Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Il problema dell'ordinamento: algoritmo Heapsort

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Sulla base delle slides a cura di T. Calamoneri e G. Bongiovanni per il corso di informatica generale AA 2019/2020

Heapsort

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

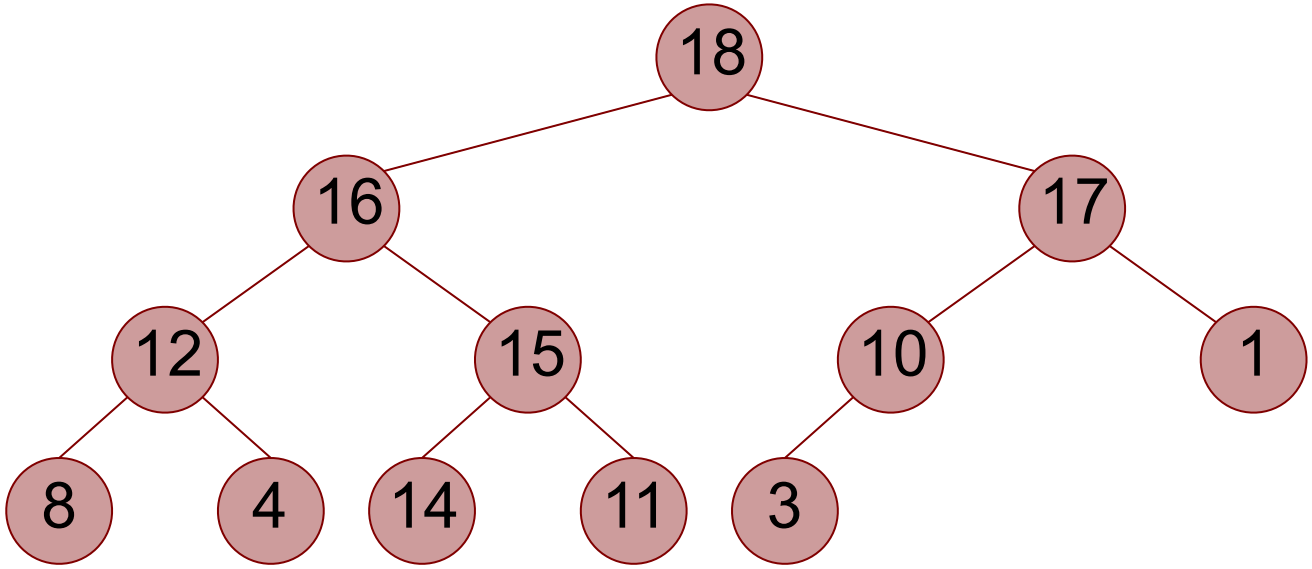
- come Mergesort ha un costo computazionale di $O(n \log n)$ anche nel caso peggiore
- come Selection sort ordina in loco.

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

Struttura dati **Heap**

La struttura dati Heap

Uno ***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 l'ulteriore proprietà che la chiave di ogni nodo è *maggiore o uguale* alla chiave dei suoi figli (proprietà di ordinamento verticale).

La struttura dati Heap

Sia ***heap_size*** il numero di nodi dell' heap.

Il modo più naturale per memorizzare uno heap è utilizzare ***heap_size*** locazioni di un vettore A.

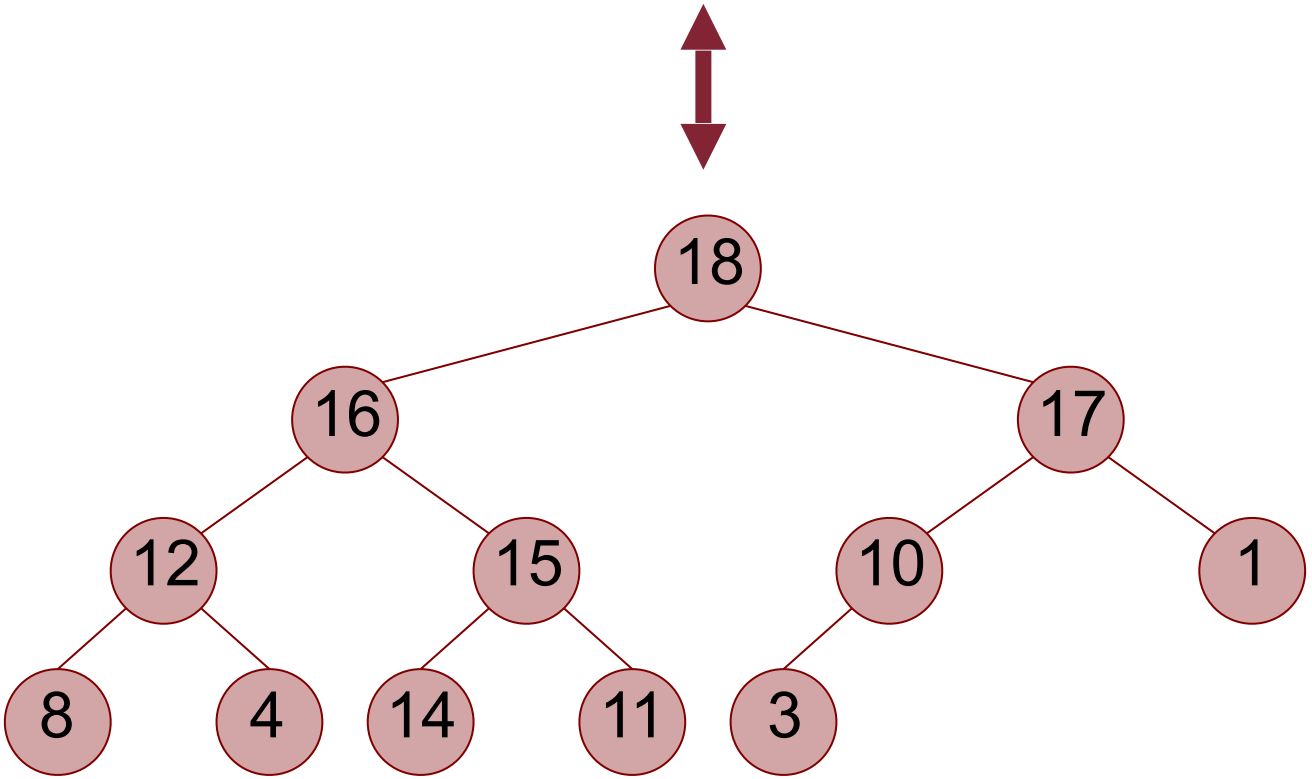
- il vettore è riempito a partire da sinistra; se contiene più elementi del numero *heap_size*, allora i suoi elementi di indice maggiore o uguale a *heap_size* non fanno parte dell'heap;

La struttura dati Heap (continua)

- ogni nodo dell'albero binario corrisponde a uno e un solo elemento del vettore *A.*
- la radice dell'albero corrisponde ad $A[0]$.
- il figlio sinistro del nodo che corrisponde all'elemento $A[i]$, se esiste, corrisponde all'elemento $A[2i+1]$: ***left(i)*** **=** $2 \cdot i + 1$
- il figlio destro del nodo che corrisponde all'elemento $A[i]$, se esiste, corrisponde all'elemento $A[2i+2]$: ***right(i)*** **=** $2 \cdot i + 2$
- il padre del nodo che corrisponde all'elemento $A[i]$ corrisponde all'elemento $A\left[\left\lfloor \frac{i-1}{2} \right\rfloor\right]$: ***parent(i)*** **=** $\left\lfloor \frac{i-1}{2} \right\rfloor$.

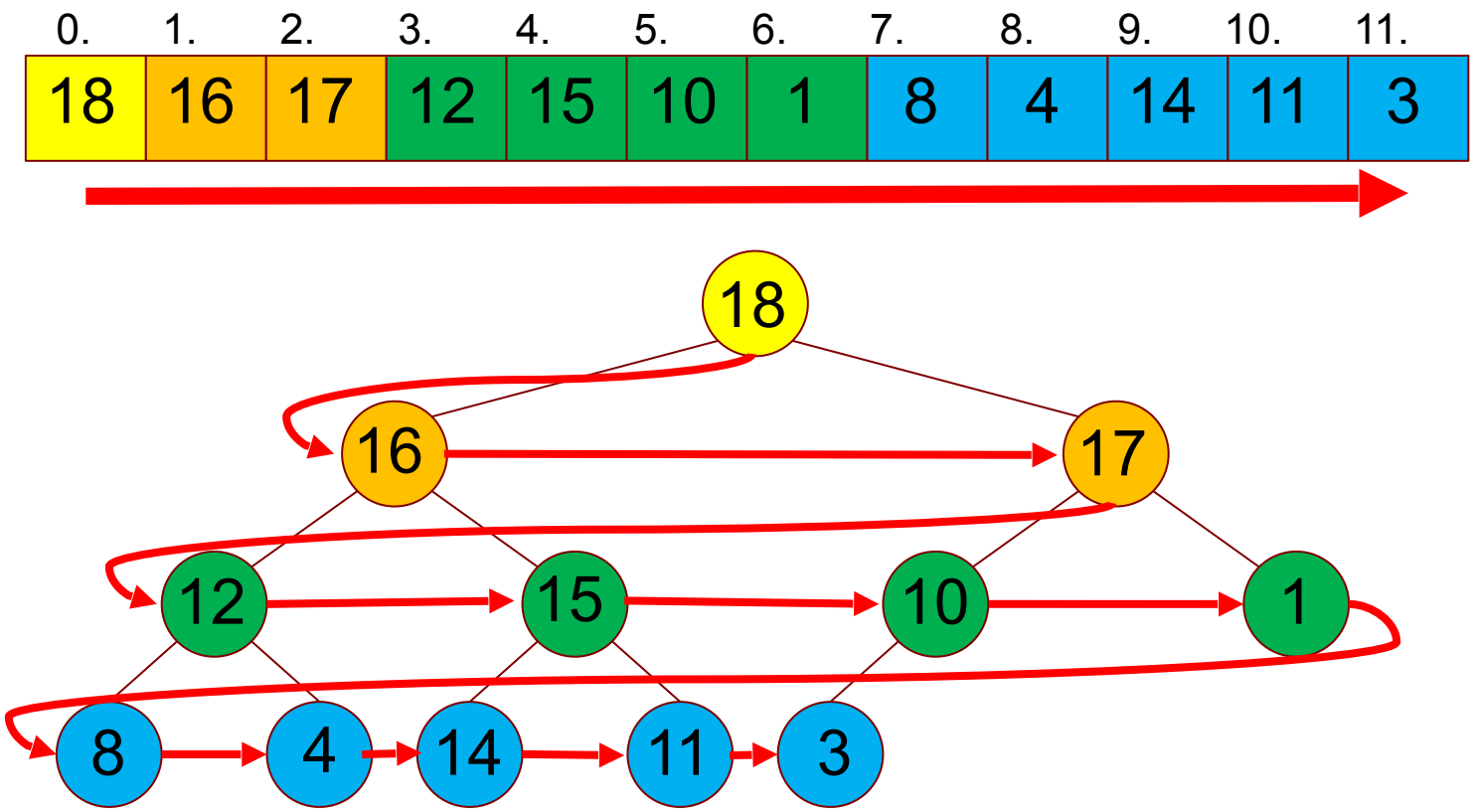
La struttura dati Heap

0.	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.
18	16	17	12	15	10	1	8	4	14	11	3



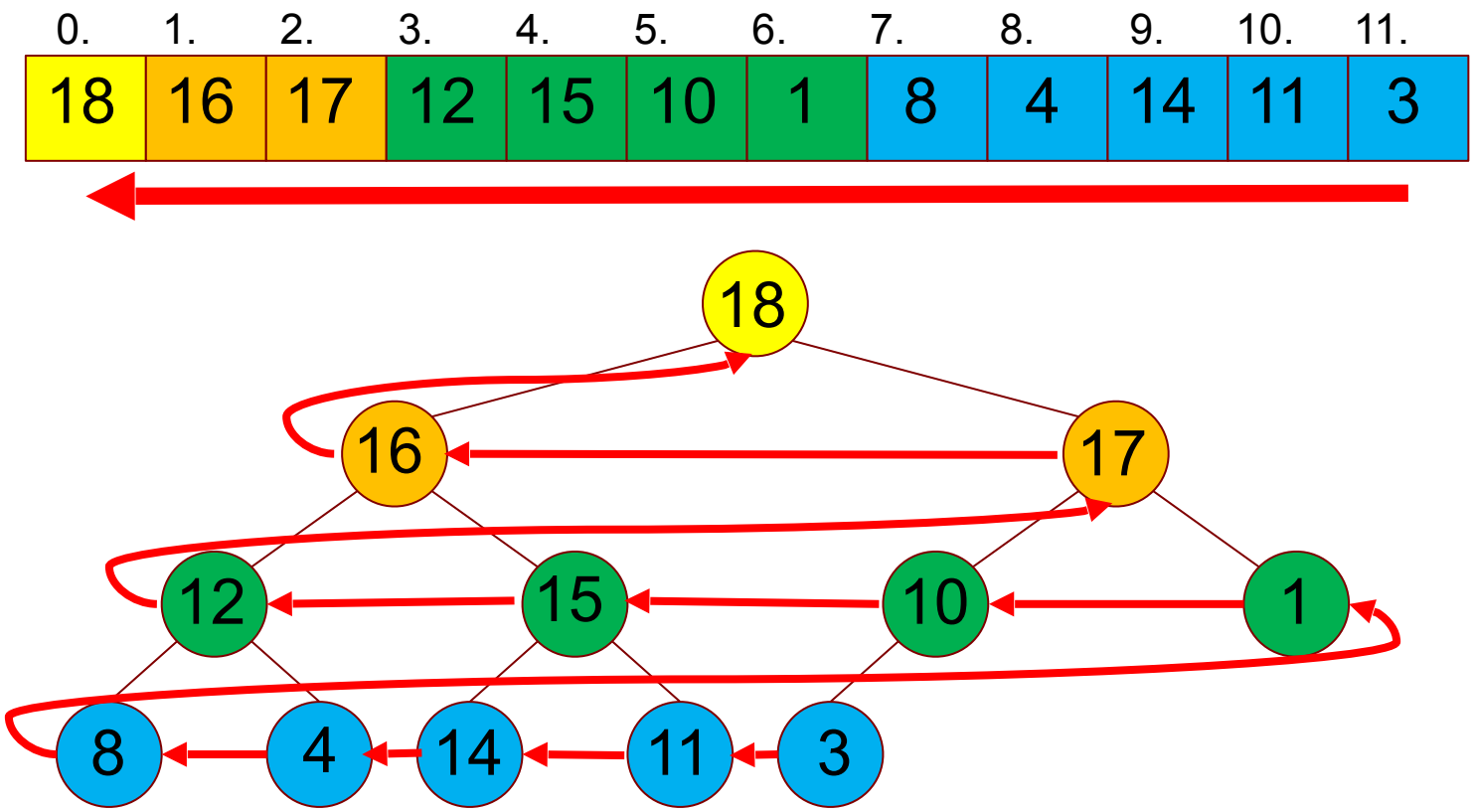
La struttura dati Heap

Si noti che scorrere il vettore da sinistra a destra corrisponde a muoversi sull'albero per livelli, dall'alto verso il basso e da sinistra a destra in ciascun livello



La struttura dati Heap

Simmetricamente, scorrere il vettore da destra a sinistra corrisponde a muoversi sull'albero per livelli, dal basso verso l'alto e da destra a sinistra in ciascun livello



La struttura dati Heap

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 $A[0]$ (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)*.

Funzione ausiliarie

L’algoritmo Heapsort si avvale di due funzioni ausiliarie, necessarie per il suo corretto funzionamento

- Funzione Heapify
- Funzione Buildheap

Illustreremo tali funzioni prima di descrivere l’algoritmo Heapsort.

Funzione Heapify

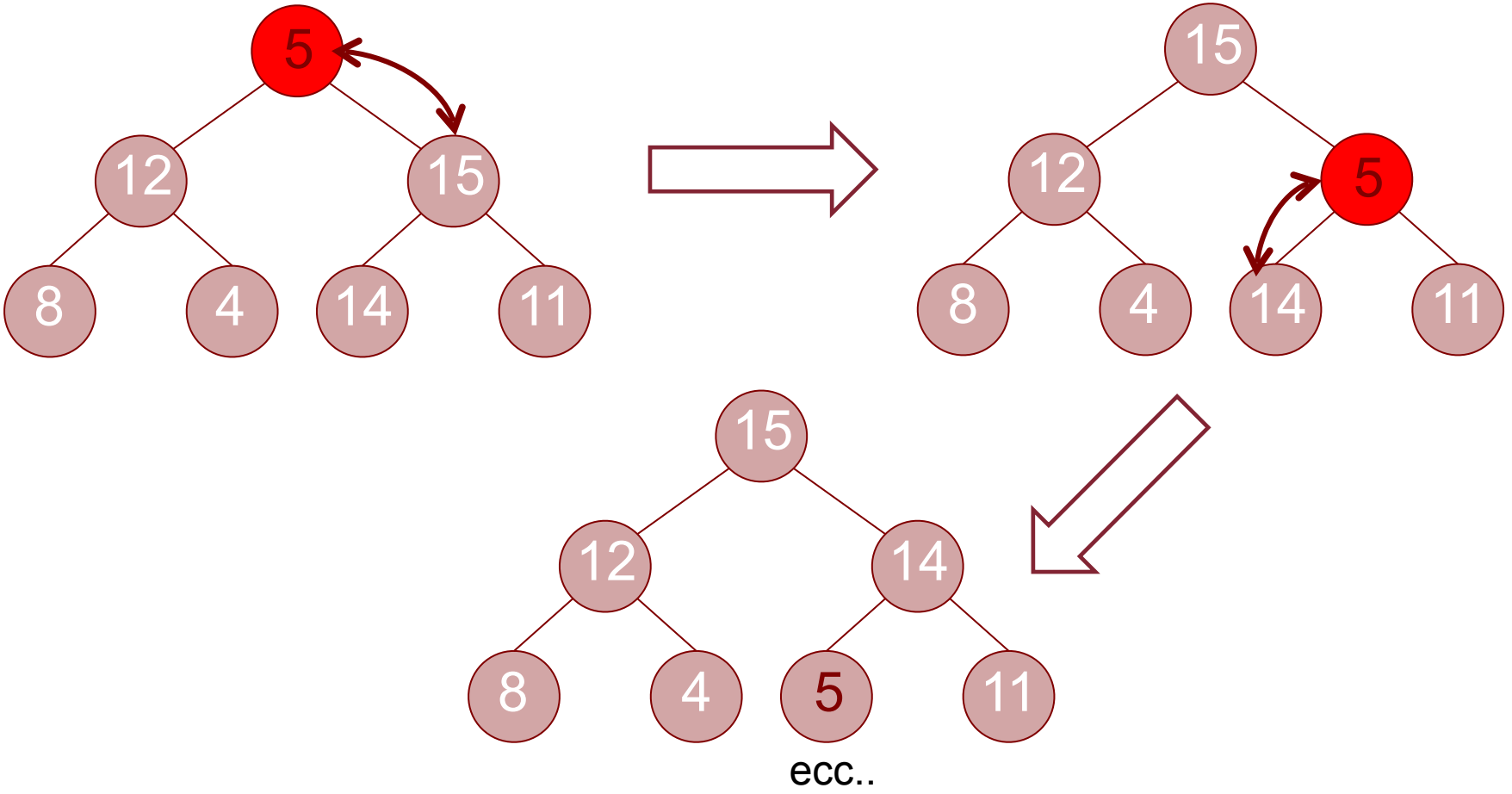
- La funzione Heapify ha lo scopo di mantenere la proprietà di heap, sotto l'ipotesi che nell'albero su cui viene fatta lavorare sia garantita la proprietà di heap per entrambi i sottoalberi (sinistro e destro) della radice. Di conseguenza l'unico nodo che può violare la proprietà di heap è la radice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

- La funzione opera sulla radice confrontandola coi suoi figli e, se necessario, la scambia col maggiore dei suoi figli.

- Dopo che lo scambio si verifica se la violazione si è trasferita sul figlio scambiato si ripete ricorsivamente l'operazione su tale nodo.

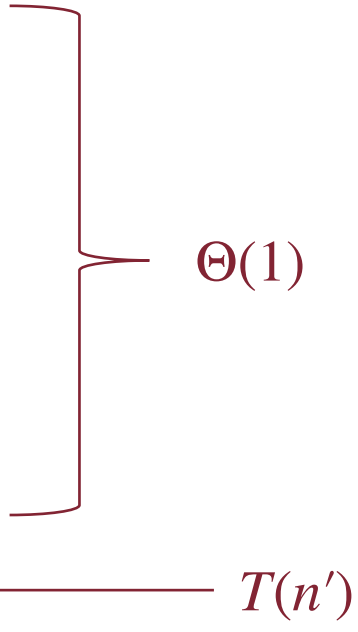
Funzione Heapify

Esempio:



Funzione Heapify

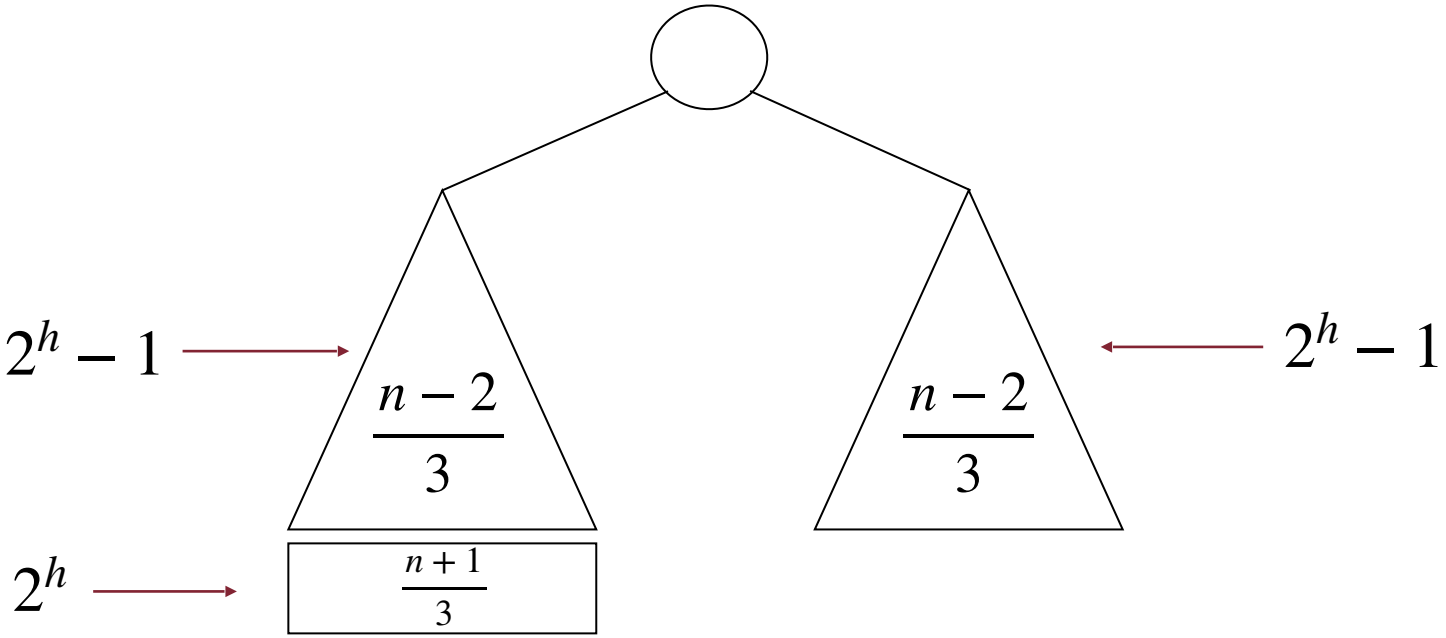
```
def Heapify (A, i, heap_size):
   L=2*i+1, R=2*i+2
    indice_max = i
   if L < heap_size and A[L] > A[i]:
      indice_max = L
   if R < heap_size and A[R] > A[indice_max]:
      indice_max = R
   if indice_max != i:
      A[i],A[indice_max]=A[indice_max],A[i]
      Heapify (A, indice_max, heap_size)
```



$T(n) = \Theta(1) + T(n')$, dove n' è il numero di nodi del sottoalbero che ha più nodi.

Funzione Heapify

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



Quindi l'equazione di ricorrenza diventa $T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$, che ha soluzione:

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

Funzione Buildheap

La funzione Buildheap serve per trasformare qualunque vettore contenente n elementi in uno heap, chiamando ripetutamente Heapify sugli opportuni nodi dello heap.

Osservazioni:

- poiché Heapify presuppone che entrambi i sottoalberi della radice siano heap, deve essere chiamata scorrendo l'albero per livelli dal basso verso l'alto (quindi, sul vettore, da destra a sinistra)

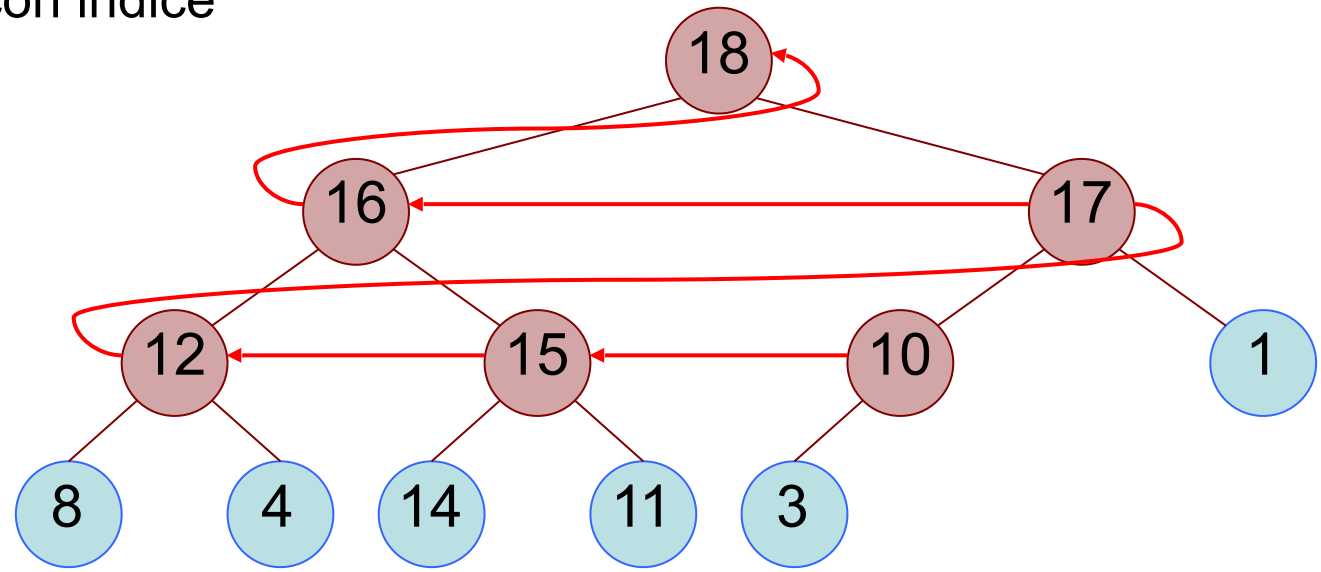
- ogni foglia è già uno heap, quindi basta chiamare Heapify a partire dal nodo interno più a destra, che ha indice:

$$\left\lfloor \frac{n}{2} \right\rfloor - 1$$

Funzione Buildheap

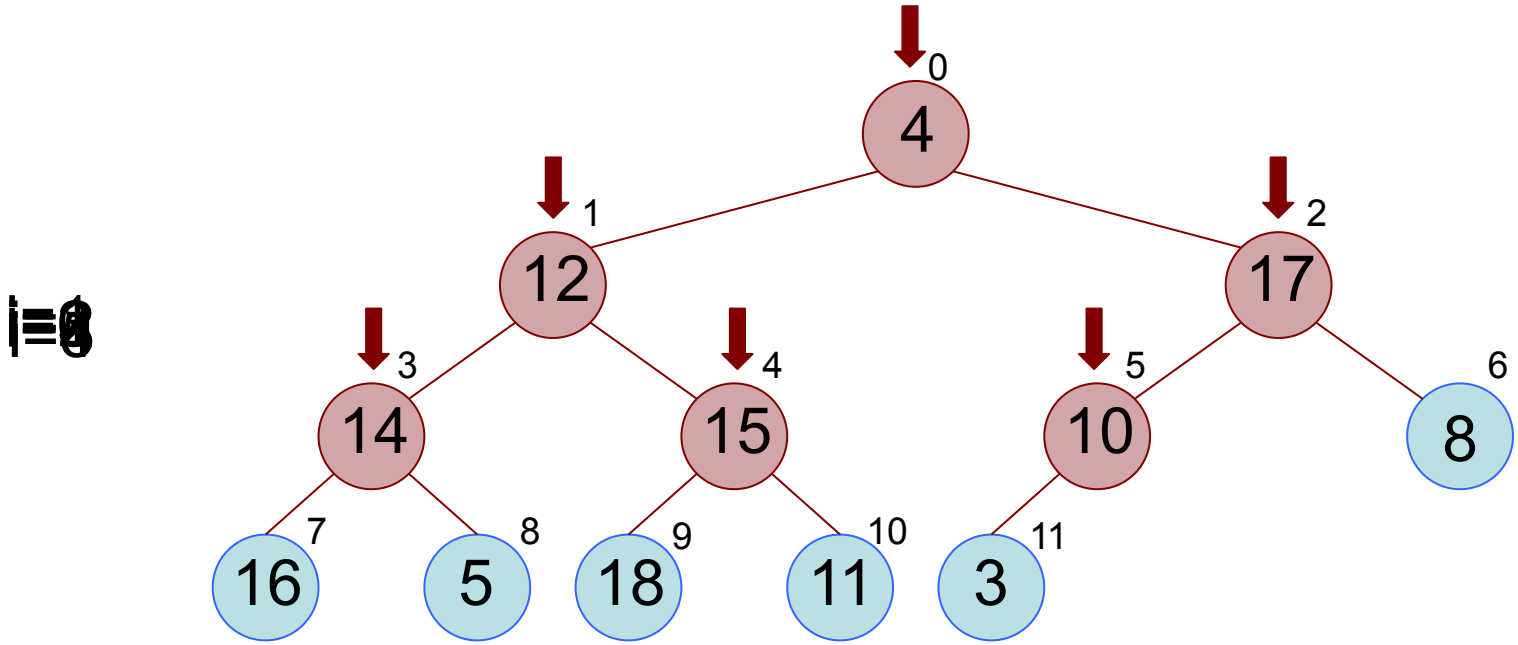
```
def Build_heap (A):
   for i in reversed(range(len(A)//2)):
       Heapify (A, i, len(A))
```

In questo esempio di $n = 12$, Buildheap chiama Heapify sui nodi con indice 5, 4, 3, 2, 1, 0:



Funzione Buildheap

```
def Build_heap (A):
   for  i in reversed(range(len(A)//2)):
      Heapify (A, i, len(A))
```



Funzione Buildheap

```
def Build_heap(A):
   for i in reversed(range(len(A)//2)):
      Heapify(A, i, len(A))
```

La funzione Build_heap effettua $\Theta(n)$ chiamate di Heapify, che sappiamo avere ciascuna costo $O(\log\ n)$, quindi:

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

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

Funzione Buildheap

Mostriamo che $T(n) = O(n)$.

- Considera un heap di altezza h. Nell'heap a livello i ci sono al più 2^i nodi quindi il numero di nodi dell'heap che sono radice di un sottoalbero di altezza i sono $2^{h-i} = \frac{2^h}{2^i} = \frac{2^{h+1}}{2^{i+1}} \leq \left\lceil \frac{n}{2^{i+1}} \right\rceil$
- Il tempo richiesto da Heapify applicata ad un nodo che è radice di un sottoalbero di altezza *i* è $O(i)$ per quanto già detto;

Quindi possiamo scrivere:

$$T(n) = \sum_{i=1}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{i+1}} \right\rceil O(i) = O\left(\frac{n}{2} \sum_{i=1}^{\lfloor \log n \rfloor} \frac{i}{2^i} \right)$$

ricorda che per $x < 1$ vale $\sum_{i=1}^{\infty} i \cdot x^i = \frac{x}{(1-x)^2}$ e quindi $\sum_{i=1}^{\infty} \frac{i}{2^i} = \sum_{i=1}^{\infty} i \left(\frac{1}{2}\right)^i = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2 = \Theta(1)$

Otteniamo quindi : $T(n) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right) = O\left(\frac{n}{2} \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h \right) = O(n) \cdot \Theta(1) = O(n)$

Funzione Heapsort

Vediamo ora il funzionamento di Heapsort.

- Trasforma un vettore *A* di dimensione *n* in un heap (di *n* nodi), mediante Build_heap.
- Ora il max del vettore è in $A[0]$ e, per metterlo nella corretta posizione dell'ordinamento, basta scambiarlo con $A[n-1]$.
- La dimensione 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 $A[0]$ col penultimo elemento;
- Riapplica il procedimento riducendo via via la dimensione dell'heap a $n-2$, $n-3$, ecc., fino ad arrivare a 2.

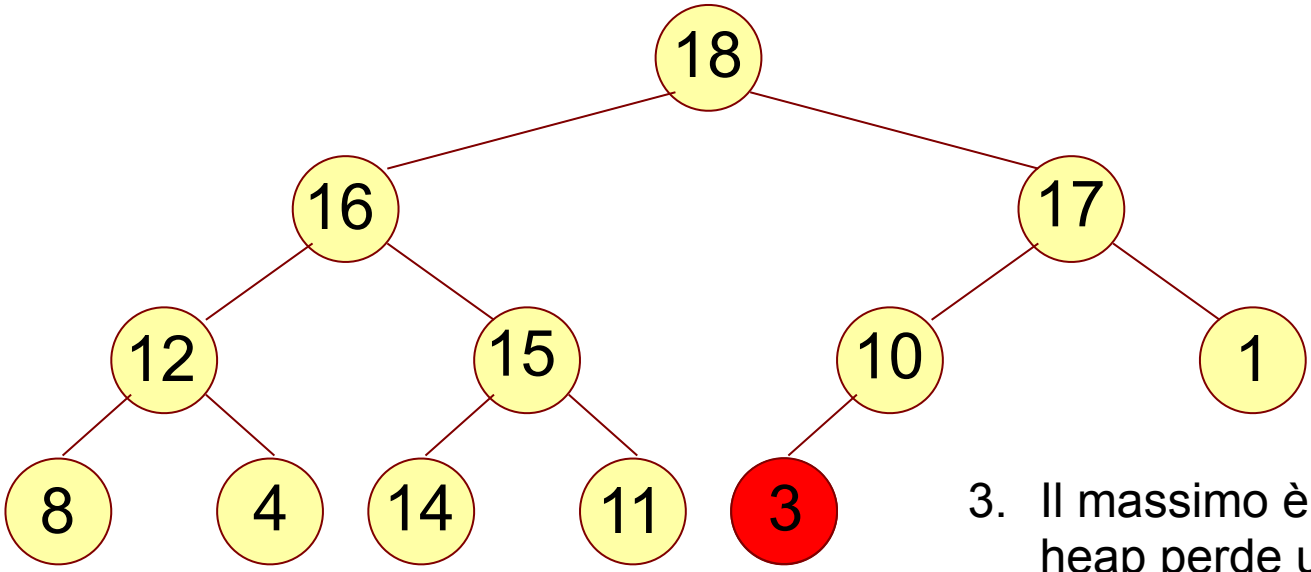
Esempio di Heapsort

Prima iterazione, Heap_size= 12

1. Scambio del massimo:

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

2. Lavoro di Heapify (su 11 elementi):



3. Il massimo è a posto, lo heap perde un elemento

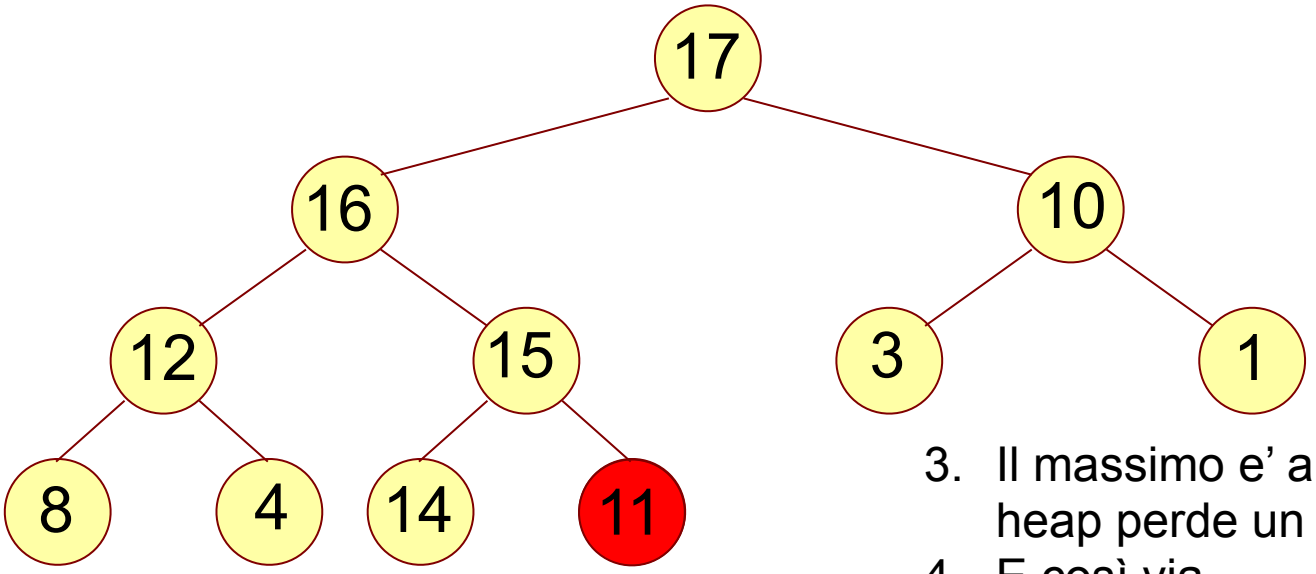
Esempio di Heapsort

Seconda iterazione, Heap_size= 11

1. Scambio del massimo:

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

2. Lavoro di Heapify (su 10 elementi):



3. Il massimo e' a posto, lo heap perde un elemento
4. E così via…

Heapsort

Vediamo ora lo pseudocodice di Heapsort:

```
def Heapsort(A):
    Build_heap(A)                                  O(n)
    for x in reversed(range(1,len(A))):            (n-1) iterazioni
        A[0], A[x] = A[x], A[0]                    Θ(1)
        Heapify(A, 0, x-1)                         O(log n)
```

$$T(n) = O(n) + (n-1)O(\log n) = O(n \log n)$$

Corso di laurea in Informatica
Introduzione agli Algoritmi
Didattica blended

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

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.

Esercizi

- Dimostrare che un heap con n nodi ha esattamente $\left\lceil \frac{n}{2} \right\rceil$ foglie.

- Sia $x < 1$. Sapendo che $\sum_{h=0}^{\infty} x^h = \frac{1}{1-x}$ dimostrare che $\sum_{h=0}^{\infty} h \cdot x^h = \frac{x}{(1-x)^2}$