In questa lezione

- **Heapsort**
  - **ordinamento con complessità, nel caso peggiore, O(nlogn)**

[CLRS01] cap. 6 da pag. 106 a pag. 114

Paternità

L'heapsort è stato pubblicato da J. W. J. Williams nel 1064.
Pochi mesi dopo Robert Floyd ne pubblicava una versione migliorata, in loco, e in cui propone la funzione che trasforma un array qualsiasi in un max-heap in O(n).

Robert Floyd

(1936 - 2001)

Robert Floyd è anche noto per l'algoritmo, chiamato appunto Floyd-Warshal, che trova tutti i più corti cammini in un grafo, per l'algoritmo che trova i cicli in una sequenza e infine per la logica Floyd-Hoare per la verifica della correttezza dei programmi.
E' stato insignito del premio Turing nel 1978.

Costruire un Max-Heap 1

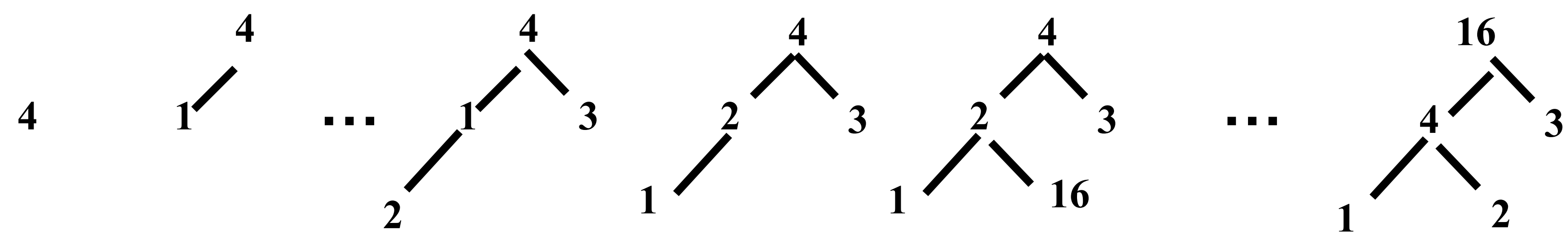
In una struttura dati in cui si dispone di un metodo per inserire nuovi oggetti sembra naturale usarlo come costruttore:

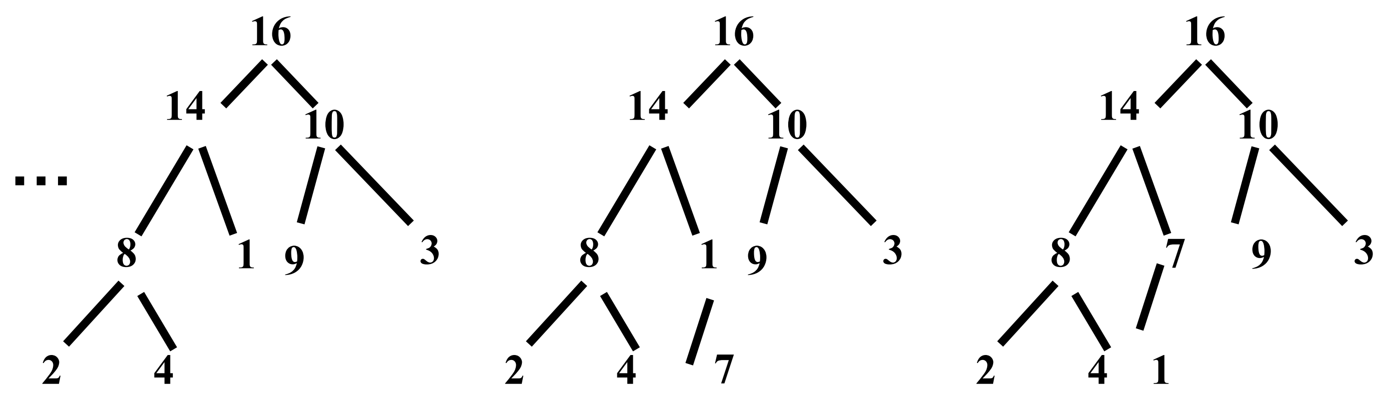
```
Costruisci-Max-Heap(A)
input: A è un array
postc: A è un max-Heap
    A.heapsize = 1
    i=2
    while i ≤ A.length do
    A[1...i-1] è un maxheap.
          Max-Heap-insert(A,A[i])
```

A=

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

Complessità ?





Complessità

L' inserimento di un elemento in un max-heap di altezza h costa O(h). Poiché inseriamo n elementi, maggiorando con lg n l'altezza di ogni albero intermedio si conclude che la complessità è O(n lg n).

Facendo un calcolo più dettagliato il risultato non cambia:
per i 2 nodi di livello 1 si ha al più un costo 2*c,
per i 2^2 nodi di livello 2 si ha al più un costo 2^2 *2c
per i 2^3 nodi di livello 3 si ha al più un costo 2^3 *3c
…
per i 2^h nodi di livello h si ha la più un costo 2^h *hc

Si può fare meglio?

Quindi bisogna valutare la somma:

$$c(1*2^1 + 2*2^2 + 3*2^3 + \ldots + h2^h) =$$

$$c \sum_{1 \le i \le h} i2^i = O(h2^h) = O(n \lg n)$$

Max-Heapify (A,i) può essere utilizzata per costruire un max-heap a partire da un array qualunque di n elementi in O(n).

Max-heapify

Max-Heapify (A,i)

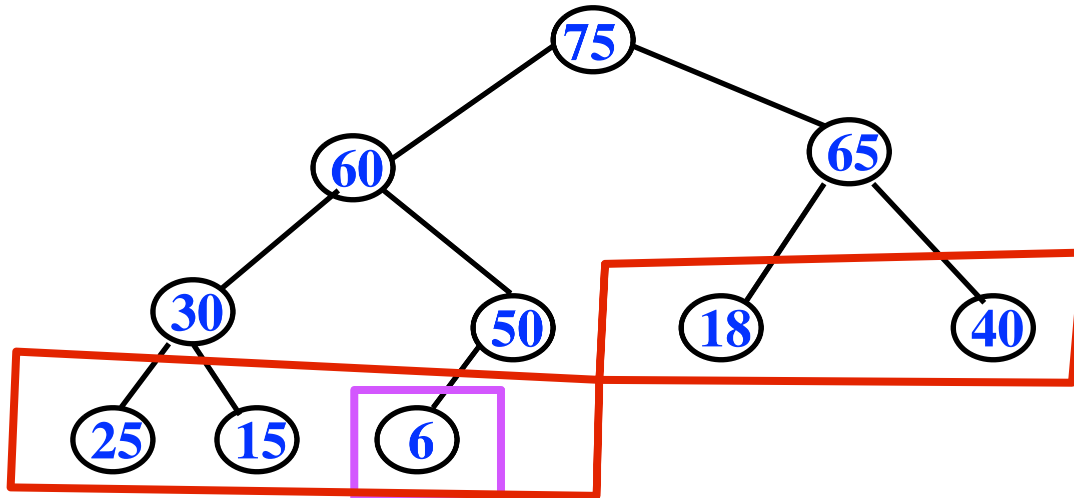
Input: A è un array e i è un indice

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
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]
                  Max-Heapify (A,max)
```

Costruire un Max-heap 2



Notiamo che in un albero quasi completo la foglia più a destra sull'ultimo livello è il nodo n-simo, quindi suo padre, $\lfloor n/2 \rfloor$, è l'ultimo, da sinistra, nodo non foglia del livello precedente.

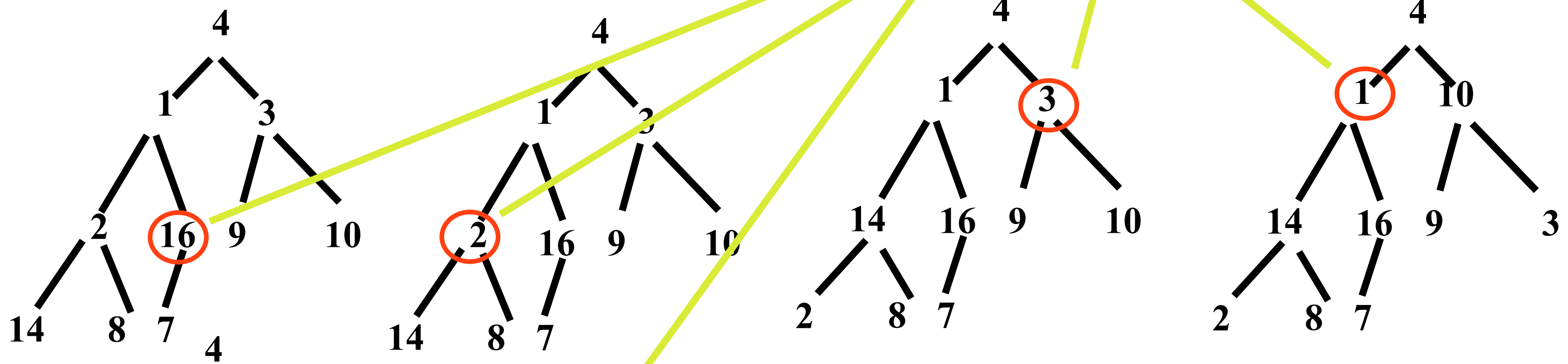
Equivalentemente le foglie sono i nodi $\lfloor n/2 \rfloor + 1$,..., n.

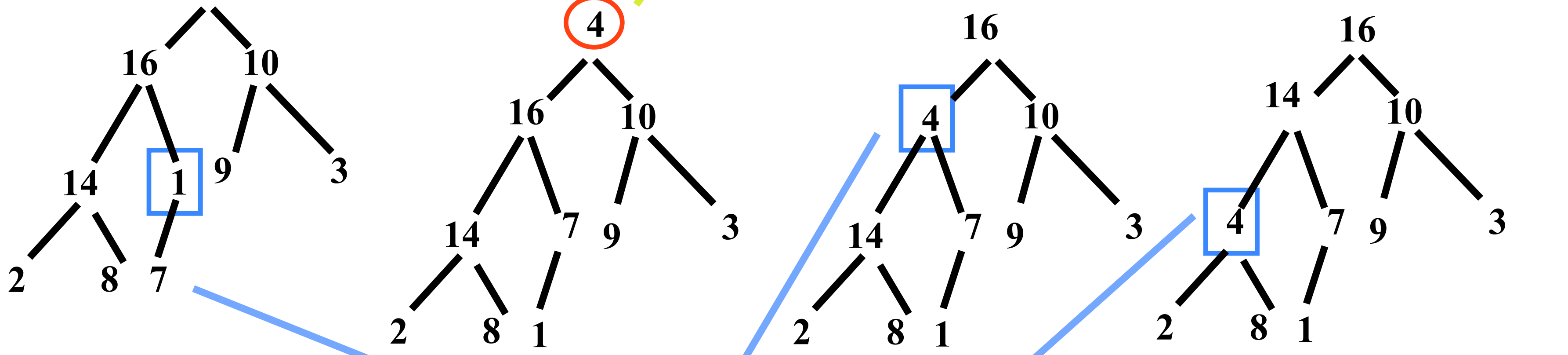
I nodi foglia sono banalmente dei max-heap, quindi ai loro padri si può applicare la Max-heapify, che li rende radici di max-heap. Quindi di nuovo ai loro padri si può applicare la Max-heapify, …

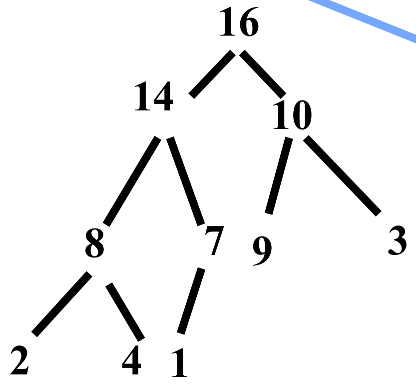
A=

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

chiamate di Max-heapify







chiamate ricorsive Max-heapify

n = A.length

Build-Max-Heap

Build-Max-Heap (A)

▷ **Build-Max- Heap trasforma un array A,dato in input, in un max-heap.**

postc: A è un max-heap

A.heapsize = A.length

Partiamo da $\lfloor n/2 \rfloor$, i nodi da $\lfloor n/2 \rfloor$ +1 a n sono radici di max-heap. Quindi si applica la Max-Heapify risalendo nell'albero dal nodo $\lfloor n/2 \rfloor$ fino alla radice, percorrendo ogni livello da destra verso sinistra.

n = A.length

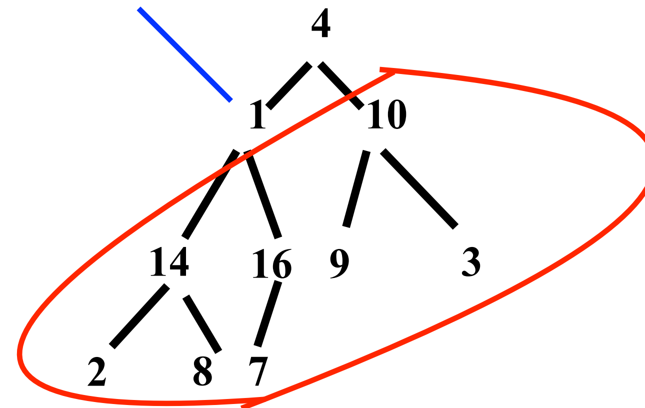
Build-Max-Heap

Build-Max-Heap (A)

▷ Build-Max- Heap trasforma un array A,dato in input, in un max-heap.

postc: A è un max-heap

A.heapsize = A.length



Nel ciclo di risalita dell'albero quando si considera il nodo i ogni nodo i+1,..., n è radice di un max-heap. Quindi la chiamata di Max-Heapify sul nodo i è legittima perché rispetta la precondizione di applicazione del metodo e rende anche i radice di un max-heap.

Build-Max-Heap

Build-Max-Heap (A)

▷ Build-Max- Heap trasforma un array A,dato in input, in un max-heap.

postc: A è un max-heap

n = A.length

A.heapsize = A.length

for i = ⌊heapsize[A]/2⌋ downto 1 do

ogni nodo i+1,..., n è radice di un max-heap

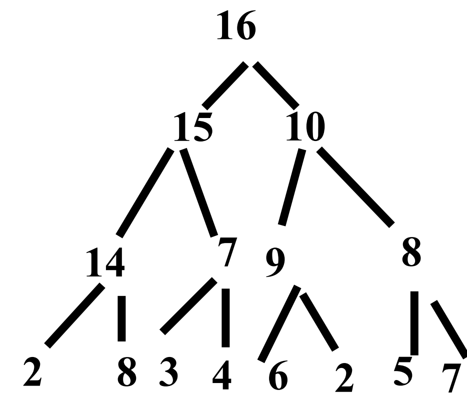
    Max-Heapify (A,i)

Max-heapify ha una complessità O(h) se applicata a un nodo di altezza h. Poiché la applichiamo n volte, maggiorando con lg n l'altezza di ogni albero intermedio si conclude che la complessità è O(n lg n). Ma ora il calcolo più preciso ci dà un risultato migliore.

Build-Max-Heap (A)

A.heapsize = A.length

for i = heapsize[A]/2 downto 1 do Max-Heapify (A,i)



Quante chiamate in tutto di Max-heapify al più si fanno?

Sia $n = 2^{h+1}-1$, ci sono 2^h foglie sulle quali non ci sono chiamate,
2^{h-1} nodi di livello h-1 sui quali si esegue al più 1 chiamata
2^{h-2} nodi di livello h-2 sui quali si eseguono al più 2 chiamate
2^{h-3} nodi di livello h-3 sui quali si eseguono al più 3 chiamate,
…
$2^0 = 1$ nodi di livello 0 sul quale si eseguono al più h chiamate

Build-Max-Heap: analisi 2

Quindi <u>il numero delle chiamate</u> in totale è al più

$1*2^{h-1} + 2*2^{h-2} + 3*2^{h-3} + \ldots + h2^{0} =$

$2^{h} (1*2^{-1} + 2*2^{-2} + 3*2^{-3} + \ldots + h2^{-h})$

$= O(2^{h})$, perché

$$\sum_{1 \leq k \leq h} k2^{-k} \leq \sum_{0 \leq k \leq \infty} k(1/2)^{k} = 2 = O(1)$$

Poiché ogni singola chiamata ha un costo costante il <u>costo totale</u> è

$$O(2^{h}) = O(n)$$

Se il max-heap non è un albero completo ha meno nodi e quindi questo limite superiore vale in generale.

Heapsort: l'idea

1. trasforma l'array dato in un max-heap
2. porta nell'ultima posizione del maxheap il primo elemento (è il massimo!), scambiandoli
3. "scarta" l'ultimo elemento
4. ripristina l'heap sugli elementi rimanenti,
5. ritorna al punto 2

Quindi si ha un ciclo che inizia sull'ultimo elemento di un maxheap A con n elementi e con

postc: A è ordinato, in ordine crescente.

In una fase intermedia, quando considero l'elemento i-simo per lo scambio,

gli elementi A[1.. i] costituiscono un maxheap su i elementi di A e gli elementi A[i+1... n] sono più grandi di quelli in A[1.. i] e sono ordinati in ordine crescente

Heapsort: lo pseudocodice

A.length= n

Heapsort(A)

Build-Max-Heap(A)

for i = A.length downto 2 do

gli elementi A[1.. i] costituiscono un maxheap su i elementi di A e gli elementi A[i+1... n] sono più grandi di quelli in A[1.. i] e sono ordinati in ordine crescente

scambia A[1] e A[i]

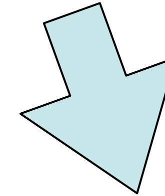
A.heap-size = A.heap-size - 1

Max-Heapify (A,1)

Heapsort: esempio

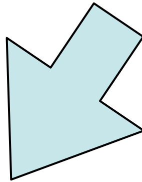
4	1	3	2	16	9	10	14	8	7
1	2	3	4	5	6	7	8	9	10

Build-Max-Heap



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

scambio



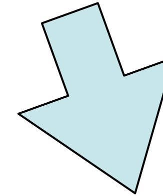
1	14	10	8	7	9	3	2	4	16
1	2	3	4	5	6	7	8	9	10

Max-Heapify

Heapsort: esempio

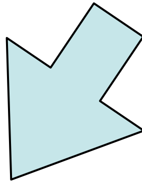
1	14	10	8	7	9	3	2	4	16
1	2	3	4	5	6	7	8	9	10

Max-Heapify



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

scambio



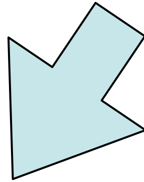
1	8	10	4	7	9	3	2	14	16
1	2	3	4	5	6	7	8	9	10

Max-Heapify(A,1)

Heapsort: esempio

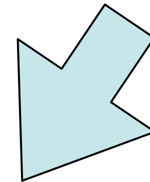
Max-Heapify(A,1)

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



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

scambio



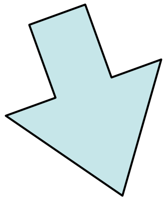
2	8	9	4	7	1	3	10	14	16
1	2	3	4	5	6	7	8	9	10

Max-Heapify(A,1)

Heapsort: esempio

Max-Heapify(A,1)

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



...

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

Heapsort: analisi

n = A.length

```
Heapsort(A)
Build-Max-Heap(A)                    O(n)
for i = A.length downto 2  do
    scambia A[1] e A[i]              Θ(1)
    A.heap-size = A.heap-size - 1    Θ(1)
    Max-Heapify (A,1)                O(log n)
```

Quindi, nel caso peggiore, per l'Heapsort T(n) = O(n log n)