- **Durante la modifica di un albero Red-Black:**
  - è possibile che le condizioni di bilanciamento risultino violate.
- **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
    - Rotazione sinistra

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.right$

TREE $p \leftarrow x.parent$

(1) $x.right \leftarrow y.left$ — % Il sottoalbero B diventa figlio destro di x

(1) **if** $y.left \neq$ **nil then** $y.left.parent \leftarrow x$

(2) $y.left \leftarrow x$ — % x diventa figlio sinistro di y

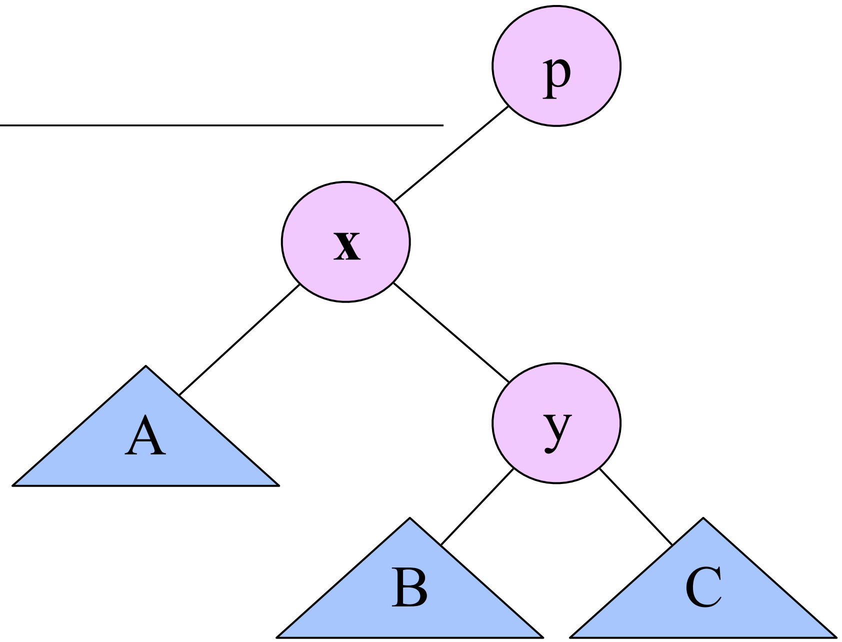
(2) $x.parent \leftarrow y$

(3) $y.parent \leftarrow p$ — % y diventa figlio di p

(3) **if** $p \neq$ **nil then**

  if $p.left = x$ **then** $p.left \leftarrow y$ **else** $p.right \leftarrow y$

return y



✦ Operazioni

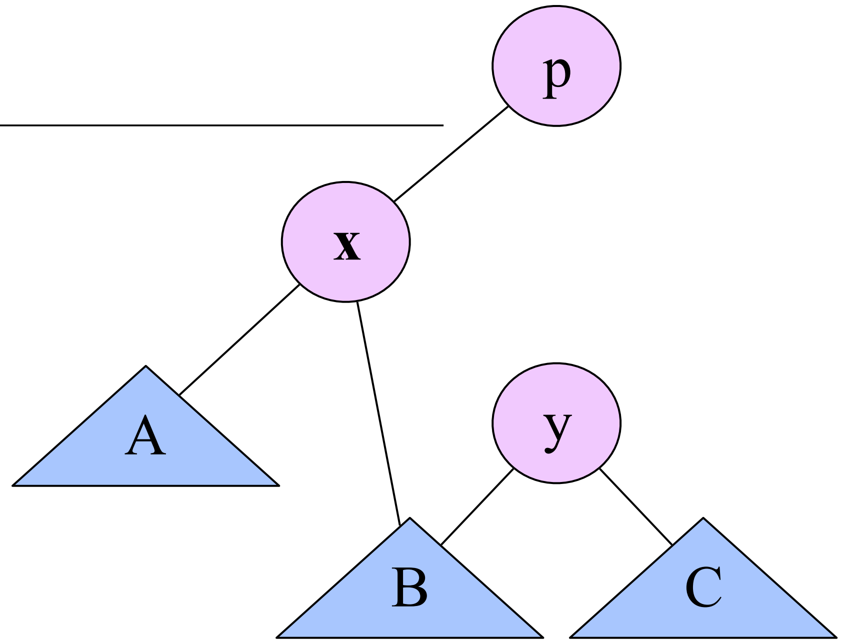
(1) far diventare *B* figlio destro di *x*

(2) far diventare *x* il figlio sinistro di *y*

(3) far diventare *y* figlio di *p*, il vecchio padre di *x*

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.right$
TREE $p \leftarrow x.parent$
(1) $x.right \leftarrow y.left$ % Il sottoalbero B diventa figlio destro di x
(1) **if** $y.left \neq$ **nil then** $y.left.parent \leftarrow x$
(2) $y.left \leftarrow x$ % x diventa figlio sinistro di y
(2) $x.parent \leftarrow y$
(3) $y.parent \leftarrow p$ % y diventa figlio di p
(3) **if** $p \neq$ **nil then**
  if $p.left = x$ **then** $p.left \leftarrow y$ **else** $p.right \leftarrow y$
return y



✦ Operazioni

(1) far diventare B figlio destro di x

(2) far diventare x il figlio sinistro di y

(3) far diventare y figlio di p, il vecchio padre di x

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.right$

TREE $p \leftarrow x.parent$

(1) $x.right \leftarrow y.left$ % Il sottoalbero B diventa figlio destro di x

(1) **if** $y.left \neq$ **nil then** $y.left.parent \leftarrow x$

(2) $y.left \leftarrow x$ % x diventa figlio sinistro di y

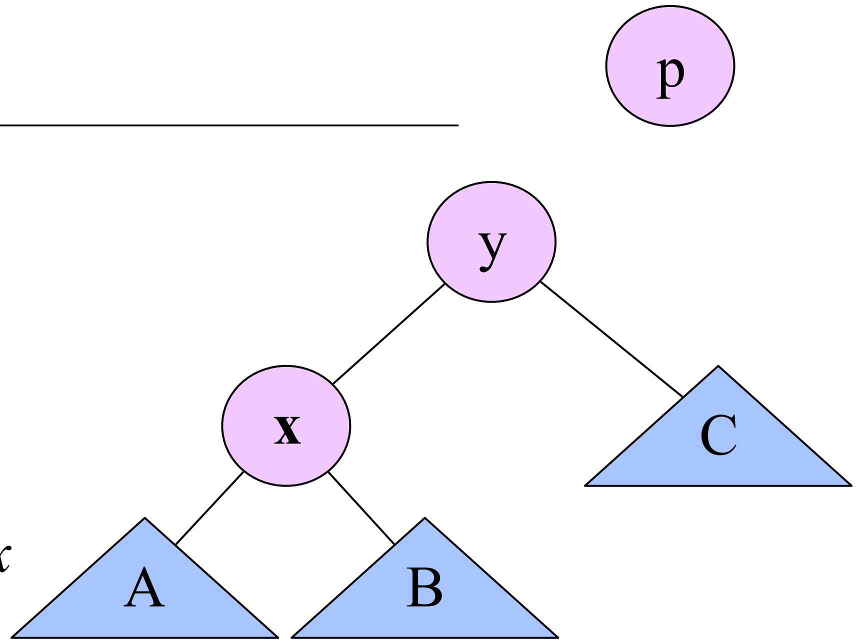
(2) $x.parent \leftarrow y$

(3) $y.parent \leftarrow p$ % y diventa figlio di p

(3) **if** $p \neq$ **nil then**

  if $p.left = x$ **then** $p.left \leftarrow y$ **else** $p.right \leftarrow y$

return y



✦ Operazioni

(1) far diventare B figlio destro di x

(2) far diventare x il figlio sinistro di y

(3) far diventare y figlio di p, il vecchio padre di x

TREE rotateLeft(TREE x)

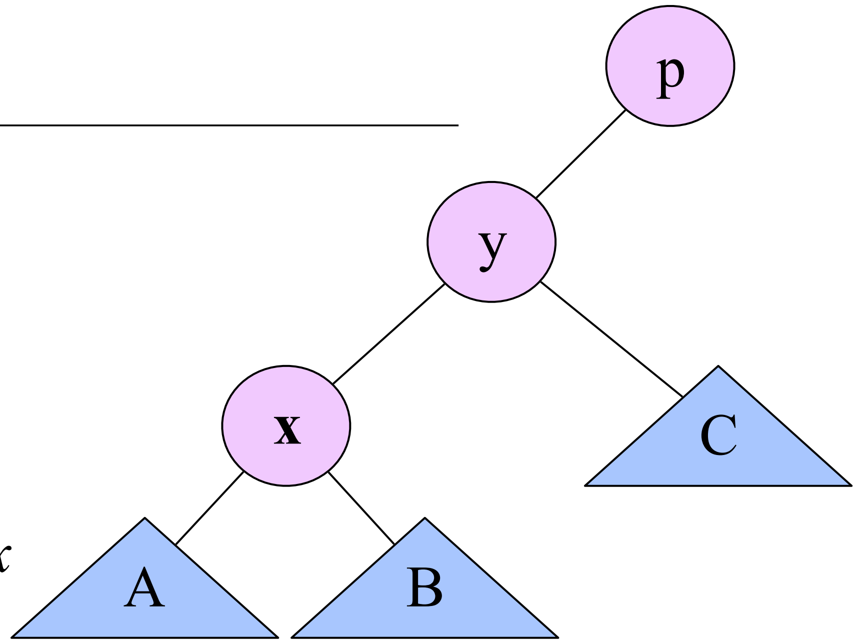
TREE $y \leftarrow x.right$
TREE $p \leftarrow x.parent$
(1) $x.right \leftarrow y.left$ % Il sottoalbero B diventa figlio destro di x
(1) **if** $y.left \neq$ **nil then** $y.left.parent \leftarrow x$
(2) $y.left \leftarrow x$ % x diventa figlio sinistro di y
(2) $x.parent \leftarrow y$
(3) $y.parent \leftarrow p$ % y diventa figlio di p
(3) **if** $p \neq$ **nil then**
  if $p.left = x$ **then** $p.left \leftarrow y$ **else** $p.right \leftarrow y$
return y

Operazioni

(1) far diventare B figlio destro di x

(2) far diventare x il figlio sinistro di y

(3) far diventare y figlio di p, il vecchio padre di x

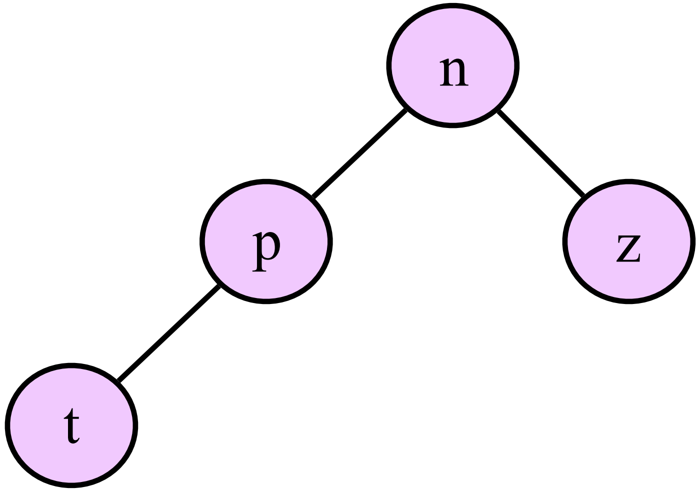


- **Inserimento**
  - Ricerca della posizione usando la stessa procedura usata per gli alberi binari di ricerca
  - Coloriamo il nuovo nodo di rosso
  - Quale delle quattro proprietà può essere violata?
- **Ripasso:**
  - la radice è nera
  - i nodi **Nil** sono neri;
  - se un nodo è rosso, allora entrambi i suoi figli sono neri;
  - ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri

```
TREE insertNode(TREE T, ITEM x, ITEM v)
    TREE p ← nil                                          % Padre
    TREE u ← T
    while u ≠ nil and u.key ≠ x do                        % Cerca posizione inserimento
        p ← u
        u ← iif(x < u.key, u.left, u.right)
    if u ≠ nil and u.key = x then
        u.value ← v                                       % Chiave già presente
    else
        TREE n ← Tree(x, v)
        link(p, n, x)
                       balanceInsert(n)
        if p = nil then return n                          % Primo nodo ad essere inserito
    return T                                              % Ritorna albero non modificato
```

- **Come sistemare:**
  - Ci spostiamo verso l’alto lungo il percorso di inserimento
    - per ripristinare la proprietà 3 (red-black)
    - spostando le violazioni verso l’alto rispettando il vincolo 4 (mantenendo l’altezza nera dell’albero)
  - Al termine, coloriamo la radice di nero
- **Nota**
  - Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi!

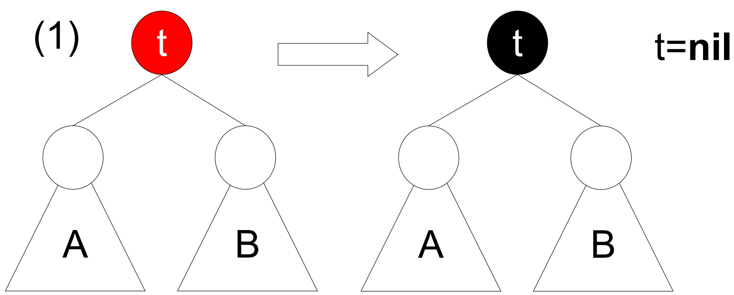
- **Nodi coinvolti**
  - Il nodo inserito t
  - Suo padre p
  - Suo nonno n
  - Suo zio z



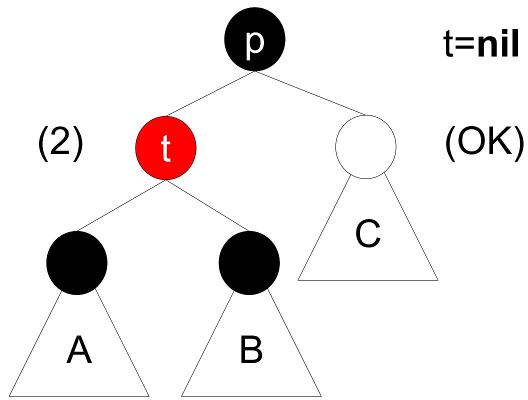
balanceInsert(TREE t)

$t.color \leftarrow$ RED
while $t \neq$ **nil do**
  TREE $p \leftarrow t.parent$ % Padre
  TREE $n \leftarrow$ iif($p \neq$ **nil**, $p.parent$, **nil**) % Nonno
  TREE $z \leftarrow$ iif($n =$ **nil**, **nil**, iif($n.left = p$, $n.right$, $n.left$)) % Zio

- **Caso 1:**
  - Nuovo nodo *t* non ha padre
  - Primo nodo ad essere inserito o siamo risaliti fino alla radice
  - Si colora *t* di nero



- **Caso 2**
  - Padre *p* di *t* è nero
  - Nessun vincolo violato



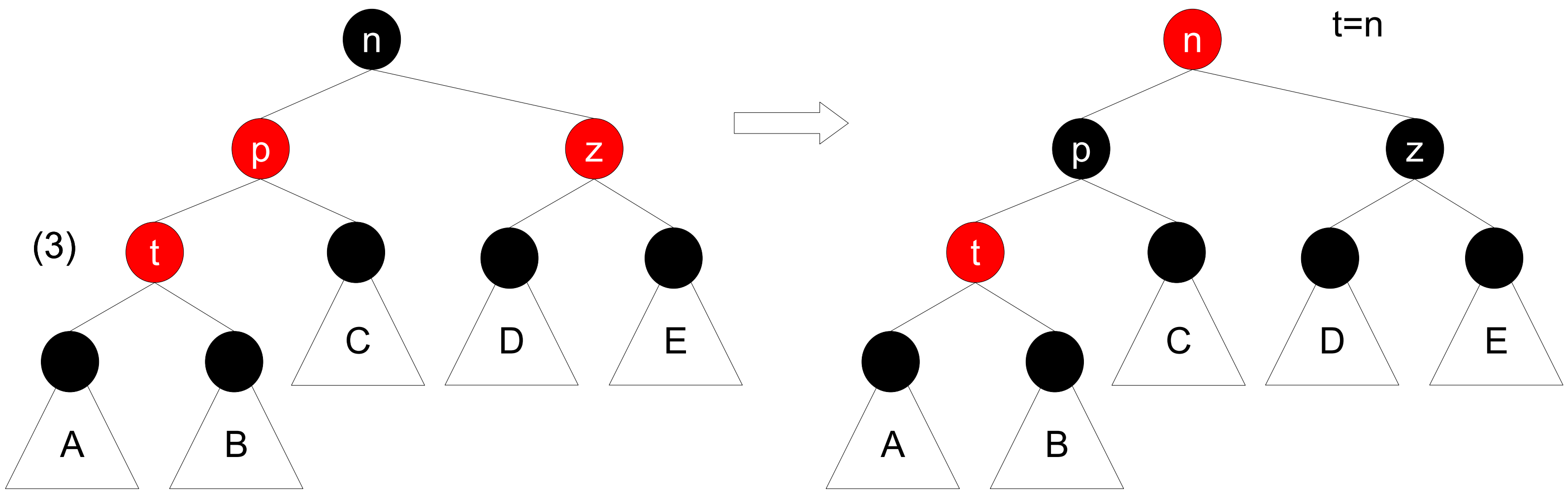
- **Caso 1:**
  - Nuovo nodo *t* non ha padre
  - Primo nodo ad essere inserito o siamo risaliti fino alla radice
  - Si colora *t* di nero

- **Caso 2**
  - Padre *p* di *t* è nero
  - Nessun vincolo violato

```
while t ≠ nil do
    TREE p ← t.parent                                          % Padre
    TREE n ← iif(p.parent ≠ nil, p.parent, nil)                % Nonno
    TREE z ← iif(n = nil, nil, iif(n.left = p, right, left))   % Zio
    if p = nil then                                            % Caso (1)
        t.color ← BLACK
        t ← nil
    else if p.color = BLACK then                               % Caso (2)
        t ← nil
```

- **Caso 3**
  - *t* rosso
  - *p* rosso
  - *z* rosso

- Se *z* è rosso, è possibile colorare di nero *p*, *z*, e di rosso *n*.
- Poiché tutti i cammini che passano per *z* e *p* passano per *n*, la lunghezza dei cammini neri non è cambiata.
- Il problema può essere ora sul nonno:
  - violato vincolo (1), ovvero *n* può essere una radice rossa
  - violato vincolo (3), ovvero *n* rosso può avere un padre rosso.
- Poniamo *t* = *n*, e il ciclo continua.



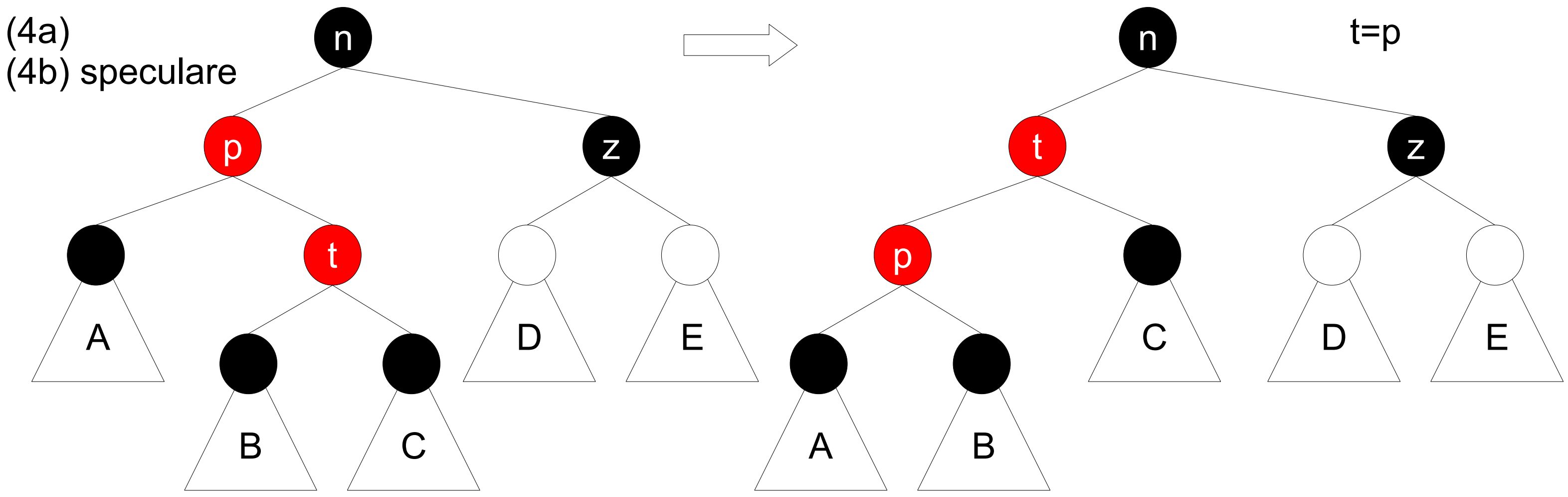
- **Caso 3**
  - t rosso
  - p rosso
  - z rosso

- Se z è rosso, è possibile colorare di nero p, z, e di rosso n.
- Poiché tutti i cammini che passano per z e p passano per n, la lunghezza dei cammini neri non è cambiata.
- Il problema può essere ora sul nonno:
  - violato vincolo (1), ovvero n può essere una radice rossa
  - violato vincolo (3), ovvero n rosso può avere un padre rosso.
- Poniamo $t = n$, e il ciclo continua.

else if $z.color = \text{RED}$ **then** % Caso (3)
  $p.color \leftarrow z.color \leftarrow \text{BLACK}$
  $n.color \leftarrow \text{RED}$
  $t \leftarrow n$

- **Caso 4a,4b**
  - *t* rosso
  - *p* rosso
  - *z* nero

- Si assuma che *t* sia figlio <u>destro</u> di *p* e che *p* sia figlio <u>sinistro</u> di *n*
- Una rotazione a <u>sinistra</u> a partire dal nodo *p* scambia i ruoli di *t* e *p* ottenendo il caso (5a), dove i nodi rossi in conflitto sul vincolo (3) sono entrambi figli <u>sinistri</u> dei loro padri
- I nodi coinvolti nel cambiamento sono *p* e *t*, entrambi rossi, quindi la lunghezza dei cammini neri non cambia



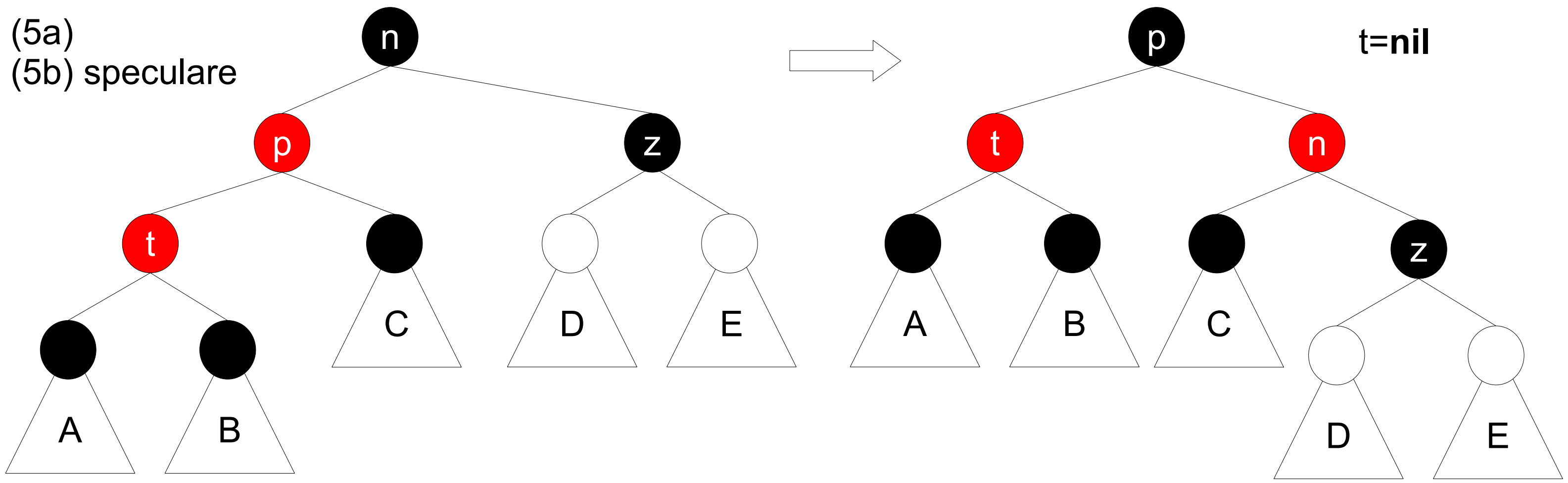
- **Caso 4a,4b**
  - t rosso
  - p rosso
  - z nero

- Si assuma che t sia figlio <u>destro</u> di p e che p sia figlio <u>sinistro</u> di n
- Una rotazione a <u>sinistra</u> a partire dal nodo p scambia i ruoli di t e p ottenendo il caso (5a), dove i nodi rossi in conflitto sul vincolo (3) sono entrambi figli <u>sinistri</u> dei loro padri
- I nodi coinvolti nel cambiamento sono p e t, entrambi rossi, quindi la lunghezza dei cammini neri non cambia

```
else
    if (t = p.right) and (p = n.left) then                % Caso (4.a)
        n.left ← rotateLeft(p)
        t ← p
    else if (t = p.left) and (p = n.right) then           % Caso (4.b)
        n.right ← rotateRight(p)
        t ← p
```

- **Caso 5a,5b**
  - *t* rosso
  - *p* rosso
  - *z* nero

- Si assuma che *t* sia figlio <u>sinistro</u> di *p* e *p* sia figlio <u>sinistro</u> di *n*
- Una rotazione a <u>destra</u> a partire da *n* ci porta ad una situazione in cui *t* e *n* sono figli di p
- Colorando *n* di rosso e *p* di nero ci troviamo in una situazione in cui tutti i vincoli Red-Black sono rispettati
- in particolare, la lunghezza dei cammini neri che passano per la radice è uguale alla situazione iniziale

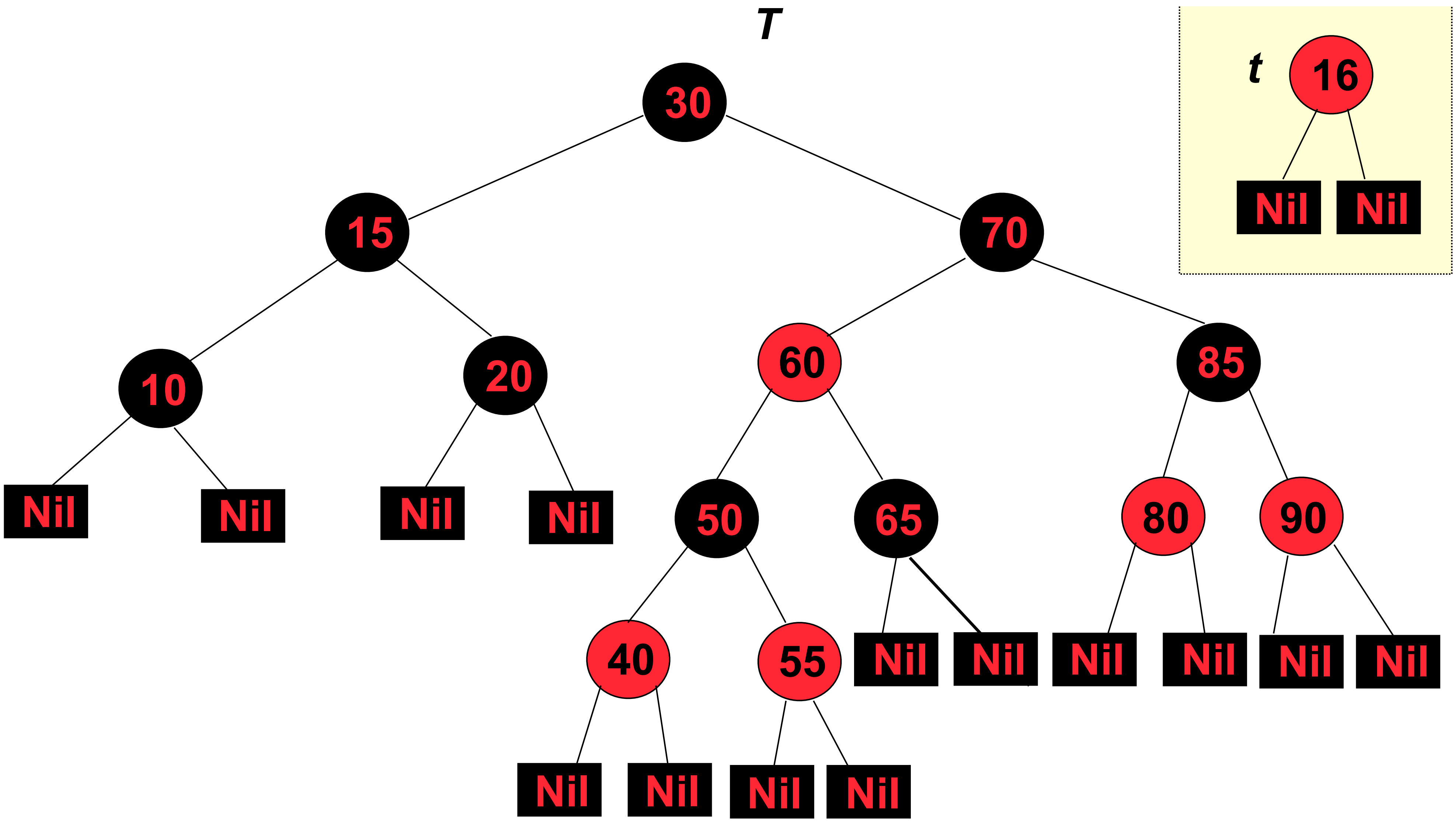


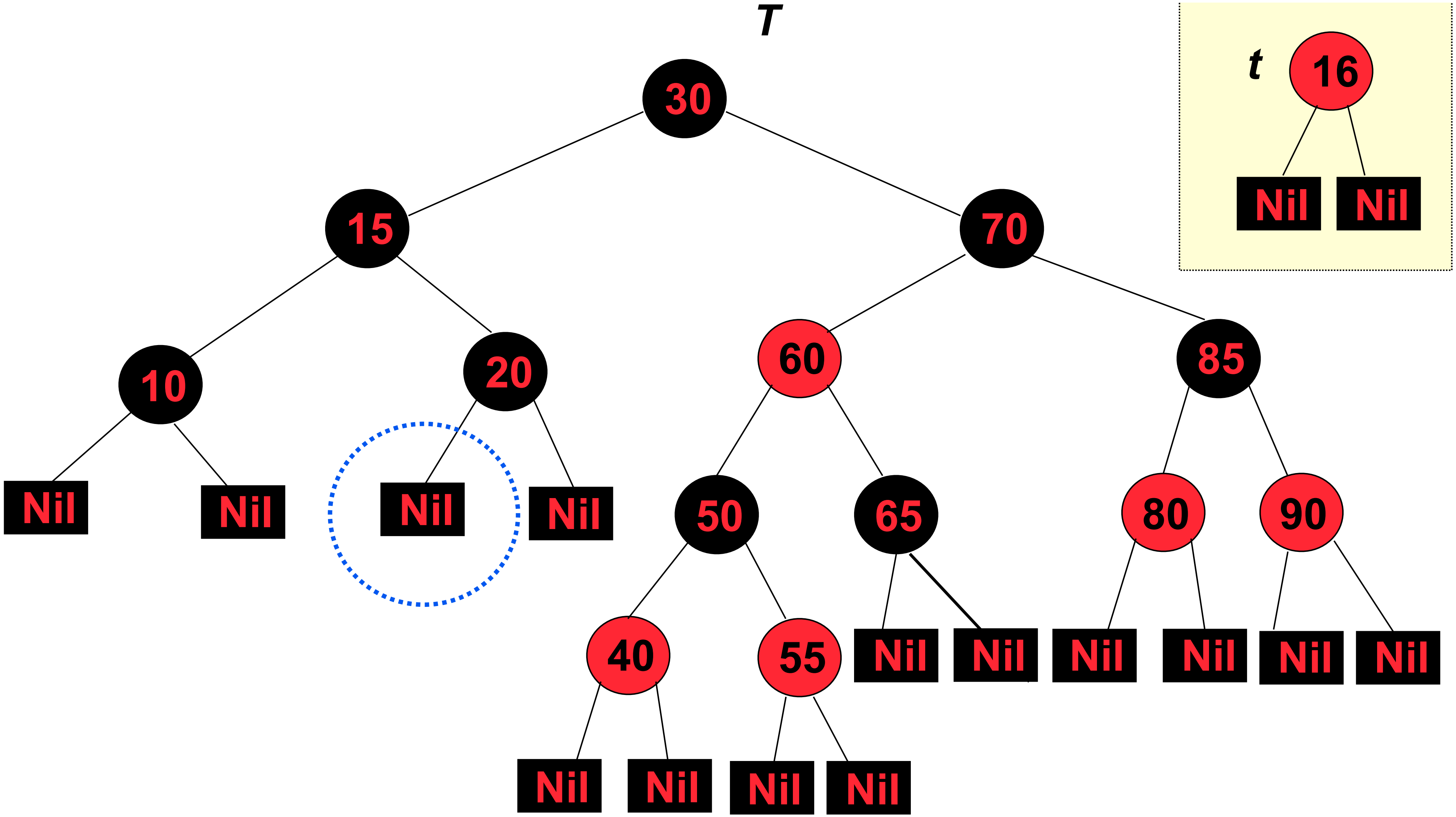
- **Caso 5a,5b**
  - t rosso
  - p rosso
  - z nero

- Si assuma che t sia figlio sinistro di p e p sia figlio sinistro di n
- Una rotazione a destra a partire da n ci porta ad una situazione in cui t e n sono figli di p
- Colorando n di rosso e p di nero ci troviamo in una situazione in cui tutti i vincoli Red-Black sono rispettati
- in particolare, la lunghezza dei cammini neri che passano per la radice è uguale alla situazione iniziale

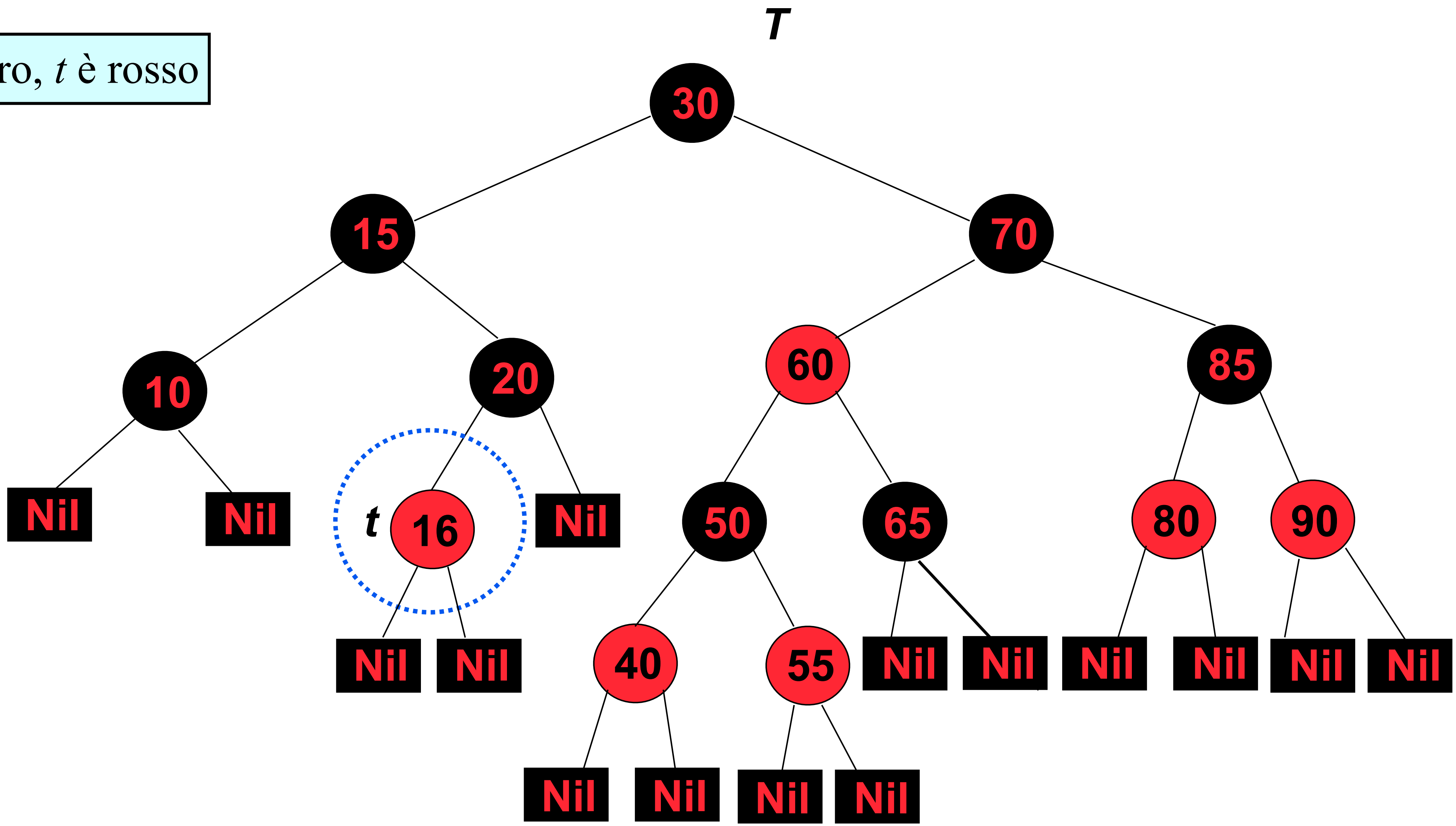
```
else
    if (t = p.left) and (p = n.left) then                 % Caso (5.a)
        n.left ← rotateRight(n)
    else if (t = p.right) and (p = n.right) then          % Caso (5.b)
        n.right ← rotateLeft(n)
    p.color ← BLACK
    n.color ← RED
    t ← nil
```

```
balanceInsert(TREE t)
t.color ← RED
while t ≠ nil do
    TREE p ← t.parent                                              % Padre
    TREE n ← iif(p ≠ nil, p.parent, nil)                           % Nonno
    TREE z ← iif(n = nil, nil, iif(n.left = p, n.right, n.left))   % Zio
    if p = nil then                                                % Caso (1)
        t.color ← BLACK
        t ← nil
    else if p.color = BLACK then                                   % Caso (2)
        t ← nil
    else if z.color = RED then                                     % Caso (3)
        p.color ← z.color ← BLACK
        n.color ← RED
        t ← n
    else
        if (t = p.right) and (p = n.left) then                     % Caso (4.a)
            rotateLeft(p)
            t ← p
        else if (t = p.left) and (p = n.right) then                % Caso (4.b)
            rotateRight(p)
            t ← p
        else
            if (t = p.left) and (p = n.left) then                  % Caso (5.a)
                rotateRight(n)
            else if (t = p.right) and (p = n.right) then           % Caso (5.b)
                rotateLeft(n)
            p.color ← BLACK
            n.color ← RED
            t ← nil
```

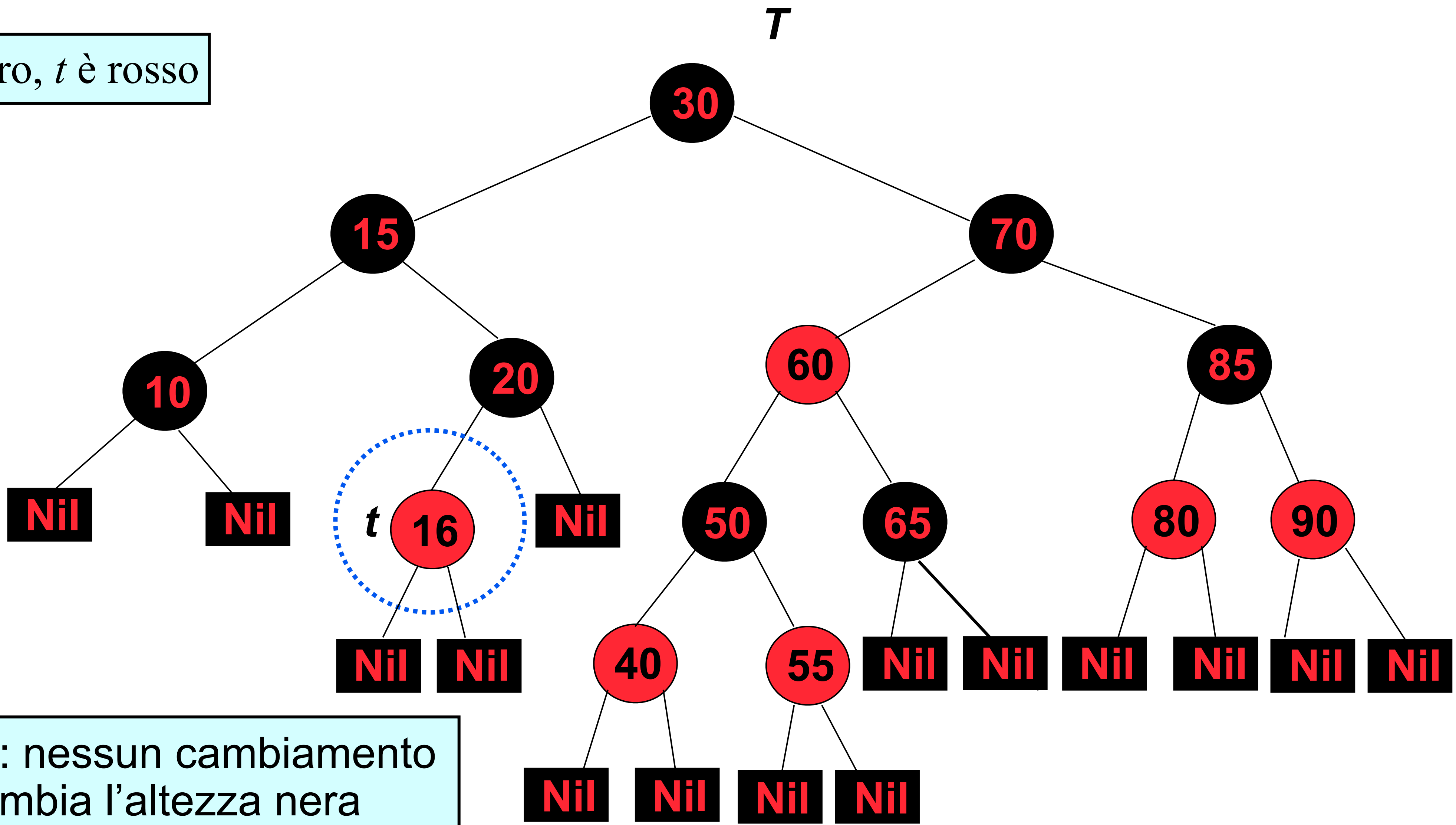

T
30
15
70
10
20
60
85
Nil
Nil
Nil
Nil
50
65
80
90
40
55
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
t
16
Nil
Nil


T
30
15
70
10
20
60
85
50
65
80
90
40
55
Nil
t
16
Nil
Nil

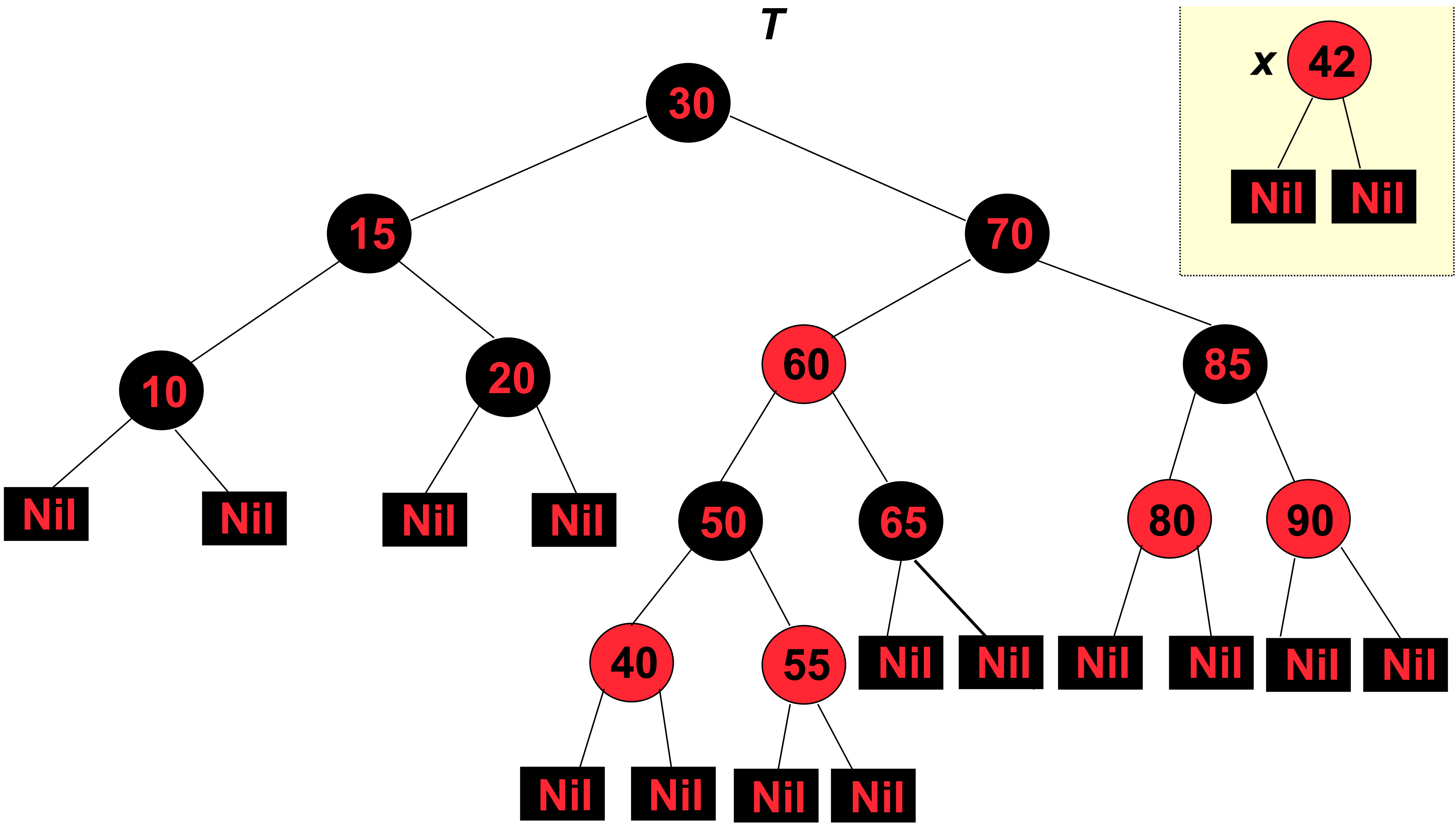
p è nero, t è rosso

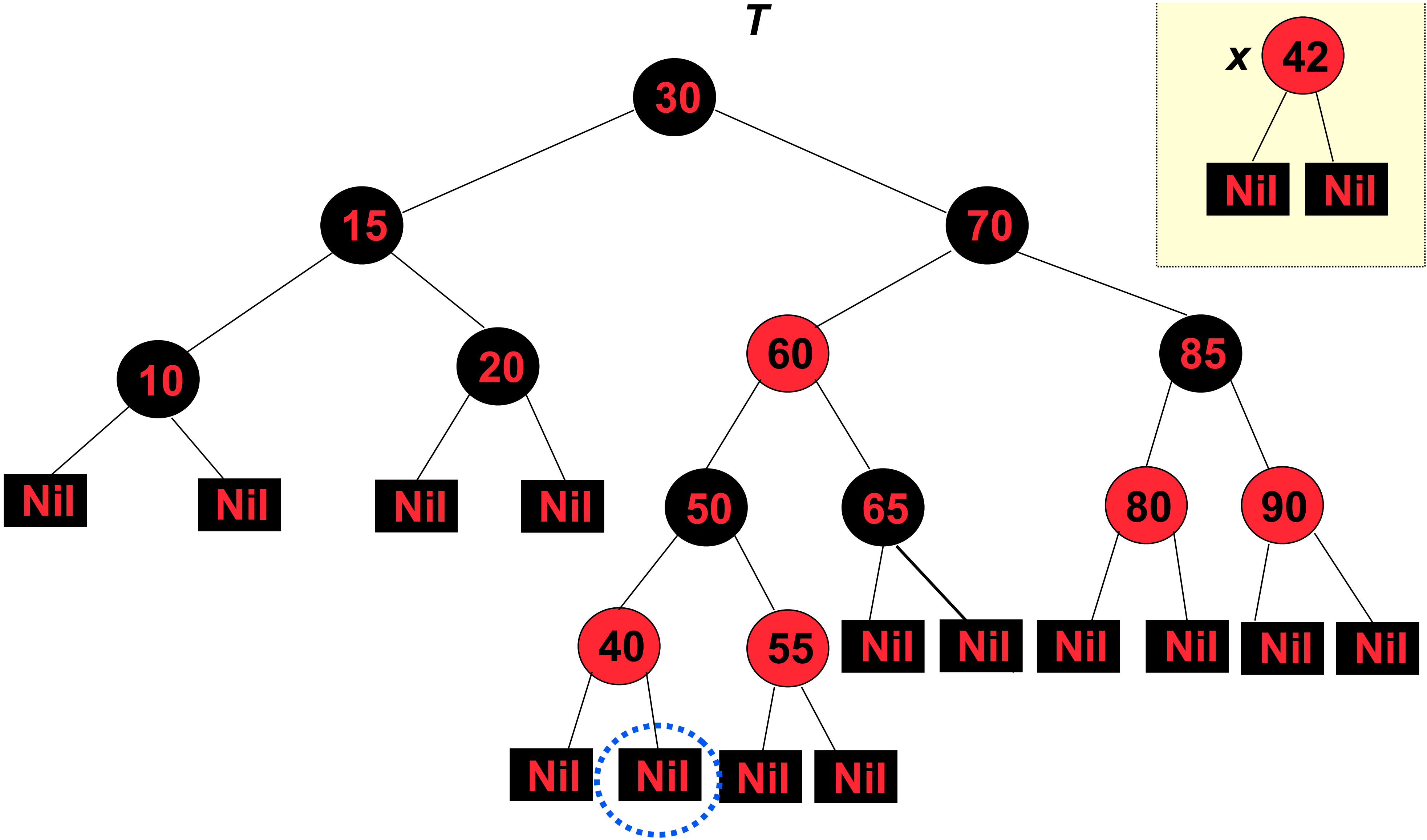


p è nero, t è rosso

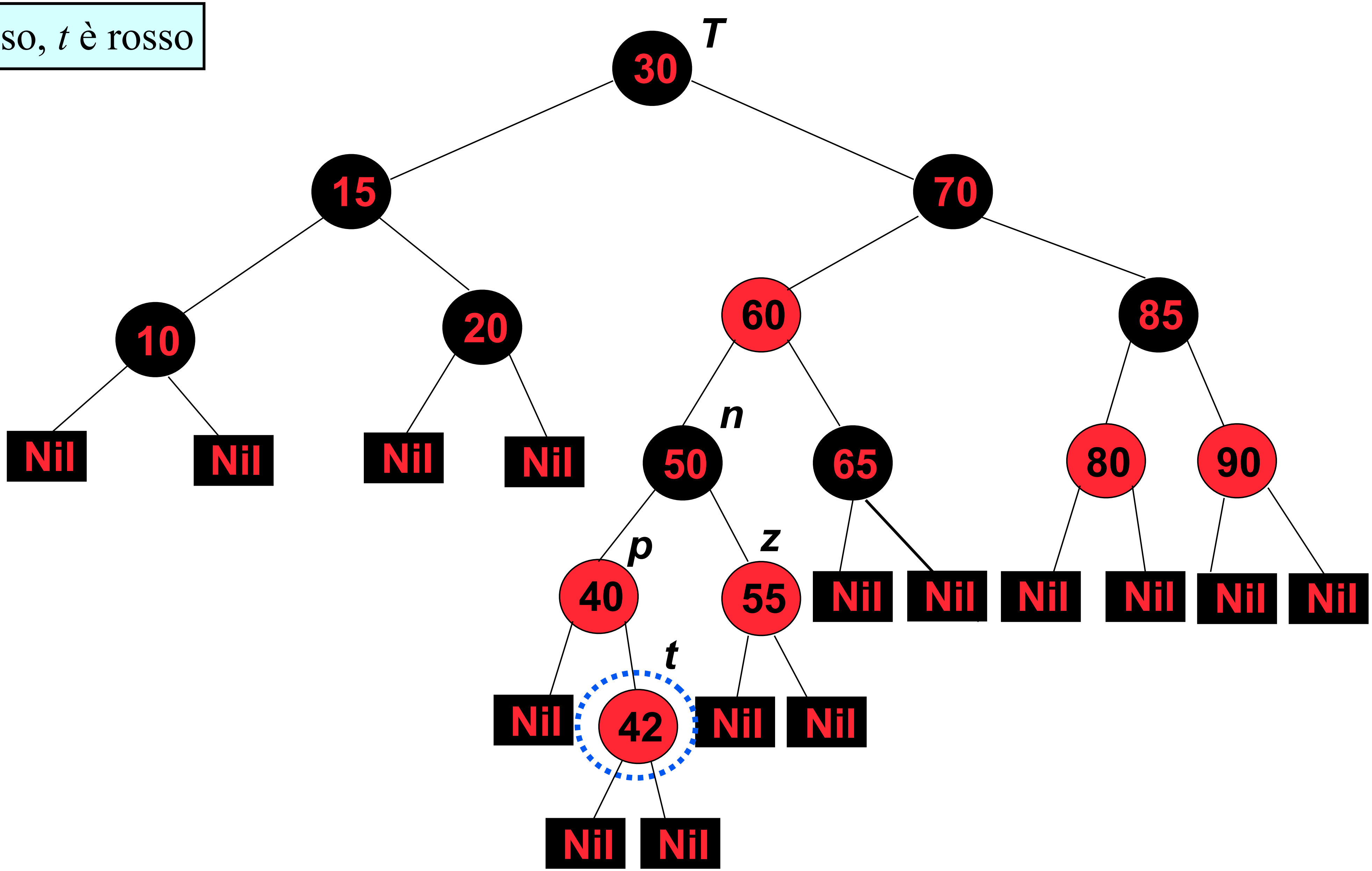


Caso 2: nessun cambiamento
Non cambia l'altezza nera
di nessun nodo!

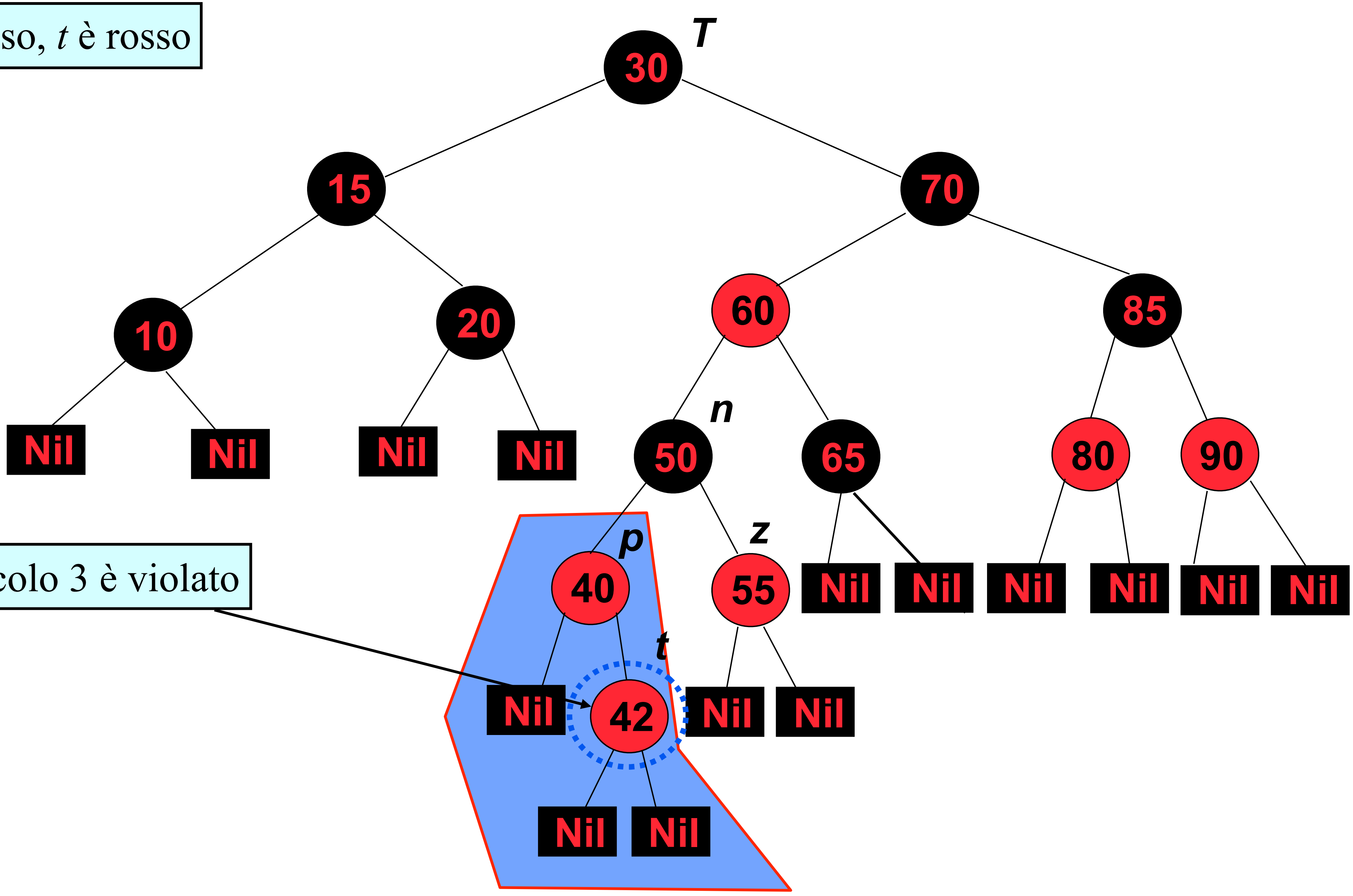

T
30
15
70
10
20
60
85
50
65
80
90
40
55
x
42
Nil


T
30
15
70
10
20
60
85
50
65
80
90
40
55
Nil
x
42

p è rosso, t è rosso

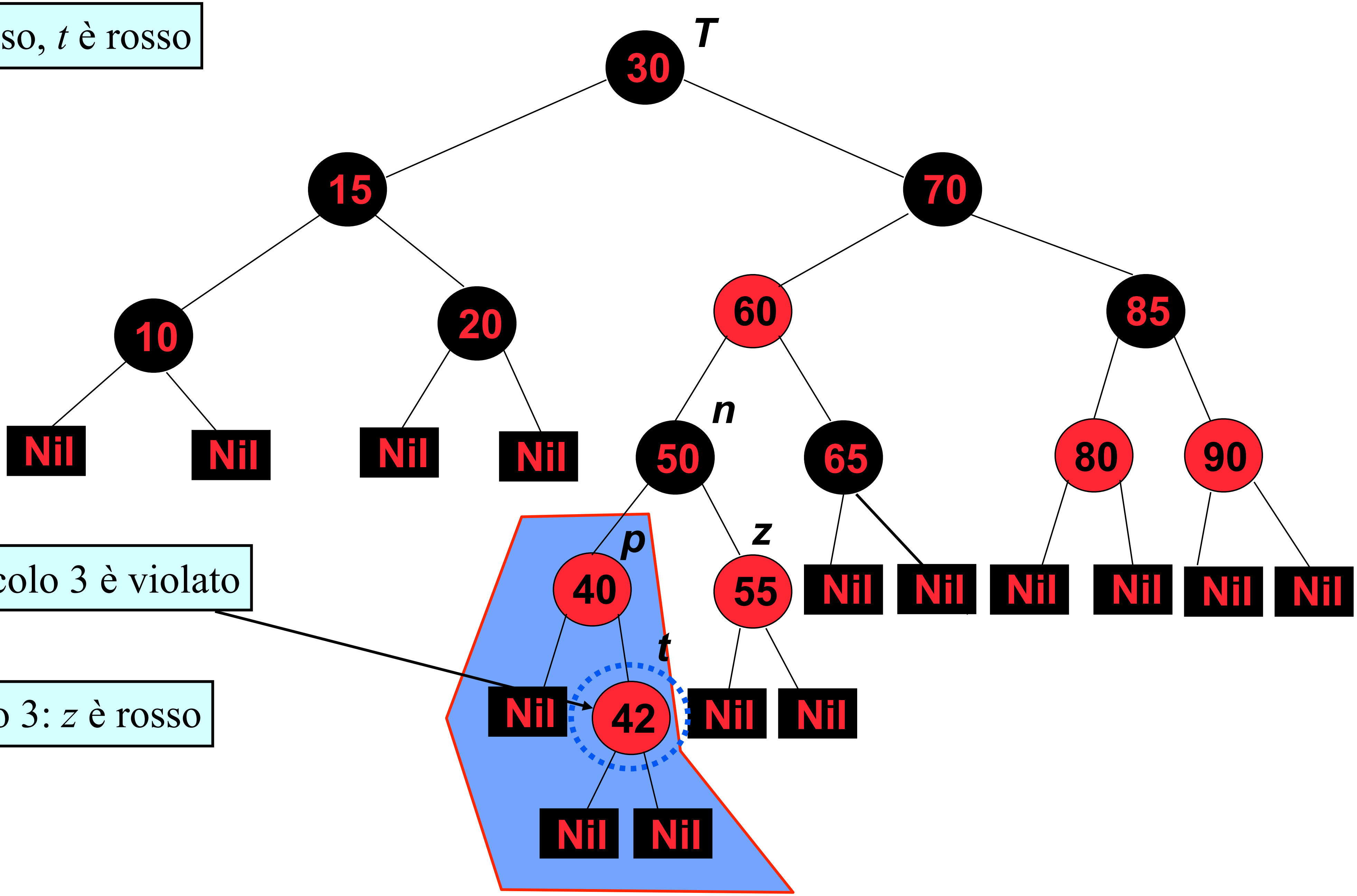


p è rosso, t è rosso



Vincolo 3 è violato

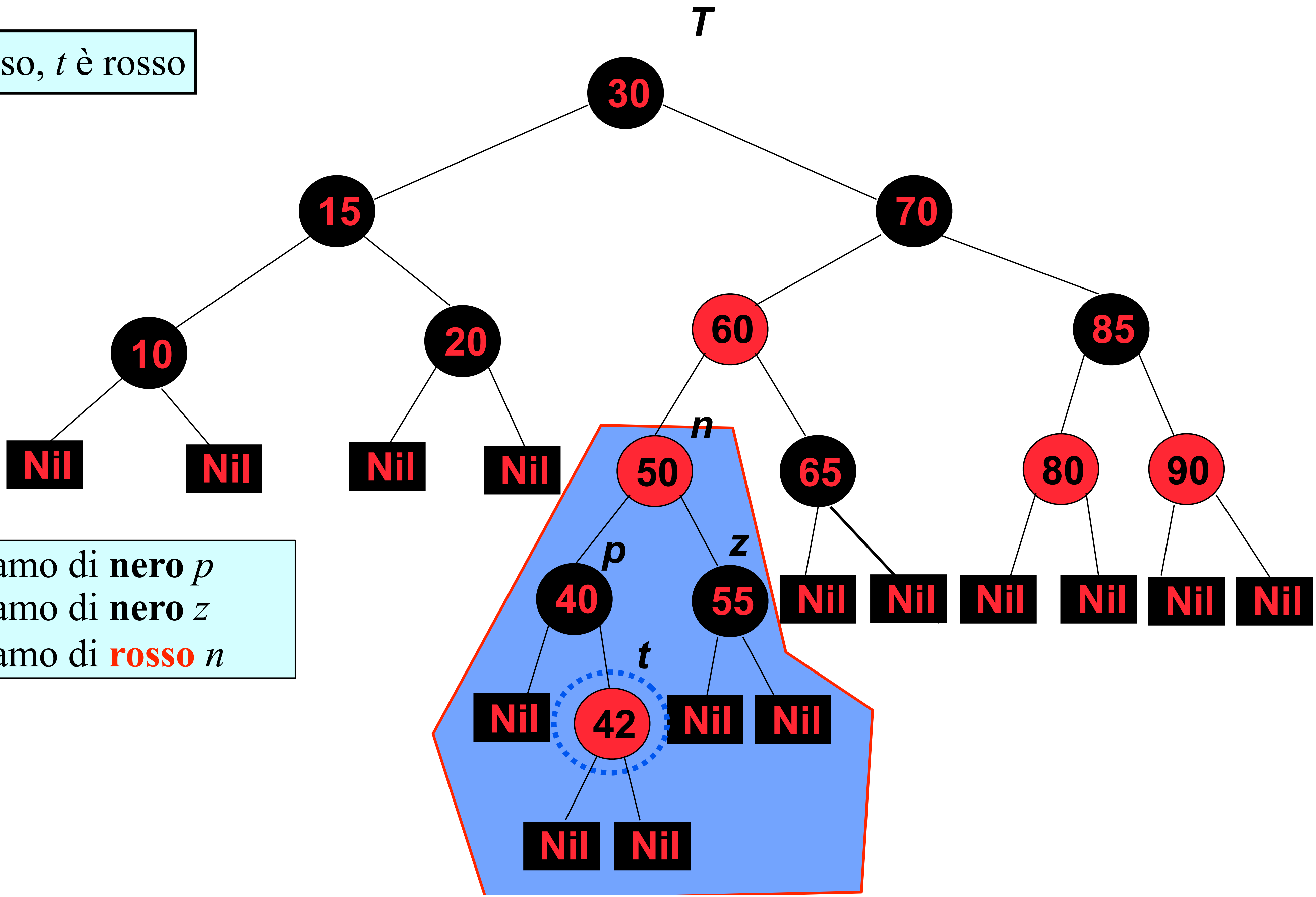
p è rosso, t è rosso



Vincolo 3 è violato

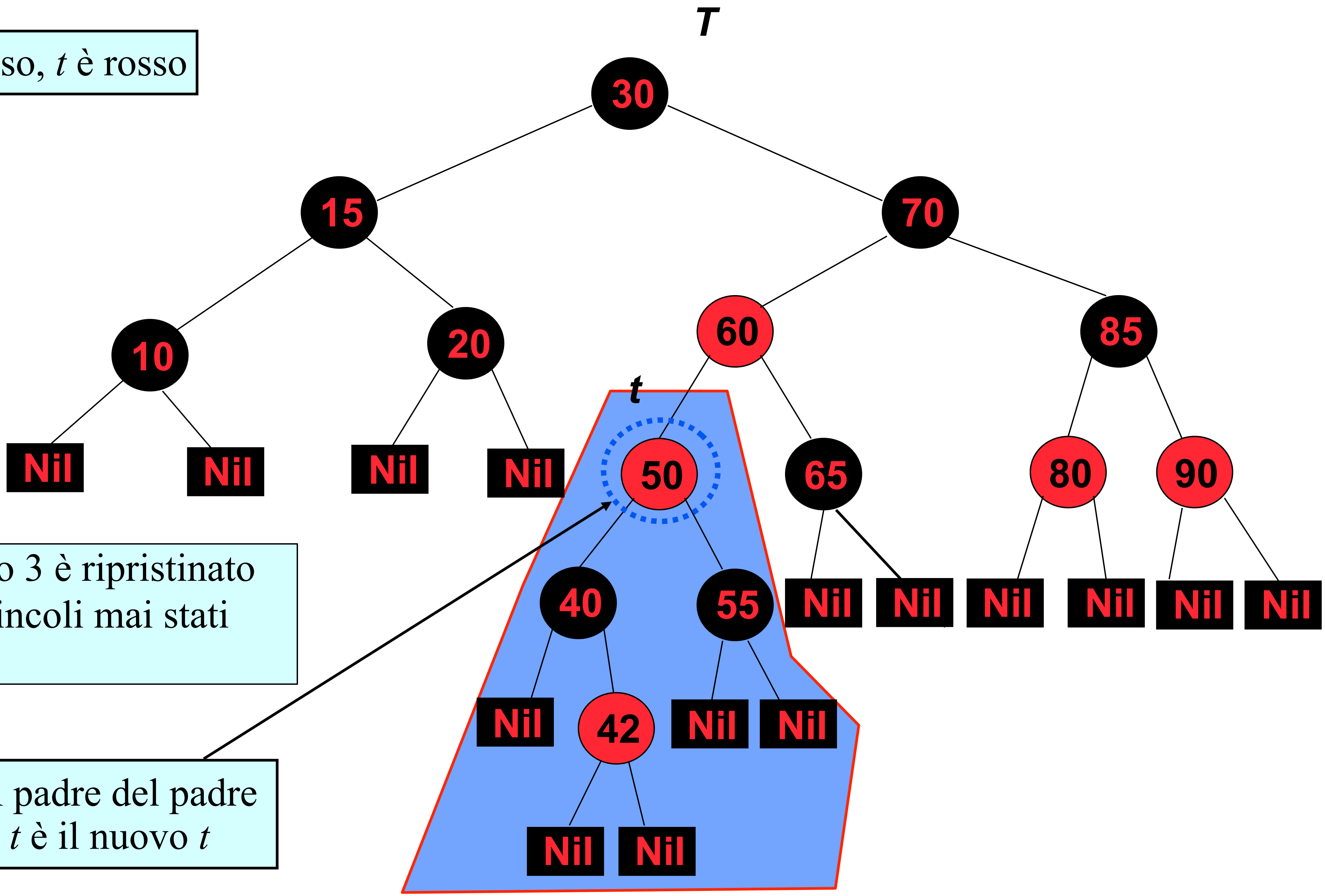
Caso 3: z è rosso

p è rosso, t è rosso



Coloriamo di **nero** p
Coloriamo di **nero** z
Coloriamo di **rosso** n

p è rosso, t è rosso

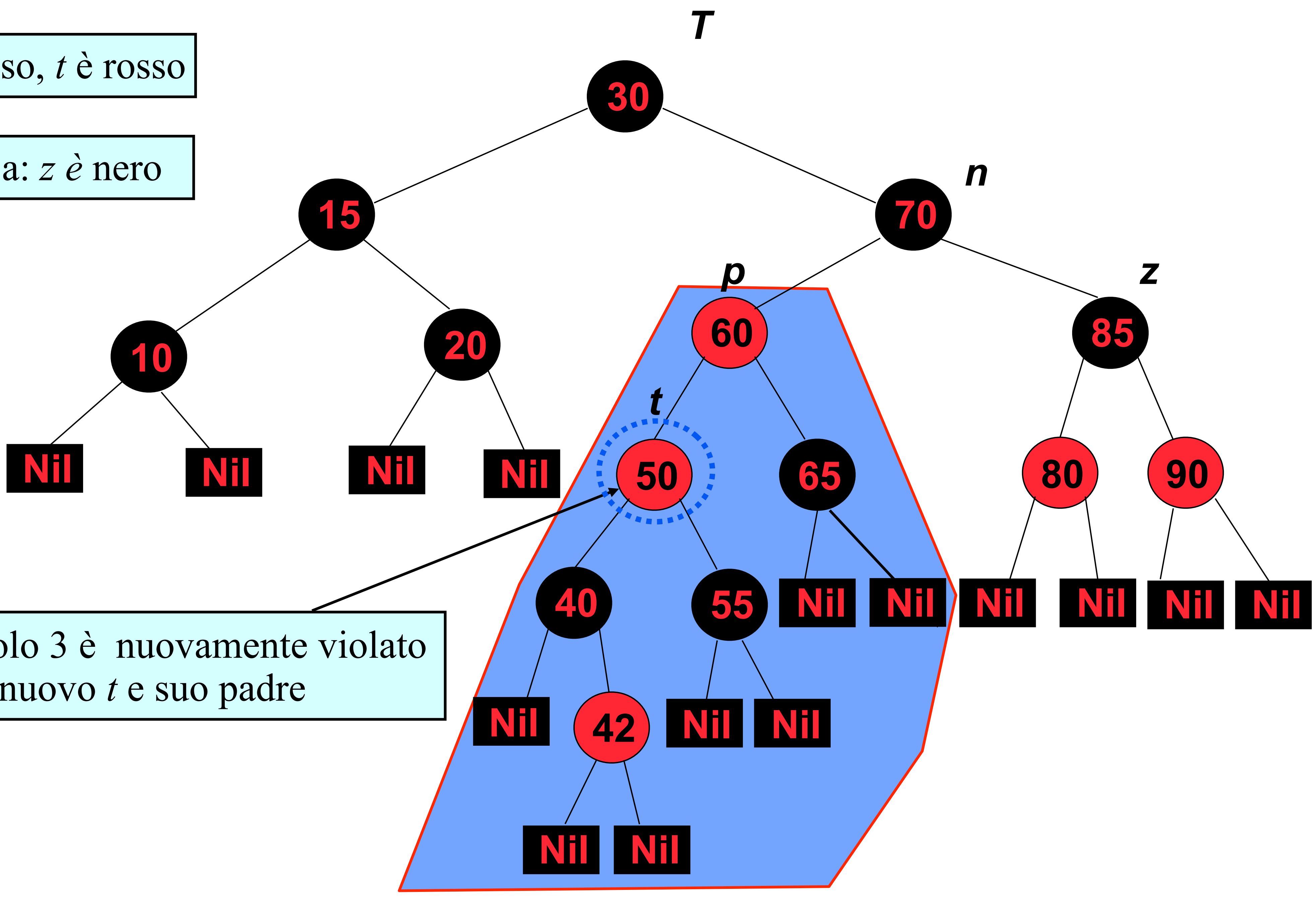


Vincolo 3 è ripristinato
Altri vincoli mai stati violati

Il padre del padre di t è il nuovo t

p è rosso, *t* è rosso

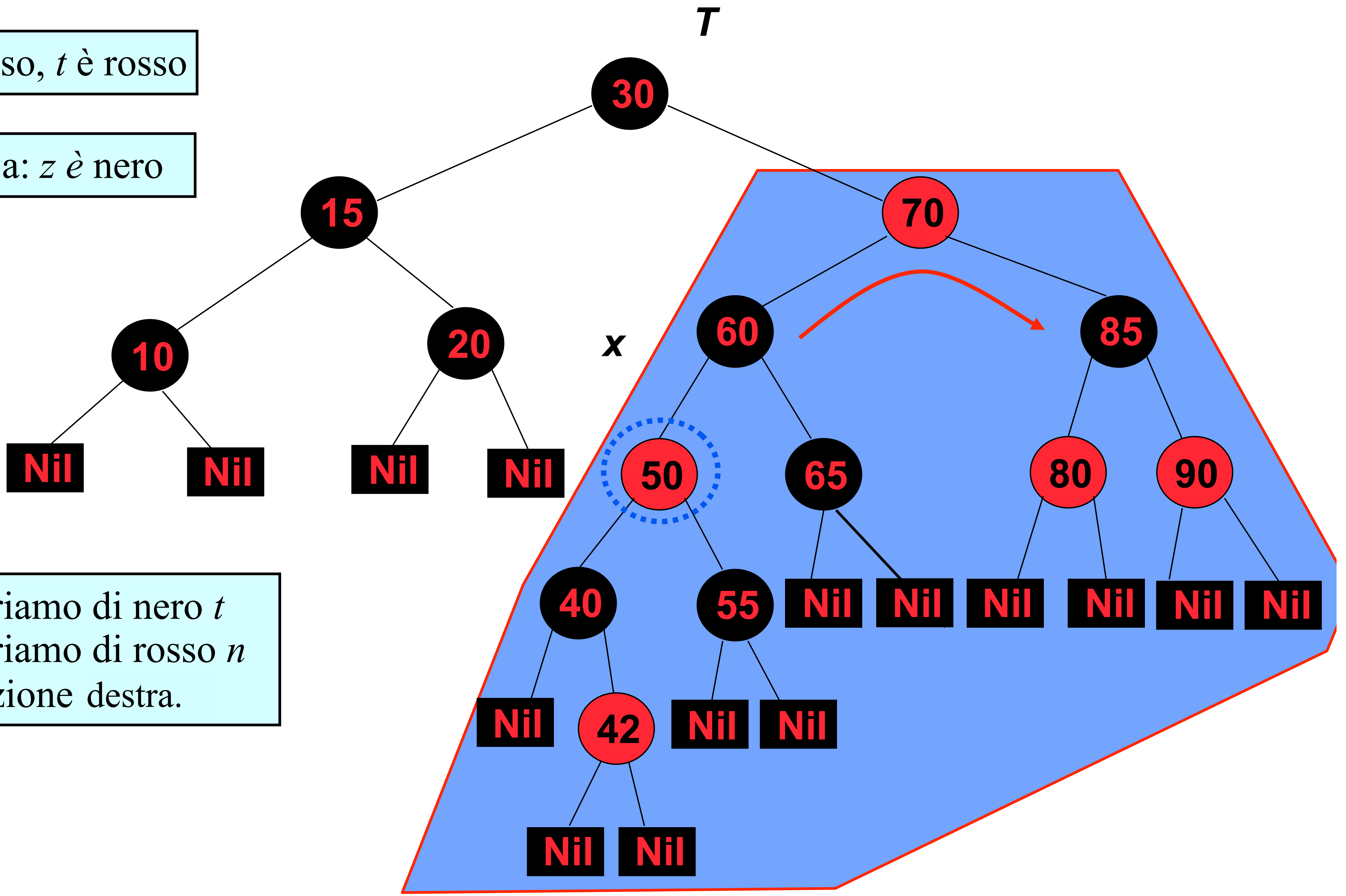
Caso 5a: *z è* nero



Vincolo 3 è nuovamente violato tra il nuovo *t* e suo padre

p è rosso, t è rosso

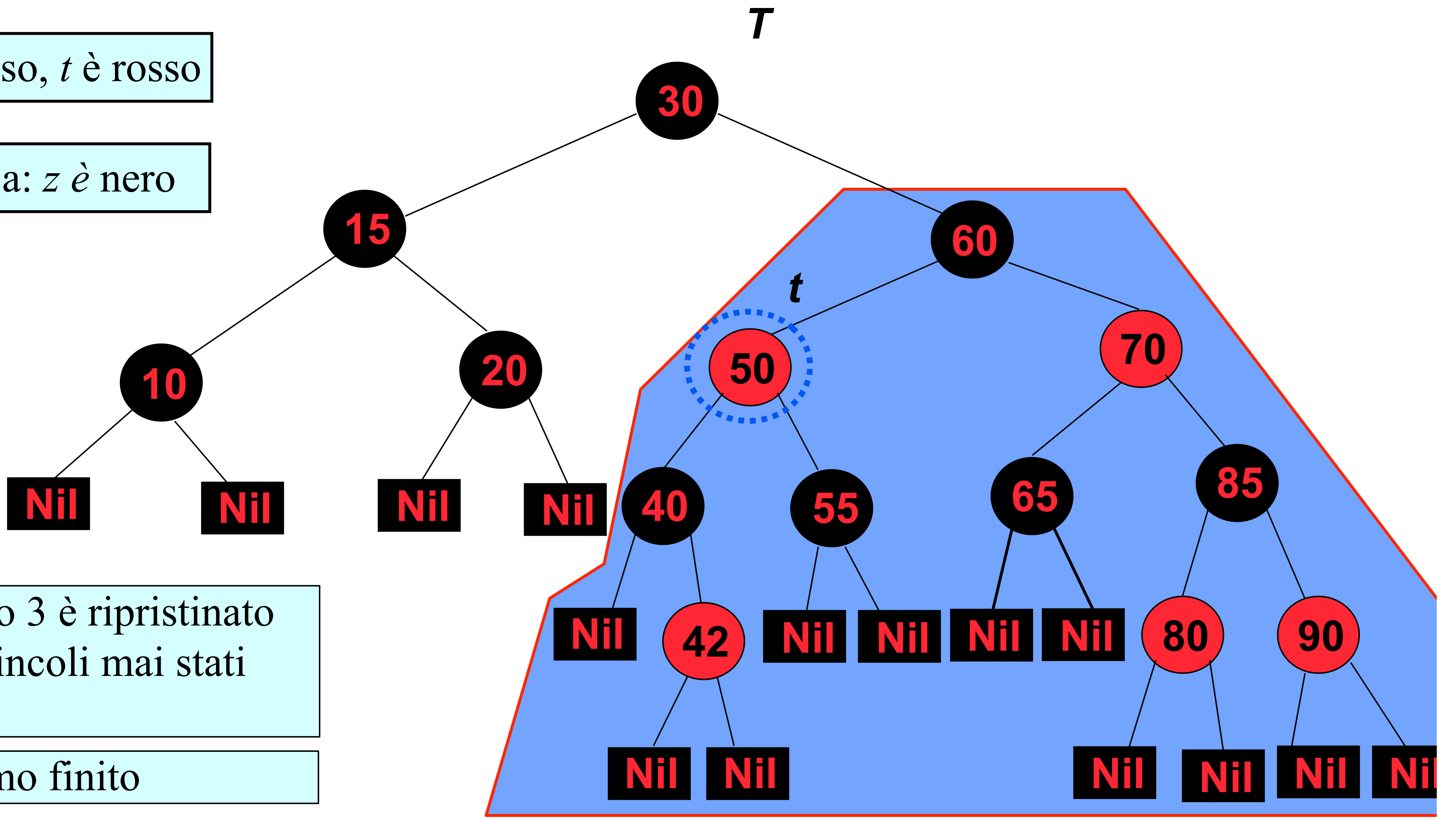
Caso 5a: z è nero



Coloriamo di nero t
Coloriamo di rosso n
Rotazione destra.

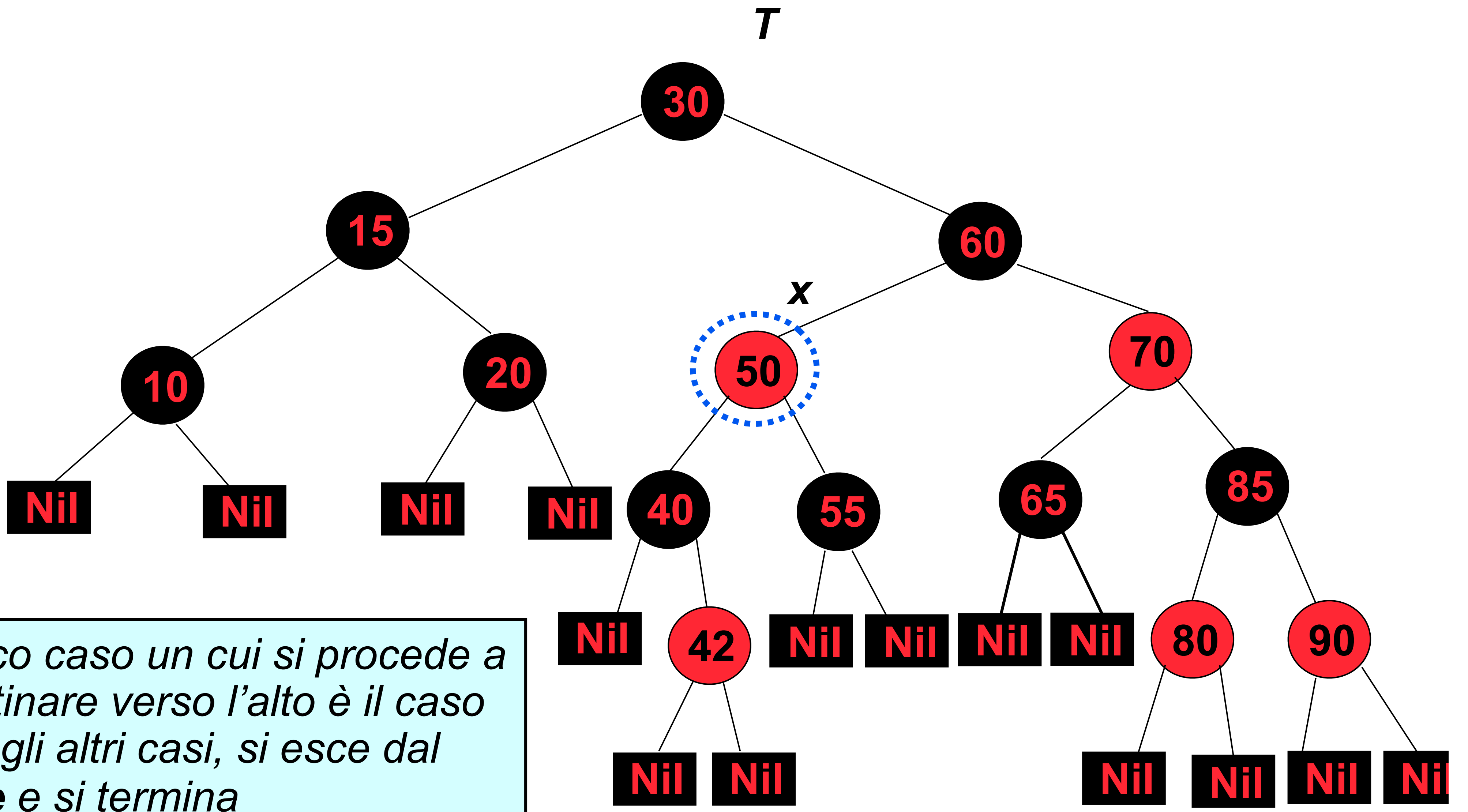
p è rosso, *t* è rosso

Caso 5a: *z è* nero



Vincolo 3 è ripristinato
Altri vincoli mai stati violati

Abbiamo finito



L'unico caso un cui si procede a ripristinare verso l'alto è il caso 3. Negli altri casi, si esce dal ***while*** *e si termina*

- **Totale: *O(log n)***
  - *O(log n)* per scendere fino al punto di inserimento
  - *O(1)* per effettuare l'inserimento
  - *O(log n)* per risalire e “aggiustare”
    (caso pessimo – solo nel caso 3)
- **Esiste anche la possibilità di effettuare una “top-down” insertion**
  - Scendere fino al punto di inserimento, “aggiustando” l'albero mano a mano
  - Effettuare l'inserimento in una foglia

- **L’algoritmo di cancellazione per alberi RB è costruito sull’algoritmo di cancellazione per alberi binari di ricerca**

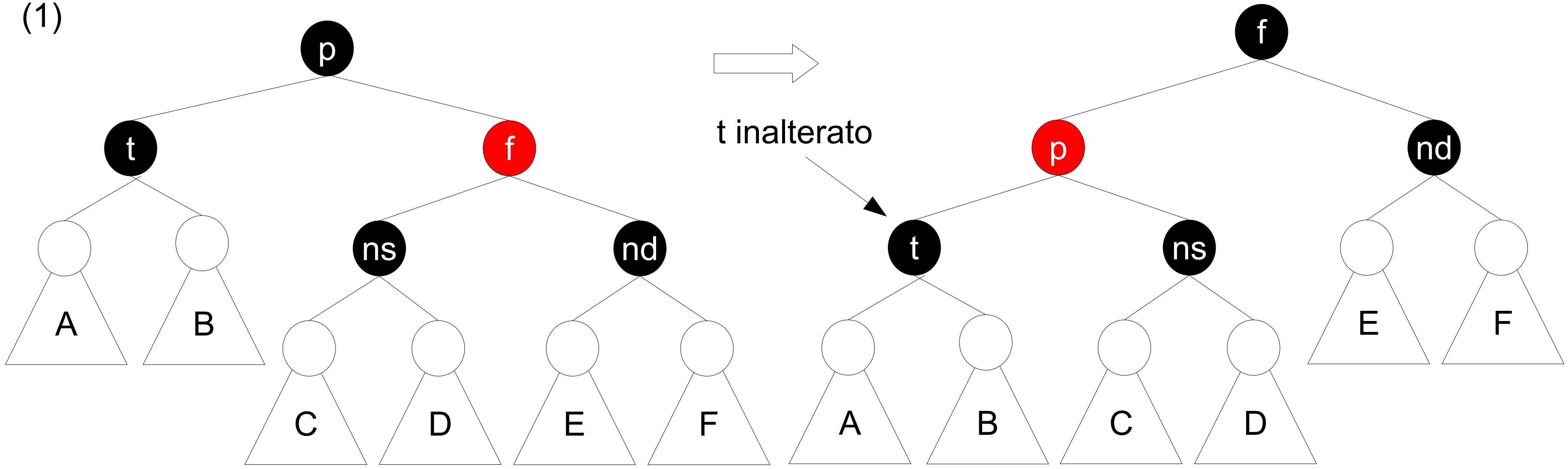
- **Dopo la cancellazione si deve decidere se è necessario ribilanciare o meno**

- **Le operazioni di ripristino del bilanciamento sono necessarie solo quando il nodo cancellato è nero! (perché?)**

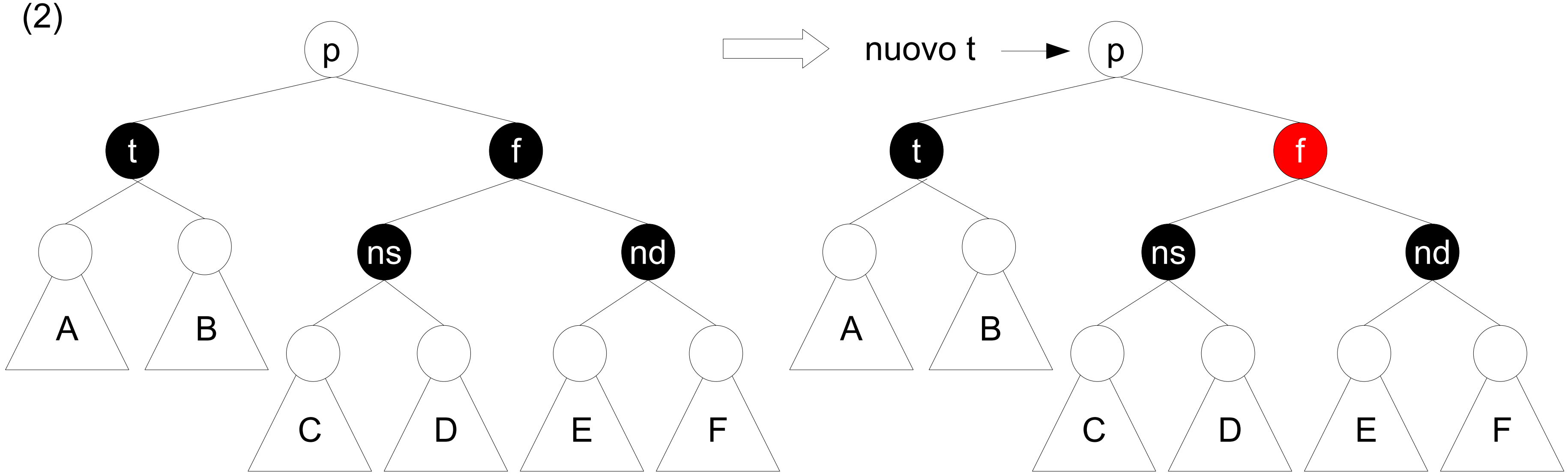
- **Se il nodo "cancellato" è rosso**
  - altezza nera invariata
  - non sono stati creati nodi rossi consecutivi
  - la radice resta nera
- **Se il nodo "cancellato" è nero**
  - possiamo violare il vincolo 1: la radice può essere un nodo rosso
  - possiamo violare il vincolo 3: se il padre e uno dei figli del nodo cancellato erano rossi
  - abbiamo violato il vincolo 4: altezza nera cambiata
- **L'algoritmo balanceDelete(*T*,*t*) ripristina la proprietà Red-Black con rotazioni e cambiamenti di colore:**
  - ci sono 4 casi possibili (e 4 simmetrici)!

```
Tree removeNode(Tree T, Item x)
  Tree u ← lookupNode(T, x)
  if u ≠ nil then
      if u.left ≠ nil and u.right ≠ nil then                        % Caso 3
          Tree s ← u.right
          while s.left ≠ nil do s ← s.left
          u.key ← s.key
          u.value ← s.value
          u ← s
      Tree t
      if u.left ≠ nil and u.right = nil then          % Caso (2) - Solo figlio sx
          t ← u.left
      else                              % Caso (2) - Solo figlio dx / Caso (1)
          t ← u.right
      link(u.parent, t, x)
      if u.parent = nil then T = t      if u.color = BLACK then balanceDelete(T, t)
      delete u
  return T
```

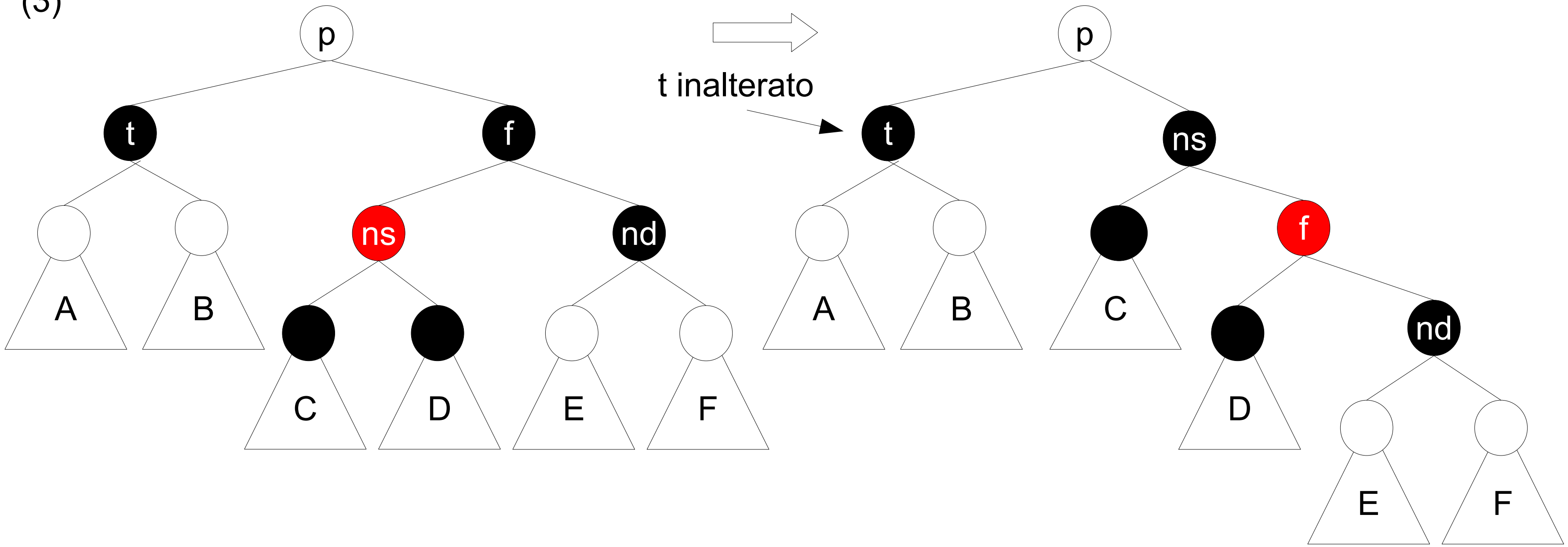
(1)


p
t
f
ns
nd
A
B
C
D
E
F
t inalterato
f
p
nd
t
ns
E
F
A
B
C
D

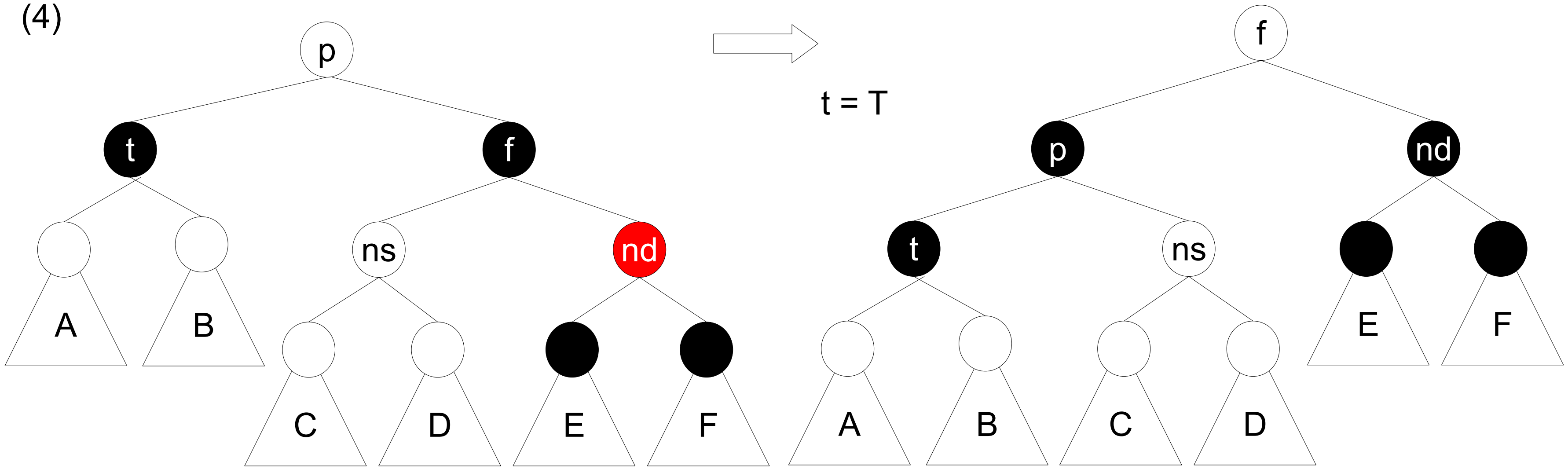
(2)


p
t
f
ns
nd
A
B
C
D
E
F
nuovo t
p
t
f
ns
nd
A
B
C
D
E
F

(3)


p
t
f
A
B
ns
nd
C
D
E
F
t inalterato
p
t
ns
A
B
C
f
D
nd
E
F

(4)


p
t
f
A
B
ns
nd
C
D
E
F
t = T
f
p
nd
t
ns
E
F
A
B
C
D

```
balanceDelete(TREE T, TREE t)
while t ≠ T and t.color = BLACK do
    TREE p ← t.parent                                              % Padre
    if t = p.left() then
        TREE f ← p.right                                           % Fratello
        TREE ns ← f.left                                           % Nipote sinistro
        TREE nd ← f.right                                          % Nipote destro
        if f.color = RED then                                      % (1)
            p.color ← RED
            f.color ← BLACK
            rotateLeft(p)
            % t viene lasciato inalterato, quindi si ricade nei casi 2,3,4
        else
            if ns.color = nd.color = BLACK then                    % (2)
                f.color ← RED
                t ← p
            else if ns.color = RED and nd.color = BLACK then       % (3)
                ns.color ← BLACK
                f.color ← RED
                rotateRight(f)
                % t viene lasciato inalterato, quindi si ricade nel caso 4
            else if nd.color = RED then                            % (4)
                f.color ← p.color
                p.color ← BLACK
                nd.color ← BLACK
                rotateLeft(p)
                t ← nil
    else
        % Casi (5)-(8) speculari a (1)-(4)
if t ≠ nil then t.color ← BLACK
```

- **L’operazione di cancellazione è concettualmente complicata!**
- **Ma efficiente:**
  - Dal caso (1) si passa ad uno dei casi (2), (3), (4)
  - Dal caso (2) si torna ad uno degli altri casi, ma risalendo di un livello l’albero
  - Dal caso (3) si passa al caso (4)
  - Nel caso (4) si termina
- **Quindi**
  - In altre parole, è possibile visitare al massimo un numero $O(\log n)$ di casi, ognuno dei quali è gestito in tempo $O(1)$