UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

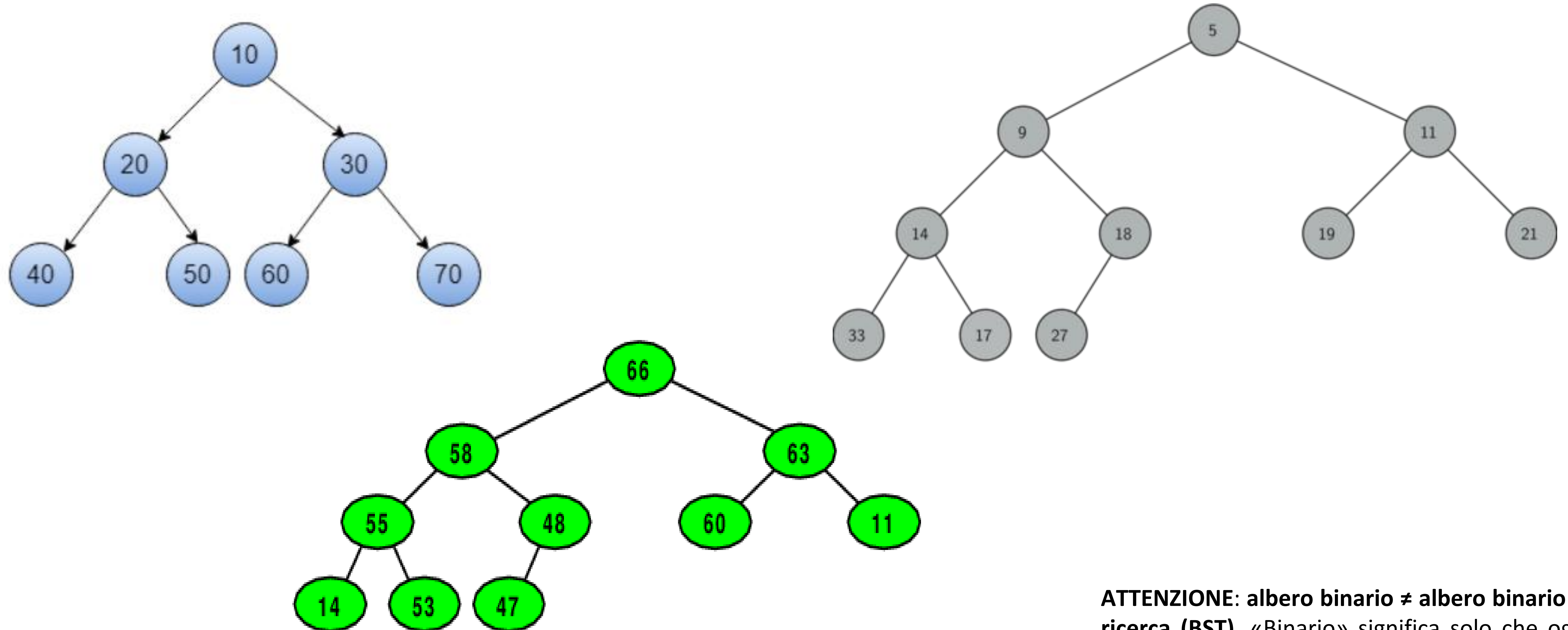
MODULO 2: Heap, Heapsort, Cammini minimi su grafo

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

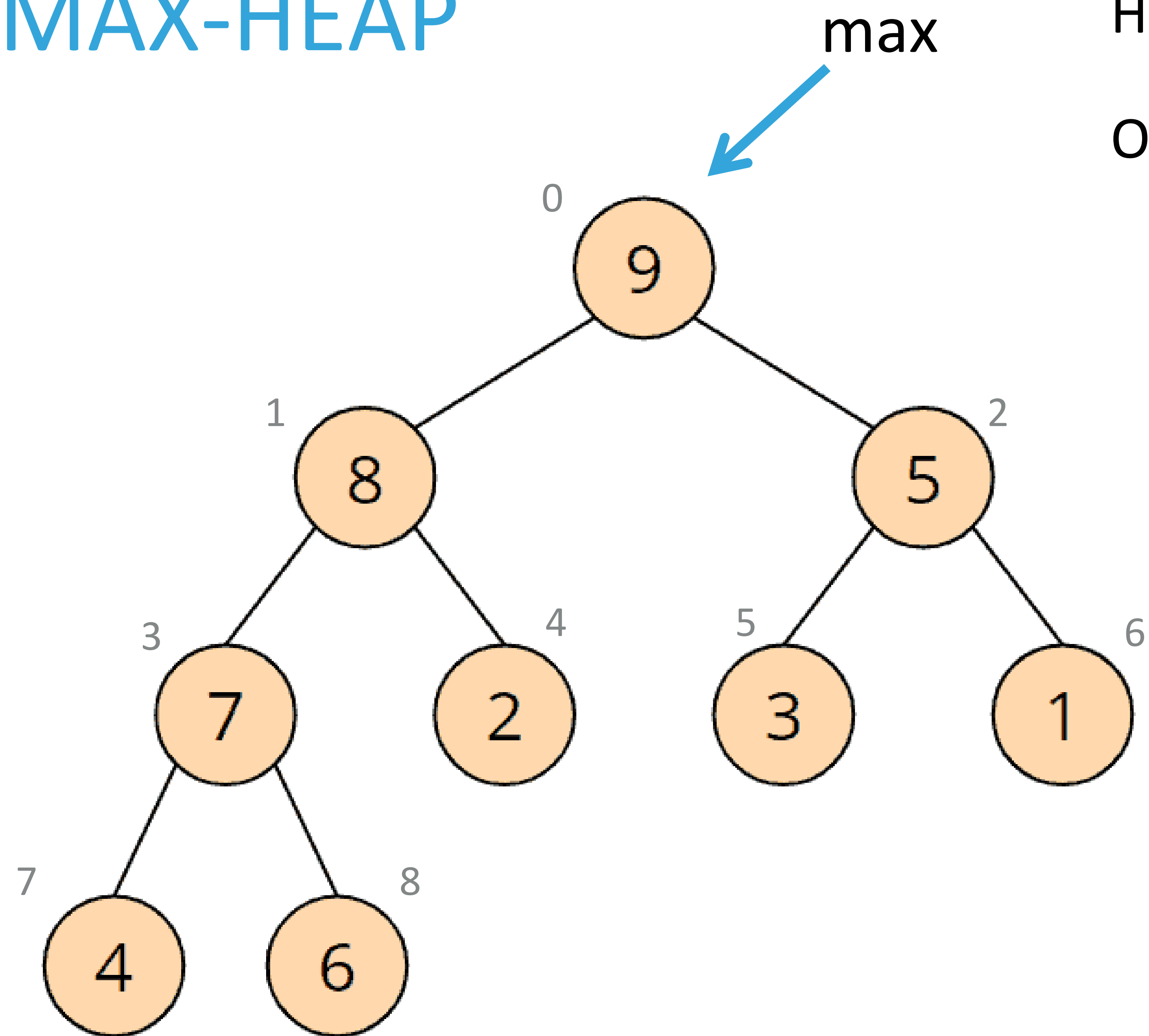
Trieste, 13 maggio 2026

UNA VARIANTE DI ALBERO BINARIO*: HEAP



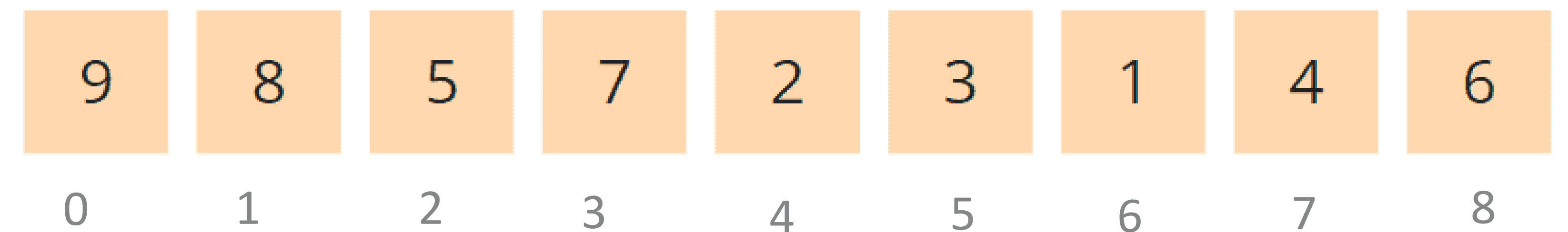
ATTENZIONE: albero binario ≠ albero binario di ricerca (BST). «Binario» significa solo che ogni nodo può avere al massimo 2 figli.

MAX-HEAP



Heap è un albero binario completo o quasi completo

Ogni nodo è maggiore (o minore nel caso min-heap) dei suoi figli.



Memorizzazione su array (non servono puntatori):

Radice (max) ------> posizione 0 su array

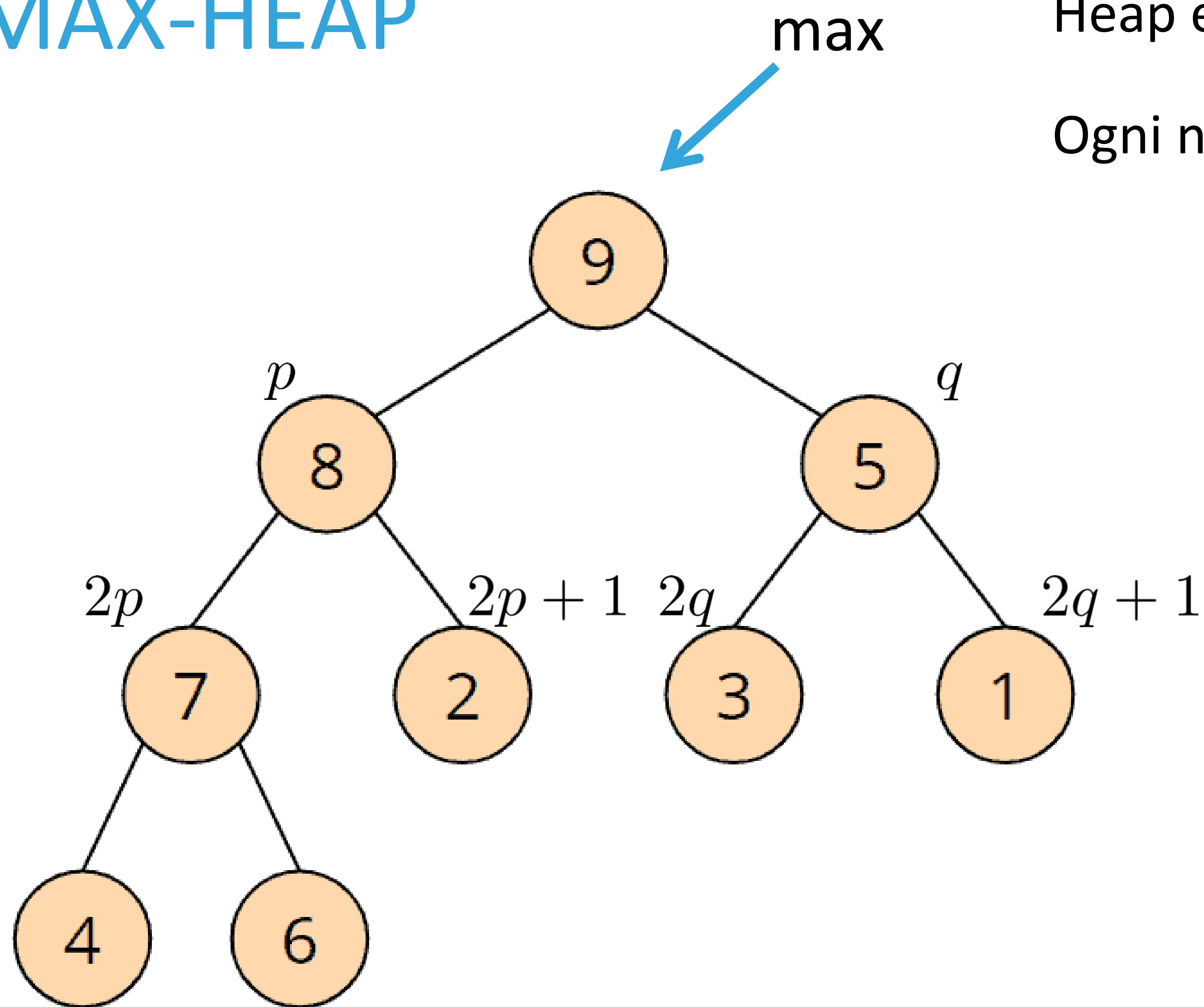
genitore ------> posizione p

figlio sinistro ------> posizione 2p + 1

figlio destro ------> posizione 2p + 2

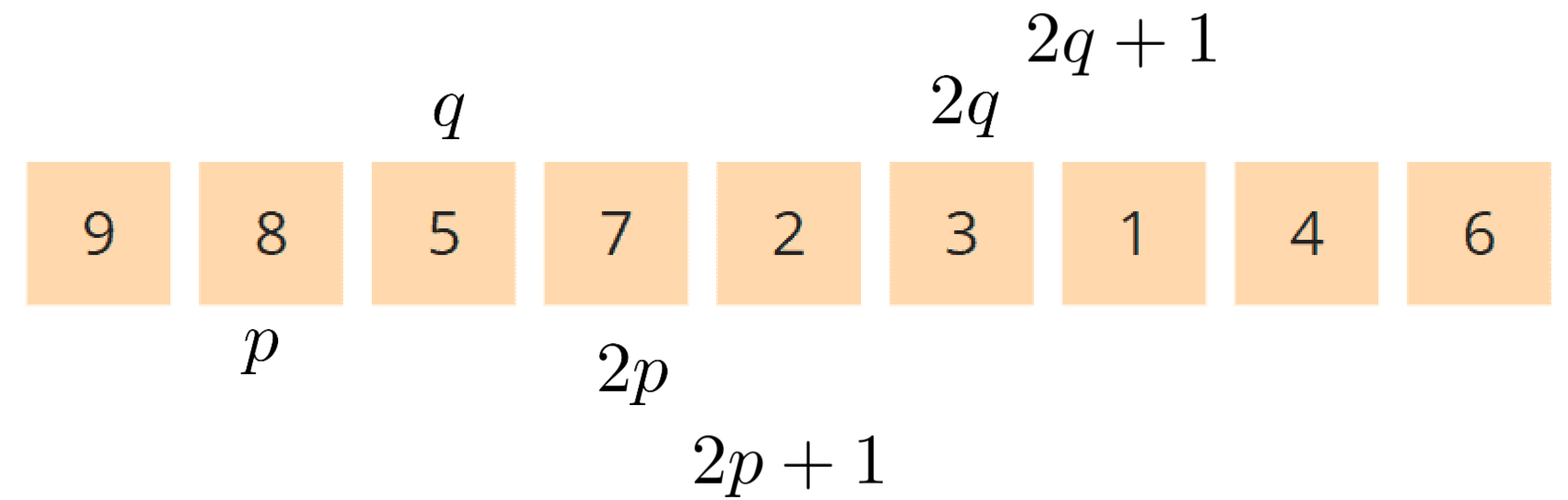
Il massimo (o il minimo) sta nella radice!

MAX-HEAP



Heap è un albero binario completo o quasi completo

Ogni nodo è maggiore (o minore nel caso min-heap) dei suoi figli.



Memorizzazione su array (non servono puntatori):

Radice (max) ------> posizione 0 su array

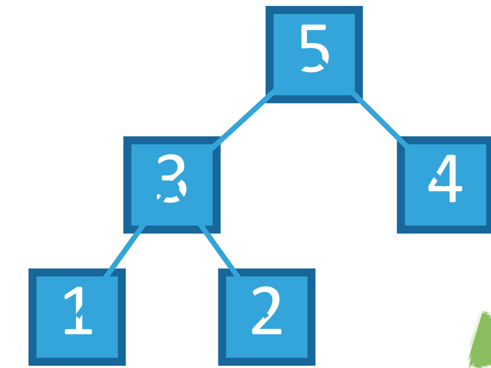
genitore ------> posizione p

figlio sinistro ------> posizione 2p + 1

figlio destro ------> posizione 2p + 2

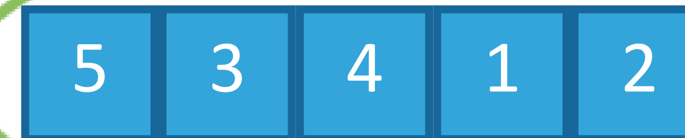
Il massimo (o il minimo) sta nella radice!

ARRAY COME MAX-HEAP: QUIZ



Quale dei seguenti array rappresenta un max-heap?

1)

2) 

3)

4)

HEAPSORT

Nuovo algoritmo di ordinamento per strutture dati lineari (array).

Algoritmo:

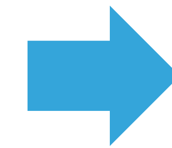
1. l'array viene trasformato in un **max-heap (procedura «heapify»)**;
2. l’elemento più grande (radice dell’heap) viene portato in fondo all’array;
3. si ripete 1. (*heapify()* per ripristinare il max-heap) sull’array di dimensione n-1;
4. *Si ripetono i passi 1.-3.* finchè non rimane un solo elemento da ordinare.

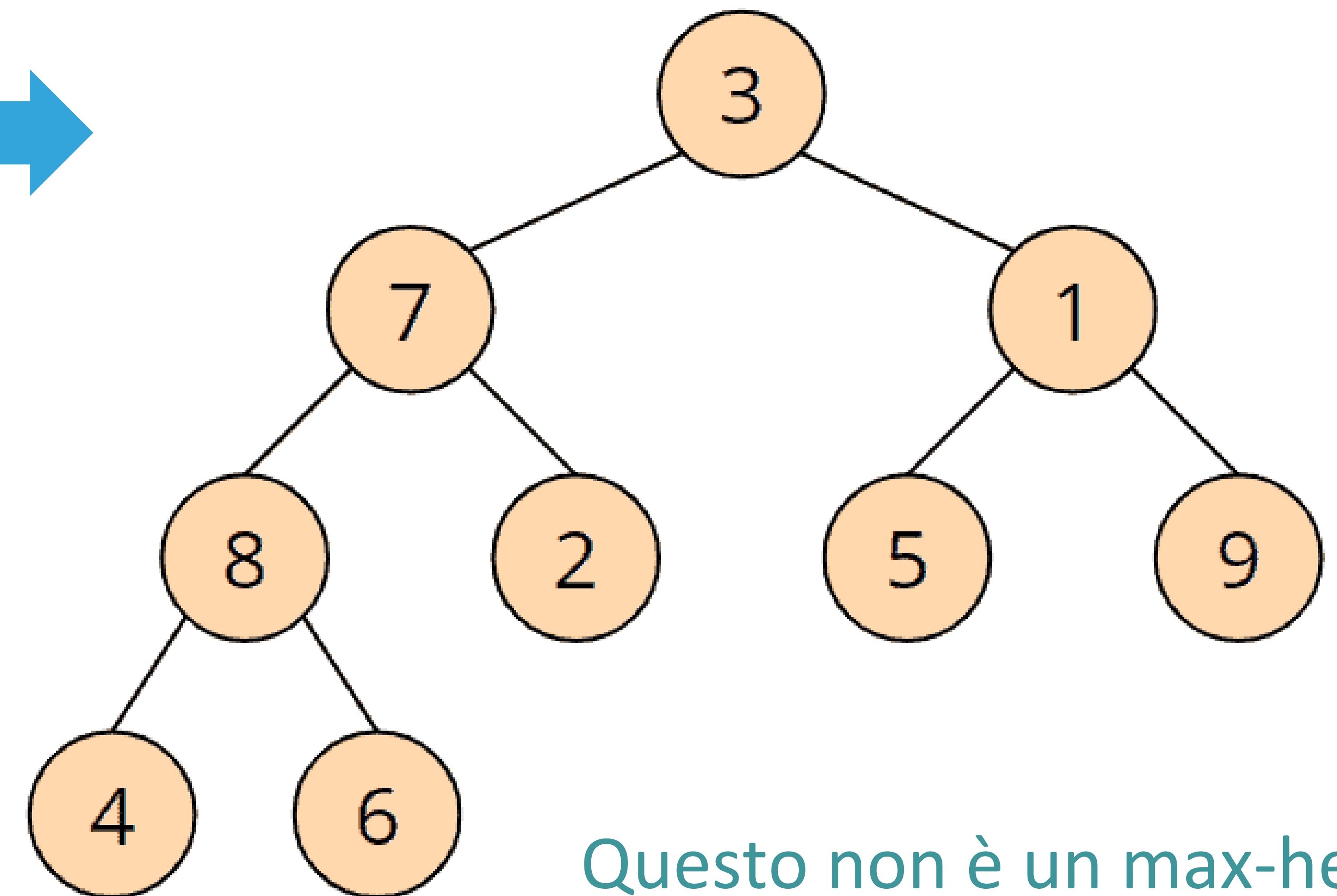
3	7	1	8	2	5	9	4	6

HEAPSORT: COSTRUZIONE HEAP DA ARRAY

Algoritmo:

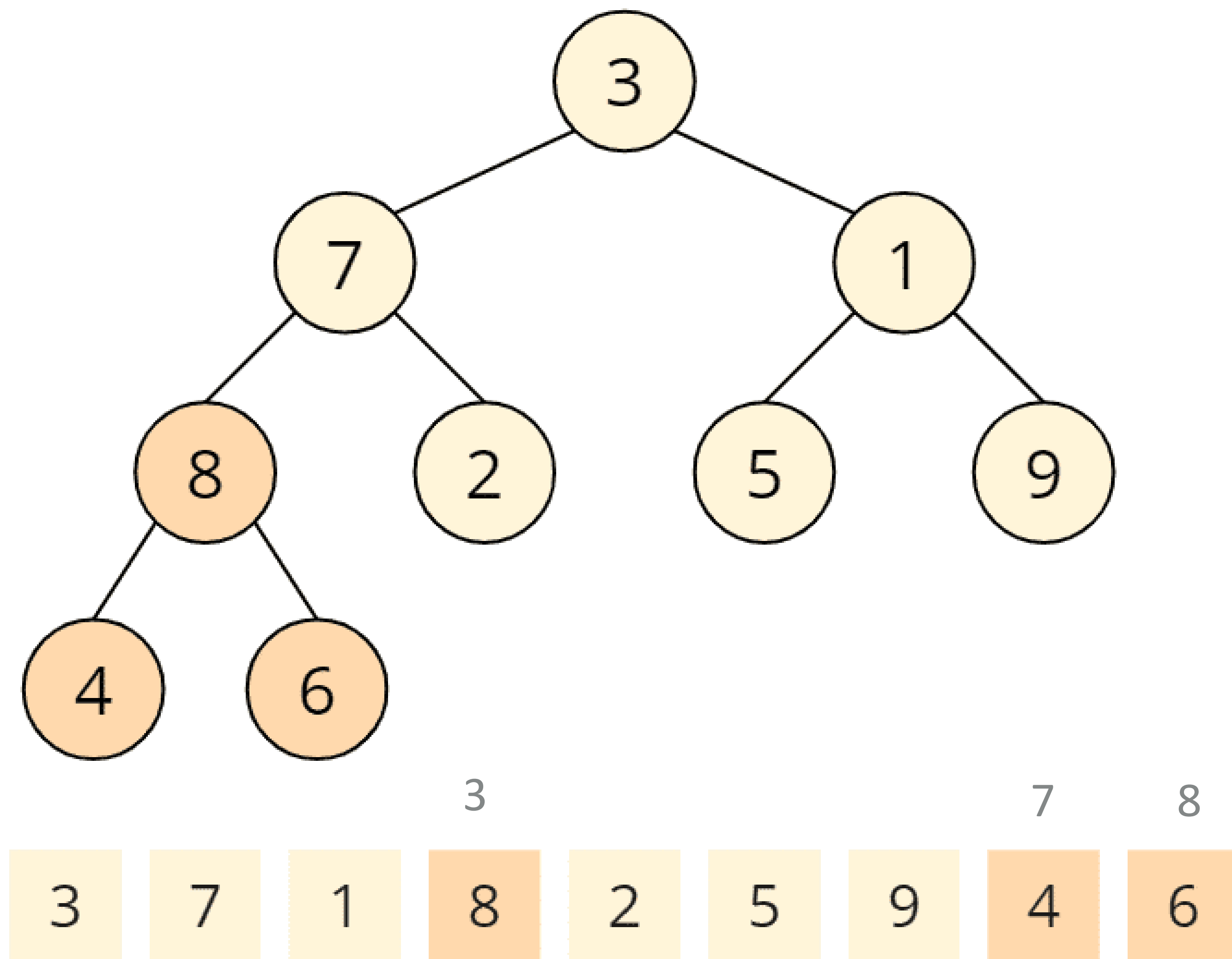
1. l'array da ordinare viene convertito in un **max-heap**;





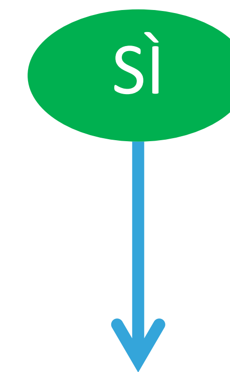
Questo non è un max-heap: quindi.. *heapify()*!

«HEAPIFY»



Ultimo nodo genitore

I suoi figli sono entrambi più piccoli?



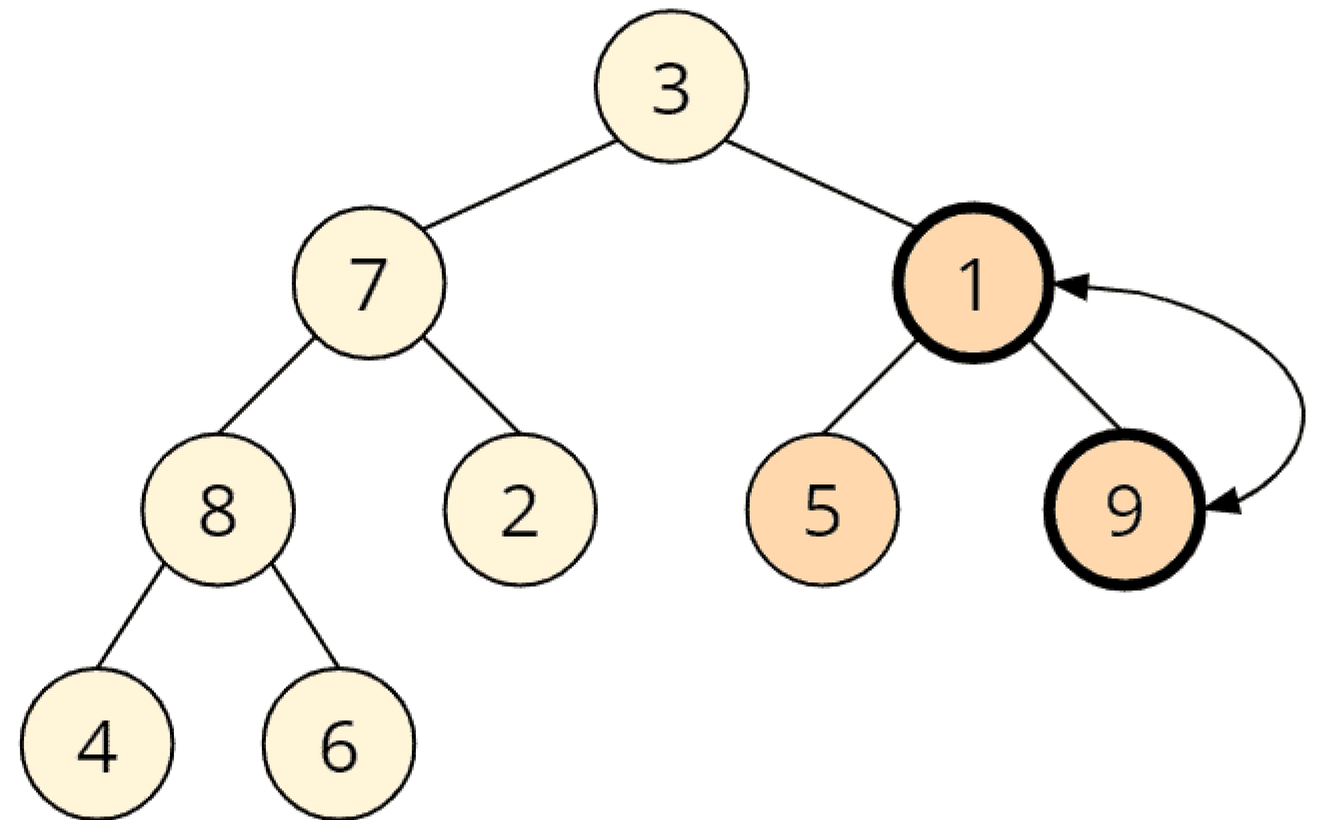
Allora non devo fare altro: il sottoalbero che ha per radice il nodo 8 è già un max-heap.

Nota. Un nodo è genitore se ha almeno un figlio, cioè se $2i+1 < n$. In questo caso $n=9$, quindi i nodi genitori sono nelle posizioni $i=\{0,1,2,3\}$.

«HEAPIFY»

I suoi figli sono entrambi più piccoli?

NO

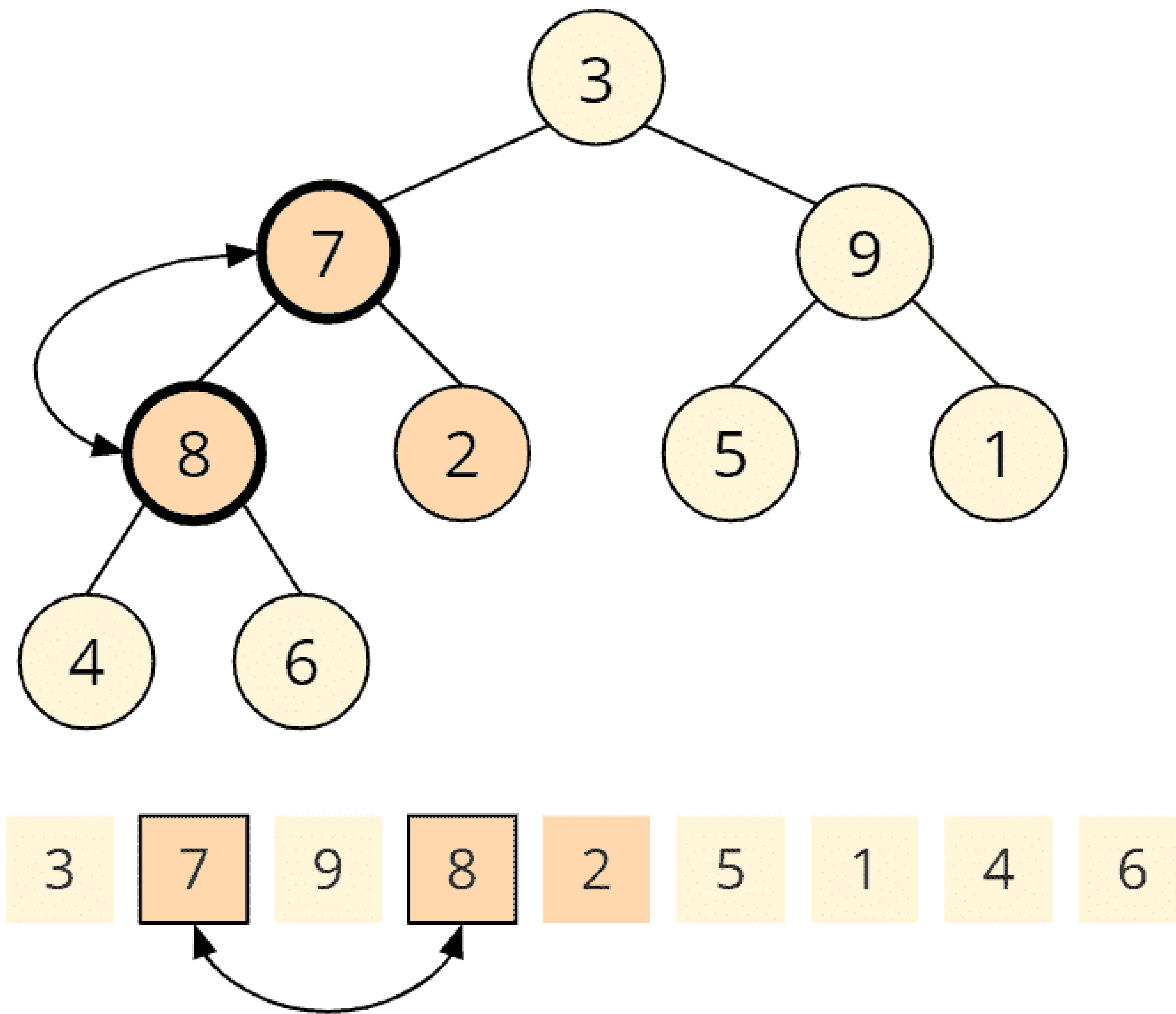


Penultimo nodo genitore

Allora scambio il genitore con il figlio più grande

«HEAPIFY»

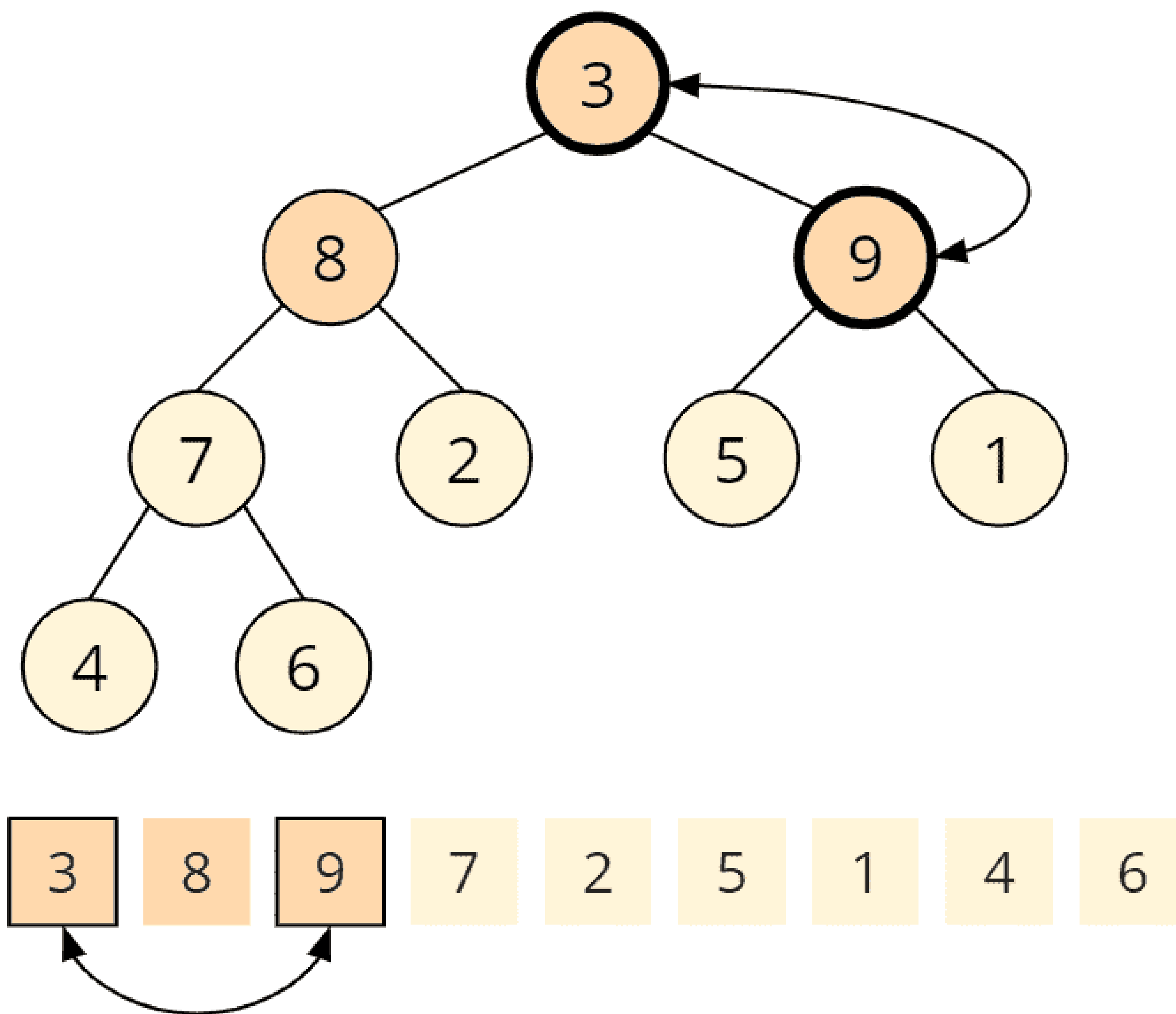
Ripetiamo la stessa procedura per il genitore che era stato inserito prima, ovvero il nodo 7



Controlliamo che questo spostamento continui a verificare la condizione dell'heap: 7 è maggiore dei suoi figli 4 e 6, quindi OK.

«HEAPIFY»

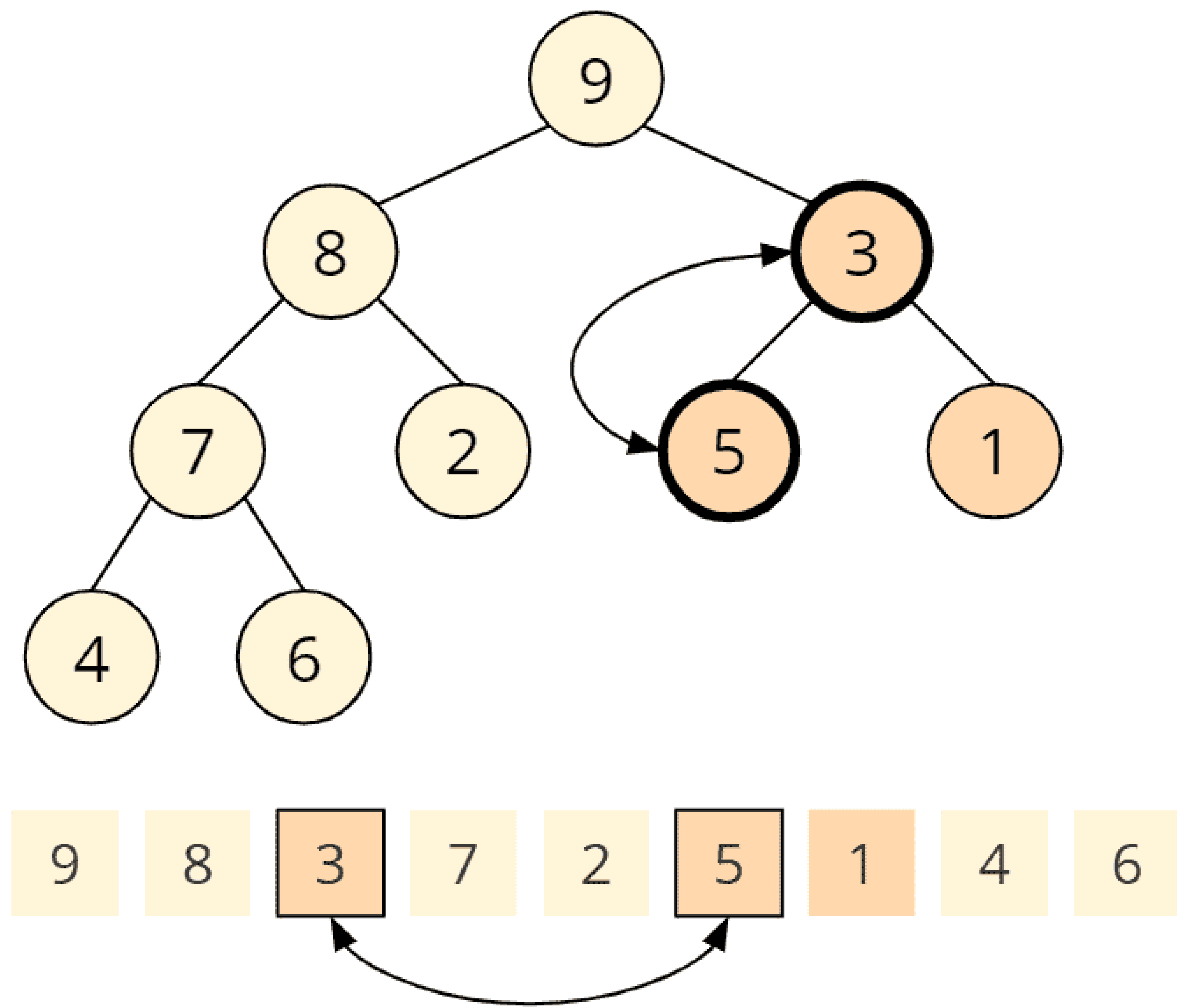
Ripetiamo la stessa procedura per il genitore che era stato inserito prima, ovvero il nodo 3 (la radice)

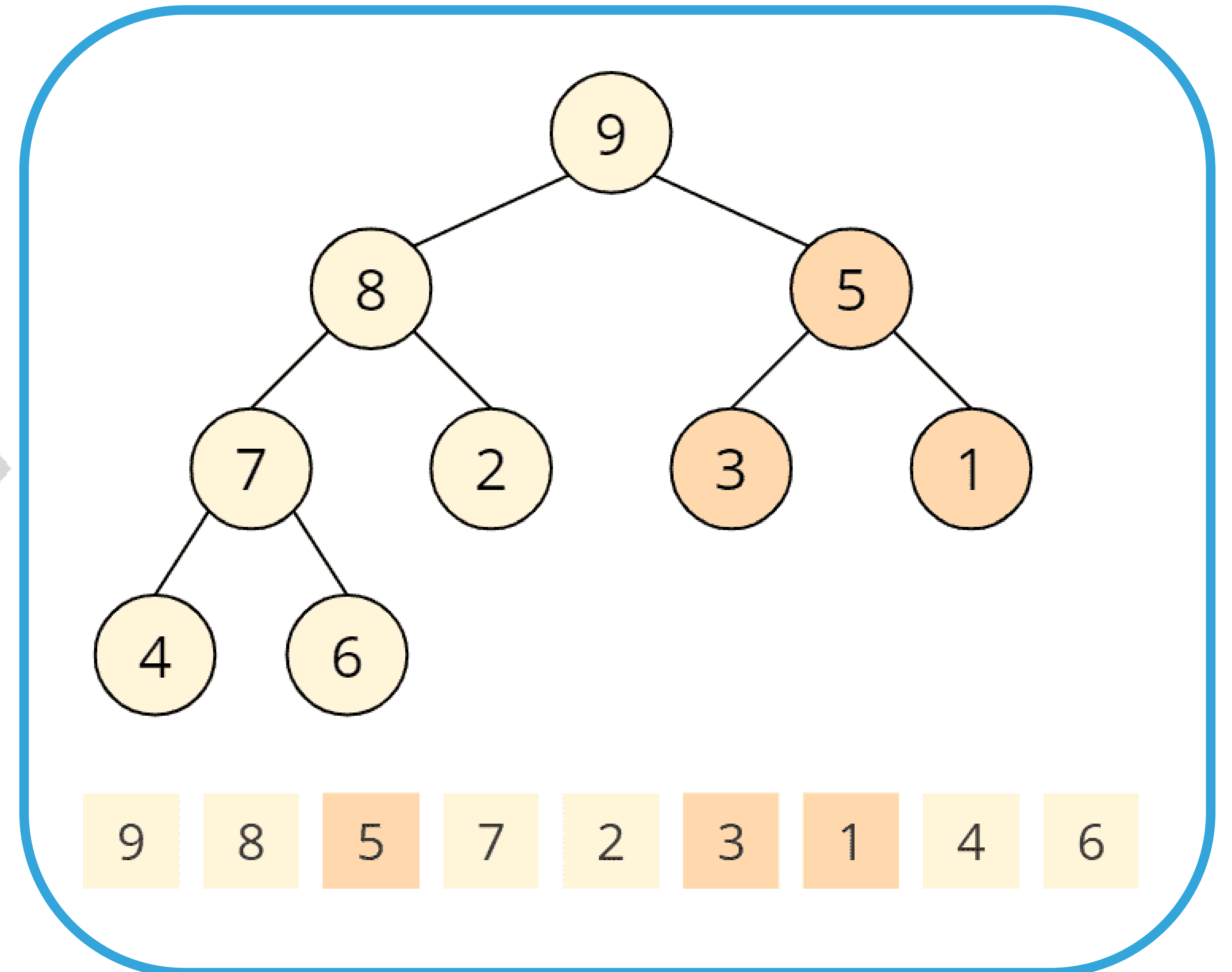


Verifica la condizione dell'heap? 9 > figli 8 e 3. OK!
Questa volta il sottoalbero destro non mantiene la proprietà, per cui va sistemato.

«HEAPIFY»

Sistemiamo il sottoalbero di destra..





Questo è un max-heap!

PSEUDOCODICE (HEAPIFY)

```
MAX-HEAPIFY(A,i)              //input: array A, i=posizione
L = left(i)                   // figlio sinistro di i
R = right(i)                  // figlio destro di 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
  tmp = A[i]                  // scambiamo "i" e "largest"
  A[i] = A[largest]
  A[largest] = tmp
  max-heapify(A, largest) // chiamata ricorsiva sul sottoalbero con radice<max
```

Nota. La prima volta che viene chiamata la procedura sarà per applicarla alla radice: max-heapify (A,0).

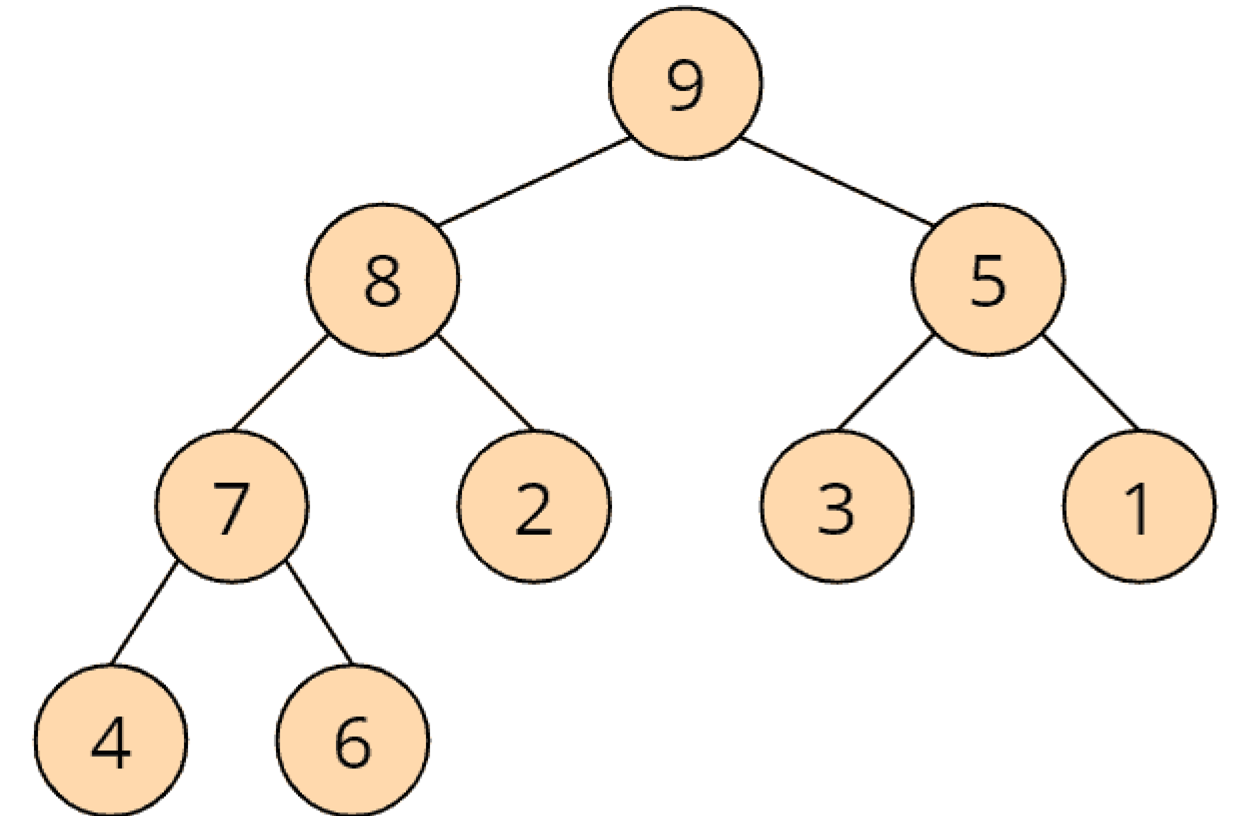
MAX-HEAPIFY: COMPLESSITÀ

- Le operazioni non ricorsive che facciamo ad ogni passo richiedono un tempo costante (confronti e scambi): tempo $\Theta(1)$
- Ad ogni chiamata ricorsiva si confrontano 3 nodi (genitore e 2 figli) ed eventualmente si fa uno scambio. Ad ogni chiamata ricorsiva si scende, al più, di un livello.
- L'albero binario ha al più $\log n$ livelli.
- Quindi la complessità è: $\boldsymbol{T(n) = O(logn)}$

HEAPSORT: FASE 2

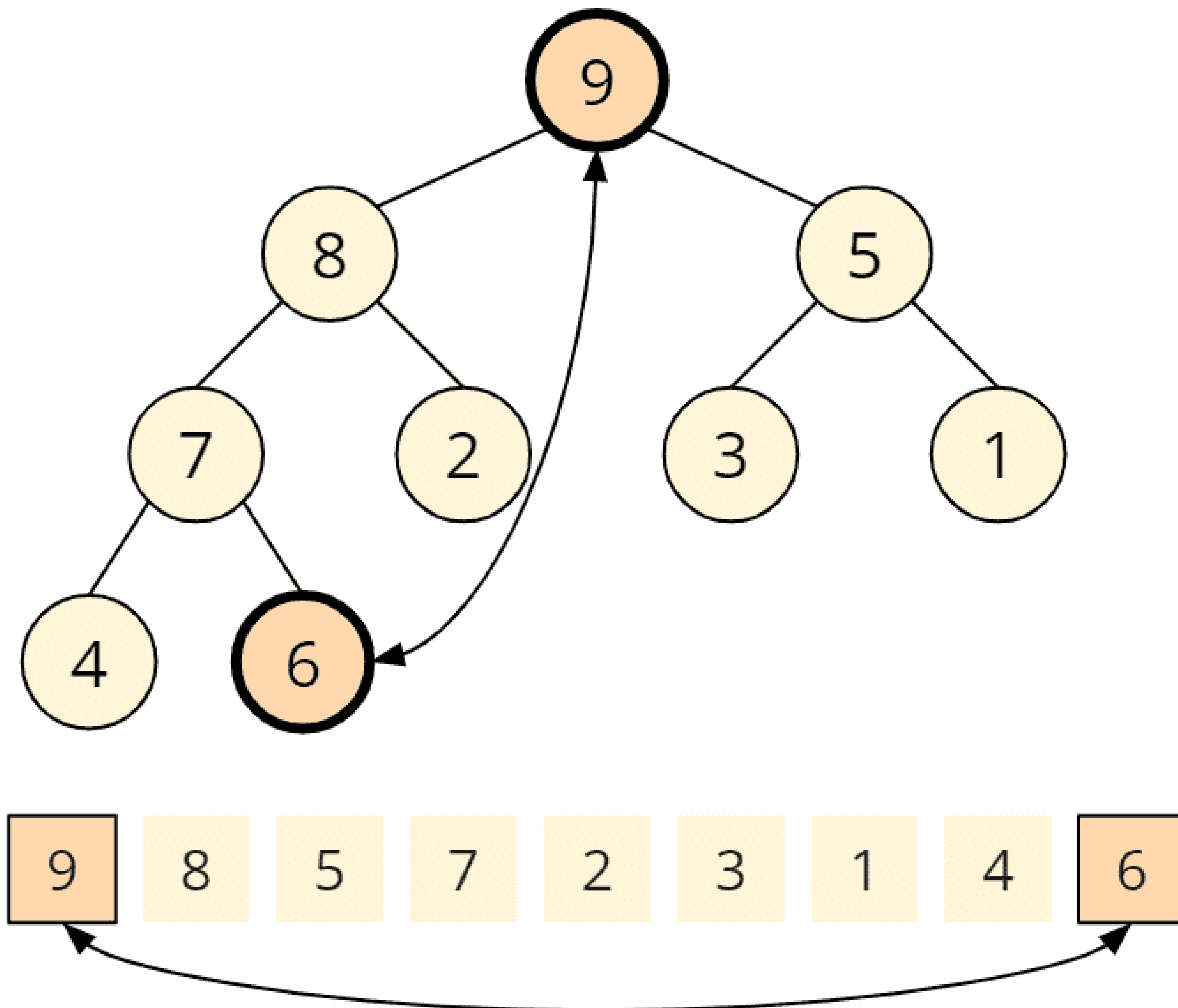
2. l'elemento più grande (cioè quello alla radice dell'albero) viene rimosso;

Sappiamo che la radice è il massimo tra tutti, quindi lo possiamo portare in fondo.

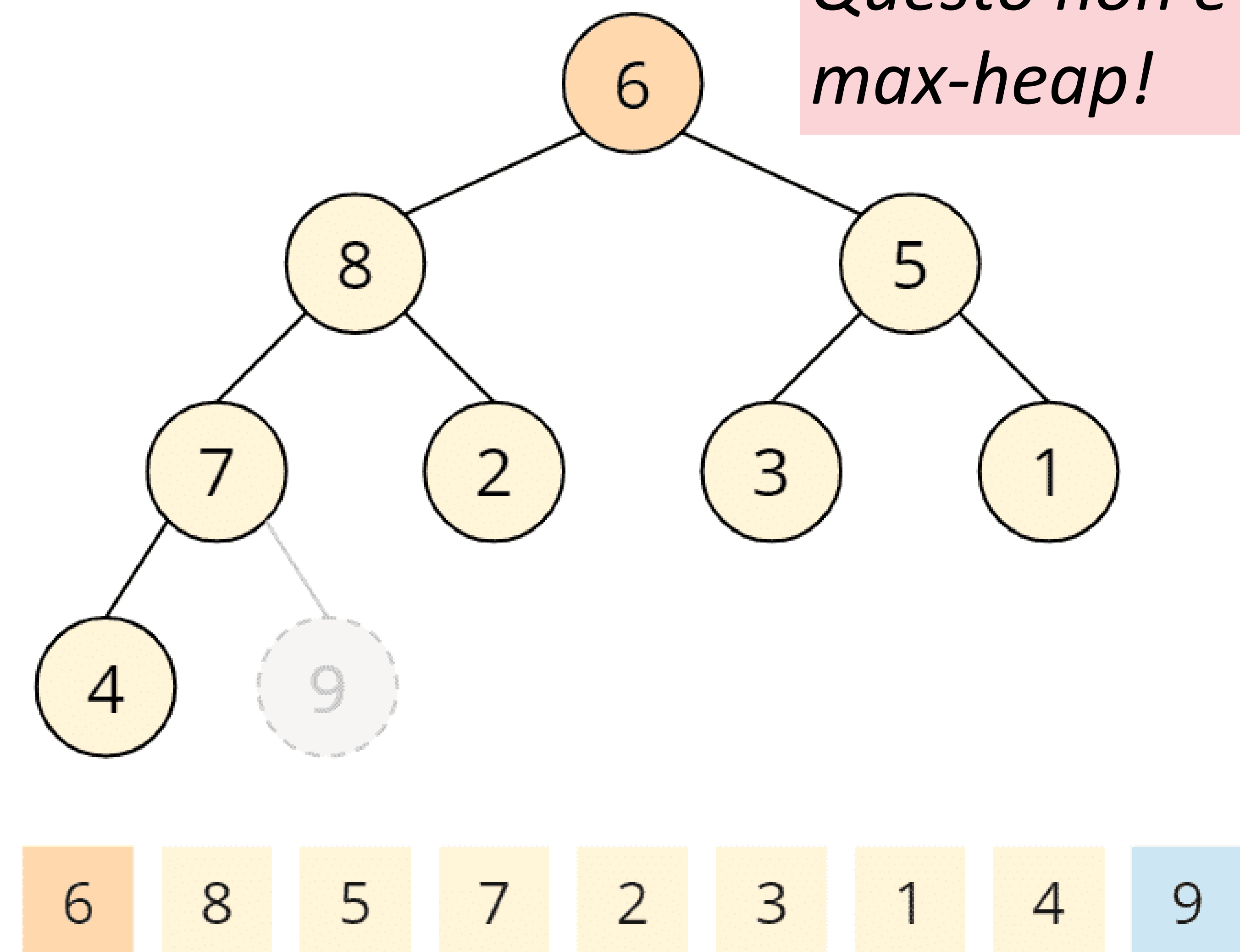


HEAPSORT: FASE 2

2. l'elemento più grande (cioè quello alla radice dell'albero) viene rimosso *e sostituito con l'ultima foglia*;

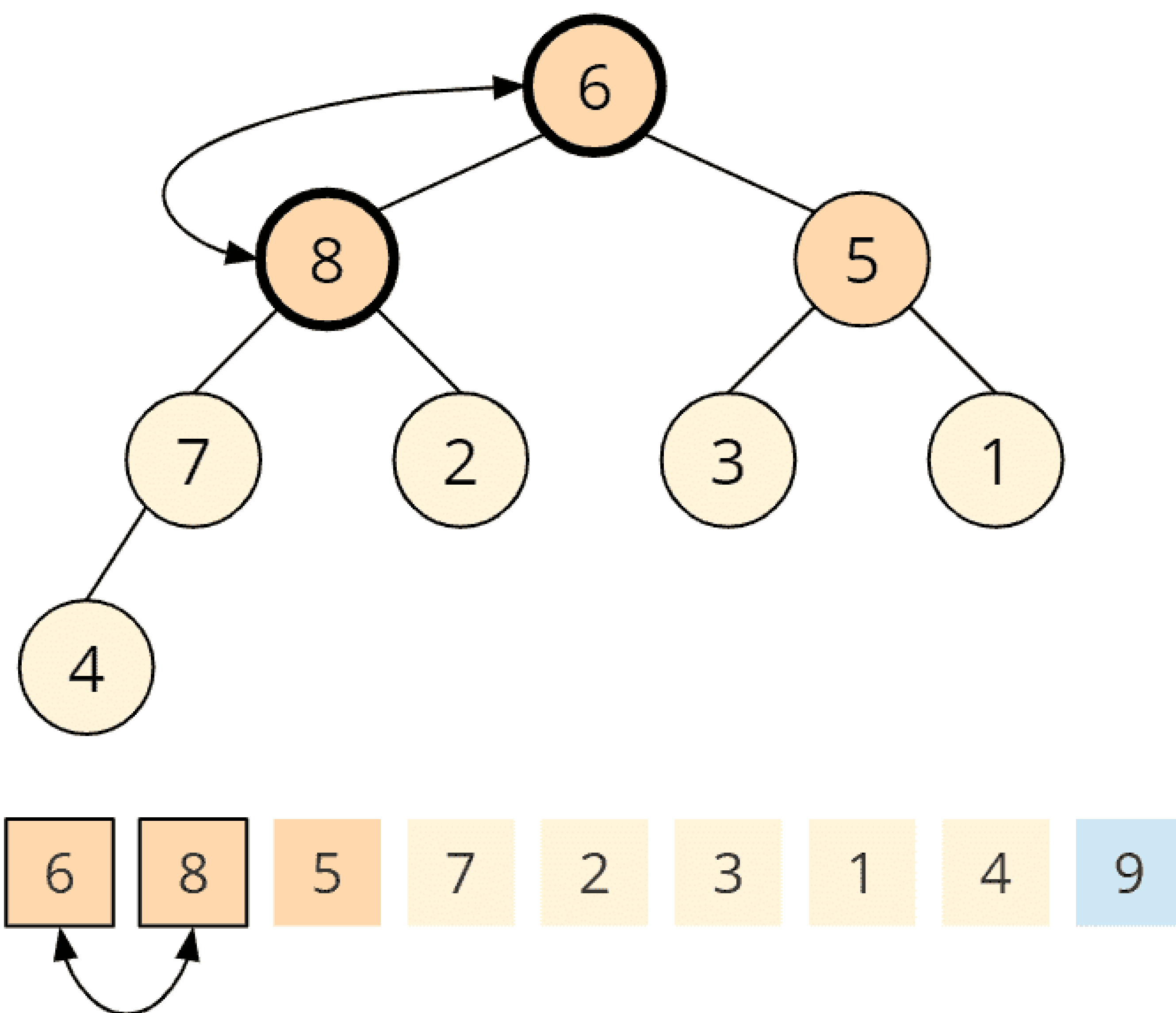


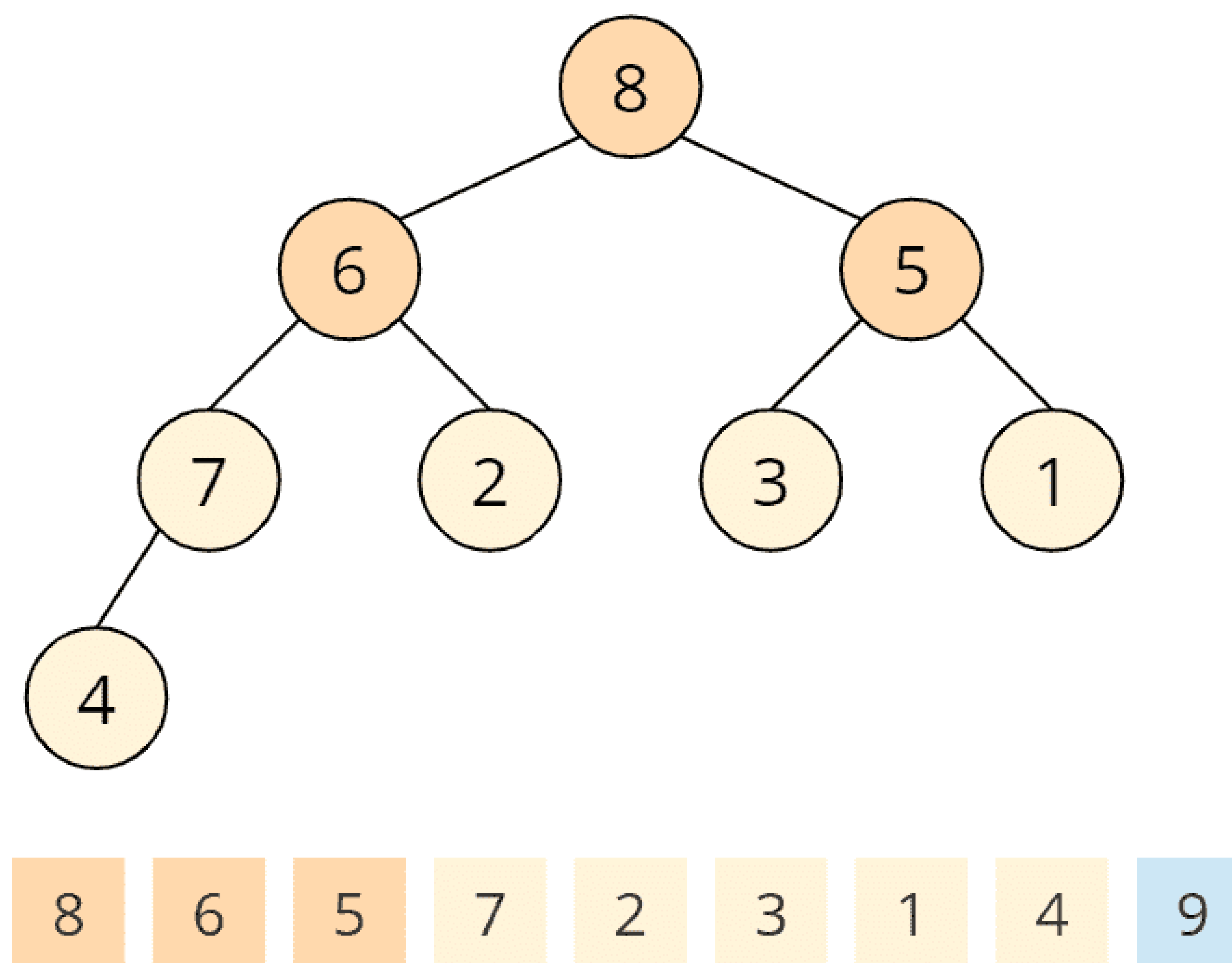
Questo non è più un max-heap!



HEAPSORT: FASE 3

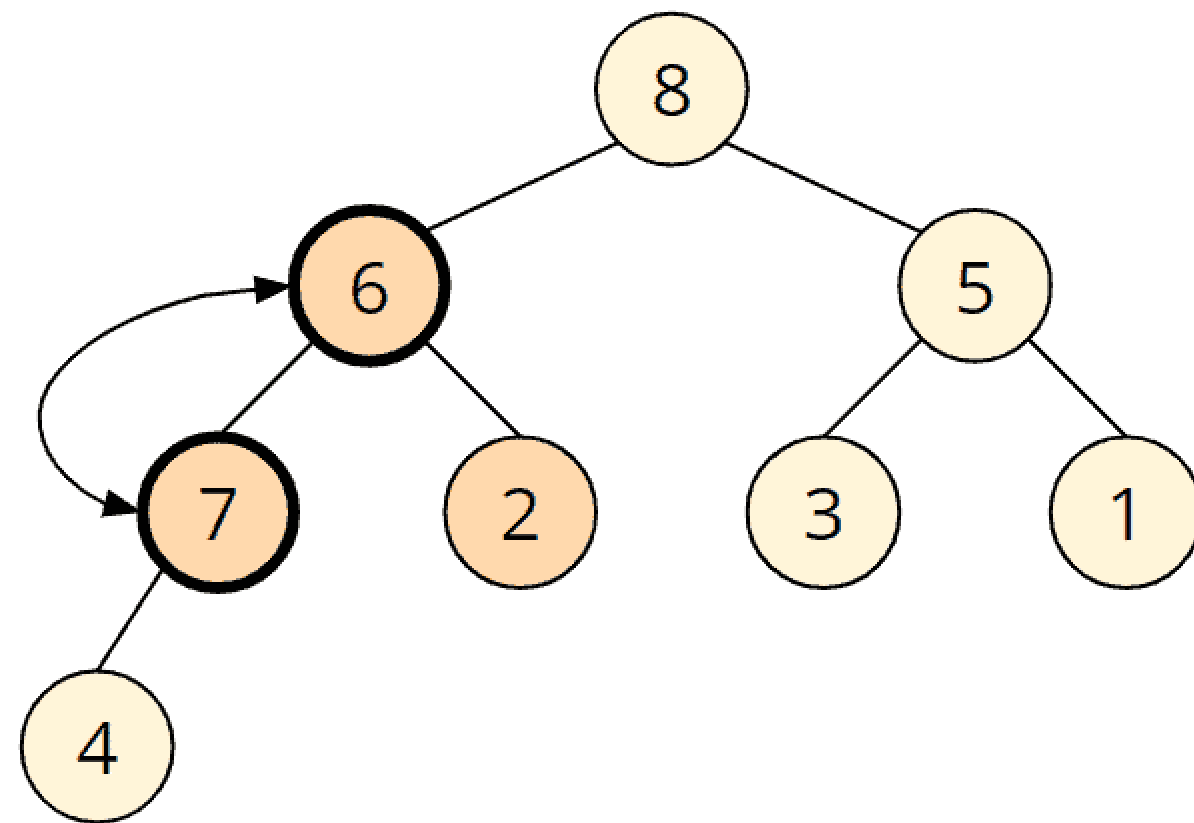
3. si aggiorna la struttura rimanente in modo che garantisca ancora le proprietà di un **max-heap (di nuovo usiamo procedura «heapify»)**

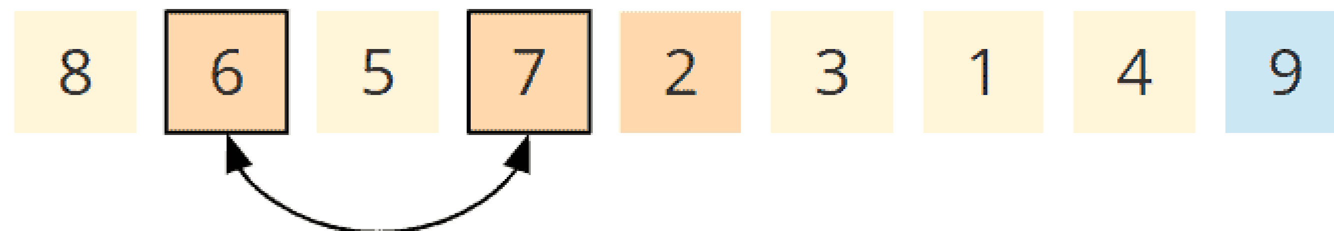


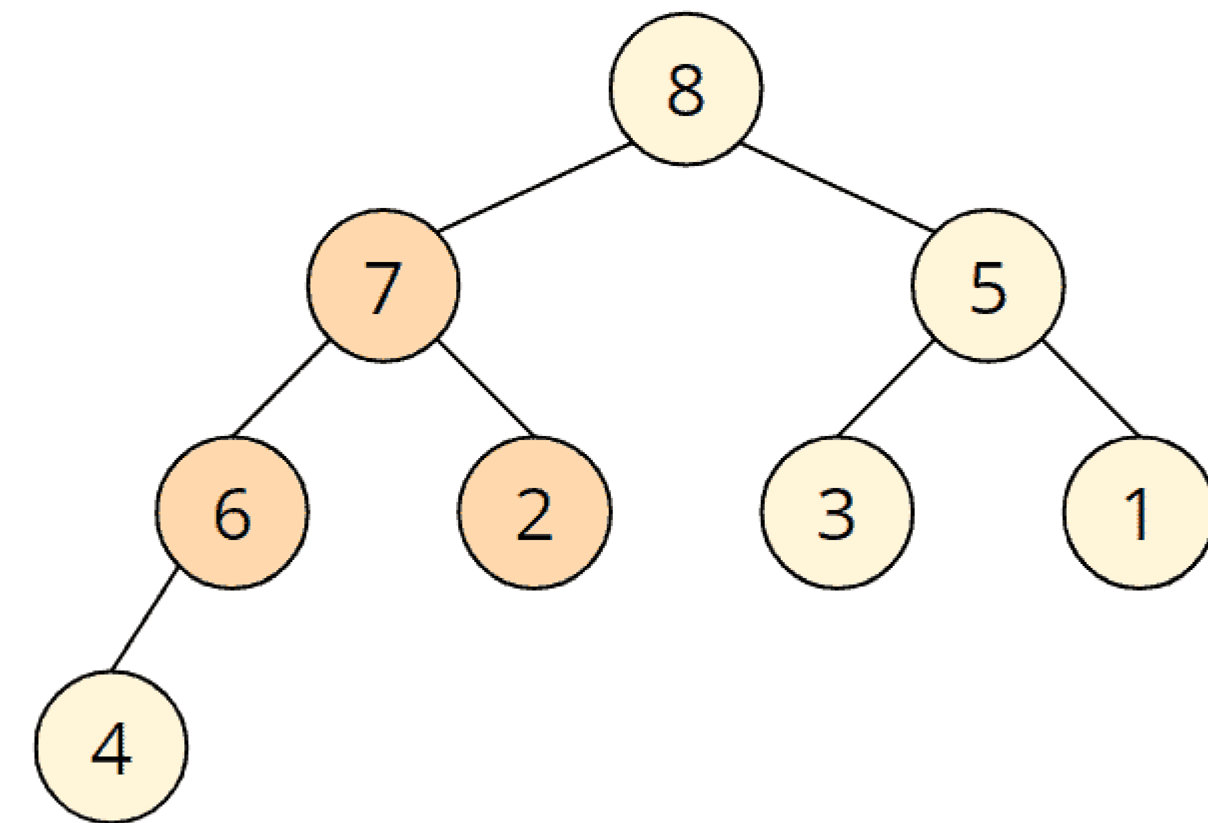


HEAPSORT: FASE 3

3. si aggiorna la struttura rimanente in modo che garantisca ancora le proprietà di un **max-heap (di nuovo usiamo procedura «heapify»)**

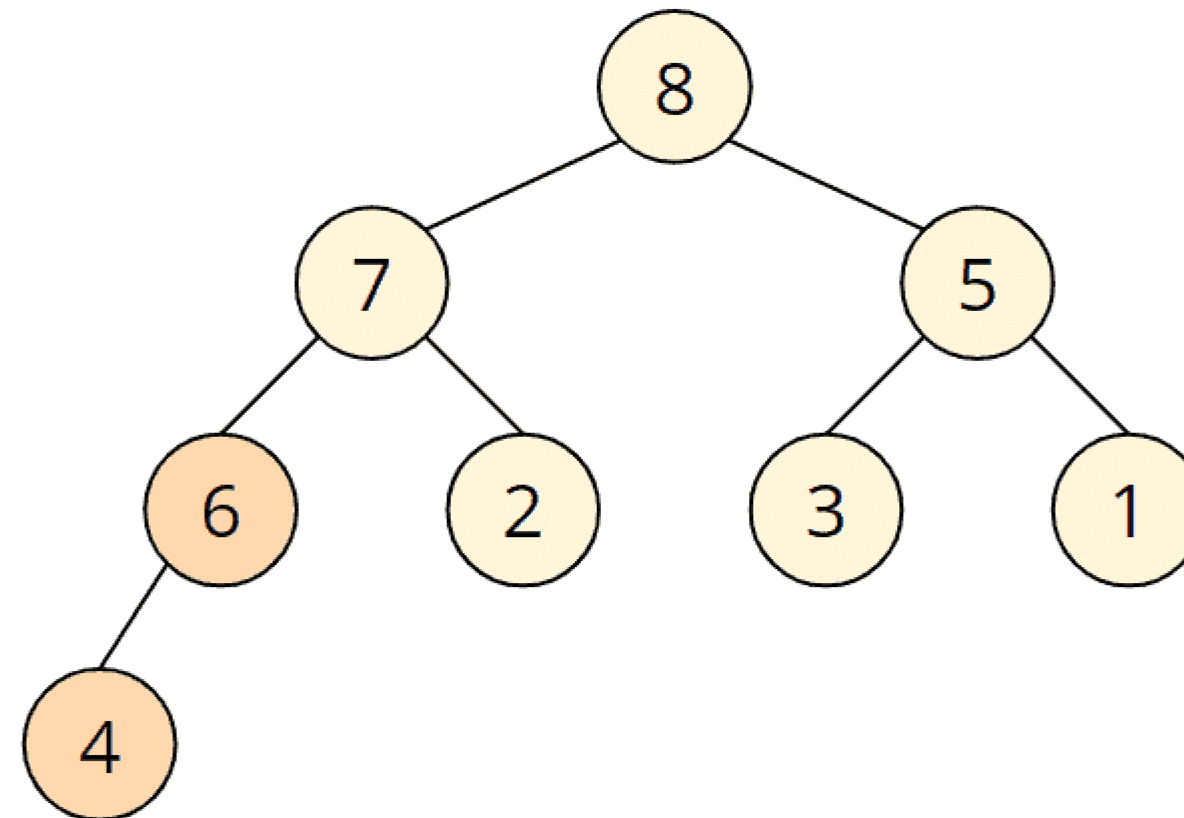






HEAPSORT: FASE 3

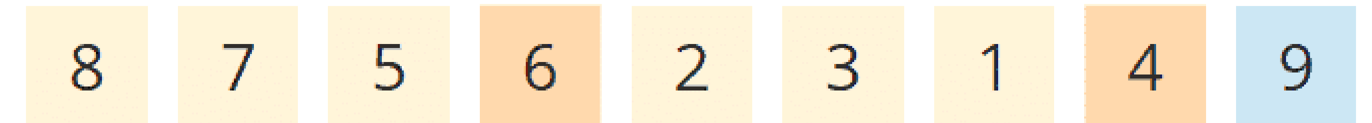
3. si aggiorna la struttura rimanente in modo che garantisca ancora le proprietà di un **max-heap (di nuovo usiamo procedura «heapify»)**



Abbiamo un nuovo max-heap!

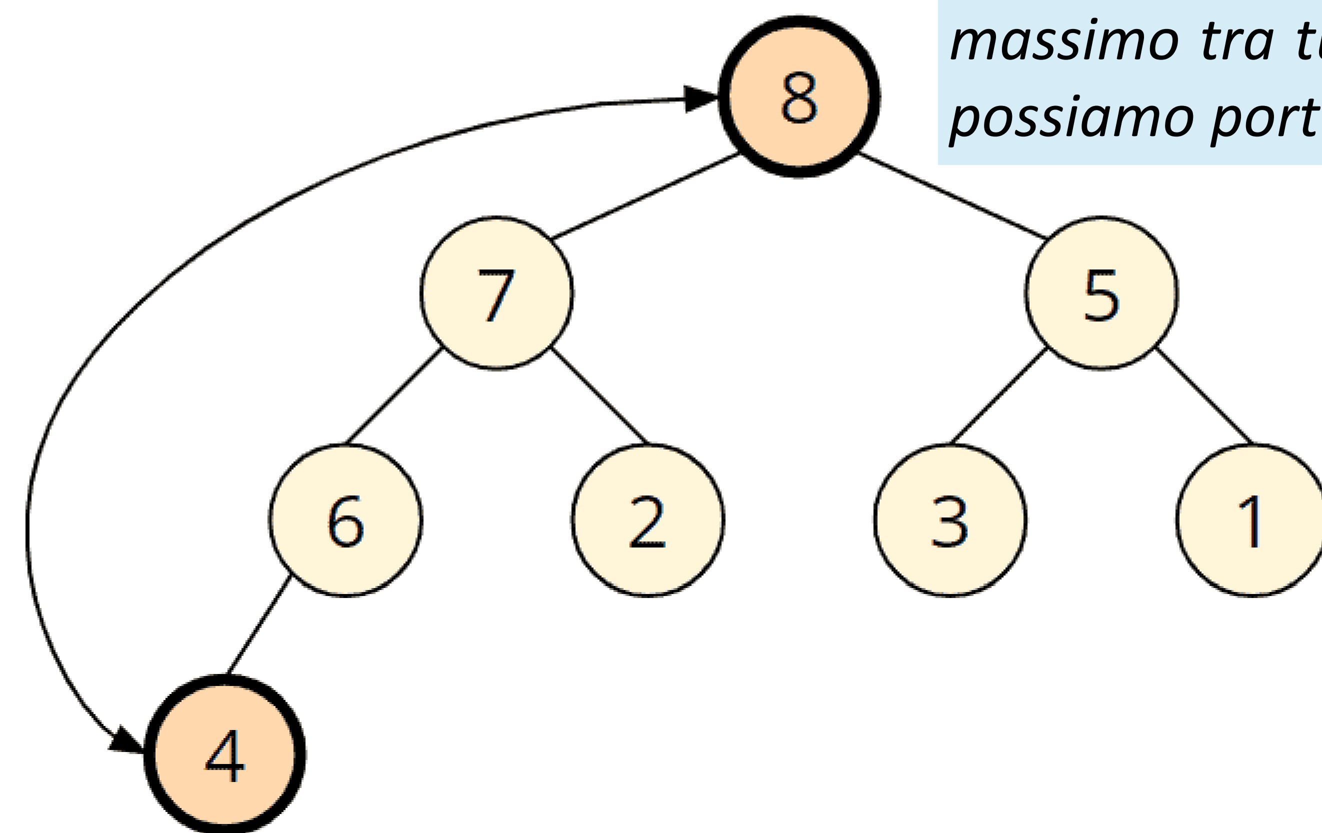
Ripetiamo di nuovo il passo 2 e 3 fino a che l'array non è ordinato.

HEAPSORT: FASE 2



2. l'elemento più grande (cioè quello alla radice dell'albero) viene rimosso;

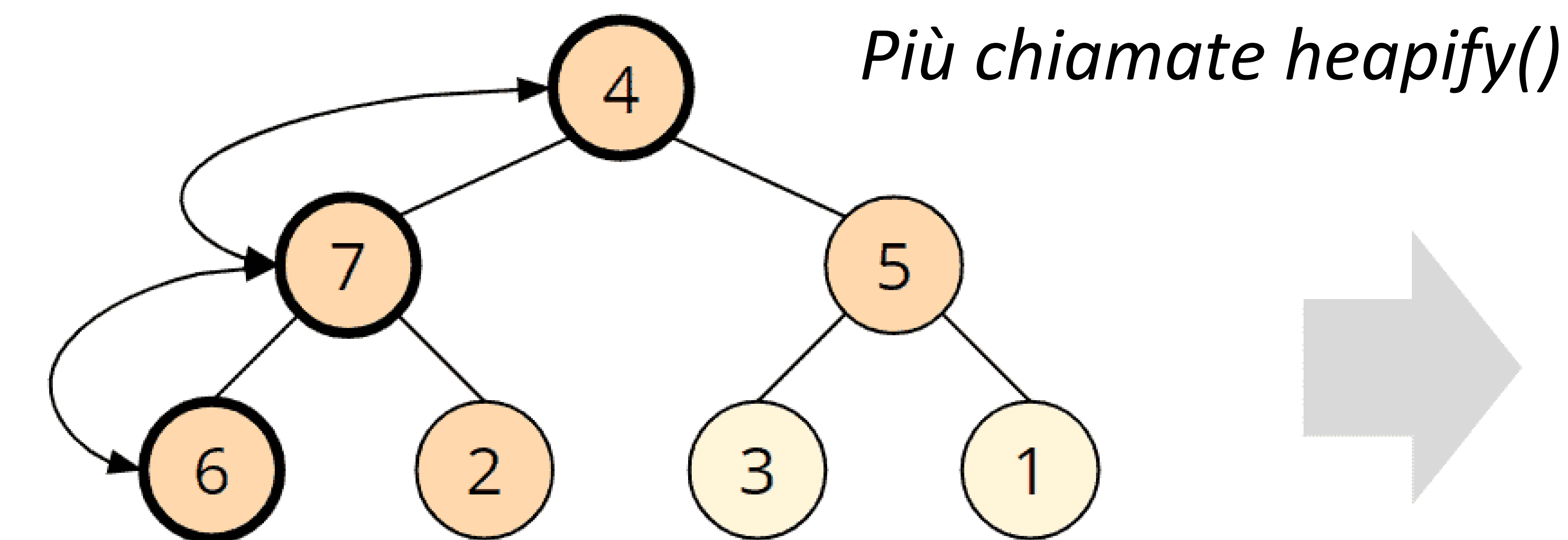
Sappiamo che la radice è il massimo tra tutti, quindi lo possiamo portare in fondo.



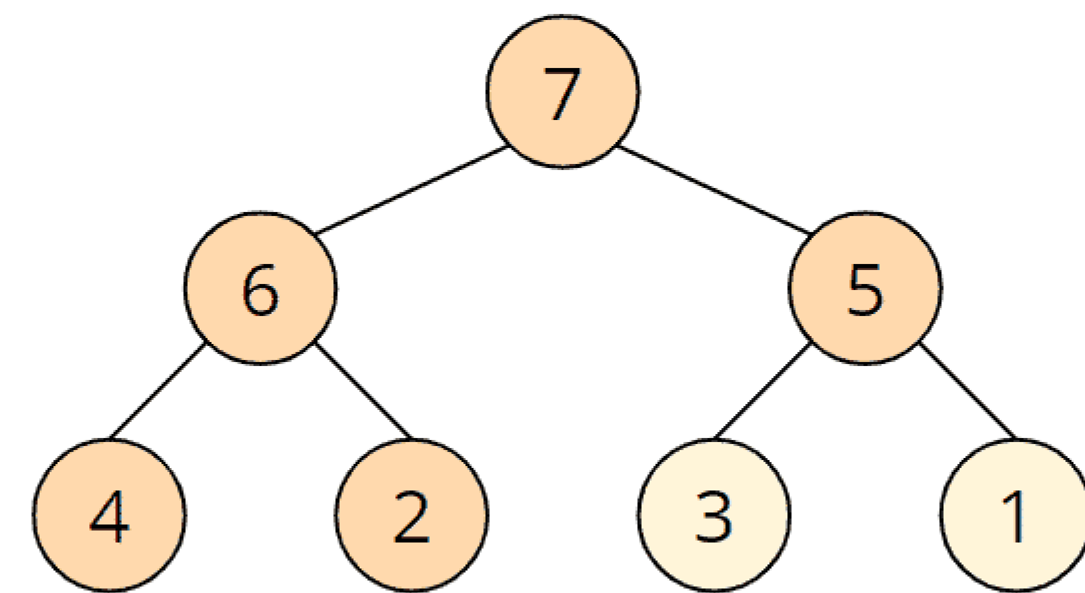
Gli ultimi due elementi sono ordinati!

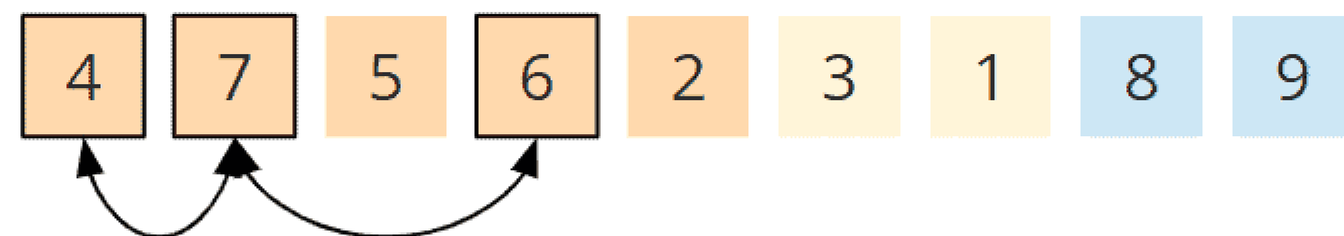
HEAPSORT: FASE 3

3. si aggiorna la struttura rimanente in modo che garantisca ancora le proprietà di un **max-heap (di nuovo usiamo procedura «heapify»)**



Più chiamate heapify()

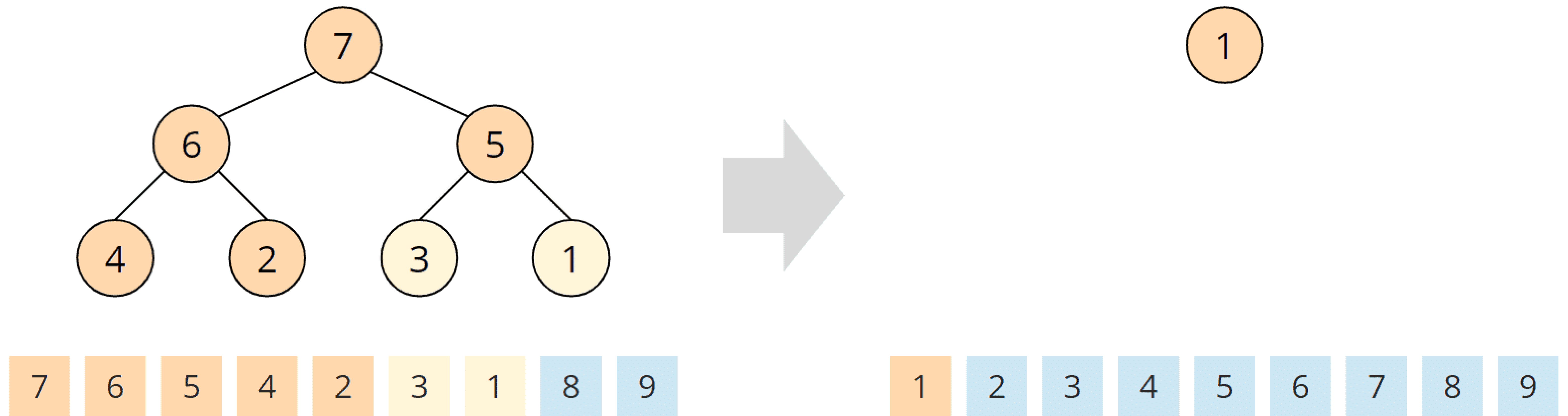




Abbiamo un nuovo max-heap!

HEAPSORT: FINE

Ripetiamo questi step fino a che non c'è un unico elemento ancora da ordinare: essendo un solo elemento è, per forza, ordinato.

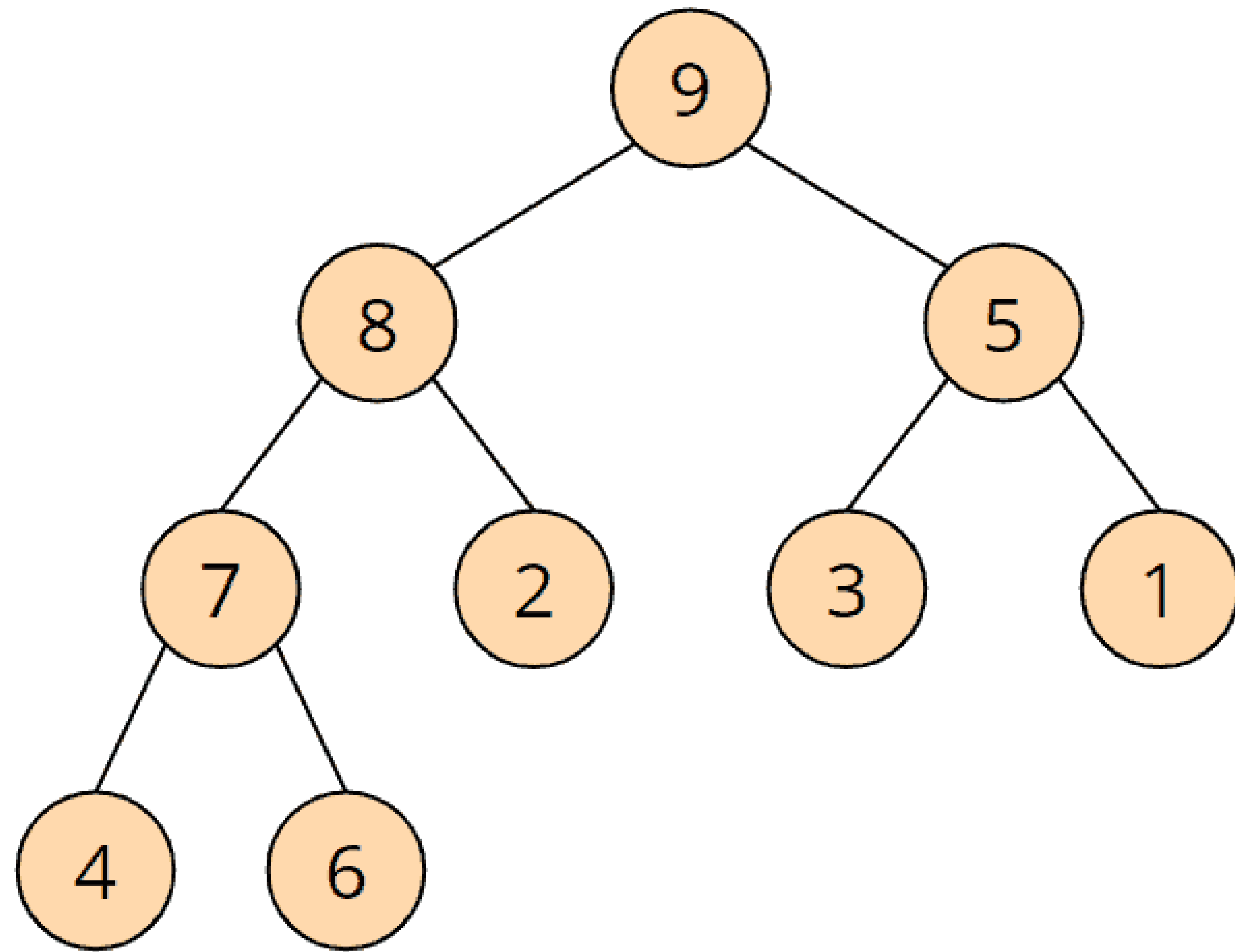


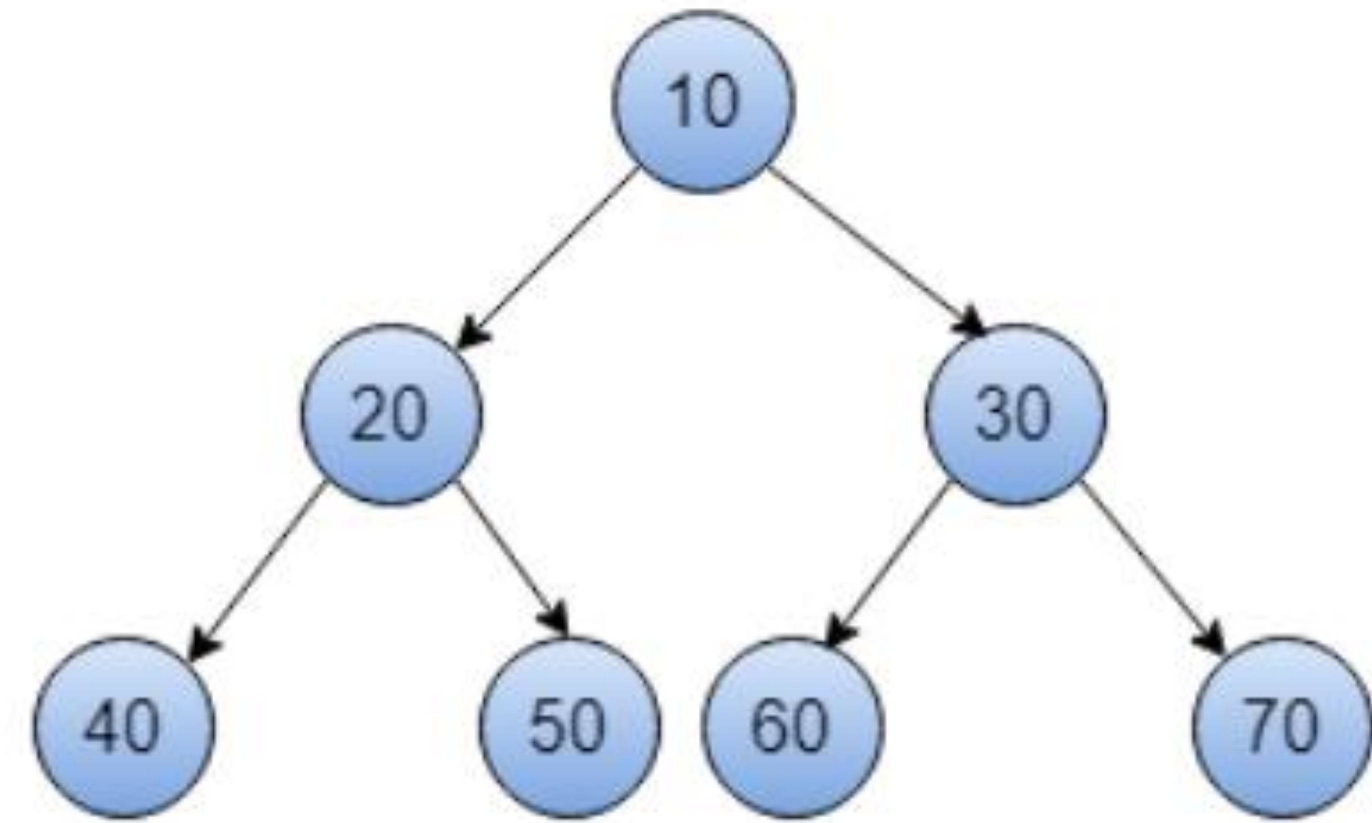
PSEUDOCODICE (HEAPSORT)

```
HEAPSORT(A)                 //input: array A
n = length(A)

// costruzione del max-heap
heap-size(A) = n              // diminuisce fino a 1
for i = floor(n/2) - 1 down to 0 // scorro sui genitori
     MAX-HEAPIFY(A, i)        // O(n) perchè il numero di chiamate è n/2

// fase di ordinamento (estrae il max e risistema max-heap)
for i = n - 1 down to 1       // parte dalle foglie
     swap A[0] and A[i]
     heap-size(A) = heap-size(A) - 1
     MAX-HEAPIFY(A, 0)         // O(log n)
```

Complessità finale di heapsort è:

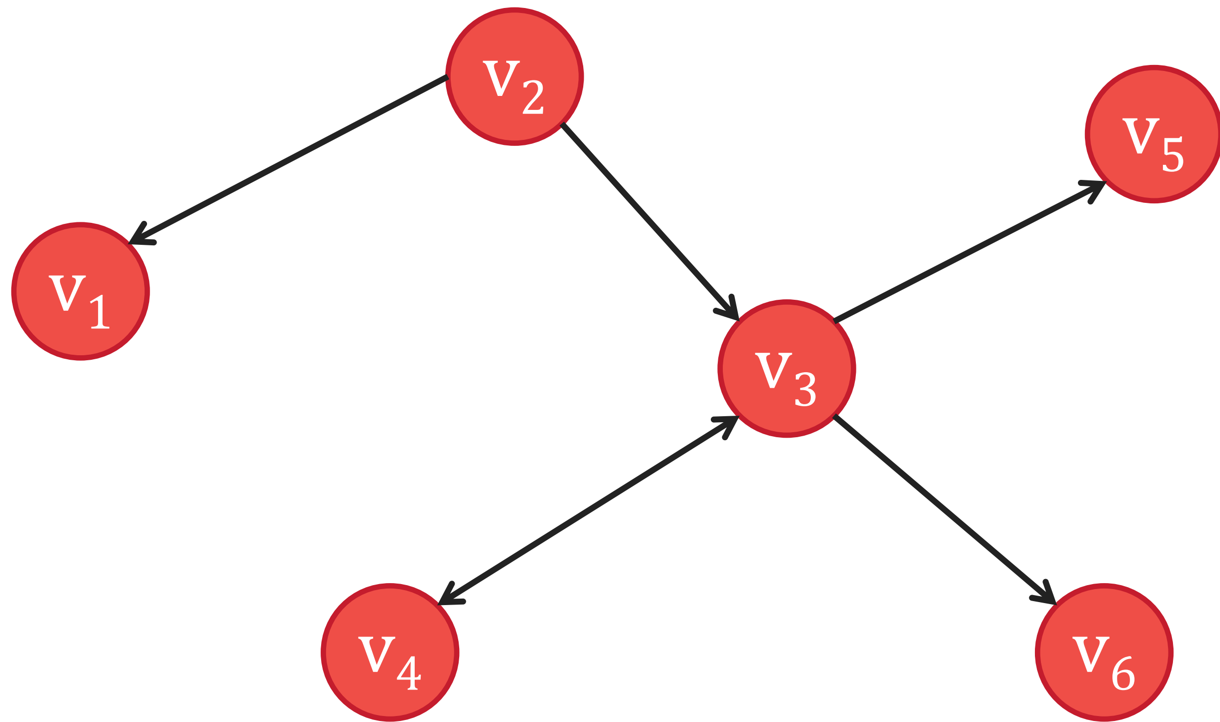
$$T(n) = O(n \log n)$$



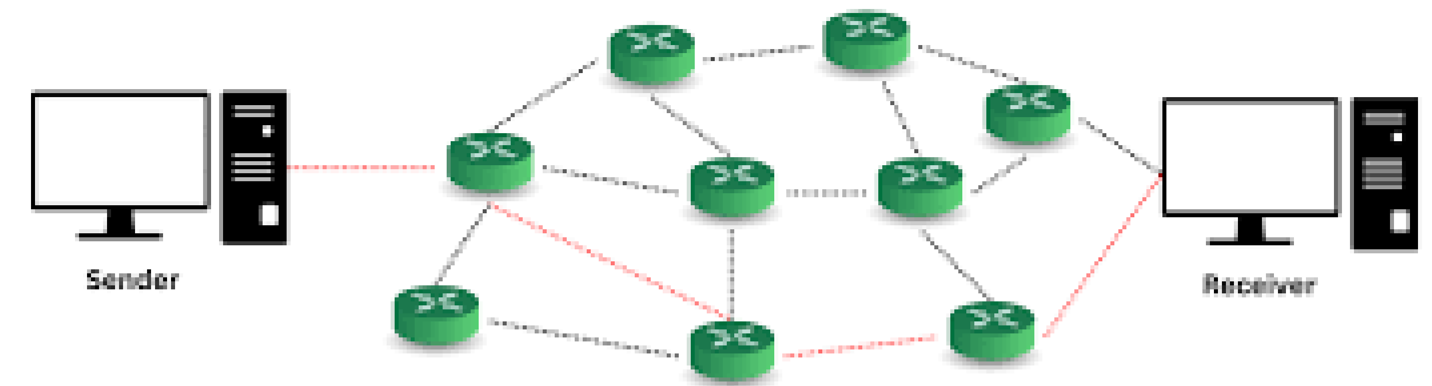


Max-Heap e min-heap sono strutture dati ASTRATTE che possono servire *anche nel contesto di ricerca su grafo!*

Dato un grafo $G = (V, E)$ con n vertici ed m archi, 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).



GPS

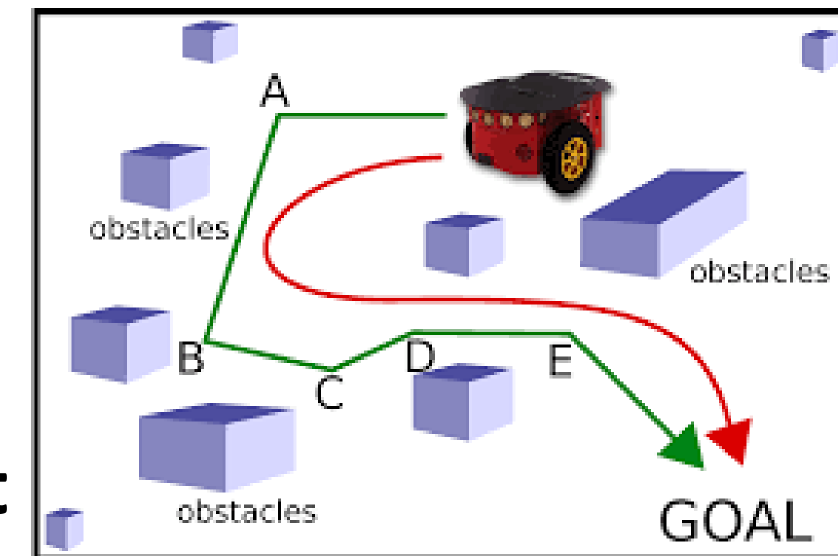


Routing

Il **COSTO o PESO di un percorso** $(v_1, v_2), (v_2, v_3), \dots, (v_{n-1}, v_n)$ è la somma dei pesi di tutti gli archi: $\sum_{i=1}^{n} w_{v_i,v_j}$

Assumiamo di volere **solo percorsi semplici**, ovvero privi di cicli (ogni nodo visitato al più una volta in ogni percorso)

Navigazione robot



Sorgente/Destinazione	1 nodo	Tutti i nodi
1 nodo	singola coppia	singola sorgente
Tutti i nodi	singola destinazione	tutte le coppie

Il **problema del cammino minimo da una sorgente ad una destinazione**: si cerca il cammino minimo da un vertice sorgente *v* ad una destinazione *d*.

Il **problema del cammino minimo da una sorgente**: si cercano i cammini minimi da un vertice sorgente *v* a tutti gli altri vertici del grafo.

Il **problema del cammino minimo verso una destinazione**: si cercano i cammini minimi da tutti i vertici del *grafo orientato* verso un unico vertice di destinazione *v*. Questo può essere ricondotto al problema del cammino minimo da una sorgente invertendo gli archi del grafo orientato.

Il **problema del cammino minimo tra tutte le coppie di nodi**: si cercano i cammini minimi tra ogni coppia di vertici *v, v'* nel grafo.

- 1956 — **Dijkstra**
- 1957 — **Bellman-Ford**
- 1965 — **(+) BFS (per grafi non pesati)**
- 1977 — **A***
- 1978 — **Johnson's Algorithm**
- 1983 — **Dial's Algorithm**
- 1999 — **Thorup's Algorithm**
- Anni 2000+ — **SPFA, algoritmi euristici e metaeuristici**

Gli algoritmi classici (**Dijkstra**, Bellman-Ford-*Moore*, Floyd-Warshall) risalgono agli **anni '50-'60**.

Negli anni **'70–'90** si lavora su efficienza e casi speciali (grafi sparsi, pesi negativi).

Dagli **anni 2000 in poi** emergono algoritmi **euristici** e **metaeuristici** per problemi realistici più complessi e vincolati (AI, robotica, logistica).

Edsger W. Dijkstra (1930 – 2002) fisico olandese

Lester R. Ford (1886 – 1967) Matematico americato

Richard E. Bellman (1920 – 1984) Matematico statunitense

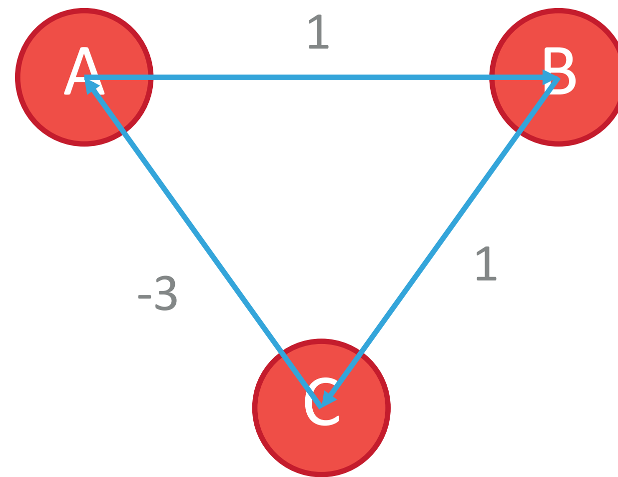
Edward F. Moore (1925–2003) Matematico e informatico

NON E' Gordon Moore (1929–2023), co-fondatore di Intel, noto per la celebre Legge di Moore che prevede il raddoppio del numero di transistor nei circuiti integrati ogni due anni

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 se 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

- Supponiamo di avere una stima del peso del **percorso da $\boldsymbol{s}$ ad un nodo $\boldsymbol{u}$**, indicata con $\boldsymbol{distanza}[\boldsymbol{u}]$, **e ad un nodo $\boldsymbol{v}$,** indicata con $\boldsymbol{distanza}[\boldsymbol{v}]$ e che **esista l'arco** $(\boldsymbol{u}, \boldsymbol{v})$ 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, ovvero se conviene arrivare a v passando per u.

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:

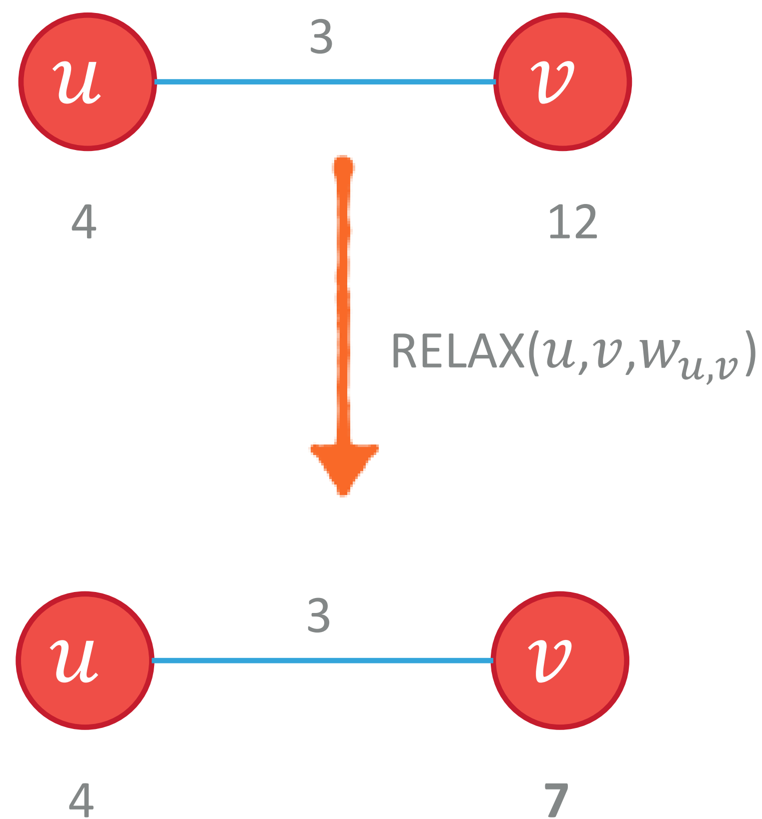
$$\text{distanza}[v] = \text{distanza}[u] + w_{u,v}$$

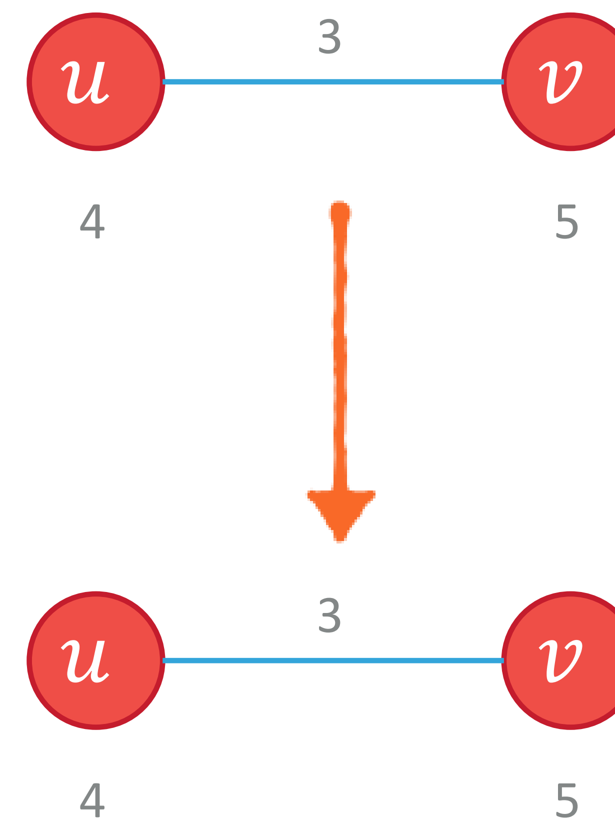
- Altrimenti non la modifichiamo
- Indichiamo questa operazione come **RELAX($\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w_{u,v}}$)**

OPERAZIONE DI RILASSAMENTO/RELAX DEGLI ARCHI

Costi attuali dalla sorgente s:

Costi aggiornati:





if distanza[u] + w < distanza[v]
distanza[v] = distanza[u] + w
predecessore[v] = u

Mantengo come prima.

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

SE DOPO $|V| - 1$ ITERAZIONI FOSSE ANCORA POSSIBILE AGGIORNARE LE DISTANZE SIGNIFICHEREBBE CHE IL GRAFO CONTIENE UN CICLO NEGATIVO: POSSIAMO USARE L'ALGORITMO PER IDENTIFICARE QUESTO TIPO DI GRAFI

PSEUDOCODICE

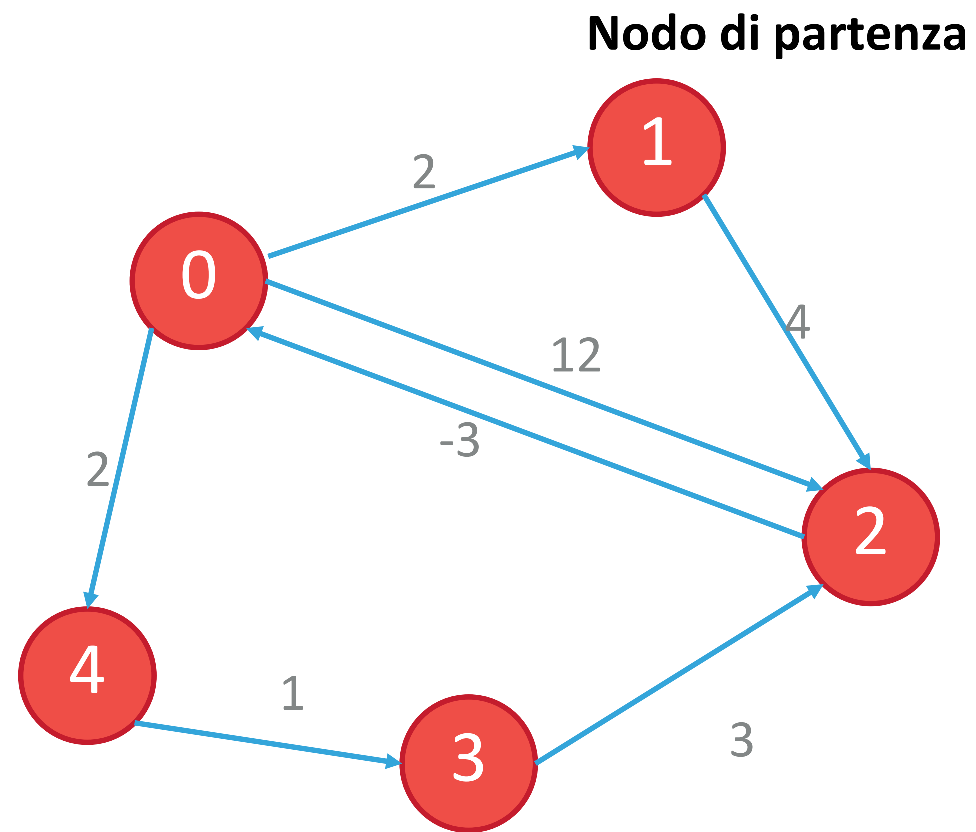
```
Parametri: grafo G, nodo sorgente s
# inizialmente impostiamo distanza e predecessore di tutti i nodi
for all v ∈ V:
    distanza[v] = +∞
    predecessore[v] = None
# e poi impostiamo il nodo s come sorgente a distanza 0
distanza[s] = 0
for i in range(0, |V| − 1)
    for all (u,v) ∈ E # effettuiamo il rilassamento di tutti gli archi
        RELAX(u, v, w_{u,v})
```

esplora tutti gli archi (si può decidere un ordine con cui visitarlo ad ogni iterazione da 0 a |V|-1 ma l'ordine non è critico)

A seconda dell'ordine, quello che può cambiare è il predecessore, quindi il cammino minimo specifico ricostruito, se esistono più cammini minimi con lo stesso costo.

ESEMPIO DI ESECUZIONE

Iter #1



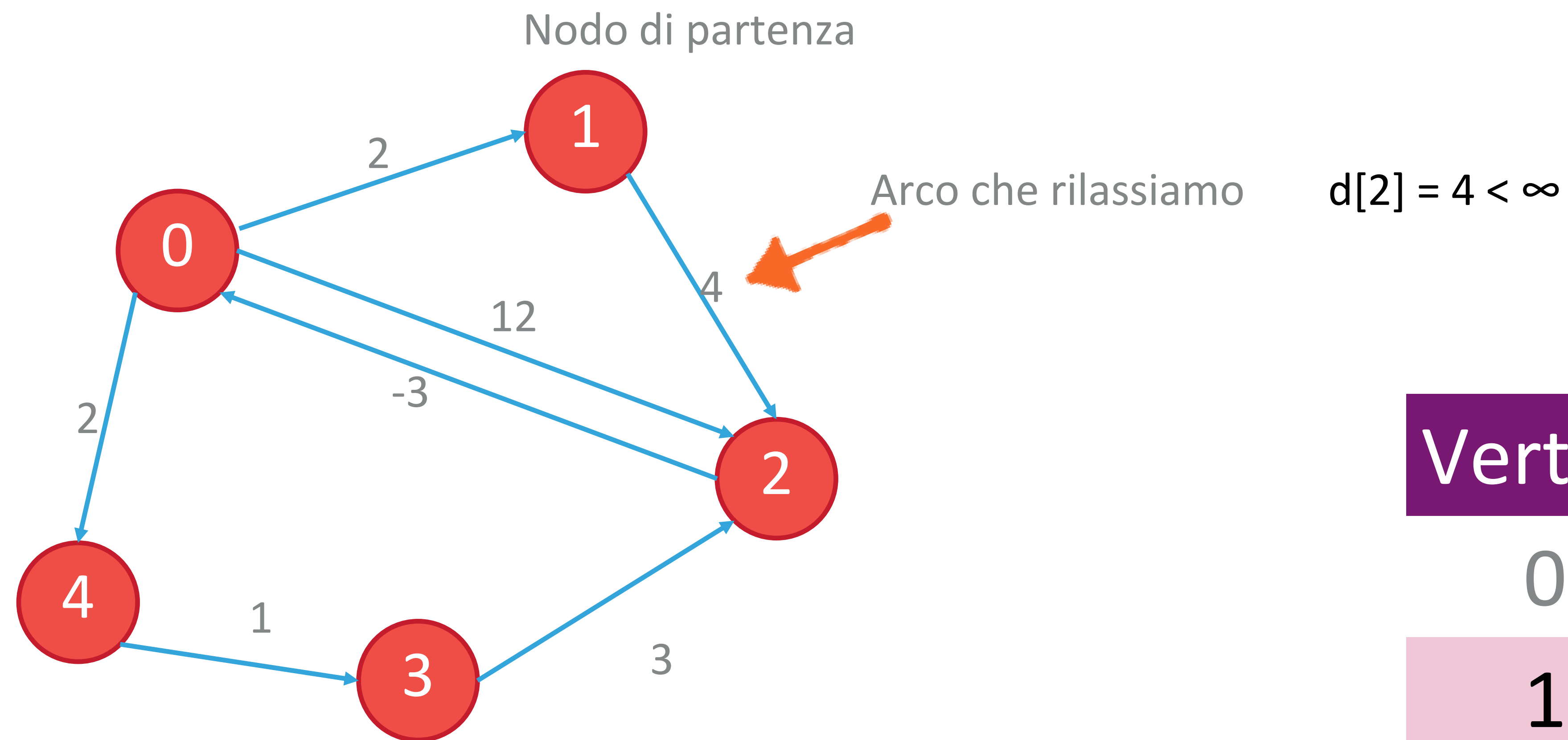
Per l’esempio usiamo un grafo orientato, altrimenti non potremmo mai avere pesi negativi (avremmo un ciclo negativo)

Nota. Scegliamo di processare gli archi con il seguente ordine (casuale o seguendo l’ordine in cui sono memorizzati):
1->2
2->0
0->2
0->1
0->4
4->3
3->2

Vertice	Peso	Predecessore
0	∞	-
1	0	-
2	∞	-
3	∞	-
4	∞	-

ESEMPIO DI ESECUZIONE

Iter #1

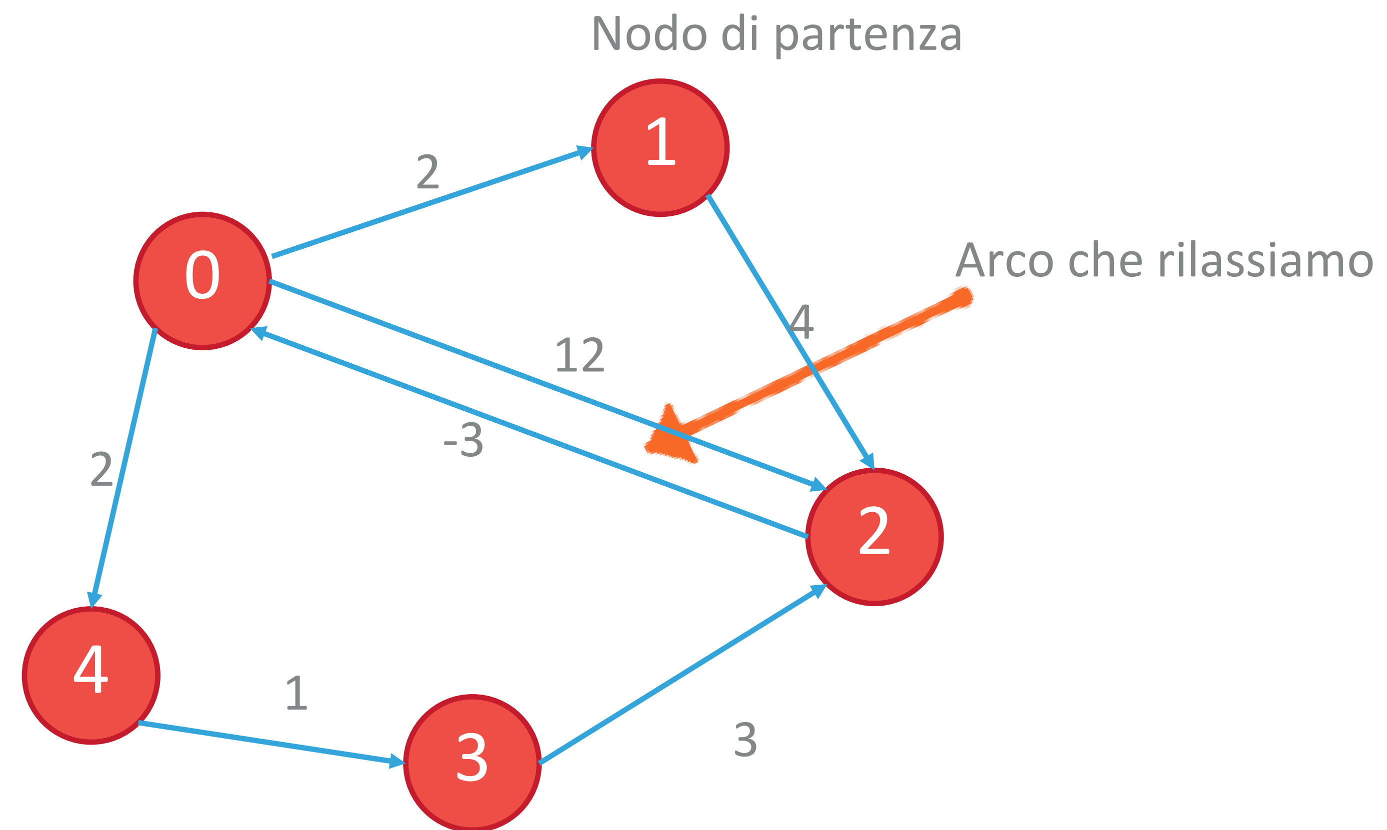


d[2] = 4 < ∞

Vertice	Peso	Predecessore
0	∞	-
1	0	-
2	4	1
3	∞	-
4	∞	-

ESEMPIO DI ESECUZIONE

Iter #1

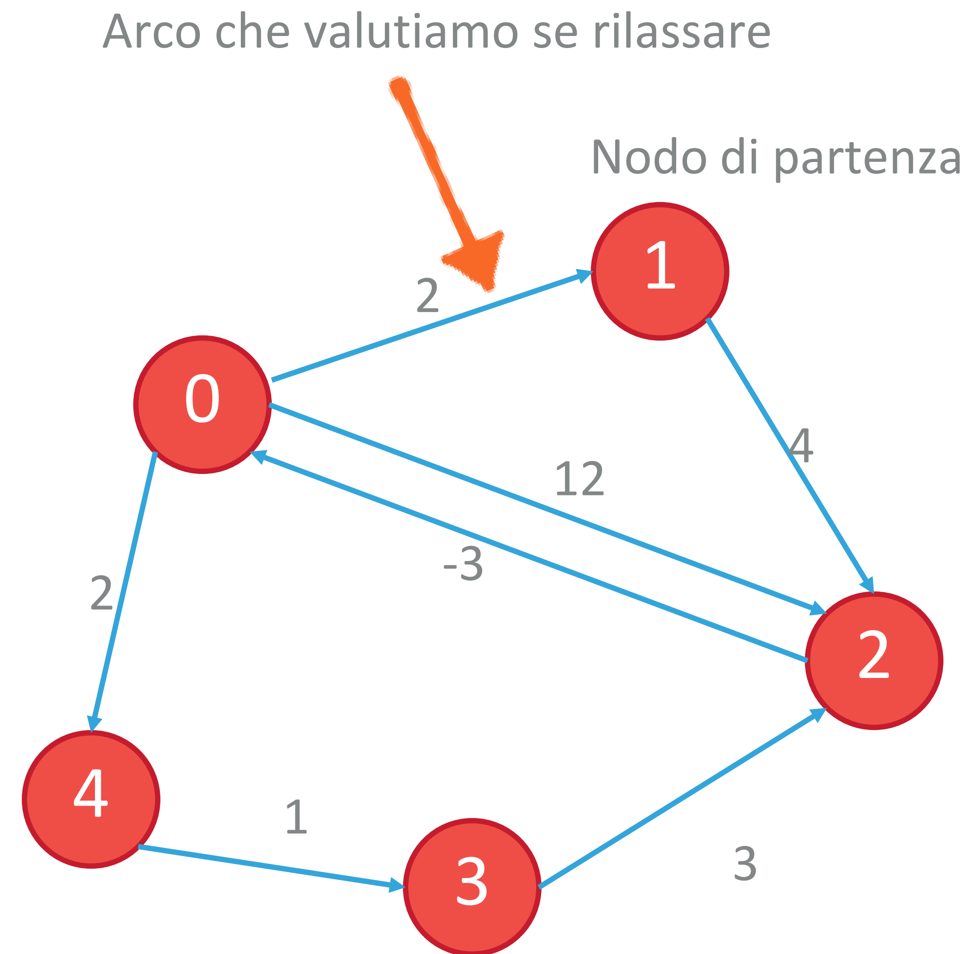


$d[0] = d[2] -3 = 4-3 = 1 < \infty$

Vertice	Peso	Predecessore
0	1	2
1	0	-
2	4	1
3	∞	-
4	∞	-

ESEMPIO DI ESECUZIONE

Iter #1

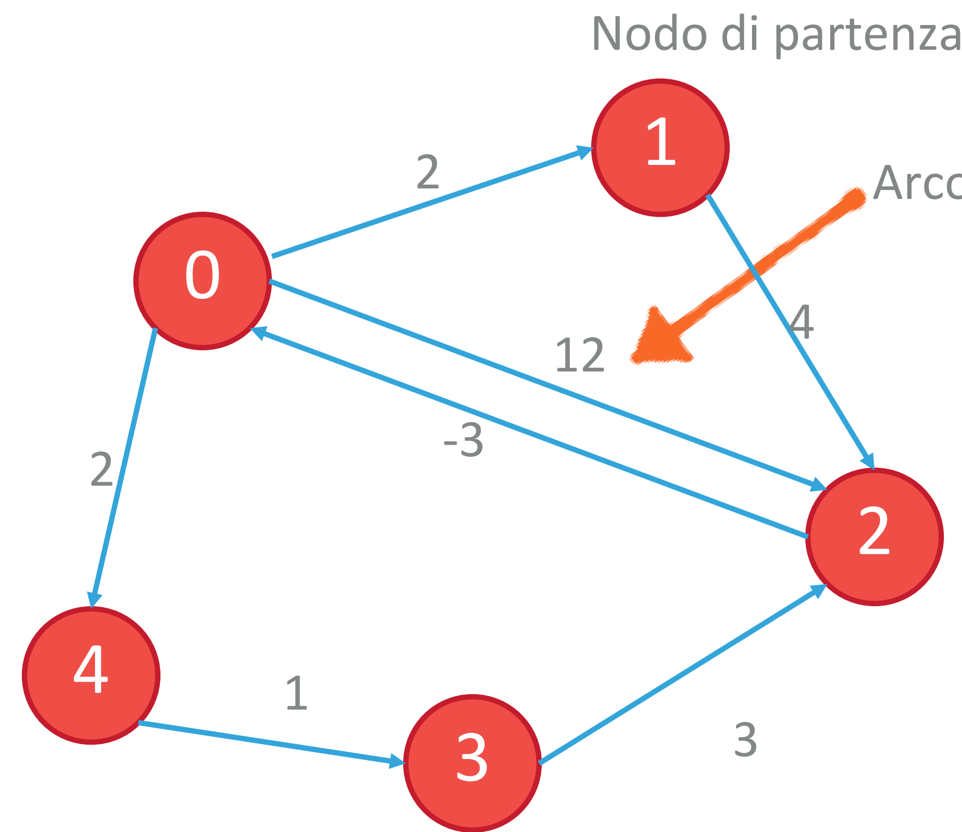


d[1] = d[0] + 2 = 1+3 = 4 > 0

Vertice	Peso	Predecessore
0	1	2
1	0	-
2	4	1
3	∞	-
4	∞	-

ESEMPIO DI ESECUZIONE

Iter #1

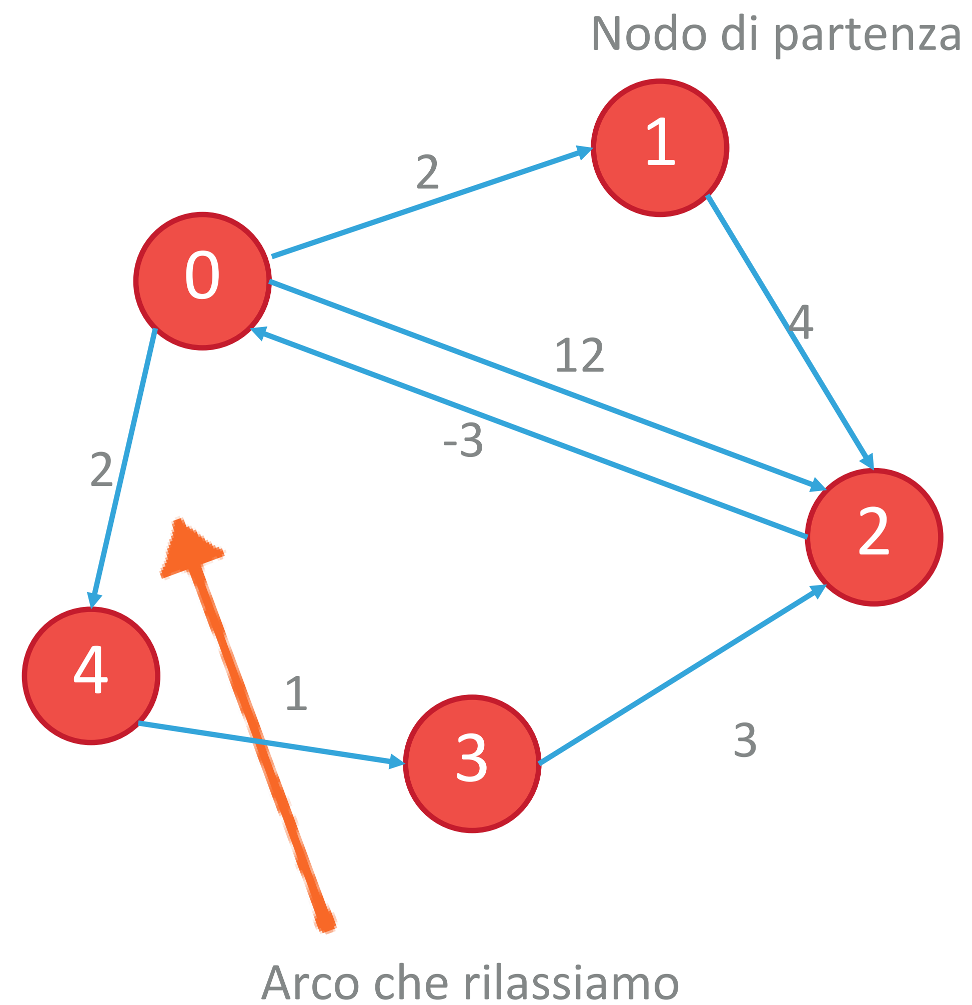


d[2] = d[0] + 12 = 1+12 = 13 > 4

Vertice	Peso	Predecessore
0	1	2
1	0	-
2	4	1
3	∞	-
4	∞	-

ESEMPIO DI ESECUZIONE

Iter #1

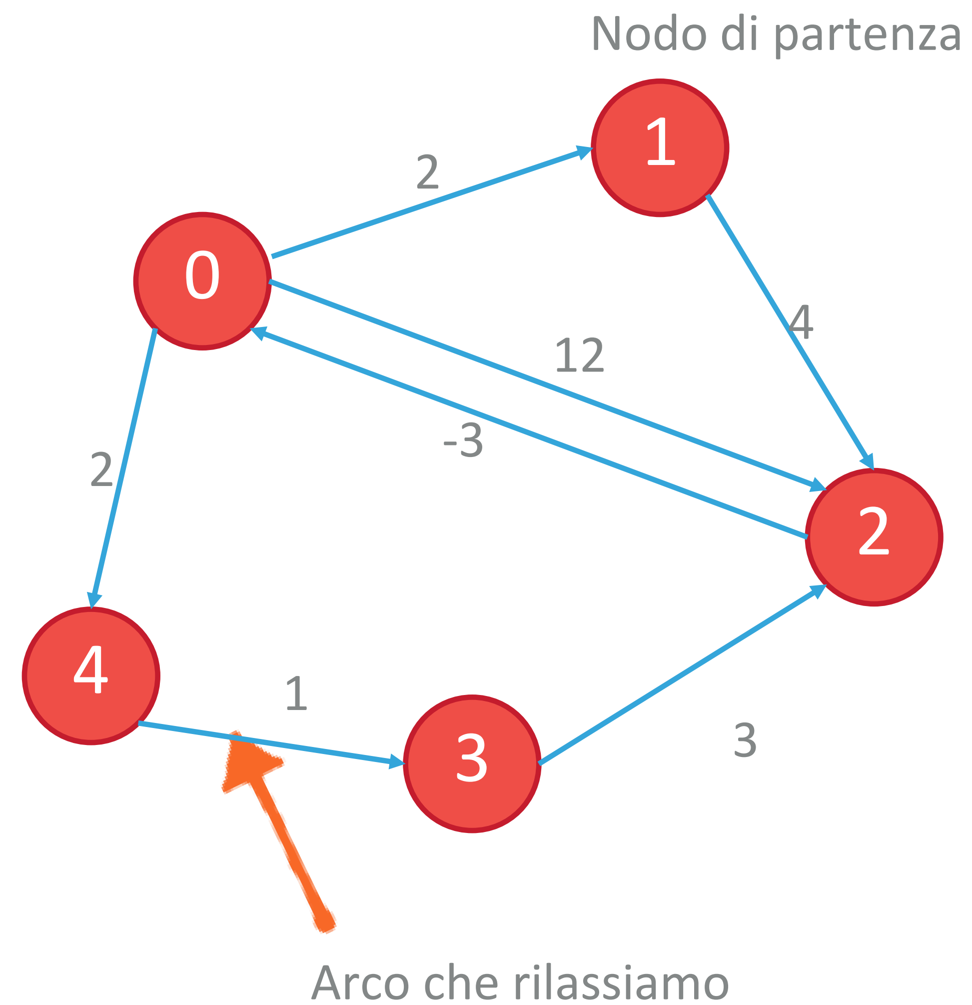


d[4] = d[0] + 2 = 1+2 = 3 < ∞

Vertice	Peso	Predecessore
0	1	2
1	0	-
2	4	1
3	∞	-
4	3	0

ESEMPIO DI ESECUZIONE

Iter #1

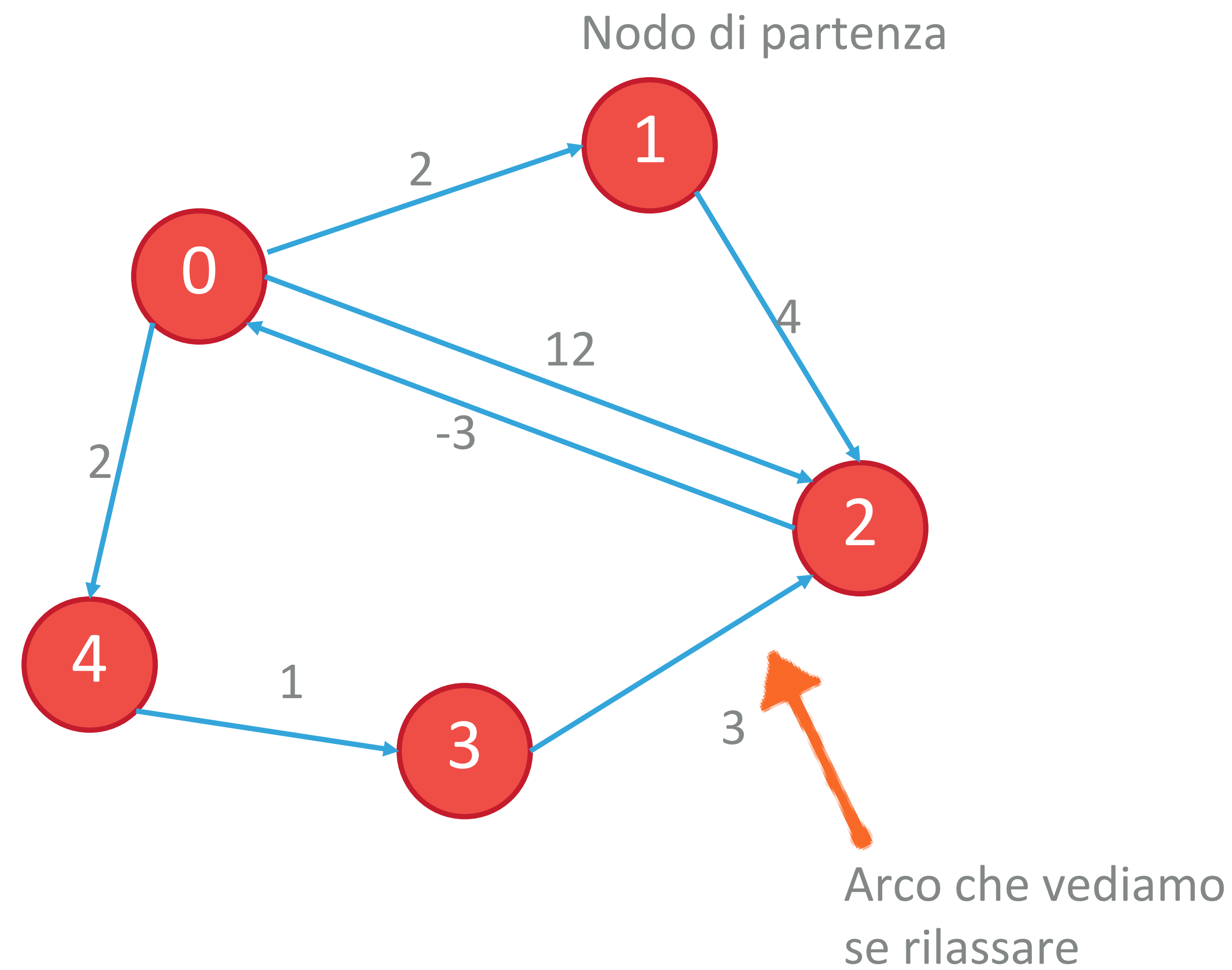


$d[3] = d[4] + 1 = 3+1 = 4 < \infty$

Vertice	Peso	Predecessore
0	1	2
1	0	-
2	4	1
3	4	4
4	3	0

ESEMPIO DI ESECUZIONE

Iter #1



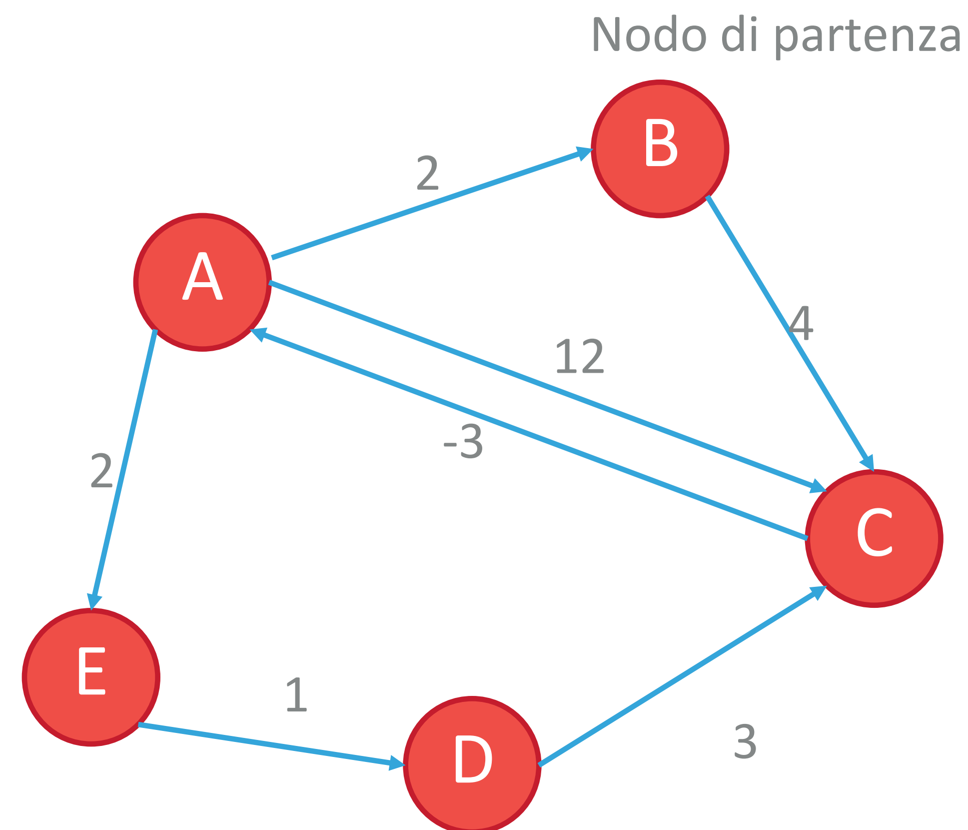
d[2] = d[3] + 3 = 4+3 = 7 > 4

Vertice	Peso	Predecessore
0	1	2
1	0	-
2	4	1
3	4	4
4	3	0

ESEMPIO DI ESECUZIONE

In questo caso già con la conclusione della prima iterazione (iter #1) si ottengono le distanze minime per ogni nodo. Tuttavia, **l'algoritmo continua per altre 3 iterazioni***, senza però trovare alcun peso da aggiornare.

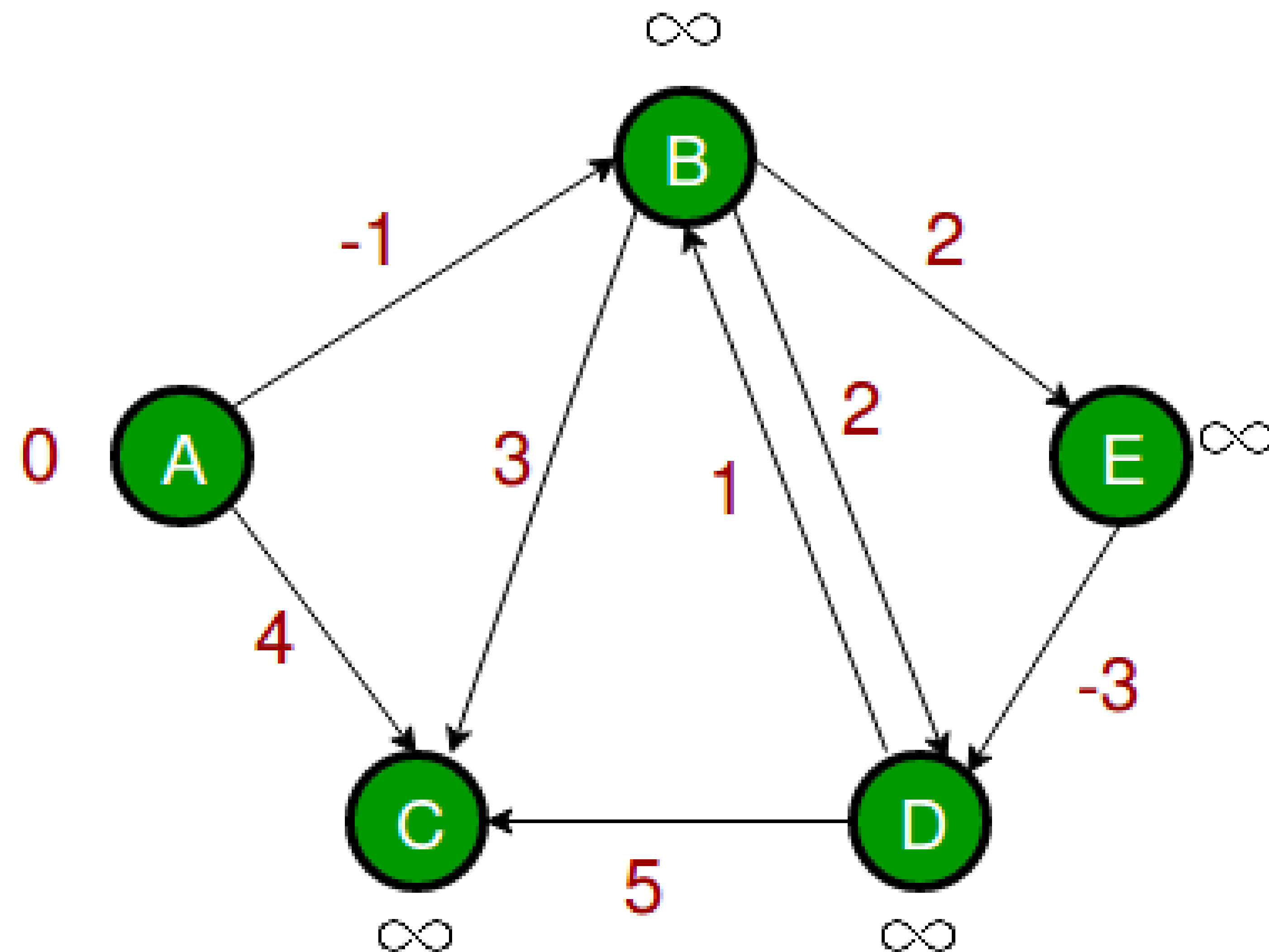
**ad ogni iterazione, vengono di nuovo processati tutti gli archi con lo stesso ordine della prima iterazione.*



Vertice	Peso	Predecessore
A	1	C
B	0	-
C	4	B
D	4	E
E	3	A

ESERCIZIO DA RISOLVERE

Dato il seguente grafo e dato **A come nodo sorgente**, trovare tramite l'algoritmo di Bellman-Ford, il cammino minimo tra il nodo sorgente e ogni altro nodo del grafo.



La scelta dell'ordine di rilassamento degli archi è a vostra discrezione. Una tra le varie possibilità è la seguente:

(B, E) (D, B) (B, D) (A, B) (A, C) (D, C) (B, C) (E, D)

In questo caso **saranno necessarie due iterazioni** e il **risultato finale** sarà quello in tabella.

Vertice	Peso	Predecessore
A	0	-
B	-1	A
C	2	B
D	-2	E
E	1	B

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 $\boldsymbol{O(VE)}$**
- Peggio di $O(V + E)$ richiesto da BFS nel caso di grafo non pesato (o con pesi tutti uguali alla stessa costante positiva)

Notazione:
E = |E| = n
V = |V| = m

ALGORITMI SU GRAFI (VARI!)

BFS:
- utile per cammini minimi in **grafi non pesati**
- visita i nodi per livelli

DFS:
- utile per esplorazione, ordinamento topologico, componenti, cicli, ecc.
- *non è un algoritmo generale per cammini minimi pesati*

Bellman-Ford:
- **utile per cammini minimi da una sorgente**
- funziona *anche con pesi negativi*
- rilassa tutti gli archi più volte

UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Prossima e ultima lezione (teorica): 14 ***maggio, h.14:00, aula 4C***