

# Algoritmi e Strutture Dati

## Heap e Code con Priorità

Alberto Montresor and Davide Rossi

Università di Bologna

9 dicembre 2024

This work is licensed under a Creative Commons  
Attribution-ShareAlike 4.0 International License.



# Sommario

## 1 Heap

- Introduzione
- Vettore heap
- HeapSort

## 2 Code con priorità

- Introduzione
- Implementazione con Heap

# Heap

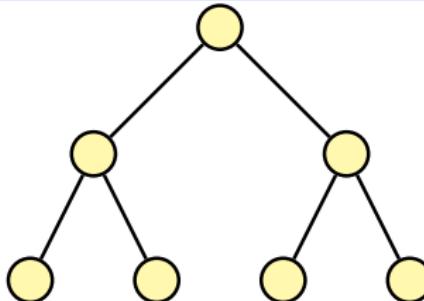
## Storia

- Struttura dati inventata da J. Williams nel 1964
- Utilizzata per implementare un nuovo algoritmo di ordinamento: HeapSort → *Algoritmo di ordinamento, come l'insertion sort effettua un ordinamento sul posto: in ogni istante soltanto un numero costante di elementi dell'array sono memorizzati all'esterno dell'array di input. Ma un costo computazionale di  $O(n \lg n)$*
- Williams intuì subito che poteva essere usata per altri scopi
- Seguiamo l'approccio storico nel presentare gli heap
  - Prima HeapSort
  - Poi Code con priorità

# Alberi binari

## Albero binario perfetto

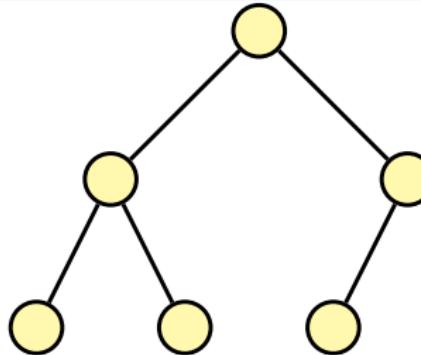
- Tutte le foglie hanno la stessa profondità  $h$
- Nodi interni hanno tutti grado 2
- Dato il numero di nodi  $n$ , ha altezza  $h = \lfloor \log n \rfloor$
- Dato l'altezza  $h$ , ha numeri di nodi  $n = 2^{h+1} - 1$



# Alberi binari

## Albero binario completo

- Tutte le foglie hanno profondità  $h$  o  $h - 1$
- Tutti i nodi a livello  $h$  sono “accatastati” a sinistra
- Tutti i nodi interni hanno grado 2, eccetto al più uno
- Dato il numero di nodi  $n$ , ha altezza  $h = \lfloor \log n \rfloor$



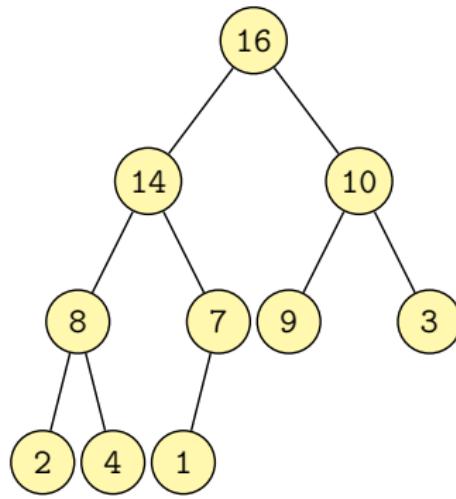
# Alberi binari heap

## Proprietà heap

Un **albero max-heap** (**min-heap**) è un albero binario completo tale che il valore memorizzato in ogni nodo è maggiore (minore) dei valori memorizzati nei suoi figli.

## Note

Le definizioni e gli algoritmi per alberi max-heap sono simmetrici rispetto agli algoritmi per alberi min-heap



# Alberi binari heap

- Un albero heap non impone una relazione di **ordinamento totale** fra i figli di un nodo
- Un albero heap è un **ordinamento parziale**
  - **Riflessivo**: Ogni nodo è  $\geq$  di se stesso
  - **Antisimmetrico**: se  $n \geq m$  e  $m \geq n$ , allora  $m = n$
  - **Transitivo**: se  $n \geq m$  e  $m \geq r$ , allora  $n \geq r$
- Ordinamenti parziali
  - Nozione più debole di un ordinamento totale...
  - ... ma più semplice da costruire

NOTA: i padri sono sempre a sx dei figli

# Alberi binari heap

## Vettore heap

Un albero heap può essere rappresentato tramite un **vettore heap**

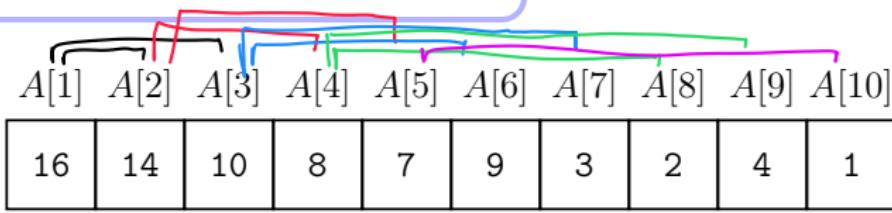
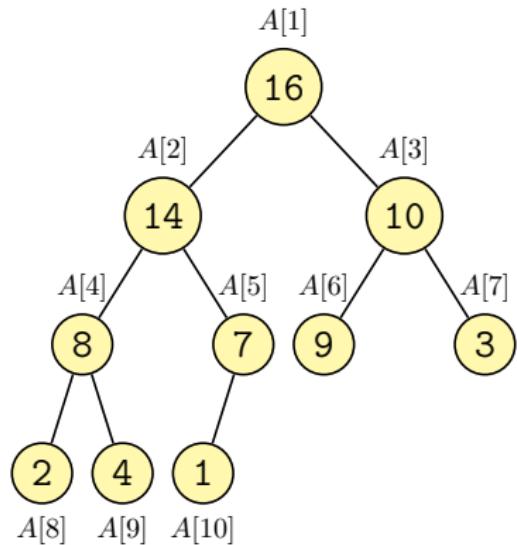
## Memorizzazione ( $A[1 \dots n]$ )

**Radice**  $\text{root}() = 1$

**Padre nodo  $i$**   $p(i) = \lfloor i/2 \rfloor$

**Figlio sx nodo  $i$**   $l(i) = 2i$

**Figlio dx nodo  $i$**   $r(i) = 2i + 1$



# Alberi binari heap

qua inizia a contare da 0

## Vettore heap

Un albero heap può essere rappresentato tramite un **vettore heap**

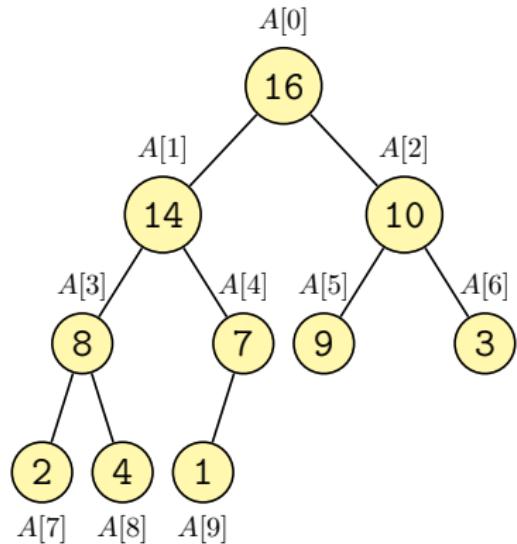
## Memorizzazione ( $A[0 \dots n - 1]$ )

**Radice**  $\text{root}() = 0$

**Padre nodo  $i$**   $p(i) = \lfloor (i - 1)/2 \rfloor$

**Figlio sx nodo  $i$**   $l(i) = 2i + 1$

**Figlio dx nodo  $i$**   $r(i) = 2i + 2$



$A[0] \ A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9]$

16	14	10	8	7	9	3	2	4	1
----	----	----	---	---	---	---	---	---	---

# Alberi binari heap

## Proprietà max-heap su vettore

$$\bullet A[i] \geq A[l(i)], A[i] \geq A[r(i)]$$

## Proprietà min-heap su vettore

$$\bullet A[i] \leq A[l(i)], A[i] \leq A[r(i)]$$

Ci sono due tipi di Heap binari: i max-heap e i min-heap. In entrambi i valori nei nodi soddisfano una proprietà di heap, che dipende dal tipo di heap.

**La proprietà del Max-heap** richiede che per ogni nodo  $i$  diverso dalla radice si abbia  $\bullet$  il valore di un nodo sia al massimo quello di suo padre. Di conseguenza l'elemento più grande di un max-heap è memorizzato nella radice e il sottosalbero di un nodo contiene valori non maggiori di quello contenuto nel nodo stesso.

**La proprietà del Min-Heap** richiede che per ogni nodo  $i$  diverso dalla radice si abbia  $\bullet$  il più piccolo elemento in un min-heap è nella radice (min-heap è organizzato nel modo opposto).

L'algoritmo di HeapSort utilizza il max-heap. I min-heap sono comunque usati per implementare le code di priorità.

# HeapSort

## Organizzazione heapsort()

Ordina un max-heap "in-place", prima costruendo un max-heap nel vettore e poi spostando l'elemento max in ultima posizione, ripristinando la proprietà max-heap

- **heapBuild()**

Costruisce un max-heap a partire da un vettore non ordinato

- **maxHeapRestore()**

Ripristina la proprietà max-heap

# maxHeapRestore()

## Input

Un vettore  $A$  e un indice  $i$ , tale per cui gli alberi binari con radici  $l(i)$  e  $r(i)$  sono max-heap

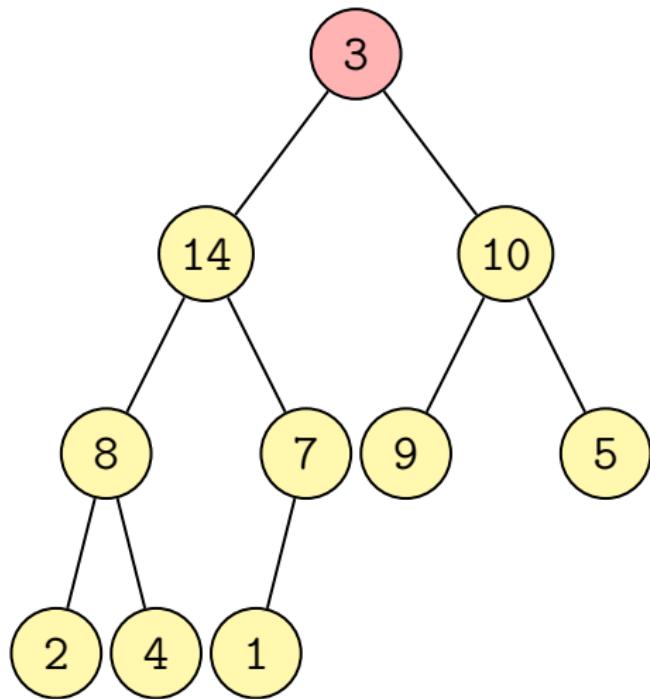
## Osservazione

- È possibile che  $A[i]$  sia minore di  $A[l(i)]$  o  $A[r(i)]$
- In altre parole, non è detto che il sottoalbero con radice  $i$  sia un max-heap

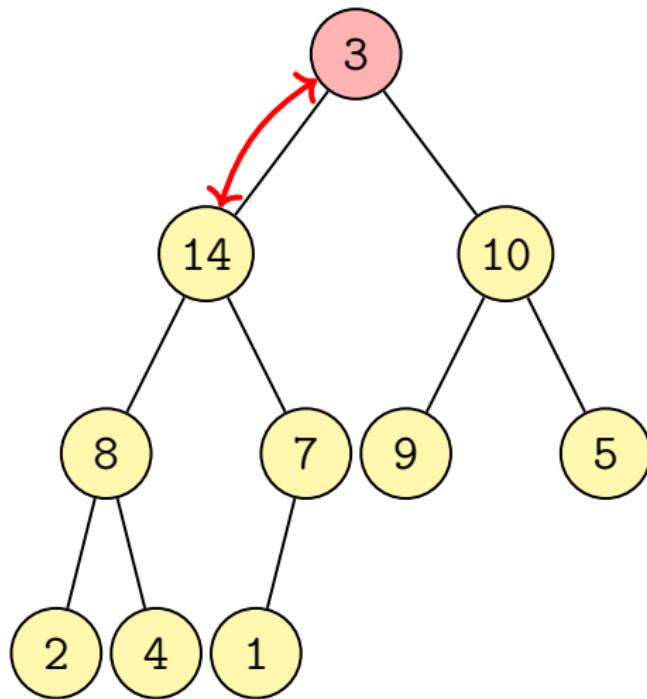
## Goal

Modificare in-place il vettore  $A$  in modo tale che l'albero binario con radice  $i$  sia un max-heap

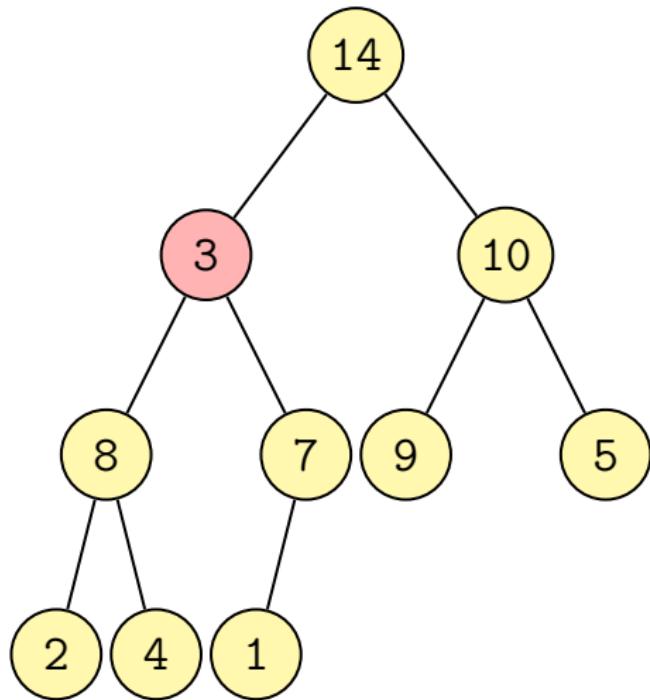
## Esempio



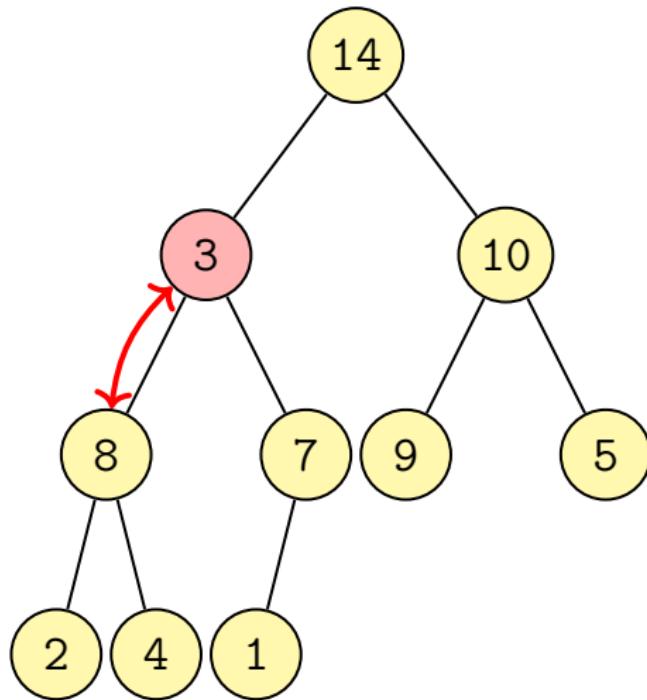
## Esempio



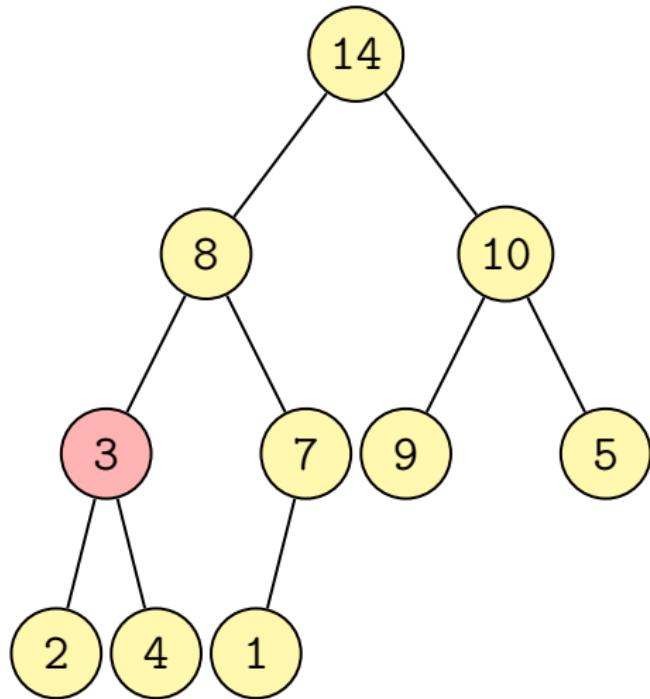
## Esempio



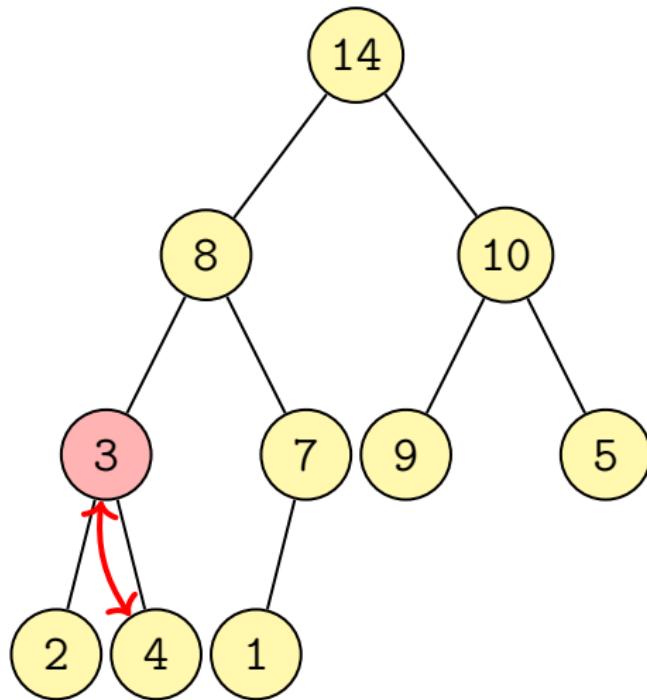
# Esempio



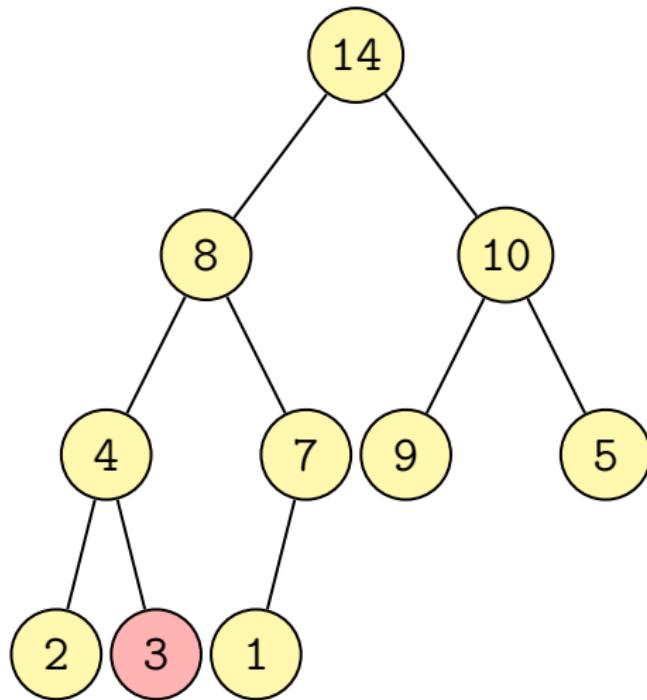
## Esempio



# Esempio



## Esempio



## Ripristinare la proprietà max-heap

---

`maxHeapRestore(ITEM[] A, int i, int dim)`

---

```

int max = i
if l(i) ≤ dim and A[l(i)] > A[max] then
    max = l(i)
if r(i) ≤ dim and A[r(i)] > A[max] then
    max = r(i)
if i ≠ max then
    swap(A, i, max) → scambia A[i] con A[max]
    maxHeapRestore(A, max, dim)

```

---

A ogni passo viene determinato il più grande tra gli elementi  $A[i]$ ,  $A[\text{left}(i)]$  e  $A[\text{right}(i)]$ , il suo indice viene memorizzato in  $\text{max}$

Qual è la complessità computazionale di `maxHeapRestore()`?

## maxHeapRestore() – Complessità computazionale

Qual è la complessità computazionale di `maxHeapRestore()`?

- Ad ogni chiamata, vengono eseguiti  $O(1)$  confronti
- Se il nodo  $i$  non è massimo, si richiama ricorsivamente `maxHeapRestore()` su uno dei figli
- L'esecuzione termina quando si raggiunge una foglia
- L'altezza dell'albero è pari a  $\lfloor \log n \rfloor$

Complessità

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

Ricchezza maxHeapRestore  $T(2n/3) + \Theta(1)$

1) Struttura di un Heap: Un heap binario è un albero binario quasi completo. Nel caso peggiore, quando stiamo ripristinando la proprietà della radice, la ricorsione potrebbe procedere lungo un percorso dove ad ogni passo la dimensione del problema si riduce a circa  $2/3$  della dimensione originale.

2) Caso peggiore: Si verifica quando l'albero è il più sbilanciato possibile. In un heap con  $n$  elementi, il sottoalbero più grande può avere al massimo  $2n/3$  elementi (questo accade quando un livello viene riempito a metà).

3) Operazioni costanti: Tutte le operazioni non ricorsive sono  $O(1)$

$$T(n) = T(2n/3) + O(1)$$

Teorema Master  $T(n) = \alpha T(n/b) + f(n)$

$$\alpha = 1$$

$$b = 3/2$$

$$f(n) = 1$$

$$f(n) = O(n^{\log_b a}) = O(n^{\log_{3/2} 1}) = O(n^0) = O(1)$$

$$\Rightarrow T(n) = G(n^{\log_b a} \log n) = \Theta(\log n)$$

# Dimostrazione correttezza (per induzione sull'altezza)

## Teorema

Al termine dell'esecuzione, l'albero radicato in  $A[i]$  rispetta la proprietà max-heap

# Dimostrazione correttezza (per induzione sull'altezza)

## Teorema

Al termine dell'esecuzione, l'albero radicato in  $A[i]$  rispetta la proprietà max-heap

## Caso base: altezza $h = 0$

Se  $h = 0$ , l'albero è dato da un solo nodo che rispetta la proprietà heap

## Ipotesi induttiva

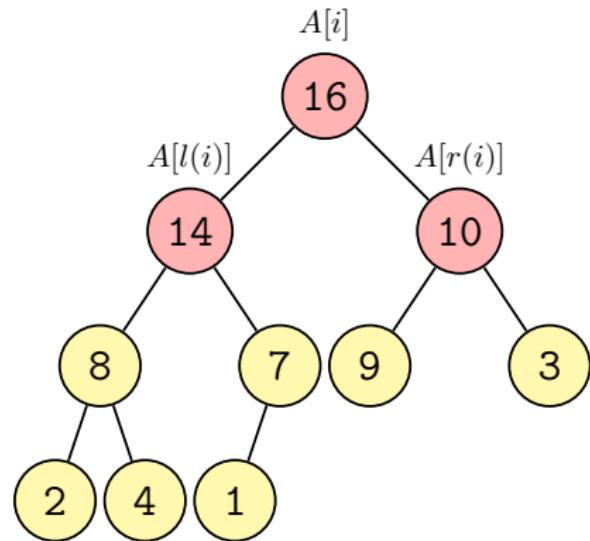
L'algoritmo funziona correttamente su tutti gli alberi di altezza minore di  $h$

# Dimostrazione correttezza (per induzione sull'altezza)

Induzione - Altezza  $h$  - Caso 1

$A[i] \geq A[l(i)], A[i] \geq A[r(i)]$ :

- L'albero radicato in  $A[i]$  rispetta la proprietà max-heap
- L'algoritmo termina (CVD)

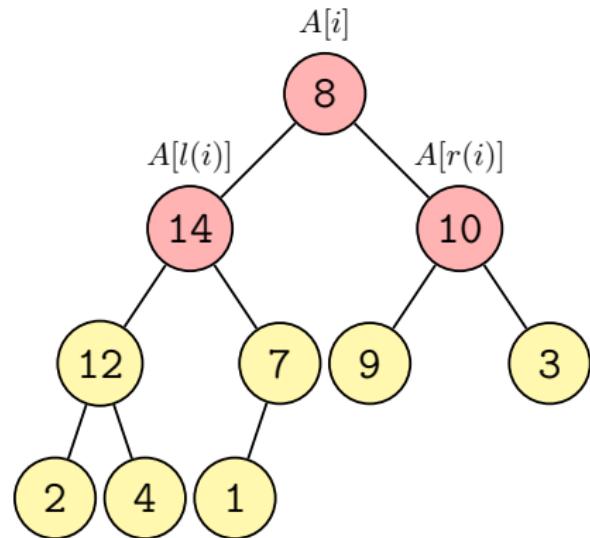


# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

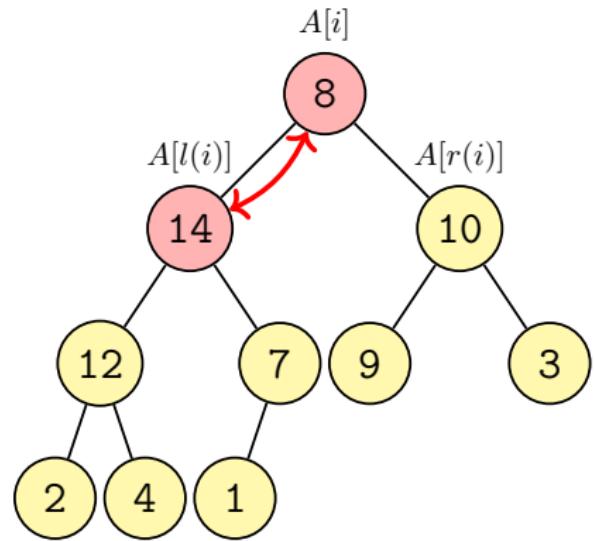
Simmetrico rispetto al Caso 2

# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  
 $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  
 $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

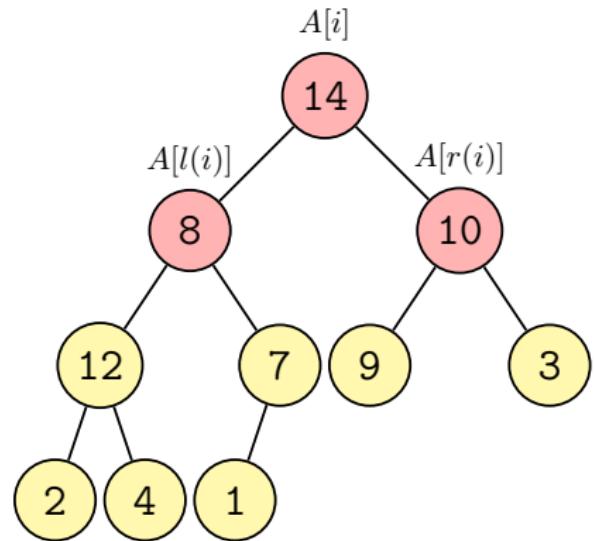
Simmetrico rispetto al Caso 2

# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

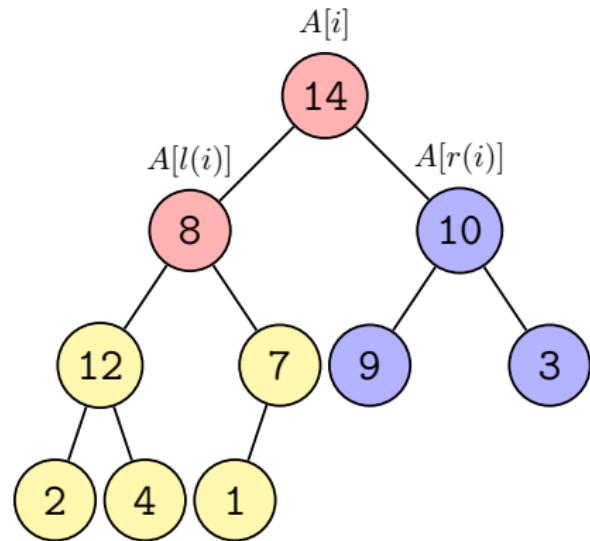
Simmetrico rispetto al Caso 2

# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

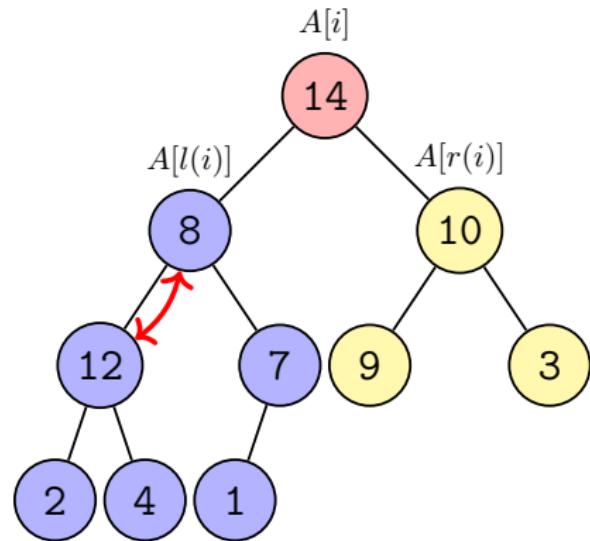
Simmetrico rispetto al Caso 2

# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

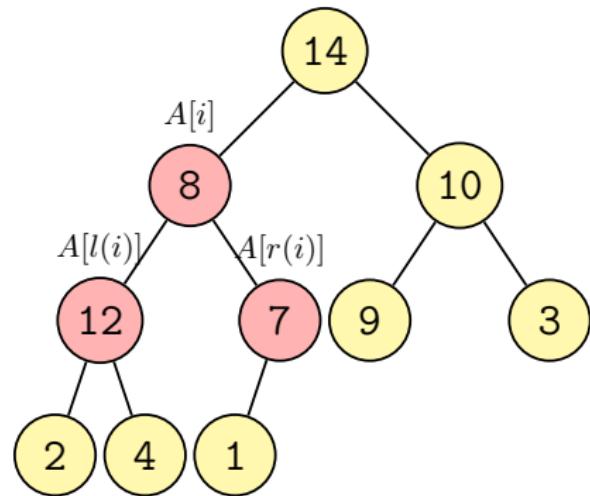
Simmetrico rispetto al Caso 2

# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  
 $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  
 $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

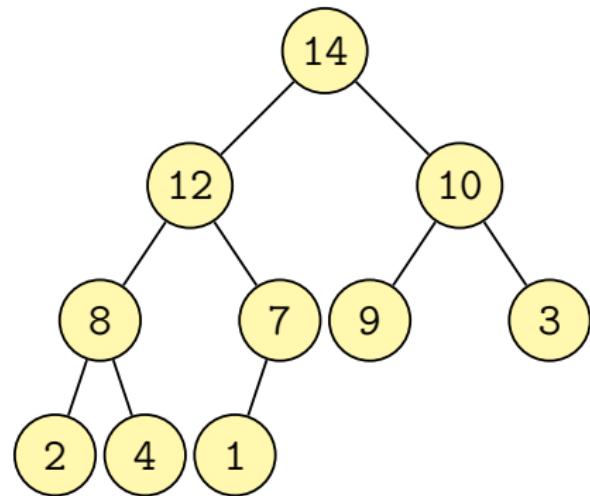
Simmetrico rispetto al Caso 2

# Dimostrazione correttezza (per induzione sull'altezza)

## Induzione - Altezza $h$ - Caso 2

$A[l(i)] > A[i], A[l(i)] > A[r(i)]$ :

- Viene fatto uno scambio  
 $A[i] \leftrightarrow A[l(i)]$
- Dopo lo scambio,  
 $A[i] \geq A[l(i)], A[i] \geq A[r(i)]$
- Il sottoalbero  $A[r(i)]$  è inalterato e rispetta la proprietà heap
- Il sottoalbero  $A[l(i)]$  può aver perso la proprietà heap
- Si applica `maxHeapRestore()` ricorsivamente su di  $A[l(i)]$ , che ha altezza minore di  $h$



## Passo induttivo - Caso 3

Simmetrico rispetto al Caso 2

## heapBuild()

### Principio di funzionamento

- Sia  $A[1 \dots n]$  un vettore da ordinare
- Tutti i nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap contenenti **1** elemento
- La procedura `heapBuild()`
  - attraversa i restanti **nodi dell'albero**, a partire da  $\lfloor n/2 \rfloor$  fino ad 1
  - esegue `maxHeapRestore()` su ognuno di essi

---

`heapBuild(ITEM[] A, int n)`

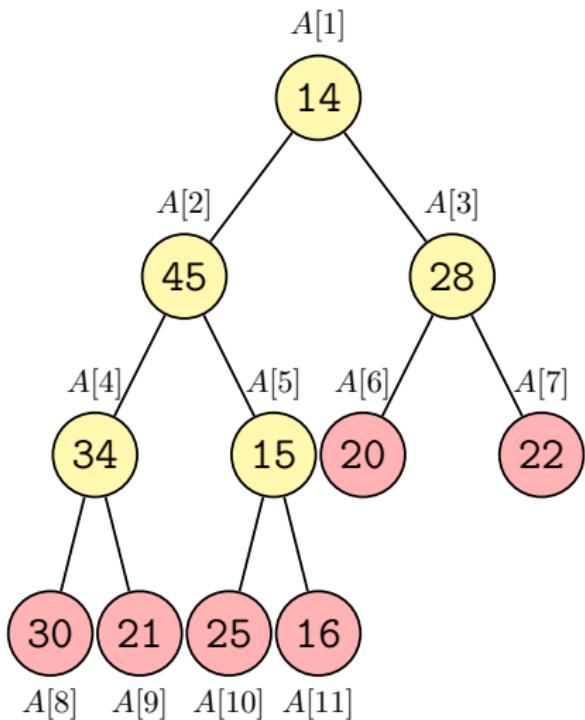
---

**for**  $i = \lfloor n/2 \rfloor$  **downto** 1 **do**  
  └ `maxHeapRestore(A, i, n)`

---

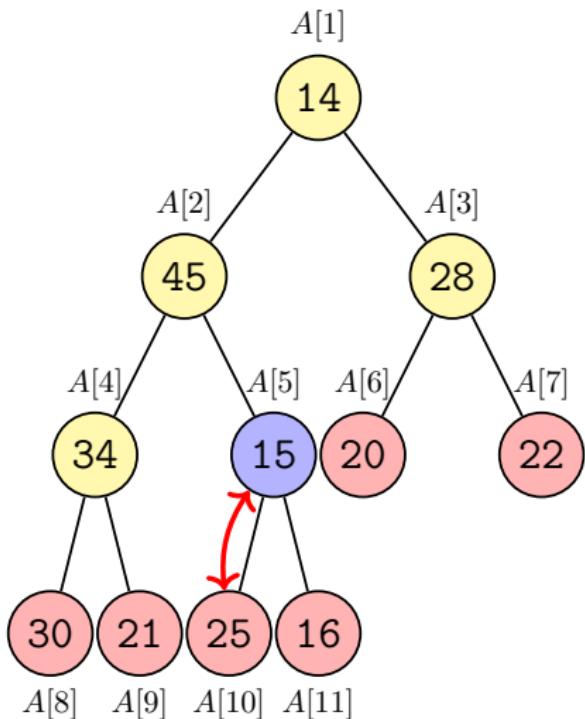
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



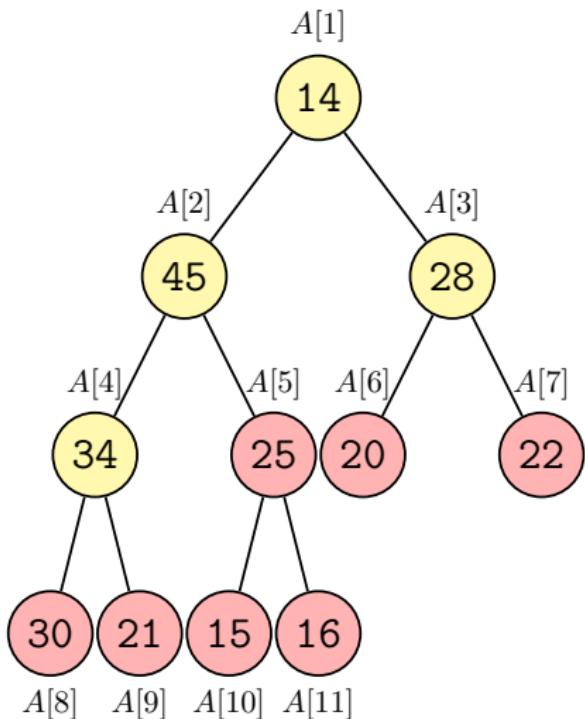
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



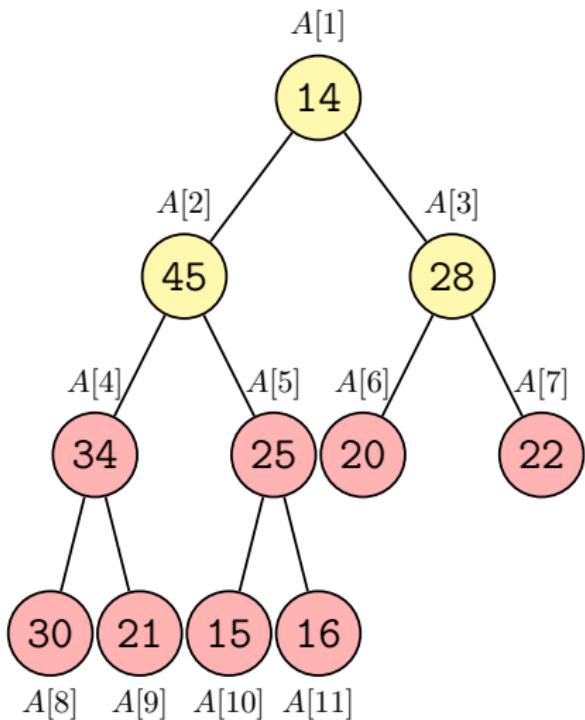
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



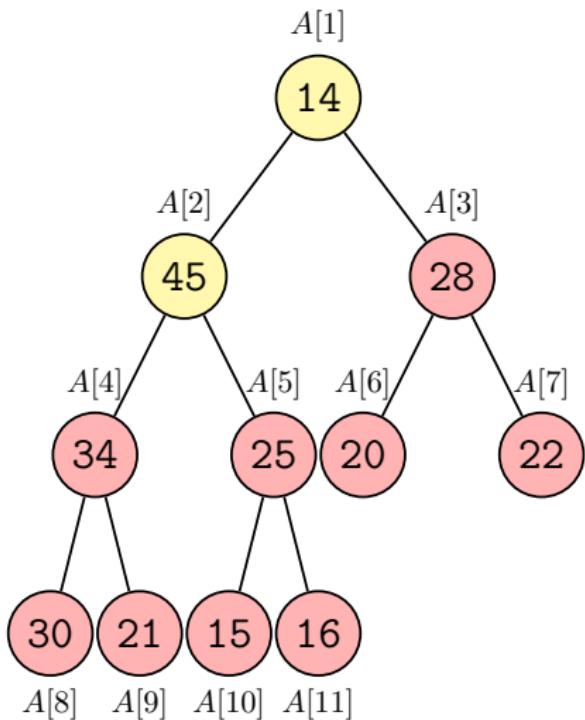
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



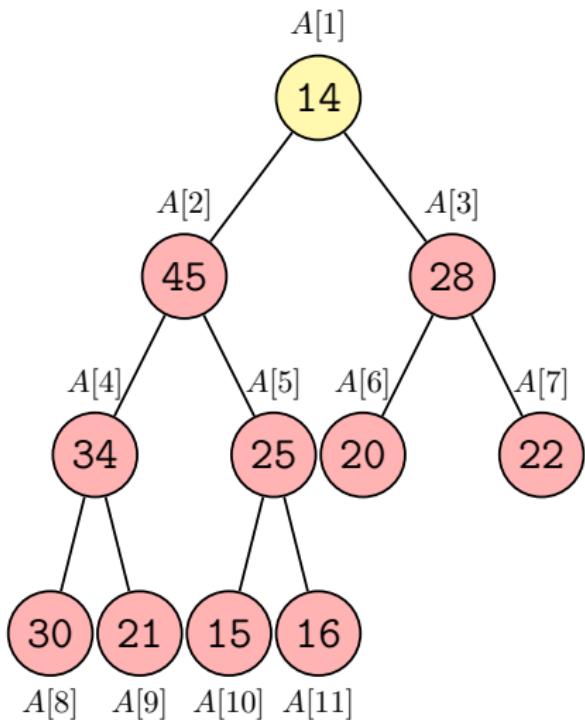
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



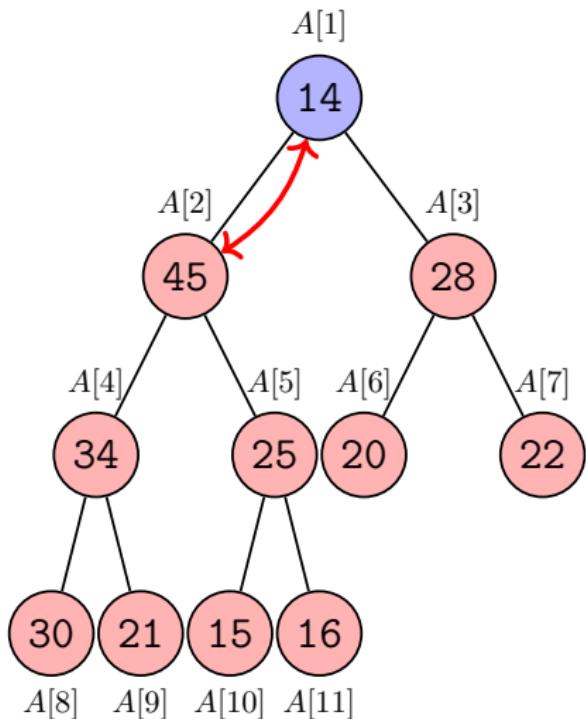
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



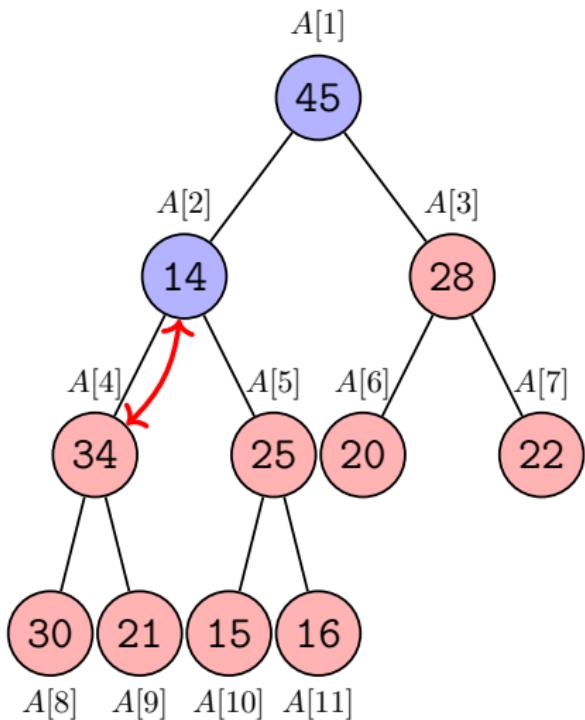
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



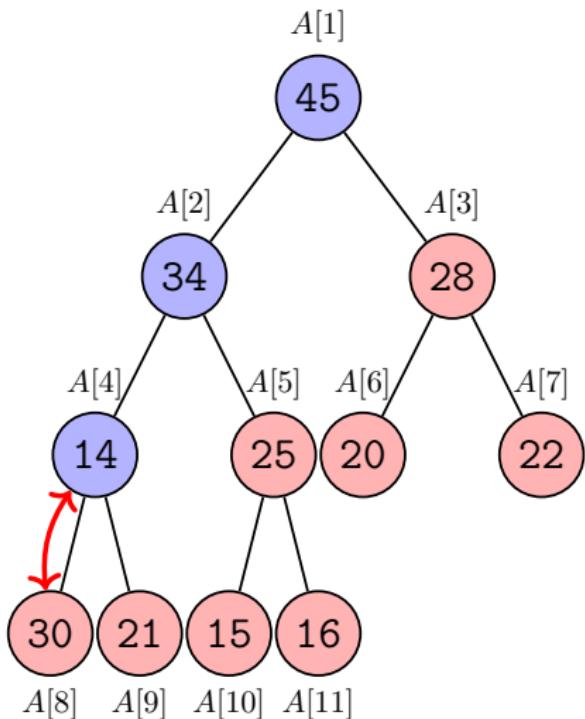
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



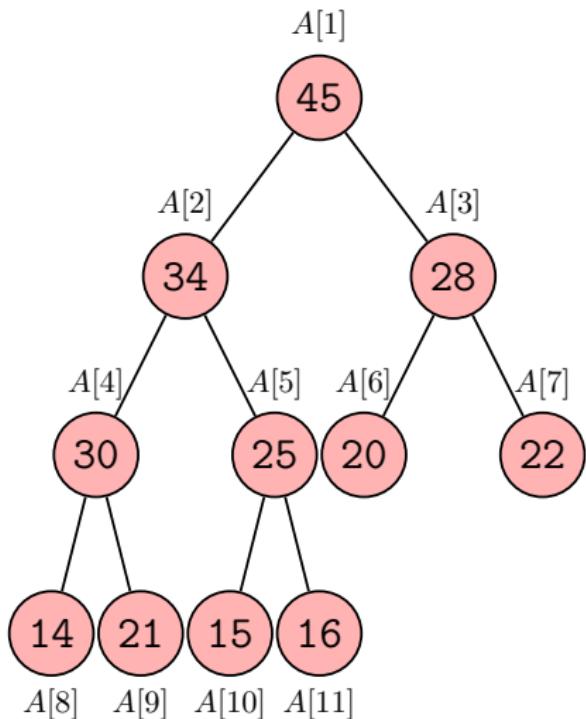
## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



## Esempio

- I nodi  $A[\lfloor n/2 \rfloor + 1 \dots n]$  sono foglie dell'albero e quindi heap di un elemento
- Per ogni posizione da  $\lfloor n/2 \rfloor$  fino ad 1, si esegue `maxHeapRestore()`



# Correttezza

Dobbiamo dimostrare che questa invariante è vera prima della prima iterazione del ciclo, che ogni iterazione conserva l'invariante, che il ciclo termina quando ciò avviene, che l'invariante fornisce un'utile proprietà per dimostrare la correttezza.

## Invariante di ciclo

All'inizio di ogni iterazione del ciclo **for**, i nodi  $[i + 1, \dots, n]$  sono radice di uno heap.

*Le foglie sono già heap banali*

## Dimostrazione – Inizializzazione

- All'inizio,  $i = \lfloor n/2 \rfloor$ .
- Supponiamo che  $\lfloor n/2 \rfloor + 1$  non sia una foglia
- Quindi ha almeno il figlio sinistro:  $2\lfloor n/2 \rfloor + 2$
- Questo ha indice  $n + 1$  oppure  $n + 2$ , assurdo perché  $n$  è la dimensione massima
- La dimostrazione vale per tutti gli indici successivi

*↳ L'ipotesi che  $\lfloor n/2 \rfloor + 1$  non sia una foglia porta a un indice del figlio scuro da licenzi, il che è impossibile  $\Rightarrow$  deve essere una foglia*

# Correttezza

## Invariante di ciclo

All'inizio di ogni iterazione del ciclo **for**, i nodi  $[i + 1, \dots, n]$  sono radice di uno heap.

## Dimostrazione – Conservazione

- È possibile applicare `maxHeapRestore` al nodo  $i$ , perché  $2i < 2i + 1 \leq n$  sono entrambi radici di heap
- Al termine dell'iterazione, tutti i nodi  $[i \dots n]$  sono radici di heap

## Dimostrazione – Conclusione

- Al termine,  $i = 0$ . Quindi il nodo 1 è radice di uno heap.

# Complessità

---

```
heapBuild(ITEM[ ] A, int n)
```

---

```
for i = ⌊n/2⌋ downto 1 do
    maxHeapRestore(A, i, n)
```

---

Qual è la complessità di HEAPBUILD()?

# Complessità

---

```
heapBuild(ITEM[ ] A, int n)
```

---

```
for i = ⌊n/2⌋ downto 1 do
    maxHeapRestore(A, i, n)
```

---

Qual è la complessità di HEAPBUILD()?

- Limite superiore:

# Complessità

---

```
heapBuild(ITEM[ ] A, int n)
```

---

```
for i = ⌊n/2⌋ downto 1 do
    maxHeapRestore(A, i, n)
```

---

Qual è la complessità di HEAPBUILD()?

- Limite superiore:  $T(n) = O(n \log n)$
- Limite inferiore:

# Complessità

---

```
heapBuild(ITEM[ ] A, int n)
```

---

```
for i = ⌊n/2⌋ downto 1 do
    maxHeapRestore(A, i, n)
```

---

Qual è la complessità di HEAPBUILD()?

- Limite superiore:  $T(n) = O(n \log n)$
- Limite inferiore:  $T(n) = \Omega(n \log n)$ ?

# Complessità

Le operazioni `maxHeapRestore()`  
vengono eseguite un numero  
decrescente di volte su heap di  
altezza crescente

Altezza	# Volte
0	$\lfloor n/2 \rfloor$
1	$\lfloor n/4 \rfloor$
2	$\lfloor n/8 \rfloor$
...	...
$h$	$\lfloor n/2^{h+1} \rfloor$

Formula:  $\sum_{h=1}^{+\infty} h x^h = \frac{x}{(1-x)^2}$ , per  $x < 1$

$$\begin{aligned}
 T(n) &\leq \sum_{h=1}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} h \\
 &= n \sum_{h=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^{h+1} h \\
 &= n/2 \sum_{h=1}^{\lfloor \log n \rfloor} \left(\frac{1}{2}\right)^h h \\
 &\leq n/2 \sum_{h=1}^{+\infty} \left(\frac{1}{2}\right)^h h \quad = n = O(n)
 \end{aligned}$$

Dunque possiamo costruire un max-heap a partire a partire da un array non ordinato in un tempo lineare

# heapSort()

## Principio di funzionamento

- L'elemento in prima posizione contiene il massimo
- Viene collocato in fondo
- L'elemento in fondo viene spostato in testa
- Si chiama `maxHeapRestore()` per ripristinare la situazione
- La dimensione dello heap viene progressivamente ridotta (indice  $i$ )

---

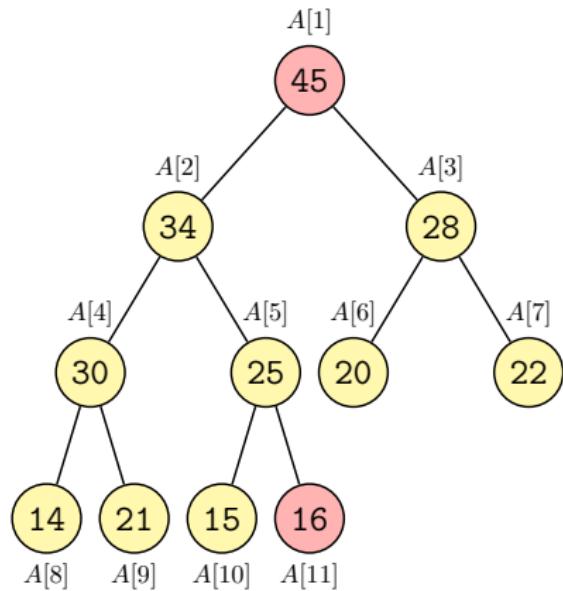
HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )  $\mathcal{O}(n)$   
for  $i = n$  downto 2 do  $n-1$   
    swap( $A, 1, i$ )  $\mathfrak{1}$   
    maxHeapRestore( $A, 1, i - 1$ )  $\mathcal{O}(\log n)$

---

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

45	34	28	30	25	20	22	14	21	15	16
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

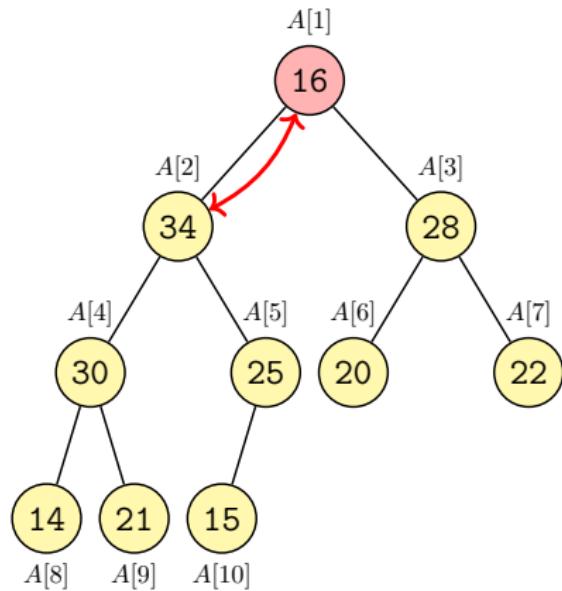
swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

Lo swap scambia la radice con l'ultimo elemento dell'heap. Successivamente viene fatto un maxHeapRestore senza l'ultimo elemento dell'heap

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

16	34	28	30	25	20	22	14	21	15	45
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

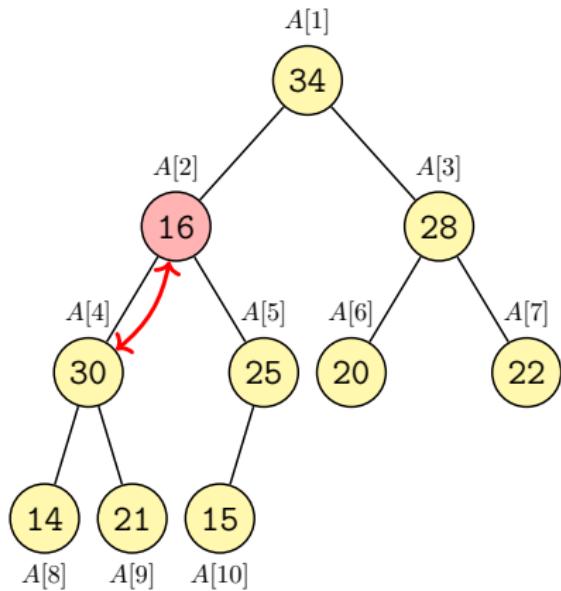
for  $i = n$  downto 2 do

swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

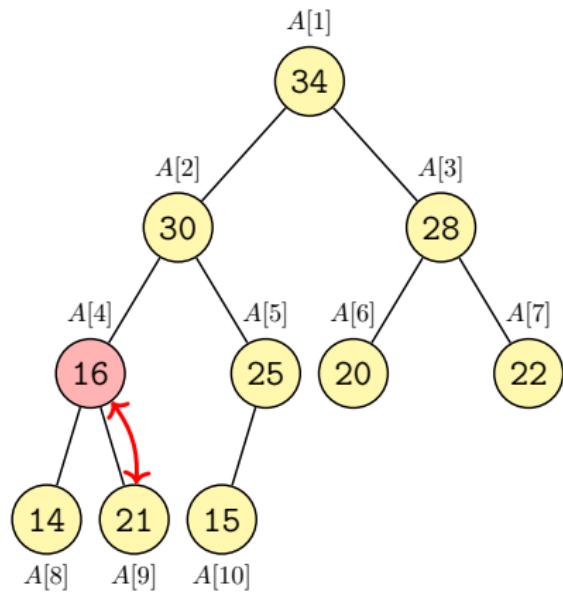
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

34	16	28	30	25	20	22	14	21	15	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

34	30	28	16	25	20	22	14	21	15	45
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

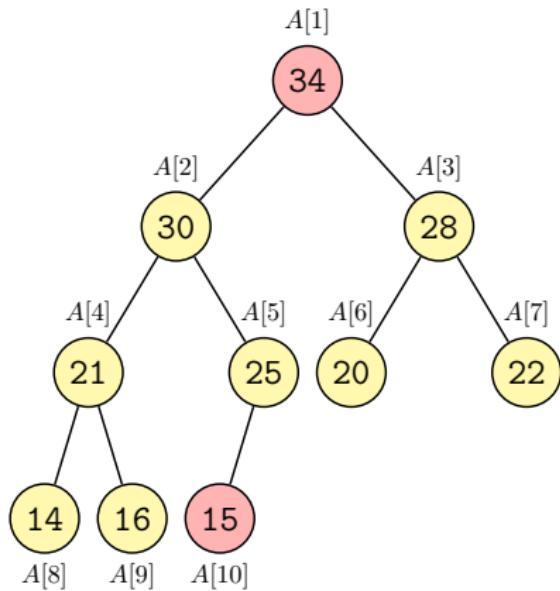
for  $i = n$  downto 2 do

    swap( $A, 1, i$ )

    maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

34	30	28	21	25	20	22	14	16	15	45
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

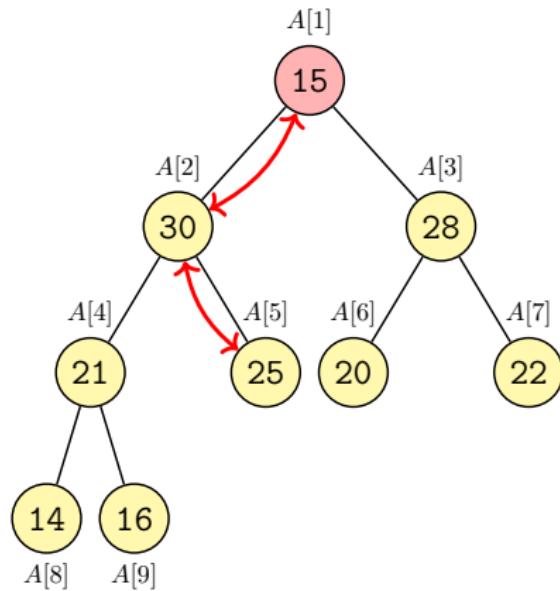
for  $i = n$  downto 2 do

swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

15	30	28	21	25	20	22	14	16	34	45
----	----	----	----	----	----	----	----	----	----	----

---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

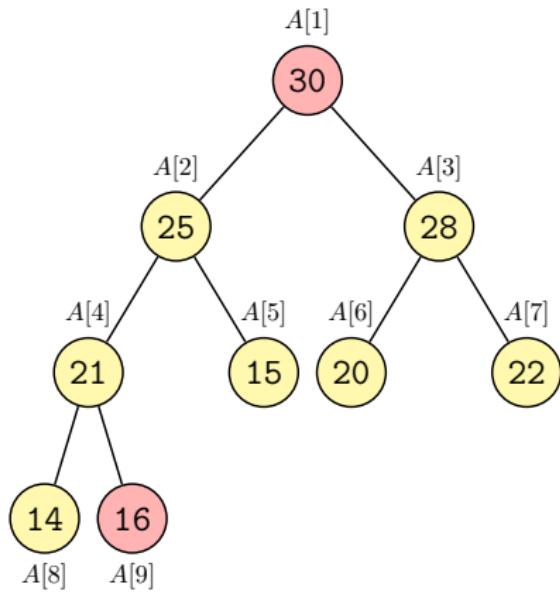
for  $i = n$  downto 2 do

swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

30	25	28	21	15	20	22	14	16	34	45
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

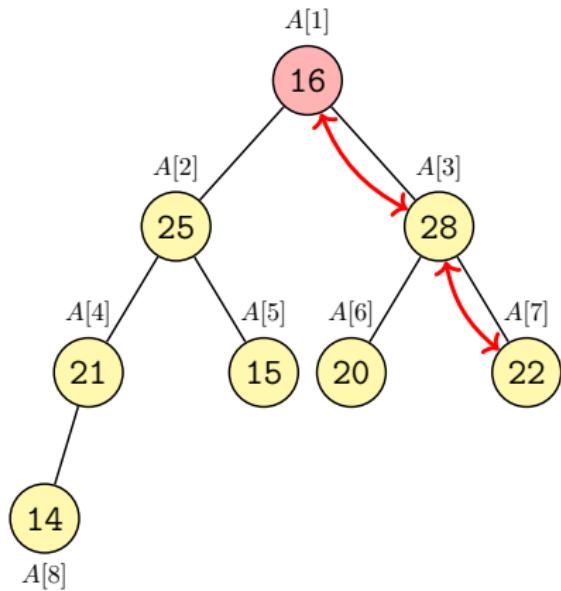
for  $i = n$  downto 2 do

swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

16	25	28	21	15	20	22	14	30	34	45
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

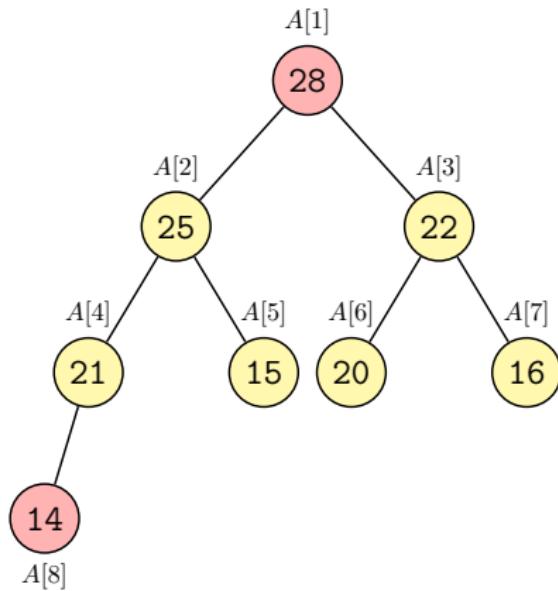
for  $i = n$  downto 2 do

swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio



$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

28	25	22	21	15	20	16	14	30	34	45
----	----	----	----	----	----	----	----	----	----	----

---

HEAPSORT(ITEM[]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

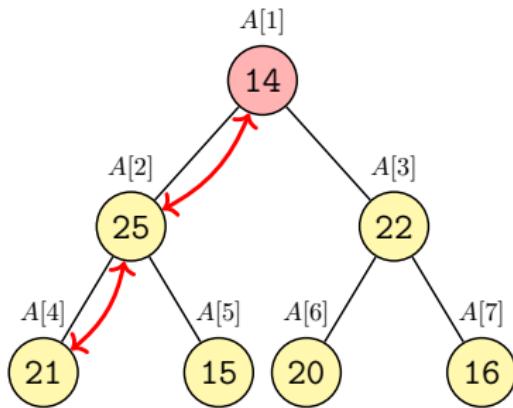
for  $i = n$  downto 2 do

swap( $A, 1, i$ )

maxHeapRestore( $A, 1, i - 1$ )

---

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

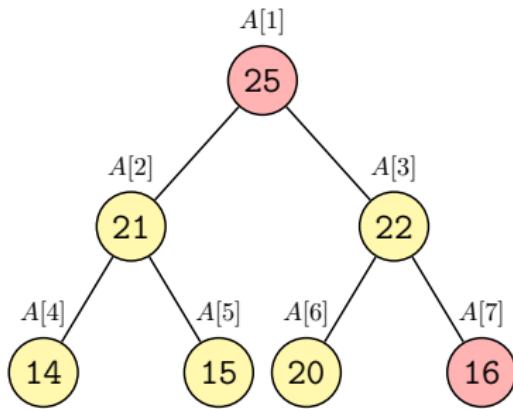
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

14	25	22	21	15	20	16	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  **downto** 2 do

swap( $A, 1, i$ )

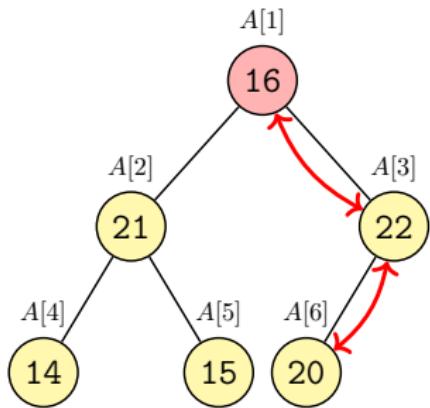
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

25	21	22	14	15	20	16	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

    swap( $A, 1, i$ )

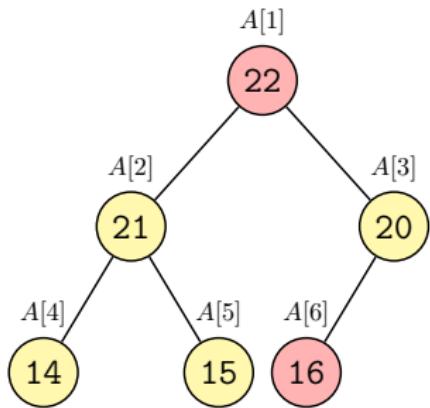
    maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

16	21	22	14	15	20	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

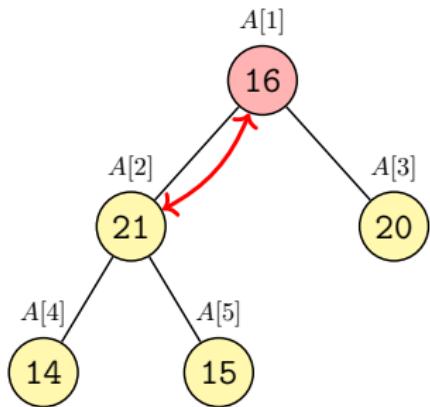
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

22	21	20	14	15	16	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[ ]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

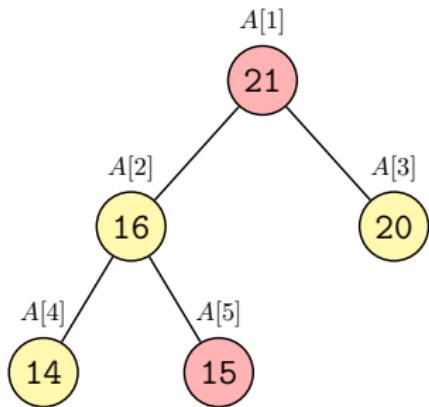
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

16	21	20	14	15	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

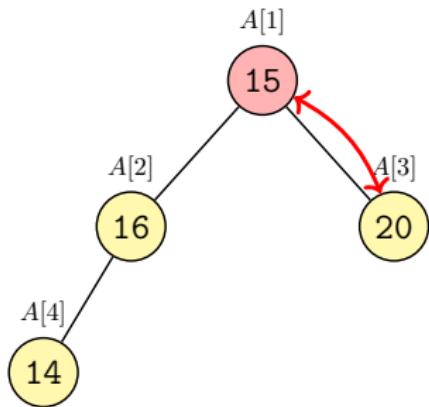
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

21	16	20	14	15	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[ ]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

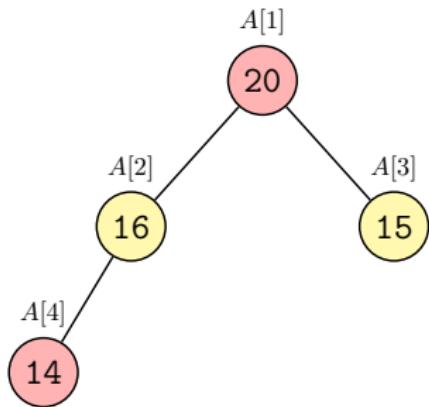
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

15	16	20	14	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

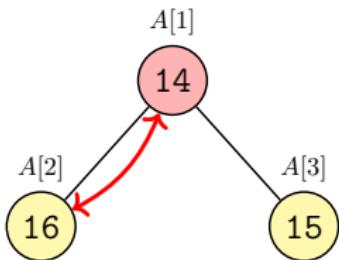
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

20	16	15	14	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[ ]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  **downto** 2 do

swap( $A, 1, i$ )

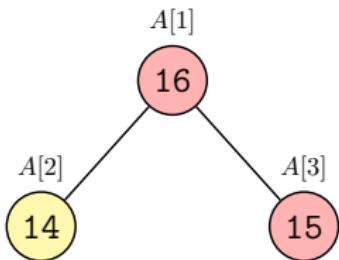
**maxHeapRestore**( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

14	16	15	20	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio




---

**HEAPSORT(ITEM[]  $A$ , int  $n$ )**

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

swap( $A, 1, i$ )

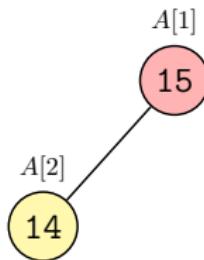
maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

16	14	15	20	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio



---

HEAPSORT(ITEM[ ]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

    swap( $A, 1, i$ )

    maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] \ A[2] \ A[3] \ A[4] \ A[5] \ A[6] \ A[7] \ A[8] \ A[9] \ A[10] \ A[11]$

15	14	16	20	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Esempio

$A[1]$   
14

---

---

HEAPSORT(ITEM[ ]  $A$ , int  $n$ )

---

heapBuild( $A, n$ )

for  $i = n$  downto 2 do

    swap( $A, 1, i$ )

    maxHeapRestore( $A, 1, i - 1$ )

---

$A[1] A[2] A[3] A[4] A[5] A[6] A[7] A[8] A[9] A[10] A[11]$

14	15	16	20	21	22	25	28	30	34	45
----	----	----	----	----	----	----	----	----	----	----

# Complessità

## Complessità

- `heapBuild()` costa  $\Theta(n)$
- `maxHeapRestore()` costa  $\Theta(\log i)$  in un heap con  $i$  elementi
- Viene eseguita con  $i$  che varia da 2 a  $n$

$$T(n) = \sum_{i=2}^n \log i + \Theta(n) = \Theta(n \log n)$$


# Correttezza

## Invariante di ciclo

Al passo  $i$

- il sottovettore  $A[i + 1 \dots n]$  è ordinato;
- $A[1 \dots i] \leq A[i + 1 \dots n]$
- $A[1]$  è la radice di un vettore heap di dimensione  $i$ .

## Dimostrazione

Per esercizio

## Dimostrazione dell'Invariante di ciclo dell'HeapSort

Base dell'induzione ( $i=n$ )

Sottovettore  $\rightarrow A[n+1, \dots, n]$  è vuoto quindi è ordinato per definizione

$A[1, \dots, n] \leq A[n+1, \dots, n]$  è vero perché il secondo vettore è vuoto

oss/

La relazione " $\leq$ " tra un array e un array vuoto è considerata vera per definizione, perché non esiste alcun elemento nell'array vuoto che violi la relazione

$A[1]$  è radice di un heap perché abbiamo appena costruito l'heap classico sull'intero array

Passo induttivo

Assumiamo che l'invariante valga al passo  $i$  e dimostriamo che vale al passo  $i-1$

1) Scambia  $A[i]$  con  $A[i]$

- $A[1]$  (classico dell'heap) viene spostato in posizione  $i$
- Per ipotesi induttiva,  $A[i+1, \dots, n]$  è ordinato e contiene elementi  $\geq A[i]$
- Quindi  $A[i, \dots, n]$  è ordinato

2) Ricomposta l'heap a  $i-1$

- Riduciamo le dimensioni dell'heap a  $i-1$
- Chiamiamo heapify su  $A[1]$ , che riporta in ordine la proprietà di Heap
- Per costruzione,  $A[1]$  è ora radice di un heap di dimensione  $i-1$

3) Relazione tra le parti

- Tutti gli elementi in  $A[1, \dots, i-1]$  sono  $\leq A[1]$  (per prop. di heap)
- $A[1] \leq$  tutti gli elementi  $A[i, \dots, n]$  (perché era il minimo tra i classici)
- Quindi  $A[1, \dots, i-1] \leq A[i, \dots, n]$

## Reality check

**Utilizzo** (<https://en.wikipedia.org/wiki/Heapsort>)

*Because of the  $O(n \log n)$  upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort, such as the Linux kernel.*

# Sommario

## 1 Heap

- Introduzione
- Vettore heap
- HeapSort

## 2 Code con priorità

- Introduzione
- Implementazione con Heap

# Code con priorità

## Definizione (Priority Queue)

Una **coda con priorità** è una struttura dati astratta, simile ad una coda, in cui ogni elemento inserito possiede una sua "**priorità**"

- **Min-priority queue:** estrazione per valori crescenti di priorità
- **Max-priority queue:** estrazione per valori decrescenti di priorità

## Operazioni permesse

- Inserimento in coda
- Estrazione dell'elemento con priorità di valore min/max
- Modifica priorità (decremento/incremento) di un elemento inserito

# Specifica

## MINPRIORITYQUEUE

% Crea una coda con priorità con capacità  $n$ , vuota

PRIORITYQUEUE PriorityQueue(**int**  $n$ )

% Restituisce **true** se la coda con priorità è vuota

**boolean** isEmpty()

% Restituisce l'elemento minimo di una coda con priorità non vuota

ITEM min()

% Rimuove e restituisce il minimo da una coda con priorità non vuota

ITEM deleteMin()

% Inserisce l'elemento  $x$  con priorità  $p$  nella coda con priorità. Restituisce

% un oggetto PRIORITYITEM che identifica  $x$  all'interno della coda

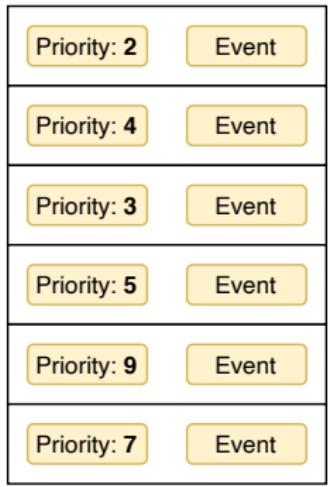
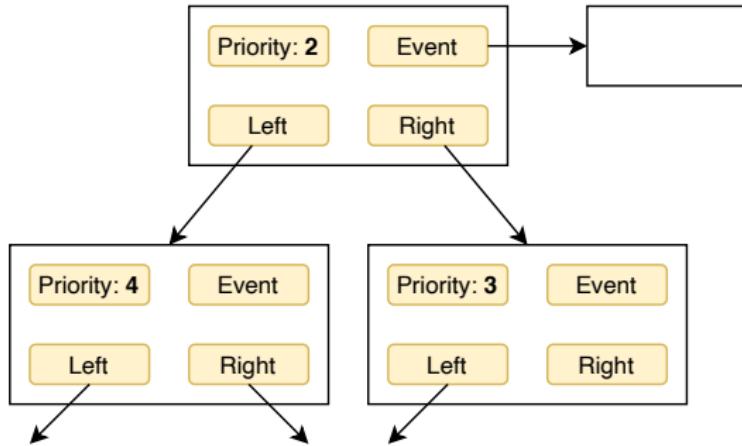
PRIORITYITEM insert(ITEM  $x$ , **int**  $p$ )

% Diminuisce la priorità dell'oggetto identificato da  $y$  portandola a  $p$

decrease(PRIORITYITEM  $y$ , **int**  $p$ )

# Reality Check – Simulatore event-driven

- Ad ogni evento è associato un timestamp di esecuzione
- Ogni evento può generare nuovi eventi, con timestamp arbitrari
- Una coda con min-priorità può essere utilizzata per eseguire gli eventi in ordine di timestamp



# Implementazioni

Metodo	Lista/vettore non ordinato	Lista Ordinata	Vettore Ordinato	Albero RB
<code>min()</code>	$O(n)$	$O(1)$	$O(1)$	$O(\log n)$
<code>deleteMin()</code>	$O(n)$	$O(1)$	$O(n)$	$O(\log n)$
<code>insert()</code>	$O(1)$	$O(n)$	$O(n)$	$O(\log n)$
<code>decrease()</code>	$O(1)$	$O(n)$	$O(\log n)$	$O(\log n)$

## Heap

Una struttura dati speciale che associa

- i vantaggi di un albero (esecuzione in tempo  $O(\log n)$ ), e
- i vantaggi di un vettore (memorizzazione efficiente)

# Implementazione con Heap

## Quale versione

Implementiamo una min-priority queue, in quanto negli esempi che vedremo in seguito daremo la precedenza a elementi con priorità minore

## Dettagli implementativi

- Vedremo come strutturare un vettore che memorizza coppie  $\langle \text{valore}, \text{priorità} \rangle$
- Vedremo come implementare `minHeapRestore()`
- Vedremo come implementare i singoli metodi

# Memorizzazione

Rappresenta un elemento in una coda con priorità implementata tramite heap

---

PRIORITYITEM

---

int priority

% Priorità

ITEM value

% Elemento

int pos

% Posizione nel vettore heap

---

Scambia due elementi nell'array heap H e aggiorna le loro posizioni

---

swap(PRIORITYITEM[] H, int i, int j)

---

PRIORITYITEM temp = H[i]

H[i] = H[j]

H[j] = temp

H[i].pos = i

H[j].pos = j

---

# Inizializzazione

---

## PRIORITYQUEUE

---

**int** *capacity* % Numero massimo di elementi nella coda  
**int** *dim* % Numero attuale di elementi nella coda  
**PRIORITYITEM[ ]** *H* % Vettore *heap*

PRIORITYQUEUE PriorityQueue(**int** *n*)

PRIORITYQUEUE *t* = **new** PRIORITYQUEUE

*t.capacity* = *n*

*t.dim* = 0

*t.H* = **new** PRIORITYITEM[1...*n*] —> viene inizializzato il vettore

**return** *t*

---

# Inserimento

---

PRIORITYITEM insert(ITEM  $x$ , int  $p$ )

**precondition:**  $dim < capacity$

$dim = dim + 1$

$H[dim] = \text{new PRIORITYITEM}()$

$H[dim].value = x$

$H[dim].priority = p$

- $H[dim].pos = dim$

int  $i = dim$

while  $i > 1$  and  $H[i].priority < H[p(i)].priority$  do

swap( $H, i, p(i)$ )

$i = p(i)$

return  $H[i]$

---

Il ciclo while serve a ripristinare la priorità di Heap originale dopo l'inserimento di un nuovo elemento

$i > 1 \rightarrow$  continua finché non raggiungiamo la radice

$H[i].priority < H[p(i)].priority$

$\rightarrow$  Continua finché l'elemento corrente ha priorità minore del suo genitore (viola la priorità di heap originale)

Swap  $\rightarrow$  L'elemento viene scambiato con il genitore  
 $\hookrightarrow$  il processo continua verso la radice finché:  
 - Si raggiunge la radice ( $i=1$ ) oppure

- L'elemento non è più minore del suo genitore (proprietà heap rispettata)

## minHeapRestore()

*→ ripristina la proprietà del min-heap*

---

minHeapRestore(PRIORITYITEM[ ]  $A$ , int  $i$ , int  $dim$ )

---

int  $min = i$

if  $l(i) \leq dim$  and  $A[l(i)].priority < A[min].priority$  then

$min = l(i)$

if  $r(i) \leq dim$  and  $A[r(i)].priority < A[min].priority$  then

$min = r(i)$

if  $i \neq min$  then

  swap( $A, i, min$ )

  minHeapRestore( $A, min, dim$ )

---

# Cancellazione / lettura minimo

---

ITEM deleteMin()

---

**precondition:**  $dim > 0$

$\text{swap}(H, 1, dim)$

$dim = dim - 1$

$\text{minHeapRestore}(H, 1, dim)$

**return**  $H[dim + 1].value$   $\rightarrow$  *ritorna l'item cancellato*

---

---

ITEM min()

---

**precondition:**  $dim > 0$

**return**  $H[1].value$

---

# Decremento priorità

---

decrease(PRIORITYITEM  $x$ , int  $p$ )

---

**precondition:**  $p < x.priority$

$x.priority = p$

int  $i = x.pos$

**while**  $i > 1$  **and**  $H[i].priority < H[p(i)].priority$  **do**

swap( $H, i, p(i)$ )  
  |  
  |  $i = p(i)$

---

# Complessità

- Tutte le operazioni che modificano gli heap sistemano la proprietà heap
  - lungo un cammino radice-foglia (`deleteMin()`)
  - oppure lungo un cammino nodo-radice (`insert()`, `decrease()`)
- Poichè l'altezza è  $\lfloor \log n \rfloor$ , il costo di tali operazioni è  $O(\log n)$

Operazione	Costo
<code>insert()</code>	$O(\log n)$
<code>deleteMin()</code>	$O(\log n)$
<code>min()</code>	$\Theta(1)$
<code>decrease()</code>	$O(\log n)$

# Heap (mucchio) of presents

