Algoritmi e Strutture Dati

Alberi Bilanciati: Alberi Red-Black

Alberi bilanciati di ricerca

- **Gli *alberi binari di ricerca* sono semplici da gestire (inserimenti e cancellazioni facili da implementare) ma hanno prestazioni poco prevedibili e potenzialmente basse**
- **Gli *alberi perfettamente bilanciati* hanno prestazioni ottimali (*log n* garantito) ma inserimenti e cancellazioni complesse (ri-bilanciamenti)**
- **Alberi AVL (*Adelson-Velskii e Landis*): *alberi quasi bilanciati*. Buone prestazioni e gestione relativamente semplice.**

Alberi AVL: definizione

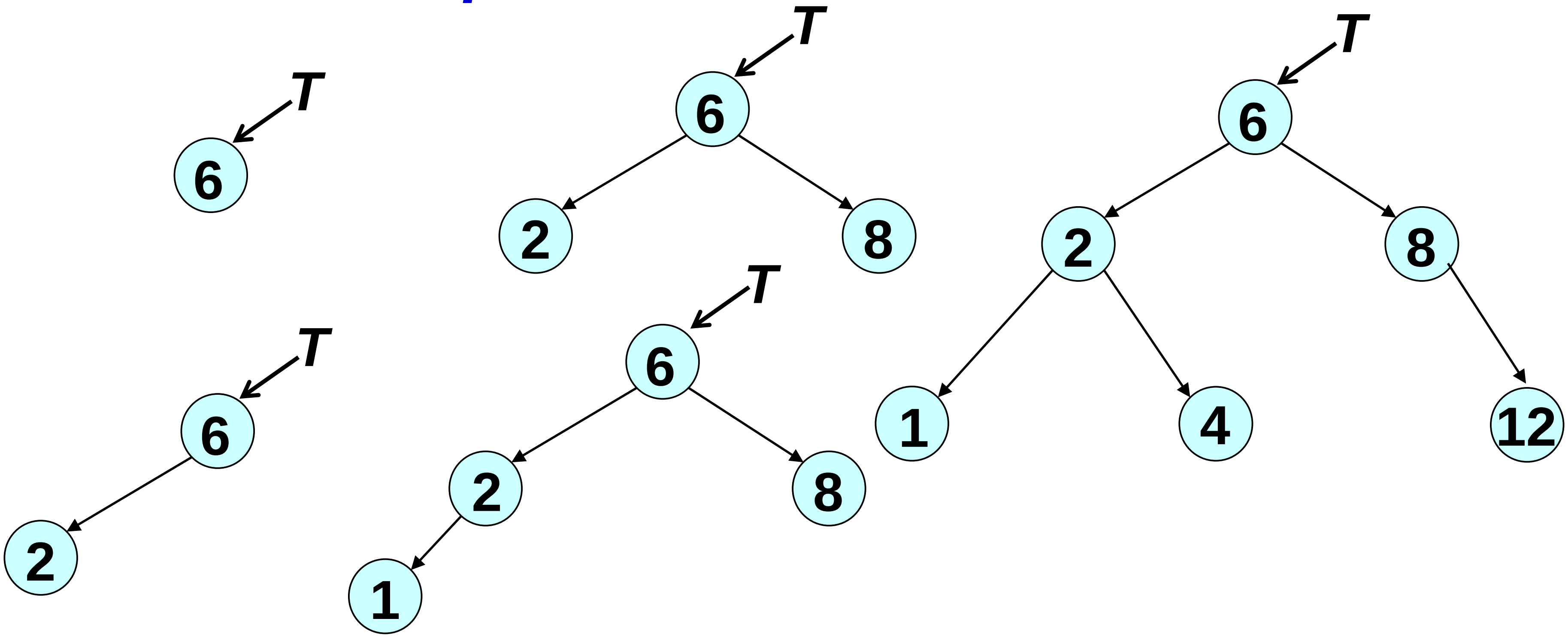
Definizione: **Un albero binario di ricerca è un *Albero AVL* se per ogni nodo *x*:**

- **l'*altezza* del *sottoalbero sinistro di x* e quella del *sottoalbero destro di x* differiscono al più di uno, e**

- ***entrambi i sottoalberi* sinistro e destro di *x* sono *alberi AVL***

Alberi perfettamente bilanciati

Definizione: **Un albero binario si dice *<u>Perfettamente Bilanciato</u>* se, per ogni nodo *i*, il *numero dei nodi* nel suo *sottoalbero sinistro* e il *numero dei nodi* del suo *sottoalbero destro differiscono al più* di 1**



Alberi perfettamente bilanciati

Definizione: **Un albero binario si dice *Perfettamente Bilanciato* se, per ogni nodo *i*, il *numero dei nodi* nel suo *sottoalbero sinistro* e il *numero dei nodi* del suo *sottoalbero destro* *differiscono al più* di 1**

L'*altezza* di un *albero perfettamente bilanciato (APB)* con *n nodi* è $h = \log_2 n$

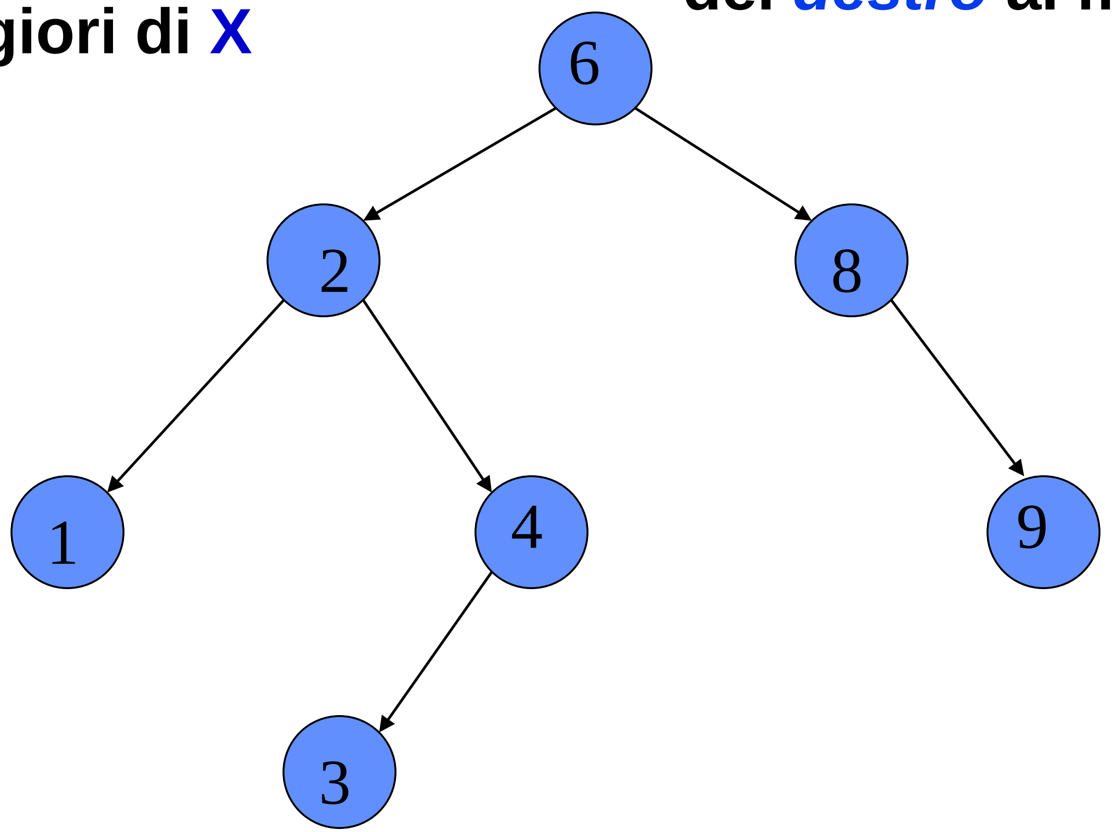
Alberi AVL: alberi binari bilanciati

Proprietà* degli *ABR

Per ogni nodo X, i nodi del sottoalbero sinistro sono minori del nodo X, e i nodi del sottoalbero destro sono maggiori di X

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.

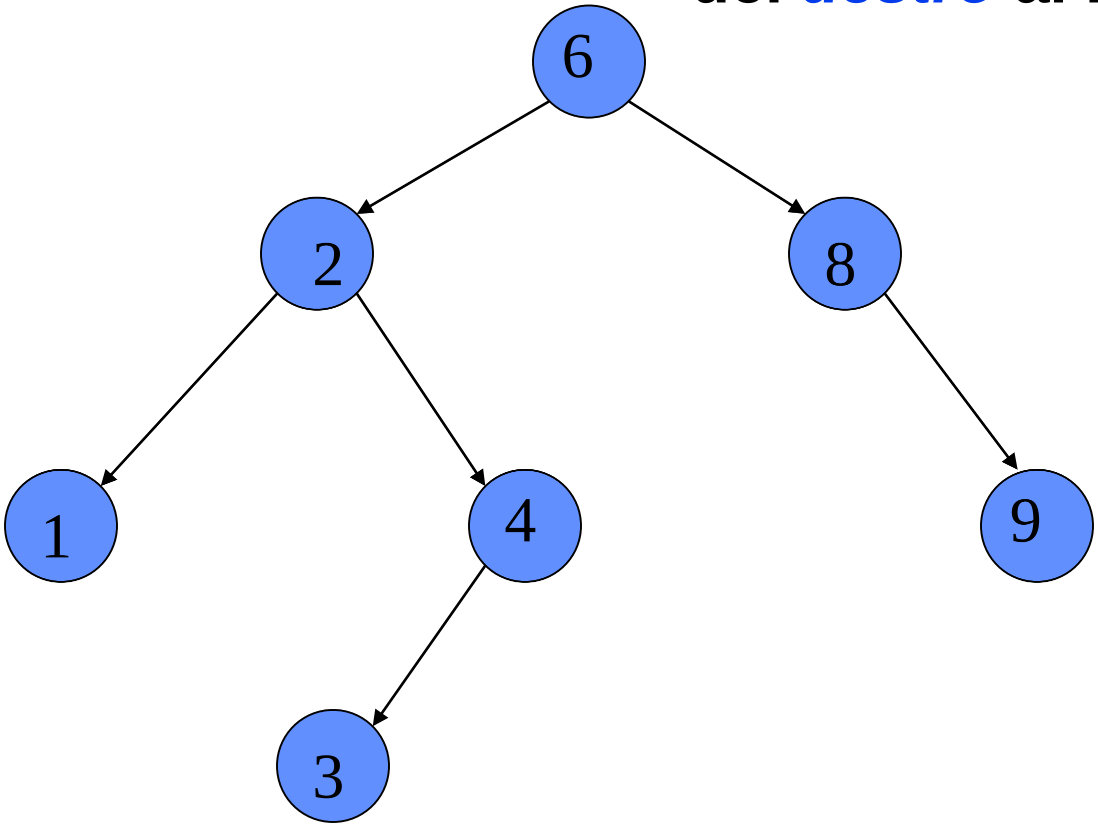


Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.

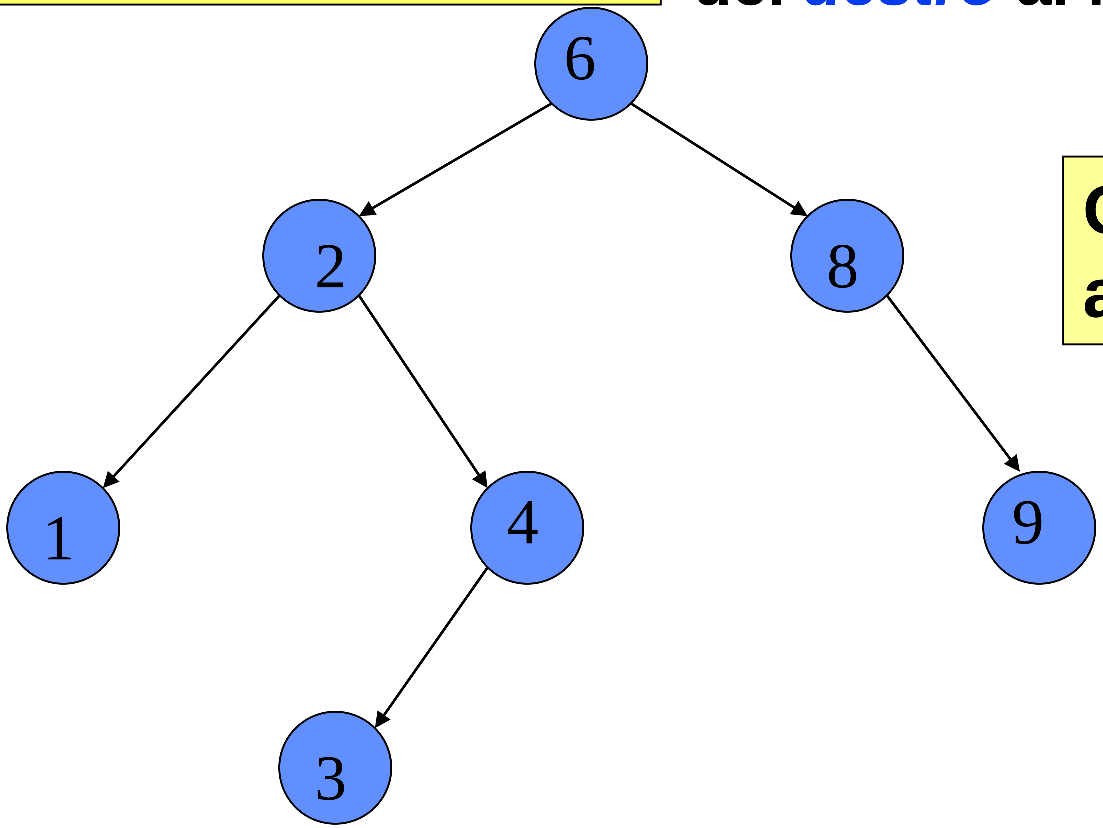


Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1
Altezza(∅) = -1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.



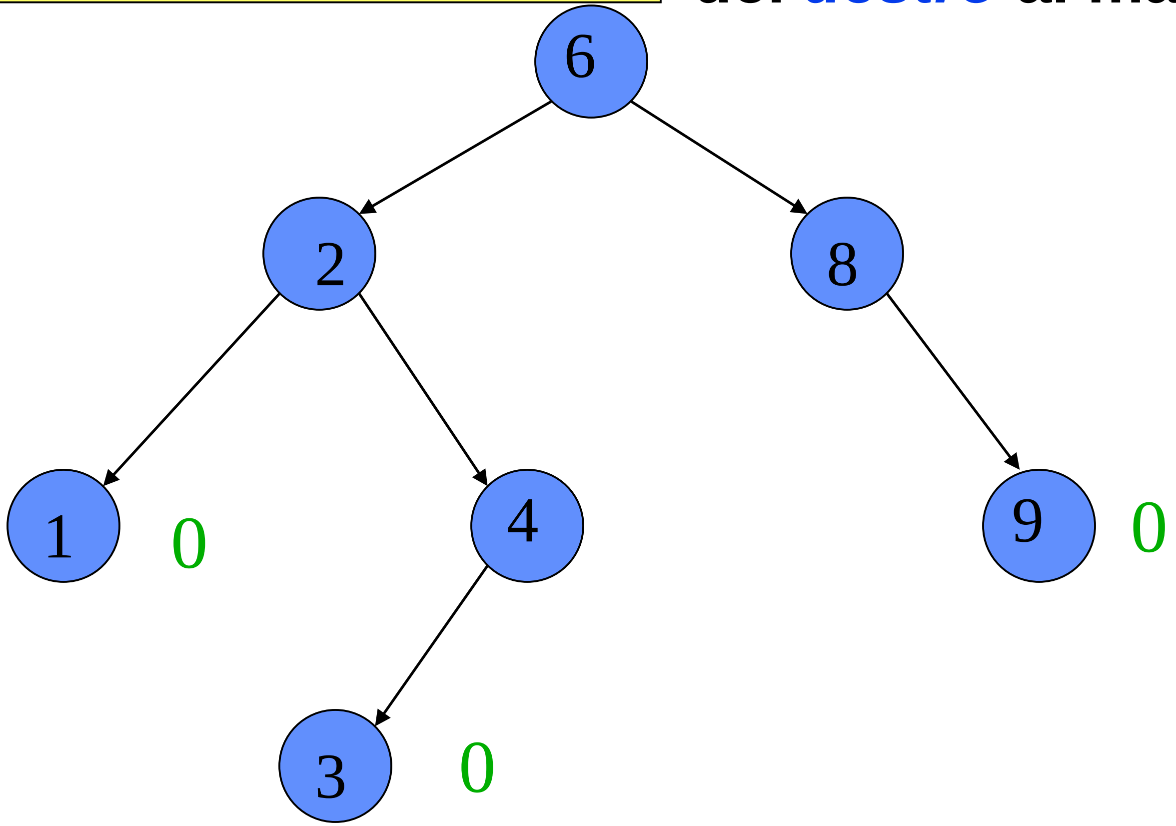
Questo è un albero AVL?

Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1
Altezza(∅) = -1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.

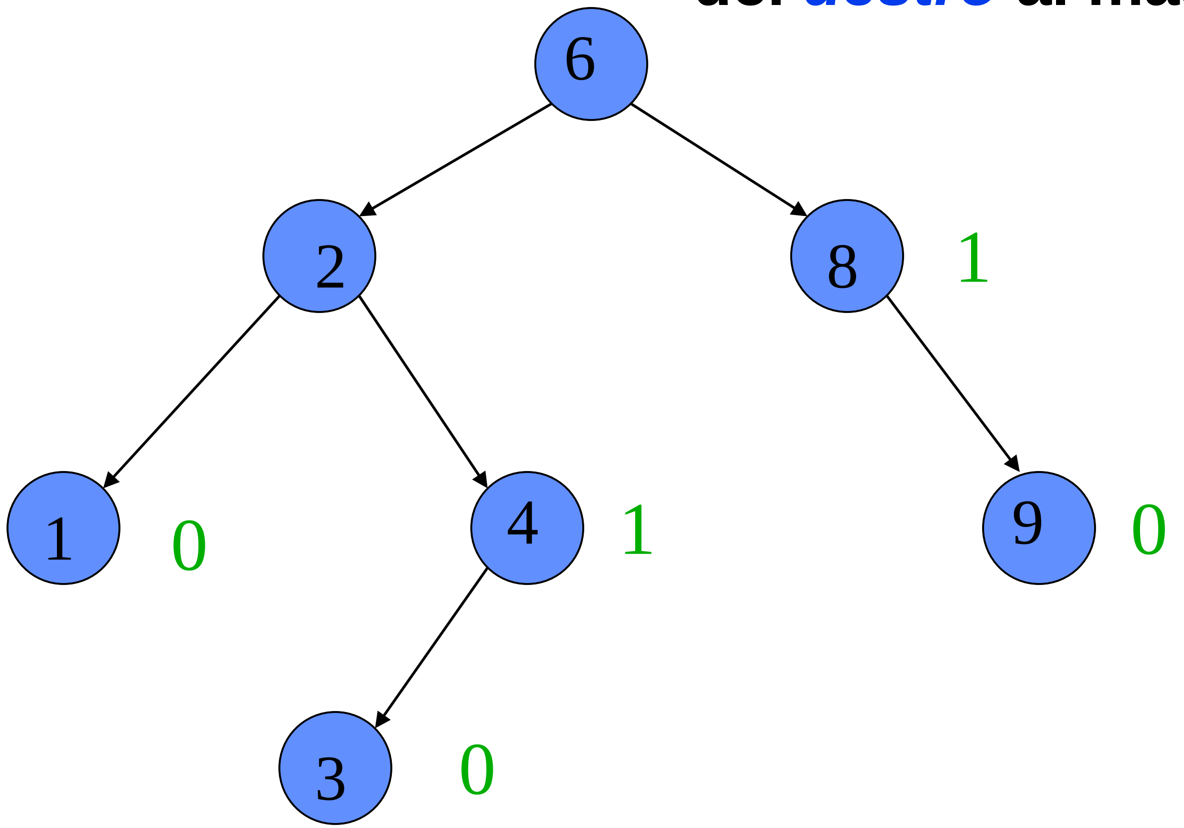


Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.

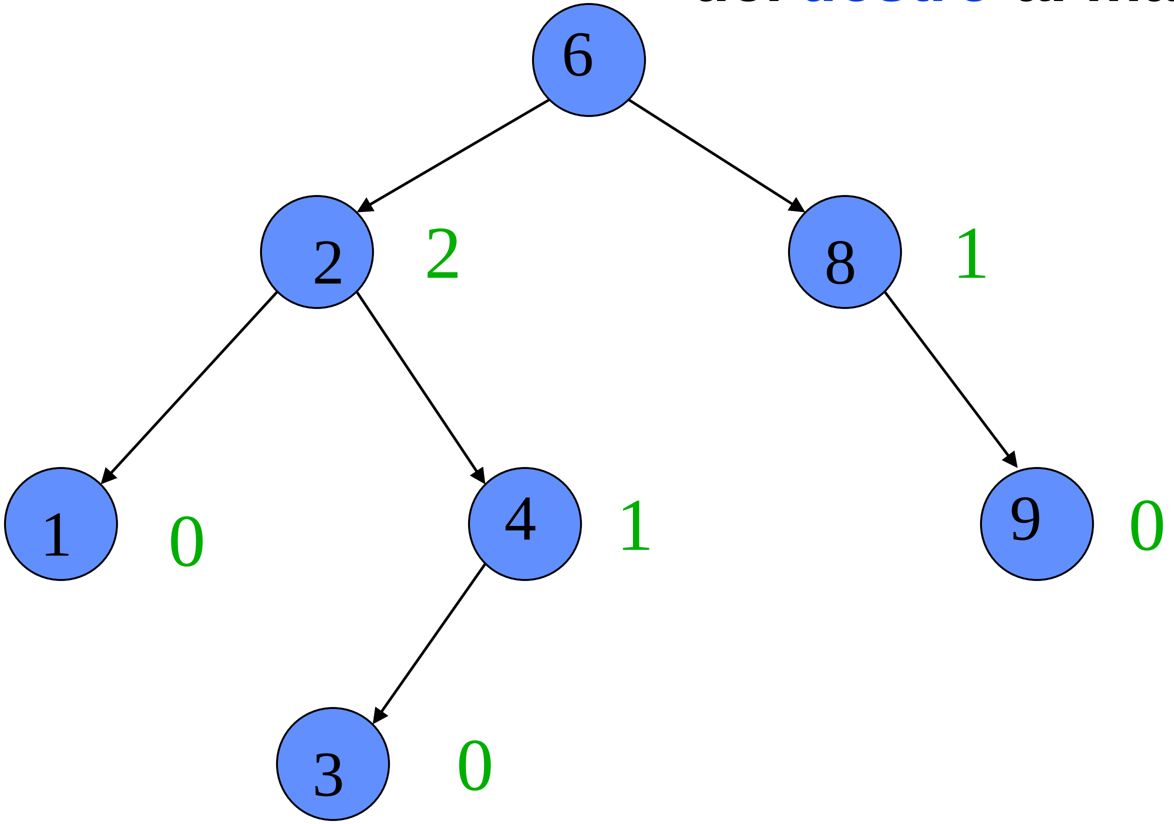


Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.

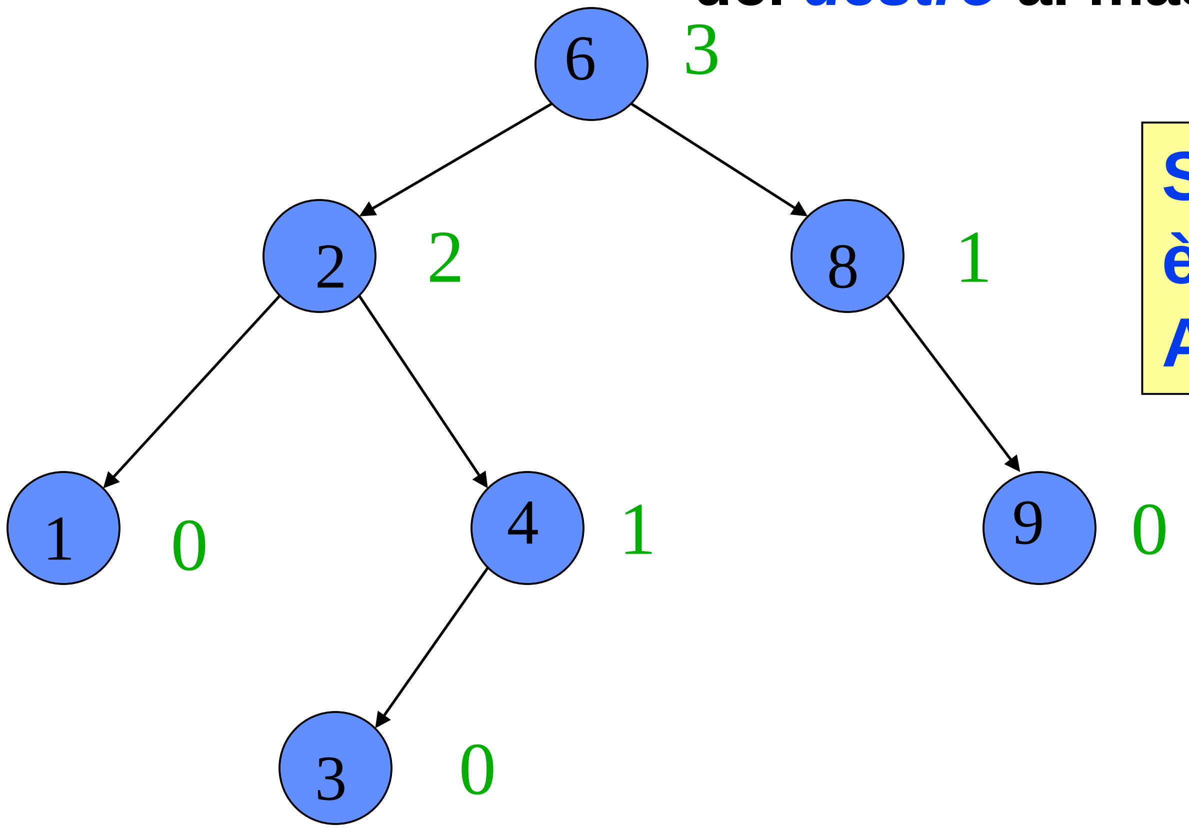


Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.



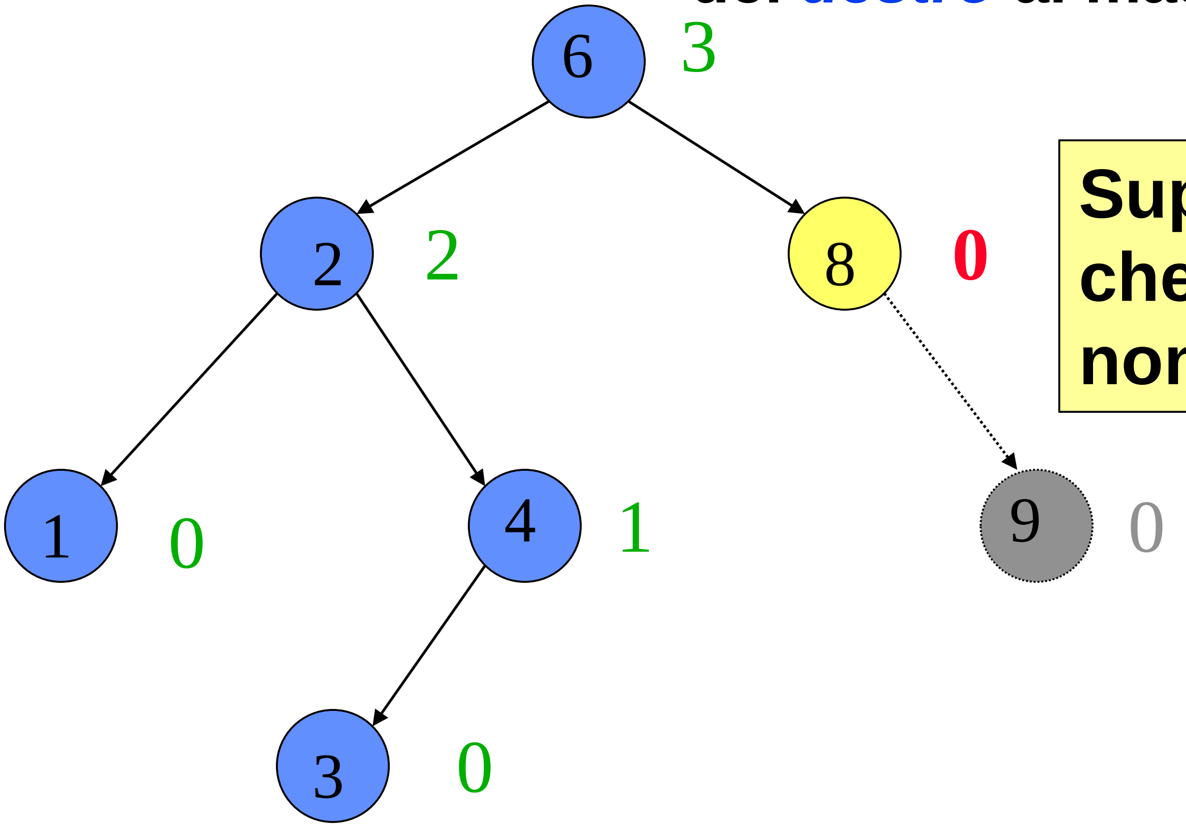
Sì! Questo è un albero AVL.

Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.



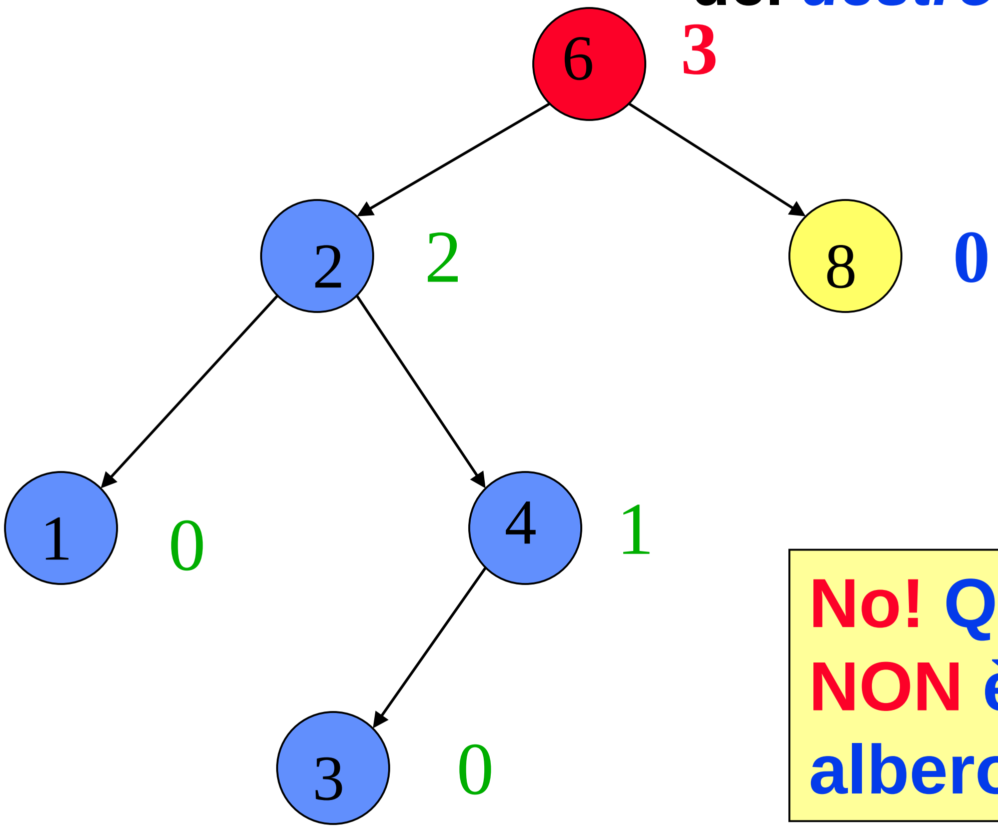
Supponiamo che il nodo 9 non ci sia.

Alberi AVL: alberi binari bilanciati

Altezza(*X*) =
max(Altezza(sinistro(*X*)),
Altezza(destro(*X*))) + 1

Properietà AVL

ABR dove per ogni nodo *X*, l'*altezza* del *suttoalbero sinistro differisce* da quella del *destro* al massimo di 1.



La *Proprietà* AVL *non* è *soddisfatta* dal *nodo* 6.

No! Questo NON è un albero AVL.

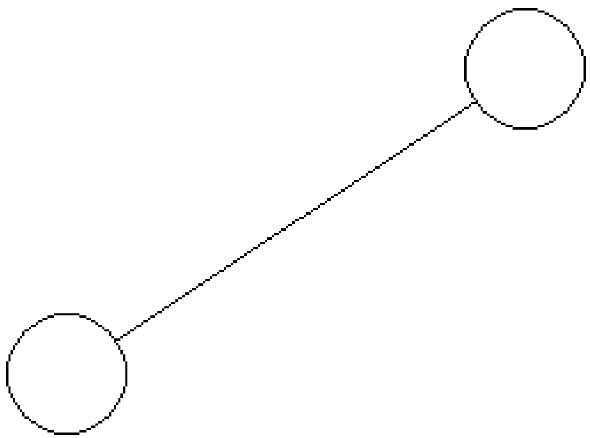
Alberi AVL: definizione

Definizione: **Un albero binario di ricerca è un *Albero AVL* se per ogni nodo *x*:**

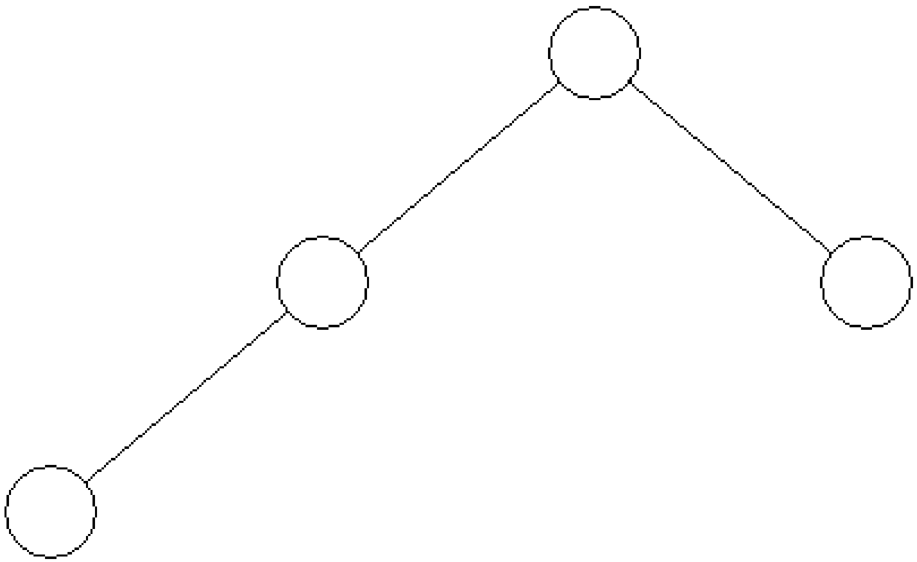
- **l'*altezza* del *sottoalbero sinistro di x* e quella del *sottoalbero destro di x* differiscono al più di uno, e**
- ***entrambi i sottoalberi* sinistro e destro di *x* sono *alberi AVL***

Esempi di alberi AVL minimi di diverse altezze (cioè con il numero minimo possibile di nodi)

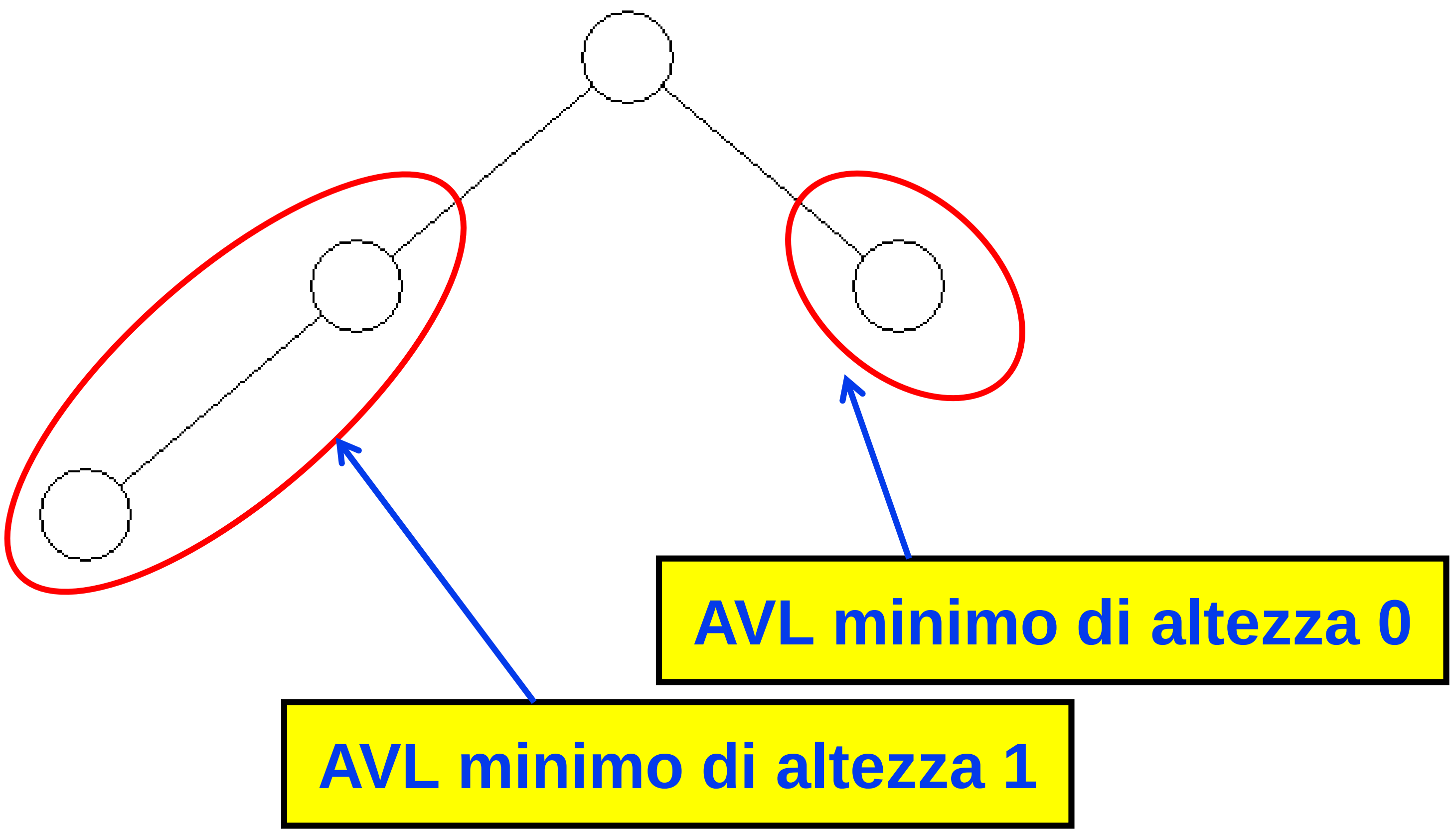
Albero AVL minimo di altezza 1



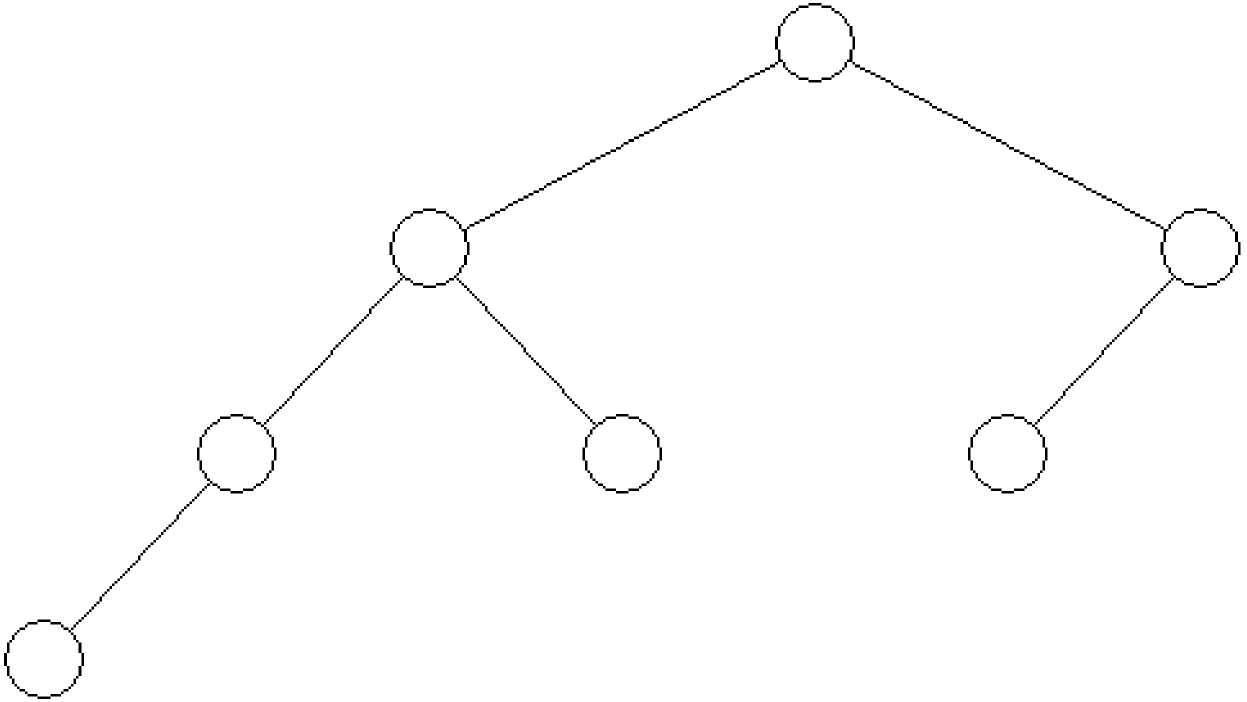
Albero AVL minimo di altezza 2



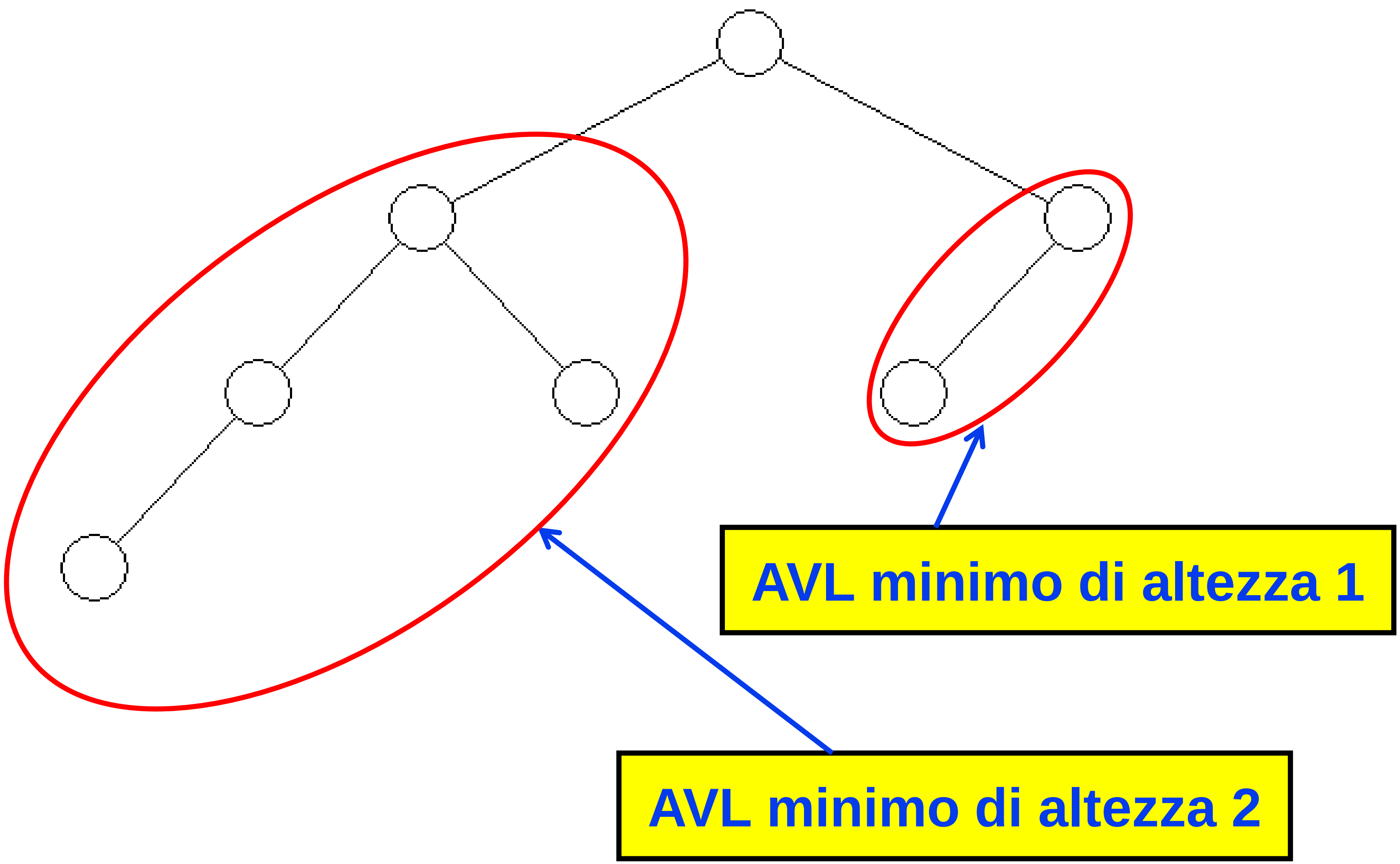
Albero AVL minimo di altezza 2



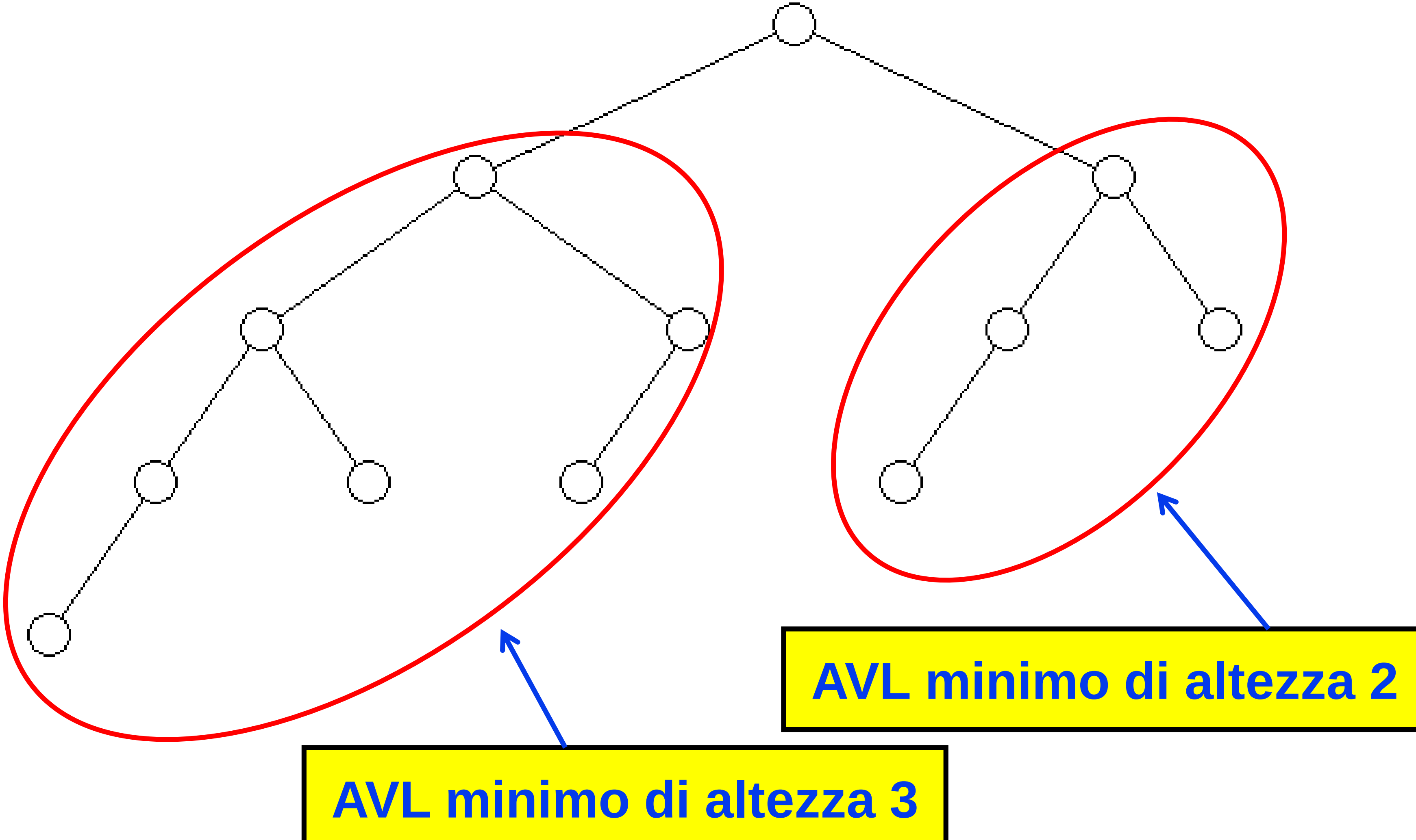
Albero AVL minimo di altezza 3



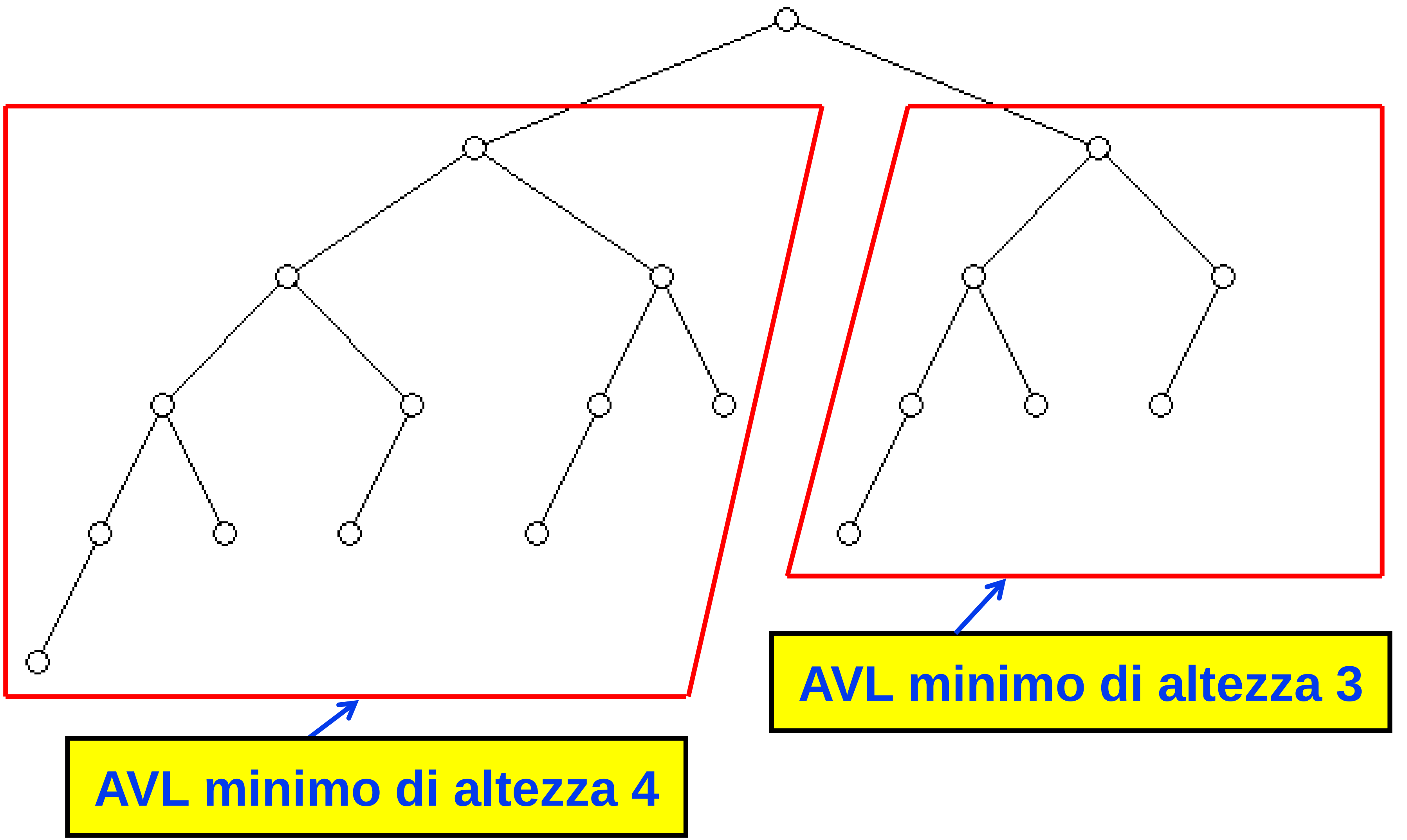
Albero AVL minimo di altezza 3



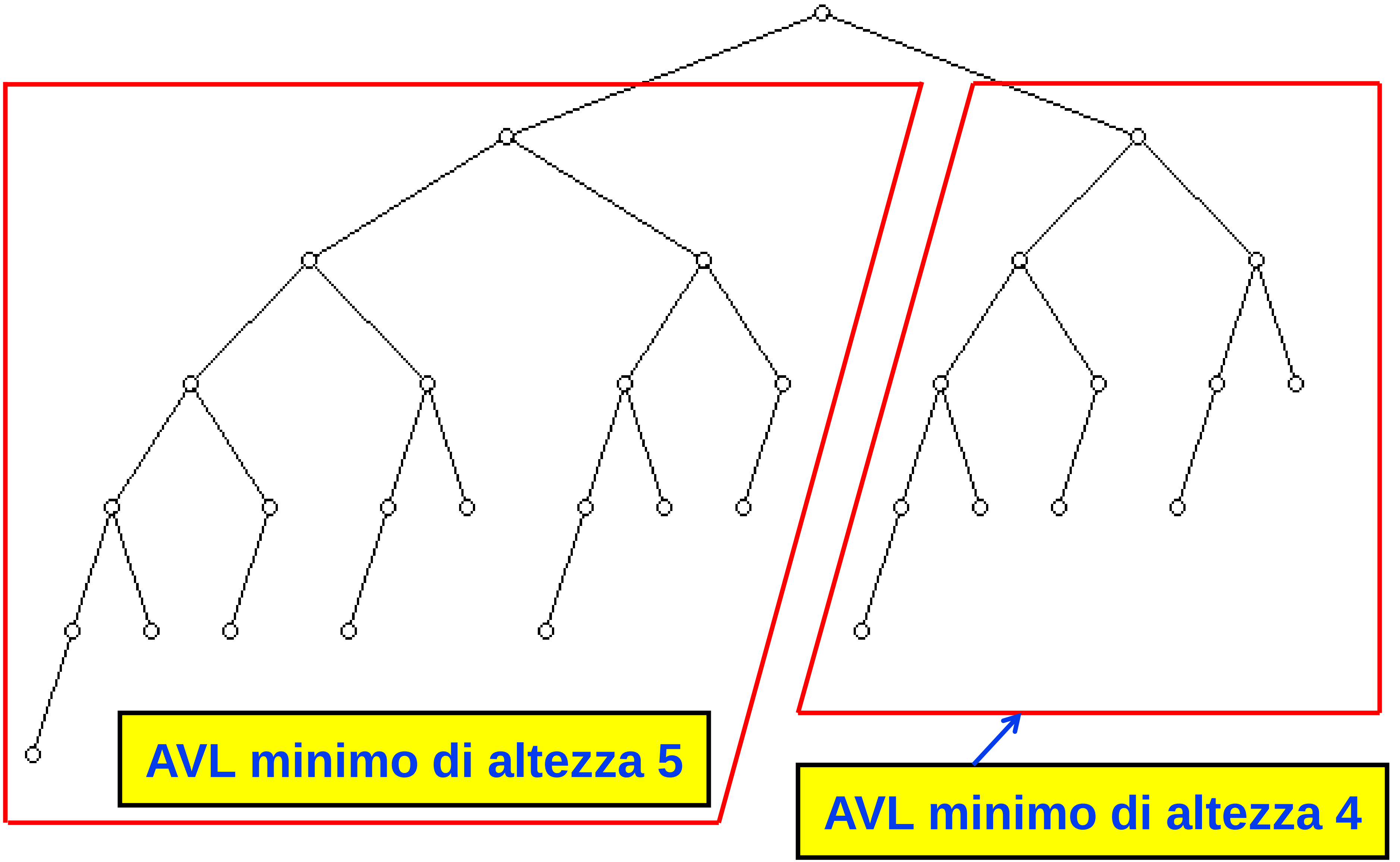
Albero AVL minimo di altezza 4



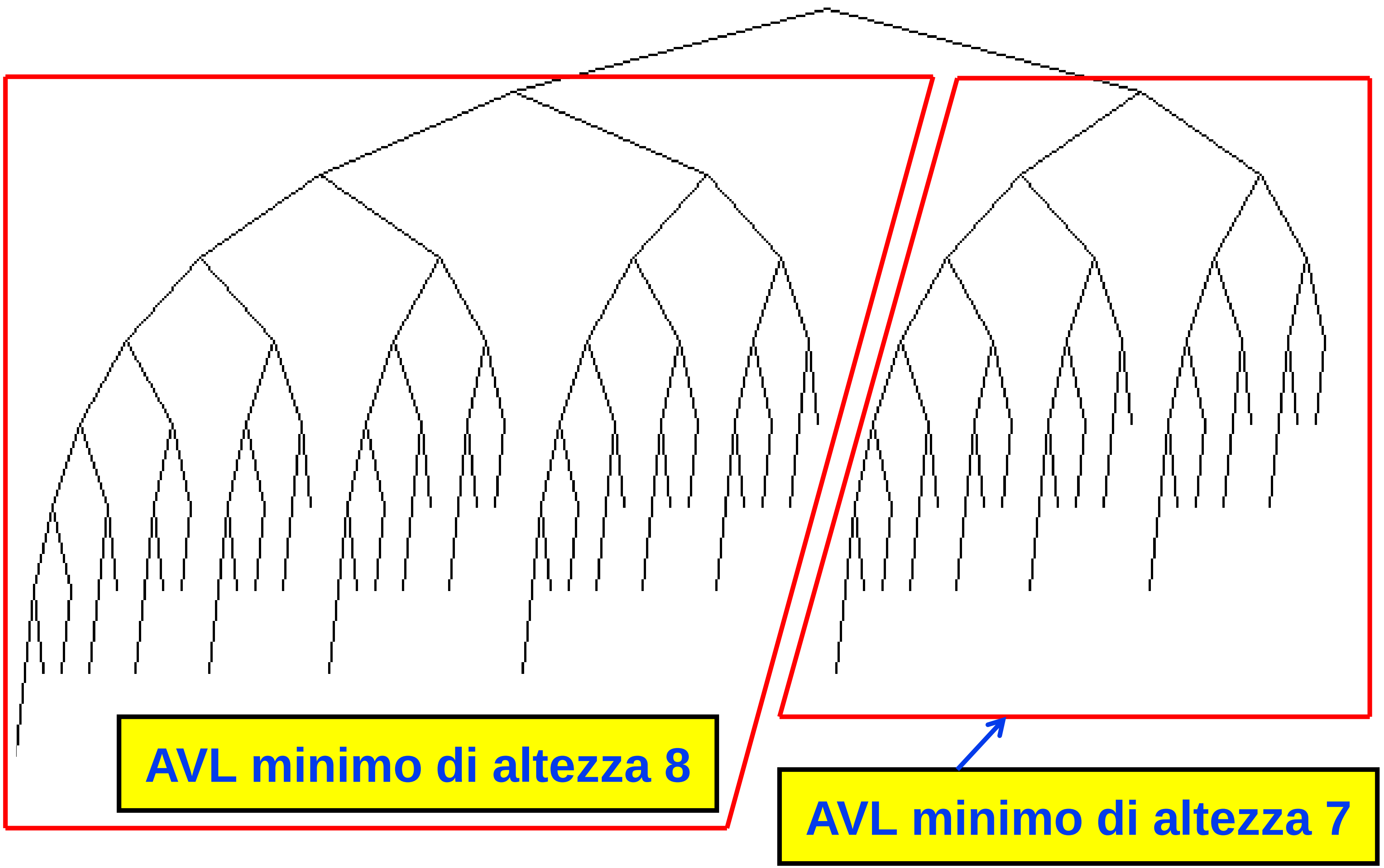
Albero AVL minimo di altezza 5



Albero AVL minimo di altezza 6



Albero AVL minimo di altezza 9



Proprietà degli alberi AVL

Dato un qualsiasi *albero AVL* con *n* nodi, si può dimostrare che la sua *altezza* è determinata dalla seguente formula:

$$\boxed{h \cong 1.44 \log(n+2) - 0.328}$$

Alberi Red-Black (alberi rossi-neri)

Un *Albero Red-Black* *(rosso-nero)* è essenzialmente un albero binario di ricerca in cui:

1. Le *chiavi* vengono mantenute *solo* nei *nodi interni* dell'albero

2. Le foglie sono costituite da speciali *nodi NIL*, cioè nodi *"sentinella"* il cui contenuto è *irrilevante* e che evitano di trattare diversamente i puntatori ai nodi dai puntatori *NIL*.
   - In altre parole, al posto di un puntatore *NIL* si usa un puntatore ad un nodo *NIL*.
   - Quando un nodo ha come figli nodi *NIL*, quel nodo sarebbe una foglia nell'albero binario di ricerca corrispondente.

Alberi Red-Black (alberi rossi-neri)

Un *Albero Red-Black* *(rosso-nero)* deve soddisfare le seguenti proprietà (vincoli):

1. ***Ogni nodo* è colorato o di *rosso* o di *nero*;**
2. **Per convezione, i *nodi NIL* si considerano *nodi neri*;**
3. **Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;**
4. **Ogni percorso da un *nodo interno* ad un *nodo NIL* (figlio di una foglia) ha lo *stesso numero* di *nodi neri*;**

Alberi Red-Black (alberi rossi-neri)

Un *Albero Red-Black* *(rosso-nero)* deve soddisfare le seguenti proprietà (vincoli):

1. ***Ogni nodo* è colorato o di *rosso* o di *nero*;**

2. **Per convezione, i *nodi NIL* si considerano *nodi neri*;**

3. **Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;**

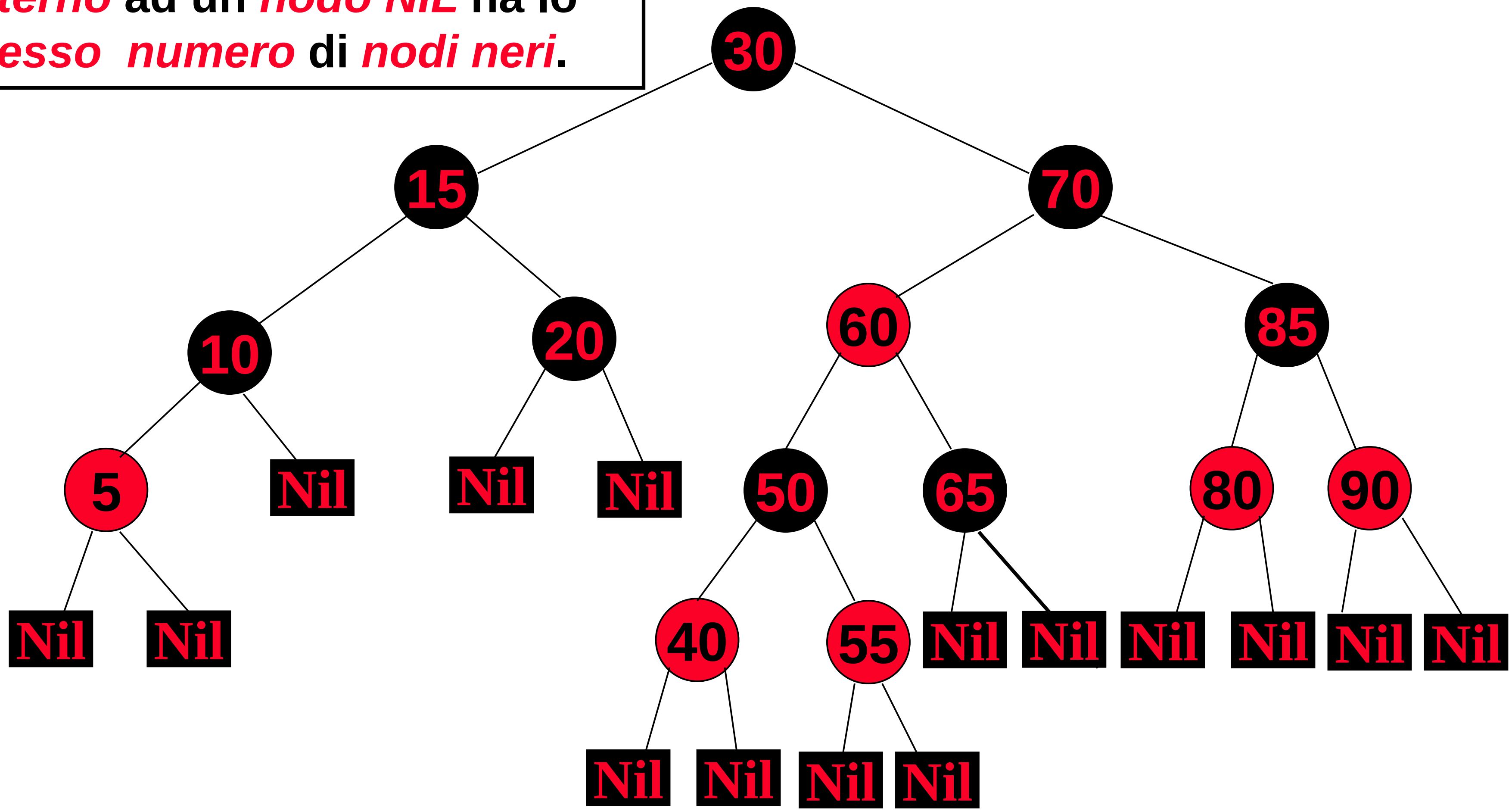
4. **Ogni percorso da un *nodo interno* ad un *nodo NIL* (figlio di una foglia) ha lo *stesso numero* di *nodi neri*;**

Considereremo solo *alberi Red-Black* in cui la *radice è nera*.

Alberi Red-Black: esempio I

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;

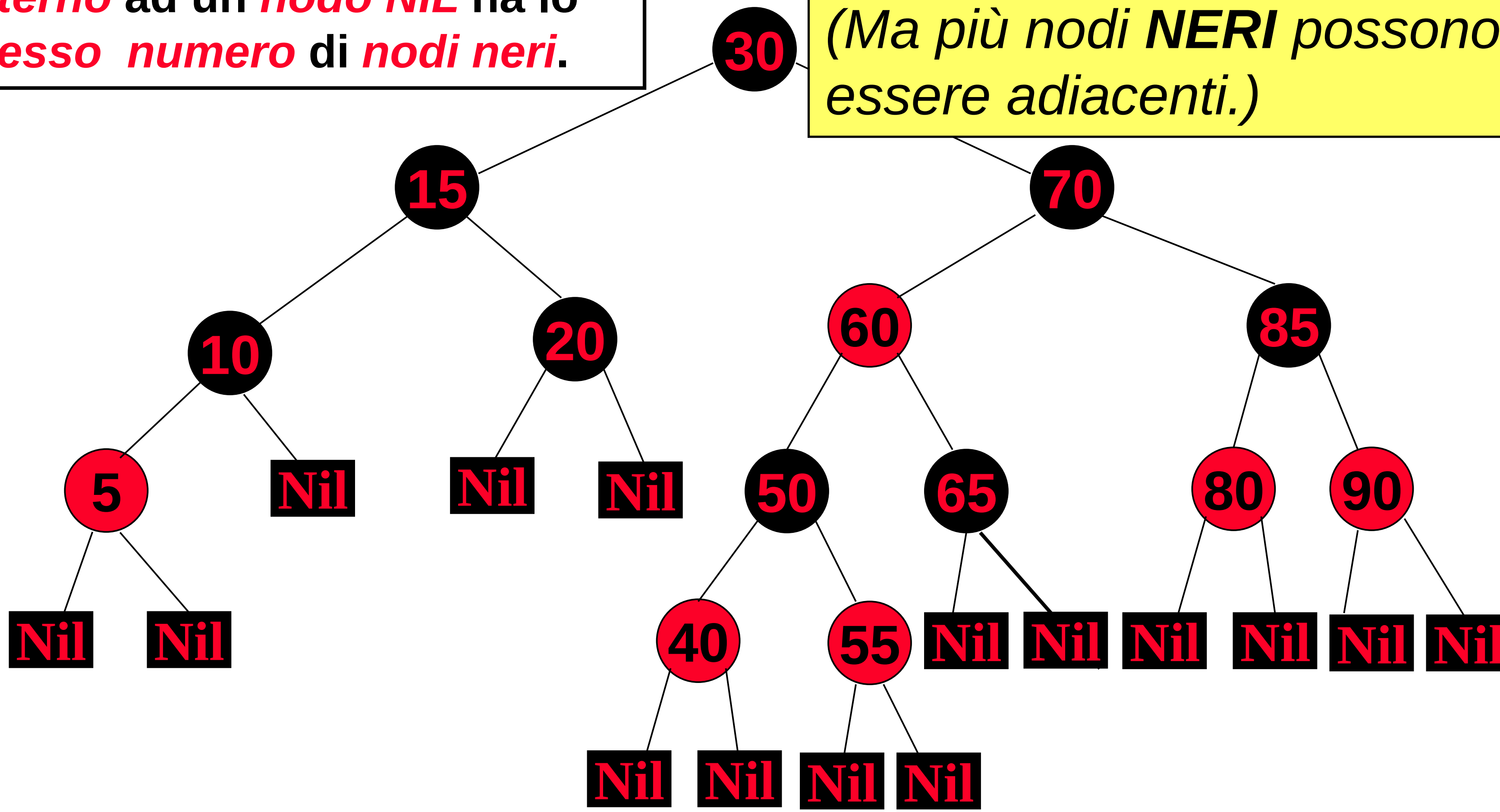
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.



Alberi Red-Black: esempio I

3 **Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;**
4 **Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.**

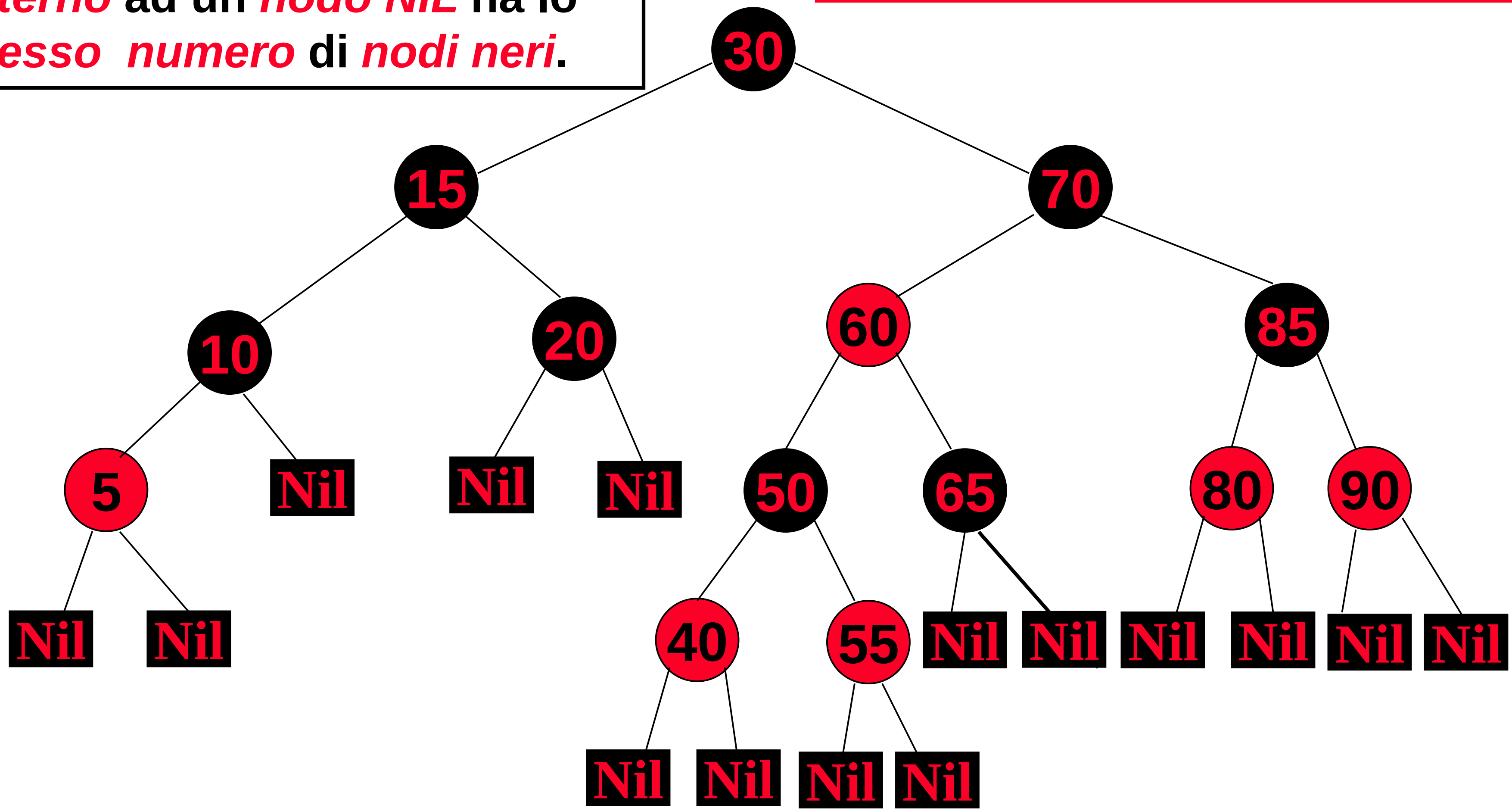
***Vincolo 3* impone che *non* possano esserci *due nodi ROSSI adiacenti*.**
*(Ma più nodi **NERI** possono essere adiacenti.)*



Alberi Red-Black: esempio I

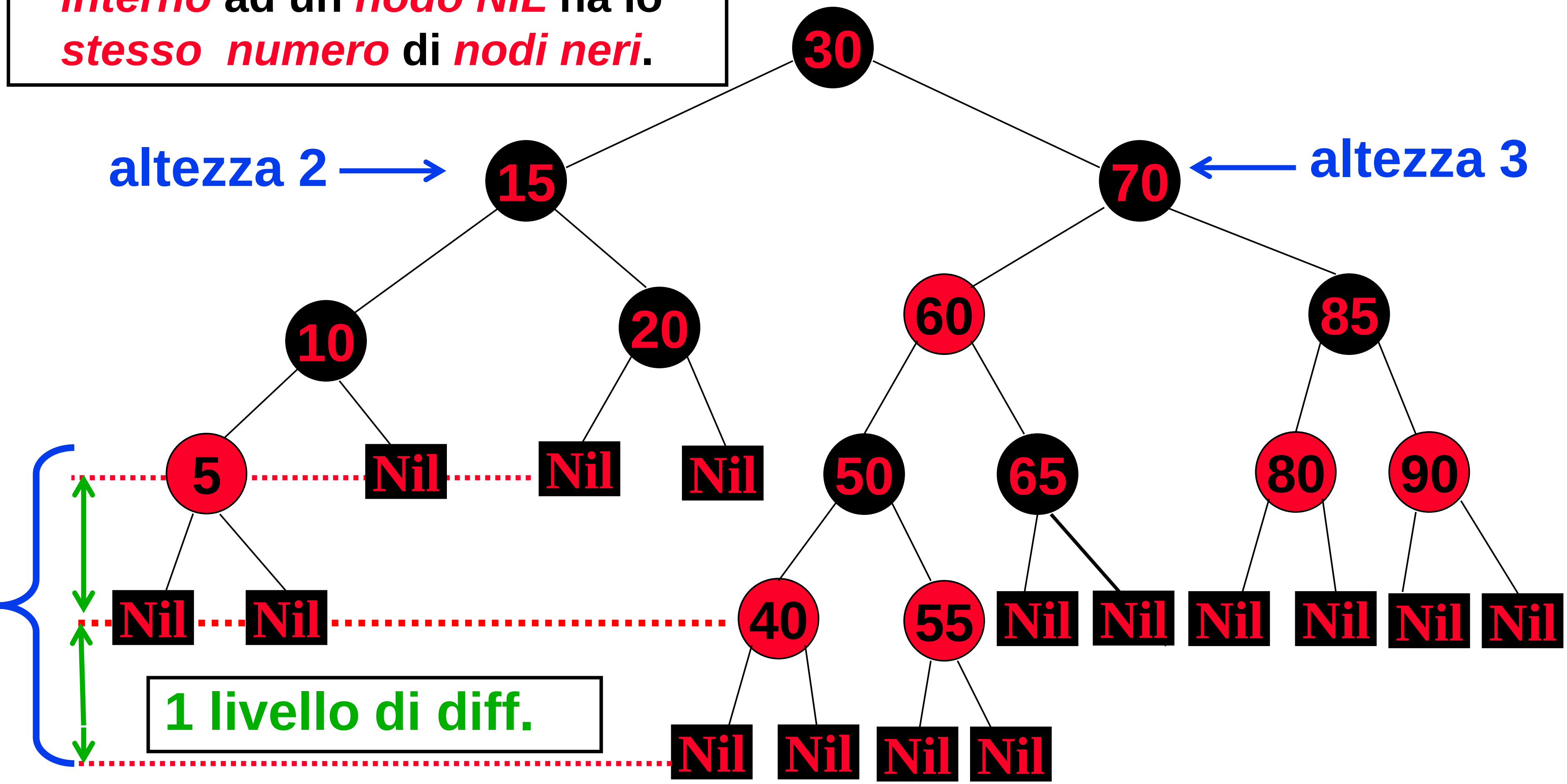
3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Ci sono *3 nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



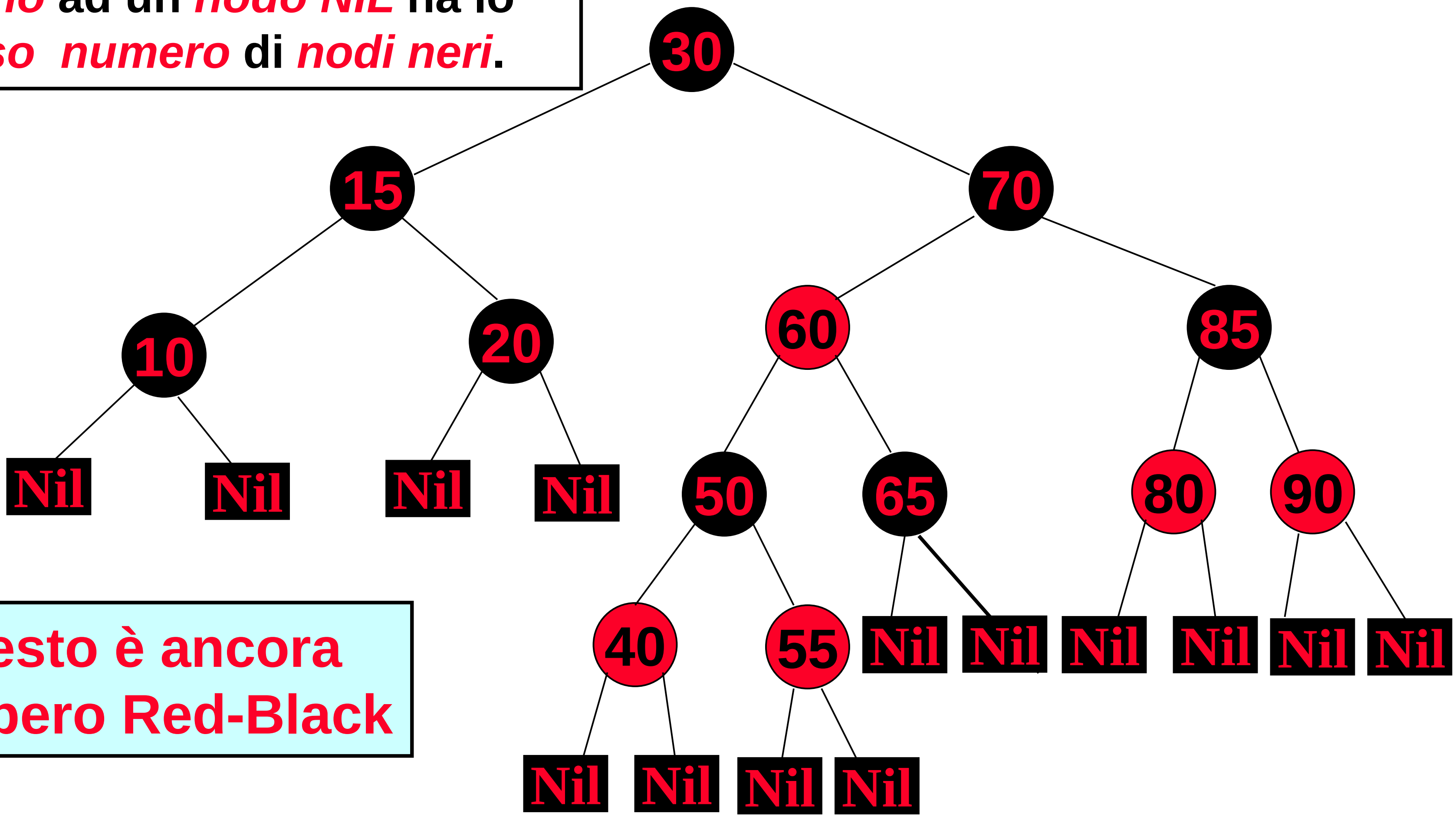
Alberi Red-Black: esempio I

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.



Alberi Red-Black: esempio II

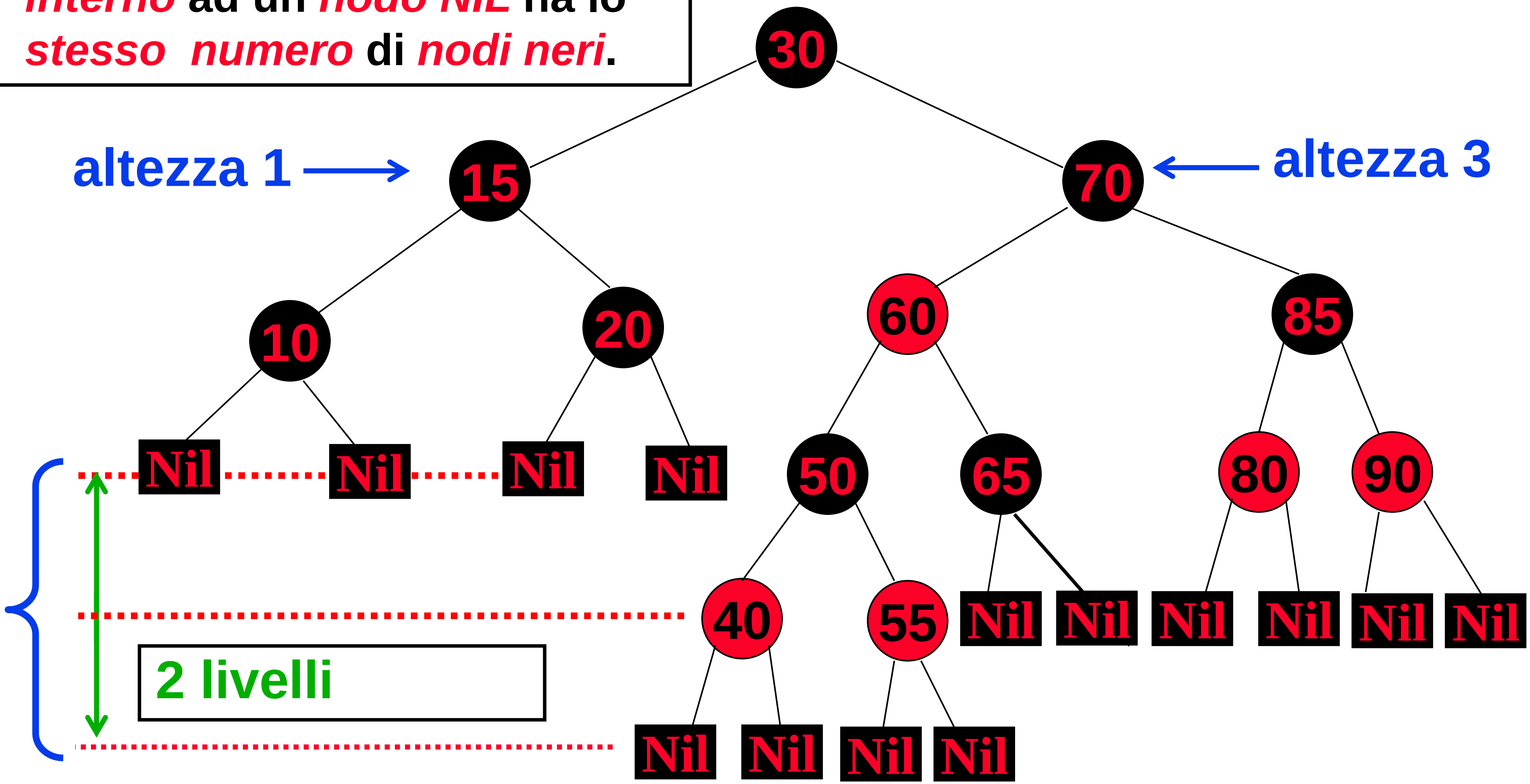
3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.



Questo è ancora un albero Red-Black

Alberi Red-Black: esempio II

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

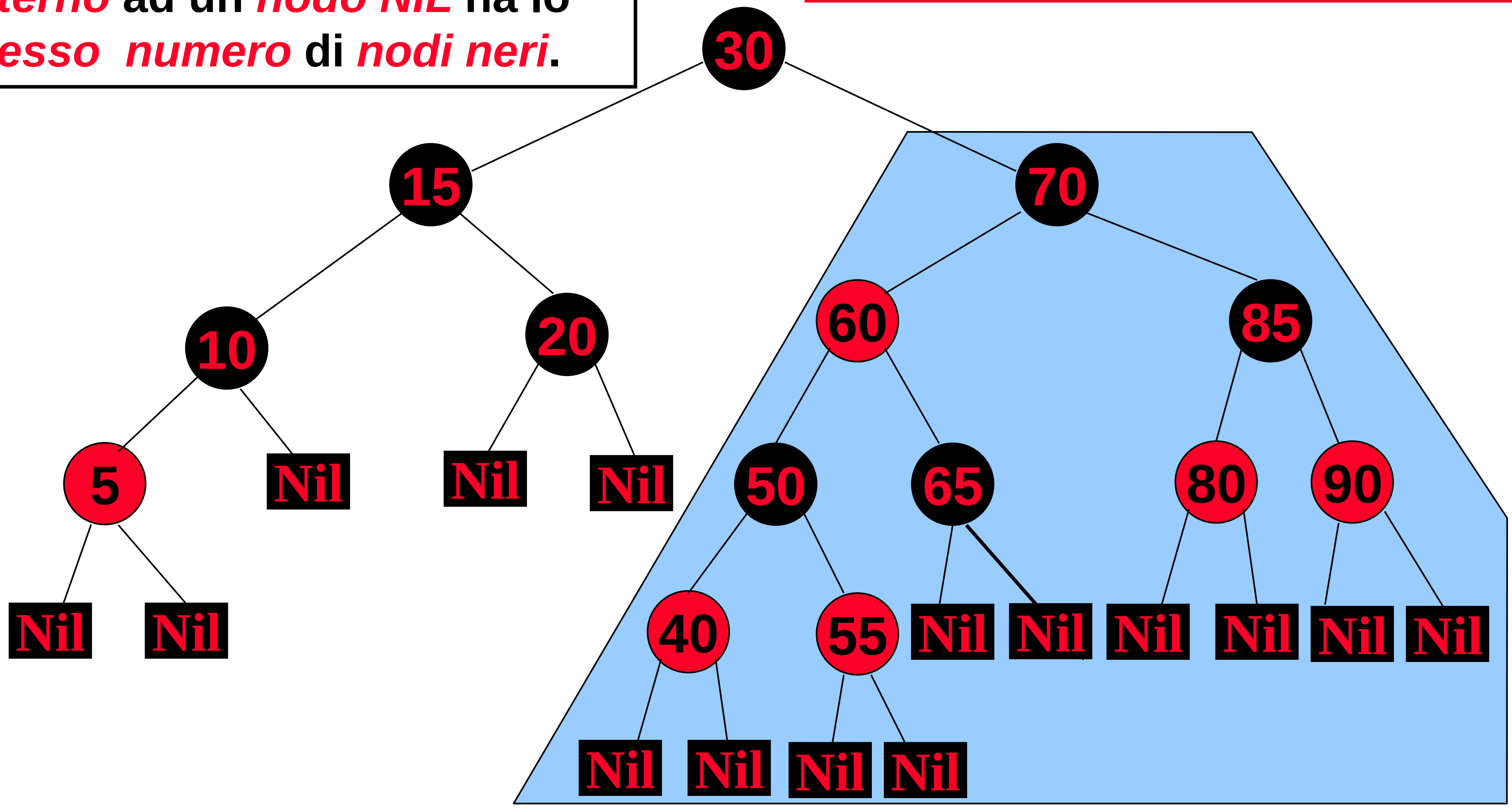


Alberi Red-Black: esempio I

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;

4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

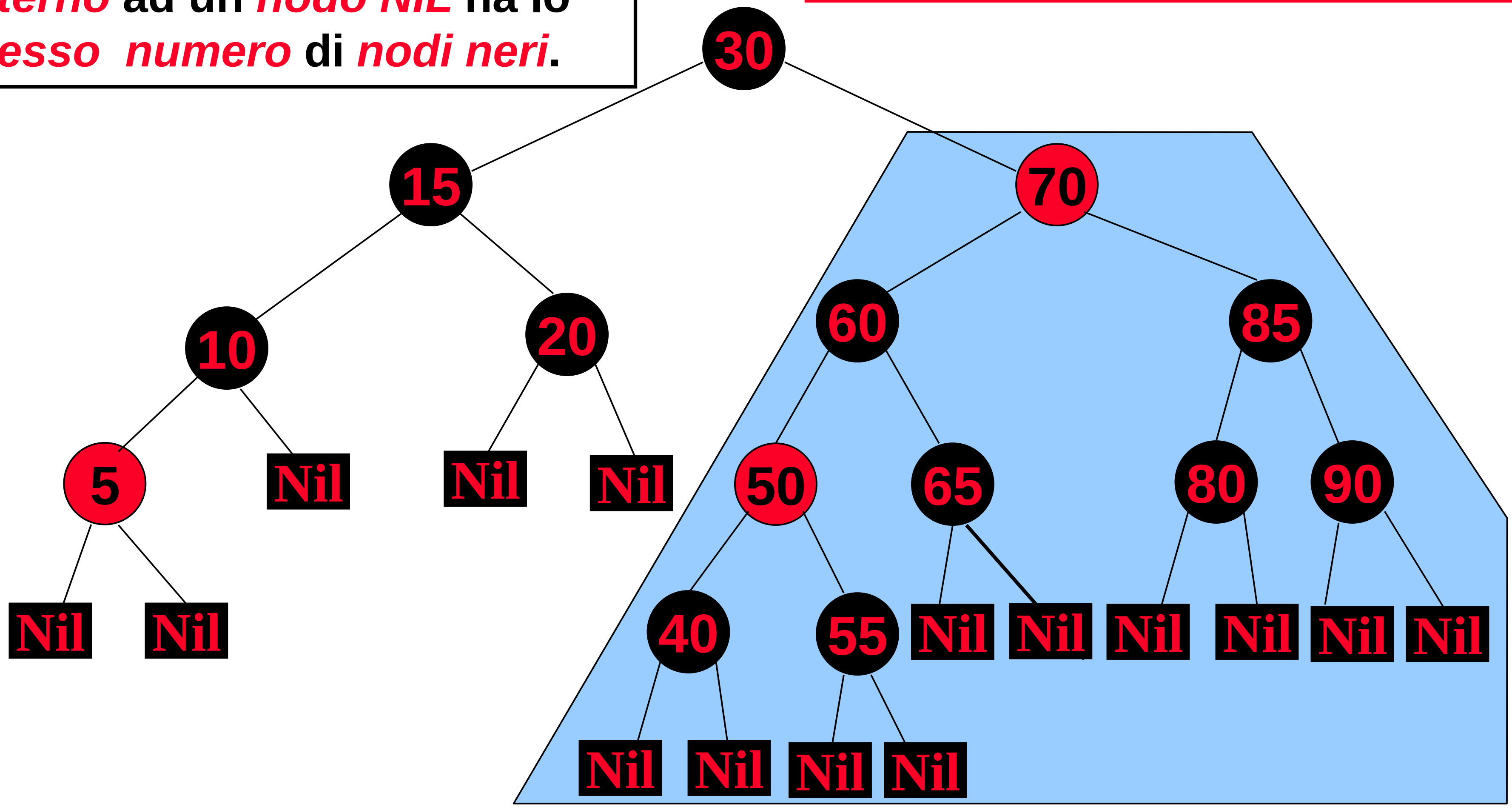
Ci sono *3 nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio III

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

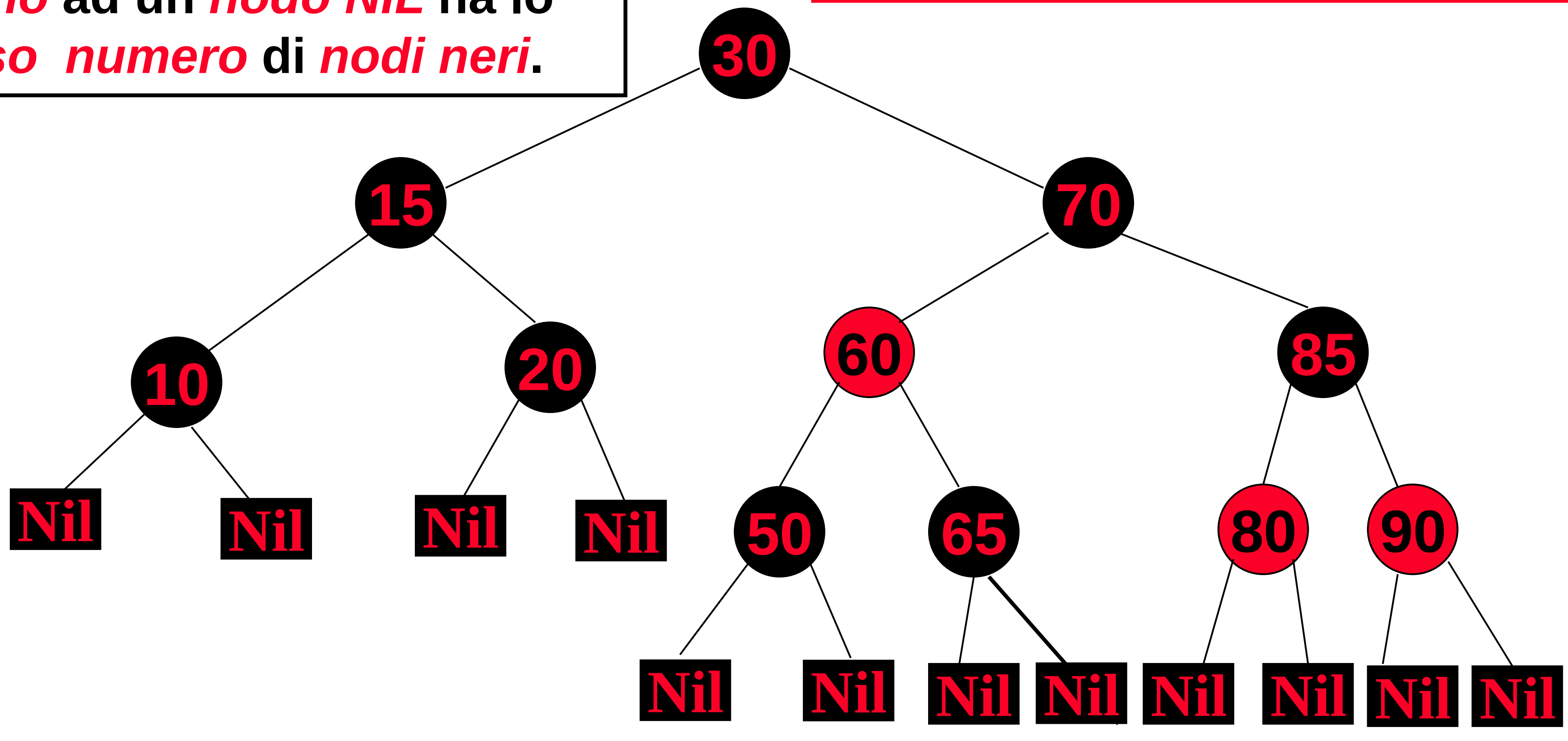
Ci sono *3 nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio IV

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Albero RB con *3 nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*

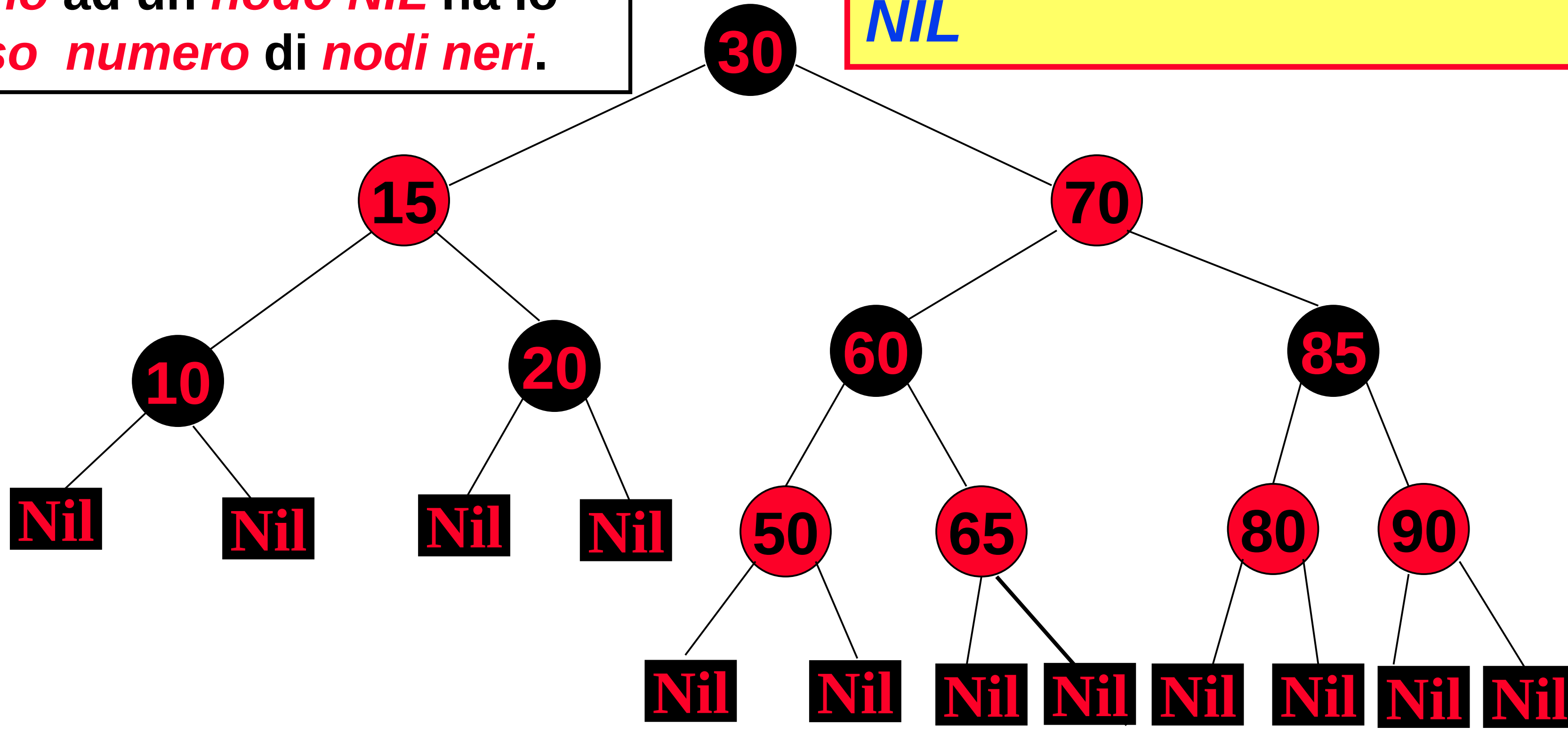


Alberi Red-Black: esempio V

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;

4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Stesso albero con *2 nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*

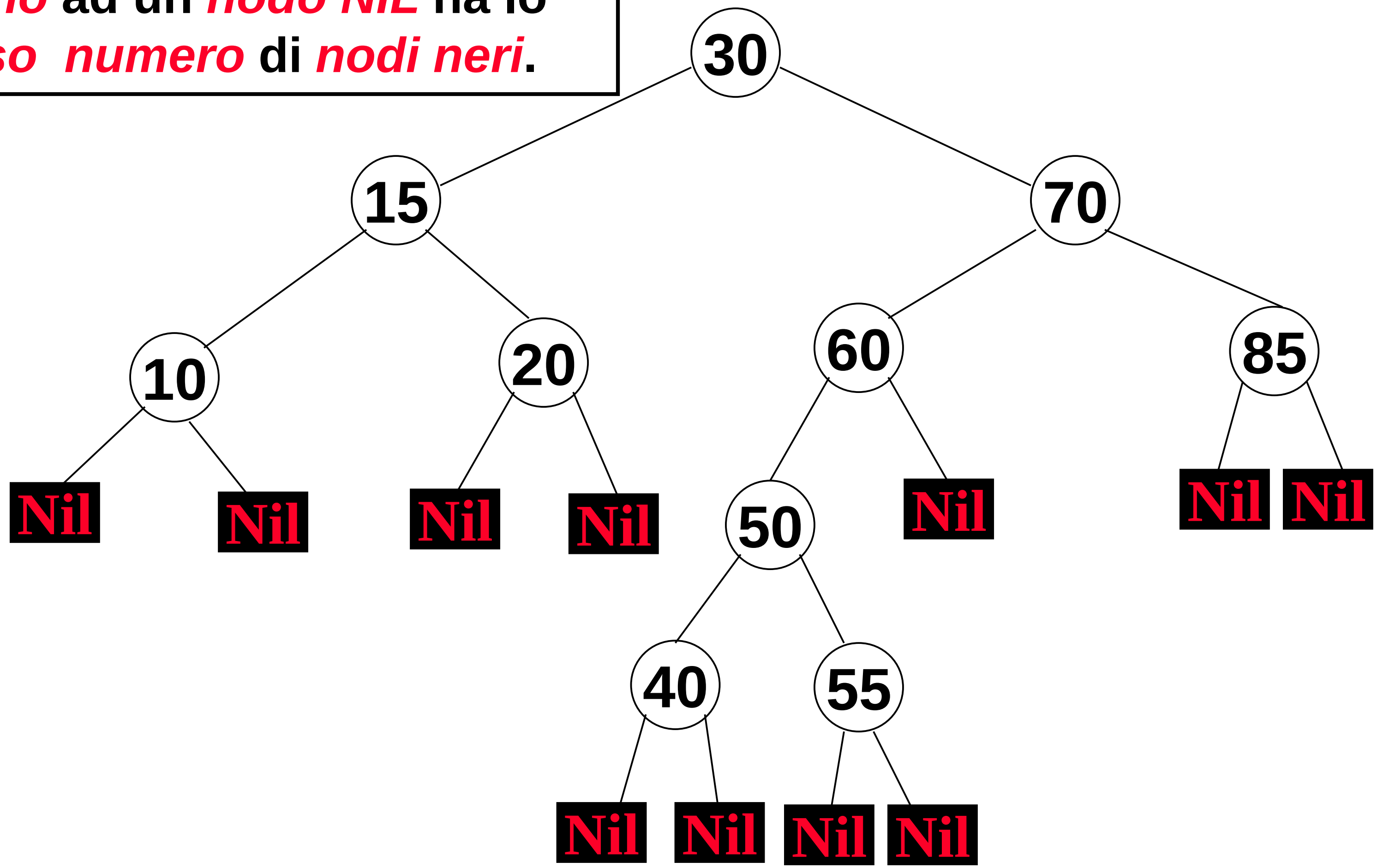


Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;

4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

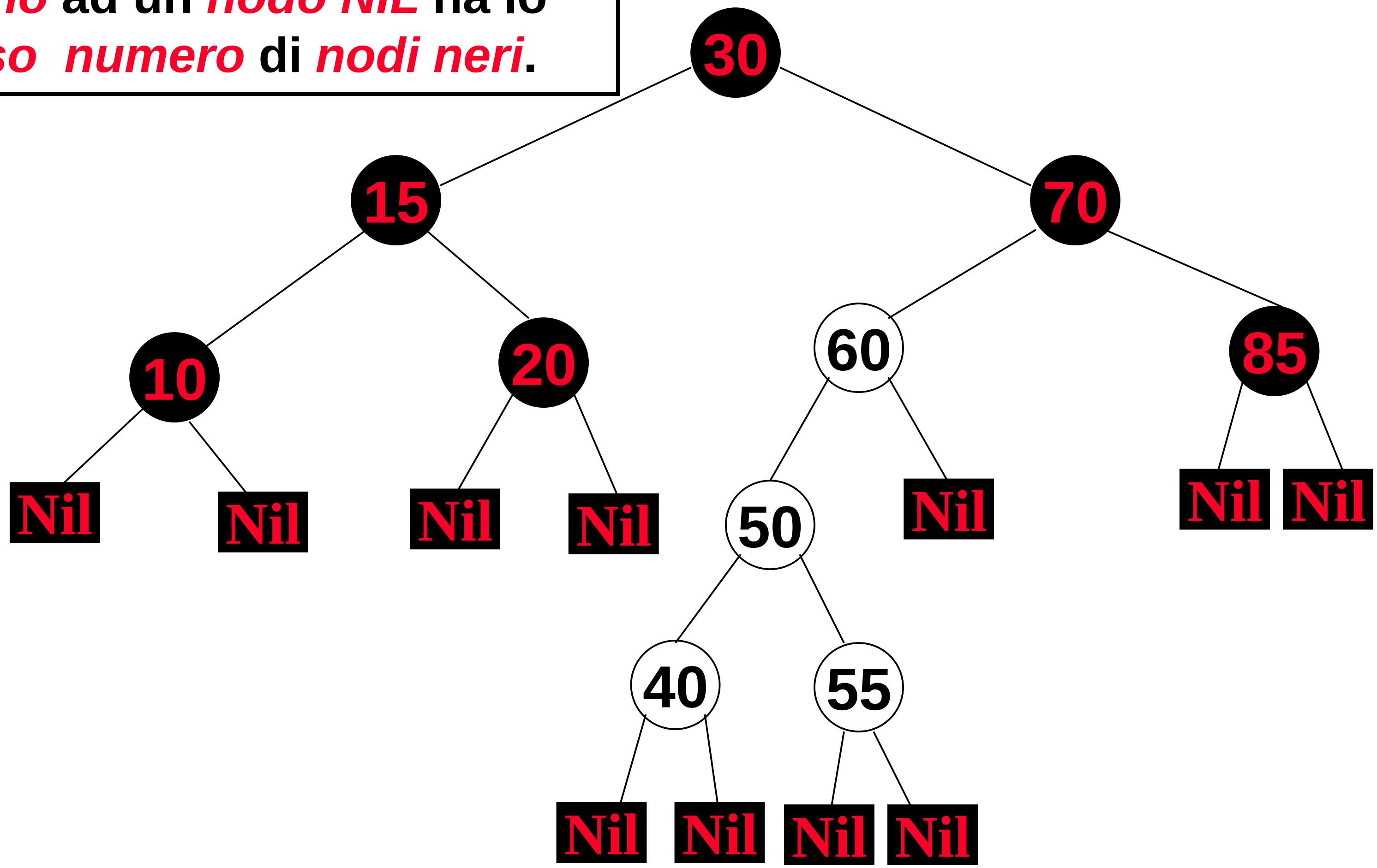
Questo albero è un albero Red-Black?



Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

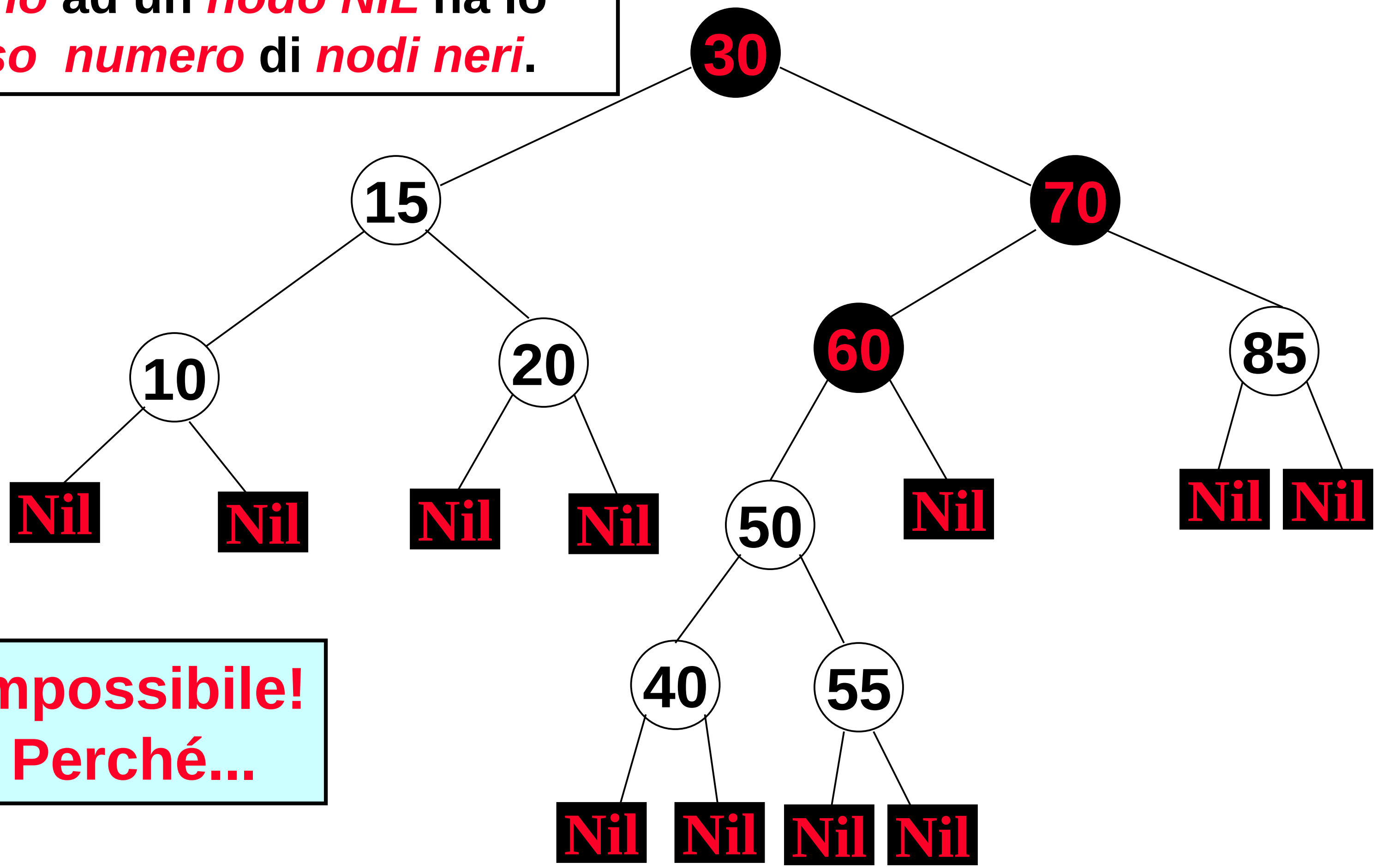
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!

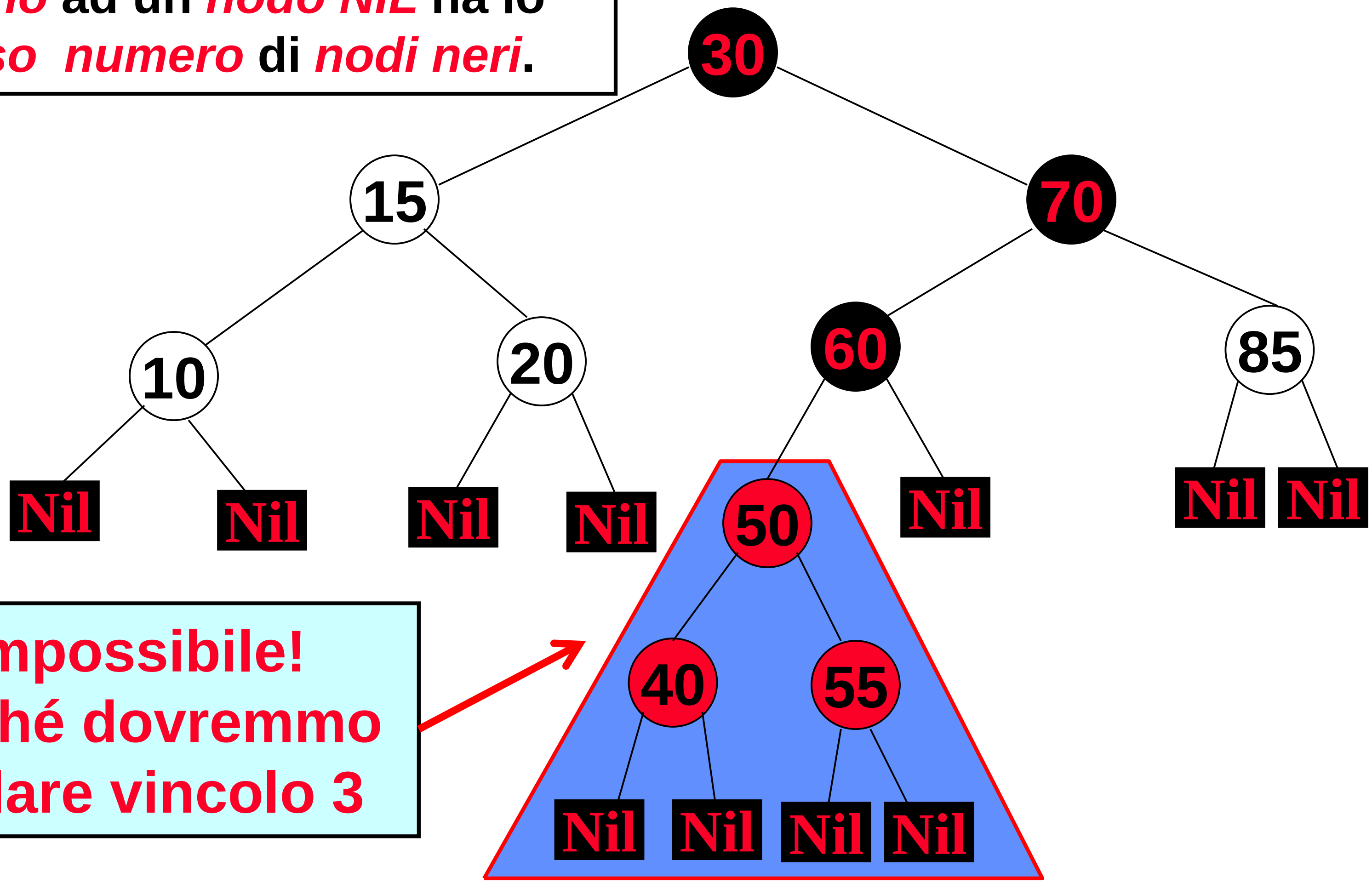


Impossibile!
Perché...

Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!

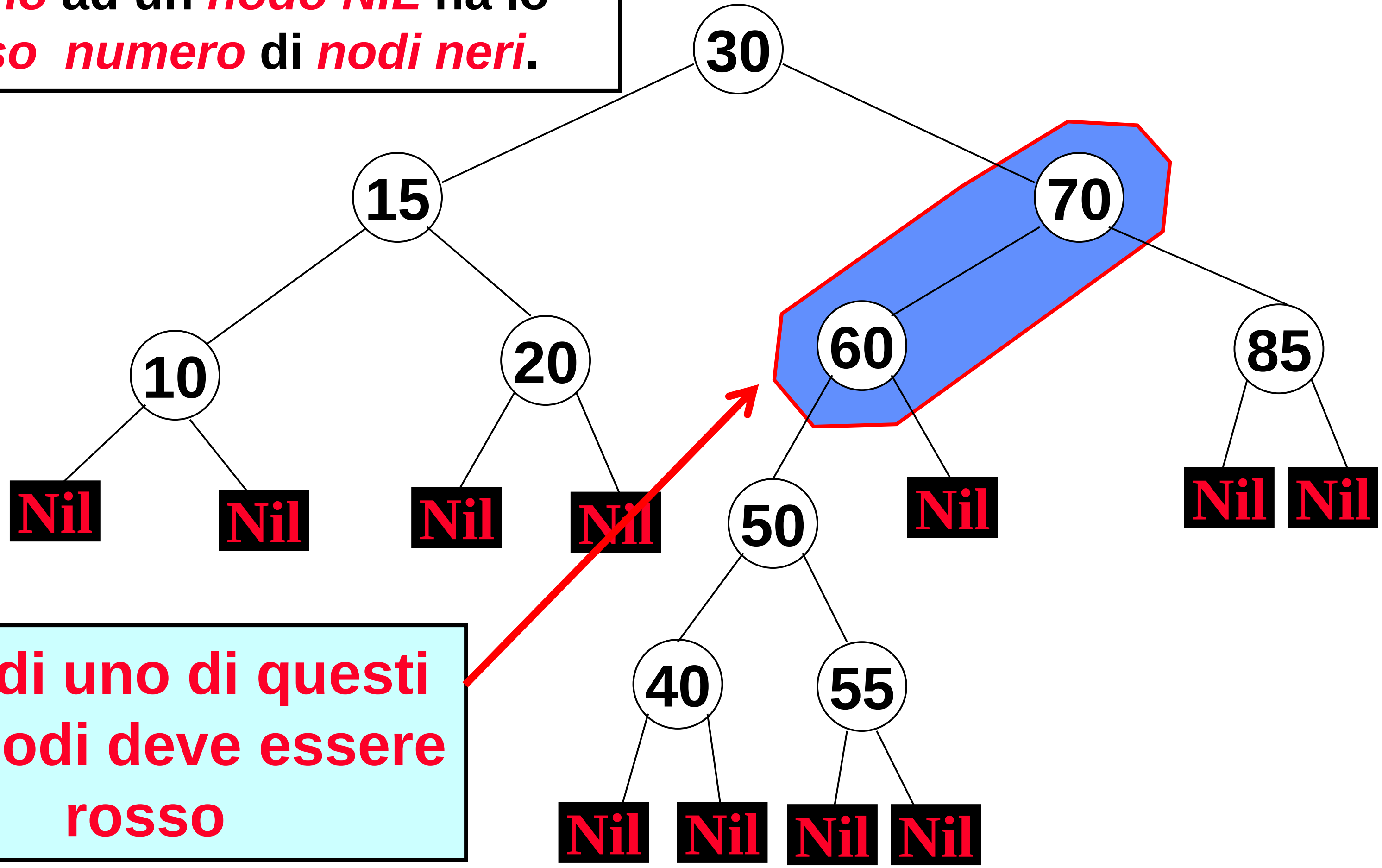


Impossibile!
Perché dovremmo violare vincolo 3

Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!

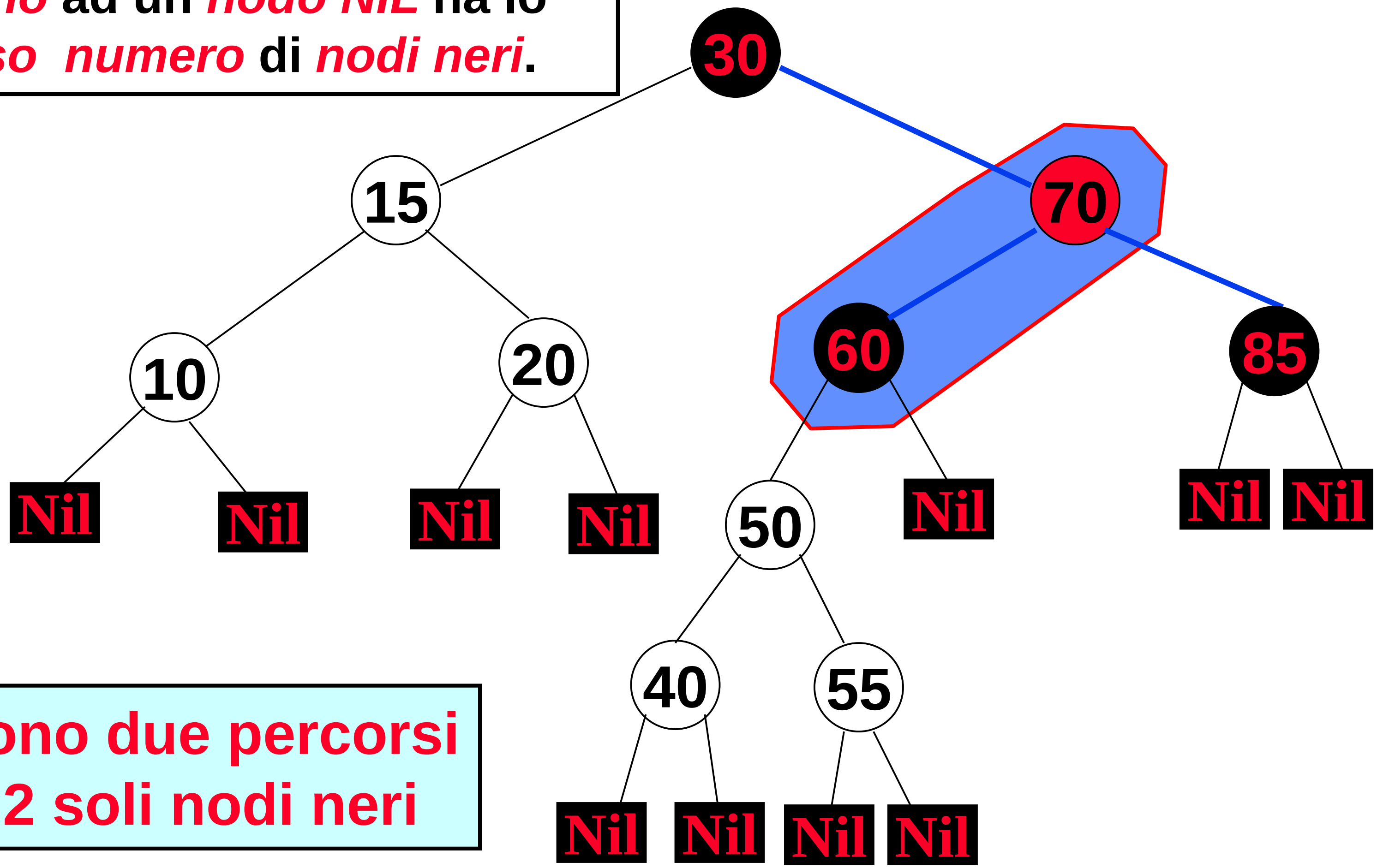


Quindi uno di questi due nodi deve essere rosso

Alberi Red-Black: esempio VI

3 **Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;**
4 **Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.**

Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!

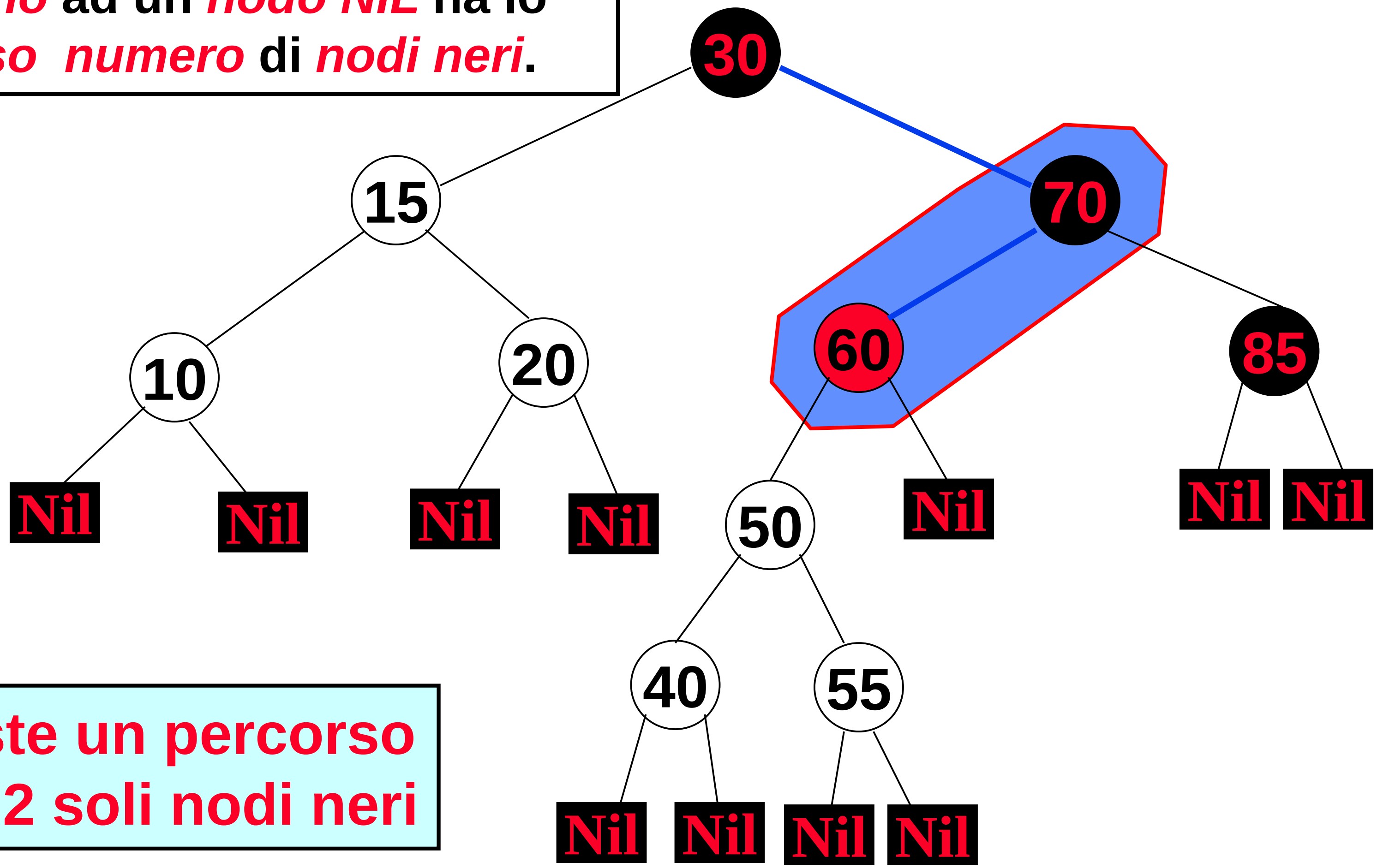


Esistono due percorsi con 2 soli nodi neri

Alberi Red-Black: esempio VI

3 **Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;**
4 **Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.**

Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!

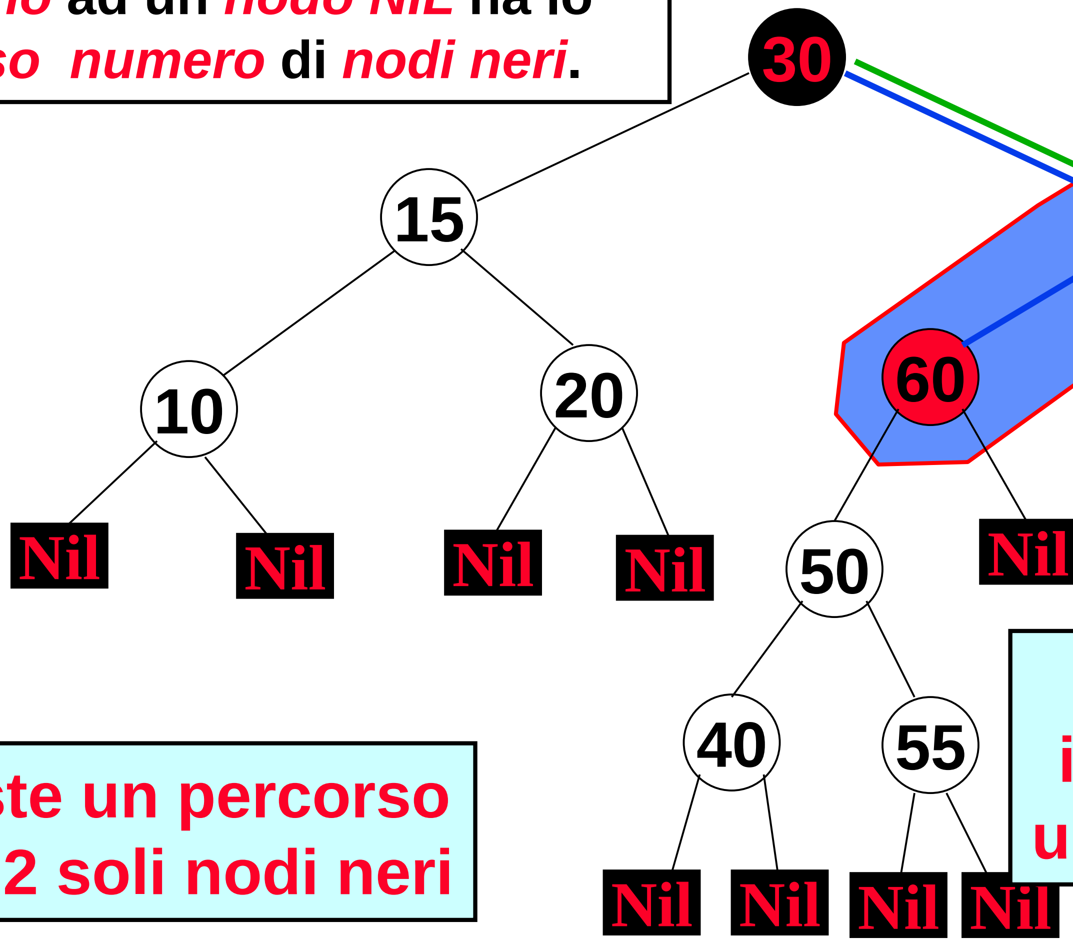


Esiste un percorso con 2 soli nodi neri

Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



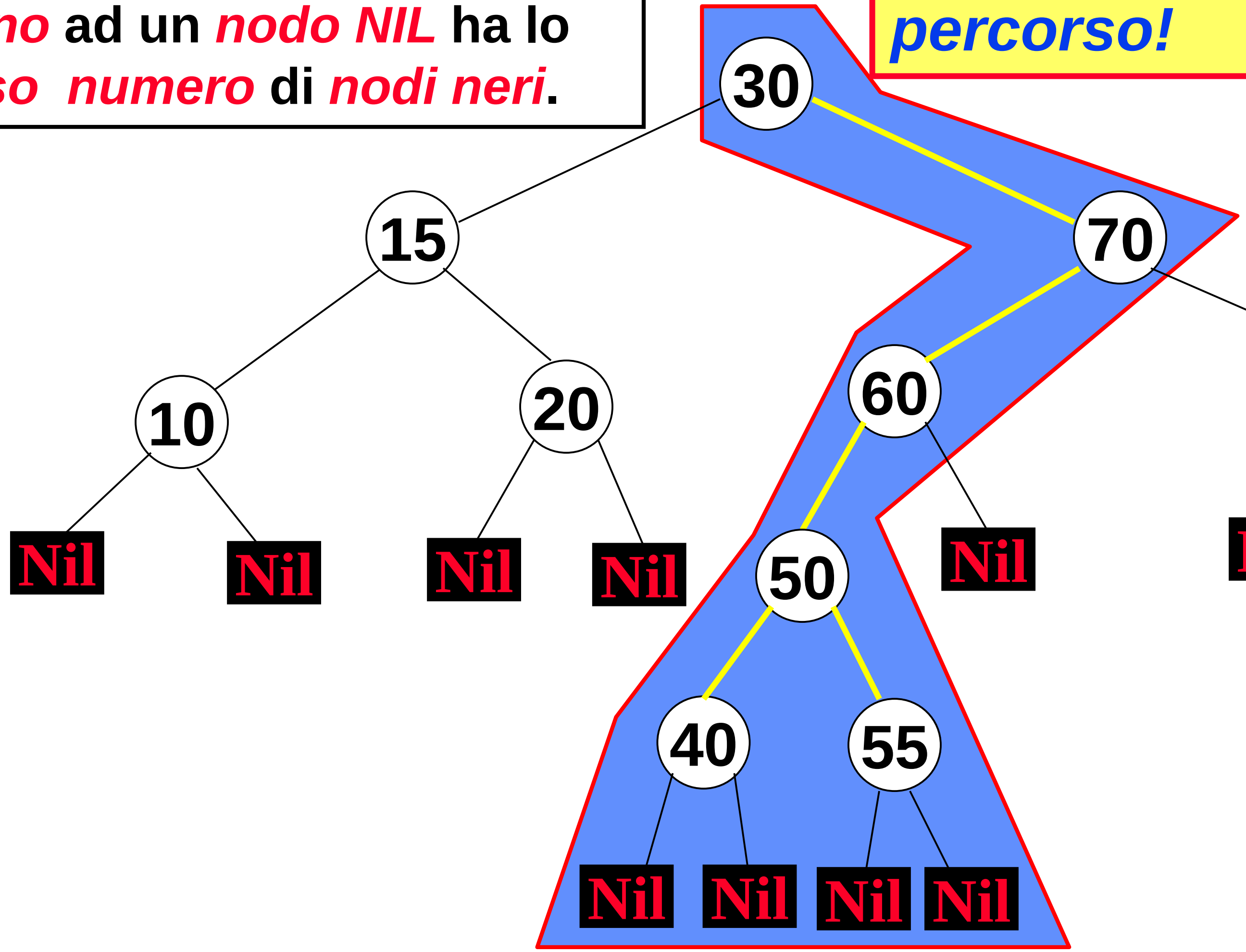
Esiste un percorso con 2 soli nodi neri

Questo caso è impossibile perchè un percorso ha 3 neri

Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Per vincolo 4 e il vincolo 3, ci possono essere <u>al più 2 nodi neri</u> lungo un percorso!

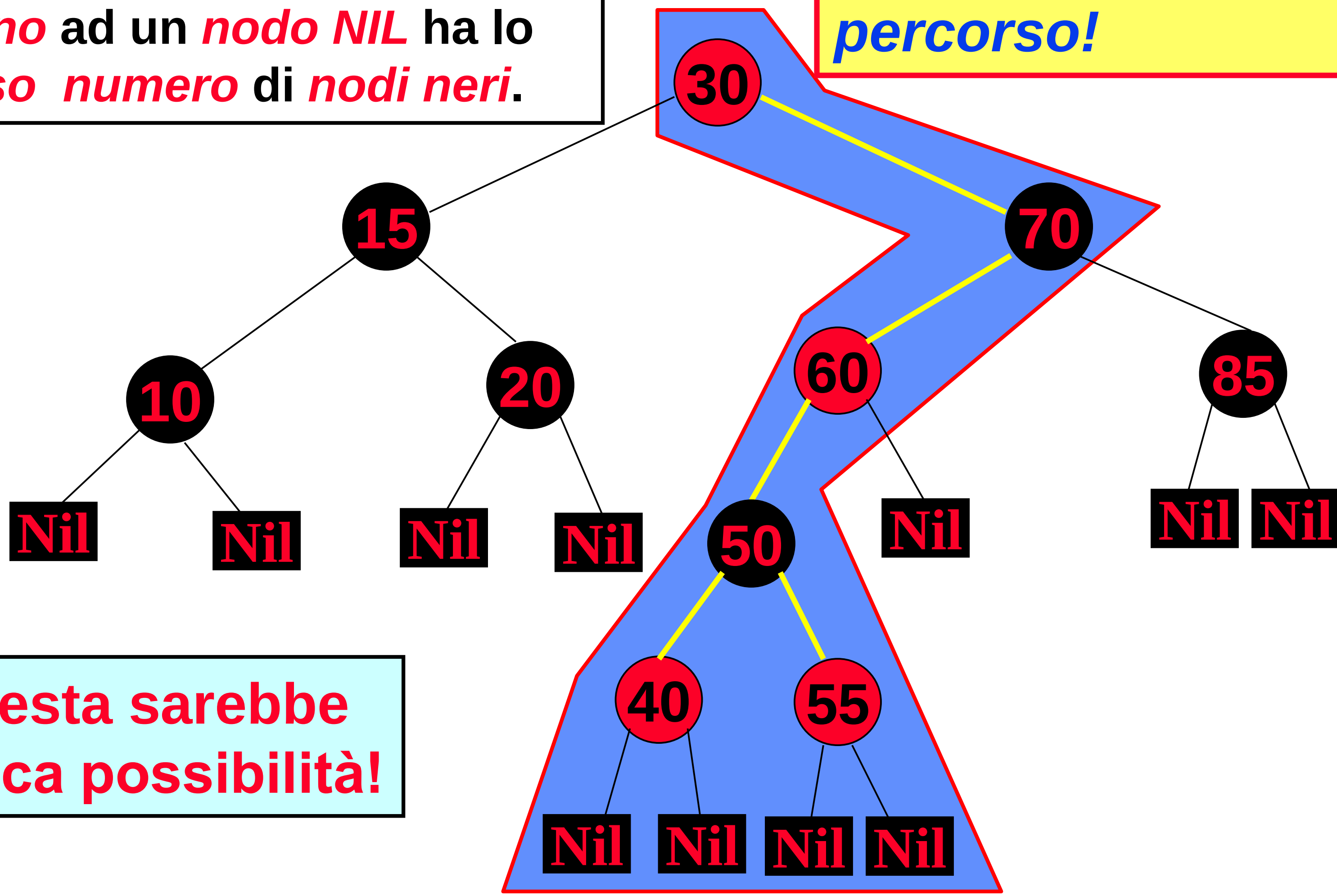


Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;

4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Per vincolo 4 e il vincolo 3, ci possono essere <u>al più 2 nodi neri</u> lungo un percorso!

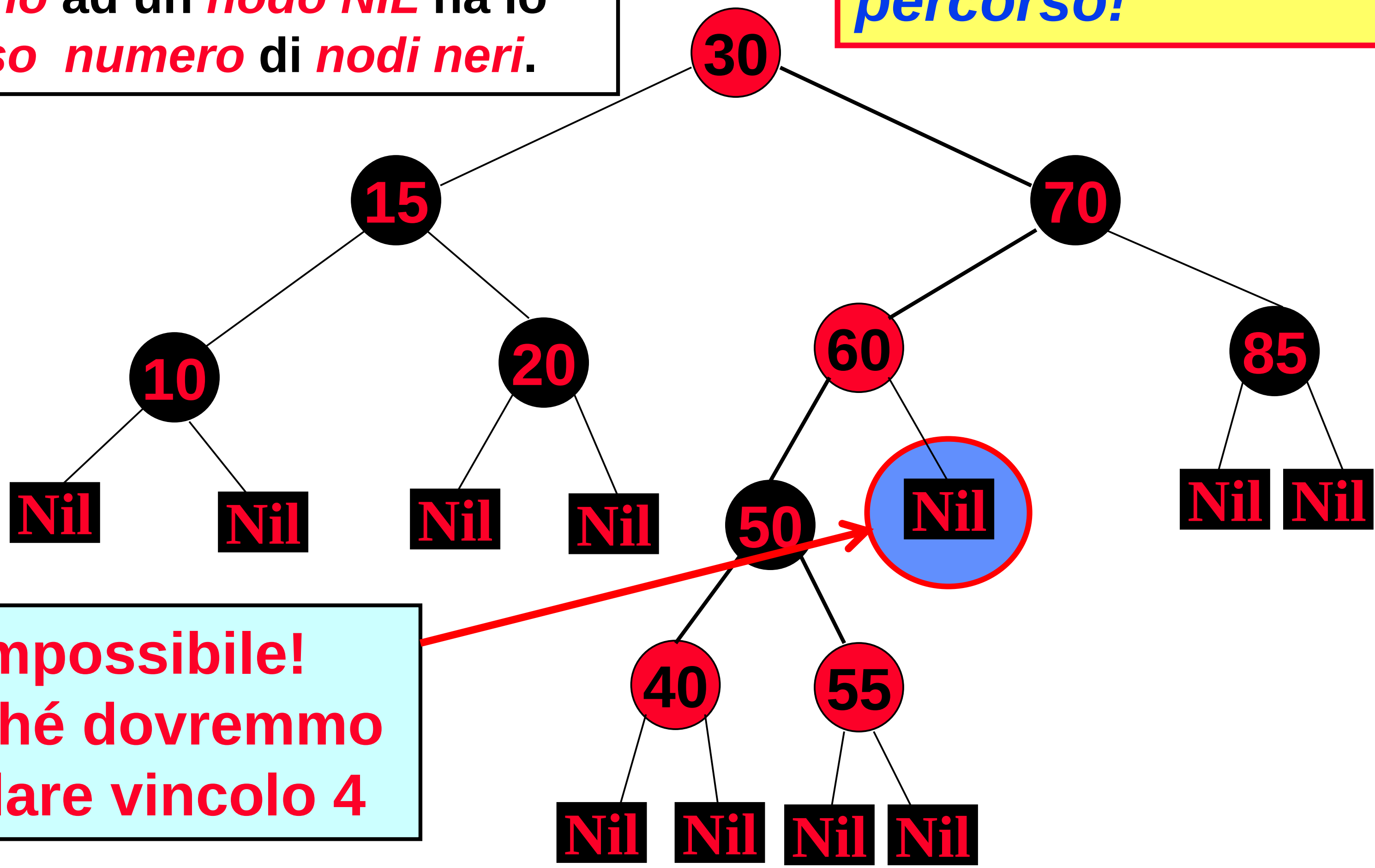


Questa sarebbe l'unica possibilità!

Alberi Red-Black: esempio VI

3 **Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;**
4 **Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.**

Per vincolo 4 e il vincolo 3, ci possono essere <u>al più 2 nodi neri</u> lungo un percorso!

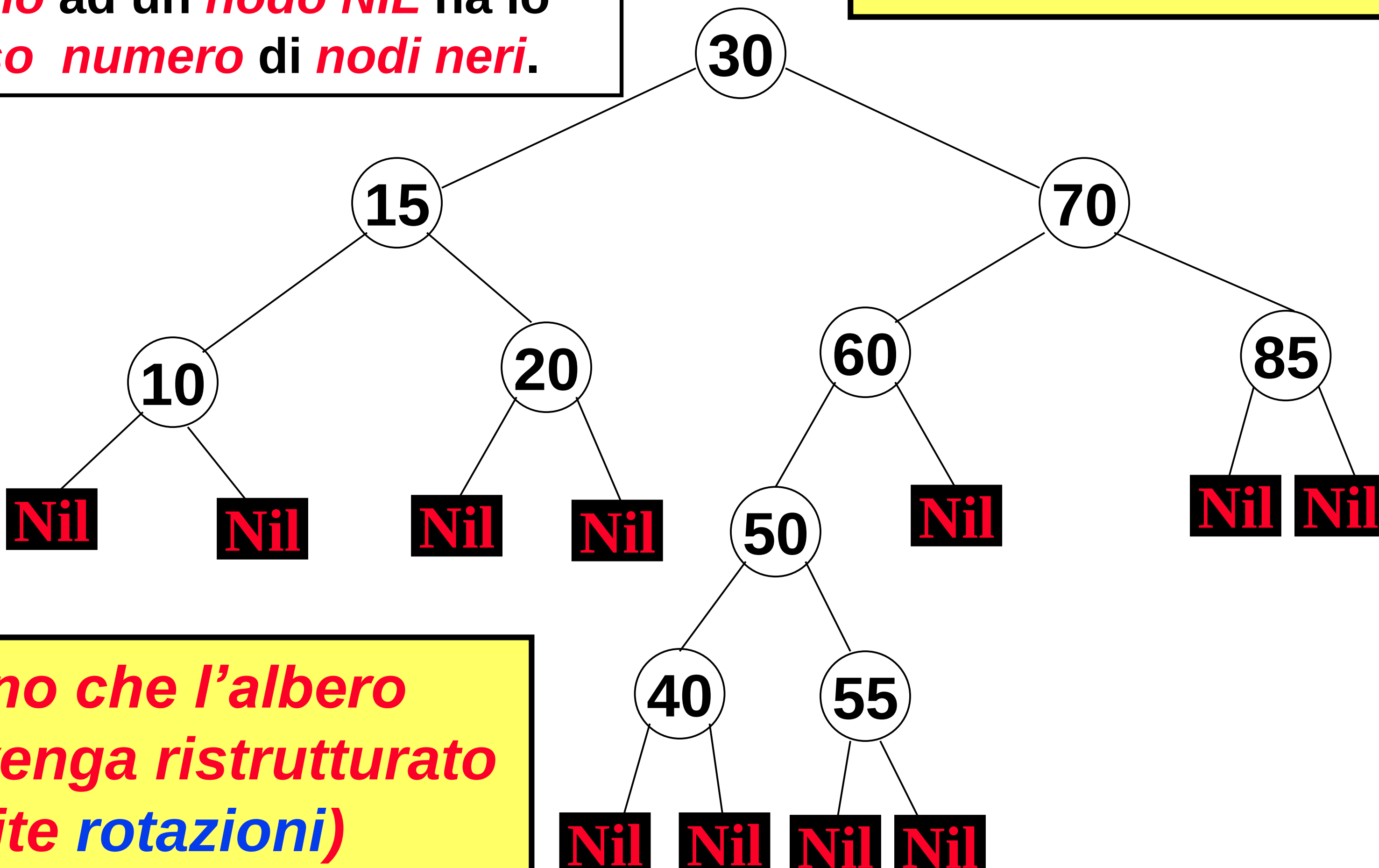


Impossibile!
Perché dovremmo violare vincolo 4

Alberi Red-Black: esempio VI

3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo *stesso numero* di *nodi neri*.

Questo albero NON può essere un albero Red-Black!



A meno che l'albero non venga ristrutturato (tramite rotazioni)

Percorso Nero in alberi Red-Black

Definizione: **Il *numero* di *nodi neri* lungo ogni percorso da un *nodo x* (escluso) ad una *foglia* (nodo *NIL*) è detto *altezza nera di x***

Definizione: **L'*altezza nera di un albero Red-Black* è l'*altezza nera della sua radice*.**

Percorso massimo in alberi Red-Black

Lemma: **Un *albero Red-Black* con *n* nodi ha altezza pari al più a:**

$$2\log(n+1)$$

Dimostrazione: ***Iniziamo col dimostrare per induzione che per il sottoalbero radicato in x vale***

$$\textit{Nodi\_Interni}(x) \geq \boxed{2^{bh(x)} - 1}$$

dove bh(x) è l'altezza nera dell'albero radicato in x.

Faremo induzione sull'altezza del nodo x.

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) $\geq$ $\boxed{2^{bh(x)} - 1}$

dove bh(x) è l'altezza nera di x.

Passo Base:

Se x ha altezza 0: allora x è un nodo NIL e il suo sottoalbero contiene 0 nodi interni.

La sua altezza nera è inoltre bh(x) = 0.

Otteniamo quindi:

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

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

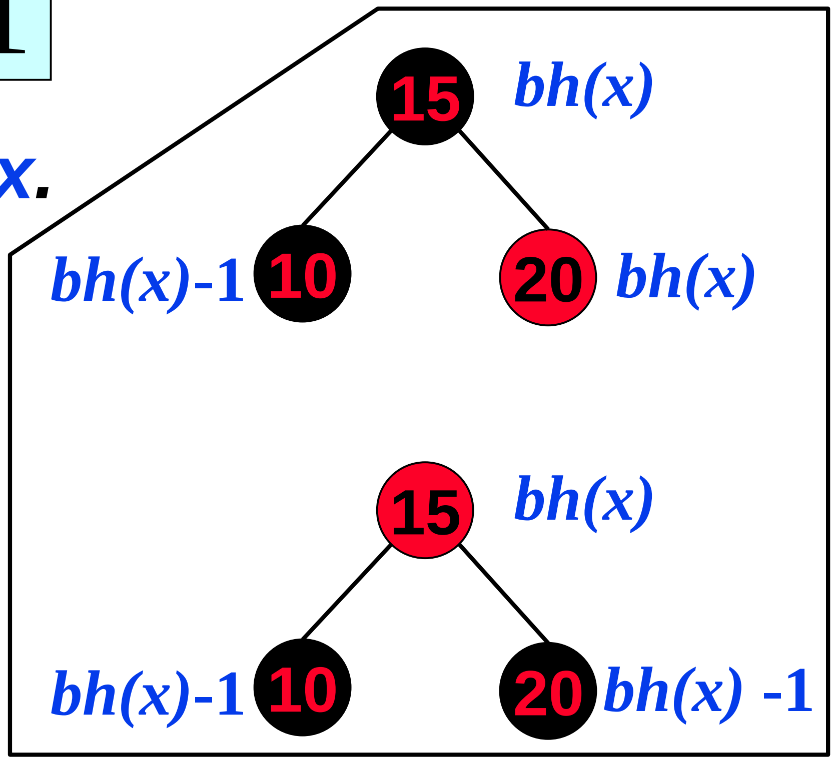
Nodi_Interni(x) $\geq$ $\boxed{2^{bh(x)} - 1}$

dove bh(x) è l'altezza nera di x.

Passo Induttivo:

Se x sia interno con 2 figli e abbia altezza bh(x)>1

Entrambi i suoi figli avranno altezza bh(x) o bh(x)-1



Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) $\geq$ $\boxed{2^{bh(x)}-1}$

dove bh(x) è l'altezza nera di x.

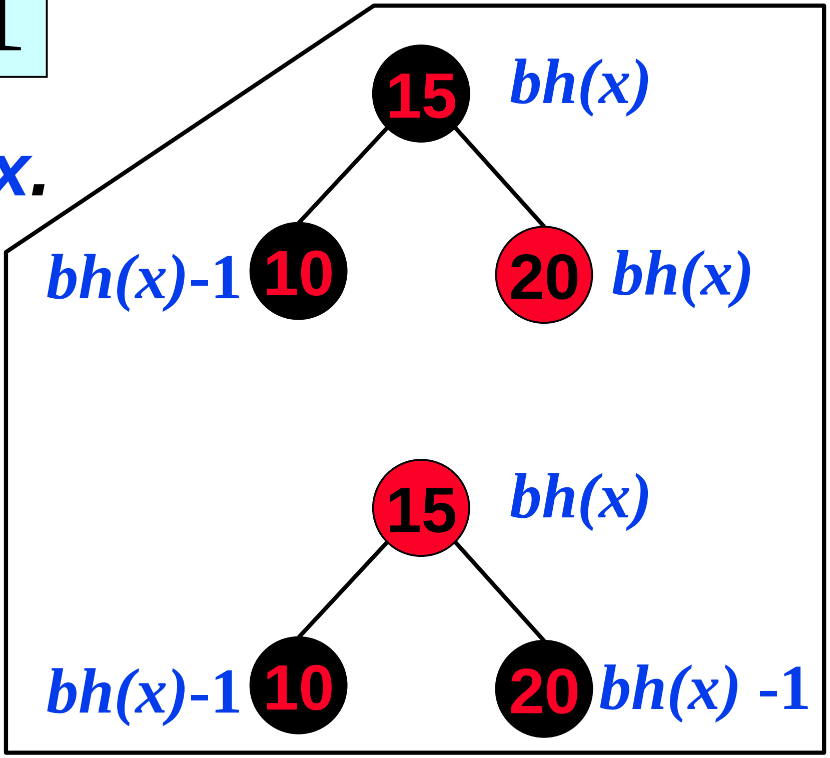
Passo Induttivo:

Sia x nodo interno con 2 figli

e abbia altezza > 0

Entrambi i suoi figli avranno

altezza nera bh(x) o bh(x)-1



Entrambi i suoi figli avranno certamente altezza minore di x, quindi possiamo usare ipotesi induttiva

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) $\geq$ $\boxed{2^{bh(x)} - 1}$

dove bh(x) è l'altezza nera di x.

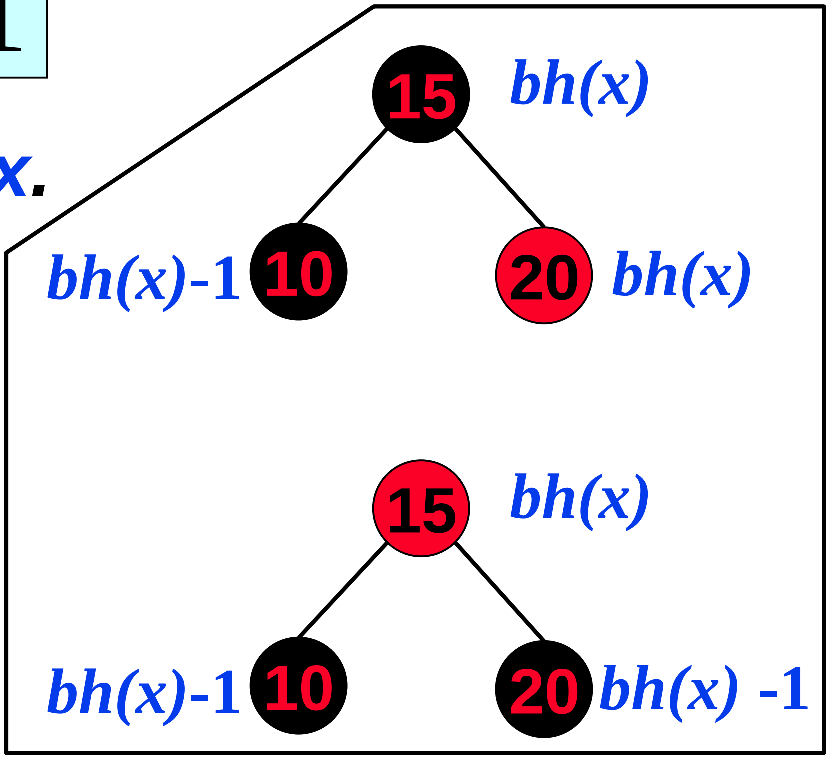
Passo Induttivo:

Se x è interno con 2 figli e

ha altezza > 0 allora

entrambi i suoi figli avranno

altezza nera bh(x) o bh(x)-1



Ogni figlio, per l'ipotesi induttiva, avrà almeno

$\boxed{2^{bh(x)-1} - 1}$ ***nodi interni***

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) $\geq$ $$\boxed{2^{bh(x)} - 1}$$

dove bh(x) è l'altezza nera di x.

Passo Induttivo: Sia x nodo interno con 2 figli e abbia altezza >0. Ogni figlio avrà almeno

$$\boxed{2^{bh(x)-1} - 1}$$ ***nodi interni***

Quindi il sottoalbero con radice x conterrà almeno

$$\boxed{(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = (2^{bh(x)} - 1)}$$ ***nodi interni***

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) $\geq \boxed{2^{bh(x)} - 1}$

dove bh(x) è l'altezza nera di x.

Completiamo la dimostrazione del lemma.

Sia ora h l'altezza dell'albero.

Per il vincolo 3 almeno metà dei nodi lungo ogni percorso (esclusa la radice) sono neri.

Quindi l'altezza nera dell'albero dovrà essere almeno h/2 (cioè bh $\geq$ ***h/2).***

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) $\geq$ $\boxed{2^{bh(x)} - 1}$

dove bh(x) è l'altezza nera di x.

Completiamo la dimostrazione del lemma.

Sia ora h l'altezza dell'albero.

Quindi l'altezza nera dell'albero dovrà essere almeno h/2 (cioè bh $\geq$ ***h/2).***

Ma allora, il numero di nodi dell'albero sarà

$$\boxed{n \geq (2^{bh(x)} - 1) \geq 2^{h/2} - 1}$$

Percorso massimo in alberi Red-Black

Lemma: **Un *albero Red-Black* con *n* nodi ha altezza <u>al più</u> *2 log(n+1)***

Dimostrazione: ….

Completiamo la dimostrazione del lemma.

Sia ora h l'altezza dell'albero.

Ma allora, il numero di nodi dell'albero sarà

$$\boxed{n \geq (2^{bh(x)} - 1) \geq 2^{h/2} - 1}$$

Portando 1 a sinistra e applicando il logaritmo:

$$\boxed{\log(n+1) \geq h/2}$$ ***cioè*** $$\boxed{h \leq 2\log(n+1)}$$

Costo operazioni su alberi Red-Black

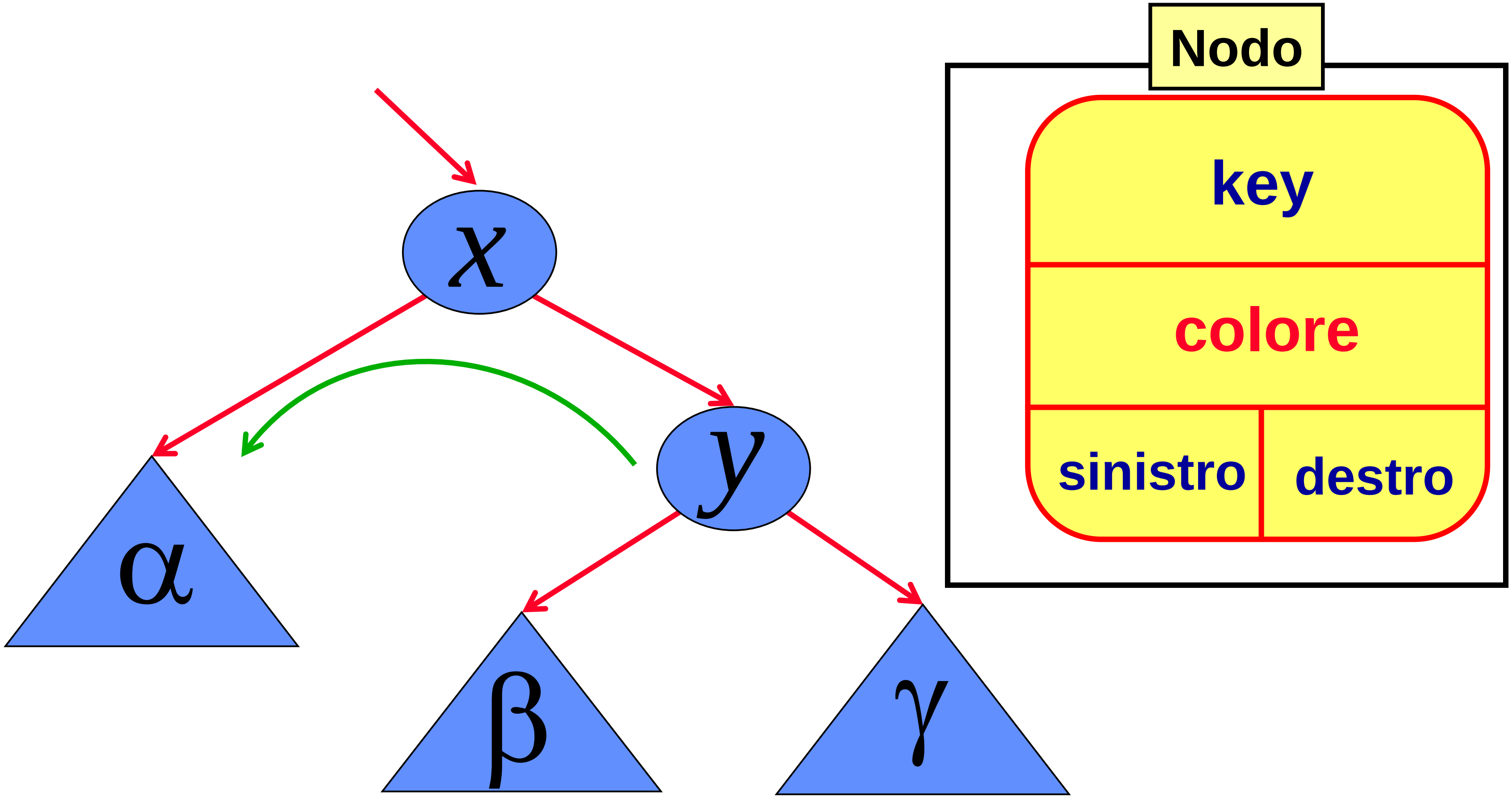
Teorema: **In un *albero Red-Black* le operazioni di *ricerca*, *inserimento* e *cancellazione* hanno *costo O(log(n)).***

Dimostrazione: ***Conseguenza diretta del Lemma precedente e dal fatto che tutte le operazioni hanno costo O(h), con h l'altezza dell'albero.***

Violazione delle proprietà per inserimento

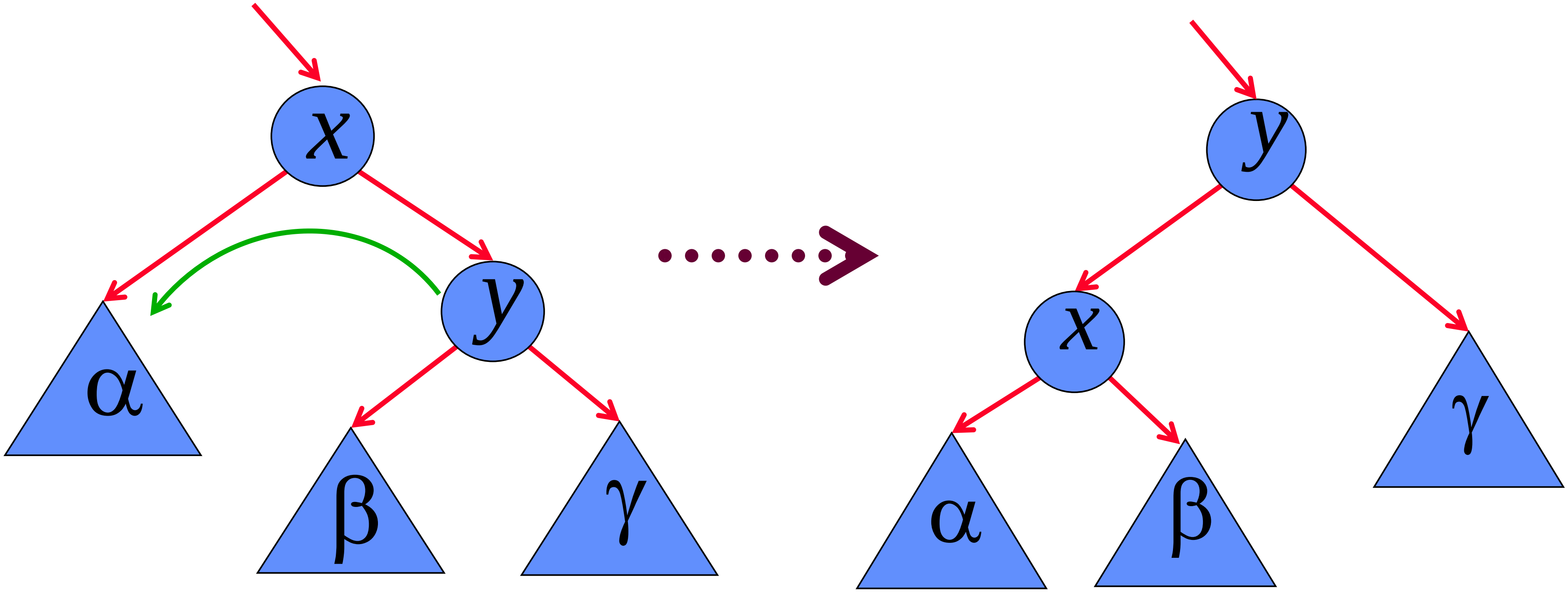
- **Durante la *costruzione* di un *albero Red-Black*, quando un *nuovo nodo* viene *inserito* è possibile che *le condizioni di bilancia-mento* risultino *violate*.**
- **Lo stsso accade per le *cancellazioni*.**
- **Quando le *proprietà Red-Black* vengono *violate* si può agire:**
  - ***modificando i colori nella zona della violazione;***
  - ***operando dei ribilanciamenti dell'albero tramite rotazioni: Rotazione destra e Rotazione sinistra;***

Alberi Red-Black: Rotazione destra

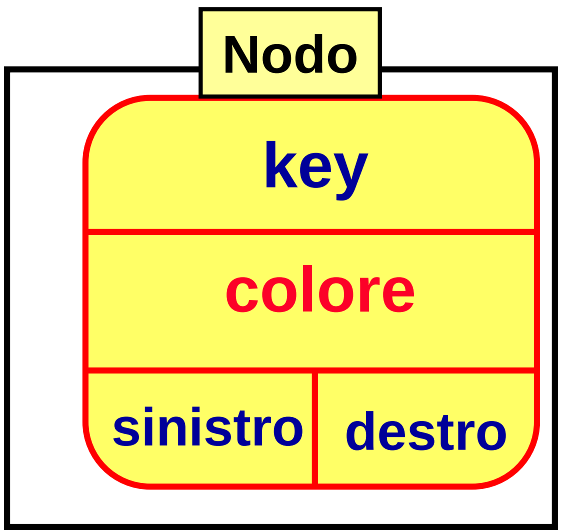


Rotazione col figlio destro
Rotazione va da destra a sinistra
Il Cormen la chiama Rotazione a Sinistra

Alberi Red-Black: Rotazione destra



Rotazione col figlio destro
Rotazione va da destra a sinistra
Il Cormen la chiama Rotazione a Sinistra



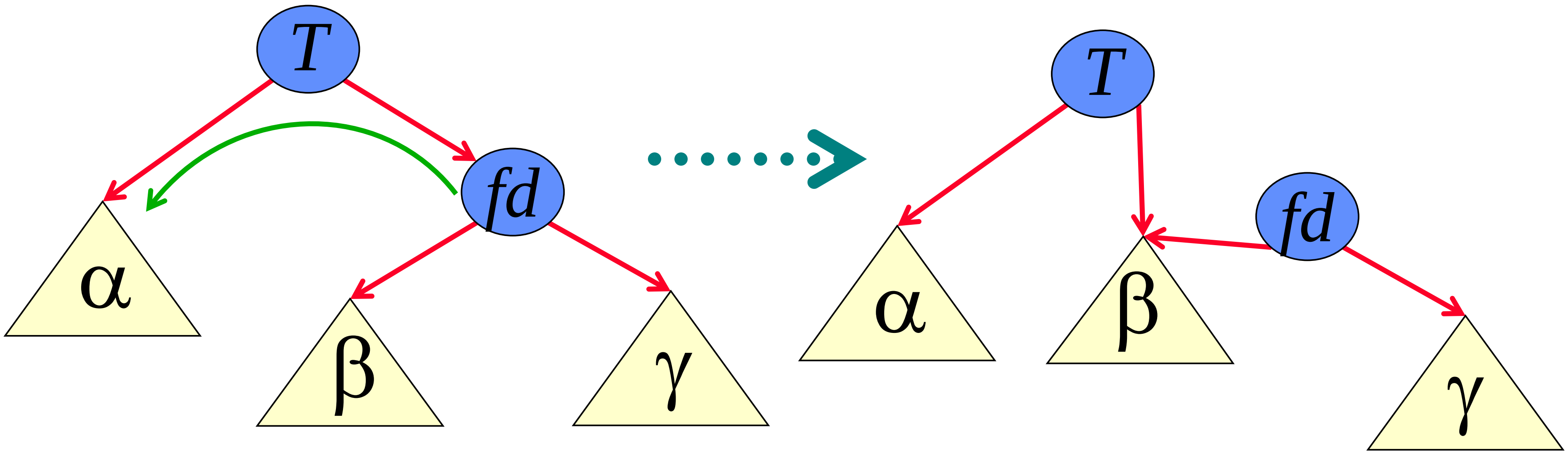
Alberi Red-Black: Rotazioni

```
albero-RB Ruota-destro(T:albero-RB)
    fd = T->dx
    T->sx = fd->sx
    fd->sx = T
    return fd
```

```
albero-RB Ruota-sinistro(T:albero-RB)
    fs = T->sx
    T->sx = fd->dx
    fd->dx = T
    return fs
```

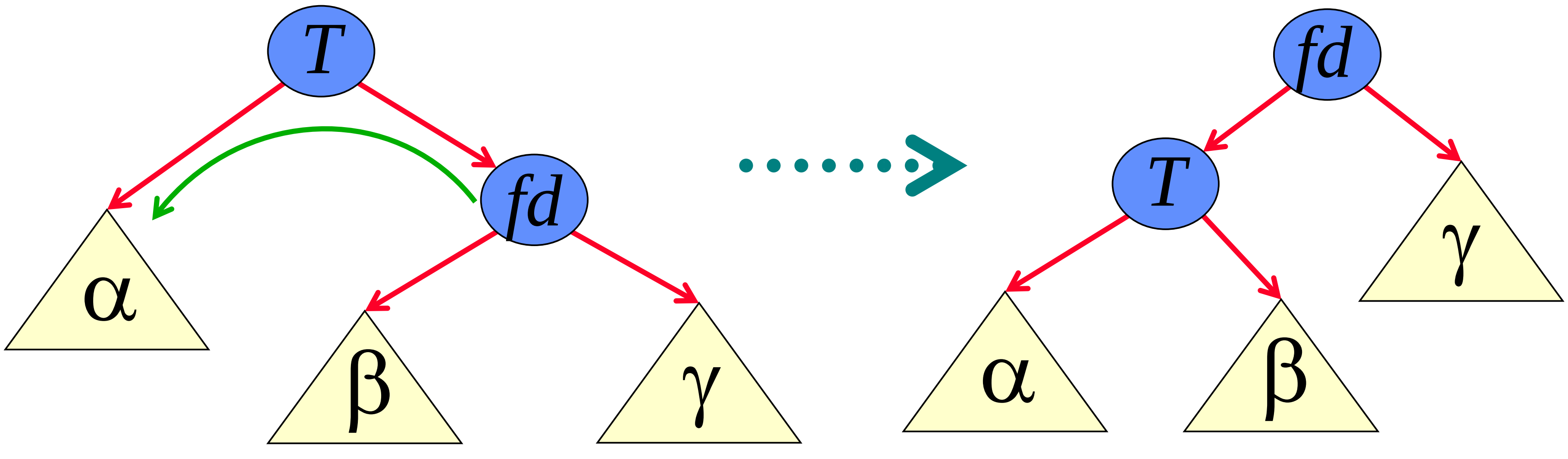
Alberi Red-Black: Rotazioni

```
albero-RB Ruota-destro(T:albero-RB)
    fd = T->dx
    T->dx = fd->sx
    fd->sx = T
    return fd
```



Alberi Red-Black: Rotazioni

```
albero-RB Ruota-destro(T:albero-RB)
    fd = T->dx
    T->dx = fd->sx
    fd->sx = T
    return fd
```



Inserimento in alberi Red-Black

- ***Inseriamo un nuovo nodo (chiave) usando la stessa procedura usata per gli ABR;***
- ***Coloriamo il nuovo nodo di rosso ;***
- ***La ritorno dalle chiamate ricorsive di inserimento, ripristiniamo la proprietà Red-Black se è il caso:***
  - ***spostiamo le violazioni verso l'alto rispettando sempre il vincolo 4 (mantenendo l'altezza nera dell'albero);***
- ***Al termine dell'inserimento, coloriamo sempre la "radice di nero".***

Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi!

Inserimento in alberi Red-Black

```
albero-RB INS_RB(k, T)
  IF not IS-NIL(T) THEN
     IF k < T->key THEN
         T->sx = INS_RB(k, T->sx);
         T = Bilancia_sx(T);
     ELSE   /* k ≥ K[T] */
         T->dx = INS_RB(k, T->dx);
         T = Bilancia_dx(T);
  ELSE
     T = alloca nodo;  T->key = k;
     T->dx = NIL[T]; T->sx = NIL[T];
     T->color = red;
  return T;
```

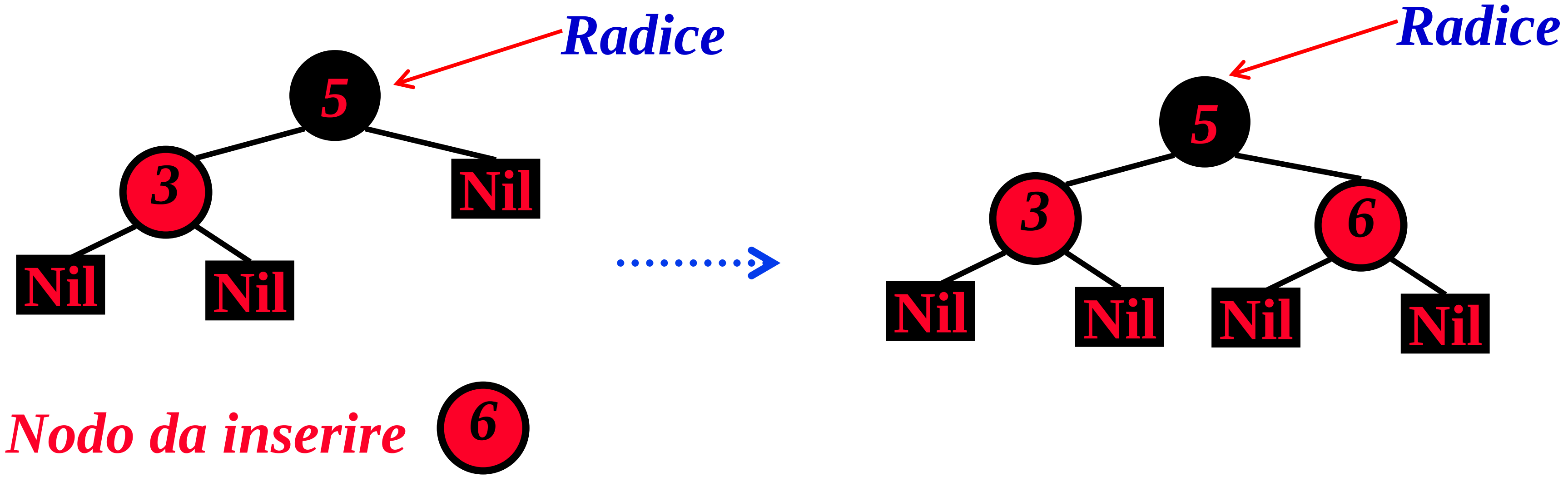
Al termine dell'inserimento, la proc. chiamante deve sempre colorare la "radice di nero".

Inserimento in alberi Red-Black

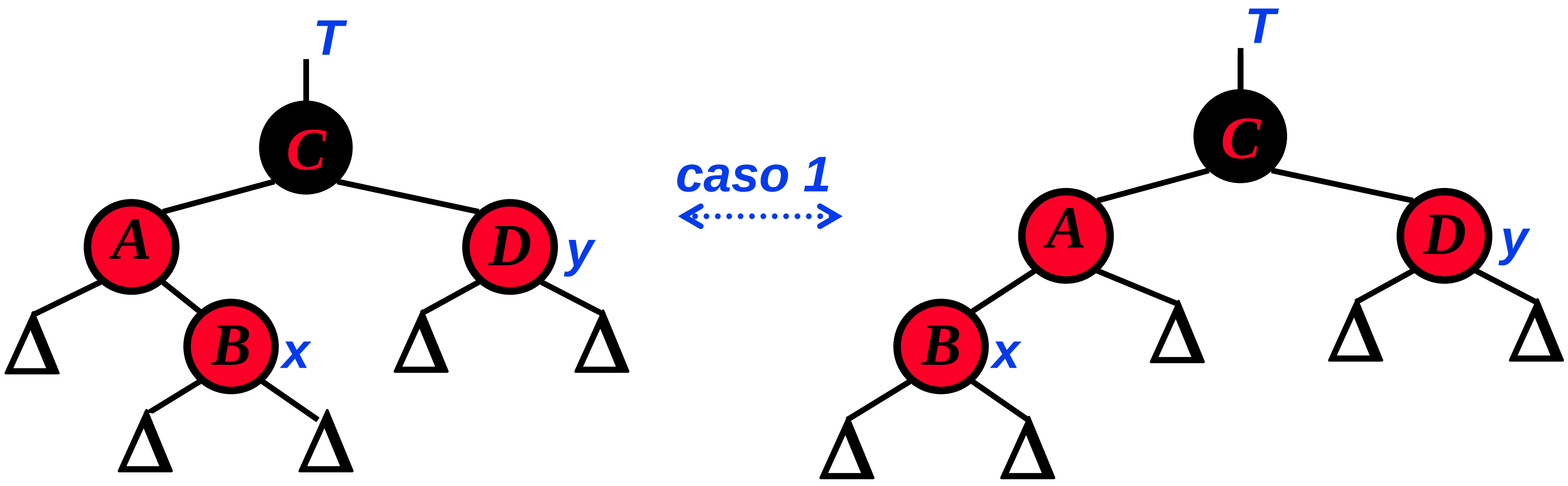
Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi!

Se la radice dell'albero è sempre nera, non si presenta mai la necessità di ribilanciare in un albero (o sottoalbero) di altezza minore di 3!

Non si possono, infatti, verificare violazioni!



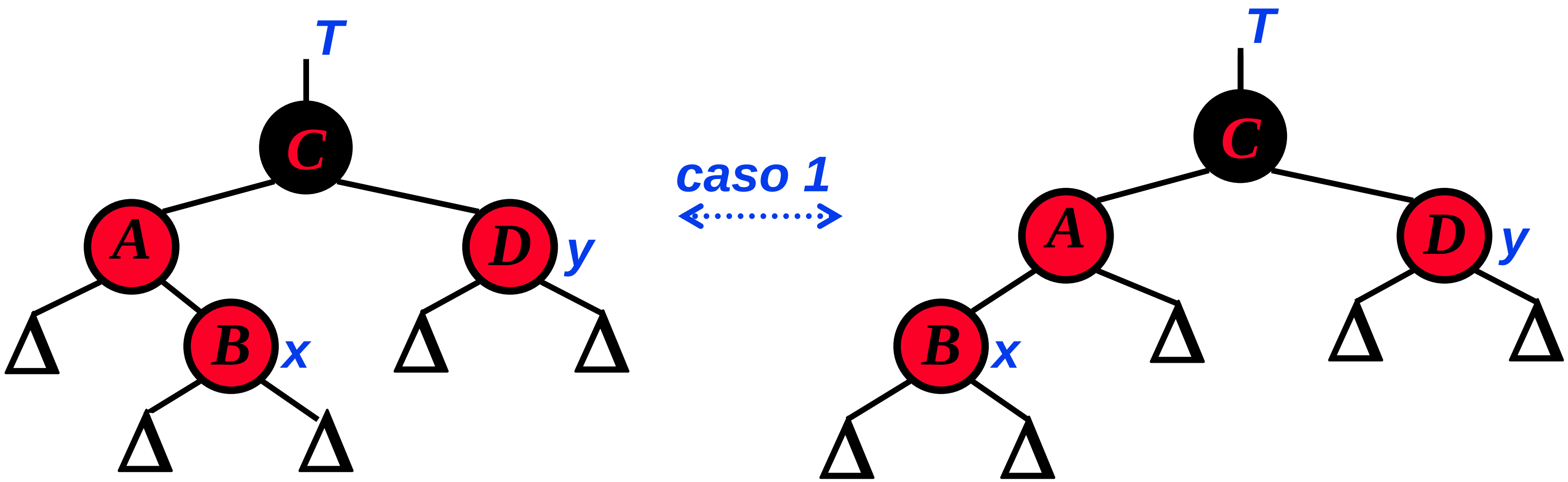
Ribilanciamenti: casi 1-3

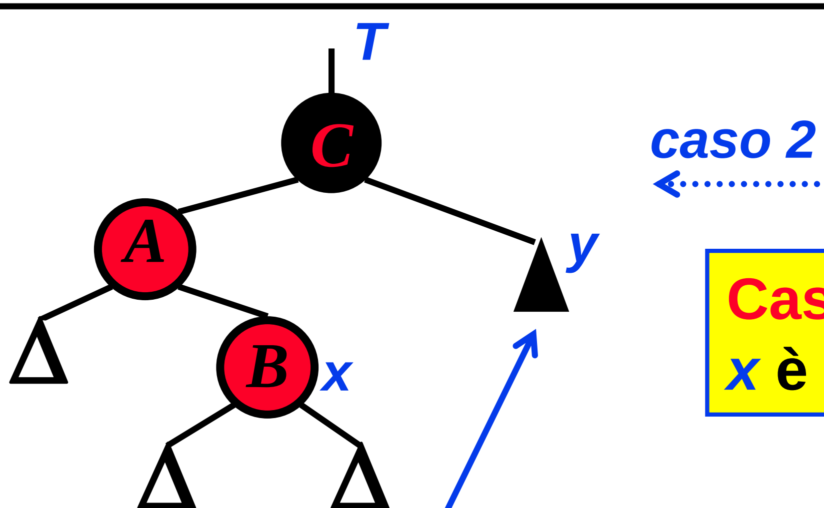


Caso 1: il figlio *y* di *T* è rosso

- **x è il *nodo modificato* che provoca la violazione Red-Black**
- **y è il figlio destro di T**

Ribilanciamenti: casi 1-3

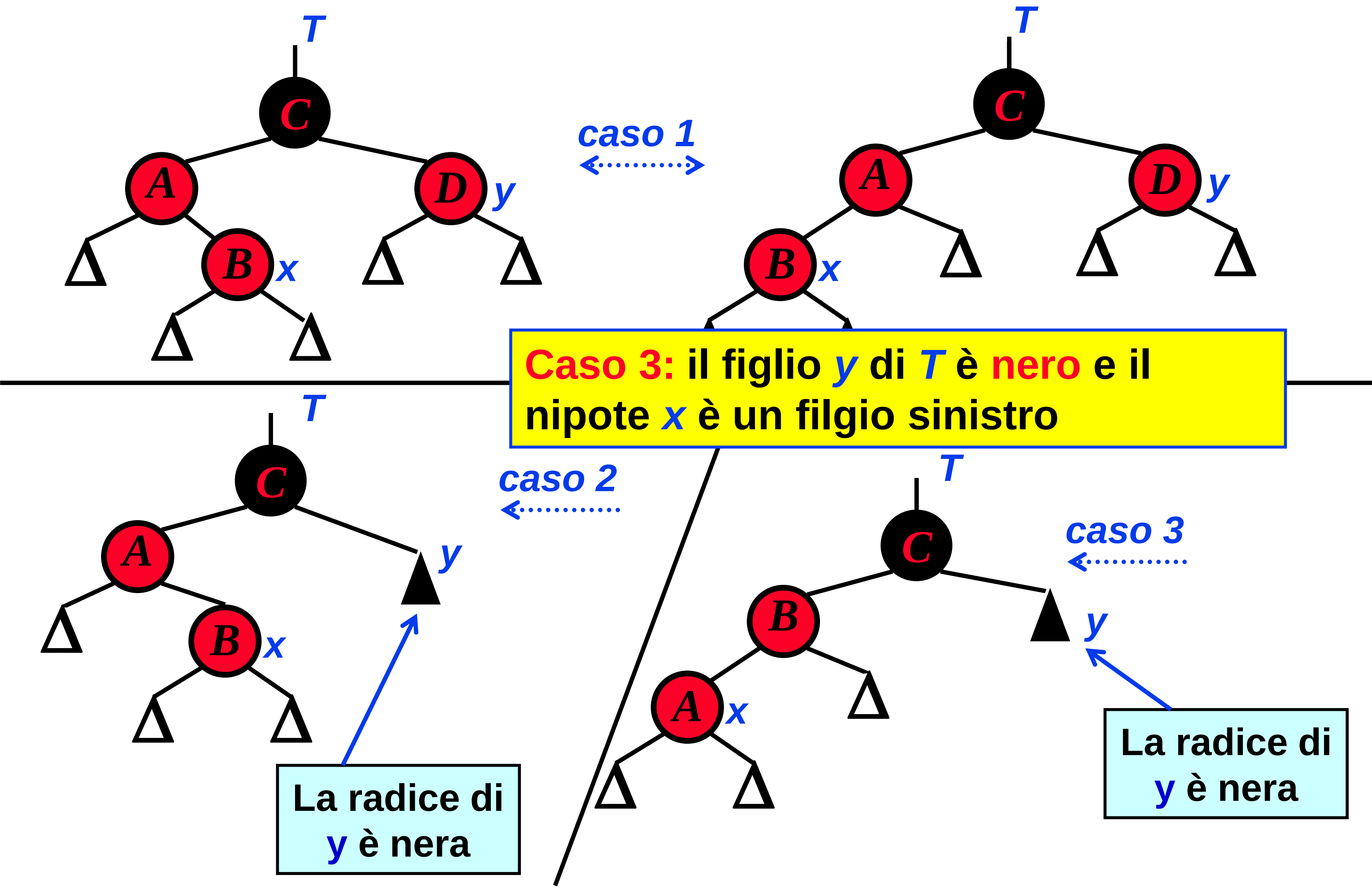




Caso 2: **il figlio *y* di *T* è nero e il nipote *x* è un filgio destro**

Questa radice è nera

Ribilanciamenti: casi 1-3



Bilanciamento del sottoalbero sinistro

Verifica se l'altezza del sottoalbero sinistro di T è maggiore di 0

```
albero-RB Bilancia_sx(T)
   IF ha_figlio(T->sx)
          v = Tipo_violazione_sx(T->sx,T->dx)
                    /* v = 0   nessuna violazione */
          CASE v OF
                    1:      T = Bilancia_sx_1(T);
                    2:      T = Bilancia_sx_2(T);
                            T = Bilancia_sx_3(T);
                    3:      T = Bilancia_sx_3(T);
  return T;
```

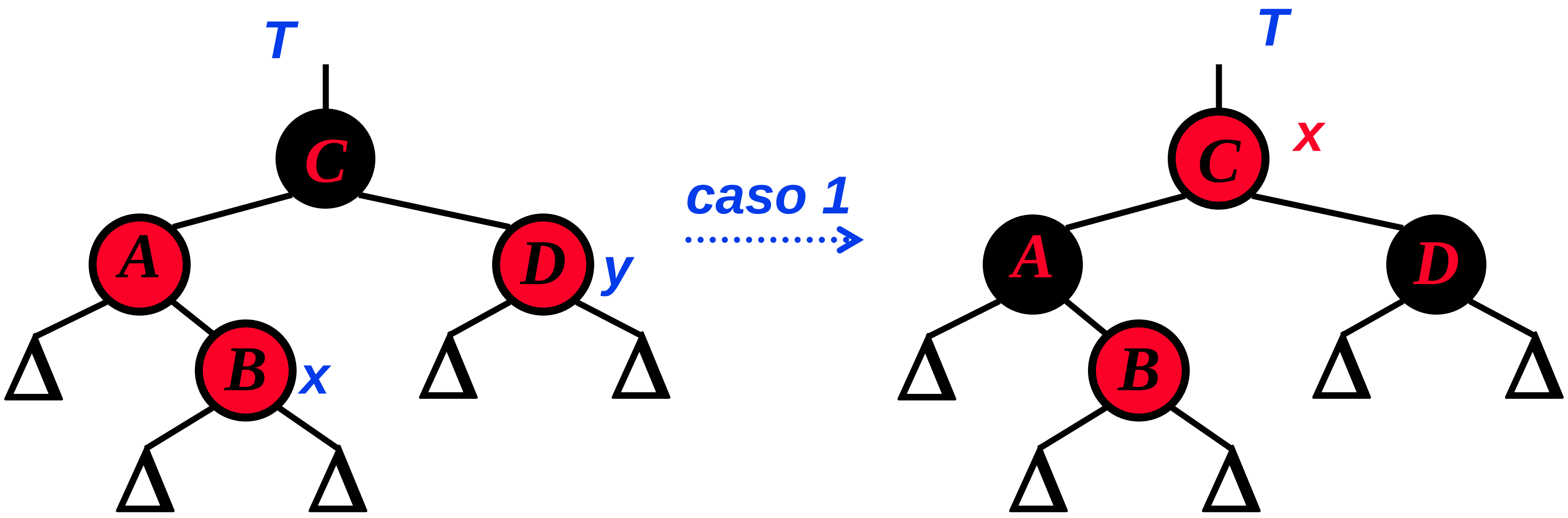
Calcolo del tipo di violazione

```
int Tipo_violazione_sx(s,d)
 violazione = 0
  IF s->color = red  AND d->color = red THEN
       IF s->sx->color = red OR s->dx->color = red THEN
            violazione = 1;
  ELSE /* d->color = black o s->color = black */
       IF s->color = red THEN
           IF s->dx->color = red
               violazione = 2;
            ELSE
                IF s->sx->color = red
                      violazione = 3;
return violazione;
```

Inserimento in alberi RB: Caso 1

Caso 1: **il figlio *y* di *T* è rosso**

Tutti i Δ sono sottoalberi di uguale altezza nera



Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

Caso 1: il figlio *y* del di *T* è **rosso**

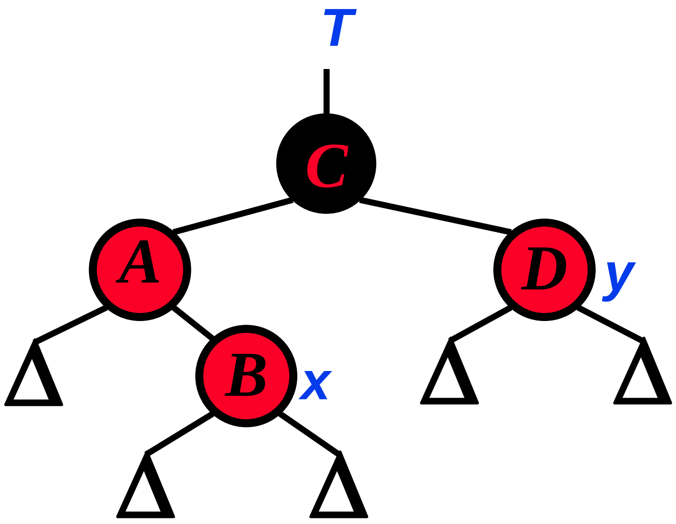
Tutti i Δ sono sottoalberi di uguale altezza nera

***T->dx->color=black*;**

***T->sx->color=black*;**

***T->color=red*;**

return *T*;



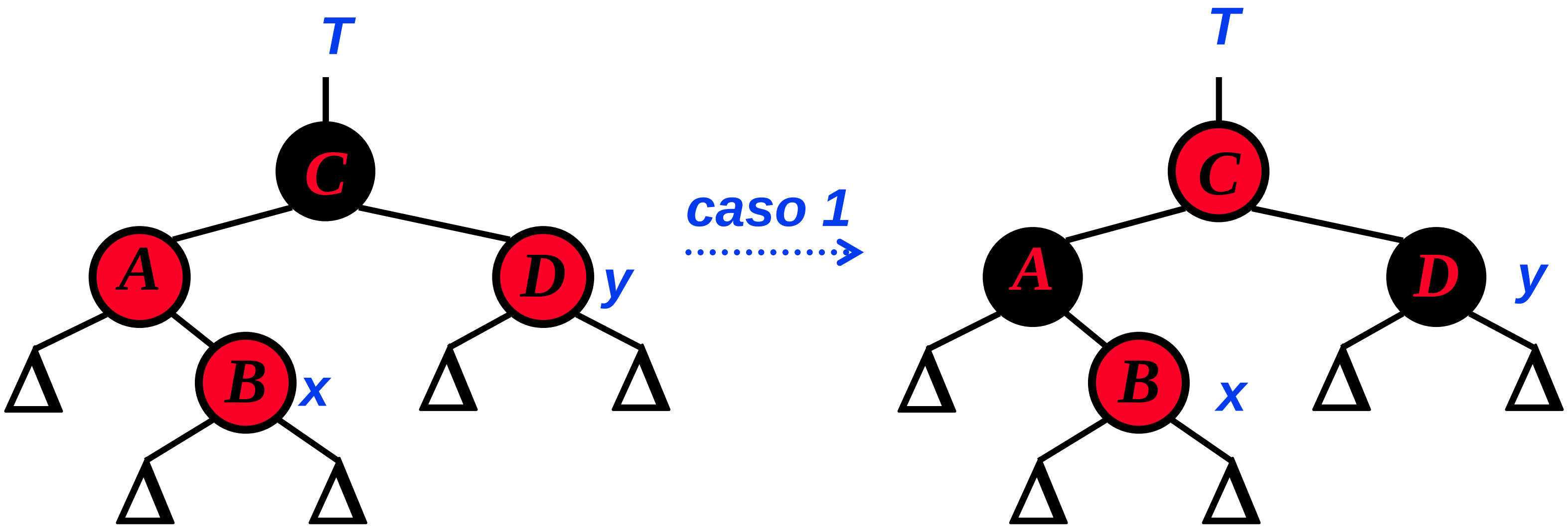
Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

Caso 1: il figlio *y* del di *T* è rosso

Tutti i Δ sono sottoalberi di uguale altezza nera

```
T->dx->color=black;
T->sx->color=black;
T->color=red;
return T;
```

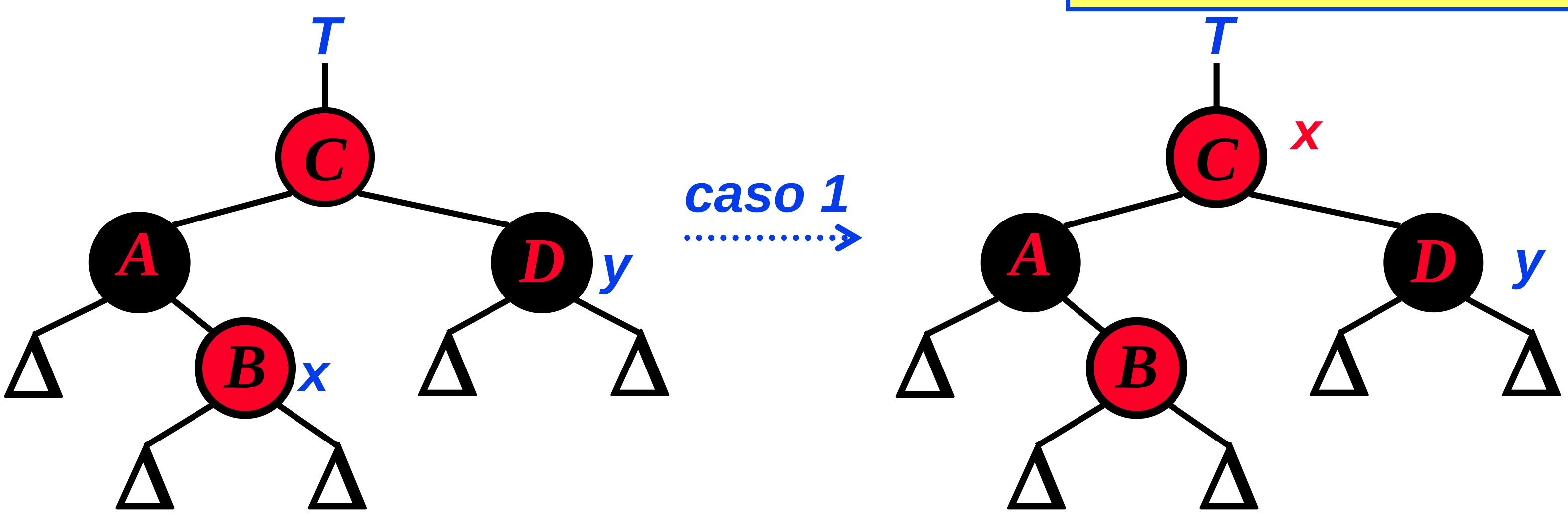


Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

```
T->dx->color=black;
T->sx->color=black;
T->color=red;
return T;
```

Poiché il padre di C può essere rosso, può essere necessario ripristinare la proprietà RB più in alto



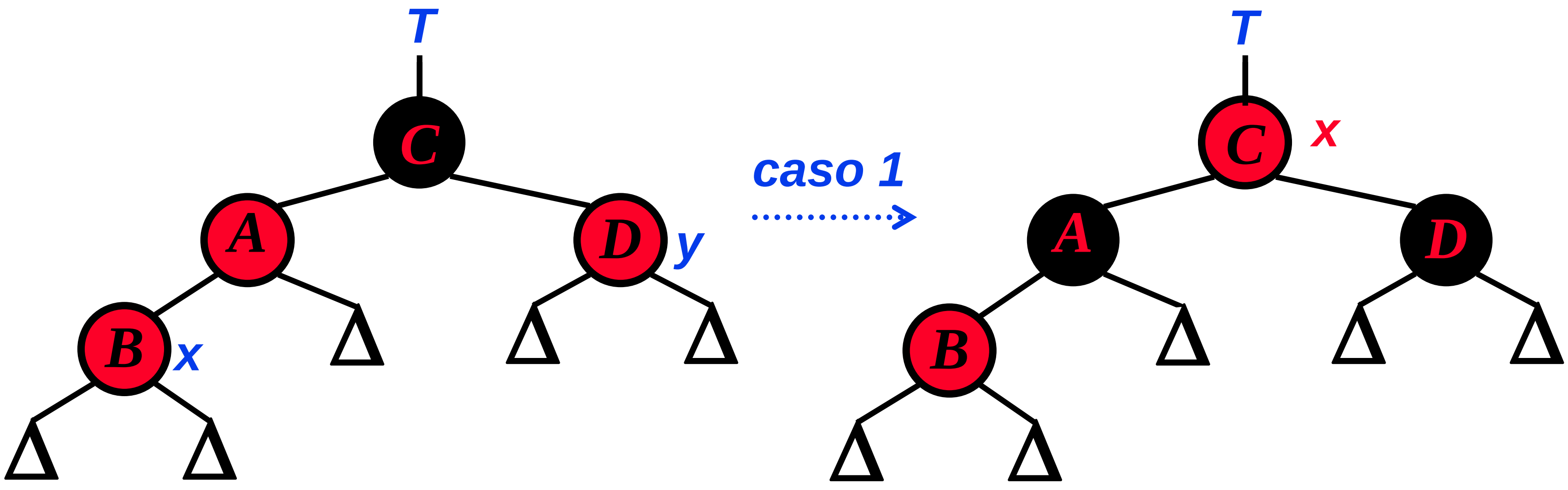
Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

```
T->dx->color=black;
T->sx->color=black;
T->color=red;
return T;
```

Caso 1: il figlio *y* del padre del padre di *x* è rosso

Tutti i Δ sono sottoalberi di uguale altezza nera

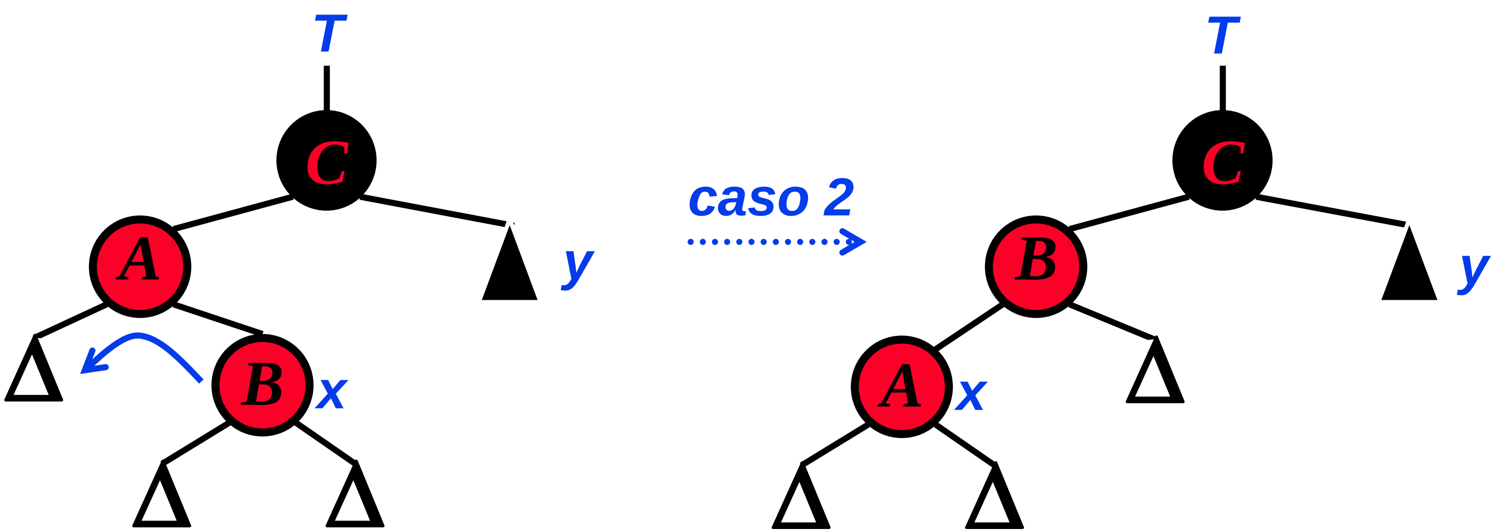


Nel Caso 1, si eseguono le stesse azioni sia quando x è figlio sinistro o figlio destro.

Inserimento in alberi RB: Caso 2

Caso 2:

- **il figlio *y* di *T* è nero**
- ***x* è un filgio destro**
- **Trasformiamo nel caso 3 con rotazione destra**



Trasformiamo il Caso 2 nel Caso 3 (x è figlio sinistro) con una rotazione destra. Ciò preserva il vincolo 4: tutti i percorsi sotto x contengonolo stesso numero di nodi neri

Inserimento in alberi RB: Caso 2

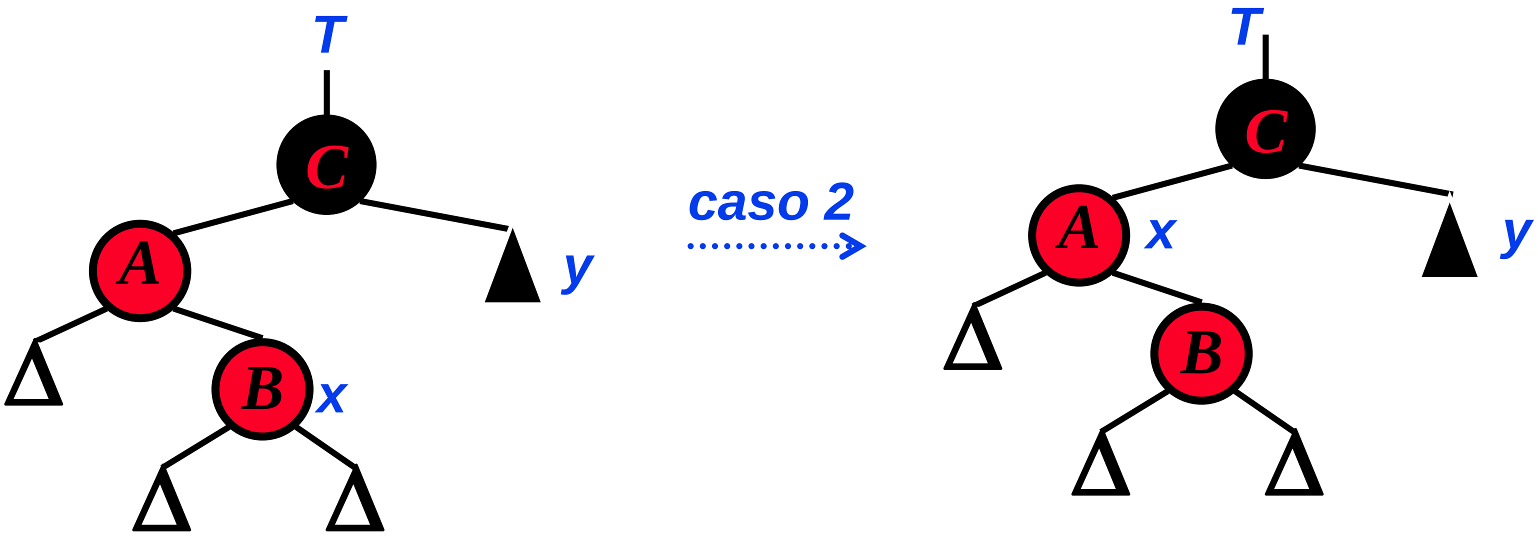
***T->sx* = Ruota-dx(*T->sx*)**

return *T*

// continua con caso 3

Caso 2:

- **il figlio *y* di *T* è nero**
- ***x* è un filgio destro**
- **Trasformiamo nel caso 3 con rotazione destra**



Trasformiamo il Caso 2 nel Caso 3 (x è figlio sinistro) con una rotazione destra. Ciò preserva il vincolo 4: tutti i percorsi sotto x contengonolo stesso numero di nodi neri

Inserimento in alberi RB: Caso 2

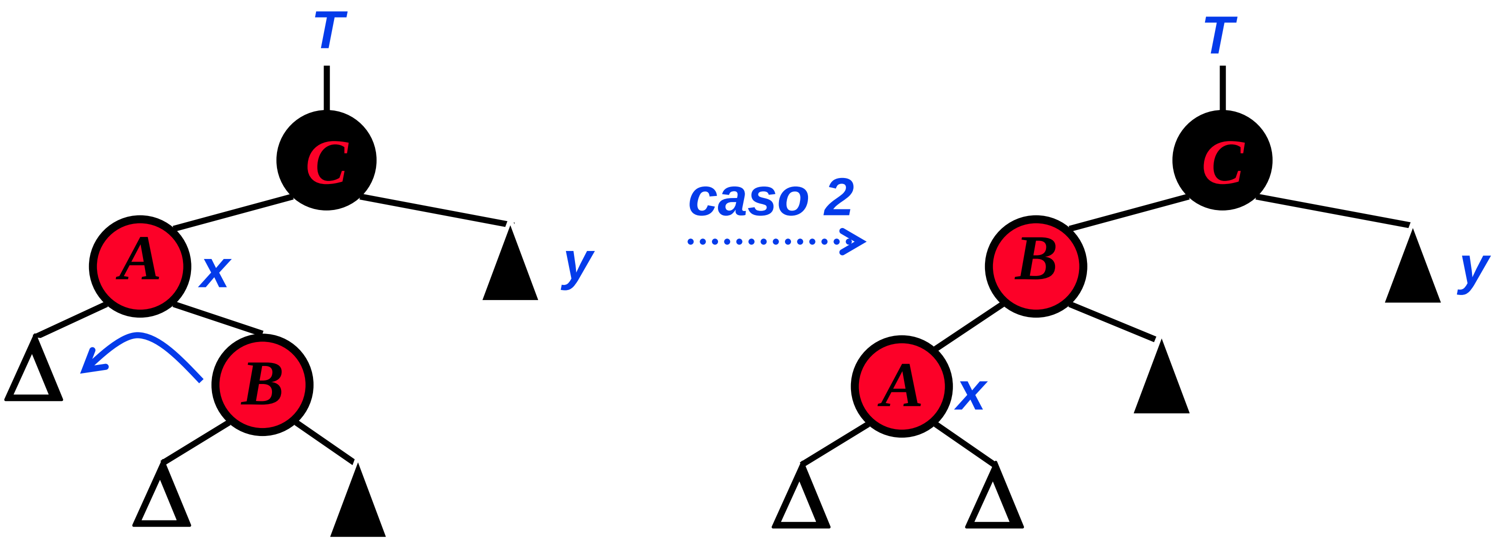
T->sx = Ruota-dx(*T->sx*)

return *T*

// continua con caso 3

Caso 2:

- il figlio *y* di *T* è nero
- *x* è un filgio destro
- Trasformiamo nel caso 3 con rotazione destra

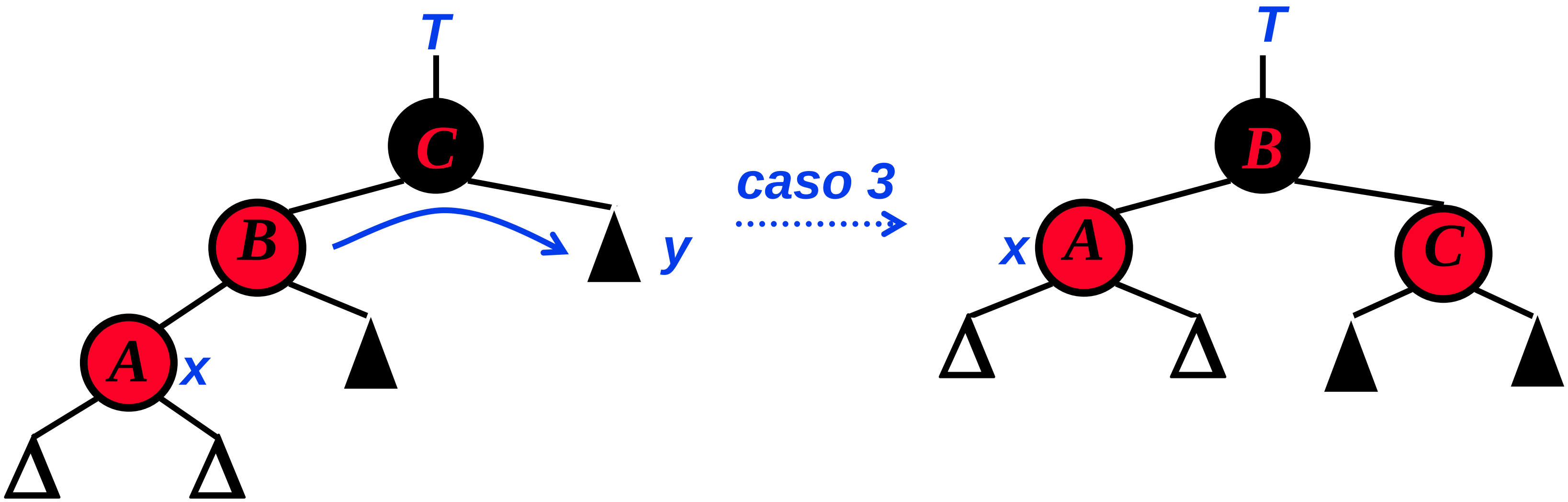


Trasformiamo il Caso 2 nel Caso 3 (x è figlio sinistro) con una rotazione destra. Ciò preserva il vincolo 4: tutti i percorsi sotto x contengonolo stesso numero di nodi neri

Inserimento in alberi RB: Caso 3

Caso 3:

- **il figlio *y* di *T* è nero**
- ***x* è un filgio sinistro**
- **Cambiare colori e rotazione sinistra**



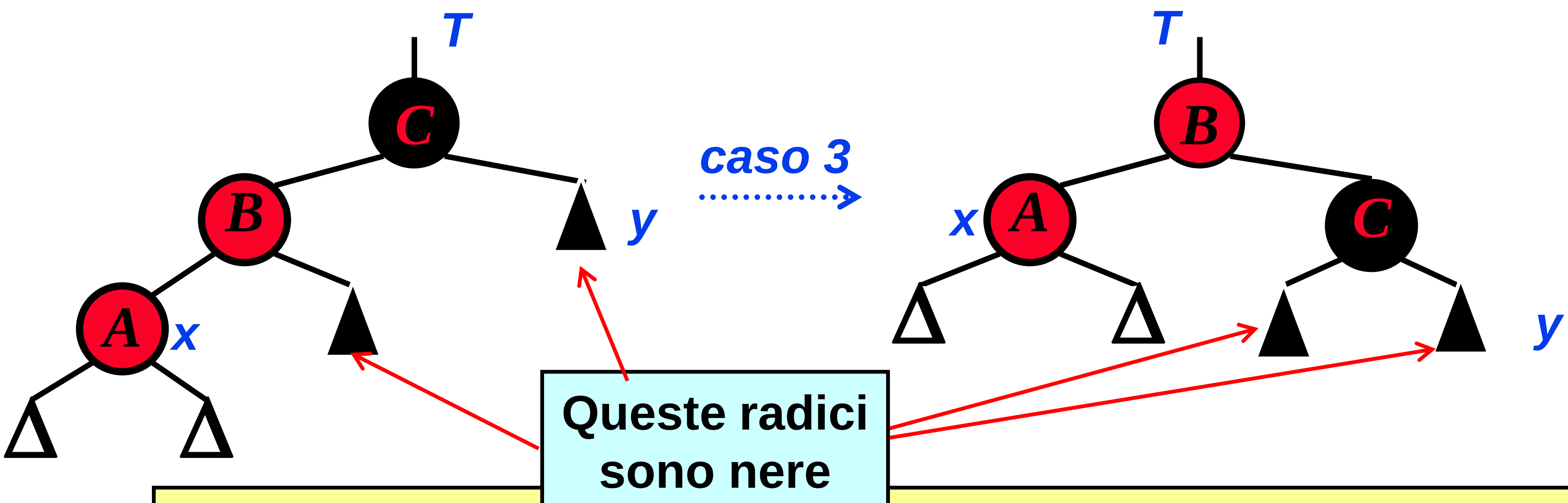
Eseguiamo alcuni cambi di colore e facciamo una rotazione sinistra. Di nuovo, preserviamo il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri.

Inserimento in alberi RB: Caso 3

```
T = Ruota_sx(T);
T->color=black;
T->dx->color=red;
return T;
```

Caso 3:

- **il figlio *y* del padre del padre di *x* è nero**
- ***x* è un filgio sinistro**
- **Cambiare colori e rotazione sinistra**



Eseguiamo alcuni cambi di colore e facciamo una rotazione sinistra. Di nuovo, preserviamo il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri.

Inserimento in alberi RB: Caso 3

T = Ruota_sx(*T*)

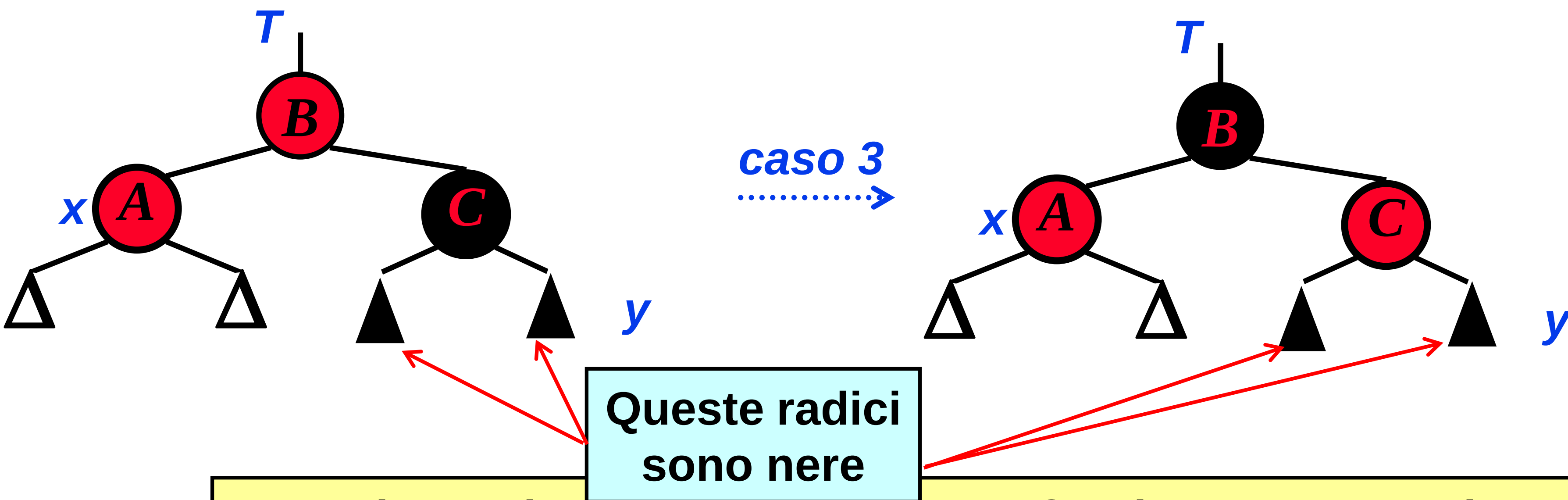
T->color=black;

T->dx->color=red;

return *T*

Caso 3:

- il figlio *y* del padre del padre di *x* è nero
- *x* è un filgio sinistro
- Cambiare colori e rotazione sinistra



Eseguiamo alcuni cambi di colore e facciamo una rotazione sinistra. Di nuovo, preserviamo il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri.

Bilanciamento in alberi Red-Black

```
albero-RB Bilancia_sx_1(T)
    T->color = red;
    T->dx->color = black;
    T->sx->color = black;
    return T;
```

```
albero-RB Bilancia_sx_2(T)
    T->sx = Ruota_dx(T ->sx);
    return T;
```

```
albero-RB Bilancia_sx_3(T)
    T = Ruota_sx(T )
    T->color = black;
    T->dx->color = red;
    return T;
```

Bilanciamento del sottoalbero sinistro

```
albero-RB Bilancia_sx(T)
   IF ha_figlio(T->sx)
          v = Tipo_violazione_sx(T->sx,T->dx)
                    /* v = 0   nessuna violazione */
           CASE v OF
                    1:      T = Bilancia_sx_1(T);
                    2:      T = Bilancia_sx_2(T);
                            T = Bilancia_sx_3(T);
                    3:      T = Bilancia_sx_3(T);
  return T;
```

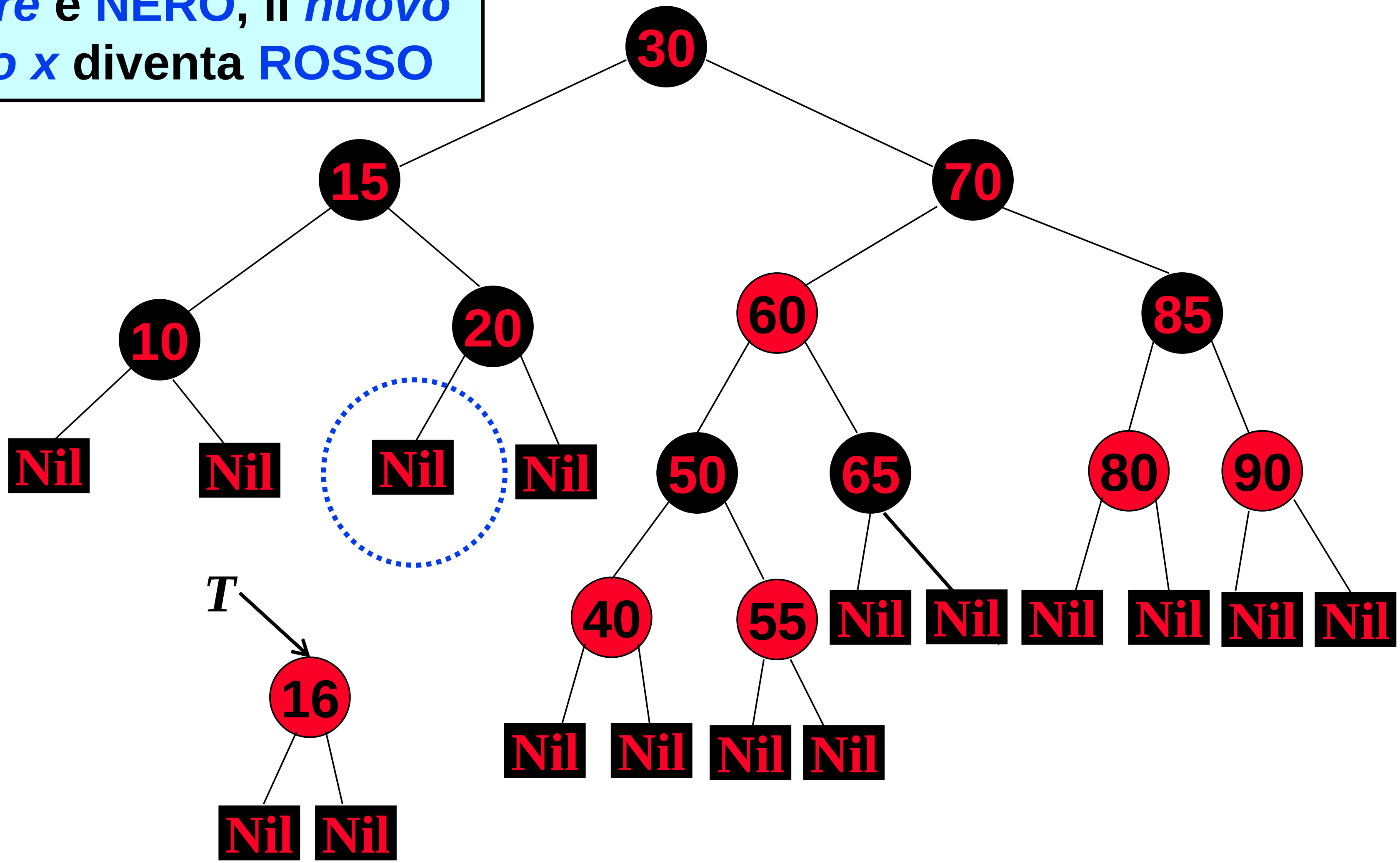
Inserimento in alberi RB: Casi 4-6

- **I casi 1-3 assumono che il figlio di T, che vìola con un nipote di T, sia un figlio sinistro.**
- **Se il figlio di T, che vìola con nipote di T, è un figlio destro si applicano i casi 4-6, che sono simmetrici (dobbiamo solo scambiare *sinistro* con *destro* e viceversa).**

L'estensione dell'algoritmo ai *casi 4-6* è lasciato come *esercizio*

Inserimento in alberi Red-Black: I

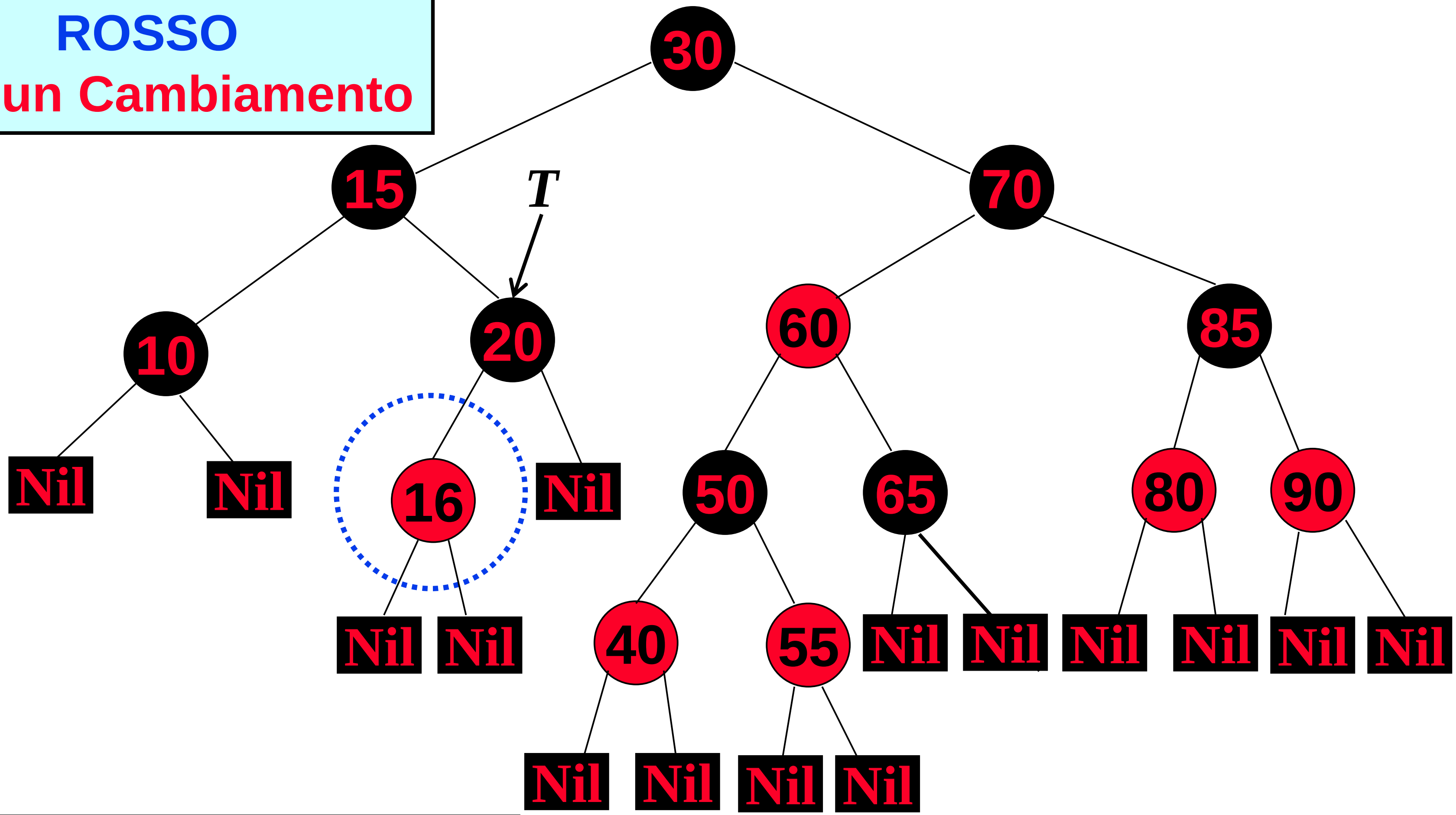
Il *padre* è NERO, il *nuovo nodo x* diventa ROSSO



Inserimento in alberi Red-Black: I

Il *T* è NERO, il *nuovo nodo inserito* diventa ROSSO

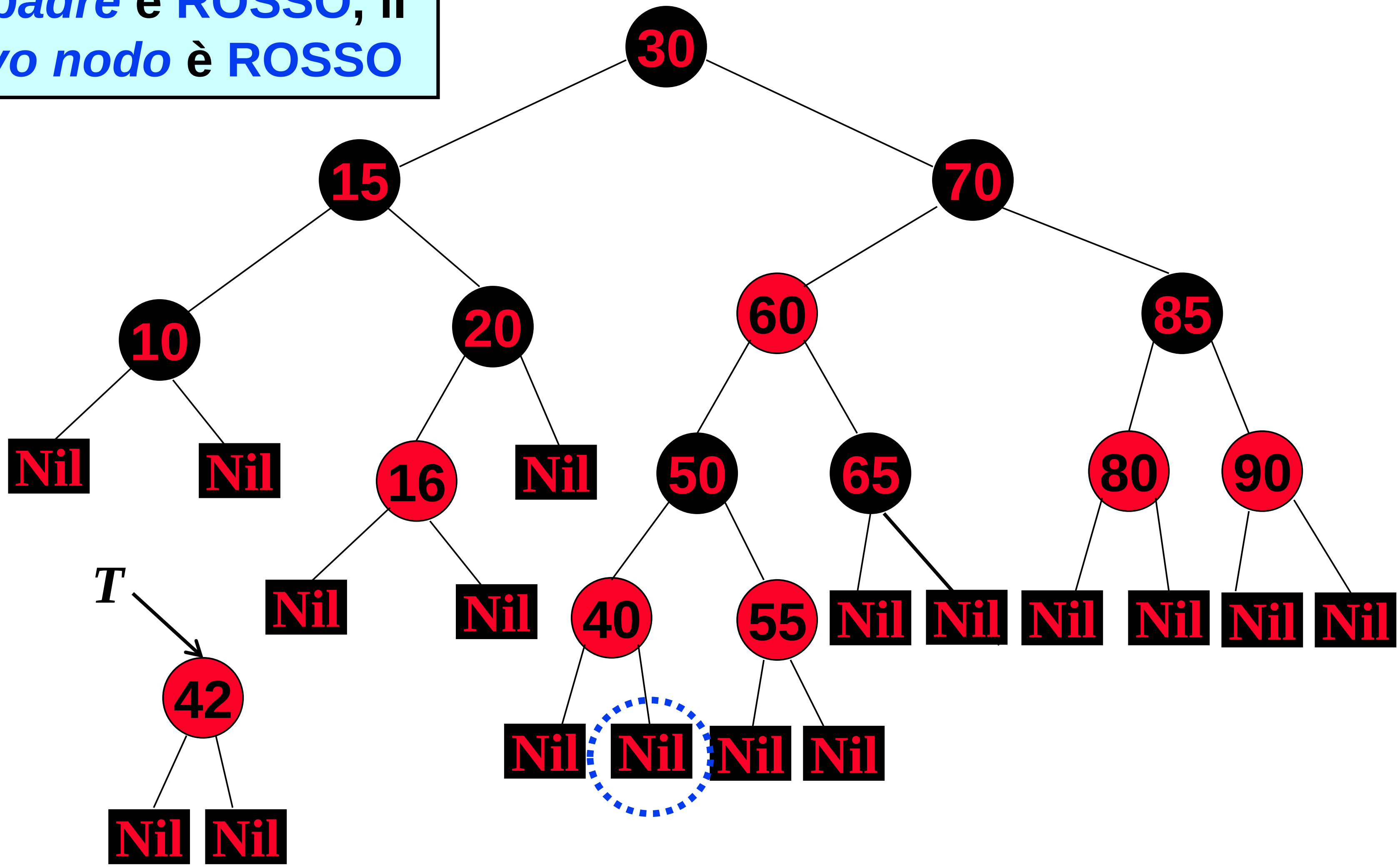
Nessun Cambiamento



Non cambia l'altezza nera di nessun nodo!

Inserimento in alberi Red-Black: II

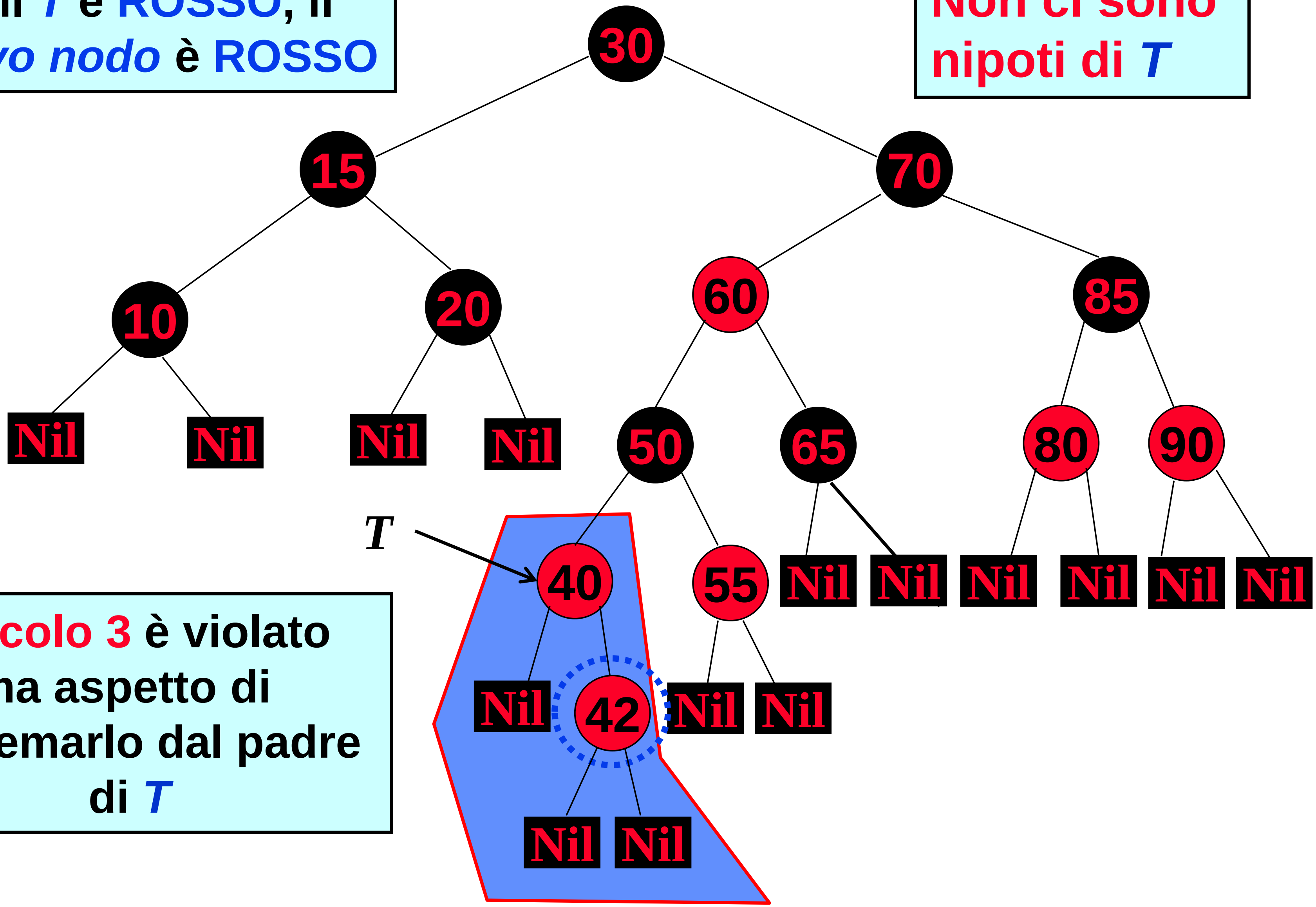
Se il *padre* è ROSSO, il *nuovo nodo* è ROSSO



Inserimento in alberi Red-Black: II

Se il *T* è ROSSO, il *nuovo nodo* è ROSSO

Non ci sono nipoti di *T*

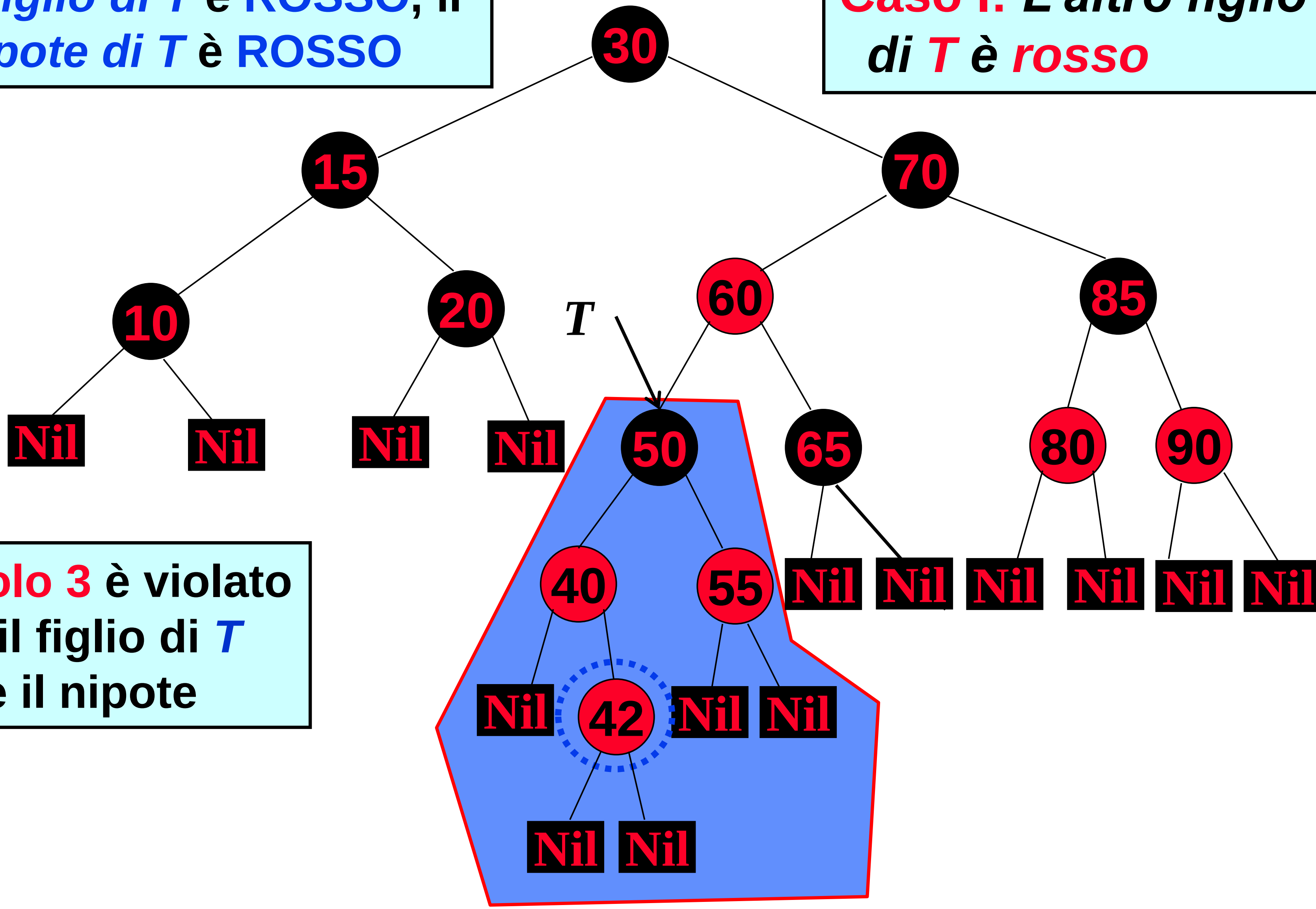


Vincolo 3 è violato ma aspetto di risistemarlo dal padre di *T*

Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il *nipote di T* è ROSSO

Caso I: ***L'altro figlio di T è rosso***

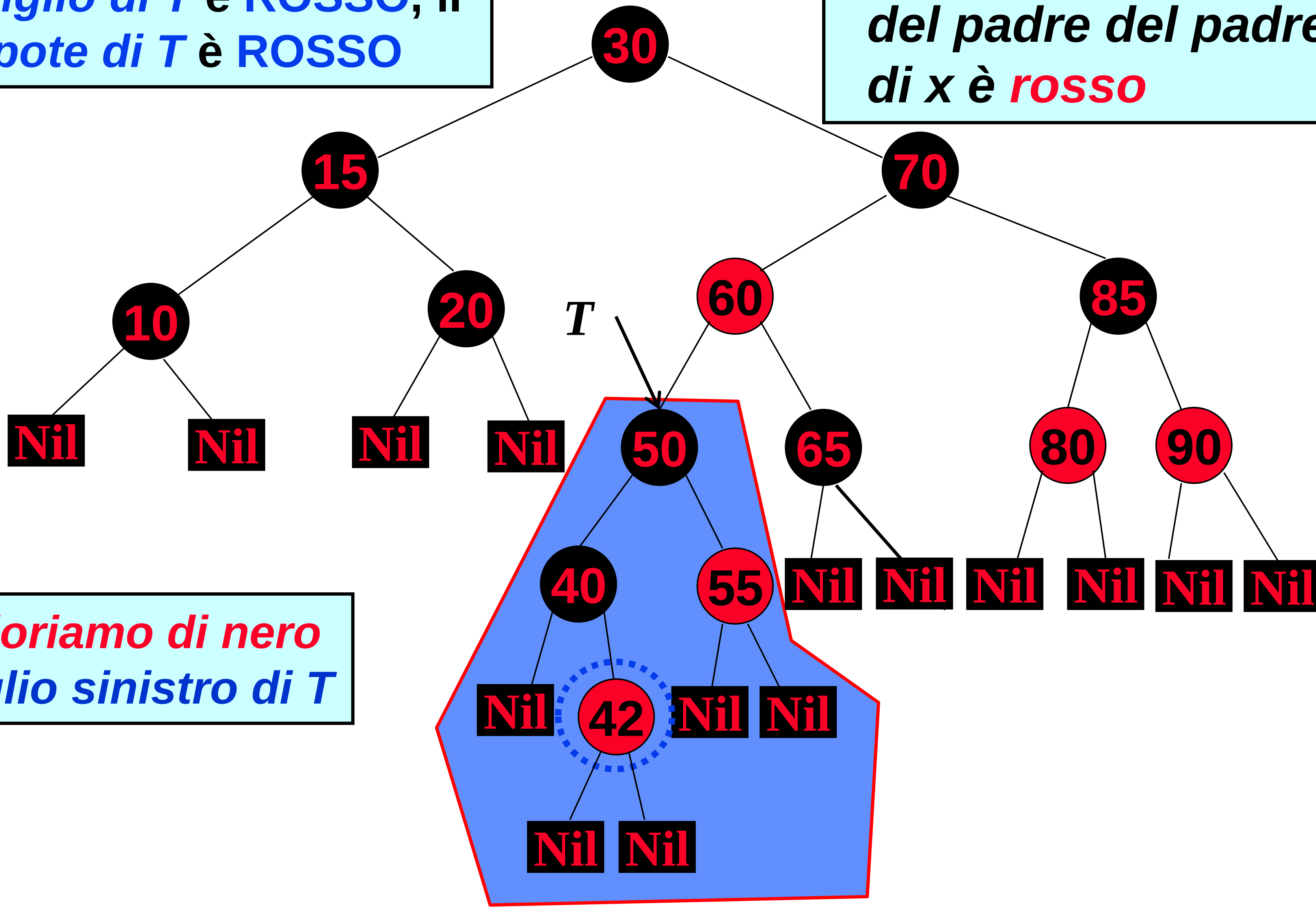


Vincolo 3 è violato tra il figlio di *T* e il nipote

Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il *nipote di T* è ROSSO

Caso I: *L'altro figlio del padre del padre di x è rosso*

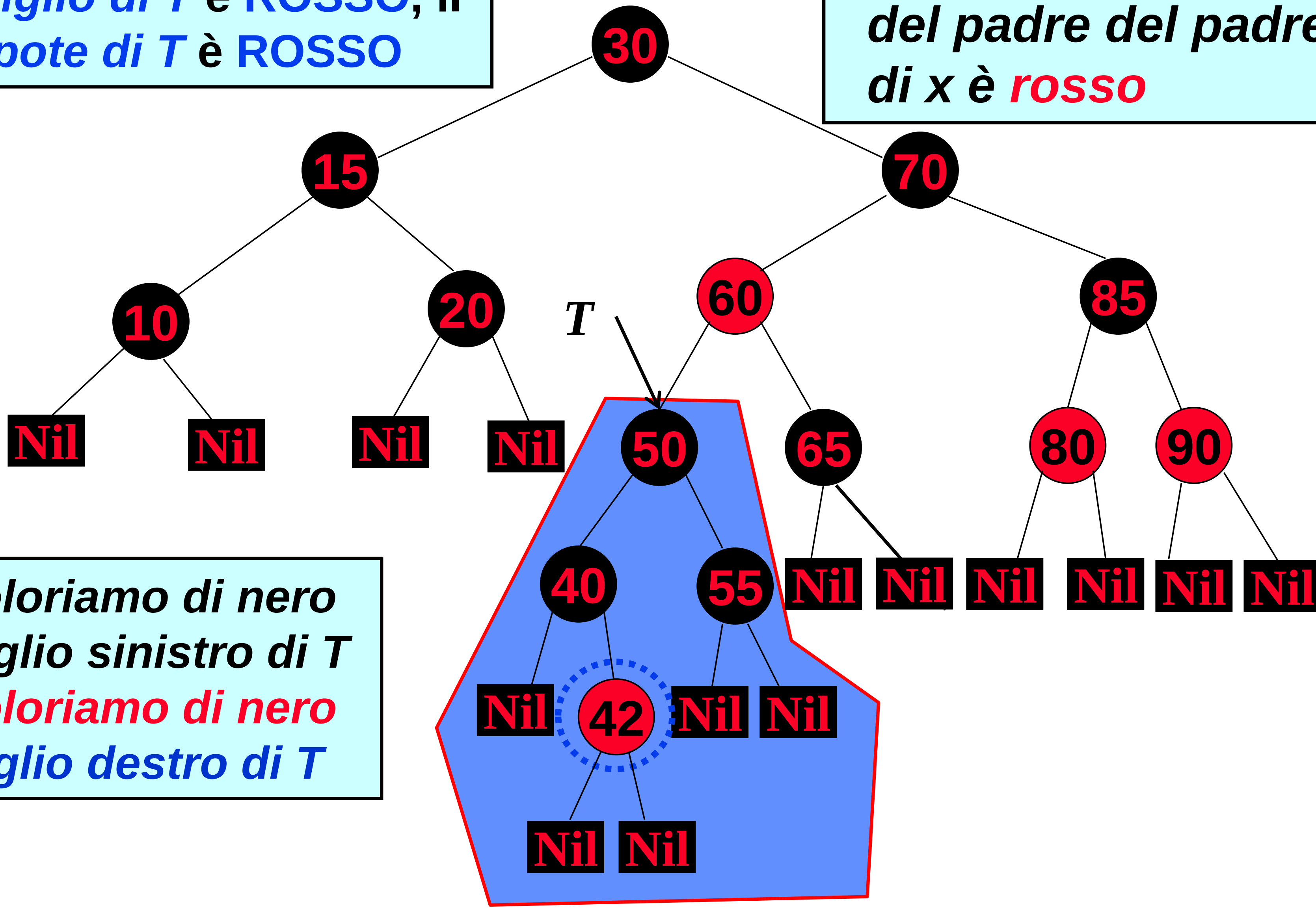


•Coloriamo di nero il figlio sinistro di T

Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il *nipote di T* è ROSSO

Caso I: ***L'altro figlio del padre del padre di x è rosso***

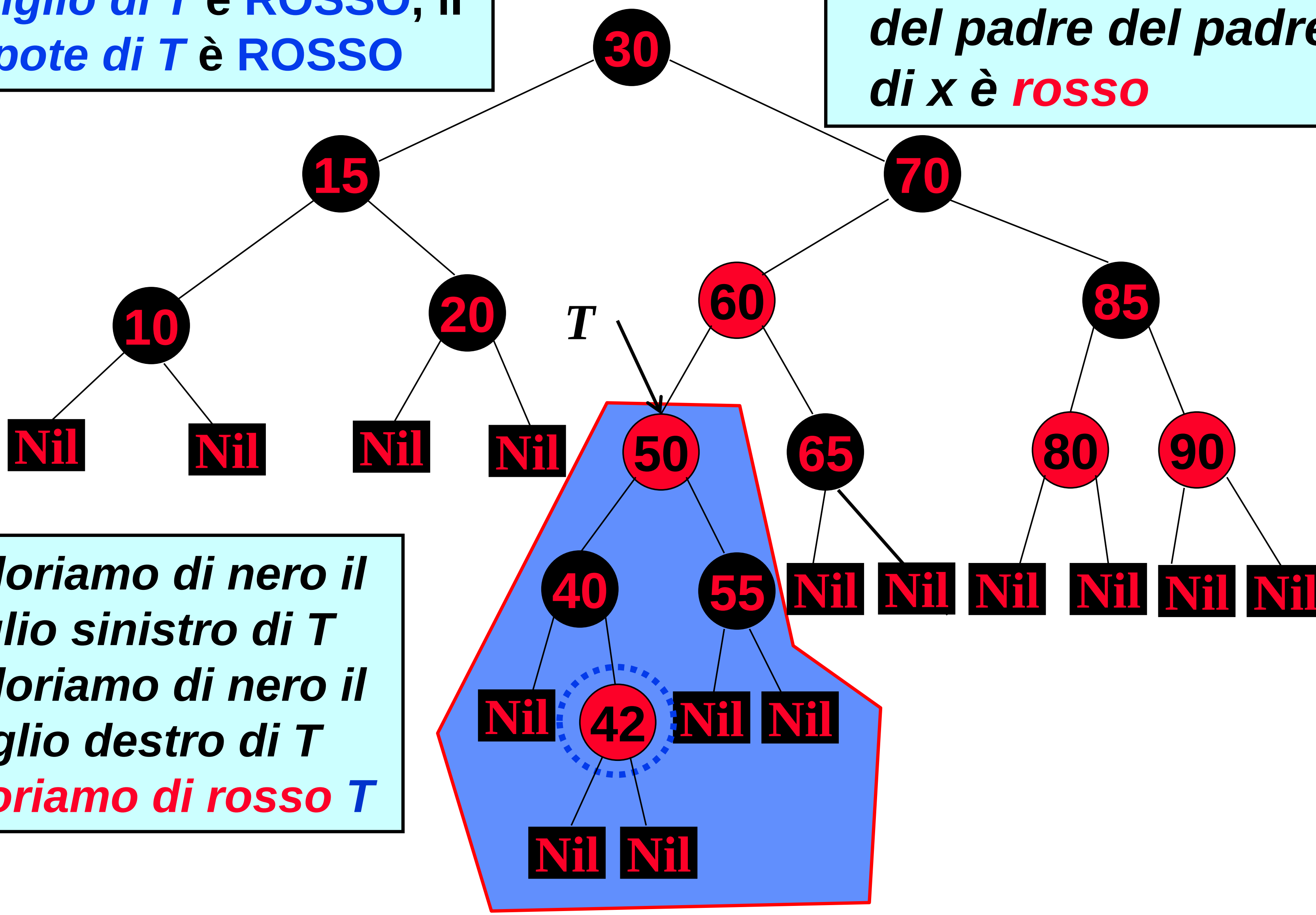


•Coloriamo di nero il figlio sinistro di T
•Coloriamo di nero il figlio destro di T

Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il *nipote di T* è ROSSO

Caso I: ***L'altro figlio del padre del padre di x è rosso***



- ***Coloriamo di nero il figlio sinistro di T***
- ***Coloriamo di nero il figlio destro di T***
- ***Coloriamo di rosso T***

Inserimento in alberi Red-Black: II

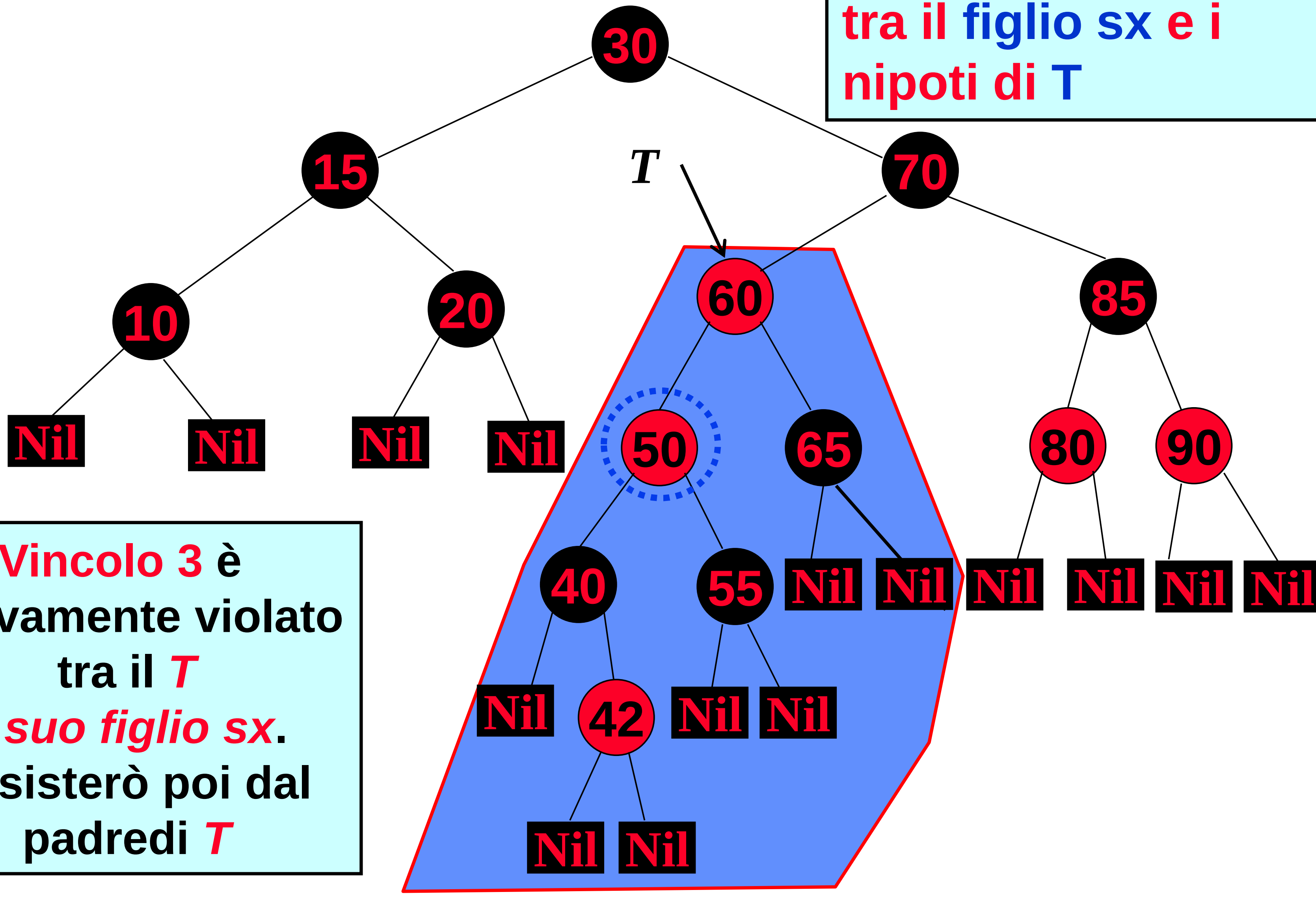
Se il *figlio di T* è ROSSO, il *nipote di T* è ROSSO

Caso I: *L'altro figlio del padre del padre di x è rosso*

Vincolo 3 è ripristinato
Vincolo 4 è ripristinato

Inserimento in alberi Red-Black: II

Nessuna violazione tra il figlio sx e i nipoti di T

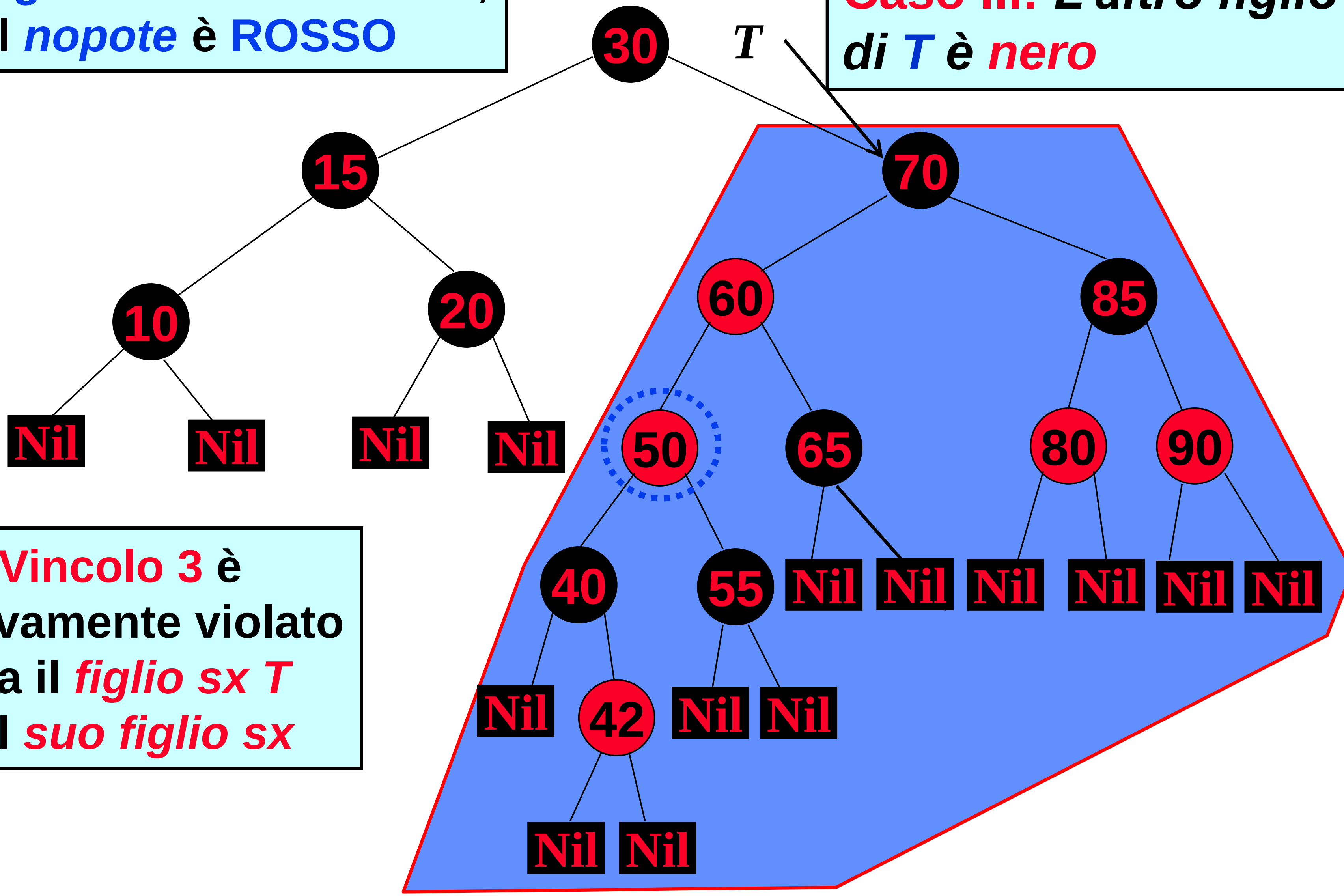


Vincolo 3 è nuovamente violato tra il *T* e *suo figlio sx*. Lo sisterò poi dal padredi *T*

Inserimento in alberi Red-Black: II

Se il *figlio sx di T* è ROSSO, il *nopote* è ROSSO

Caso III: ***L'altro figlio di T*** **è** ***nero***

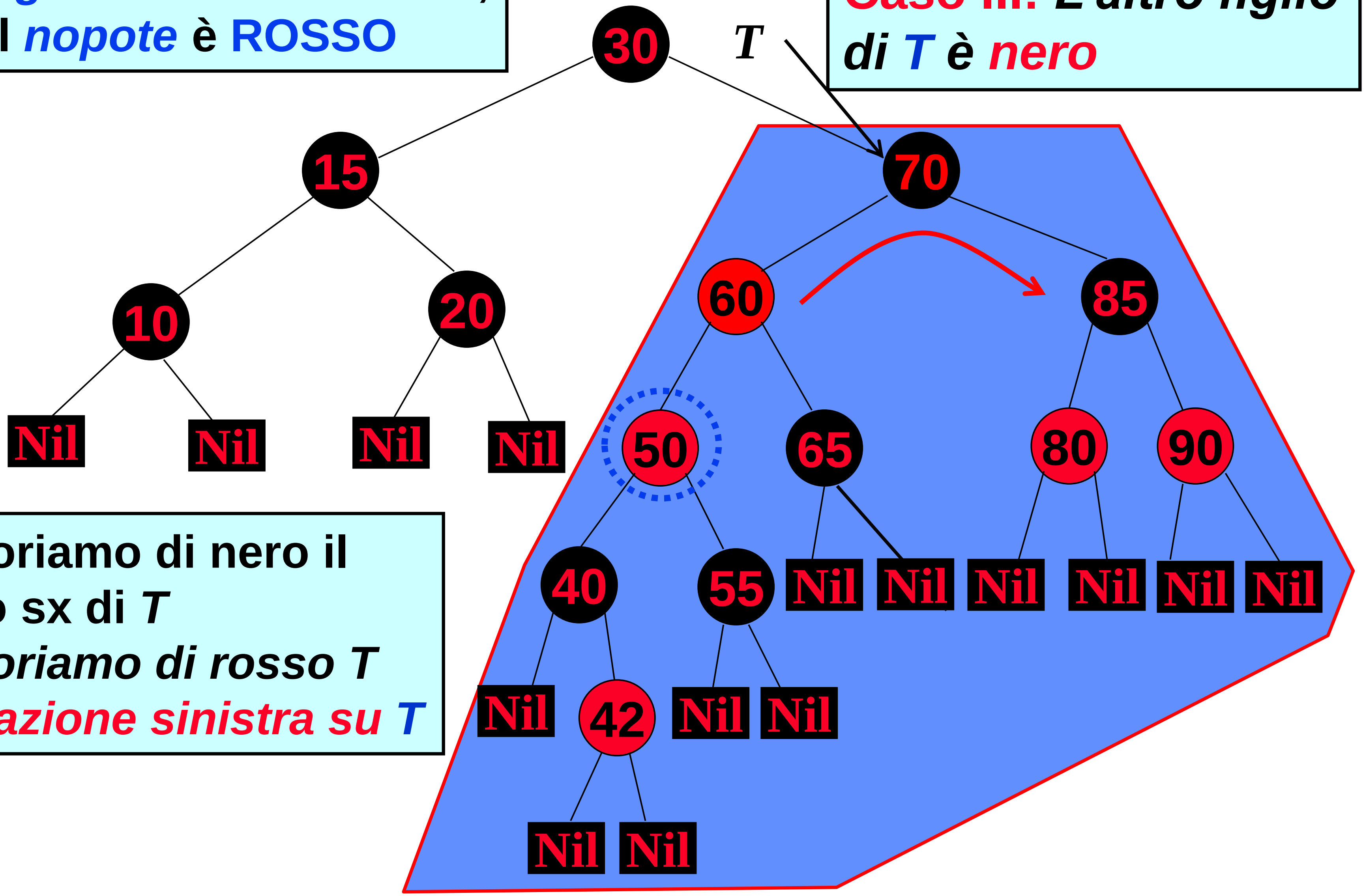


Vincolo 3 è nuovamente violato tra il *figlio sx T* e il *suo figlio sx*

Inserimento in alberi Red-Black: II

Se il *figlio sx di T* è ROSSO, il *nopote* è ROSSO

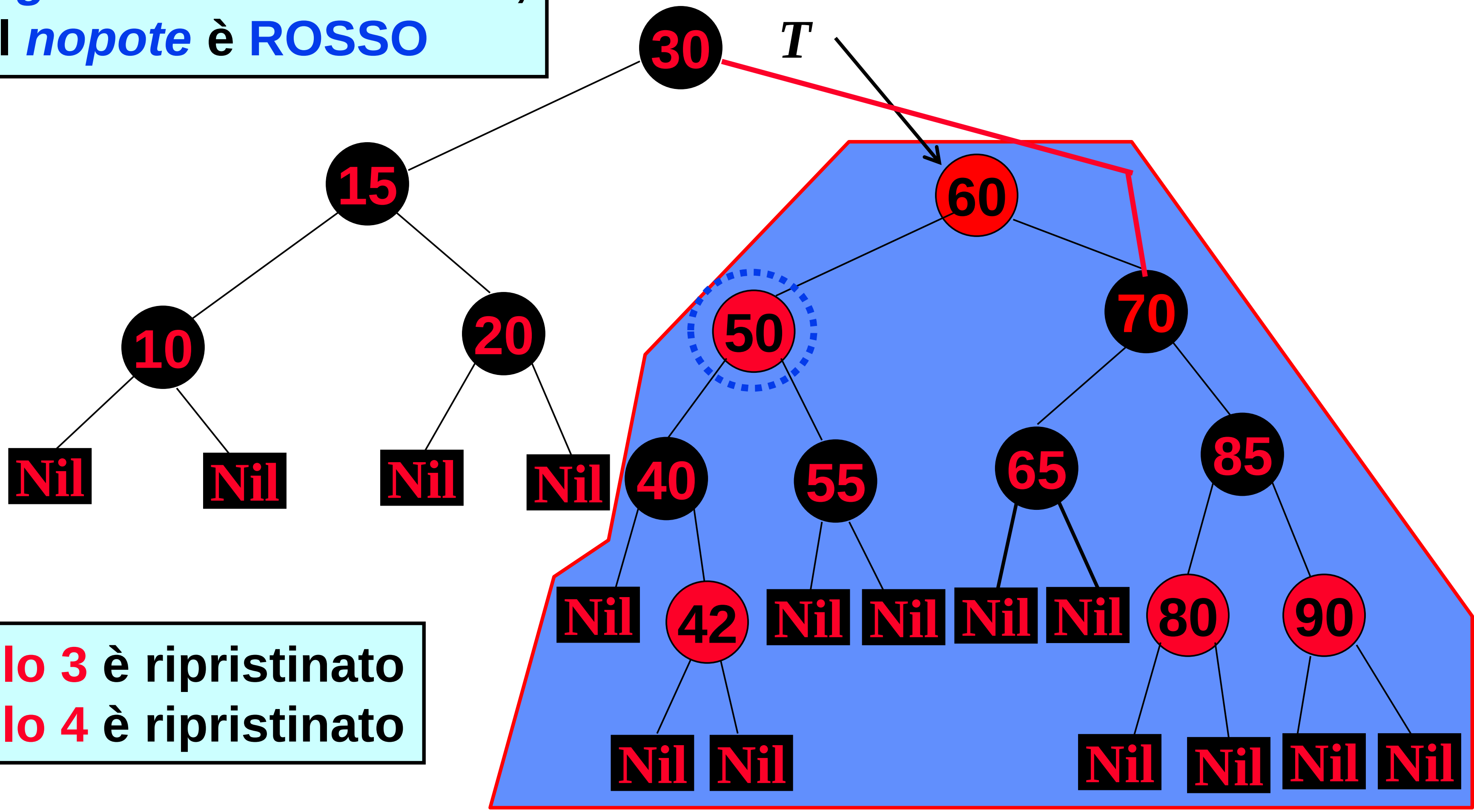
Caso III: *L'altro figlio di T è nero*



- Coloriamo di nero il figlio sx di *T*
- *Coloriamo di rosso T*
- *Rorazione sinistra su T*

Inserimento in alberi Red-Black: II

Se il *figlio sx di T* è ROSSO, il *nopote* è ROSSO

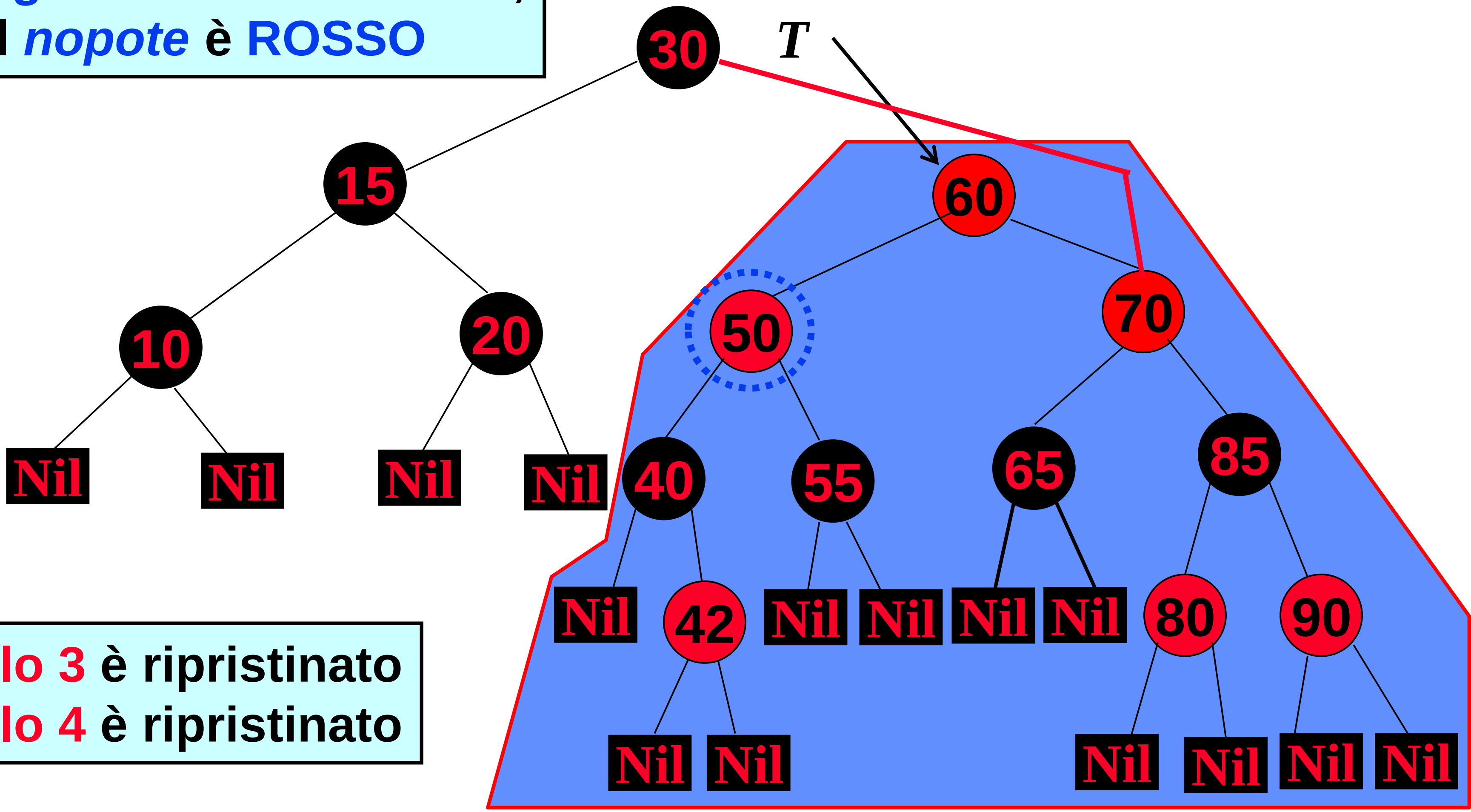


Vincolo 3 è ripristinato
Vincolo 4 è ripristinato

Ma attenzione: la radice del sottoalbero è cambiata

Inserimento in alberi Red-Black: II

Se il *figlio sx di T* è ROSSO, il *nopote* è ROSSO

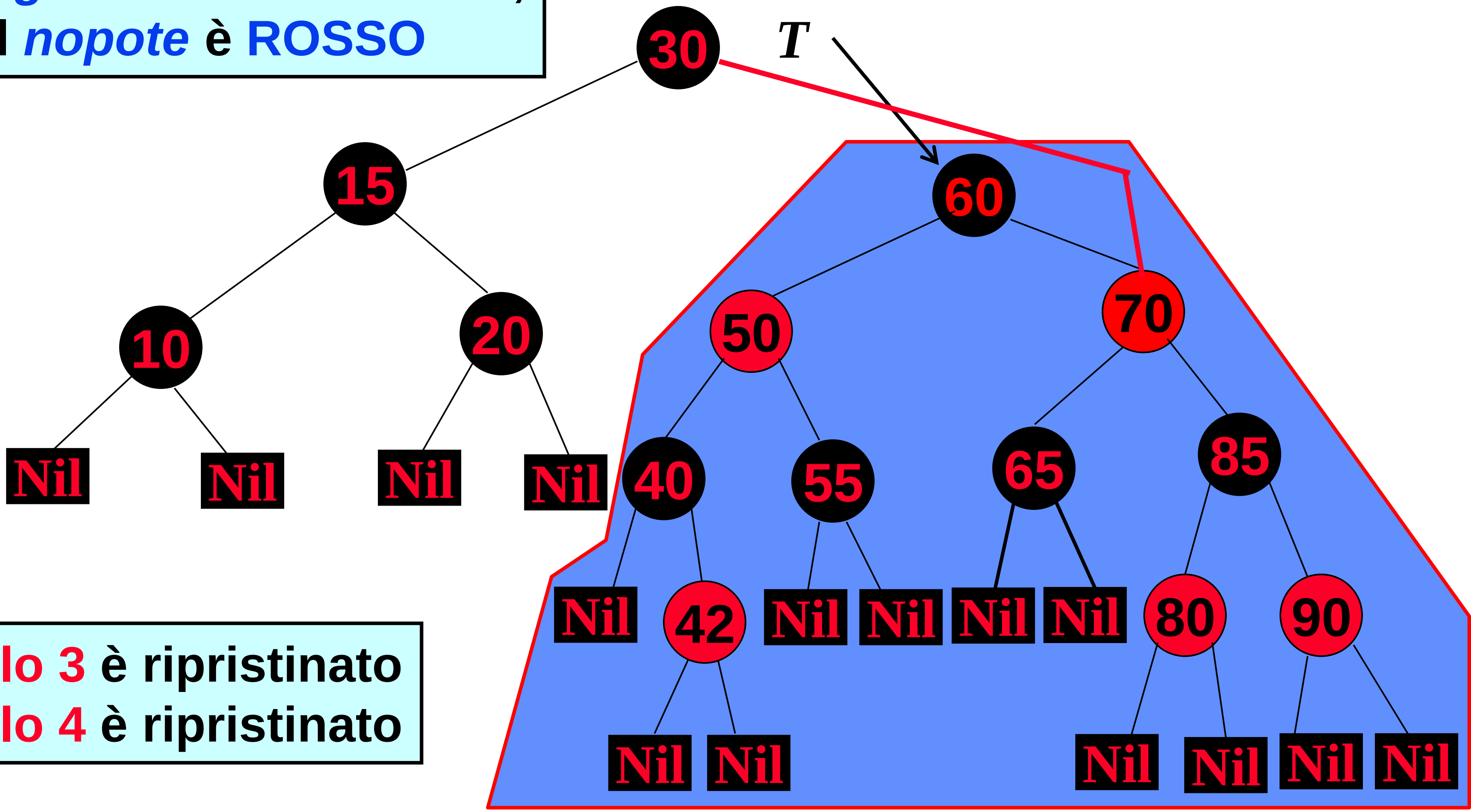


Vincolo 3 è ripristinato
Vincolo 4 è ripristinato

Ma attenzione: la radice del sottoalbero è cambiata

Inserimento in alberi Red-Black: II

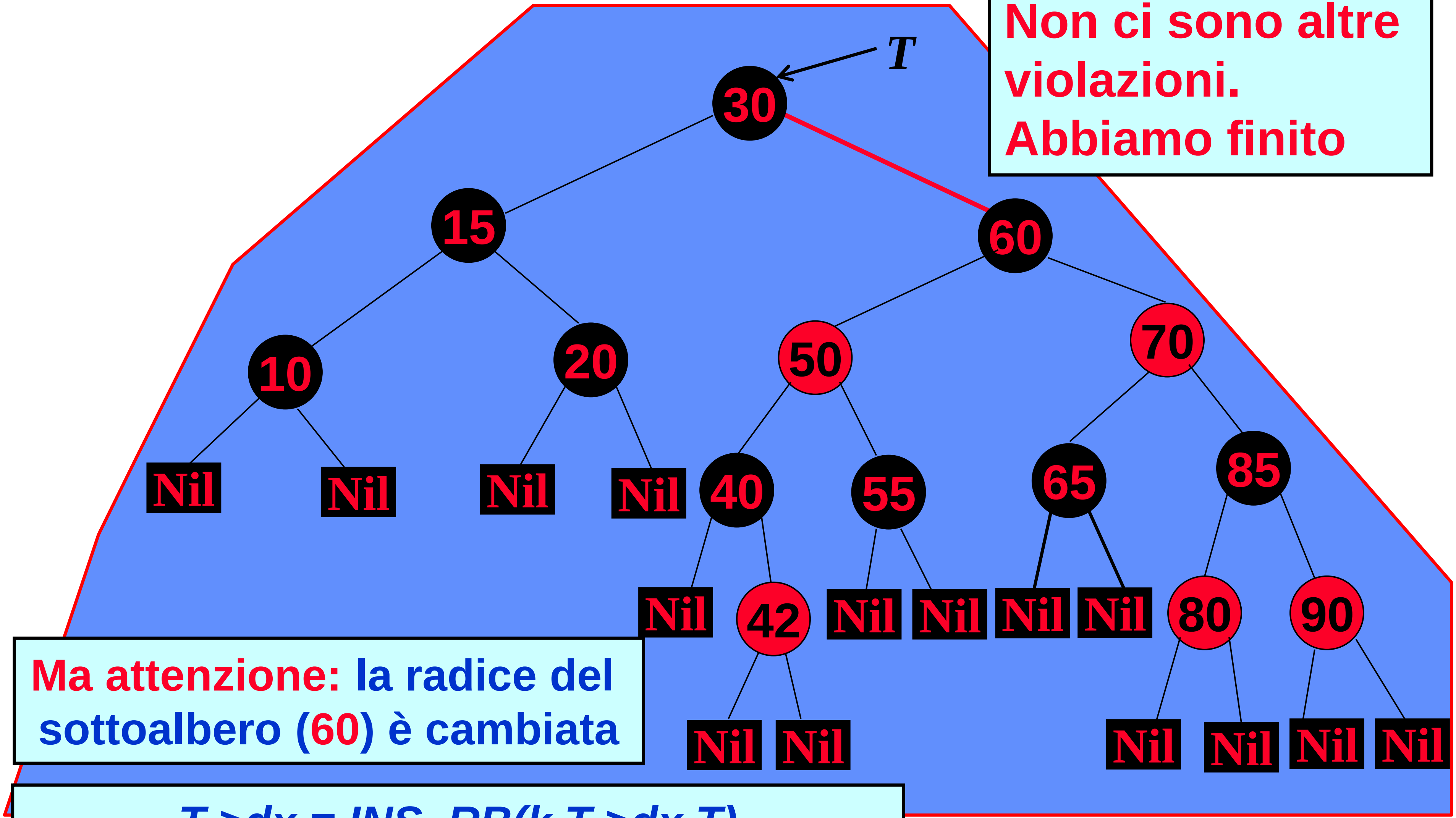
Se il *figlio sx di T* è ROSSO,
il *nopote* è ROSSO



Vincolo 3 è ripristinato
Vincolo 4 è ripristinato

Ma attenzione: la radice del sottoalbero è cambiata

Inserimento in alberi Red-Black: II



Non ci sono altre violazioni. Abbiamo finito

Ma attenzione: la radice del sottoalbero (60) è cambiata

T->dx = INS_RB(k,T->dx,T)

al ritorno della chiamata ricorsiva risistema l'albero dopo le rotazioni

Cancellazione in RB

- **L'algoritmo di cancellazione per alberi RB è costruito sull'algoritmo di cancellazione per alberi binari di ricerca**
  - **Useremo una variante con delle *sentinelle Nil[T]*, una per ogni nodo NIL per semplificare l'algoritmo**
- **Dopo la cancellazione si deve decidere se è necessario ribilanciare o meno**
- ***Le operazioni di ripristino del bilanciamento sono necessarie solo quando il nodo cancellato è nero! (perché?)***

Cancellazione in RB

- **Dobbiamo ribilanciare se il *nodo y cancellato* è *nero* (perché è *cambiata l'altezza nera*)**
- **Possiamo immaginare di *spostare di nero y* sul *nodo x* che *sostituisce* il *nodo cancellato y***
- **In tal modo la *cancellazione non vìola* più il *vincolo 4* ...**
- **… ma potrebbe violare il *vincolo 1* (*perché?*)**
- **Gli algoritmi `Canc-Bil-dx(T)` e `Canc-Bil-sx(T)` tentano di ripristinare il *vincolo 1* con rotazioni e cambiamenti di colore:**
  - **ci sono *4 casi possibili* (e 4 simmetrici)**

```
Canc-RB(T,K)
  IF not IS-NIL(T) THEN
     IF T->key < K THEN
        T->dx = Canc-RB(T->dx,K)
        T = Canc-Bil-dx(T)
     ELSE IF T->key > K THEN
        T->sx = Canc-RB(T->sx,K)
        T = Canc-Bil-sx(T)
     ELSE
        T = Canc-Radice-RB(T)
  RETURN T
```

```
Canc-Radice-RB(T)
  IF IS-NIL(T->sx) || IS-NIL(T->dx) THEN
     tmp = T
     IF IS-NIL(T->sx) THEN
        T = T->dx
     ELSE IF IS-NIL(T->dx) THEN
        T = T->sx
     IF tmp->color = black THEN
         Propagate-Black(T)
  ELSE
     tmp = Stacca-Min-RB(T->dx,T)
     T->Key = tmp->Key
     T = Canc-Bil-dx(T)
  dealloca tmp
  RETURN T
```

```
Propagate-Black(T)
  IF T->color = red THEN
     T->color = black
  ELSE
     T->color = d-black
```

```
Stacca-Min-RB(T,P)
 IF not IS-NIL(T) THEN
   IF not IS-NIL(T->sx) THEN
     tmp = Stacca-Min-RB(T->sx,T)
     IF T = P->sx THEN
        P->sx = Canc-Bil-sx(T)
     ELSE
        P->dx = Canc-Bil-dx(T)
     T = tmp
   ELSE
     tmp = T
     IF T = P->sx THEN
        P->sx = T->dx
     ELSE
        P->dx = T->dx
     IF T->color = black
        Propagate-Black(T->dx)
 return tmp
```

```
Propagate-Black(T)
  IF T->color = red THEN
     T->color = black
  ELSE
     T->color = d-black
```

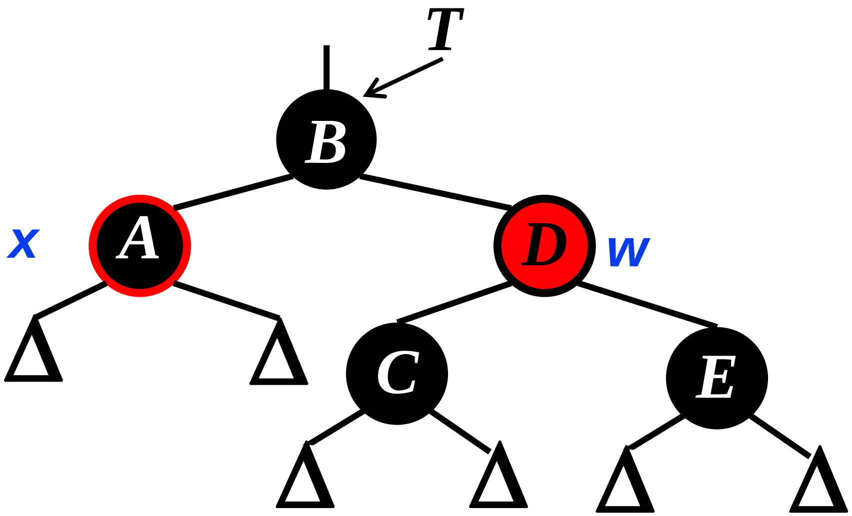
Cancellazione in RB: casi

- **Abbiamo visto i *4 casi possibili* quando la cancellazione è avvenuta a *sinistra***

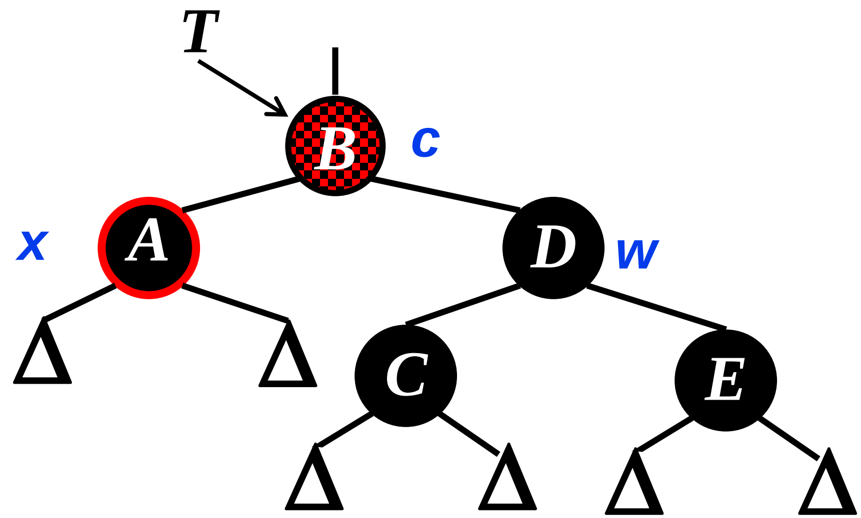
- **Esistono anche i *4 casi simmetrici* (con destro e sinistro scambiati) quando la cancellazione è avvenuta a *destra***

Esercizio: ***Illustrare i 4 casi simmetrici e scrivere lo pseudo-codice che li gestisce***

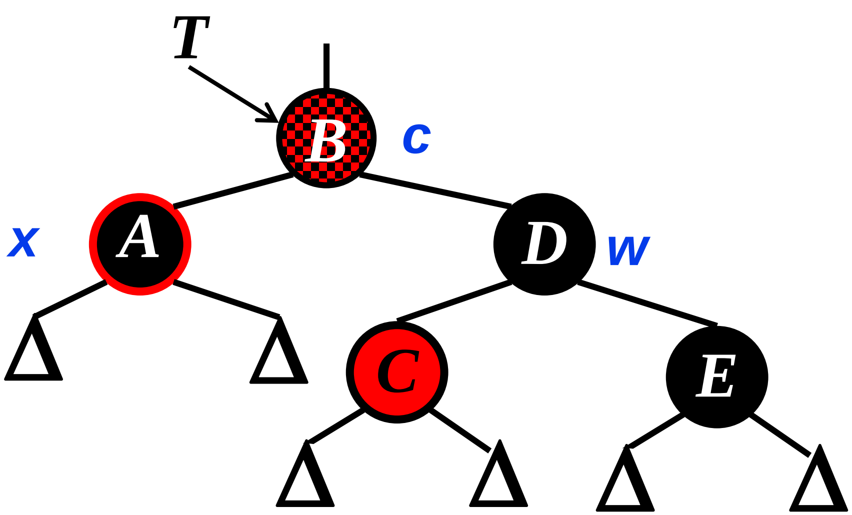
caso 1



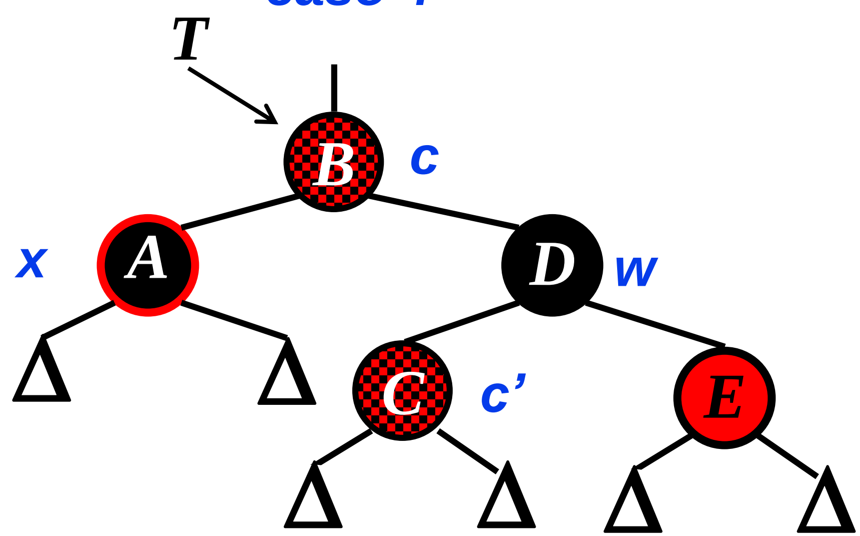
caso 2



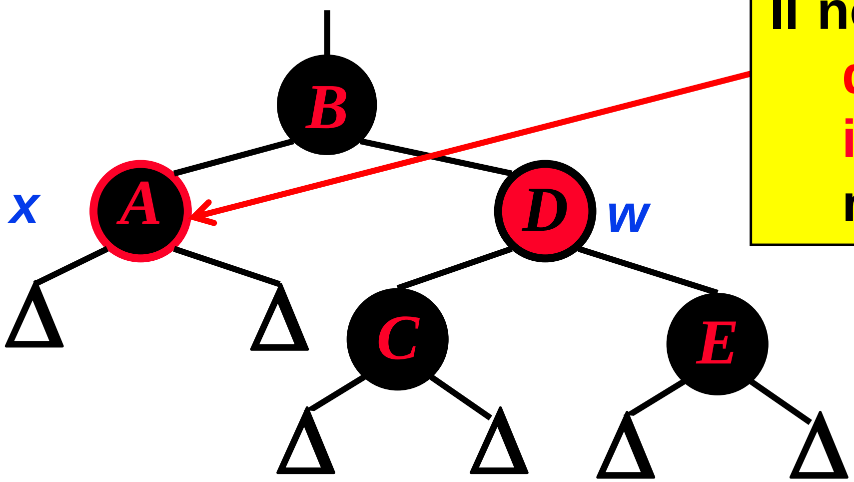
caso 3



caso 4



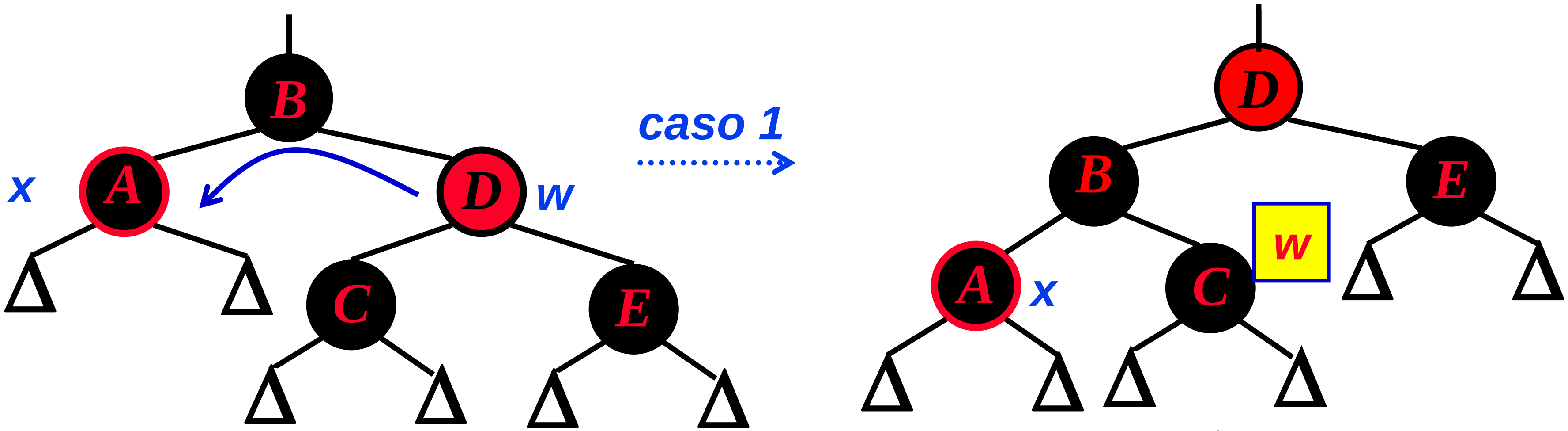
Cancellazione in RB: caso 1

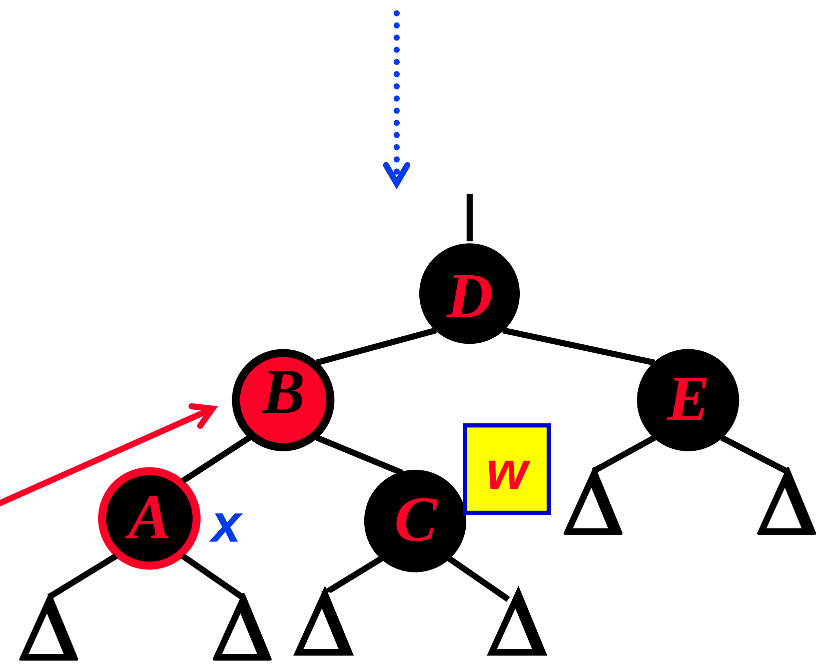


Il nodo x (cioè *A* nell'esempio) è bordato di rosso ad indicare che è il nodo con il colore nero in più da ridistribuire nell'albero

- Il *fratello w* di *x* è *rosso*
- *w* deve avere figli *neri*

Cancellazione in RB: caso 1

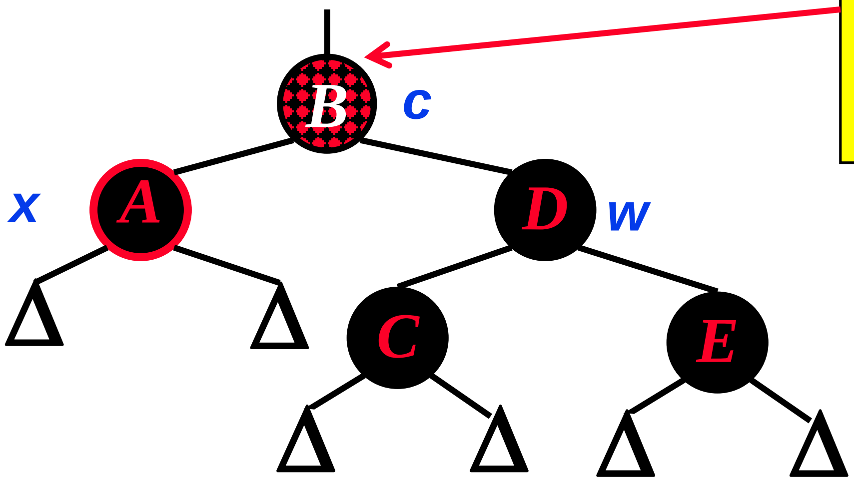




- **Il *fratello w* di *x* è *rosso***
- ***w* deve avere figli *neri***
- **cambiamo i colori di *w* e del padre di *x* e li ruotiamo tra loro**

Non violiamo né il *vincolo 3* né il 4 e ci riduciamo ad uno degli altri casi, chiamando nuovamente il bilanciamento sul nodo B

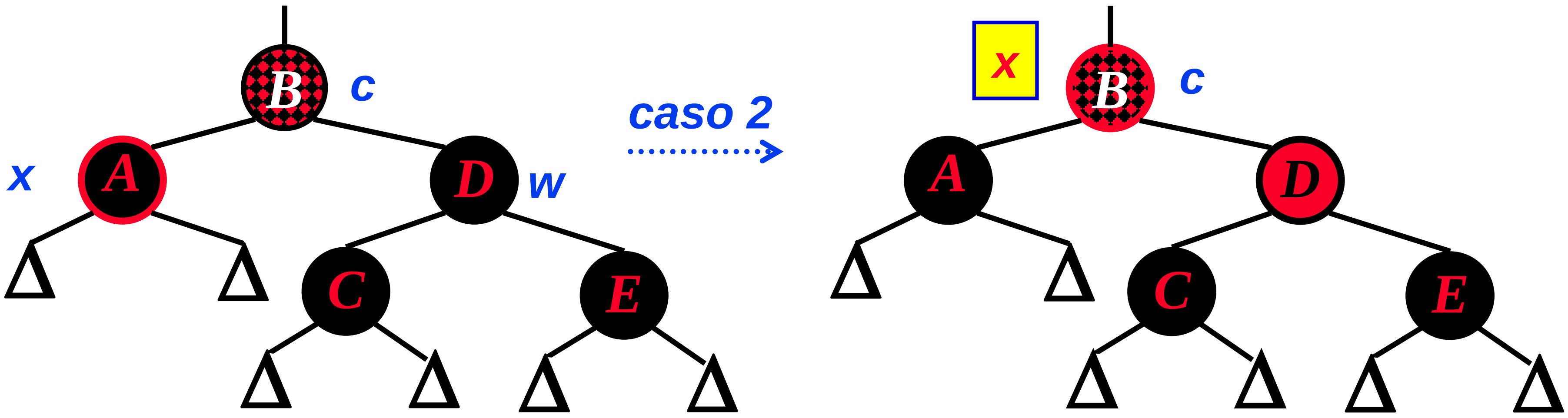
Cancellazione in RB: caso 2



Il nodo c (cioè *B* nell'esempio) può essere sia rosso che nero!

- **Il *fratello w* di *x* è *nero***
- ***w* ha in questo caso entrambi i figli *neri***

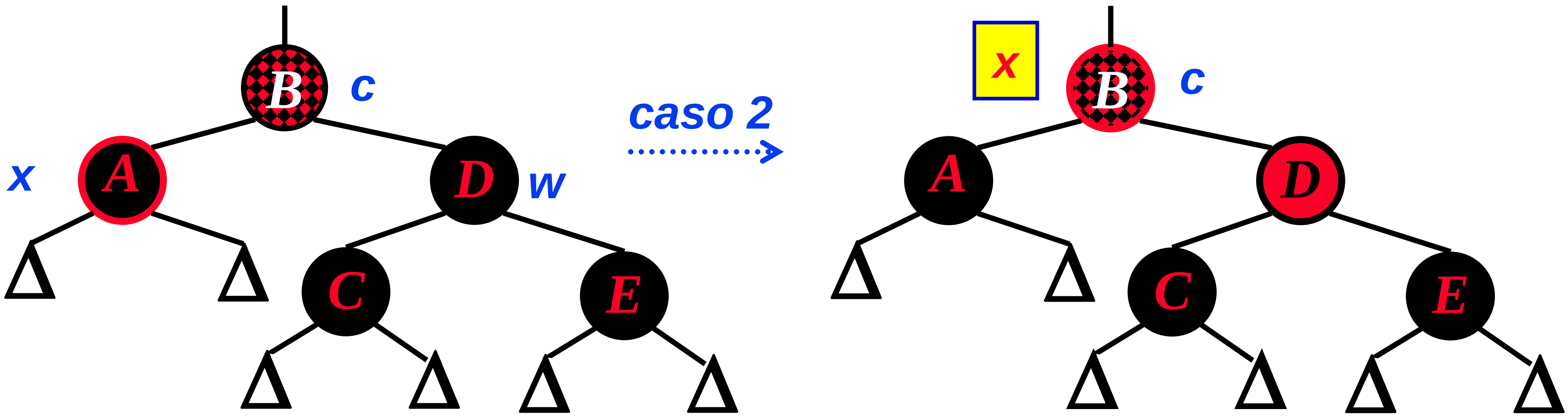
Cancellazione in RB: caso 2



- Il *fratello* *w* di *x* è *nero*
- *w* ha in questo caso entrambi i figli *neri*
- cambiamo il colore di *w* e il nuovo *x* diventa il padre

Spostiamo il *nero in più* da *x* a *c* (il *padre*) e togliamo il nero da *w* per rispettare *vincolo 4* lungo il sottoalbero destro di *c*

Cancellazione in RB: caso 2

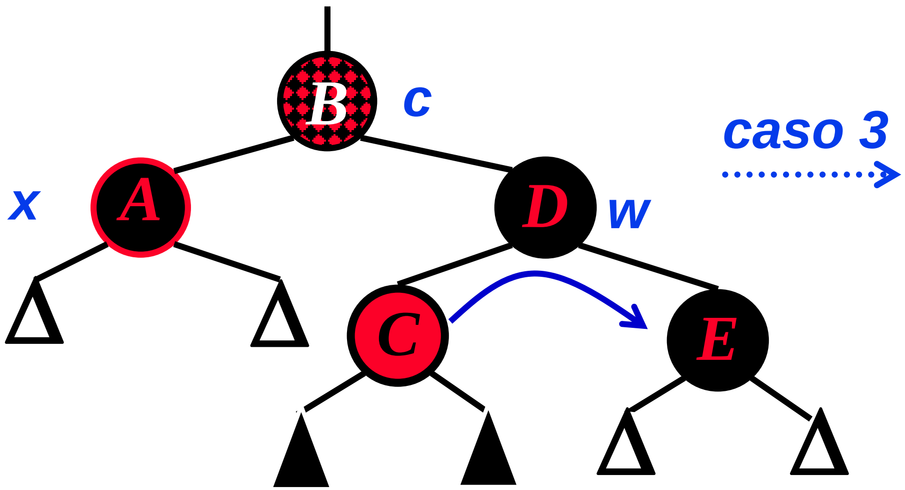


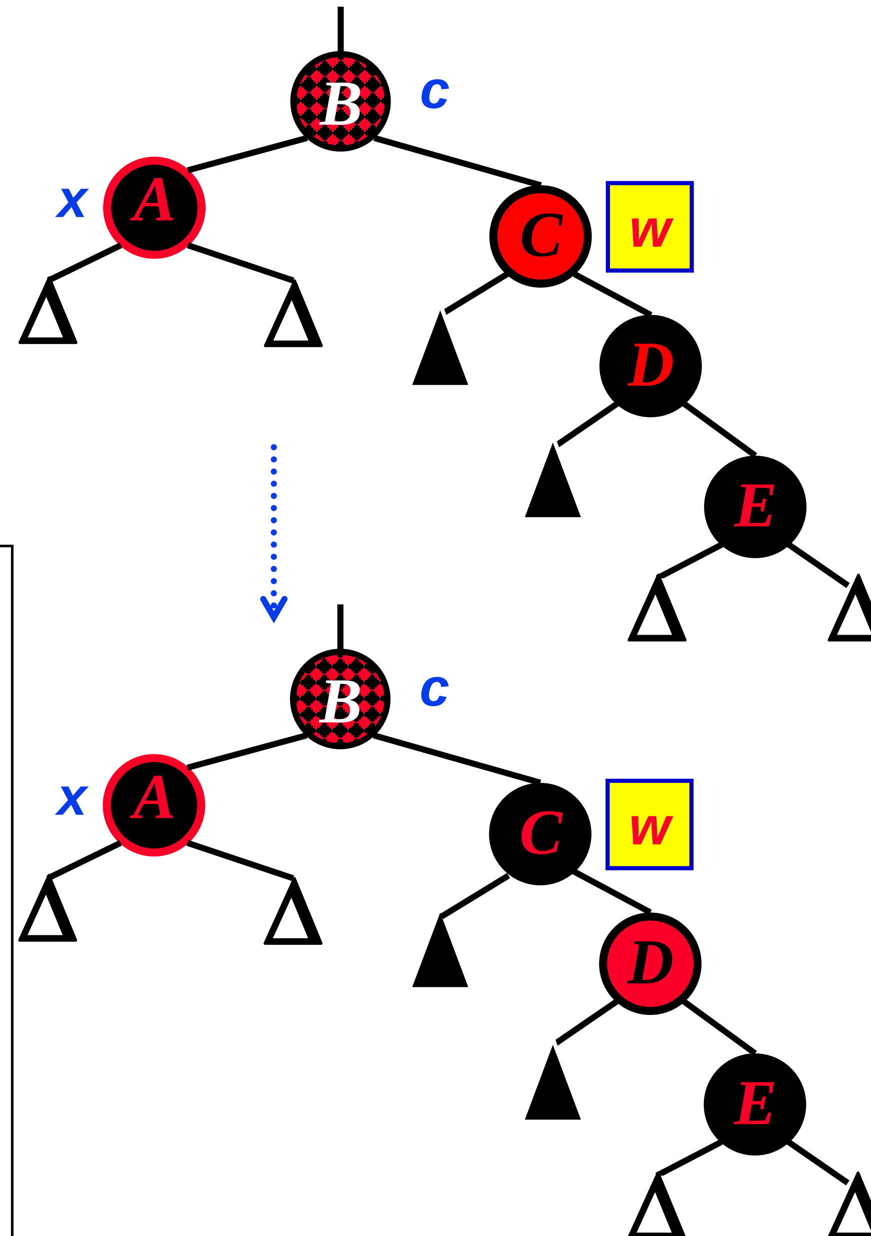
- Il *fratello w* di *x* è *nero*
- *w* ha in questo caso entrambi i figli *neri*
- cambiamo il colore di *w* e il nuovo *x* diventa il padre

Spostiamo il *nero in più* da *x* a *c* (il *padre*) e togliamo il nero da *w* per rispettare *vincolo*

Se si arriva dal *caso 1*, B è sicuramente rosso, quindi dopo il *caso 2* non c'è più bisogno di ribilanciare, perché ora B ha un solo nero (il nero in più).

Cancellazione in RB: caso 3

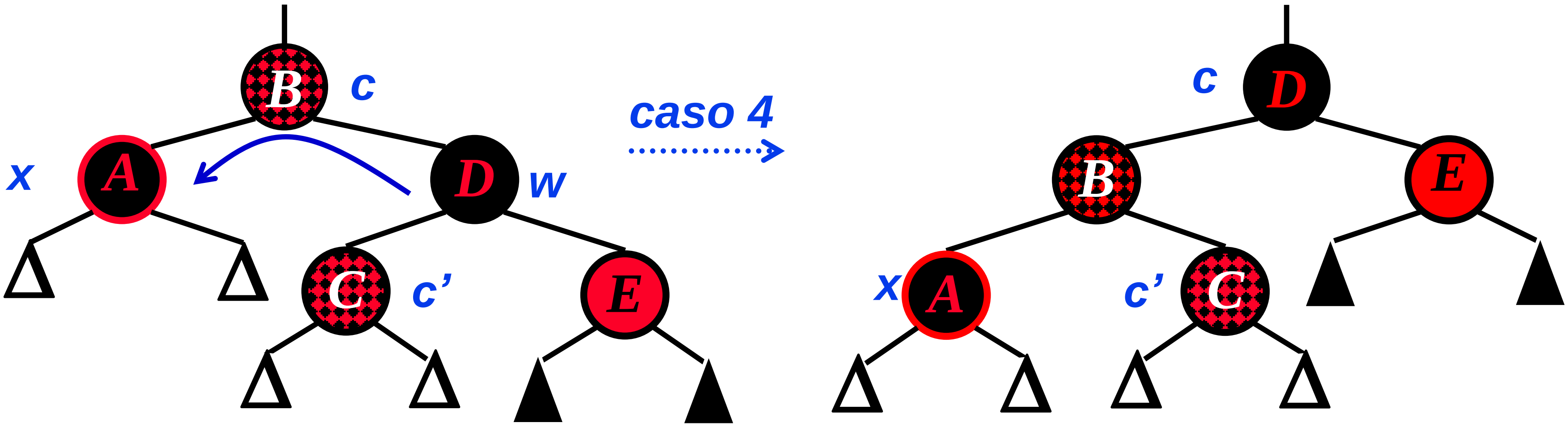


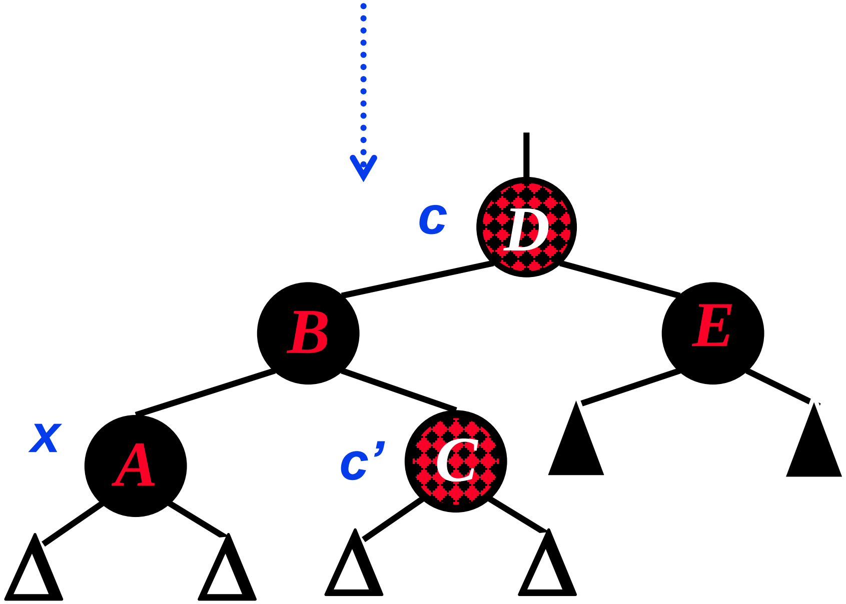


- Il *fratello w* di *x* è *nero*
- *w* ha in questo caso solo il figlo sinistro *rosso*
- *ruotiamo w* col suo *figlio sinistro*
- cambiamo il colore di *w* e quello del destro di *w*

Non violiamo alcun *vincolo (3* e *4)* e il nuovo fratello si *x* è ora nero con figlio sinistro nero, quindi andiamo nel *caso 4*

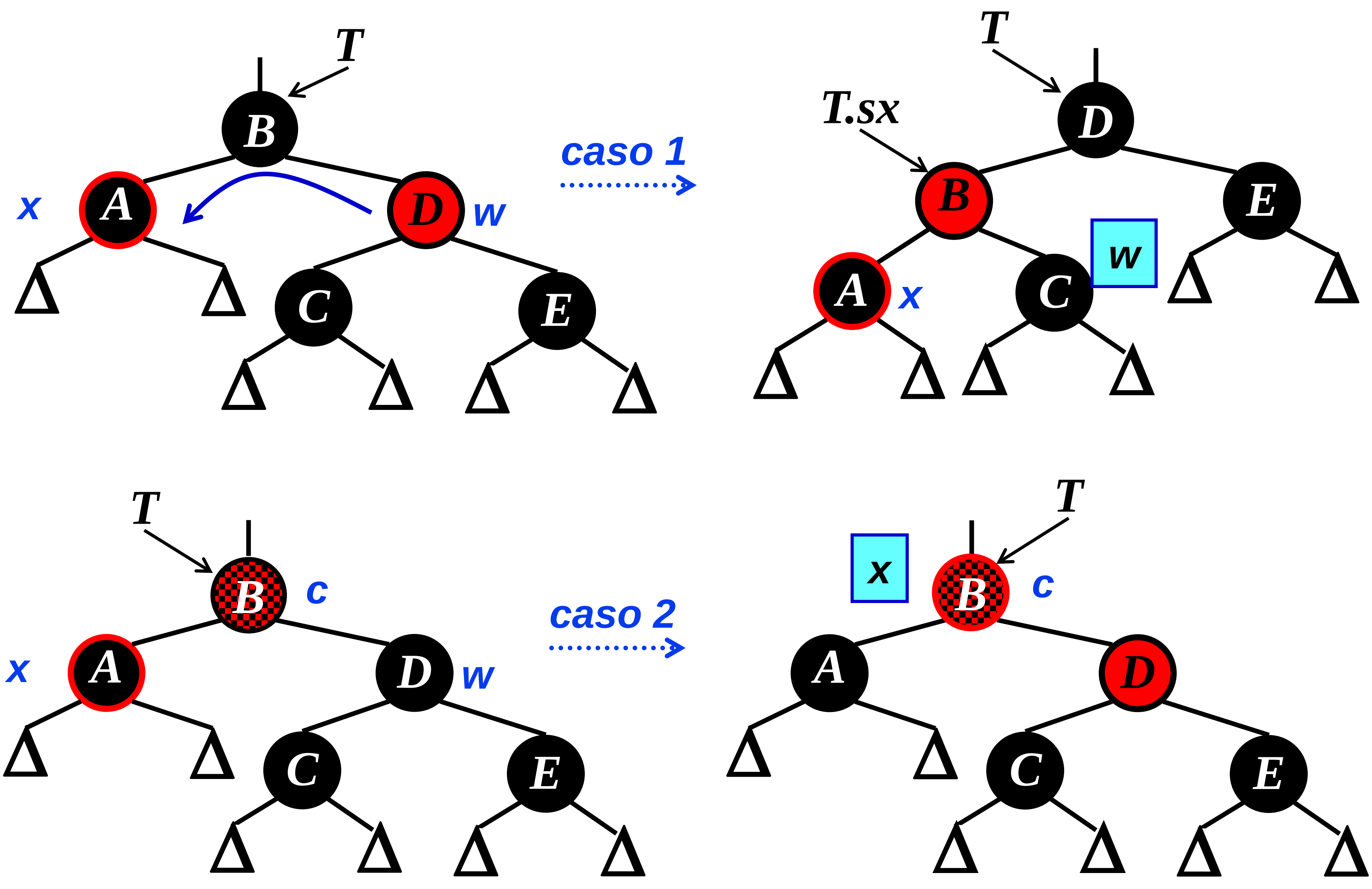
Cancellazione in RB: caso 4

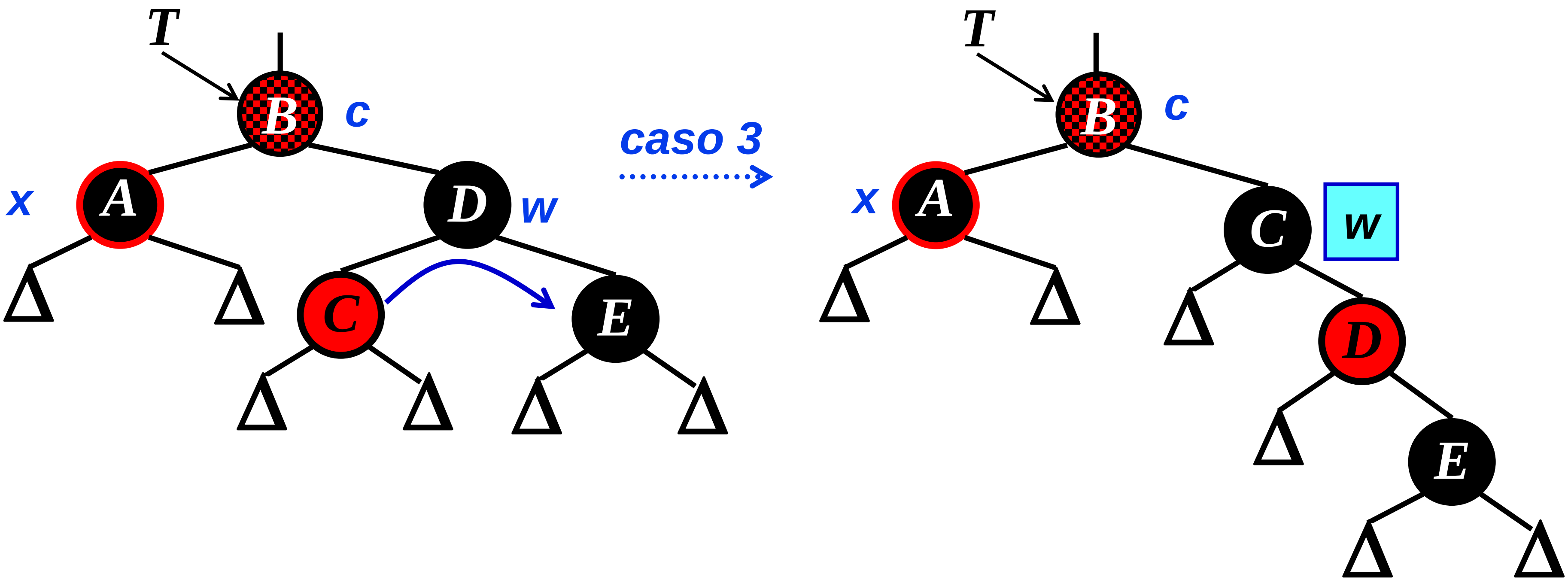


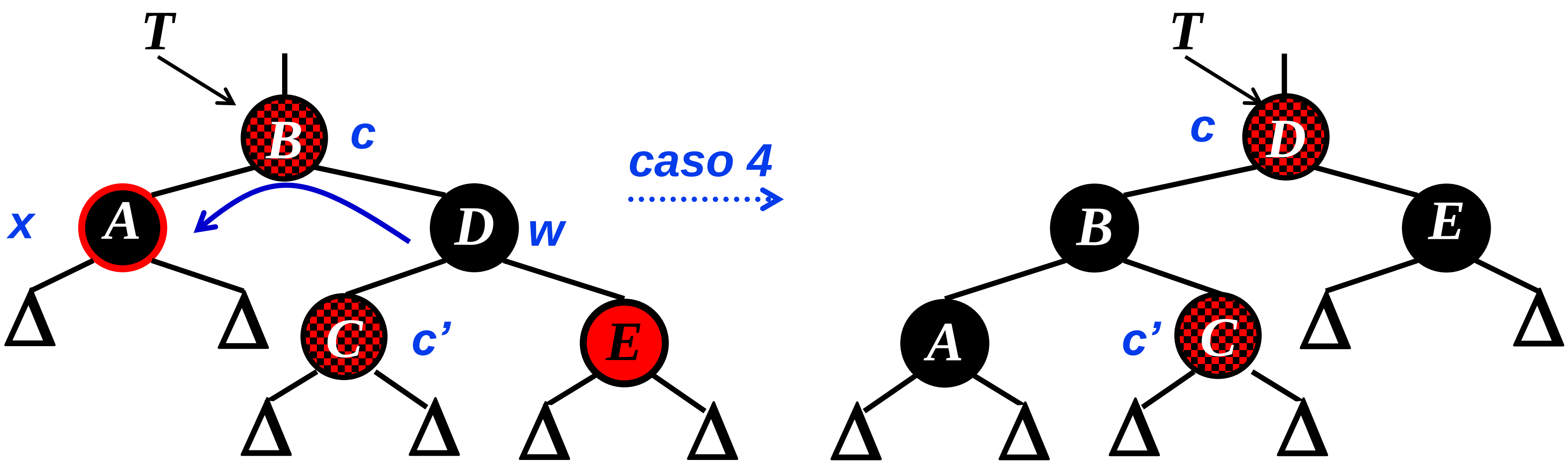


- Il ***fratello w*** di ***x*** è ***nero***
- ***w*** **ha in questo caso solo il figlo destro** ***rosso***
- **eseguiamo una** ***rotazione del padre di x con w*** **e cambiamo i colori** ***opportunamente*****,** ***eliminando il nero in più su x***

Non violiamo **alcun** ***vincolo (3*** **e** ***4)*** **e abbiamo** ***finito!***


T
B
x
A
D
w
C
E
caso 1
T
T.sx
D
B
E
A
x
C
w
T
B
c
x
A
D
w
C
E
caso 2
T
x
B
c
A
D
C
E


T
B
c
caso 3
x
A
D
w
C
E
T
B
c
x
A
C
w
D
E


T
B
c
caso 4
x
A
D
w
C
c'
E
T
D
c
B
E
A
c'
C

Cancellazoine in RB

```
Cancella_Bil_sx(T)
   IF ha_figli(T)
        v = Violazione_sx(T->sx, T->dx)
        /* nessuna violazione se v = 0 */
        CASE v OF
                1: T = Canc_bil_1_sx(T);
                   T->sx = Cancella_Bil_sx(T->sx);
                2: T = Canc_bil_2_sx(T);
                3: T = Canc_bil_3_sx(T);
                4: T = Canc_bil_4_sx(T);
   return T;
```

```
Violazione-sx(X,W)
 viola = 0
 IF X->color = d-black THEN
   IF W->color=red THEN
      viola = 1
   ELSE IF W->dx = black &
           W->sx = black  THEN
      viola = 2
   ELSE IF W->dx = black THEN
      viola = 3
   ELSE /* W->dx = red */
       viola = 4
 return viola
```

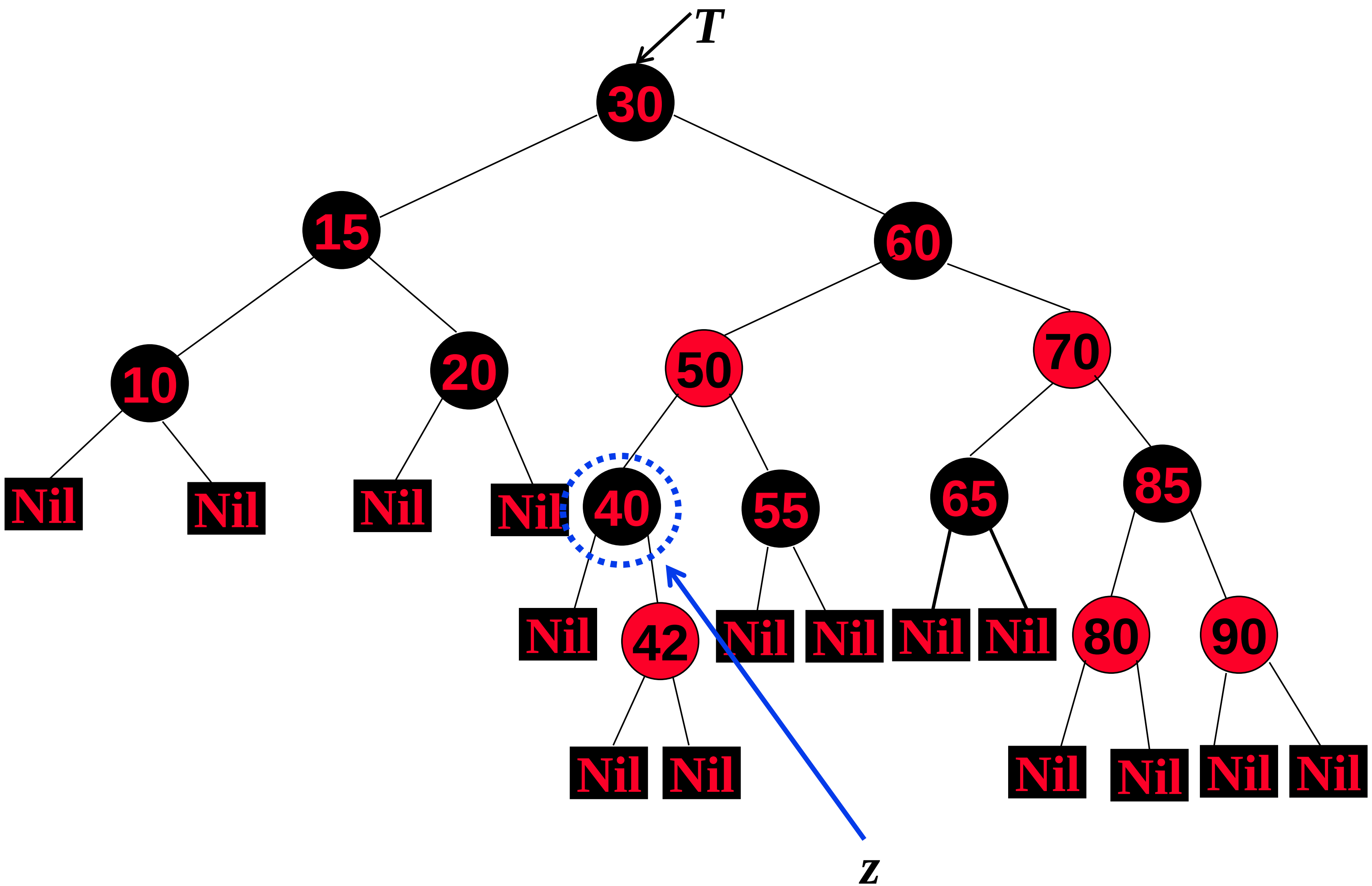
```
Canc-Bil-1-sx(T)
   T = ruota-dx(T)
   T->color = black
   T->sx->color = red
   return(T)
```

```
Canc-Bil-2-sx(T)
   T->dx->color = red
   T->sx->color = black
   propagate-black(T)
   return T
```

```
Canc-Bil-3-sx(T)
   T->dx = ruota-sx(T->dx)
   T->dx->color = black
   T->dx->dx->color = red
   T = Canc-Bil-4-sx(T)
   return T
```

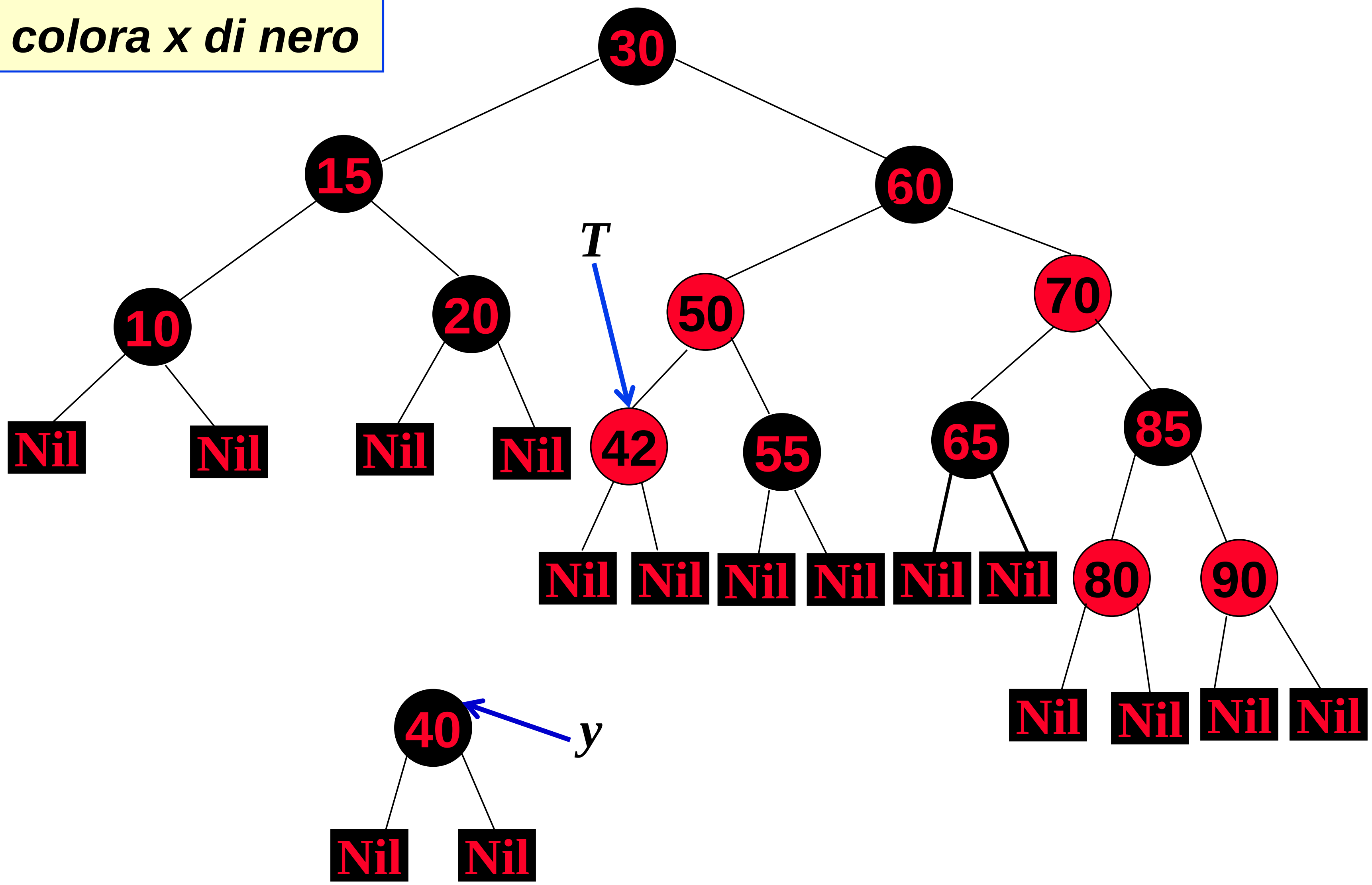
```
Canc-Bil-4-sx(T)
   T = ruota-sx(T)
   T->dx->color = T->color
   T->color = T->sx->color
   T->sx->color = black
   T->sx->sx->color = black
   return T
```

Cancellazoine in RB: esempio



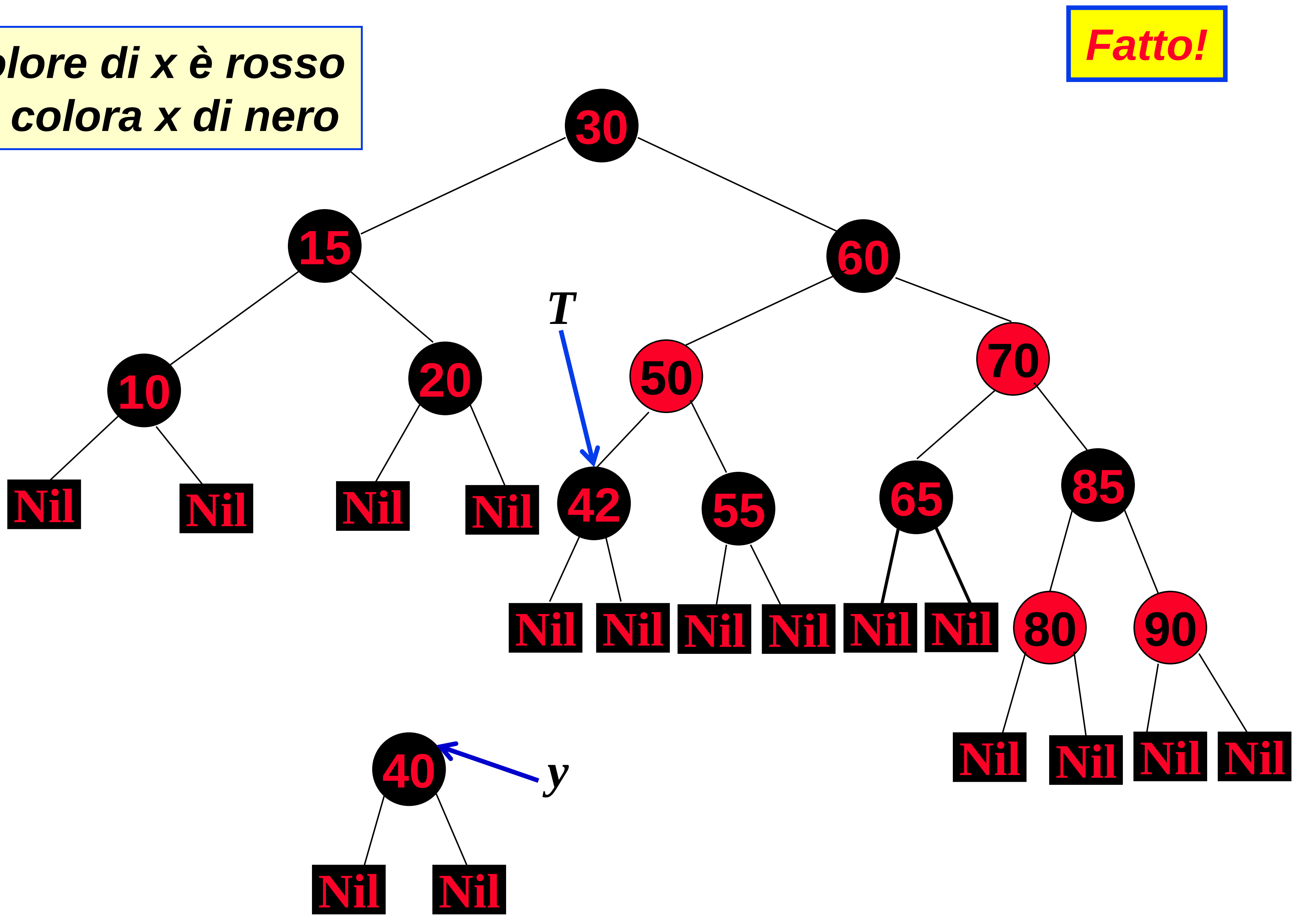
Cancellazoine in RB: esempio

Il colore di x è rosso e si colora x di nero

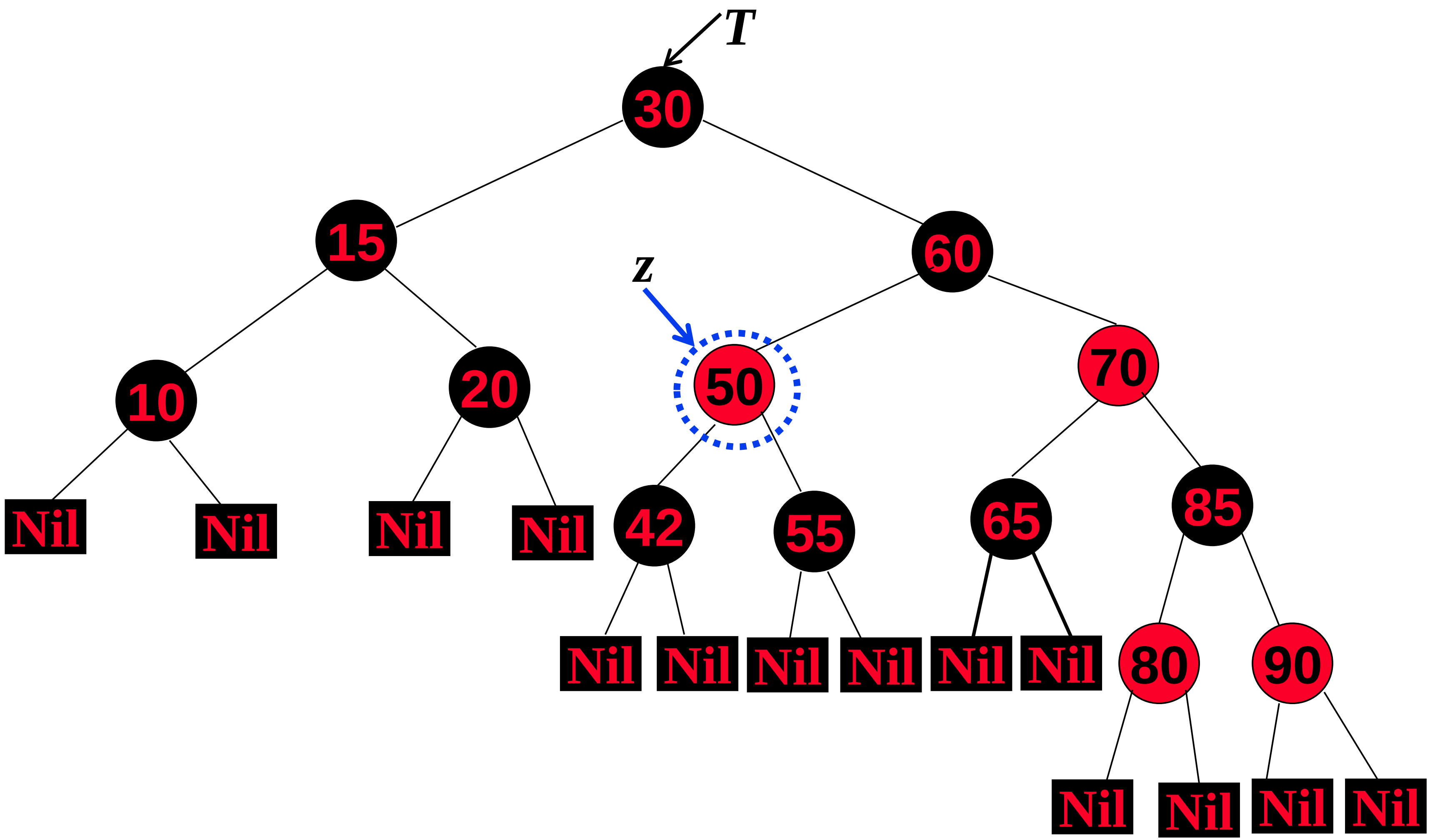


Cancellazoine in RB: esempio

Il colore di x è rosso e si colora x di nero



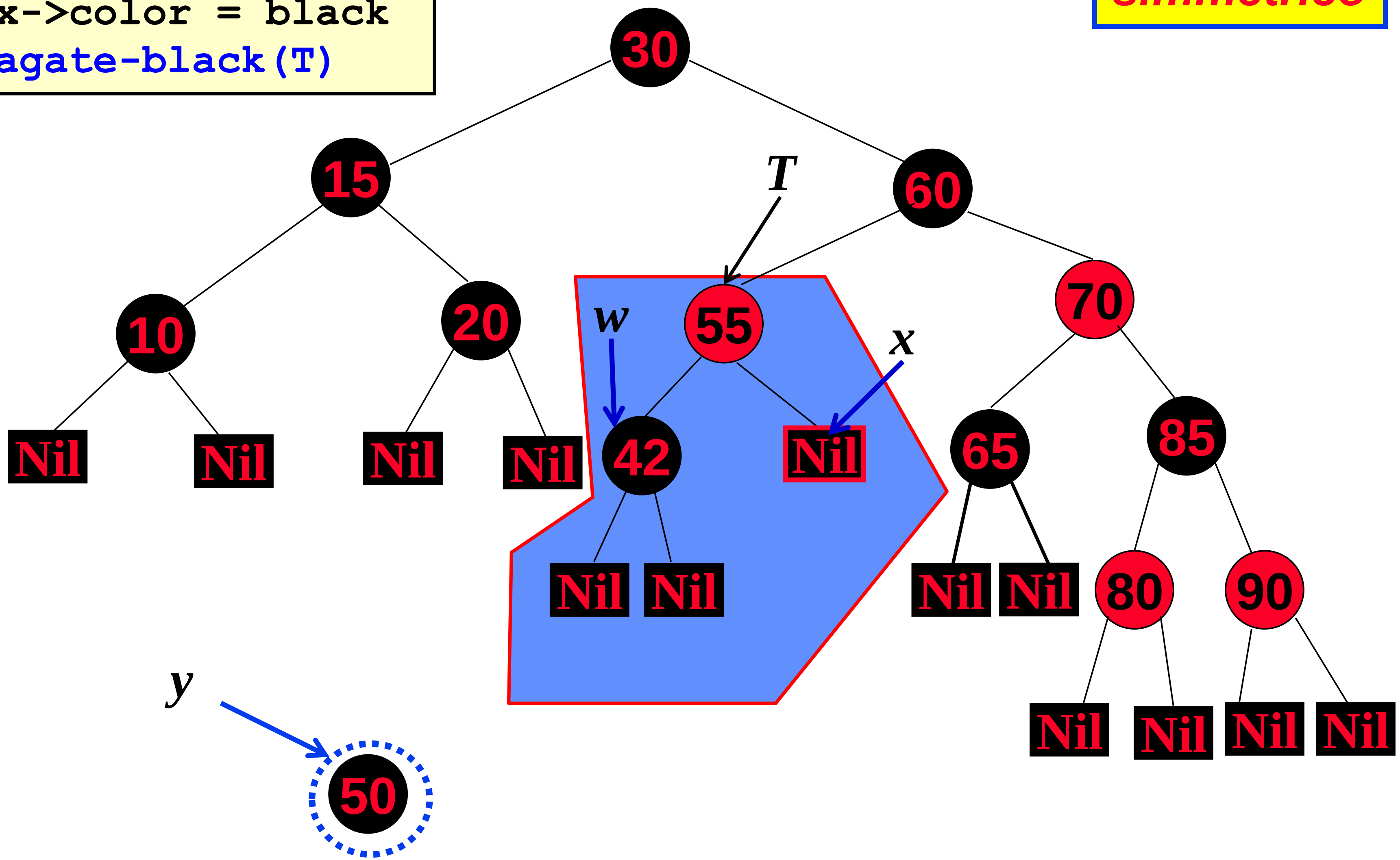
Cancellazoine in RB: esempio II



Cancellazoine in RB: esempio II

Caso II simmetrico

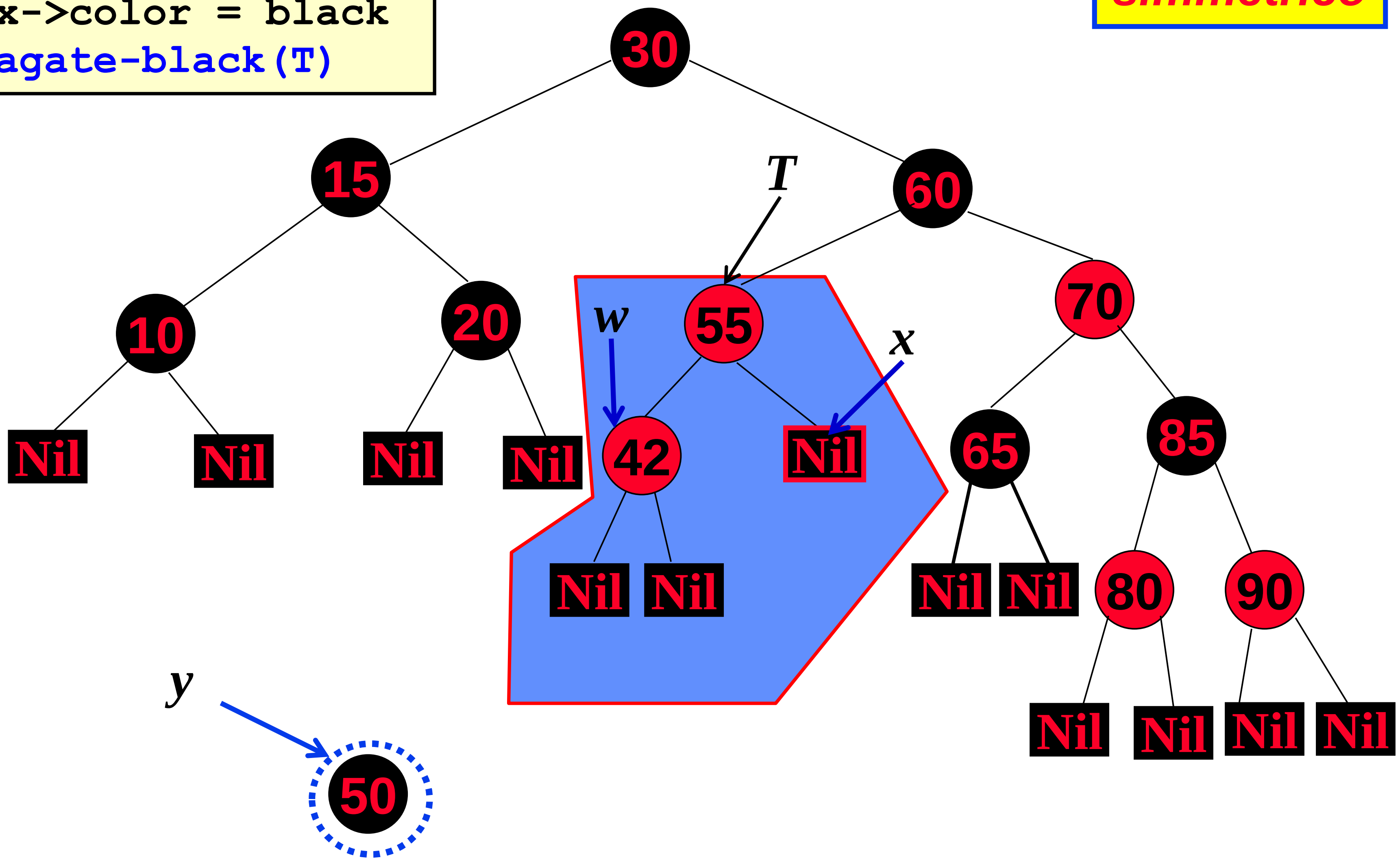
```
T->sx->color = red
T->dx->color = black
propagate-black(T)
```



Cancellazoine in RB: esempio II

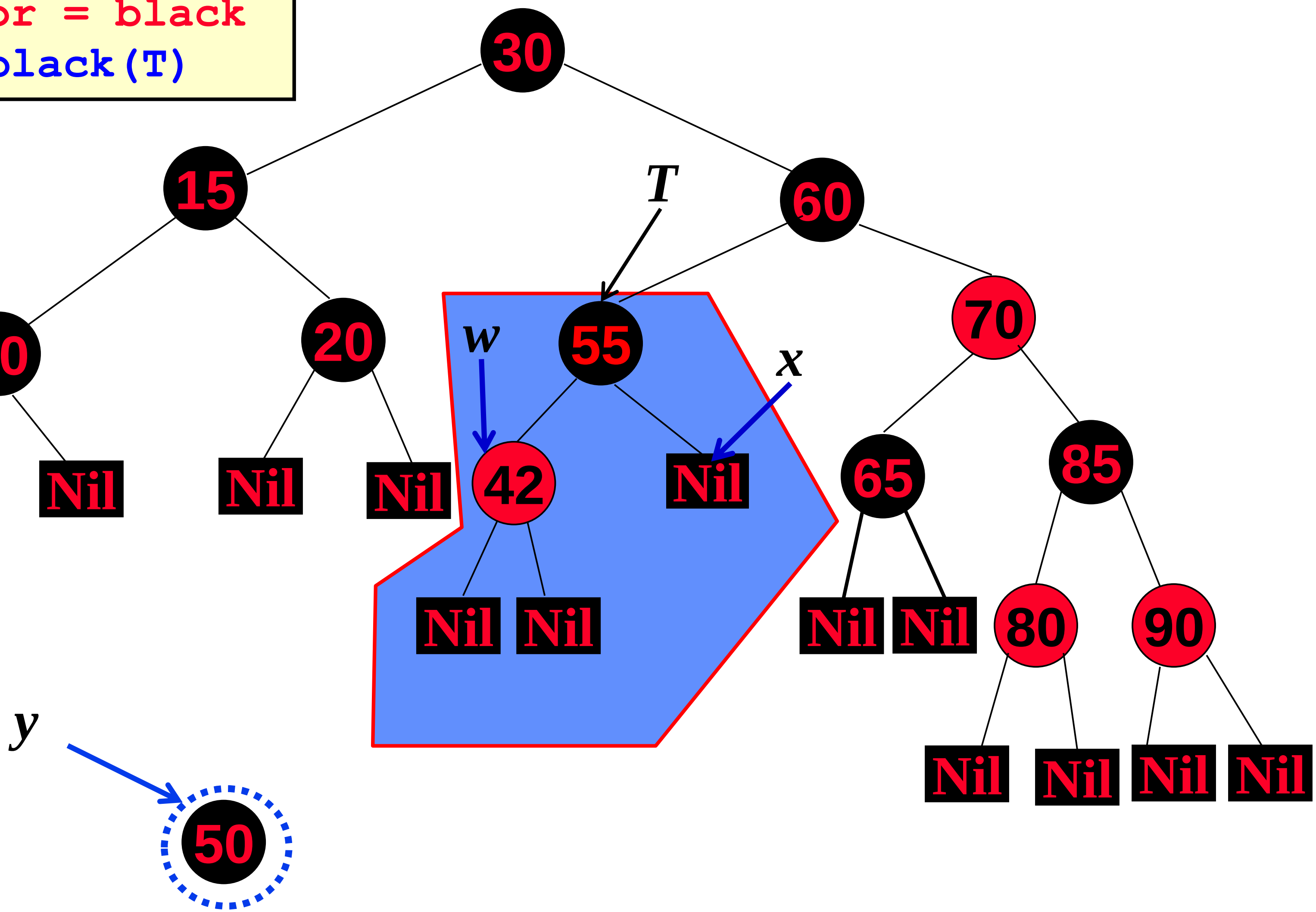
```
T->sx->color = red
T->dx->color = black
propagate-black(T)
```

Caso II simmetrico



Cancellazoine in RB: esempio II

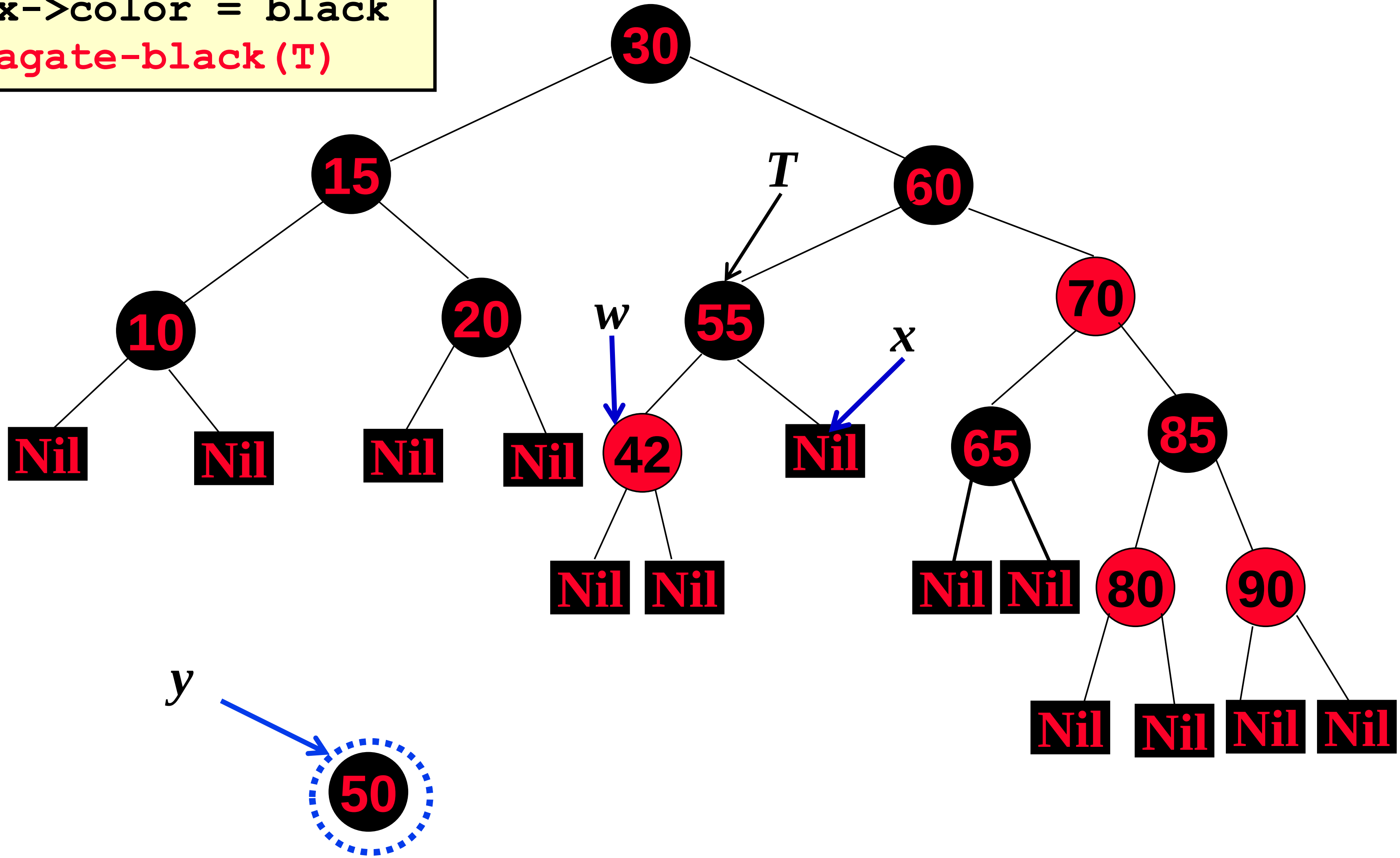
```
T->sx->color= red
T->dx->color = black
propagate-black(T)
```



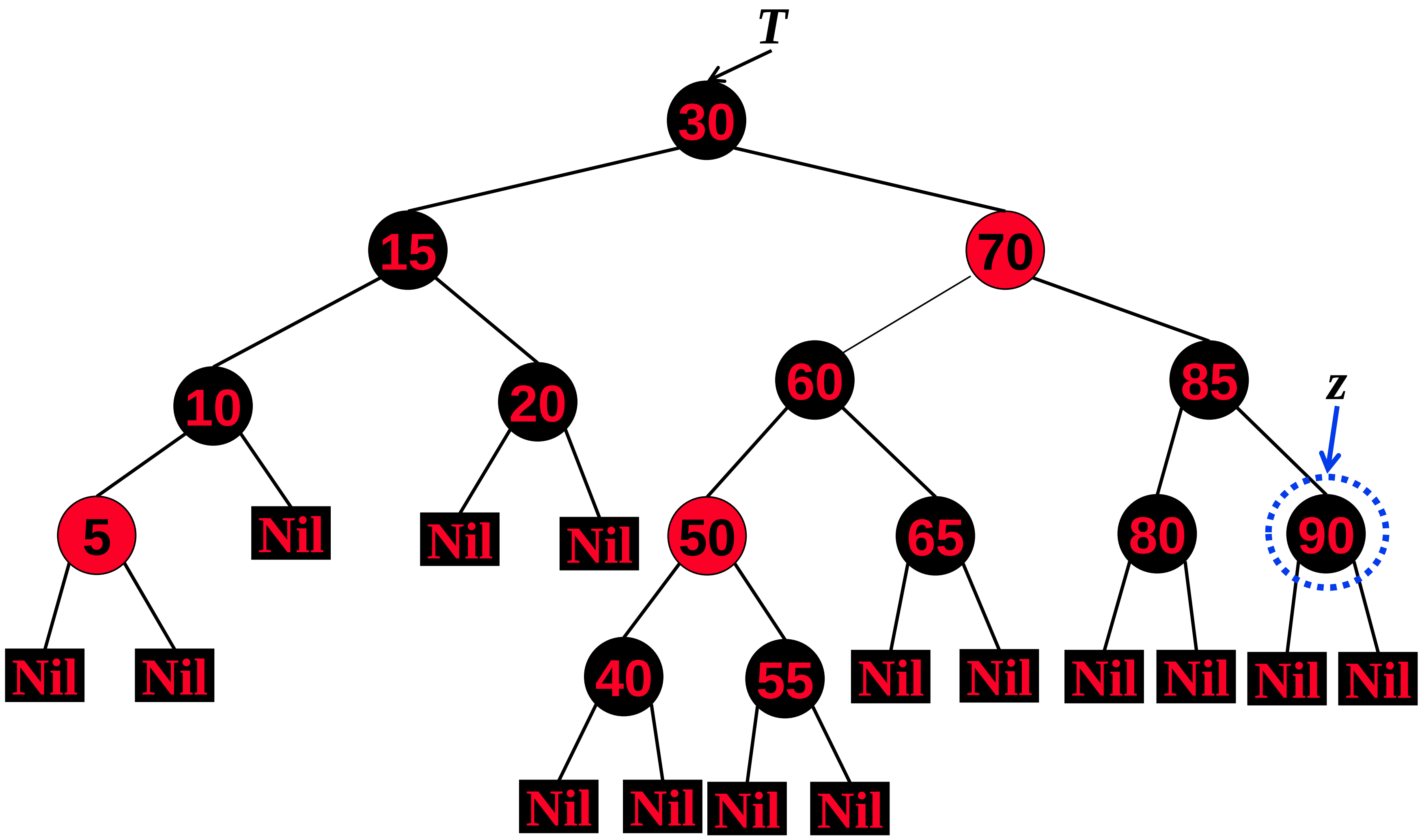
Cancellazoine in RB: esempio II

```
T->sx->color = red
T->dx->color = black
propagate-black(T)
```

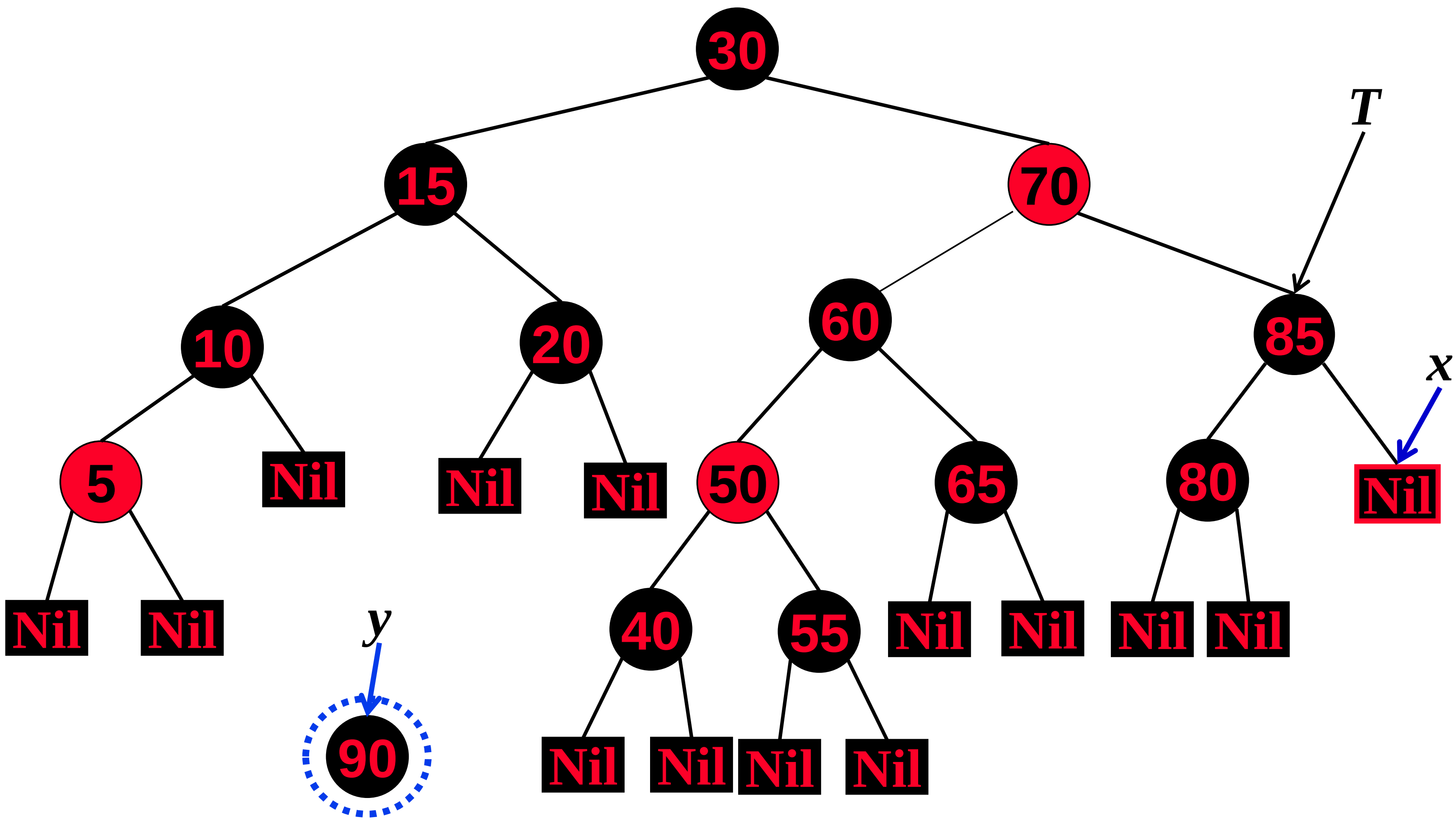
Fatto!



Cancellazoine in RB: esempio III



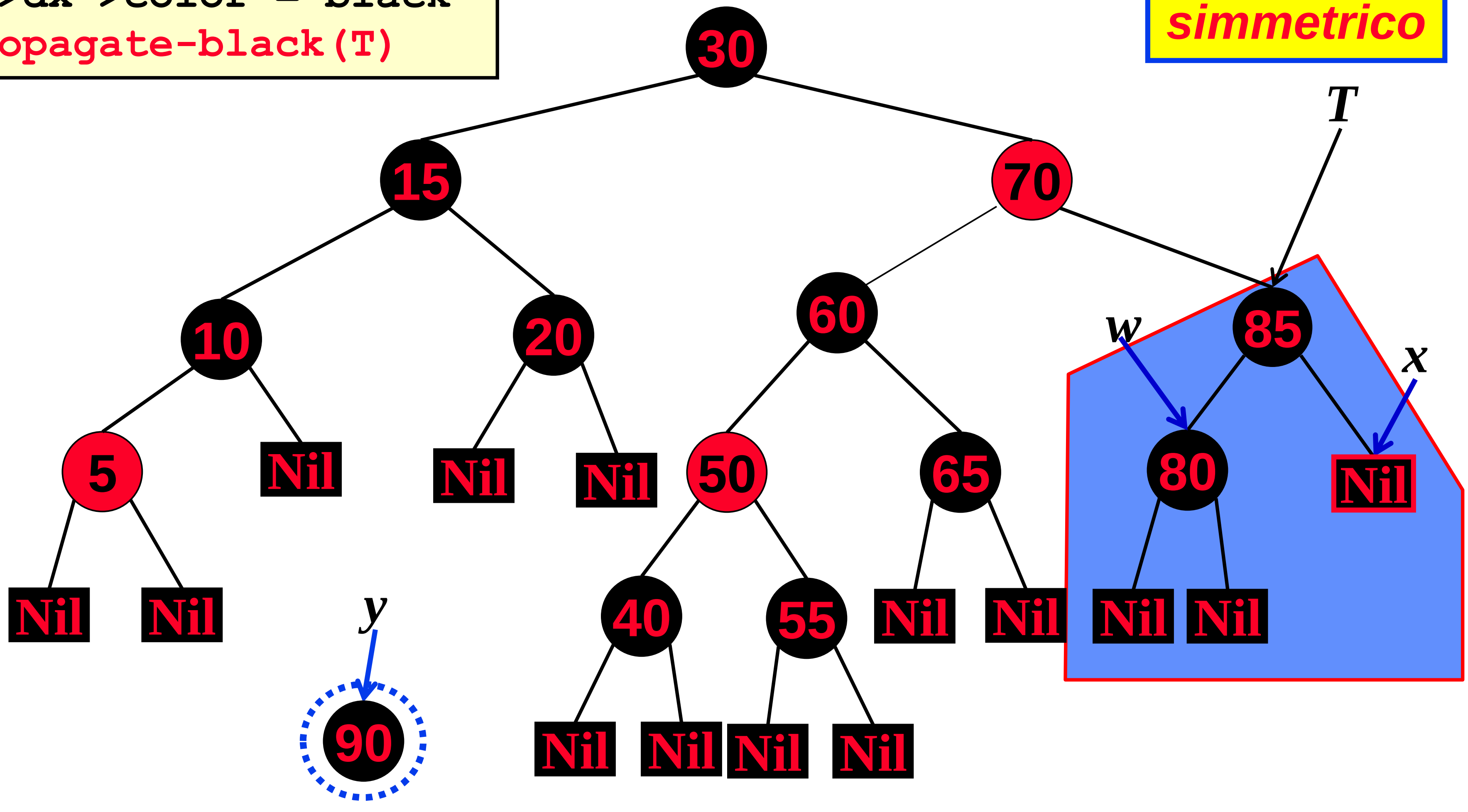
Cancellazoine in RB: esempio III



Cancellazoine in RB: esempio III

```
T->sx->color = red
T->dx->color = black
propagate-black(T)
```

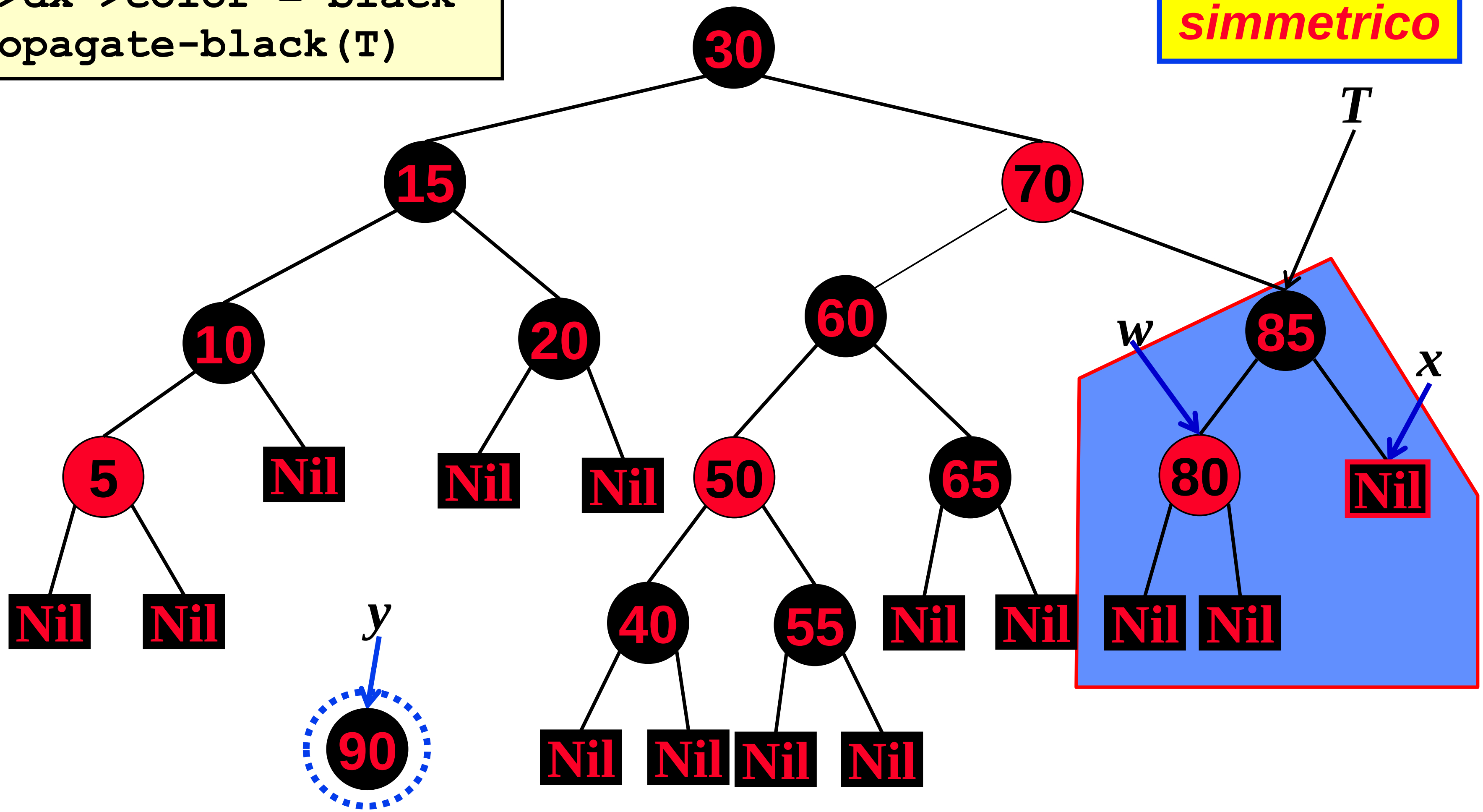
Caso II simmetrico



Cancellazoine in RB: esempio III

```
T->sx->color = red
T->dx->color = black
propagate-black(T)
```

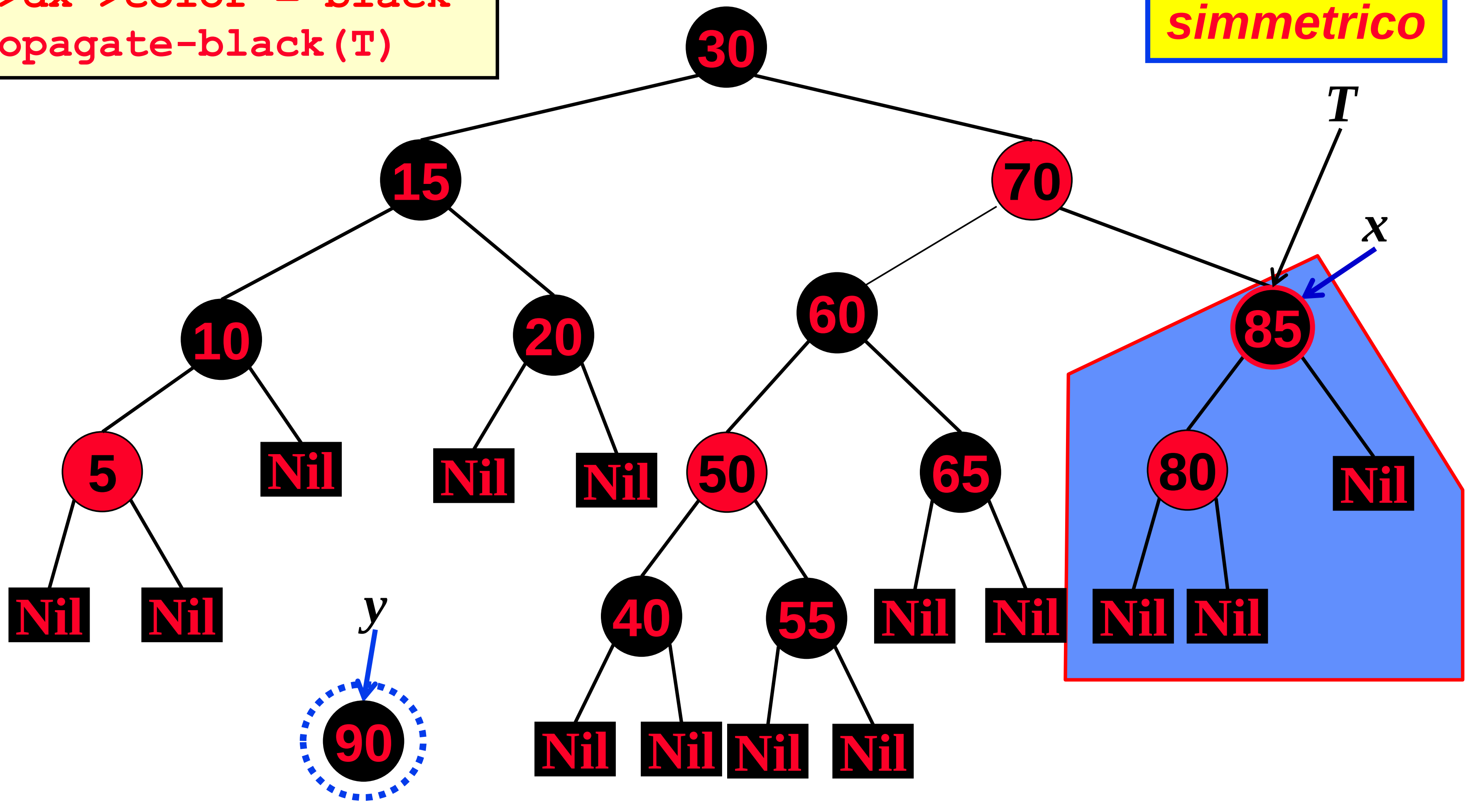
Caso II simmetrico



Cancellazoine in RB: esempio III

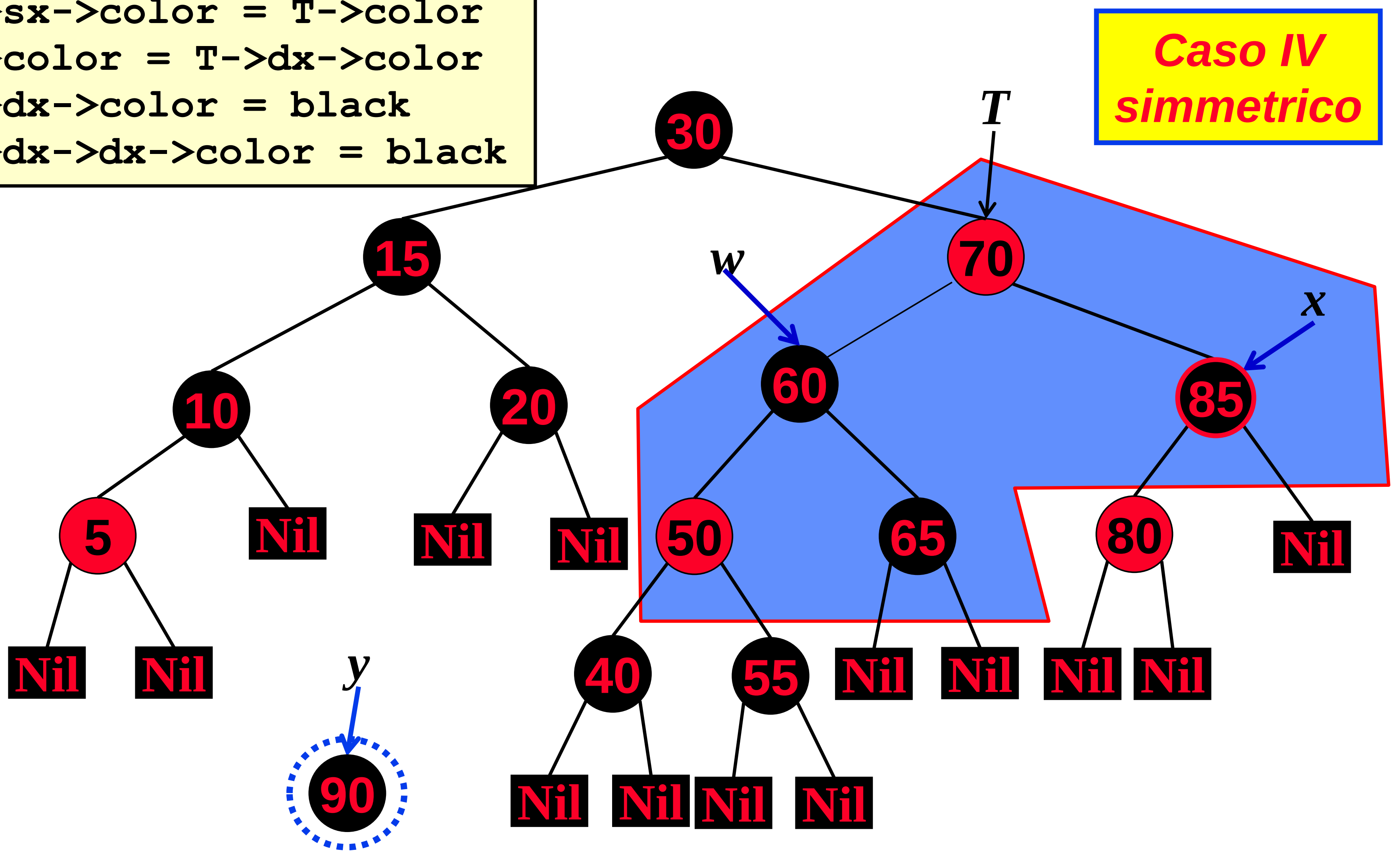
```
T->sx->color = red
T->dx->color = black
propagate-black(T)
```

Caso II simmetrico



Cancellazoine in RB: esempio III

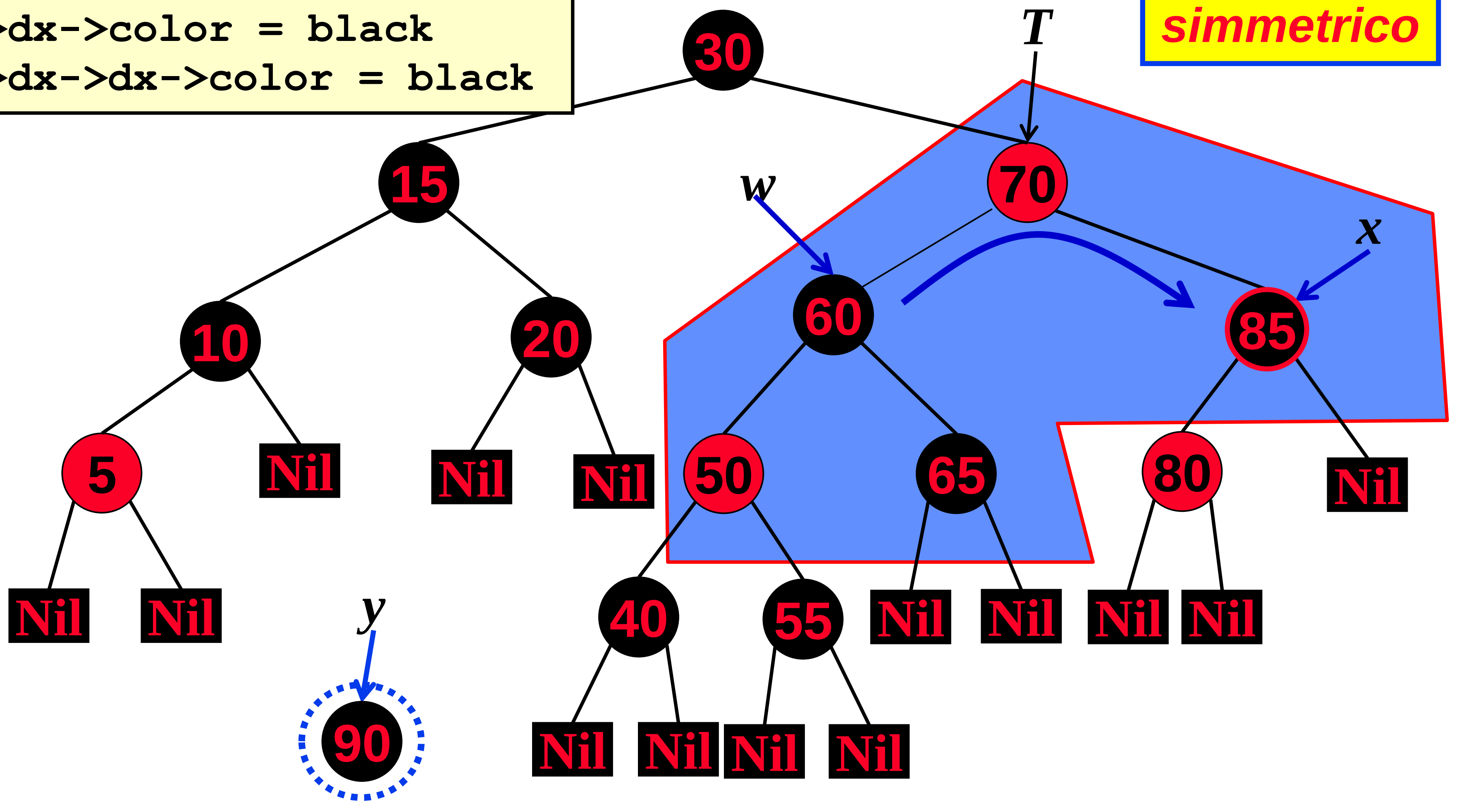
```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```



Cancellazoine in RB: esempio III

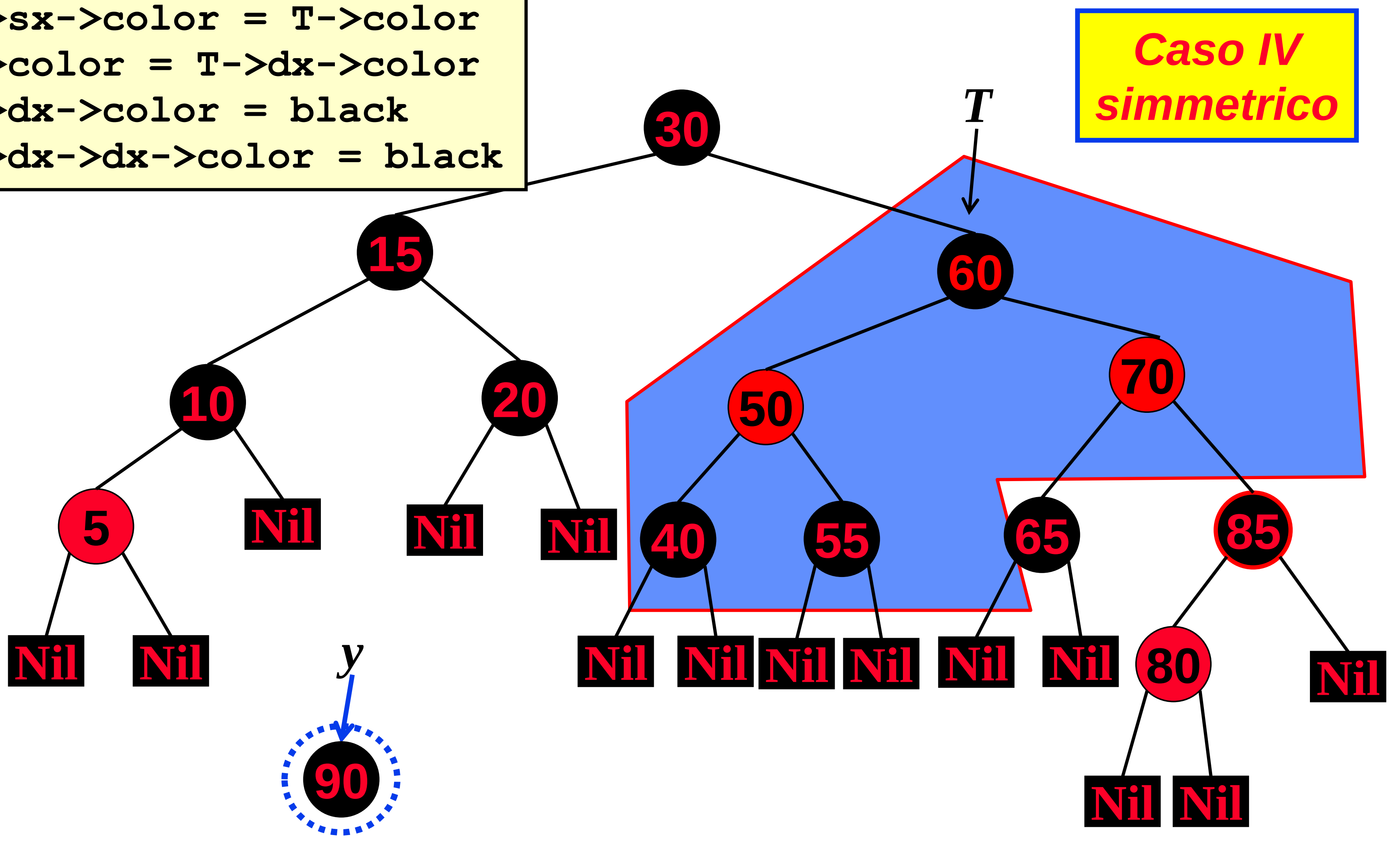
```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```

Caso IV simmetrico



Cancellazoine in RB: esempio III

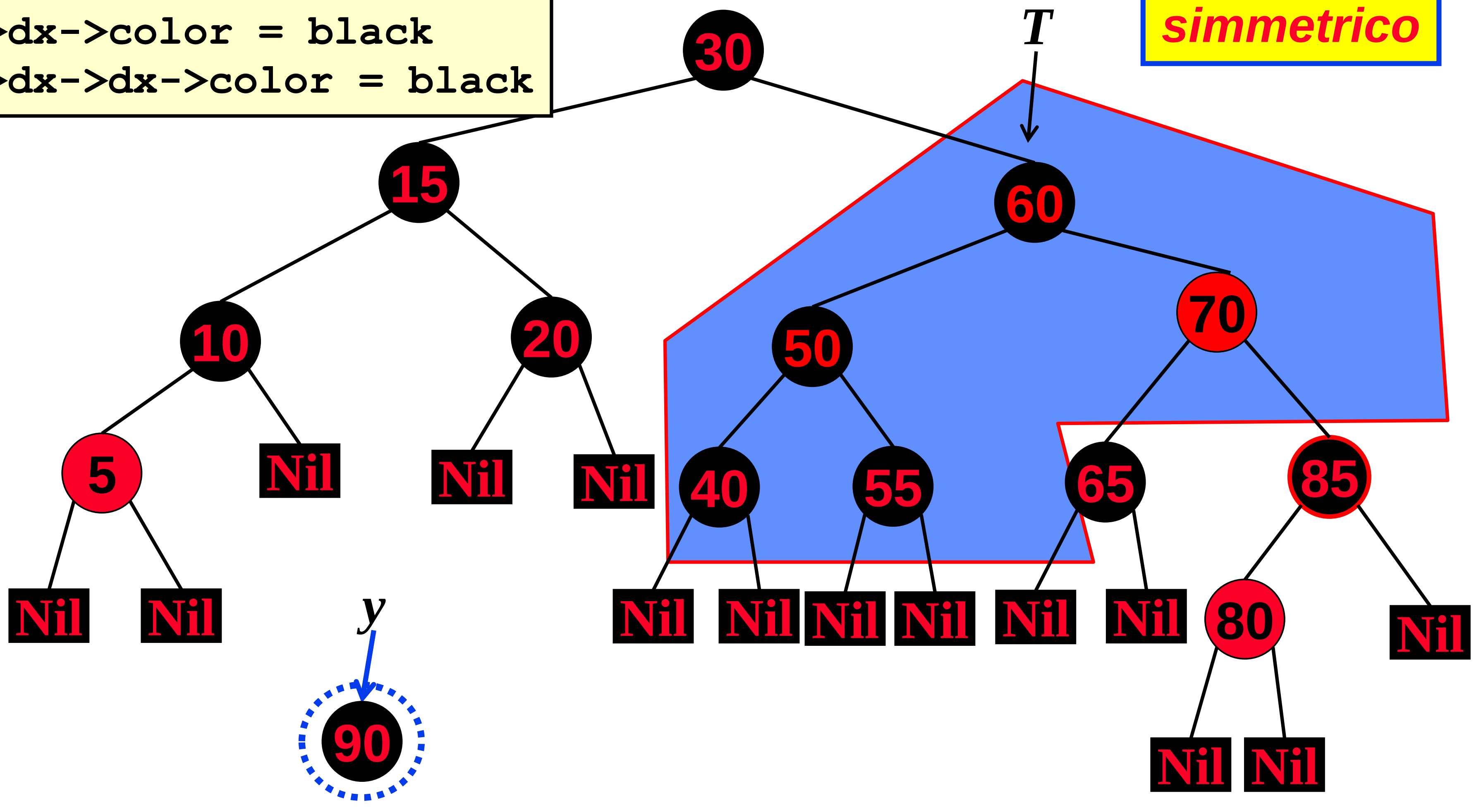
```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```



Cancellazoine in RB: esempio III

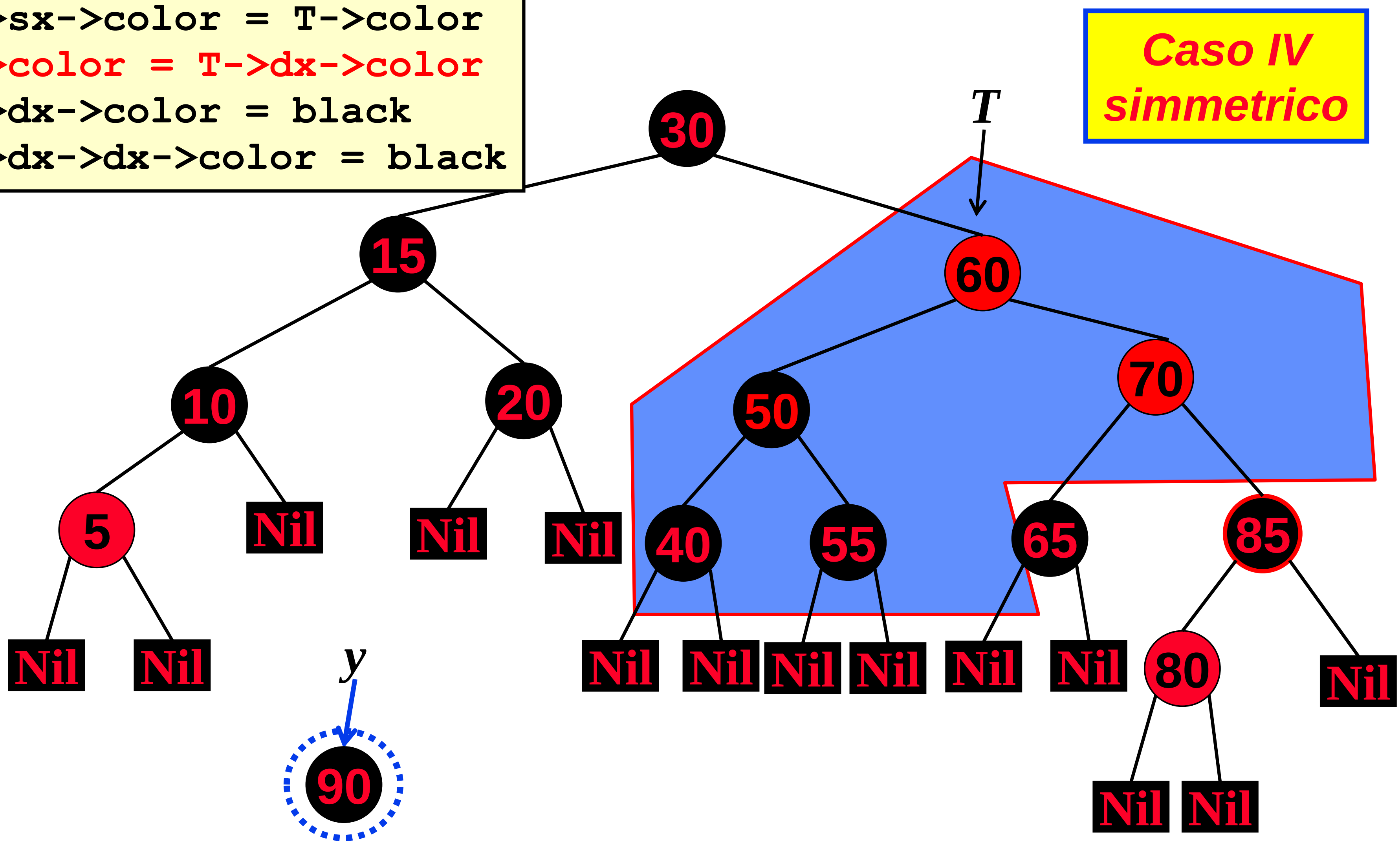
```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```

Caso IV simmetrico



Cancellazoine in RB: esempio III

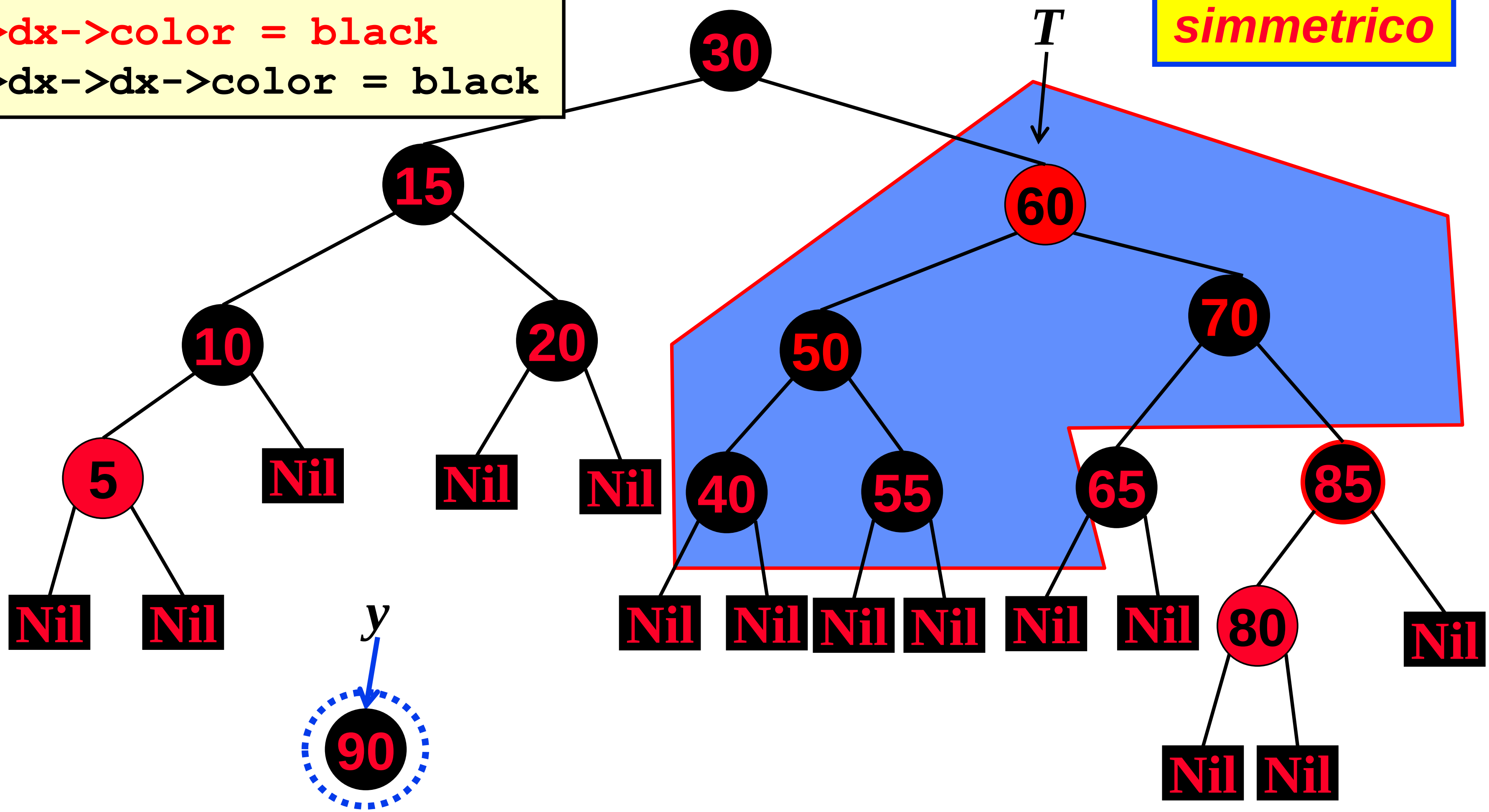
```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```



Cancellazoine in RB: esempio III

```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```

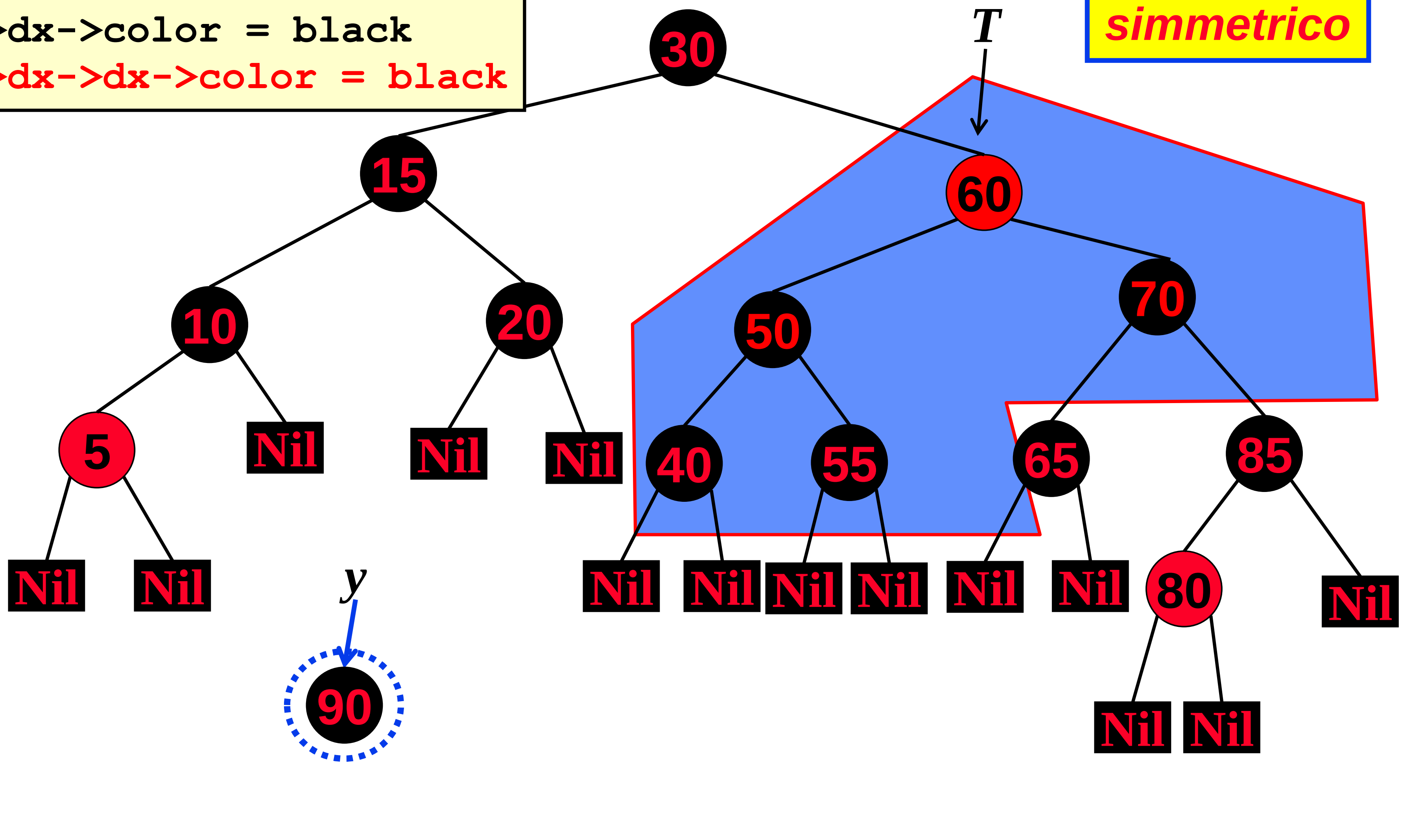
Caso IV simmetrico



Cancellazoine in RB: esempio III

```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```

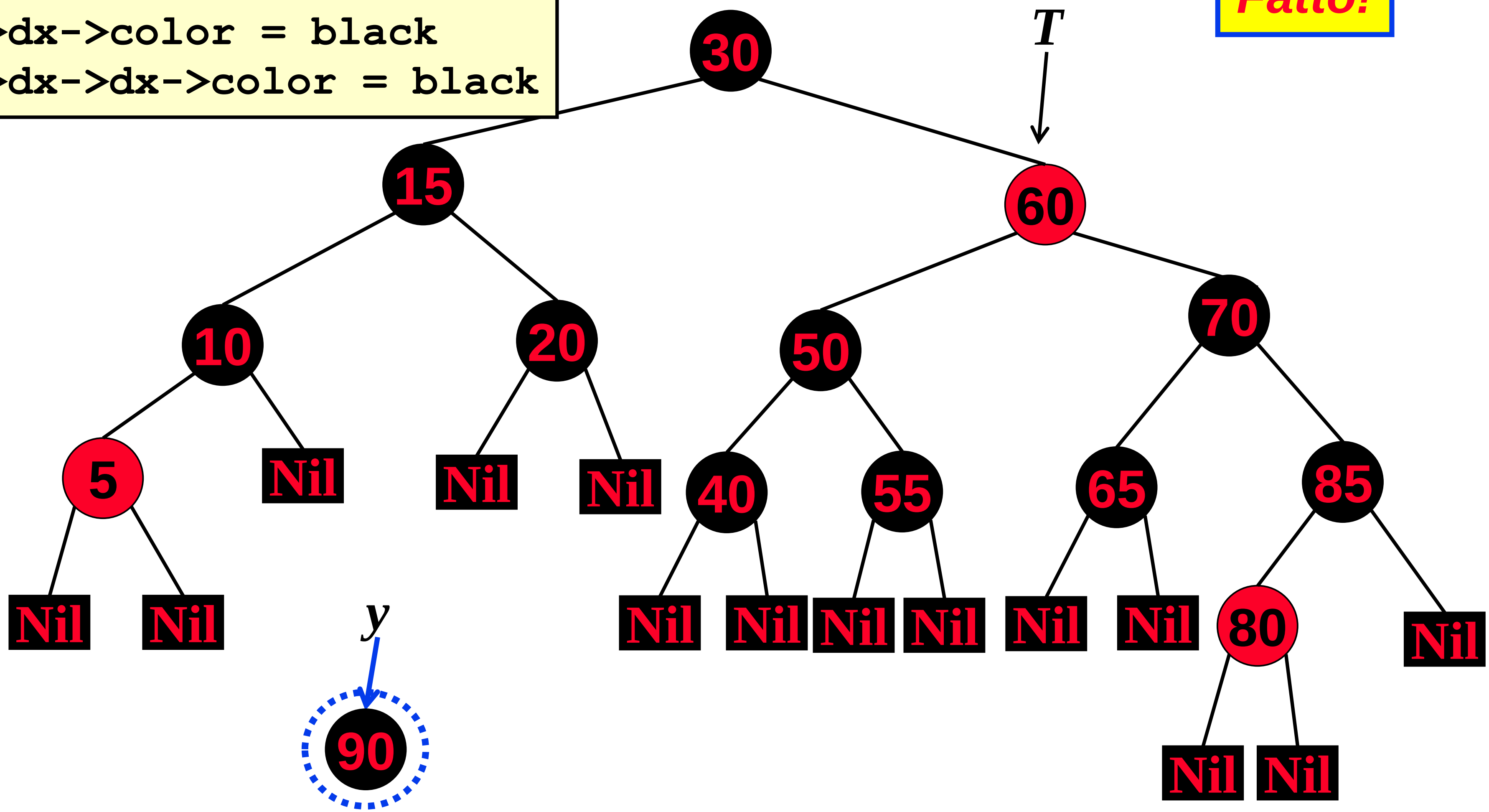
Caso IV simmetrico



Cancellazoine in RB: esempio III

```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```

Fatto!



Cancellazoine in RB

- **L'operazione di cancellazione è concettual-mente complicata!**
- **Ma efficiente:**
  - **il caso 4 è risolutivo e applica 1 sola rotazione**
  - **il caso 3 applica una rotazione e passa nel caso 4**
  - **il caso 2 non fa rotazioni e passa in uno qualsiasi dei casi *ma salendo lungo il percorso di cancellazione***
  - **il caso 1 fa una rotazione e passa in uno degli altri casi (ma se va nel caso 2, il caso 2 termina)**

Quindi

al massimo vengono eseguite 3 rotazioni