AlberiRossoNeri

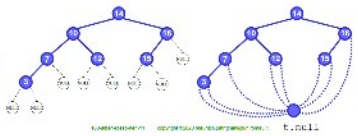
sabato 13 febbraio 2021 16:40

-Un albero binario porta a molti vantaggi quando è bilanciato

alberi binari di ricerca (complessità nel caso peggiore)		
operazione	sbilanciati	bilanciati
ricerca	$\Theta(n)$	$\Theta(\log n)$
inserimento	$\Theta(n)$	$\Theta(\log n)$
cancellazione	$\Theta(n)$	$\Theta(\log n)$

-Ha senso investire delle risorse per tenerlo bilanciato → Il bilanciamento risulta complesso

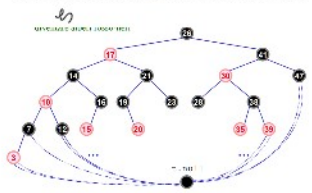
Per semplicità si usa una sentinella, ovvero un valore generico che Vale NULL, i puntatori a figli vuoti sono abbinati a un puntatore alla sentinella



ALBERO ROSSO-NERO

-Un albero rosso-nero è un albero binario di ricerca in cui :
1) Ogni nodo è rosso o nero
2) Radice t.root e la sentinella t.null sono nere
3) Un nodo rosso ha figli neri
4) Tutti cammini da radice a t.null hanno stesso numero di nodi neri

Altezza di un albero rosso-nero: lunghezza del cammino più lungo
↓
Corrisponde ad altezza + 1



k=4 h=6

L'albero deve essere un abr
-Non tutti gli abr posso essere colorati come rosso-neri

Con k = alberi neri in un cammino (4 nell'esempio)
-ogni cammino ha almeno k-1 archi
-cammino più lungo: 2(k-1)archi
-cammino più lungo è al massimo due volte quello più corto
-In (h/2)-1 è presente un sotto albero completo
-Profondità massima --> h = 2(k-1)

Alberi Rosso-neri e numero di nodi

$h' = \left(\frac{h}{2}\right) - 1$

-Ricordando che: nodi Albero completo = 2^{h+1} → 2^h Foglie + $2^h - 1$ nodi interni

$$n \geq 2^{h'+1} - 1 = 2^{\left(\frac{h}{2}-1\right)+1} - 1 = 2^{\frac{h}{2}} - 1$$

$$n + 1 \geq 2^{\frac{h}{2}}$$

$$\frac{h}{2} \leq \log(n+1)$$

$$h \leq 2\log(n+1)$$

• Dunque $h \in O(\log n)$

Foglie: $2^h + 2^h - 1$

$2 \cdot 2^h - 1$

$2^{h+1} - 1$

Sapendo inoltre che un albero binario completo ha h pari a Olog(n)

↓

Complessità totale di h = Θ(n)

Tutte le operazioni quindi di consultazione eseguibili in Θ(h) sugli abr
Sono eseguibili in Θ(log(n)) su un albero rosso-nero

-search
-minimum
-maximum

INSERT e DELETE

-Eseguibili in Θ(log n)
-Le operazione TREE_INSERT e TREE_DELETE, posso non conservare le proprietà dei rosso-neri → Vengono usate procedure che ripristinano le proprietà dei rosso-neri con costo Θ(log n)

RB_INSERT

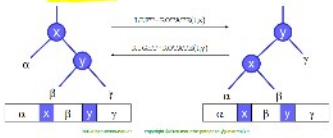
Inserimento di un nodo

```
RB_INSERT(t,new)          /* inserisce il nodo new nell'albero t */
1.  y = t.null
2.  x = t.root
3.  while x != t.null     // finché non sono arrivato a t.null
4.      y = x             // memo il padre y a cui appenderò new
5.      if new.key < x.key
6.          x = x.left
7.      else x = x.right
8.  new.p = y             // aggiorno il genitore di new
9.  if y == t.null        // se non deve diventare la radice.
10.     t.root = new      // aggiorno t.root
11. else if new.key < y.key
12.     y.left = new
13. else y.right = new
14. new.left = t.null; new.right = t.null
15. new.col = red         // il nuovo nodo inserito è rosso
16. RB_INSERT_FIXUP(t,new)  // ripristina le proprietà
```

→ Va bene qualunque strategia di inserimento, vanno però rispettate le ultime due righe

ROTAZIONI

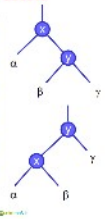
Usate per ripristinare le proprietà dei rosso-neri:
-non alterano i colori dei nodi
-l'albero rimane ABR
-eseguibile in Θ(1)



Esempio di rotazione a sinistra

Procedura LEFT_ROTATE

```
LEFT_ROTATE(t,x)
1.  y = x.right           // memo y
2.  x.right = y.left      // sposto b
3.  if y.left != t.null
4.      y.left.p = x
5.  y.p = x.p
6.  if x.p == t.null
7.      t.root = y
8.  else if x == x.p.left
9.      x.p.left = y
10.     else x.p.right = y
11. y.left = x
12. x.p = y
```



RIPRISTINO ALBERO ROSSO NERO

-Il nuovo nodo aggiunto è una foglia e ha colore rosso

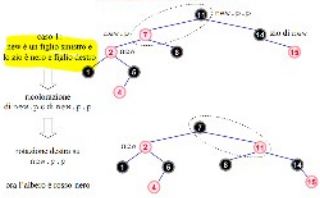
-Se albero vuoto: non rispettata la condizione che radice e sentinella siano nere
-Basta colorare la radice di nero

-La proprietà che un nodo sia rosso e suoi figli siano neri altrimenti è quella più possibile di violazione
-Situazione più complessa

VIOLAZIONE : NODO ROSSO CON FIGLIO ROSSO

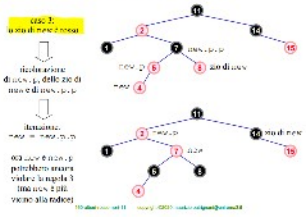
-Dopo RB-INSERT il nuovo nodo (new, che è sempre rosso) ha un genitore rosso
-chiamiamo il fratello del padre di new come "zio di new"
-lo zio esiste sempre, alla male è t.null
– sono possibili tre casi:
caso 1 : new è un figlio sinistro e lo zio è rosso figlio destro
caso 1' : new è un figlio destro e lo zio è nero e figlio sinistro
caso 2 : new è un figlio destro e lo zio è nero e figlio destro
caso 3 : lo zio è rosso e nero

Violazione: caso 1



Violazione: caso 2

Violazione: caso 3



-In questo caso new potrebbe violare la regola 3 con new.p

-Occorre rilanciare la procedure con new

-Caso peggiore quando si ha una sequenza di casi 3 fino alla radice

-La radice così però diventa rossa
-Basta ricolorarla
-Consegue che bisogna incrementare di uno il numero dei nodi in ogni cammino

• Ricordiamo i vincoli di un albero rosso-nero
1. ogni nodo è rosso o nero
2. la radice e la sentinella t.null sono nere
3. se un nodo è rosso entrambi i suoi figli sono neri
4. tutti i cammini che vanno dalla radice a t.null contengono lo stesso numero di nodi neri

COMPLESSITA' RB_INSERT_FIXUP

-Le violazioni del caso 1 e 2 si risolvono in Θ(1)
-Poiché h = Θ(log(n)), la violazione del caso 3 si può rilanciare al massimo Θ(log n)
-La complessità totale di RB_INSERT e RB_INSERT_FIXUP è Θ(log n)

CANCELLAZIONE IN ROSSO-NERO

-Analogo a RB_INSERT
-Prima si cancella un nodo come in TREE_DELETE degli abr
-Si ripristina la proprietà dei rosso-neri con RB_DELETE_FIXUP --> Che utilizza rotazioni e ricolorazioni

CONCLUSIONI

-Gli alberi rosso neri offrono una realizzazione di abr con ottime complessità nel caso peggiore
-Inserimento in Θ(log n)
-Cancellazione in Θ(log n)
-Ricerca in Θ(log n)