Algoritmi e Strutture Dati

Alberi bilanciati

Sabrina De Capitani di Vimercati

Università degli Studi di Milano

Ricordate...

- Gli alberi binari di ricerca permettono di trovare un elemento in tempo O(h)
  - nel caso peggiore h=O(n)
  - nel caso di albero bilanciato h=O(log n)
- Nozione intuitiva di bilanciamento: tutti i rami di un albero hanno approssimativamente la stessa lunghezza
- Caso ideale per un albero k-ario:
  - ciascun nodo ha 0 o k figli
  - la lunghezza di due rami qualsiasi differisce di al più una unità

1

Bilanciamento perfetto

- Un albero binario perfettamente bilanciato di n nodi ha altezza $\lfloor \log n \rfloor + 1$
- Se ogni nodo ha 0 oppure 2 figli allora si ha $n_f = n_i + 1$ che dove n_f è il numero di foglie ed n_i il numero di nodi interni
- Le foglie sono circa il 50% dei nodi
- Generalizzando, per alberi di arità k si ha $n_f=(k-1)n_i+1$
- Il costo delle operazioni di ricerca/inserimento/cancellazione è pertanto O(log n)
- Ripetuti inserimenti e cancellazioni possono distruggere il bilanciamento con un conseguente degrado delle prestazioni

2

Fattore di bilanciamento

- Il fattore di bilanciamento $\beta(v)$ di un nodo v è la massima differenza di altezza fra i sottoalberi di v
  - es.,: l'albero perfetto è un albero dove $\beta(v)=0$ per ogni nodo v
- In questo caso si parla di bilanciamento in altezza

3

Alberi bilanciati

Alberi AVL (Adel'son-Vel'skii e Landis, 1962)

- β(v) ≤ 1 per ogni nodo *v*
- Bilanciamento ottenuto tramite rotazioni

Alberi Rosso-Neri (Bayer, 1972)

Alberi 2-3 (Hopcroft, 1983)

- β(v) = 0 per ogni nodo v
- Bilanciamento ottenuto tramite merge/split, grado variabile

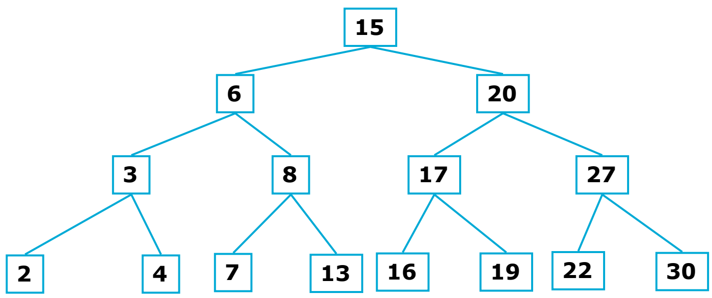
B-Alberi (Bayer, McCreight, 1972)

- β(v) = 0 per ogni nodo v
- Usati per strutture in memoria secondaria

4

Alberi AVL

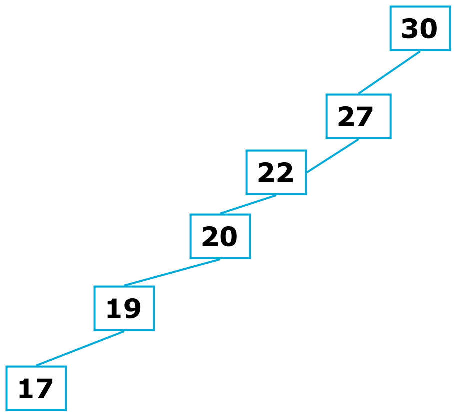
- Alberi binari di ricerca bilanciati in altezza
- β(v) = altezza del sottoalbero sinistro di v – altezza del sottoalbero destro di v



5

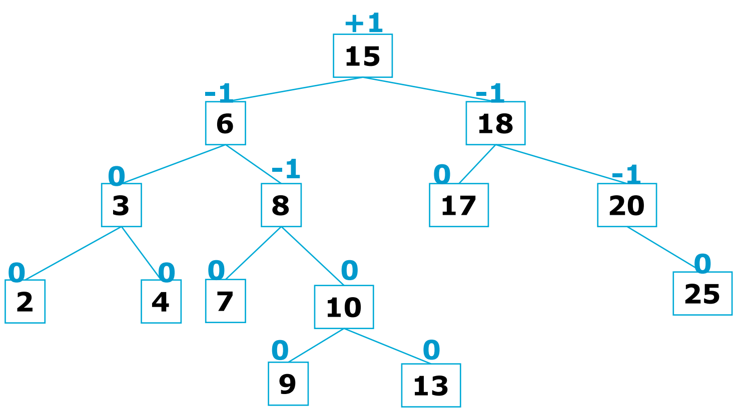
Alcuni esempi (1)

È un albero AVL? (per convenzione l'altezza di un albero vuoto è -1)



6

Alcuni esempi (2)



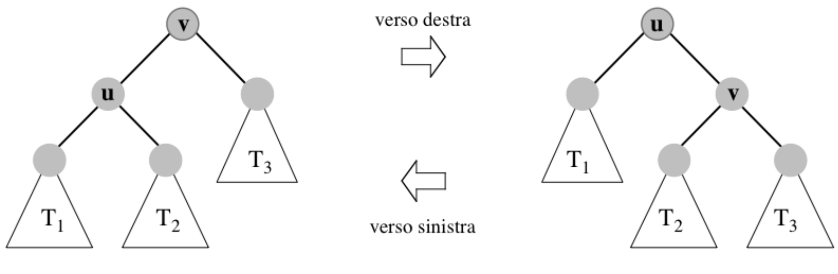
7

Operazioni su alberi AVL

- La ricerca viene effettuata applicando la stessa procedura usata per gli alberi binari di ricerca
- Inserimenti e cancellazioni possono sbilanciare l'albero
- Il bilanciamento viene preservato attraverso opportune rotazioni

8

Rotazione base



9

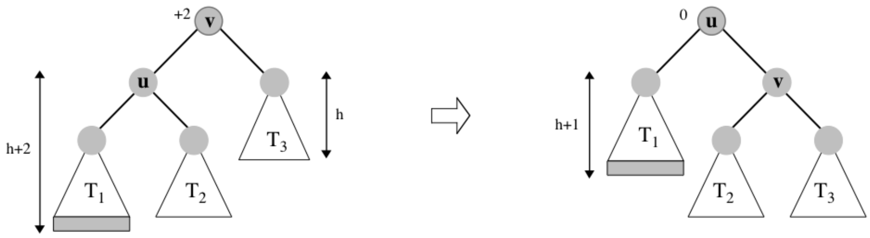
Ribilanciamento

- Si ricalcolano i fattori di bilanciamento solo nel ramo interessato dall'inserimento/cancellazione (gli altri fattori non possono mutare), dal basso verso l'alto
- Se appare un fattore di bilanciamento pari a +/- 2 occorre ribilanciare per mezzo di rotazioni
- Si distinguono i seguenti casi:
  - DD inserimento nel sottoalbero destro di un figlio destro
  - SD inserimento nel sottoalbero destro di un figlio sinistro
  - DS inserimento nel sottoalbero sinistro di un figlio destro
  - SS inserimento nel sottoalbero sinistro di un figlio sinistro

10

Rotazione SS

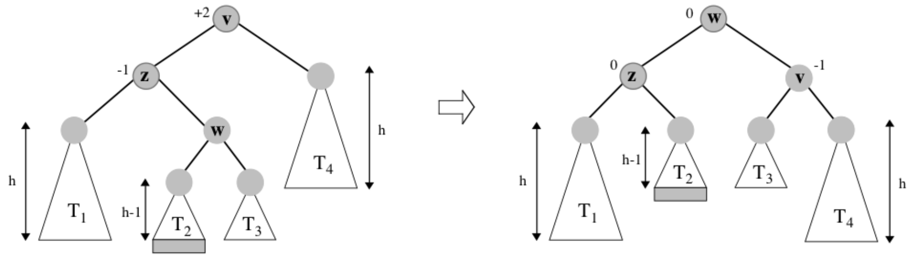
- Applicare una rotazione semplice verso destra su v
- L'altezza dell'albero coinvolto nella rotazione passa da h+3 a h+2



11

Rotazione SD

- Applicare due rotazioni semplici: una sul figlio del nodo critico, l'altra sul nodo critico
- L'altezza dell'albero coinvolto nella rotazione passa da h+3 a h+2



12

Inserimento in un albero AVL (1)

- Si trova (come per alberi binari di ricerca) la posizione in cui inserire il nuovo elemento
  - durante tale operazione memorizzato il puntatore al più basso nodo v il cui fattore di bilanciamento è -1 o +1
- Viene effettuato l'inserimento
- Vengono aggiornati i fattori di bilanciamento dei nodi lungo il percorso tra il nodo appena inserito e v
  - tutti i nodi lungo questo percorso avevano un fattore di bilanciamento 0 e vengono portati a +1 o -1
  - l'unico nodo che potrebbe registrare uno sbilanciamento è proprio v (nodo critico)

13

Inserimento in un albero AVL (2)

- Se il fattore di bilanciamento di v è +1 (-1) ed il nodo è stato inserito nel sottoalbero destro (sinistro), il sottoalbero con radice v è automaticamente bilanciato
  - in caso contrario, il sottoalbero è sbilanciato e richiede un ribilanciamento
- Viene determinata ed effettuata la rotazione necessaria
- Tutti questi passi, rotazioni comprese, interessano solo i nodi lungo il cammino dalla radice alla nuova foglia inserita

14

Cancellazione in un albero AVL

- Si effettua la cancellazione di un elemento come descritto dall'algoritmo Tree-Delete
- Si ricalcolano i fattori di bilanciamento
  - osserviamo che i soli fattori di bilanciamento che possono cambiare sono quelli dei nodi lungo il cammino dalla radice al nodo eliminato e che questi possono essere ricalcolati risalendo l'albero dal basso verso l'alto
- Eseguiamo una rotazione per ogni nodo il cui fattore di bilanciamento è ±2

15

Complessità operazioni su alberi AVL

Tutte le operazioni hanno costo O(log n) poiché l'altezza dell'albero è O(log n) e ciascuna rotazione richiede solo tempo costante

16

Alberi rosso-neri (1)

- Gli alberi rosso-neri sono alberi binari di ricerca in cui le operazioni Tree-Insert e Tree-Delete sono opportunamente modificate in modo tale da garantire un'altezza dell'albero h = O(log n)

- A tale scopo si aggiunge un bit per ogni nodo: il colore che può essere **rosso** o **nero**

- Sono alberi binari di ricerca che soddisfano delle proprietà ulteriori

17

Alberi rosso-neri (2)

1. La radice è nera;
2. I nodi nil sono neri;
3. Se un nodo è rosso, allora entrambi i suoi figli sono neri;
4. Ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri

18

Rappresentazione alberi rosso-neri

- Ogni nodo mantiene:
  - parent: puntatore al genitore
  - left, right: puntatori ai figli sinistro/destro
  - color: colore
  - key, data: chiave e dati
- Nodo nil
  - nodo sentinella che evita di trattare diversamente i puntatori ai nodi dai puntatori nil
  - al posto di un puntatore nil, si usa un puntatore ad un nodo nil
    - ne esiste solo uno, per risparmiare memoria
    - nodo con figli nil → foglia nell'albero rosso-nero corrispondente

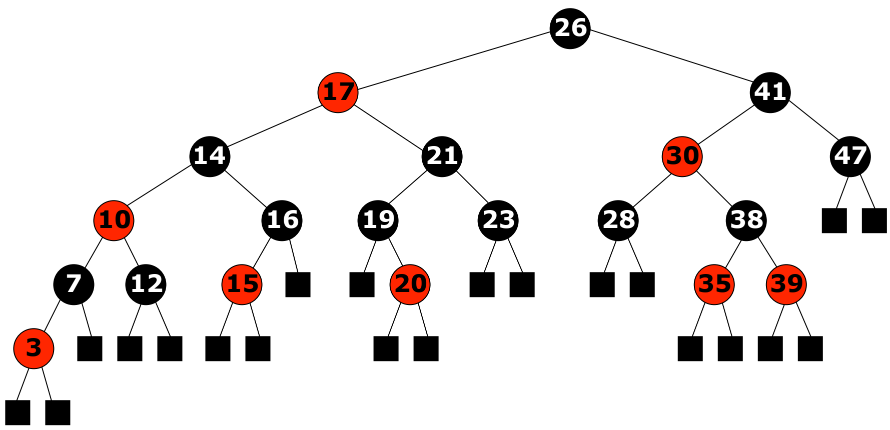
19

Alcune definizioni

- Il numero di nodi neri lungo ogni percorso da un nodo v (escluso) ad una foglia è detto altezza nera di v, indicato bh(v)
  - ben definito perché tutti i percorsi hanno lo stesso numero di nodi neri (regola 4)
- L'altezza nera di un albero RB è l'altezza nera della sua radice

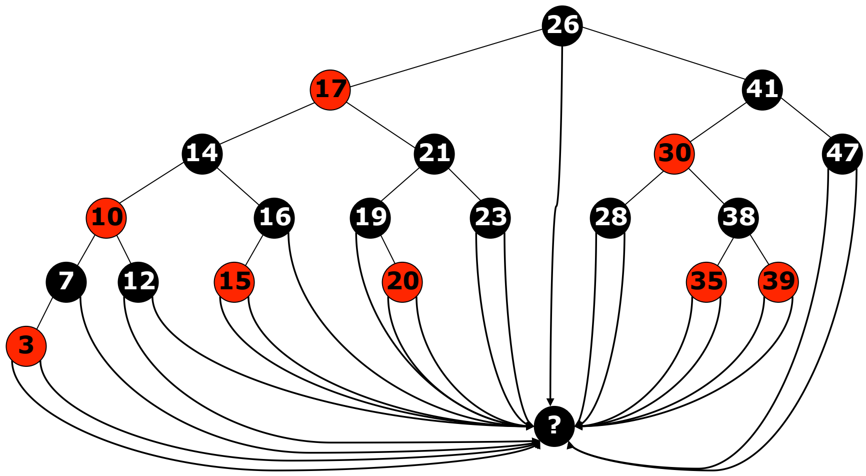
20

Esempio di albero rosso-nero senza sentinella



21

Esempio di albero rosso-nero con sentinella



22

Altezza massima di un albero rosso-nero

Un albero rosso-nero con n nodi interni ha altezza

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

Perché?

- Il sottoalbero con radice in x contiene almeno $2^{bh(x)} - 1$ nodi interni (vediamo perché)
  - se altezza di x è 0 allora x è foglia ed il sottoalbero di cui x è radice contiene almeno $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nodi interni
  - per il passo induttivo, sia x un nodo interno con due figli con altezza nera pari a bh(x) o bh(x)-1 a secondo del loro colore
    - Per ipotesi induttiva ogni figlio ha almeno $2^{bh(x)-1} - 1$ e quindi x ha almeno $2^{bh(x)} - 1$ nodi interni
- Se h è l'altezza dell'albero, per la proprietà 3 almeno metà dei nodi in qualsiasi percorso dalla radice a una foglia deve essere nera
  - L'altezza nera della radice deve essere almeno h/2 e quindi $n \geq 2^{h/2} - 1$ ovvero $h \leq 2 \log_2(n+1)$

23

Costo operazioni

- Su di un albero rosso-nero con n nodi interni le operazioni Search, Minimum, Maximum, Successor e Predecessor richiedono tempo O(log n)

- Le operazioni Insert e Delete su di un albero rosso-nero richiedono tempo O(log n) ma siccome esse modificano l'albero possono violare le proprietà degli alberi rosso-neri ed in tal caso occorre ripristinare tali proprietà

24

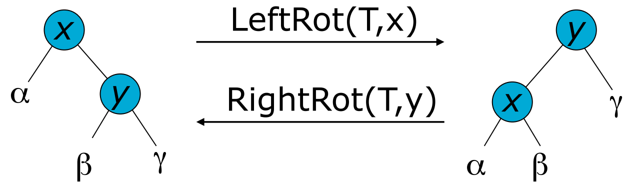
Rotazioni

Per ripristinare le proprietà degli alberi rosso-neri si deve:

- modificando i colori nella zona della violazione operando dei ri-bilanciamenti dell'albero tramite rotazioni:
  - rotazione destra
  - rotazione sinistra

25

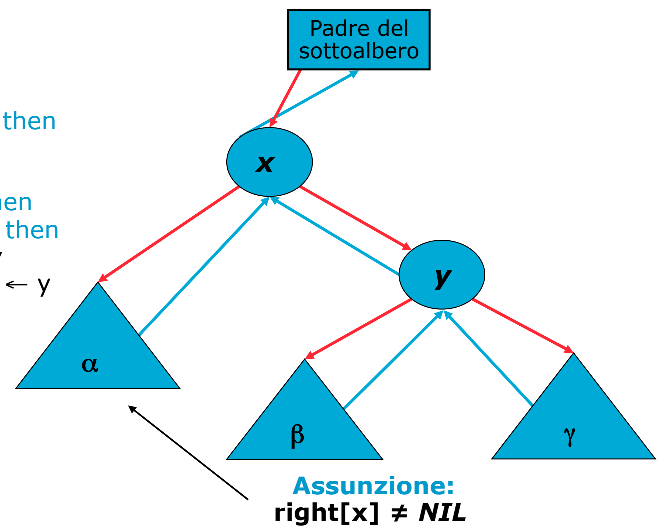
Esempio di rotazioni



26

Rotazione a sinistra (1)

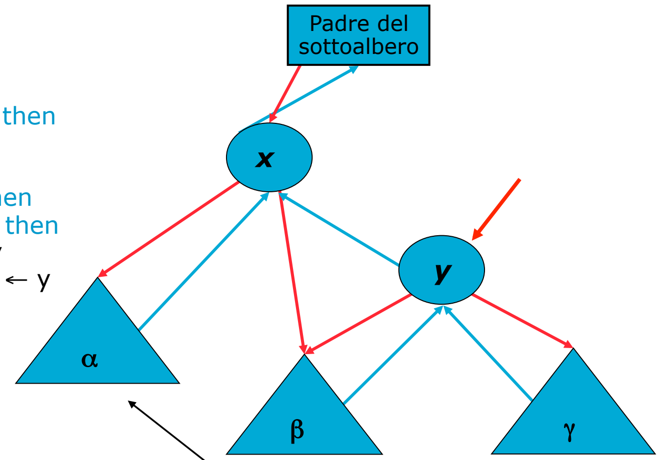
```
Left-rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if left[y] ≠ nil[T] then
    p[left[y]] ← x
  p[y] ← p[x]
  if p[x] ≠ nil[T] then
    if x = left[p[x]] then
      left[p[x]] ← y
    else right[p[x]] ← y
  else root[T] ← y
  left[y] ← x
  p[x] ← y
```



27

Rotazione a sinistra (2)

```
Left-rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if left[y] ≠ nil[T] then
    p[left[y]] ← x
  p[y] ← p[x]
  if p[x] ≠ nil[T] then
    if x = left[p[x]] then
       left[p[x]] ← y
    else right[p[x]] ← y
  else root[T] ← y
  left[y] ← x
  p[x] ← y
```

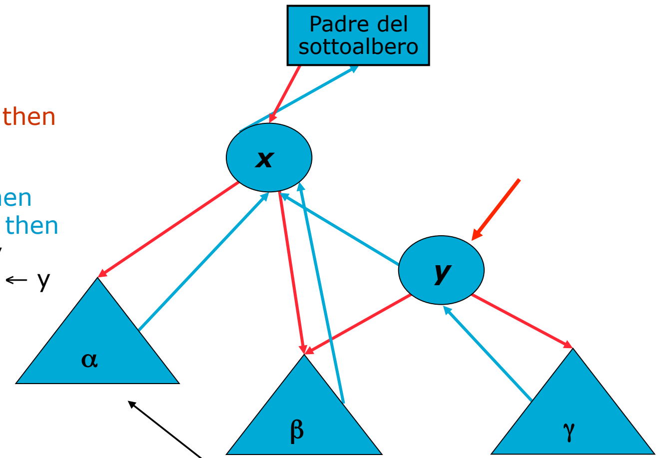


Assunzione:
right[x] ≠ *NIL*

28

Rotazione a sinistra (3)

```
Left-rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if left[y] ≠ nil[T] then
    p[left[y]] ← x
  p[y] ← p[x]
  if p[x] ≠ nil[T] then
    if x = left[p[x]] then
       left[p[x]] ← y
    else right[p[x]] ← y
  else root[T] ← y
  left[y] ← x
  p[x] ← y
```

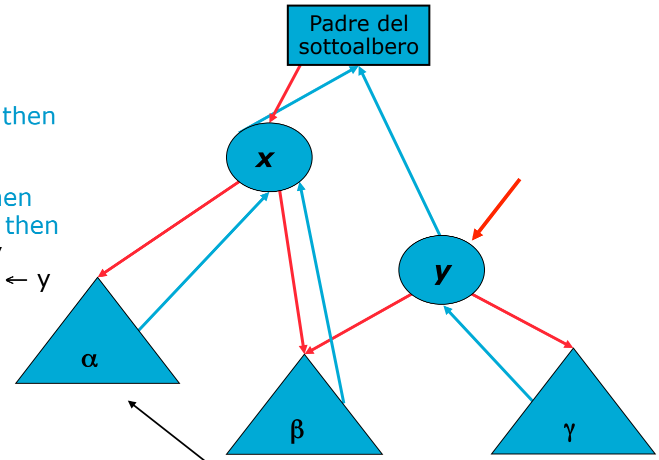


Assunzione:
right[x] ≠ *NIL*

29

Rotazione a sinistra (4)

```
Left-rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if left[y] ≠ nil[T] then
    p[left[y]] ← x
  p[y] ← p[x]
  if p[x] ≠ nil[T] then
    if x = left[p[x]] then
       left[p[x]] ← y
    else right[p[x]] ← y
  else root[T] ← y
  left[y] ← x
  p[x] ← y
```

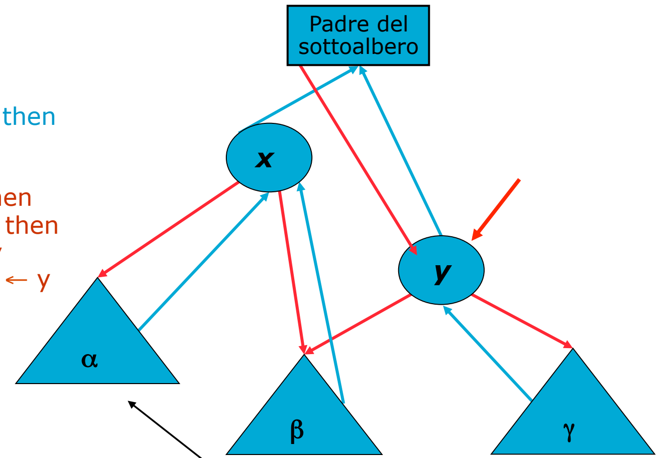


Assunzione:
right[x] ≠ *NIL*

30

Rotazione a sinistra (5)

```
Left-rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if left[y] ≠ nil[T] then
    p[left[y]] ← x
  p[y] ← p[x]
  if p[x] ≠ nil[T] then
    if x = left[p[x]] then
       left[p[x]] ← y
    else right[p[x]] ← y
  else root[T] ← y
  left[y] ← x
  p[x] ← y
```

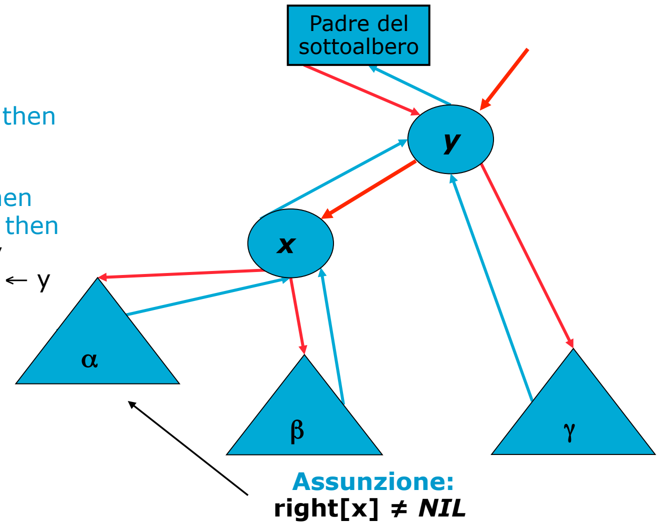


Assunzione:
right[x] ≠ *NIL*

31

Rotazione a sinistra (6)

```
Left-rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if left[y] ≠ nil[T] then
    p[left[y]] ← x
  p[y] ← p[x]
  if p[x] ≠ nil[T] then
    if x = left[p[x]] then
       left[p[x]] ← y
    else right[p[x]] ← y
  else root[T] ← y
  left[y] ← x
  p[x] ← y
```



32

Rotazione a destra

```
Right-Rotate(T,y)
   x ← left[y]
   left[y] ← right[x]
   p[right[x]] ← y     ▷ right[x] può essere nil[T]
   p[x] ← p[y]
   if p[y] = nil[T] then
        root[T] ← x
   else if y = left[p[y]] then
        left[p[y]] ← x
   else
        right[p[y]] ← x
   p[y] ← x
   right[x] ← y
```

33

Inserimento

- Ricerca della posizione usando la stessa procedura usata per gli alberi binari di ricerca
- Coloriamo il nuovo nodo di rosso
- Se l'albero risultante non soddisfa le quattro proprietà che lo caratterizzano le si devono risistemare
  - RB-Insert-Fixup ricolora i nodi ed effettua delle rotazioni

34

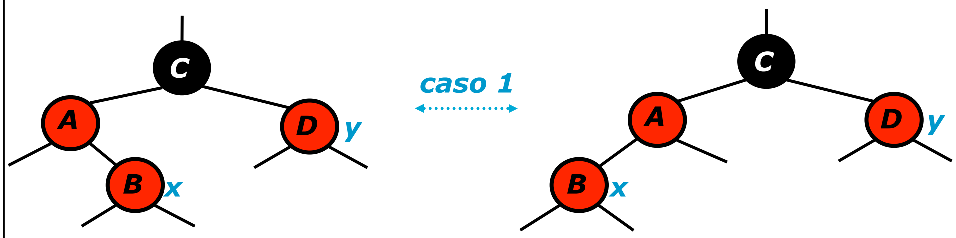
Quali proprietà possono essere violate?

- La proprietà 2 (i nodi nil sono neri) è soddisfatta perché il nodo inserito è rosso ed i due figli sono la sentinella nil[T] che è nera
- La proprietà 4 (ogni percorso da un nodo interno ad una foglia ha lo stesso numero di nodi neri) è soddisfatta perché il nodo inserito sostituisce la sentinella (nera) ed esso è rosso con figli neri
- Le proprietà che possono essere violate sono la 1 (la radice è nera) e la 3 (se un nodo è rosso entrambi i figli sono neri):
  - la 1 viene violata se il nodo inserito è la radice
  - la 3 viene violata se il padre del nodo inserito è rosso

35

Ribilanciamenti: Caso 1

x è il nodo inserito che causa il ribilanciamento
y è il fratello *del padre* di x

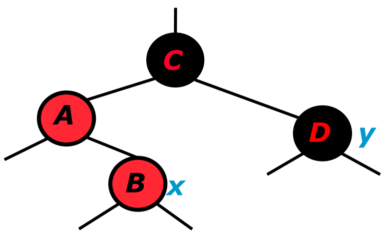


Caso 1: lo zio *y* di *x* è rosso

36

Ribilanciamenti: Caso 2

x è il nodo inserito che causa il ribilanciamento
y è il fratello *del padre* di x

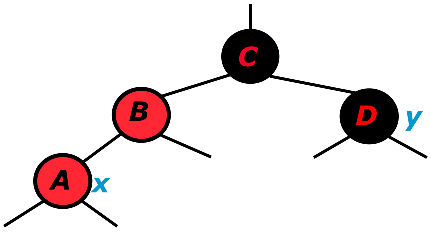


Caso 2: lo zio *y* di *x* è **nero**; x è un figlio destro

37

Ribilanciamenti: Caso 3

x è il nodo inserito che causa il ribilanciamento
y è il fratello *del padre* di x

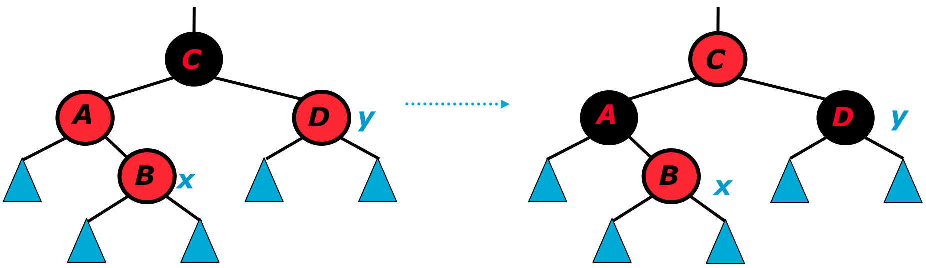


Caso 3: lo zio *y* di *x* è **nero**; x è un figlio sinistro

38

Come risolvere il caso 1

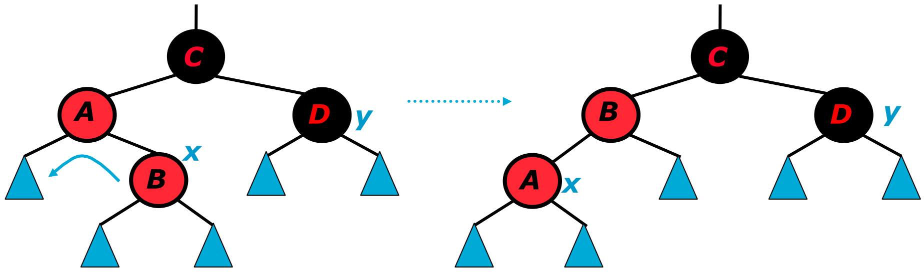
- Tutti i sono sottoalberi di uguale altezza nera
- Il colore del padre di x ed il nodo y sono posti a nero, ed il nonno di x (il nodo C) diventa rosso
- Poi si continua verso l'alto facendo del nonno di x il nuovo x



39

Come risolvere il caso 2

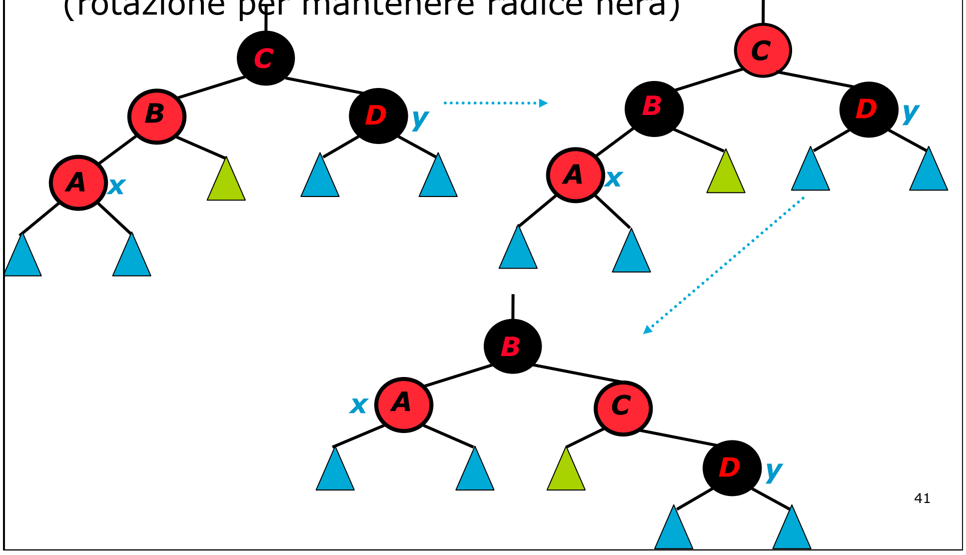
- Trasformiano il caso 2 nel caso 3 con una rotazione sinistra
- Continua a valere la proprietà 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri



40

Come risolvere il caso 3

- Cambiamo colori e si effettua una rotazione destra (rotazione per mantenere radice nera)



41

RB-Insert

```
RB-Insert(T,z)
y←nil[T] ▹tiene traccia del genitore di z
x←root[T]
while x≠nil[T] do
   y←x
   if key[z] < key[x] then
      x←left[x]
   else x←right[x]
p[z]←y
if y=nil[T] then
   root[T]←z
else if key[z] < key[y] then
         left[y]←z
      else right[y] ←z
left[z]←nil[T]
right[z]←nil[T]
color[z]←red
RB-Insert-Fixup(T,z)
```

42

RB-Insert-Fixup

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                 then color[p[z]] ← BLACK                 ▷ Case 1
                      color[y] ← BLACK                    ▷ Case 1
                      color[p[p[z]]] ← RED                ▷ Case 1
                      z ← p[p[z]]                         ▷ Case 1
                 else if z = right[p[z]]
                         then z ← p[z]                    ▷ Case 2
                              LEFT-ROTATE(T, z)           ▷ Case 2
                      color[p[z]] ← BLACK                 ▷ Case 3
                      color[p[p[z]]] ← RED                ▷ Case 3
                      RIGHT-ROTATE(T, p[p[z]])            ▷ Case 3
          else (same as then clause
                      with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

43

Complessità operazione di inserimento

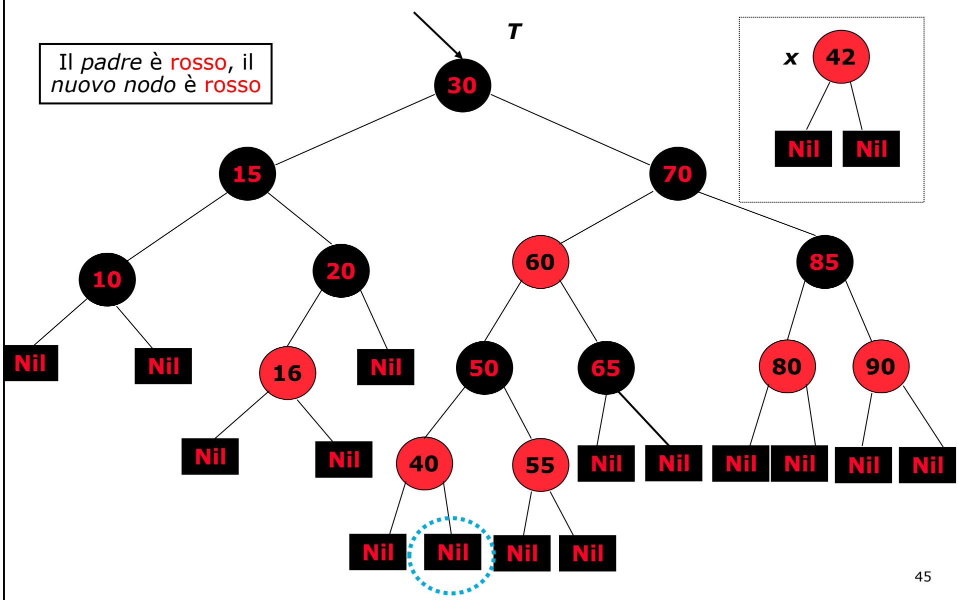
- La complessità di RB-Insert richiede un tempo O(log n) pari all'altezza di un albero rosso-nero con n nodi
- La complessità si RB-Insert-Fixup è in funzione del livello d_z in cui si trova il nodo z

$$T(d_z) \leq \begin{cases} c & \text{Casi } 0,2,3 \\ a + T(d_z - 2) & \text{Caso } 1 \end{cases}$$

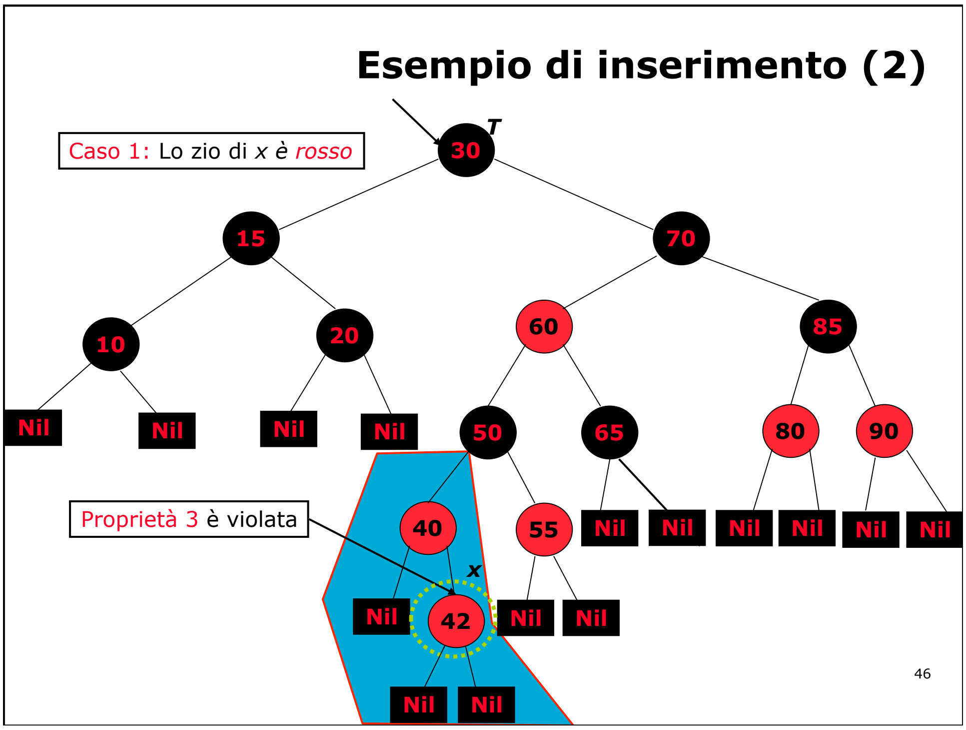
- È immediato concludere che la complessità di RB-Insert-Fixup è O(log n)

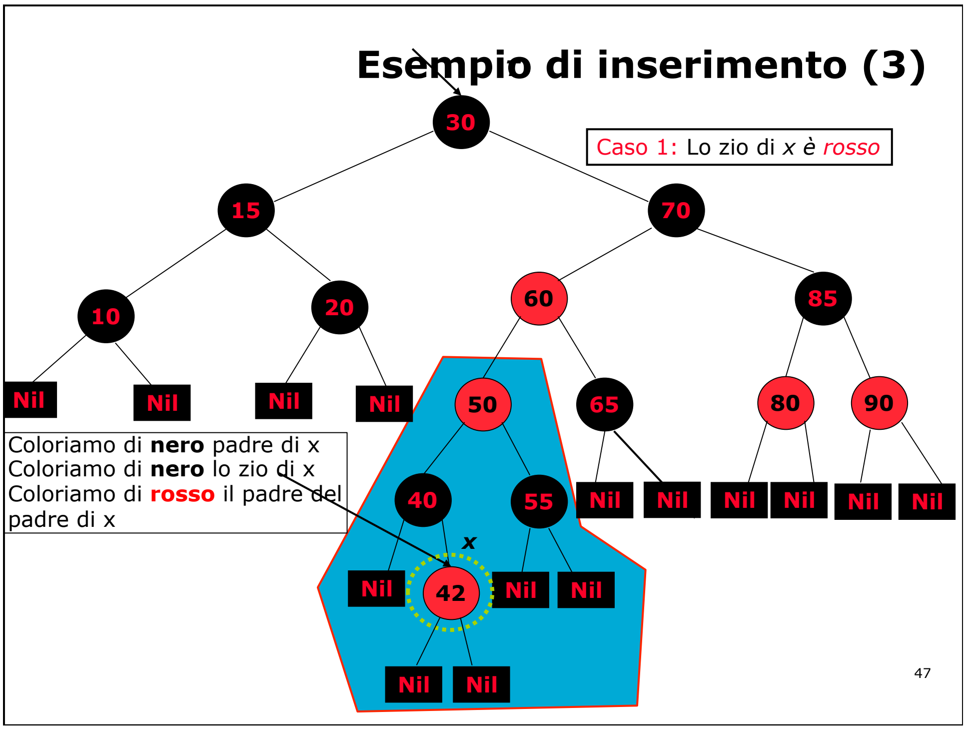
44

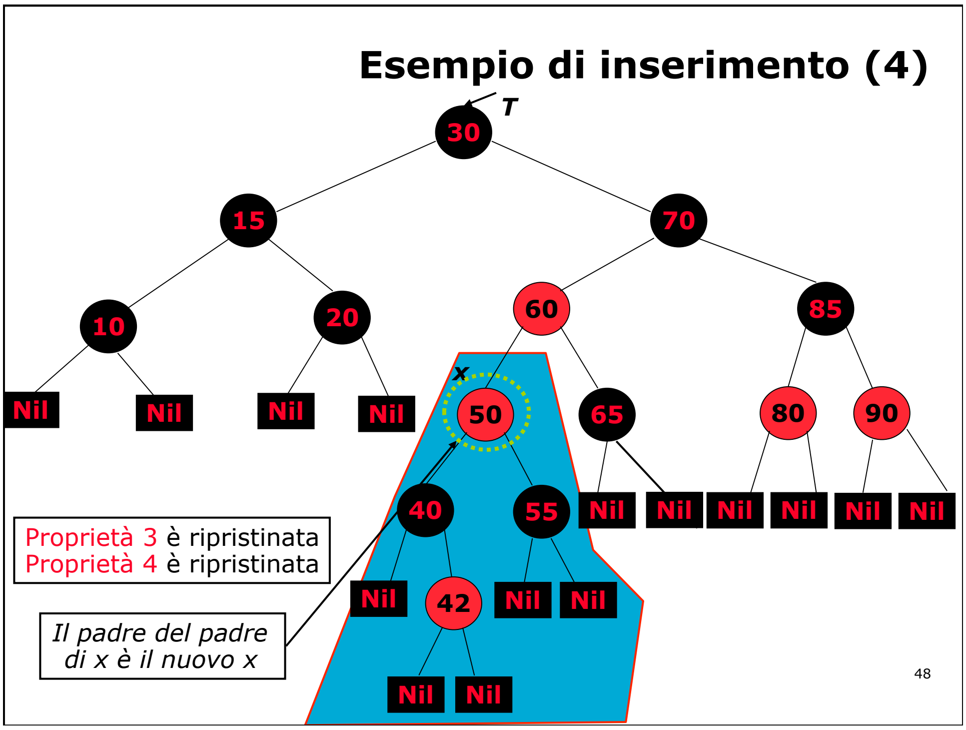
Esempio di inserimento (1)

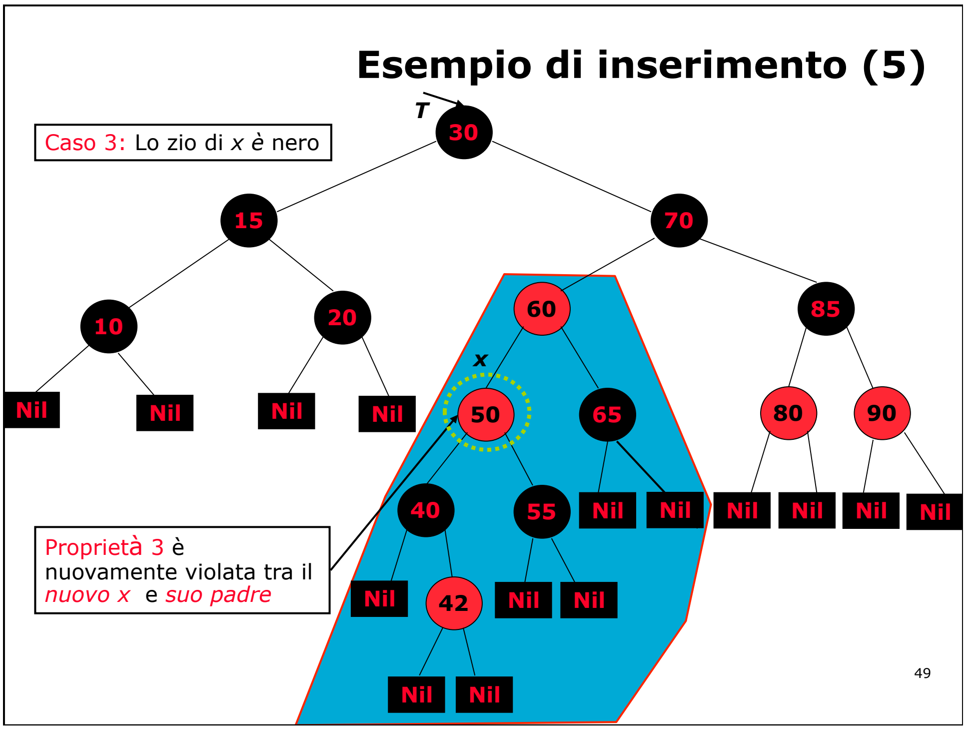


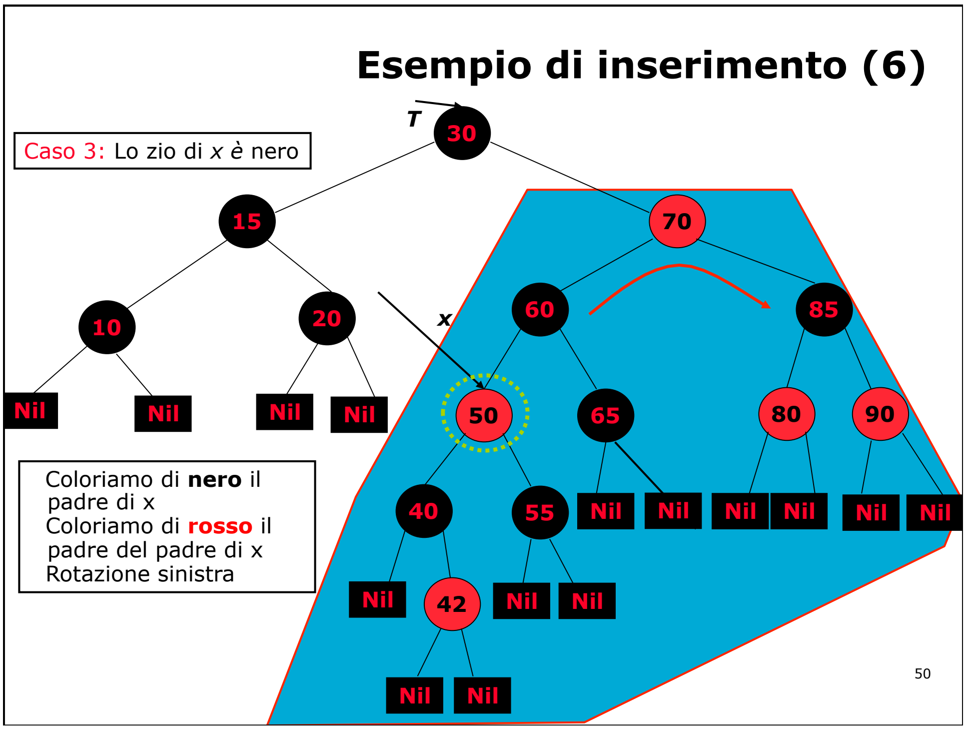
45

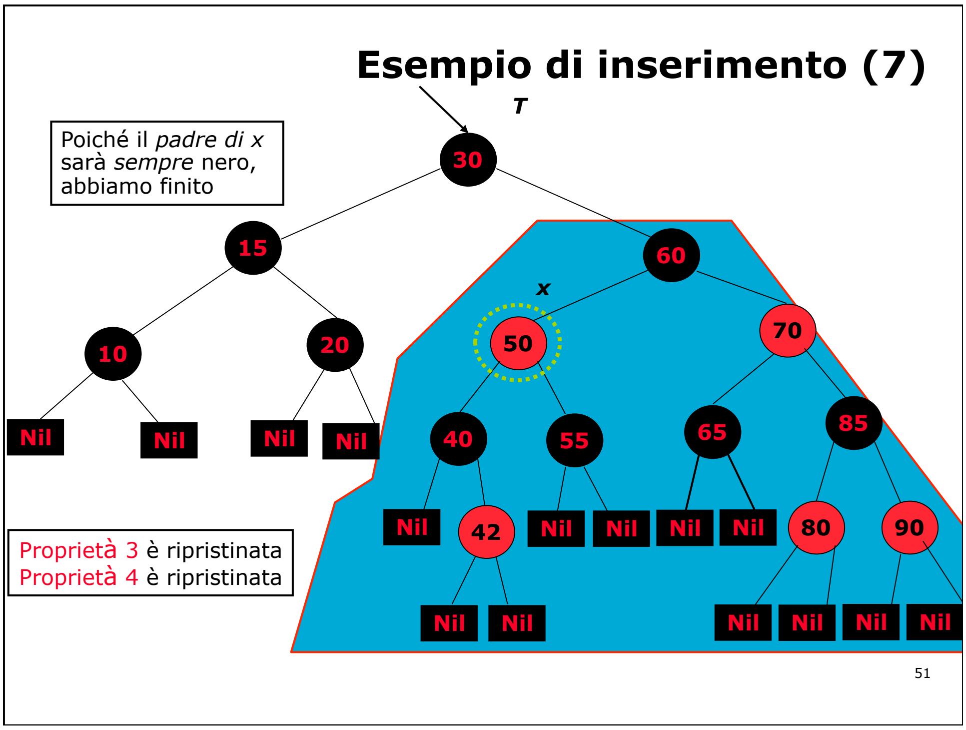

Esempio di inserimento (2)
Caso 1: Lo zio di x è rosso
T
30
15
70
10
20
60
85
Nil
Nil
Nil
Nil
50
65
80
90
Proprietà 3 è violata
40
55
Nil
Nil
Nil
Nil
Nil
Nil
x
Nil
42
Nil
Nil
Nil
Nil
46


Esempio di inserimento (3)
30
Caso 1: Lo zio di x è rosso
15
70
10
20
60
85
Nil
Nil
Nil
Nil
50
65
80
90
Coloriamo di nero padre di x
Coloriamo di nero lo zio di x
Coloriamo di rosso il padre del padre di x
40
55
Nil
Nil
Nil
Nil
Nil
Nil
x
Nil
42
Nil
Nil
Nil
Nil
47

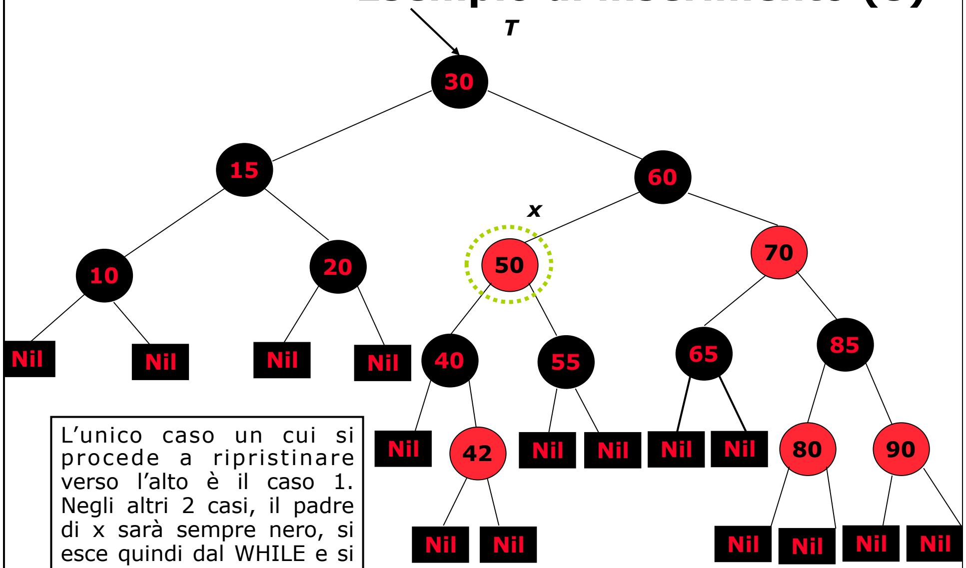

Esempio di inserimento (4)
T
30
15
70
10
20
60
85
Nil
Nil
Nil
Nil
x
50
65
80
90
40
55
Nil
Nil
Nil
Nil
Nil
Nil
Proprietà 3 è ripristinata
Proprietà 4 è ripristinata
Nil
42
Nil
Nil
Il padre del padre di x è il nuovo x
Nil
Nil
48


Esempio di inserimento (5)
T
30
Caso 3: Lo zio di x è nero
15
70
10
20
60
85
Nil
Nil
Nil
Nil
x
50
65
80
90
40
55
Nil
Nil
Nil
Nil
Nil
Nil
Proprietà 3 è nuovamente violata tra il nuovo x e suo padre
Nil
42
Nil
Nil
Nil
Nil
49


Esempio di inserimento (6)
Caso 3: Lo zio di x è nero
T
x
Coloriamo di nero il padre di x
Coloriamo di rosso il padre del padre di x
Rotazione sinistra
50


Esempio di inserimento (7)
T
x
Poiché il padre di x sarà sempre nero, abbiamo finito
Proprietà 3 è ripristinata
Proprietà 4 è ripristinata
51

Esempio di inserimento (8)



L'unico caso un cui si procede a ripristinare verso l'alto è il caso 1. Negli altri 2 casi, il padre di x sarà sempre nero, si esce quindi dal WHILE e si termina

52

Cancellazione di un elemento (1)

- L'algoritmo di cancellazione per alberi rosso-neri è costruito sull'algoritmo di cancellazione per gli alberi binari di ricerca
- Dopo la cancellazione si deve decidere se è necessario ribilanciare o meno
- Se il nodo cancellato è rosso non sono necessarie operazioni di ribilanciamento
  - le altezza nere non sono cambiate
  - non sono stati creati nodi rossi consecutivi
  - la radice resta nera perché se il nodo cancellato è rosso, allora non può essere la radice

53

Cancellazione di un elemento (2)

- Se il nodo cancellato è **nero**:
  - possiamo violare la proprietà 1: la radice può essere un nodo rosso
  - possiamo violare la proprietà 3: se il genitore e uno dei figli del nodo cancellato erano rossi
  - abbiamo violato la proprietà 4: altezza nera cambiata

54

Cancellazione di un elemento (3)

```
RB-Delete(T,z) ▹  z ≠ nil[T]; nodo y da cancellare
   if left[z] = nil[T] or right[z] = nil[T] then
      y ← z
   else
      y ← Tree-Successor(z)
   if left[y] = nil[T] then ▹ elimino y che ha un sottoalbero vuoto
      x ← right[y]
   else
      x ← left[y]
   p[x] ← p[y] ▹ x sottoalbero di y, l’altro è vuoto
   if p[y] = nil[T] then
      root[T] ← x
   else if y = left[p[y]] then
           left[p[y]] ← x
        else
           right[p[y]] ← x
   if y ≠ z then key[z] ← key[y]; copia i dati satellite di y in z
   if color[y] = BLACK then
      RB-Delete-FixUp(T,x)
   return y
```

55

Cancellazione di un elemento (4)

- Le differenze con la procedura Tree-Delete sono:
  - i riferimenti a nil sono sostituiti con i riferimenti alla sentinella nil[T]
  - è stato eliminato il controllo per verificare se x è nil e p[x] ← p[y] viene fatto in modo incondizionato
    - se x è la sentinella nil[T], il suo puntatore p punta al padre del nodo y cancellato
  - se y è nero si fa una chiamata alla RB-Delete-Fixup

56

RB-Delete-Fixup

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
          then w ← right[p[x]]
               if color[w] = RED
                  then color[w] ← BLACK                        ▷ Case 1
                       color[p[x]] ← RED                       ▷ Case 1
                       LEFT-ROTATE(T, p[x])                    ▷ Case 1
                       w ← right[p[x]]                         ▷ Case 1
               if color[left[w]] = BLACK and color[right[w]] = BLACK
                  then color[w] ← RED                          ▷ Case 2
                       x ← p[x]                                ▷ Case 2
                  else if color[right[w]] = BLACK
                          then color[left[w]] ← BLACK          ▷ Case 3
                               color[w] ← RED                  ▷ Case 3
                               RIGHT-ROTATE(T, w)              ▷ Case 3
                               w ← right[p[x]]                 ▷ Case 3
                       color[w] ← color[p[x]]                  ▷ Case 4
                       color[p[x]] ← BLACK                     ▷ Case 4
                       color[right[w]] ← BLACK                 ▷ Case 4
                       LEFT-ROTATE(T, p[x])                    ▷ Case 4
                       x ← root[T]                             ▷ Case 4
          else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

57

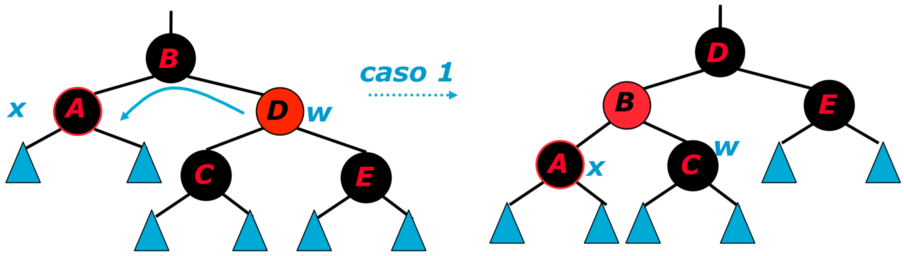
RB-Delete-Fixup

- Riceve come input un nodo x che può essere:
  - il nodo che era l'unico figlio del nodo cancellato y prima della sua rimozione
  - la sentinella nil[T] se y non aveva figli
- Ripristina la proprietà rosso-nero con cambi di colore e rotazioni
- Deve gestire quattro casi diversi (e quattro casi simmetrici)

58

Delete: Caso 1

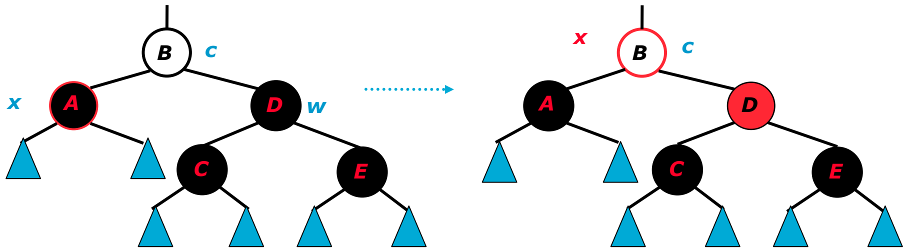
Fratello rosso, padre nero



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 la proprietà 3 né la 4 e ci riduciamo ad uno degli altri casi

59

Delete: 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

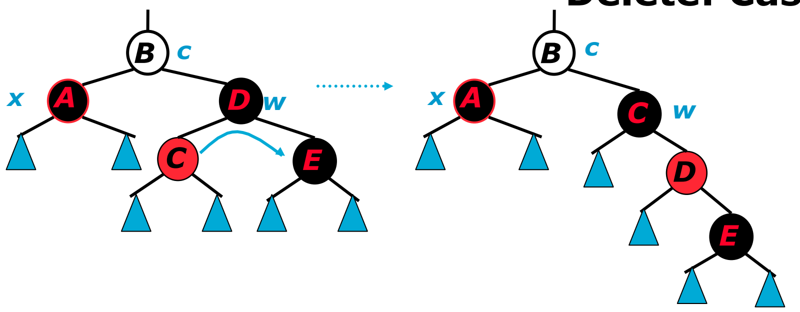
Spostiamo il *nero in più* da x al nuovo x (il padre) e togliamo il nero da w per rispettare proprietà 4

Si ripristina se è il caso (se il padre era nero) il bilanciamento, altrimenti si termina

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) e può essere semplicemente colorato di nero.

60

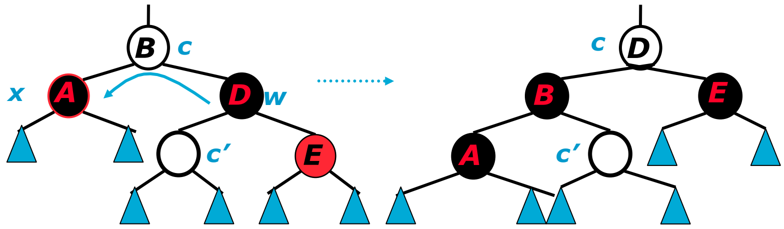
Delete: Caso 3



- Il fratello w di x è *nero*
- w ha il figlio sinistro rosso, il figlio destro nero
- cambiamo il colore del sinistro di w e cambiamo quello di w
- ruotiamo w col suo figlio sinistro

Non violiamo alcuna proprietà (3 e 4) e il nuovo fratello di x è ora nero con figlio destro rosso, quindi ci portiamo nel caso 4

Delete: Caso 4



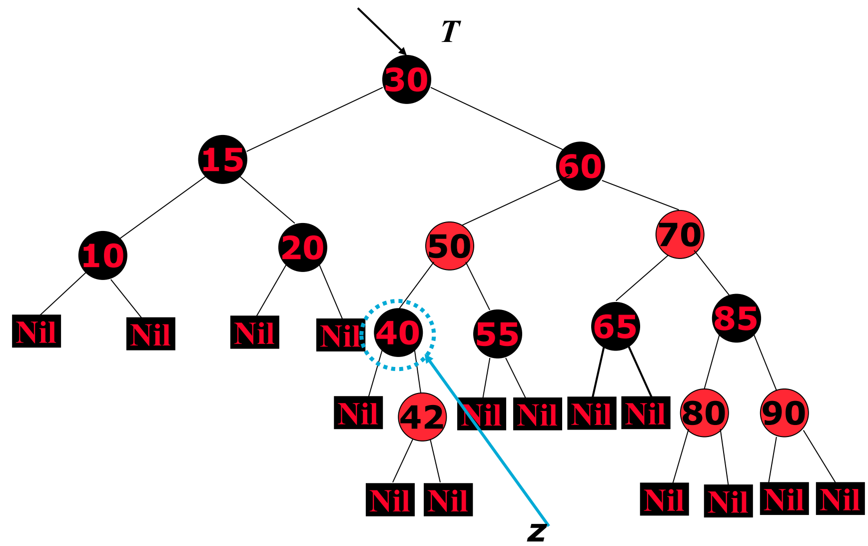
- Il *fratello w* di *x* è *nero*
- *w* ha in questo caso solo il figlio destro *rosso*
- cambiamo i colori *opportunamente* e con una *rotazione del padre di x con w* si *elimina il nero in più su x*

Non violiamo alcuna proprietà (3 e 4) e abbiamo *finito!*

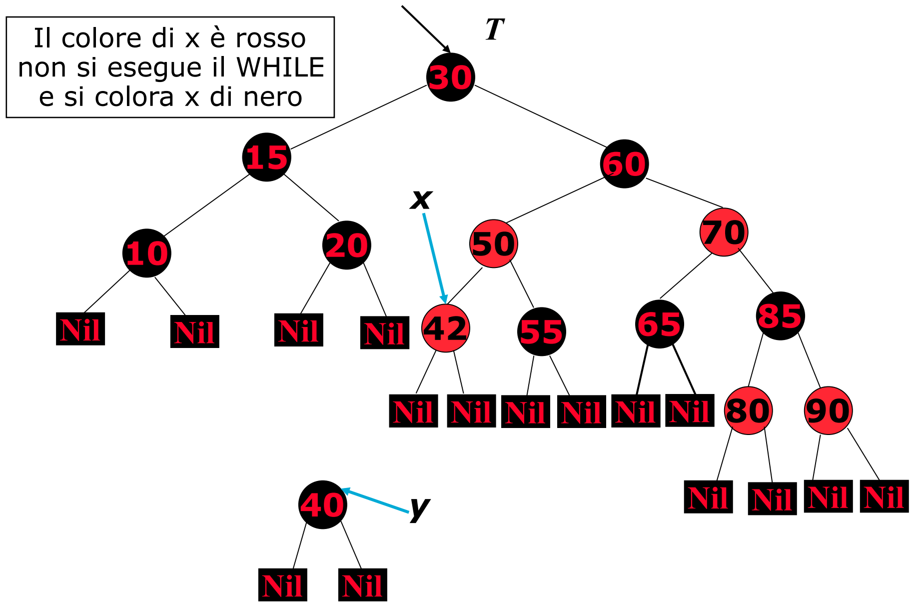
Delete: Simmetria dei casi

- Abbiamo visto i 4 casi possibili quando il nodo x che sostituisce y (cancellato) è un figlio sinistro
- Esistono anche i 4 casi simmetrici (con destro e sinistro scambiati) quando x è figlio destro

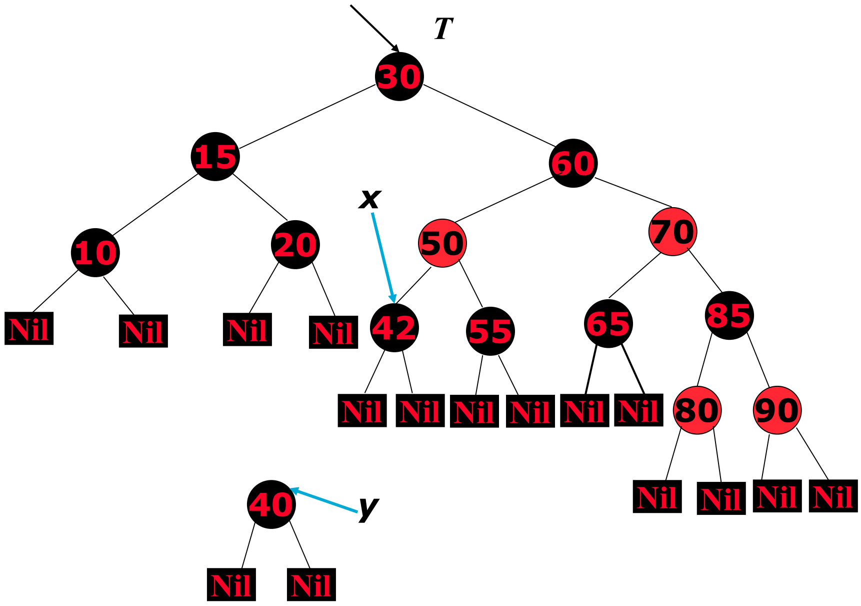
Delete: Primo esempio (1)



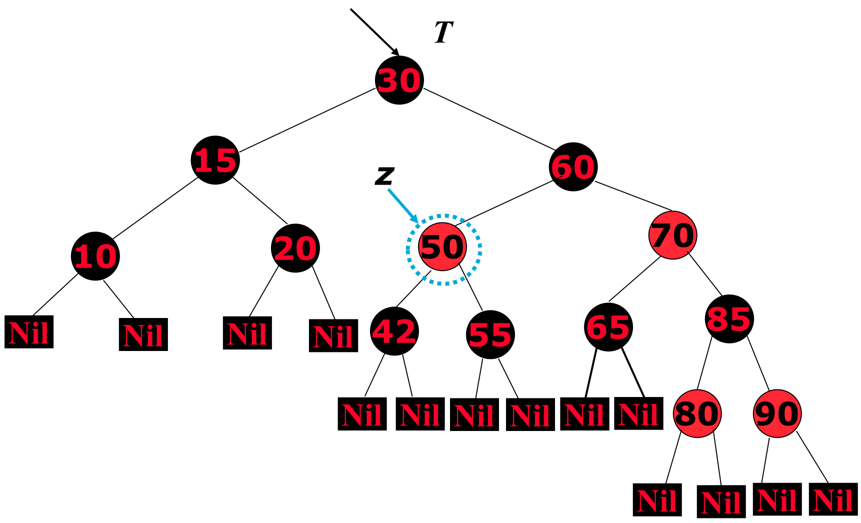
Delete: Primo esempio (2)

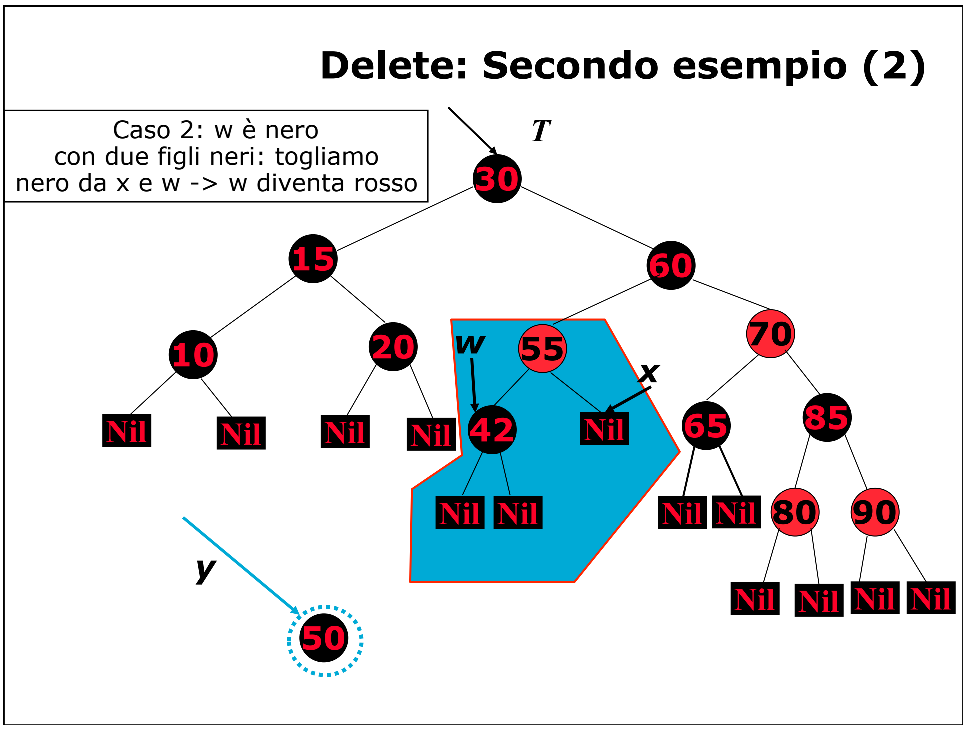


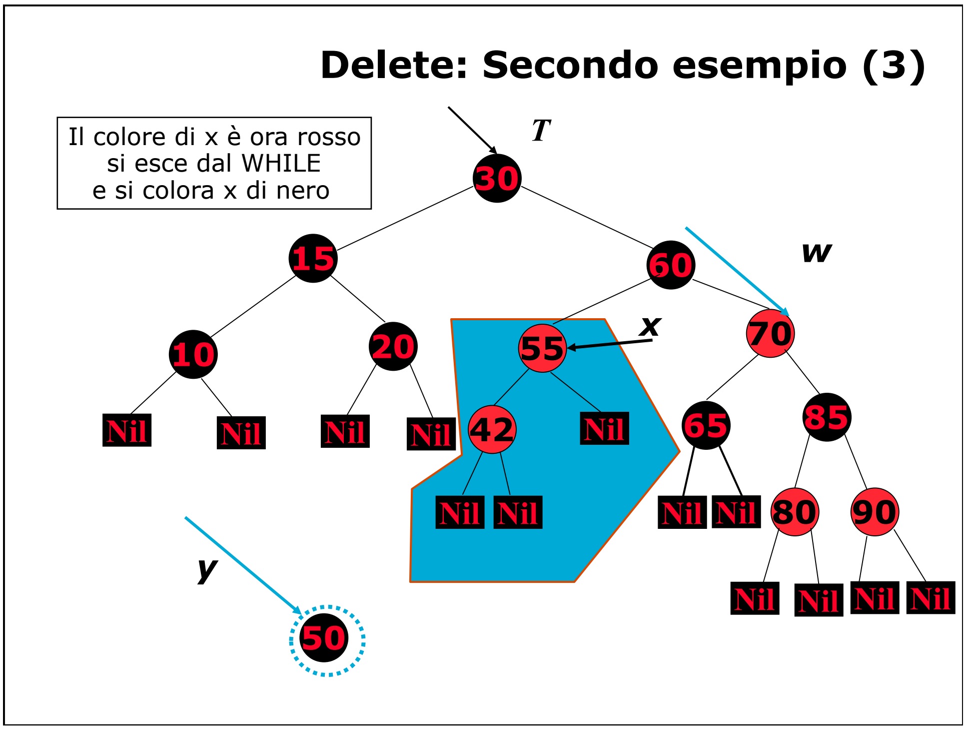
Delete: Primo esempio (3)

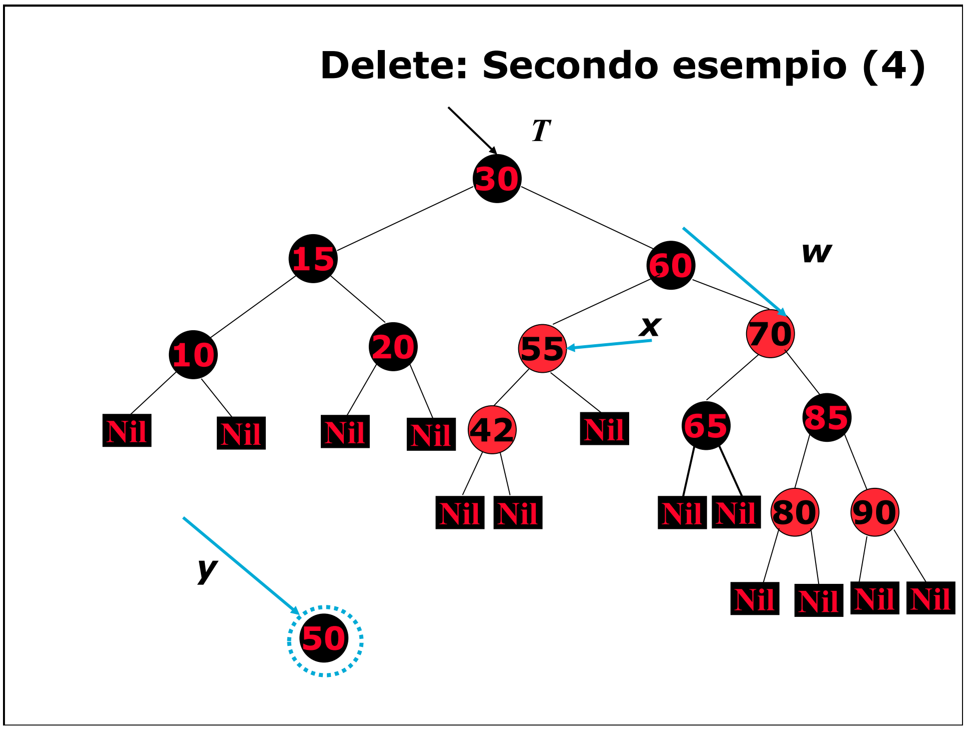


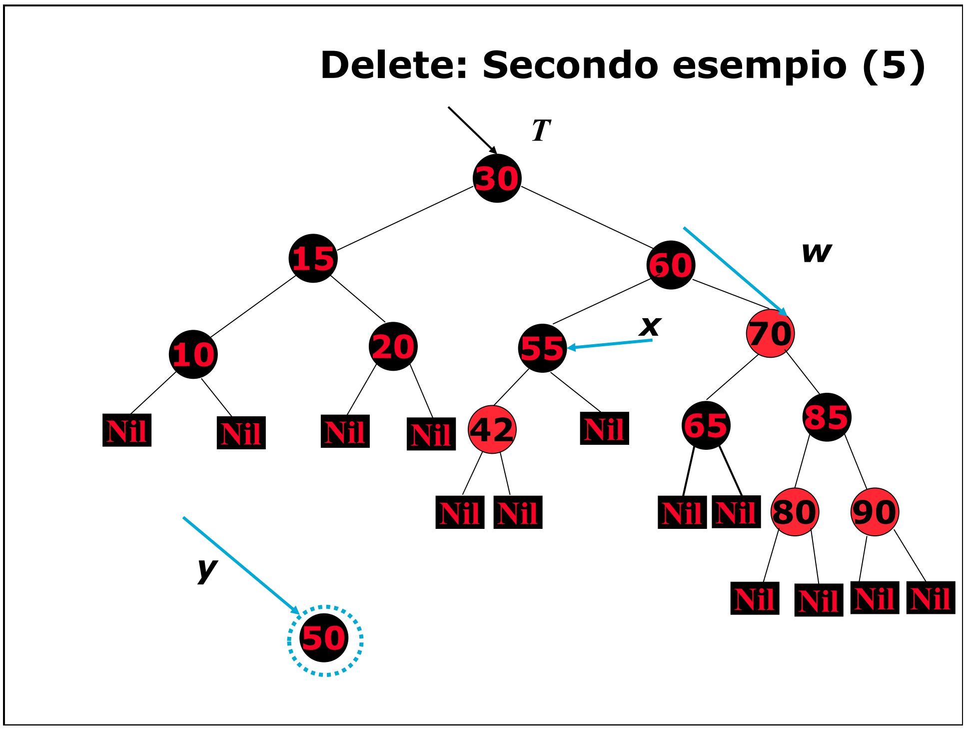
Delete: Secondo esempio (1)



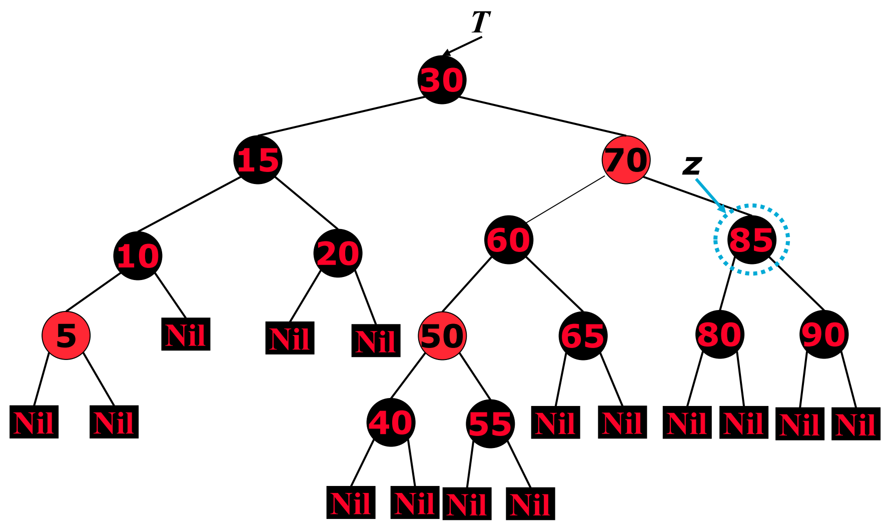

Delete: Secondo esempio (2)
Caso 2: w è nero
con due figli neri: togliamo
nero da x e w -> w diventa rosso
T
30
15
60
10
20
w
55
x
70
42
65
85
80
90
y
50
Nil


Delete: Secondo esempio (3)
Il colore di x è ora rosso
si esce dal WHILE
e si colora x di nero
T
30
15
w
60
x
10
20
55
70
42
65
85
80
90
y
50
Nil

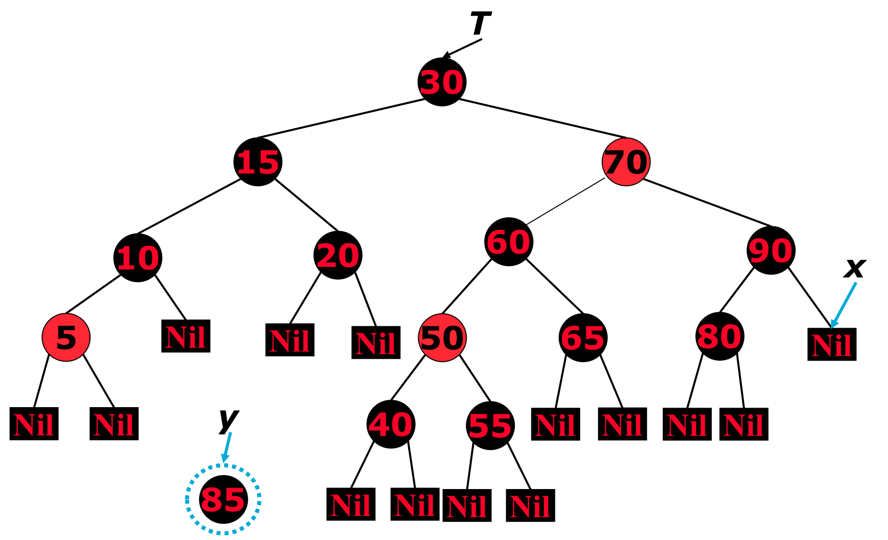

Delete: Secondo esempio (4)
T
30
15
60
w
10
20
55
x
70
Nil
Nil
Nil
Nil
42
Nil
65
85
Nil
Nil
Nil
Nil
80
90
y
Nil
Nil
Nil
Nil
50


Delete: Secondo esempio (5)
T
30
15
60
w
10
20
55
x
70
Nil
Nil
Nil
Nil
42
Nil
65
85
Nil
Nil
Nil
Nil
80
90
y
Nil
Nil
Nil
Nil
50

Delete: Terzo esempio (1)

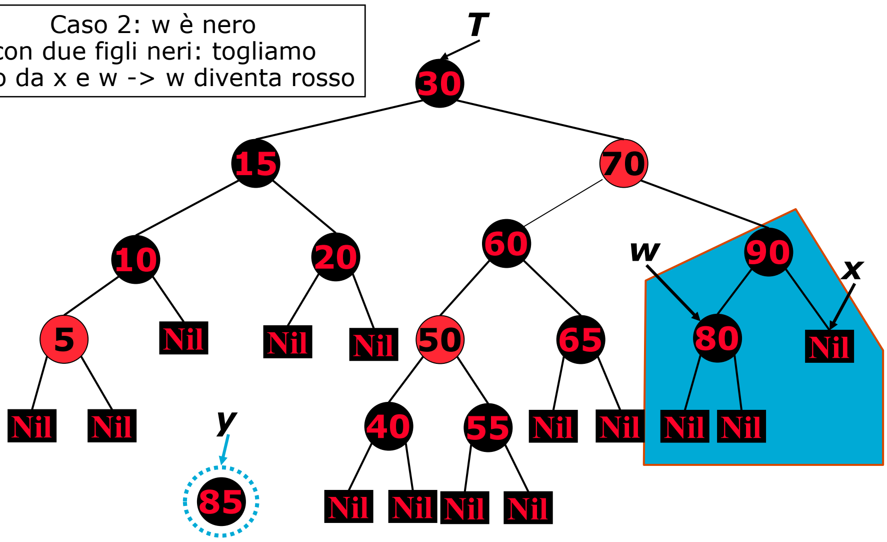


Delete: Terzo esempio (2)

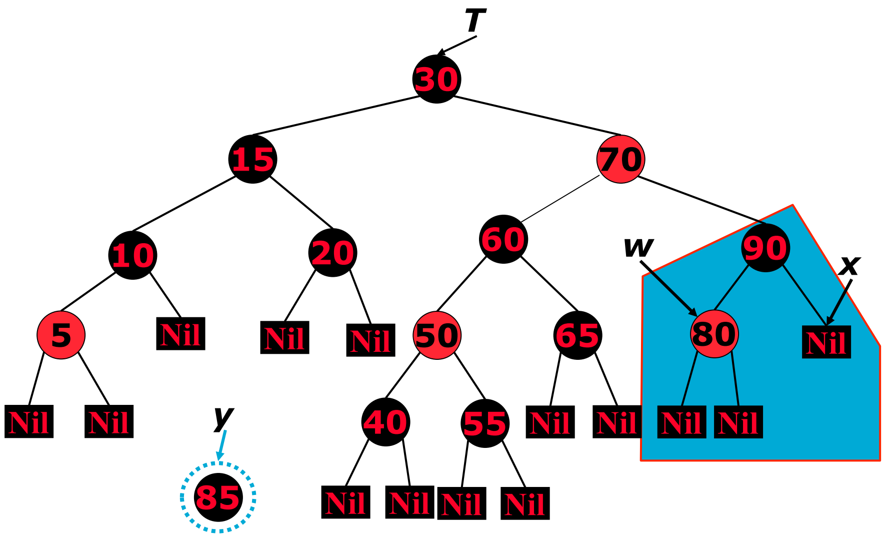


Delete: Terzo esempio (3)

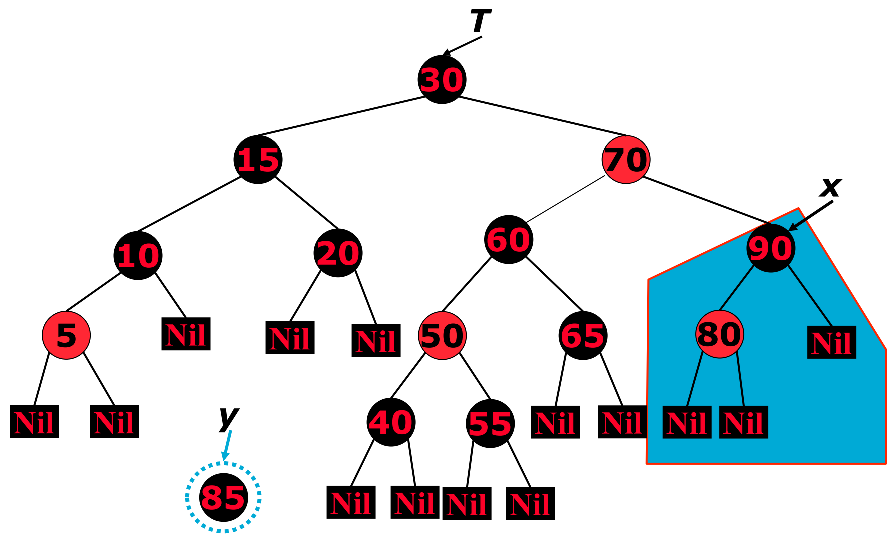
Caso 2: w è nero
con due figli neri: togliamo
nero da x e w -> w diventa rosso



Delete: Terzo esempio (4)

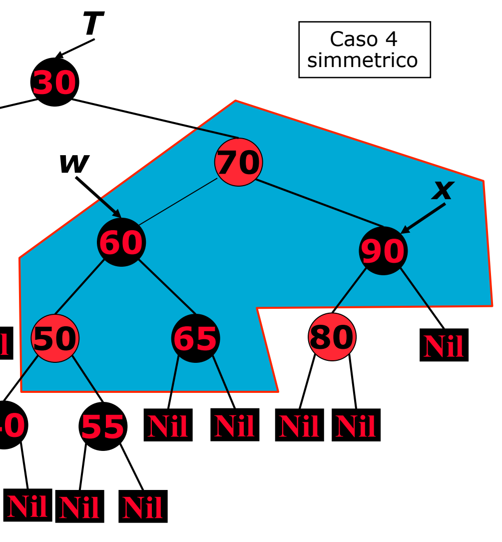


Delete: Terzo esempio (5)



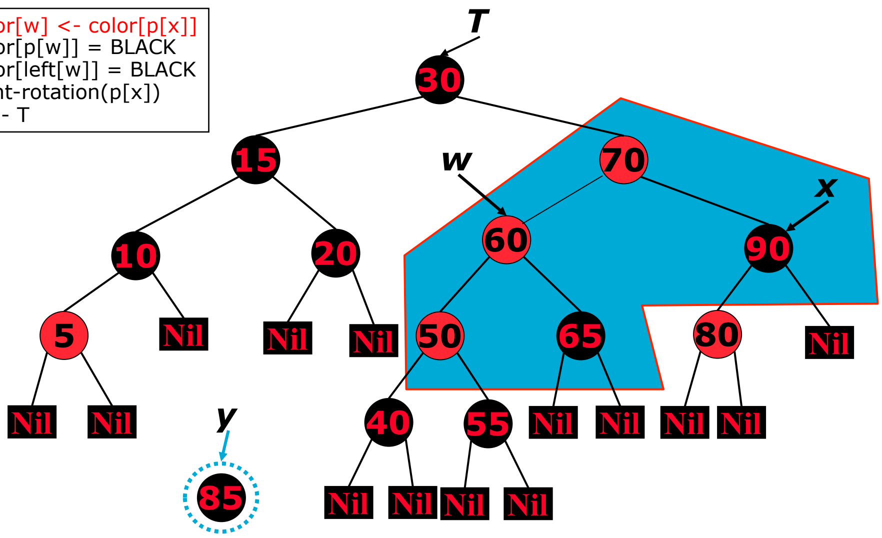
Delete: Terzo esempio (6)

```
color[w] <- color[p[x]]
color[p[w]] = BLACK
color[left[w]] = BLACK
right-rotation(p[x])
x <- T
```



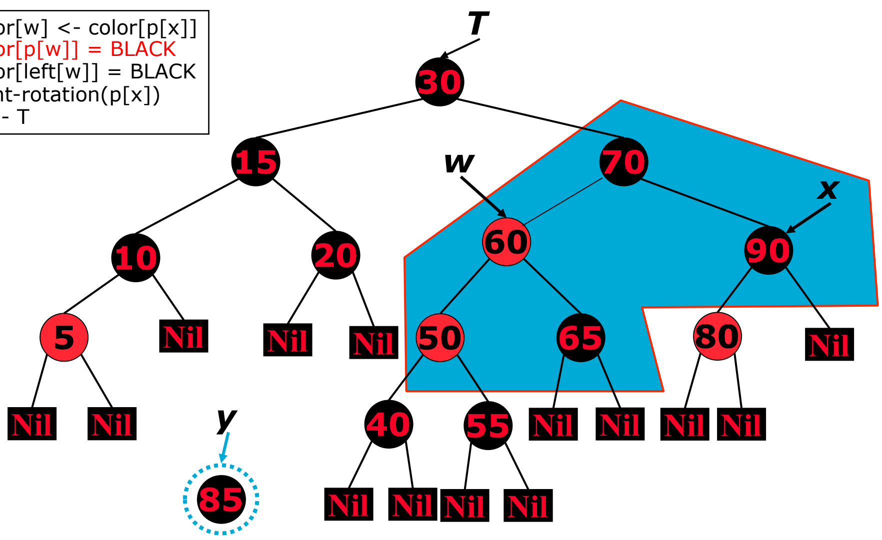
Delete: Terzo esempio (7)

color[w] <- color[p[x]]
color[p[w]] = BLACK
color[left[w]] = BLACK
right-rotation(p[x])
x <- T



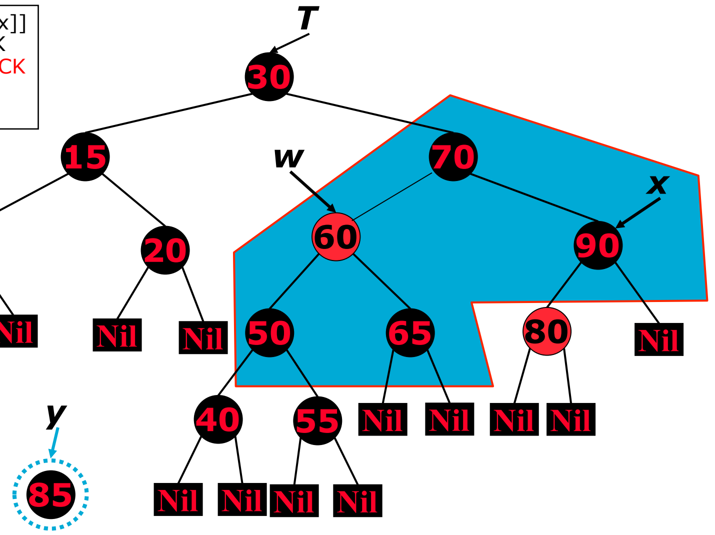
Delete: Terzo esempio (8)

color[w] <- color[p[x]]
color[p[w]] = BLACK
color[left[w]] = BLACK
right-rotation(p[x])
x <- T



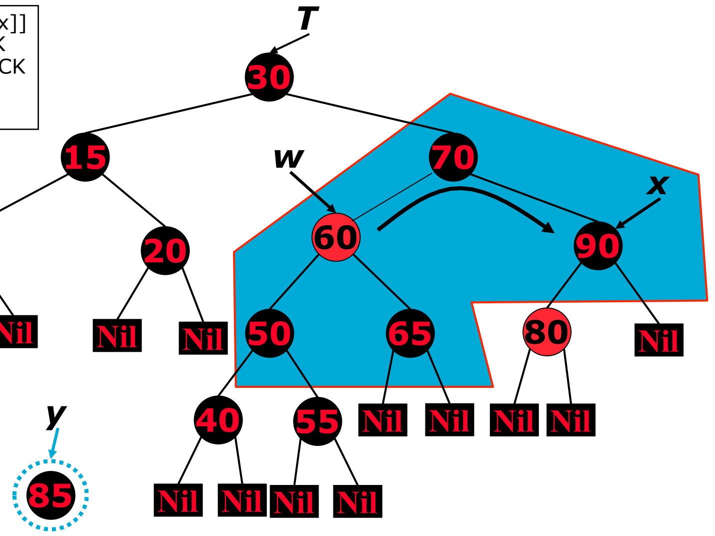
Delete: Terzo esempio (9)

color[w] <- color[p[x]]
color[p[w]] = BLACK
color[left[w]] = BLACK
right-rotation(p[x])
x <- T

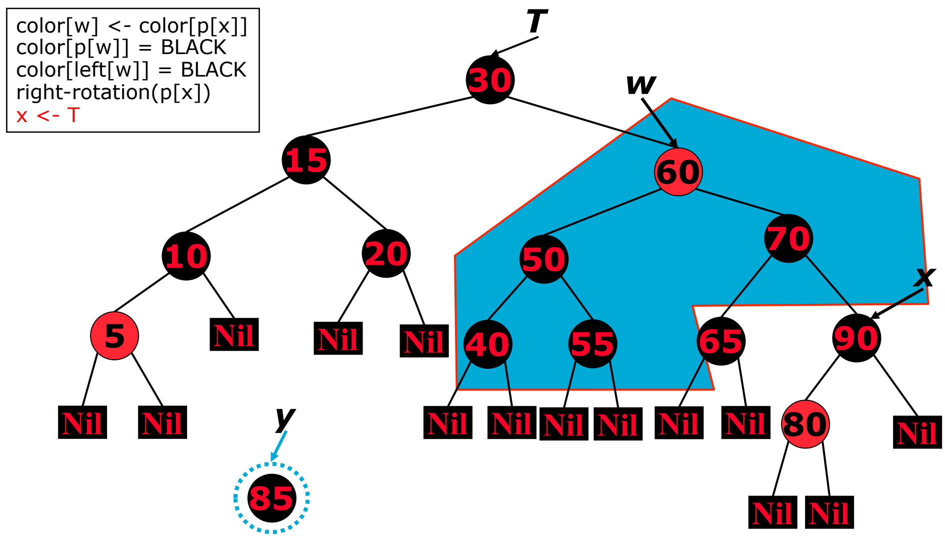


Delete: Terzo esempio (10)

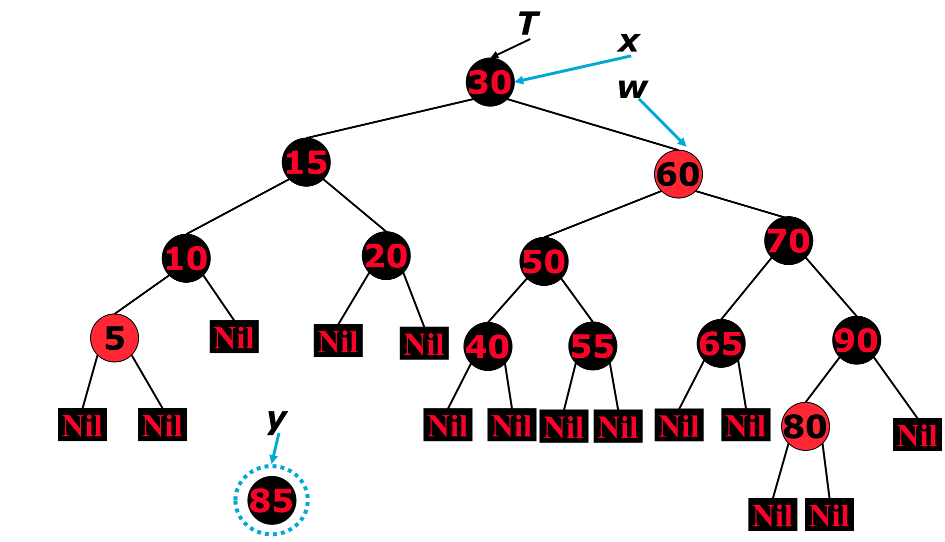
color[w] <- color[p[x]]
color[p[w]] = BLACK
color[left[w]] = BLACK
right-rotation(p[x])
x <- T



Delete: Terzo esempio (11)



Delete: Terzo esempio (12)



Cancellazione

L'operazione di cancellazione è concettualmente complicata!

Ma efficiente:

- il caso 4 è risolutivo e applica una 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

per iterazione del ciclo while

Alberi 2-3 (1)

- Introduciamo prima la nozione di nodo 2-3
- Un nodo 2-3 è un insieme ordinato contenente due chiavi k_1 e k_2 (in ordine crescente) e tre puntatori p_0, p_1 e p_2 tali che:
  - p_0 punta al sottoalbero contenente chiavi k tali che $k < k_1$
  - p_1 punta al sottoalbero contenente chiavi k tali che $k_1 < k < k_2$
  - p_2 punta al sottoalbero contenente chiavi k tali che $k > k_2$

85

Alberi 2-3 (2)

Un albero 2-3 è un insieme di nodi 2-3 che soddisfa la definizione di albero e che verificano le seguenti proprietà:

- le foglie sono tutte sullo stesso livello
- ogni nodo ha al più tre figli

86

Ricerca in un albero 2-3

- La ricerca di una chiave k avviene confrontando la chiave da ricercare con il contenuto di un nodo a partire dal nodo radice
  - se $k < k_1$ la ricerca prosegue nel sottoalbero indicato da p_0
  - se $k = k_1$ la ricerca termina con successo
  - se nel nodo c'è una sola chiave la ricerca prosegue nel sottoalbero indicato da p_1
  - se nel nodo ci sono due chiavi è necessario eseguire ulteriori confronti:
    - se $k_1<k< k_2$ la ricerca prosegue nel sottoalbero indicato da p_1
    - se $k = k_2$ la ricerca termina con successo
    - se $k > k_2$, la ricerca prosegue nel sottoalbero indicato da p_2

87

Inserimento in un albero 2-3

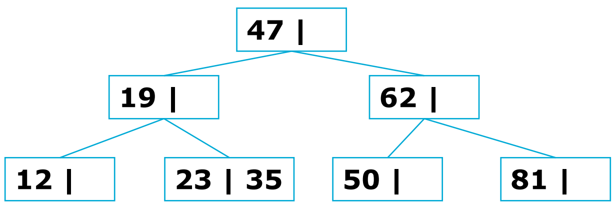
- L'inserimento di un nuovo elemento avviene dopo aver effettuato una ricerca senza successo che si arresta su un nodo v dal quale dovrebbe proseguire tramite un puntatore vuoto
- Se v contiene una sola chiave, la nuova chiave viene inserita nel nodo stesso, ordinatamente con quella preesistente
- Se v contiene già due chiavi, si considera l'insieme ordinato (a_1,a_2,a_3) costituito dalle due chiavi del nodo e dalla nuova chiave
  - v viene suddiviso (split) in due nodi contenenti rispettivamente a_1 e a_3, mentre a_2 viene inserito (ricorsivamente) nel padre del nodo d'arresto assieme a due puntatori ai nodi contenenti a_1 e a_3
  - questa operazione può ripetersi a ritroso fino alla radice e, nel caso in cui sia necessario suddividere anche la radice, l'altezza dell'albero viene incrementata di uno

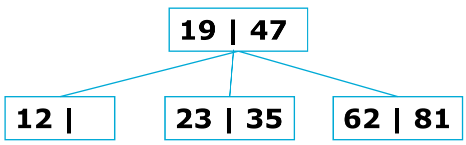
88

Cancellazione in un albero 2-3

Può essere eseguita ri-compattando (operazione di merge) eventualmente i nodi interessati

Cancello il valore 50





89

Complessità operazioni su alberi 2-3

Anche in questo caso tutte le operazioni hanno costo O(log n) poiché l'altezza dell'albero è O(log n) e le operazioni di merge e split richiedono tempo costante

90