Corso di Algoritmi e Strutture Dati
Anno Accademico 2019–2020
Prof. Giovanni Pighizzini

7. Algoritmi di Ordinamento HeapSort

Descriviamo un algoritmo di ordinamento molto raffinato, basato sull'uso di particolari alberi binari che prendono il nome di *heap*. Come vedremo, questo algoritmo oltre a ordinare n chiavi con un numero di confronti dell'ordine di $n \log n$ (come `mergeSort`) può essere implementato facilmente *in loco* e senza l'uso della ricorsione. Pertanto la memoria richiesta, in aggiunta all'array da ordinare, è di dimensione costante.

7.1 Ordinamento mediante heap (`heapSort`)

La presentazione è suddivisa in due parti. Nella prima parte presentiamo la struttura dati *heap*, illustriamo l'algoritmo in dettaglio, facendo riferimento a tale struttura, e ne valutiamo la complessità in termini di numero di confronti. Nella seconda parte vedremo come uno *heap* possa essere rappresentato direttamente nell'array stesso da ordinare. Questo ci permetterà di ottenere un'implementazione *in loco* di `heapSort`.

7.1.1 La struttura *Heap*

Uno *heap* è un albero binario "quasi" completo, cioè completo almeno fino al penultimo livello[1] e tale che la chiave contenuta in ogni suo nodo è maggiore o uguale rispetto alla chiave contenuta nei figli.[2]

La figura seguente rappresenta uno heap. Per comodità utilizzeremo esclusivamente heap in cui le foglie dell'ultimo livello si trovano più a sinistra possibile.[3]

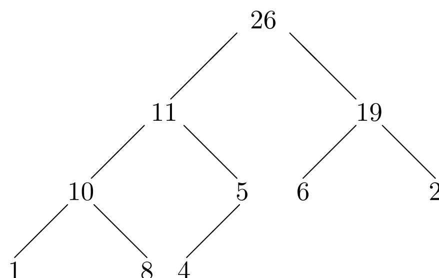
[1] In altri termini, uno heap di altezza h contiene tutti i nodi fino a profondità $h-1$. Di conseguenza, le foglie si trovano a profondità h o $h-1$ e tutti i nodi di profondià minore di $h-1$ hanno entrambi i figli.

[2] Talvolta uno heap che soddisfa questa condizione viene anche detto *max-heap*. Si può definire in maniera analoga un *min-heap*, richiedendo che, per ogni nodo, la chiave in esso contenuta sia minore o uguale della chiave contenuta nei suoi figli.

[3] Questa scelta risulterà comoda per l'implementazione in loco dell'algoritmo, mentre è del tutto irrilevante per la descrizione dell'algoritmo e per il calcolo del numero di confronti.

Il presente materiale integra *ma non sostituisce* il libro di testo consigliato.
Data pubblicazione: 18 novembre 2019

© 2019 Giovanni Pighizzini
Il contenuto di queste pagine è protetto dalle leggi sul copyright e dalle disposizioni dei trattati internazionali. Il titolo ed i copyright relativi alle pagine sono di proprietà dell'autore. Le pagine possono essere riprodotte ed utilizzate liberamente dagli studenti, dagli istituti di ricerca, scolastici e universitari afferenti al Ministero dell'Istruzione, dell'Università e della Ricerca, per scopi istituzionali, non a fine di lucro. Ogni altro utilizzo o riproduzione (ivi incluse, ma non limitatamente a, le riproduzioni a mezzo stampa, su supporti magnetici o su reti di calcolatori) in toto o in parte è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte dell'autore.
L'informazione contenuta in queste pagine è ritenuta essere accurata alla data della pubblicazione. Essa è fornita per scopi meramente didattici e non per essere utilizzata in progetti di impianti, prodotti, ecc.
L'informazione contenuta in queste pagine è soggetta a cambiamenti senza preavviso. L'autore non si assume alcuna responsabilità per il contenuto di queste pagine (ivi incluse, ma non limitatamente a, la correttezza, completezza, applicabilità ed aggiornamento dell'informazione). In ogni caso non può essere dichiarata conformità all'informazione contenuta in queste pagine. In ogni caso questa nota di copyright non deve mai essere rimossa e deve essere riportata anche in utilizzi parziali.

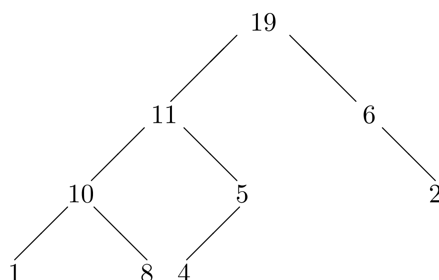


Poiché, come abbiamo dimostrato nelle lezioni, un albero binario di altezza h completo contiene $2^{h+1} - 1$ nodi, essendo uno heap completo almeno fino al penultimo livello, possiamo affermare che in uno heap di altezza h il numero n di nodi soddisfa $2^h \leq n < 2^{h+1}$, da cui otteniamo $h \leq \log_2 n < h + 1$ e dunque $h = \lfloor \log_2 n \rfloor$.

Osserviamo che, come conseguenza della definizione, la radice di uno heap contiene sempre la chiave maggiore. Pertanto, disponendo di uno heap contenente le chiavi che dobbiamo ordinare, possiamo prelevare l'elemento che si trova nella radice e iniziare a collocarlo, come unico elemento, nella sequenza ordinata che dobbiamo produrre come risultato, che costruiremo a partire dal fondo.

Una volta fatto ciò possiamo modificare la struttura in modo da riottenere uno heap ed applicare lo stesso procedimento, inserendo la nuova radice all'inizio della sequenza ordinata che stiamo costruendo.

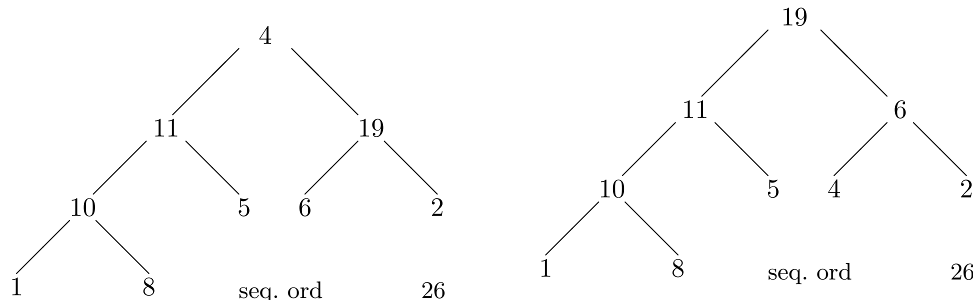
Una volta prelevato l'elemento che si trova alla radice, come facciamo ad ottenere di nuovo uno heap? La soluzione più naturale sembra quella di "far risalire" il maggiore tra i due figli, e proseguire allo stesso modo sino a raggiungere una foglia. La struttura che otteniamo, tuttavia, non è necessariamente uno heap. Nell'esempio precedente si otterrebbe infatti il seguente albero che non è quasi completo:



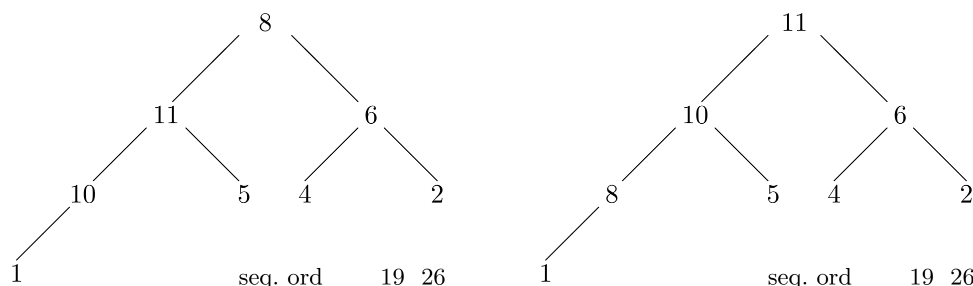
Per evitare questo problema, utilizziamo un procedimento differente: sostituiamo la chiave nella radice con quella contenuta nell'ultima delle foglie, cioè quella che si trova più a destra nell'ultimo livello (nell'esempio quella che contiene 4), rimuovendo tale foglia. Tutti i nodi rispettano la condizione di heap, salvo la radice che potrebbe contenere una chiave inferiore rispetto a uno o a entrambi i figli. In questo caso facciamo "scendere" il dato presente nella radice, scambiandolo con quello di chiave maggiore tra i figli. Se la condizione di heap non è rispettata dal figlio in cui abbiamo spostato il dato, iteriamo su di esso lo stesso procedimento.[4]

Nella seguente figura, a sinistra è rappresentata la struttura iniziale dopo che nella radice è stata messo il dato di chiave 4 dell'ultima foglia, a destra è rappresentato lo heap dopo che il dato di chiave 4 è stato riportato verso il basso con il procedimento appena descritto.

[4]Naturalmente se, oltre alla chiave, vi sono altri campi, questi dovranno seguire la chiave durante gli spostamenti.



A questo punto la radice contiene l'elemento di chiave massima tra quelli rimasti, che può essere memorizzato all'inizio della sequenza ordinata che si sta costruendo (cioè prima di 26). Di nuovo, inserendo nella radice il dato dell'ultima foglia, cioè quello di chiave 8, e poi risistemando lo heap, si ottengono i seguenti alberi:



Lo pseducodice della procedura che "risistema" lo heap dopo che il contenuto di una foglia è stato spostato nella radice è rappresentato nel Codice 7.1. Osserviamo che il numero di confronti utilizzati da `risistema` è, nel caso peggiore, $\Theta(h)$ dove h è l'altezza dello heap. Infatti il valore presente nella radice viene "fatto scendere" lungo un cammino fino a raggiungere la posizione corretta che, nel caso peggiore, potrebbe essere una foglia a distanza massima dalla radice. In questo processo, ad ogni passo viene ispezionato un nodo lungo il cammino, determinando la chiave massima dei figli (un confronto) e confrontandola con la chiave del nodo che si sta ispezionando (un ulteriore confronto). Pertanto per ogni nodo lungo il cammino, salvo la foglia, si effettuano 2 confronti.

Costruzione in uno heap

Consideriamo ora il seguente problema. Supponiamo di disporre di un albero binario quasi completo, cioè la cui struttura soddisfi la condizioni di heap, ma le cui chiavi non rispettino tale condizione. Possiamo, con un metodo efficiente, riorganizzare i contenuti dei nodi in modo che la condizione venga soddisfatta, cioè in modo che la chiave contenuta in ciascun nodo sia maggiore o uguale rispetto alle chiavi contenute nei figli?

Studieremo due soluzioni per questo problema, scegliendo poi la seconda in quanto meno dispendiosa in termini di memoria.

Soluzione ricorsiva

Un metodo molto semplice per risolvere il problema utilizza una strategia *divide-et-impera* (Codice 7.2):

- se l'albero è vuoto non dobbiamo fare nulla;
- se l'albero non è vuoto trasformiamo ricorsivamente ciascuno dei due sottoalberi sinistro e destro in heap; a questo punto tutti i nodi, eccetto la radice, soddisfano la condizione di heap; applicando la procedura `risistema` possiamo trasformare l'intero albero in uno heap.

Codice 7.1 Risistema (fixHeap)

Si ricordi che nell'ordinamento di solito si devono ordinare dei record rispetto a un campo chiave. Nello pseudocodice sono indicati esplicitamente come "altri campi" tutti gli altri campi, che devono seguire la chiave durante gli spostamenti. Nel caso particolare di uno heap contenente sono numeri interi, vi sarà solo il campo chiave, mentre gli altri campi saranno assenti.

```
Procedura risistema (heap H)
  v ← H
  x ← v.chiave                                   // chiave del nodo radice
  y ← v.altri campi                              // altri campi del nodo radice
  daCollocare ← true
  do
      if v è una foglia then
          daCollocare ← false   // la posizione appropriata per x è stata trovata
      else
          u ← figlio di v di valore massimo
          if u.chiave > x then
              v.chiave ← u.chiave                  // i dati in u risalgono: chiave
              v.altri campi ← u.altri campi   // i dati in u risalgono: altri campi
              v ← u                                      // si prosegue su u
          else
              daCollocare ← false   // posizione appropriata per x è stata trovata
  while daCollocare
  v.chiave ← x           // copia la ex-radice nella posizione trovata: chiave
  v.altri campi ← y  // copia la ex-radice nella posizione trovata: altri campi
```

Codice 7.2 Creazione dello heap (versione ricorsiva)

```
Procedura creaHeap (albero binario T)
  /* Trasforma l'albero binario T in uno heap                                 */
  if T ≠albero vuoto then
      creaHeap(T.sx)
      creaHeap(T.dx)
      risistema(T)
```

È possibile effettuare un'analisi del numero di confronti utilizzato da questa procedura, risolvendo un'opportuna equazione di ricorrenza. Tuttavia abbandoniamo questa strada in quanto questa soluzione utilizza memoria aggiuntiva per la ricorsione, nella quale l'altezza dello stack raggiunge quella dello heap e, dunque, è logaritmica rispetto al numero di chiavi.

Soluzione iterativa

Anziché costruire lo heap in maniera top-down, partendo dalla radice, possiamo procedere in modo bottom-up partendo dalle foglie dell'albero. Ispezioniamo cioè l'albero *a partire dall'ultima foglia*, trasformando ogni sottoalbero in uno heap. Quindi:

- Iniziamo a considerare ciascuno nodo di profondità h, da destra verso sinistra, e trasformiamo in heap il sottoalbero che ha tale nodo come radice (questi nodi sono foglie, quindi i relativi sottoalberi sono già heap e dunque per essi non occorre fare nulla).

- Passiamo a considerare ciascun nodo di profondità $h-1$ (sempre da destra verso sinistra[5]) e trasformiamo in heap il sottoalbero che ha radice in esso.
- Ripetiamo la stesso procedimento considerando man mano profondità inferiori sino ad arrivare alla radice. A questo punto l'intero albero è uno heap.

Poiché i sottoalberi sono trasformati in heap a partire dal basso, quando in questo procedimento dobbiamo trasformare in heap il sottoalbero T_x che ha radice in un certo nodo x di profondità p, i sottoalberi di x, avendo radice a profondità $p+1$, sono già stati trasformati in heap in passi precedenti. Dunque, l'unico nodo di T_x che potrebbe non rispettare la condizione di heap è la radice x. È sufficiente applicare la procedura `risistema` per trasformare T_x in uno heap. Lo pseudocodice completo della procedura di creazione dello heap è dato nel Codice 7.3.

Codice 7.3 Creazione dello heap (versione iterativa)

```
Procedura creaHeap (albero binario T)
  /* Trasforma l'albero binario T in uno heap */
  h ← altezza di T
  for p ← h downto 0 do
      foreach nodo x di profondità p do
          risistema(sottoalbero T_x di radice x)
```

Numero di confronti di `creaHeap`

Abbiamo osservato che la procedura `risistema`, nel caso peggiore, effettua un numero di confronti proporzionale all'altezza dell'albero. Pertanto, se applichiamo `risistema` a un sottoalbero di altezza k, il numero di confronti è $\Theta(k)$.

La procedura `creaHeap` richiama `risistema` un certo numero di volte, per sottoalberi di altezze differenti. Per effettuare l'analisi nel caso peggiore, supponiamo di applicare `creaHeap` a un albero binario *completo* di altezza h. Possiamo osservare che, per ogni nodo n di profondità p, il sottoalbero T_n di radice n ha altezza $h-p$. Pertanto la procedura risistema su T_n utilizza $\Theta(h-p)$ confronti. Inoltre a profondità p vi sono 2^p nodi. Pertanto, il numero di confronti per trasformare in heap *tutti* i sottoalberi di profondità p è $\Theta((h-p)2^p)$. Nel ciclo esterno, p varia su tutte le profondità, cioè da 0 a h. Sommando su di esse otteniamo che il numero di confronti è

$$\Theta\left(\sum_{p=0}^{h}(h-p)2^p\right).$$

Calcoliamo la precedente sommatoria:

$$\begin{aligned}
\sum_{p=0}^{h}(h-p)2^p &= \sum_{p=0}^{h-1}(h-p)2^p \\
&= \sum_{p=0}^{h-1}h2^p - \sum_{p=0}^{h-1}p2^p \\
&= h\sum_{p=0}^{h-1}2^p - \sum_{p=0}^{h-1}p2^p \\
&= h(2^h-1)-(h-2)2^h-2 \qquad \text{v. nota}^6 \\
&= 2^{h+1}-2-h
\end{aligned}$$

[5] Questa scelta ci permetterà sccessivamente una facile implementazione *in loco* di `heapSort`.

Essendo l'albero completo, la sua altezza h è logaritmica rispetto al numero di nodi. Questo permette di concludere che il numero di confronti di `creaHeap` è $\Theta(n)$, cioè *lineare* rispetto al numero di chiavi.[7] Si osservi che questo numero è estremamente basso (con meno di un numero lineare di confronti non riusciremmo nemmeno ad ispezionare tutte le chiavi).

Schema di `heapSort`

A questo punto siamo in grado di descrivere l'algoritmo, *nell'ipotesi che gli elementi da ordinare si trovino in un albero binario quasi completo* (Codice 7.4). Prima di tutto, mediante la procedura `creaHeap`, trasformiamo l'albero in uno heap. Poi applichiamo il procedimento iterativo descritto in precedenza, consistente nel prelevare l'elemento maggiore dalla radice, inserirlo all'inizio della sequenza ordinata (che inizialmente è vuota) e risistemare lo heap.

Codice 7.4 Schema di Heapsort

```
Algoritmo heapSort (albero binario A)→ lista
  crea uno heap H da A
  X ← lista vuota
  while H ≠ ∅ do
  |  aggiungi all'inizio di X l'elemento presente nella radice di H
  |  colloca nella radice di H l'elemento che si trova nella foglia più in basso a destra
  |  rimuovi tale foglia
  |  risistema(H)
  return X
```

Numero di confronti di `heapSort`

Seguendo lo schema del Codice 7.4 e utilizzando i risultati già ottenuti, stimiano il numero di confronti utilizzati nel caso peggiore da `heapSort` per ordinare n elementi.

Per ottenere uno heap, la procedura `creaHeap` effettua $\Theta(n)$ confronti. Segue poi la parte iterativa in cui, ad ogni passo, si preleva la radice e si risistema lo heap. Queste operazioni vengono ripetute sino a svuotare lo heap, cioè n volte. Risistemare lo heap utilizza, nel caso peggiore, un numero di confronti proporzionale alla sua altezza che è logaritmica nel numero di chiavi. Dunque, il numero totale di confronti è al più dell'ordine di $n \log n$ (che prevale rispetto alla costruzione dello heap).

Poiché, come dimostreremo, non è possibile ordinare n elementi utilizzando un numero di confronti che, nel caso peggiore, cresca meno della funzione $n \log n$, possiamo concludere che il numero di confronti nel caso peggiore è $\Theta(n \log n)$.

Ordinamento *in loco* di array mediante `heapSort`

Consideriamo ora il problema di ordinare un array. Per utilizzare l'algoritmo appena descritto, sembra abbastanza naturale costruire un albero binario quasi completo, contenente gli elementi dell'array, applicarvi l'algoritmo, utilizzando l'array per memorizzare gli elementi man mano che vengono prelevati (sequenza X del Codice 7.4). In questo modo, tuttavia, l'algoritmo utilizzerebbe una struttura aggiuntiva e non sarebbe *in loco*.

In realtà, è possible implementare l'algoritmo in maniera estremamente più semplice, senza ricorrere a strutture aggiuntive, servendosi di una semplice corrispondenza tra alberi binari quasi

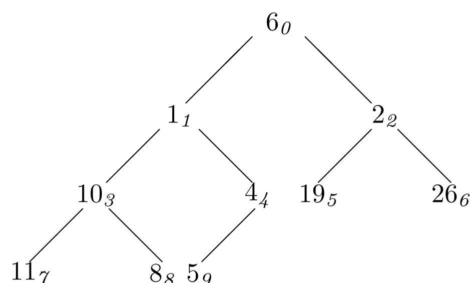
[6] Per calcolare $\sum_{p=0}^{h-1} p2^p$ abbiamo utilizzato l'uguaglianza $\sum_{i=0}^{n} i2^i = (n-1)2^{n+1} + 2$ che può essere verificata per induzione.

[7] Per il numero di confronti della versione ricorsiva si ottiene un risultato analogo.

completi e array, che descriviamo a partire dal seguente esempio. Supponiamo di disporre del seguente array:

indice	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
contenuto	6	1	2	10	4	19	26	11	8	5

Immaginiamo di collocare gli elementi dell'array, nell'ordine in cui compaiono, in un albero binario, riempendo ciascun livello da sinistra verso destra a partire dalla radice, come in una visita in ampiezza. L'albero che otteniamo è il seguente (per comodità indichiamo accanto ad ogni elemento l'indice corrispondente nell'array):



Poiché abbiamo riempito ogni livello da sinistra verso destra, l'albero ottenuto è quasi completo, con le foglie dell'ultimo livello più a sinistra possibile, proprio come negli alberi che abbiamo utilizzato per descrivere `heapSort`. Possiamo inoltre osservare che nell'albero, i figli del nodo che contiene l'elemento di indice i dell'array, se ci sono, contengono gli elementi di indici $2i+1$ e $2i+2$.[8] Per esercizio dimostrate questa proprietà.

Questa corrispondenza permette di interpretare ogni array come albero binario quasi completo e, viceversa, di rappresentare ogni albero binario quasi completo, in cui le foglie all'ultimo livello si trovano più a sinistra possible, con un array, *senza l'uso di puntatori.* L'array risultante viene anche detto *vettore posizionale.*

A questo punto possiamo implementare l'algoritmo `heapSort` descritto in precedenza, *direttamente sull'array dato*, che interpreteremo come albero binario, senza ricorrere all'uso di puntatori.

Poiché, dopo la costruzione dello heap, ogni volta che l'algoritmo preleva un elemento dallo heap (la radice) e lo colloca nella sequenza ordinata, la dimensione dello heap si riduce di un elemento. Durante i passi dell'algoritmo utilizzeremo la prima parte dell'array per rappresentare lo heap contenente gli elementi ancora da ordinare, e la seconda per memorizzare gli elementi già ordinati. Lo schema ad alto livello è dato nel Codice 7.5. Una variabile ℓ indica di volta in volta, l'indice dell'ultima foglia dello heap. Pertanto la porzione di array $A[0..\ell]$ contiene lo heap con gli elementi che sono ancora da ordinare, mentre la parte $A[\ell+1..n-1]$ contiene gli elementi già ordinati, collocati nella loro posizione definitiva. Partendo con $\ell = n-1$, ad ogni iterazione si sistema l'elemento alla radice dello heap (indice 0) nella propria posizione finale, andando ad occupare la posizione ℓ, in cui si trova l'ultima foglia. A tale scopo i due elementi vengono scambiati, risistemando poi lo heap, con lo stesso procedimento descritto in precedenza, implementato direttamente sull'array. Nello schema, alla procedura `risistema` viene fornito, oltre all'array, l'indice della posizione in cui inizia lo heap, cioè della sua radice, in questo caso 0, e l'indice della prima posizione dopo lo heap, cioè ℓ.

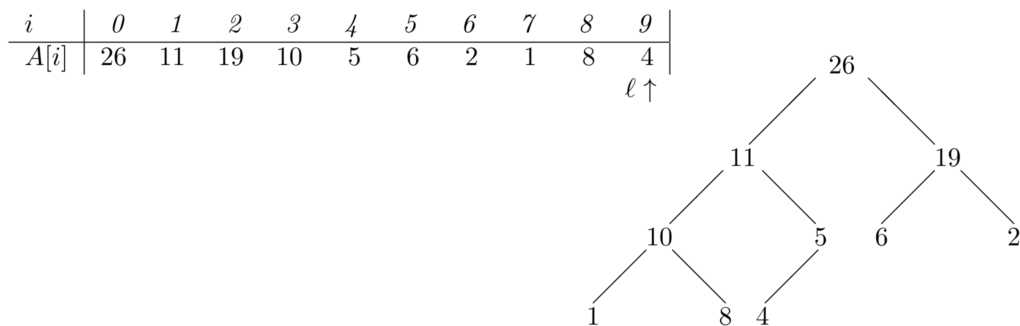
Esempio.
Le figure seguenti mostrano in alcuni passi di esecuzione del Codice 7.5, successivamente alla creazione dello heap, il contenuto dell'array prima di eseguire lo scambio della radice con l'ultima

[8] Se per gli array, anziché considerare indici da 0 considerassimo indici da 1, otterremmo, come figli del nodo contenente l'elemento di indice i, gli elementi di indici $2i$ e $2i+1$.

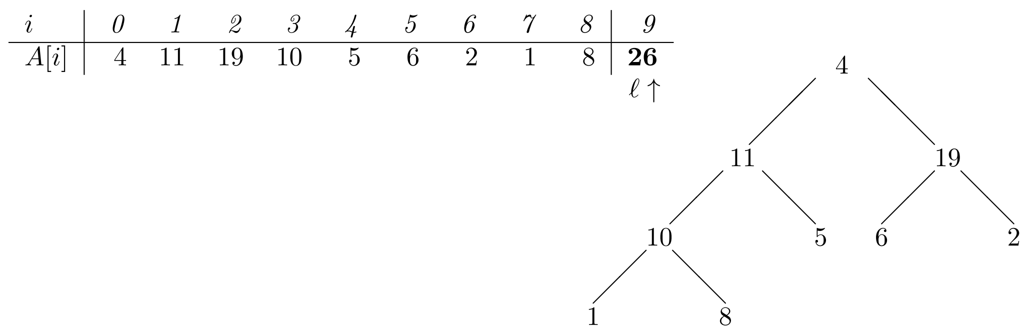
Codice 7.5 Heapsort

Algoritmo *heapSort (array* $A[0..n-1]$*)*
  $creaHeap(A)$
  for $\ell \leftarrow n-1$ **downto** 1 **do**
    scambia $A[0]$ e $A[\ell]$
    $risistema(A, 0, \ell)$

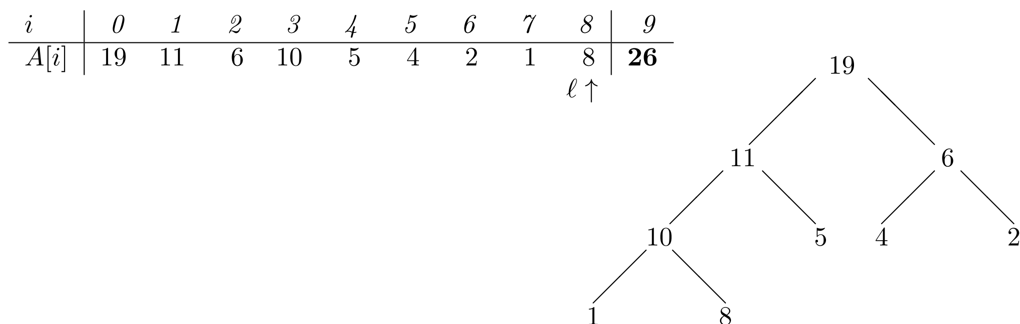
foglia e il contenuto dell'array dopo avere eseguito lo scambio, prima di risistemare lo heap. Gli elementi collocati nella posizione definitiva sono evidenziati in grassetto. A destra dell'array è rappresentato graficamente lo heap corrispondente.



i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$A[i]$	26	11	19	10	5	6	2	1	8	4
										$\ell\uparrow$

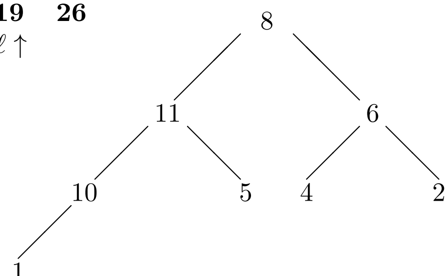


i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$A[i]$	4	11	19	10	5	6	2	1	8	**26**
										$\ell\uparrow$

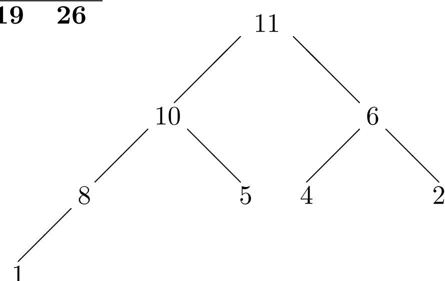


i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$A[i]$	19	11	6	10	5	4	2	1	8	**26**
									$\ell\uparrow$	

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$A[i]$	8	11	6	10	5	4	2	1	**19**	**26**
									$\ell\uparrow$	



i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$A[i]$	11	10	6	8	5	4	2	1	**19**	**26**
								$\ell\uparrow$		



...

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$A[i]$	**1**	**2**	**4**	**5**	**6**	**8**	**10**	**11**	**19**	**26**

□

Per la creazione dello heap a partire dall'albero binario quasi completo abbiamo utilizzato l'algoritmo descritto nel Codice 7.3 che, in modo bottom-up, trasforma in uno heap ogni sottoalbero, partendo da quello radicato nell'ultima foglia. Possiamo riformulare la procedura direttamente sull'array: partendo dall'indice $i = n - 1$ (ultima foglia, cioè ultimo elemento dell'array) decrescendo fino a $i = 0$ (radice), trasformiamo ogni sottoalbero in uno heap. A tale scopo, possiamo invocare `risistema` fornendo come primo indice quello della radice del sottoalbero da risistemare. Inoltre, poiché ogni foglia è già uno heap, evitiamo di chiamare `risistema` per le foglie. Pertanto anziché iniziare con $i = n - 1$, iniziamo con $i = \lfloor n/2 \rfloor$:

Codice 7.6 Creazione di uno heap rappresentato implicitamente in un array (confrontare con il Codice 7.3)

Procedura *creaHeap (array* $A[0..n-1]$*)*
  for $i \leftarrow \lfloor n/2 \rfloor$ **downto** 0 **do**
    risistema(A, i, n)

Infine, lo pseudocodice di `risistema` per lavorare direttamente sull'array è dato nel Codice 7.7.

Spazio

Utilizzando per `creaHeap` la versione iterativa e l'implementazione *in loco*, mediante vettore posizionale, l'algoritmo utilizza spazio costante, oltre all'array da ordinare.

Codice 7.7 Risistema (fixHeap) su array posizionale (confrontare con il Codice 7.1)
Per semplicità qui stiamo schematizzando l'ordinamento di interi e, a differenza della versione ad alto livello presentata nel Codice 7.1, non indichiamo esplicitamente gli altri campi. Pertanto qui il singolo elemento dell'array è direttamente la chiave.

```
Procedura risistema (array A[0..n − 1], intero r, intero ℓ)
 /* A è un array i cui primi ℓ elementi formano uno heap.  Risistema la
    parte di heap la cui radice si trova alla posizione di indice r di A.
    Quando viene richiamata all'interno di heapSort (Codice 7.5), il
    parametro ℓ (numero di elementi nello heap, cioè ancora da ordinare)
    coincide con l'indice di A in cui inizia la parte già ordinata.      */
 v ← r
 x ← A[v]                 // dato da ``far scendere'' nella posizione appropriata
 daCollocare ← true
 do
    if 2 ∗ v + 1 ≥ ℓ then                             // v è l'indice di una foglia
       daCollocare ← false    // la posizione appropriata per x è stata trovata
    else
       u ← 2 ∗ v + 1                                     // indice figlio sinistro
       if u + 1 < ℓ and A[u + 1] > A[u] then    // c'è figlio destro ed è maggiore
          u ← u + 1                                                // del sinistro
       // ora u contiene l'indice del figlio di v di valore massimo
       if A[u] > x then
          A[v] ← A[u]                               // il dato in posizione u risale
          v ← u                                  // si prosegue sul figlio selezionato
       else
          daCollocare ← false   // posizione appropriata per x è stata trovata
 while daCollocare
 A[v] ← x                                         // copia x nella posizione trovata
```

Riassumendo

`HeapSort` è un algoritmo di ordinamento *in loco* che, per ordinare n elementi, effettua $\Theta(n \log n)$ confronti. Pertanto, se ciascun confronto viene effettuato in tempo $O(1)$, il tempo complessivo è $\Theta(n \log n)$.

Si può verificare che questo metodo *non è stabile* (provate ad esempio a vedere cosa succede ordinando un array contenente 3 record, con chiavi `10`, `10` e `20`).