Dati e Algoritmi: A. Pietracaprina

Priority Queue

Nozione di Entry

Definizione

Una Entry è una coppia (chiave, valore), dove la chiave proviene da un dominio K e il valore da un dominio V.

```
public interface Entry<K,V> {
  /** Returns the key of the entry */
  K getKey();
  /** Returns the value of the entry */
  V getValue();
}
```

Esempio:

ENTRY ≡ STUDENTE
- key = #matricola
- value = info come anagrafica, esami, ecc.

Priority Queue

Definizione

Priority Queue: collezione di entry le cui chiavi rappresentano *priorità* e provengono da un universo totalmente ordinato K.

Come tipo di dato astratto, la Priority Queue deve permettere di

- Trovare/rimuovere la entry di massima priorità.
- Inserire una nuova entry.

Le chiavi delle entry non sono necessariamente distinte

N.B.: convenzionalmente si assume che più piccolo è il valore della chiave e più alta è la priorità ma vedremo anche utilizzi in cui vale l'opposto.

Priority Queue: interfaccia

```
public interface PriorityQueue<K,V> {
  int size();
  boolean isEmpty();
  /** Inserts and returns a new entry (key,value) */
  Entry<K,V> insert(K key, V value);
  /** Returns an entry with min key, without removing it */
  Entry<K,V> min();
  /** Returns and removes an entry with min key */
  Entry<K,V> removeMin();
}
```

Osservazione: Se esistono più entry con chiave minima, `min` e `removeMin` ne restituiscono (e `removeMin` ne rimuove) una arbitraria

Esempio

Sequenza di operazioni a partire da una Priority Queue vuota:

Operazione	Output	Priority Queue risultante
`insert`(5,A)	(5,A)	(5,A)
`insert`(9,C)	(9, C)	(5,A)(9, C)
`insert`(3,B)	(3,B)	(5,A) (9,C)(3, B)
`insert`(7,D)	(7, D)	(5,A)(9,C)(3,B)(7,D)
`min`()	(3,B)	(5,A)(9,C)(3,B)(7,D)
`removeMin`()	(3,B)	(5,A)(9,C) (7, D)
`size`()	3	11
`isEmpty`()	FALSE	11

Applicazioni delle priority queue

- Algoritmo di Dijkstra per trovare i cammini minimi su un grafo
- Pattern discovery
- Scheduling di processi in sistema operativo o di richieste di banda nelle reti in base a priorità per garantire QoS
- Simulazione discreta a eventi

Esercizio

Sia $S = S[0], S[1], \ldots, S[n-1]$ una sequenza di n entry con chiavi intere distinte (ad es., profili IG con i rispettivi numeri di follower), e sia τ un intero in $[1, n]$. Il seguente algoritmo (da completare) trova le τ entry in S con chiave maggiore (*Top-τ entry*).

Algoritmo `Top`(S,τ)
Input: Sequenza S di n entry, intero $\tau \leq n$
Output: τ entry di S con chiave più grande
Q $\longleftarrow$ priority queue vuota;
for $i \leftarrow 0$ *to* $n-1$ **do**
    if $(i < \tau)$ **then** Q.insert($S[i]$.getKey(),$S[i]$.getValue());
    else
return Q

Esercizio (Continua)

1. Completare l'algoritmo in modo che sia corretto e che Q non contenga mai più di τ entry.
2. Provare la correttezza dell'algoritmo usando un opportuno invariante per il ciclo for.

Osservazione

In generale, il problema di trovare le Top-τ entry rappresenta una primitiva fondamentale nella data analysis. L'algoritmo dell'esercizio è efficiente in quanto non ricorre all'ordinamento e può essere utilizzato anche in applicazioni in cui la sequenza S di input non può essere salvata in memoria (perché molto grande oppure fornita "on-the-fly" come stream).

Implementazione di Priority Queue tramite Liste

Implementazione di Priority Queue: Lista non ordinata

Sia P una lista NON ORDINATA di n entry
(PositionalList<Entry< K, V >>)

Implementazione dei metodi della Priority Queue:

Metodo `min()`

```
v ←P.first();
e ← v.getElement();
while (P.after(v) ≠ null) do
    v ←P.after(v);
    if (v.getElement().getKey() < e.getKey()) then
        e ← v.getElement()
return e
```

Complessità = $\Theta(n)$ Devo vedere tutte le entry per trovare quella con chiave minima

Metodo `insert(`k`, `x`)`

$e \leftarrow (k, x)$;

`P.addLast(`e`);`

return `e`

Complessità = $\Theta(1)$ Posto inserire dove voglio

Metodo `removeMin()` (esercizio)

Complessità = $\Theta(n)$ come il caso di min

Implementazione di Priority Queue: Lista ordinata

Sia P una lista ORDINATA (in senso crescente) di n entry
(PositionalList<Entry< K, V >>)

Implementazione dei metodi della Priority Queue:

Metodo `min()`
return `P.first().getElement()`

Complessità = $\Theta(1)$

Metodo `removeMin()` (esercizio)

Complessità = $\Theta(1)$

} Una entry con chiave minima è in testa

Metodo `insert(`k`, `x`)`

```
e ← (k, x);
v ←P.first();
while v ≠ null do
    if (v.getElement().getKey()≥ k) then
        P.addBefore(v,e);
        return e
    else
        v ←P.after(v)
P.insertLast(e);
return e
```

Complessità = $O(n)$ Devo trovare la posizione in cui inserire la entry

Priority Queue tramite Liste: riepilogo

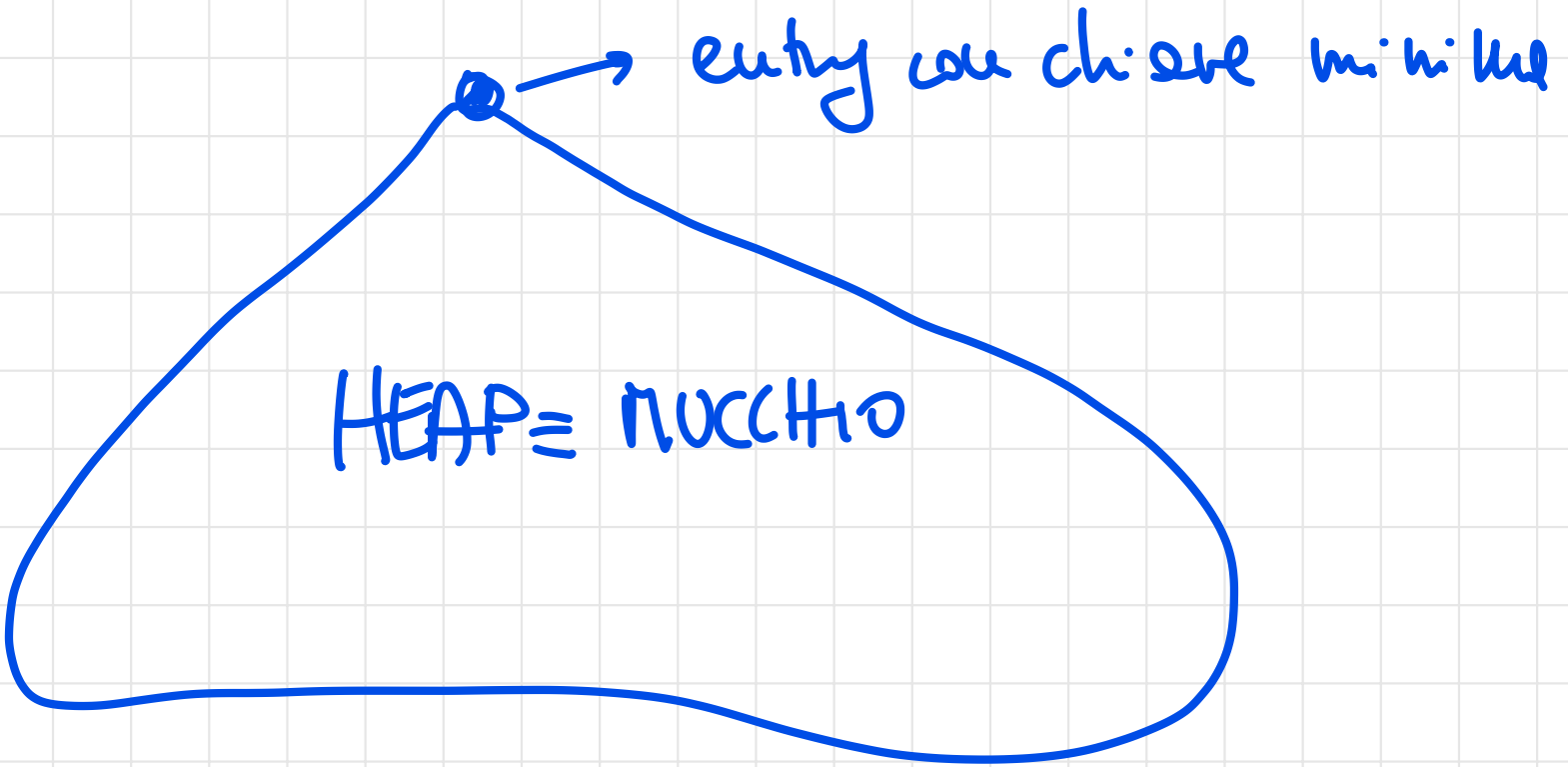
Lista	**insert**	**min**	**removeMin**
NON ORDINATA	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
ORDINATA	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$

È possibile *bilanciare* il costo delle diverse operazioni?

Oss.: per le strutture dati non discutiamo l'implementazione dei metodi `size` e `isEmpty` in quanto sono banali

Implementazione di Priority Queue tramite Heap

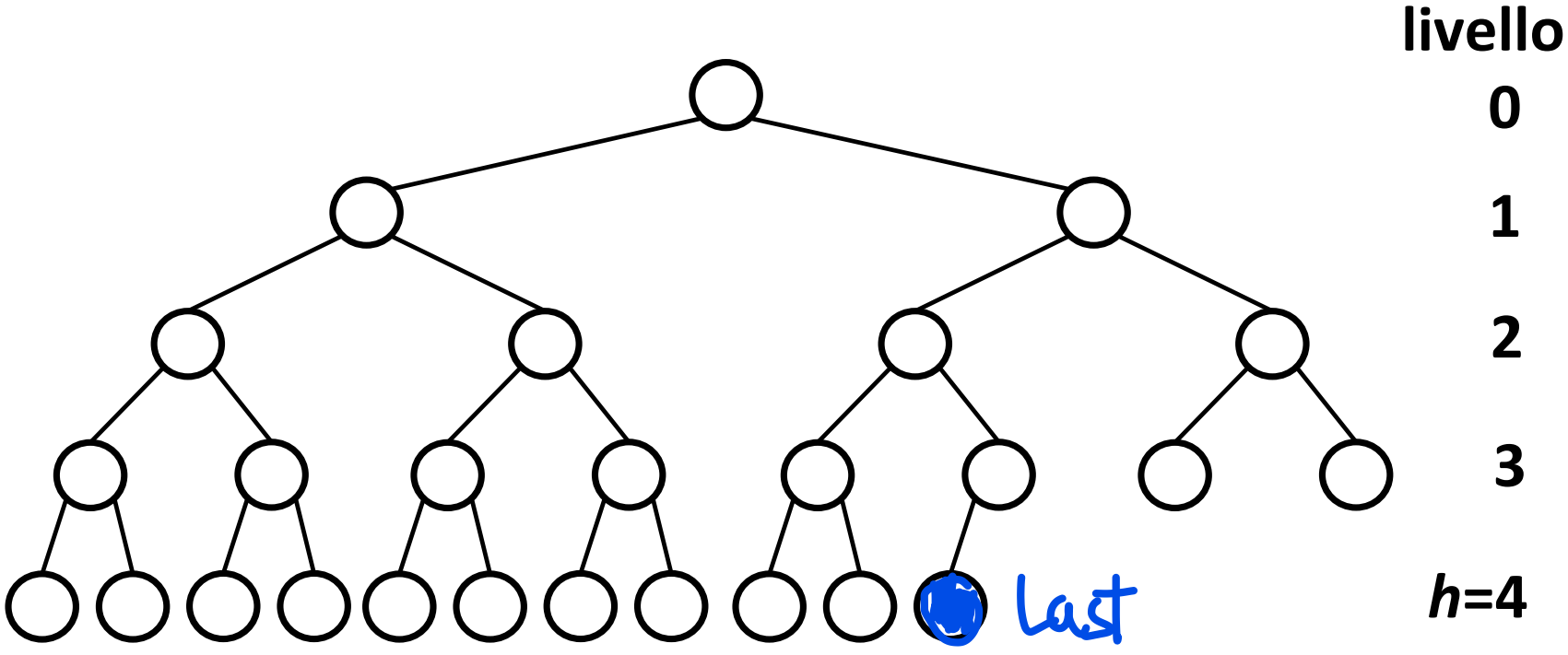
Idea



Albero Binario Completo T

Albero binario di altezza $h \geq 0$ tale che

- $\forall i, 0 \leq i \leq h-1$: il livello i ha 2^i nodi (=max numero di nodi)
- al livello $h-1$ tutti i nodi interni sono alla sx delle eventuali foglie e hanno tutti 2 figli tranne, eventualmente, quello più a dx che, se ha un solo figlio, ha il figlio sx.



Proposizione

Un albero binario completo con n nodi ha altezza $h = \lfloor \log_2 n \rfloor$.

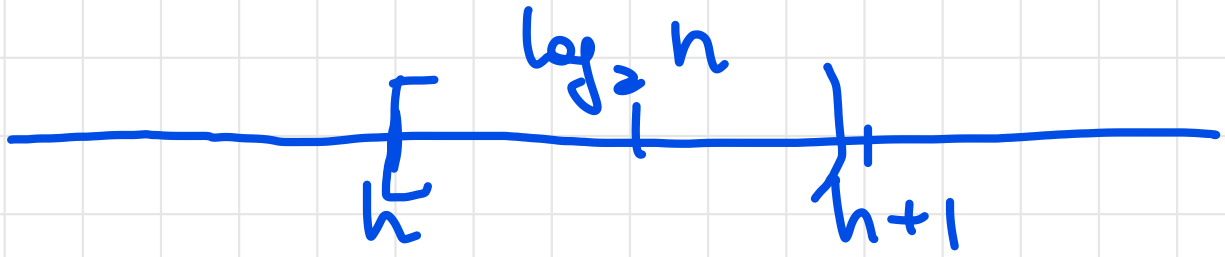
Dimostrazione

Oss. Al livello h (ultimo livello dell'albero) il numero di nodi è compreso tra 1 e 2^h

$$1 + \sum_{i=0}^{h-1} 2^i \leq n \leq 2^h + \sum_{i=0}^{h-1} 2^i = \sum_{i=0}^{h} 2^i \iff$$

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

$$h \leq \log_2 n < h+1$$



$\Rightarrow \lfloor \log_2 n \rfloor = h$ □

Osservazione importante:

$\forall n \geq 1 \; \exists$ un unico albero binario completo con n nodi

Heap

Definizione: min-heap (heap)

Un min-heap (o semplicemente heap) è un albero binario completo in cui ogni nodo v memorizza una entry e soddisfa la

heap-order property: *la chiave in v è minore o uguale della chiave in ciascun figlio di v*

Definizione: max-heap

Max-heap: uno heap in cui la definizione della heap-order property cambia come segue

$$\text{minore o uguale} \quad \Rightarrow \quad \text{maggiore o uguale}$$

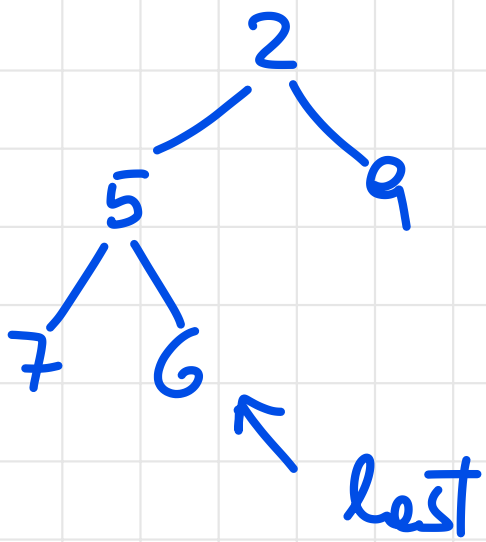
Definizione: nodo last

Il nodo last in uno heap di altezza h è il nodo più a dx del livello h.

Osservazione: uno heap è caratterizzato da 2 proprietà: (i) albero binario completo; e (ii) heap-order property (per ogni nodo!)

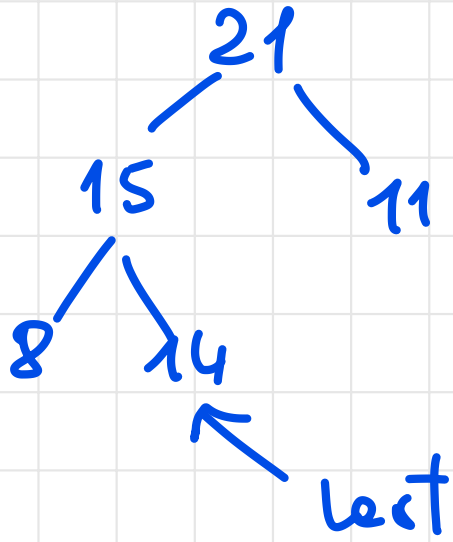
Esempio (6 chiavi)

min-heap



n = 5

max-heap



n = 5

Proprietà di uno heap

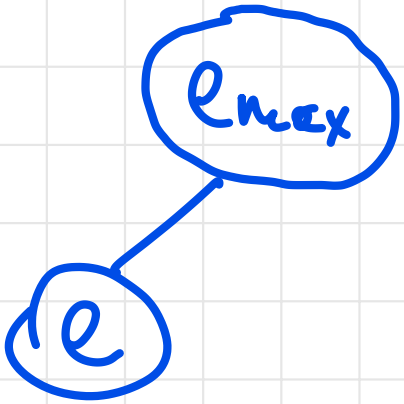
Sia P uno heap con n entry. Dalla definizione si ricavano facilmente le seguenti proprietà:

1. Le chiavi incontrate lungo un cammino dalla radice verso le foglie formano una sequenza non decrescente.
2. Per qualsiasi discendente u di un nodo $v \in P$ si ha che e_u.getKey() $\geq$ e_v.getKey().
3. La radice contiene una entry con chiave minima.
4. Se le chiavi sono tutte distinte, la entry con chiave massima ($e_{\max}$) sta in una foglia di P (es. R-9.10 in [GTG14]).

① e ② Immediate conseguenze della heap-order property

③ Corollario di ② in quanto ogni nodo è discendente della radice

④ Per assurdo se



nodo interno

$\Rightarrow$ chiave $e >$ chiave di e_{max}

che contraddice la scelta di e_{max}

Osservazione. Da queste proprietà si evince che uno heap non assicura necessariamente un ordinamento totale tra le entry, ma assicura l'ordinamento solo lungo percorsi radice-foglia

Implementazione di alberi binari su array

Level Numbering

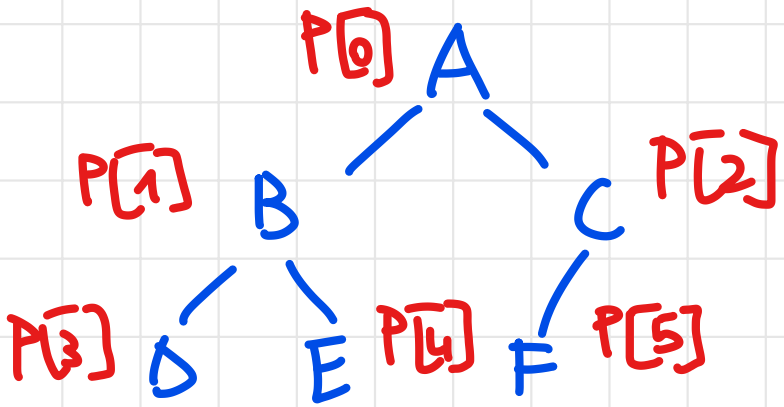
Il seguente schema (level numbering) consente di mappare un albero binario su un array $P = P[0], P[1], \ldots$:

- Radice $\rightarrow P[0]$
- Figli di $P[i] \rightarrow P[2i+1], P[2i+2]$
- Padre di $P[i] \rightarrow P\left[\lfloor (i-1)/2 \rfloor\right]$

conseguenza di

Osservazione: La rappresentazione è space-efficient per alberi molto bilanciati (ad es., per alberi binari completi) ma non lo è affatto per alberi sbilanciati, come si vede nei seguenti esempi.

Esempio 1: Albero binario completo (molto bilanciato)



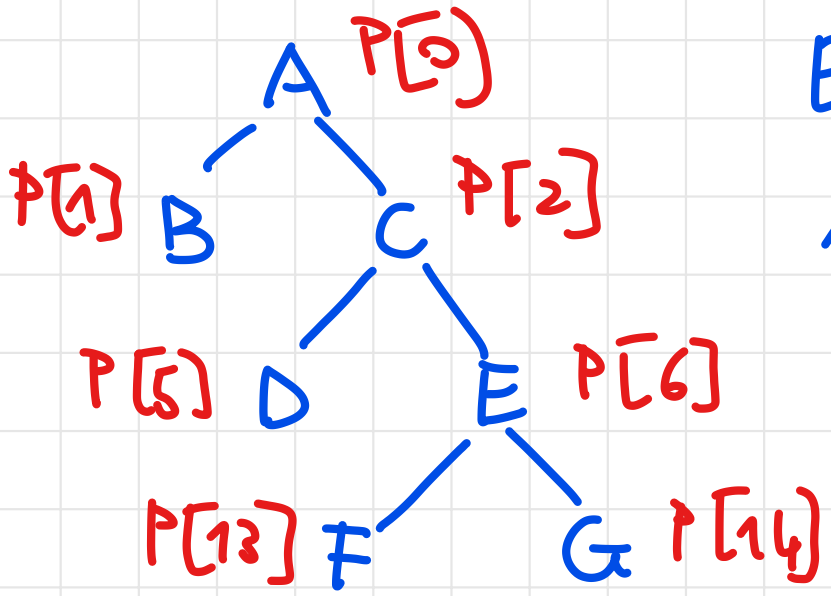
	0	1	2	3	4	5
P ≡	A	B	C	D	E	F

nod dell'albero = 6

|P| = 6

} memoria space-efficiency

Esempio 2: Albero molto sbilanciato



Esercizio: generalizzare l'esempio cattivo a n arbitrario

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P≡ A	B	C			D	E							F	G

nodi dell'albero = 7

|P| = 15

} minima space-efficiency

Implementazione di alberi binari completi su array

È facile dimostrare che usando il level numbering per mappare un albero binario completo con $n \geq 1$ nodi e altezza h su un array P, si ha che:

- Per ogni $0 \leq i < h$: i 2^i nodi del livello i, presi da sx a dx, sono mappati in $P[2^i - 1], P[2^i], \ldots, P[2^{i+1} - 2]$
- I nodi del livello h, presi da sx a dx, sono mappati in $P[2^h - 1]$, $P[2^h], \ldots, P[n - 1]$. Il nodo *last* è quindi mappato in $P[n - 1]$

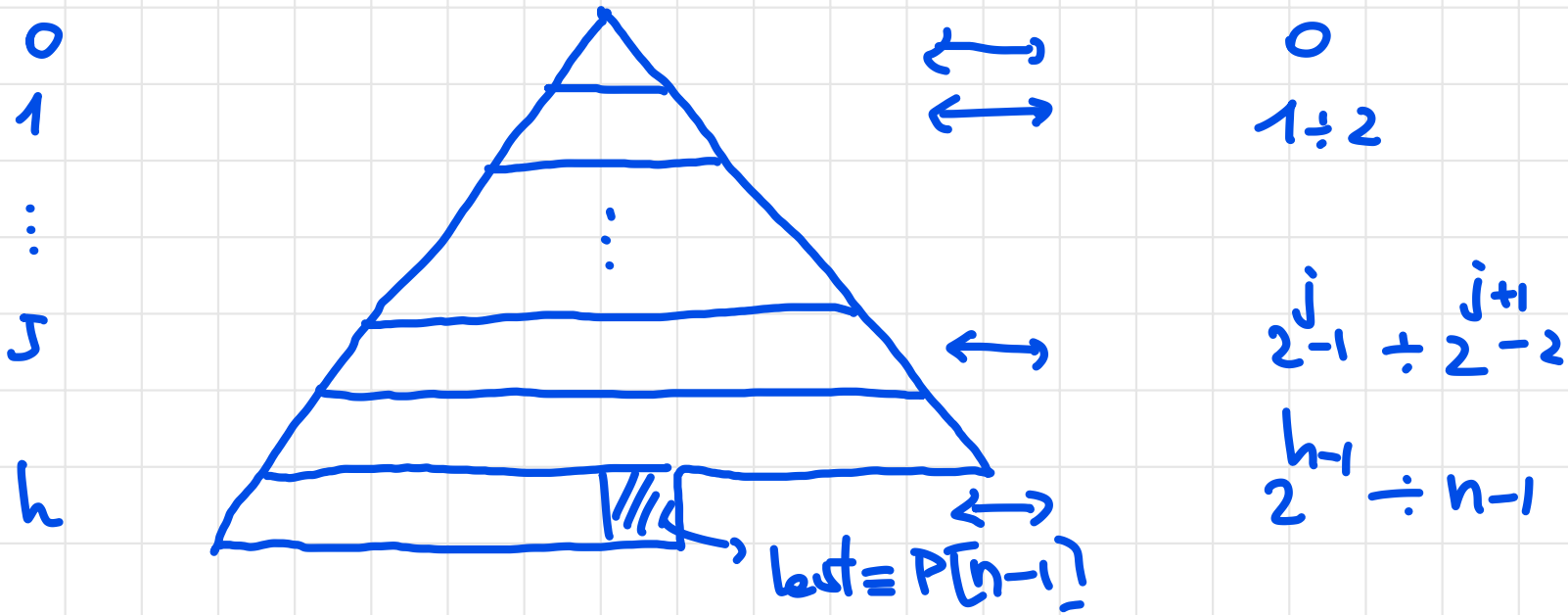
Livello	Indici
0	0
1	$1 \div 2$
2	$3 \div 6$
$\vdots$	$\vdots$
j	$2^j - 1 \div 2^{j+1} - 2$
$\vdots$	$\vdots$
$h-1$	$2^{h-1} - 1 \div 2^h - 2$
h	$2^h - 1 \div n - 1$

Oss.: il level numbering definisce quindi una corrispondenza 1-1 tra array e alberi binari completi.

Corrispondenza 1-1 tra Alberi binari completi e Array

Livello

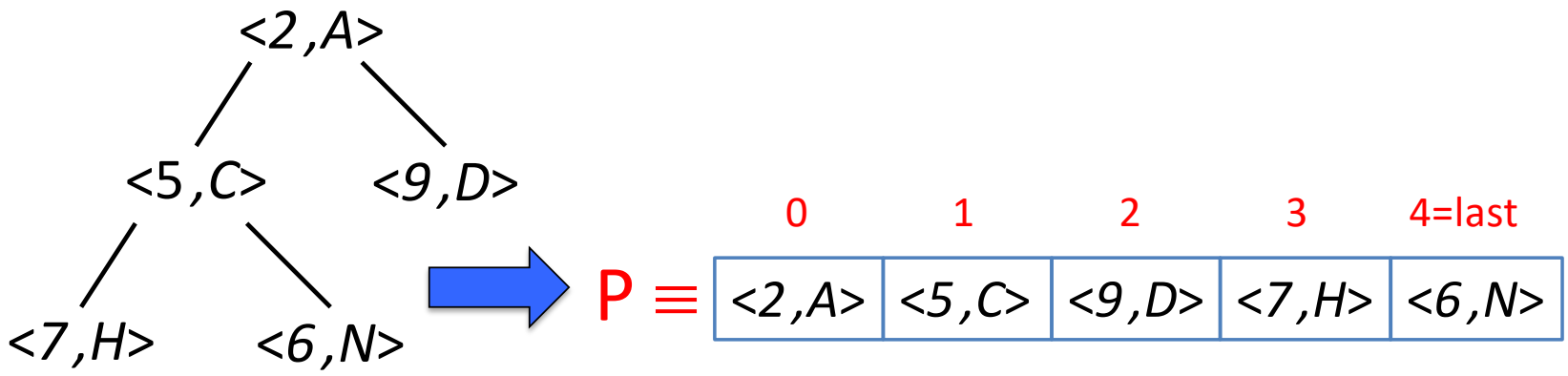
Indici su array



⇒ Corrispondenza biunivoca tra gli n nodi di un albero binario completo e le celle di un array di taglia n

Implementazione di heap su array

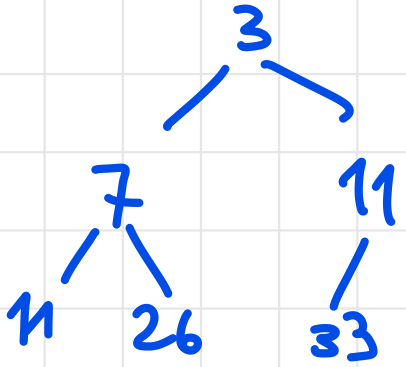
Se non diversamente specificato, assumeremo sempre che uno heap sia realizzato tramite array.



Esercizio

Sia P un array di n entry $P[0]P[1]\ldots P[n-1]$ le cui chiavi sono in ordine non decrescente. P rappresenta uno heap? Motivare la risposta.

Esempio $P \equiv [3 \quad 7 \quad 11 \quad 11 \quad 26 \quad 33] \qquad n = 6$



* Albero binario completo: Sì
* Heap order property: Sì

In generale è vero dato che se vedo l'albero binario completo che corrisponde a P le entry nel nodo associato a $P[i]$ avrà

chiave $\leq$ delle entry nei nodi associati ai figli di $P[i]$ $(P[2i+1], P[2i+2])$ grazie all'ordinamento di P

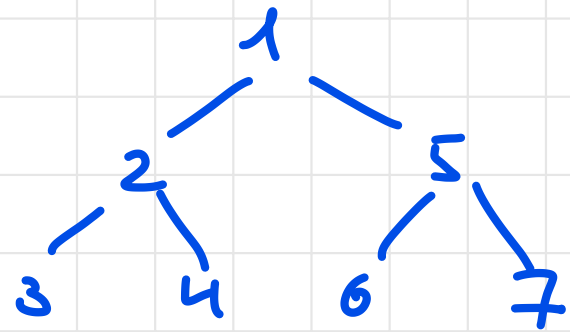
Oss. Non è vero il viceversa, cioè non è vero che uno heap è sempre associato a un array ordinato

Esercizio (modifica di R-9.15 [GTG14])

Si risponda alle seguenti domande:

1. Determinare uno heap P con 7 entry con chiavi 1,2,...,7 in cui la visita in preorder tocca le entry in ordine crescente di chiave.
2. Determinare uno heap P con 7 entry con chiavi 1,2,...,7 in cui la visita in preorder non tocca le entry in ordine crescente di chiave.
3. Dimostrare che la visita inorder e in postorder di uno heap con $n > 1$ entry, nonpuò mai toccare le entry in ordine crescente di chiave.

↳ con chiavi distinte

① 

② Basta scambiare 5 e 3

③ Perché la radice contiene la chiave minima e non è ristretta per prima

Implementazione dei metodi della P.Q. su heap

Notazione (per pseudocodice)

$P[i] \equiv$ *entry*:

- chiave: $P[i]$.getKey()
- valore: $P[i]$.getValue()

$P[last] \equiv$ entry più a dx al livello h

Sia P uno heap con n entry implementato tramite array,

- **Metodo** `min()`: **return** $P[0]$ ($\Rightarrow$ Complessità $\Theta(1)$)
- **Metodo** `insert(`k`,` x`)` ??
- **Metodo** `removeMin()` ??

Inserimento: `insert`

IDEA

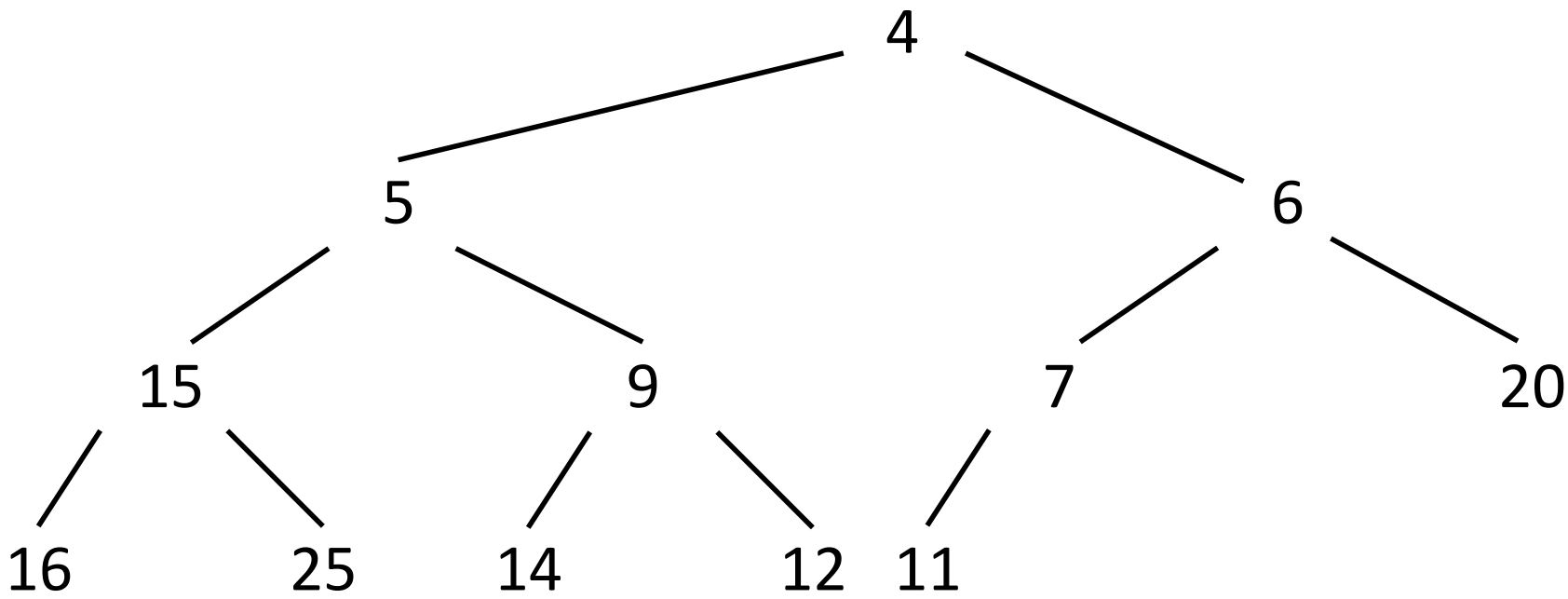
- inserisci la nuova entry come successore (nel level numbering) del nodo last
- ricostruisci la heap-order property dello heap lungo il cammino dal nodo *last* alla radice

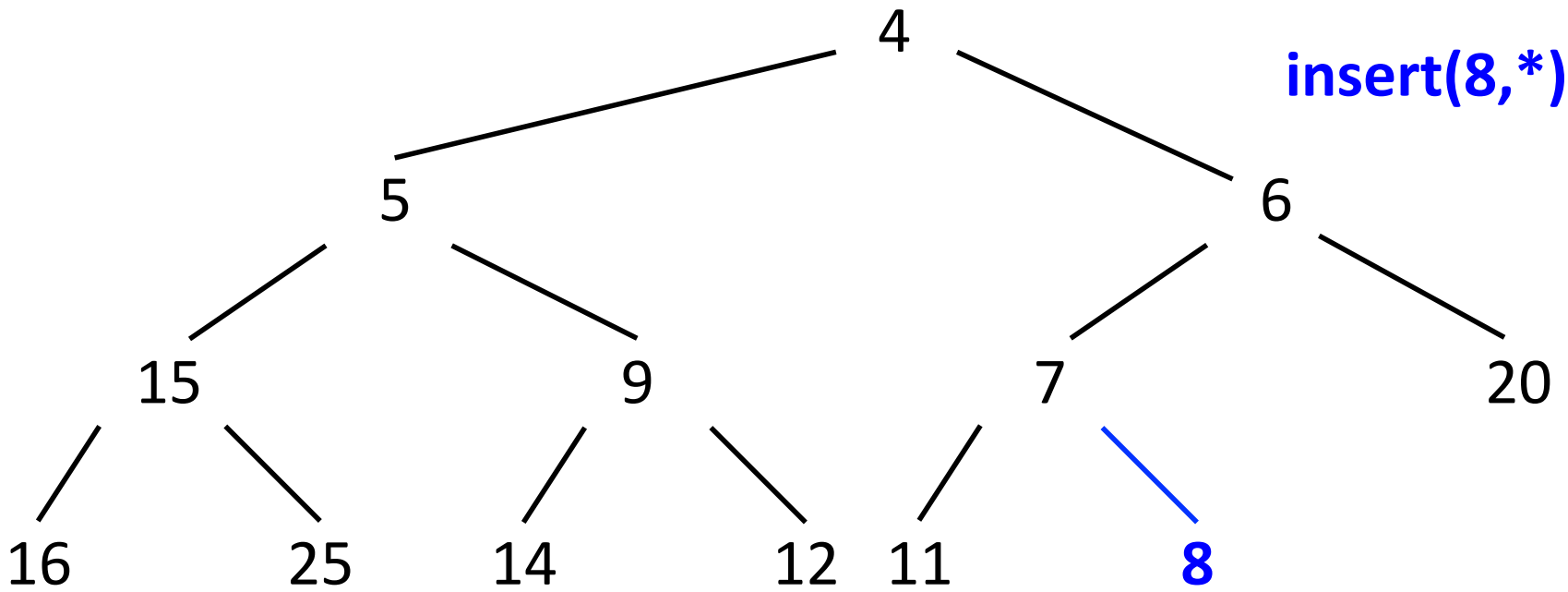
Metodo `insert(`k`,`x`)`

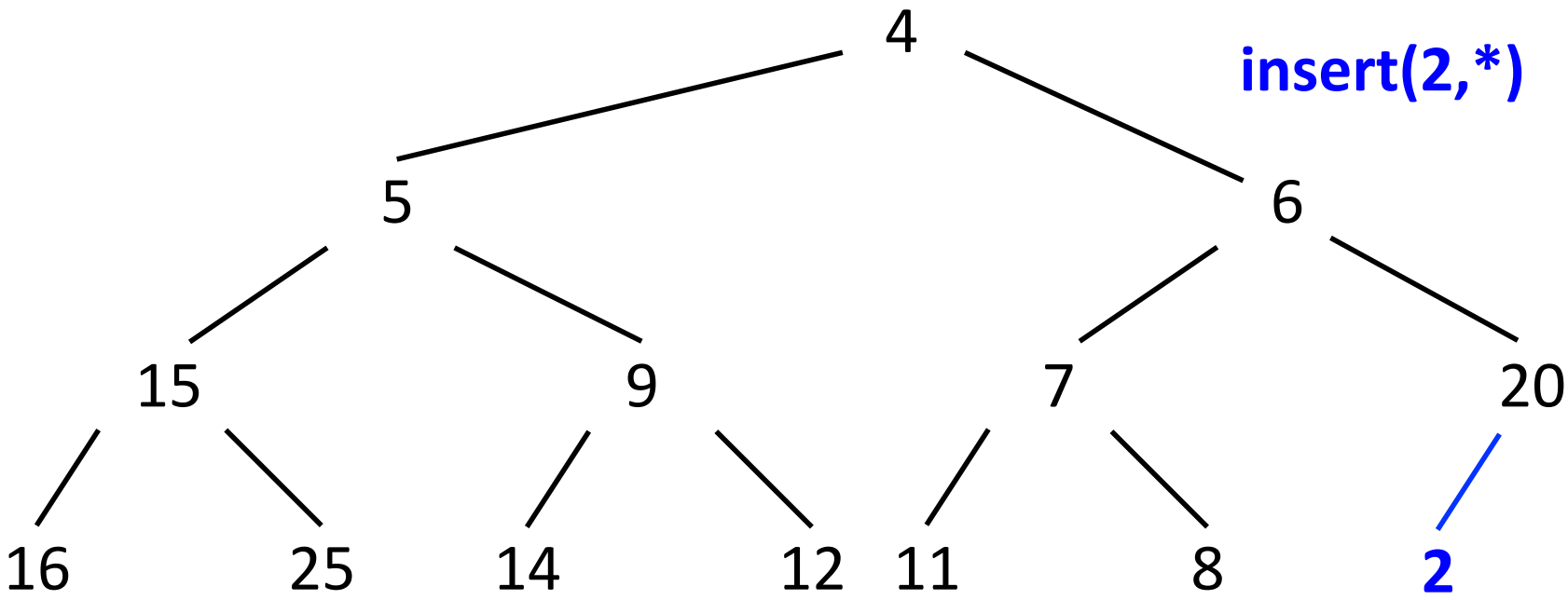
$e \leftarrow (k,x)$;
$P[++\text{last}] \leftarrow e$;
$i \leftarrow \text{last}$;
// *Up-heap bubbling*
while $((i > 0)$ `AND` $(P[\lfloor \frac{i-1}{2} \rfloor]$`.getKey()`$> P[i]$`.getKey()))` **do**
    `swap(` $P[i]$, $P[\lfloor \frac{i-1}{2} \rfloor]$`);`
    $i \leftarrow \lfloor \frac{i-1}{2} \rfloor$;
return e;

Oss. In caso di overflow, si devono prima trasferire le entry in un array più capiente (di solito di taglia doppia di quello corrente)

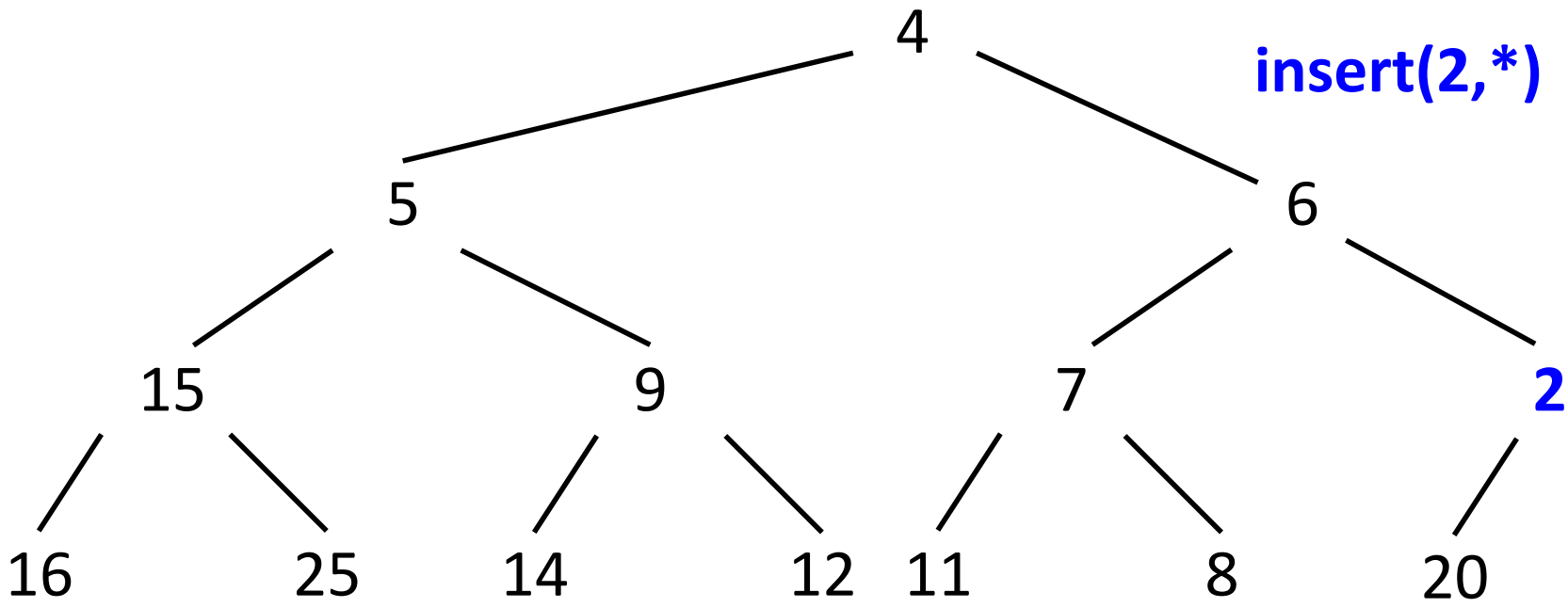
Esempio (solo chiavi)



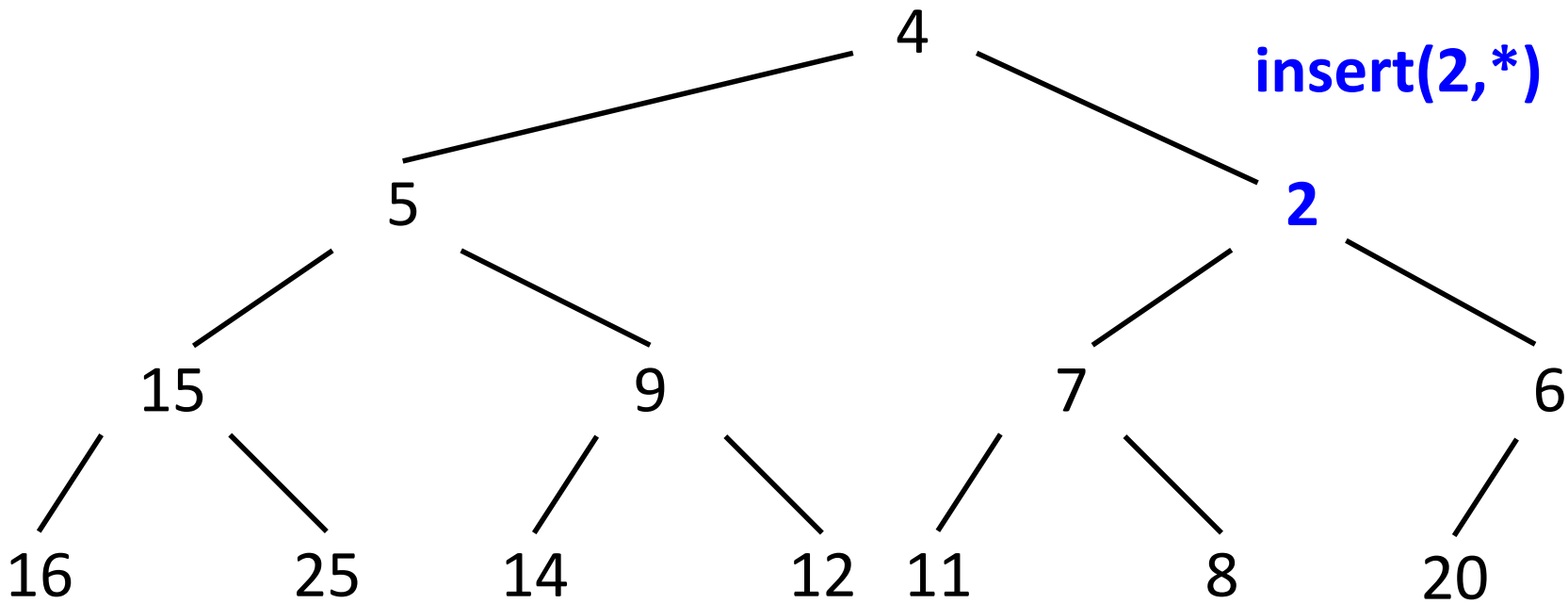

4
insert(8,*)
5
6
15
9
7
20
16
25
14
12
11
8


4
insert(2,*)
5
6
15
9
7
20
16
25
14
12
11
8
2

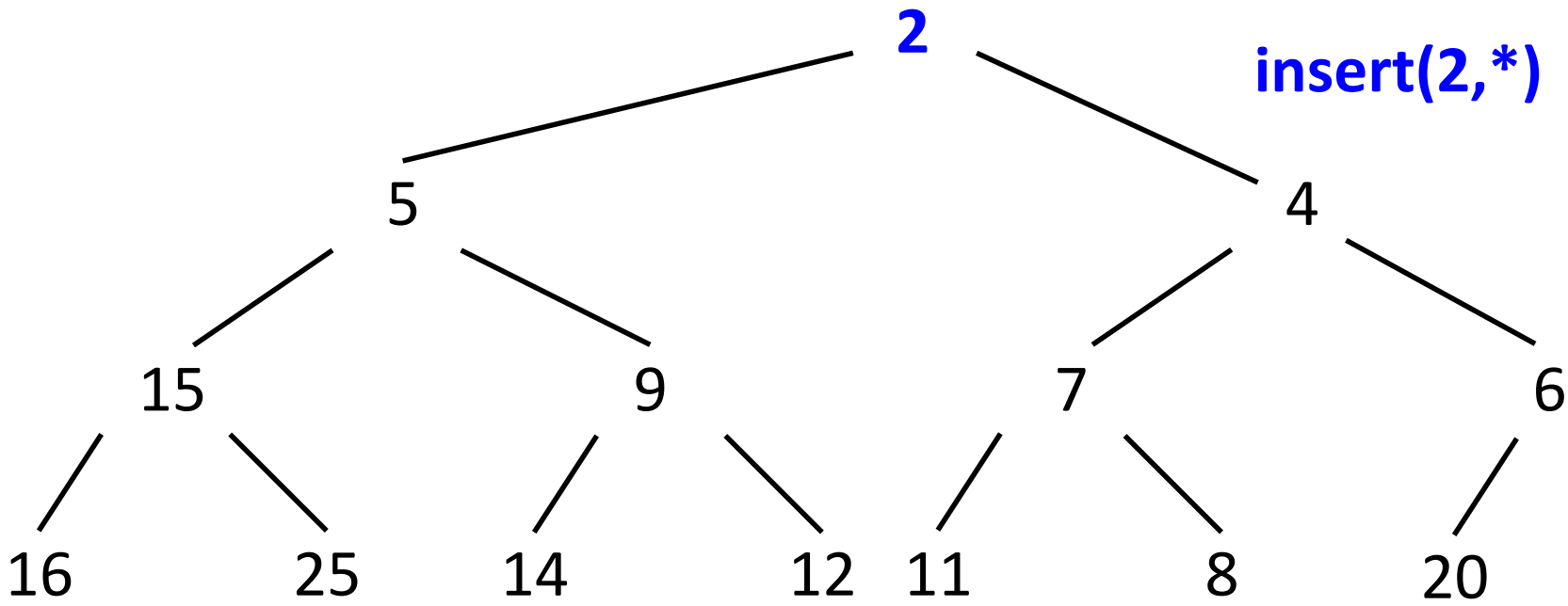
Dopo lo swap tra 2 e 20



Pop swap the 2 & 6



Dopo lo swap tra 2 e 4



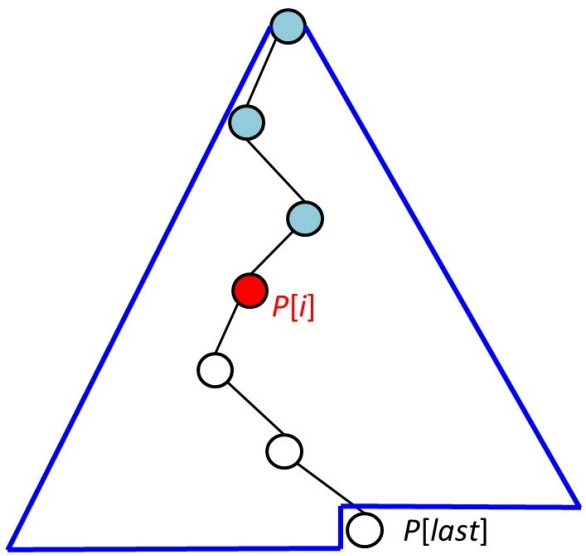
Fine di insert(2,*)

Correttezza di `insert`

Heap-order property estesa: *in uno heap, la chiave in un nodo deve essere maggiore o uguale di quella in qualsiasi suo antenato*.

La correttezza di `insert` discende delle seguenti proprietà:

- Non viene mai violata la struttura di albero binario completo.
- Il ciclo while mantiene il seguente invariante (dimostrarlo per esercizio): le uniche violazioni della heap-order property estesa possono essere tra $P[i]$ e un suo antenato.



Complessità di `insert`

Consideriamo l'esecuzione di insert su uno heap P con n entry, e sia $h = \lfloor \log_2(n+1) \rfloor$ l'altezza risultante dopo l'inserimento.

- Complessità proporzionale al numero di iterazioni del while, dato che ogni iterazione richiede $\Theta(1)$ operazioni
- Numero di iterazioni del while $\leq h$
- Esiste un'istanza che richiede esattamente h iterazioni (si inserisce una entry con chiave minore di tutte quelle presenti)

$\Rightarrow$ Complessità di `insert`: $\Theta(\log n)$

Osservazione

La complessità non tiene conto del costo del trasferimento delle entry in un array più grande nel caso in cui l'array si riempia (*overflow*). Tuttavia, è facile vedere che raddoppiando la taglia dell'array ogni volta che si presenta un overflow, il costo di ciascun overflow non supera asintoticamente il costo aggregato delle precedenti invocazioni di `insert`, e quindi può essere nascosto (*ammortizzato*) da quest'ultimo.

Rimozione: `removeMin`

IDEA

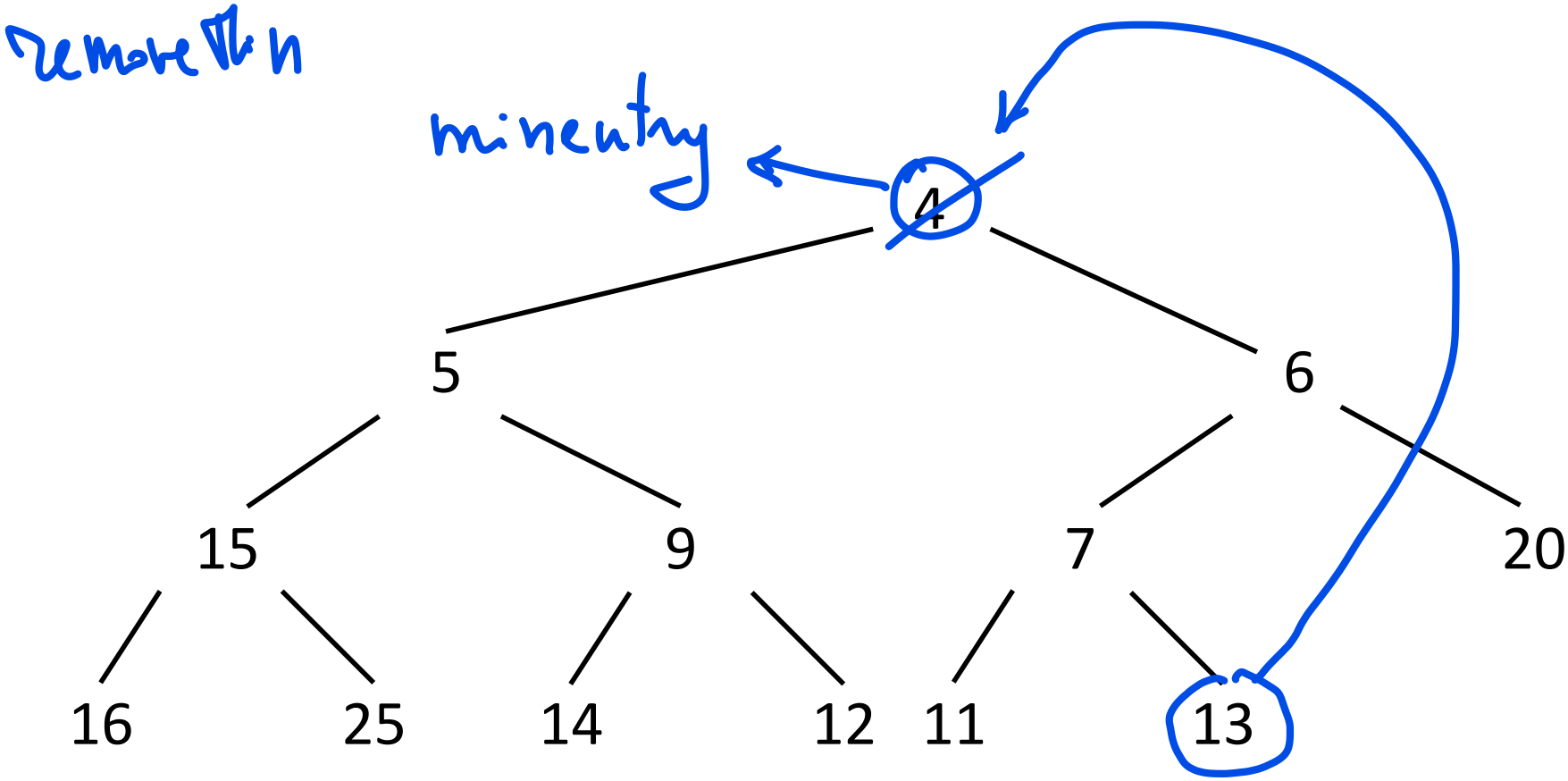
- rimuovi la entry presente nella radice dello heap
- metti la entry $P[\mathrm{last}]$ nella radice
- ricostruisci la heap-order property dello heap a partire dalla radice verso le foglie

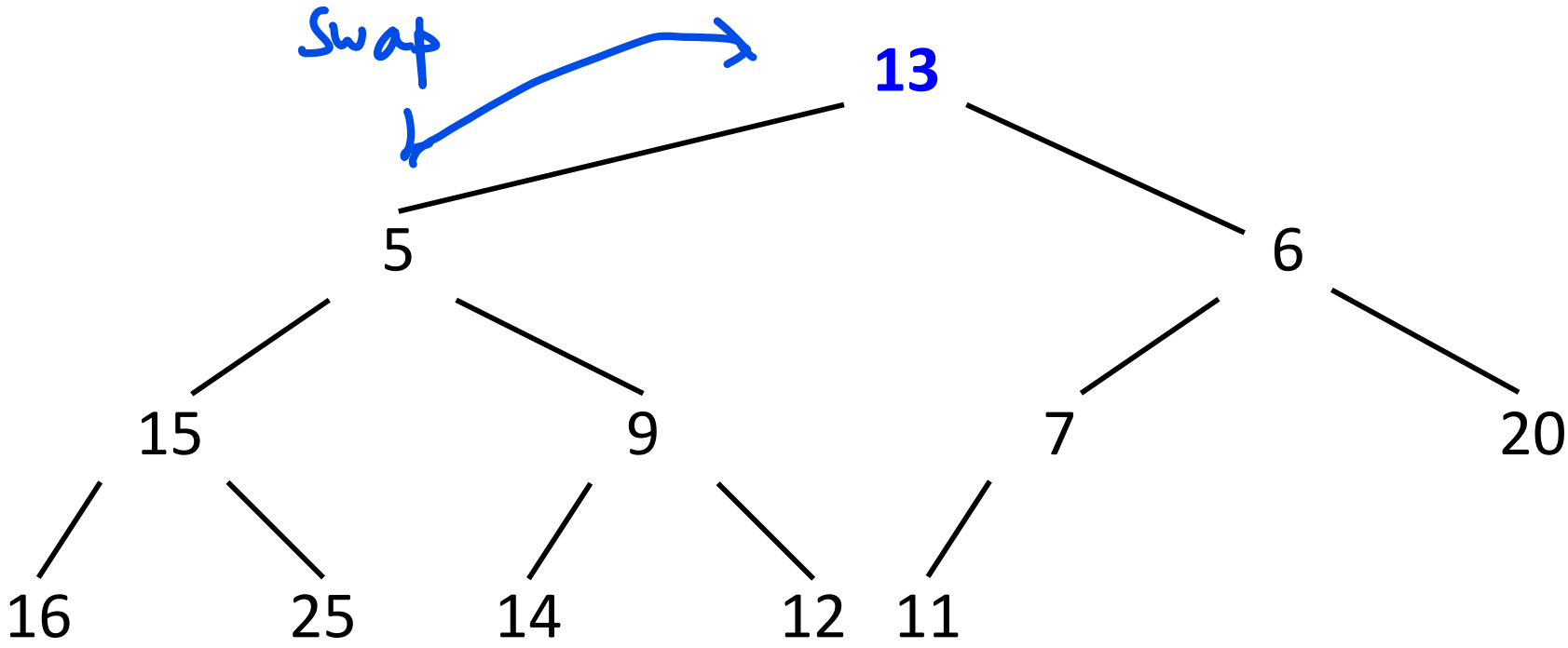
Sia `indexMinChild`(P, i) un metodo che restituisce l'indice del figlio di $P[i]$ con chiave minima ($2i+1$ o $2i+2$), se $P[i]$ è un nodo iterno (cioè $2i+1 \leq \text{last}$), e restituisce `null`, se $P[i]$ è foglia.

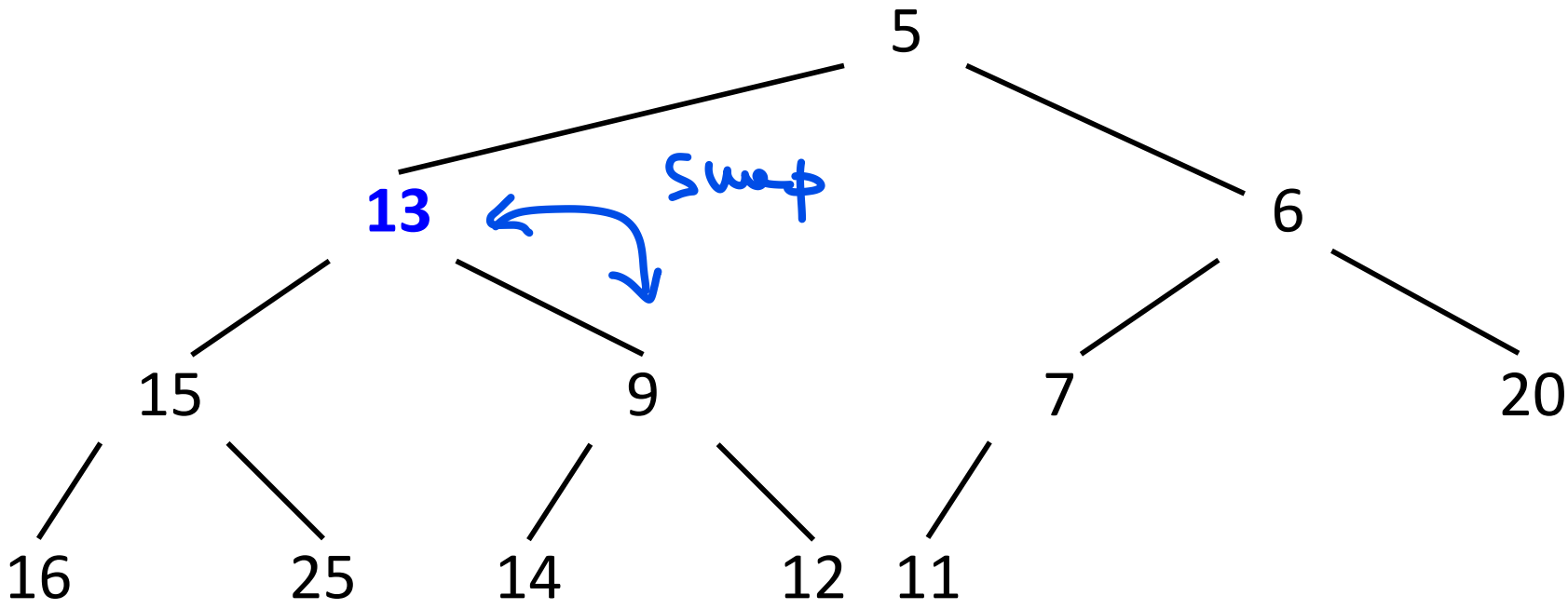
Metodo `removeMin()`

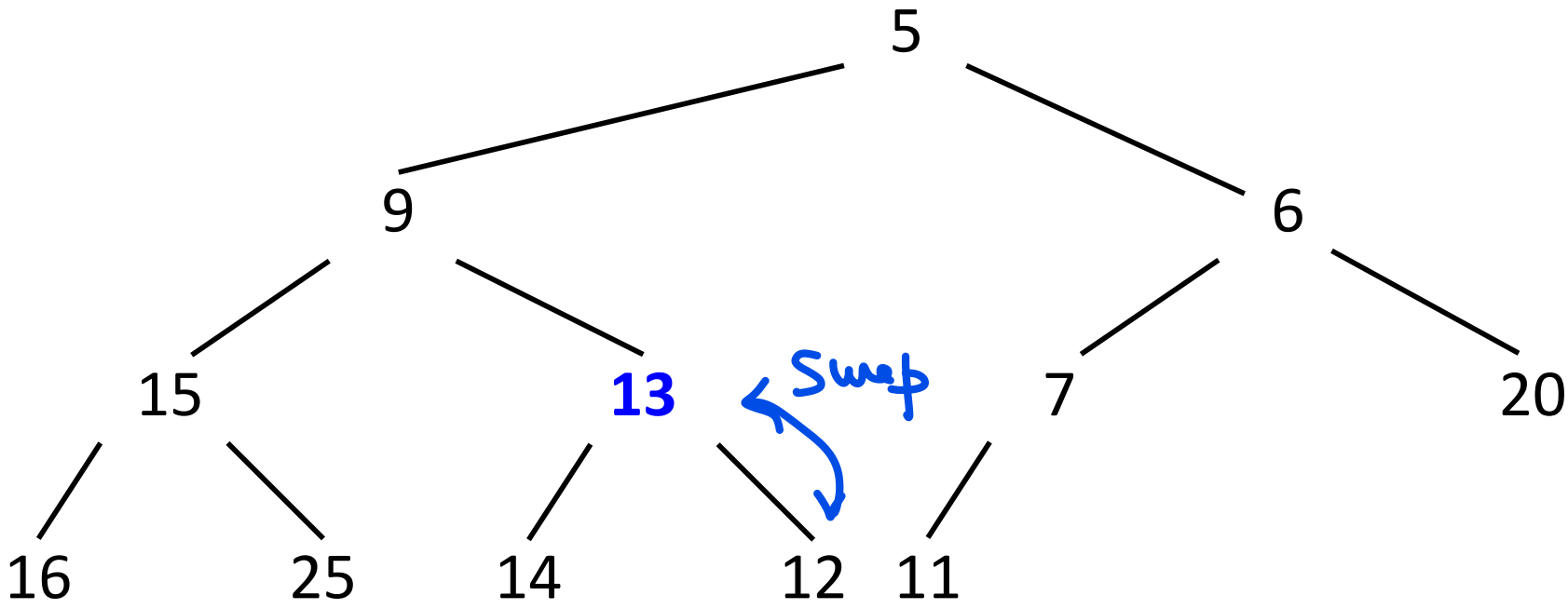
$minentry \leftarrow P[0]$;
$P[0] \leftarrow P[\texttt{last--}]$;
$i \leftarrow 0$;
$j \leftarrow$ indexMinChild(P, i);
// *Down-heap bubbling*
while $((j \neq \texttt{null})$ `AND` $(P[i].\texttt{getKey}() > P[j].\texttt{getKey}()))$ **do**
    swap($P[i]$,$P[j]$);
    $i \leftarrow j$;
    $j \leftarrow$ indexMinChild(P, i);
return *minentry*

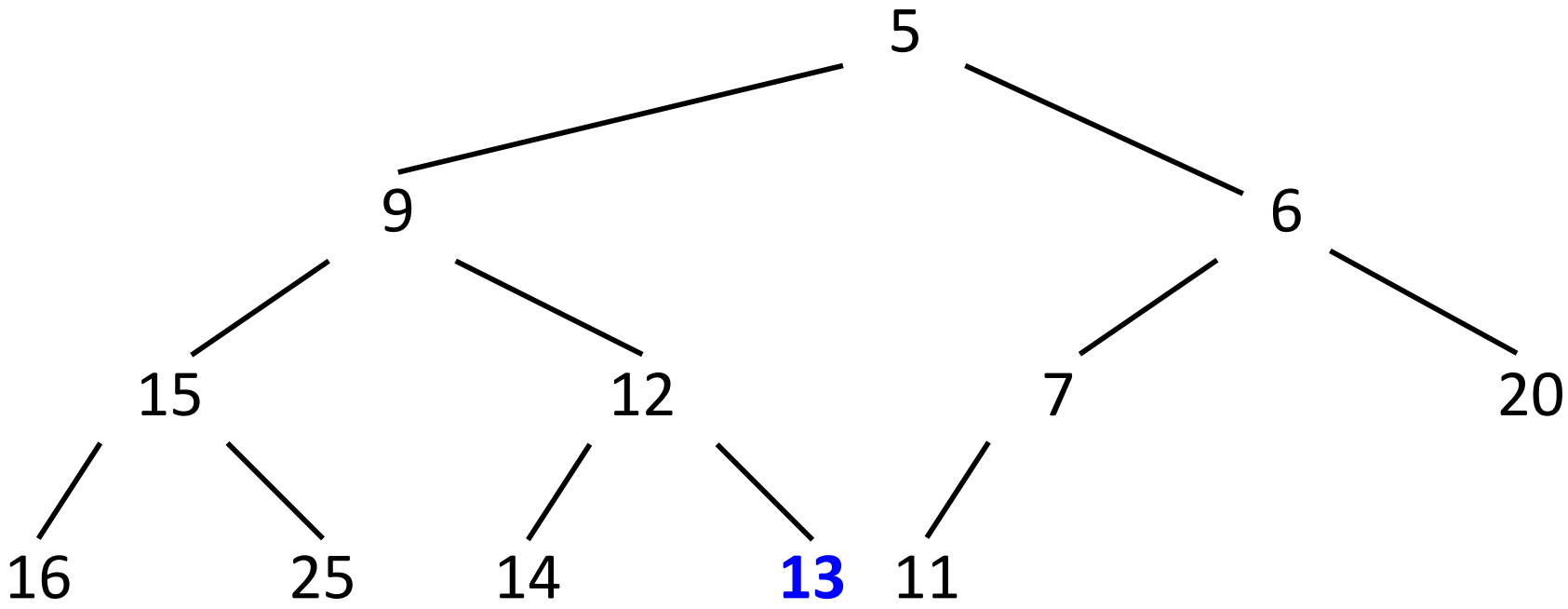
Esempio (solo chiavi)




Swap
13
5
6
15
9
7
20
16
25
14
12
11


5
13
Swap
6
15
9
7
20
16
25
14
12
11


5
9
6
15
13
Swap
7
20
16
25
14
12
11



Fine d remove Min

Correttezza di `removeMin`

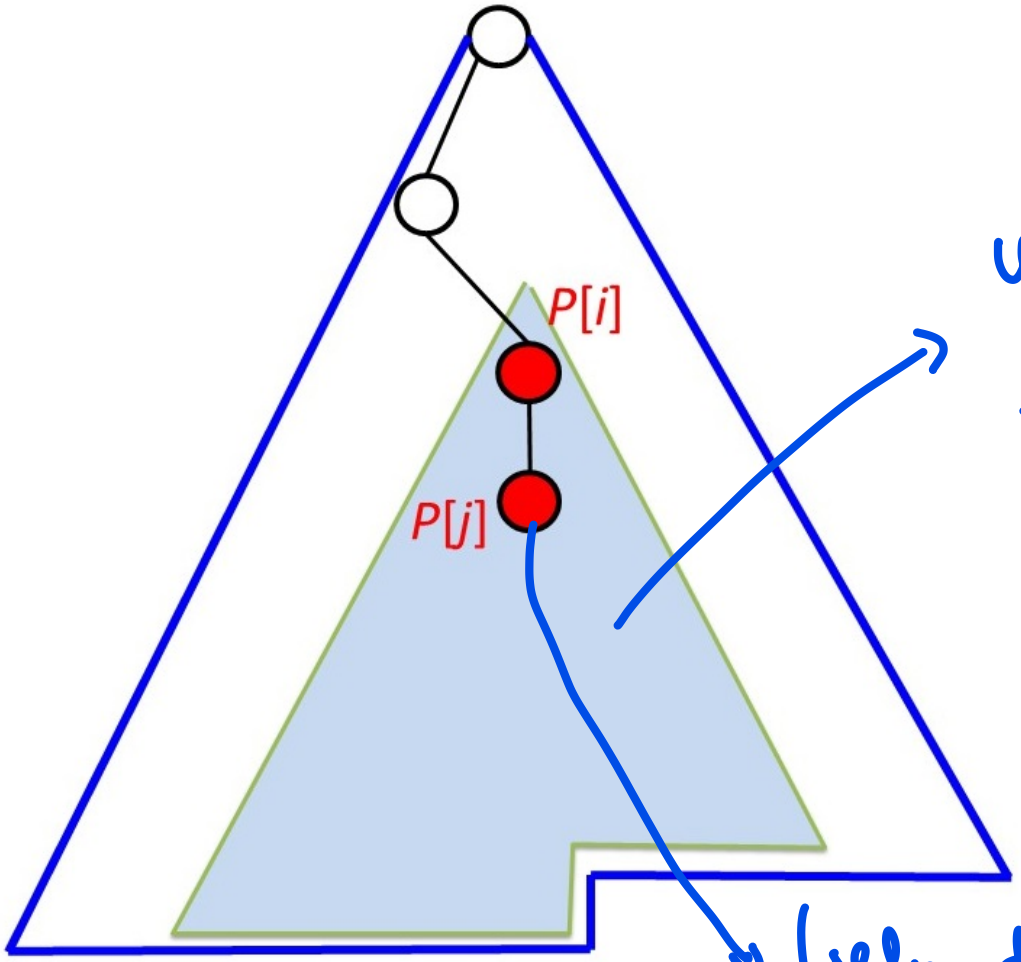
La correttezza discende dal seguente invariante per il while

Invariante

- $P[j]$ figlio di $P[i]$ con chiave minima (se $P[i]$ è interno)
- Le uniche coppie antenato-discendente che possono violare la heap-order property estesa sono coppie in cui l'antenato è $P[i]$.

Esercizio

Dimostrare che l'invariante vale all'inizio del while, e alla fine di ciascuna sua iterazione.



una delle violazioni della HOP estesa (se possibile) solo tra P[i] e i suoi discendenti

figlio di P[i] con chiave minima

Complessità di `removeMin`

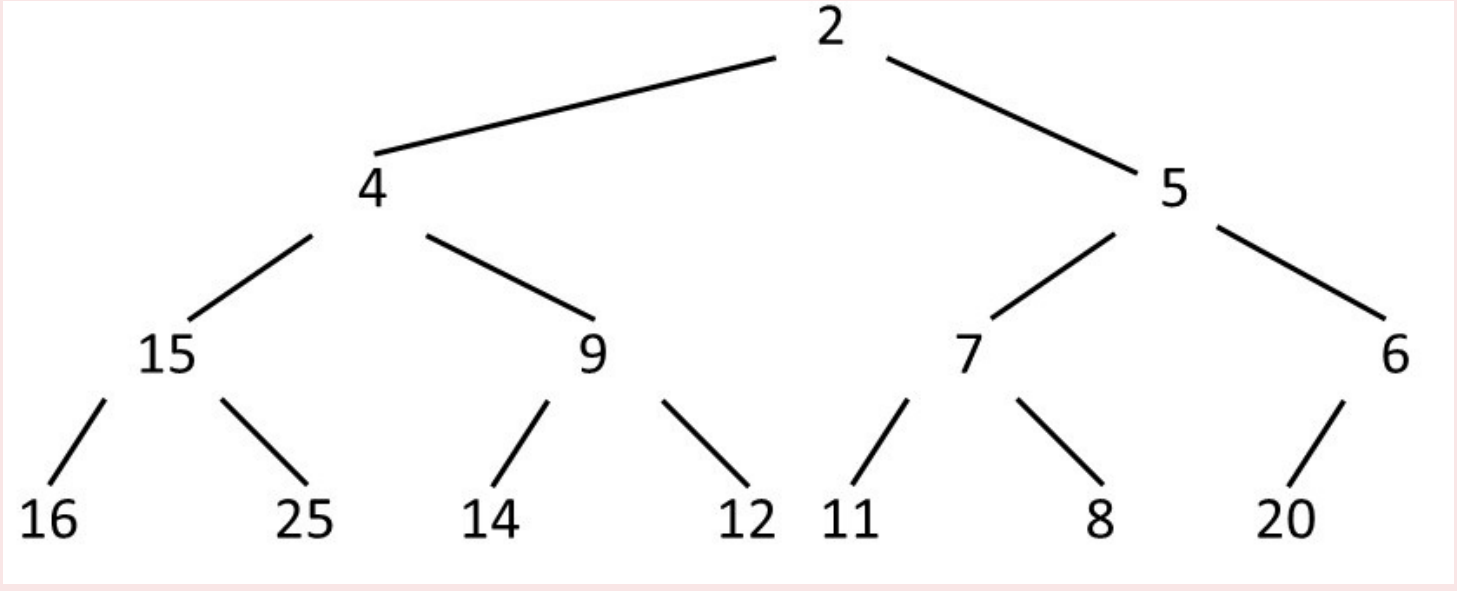
Consideriamo l'esecuzione di removeMin su uno heap P con n entry, e sia $h = \lfloor \log_2(n-1) \rfloor$ l'altezza risultante dopo la rimozione.

- Complessità proporzionale al numero di iterazioni del while, dato che ogni iterazione richiede $\Theta(1)$ operazioni
- Numero di iterazioni del while $\leq h$
- Esiste un'istanza che richiede esattamente h iterazioni (la entry in $P[\text{last}]$ ha chiave maggiore di tutte quelle presenti)

$\Rightarrow$ Complessità di `removeMin`: $\Theta(\log n)$

Esercizio

Si consideri il seguente Heap



Far vedere lo Heap risultate dopo l'esecuzione delle seguenti operazioni: `removeMin`, `removeMin`, `insert`$(3, *)$, `insert`$(4, *)$.

Esercizio

Progettare e analizzare un algoritmo per rimuovere da uno heap P con n entry una entry $P[\ell]$, per un qualche $0 \leq \ell < n$, ripristinando poi le proprietà dello heap.

Esercizio (C-9.34 [GTG14])

Progettare un algoritmo che dato uno heap T con n entry e una chiave k, stampi tutte le entry in T con chiave $\leq k$. La complessità deve essere proporzionale al numero di entry stampate ($+1$ nel caso siano 0).

Riepilogo sulle implementazioni della Priority Queue

$n = \#$ entry

STRUTTURA	COMPLESSITÀ		
	min	insert	removeMin
LISTA NON ORDINATA	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
LISTA ORDINATA	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
HEAP	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

→ implementazione efficiente in spazio su array

Esercizio

Analizzare la complessità dell'algoritmo $\texttt{Top}(S,\tau)$ sviluppato nell'esercizio dei Lucidi 7 e 8.

Esercizio (C-9.36 [GTG14])

Siano T_1 e T_2 due alberi binari (non necessariamente completi e implementati tramite struttura linkata) i cui nodi memorizzano delle entry in modo da soddisfare la heap-order property. Descrivere un algoritmo per combinare T_1 e T_2 in un unico albero binario T i cui nodi contengano tutte le entry di T_1 e T_2, mantenendo la heap-order property. La complessità dell'algoritmo deve essere $O(h_1 + h_2)$, dove h_1 e h_2 rappresentano le altezze di T_1 e T_2, rispettivamente.

Costruzione di uno heap a partire da n entry date

La specifica input-output per il problema è la seguente:

Input. Array P con n entry $P[0 \div n-1]$

Output. Array P riorganizzato per rappresentare uno heap.

Definizione: Algoritmo IN-PLACE

Un algoritmo si dice IN-PLACE se usa $O(1)$ memoria aggiuntiva oltre a quella necessaria per l'input.

SOLUZIONI BANALI

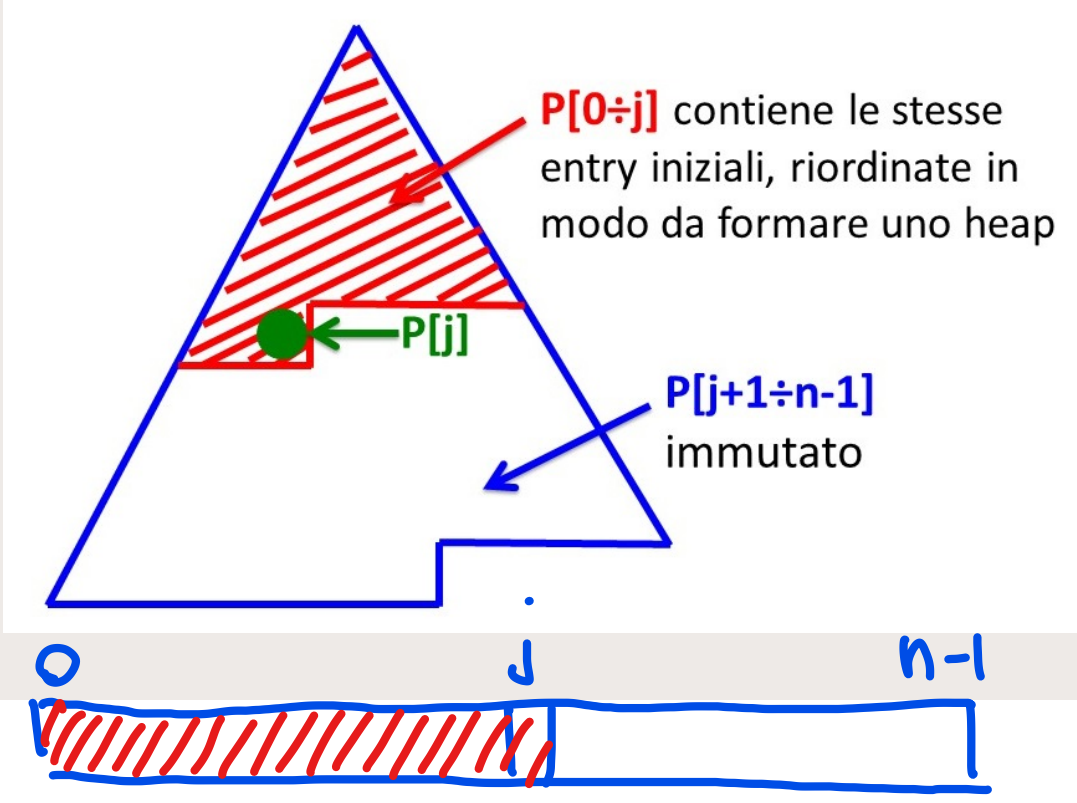
(A) Ordinare P in senso non decrescente usando insertionSort (tempo $\Theta(n^2)$, IN-PLACE) o mergeSort (tempo $\Theta(n \log n)$, non IN-PLACE)

(B) Trasferisco le entry da P a un array d appoggio Q e poi le rimetto in P invocando n volte insert (tempo $\Theta(n \log n)$, la vedremo ma non è IN-PLACE)

Soluzione 1 (approccio top-down)

Idea: Si eseguono $n-1$ iterazioni successive mantenendo il seguente invariante alla fine di ciascuna iterazione j, con $1 \leq j \leq n-1$

Invariante (alla fine della iterazione j)



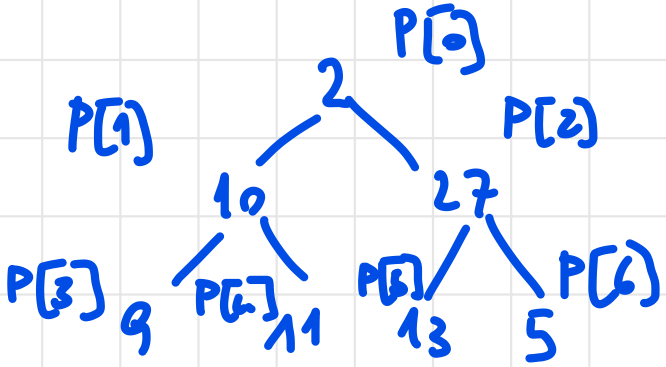
Soluzione 1 (approccio top-down)

Implementazione: up-heap bubbling da $P[j]$ in ciascuna iterazione j

```
last ← n − 1;
for j ← 1 to n − 1 do
    // Up-heap bubbling a partire da P[j]
    i ← j;
    while ((i > 0)AND(P[⌊(i−1)/2⌋].getKey() > P[i].getKey())) do
        swap(P[i], P[⌊(i−1)/2⌋]);
        i ← ⌊(i−1)/2⌋;
```

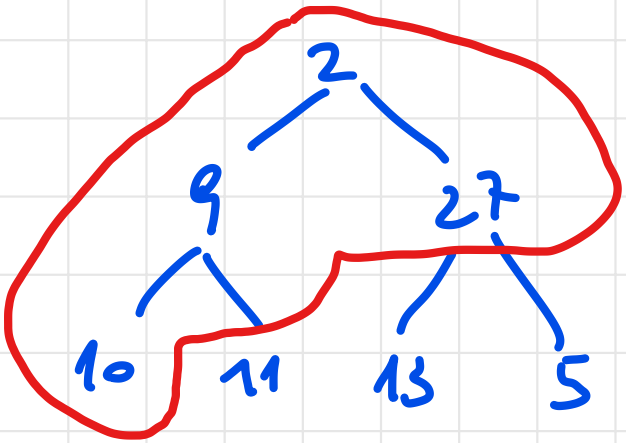
CORRETTEZZA: discende immediatamente dall'invariante riportato nella slide precedente. Si lascia come facile esercizio la dimostrazione che l'invariante vale alla fine di ciascuna iterazione.

Esempio

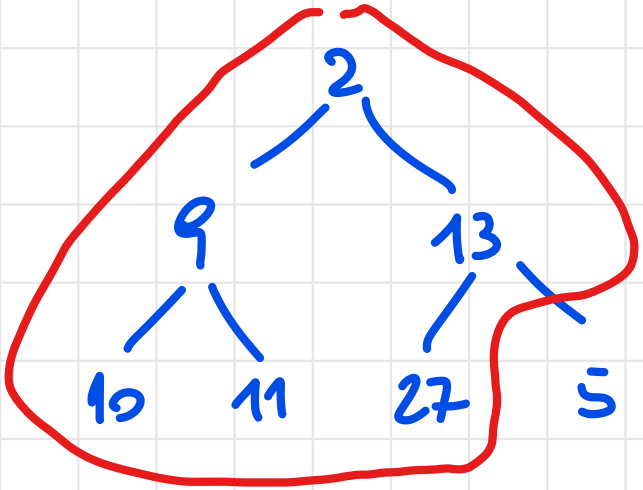


$P \equiv [2\ 10\ 27\ 9\ 11\ 13\ 5]$

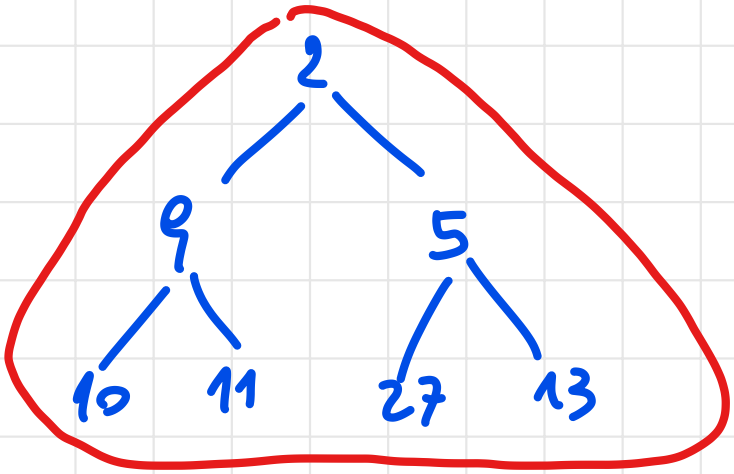
Dopo iterazione $j = 3$



Dopo l'iterazione j = 5



Dopo l'iterazione j = 6



Complessità dell'approccio top-down

Dimostriamo che la complessità è $\Theta\left(\sum_{j=1}^{n-1} \log j\right)$:

- $O\left(\sum_{j=1}^{n-1} \log j\right)$: banale, dato che l'iterazione j equivale a inserire $P[j]$ in $P[0 \div j-1]$.
- $\Omega\left(\sum_{j=1}^{n-1} \log j\right)$: considerando come istanza "cattiva" quella in cui P è inizialmente ordinato in senso decrescente

Nei lucidi seguenti dimostreremo che

$$\sum_{j=1}^{n-1} \log j \in \Theta(n \log n).$$

Osservazione: la base (non indicata) dei logaritmi è 2, ma comunque la prova vale per qualsiasi base costante.

Claim

$\sum_{j=1}^{n-1} \log j \in \Theta(n \log n)$ (basi costanti per i logaritmi)

* $\sum_{j=1}^{n-1} \lg j < \sum_{j=1}^{n-1} \lg n = (n-1)\lg n < n \lg n$

$\Rightarrow \sum_{j=1}^{n-1} \lg j \in O(n \lg n)$

* $\sum_{j=1}^{n-1} \lg j \geq \sum_{j=\lfloor n/2 \rfloor}^{n-1} \lg j \geq \sum_{j=\lfloor n/2 \rfloor}^{n-1} \lg \lfloor n/2 \rfloor$

$\geq \lfloor n/2 \rfloor \cdot \lg \lfloor n/2 \rfloor$

$\Rightarrow \sum_{j=1}^{n-1} \lg j \in \Omega(n \lg n)$ □

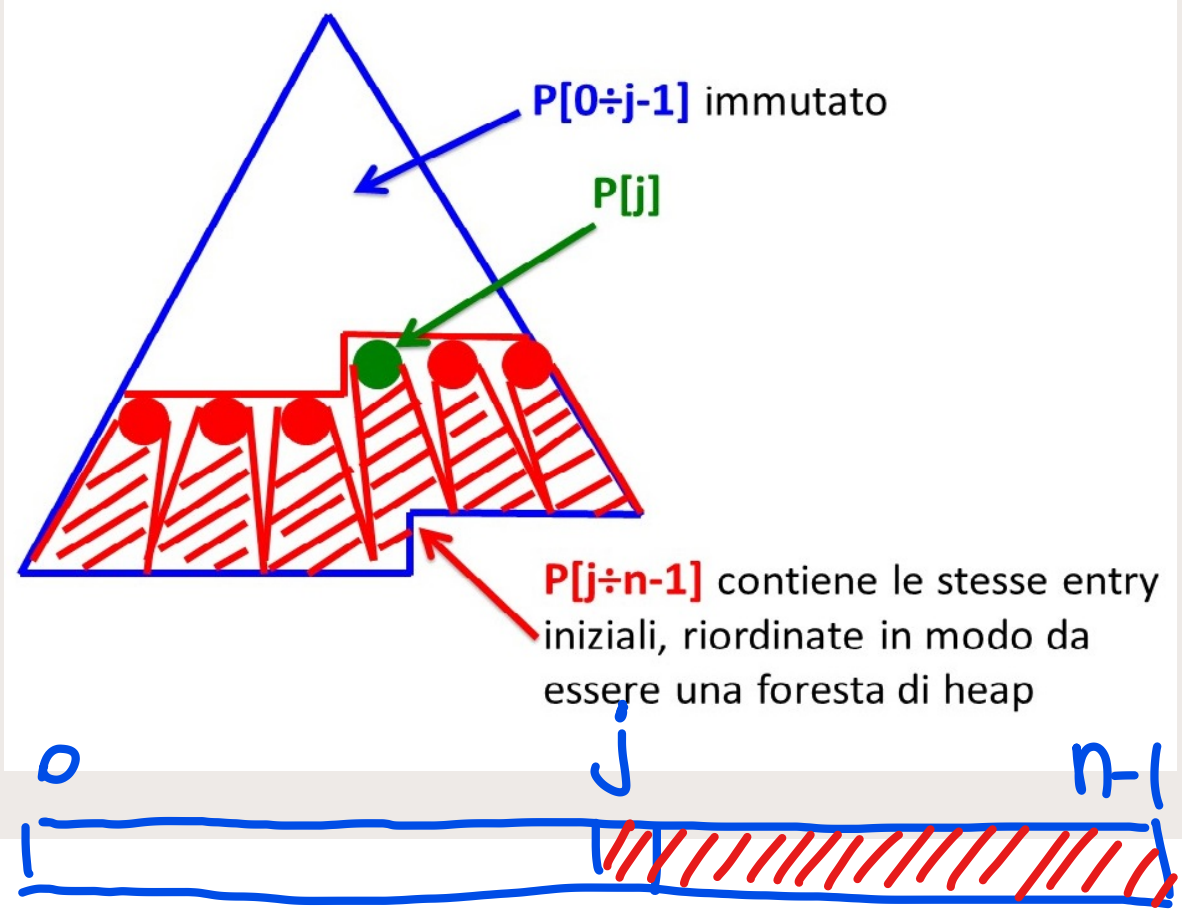
L'analisi mostra che la complessità della costruzione top-down di un heap a partire da un array di n entry è $\Theta(n \log n)$

Inoltre dimostra che in generale la complessità richiesta dalla invocazione di n insert in un heap inizialmente vuoto è $\Theta(n \log n)$ (SOLUZIONE BANALE B)

Soluzione 2 (approccio bottom-up)

Idea: Si eseguono $\lfloor n/2 \rfloor$ iterazioni successive manenendo il seguente invariante alla fine di ciascuna iterazione j, con $\lfloor (n-2)/2 \rfloor \geq j \geq 0$.

Invariante (alla fine della iterazione j)



Soluzione 2 (approccio bottom-up)

Implementazione: down-heap bubbling da $P[j]$ in ciascuna iterazione j

N.B.: $P[\lfloor (n-2)/2 \rfloor]$ = nodo interno più a dx del penultimo livello.

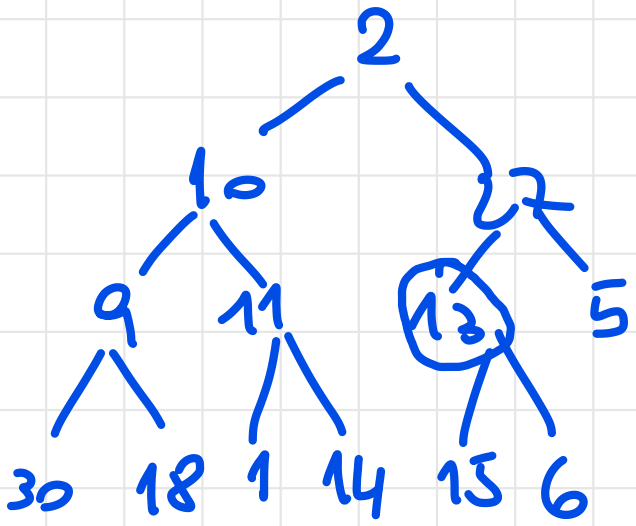
padre di last last $\equiv n-1$ padre di last $\left\lfloor \frac{n-1-1}{2} \right\rfloor = \left\lfloor \frac{n-2}{2} \right\rfloor$

```
last ← n − 1;
for j ← ⌊(n − 2)/2⌋ downto 0 do
    // Down-heap bubbling a partire da P[j]
    i ← j;
    k ← indexMinChild(P, i);
    while ((k ≠ null)AND(P[i].getKey()> P[k].getKey())) do
        swap(P[i],P[k]);
        i ← k;
        k ← indexMinChild(P, i)
```

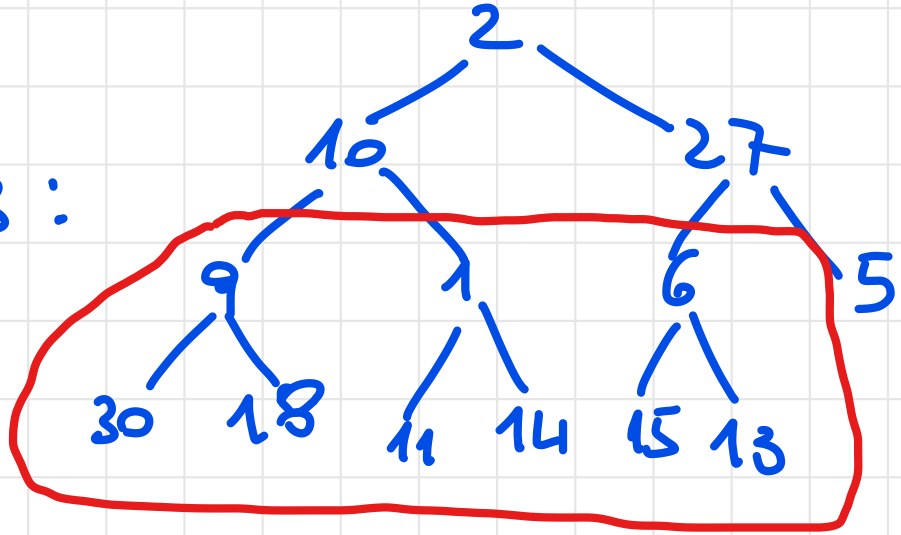
CORRETTEZZA: discende immediatamente dall'invariante riportato nella slide precedente. Si lascia come facile esercizio la dimostrazione che l'invariante vale alla fine di ciascuna iterazione.

Esempio: $P = [2\ 10\ 27\ 9\ 11\ 13\ 5\ 30\ 18\ 1\ 14\ 15\ 6]$

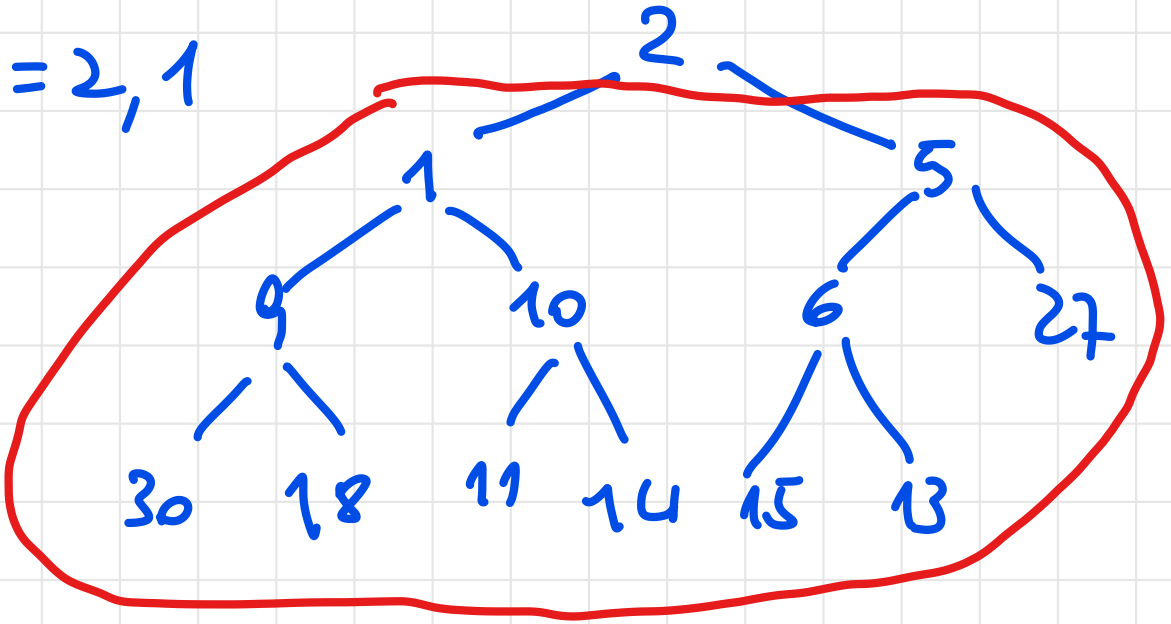
$h = 13 \qquad \lfloor (h-2)/2 \rfloor = 5$



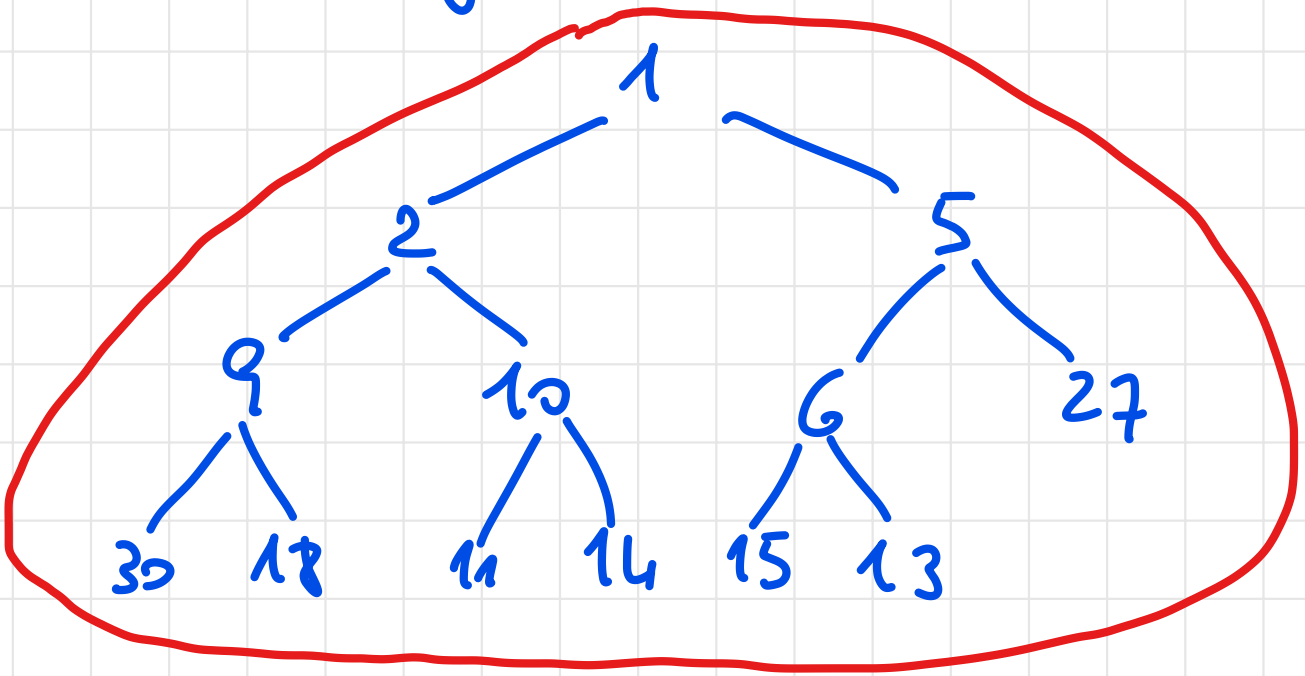
Dopo le iterazioni $j = 5, 4, 3$:



Dopo le iterazioni $j = 2, 1$



Dopo l'iterazione $j = 0$:



Complessità dell'approccio bottom-up

- $t_{P[j]} \equiv$ costo del down-heap bubbling a partire da $P[j]$
- per ogni nodo al livello i, $0 \leq i \leq h-1$ ($h = \lfloor \log_2 n \rfloor$) il down-heap bubbling costa $O(h-i)$
- 2^i nodi al livello i

La complessità è quindi:

$$O\left(\sum_{j=0}^{\lfloor (n-2)/2 \rfloor} t_{P[j]}\right) = O\left(\sum_{i=0}^{h-1} 2^i (h-i)\right).$$

Dimostreremo ora (Claim 1 + Claim 2) che $\sum_{i=0}^{h-1} 2^i (h-i) \in O(n)$, che implica che la complessità totale della costruzione bottom-up dello heap è $O(n)$ ($\Rightarrow \Theta(n)$).

Claim 1 (Esercizio C-4-36 [GTG14])

$\sum_{\ell=1}^{h} \ell \left(\frac{1}{2}\right)^{\ell} < 3$

Dim. Osserviamo che $\forall \ell \geq 1 \quad \ell (1/2)^{\ell} \leq (3/4)^{\ell}$

(dimostrabile per induzione su $\ell \geq 1$ - ESERCIZIO)

$$\sum_{\ell=1}^{h} \ell (1/2)^{\ell} \leq \sum_{\ell=1}^{h} (3/4)^{\ell}$$

$$= \frac{(3/4)^{h+1} - (3/4)}{(3/4) - 1}$$

$$= \frac{(3/4) - (3/4)^{h+1}}{1 - (3/4)}$$

$$< \frac{3/4}{1/4} = 3$$

□

Claim 2

$\sum_{i=0}^{h-1} 2^i (h-i) \in O(n)$

$$\sum_{i=0}^{h-1} 2^i (h-i) = 2^h \sum_{i=0}^{h-1} \frac{2^i}{2^h} (h-i) = 2^h \cdot \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}}$$

Cambio di variabile: $\ell = h - i$

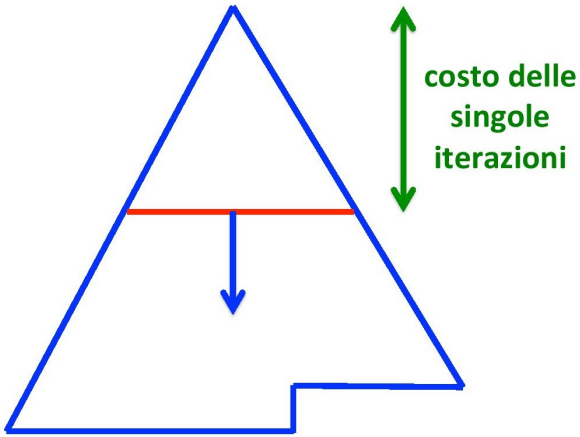
$$2^h \cdot \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}} = 2^h \sum_{\ell=1}^{h} \ell (1/2)^\ell < 2^h \cdot 3$$

$h = \lfloor \log_2 n \rfloor \Rightarrow 2^h \cdot 3 \in O(n)$

$$\Rightarrow \sum_{i=0}^{h-1} 2^i (h-i) \in O(n)$$

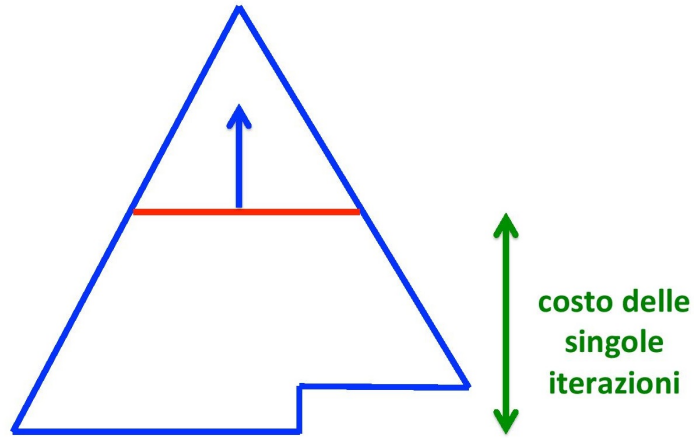
□

Confronto degli approcci top-down e bottom-up



Top-Down

- 2^i operazioni di costo $\Theta(i)$
- più aumenta la taglia del livello e più aumenta il costo dell'up-heap bubbling
- In tutto: $\Theta(n \log n)$



Bottom-Up

- 2^i operazioni di costo $\Theta((\log n) - i)$
- più aumenta la taglia del livello e più diminuisce il costo del down-heap bubbling
- In tutto: $\Theta(n)$

N.B. Entrambe le soluzioni (top-down e bottom-up) possono essere eseguite direttamente su array senza utilizzare spazio aggiuntivo ($\Rightarrow$ *IN-PLACE*)

Sorting tramite Priority Queue

Sia $S = S[0]\ S[1] \ldots S[n-1]$ una sequenza di n chiavi da ordinare.

Algoritmo `pqSort(`S`)`

n chiavi Priority Queue n chiavi ordinate

Fase A: si inseriscono le n chiavi in P una alla volta invocando il metodo **insert** (considerando le chiavi come entry).

Fase B: si rimuovono le n chiavi da P una alla volta invocando il metodo **removeMin**.

Complessità di pqSort(S)

- P lista non ordinata
  - **Fase A:** $\Theta(n)$
  - **Fase B:** $\Theta\left(\sum_{i=1}^{n} i\right) \in \Theta\left(n^2\right)$ (*SelectionSort*)
- P lista ordinata
  - **Fase A:** $\Theta\left(\sum_{i=1}^{n} i\right) \in \Theta\left(n^2\right)$ (*InsertionSort*)
  - **Fase B:** $\Theta(n)$
- P heap (su array)
  - **Fasi A, B:**: $\Theta\left(\sum_{i=1}^{n} \log i\right) \in \Theta(n \log n)$ (*HeapSort*)
  - con costruzione bottom-up la Fase A scende a $\Theta(n)$, mentre la B rimane a $\Theta(n \log n)$

Osservazione: *InsertionSort* e *SelectionSort* possono essere implementati *in-place* in modo semplice. **E HeapSort?**

HeapSort in-place

Idea: Implementiamo una variante di HeapSort in place usando la stessa sequenza S come sequenza di input, priority queue e sequenza di output. La variazione rispetto a quella presentata prima consiste nell'usare un max-heap invece che uno heap standard.

Fase A: riorganizza $S[0 \div n-1]$ in modo che le chiavi rappresentino un max-heap (la chiave in un nodo interno è $\geq$ delle chiavi nei figli).

Fase B: riorganizza $S[0 \div n-1]$ in modo che le chiavi risultino ordinate

Fase A: $S \rightarrow$ max-heap

La trasformazione di S in un max-heap è implementata come segue:

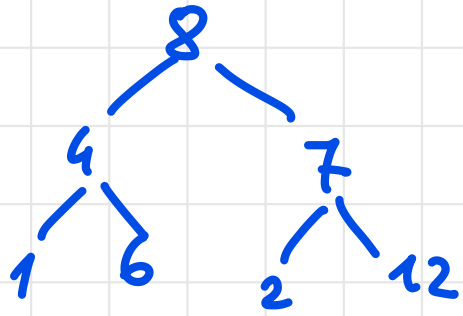
Sia `indexMaxChild`(S, i) un metodo che restituisce l'indice del figlio di $S[i]$ con chiave massima ($2i+1$ o $2i+2$), se $S[i]$ è un nodo interno, (cioè $2i+1 \leq$ last), e restituisce `null`, se $S[i]$ è foglia.

```
last ← n − 1;
for j ← ⌊(n − 2)/2⌋ downto 0 do
    /* Down-heap bubbling a partire da S[j]                    */
    i ← j;
    k ← indexMaxChild(S, i);
    while ((k ≠ null) AND (S[i] < S[k])) do
        swap(S[i],S[k]);
        i ← k;
        k ← indexMaxChild(S, i);
```

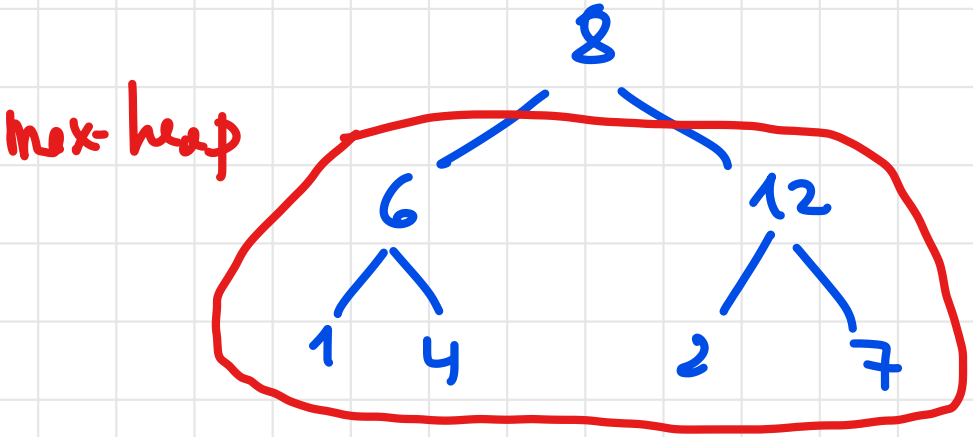
costruzione bottom-up di un max-heap (solo chiavi)

Esempio

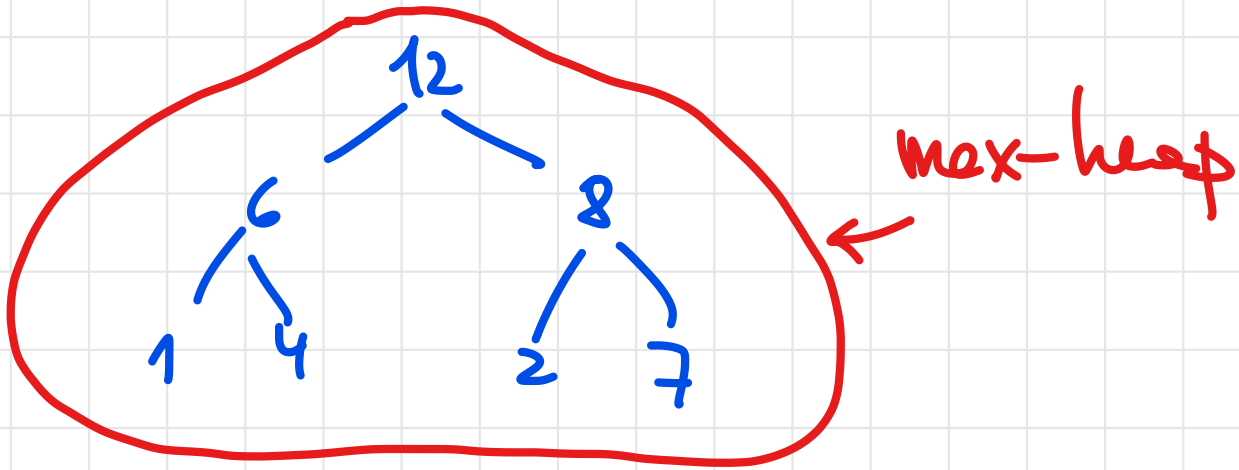
$n = 7$ $\quad S = [8 \quad 4 \quad 7 \quad 1 \quad 6 \quad 2 \quad 12]$



Dopo iterazione 2, 1:



Dop l'iterazione j=0

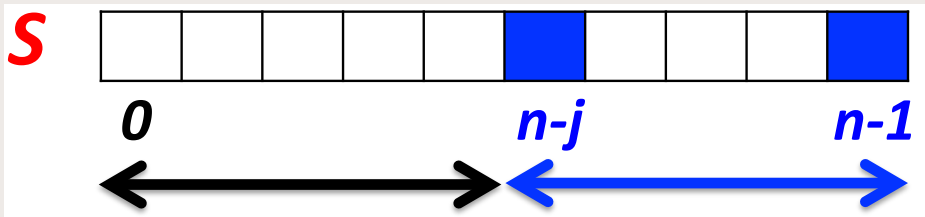


Come vettore: $S = [12\ 6\ 8\ 1\ 4\ 2\ 7]$

Fase B: max-heap $\rightarrow$ S ordinata

La Fase B è basata su un ciclo `for` di $n-1$ iterazioni che mantiene il seguente invariante alla fine della j-esima iterazione: ($j = 0, \ldots, n-1$, dove $j = 0$ è l'inizio del ciclo)

Invariante



- $S[0 \div n-j-1]$ contiene le $n-j$ chiavi più piccole organizzate come max heap
- $S[n-j \div n-1]$ contiene le j chiavi più grandi in ordine crescente

Fase B: max heap $\rightarrow$ S ordinata

L'implementazione è la seguente:

```
last ← n − 1;
for j ← 1 to n − 1 do
    /* Down-heap bubbling a partire da S[0]                    */
    swap(S[last],S[0]);
    last ← n − j − 1;
    i ← 0;
    k ← indexMaxChild(S, i);
    while (k ≠ null) AND (S[i] < S[k]) do
        swap(S[i],S[k]);
        i ← k;
        k ← indexMaxChild(S, i);
```

mi segnala l'ultima cella di S che fa parte del max-heap

deve tenere conto che il sotto-albero del max-heap scende di 1 ad ogni iterazione

Esercizio

Argomentare che l'invariante indicato prima viene effettivamente mantenuto dal ciclo for.

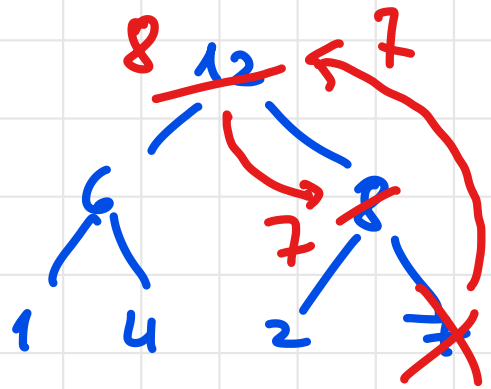
Esempio: dopo costruzione bottom-up

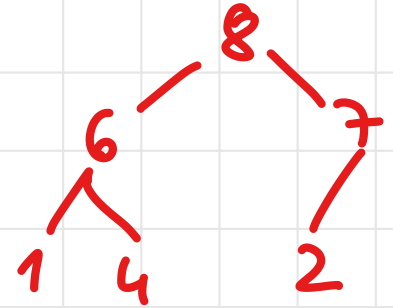
$S \equiv [12\ 6\ 8\ 1\ 4\ 2\ 7]$ ottenuto dopo FASE A

Iterazione $j = 1$. Dopo questa iterazione

$S = [8\ 6\ 7\ 1\ 4\ 2 \mid 12]$

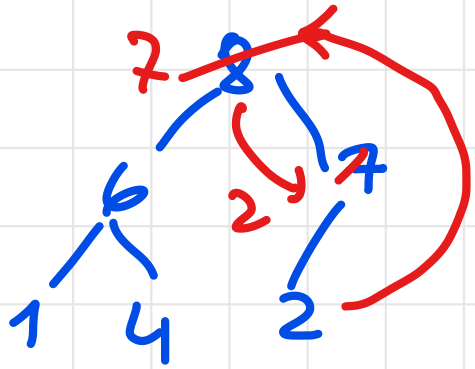
max heap | sequenza ordinata



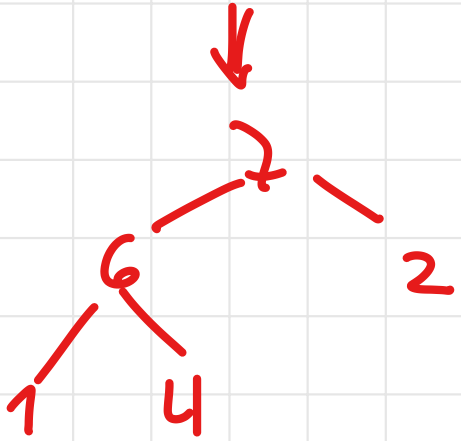


max heap dopo iterazione j

Dopo l'iterazione $j = 2$



$S \equiv [\underbrace{7\ 6\ 2\ 1\ 4}_{\text{max-heap}}\ !\ \underbrace{8\ 12}_{\text{sequenza ordinata}}]$



max-heap dopo iterazione 2

∴ continuare per esercizio

Complessità di HeapSort in place

- **Fase A:** equivale alla costruzione bottom-up

  $\Rightarrow \Theta(n)$

- **Fase B:** equivale all'esecuzione di $n-1$ removeMax da heap progressivamente più piccoli

  $\Rightarrow \Theta\left(\sum_{i=1}^{n-1} \log i\right) \in \Theta(n \log n)$

La complessità di HeapSort in place è $\Theta(n \log n)$

Esercizio

Sia $S = S[0], S[1], \ldots, S[n-1]$ una sequenza contenente n chiavi distinte e sia m un indice intero tale che $1 \leq m \leq n/\log_2 n$. Progettare un algoritmo che in tempo $O(n)$ determini la m-esima chiave più piccola di S. (Analizzare la complessità dell'algoritmo per far vedere che è $O(n)$.)

Implementazione di Priority Queue in Java

Il package `java.util` contiene la classe `PriorityQueue<E>`, i cui ogetti sono Priority Queue Q implementate tramite heap.

- Q non contiene coppie (chiave, valore) ma oggetti di un tipo E
- La priorità è definita utilizzando tutto l'oggetto come chiave, come se gli elementi di Q fossero le chiavi di entry vuote.
- Di default, gli elementi di Q sono confrontati in base all'ordine naturale associato al tipo E.
- In pratica, se vogliamo usare la classe per realizzare una Priority Queue le cui entry sono coppie (chiave,valore), dobbiamo:
  1. Rappresentare le entry tramite una classe che estende l'interfaccia `Comparable`.
  2. Definire il metodo `compareTo` imponendo un *ordinamento naturale delle entry* basato sulle chiavi.

Ad esempio:

```
class MyEntry implements Comparable<MyEntry>{

  /* Variabili e metodi che definiscono la entry */

  @Override
  public int compareTo(MyEntry x) {

  /* confronto tra chiamante e x basato sulla chiave */

  }
}
```

E creiamo la Priority Queue come segue

```
PriorityQueue<MyEntry> Q = new PriorityQueue<MyEntry>();
```

A questo punto, il metodo `Q.poll()` restituirà la entry con chiave minima.

Riferimenti: API di Java e i Par. 9.2.2 e 9.3.5 di [GTG14]

Riepilogo

- Priority Queue: definizione e implementazione tramite liste.
- Albero binario completo: definizione e altezza.
- Heap: definizione, e mapping efficiente su array tramite level numbering.
- Implementazione dei metodi della Priority Queue tramite heap (su array).
- Costruzione (top-down e bottom-up) di uno heap partendo da un array di n entry:
- pqSort
  - Implementazione con lista non ordinata (*SelectionSort*).
  - Implementazione con lista ordinata (*InsertionSort*).
  - Implementazione con heap su array (*HeapSort*).