Progettazione di Algoritmi **Anno Accademico 2021–2022**

Note per la Lezione 25

Ugo Vaccaro

In questa lezione introdurremo un'utile struttura dati, l' Heap, che useremo nelle lezioni seguenti. L'Heap ha numerose applicazioni anche in ambiti differenti da quelli che consideremo in questo corso..

L'impiego di tecniche generali per il progetto di algoritmi (quali Divide et Impera, Programmazione Dinamica, Tecnica Greedy) ci ha permesso di ottenere vari algoritmi efficienti per svariati problemi. Tuttavia, l'uso delle sopracitate tecniche non garantisce, da solo, la realizzazione di algoritmi efficienti. Infatti: l'uso intelligente di una struttuta dati scelta con accortezza è spesso un fattore cruciale per il progetto di algoritmi efficienti.

Iniziamo con il parlare, in generale, di *Insiemi Dinamici* e loro elementi. Molte algoritmi richiedono di manipolare insiemi la cui composizione può variare nel tempo (ad es. a causa di inserimenti di nuovi elementi, o cancellazioni di vecchi elementi, etc.) Insiemi siffatti vengono chiamati *Insiemi Dinamici*. "Tipiche" assunzioni sugli elementi di un insieme dinamico:

• ciascun elemento è rappresentato da un oggetto i cui campi possono essere manipolati ed esaminati se abbiamo un **puntatore** all'oggetto,
• uno dei campi dell'oggetto è un campo **chiave** per la identificazione dell'oggetto,
• i valori contenuti nel campo chiave si suppongono essere presi da un insieme **ordinato**,
• l'oggetto può anche contenere dei cosiddetti **dati satelliti**, memorizzati in differenti campi dell'oggetto.

Le operazioni su insiemi dinamici possono essere raggruppati in due categorie: operazioni di **interrogazione**, che ritornano informazioni circa l'insieme, e operazioni di **modifica**, che cambiano l'insieme.

Le operazioni che vedremo in questa lezione sono:

• INSERT(S, x) : inserisce un elemento x nell'insieme S.
• MINIMUM(S) : ritorna l'elemento di S con la chiave di minimo valore.
• EXTRACT-MIN(S) : ritorna l'elemento di S con la chiave di minimo valore e lo cancella da S.
• DIMINUISCI-CHIAVE(S, x, k) : decrementa il valore della chiave dell'elemento puntato da x ad un nuovo valore k.
• AUMENTA-CHIAVE(S, x, k) : aumenta il valore della chiave dell'elemento puntato da x ad un nuovo valore k.

Strutture dati che supportano le operazioni di sopra sono dette **(min)code a priorità**. Alternativamente, strutture dati che supportano le analoghe operazioni di INSERT(S, x) , MAXIMUM(S) , EXTRACT-MAX(S) , DIMINUISCI-CHIAVE(S, x, k) , e AUMENTA-CHIAVE(S, x, k) sono dette **(max)code a priorità**.

Le strutture dati**(min)code a priorità** e **(max)code a priorità** hanno molte applicazioni. Ad esempio, **(max)code a priorità** hanno applicazioni nella schedulazione di job in calcolatori.

In una tale situazione vi è una coda che mantiene traccia dei job ancora da eseguire da parte del calcolatore, e delle loro relative priorità. Quando un job è stato eseguito, oppure interrotto, il job con la priorità massima viene selezionato dalla coda con EXTRACT-MAX(S), ed eseguito dal calcolatore. Ogni nuovo job che si presenta per essere eseguito potrà essere inserito nella coda con INSERT(S, x). Le priorità dei job possono essere modificate con DIMINUISCI-CHIAVE(S, x, k) e AUMENTA-CHIAVE(S, x, k), se occorre.

Le strutture dati **(min)code a priorità** hanno applicazioni (oltre che negli algoritmi che vedremo nelle prossime lezioni) anche in classi di applicazioni cosiddette "event-driven".

In tali situazioni, ci sono dei compiti da eseguire (job da eseguire, pacchetti da leggere, ...) in un ordine crescente. Per vari motivi, tali compiti si possono presentare al loro "esecutore", in tempi differenti ed in maniera disordinata. L'esecutore può usare una **(min)code a priorità** per ristabilire l'ordine di esecuzione, relativamente ai compiti nella coda, chiamando EXTRACT-MIN(S) per determinare qual è il nuovo compito da eseguire, ed appena se ne presenta un altro, con il suo relativo numero d'ordine, inserirlo nella coda con INSERT(S, x), usando il numero d'ordine come chiave.

In questa lezione discuteremo di strutture dati per la implementazione efficiente di **(min)code a priorità**. Semplici modifiche a tali strutture forniranno implementazioni efficienti anche per **(max)code a priorità**.

Vediamo innanzitutto una semplice implementazione di **(min)code a priorità** mediante liste.

La prima implementazione che potremmo realizzare è attraverso una semplice lista a puntatori:

Implementazione delle operazioni:

• INSERT(S, x) : inserisci l'elemento x alla testa della coda.
Complessità: $\Theta(1)$

• MINIMUM(S) : scorri tutta la coda e ritorna l'elemento con la chiave di minimo valore.
Complessità: $\Theta(n)$

• EXTRACT-MIN(S) : scorri tutta la coda, ritorna l'elemento con la chiave di minimo valore, e cancellalo dalla coda.
Complessità: $\Theta(n)$

• DELETE(S, x) : individua nella lista l'elemento puntato da x ed eliminalo.
Complessità: $\Theta(1)$

• DIMINUISCI-CHIAVE(S, x, k) : Individua l'elemento puntato da x, e decrementa il suo valore chiave a k.
Complessità: $\Theta(1)$

Se invece usassimo un array ordinato:

Implementazione delle operazioni:

• INSERT(S, x) : inserisci l'elemento x nell'array.
Complessità: $\Theta(n)$

• DELETE(S, x) : individua nell'array l'elemento x ed eliminalo.
Complessità: $\Theta(n)$

• MINIMUM(S) : ritorna l'elemento con la chiave di minimo valore.
Complessità: $\Theta(1)$

• EXTRACT-MIN(S) : ritorna l'elemento con la chiave di minimo valore, e cancellalo dall'array.
Complessità: $\Theta(n)$

• DIMINUISCI-CHIAVE(S, x, k) : individua l'elemento puntato da x, e decrementa il suo valore chiave a k.
Complessità: $\Theta(n)$

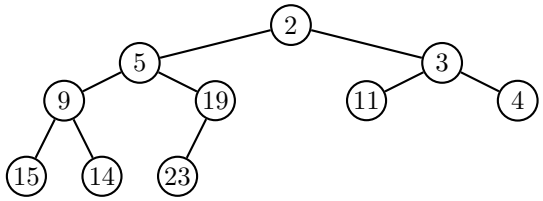
Riassumendo, abbiamo le seguenti complessità per la esecuzione di ciascuna operazione:

Operazione	*Lista a puntatori*	*Array ordinato*
INSERT(S, x)	$\Theta(1)$	$\Theta(n)$
DELETE(S, x)	$\Theta(1)$	$\Theta(n)$
MINIMUM(S)	$\Theta(n)$	$\Theta(1)$
EXTRACT-MIN(S)	$\Theta(n)$	$\Theta(n)$
DIMINUISCI-CHIAVE(S, x, k)	$\Theta(1)$	$\Theta(n)$

La complessità lineare di varie di esse è inaccettabile dal punto di vista pratico. Introduciamo quindi una struttura molto più efficiente, lo Heap. Definizione di Heap:

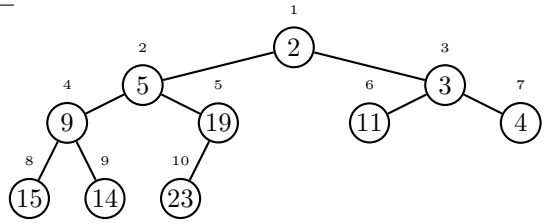
- Albero binario perfettamente bilanciato (l'ultimo livello può essere incompleto (ma è riempito da sinistra a destra)
- Per tutti i nodi v vale che $key(v) \geq key(parent(v))$

Esempio:

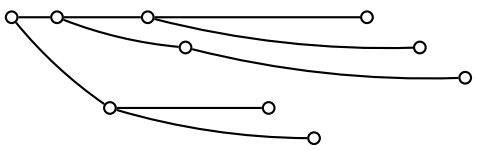


Gli Heap possono essere memorizzati in (almeno) due modi:
1. Usando puntatori
2. In un array, livello per livello da sinistra a destra

Esempio:



1	2	3	4	5	6	7	8	9	10
2	5	3	9	19	11	4	15	14	23

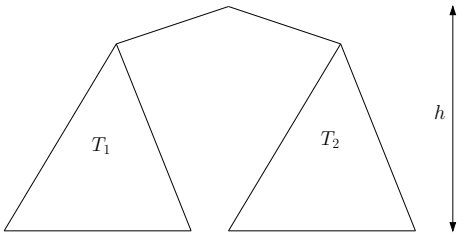


Notiamo che il figlio sinistro del nodo memorizzato nell'entrata i del vettore si trova nell'entrata $2i$, ed il figlio destro nell'entrata $2i + 1$. Analogamente, il padre si trova nell'entrata $\lfloor i/2 \rfloor$

Proprietà degli Heap:

L'elemento di valore minimo è memorizzato nella radice, e l'altezza di uno heap con n nodi è $\Theta(\log n)$

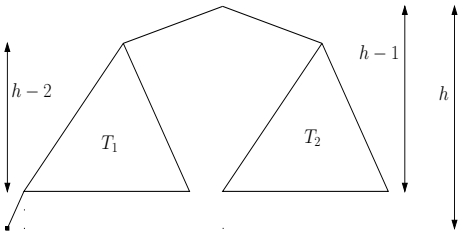
Lo heap di altezza h che ha il *massimo* numero di nodi è fatto così:



Detto n il suo numero di nodi, è ovvio che

$$n = 1 + 2 + 4 + \ldots + 2^h = 2^{h+1} - 1$$

Invece lo heap di altezza h con il *minimo* numero di nodi è fatto così :



Detto n il suo numero di nodi, è ovvio che

$$n = 1 + \#\text{ nodi in } T_1 + \#\text{ nodi in } T_2 = 1 + 2^{h-1} - 1 + 2^{h-1} - 1 + 1 = 2^h$$

Di conseguenza, il numero di nodi n di un generico heap di altezza h soddisfa la relazione $2^h \leq n \leq 2^{h+1} - 1$, ovvero $(n+1)/2 \leq 2^h \leq n$, e quindi

$$h = \Theta(\log n)$$

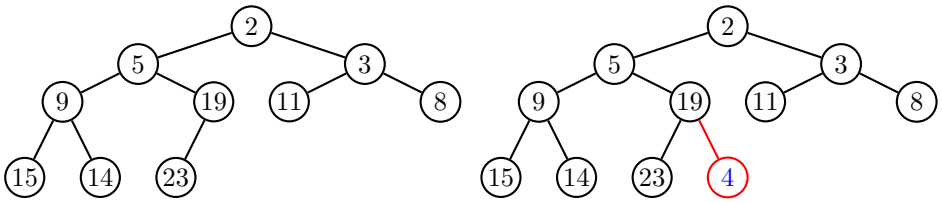
Operazioni sullo heap:

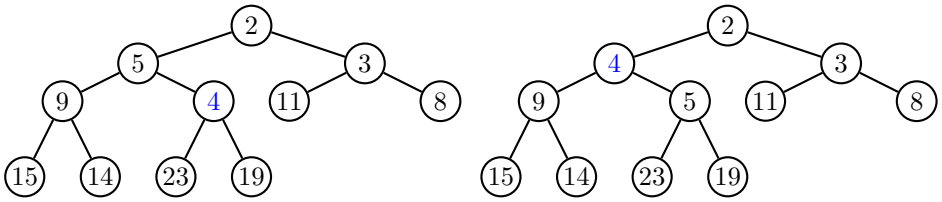
• INSERT(S, x) :

1. Inserisci x in una nuova foglia, piú a sinistra possibile nell'ultimo livello dell'albero.

2. Iterativamente, scambia di posizione elementi con il loro padre, fin quando la proprietà di base dello heap $\forall v\ key(v) \geq key(parent(v))$ non sia stata ristabilita

Esempio: inserimento di 4 nello heap.





Operazione di INSERT(S, x) in uno heap.

La procedura prende in input il vettore A in cui è memorizzato lo heap, ed il valore $k = key[x]$ della chiave del nuovo elemento da inserire. Il campo `heap-size`$[A]$ contiene il numero di elementi attualmente nello heap. Ricordiamo che per ogni nodo memorizzato in $A[i]$, il suo padre nello heap è memorizzato in $A[\texttt{Parent}(i)]$, dove $\texttt{Parent}(i) = \lfloor i/2 \rfloor$

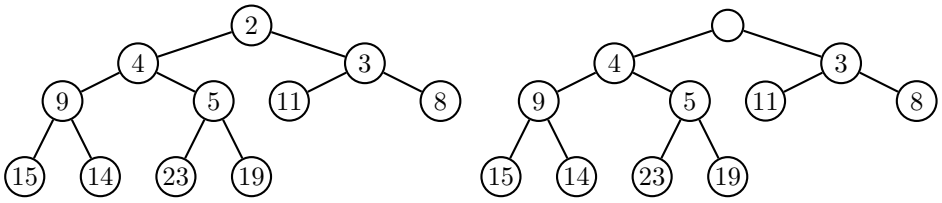
HEAP-INSERT(A, k)
1. `heap-size`$[A] \leftarrow$ `heap-size`$[A] + 1$
2. $A[$`heap-size`$[A]] \leftarrow k$
3. **WHILE** $i > 1$ AND $A[\texttt{Parent}(i)] > A[i]$
4. scambia $A[i] \leftrightarrow A[\texttt{Parent}(i)]$
5. $i \leftarrow \texttt{Parent}(i)$

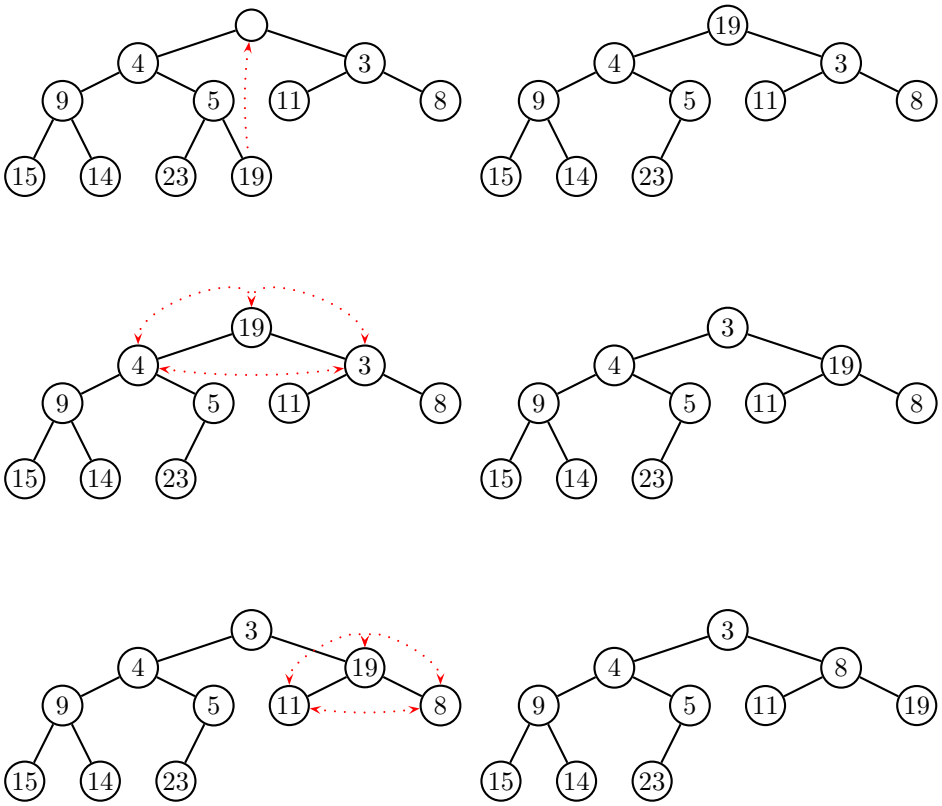
Complessità: Il numero di volte che si cicla nel `WHILE`è al piú pari alla lunghezza del percorso foglia-radice nello heap, che sappiamo essere $\Theta(\log n)$. Quindi, la complessità di HEAP-INSERT(A, k) è $\Theta(\log n)$.

L'operazione di EXTRACT-MIN(S) in uno heap:

• EXTRACT-MIN(S) :
1. Restituisci l'elemento che si trova nella radice dello heap;
2. Sostituisci l'elemento che si trova nella radice con quello che si trova nella foglia piú a destra del livello piú in basso dello heap;
3. Ripristina la proprietà dello heap: $\forall v\ key(v) \geq key(parent(v))$, (se e dove essa è stata violata), scambiando iterativamente un nodo con il figlio che ha il valore chiave *minore*.

Esempio di EXTRACT-MIN(S) in uno heap:





L'operazione di EXTRACT-MIN(S) in uno heap:

HEAP-EXTRACT-MIN(A)
1. `RETURN`($A[1]$)
2. $A[1] \leftarrow A[$`heap-size`$[A]]$, `heap-size`$[A] \leftarrow$ `heap-size`$[A] - 1$
3. `Min-Heapify`$(A, 1)$

dove

`Min-Heapify`(A, i)
1. $\ell \leftarrow$ `Left`(i), $r \leftarrow$ `Right`(i)
2. `IF` $\ell \leq$ `heap-size`$[A]$ AND $A[\ell] < A[i]$
3. `THEN` $smallest \leftarrow \ell$
4. `ELSE` $smallest \leftarrow i$
5. `IF` $r \leq$ `heap-size`$[A]$ AND $A[r] < A[smallest]$
6. `THEN` $smallest \leftarrow r$
7. `IF` $smallest \neq i$
8. `THEN` scambia $A[i] \leftrightarrow A[smallest]$
9. `Min-Heapify`$(A, smallest)$

Nelle linee 2-6 si trova l'elemento minimo tra $A[i], A[\ell], A[r]$. Se esso è $\neq$ da $A[i]$, lo si scambia con $A[i]$ e si itera nel sottoalbero con la cui radice si è effettuato lo scambio.

Complessità di HEAP-EXTRACT-MIN(A, k): ogni chiamata a `Min-Heapify`(A, i) richiede un tempo costante. Chiamate successive a `Min-Heapify`(A, i) vengono effettuate su nodi i di livello sempre inferiore nell'albero,

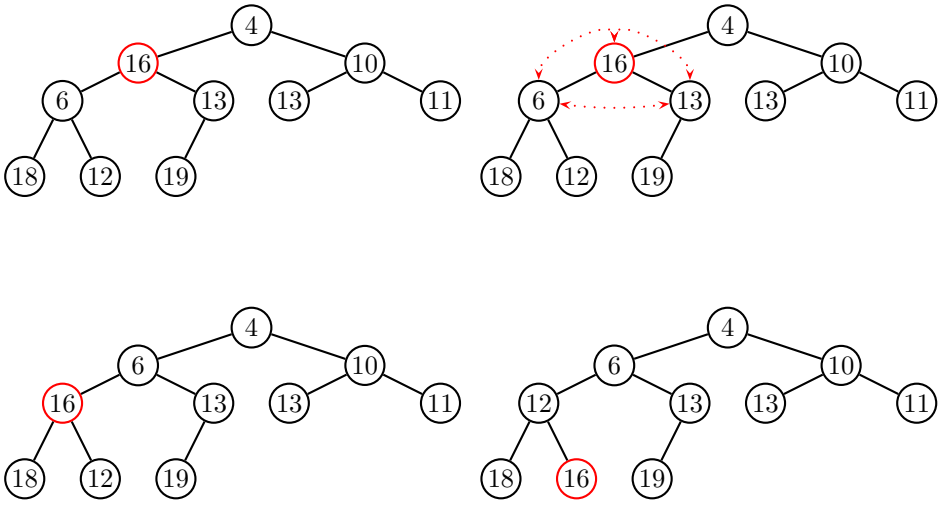
pertanto vengono effettuate al piú $\Theta(h)$ chiamate a `Min-Heapify`(A, i), dove h è l'altezza dello heap. Avendo mostrato in precedenza che $h = \Theta(\log n)$, otteniamo che la complessità di HEAP-EXTRACT-MIN(A, k) è $\Theta(\log n)$.

É molto utile ricordare che la procedura `Min-Heapify`(A, i) è in generale utilizzabile ogni qualvolta si voglia ripristinare in uno heap la proprietà di base dello heap: $key(v) \geq key(parent(v))\ \forall v$, eventualmente violata a causa di un cambiamento del **solo** valore di $key(i)$.

In altri termini, quando `Min-Heapify`(A, i) viene chiamata, si assume che gli alberi binari radicati in `Left`(i) ed in `Right`(i) sia heap corretti, ma che $A[i]$ possa essere **piú grande** dei suoi figli, quindi violando la proprietà dello heap.

La funzione di `Min-Heapify`(A, i) è di scambiare l'elemento $A[i]$ con il minore dei suoi figli, e farlo successivamente " scendere" nel sottoalbero corrispondente, fino a riprisitinare la proprietà dello heap.

Esempio: esecuzione di `Min-Heapify`$(A, 2)$.



Come eseguire ulteriori operazioni su heap:

• MINIMUM(A) :

`ritorna` $A[1]$`.`

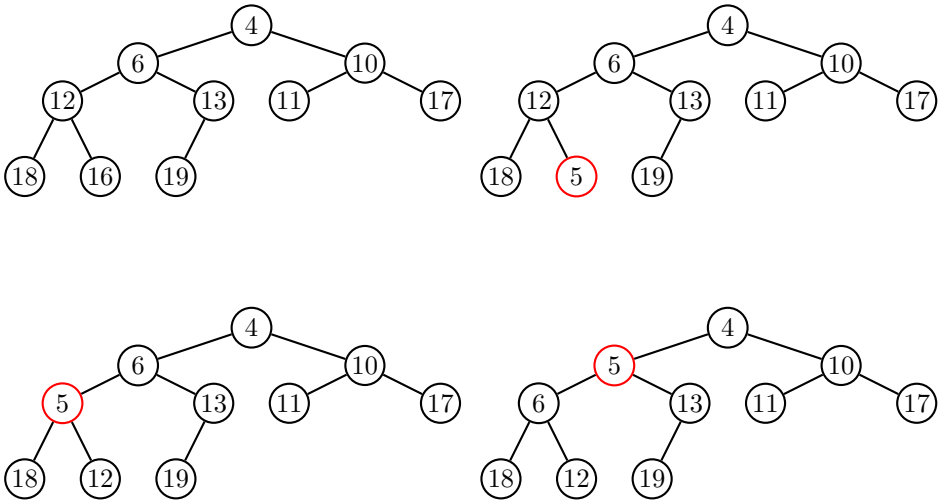
• AUMENTA-CHIAVE(A, i, k) :

$A[i] \leftarrow k$
`Min-Heapify`(A, i)

• DIMINUISCI-CHIAVE(A, i, k) :

1. $A[i] \leftarrow k$
2. `WHILE`$i > 1$ and $A[$`Parent`$(i)] > A[i]$
3. scambia $A[i] \leftrightarrow A[$`Parent`$(i)]$
4. $i \leftarrow$ `Parent`(i)

Esempio di esecuzione di DIMINUISCI-CHIAVE$(A, 9, 5)$



Riassumendo,

Abbiamo le seguenti complessità per la esecuzione di ciascuna operazione:

Operazione	*Lista*	*Array*	*Heap*
INSERT(S, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
DELETE(S, x)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
MINIMUM(S)	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN(S)	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
DIMINUISCI-CHIAVE(S, x, k)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$
AUMENTA-CHIAVE(S, x, k)	$\Theta(1)$	$\Theta(n)$	$\Theta(\log n)$

Applicazioni dello heap all'ordinamento.

La struttura dati heap e le operazioni che essa supporta ci forniscono un semplice algoritmo per l'ordinamento di un vettore $A[1 \dots n]$ di n numeri

1. FOR $i \leftarrow 1$ TO n
2. HEAP-INSERT$(B, A[i])$ %(costruiamo lo heap B mediante n inserimenti degli elementi del vettore A)
3. FOR $i \leftarrow 1$ TO n
4. $A[i] \leftarrow$ HEAP-EXTRACT-MIN(B) %(ordiniamo A mediante n estrazioni di minimo dallo heap B)

Complessità: L'istruzione 2. prende tempo $O(\log n)$ pertanto il `FOR`sulle linee 1. e 2. prende tempo $O(n \log n)$. L'istruzione 4. prende tempo $O(\log n)$, pertanto il `FOR`sulle linee 3. e 4. prende tempo $O(n \log n)$. In totale, l'algoritmo prende tempo $O(n \log n)$ per ordinare n numeri.

É possibile migliorare il precedente algoritmo in due punti:

1. Nella costruzione dello heap (effettuandola quindi in tempo $\Theta(n)$)

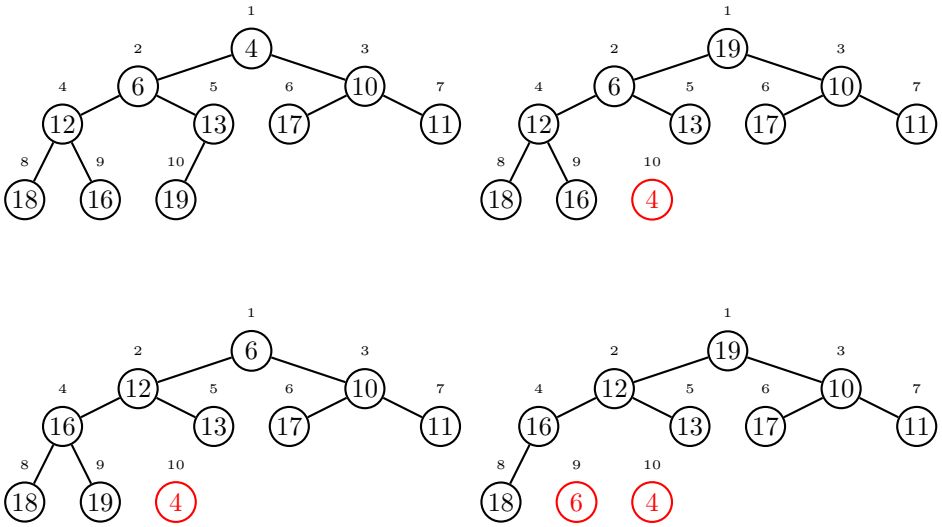
2. Nella quantità di memoria richiesta dall'algoritmo (in modo da non dover far uso di un vettore ausiliario).

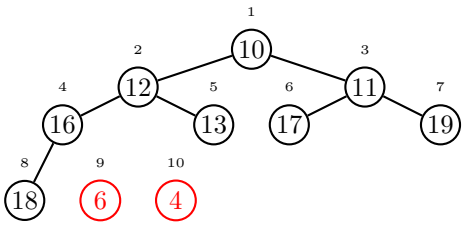
Il seguente algoritmo ordina un array A senza uso di un vettore ausiliario:

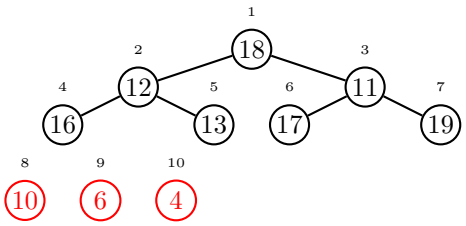
HeapSort(A)
1. Costruisci-Min-Heap(A)
2. `FOR`$i \leftarrow lenght[A]$ **downto** 2
3. scambia $A[1] \leftrightarrow A[i]$
4. $heap - size[A] \leftarrow heap - size[A] - 1$
5. `Min-Heapify`$(A, 1)$

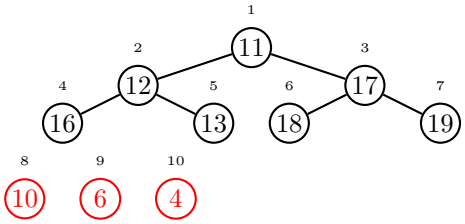
L'algoritmo ordina gli elementi dell'array A nel verso dal piú grande al piú piccolo, in tempo $\Theta(n \log n)$ nel caso peggiore (infatti ognuna delle $n = lenght[A]$ iterazioni del ciclo `FOR`prende al piú tempo $\Theta(\log n)$ nel caso peggiore).

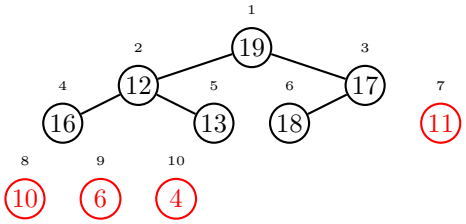
Vediamo un esempio di esecuzione di HeapSort(A)

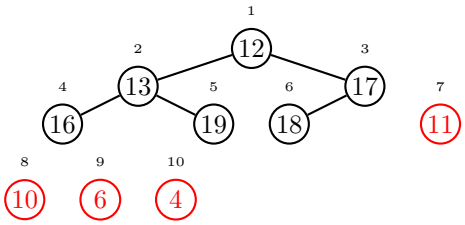


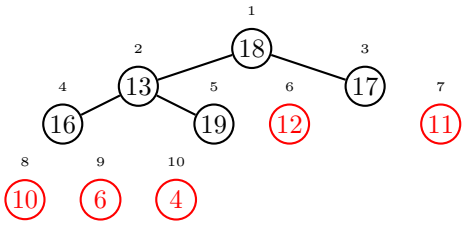

1 10
2 12
3 11
4 16
5 13
6 17
7 19
8 18
9 6
10 4

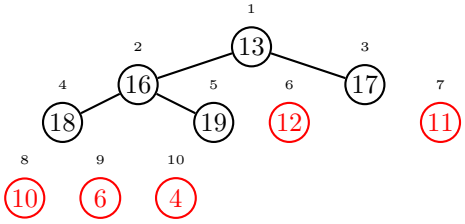

1 18
2 12
3 11
4 16
5 13
6 17
7 19
8 10
9 6
10 4

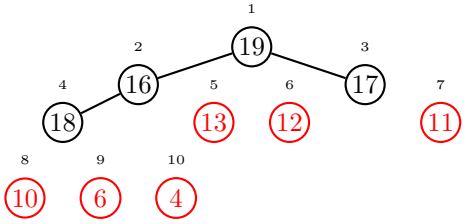

1 11
2 12
3 17
4 16
5 13
6 18
7 19
8 10
9 6
10 4

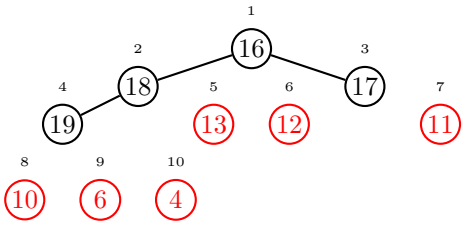

1 19
2 12
3 17
4 16
5 13
6 18
7 11
8 10
9 6
10 4

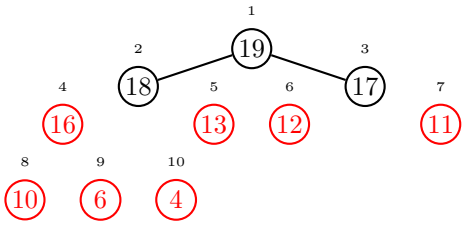

1 12
2 13
3 17
4 16
5 19
6 18
7 11
8 10
9 6
10 4

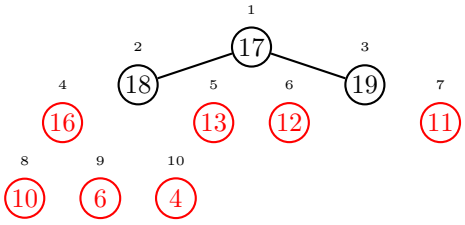

1 18
2 13
3 17
4 16
5 19
6 12
7 11
8 10
9 6
10 4

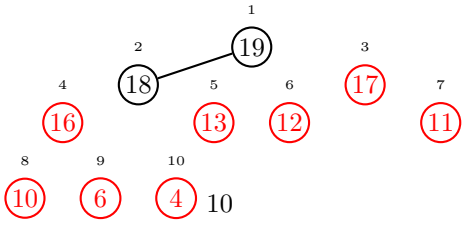

1 13
2 16
3 17
4 18
5 19
6 12
7 11
8 10
9 6
10 4


1 19
2 16
3 17
4 18
5 13
6 12
7 11
8 10
9 6
10 4

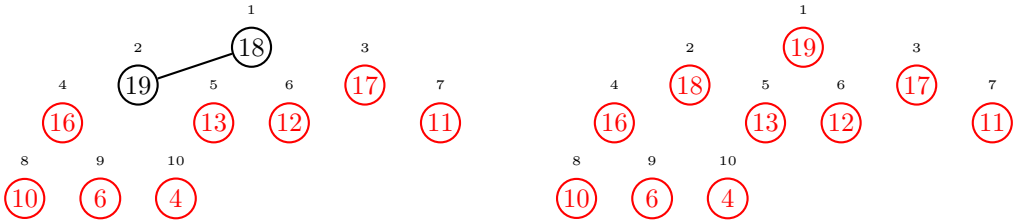

1 16
2 18
3 17
4 19
5 13
6 12
7 11
8 10
9 6
10 4


1 19
2 18
3 17
4 16
5 13
6 12
7 11
8 10
9 6
10 4


1 17
2 18
3 19
4 16
5 13
6 12
7 11
8 10
9 6
10 4


1 19
2 18
3 17
4 16
5 13
6 12
7 11
8 10
9 6
10 4

10



E finalmente, otteniamo l'array ordinato:

1	2	3	4	5	6	7	8	9	10
19	18	17	16	13	12	11	10	6	4

◇

Avremmo anche potuto presentare la struttura dati Max-Heap, in cui la seguente proprietà è verificata:

– Per tutti i nodi v vale che $key(v) \leq key(parent(v))$

La struttura dati Max-Heap può essere utlizzata per implementare **(max)code a priorità** (noi abbiamo visto heap per implementare **(min)code a priorità**) e, analogamente a quanto visto nella lezione, per ordinare un array di numeri. La teoria sviluppabile per ottenere Max-Heap è perfettamente equivalente a quella vista in questa lezione, con le dovute ed ovvie modifiche..