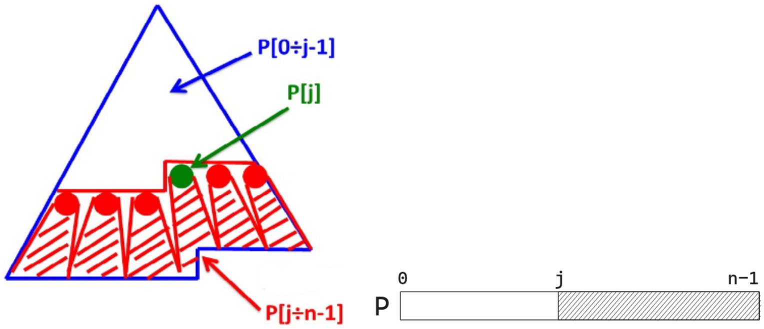
Lezione_17_DeA

5.5.3. Approccio bottom-up

Con questo metodo si eseguono $\lfloor \frac{n}{2} \rfloor$ iterazioni successive, mantenendo il seguente invariante alla fine di ciascuna iterazione j, con $\lfloor \frac{n-2}{2} \rfloor \geq j \geq 0$:

- $P[0 \div j-1]$ rimane immutato;
- $P[j \div n-1]$ contiene le stesse entry iniziali, riordinate in modo da essere una foresta di heap.



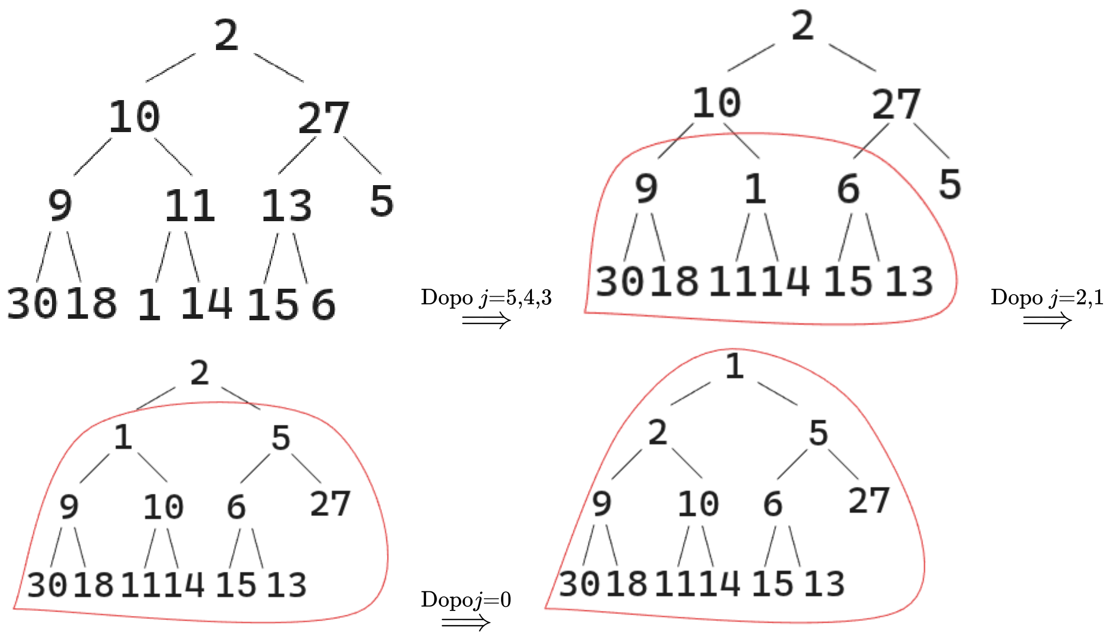
Per l'implementazione si effettua un down-heap bubbling da $P[j]$ in ciascuna iterazione j.

> **Ricorda**
>
> $P\left[\lfloor \frac{n-2}{2} \rfloor\right]$ è il nodo interno più a destra del penultimo livello.

```
last <- n-1;
for j <- \floor{(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);
     }
}
```

La *correttezza* discende immediatamente dall'invariante.

5.5.3.1. Esempio



5.5.3.2. Complessità

Sia $t_{P[j]}$ il costo del down-heap bubbling a partire da $P[j]$.

- Per ogni nodo al livello i, $0 \le i \le h-1$ ($h = \lfloor \log_2 n \rfloor$) il down-heap bubbling costa $O(h-i)$;
- Ci sono 2^i nodi al livello i.
  La complessità è quindi $O\left(\sum_{j=0}^{\lfloor (n-2)/2 \rfloor} t_{P[j]}\right) = O\left(\sum_{i=0}^{h-1} 2^i (h-i)\right)$.

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

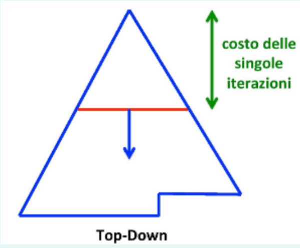
5.5.3.2.1. Dimostrazione

Dimostriamo prima che $\sum_{l=1}^{h} l(\frac{1}{2})^l < 3$:
Osserviamo che $\forall l \ge 1$ vale $l\left(\frac{1}{2}\right)^l \le (\frac{3}{4})^l$, dimostrabile per induzione.
$\sum_{l=1}^{j} l\left(\frac{1}{2}\right)^l \le \sum_{l=1}^{h} (\frac{3}{4})^l = \frac{(\frac{3}{4})^l - \frac{3}{4}}{\frac{3}{4}-1} = \frac{(\frac{3}{4}) - (\frac{3}{4})^{h+1}}{1-\frac{3}{4}} < \frac{\frac{3}{4}}{\frac{1}{4}} = 3$.
$\square$

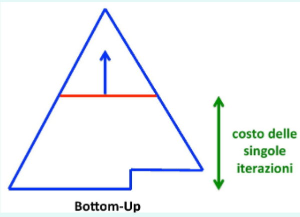
Dimostriamo quindi che $\sum_{i=0}^{h-1} 2^i (h-i) \in O(n)$:
$\sum_{i=0}^{h-1} 2^i (h-i) = 2^h \sum_{i=0}^{h-1} \frac{2^i}{2^h} (h-i) = 2^h \cdot \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}}$
Cambio di variabile $l = h - i$:
$2^h \cdot \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}} = 2^h \sum_{l=1}^{h} l(\frac{1}{2})^l < 2^h \cdot 3$
$h = \lfloor \log_2 n \rfloor \implies 2^h \cdot 3 \in O(n)$
$\impliedby \sum_{i=0}^{h-1} 2^i (h-i) \in O(n)$.
$\square$

Confronto approcci top-down e bottom-up

Nell'approccio *top-down* vi sono 2^i operazioni di costo $\Theta(i)$, quindi *più aumenta la taglia* del livello, *più aumenta il costo dell'up-heap bubbling*. In tutto sarà $\Theta(n \log n)$.



Nell'approccio *bottom-up* vi sono 2^i operazioni di costo $\Theta((\log n) - i)$, quindi *più aumenta la taglia* del livello, *più aumenta il costo del down-heap bubbling*. In tutto sarà $\Theta(n)$.



Entrambe le soluzioni possono essere eseguite direttamente su array senza utilizzare spazio aggiuntivo, quindi sono in-place.

5.6. Sorting tramite Priority Queue

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

Algoritmo `pqSort(S)`

$$\underset{n \text{ chiavi}}{S} \xrightarrow{A} \underset{\text{Priority Queue}}{P} \xrightarrow{B} \underset{n \text{ chiavi ordinate}}{S}$$

Si divide l'algoritmo in due fasi. Nella *fase A* si inseriscono le n chiavi in P una alla volta, invocando il metodo `insert` (considerando le chiavi come entry); nella *fase B* i rimuovono le n chiavi da P una alla volta, invocando il metodo `removeMin`.

5.6.1. Complessità di `psSort(S)`

- Sia P una *lista non ordinata*: La fase A ha complessità $\Theta(n)$, la fase B $\Theta\left(\sum_{i=1}^{n} i\right) \in \Theta(n^2)$ (viene effettuata con un `SelectionSort`);
- Sia P una *lista ordinata*: La fase A ha complessità $\Theta\left(\sum_{i=1}^{n} i\right) \in \Theta(n^2)$ (viene effettuata con un `InsertionSort`), la fase B $\Theta(n)$;
- Sia P uno heap (su array): Sia la fase A che la B hanno complessità $\Theta\left(\sum_{i=1}^{n} \log i\right) \in \Theta(n \log n)$ (vengono effettuate con un `HeapSort`). Con una 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. Ora vediamo come implementare `HeapSort`.

5.6.2. HeapSort in-place

Per realizzare `HeapSort` in-place, 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.

Nella fase A riorganizziamo $S[0 \div n-1]$ in modo che le chiavi rappresentino un max-heap (la chiave in un nodo interno è maggiore o uguale delle chiavi nei figli).
Nella fase B si riorganizza $S[0 \div n-1]$ in modo che le chiavi risultino ordinate.

5.6.2.1. 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 $2i+1 \leq \text{last}$), se invece $S[i]$ è foglia restituisce `null`.

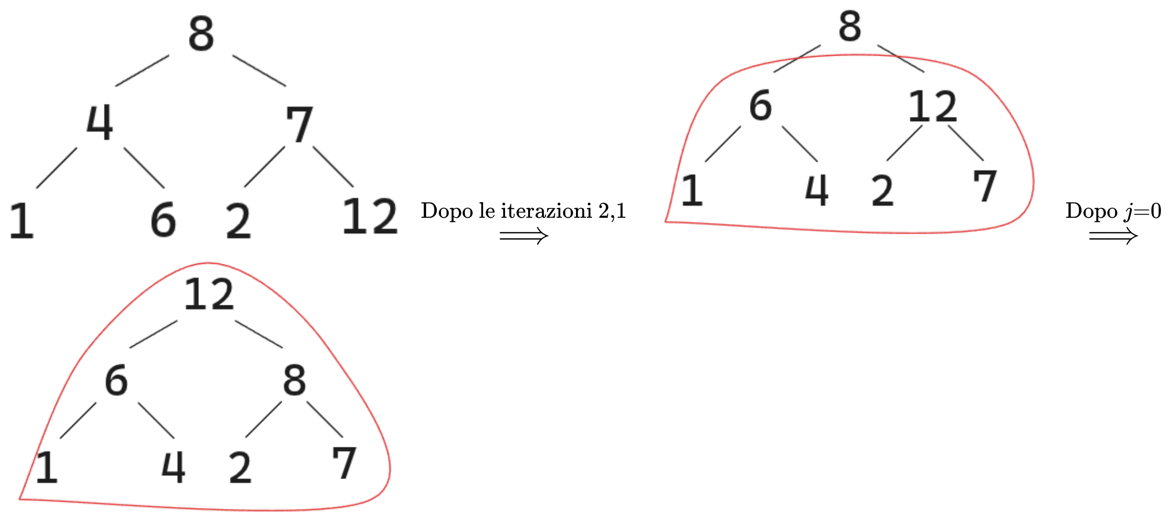
```
last  <- n-1;
for j <- \floor{(n-2)/2} downto 0 do{
     //Down-heap bubbling a partire da S[j]
     i <- j;
     k <- indexMaxChild(S,i);
     while ((j != null) AND (S[i] < S[k])) do {
          swap(S[i], S[k]);
          i <- k;
          k <- indexMaxChild(S,i);
     }
}
```

> **Osservazione**

È la costruzione bottom-up di un max-heap (considerando solamente le chiavi).

5.6.2.1.1. Esempio

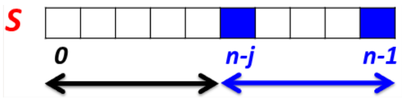
$S = [8\ 4\ 7\ 1\ 6\ 2\ 12]$



5.6.2.2. Fase B max-heap $\rightarrow S$ ordinata

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

- $S[0 \div n-j-1]$ 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.



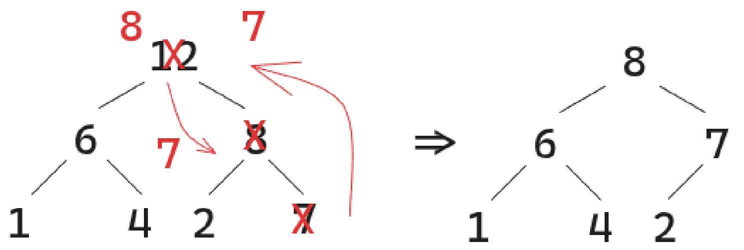
L'implementazione è la seguente.

```
last <- n-1; //segnala l'ultima cella di S che fa parte del max-heap
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); //deve tenere conto che il confine del max-heap segnalato da last arretra in ciascuna iterazione
     while((k != null) AND (S[i] < S[k])) do{
          swap(S[i], S[k]) do {
```

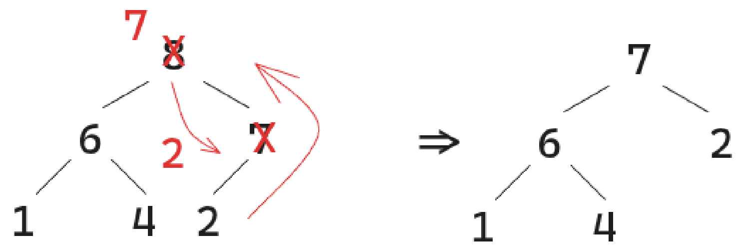
```
                swap(S[i], S[k]);
                i <- k;
                k <- indexMaxChild(S,i);
            }
        }
    }
```

5.6.2.2.1. Esempio

$S \equiv [12\ 6\ 8\ 1\ 4\ 2\ 7]$ ottenuto dopo la fase A.
Dopo l'iterazione $j = 1$ diventa :



$S \equiv [8\ 6\ 7\ 1\ 4 \mid 12]$
Dopo $j = 2$ diventa:



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

5.6.2.3. Complessità

- La fase A equivale alla costruzione bottom-up $\implies \Theta(n)$;
- La fase B equivale all'esecuzione di $n-1$ `removeMax` da heap progressivamente più piccoli $\implies \Theta\left(\sum_{i=1}^{n-1} \log i\right) \in \Theta(n \log n)$.

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

[lezione 18] - Risoluzione di esercizi