione: raddoppiando la dimensione quando l'array è pieno il costo ammortizzato di ogni singola operazione è costante

METODO DEL BANCHIERE

Il metodo del banchiere si basa sull'idea di associare ad ogni operazione un costo \hat{c}_i che può essere diverso dal costo reale.

La differenza $\hat{c}_i - c_i$ può essere positiva (in questo caso l'operazione i-esima "deposita" la differenza di costo)

Se la differenza $\hat{c}_i - c_i$ è negativa, allora l'i-esima operazione deve "prelevare" usando quanto "depositato" dalle operazioni precedenti

METODO DEL BANCHIERE

Abbiamo quindi che, per ogni $1 \leq k \leq m$ deve valere che:

$$\sum_{i=1}^k \hat{c}_i - \sum_{i=1}^k c_i \geq 0,$$

ovvero ad ogni operazione abbiamo sempre “pagato” abbastanza da sostenere il costo (i.e., il bilancio è non negativo)

Abbiamo quindi che per $k = m$ vale $\sum_{i=1}^m \hat{c}_i \geq \sum_{i=1}^m c_i$

e quindi $\sum_{i=1}^m \hat{c}_i$ ci fornisce un upper bound per $T(m)$

METODO DEL BANCHIERE

Ovviamente dipende tutto dalla scelta dei diversi \hat{c}_i .

Se scegliamo valori troppo piccoli la diseguaglianza non vale e non otteniamo un bound per $T(m)$.

Se li scegliamo troppo grandi non otteniamo dei bound stretti per il costo ammortizzato.

METODO DEL BANCHIERE - ESEMPIO

Per il nostro esempio scegliamo $\hat{c}_i = 3$ indipendentemente dall'operazione

Ipotesi: tutte le operazioni tra un raddoppio e il successivo "depositano" abbastanza per pagare il raddoppio

Consideriamo le prime i operazioni mostriamo che il bilancio rimane sempre non negativo.

METODO DEL BANCHIERE - ESEMPIO

$$3 - 1 = 2 \quad \text{operazione 1}$$

$$6 - (1 + 2) = 3 \quad \text{operazione 2}$$

$$9 - (1 + 2 + 3) = 3 \quad \text{operazione 3}$$

Il bilancio rimane sempre positivo dopo le prime tre operazioni. Questo è il caso base.

Assumiamo quindi di avere bilancio non negativo dopo un'operazione di ridimensionamento e mostriamo che il bilancio acquisto durante le successive operazioni ci permette di “pagare” la successiva operazione di ridimensionamento

METODO DEL BANCHIERE - ESEMPIO

Consideriamo la variazione del bilancio tra l'operazione $k + 2$ e l'operazione $2k$ dove k è una potenza di 2 con $k \geq 2$.

L'ultima operazione di ridimensionamento è stata con l'operazione $k + 1$ e la prossima sarà all'operazione $2k + 1$

Quindi $\sum_{i=k+2}^{2k} \hat{c}_i - c_i = 3(2k - (k + 1)) - (2k - (k + 1)) = 2k - 2$

Questo è quanto "depositato" tra un'espansione e la successiva

METODO DEL BANCHIERE - ESEMPIO

Consideriamo ora l'operazione $2k + 1$, che è una espansione.

Il costo è $2k + 1$, ma:

$$\sum_{i=1}^{2k+1} \hat{c}_i - \sum_{i=1}^{2k+1} c_i \geq 2k - 1 + \hat{c}_{2k+1} - c_{2k+1} = 2k - 2 + 3 - (2k + 1) = 0$$

Questo mostra che il bilancio rimane non negativo dopo l'operazione di ridimensionamento.

Quindi il costo ammortizzato $\hat{c}_i = 3$ è sufficiente

METODO DEL BANCHIERE - ESEMPIO

$$T(m) = \sum_{i=1}^m c_i \leq \sum_{i=1}^m \hat{c}_i = 3m$$

Quindi il costo di una singola operazione è costante.

Nota: la parte difficile è la scelta di un buon costo \hat{c}_i per le operazioni: usare 2 sarebbe stato troppo poco, usare m avrebbe funzionato ma ci avrebbe dato un risultato non stretto

METODO DEL POTENZIALE

Se le m operazioni che svolgiamo sono su una struttura dati che potenzialmente viene modificata da ogni operazione, otteniamo una sequenza stati della struttura dati:

$D_0, D_1, D_2, \dots, D_m$ dove D_i è lo stato della struttura dati dopo l'esecuzione dell'i-esima operazione

D_0 è lo stato iniziale

D_m è lo stato finale

METODO DEL POTENZIALE

Definiamo una funzione Φ che mappa gli stati della struttura dati in valori reali non negativi.

La funzione Φ è detta **funzione potenziale** e $\Phi(D_i)$ è il **potenziale associato alla struttura dati D_i**

Definiamo quindi il costo ammortizzato come il costo reale sommato ad una variazione di potenziale:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

METODO DEL POTENZIALE

Osserviamo ora come è definita la somma di tutti i costi ammortizzati

$$\sum_{i=1}^m \hat{c}_i = \sum_{i=1}^m (c_i + \Phi(D_i) - \Phi(D_{i-1}))$$

La serie $\sum_{i=1}^m \Phi(D_i) - \Phi(D_{i-1})$ è una serie telescopica:

$$\Phi(1) - \Phi(0) + \Phi(2) - \Phi(1) + \cdots + \Phi(m) - \Phi(m-1) = \Phi(m) - \Phi(0)$$

METODO DEL POTENZIALE

Possiamo quindi riscrivere la somma dei costi ammortizzati come

$$\sum_{i=1}^m \hat{c}_i = \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^m c_i$$

E, quindi, il costo reale della sequenza di m operazioni come:

$$\sum_{i=1}^m c_i = \Phi(D_0) - \Phi(D_m) + \sum_{i=1}^m \hat{c}_i$$

ovvero la somma dei costi ammortizzati meno la differenza di potenziale (solitamente definiamo $\Phi(D_0) = 0$)

METODO DEL POTENZIALE

Se riusciamo a definire $\Phi(D_m) \geq \Phi(D_0)$, abbiamo

$$\sum_{i=1}^m \hat{c}_i = \Phi(D_m) - \Phi(D_0) + \sum_{i=1}^m c_i \geq \sum_{i=1}^m c_i$$

E quindi la somma dei costi ammortizzati ci fornisce un buon bound sulla somma dei costi reali

Come per i due metodi precedenti, la scelta della giusta funzione potenziale è quella che permette all'analisi del costo ammortizzato di funzionare.

METODO DEL POTENZIALE

Nel nostro esempio scegliamo come funzione potenziale:

$$\Phi(D_i) = 2i - k$$

dove i è il numero di elementi contenuti nell'array e k è la dimensione dell'array

Notiamo che $\Phi(D_i) \geq 0$ per ogni $1 \leq i \leq m$, dato che ogni array è sempre pieno almeno per metà

Calcoliamo il costo ammortizzato di una operazione

METODO DEL POTENZIALE

Se l'operazione non richiede un incremento di dimensione dell'array allora:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (2i - k) - (2(i - 1) - k) = 3$$

Se invece fosse richiesto di incrementare la dimensione dell'array, allora la dimensione dell'array era $i - 1$ e quindi:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = i + (2i - 2(i - 1)) - (2(i - 1) - (i - 1)) = 3$$

Quindi otteniamo che il costo ammortizzato di ogni operazione rimane costante

ANALISI AMMORTIZZATA

ANALISI AMMORTIZZATA DEGLI ALBERI SPLAY

METODO DEL POTENZIALE: ALBERI SPLAY

Per calcolare il costo ammortizzato di una sequenza di m operazioni in un albero splay che contiene n elementi useremo il metodo del potenziale

Per fare questo dobbiamo definire una funzione potenziale.

METODO DEL POTENZIALE: ALBERI SPLAY

Definiamo:

$\text{size}(x)$ come il numero di elementi contenuti nel sottoalbero a radice x

Noi considereremo il rango di un nodo:

$$\text{rank}(x) = \log_2(\text{size}(x))$$

Il potenziale è definito quindi come la somma del rango di tutti i nodi dell'albero $\Phi(D_i) = \sum_{x \in D_i} \text{rank}(x)$

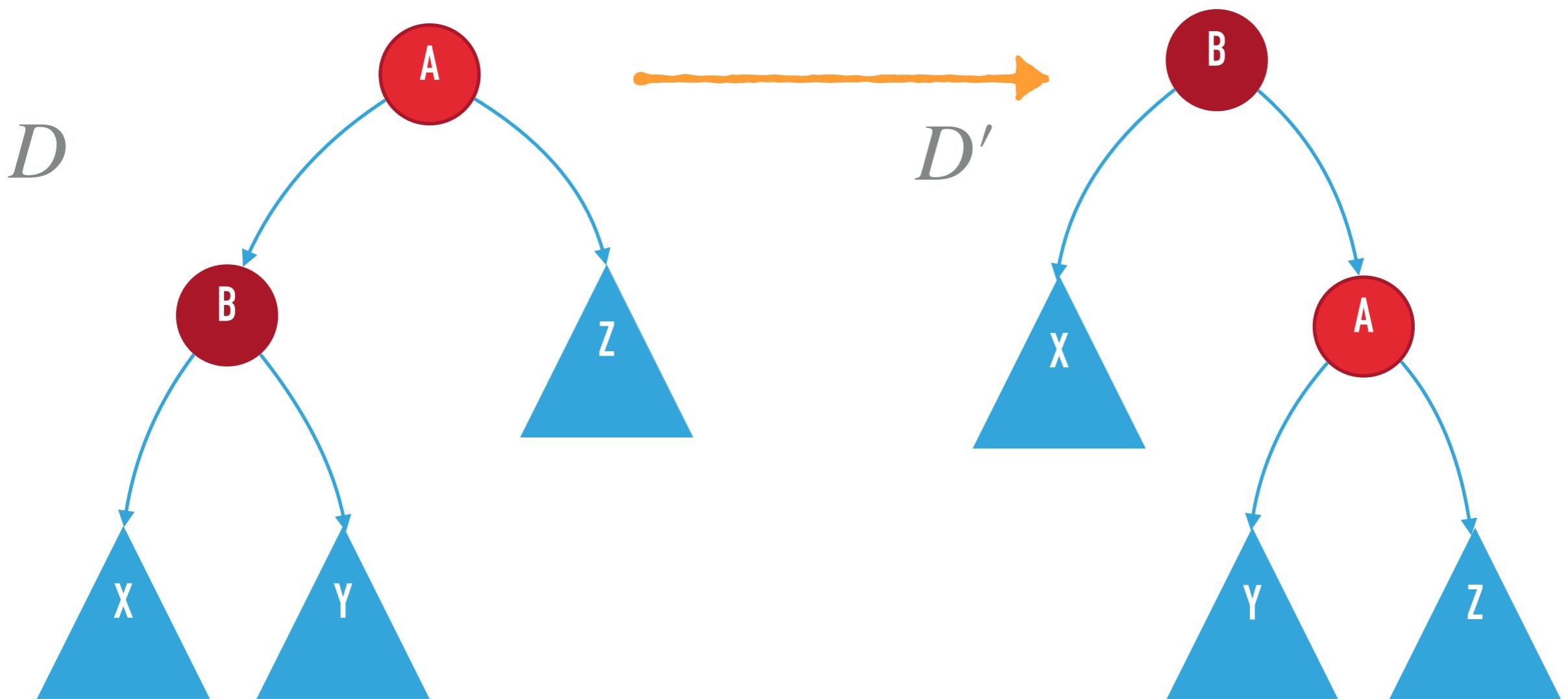
METODO DEL POTENZIALE: ALBERI SPLAY

Studiamo ora come ognuna delle operazioni usate per spostare il nodo cercato alla radice modifica il potenziale.

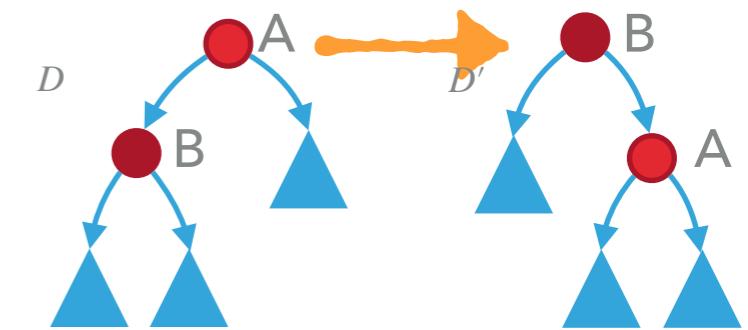
Le operazioni sono quindi zig, zig zig e zig zag. Ognuna di queste può però venire applicata più volte all'interno di una singola operazione c_i di ricerca.

CASO ZIG

Il nodo da muovere è figlio sinistro della radice



METODO DEL POTENZIALE: ALBERI SPLAY



Per il caso zig la variazione di potenziale è

$$\Phi(D') - \Phi(D) = \text{rank}'(A) + \text{rank}'(B) - \text{rank}(A) - \text{rank}(B)$$

Perché questi sono gli unici nodi per cui cambia il valore

Ma $\text{rank}(A) = \text{rank}'(B)$, quindi $\Phi(D') - \Phi(D) = \text{rank}'(A) - \text{rank}(B)$

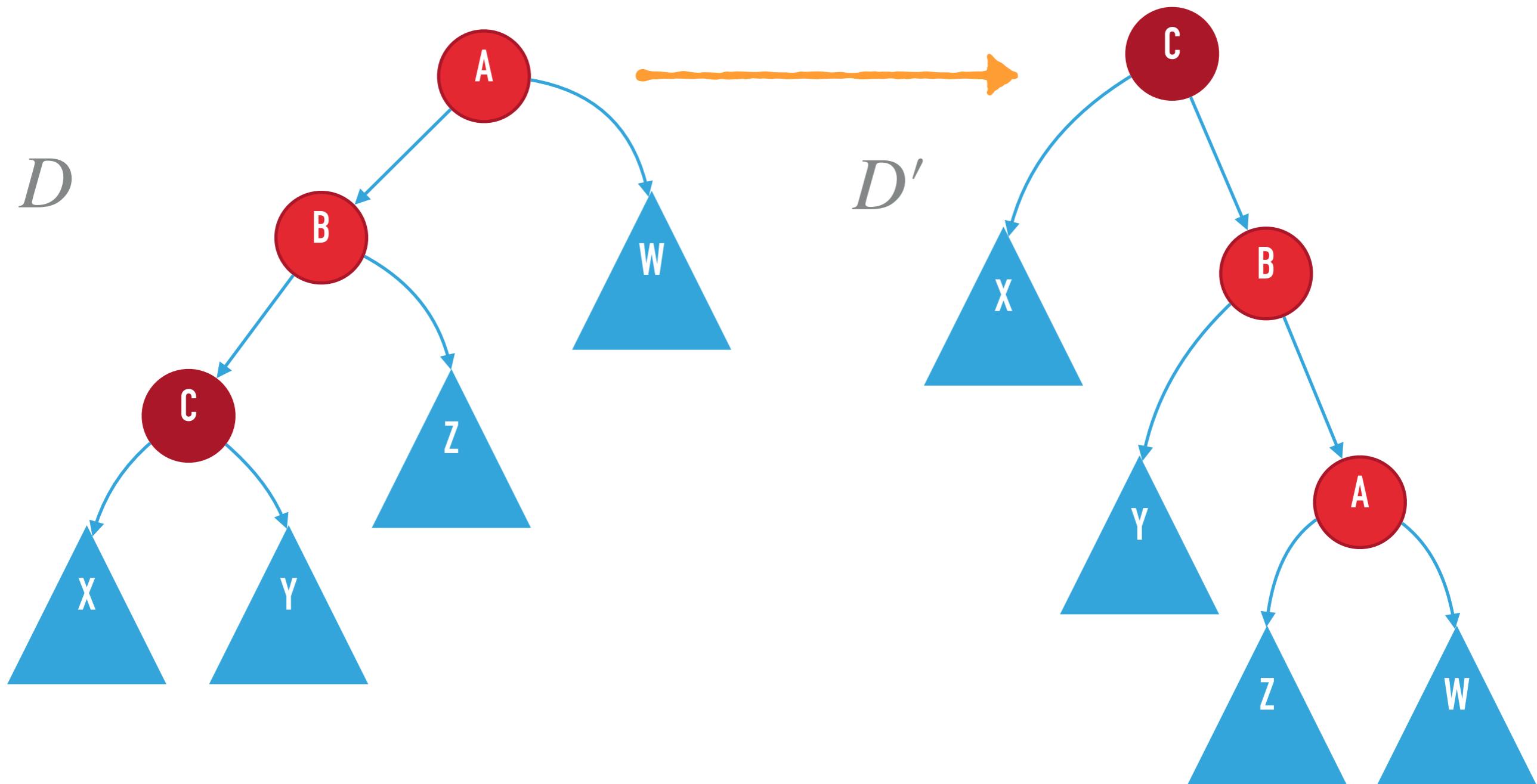
Dato che $\text{rank}'(A) \leq \text{rank}'(B)$ possiamo scrivere:

$$\Phi(D') - \Phi(D) \leq \text{rank}'(B) - \text{rank}(B)$$

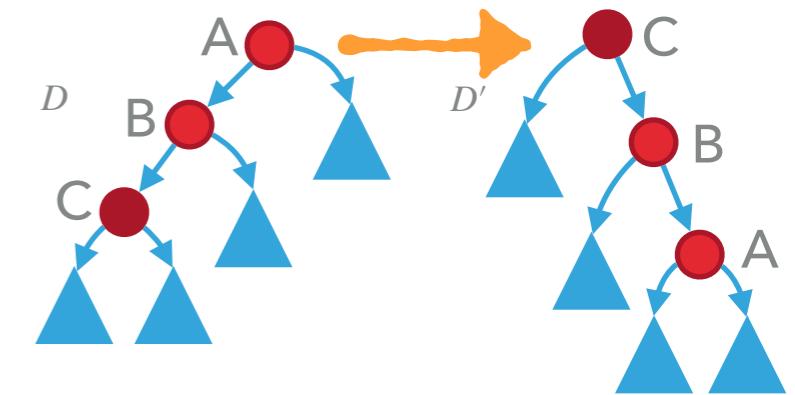
ovvero la variazione del rango solo del nodo cercato

CASO ZIG ZIG

Il nodo da muovere è figlio sinistro di un nodo che è a sua volta figlio sinistro



METODO DEL POTENZIALE: ALBERI SPLAY



Per il caso zig zig la variazione di potenziale è

$$\Phi(D') - \Phi(D) = \text{rank}'(A) + \text{rank}'(B) + \text{rank}'(C) - \text{rank}(A) - \text{rank}(B) - \text{rank}(C)$$

Perché questi sono gli unici nodi per cui cambia il valore

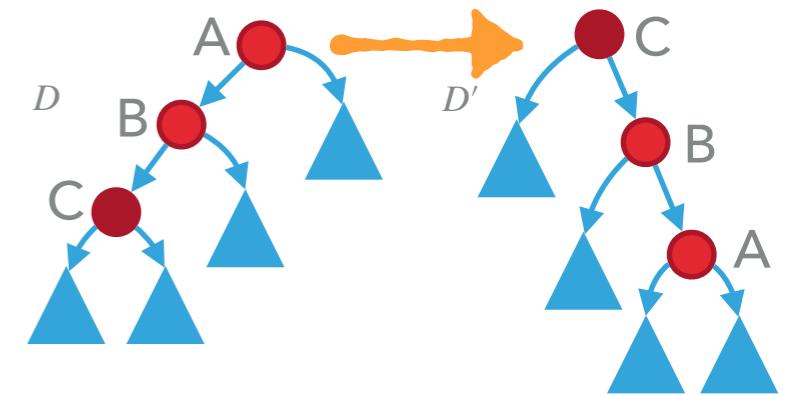
Ma $\text{rank}(A) = \text{rank}'(C)$, quindi

$$\Phi(D') - \Phi(D) = \text{rank}'(A) + \text{rank}'(B) - \text{rank}(B) - \text{rank}(C)$$

Ricordando che $\text{rank}(B) \geq \text{rank}(C)$:

$$\Phi(D') - \Phi(D) \leq \text{rank}'(A) + \text{rank}'(B) - \text{rank}(C) - \text{rank}(C)$$

METODO DEL POTENZIALE: ALBERI SPLAY



Ricordando anche che $\text{rank}'(B) \leq \text{rank}'(C)$:

$$\Phi(D') - \Phi(D) \leq \text{rank}'(A) + \text{rank}'(C) - 2\text{rank}(C)$$

poiché $\text{rank}'(C) > \text{rank}'(A)$:

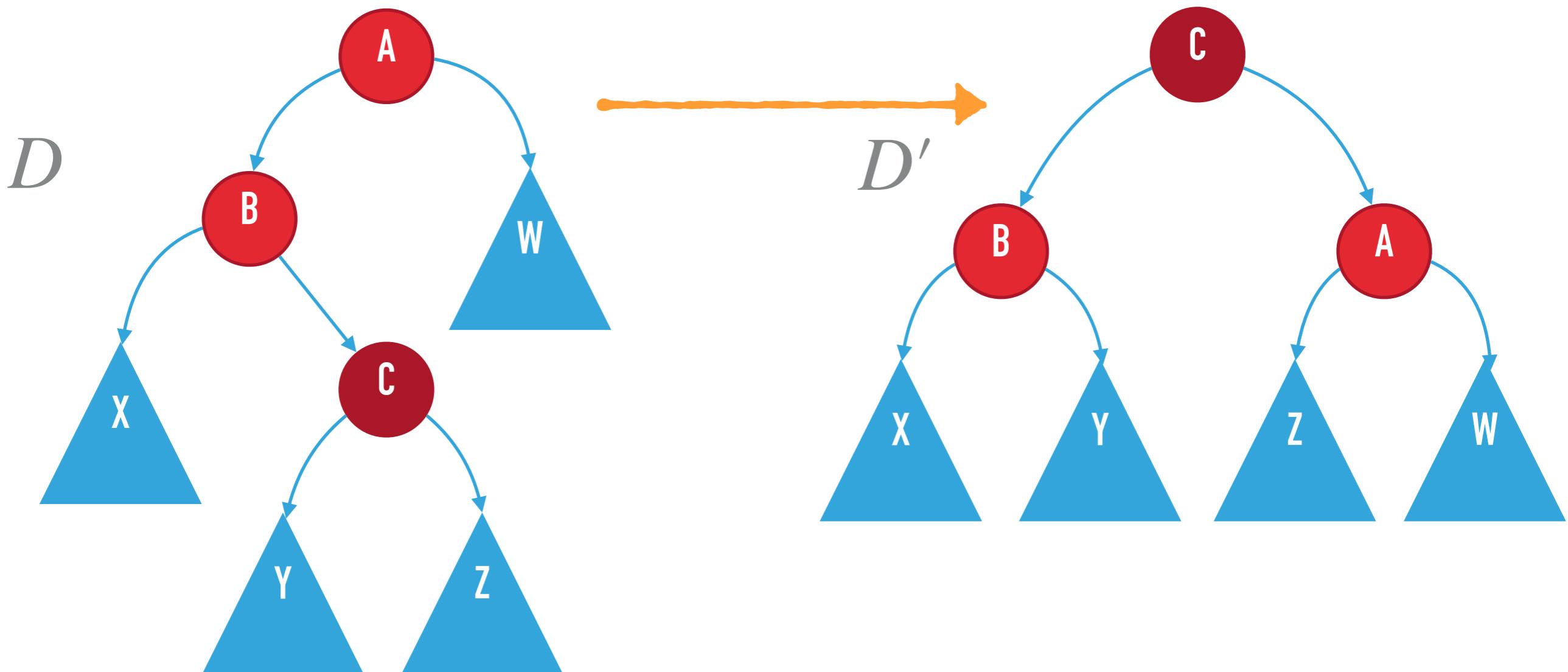
$$\Phi(D') - \Phi(D) \leq 2(\text{rank}'(C) - \text{rank}(C))$$

Analizzando il passo intermedio, vale questa maggiorazione

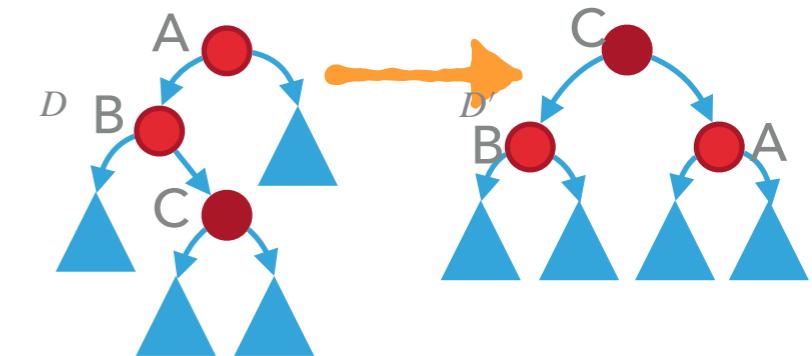
$$\Phi(D') - \Phi(D) \leq 3(\text{rank}'(C) - \text{rank}(C)) - 1$$

CASO ZIG ZAG

Il nodo da muovere è figlio destro di un nodo che è figlio sinistro



METODO DEL POTENZIALE: ALBERI SPLAY



Per il caso zig zag l'analisi è uguale al caso precedente:

$$\Phi(D') - \Phi(D) = \text{rank}'(A) + \text{rank}'(B) + \text{rank}'(C) - \text{rank}(A) - \text{rank}(B) - \text{rank}(C)$$

Perché questi sono gli unici nodi per cui cambia il valore

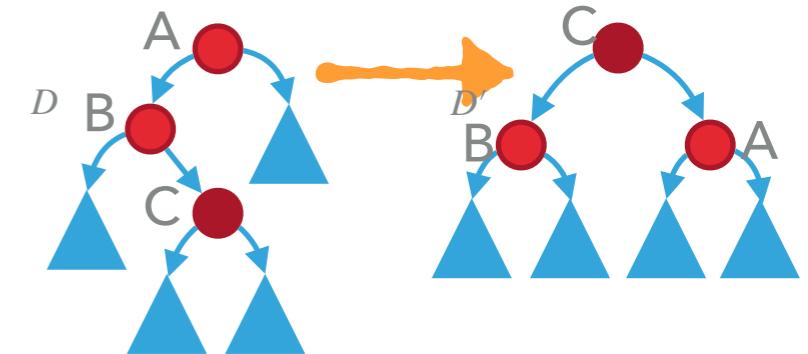
Ma $\text{rank}(A) = \text{rank}'(C)$, quindi

$$\Phi(D') - \Phi(D) = \text{rank}'(A) + \text{rank}'(B) - \text{rank}(B) - \text{rank}(C)$$

Ricordando che $\text{rank}(B) \geq \text{rank}(C)$:

$$\Phi(D') - \Phi(D) \leq \text{rank}'(A) + \text{rank}'(B) - \text{rank}(C) - \text{rank}(C)$$

METODO DEL POTENZIALE: ALBERI SPLAY



Ricordando anche che $\text{rank}'(B) \leq \text{rank}'(C)$:

$$\Phi(D') - \Phi(D) \leq \text{rank}'(A) + \text{rank}'(C) - 2\text{rank}(C)$$

poiché $\text{rank}'(C) > \text{rank}'(A)$:

$$\Phi(D') - \Phi(D) \leq 2(\text{rank}'(C) - \text{rank}(C))$$

Anche in questo caso, vale questa maggiorazione

$$\Phi(D') - \Phi(D) \leq 3(\text{rank}'(C) - \text{rank}(C)) - 1$$

METODO DEL POTENZIALE: ALBERI SPLAY

Se contiamo ogni passo splay come costo 1, abbiamo che il costo ammortizzato per spostare il nodo x è dato da

$1 + \text{rank}'(x) - \text{rank}(x)$ per le operazioni di zig

$1 + 3(\text{rank}'(x) - \text{rank}(x)) - 1$ per zig zig e zig zag

Se consideriamo la sequenza di operazioni di un passo splay vediamo che tutti i $\text{rank}(x)$ e $\text{rank}'(x)$ si cancellano tranne quello iniziale e l'ultimo, che è $\text{rank}(\text{root}) = \log_2 n$

METODO DEL POTENZIALE: ALBERI SPLAY

Il costo ammortizzato \hat{c}_i di una singola operazione è quindi

$$\hat{c}_i \leq \text{rank}(\text{root}) - \text{rank}_{D_{i-1}}(x_i)$$

Dove $\text{rank}_{D_{i-1}}(x_i)$ indica il rango del nodo cercato nell'istruzione i -esima nella struttura D_{i-1} .

Quindi $\hat{c}_i \leq \text{rank}(\text{root}) = \log_2 n$

Per una sequenza di m operazioni il costo ammortizzato è quindi $O(m \log n)$

METODO DEL POTENZIALE: ALBERI SPLAY

Abbiamo completato l'analisi?

No, ricordate: $\sum_{i=1}^m c_i = \Phi(D_0) - \Phi(D_m) + \sum_{i=1}^m \hat{c}_i$

Che nel nostro caso è $\sum_{i=1}^m c_i = \Phi(D_0) - \Phi(D_m) + O(m \log_2 n)$

Quindi dobbiamo anche trovare quale sia la variazione di potenziale nell'albero

METODO DEL POTENZIALE: ALBERI SPLAY

Possiamo sommare elemento per elemento

$$\sum_{x \in D_m} \text{rank}_{D_m}(x) - \sum_{x \in D_0} \text{rank}_{D_0}(x)$$

ma dato che l'albero contiene n nodi e la differenza massima di potenziale tra due nodi è $\log n$, sappiamo che possiamo maggiorare la somma sopra con $\sum_x \log n = n \log n$

METODO DEL POTENZIALE: ALBERI SPLAY

Otteniamo quindi che

$$\sum_{i=1}^m c_i = \Phi(D_0) - \Phi(D_m) + \sum_{i=1}^m \hat{c}_i = O(n \log n + m \log n)$$

Che è il risultato che volevamo per il **Balance Theorem**.

TABELLE HASH

ALGORITMI E STRUTTURE DATI

DIZIONARI: OLTRE GLI ALBERI BINARI DI RICERCA

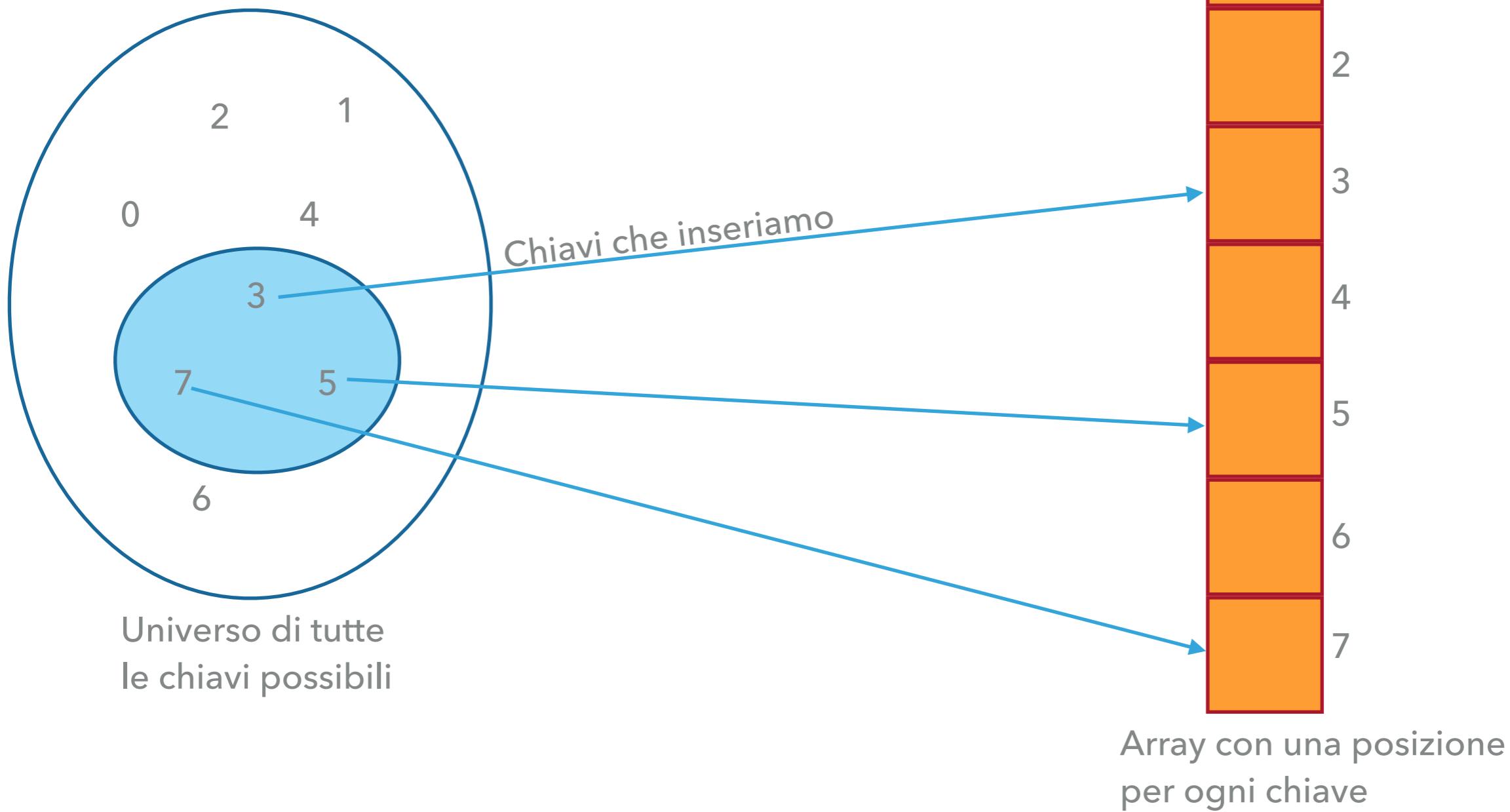
- ▶ Abbiamo visto come possiamo implementare un dizionario usando alberi binari di ricerca
- ▶ Con gli alberi bilanciati (AVL, rosso-neri) possiamo ottenere tempi di ricerca, inserimento e rimozione $O(\log n)$
- ▶ Possiamo fare di meglio se, per esempio, assumiamo di non dover implementare minimo, massimo, successore, predecessore?

TABELLE DI HASH

DIZIONARI: OLTRE GLI ALBERI BINARI DI RICERCA

- ▶ Supponiamo di conoscere tutti i possibili valori che una chiave può assumere: $K = \{k_1, \dots, k_m\}$. Assumiamo $K = \{0, \dots, m - 1\}$
- ▶ Se costruiamo un array di m elementi possiamo assegnare ad ogni chiave uno slot.
 - ▶ Inserimento: mettere il valore nello slot k_i
 - ▶ Ricerca: ritornare il contenuto dello slot k_i
 - ▶ Rimozione: cancellare il valore contenuto nello slot k_i

IDEA DI BASE



OPERAZIONI DEI DIZIONARI

Inserimento

Parametri: x (l'oggetto da inserire) e la tabella T

$T[x.key] = v$

Ricerca

Parametri: k (la chiave) e la tabella T

return $T[k]$

Rimozione

Parametri: x (l'oggetto da rimuovere) e la tabella T

$T[x.key] = \text{None}$

CI SONO DEI PROBLEMI?

- ▶ Questo approccio ci permette di fare ricerca, inserimento e rimozione in tempo costante!
- ▶ Problema: l'insieme di tutte le chiavi potrebbe non essere abbastanza piccolo da permettere questo approccio.
- ▶ Esempio: con numeri di 32 bit si avrebbero circa quattro miliardi di slot, anche salvando solo un byte per ogni slot avremmo 4GB di memoria occupati!
- ▶ Possiamo riformulare l'idea in modo che possa funzionare?

TABELLE HASH

- ▶ Invece di utilizzare direttamente le chiavi come indici, usiamo una funzione (detta *funzione di hash*) che, data una chiave, ci dice dove trovarla all'interno di una tabella
- ▶ Questo ci permette di definire quelle che sono chiamate le *tabelle hash*:
 - ▶ Dato un array, una chiave $k \in \mathcal{U}$ (universo delle chiavi), ed una funzione di hash h , la posizione in cui inserire/trovare k nell'array è $h(k) \in \{0, \dots, m - 1\}$.
 - ▶ Il codominio di h può essere molto ridotto!

UNA PRIMA TABELLA HASH

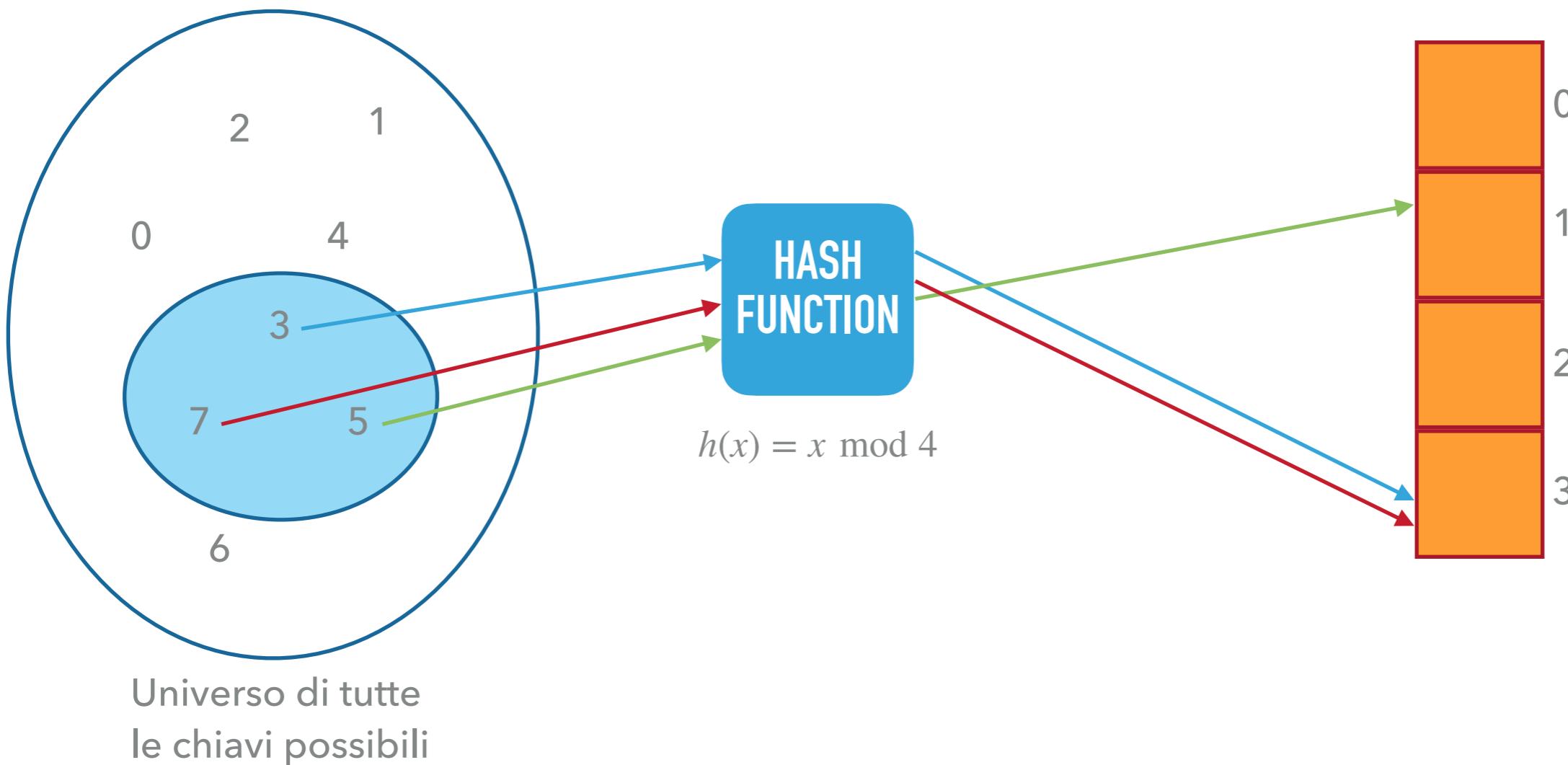


TABELLE HASH

- ▶ Finché non abbiamo due chiavi distinte con lo stesso hash (i.e., $k_1 \neq k_2$ ma $h(k_1) = h(k_2)$) tutte le operazioni continuano ad essere effettuabili in tempo costante (assumendo che h richieda tempo costante)
- ▶ Però dobbiamo gestire questo caso (le **collisioni**).
- ▶ A seconda di come decidiamo di gestirlo abbiamo diverse varianti di tabelle hash.

GESTIRE LE COLLISIONI

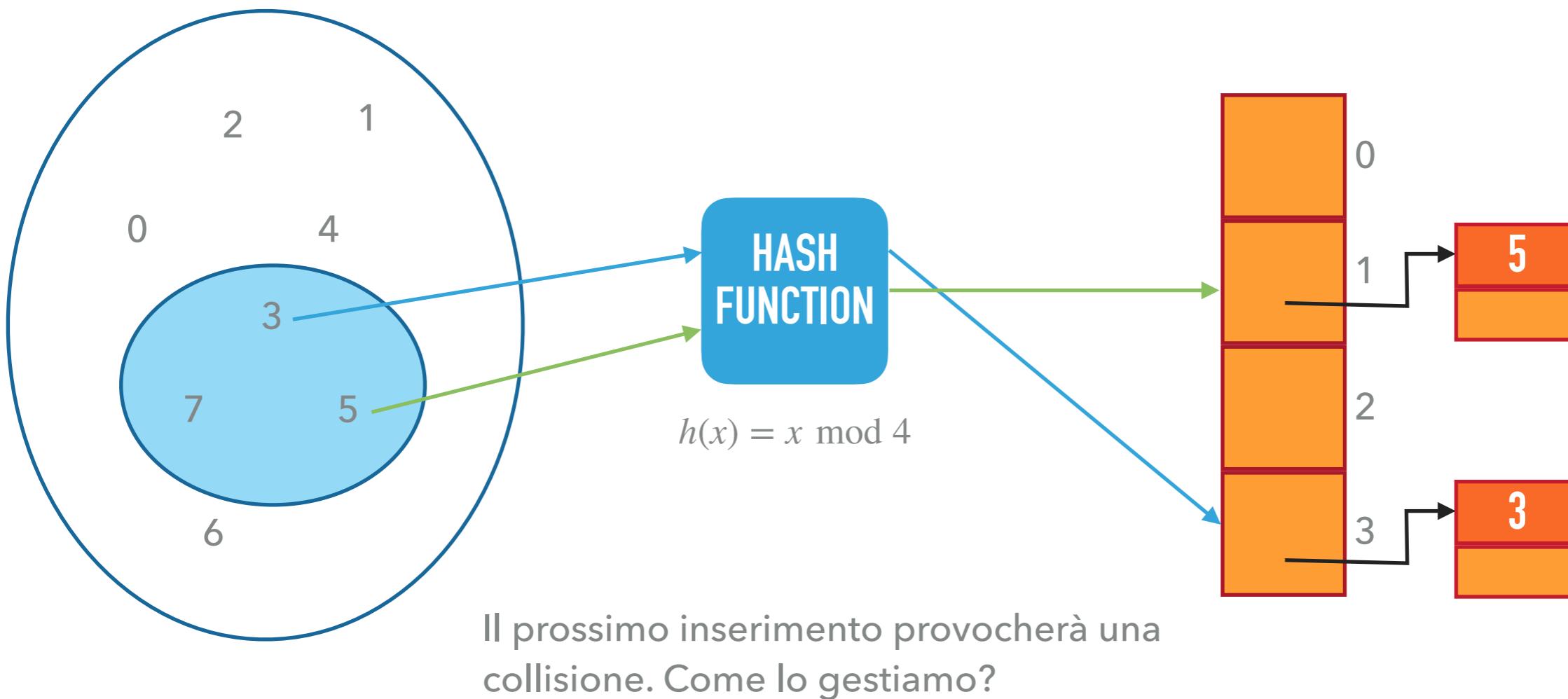
- ▶ Possiamo tenere per ogni slot una lista concatenata di valori che hanno lo stesso hash: **chaining** o “**hash con concatenazione**”
- ▶ Possiamo invece cercare un altro posto libero nella tabella: **open addressing** o **indirizzamento aperto**. Ne esistono diverse varianti, tra cui:
 - ▶ Ispezione lineare o quadratica
 - ▶ Doppio hashing

CHAINING

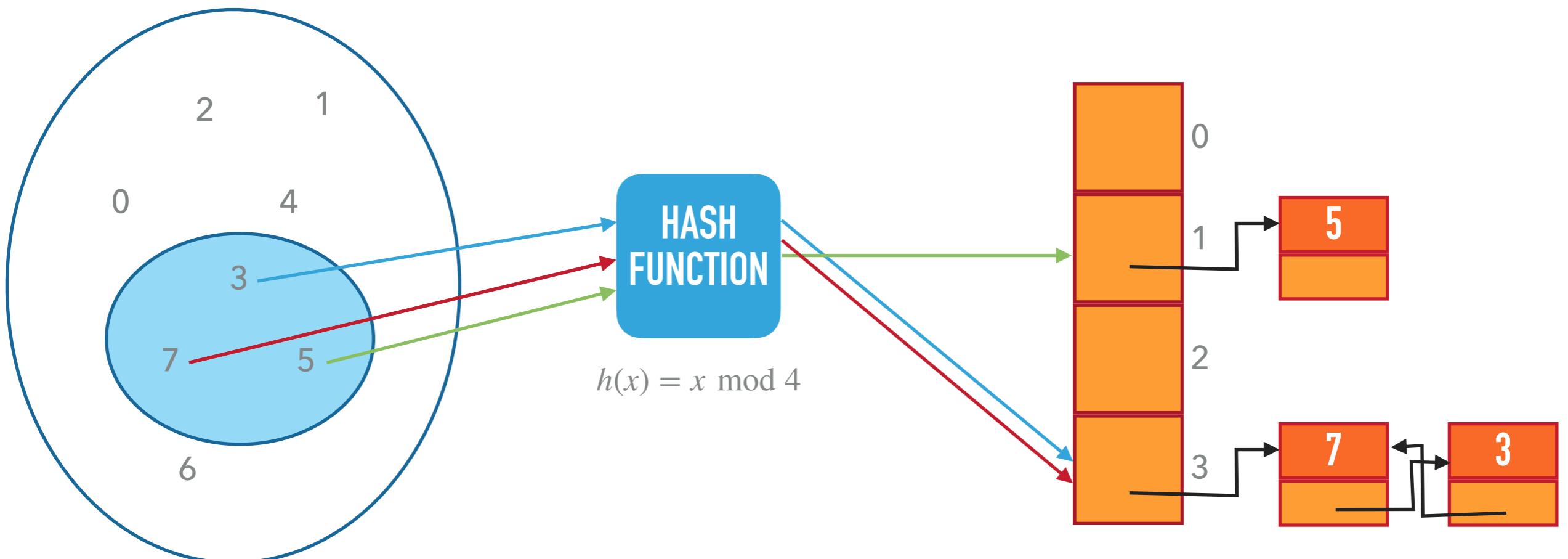
CHAINING / CONCATENAZIONE

- ▶ Abbiamo un array di m elementi, ognuno una lista concatenata (di solito doppia)
- ▶ L'inserimento di un elemento x di chiave k si riconduce a un inserimento in testa alla lista di indice $h(k)$
- ▶ La rimozione di un elemento x di chiave k si riconduce a una rimozione dalla lista di indice $h(k)$
- ▶ La ricerca data una chiave k si riconduce alla ricerca di k nella lista di indice $h(k)$

HASH CON CHAINING



HASH CON CHAINING



Effettuiamo l'inserimento in testa per avere sempre inserimenti in $O(1)$.
Usiamo una lista concatenata doppia per avere rimozioni in $O(1)$.

CHAINING / CONCATENAZIONE

Inserimento

Parametri: x (l'oggetto da inserire) e la tabella T
inserisci x in testa a $T[h(x.key)]$

Ricerca

Parametri: k (la chiave) e la tabella T
ricerca lineare nella lista $T[h(k)]$

Rimozione

Parametri: x (l'oggetto da rimuovere) e la tabella T
rimuovi x da $T[h(x.key)]$

CHAINING / CONCATENAZIONE

- ▶ Perché questo ci dovrebbe aiutare?
- ▶ Nel caso peggiore abbiamo tutti gli n elementi nella stessa lista concatenata
- ▶ Quindi, anche se inserimento e rimozione richiedono tempo $O(1)$, la ricerca richiede invece tempo $O(n)$
- ▶ L'analisi è corretta per il caso peggiore, ma possiamo ottenere qualcosa di meglio guardando il tempo medio?

TABELLE HASH: ALCUNE DEFINIZIONI

- ▶ Come al solito, indichiamo con n il numero di elementi contenuti nella tabella
- ▶ Indichiamo con m la dimensione della tabella
- ▶ $\alpha = n/m$ è il **load factor** o **fattore di carico** della tabella.
- ▶ Il fattore di carico indica quanto “piena” è la tabella:
 - ▶ $\alpha < 1$ abbiamo più posti nella tabella che elementi inseriti
 - ▶ $\alpha > 1$ abbiamo più elementi inseriti che posti nella tabella

CHAINING / CONCATENAZIONE

- ▶ Assumiamo che la funzione di hash distribuisca uniformemente le chiavi negli m slot:
- ▶ Sia K una variabile aleatoria su \mathcal{U} , con $p(h(K) = j) = \frac{1}{m}$
- ▶ Sotto queste assunzioni, mostriamo che il tempo medio per cercare un elemento in una tabella hash è $\Theta(1 + \alpha)$
- ▶ Dividiamo la dimostrazione in due parti:
 - ▶ Il caso in cui l'elemento cercato non sia nella tabella
 - ▶ Il caso in cui l'elemento cercato sia nella tabella

CHAINING / CONCATENAZIONE

Se l'elemento cercato non è presente, la chiave k con cui lo cerchiamo ha uguale probabilità di finire in uno qualsiasi degli m slot.

Quindi il tempo atteso per scoprire che la chiave non è presente è dato dalla lunghezza attesa della lista di indice $h(k)$.

Siano x_1, \dots, x_n gli elementi inseriti di chiavi k_1, \dots, k_n . Indichiamo con $Z_{i,j} = 1$ il caso $h(k_i) = j$ e $Z_{i,j} = 0$ altrimenti

Lunghezza attesa di una lista: $E \left[\sum_{i=1}^n Z_{ij} \right] = \sum_{i=1}^n E[Z_{ij}] = \frac{n}{m} = \alpha$ (fattore di carico)

Quindi il tempo atteso richiesto è un numero costante di passi più il tempo di cercare una lista di lunghezza α : $\Theta(1 + \alpha)$

CHAINING / CONCATENAZIONE

Se assumiamo che l'elemento x di chiave k sia presente nella lista, allora è egualmente probabile che sia uno qualsiasi degli n elementi presenti nella lista.

Gli elementi sono inseriti in testa alla lista, quindi il tempo richiesto per trovare x dipende da quanti elementi con lo stesso hash sono stati inseriti dopo di lui

Siano x_1, \dots, x_n gli elementi inseriti di chiavi k_1, \dots, k_n

Indichiamo con $X_{i,j} = 1$ il caso $h(k_i) = h(k_j)$ e $X_{i,j} = 0$ altrimenti

CHAINING / CONCATENAZIONE

Il numero di elementi da visitare prima di trovare x_i sarà quindi:

$$1 + \sum_{j=i+1}^n X_{i,j}$$

ovvero uno più tutti gli elementi inseriti dopo di lui

Dato che x potrebbe essere uno qualsiasi degli x_i , facciamo la media su tutte le possibili posizioni in cui potrebbe essere x , ottenendo il seguente tempo atteso:

$$E \left[\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n X_{i,j} \right) \right]$$

CHAINING / CONCATENAZIONE

Possiamo portare dentro il valore atteso per linearità:

$$\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[X_{i,j}] \right)$$

Dato che assumiamo che la funzione di hash distribuisca in modo uniforme le chiavi, abbiamo che la probabilità che $X_{i,j}$

sia 1 è $\frac{1}{m}$, quindi $E[X_{i,j}] = \frac{1}{m}$

CHAINING / CONCATENAZIONE

Otteniamo quindi

$$\frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m} \right) = \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{n} \sum_{i=1}^n \left(\frac{1}{m} \sum_{j=i+1}^n 1 \right)$$

$$= 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) = 1 + \frac{1}{nm} \left(\sum_{i=1}^n n - \sum_{i=1}^n i \right)$$

$$= 1 + \frac{1}{nm} \left(n^2 - \frac{n(n+1)}{2} \right) = 1 + \frac{n}{2m} - \frac{1}{2m}$$

CHAINING / CONCATENAZIONE

Sostituiamo quindi $\alpha = n/m$ ovunque ottenendo

$$1 + \frac{n}{2m} - \frac{1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1 + \alpha)$$

Sommato al tempo di calcolo per la funzione di hash (che è costante), otteniamo $\Theta(1 + \alpha)$.

Questo mostra che finché $n = O(m)$, abbiamo

$$\alpha = \frac{O(m)}{m} = O(1) \text{ e quindi il tempo medio per la ricerca è } O(1).$$

CHAINING / CONCATENAZIONE: DISCUSSIONE

- ▶ Questi risultati valgono sotto le assunzioni di avere un “buona” funzione di hash che distribuisce in modo uniforme le chiavi
- ▶ Serve inoltre che $n = O(m)$, quindi se la tabella su riempie troppo servirà sostituirla con una più grande (e.g., raddoppiando il numero di slot) e reinserire tutti i valori contenuti nella tabella vecchia.

FUNZIONI DI HASH

- ▶ Fino ad ora non abbiamo detto quali funzioni di hash sono considerate buone.
- ▶ Una buona funzione di hash dovrebbe rispettare la proprietà di distribuire le chiavi uniformemente negli m slot a disposizione...
- ▶ ... ma solitamente non sappiamo con che distribuzione di probabilità sono prese le chiavi che inseriamo

FUNZIONI DI HASH

- ▶ Se abbiamo buona conoscenza della distribuzione con cui sono ottenute le chiavi possiamo costruire una funzione di hash ad-hoc
- ▶ In generale utilizziamo alcune euristiche che funzionano bene in pratica e in cui vediamo le chiavi come numeri naturali:
 - ▶ Il metodo della divisione
 - ▶ Il metodo della moltiplicazione

METODO DELLA DIVISIONE

- ▶ Data una tabella di dimensione m , per ogni chiave definiamo $h(k)$ come $h(k) = k \bmod m$
- ▶ Generalmente un metodo di hashing rapido
- ▶ Vogliamo però evitare alcuni valori di m che possono essere problematici.
- ▶ Vediamo alcuni di questi casi

METODO DELLA DIVISIONE

- ▶ Se $m = 2^p$ per qualche $p \in \mathbb{N}$, il valore di $h(x)$ dipende solo dai p bit meno significativi di x
- ▶ Si vede bene in base 10: se $m = 100$ abbiamo che $h(x)$ dipende solo dalle ultime due cifre di x , quindi 8298, 43298, 198 hanno tutti lo stesso hash
- ▶ Generalmente buone scelte per m sono primi vicini a potenze di 2. Questo permette di avere un valore di hash che dipende da tutti i bit della chiave

METODO DELLA MOLTIPLICAZIONE

- ▶ Creazione di una funzione di hash in due passi
- ▶ Si sceglie una costante A con $0 < A < 1$
- ▶ Si moltiplica la chiave x per A e si prende la parte frazionaria: $Ax - [Ax]$
- ▶ La parte frazionaria si moltiplica per m e del risultato si prende la parte intera:
$$h(x) = \lfloor m(Ax - [Ax]) \rfloor$$

METODO DELLA MOLTIPLICAZIONE

- ▶ Il vantaggio di questo metodo è che il valore di m non è critico, quindi possiamo scegliere una potenza di 2.
- ▶ La parte critica è invece il valore di A , ma questo non deve cambiare se dobbiamo ingrandire la tabella.
- ▶ Knuth suggerisce un valore $A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887$

TABELLE HASH

ALGORITMI E STRUTTURE DATI

INDIRIZZAMENTO APERTO

- ▶ Utilizzando il chaining andavamo ad utilizzare spazio al di fuori di quello dell'array
- ▶ Con l'indirizzamento aperto vogliamo invece tenere tutto all'interno dell'array
- ▶ Al contrario del chaining è quindi richiesto che $m \geq n$, dato che abbiamo solo m posti in cui inserire i valori

INDIRIZZAMENTO APERTO

- ▶ Chiaramente, dobbiamo applicare una nuova strategia per risolvere le collisioni
- ▶ La strategia di base è quella di avere una sequenza di posizioni da provare ed inserire nella prima che si trova libera
- ▶ Vogliamo inoltre che se esiste un posto libero questo sia nella sequenza di posizioni da trovare

INDIRIZZAMENTO APERTO

- ▶ Estendiamo la funzione di hashing con il concetto di **probe function**, ovvero $h : U \times \{0, \dots, m - 1\} \rightarrow \{0, \dots, m - 1\}$ dove U è l'insieme delle possibili chiavi
- ▶ $h(k, 0)$ indicherà la prima posizione in cui provare a inserire k , $h(k, 1)$ la seconda posizione, etc.
- ▶ Se $h(k, 0), h(k, 1), \dots, h(k, m - 1)$ è una permutazione di $0, 1, \dots, m - 1$ allora potenzialmente se esiste un posto libero lo troveremo (visitiamo tutte le posizioni dell'array)

INDIRIZZAMENTO APERTO

- ▶ Mentre l'inserimento è relativamente facile da definire dobbiamo stare attenti a definire la ricerca e, soprattutto, la cancellazione
- ▶ Per la ricerca, data la chiave k , non ci dobbiamo fermare a $h(k,0)$, ma continuare finché non troviamo k o una posizione vuota

INSERIMENTO (OPEN ADDRESSING)

Inserimento

Parametri: x (l'oggetto da inserire) e la tabella T

$i = 0$

$pos = h(x.key, i)$

while $T[pos]$ is not None and $i < m$:

iteriamo fino a quando non troviamo un posto libero

$i = i + 1$

$pos = h(x.key, i)$

if $i == m$: # se non c'è un posto dopo m iterazioni la tabella è piena

 Errore: Tabella piena

else:

$T[pos] = x$

RICERCA (OPEN ADDRESSING)

Ricerca

Parametri: k (chiave della ricerca) e la tabella T

$i = 0$

$pos = h(k, i)$

while $T[pos]$ is not None and $i < m$ and $T[pos].key \neq k$:

iteriamo fino a quando non troviamo un posto libero o non troviamo k

$i = i + 1$

$pos = h(x.key, i)$

if $i == m$ or $T[pos]$ is none: # non abbiamo trovato l'elemento

return None

else:

return $T[pos]$

HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

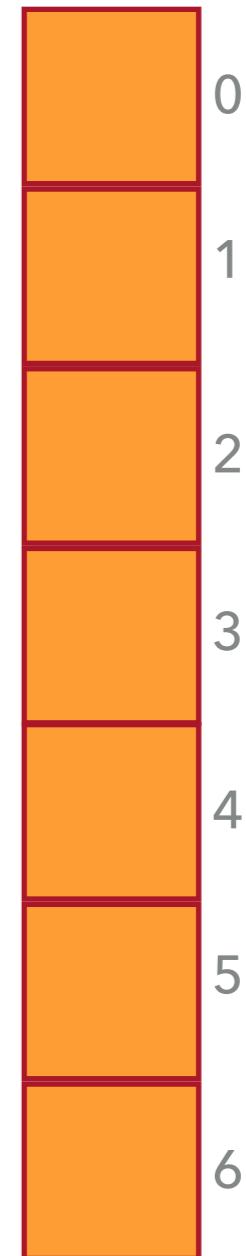
Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

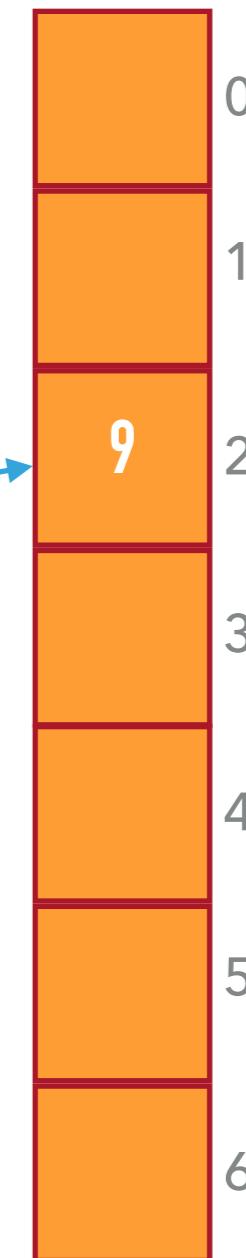
Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash

$$h(9, 0) = 2$$



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

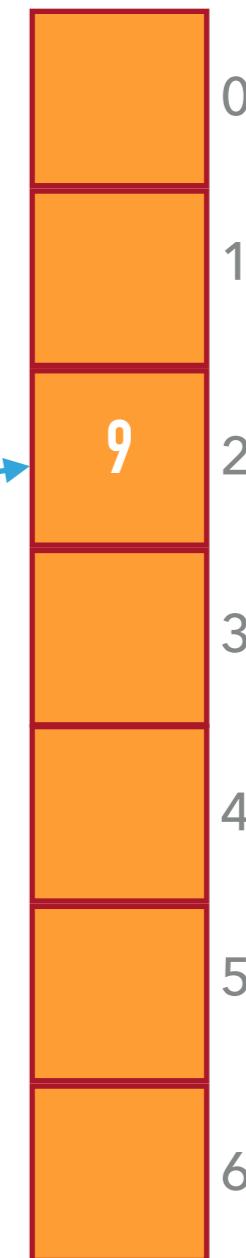
Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione
di hash**

$$h(23, 0) = 2$$

Non possiamo inserirlo
perché la posizione è
già occupata



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione
di hash**

$$h(23, 1) = 3$$

L'inserimento va a buon fine



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

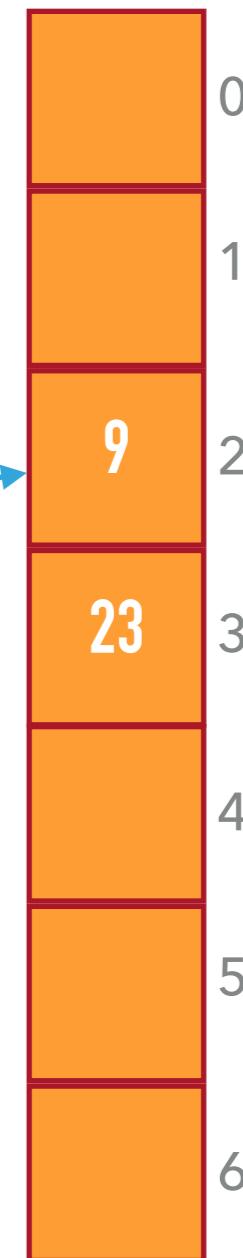
Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash

$$h(16, 0) = 2$$

Non possiamo inserirlo
perché la posizione è
già occupata



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione
di hash**

$$h(16, 1) = 3$$

Non possiamo inserirlo
perché la posizione è
già occupata



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

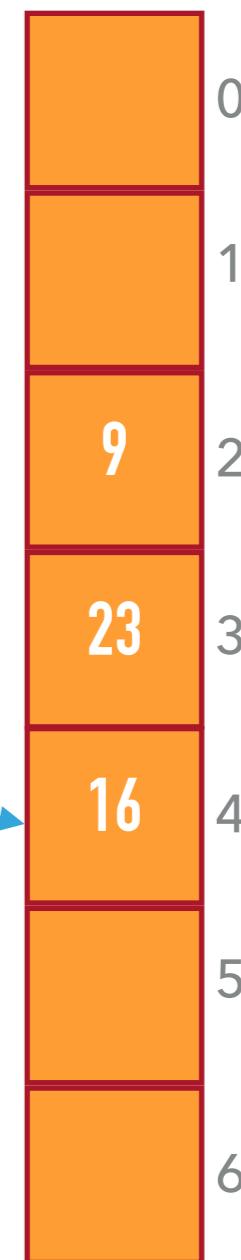
Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash

$$h(16, 2) = 4$$

L'inserimento va a buon fine



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

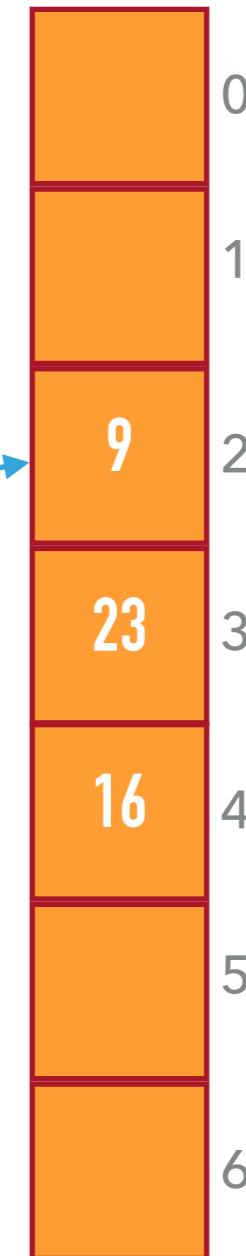
Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione
di hash**

$$h(23, 0) = 2$$

Non cancelliamo perché
la chiave salvata in
posizione 2 non è 23



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

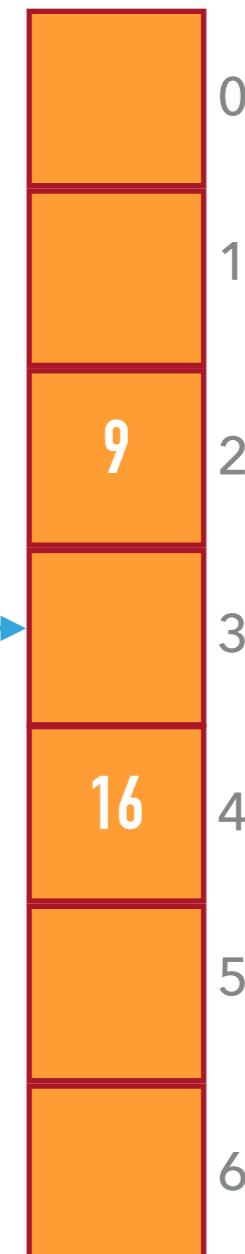
Funzione
di hash

$$h(23, 1) = 3$$

Attenzione, è corretto cancellare semplicemente
il contenuto in posizione 3?

NO! Se cercassimo "16" incontreremmo un posto vuoto
e ci fermeremmo erroneamente prima di aver completato
la ricerca

Questa volta cancelliamo
perché la chiave in
posizione 3 è 23.



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

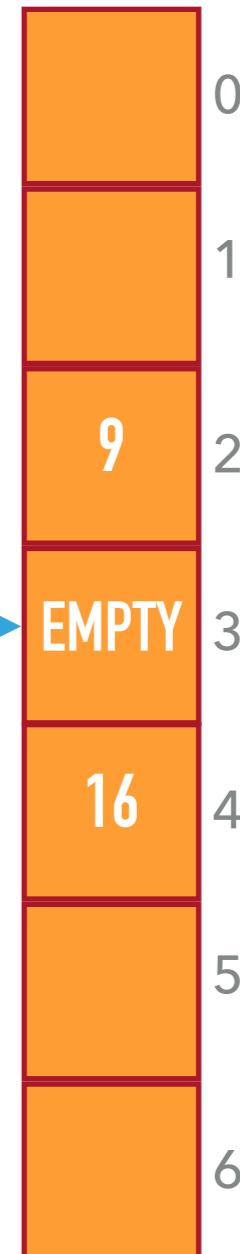
Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

**Funzione
di hash**

$$h(23, 1) = 3$$

Inseriamo un valore "segnaposto"
Indica che la casella è libera ma
conteneva un elemento cancellato



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash

$$h(16, 0) = 2$$

Non trovato, proseguiamo



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash

$$h(16, 1) = 3$$

Non trovato, proseguiamo
anche se la casella è vuota
perché è lo speciale valore
segnaposto



HASH: INSERIMENTO, RIMOZIONE E RICERCA

Inseriamo 9

Inseriamo 23

Inseriamo 16

Cancelliamo 23

Cerchiamo 16

$$h(k, i) = (k + i) \bmod 7$$

Funzione
di hash

$$h(16, 2) = 4$$

Trovato!



CANCELLAZIONE (OPEN ADDRESSING)

Cancellazione

Parametri: x (oggetto da cancellare) e la tabella T

$i = 0$

$pos = h(k, i)$

while $T[pos] \neq x$:

iteriamo fino a quando non troviamo x (assumiamo x contenuto in T)

$i = i + 1$

$pos = h(x.key, i)$

$T[pos] = \text{EMPTY}$ # valore segnaposto

Attenzione, dobbiamo modificare la procedura di inserimento
per inserire nelle caselle “EMPTY”

INSERIMENTO (OPEN ADDRESSING) - CON CANCELLAZIONE

Inserimento

Parametri: x (l'oggetto da inserire) e la tabella T

$i = 0$

$pos = h(x.key, i)$

while $T[pos]$ is not None and $i < m$ and $T[pos]$ is not EMPTY:

iteriamo fino a quando non troviamo un posto libero

$i = i + 1$

$pos = h(x.key, i)$

if $i == m$: # se non c'è un posto dopo m iterazioni la tabella è piena

 Errore: Tabella piena

else:

$T[pos] = x$

ALCUNE NOTE SULLA CANCELLAZIONE

- ▶ Se non cancelliamo otteniamo dei buoni bound sul tempo necessario alla ricerca che dipenderanno dal fattore di carico α
- ▶ Se consentiamo la cancellazione, invece la questione è molto più complessa:
 - ▶ Immaginate di riempire la tabella e poi svuotarla completamente
 - ▶ Ora ogni ricerca deve comunque passare per tutte le posizioni (che sono EMPTY e non None) prima di fallire

ALCUNE NOTE SULLA CANCELLAZIONE

- ▶ A causa di questi problemi il chaining viene di solito scelto se è necessario cancellare.
- ▶ Rimangono comunque alcune alternative: delle operazioni di “pulizia” tramite re-inserimento in una tabella nuova, “spostare” gli elementi invece marcare come EMPTY, etc.
- ▶ Noi ora proseguiremo assumendo di non avere la cancellazione ma solo inserimenti e ricerche (quindi niente valori segnaposto EMPTY)

TIPOLOGIE DI PROBING

- ▶ Ci sono diverse strategie per effettuare il probing, noi ne vediamo tre diversi tipi:
 - ▶ Linear probing / Probing lineare
 - ▶ Quadratic probing / Probing quadratico
 - ▶ Double hashing / Doppio hashing
- ▶ Vediamo per ognuna vantaggi e svantaggi

LINEAR PROBING

- ▶ Data una funzione di hash $h' : U \rightarrow \{0, \dots, m - 1\}$ la modifichiamo definendo una funzione h come:
$$h(x, i) = (h'(x) + i) \bmod m$$
- ▶ In pratica significa che iniziamo a testare da $h'(x)$, poi $h'(x) + 1$, etc. incrementando di uno alla volta ed eventualmente ricominciando da zero quando raggiungiamo il valore m

HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash $h'(k)$

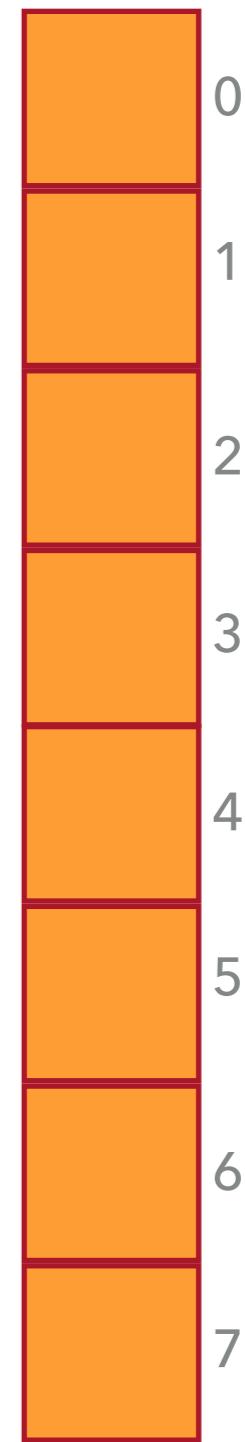
Inseriamo 33

Inseriamo 47

Inseriamo 12



$$h(k) = (h'(k) + i) \bmod m$$



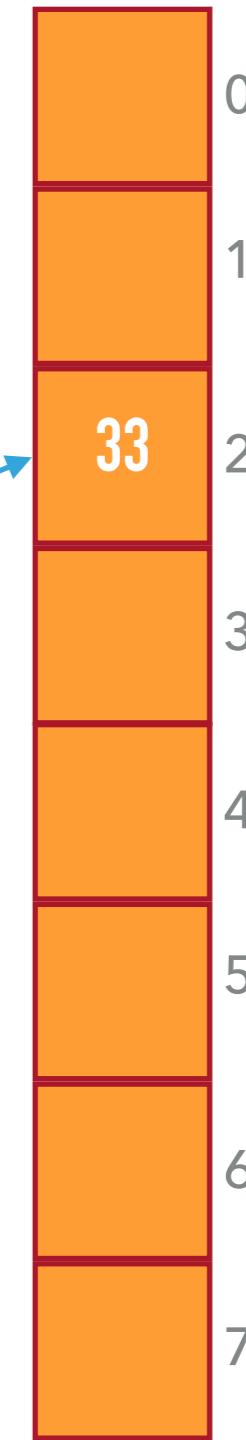
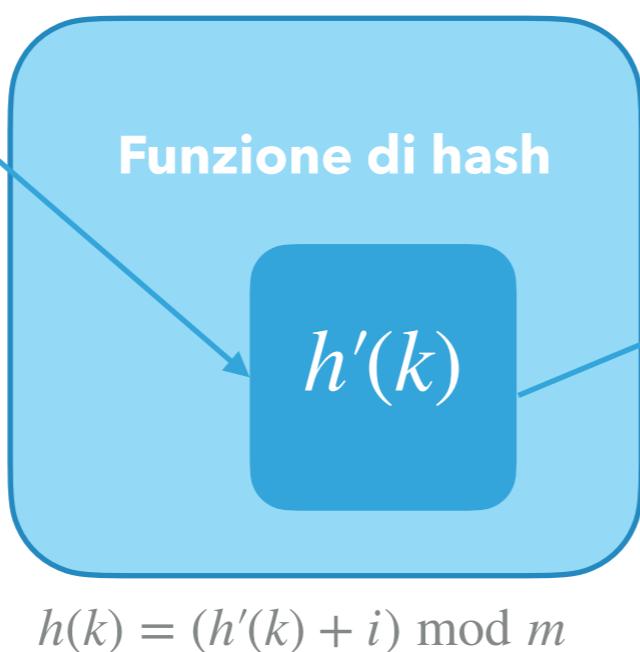
HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash $h'(k)$

Inseriamo 33

Inseriamo 47

Inseriamo 12



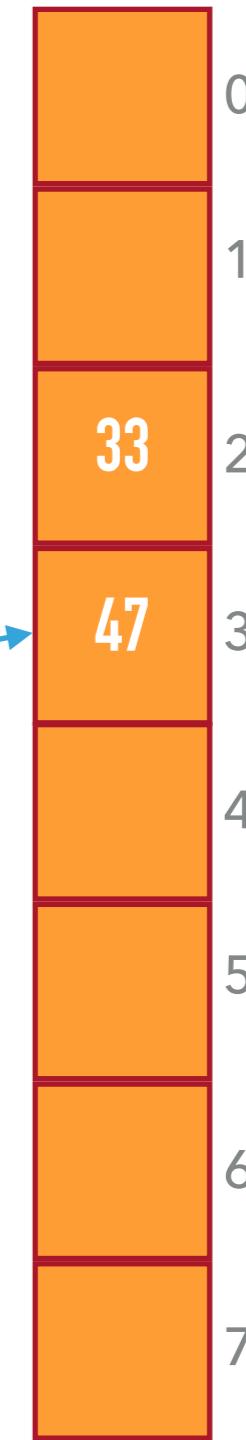
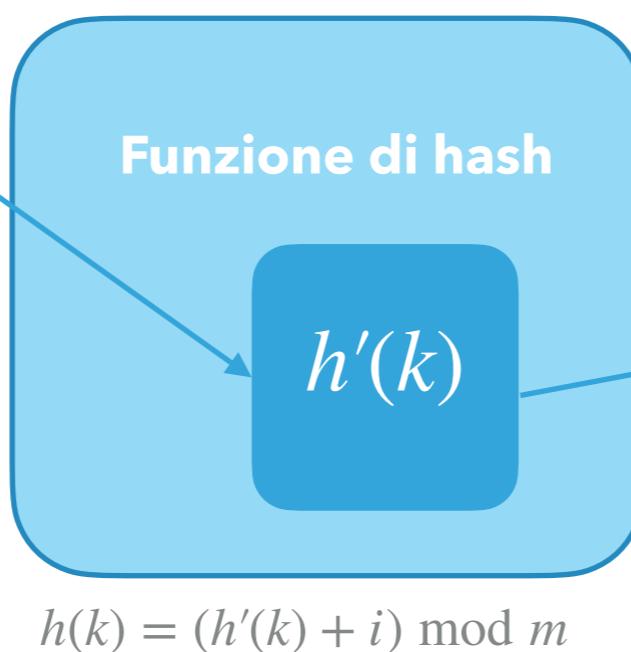
HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash $h'(k)$

Inseriamo 33

Inseriamo 47

Inseriamo 12



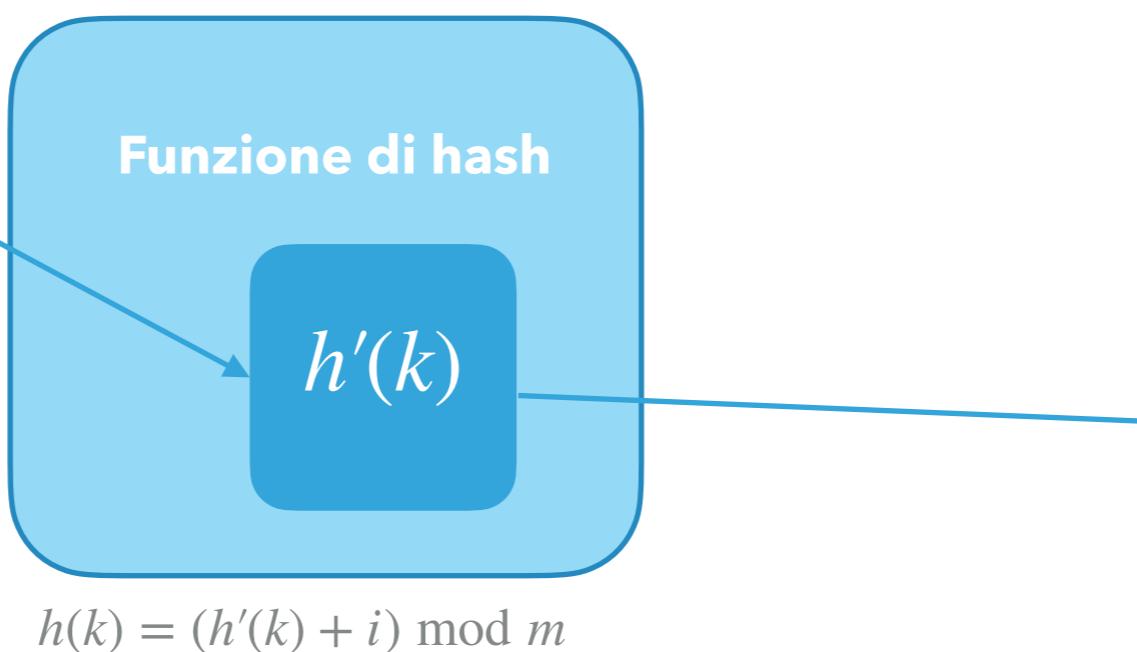
HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash $h'(k)$

Inseriamo 33

Inseriamo 47

Inseriamo 12



HASH CON LINEAR PROBING

Supponiamo che i valori 33, 47 e 12 abbiano lo stesso hash $h'(k)$

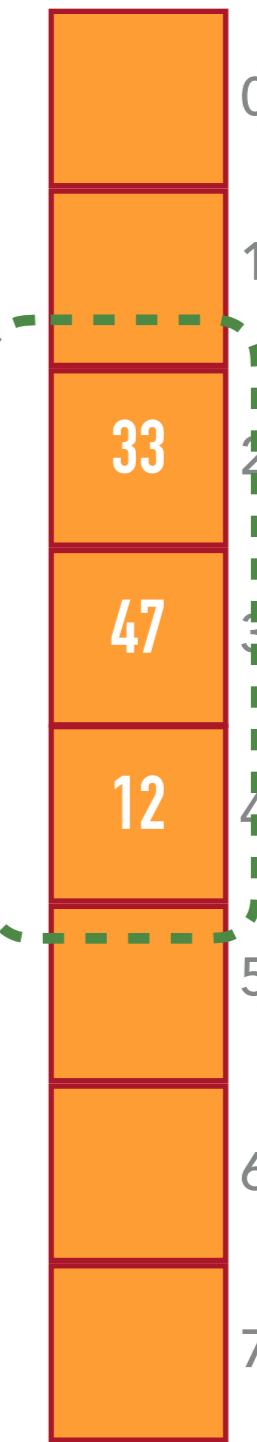
Otteniamo un raggruppamento di valori consecutivi

Questo si chiama “primary clustering” o “**clustering primario**”

Perché è un problema?

La posizione 5 ora raccoglie tutti i valori con hash 2, 3, 4 e 5

In generale una posizione preceduta da i posizioni occupate
sarà scelta per inserire il prossimo valore con probabilità $\frac{i+1}{m}$



QUADRATIC PROBING

- ▶ Data una funzione di hash $h' : U \rightarrow \{0, \dots, m - 1\}$ la modifichiamo definendo una funzione h come:
$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$
- ▶ Dobbiamo scegliere attentamente le costanti c_1 e c_2 in modo di visitare tutte le posizioni
- ▶ Alcuni esempi: se m è una potenza di 2 possiamo scegliere $c_1 = c_2 = 1/2$, questo genererà la sequenza $h'(k), h'(k) + 1, h'(k) + 3, h'(k) + 6, \dots$

QUADRATIC PROBING

- ▶ Non otteniamo lo stesso problema del clustering primario come con il probing lineare
- ▶ Otteniamo comunque una forma meno forte di clustering chiamato secondary clustering o **clustering secondario**
- ▶ Perché entrambe le strategie di probing provocano problemi (sebbene diversi) col clustering?

I PROBLEMI DI QUADRATIC E LINEAR PROBING

- ▶ Sia quadratic e linear probing hanno un problema in comune dovuto alla loro costruzione
- ▶ Dati k_1 e k_2 con $h'(k_1) = h'(k_2)$ abbiamo che per ogni i vale $h(k_1, i) = h(k_2, i)$, ovvero hanno uguale sequenza di probing
- ▶ In pratica due chiavi con uguale hash $h'(k)$ testano le stesse posizioni: abbiamo solo m sequenze di probing distinte

DOUBLE HASHING

- ▶ Date **due** funzioni di hash $h_1 : U \rightarrow \{0, \dots, m - 1\}$ e $h_2 : U \rightarrow \{0, \dots, m - 1\}$ definiamo
$$h(k, i) = (h_1(k) + ih_2(k)) \text{ mod } m$$
- ▶ Per poter visitare tutte le posizioni $h_2(k)$ deve essere coprimo rispetto a m . Possiamo assicurarlo in più modi:
 - ▶ $h_2(k)$ è sempre dispari e m una potenza di 2
 - ▶ m primo e $h_2(k)$ ritorna solo valori in positivi minori di m

DOUBLE HASHING

- ▶ Il vantaggio di utilizzare due funzioni di hash è che se anche abbiamo $k_1 \neq k_2$ con $h_1(k_1) = h_1(k_2)$ non è detto che k_1 e k_2 abbiano la stessa sequenza di probing
- ▶ Due chiavi k_1 e k_2 hanno la stessa sequenza di probing solo quando hanno lo stesso hash sia con h_1 che con h_2
- ▶ Abbiamo quindi m^2 diverse sequenze di probing (una per ogni coppia di valori $(h_1(k), h_2(k))$ possibile)

HASH CON DOUBLE HASHING

Inseriamo 9

Inseriamo 23

Inseriamo 16



$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$

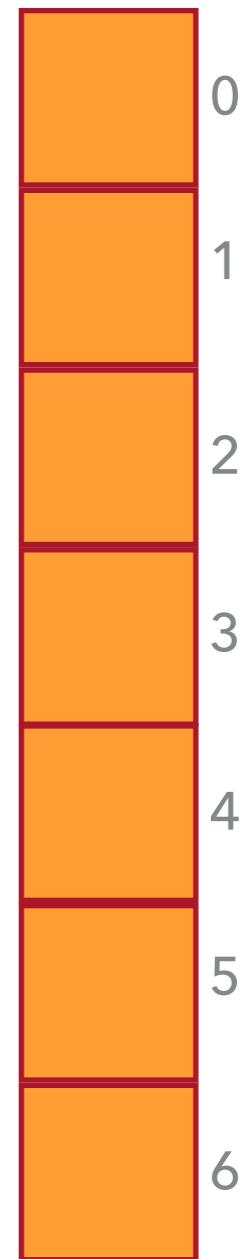
$$h_1(k) = k \bmod 7$$

$$h_2(k) = 1 + (k \bmod 6)$$

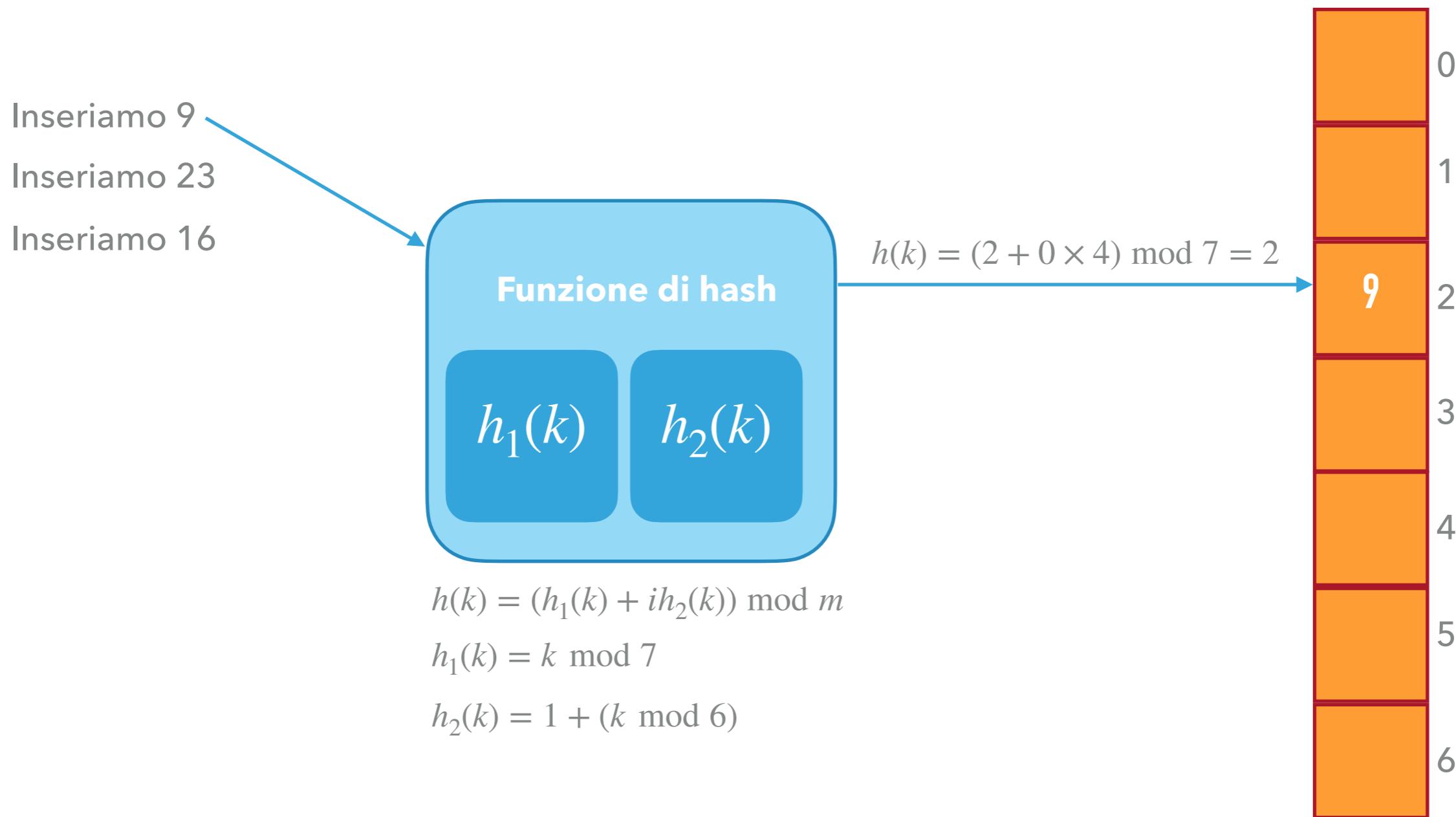
h_2 ritorna solo valori positivi minori di m .

In generale funzione prendendo

$$h_2(k) = 1 + k \bmod (m - 1)$$



HASH CON DOUBLE HASHING



HASH CON DOUBLE HASHING

Inseriamo 9

Inseriamo 23

Inseriamo 16

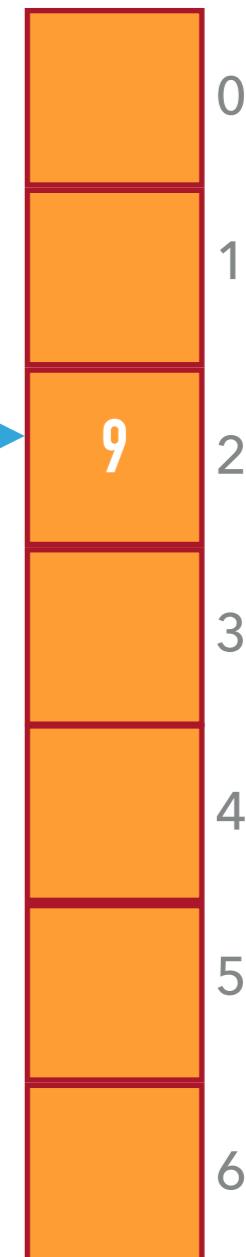


$$h(k) = (2 + 0 \times 6) \bmod 7 = 2$$

$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$

$$h_1(k) = k \bmod 7$$

$$h_2(k) = 1 + (k \bmod 6)$$

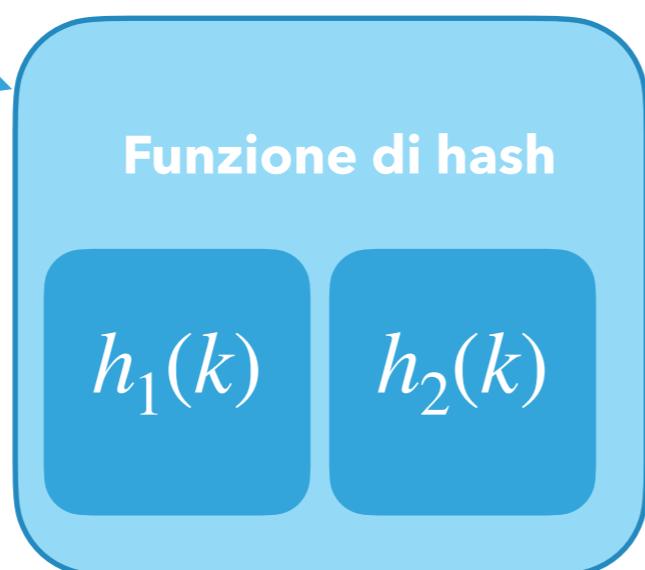


HASH CON DOUBLE HASHING

Inseriamo 9

Inseriamo 23

Inseriamo 16

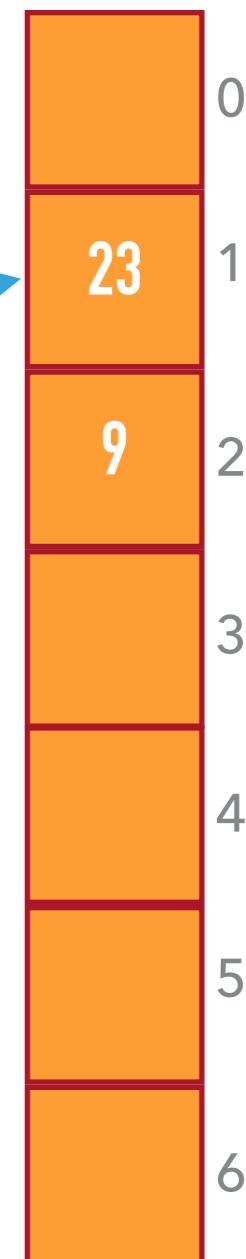


$$h(k) = (2 + 1 \times 6) \bmod 7 = 1$$

$$h(k) = (h_1(k) + ih_2(k)) \bmod m$$

$$h_1(k) = k \bmod 7$$

$$h_2(k) = 1 + (k \bmod 6)$$

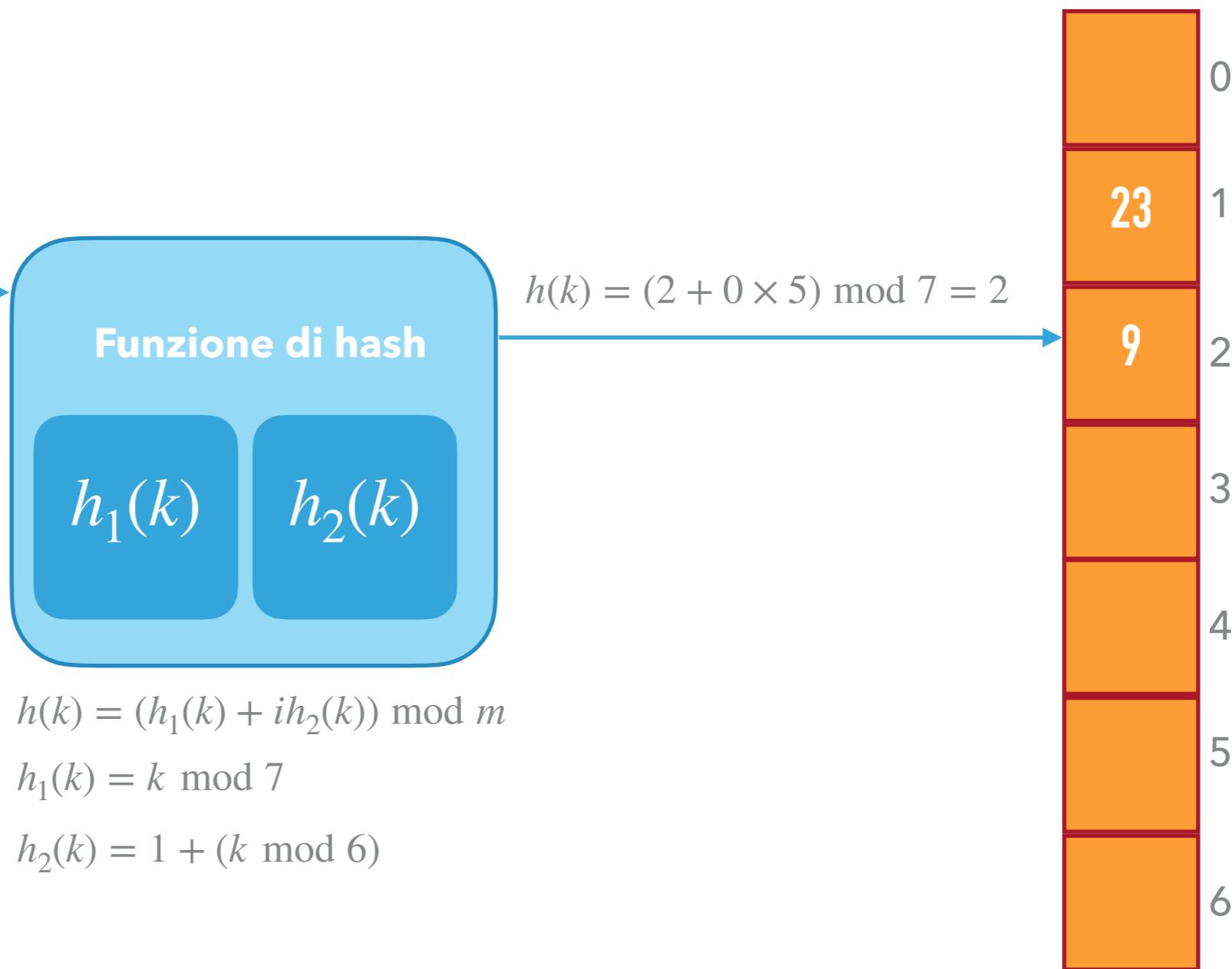


HASH CON DOUBLE HASHING

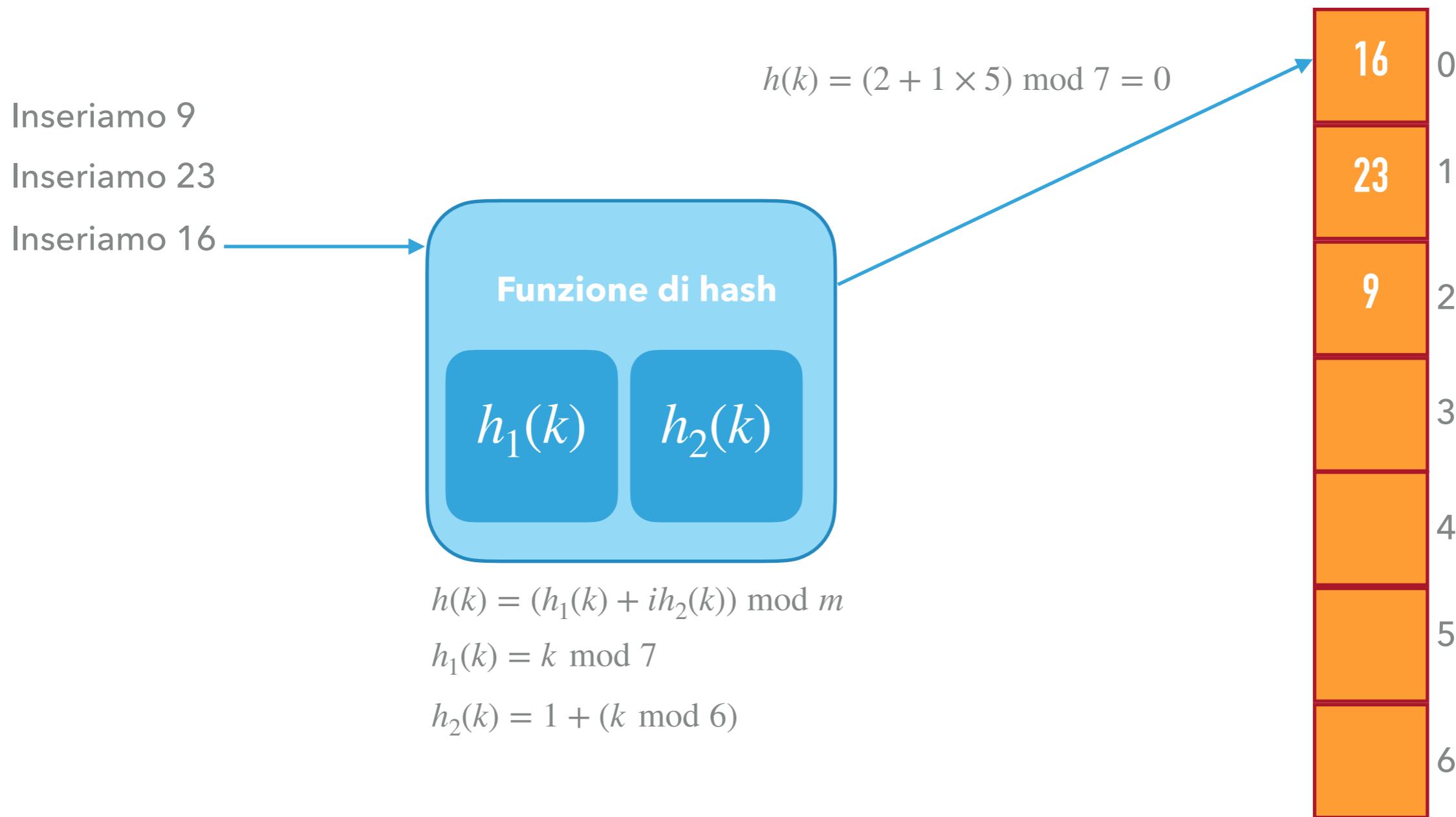
Inseriamo 9

Inseriamo 23

Inseriamo 16



HASH CON DOUBLE HASHING



ANALISI DEL TEMPO

- ▶ Dobbiamo ora studiare il tempo che ci impieghiamo a inserire e cercare un elemento all'interno di una tabella hash
- ▶ Assumiamo uniform hashing (o hashing uniforme):
 - ▶ La sequenza di probing di ogni chiave è con uguale probabilità una qualsiasi delle $m!$ permutazioni di $\{0,1,\dots,m - 1\}$
 - ▶ Il vero hashing uniforme è difficile da implementare, di solito ci accontentiamo di una sua approssimazione

ANALISI DEL TEMPO

- ▶ Sotto l'assunzione di hashing uniforme
- ▶ Se assumiamo di mantenere il fattore di carico $\alpha = n/m$ limitato da una costante minore di 1 (e.g., $\alpha \leq 0.5$, tabella al più piena per metà)
- ▶ Allora sia inserimento che ricerca richiedono, in media, tempo $O(1)$

RICERCA CHE NON HA SUCCESSO

- ▶ Assumendo uniform hashing, fattore di carico $\alpha < 1$, la ricerca di una chiave k non presente richiede di esplorare in media al più $\frac{1}{1 - \alpha}$ posizioni

Poiché la chiave k non è presente, tutte le posizioni visitate tranne l'ultima saranno occupate.

Indichiamo con A_1, A_2, \dots, A_m gli eventi “l' i -esima posizione nella sequenza di probing era occupata”

RICERCA CHE NON HA SUCCESSO

La probabilità di dover visitare *almeno i* posizioni è data da $P(A_1, A_2, \dots, A_{i-1})$ ovvero la probabilità **congiunta** che le prime $i - 1$ posizioni siano tutte occupate

Usando il fatto che $P(A, B) = P(A | B)P(B)$ possiamo riscrivere come:

$$P(A_1, A_2, \dots, A_{i-1}) = P(A_1)P(A_2 | A_1) \dots P(A_{i-1} | A_1, \dots, A_{i-2})$$

Queste sono probabilità che possiamo stimare

RICERCA CHE NON HA SUCCESSO

$P(A_1) = n/m$ ovvero il fattore di carico. Ovvero la probabilità di trovare uno slot libero su m dato che n sono occupati

$$P(A_j | A_1, \dots, A_{j-1}) = \frac{n - (j - 1)}{m - (j - 1)}$$

perché se abbiamo trovato $j - 1$ posizioni occupate, abbiamo la probabilità di trovarne una occupata scegliendo tra le $m - (j - 1)$ rimanenti di cui $n - (j - 1)$ sono occupate.

In questo punto stiamo sfruttando l'assunzione di hashing uniforme per dire che scegliamo in modo uniforme tra tutte le posizioni rimanenti

RICERCA CHE NON HA SUCCESSO

Osserviamo che $\frac{n - (j - 1)}{m - (j - 1)} \leq \frac{n}{m}$ per poter mostrare che

$$\begin{aligned} P(A_1, \dots, A_{i-1}) &= \frac{n}{m} \frac{n-1}{m-1} \cdots \frac{n-i+2}{m-i+2} \\ &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} \end{aligned}$$

Quindi la probabilità di dover guardare almeno i posizioni non è più di α^{i-1}

RICERCA CHE NON HA SUCCESSO

Possiamo esprimere il numero atteso di posizioni da visitare $E[X]$ come la probabilità di dover visitare una, due, etc. posizioni:

$$E[X] \leq \sum_{i=1}^{+\infty} \alpha^{i-1} = \sum_{i=0}^{+\infty} \alpha^i = \frac{1}{1 - \alpha}$$

Questo ci mostra che una ricerca senza successo e anche un inserimento (che richiede di trovare la prima posizione libera) richiedono in media di visitare al più $\frac{1}{1 - \alpha}$ posizioni

RICERCA CHE HA SUCCESSO

- ▶ Assumendo uniform hashing, fattore di carico $\alpha < 1$ e che tutte le chiavi presenti abbiano uguale probabilità di essere cercate, la ricerca di una chiave k presente richiede di esplorare in media $\frac{1}{\alpha} \log_e \frac{1}{1 - \alpha}$ posizioni

La ricerca di una chiave k richiede di esplorare un numero di posizioni che dipende da quante chiavi sono state inserite in precedenza.

RICERCA CHE HA SUCCESSO

Se k è stata la $(i + 1)$ -esima chiave inserita, sfruttiamo il risultato precedente per dire che in media abbiamo visitato al più $\frac{1}{1 - (i/m)} = \frac{1}{(m - i)/m} = \frac{m}{m - i}$ posizioni per inserirla

Facciamo la media su tutti i possibili valori di i , ovvero k potrebbe essere stata con uguale probabilità la prima, seconda, terza, etc. chiave inserita.

Questo ci fornirà un bound sul numero atteso di posizioni da cercare

RICERCA CHE HA SUCCESSO

$$\frac{1}{n} \sum_{i=1}^n \frac{m}{m-i} = \frac{m}{n} \sum_{i=1}^n \frac{1}{m-i} = \frac{1}{\alpha} \sum_{i=1}^n \frac{1}{m-i}$$

Ora cambiamo l'indice della somma

$$\frac{1}{\alpha} \sum_{p=m-n+1}^m \frac{1}{p} \leq \frac{1}{\alpha} \int_{m-n}^m \frac{1}{p} dp$$

Calcoliamo il valore dell'integrale

$$\frac{1}{\alpha} \log_e \frac{m}{m-n} = \frac{1}{\alpha} \log_e \frac{(1/m)m}{(1/m)(m-n)} = \frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$$

Quindi che in media dobbiamo visitare $\frac{1}{\alpha} \log_e \frac{1}{1-\alpha}$ posizioni

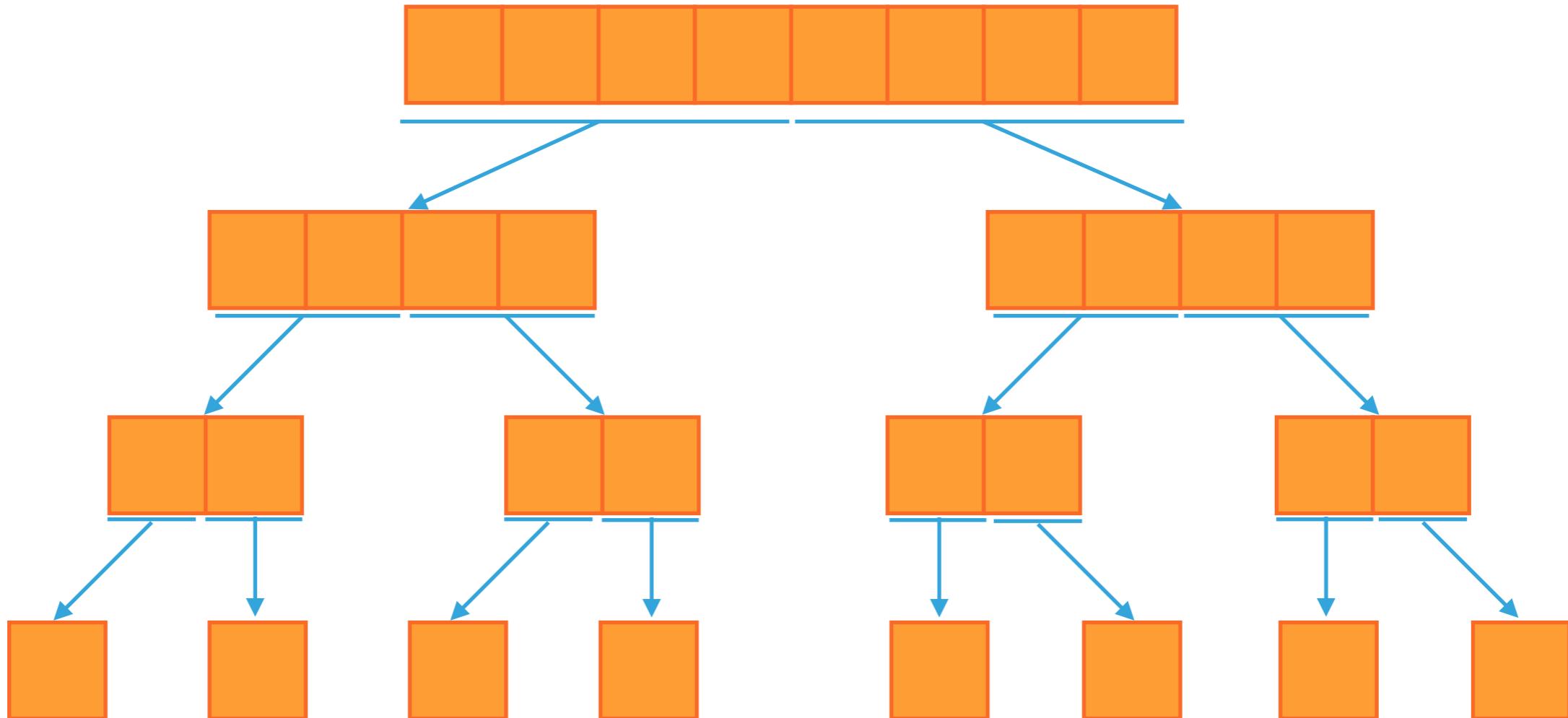
PROGRAMMAZIONE DINAMICA
PIÙ LUNGA SOTTOSEQUENZA COMUNE (LCS)

ALGORITMI E STRUTTURE DATI

DIVIDE ET IMPERA

- ▶ Metodo comune di risoluzione dei problemi
- ▶ Si basa sull'idea che possiamo esprimere una soluzione come combinazione di soluzioni di sotto-problemi più piccoli
- ▶ Un esempio standard: mergesort
- ▶ Proviamo a esplorare meglio la struttura dei sotto-problemi che andiamo a risolvere

MERGESORT



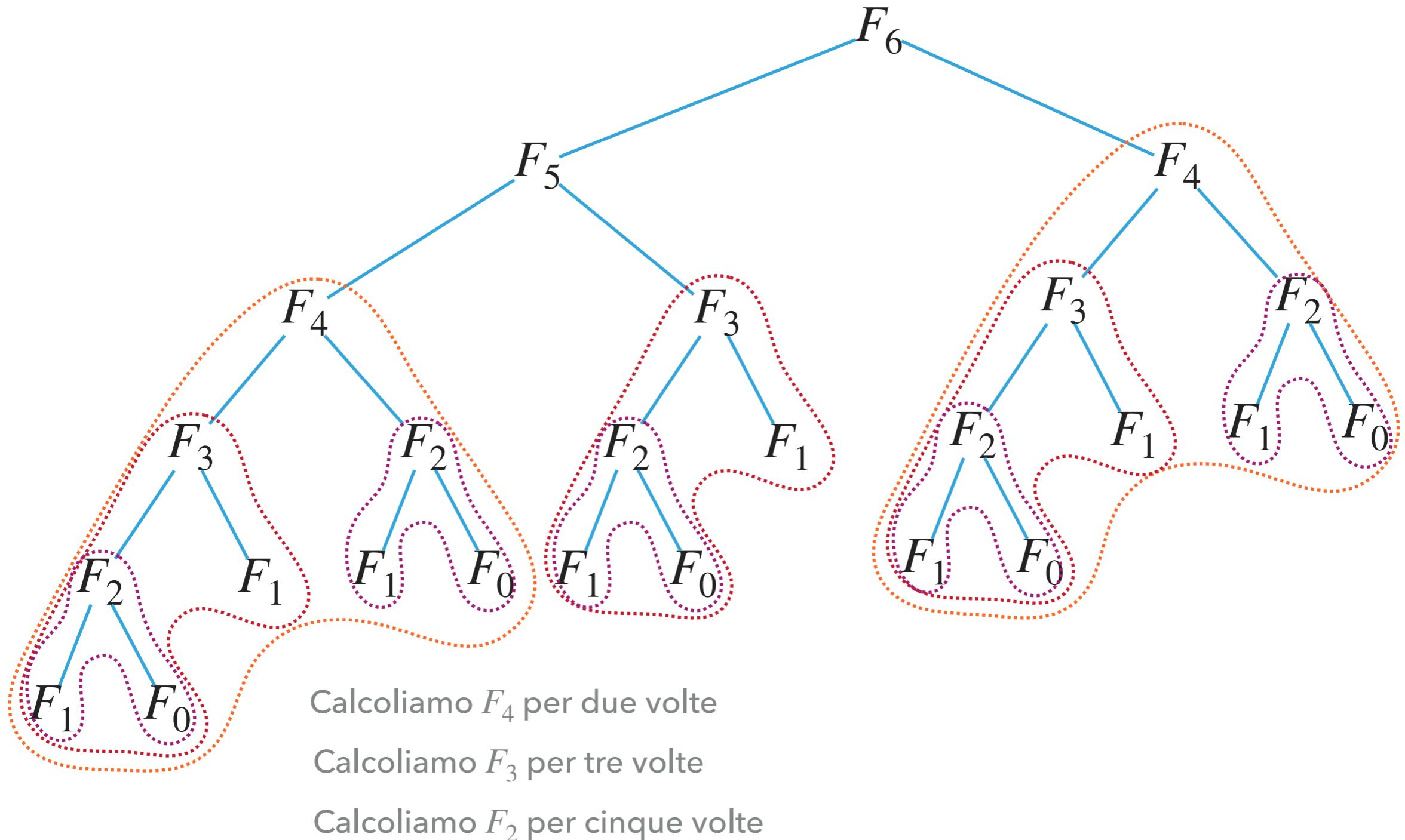
La dimensione dei sotto-array da ordinare si dimezza ma, ancora più importante, ciascuno dei sottoproblemi è indipendente!

E.g., ordinare la prima metà dell'array non è influenzato da ordinare la seconda metà.

DIVIDE ET IMPERA: LIMITAZIONI

- ▶ Ma cosa succede quando i sotto-problemi che dobbiamo risolvere non sono indipendenti?
- ▶ Un esempio tipico è il calcolo diretto dei numeri di Fibonacci:
 - ▶ $F_n = F_{n-1} + F_{n-2}$, ma a sua volta la risoluzione di F_{n-1} richiede la risoluzione di F_{n-2} , dato che
$$F_{n-1} = F_{n-2} + F_{n-3}$$
 - ▶ Vediamo l'albero dei sotto-problemi

FIBONACCI: SOTTO-PROBLEMI RIPETUTI



APPROCCIARE I SOTTO-PROBLEMI RIPETUTI

- ▶ Quando abbiamo dei sotto-problemi che si ripetono un semplice approccio ricorsivo immediato non è generalmente efficiente
- ▶ Nel caso dei numeri di Fibonacci più il sotto-problema è piccolo e più volte lo risolviamo.
- ▶ Questo numero cresce come il numero di Fibonacci, quindi esponenzialmente rispetto a n

APPROCCIARE I SOTTO-PROBLEMI RIPETUTI

- ▶ Se abbiamo un sotto-problema ripetuto è inutile risolverlo più volte, possiamo salvarci il risultato e riusarla quando ci serve
- ▶ Questa è l'idea di base della **programmazione dinamica** e della **memoizzazione** (no, non è un errore di battitura)
- ▶ La principale differenza tra questi due approcci è data da come andiamo a costruire la soluzione finale: in un approccio bottom-up o top-down

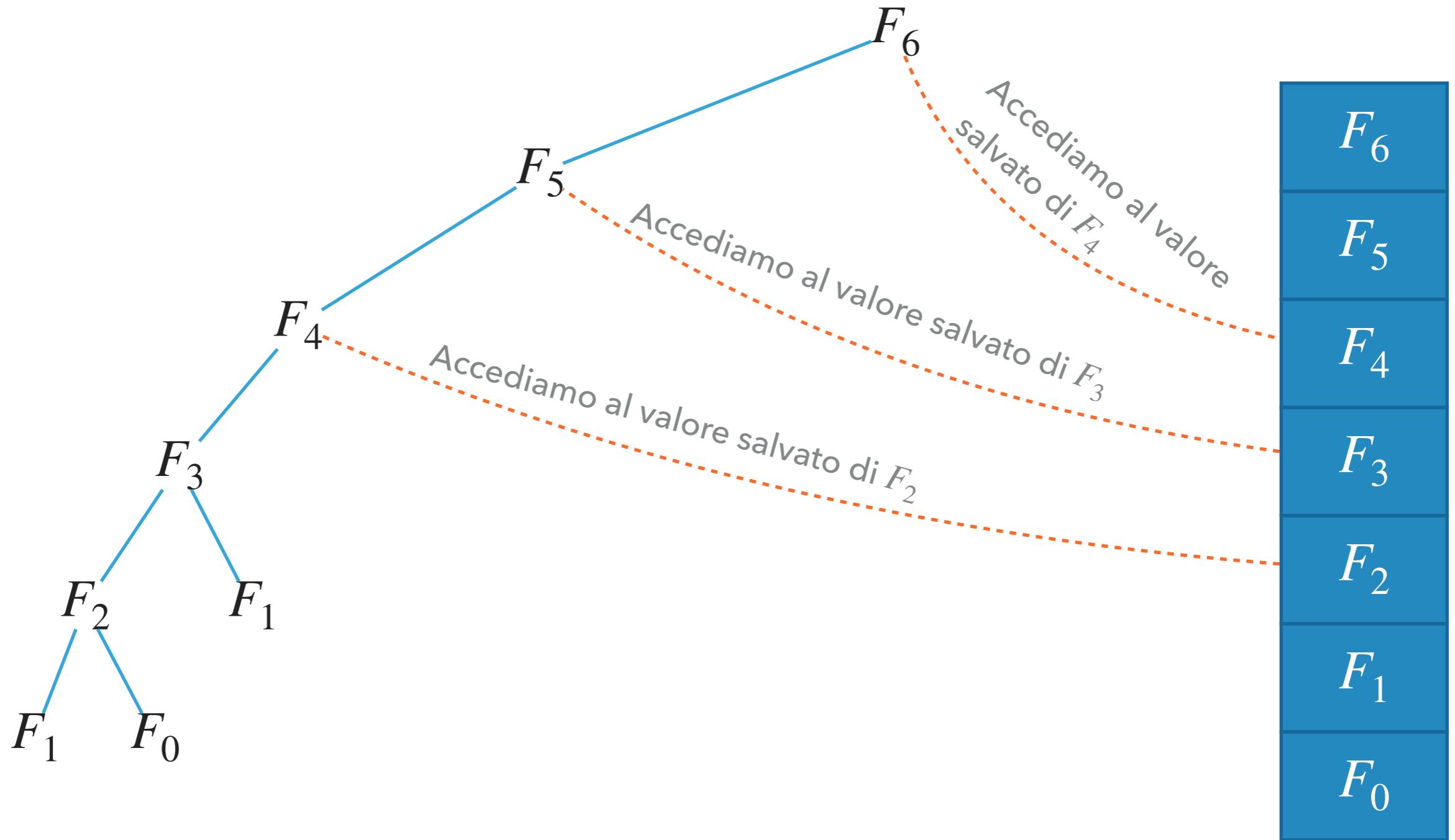
TOP-DOWN E BOTTOM-UP

- ▶ Un approccio **top-down** è quello di scomporre il problema in sotto-problemi più piccoli in modo ricorsivo
- ▶ Per Fibonacci: il normale approccio di scrivere ricorsivamente
$$F_n = F_{n-1} + F_{n-2}$$
- ▶ L'approccio **bottom-up** è quello di partire dai casi base e combinarli ripetutamente fino a quando non si ottiene il risultato atteso
- ▶ Per Fibonacci: per calcolare F_n partiamo da F_0 e F_1 e calcoliamo, in ordine F_2, F_3, \dots, F_n

MEMOIZZAZIONE

- ▶ L'idea è quella di tenere una tabella di sotto-problemi che abbiamo già risolto e, prima di fare una chiamata ricorsiva, verificare se abbiamo già la soluzione:
 - ▶ Se la soluzione è presente nella tabella usiamo quella senza fare chiamate ricorsive
 - ▶ Altrimenti facciamo la normale chiamata ricorsiva e, in aggiunta, salviamo il risultato ottenuto nella tabella
- ▶ In questo modo risolviamo ogni sotto-problema una volta sola

FIBONACCI: SOTTO-PROBLEMI RIPETUTI



IL TERMINE “PROGRAMMAZIONE DINAMICA”

- ▶ Ideata da Richard Bellman negli anni '50
(lo stesso dell'algoritmo di Bellman-Ford)
- ▶ Il nome “programmazione dinamica” non è molto informativo:
 - ▶ il termine programmazione è da intendersi nel senso di “pianificazione”
 - ▶ “Dinamica” è stato scelto, tra gli altri motivi, perché “it's impossible to use the word dynamic in a pejorative sense”

IDEA DI BASE

- ▶ Come idea di base della programmazione dinamica, pensiamo a definire una ricorrenza che lega soluzioni del problema a soluzioni di sotto-problemi
- ▶ Costruiamo una tabella di soluzioni
- ▶ Inseriamo nella tabella le soluzioni dei casi base
- ▶ Riempiamo iterativamente la tabella fino a quando non abbiamo ottenuto la soluzione al problema di partenza

UN PROBLEMA D'ESEMPIO

- ▶ “problema del taglio della barra” o “rod cutting problem”
- ▶ Abbiamo una barra di metallo di lunghezza n che possiamo tagliare in pezzi di dimensione intera: $1, 2, \dots, n$
- ▶ Un pezzo di lunghezza i viene venduto al prezzo p_i
- ▶ Vogliamo trovare un algoritmo che ci dica il modo migliore di tagliare la barra per massimizzare il prezzo di vendita totale

SOTTO-PROBLEMI RIPETUTI



Barra da tagliare, lunghezza $n = 6$



$$4.5 \times 3 = 13.5$$



$$16.7$$



$$12.6 + 4.5 + 2.6 = 19.7$$

Possiamo definire questo problema in modo ricorsivo?

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

UN PROBLEMA D'ESEMPIO

- ▶ Vogliamo massimizzare la somma dei prezzi di vendita dei singoli tagli
- ▶ Casi conosciuti: nessun taglio, barra di lunghezza $1, \dots, n$
- ▶ Data una sbarra di lunghezza n , indichiamo con r_n il miglior prezzo di vendita totale
- ▶ Proviamo ad esprimere in modo ricorsivo r_n

UN PROBLEMA D'ESEMPIO

- ▶ Data una barra di lunghezza n abbiamo le seguenti possibilità:
 - ▶ La vendiamo intera, ottenendo p_n
 - ▶ Facciamo il primo taglio di lunghezza $k < n$, ottenendo r_k per il pezzo tagliato e r_{n-k} per la parte rimanente
 - ▶ Quindi per trovare r_n calcoliamo
$$\max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots\}$$

UN PROBLEMA D'ESEMPIO

- ▶ $r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots\}$ ci mostra che il problema può essere ricondotto alla risoluzione di sottoproblemi ottimi più piccoli!
- ▶ Ora il problema è quello di calcolare r_n in modo efficiente
- ▶ Proviamo con una semplice implementazione ricorsiva

PSEUDOCODICE: TAGLIO DELLA BARRA

Parametri: lunghezza della barra n, tabella dei prezzi p

```
if n == 1: # non possiamo dividere ulteriormente la barra
```

```
    return p[0]
```

```
prezzo_vendita = p[n-1] # la nostra stima iniziale è senza tagli
```

```
for i in range(1, n): # per ogni possibile posizione di taglio
```

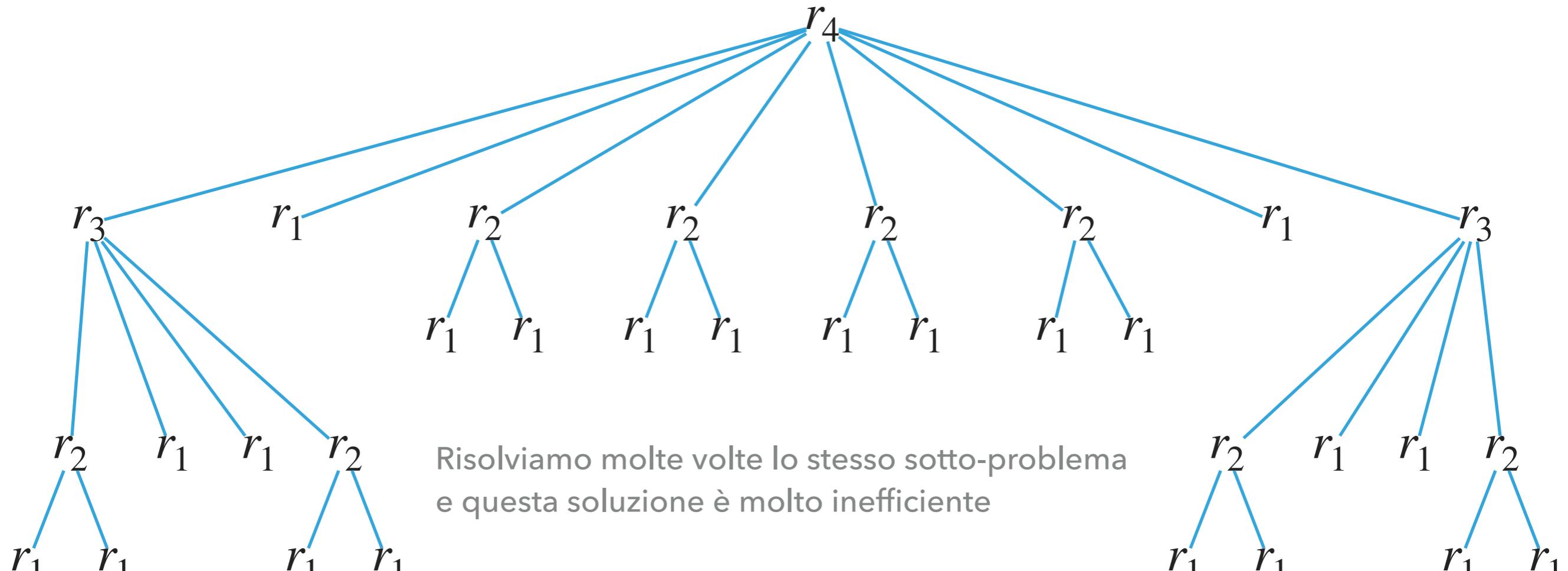
```
    tmp = taglio_barra(i, p) + taglio_barra(n - i, p) # chiamata ricorsiva
```

```
    if prezzo_vendita < tmp: # se abbiamo migliorato la stima la aggiorniamo
```

```
        prezzo_vendita = tmp
```

```
return prezzo_vendita
```

TAGLIO DELLA BARRA: SOTTO-PROBLEMI RIPETUTI



Una stima un poco grezza del tempo di calcolo:

$$T(n) = 2 \left(\sum_{i=1}^{n-1} T(i) \right) \geq 2T(n-1) = O(2^n)$$

UN PROBLEMA D'ESEMPIO: MIGLIORARE LA SOLUZIONE

- ▶ Dobbiamo trovare un modo più efficiente di risolvere il problema
- ▶ Applichiamo un approccio bottom-up per calcolare r_n
- ▶ Teniamo un array di lunghezza n che salva in posizione $i - 1$ il valore r_i ed iniziamo a riempire l'array da $r_1 = p_1$
- ▶ Per le posizioni successive usiamo la definizione:

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots\}$$

PSEUDOCODICE: TAGLIO DELLA BARRA

Parametri: lunghezza della barra n , tabella dei prezzi p

```
r = [0] * n
for i in range(0, n-1): # calcoliamo i valori di  $r_i$  a partire dal minore
    r[i] = p[i] # stima iniziale del valore di r[i]
    for j in range(0, i): # proviamo a vedere se potevamo fare un taglio
        if r[j] + r[i-j-1] > r[i]: # se il taglio migliora la situazione
            r[i] = r[j] + r[i-j-1] # aggiorniamo la nostra stima di  $r_{i-1}$ 
return r[n-1] # il valore per  $r_n$  si troverà nell'ultima posizione dell'array
```

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

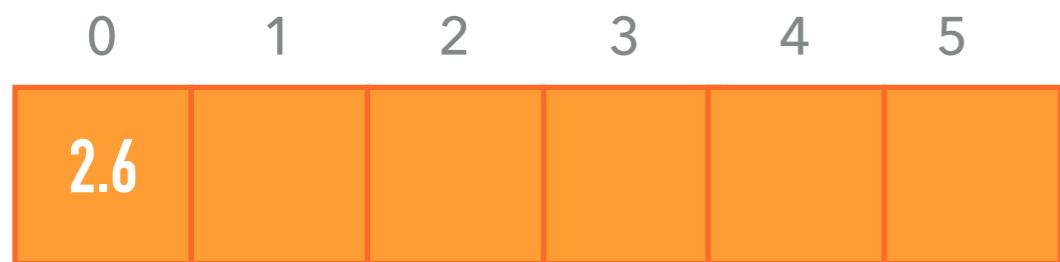


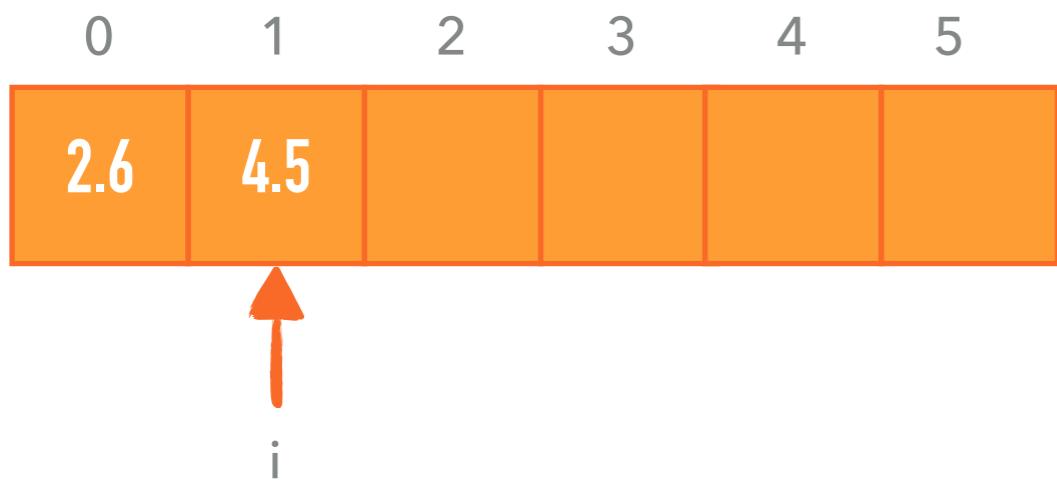
Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$



La nostra stima iniziale è
semplicemente caso senza tagli

Tabella dei prezzi

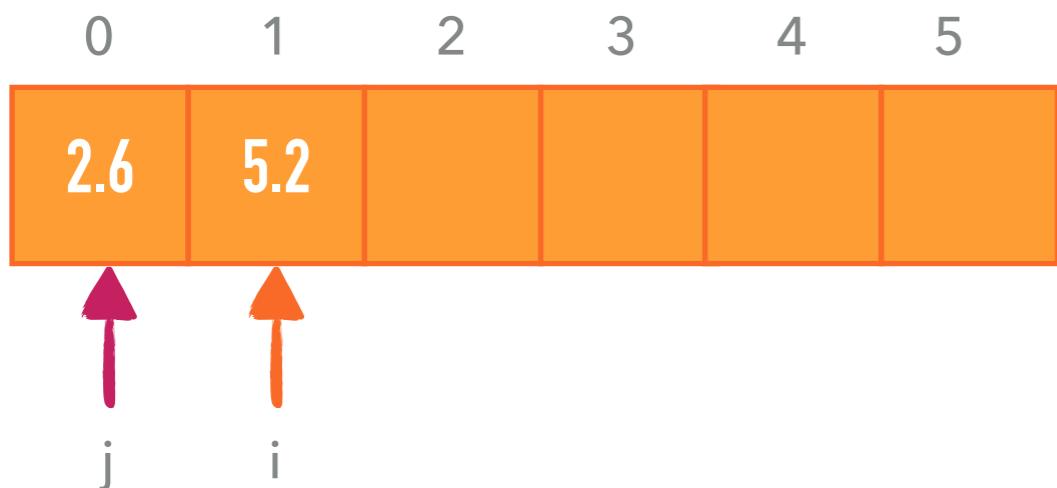
| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$2.6 + 2.6 > 4.5$$



Facciamo variare j aggiornando la nostra stima

Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

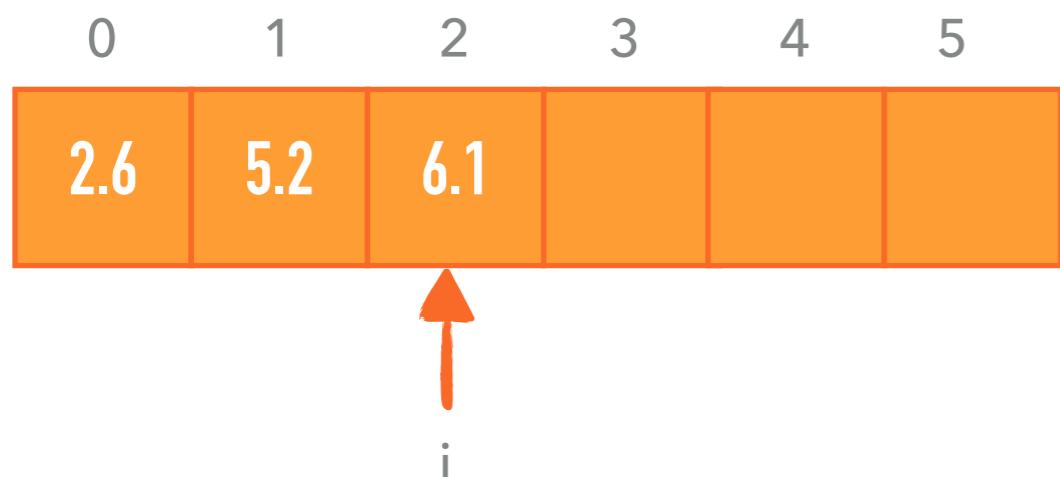


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$2.6 + 5.2 > 6.1$

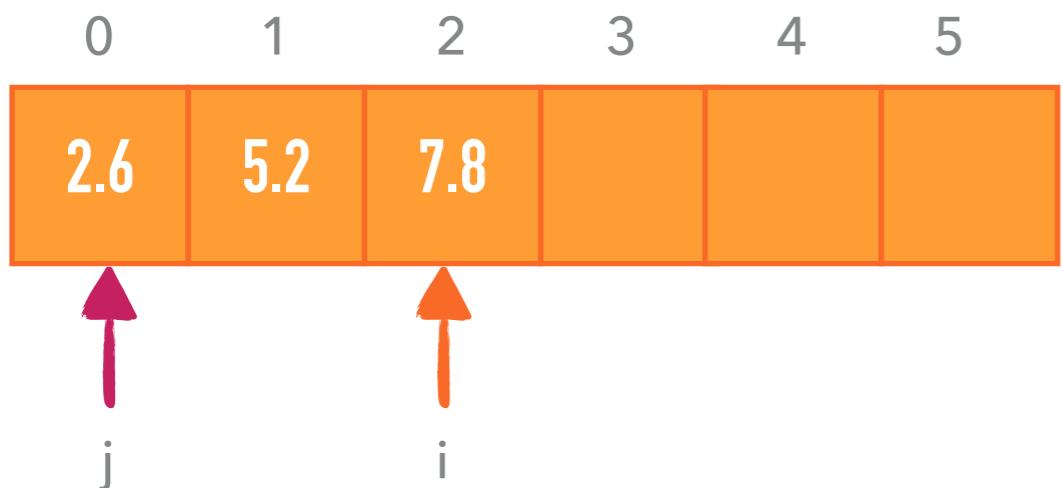


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$2.6 + 5.2 = 7.8$$

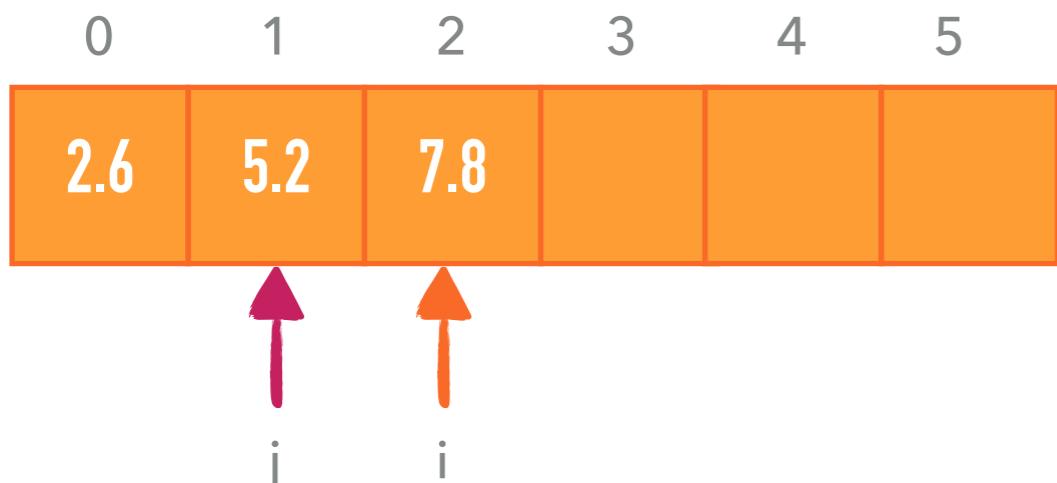


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

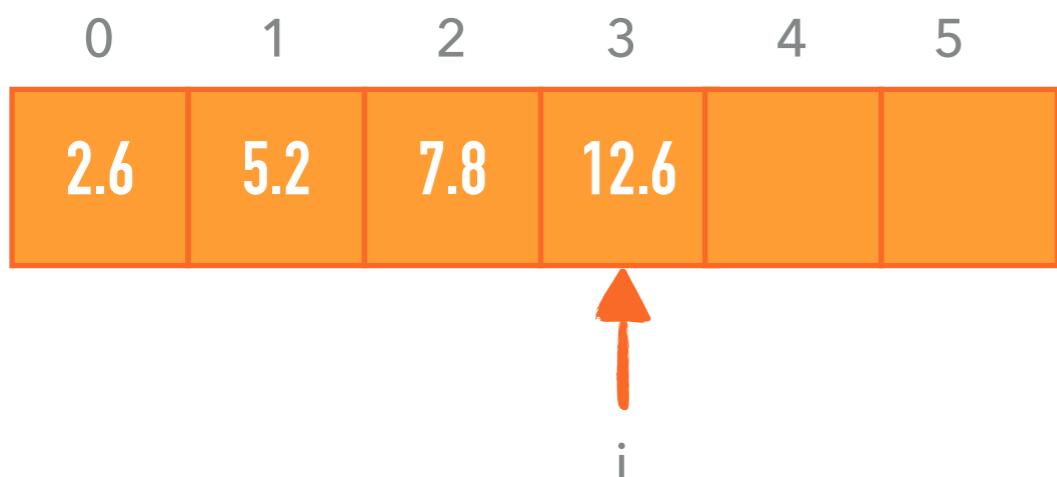


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$2.6 + 7.8 < 12.6$$

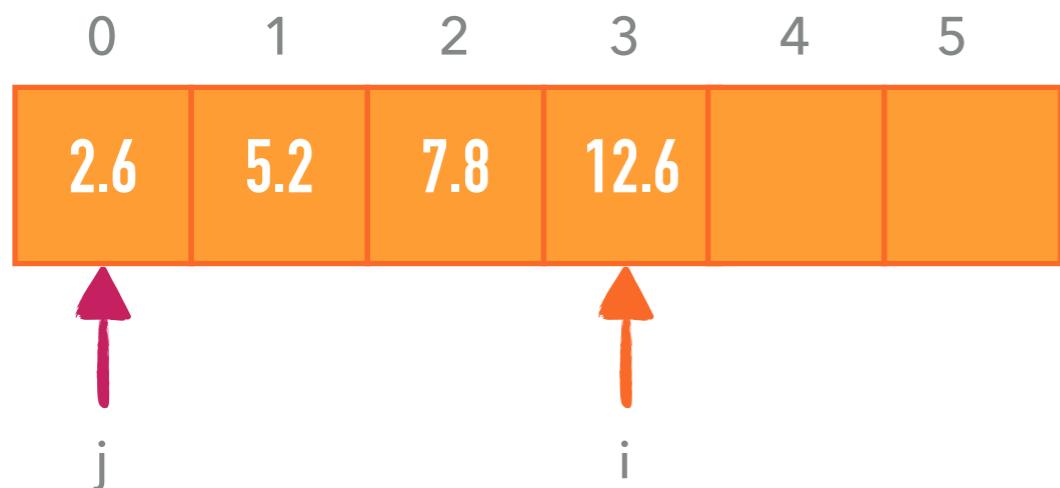


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$5.2 + 5.2 < 12.6$$

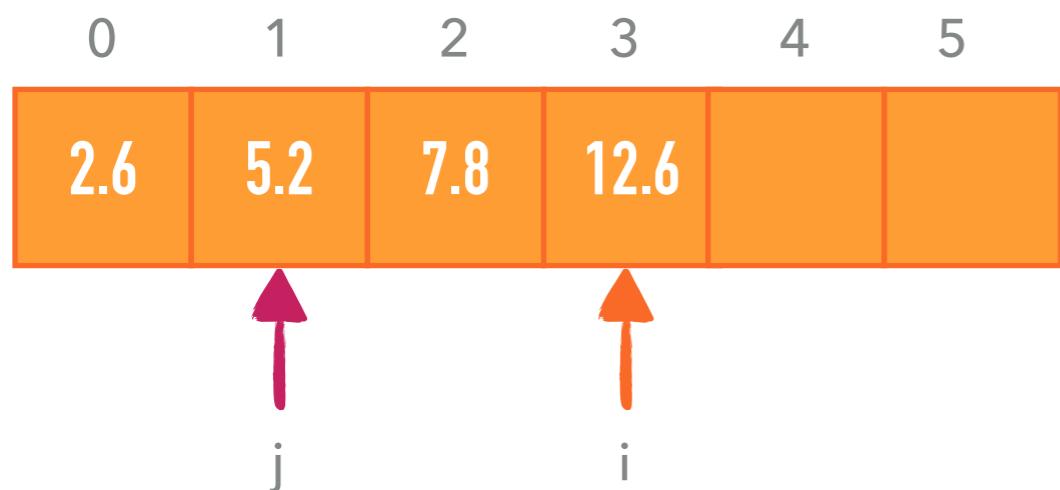


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$2.6 + 7.8 < 12.6$$

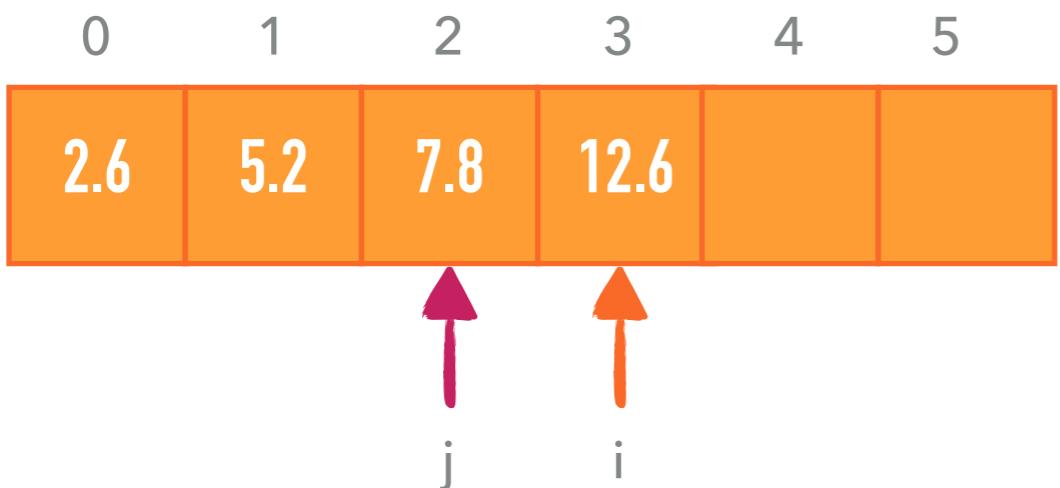


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

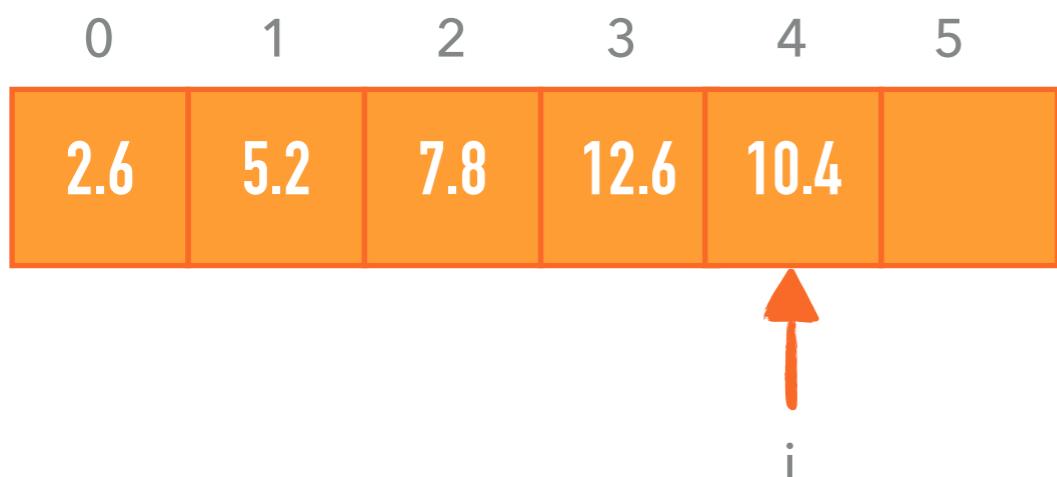


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$12.6 + 2.6 > 10.4$$

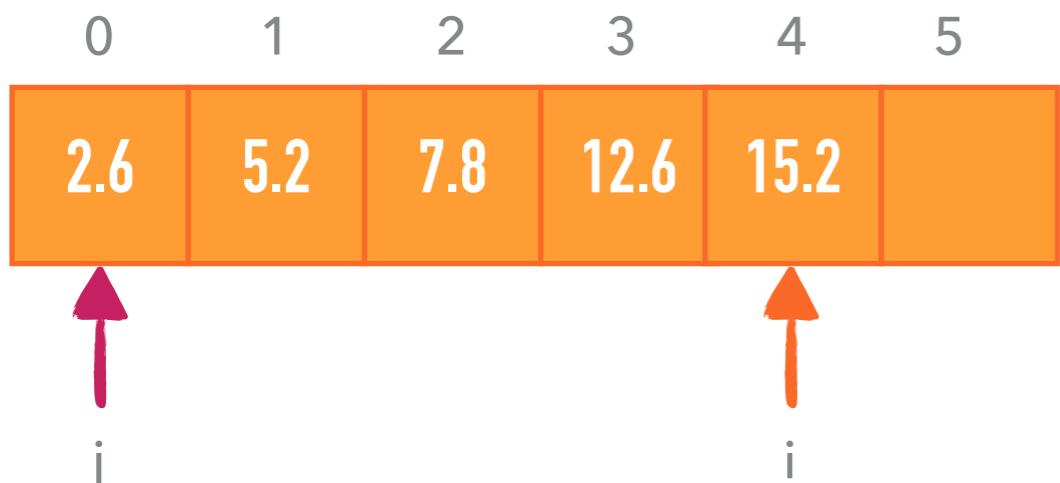


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$5.2 + 7.8 < 15.2$$

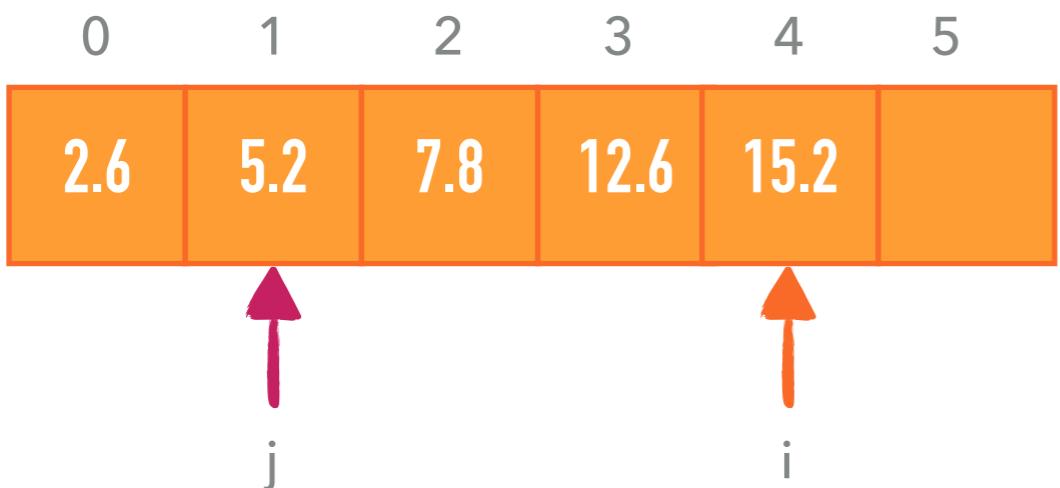


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$5.2 + 7.8 < 15.2$$

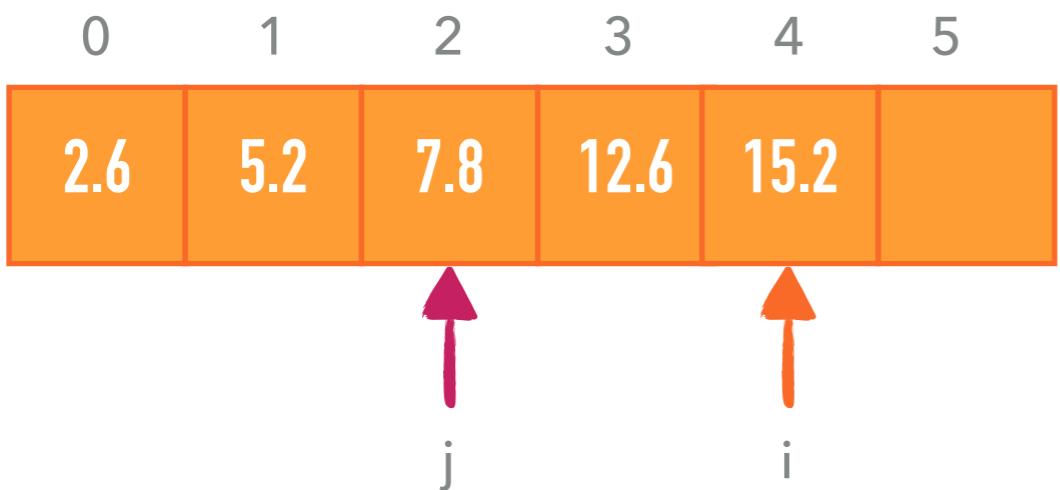


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$12.6 + 2.6 == 15.2$$

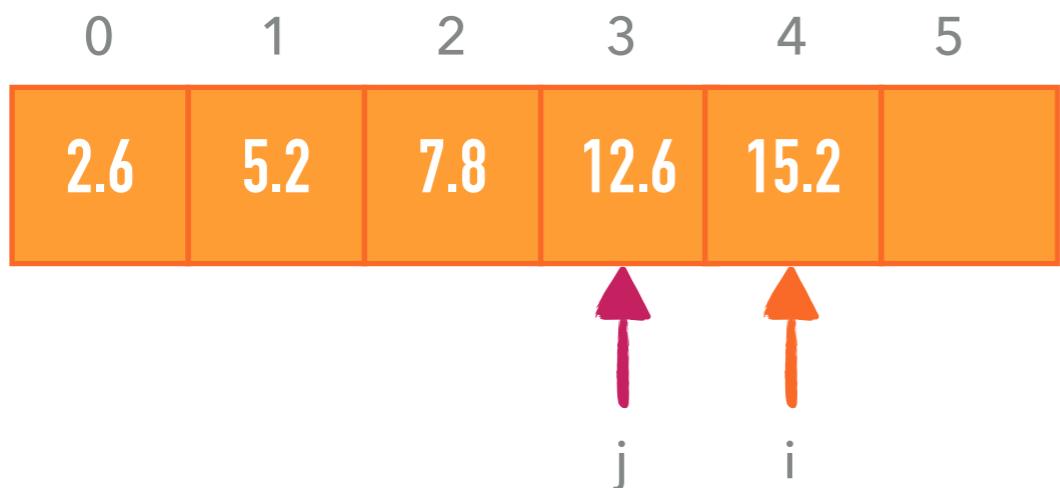


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$



Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

15.2+2.6>16.7



Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$5.2 + 12.6 == 17.8$$

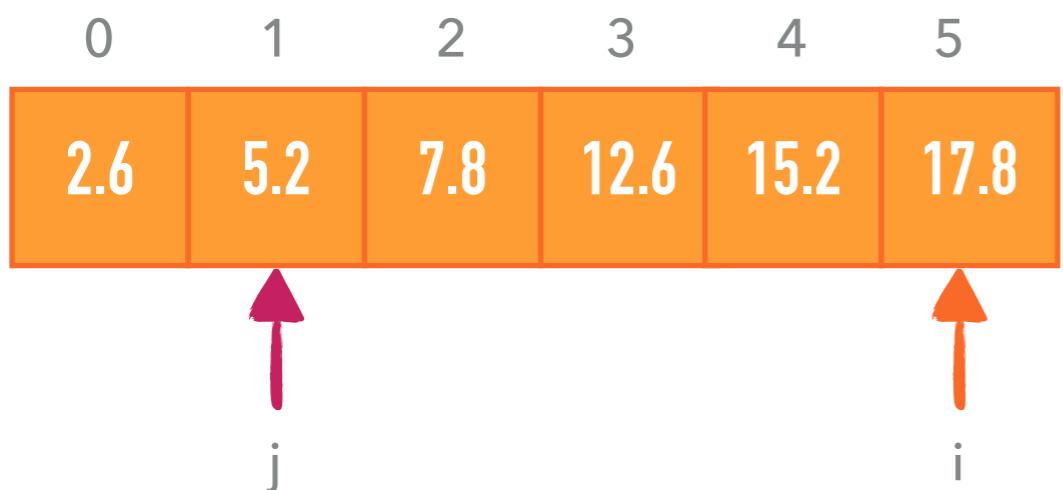


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$7.8 + 7.8 < 17.8$$

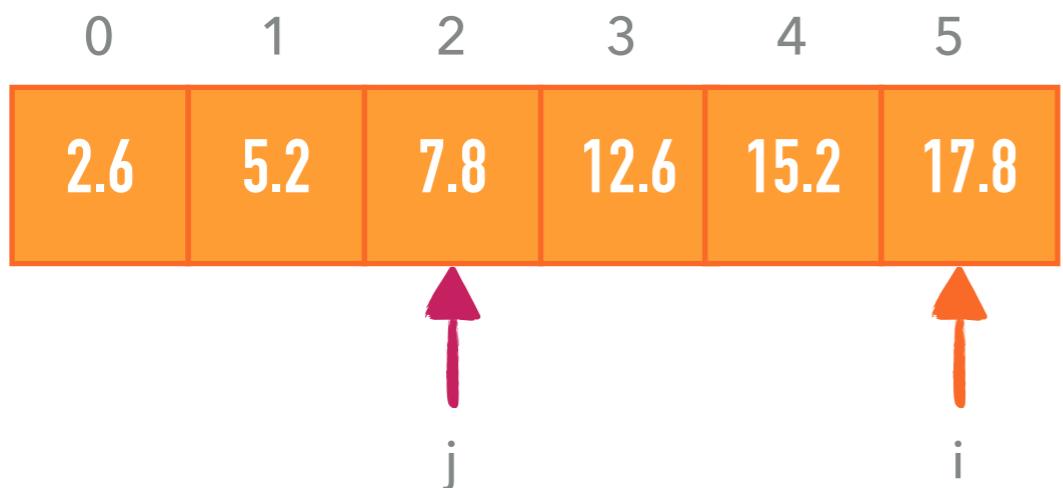


Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$5.2 + 12.6 == 17.8$$

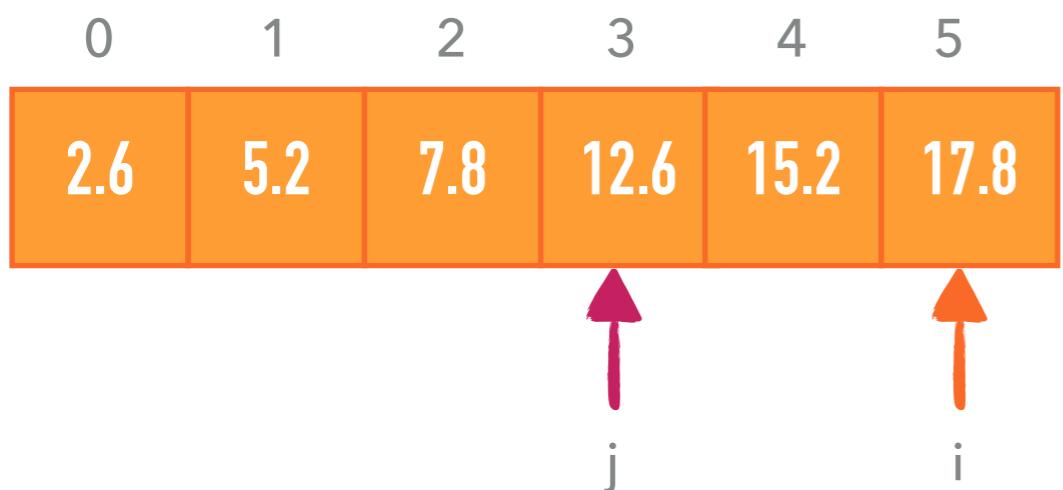


Tabella dei prezzi

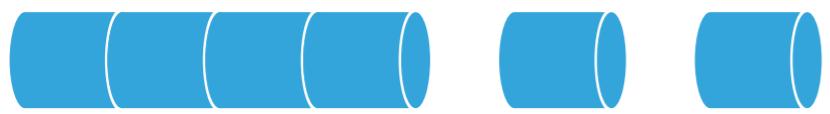
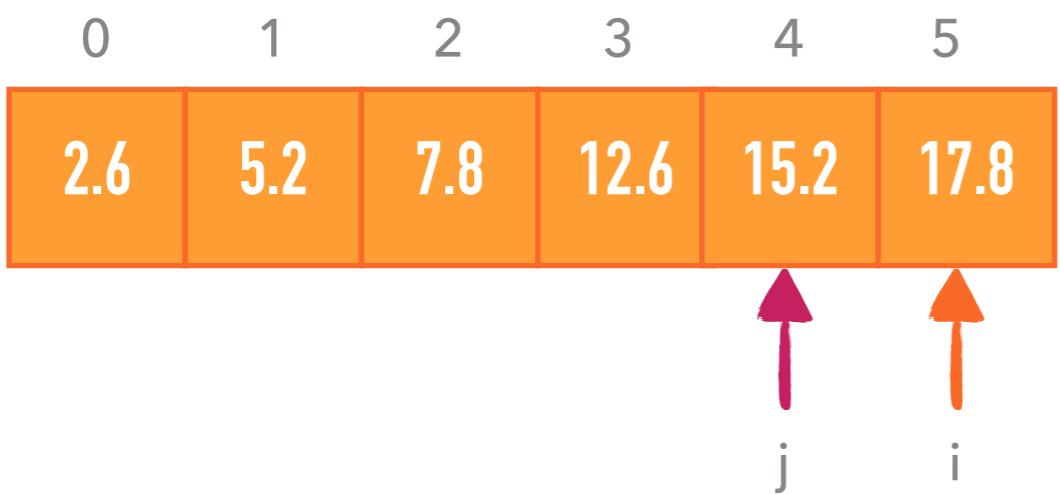
| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

SOLUZIONE CON PROGRAMMAZIONE DINAMICA



Barra da tagliare, lunghezza $n = 6$

$$15.2 + 2.6 == 17.8$$



$$12.6 + 2.6 + 2.6$$

La soluzione ottima
consiste quindi in questa
sequenza di tagli

Tabella dei prezzi

| Lunghezza | Prezzo |
|-----------|--------|
| 1 | 2.6 |
| 2 | 4.5 |
| 3 | 6.1 |
| 4 | 12.6 |
| 5 | 10.4 |
| 6 | 16.7 |

LA SOLUZIONE DI PROGRAMMAZIONE DINAMICA

- ▶ Riempiendo la tabella con le soluzioni dei sotto-problemi non dobbiamo mai ricalcolarli
- ▶ Complessità dell'algoritmo?
 - ▶ Due cicli for innestati, ognuno che esegue al più n iterazioni, quindi $O(n^2)$
 - ▶ Siano quindi passati da un tempo esponenziale (in termini pratici intrattabile) ad un tempo quadratico

QUANDO APPLICARE LA PROGRAMMAZIONE DINAMICA

- ▶ **Sotto-struttura ottima:** la soluzione ottima ad un problema è composta da soluzioni ottime a sotto-problemi più piccoli
- ▶ **Sotto-problemi ripetuti:** trovare la soluzione ottima richiede di risolvere più volte lo stesso sotto-problema
- ▶ Sotto queste due condizioni possiamo pensare di applicare un algoritmo di programmazione dinamica

LONGEST COMMON SUBSEQUENCE

Quanto "simili" sono queste due sequenze di basi?

ACCGGTCGAGTGC~~GCGGAA~~AGCCGGCCGAA

GTCGTT~~CGGA~~ATGCCGTTGCTCTGTAAA

Un modo per valutare la similarità è trovare la già lunga sottosequenza di caratteri comune ad entrambe le sequenze

Cosa è una sottosequenza?

Cosa significa che è comune?

LONGEST COMMON SUBSEQUENCE

- ▶ Data una sequenza di simboli $X = \langle x_1, x_2, \dots, x_m \rangle$, una sequenza $Z = \langle z_1, z_2, \dots, z_k \rangle$ è una sottosequenza di X se esiste una sequenza strettamente crescente di indici $\langle i_1, \dots, i_k \rangle$ tale per cui per ogni $j = 1, \dots, k$ abbiamo che $x_{i_j} = z_j$
- ▶ Esempio: CASALE ha CASA, CAAE, ALE, SALE come sottosequenze (le otteniamo considerando – in ordine – solo alcuni dei caratteri di CASALE), ma non LESA (abbiamo tutte le lettere ma non preserviamo l'ordine)

LONGEST COMMON SUBSEQUENCE

- ▶ Data tue sequenze $X = \langle x_1, x_2, \dots, x_m \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$
vogliamo trovare la lunghezza della più lunga
sottosequenza comune ad entrambe (possono esserci più
sottosequenze di lunghezza massimale)
- ▶ Esempio: se $X = \text{CASSA}$ e $Y = \text{ASSICURAZIONE}$, la già
lunga sottosequenza comune è ASSA, di lunghezza 4:
CASSA e **ASSICURAZIONE**

LONGEST COMMON SUBSEQUENCE

- ▶ Un modo per risolvere il problema è enumerare tutte le possibili sottosequenze e trovare quelle in comune
- ▶ Però il numero di sottosequenze possibili è estremamente elevato! (quante ce ne sono?)
- ▶ Proviamo a stabilire come deve essere la struttura di una soluzione ottima

LCS: CASI BASE

- ▶ Se $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle \rangle$ (la sequenza vuota) allora la più lunga sottosequenza comune ha lunghezza 0 ed è $Z = \langle \rangle$
- ▶ Simmetricamente se è X a essere la sequenza vuota
- ▶ Quindi sappiamo la soluzione ottima nel caso in cui una (o entrambe) le sequenze siano vuote

LONGEST COMMON SUBSEQUENCE

- ▶ Sia $Z = \langle z_1, \dots, z_k \rangle$ una più lunga sottosequenza comune tra $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$
- ▶ Consideriamo gli ultimi due elementi delle sequenze X e Y
 - ▶ Se $x_m = y_n$ allora $z_k = x_m = y_n$, altrimenti sarebbe possibile allungare Z concatenandoci x_m .
In questo caso $Z' = \langle z_1, \dots, z_{k-1} \rangle$ è la più lunga sottosequenza comune di $X' = \langle x_1, \dots, x_{m-1} \rangle$ e $Y' = \langle y_1, \dots, y_{n-1} \rangle$

LONGEST COMMON SUBSEQUENCE

- ▶ Se $x_m \neq y_n$ allora possiamo ignorare uno tra x_m e y_n
(almeno uno dei due **non** farà parte della LCS di X e Y)
- ▶ Potremmo avere che Z è la LCS di $X = \langle x_1, \dots, x_m \rangle$ e
 $Y' = \langle y_1, \dots, y_{n-1} \rangle$
- ▶ Oppure che Z è la LCS di $X' = \langle x_1, \dots, x_{m-1} \rangle$ e
 $Y = \langle y_1, \dots, y_n \rangle$

LONGEST COMMON SUBSEQUENCE

Proviamo ora a definire una soluzione.

Date $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$ indichiamo con $c_{i,j}$ la lunghezza della più lunga sottosequenza comune tra $\langle x_1, \dots, x_i \rangle$ e $\langle y_1, \dots, y_j \rangle$

A noi interessa quindi il valore $c_{m,n}$

Ma $c_{i,j}$ è definito come:

$$c_{i,j} = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ 1 + c_{i-1,j-1} & \text{se } x_i = y_j \\ \max\{c_{i-1,j}, c_{i,j-1}\} & \text{se } x_i \neq y_j \end{cases}$$

Possiamo quindi creare una matrice di $m + 1$ righe e $n + 1$ colonne in cui calcolare i valori di $c_{i,j}$

PSEUDOCODICE: LCS

Parametri: sequenza X di lunghezza m e Y di lunghezza n

```
c = matrice (m + 1) × (n + 1)
for i in range(0, m+1): # casi base: sequenza vuota
    c[i][0] = 0
for j in range(0, n+1):
    c[0][j] = 0
for i in range(1, m+1):
    for j in range(1, n+1):
        if X[i-1] == Y[j-1]: # Se il carattere in posizione i,j coincide
            c[i][j] = 1 + c[i-1][j-1]
        else: # altrimenti prendiamo il migliore dei due sottoproblemi
            c[i][j] = max(c[i-1][j], c[i][j-1])
return c[m][n]
```

LONGEST COMMON SUBSEQUENCE

| | | A | B | B | A | |
|---|--|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 |
| C | | 0 | 0 | 0 | 0 | 0 |
| A | | 1 | 0 | 0 | 0 | 0 |
| A | | 2 | 0 | 1 | 1 | 1 |
| S | | 3 | 0 | 1 | 1 | 1 |
| A | | 4 | 0 | 1 | 1 | 2 |

PROGRAMMAZIONE DINAMICA
DISTANZA DI LEVENSHTEIN

ALGORITMI E STRUTTURE DATI

DISTANZA DI LEVENSHTEIN

- ▶ La distanza di Levenshtein serve a quantificare quanto due stringhe siano diverse
- ▶ Abbiamo due sequenze $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$ e ci chiediamo quale sia il numero minimo di operazioni di **inserimento, rimozione e sostituzione** necessarie per trasformare X in Y .
- ▶ Utile, per esempio, per trovare la correzione automatica (i.e., quale è la parola più simile a quella che abbiamo digitato?)

DISTANZA DI LEVENSHTEIN: OPERAZIONI

- ▶ Inserimento di un carattere z in posizione i nella sequenza $X = \langle x_1, \dots, x_m \rangle$: otteniamo $X' = \langle x_1, \dots, x_{i-1}, z, x_i, \dots, x_m \rangle$
- ▶ Inserimento di “S” in posizione 3 di “CASA”: “CASSA”
- ▶ Cancellazione del carattere in posizione i nella sequenza $X = \langle x_1, \dots, x_m \rangle$: otteniamo $X' = \langle x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_m \rangle$
- ▶ Cancellazione del carattere in posizione 1 di “CASA”: “ASA”

DISTANZA DI LEVENSHTEIN: OPERAZIONI

- ▶ Sostituzione del carattere in posizione i con il carattere z nella sequenza $X = \langle x_1, \dots, x_m \rangle$: otteniamo
$$X' = \langle x_1, \dots, x_{i-1}, z, x_{i+1}, \dots, x_m \rangle$$
- ▶ Sostituzione del carattere in posizione 4 con "E" in "CASA": "CASE"

DISTANZA DI LEVENSHTEIN

- ▶ Come possiamo approcciare il problema di trovare il minimo numero di operazioni?
- ▶ Ci riconduciamo a sotto-problemi più piccoli
- ▶ E vediamo come comporre le soluzioni ottime di problemi più piccoli in modo da formare la soluzione
- ▶ Iniziamo cercando dei casi base

DISTANZA DI LEVENSHTEIN: CASI BASE

- ▶ Se $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle \rangle$ allora dobbiamo cancellare tutti i caratteri di X per ottenere Y : servono m operazioni
- ▶ Se $X = \langle \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$ dobbiamo inserire in X tutti i caratteri di Y : servono n operazioni
- ▶ Quindi se una delle due sequenze è vuota ci servono tante operazioni di inserimento o cancellazione quanto è la lunghezza dell'altra sequenza

DISTANZA DI LEVENSHTEIN: SOTTO-STRUTTURA OTTIMA (1)

- ▶ Se abbiamo le due sequenze $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$, come possiamo agire guardano l'ultimo elemento?
- ▶ Possiamo sostituire x_m con y_n se $x_m \neq y_n$. Dopo questa operazione le due sequenze coincidono sull'ultimo elemento e dobbiamo trovare quante operazioni svolgere per trasformare $X' = \langle x_1, \dots, x_{m-1} \rangle$ in $Y' = \langle y_1, \dots, y_{n-1} \rangle$

DISTANZA DI LEVENSHTEIN: SOTTO-STRUTTURA OTTIMA (2)

- ▶ Se abbiamo le due sequenze $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$, come possiamo agire guardano l'ultimo elemento?
- ▶ Se $x_m = y_n$ non svolgiamo alcuna operazione e dobbiamo trovare quante operazioni svolgere per trasformare $X' = \langle x_1, \dots, x_{m-1} \rangle$ in $Y' = \langle y_1, \dots, y_{n-1} \rangle$

DISTANZA DI LEVENSHTEIN: SOTTO-STRUTTURA OTTIMA (3)

- ▶ Se abbiamo le due sequenze $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$, come possiamo agire guardano l'ultimo elemento?
- ▶ Possiamo decidere di inserire y_n come ultimo carattere di X e quindi chiederci il numero minimo di operazioni da svolgere per trasformare X in $Y' = \langle y_1, \dots, y_{n-1} \rangle$

DISTANZA DI LEVENSHTEIN: SOTTO-STRUTTURA OTTIMA (4)

- ▶ Se abbiamo le due sequenze $X = \langle x_1, \dots, x_m \rangle$ e $Y = \langle y_1, \dots, y_n \rangle$, come possiamo agire guardano l'ultimo elemento?
- ▶ Possiamo decidere di cancellare l'ultimo carattere di X e quindi chiederci il numero minimo di operazioni da svolgere per trasformare $X' = \langle x_1, \dots, x_{m-1} \rangle$ in Y

DISTANZA DI LEVENSHTEIN

- ▶ Abbiamo coperto tutti i possibili casi con cui possiamo applicare una qualche operazione per far coincidere le due stringhe sull'ultimo elemento
- ▶ Una volta fatto questo ci ritroviamo un problema già semplice da risolvere:
 - ▶ O entrambe le sequenze sono più corte di un carattere
 - ▶ O almeno una è più corta di un carattere
- ▶ Vediamo ora come calcolare il costo minimo

DISTANZA DI LEVENSHTEIN

- ▶ Indichiamo con $c_{i,j}$ il costo minimo per trasformare la sotto-sequenza $X' = \langle x_1, \dots, x_i \rangle$ in $Y' = \langle y_1, \dots, y_j \rangle$
- ▶ Se $i = 0$ o $j = 0$ siamo in uno dei casi base:
 - ▶ $c_{i,0} = i$
 - ▶ $c_{0,j} = j$

DISTANZA DI LEVENSHTEIN

- ▶ Se $i \neq 0$ e $j \neq 0$ allora abbiamo che dobbiamo scegliere il costo minimo tra tutte le operazioni che possiamo effettuare:
 - ▶ $1 + c_{i-1,j-1}$ se effettuiamo una sostituzione e $x_i \neq y_j$
 - ▶ $c_{i-1,j-1}$ se effettuiamo una sostituzione e $x_i = y_j$
 - ▶ $1 + c_{i-1,j}$ se cancelliamo l'ultimo carattere di X'
 - ▶ $1 + c_{i,j-1}$ se inseriamo l'ultimo carattere di Y'

DISTANZA DI LEVENSHTEIN

- ▶ Abbiamo quindi che

$$c_{i,j} = \begin{cases} \max(i, j) & \text{se } i = 0 \text{ o } j = 0 \\ \min(1 + c_{i-1,j-1}, 1 + c_{i,j-1}, 1 + c_{i-1,j}) & \text{se } i, j \neq 0 \text{ e } x_i \neq y_j \\ \min(c_{i-1,j-1}, 1 + c_{i,j-1}, 1 + c_{i-1,j}) & \text{se } i, j \neq 0 \text{ e } x_i = y_j \end{cases}$$

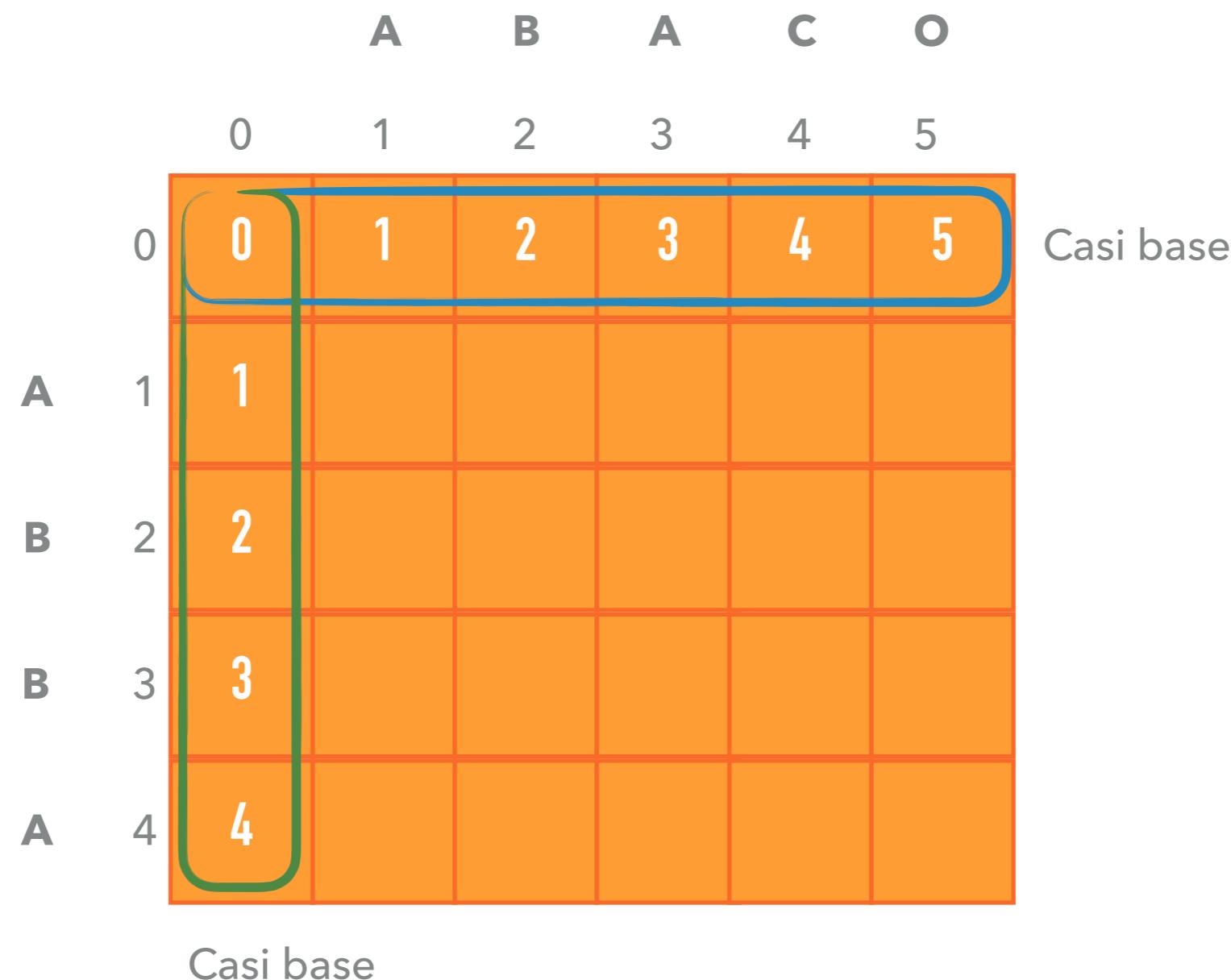
- ▶ Questo ci permette di creare una tabella di $m + 1$ righe e $n + 1$ colonne che riempiremo coi valori di $c_{i,j}$

PSEUDOCODICE: DISTANZA DI LEVENSHTEIN

Parametri: sequenze X e Y di lunghezza m e n

```
c = matrice (m + 1) × (n + 1) # la matrice che conterrà tutti i  $c_{i,j}$ 
for i in range(0, m+1): # primo caso base, la sequenza Y è vuota
    c[i][0] = i
for j in range(0, n+1): # secondo caso base, la sequenza X è vuota
    c[0][j] = j
for i in range(1, m+1):
    for j in range(1, n+1):
        cancellazione = 1 + c[i-1][j]
        inserimento = 1 + c[i][j-1]
        sostituzione = c[i-1][j-1]
        if X[i] ≠ Y[j]: # la sostituzione è effettuata solo se  $x_i$  non coincide con  $y_j$ 
            sostituzione = sostituzione + 1
        c[i][j] = min(sostituzione, inserimento, cancellazione)
return c[m][n]
```

DISTANZA DI LEVENSHTEIN



DISTANZA DI LEVENSHTEIN

| | | A | B | A | C | O | |
|---|--|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| A | | 1 | 0 | | | | |
| B | | 2 | | | | | |
| B | | 3 | | | | | |
| A | | 4 | | | | | |

Dobbiamo fare il minimo tra
 $1+C[0][1]$
 $1+C[1][0]$
 $C[0][0]$

Questo perché $x_1 = y_1$

DISTANZA DI LEVENSHTEIN

| | | A | B | A | C | O | | |
|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | |
| | | 0 | 0 | 1 | 2 | 3 | 4 | 5 |
| A | 1 | 1 | 0 | 1 | | | | |
| B | 2 | 2 | | | | | | |
| B | 3 | 3 | | | | | | |
| A | 4 | 4 | | | | | | |

Dobbiamo fare il minimo tra

$$1+C[0][2]$$

$$1+C[1][1]$$

$$1+C[0][1]$$



Questo perché $x_1 \neq y_2$

DISTANZA DI LEVENSHTEIN

| | | A | B | A | C | O | |
|---|--|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| 0 | | 0 | 1 | 2 | 3 | 4 | 5 |
| A | | 1 | 0 | 1 | 2 | 3 | 4 |
| B | | 2 | 1 | 0 | 1 | 2 | 3 |
| B | | 3 | 2 | 1 | 1 | 2 | 3 |
| A | | 4 | 3 | 2 | 1 | 2 | 3 |

Distanza di Levenshtein
tra "ABBA" e "ABACO"

DISTANZA DI LEVENSHTEIN: COMPLESSITÀ

- ▶ Dobbiamo calcolare tutti i valori contenuti in una tabella di dimensione $(m + 1) \times (n + 1)$, quindi $O(mn)$ valori
- ▶ Riempire ogni valore richiede solamente un numero costante di operazioni, quindi il costo totale è $O(mn)$

DISTANZA DI LEVENSHTEIN: VARIANTI

- ▶ Sfruttando la stessa idea possiamo definire molteplici varianti dello stesso algoritmo
- ▶ Possiamo dare costi diversi a seconda dell'operazione, per esempio w_{del} , w_{ins} e w_{sub} per cancellazione, inserimento e sostituzione. In quel caso è sufficiente cambiare lievemente la definizione di $c_{i,j}$:

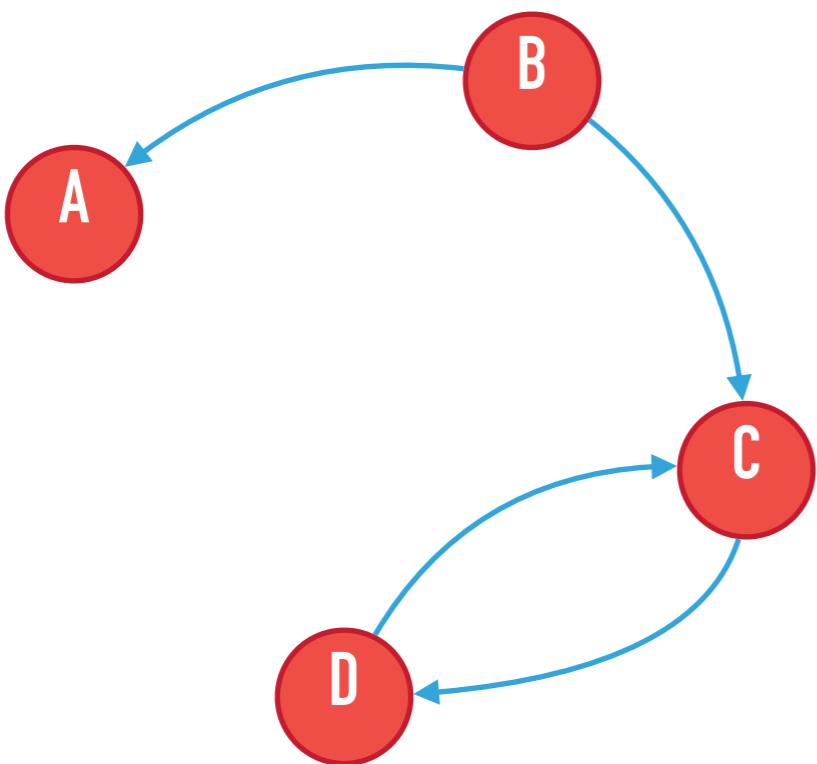
$$c_{i,j} = \begin{cases} \max(w_{\text{del}}i, w_{\text{ins}}j) & \text{se } i = 0 \circ j = 0 \\ \min(w_{\text{sub}} + c_{i-1,j-1}, w_{\text{ins}} + c_{i,j-1}, w_{\text{del}} + c_{i-1,j}) & \text{se } i, j \neq 0 \text{ e } x_i \neq y_j \\ \min(c_{i-1,j-1}, w_{\text{ins}} + c_{i,j-1}, w_{\text{del}} + c_{i-1,j}) & \text{se } i, j \neq 0 \text{ e } x_i = y_j \end{cases}$$

GRAFI
VISITA IN AMPIEZZA

ALGORITMI E STRUTTURE DATI

GRAFI: NOZIONI DI BASE

COSA È UN GRAFO?



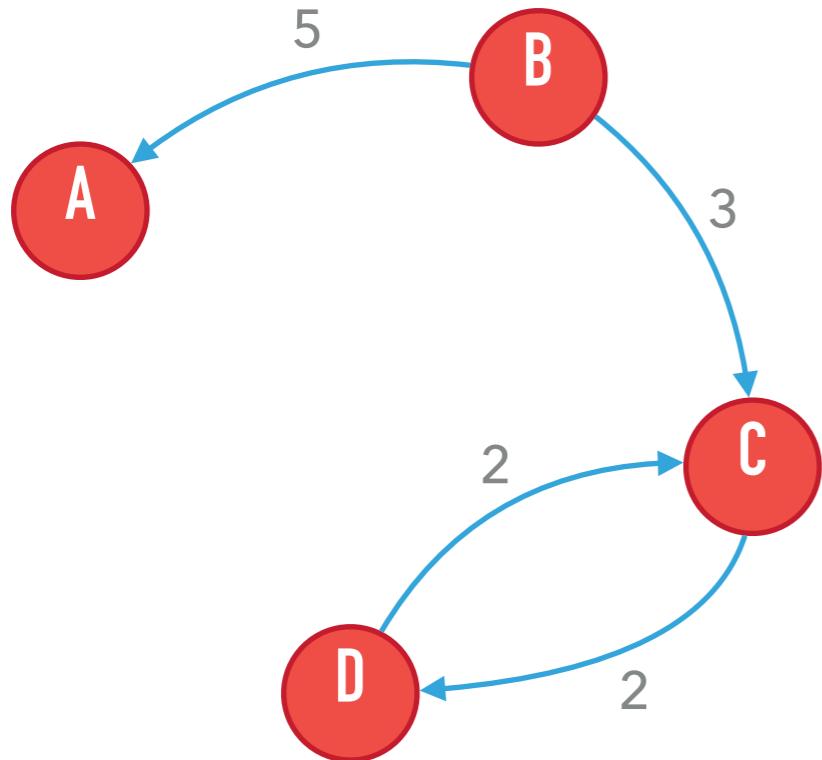
Insieme di nodi:

$$V = \{a, b, c, d\}$$

Insieme di archi:

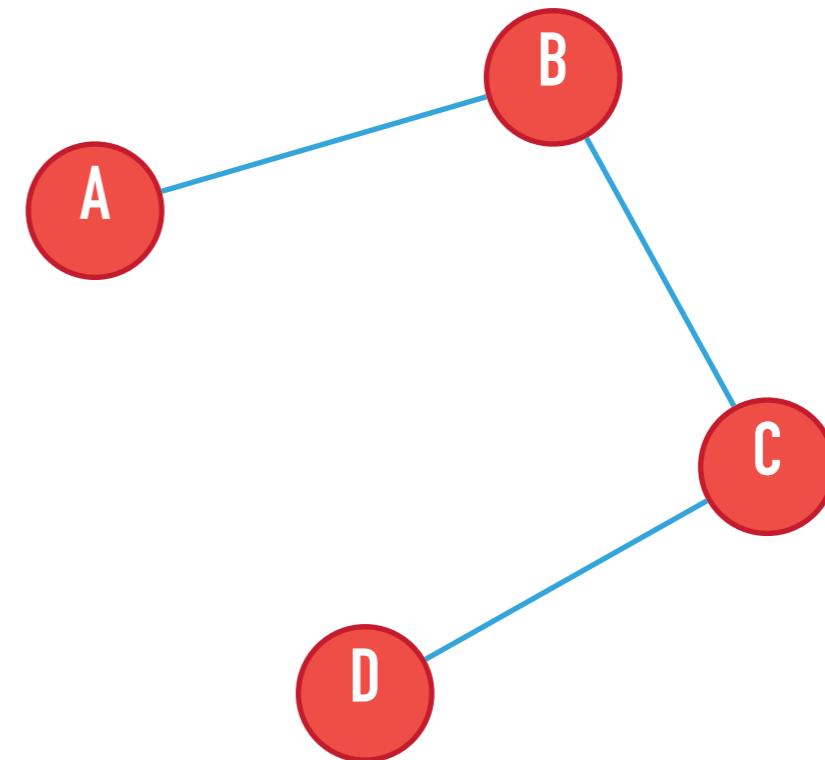
$$E = \{(b, a), (b, c), (c, d), (d, c)\}$$

COSA È UN GRAFO (VARIANTI)?



Archi pesati

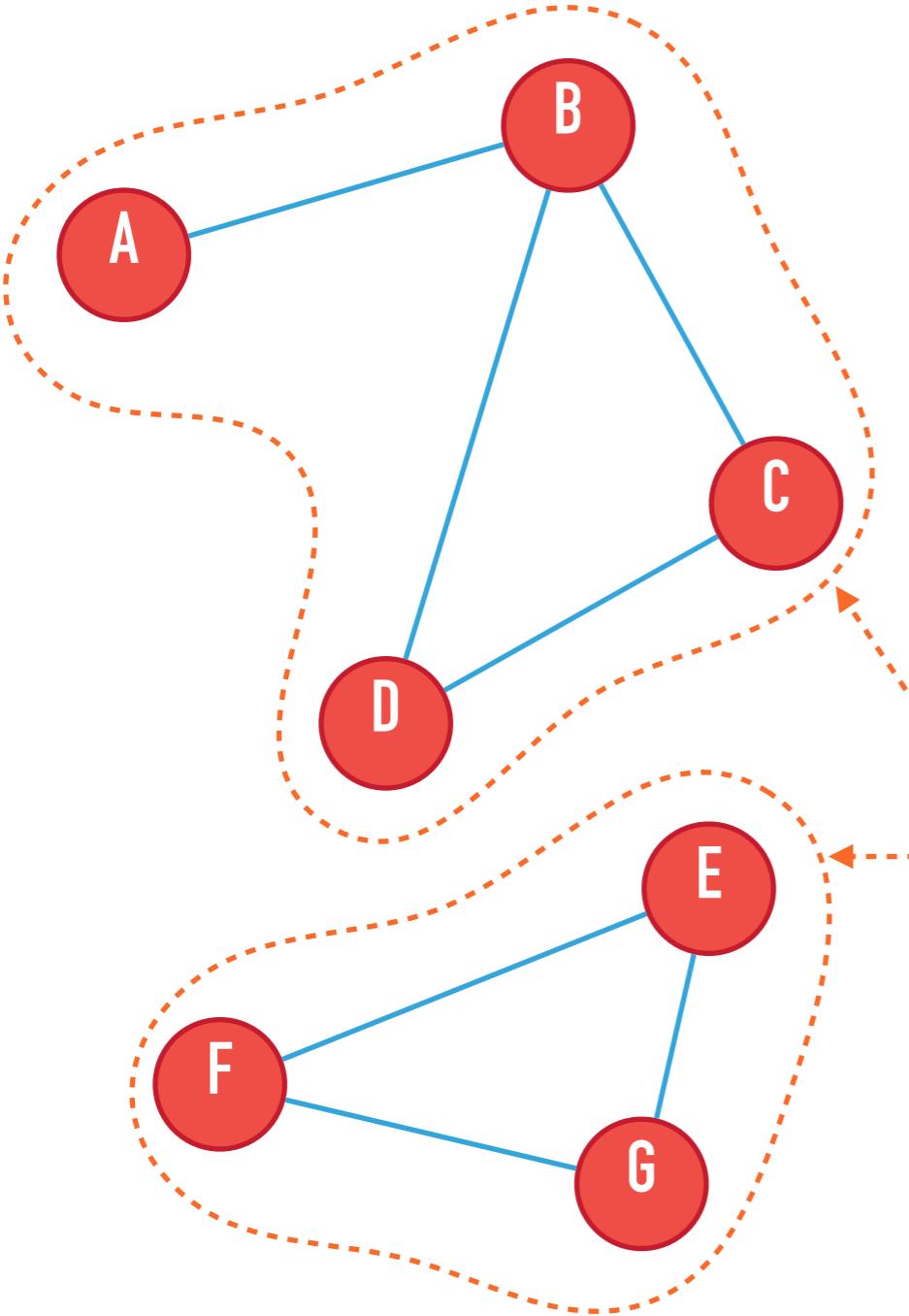
Ovvero esiste una funzione $w : E \rightarrow \mathbb{R}$
e indichiamo $w((i, j))$ come $w_{i,j}$



Non orientato

Ovvero $(a, b) \in E \iff (b, a) \in E$
per ogni $a, b \in V$

GRAFI: NOZIONI DI BASE



Cammino o Percorso: sequenza di archi $(v_0, v_1), (v_1, v_2), \dots$ dove due archi consecutivi nella sequenza sono adiacenti nel grafo (i.e., sono nella forma $(a, b)(b, c)$)

Lunghezza del cammino: numero di archi che il percorso contiene

Distanza tra due nodi a e b : lunghezza del cammino più corto che inizia al nodo a e termina al nodo b . Se non esiste un percorso diremo che la distanza è $+\infty$

Componenti connesse

NOTAZIONE

- ▶ Dato un grafo $G = (V, E)$ solitamente l'input viene misurato in modi dipendente da $|V|$ e $|E|$, quindi due parametri e non uno
- ▶ All'interno della notazione asintotica (e solo in quel caso) faremo la semplificazione di utilizzare V e E per indicare $|V|$ e $|E|$.
- ▶ Quindi un algoritmo che richiede tempo $O(V + E)$ è da leggersi come $O(|V| + |E|)$

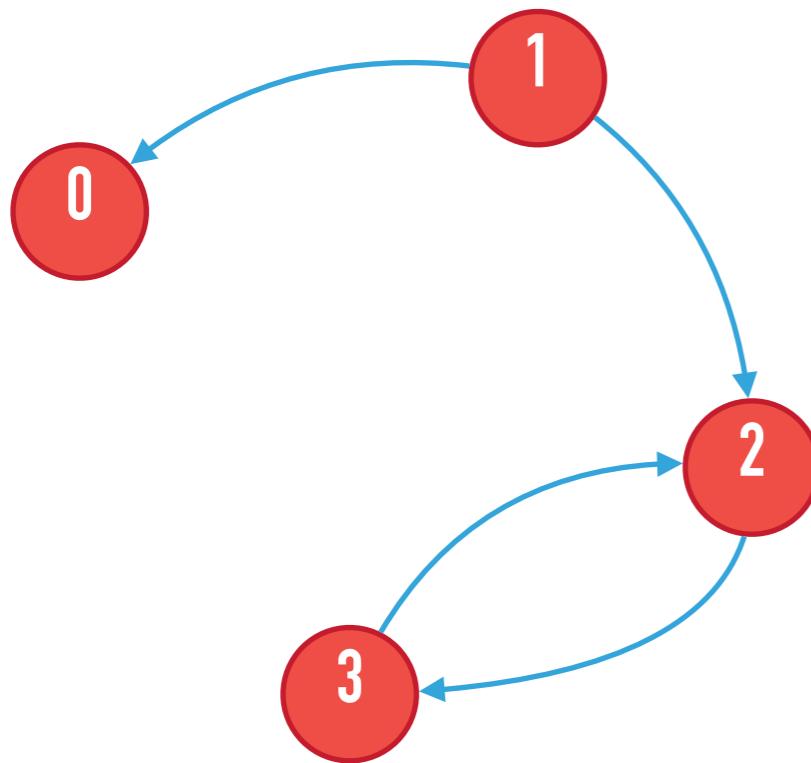
NOTAZIONE

- ▶ Un grafo può avere al più $O(V^2)$ archi, con il valore esatto che dipende dal fatto che il grafo sia non diretto o diretto
- ▶ Solitamente un grafo con un numero di archi vicino al massimo possibile viene detto **denso** mentre uno con un pochi archi viene detto **sparso**.
- ▶ Cosa effettivamente sia considerato un grafo denso o sparso dipende molto dall'applicazione

COSA DOBBIAMO GESTIRE?

- ▶ Dobbiamo essere in grado di salvare i nodi...
- ▶ ...e gli archi (eventualmente con un peso).
- ▶ Assumiamo che i nodi abbiano nomi $\{0, \dots, n-1\}$
- ▶ Ci sono due modi “standard” di rappresentare un grafo:
 - ▶ Matrici di adiacenza
 - ▶ Liste di adiacenza
- ▶ Assumiamo che il grafo sia statico (i.e., non cambi nel tempo)

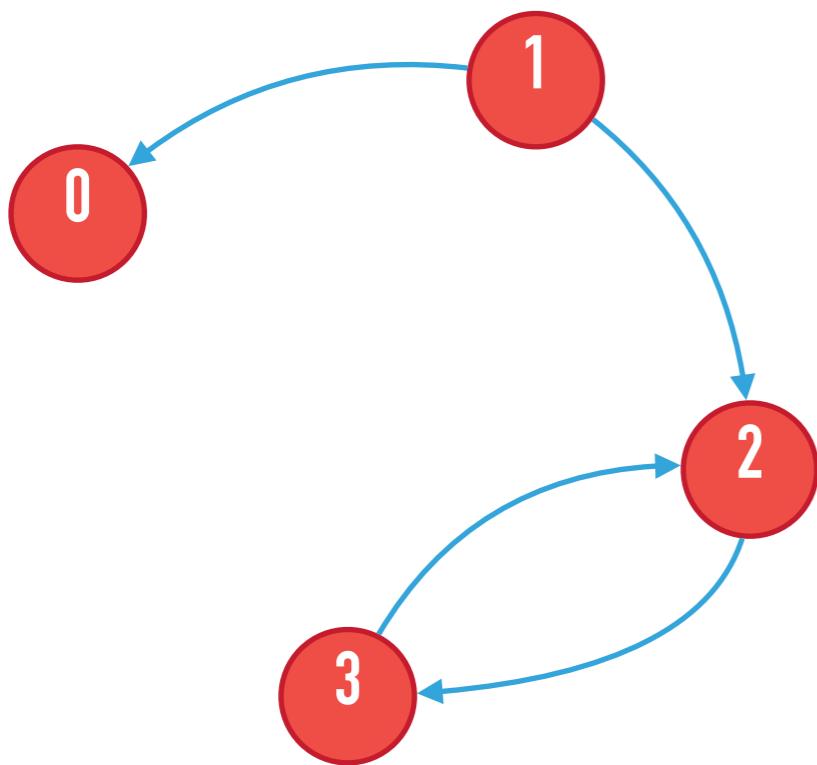
MATRICI DI ADIACENZA



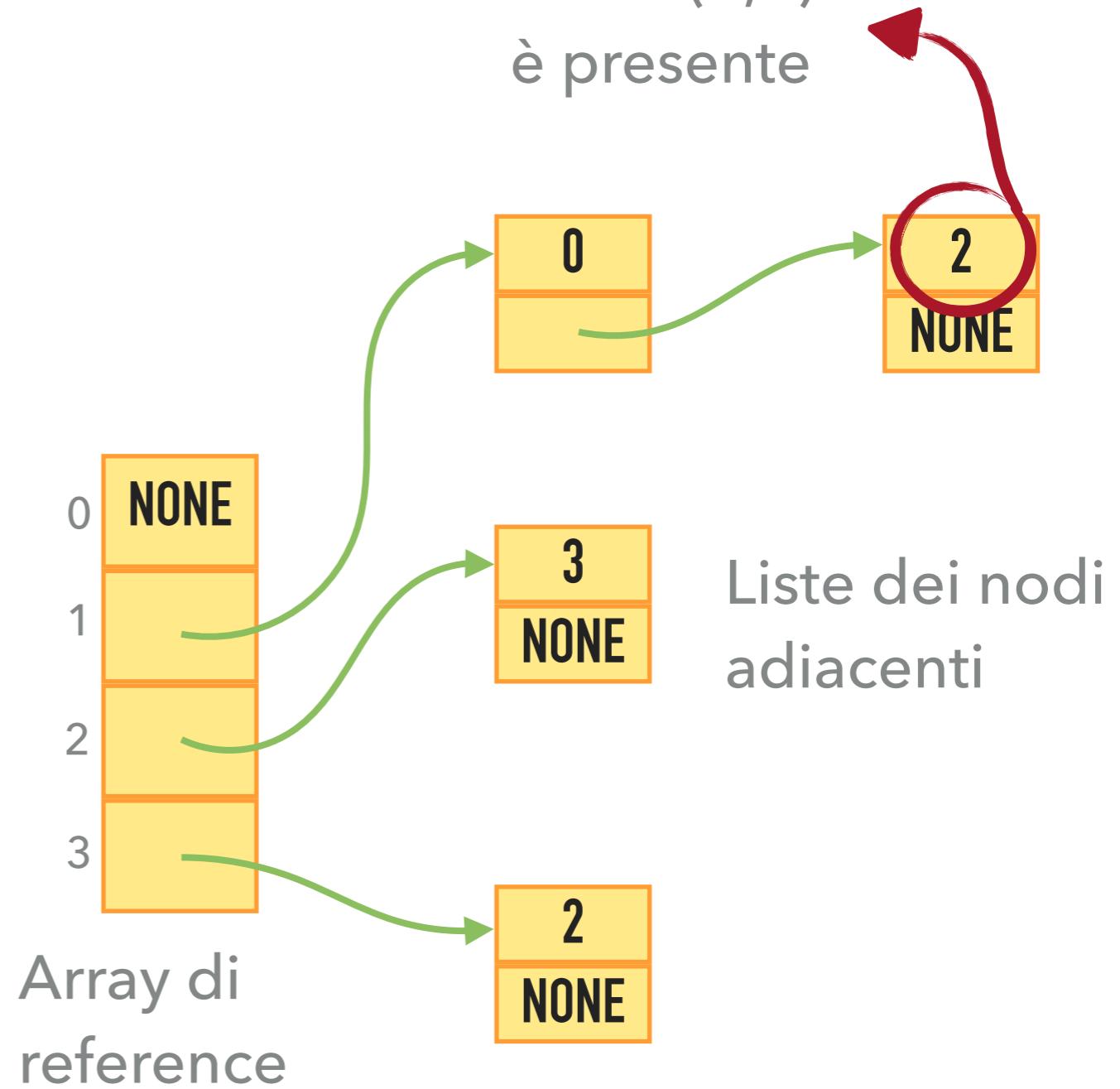
| | | Destinazione | | | |
|----------|---|--------------|---|---|---|
| | | 0 | 1 | 2 | 3 |
| Sorgente | 0 | 0 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 | 1 | |
| 3 | 0 | 0 | 1 | 0 | |

L'arco (1,0) è presente

LISTE DI ADIACENZA



L'arco (1,2) è presente



QUALE RAPPRESENTAZIONE USARE

Matrici di adiacenza

Veloce (tempo costante)
stabilire se un arco esiste

Lento (serve scandire una lista)
stabilire se un arco esiste

Occupazione quadratica di memoria
rispetto al numero di vertici $O(V^2)$

Liste di adiacenza

Occupazione lineare di memoria
rispetto al numero di vertici e archi
 $O(V + E)$

Funziona bene per grafi sparsi

VISITA IN AMPIEZZA

VISITA IN AMPIEZZA

RICERCA IN AMPIEZZA

- ▶ La visita o ricerca in ampiezza (breadth-first search o BFS) è uno degli algoritmi di base per la ricerca su grafi
- ▶ Dato un grafo $G = (V, E)$ e un nodo $s \in V$ detto nodo sorgente la ricerca in ampiezza esplora tutti i nodi raggiungibili a partire da s individuando:
 - ▶ La distanza da s a ognuno dei vertici raggiungibili
 - ▶ Un albero (detto albero BFS) che contiene tutti i vertici raggiungibili

RICERCA IN AMPIEZZA

- ▶ Perché ricerca in ampiezza?
- ▶ A partire dal nodo s si esplorano prima tutti i nodi direttamente raggiungibili da s (quelli a distanza 1)
- ▶ Poi tutti i nodi raggiungibili in due passi (distanza 2), etc.
- ▶ Quindi prima di allontanarci dal nodo sorgente esploriamo tutti quelli vicini, poi i loro vicini, etc.

AMPIEZZA VS PROFONDITÀ



Ricerca in ampiezza:
esploriamo tutti nodi vicini prima di allontanarci



Ricerca in profondità:
seguiamo un singolo percorso il più possibile
prima di cambiare strada

COLORARE I NODI

- ▶ Durante la ricerca in ampiezza (e in generale per le ricerche nei grafi) dobbiamo evitare di visitare un nodo più volte. Per questo assegnamo ad ogni nodo un colore:
- ▶ **Bianco**: il nodo non è ancora stato visitato

- ▶ **Grigio**: il nodo è stato visitato ma potrebbe avere dei vicini non visitati
- ▶ **Nero**: il nodo è stato visitato ed anche tutti i suoi vicini

IDEA DELL'ALGORITMO

- ▶ Teniamo una coda di nodi grigi (dei quali potrebbero mancarci dei vicini da esplorare)
- ▶ Inizialmente solo il nodo sorgente è grigio e viene accodato
- ▶ Estraiamo un nodo dalla coda, coloriamo di grigio tutti i vicini bianchi e li aggiungiamo in coda
- ▶ Ripetiamo finché la coda non è vuota

PSEUDOCODICE: INIZIALIZZAZIONE

Parametri: grafo G , nodo sorgente s

inizialmente impostiamo distanza, colore e predecessore di tutti i nodi

for all $v \in V$:

 colore $[v]$ = bianco

 distanza $[v]$ = $+\infty$

 predecessore $[v]$ = None

il nodo sorgente è il primo che visitiamo e quindi

colore $[s]$ = grigio # assume colore grigio

distanza $[s]$ = 0 # e distanza 0 da sé stesso

$Q = \text{Coda}()$

nella coda dei nodi che potrebbero avere ancora vicini da visitare

viene aggiunto s

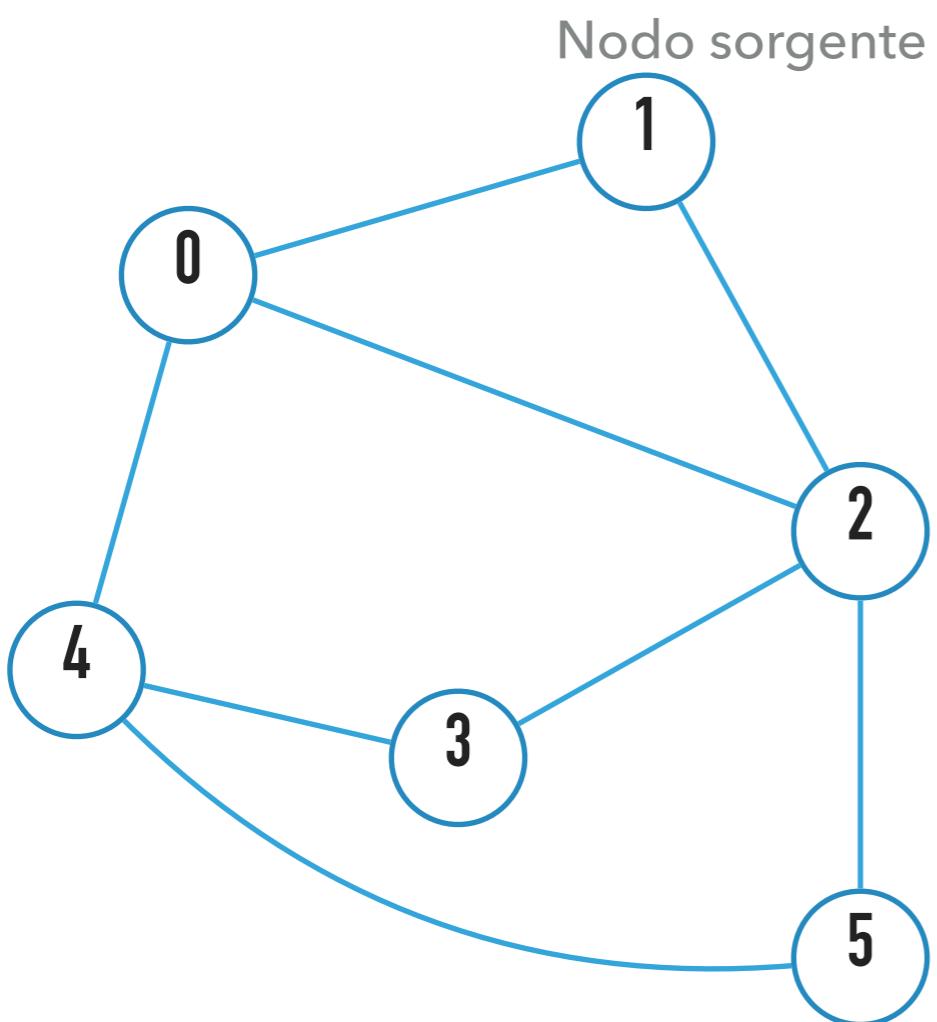
enqueue(Q, s)

PSEUDOCODICE: CICLO PRINCIPALE

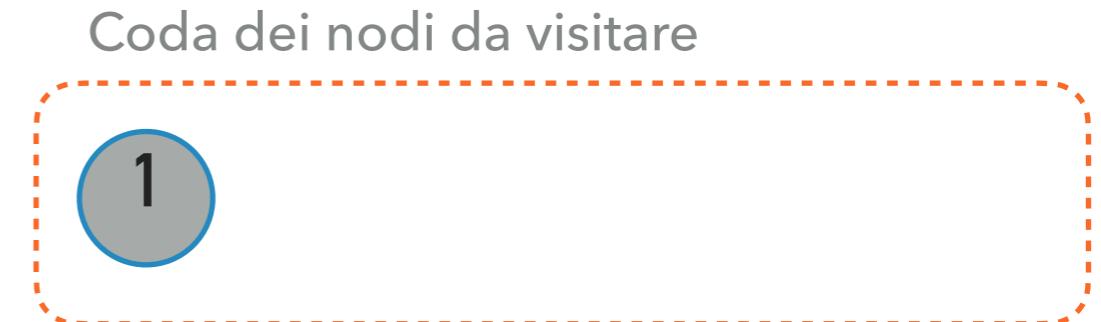
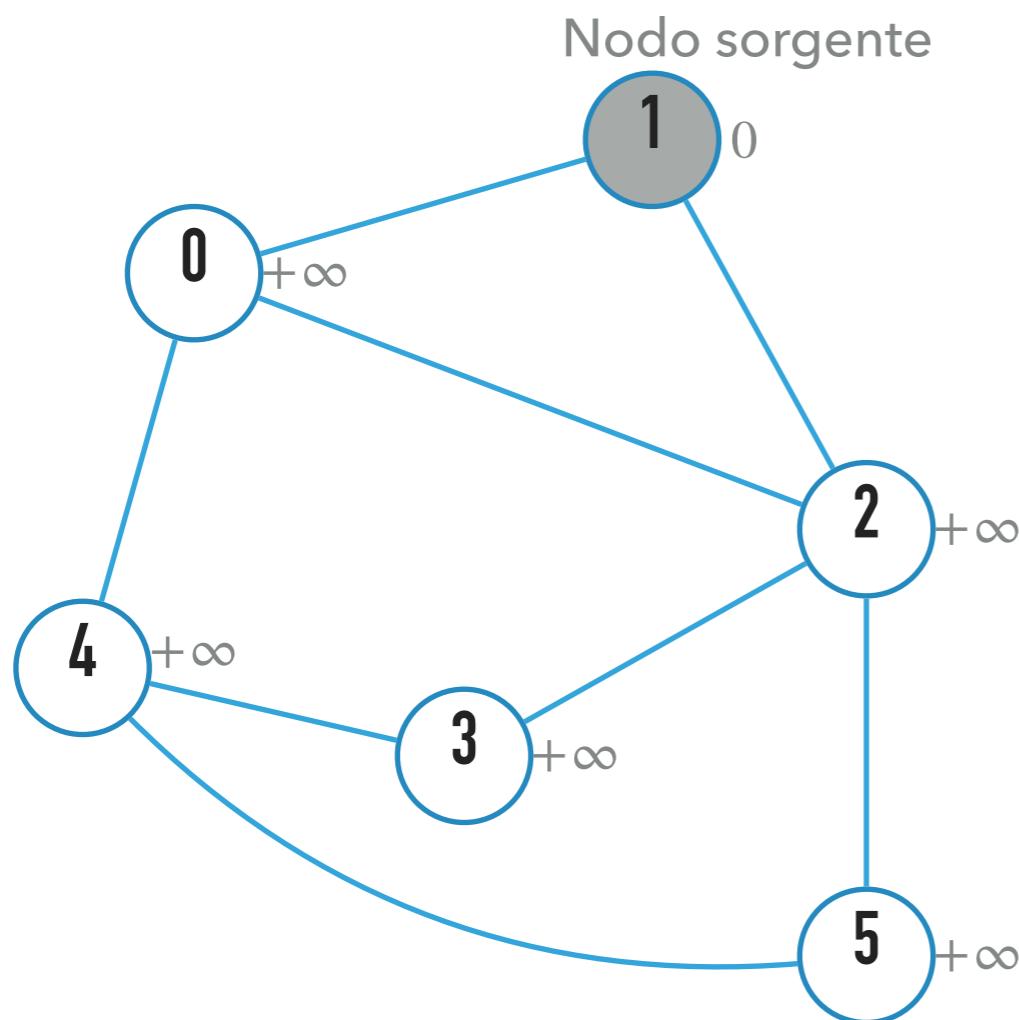
while Q is not empty:

```
# questo ciclo deve continuare finché ci rimangono dei nodi da visitare
u = dequeue(Q) # prendiamo il primo nodo dalla coda
for all v adiacenti a u # e ne esploriamo tutti i vicini
    if color[v] == bianco # consideriamo solo i vicini mai visitati prima
        colore[v] = grigio
        # possiamo arrivare a v con un passo partendo da u
        distanza[v] = distanza[u] + 1
        predecessore[v] = u
        enqueue(Q, v) # accodiamo perché potrebbe avere dei vicini bianchi
    colore[u] = nero # abbiamo finito di visitare tutti i vicini di u
# una volta usciti dal ciclo abbiamo visitato tutti i nodi raggiungibili da s
```

ESEMPIO DI ESECUZIONE



ESEMPIO DI ESECUZIONE

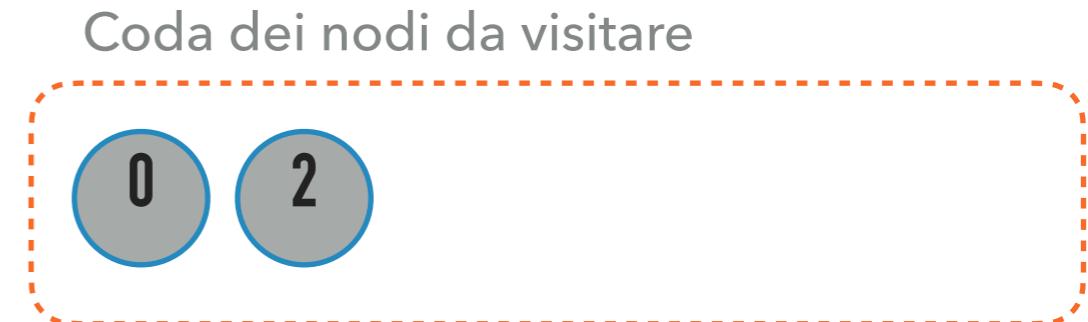
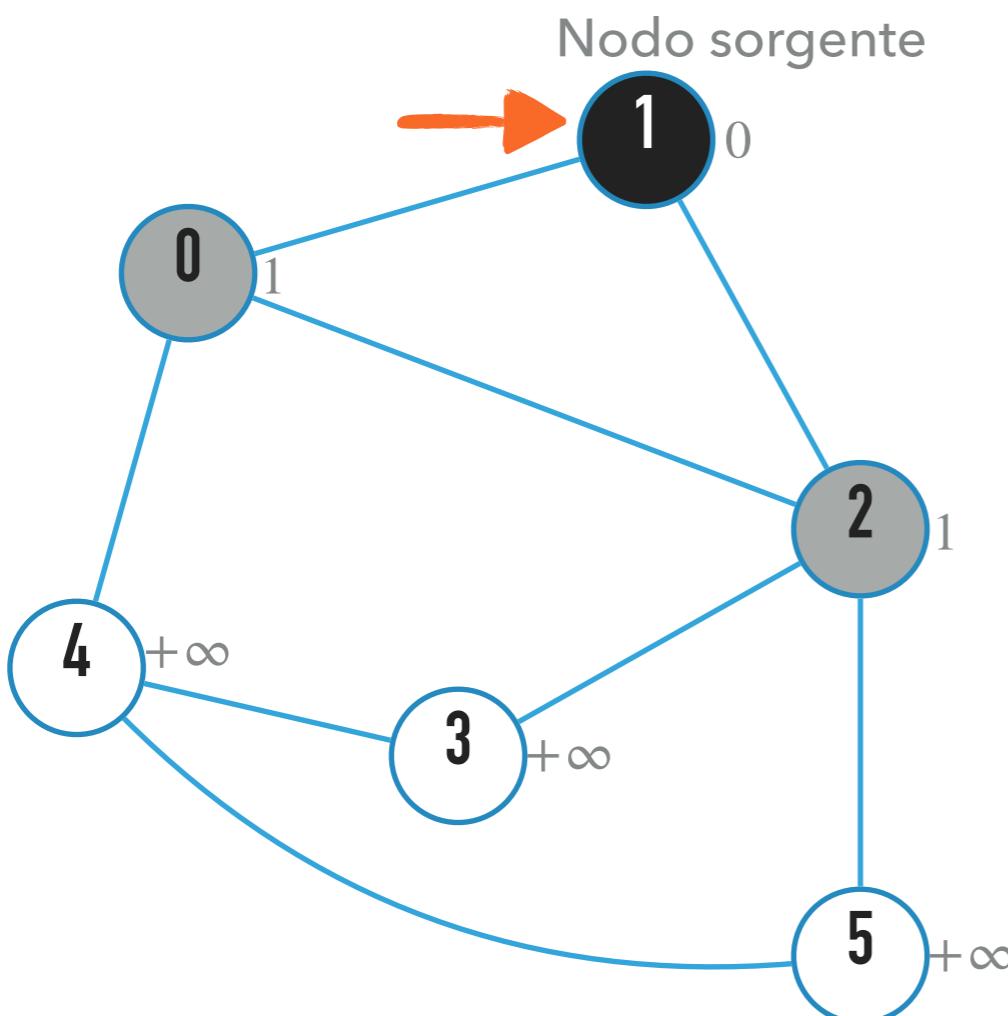


Inizialmente la distanza conosciuta di tutti i nodi dal nodo di partenza è $+\infty$

Solo il nodo iniziale ha distanza 0 da se stesso e colore grigio

| Distanza | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| Predecessore | - | - | - | - | - | - |

ESEMPIO DI ESECUZIONE



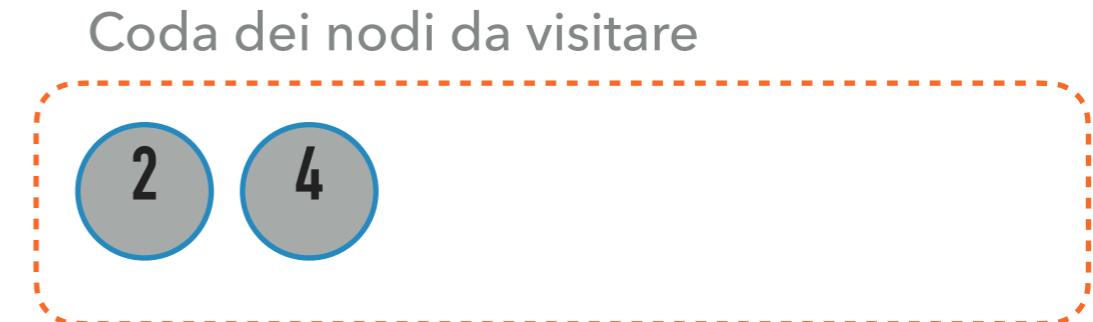
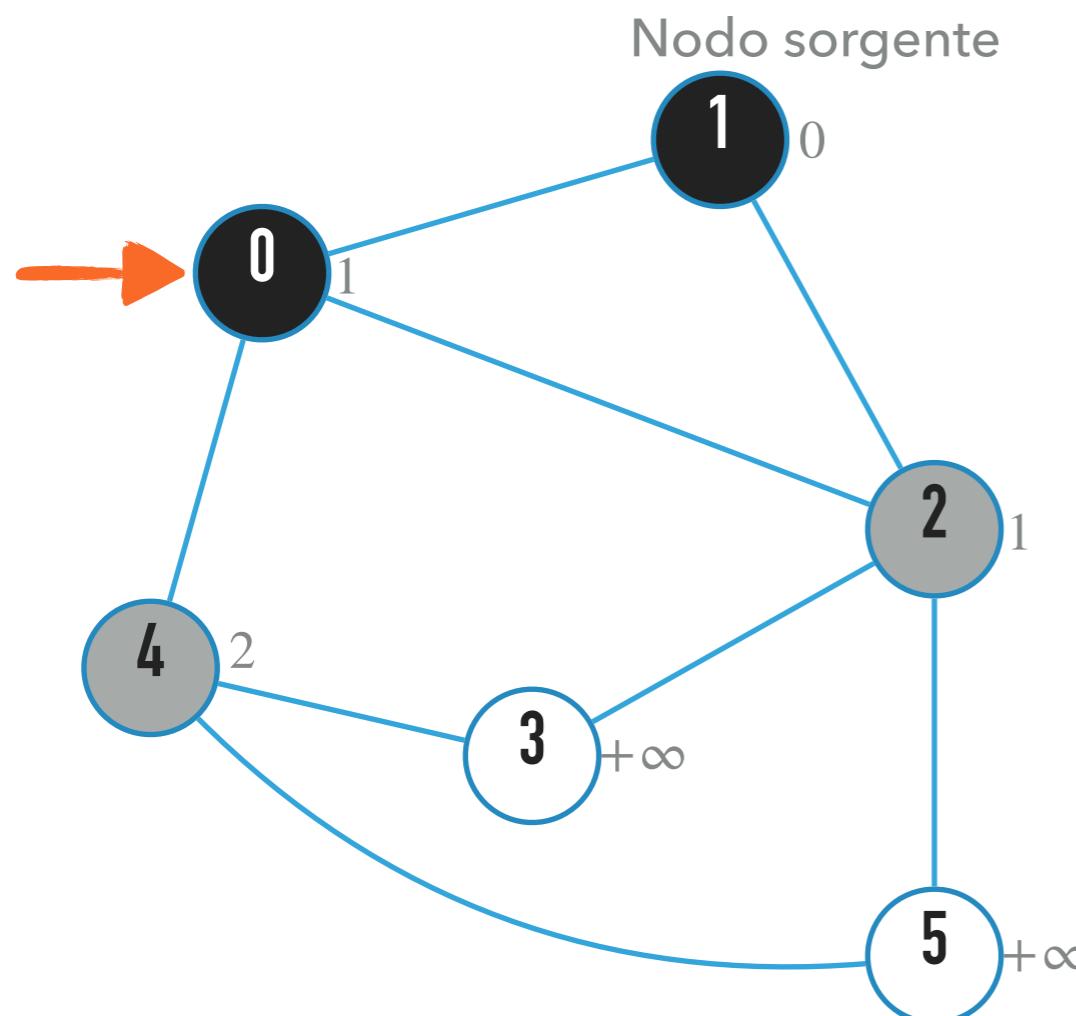
Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio,
aggiorniamo distanza (come $distanza[u] + 1$)
e predecessore (u) e lo accodiamo

Coloriamo u di nero

| | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|----------|----------|----------|
| Distanza | 1 | 0 | 1 | ∞ | ∞ | ∞ |
| Predecessore | 1 | - | 1 | - | - | - |

ESEMPIO DI ESECUZIONE



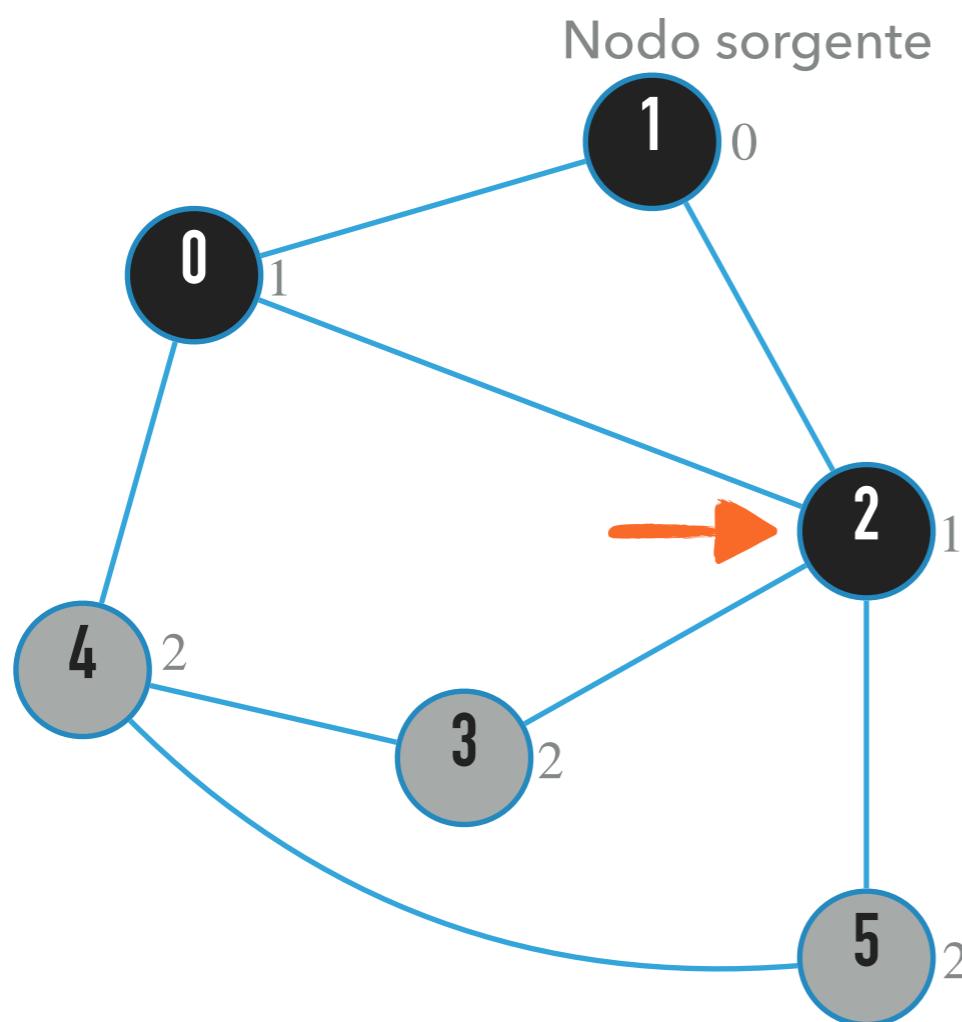
Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio,
aggiorniamo distanza (come $distanza[u] + 1$)
e predecessore (u) e lo accodiamo

Coloriamo u di nero

| | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|----------|---|----------|
| Distanza | 1 | 0 | 1 | ∞ | 2 | ∞ |
| Predecessore | 1 | - | 1 | - | 0 | - |

ESEMPIO DI ESECUZIONE



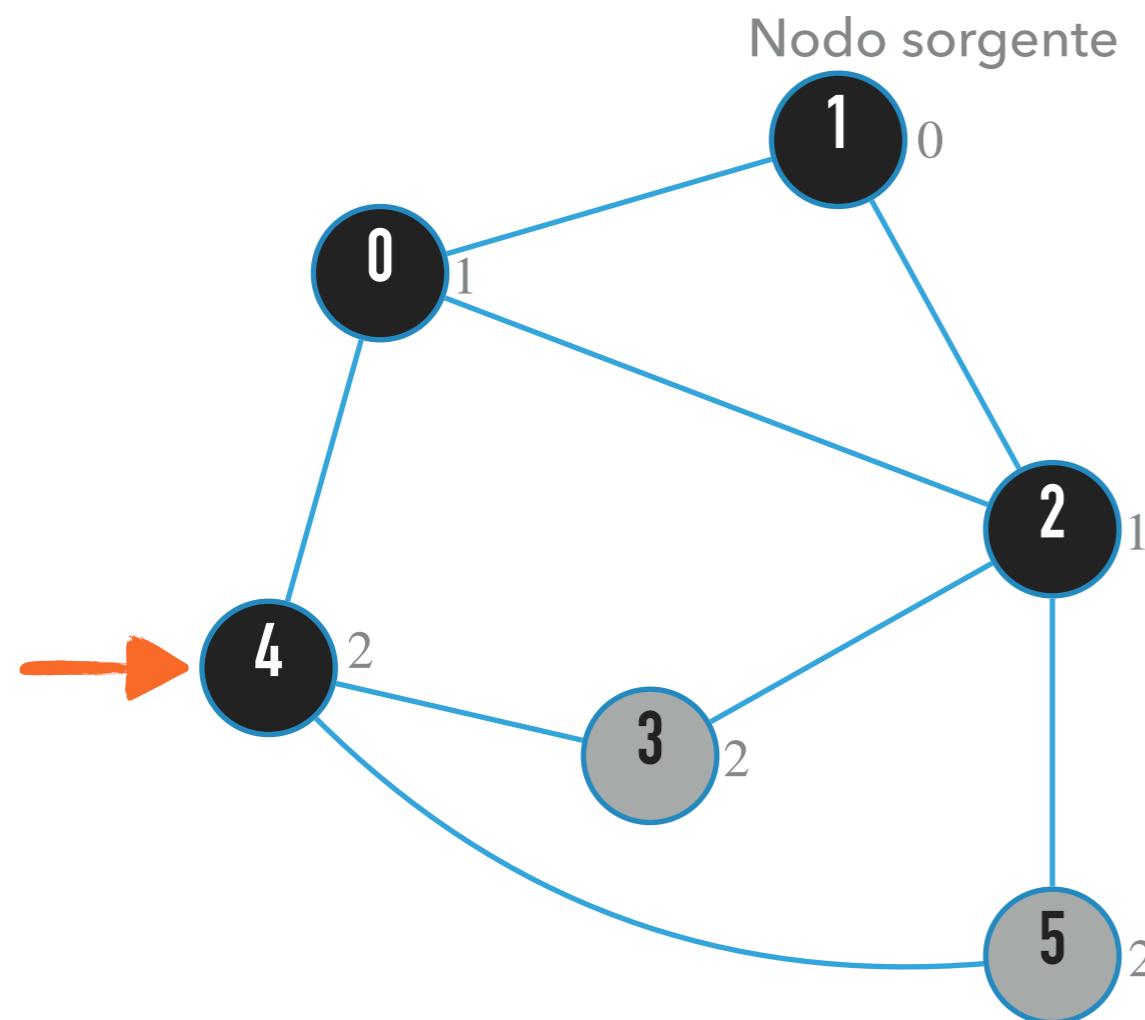
Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio,
aggiorniamo distanza (come $distanza[u] + 1$)
e predecessore (u) e lo accodiamo

Coloriamo u di nero

| Distanza | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| Predecessore | 1 | - | 1 | 2 | 0 | 2 |

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare

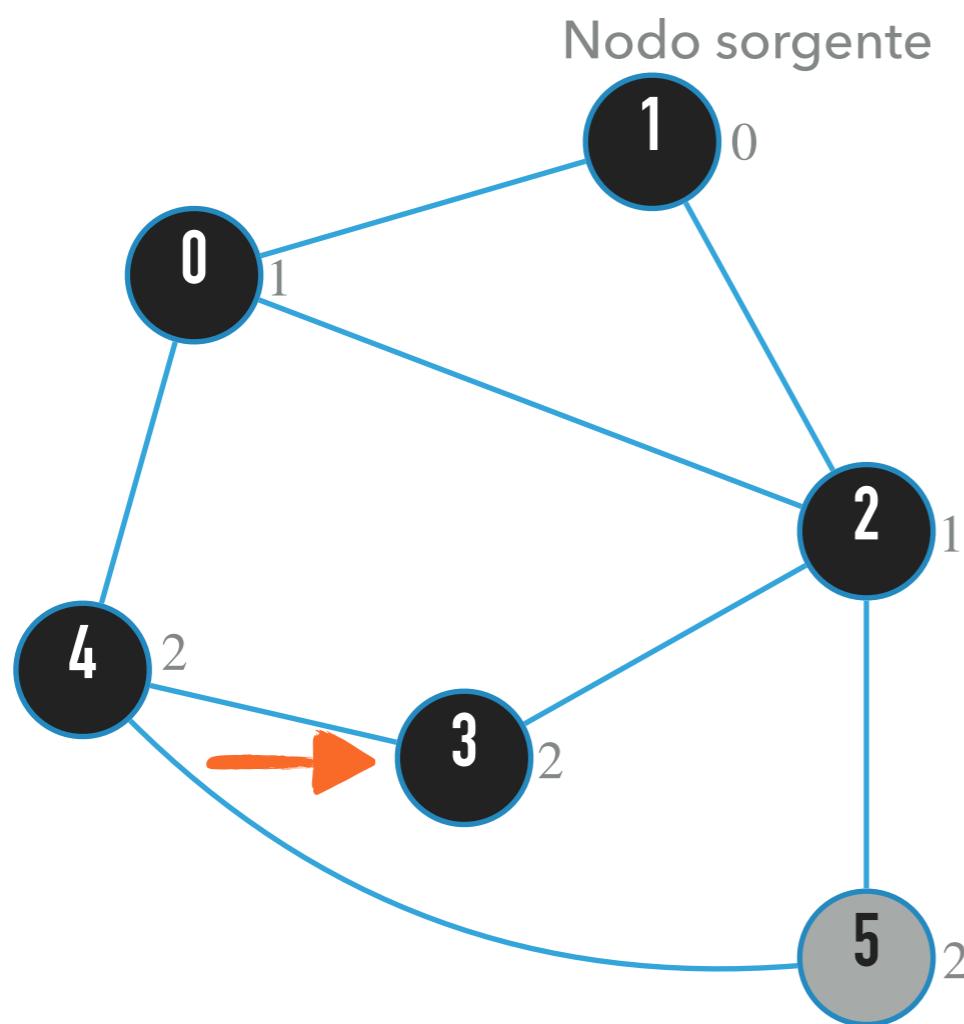
Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio,
aggiorniamo distanza (come $distanza[u] + 1$)
e predecessore (u) e lo accodiamo

Coloriamo u di nero

| | | | | | | |
|--------------|---|---|---|---|---|---|
| Distanza | 0 | 1 | 2 | 3 | 4 | 5 |
| | 1 | 0 | 1 | 2 | 2 | 2 |
| Predecessore | 0 | 1 | 2 | 3 | 4 | 5 |
| | 1 | - | 1 | 2 | 0 | 2 |

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare

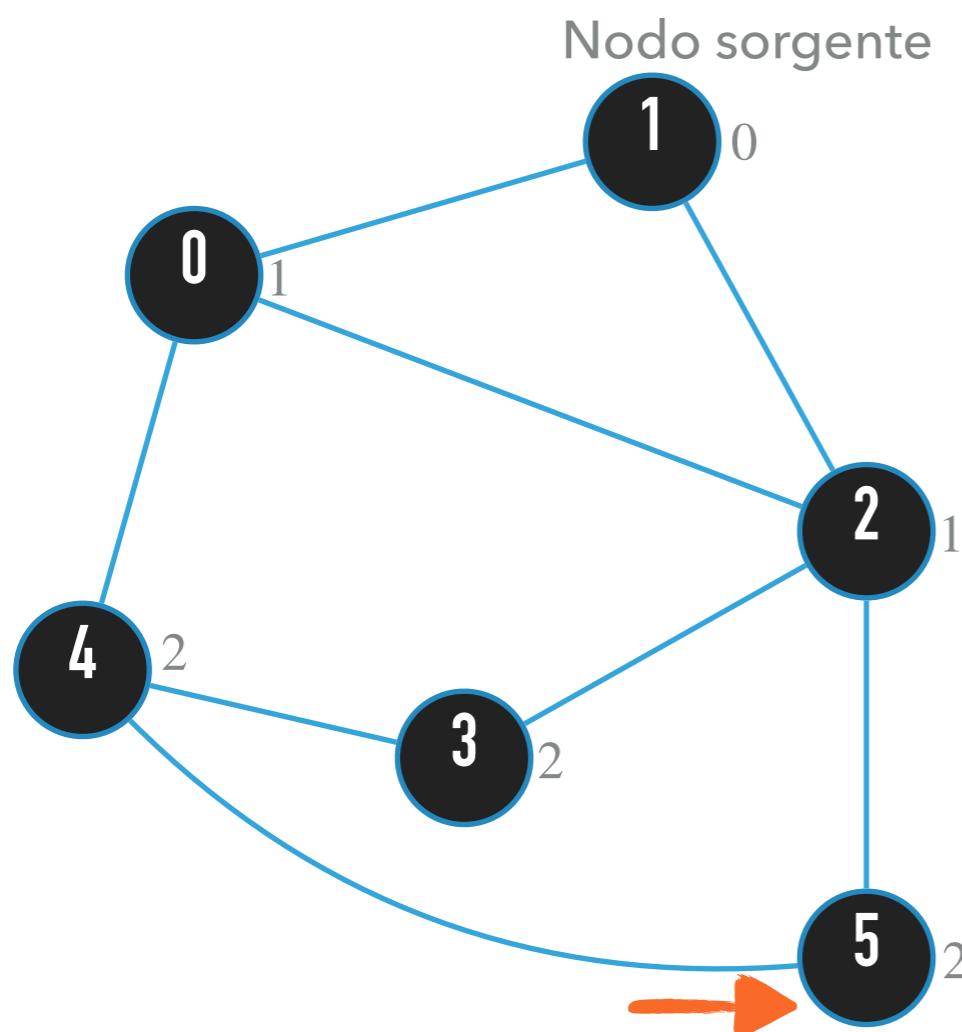
Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio,
aggiorniamo distanza (come $distanza[u] + 1$)
e predecessore (u) e lo accodiamo

Coloriamo u di nero

| Distanza | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| Predecessore | 1 | - | 1 | 2 | 0 | 2 |

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare



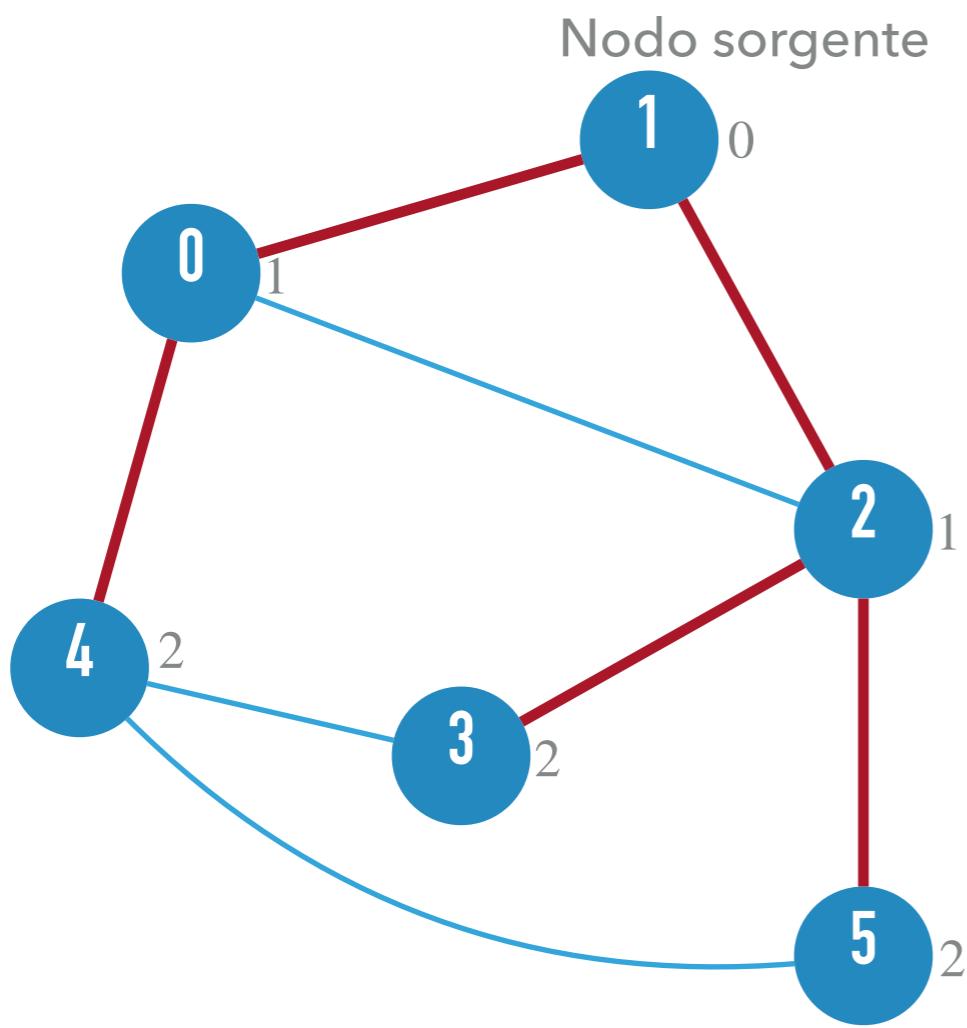
Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio,
aggiorniamo distanza (come $distanza[u] + 1$)
e predecessore (u) e lo accodiamo

Coloriamo u di nero

| Distanza | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| Predecessore | 1 | - | 1 | 2 | 0 | 2 |

COSA ABBIAMO OTTENUTO?



| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 0 | 1 | 2 | 2 | 2 |
| 0 | 1 | 2 | 3 | 4 | 5 |

Distanza

Predecessore

In aggiunta alla distanza, l'array dei predecessori ci permette di ottenere un albero (non necessariamente binario) che è un sottografo del grafo di partenza, detto *albero BFS (BFS tree)*

Per ogni nodo v , l'albero ha un unico percorso dal nodo sorgente a v e questo è un percorso di lunghezza minima

ANALISI DELLA COMPLESSITÀ

- ▶ Assumiamo di utilizzare liste di adiacenza per rappresentare il grafo $G = (V, E)$
- ▶ Le operazioni di inserimento e rimozione dalla coda richiedono $O(1)$
- ▶ Notiamo che ogni nodo può venire accodato al più una volta, perché deve passare da bianco a grigio prima di venire accodato e non tornerà mai più bianco, quindi il ciclo while esegue $O(V)$ volte

ANALISI DELLA COMPLESSITÀ

- ▶ Il ciclo for che itera su tutti i vicini di un nodo è trattabile in un modo un poco particolare. Invece di vedere una singola esecuzione, vediamo il numero totale di volte che viene eseguito
- ▶ Questo numero dipende dal numero di archi del grafo (per andare da un nodo al suo vicino ci serve un arco), quindi è limitato da $O(E)$
- ▶ Il tempo totale di esecuzione è quindi $O(V + E)$

CORRETTEZZA

- ▶ Sia $d[u] = \text{distanza}[u]$ il risultato di BFS. Vogliamo dimostrare che $d[u] = \delta(s, u)$, la distanza in G .
- ▶ Vogliamo anche dimostrare che $\pi[u]$, il predecessore di u in BFS, sia un vertice in un cammino minino da s a u .
- ▶ Vogliamo infine dimostrare che BFS scopre tutti e soli i vertici raggiungibili da s

CORRETTEZZA

- ▶ Lemma: Se $(u, v) \in E$, allora $\delta(s, v) \leq \delta(s, u) + 1$
- ▶ Lemma: Alla fine di BFS, vale che $d[u] \geq \delta(s, u)$
- ▶ Dimostrazione: per induzione sul numero di operazioni e di inserimento nella coda Q.
 - ▶ Caso base: si ha quando si inserisce s.
 - ▶ Passo induttivo: consideriamo il nodo w mentre esaminiamo (u, w)

CORRETTEZZA

- ▶ Lemma: durante BFS, sia $Q = \langle v_1, \dots, v_r \rangle$ la coda in un qualche passo dell'algoritmo. Allora vale:
 - ▶ $d[v_r] \leq d[v_1] + 1$
 - ▶ $\forall i, d[v_i] \leq d[v_{i+1}]$
- ▶ Dimostrazione per induzione sul numero di operazioni sulla coda.

CORRETTEZZA

- ▶ Teorema: correttezza di BFS. Dopo l'esecuzione di BFS:
 - ▶ $\forall v \in V, d[v] = \delta(s, v)$
 - ▶ Se $d[v] < \infty$, uno dei cammini minimi tra s e v passa per $\pi[v]$

CORRETTEZZA

- ▶ Dimostrazione: per assurdo.
- ▶ Sia v il vertice con $\delta(s, v)$ minimo tale che $d[v] > \delta(s, v)$, ovvero che viola il teorema.
- ▶ Necessariamente $\delta(s, v) < \infty$ in quanto vale sempre che $d[v] \geq \delta(s, v)$ e $\delta(s, v) = \infty$ implicherebbe $d[v] = \delta(s, v)$
- ▶ Sia u un vertice che precede v in un cammino minimo da s , allora $d[v] > \delta(s, v) = \delta(s, u) + 1 = d[u] + 1$ per la scelta di v minimale rispetto a $\delta(s, v)$

CORRETTEZZA

- ▶ Consideriamo l'istante in cui u viene rimosso dalla coda, ed il clore di v in quell'istante.
- ▶ Se v è bianco, allora diventa grigio esaminando u , e quindi $d[v] = d[u] + 1$, contraddizione.
- ▶ Se v è grigio, allora è nella coda quando rimuovo u e quindi $d[v] \leq d[u] + 1$, contraddizione.
- ▶ Se v è nero, allora è già stato esaminato prima di u e quindi $d[v] \leq d[u]$, contraddizione.
- ▶ Segue che un tale v non può esistere ed il teorema vale.

RICERCA IN PROFONDITÀ
COMPONENTI FORTEMENTE CONNESSE
ORDINAMENTO TOPOLOGICO

ALGORITMI E STRUTTURE DATI

VISITA IN PROFONDITÀ

VISITA IN PROFONDITA'

RICERCA IN PROFONDITÀ

- ▶ La ricerca in profondità (depth-first search o DFS) è l'altro algoritmo standard di visita dei grafi
- ▶ Dato un grafo $G = (V, E)$ e un nodo $s \in V$ detto nodo sorgente la ricerca in profondità esplora tutti i nodi raggiungibili a partire da s .
- ▶ Se rimangono nodi non esplorati si ripete la ricerca in profondità su di essi*

* questo è fattibile anche con BFS, ma generalmente BFS si usa per trovare la distanza minima da un nodo sorgente, DFS si usa per altri scopi.

RICERCA IN PROFONDITÀ

- ▶ Solitamente DFS salva due tempi:
 - ▶ Il tempo di scoperta $d[v]$ di v , quando il nodo v è stato visitato per la prima volta (quando v diventa grigio)
 - ▶ Il tempo di fine visita $f[v]$ di v , quando tutti i vicini di v sono stati visitati (quando v diventa nero)
- ▶ Questi due tempi sono poi utilizzati da altri algoritmi che hanno DFS come subroutine

RICERCA IN PROFONDITÀ

- ▶ La ricerca in profondità può essere espressa in due modi diversi: ricorsivo e iterativo:
 - ▶ Ricorsivo è come viene solitamente presentata, ed il caso che vedremo.
 - ▶ Una versione iterativa può essere ottenuta sostituendo nella BFS la coda con uno stack.

RICERCA IN PROFONDITÀ: IDEA

- ▶ Dato un nodo $u \in V$
- ▶ Colora il nodo di grigio
- ▶ Se esiste scegli un nodo bianco adiacente e richiama ricorsivamente la visita in profondità su quel nodo
- ▶ Ripeti il punto precedente finché rimangono nodi bianchi
- ▶ Colora il nodo di nero

PSEUDOCODICE: INIZIALIZZAZIONE

Parametri: grafo G

inizialmente impostiamo colore e predecessore per tutti i nodi

for all $v \in V$:

 colore $[v]$ = bianco

 predecessore $[v]$ = None

tempo = 0 # un contatore globale per il tempo di visita dei nodi

for all $u \in V$

 if colore $[u]$ == bianco

 DFS-VISIT(G, u) # chiamiamo la procedura di ricorsiva visita

PSEUDOCODICE: PROCEDURA DFS-VISIT

Parametri: grafo G , nodo u

tempo = tempo + 1 # incrementiamo il tempo globale

tempo_inizio[u] = tempo

colore[u] = grigio

for all v adiacenti a u

 if colore[v] == bianco # se è la prima volta che vediamo v

 preceditore[v] = u # ci siamo arrivati da u

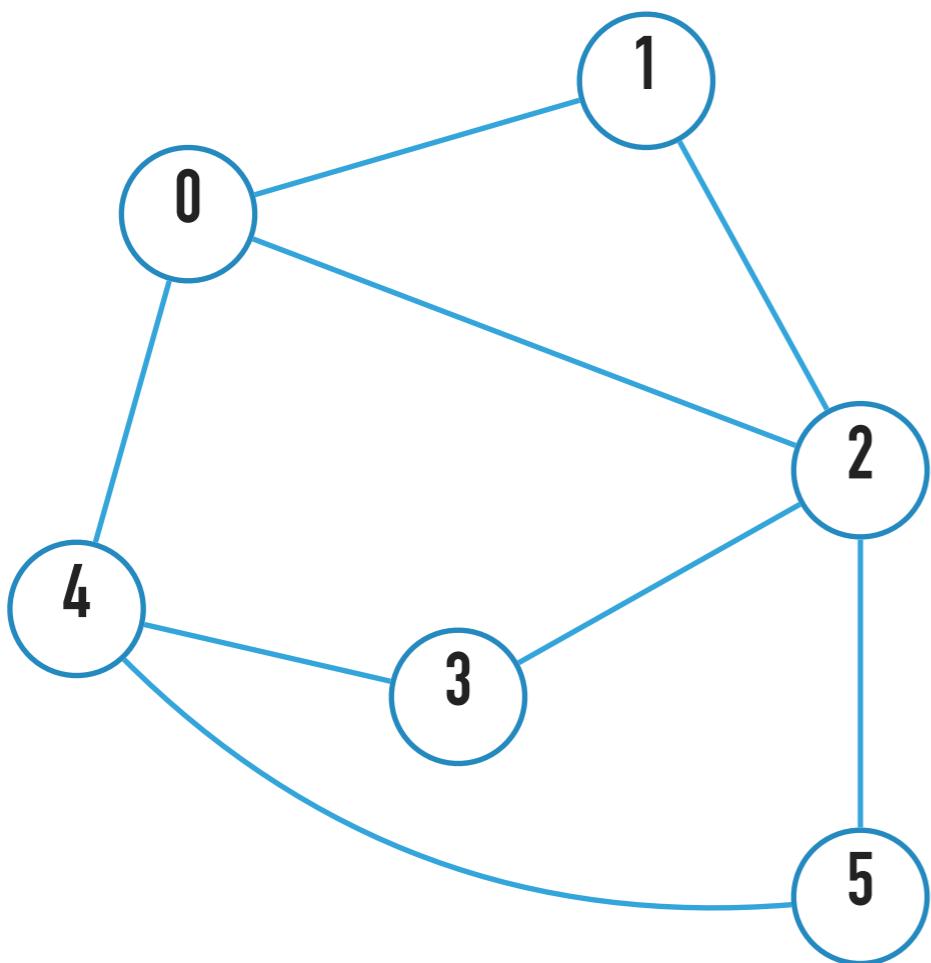
 DFS-VISIT(G , v) # e ricorsivamente iniziamo la procedura di visita

colore[u] = nero # arrivati qui abbiamo visitato tutti i nodi adiacenti a u

tempo = tempo + 1

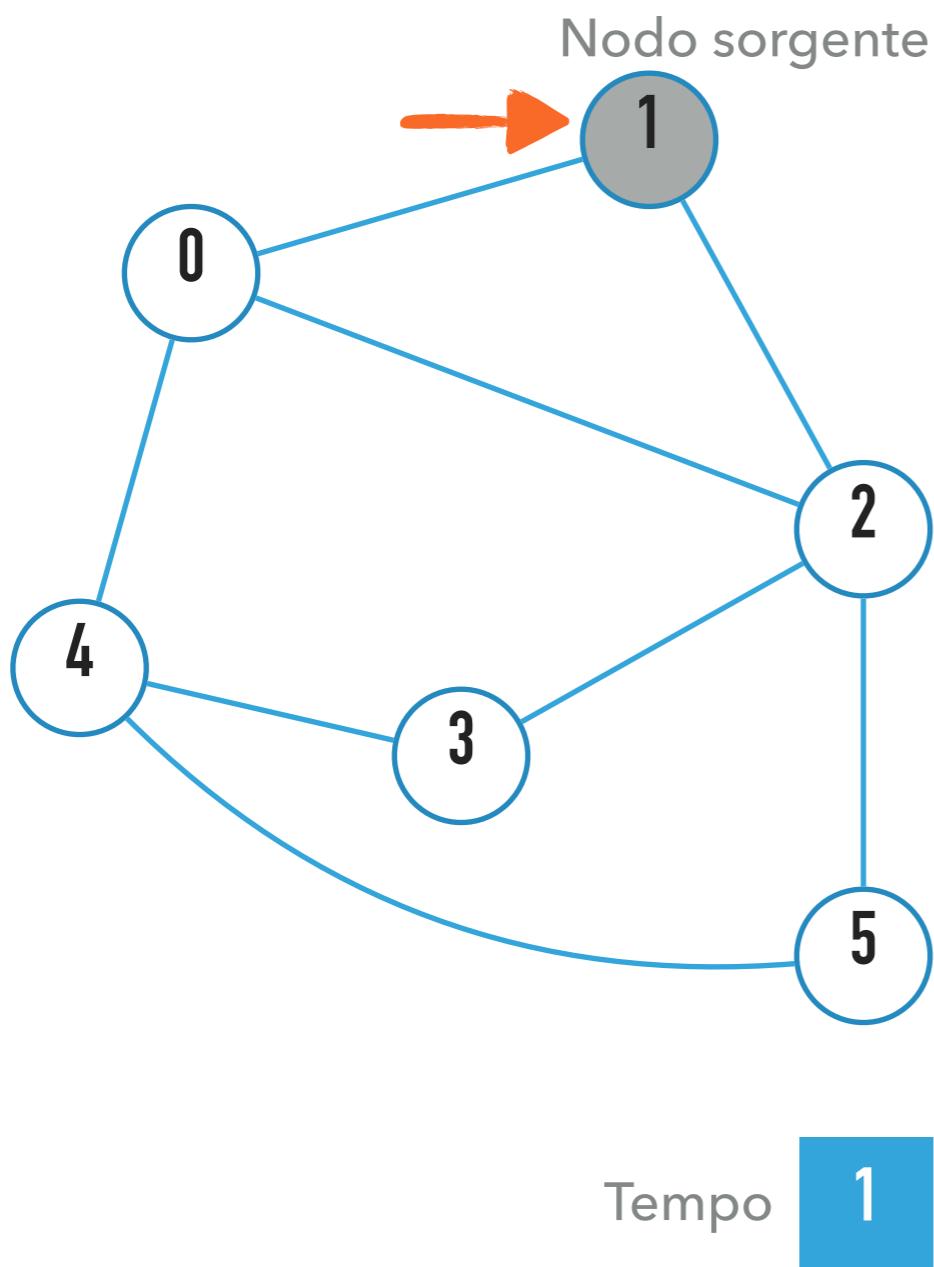
tempo_fine[u] = tempo

ESEMPIO DI ESECUZIONE



Tempo 0

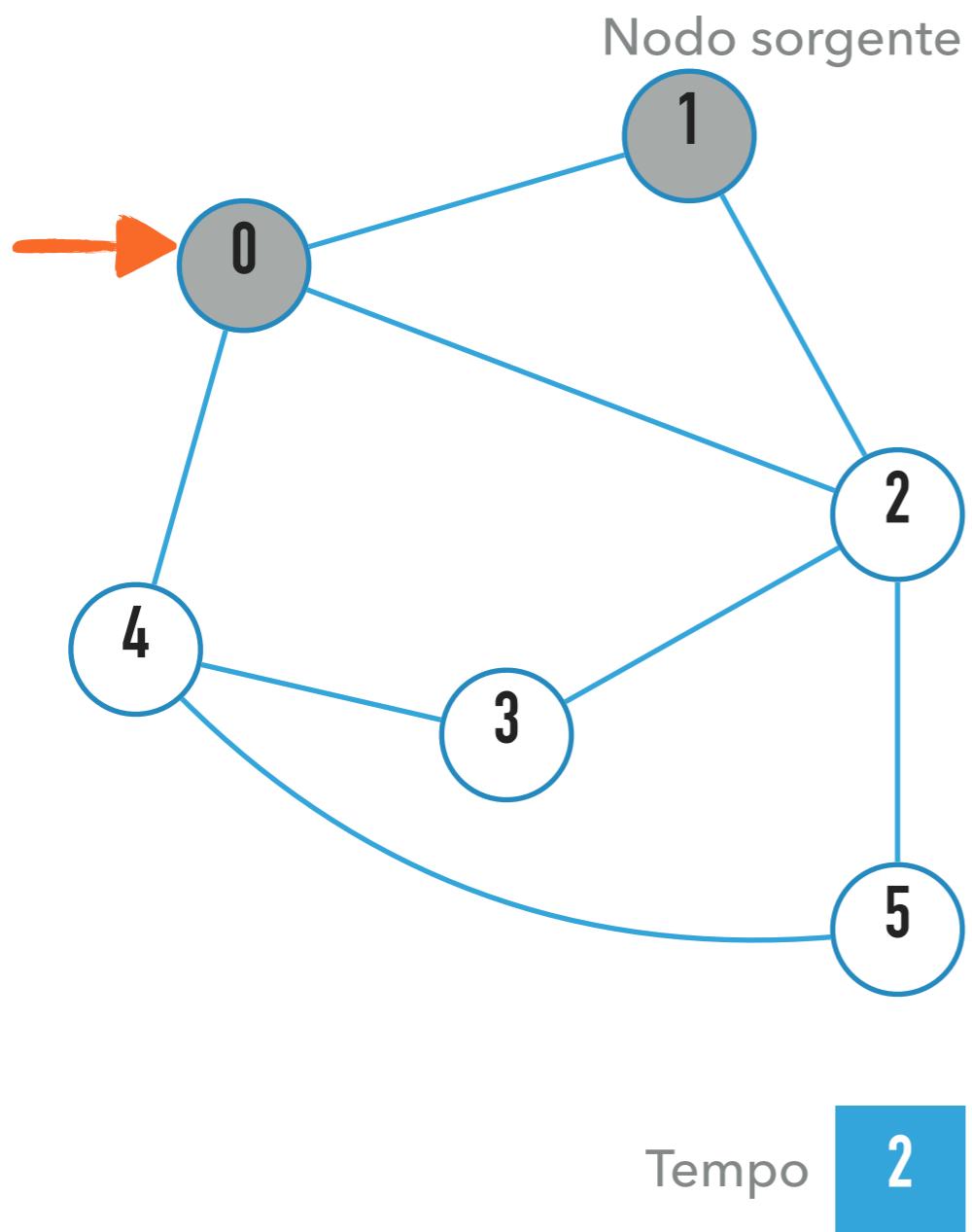
ESEMPIO DI ESECUZIONE



Iniziamo chiamando la procedura di visita sul primo nodo bianco che troviamo e lo coloriamo di grigio



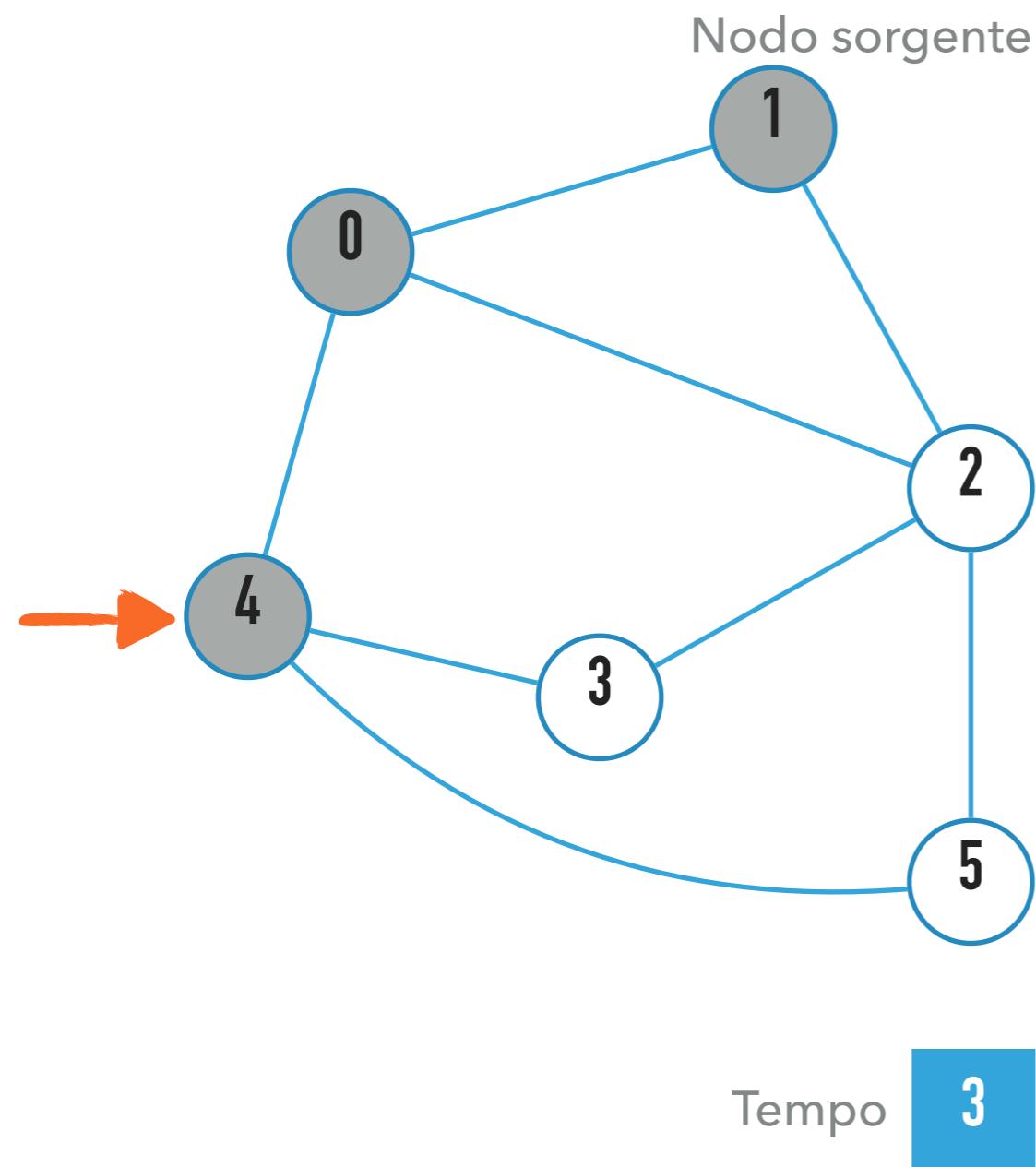
ESEMPIO DI ESECUZIONE



Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente



ESEMPIO DI ESECUZIONE

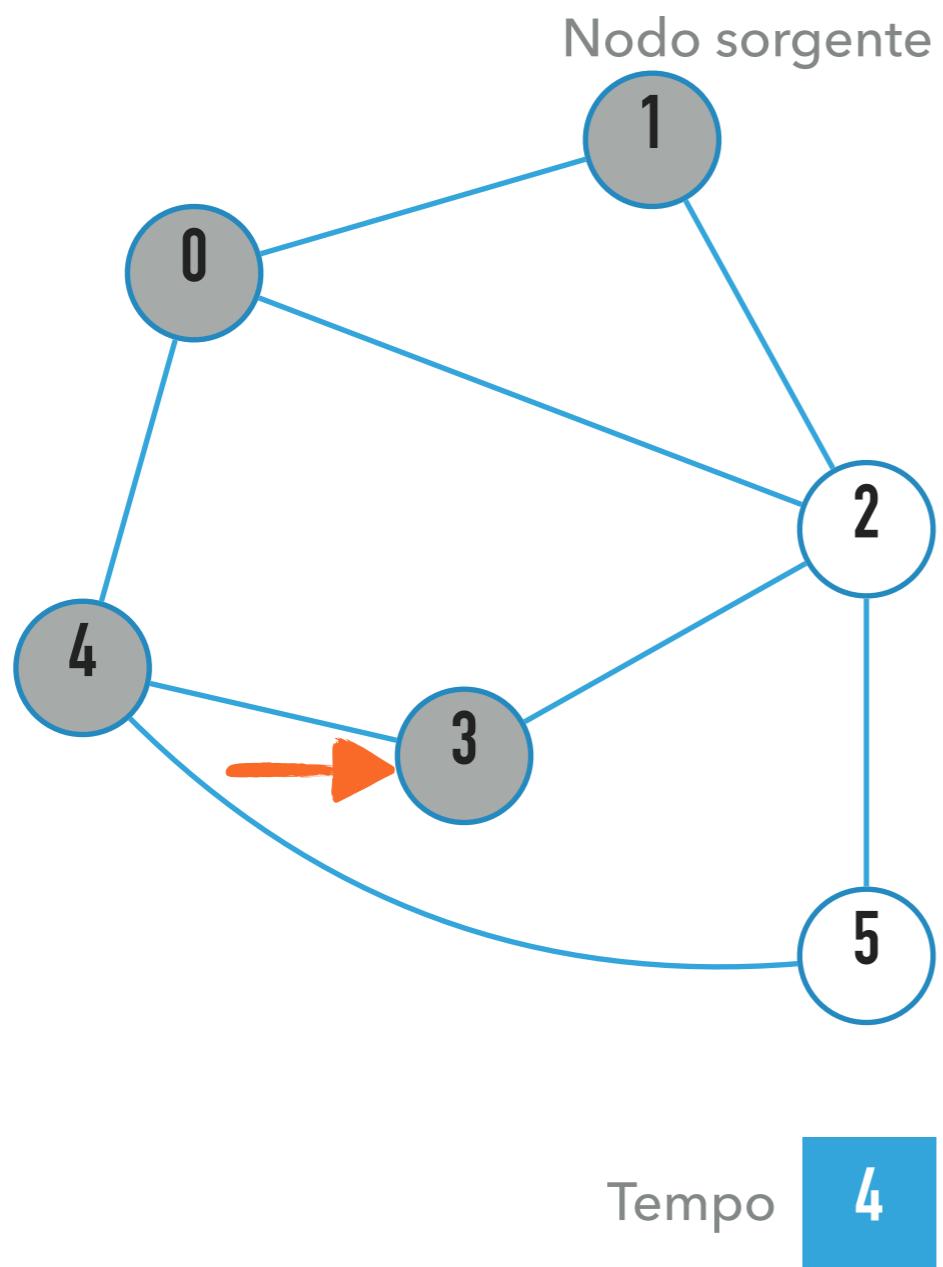


Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| 2 | 1 | | | 3 | 3 | |
| Tempo_FINE | 0 | 1 | 2 | 3 | 4 | 5 |
| | | | | | | |
| Predecessore | 1 | - | - | - | 0 | - |



ESEMPIO DI ESECUZIONE

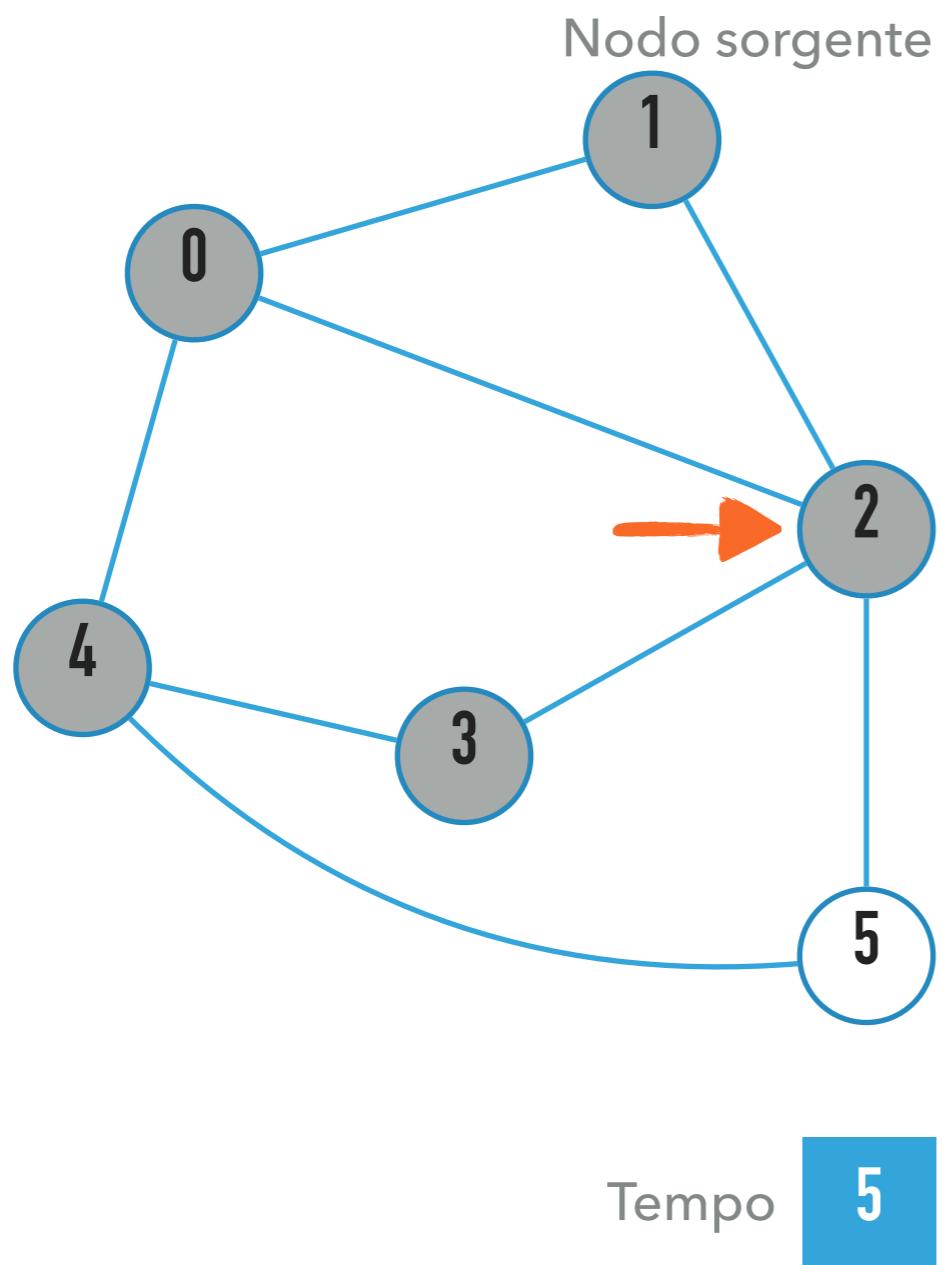


Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| 2 | 1 | | 4 | 3 | | |
| Tempo_fine | 0 | 1 | 2 | 3 | 4 | 5 |
| | | | | | | |
| Predecessore | 1 | - | - | 4 | 0 | - |

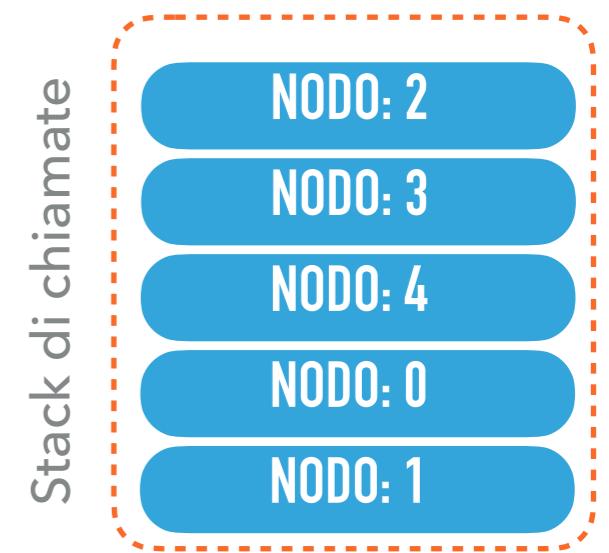


ESEMPIO DI ESECUZIONE



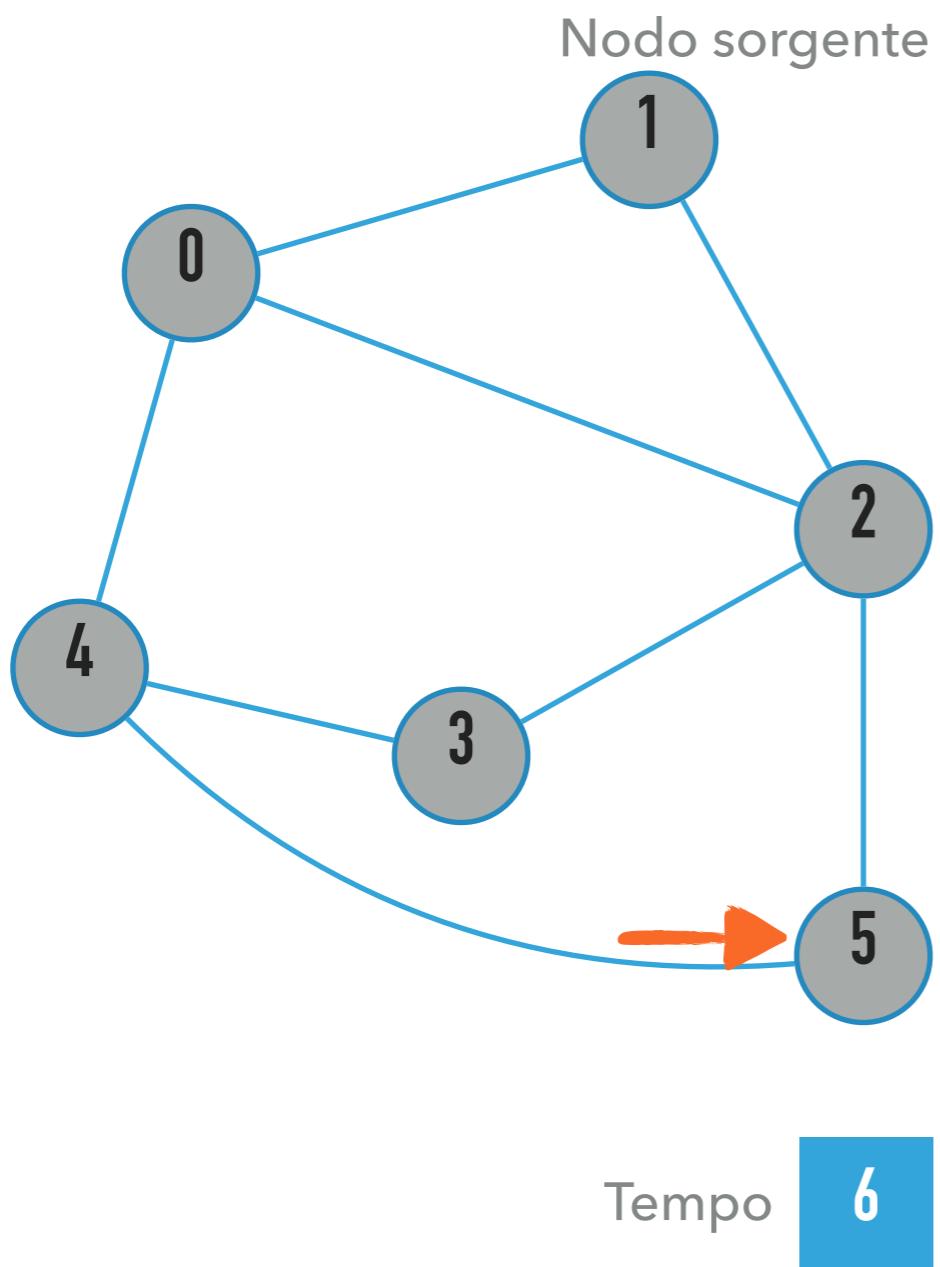
Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| 2 | 1 | 5 | 4 | 3 | | |
| Tempo_fine | 0 | 1 | 2 | 3 | 4 | 5 |
| | | | | | | |
| Predecessore | 1 | - | 3 | 4 | 0 | - |



RICERCA IN PROFONDITÀ

ESEMPIO DI ESECUZIONE



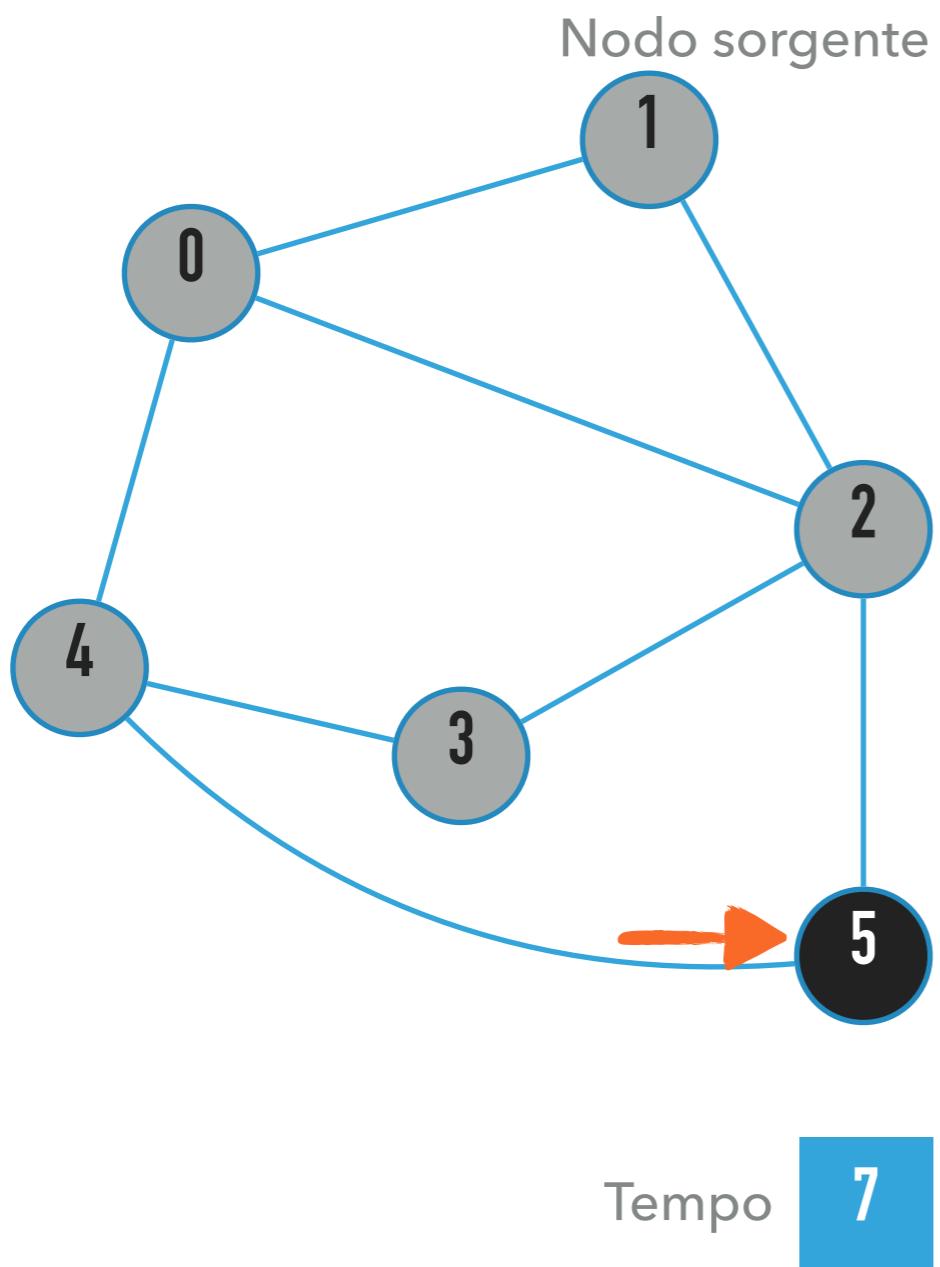
Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| 2 | 1 | 5 | 4 | 3 | 3 | 6 |
| Tempo_fine | 0 | 1 | 2 | 3 | 4 | 5 |
| | | | | | | |
| Predecessore | 1 | - | 3 | 4 | 0 | 2 |



RICERCA IN PROFONDITÀ

ESEMPIO DI ESECUZIONE

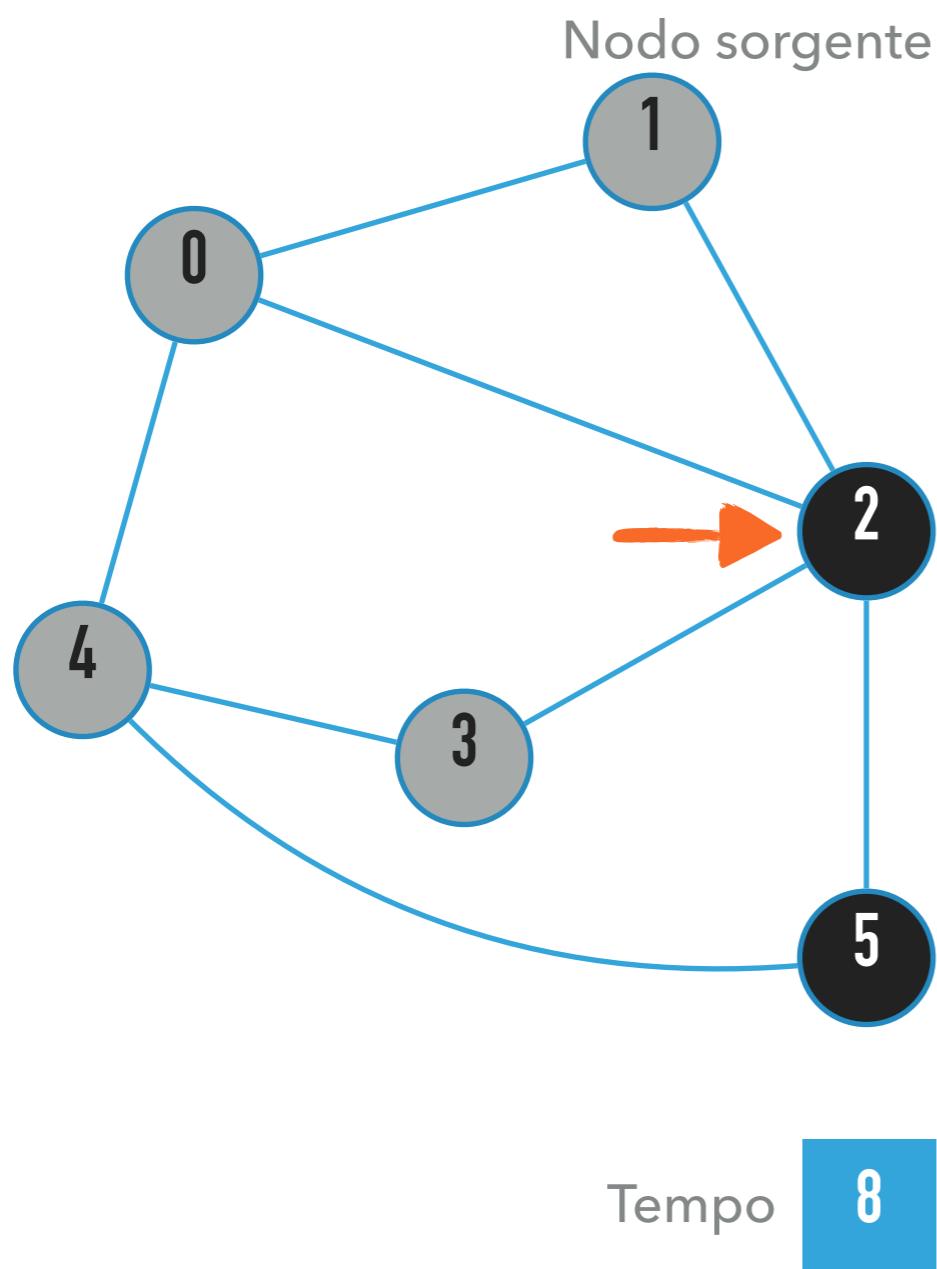


Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| 2 | 1 | 5 | 4 | 3 | 6 | |
| Tempo_fine | 0 | 1 | 2 | 3 | 4 | 5 |
| | | | | | | 7 |
| Predecessore | 1 | - | 3 | 4 | 0 | 2 |



ESEMPIO DI ESECUZIONE

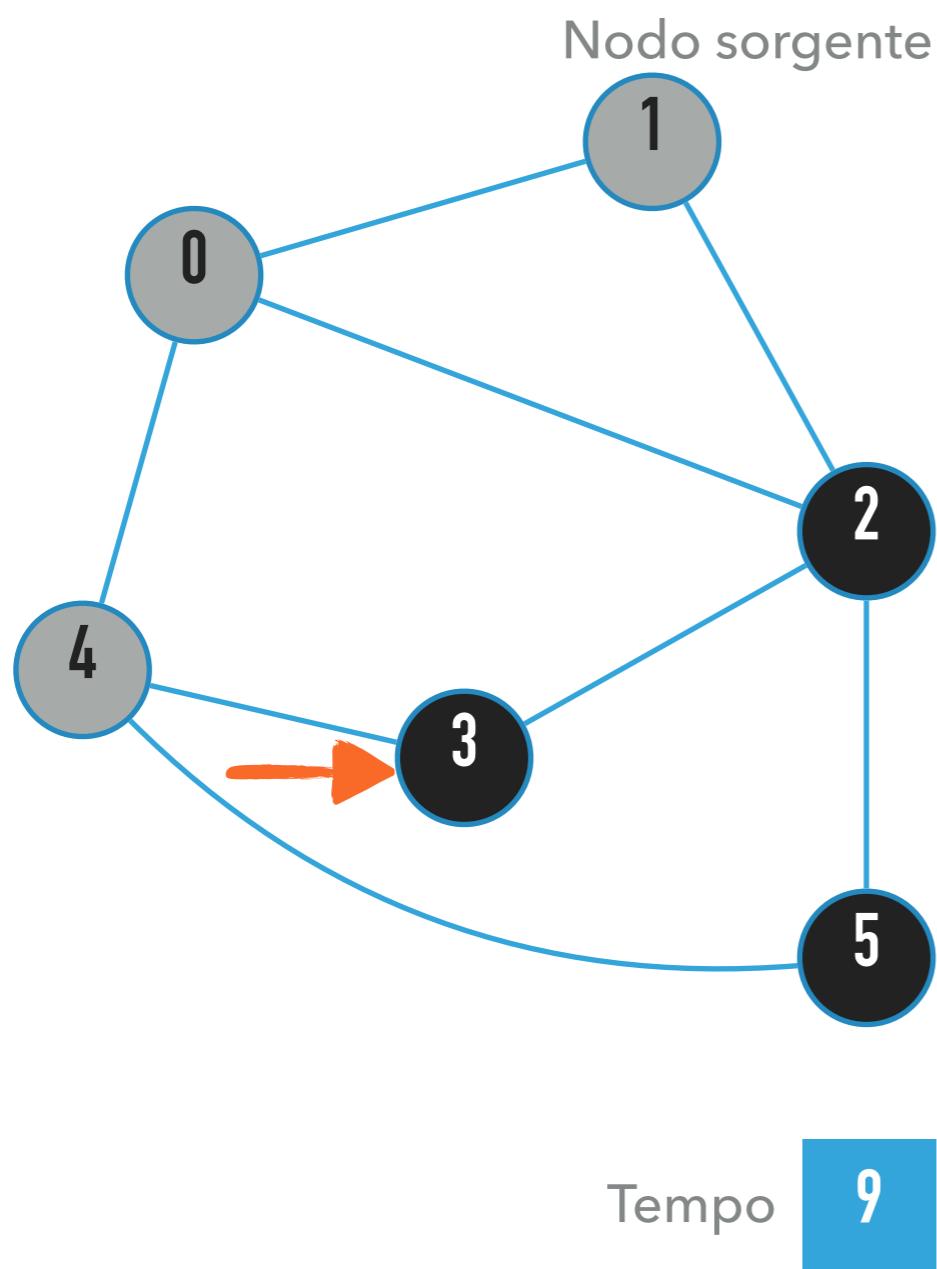


Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| Tempo_fine | 2 | 1 | 5 | 4 | 3 | 6 |
| Predecessore | 1 | - | 3 | 4 | 0 | 2 |



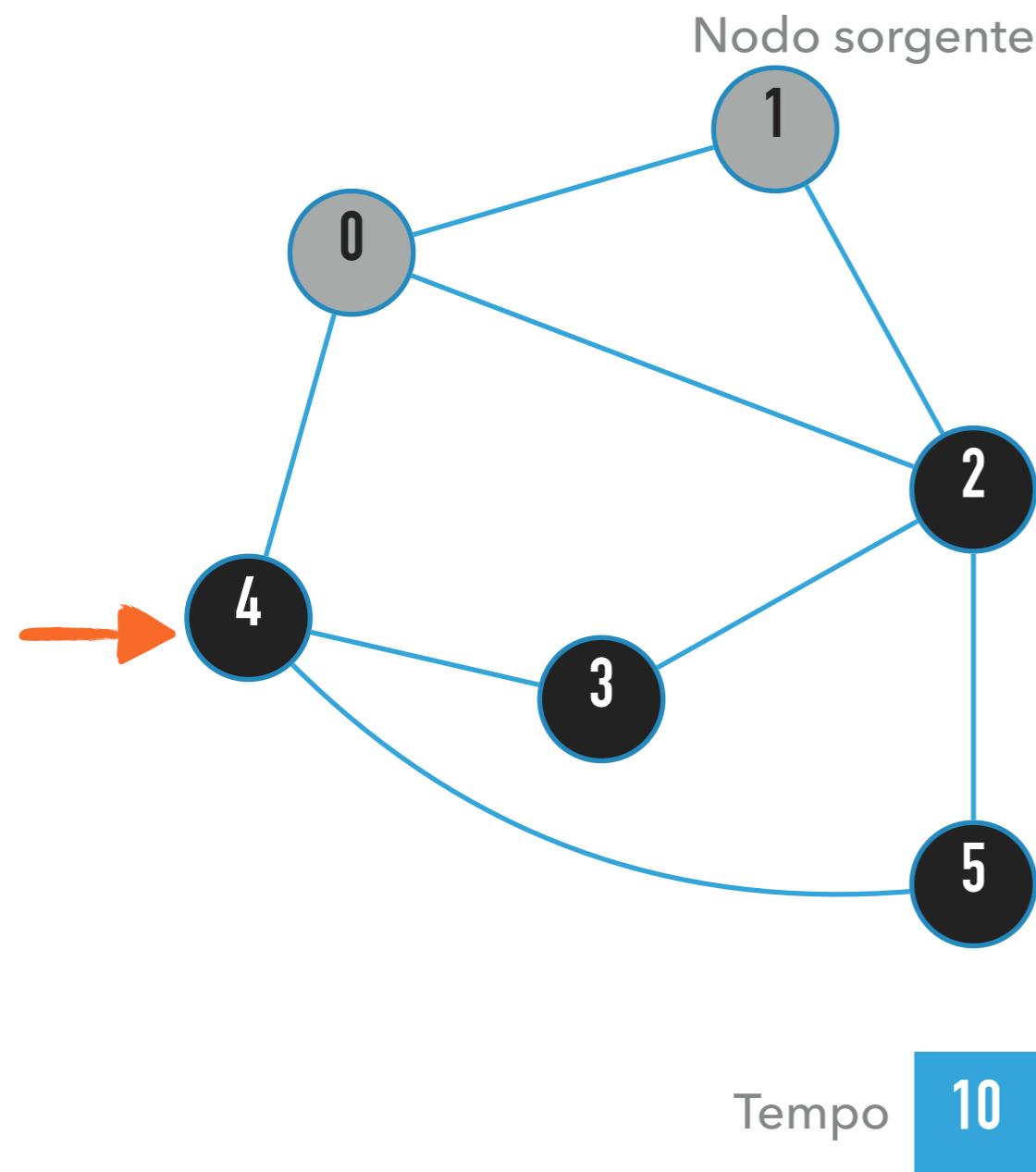
ESEMPIO DI ESECUZIONE



Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|---|---|---|---|---|---|
| Tempo_FINE | 2 | 1 | 5 | 4 | 3 | 6 |
| Predecessore | 1 | - | 3 | 4 | 0 | 2 |

ESEMPIO DI ESECUZIONE



Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

Stack di chiamate

| |
|---------|
| NODO: 4 |
| NODO: 0 |
| NODO: 1 |

Tempo_inizio

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 2 | 1 | 5 | 4 | 3 | 6 |

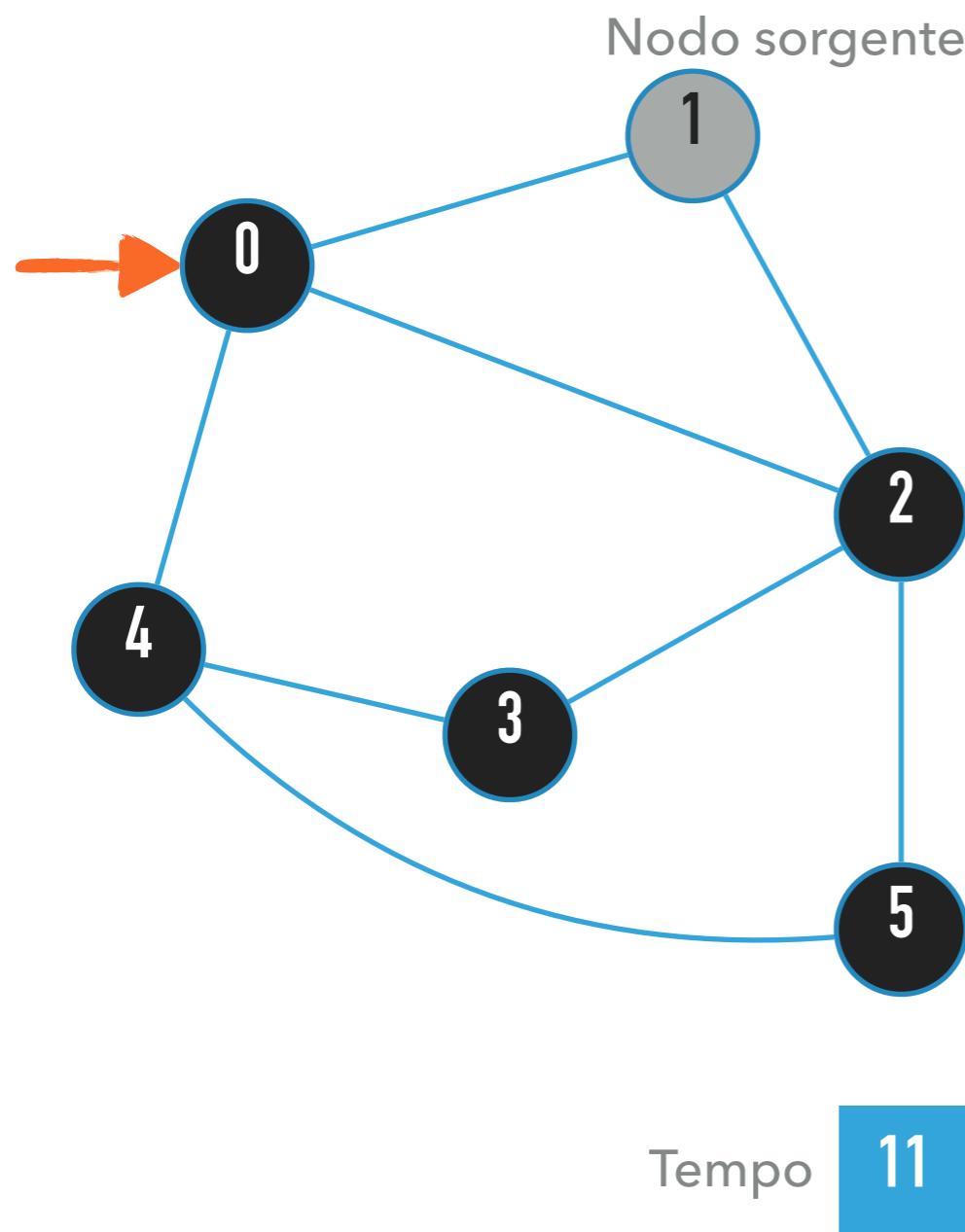
Tempo_fine

| | | | | | |
|---|---|---|---|----|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| | | 8 | 9 | 10 | 7 |

Predecessore

| | | | | | |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | - | 3 | 4 | 0 | 2 |

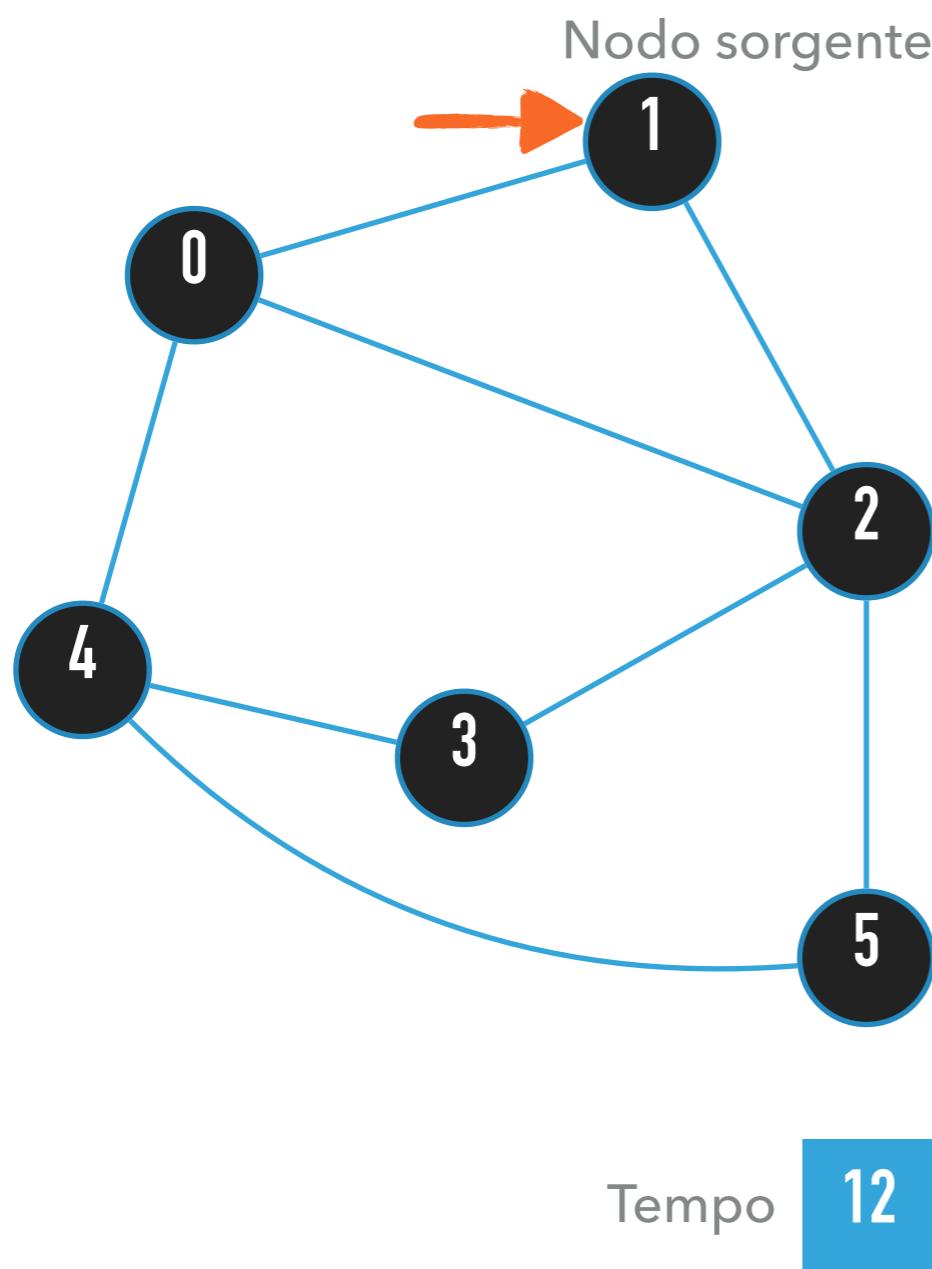
ESEMPIO DI ESECUZIONE



Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

| | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|----|---|---|---|----|---|
| Tempo_inizio | 2 | 1 | 5 | 4 | 3 | 6 |
| Tempo_FINE | 11 | | 8 | 9 | 10 | 7 |
| Predecessore | 1 | - | 3 | 4 | 0 | 2 |

ESEMPIO DI ESECUZIONE

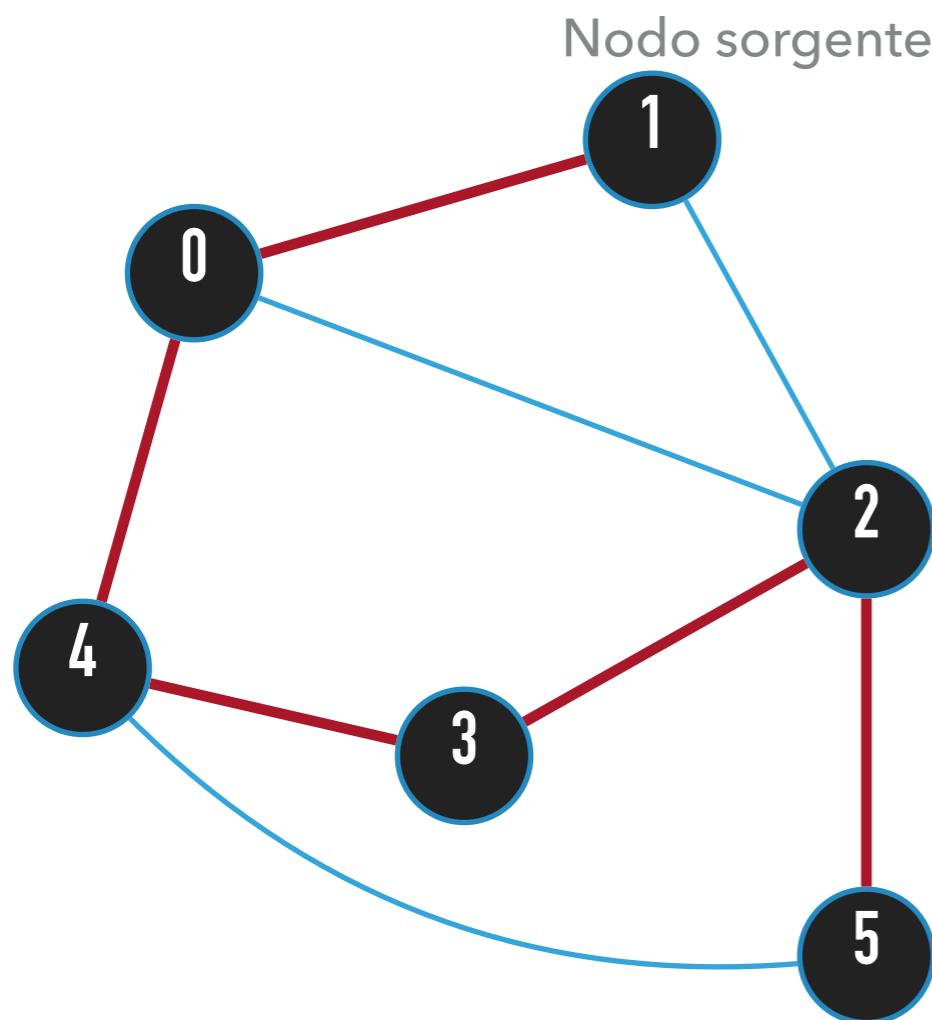


Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

| Tempo_inizio | 0 | 1 | 2 | 3 | 4 | 5 |
|--------------|----|----|---|---|----|---|
| Tempo_FINE | 11 | 12 | 8 | 9 | 10 | 7 |
| Predecessore | 1 | - | 3 | 4 | 0 | 2 |



RISULTATI



Abbiamo ottenuto un albero DFS. Notate come sia diverso dall'albero BFS e come i percorsi su di esso non rappresentino, in generale, il percorso di lunghezza minima dal nodo di partenza

| 0 | 1 | 2 | 3 | 4 | 5 |
|----|----|---|---|----|---|
| 2 | 1 | 5 | 4 | 3 | 6 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 11 | 12 | 8 | 9 | 10 | 7 |
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | - | 3 | 4 | 0 | 2 |

VISITA IN PROFONDITÀ

PROPRIETA DELLA VISITA IN PROFONDITA'

ANALISI DELLA COMPLESSITÀ

- ▶ Notiamo che DFS-VISIT viene chiamata al più una volta per ogni nodo, dato che la chiamata avviene solo se il nodo viene visitato per la prima volta (era bianco) e viene immediatamente ricolorato di grigio, quindi $\Theta(V)$ chiamate a DFS-VISIT
- ▶ Su **tutte** le chiamate a DFS-VISIT, il ciclo che itera sui nodi adiacenti viene eseguito in totale $\Theta(E)$ volte
- ▶ Otteniamo quindi una complessità di $\Theta(V + E)$

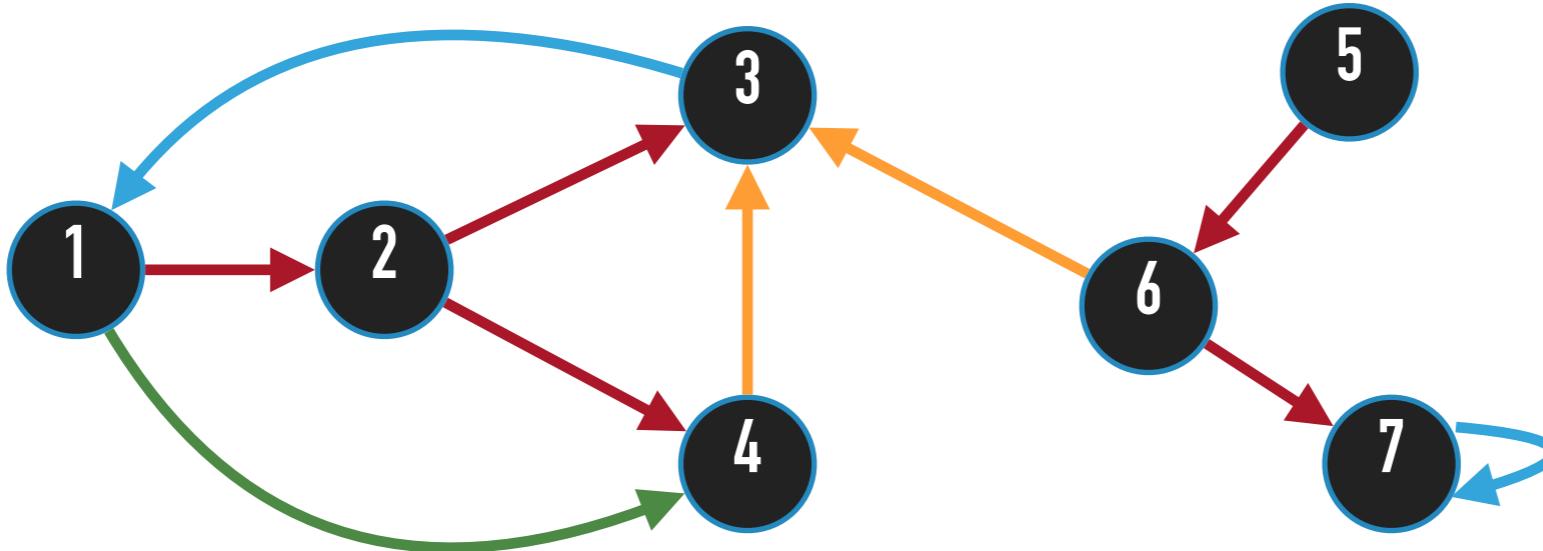
TEMPI DI INIZIO E FINE VISITA: TEOREMA DELLE PARENTESI

- ▶ Sia $G=(V,E)$. Dopo DFS, $\forall u, v \in V$, vale una ed una sola di:
 - ▶ $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$
 - ▶ $[d[u], f[u]] \subset [d[v], f[v]]$
 - ▶ $[d[v], f[v]] \subset [d[u], f[u]]$
- ▶ **Dimostrazione:** assumiamo $d[u] < d[v]$. Ci sono due casi:
 - ▶ $d[v] < f[u]$, ergo v è scoperto mentre u è grigio, e diventa discendente di u , e la sua visita finisce prima: $f[v] < f[u]$
 - ▶ $f[u] < d[v]$, ergo siamo nel primo caso.
- ▶ **Corollario:** $[d[v], f[v]] \subset [d[u], f[u]]$ se e solo se v è discendente di u nel DFS-tree

TEOREMA DEL CAMMINO BIANCO

- ▶ Sia $G=(V,E)$ su cui eseguiamo DFS. Siano $u, v \in V$.
 v è discendente di u nel DFS-tree se e solo se all'istante $d[u]$ esiste un cammino da u a v fatto solo di nodi bianchi.
- ▶ **Dimostrazione:**
 - ▶ \Rightarrow : Tutti i nodi w nel cammino nel DFS-tree da u a v sono tali che $d[w] > d[u]$ ergo al tempo $d[u]$ sono bianchi.
 - ▶ \Leftarrow : Per assurdo, dato un cammino di nodi bianchi da u , sia v il primo nodo a non essere discendente di u in tale cammino, e w il predecessore. Allora w è discendente di u , e v sarà visitato prima che w diventi nero. Quindi $f[v] < f[w] < f[u]$ e $d[v] > d[w] > d[u]$. Quindi v è discendente di u .

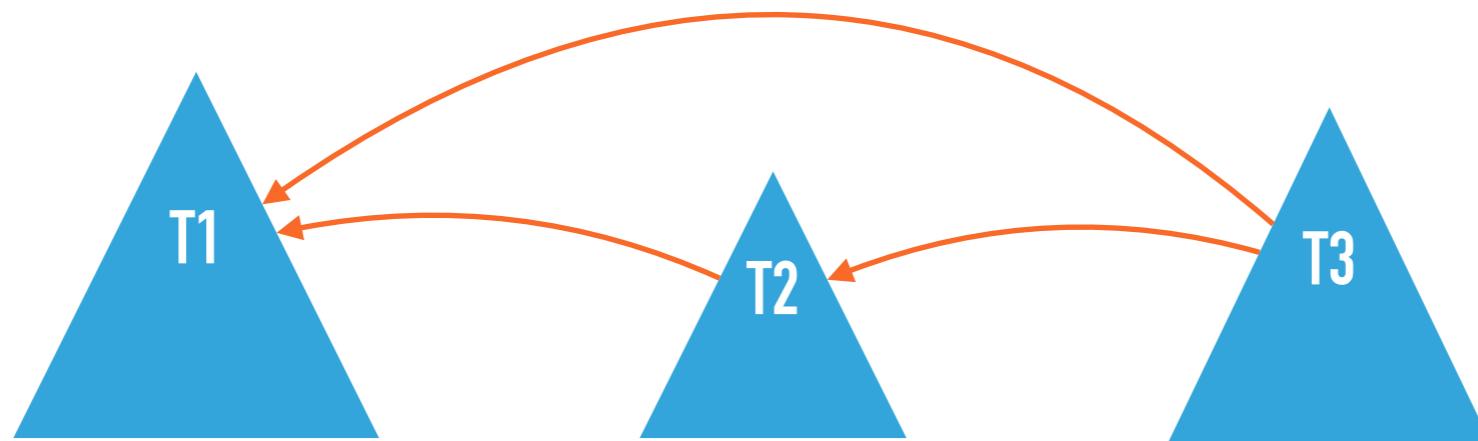
CLASSIFICAZIONE DEGLI ARCHI



- **Archi dell'albero** (u,v): $\text{color}[v]=\text{bianco}$ durante l'esame di (u,v)
- **Archi all'indietro** (u,v): connettono un vertice u ad un suo antenato v nell'albero - $\text{color}[v]=\text{grigio}$ durante $\text{DFS-VISIT}(u)$
- **Archi in avanti** (u,v): $\text{color}[v]=\text{nero}$ e $d[v] > d[u]$, quindi v è discendente di u nell'albero DFS.
- **Archi di attraversamento** (u,v): $\text{color}[v]=\text{nero}$ e $d[v] < d[u]$. L'arco collega parti diverse dell'albero o alberi DFS diversi della foresta DFS.

FORESTA DFS

- ▶ L'esecuzione di DFS può generare un certo numero di alberi T_j disgiunti (una per ogni chiamata di DFS-VISIT da DFS).
- ▶ Gli archi di attraversamento possono andare solo da T_j a T_i per $i < j$.



AMPIEZZA VS PROFONDITÀ

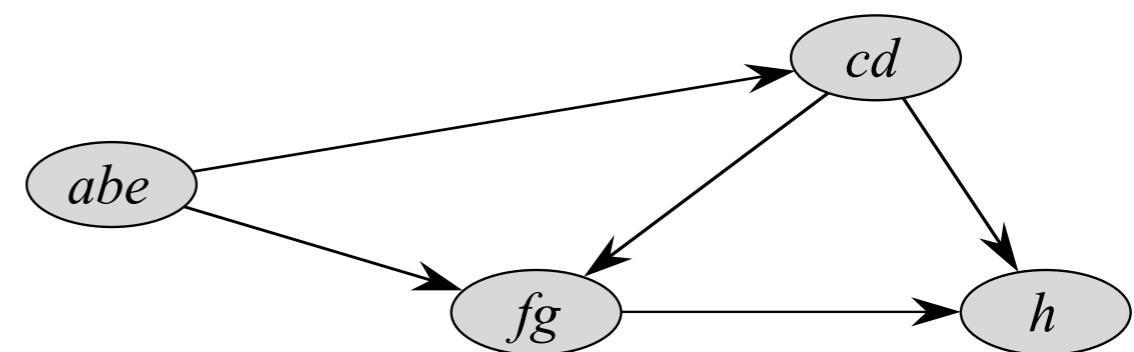
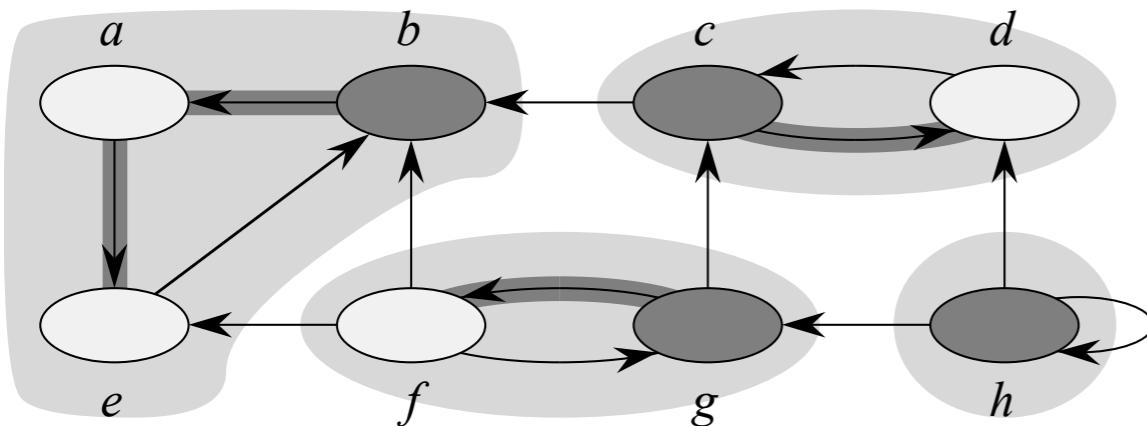
- ▶ La scelta di una o dell'altra tipologia di visita dipende dalle specifiche dell'algoritmo
- ▶ Per trovare il percorso di lunghezza minima? BFS
- ▶ Spesso per grafi diretti si utilizza DFS
- ▶ Notate come in BFS la coda sia esplicita, mentre in DFS lo stack è implicitamente fornito dalle chiamate ricorsive

COMPONENTI FORTEMENTE CONNESSE

COMPONENTI FORTEMENTE CONNESSE

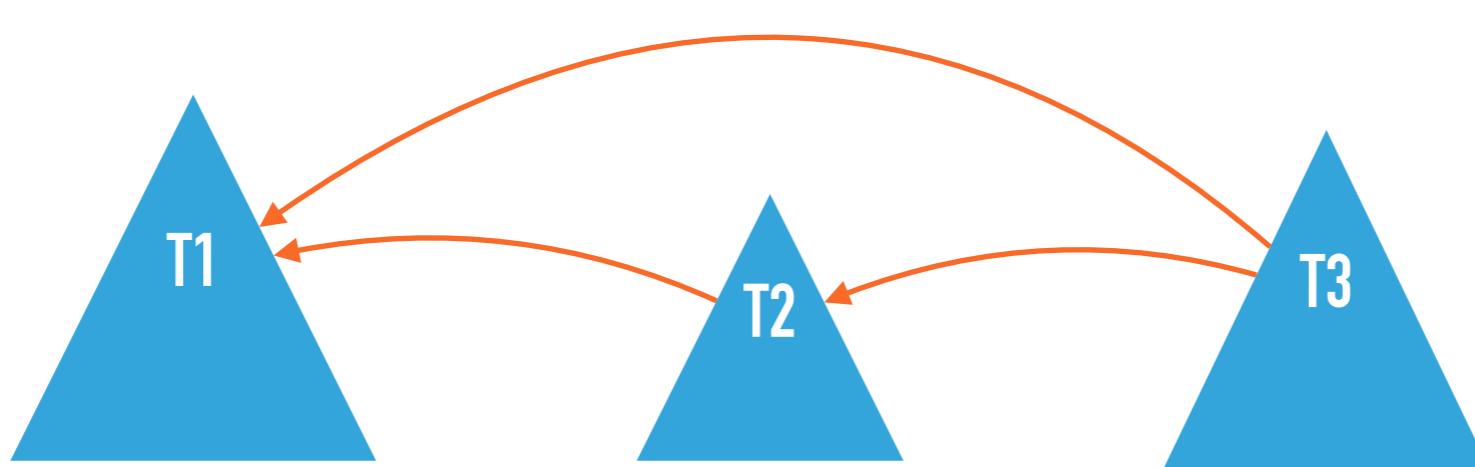
COMPONENTI FORTEMENTE CONNESSE (GRAFI ORIENTATI)

- ▶ Diciamo che v è raggiungibile da u se esiste un cammino da u a v : $u \longrightarrow v$.
- ▶ Se u è raggiungibile da v e v è raggiungibile da u , diciamo che u e v sono **mutualmente raggiungibili**: $u \longleftrightarrow v$ sse $u \longrightarrow v$ e $v \longrightarrow u$
- ▶ $u \longleftrightarrow v$ è una relazione di equivalenza. Le sue classi di equivalenza sono le **componenti fortemente connesse** del grafo.
- ▶ Il grafo delle componenti fortemente connesse (scc-graph) ha un nodo per ogni scc di G , ed un arco tra C_1 e C_2 sse c'è un arco in G da un nodo u di C_1 ad un nodo v di C_2 . Il **scc-graph** è aciclico. Perchè?



COMPONENTI FORTEMENTE CONNESSE (GRAFI ORIENTATI)

- ▶ Il grafo trasposto di $G=(V,E)$ è $G^T=(V,E^T)$, con
$$E^T = \{(v, u) \mid (u, v) \in E\}$$
- ▶ Se $u \longleftrightarrow v$ in G allora $u \longleftrightarrow v$ anche in G^T . Le componenti fortemente connesse di G e G^T sono le stesse.
- ▶ Ogni componente fortemente connessa è tutta contenuta in un singolo albero DFS T_j di G e in un singolo albero DFS T_i di G^T .

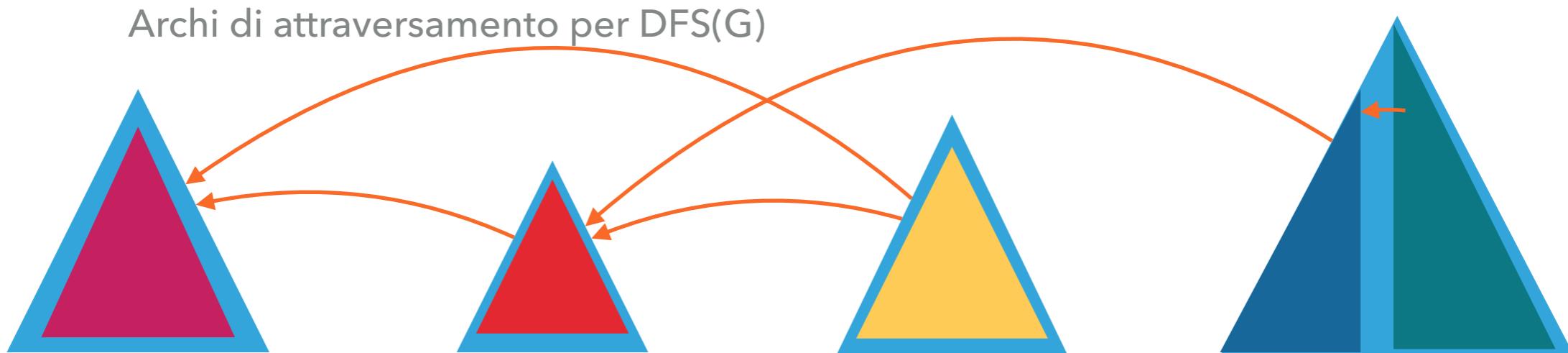


COMPONENTI FORTEMENTE CONNESSE (GRAFI ORIENTATI)

- ▶ Siano C_1 e C_2 due componenti fortemente connesse, e sia $(u, v) \in E$ un arco di G , con $u \in C_1, v \in C_2$.
Dopo l'esecuzione di $\text{DFS}(G)$, vale $f[C_1] > f[C_2]$, dove $f[C] = \max\{f[v] \mid v \in C\}$.
- ▶ Dimostrazione: si considera il caso in cui viene prima scoperta la componente C_1 (i nodi di C_2 diventano tutti discendenti del primo nodo di C_1 scoperto per teorema del cammino bianco) ed il caso in cui si scopre prima C_2 (C_1 è visitata quando tutti i nodi di C_2 sono neri).
- ▶ Siano C_1 e C_2 due componenti fortemente connesse, e sia $(u, v) \in E^T$ un arco di G^T , con $u \in C_1, v \in C_2$.
Dopo l'esecuzione di $\text{DFS}(G)$, vale $f[C_1] < f[C_2]$.
- ▶ Se lancio $\text{DFS}(G^T)$ e parto prima dalla componente C_2 come subito sopra, la componente C_2 e la componente C_1 staranno in alberi DFS diversi della foresta DFS.

ALGORITMO DI TARJAN PER LE COMPONENTI FORTEMENTE CONNESSE

- ▶ Tarjan-SCC(G)
 - DFS(G), memorizza i tempi di fine visita $f[v]$
 - calcola G^T
 - DFS(G^T), in ordine decrescente di $f[v]$
 - return** alberi DFS di G^T
- ▶ Correttezza: per le proprietà del scc-graph, chiamando DFS su G^T in ordine decrescente di $f[v]$ di DFS(G), le componenti connesse vengono scoperte dalle foglie alle radici nel scc-graph di G^T , e quindi ognuna finisce in un albero DFS diverso. Ma ogni albero contiene almeno una scc.

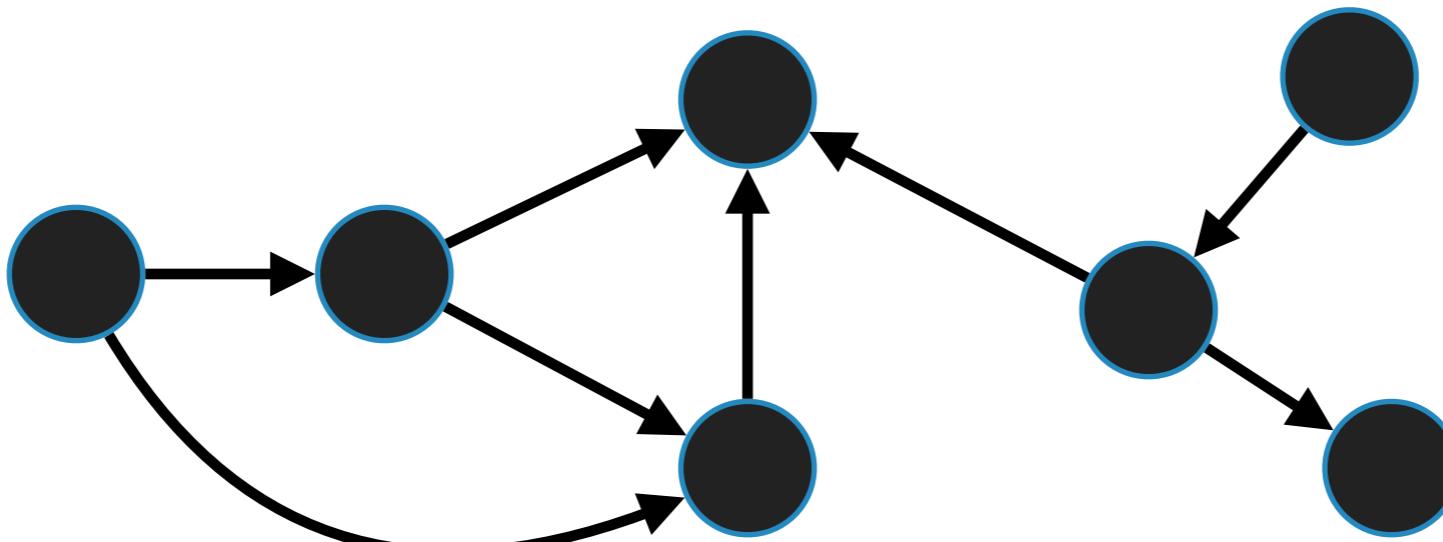


ORDINAMENTO TOPOLOGICO

ORDINAMENTO TOPOLOGICO

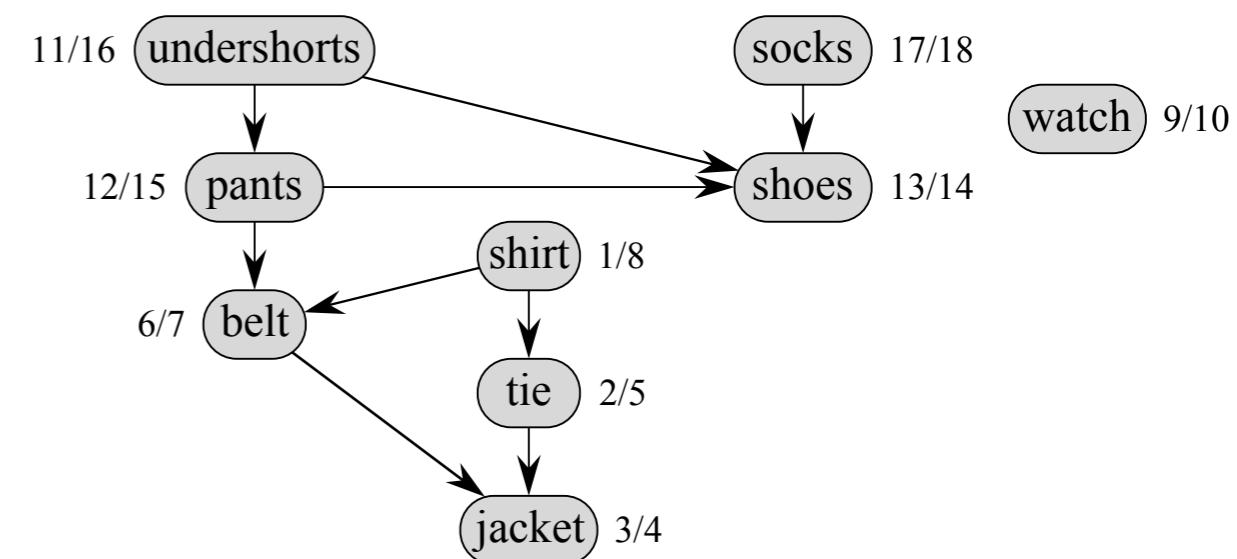
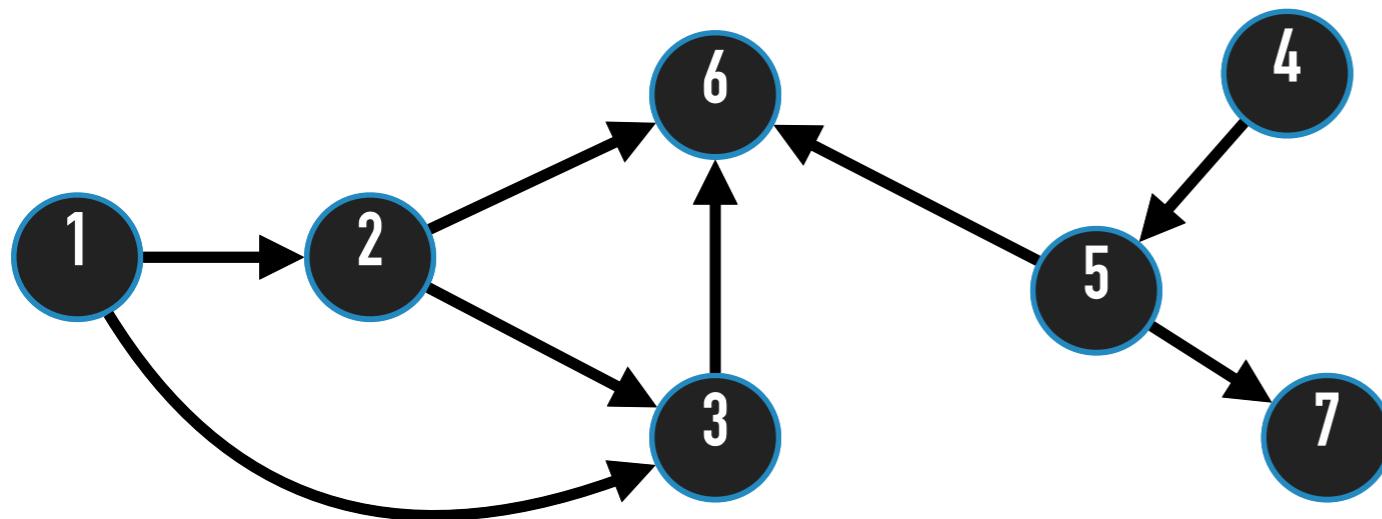
GRAFI ORIENTATI ACICLICI (DAG)

- ▶ Un grafo diretto è aciclico se non ha cicli (cammini semplici che iniziano e terminano nello stesso vertice). Tutti gli alberi sono aciclici, ma ci sono grafi aciclici che non sono alberi.
- ▶ Teorema: un grafo diretto è aciclico sse DFS non produce archi all'indietro.
 - ▶ \Rightarrow : (contronominale) Sia (u,v) arco all'indietro. Quindi u è discendente di v nel DFS-tree. Otteniamo un ciclo.
 - ▶ \Leftarrow (contronominale) sia c un ciclo e v il primo vertice del ciclo scoperto, w il suo predecessore nel ciclo. Usare il teorema del cammino bianco per mostrare che si ottiene un arco all'indietro.



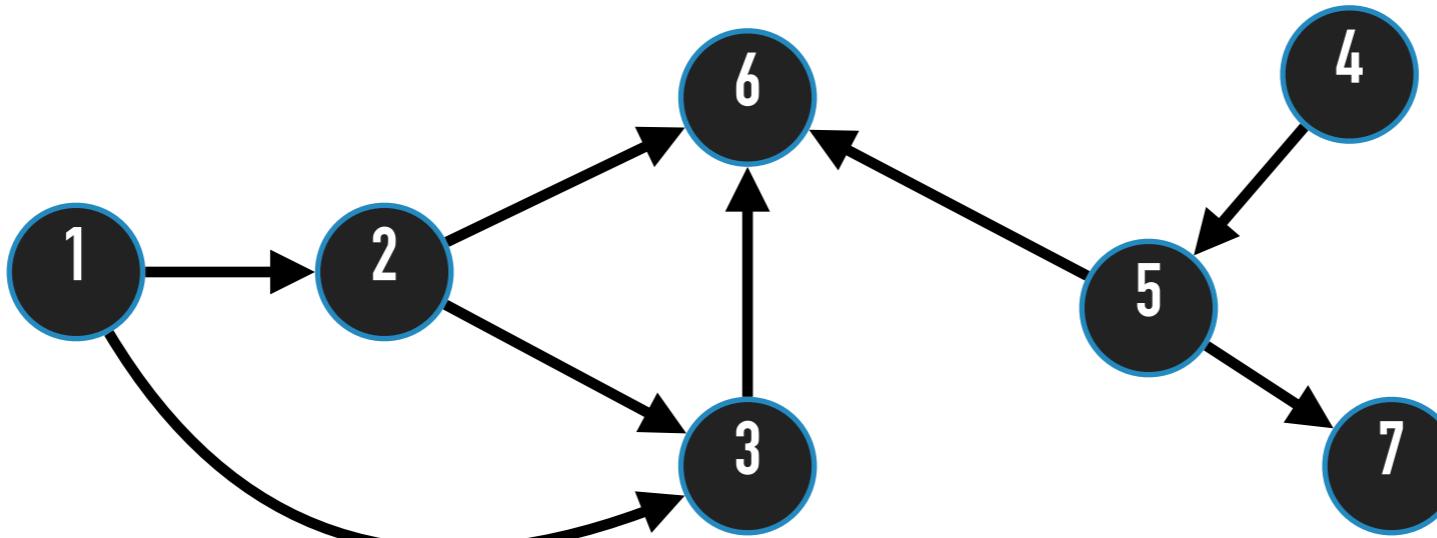
ORDINAMENTO TOPOLOGICO

- ▶ Dato un grafo aciclico, un **ordinamento topologico** è un ordinamento dei vertici tale che, per ogni arco (u,v) , u precede v nell'ordinamento.
- ▶ Topological-Sort(G)
 $\text{DFS}(G)$
return lista dei vertici v di G in ordine decrescente di $f[v]$



ORDINAMENTO TOPOLOGICO

- ▶ Topological-Sort(G)
DFS(G)
return lista dei vertici v di G in ordine decrescente di $f[v]$
- ▶ Correttezza: per ogni $(u, v) \in E$, mostriamo $f[v] < f[u]$.
- ▶ Quando esaminiamo (u, v) , v non può essere grigio (no archi all'indietro). Se v è bianco, diventa discendente di u e quindi $f[v] < f[u]$, se v è nero, $f[v] < d[u] < f[u]$.



ALGORITMO DI BELLMAN-FORD
ALGORITMO DI DIJKSTRA

ALGORITMI E STRUTTURE DATI



Dijkstra

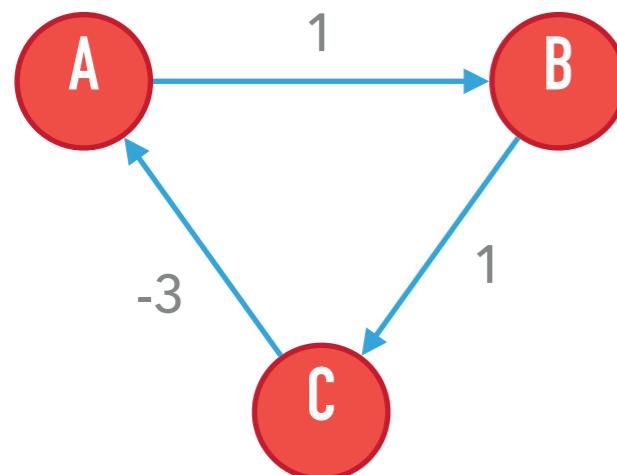
IL PROBLEMA CHE VOGLIAMO RISOLVERE

- ▶ Il peso di un percorso $p = (v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ è la somma dei pesi di tutti gli archi: $w(p) = \sum_{i=1}^n w_{v_i, v_j}$
- ▶ Assumiamo di volere solo percorsi **semplici**, ovvero privi di cicli (ogni nodo al più una volta in ogni percorso)
- ▶ Dato un grafo si vuole trovare il percorso **di peso minore** tra un vertice di partenza s e tutti gli altri vertici (o verso un vertice di destinazione d)

IL PROBLEMA CHE VOGLIAMO RISOLVERE

- ▶ Il grafo può essere orientato o non orientato
- ▶ A seconda delle assunzioni che facciamo sui pesi possiamo utilizzare algoritmi diversi:
 - ▶ I pesi possono essere negativi:
algoritmo di Bellman-Ford
 - ▶ I pesi sono solo non negativi:
algoritmo di Dijkstra

IL CASO DA EVITARE: CICLI NEGATIVI



Quale è il percorso meno costoso per andare da A a sé stesso?

Intuitivamente diremmo "il percorso vuoto" che ha costo zero

Ma il percorso $(A,B),(B,C),(C,A)$ ha peso totale -1

Quindi non esiste un percorso più corto, dato che possiamo sempre accorciare facendo "un altro giro"

Assumiamo assenza di cicli di peso negativo

OPERAZIONE DI RILASSAMENTO/RELAX DEGLI ARCHI

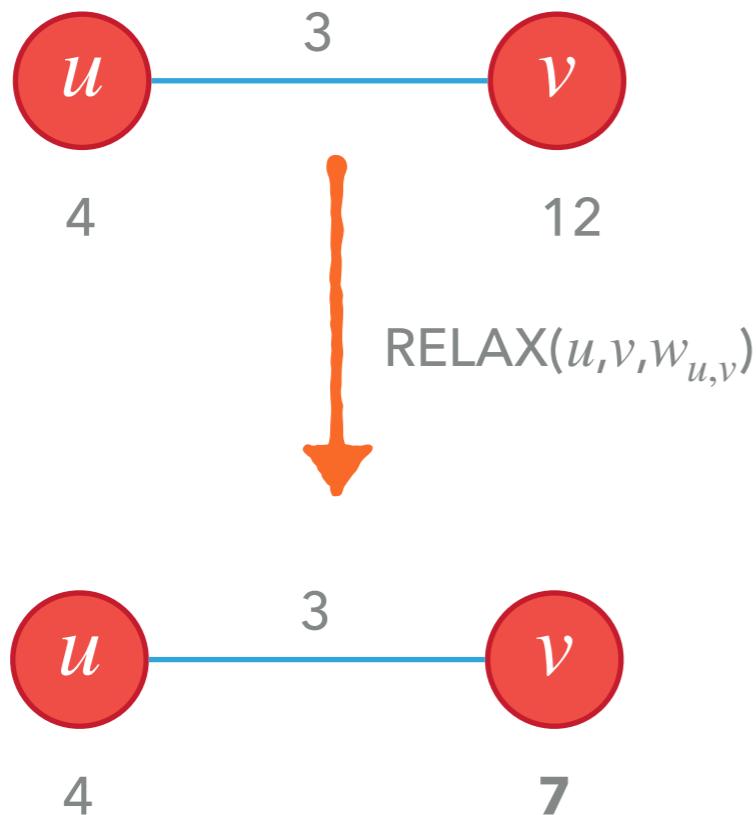
- ▶ $\delta(u, v) = \min\{w(p) \mid p : u \longrightarrow v \text{ semplice}\}$
- ▶ Lemma: Sia $(u, v) \in E$, allora $\delta(s, v) \leq \delta(s, u) + w_{u,v}$
- ▶ Supponiamo di avere una stima della del peso del percorso da s ad un nodo u , indicata con $distanza[u]$, e ad un nodo v , indicata con $distanza[v]$ e che esista l'arco (u, v) di peso $w_{u,v}$
- ▶ L'operazione di rilassamento consiste nel vedere se possiamo usare la nostra stima per u per migliorare la stima della distanza per v

OPERAZIONE DI RILASSAMENTO/RELAX DEGLI ARCHI

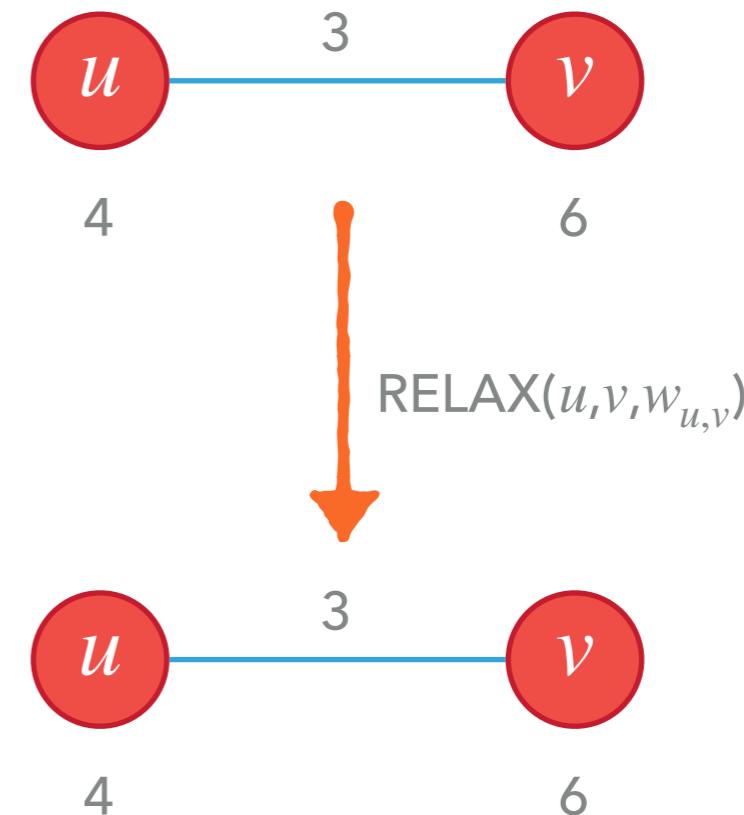
- ▶ Se $\text{distanza}[u] + w_{u,v} < \text{distanza}[v]$ allora possiamo andare da s a v passando per u con un costo minore
- ▶ Se quello è il caso aggiorniamo la nostra stima della distanza: $\text{distanza}[v] = \text{distanza}[u] + w_{u,v}$
- ▶ Altrimenti non la modifichiamo
- ▶ Indichiamo questa operazione come $\text{RELAX}(u, v, w_{u,v})$

OPERAZIONE DI RILASSAMENTO/RELAX DEGLI ARCHI

Caso 1



Caso 2



```
if distanza[u] + w < distanza[v]
    distanza[v] = distanza[u] + w
    predecessore[v] = u
```

OPERAZIONE DI RILASSAMENTO/RELAX DEGLI ARCHI

- ▶ Assumiamo che (u,v) appartenga ad un cammino minimo da s a v .
- ▶ Assumiamo che prima di RELAX, valga
 $distanza[u] = \delta(s, u)$
- ▶ Allora dopo $\text{RELAX}(u, v, w_{u,v})$, varrà $distanza[v] = \delta(s, v)$
- ▶ In generale, si dimostra facilmente che se $d[v] \geq \delta(s, v)$ prima di relax, allora $d[v] \geq \delta(s, v)$ anche dopo relax.

IDEA GENERALE: ALGORITMO DI BELLMAN-FORD

- ▶ Il percorso più lungo in un grafo di $|V|$ nodi è composto da al più $|V| - 1$ archi (altrimenti siamo in un ciclo)
- ▶ Se effettuiamo $|V| - 1$ operazioni di rilassamento per ogni arco allora abbiamo che le nostre stime delle distanze non si possono più modificare
- ▶ Abbiamo quindi trovato il percorso di peso/costo minimo da s a ogni altro nodo

PSEUDOCODICE

Parametri: grafo G , nodo sorgente s

inizialmente impostiamo distanza e predecessore di tutti i nodi

for all $v \in V$:

 distanza $[v]$ = $+\infty$

 predecessore $[v]$ = None

e poi impostiamo il nodo s come sorgente a distanza 0

distanza $[s]$ = 0

for i in range($0, |V| - 1$)

 for all $(u, v) \in E$ # effettuiamo il rilassamento di tutti gli archi

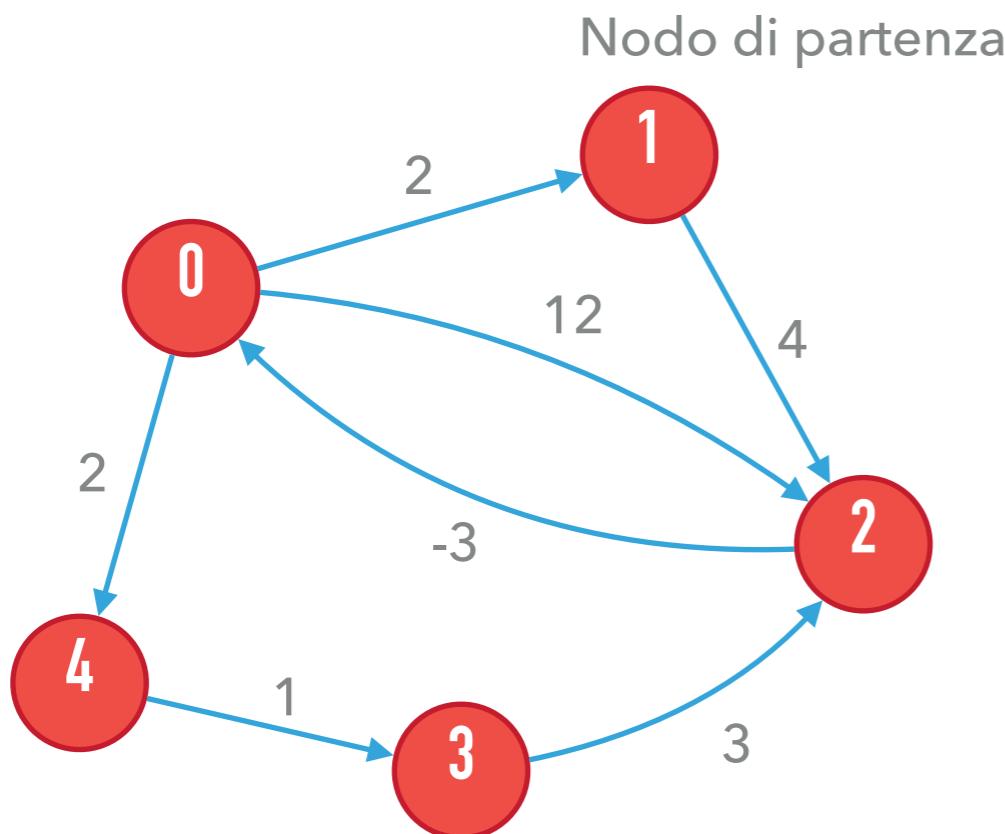
 RELAX($u, v, w_{u,v}$)

SE DOPO $|V| - 1$ ITERAZIONI FOSSE ANCORA POSSIBILE AGGIORNARE LE DISTANZE SIGNIFICHERREBBE CHE IL GRAFO CONTIENE UN CICLO NEGATIVO: POSSIAMO USARE L'ALGORITMO PER IDENTIFICARE QUESTO TIPO DI GRAFI

CICLI DI PESO NEGATIVO

- ▶ Sia $c : v_0, v_1, \dots, v_k$ un ciclo semplice ($v_0 = v_k$, tutti gli altri vertici diversi).
- ▶ c ha peso negativo ($w(c) < 0$) solo se a fine algoritmo esiste $(u, v) \in E$ tale che $\text{distanza}[v] > \text{distanza}[u] + w_{u,v}$
- ▶ Per assurdo, se $c : v_0, v_1, \dots, v_k$ ha peso negativo ma $\text{distanza}[v_i] \leq \text{distanza}[v_{i-1}] + w_{i-1,i}$ per ogni i , allora
 - ▶ $\sum_{i=1}^k \text{distanza}[v_i] \leq \sum_{i=1}^k \text{distanza}[v_{i-1}] + \sum_{i=1}^k w_{i-1,i}$ e quindi
 - ▶ $\sum_{i=1}^k w_{i-1,i} \geq 0$, assurdo.

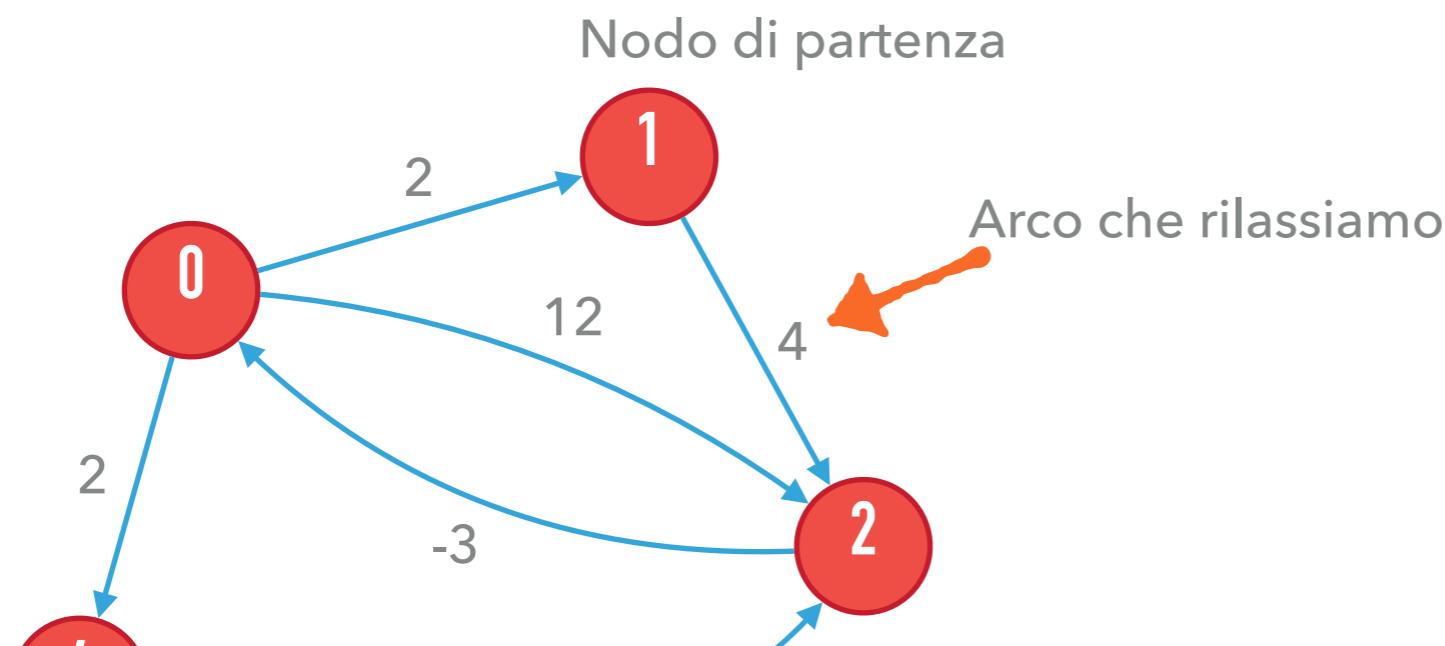
ESEMPIO DI ESECUZIONE



Per l'esempio usiamo un grafo orientato, altrimenti non potremmo mai avere pesi negativi (avremmo un ciclo negativo)

| Vertice | Peso |
|---------|----------|
| 0 | ∞ |
| 1 | 0 |
| 2 | ∞ |
| 3 | ∞ |
| 4 | ∞ |

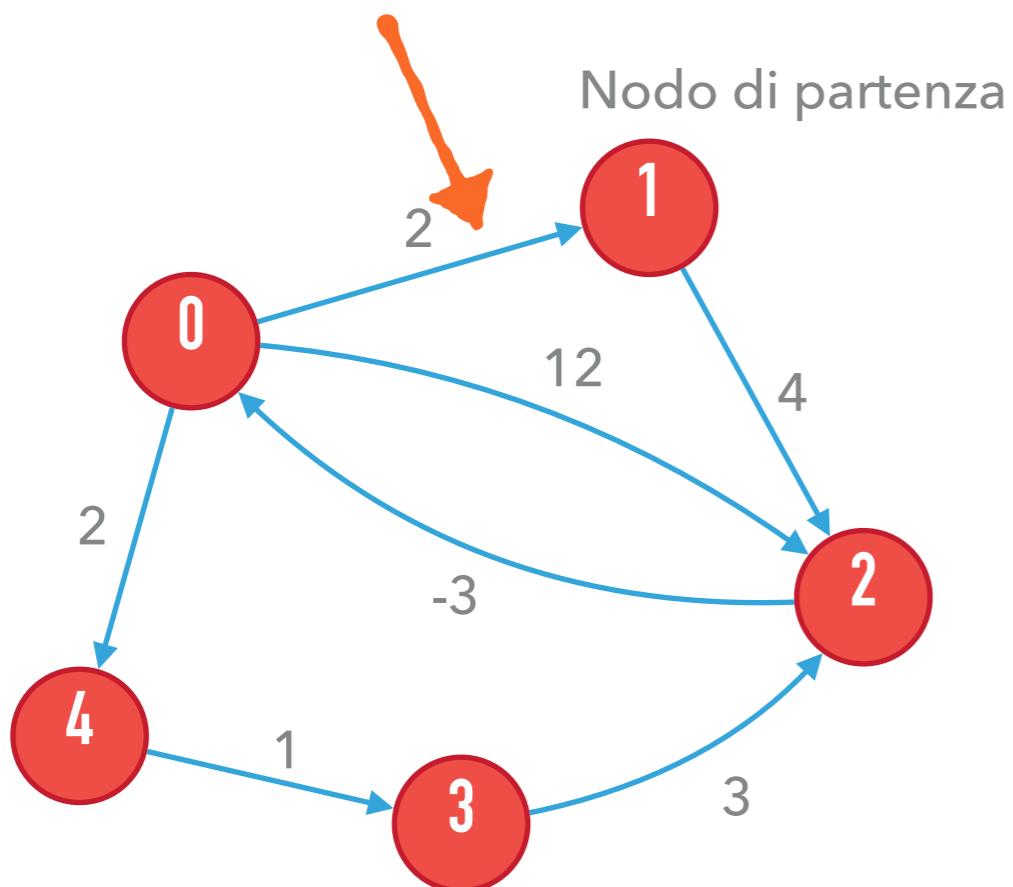
ESEMPIO DI ESECUZIONE



| Vertice | Peso |
|---------|----------|
| 0 | ∞ |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |

ESEMPIO DI ESECUZIONE

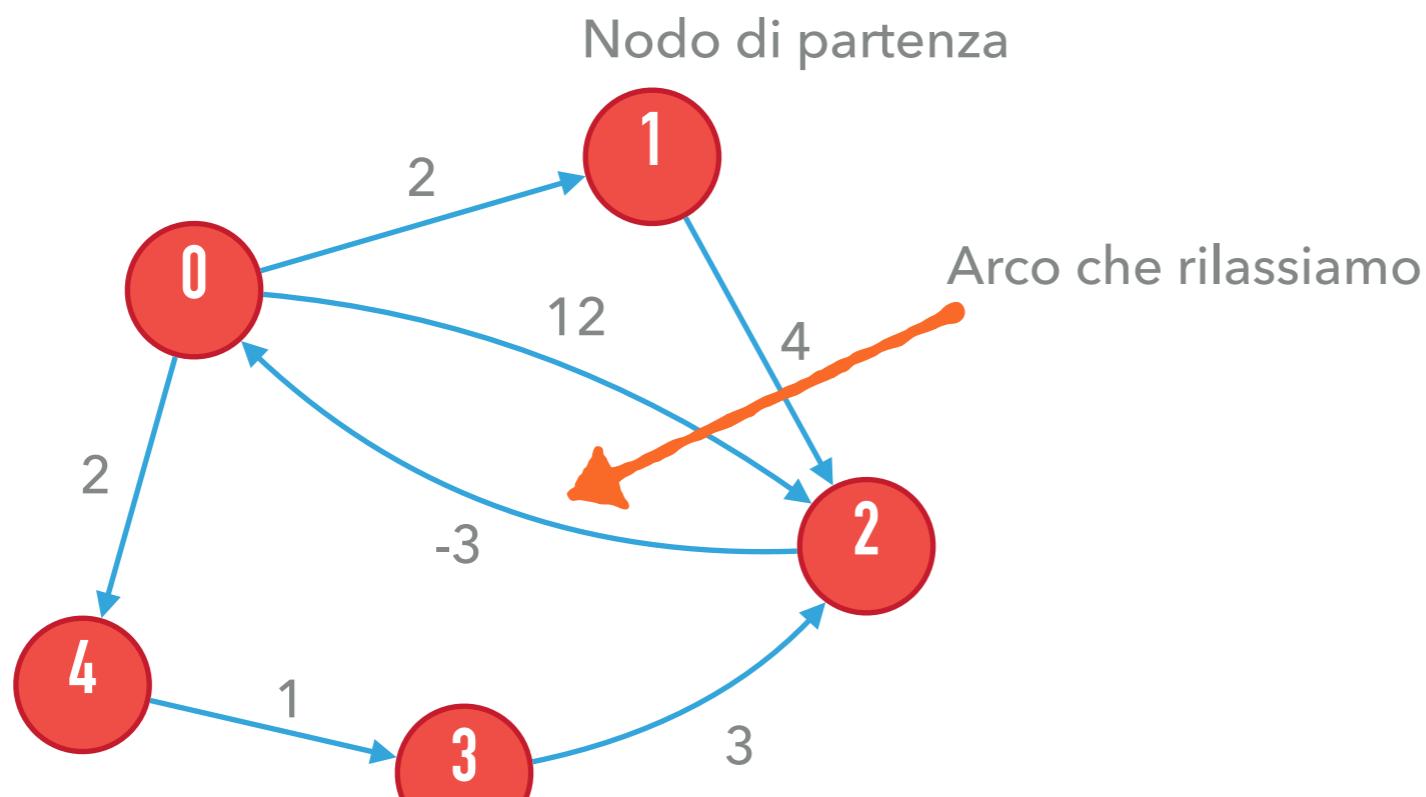
Arco che rilassiamo



Nodo di partenza

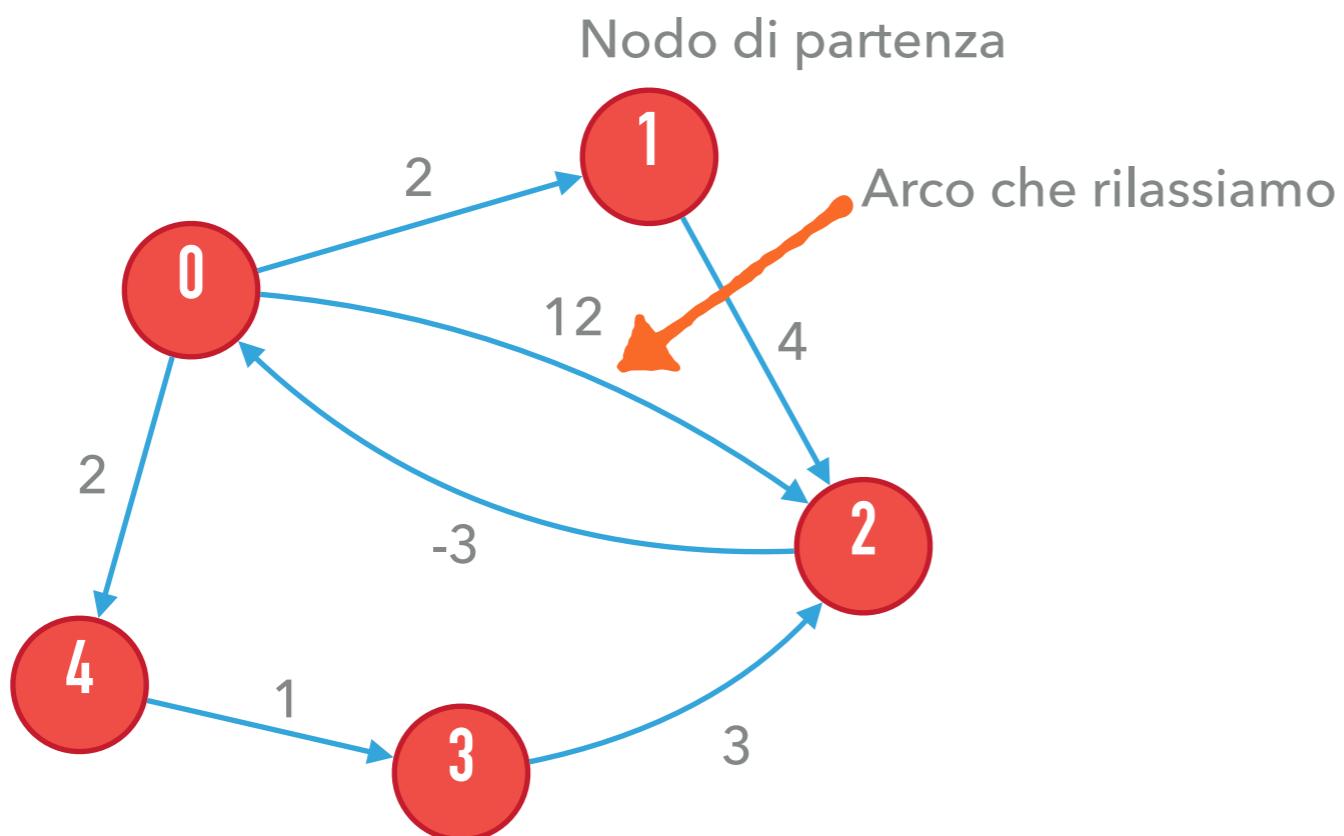
| Vertice | Peso |
|---------|----------|
| 0 | ∞ |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |

ESEMPIO DI ESECUZIONE



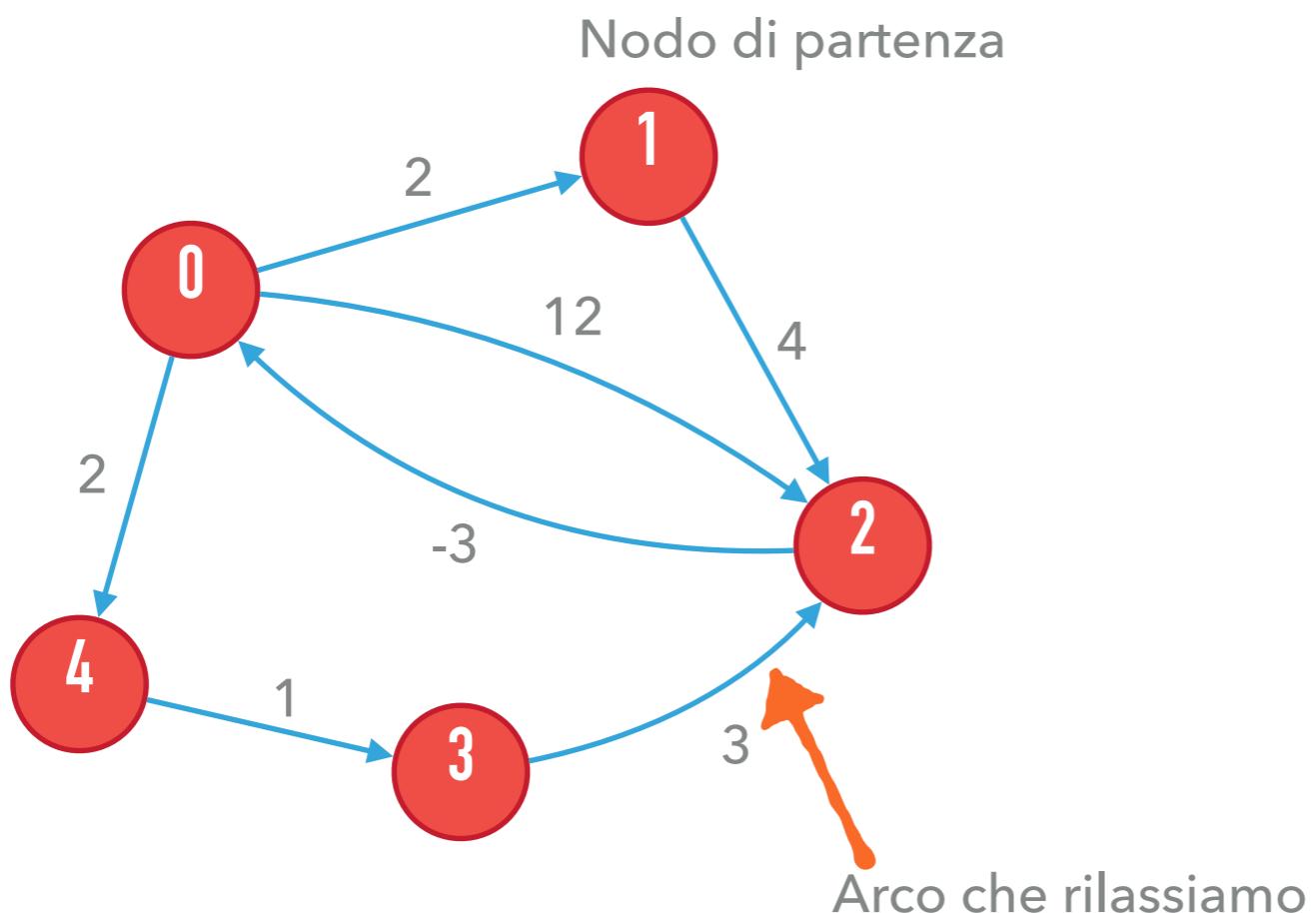
| Vertice | Peso |
|---------|----------|
| 0 | 1 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |

ESEMPIO DI ESECUZIONE



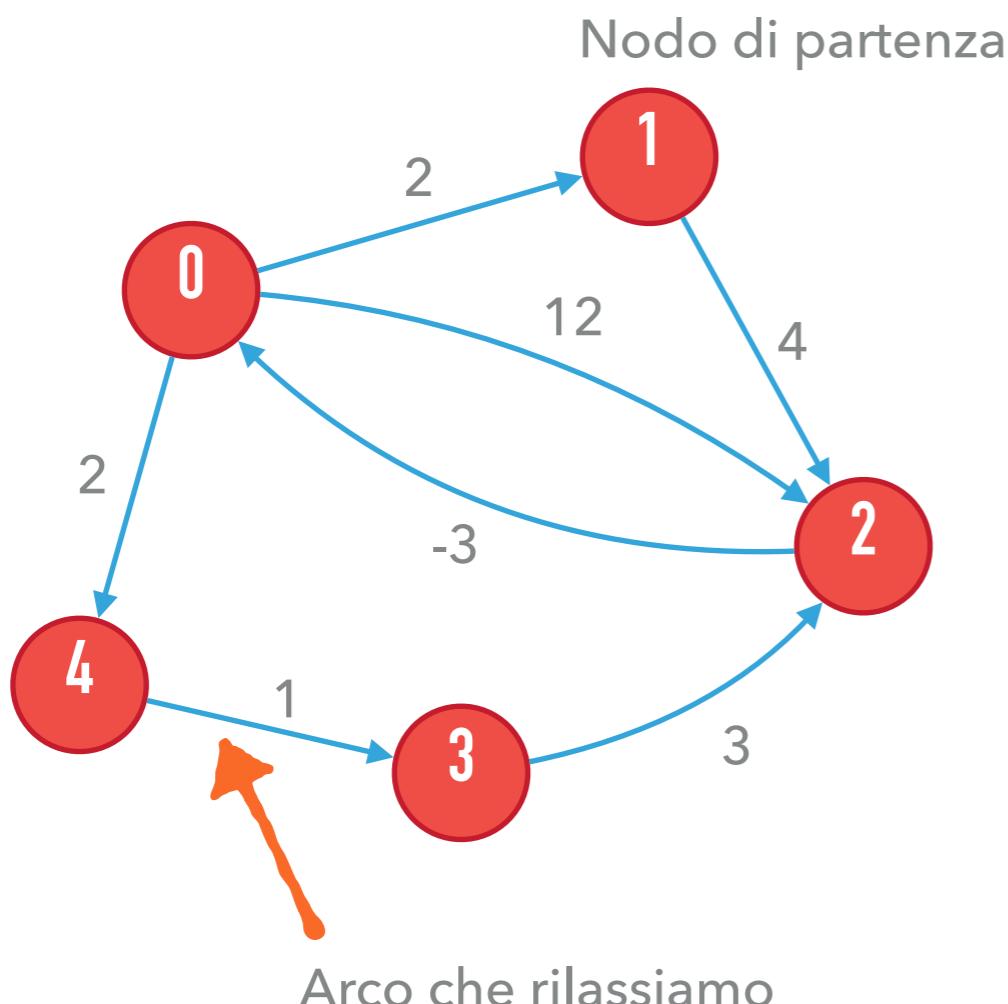
| Vertice | Peso |
|---------|----------|
| 0 | 1 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |

ESEMPIO DI ESECUZIONE



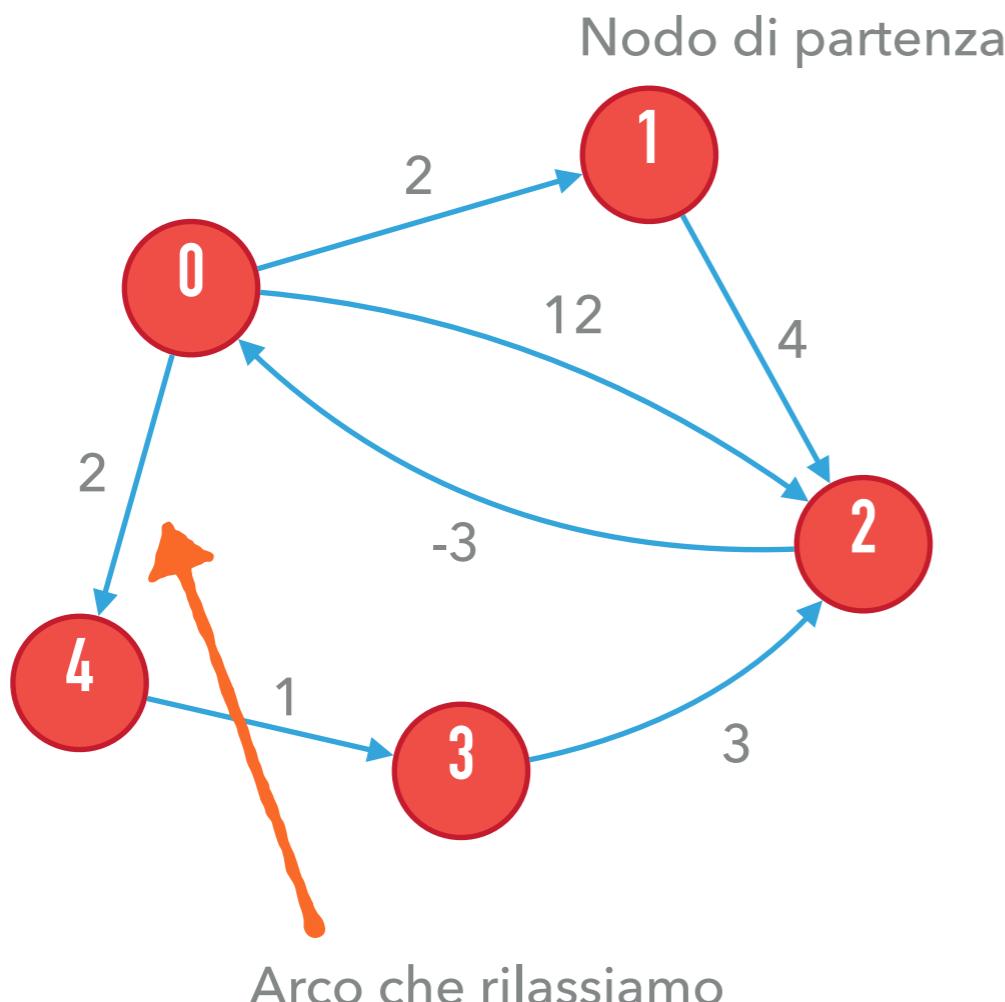
| Vertice | Peso |
|---------|----------|
| 0 | 1 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |

ESEMPIO DI ESECUZIONE



| Vertice | Peso |
|---------|----------|
| 0 | 1 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |

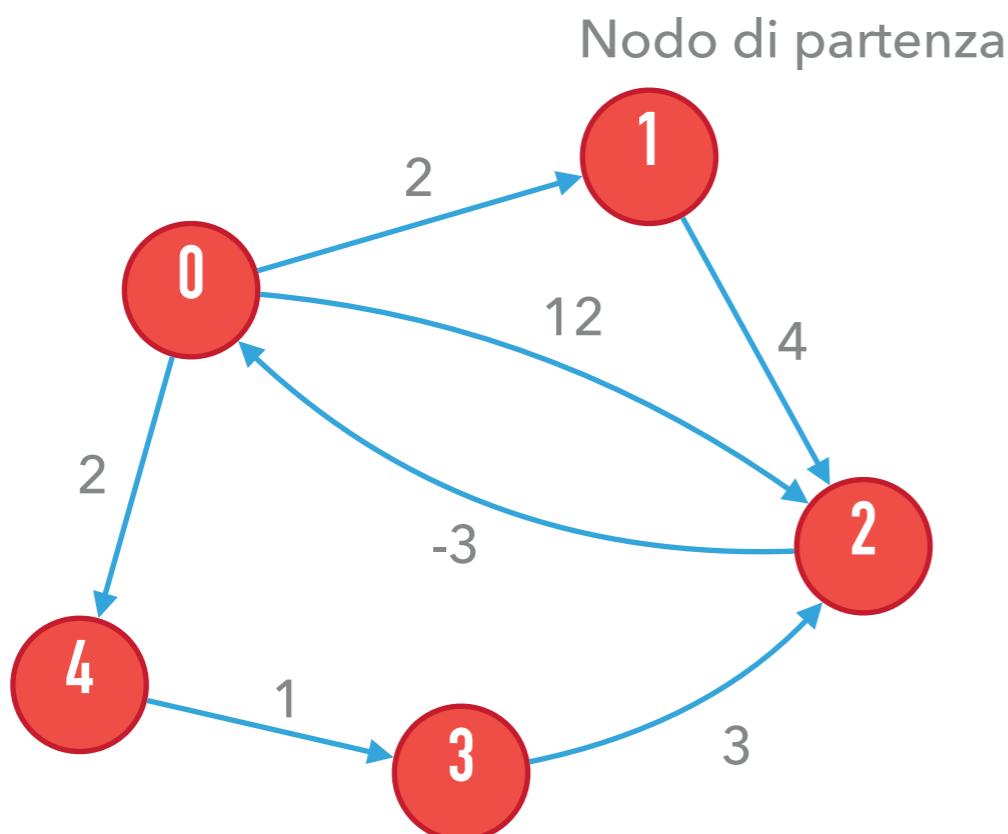
ESEMPIO DI ESECUZIONE



Ora svolgiamo altre 3 iterazioni su tutti i nodi...

| Vertice | Peso |
|---------|------|
| 0 | 1 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | 3 |

ESEMPIO DI ESECUZIONE



Ottenendo questo risultato:

| Vertice | Peso |
|---------|------|
| 0 | 1 |
| 1 | 0 |
| 2 | 4 |
| 3 | 4 |
| 4 | 3 |

ALGORITMO DI BELLMAN-FORD: COMPLESSITÀ

- ▶ La singola operazione di rilassamento richiede tempo costante
- ▶ Ogni iterazione del ciclo interno richiede $O(E)$ operazioni di rilassamento
- ▶ Ed il ciclo esterno esegue $O(V)$ volte
- ▶ Il costo totale è quindi $O(VE)$
- ▶ Peggio di $O(V + E)$ richiesto da BFS nel caso di grafo non pesato (o con pesi tutti uguali alla stessa costante positiva)

ALGORITMO DI DIJKSTRA

- ▶ L'algoritmo di Dijkstra è applicabile nel caso più ristretto in cui tutti i pesi sono **non negativi**
- ▶ Questo caso si presenta spesso in casi pratici: tempi di percorrenza, distanze in km, etc
- ▶ Rispetto all'algoritmo di Bellman-Ford riesce ad essere più efficiente sfruttando una coda di priorità: una struttura dati che permette di inserire elementi e rimuovere l'elemento di valore minimo

IDEA GENERALE: ALGORITMO DI DIJKSTRA

- ▶ Teniamo una lista di nodi non visitati
- ▶ Il nodo iniziale ha distanza 0 e tutti gli altri infinito
- ▶ Prendiamo il nodo v con distanza minima tra quelli non visitati:
 - ▶ Per ogni vicino aggiorniamo la distanza vedendo se possiamo raggiungerlo più velocemente passando da v
 - ▶ Rimuoviamo v dalla lista dei nodi non visitati
 - ▶ Continuiamo finché non possiamo più aggiornare le distanze dei nodi

PSEUDOCODICE

Parametri: grafo G , nodo sorgente s

inizialmente impostiamo distanza e predecessore di tutti i nodi

for all $v \in V$:

 distanza $[v] = +\infty$ # indichiamo distanza $[v]$ anche come $d[v]$

 predecessore $[v] = \text{None}$

distanza $[s] = 0$ # e poi impostiamo il nodo s come sorgente a distanza 0

$S = \{\}$ # insieme dei nodi già visitati (inizialmente vuoto)

$Q = \text{coda_di_priorità}(V)$ # coda di priorità con tutti i vertici già inseriti

while Q is non-empty

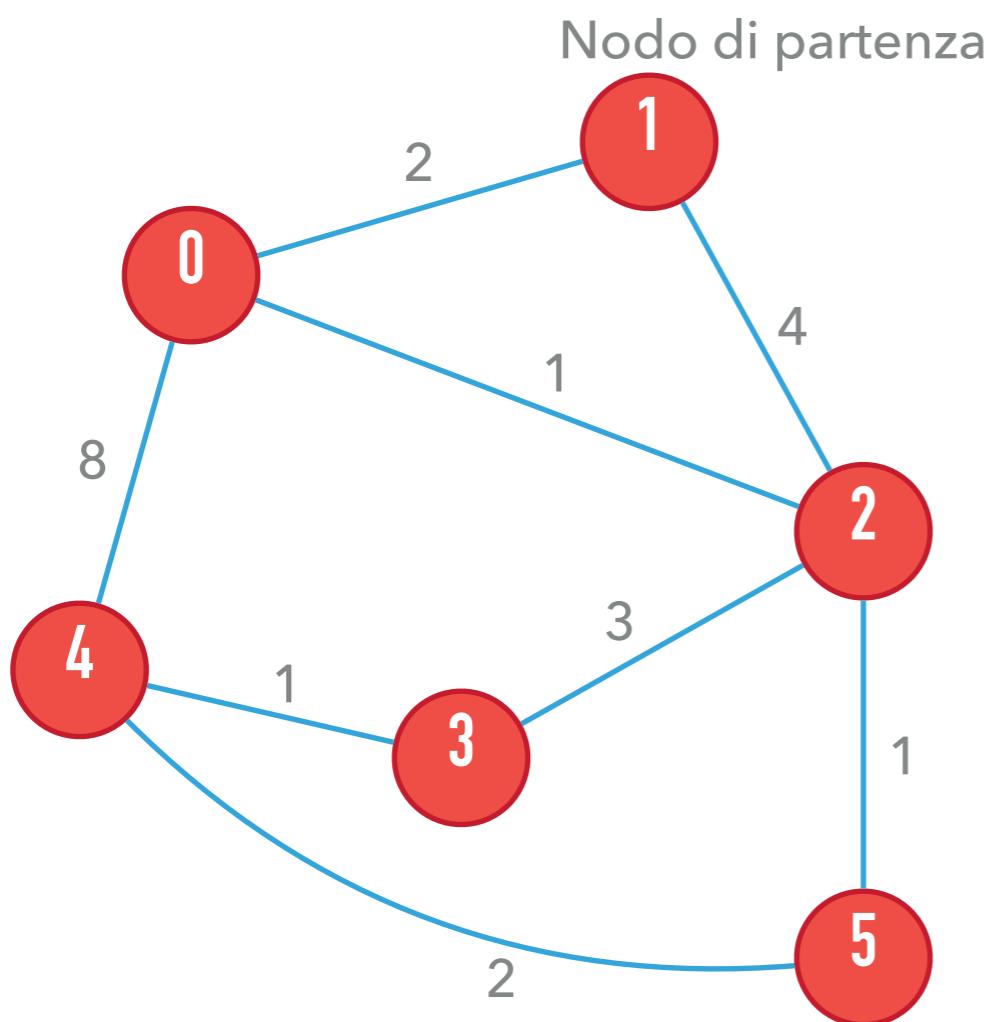
$u = \text{estrai_minimo}(Q)$ # estraiamo il prossimo vertice

$S = S \cup \{u\}$ # lo inseriamo nella lista dei visitati

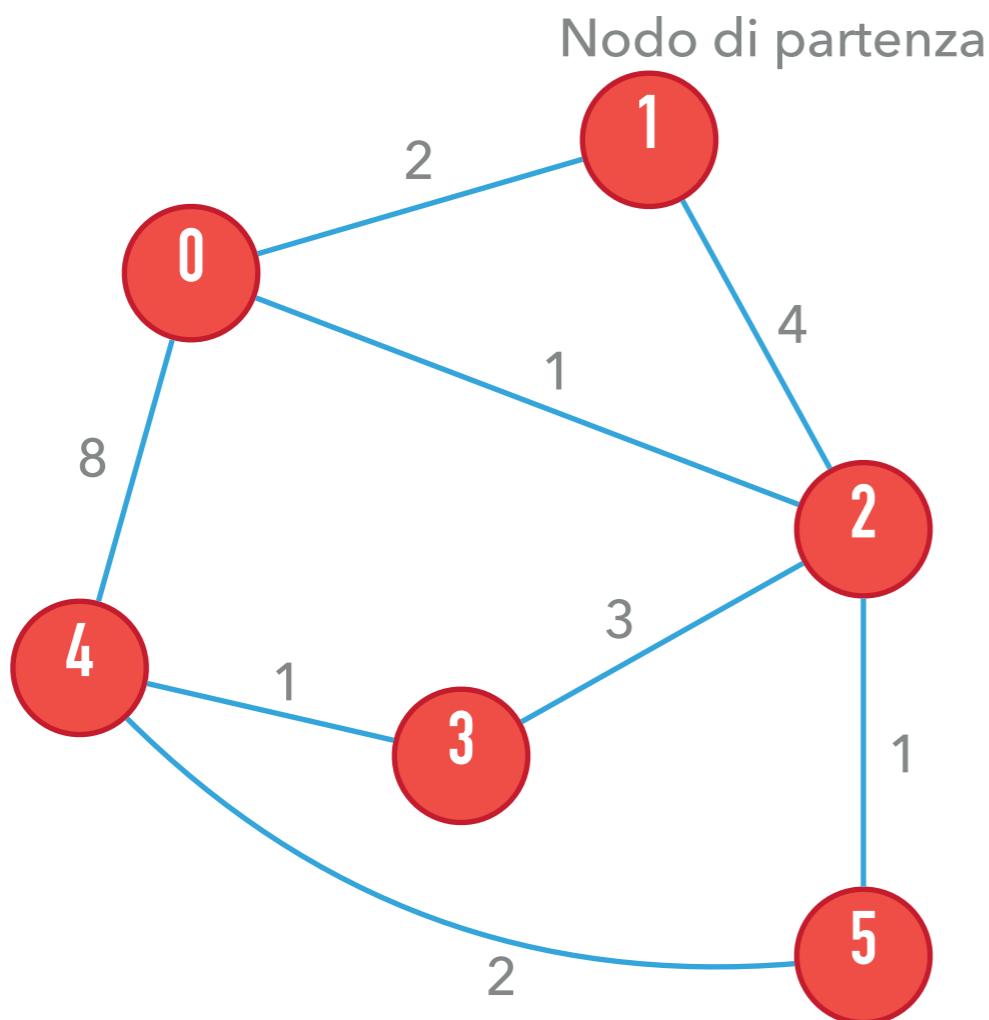
 for all v adiacenti a u # e aggiorniamo la distanza di tutti i vicini

 RELAX($u, v, w_{u,v}$)

ESEMPIO DI ESECUZIONE



ESEMPIO DI ESECUZIONE



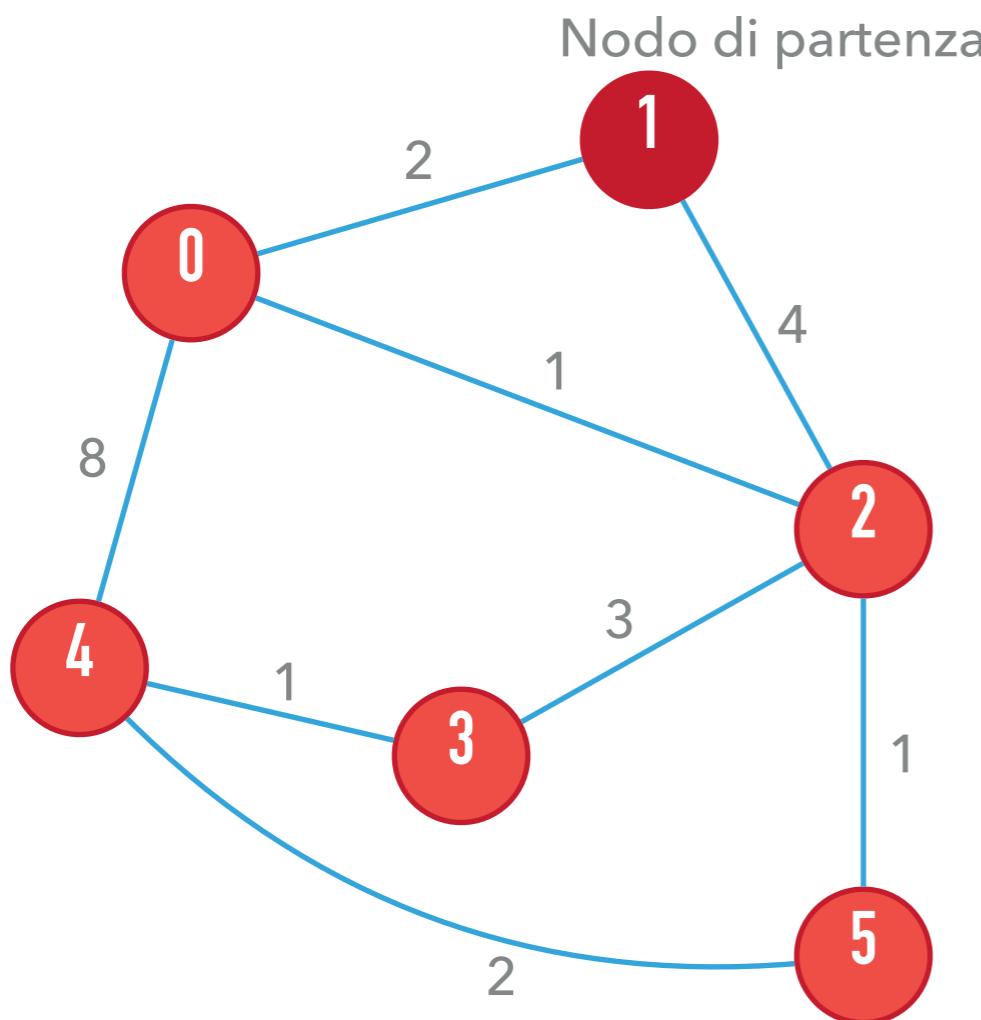
Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
|---------|------|

| Vertice | Peso |
|---------|----------|
| 0 | ∞ |
| 1 | 0 |
| 2 | ∞ |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

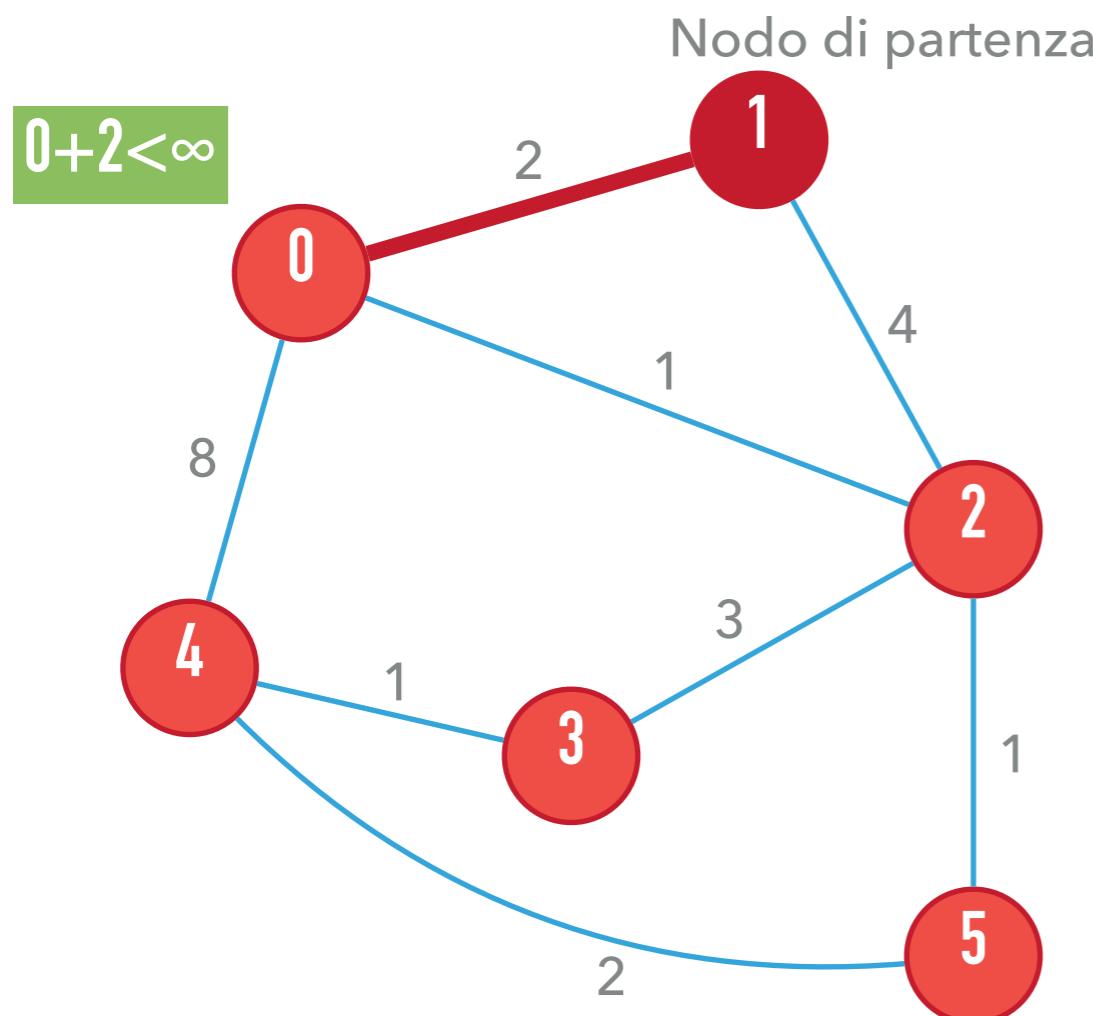
1

Lista dei nodi non visitati

0 2 3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | ∞ |
| 1 | 0 |
| 2 | ∞ |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

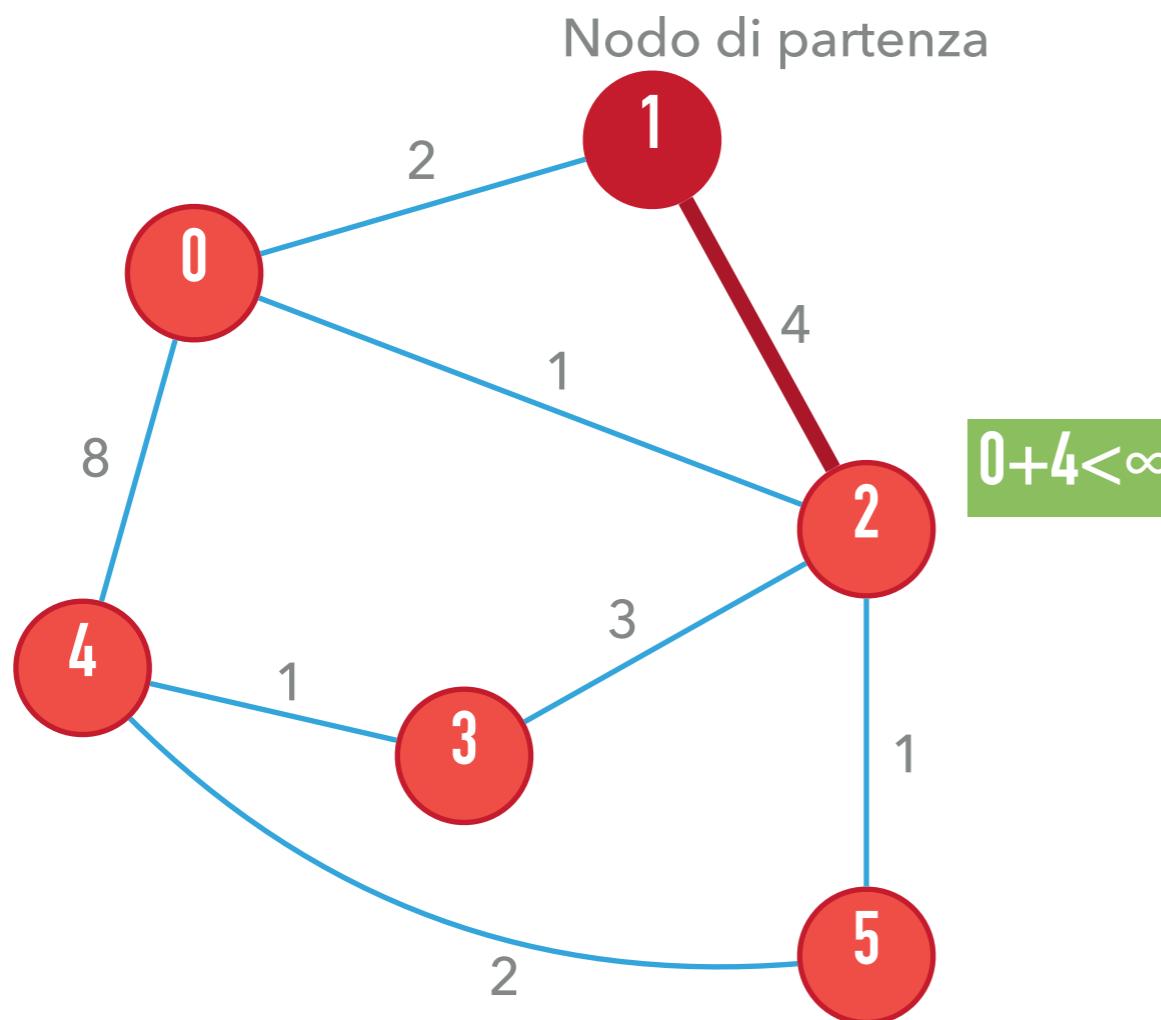
1

Lista dei nodi non visitati

0 2 3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | ∞ |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

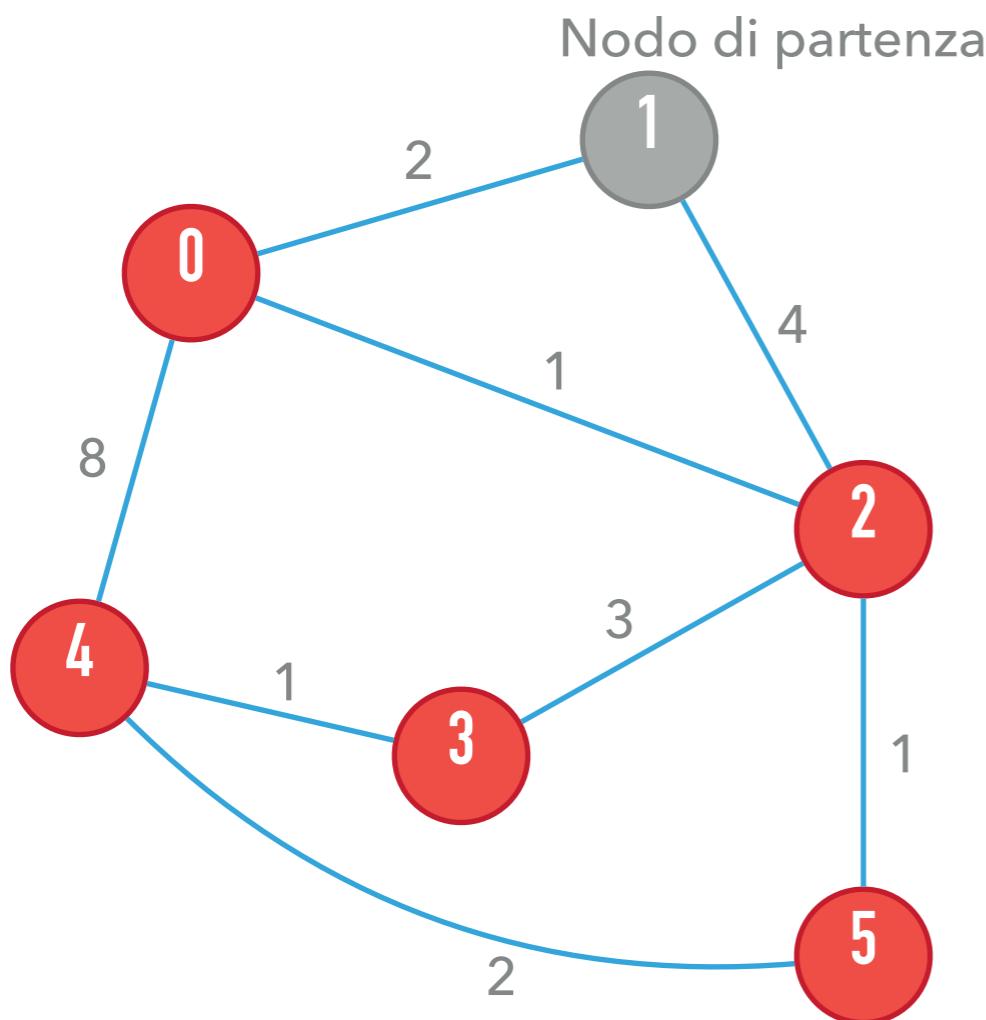
1

Lista dei nodi non visitati

0 2 3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



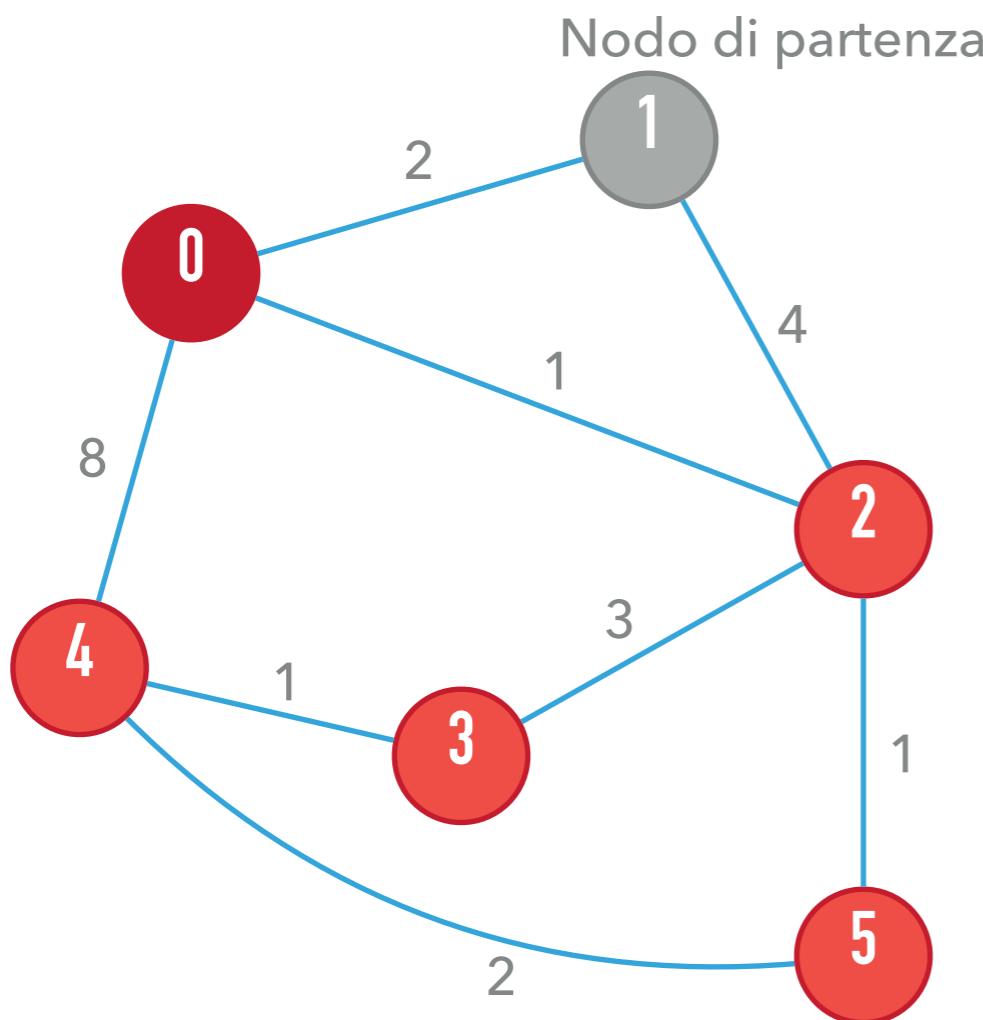
Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
|---------|------|

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

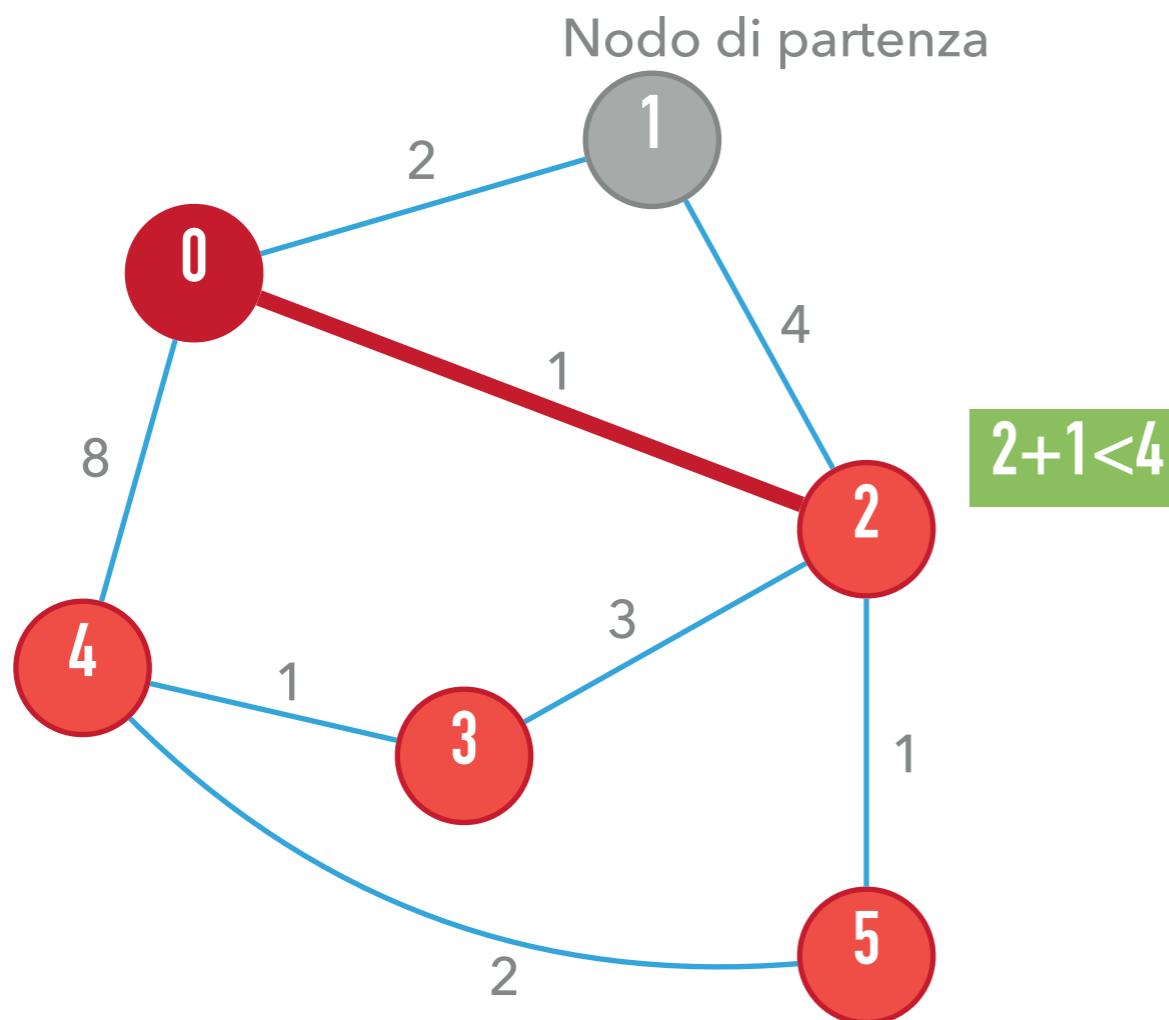
0

Lista dei nodi non visitati

2 3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 4 |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

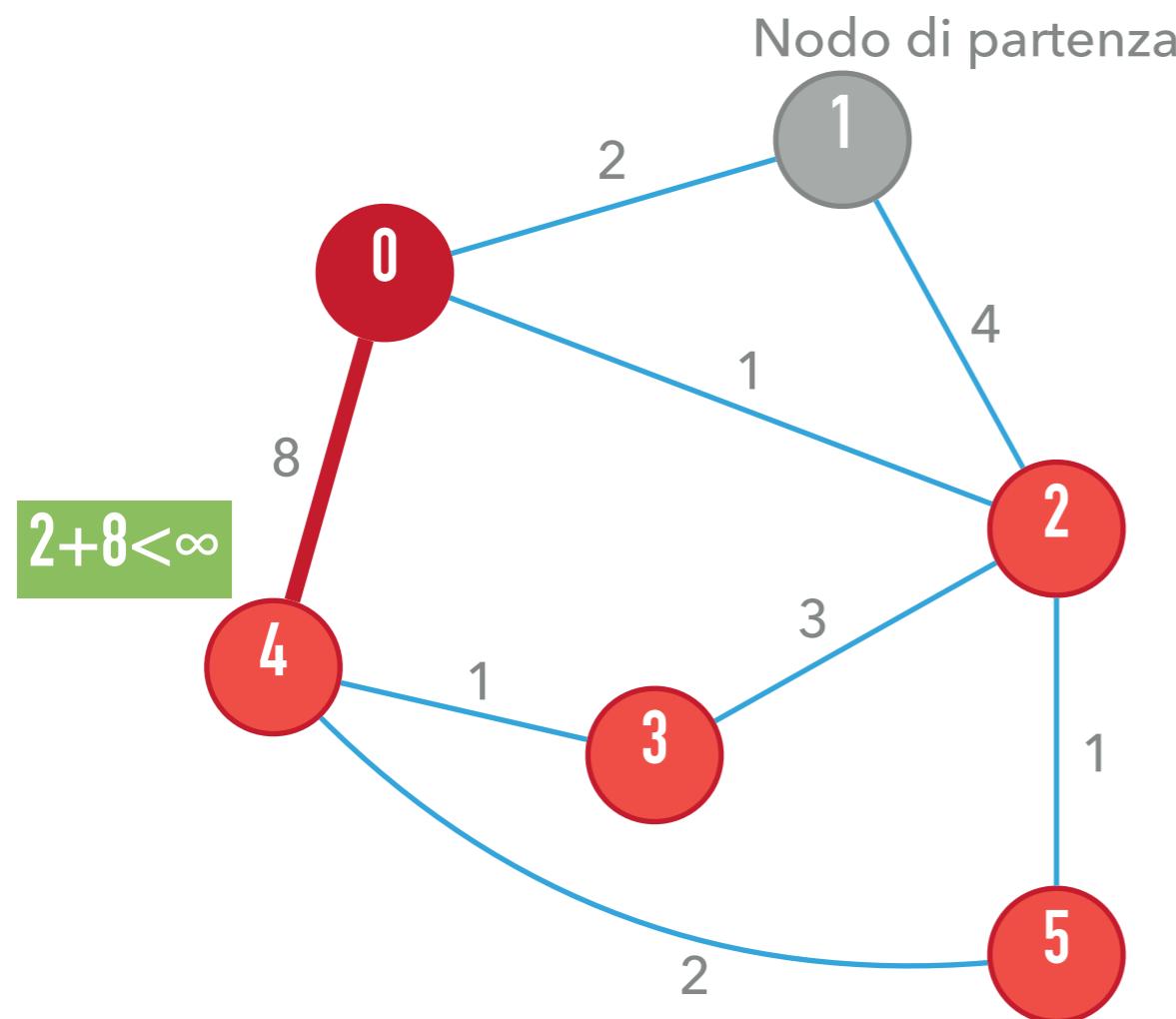
0

Lista dei nodi non visitati

2 3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | ∞ |
| 4 | ∞ |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

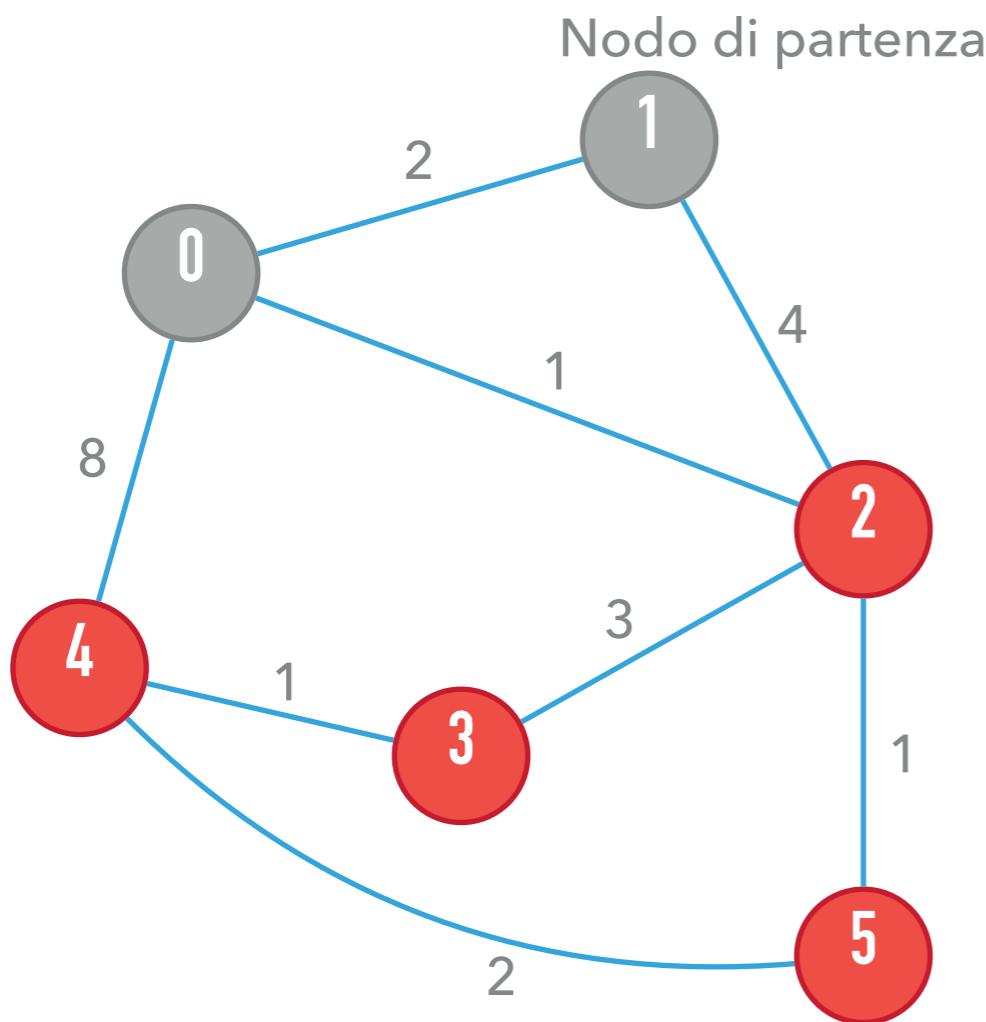
0

Lista dei nodi non visitati

2 3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | ∞ |
| 4 | 10 |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



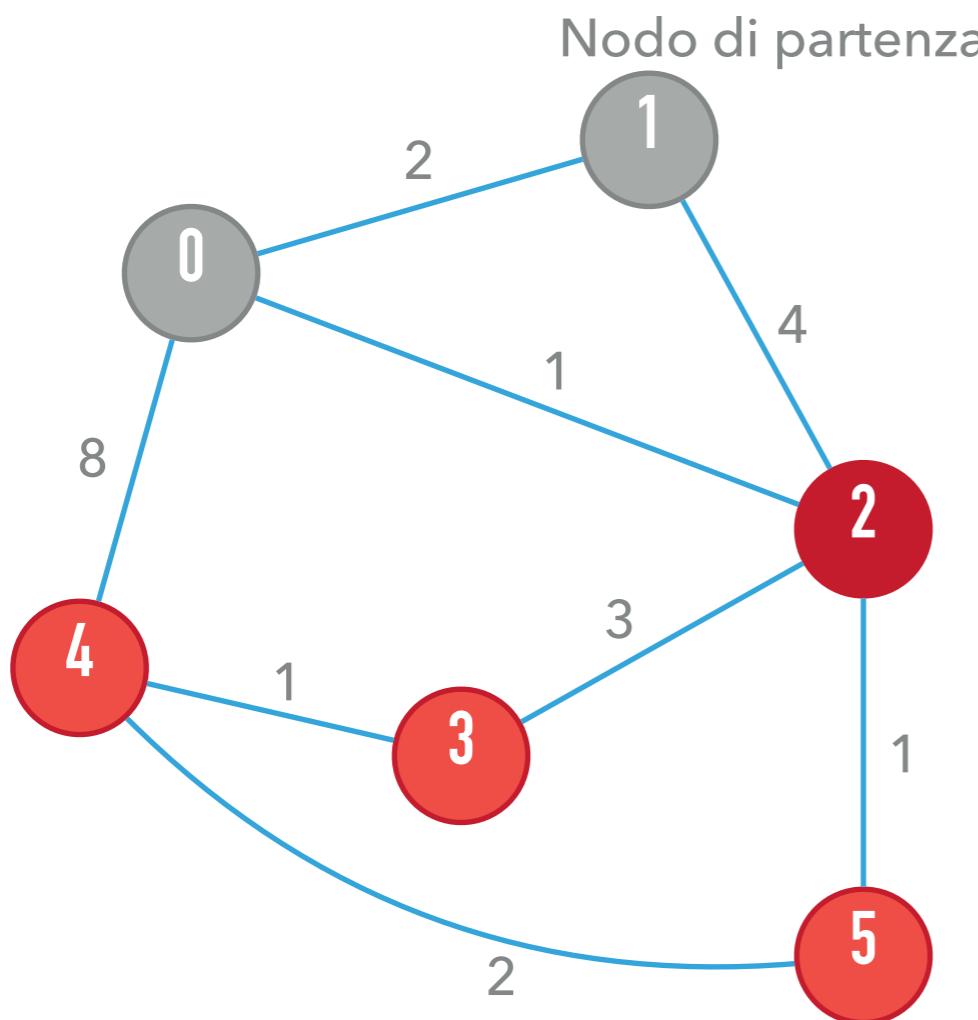
Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
|---------|------|

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | ∞ |
| 4 | 10 |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

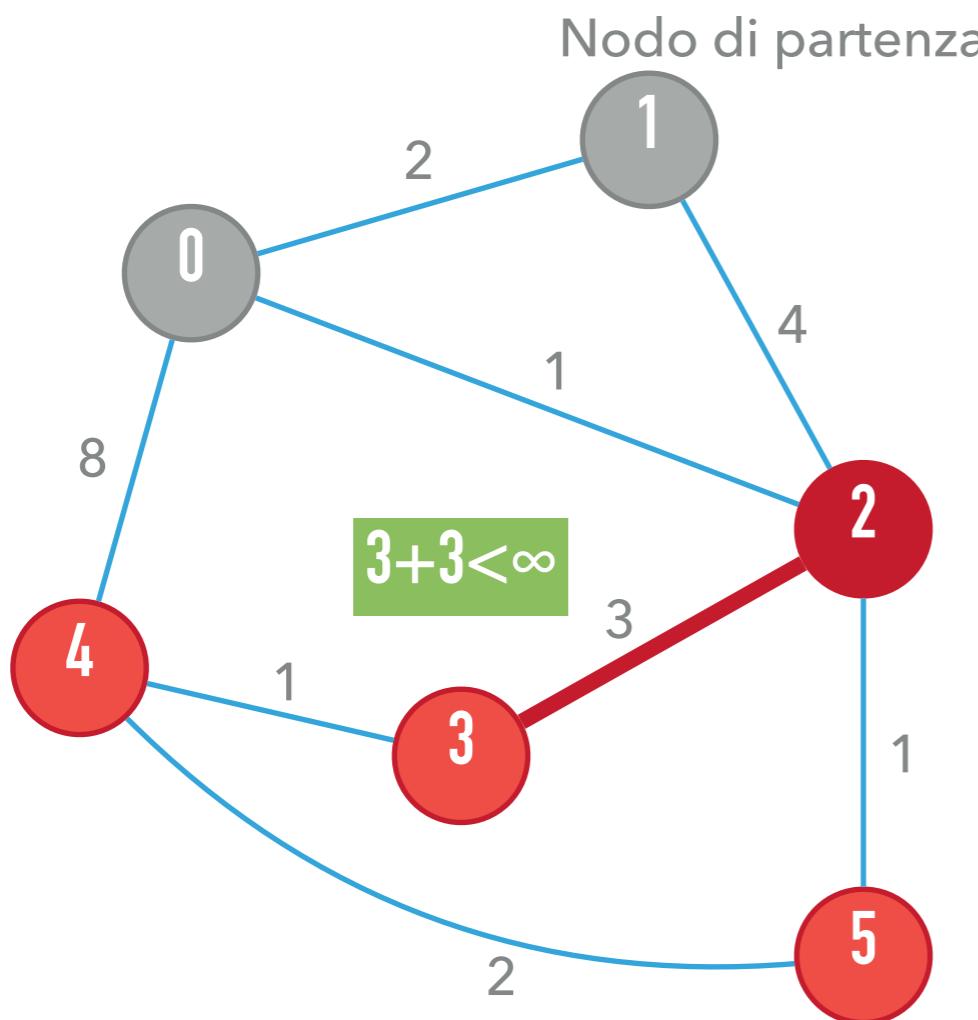
2

Lista dei nodi non visitati

3 4 5

| Vertice | Peso |
|---------|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | ∞ |
| 4 | 10 |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima: 2

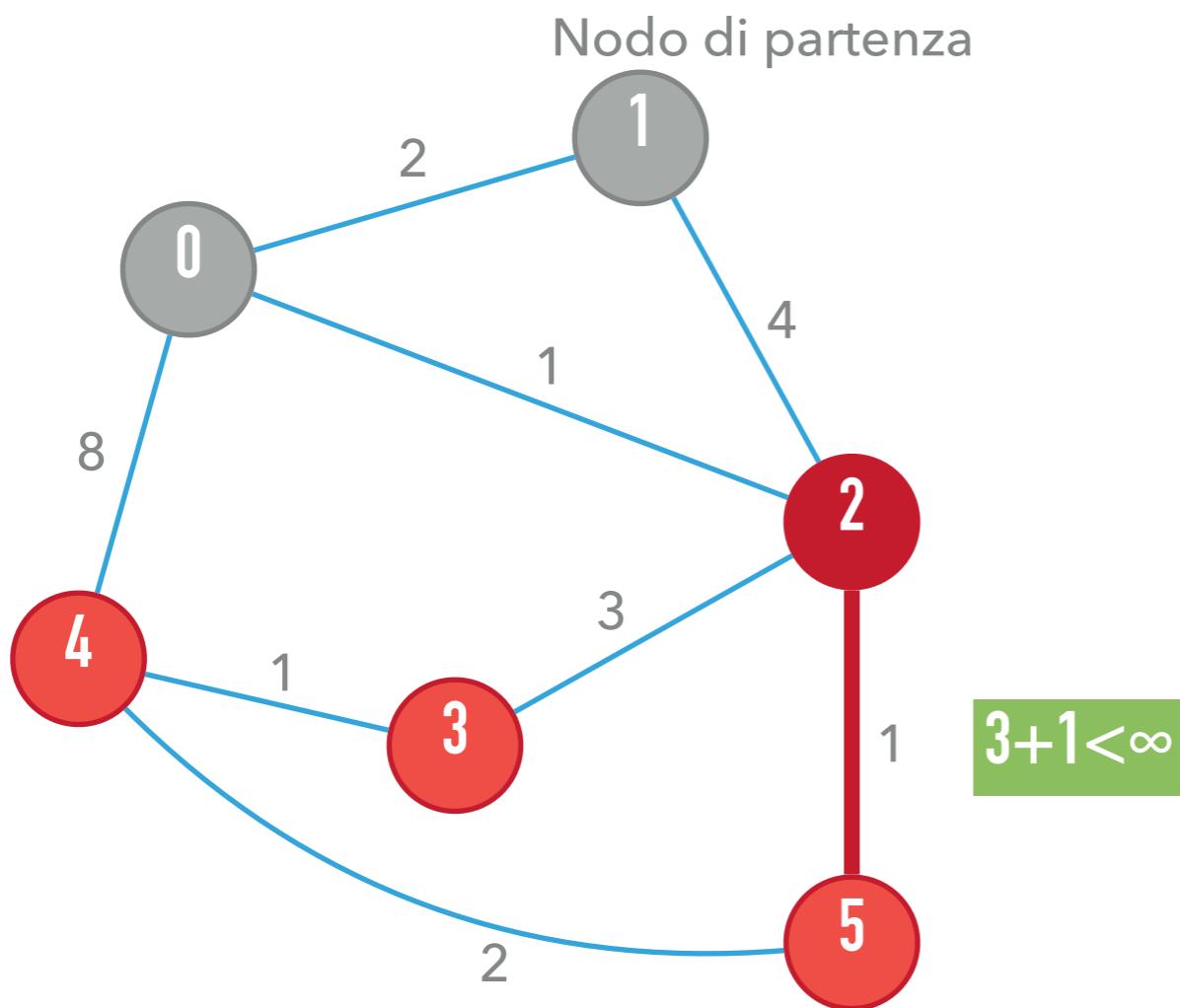
Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
|---------|------|

| | |
|---|----------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | ∞ |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima:

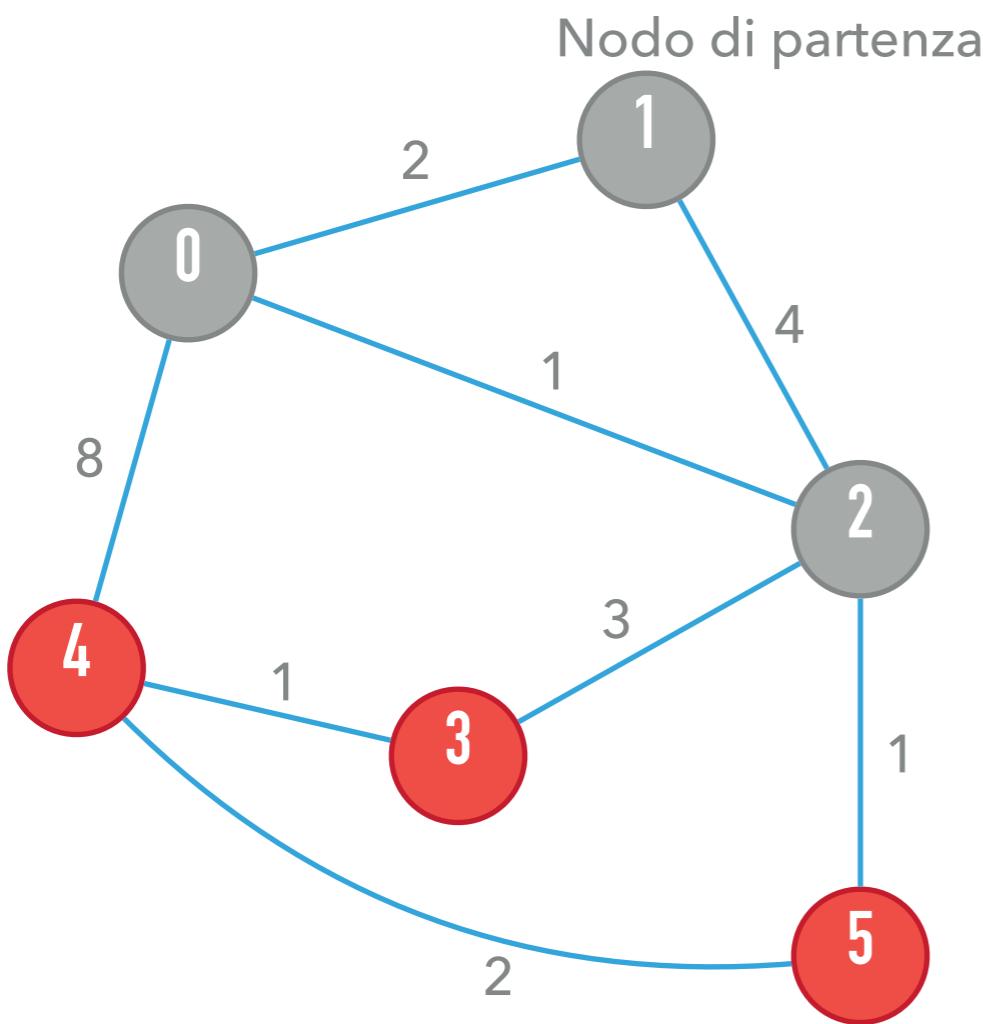
2

Lista dei nodi non visitati

3 4 5

| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE

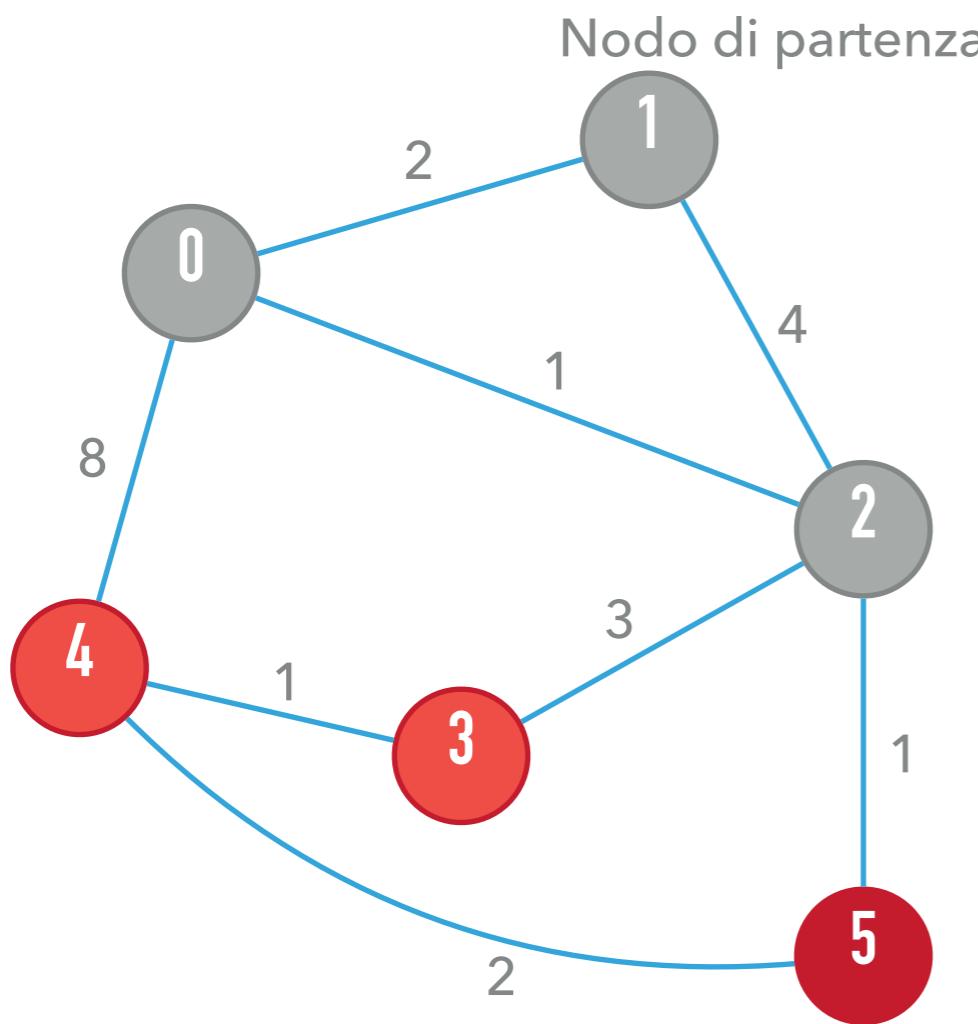


Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE



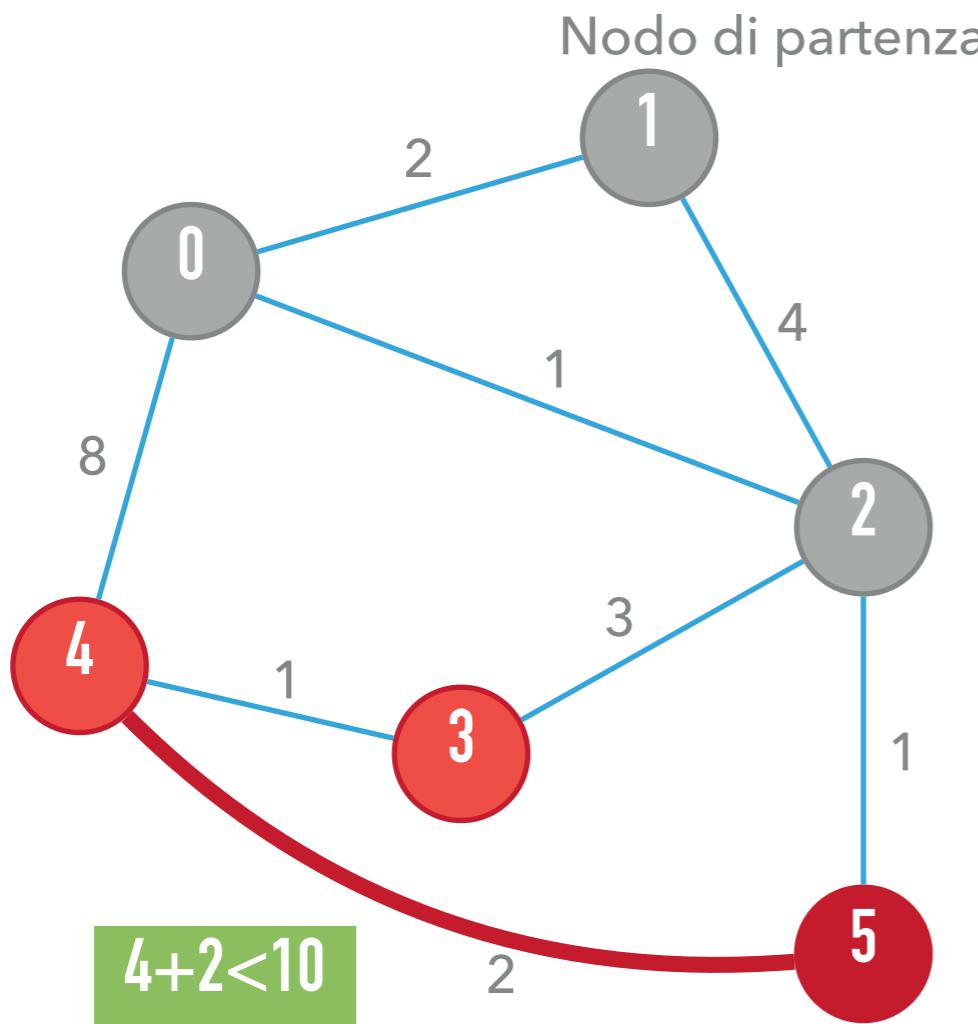
Nodo con distanza minima: 5

Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 10 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE



Nodo con distanza minima: 5

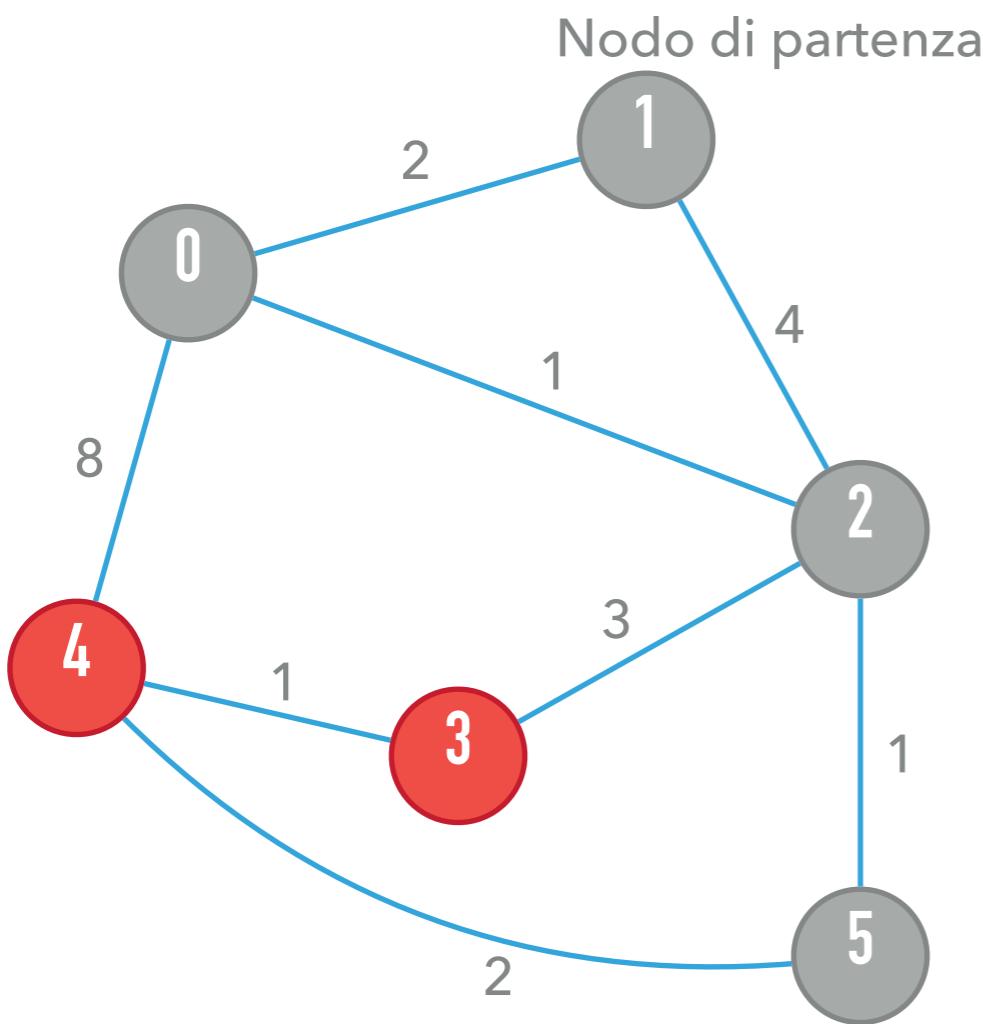
Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
|---------|------|

| | |
|---|---|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE

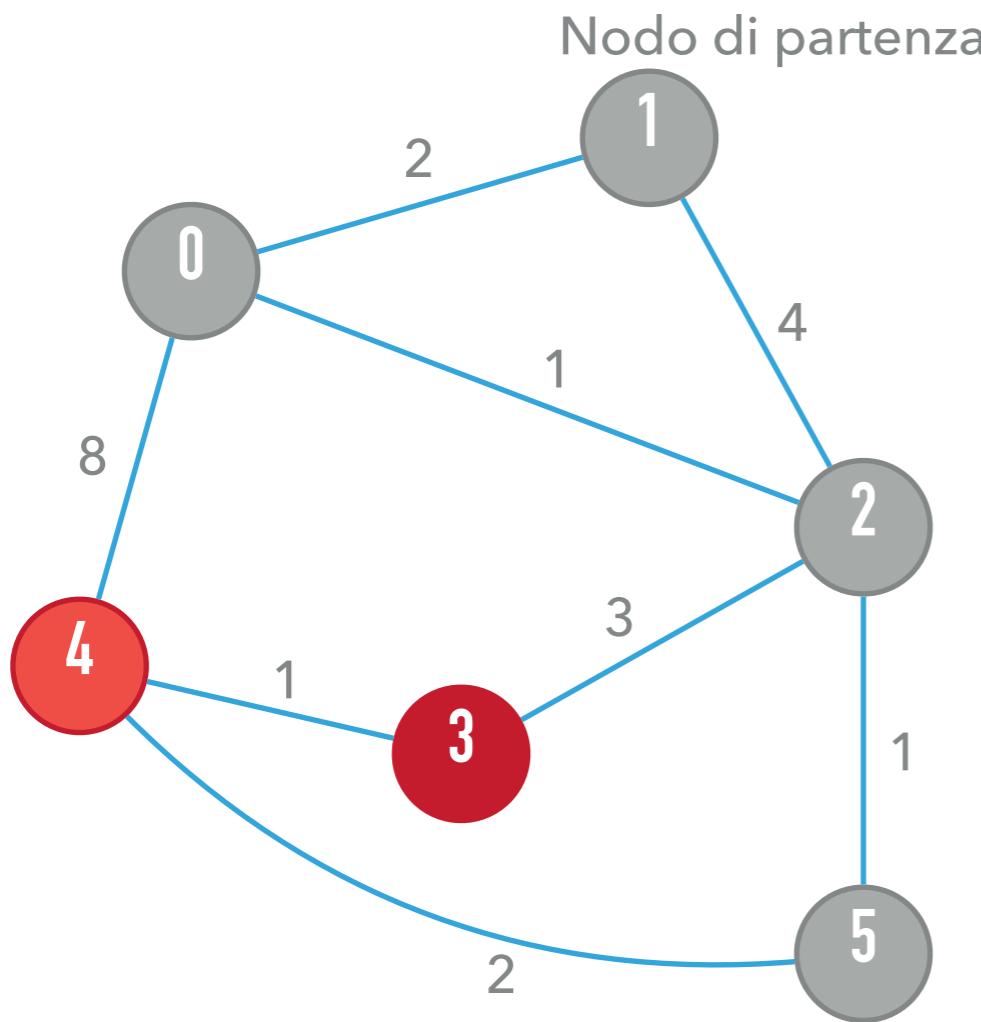


Lista dei nodi non visitati



| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE



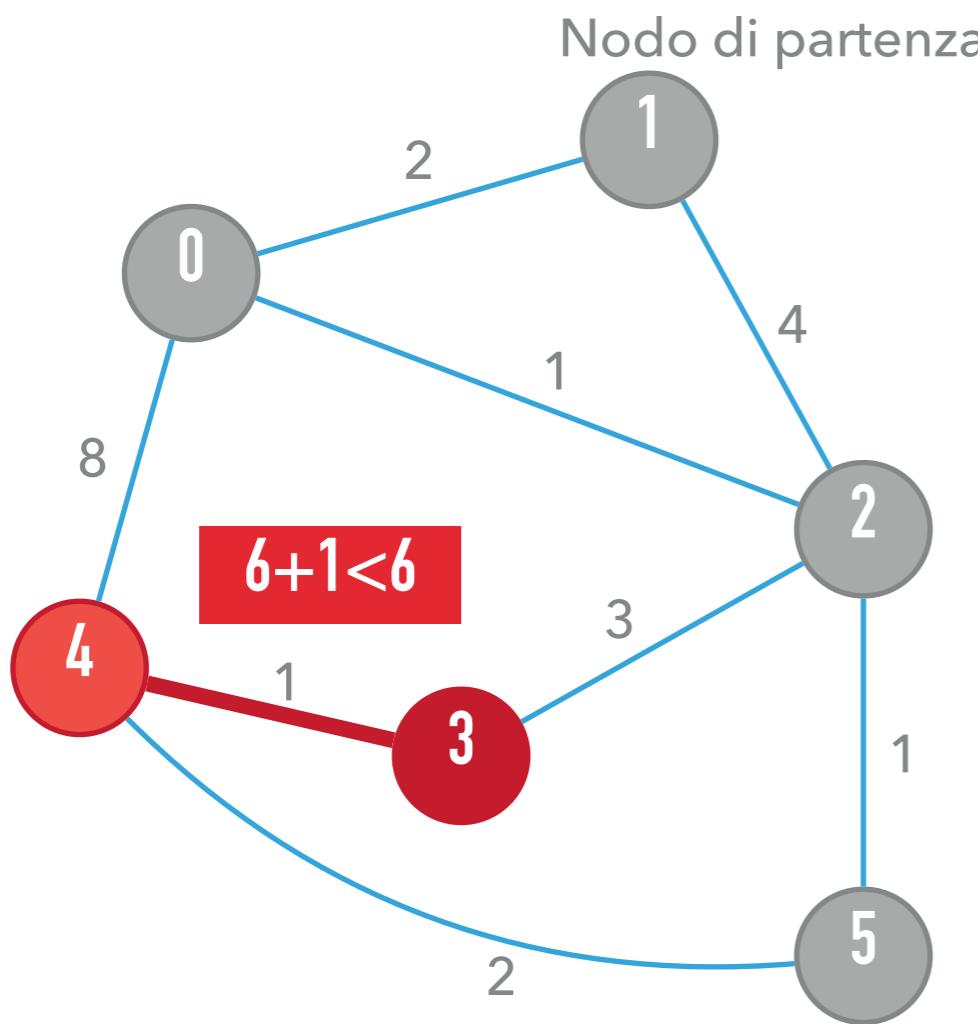
Nodo con distanza minima: 3

Lista dei nodi non visitati

4

| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE



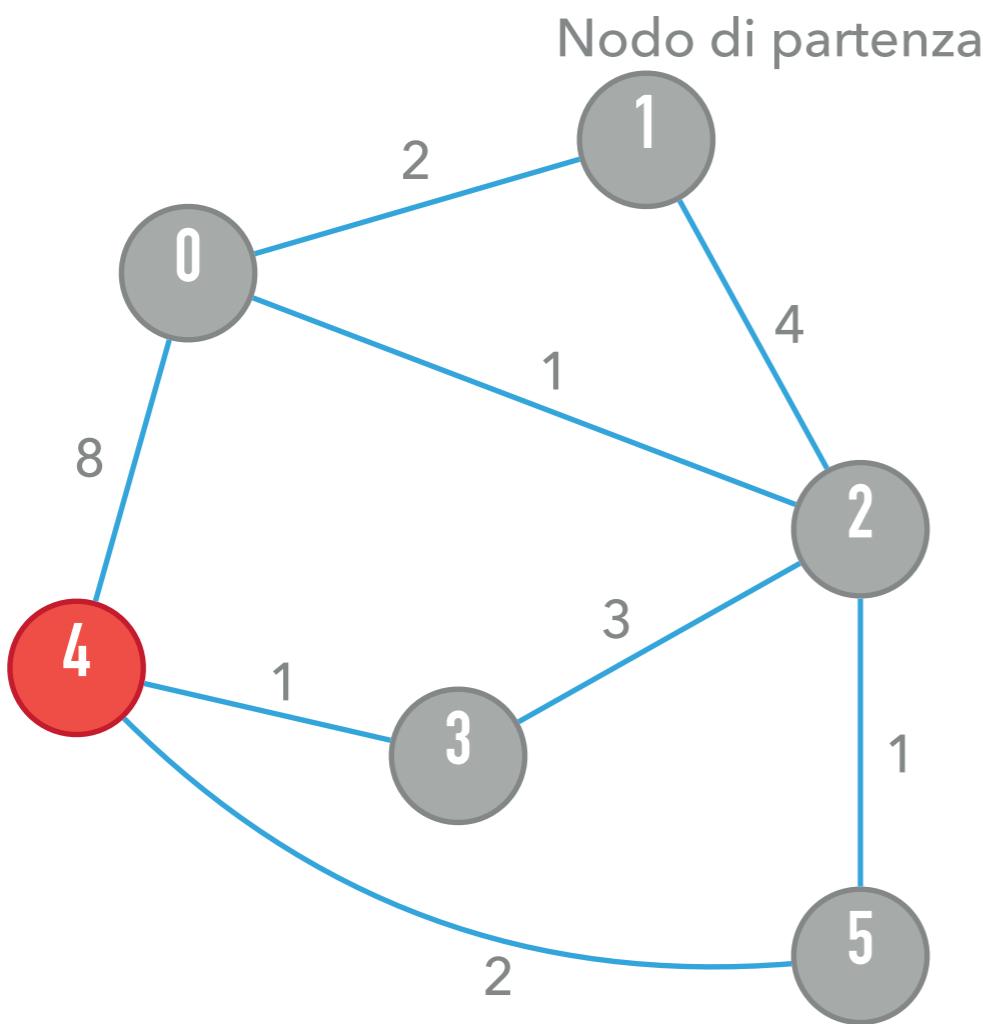
Nodo con distanza minima: 3

Lista dei nodi non visitati

4

| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE

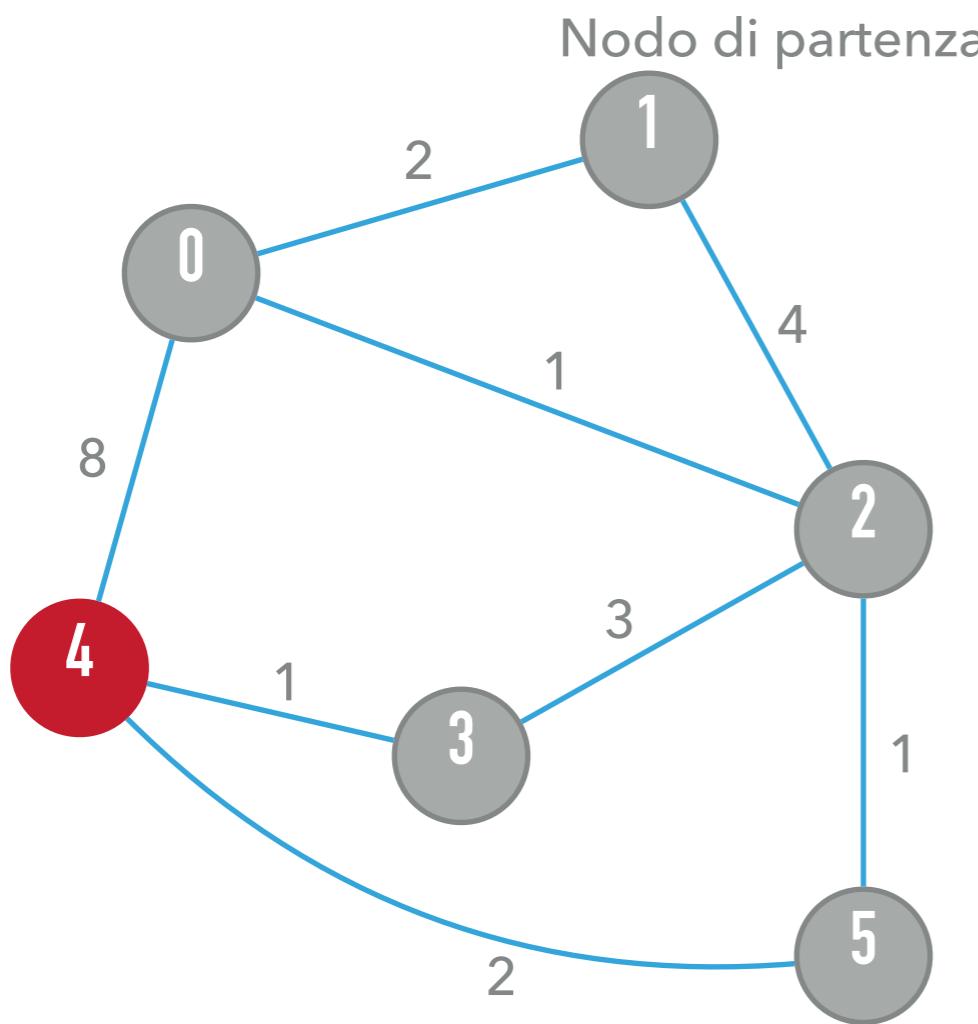


Lista dei nodi non visitati

4

| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE



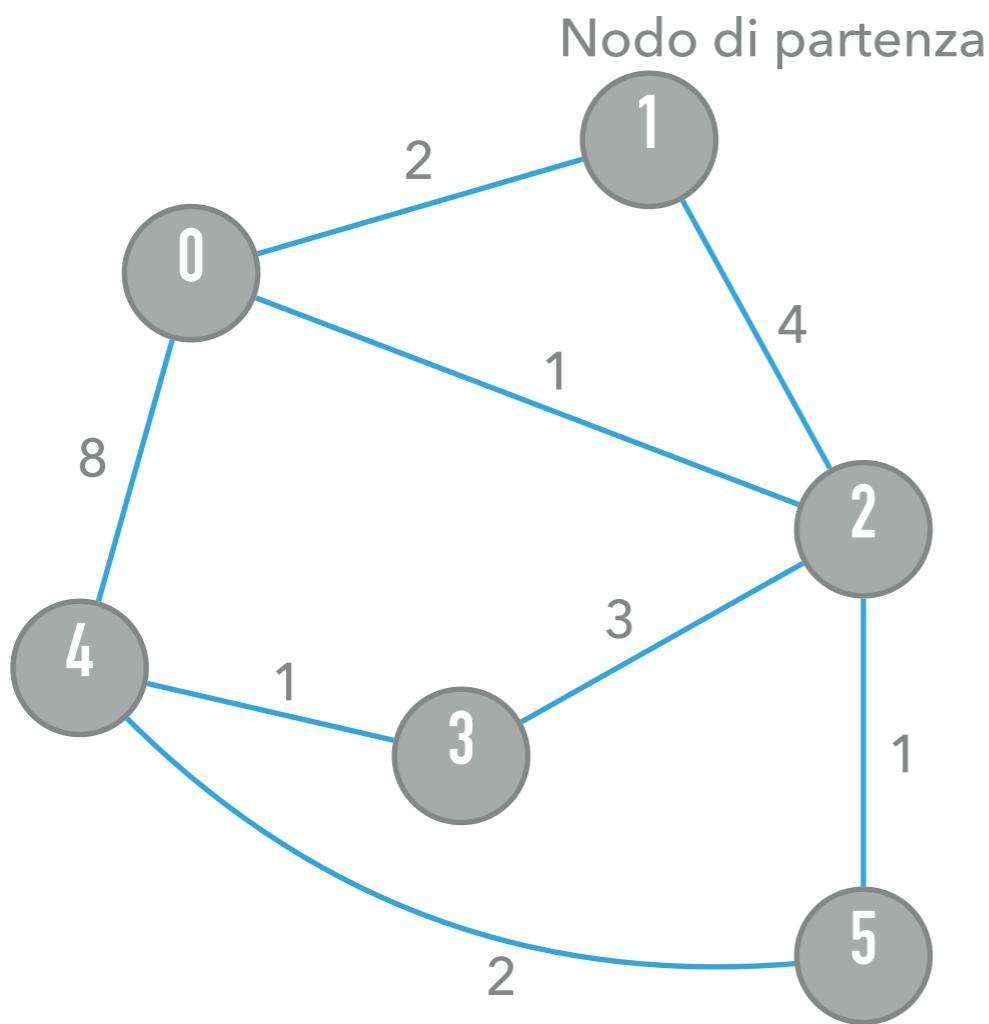
Nodo con distanza minima:

4

Lista dei nodi non visitati

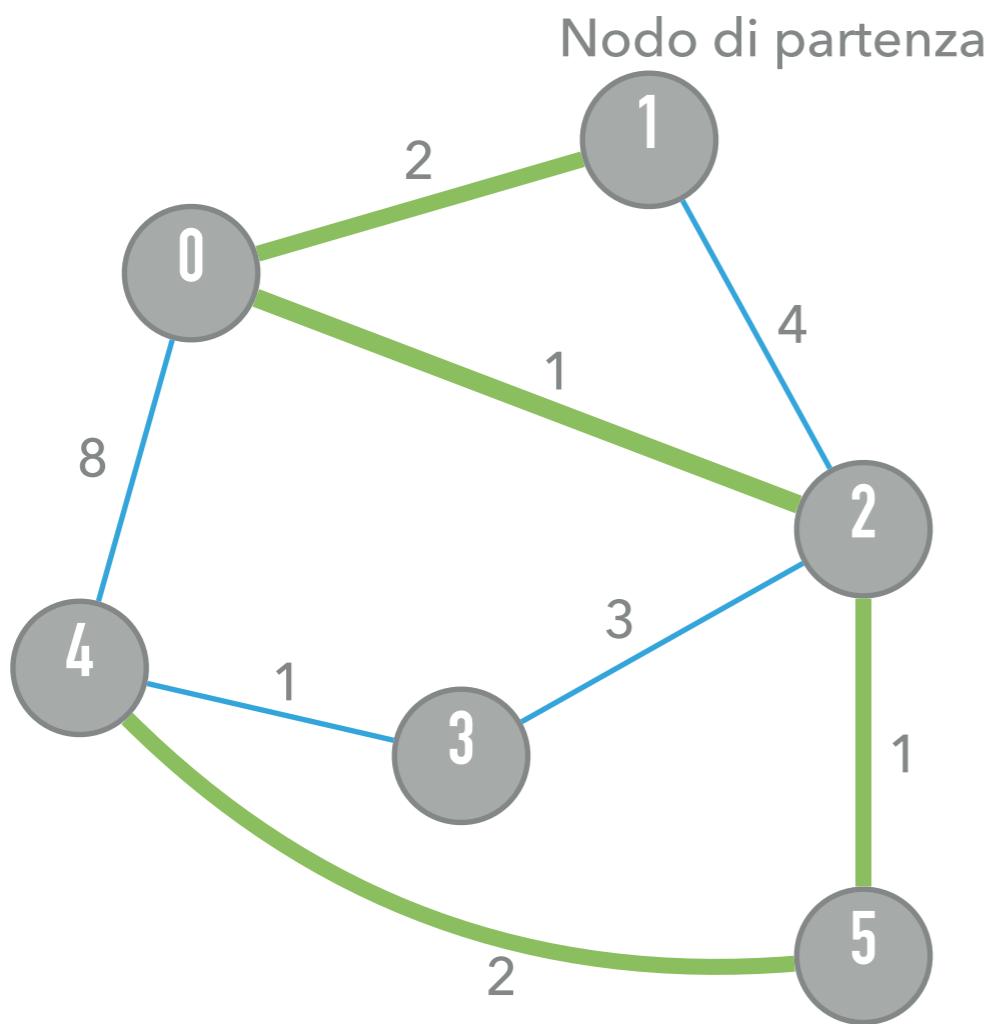
| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE



| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

ESEMPIO DI ESECUZIONE

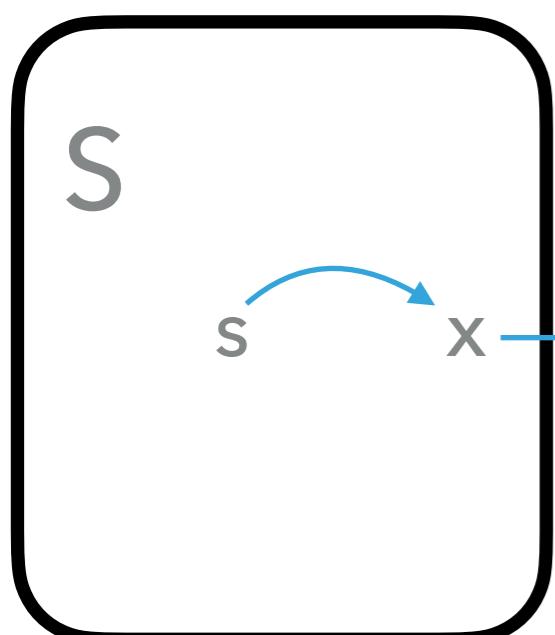


Percorso di lunghezza minima
per andare dal nodo (1) al nodo (4)

| Vertice | Peso |
|---------|------|
| 0 | 2 |
| 1 | 0 |
| 2 | 3 |
| 3 | 6 |
| 4 | 6 |
| 5 | 4 |

CORRETTEZZA DI DIJKSTRA

- ▶ Alla fine di Dijkstra, $\text{distanza}[v] = d[v] = \delta(s, v), \forall v \in V$
- ▶ Teorema: $\forall u$, all'inserimento di u in S , vale $d[u] = \delta(s, u)$.
- ▶ Per assurdo, sia u il primo vertice inserito in S che viola questa proprietà, e y il primo vertice su un **cammino minimo** da s a u non ancora in S all'inserimento di u .



- ▶ Quindi $y \in Q$ e $d[y] \geq d[u]$
- ▶ $x \in S \Rightarrow d[x] = \delta(s, x)$, per scelta di u
- ▶ Il cammino $s \rightarrow x \rightarrow y$ è minimo, ergo $d[y] = \delta(s, x) + w_{x,y} = \delta(s, y)$, perché x è stato rilassato e valeva $d[x] = \delta(s, x)$ in tal momento.
- ▶ Da $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$ e $d[u] \leq d[y]$ (scelta di u) segue $d[u] = \delta(s, u)$, assurdo.

CONSIDERAZIONI SUL TEMPO DI CALCOLO

- ▶ Visitiamo ogni nodo al più una volta:
quando lo estraiamo dalla lista
- ▶ Visitiamo ogni arco al più una volta:
la prima volta che incontriamo una delle sue estremità
- ▶ Il tempo di calcolo quindi dipende da questi due valori e dalle operazioni di:

- ▶ Trovare il minimo tra i nodi non visitati
- ▶ Aggiornare la distanza dei nodi

Dipendono da come rappresentiamo il grafo e le altre strutture che ci servono

CONSIDERAZIONI SUL TEMPO DI CALCOLO

- ▶ Se usiamo un array per mantenere le distanze di ogni nodo:
 - ▶ Ogni aggiornamento di distanza richiede $O(1)$, dato che è sufficiente cambiare il valore
 - ▶ Trovare il minimo richiede tempo $O(V)$ perché l'array va scandito
 - ▶ Otteniamo quindi tempo $O(V^2 + E) = O(V^2)$ perché per ogni nodo dobbiamo estrarre il minimo e per ogni arco aggiornare un peso

CONSIDERAZIONI SUL TEMPO DI CALCOLO

- ▶ Se abbiamo $E = o(V^2/\log V)$ possiamo usare un min-heap,
(abbiamo usato un sistema simile nell'algoritmo di heapsort)
- ▶ Nel caso di min-heap:
 - ▶ Rimozione del minimo: $O(\log V)$
 - ▶ Aggiornamento del peso: $O(\log V)$
 - ▶ Otteniamo quindi $O(V \log V + E \log V) = O((V + E)\log V)$
che è meglio dell'implementazione come array quando
abbiamo $o(V^2/\log V)$ archi

ALGORITMO DI FLOYD-WARSHALL

ALGORITMI E STRUTTURE DATI

ALGORITMO DI FLOYD-WARSHALL

- ▶ L'algoritmo di Floyd-Warshall è un algoritmo per trovare il percorso più corto tra ogni coppia di nodi in un grafo
- ▶ Ovvero, dato un grafo $G = (V, E)$ avremo come risultato una tabella D in cui in posizione (i, j) è la lunghezza del percorso di peso/costo minimo tra i e j .
- ▶ Il grafo può essere orientato, pesato e anche con pesi negativi sugli archi

ALGORITMO DI FLOYD-WARSHALL

- ▶ Supponiamo che i nodi corrispondano agli interi $\{0, \dots, k\}$
- ▶ Proviamo a definire in modo ricorsivo il percorso tra due nodi i e j
- ▶ Per fare questo consideriamo una restrizione sui nodi intermedi che possono essere contenuti nel percorso tra i e j
- ▶ Indichiamo con $d_{i,j}^k$ la lunghezza del percorso più corto tra i e j che contenga nodi intermedi solo nodi in $\{0, \dots, k - 1\}$

ALGORITMO DI FLOYD-WARSHALL

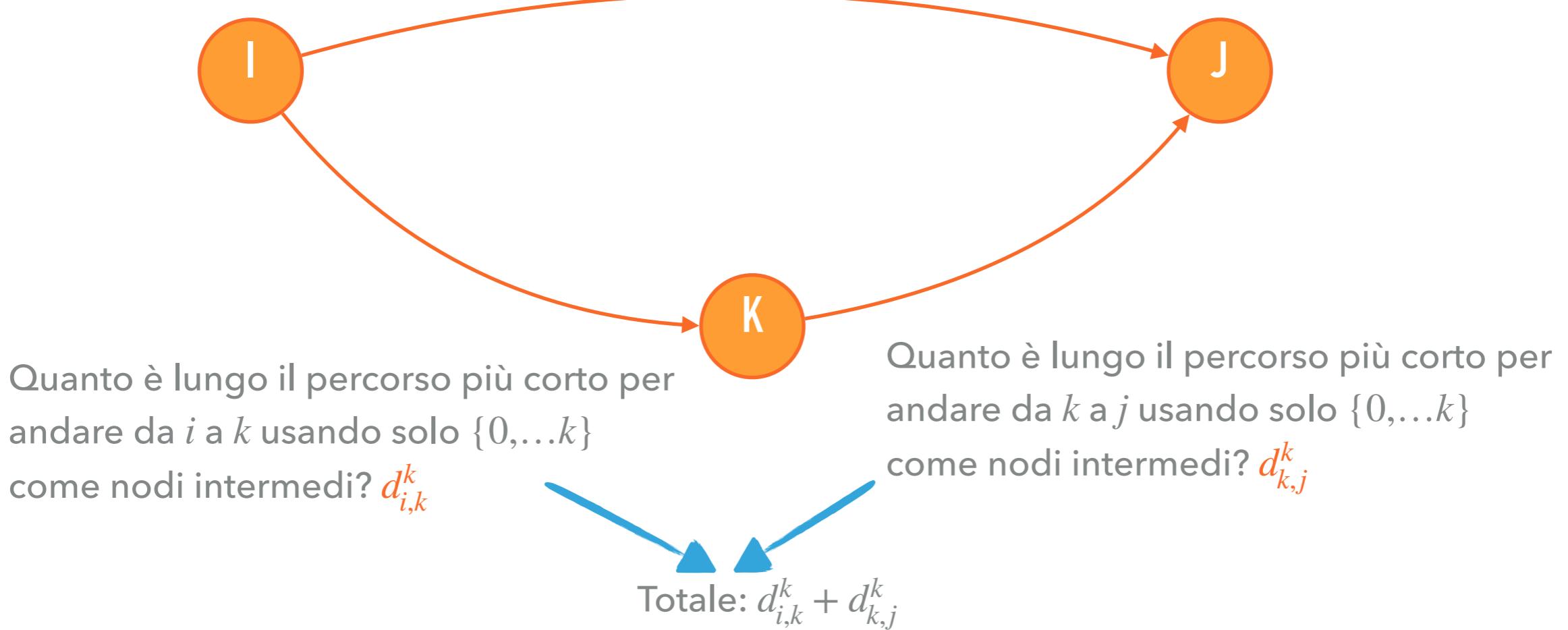
- ▶ Il caso base è quando non possiamo avere nodi intermedi, quindi $d_{i,j}^0$.
- ▶ Dato che non possiamo avere nodi intermedi ci sono solo tre possibilità:
 - ▶ $d_{i,i}^0 = 0$, dato che nodo di destinazione e partenza sono lo stesso
 - ▶ $d_{i,j}^0 = w_{i,j}$ se esiste un arco tra i e j
 - ▶ $d_{i,j}^0 = +\infty$ se nessun arco connette i con j

ALGORITMO DI FLOYD-WARSHALL

- ▶ Per un generico $d_{i,j}^{k+1}$ abbiamo due casi:
 - ▶ Il nodo k non viene usato nel percorso di lunghezza minima per andare da i a j : in quel caso $d_{i,j}^{k+1} = d_{i,j}^k$
 - ▶ Il nodo k viene usato nel percorso di lunghezza minima per andare da i a j . Dato che un nodo può apparire una sola volta nel percorso non useremo k per andare da i a k e per andare da k a j , quindi $d_{i,j}^{k+1} = d_{i,k}^k + d_{k,j}^k$

ALGORITMO DI FLOYD-WARSHALL

Quanto è lungo il percorso più corto per andare da i a j usando solo $\{0, \dots, k\}$ come nodi intermedi e senza passare per k ? $d_{i,j}^k$



ALGORITMO DI FLOYD-WARSHALL

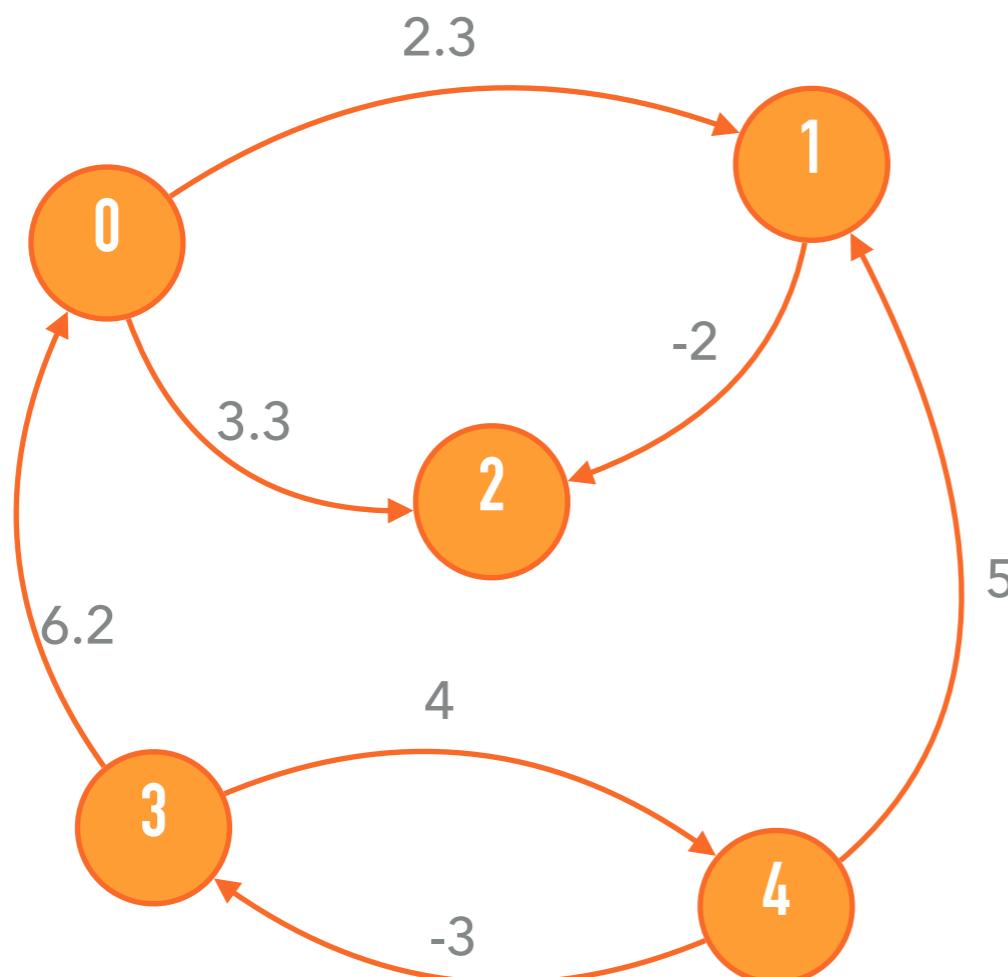
- ▶ Abbiamo quindi che $d_{i,j}^{k+1} = \min\{d_{i,j}^k, d_{i,k}^k + d_{k,j}^k\}$
dato che abbiamo solo due scelte possibili e vogliamo il percorso di lunghezza/peso minimo
- ▶ Possiamo iniziare da $k = 0$, in cui conosciamo tutti i valori, fino ad arrivare a $k = n$, in cui possiamo utilizzare tutti gli n vertici del grafo come nodi intermedi

PSEUDOCODICE: FLOYD-WARSHALL

Parametri: Matrice di adiacenza del grafo W di n nodi con $+\infty$ dove gli archi sono assenti

```
D = array di dimensione  $(n + 1) \times n \times n$  # contiene tutti i  $d_{i,j}^k$ 
D[0] = W # la matrice W contiene già i casi base
for k in range(0,n):
    # calcoliamo  $d_{i,j}^{k+1}$  al variare di  $i$  e  $j$ 
    for i in range(0,n):
        for j in range(0,n):
            D[k+1][i][j] = min(D[k][i][j], D[k][i][k] + D[k][k][j])
return D[n]
```

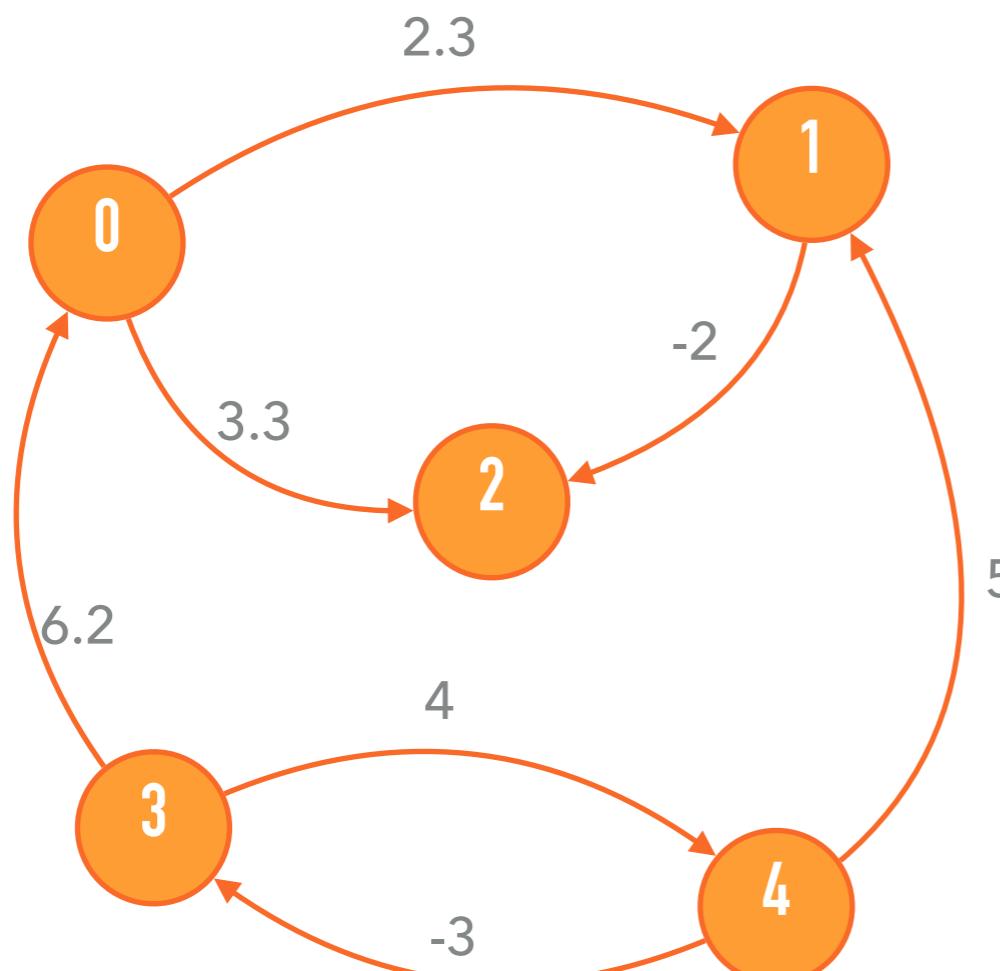
ALGORITMO DI FLOYD-WARSHALL



$W=D[0]=$

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 2.3 | 3.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | ∞ | ∞ | 0 | 4 |
| ∞ | 5 | ∞ | -3 | 0 |

ALGORITMO DI FLOYD-WARSHALL



Possiamo passare per il nodo 0

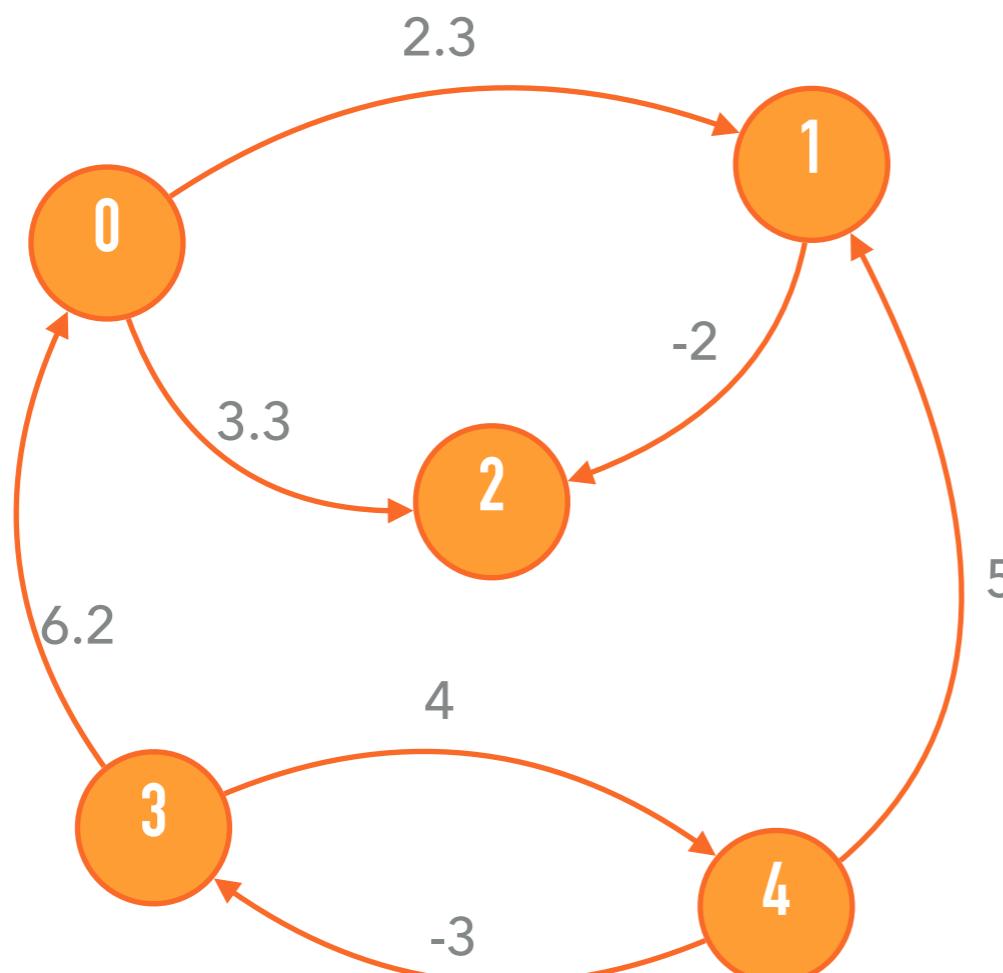
$W = D[0] =$

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 2.3 | 3.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | ∞ | ∞ | 0 | 4 |
| ∞ | 5 | ∞ | -3 | 0 |

$D[1] =$

| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 2.3 | 3.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 9.5 | 0 | 4 |
| ∞ | 5 | ∞ | -3 | 0 |

ALGORITMO DI FLOYD-WARSHALL

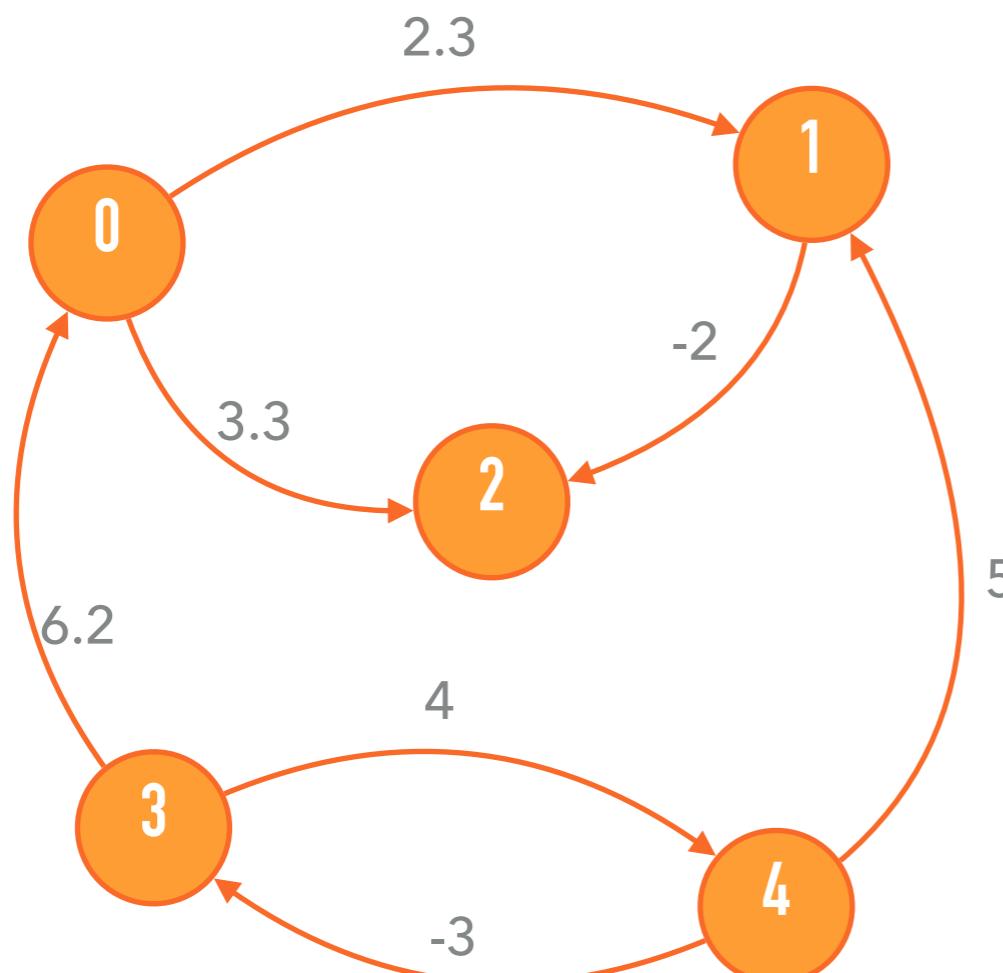


| | | | | |
|----------|----------|----------|----------|----------|
| 0 | 2.3 | 3.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 9.5 | 0 | 4 |
| ∞ | 5 | ∞ | -3 | 0 |

| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| ∞ | 5 | 3 | -3 | 0 |

Possiamo passare per i nodi 0 e 1

ALGORITMO DI FLOYD-WARSHALL

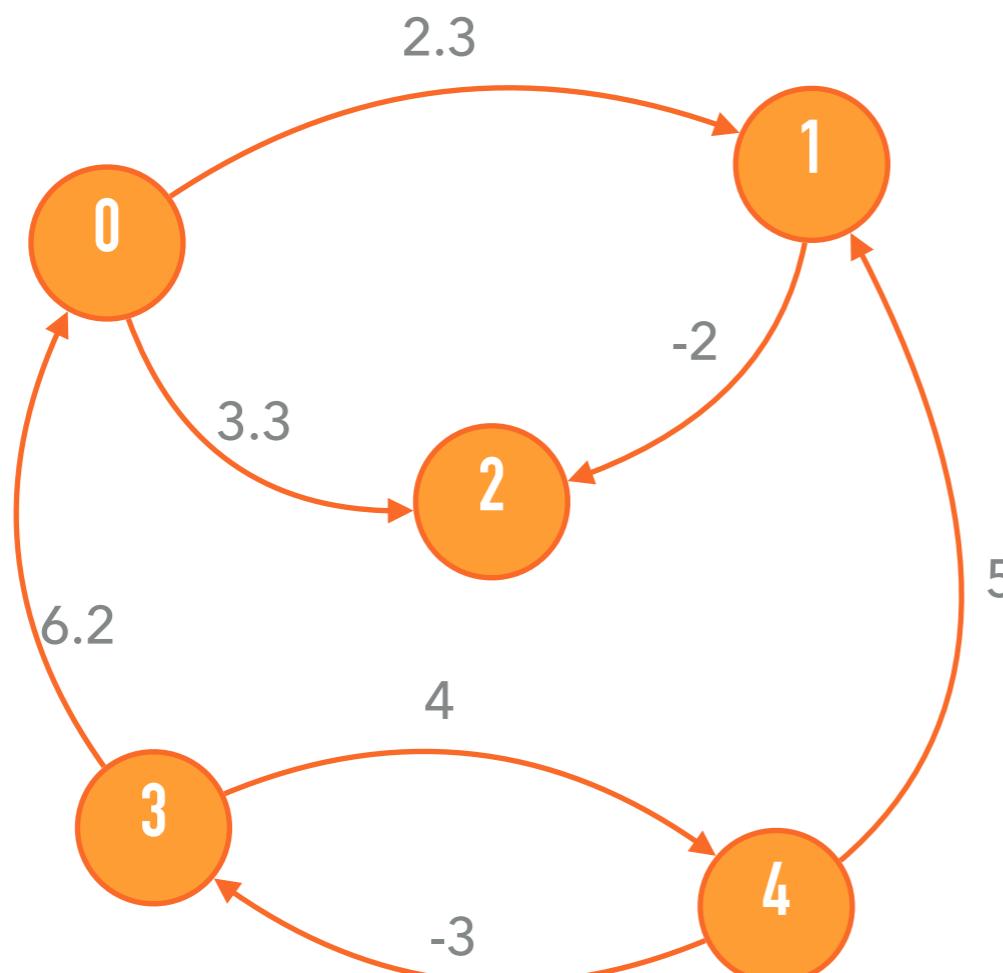


| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | infinity | infinity |
| infinity | 0 | -2 | infinity | infinity |
| infinity | infinity | 0 | infinity | infinity |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| infinity | 5 | 3 | -3 | 0 |

| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | infinity | infinity |
| infinity | 0 | -2 | infinity | infinity |
| infinity | infinity | 0 | infinity | infinity |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| infinity | 5 | 3 | -3 | 0 |

Possiamo passare per i nodi 0, 1 e 2

ALGORITMO DI FLOYD-WARSHALL

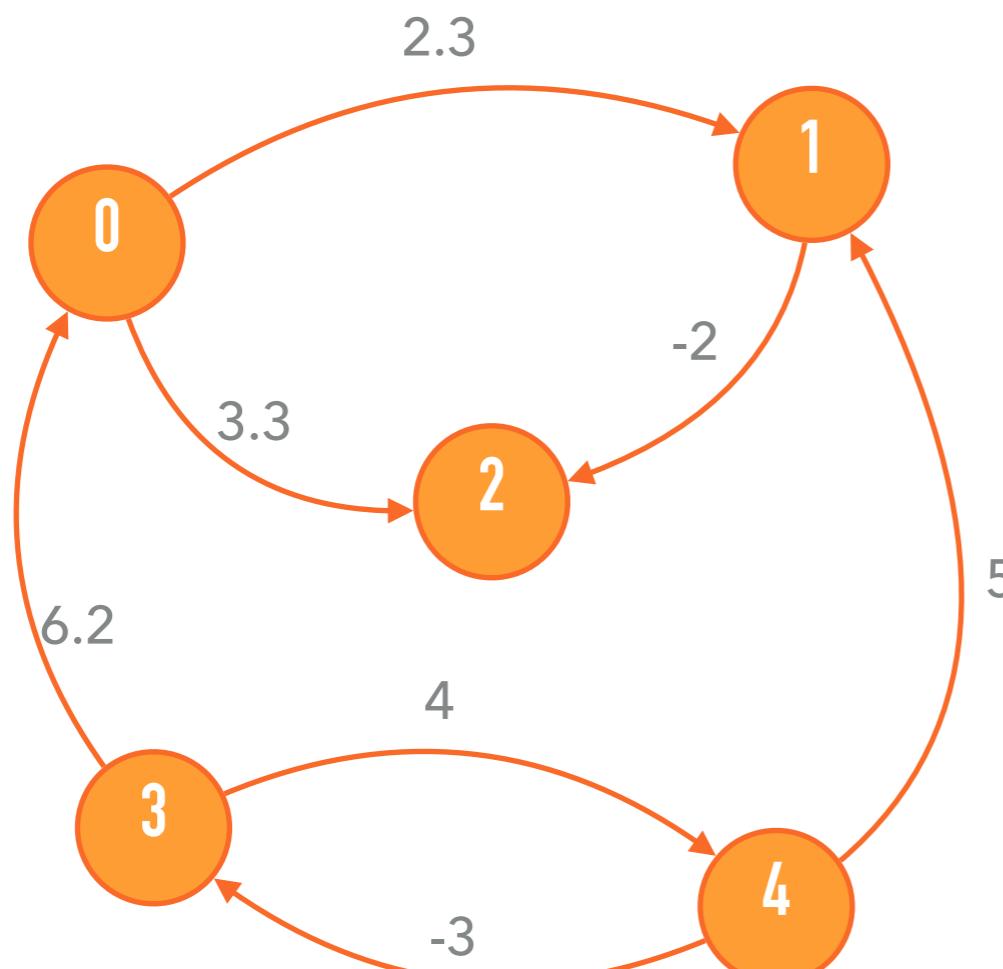


| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| ∞ | 5 | 3 | -3 | 0 |

| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| 3.2 | 5 | 3 | -3 | 0 |

Possiamo passare per i nodi 0, 1, 2 e 3

ALGORITMO DI FLOYD-WARSHALL



Possiamo passare per i nodi 0, 1, 2, 3 e 4

| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| 3.2 | 5 | 3 | -3 | 0 |

| | | | | |
|----------|----------|-----|----------|----------|
| 0 | 2.3 | 0.3 | ∞ | ∞ |
| ∞ | 0 | -2 | ∞ | ∞ |
| ∞ | ∞ | 0 | ∞ | ∞ |
| 6.2 | 8.5 | 6.5 | 0 | 4 |
| 3.2 | 5 | 3 | -3 | 0 |

D[5]=

Matrice con la lunghezza dei percorsi di costo minimo tra tutte le coppie di vertici

ALGORITMO DI FLOYD-WARSHALL: COMPLESSITÀ

- ▶ Abbiamo tre cicli for innestati e ognuno di questi cicli for svolge $|V| = n$ iterazioni
- ▶ Ne segue che il tempo di esecuzione è $O(V^3)$, quindi cubico
- ▶ Notiamo che rispetto all'algoritmo di Bellman-Ford, che richiede tempo $O(VE)$ con l'algoritmo di Floyd-Warshall otteniamo la distanza tra ogni coppia di vertici, non solo a partire da un nodo sorgente dato

ALGORITMO DI FLOYD-WARSHALL: VARIANTI

- ▶ Possiamo aggiungere molteplici restrizioni e varianti all'algoritmo di Floyd-Warshall a seconda delle necessità
- ▶ Come esempio, pensiamo di voler stabilire per ogni coppia di nodi i e j il cammino semplice meno costoso per andare da i a j che però contenga un numero pari di nodi
- ▶ Come possiamo modificare l'algoritmo?

ALGORITMO DI FLOYD-WARSHALL: VARIANTI

- ▶ Teniamo traccia di una informazione aggiuntiva: se il cammino che abbiamo è con un numero pari o dispari di nodi:
 - ▶ $d_{i,j}^{k,\text{pari}}$ è il costo minimo per andare da i a j passando per un numero pari di nodi usando solo $\{0, \dots, k - 1\}$ come nodi intermedi
 - ▶ $d_{i,j}^{k,\text{dispari}}$ è definito in modo simile ma con un numero dispari di nodi

ALGORITMO DI FLOYD-WARSHALL: VARIANTI

- ▶ I casi base sono:
 - ▶ $d_{i,i}^{0,\text{pari}} = 0$
 - ▶ $d_{i,j}^{0,\text{pari}} = w_{i,j}$ se esiste l'arco (i, j)
 - ▶ $d_{i,j}^{0,\text{pari}} = +\infty$ altrimenti
 - ▶ $d_{i,j}^{0,\text{dispari}} = +\infty$ dato che senza nodi intermedi ogni percorso ha solo un numero pari di nodi intermedi

ALGORITMO DI FLOYD-WARSHALL: VARIANTI

- ▶ Il calcolo delle soluzioni di sotto-problemi cambia solo per mantenere corretta la parità:
- ▶ $d_{i,j}^{k+1,\text{pari}} = \min\{d_{i,j}^{k,\text{pari}}, d_{i,k}^{k,\text{dispari}} + d_{k,j}^{k,\text{pari}}, d_{i,k}^{k,\text{pari}} + d_{k,j}^{k,\text{dispari}}\}$
- ▶ $d_{i,j}^{k+1,\text{dispari}} = \min\{d_{i,j}^{k,\text{dispari}}, d_{i,k}^{k,\text{pari}} + d_{k,j}^{k,\text{pari}}, d_{i,k}^{k,\text{dispari}} + d_{k,j}^{k,\text{dispari}}\}$
- ▶ Ovvero trattiamo in modo esplicito come combinare percorsi di parità diversa cambia la parità del percorso finale

ALGORITMO DI FLOYD-WARSHALL: ESERCIZI

- ▶ Alcune varianti che si possono provare a definire, come esercizio:
- ▶ Dato un grafo stabilire il costo del percorso di lunghezza minima che contiene un numero **dispari di archi**.
- ▶ In questo caso cambia un poco la ricorrenza (e i casi base) perché contiamo gli archi e non i nodi