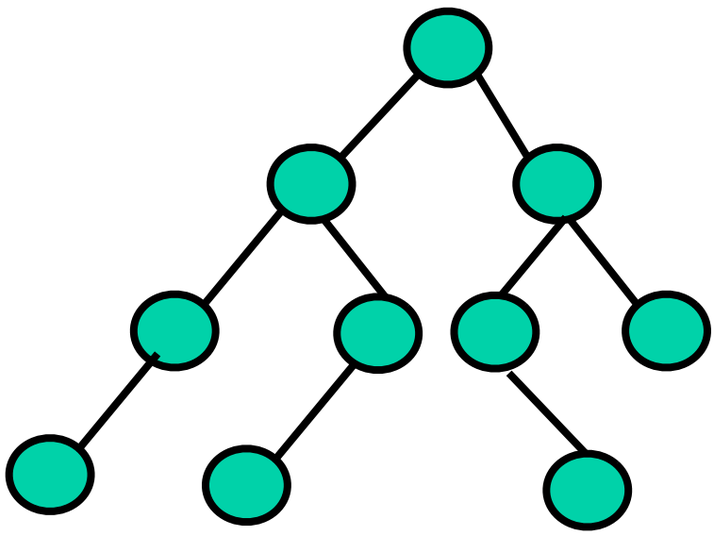
In questa lezione

Alberi rosso-neri

ABR di altezza logaritmica

Esistono vari criteri di bilanciamento di alberi binari che ne garantiscono l'altezza logaritmica, per esempio:

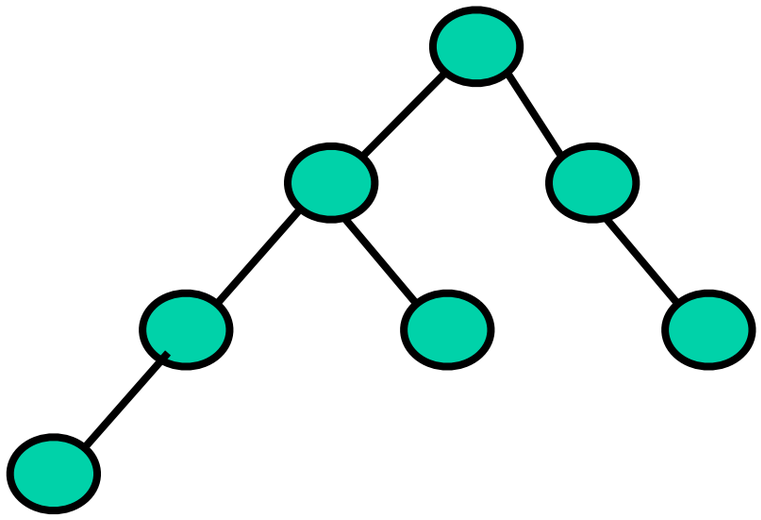
- **bilanciamento nel numero dei nodi: per ogni nodo la differenza in valore assoluto tra il numero dei nodi del sottoalbero sinistro di v e il numero dei nodi del sottoalbero destro di v è ≤ 1**



bilanciato nel numero dei nodi

ABR di altezza logaritmica

- **bilanciamento in altezza (alberi AVL, 1962, acronimo dagli autori Georgy Maximovich Adel'son-Vel'skii e Yevgeniy Mikhailovich Landis): per ogni nodo la differenza in valore assoluto tra l'altezza del sottoalbero sinistro di v e l' altezza del sottoalbero destro di v è ≤ 1.**



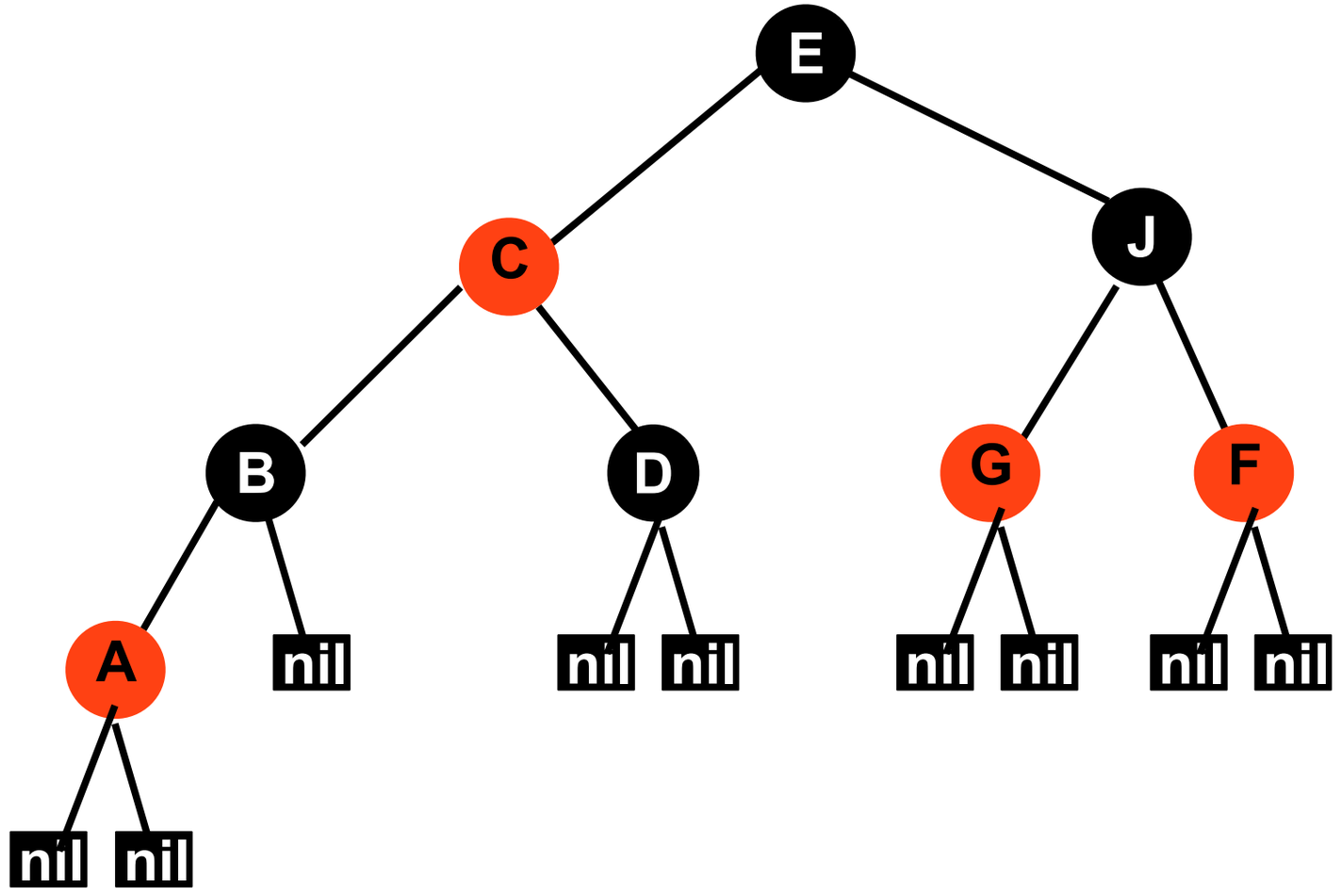
bilanciato in altezza

e non nel numero dei nodi

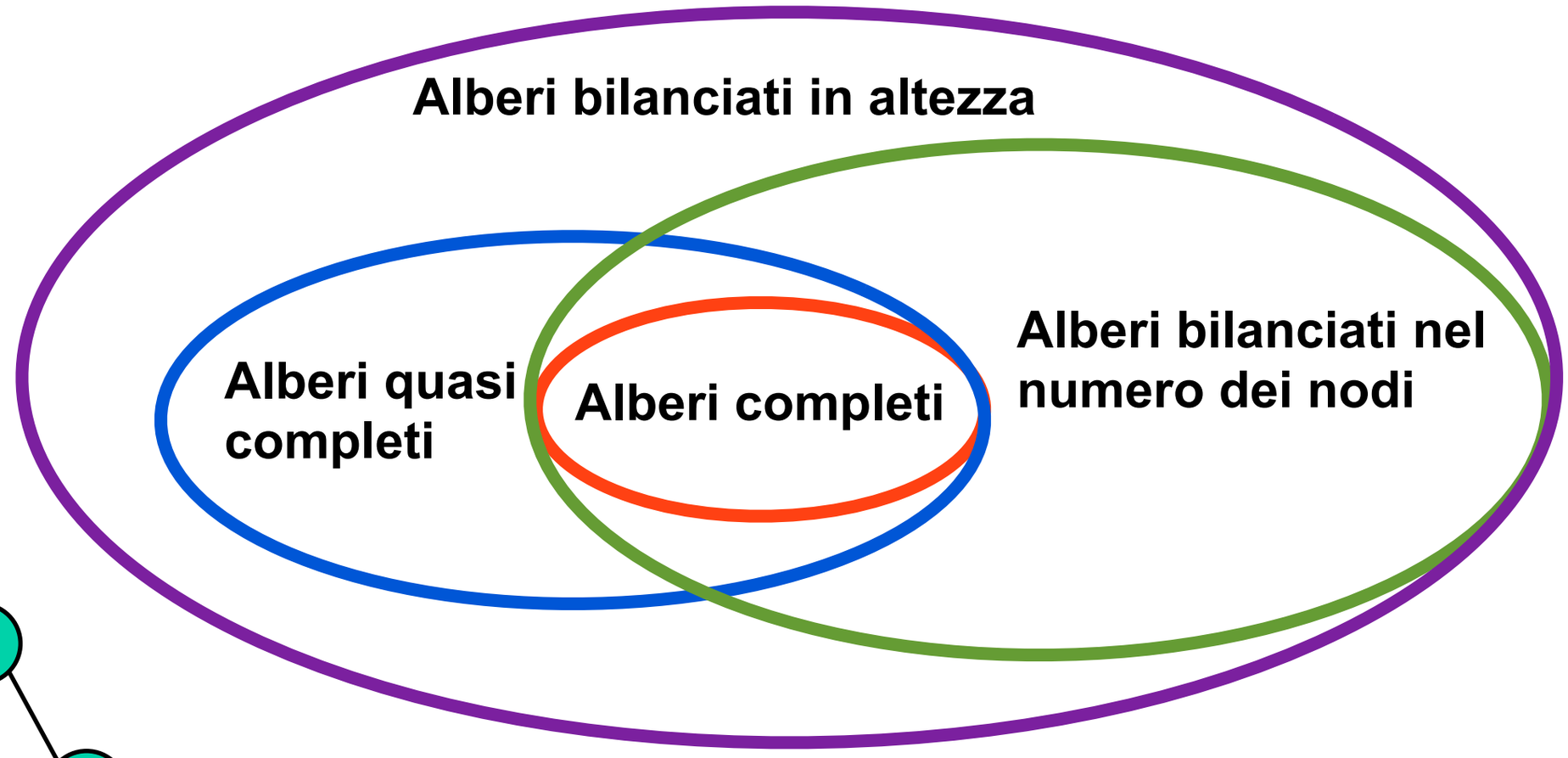
ABR di altezza logaritmica

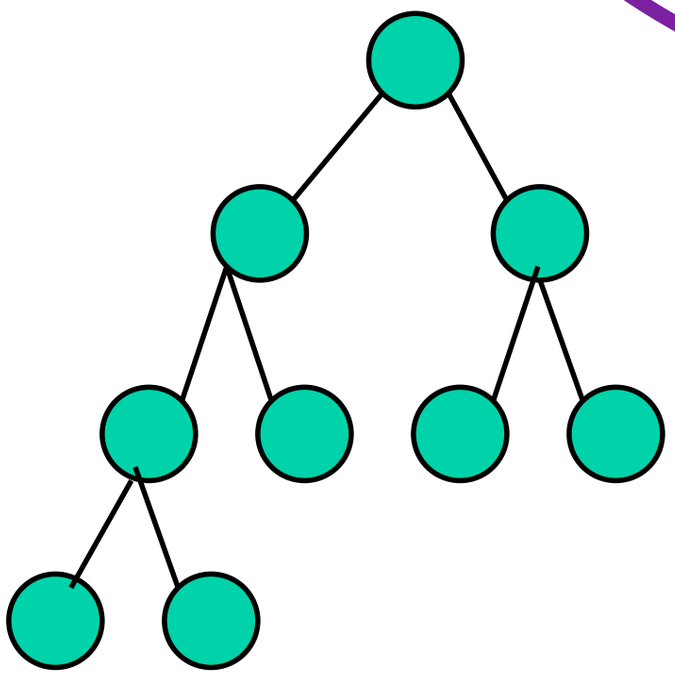
- **alberi rosso-neri, introdotti nel 1972 da Rudolf Bayer, professore alla Technische Universität München, con il nome di "B-alberi simmetrici". Leonidas J. Guibas, Professore alla Stanford University e di Robert Sedgewick, professore alla Princeton University ne provano molte proprietà, chiamandoli alberi rosso-neri, in un successivo articolo, apparso nel 1978. Una variante, gli alberi rossi-a-sinistra-neri è stata introdotta nel 2007 da Sedgewick.**

Ogni nodo in un albero rosso-nero è colorato di rosso o di nero, ma la colorazione deve soddisfare dei vincoli che garantiscono che nessun percorso radice-foglia è lungo più del doppio di ogni altro, così l'albero è abbastanza bilanciato da avere altezza logaritmica nel numero dei nodi.



Relazioni tra classi di alberi





Questo e' un esempio di albero quasi completo non bilanciato nel numero dei nodi

Le operazioni

Perchè le operazioni di ricerca, inserimento e cancellazione siano di complessità O(log n), dove n è il numero dei nodi dell'albero bisogna che le operazioni di ribilanciamento dell'albero, sbilanciato a seguito di inserimenti o cancellazioni, siano di complessità O(log n).

Gli alberi bilanciati in altezza o alberi AVL, sono i primi per i quali si ottiene questo risultato poi ottenuto anche per gli alberi rosso-neri. Questi alberi sono quindi detti autobilancianti (self-balancing BST)

Esistono altri alberi di ricerca (non binari) con i quali possiamo implementare un dizionario di n elementi in tempo O(log n):

2-3 alberi

2-3-4-alberi

B-alberi

} **si trasformano in alberi rosso-neri**

Alberi rosso-neri: motivazioni

Poichè per i rosso-neri le operazioni di ricerca, inserimento e cancellazione sono considerate le più efficienti, numerose tabelle dei simboli di sistemi esistenti, compilatori C++, Java, Python, ecc., sono implementate con gli alberi rosso-neri.

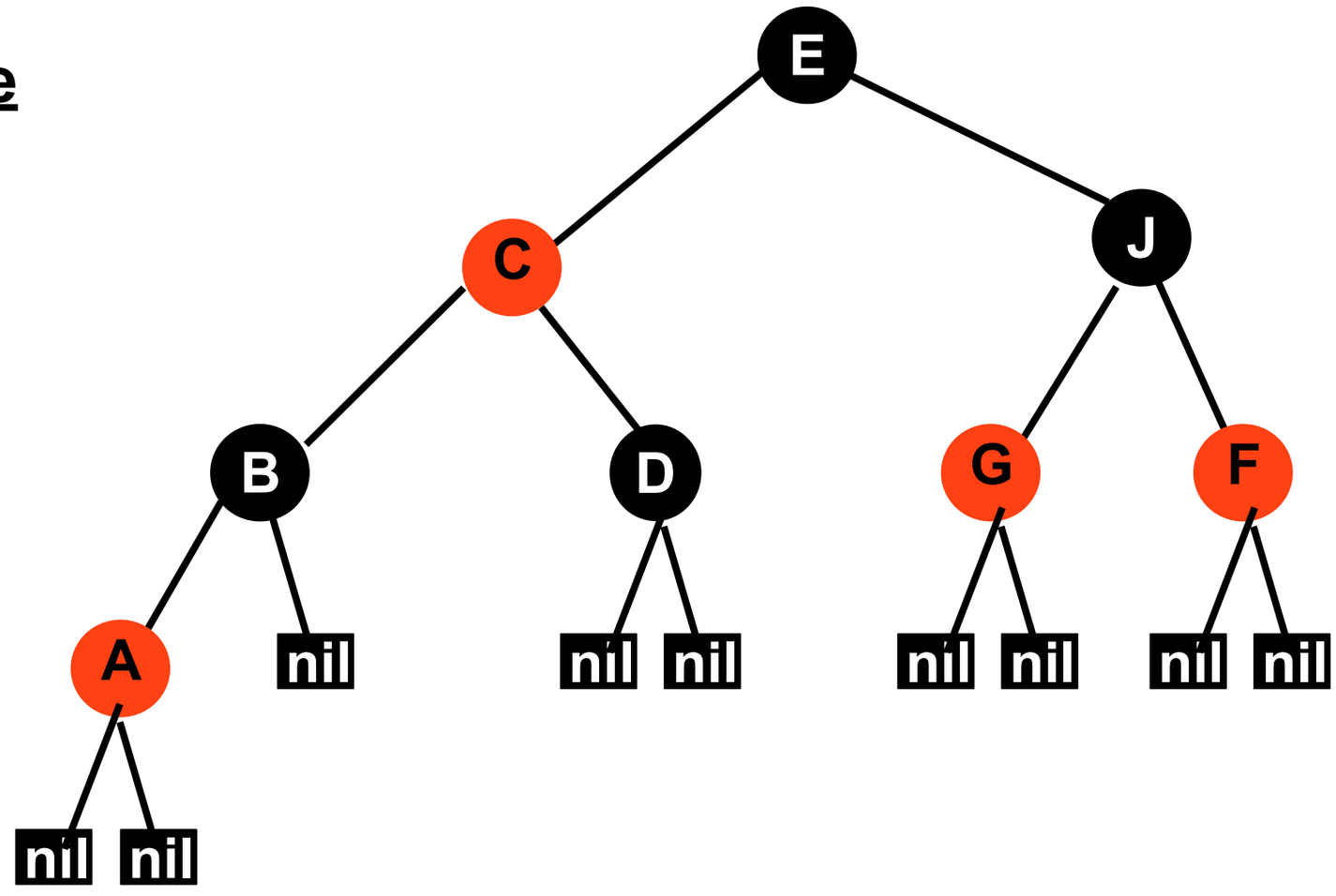
Inoltre tutti gli algoritmi noti su alberi bilanciati come gli alberi 2-3 o 2-3-4 e gli AVL possono essere implementati sugli alberi rosso-neri.

Alberi rosso-neri

Un albero rosso-nero è un ABR che rispetta queste proprietà:

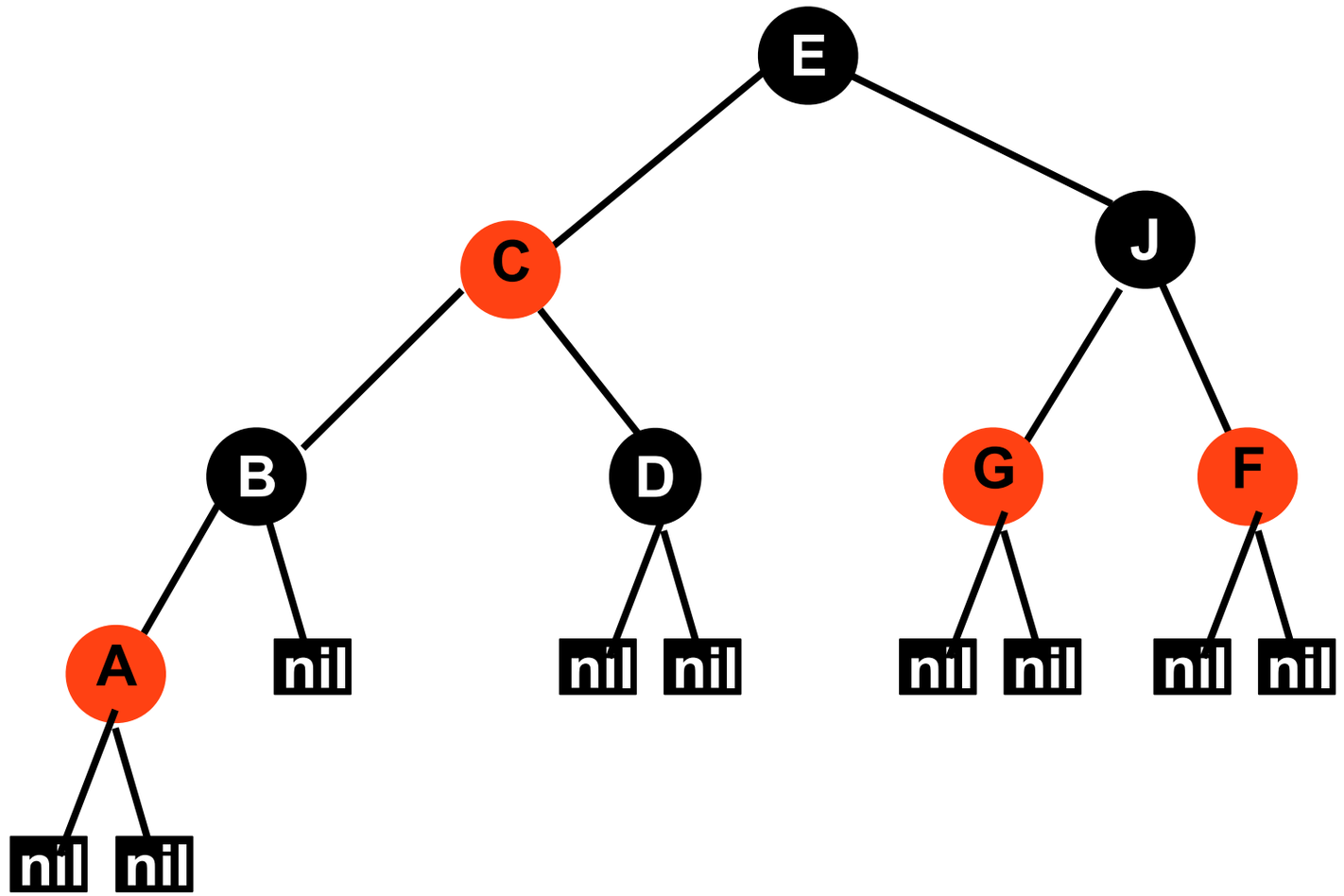
1. **ogni nodo è colorato rosso o nero**
2. **la radice è nera**
3. **le foglie sono nere**

i puntatori ai figli con valore nil sono considerati puntatori a nodi esterni (foglie) fittizi



Alberi rosso-neri

4. **se un nodo è rosso allora suo padre è nero**

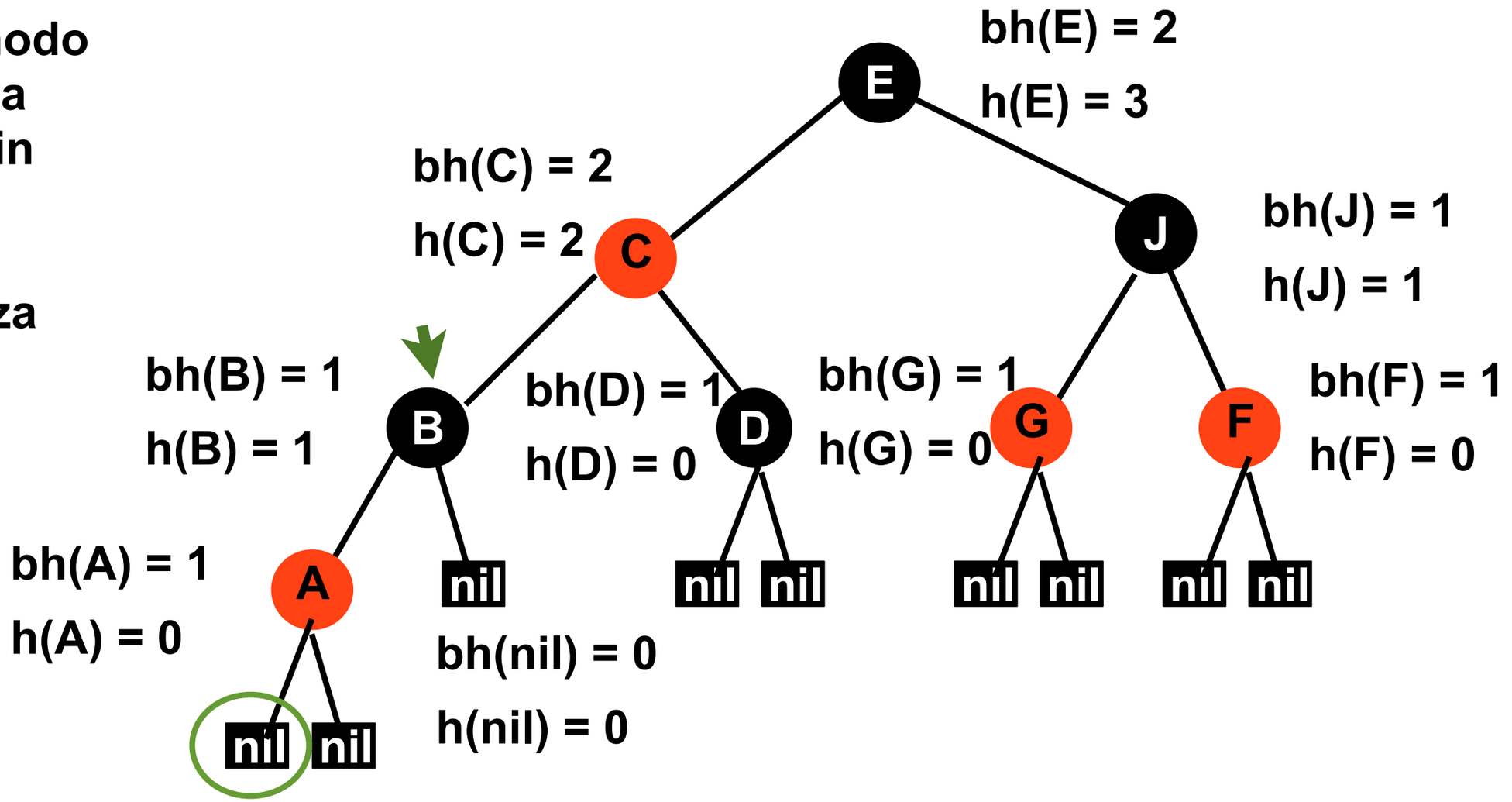


Alberi rosso-neri

5. **per ogni nodo x tutti i cammini da x alle foglie sue discendenti contengono lo stesso numero di nodi neri , compresa la foglia ma escluso il nodo x.**

Chiamiamo bh(x) (black-height) questo numero

L'altezza di un nodo x, h(x), è l'altezza del sottoalbero in esso radicato. L'altezza della radice, è l' altezza di tutto l'albero

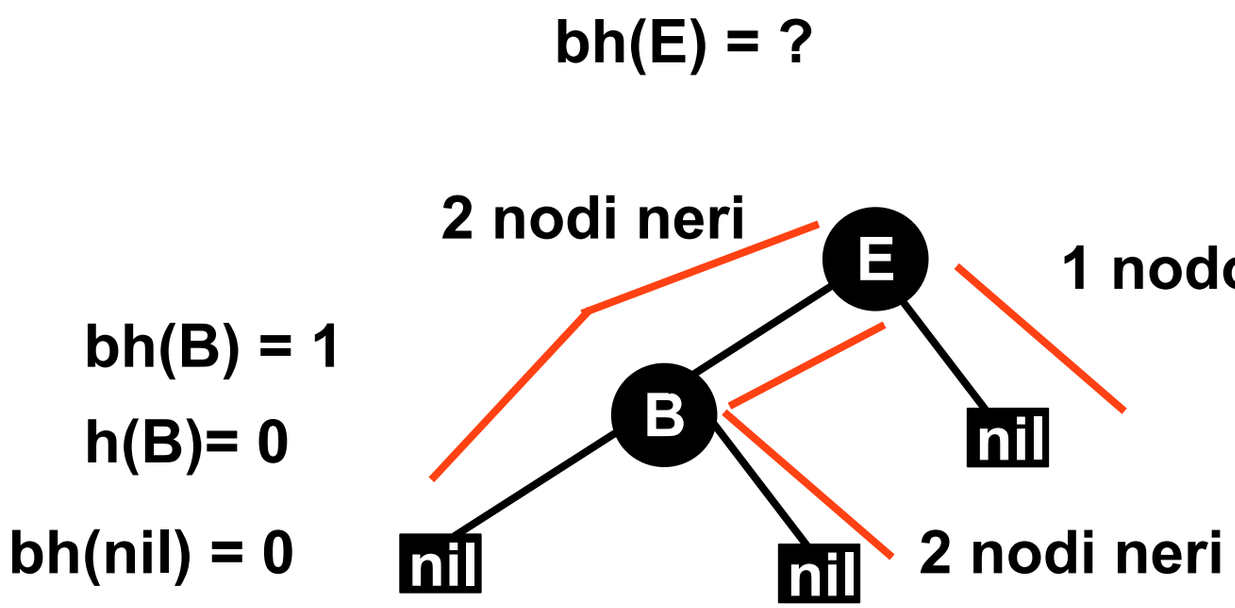


Alberi rosso-neri

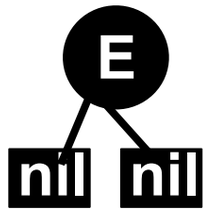
Un albero rosso-nero è un ABR che rispetta queste proprietà:

1. **ogni nodo è colorato rosso o nero**
2. **la radice è nera**
3. **le foglie sono nere**
4. **se un nodo è rosso allora suo padre è nero**
5. **per ogni nodo x tutti i cammini da x alle foglie sue discendenti contengono lo stesso numero di nodi neri, compresa la foglia ma escluso il nodo x.**

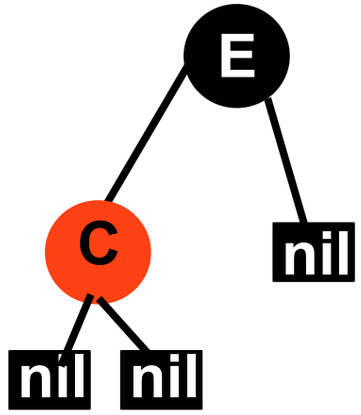
Esempio di ABR non rosso-nero

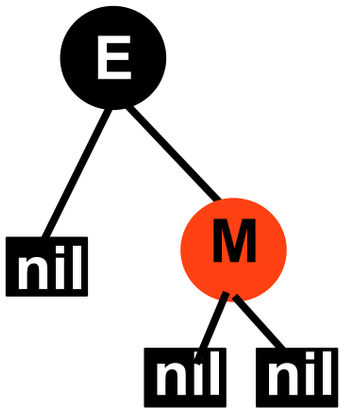


Alberi rosso-neri: esempi

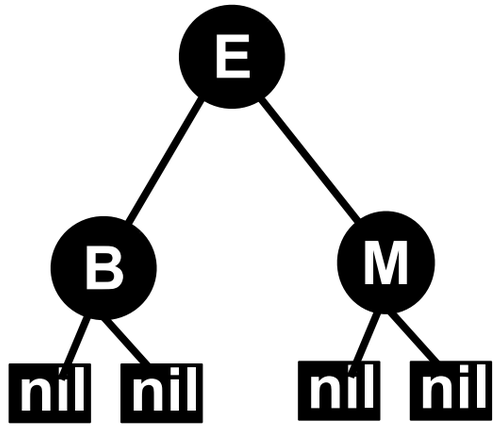


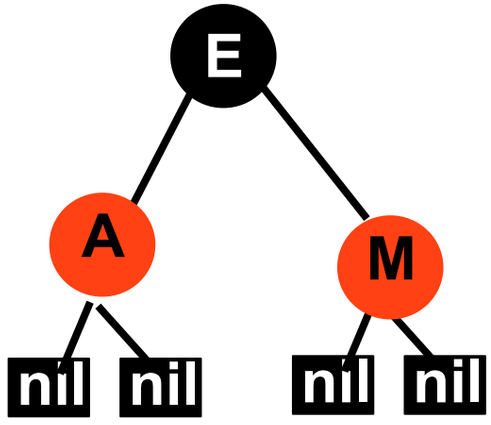
L'unico albero rosso-nero con un solo nodo



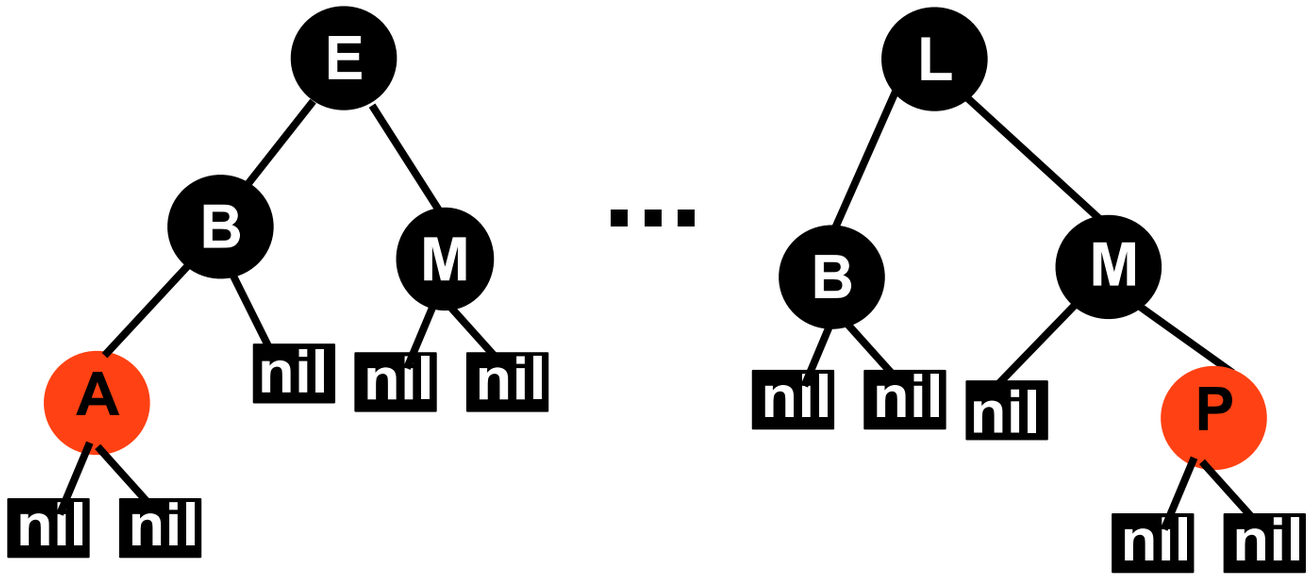


I due alberi rosso-neri con due nodi

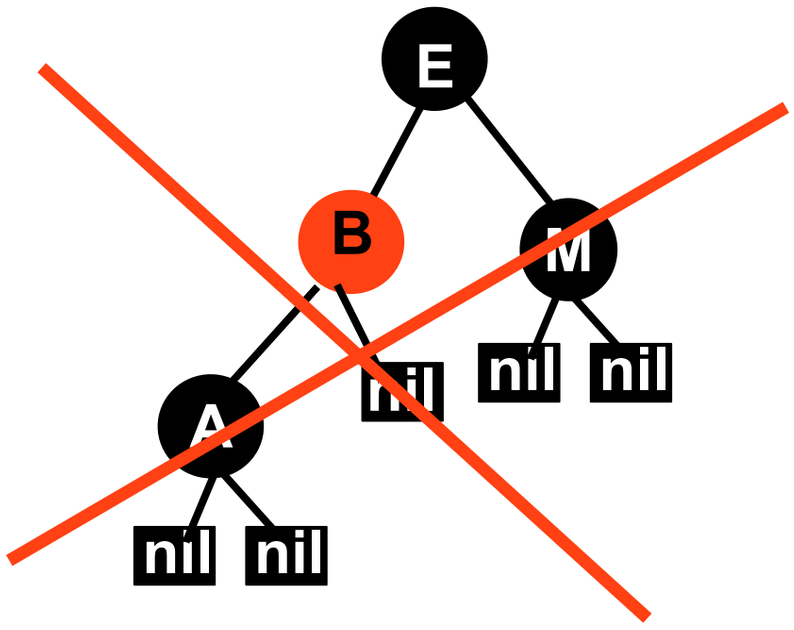




I due alberi rosso-neri con tre nodi

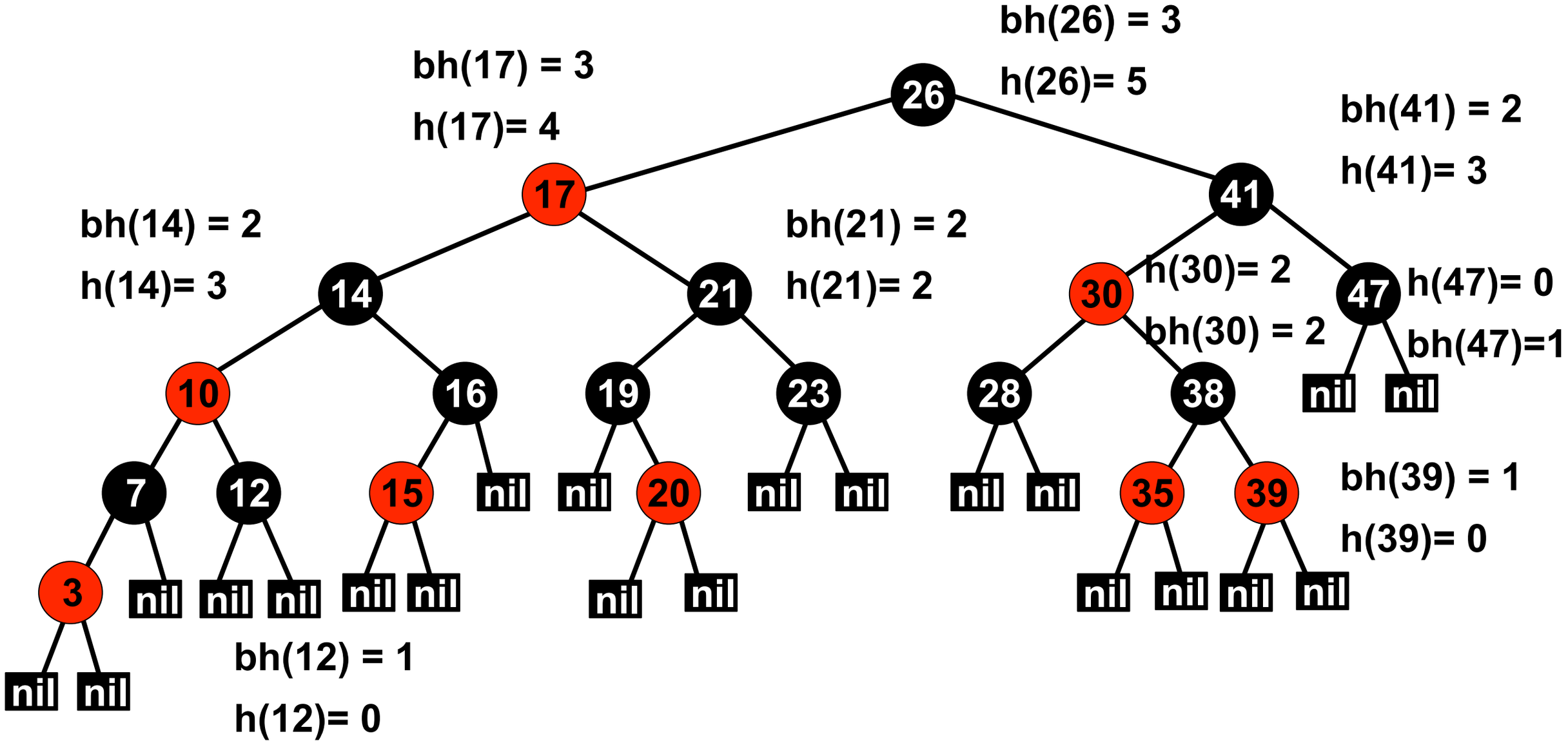


4 alberi rosso-neri con quattro nodi

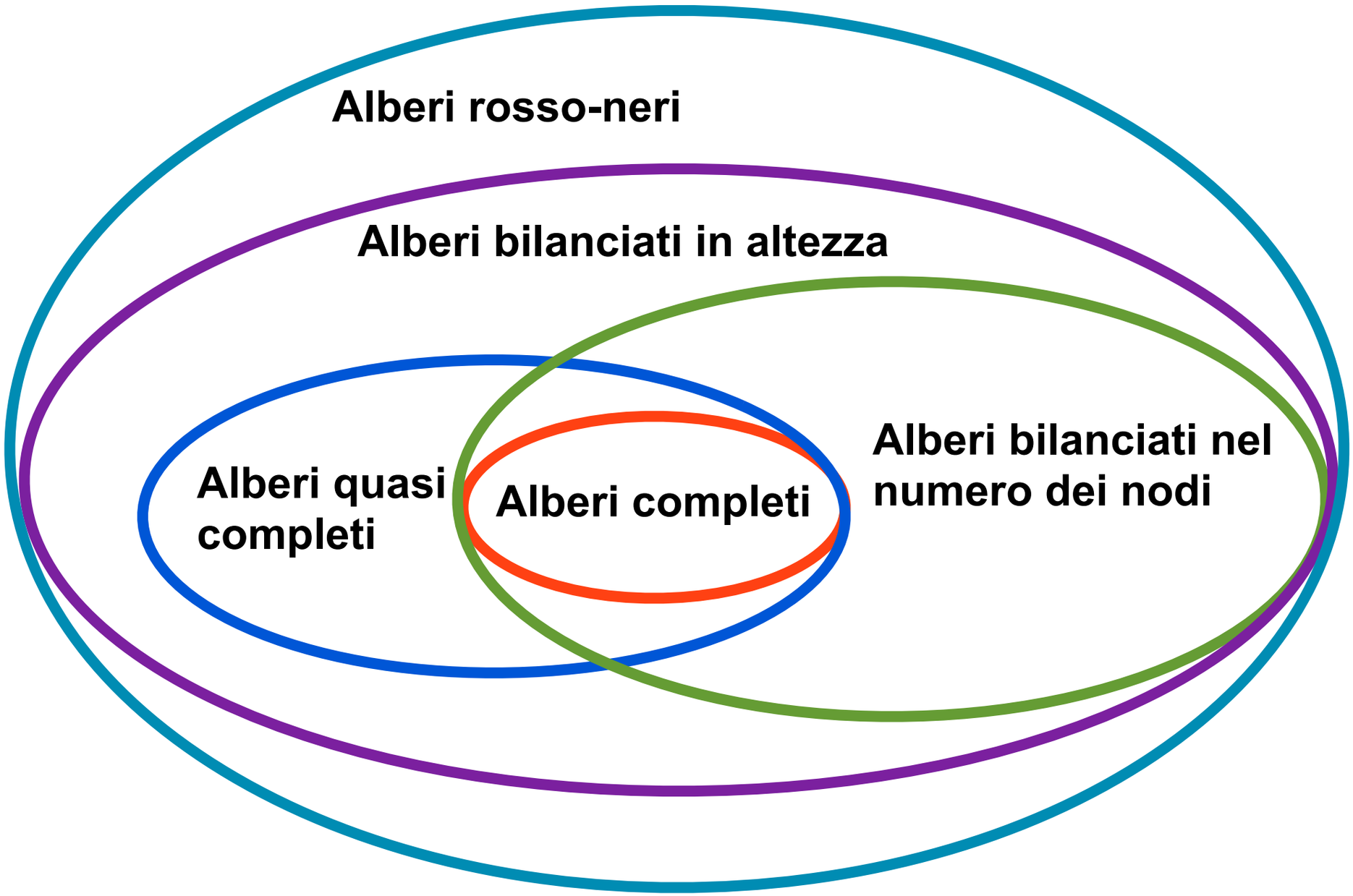


Esempio di albero rosso-nero

Non bilanciato in altezza!

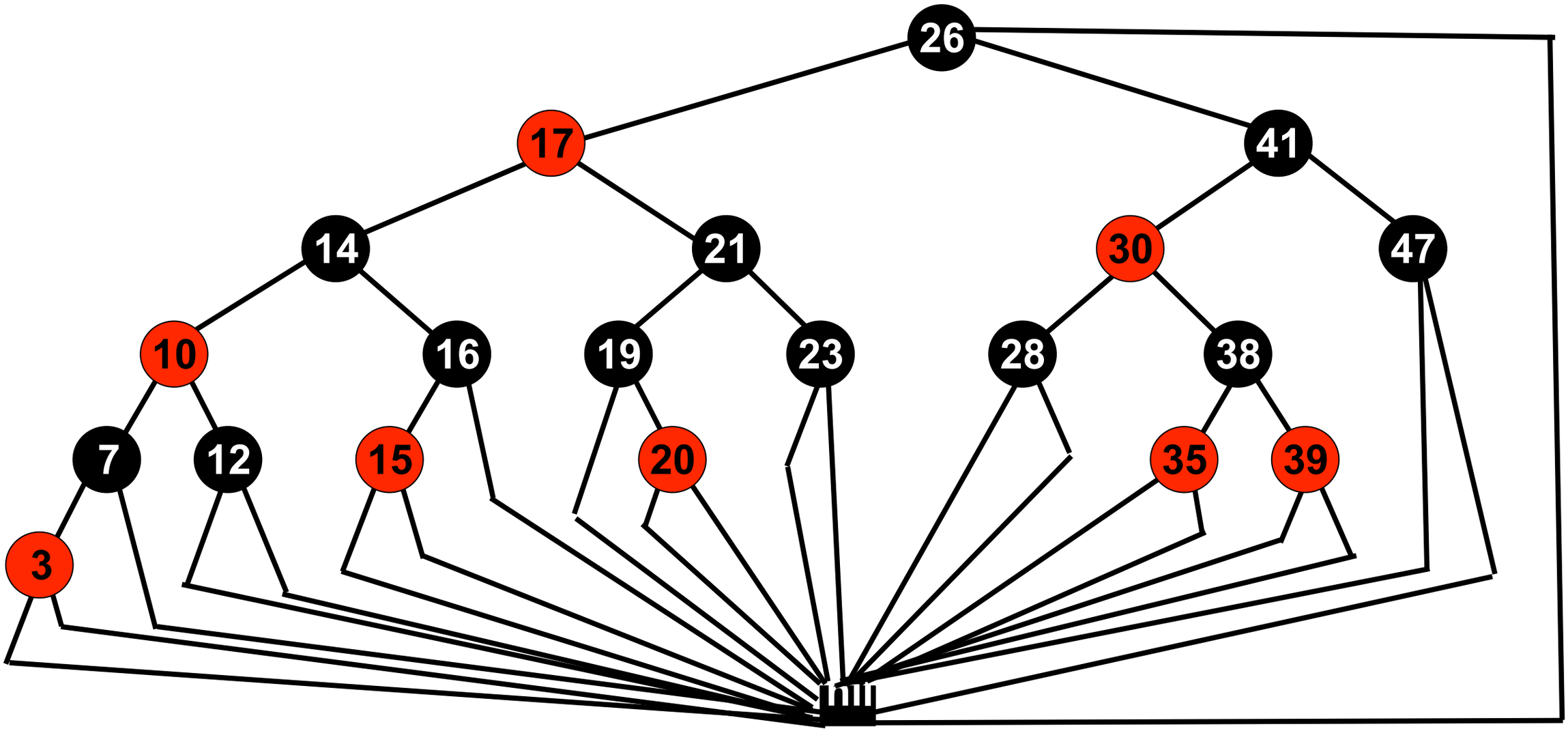


Relazioni tra le classi di alberi



Rappresentazione in memoria: ridurre lo spazio e semplificare il codice

Per semplificare i controlli sul nodo radice e sulle foglie si introduce un'unica sentinella T.nil: un nodo con campo color posto a nero e gli altri (padre, figlio sinistro, figlio destro e chiave) a valori arbitrari.



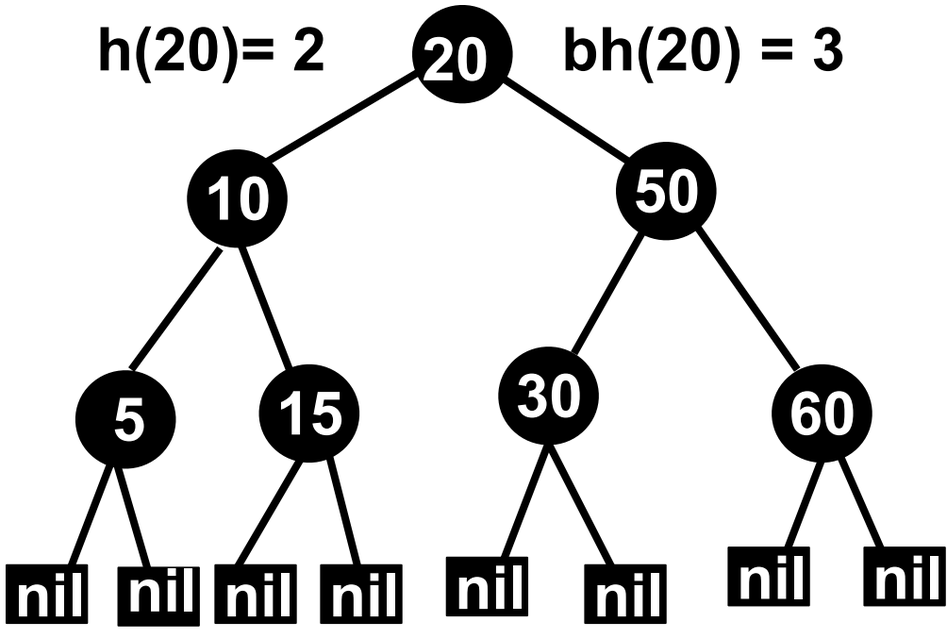
Altezza alberi rosso-neri

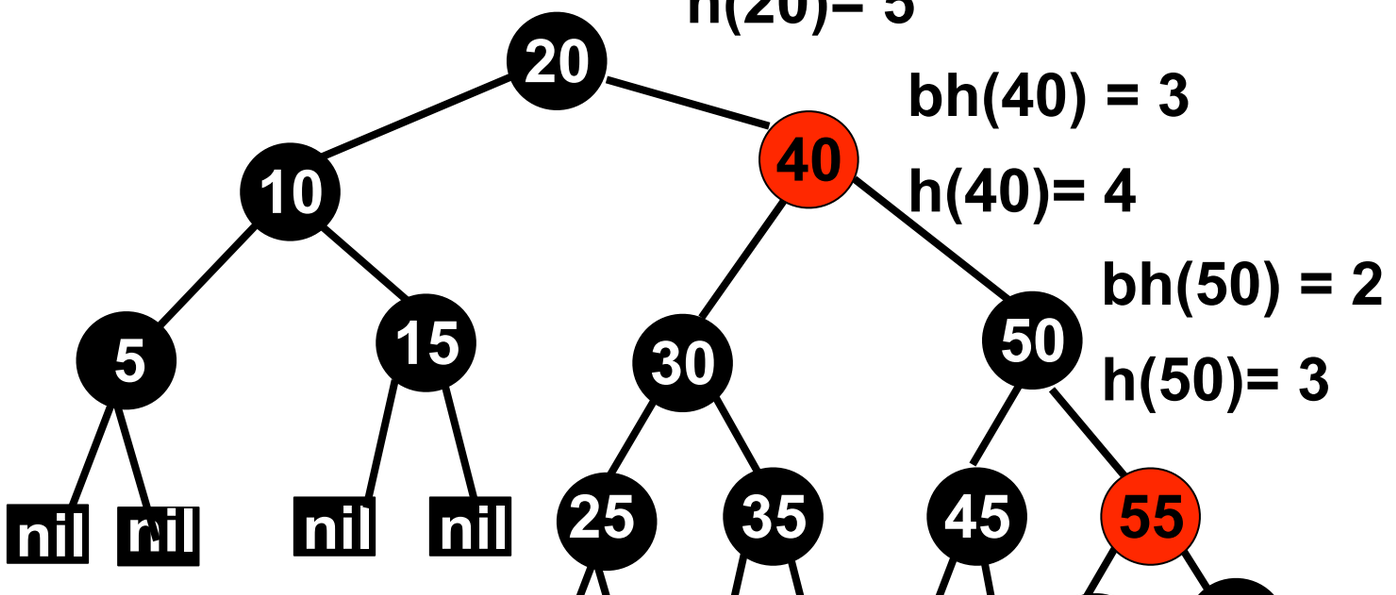
Lemma 1. In un albero rosso-nero per ogni nodo x

$$bh(x) \geq h(x)/2$$

Prova: Ogni nodo rosso ha padre nero, quindi al più la metà dei nodi su un cammino può essere rosso.

Quindi $bh(x) \geq h(x)/2$, per ogni nodo x.





Per esempio se partiamo da un albero tutto nero, che deve essere perfetto,

alternando rossi e neri si riesce al più a raddoppiare l'altezza, $h(x) \leq 2bh(x)$

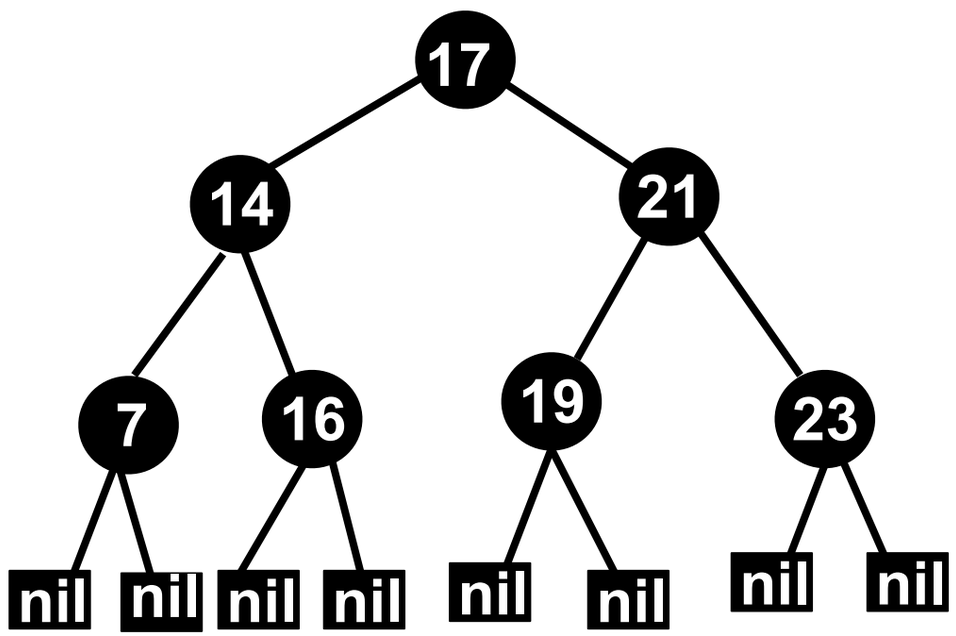
Altezza alberi rosso-neri

Lemma 2. In un albero rosso-nero, per ogni nodo x

$$Nodi(x) \geq 2^{bh(x)}-1 ,$$

dove Nodi(x) = numero di nodi interni (cioè i nodi con chiave) nel sottoalbero radicato in x

Prova. Basta osservare che $2^{bh(x)}-1$ è il numero dei nodi interni di un albero radicato in x tutto nero di altezza h = bh(x)-1, perchè il numero dei nodi di un albero perfetto di altezza h è $2^{h+1}-1$, quindi $Nodi(x) = 2^{bh(x)-1+1}-1 = 2^{bh(x)}-1$. Ogni altro albero con la stessa altezza nera ha più nodi interni, perchè mantenendo la stessa di altezza nera posso aggiungere solo nodi rossi.



Esempio con bh(17) = 3 e h= 2

Un albero tutto nero deve essere perfetto

Gli alberi rosso-neri hanno altezza logaritmica nel numero dei nodi

Abbiamo dimostrato:

Lemma 1 **l'altezza nera della radice, x, soddisfa**

$$bh(x) \geq h/2$$

Lemma 2 **il numero di nodi interni dell'albero soddisfa**

$$Nodi(x) \geq 2^{bh(x)}-1 ,$$

Quindi $n = Nodi(x) \geq 2^{bh(x)}-1$

$$\geq 2^{h/2}-1,$$

da cui $n+1 \geq 2^{h/2}$ **prendendo il logaritmo**

$lg(n+1) \geq h/2$ **e quindi**

$2lg(n+1) \geq h$

Possiamo concludere:

Lemma 3. **In un albero rosso-nero con n nodi interni e altezza h**

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

Altezza nera e numero di nodi interni, la prova induttiva

Lemma 2. In un albero rosso-nero, per ogni nodo x

$$Nodi(x) \geq 2^{bh(x)}-1 ,$$

dove Nodi(x) = numero di nodi interni nel sottoalbero radicato in x

per induzione su h(x).

Base: Se x = nil h(x) = -1 e bh(x) = 0 e $Nodi(x)=0 \geq 2^{bh(x)} - 1 = 2^0 - 1 = 0$.

Passo induttivo:

Sia vero per alberi di altezza < h. Prendiamo un albero di altezza h, con radice x. Chiamiamo i figli della radice y e z.

Sappiamo che Nodi(x) = Nodi(y) + Nodi(z) +1.

Per ipotesi induttiva, visto che i sottoalberi hanno altezza minore di h, si ottiene che

$Nodi(y) \geq 2^{bh(y)}-1$ e

$Nodi(z) \geq 2^{bh(z)}-1$. Quindi

$Nodi(x) \geq 2^{bh(y)}-1+ 2^{bh(z)}-1 + 1 = 2^{bh(y)}+ 2^{bh(z)}-1$

Altezza alberi rosso-neri

Lemma 2. In un albero rosso-nero, per ogni nodo x

$$Nodi(x) \geq 2^{bh(x)}-1 ,$$

dove Nodi(x) = numero di nodi interni nel sottoalbero radicato in x

Prova.

Per induzione su h(x).

Passo induttivo (continua):

Abbiamo che

$Nodi(x) \geq 2^{bh(y)}+ 2^{bh(z)}-1$

Se si dimostra che bh(y), bh(z) ≥ bh(x)-1, si ottiene il risultato voluto:

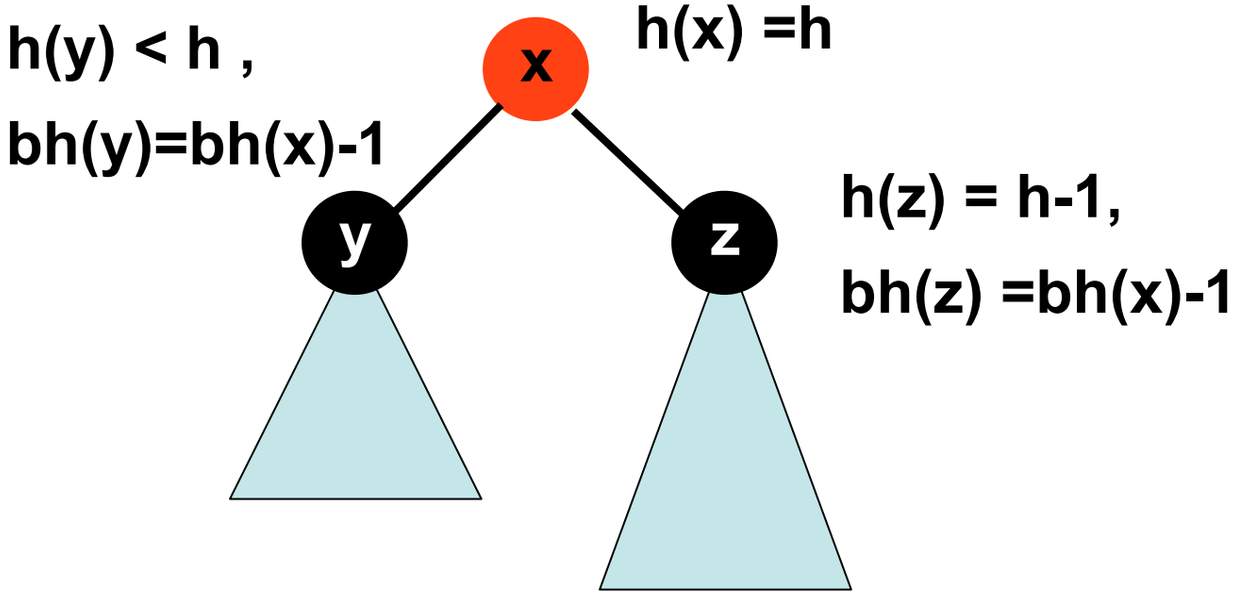
$$Nodi(x) \geq 2^{bh(y)}+ 2^{bh(z\,)}-1$$

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

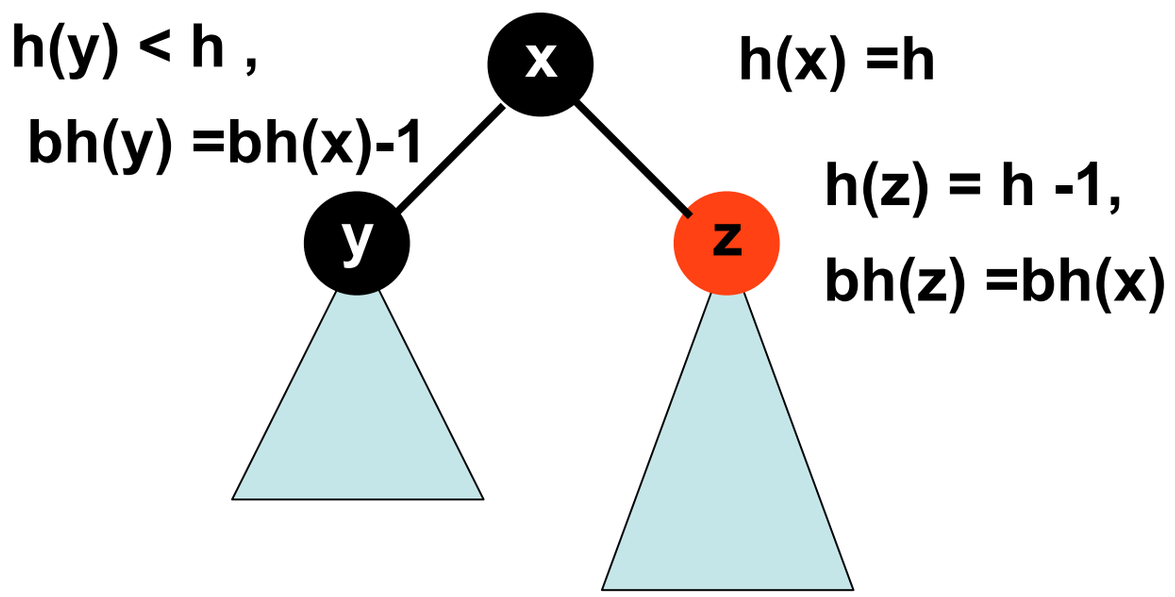
Altezza nera di un figlio è almeno l'altezza del padre meno 1

Preso un albero di altezza h, con radice x e figli y e z, vogliamo dimostrare che bh(y), bh(z) ≥ bh(x)-1

Caso 1 : x è rosso



Caso 2 : x è nero



Nota: uno almeno dei due sottoalberi deve avere altezza h-1

Operazioni su ABR in O(lg n)

In un ABR con n nodi e di altezza h le operazioni di

- **ricerca**
- **inserimento**
- **cancellazione**

costano O(h), e lg n ≤ h < n

in un albero rosso-nero dimostreremo che costano O(lg n), perchè l'altezza è O(lg n) e il ribilanciamento dell'albero può essere fatto in O(lg n)

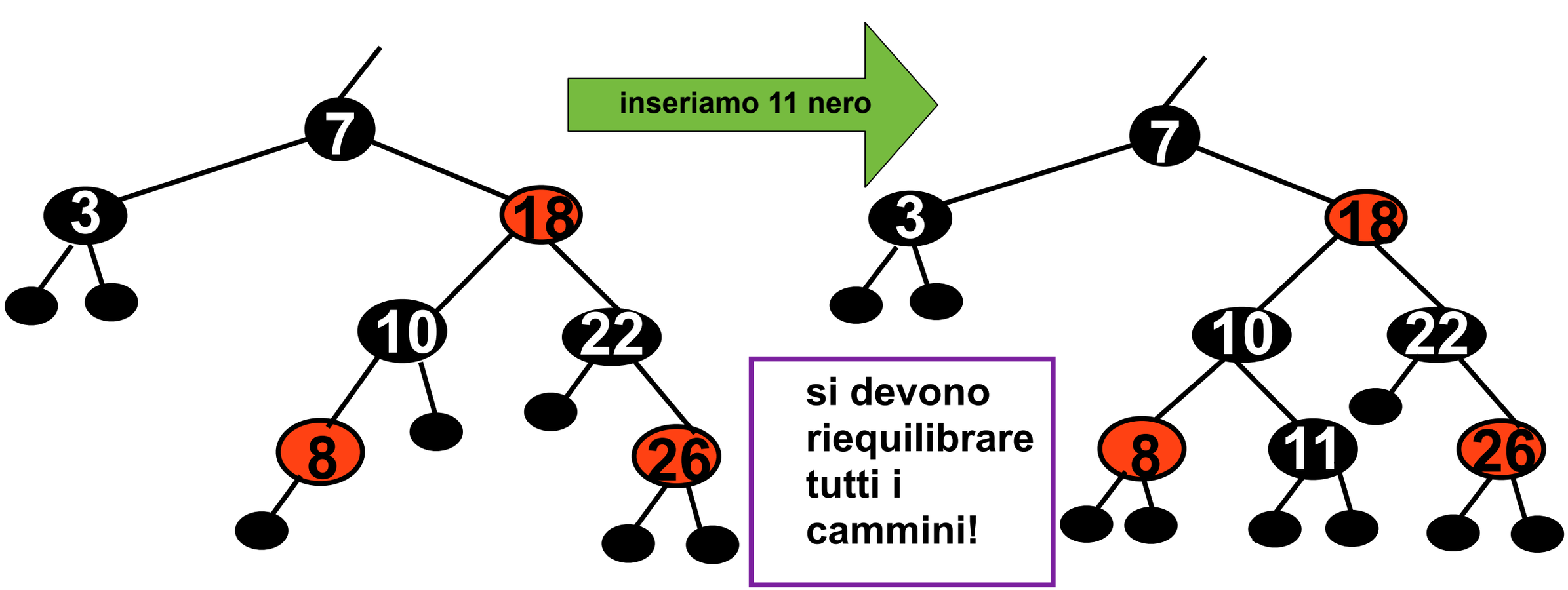
Inserimento in un albero rosso-nero

Per inserire un nuovo nodo si usa l'inserimento visto per gli ABR, fatte le dovute modifiche dovute al nodo sentinella, cui si fa seguire un'operazione di ripristino di proprietà della colorazione eventualmente violate.

Per prima cosa bisogna decidere quale colore attribuire al nuovo nodo.

- **se nero: si potrebbe violare la regola:**

5. per ogni nodo x tutti i cammini da x alle foglie sue discendenti contengono lo stesso numero di nodi neri , compresa la foglia ma escluso il nodo x



Inserimento in un albero rosso-nero

Per inserire un nuovo nodo si usa l'inserimento visto per gli ABR, fatte le dovute modifiche dovute al nodo sentinella, cui si fa seguire un'operazione di ripristino di proprietà della colorazione eventualmente violate.

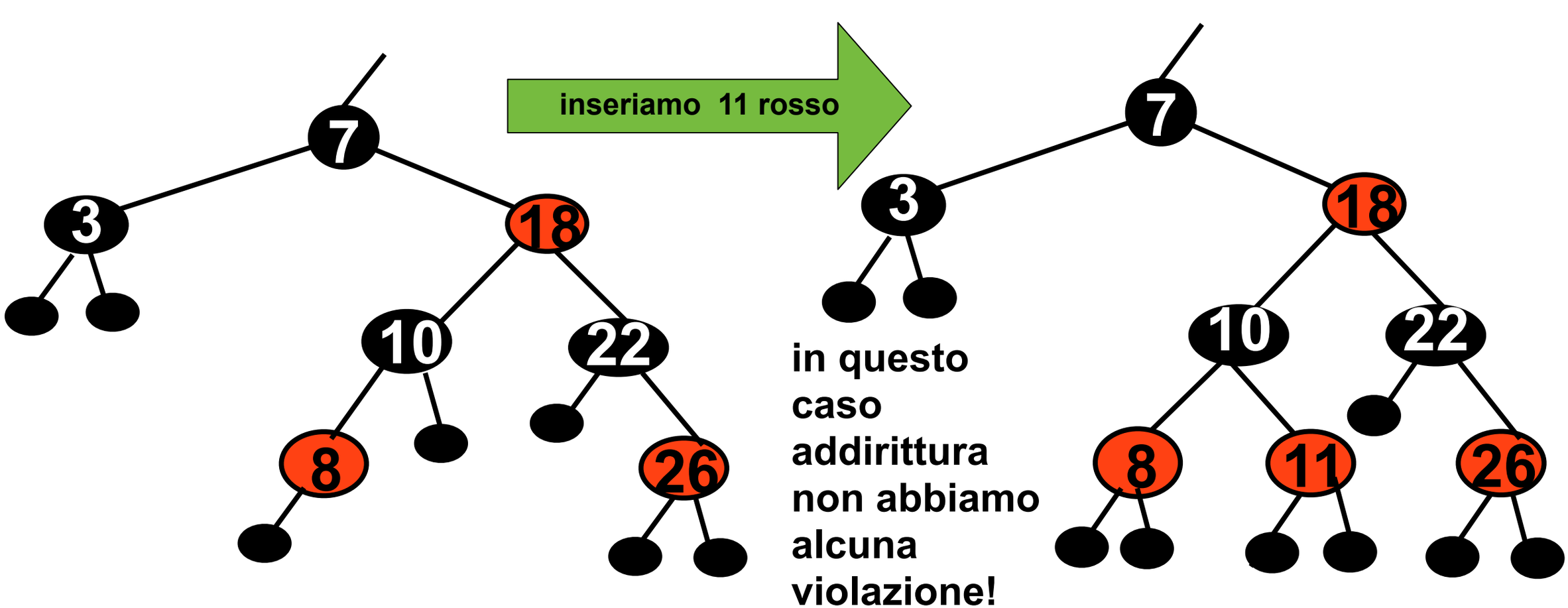
Per prima cosa bisogna decidere quale colore attribuire al nuovo nodo.

- **se rosso: si potrebbero violare le regole:**

Meglio rosso!

2. la radice è nera

4. se un nodo è rosso allora suo padre è nero

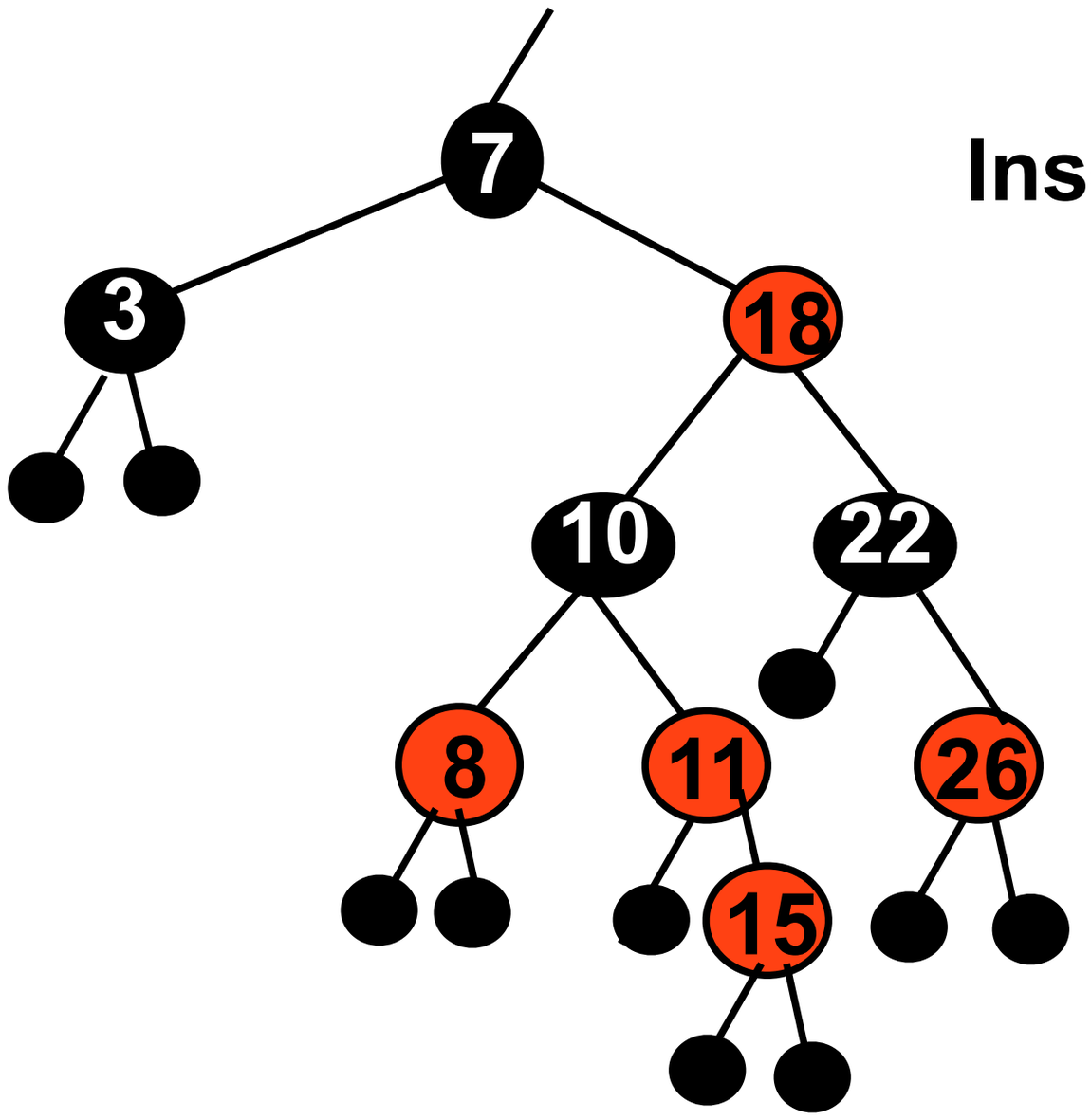


in questo caso addirittura non abbiamo alcuna violazione!

Esempio inserimento con violazione della proprietà 2

Inseriamo 15 rosso in un albero vuoto

Esempio inserimento con violazione della proprietà 4



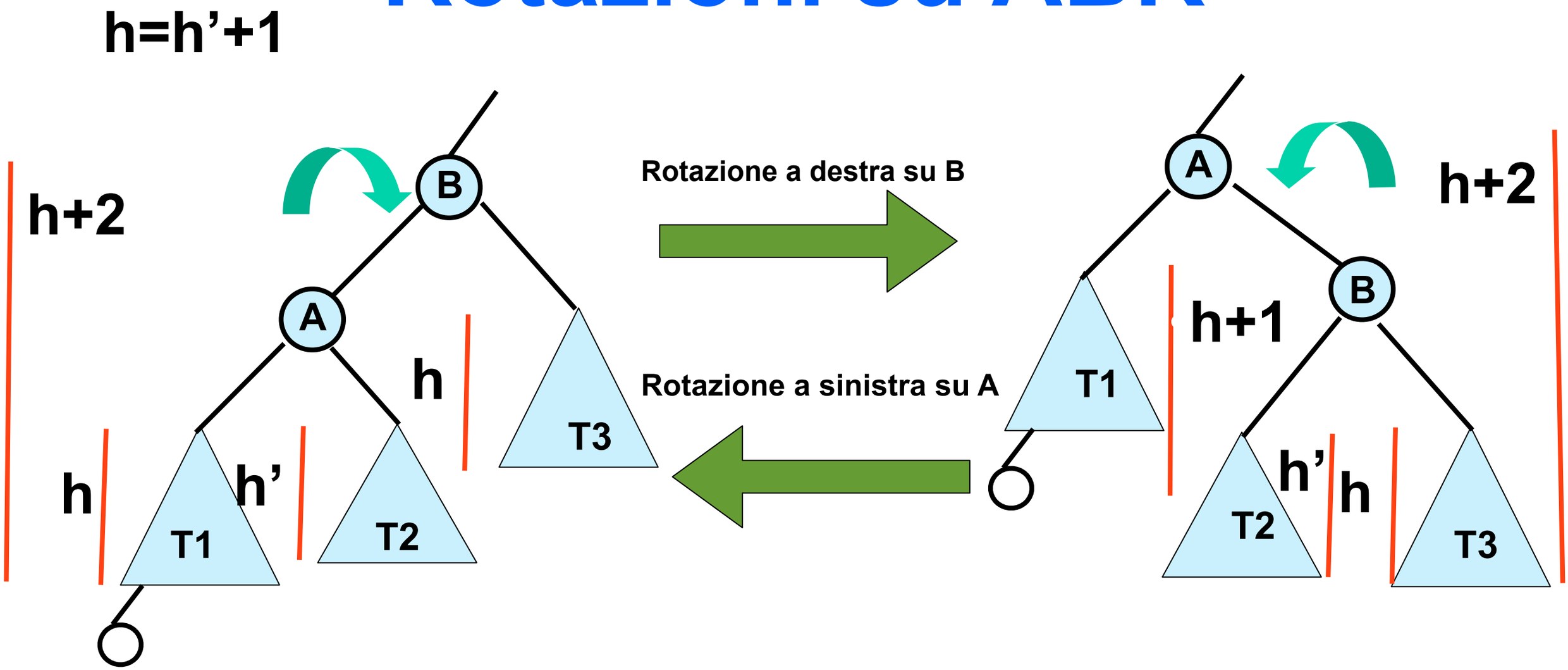
Inseriamo 15 rosso

Operazioni di ribilanciamento

Descriviamo nel seguito le operazioni sugli alberi rosso-neri che ci consentiranno di ripristinare la proprietà 3 eventualmente violata in un inserimento.

Le prime operazioni sono le rotazioni semplici

Rotazioni su ABR

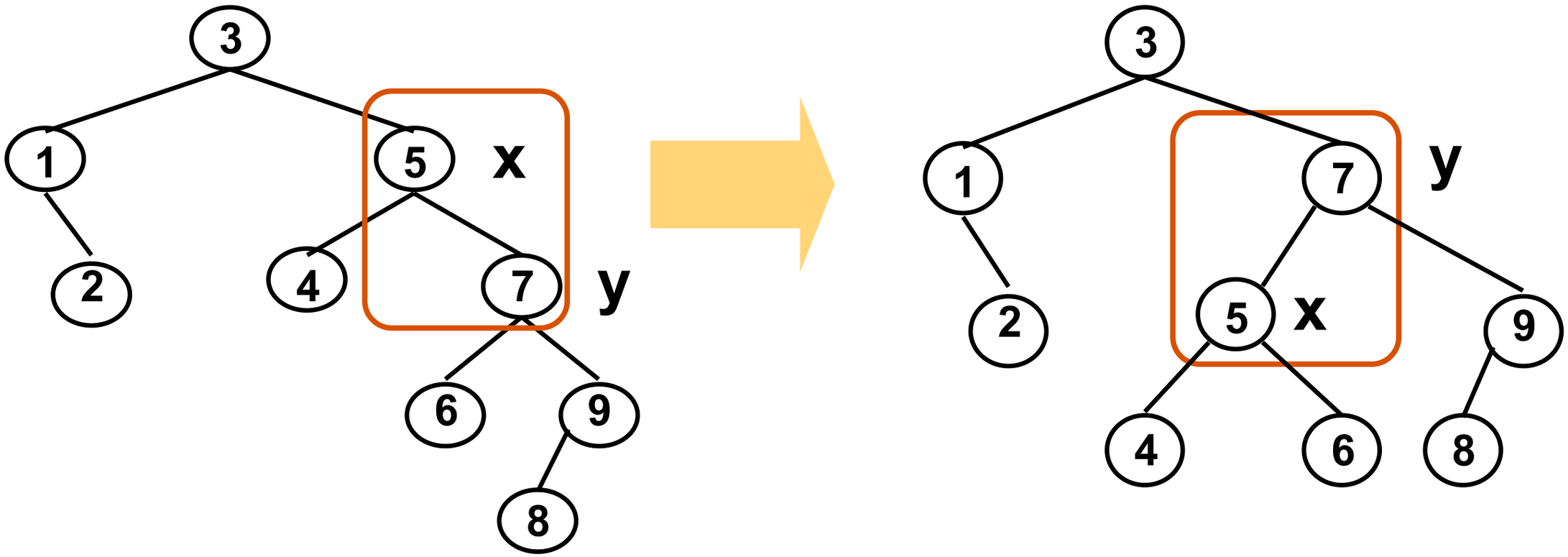


Dopo la rotazione l'albero è ancora un ABR, perchè se a è un nodo in T1, b in T2, c in T3 allora vale

a < A < b < B < c.

Richiede tempo O(1)

Esempio di rotazione a sinistra su 5



Pseudocodice rotazione

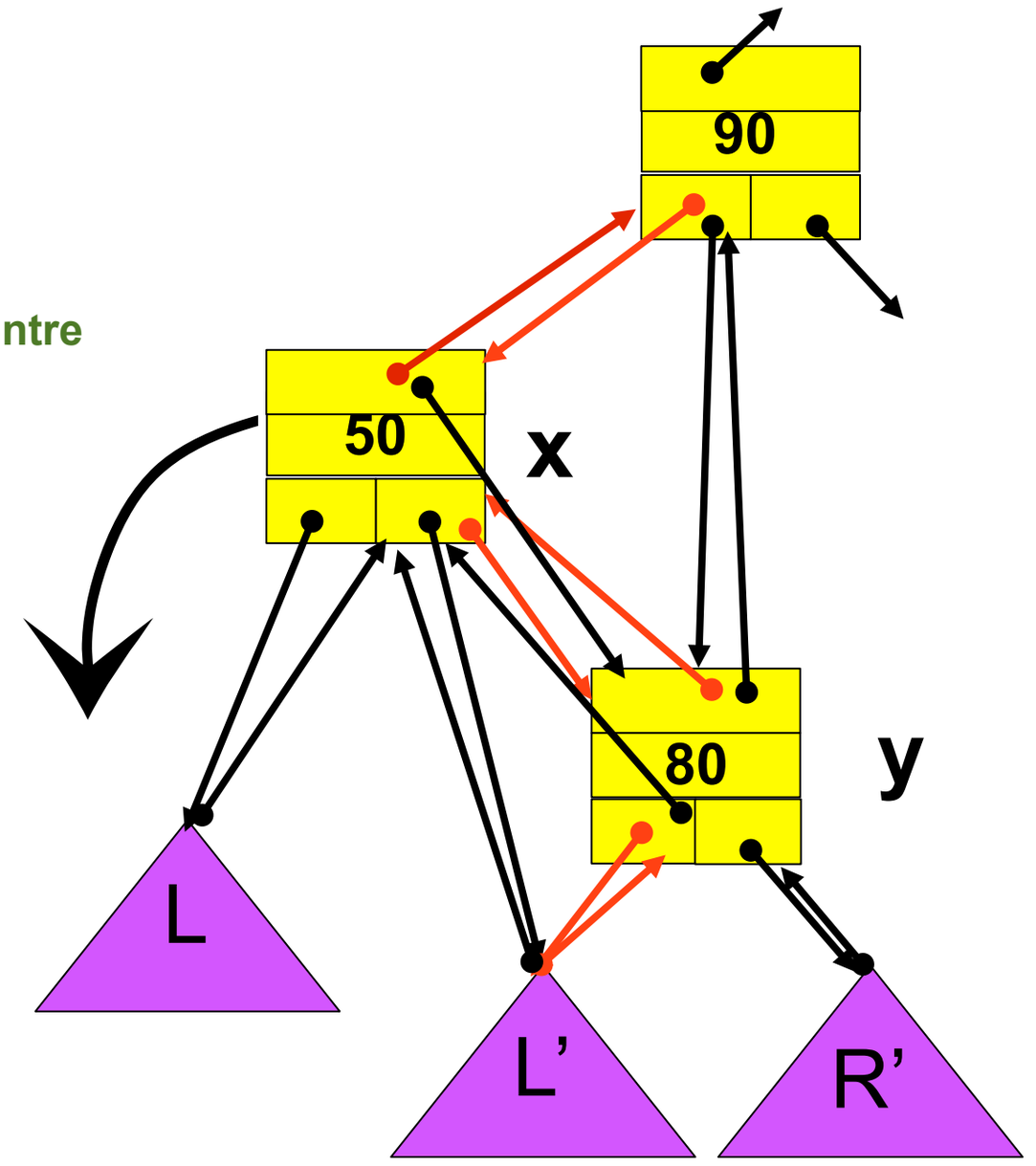
Left-Rot(T,x)

input: T è un puntatore a un albero e x a un suo nodo

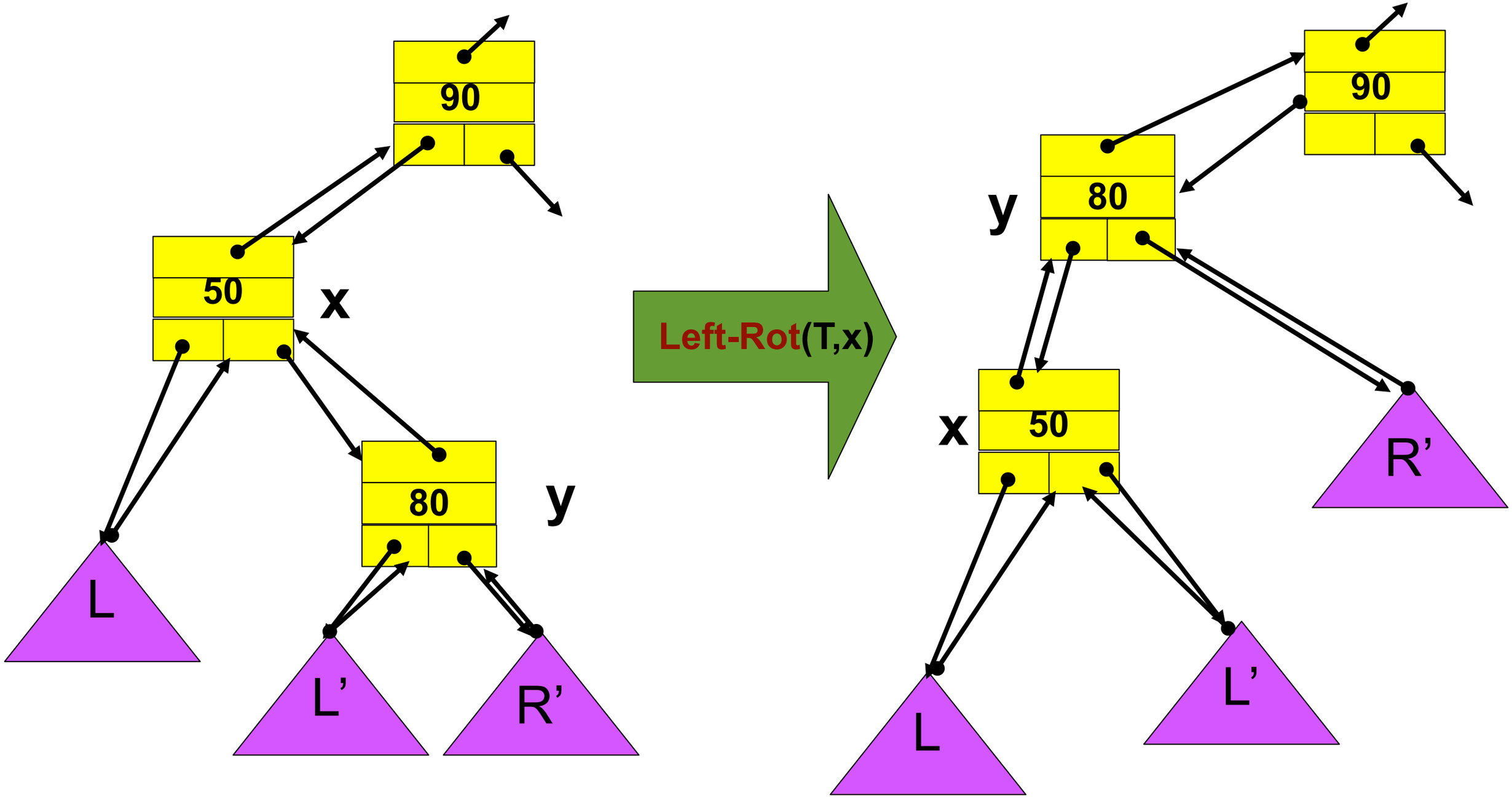
Prec: T è un ABR rosso-nero e y=right[x]≠ nil

postc: il figlio destro, y, di x prende il suo posto come figlio, conserva il suo figlio destro, e prende x come figlio sinistro mentre il figlio sinistro di y diventa figlio destro di x.

```
y = x.right              //  y è il figlio destro di x
x.right = y.left
// il sottoalbero sinistro di y diventa il destro di x
if y.left ≠ NIL  then y.left.p = x
y.p = x.p      // il padre di x diventa padre di y
if x.p == NIL then
      T.root = y
       else if x == x.p.left
                    then  x.p.left = y
                    else x.p.right = y
y.left = x      // x diventa il figlio sinistro di y
   x.p = y
```



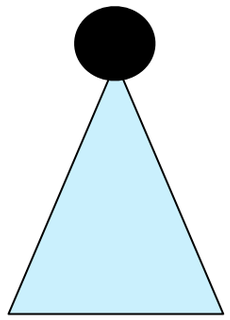
Pseudocodice rotazione: fine



$T(n) = \Theta(1)$

Convenzione

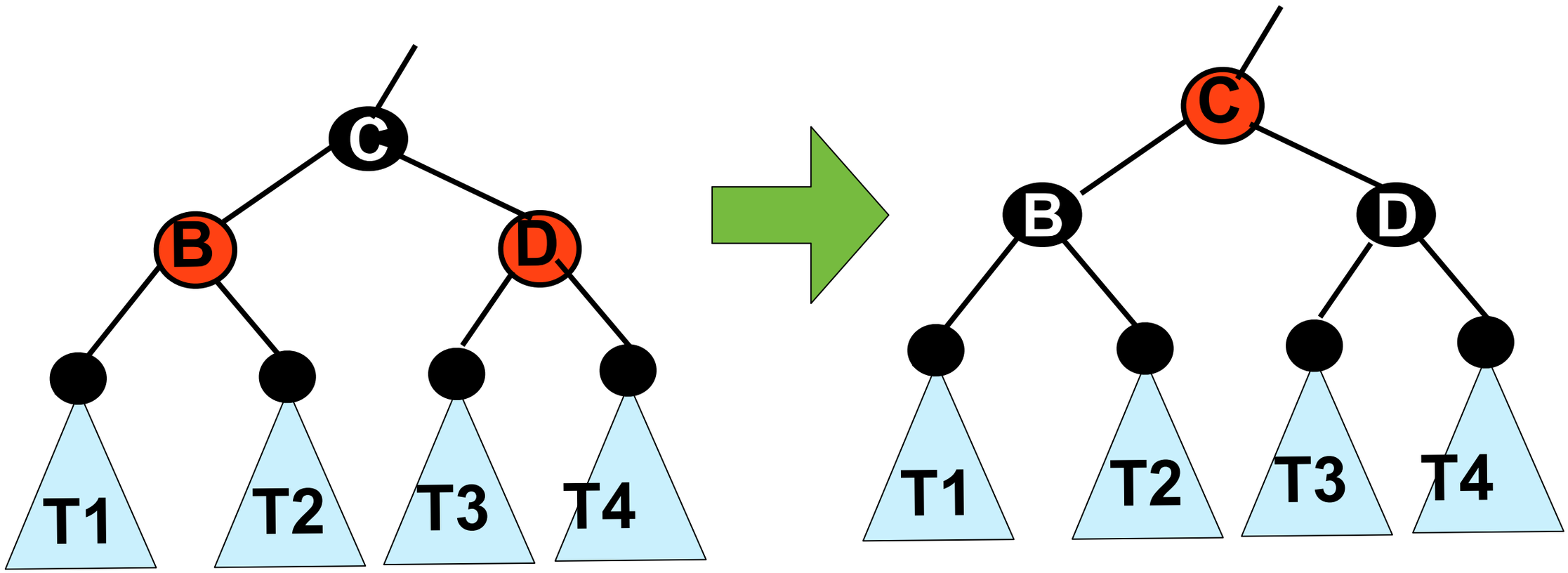
D’ora in poi



rappresenta un albero rosso-nero con radice nera.

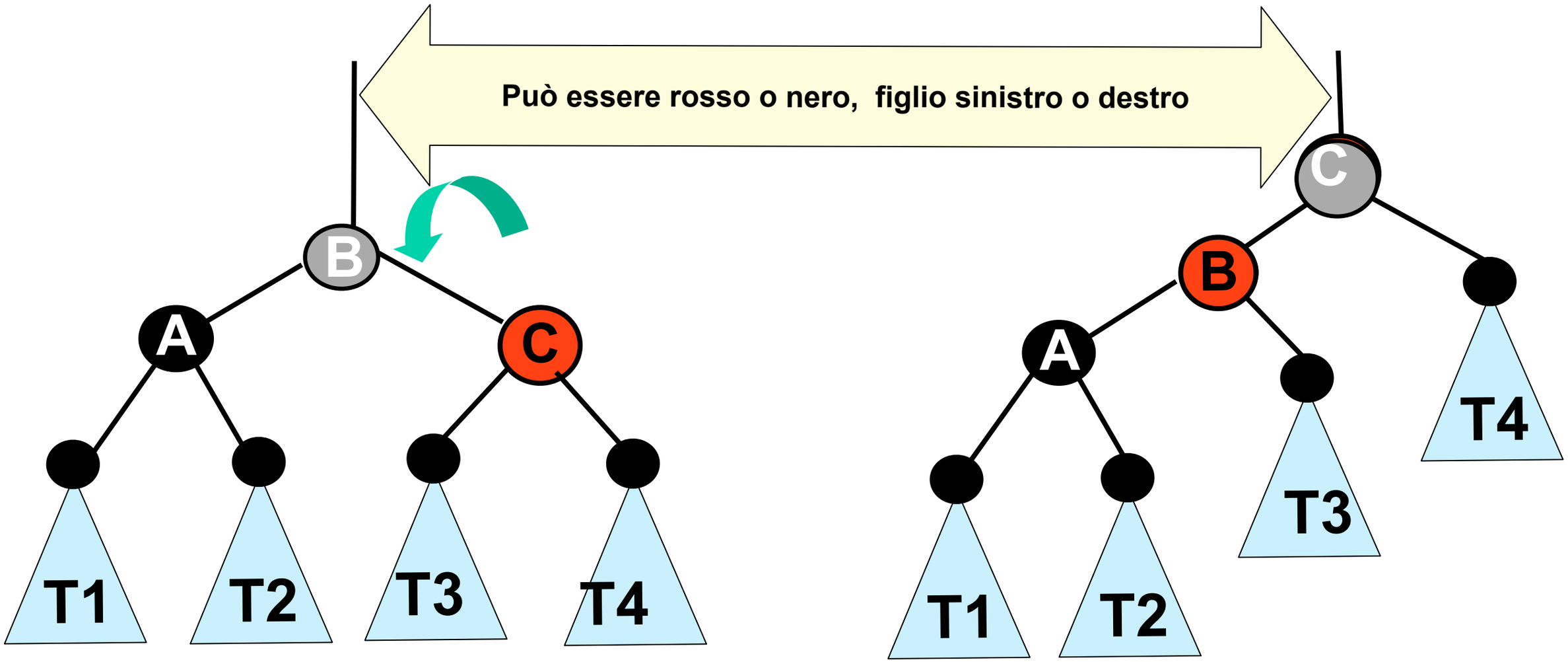
Descriviamo nel seguito le altre operazioni sugli alberi rosso-neri che ci consentiranno di ripristinare le proprietà eventualmente violate a seguito di inserimenti o di cancellazioni, (ma per le cancellazioni sarà necessario introdurre un’altra operazione)

Scambio di colori



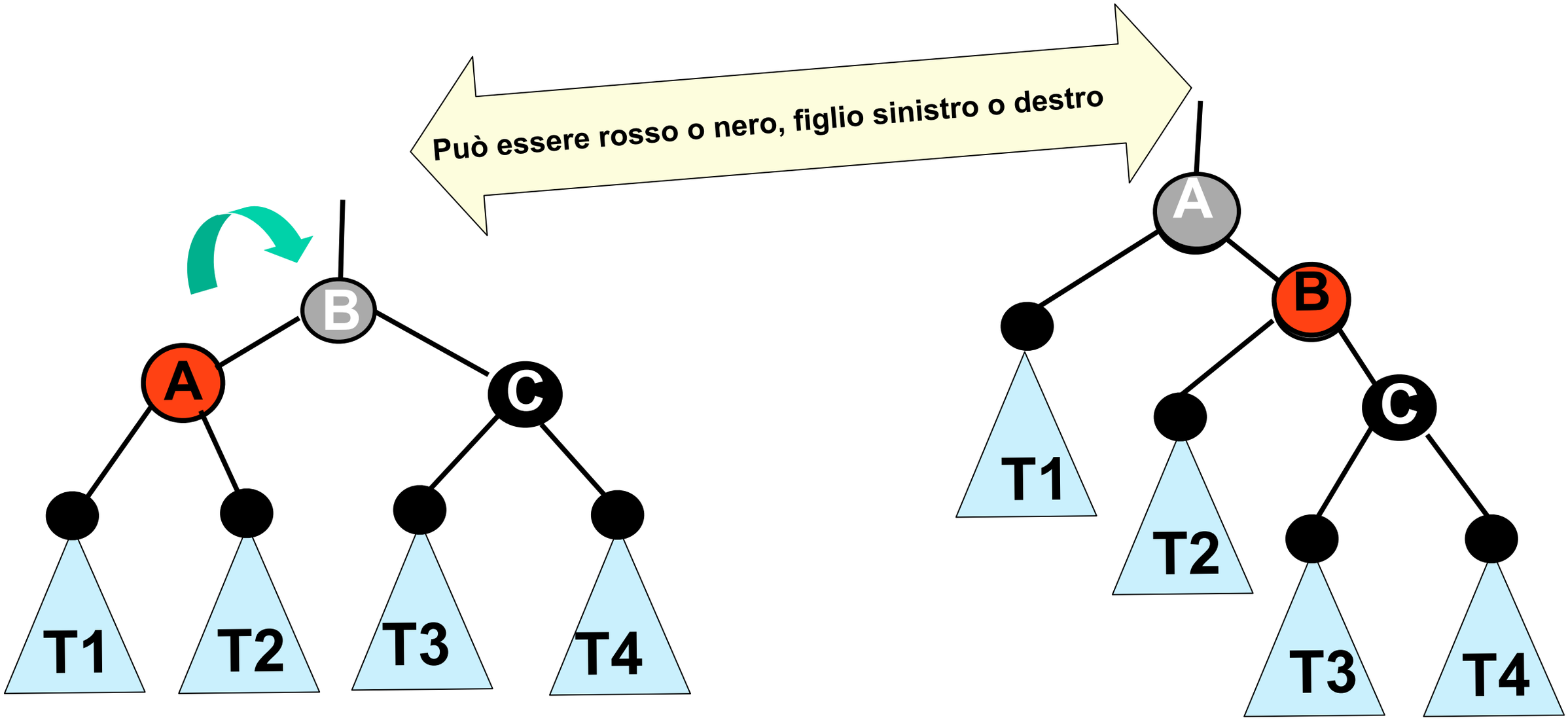
Il padre e i figli scambiano il colore. Nell'inserimento si applicherà quando il padre è nero e i figli rossi.
Il numero di nodi neri dalla radice di Ti, per i=1,2,3,4 fino al padre di C (escluso) è il medesimo

RB-rotazione a sinistra su B



Si esegue una rotazione a sinistra su C e i nodi B e C si scambiano di colore.
Il numero di nodi neri dalla radice di Ti, per i=1,2,3,4 fino alla radice B a sinistra, C a destra è il medesimo

RB-rotazione a destra su B



Si esegue una rotazione a destra su B e i nodi A e B si scambiano di colore.
Il numero di nodi neri dalla radice di Ti, per i=1,2,3,4 fino alla radice B a sinistra, A a destra è il medesimo

Proprietà delle operazioni

- preservano le altezze nere e
- non introducono due nodi rossi uno figlio dell'altro
- si eseguono in Θ(1)

Inserimento

- **Ora vedremo come queste operazioni si utilizzano per ripristinare in un albero rosso-nero le proprietà:**

2. la radice è nera

4. se un nodo è rosso allora suo padre è nero

I casi

Poichè dobbiamo rimediare, risalendo verso la radice, a una violazione della proprietà 4 senza modificare l'altezza nera, dovremo agire sullo "zio", cioè il fratello del padre rosso del (nuovo) nodo (inserito) rosso.

I casi sono

Caso 1: zio rosso

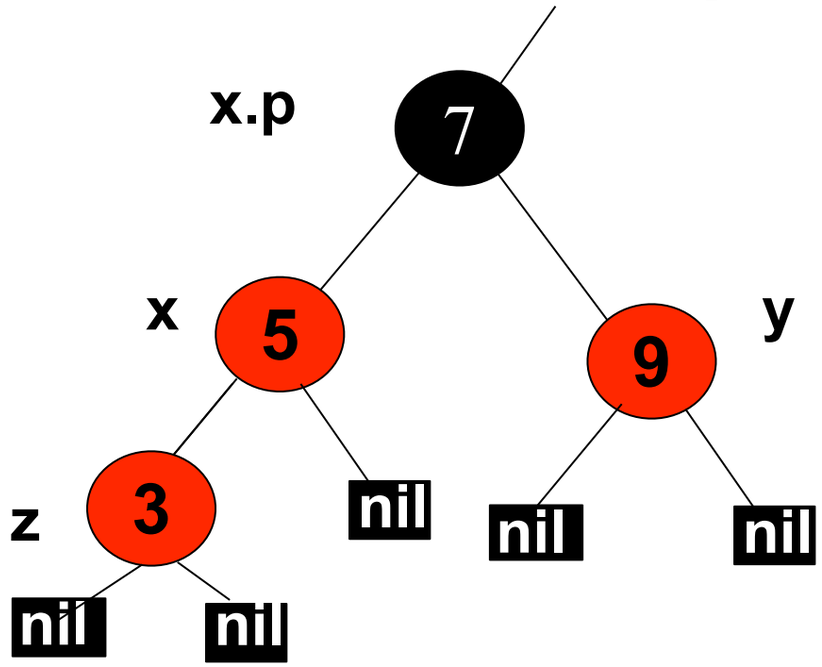
Caso 2: zio nero e nodo rosso figlio destro (sinistro) di un padre rosso figlio sinistro (destro)

Caso 3: zio nero e nodo rosso figlio sinistro (destro) di un padre rosso figlio sinistro (destro).

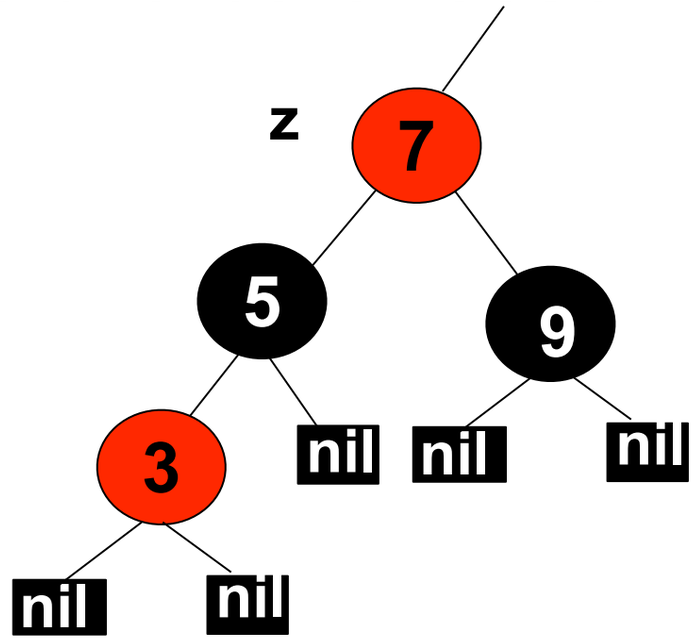
Caso 1: padre e zio rossi

inseriamo z come figlio sinistro di un figlio sinistro

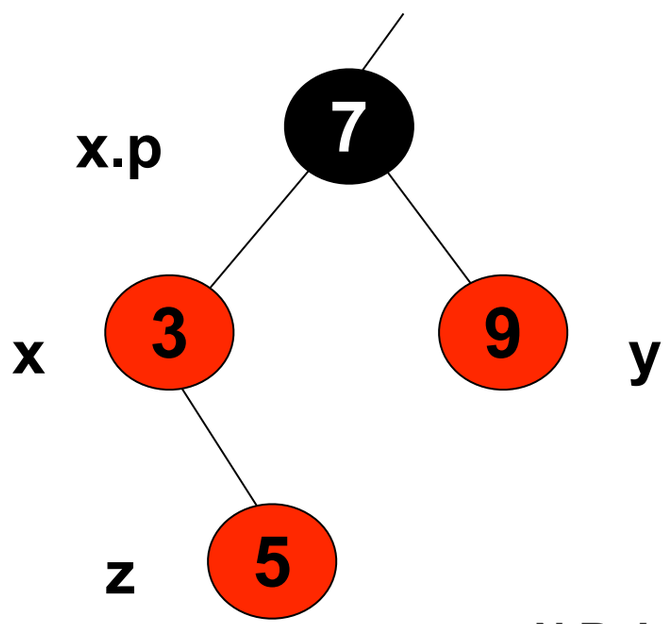
lo zio, y, è rosso, x.p è nero per definizione di albero rosso-nero



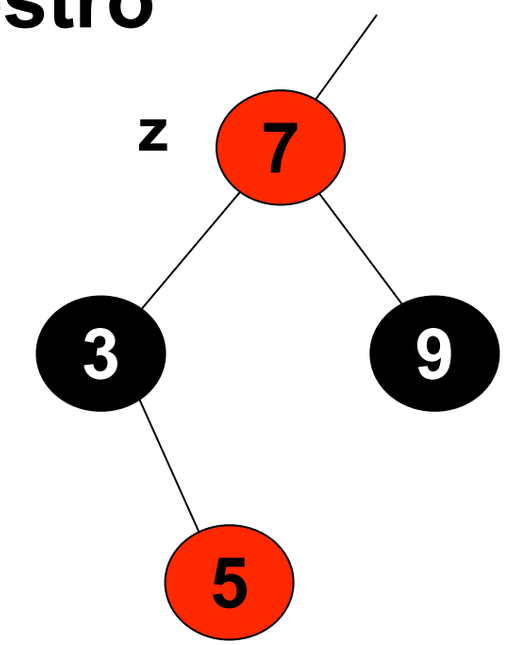
scambio di colori



Il caso simmetrico: il nuovo nodo è un figlio destro

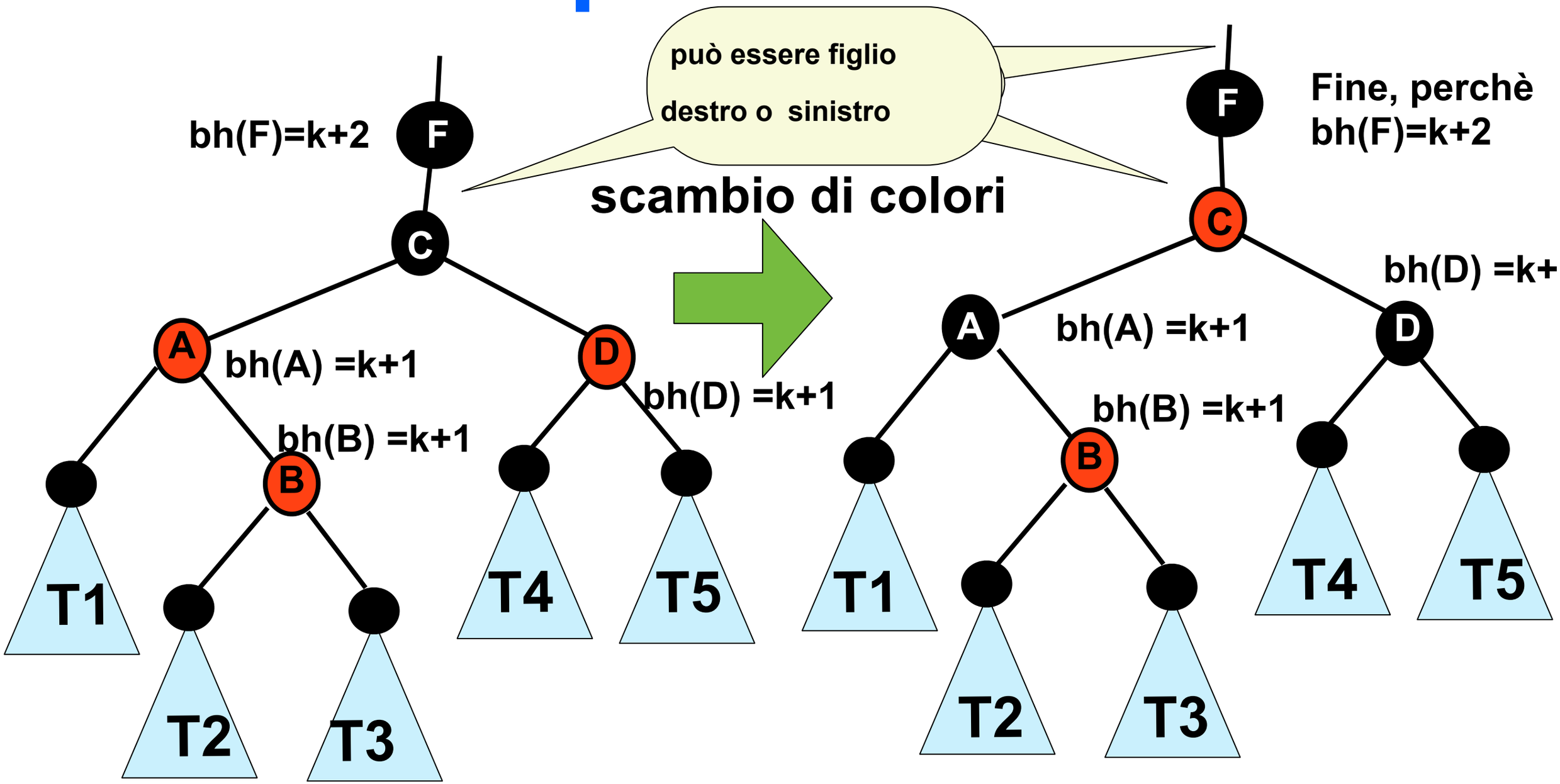


scambio di colori



N.B. Le foglie nil non saranno disegnate in tutti i casi

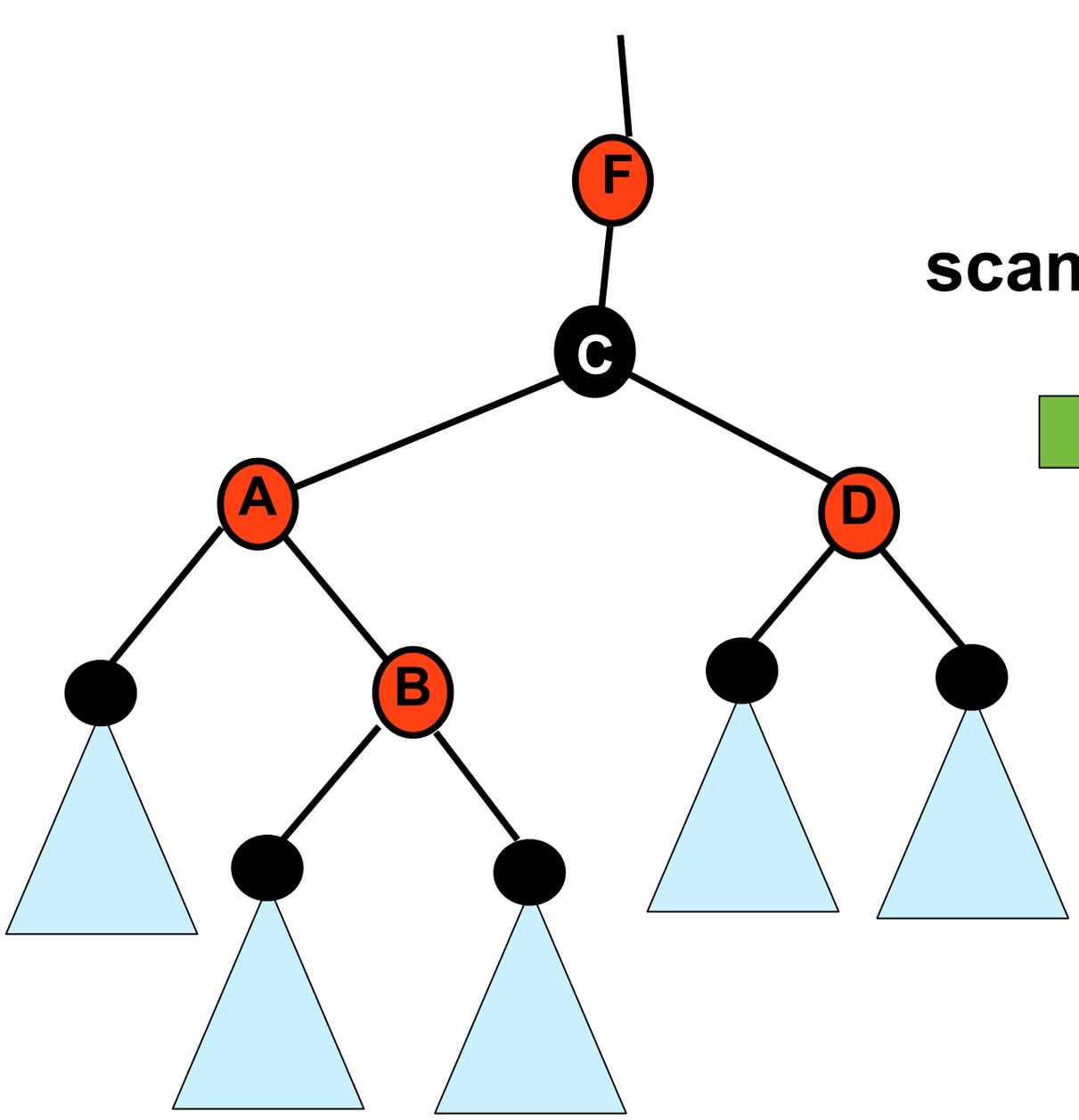
Caso 1: Il padre e lo zio rossi



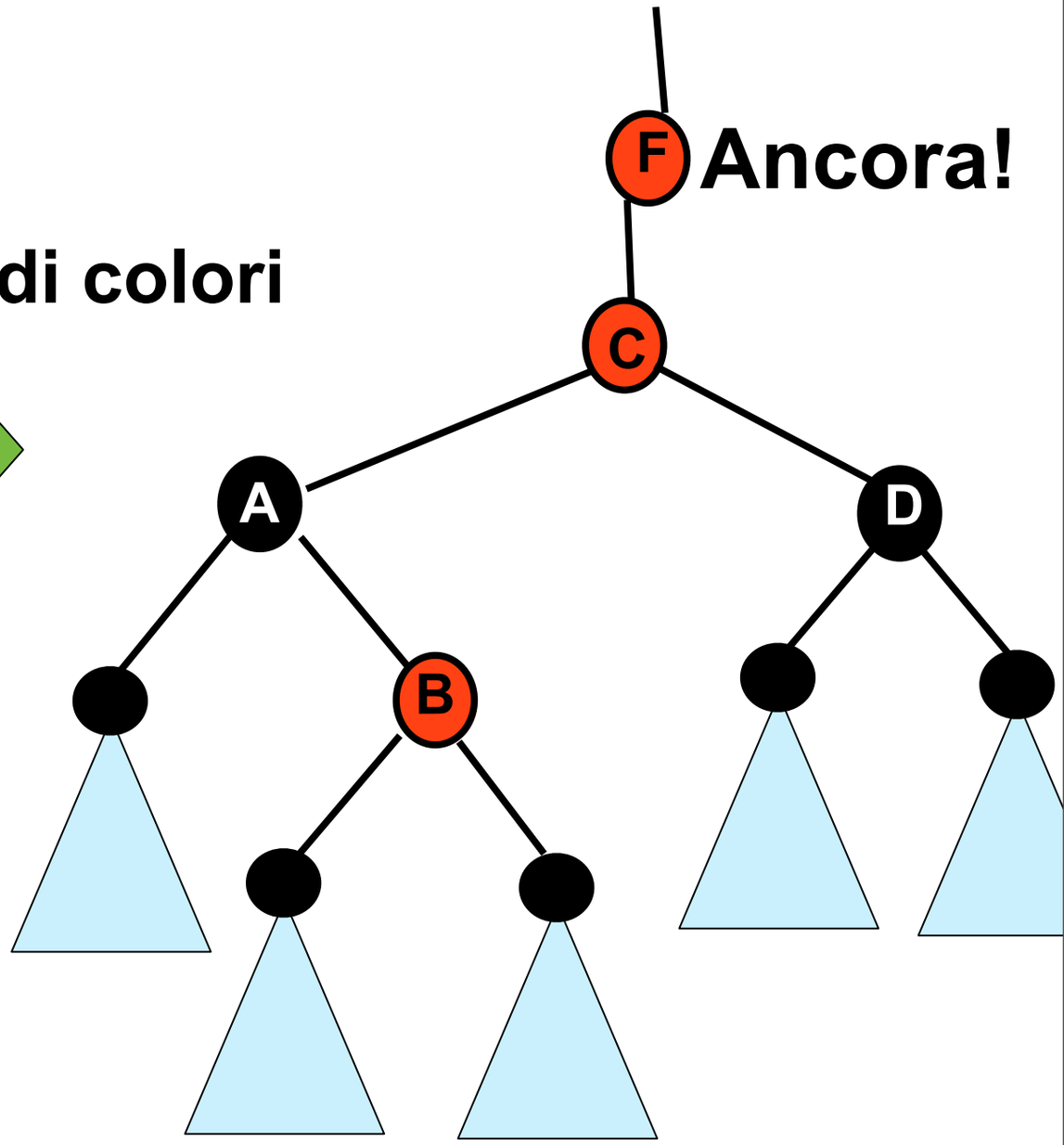
Se k è l'altezza nera dei sottoaberi Ti,1≤i≤5, bh(A)=bh(B)= bh(D) non cambia, mentre bh(C) = k+1prima della rotazione e bh(C)=k+2, dopo la rotazione, ma quella del padre di C invece, era k+2 e rimane k+2.

Se Il padre è nero quindi non ci sarà bisogno di altre operazioni.

Caso 1: Il padre e lo zio rossi



scambio di colori



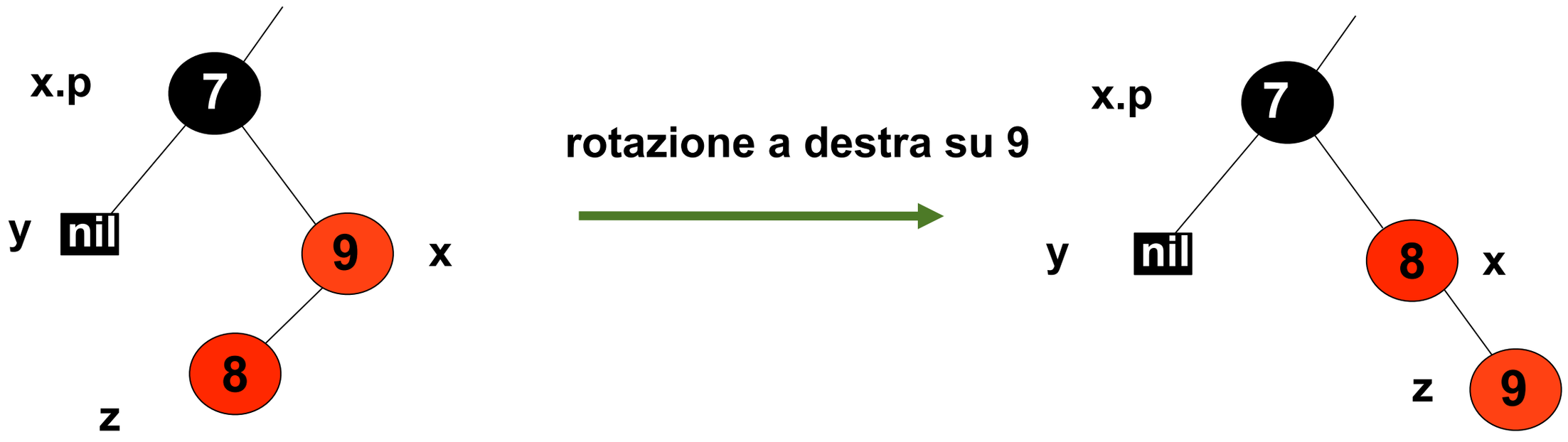
Il padre è rosso: violazione proprietà 4

Caso 2: nodo inserito come figlio destro di un nodo rosso figlio sinistro e zio nero

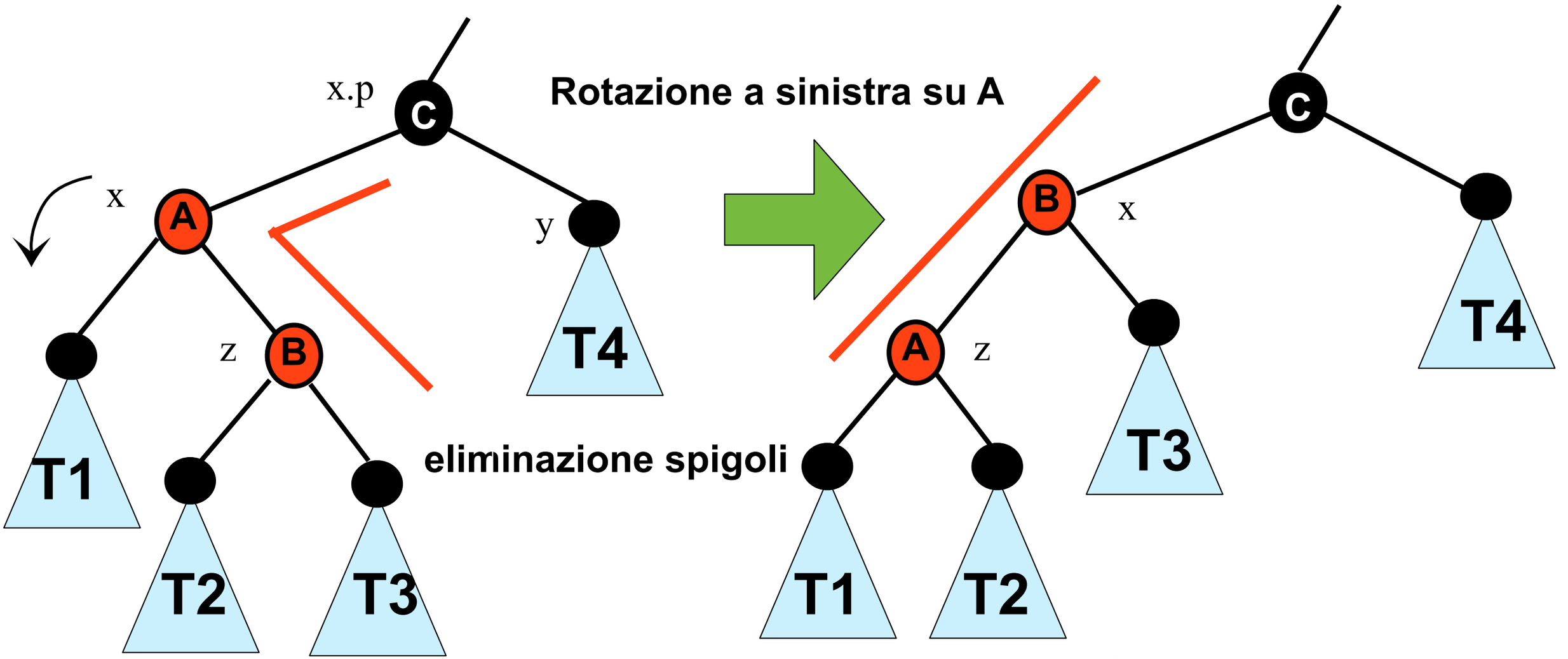
inseriamo z come figlio destro di un figlio sinistro

lo zio, y, è nero, il padre di x è nero per definizione di albero rosso-nero

Il caso simmetrico: il nuovo nodo è un figlio sinistro di un figlio destro



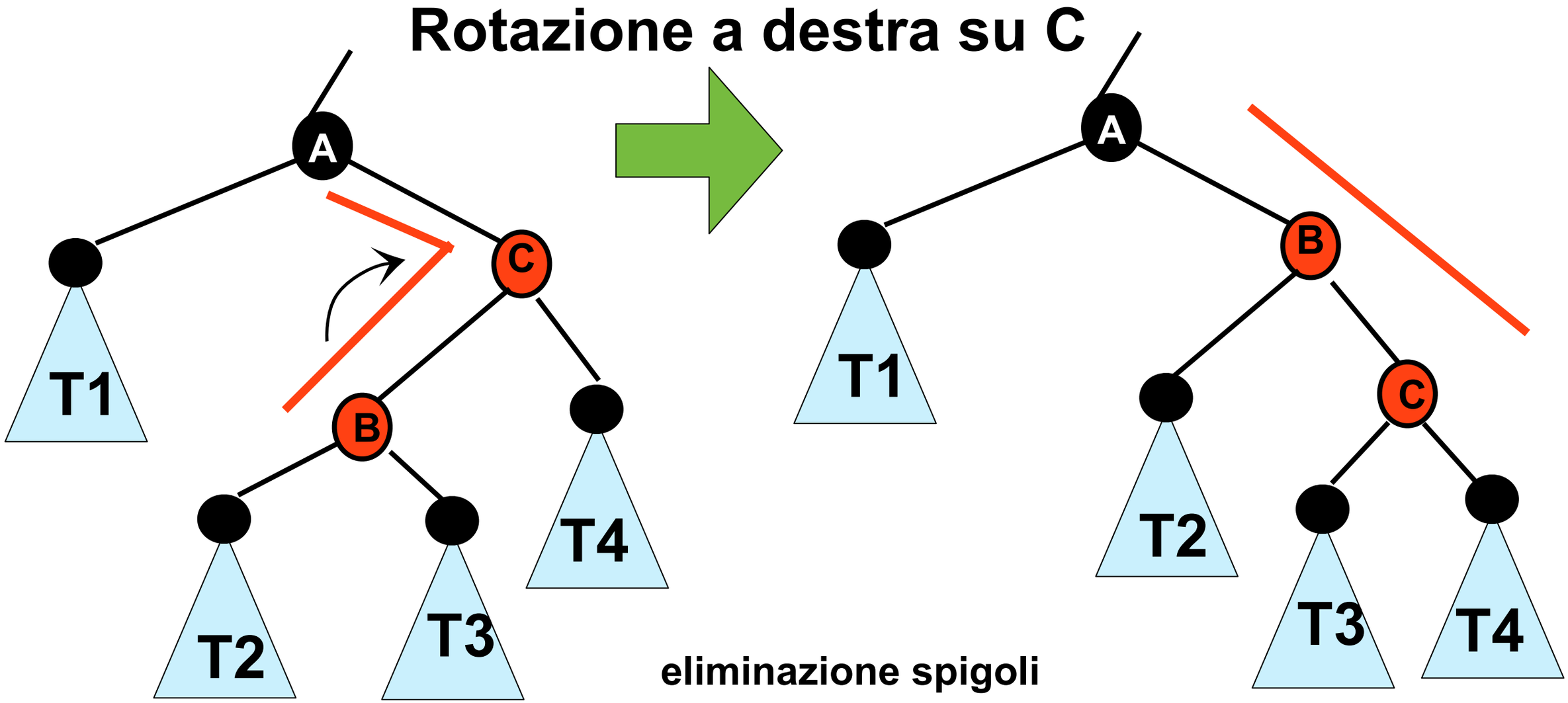
Caso 2: nodo rosso figlio destro di un nodo rosso figlio sinistro e zio nero



Si va nel caso 3

Se k è l'altezza nera dei sottoaberi Ti,1≤i≤4, allora bh(C)=k+1 prima e bh(C)=k+1 dopo l'operazione

Caso 2 simmetrico: nodo figlio sinistro di un nodo rosso figlio destro e zio nero

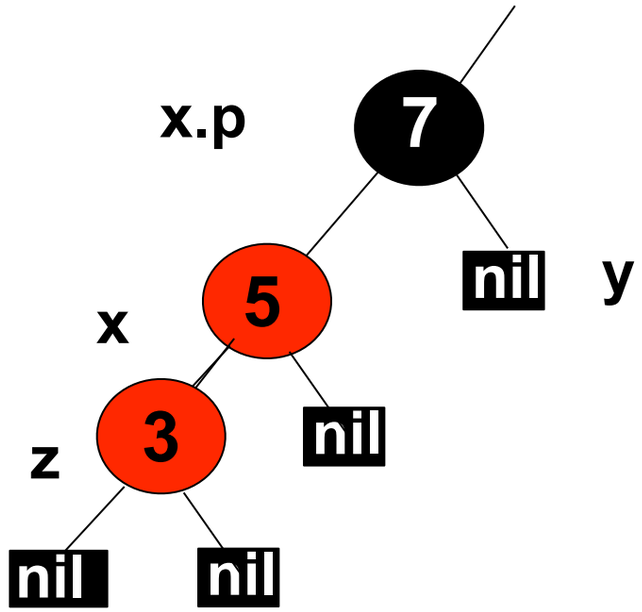


Si va nel caso 3

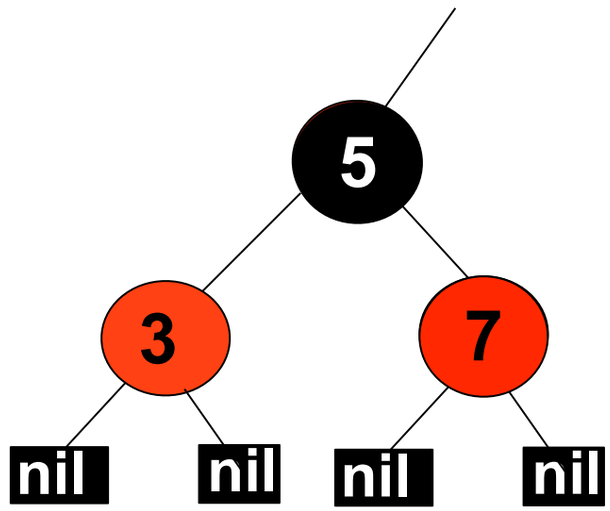
Caso 3: zio nero e nodo inserito come figlio sinistro di un nodo rosso figlio sinistro

inseriamo z come figlio sinistro di un figlio sinistro

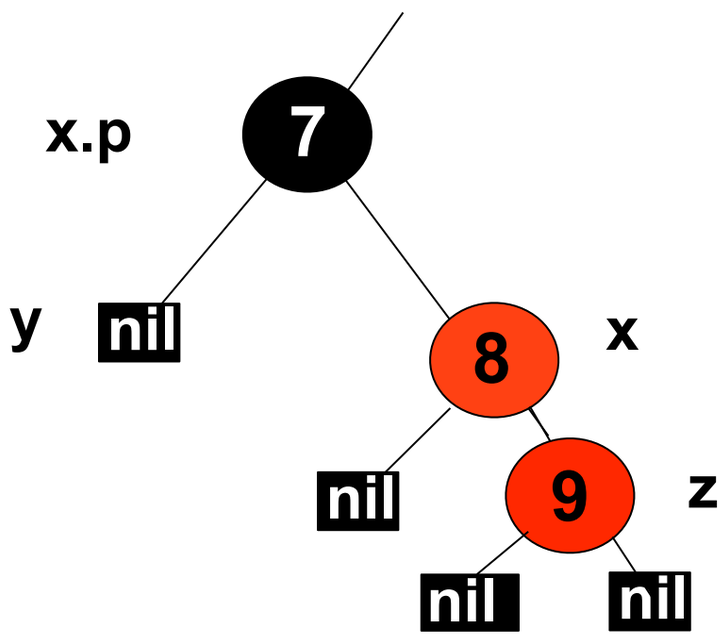
lo zio, y, è nero, x.p è nero per definizione di albero rosso-nero



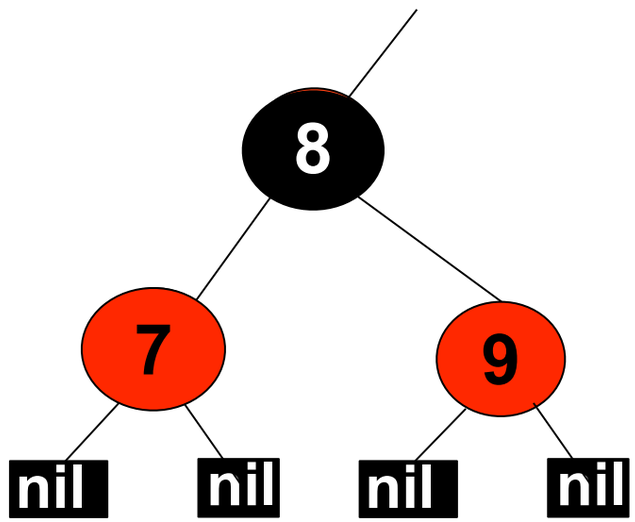
RB-rotazione a destra su 7



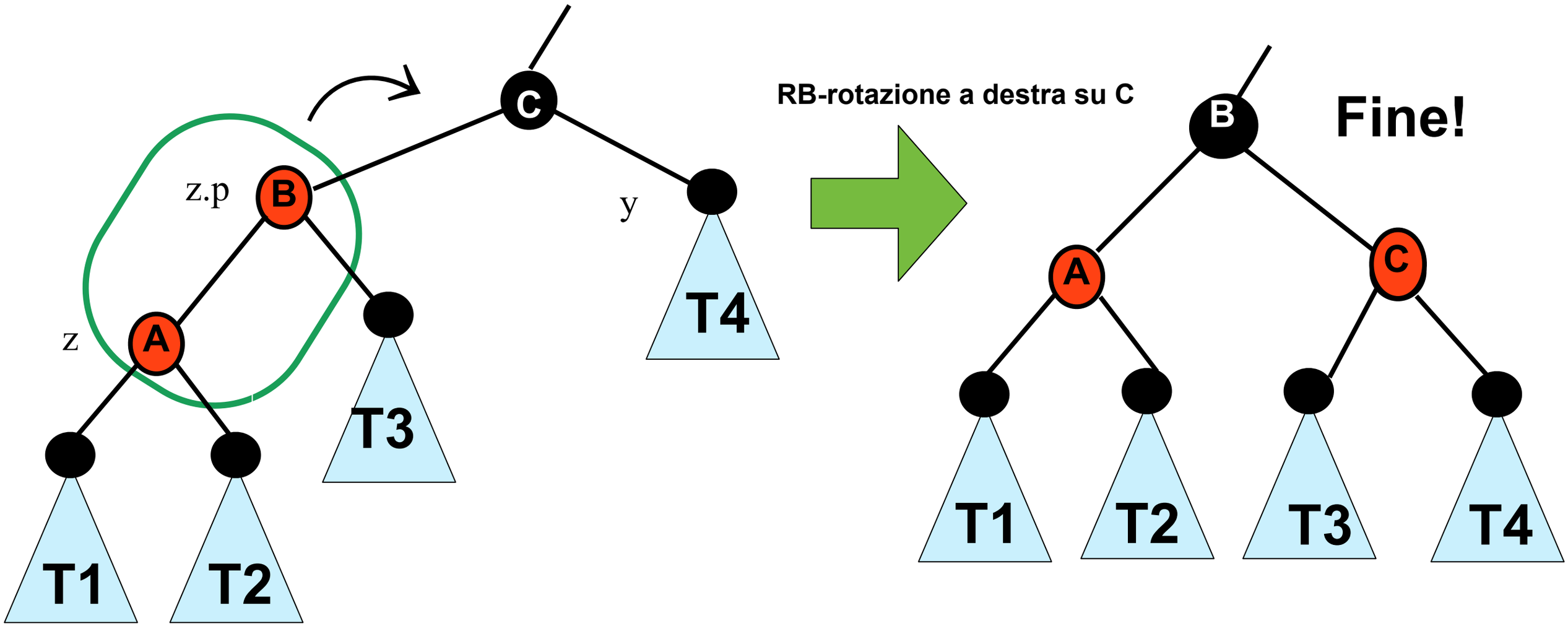
Il caso simmetrico: il nuovo nodo è un figlio destro



RB-rotazione a sinistra su 7

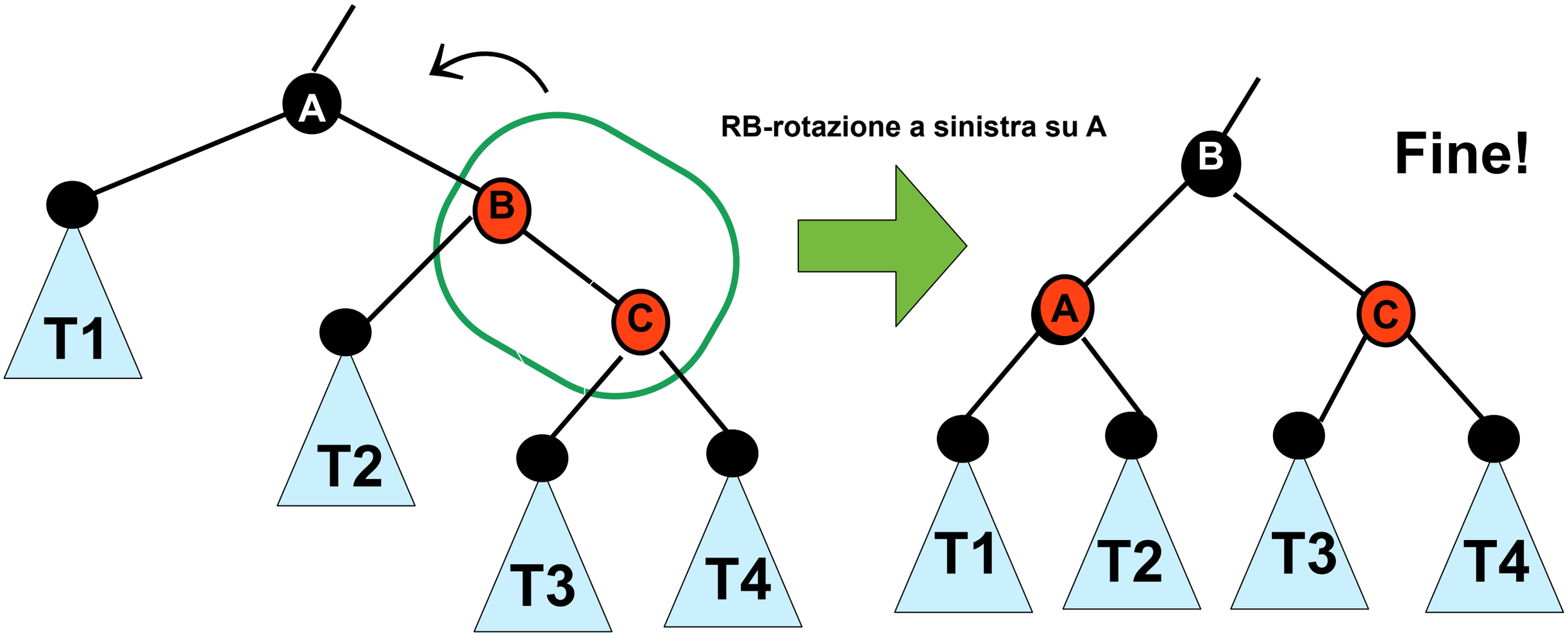


caso 3: il padre e figlio rossi entrambi figli sinistri e lo zio nero



Se k è l'altezza nera dei sottoaberi Ti,1≤i≤4, allora bh(C) = k+1 prima e bh(B)=k+1 dopo l'operazione, quindi non ci sono altre violazioni da prendere in considerazione

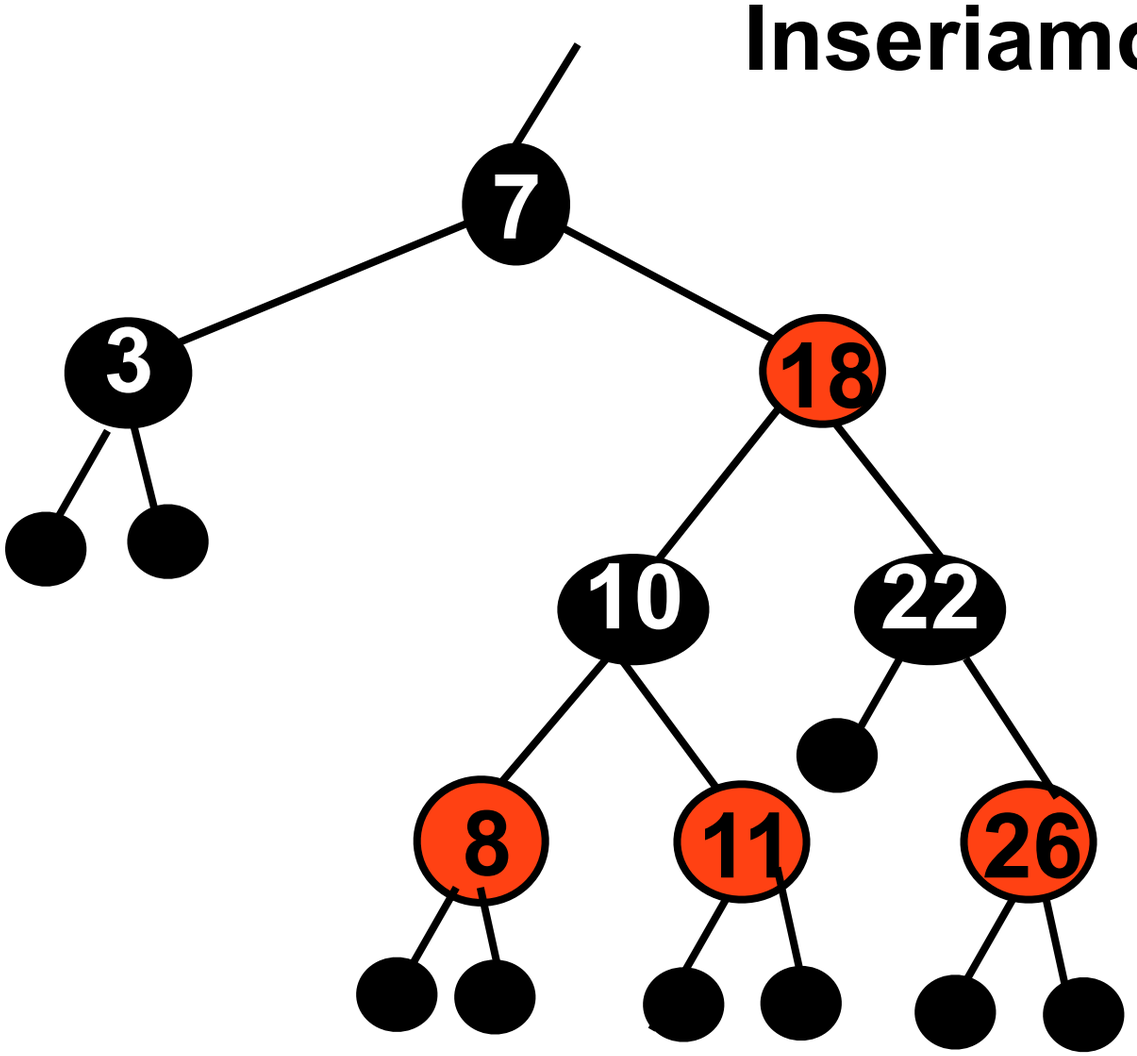
caso 3 simmetrico: il padre e figlio rossi entrambi figli destri e lo zio nero

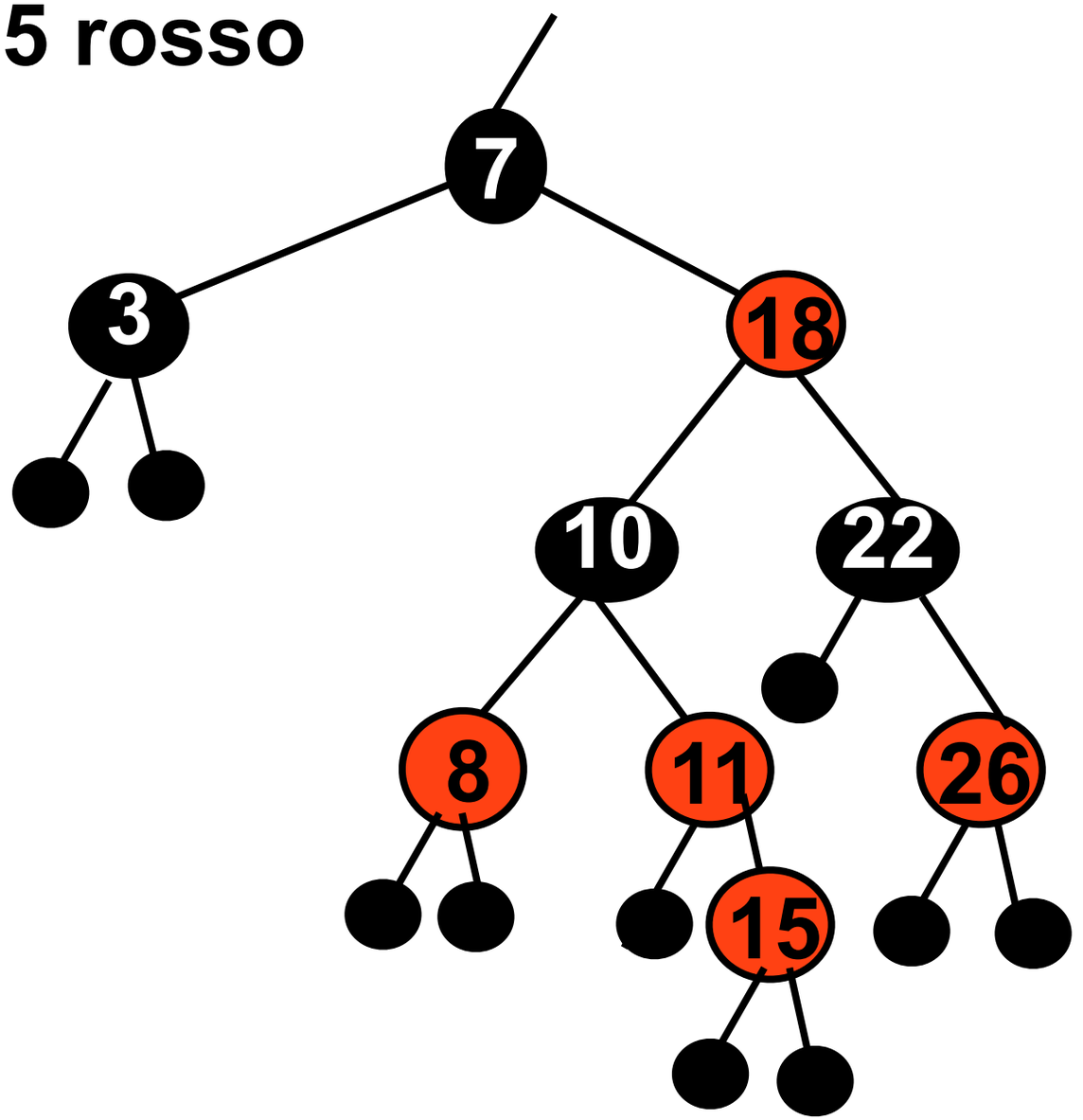


Fine!

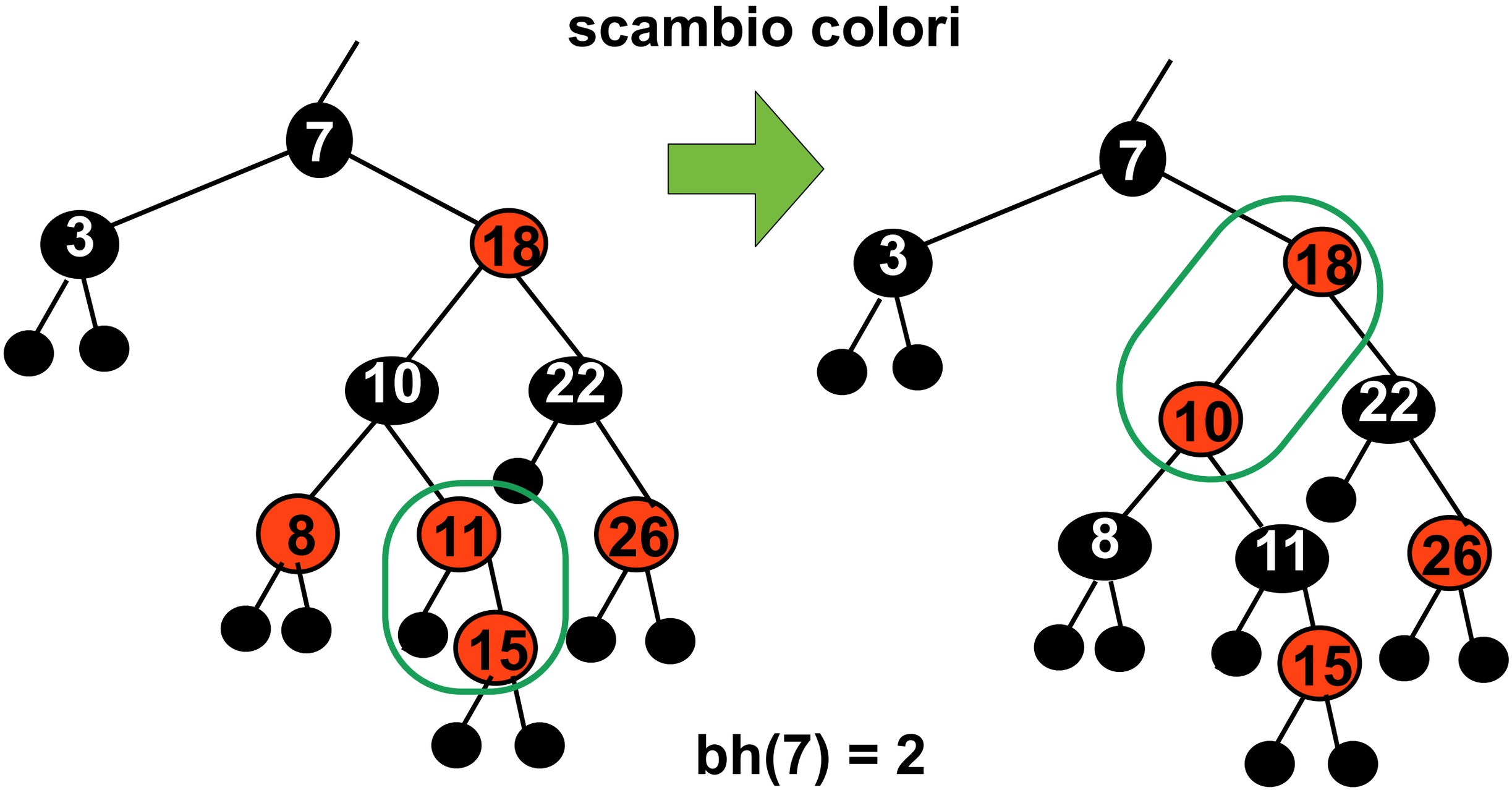
Esempio inserimento 1

Inseriamo 15 rosso



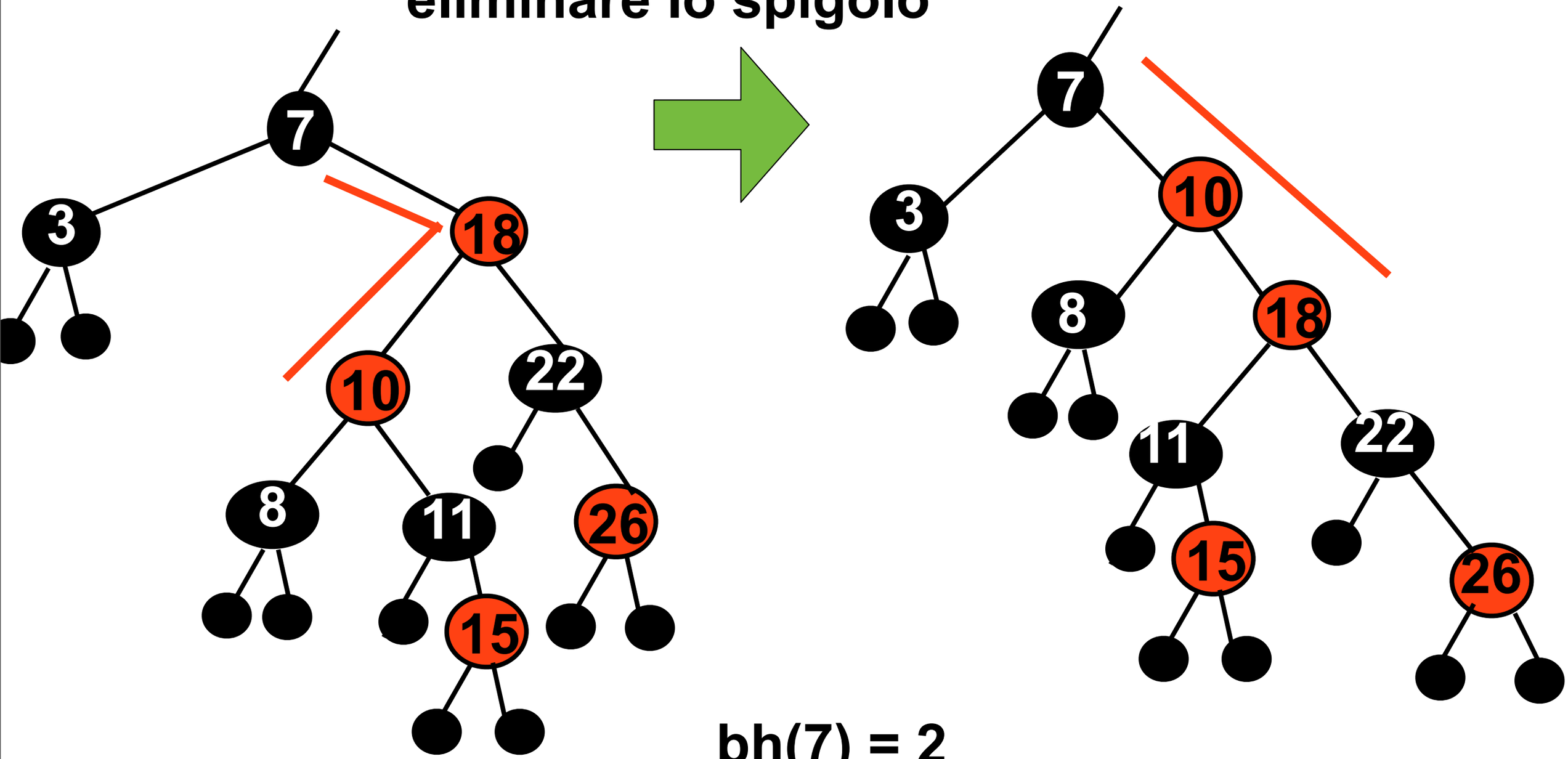


Esempio inserimento 2



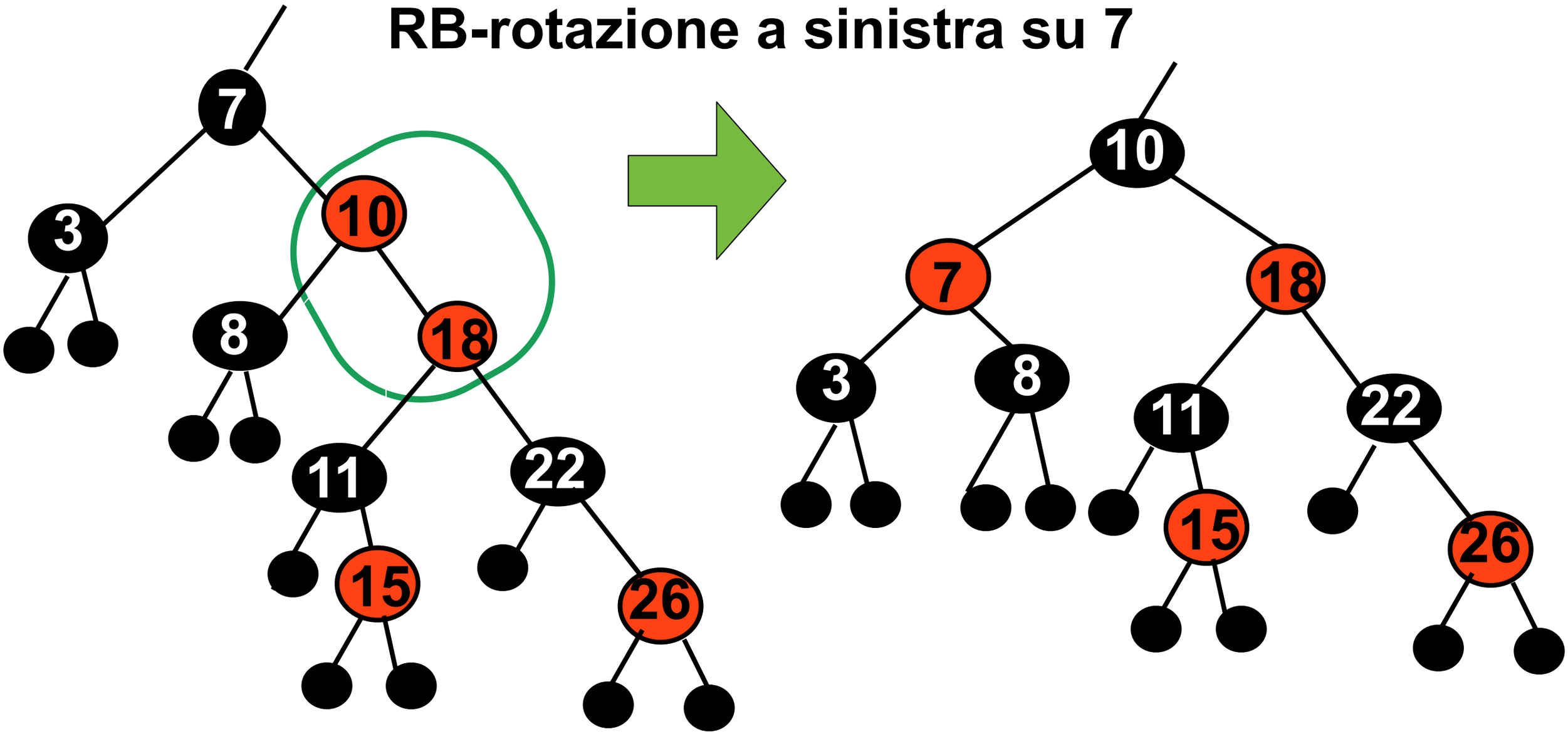
Esempio inserimento 3

rotazione a destra su 18, per eliminare lo spigolo



bh(7) = 2

Esempio inserimento, fine



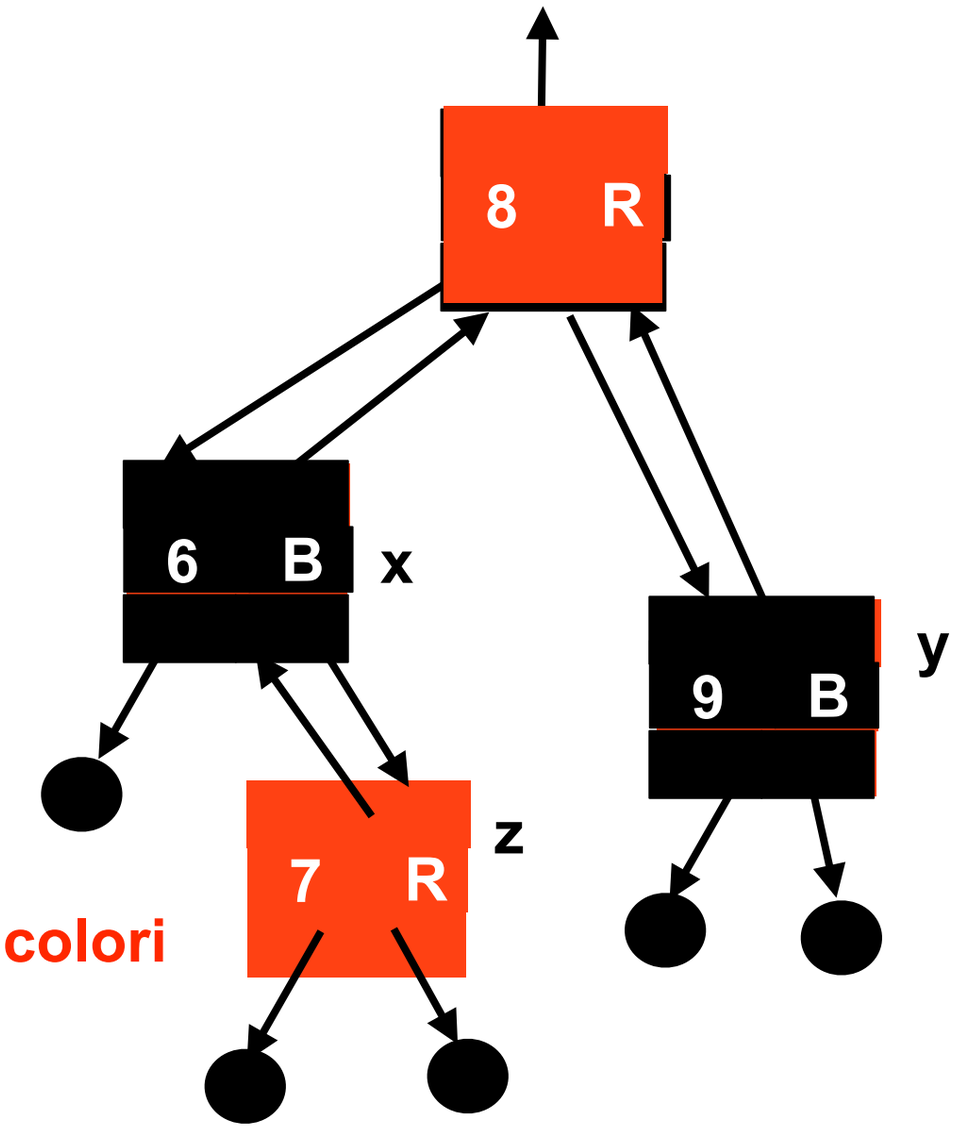
Pseudocodice inserimento

```
RB-Insert(T, z)        // z.left = z.right = T.NIL
Insert(T, z)           // qualche  modifica
dell'inserimento in un ABR
   z.color = RED
    // z è rosso. L'unica violazione
   // possibile delle proprietà degli alberi
    // rosso-neri è che z sia radice (proprietà 2)
  // oppure che z.p sia rosso (proprietà 4).
    RB-Insert-Fixup(T, z)
```

Ricordiamo che se z è la radice, suo padre è il nodo fittizio NIL, che è nero.

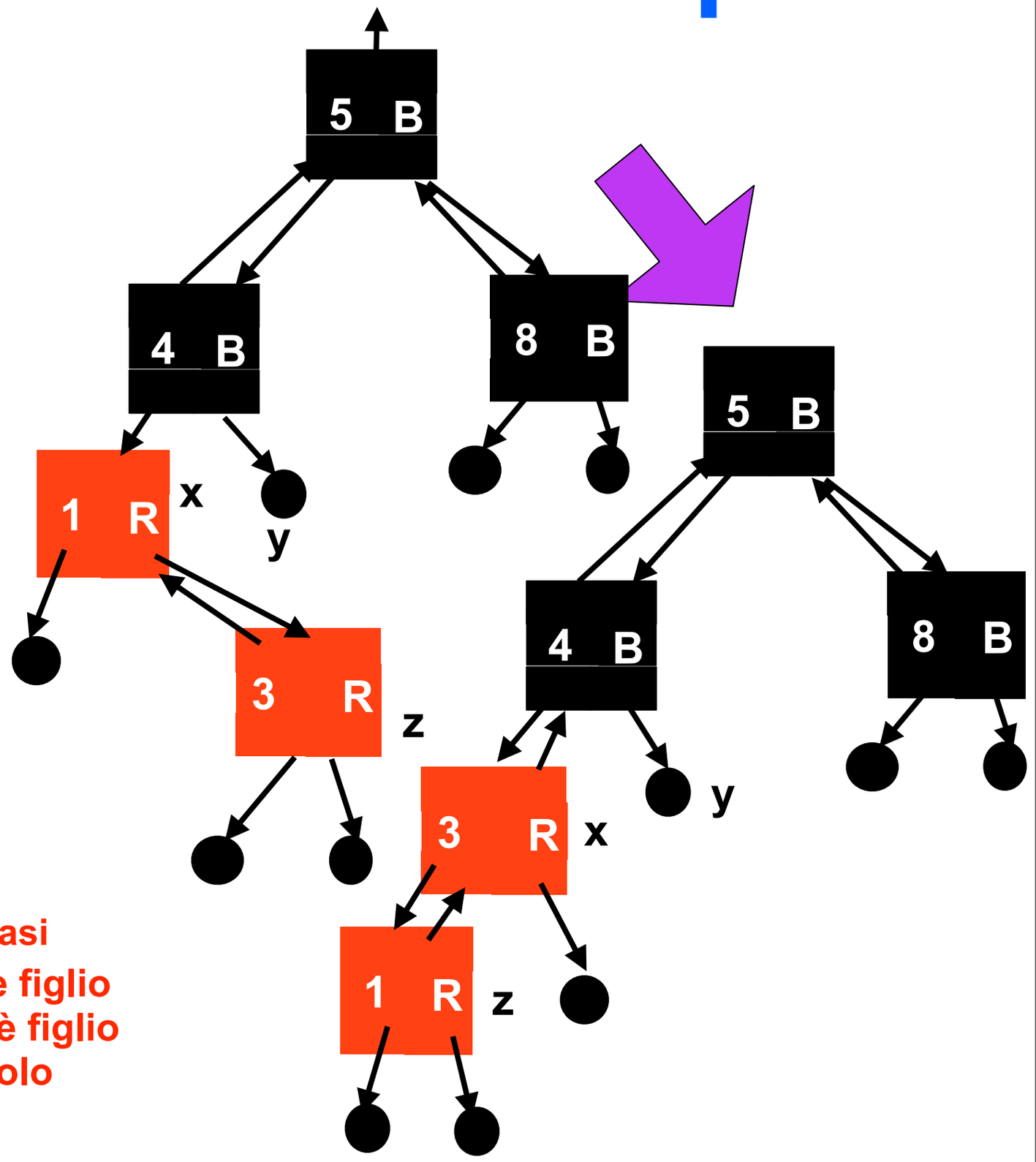
Pseudocodice InsertFixUp

```
RB-Insert-Fixup(T, z)
x = z.p  // z è il nodo eventualmente responsabile
della violazione della proprietà 3
 while    x.color == RED do
  Invariante:
1.      z è rosso
2.      se x = z.p è la radice allora è nero
3.      c'è al più una violazione della proprietà 4,
perchè z e suo padre sono rossi oppure c'è una
violazione della proprietà 2 perchè il nodo z è la
radice.
      if x == x.p.left // se x è un figlio sinistro
              then y = x.p.right
                        // y è lo zio di z: fratello di x
                    if y.color == RED  // caso 1: scambio di colori
                              then  x.color = BLACK
                                    y.color = BLACK
                                    x.p.color = RED
                                    z = z.p.p
                                    x = z.p
                              else     // y non è rosso: gli altri casi
```



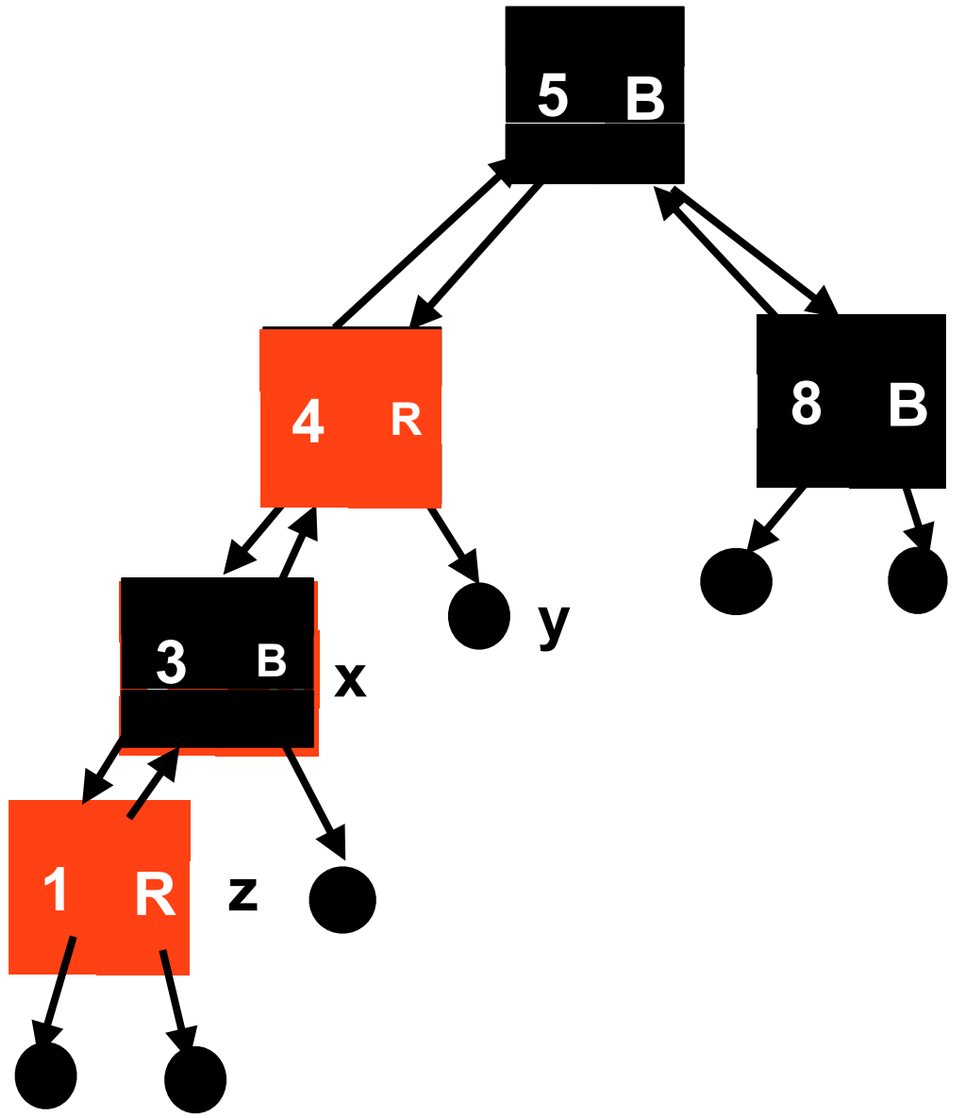
Pseudocodice InsertFixUp 2

```
RB-Insert-Fixup(T, z)
x = z.p  // z è il nodo eventualmente
responsabile della violazione della proprietà 3
while    x.color == RED do
Invariante:
1.     z è rosso
2.     se x = z.p è la radice allora è nero
3.     c'è al più una violazione della proprietà 3,
perchè z e suo padre sono rossi oppure c'è una
violazione della proprietà 2 perchè il nodo z è la
radice.
      if x == x.p.left // se x è un figlio sinistro
            then y = x.p.right
                  // y è lo zio di z: fratello di x
            if y.color == RED  // caso 1: scambio
            di colori
                  then  x.color = BLACK
                        y.color = BLACK
                        x.p.color = RED
                        z = z.p.p
                        x = z.p
                  else  // y non è rosso: gli altri casi
                        if z == x.right  // caso 2: z è figlio
                        destro, mentre suo padre è figlio
                        sinistro:eliminazione spigolo
                              then  z = x
                                    Left-Rot(T,z)
                                    x=z.p
```



Pseudocodice InsertFixUp 2

```
RB-Insert-Fixup(T, z)
x = z.p  // z è il nodo eventualmente
responsabile della violazione della proprietà 3
while    x.color == RED do
Invariante:
1.     z è rosso
2.     se x = z.p è la radice allora è nero
3.     c'è al più una violazione della proprietà 3,
perchè z e suo padre sono rossi oppure c'è una
violazione della proprietà 2 perchè il nodo z è la
radice.
      if x == x.p.left // se x è un figlio sinistro
            then y = x.p.right
                        // y è lo zio di z: fratello di x
            if y.color == RED  // caso 1: scambio
            di colori
                  then  x.color = BLACK
                              y.color = BLACK
                              x.p.color = RED
                              z = z.p.p
                              x = z.p
                  else    // y non è rosso: gli altri casi
// caso 2: z è                if z == x.right
figlio destro,                   then  z = x
mentre suo                             Left-Rot(T,z)
padre è figlio                         x = z.p
sinistro:elimina
zione spigolo          x.color = BLACK   Caso 3:
                       x.p.color = RED   RB rotazione
```

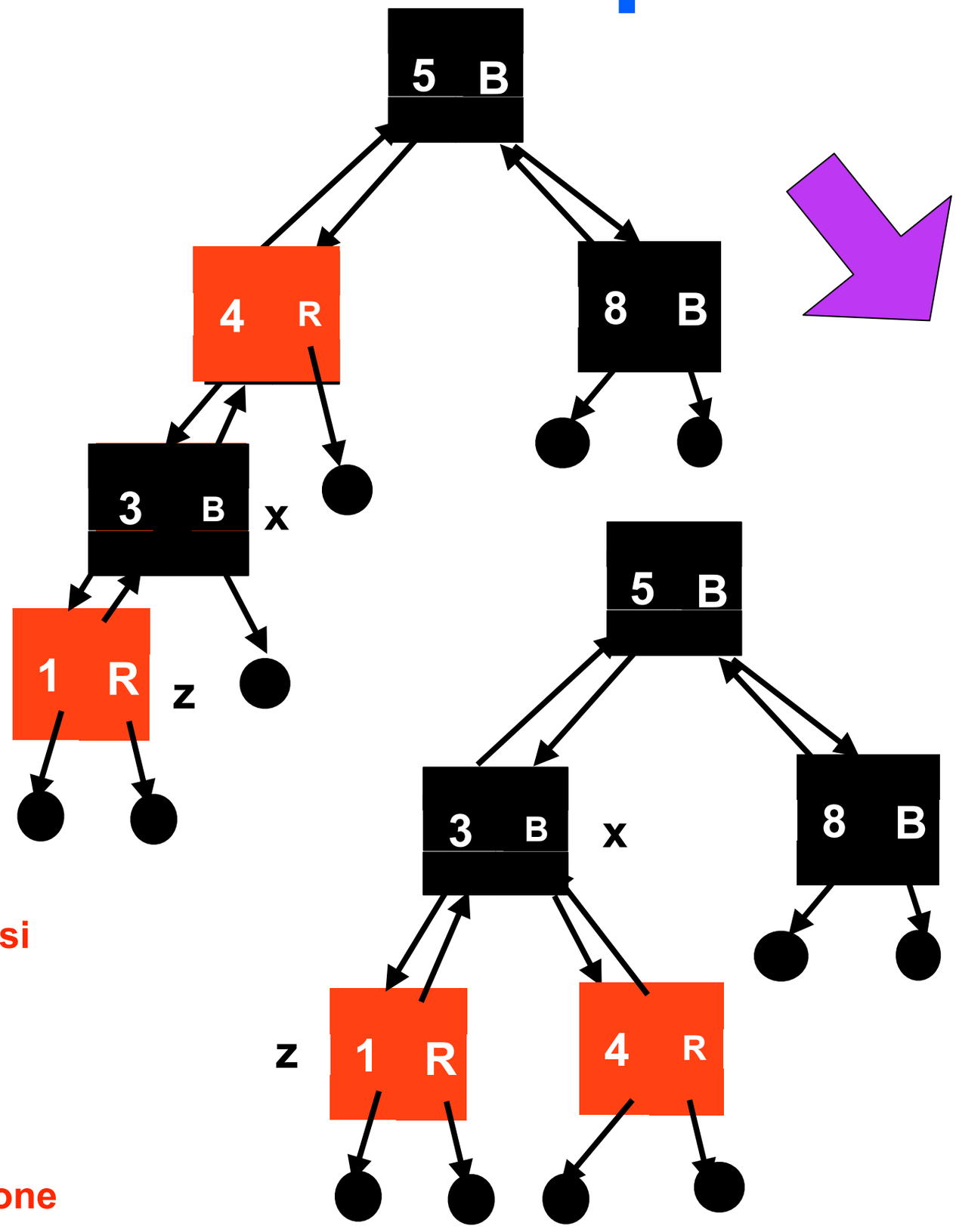


Pseudocodice InsertFixUp 2

```
RB-Insert-Fixup(T, z)
x = z.p  // z è il nodo eventualmente
responsabile della violazione della proprietà 3
while    x.color == RED do
Invariante:
1.     z è rosso
2.     se x = z.p è la radice allora è nero
3.     c'è al più una violazione della proprietà 3,
perchè z e suo padre sono rossi oppure c'è una
violazione della proprietà 2 perchè il nodo z è la
radice.
     if x == x.p.left // se x è un figlio sinistro
          then y = x.p.right
                    // y è lo zio di z: fratello di x
               if y.color == RED  // caso 1: scambio
               di colori
                    then  x.color = BLACK
                          y.color = BLACK
                          x.p.color = RED
                          z = z.p.p
                          x = z.p
                    else    // y non è rosso: gli altri casi
                         if z == x.right
                              then  z = x
                                    Left-Rot(T,z)
                                    x = z.p
                         x.color = BLACK
                         x.p.color = RED
                         Right-Rot(T,x.p)
```

// caso 2: z è figlio destro, mentre suo padre è figlio sinistro:elimina zione spigolo

Caso 3: RB rotazione



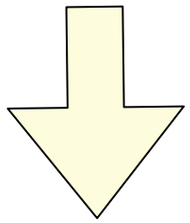
```
RB-Insert-Fixup(T, z)
x = z.p  // z è il nodo eventualmente
responsabile della violazione della proprietà 3
while    x.color == RED do
Invariante:
1.    z è rosso
2.    se x = z.p è la radice allora è nero
3.    c'è al più una violazione della proprietà 3,
perchè z e suo padre sono rossi oppure c'è una
violazione della proprietà 2 perchè il nodo z è la
radice.
     if x == x.p.left // se x è un figlio sinistro
          then y = x.p.right
                        // y è lo zio di z: fratello di x
               if y.color == RED  // caso 1: scambio
               di colori
                        then  x.color = BLACK
                              y.color = BLACK
                              x.p.color = RED
                              z = z.p.p
                              x = z.p
               else    // y non è rosso: gli altri casi
// caso 2: z è       if z == x.right
figlio destro,              then  z = x
mentre suo                        Left-Rot(T,z)
padre è figlio                    x = z.p
sinistro:elimina          x.color = BLACK    Caso 3:
zione spigolo             x.p.color = RED    RB rotazione
                          Right-Rot(T,x.p)
     else      ...
// il caso simmetrico: x==x.p.right  come il ramo then, ma con left e right scambiati
T.root.color = BLACK
// all'uscita dal ciclo si colora di nero la radice
```

$T(n) = O(\lg n)$

I casi

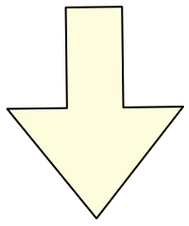
Riassumendo i casi sono

Caso 1: zio rosso (con figli neri, per definizione)



può essere conclusivo o portare ai casi 1,2 o 3 <u>per il nonno</u> dopo uno scambio di colori

Caso 2 (spigolo): zio nero e figlio destro rosso (sinistro) di un figlio sinistro rosso (destro)



porta al caso 3, dopo una rotazione semplice che elimina lo spigolo

Caso 3: zio nero e figlio sinistro rosso (destro) di un filgio sinistro (destro) rosso, comporta una RB-rotazione.

È conclusivo

Complessità

L' inserimento da solo prende un tempo O(lg n)

I casi 2 e 3 vengono eseguiti e terminano immediatamente e dunque essi richiedono tempo costante, avendo effettuato al più due rotazioni e un numero costante di ricolorazioni.

Solo il caso 1 può comportare altre operazioni sul nonno del nuovo nodo inserito, al quale si può applicare uno dei tre casi.
Quindi anche le operazioni di ripristino delle proprietà comportano un costo O(lg n).

Alberi binari di ricerca: confronto

Ben Pfaf, dip. Computer Science della Stanford Un. ha pubblicato nel 2004 un articolo in cui dà conto di una serie di risultati di confronto sperimentali sulle operazioni di ricerca, inserimento e cancellazione per i normali BST (ABR), gli AVL, i rosso-neri e gli "splay trees". Uno splay tree (albero aperto, disteso?) è un BST autobilanciante in cui si spostano verso la radice gli elementi cui si accede. Gli splay alberi erano introdotti da D. Sleator e R. Tarjan nel 1985, e li studierete alla magistrale.

Qui riportiamo alcuni esiti dei confronti sperimentali tra rosso-neri, BST e AVL:

Tempo: i **rosso-neri sono i migliori quando le sequenze di elementi inseriti sono costruite a caso (la regola di bilanciamento per gli alberi rosso-neri è più permissiva di quella degli AVL quindi si fanno meno operazioni di ribilanciamento) con occasionali inserimenti di sequenze di elementi ordinati, se invece gli inserimenti di sequenze di elementi ordinati sono prevalenti allora sono gli AVL a comportarsi meglio.**

Se le sequenze di elementi inseriti sono costruite a caso a comportarsi meglio sono i semplici BST, ma se i dati sono inseriti in ordine allora i BST degenerano in liste.

Spazio: Per l'occupazione della memoria gli AVL richiedono 2 extra bits di memoria e i rosso-neri solo uno.

Se ci sono i puntatori al padre si deve aggiungere un campo puntatore a ogni nodo, ma questa rappresentazione risulta la più efficiente in tempo per tutte le classi di alberi considerate.