In questa lezione

- **Heap binario**
- **heapsort**

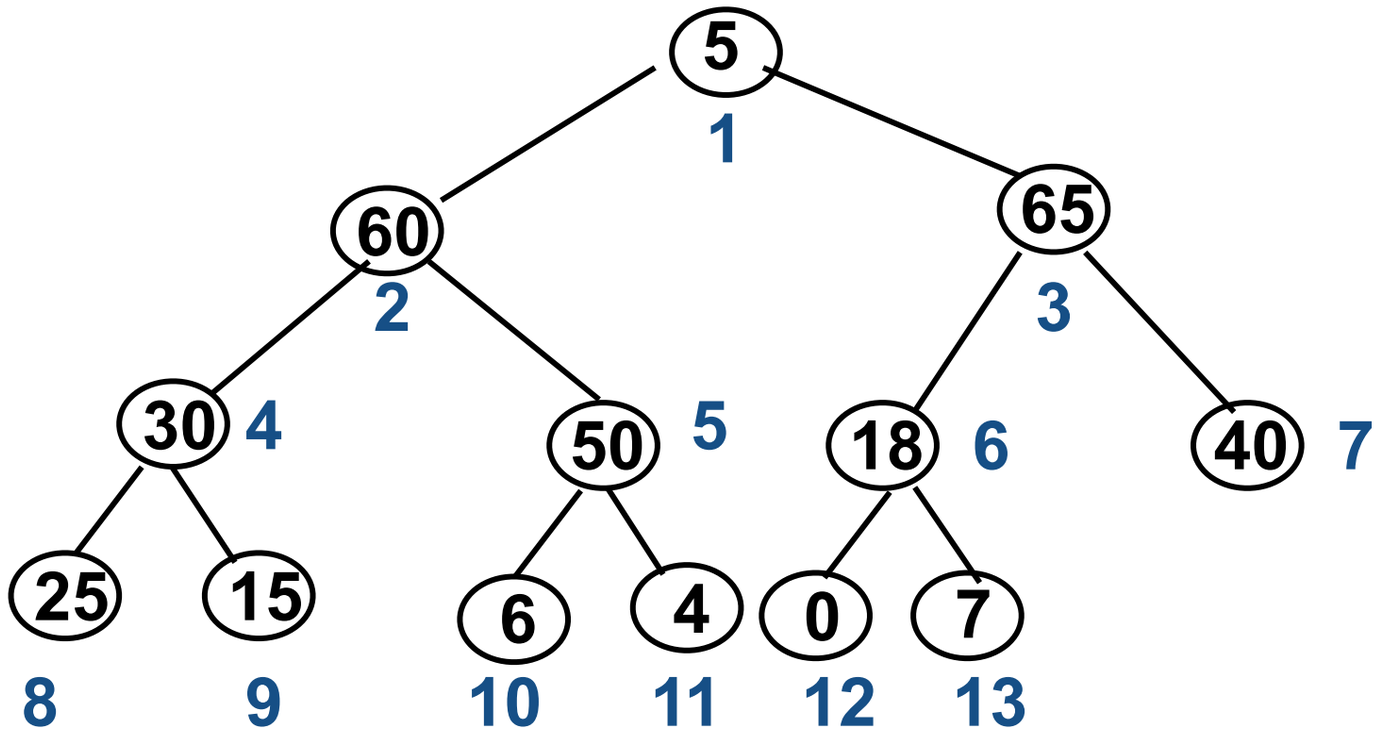
[CLRS10] cap. 6, par. 6.1 - 6.4

Heap binari

Un heap binario è una struttura dati consistente di un array visto come un albero binario.

A=

5	60	65	30	50	18	40	25	15	6	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

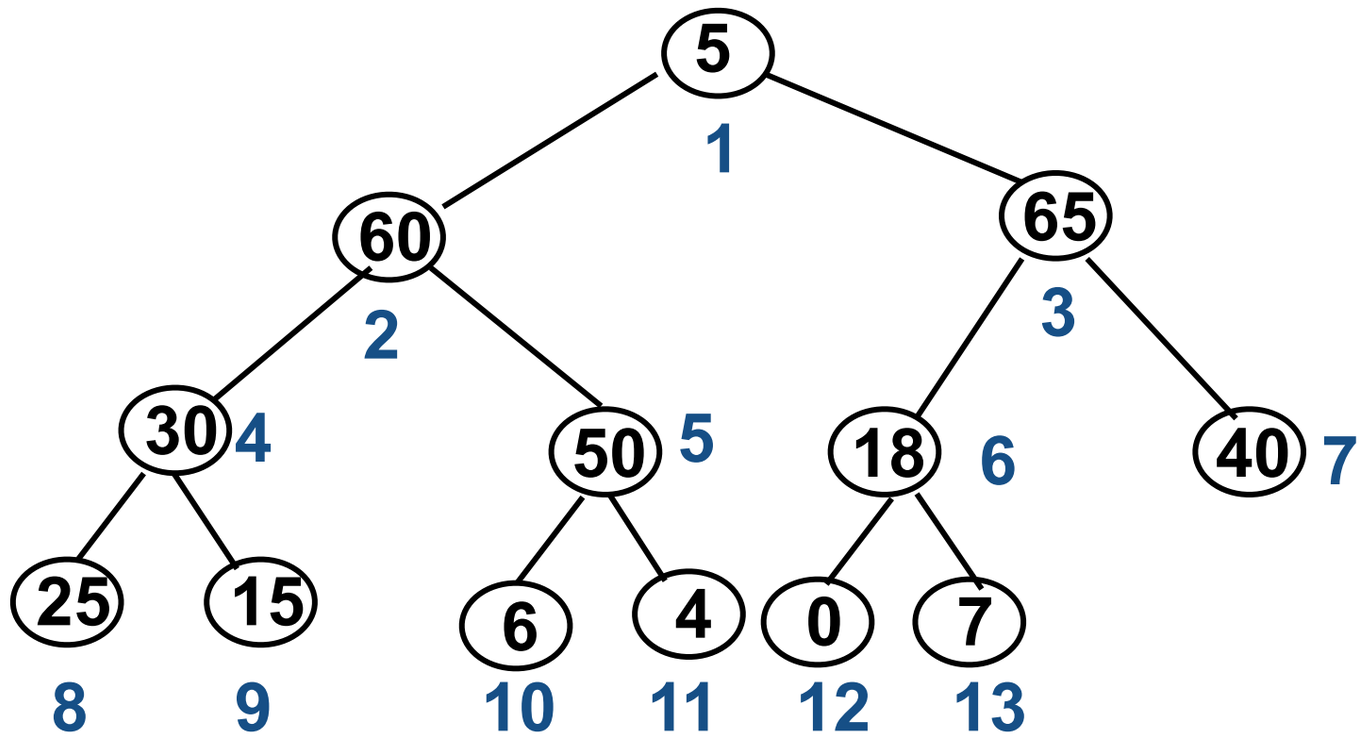


Nel livello 0 mettiamo l' elemento di indice 1. Proseguiamo riempiendo i livelli successivi da sinistra verso destra con gli elementi dell'array, nell'ordine in cui compaiono nell'array.

Heap binari e alberi quasi completi

A=

5	60	65	30	50	18	40	25	15	6	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13



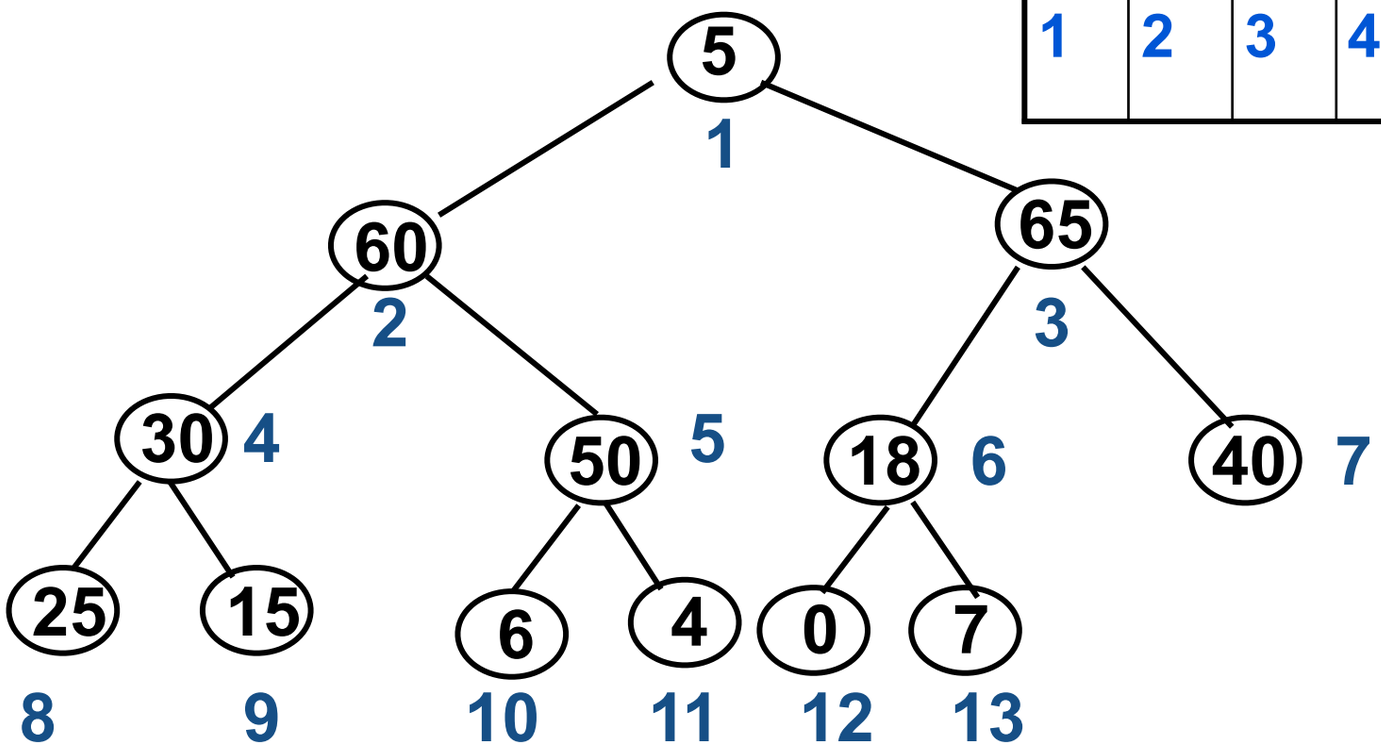
Chiamiamo quasi completi gli alberi di questa forma: completi fino al penultimo livello, con le foglie al più su due livelli consecutivi e quelle del livello massimo tutte spostate a sinistra.

Nel livello 0 mettiamo l' elemento di indice 1. Nel livello i≥1 disponiamo nell'ordine gli elementi di indice compreso tra 2^i e $2^{i+1}-1$.
Se $2^k < n < 2^{k+1}$, l'albero ha k+1 livelli, altezza k e le foglie nei livelli k-1 e k, con quelle del livello k tutte di seguito a sinistra.
Se $n = 2^{k+1}-1$ l'albero ha k+1 livelli, ed è un albero completo di altezza k.

Heap binari

A=

5	60	65	30	50	18	40	25	15	6	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

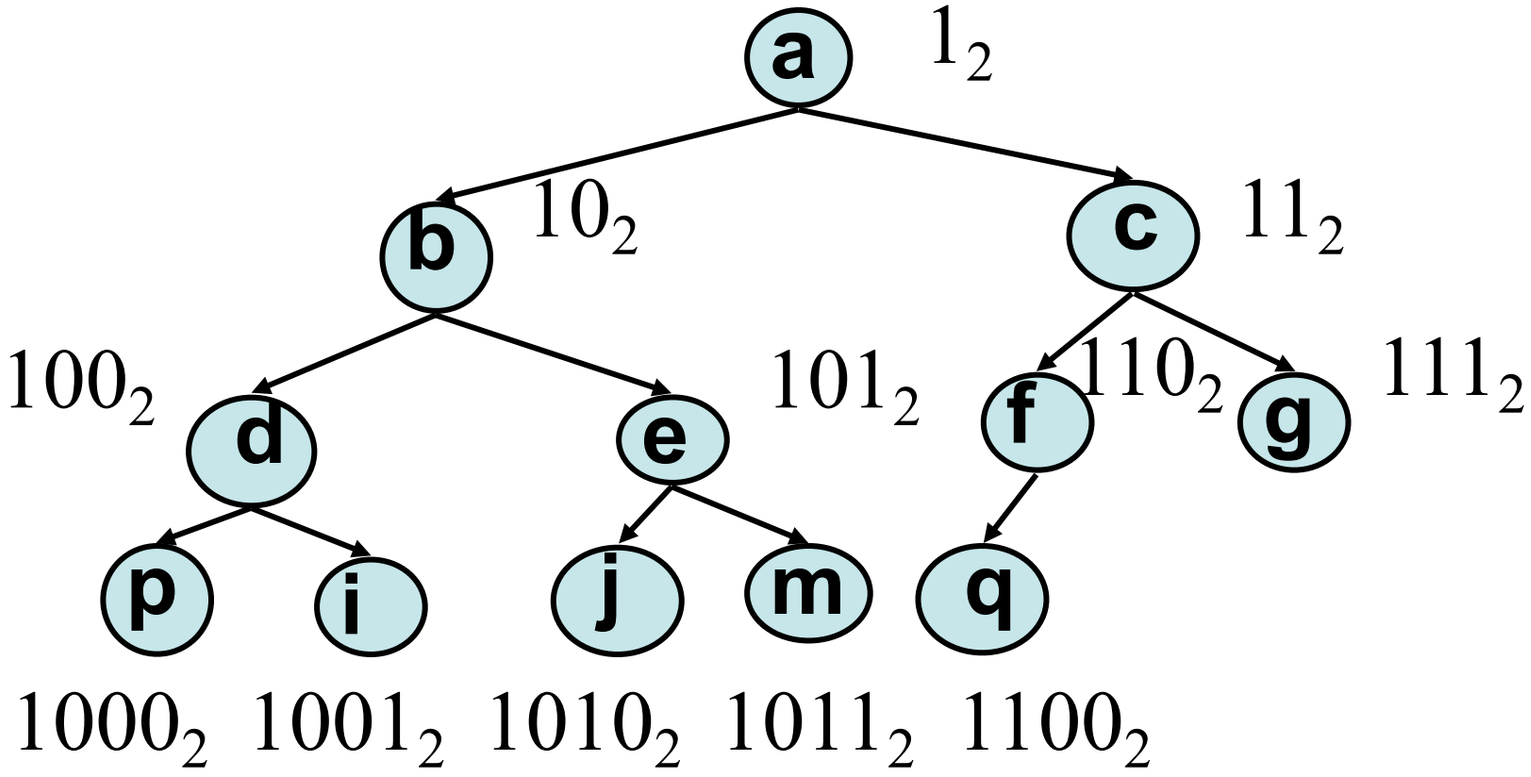


In un albero è fondamentale poter risalire da figlio a padre o da padre a figlio.

Supponiamo che nel livello i abbiamo gli elementi di indice m, m+1,m+2, …nel livello successivo avremo gli indici 2m, 2m+1,2m+2,..

In generale A[2i] e A[2i+1] sono i figli di A[i] e A[i/2] è il padre di A[i].

Calcolo di figli e padri



parent (i)
return ⌊i/2⌋

left(i)
return 2i

right(i)
return (2i+1)

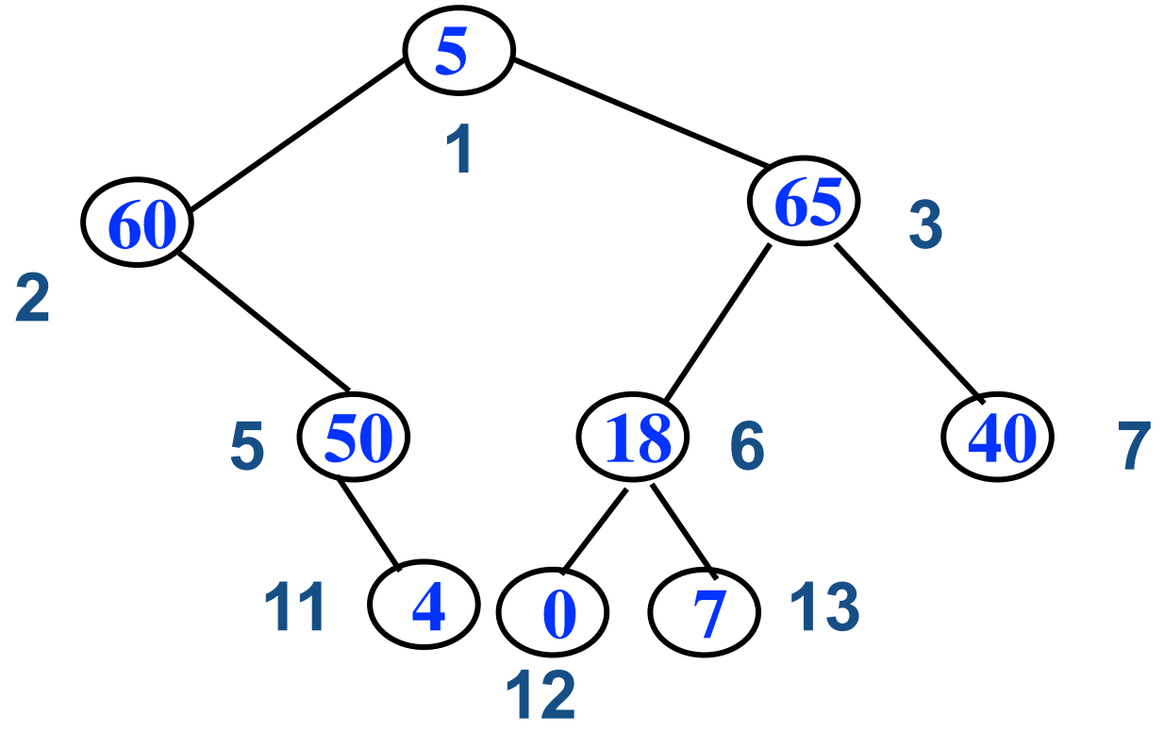
⌊i/2⌋ = scorrimento di una posizione a destra dei bit di i

2i = scorrimento a sinistra di una posizione e aggiunta di uno 0 a destra di i

2i +1 = scorrimento a sinistra di una posizione e aggiunta di un 1 a destra di i

Alberi binari qualunque in un arrays?

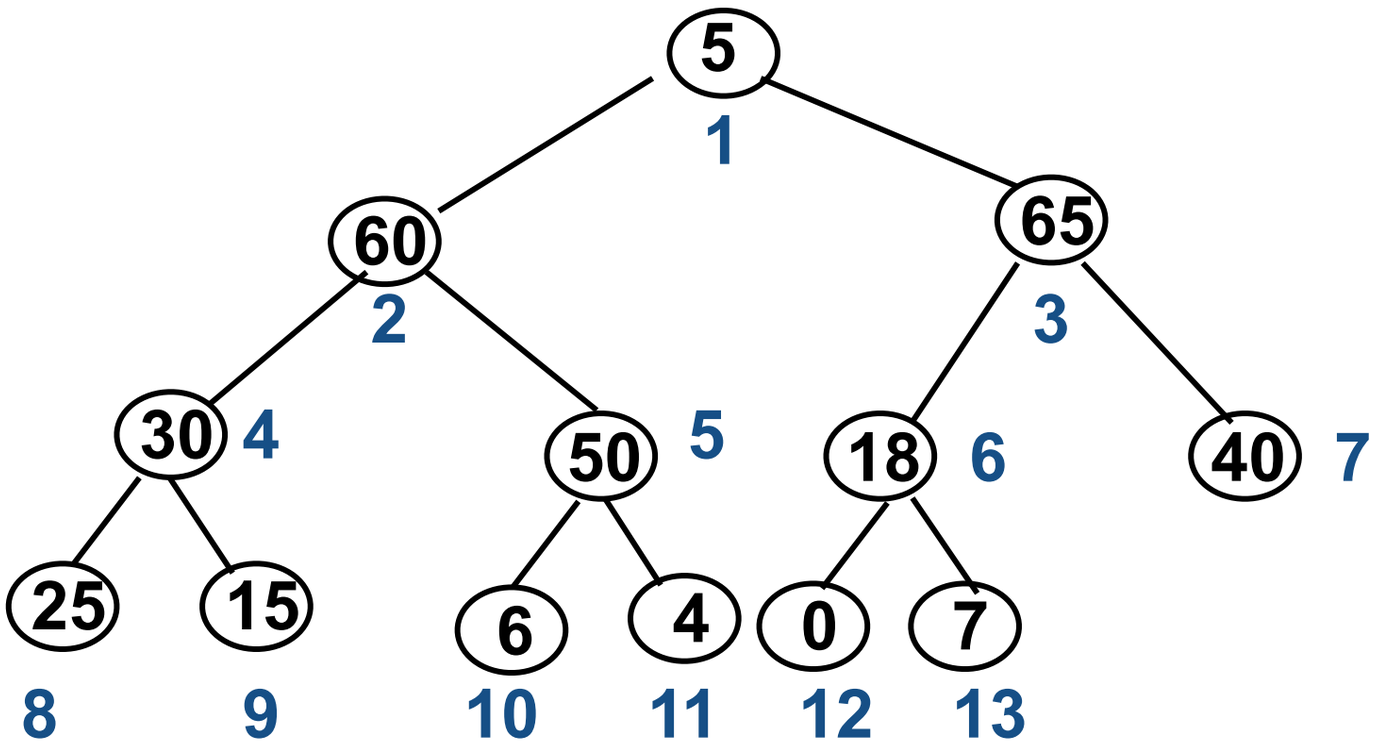
Abbiamo detto che un heap binario è una struttura dati consistente di un array visto come un albero binario.
Ma è anche vero che un qualsiasi albero può essere memorizzato in un array?
Ricordiamo che memorizzare gli elementi di un albero significa poter risalire da padre a figlio e viceversa.



A=

5	60	65	?	50	18	40	?	?	?	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

Un array con due attributi



Due attributi:

A.length = 16

A.heapsize = 13

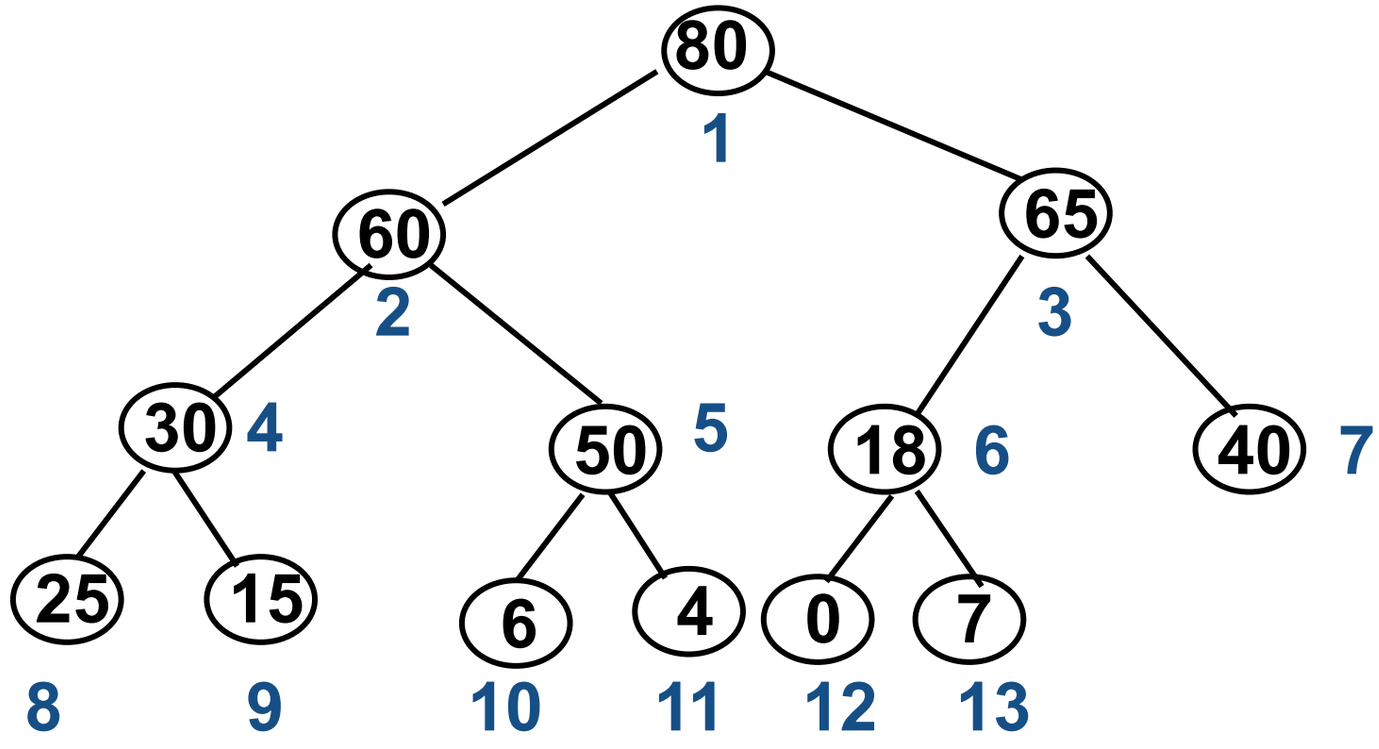
A=

5	60	65	30	50	18	40	25	15	6	4	0	7	17	2	8
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Definizione di MaxHeap

Diciamo che un heap binario A è un max-heap se ogni elemento soddisfa la proprietà del max-heap, cioè se ogni elemento (tranne la radice) è minore o uguale a suo padre.

***A*[*parent*(i)] ≥ *A*[i]**



Max-Heap

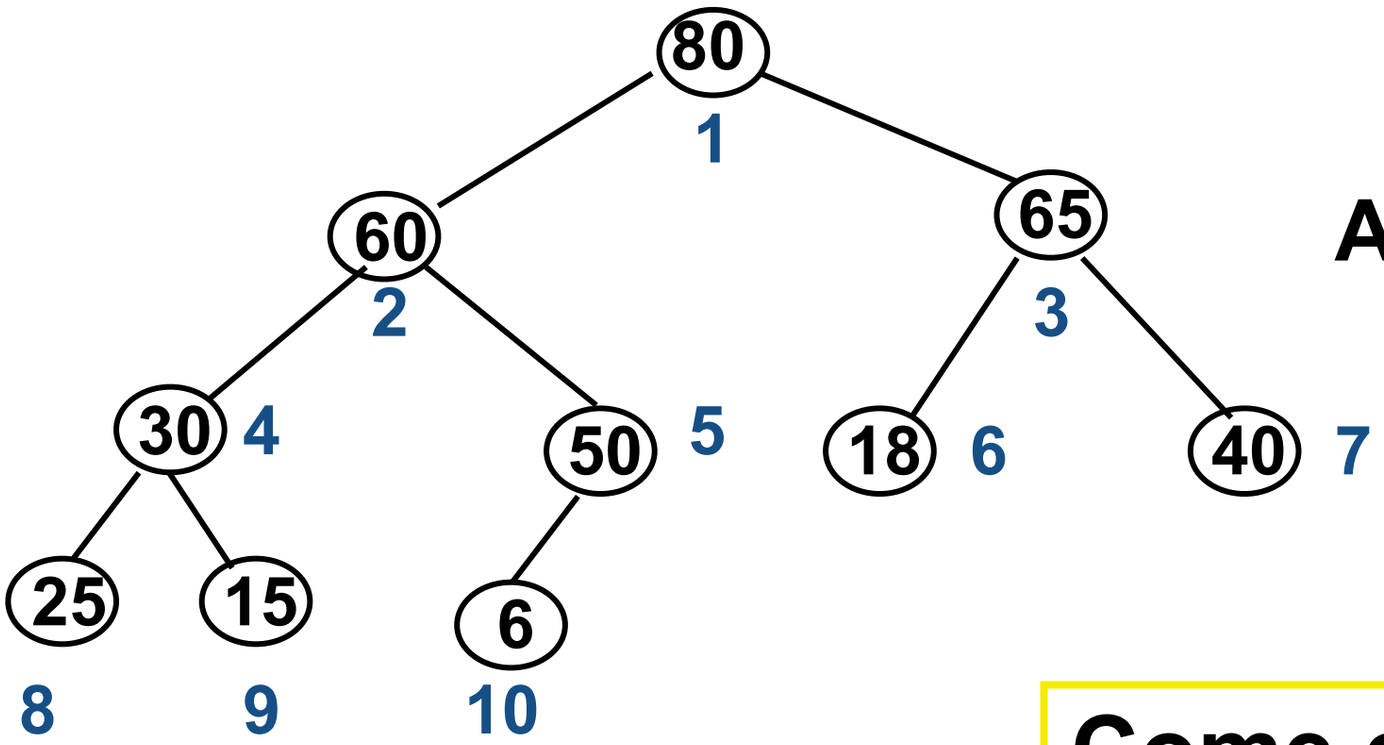
A=

80	60	65	30	50	18	40	25	15	6	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

Due attributi:

A.length = 13

A.heapsize = 10



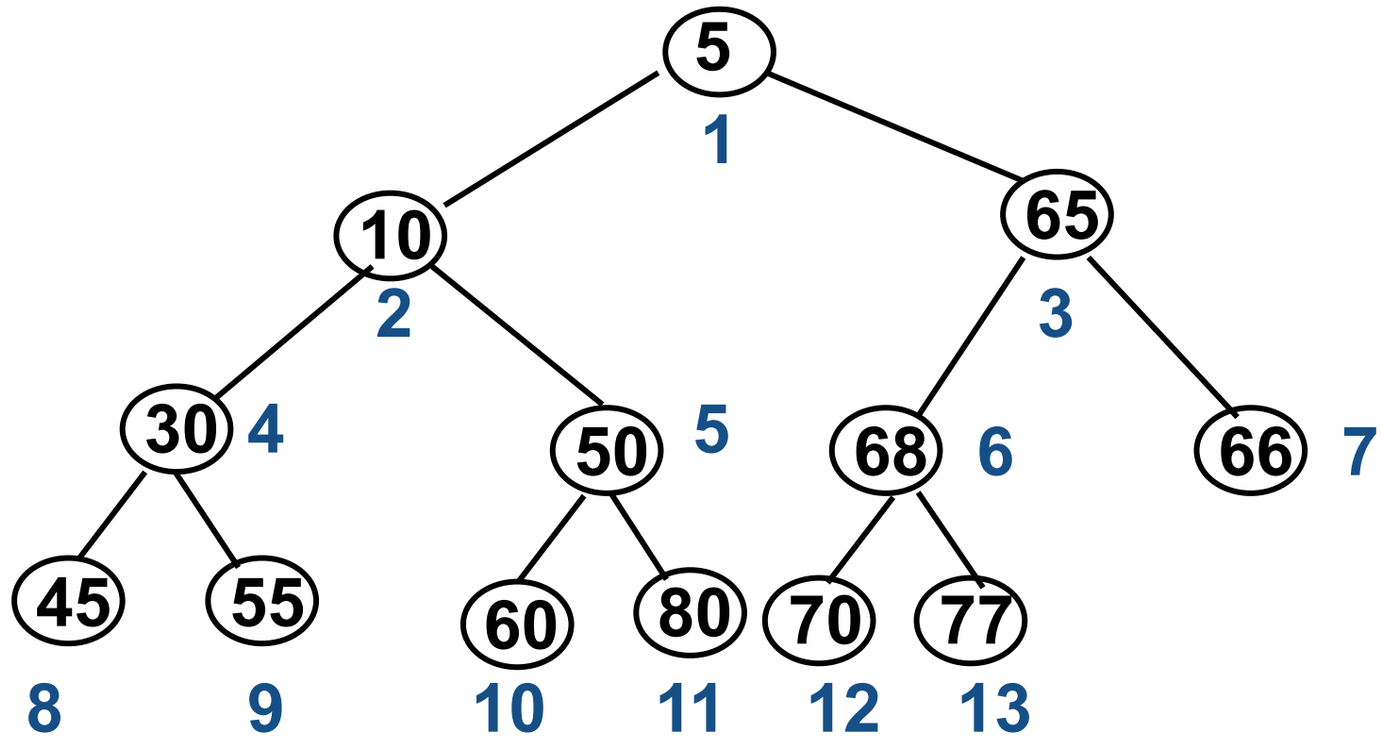
A.heapsize ≤ A.length

Come conseguenza il massimo è contenuto nella radice

Definizione di MinHeap

Diciamo che un heap binario è un min-heap se ogni elemento soddisfa la proprietà del min-heap, cioè se ogni elemento (tranne la radice) è maggiore o uguale a suo padre.

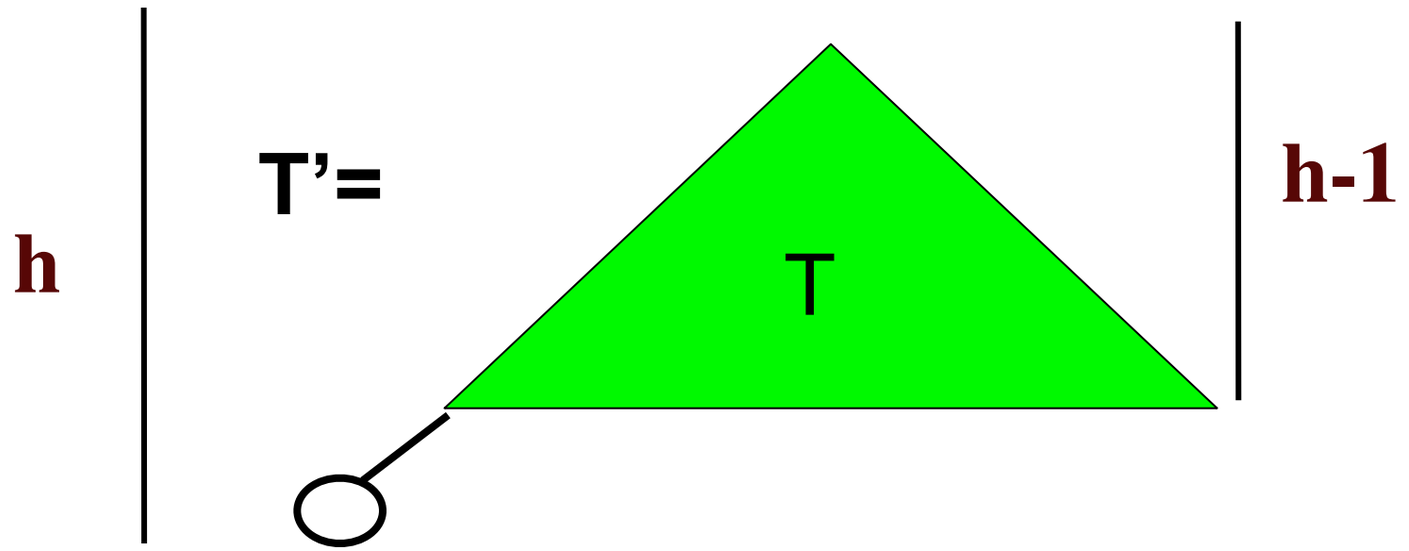
$A[parent(i)] \leq A[i]$



heap binario: altezza

Sappiamo che l'altezza h di un albero binario con n nodi é almeno la parte intera inferiore di lg n . Dimostriamo che h è anche al più la parte intera inferiore di lg n.

Prendiamo l' albero T, completo e di altezza h-1 e aggiungiamo una foglia:



numNodi(T) =

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

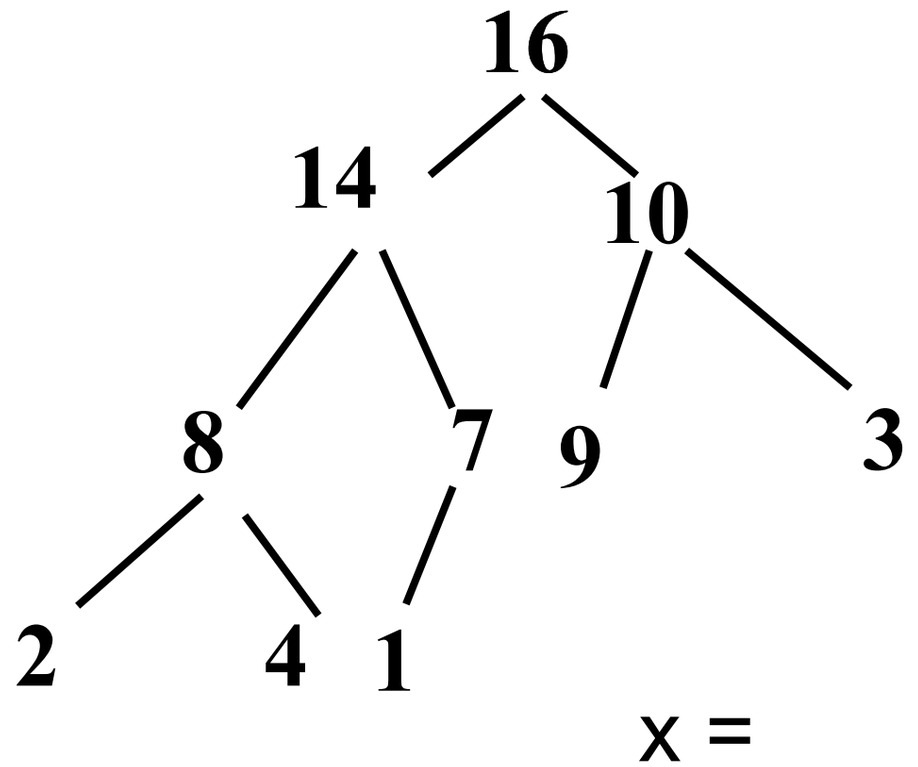
T' è l' albero quasi completo di altezza h con il minimo numero di nodi e numNodi(T') = 2^h. Quindi per ogni albero quasi completo con n nodi e di altezza h vale che $n \geq 2^h \Rightarrow \lg n \geq h$

Determinare il massimo in un maxHeap è immediato

A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

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



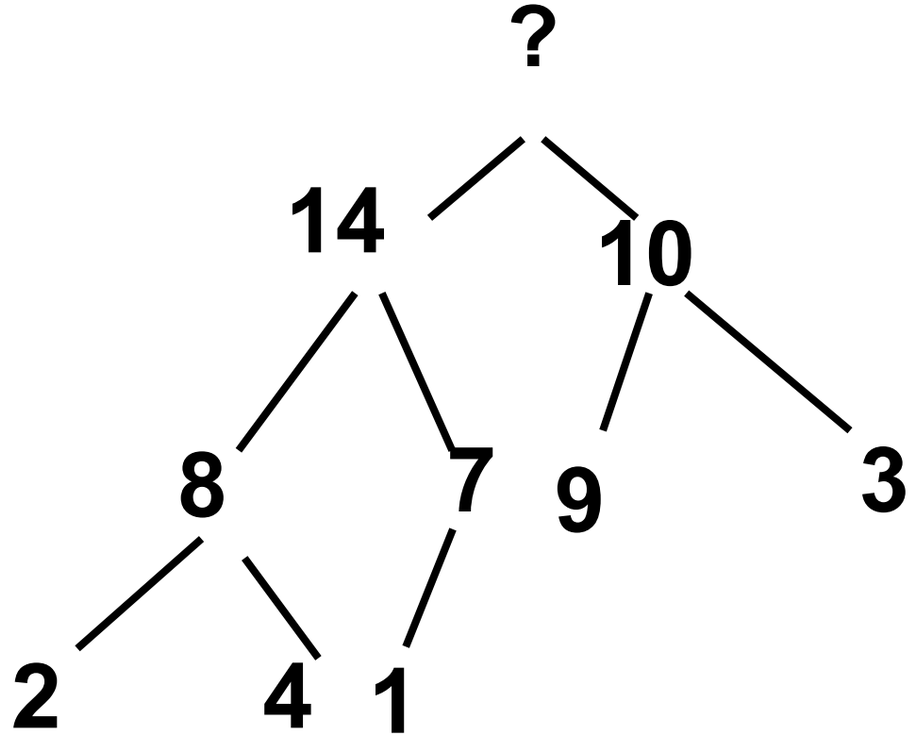
Θ(1)

Estrazione del massimo

A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

x =



Dopo l'operazione bisogna avere un MaxHeap con un elemento in meno.

Estrazione Massimo

Proprietà MaxHeap violata alla radice

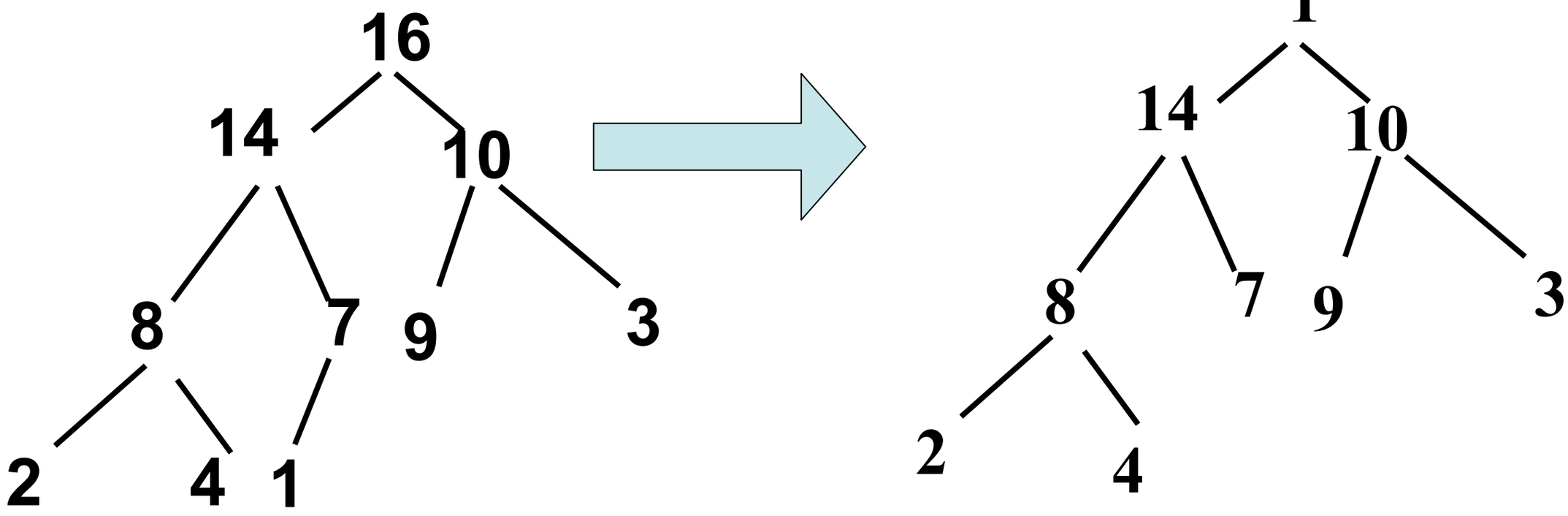
A=

16	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

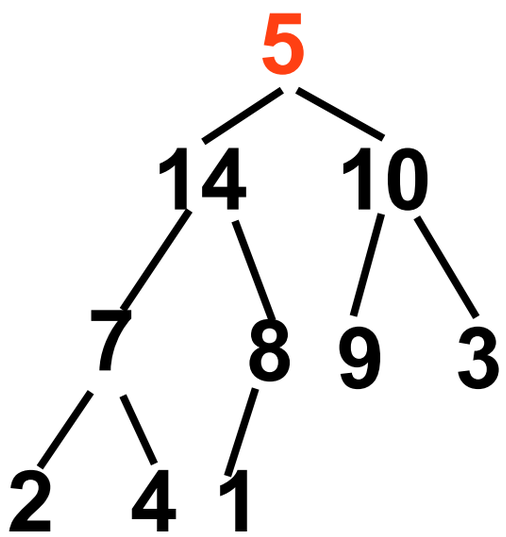
A=

1	14	10	8	7	9	3	2	4	1	4	0	7
1	2	3	4	5	6	7	8	9	10	11	12	13

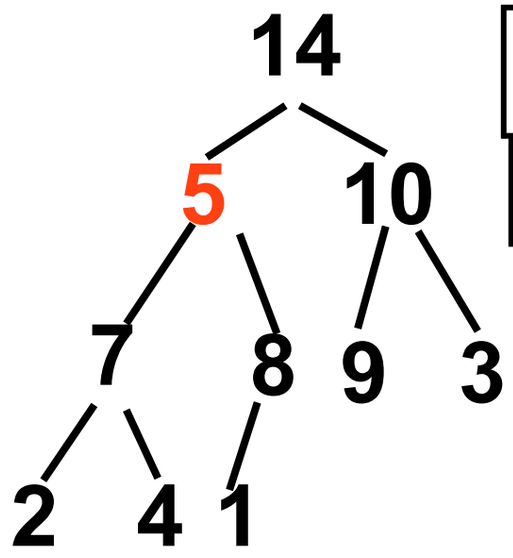
max =



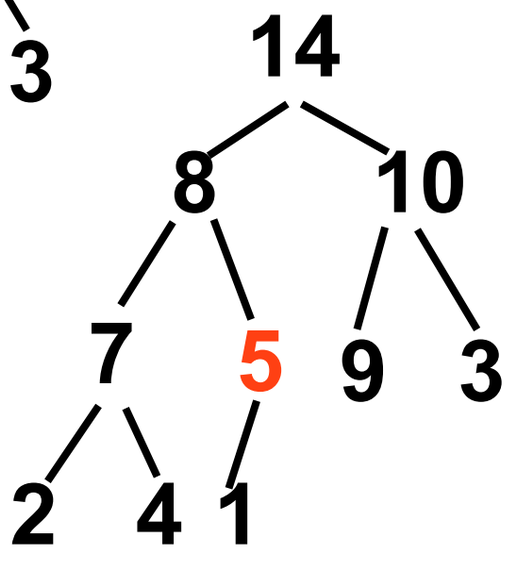
ripristino proprietà del Max-heap



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



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



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

Max-Heapify

Max-Heapify (A,i)

Input: A è un array e i è un indice

Prec: i figli o il figlio di A[i] sono radici di un max-heap, mentre A[i] può essere più piccolo di uno o di entrambi i suoi figli
Output: A[i] è radice di un max-Heap

se A[i] ha due figli prendiamo il figlio con valore massimo
se ha un figlio prendiamo quell'unico figlio
confrontiamo il valore di A[i] con quello del figlio selezionato
se il valore del figlio è maggiore di quello del padre, li scambiamo
e proseguiamo nello stesso modo sul figlio fino a quando la proprietà è ristabilita.

```
Max-Heapify (A,i)
```
Input: A è un array e i è un indice

Prec: A[left(i)] e A[right(i)] sono radici di max-heap mentre A[i] può essere più piccolo dei suoi figli

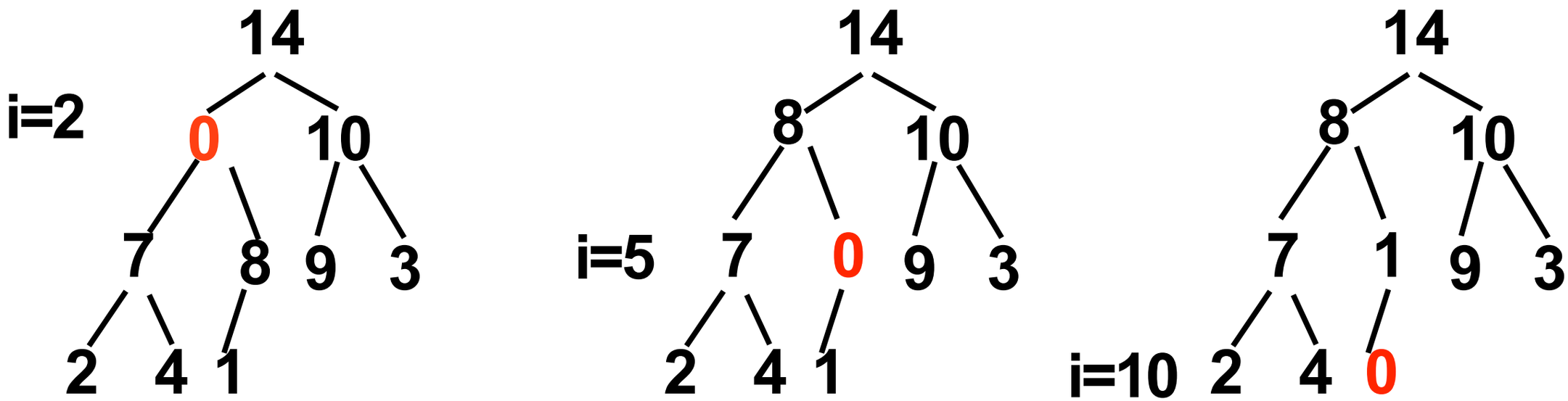
Postc: A[i] è radice di un max-Heap

```
max = i
while max == i do
if (2i +1 ≤ A.heap.size) then
    if (A[2i+1] > A[2i]) then max = 2i+1 else max = 2i
```
(tra i due figli , se ci sono, si prende l'indice del massimo)

```
elseif (2i ≤ A.heap-size) then max = 2i
```
(max è l'indice del figlio sinistro se c'è)

```
 if (A[max] > A[i]) then
                    scambia A[i]  e  A[max]
                    i = max
                    else max = i+1
```
(max ≠ i per uscire dal ciclo nel caso che i sia una foglia)

Max-Heapify: esempio di esecuzione



Max-Heapify(A,2).
max = i = 2, A[4] = 7 e A[5] = 8 quindi max = 5
A[max] > A[2]
quindi si scambiano e i = max = 5, si ripete, ora max = 10,
ma A[10] > A[5], si scambiano e i = max = 10, ora però i due if danno falso e A[i] = A[max], perché i=max, e max diventa i+1 per uscire dal ciclo

Max-Heapify: complessità

```
Max-Heapify (A,i)
max = i
while max == i do
if (2i +1 ≤ A.heap.size) then
    if (A[2i+1] > A[2i]) then max = 2i+1 else max = 2i
```

(tra i due figli , se ci sono, si prende l'indice del massimo)

```
else if (2i ≤ A.heap-size) then max = 2i
```

(al più c'è il figlio sinistro)

```
 if (A[max] > A[i]) then
                    scambia A[i]  e  A[max]
                    i = max
                        else max = i+1
```

(se A[max] <= A[i] si esce dal ciclo)

Ad ogni esecuzione del ciclo si "scende" di padre in figlio, quindi al più si va dalla radice alla foglia più lontana.
Caso peggiore:
$T(n) = \Theta(\log n)$

Rimozione del massimo(A)

Heap-Extract-Max (A)

Input: A è un array

Prec: A è un max-heap

Postc: dà in output il massimo, che viene rimosso da A rispristinando la proprietà del max-heap

```
if A.heap.size < 1 then error "l'heap è vuoto"
max = A[1]
A[1]  = A[A.heap.size]
A.heap.size = A.heap.size - 1
Max-Heapify (A,1)
return max
```

O(lg n),
n = heap-size(A)