Algoritmi e Strutture Dati

Alberi Rosso-Neri (RB-Trees)

Simone Faro

Dipartimento di Matematica e Informatica
Università degli Studi di Catania

Alberi Rosso-Neri

- Un albero Rosso Nero (Red Black Tree, RB tree) è un albero binario di ricerca in cui ad ogni nodo viene associato un colore (**rosso** o **nero**)
- Ogni nodo di un RB tree ha 5 campi: *key*, *left*, *right* e *p* (vedi BST ordinari) + *color*
- Vincolando il modo in cui è possibile colorare i nodi lungo un qualsiasi percorso che va dalla radice ad una foglia, riusciamo a garantire che l'abero sia **approssivatimente bilanciato**

Alberi bilanciati

Definizione (*fattore di bilanciamento*): sia T un albero binario e v un nodo dili T. Il fattore di bilanciamento $\beta(v)$ di v è definito come la differenza tra l'altezza del suo sottoalbero sinistro e quella del suo sottoalbero destro

$$\beta(v) = altezza[left[v]] - altezza[right[v]]$$

Definizione (*bilanciamento in altezza*): un albero è bilanciato in altezza se, per ogni suo nodo v, $|\beta(v)| \leq 1$

Alberi Rosso-Neri: proprietà

Un RB tree è un albero binario di ricerca che soddisfa le seguenti proprietà (dette RB-properties):

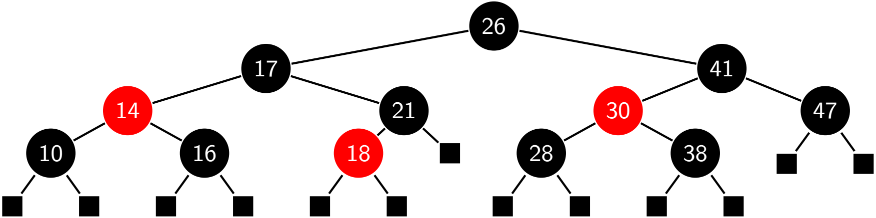
1. ogni nodo è rosso o **nero**
2. la radice è **nera**
3. ogni foglia è **nera**
4. se un nodo è rosso, entrambi i suoi figli devono essere **neri**
5. **tutti** i percorsi da un qualsiasi nodo n ad una qualsiasi delle sue foglie discendenti contengono lo stesso numero di nodi neri

Alberi Rosso-Neri: proprietà

Prop. 1 ogni nodo è rosso o **nero**

Prop. 2 la radice è **nera**

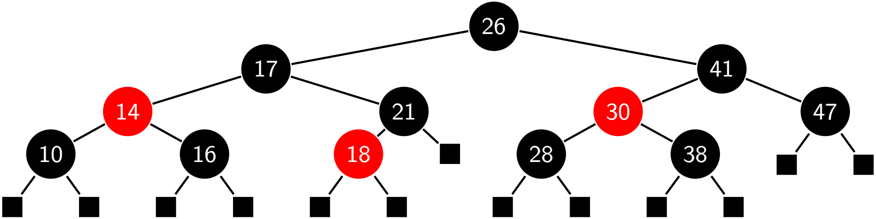
Prop. 3 ogni foglia è **nera**: spesso si aggiungono dei nodi NIL fittizi



 è un abbreviazione per Nil

Alberi Rosso-Neri: proprietà

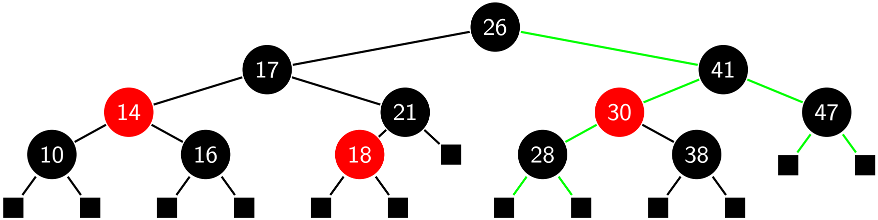
Prop. 4 se un nodo è **rosso**, entrambi i suoi figli devono essere neri



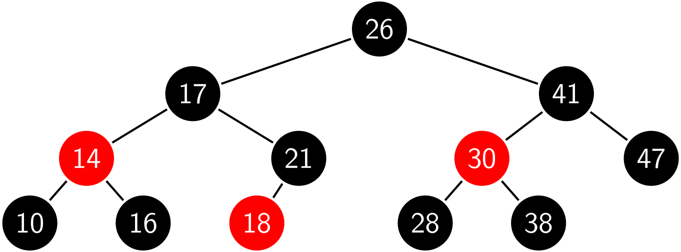
Conseguenza: almeno la metà dei nodi lungo un qualsiasi cammino dalla radice alle foglie devono essere neri

Alberi Rosso-Neri: proprietà

Prop. 5 per ogni nodo n, tutti i percorsi da n ad una qualsiasi delle sue foglie discendenti contengono lo stesso numero di nodi neri



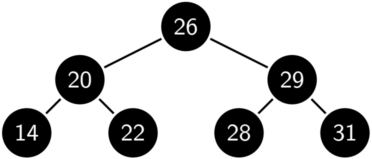
Alberi Rosso-Neri: un esempio

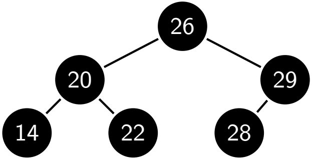


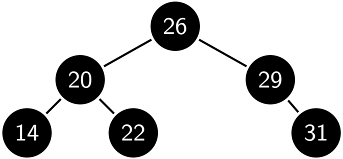
Una versione semplificata, senza NIL fittizi

Bilanciamento di un RB tree

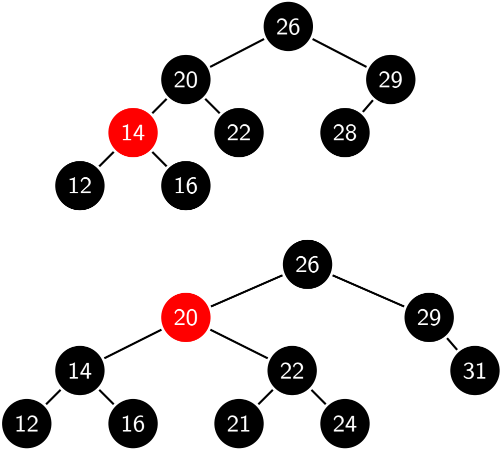
Un albero rosso-nero senza nodi rossi è bilanciato: tutti i suoi livelli sono completi tranne al più l'ultimo, al quale può mancare qualche foglia







Bilanciamento di un RB tree



Idea di base

Per la Prop. 5, un RB tree senza nodi rossi deve essere bilanciato: tutti i suoi livelli sono completi, tranne, al più, l'ultimo al quale può mancare qualche foglia

Non è però quasi completo (vedi gli heap) perchè le foglie mancanti non sono necessariamente quelle più a destra

A questo albero bilanciato possiamo aggiungere alcuni ("non troppi") nodi rossi; per la Prop. 4, infatti, se un nodo è rosso i suoi figli deveno essere neri

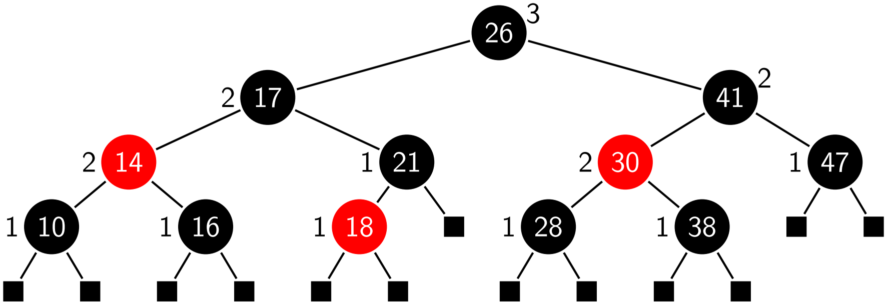
Altezza nera di un RB tree

Definizione Sia x un nodo di un RB tree T. L'**altezza nera** $bh(x)$ di x è pari al numero di nodi neri (x escluso) lungo un cammino da x ad una delle sue foglie discendenti.

L'altezza nera di T è l'altezza nera della root r: $bh(T) = bh(r)$

Per la proprietà 5, il concetto di altezza nera è **ben definito**

Alberi Rosso-Neri: altezza nera



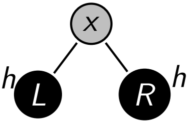
Altezza nera: alcune proprietà

Sia x un nodo di un RB tree e p il padre di x (ossia, $p = p[x]$). Allora:

1. x rosso implica $bh(x) = bh(p)$;
2. x nero implica $bh(x) = bh(p) - 1$;
3. in entrambi i casi, $bh(x) \geq bh(p) - 1$.

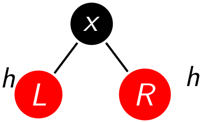
Calcolo dell'altezza nera di x

■ $$bh(x) = 0$$



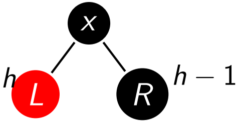
(a)

$$bh(x) = h + 1$$



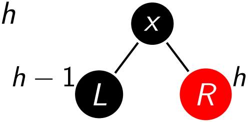
(b)

$$bh(x) = h$$



(c)

$$bh(x) = h$$



(d)

$$bh(x) = h$$

Un paio di risultati utili

Lemma 1: Il numero di nodi interni di un sottoalbero radicato in x è maggiore o uguale di $2^{bh(x)} - 1$.

Teorema 1: L'altezza di un RB tree con n nodi interni è minore o uguale di $2\log(n+1)$

Riusciamo a mantenere l'altezza logaritmica nel numero dei nodi.

Un paio di risultati utili

Lemma 1: Il numero di nodi interni di un sottoalbero radicato in x è maggiore o uguale di $2^{bh(x)} - 1$

Proof: Indichiamo con $int(x)$ il numero dei nodi interni del sottoalbero radicato in x, e dimostriamo (per induzione sull'altezza (ordinaria) h di x) che $int(x) \geq 2^{bh(x)-1}$.

Caso base: $h = 0$. In questo caso x è una foglia, $int(x) = 0$ e $bh(x) = 0$. Allora:

$$int(x) = 0 \geq 2^{bh(x)} - 1 = 2^0 - 1 = 0$$

Un paio di risultati utili

Passo induttivo: $h > 0$. In questo caso, x ha due figli (non entrambi Nil) $l = left[x]$ ed $r = right[x]$. Inoltre: $bh(l), bh(r) \geq bh(x) - 1$ (vedi proprietà dell'altezza nera).

Per ipotesi induttiva:
$int(l) \geq 2^{bh(l)} - 1 \geq 2^{bh(x)-1} - 1$
e
$int(r) \geq 2^{bh(r)} - 1 \geq 2^{bh(x)-1} - 1$. Allora:

$$\begin{aligned} int(x) &\geq 1 + int(l) + int(r) \\ &\geq 1 + (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) \\ &= 2 \cdot 2^{bh(x)-1} - 1 \\ &= 2^{bh(x)} - 1 \end{aligned}$$

Un paio di risultati utili

Teorema 2: L'altezza di un RB tree con n nodi interni è minore o uguale di $2\log(n+1)$.
Proof:

- sia h l'altezza dell'albero
- per Prop. 4, almeno la metà dei nodi in un qualsiasi cammino dalla radice ad una foglia (esclusa la radice) devono essere neri (dopo ogni nodo rosso c'è almeno un nodo nero)
- di conseguenza, $bh(T) = bh(root) \geq h/2$

Un paio di risultati utili

Teorema 2: L'altezza di un RB tree con n nodi interni è minore o uguale di $2\log(n+1)$.
Proof:

- sia h l'altezza dell'albero
- per Prop. 4, almeno la metà dei nodi in un qualsiasi cammino dalla radice ad una foglia (esclusa la radice) devono essere neri (dopo ogni nodo rosso c'è almeno un nodo nero)
- di conseguenza, $bh(T) = bh(root) \geq h/2$
- Per il Lemma 1: $n = int(T) \geq 2^{bh(T)} - 1 \geq 2^{h/2} - 1$, ossia $2^{h/2} \leq n + 1$

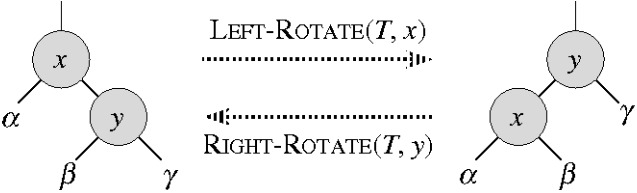
Un paio di risultati utili

Teorema 2: L'altezza di un RB tree con n nodi interni è minore o uguale di $2\log(n+1)$.
Proof:

- sia h l'altezza dell'albero
- per Prop. 4, almeno la metà dei nodi in un qualsiasi cammino dalla radice ad una foglia (esclusa la radice) devono essere neri (dopo ogni nodo rosso c'è almeno un nodo nero)
- di conseguenza, $bh(T) = bh(root) \geq h/2$
- Per il Lemma 1: $n = int(T) \geq 2^{bh(T)} - 1 \geq 2^{h/2} - 1$, ossia $2^{h/2} \leq n+1$
- Allora: $h/2 \leq \log(n+1)$ e $h \leq 2\log(n+1)$

Rotazioni

Sono delle operazioni di **ristrutturazione locale** dell'albero



Dove:

- α è il sottoalbero la cui radice è il figlio sinistro di x (può anche essere vuoto)
- β, γ sono i sottoalberi le cui radici sono il figlio sinistro e destro (risp.) di y (possono anche essere vuoti)

Operazioni su RB trees

Vediamo nel dettaglio le operazioni di **inserimento** e **cancellazione** di un nodo

Le operazioni **Search**, **Minimum** e **Maximum**, **Successor** e **Predecessor** possono essere implementate esattamente come per gli alberi binari di ricerca "ordinari"

Inserimento di un nodo z

- Come per gli alberi binari di ricerca, l'inserimento di un nodo z in un RB tree cerca un cammino dalla root dell'albero fino al nodo p che diventerà suo padre

Inserimento di un nodo z

- Come per gli alberi binari di ricerca, l'inserimento di un nodo z in un RB tree cerca un cammino dalla root dell'albero fino al nodo p che diventerà suo padre
- Una volta identificato il padre p, z viene aggiunto come figlio sinistro (se $key[z] < key[p]$) o destro (se $key[z] \geq key[p]$) di p e colorato di **rosso** (per evitare inconsistenze della black height)

Inserimento di un nodo z

- Come per gli alberi binari di ricerca, l'inserimento di un nodo z in un RB tree cerca un cammino dalla root dell'albero fino al nodo p che diventerà suo padre
- Una volta identificato il padre p, z viene aggiunto come figlio sinistro (se $key[z] < key[p]$) o destro (se $key[z] \geq key[p]$) di p e colorato di **rosso** (per evitare inconsistenze della black height)
- Questo inserimento può causare una violazione della:
  - proprietà 2, se z viene inserito in un albero vuoto
  - proprietà 4, se z viene aggiunto come figlio di un nodo rosso

Inserimento di un nodo z

- Come per gli alberi binari di ricerca, l'inserimento di un nodo z in un RB tree cerca un cammino dalla root dell'albero fino al nodo p che diventerà suo padre
- Una volta identificato il padre p, z viene aggiunto come figlio sinistro (se $key[z] < key[p]$) o destro (se $key[z] \geq key[p]$) di p e colorato di **rosso** (per evitare inconsistenze della black height)
- Questo inserimento può causare una violazione della:
  - proprietà 2, se z viene inserito in un albero vuoto
  - proprietà 4, se z viene aggiunto come figlio di un nodo rosso
- La procedura RB-INSERT-FIXUP(T, z) (dove z è il nodo che da luogo alla violazione) ci consente di ripristinare le proprietà dei RB trees

RB-Insert-Fixup(T, z)

La RB-INSERT-FIXUP(T, z) ripristina:

- la Proprietà 2 colorando la root z (rossa) di nero
- la Proprietà 4, eseguendo delle **rotazioni** e **ricolorazioni** su z.

RB-Insert-Fixup(T, z)

La RB-Insert-Fixup(T, z) ripristina:

- la Proprietà 2 colorando la root z (rossa) di nero
- la Proprietà 4, eseguendo delle **rotazioni** e **ricolorazioni** su z.
- decidiamo cosa fare confrontando il colore di z con quello di suo **zio** y.

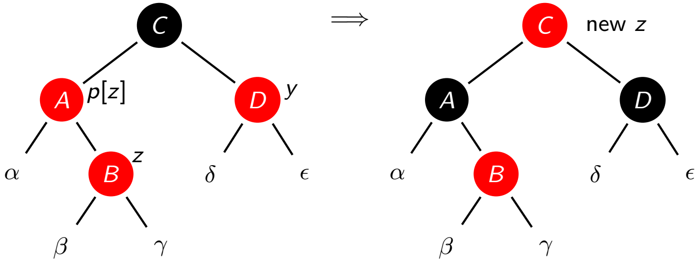
RB-Insert-Fixup(T, z)

La RB-INSERT-FIXUP(T, z) ripristina:

- la Proprietà 2 colorando la root z (rossa) di nero
- la Proprietà 4, eseguendo delle **rotazioni** e **ricolorazioni** su z.
- decidiamo cosa fare confrontando il colore di z con quello di suo **zio** y.
- Distinguiamo tre possibili casi:

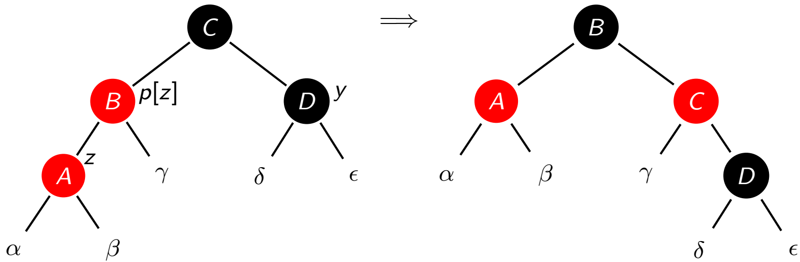
Caso 1 lo zio y di z è rosso
Caso 2 lo zio y di z è nero e z è un figlio sinistro
Caso 3 lo zio y di z è nero e z è un figlio sinistro

Caso 1: lo zio y di z è rosso



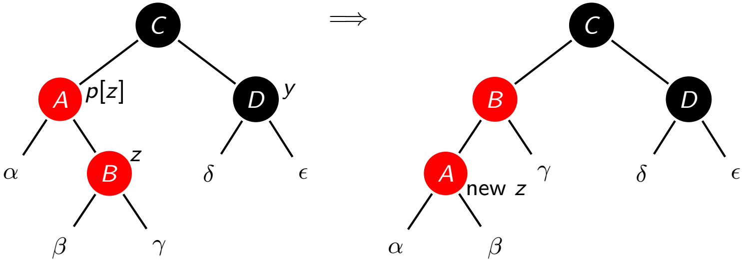
Lo zio y e $p[z]$ diventano neri mentre $p[p[z]]$ diventa rosso. A questo punto, il cambiamento di colore di $p[p[z]]$ potrebbe aver causato una nuova violazione della Proprietà 4; quindi $p[p[z]]$ diventa il nuovo z (è necessario ripristinare violazioni delle RB-properties causate da z)

Caso 2: lo zio y di z è nero e z è un figlio sinistro



$p[z]$ diventa nero e $p[p[z]]$ diventa rosso; infine, per ripristinare inconsistenze della black height, ruotiamo $p[p[z]]$ a destra

Caso 3: lo zio y di z è nero e z è un figlio destro

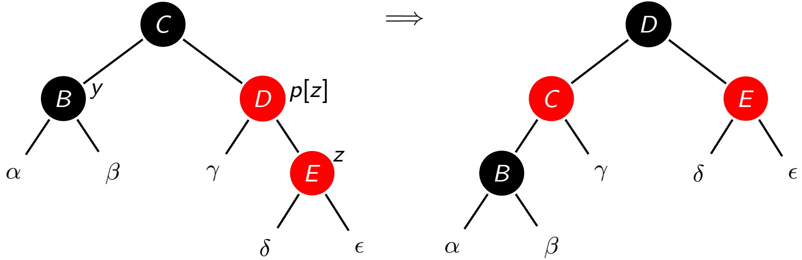


viene ricondotto al Caso 2 ruotando $p[z]$ a sinistra.

Quindi: coloriamo $p[new\ z]$ (i.e. B) di nero, $p[p[new\ z]]$ (i.e. C) di rosso e ruotiamo $p[p[new\ z]]$ a destra

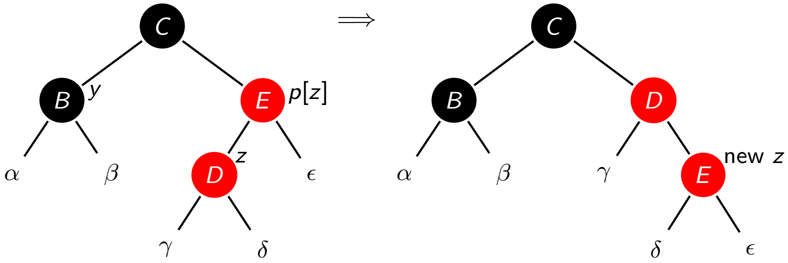
Caso 3: lo zio y di z è nero e z è un figlio destro

Ma y è un figlio sinistro (in tutti i casi esaminati finora, y era un figlio destro). Questo caso è il simmetrico del Caso 2 quando y è figlio destro. Attenzione alle simmetrie!!!



$p[z]$ diventa nero, il $p[p[z]]$ rosso e ruotiamo $p[p[z]]$ a sinistra

Caso 2: lo zio y di z è nero e z è un figlio sinistro



Viene ricondotto al caso precedente ruotando $p[z]$ a destra; ed $p[z]$ diventa il nuovo z

Analisi

- RB-INSERT-FIXUP(T, z) richiede un tempo $O(\log_2 n)$
- Di conseguenza, anche RB-INSERT(T, z) richiede un tempo $O(\log_2 n)$

Inserimento: un esercizio

Figura: **inserimento di 41 e 38**

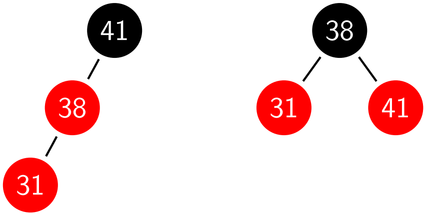


Figura: **inserimento di 31**

Inserimento: un esercizio

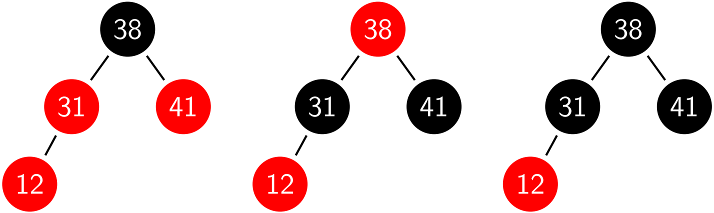


Figura: **inserimento di 12**

Inserimento: un esercizio

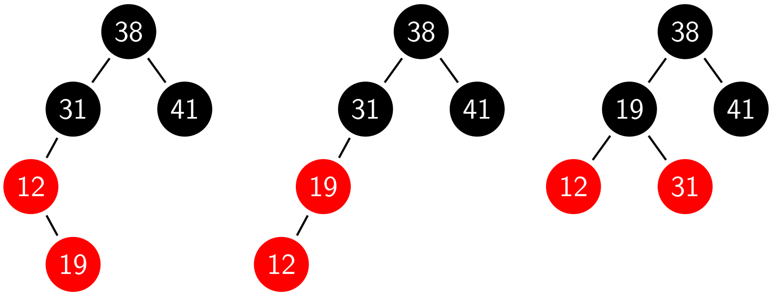


Figura: **inserimento di 19**

Inserimento: un esercizio

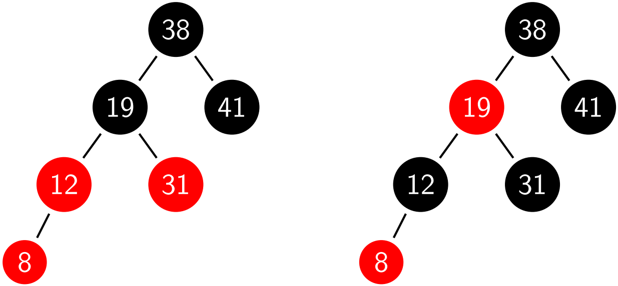


Figura: **inserimento di 8**

Cancellazione di un nodo con due figli

Osservazione: assumiamo sempre di dover eliminare un nodo che ha al più un figlio

Infatti, nel caso in cui il nodo z da eliminare ha due figli, possiamo sostituire la chiave di z con quella di $y = \text{TREE-SUCCESSOR}(T, z)$ (il successore di z), e poi rimuovere y

Poichè z è un nodo con due figli, il suo successore y è il nodo più a sinistra del sottoalbero destro; (ha al più il figlio destro)

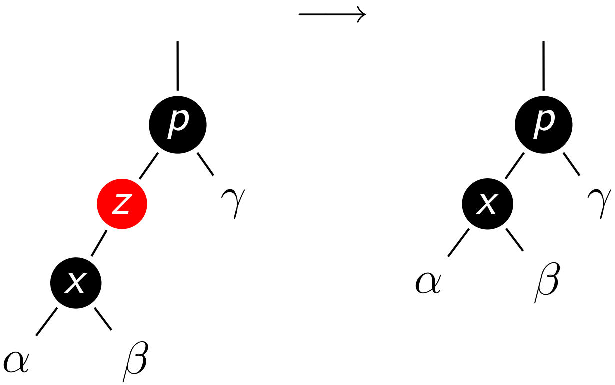
Cancellazione di un nodo con al più un figlio

Sia z il nodo da cancellare

Siano, inoltre, x l'unico figlio e p il padre di z (se z è una foglia, allora x è NIL). Per eliminare z, eseguiamo i seguenti passi:

1. inanzitutto, rimuoviamo z collegando p con x (p diventa il padre di x ed x diventa il figlio di p);
2. z era rosso: terminiamo perchè l'eliminazione di z non causa violazioni delle RB-properties
3. z **era nero**: potremmo causare una violazione della proprietà 5

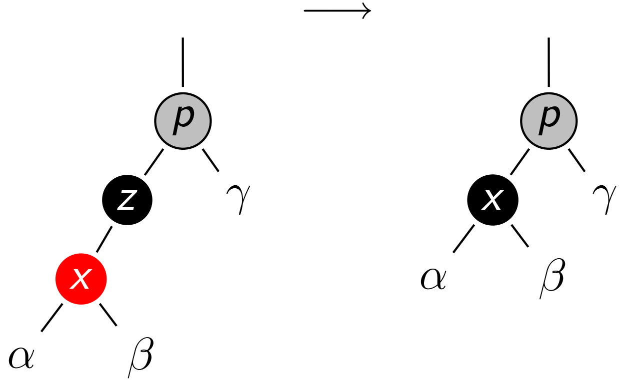
Eliminazione: z rosso



Eliminare un nodo z rosso non causa violazioni delle RB-properties

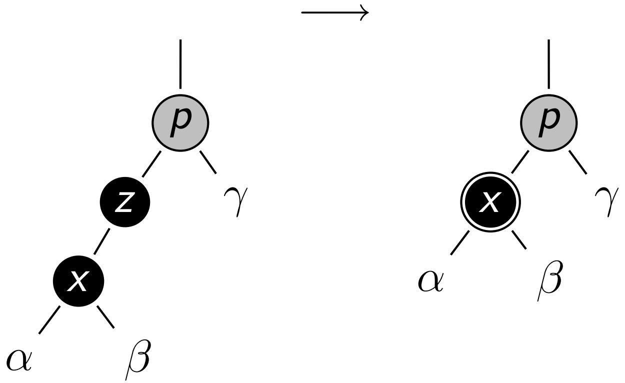
Eliminazione: z nero e suo figlio x rosso

Se z è nero, suo padre $p[z]$ può essere sia nero che rosso (per questo è griggio)



Il figlio rosso di z acquisice un "extra credito" diventando nero

Eliminazione: z nero e suo figlio x nero



Il figlio nero di z acquisisce un extra credito diventando "doppio nero".

Eseguiamo RB-DELETE-FIXUP(T, x) per ripristinare la proprietà 5

Cancellazione di un nodo con al più un figlio

Per ristabilire la proprietà 5 nel caso di eliminazione di un nodo z nero), si attribuisce al nodo x (figlio di z) un extra credito

Questo significa che se x è rosso lo coloriamo di nero, mentre se x è gia nero assume un colore fittizio detto **doppio nero**, che serve per ricordarci che abbiamo collassato due nodi neri in uno. Nel calcolo della black-height un nodo doppio nero conta due volte

Infine, eseguiamo la procedura $\text{RB-DELETE-FIXUP}(T, x)$ che spingerà, mediante rotazioni e ricolorazioni, l'extra credito verso la radice dove verrà ignorato

Se lungo il cammino verso la radice incontriamo un nodo rosso, esso sarà semplicemente colorato di nero

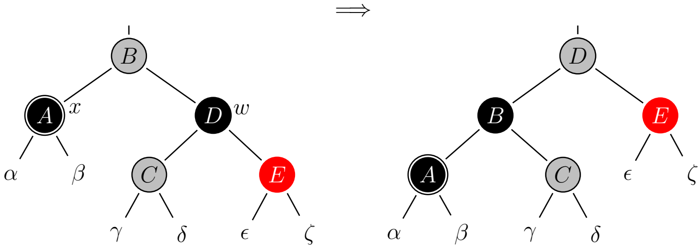
RB-Delete-Fixup(T, x)

Nel ripristinare la proprietà 5, teniamo conto di una serie di casi ottenuti confrontando il colore di x con quello di suo fratello w

1. w è nero ed ha almeno un figlio rosso. Possiamo distinguere ulteriori due sottocasi:
   1.1 il figlio destro di w è rosso
   1.2 il figlio destro di w è nero e quello sinistro è rosso
2. w è nero ed ha entrambi i figli neri. Anche in questo caso distinguiamo due possibili sottocasi
   2.1 il nodo $p[x]$ (che è anche il padre di w) è rosso
   2.2 il nodo $p[x]$ è nero
3. w è rosso

N.B: la RB-DELETE-FIXUP(T, x) viene chiamata su un nodo generico (che solo all'inizio è un figlio di z)

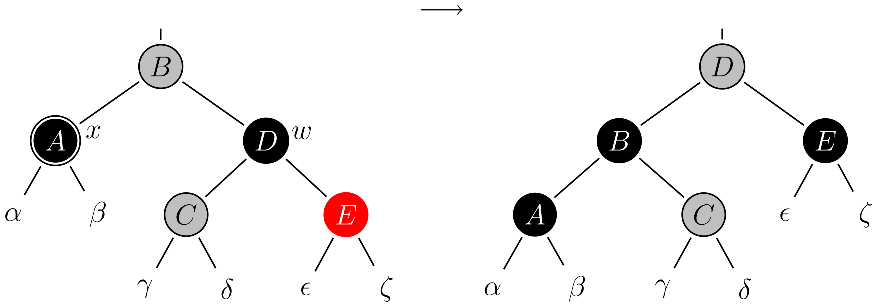
Caso 1.1: *w* nero e figlio destro di *w* rosso



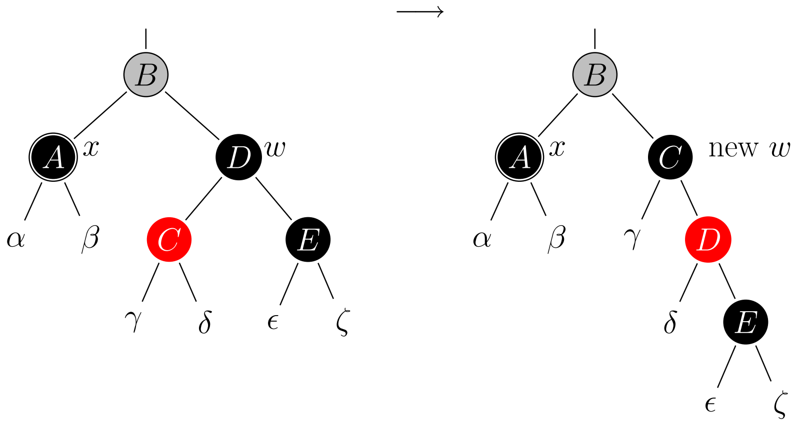
scambiamo il colore del $p[x]$ con quello di w e poi ruotiamo il $p[x]$ a sinistra

- abbiamo aggiunto un nodo nero a sinistra: possiamo togliere il doppio nero da x
- compensiamo il fatto che abbiamo tolto un nodo nero a destra colorando $right[w]$ (i.e. E) di nero

Caso 1.1: *w* nero e figlio destro di *w* rosso (cont.)

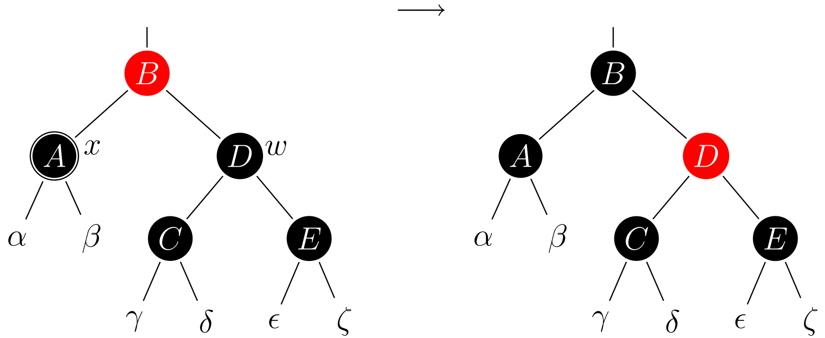


Caso 1.2: *w* ed il figlio destro di *w* sono neri, il figlio sinistro di *w* è rosso



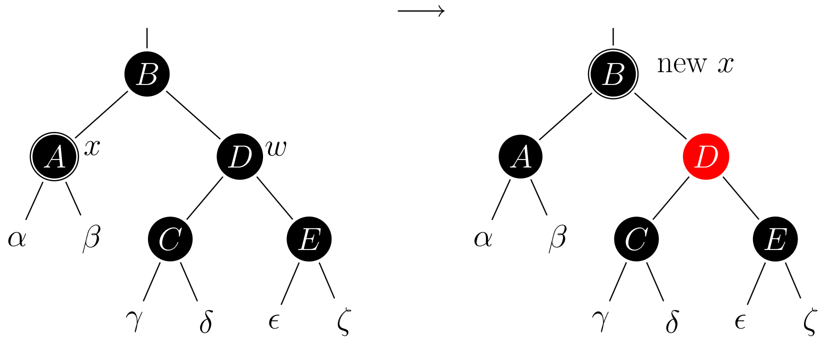
viene trasformato nel Caso 1.1 colorando *left*[*w*] (i.e. *C*) di nero, *w* (i.e. *D*) di rosso e ruotando *w* a destra

Caso 2.1: w ha entrambi i figli neri e $p[x]$ è rosso



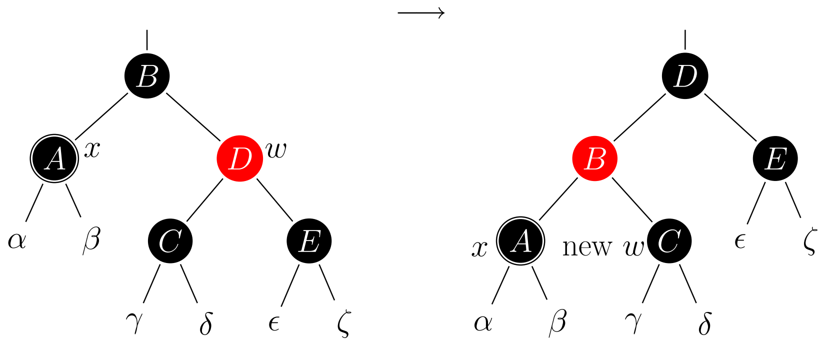
togliamo un credito nero sia ad x che a w (che diventano quindi nero ordinario e rosso, risp.) e facciamo acquisire un extra credito a $p[x]$ che da rosso diventa nero

Caso 2.2: w ha entrambi i figli neri e $p[x]$ è nero



simile al Caso 3.1; togliamo un credito nero sia ad x che a w (che diventano nero ordinario e rosso) e facciamo acquisire un extra credito a $p[x]$ che da nero diventa doppio nero. A questo punto $p[x]$ diventa il nuovo x

Caso 3: w è rosso



viene trasformato in uno dei casi precedenti colorando w (i.e. D) di nero, $p[x]$ (i.e. B) di rosso e ruotando $p[x]$ a sinistra

Analisi

- RB-DELETE-FIXUP(T, x) richiede un tempo $O(\log_2 n)$
- Di conseguenza, anche RB-DELETE(T, x) richiede un tempo $O(\log_2 n)$

Cancellazione: un esempio

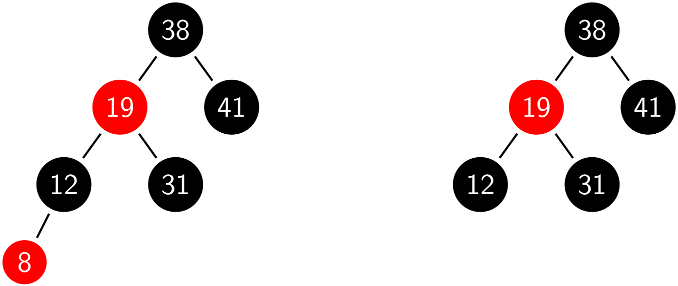


Figura: **cancellazione di 8**

Cancellazione: un esempio

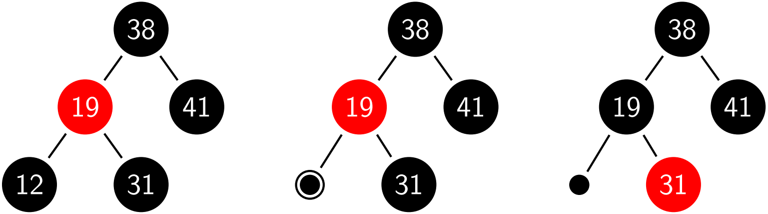


Figura: **cancellazione di 12**

Cancellazione: un esempio

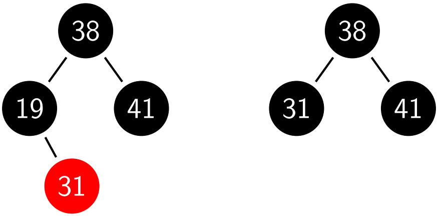


Figura: **cancellazione di 19**

Cancellazione: un esempio

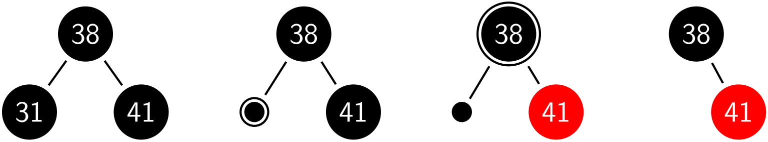


Figura: **cancellazione di 31**

Figura: **cancellazione di 38**