Corso di laurea in Informatica
Introduzione agli Algoritmi

Il problema dell'ordinamento: Heap e algoritmo Heapsort

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

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**

Supponiamo di avere una lista A di n elementi su cui nel tempo dovrò effettuare un numero arbitrario di questi due tipi di operazioni:

estrazione del minimo

aggiunta di un elemento

se mantengo la lista cosi com'è senza strutturarla:

crea struttura $O(1)$

estrazione del minimo $\Theta(|A|)$

aggiunta di un elemento $O(1)$

se strutturo la lista come **lista ordinata**

crea struttura $\Theta(n \log n)$

estrazione del minimo $O(1)$ (se ordinamento decrescente)

aggiunta di un elemento $O(|A|)$

se mantengo la lista come un **heap** vedremo che:

crea struttura $\Theta(n)$

estrazione del minimo $O(\log |A|)$

aggiunta di un elemento $O(\log |A|)$

la struttura dati heap in python è implementata nella libreria standard **heapq**.
Questa libreria offre le tre funzioni per gestire liste che fungono da heap:

crea struttura: $heapify(A)$ trasforma una lista A arbitraria di n elementi in un heap minimo. Tempo richiesto $O(n)$.

estrai minimo: $heappop(A)$ rimuove e restituisce l'elemento minimo della lista e ristabilisce la proprietà di heap minimo. Tempo richiesto $O(\log|A|)$.

inserisci elemento: $heappush(A, x)$ inserisce l'elemento x in modo che l'heap mantenga la proprietà di essere heap minimo. Tempo richiesto $O(\log|A|)$.

L'algoritmo dell'heapsort si basa sulla seguente idea:

- organizza gli elementi in A da ordinare come heap minimo,
- estrai gli elementi dell'heap uno alla volta ed accodali ad un nuovo vettore B inizialmente vuoto.
- restituisci il vettore B.

Ad ogni estrazione si estrae da A il valore minimo quindi gli elementi finiranno nel nuovo vettore in ordine crescente.

```
def Heapsort(A):
    from heapq import heapify, heappop
    heapify(A)
    B = [ ]
    while A:
        B.append(heappop(A))
    return B
```

$\Theta(n)$

eseguito n volte

$O(\log n)$

La complessità temporale dell'algoritmo è dunque $\Theta(n \log n)$.

Nella versione proposta qui l'algoritmo non ordina in loco e lo spazio di lavoro è $\Theta(n)$ a causa dell'utilizzo del vettore B ma non è difficile implementare l'algoritmo in modo che lo spazio di lavoro sia $O(1)$.

Nell'implementazione dell'algoritmo di heapsort è bastato utilizzare le sole due funzioni *heapify* e *heappop* nel seguito descriveremo come implementare tutte e tre le funzioni tipiche della gestione di un heap.

Cominciamo con l'***heapify***:

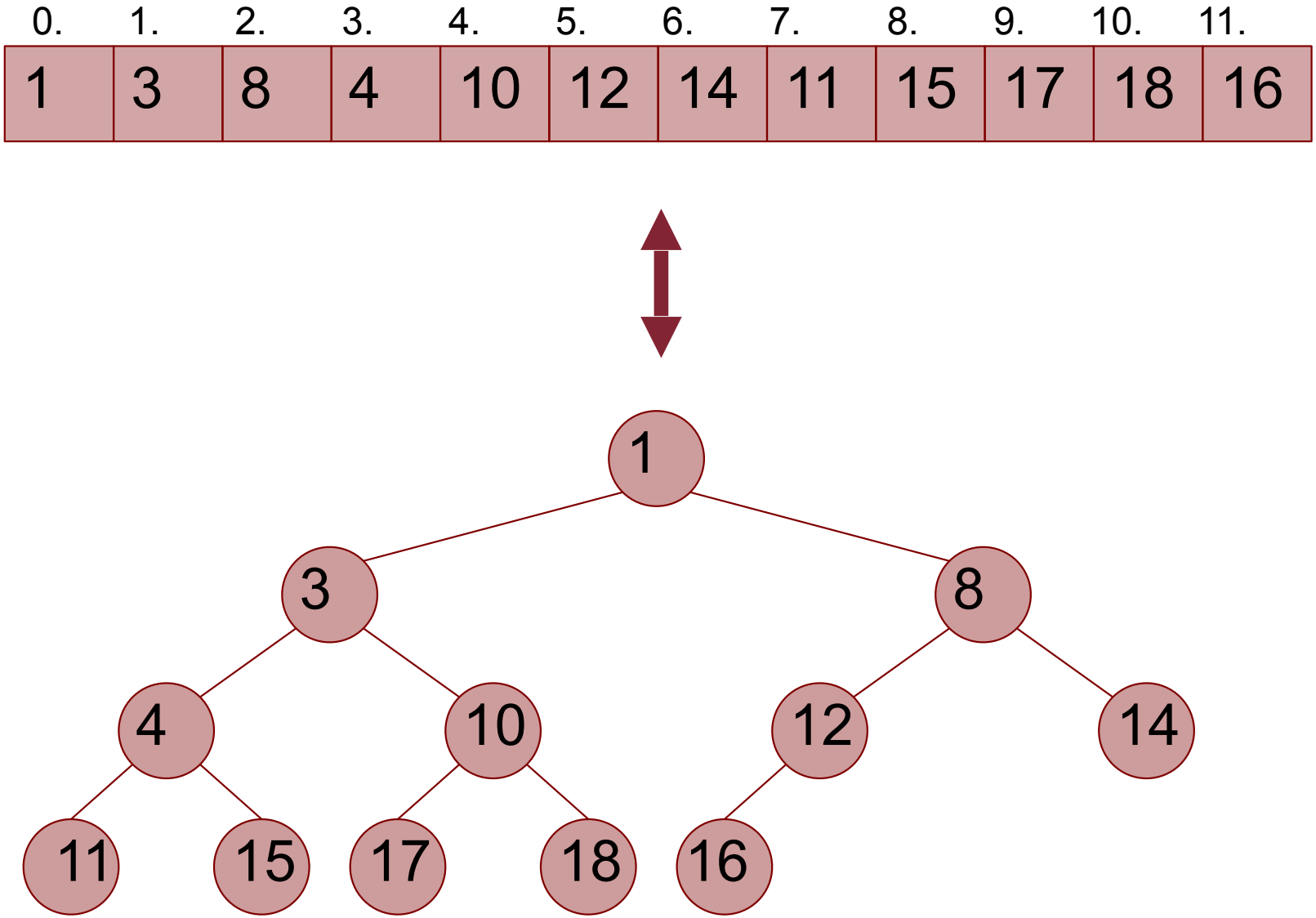
```
>>> inport heapq
>>> A = [12, 3, 8, 4, 18, 16, 14, 11, 15, 17, 10, 1]
>>> heapq.heapify( A )
>>> print(A)
[1,  3,  8,  4, 10, 12, 14, 11, 15, 17, 18, 16]
```

Nell'heap gli elementi di A vengono risistemati in modo da rispettare una forma debole di ordinamento: il cosiddetto **ordinamento verticale**, fare in modo che gli elementi rispettino l'ordinamento standard richiederebbe infatti un tempo di calcolo $\Omega(n \log n)$ che non possiamo permetterci.
L'ordinamento verticale significa che guardando agli elementi di A come appartenenti ad un albero binario completo o quasi completo, i nodi presenti in ogni cammino radice foglia risultano ordinati in modo crescente.
Stando cosi le cose l'elemento minimo si troverà necessariamente al primo posto.

La struttura dati Heap (continua)

- ogni nodo dell'albero binario corrisponde esattamente ad un 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



La struttura dati Heap

Proprietà:

- Poiché lo heap ha tutti i livelli completamente pieni tranne al più l'ultimo, la sua altezza è $\Theta(\log n)$.

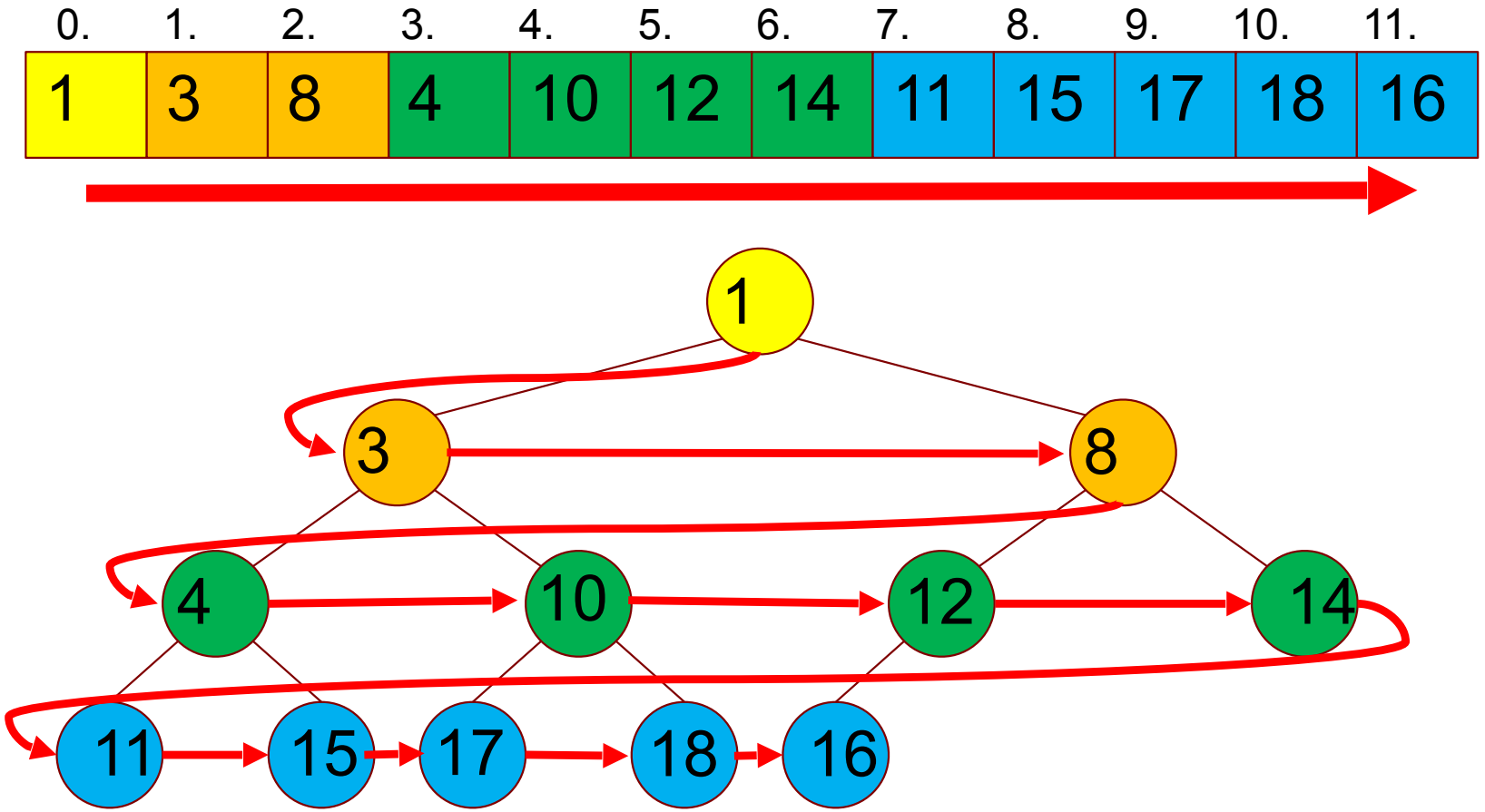
- Con questa implementazione, la proprietà di ordinamento verticale implica che per tutti gli elementi tranne $A[0]$ (poiché esso corrisponde alla radice dell'albero e quindi non ha genitore) vale:

$$A[parent(i)] \leq A[i].$$

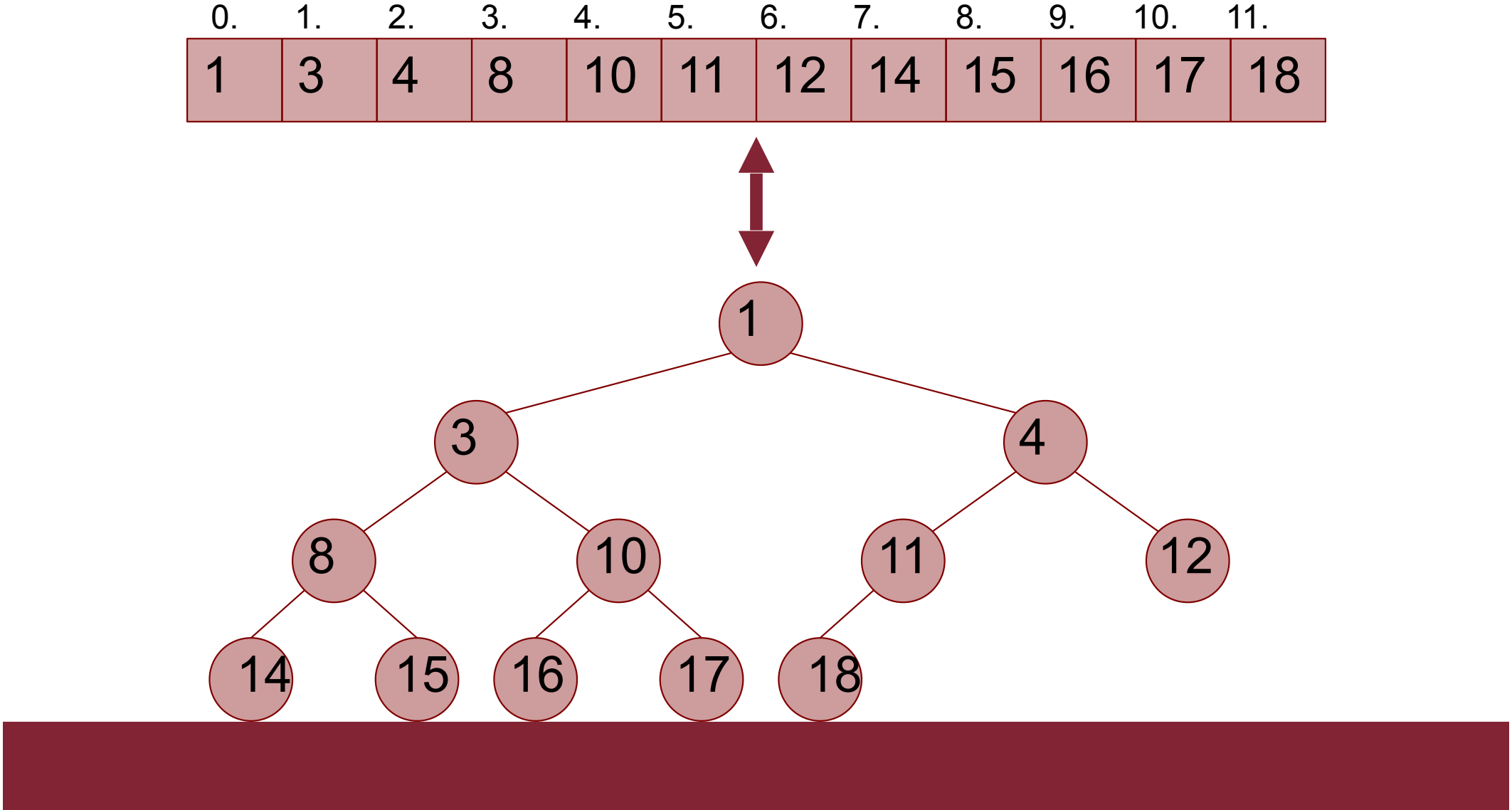
- L'elemento minimo risiede nella radice, quindi può essere trovato in tempo $\Theta(1)$.

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

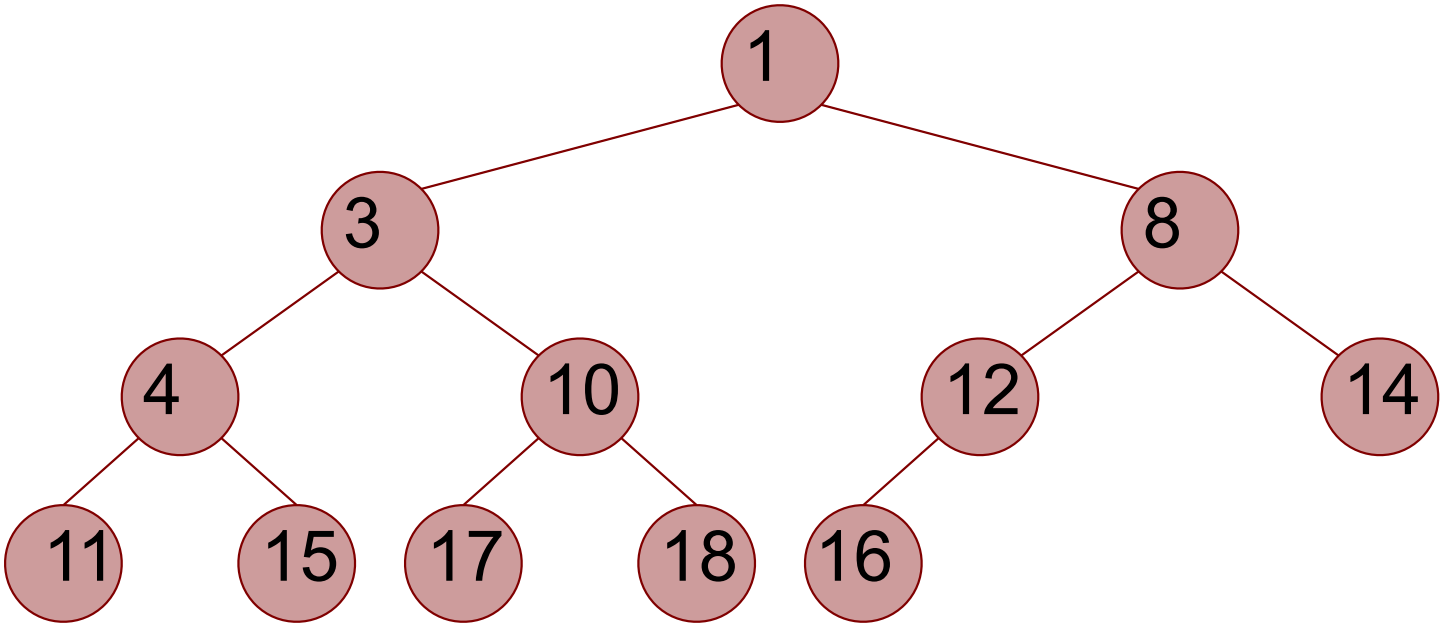


Nota che **un ordinamento è un ordinamento verticale che garantisce anche l'ordinamento orizzontate mentre il viceversa ovviamente non vale**.
Ecco perché l'ordinamento verticale è qualcosa di meno forte del classico ordinamento e vedremo che è possibile calcolarlo in tempo O(n), inferiore quindi a $\Omega(n \log n)$



La struttura dati Heap

Uno ***heap minimo*** è dunque un albero binario *completo o quasi completo*, ossia un albero binario in cui tutti i livelli sono pieni, tranne eventualmente l'ultimo, i cui nodi sono addensati a sinistra…



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

L' ***heap massimo*** *al contrario ha la* proprietà che la chiave di ogni nodo è *maggiore o uguale* alla chiave dei suoi figli.

Funzione Heapify

- L'algoritmo di heapify si avvale di una funzione ausiliaria heapify1, necessaria per il suo corretto funzionamento

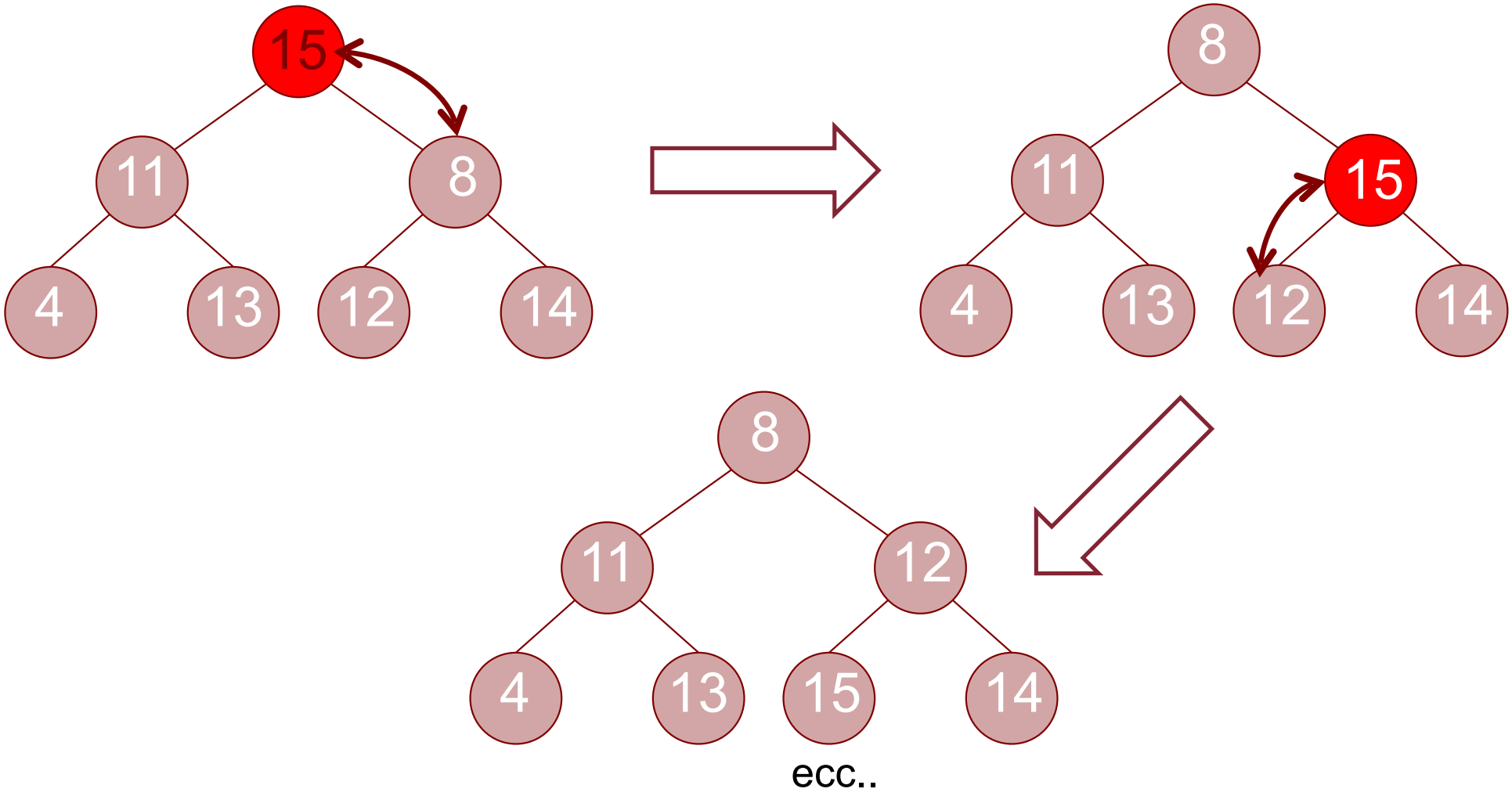
•La funzione Heapify1 ha lo scopo di mantenere la proprietà di heap, sotto l'ipotesi che nell'albero su cui viene fatta lavorare sia garantita la proprietà di heap per entrambi i sottoalberi (sinistro e destro) della radice. Di conseguenza l'unico nodo che può violare la proprietà di heap è la radice 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 minore 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

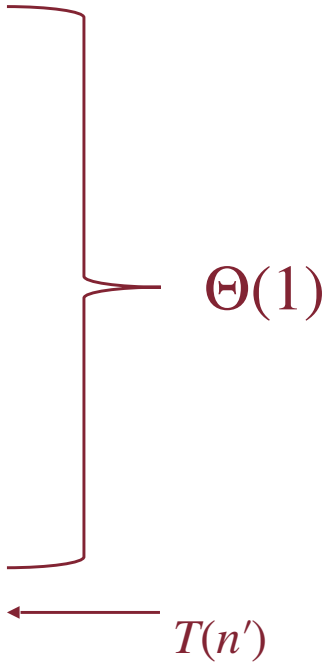
Esempio:



ecc..

Funzione Heapify1

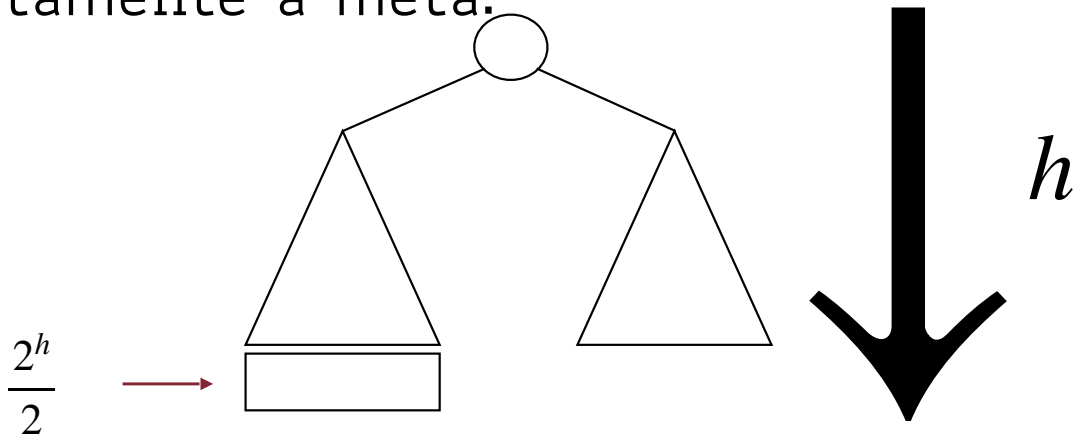
```
def Heapify1(A, i):
    n = len(A)
    L = 2*i + 1; R = 2*i + 2
    indice_min = i
    if L < n and A[L] < A[indice_min]:
        indice_min = L
    if R < n and A[R] < A[indice_min]:
        indice_min = R
    if indice_min != i:
        A[i], A[indice_min] = A[indice_min], A[i]
        Heapify1(A, indice_min)
```



$T(n) = T(n') + \Theta(1)$ dove n è il numero di nodi del sottoalbero radicato in i e dove n' è il numero di nodi del sottoalbero di i 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:



Sia $n' = 2^h - 1$ il numero di nodi nel sottoalbero di sinistra.

$$n = 2^{h+1} - 1 - \frac{2^h}{2} = \frac{3 \cdot 2^h - 2}{2} = \frac{3\left(2^h - 1\right) + 1}{2} = \frac{3n' + 1}{2}$$

da cui ricaviamo: $n' = \frac{2n - 1}{3} < \frac{2n}{3}$

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

Funzione Heapyfy

serve per trasformare qualunque vettore A contenente n elementi in uno heap, chiamando ripetutamente Heapify1 sugli opportuni nodi dello heap.

Osservazioni:

- poiché Heapify1 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 (il padre del nodo in posizione $n-1$), che ha indice:

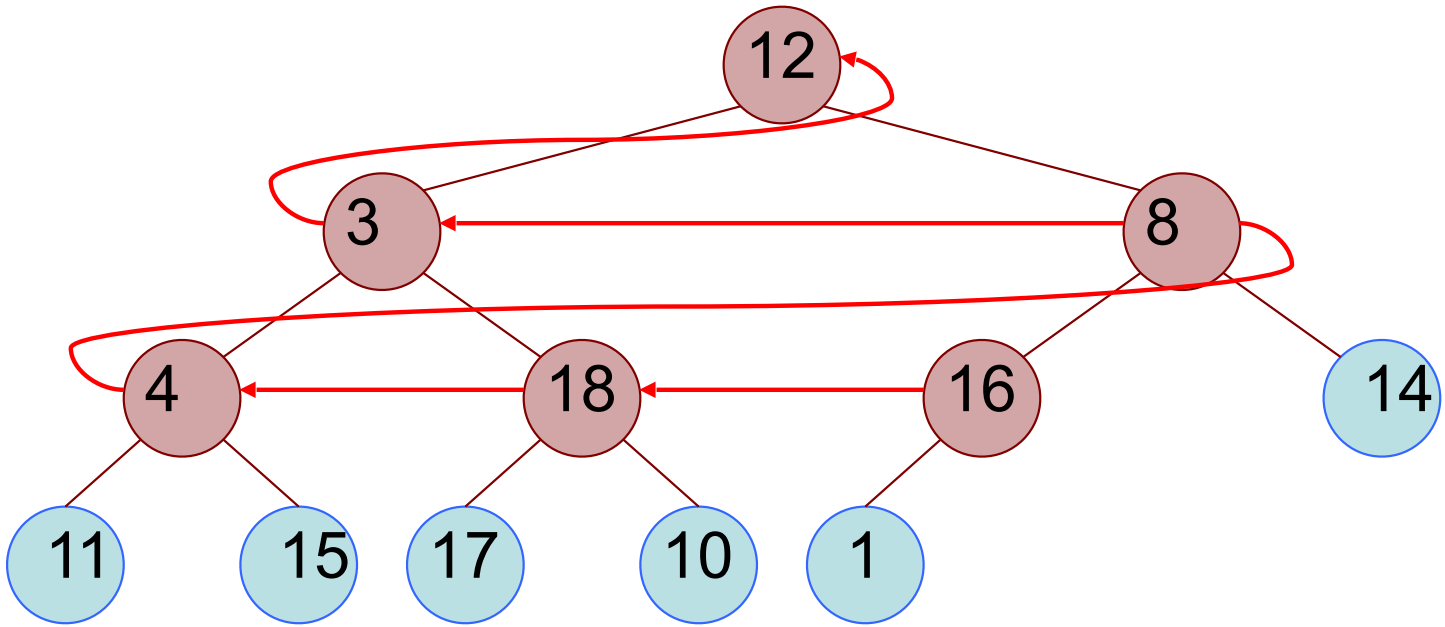
$$\left\lfloor \frac{n-2}{2} \right\rfloor$$

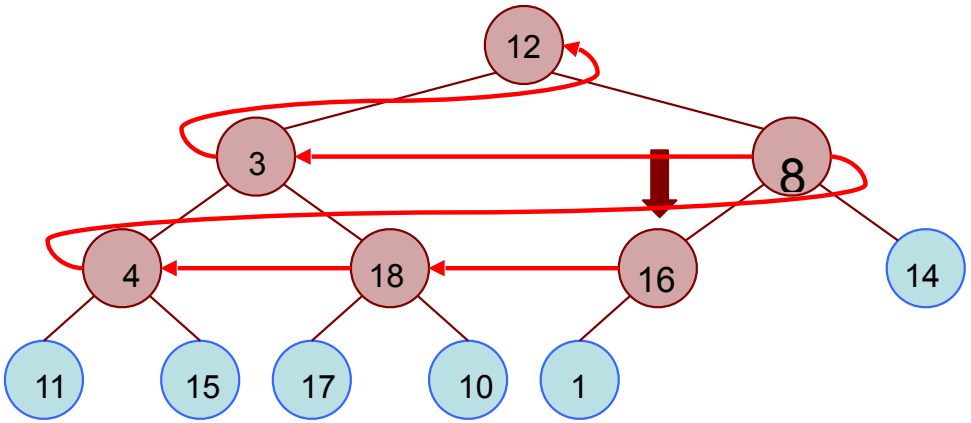
Funzione Heapify

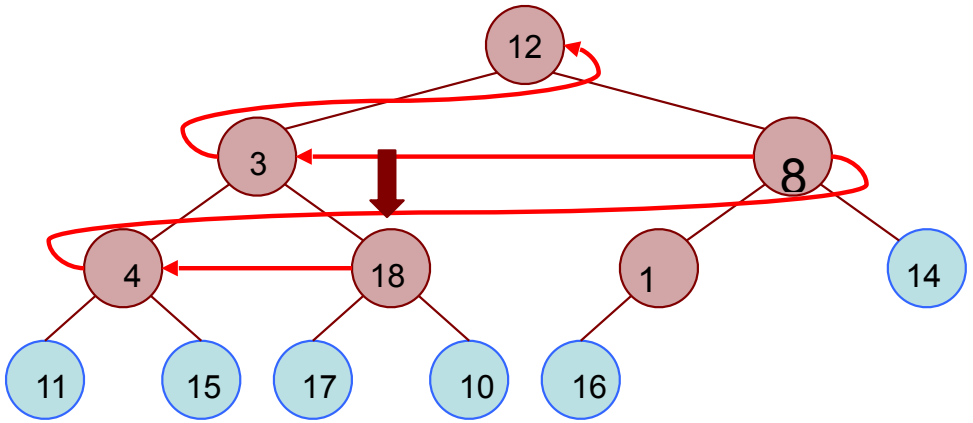
```
def Heapify(A):
    for i in range((len(A)-2)//2,-1,-1):
        Heapify1(A, i)
```

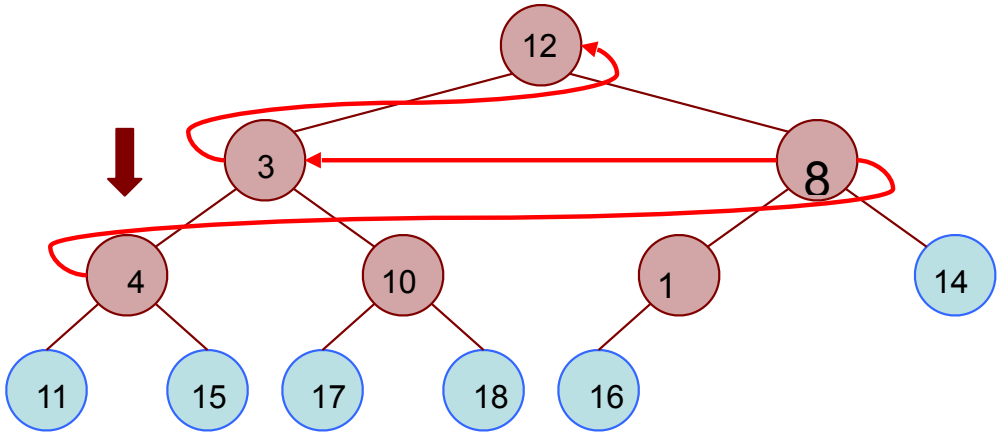
```
>>> A
[12, 3, 8, 4, 18, 16, 14, 11, 15, 17, 10, 1]
>>> Heapify(A)
>>> A
[1, 3, 8, 4, 10, 12, 14, 11, 15, 17, 18, 16]
```

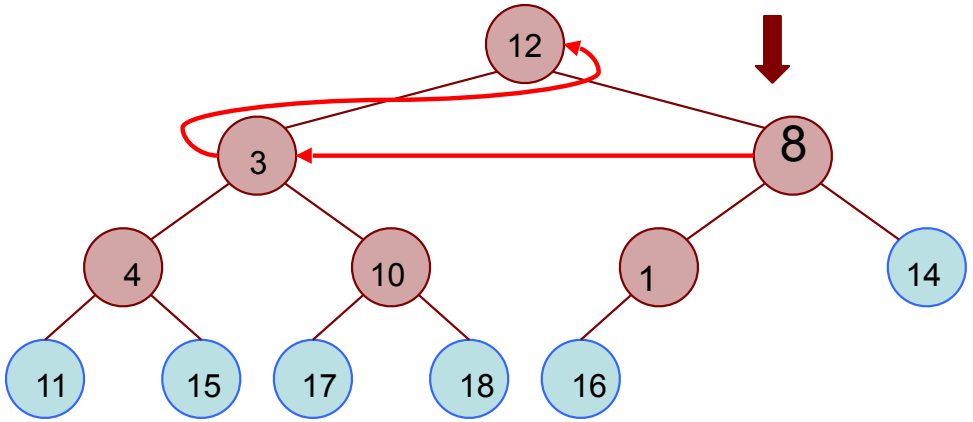
In questo esempio di $n = 12$, Heapify chiama Heapify1 sui nodi con indice 5, 4, 3, 2, 1, 0:

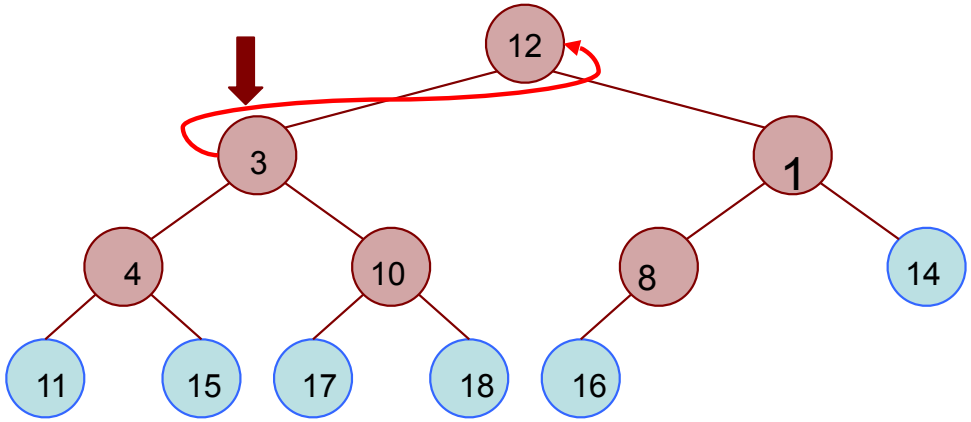


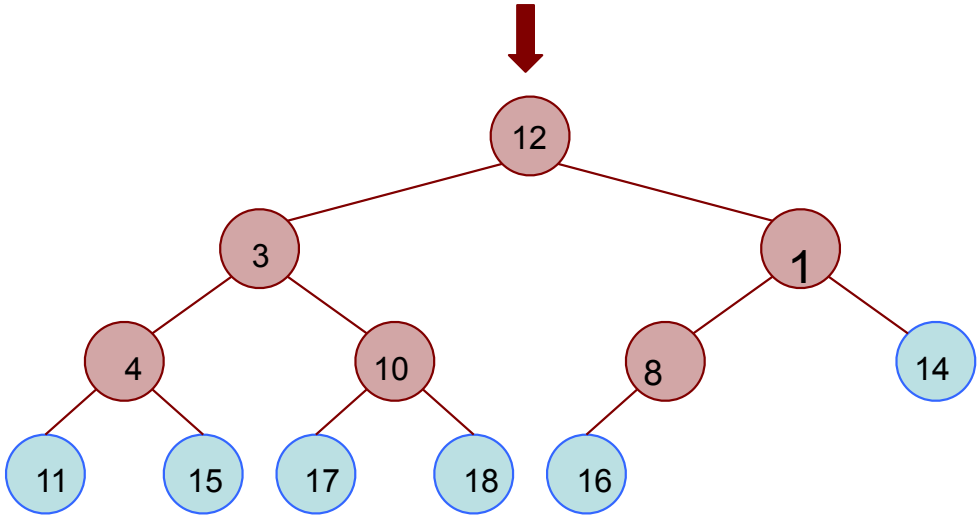

12
3
8
4
18
16
14
11
15
17
10
1

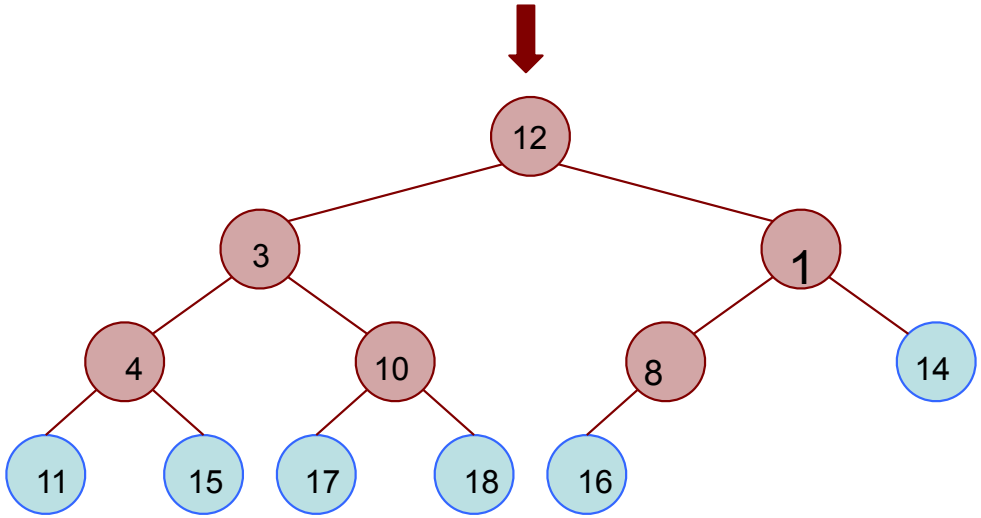

12
3
8
4
18
1
14
11
15
17
10
16

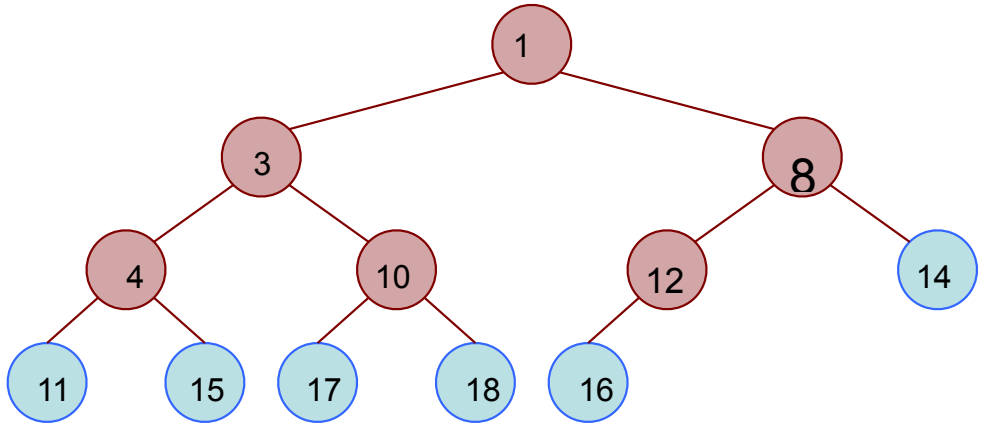

12
3
8
4
10
1
14
11
15
17
18
16


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


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


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






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

Heapify(A)
A = [1, 3, 8, 4, 10, 12, 14, 11, 15, 17, 18, 16]
```

Funzione Buildheap

```
def Heapify(A):
    for i in range((len(A)-2)//2,-1,-1):
        Heapify1(A, i)
```

La funzione *Heapify* effettua non più di n chiamate di *Heapify*1, che sappiamo avere ciascuna costo $O(\log n)$, quindi:

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

Con un calcolo più accurato mostreremo ora che $T(n) = \Theta(n)$

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 $2^{h-i} = \frac{2^h}{2^i}$
- 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:

$$T(n) \le \sum_{i=1}^{h} 2^{h-i} O(i) = O\left(2^h \sum_{i=1}^{h} \frac{i}{2^i}\right) = O\left(2^h \sum_{i=1}^{h} \left(\frac{3}{4}\right)^i\right)$$

dove l'ultima diseguaglianza segue perche' $\frac{i}{2^i} < \left(\frac{3}{4}\right)^i$

ricorda ora che per una qualunque costante $c < 1$ vale

$\sum_{i=1}^{h} c^i = \frac{c^{h+1} - 1}{c - 1} = \frac{1 - c^{h+1}}{1 - c} = \Theta(1)$ e quindi otteniamo $T(n) = O\left(2^h\right) = O(n)$

Implementazione della funzione $heappop(A)$ in $O(\log n)$:

IDEA:

- salviamo in una variabile x il minimo presente in $A[0]$,
- ricopiamo in A[0] l'ultimo elemento A[n-1] dell'heap e cancelliamo da A l'ultimo elemento
- ci ritroviamo con un heap dove l'unico elemento fuori posto è quello alla radice e possiamo riaggiustare l'heap con una sola chiamata di $Heapify1$ sull'elemento in posizione 0.
- restituiamo l'elemento x

```
def Heappop(A):
    x=A[0]
    A[0]=A[len(A)-1]
    A.pop()
    Heapify1(A,0)
    return x
```

$\left.\begin{array}{l} \\ \\ \\ \end{array}\right\} O(1)$

$O(\log n)$

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

>>> Heapify(A)
>>> print(A)
[1, 3, 8, 4, 10, 12, 14, 11, 15, 17, 18, 16]
>>> Heappop(A)
1
>>> print(A)
[3, 4, 8, 11, 10, 12, 14, 16, 15, 17, 18]
>>> Heappop(A)
3
>>> print(A)
[4, 10, 8, 11, 17, 12, 14, 16, 15, 18]
>>> Heappop(A)
4
>>> print(A)
[8, 10, 12, 11, 17, 18, 14, 16, 15]
```

Implementazione della funzione $heappush(A, x)$ in $O(\log n)$:

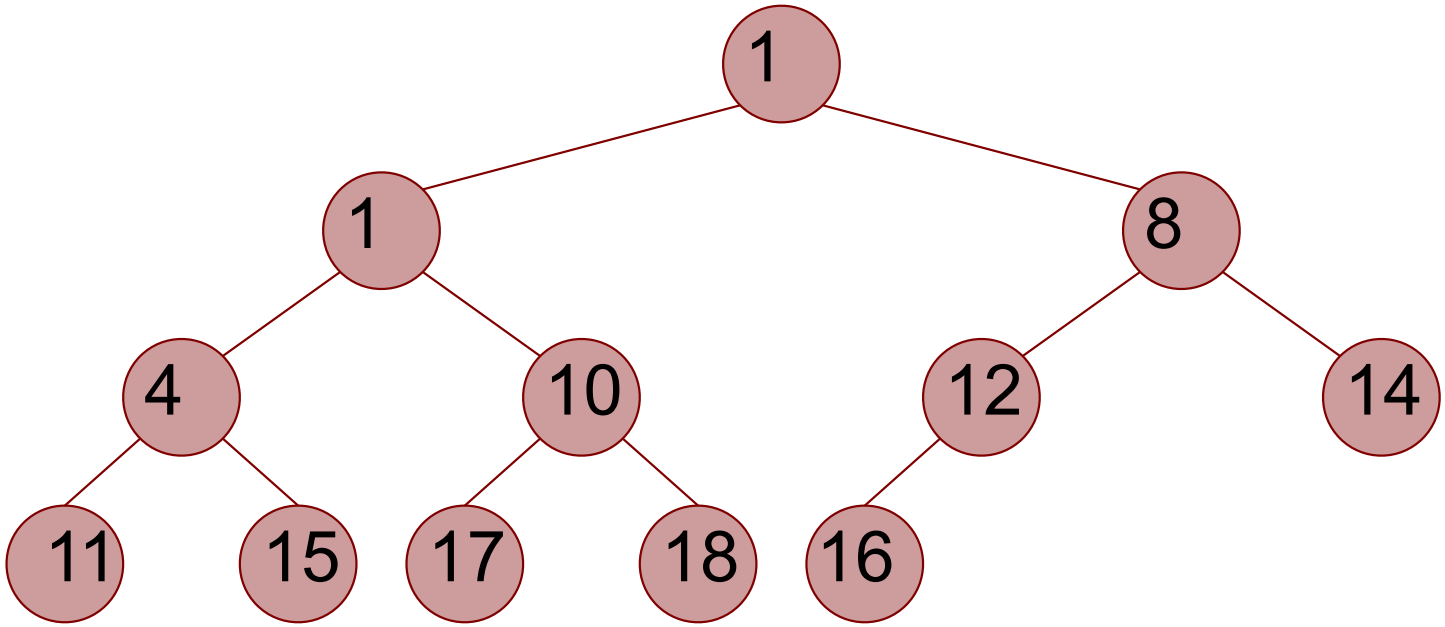
IDEA:

- aggiungiamo l'elemento x all'ultimo posto dell'heap (vale a dire $A.append(x)$),
- Facciamo risalire poi x nell'heap fino a che non risulta maggiore del padre o raggiunge la radice

```
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
```

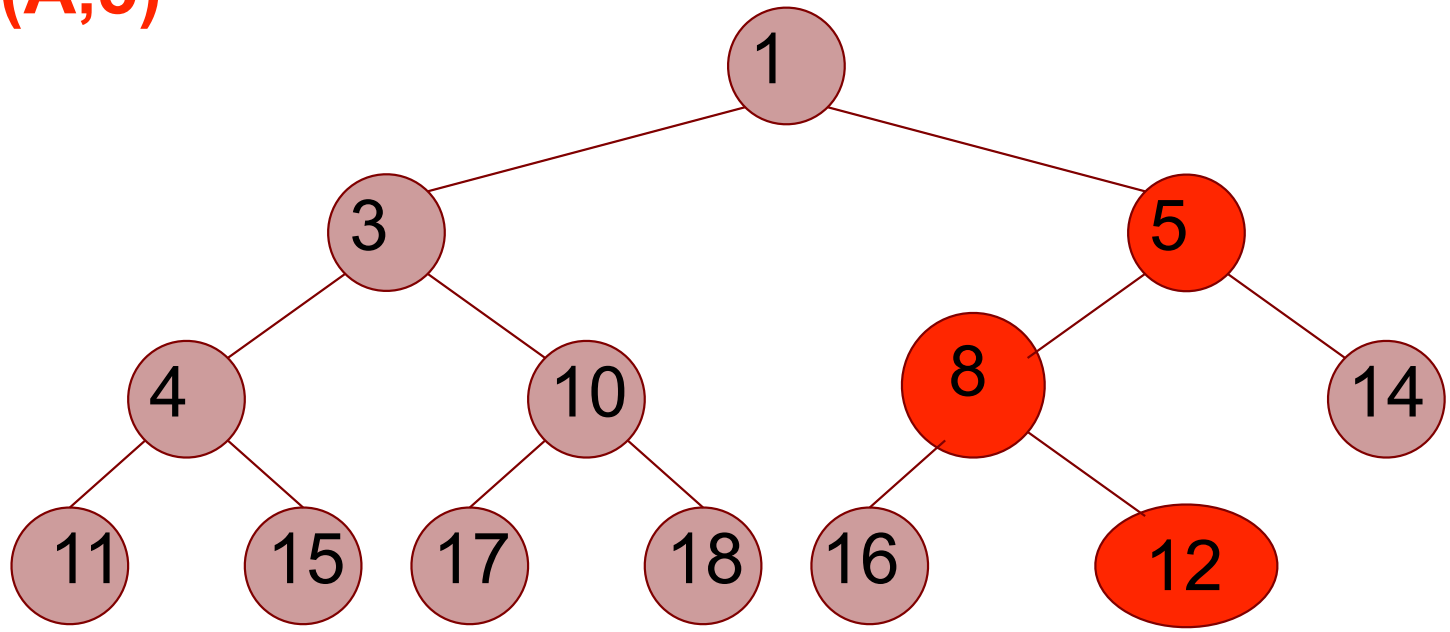
La complessità della procedura dipende dal numero di iterazioni del *while* e questo numero è $O(\log n)$. Infatti il caso pessimo si ha quando l'elemento da inserire è il minimo dell'heap e deve quindi finire alla radice dell'heap ed in questo caso si ha $\Theta(h)$ dove h è l'altezza dell'heap. Poiché l'heap contiene n elementi l'altezza dell'albero binario completo o quasi completo è $\Theta(\log n)$.

$A =$



Heappush(A,5)

$A =$



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

>>> Heapify(A)
>>> print(A)

>>> A=[1, 3, 8, 4, 10, 12, 14, 11, 15, 17, 18, 16]
>>> Heappush(A,0)
>>> print(A)
[0, 3, 1, 4, 10, 8, 14, 11, 15, 17, 18, 16, 12]
>>> Heappush(A,2)
>>> print(A)
[0, 3, 1, 4, 10, 8, 2, 11, 15, 17, 18, 16, 12, 14]
>>> Heappush(A,30)
>>> print(A)
[0, 3, 1, 4, 10, 8, 2, 11, 15, 17, 18, 16, 12, 14, 30]
```

In alcune situazioni può essere necessario estrarre il massimo anziché il minimo dall'heap.

Implementazione di heap massimo

IDEA: Basta inserire nell'heap per ogni elemento x l'elemento $-x$ e ad ogni estrazione cambiare di segno a quanto estratto.
Ecco di seguito possibili implementazioni delle tre funzioni per l'heap massimo:

```
def HeapifyMax(A):
    for i in range(len(A)):
        A[i]=-A[i]
    Heapify(A)

def HeappopMax(A):
    return - Heappop(A)

def HeappushMax(A,x):
    Heappush(A,-x)
```

Ordinamenti stabili

- Un algoritmo di ordinamento si dice stabile se mantiene l'ordine relativo per gli elementi che hanno chiavi uguali. In altre parole, se due elementi hanno la stessa chiave, un algoritmo di ordinamento stabile garantirà che l'elemento che compare prima nella lista originale venga posizionato prima nella lista ordinata. Al contrario, un algoritmo di ordinamento non stabile può riordinare gli elementi con chiavi uguali in modo imprevedibile.
- La stabilità è una proprietà utile in diversi contesti. Ad esempio se si deve ordinare una lista rispetto a più chiavi, è possibile utilizzare un approccio noto come ordinamento stabile iterato. L'idea è di effettuare l'ordinamento in modo iterativo, partendo dalla chiave meno significativa e procedendo gradualmente alla chiave più significativa. In questo modo, l'ordinamento finale rispetterà l'ordine relativo degli elementi anche rispetto alle chiavi meno significative.

Ad esempio l'algoritmo di *SelectionSort* non è stabile. Per convincersene basta considerare cosa accade quando si ordina con questo metodo la lista di tre elementi $[2, 2, 1]$, si ottiene la lista ordinata $[1, 2, 2]$ ma l'ordinamento originario dei due elementi uguali risulta invertito.

Degli algoritmi di ordinamento passati in rassegna non sono stabili gli algoritmi di *SelectionSort*, *QuickSort* ed *HeapSort*.
Sono stabili gli algoritmi di *BubbleSort*, *InsertionSort*, *MergeSort* e *TimSort*.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizi

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

- Progettare un algoritmo di heap-sort che non utilizzi vettori di appoggio e quindi abbia spazio di lavoro $O(1)$. Suggerimento: utilizzare un heap massimo e inserire nello stesso vettore A i vari elementi che via via vengono estratti dall'heap.