Lezione_16_DeA

5.4.5. Implementazione dei metodi della Priority Queue su heap

Notazione

- $P[i] \equiv$ entry, la cui chiave si ottiene con `P[i].getKey()` e il valore `P[i].getValue()`;
- $P[last]$ è la entry più a destra al livello h.

Sia P uno heap con n entry implementato tramite array.

5.4.5.1. `min`

Metodo `min()`

```
return P[0];
```

Complessità: $\Theta(1)$;

5.4.5.2. `insert`

Per realizzare il metodo insert inseriamo la nuova entry come successore (nel level numbering) del nodo last, poi ricostruiamo la heap-order property dello heap lungo il cammino dal nodo last alla radice.

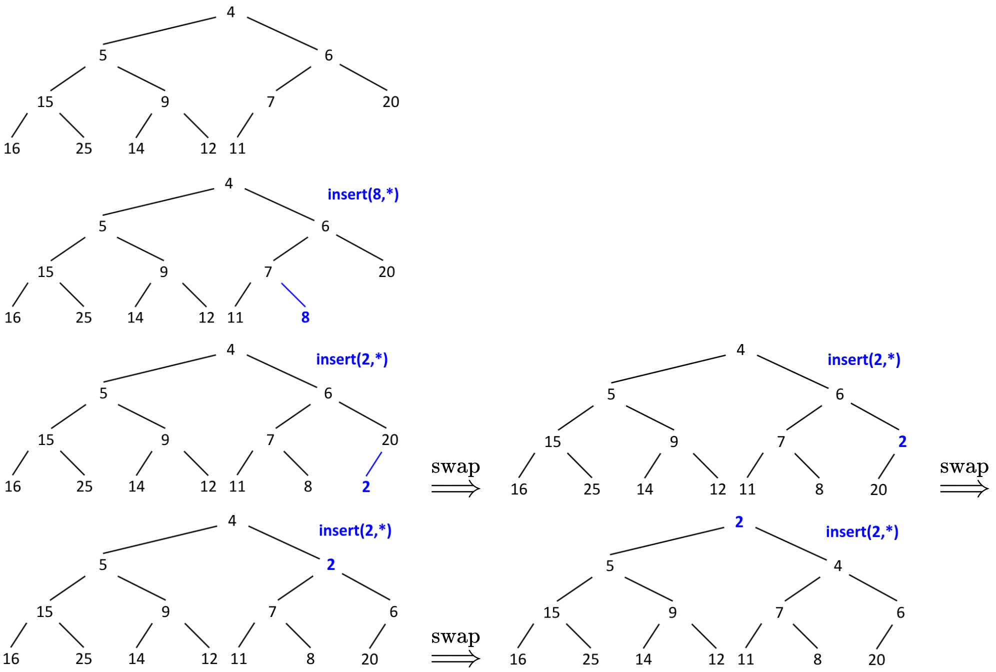
Metodo `insert(k,x)`

```
e <- (k,x);
P[++last] <- e;
i <- last;
//Up-heap bubbling
while ((i > 0) AND (P[ \floor{(i-1)/2} ].getKey() > P[i].getKey())) do {
     swap(P[i], P[ \floor{(i-1)/2} ]);
     i <- \floor{(i-1)/2};
}
return e;
```

Osservazione

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

5.4.5.2.1. Esempio (solo chiavi)

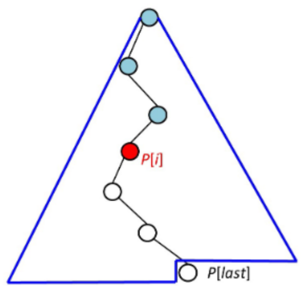


5.4.5.2.2. Correttezza

> **Ricorda - Heap-oder property estesa**
>
> In uno heap, la chiave di un nodo deve essere maggiore o uguale di quella di un qualsiasi suo antenato.

La correttezza di `insert` discende dalle seguenti proprietà:

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



5.4.5.2.3. Complessità

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.

- La complessità è proporzionale al numero di iterazioni del while, dato che ogni iterazione richiede $\Theta(1)$ operazioni;
- Il numero di iterazioni del while è $\leq h$;
- Esiste un'istanza che richiede esattamente h iterazioni (quella in cui si inserisce una entry con chiave minore di tutte quelle presenti).
  $\Longrightarrow$ Complessità $\in \Theta(\log n)$.

> **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`, quindi può essere nascosto (*ammortizzato*) da quest'ultimo.

5.4.5.3. `removeMin`

Per realizzare `removeMin` rimuoviamo la entry presente nella radice dello heap, mettiamo la entry $P[\text{last}]$ nella radice, poi ricostruiamo la heap-order property dello heap a partire dalla radice verso le foglie.

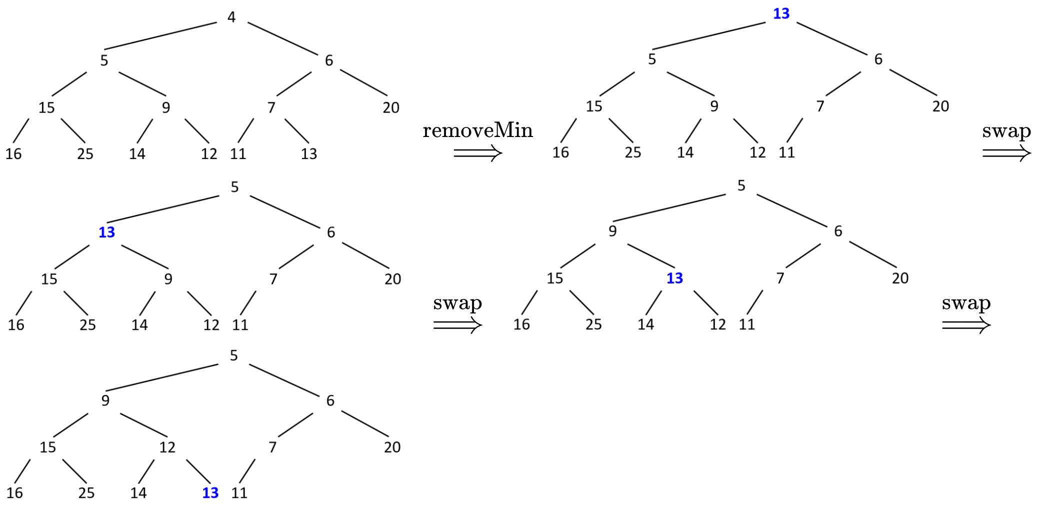
Sia `indexMinChild(P,i)` un metodo che restituisce l'indice del figlio di $P[i]$ con chiave minima ($2i+1$ o $2i+2$), se $P[i]$ è un nodo interno (cioè $2i+1 \leq \text{last}$), e restituisce `null` se $P[i]$ è foglia.

Metodo `removeMin()`

```
minentry <- P[0];
P[0] <- P[last--];
i <- 0;
j <- indexMinChild(P,i);
//Down-heap bubbling
while ((j != null) AND (P[i].getKey() > P[j].getKey())) do{
     swap(P[i], P[j]);
     i <- j;
```

```
    j <- indexMinChild(P,i);
}
return minentry;
```

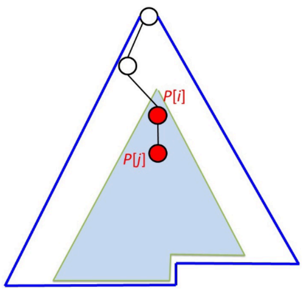
5.4.5.3.1. Esempio (solo chiavi)



5.4.5.3.2. Correttezza

La correttezza discende dal seguente invariante per il while:

- Sia $P[j]$ 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 $P[i]$.



5.4.5.3.3. Complessità

Consideriamo l'esecuzione di `removeMin` su uno heap P con n entry; 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 $\Theta(1)$;
- Numero di iterazioni del while $\leq h$;
- Esiste un'istanza che richiede esattamente h iterazioni (quella in cui la entry in $P[\text{last}]$) ha chiave maggiore di tutte quelle presenti).
  $\implies$ Complessità $\in \Theta(\log n)$.

Riepilogo sulle implementazioni della Priority Queue

Struttura	`min`	`insert`	`removeMin`
Lista non ordinata	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$
Lista ordinata	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$
Heap (implementazione efficiente in spazio su array)	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$

5.5. 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.

> Un algoritmo di dice *in-place* se usa $O(1)$ memoria aggiuntiva oltre a quella necessaria per l'input.

Per esempio: `mergeSort` non è un algoritmo in-place, mentre `insertionSort` lo è.

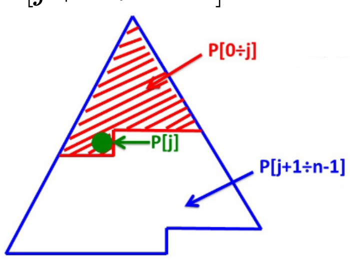
5.5.1. Soluzioni banali

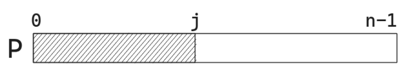
1. Ordinare P in senso non decrescente usando `insertionSort` (tempo $\Theta(n^2)$, in-place) o `mergeSort` (tempo $\Theta(n \log n)$, non in-place);
2. Trasferisco le entry da P a un array di appoggio Q e lo rimetto in P invocando n volte `insert` (tempo $\Theta(n \log n)$, lo vedremo, non è in-place).

5.5.2. Approccio top-down

Con questo metodo si eseguono $n-1$ iterazioni successive, mantenendo il seguente invariante alla fine di ciascuna iterazione j, con $1 \leq j \leq n-1$:

- $P[0 \div j]$ contiene le stesse entry iniziali, riordinate in modo da formare uno heap;
- $P[j+1 \div n-1]$ rimane immutato.





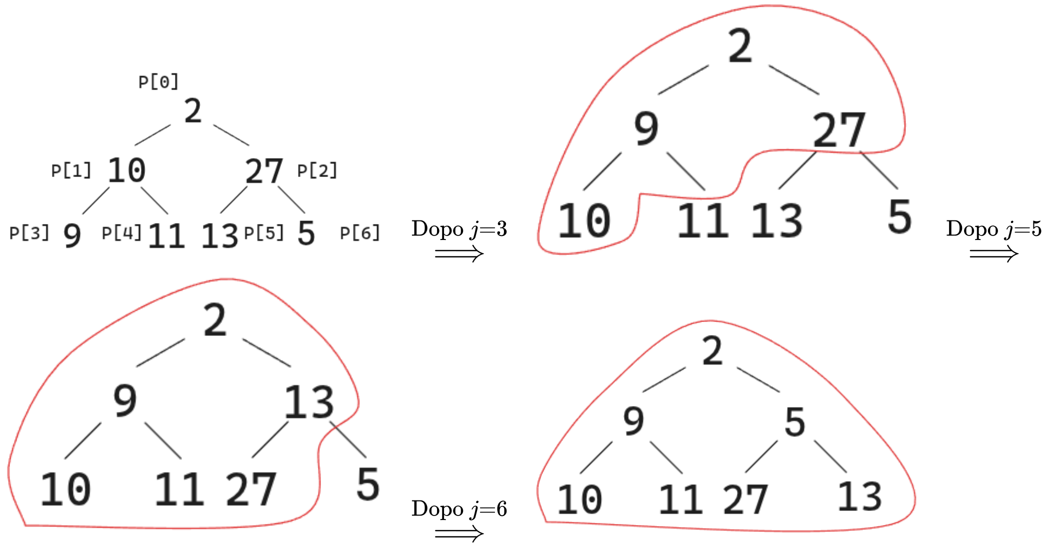
Per l'implementazione si effettua un up-heap bubbling da $P[j]$ in ciascuna iterazione j.

```
for j <- 1 to n-1 do{
     //Up-heap bubbling a partire da P[j]
     i <- j;
     while ((i > 0) AND (P[\floor{(i-1)/2}].getKey() > P[i].getKey())) do{
          swap(P[i], P[\floor{(i-1)/2}]);
          i <- \floor{(i-1)/2};
     }
}
last <- n-1;
```

La *correttezza* discende immediatamente dall'invariante.

5.5.2.1. Esempio

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



5.5.2.2. Complessità

Dimostriamo che la complessità è $\Theta\left(\sum_{j=0}^{n-1} \log j\right)$:

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

Dimostriamo ora 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.

5.5.2.2.1. Dimostrazione

Dimostriamo prima che $\sum_{j=1}^{n-1} \log j \in O(n \log n)$:
$\sum_{j=1}^{n-1} \log j \leq \sum_{j=1}^{n-1} \log n = (n-1) \log n < n \log n \implies \sum_{j=1}^{n-1} \log j \in O(n \log n)$.
Ora dimostriamo che $\sum_{j=1}^{n-1} \log j \in \Omega(n \log n)$:
$\sum_{j=1}^{n-1} \log j \geq \sum_{j=\lfloor \frac{n}{2} \rfloor}^{n-1} \log j \geq \sum_{j=\lfloor \frac{n}{2} \rfloor}^{n-1} \log \lfloor \frac{n}{2} \rfloor \geq \lfloor \frac{n}{2} \rfloor \cdot \log \lfloor \frac{n}{2} \rfloor \implies \sum_{j=1}^{n-1} \log j \in \Omega(n \log n)$.
$\square$

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 la complessità richiesta dall'invocazione di n insert in un heap inizialmente vuoto è $\Theta(n \log n)$ (ovvero la seconda soluzione banale).