Alberi Rosso-Neri: definizione

- 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 quattro campi: *key*, *left*, *right* e *p* (come nel caso degli alberi di ricerca ordinari) + *color*

- Vincolando il modo in cui possiamo colorare i nodi lungo un qualsiasi percorso che va dalla radice ad una foglia, riusciamo a garantire che l'abero sia approssivatimente **bilanciato**

Alberi Rosso-Neri: proprietà

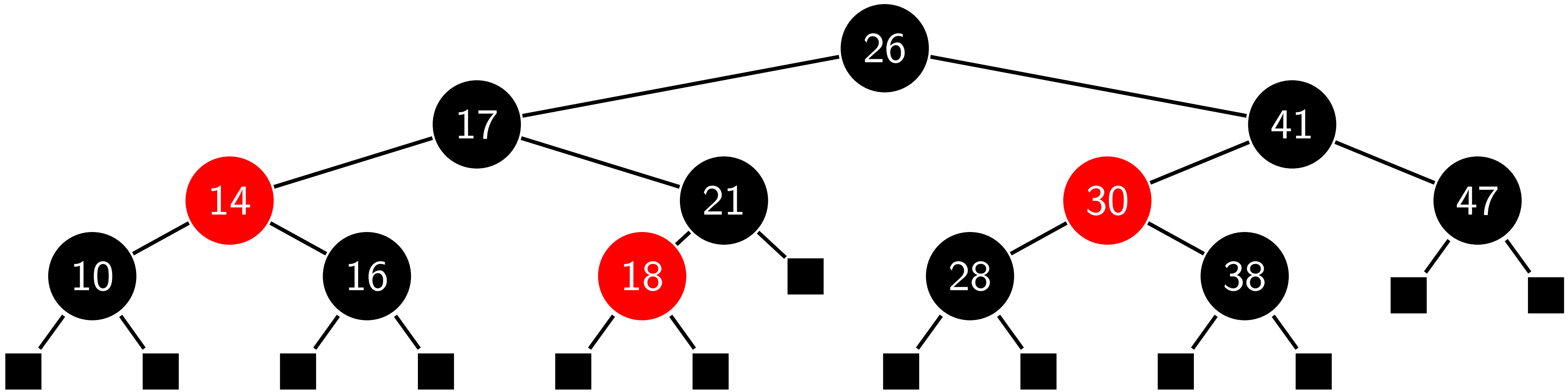
Un RB tree è un albero binario di ricerca che soddisfa le seguenti proprietà (dette RB-properties):

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. 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: proprietà

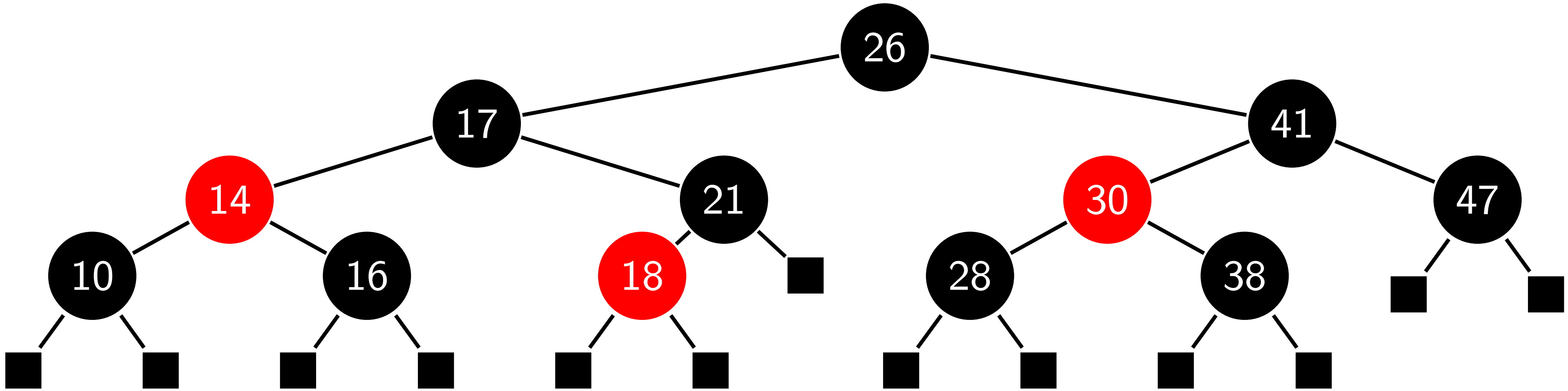
(1) ogni nodo è rosso o **nero**

(2) la radice è **nera**



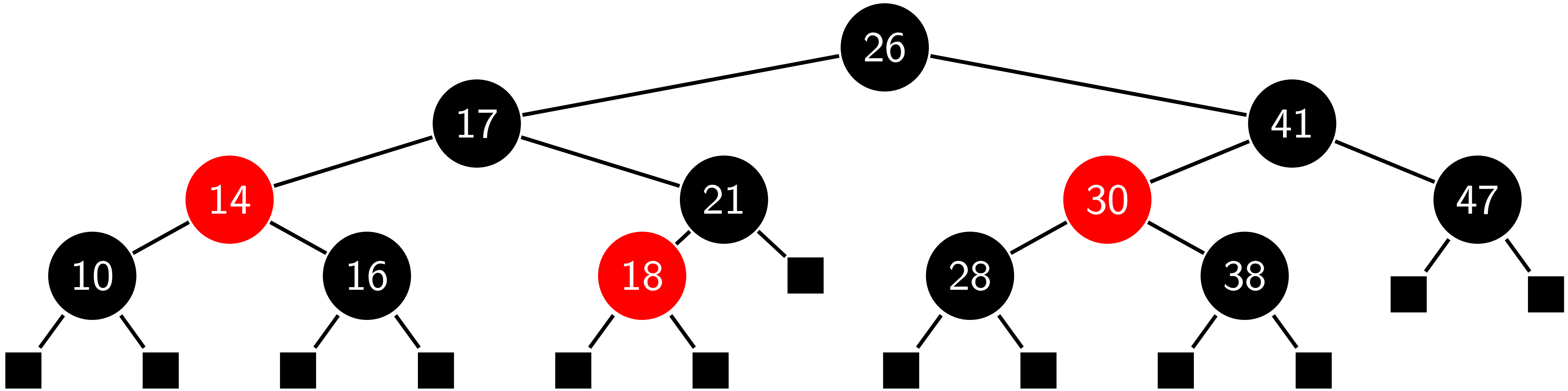
Alberi Rosso-Neri: proprietà

(3) ogni foglia è **nera** (basta aggiungere un ulteriore livello fittizio)



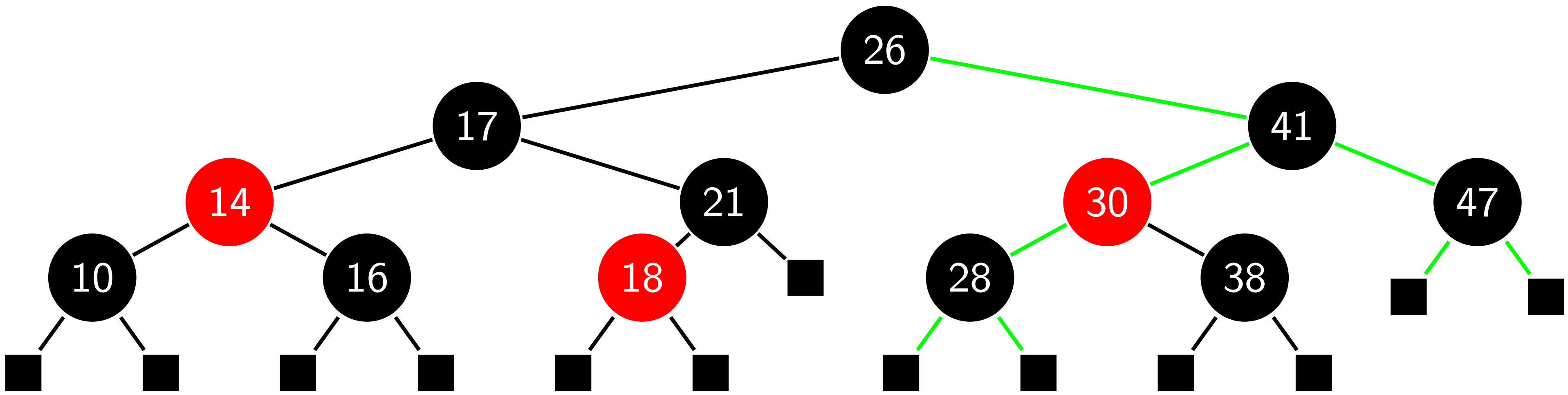
Alberi Rosso-Neri: proprietà

(4) se un nodo è rosso entrambi i suoi figli sono neri



Alberi Rosso-Neri: proprietà

(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

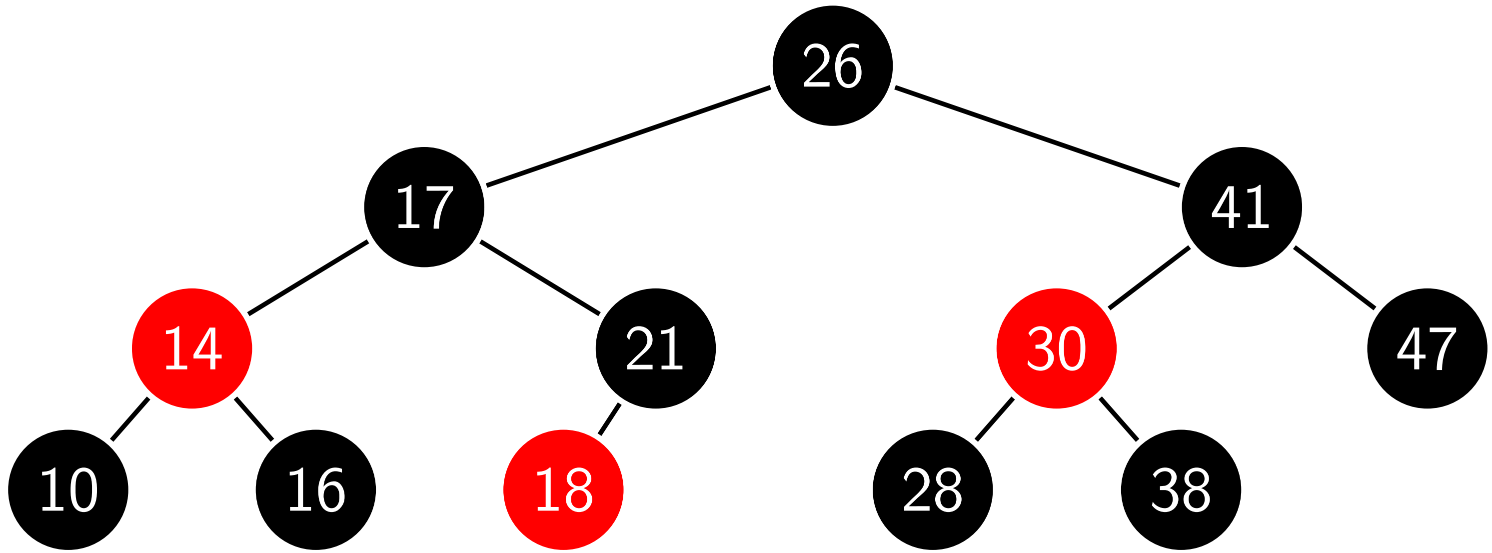
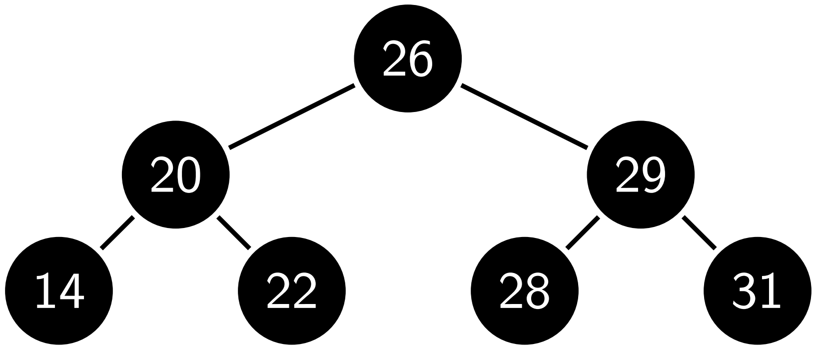
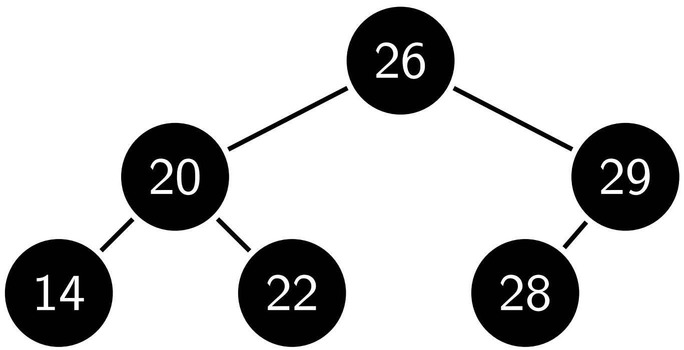


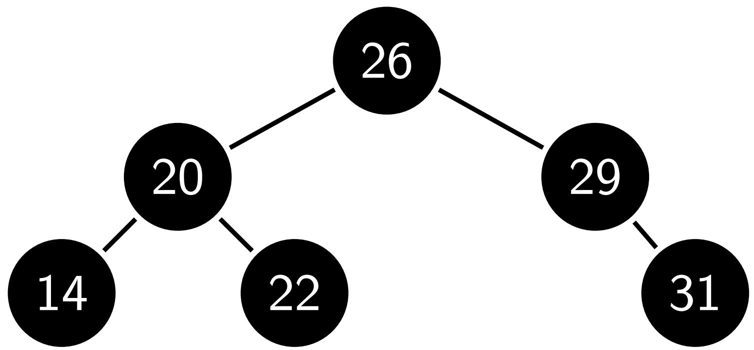
Figura: **red-black trees – una versione semplificata**

Alberi Rosso-Neri: proprietà 5

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







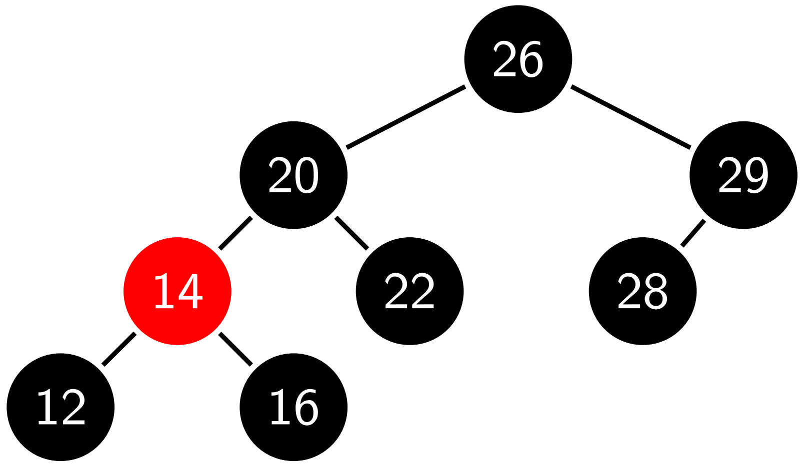
Alberi bilanciati

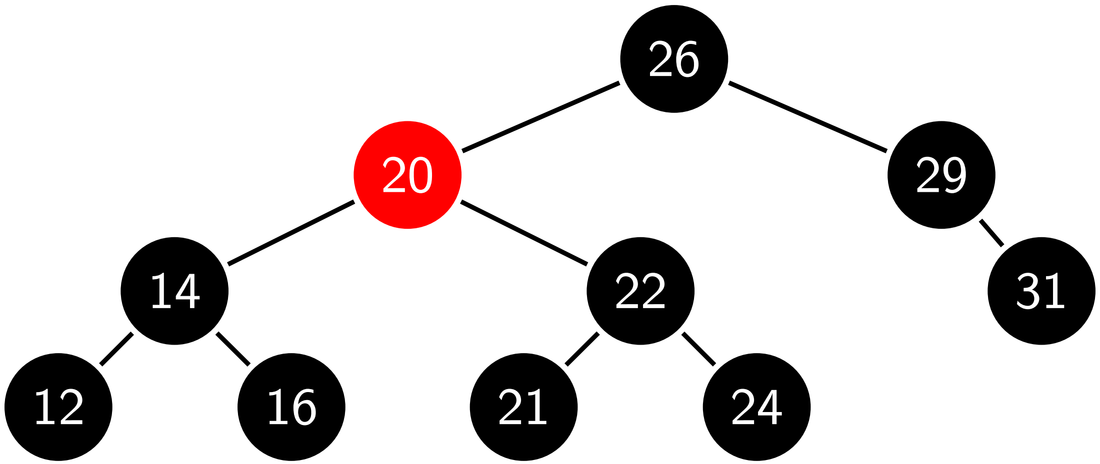
Definizione (*fattore di bilanciamento*): il fattore di bilanciamento $\beta(v)$ di un nodo v è definito come la differenza tra l'altezza del suo sottoalbero sinistro e quella del suo sottoalbero destro

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

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

Alberi Rosso-Neri: proprietà 4





Idea di base

Per la proprietà 5 (tutti i cammini da un nodo x alle foglie contengono lo stesso numero di nodi neri) un albero rosso-nero 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 (come nel caso di alberi che rappresentano un heap) perchè le foglie mancanti non sono necessariamente quelle più a destra

A questo albero bilanciato possiamo aggiungere "non troppi" nodi rossi (grazie alla proprietà 4: se un nodo è rosso i suoi figli deveno essere neri)

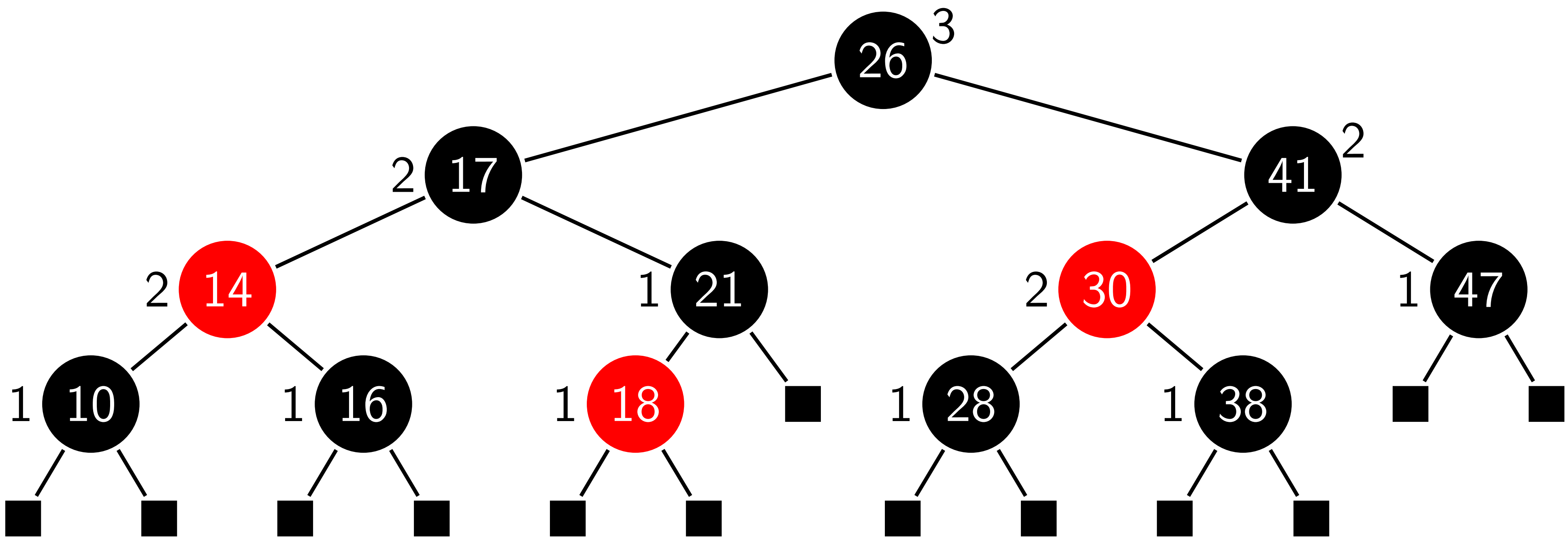
Ciò rende l'albero "quasi bilanciato". Cerchiamo di formalizzare questo concetto

Black-height di un RB tree

Sia T un red-black tree:

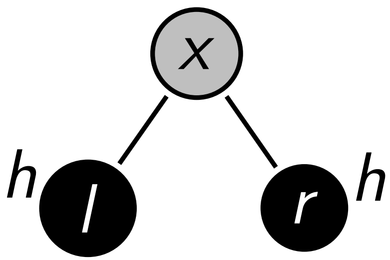
- Per ogni nodo x di T, l'**altezza nera** di x – $bh(x)$ – è pari al numero di nodi neri (escluso x) che si incontrano lungo un cammino da x ad una foglia
- L' altezza nera dell'albero è definita come l'altezza nera della sua radice r, ossia $bh(T) = bh(r)$
- Per la proprietà 5, il concetto di altezza nera è ben definito,in quanto tutti i percorsi che scendono da un nodo contengono lo stesso numero di nodi neri

Alberi Rosso-Neri: altezza nera



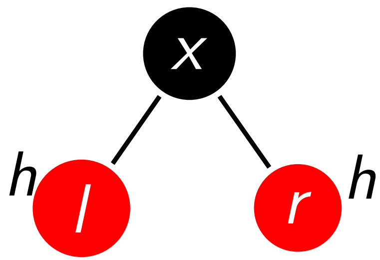
Calcolo dell'altezza nera di x

$$bh(x) = 0$$



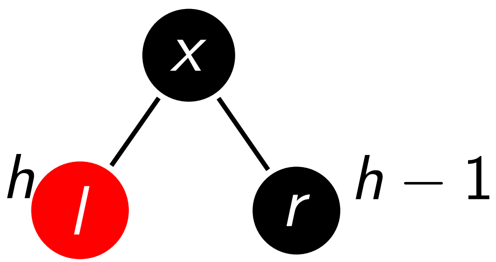
(a)

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



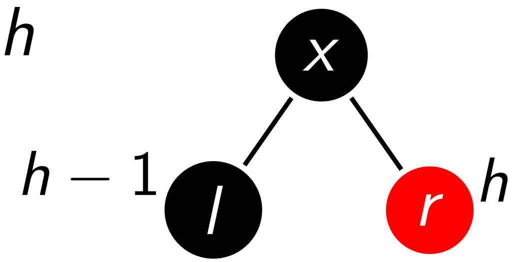
(b)

$$bh(x) = h$$



(c)

$$bh(x) = h$$



(d)

$$bh(x) = h$$

Altre proprietà dell'altezza nera

1. se x è rosso $bh(x) = bh(p[x])$
2. se x è nero $bh(x) = bh(p[x]) - 1$
3. in entrambi i casi $bh(x) \geq bh(p[x]) - 1$

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 massima di un RB tree con n nodi interni è $2\log(n+1)$.

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: Procediamo per induzione sull'altezza h di x. Nel seguito, denoteremo con $int(x)$ il numero dei nodi interni del sottoalbero radicato in x.

- $h = 0$ e quindi x è una foglia (i.e. $x = Nil$). In questo caso $int(x) = 0$ e $bh(x) = 0$. Quindi:

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

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:

- $h > 0$. In questo caso x ha due figli $l = left[x]$ ed $r = right[x]$ con $bh(l), bh(r) \geq bh(x) - 1$ (proprietà 3 dell'altezza nera). Inoltre, 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:

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

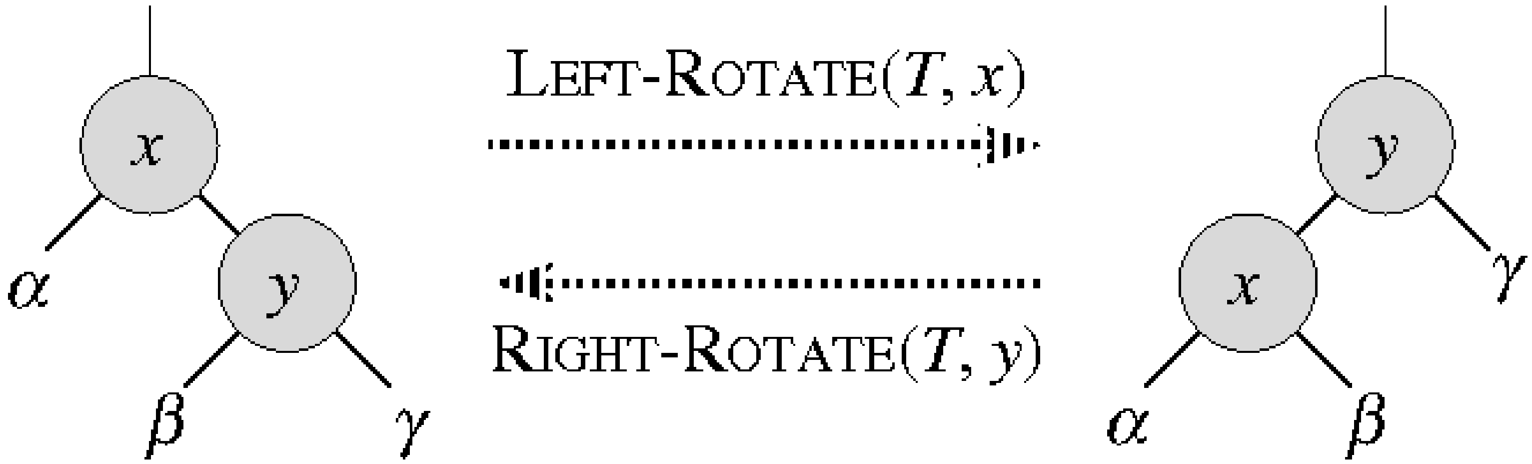
Un paio di risultati utili

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

- sia h l'altezza dell'albero
- per la un prop. 4, almeno metà dei nodi in 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, $n = int(root) \geq 2^{bh(root)} - 1 \geq 2^{h/2} - 1$, 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



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
- Una volta identificato p, z viene aggiunto come figlio sinistro (se $key[z] < key[p]$) o destro (se $key[z] > key[p]$) di p e colorato di rosso
- Quali RB-properties possono essere violate in conseguenza di questo inserimento? Solo due:
  - la proprietà 2 (se z viene inserito in un albero vuoto)
  - la proprietà 4 (se z viene aggiunto come figlio di un nodo rosso)
- La procedura che elimina violazioni delle RB-properties dovute all'inserimento di una chiave è la $\textsc{RB-Insert-Fixup}(T, z)$; qui z è il nodo che da luogo alla violazione

RB-Insert-Fixup(T, z)

- Ripristina la proprietà 2 colorando la root z (che in questo caso è rossa) di nero
- Ripristina la proprietà 4, eseguendo (ricorsivamente) delle rotazioni e ricolazioni sul nodo z; decidiamo in base a tre possibili casi:
  1. lo zio y di z è rosso
  2. lo zio y di z è nero e z è un figlio destro
  3. lo zio y di z è nero e z è un figlio sinistro

Caso 1: lo zio y di z è rosso

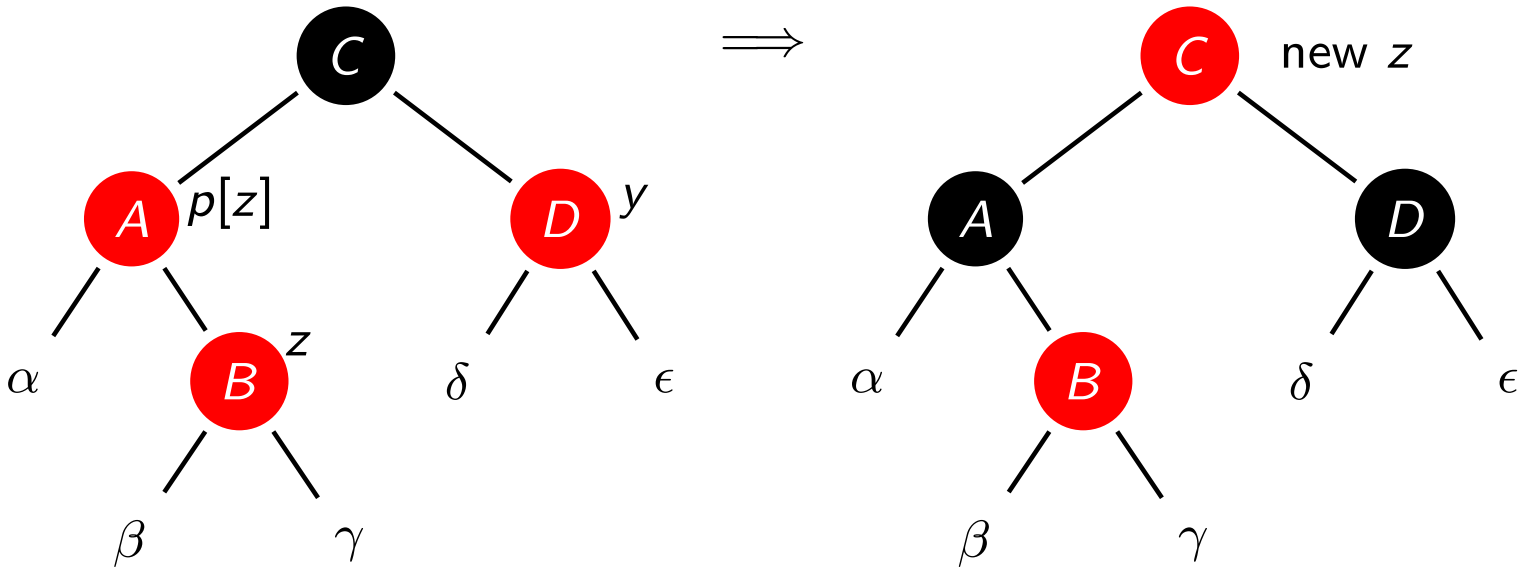


Figura: **in questo caso, A e D diventano neri e C diventa rosso. Inoltre C diventa il nuovo z dato che il suo cambiamento di colore potrebbe aver causato una violazione della proprietà 4**

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

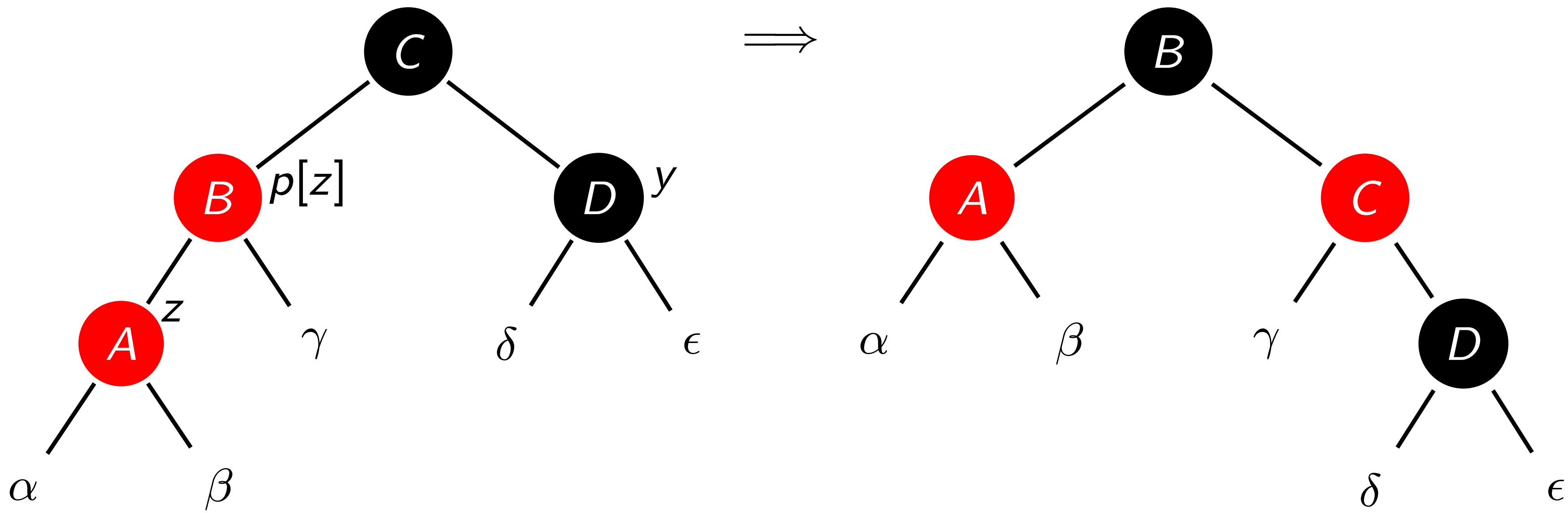


Figura: B **(i.e. $p[z]$) diventa nero; C (i.e. $p[p[z]]$) diventa rosso; ruotiamo C a destra**

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

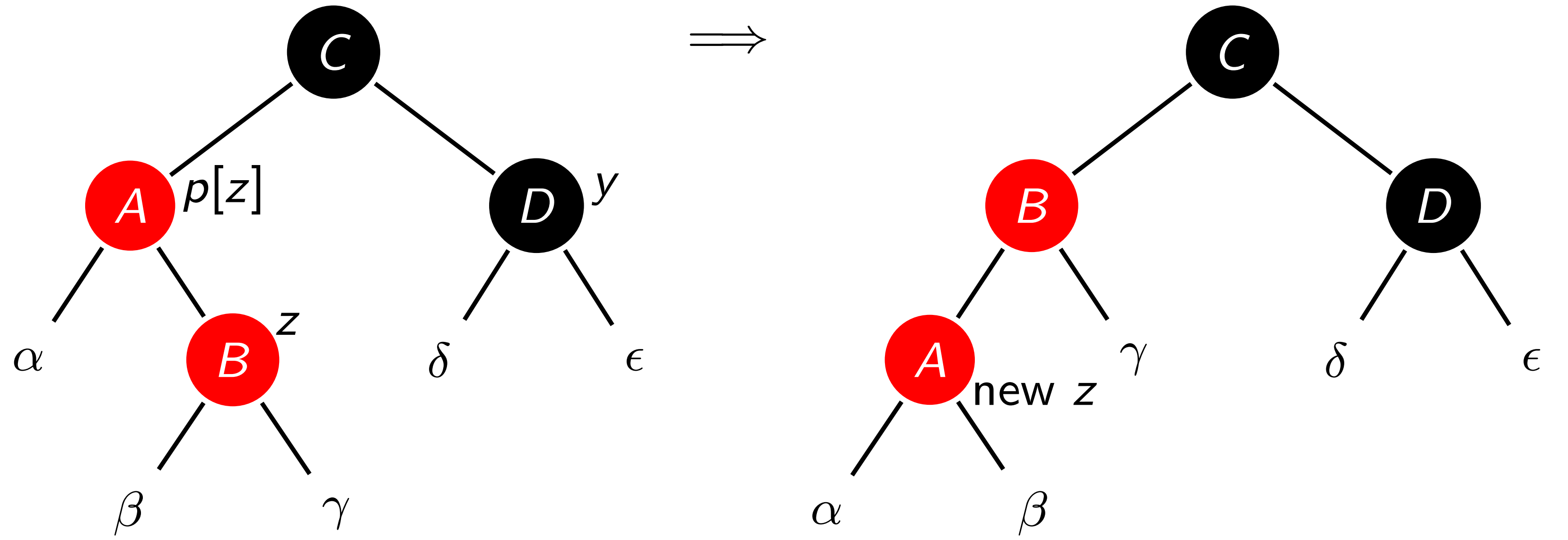


Figura: **viene ricondotto al caso 2 ruotando A, i.e. $p[z]$ a sinistra. A questo punto non ci resta che colorare C di rosso, B di rosso e ruotare C a destra**

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

Attenzione alle simmetrie!!!

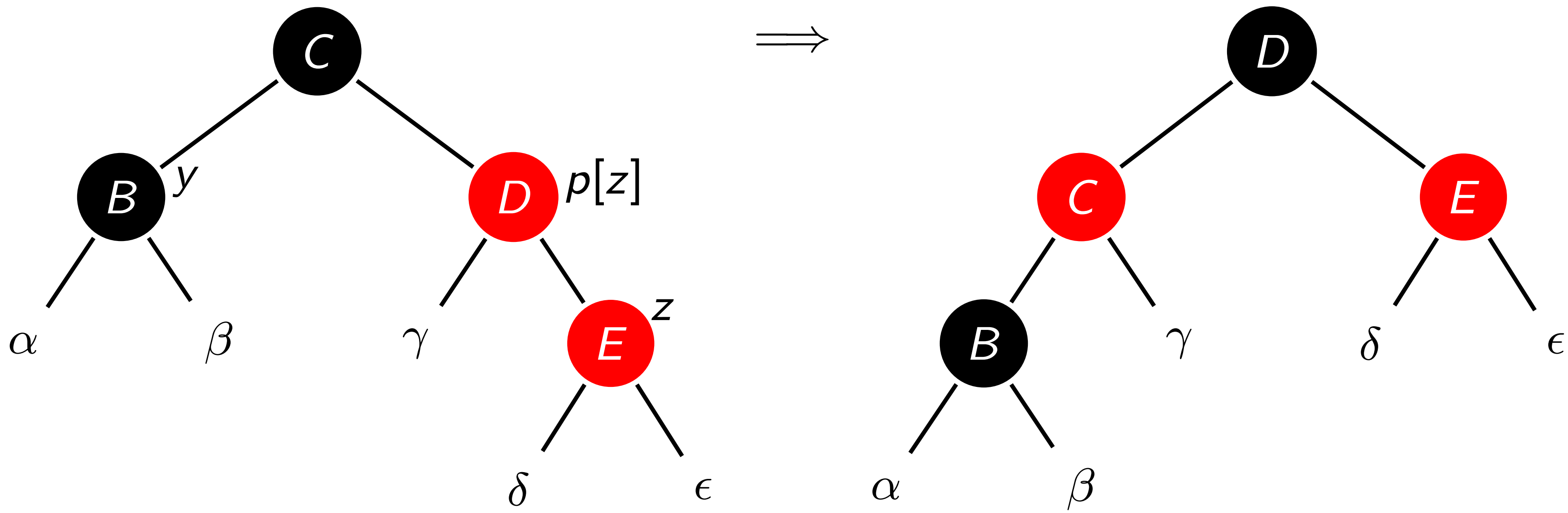


Figura: D **(i.e. $p[z]$) diventa nero; C (i.e. $p[p[z]]$) diventa rosso; ruotiamo C a sinistra**

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

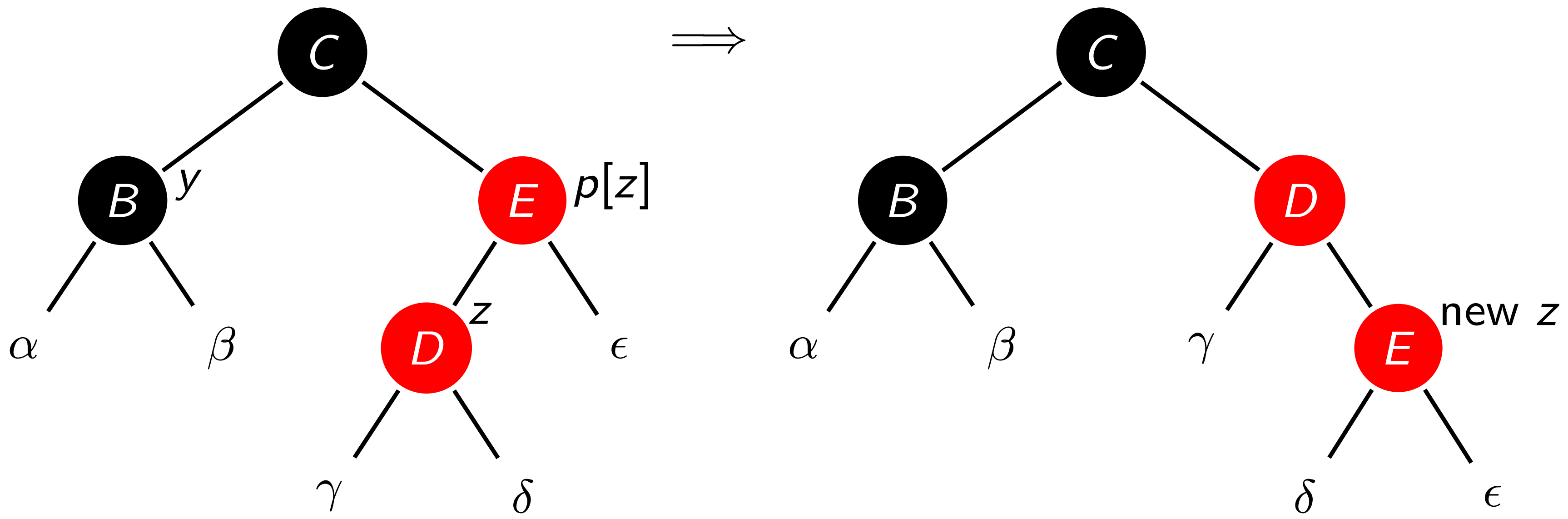


Figura: **ruotiamo E a destra ed E diventa il nuovo z. Osservate che, dopo la rotazione, z è un figlio destro di un nodo – D – che a sua volta è un figlio destro**

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**

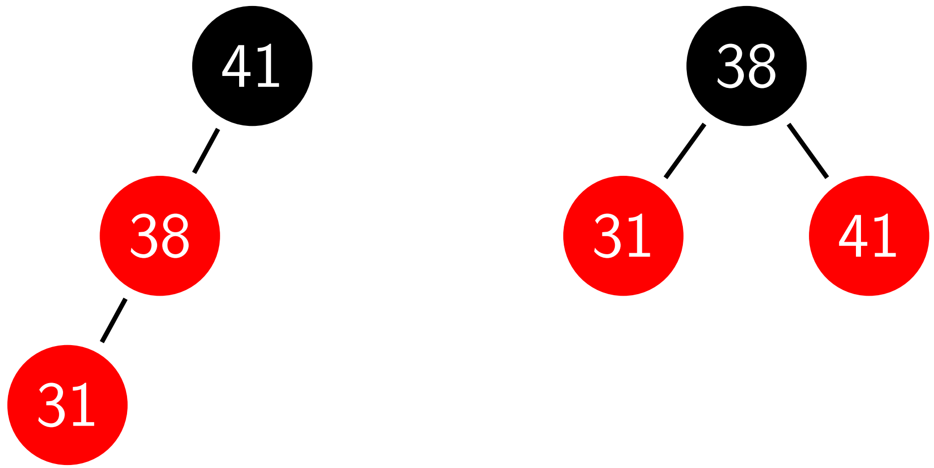


Figura: **inserimento di 31**

Inserimento: un esercizio

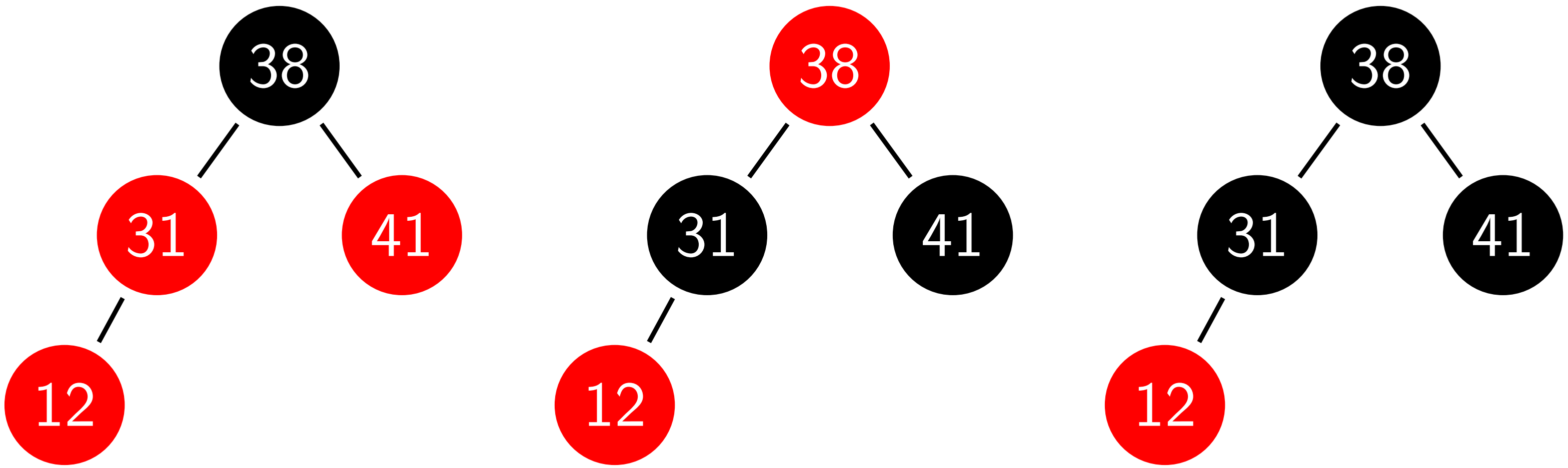


Figura: **inserimento di 12**

Inserimento: un esercizio

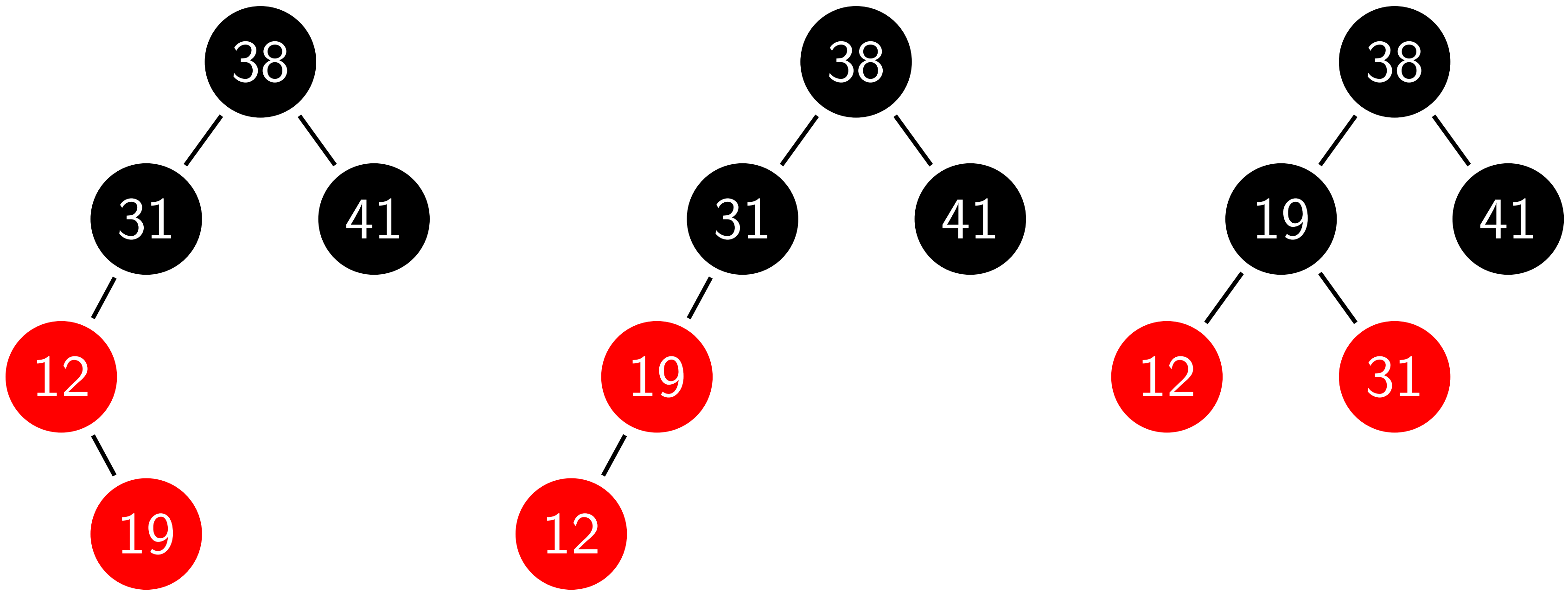


Figura: **inserimento di 19**

Inserimento: un esercizio

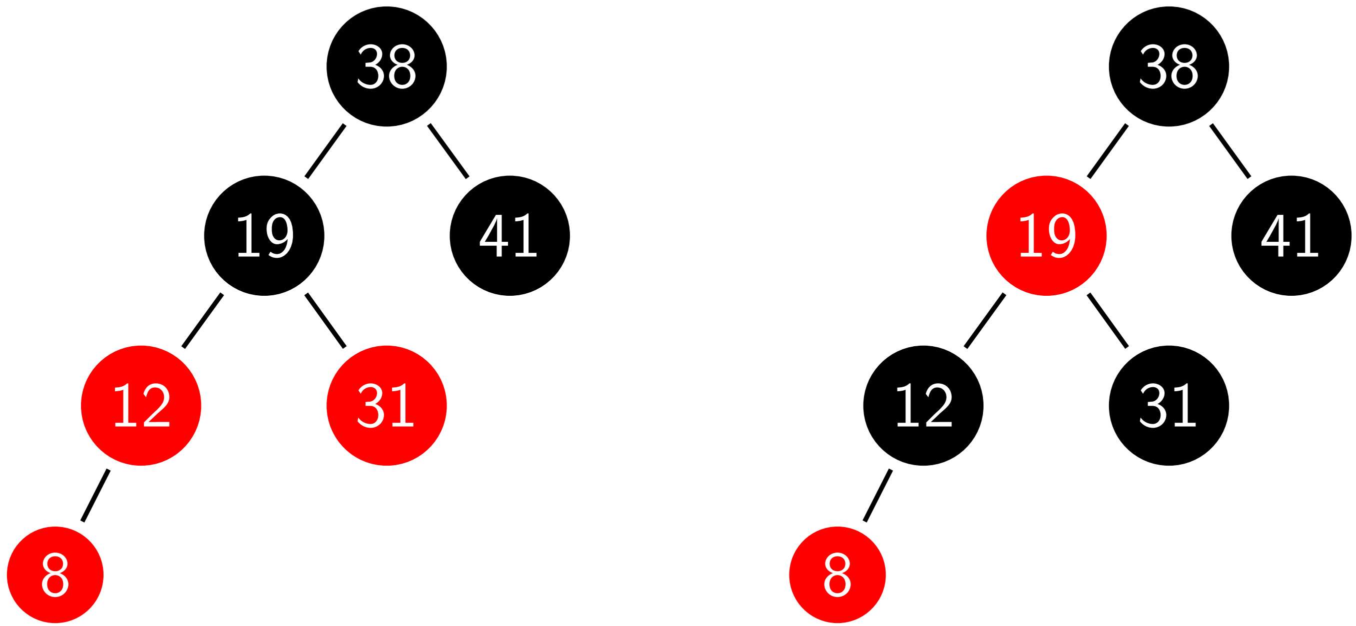


Figura: **inserimento di 8**

Cancellazione di un nodo con due figli

Nota: possiamo sempre assumere di 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 $y = \textsc{Tree-Successor}(T, z)$ (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; inoltre:

- y non può avere un figlio sinistro (altrimenti non sarebbe il nodo più a sinistra del sottoalbero destro)
- di conseguenza, y ha al più il figlio destro

Cancellazione di un nodo con al più un figlio

Sia z il nodo da cancellare, siano x e p l'unico figlio ed il padre di z, rispettivamente (**nota**: 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: possiamo semplicemente terminare 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

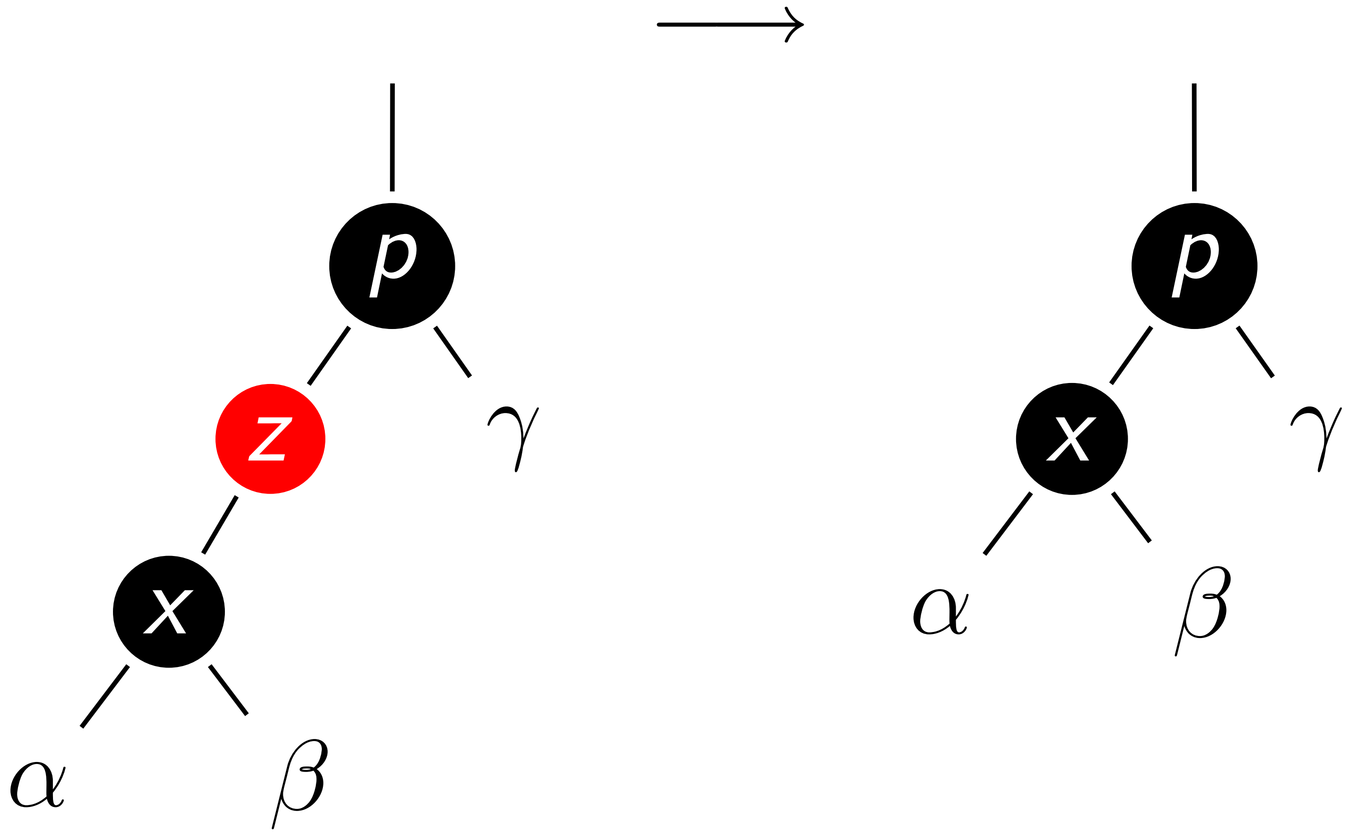


Figura: **Eliminare un nodo rosso non causa violazioni delle RB-properties**

Eliminazione: z nero e suo figlio x rosso

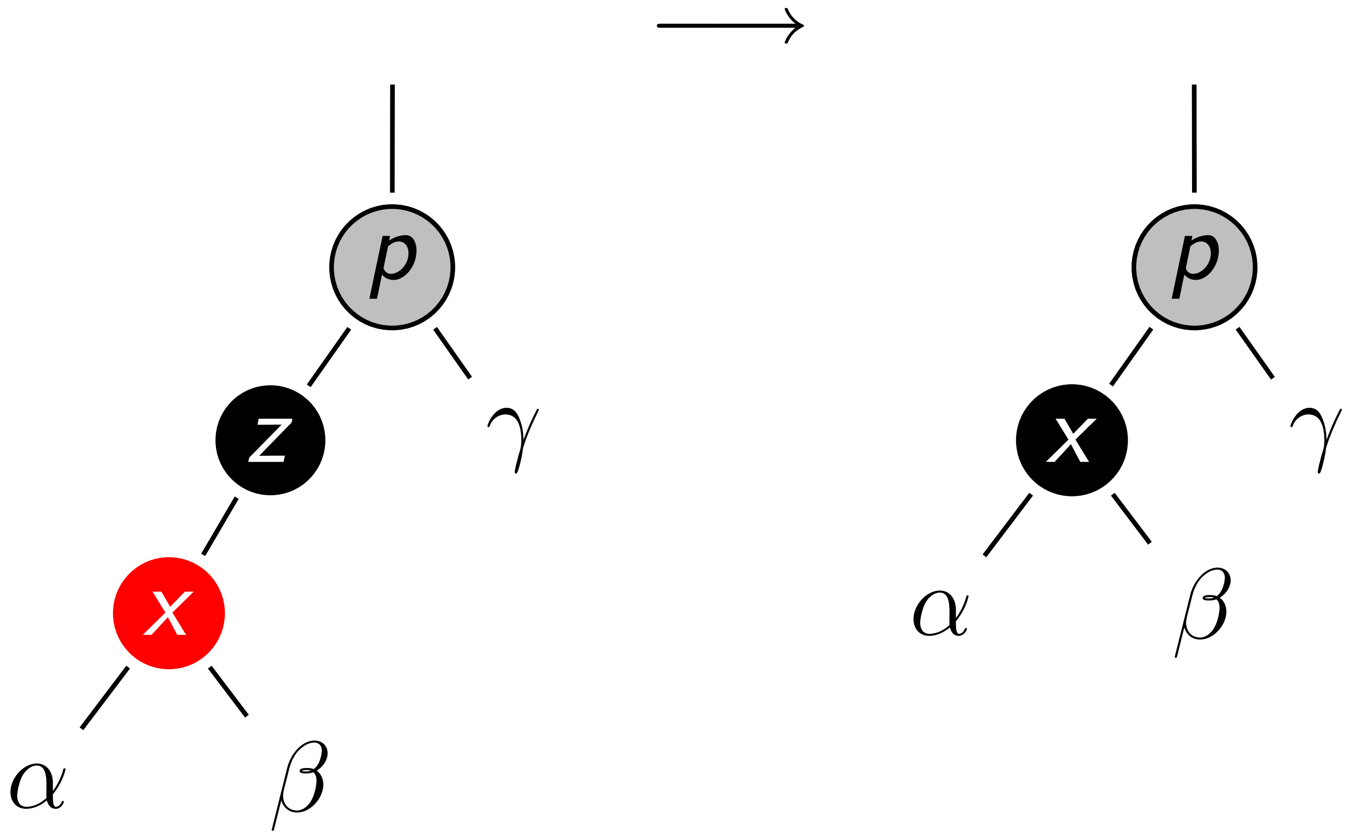


Figura: **Il figlio rosso di z acquisisce un extra credito diventando nero**

Eliminazione: z nero e suo figlio x nero

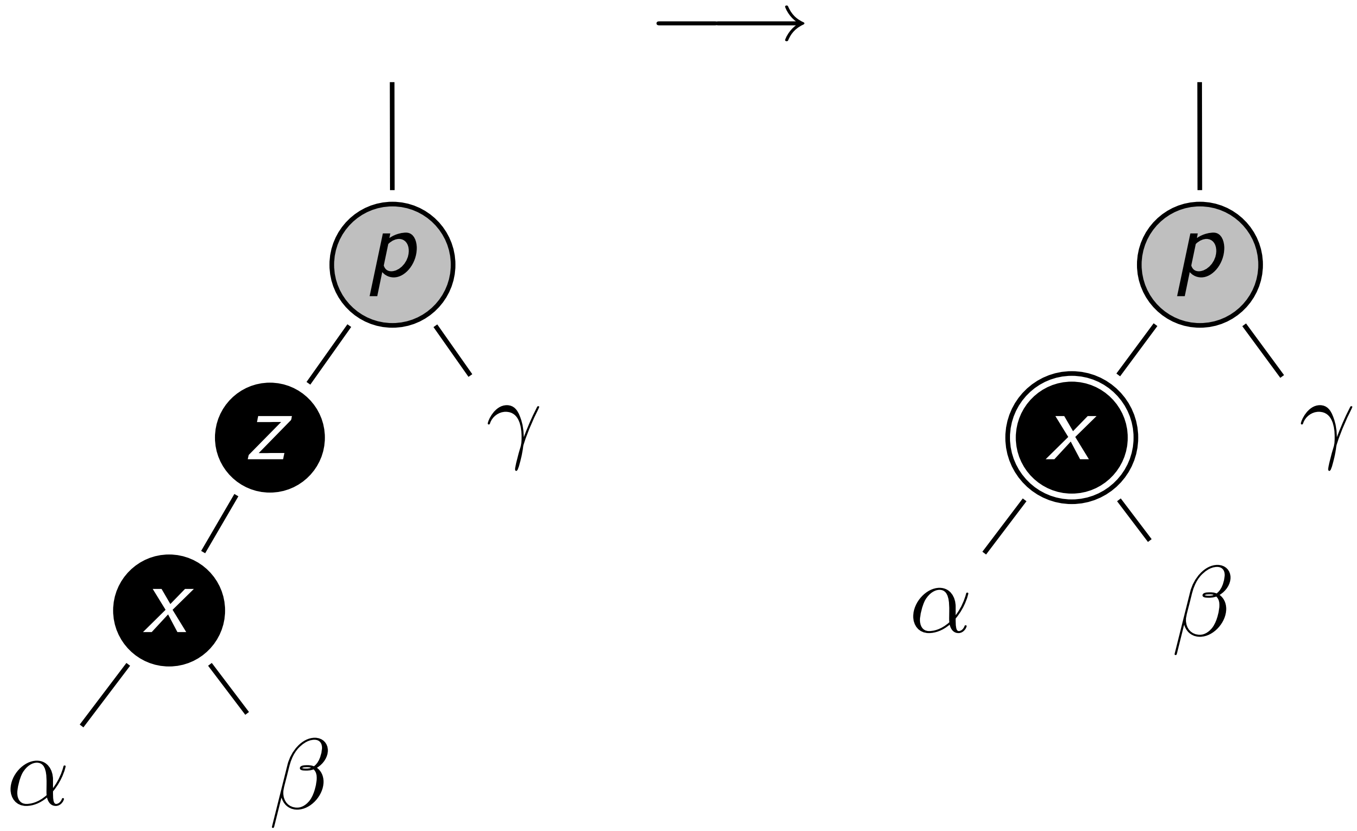


Figura: **Il figlio nero di z acquisisce un extra credito diventando "doppio nero". Eseguiamo RB-Delete-FixUp(T, x) per ripristinare la proprietà 5**

Cancellazione di un nodo con al più un figlio

Per ristabilire la proprietà 5 nel caso 3 (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 una procedura ($\textsc{RB-Delete-Fixup}(T, x)$) che spingerà, mediante rotazioni e ricolazioni, 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 sinistro di w è rosso e quello destro è nero
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

Caso 1.1: w nero e figlio destro di w rosso

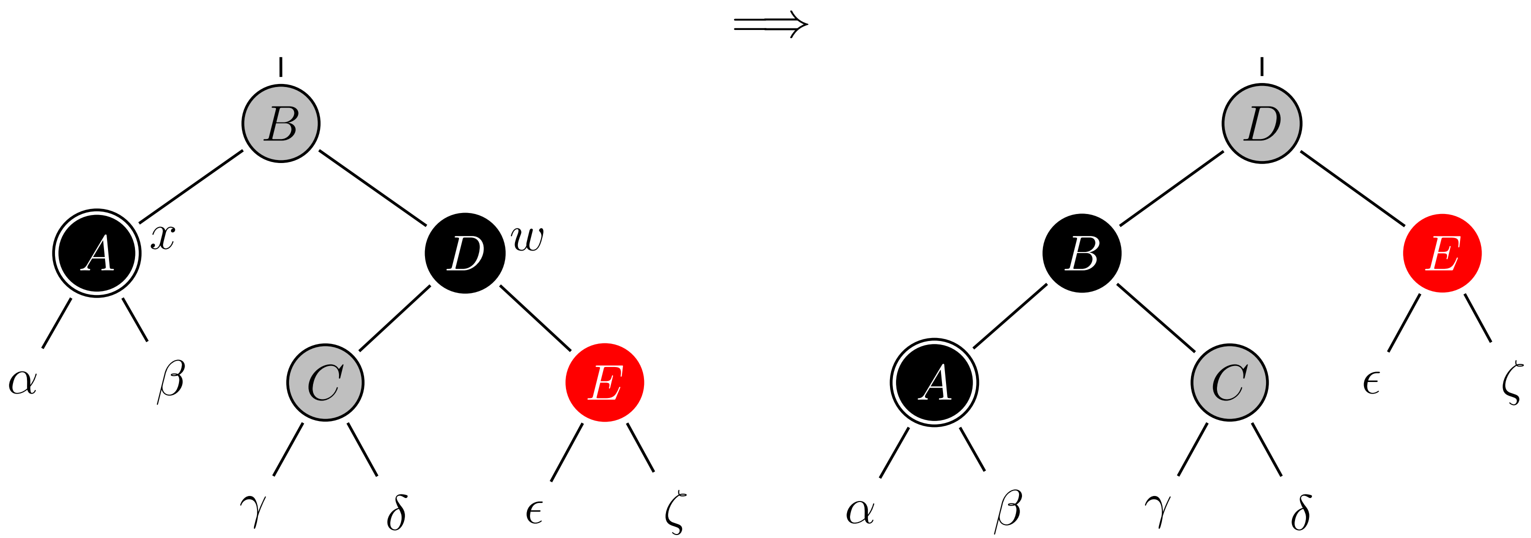


Figura: **scambiamo il colore di B con quello di w; ruotiamo B a sinistra**

1. abbiamo aggiunto un nodo nero a sinistra (possiamo togliere il doppio nero da A)
2. eliminato un livello nero a destra; coloriamo E di rosso

Caso 1.1: w nero e figlio destro di w rosso

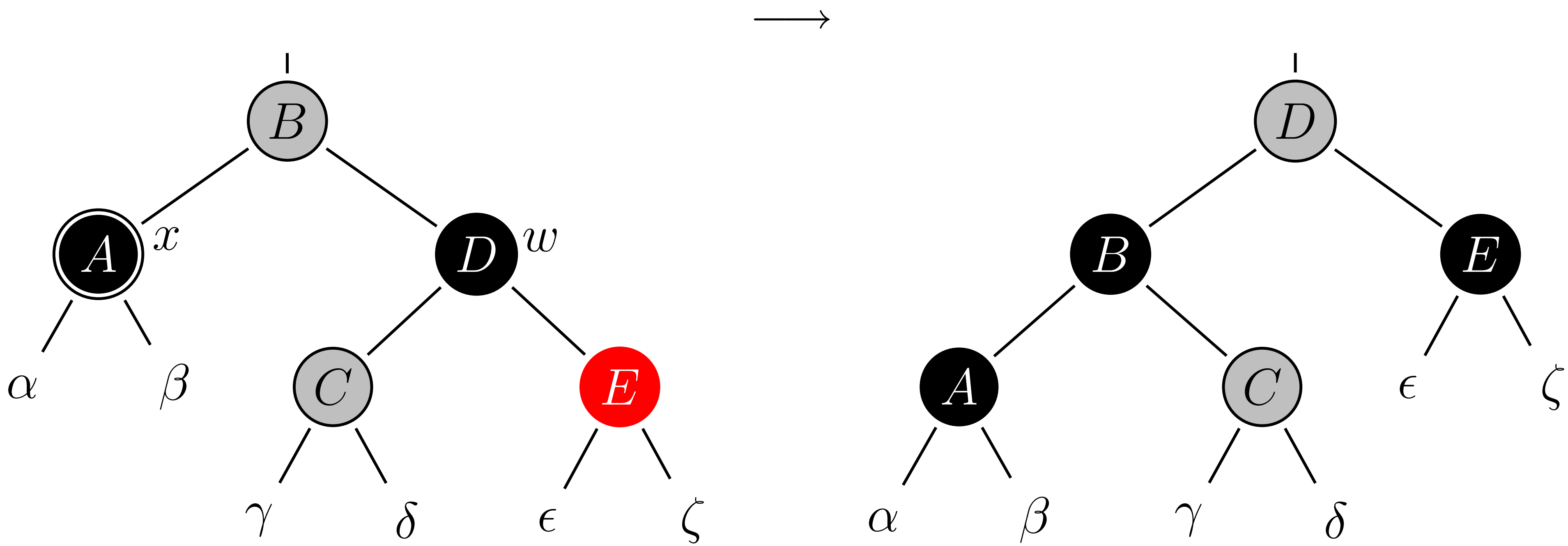


Figura: **eliminamo l'extra nero rappresentato da x colorando B (il padre di x) ed E (il figlio destro di w) di nero e ruotando B a sinistra**

RB-Delete-Fixup, caso 1.2: *w* è nero, il figlio sinistro di *w* è rosso e quello destro è nero

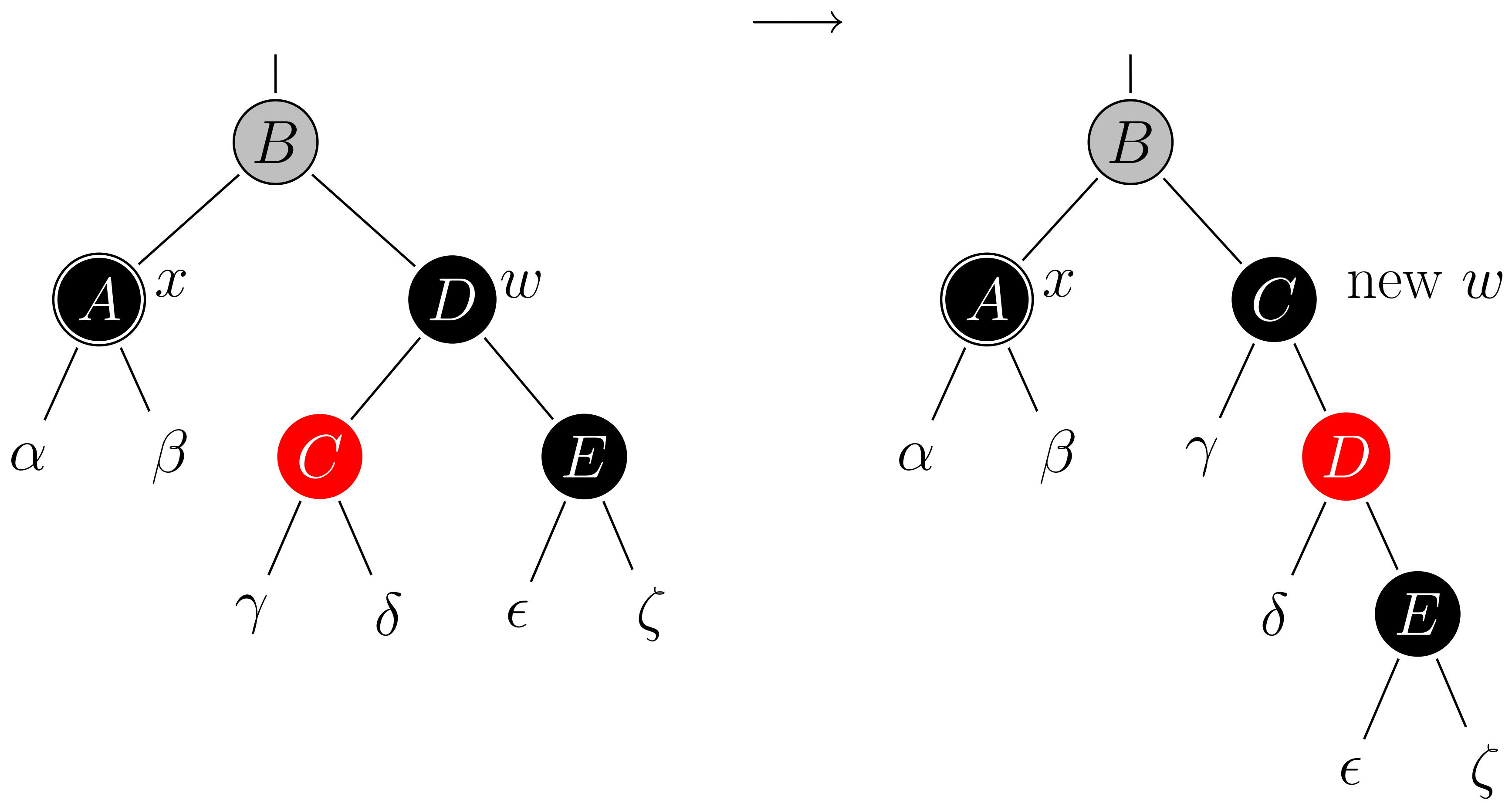


Figura: **questo caso è trasformato nel caso 1.1 colorando *C* (i.e. *left*[*w*]) di nero, *D* (i.e. *w*) di rosso e ruotando *D* a dest**

RB-Delete-Fixup, caso 2.1: *w* ha entrambi i figli neri e *p*[*x*] è rosso

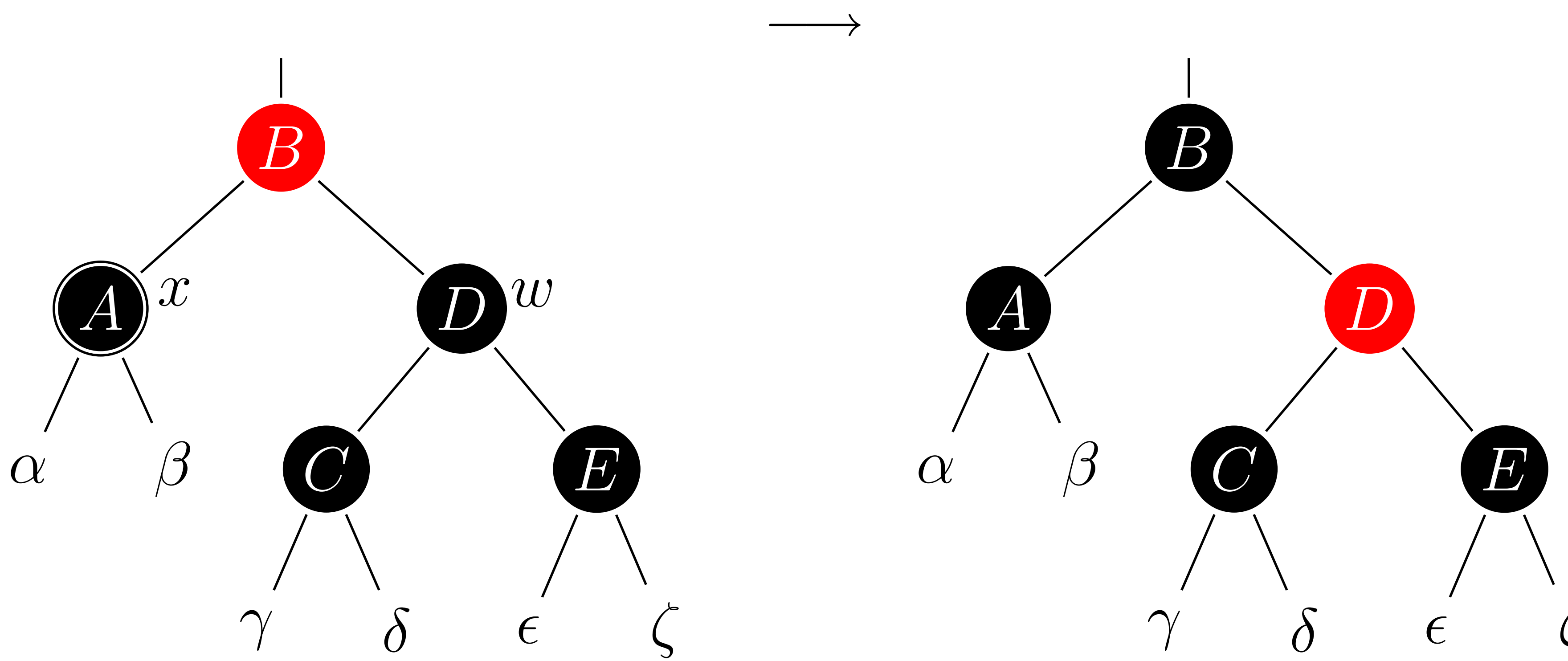


Figura: **eliminamo il doppio nero del nodo x togliendo un credito nero sia ad A che a D (quindi A diventa nero ordinario e D diventa rosso) e facendo acquisire un extra credito a B (il padre di x) che da rosso diventa nero**

RB-Delete-Fixup, caso 2.2: *w* ha entrambi i figli neri e $p[x]$ è nero

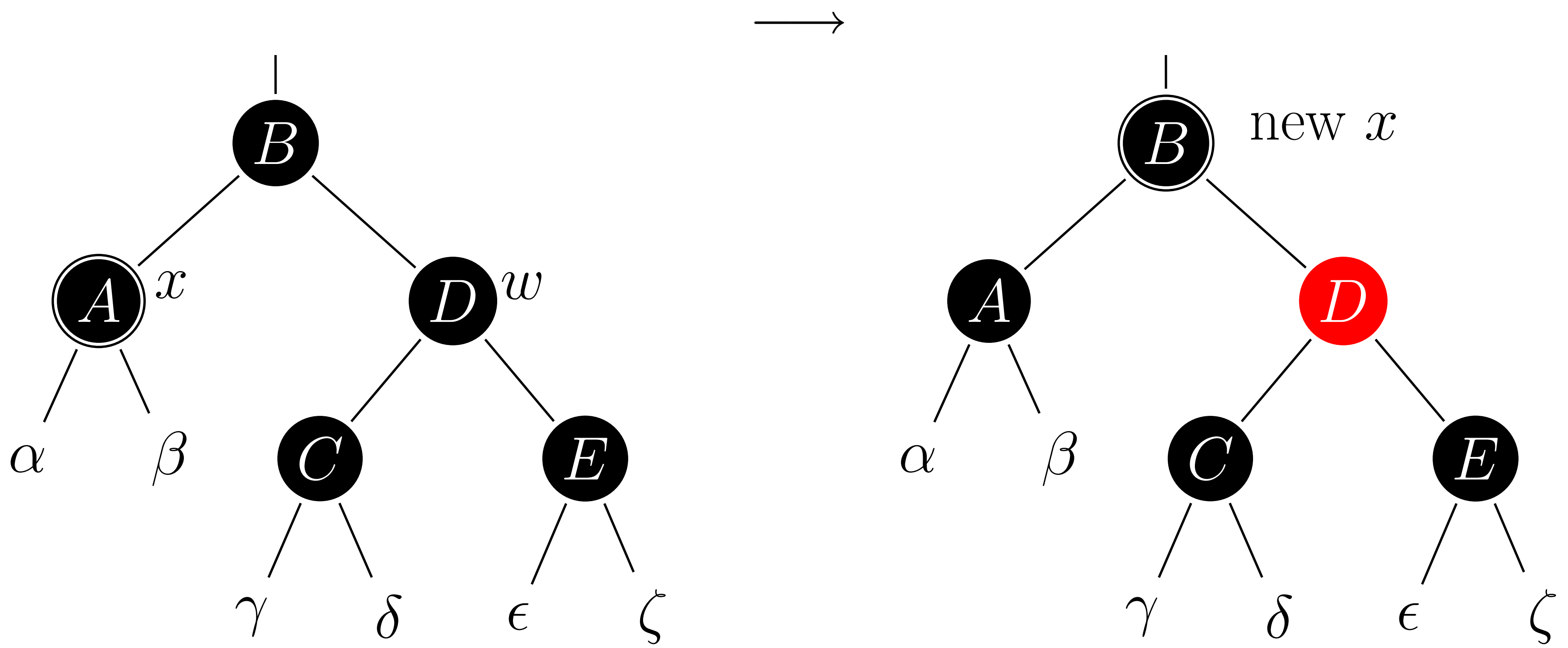


Figura: **è simile al caso 3.1, di nuovo togliamo un credito nero sia ad A che a D (A diventa nero ordinario e D diventa rosso) e facendo acquisire un extra credito a B che da nero diventa doppio-nero. A questo punto B diventa il nuovo x**

RB-Delete-Fixup: caso 3

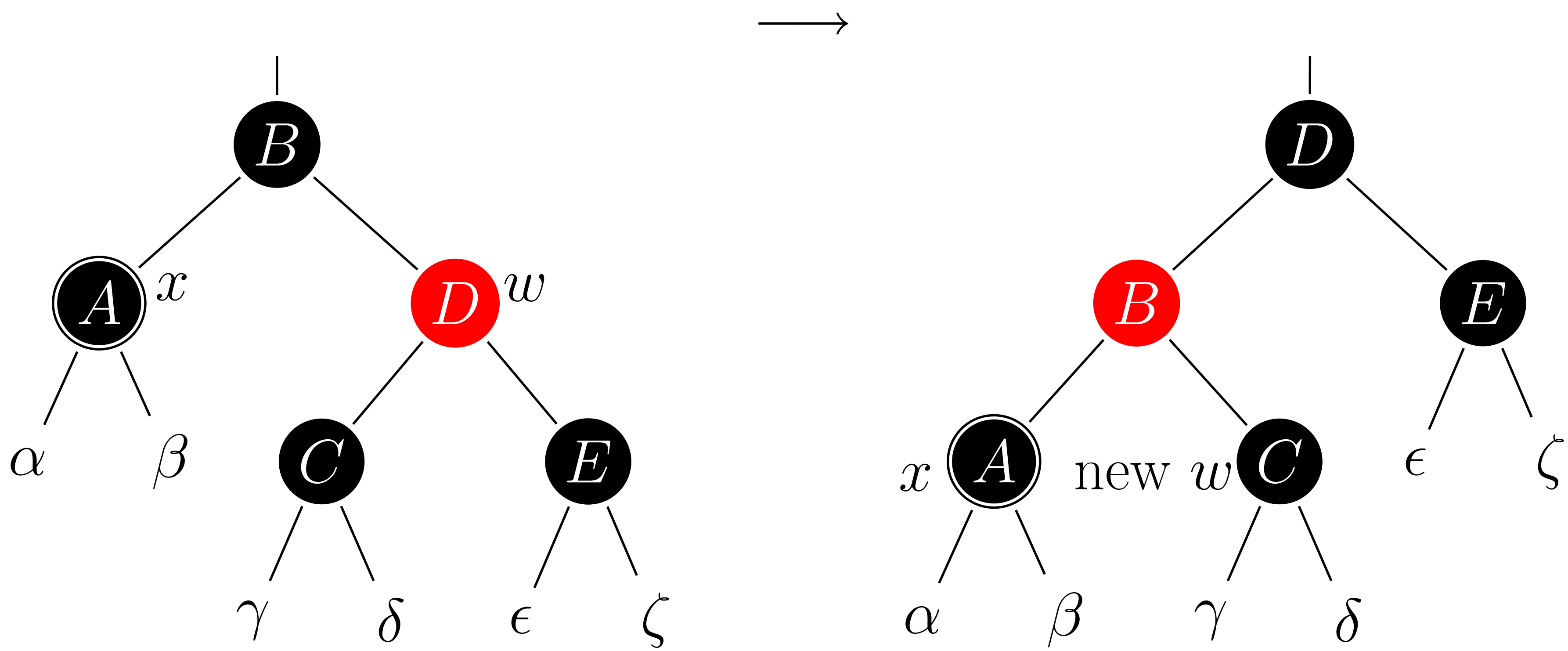


Figura: **viene trasformato in uno dei casi precedenti colorando D (i.e. w) di rosso, B (i.e. $p[x]$) di nero e ruotando il padre di x a sinistra**

Analisi

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

Cancellazione: un esempio

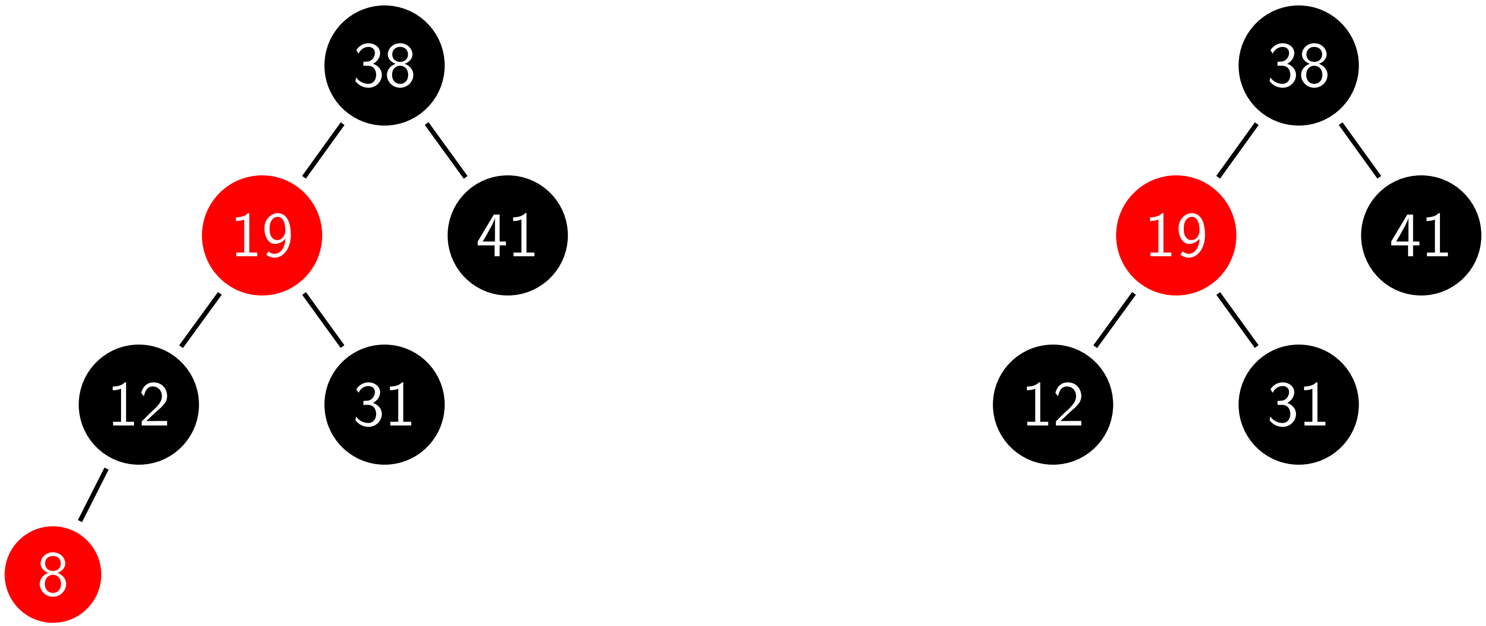


Figura: **cancellazione di 8**

Cancellazione: un esempio

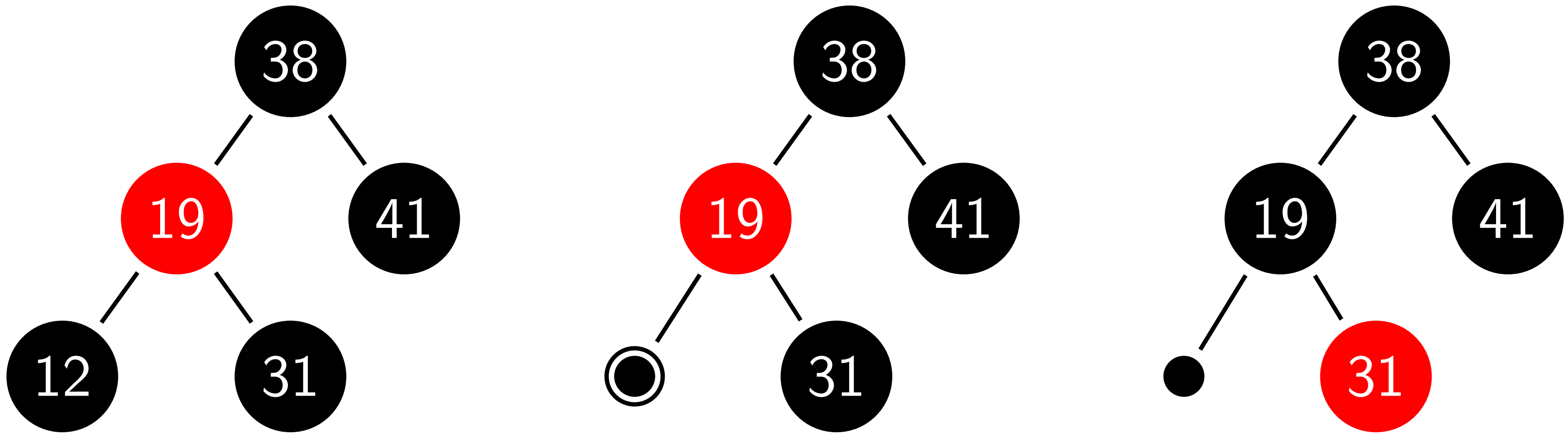


Figura: **cancellazione di 12**

Cancellazione: un esempio

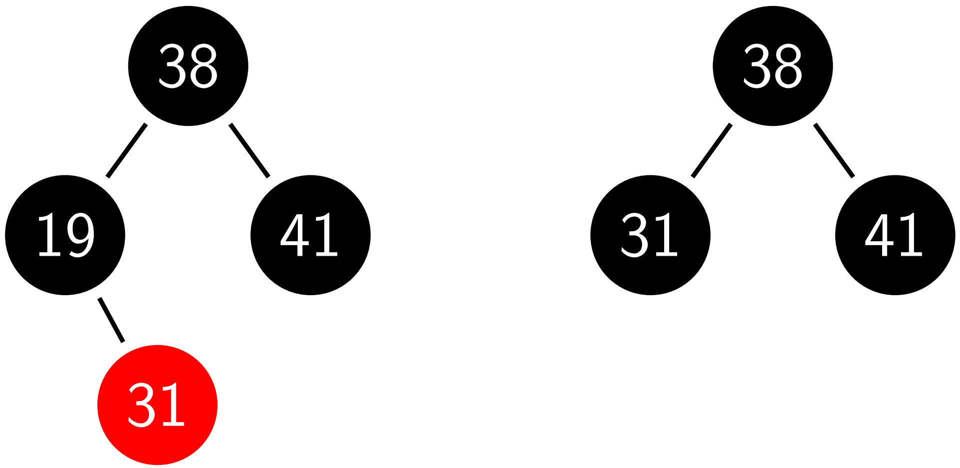


Figura: **cancellazione di 19**

Cancellazione: un esempio

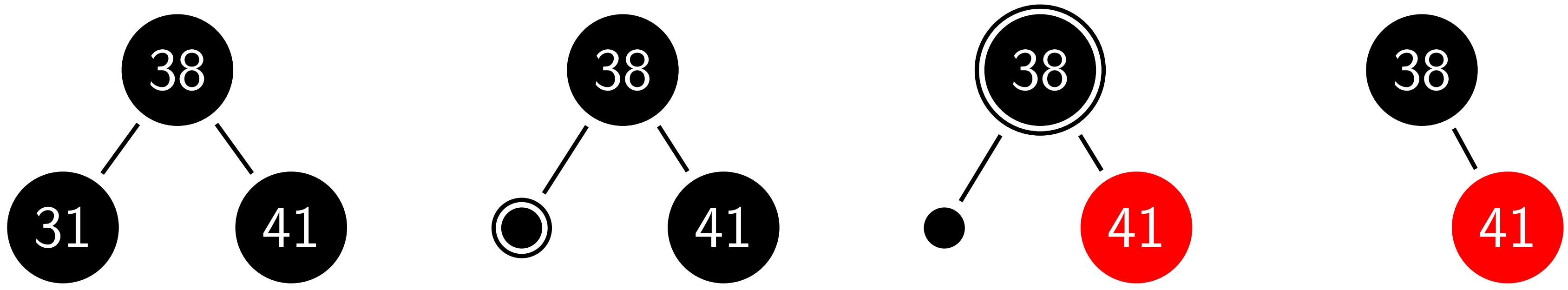


Figura: **cancellazione di 31**

Figura: **cancellazione di 38**