ALGORITMI E STRUTTURE DATI

Giulia Ponassi

MERGE SORT

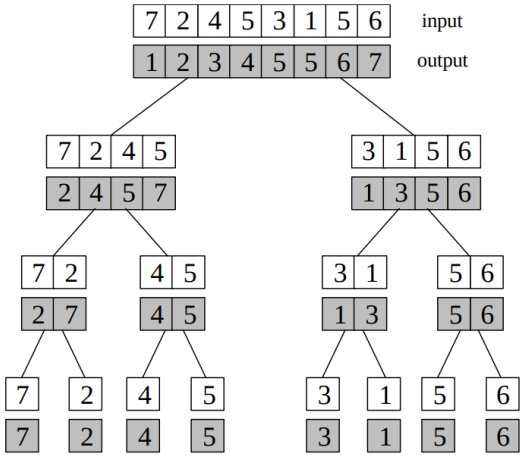
La complessità di merge sort è $\vartheta(n \log n)$ sia nel caso migliore che nel caso peggiore; infatti, non essendo un algoritmo adattivo, effettua tutte le chiamate ricorsive e tutti i confronti necessari in qualsiasi caso.
La complessità deriva dal prodotto tra il numero di livelli e il costo di ciascun problema. Ad ogni livello J si hanno 2^J sottoproblemi, ognuno lungo $n/2^J$, e quindi risolvibili in $\vartheta(n/2^J)$. Per conoscere il costo di ogni livello, bisogna quindi fare $2^J * n/2^J = \vartheta(n)$.
Il numero dei livelli dell'albero ricorsivo è $\log n$, quindi il costo di merge sort è:

$$\log n * \vartheta(n) = \vartheta(n \log n)$$

(numero di livelli * costo di ciascun problema)

Caso migliore: array già ordinato.
Caso medio: elementi dell'array disposti in ordine casuale.
Caso peggiore: array ordinato in ordine decrescente.

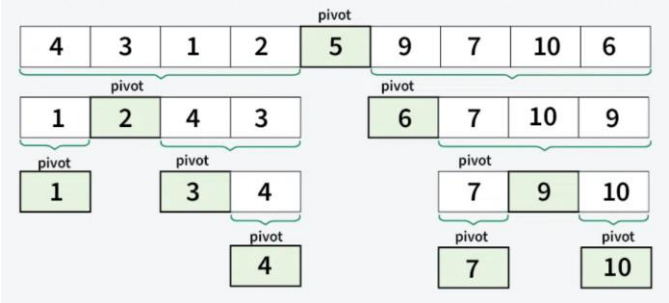
Esempio di merge sort:



QUICK SORT

Caso migliore:
Il caso migliore è quando si sceglie come pivot il mediano della sequenza; questo è però solo un caso ipotetico in quanto non si può sapere il mediano senza aver prima riordinato la sequenza.
Essendo che il pivot è sempre l'elemento mediano, la sequenza viene divisa dalla partition in due sottosequenze di lunghezza circa $n/2$, ottenendo un albero binario che ha altezza $\log n$, ossia il numero di iterazioni della funzione ricorsiva. Ad ogni livello J dell'albero, la partition viene eseguita su 2^J sottoproblemi, ognuno lungo $n/2^J$: la complessità è quindi $2^J * \vartheta\left(n/2^J\right) = \vartheta(n)$. La complessità finale sarà quindi $\vartheta(n \log n)$.

Esempio caso migliore:



Caso peggiore:
Si ricade nel caso peggiore quando si usa come pivot l'elemento maggiore o minore della sequenza, creando una sottosequenza vuota e una sottosequenza che contiene tutti gli elementi tranne uno (il pivot).
Per calcolare la complessità si sommano le operazioni fatte ad ogni chiamata ricorsiva: alla prima chiamata, si eseguono n operazioni (con n = numero degli elementi della sequenza), alla seconda si eseguono n-1 operazioni e così via fino all'ultimo livello dell'albero ricorsivo in cui si esegue 1 operazione. Questo calcolo si sviluppa nella sommatoria per i che va da 1 a n. Quindi $n + (n-1) + \ldots + 1 = n^2$ e la complessità è $\vartheta(n^2)$.

Esempio caso peggiore:

4	5	2	3	8	6
2	4	5	3	8	6
2	3	4	5	8	6
2	3	4	5	8	6
2	3	4	5	8	6
2	3	4	5	6	8
2	3	4	5	6	8

Caso medio:
Nel caso medio il pivot è scelto in modo randomizzato e la complessità è $\vartheta(n \log n)$.

Esempio caso medio:

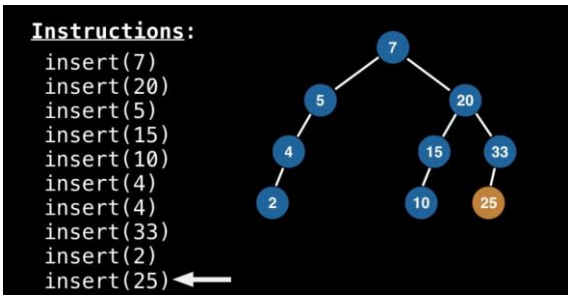
BINARY SEARCH TREE

Inserimento:
Per inserire un nuovo elemento in un albero binario di ricerca, confronto il nodo da inserire con il nodo corrente, partendo dalla radice: se il valore del nodo da inserire è minore del nodo corrente, mi sposto nel sottoalbero sinistro, mentre se il valore del nodo da inserire è maggiore del nodo corrente, mi sposto nel sottoalbero destro. Questo confronto continua ricorsivamente fino a quando non trovo un nodo che non ha figli nel sottoalbero in cui deve essere inserito in nuovo nodo, e lo inserisco come nuovo figlio nel rispettivo sottoalbero.

Caso migliore o caso medio:
In un albero, il percorso dalla radice alla foglia è proposizionale al logaritmo del numero di nodi, ossia all'altezza dell'albero. La complessità sarà quindi $\vartheta(\log n)$ oppure $\vartheta(h)$ (con h = altezza dell'albero).

Caso peggiore:
Il caso peggiore si verifica quando l'albero diventa completamente sbilanciato, con un solo figlio per nodo; in questo caso, ad ogni inserimento l'elemento viene aggiunto alla fine dell'albero, che si comporta come una lista. Quindi, il numero di confronti necessari è pari al numero di nodi già presenti. La complessità sarà allora $\vartheta(n)$.

Esempio di insert in un BST:



Cancellazione:
La cancellazione in un albero binario di ricerca richiede la rimozione dell'elemento mantenendo le proprietà dell'albero. Prima di tutto, si ricerca il nodo che deve essere eliminato confrontando il valore del nodo da eliminare con quello corrente, e, usando la proprietà di ordinamento del BST, procedere con la ricerca a sinistra a o a destra. Una volta trovato il nodo, si procede ad eliminarlo:

- Se il nodo da eliminare è una foglia, si elimina semplicemente;
- Se il nodo ha solo un figlio, il nodo viene rimosso e il figlio viene collegato direttamente al padre del nodo eliminato;
- Se il nodo ha due figli, posso usare il predecessore in ordine (nodo più grande nell'albero sinistro) o il successore in ordine (nodo più piccolo nell'albero destro) del nodo da eliminare, ed usarli per sostituire il nodo da eliminare. Uso la funzione ausiliare (deletemin o deletemax) per trovare e rimuovere il nodo minimo (o massimo), e sostituire il valore del nodo da eliminare con il valore di questo nodo minimo (o massimo).

Caso migliore:
Il caso migliore si verifica quando il nodo da eliminare è una foglia o ha un solo figlio. La complessità è quindi $\vartheta(\log n)$.
Caso peggiore:
Il caso peggiore si verifica quando il nodo da eliminare ha due figli, il che richiede la sostituzione con il successore o con il predecessore. La complessità è $\vartheta(n)$.

Esempio di delete in un BST:

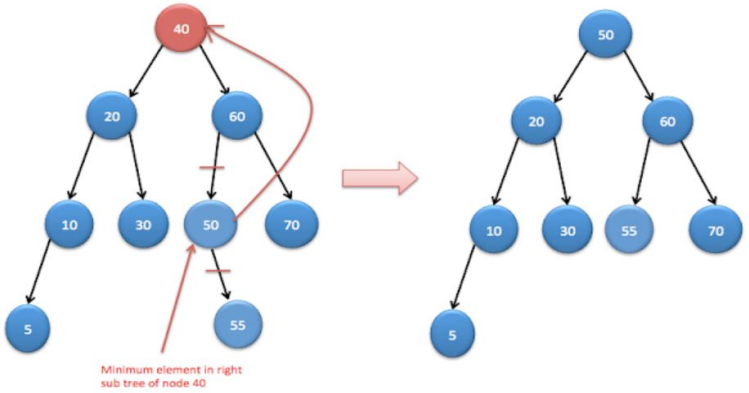


TABELLE DI HASH

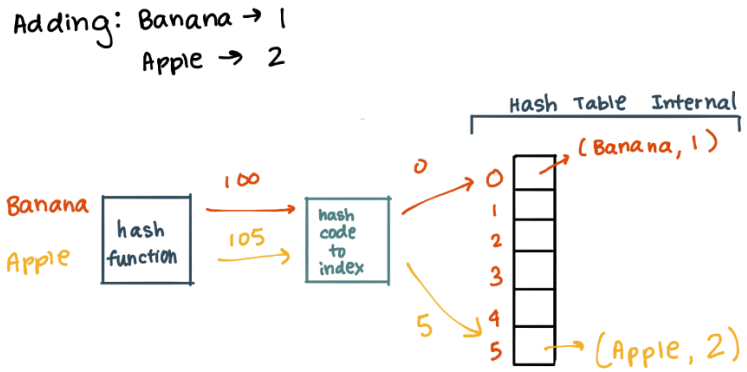
Per accedere ad una cella dell'array associata ad una chiave, esistono delle funzioni (di hash) che trasformano attraverso un algoritmo la chiave in un numero.

- Una buona funzione di hash dovrebbe distribuire gli elementi in maniera uniforme all'interno dell'array. Ad esempio, se ho 100 chiavi e 10 celle, una buona funzione di hash associa 10 chiavi ad ogni cella;
- La funzione di hash dev'essere calcolabile in tempo costante;
- Se volessimo aggiungere un elemento nella tabella in una posizione già occupata si crea una collisione. Per risolverla, si creano delle liste semplici (di collisione o bucket) in cui vengono memorizzati tutti gli elementi associati alla stessa cella, all'interno della quale si inserisce un puntatore alla lista;
- Una funzione di hash è perfetta quando non ci sono collisioni;
- Una funzione di hash perfetta deve essere iniettiva e calcolabile in tempo costante.

Inserimento:
La chiave viene trasformata in un indice della tabella attraverso una funzione hash. Dopo di che, se la cella nell'indice è vuota, inserisco la coppia chiave–valore nella posizione dell'indice; se la cella punta ad un bucket, comparo ogni elemento con quelli della lista e se la nuova coppia chiave-valore non è ripetuta, la inserisco come ultimo elemento.

Cancellazione:
La chiave viene trasformata in un indice della tabella attraverso una funzione hash. Dopo di che, comparo la coppia chiave–valore con gli elementi del bucket. Quando la trovo elimino l'elemento e collego l'elemento precedente all'elemento eliminato con il suo successivo.

Esempio di hash table:

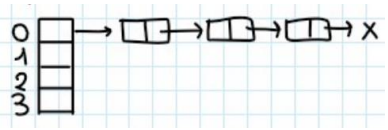


Esempi di domande:

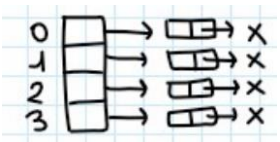
- **Quali sono le due principali buone proprietà che una funzione di hash deve avere?**
  - La prima proprietà è quella di essere calcolabile in tempo costante; la seconda proprietà è quella dell'essere uniforme, ossia distribuire le chiavi nella tabella in maniera equa.
- **La funzione di hash h applicata alle chiavi nello schema disegnato gode della prima di queste due proprietà?**
  - La prima proprietà è soddisfatta perché la nostra funzione di hash in richiede operazioni aritmetiche semplici (modulo, somma, moltiplicazione, divisione, ecc.), quindi calcolabili in tempo costante. Inoltre, in questo caso la funzione viene sempre calcolata su chiavi di 4 cifre decimali.
- **La funzione di hash h applicata alle chiavi nello schema disegnato gode della seconda di queste due proprietà?**
  - La seconda proprietà non viene rispettata perché alcune chiavi generano più frequentemente lo stesso valore rispetto ad altri, quindi l'uniformità non è possibile.
- **Ipotizzando che la funzione di hash abbia tutte le buone proprietà, qual è la complessità dell'operazione di cancellazione di un elemento nel caso peggiore?**

- In questo caso, il caso peggiore ha complessità $\vartheta(^n/_m)$. Questo perché, se la funzione di hash ha tutte le buone proprietà, allora ogni bucket ha $^n/_m$ elementi. Quindi se la ricerca avviene in $\vartheta(^n/_m)$ e la cancellazione avviene in $\vartheta(1)$, allora la delete è $\vartheta(^n/_m * 1) = \vartheta(^n/_m)$.

- **Sotto quali condizioni l'inserimento (o la cancellazione) ha complessità $\vartheta(n)$?**
  - L'inserimento (o la cancellazione) ha complessità $\vartheta(n)$ quando la funzione di hash smista tutte le chiavi in un solo bucket.



- **Sotto quali condizioni l'inserimento (o la cancellazione) ha complessità $\vartheta(^n/_m)$?**
  - L'inserimento ha complessità $\vartheta(^n/_m)$ quando la funzione di hash distribuisce le chiavi in maniera uniforme.



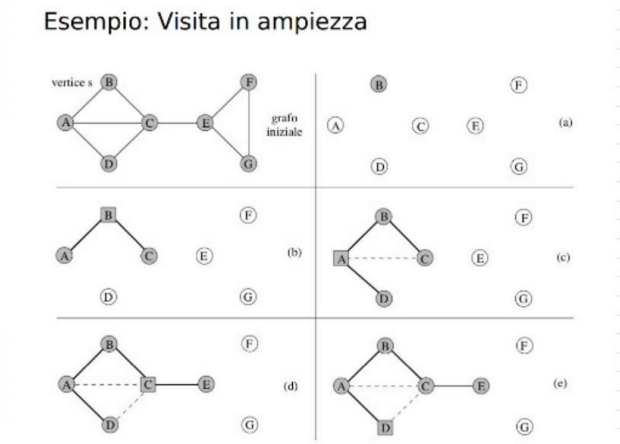
- **Sotto quali condizioni l'inserimento (o la cancellazione) ha complessità $\vartheta(1)$?**
  - l'inserimento ha complessità $\vartheta(1)$ quando la funzione di hash è iniettiva, o quando non ci sono collisioni.

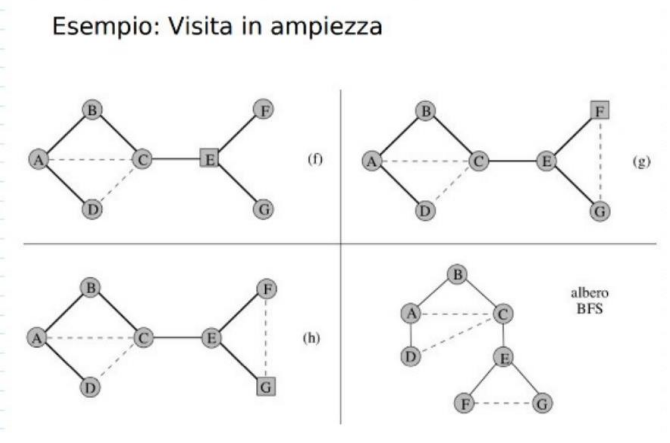
GRAFI

BFS, Breadth-First Search:

Corrisponde ad una visita a livelli di un albero; la visita procede in ampiezza sul grafo a partire dai vertici incontrati. Utilizza come struttura una coda che registra l'ordine in cui i vertici sono visitati dall'algoritmo.

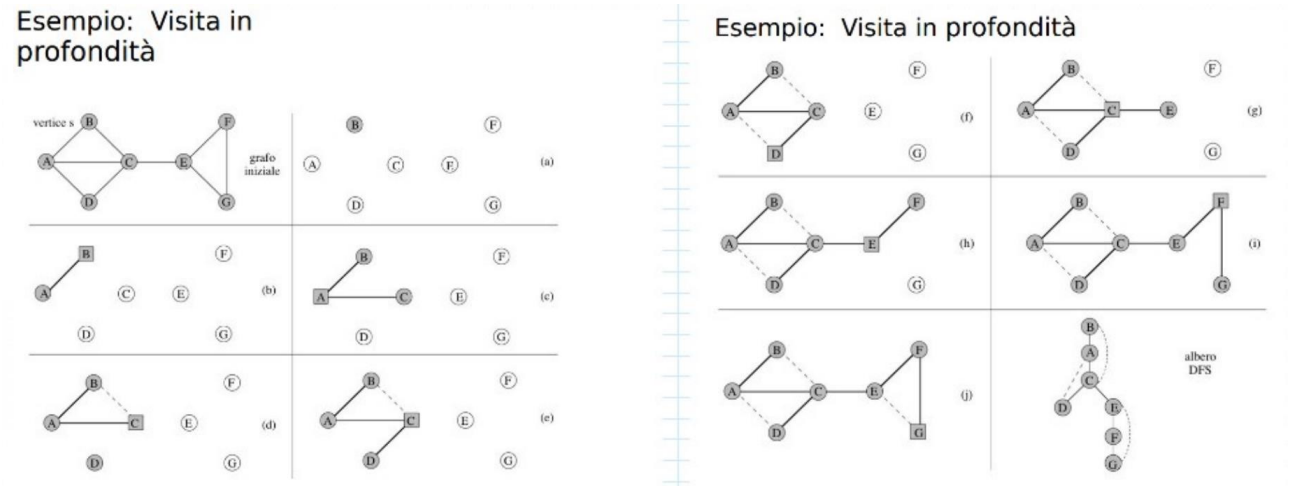
Esempio di BFS:





DFS, Depth-First Search:
Corrisponde alla visita in profondità di un albero. Utilizza uno stack (al posto della coda).

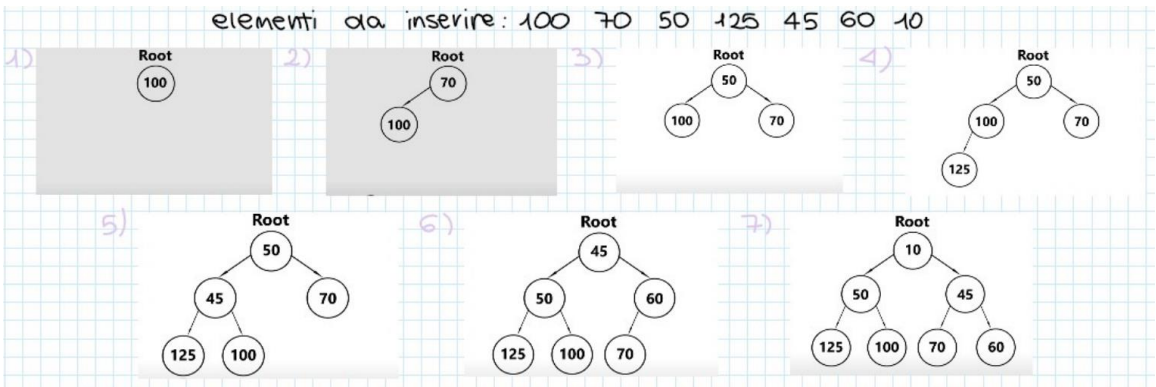
Esempio di DFS:



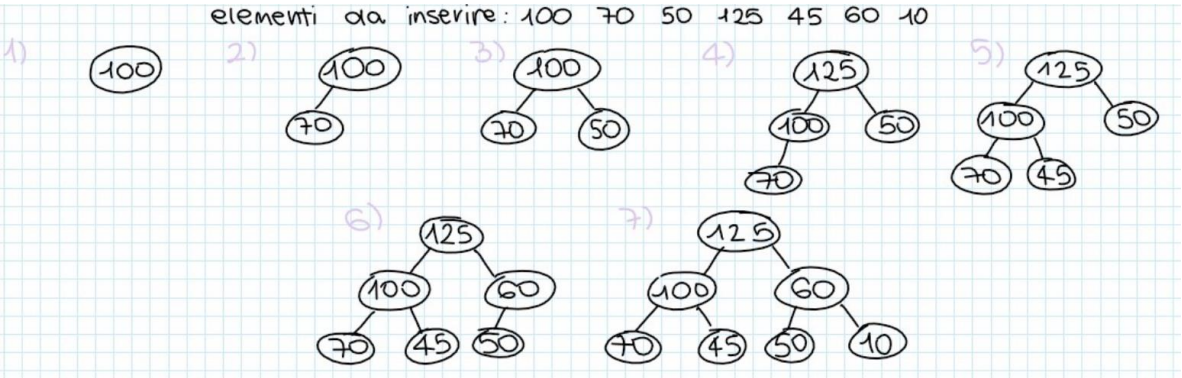
HEAP BINARI

Inserimento:
Min-heap: quando inserisco un elemento in un heap di tipo min, comparo l'elemento appena inserito (come ultimo elemento nell'array che codifica l'heap) con il padre: se è minore del padre, eseguo uno swap. La comparazione e lo swap continuano ricorsivamente fino alla radice.
Max-heap: quando inserisco un elemento in un heap di tipo max, comparo l'elemento appena inserito (come ultimo elemento nell'array che codifica l'heap) con il padre: se è maggiore del padre, eseguo uno swap. La comparazione e lo swap continuano ricorsivamente fino alla radice.

Esempio con Min-heap:



Esempio con Max-heap:



Cancellazione:

Per eseguire una delete dobbiamo seguire questi passaggi:

1. Salvare l'elemento nel nodo della radice;
2. Eseguire uno swap tra il nodo della radice e l'ultimo elemento nell'array che codifica l'heap;
3. Se l'heap è di tipo:

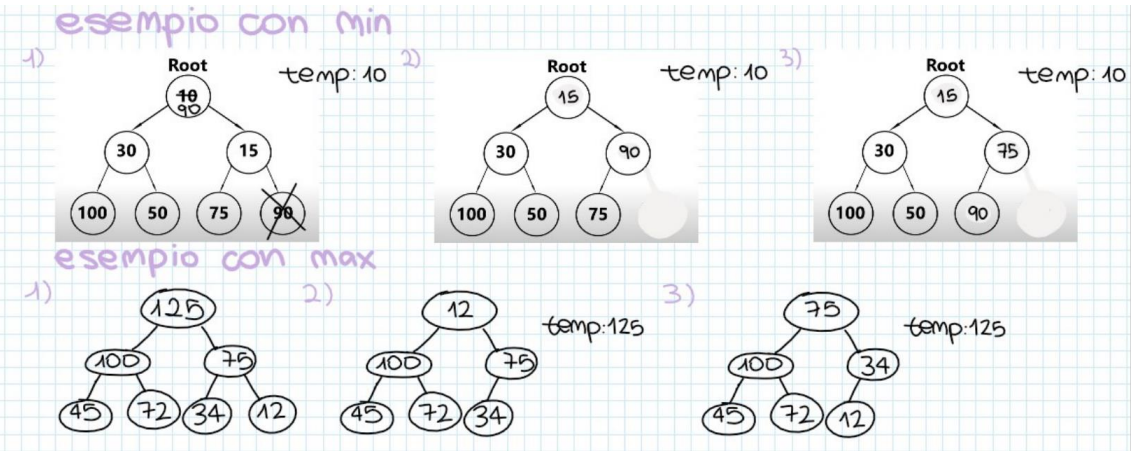
   Min:

   - Se uno dei due figli è più piccolo del padre, swappo;
   - Se entrambi i figli sono più piccoli del padre, eseguo lo swap con il più piccolo dei due.

   Max:

   - Se uno dei due figli è più grande del padre, swappo;
   - Se entrambi i figli sono più grandi del padre, eseguo lo swap con il più grande dei due.
4. Ritorna l'elemento salvato.

Esempi di delete:



Complessità:

Inserimento:
Caso migliore: il caso migliore avviene quando l'elemento da inserire viene posizionato in una posizione che non richiede alcuna operazione di risalita. Questo succede quando l'elemento viene inserito come figlio di un nodo che è già più piccolo in un min-heap, o già più grande in un max-heap. La complessità è quindi $\vartheta(1)$.
Caso peggiore: il caso peggiore si verifica quando l'elemento inserito deve risalire fino alla radice dell'heap. Questo succede quando l'elemento è più piccolo (in min-heap) o più grande (in max heap) di tutti gli altri elementi lungo il percorso dall'ultima posizione (ultimo elemento nell'array che codifica l'heap) fino alla radice. La risalita può richiedere un massimo di $\log n$ operazioni, dove n è il numero di elementi nell'heap. La complessità è quindi $\vartheta(\log n)$.

Cancellazione:
Caso migliore: il caso migliore si verifica quando l'elemento spostato alla radice (per eseguire la delete) è già in posizione corretta, ovvero è più piccolo (in min-heap) o più grande (in max-heap) di entrambi i suoi figli. Un'altra situazione in cui si verifica è quando l'elemento da eliminare è l'ultimo elemento dell'heap e la cancellazione può essere completata semplicemente riducendo la dimensione dell'heap. La complessità è quindi $\vartheta(1)$.
Caso peggiore: il caso peggiore avviene quando l'elemento spostato alla radice (per eseguire la delete) deve scendere fino alla foglia più bassa per mantenere le proprietà dell'heap. In questo caso, la discesa può richiedere un massimo di $\log n$ operazioni, dove n è il numero di elementi nell'heap. La complessità è quindi $\vartheta(\log n)$.