Dati e Algoritmi 1: 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.

Interfaccia:

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

Priority Queue

Definizione

Priority Queue: collezione di entry le cui chiavi (non necessariamente distinte) rappresentano *priorità* e provengono da un universo totalmente ordinato K.
Come tipo di dato astratto, la Priority Queue deve permettere di

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

Osservazioni:

- Di solito si assume che la entry di massima priorità sia quella con chiave minima. Esistono però dei casi in cui conviene considerare di massima priorità la entry con chiave massima.
- Come vedremo mantenere le entry ordinate permette di identificare immediatamente la entry di massima priorità, ma non è la scelta implementativa migliore.

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();
}
```

Esempio

Partiamo da Priority Queue vuota

Operazione	Output	Priority Queue
`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	$(5,A)$ $(9,C)$ $(7,D)$
`isEmpty`$()$	*false*	$(5,A)$ $(9,C)$ $(7,D)$

Applicazioni delle priority queue

- Gestione liste d'attesa (ad es. aeroporti)
- Scheduling di processi in sistema operativo o di richieste di banda nelle reti in base a priorità per garantire QoS
- Simulazione discreta a eventi
- Dijkstra: algoritmo per *Single Source Shortest Path*
- Top-K pattern discovery

Implementazione di Priority Queue tramite Liste

Caso 1: Lista doubly linked non ordinata

$P \equiv$ lista non ordinata di n entry (PositionalList<Entry< K, V >>)

Implementazione dei metodi della Priority Queue:

Metodo `min()` Complessità $= \Theta(n)$

```
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
```

Implementazione di Priority Queue tramite Liste
Caso 1: Lista doubly linked non ordinata (continua)

Metodo `insert(`k`, `x`)` Complessità $= \Theta(1)$

$e \leftarrow (k, x)$;
`P.addLast(`e`);`
return `e`

Metodo `removeMin()` Complessità $= \Theta(n)$ (esercizio)

Implementazione di Priority Queue tramite liste
Caso 2: Lista doubly linked ordinata

$P \equiv$ lista ordinata di n entry (PositionalList<Entry< K, V >>)

N.B. Si assume un ordinamento per chiave crescente

Metodo `min()` Complessità $= \Theta(1)$

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

Metodo `removeMin()` Complessità $= \Theta(1)$ (esercizio)

Implementazione di Priority Queue tramite Lista ordinata doubly linked (continua)

Metodo `insert(`k`, `x`)` Complessità $= \Theta(n)$

```
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
```

Priority Queue tramite Liste: riepilogo

1. Lista non ordinata doubly-linked
   - i nodi della lista contengono entry
   - P.insert(k,x): complessità $\Theta(1)$
   - P.min(), P.removeMin(): complessità $\Theta(n)$

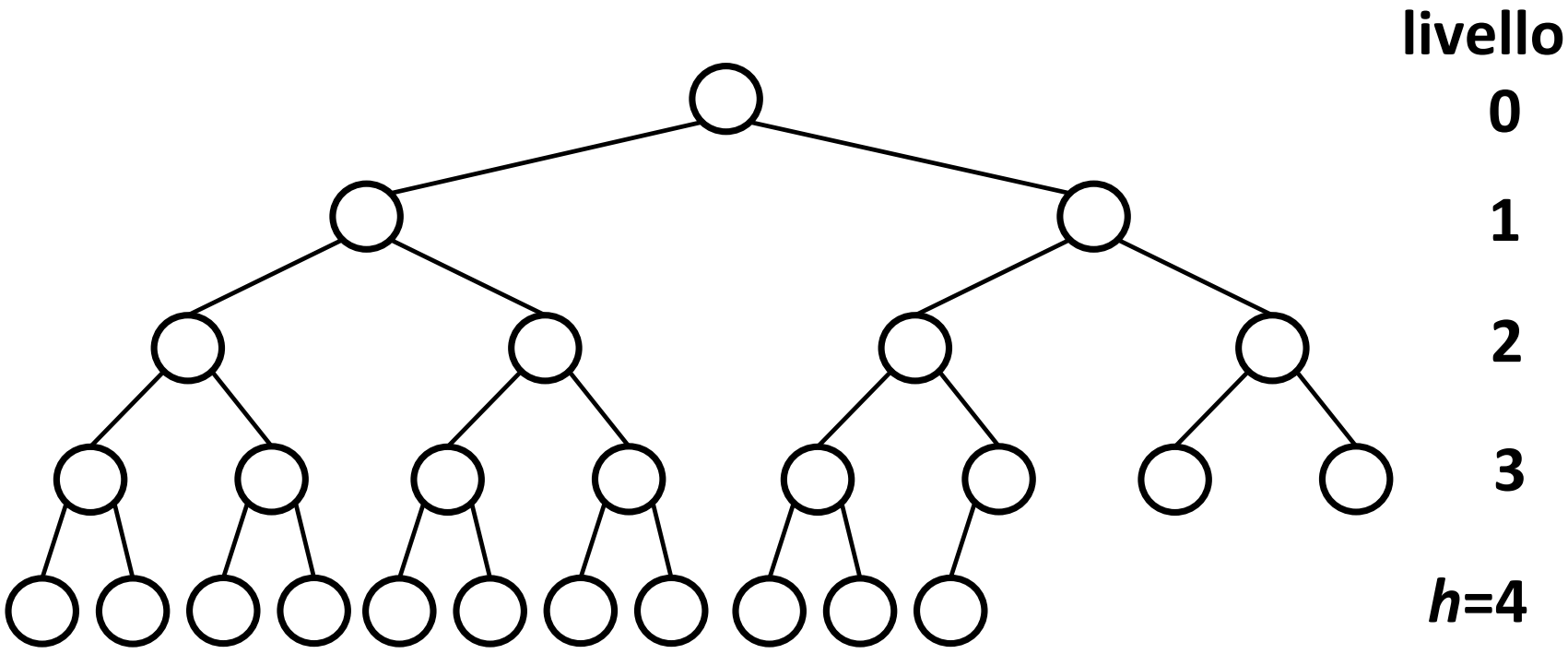
2. Lista ordinata doubly-linked
   - i nodi della lista contengono entry ordinate per chiave
   - P.insert(k,x): complessità $\Theta(n)$
   - P.min(), P.removeMin(): complessità $\Theta(1)$

È possibile *bilanciare* il costo delle diverse operazioni?

Albero Binario Completo T

Albero binario di altezza h 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 può avere il solo figlio sx.



Proposizione

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

Dimostrazione

Si ha che

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

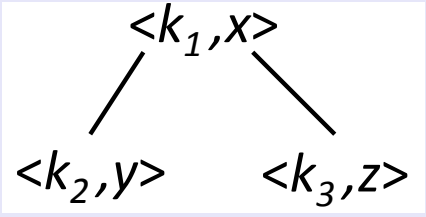
$\Rightarrow h \leq \log_2 n < h + 1$

$\Rightarrow (\log_2 n) - 1 < h \leq \log_2 n$ $\square$

Heap

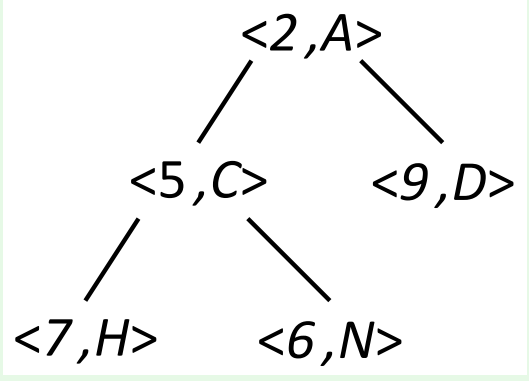
Definizione

Un min-heap (o semplicemente heap) è un albero binario completo i cui nodi memorizzano entry e soddisfano la seguente heap-order property:



$$k_1 \leq \min\{k_2, k_3\}$$

Esempio



Definizione

Max-heap: uno heap nella cui definizione "$\leq$" $\rightarrow$ "$\geq$"

Nota

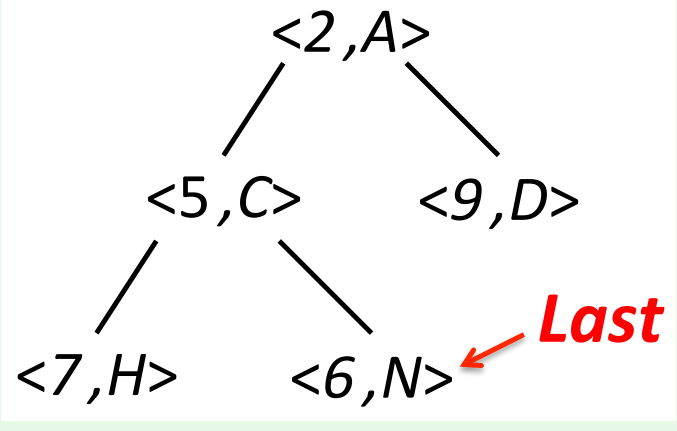
Uno heap è caratterizzato da 2 proprietà:

- albero binario completo $\Rightarrow$ altezza $\Theta(\log n)$
- heap-order property $\Rightarrow$ `min()` in $O(1)$

Definizione

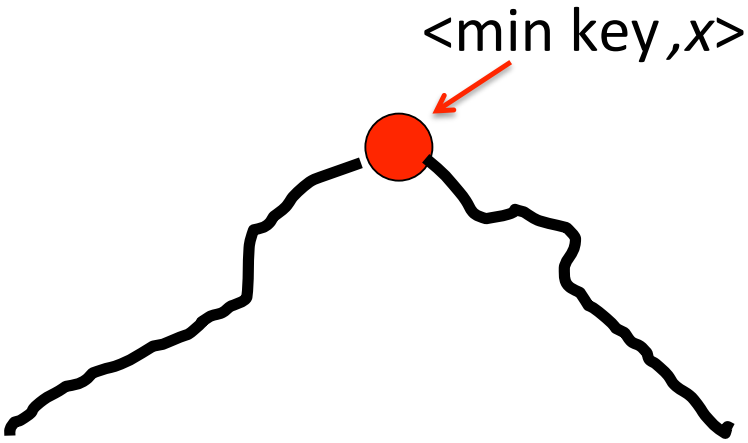
Nodo Last in uno heap di altezza h: nodo più a dx al livello h.

Esempio



Proprietà di uno heap

1. La radice contiene una entry con chiave minima.



heap = "mucchio"

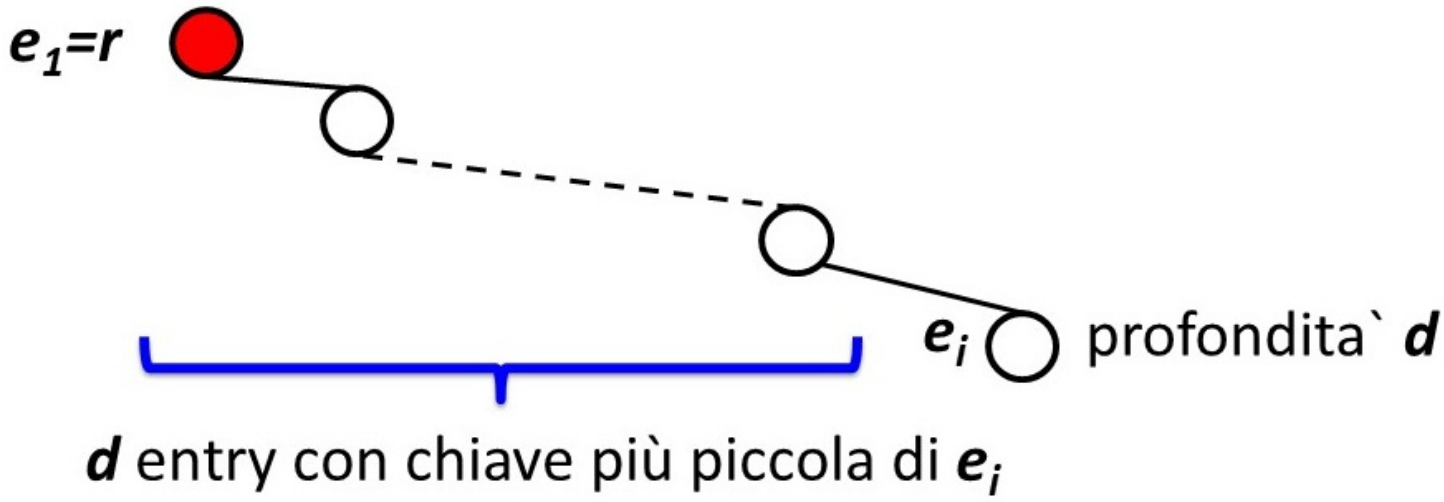
2. le chiavi incontrate lungo un cammino dalla radice verso le foglie formano una sequenza non decrescente

Sia P uno heap con n entry. Dalla definizione si possono anche ricavare le seguenti proprietà aggiuntive.

1. Se le chiavi sono tutte distinte, detta e_i la entry con l'i-esima chiave più piccola si ha che e_i è a profondità $d < i$ in P.
2. Se le chiavi sono tutte distinte, detta $e_{\max}$ la entry con chiave massima si ha che $e_{\max}$ sta in una foglia di P (es. R-9.10 in [GTG14]).
3. Sia e_v la entry nel nodo v di P. Allora per qualsiasi discendente u di v in P si ha che e_u.getKey() $\geq$ e_v.getKey().

Dimostriamo le proprietà enunciate nel lucido precedente.

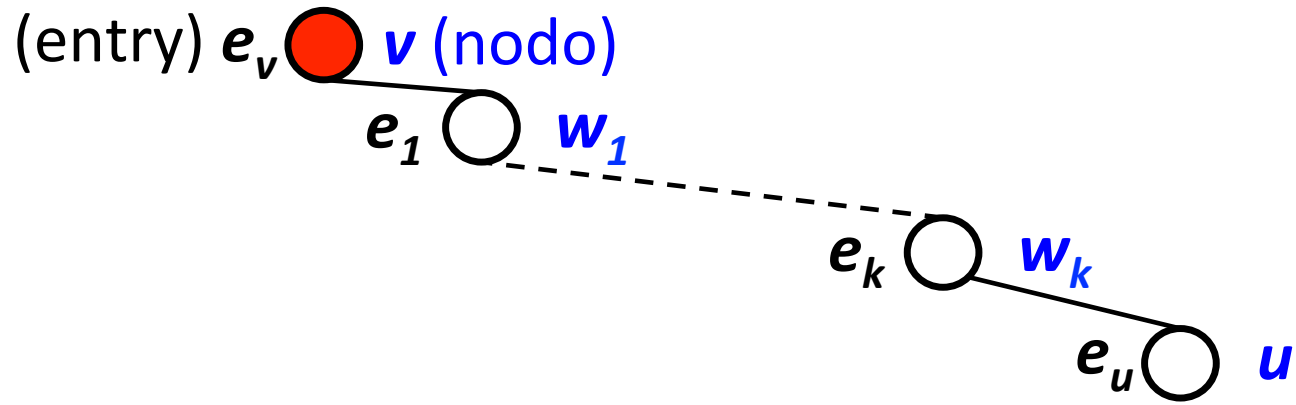
1. $e_i \equiv$ entry con l'i-esima chiave più piccola



$\Rightarrow d < i$

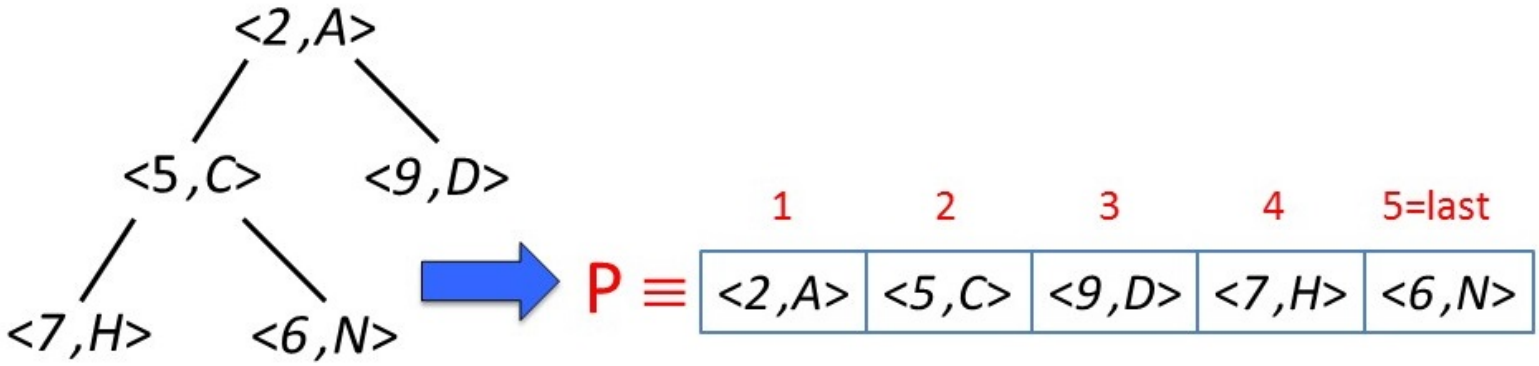
2. Se e_{max} fosse in un nodo interno, tale nodo avrebbe un nodo figlio con chiave maggiore di e_{max}.key(), quindi e_{max} non sarebbe l'entry con chiave massima $\Rightarrow$ assurdo. (N.B.: *Esempio di dimostrazione per assurdo*)

❸ nodo v con entry e_v



$\Rightarrow e_v.\text{getKey}() \leq e_1.\text{getKey}() \leq \cdots \leq e_k.\text{getKey}() \leq e_u.\text{getKey}()$

Realizzazione di uno heap tramite array



Level Numbering

Entry radice $\equiv P[1]$

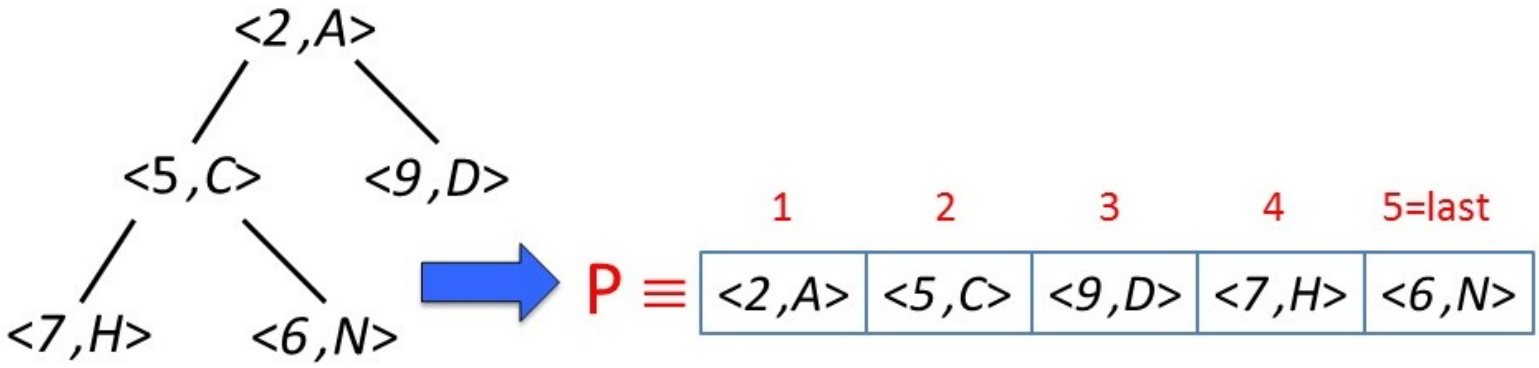
Entry figlie di $P[i]$:

- $P[2i]$
- $P[2i+1]$

Entry padre di $P[i] = P\left[\lfloor i/2 \rfloor\right]$

Oss. In [GTG14, Par. 8.3.2] il level numbering inizia da 0

Realizzazione di uno Heap tramite array (continua)



Osservazione: La realizzazione tramite array riduce lo spazio occupato rappresentando implicitamente i link padre-figlio (*struttura dati implicita*).

Notazione (per pseudocodice)

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

- $P[i]$.getKey()
- $P[i]$.getValue()

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

Proprietà del level numbering

Per ogni albero binario completo con $n \geq 1$ nodi e altezza h mappato su un array P tramite level numbering, vale che:

- $\forall i, 0 \leq i < h$: i 2^i nodi del livello i, presi da sx a dx, sono mappati in $P[2^i], P[2^i + 1], \ldots, P[2^{i+1} - 1]$
- i nodi del livello h, presi da sx a dx, sono mappati in $P[2^h], P[2^h + 1], \ldots, P[n]$ ($\Rightarrow last = n$)

Per ogni n, la dimostrazione può essere fatta per induzione sul numero del livello.

Osservazione

Rappresentare un albero binario su array tramite level numbering risulta molto *space-efficient* nel caso in cui l'albero sia completo. Però senza questa ipotesi tale rappresentazione può arrivare a richiedere uno spazio esponenziale nel numero di nodi dell'albero.

Pensare a un esempio come esercizio.

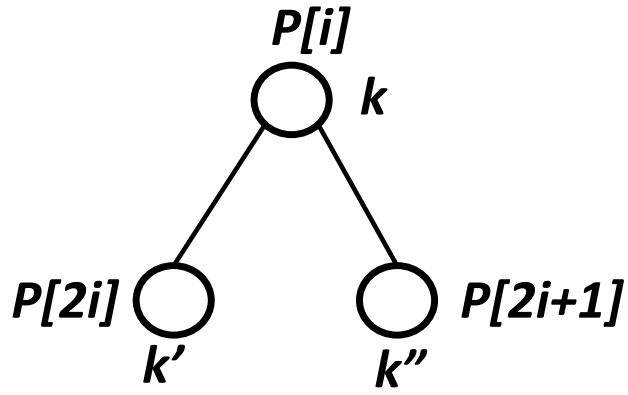
Esercizio

Sia P un array di n entry $P[1]P[2]\ldots P[n]$ le cui chiavi sono in ordine non decrescente. P rappresenta uno heap? Motivare la risposta.

Svolgimento

Deve soddisfare due proprietà:

- albero binario completo $\Rightarrow$ OK
- heap-order property



grazie all'ordinamento $k \leq k'$ e $k \leq k''$ $\Rightarrow$ OK

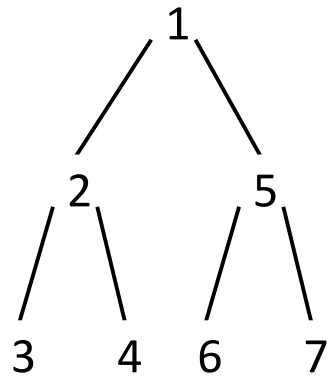
Esercizio (R-9.15 [GTG14])

Si consideri uno heap P con 7 entry con chiavi distinte.

1. È possibile che la visita in preorder restituisca le entry in ordine crescente? (Far vedere un esempio o dimostrare che non è possibile)
2. Ripetere il punto sopra per la visita inorder e postorder.

Svolgimento

1. **Si!**



2. **No:** Nella visita inorder $P[1]$ (che ha la chiave minima) appare in posizione 4, e in quella in postorder appare in ultima posizione.

Implementazione di una Priority Queue tramite heap

Consideriamo uno heap P, implementato tramite array, contenente n entry, e vediamo come implementare i metodi dell'interfaccia `PriorityQueue`

- `size()` $\Rightarrow O(1)$ (banale)
- `isEmpty()` $\Rightarrow O(1)$ (banale)
- `min()` $\Rightarrow O(1)$ (banale: **return** $P[1]$)

- `insert(`k`,` x`)` ??
- `removeMin()` ??

Inserimento: `insert`

Idea

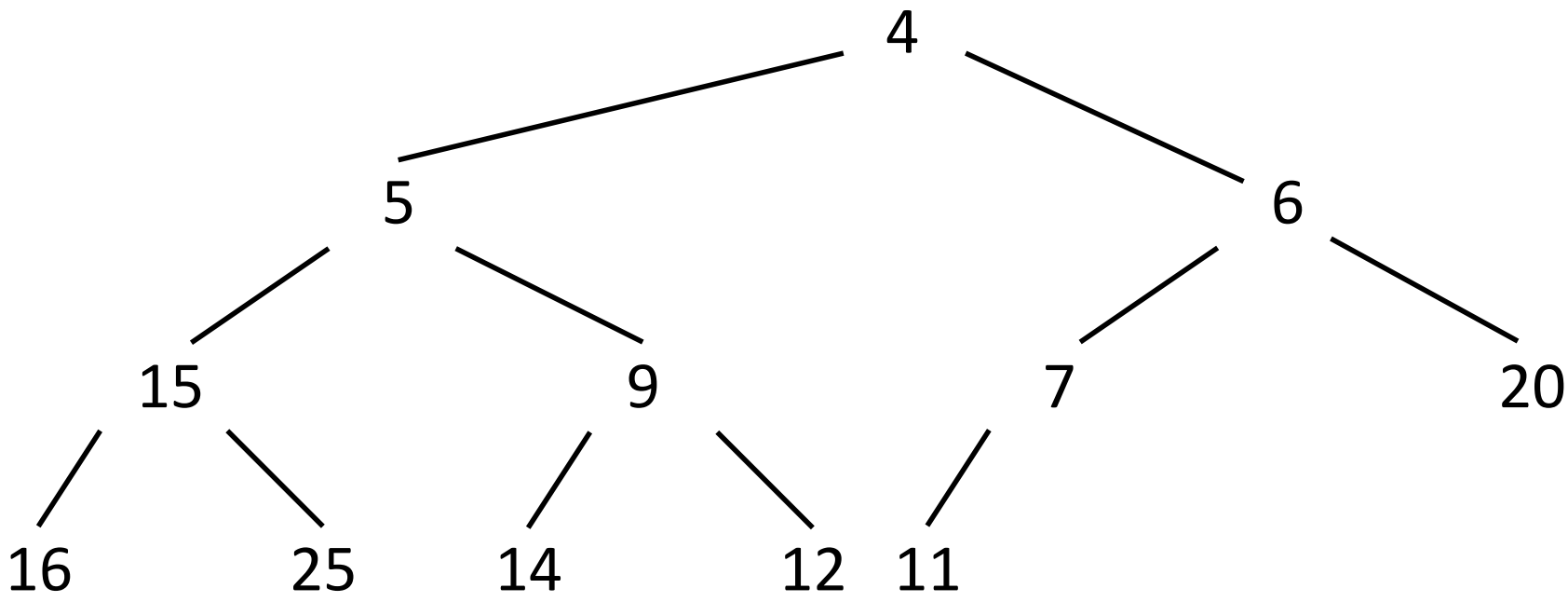
- inserisci la nuova entry alla destra del nodo last
- ricostruisci la heap-order property dello heap lungo il cammino da *last* alla radice

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

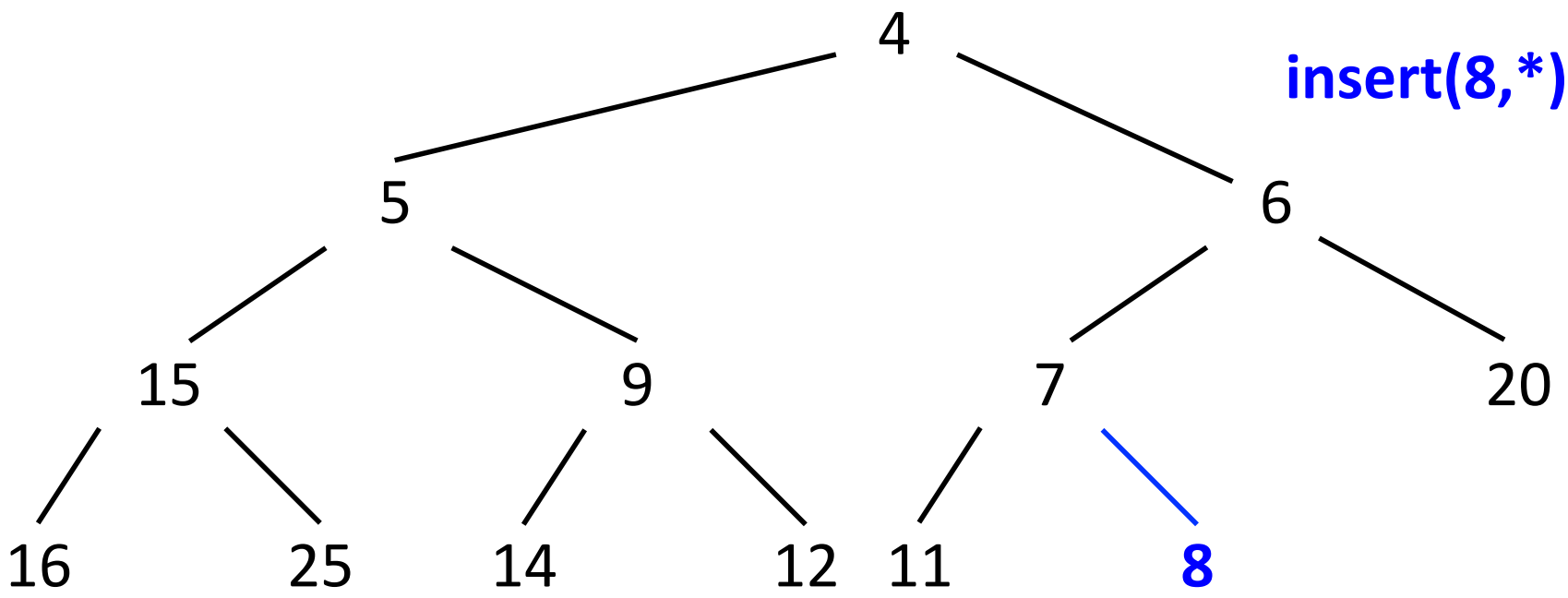
$e \leftarrow (k,x)$;
$P[++last] \leftarrow e$;
$i \leftarrow last$;
// *Up-heap bubbling*
while `((`$i > 1$`)AND(`$P[\lfloor i/2 \rfloor]$`.getKey()>`$P[i]$`.getKey()))` **do**
    `swap(` $P[i]$, $P[\lfloor i/2 \rfloor]$);
    $i \leftarrow \lfloor i/2 \rfloor$;
return e;

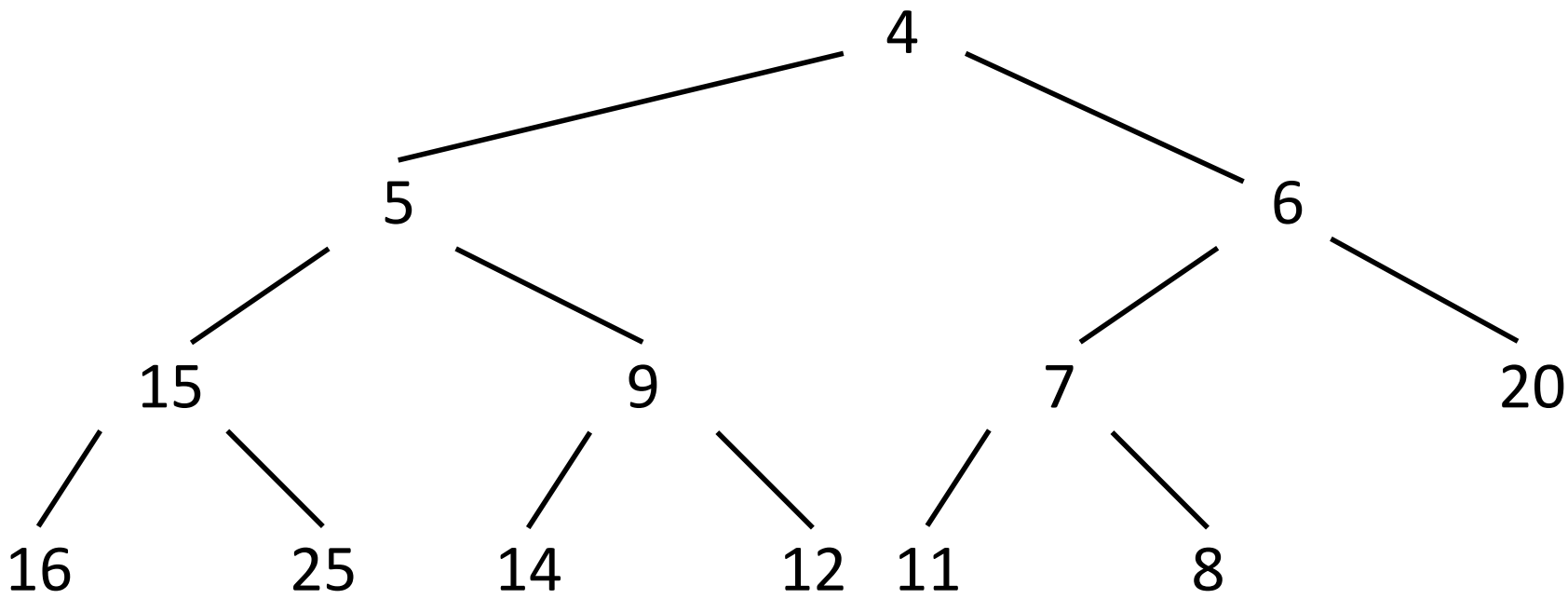
Oss. In caso di overflow, si devono prima trasferire le entry in un array più capiente (di solito di taglia doppia di quello corrente)

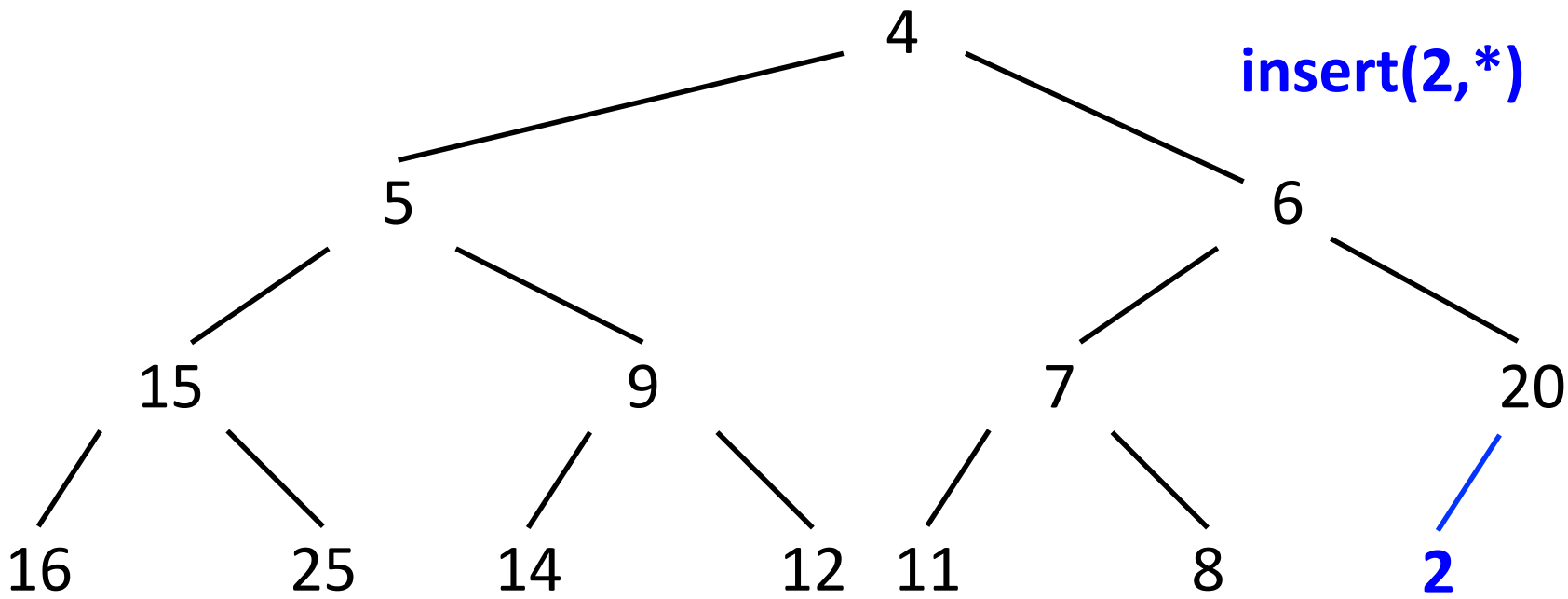
Esempio (solo chiavi)

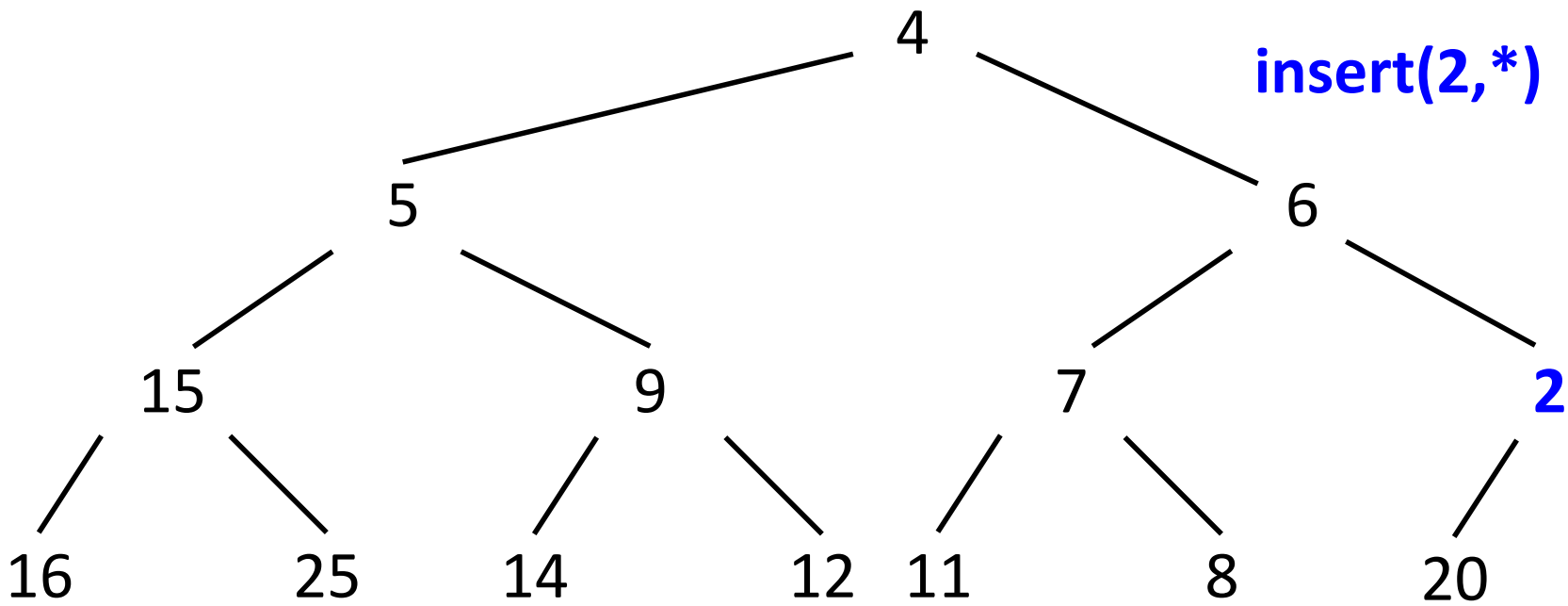


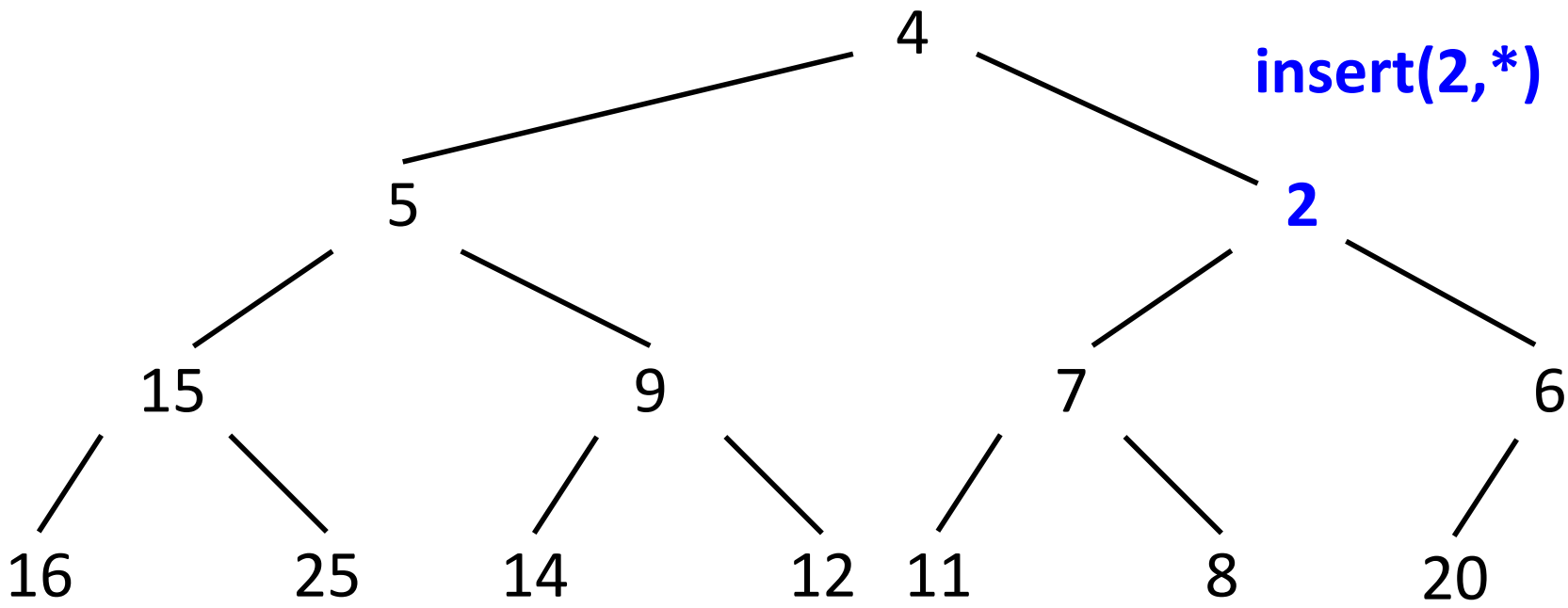
Esempio (continua)

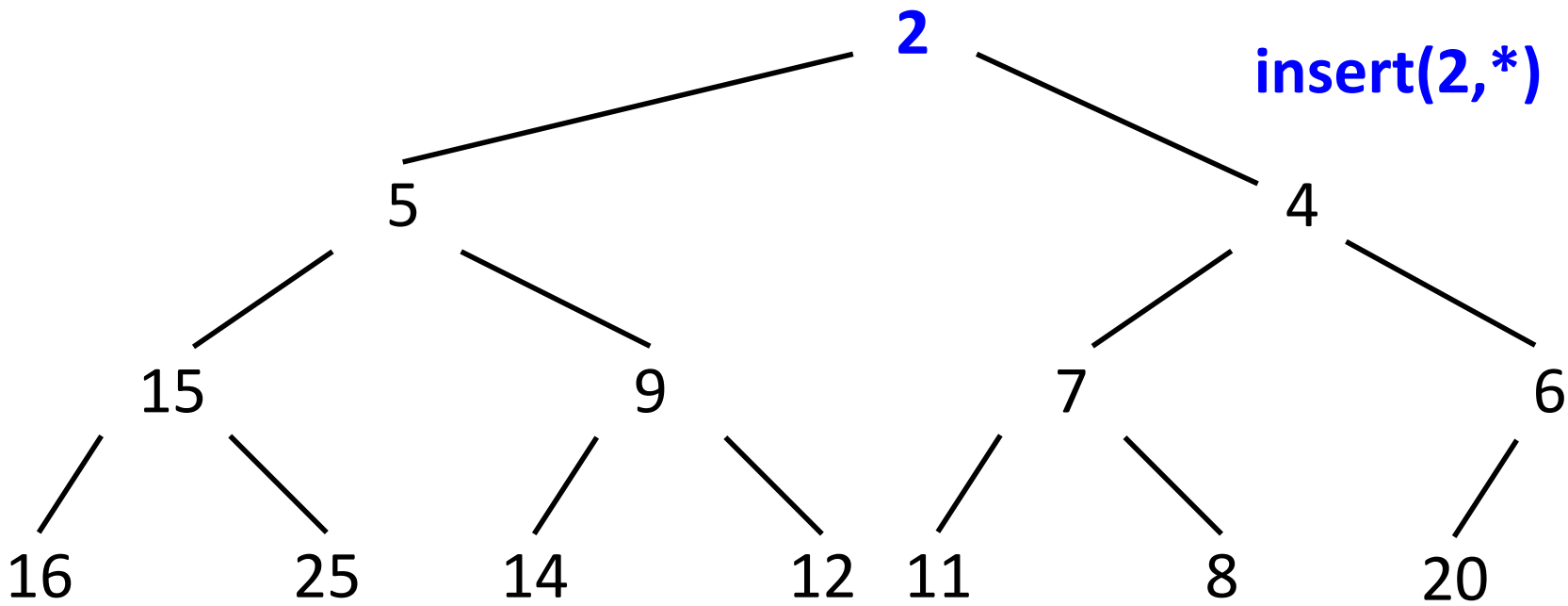


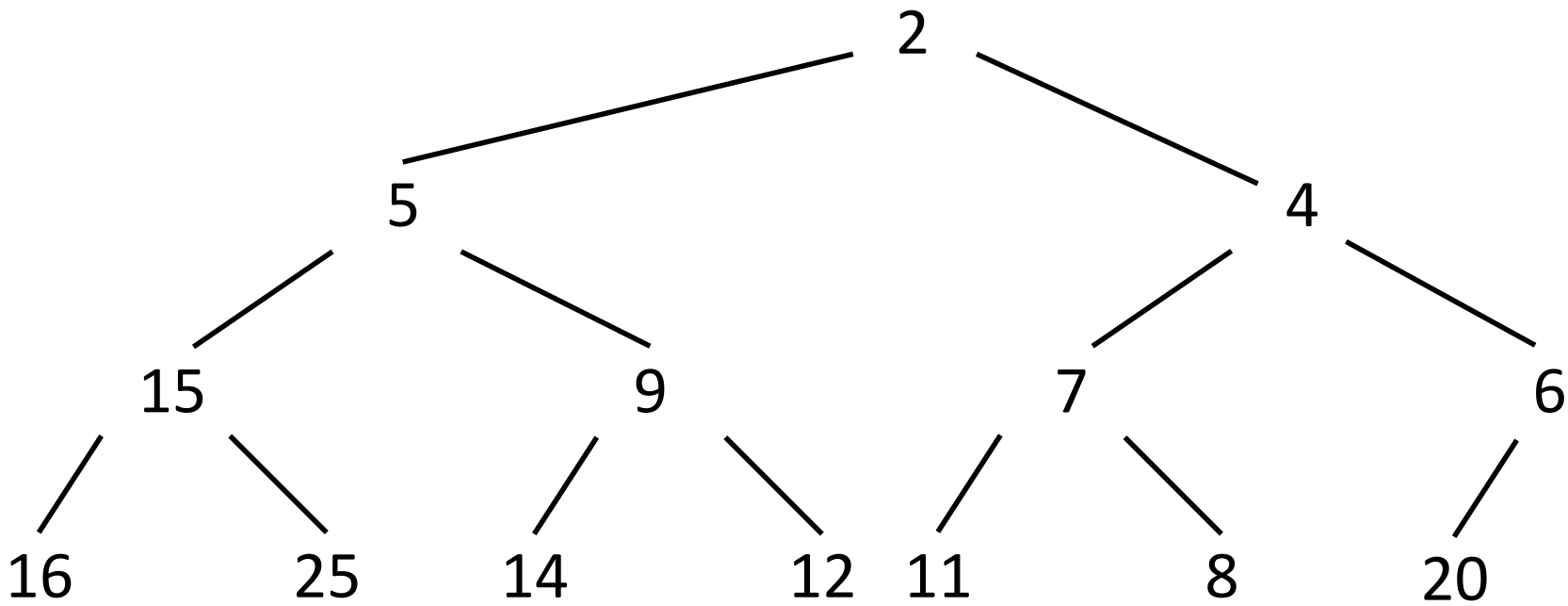

4
5
6
15
9
7
20
16
25
14
12
11
8


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


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


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


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

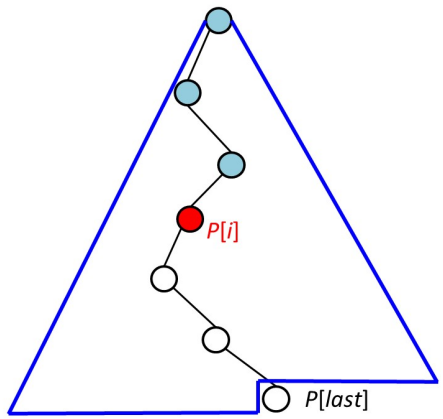

2
5
4
15
9
7
6
16
25
14
12
11
8
20

Correttezza di `insert`

Basata sul seguente invariante per il while

Invariante

Le uniche coppie antenato-discendente che possono violare la heap-order property estesa hanno come discendente $P[i]$. (Per "heap-order property estesa" si intende che la chiave nell'antenato deve essere $\leq$ di quella nel discendente.)



N.B. La struttura di albero binario completo non è mai violata

Esercizio

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

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
- 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`

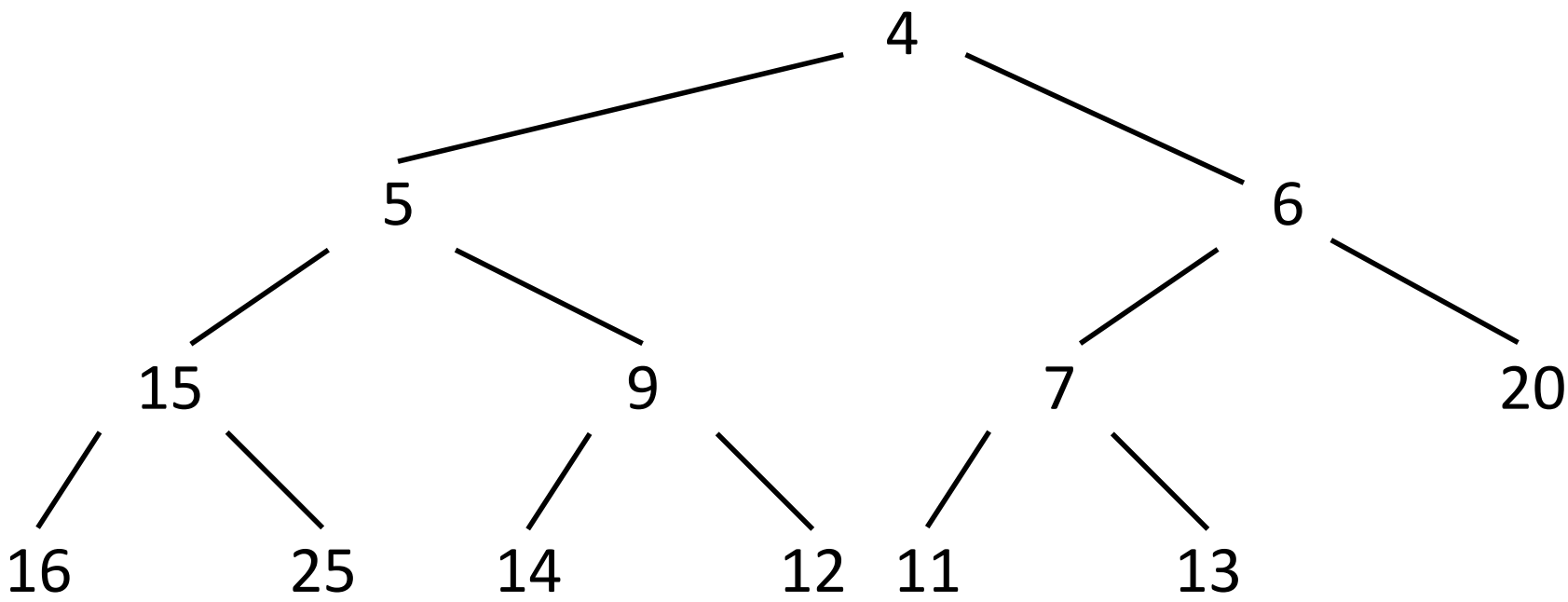
Idea

- rimuovi la entry presente nella radice dello heap
- metti la entry del nodo *Last* nella radice
- ricostruisci la heap-order property dello heap a partire dalla radice verso le foglie

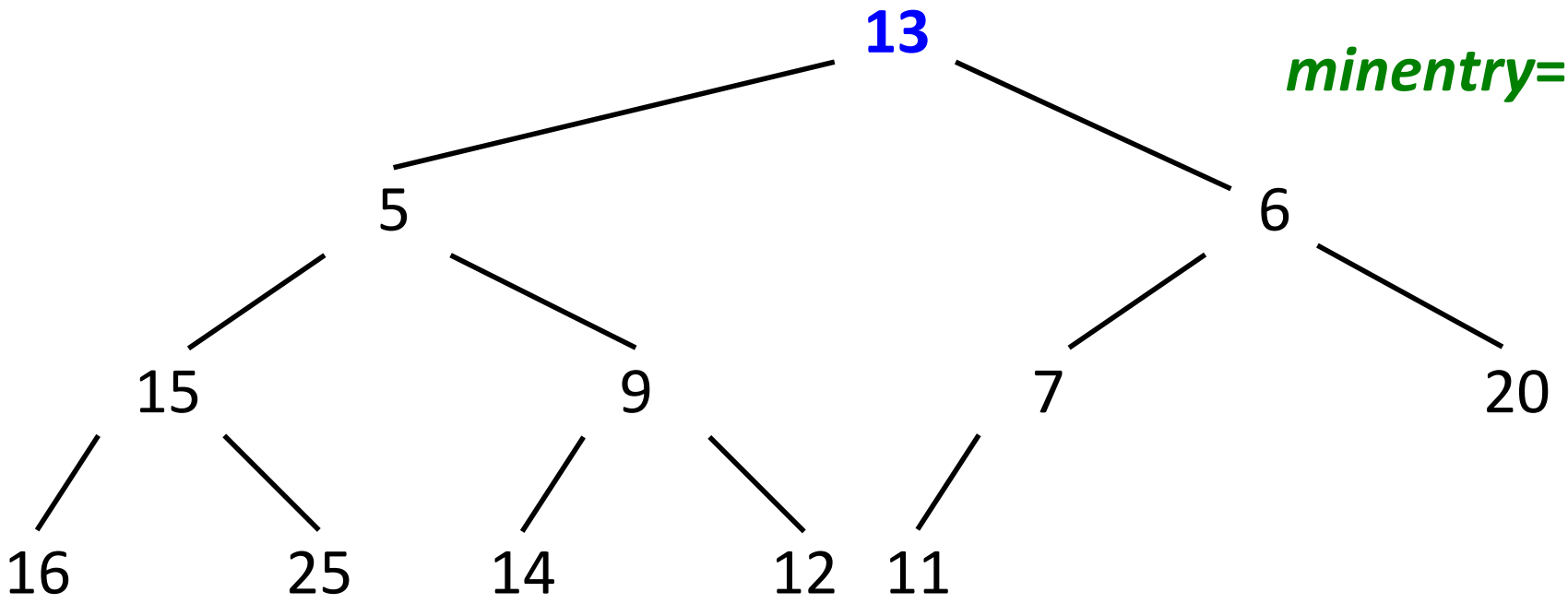
Metodo `removeMin()`

$minentry \leftarrow P[1]$;
$P[1] \leftarrow P[last--]$;
$i \leftarrow 1$;
if $2i \leq last$ **then**
    $j \leftarrow$ indice del figlio di $P[i]$ con chiave min;
    // $j = 2i$ o $j = 2i + 1$
// *Down-heap bubbling*
while $(2i \leq last)$`AND(`$P[i]$`.getKey()>`$P[j]$`.getKey())` **do**
    swap($P[i]$,$P[j]$);
    $i \leftarrow j$;
    if $2i \leq last$ **then** $j \leftarrow$ indice del figlio di $P[i]$ con chiave min;
return *minentry*

Esempio (solo chiavi)



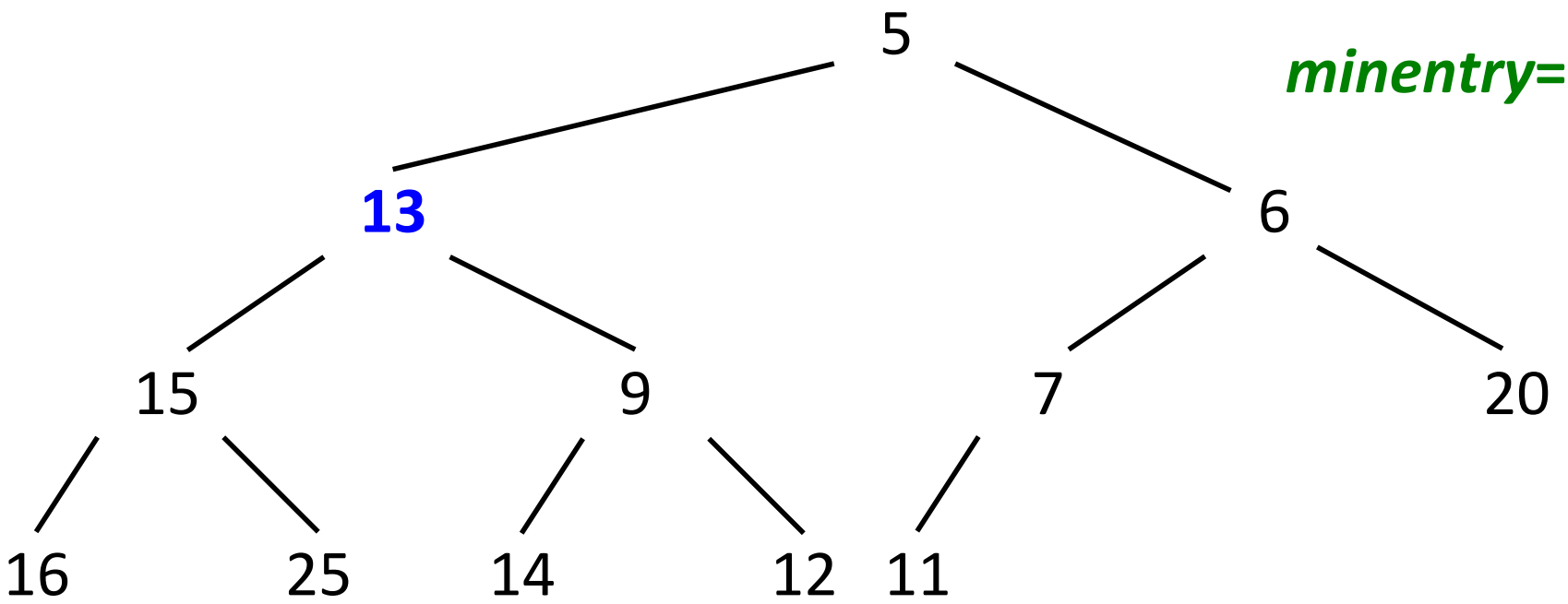
removeMin()



***minentry*= 4**

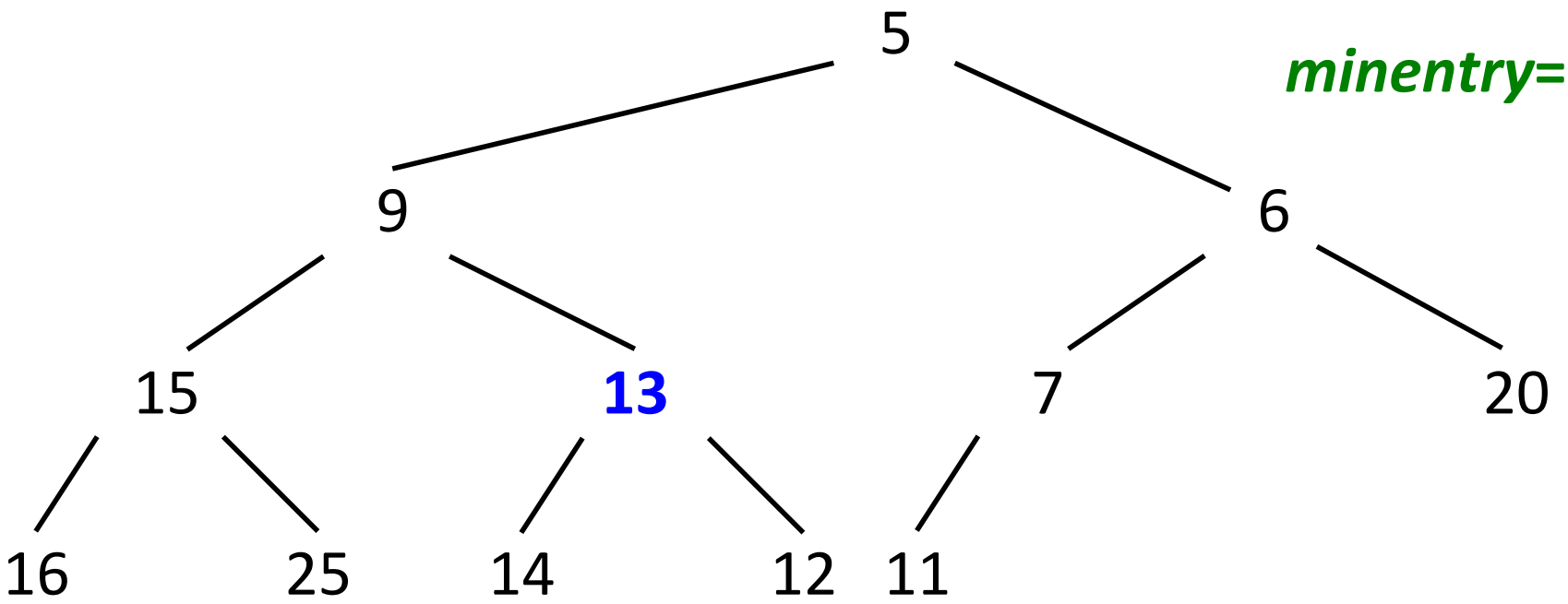
removeMin()

***minentry*= 4**



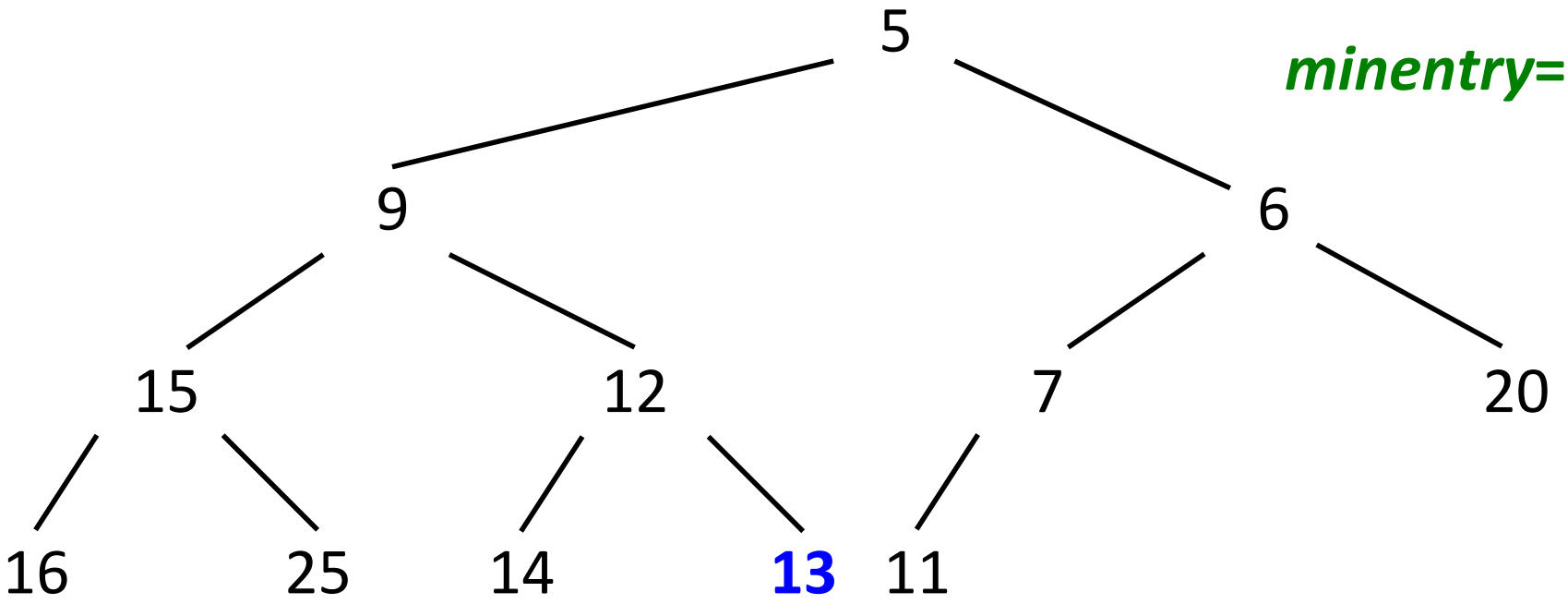
removeMin()

***minentry*= 4**



removeMin()

***minentry*= 4**



Correttezza di `removeMin`

Basata sul 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 e' $P[i]$.

Esercizio

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

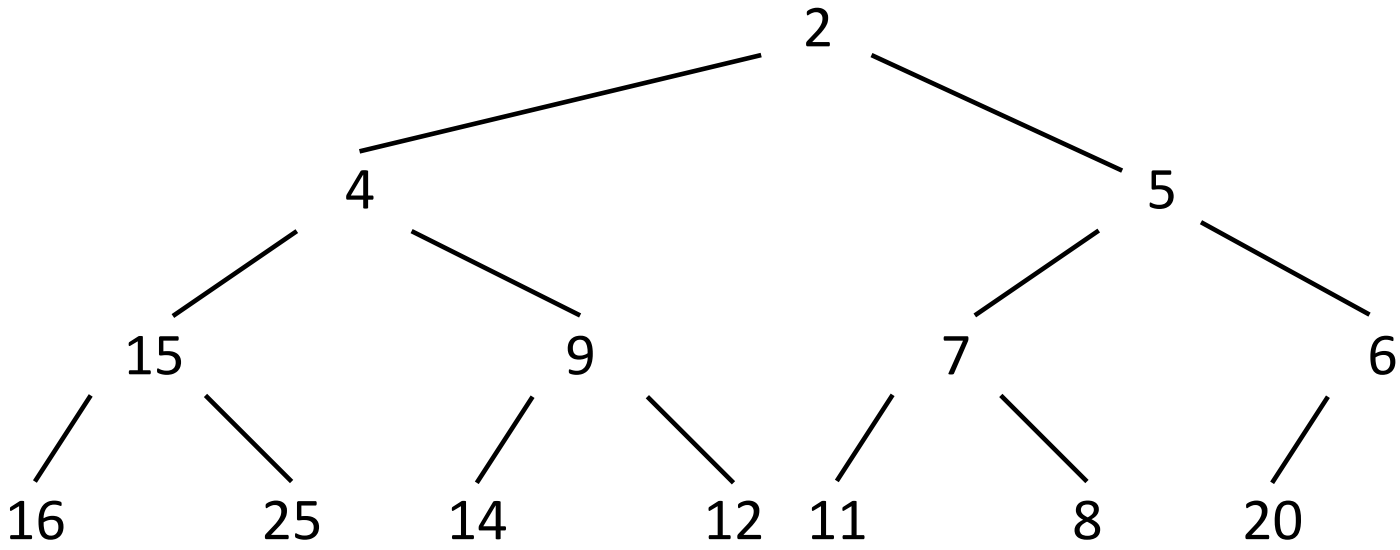
Complessità di `removeMin`

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.

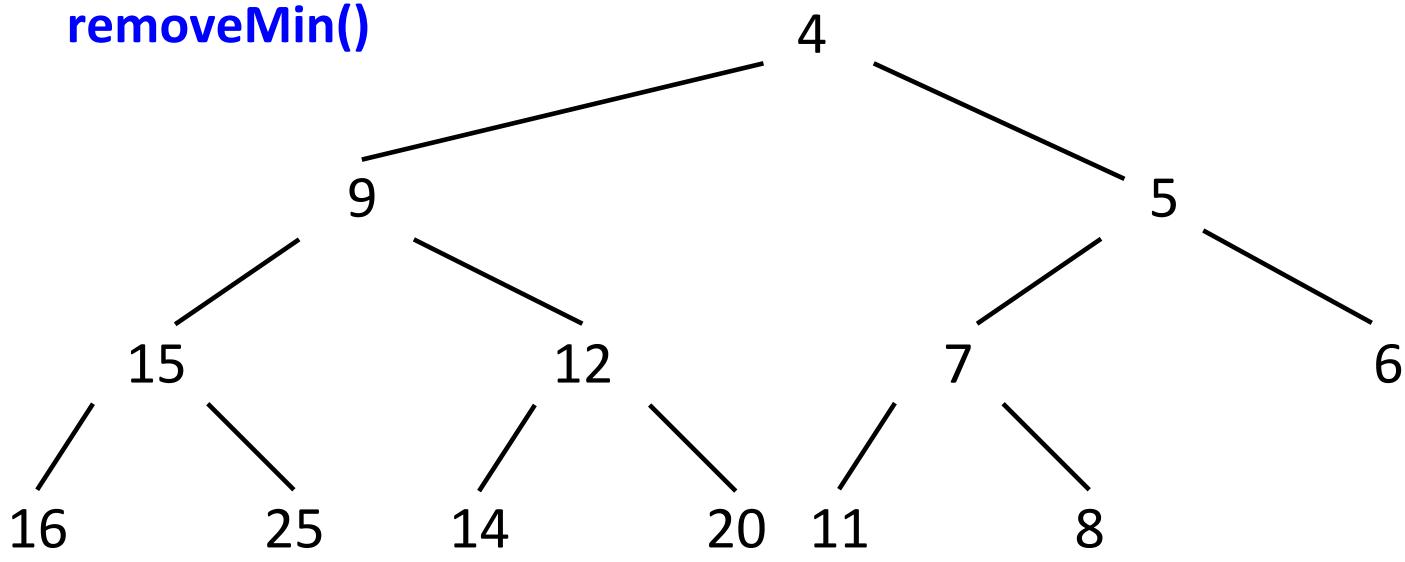
- Complessità proporzionale al numero di iterazioni del while
- Numero di iterazioni del while $\leq h$
- Esiste un'istanza che richiede esattamente h iterazioni (la entry in $P[\text{last}]$ ha chiave maggiore di tutte quelle presenti)

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

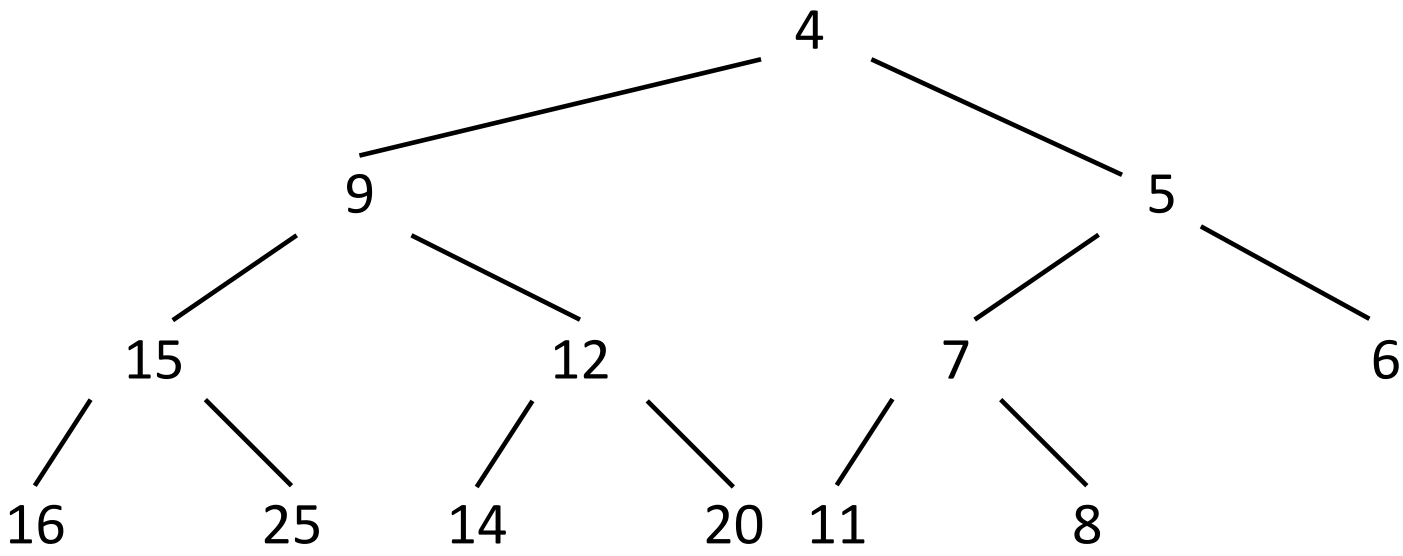
Esempio



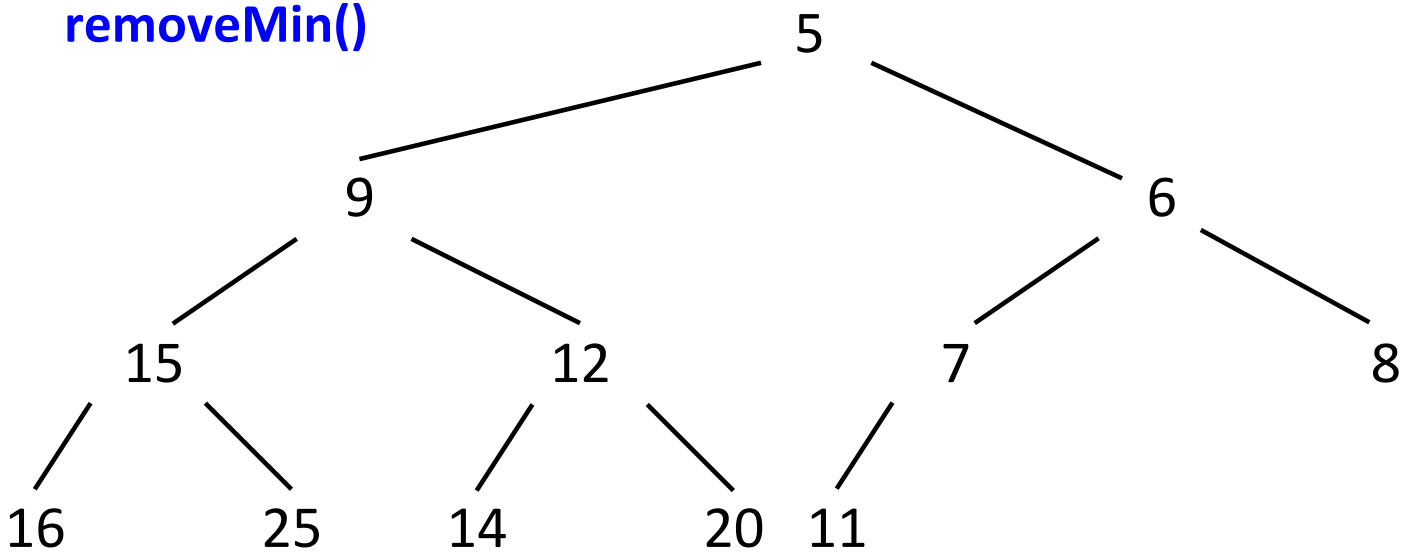
removeMin()



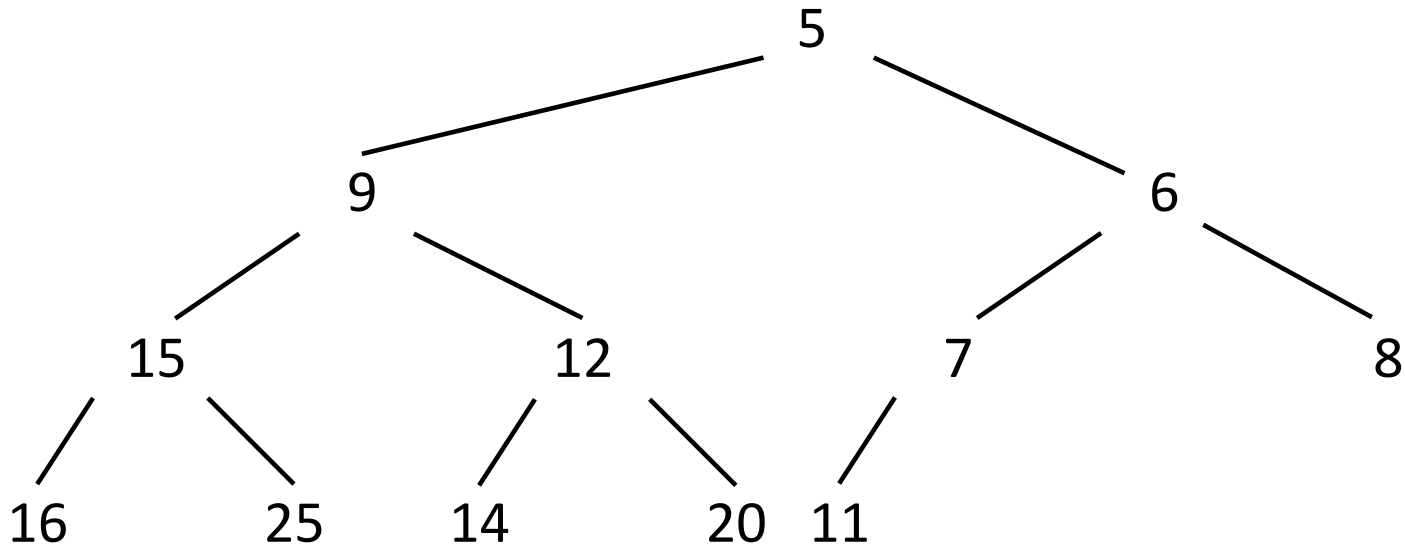
Esempio (2)



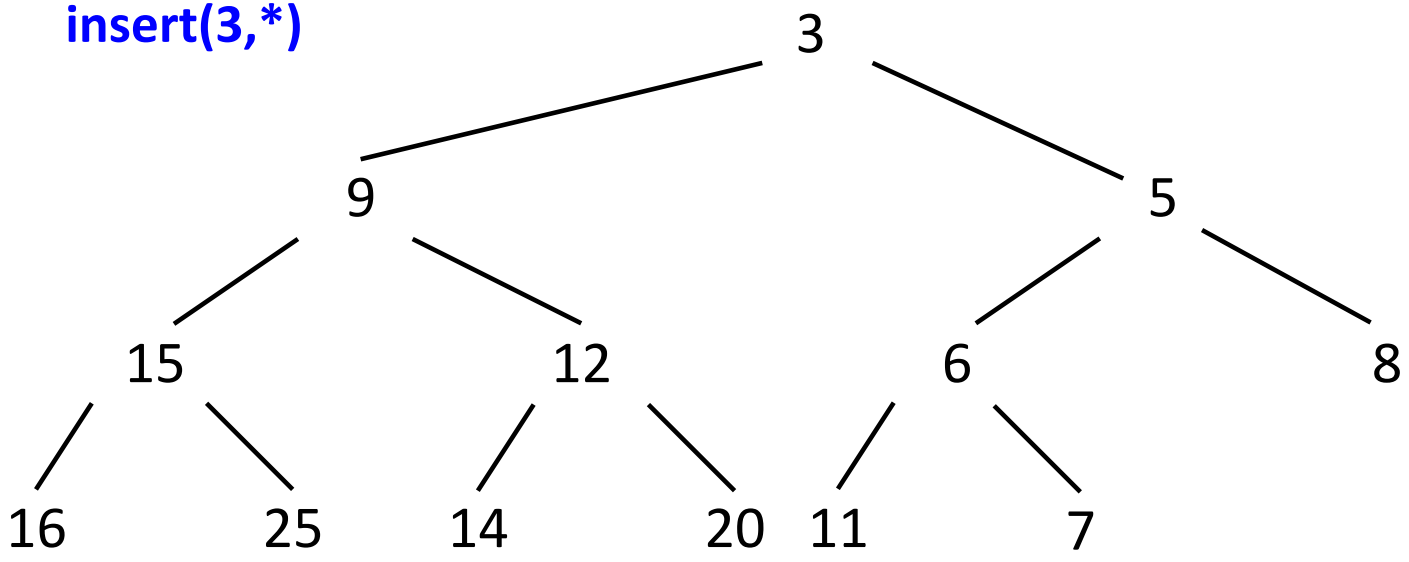
removeMin()



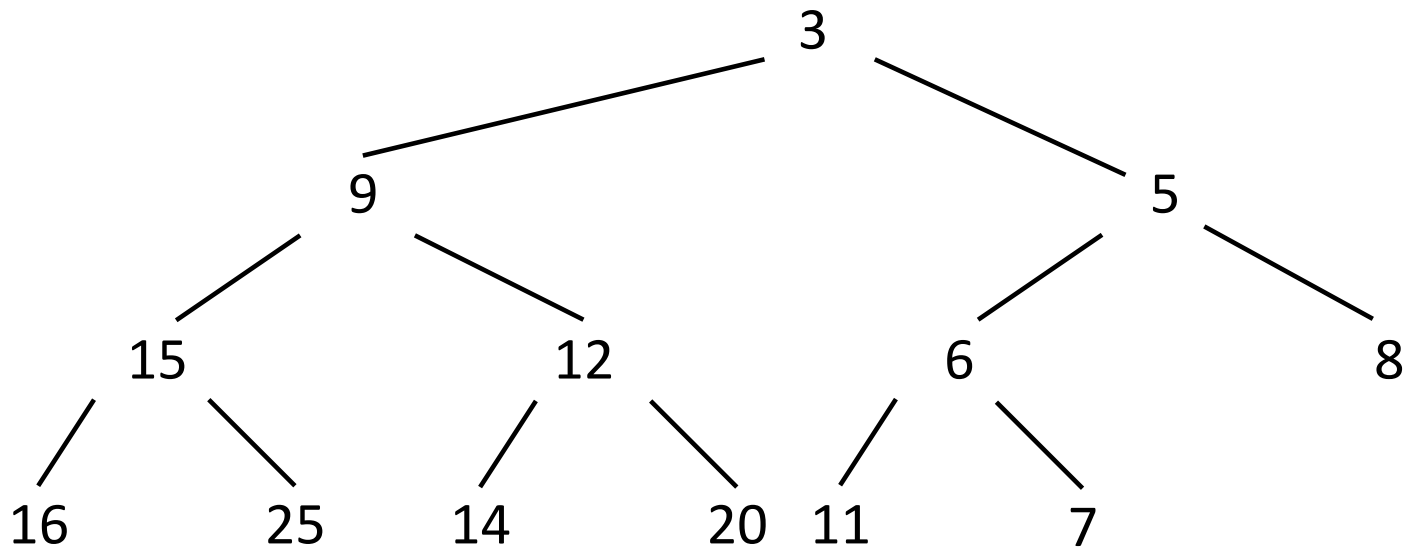
Esempio (3)



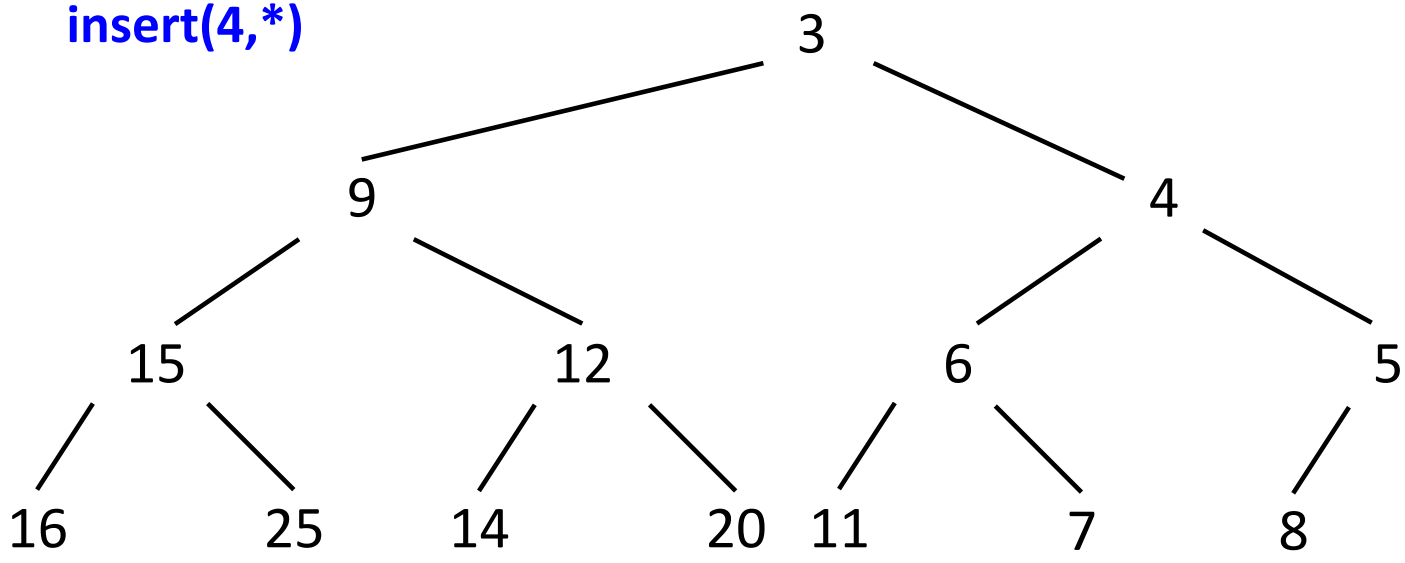
insert(3,*)



Esempio (4)



insert(4,*)



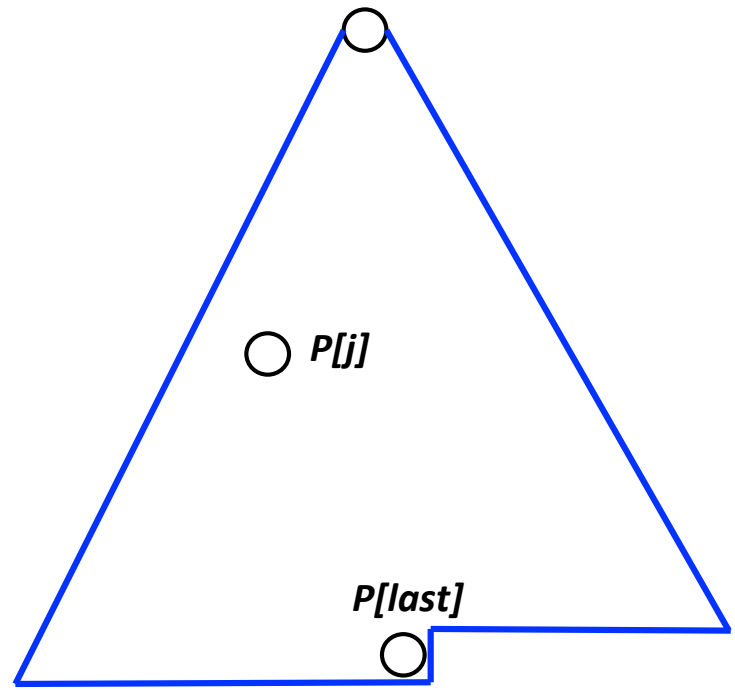
Esercizio

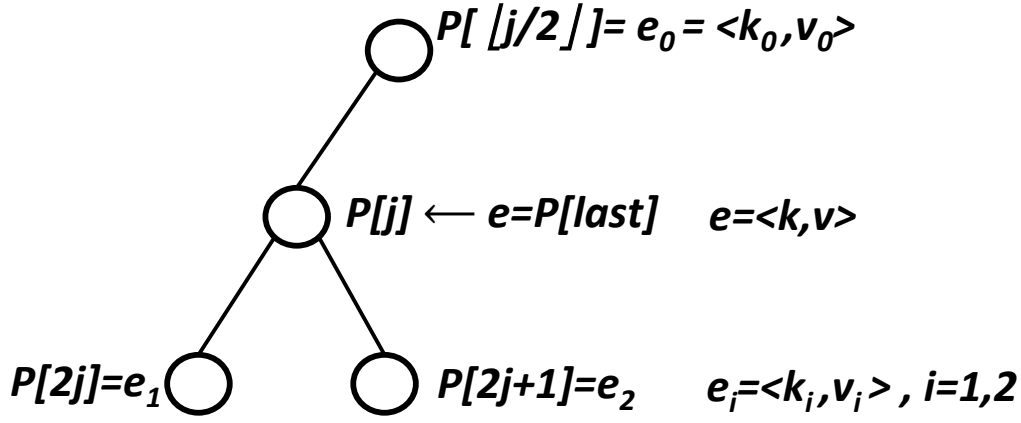
Progettare e analizzare un algoritmo per rimuovere da uno heap P con n entry la entry $P[j]$ ($1 \leq j \leq n$), ripristinando poi le proprietà dello heap.

Svolgimento

Idea:

- $P[j] \leftarrow P[last --]$
- ripristina le proprietà dell'heap





Tre casi:

1. $k_0 \leq k \leq \min\{k_1, k_2\} \Rightarrow$ OK
2. $k < k_0$ $(\Rightarrow k < \min\{k_1, k_2\})$
   $\Rightarrow$ up-heap bubbling a partire da $P[j]$
   N.B. L'unica coppia padre-figlio che viola la heap-order property è $(P[\lfloor j/2 \rfloor], P[j])$.
3. $k > \min\{k_1, k_2\}$ $(\Rightarrow k > k_0)$
   $\Rightarrow$ down-heap bubbling a partire da $P[j]$
   N.B. Le uniche coppie padre-figlio che possono violare la heap-order property sono $(P[j], P[\ell])$ con $\ell = 2j, 2j+1$.

Algoritmo `removeEntry(`P`,`j`)`
Input: Heap P con n= last entry, $j : 1 \leq j \leq n$
Output: Heap P con $n-1$ entry ottenuto rimuovendo $P[j]$
$P[j] \leftarrow P[last--]$;
/* *Caso2: up-heap bubbling a partire da* $P[j]$ */
while $j > 1$ *and* $P[j]$*.getKey()*$< P[\lfloor j/2 \rfloor]$*.getKey()* **do**
    swap ($P[j]$, $P[\lfloor j/2 \rfloor]$);
    $j \leftarrow \lfloor j/2 \rfloor$;
if $2j \leq last$ **then** $i \leftarrow$ indice figlio di $P[j]$ con chiave minima;
/* *Caso 3: down-heap bubbling a partire da* $P[j]$ */
while $2j \leq last$ *and* $P[j]$*.getKey()*$> P[i]$*.getKey()* **do**
    swap ($P[j]$, $P[i]$);
    $j \leftarrow i$;
    if $2j \leq last$ **then** $i \leftarrow$ indice figlio di $P[j]$ con chiave minima;

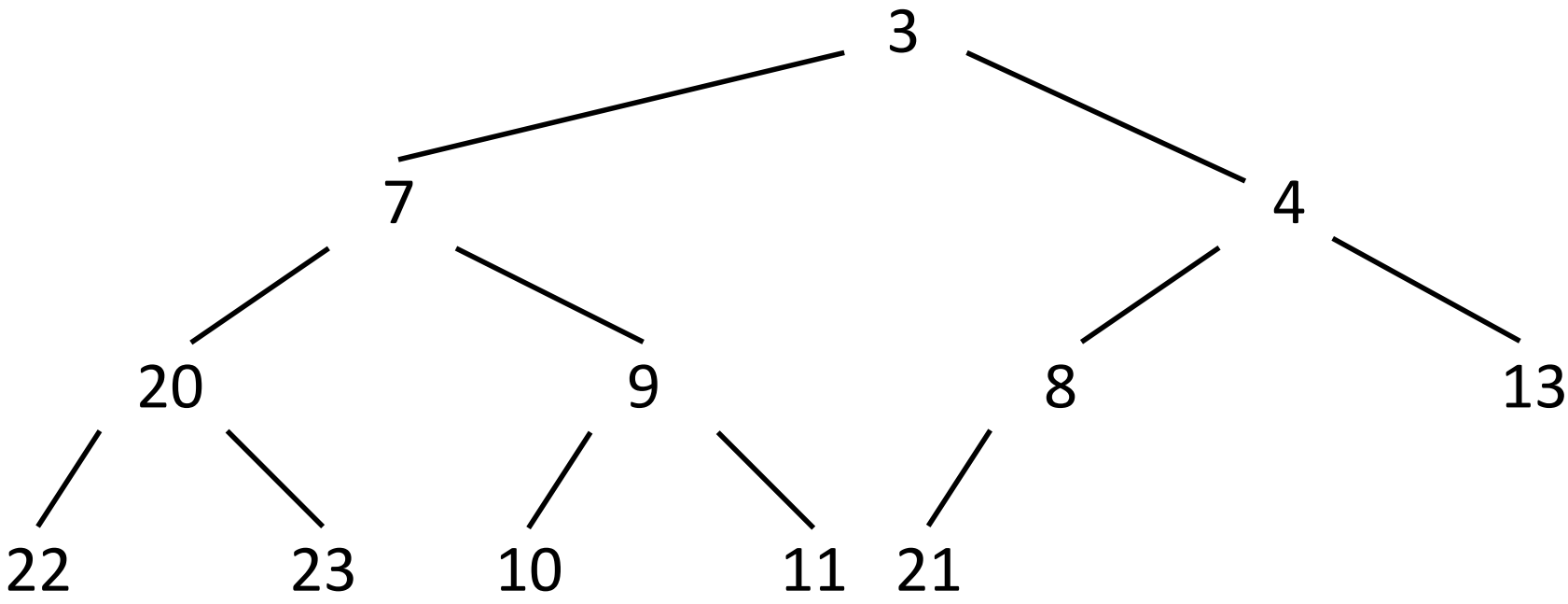
Osservazione: il Caso 1 si ha quando nessuno dei due cicli while viene eseguito

Complessità:

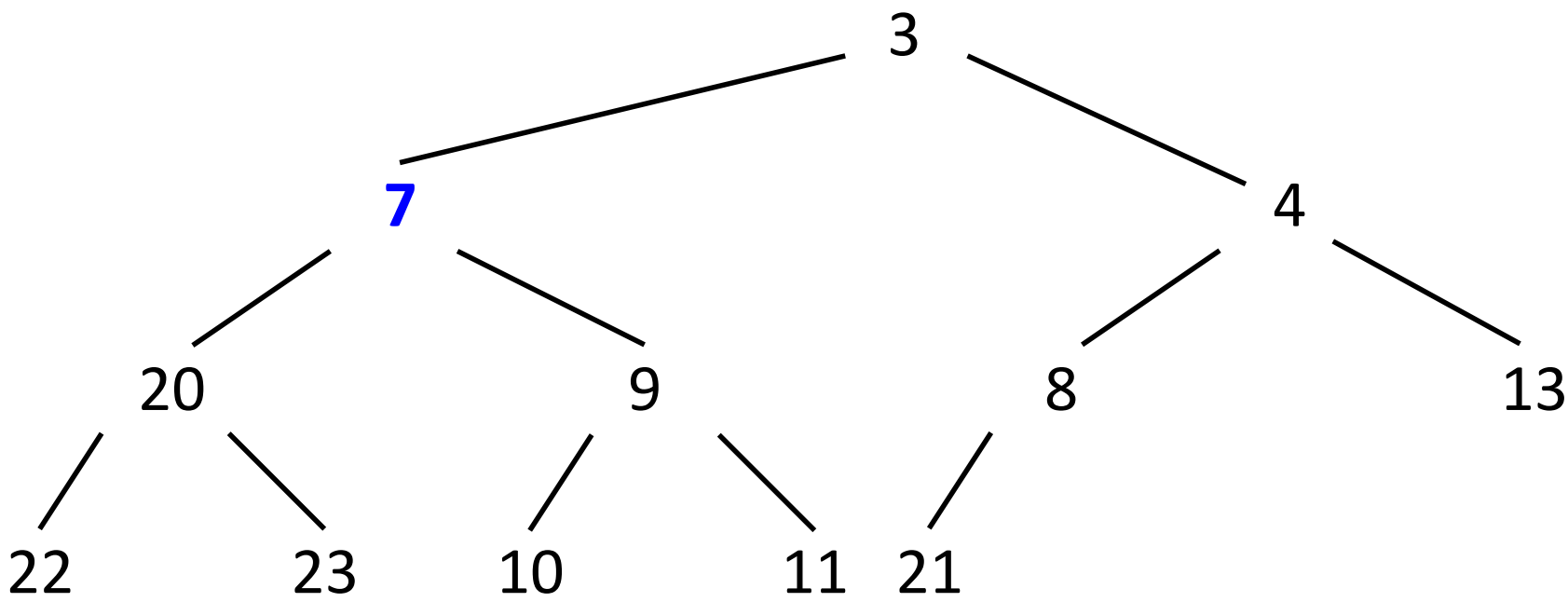
- Ripetendo l'analisi dei metodi `insert` e `removeMin` si vede che ciascun ciclo while esegue $O(\log n)$ iterazioni.
- Esiste un'istanza per la quale il secondo ciclo while esegue $\Omega(\log n)$ iterazioni ($j = 1$ e P[last] che contiene la entry con chiave massima)

$\Rightarrow$ Complessità:$\Theta(\log n)$

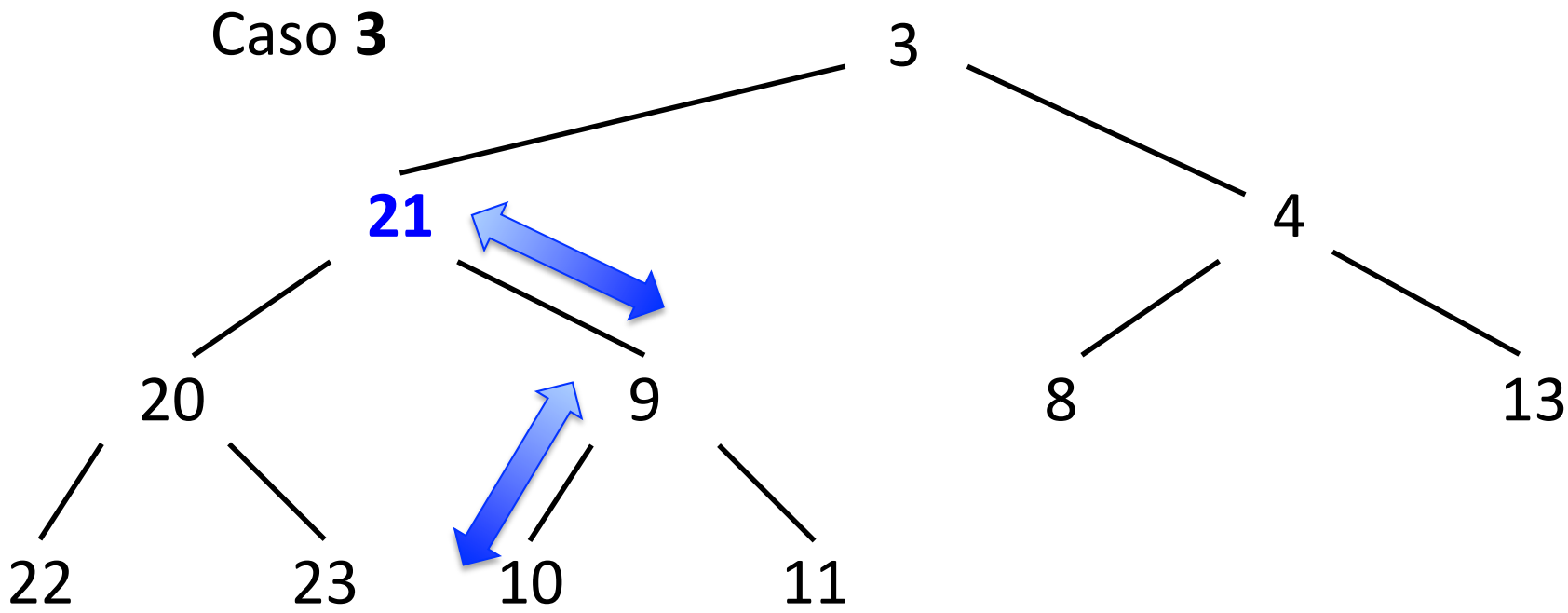
Esempio (solo chiavi)

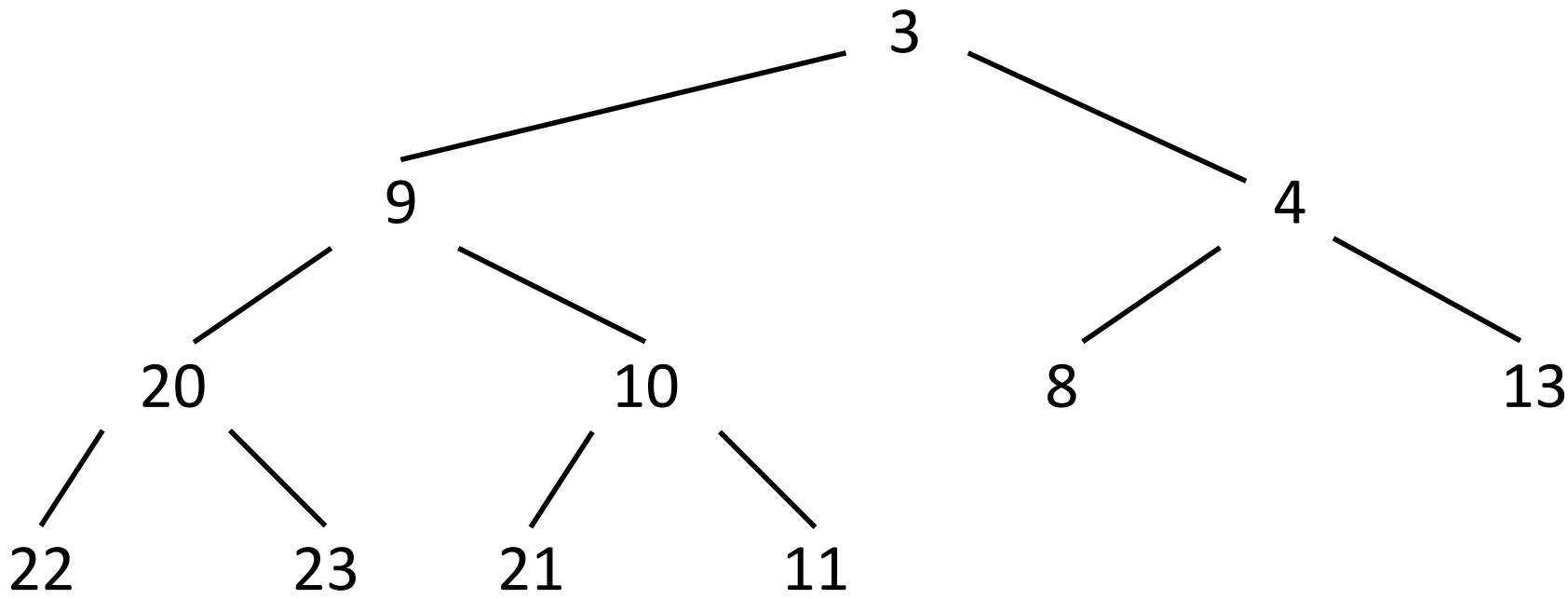


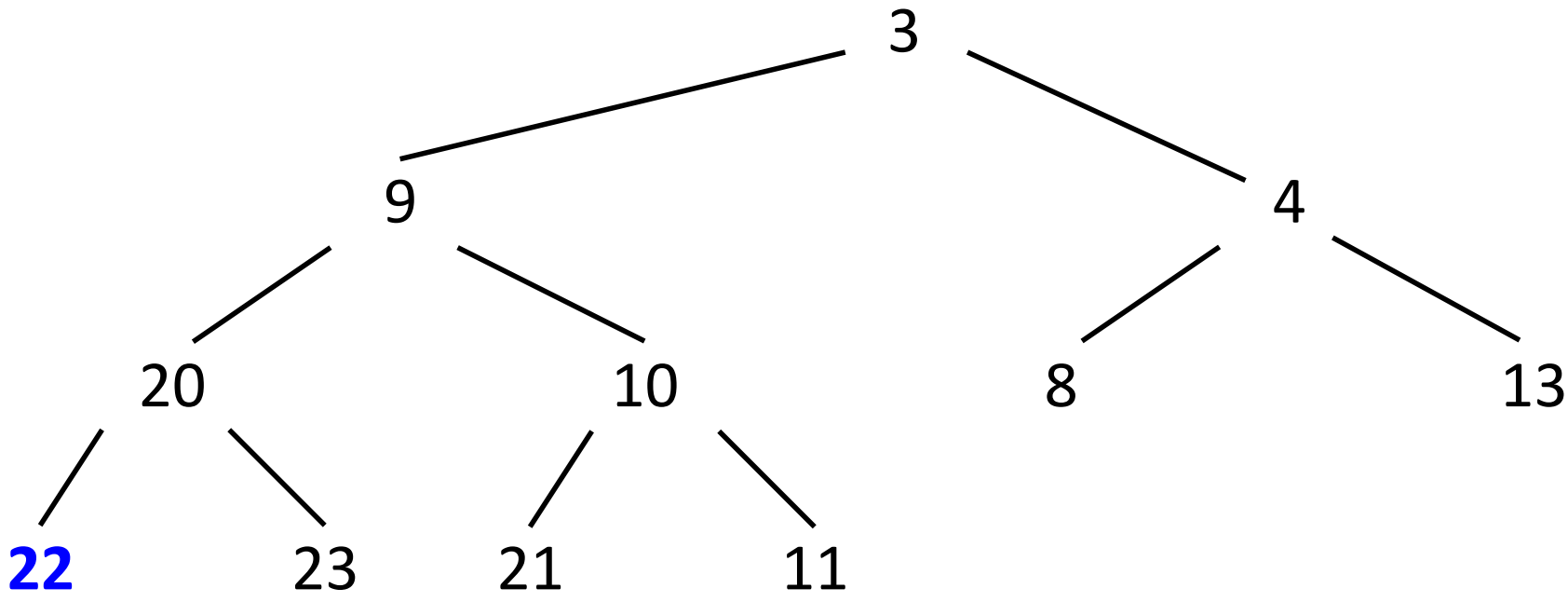
Rimuovi ***P[2]***



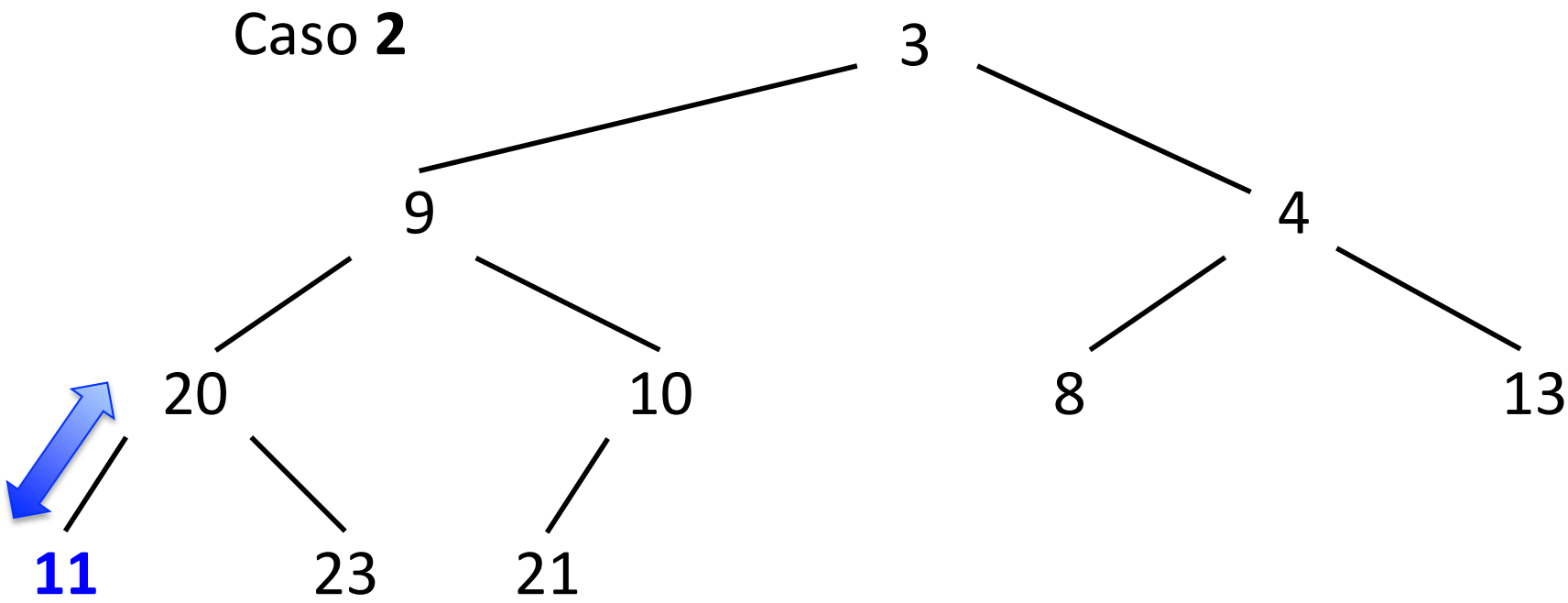
Rimuovi ***P[2]***

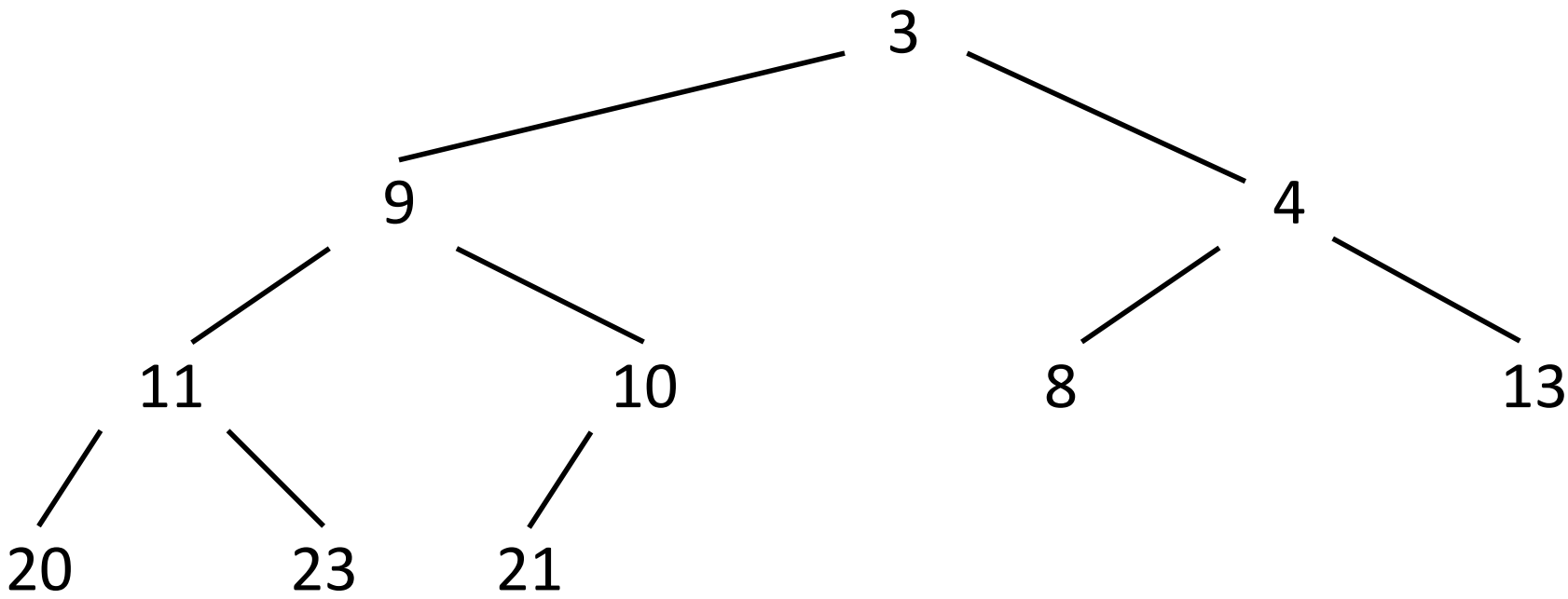

Caso **3**
3
21
4
20
9
8
13
22
23
10
11


3
9
4
20
10
8
13
22
23
21
11



Rimuovi ***P[8]***


Caso **2**
3
9
4
20
10
8
13
11
23
21


3
9
4
11
10
8
13
20
23
21

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.

Svolgimento

Idea:

1. si rimuove una foglia v da T_1;
2. si crea un nuovo radice r contenente la entry e precedentemente contenuta in v, assegnandole T_1 e T_2 come figli sx e dx, rispettivamente;
3. si esegue il down-heap bubbling a partire da r.

La scrittura dello pseudocodice e l'analisi sono lasciati come esercizio.

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).

Svolgimento

Soluzione Banale

Invocare `removeMin()` sino a quando $T[1]$.getKey() $\leq k$.

Se m è il numero di entry stampate, la complessità è $O(m \log n)$. Non è difficile trovare un'istanza per cui la complessità di questa strategia sia $\Theta(m \log n)$. Inoltre T viene modificato, cosa che in alcune applicazioni potrebbe non essere permessa.

Possiamo fare meglio?

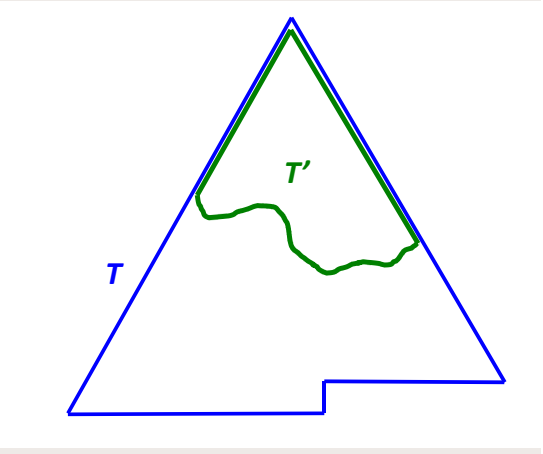
Soluzione migliore

Idea:

Le entry con chiave $\leq k$ formano un sottoalbero T' (eventualmente vuoto) radicato nella radice e senza "buchi"

Le foglie di T' sono i nodi che hanno entry con chiave $\leq k$ ma i cui figli hanno entri con chiave $> k$.

$\Rightarrow$ visita in preorder di T' senza modificare T' (e quindi T)



Soluzione migliore (continua)

Algoritmo `Print`(T,i,k):

Input: Heap T, indice $i \in [1, last]$, chiave k

Output: stampa le entry con chiave $\leq k$ nel sottoalbero con radice $T[i]$

if $T[i].getKey() \leq k$ **then** stampa $T[i]$;

if $2i \leq last$ *and* $T[2i].getKey() \leq k$ **then** `Print`(T,$2i$,k);

if $2i+1 \leq last$ *and* $T[2i+1].getKey() \leq k$ **then** `Print`(T,$2i+1$,k);

Prima invocazione: $i = 1$

Complessità: come la visita in preorder di $T' \Rightarrow \Theta(m+1)$

Esercizio

Sia P uno heap su array contenente n entry con chiavi distinte che denotiamo con $e_1, e_2, \ldots, e_n$ numerandole in ordine di chiave (e_1 è la entry con chiave più piccola). Progettare e analizzare un algoritmo efficiente che dato un intero $m \in [1, n]$ determini e_m senza modificare P.

Esercizio

Sia S una sequenza di n entry, $(S[1], S[2], \ldots, S[n])$, con chiavi intere distinte, e sia τ un intero in $[1, n]$. (Ad es., le entry potrebbero rappresentare film e le chiavi i loro rating.) Il seguente algoritmo trova le *Top-*τ entry di S, ovvero quelle con chiave maggiore, eseguendo un'unica scansione della sequenza.

Algoritmo `Top`(S,τ)
Input: Sequenza S di n entry, intero $\tau \leq n$
Output: τ entry di S con chiave più grande

```
Q ⟵ priority queue vuota;
for i ← 1 to n do
    if (i ≤ τ) then Q.insert(S[i].getKey(),S[i].getValue());
    else
        if S[i].getKey() > Q.min().getKey() then
            Q.removeMin();
            Q.insert(S[i].getKey(),S[i].getValue());
return Q
```

Esercizio (Continua)

1. Analizzare la complessità dell'algoritmo in funzione di n e di τ, assumendo di implementare Q tramite uno heap su array.
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 analizzato nel precedente esercizio può essere utilizzato 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). In questi casi l'algoritmo trova le τ entry con chiave maggiore con spazio porporzionale a τ.

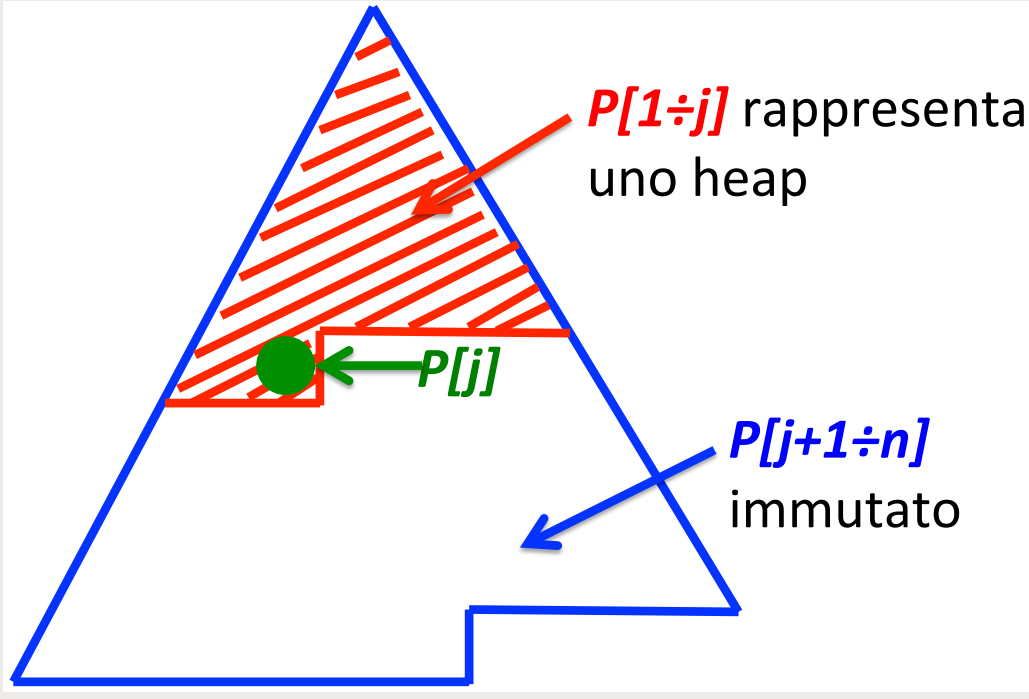
Costruzione di uno heap a partire da n *entry date*

Input. Array P con n entry $P[1 \div n]$
Output. Array P riorganizzato per rappresentare uno heap.

Soluzione 1 (approccio top-down)

Idea: Si eseguono $n-1$ iterazioni successive manenendo il seguente invariante alla fine di ciascuna iterazione j, con $2 \leq j \leq n$

Invariante (alla fine della iterazione j)



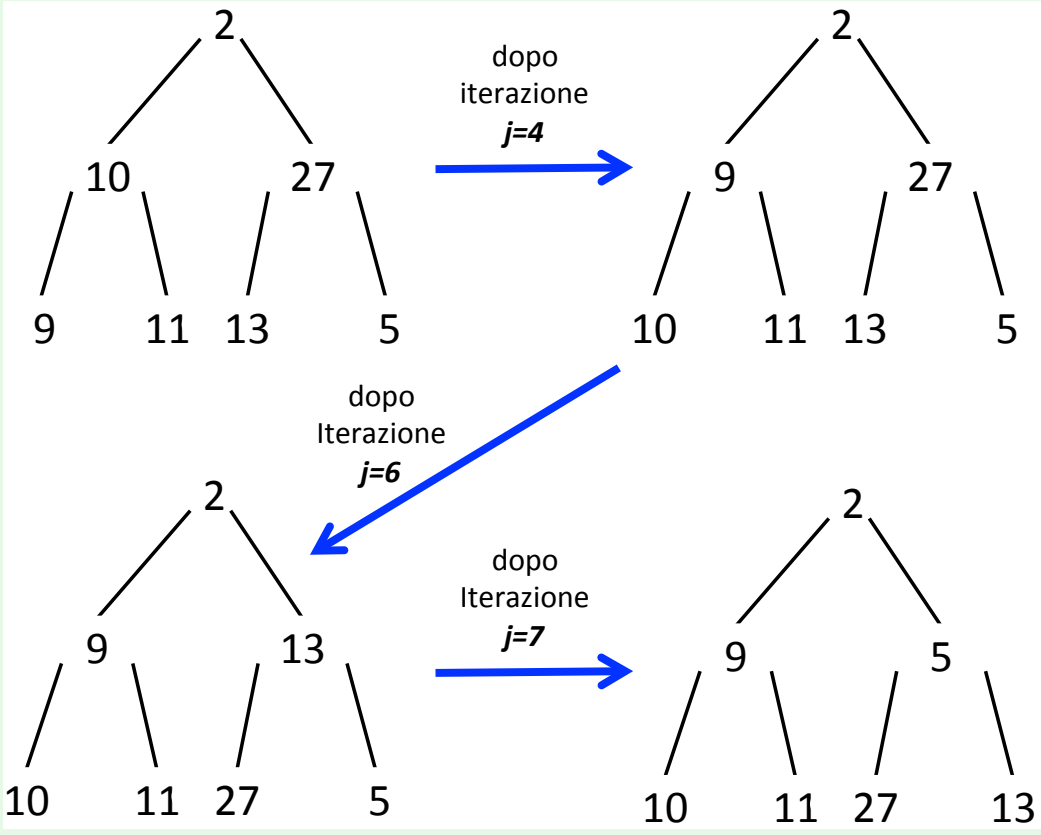
Soluzione 1 (approccio top-down)

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

for $j \leftarrow 2$ *to* n **do**
    // *Up-heap bubbling a partire da* $P[j]$
    $i \leftarrow j$;
    while
     ((*i* > 1)AND(*P*[⌊*i*/2⌋].getKey()> *P*[*i*].getKey())) **do**
        swap(*P*[*i*], *P*[⌊*i*/2⌋]);
        $i \leftarrow \lfloor i/2 \rfloor$;

Esempio

Esempio: $P = 2\ 10\ 27\ 9\ 11\ 13\ 5$



Complessità dell'approccio top-down

Dimostriamo che la complessità è $\Theta\left(\sum_{j=2}^{n} \log j\right)$:

- $O\left(\sum_{j=2}^{n} \log j\right)$: banale
- $\Omega\left(\sum_{j=2}^{n} \log j\right)$: si consideri come istanza "cattiva" quella in cui P è inizialmente ordinato in senso decrescente

Claim

$\sum_{j=2}^{n} \log j \in \Theta(n \log n)$

Dimostrazione

Si osservi che $\sum_{j=2}^{n} \log j = \sum_{j=1}^{n} \log j$. Il claim segue dalla relazione

$$\frac{n}{2} \log_2 \left\lceil \frac{n}{2} \right\rceil \leq \sum_{j=\lceil n/2 \rceil}^{n} \log_2 j \leq \sum_{j=1}^{n} \log_2 j \leq n \log_2 n$$

□

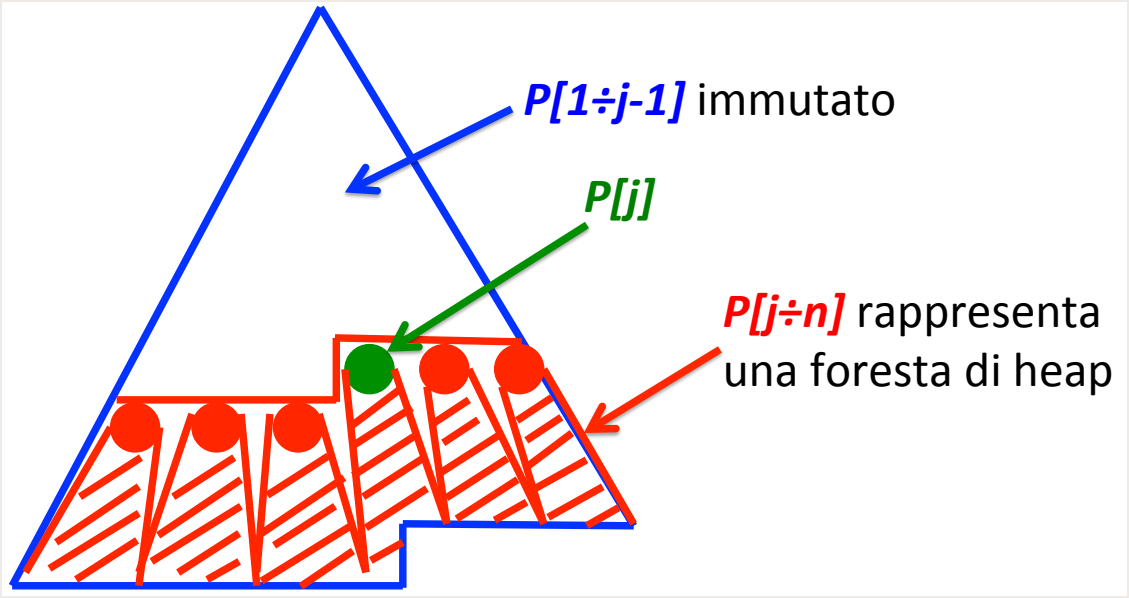
Osservazione

Per provare il claim, in alternativa si poteva usare il fatto che $\sum_{j=1}^{n} \log_2 j = \log_2 (n!)$, e l'approssimazione di Stirling per cui $n! = \sqrt{2\pi n}(n/e)^n(1 + O(1/n))$

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 $\lfloor n/2 \rfloor \geq j \geq 1$.

Invariante (alla fine della iterazione j)



Soluzione 2 (approccio bottom-up)

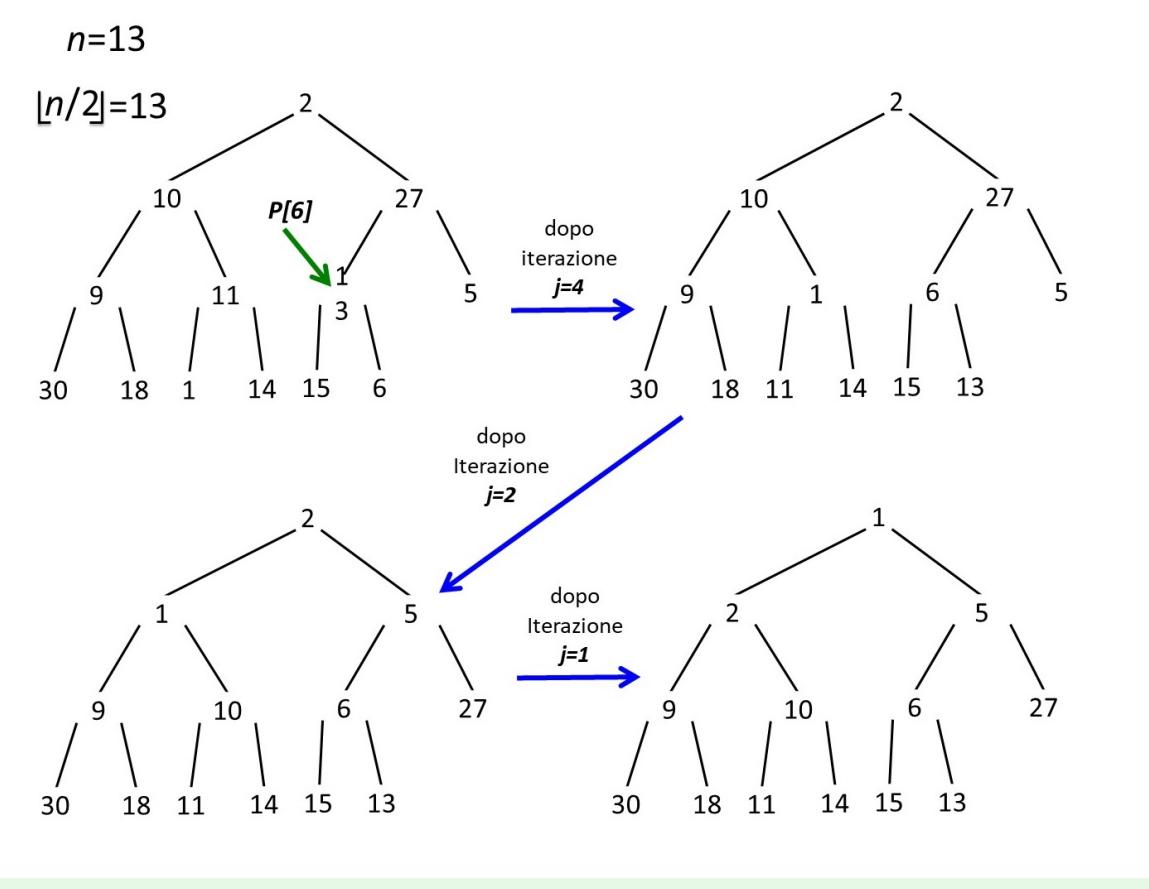
Implementazione: basta eseguire un down-heap bubbling da $P[j]$ in ciascuna iterazione j

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

for $j \leftarrow \lfloor n/2 \rfloor$ downto 1 **do**
    // *Down-heap bubbling a partire da* $P[j]$
    $i \leftarrow j$;
    if $2i \leq last$ **then**
        $k \leftarrow$ indice del figlio di $P[i]$ con chiave min;
        // $k = 2i$ o $k = 2i + 1$
    while $(2i \leq last)$`AND(`$P[i]$`.getKey()>` $P[k]$`.getKey())`
     do
        swap($P[i]$,$P[k]$);
        $i \leftarrow k$;
        if $2i \leq last$ **then** $k \leftarrow$ indice del figlio di $P[i]$ con chiave min;

Esempio

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



Complessità dell'approccio bottom-up

(C-9.35 [GTG14])

- $t_{P[j]} \equiv$ 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:

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

Dimostreremo ora (si veda il Claim 2 più avanti) 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])

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

Dimostrazione

Iniziamo limitando superiormente ciascun addendo con un termine di una progressione geometrica. Dimostriamo per induzione che $(\ell/2^{\ell}) \leq (3/4)^{\ell}$, per ogni $\ell \geq 1$.

- Base: $\ell = 1, 2$: verifica immediata
- Passo induttivo. Fissiamo $\ell \geq 2$ e supponiamo (hp. induttiva) che $(i/2^{i}) \leq (3/4)^{i}$, per ogni $1 \leq i \leq \ell$. Si ha che

$$\begin{aligned} \frac{\ell+1}{2^{\ell+1}} &= \frac{\ell}{2^{\ell}} \frac{1}{2} \frac{\ell+1}{\ell} \\ &\leq \left(\frac{3}{4}\right)^{\ell} \frac{\ell+1}{2\ell} \quad \text{(hp. ind.)} \\ &\leq \left(\frac{3}{4}\right)^{\ell+1}. \end{aligned}$$

Dimostrazione (continua)

L'ultimo passaggio segue dal fatto che

$$\frac{\ell+1}{2\ell} \leq \frac{3}{4} \Leftrightarrow 4\ell+4 \leq 6\ell \Leftrightarrow 2\ell \geq 4 \Leftrightarrow \ell \geq 2.$$

A questo punto la prova del claim si conclude ossevando che

$$\sum_{\ell=1}^{h} \ell \left(\frac{1}{2}\right)^{\ell} \leq \sum_{\ell=1}^{h} \left(\frac{3}{4}\right)^{\ell} = \frac{3/4-(3/4)^{h+1}}{1-3/4} < 3$$

□

Claim 2

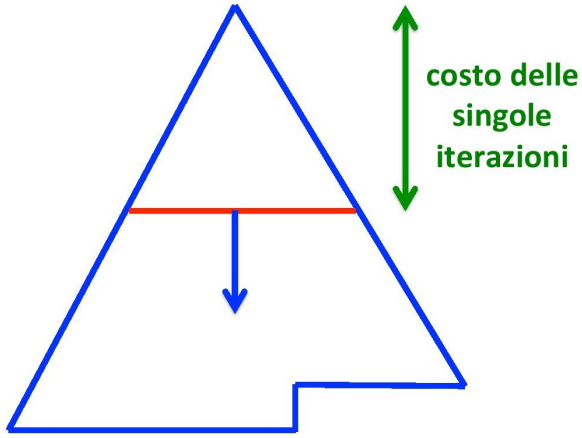
$\sum_{i=0}^{h-1} 2^i (h-i) \in O(n)$

Dimostrazione

$$\begin{aligned}
\sum_{i=0}^{h-1} 2^i (h-i) &= 2^h \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}} \\
&= 2^h \sum_{\ell=1}^{h} \ell \left(\frac{1}{2}\right)^{\ell} \\
&\in O(n),
\end{aligned}$$

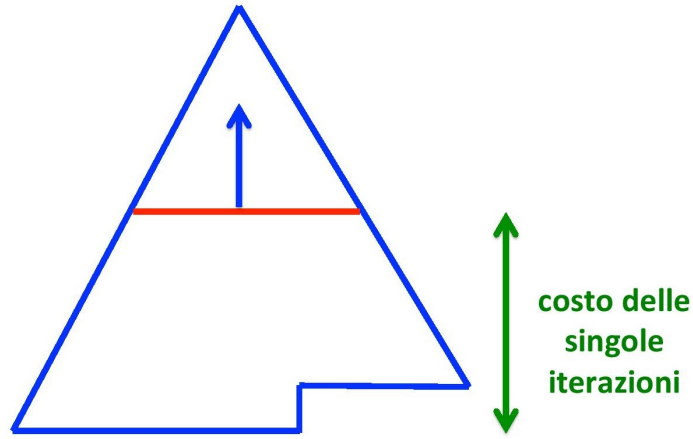
dove l'ultima relazione discende dal Claim 1.

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

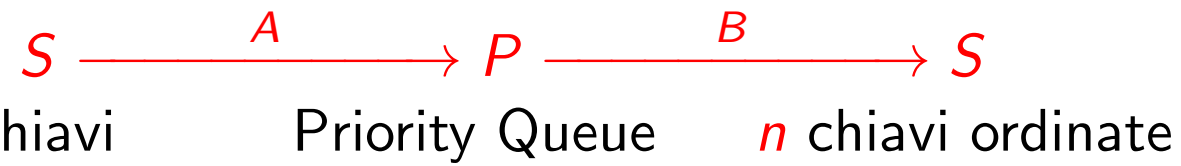
- 2^i operazioni di costo $\Theta((\log n) - i)$
- più aumenta la taglia del livello e più diminuisce il costo del down-heap bubbling
- In tutto: $\Theta(n)$

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 `PriorityQueueSort(`S`)`



Fase A: si inseriscono le n chiavi in P una alla volta invocando il metodo **insert** e incapsulando ogni chiave k in una entry (k, null).

Fase B: si rimuovono le n entry da P una alla volta invocando il metodo **removeMin** ed estraendo da ogni entry (k, null) la chiave k.

Complessità di `PriorityQueueSort(`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 $\Theta(n)$

Definizione

Algoritmo IN-PLACE: usa $O(1)$ memoria aggiuntiva oltre a quella necessaria per l'input

Nota

InsertionSort e *SelectionSort* possono essere implementati *in-place* in modo semplice. Anche una variante di `HeapSort` (coming soon...)

HeapSort in-place (Paragrafo 9.4.2 [GTG14])

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(1)

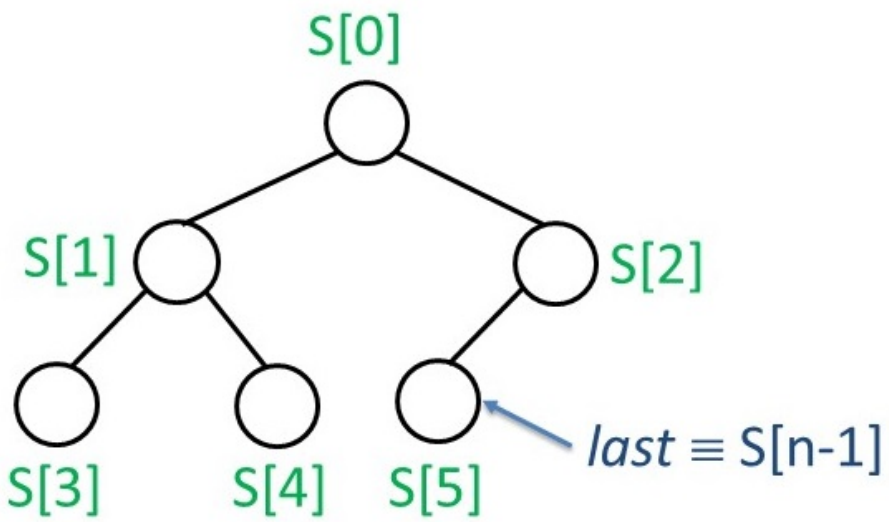
Fase B: riorganizza $S[0 \div n-1]$ in modo che le chiavi risultino ordinate

(1) Un max heap è uno heap con la heap-order property ribaltata: *la chiave in un nodo interno è $\geq$ della massima chiave dei figli*

Fase A: $S \rightarrow$ max heap

Vediamo anzitutto S come un albero binario completo usando il seguente mapping (level numbering che parte da 0):

- radice $\rightarrow S[0]$
- figli di $S[i] \rightarrow S[2i+1], S[2i+2]$
- padre di $S[i] \rightarrow S[\lfloor (i-1)/2 \rfloor]$



Fase A: $S \rightarrow$ max heap

Il level numbering che parte da 0 genera la seguente distribuzione di indici tra i livelli:

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	$2^h - 1 \div n - 1$

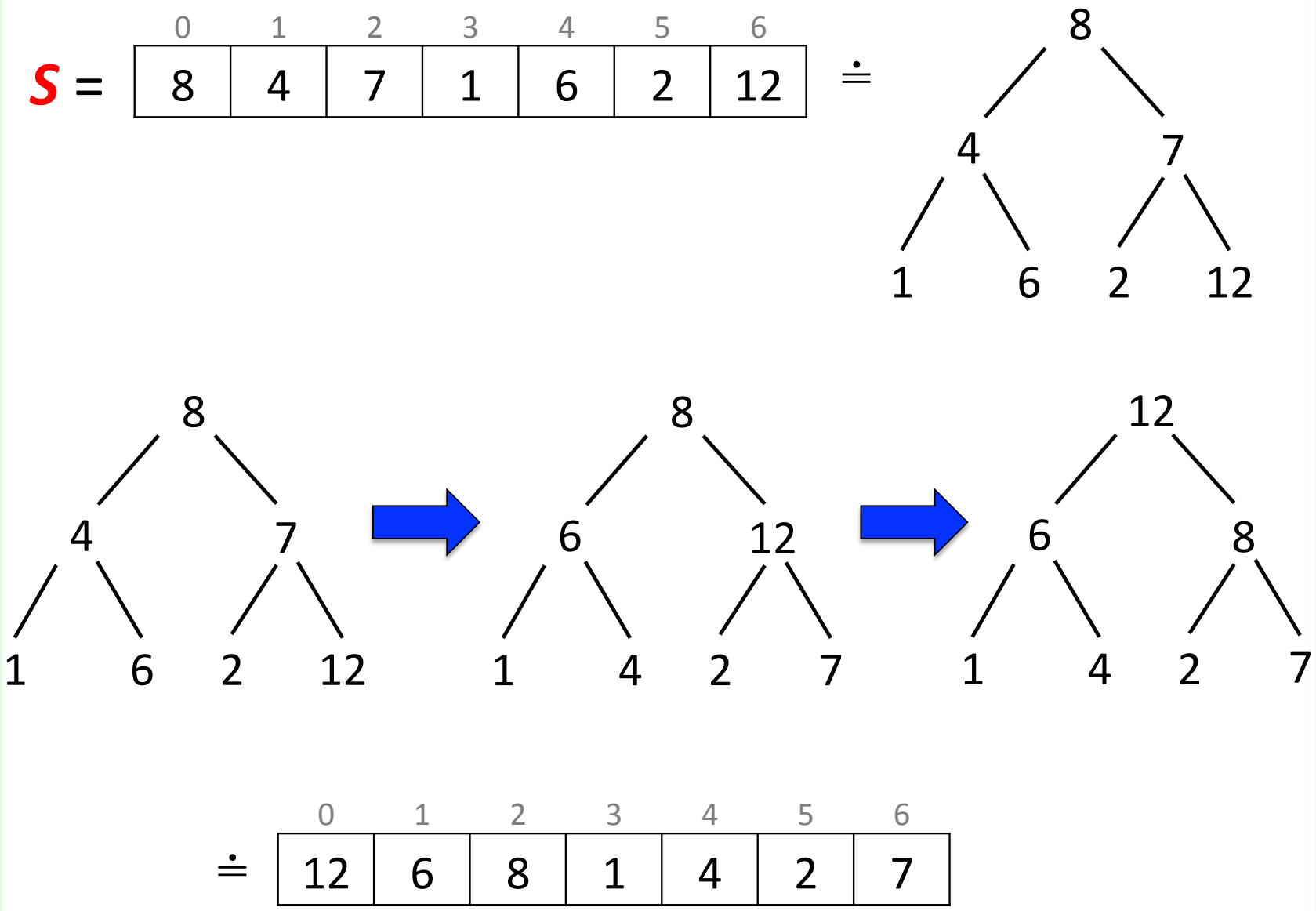
e il nodo *last* corrisponde all'indice $n - 1$.

Fase A: $S \rightarrow$ max heap

La **Fase A** deve trasformare S in un max heap. L'implementazione è la seguente:

for $j \leftarrow \lfloor \frac{n-2}{2} \rfloor$ **downto** 0 **do**
    /* $S\left[\lfloor \frac{n-2}{2} \rfloor\right]$=padre di $last$ ($S[n-1]$)) */
    $i \leftarrow j$;
    /* Down-heap bubbling di $S[j]$ */
    $k \leftarrow$ indice del figlio di $S[i]$ con chiave max;
    while $(2i+1 \leq n-1$ AND $S[i] < S[k])$ **do**
        swap($S[i]$,$S[k]$);
        $i \leftarrow k$;
        if $2i+1 \leq n-1$ **then**
            $k \leftarrow$ indice del figlio di $S[i]$ con chiave max;

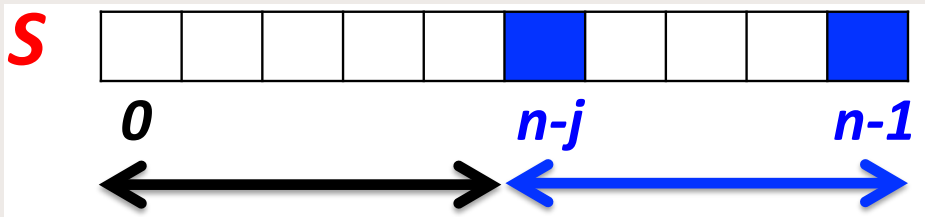
Esempio


S =
0 1 2 3 4 5 6
8 4 7 1 6 2 12
≐
8 4 7 1 6 2 12
8 4 7 1 6 2 12
8 6 12 1 4 2 7
12 6 8 1 4 2 7
≐
0 1 2 3 4 5 6
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:

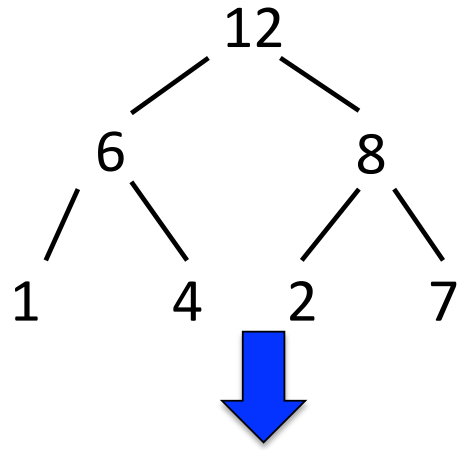
```
for j ← 1 to n − 1 do
    swap(S[n − j],S[0]);
    last ← n − j − 1;
    /* Down-heap bubbling di S[0]                                   */
    i ← 0;
    if 2i + 1 ≤ last then k ← indice del figlio di S[i] con chiave max;
    while 2i + 1 ≤ last and S[i] < S[k] do
        swap(S[i],S[k]);
        i ← k;
        if 2i + 1 ≤ last then
            k ← indice del figlio di S[i] con chiave max;
```

Esercizio

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

Esempio: dopo costruzione bottom-up

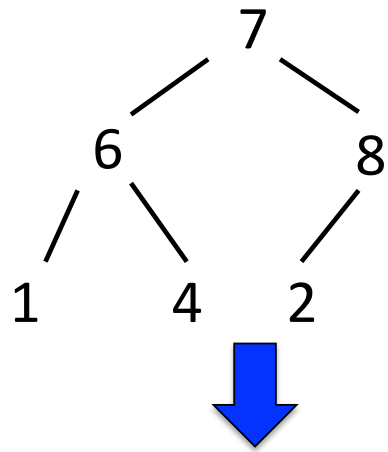
0	1	2	3	4	5	6
12	6	8	1	4	2	7

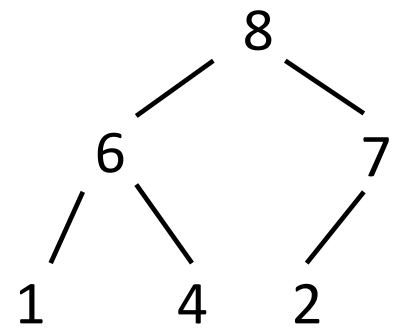


Iterazione 1:

- 12 ➔ cella 6 (***n-1***)
- 7 ➔ cella 0
- down-heap bubbling

0	1	2	3	4	5	6
8	6	7	1	4	2	12

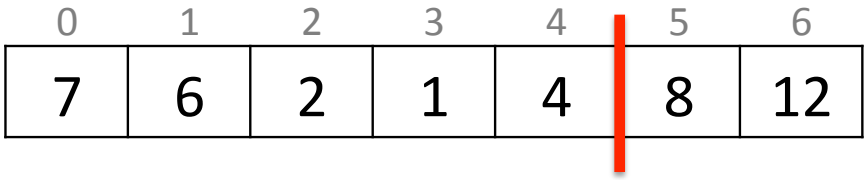


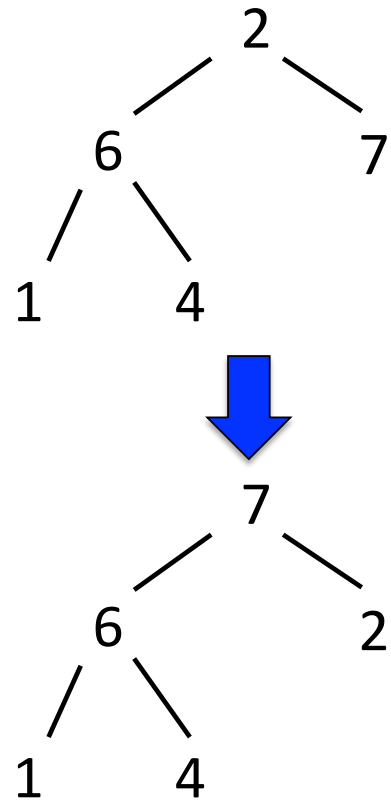


Esempio: dopo costruzione bottom-up (continua)

Iterazione 2:

- 8 ➔ cella 5 (***n-2***)
- 2 ➔ cella 0

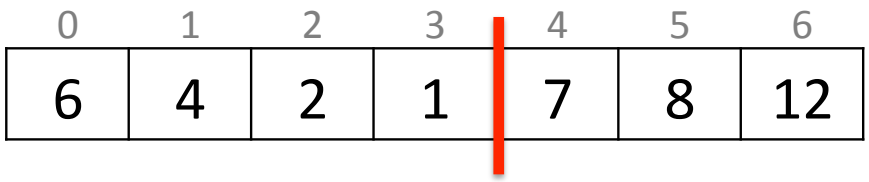


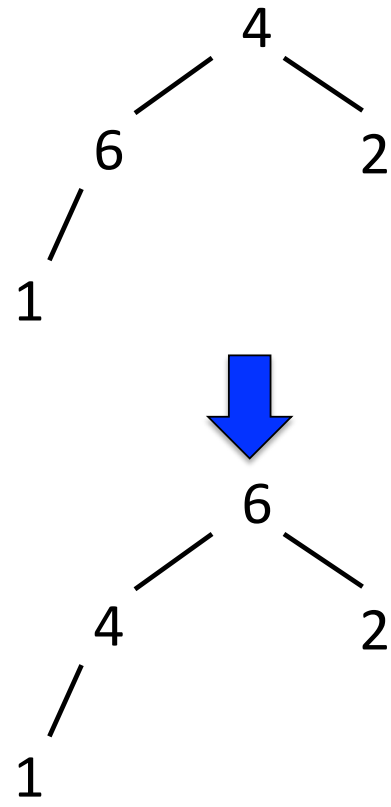


Esempio: dopo costruzione bottom-up

Iterazione 3:

- 7 ➔ cella 4 (***n-3***)
- 4 ➔ cella 0



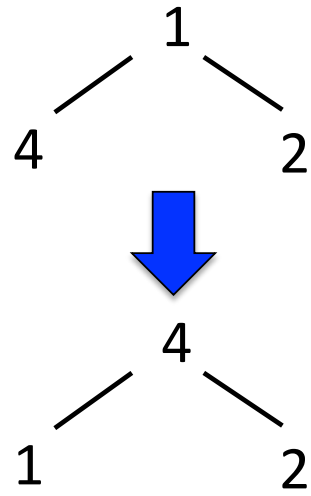


Esempio: dopo costruzione bottom-up (continua)

Iterazione 4:

- 6 ➔ cella 3
- 1 ➔ cella 0

0	1	2	3	4	5	6
4	1	2	6	7	8	12



Iterazione 5:

- 4 ➔ cella 2
- 2 ➔ cella 0

0	1	2	3	4	5	6
2	1	4	6	7	8	12

Iterazione 6:

- 2 ➔ cella 1
- 1 ➔ cella 0

0	1	2	3	4	5	6
1	2	4	6	7	8	12

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 $S = S[1], S[2], \ldots, S[n]$ 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 $O(n)$.)

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 tremite heap (su array).
- Costruzione (top-down e bottom-up) di uno heap partendo da un array di n entry:
- PriorityQueueSort
  - Implementazione con lista non ordinata (*SelectionSort*).
  - Implementazione con lista ordinata (*InsertionSort*).
  - Implementazione con heap su array (*HeapSort*).

Esempio domande prima parte

- Dare la definizione di heap.
- Definire il level numbering indicandone la principale proprietà.
- Descrivere l'implementazione del metodo `removeMin` per uno heap P realizzato tramite array, e analizzarne la complessià.
- Si consideri la strategia bottom-up di costruzione di uno Heap a partire da un array P di n entry, descritta dal seguente ciclo:

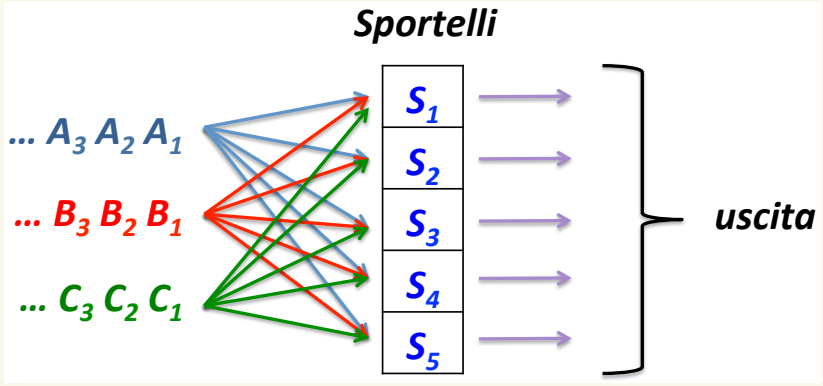
  for $j \leftarrow \lfloor n/2 \rfloor$ **downto** 1 **do** down-heap bubbling a partire da $P[j]$.

  Dimostrare che essa richede tempo $O(n)$ assumendo che per ogni intero h valga che $\sum_{\ell=1}^{h} \ell(1/2)^{\ell} = O(1)$.

PROBLEMA: stimare il *tempo medio di permanenza* (attesa in coda e servizio) degli utenti di un ufficio pubblico.

Parametri del sistema:

- K tipologie di utenti. Ad es., $K = 3 : A, B, C$
- S sportelli multifunzione. Ad es., $S = 5$
- X_A, X_B, X_C: tempi di interarrivo (variabili aleatorie)
- O_A, O_B, O_C: tempi di servizio operativi (variabili aleatorie)
- Ogni utente quando arriva sceglie lo sportello con meno persone



Caso di studio: Simulazione Discreta a Eventi

Eventi simulati:

- A_i, B_i, C_i: arrivo i-esimo utente di tipo A/B/C
- $\bar{A}_i$, $\bar{B}_i$, $\bar{C}_i$: uscita i-esimo utente di tipo A/B/C

Metriche di interesse:

- T_A, T_B, T_C: somma tempi di permanenza di utenti di tipo A/B/C
- m_A, m_B, m_C: numero utenti entrati di tipo A/B/C

Variabili:

- $t(e)$: tempo dell'evento e
- n_j: numero di persone allo sportello j, per $1 \leq j \leq 5$
- α_j: tempo in cui si libera lo sportello j

Caso di studio: Simulazione Discreta a Eventi

Si simulano i vari eventi nell'ordine temporale in cui occorrono, utilizzando una priority queue Q di appoggio.

Inizializzazione di Q: si inseriscono le entry $(t(A_1), A_1)$, $(t(B_1), B_1)$, $(t(C_1), C_1)$, dove i tempi (chiavi delle entry), sono determinati generando tempi di interarrivo con le opportune variabili aleatorie, ovvero: $t(A_1) = X_A, t(B_1) = X_B, t(C_1) = X_C$. Il valore della entry è una descrizione dell'evento.

Generica iterazione di Q: si estrae da Q il prossimo evento e (invocando Q.removeMin()) e lo si simula. In funzione della tipologia di evento, si generano eventuali altri eventi da inserire e si aggiornano le metriche di interesse.

Simulazione di $e = (t(A_i), A_i)$:

arrivo dell'i-esimo utente di tipo A (analogo per le altre tipologie)

- Si incrementa m_A
- Si sceglie lo sportello j e si incrementa n_j
- Si aggiunge a α_j il tempo di servizio dell'utente (stimato con O_A)
- Usando X_A, si determina il tempo di arrivo dell'$(i+1)$-esimo utente di tipo A ($t(A_{i+1}) = t(A_i) + X_A$)
- Si inseriscono in Q le seguenti entry:
  - $(t(A_{i+1}), A_{i+1})$
  - $(t(\bar{A}_i), (\bar{A}_i, j, t(A_i)))$. Questa entry rappresenta l'uscita dell'utente dall'ufficio, che avverrà al tempo $t(\bar{A}_i) = \alpha_j$. Il valore della entry è una tripla: descrizione dell'evento, sportello scelto dall'utente, tempo di ingresso dell'utente.

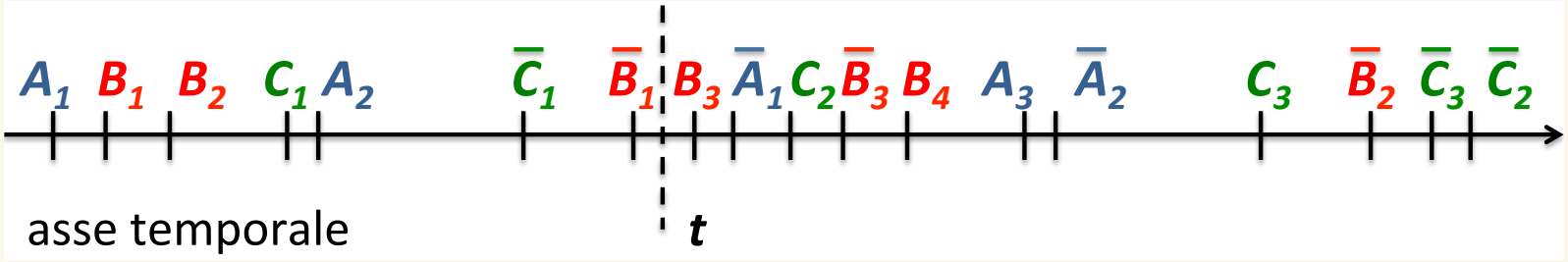
Caso di studio: Simulazione Discreta a Eventi

Simulazione di $e = (t(\bar{A}_i), (\bar{A}_i, j, t(A_i)))$**:**

uscita dell'i*-esimo utente di tipo* A *(analogo per le altre tipologie)*

- Si decrementa n_j
- Si aggiunge $t(\bar{A}_i) - t(A_i)$ a T_A.

Esempio:



Dopo aver simulato tutti gli eventi sino al tempo t, in Q ho le entry relative ai seguenti eventi (in ordine di priorità): B_3 $\bar{A}_1$ C_2 A_3 $\bar{A}_2$ $\bar{B}_2$

Errata

Cambiamenti rispetto alla prima versione dei lucidi:

- Lucidi 7 e 9: Position $\rightarrow$ PositionalList.
- Lucido 13: aggiunta la base del logaritmo nella penultima riga
- Lucido 35: modificato invariante e figura.
- Lucido 44: modificato invariante.
- Lucido 65: corretto il range di τ nelt esto dell'esercizio.
- Lucido 89: modificata notazione nell'if prima del while.
- Lucidi 98-102: aggiunto caso di studio.