Università di Napoli Federico II – Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Corso di Algoritmi e Strutture Dati

Heapsort

Heapsort

- Abbiamo visto
  - Insertion Sort
    - $\Theta(n^2)$ nel caso peggiore, $\Theta(n)$ nel caso migliore
    - Ordina sul posto
- Merge Sort
  - $\Theta(n \lg n)$
  - Non ordina sul posto

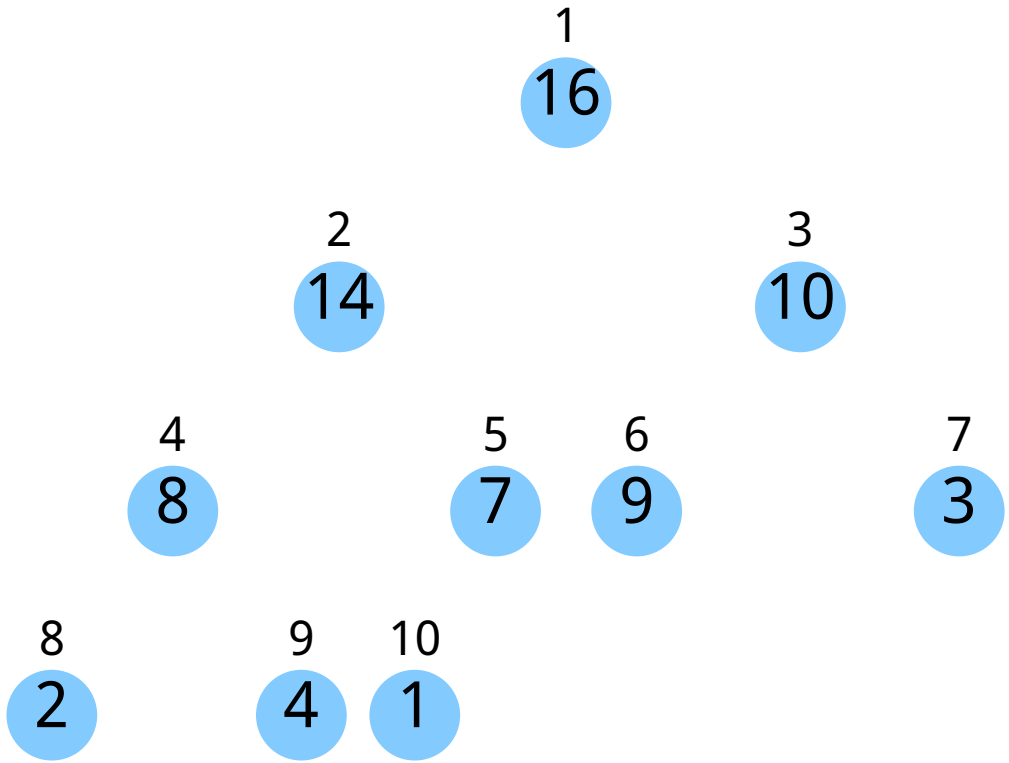
- **Vediamo ora Heap Sort**
  - **$O(n \lg n)$**
  - **Ordina sul posto**

- Basato sulla struttura dati heap

Heap

- Un heap è una struttura dati che organizza gli elementi in un array in modo da verificare una data proprietà
- Un modo alternativo per vedere la disposizione degli elementi è utilizzare un albero binario (quasi) completo

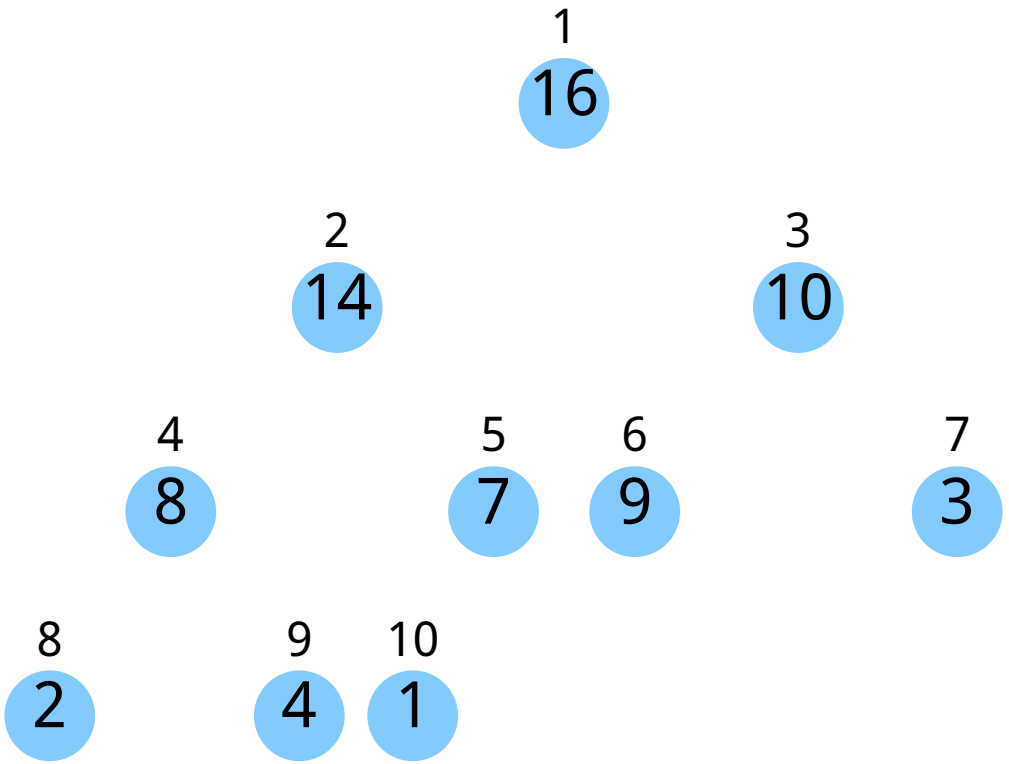
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Heap

- heap-size[A] <= length[A]
- Parent(i) = ⌊i/2⌋
- Left(i) = 2*i
- Right(i) = 2*i + 1
- Operazioni implementabili in maniera efficiente

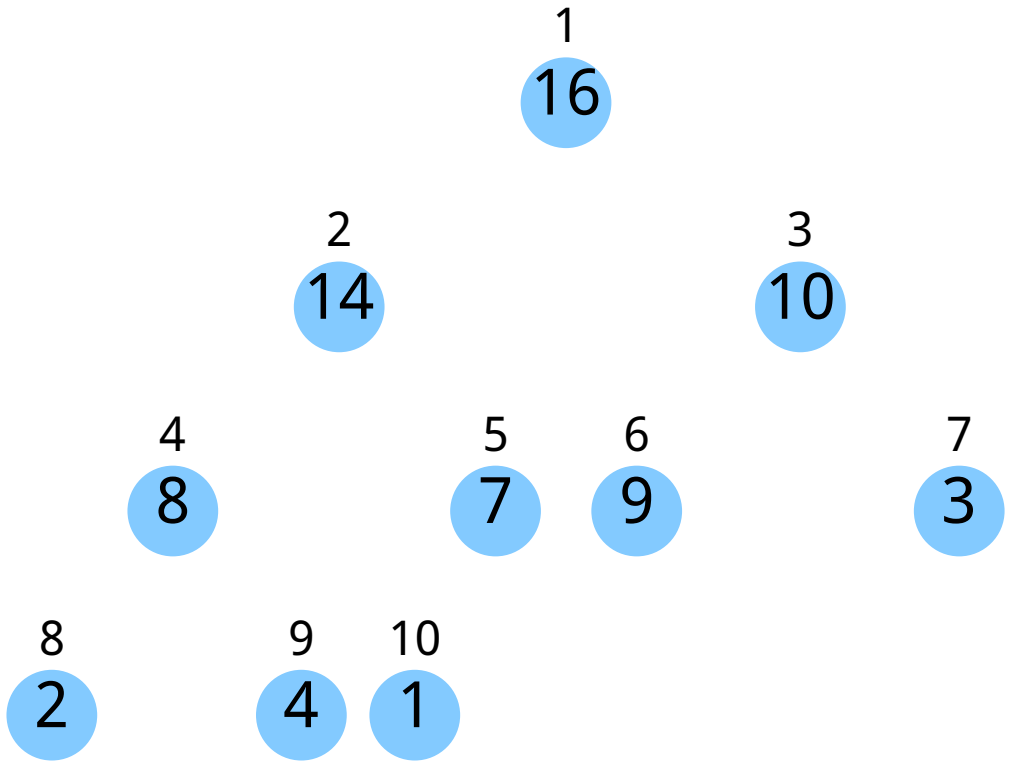
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Heap

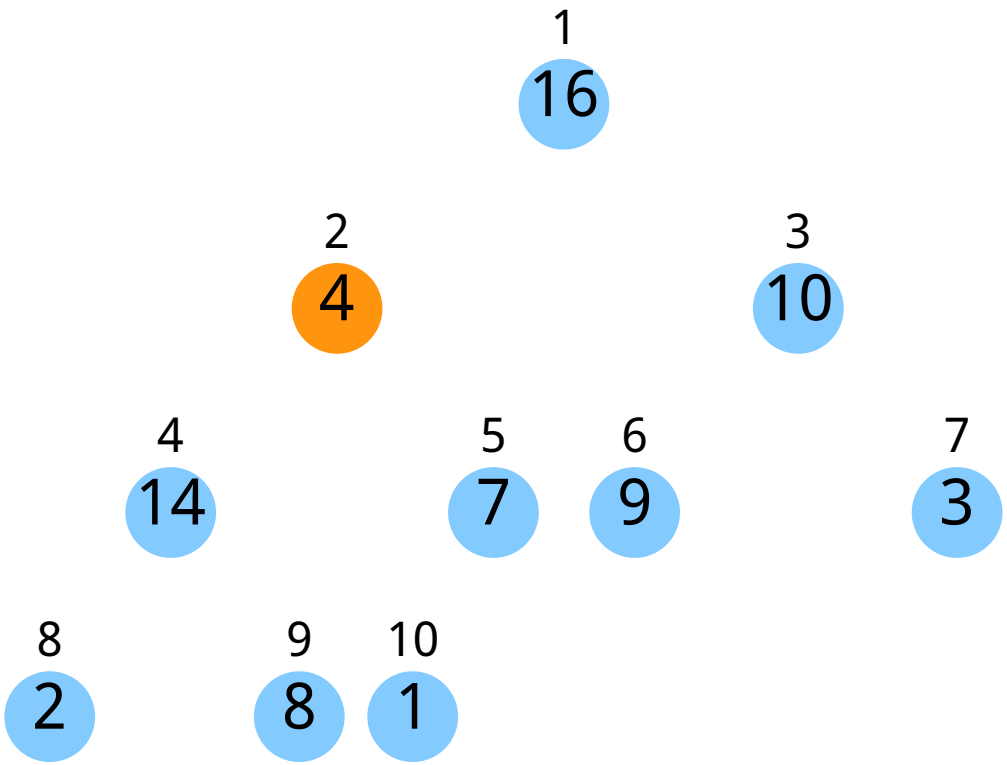
- Per un **max-heap**, A[i] <= A[Parent(i)] per ogni i>1
  - Nella radice c'è il valore più grande
- Per un **min-heap**, A[i] >= A[Parent(i)] per ogni i>1
  - Nella radice c'è il valore più piccolo

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



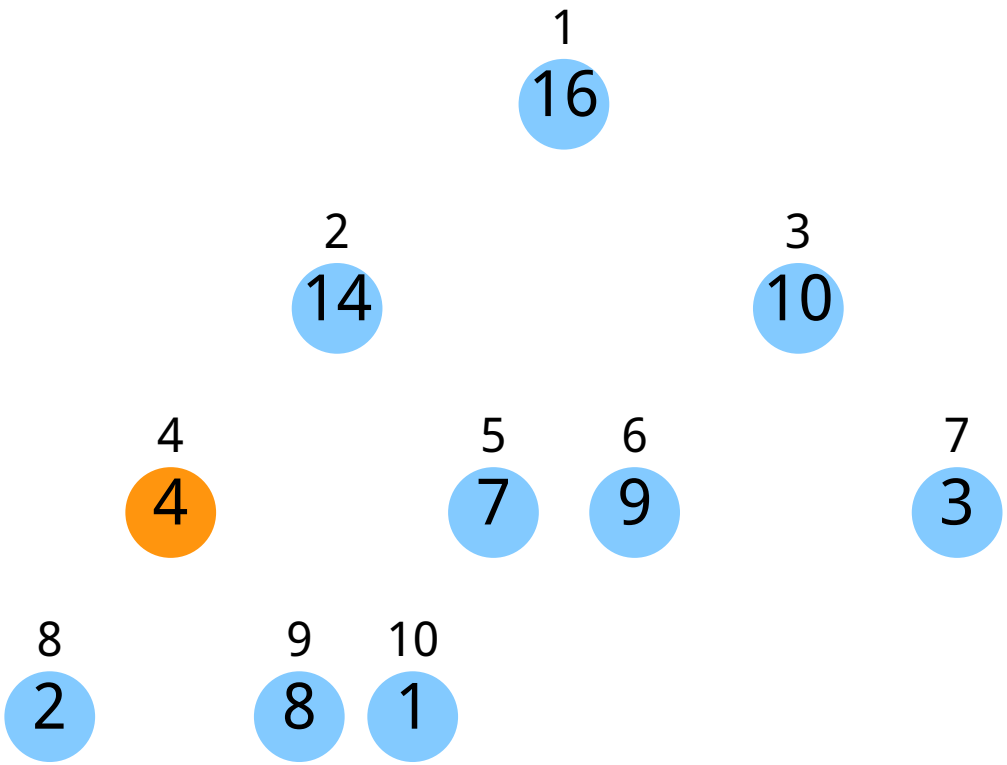
Max-Heapify

- Assumendo che gli alberi aventi radice in Left(i) e Right(i) siano max-heap, vogliamo fare in modo che l'albero con radice in i sia anch'esso un heap



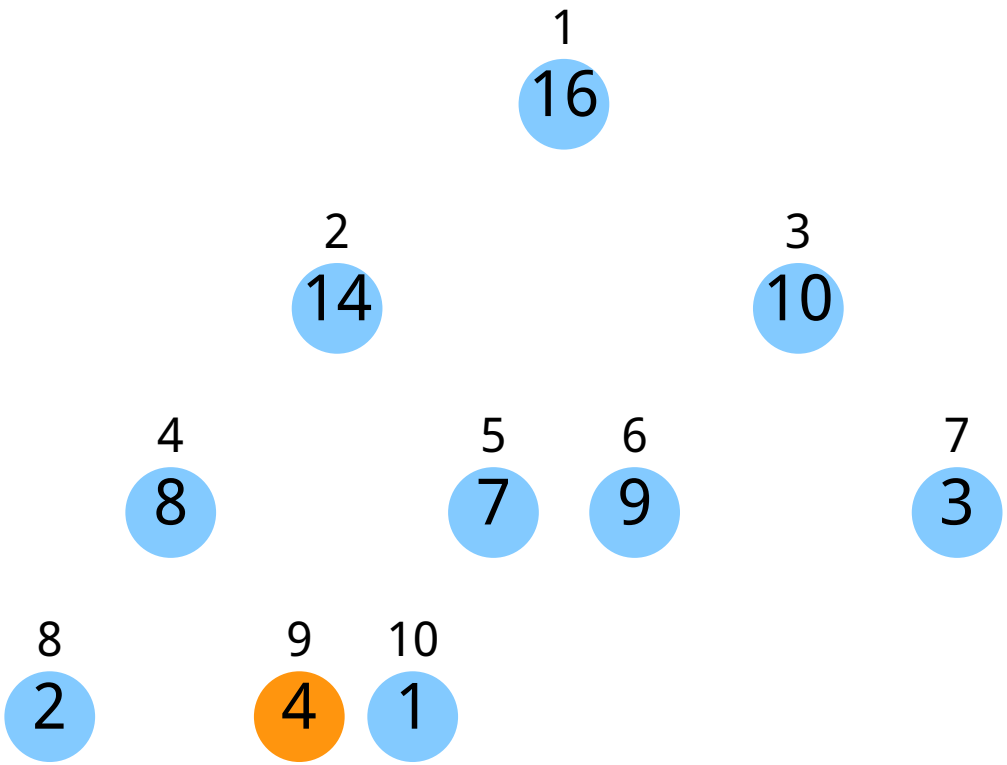
Max-Heapify

- Assumendo che gli alberi aventi radice in Left(i) e Right(i) siano max-heap, vogliamo fare in modo che l'albero con radice in i sia anch'esso un heap

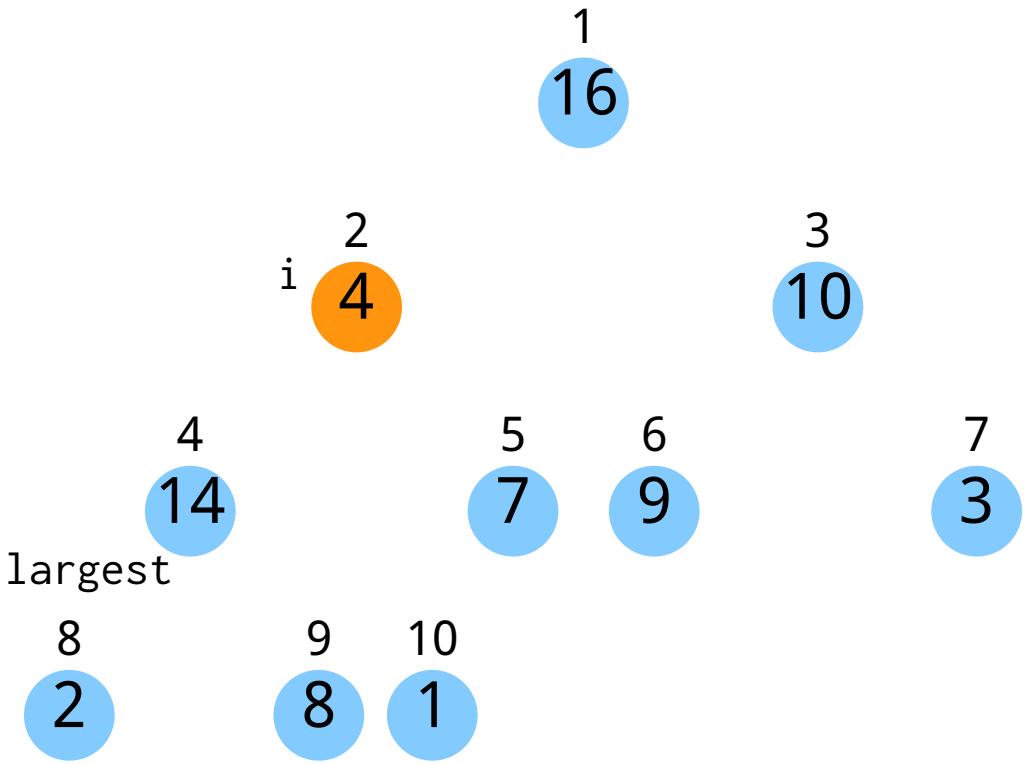


Max-Heapify

- Assumendo che gli alberi aventi radice in Left(i) e Right(i) siano max-heap, vogliamo fare in modo che l'albero con radice in i sia anch'esso un heap



Max-Heapify

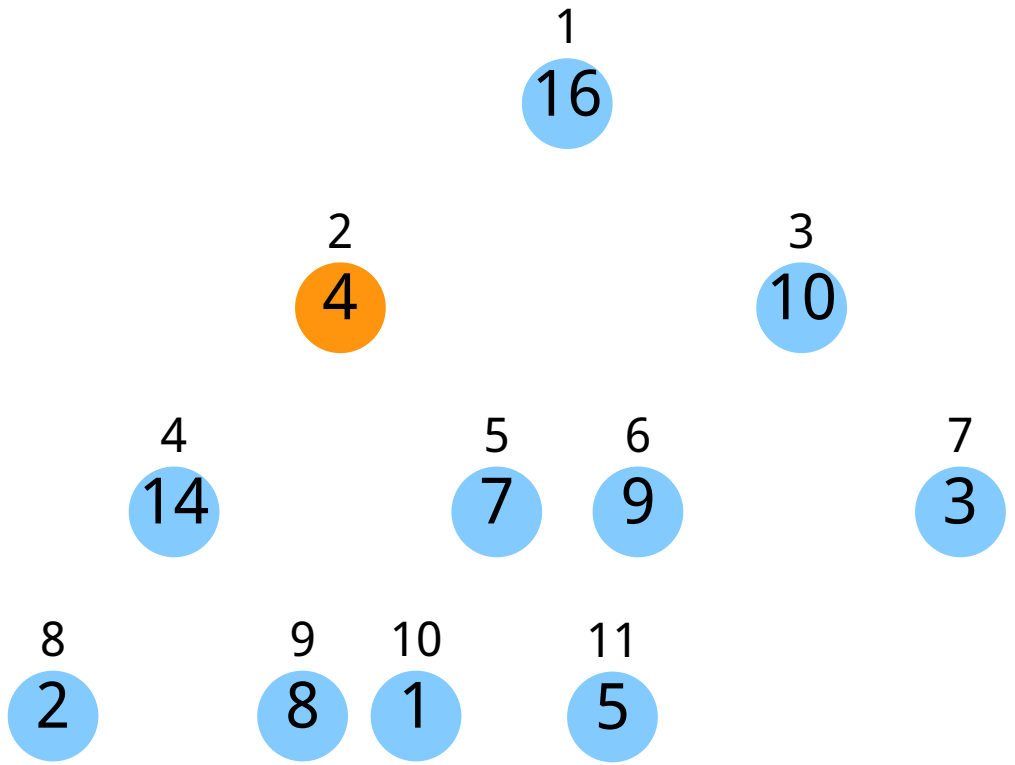



```
Max-Heapify (A,i)
l ← Left(i)
r ← Right(i)
largest ← i
if l≤heap-size[A] and A[l]>A[largest]
    largest ← l
if r ≤heap-size[A] and A[r]>A[largest]
    largest ← r
if largest ≠i
    exchange A[i]⇔A[largest]
    Max-Heapify (A,largest)
```

- Tempo di esecuzione
  - Tempo costante per individuare il massimo fra tre ed effettuare lo scambio
  - Tempo per eseguire Max-Heapify su problema di dimensione minore

Max-Heapify

- Quanto vale la dimensione del sottoproblema da risolvere?
  - Detta *n* la dimensione del problema originario
  - *m* la dimensione del sottoalbero con più elementi
  - *h* l'altezza del nodo oggetto del problema originario



Caso 1: ultimo livello pieno

$$n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$m = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

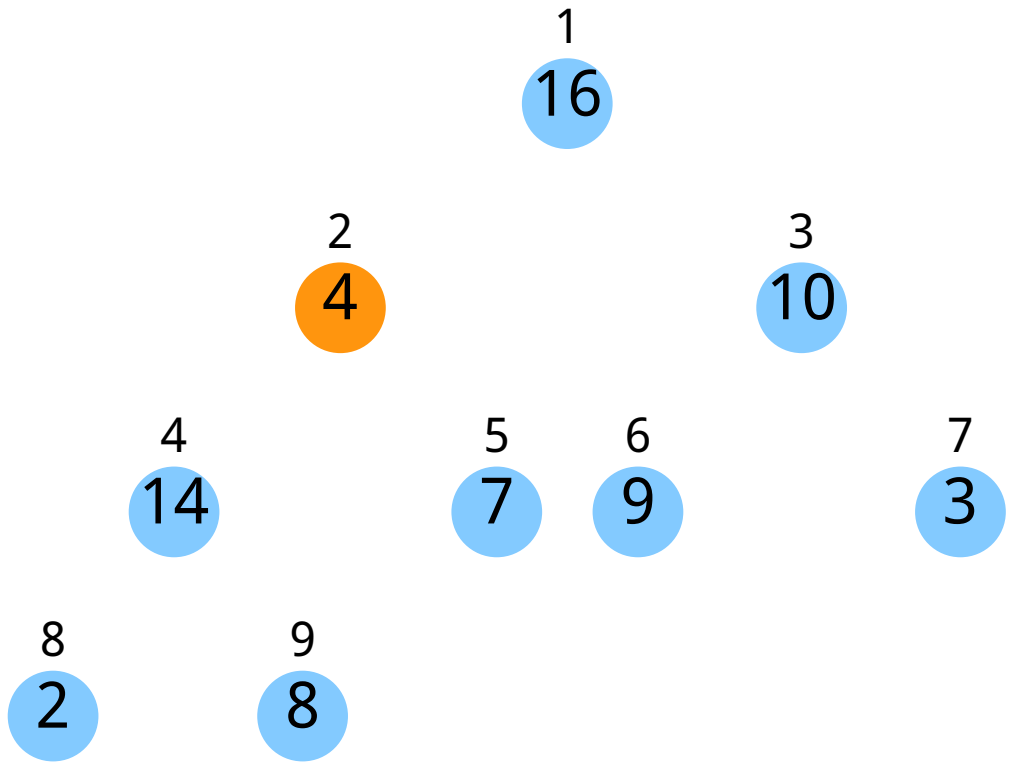
$$\frac{m}{n} = \frac{2^h - 1}{2^{h+1} - 1} = \frac{2^h - 1}{2 \cdot 2^h - 1} \xrightarrow{h \to \infty} \frac{1}{2}$$

O(n/2)

–Non è il caso peggiore

Max-Heapify

- Quanto vale la dimensione del sottoproblema da risolvere?
  - Detta n la dimensione del problema originario
  - m la dimensione del sottoalbero con più elementi
  - h l'altezza del nodo oggetto del problema originario



Caso 2: ultimo livello pieno a metà

$$n = (2^h - 1) + (2^{h-1} - 1) + 1 = 3 \cdot 2^{h-1} - 1$$

$$m = 2^h - 1$$

$$\frac{m}{n} = \frac{2^h - 1}{3 \cdot 2^{h-1} - 1} = \frac{2 \cdot 2^{h-1} - 1}{3 \cdot 2^{h-1} - 1} \xrightarrow{h \to \infty} \frac{2}{3}$$

O(2n/3)

– **È il caso peggiore**

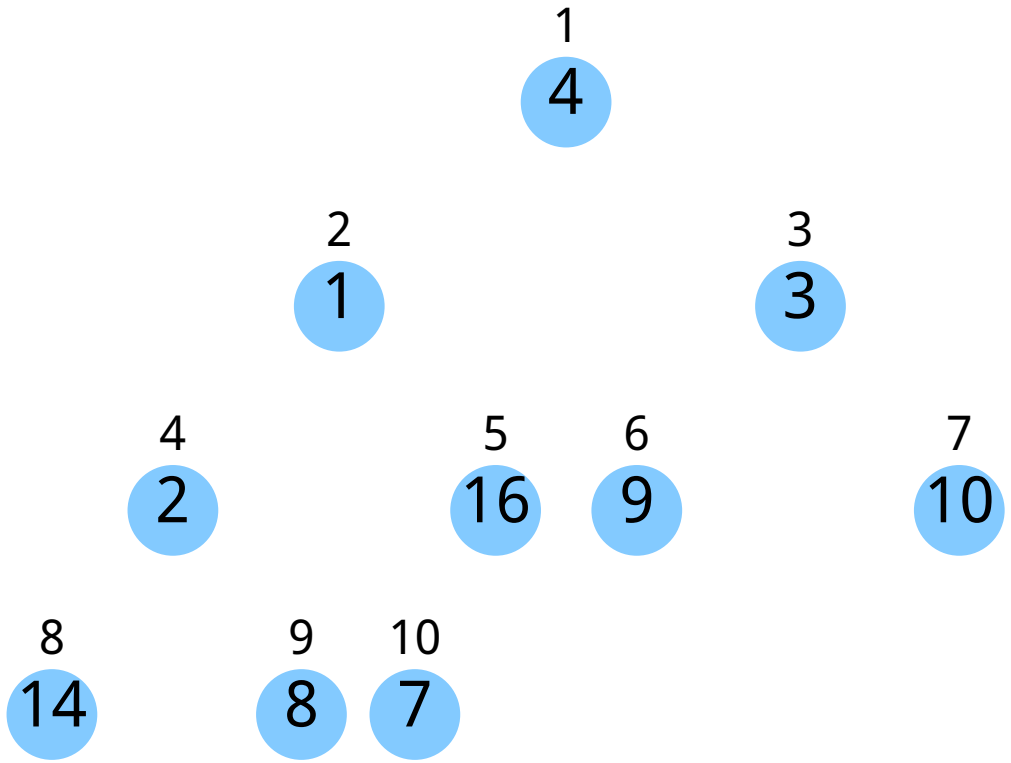
Max-Heapify

- Il tempo di esecuzione è $T(n) \leq T(2n/3) + \Theta(1)$
- Rientra nel caso 2 del teorema dell'esperto
- La complessità di **Max-Heapify è O(lg n)**

- Max-Heapify può essere utilizzato per “convertire” un array in un heap
  - Partendo dal basso e invocando Max-Heapify
  - I nodi foglia si saltano perché costituiscono un heap di 1 elemento

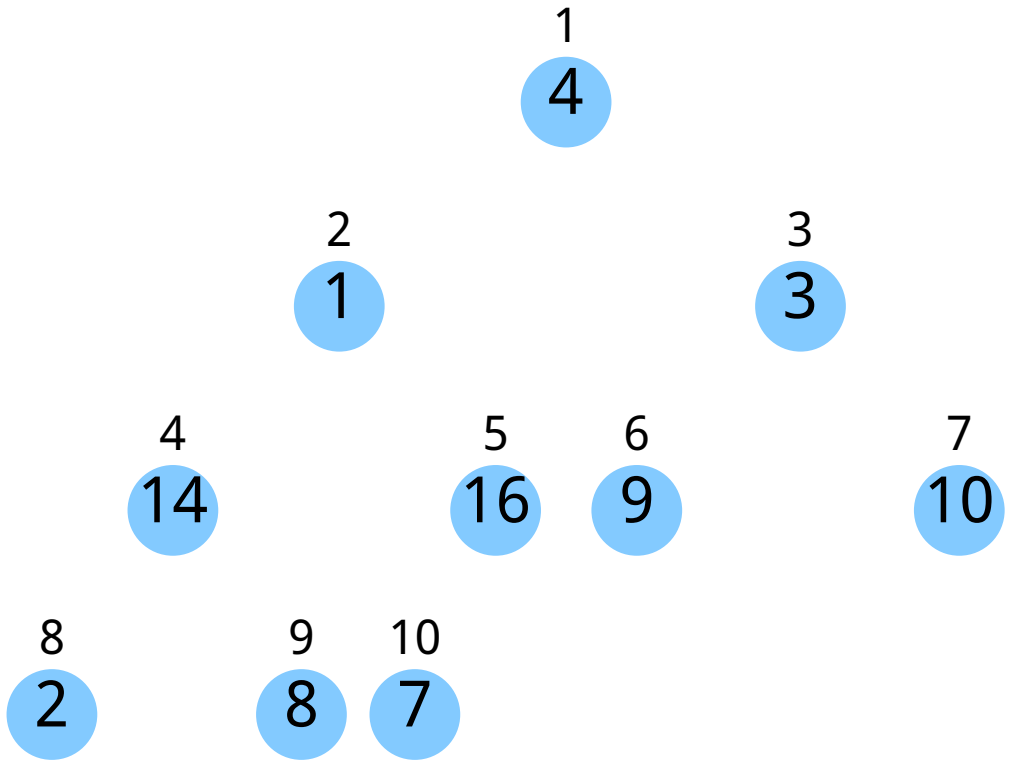
Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



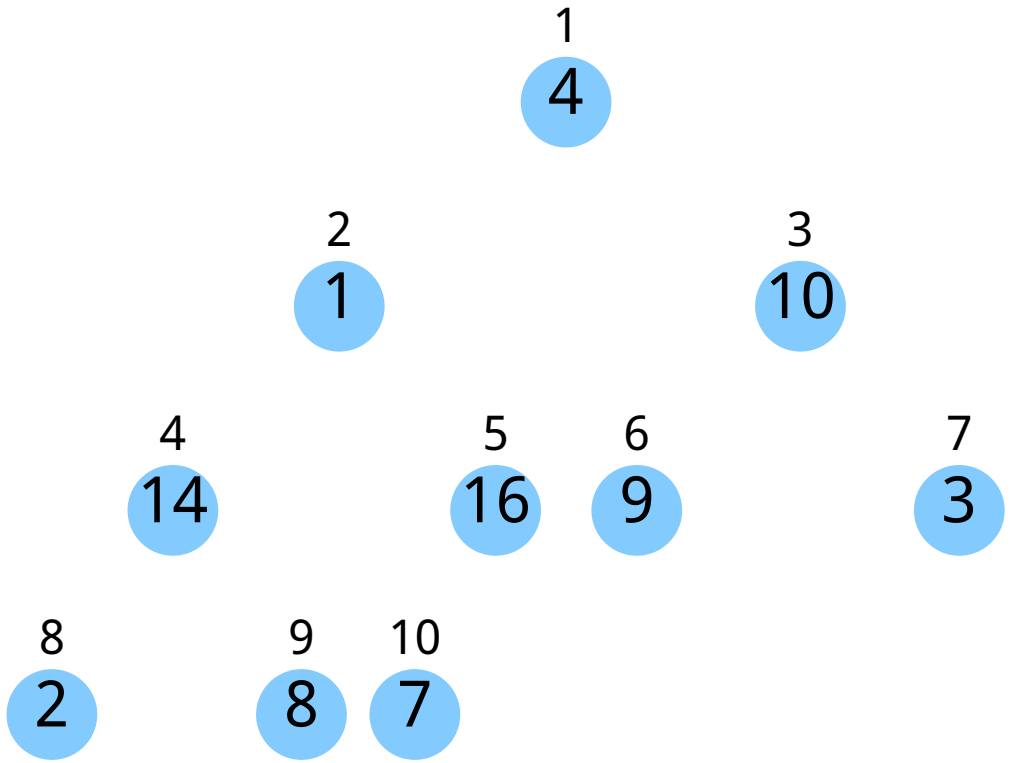
Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
4	1	3	14	16	9	10	2	8	7



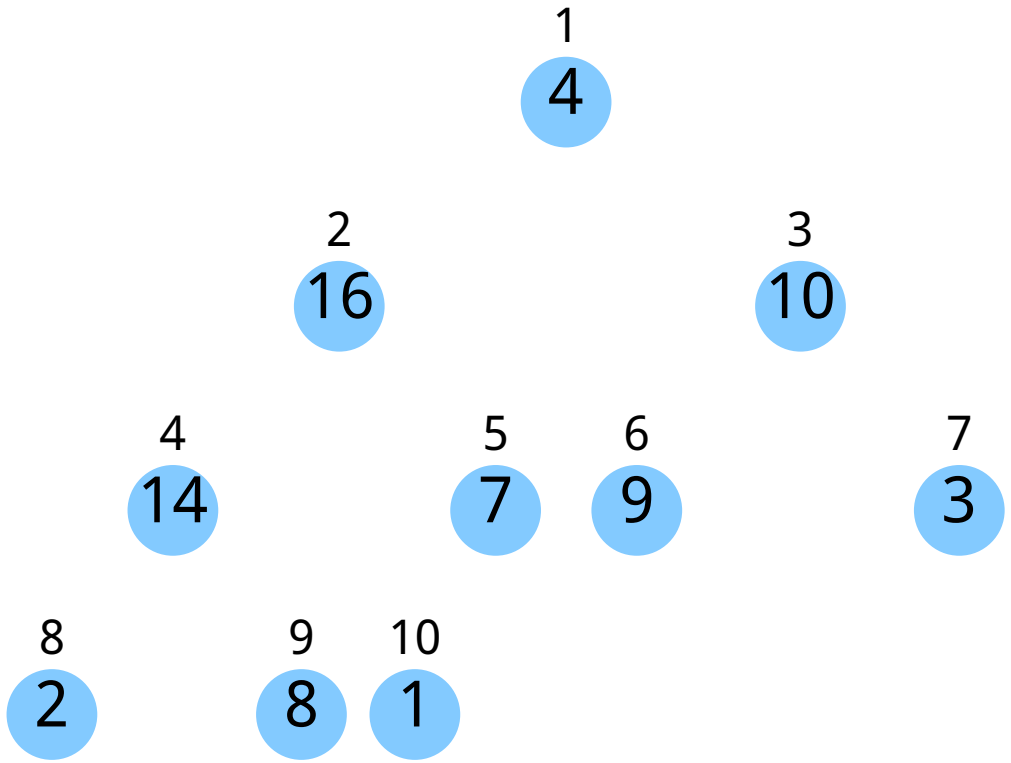
Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
4	1	10	14	16	9	3	2	8	7



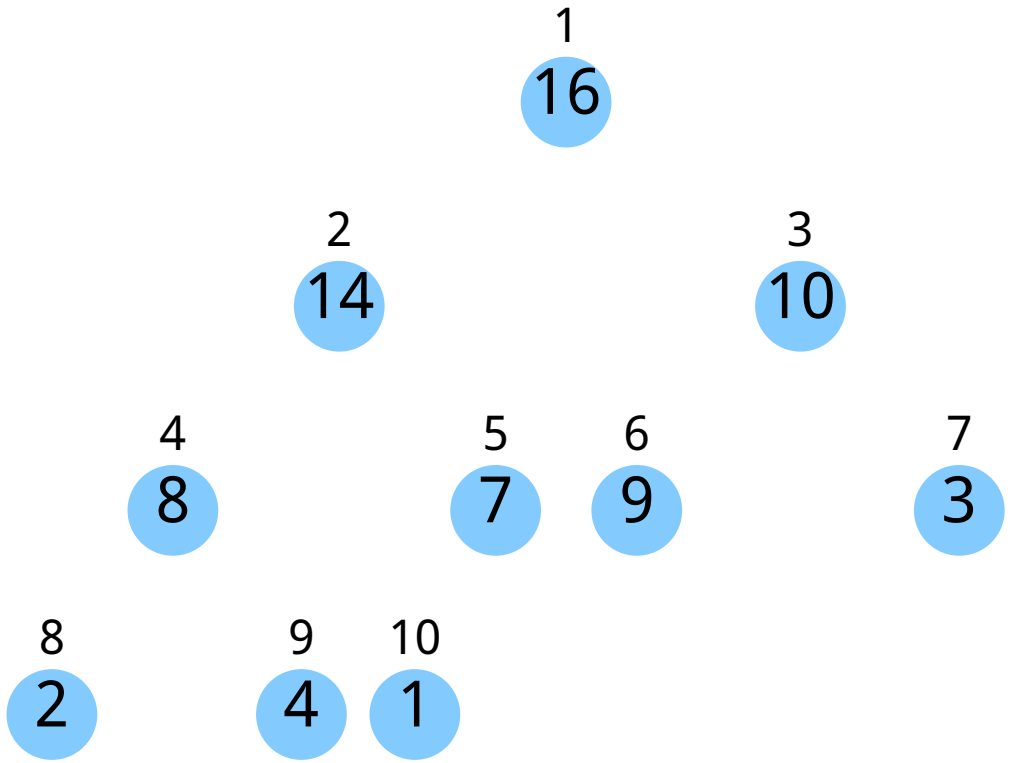
Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
4	16	10	14	7	9	3	2	8	1



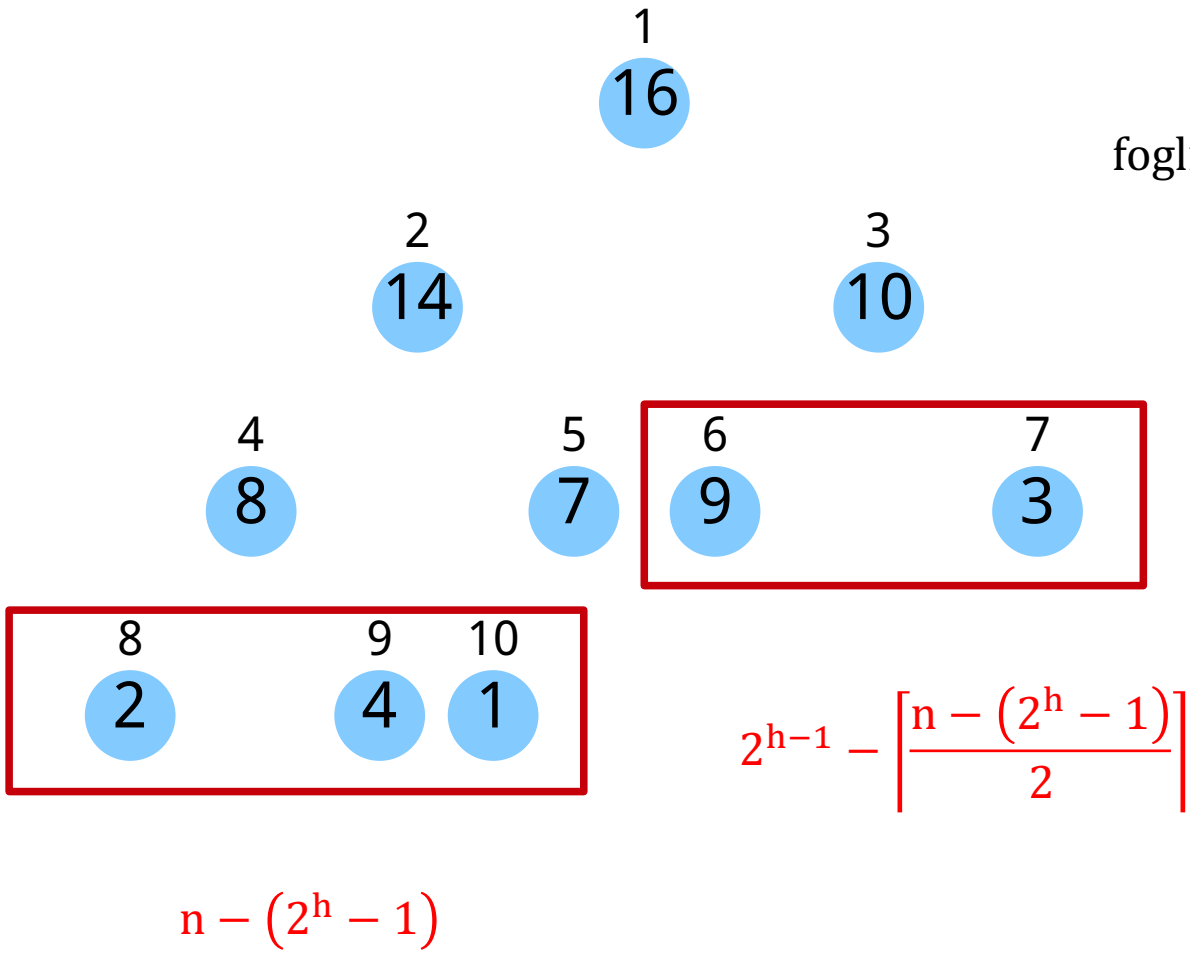
Build-Max-Heap

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Build-Max-Heap

Qual è la posizione nell'array dell'ultimo nodo *non* foglia?



$$\text{foglie} = n - (2^h - 1) + 2^{h-1} - \left\lceil \frac{n - (2^h - 1)}{2} \right\rceil$$

$$= n - 2^{h-1} + 1 - \left\lceil \frac{n+1}{2} \right\rceil + \frac{2^h}{2}$$

$$= n + 1 - \left\lceil \frac{n+1}{2} \right\rceil$$

$$\text{posizione} = n - \text{foglie} = \left\lceil \frac{n+1}{2} \right\rceil - 1 = \left\lfloor \frac{n}{2} \right\rfloor$$

$$2^{h-1} - \left\lceil \frac{n - (2^h - 1)}{2} \right\rceil$$

$$n - (2^h - 1)$$

Build-Max-Heap

```
Build-Max-Heap (A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do Max-Heapify (A,i)
```

- Invariante
  - *All'inizio di ciascuna iterazione del ciclo for, ogni nodo i+1, i+2, … n è la radice di un max-heap*
- Vero prima della prima iterazione
  - i=⌊n/2⌋, i nodi i+1..n sono foglie e quindi max-heap
- L'invariante si conserva attraverso le iterazioni
  - I nodi figli di *i* sono radici di max-heap (per ipotesi), quindi è verificata la pre-condizione di Max-Heapify, che rende anche *i* radice di un max-heap; incrementando *i* si conserva l'invariante
- Al termine, tutti i nodi i>=1 sono radici di max-heap, in particolare i=1 ci dice che il vettore è un max-heap

Build-Max-Heap

```
Build-Max-Heap (A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    Max-Heapify (A,i)
```

- Complessità computazionale
  - O(n) invocazioni di Max-Heapify ⇒ O(n lg n)
  - Non è un limite stretto
- Dato che
  - Un heap di n elementi ha altezza $\lfloor \lg n \rfloor$
  - Un heap di n elementi ha al più $\lceil n/2^{h+1} \rceil$ elementi di altezza h
    - num foglie (altezza 0) è n - $\lfloor n/2 \rfloor = \lceil n/2 \rceil$
    - Num nodi altezza 1 è la metà $\lceil n/2^2 \rceil$
  - Max-Heapify ha complessità O(h) per nodi di altezza h

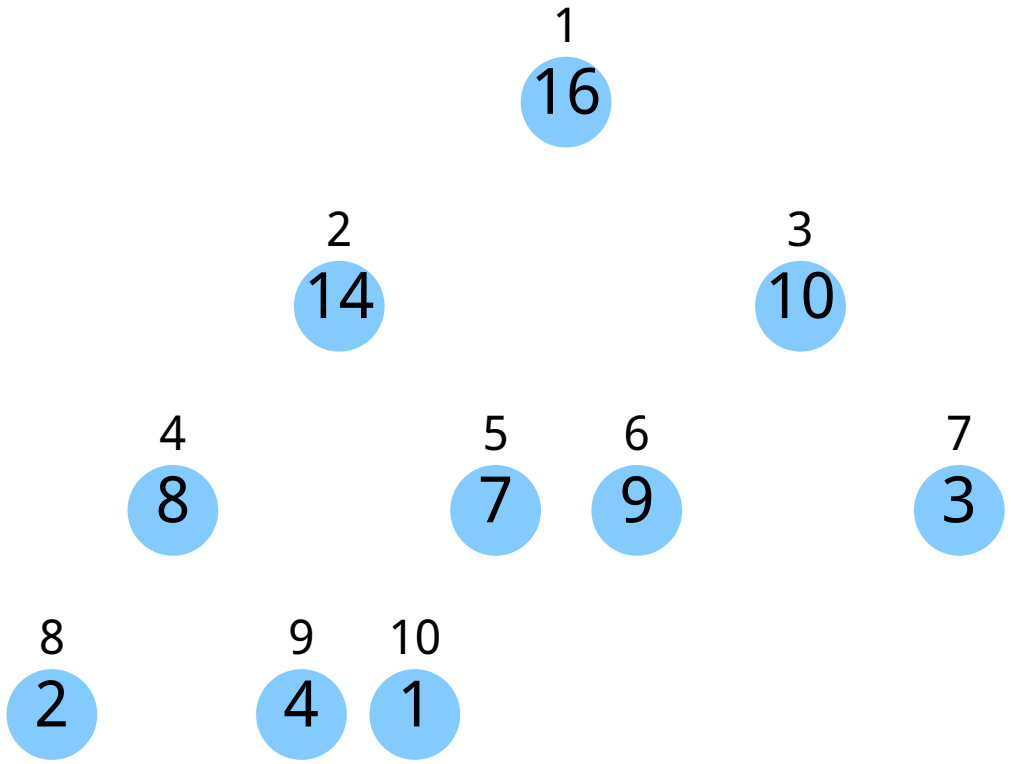
$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)$$

Heap-Sort

- Si rende l'array un max-heap (con Build-Max-Heap)
  - Il primo elemento è l'elemento massimo
- Si scambia il primo elemento con l'ultimo
  - L'elemento massimo è “sistemato”
- Si decrementa la dimensione dell'heap e si invoca Max-Heapify per mantenere la proprietà del max-heap

Heap-Sort

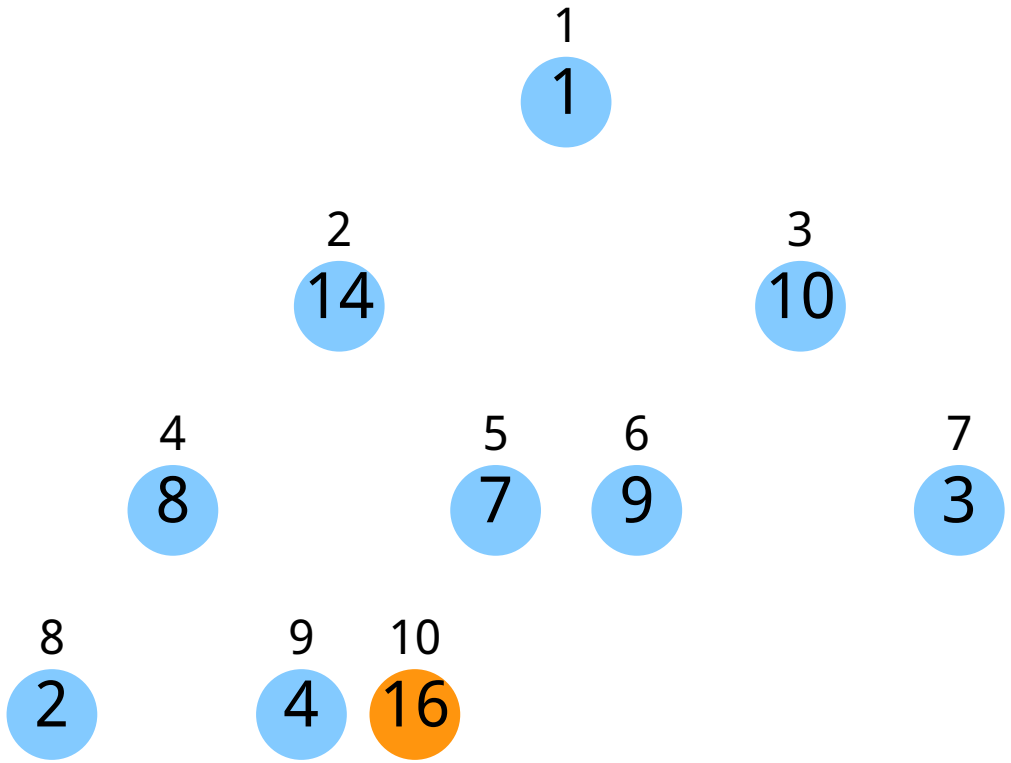
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



- Scambiamo 16 con 1 e decrementiamo la dimensione dell'heap

Heap-Sort

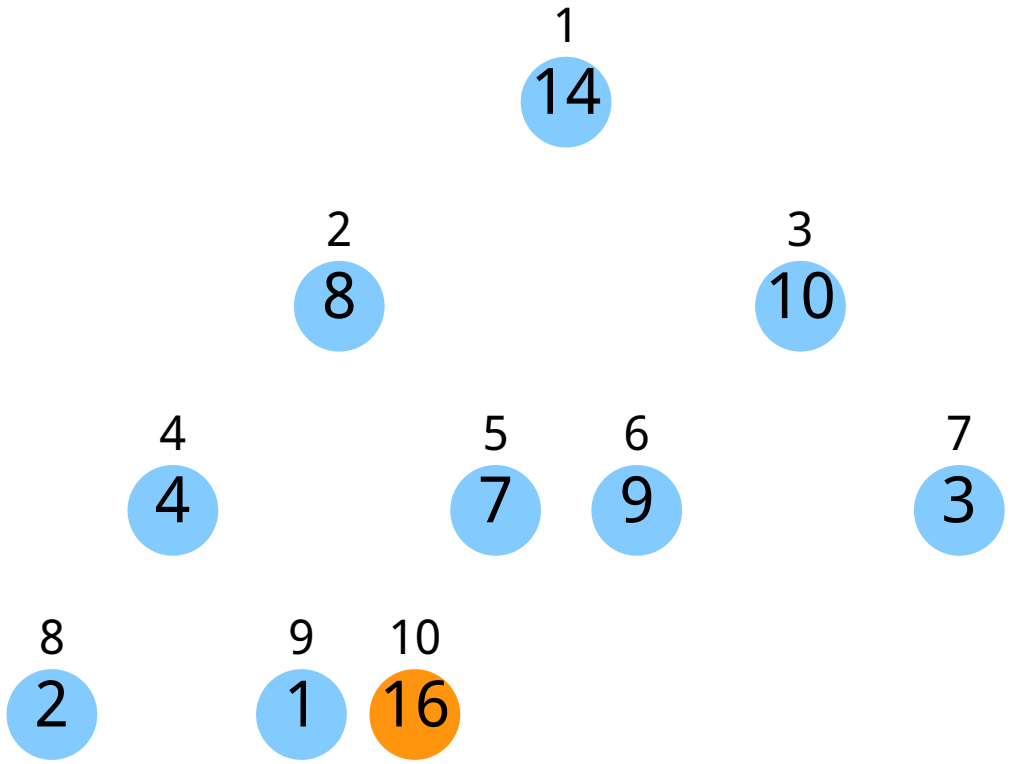
1	2	3	4	5	6	7	8	9	10
1	14	10	8	7	9	3	2	4	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

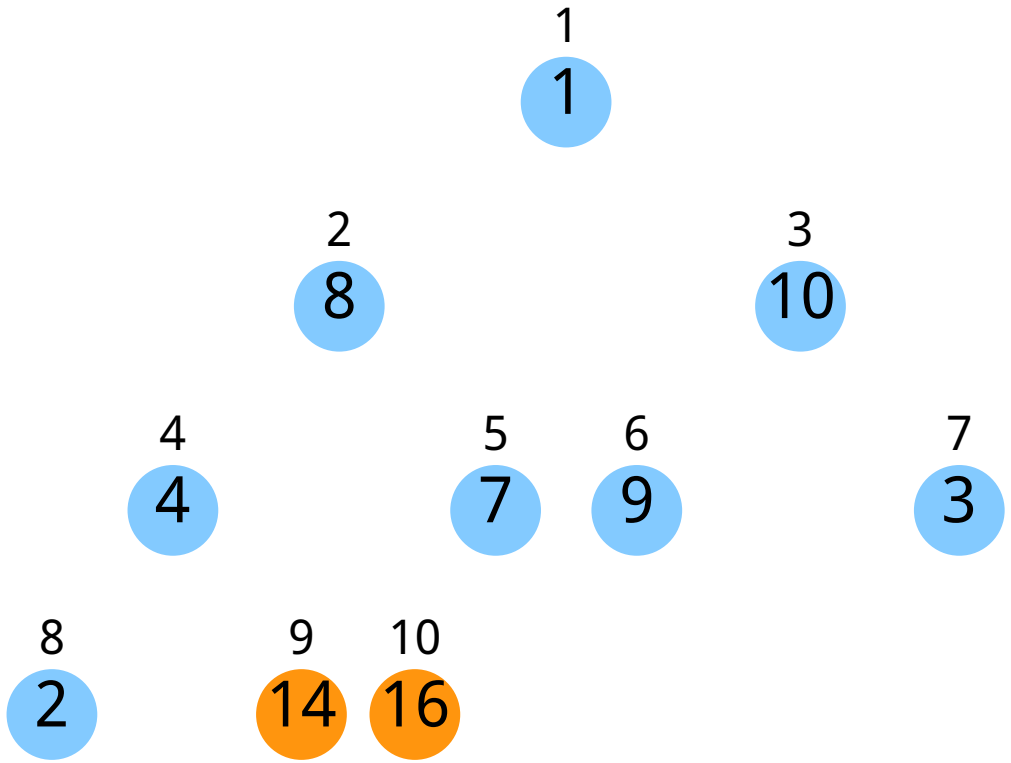
1	2	3	4	5	6	7	8	9	10
14	8	10	4	7	9	3	2	1	16



- Scambiamo 14 con 1 e decrementiamo la dimensione dell'heap

Heap-Sort

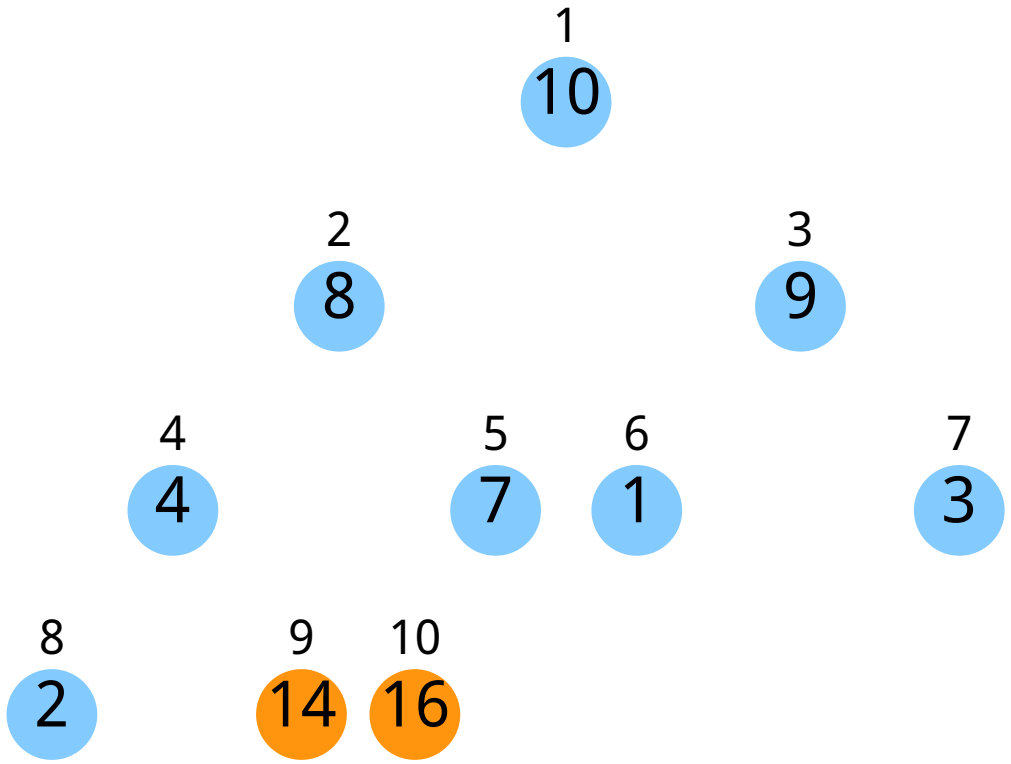
1	2	3	4	5	6	7	8	9	10
1	8	10	4	7	9	3	2	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

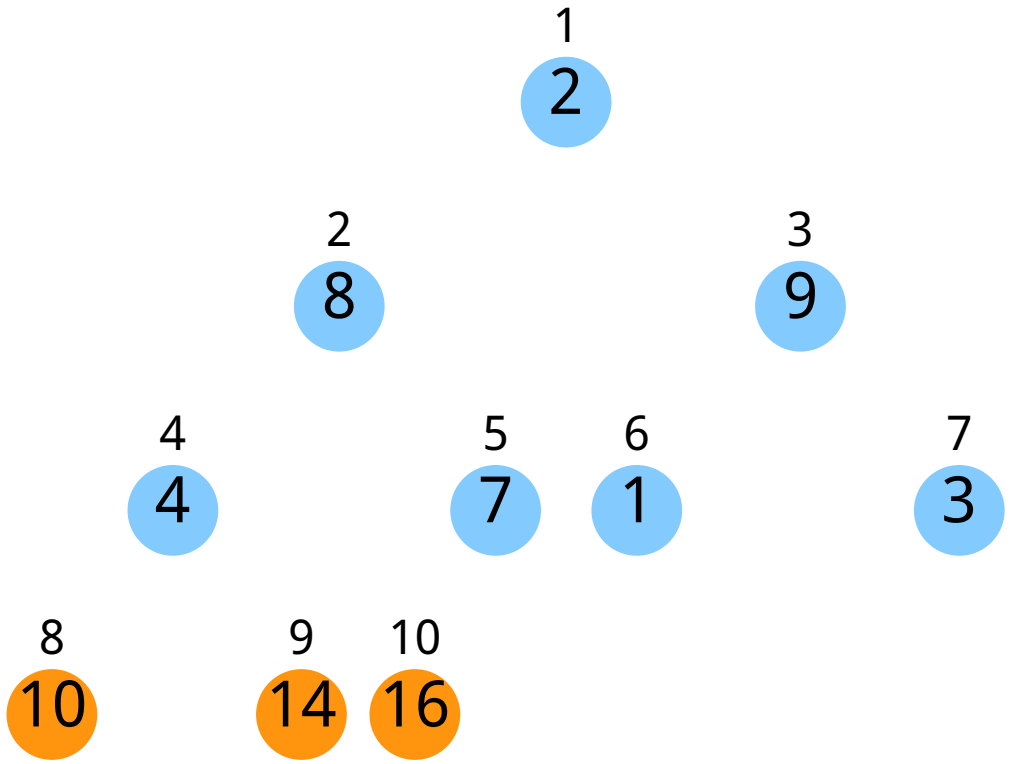
1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16



- Scambiamo 10 con 2 e decrementiamo la dimensione dell'heap

Heap-Sort

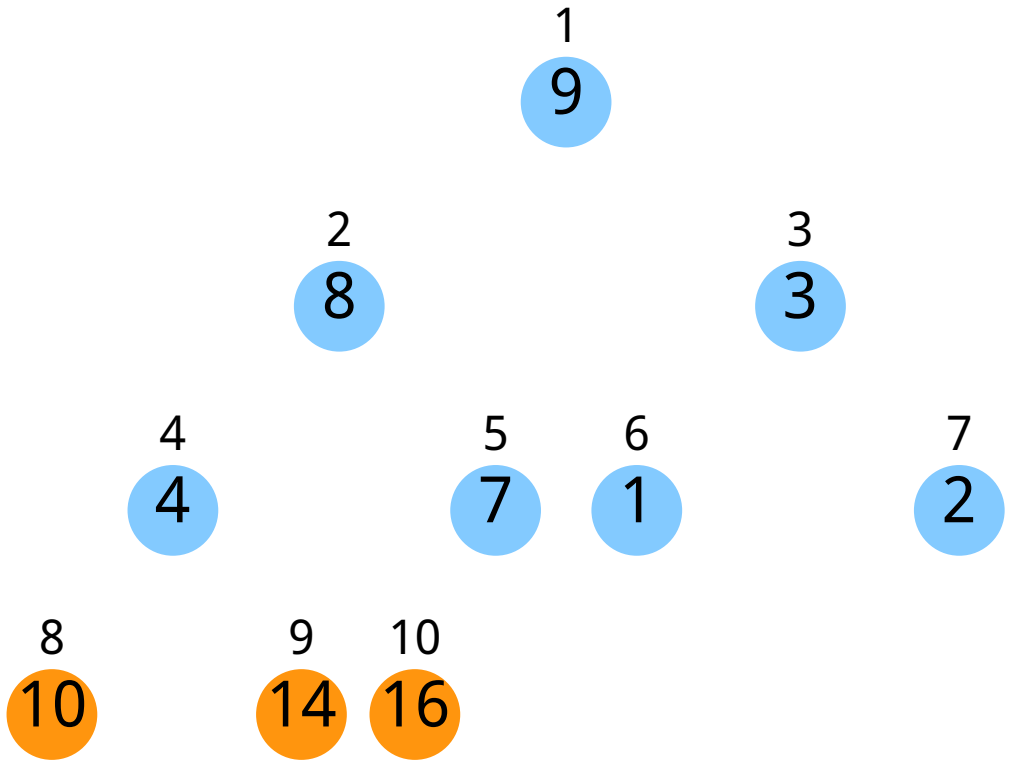
1	2	3	4	5	6	7	8	9	10
2	8	9	4	7	1	3	10	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

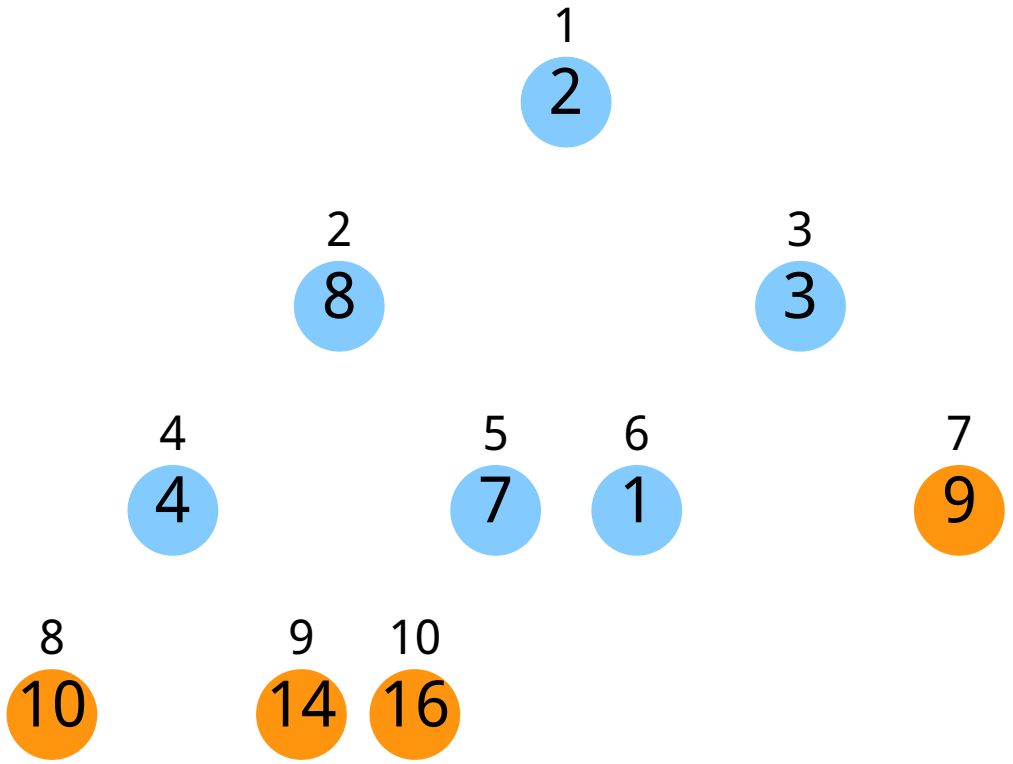
1	2	3	4	5	6	7	8	9	10
9	8	3	4	7	1	2	10	14	16



- Scambiamo 9 con 2 e decrementiamo la dimensione dell'heap

Heap-Sort

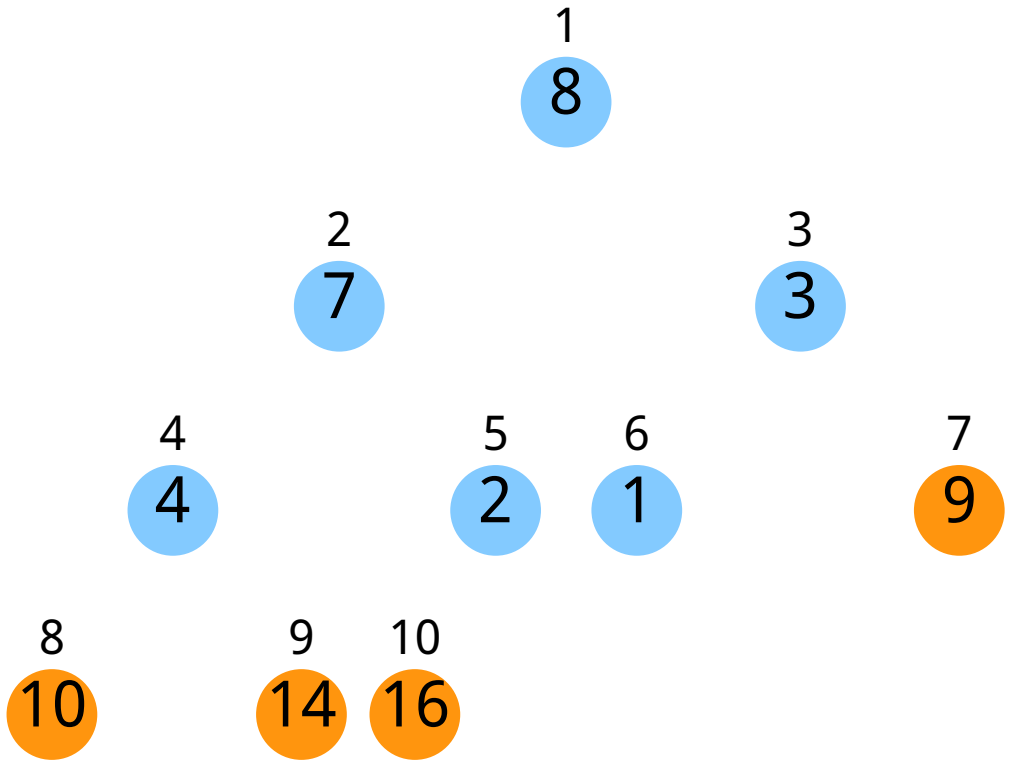
1	2	3	4	5	6	7	8	9	10
2	8	3	4	7	1	9	10	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

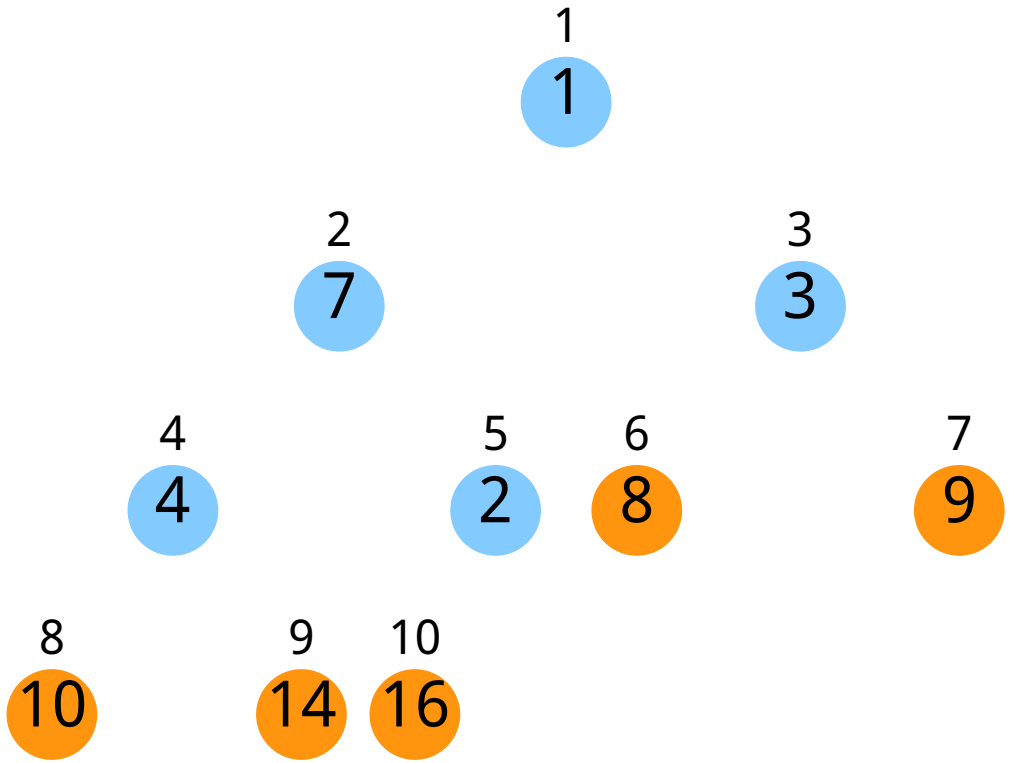
1	2	3	4	5	6	7	8	9	10
8	7	3	4	2	1	9	10	14	16



- Scambiamo 8 con 1 e decrementiamo la dimensione dell'heap

Heap-Sort

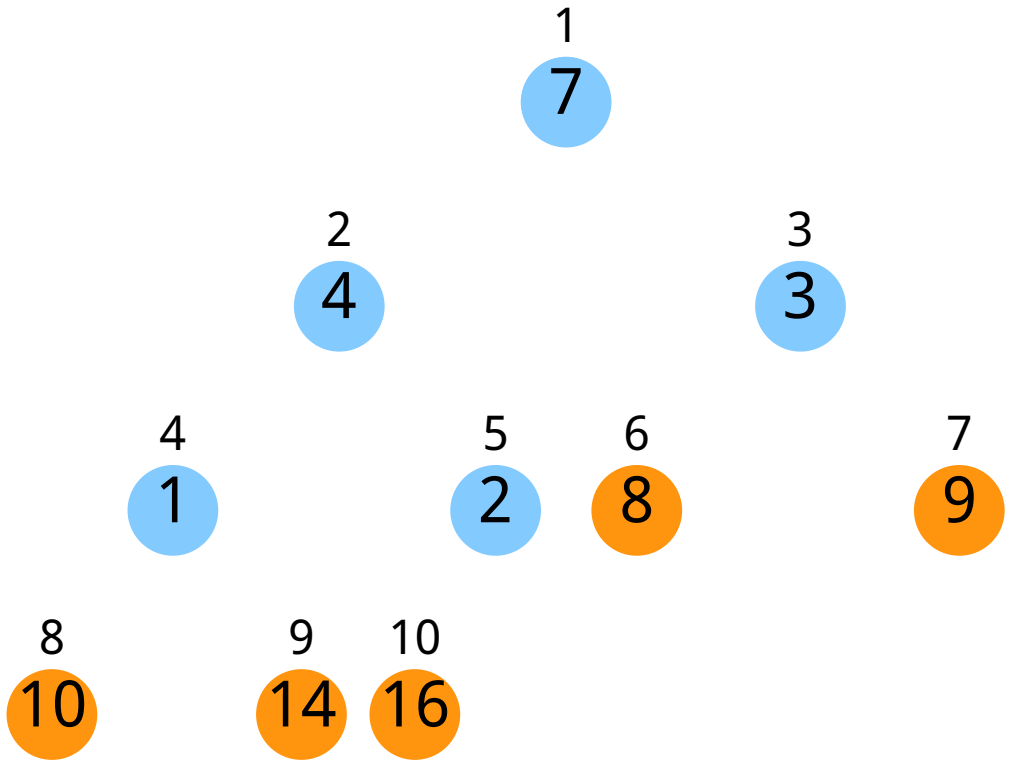
1	2	3	4	5	6	7	8	9	10
1	7	3	4	2	8	9	10	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

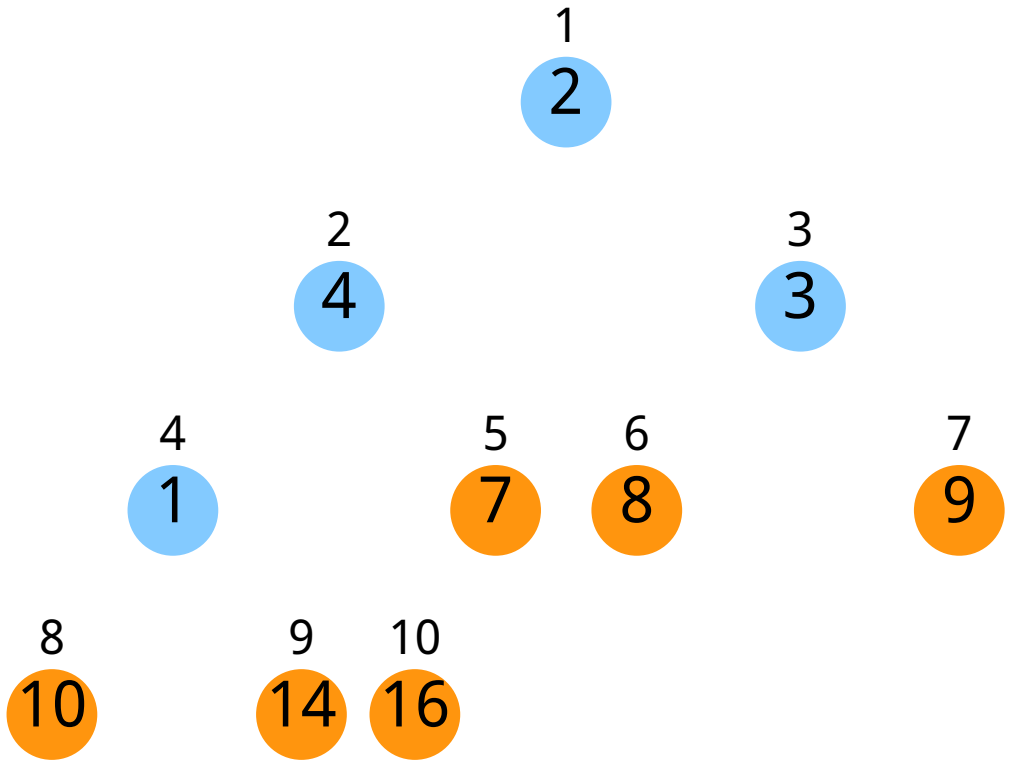
1	2	3	4	5	6	7	8	9	10
7	4	3	1	2	8	9	10	14	16



- Scambiamo 7 con 2 e decrementiamo la dimensione dell'heap

Heap-Sort

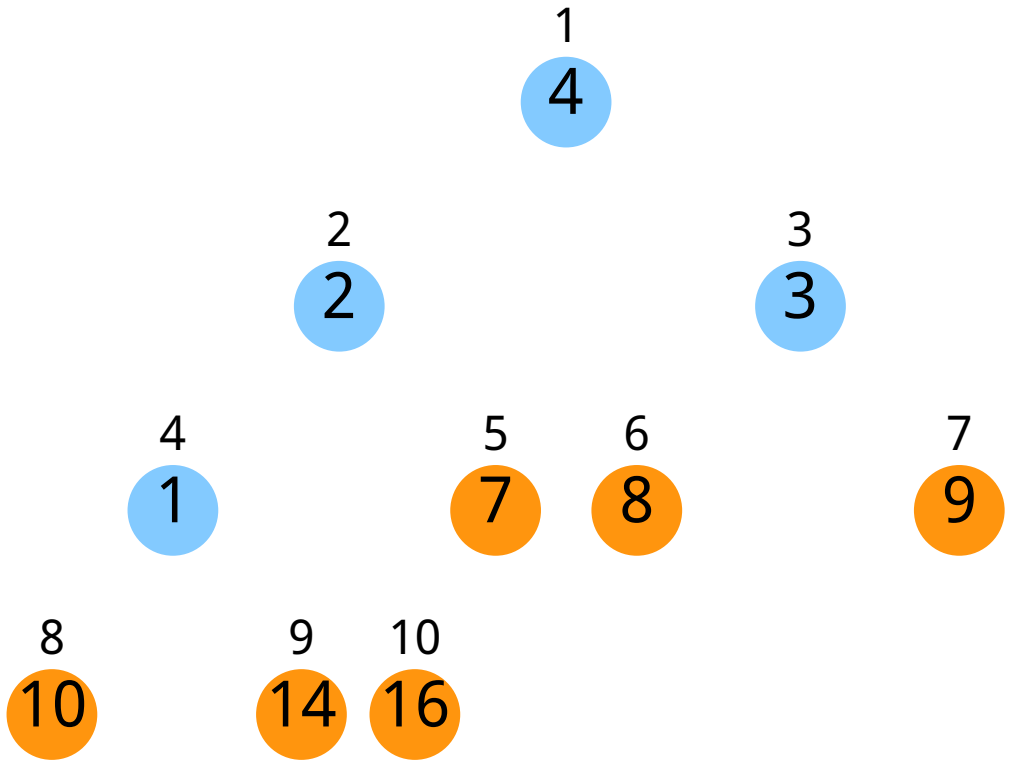
1	2	3	4	5	6	7	8	9	10
2	4	3	1	7	8	9	10	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

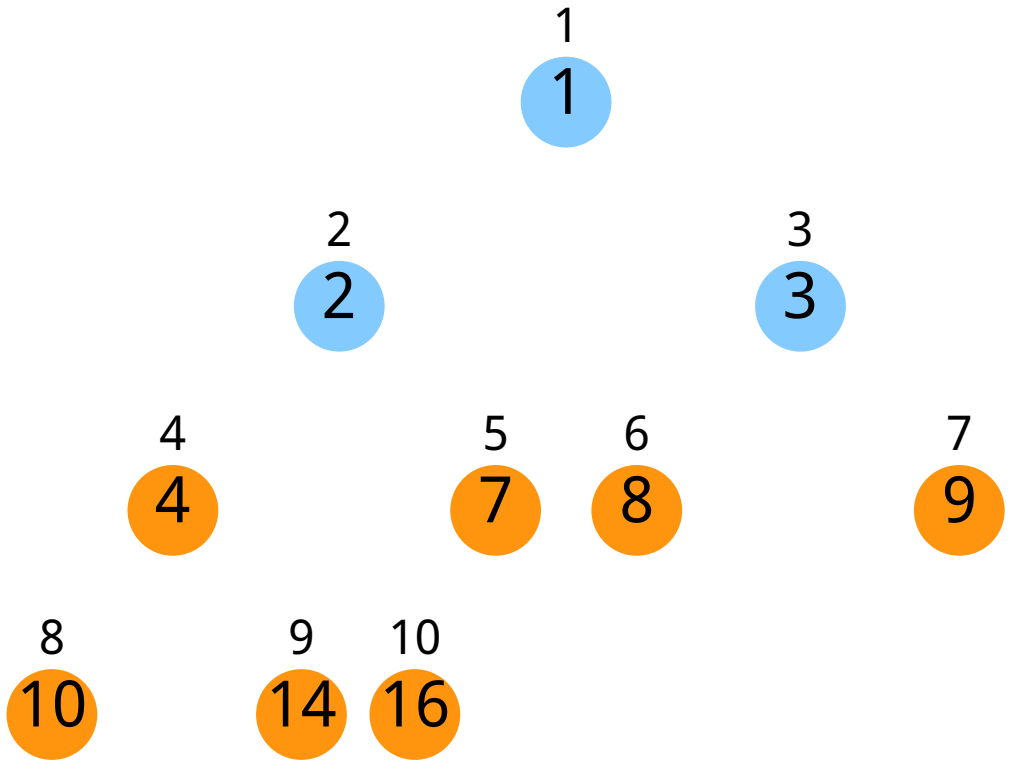
1	2	3	4	5	6	7	8	9	10
4	2	3	1	7	8	9	10	14	16



- Scambiamo 4 con 1 e decrementiamo la dimensione dell'heap

Heap-Sort

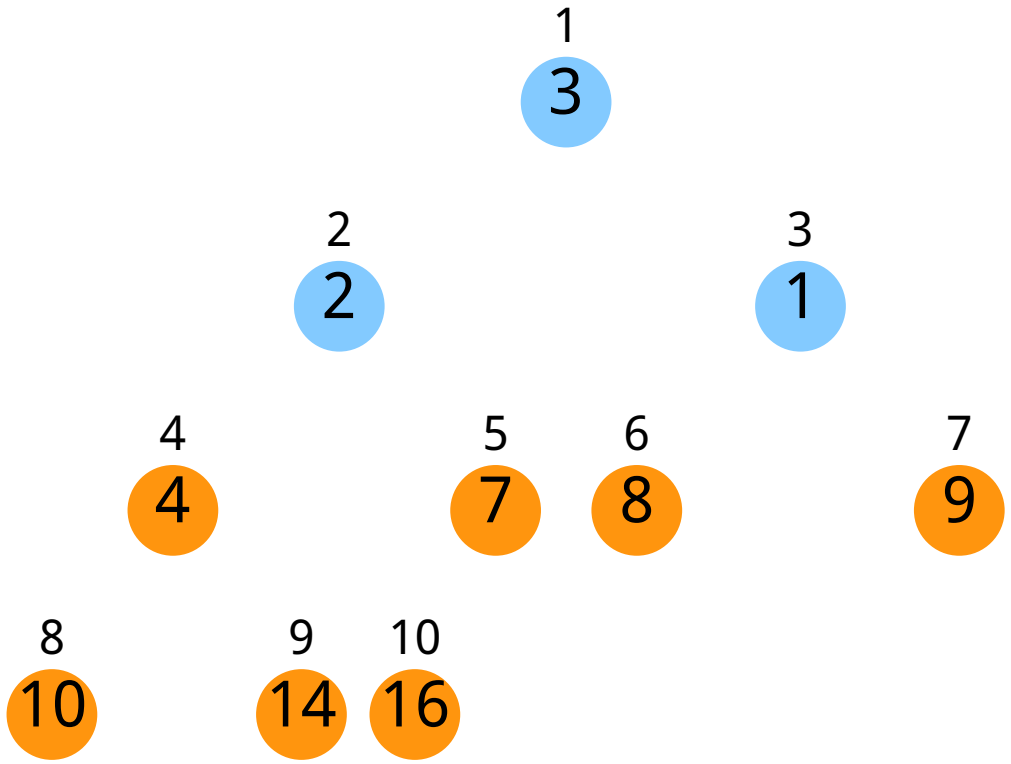
1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

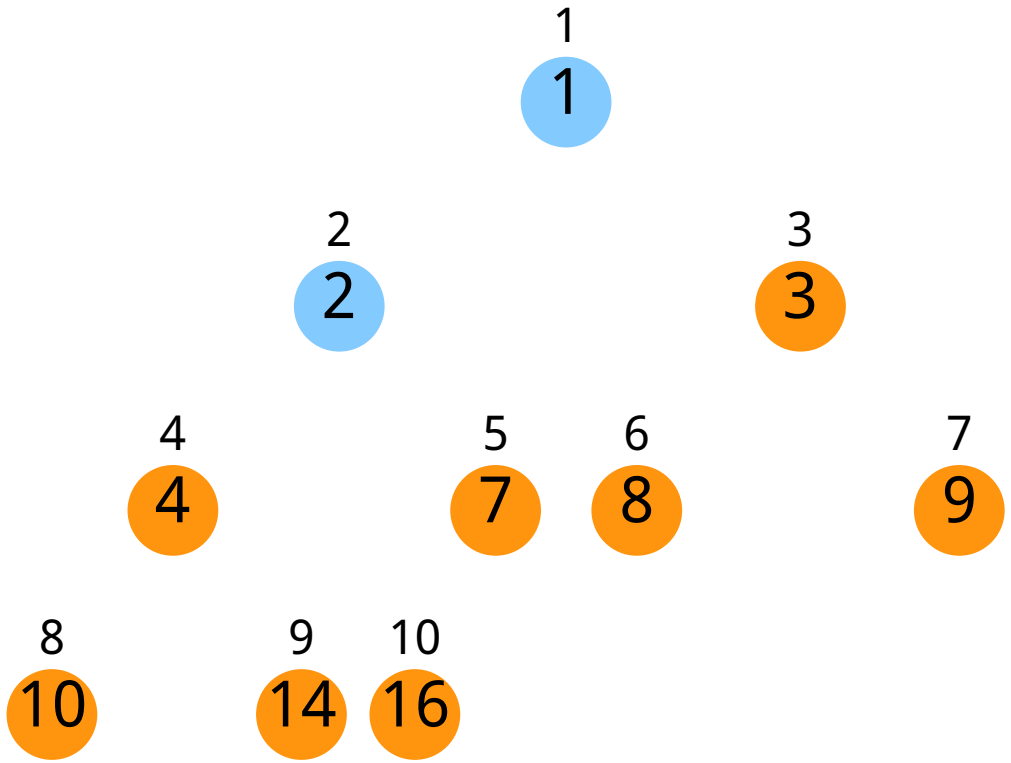
1	2	3	4	5	6	7	8	9	10
3	2	1	4	7	8	9	10	14	16



- Scambiamo 3 con 1 e decrementiamo la dimensione dell'heap

Heap-Sort

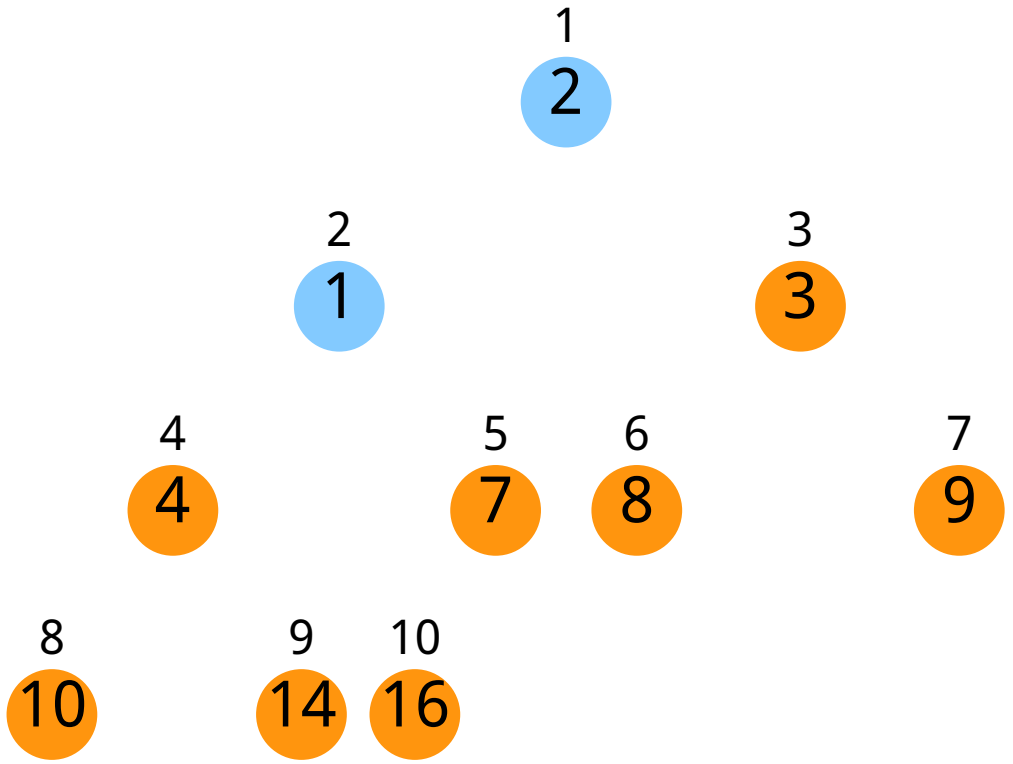
1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



- Invochiamo Max-Heapify su A[1]

Heap-Sort

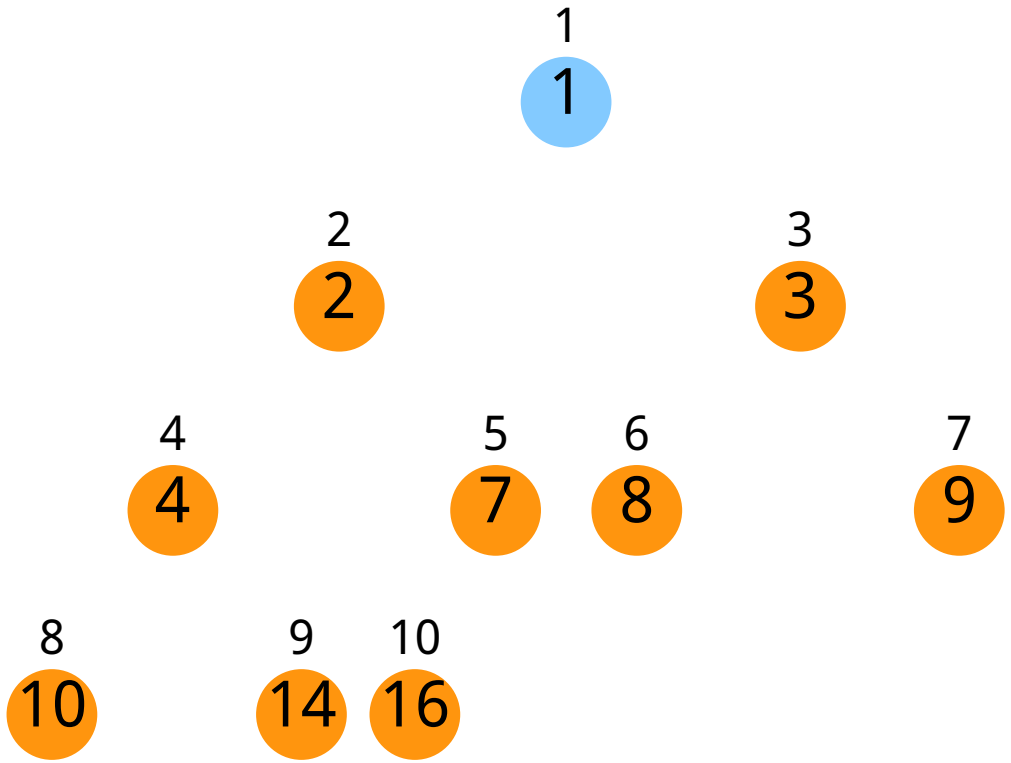
1	2	3	4	5	6	7	8	9	10
2	1	3	4	7	8	9	10	14	16



- Scambiamo 2 con 1 e decrementiamo la dimensione dell'heap

Heap-Sort

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



- Essendo rimasto un solo elemento, l'algoritmo termina

Heap-Sort

```
Heap-Sort (A)
Build-Max-Heap(A)
for i ← length[A] downto 2
    exchange A[1]↔A[i]
    heap-size[A] ← heap-size[A]-1
    Max -Heapify (A,1)
```

- Tempo di esecuzione è O(nlg n)
  - Build-Max-Heap impiega O(n)
  - n-1 invocazioni di Max-Heapify, ciascuna impiega O(lg n)

Rispetto al merge sort lui ordina sul posto !

Code a priorità

- Un max-heap (min-heap) può essere usato per realizzare in maniera efficiente una coda a massima (minima) priorità
- Una **coda a massima (minima) priorità** è una struttura dati che mantiene un insieme S di elementi e supporta le operazioni:
  - **Insert (S, x)**
- Inserisce l'elemento x in S
  - **Maximum (S) (Minimum (S))**
- Restituisce l'elemento in S con la chiave più grande (piccola)
  - **Extract-Max (S) (Extract-Min(S))**
- Rimuove e restituisce l'elemento con la chiave più grande (piccola)
  - **Increase-Key (S, x, k) (Decrease-Key (S, x, k))**
- Incrementa (decrementa) la chiave di x al valore k

Heap-Maximum

```
Heap-Maximum (A)
return A[1]
```

- Tempo esecuzione $\Theta(1)$

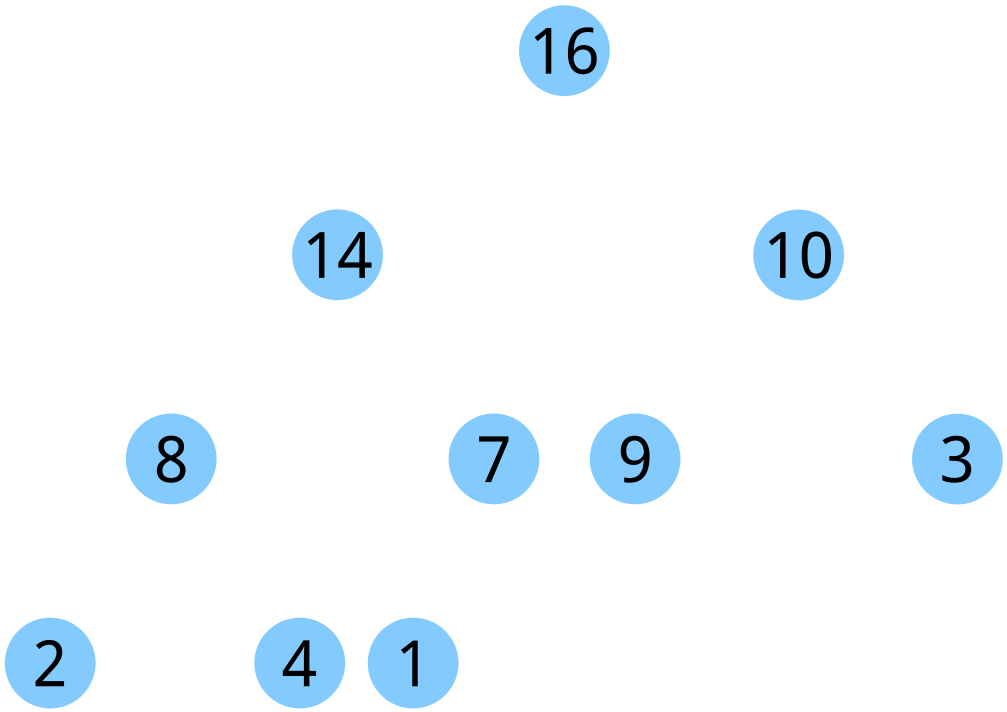
Heap-Extract-Max

```
Heap-Extract-Max (A)
if heap-size[A]<1
    error “heap empty”
max ← A[1]
A[1] ← A[heap-size[A]]
heap-size[A] ← heap-size[A] - 1
Max-Heapify (A,1)
return max
```

- L'ultimo elemento viene messo al posto della radice e viene invocata Max-Heapify per ristabilire la proprietà del max-heap
- Tempo esecuzione O(lg n)
- Una sola invocazione di Max-Heapify

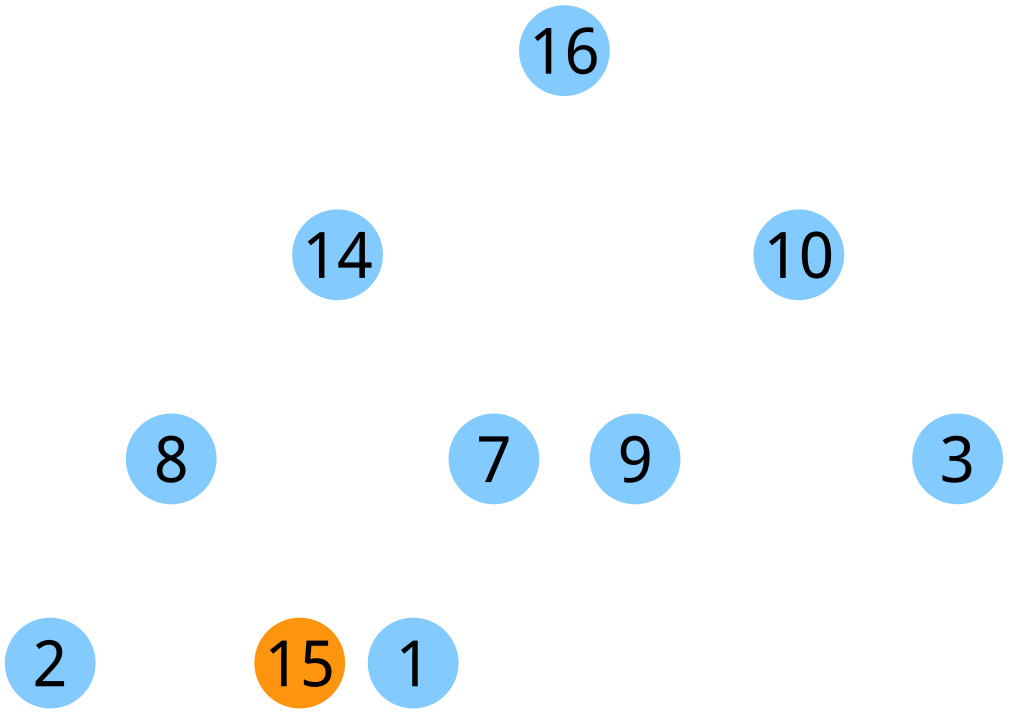
Heap-Increase-Key

- Si fa risalire l'elemento di cui si cambia la chiave in direzione della radice fino a trovare la giusta collocazione



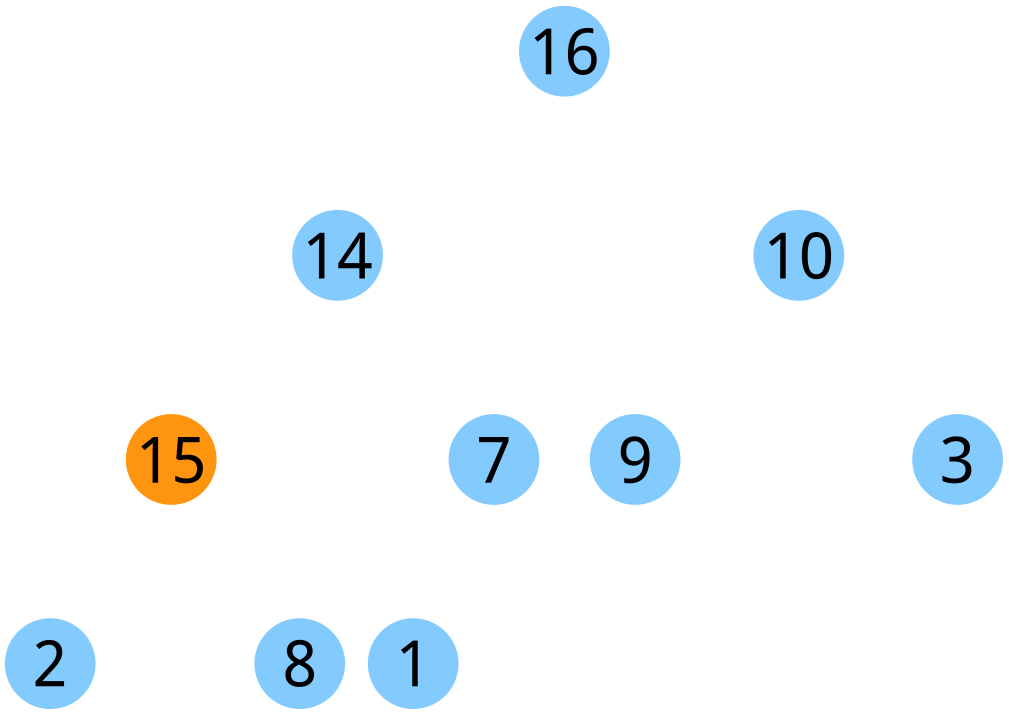
Heap-Increase-Key

- Si fa risalire l'elemento di cui si cambia la chiave in direzione della radice fino a trovare la giusta collocazione



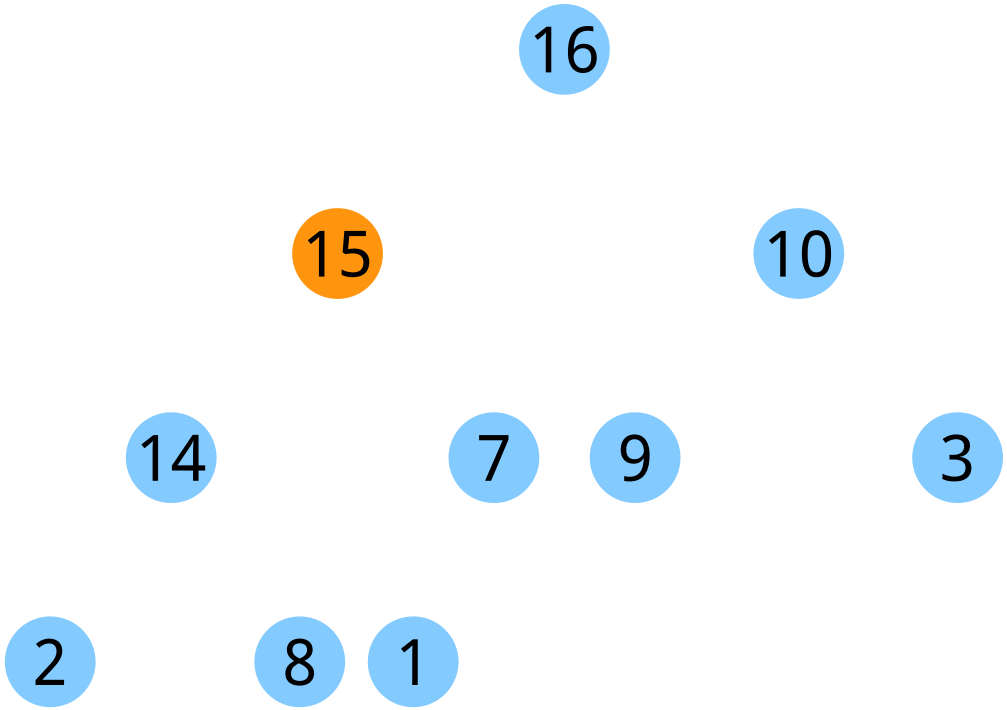
Heap-Increase-Key

- Si fa risalire l'elemento di cui si cambia la chiave in direzione della radice fino a trovare la giusta collocazione



Heap-Increase-Key

- Si fa risalire l'elemento di cui si cambia la chiave in direzione della radice fino a trovare la giusta collocazione




```
Heap-Increase-Key (A,i,key)
if key<A[i]
     error “new key is smaller”
A[i] ← key
while i>1 and A[Parent(i)]<A[i]
     exchange A[i]↔A[Parent(i)]
    i ← Parent(i)
```

- Tempo di esecuzione O(lg n)
  - O(lg n) scambi

Max-Heap-Insert

```
Max-Heap-Insert (A,key)
heap-size[A] ← heap-size[A] + 1
A[heap-size[A]] ← -∞
Heap-Increase-Key (A,heap-size[A],key)
```

- Il nuovo elemento viene messo dopo l'ultimo elemento con un valore -∞
- Viene invocata Heap-Increase-Key per impostare la chiave
- Tempo esecuzione O(lg n)
  - Una sola invocazione di Heap-Increase-Key