

Alberi rosso-neri

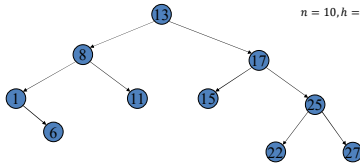
Algoritmi e strutture dati

Ugo de'Liguoro, Andras Horvath

1

Alberi di ricerca

$n = 10, h = 3$

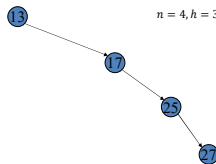


In un albero binario di ricerca con n nodi il costo delle operazione dipende dall'altezza dell'albero h , cioè dal numero di archi che compongono il ramo più lungo. Conviene avere rami che non hanno lunghezze troppo diverse.

2

Alberi di ricerca

$n = 4, h = 3$



In un albero binario di ricerca con n nodi il costo delle operazione dipende dall'altezza dell'albero h , cioè dal numero di archi che compongono il ramo più lungo. Conviene avere rami che non hanno lunghezze troppo diverse.

3

Alberi rosso-neri

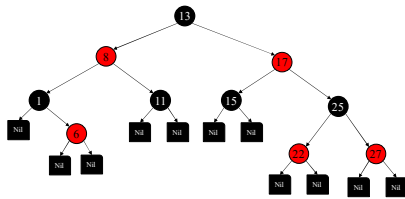
Def. Un *albero rosso-nero (R-N)* è un albero binario di ricerca aumentato, i cui vertici sono colorati di rosso o nero in modo che:

- (regola del **nero**) la radice e tutte le foglie (i Nil) sono nere
- (regola del **rosso**) se un nodo è rosso tutti i suoi figli sono neri
- (regola del **cammino**) per ogni nodo x tutti i cammini da x ad una foglia hanno lo stesso numero di nodi neri

$bh(x)$ = altezza nera di x
 = il numero di nodi neri su un ramo da x ad una foglia (x escluso)

4

Alberi rosso-neri (R-N)



5

Proprietà degli alberi R-N

Prop. L'altezza massima di un albero R-N con n nodi è $2 \log_2(n + 1)$.

Dim. Consideriamo un albero R-N con n nodi interni.

- dimostriamo che per ogni nodo x il sottoalbero con radice in x ha almeno $2^{bh(x)} - 1$ nodi. procediamo con induzione sull'altezza di x :
 - *caso base*: se $bh(x) = 0$ (x è una foglia) allora è vero perché $2^0 - 1 = 0$
 - se x non è foglia allora i figli hanno altezza nera $bh(x)$ oppure $bh(x) - 1$ secondo il colore che hanno
 - *passo induttivo*: secondo l'ipotesi induttiva ogni figlio ha almeno $2^{(bh(x)-1)} - 1$ nodi interni nel proprio sottoalbero
 - quindi almeno $2(2^{(bh(x)-1)} - 1) + 1 = 2^{bh(x)} - 1$ nodi interni nel sottoalbero con radice in x

6

Proprietà degli alberi R-N

Prop. L'altezza massima di un albero R-N con n nodi è $2 \log_2(n + 1)$.

- in un albero con altezza h , l'altezza nera della radice r soddisfa per via della regola del rosso

$$bh(r) \geq \frac{h}{2}$$

(perché lungo un cammino al massimo metà dei nodi può essere rosso)

- quindi:

$$\begin{aligned} n &\geq 2^{h/2} - 1 \\ n + 1 &\geq 2^{h/2} \\ \log_2(n + 1) &\geq \log_2(2^{h/2}) = h/2 \\ 2 \log_2(n + 1) &\geq h \end{aligned}$$

□

7

Proprietà degli alberi R-N

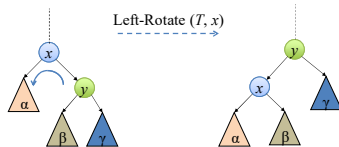
Prop. L'altezza massima di un albero R-N con n nodi è $2 \log_2(n + 1)$.

Quindi la ricerca in un albero R-N è $O(h) = O(\log n)$

Anche inserimento e cancellazione sono $O(h)$:
è possibile senza violare le regole R-N ?

8

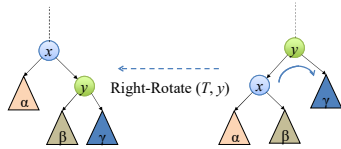
Rotazioni



Dopo una rotazione l'albero rimane di ricerca.

9

Rotazioni



Dopo una rotazione l'albero rimane di ricerca.

10

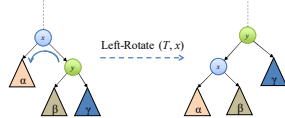
Rotazioni

LEFT-ROTATE(T, x)

```

y ← x.right
x.right ← y.left
if y.left ≠ nil then y.left.parent ← x
end if
y.parent ← x.parent
if x.parent = nil then T ← y
else
  if x = x.parent.left then x.parent.left ← y
  else x.parent.right ← y
end if
end if
y.left ← x
x.parent ← y

```



11

Inserimento

L' inserimento di x in T avviene in due fasi:

1. Inserimento di x rosso come per gli alberi di ricerca



2. Ripristino delle proprietà R-N con rotazioni e ricolorazioni

12

Inserimento

- x : il nuovo nodo
- $p = x.parent$ (padre)
- $g = p.parent$ (nonno)
- u = il fratello di p (zio)

13

Inserimento



Se l'inserimento è in radice basta cambiare il colore in nero.

14

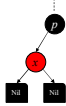
Inserimento

- i lucidi successivi elencano i vari casi
- per i casi 1, 2 e 3 assumiamo che p sia figlio sinistro
- per i casi 0 e 1 assumiamo che x sia figlio sinistro
- se non è così bisogna agire in modo «speculare»

15

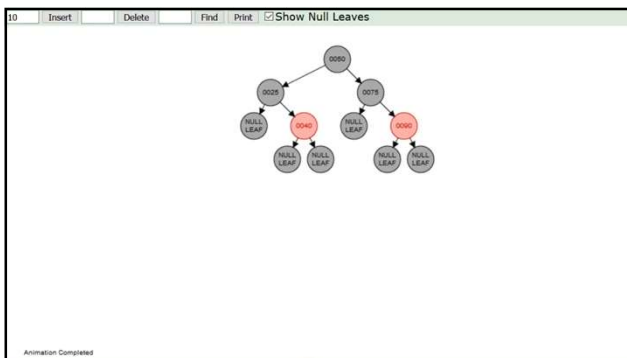
Inserimento

- Caso 0: il padre p è nero

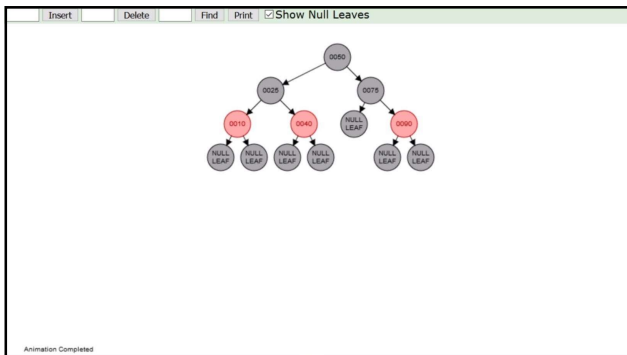


- altezza nera di p non cambia
- nessun regola viene violata, non bisogna fare niente

16



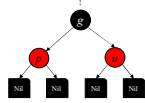
17



18

Inserimento

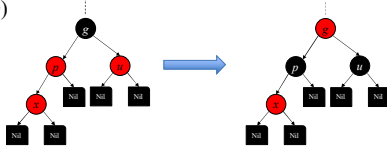
- Caso 1 a livello foglia prima dell'inserimento: lo zio u è rosso (e p è rosso, altrimenti sarebbe il caso 0)



19

Inserimento

- Caso 1: lo zio u è rosso (e p è rosso, altrimenti sarebbe il caso 0)

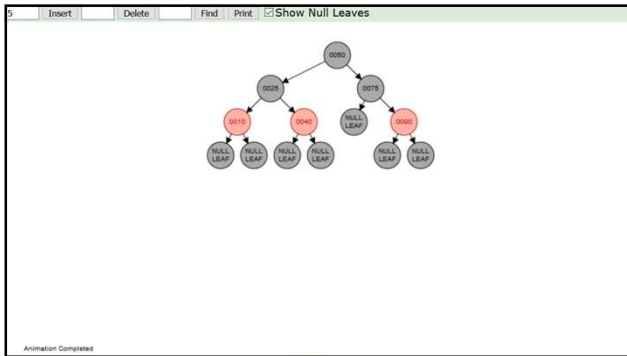


20

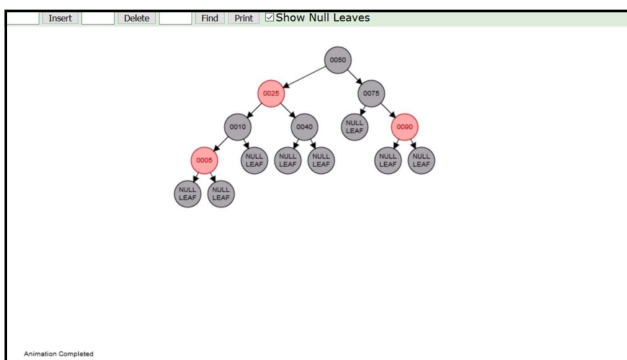
Inserimento

- Caso 1:
- dopo le modifiche
 - le regole localmente (da g in giù) sono rispettate
 - l'altezza nera del padre di g non cambia
- se il padre di g è nero allora le regole sono rispettate
- se il padre di g è rosso allora la regola del rosso è violata
- se g è la radice allora bisogna colorarlo di nero e l'albero è ok

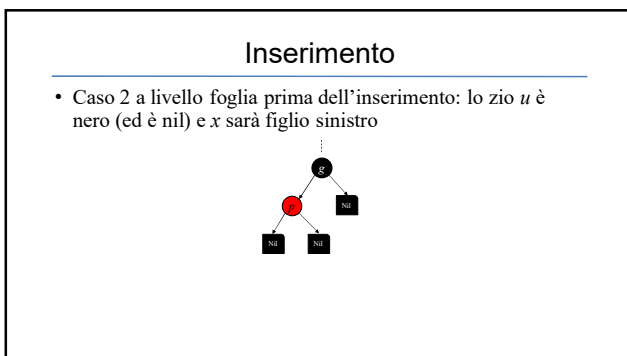
21



22



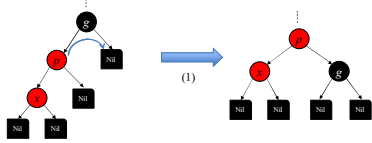
23



24

Inserimento

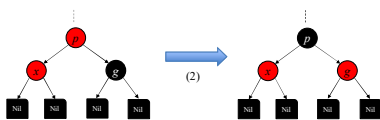
- Caso 2 a livello foglia: lo zio u è nero (ed è nil) e x è figlio sinistro



25

Inserimento

- Caso 2 a livello foglia: lo zio u è nero (ed è nil) e x è figlio sinistro

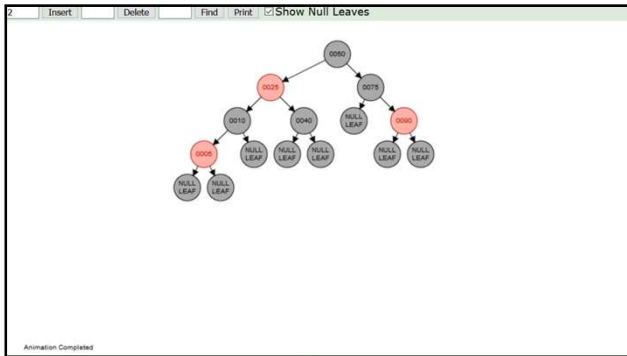


26

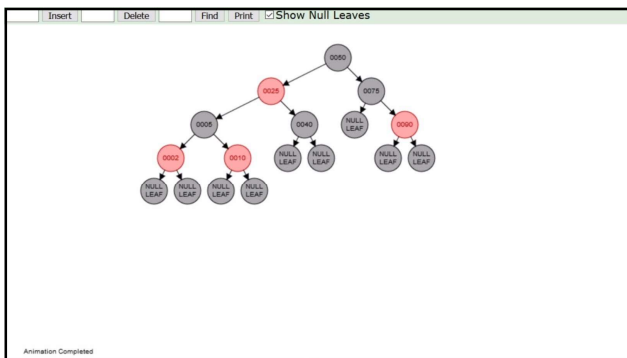
Inserimento

- Caso 2:
- dopo le modifiche
 - le regole localmente (da p in giù) sono rispettate
 - l'altezza nera del padre di p non cambia
 - visto che p diventa nero anche la regola di rosso viene rispettata
- l'albero è ok

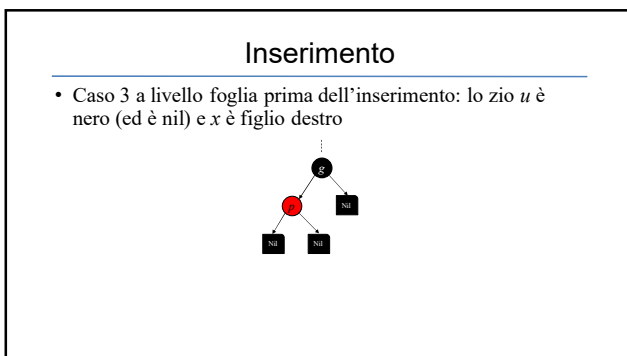
27



28



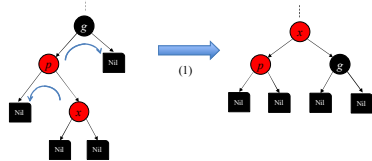
29



30

Inserimento

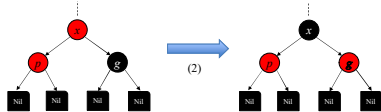
- Caso 3 a livello foglia: lo zio u è nero (ed è nil) e x è figlio destro



31

Inserimento

- Caso 3 a livello foglia: lo zio u è nero (ed è nil) e x è figlio destro

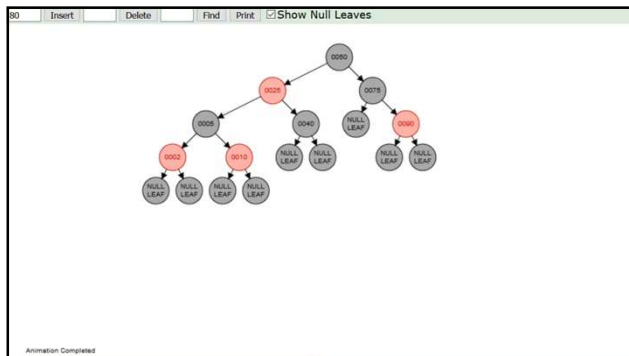


32

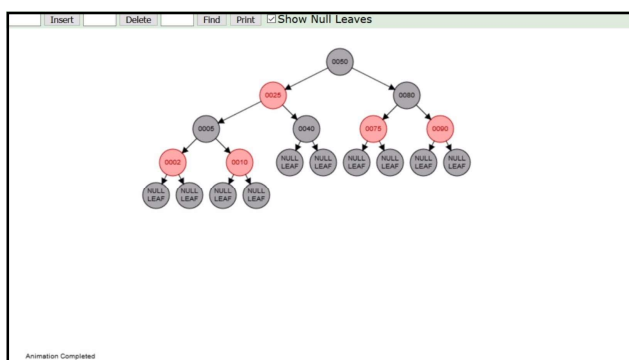
Inserimento

- Caso 3:
- dopo le modifiche
 - le regole localmente (da x in giù) sono rispettate
 - l'altezza nera del padre di x non cambia
 - visto che x diventa nero anche la regola di rosso viene rispettata
- l'albero è ok

33



34



35

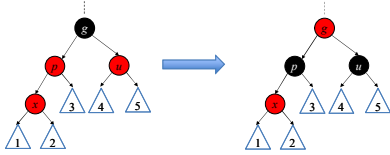
Inserimento

- nel caso 1 può capitare che la regola del rosso è violata se il padre di g è rosso
- se è così abbiamo i stessi tre casi da risolvere con la differenza che i *nil* sono sottoalberi non vuoti
- appena incontriamo il caso 2 o il caso 3 le modifiche mettono a posto l'albero

36

Inserimento

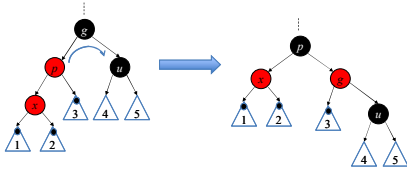
- Caso 1 a qualsiasi livello: lo zio u è rosso



37

Inserimento

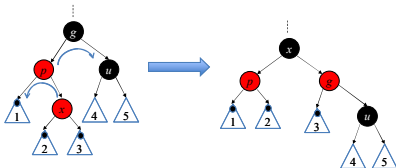
- Caso 2 a qualsiasi livello: lo zio u è nero e x è figlio sinistro



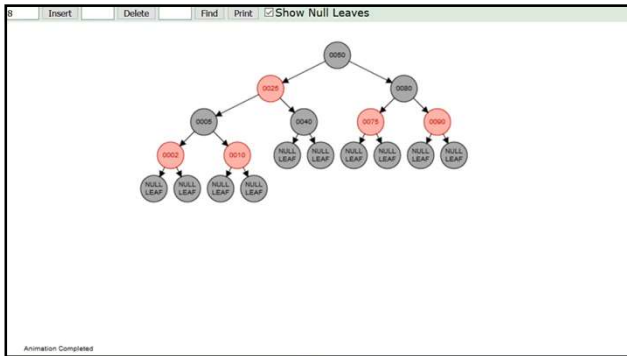
38

Inserimento

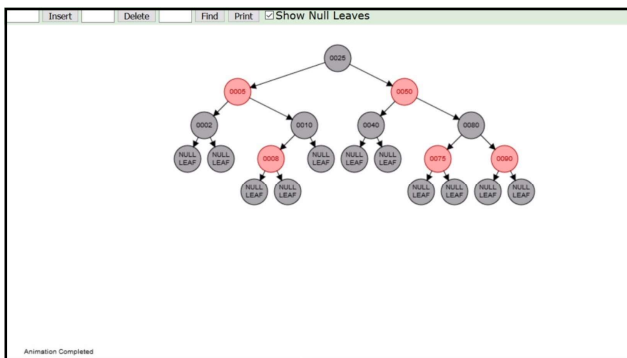
- Caso 3 a qualsiasi livello: lo zio u è nero e x è figlio destro



39



40



41



42

Inserimento in
pseudo codice

Alberi rosso-neri

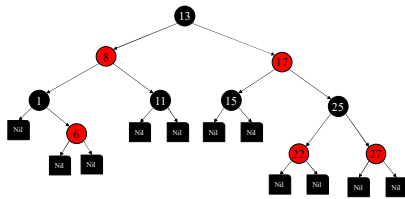
Def. Un *albero rosso-nero (R-N)* è un albero binario di ricerca aumentato, i cui vertici sono colorati di rosso o nero in modo che:

- (regola del **nero**) la radice e tutte le foglie (i Nil) sono nere
- (regola del **rosso**) se un nodo è rosso tutti i suoi figli sono neri
- (regola del *cammino*) per ogni nodo x tutti i cammini da x ad una foglia hanno lo stesso numero di nodi neri

Inserimento e cancellazione sono accompagnati da meccanismi tali da rendere l'albero un albero rosso-nero corretto.

43

Alberi rosso-neri (R-N)



44

Cancellazione

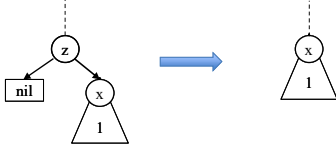
Come l'inserimento, anche la cancellazione avviene in due fasi:

1. Cancellazione come in un albero di ricerca ordinario
2. Ripristino delle regole per ricolorazioni/rotazioni

45

Cancellazione «normale»

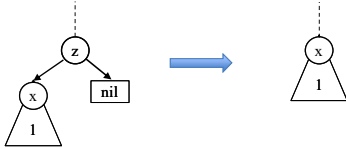
- Caso A: nodo da cancellare non ha figlio sinistro ma ha figlio destro



46

Cancellazione «normale»

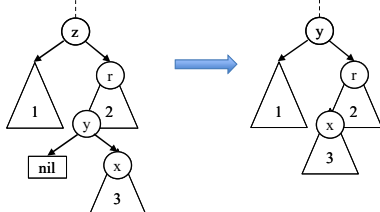
- Caso B: nodo da cancellare non ha figlio destro ma ha figlio sinistro



47

Cancellazione «normale»

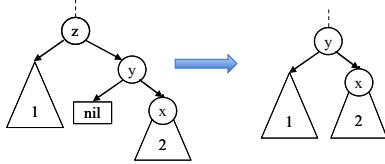
- Caso C: nodo da cancellare ha due figli e il suo successore non è suo figlio



48

Cancellazione «normale»

- Caso D: nodo da cancellare ha due figli e il suo successore è suo figlio



49

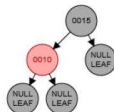
Cancellazione, fix-up

- caso A e B: z ha un figlio, x, che prende il suo posto
- caso C e D: z ha due figli; il suo successore, y, prende il suo posto e il figlio di y, x, prende il posto di y
- durante la cancellazione «normale»:
 - x mantiene il suo colore
 - y prende il colore di z
 - una variabile *lost-color* per memorizzare il colore del nodo effettivamente perso: nel caso A e B è il colore di z, nel caso C e D è il colore di y

50

Cancellazione, violazioni delle regole

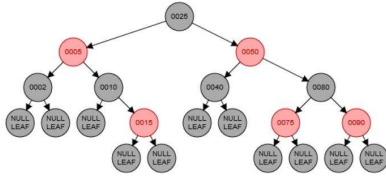
- la radice diventa rosso se cancelliamo il nodo 15
- la radice può diventare rosso



51

Cancellazione, violazioni delle regole

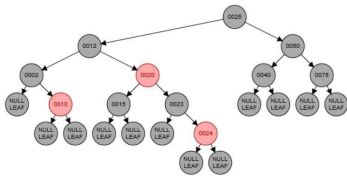
- se cancello il nodo 5, il nodo 10 prende il suo posto e il suo colore
- il nodo 15 rimane rosso e questo viola la regola del rosso
- x può essere rosso e può diventare figlio di un nodo rosso



52

Cancellazione, violazioni delle regole

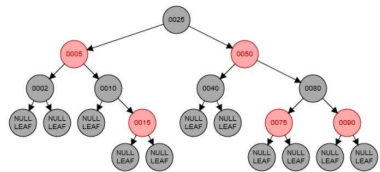
- se cancello il nodo 12, il nodo 15 viene tolto, regola del cammino si viola
- se il nodo tolto era nero allora i cammini che lo contenevano hanno altezza nera diminuita che può violare la regola del cammino



53

Cancellazione, fix-up, caso -1

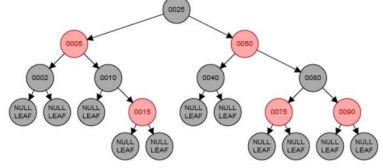
- canc. il nodo 50: sua etichetta diventa 75, nodo 75 sostituito da Nil
- Caso -1: il colore memorizzato in *lost-color* è rosso
- bisogna agire?
- no, l'albero è ok



54

Cancellazione, fix-up, caso 0

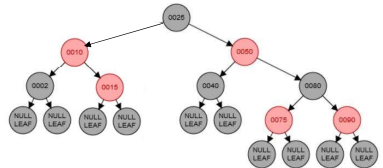
- can. il nodo 5 (z): la sua etichetta diventa 10 (y), 15 sale al suo posto (x)
- Caso 0: *lost-color* è nero e il nodo x è rosso
- bisogna agire?



55

Cancellazione, fix-up, caso 0

- la regola del cammino violata perché *lost-color* è nero
- può essere che x è la radice ed è rosso (regola del nero violata)
- può essere che x è rosso e suo padre anche (la regola del rosso violata)
- colora il 15 di nero
- basta colorare x di nero



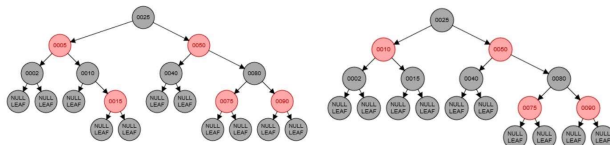
56

Cancellazione, fix-up, caso 0

- can. il nodo 5 (z): la sua etichetta diventa 10 (y), 15 sale al suo posto (x)
- basta colorare il nodo 15 (x) di nero

prima

dopo



57

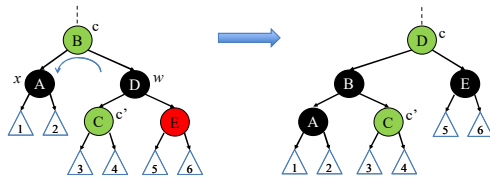
Cancellazione, fix-up

- analizziamo i casi che si possono capitare se *lost-color* è nero e il nodo x è nero
- come nel caso dell'inserimento i casi possono capitare da x in su
- c'è un caso solo che può provocare violazione ai livello più alti
- i lucidi fanno vedere i casi ad un livello qualsiasi
- w denota il fratello di x assumiamo che x è figlio sinistro (se è destro bisogna il caso analogo speculare)
- il colore di certi nodi non sarà definito: quelli verdi
- le variabili accanto i nodi verdi rappresentano il colore del nodo

58

Cancellazione, fix-up, caso 1

- caso 1: w è nero e il figlio destro di w è rosso

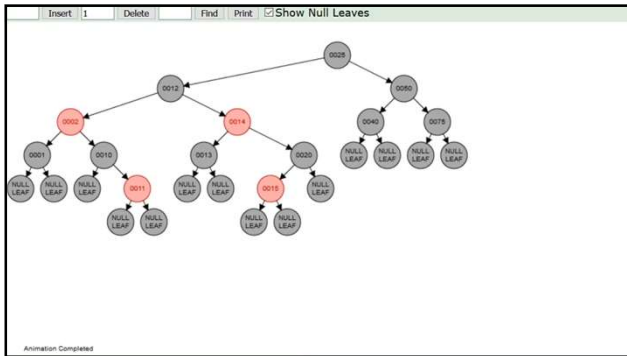


59

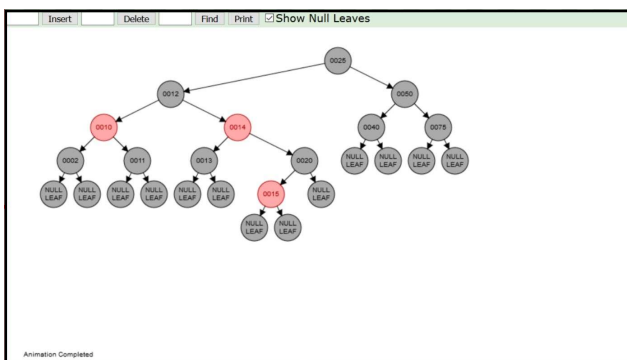
Cancellazione, fix-up, caso 1

- la regola dei cammini è violata perché dal x in giù manca un nero
- con le modifiche viene aggiunto un nodo nero (B) sui rami che attraversano x
- colorando il nodo E di nero, sulla destra di D manteniamo lo stesso numero di neri sui rami
- regola dei cammini non è più violata
- se D è la radice ed è rosso allora bisogna colorarlo di nero
- l'albero è ok
- nuovo x : *Root* (perché solo al livello della radice può esserci ancora un problema se la radice è diventata rossa)

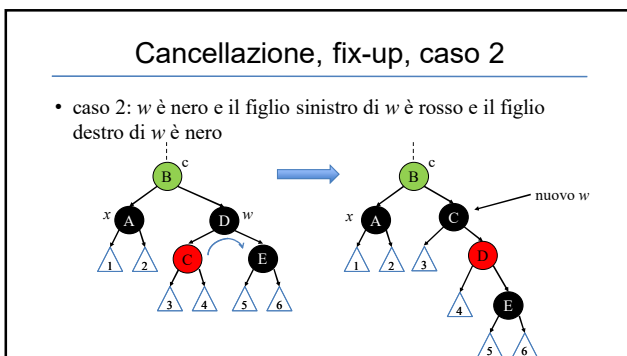
60



61



62

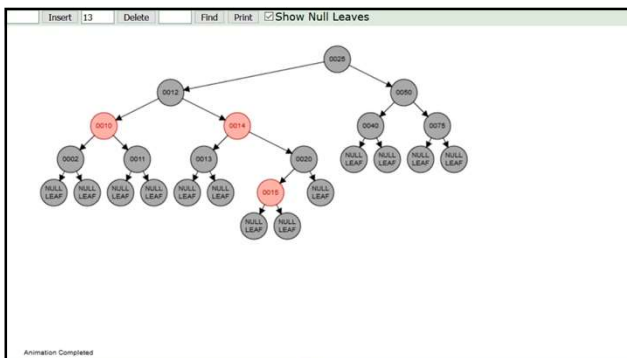


63

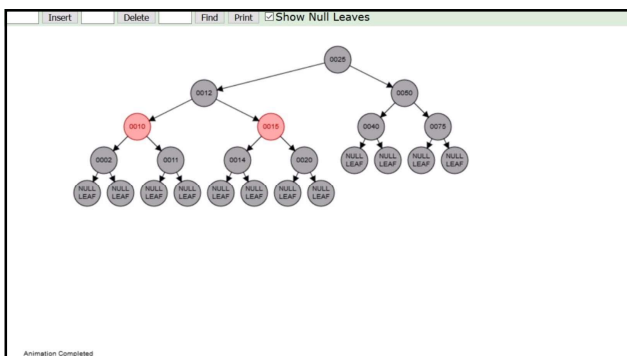
Cancellazione, fix-up, caso 2

- la regola dei cammini è violata perché dal x in giù manca un nero
- con le modifiche ci troviamo nella situazione del caso 1
- il nuovo w è il nodo C
- dopo le modifiche del caso 1, l'albero è ok

64



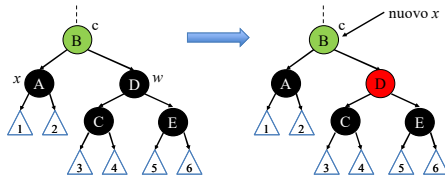
65



66

Cancellazione, fix-up, caso 3

- caso 3: w è nero e i figli di w sono neri

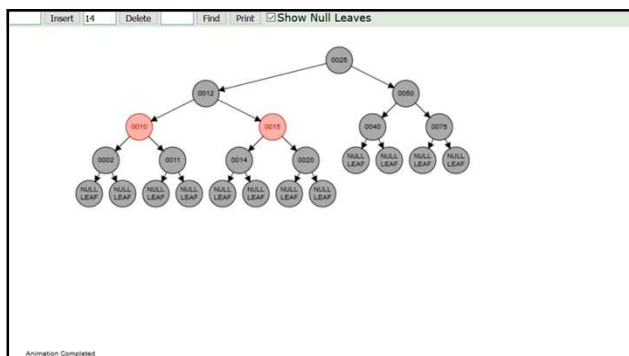


67

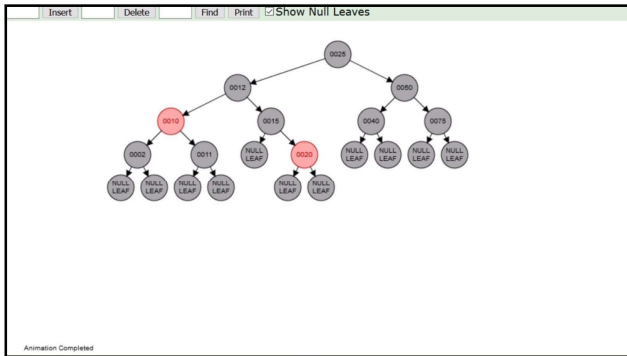
Cancellazione, fix-up, caso 3

- la regola dei cammini è violata perché dal x in giù manca un nero
- l'unica modifica è che il nodo D viene colorato di rosso
- di conseguenza il nodo B non viola più la regola dei cammini e il nero manca dal livello di B in su
- se B è rosso allora basta colorarlo di nero e l'albero è a posto
- se B è nero allora si ricomincia dal livello di B
- il nuovo nodo x è B

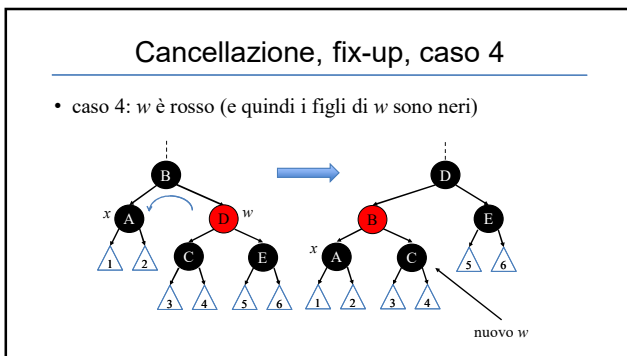
68



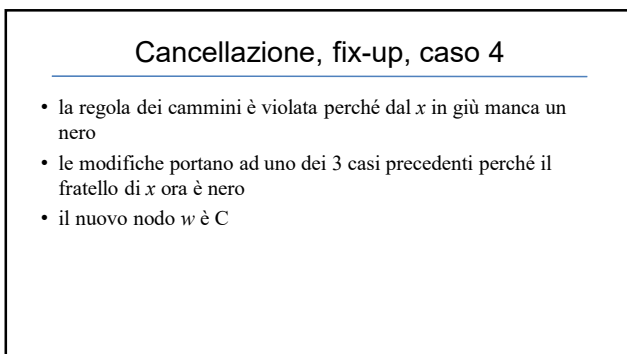
69



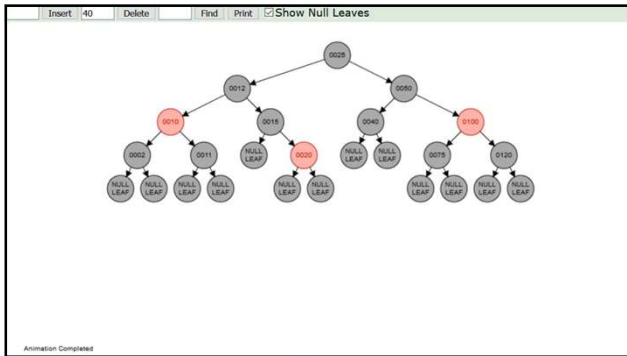
70



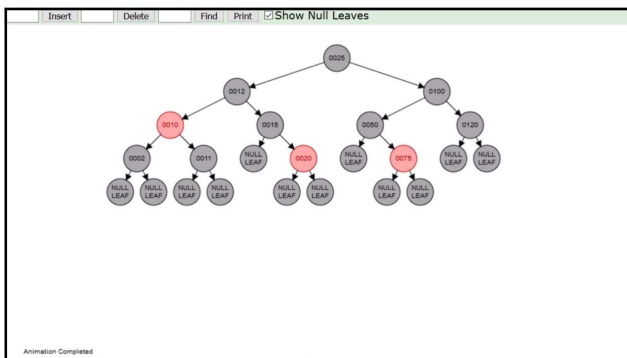
71



72



73



74

```

RB-DELETE-FIXUP(T, x)    ▷ x è il nodo che può creare problemi
while x ≠ T.root ∧ x.color = black do
  if x = x.p.left then    ▷ x è figlio sinistro?
    w ← x.p.right         ▷ w è il fratello di x
    if w.color = red then  ▷ caso 4?
      w.color ← black; x.p.color ← red
      LEFT-ROTATE(T, x.p); w ← x.p.right    ▷ caso 4
    if w.left.color = black ∧ w.right.color = black then  ▷ caso 3?
      w.color ← red; x ← x.p                ▷ caso 3
    else
      if w.right.color = black then          ▷ caso 2?
        w.left.color ← black; w.color ← red
        RIGHT-ROTATE(T, w); w ← x.p.right    ▷ caso 2
      w.color ← x.p.color; x.p.color ← black
      w.right.color ← black; LEFT-ROTATE(T, x.p)  ▷ caso 1
      x ← T.root                             ▷ caso 1
    else
      {tutto il corpo del if esterno con left e right scambiati}
      x.color ← black
  
```

75

Complessità

- altezza di un albero rosso-nero con n nodi è $O(\log n)$
- l'algoritmo di ricerca scende lungo un ramo e quindi è $O(\log n)$
- gli algoritmi di inserimento e cancellazione
 - scendono lungo un ramo e
 - risalgono al massimo fino la radice effettuando rotazioni e/o ricolorazioni (operazioni $O(1)$)
 - quindi sono operazioni $O(2\log n) = O(\log n)$
