# Struttura Dati: Min-Heap

## Algoritmi 1 – Lezione 9

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

# Indice

# Motivazione Iniziale

## Domanda

Hai una lista di $n$ numeri e vuoi sempre sapere qual è il più piccolo, anche mentre ne arrivano altri. Come faresti?

Ecco due possibili approcci:

| **Struttura dati** | **Creazione** | **Minimo** | **Inserimento** |
|---|---|---|---|
| **Array ordinato** | $O(n \log n)$ | $O(1)$ | $O(n)$ |
| **Array non ordinato** | $O(1)$ | $O(n)$ | $O(1)$ |

- **Array ordinato:** trovi subito il minimo (è in testa), ma inserire richiede di scorrere l'array.
- **Array non ordinato:** inserisci subito all'ultimo posto, ma trovare il minimo richiede una scansione.

**Domanda:** Possiamo avere il meglio di entrambi i mondi?

# Motivazione Iniziale

**Problema:** trovare rapidamente il minimo, ma anche poter aggiungere elementi senza riordinare tutto ogni volta.

**Soluzione:** una struttura dati chiamata **min-heap**.

**Con un min-heap possiamo ottenere:**

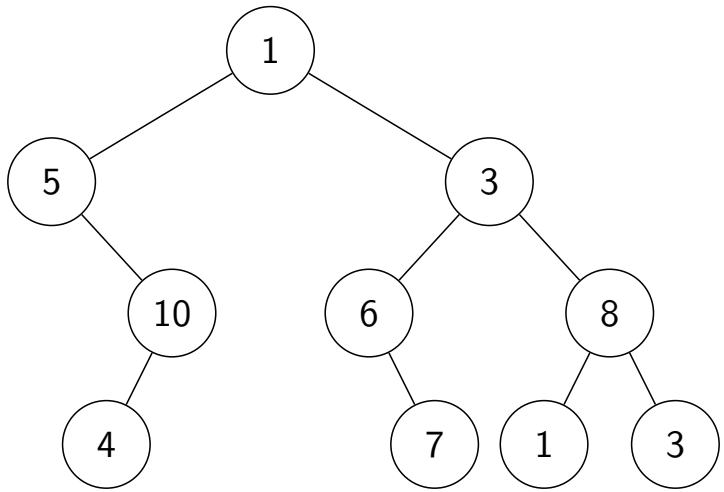
- **Minimo**: $O(1)$
- **Inserimento**: $O(\log n)$
- **Creazione (heapify)**: $O(n)$
- **Estrazione del minimo**: $O(\log n)$

*Un ottimo compromesso tra efficienza nella lettura e aggiornamento dei dati!*

# Cos'è un albero binario (informale)

Un **albero binario** è una struttura dati gerarchica fatta di nodi con valore collegati.

- C'è un nodo iniziale chiamato **radice**.
- Ogni nodo può avere al più **due figli** (figlio sinistro e figlio destro).
- I nodi che non hanno figli si chiamano **foglie**.
- la distanza di un nodo dalla radice è il suo livello
- La **distanza massima** si chiama **altezza** dell'albero (rappresenta il numero massimo di step da fare per raggiungere un nodo dalla radice).

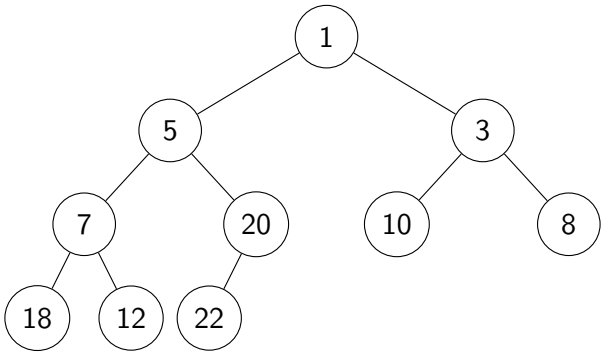



In figura un albero di altezza 3

# Definizione di Min-Heap

Un **min-heap** è un albero binario che soddisfa due proprietà fondamentali:

1. **Proprietà di ordine:**
   Ogni nodo ha valore **minore o uguale** a quello dei suoi figli.
2. **Proprietà di compattezza:**
   L'albero è **completo** o **quasi completo**:
   - Tutti i livelli sono **completamente pieni**, eccetto forse l'ultimo.
   - L'ultimo livello è riempito da **sinistra verso destra**, senza "buchi".




In figura un min-heap di altezza 3.

**Nota:** in un min-heap il minimo è sempre in cima!

# Altezza di un Min-Heap: $O(\log n)$

**Proprietà:** l'altezza $h$ di un min-heap con $n$ nodi è $h = O(\log n)$.

**Dimostrazione:**

- In un min-heap tutti i livelli, tranne al più l'ultimo, sono pieni.
- Detto $i$ un livello dell'albero (con $i = 0$ la radice), ci sono esattamente $2^i$ nodi a quel livello.
- Dunque, il numero totale di nodi $n$ supera la somma dei nodi su tutti i livelli da 0 a $h - 1$:

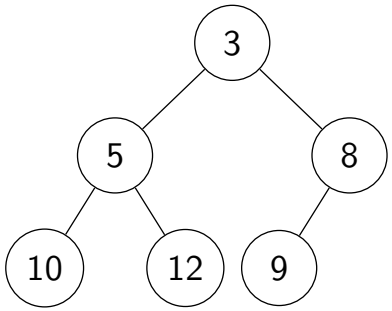
$$n > \sum_{i=0}^{h-1} 2^i = 2^h - 1.$$

- Risolvendo per $h$ si ha $h < \log_2(n + 1)$
- Quindi, per l'altezza vale $h = O(\log n)$.

**Conclusione:** l'altezza cresce al massimo in modo logaritmico rispetto al numero di nodi.

# Rappresentazione con Array

Un min-heap di $n$ nodi può essere rappresentato in modo linearizzato con un **array** di $n$ elementi. Rappresentazione utile per implementare l'heap in modo compatto ed efficiente.




Gli elementi dell'heap compariranno nell'array ordinati per livelli e per ciascun livello ordinati da sinistra a destra.

$$A = [3, 5, 8, 10, 12, 9]$$

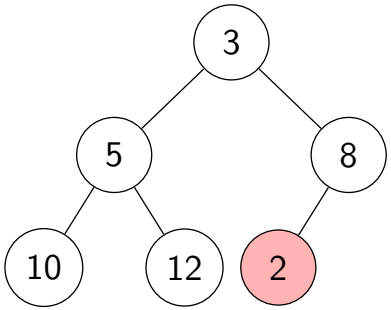
Possiamo ricavare il padre ed i figli del nodo $i$ di valore $A[i]$ come segue:

- Padre di $i$: $(i-1)//2$ se $i > 0$
- Figlio sinistro: $2i + 1$
- Figlio destro: $2i + 2$

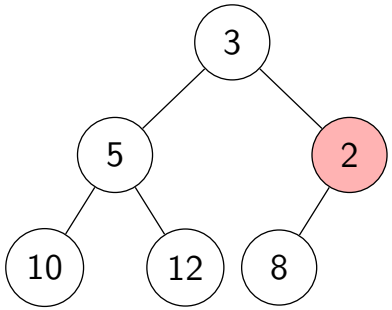
# Inserimento in un Min-Heap

**Step 1:** Inseriamo il nuovo elemento come ultimo elemento dell'heap
**Step 2:** Confrontiamo con il padre e "risaliamo" se necessario

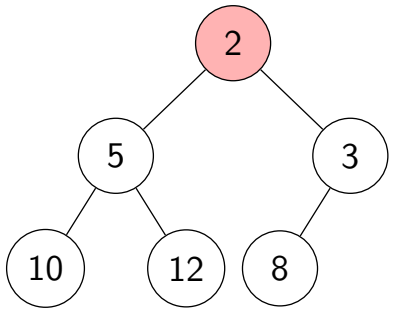
**Esempio: Inseriamo 2**




*Passo 1: il nodo 2 deve risalire* $2 < 8$




*Passo 2: il nodo 2 deve risalire* $2 < 3$




*Passo 3: 2 è diventato radice*

Ogni risalita è un passo: al massimo $\log n$ passi $\rightarrow$ **O(log n)**

# Codice Python: Inserimento (heappush)

La funzione *heappush* inserisce un nuovo elemento in un min-heap rappresentato come array, mantenendo le proprietà dell'heap.

```
def heappush(A, x):
    A.append(x)
    i = len(A) - 1
    while i > 0 and A[i] < A[(i - 1) // 2]:
        A[i], A[(i - 1) // 2] = A[(i - 1) // 2], A[i]
        i = (i - 1) // 2
```

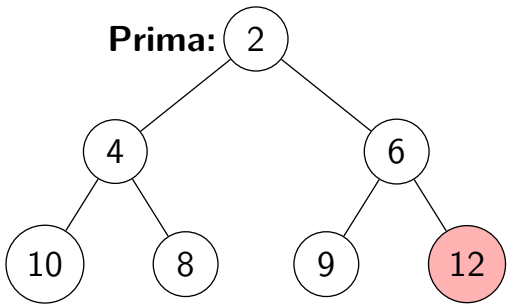
**Esempio:** `A = [3,5,8,10,12]` $\xrightarrow{\texttt{heappush(A,2)}}$ `A = [2,5,3,10,12,8]`

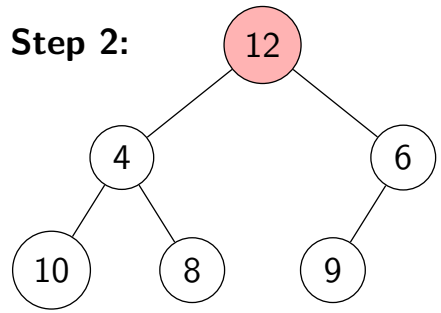
Il valore 2 viene inserito in modo che l'heap resti valido. Complessità: $\mathcal{O}(\log n)$.

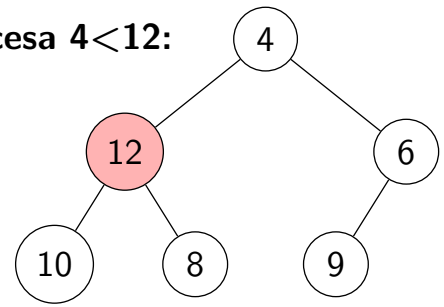
# Estrazione del minimo

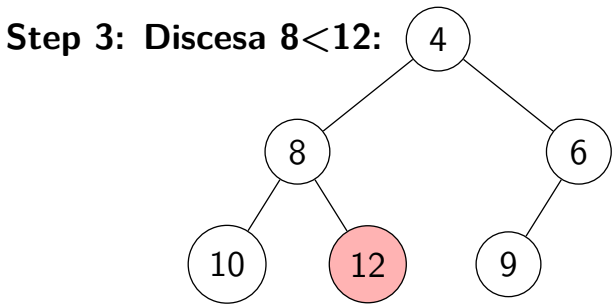
**Step 1:** Rimuovi la radice (minimo)
**Step 2:** Sposta l'ultimo elemento in cima
**Step 3:** "Scendi" scambiando con il figlio più piccolo

Ogni discesa è un passo: al massimo $\log n$ passi $\rightarrow$ **O(log n)**
**Esempio: Estrai 2**



# Codice Python: Estrazione (heappop)

La funzione heappop rimuove il minimo (radice) da un min-heap e ripristina le proprietà dell'heap.

```
def heappop(A):
    x = A[0]
    A[0] = A[-1]
    A.pop()
    i = 0
    n = len(A)
    while 2*i + 1 < n:
        # i non è una foglia
        figlio = 2*i + 1  # figlio sinistro
        if figlio + 1 < n and A[figlio + 1] < A[figlio]:
            # c'è anche il figlio destro ed ha valore minore del figlio sinistro
            figlio  += 1  # usa il figlio più piccolo
        if A[i] <= A[figlio]:
            break
        #scambia il valore del padre con quello del figlio più piccolo
        A[i], A[figlio] = A[figlio], A[i]
        # scendi al nodo figlio
        i = figlio
    return x
```

**Esempio:**

`A = [2, 4, 6, 10, 8, 9, 12]` $\xrightarrow{\texttt{heappop(A)}}$ `A = [4, 8, 6, 10, 12, 9]`

Il valore 2 viene rimosso e l'heap resta valido. Complessità: $\mathcal{O}(\log n)$.

# Costruzione di un Min-Heap: Concetto di Heapify

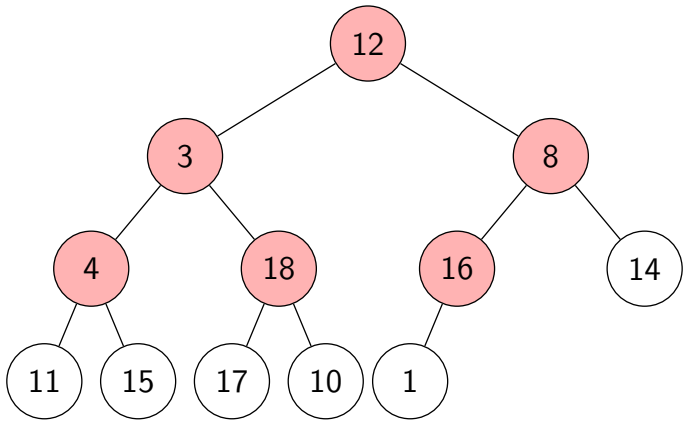
Dato un array arbitrario di $n$ elementi, possiamo trasformarlo in un **min-heap** eseguendo una procedura detta **heapify**.

**Idea:** Si considerano i nodi interni dell'albero per indice decrescente e a ciascuno di questi si applica "la discesa" in modo che il il valore del nodo risulti minore o quale a quello dei suoi figli.

Diamo ora un esempio del funzionamento di *heapify* sul vettore

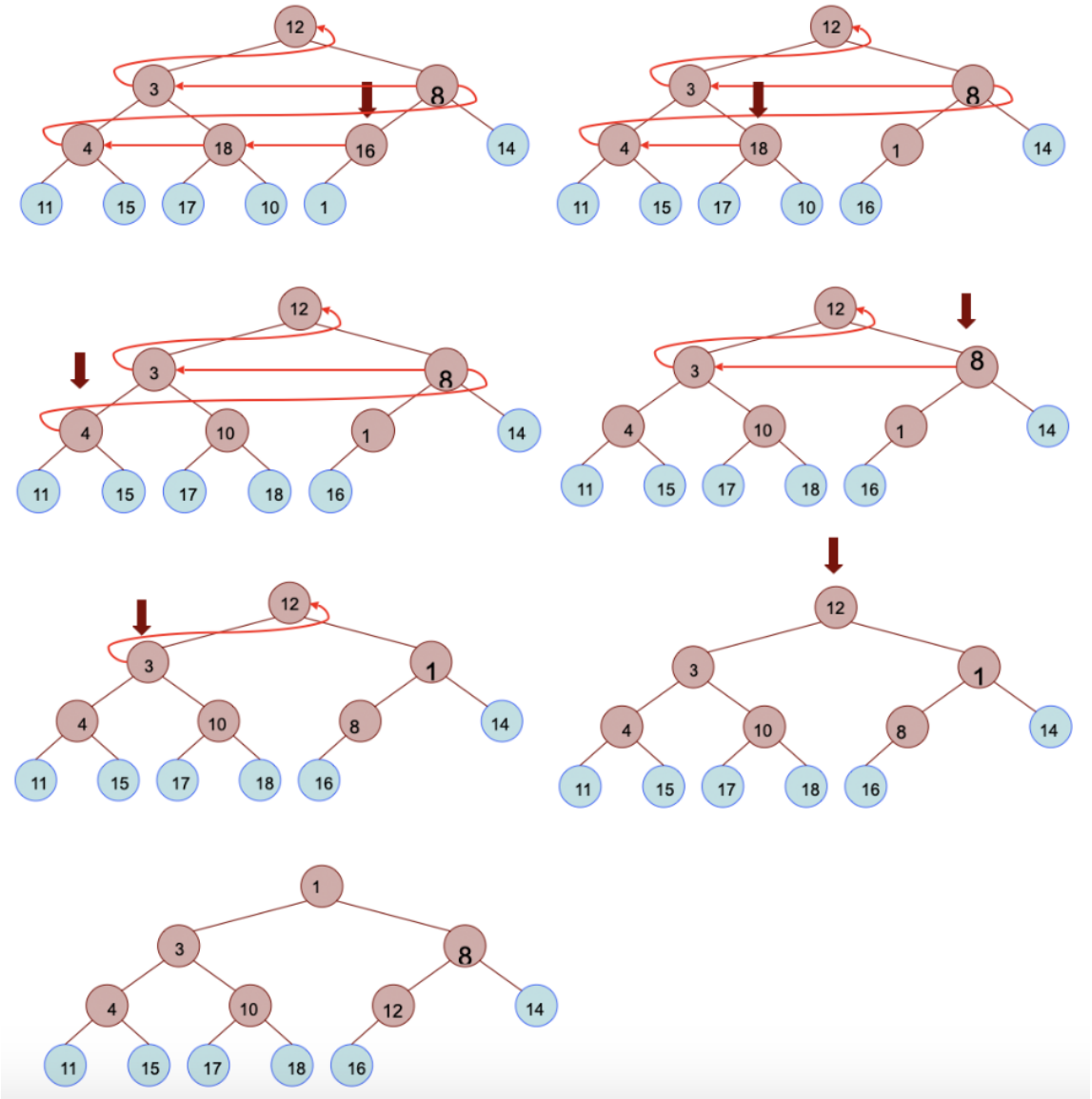
$$A = [12, 3, 8, 4, 18, 16, 14, 11, 15, 17, 10, 1].$$

In quest'esempio con $n = 12$ nodi la funzione *heapify* richiama un aggiustamento su ciascuno dei 6 nodi interni, quelli con indice 5, 4, 3, 2, 1, 0 evidenziati in rosso in figura



# Costruzione di un Min-Heap da un Array (Heapify)

Di seguito l'effetto di ciascuna delle 6 iterazioni sui nodi interni di $A$:



# Implementazione della funzione `heapify`

Dato un array arbitrario di $n$ elementi, possiamo trasformarlo in un **min-heap** eseguendo una procedura detta **heapify**.

**Idea:** Si parte dagli ultimi nodi interni e si applica l'aggiustamento (*heapify down*) a ciascuno di essi.

```
def heapify(A):
    n = len(A)
    for i in range(n // 2 - 1, -1, -1):
        heapify_down(A, i)

def heapify_down(A, i):
    n = len(A)
    while 2*i + 1 < n:
        j = 2*i + 1
        if j + 1 < n and A[j + 1] < A[j]:
            j += 1
        if A[i] <= A[j]:
            break
        A[i], A[j] = A[j], A[i]
        i = j
```

*heapify_down* prende un nodo che potrebbe violare la proprietà di min-heap, e lo fa "scendere" finché è più piccolo dei suoi figli, ristabilendo la proprietà dell'heap.

**Complessità:** $\mathcal{O}(n)$ — anche se ogni *heapify_down* ha costo $\mathcal{O}(\log n)$, i nodi più profondi richiedono meno lavoro. (Lo vedremo nel prossimo frame.)

# Perché `heapify` è $\mathcal{O}(n)$

Sia $A$ un array di $n$ elementi. La procedura `build_heap` chiama la funzione `heapify_down` su tutti i nodi interni.

**Osservazione:** Il costo per ciascun nodo interno dipende dalla sua **altezza** (cioè quanto può "scendere" nel peggiore dei casi).

**Fatti chiave:**

- In un heap binario completo di altezza $h$, ci sono al più $2^{h-i}$ nodi a distanza $i$ dalle foglie.
- Ognuno di questi nodi può costare al più $i$ operazioni

**Costo totale:** $T(n) \leq \sum_{i=0}^{h} i \cdot 2^{h-i} = 2^h \cdot \sum_{i=0}^{h} \frac{i}{2^i} < 2^h \cdot 2 = O(n)$
L'ultima diseguaglianza segue dal fatto che la serie $\sum_{i=0}^{\infty} \frac{i}{2^i}$ converge a 2.

**Conclusione:** `build_heap` è molto più efficiente di eseguire *heappush* $n$ volte (che sarebbe $\mathcal{O}(n \log n)$)!

# Python e Heap: la libreria `heapq`

Il linguaggio **Python** fornisce una libreria standard chiamata `heapq` per lavorare con **min-heap**.

**Le principali funzioni della libreria `heapq`:**

- `heapify(A)`:
  Trasforma una lista arbitraria `A` in un **min-heap** in tempo $\mathcal{O}(n)$.
- `heappop(A)`:
  Estrae il **minimo** dall'heap `A` in tempo $\mathcal{O}(\log n)$.
- `heappush(A, x)`:
  Inserisce l'elemento `x` nell'heap `A` in tempo $\mathcal{O}(\log n)$.

**Nota:** Gli heap in `heapq` sono implementati come `list`, e devono essere trattati come min-heap tramite l'uso delle funzioni appropriate.

# `heapsort`: come funziona?

**Obiettivo:** ordinare una lista di elementi usando un **heap**.

1. **Costruzione dell'heap**:
   - Trasforma l'array iniziale in un `min-heap`.
   - Utilizza la funzione `heapify` in tempo $\mathcal{O}(n)$.
2. **Estrazione degli elementi**:
   - Estrai ripetutamente il minimo (radice dell'heap).
   - Dopo ogni estrazione, ricostruisci l'heap.
   - Gli elementi estratti vengono ordinati progressivamente.
3. **Risultato**:
   - Array ordinato al termine delle estrazioni.
   - Complessità totale: $\mathcal{O}(n \log n)$.

# HeapSort con Min-Heap in Python

Utilizziamo le funzioni della libreria `heapq` per implementare HeapSort in modo semplice ed efficace.

```
import heapq

def heapsort(A):
    heapq.heapify(A)
    B = [ ]
    while A:
        x = heapq.heappop(A)
        B.append(x)
    return B
```

**Nota:** Questo heapsort non è `in-place` perché alloca una lista $B$ di output.

# ESERCIZI

**Suggerimento:** Prima di cercare le soluzioni in rete, prova a ragionarci da solo: gli esercizi che seguono sono pensati per aiutarti a mettere alla prova la tua comprensione, la tua capacità di sviluppare una prova di correttezza e costruire un algoritmo corretto.
Solo dopo averci riflettuto, confronta la tua soluzione con altre possibili versioni.

# Esercizi

1. Dato il vettore
$$A = [60, 30, 50, 20, 10, 30]$$
,
   1. Costruire manualmente l'heap minimo che si ottiene inserendo uno dopo l'altro in un heap vuoto gli elementi di $A$. Mostrare lo stato del vettore dopo ogni inserimento.
   2. Effettuate 4 estrazioni dall'heap ottenuto al punto 1. Mostrare lo stato del vettore dopo ogni estrazione.