# Capitolo 4
# Dizionari

Lucidi tratti da

P. Crescenzi · G. Gambosi · R. Grossi · G. Rossi

*Strutture di dati e algoritmi*

*Progettazione, analisi e visualizzazione*

Addison-Wesley, 2012

`http://algoritmica.org`

1. I lucidi sono utilizzabili dai soli docenti e se ne sconsiglia la distribuzione agli studenti: oltre al rischio di violare una qualche forma di copyright, il problema principale è che gli studenti studino in modo superficiale la materia senza il necessario approfondimento e la dovuta riflessione che la lettura del libro fornisce
2. Il simbolo [alvie] nei lucidi indica l'uso di ALVIE per visualizzare il corrispettivo algoritmo: per un proficuo rendimento dello strumento, conviene esaminare in anticipo la visualizzazione per determinare i punti salienti da mostrare a lezione (l'intera visualizzazione potrebbe risultare altrimenti noiosa)

# Dizionari

- **Universo U** delle chiavi
- Ipotesi: e $\in S$ ha il campo `e.chiave` $\in U$ e quello dei dati satellite `e.sat` (che dipende dall'applicazione)
- Operazioni di base su $S$ per una chiave $k \in U$:
  - **Ricerca**$(k)$ trova e tale che `e.chiave` $= k$ (`null` se non esiste)
  - **Inserisci**(e) esegue $S = S \cup \{\texttt{e}\}$ (ipotesi: chiavi distinte in $S$)
  - **Cancella**$(k)$ esegue $S = S - \{\texttt{e}\}$ dove `e.chiave` $= k$

  **Appartiene**$(k)$ definita come `Ricerca`$(k) \neq$ `null`

Dizionario **statico** se solo `Ricerca`, altrimenti **dinamico**

# Dizionario ordinato

- Per ogni coppia di chiavi $k, k' \in U$ vale $k \leq k'$ o $k' \leq k$
- Estensione agli elementi dell'insieme $S$:
  e $\leq$ f se e solo se e.chiave $\leq$ f.chiave
- Operazioni supportate:
  - **Predecessore**$(k)$ = e se e.chiave $= \max_{\mathtt{f} \in S}\{\mathtt{f.chiave} \leq k\}$
  - **Successore**$(k)$ = e se e.chiave $= \min_{\mathtt{f} \in S}\{\mathtt{f.chiave} \geq k\}$
  - **Intervallo**$(k, k') = \{\mathtt{e} \in S : k \leq \mathtt{e.chiave} \leq k'\}$
  - **Rango**$(k)$ = cardinalità di $\{\mathtt{e} \in S : \mathtt{e.chiave} \leq k\}$

# Liste doppie e dizionari

- Lista doppia `L` è un descrittore con tre campi
  - `L.inizio` (=`null` per la lista vuota)
  - `L.fine` (=`null` per la lista vuota)
  - `L.lunghezza` (=0 per la lista vuota)
- Ogni nodo `p` della lista ha tre campi (capitolo 3)
  - `p.pred`
  - `p.succ`
  - `p.dato`
    - `p.dato.chiave` è la chiave di ricerca dell'elemento nel nodo `p`
    - `p.dato.sat` sono gli eventuali dati satellite

# Inserimento in cima e in fondo alla lista

```
InsCima(l,e):
  p = NuovoNodo( );
  p.dato = e;
  lun = l.lunghezza;
  IF (lun == 0) {
    p.succ = p.pred = null;
    l.inizio = p;
    l.fine = p;
  } ELSE {
    p.succ = l.inizio;
    p.pred = null;
    l.inizio.pred = p;
    l.inizio = p;
  }
  l.lunghezza = lun + 1;
  RETURN l;
```

```
InsFondo(l,e):
  p = NuovoNodo( );
  p.dato = e;
  lun = l.lunghezza;
  IF (lun == 0) {
    p.succ = p.pred = null;
    l.inizio = p;
    l.fine = p;
  } ELSE {
    p.succ = null;
    p.pred = l.fine;
    l.fine.succ = p;
    l.fine = p;
  }
  l.lunghezza = lun + 1;
  RETURN l;
```

# Ricerca e cancellazione con complessità lineare

```
Ricerca( lista, k ):
  p = lista.inizio;
  WHILE ((p != null) && (p.dato.chiave != k))
    p = p.succ;
  RETURN p;
Canc( lista, k ):
  p = Ricerca( lista, k );
  IF (p != null) {
    IF (lista.lunghezza == 1) {
      lista.inizio = lista.fine = null;
    } ELSE IF (p.pred == null) {
      p.succ.pred = null; lista.inizio = p.succ;
    } ELSE IF (p.succ == null) {
      p.pred.succ = null; lista.fine = p.pred;
    } ELSE {
      p.succ.pred = p.pred; p.pred.succ = p.succ;
    }
    lista.lunghezza = lista.lunghezza - 1;
  }
  RETURN lista;
```

# DIZIONARI CON FUNZIONI HASH

- Funzione **hash**: universo $U \Rightarrow$ intervallo $[0, m-1]$
- Esempi, dove $k$ è vista come sequenza di bit:
  *tradizionali:*
  - $\texttt{Hash}(k) = k \,\%\, m$ (modulo un numero primo $m$)
  - $\texttt{Hash}(k) = k_0 \oplus k_1 \oplus \cdots \oplus k_{s-1}$ iterativa (divide $k$ in blocchi $k_i$)

  *crittografiche sicure iterative (RSA e NSA)*:
  - $\texttt{Hash}(k) = \text{MD5}(k) \,\%\, m$ (dove $m$ è primo minore di $2^{128}$)
  - $\texttt{Hash}(k) = \text{SHA-1}(k) \,\%\, m$ (dove $m$ è primo minore di $2^{160}$)
  - $\texttt{Hash}(k) = \text{SHA-2}(k) \,\%\, m$ (dove $m$ è primo minore di $2^{512}$)
- **Collisione**: $k \neq k'$ ma $\texttt{Hash}(k) = \texttt{Hash}(k')$

# Applicazione P2P

- Nei sistemi *peer-to-peer* (BitTorrent, FreeNet, Gnutella, E-Mule e così via) i file sono condivisi attraverso sistemi distribuiti di calcolatori (*peer*)
- `Hash`$(f)$ spedita al posto di $f$ per capire se il file è lo stesso su due peer
- Scambio di blocchi:
  file $f$ diviso in blocchi $f_0, f_1, \ldots, f_{s-1}$ e le richieste per un blocco $f_i$ vengono gestite utilizzando SHA-$1(f_i)$

# Dizionari: tabelle hash

- Memorizzano un insieme $S$ di $n$ elementi utilizzando le funzioni hash (*hash map*)
- È un array associativo, in cui gli elementi e $\in S$ sono indirizzati utilizzando `e.chiave` come indice
- Mondo ideale: **hash perfetto**, senza collisioni in $S$
  - $\texttt{tabella}[h] = 1$ se e solo se e $\in S$ e $h = \texttt{Hash}(\texttt{e.chiave})$
  - $O(1)$ tempo al caso pessimo per la ricerca!
  - $O(1)$ tempo medio ammortizzato per inserimento e cancellazione
  - nessuna contraddizione con il limite $\Omega(\log n)$ di confronti

# Tabelle hash in pratica (1)

- Gestione delle collisioni: **liste di trabocco**
- Idea per una data funzione $\texttt{Hash} : U \Rightarrow [0, m-1]$:
  array di $m$ liste doppie in cui la lista $h$ contiene l'elemento e $\in S$ tale che $\texttt{Hash}(\texttt{e.chiave}) = h$
- Ricerca con chiave $k$: scandisci la lista $h = \texttt{Hash}(k)$ e cerca, in tale lista, gli elementi e tali che $\texttt{e.chiave} = k$
- Inserimento e cancellazione di un elemento e: opera sulla lista $h = \texttt{Hash}(\texttt{e.chiave})$

[alvie]

# Codice per tabelle hash con liste di trabocco

```
Ricerca( k ):
  h = Hash(k);
  p = tabella[h].Ricerca( k );
  IF (p != null) RETURN p.dato ELSE RETURN null;
Inserisci( e ):
  IF (Ricerca( e.chiave ) == null) {
    h = Hash( e.chiave );
    tabella[h].Insfondo( e );
  }
Cancella( k ):
  IF (Ricerca( k ) != null) {
    h = Hash(k);
    tabella[h].Cancella( k );
  }
```

- Caso pessimo: $O(n)$ tempo
- Ipotesi: hash distribuisce in modo uniforme e casuale gli $n$ elementi di $S$ nelle m liste
- Lunghezza media di una lista è $O(n/m)$ dove $n/m = \alpha$ è chiamato **fattore di caricamento**
- Costo medio delle operazioni è $O(1 + \alpha) = O(1)$ se manteniamo l'invariante che $m$ è circa il doppio di $n$ (vedi array a dimensione variabile del capitolo 2)

# Tabelle hash in pratica (2)

- Gestione delle collisioni: **indirizzamento aperto**
- Idea: array di $m$ posizioni ($m > n$) dove cerchiamo la prima posizione vuota (contenente `null`) a partire dalla posizione $h = \texttt{Hash}(\texttt{e.chiave})$
- Sequenza di funzioni $\texttt{Hash}[i]$ $(0 \leq i \leq m-1)$ tale che $\texttt{Hash}[0](k)$, $\texttt{Hash}[1](k)$, ..., $\texttt{Hash}[m-1](k)$ è una permutazione di 0, 1, ..., $m-1$ per ogni chiave $k \in U$
- Esempi, dove $\texttt{Hash}$ e $\texttt{Hash}'$ sono date per $m$ primo:
  - $\texttt{Hash}[i](k) = \big(\texttt{Hash}(k) + i\big) \,\%\, m$ (scansione **lineare**)
  - $\texttt{Hash}[i](k) = \big(\texttt{Hash}(k) + ai^2 + bi + c\big) \,\%\, m$ (scansione **quadratica**)
  - $\texttt{Hash}[i](k) = \big(\texttt{Hash}(k) + i \times \texttt{Hash}'(k)\big) \,\%\, m$ (scansione basata su **hash doppio**)

# Codice per tabelle hash con indirizzamento aperto

```
Ricerca( k ):
  FOR (i = 0; i < m; i = i+1) {
    h = Hash[i](k);
    IF (tabella[h] == null) RETURN -1;
    IF (tabella[h].chiave == k) RETURN tabella[h];
  }
Inserisci( e ):
  IF (Ricerca( e.chiave ) == null) {
    i = -1;
    DO {
      i = i+1;
      h = Hash[i]( e.chiave );
      IF (tabella[h] == null) tabella[h] = e;
    } WHILE (tabella[h] != e);
  }
```

# Analisi

- Caso pessimo $O(n)$ tempo
- Ipotesi: per ogni chiave $k$, $\texttt{Hash}[0](k)$, ..., $\texttt{Hash}[m-1](k)$ è una delle $m!$ permutazioni in modo uniforme e casuale
- $T(n, m)$ = numero medio di accessi effettuati per inserire un'ulteriore chiave in una tabella di $m > n$ posizioni
- $T(0, m) = 1$ (inseriamo al primo colpo)
- $T(n, m)$ per $n > 0$: $n$ volte su $m$ la posizione è occupata ⇒ effettuiamo ulteriori $T(n-1, m-1)$ accessi, altrimenti è vuota e facciamo un solo accesso ⇒ la media pesata è

$$\frac{m-n}{m} \times 1 + \frac{n}{m} \times (1 + T(n-1, m-1))$$

# Analisi

Quindi

$$T(n,m) = \begin{cases} 1 & \text{se } n = 0 \\ 1 + \frac{n}{m} T(n-1, m-1) & \text{altrimenti} \end{cases}$$

Per induzione,

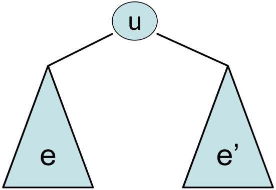
$$T(n,m) = 1 + \frac{n}{m} T(n-1, m-1) < 1 + \frac{n}{m} \times \frac{m}{m-n} = \frac{m}{m-n}$$

Da cui deriviamo

$$T(n,m) \leq \frac{m}{m-n} = (1-\alpha)^{-1} = O(1)$$

# Dizionario con albero binario di ricerca

- Utilizzato in molti contesti (per esempio, la traduzione da indirizzi logici a indirizzi fisici nella memoria virtuale)
- Proprietà: $\texttt{e} \in T(\texttt{u.sx}) \leq \texttt{u.dato.chiave} \leq \texttt{e}' \in T(\texttt{u.dx})$




- Visita simmetrica $\Rightarrow$ sequenza ordinata delle chiavi

# Ricerca con una chiave $k$

Ricorsione con tre casi (come la ricerca binaria):

- ($k$ = chiave dell'elemento in u) $\Rightarrow$ restituisci tale elemento
- ($k <$ chiave dell'elemento in u) $\Rightarrow$ cerca in $T(\text{u.sx})$
- ($k >$ chiave dell'elemento in u) $\Rightarrow$ cerca in $T(\text{u.dx})$

```
Ricerca( u, k ):
  IF (u == null) RETURN null;
  IF (k == u.dato.chiave) {
    RETURN u.dato;
  } ELSE IF (k < u.dato.chiave) {
    RETURN Ricerca( u.sx, k );
  } ELSE {
    RETURN Ricerca( u.dx, k );
  }
```

$O(h)$ tempo dove $h$ = altezza dell'albero
[alvie]

# Inserimento di un elemento e

Simile alla ricerca di $k =$ e.chiave
Arriva a un riferimento `null` che va sostituito con la foglia contenente e

```
Inserisci( u, e ):
  IF (u == null) {
    u = NuovoNodo();
    u.dato = e;
    u.sx = u.dx = null;
  } ELSE IF (e.chiave < u.dato.chiave) {
    u.sx = Inserisci( u.sx, e );
  } ELSE IF (e.chiave > u.dato.chiave) {
    u.dx = Inserisci( u.dx, e );
  }
  RETURN u;
```

La ricorsione aiuta ad agganciare la foglia al padre
$O(h)$ tempo dove $h =$ altezza dell'albero
[alvie]

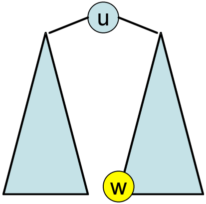
# Cancellazione dell'elemento con chiave $k$ in $O(h)$ tempo

Caso semplice: il nodo u è una foglia oppure ha un solo figlio

```
Cancella( u, k ):
  IF (u != null) {
    IF (u.dato.chiave == k) {
      IF (u.sx == null) {
        u = u.dx;
      } ELSE IF (u.dx == null) {
        u = u.sx;
      } ELSE {
        ...
      }
    } ELSE IF (k < u.dato.chiave) {
      u.sx = Cancella( u.sx, k );
    } ELSE IF (k > u.dato.chiave) {
      u.dx = Cancella( u.dx, k );
    }
  }
  RETURN u;
```

# Cancellazione dell'elemento con chiave $k$ in $O(h)$ tempo

Caso difficile: nodo u con due figli
$\Rightarrow$ sostituisci il suo elemento con il suo successore $w$ ($=$ minimo in $T(\mathrm{u.dx})$) e cancella tale successore (accade una sola volta)




```
w = MSA( u.dx );
u.dato = w.dato;
u.dx = Cancella( u.dx, w.dato.chiave );
```

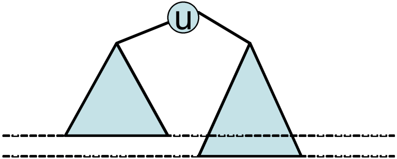
```
MSA( u ):
  WHILE (u.sx != null)
    u = u.sx;
  RETURN u;
```

# Caso pessimo $h = \Theta(n)$

- La forma dell'albero dipende dall'ordine d'inserimento delle chiavi
- Esempio: chiavi inserite in ordine crescente o decrescente
- Se le chiavi sono inserite in ordine casuale, l'albero risultante ha altezza logaritmica in media
- Vediamo come garantire altezza logaritmica al caso pessimo

# Albero 1-bilanciato

- $h(\mathtt{u})$ = altezza del sottoalbero $T(\mathtt{u})$ radicato in $\mathtt{u}$ (convenzione $h(\mathtt{null}) = -1$ come nel capitolo 4)
- Nodo $\mathtt{u}$ è **1-bilanciato** se $|h(\mathtt{u.sx}) - h(\mathtt{u.dx})| \leq 1$




- Un albero è 1-bilanciato se lo sono **tutti** i suoi n nodi

  *Albero 1-bilanciato* $\Rightarrow$ *altezza* $h = O(\log n)$

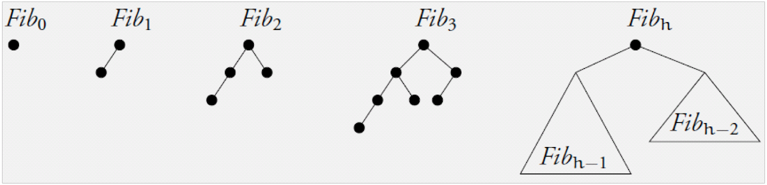
# Dimostrazione

1. **Albero di Fibonacci** $F_h$ di altezza $h$ con $n_h$ nodi
2. Vale $n_h \geq c^h$ per una costante $c > 1$
3. Ogni albero 1-bilanciato di altezza $h$ con $n$ nodi soddisfa $n \geq n_h$
4. $n \geq n_h \geq c^h \Rightarrow h = O(\log n)$

Vediamo i primi 3 punti (il quarto segue)

# Alberi di Fibonacci

Definizione ricorsiva sull'altezza $h$ (sono 1-bilanciati):



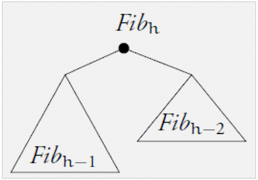

Vale $n_h = n_{h-1} + n_{h-2} + 1$ (da qui il nome di Fibonacci) e la relazione $n_h = F_{h+3} - 1$ con i numeri di Fibonacci $F_h$

| $h$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $n_h$ | 1 | 2 | 4 | 7 | 12 | 20 | 33 | 54 | 88 | 143 | 232 | 376 | 609 |
| $F_h$ | 0 | 1 | 1 | 2 | 3 | 5 | 8 | 13 | 21 | 34 | 55 | 89 | 144 |

Poiché

$$F_h = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}, \quad \text{dove } \phi = \frac{1+\sqrt{5}}{2} \approx 1,6180339\cdots$$

ne deduciamo che $F_h > c^h$ e quindi $n_h = F_{h+3} - 1 \geq c_h$
Inoltre, togliendo un nodo da $Fib_h$, o lo rendiamo di altezza $h-1$ oppure non è più 1-bilanciato




**$\mathbf{Fib_{h-1}}$ e $\mathbf{Fib_{h-2}}$ hanno il minimo numero di nodi $\Rightarrow$ $\mathbf{Fib_h}$ ha il minimo numero di nodi**

# Alberi AVL

- AVL = alberi binari di ricerca che sono 1-bilanciati
- Nodo `u` mantiene $h(\mathtt{u})$ nel campo `u.altezza`
  [alvie]
- Ricerca identica a quella degli alberi binari di ricerca
- Inserimento identico fino all'inserimento della foglia $\Rightarrow$ necessita della ristrutturazione dei nodi lungo il cammino dalla radice fino a quella foglia

# Inserimento in AVL

- Dopo la creazione della foglia $f$, cerca il suo nodo critico = minimo antenato $\mathtt{u}$ di $f$ che non è più 1-bilanciato
- Aggiorna anche i campi altezza degli antenati di $f$
- Usa la ricorsione per percorrere tale cammino a ritroso

# Inserimento di un elemento e in un AVL

Quando arriva a un puntatore `null`, lo sostituisce con la foglia $f$ contenente e

```
Inserisci( u, e ):
  IF (u == null) {
    RETURN f = NuovaFoglia( e );
  } ELSE IF (e.chiave < u.dato.chiave) {
    ...
  } ELSE IF (e.chiave > u.dato.chiave) {
    ...
  }
  ...
NuovaFoglia( e ):
  u = NuovoNodo();
  u.dato = e;
  u.altezza = 0;
  u.sx = u.dx = null;
  RETURN u;
```

# Inserimento di un elemento e in un AVL

La ricorsione aiuta ad agganciare la radice del sottoalbero (eventualmente modificato) con suo padre

```
Inserisci( u, e ):
  IF (u == null) {
    ...
  } ELSE IF (e.chiave < u.dato.chiave) {
    u.sx = Inserisci( u.sx, e );
    ...
  } ELSE IF (e.chiave > u.dato.chiave) {
    u.dx = Inserisci( u.dx, e );
    ...
  }
  ...
```

L'altezza viene ricalcolata usando quella dei figli (già aggiornata per induzione)

```
Inserisci( u, e ):
  IF (u == null) {
    ...
  } ELSE IF (e.chiave < u.dato.chiave) {
    ...
  } ELSE IF (e.chiave > u.dato.chiave) {
    ...
  }
  u.altezza = max( Altezza(u.sx), Altezza(u.dx) ) + 1;
  RETURN u;
```

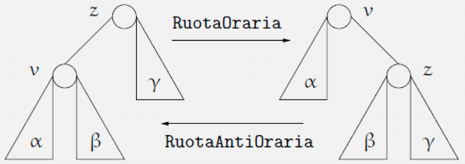
```
Altezza( u ):
  IF (u == null) {
    RETURN -1;
  } ELSE {
    RETURN u.altezza;
  }
```

# Inserimento di un elemento e in un AVL

Ribilanciamento del nodo critico (segue)

```
Inserisci( u, e ):
  IF (u == null) {
    ...
  } ELSE IF (e.chiave < u.dato.chiave) {
    ...
    IF (Altezza(u.sx) - Altezza(u.dx) == 2) {
      IF (e.chiave>u.sx.dato.chiave) u.sx=RAO(u.sx);
      u = RO( u );
    }
  } ELSE IF (e.chiave > u.dato.chiave) {
    ...
    IF (Altezza(u.dx) - Altezza(u.sx) == 2) {
      IF (e.chiave<u.dx.dato.chiave) u.dx = RO(u.dx);
      u = RAO( u );
    }
  }
  ...
```

# Ribilanciamento: rotazioni

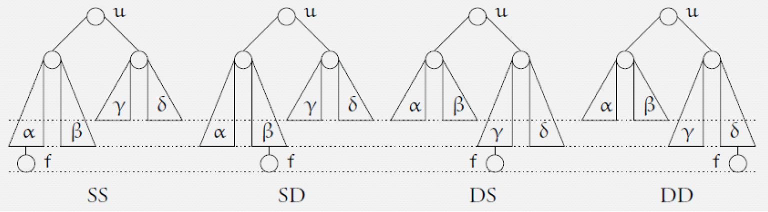



```
RuotaOraria( z ):
  v = z.sx;
  z.sx = v.dx;
  v.dx = z;
  z.altezza = max( Altezza(z.sx), Altezza(z.dx) ) + 1;
  v.altezza = max( Altezza(v.sx), Altezza(v.dx) ) + 1;
  RETURN v;
```

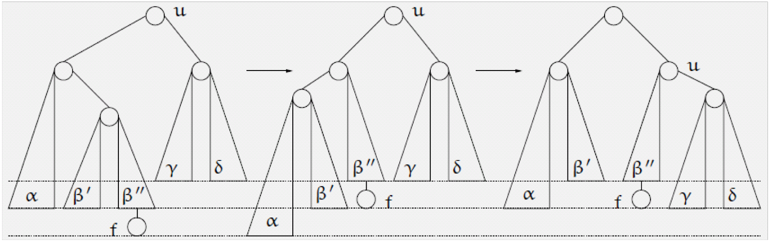
```
RuotaAntiOraria( v ):
  z = v.dx;
  v.dx = z.sx;
  z.sx = v;
  v.altezza = max( Altezza(v.sx), Altezza(v.dx) ) + 1;
  z.altezza = max( Altezza(z.sx), Altezza(z.dx) ) + 1;
  RETURN z;
```

$\alpha \leq v$
$v \leq \beta$
$\beta \leq z$
$z \leq \gamma$

# SS, SD, DS, DD: ROTAZIONI SU NODO CRITICO u




**rotazioni per il caso SD:**



# Inserimento e cancellazione

- Inserimento richiede $O(h) = O(\log n)$ tempo al caso pessimo [alvie]
- Cancellazione può essere realizzata in $O(\log n)$ tempo al caso pessimo (più casi da trattare)
- Semplice idea pratica (*global rebuilding*):
  - marcare i nodi come cancellati logicamente;
  - quando il numero di nodi marcati è circa la metà della dimensione dell'albero, ricostruire completamente l'AVL
  - costo $O(\log n)$ ammortizzato (ogni nodo inserito viene cancellato logicamente una sola volta)