# 9. Heap e Code con Priorità

## 9.1 Altre operazioni sugli heap

Gli heap hanno altre applicazioni, oltre all'algoritmo `heapSort` precedentemente descritto. In particolare sono utili per implementare le *code con priorità*, strutture dati utilizzate in numerosi algoritmi. Prima di descriverle, introduciamo alcune operazioni sugli heap che risulteranno utili nel seguito, discutendone l'implementazione. Per coerenza con la presentazione precedente, ci riferiamo ai *Max-heap*, cioè agli heap in cui la chiave contenuta in ciascun nodo è maggiore o uguale delle chiavi contenute nei figli. In maniera del tutta analoga possono essere trattati i *Min-heap*, nei quali la chiave in ogni nodo è minore o uguale rispetto alle chiavi nei figli e ai quali ci si riferisce più spesso per le code di priorità. Nel seguito supponiamo che lo heap contenga $n$ elementi.

- *Trova l'elemento di chiave massima.*
  Poiché il massimo si trova alla radice questa operazione può essere eseguita in tempo $O(1)$

- *Cancella l'elemento di chiave massima.*
  Come abbiamo visto per `heapSort`, si può sostituire alla radice, che contiene la chiave massima, la foglia più a destra dell'ultimo livello, e applicare la procedura `risistema`. Il numero di passi è proporzionale all'altezza dello heap, e dunque è $O(\log n)$.

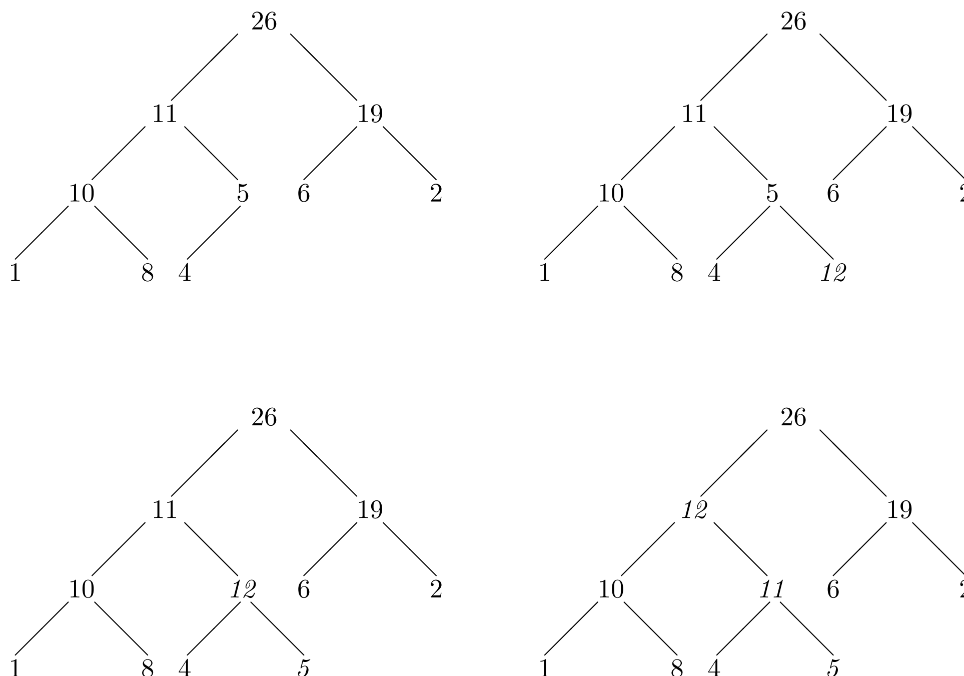
- *Inserisci un nuovo elemento.*
  Per inserire un nuovo elemento possiamo aggiungere una nuova foglia (nella prima posizione libera più a sinistra dell'ultimo livello, se non è pieno, oppure come foglia più a sinistra di un nuovo livello). Poiché la chiave della nuova foglia potrebbe non rispettare la condizione di heap, la struttura va risistemata. Mentre la procedura `risistema` che abbiamo sviluppato, serve per risistemare una radice fuori posto (cioè lavora dall'altro verso il basso), in questo caso bisogna procedere dal basso. Ci occorre pertanto una procedura `risistemaDalBasso` che controlla il padre della nuova foglia: se la chiave in esso è minore, ne scambia il contenuto con la foglia (poiché, prima dell'inserimento della nuova foglia, la struttura era uno heap, se il nodo padre ha un altro figlio, la chiave contenuta in esso non può superare la chiave che c'era nel nodo padre e dunque nemmeno quella che è stata fatta risalire). Si ripete lo stesso procedimento poi sul nodo padre, facendo "risalire" l'elemento inserito sino alla

Il presente materiale integra *ma non sostituisce* il libro di testo consigliato.
Data pubblicazione: 20 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.

posizione appropriata. Nel caso peggiore si arriva alla radice, muovendosi lungo il cammimo che inizia nella foglia dove si è effettuato l'inserimento. Il numero di passi è proporzionale alla profondità dello heap e dunque è $O(\log n)$.

Nella figura seguente è rappresentato uno heap iniziale e la sua evoluzione risistemandolo dal basso, dopo l'inserimento di un nodo di chiave 12.

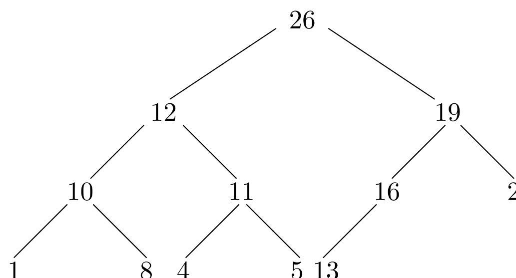



- *Cancella un elemento.*
  Per cancellare un elemento lo si può sostituire con la foglia più a destra dell'ultimo livello, che viene rimossa, e poi risistemare lo heap. Sia $x$ la chiave del nodo $n$ da cancellare e $f$ la chiave della foglia più a destra dell'ultimo livello. Se $x \neq f$ si risistema lo heap in base ai seguenti casi:

  - $f < x$: la chiave di $f$ sicuramente è minore della chiave del padre del nodo $n$, ma potrebbe essere minore di una o entrambe le chiavi dei figli. In questo caso si risistema dall'alto verso i basso il sottoalbero con radice $n$, applicando la procedura `risistema`.

  - $f > x$: la chiave di $f$ sicuramente è maggiore della chiave dei figli del nodo $n$, ma potrebbe essere maggiore della chiavi del padre. In questo caso si risistema dal basso verso l'alto a partire dal nodo $n$, applicando la procedura `risistemaDalBasso`.

In entrambi i casi, per risistemare ci si muove lungo un cammino dal nodo $n$ verso la radice o verso una foglia. Pertanto il numero di passi è $O(\log n)$. Se ad esempio dallo heap nell'ultima figura precedente, si volesse rimuovere il nodo con chiave 19, esso andrebbe sostituito con la foglia contenente 5 che andrebbe poi risistemata verso il basso. D'altra parte, per rimuovere il nodo di chiave 10 dallo heap nella seguente figura, lo si sostituisce con la foglia contenente 13 che viene poi fatta risalire fino a raggiungere la posizione di figlio sinistro della radice:




- *Modifica la chiave di un elemento.*
  Modifica la chiave associata a un nodo $n$. Abbiamo due casi, analoghi ai precedenti:
  - *La chiave viene diminuita:*
    Il valore della nuova chiave in $n$ è sicuramente minore della chiave del padre, ma potrebbe risultare minore di una o entrambe le chiavi dei figli. Applichiamo la procedura `risistema` a partire dal nodo $n$, per far "scendere" il nodo con la chiave aggiornata nella posizione appropriata.
  - *La chiave viene aumentata:*
    Il valore della nuova chiave in $n$ è sicuramente maggiore delle chiavi dei figli, ma potrebbe risultare maggiore anche della chiave del padre di $n$. Applichiamo la procedura `risistemaDalBasso` a partire dal nodo $n$, per far "salire" il nodo con la chiave aggiornata nella posizione appropriata.

  Anche in questo caso il numero di passi è $O(\log n)$.

## 9.2 Code con priorità

Utilizzando gli heap e le operazioni su di essi descritte in precedenza, si possono implementare facilmente delle strutture a *coda* in cui gli elementi vengono prelevati con un criterio di priorità. Solitamente la priorità è indicata da una chiave numerica con la convenzione che *chiavi inferiori indicano priorità più alta*. Pertanto, prelevare il primo elemento, cioè quello con priorità più alta, equivale a prelevare quello con chiave minima (numero più basso). Consideriamo le seguenti operazioni:

- `findMin()`
  Restituisce l'elemento minimo nella coda (senza rimuoverlo).
- `deleteMin()`
  Rimuove l'elemento minimo dalla coda e lo restituisce.
- `insert(elem e, chiave k)`
  Inserisce nella coda un elemento $e$ con associata una chiave (priorità) $k$.
- `delete(elem e)`
  Cancella l'elemento $e$.
- `changeKey(elem e, chiave d)`
  Modifica la priorità dell'elemento $e$, assegnando come nuovo valore $d$.

Le code con priorità possono essere implementate utilizzando dei *Min-heap* con le operazioni descritte nel paragrafo 9.1.[1] Come nell'implementazione di `heapSort`, lo heap può essere rappresentato mediante un array (o meglio la prima parte di un array, lasciando nella seconda spazio per eventuali inserimenti).

[1] L'implementazione delle operazioni è stata descritta per i *Max-heap*. È facile adattarla ai *Min-heap*.

In particolare, se la coda contiene $n$ elementi e assumendo il criterio di costo uniforme, l'operazione di prelevare il primo elemento può essere effettuata in tempo costante, mentre le altre operazioni `deleteMin` e `insert` in tempo $\Theta(\log n)$ nel caso peggiore. Anche le operazioni `delete` e `changeKey` possono essere realizzate in tempo $\Theta(\log n)$, *se è nota* la posizione nello heap dell'elemento da cancellare o modificare. Per evitare di cercare tale posizione (operazione che potrebbe richiedere l'attraversamento dell'intero heap), si può tenere una struttura ausiliaria che fornisca, per ogni elemento, la sua posizione all'interno dello heap. Se ad esempio gli elementi sono interi tutti differenti tra loro nell'intervallo $1, .., k$, si potrebbe utilizzare un array `posizione`, in cui `posizione[i]` indica l'indice in cui l'elemento $i$ è memorizzato nello heap. Naturalmente, ogni volta che si manipola lo heap, spostandone gli elementi, tale struttura deve essere aggiornata.