# Appunti 15-10-24 — Heap — 15/10/24

Minheap -> priorità numeri piccoli

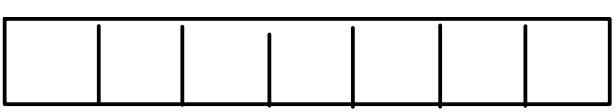
Maxheap -> priorità numeri grandi

Ogni nodo ha sempre priorità più alta o uguale rispetto ai suoi figli.
La maggior parte delle operazioni impiega $O(\log n)$ poiché è un albero bilanciato tranne l'ultimo livello in cui i nodi si "ammassano" su un solo lato.

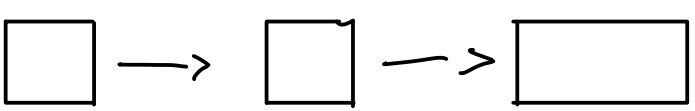
## Heap struttura dati efficiente

La sua rappresentazione fisica è un array.
Il livello fisico è un array, mentre il livello astratto è un albero.
Caso diverso per l'array, dove rappresentazione fisica e astratta coincidono.

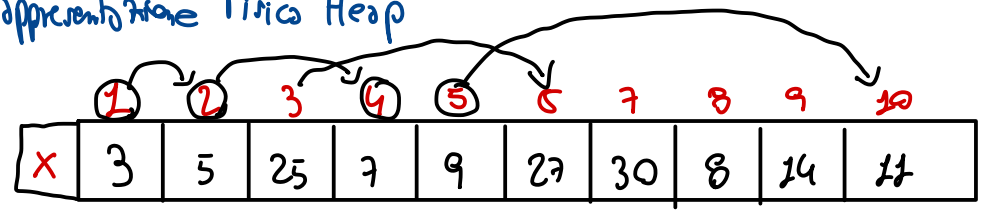
## Array



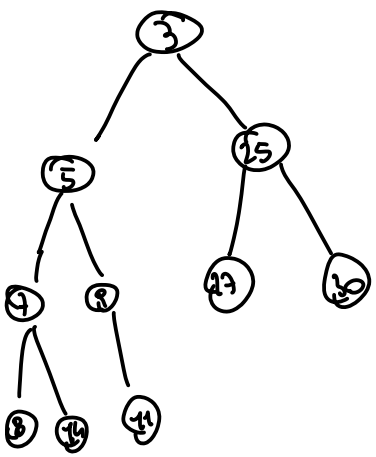
## Lista (Rappresentazione astratta)



# Rappresentazione Fisica Heap

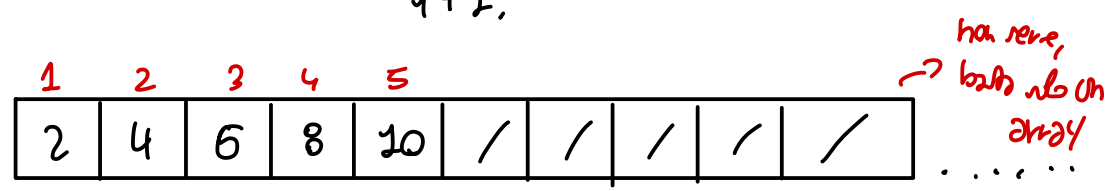



Min Heap




Ordinandoli così però perdiamo le informazioni che avevamo nell'albero. Un modo sarebbe allocare un altro array e tenere le informazioni che ci servono.

I fratelli in questa configurazione sono cattivi. Basta che trovo il figlio sinistro, trovando anche il destro, visto che la sua posizione sarà $i+1$.




L'albero è "orientato" da sinistra e destra.

Se trovo il figlio sinistro => ho trovato il figlio destro.

Figlio destro =/=> che trovo il figlio sinistro.

Rappresentare tutto con 2 array solo ha un vantaggio di usare poca memoria.

## Trovare i figli con un unico array

Se voglio trovare informazioni padre-figlio, basta un solo array.

padre $i \rightarrow 2i$.

I puntatori sono IMPLICITI nella struttura che sto utilizzando.

## Pseudo-Codice

```
LEFT(x)
return x->getleft();
```

---

PSEUDOCODICE TROVARE I FIGLI

```
LEFT(i)          RIGHT(i)
return 2i;       return 2i+1;
```

In questo modo l'operazione è veloce, e gli elementi dell'array sono contigui.

Conviene per queste operazioni, indice 0 cella vuota ed inizio da indice 1.

PSEUDO-CODICE TROVARE IL PARENT

```
PARENT(i)
return ⌊i/2⌋;   (floor di i/2)
```

MOLTIPLICAZIONE BIT A BIT

```
LEFT(i)          RIGHT(i)
return (i<<1);   return (i<<1|1);
```

$101 \rightarrow 1010$ OR $0001$ $= 1011$

Piuttosto di moltiplicare, usiamo i bit per shiftare i numeri binari e moltiplicare per 2.

PARENT(i)

return (i >> 1);

1010 >> 2 -> 101

1011 >> 2 -> 101

> dati e dispari non importa, il LSB viene perso.

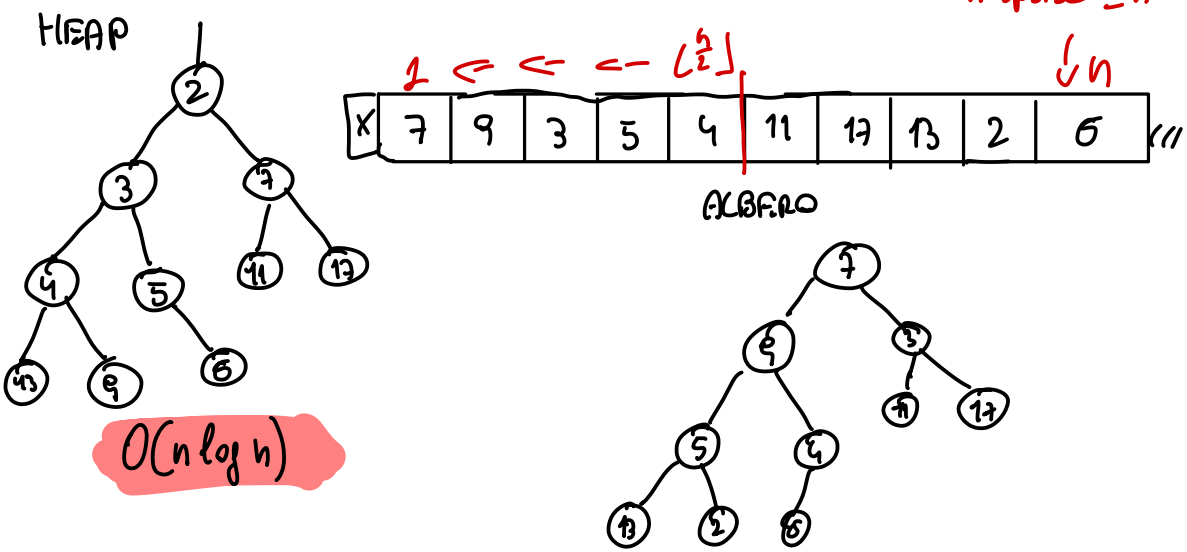
L'Heap è quello che conosciamo a livello astratto.

Fisicamente, l'Heap è un array in cui è possibile ricostruire la struttura astratta grazie agli indici. (Elementi ordinati da sin. a destra)

## Come costruire un Heap?

ESEMPIO -> $S = < 7, 9, 3, 5, 4, 11, 17, 13, 2, 6 >$

Heapsize $\leq n$




In un albero generico, possono esserci sottoheap. In questo caso abbiamo 5 foglie (nnodi/2).

Le foglie sono sempre Heap, chiamo heapify dai livelli più bassi fino a livelli più alti.

Inizio a percorre le foglie da sinistra a destra, trovando il primo nodo in cui chiamare la procedura. $\frac{n}{2}$ è il primo elemento a non avere una foglia nell'array.

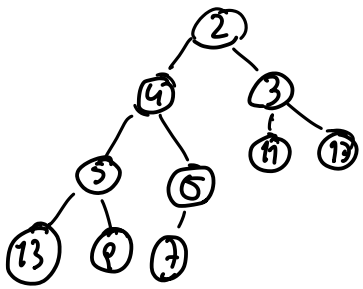
## PSEUDO-CODICE BUILD MIN-HEAP

BUILD_MIN-HEAP (A, n)

FOR i ← $\lfloor \frac{n}{2} \rfloor$ DOWN TO 1 DO

  HEAPIFY (A, i); $O(n)$

DOPO PROCEDURA:




INP:

Se costruisco un heap inserendo elementi → $O(n \log n)$;

Se costruisco un heap da un array → $O(n)$.

## PSEUDO-CODICE HEAPIFY IN UN ARRAY

HEAPIFY (A, i)

l ← left (i)  cicla un indice positivo

r ← right (i)

min ← i

if (l ≤ Heapsize AND A[l] < A[min]) then min ← l

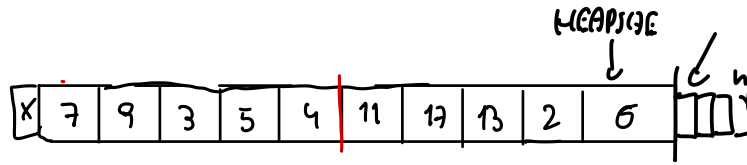
if (r ≤ Heapsize AND A[r] < A[min]) then min ← r

if (min ≠ i) then {swap(A, i, min); Heapify (A, min); }

# Inserimento

Se con un Heap c'era il problema di capire come far avere il numero "nel posto giusto", ora nell'array è molto più semplice. L'elemento in più verrà messo in posizione HeapSize + 1.

## EXTRACT-MIN(A)




```
SWAP(A, 1, HEAPSIZE);
HEAPSIZE ← HEAPSIZE - 1;
HEAPIFY(A, 1);
```

## INSERT(A, K)

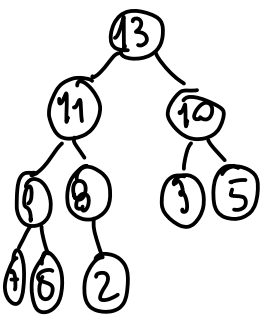
```
A[HEAPSIZE+1] ← K;
HEAPSIZE ← HEAPSIZE + 1;
i ← HEAPSIZE ;   elemento appena inserito
P ← PARENT(i) ;   padre dell'elemento nuovo
WHILE (P ≠ 0 AND A[P] > A[i])   ciclo per inserirlo e mantenere l'ord. parziale
    SWAP(A, i, P);
    i ← P;
    P ← PARENT(i);
```

ORDINAMENTO ARRAY

A

| 7 | 13 | 5 | 6 | 2 | 3 | 10 | 9 | 11 | 8 |
|---|---|---|---|---|---|---|---|---|---|

BUILD MAX-HEAP




PROCEDURA

| 13 | 11 | 10 | 9 | 8 | 3 | 5 | 7 | 6 | 2 |
|---|---|---|---|---|---|---|---|---|---|

PROCEDURA EXTRACT-MAX

| 11 | 9 | 10 | 7 | 8 | 3 | 5 | 2 | 6 | 13 |
|---|---|---|---|---|---|---|---|---|---|

Possiamo ordinare un array tramite l'extract max della procedura heap.

RISULTATO FINALE .....

| 2 | 3 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 13 |
|---|---|---|---|---|---|---|---|---|---|

Algoritmo HeapSort(A, n)

Come il selection sort ma in tempo $\Theta(n \lg n)$.

HeapSort(A, n)

BUILD-MAX-HEAP(A, n) ← $O(n)$

for i ← 1 to n-1 do

EXTRACT-MAX(A) ← $O(\lg n)$

Abbiamo usato una struttura dati per creare algoritmi molto efficienti a volte anche "banali".

Avendo l'Heap, il massimo lo trovi in tempo logaritmico.

L'HeapSort lavora in loco. L'Heap implementa l'albero direttamente dentro l'Array. L'HeapSort si potrebbe implementare anche in maniera stabile. HeapSort è migliore del mergesort.