Heap binario

Proprietà e implementazione

T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduzione agli Algoritmi e Strutture Dati, McGraw-Hill, 2005

Pagg. 133-139

Heap Binario

- Si rappresenta con un albero binario quasi completo
- Nella radice è sempre presente il massimo valore
- L’albero è rappresentato da un array con indici 0..N-1 dove N è il numero di elementi
- Padre(i) = trunc((i-1)/2)
- FiglioSinistro(i) = 2*i + 1
- FiglioDestro(i) = 2*i + 2
- Foglie: dall’indice trunc(N/2) all’indice N-1

Esempio di Heap binario

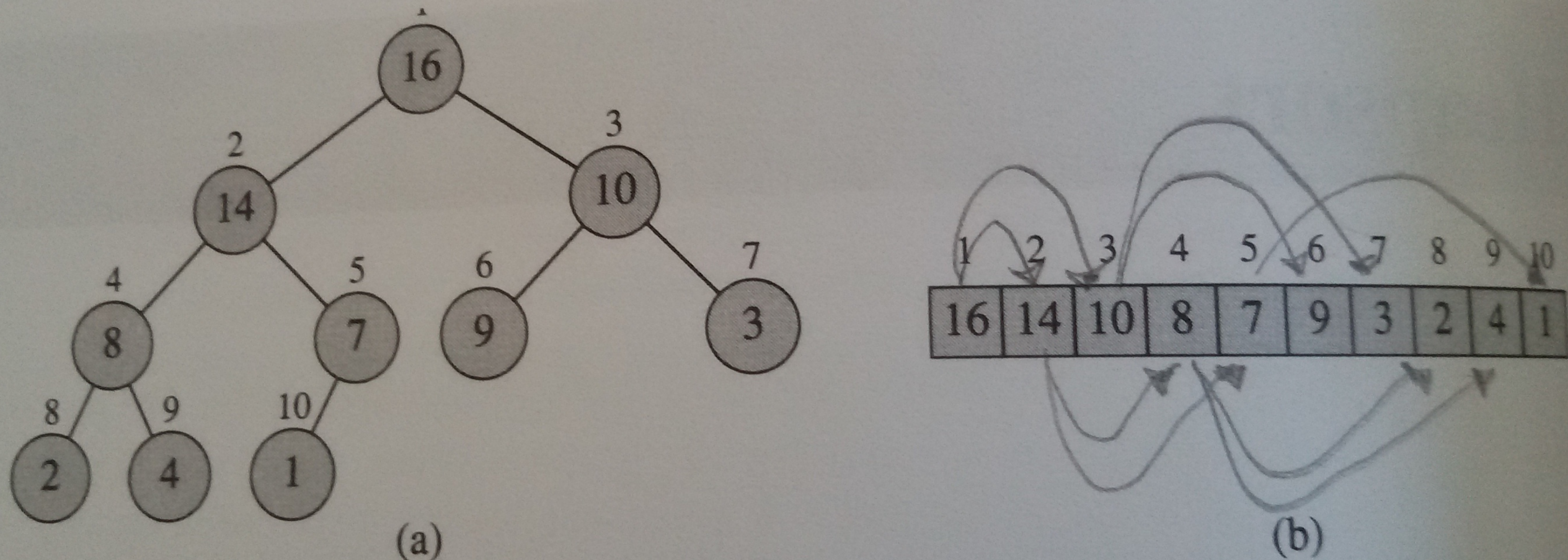


Figura 7.1 *Uno heap visto come un albero binario* (**a**) *e un array* (**b**). *Il numero dentro ogni nodo è il valore memorizzato in quel nodo. Il numero vicino al nodo è il corrispondente indice nell'array.*

Nell'esempio le posizioni dell'array sono numerate da 1 a N, mentre poi nell'implementazione useremo gli indici da 0 a N − 1

Heapify

- Operazione fondamentale che permette di creare uno Heap a partire da una nuova radice e due Heap che rappresentano i due sottoalberi del nuovo heap
- Dato un indice i dell’array si assume che FiglioSinistro(i) e FiglioDestro(i) siano indici che nell’array sono radici di sottoalberi che rispettano la proprietà dell’heap
- Non c’è nessuna ipotesi sulle altre posizioni dell’array

Heapify

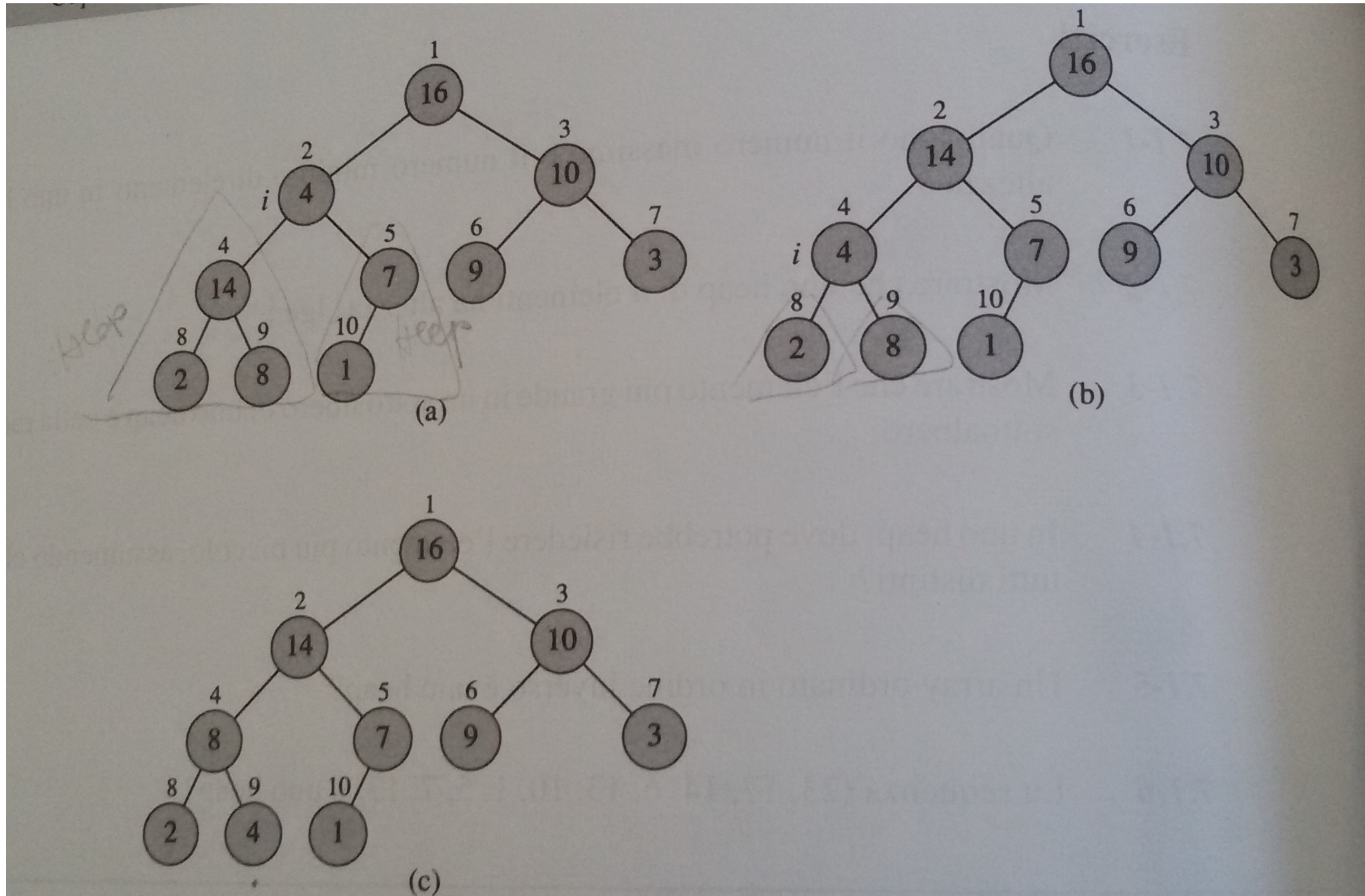


Figura 7.2 *L'azione di* HEAPIFY$(A, 2)$, *dove heap-size*$[A] = 10$. (**a**) *La configurazione iniziale dello heap viola la proprietà dello heap poiché* $A[2]$ *nel nodo* $i = 2$ *non è più grande dei suoi figli. La proprietà dello heap è ripristinata per il nodo 2 in* (**b**) *scambiando* $A[2]$ *con* $A[4]$, *ma ciò porta a violare la proprietà dello heap per il nodo 4. La chiamata ricorsiva* HEAPIFY$(A, 4)$ *pone* $i = 4$. *Dopo lo scambio di* $A[4]$ *con* $A[9]$, *come mostrato in* (**c**), *il nodo 4 è sistemato e la chiamata ricorsiva* HEAPIFY$(A, 9)$ *non richiede ulteriori cambiamenti della struttura di dati.*

Heapify

- Una volta eseguita (e dopo eventuali chiamate ricorsive) il nodo in posizione i è radice di uno heap corretto
- La procedura ha complessità $O(\log_2 N)$ nel caso pessimo

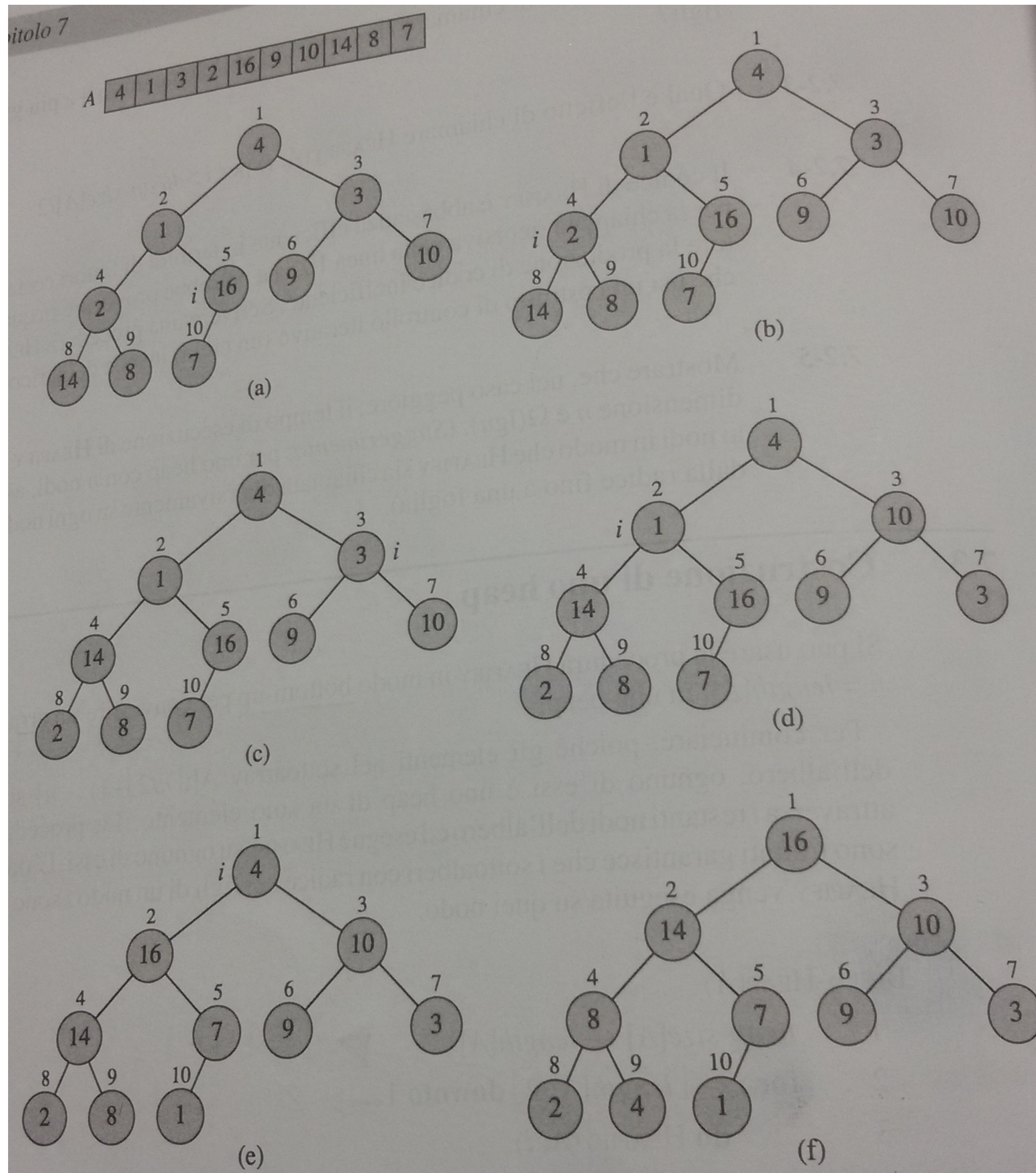
Altre operazioni

- **Restituzione del massimo**, complessità O(1), operazione fondamentale
- **Estrazione del massimo**, complessità $O(\log_2 N)$, in quanto bisogna richiamare Heapify(0) dopo aver copiato in posizione 0 l'elemento in posizione $N - 1$ e dopo aver eliminato quest'ultimo dall'array

Costruzione di uno heap

- Si può facilmente trasformare qualsiasi array in uno heap
- La procedura considera tutte le foglie come heap (questo è sempre vero per una foglia)
- Poi dall'ultimo elemento non-foglia fino alla posizione 0, viene chiamata Heapify
- Complessità O(N) nel caso pessimo

Costruzione di uno heap



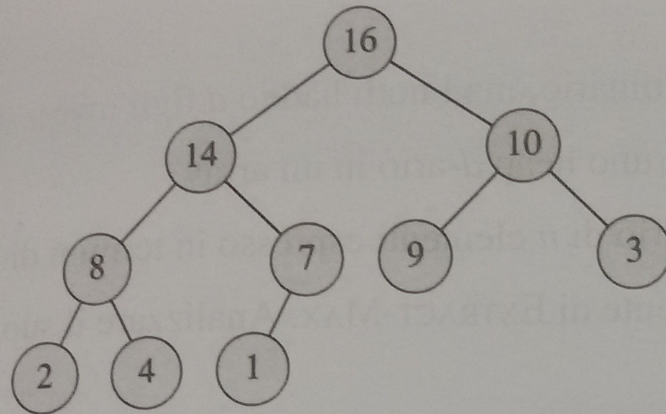
Costruzione di uno heap

- Il primo nodo non foglia è il nodo che contiene 16 (indice 4 se indici 0..N-1)
- Viene quindi chiamata Heapify(4) e il risultato è l'albero successivo
- Su questo viene chiamato Heapify(3), cioè sul nodo che contiene 2
- E così via fino all'ultima chiamata su Heapify(0)

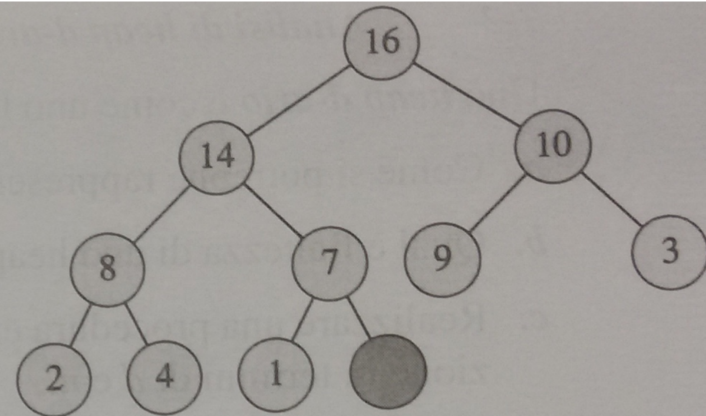
Inserzione di un elemento

- Per inserire un nuovo elemento nello heap si posiziona il nuovo valore in fondo all'array
- La foglia viene confrontata con il padre e scambiata con esso se maggiore
- Il procedimento continua fino a quando non si trova che il padre corrente del nodo è maggiore di esso

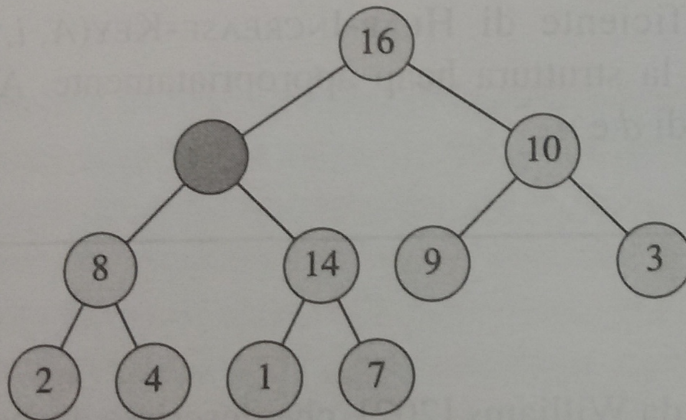
Inserzione di un elemento



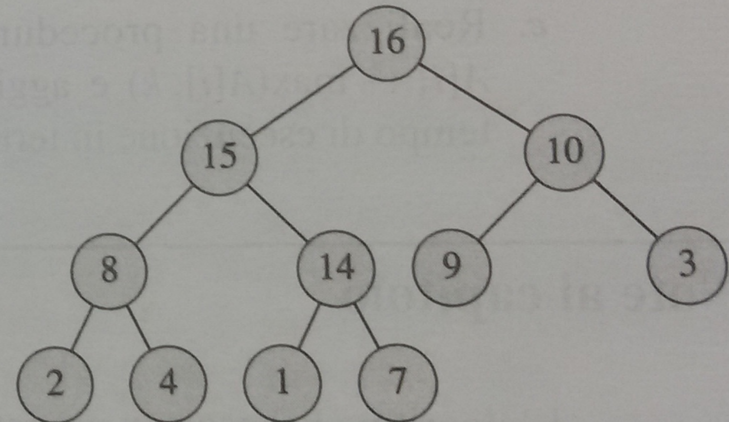
(a)



(b)



(c)



(d)

Figura 7.5 *L'esecuzione di* HEAP-INSERT. (**a**) *Lo heap della figura* 7.4 (a) *prima che si inserisca un nodo con chiave* 15. (**b**) *Una nuova foglia è aggiunta all'albero.* (**c**) *I valori sul cammino dalla nuova foglia alla radice sono risistemati finché non si trova un posto per la chiave* 15. (**d**) *La chiave* 15 *è inserita.*

Implementazione

- Si veda il codice allegato per l’implementazione in Java