

Algoritmi e Strutture Dati

Alberi binari di ricerca

Alberto Montresor

Università di Trento

2018/10/25

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Sommario

1 Alberi binari di ricerca

- Ricerca
- Minimo-massimo
- Successore-predecessore
- Inserimento
- Cancellazione
- Costo computazionale

2 Alberi binari di ricerca bilanciati

- Alberi Red-Black – Definizioni
- Alberi Red-Black – Esempi
- Alberi Red-Black – Inserimento
- Alberi Red-Black – Cancellazione

Introduzione

Dizionario

E' un insieme dinamico che implementa le seguenti funzionalità:

- ITEM `lookup(ITEM k)`
- `insert(ITEM k, ITEM v)`
- `remove(ITEM k)`

Possibili implementazioni

Struttura dati	lookup	insert	remove
Vettore ordinato	$O(\log n)$	$O(n)$	$O(n)$
Vettore non ordinato	$O(n)$	$O(1)^*$	$O(1)^*$
Lista non ordinata	$O(n)$	$O(1)^*$	$O(1)^*$

* Assumendo che l'elemento sia già stato trovato, altrimenti $O(n)$

Alberi binari di ricerca (ABR)

Idea ispiratrice

Portare l'idea di ricerca binaria negli alberi

Memorizzazione

- Le **associazioni chiave-valore** vengono memorizzate in un albero binario
- Ogni nodo u contiene una coppia $(u.key, u.value)$
- Le chiavi devono appartenere ad un insieme **totalmente ordinato**

Nodo albero

TREE

TREE *parent*

TREE *left*

TREE *right*

ITEM *key*

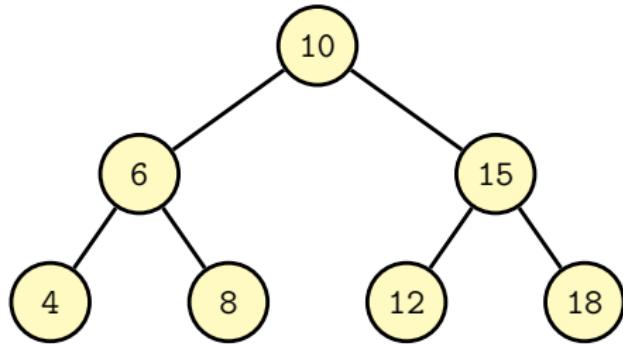
ITEM *value*

Alberi binari di ricerca (ABR)

Proprietà

- ① Le chiavi contenute nei nodi del **sottoalbero sinistro** di u sono **minori** di $u.key$
- ② Le chiavi contenute nei nodi del **sottoalbero destro** di u sono **maggiori** di $u.key$

Le proprietà 1. e 2. permettono di realizzare un algoritmo di ricerca dicotomica



Alberi binari di ricerca – Specifica

Getters

- ITEM `key()`
- ITEM `value()`
- TREE `left()`
- TREE `right()`
- TREE `parent()`

Dizionario

- ITEM `lookup(ITEM k)`
- insert(ITEM k , ITEM v)
- remove(ITEM k)

Ordinamento

- TREE `successorNode(TREE t)`
- TREE `predecessorNode(TREE t)`
- TREE `min()`
- TREE `max()`

Alberi binari di ricerca – Funzioni interne

- TREE `lookupNode(TREE T , ITEM k)`
- TREE `insertNode(TREE T , ITEM k , ITEM v)`
- TREE `removeNode(TREE T , ITEM k)`

DICTIONARY

TREE $tree$

Dictionary()

└ $tree = \text{nil}$

Ricerca – `lookupNode()`

ITEM `lookupNode(TREE T, ITEM k)`

- Restituisce il nodo dell'albero T che contiene la chiave k , se presente
- Restituisce **nil** se non presente

Implementazione dizionario

ITEM `lookup(ITEM k)`

`TREE t = lookupNode(tree, k)`

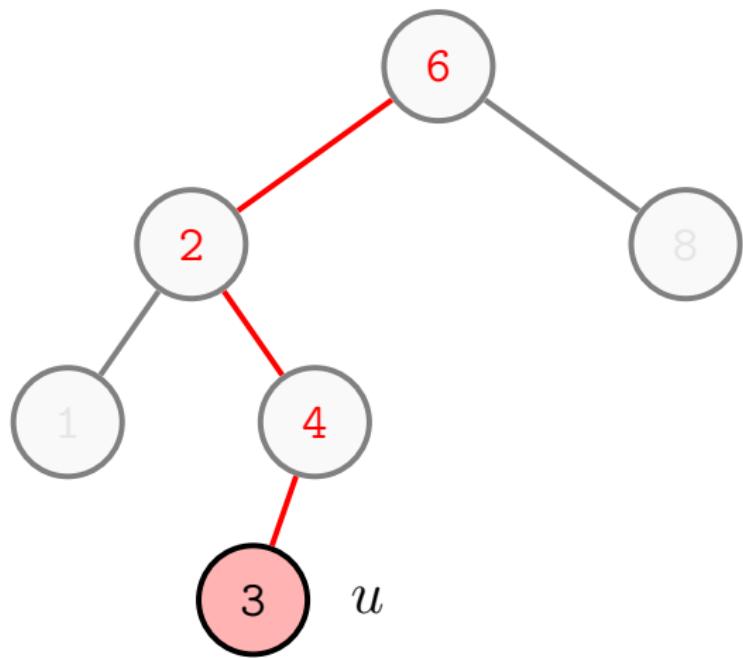
if $t \neq \text{nil}$ **then**

 | **return** $t.\text{value}()$

else

 | **return** **nil**

Ricerca – esempio



Valore cercato: 3

- $u = \textcircled{6}$
- $3 < 6$; $u = \textcircled{2}$ (Sinistra)
- $3 > 2$; $u = \textcircled{4}$ (Destra)
- $3 < 4$; $u = \textcircled{3}$ (Sinistra)
- $3 = 3$; Trovato

Ricerca – Implementazione

Iterativa

```
TREE lookupNode(TREE T, ITEM k)
```

```
TREE u = T
```

```
while u ≠ nil and u.key ≠ k do
```

```
    if k < u.key then
```

```
        | u = u.left
```

% Sotto-albero di sinistra

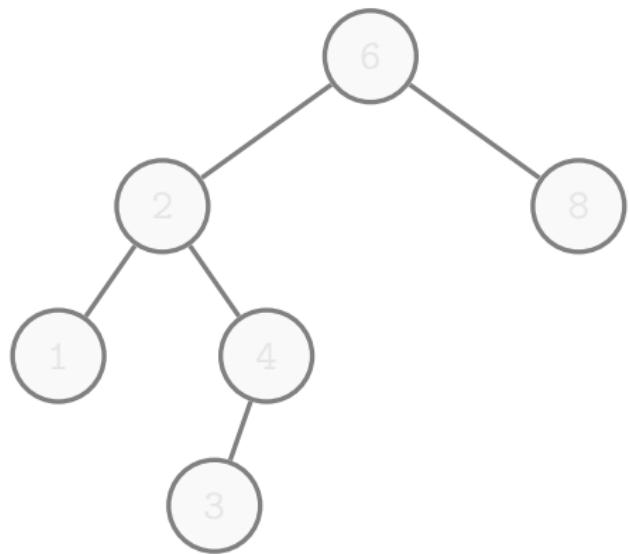
```
    else
```

```
        | u = u.right
```

% Sotto-albero di destra

```
return u
```

Ricerca – esempio



TREE `lookupNode(TREE T, ITEM k)`

TREE $u = T$

while $T \neq \text{nil}$ and $T.\text{key} \neq k$ **do**
 $u = \text{iif}(k < u.\text{key}, u.\text{left}, u.\text{right})$

return u

Valore cercato: 5

- $u = (6)$
- $5 < 6$; $u = (2)$
- $5 > 2$; $u = (4)$
- $5 > 4$; $u = \text{nil}$ (Destra)
- **return nil** (Non trovato)

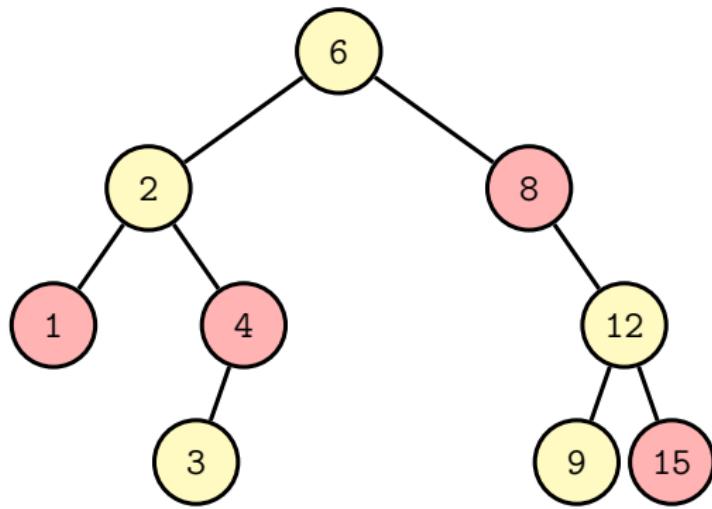
Ricerca – Implementazione

Ricorsiva

```
TREE lookupNode(TREE T, ITEM k)
```

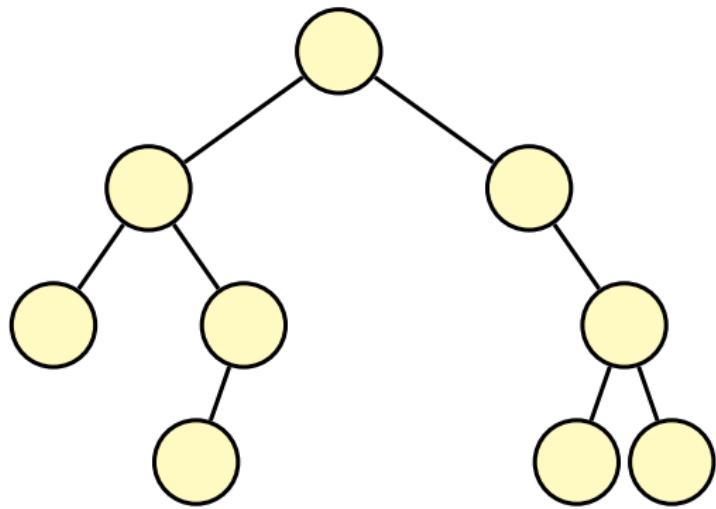
```
if T == nil or T.key == k then
    return T
else
    return lookupNode(iif(k < T.key, T.left, T.right), k)
```

Minimo-massimo



- min albero radice (6) ?
1
- max albero radice (6) ?
15
- max albero radice (2) ?
4
- min albero radice (8) ?
8

Minimo-massimo



TREE $\min(\text{TREE } T)$

TREE $u = T$

while $u.\text{left} \neq \text{nil}$ **do**
 $u = u.\text{left}$

return u

TREE $\max(\text{TREE } T)$

TREE $u = T$

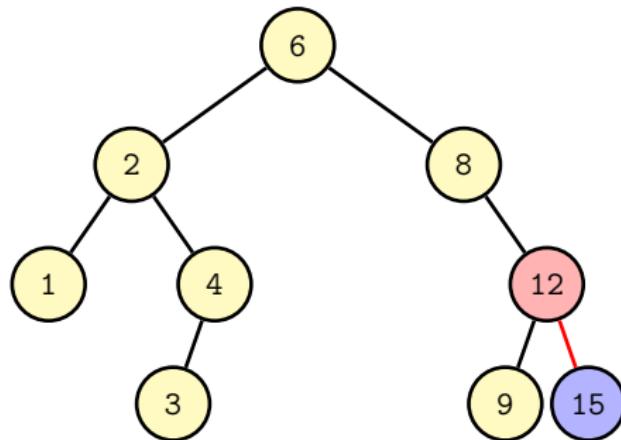
while $u.\text{right} \neq \text{nil}$ **do**
 $u = u.\text{right}$

return u

Successore-predecessore – Esempio 1

Definizione

Il **successore** di un nodo u è il più piccolo nodo maggiore di u

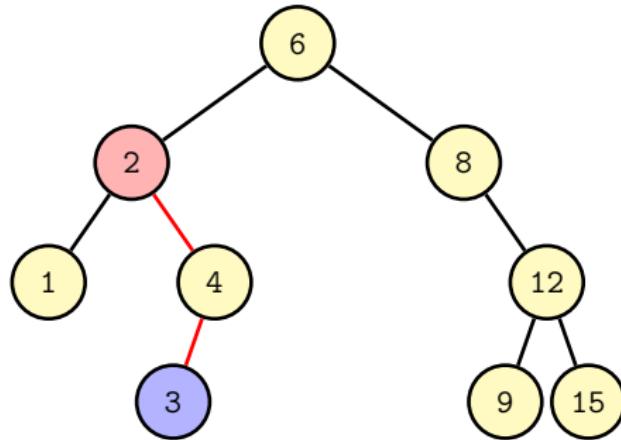


Successore di 12 ? 15

Successore-predecessore – Esempio 2

Definizione

Il **successore** di un nodo u è il più piccolo nodo maggiore di u

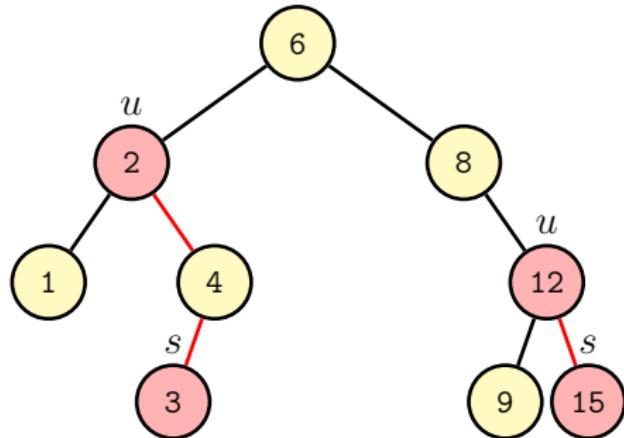


Successore di **2** ? **3**

Successore-predecessore – Caso 1

Definizione

Il **successore** di un nodo u è il più piccolo nodo maggiore di u



Successore di u ?

Caso 1

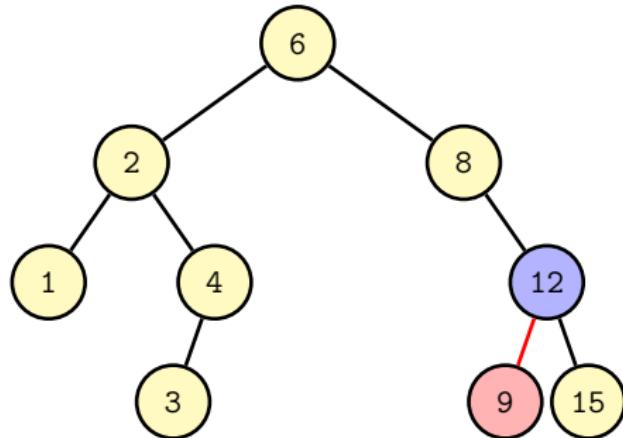
u ha figlio destro

Il successore v è il **minimo del sottoalbero destro** di u

Successore-predecessore – Esempio 3

Definizione

Il **successore** di un nodo u è il più piccolo nodo maggiore di u

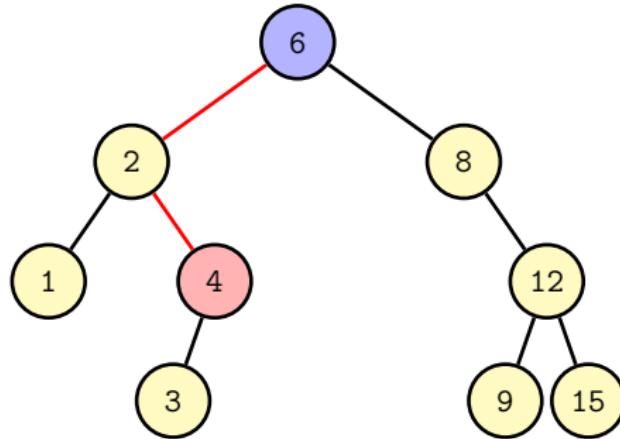


Successore di 9 ? 12

Successore-predecessore – Esempio 4

Definizione

Il **successore** di un nodo u è il più piccolo nodo maggiore di u

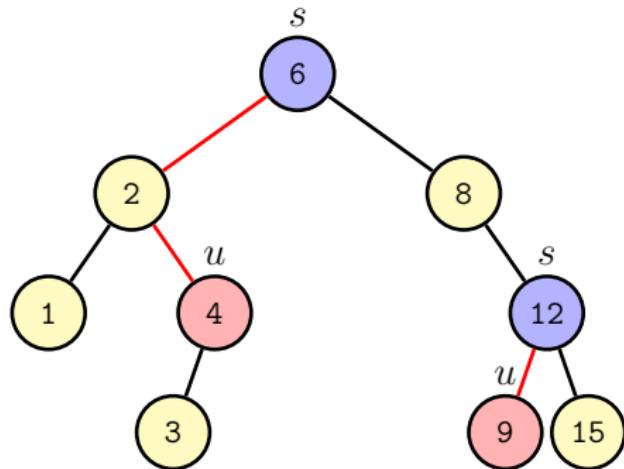


Successore di 4 ? 6

Successore-predecessore – Caso 2

Definizione

Il **successore** di un nodo u è il più piccolo nodo maggiore di u



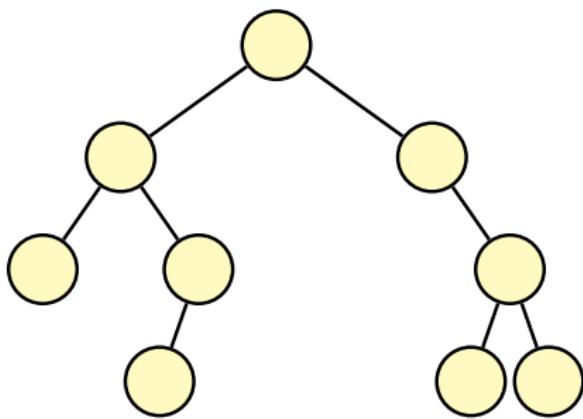
Successore di u ?

Caso 2

u non ha figlio destro

Risalendo attraverso i padri, il successore è il **primo avo** v tale per cui u sta nel **sottoalbero sinistro** di v

Successore-predecessore – Implementazione



TREE successorNode(TREE t)

```

if  $t == \text{nil}$  then
    return  $t$ 
if  $t.\text{right} \neq \text{nil}$  then      % Caso 1
    return min( $t.\text{right}$ )
else
    TREE  $p = t.\text{parent}$ 
    while  $p \neq \text{nil}$  and  $t == p.\text{right}$ 
        do
             $t = p$ 
             $p = p.\text{parent}$ 
    return  $p$ 
  
```

Successore-predecessore – Implementazione

TREE predecessorNode(TREE t)

```

if  $t == \text{nil}$  then
    return  $t$ 
if  $t.\text{left} \neq \text{nil}$  then          % Caso 1
    return max( $t.\text{left}$ )
else                                % Caso 2
    TREE  $p = t.\text{parent}$ 
    while  $p \neq \text{nil}$  and  $t == p.\text{left}$  do
         $t = p$ 
         $p = p.\text{parent}$ 
    return  $p$ 

```

TREE successorNode(TREE t)

```

if  $t == \text{nil}$  then
    return  $t$ 
if  $t.\text{right} \neq \text{nil}$  then          % Caso 1
    return min( $t.\text{right}$ )
else                                % Caso 2
    TREE  $p = t.\text{parent}$ 
    while  $p \neq \text{nil}$  and  $t == p.\text{right}$ 
        do
             $t = p$ 
             $p = p.\text{parent}$ 
    return  $p$ 

```

Per passare da successore a predecessore

- right diventa left
- min diventa max

Inserimento – `insertNode()`

TREE insertNode(TREE T , ITEM k , ITEM v)

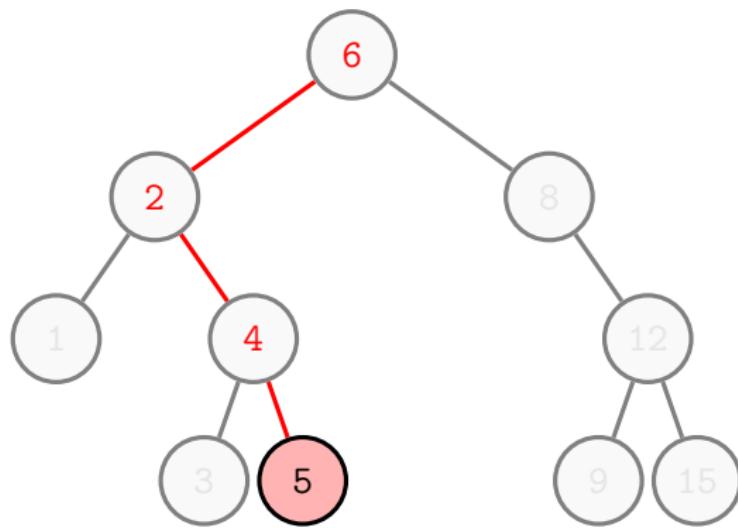
- Inserisce un'associazione chiave-valore (k, v) nell'albero T
- Se la chiave è già presente, sostituisce il valore associato; altrimenti, viene inserita una nuova associazione.
- Se $T == \text{nil}$, restituisce il primo nodo dell'albero.
- Altrimenti, restituisce T inalterato

Implementazione dizionario

`insert(ITEM k , ITEM v)`

`tree = insertNode(tree, k , v)`

Inserimento – esempio



Valore da inserire: 5

- $u = \textcircled{6}$
- $5 < 6$; $u = \textcircled{2}$ (Sinistra)
- $5 > 2$; $u = \textcircled{4}$ (Destra)
- $5 > 4$; $u = \textbf{nil}$ (Destra)

Inserito

Inserimento – implementazione

TREE insertNode(TREE T , ITEM k , ITEM v)

TREE $p = \text{nil}$ % Padre
TREE $u = T$
while $u \neq \text{nil}$ and $u.key \neq k$ **do** % Cerca posizione inserimento
 $p = u$
 $u = \text{iif}(k < u.key, u.left, u.right)$
if $u \neq \text{nil}$ and $u.key == k$ **then**
 $u.value = v$ % Chiave già presente
else
 TREE $new = \text{Tree}(k, v)$ % Crea un nodo coppia chiave-valore
 link(p, new, k)
 if $p == \text{nil}$ **then**
 $T = new$ % Primo nodo ad essere inserito
return T % Restituisce albero non modificato o nuovo nodo

Inserimento – implementazione

link(TREE p , TREE u , ITEM k)

if $u \neq \text{nil}$ then
 $u.parent = p$ % Registrazione padre

if $p \neq \text{nil}$ then
 if $k < p.key$ then $p.left = u$ % Attaccato come figlio sinistro
 else $p.right = u$ % Attaccato come figlio destro

Cancellazione

TREE removeNode(TREE T , ITEM k)

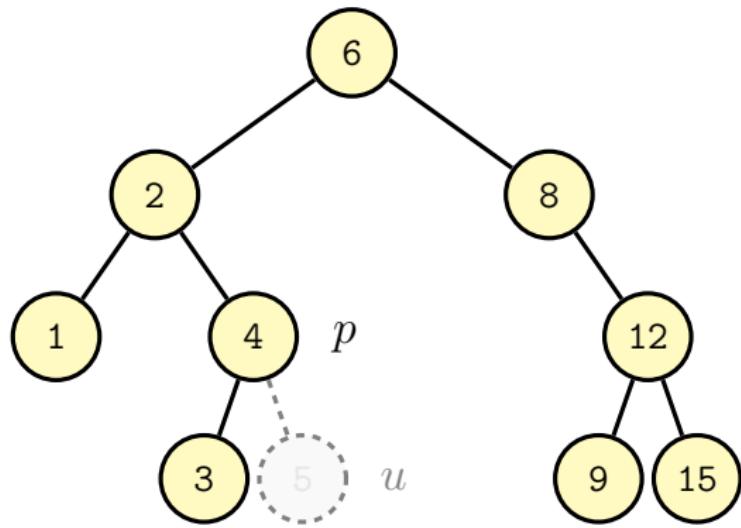
- Rimuove il nodo contenente la chiave k dall'albero T
- Restituisce la radice dell'albero (potenzialmente cambiata)

Implementazione dizionario

`remove(ITEM k)`

`tree = removeNode(tree, k)`

Cancellazione



Caso 1

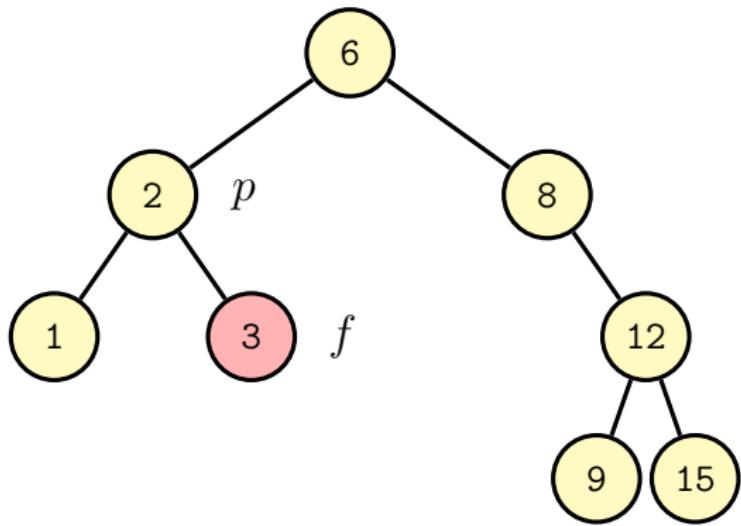
Il nodo da eliminare u non ha figli

Semplicemente si elimina!

Esempio

- Eliminazione 5

Cancellazione



Caso 2

Il nodo da eliminare u ha un solo figlio f

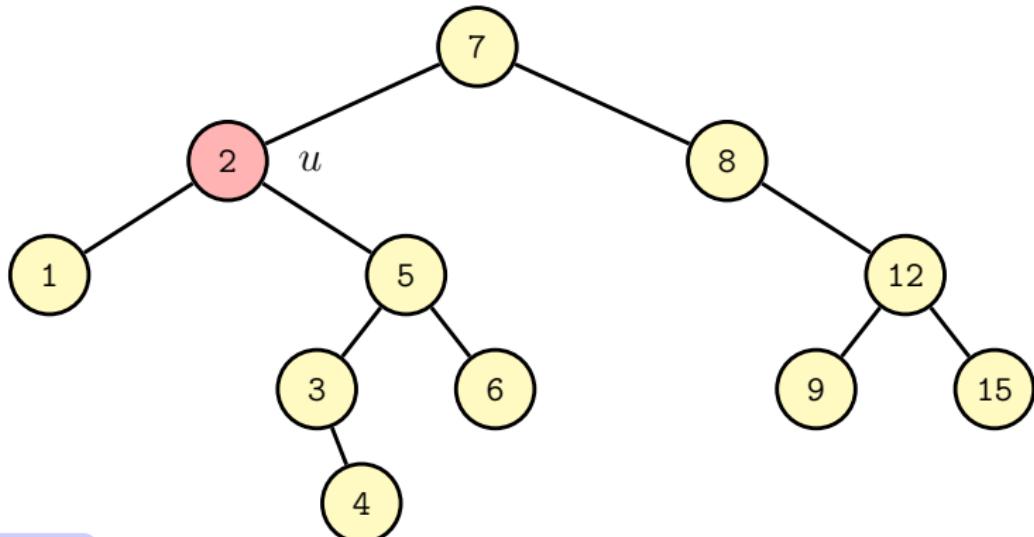
Si elimina u

Si attacca f all'ex-padre p di u in sostituzione di u (**short-cut**)

Esempio

- Eliminazione 4

Cancellazione

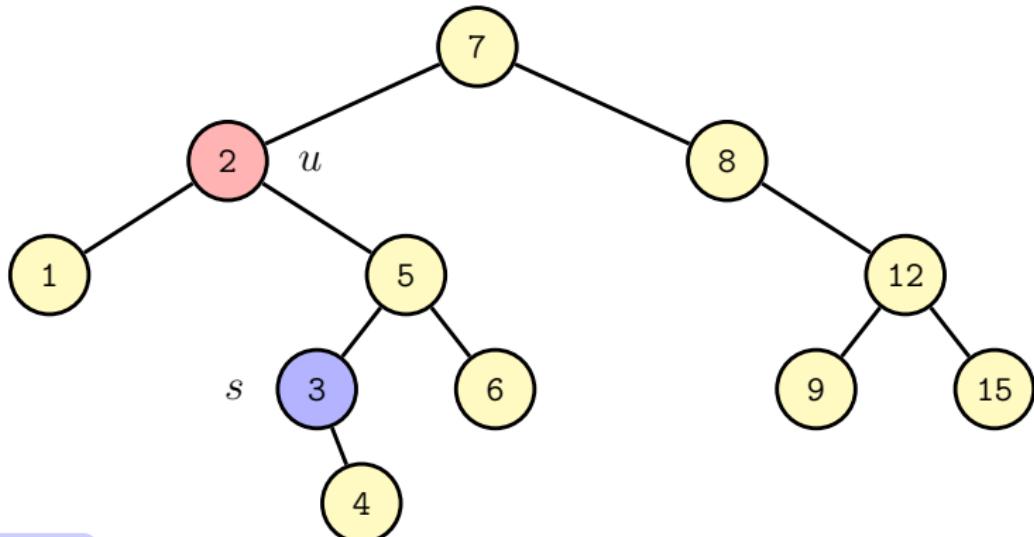


Caso 3

Il nodo da eliminare u ha due figli

- Eliminazione 2

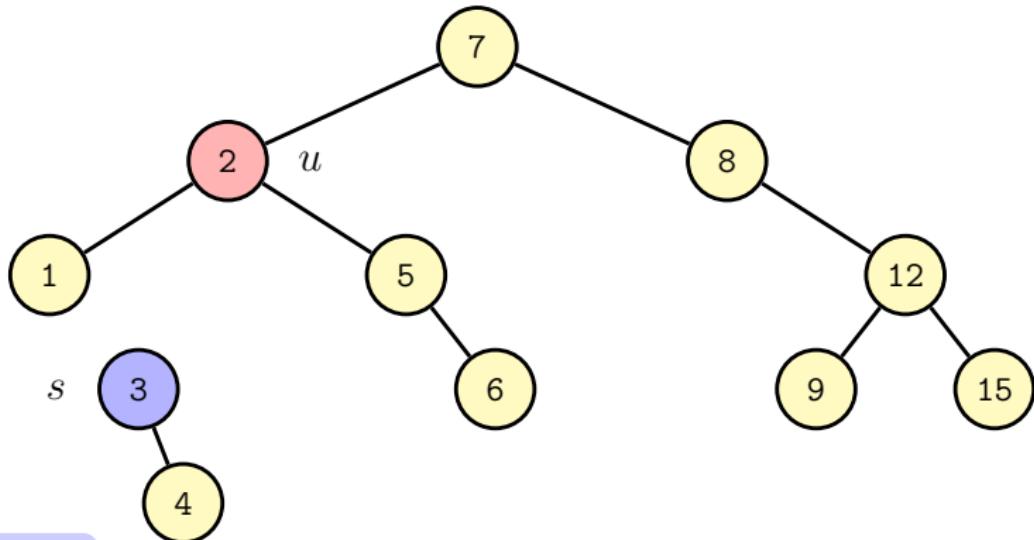
Cancellazione



Caso 3

- Si individua il successore s di u
- Il successore non ha figlio sinistro

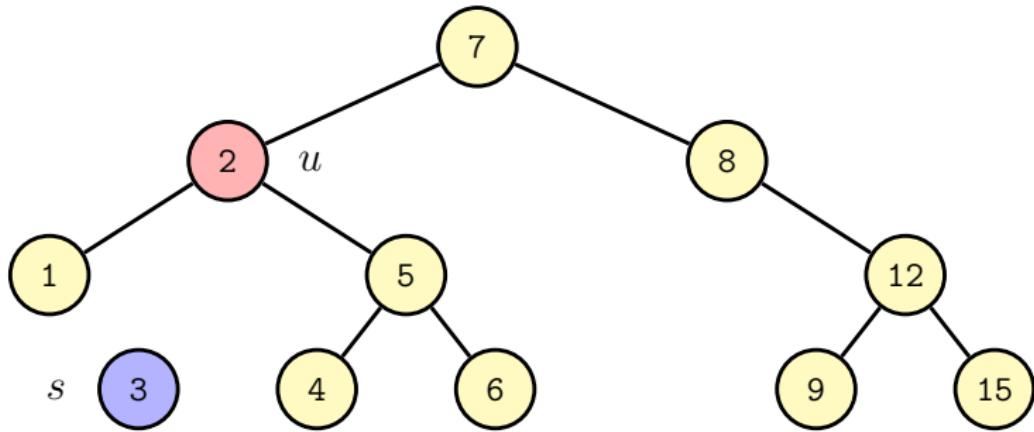
Cancellazione



Caso 3

- Si “stacca” il successore

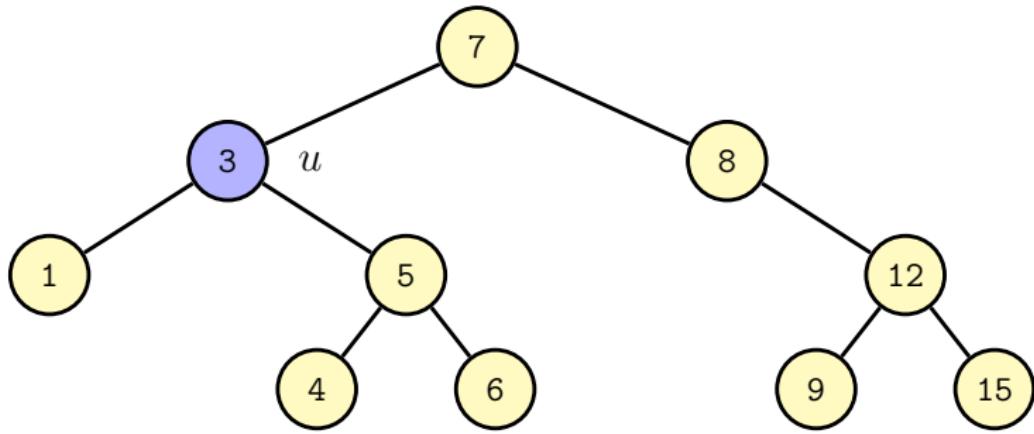
Cancellazione



Caso 3

- Si attacca l'eventuale figlio destro di s al padre di s (**short-cut**)

Cancellazione



Caso 3

- Si copia s su u
- Si rimuove il nodo s

Cancellazione – Implementazione

TREE removeNode(TREE T , ITEM k)

TREE t

TREE $u = \text{lookupNode}(T, k)$

if $u \neq \text{nil}$ then

if $u.\text{left} == \text{nil}$ and $u.\text{right} == \text{nil}$ then % Caso 1

link($u.\text{parent}$, nil, k)

delete u

else if $u.\text{left} \neq \text{nil}$ and $u.\text{right} \neq \text{nil}$ then % Caso 3

[...]

else % Caso 2

[...]

return T

Cancellazione – Implementazione

TREE removeNode(TREE T , ITEM k)

TREE t

TREE $u = \text{lookupNode}(T, k)$

if $u \neq \text{nil}$ **then**

if $u.\text{left} == \text{nil}$ **and** $u.\text{right} == \text{nil}$ **then** % Caso 1

[...]

else if $u.\text{left} \neq \text{nil}$ **and** $u.\text{right} \neq \text{nil}$ **then** % Caso 3

TREE $s = \text{successorNode}()$

link($s.\text{parent}, s.\text{right}, s.\text{key}$)

$u.\text{key} = s.\text{key}$

$u.\text{value} = s.\text{value}$

delete s

else % Caso 2

[...]

return T

Cancellazione – Implementazione

TREE removeNode(TREE T , ITEM k)

TREE t

TREE $u = \text{lookupNode}(T, k)$

if $u \neq \text{nil}$ **then**

if $u.\text{left} == \text{nil}$ **and** $u.\text{right} == \text{nil}$ **then** % Caso 1

 [...]

else if $u.\text{left} \neq \text{nil}$ **and** $u.\text{right} \neq \text{nil}$ **then** % Caso 3

 [...]

else if $u.\text{left} \neq \text{nil}$ **and** $u.\text{right} == \text{nil}$ **then** % Caso 2

 link($u.\text{parent}$, $u.\text{left}$, k)

if $u.\text{parent} = \text{nil}$ **then**

$T = u.\text{left}$

else

 link($u.\text{parent}$, $u.\text{right}$, k)

if $u.\text{parent} = \text{nil}$ **then**

$T = u.\text{right}$

return T

Cancellazione – Dimostrazione

Caso 1 - nessun figlio

- Eliminare foglie non cambia l'ordine dei nodi rimanenti

Caso 2 - solo un figlio (destro o sinistro)

- Se u è il figlio destro (sinistro) di p , tutti i valori nel sottoalbero di f sono maggiori (minori) di p
- Quindi f può essere attaccato come figlio destro (sinistro) di p al posto di u

Cancellazione – Dimostrazione

Caso 3 - due figli

- Il successore s
 - è sicuramente \geq dei nodi nel sottoalbero sinistro di u
 - è sicuramente \leq dei nodi nel sottoalbero destro di u
- quindi può essere sostituito a u
- A quel punto, si ricade nel caso 2

Costo computazionale

Osservazione

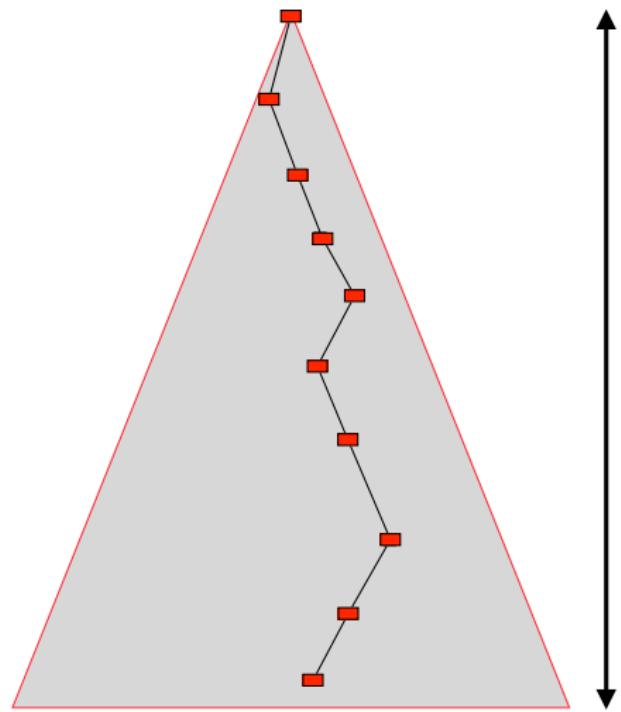
Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia

$h = \text{Altezza dell'albero}$

Tempo di ricerca: $O(h)$

Domande

- Qual è il caso pessimo?
- Qual è il caso ottimo?



Costo computazionale

Osservazione

Le operazioni di ricerca sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia

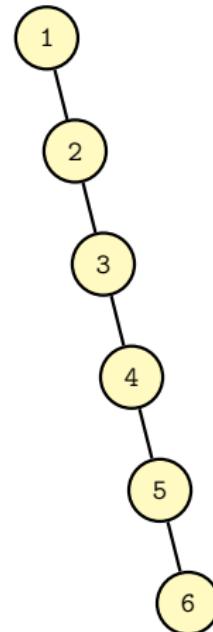
$h = \text{Altezza dell'albero}$

Tempo di ricerca: $O(h)$

Domande

- Qual è il caso pessimo?
- Qual è il caso ottimo?

Caso pessimo: $h = O(n)$



Costo computazionale

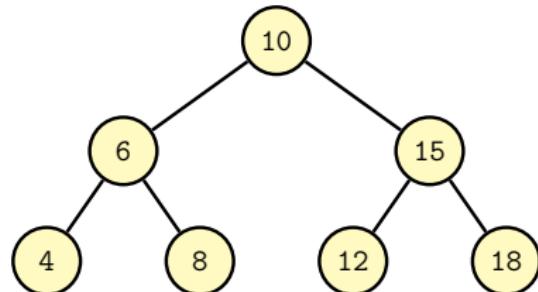
Osservazione

Le operazioni descritte sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia

$h = \text{Altezza dell'albero}$

Tempo di ricerca: $O(h)$

Caso ottimo: $h = O(\log n)$



Domande

- Qual è il caso pessimo?
- Qual è il caso ottimo?

Algoritmi e Strutture Dati

Alberi binari di ricerca bilanciati

Alberto Montresor

Università di Trento

2018/10/25

This work is licensed under a Creative Commons
Attribution-ShareAlike 4.0 International License.



Altezza degli ABR

Qual è l'altezza media di un albero di ricerca?

- Caso "semplice": inserimenti in ordine casuale
 - E' possibile dimostrare che l'altezza media è $O(\log n)$
- Caso generale (inserimenti + cancellazioni):
 - Difficile da trattare

Nella realtà

- Non ci si affida al caso
- Si utilizzano tecniche per mantenere l'albero bilanciato

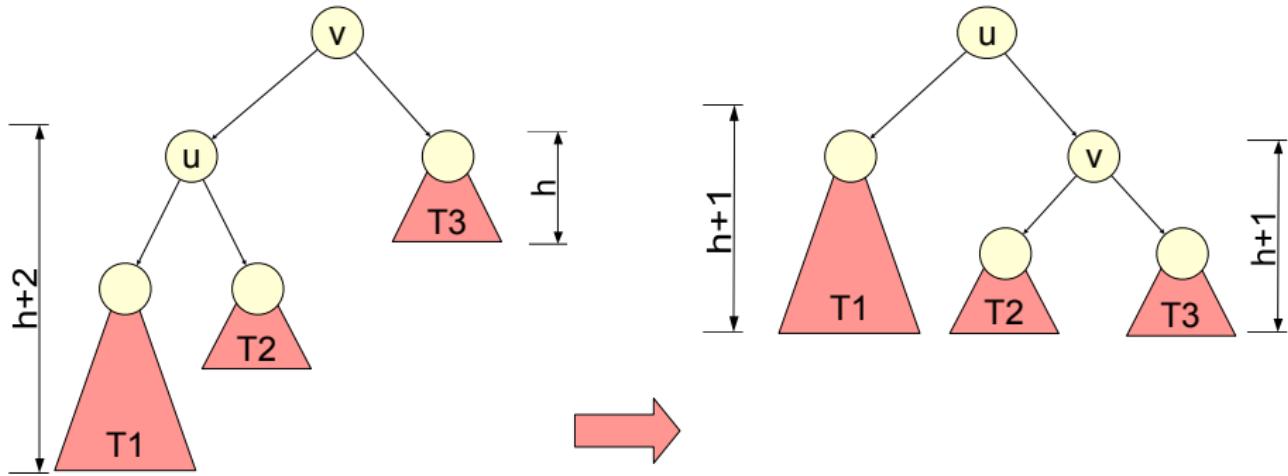
ABR bilanciati

Fattore di bilanciamento

Il fattore di bilanciamento $\beta(v)$ di un nodo v è la massima differenza di altezza fra i sottoalberi di v

- **Alberi AVL** (Adelson-Velskii e Landis, 1962)
 - $\beta(v) \leq 1$ per ogni nodo v
 - Bilanciamento ottenuto tramite rotazioni
- **B-Alberi** (Bayer, McCreight, 1972)
 - $\beta(v) = 0$ per ogni nodo v
 - Specializzati per strutture in memoria secondaria
- **Alberi 2-3** (Hopcroft, 1983)
 - $\beta(v) = 0$ per ogni nodo v
 - Bilanciamento ottenuto tramite merge/split, grado variabile

Rotazione



Alberi Red-Black

Un **albero red-black** è un albero binario di ricerca in cui:

- Ogni nodo è colorato di **rosso** o di **nero**
- Le chiavi vengono mantenute solo nei nodi interni dell'albero
- Le foglie sono costituite da nodi speciali **Nil**

Un **albero red-black** deve rispettare questi vincoli

- ① La radice è nera
- ② Tutte le foglie sono nere
- ③ Entrambi i figli di un nodo rosso sono neri
- ④ Ogni cammino semplice da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri

Alberi Red-Black – Memorizzazione

TREE

TREE *parent*

TREE *left*

TREE *right*

int color

ITEM *key*

ITEM *value*

Nodi Nil

- Nodo sentinella il cui scopo è evitare 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'ABR corrispondente

Altezza nera

Altezza nera di un nodo v

L'altezza nera $b(v)$ di un nodo v è il numero di nodi neri lungo ogni percorso da v (escluso) ad ogni foglia (inclusa) del suo sottoalbero.

Altezza nera di un albero Red-Black

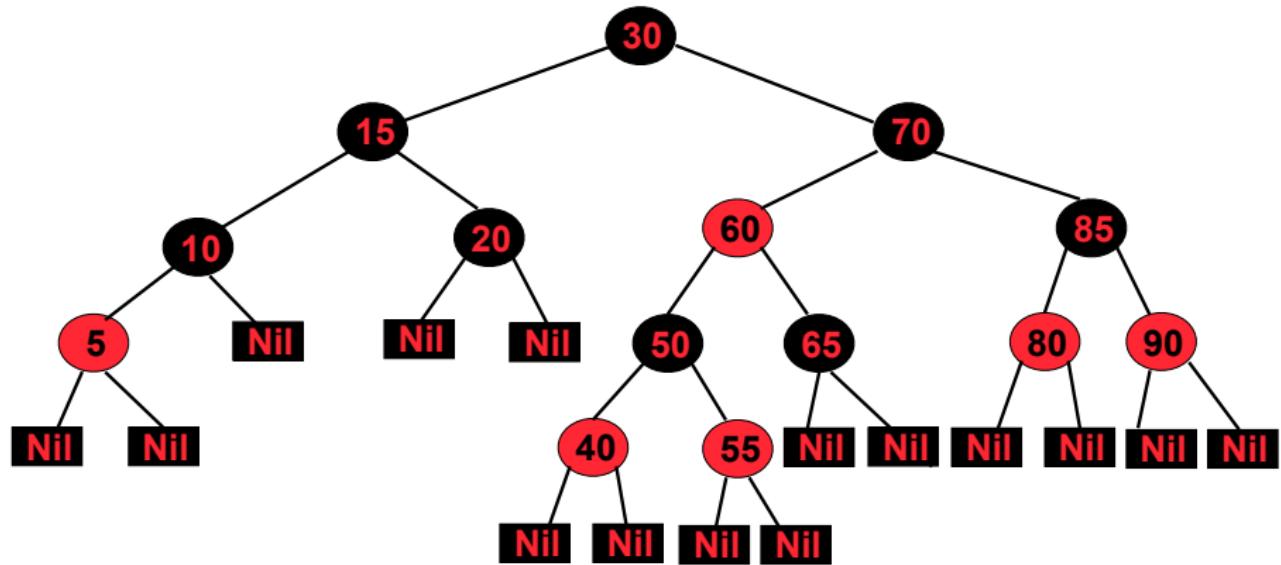
L'altezza nera di un albero Red-Black è pari all'altezza nera della sua radice

Entrambe ben definite perché tutti i percorsi hanno lo stesso numero di nodi neri (Vincolo 4)

Esempi

3. Entrambi i figli di un nodo rosso sono neri.

Ma un nodo nero può avere figli neri!



Esempi

4. *Ogni percorso da un nodo interno ad un nodo Nil ha lo stesso numero di nodi neri.* Altezza nera di questo albero: 3

Esempi

Più colorazioni sono possibili – Versione 1

Altezza nera di questo albero: 3

Esempi

Più colorazioni sono possibili – Versione 2

Altezza nera di questo albero: 3

Esempi

Cambiare colorazione può cambiare l'altezza nera

Altezza nera di questo albero: 3

Esempi

Cambiare colorazione può cambiare l'altezza nera

Stesso albero, altezza nera di questo albero: 2

Esempi

Questo albero può essere un albero Red-Black?

Esempi

Per il vincolo 4, l'altezza nera massima deve essere 3.

Esempi

Supponiamo che 60 e 70 siano entrambi neri.
Impossibile per il vincolo 3.

Esempi

Quindi, almeno uno fra 60 e 70 deve essere rosso.

Per il vincolo 3, al massimo uno fra 60 e 70 deve essere rosso.

Esempi

Proviamo a colorare di rosso il nodo 60. Esistono percorsi con 2 nodi neri e con 3 nodi neri. **Impossibile per il vincolo 4.**

Esempi

Proviamo a colorare di rosso il nodo 70. Esistono 2 percorsi con 2 nodi neri; per il vincolo 4, devono esserci al più 2 nodi neri lungo tutti i percorsi!

Esempi

Questa è l'ultima possibilità. Impossibile perchè non rispetta il vincolo 1 (La radice deve essere nera)

Esempi

Questo albero non può essere un albero Red-Black!

Inserimento

Durante la modifica di un albero Red-Black

- È possibile che le condizioni di bilanciamento risultino violate

Quando i vincoli 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

Rotazione a sinistra

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.\text{right}$

TREE $p \leftarrow x.\text{parent}$

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

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

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

(2) $x.\text{parent} \leftarrow y$

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

(3) **if** $p \neq \text{nil}$ **then**

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

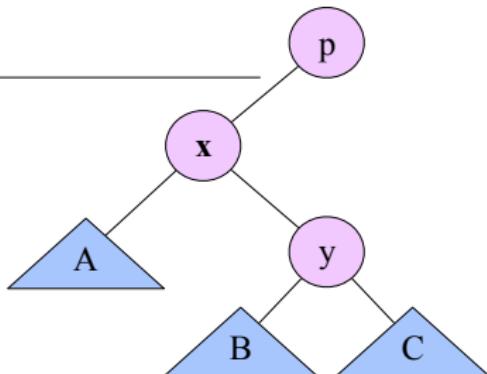
return y

• 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



Rotazione a sinistra

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.\text{right}$

TREE $p \leftarrow x.\text{parent}$

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

(1) **if** $y.\text{left} \neq \text{nil}$ **then** $y.\text{left.parent} \leftarrow x$ % x diventa figlio sinistro di y

(2) $y.\text{left} \leftarrow x$

(2) $x.\text{parent} \leftarrow y$

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

(3) **if** $p \neq \text{nil}$ **then**

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

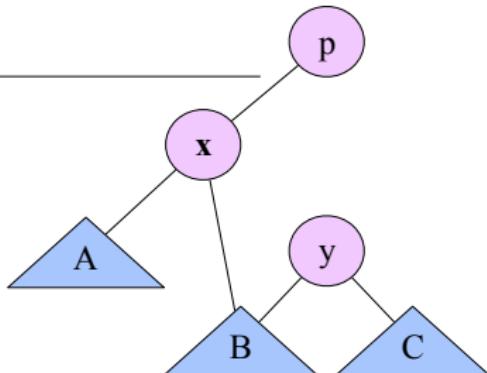
return y

• 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



Rotazione a sinistra

TREE rotateLeft(TREE x)

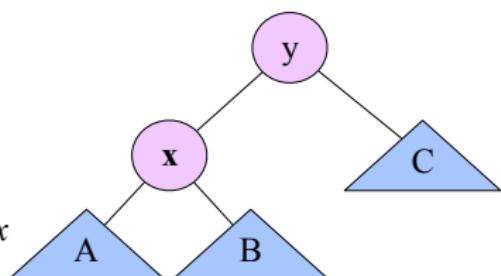
```

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

p

• 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



Rotazione a sinistra

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.\text{right}$

TREE $p \leftarrow x.\text{parent}$

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

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

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

(2) $x.\text{parent} \leftarrow y$

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

(3) **if** $p \neq \text{nil}$ **then**

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

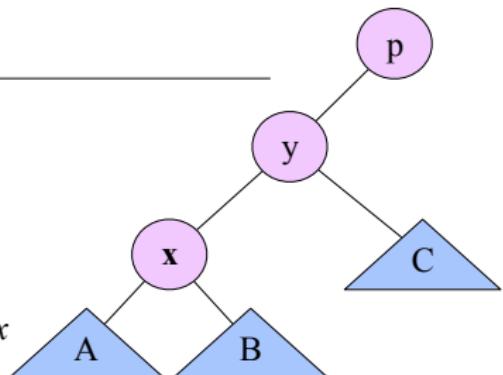
return y

• 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 in alberi Red-Black

Inserimento

- Si cerca la posizione usando la stessa procedura usata per gli alberi binari di ricerca
- Si colora il nuovo nodo di **rosso**

Quale dei quattro vincoli può essere violato?

- ➊ La radice è nera
- ➋ Tutte le foglie sono nere
- ➌ Entrambi i figli di un nodo rosso sono neri
- ➍ Ogni cammino semplice da un nodo u ad una delle foglie contenute nel sottoalbero radicato in u hanno lo stesso numero di nodi neri

Come modificare la `insertNode()`

TREE `insertNode(TREE T, ITEM k, ITEM v)`

```

TREE p = nil                                % Padre
TREE u = T
while u ≠ nil and u.key ≠ k do          % Cerca posizione inserimento
    p = u
    u = iif(k < u.key, u.left, u.right)
if u ≠ nil and u.key == k then           % Chiave già presente
    u.value = v
else
    TREE new = Tree(k, v)                % Crea un nodo coppia chiave-valore
    link(p, new, k)
    balanceInsert(new)
    if p == nil then
        T = n                            % Primo nodo ad essere inserito
return T                                    % Restituisce albero non modificato o nuovo nodo

```

Inserimento in alberi Red-Black

Principi generali

- Ci spostiamo verso l'alto lungo il percorso di inserimento
- Ripristinare il vincolo 3 (figli neri di nodo rosso)
- Spostiamo le violazioni verso l'alto rispettando il vincolo 4 (mantenendo l'altezza nera dell'albero)
- Al termine, coloriamo la radice di nero (vincolo 1)

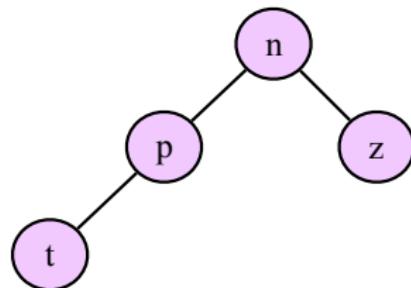
Nota

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

balanceInsert(TREE t)

• Nodi coinvolti

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



balanceInsert(TREE t)

$t.color \leftarrow \text{RED}$

while $t \neq \text{nil}$ **do**

 TREE $p \leftarrow t.parent$

% Padre

 TREE $n \leftarrow \text{if}(p \neq \text{nil}, p.parent, \text{nil})$

% Nonno

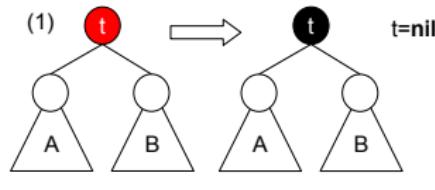
 TREE $z \leftarrow \text{if}(n = \text{nil}, \text{nil}, \text{if}(n.left = p, n.right, n.left))$

% Zio

Inserimento – 7 casi possibili

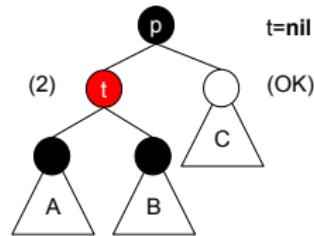
• Caso 1:

- 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



Inserimento – 7 casi possibili

• 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 $t \neq \text{nil}$ **do**

 TREE $p \leftarrow t.parent$

 % Padre

 TREE $n \leftarrow \text{iif}(p.parent \neq \text{nil}, p.parent, \text{nil})$

 % Nonno

 TREE $z \leftarrow \text{iif}(n = \text{nil}, \text{nil}, \text{iif}(n.left = p, right, left))$

 % Zio

if $p = \text{nil}$ **then**

 % Caso (1)

$t.color \leftarrow \text{BLACK}$

$t \leftarrow \text{nil}$

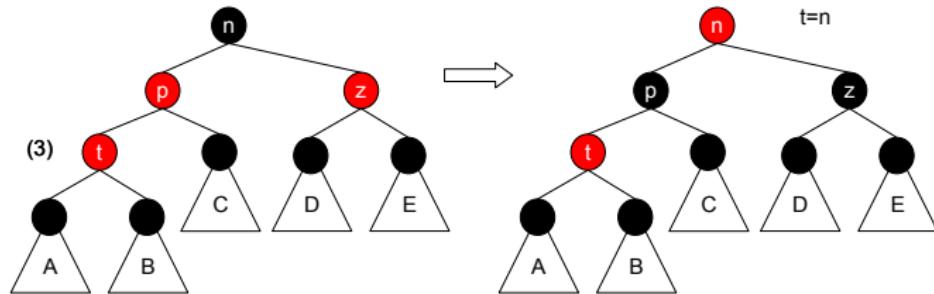
else if $p.color = \text{BLACK}$ **then**

 % Caso (2)

$t \leftarrow \text{nil}$

Inserimento – 7 casi possibili

- ♦ **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.



Inserimento – 7 casi possibili

- **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.

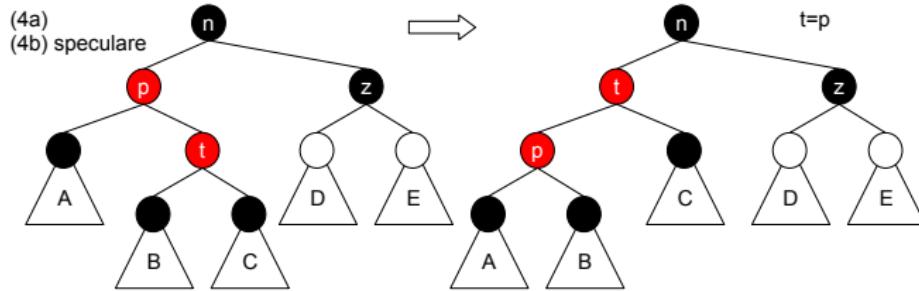
else if $z.color = \text{RED}$ **then**

$p.color \leftarrow z.color \leftarrow \text{BLACK}$
 $n.color \leftarrow \text{RED}$
 $t \leftarrow n$

% Caso (3)

Inserimento – 7 casi possibili

- **Caso 4a,4b**
 - t rosso
 - p rosso
 - z nero
 - Si assuma che t sia figlio destro di p e che p sia figlio sinistro di n
 - Una rotazione a sinistra 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 sinistri dei loro padri
 - I nodi coinvolti nel cambiamento sono p e t , entrambi rossi, quindi la lunghezza dei cammini neri non cambia



Inserimento – 7 casi possibili

- **Caso 4a,4b**
 - t rosso
 - p rosso
 - z nero
 - Si assuma che t sia figlio destro di p e che p sia figlio sinistro di n
 - Una rotazione a sinistra 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 sinistri dei loro padri
 - I nodi coinvolti nel cambiamento sono p e t , entrambi rossi, quindi la lunghezza dei cammini neri non cambia

else

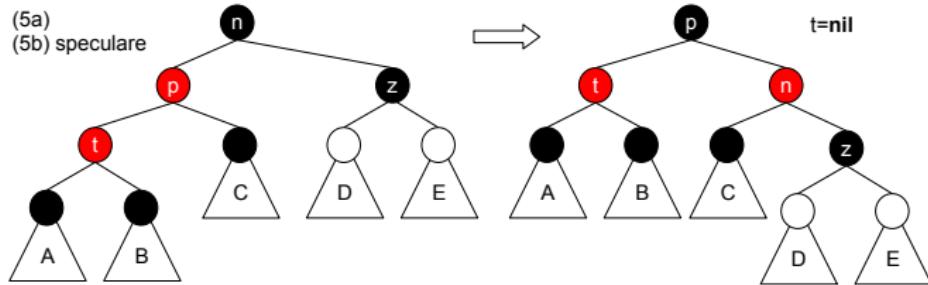
```

if ( $t = p.right$ ) and ( $p = n.left$ ) then % Caso (4.a)
     $n.left \leftarrow \text{rotateLeft}(p)$ 
     $t \leftarrow p$ 
else if ( $t = p.left$ ) and ( $p = n.right$ ) then % Caso (4.b)
     $n.right \leftarrow \text{rotateRight}(p)$ 
     $t \leftarrow p$ 

```

Inserimento – 7 casi possibili

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



Inserimento – 7 casi possibili

- **Caso 5a,5b**
 - t rosso
 - p rosso
 - z nero
 - Si assume 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)
    |  $n.left \leftarrow \text{rotateRight}(n)$ 
  else if ( $t = p.right$ ) and ( $p = n.right$ ) then % Caso (5.b)
    |  $n.right \leftarrow \text{rotateLeft}(n)$ 
   $p.color \leftarrow \text{BLACK}$ 
   $n.color \leftarrow \text{RED}$ 
   $t \leftarrow \text{nil}$ 

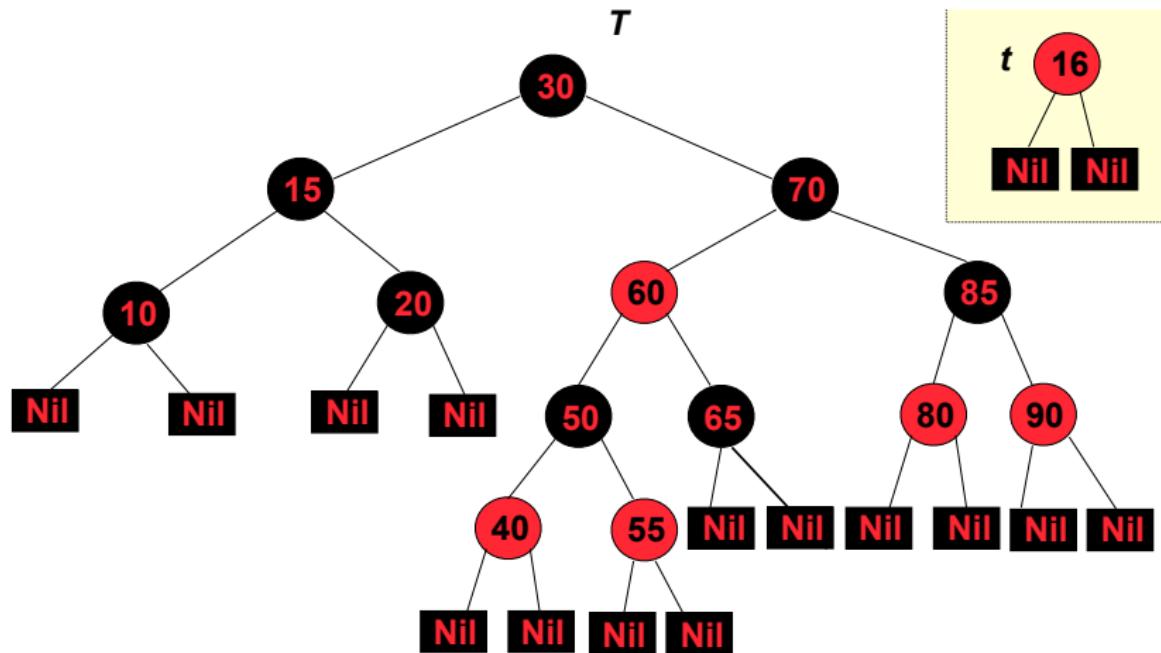
```

All together, now!

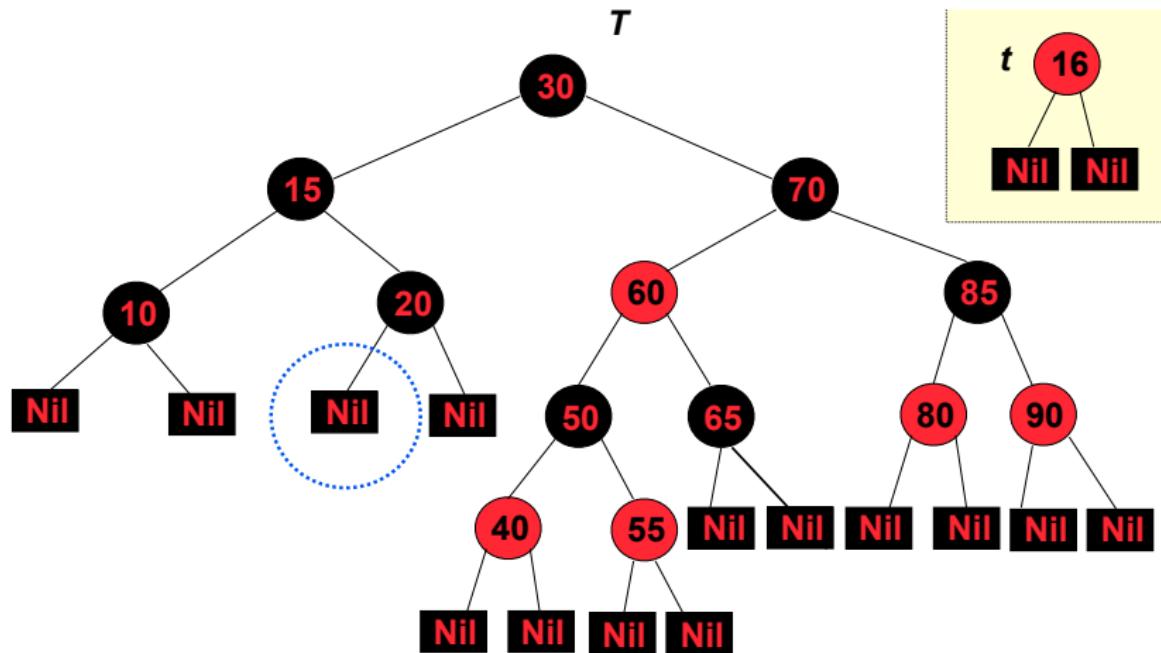
```

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
  
```

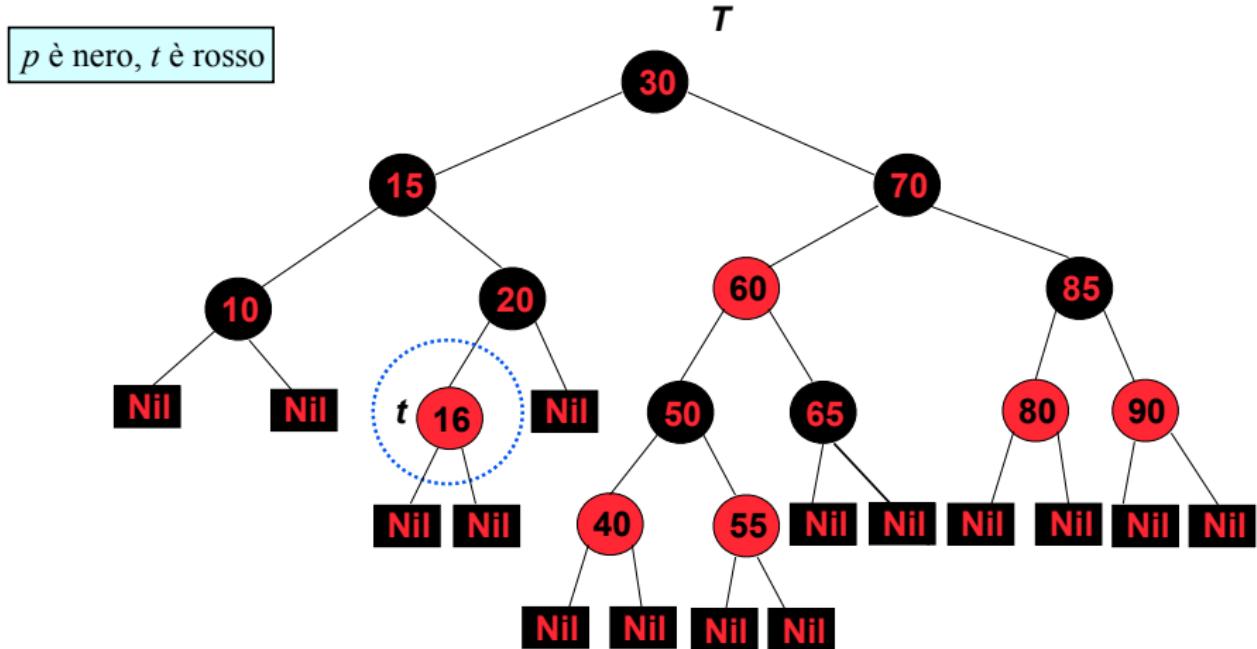
Inserimento – Esempio



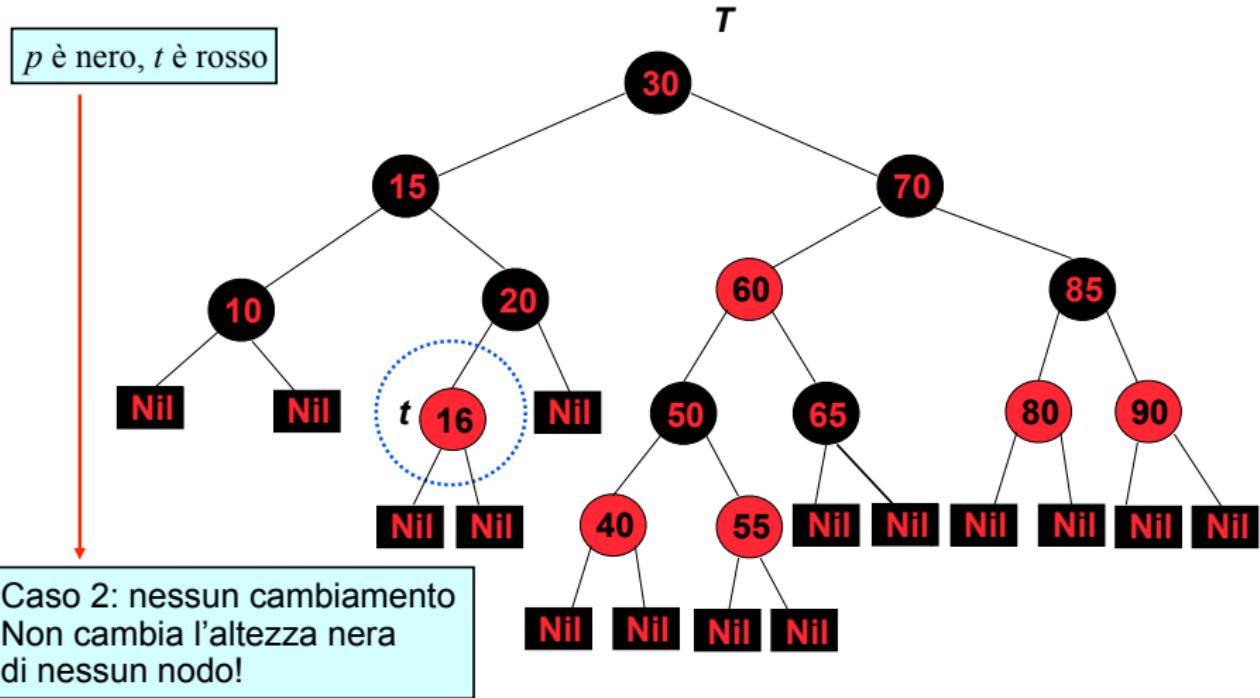
Inserimento – Esempio



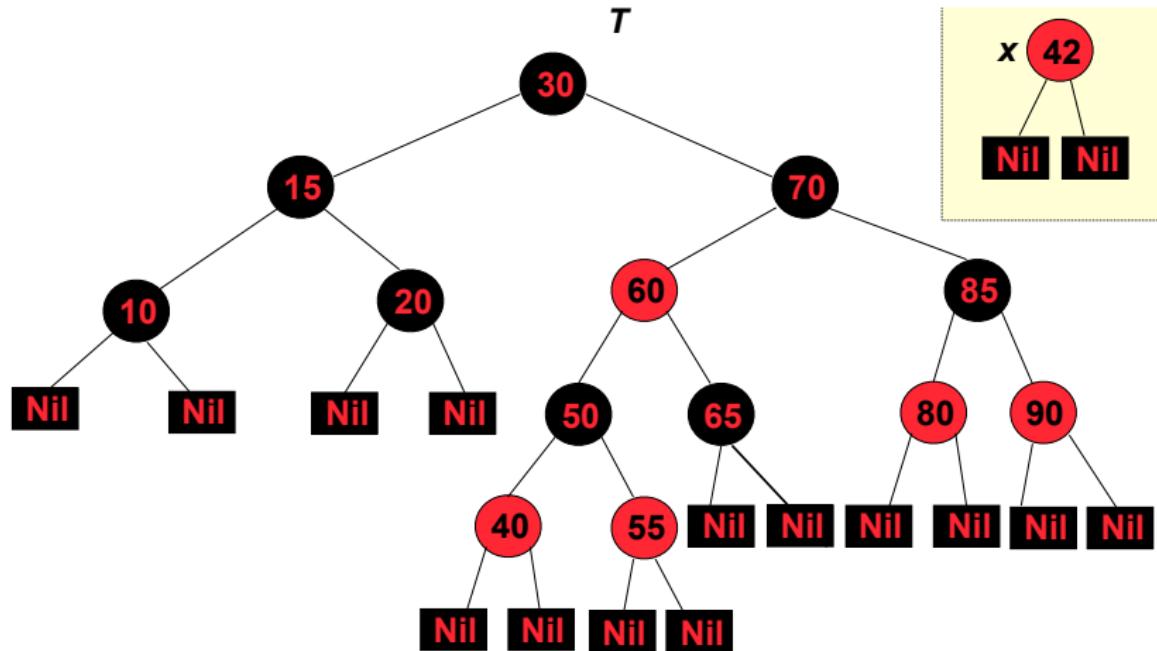
Inserimento – Esempio



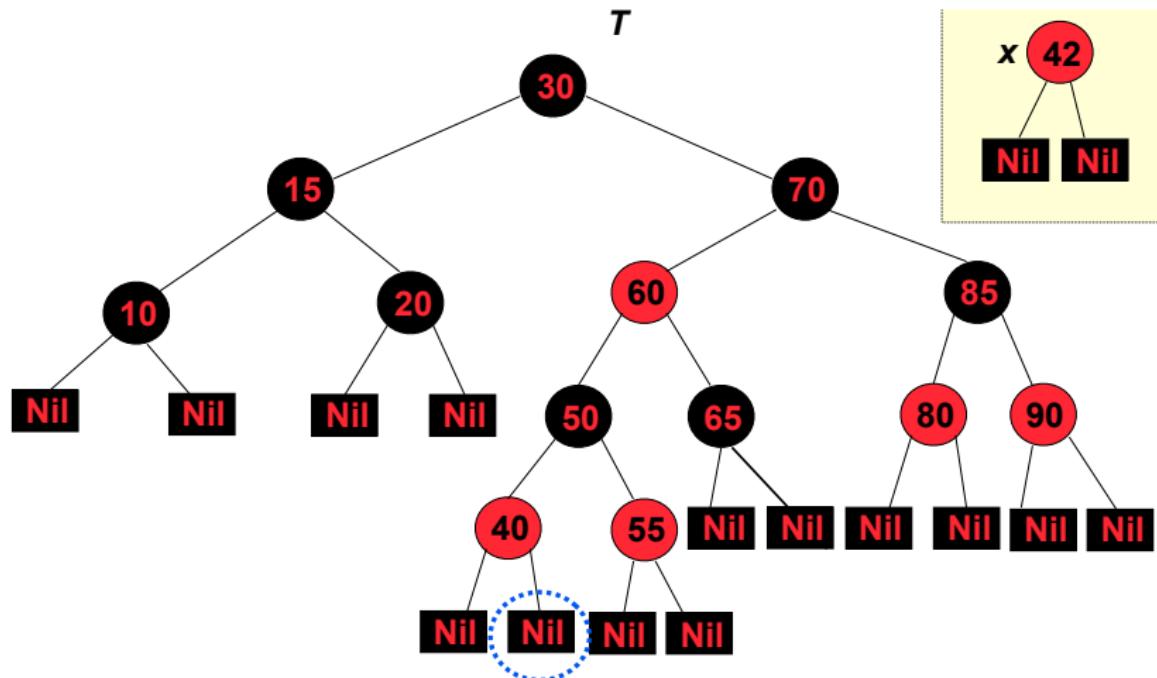
Inserimento – Esempio



Inserimento – Esempio

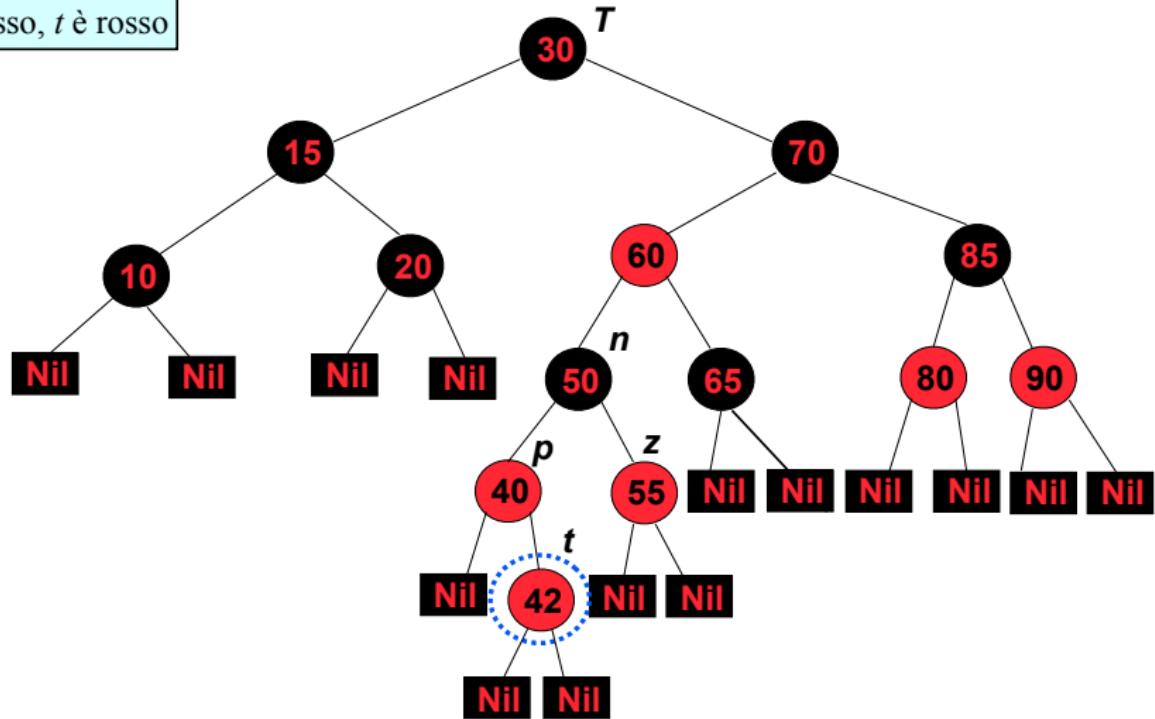


Inserimento – Esempio

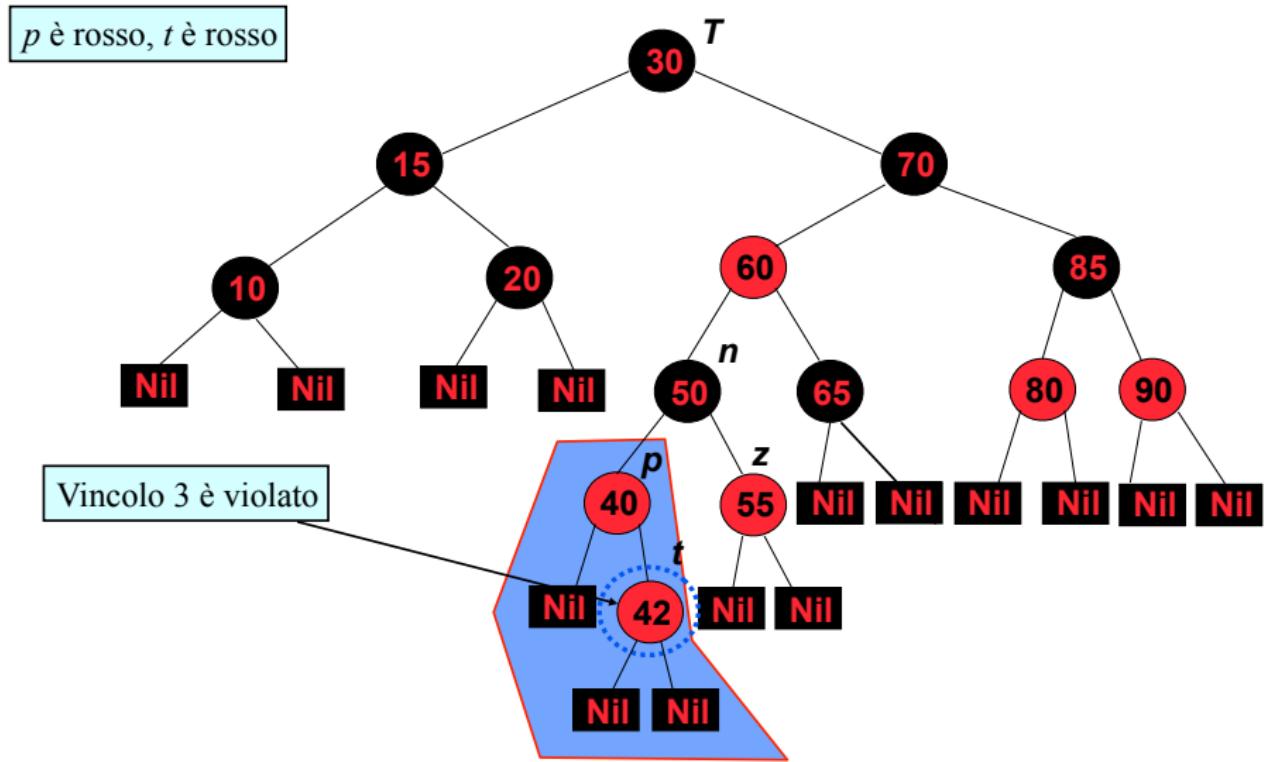


Inserimento – Esempio

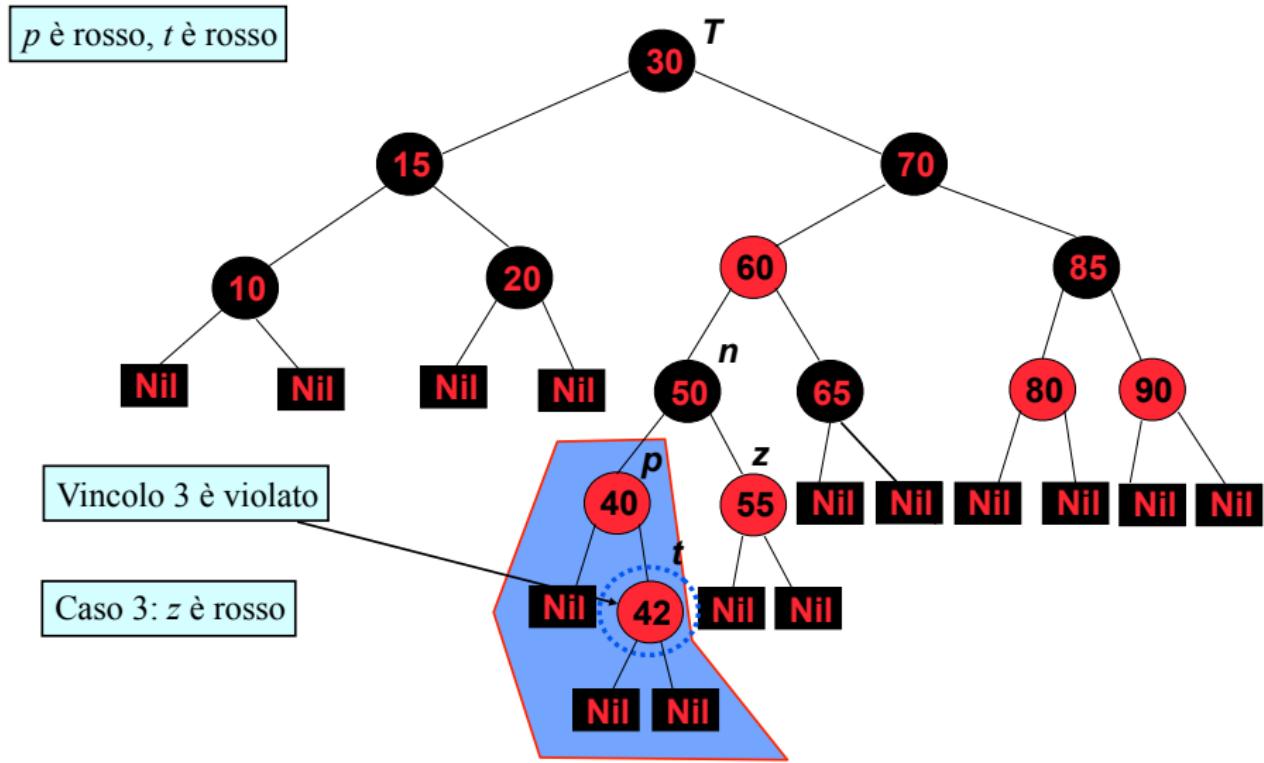
p è rosso, t è rosso



Inserimento – Esempio

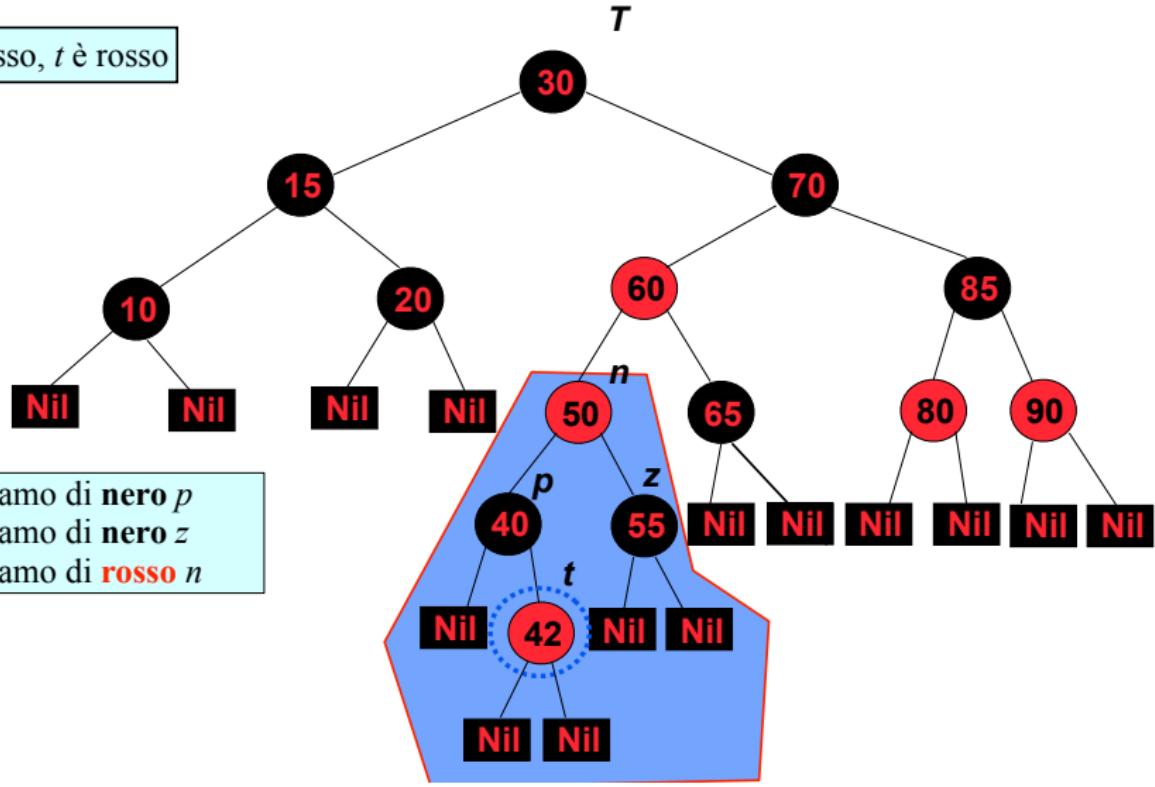


Inserimento – Esempio

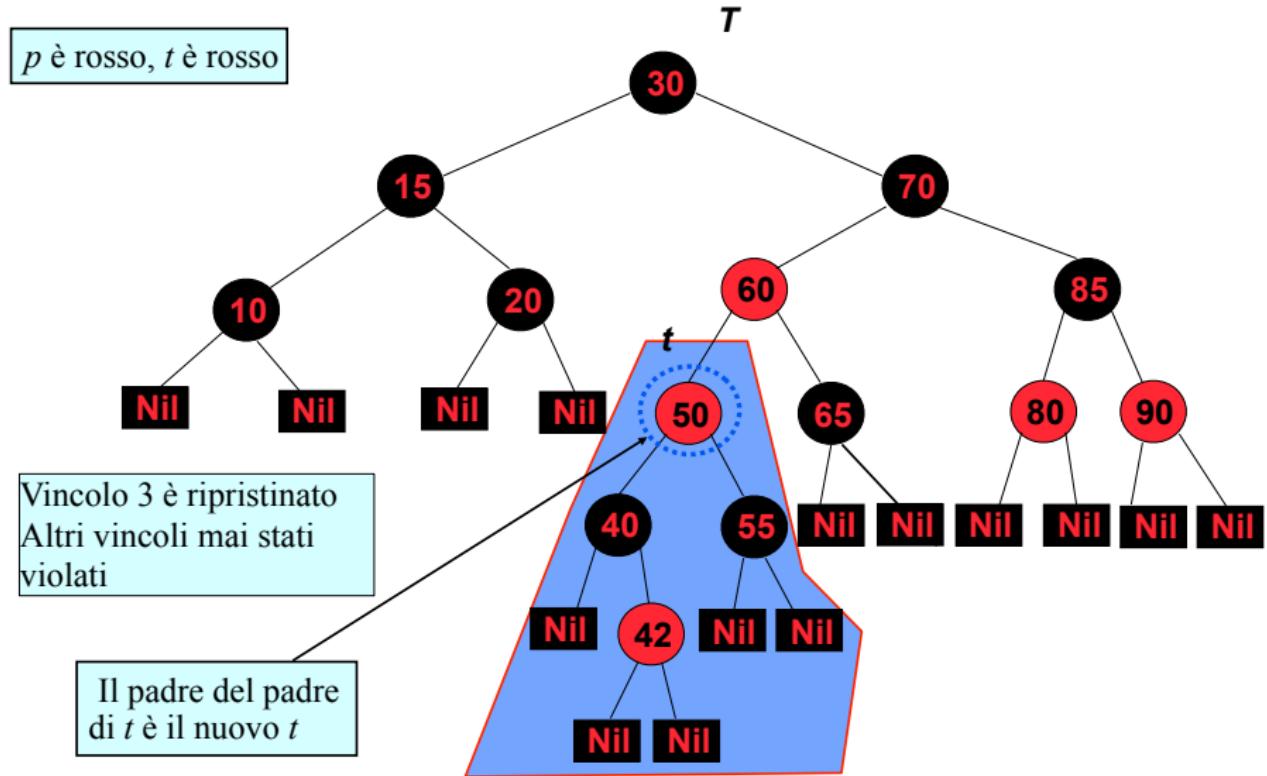


Inserimento – Esempio

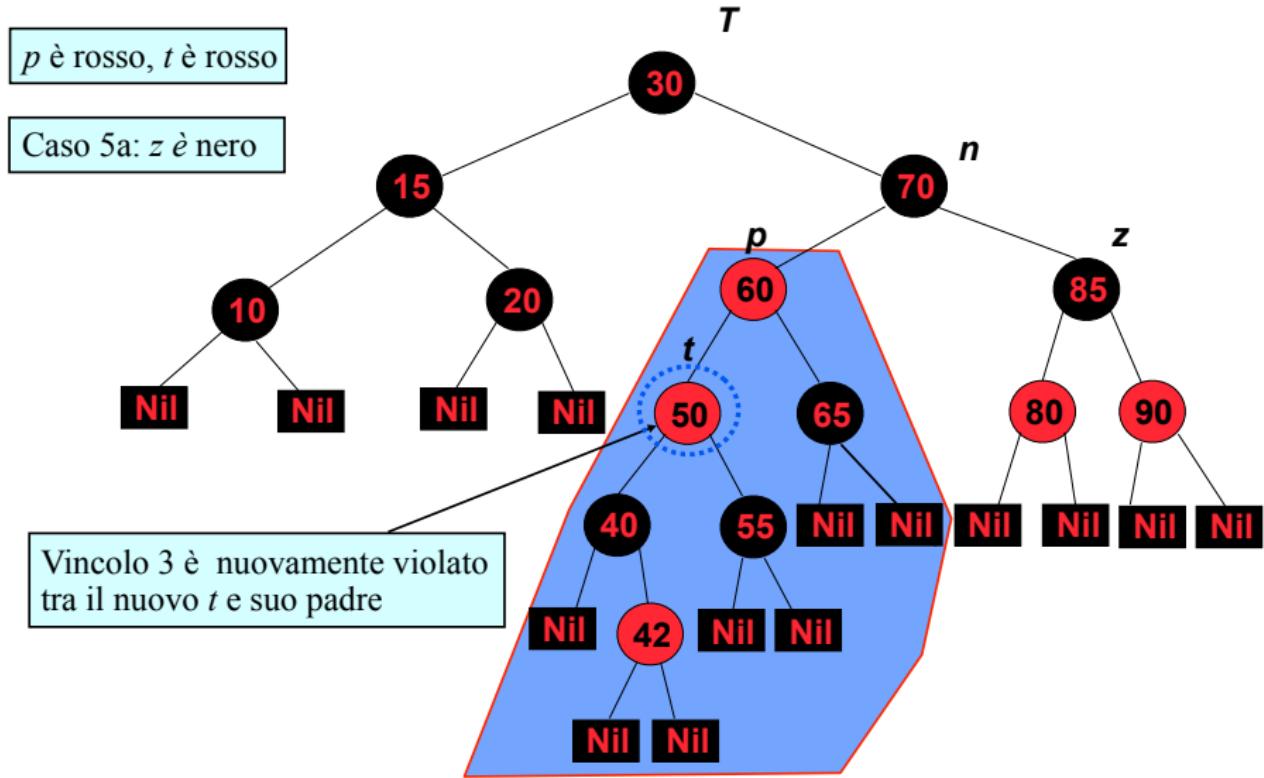
p è rosso, *t* è rosso



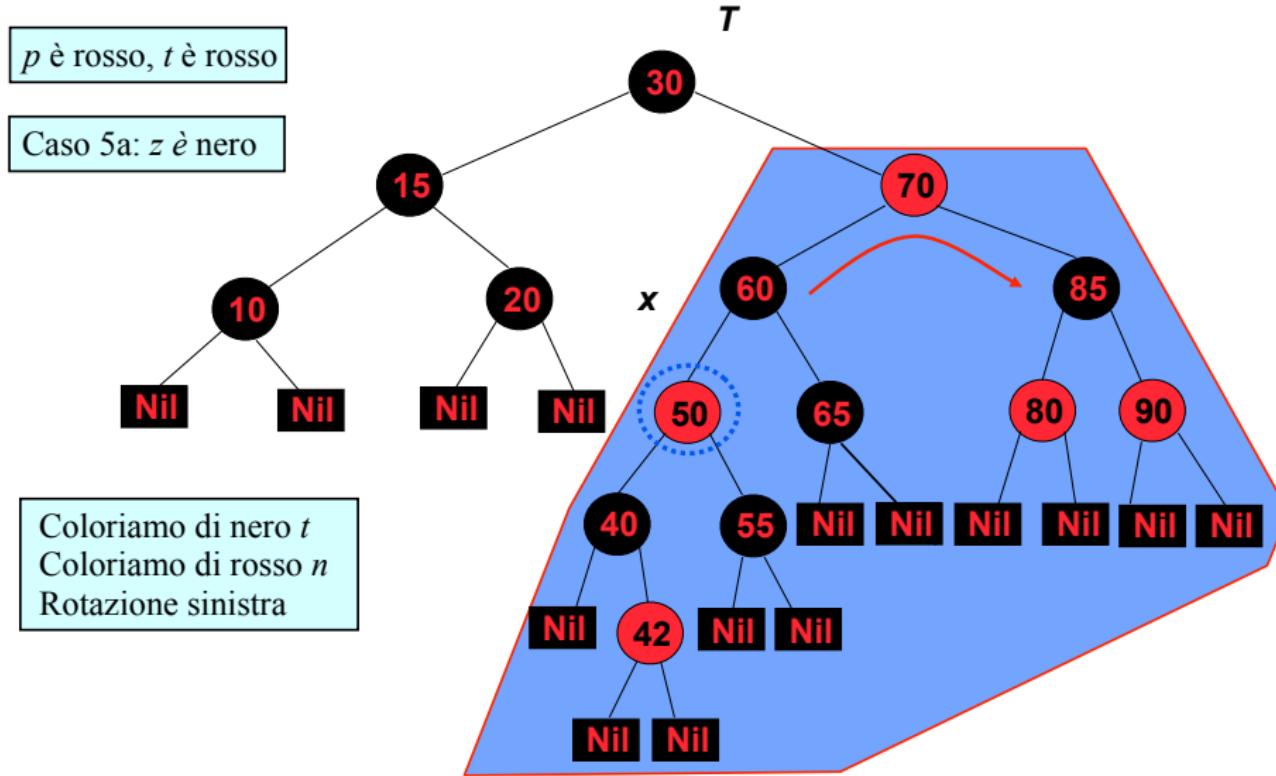
Inserimento – Esempio



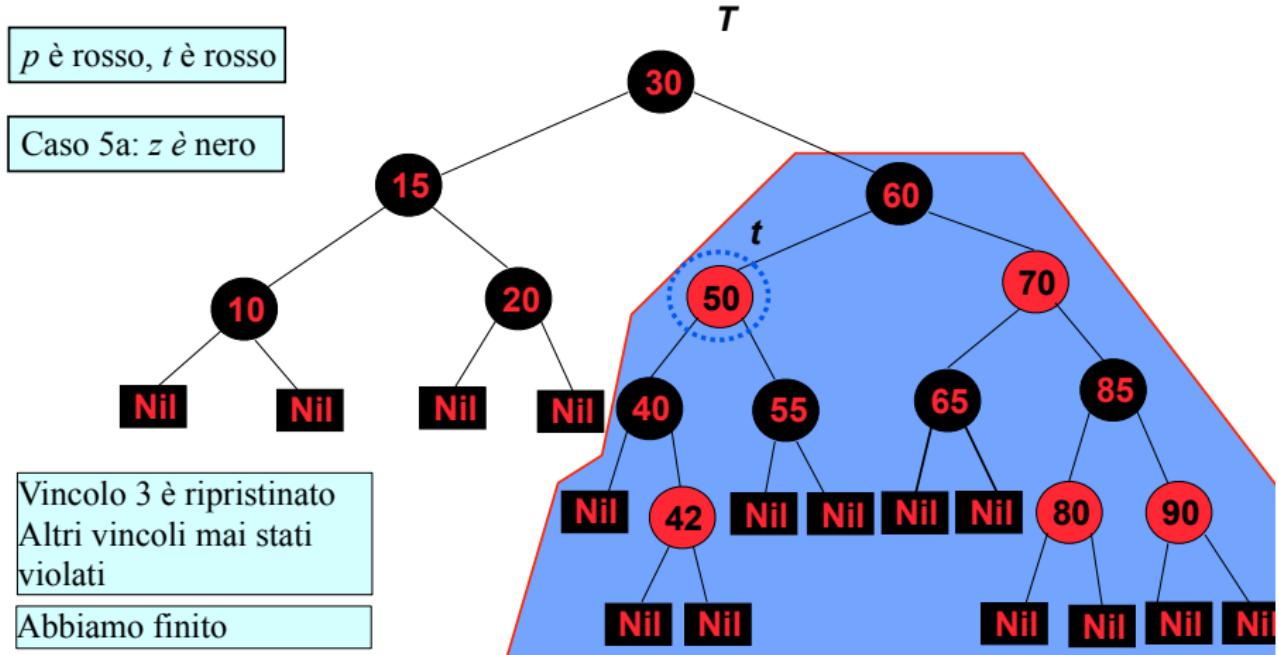
Inserimento – Esempio



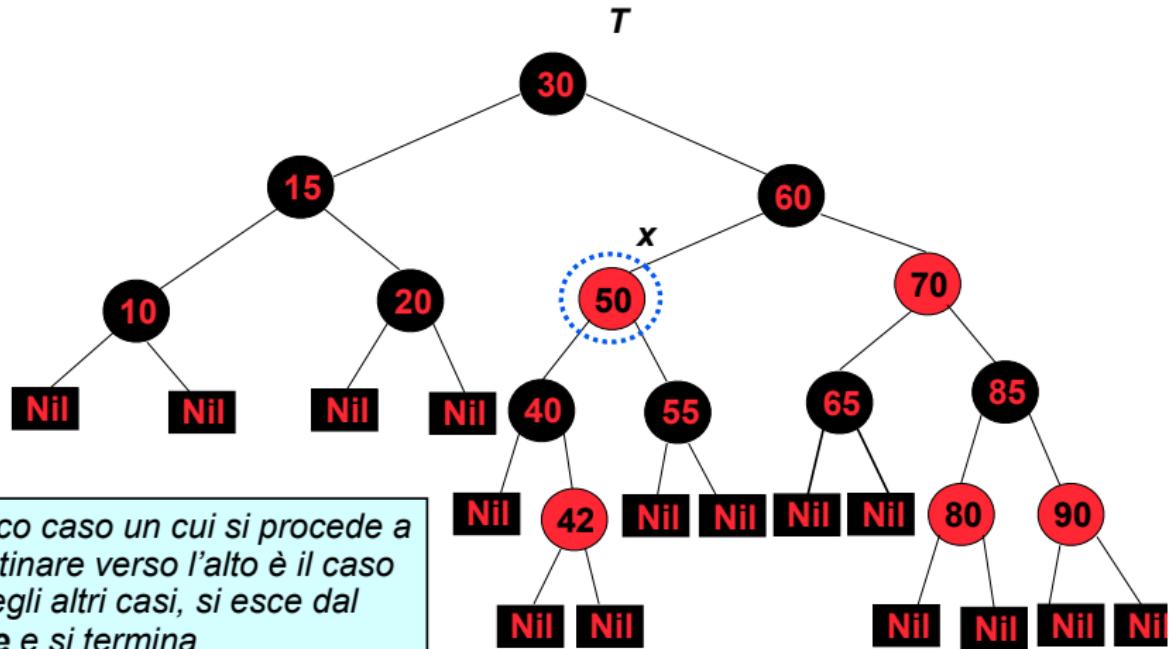
Inserimento – Esempio



Inserimento – Esempio



Inserimento – Esempio



Inserimento – Complessità

Complessità 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 3)

Nota

- E' possibile effettuare una “top-down” insertion
- Si scende fino al punto di inserimento, “aggiustando” l'albero mano a mano
- Si effettua l'inserimento in una foglia

Cancellazione in Alberi Red-Black

- L'algoritmo di cancellazione per alberi Red-Black 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?

Cancellazione in Alberi Red-Black

- 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)!

Cancellazione in Alberi Red-Black

balanceDelete(TREE T , TREE t)

```

while  $t \neq T$  and  $t.\text{color} = \text{BLACK}$  do
    TREE  $p = t.\text{parent}$                                 % Padre
    if  $t = p.\text{left}$  then
        TREE  $f = p.\text{right}$                             % Fratello
        TREE  $ns = f.\text{left}$                             % Nipote sinistro
        TREE  $nd = f.\text{right}$                            % Nipote destro
        if  $f.\text{color} == \text{RED}$  then                   % (1)
             $p.\text{color} = \text{RED}$ 
             $f.\text{color} = \text{BLACK}$ 
            rotateLeft( $p$ )
            %  $t$  viene lasciato inalterato, quindi si ricade nei casi (2),(3),(4)
        else
            if  $ns.\text{color} == nd.\text{color} == \text{BLACK}$  then           % (2)
                 $f.\text{color} = \text{RED}$ 
                 $t = p$ 
            else if  $ns.\text{color} == \text{RED}$  and  $nd.\text{color} == \text{BLACK}$  then       % (3)
                 $ns.\text{color} = \text{BLACK}$ 
                 $f.\text{color} = \text{RED}$ 
                rotateRight( $f$ )
                %  $t$  viene lasciato inalterato, quindi si ricade nel caso (4)
            else if  $nd.\text{color} == \text{RED}$  then                   % (4)
                 $f.\text{color} = p.\text{color}$ 
                 $p.\text{color} = \text{BLACK}$ 
                 $nd.\text{color} = \text{BLACK}$ 
                rotateLeft( $p$ )
                 $t = T$ 
            else

```

Cancellazione in Alberi Red-Black

La 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

Complessità

- In altre parole, è possibile visitare al massimo un numero $O(\log n)$ di casi, ognuno dei quali è gestito in tempo $O(1)$