ALGORITMI E STRUTTURE DATI

Prof. Manuela Montangero

A.A. 2022/23

STRUTTURE DATI:
Heap Binario

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

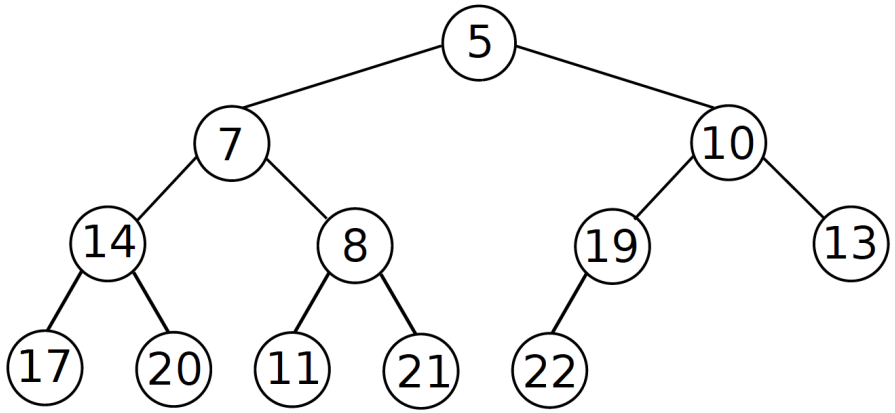
HEAP BINARIO

MIN-HEAP BINARIO:

Dato un insieme di chiavi totalmente ordinabile, un heap binario è un **albero binario** che memorizza chiavi dell'insieme e:

- Ogni nodo memorizza una sola chiave
- **PROPRIETÀ STRUTTURALE (topologia)**: l'albero è binario **completo quasi perfettamente bilanciato a sinistra**
- **PROPRIETÀ di ORDINAMENTO**: per ogni nodo v dell'albero, la chiave memorizzata in v è $\leq$ delle chiavi memorizzate nel sottoalbero radicato in v

L'albero è completo fino al penultimo livello. Le foglie dell'ultimo livello sono ammassate a sinistra.



HEAP BINARIO

MAX-HEAP BINARIO:

Dato un insieme di chiavi totalmente ordinabile, un heap binario è un **albero binario** che memorizza chiavi dell'insieme e:

- Ogni nodo memorizza una sola chiave
- **PROPRIETÀ STRUTTURALE (topologia)**: l'albero è binario **completo quasi perfettamente bilanciato a sinistra**
- **PROPRIETÀ di ORDINAMENTO**: per ogni nodo v dell'albero, la chiave memorizzata in v è $\geq$ delle chiavi memorizzate nel sottoalbero radicato in v

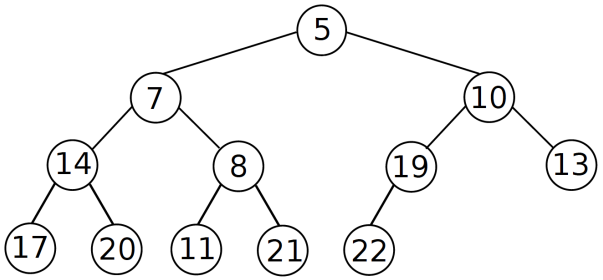
L'albero è completo fino al penultimo livello. Le foglie dell'ultimo livello sono ammassate a sinistra.

HEAP BINARIO

PROPOSIZIONE:

Un Heap con n nodi ha altezza $h = \lfloor \log n \rfloor$

REMINDER:
un albero binario completo perfettamente bilanciato di altezza h ha esattamente $n = 2^{h+1} - 1$ nodi



$n = 12, \lfloor \log 12 \rfloor = 3$

- L'heap ha almeno tanti nodi quanti ne ha un albero binario completo perfettamente bilanciato di altezza $h - 1$

$$n > 2^{(h-1)+1} - 1 = 2^h - 1$$

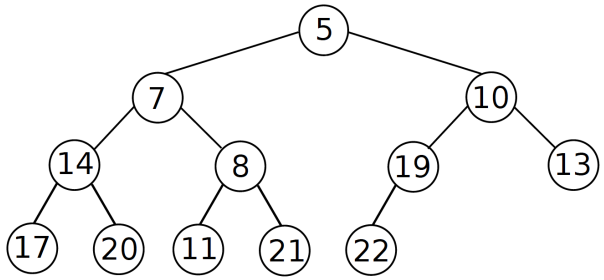
- L'heap non ha più nodi di quanti ne abbia un albero binario completo perfettamente bilanciato di altezza h

$$n \leq 2^{h+1} - 1$$

HEAP BINARIO

PROPOSIZIONE:

Un Heap con n nodi ha altezza $h = \lfloor \log n \rfloor$



$n = 12, \lfloor \log 12 \rfloor = 3$

$$n \leq 2^{h+1} - 1$$

$$n + 1 \leq 2^{h+1}$$

$$\log(n+1) \leq h + 1$$

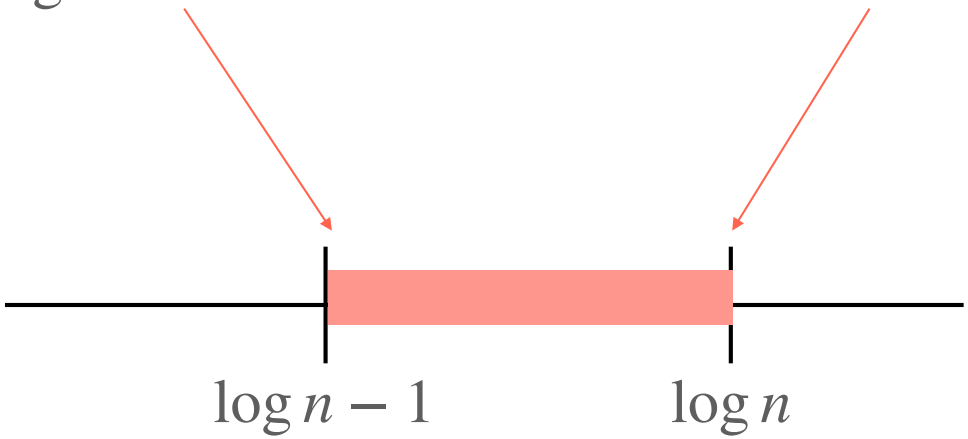
$$\log(n+1) - 1 \leq h$$

$$\log n - 1 < h$$

$$n > 2^h - 1$$

$$n \geq 2^h$$

$$\log n \geq h$$

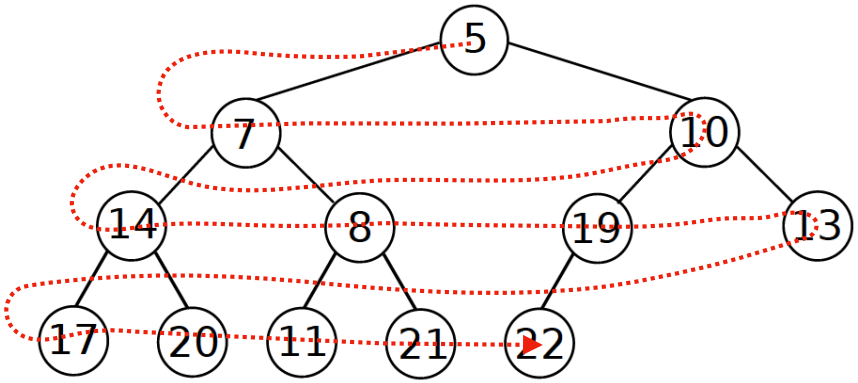


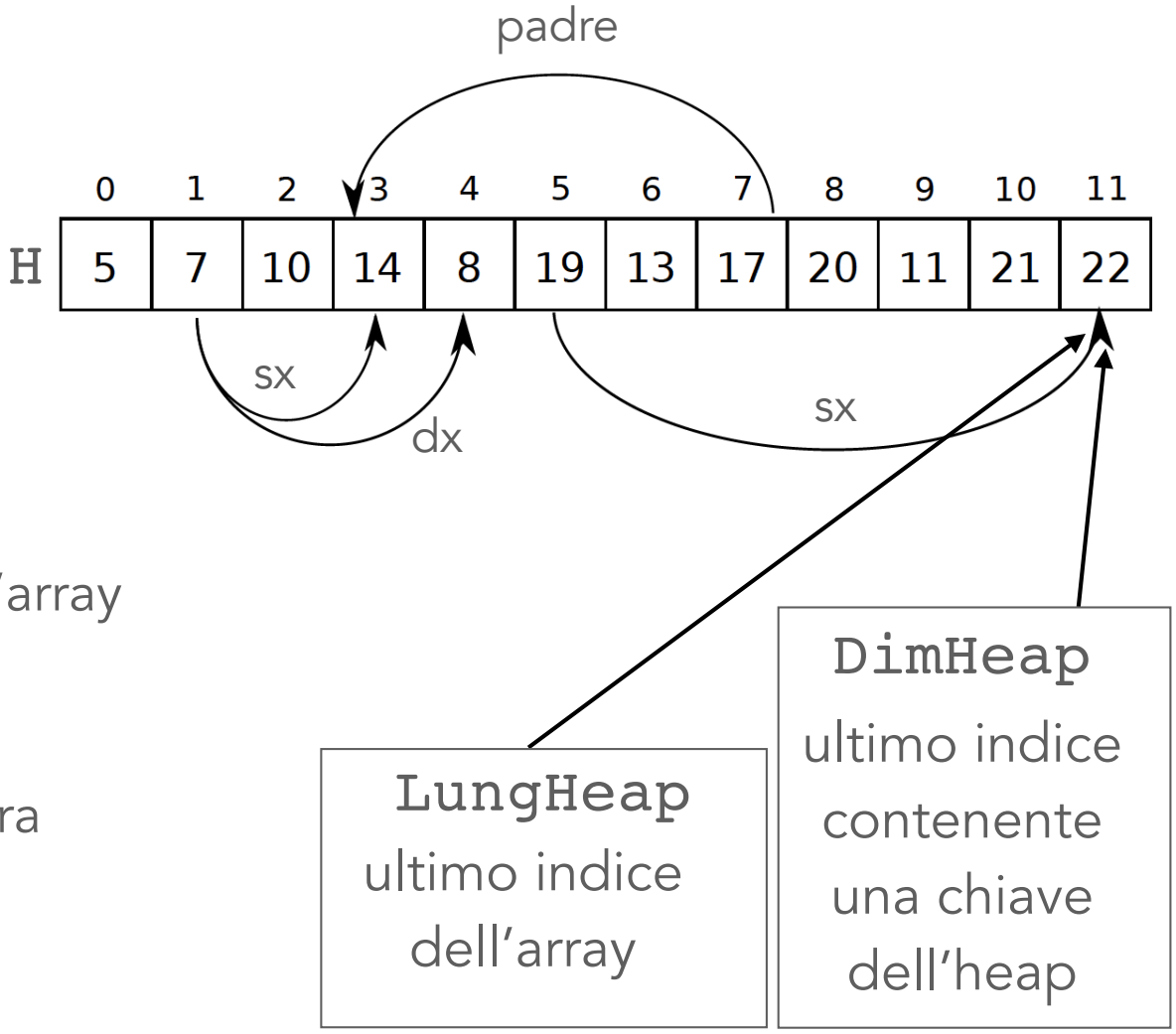
$\Rightarrow$ $h = \lfloor \log n \rfloor$

Perché h deve essere un intero

Rappresentazione implicita

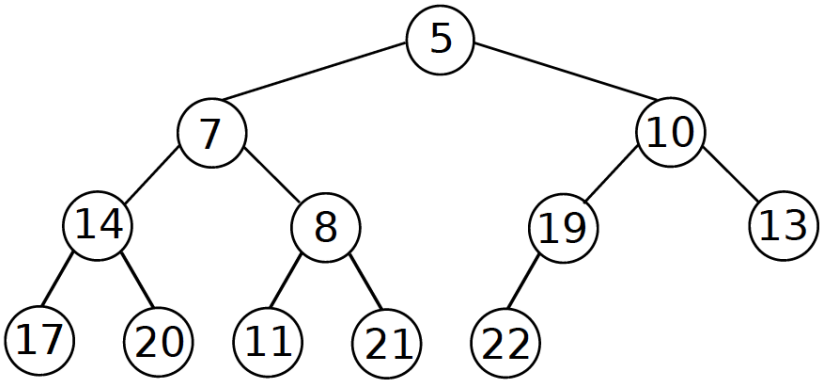
min-heap

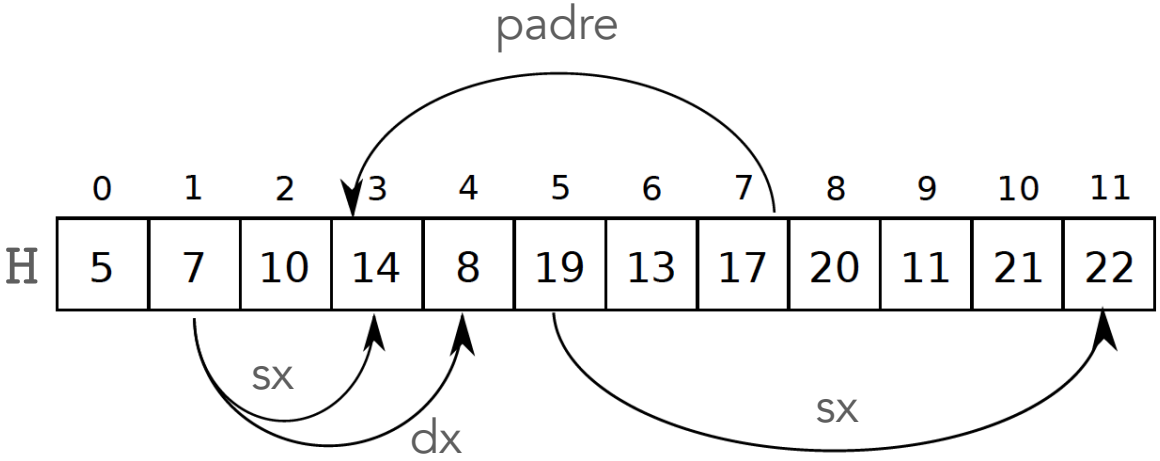




Le chiavi dell'heap sono inserite nell'array
livello per livello
e
per ogni livello, da sinistra a destra

Rappresentazione implicita






```
Padre(i)
  return ⌈i/2⌉ − 1
```

```
FiglioSinistro(i)
  return 2i + 1
```

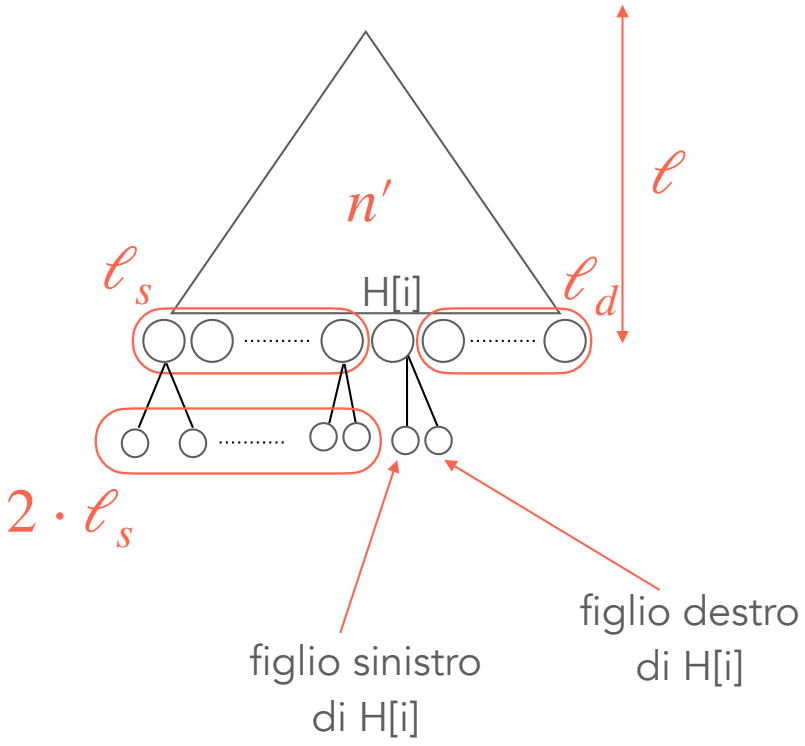
```
FiglioDestro(i)
  return 2(i + 1)
```

Rappresentazione implicita

```
Padre(i)
  return ⌈i/2⌉ − 1
```

```
FiglioSinistro(i)
  return 2i + 1
```

```
FiglioDestro(i)
  return 2(i + 1)
```



$n' = 2^{(\ell-1)+1} - 1 = 2^{\ell} - 1$ #nodi fino livello $\ell - 1$

2^{ℓ} #nodi a livello ℓ $\Rightarrow$ $2^{\ell} = n' + 1$

$i = n' + \ell_s$ posizione i

posizione del figlio sinistro: $i + \ell_d + 2 \cdot \ell_s + 1$

posizione del figlio destro: $i + \ell_d + 2 \cdot \ell_s + 2$

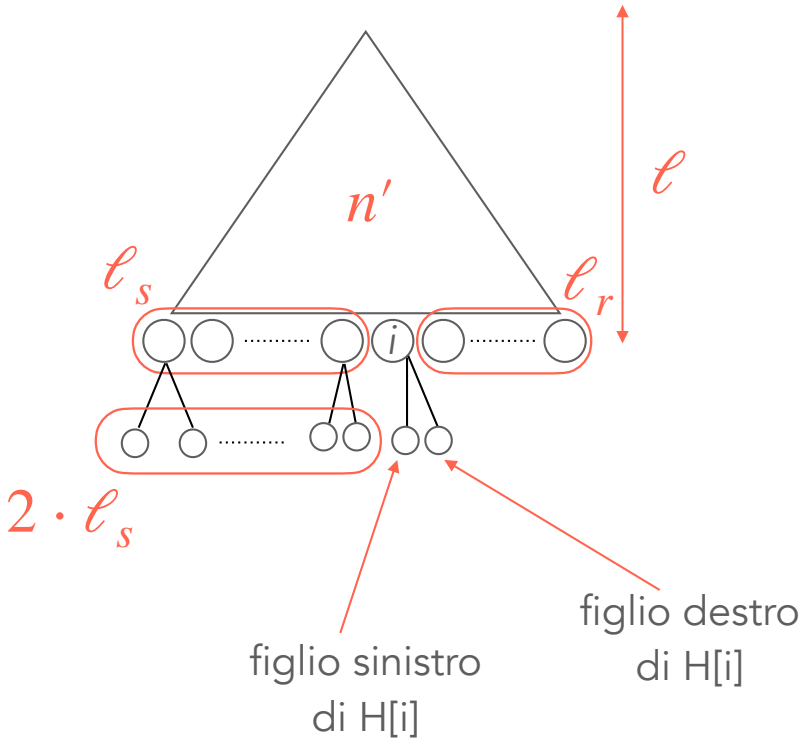
$$i + \ell_d + 2 \cdot \ell_s + 1 = i + \underbrace{(\ell_d + \ell_s + 1)}_{\text{\#nodi a livello } \ell} + \ell_s = i + (n' + 1) + \ell_s = i + \underbrace{(n' + \ell_s)}_{= i} + 1 = 2i + 1$$

Rappresentazione implicita

```
Padre(i)
  return ⌈i/2⌉ − 1
```

```
FiglioSinistro(i)
  return 2i + 1
```

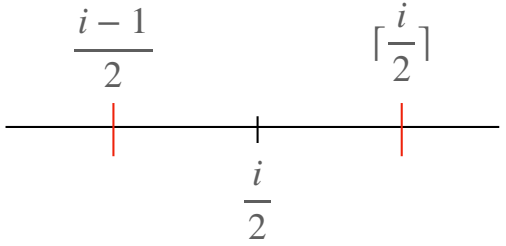
```
FiglioDestro(i)
  return 2(i + 1)
```



Se *i* è il figlio sinistro di *j*

dispari

$$i = 2j + 1 \Rightarrow j = \frac{i-1}{2} = \lceil \frac{i}{2} \rceil - 1$$



Se *i* è il figlio destro di *j*

pari

$$i = 2(j+1) \Rightarrow j = \frac{i}{2} - 1 = \lceil \frac{i}{2} \rceil - 1$$

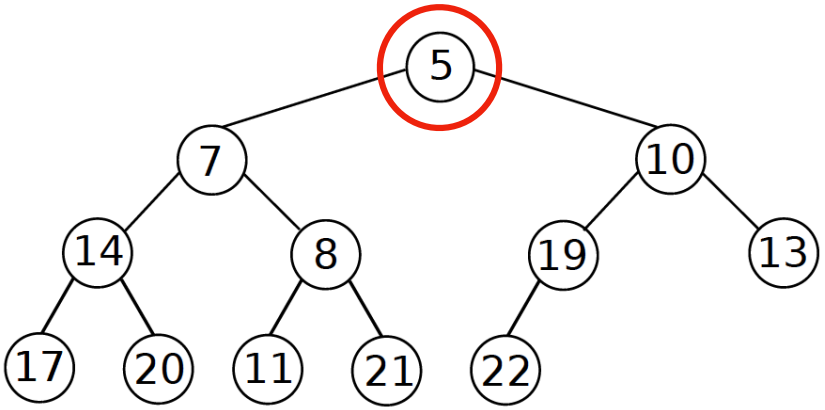
Primitive HEAP

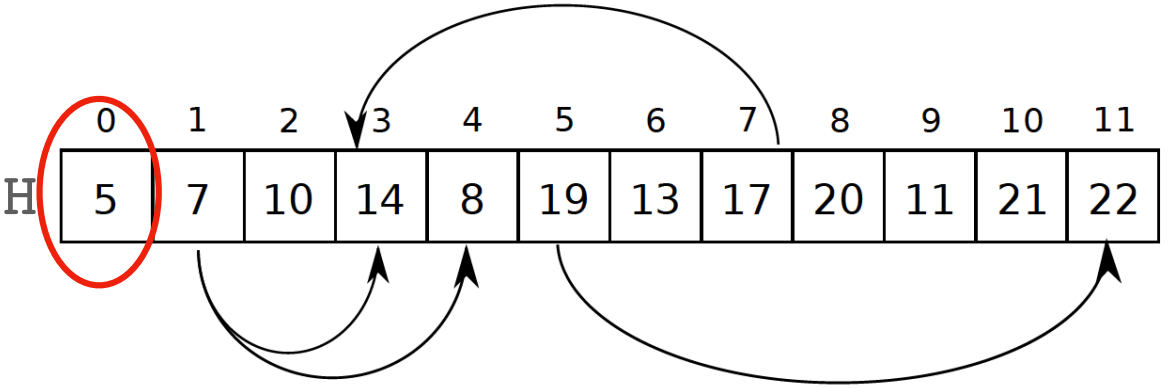
PRIMITVE MIN-HEAP (assumendo rappresentazione implicita con array):

Intestazione	Descrizione
CostruisciHeap(H)	**Costruire un heap** a partire da un insieme di chiavi memorizzate nell'array H
MinimoHeap(H) $\rightarrow k$	**Restituire la chiave minima** memorizzata nell'heap H
DecrementaChiaveHeap(H, i, k)	**Decrementare il valore della chiave** all'indice i nell'heap H, impostando il suo valore a k
InserzioneChiaveHeap(H, k)	**Inserire la nuova chiave** k nell'heap H
Heapify(H, i)	**Ripristinare la proprietà di ordinamento** di un heap in seguito all'incremento di una (sola) chiave (quella in posizione i) già presente nell'heap H
EstrazioneMinimoHeap(H) $\rightarrow k$	**Restituire la chiave minima ed eliminarla** dall'heap H

Primitive HEAP

`MinimoHeap(H)`: restituisce la chiave minima memorizzata nell'heap





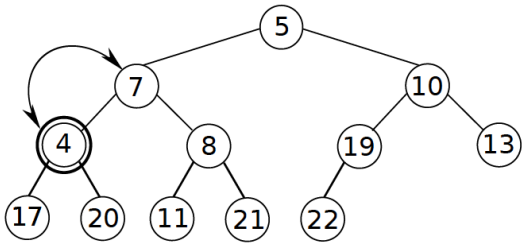

```
MinimoHeap(H)
 if DimHeap < 0
  then return error
  else return H[0]
```

Costo computazionale $O(1)$

Primitive HEAP

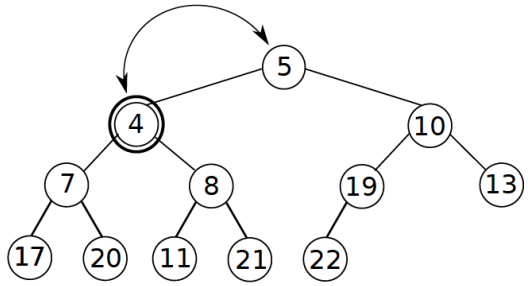
`DecrementaChiaveHeap(H,i,k)`: decrementa il valore della chiave all'indice `i` dell'heap, impostando il valore a `k`

(a) `DecrementaChiaveHeap(H,3,4)`



0	1	2	3	4	5	6	7	8	9	10	11
5	7	10	4	8	19	13	17	20	11	21	22

(b)

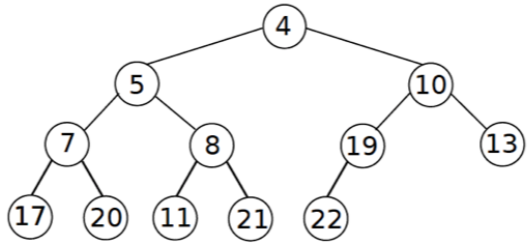


0	1	2	3	4	5	6	7	8	9	10	11
5	4	10	7	8	19	13	17	20	11	21	22

Costo computazionale

$O(\log n)$

(c)



0	1	2	3	4	5	6	7	8	9	10	11
4	5	10	7	8	19	13	17	20	11	21	22

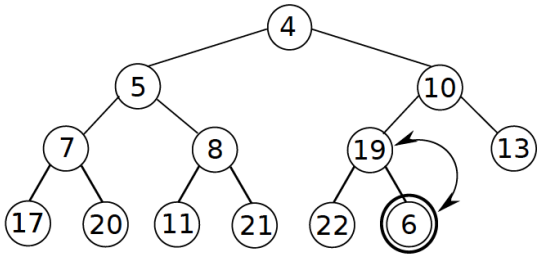
```
DecrementaChiaveHeap(H,i,k)
if i > DimHeap OR k > H[i]
  then return error
H[i] := k
while i > 0 AND H[Padre(i)] > k do
  Scambia H[i] con H[Padre(i)]
  i := Padre(i)
```

Primitive HEAP

`InserzioneChiave(H,k)`: inserisce la chiave nell'heap

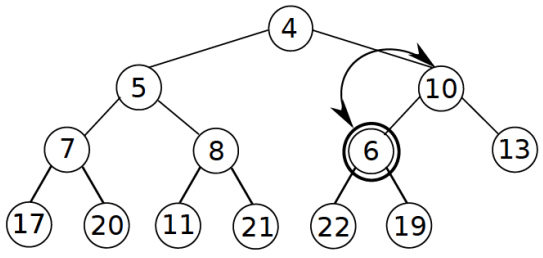
`InserzioneChiave(H,6)`

(a)



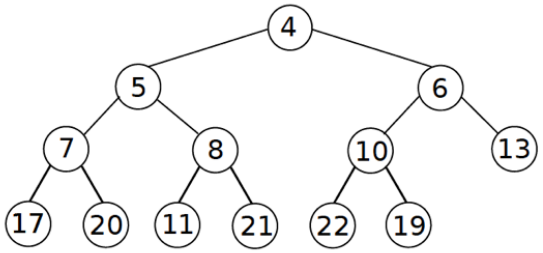
0	1	2	3	4	5	6	7	8	9	10	11	12
4	5	10	7	8	19	13	17	20	11	21	22	6

(b)



0	1	2	3	4	5	6	7	8	9	10	11	12
4	5	10	7	8	6	13	17	20	11	21	22	19

(c)



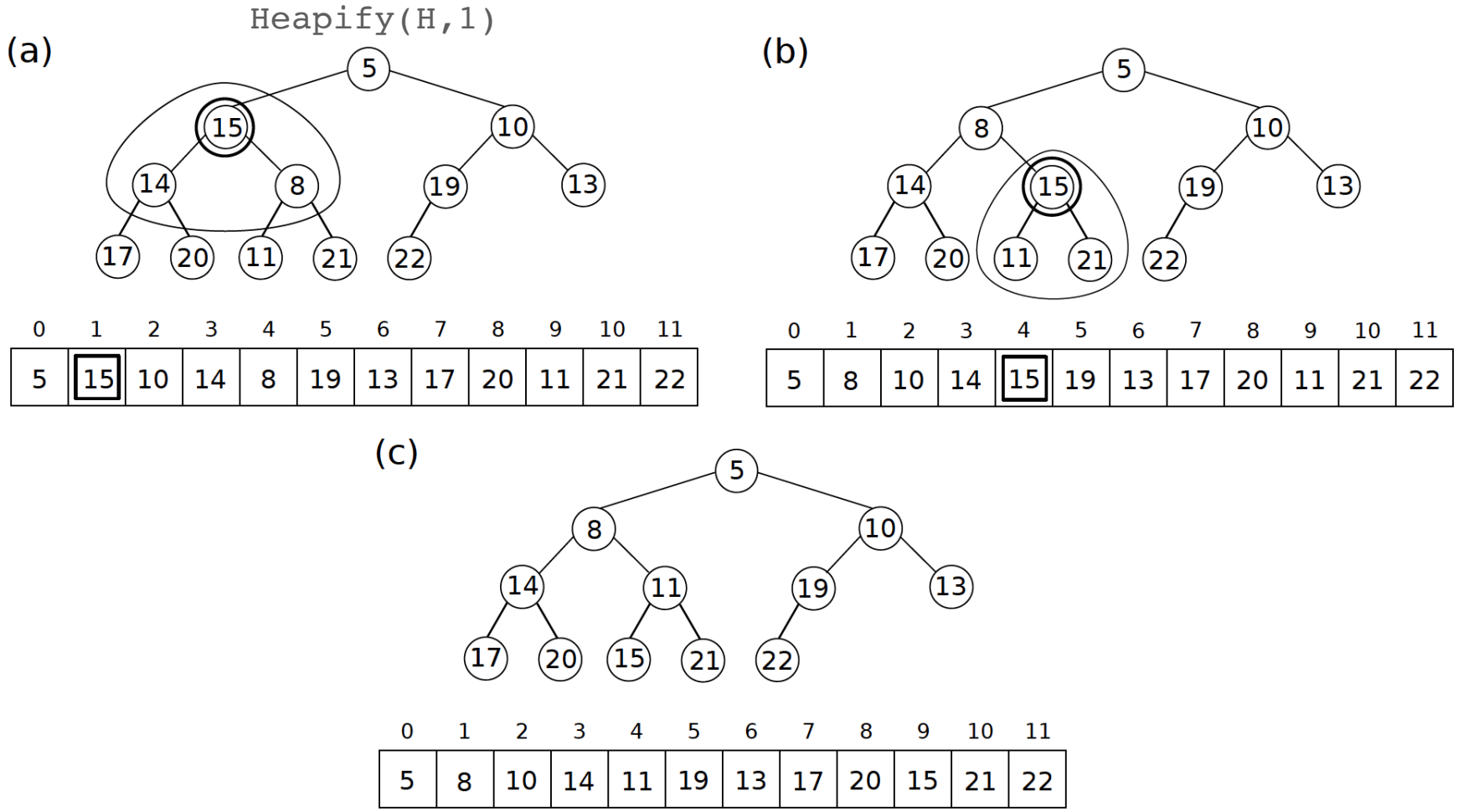
0	1	2	3	4	5	6	7	8	9	10	11	12
4	5	6	7	8	10	13	17	20	11	21	22	19

```
InserzioneChiaveHeap(H,k)
if DimHeap = LungHeap
  then return error
DimHeap := DimHeap + 1
H[DimHeap] := k
DecrementaChiaveHeap(H,DimHeap,k)
```

Costo computazionale $O(\log n)$

Primitive HEAP

`Heapify(H,i)`: ripristina la proprietà di ordinamento di un heap `H` a seguito dell'incremento della chiave in posizione `i`



Primitive HEAP

`Heapify(H,i)`: ripristina la proprietà di ordinamento di un heap `H` a seguito dell'incremento della chiave in posizione `i`

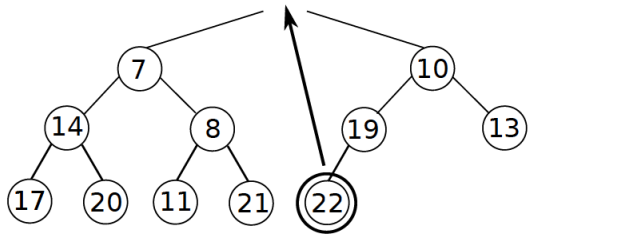
```
Heapify(H,i)
if i > DimHeap then return error
min := i
s := FiglioSinistro(i)
d := FiglioDestro(i)
// Confronto con il filgio sinistro
if s ≤ DimHeap AND H[s] < H[i]
 then min := s
// Confronto con il figlio destro
if d ≤ DimHeap AND H[d] < H[min]
 then min := d
if min ≠ i
 then
   scambia H[i] con H[min]
   // ripeti nel sottoalbero radicato in min
   Heapify(H,min)
```

Costo computazionale $O(\log n)$

Primitive HEAP

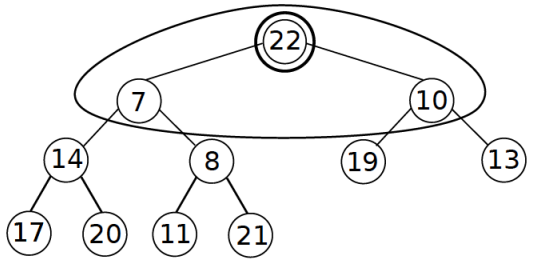
`EstrazioneMinimoHeap(H)`

(a)



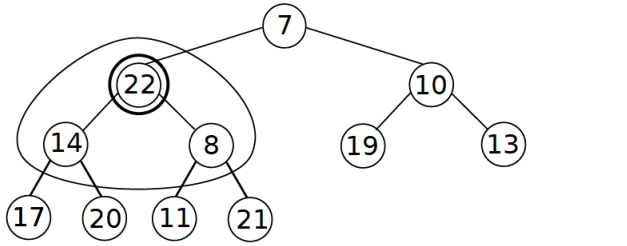
0	1	2	3	4	5	6	7	8	9	10	11
	7	10	14	8	19	13	17	20	11	21	22

(b)



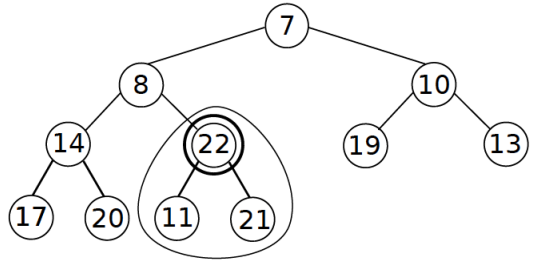
0	1	2	3	4	5	6	7	8	9	10	11
22	7	10	14	8	19	13	17	20	11	21	

(c)



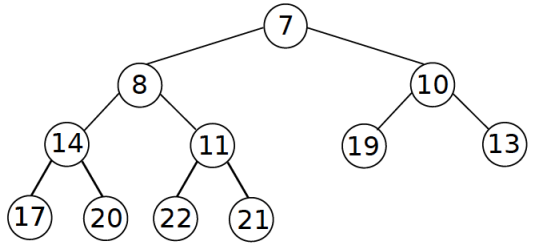
0	1	2	3	4	5	6	7	8	9	10	11
7	22	10	14	8	19	13	17	20	11	21	

(d)



0	1	2	3	4	5	6	7	8	9	10	11
7	8	10	14	22	19	13	17	20	11	21	

(e)



0	1	2	3	4	5	6	7	8	9	10	11
7	8	10	14	11	19	13	17	20	22	21	

Primitive HEAP

`EstrazioneMinimoHeap(H)`: restituisce il minimo valore memorizzato nell'heap e lo elimina dall'heap

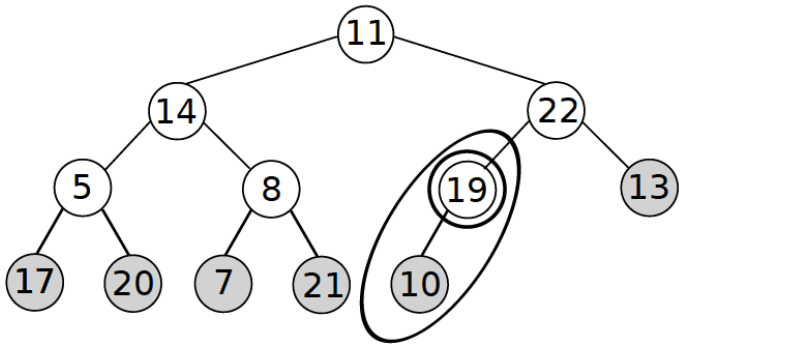
```
EstrazioneMinimoHeap(H)
if DimHeap < 0
 then return error
min := H[0]
H[0] := H[DimHeap]
DimHeap := DimHeap - 1
Heapify(H,0)
return min
```

Costo computazionale $O(\log n)$

Primitive HEAP

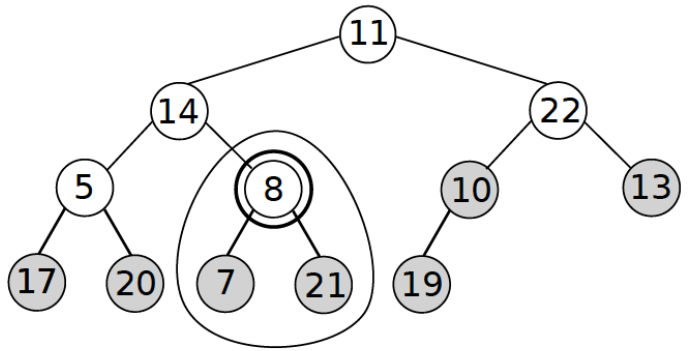
`CostruisciHeap(H)`

(a)



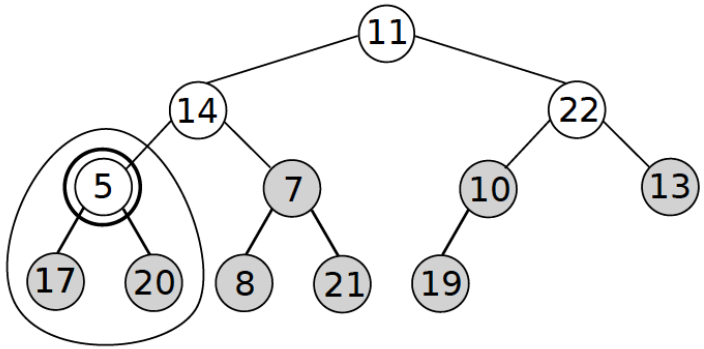
0	1	2	3	4	5	6	7	8	9	10	11
11	14	22	5	8	19	13	17	20	7	21	10

(b)



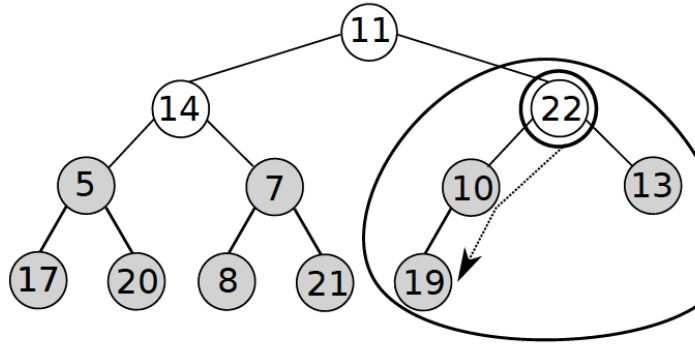
0	1	2	3	4	5	6	7	8	9	10	11
11	14	22	5	8	10	13	17	20	7	21	19

(c)



0	1	2	3	4	5	6	7	8	9	10	11
11	14	22	5	7	10	13	17	20	8	21	19

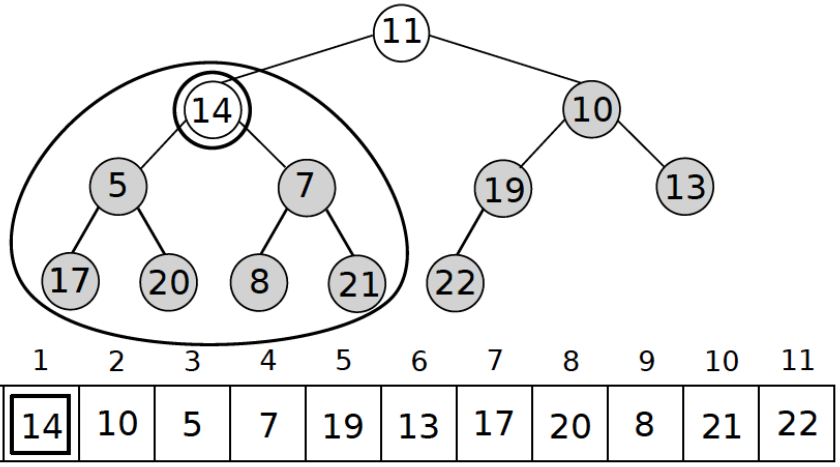
(d)



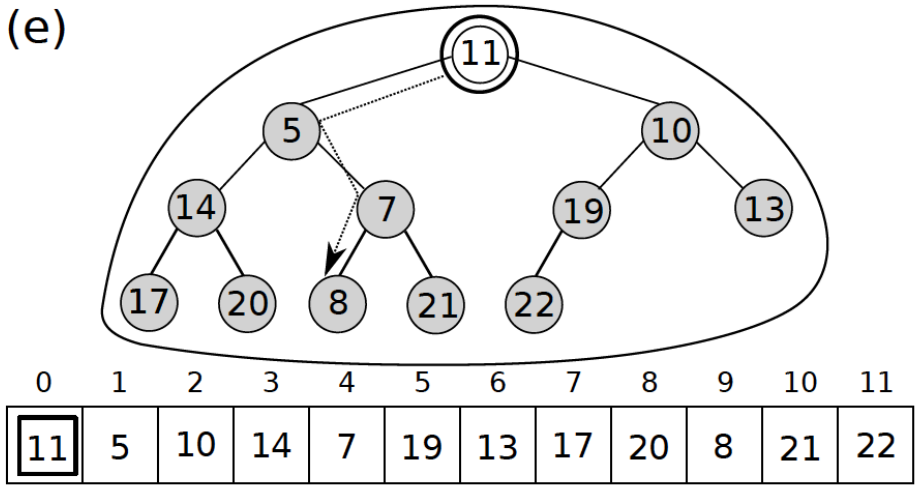
0	1	2	3	4	5	6	7	8	9	10	11
11	14	22	5	7	10	13	17	20	8	21	19

Primitive HEAP

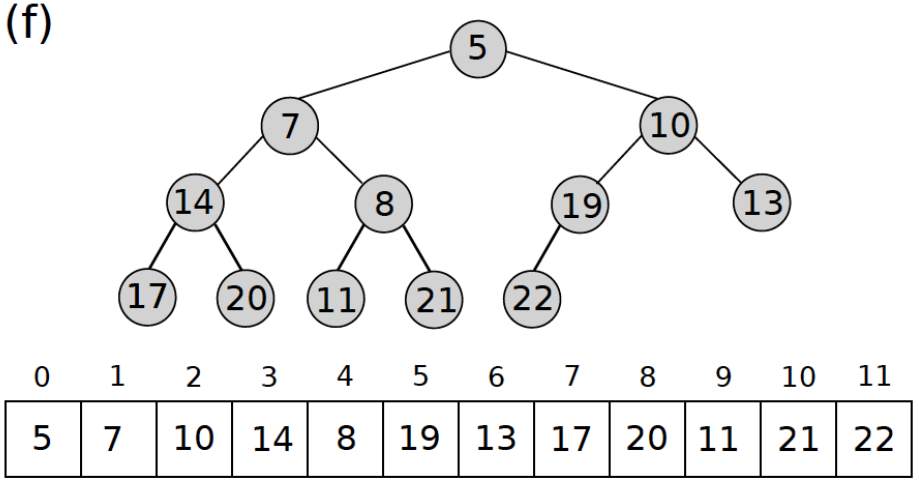
(e)



(e)



(f)




```
CostruisciHeap(H)
DimHeap := length(H)-1
LunghHeap := length(H)-1
for i = Padre(DimHeap) downto 0
 Heapify(H,i)
```

Primitive HEAP

`CostruisciHeap(H)`: costruisce un heap che memorizza le chiavi nell'array `H`

```
CostruisciHeap(H)
DimHeap := length(H)-1
LunghHeap := length(H)-1
for i = Padre(DimHeap) downto 0
 Heapify(H,i)
```

Costo computazionale $O(n \log n)$

possiamo fare un'analisi più accurata?

Primitive HEAP

`CostruisciHeap(H)`: costruisce un heap che memorizza le chiavi nell'array `H`

```
CostruisciHeap(H)
DimHeap := length(H)-1
LunghHeap := length(H)-1
for i = Padre(DimHeap) downto 0
 Heapify(H,i)
```

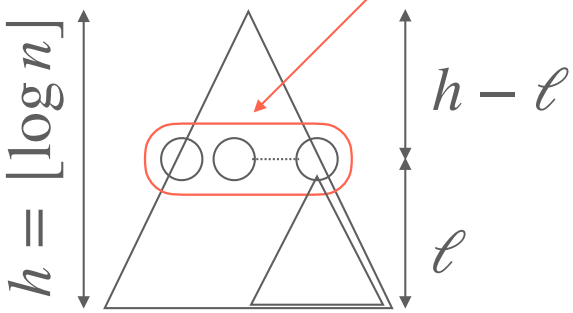
con un'analisi più accurata:
"heapify viene chiamata *molte* volte su heap bassi,
poche volte su heap alti"

Costo computazionale $O(n)$

Primitive HEAP

`CostruisciHeap(H)`: costruisce un heap che memorizza le chiavi nell'array `H`

Un albero binario completo e perfettamente bilanciato di altezza h ha $2^{h-\ell} = \left\lceil \frac{n}{2^{\ell+1}} \right\rceil$ nodi interni a livello $h - \ell$, per $\ell = 1, \ldots, h$



$$\sum_{\ell=1}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{\ell+1}} \right\rceil \cdot O(\ell) = O\left(n \sum_{\ell=2}^{\lfloor \log n \rfloor + 1} \frac{\ell}{2^{\ell}} \right) = O(n)$$

costo di `Heapify`

FACOLTATIVO

$$\sum_{\ell=2}^{\lfloor \log n \rfloor + 1} \frac{\ell}{2^{\ell}} \leq \sum_{\ell=0}^{\infty} \frac{\ell}{2^{\ell}} = \sum_{\ell=0}^{\infty} \ell \left(\frac{1}{2} \right)^{\ell} = \frac{\frac{1}{2}}{(1-\frac{1}{2})^2} = 2$$