Corso di laurea in Informatica
Algoritmi 1
A.A. 2025/26

La struttura dati Heap

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Sommario

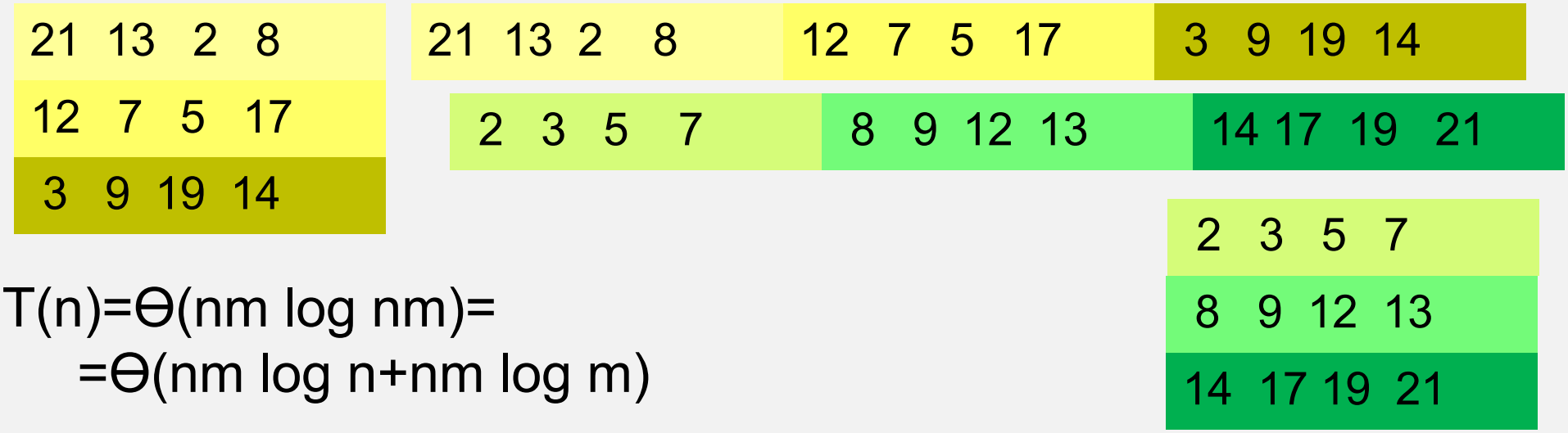
Struttura dati Heap (massimo):

- definizione
- funzione `Heapify` (per ripristinare le proprietà di un heap)
- funzione `Buildheap` (per costruire un heap)
- inserimento di una chiave in un heap
- ricerca del minimo in un heap

Esercizio svolto (1)

Data una matrice *m x n,* si vogliono rimescolare i suoi elementi in modo che tutte le righe e tutte le colonne siano ordinate in modo non decrescente.

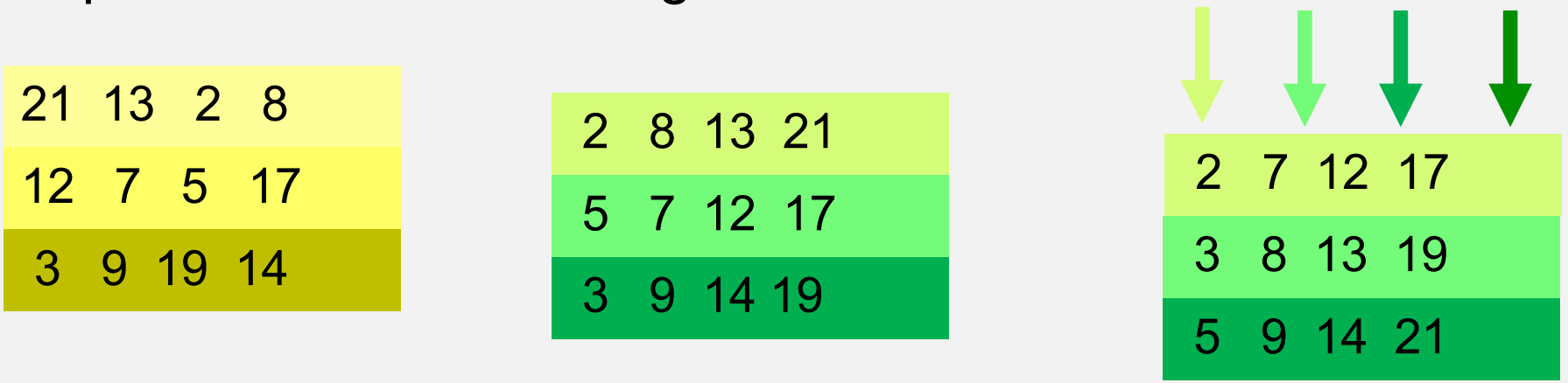
Soluzione 1. Consideriamo la matrice come un unico lungo array e ordiniamolo con un algoritmo efficiente:



$T(n)=\Theta(nm \log nm)=$
$=\Theta(nm \log n+nm \log m)$

Esercizio svolto (2)

Soluzione 2. Ordino ogni riga ed ogni colonna separatamente con un algoritmo efficiente:



$T(n)=\Theta(m\ n \log n + n\ m \log m)=\Theta(nm \log n + nm \log m)$

La struttura dati Heap (1)

La struttura dati ***heap*** è stata introdotta per eseguire in modo efficiente una sequenza di operazioni di:

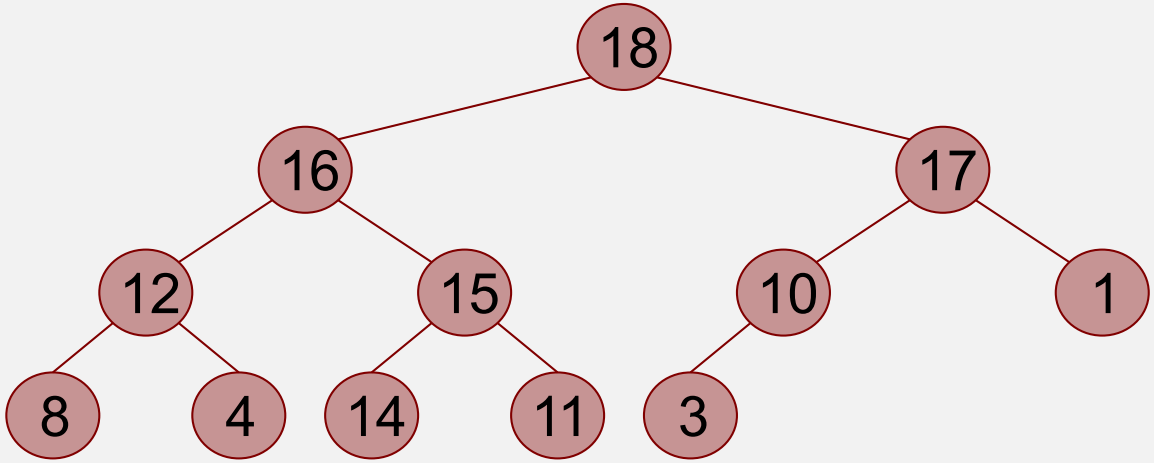
- estrazione (=ricerca+rimozione) del massimo
- inserimento e cancellazione.

Infatti, come vedremo, per compiere queste operazioni ripristinando la struttura perché sia pronta ad un'altra operazione, il tempo richiesto è:

	Estraz. max	Inserim. di una chiave	Cancellaz. di una chiave
Array qualunque	$\Theta(n)$	O(1)	O(1)
Array ordinato	O(1)	O(n)	O(n)
heap	O(log n)	O(log n)	O(log n)

La struttura dati Heap (2)

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



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

La struttura dati Heap (3)

Il modo più naturale per memorizzare uno heap è utilizzare un array *A*, con indici che vanno da *0* fino al numero di nodi dell'heap, ***heap_size***, diminuito di 1; gli elementi possono infatti essere messi in corrispondenza con i nodi dell'heap:

…

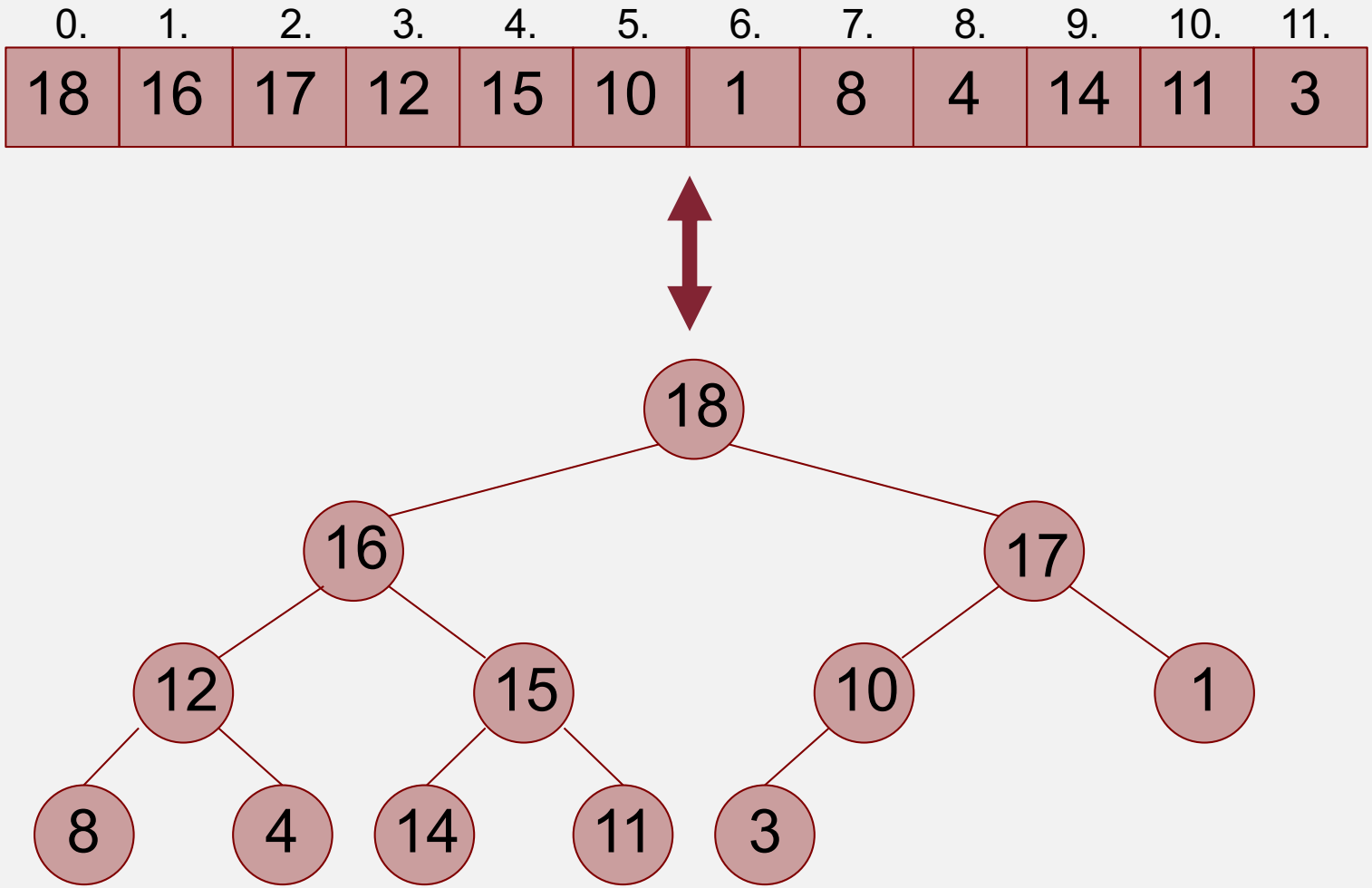
La struttura dati Heap (4)

- l'array è riempito a partire da sinistra; se contiene più elementi del numero *heap_size* di nodi dell'albero, allora i suoi elementi di indice > *heap_size* non fanno parte dell'heap;
- ogni nodo dell'albero binario corrisponde a uno e un solo elemento dell'array A;
- la radice dell'albero corrisponde ad A[0];
- …

La struttura dati Heap (5)

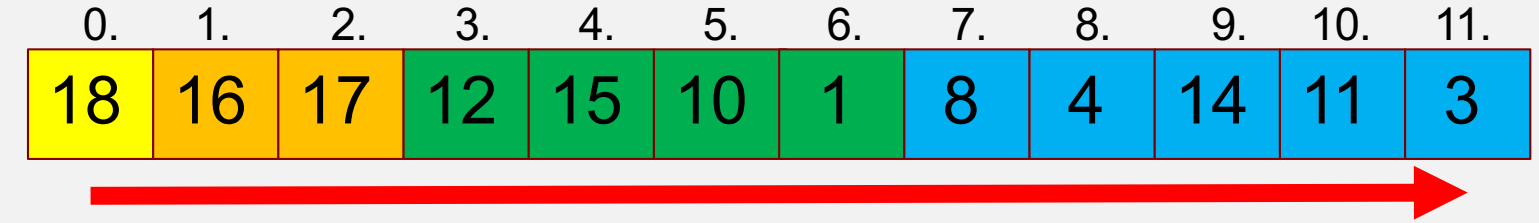
- …
- il figlio sinistro del nodo che corrisponde ad A[i], se esiste, corrisponde all'elemento A[2i+1]: **left(i) = 2i+1**;
- il figlio destro del nodo che corrisponde ad A[i], se esiste, corrisponde all'elemento A[2i + 2]: **right(i) = 2i+2**;
- il padre del nodo che corrisponde ad A[i] corrisponde all'elemento A[⌊(i-1)/2⌋]:
  parent(i) **= ⌊(i-1)/2⌋**.

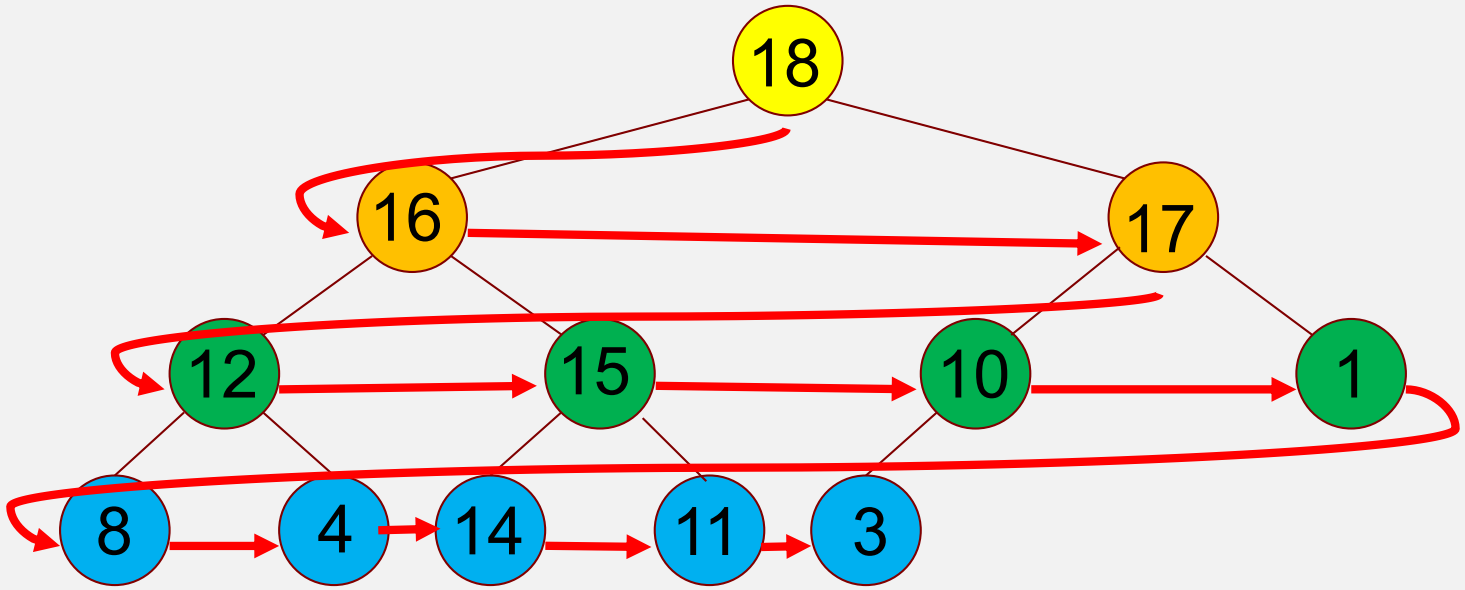
La struttura dati Heap (6)



La struttura dati Heap (7)

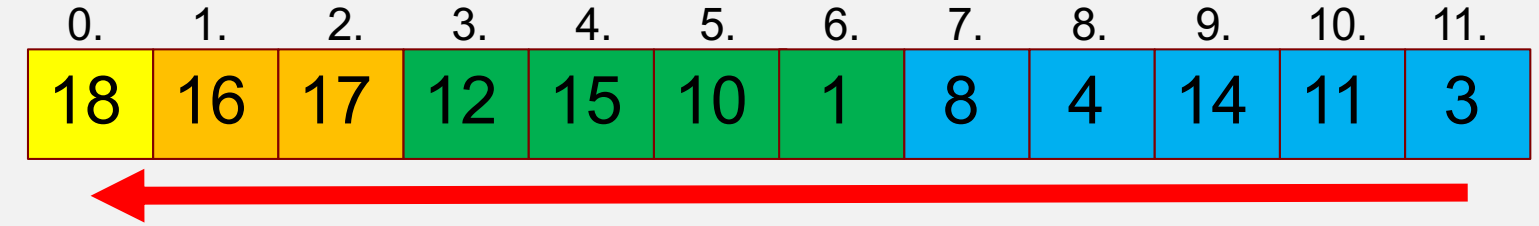
Scorrere l'array da sinistra a destra corrisponde a muoversi sull'albero per livelli, dall'alto verso il basso e da sinistra a destra

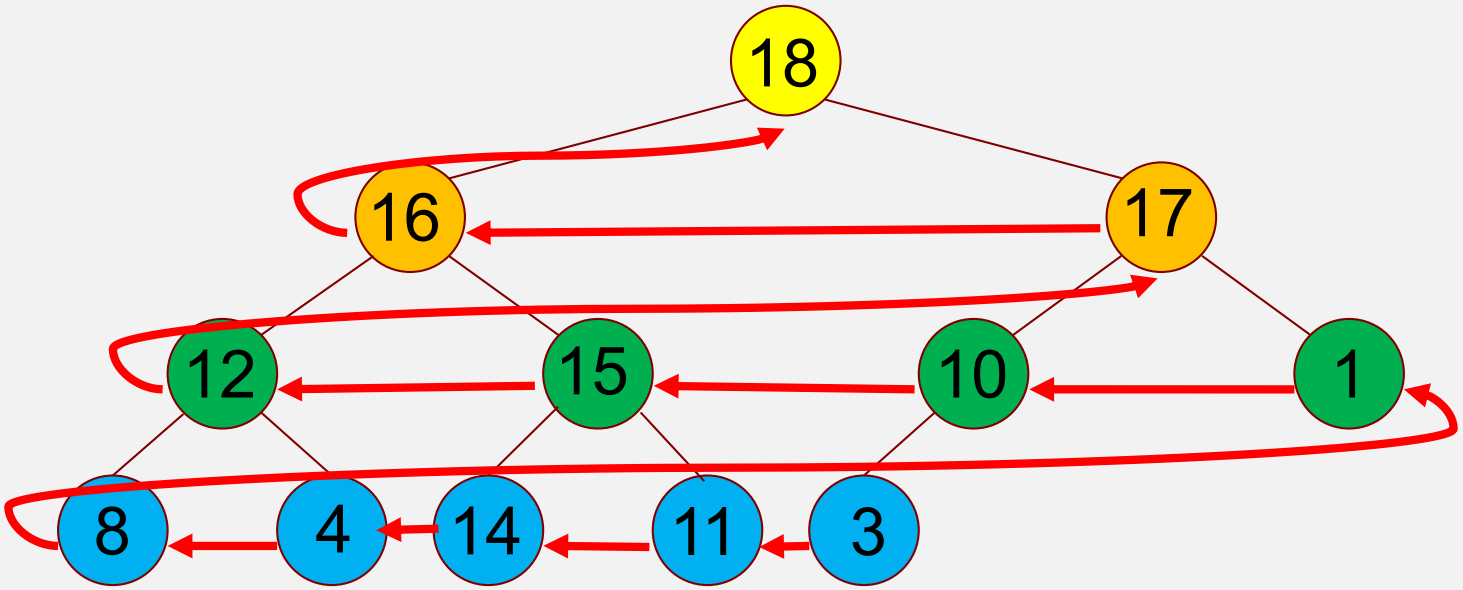




La struttura dati Heap (8)

Simmetricamente, scorrere l'array da destra a sinistra corrisponde a muoversi sull'albero per livelli, dal basso verso l'alto e da destra a sinistra in ciascun livello





La struttura dati Heap (9)

Proprietà:

- Poiché lo heap ha tutti i livelli completamente pieni tranne al più l'ultimo, la sua altezza è $\Theta(\log n)$.
- Con l'implementazione tramite array, la proprietà di ordinamento verticale implica che per tutti gli elementi tranne *A[0]* (che non ha genitore) vale:

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

- L'elemento massimo risiede nella radice, quindi può essere trovato in tempo $O(1)$.

Funzione ausiliarie

L'utilizzo della struttura dati Heap è basato su alcune funzioni ausiliarie, necessarie per il suo corretto funzionamento:

- Funzione Heapify
- Funzione Buildheap
- Funzione Heapify_bottom_up

Illustreremo tali funzioni di seguito.

Funzione Heapify (1)

La funzione Heapify mantiene la proprietà di ordinamento verticale, 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.

…

Funzione Heapify (2)

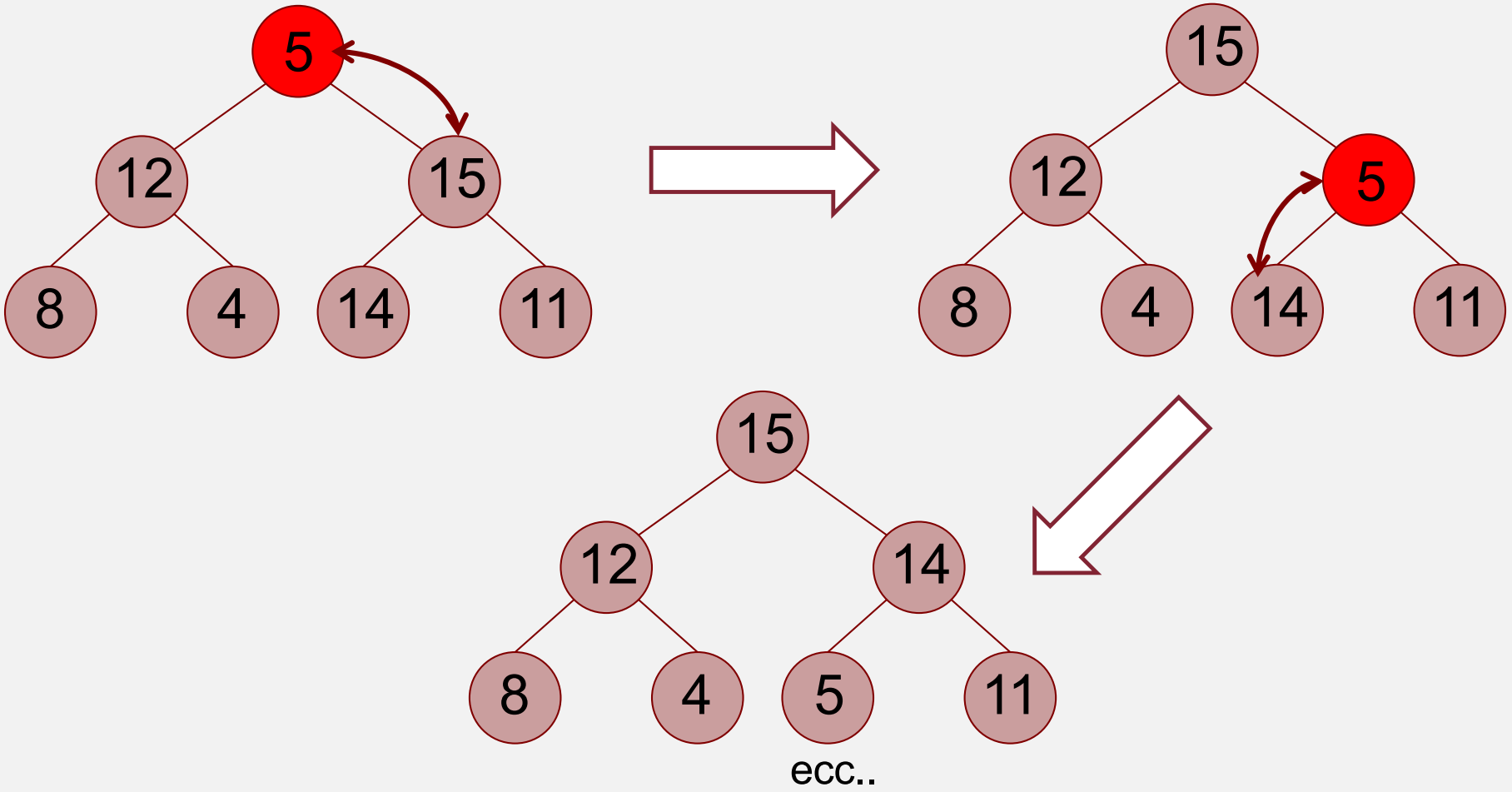
…

La funzione opera sulla radice confrontandola coi suoi figli e, se necessario, la scambia col maggiore di suoi figli.

Dopo lo scambio si verifica se la violazione si sia trasferita sul figlio scambiato e, se necessario, si ripete ricorsivamente l'operazione su tale nodo.

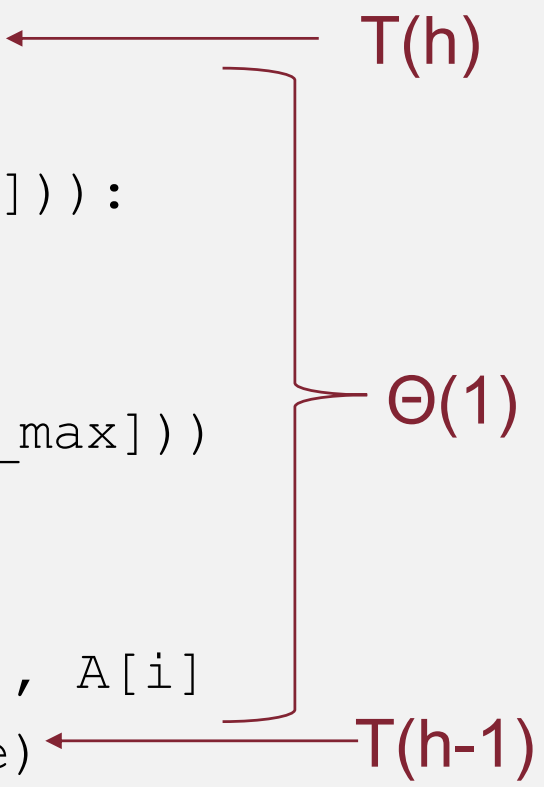
Funzione Heapify (3)

Esempio:



Funzione Heapify (4)

```
def Heapify (A, i, heap_size)
   L=left(i); R=right(i); indice_max=i
   if ((L < heap_size) and (A[L] > A[i])):
      indice_max=L
   if ((R ≤ heap_size) and (A[R] >
                              A[indice_max]))
      indice_max=R
   if (indice_max ≠ i)
      A[i], A[indice_max]=A[indice_max], A[i]
      Heapify (A, indice_max, heap_size)
```



$T(h) = \Theta(1) + T(h-1)$ e $T(1) = \Theta(1)$.
Sappiamo che $h = \Theta(\log n)$ per cui la soluzione nel caso peggiore è: $T(n) = \Theta(\log n)$. In generale $T(n) = O(\log n)$.

Funzione Buildheap (1)

La funzione Buildheap trasforma un array qualunque di n elementi in uno heap, chiamando ripetutamente Heapify.

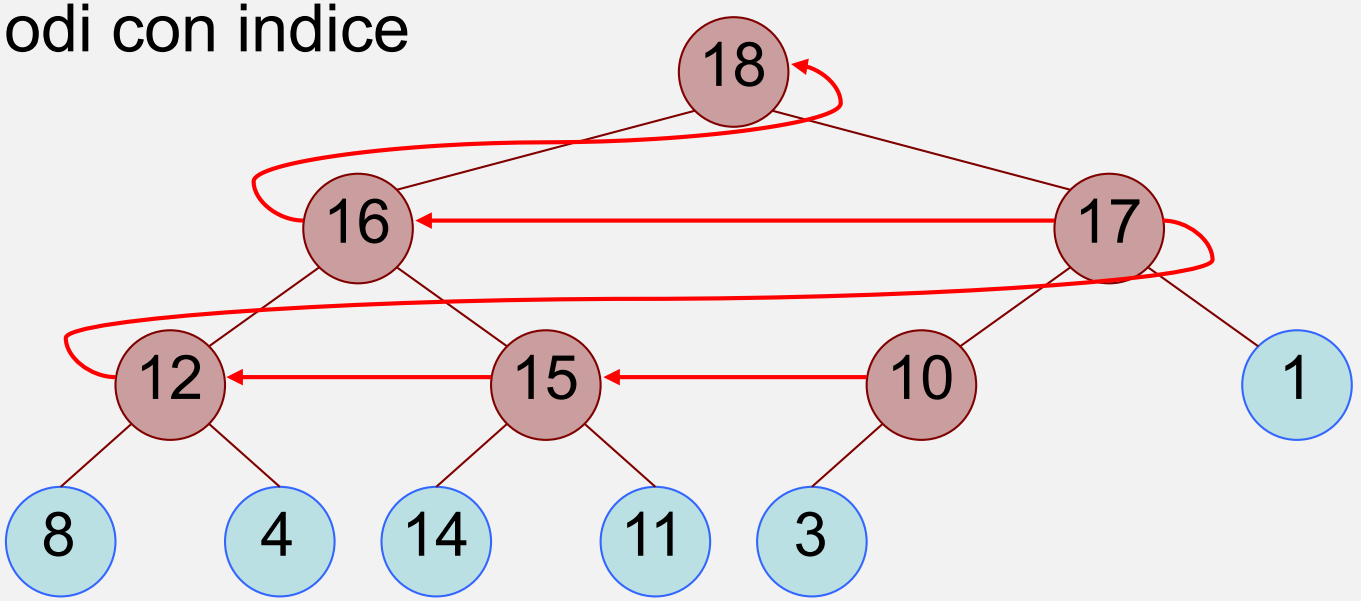
Osservazioni:

- poiché `Heapify` assume che entrambi i sottoalberi della radice siano heap, va chiamata scorrendo l'albero per livelli dal basso verso l'alto;
- ogni foglia è già uno heap, quindi basta chiamare Heapify a partire dal nodo interno più a destra, che ha indice $\lfloor n/2 \rfloor$

Funzione Buildheap (2)

```
def Build_heap(A):
  for i in reversed(range(len(A)//2)):
    Heapify (A, i,heap_size)
```

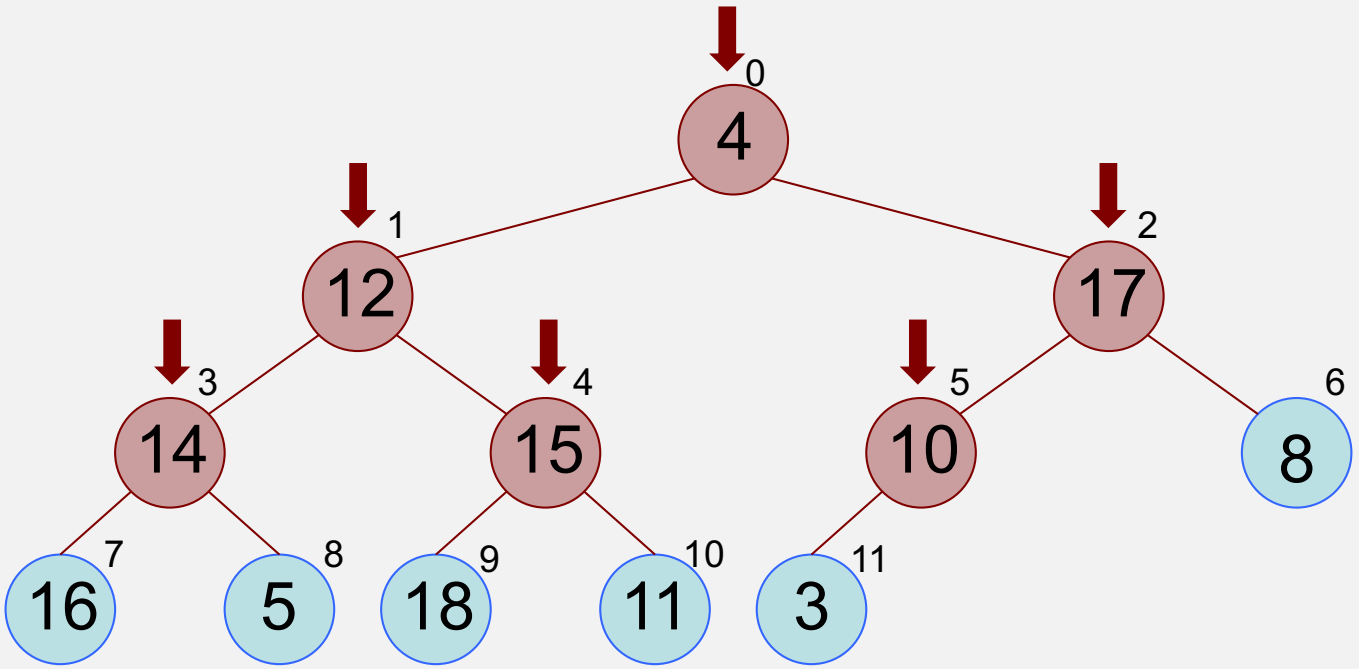
In questo esempio, `Buildheap` chiama `Heapify` sui nodi con indice 5, 4, 3, 2, 1, 0:



Funzione Buildheap (3)

```
def Build_heap(A):
  for i in reversed(range(len(A)//2)):
    Heapify (A, i, heap_size)
```

i=



Funzione Buildheap (4)

```
def Build_heap(A):
  for i in reversed(range(len(A)//2)):
    Heapify (A, i, heap_size)
```

La funzione `Build_heap` effettua Θ(n) chiamate di `Heapify`, che sappiamo avere ciascuna costo O(log n), quindi:

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

Con un calcolo più accurato si può mostrare che $T(n)=\Theta(n)$:

Funzione Buildheap (5)

Se |x| < 1:

$$\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$$

Mostriamo che T(n)=O(n).

- Il tempo richiesto da `Heapify` applicata ad un nodo che è radice di un albero di altezza h è O(h) per quanto già detto;
- il numero di nodi che sono radice di un sottoalbero di altezza h è al massimo $\left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Quindi: $T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$

Ricordando che $\sum_{h=0}^{\infty} \frac{h}{2^h} = \sum_{h=0}^{\infty} h \left(\frac{1}{2}\right)^h = \frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2} = 2$, si ha:

$$T(n) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right) = O\left(\frac{n}{2} \sum_{h=0}^{\infty} \left(\frac{1}{2}\right)^h\right) = O\left(\frac{n}{2} 2\right) = O(n)$$

Inserimento in un heap (1)

PROBLEMA: Dato uno heap H contenente n chiavi ed una nuova chiave k, inserire k in H.

SOLUZIONE: non va bene usare `BuildHeap()` perché lì partiamo dall'array disordinato e lo rendiamo interamente uno heap, mentre qui abbiamo già uno heap e dobbiamo inserire un singolo elemento.

Inserimento in un heap (2)

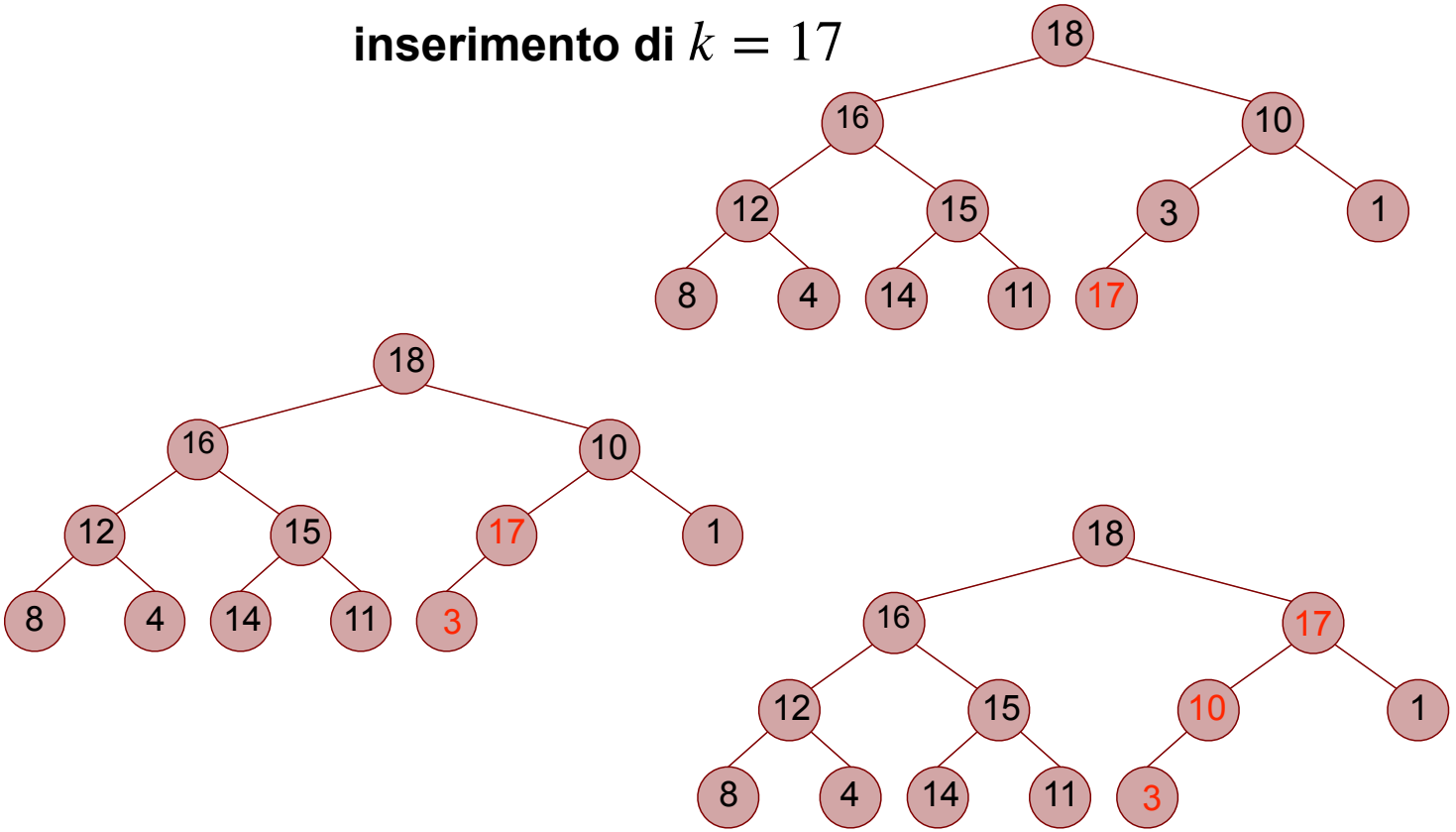
Possiamo procedere così:

- Inseriamo k come nuova foglia (a destra dell'ultimo elemento presente nell'heap)
- Ripristiniamo l'heap dal basso verso l'altro, partendo dalla nuova foglia e risalendo

NOTA: non possiamo usare `Heapify()` ma ci vuole una nuova funzione: `Heapify_bottom_up()`.

Inserimento in un heap (3)

Esempio:



Inserimento in un heap (4)

Pseudocodice (ricorsivo) della funzione `Heapify_bottom_up`:

```
def Heapify_bottom_up(A, heap_size)
  padre = ⌈heap_size/2⌉ − 1
  if (A[padre] < A[heap_size-1]):
    A[padre],A[heap_size-1]=A[heap_size-1],A[padre]
    if (padre > 0)
      Heapify_bottom_up (A, padre)
  return
```

Costo computazionale: O(log n)

Inserimento in un heap (5)

```
def Inserisci_in_Heap (A,heap_size,k):
  if (heap_size = len(A))
    scrivi "Errore: Heap pieno"
    return;
  A[heap_size] = k
  heap_size = heap_size+1
  if (heap_size > 0) //lo heap non era vuoto
    Heapify_bottom_up(A, heap_size)
  return
```

Costo computazionale: O(log n)

Nota su Heapify_bottom_up (1)

La funzione `Heapify_bottom_up` può essere sfruttata anche per **eliminare un elemento qualunque** A[x] (non necessariamente la radice) dallo heap oppure per **modificare arbitrariamente un qualunque elemento** A[x] dell'heap.

In entrambi i casi si deve:

- sostituire l'elemento da cancellare A[x] con la foglia più a destra A[n-1] oppure modificare A[x];
- riaggiustare lo heap a partire da A[x] sia verso il basso (con `Heapify`()) che verso l'alto (con `Heapify_bottom_up`()): …

Nota su Heapify_bottom_up (2)

- se A[x] è minore del maggiore dei suoi figli, si può applicare la funzione di aggiustamento dell'heap top-down (`Heapify`);
- se A[x] sembra in posizione corretta rispetto ai suoi figli, non è ancora detto che l'heap sia corretto, bisogna infatti confrontarlo con il padre: se questo è minore di A[x] si deve procedere all'aggiustamento bottom-up (`Heapify_bottom_up`).

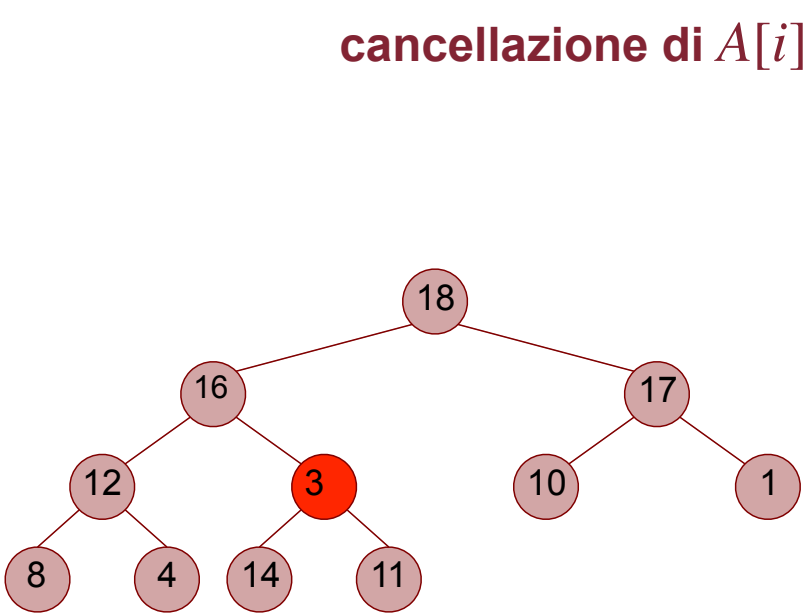
Nota: **si va verso l'alto oppure verso il basso, ma non è possibile che si debba andare in entrambe le direzioni.**

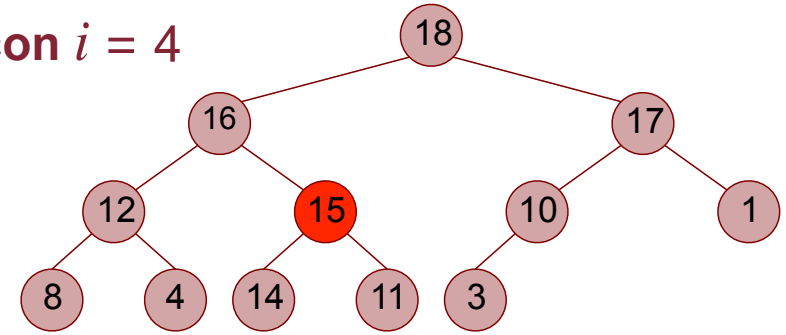
Costo computazionale: O(log n)

Nota su Heapify_bottom_up (3)

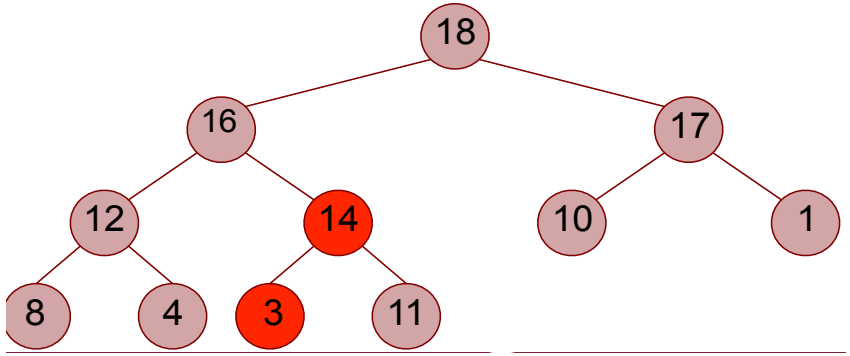
Esempio 1:

cancellazione di $A[i]$ con $i = 4$





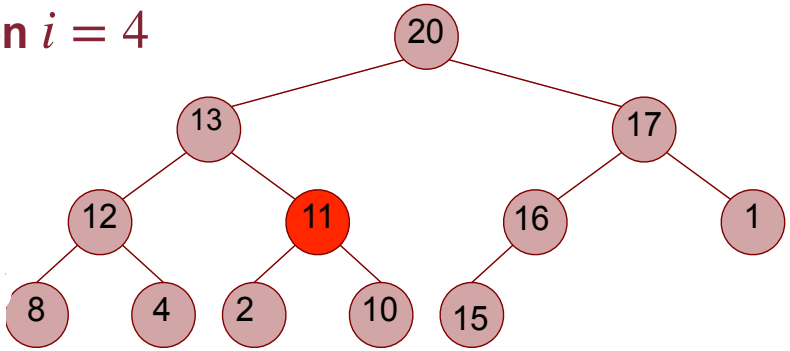
Dopo lo scambio, si aggiusta verso il basso:

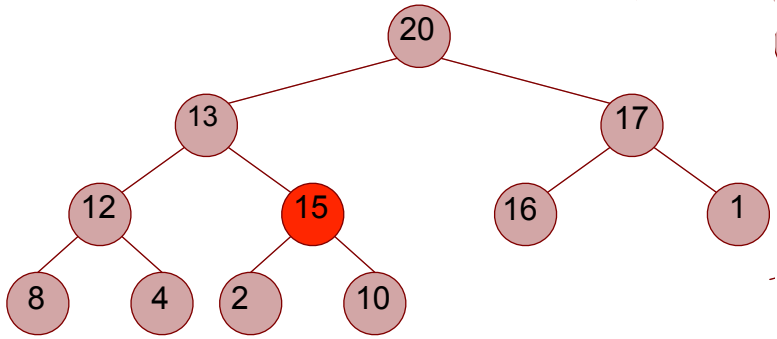


Nota su Heapify_bottom_up (4)

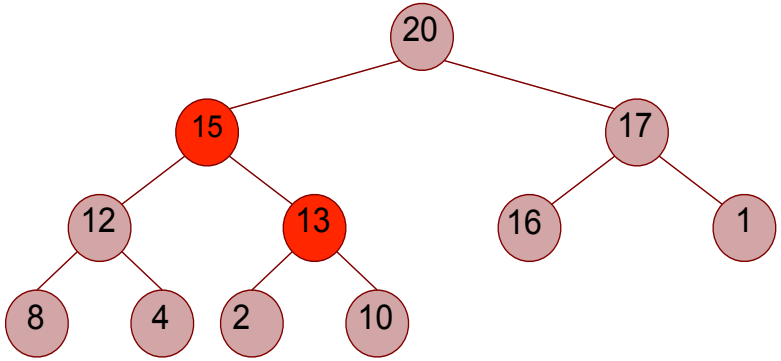
Esempio 2:

cancellazione di $A[i]$ con $i = 4$





Dopo lo scambio, si aggiusta verso l'alto:



La struttura dati Heap (10)

Abbiamo introdotto la struttura dati HEAP per parlare dell'Heap Sort, ma essa viene utilizzata indipendentemente dagli algoritmi di ordinamento e rientra, come vedremo in seguito, tra le CODE CON PRIORITA'.
E' utile ogni qual volta sia necessario estrarre da un insieme di dati il valore massimo (oppure minimo) ripetutamente.
Per questo:

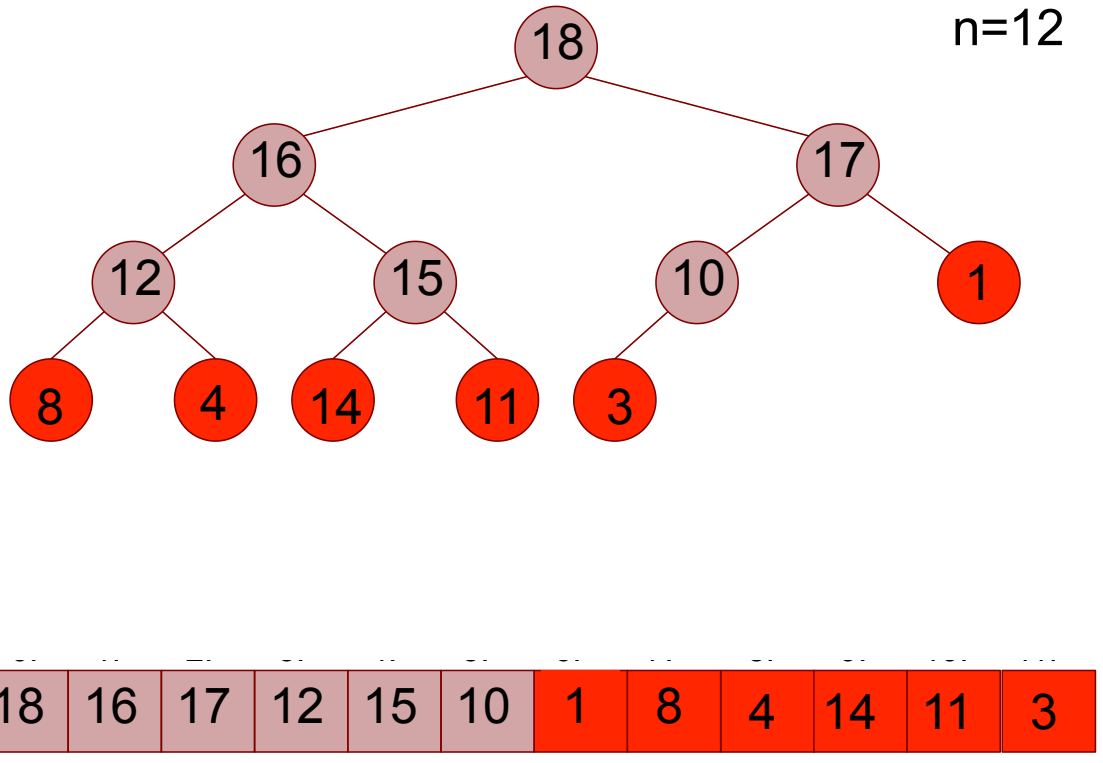
Ricerca del minimo (1)

PROBLEMA. Progettare un algoritmo che, dato in input un array che rappresenta uno heap, restituisca il valore minimo. Fare le opportune considerazioni sul costo computazionale.

Idea:

- Il minimo va cercato nelle foglie dello heap;
- non c'è alcuna proprietà dello heap che ci aiuti a trovare il minimo fra i valori contenuti nelle foglie

Ricerca del minimo (2)



Ricerca del minimo (3)

- sappiamo che le foglie stanno nell'array nelle posizioni da $\lfloor n/2 \rfloor$ (compreso) in avanti;
- quindi basta usare un ciclo che, ispezionando le posizioni dell'array da $\lfloor n/2 \rfloor$ a n-1 compresi, individui il minimo:

```
def minInHeap(A,n):
    return min([A[i] for i in
range(n//2,n)])
```

Costo computazionale: Θ(n).

Corso di laurea in Informatica
Algoritmi 1
A.A. 2025/26

Esercizi per casa

Esercizi (1)

- Progettare un algoritmo che, dato in input un array che rappresenta uno heap, restituisca il valore minimo. Fare le opportune considerazioni sul costo computazionale.
- Uno heap minimo è un albero binario completo o quasi completo con la proprietà che la chiave su ogni nodo è minore o uguale alla chiave dei suoi figli. Si modifichi l'algoritmo di Heap sort in modo che la struttura dati di riferimento sia uno heap minimo e non un heap.

Esercizi (2)

- Progettare un algoritmo che, dato in input un array che rappresenta uno heap ed un elemento x, inserisca x nello heap, collocandolo nella giusta posizione.
  Calcolare il costo computazionale.