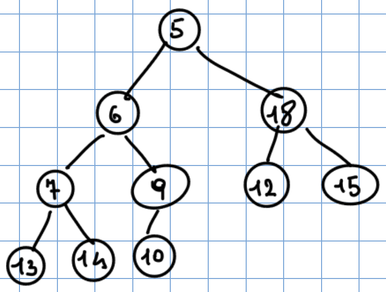


Min Max

Extract Min

Insert

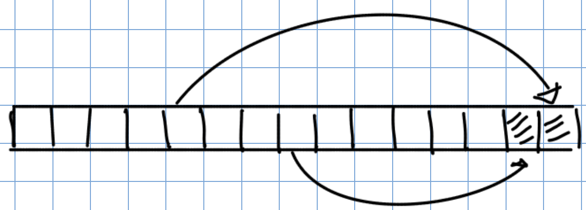
Decrease - Key

Heapify(x) $\rightarrow T(n) = T\left(\frac{2}{3}n\right) + 1$

Come l' heap ci semplifica la vita

Ordinamento:

- insertion Sort $O(n^2)$
- selection Sort $\Theta(n^2)$
- quick Sort $O(n^2)$
- Merge Sort $\Theta(n \log n)$

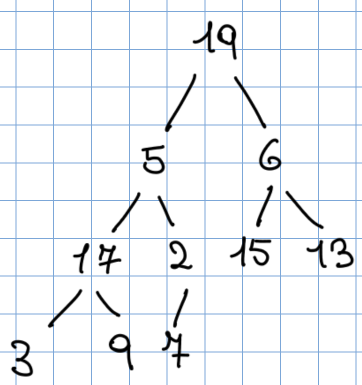


nel selection sort con un array il problema è il tempo per trovare il massimo

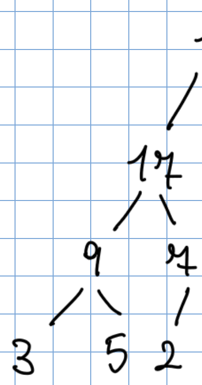
19	5	6	17	2	15	13	3	9	7

richiamando Build-max-heap $O(n)$ costruiamo un heap del tipo

heap iniziale

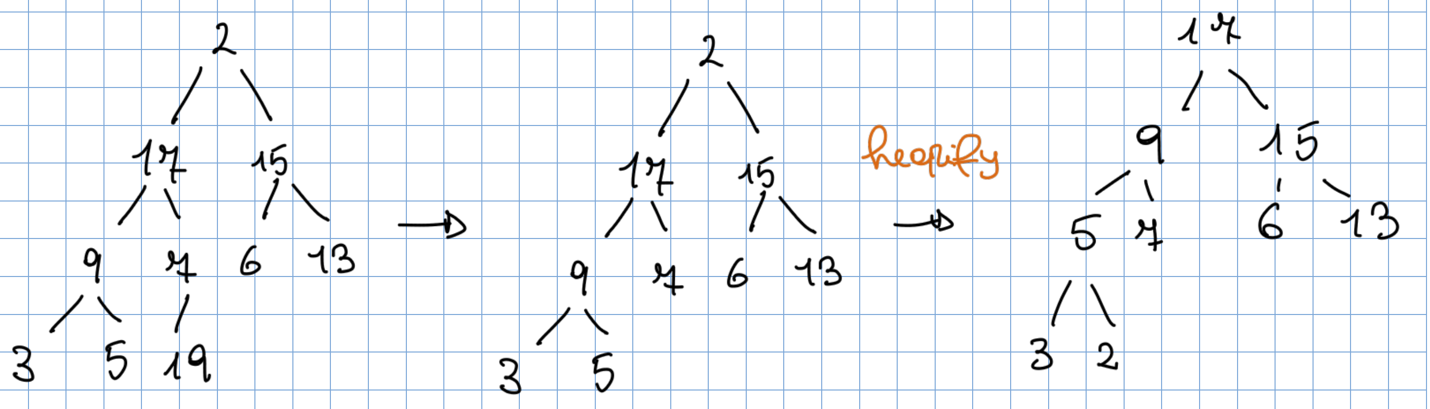


heap dopo Build-max-heap



resta comunque un array in memoria ne accediamo in modo diverso

Per prendere il massimo scambiamo l'ultimo nodo con la root, prendo il nuovo ultimo nodo e chiamo heapify



il max viene inserito alla fine del nostro array, facendo questa cosa per ogni elemento avrò l'array ordinato

da questa idea nasce l' heap sort

HeapSort (A, n)

  build-max-heap (A, n)

  for i ← to n − 1 > 0

    extractMax (A) → $\begin{cases} \text{SWAP}(A, 1, n-1) \\ n \leftarrow n - 1 \\ \text{Heapify}(A, 1) \end{cases}$

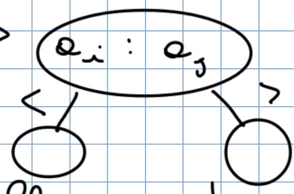
$$O(n \log n) \rightarrow \Theta(n \log n)$$

praticamente come il merge sort e consuma anche meno memoria

Un algoritmo che fa dei confronti non potrà mai scendere sotto $\Theta(n \log n)$

Dimostrazione: la dim viene fatta usando un albero di decisione

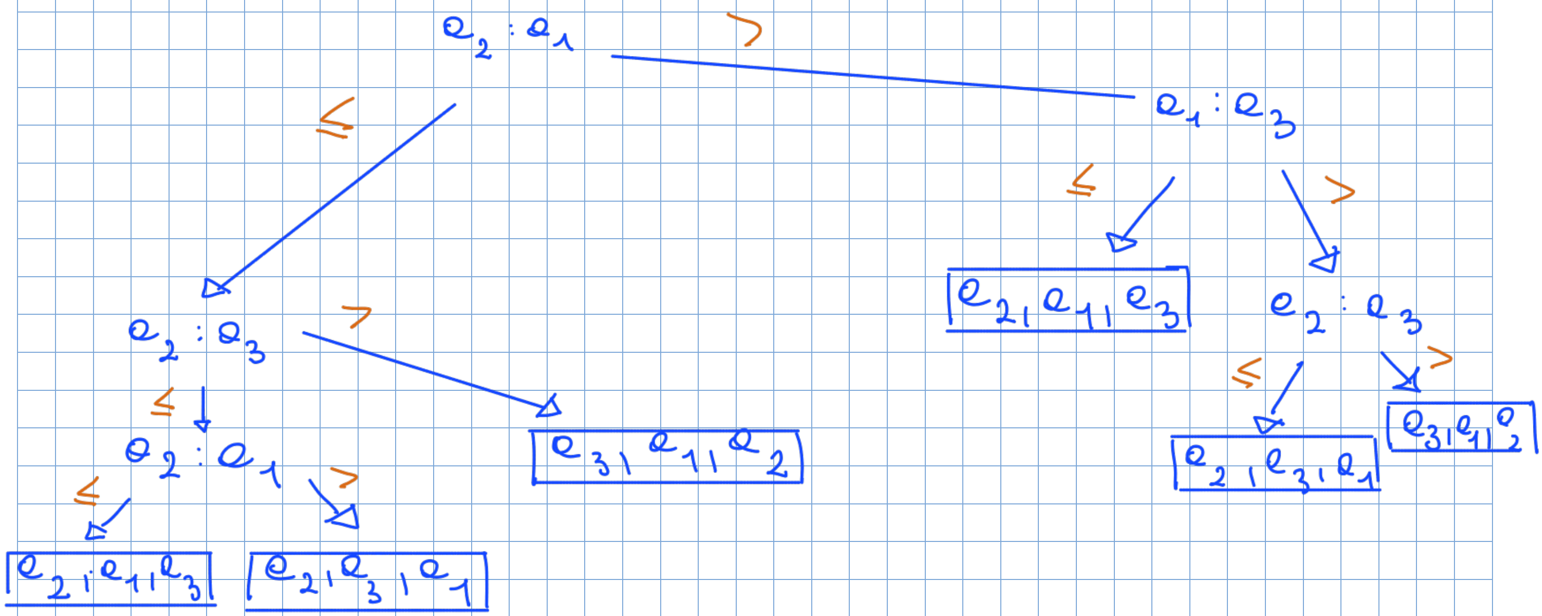
Capitolo 8 del libro



albero usato per prendere decisioni

es:

A: | a_1 | a_2 | a_3 |



il caso peggiore corrisponde all'altezza di questo albero

Continuo

- $\lg(n!) = \Theta(n \log n)$
- un albero binario di altezza h ha al più 2^h foglie
- foglie: $n!$

albero non completo

$n! \le 2^h$

albero completo

complessità albero di decisione

$n! \le 2^h \rightarrow h \ge \lg(n!) = \Omega(n \log n)$

Per l'analisi fatta sui vari algoritmi sappiamo che $O(n \log n)$ è anche un limite superiore quindi segue la tesi $\Theta(n \log n)$

Problemi degli esami sull'heap

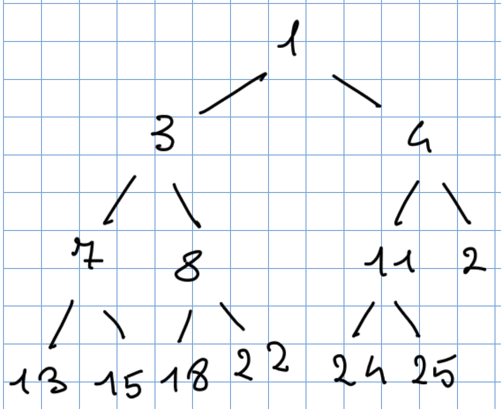
6 settembre 2024

A = [1, 3, 4, 7, 8, 11, 12, 13, 15, 18, 22, 24, 25]

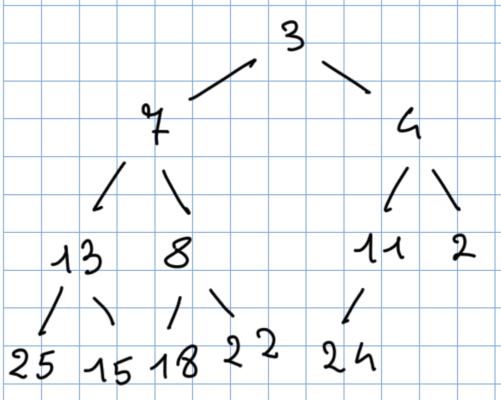
len(A) = 13

configurazione iniziale di un min-heap

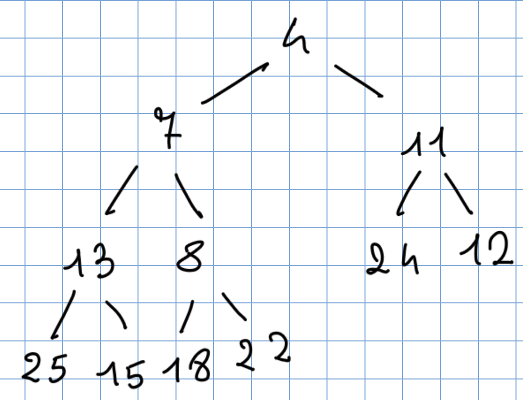
Effettuare 13 estrazioni del minimo e mostrare cosa succede al nostro array



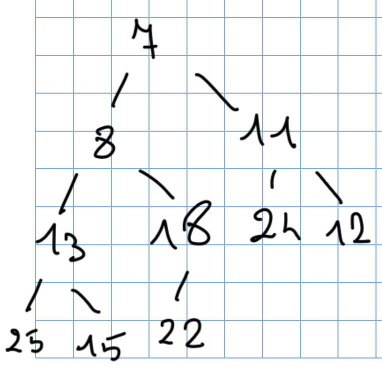
$\boxed{1}$



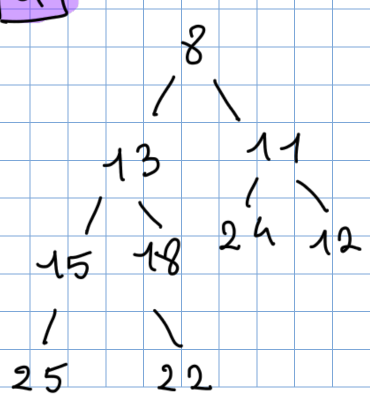
$\boxed{2}$



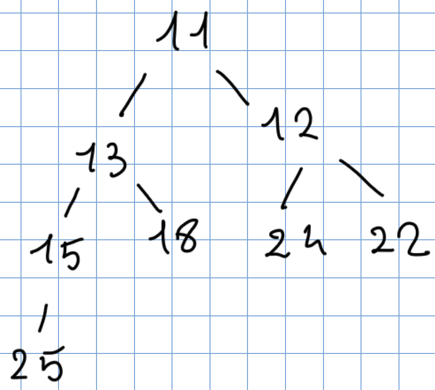
$\boxed{3}$



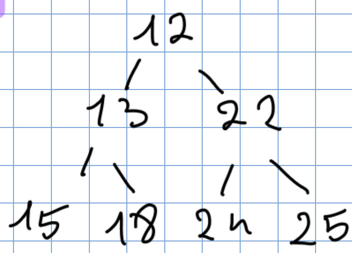
$\boxed{4}$



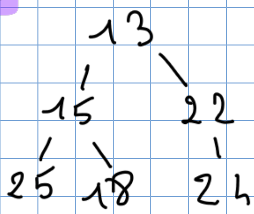
5



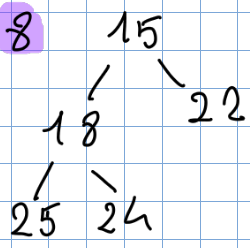
6



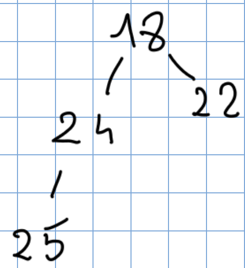
7



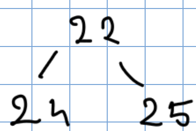
8



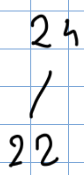
9



10



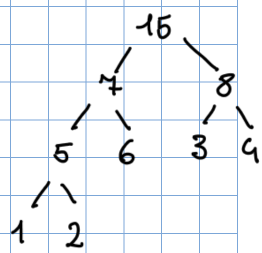
11



12 24

13 $\perp$

14 maggio 2025



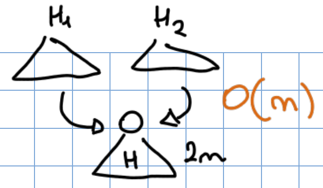
Scrivere la procedura:

```
UpdateKey(H, i, k)
    if i > heapsize return
    if k > H[i] then increaseKey(H,i,k)
    else
        H[i] ← k
        Heapify(H,i)
```

all'esame vanno scritte

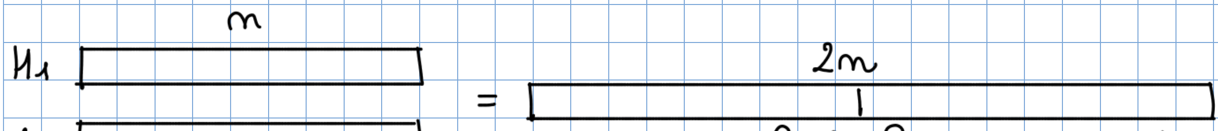
16 giugno 2025

Scrivere la procedura



numero chiavi

heapMerge(H_1, H_2, n)   Deve avere complessità $O(n)$



e poi build-heap su queste

```
heapMerge(H1, H2, n):
    H ← new Array(2n)
    for i ← 1 to n Do:
        H[i] ← H1(i)
        H[i+n] ← H2(i)
    buildMaxHeap(H, 2n)
```

---

1 settembre 2025

$$P = \{ (x_i, y_i) \mid 0 \le i < n \}$$
$$A = (a, b) \qquad k < n$$

input

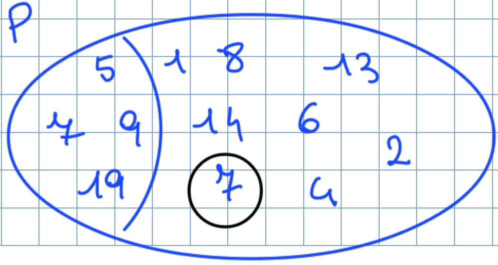
i K punti di P più vicini ad A

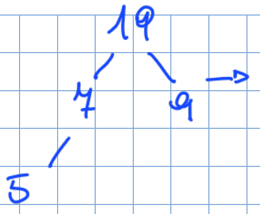
$$\delta(x, y) = \sqrt{(x-a)^2 + (y-b)^2}$$ → formula calcolo distanza

Deve avere una complessità $O(n \log(K))$

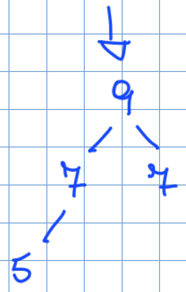
Creo un max-heap con gli elementi temporanei più vicini

es: P $k=4$





confronto 19 con 7 e prendo il più piccolo
poi chiamo heapify



Facendo questa cosa per ogni elemento alla fine dentro il mio heap avrò i K elementi più vicini

---

15 settembre 2025

$L = \{ l_1 - l_2 - l_3 \cdots l_m \}$ sono delle liste ordinate sempre più grandi

$$\sum_{i=1}^{m} |l_i| = n$$

Dobbiamo trovare H una lista ordinata che contiene tutte le altre

Complessità richiesta: $O(n \underbrace{\log m}_{})$ → m sono gli elementi del nostro heap

```
ListMerge(L, m)
  H ← new List()
  A ← new Array(m) → array di liste
  For i ← 1 to m do A[i] ← l_i
  Build-min-heap(A, m, head)
  for i ← to n DO:
    l ← estractMin(A) → estraggo il minimo
    insert(head(l), H) → inserisce l in H nel punto giusto
    deleteHead(l)
    if not isempty(l) then
        Heap-insert(A, l) → se la lista non è vuota questa va reinserita
```

funzione usata per dare priorità basandosi sulla head ritorna la stessa parola una lista

27 ottobre 2025

top ten dei punteggi più alti

caratteristiche del giocatore

H = min-heap di 10 elementi

i = indice al nodo del giocatore

i = -1 se il giocatore non è un top ten

```
topten(H, i, p)
  if i > 0 and H(i) < p then
      H[i] ← p
      i ← Heapify(H, i)
  else
      if (p > H[1]) then
        H[1] ← p
        i ← heapify(H, 1)
```