Unitelma Sapienza

Sapienza
Università di Roma
Dipartimento di Informatica

Corso di Introduzione agli algoritmi
Prof.ssa Tiziana Calamoneri

Il problema dell'Ordinamento: l'Heap sort

Heap sort (1)

L'algoritmo ***heapsort*** è un algoritmo di ordinamento piuttosto complesso che esibisce ottime caratteristiche:

- come mergesort ha un costo computazionale di $O(n \log n)$ anche nel caso peggiore
- come selection sort ordina in loco.

Sfrutta una opportuna organizzazione dei dati che garantisce una o più specifiche proprietà, il cui mantenimento è essenziale per il corretto funzionamento dell'algoritmo.

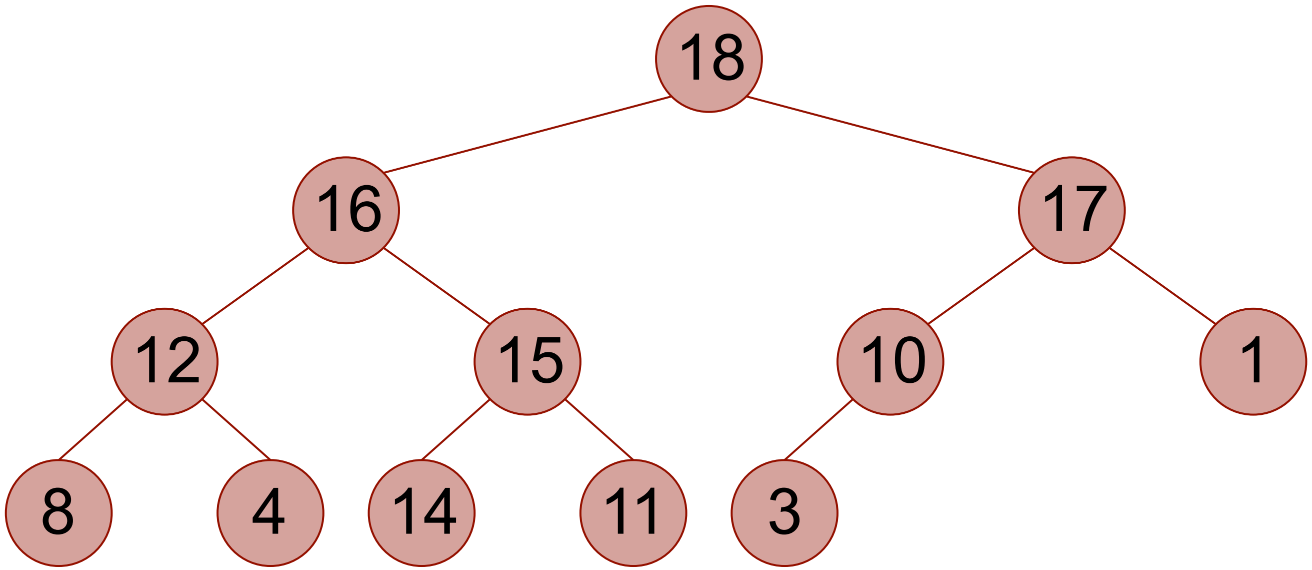
⬇

struttura dati Heap

Heap (1)

Un ***heap*** è un albero binario *completo o quasi completo*, ossia un albero binario in cui tutti i livelli sono pieni, tranne l'ultimo, i cui nodi sono addensati a sinistra…

con la proprietà che la chiave su ogni nodo è maggiore o uguale alla chiave dei suoi figli (proprietà di ordinamento verticale).



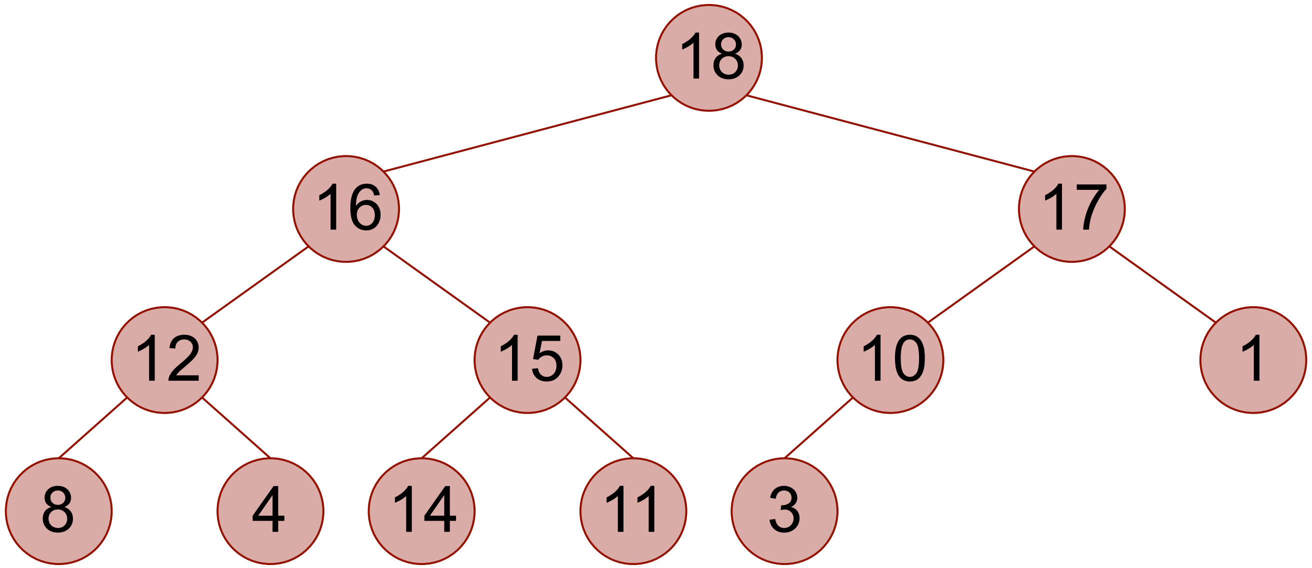
Heap (2)

Il modo più naturale per memorizzare un heap è utilizzare un vettore *A*, con indici che vanno da *1* fino al numero di nodi dell'heap, *heap_size*, i cui elementi possono essere messi in corrispondenza con i nodi dell'heap:

- il vettore è riempito a partire da sinistra; se contiene più elementi del numero *heap_size* di nodi dell'albero, allora i suoi elementi di indice > *heap_size* non fanno parte dell'heap;
- ogni nodo dell'albero binario corrisponde a uno e un solo elemento del vettore *A*;
- la radice dell'albero corrisponde ad *A[1]*;
- il figlio sinistro del nodo che corrisponde all'elemento *A[i]*, se esiste, corrisponde all'elemento A[2i]: ***left(i) = 2i***;
- il figlio destro del nodo che corrisponde all'elemento *A[i]*, se esiste, corrisponde all'elemento *A[2i + 1]*: ***right(i) = 2i+1***;
- il padre del nodo che corrisponde all'elemento A[i] corrisponde all'elemento A[$\lfloor i/2 \rfloor$]: ***parent(i)*** = $\lfloor i/2 \rfloor$.

Heap (3)

18	16	17	12	15	10	1	8	4	14	11	3



Heap (4)

Proprietà:

- Poiché lo heap ha tutti i livelli completamente pieni tranne al più l'ultimo, la sua altezza è $\Theta(\log n)$.
- Con questa implementazione, la proprietà di ordinamento verticale implica che per tutti gli elementi tranne $A[1]$ (poiché esso corrisponde alla radice dell'albero e quindi non ha genitore) vale:

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

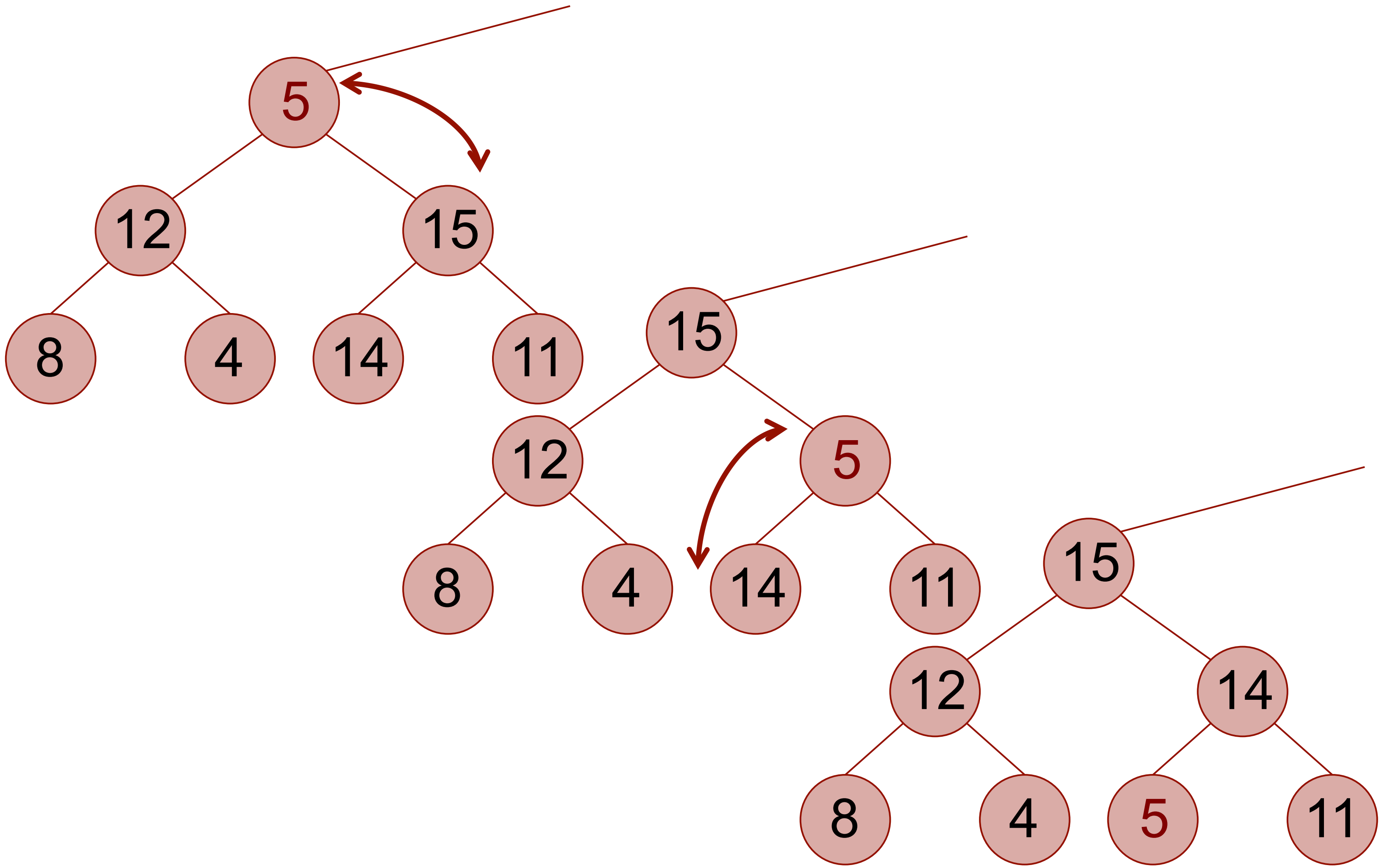
- L'elemento massimo risiede nella radice, quindi può essere trovato in tempo $O(1)$.

Heap (5)

Funzioni per la gestione di un heap:

La funzione heapify ha lo scopo di mantenere la proprietà di heap, sotto l'ipotesi che nell'albero su cui viene fatta lavorare sia garantita la proprietà di heap per entrambi i sottoalberi (sinistro e destro) della radice. Di conseguenza l'unico nodo che può violare la proprietà di heap è la radice dell'albero, che può essere minore di uno o di entrambi i figli.

Funzioni per la gestione di un heap (segue):



Heap (7)

Funzioni per la gestione di un heap (segue):

```
Funzione Heapify (A: vettore; i: intero)
    L ← left(i); R ← right(i)
    if ((L ≤ heap_size) and (A[L] > A[i]))
        indice_massimo ← L
    else
        indice_massimo ← i
    if ((R ≤ heap_size) and (A[R] > A[indice_massimo]))
        indice_massimo ← R
    if (indice_massimo ≠ i)
        scambia A[i] e A[indice_massimo]
        Heapify (A, indice_massimo)
    return
```

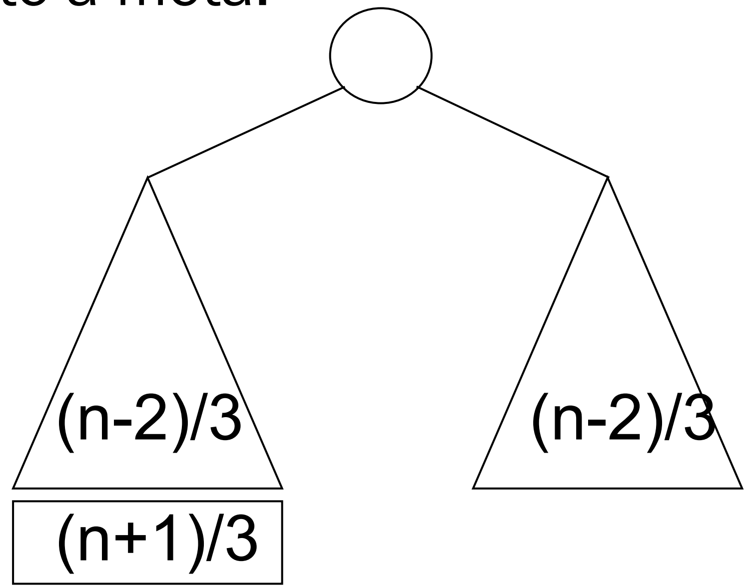
T(n)=
$\Theta(1)$ +
$\Theta(1)$ +
$\Theta(1)$ +
$\Theta(1)$ +
T(num. di nodi nel sottoalb. maggiore)

Heap (8)

Funzioni per la gestione di un heap (segue):

$T(n)=\Theta(1)+T(\text{num. di nodi nel sottoalb. maggiore})$

I sottoalberi della radice non possono avere più di $2n/3$ nodi, situazione che accade quando l'ultimo livello è pieno esattamente a metà:



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

$\Downarrow$

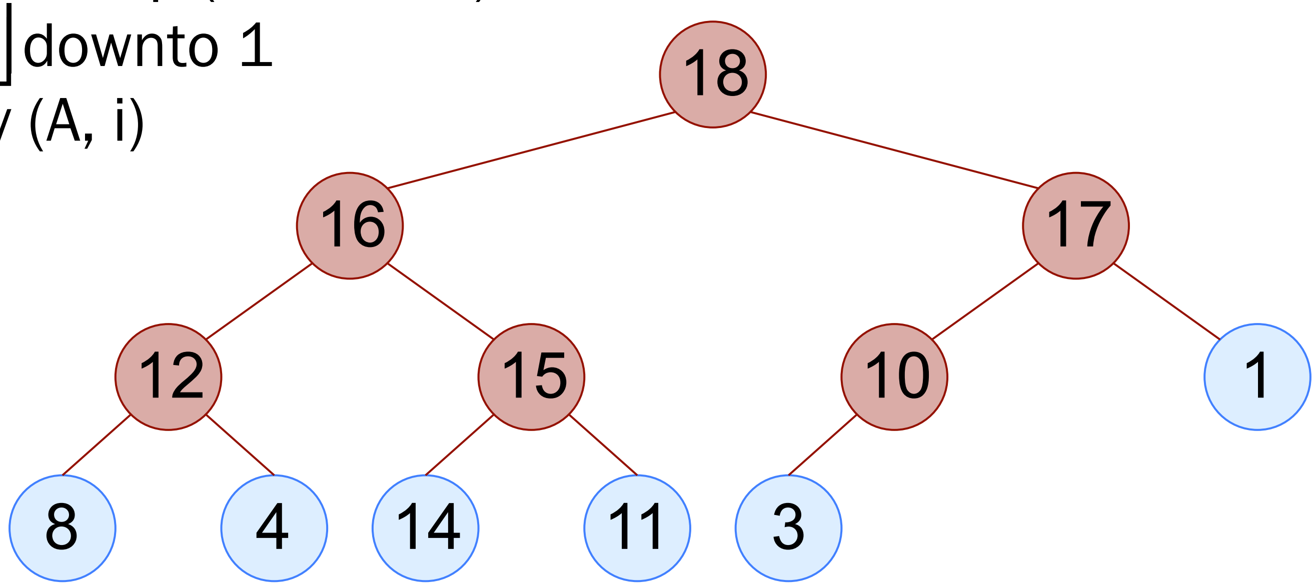
$T(n)=O(\log n)$

Heap (9)

Funzioni per la gestione di un heap (segue):

La funzione build_heap serve per trasformare qualunque vettore contenente *n* elementi in uno heap sfruttando heapify:

Funzione Build_heap (A: vettore)
    for i = $\lfloor n/2 \rfloor$ downto 1
        Heapify (A, i)
    return



Heap (10)

Funzioni per la gestione di un heap (segue):

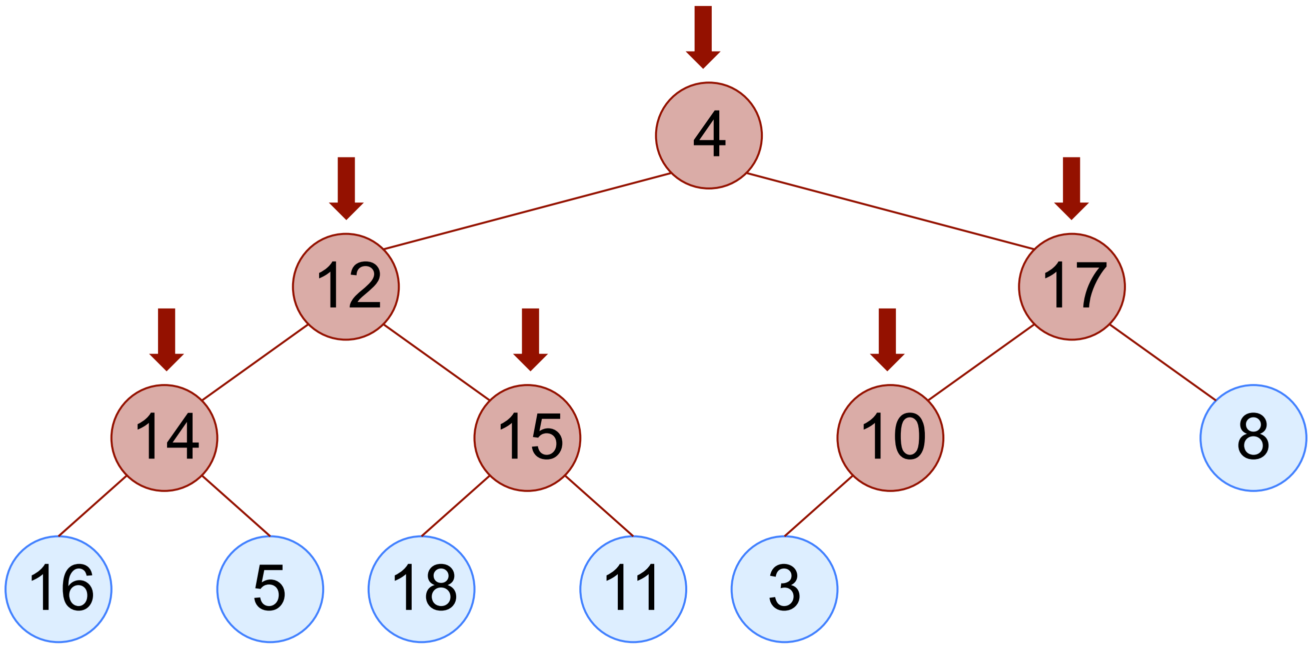
Funzione Build_heap (A: vettore)

for i = $\lfloor n/2 \rfloor$ downto 1

Heapify (A, i)

return

i=[illegible]



Heap (11)

Funzioni per la gestione di un heap (segue):

Funzione Build_heap (A: vettore)
for i =$\lfloor n/2 \rfloor$ downto 1
Heapify (A, i)
return

T(n)=
$$\Theta(1)+\sum_{i=1}^{n/2}$$
Θ(*log* #nodi nel sottoalbero radicato ad *i)*

Poiché *log* #nodi nel sottoalbero radicato ad *i* < *log n*:
T(n)=$\Theta(1)+O(n \log n)=O(n \log n)$

Con un calcolo più accurato si può mostrare che $T(n)=\Theta(n)$

Heap sort (2)

Heapsort:

- trasforma un vettore *A* di dimensione *n* in un heap (di *n* nodi), mediante Build_heap.
- Ora il max del vettore è in *A[1]* e, per metterlo nella corretta posizione dell'ordinamento, basta scambiarlo con *A[n]*.
- La dim. dell'heap viene ridotta ad *(n – 1)*, e:
  - i due sottoalberi della radice sono ancora degli heap
  - solo la nuova radice (ex foglia più a destra) può violare la proprietà del nuovo heap di dimensione (n – 1).
- ripristina la proprietà di heap sui residui *(n – 1)* elementi con Heapify;
- scambia il nuovo max *A[1]* col penultimo elemento;
- riapplica il procedimento riducendo via via la dimensione dell'heap a *(n – 2), (n –3),* ecc., fino ad arrivare a 2.

Heap sort (3)

```
Funzione Heapsort (A: vettore)
    Build_heap(A)
    for heap_size = n downto 2
        scambia A[1] e A[heap_size]
        Heapify(A, heapsize-1, 1)
    return
```

T(n)=
O(*n log n*)+
n-2 volte
  $\Theta(1)$ +
  $\Theta(\log n)$

In totale: $T(n)=\Theta(n \log n)$

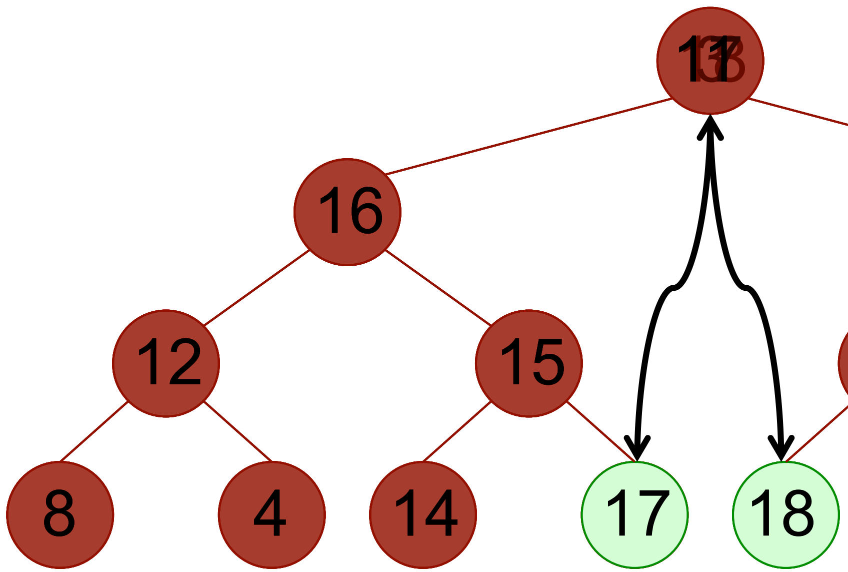
Heap sort (4)

18	16	17	12	15	10	1	8	4	14	11	3

3	16	17	12	15	10	1	8	4	14	11	18

17	16	10	12	15	3	1	8	4	14	11	18

11	16	10	12	15	3	1	8	4	14	17	18



e così via…

Esercizi

- Progettare un algoritmo che, dato in input un vettore che rappresenta un heap, restituisca il valore minimo. Fare le opportune considerazioni sul costo computazionale

- Un Heap minimo è un albero binario completo o quasi completo con la proprietà che la chiave su ogni nodo è minore o uguale alla chiave dei suoi figli. Si modifichi l’algoritmo di Heap sort in modo che la struttura dati di riferimento sia un heap minimo e non un heap.

/