Alberi rosso-neri

Le operazioni sugli alberi binari di ricerca hanno complessità proporzionale all'altezza *h* dell'albero.

Gli alberi *rosso-neri* sono alberi binari di ricerca in cui le operazioni ***Insert*** e ***Delete*** sono opportunamente modificate per garantire un'altezza dell'albero

$$h = O(\log n)$$

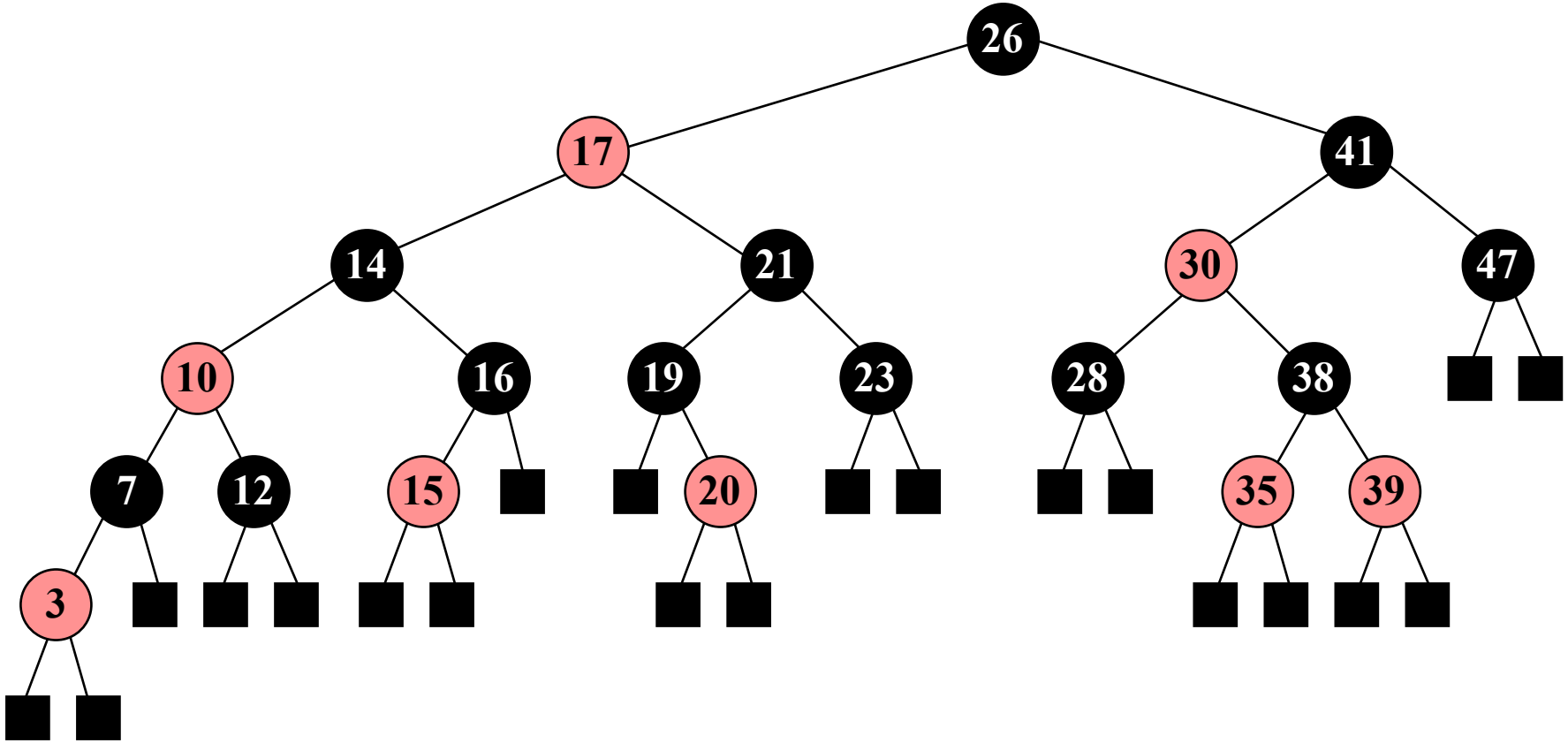
Bisogna aggiunge un bit ad ogni nodo: il *colore* che può essere ***rosso*** o ***nero.***

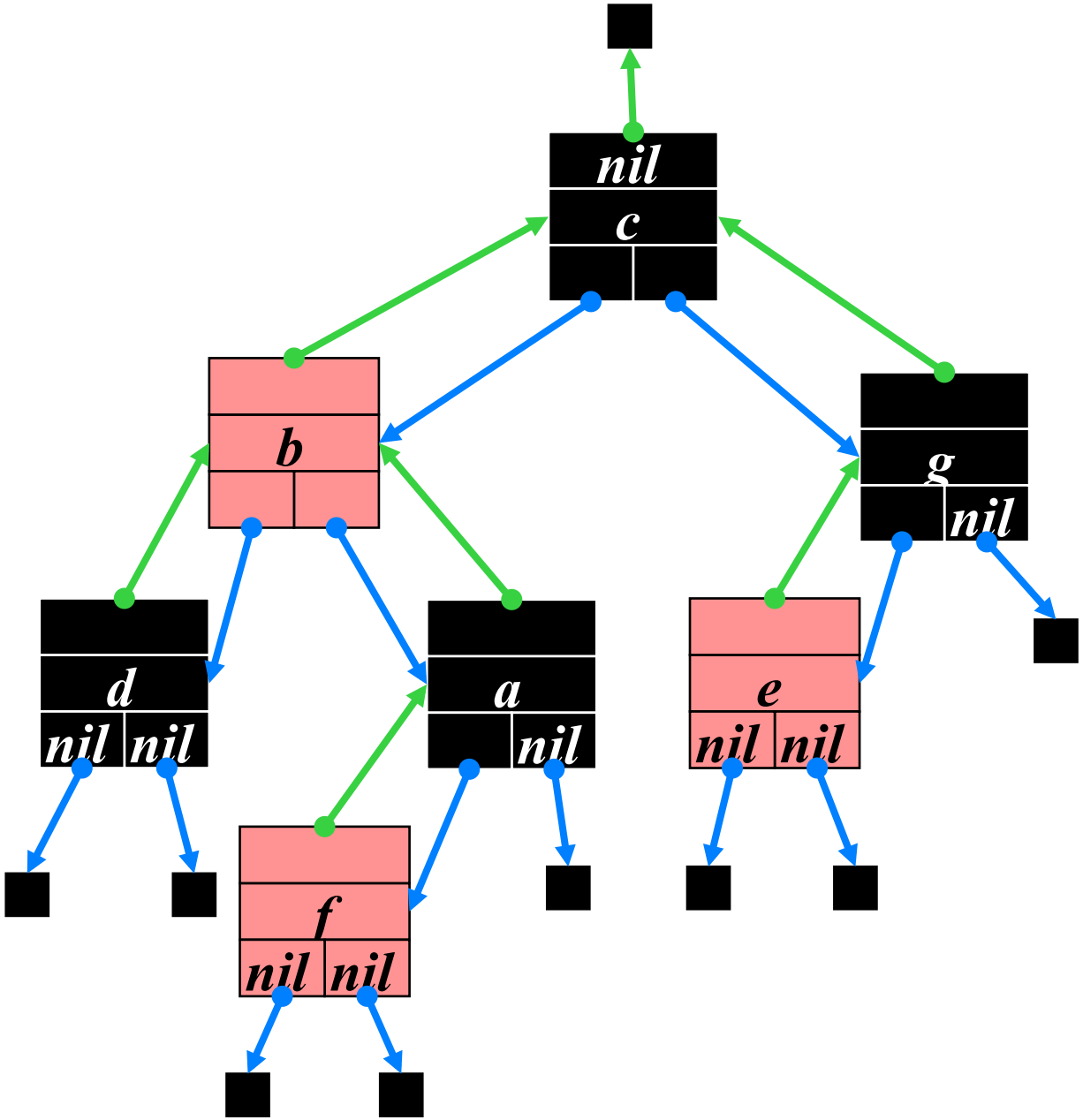
Oltre ad essere alberi binari di ricerca, gli alberi rosso-neri soddisfano le proprietà:

1. ogni nodo è o rosso o nero;
2. la radice è nera;
3. le foglie (***nil***) sono tutte nere;
4. i figli di un nodo rosso sono entrambi neri;
5. per ogni nodo ***x*** i cammini da ***x*** alle foglie sue discendenti contengono tutti lo stesso numero ***bh*(*x*)** di nodi neri: l'altezza nera di ***x***;

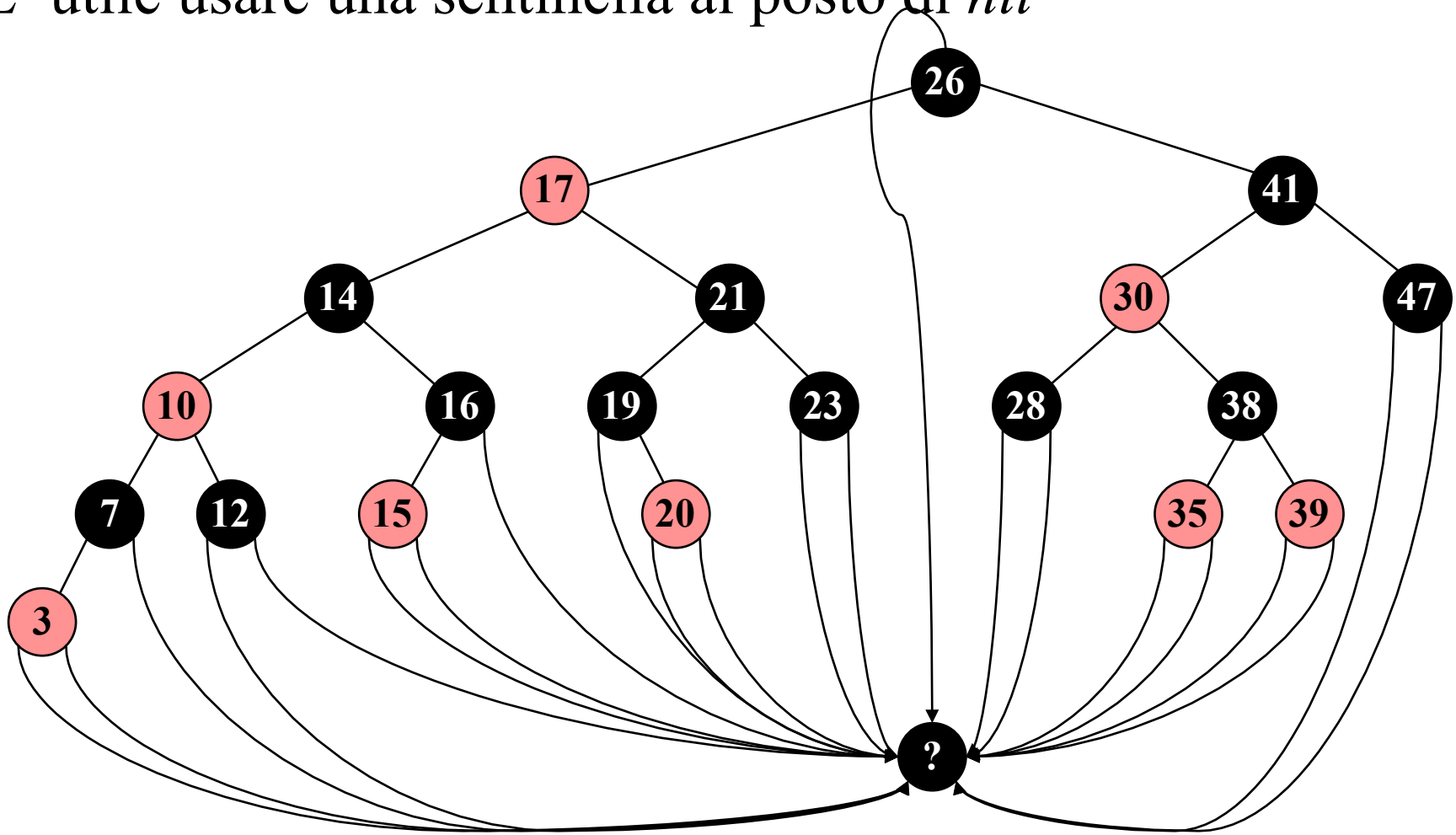
   Notare che il nodo ***x*** non viene contato in ***bh*(*x*)** anche se è nero.

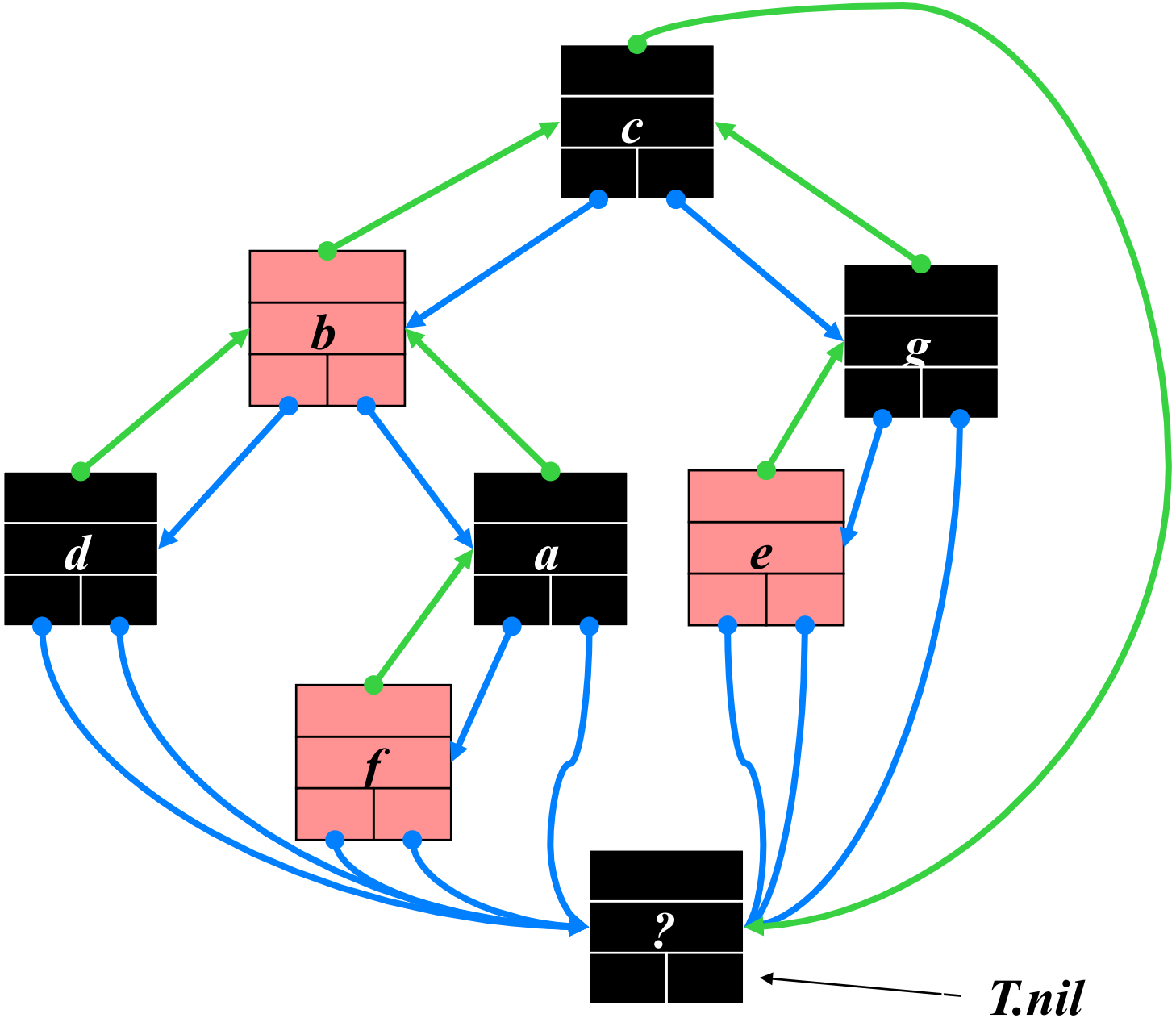
Esempio di albero rosso-nero:




nil
c
b
g
nil
d
nil
nil
a
nil
e
nil
nil
f
nil
nil

E’ utile usare una sentinella al posto di *nil*




c
b
g
d
a
e
f
?
T.nil

Proprietà: Un albero rosso-nero con n nodi interni ha altezza

$$h \leq 2 \log_2(n+1)$$

Dimostrazione: Osserviamo che i nodi rossi in un cammino dalla radice r alle foglie possono essere al più $bh(r)$ e quindi $h \leq 2\, bh(r)$.

Basta quindi dimostrare che $bh(r) \leq \log_2(n+1)$

ossia che $n \geq 2^{bh(r)} - 1$

Dimostriamo

$$n \geq 2^{bh(r)} - 1$$

per induzione sulla struttura dell'albero rosso-nero.

Se $T = \emptyset$ la radice r è una foglia, $bh(r) = 0$ e

$$n = 0 = 2^{bh(r)} - 1$$

Sia $T = (r, T_1, T_2)$ e siano r_1 ed r_2 le radici di T_1 e T_2 ed n_1 ed n_2 il numero di nodi interni di T_1 e T_2.
Allora:

$$bh(r_1) \geq bh(r)\text{-}1$$

$$bh(r_2) \geq bh(r)\text{-}1$$

$$n = 1 + n_1 + n_2$$

Per ipotesi induttiva

$$n \geq 1 + 2^{bh(r_1)} - 1 + 2^{bh(r_2)} - 1$$

$$\geq 1 + 2^{bh(r)-1} - 1 + 2^{bh(r)-1} - 1 = 2^{bh(r)} - 1$$

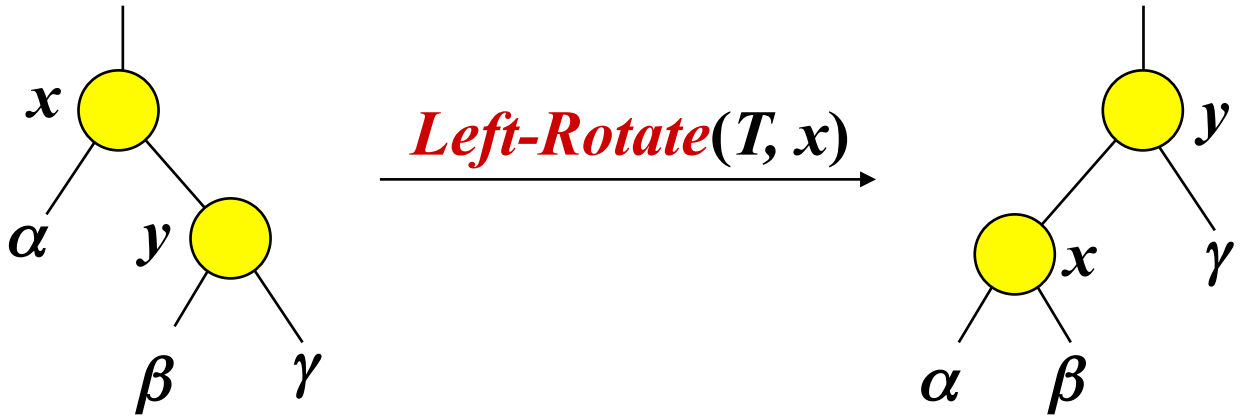
Conseguenza:

Su di un albero rosso-nero con n nodi interni le operazioni ***Search***, ***Minimum***, ***Maximum***, ***Successor*** e ***Predecessor*** richiedono tutte tempo

$$O(\log n)$$

Anche le operazioni ***Insert*** e ***Delete*** su di un albero rosso-nero richiedono tempo $O(\log n)$ ma siccome esse modificano l'albero possono introdurre delle violazioni alle proprietà degli alberi rosso-neri ed in tal caso occorre ripristinare le proprietà.

Per farlo useremo delle operazioni elementari, dette *rotazioni*, che preservano la proprietà di albero binario di ricerca.




```
Left-Rotate(T, x)
  y = x.right   // y non deve essere la sentinella T.nil
  x.right = y.left, y.left.p = x
  // y.left può essere T.nil
  y.p = x.p
  if x.p == T.nil
        T.root = y
  elseif x == x.p.left
        x.p.left = y
  else x.p.right = y
  x.p = y, y.left = x
```

Complessità Θ(1)

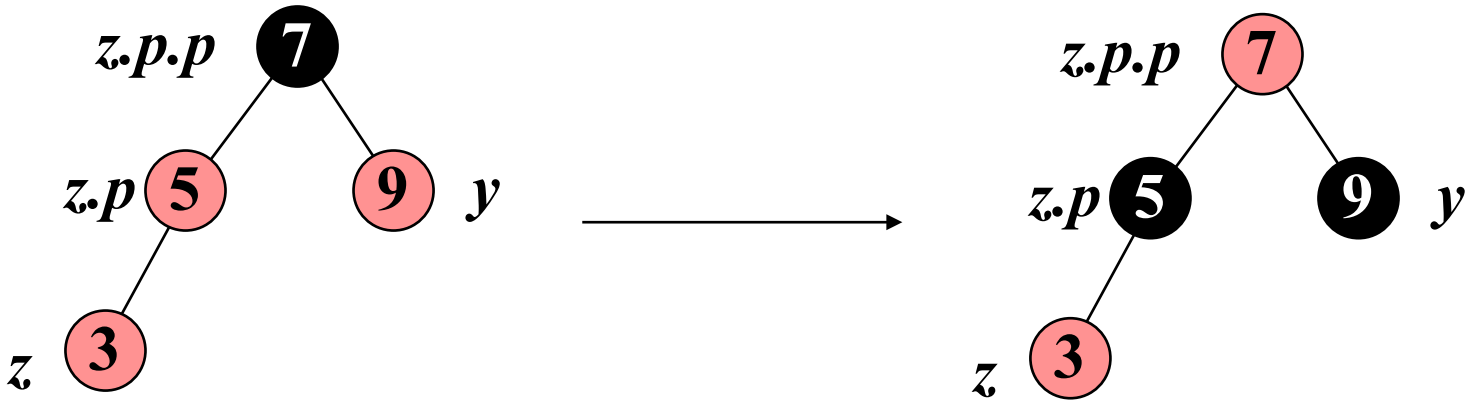

```
Right-Rotate(T, y)
    x = y.left // x non deve essere la sentinella T.nil
    y.left = x.right, x.right.p = y
    // x.right può essere T.nil
    x.p = y.p
    if y.p == T.nil
            T.root = x
    elseif y == y.p.left
            y.p.left = x
    else y.p.right = x
    y.p = x, x.right = y
```

Complessità Θ(1)

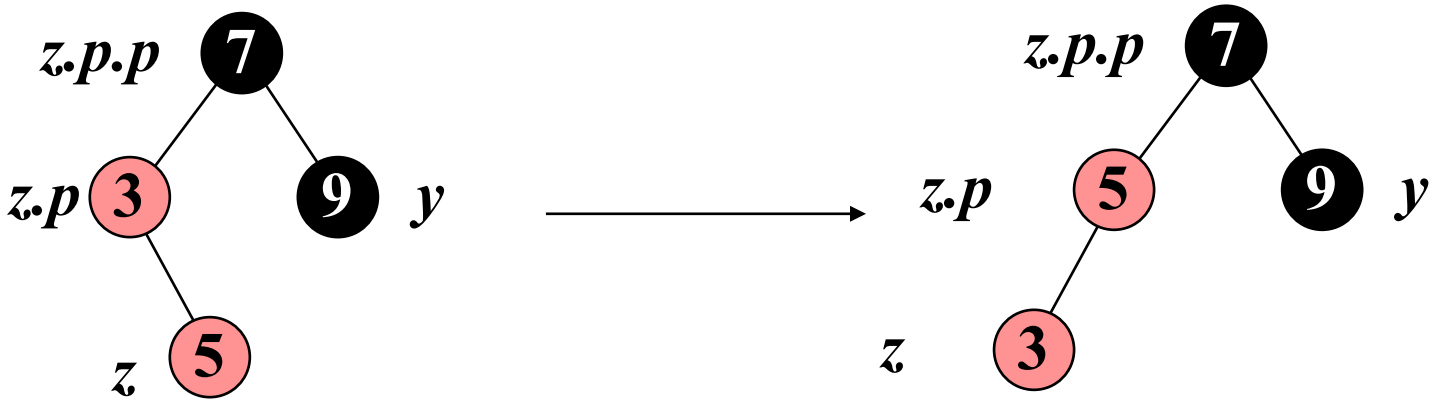
Inserimento di un nuovo elemento

```
RB-insert(T, z)   // z.left = z.right = T.nil
   Insert(T, z)
   z.color = RED
   // z è rosso. L'unica violazione
   // possibile delle proprietà degli alberi
   // rosso-neri è che z sia radice (prop. 2)
   // oppure che z.p sia rosso (prop. 4)
   RB-Insert-Fixup(T, z)
```

```
RB-Insert-Fixup(T, z)
  while z.p.color == RED
    // violata la proprietà 4
    if z.p == z.p.p.left   // l'altro caso è simmetrico
       y = z.p.p.right
       if y.color == RED                          // Caso 1
          z.p.color = y.color = BLACK
          z.p.p.color = RED
          z = z.p.p
```



```
else if z == z.p.right                    // Caso 2
        z = z.p
        Left-Rotate(T, z)
```



// *z* figlio sinistro // **Caso 3**

z.p.color = **BLACK**

z.p.p.color = **RED**

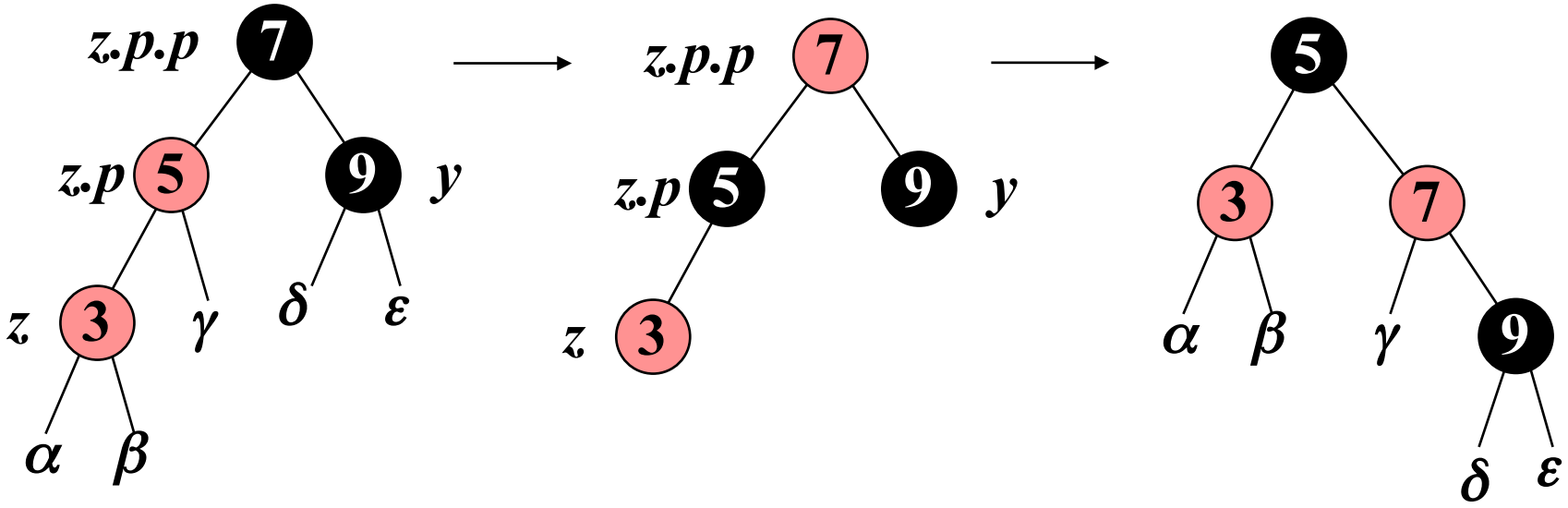
Right-Rotate(*T, z.p.p*)

else // simmetrico con ***right*** e ***left*** scambiati

// alla fine del ciclo l'unica proprietà violata può

// essere soltanto la **2**

T.root.color = **BLACK** // **Caso 0**



Complessità.

Ogni volta che si ripete il ciclo while il puntatore ***z*** risale di due posizioni.

Quindi il ciclo può essere ripetuto al più ***h*** volte e la complessità di ***RB-Insert-Fixup*** è $O(\log n)$.

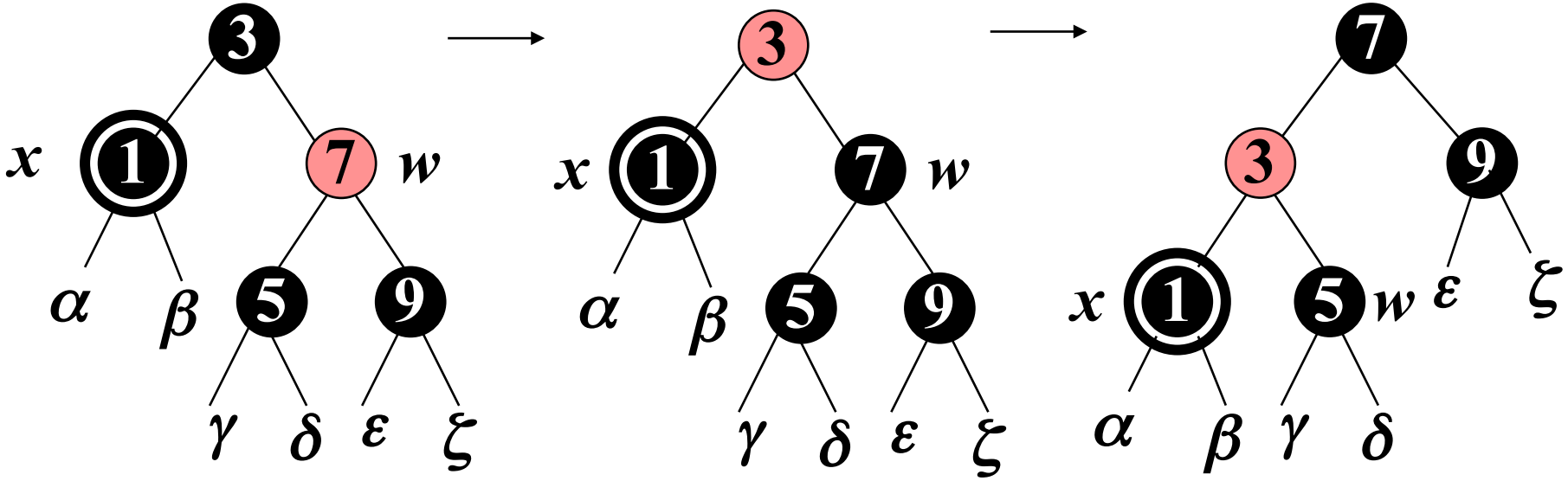
Quindi ***RB-Insert*** ha complessità $O(\log n)$.

Cancellazione di un elemento

```
Rb-Delete(T, z)                    // z ≠ T.nil
    if z.left == T.nil or z.right == T.nil
        y = z
    else
        y = Successor(z), z.key = y.key
    // elimino y che ha almeno un sottoalbero vuoto
    if y.left == T.nil
        x = y.right
    else
        x = y.left
     // x sottoalbero di y, l'altro è sicuramente vuoto
```

```
// metto x al posto del padre y
x.p = y.p
if y.p == T.nil
    T.root = x
elseif y == y.p.left
    y.p.left = x
else
    y.p.right = x
// Se y è rosso non ci sono violazioni
if y.color == BLACK
     // Se y era nero l'unica violazione delle
     // proprietà degli alberi rosso neri è che
     // i cammini che passano per x contengano
     // un nodo nero in meno
   RB-Delete-Fixup(T, x)
```

```
RB-Delete-Fixup(T, x)
  while x ≠ T.root and x.color == BLACK
    if x == x.p.left       // l'altro caso è simmetrico
      w = x.p.right
      if w.color == RED                 // Caso 1
        w.color = BLACK
        x.p.color = RED
        Left-Rotate(T, x.p)
        w = x.p.right
```



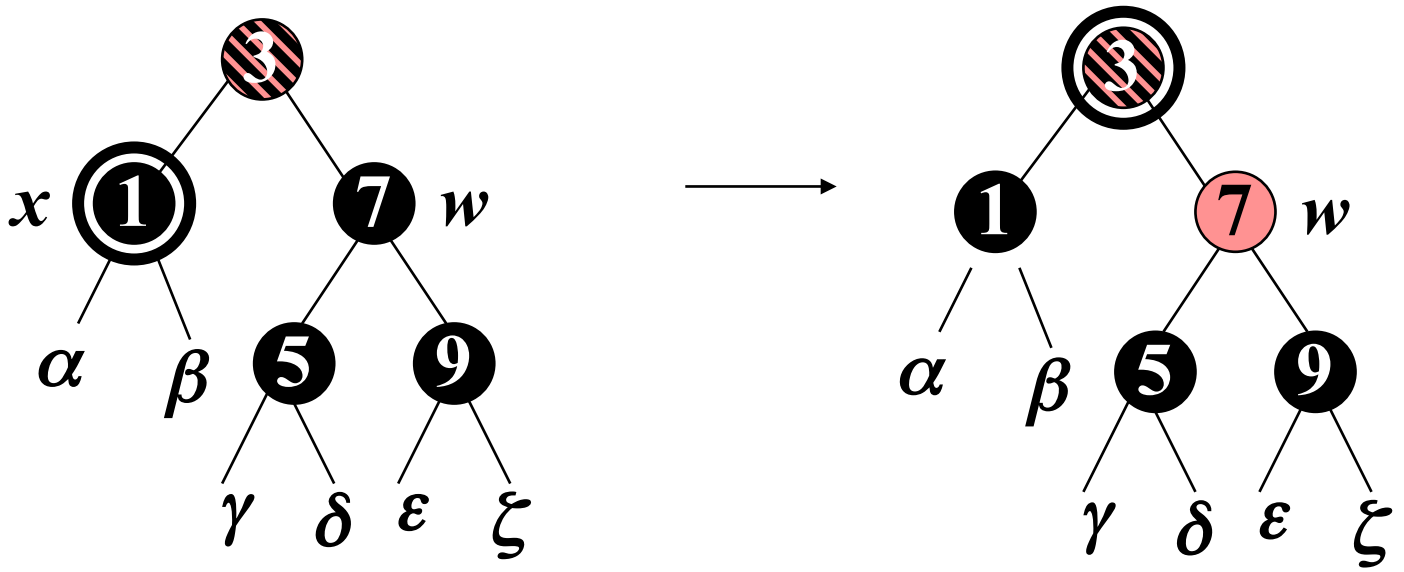
// il fratello *w* è nero

if ***w.left.color*** **== BLACK**

and ***w.right.color*** **== BLACK** **// Caso 2**

w.color **= RED**

x **=** ***x.p***



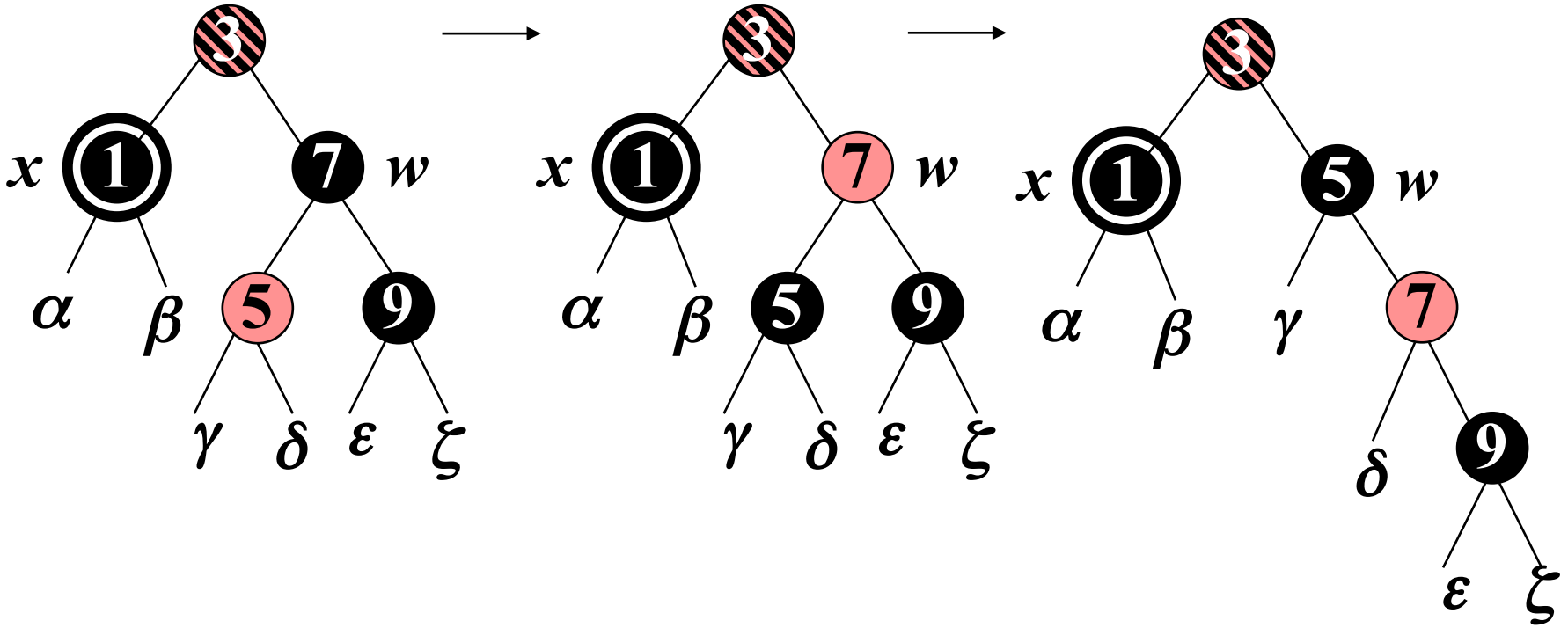
else if *w.right.color* == BLACK // Caso 3

w.left.color = BLACK

w.color = RED

Right-Rotate(*T*, *w*)

w = *x.p.right*



// **Caso 4**

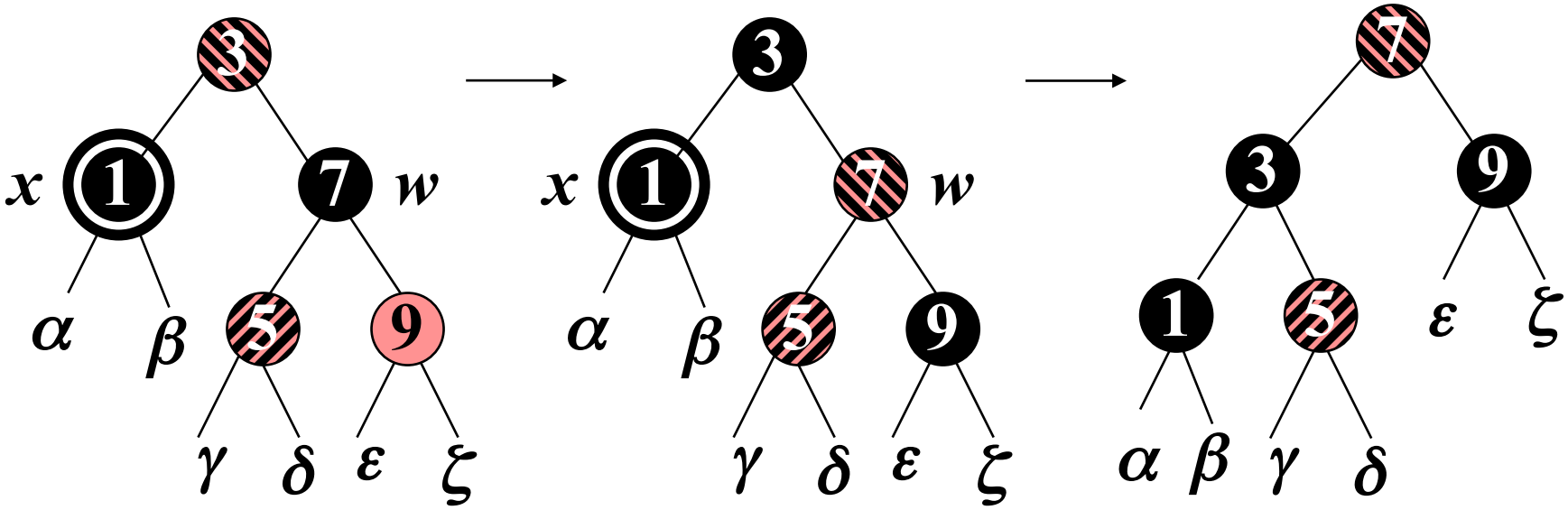
w.color = ***x.p.color***

x.p.color = ***w.right.color*** = **BLACK**

Left-Rotate(***T***, ***x.p***)

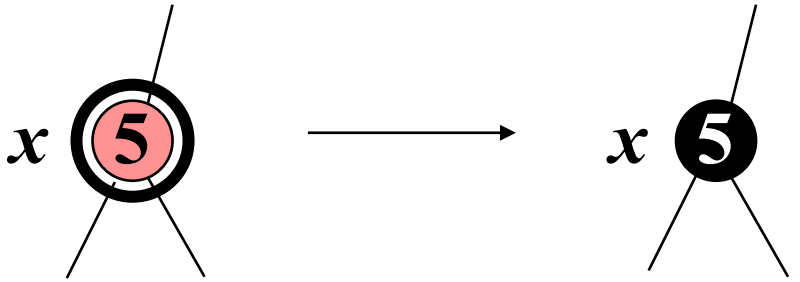
x = ***T.root***

else // simmetrico con ***right*** e ***left*** scambiati

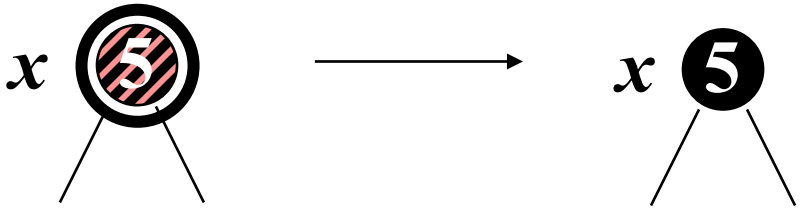


// quando termina il ciclo o ***x*** è la radice
// oppure ***x*** è rosso
x.color **= BLACK** **// Caso 0**

Caso 0: *x* rosso



Caso 0: *x* radice



Complessità di *RB-Delete-Fixup*.

Con i casi 0, 3 e 4 il ciclo **while** termina immediatamente e dunque essi richiedono tempo costante.

Dopo aver eseguito il caso 1 viene eseguito una sola volta il caso 2 e poi uno dei casi 0, 3 o 4. Quindi anche il caso 1 richiede tempo costante.

Solo il caso 2 può essere ripetuto sul padre di ***x*.**

Quindi il ciclo può essere ripetuto al più ***h*** volte e la complessità è $O(\log n)$.