Algoritmi e Strutture Dati

Alberi binari di ricerca

Alberto Montresor

Università di Trento

2020/10/22

Sommario

Introduzione

Dizionario

È 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

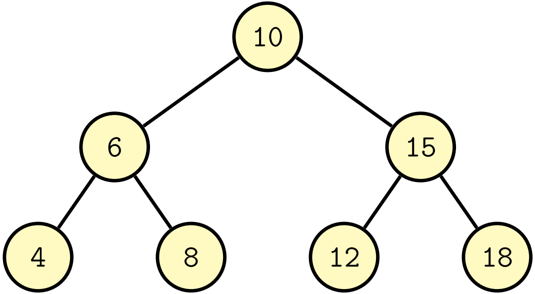
TREE
TREE *parent*
TREE *left*
TREE *right*
ITEM *key*
ITEM *value*

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

- $\textsc{Tree}$ lookupNode($\textsc{Tree}\ T, \textsc{Item}\ k$)
- $\textsc{Tree}$ insertNode($\textsc{Tree}\ T, \textsc{Item}\ k, \textsc{Item}\ v$)
- $\textsc{Tree}$ removeNode($\textsc{Tree}\ T, \textsc{Item}\ k$)

DICTIONARY

TREE *tree*

Dictionary()
  $\mathit{tree} = \mathbf{nil}$

Ricerca – lookupNode()

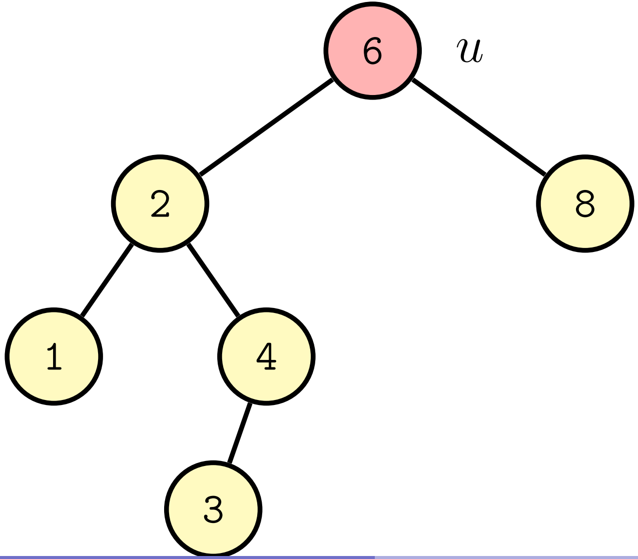
ITEM lookupNode(TREE T, ITEM k)

- 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 ≠ nil then
    return t.value()
else
    return nil
```

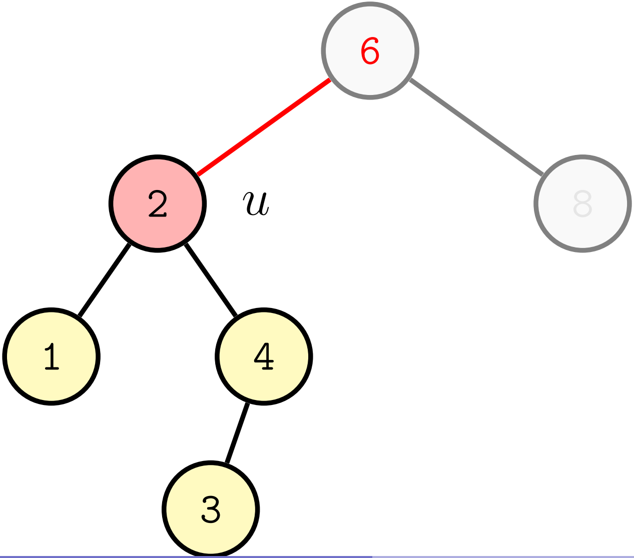
Ricerca – esempio



Valore cercato: 3

- u = 6

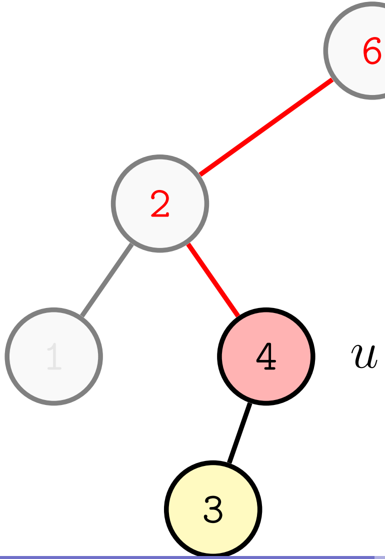
Ricerca – esempio



Valore cercato: 3

- $u =$ 6
- $3 < 6$; $u =$ 2 (Sinistra)

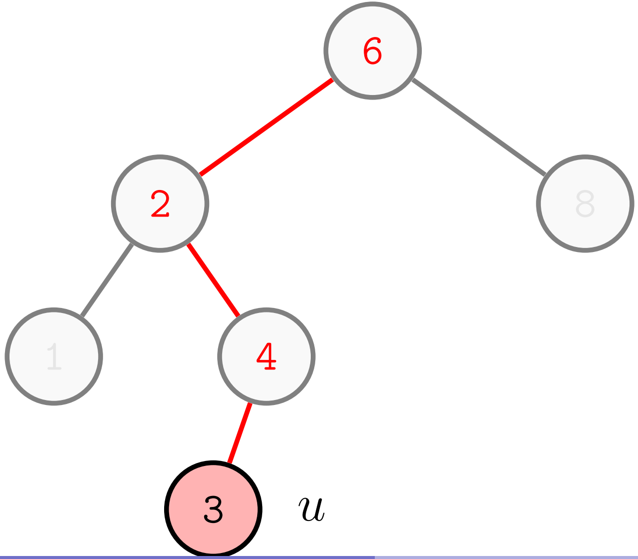
Ricerca – esempio



Valore cercato: 3

- $u =$ 6
- $3 < 6$; $u =$ 2 (Sinistra)
- $3 > 2$; $u =$ 4 (Destra)

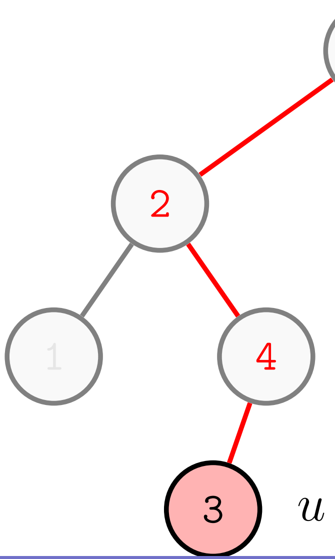
Ricerca – esempio



Valore cercato: 3

- $u =$ 6
- $3 < 6$; $u =$ 2 (Sinistra)
- $3 > 2$; $u =$ 4 (Destra)
- $3 < 4$; $u =$ 3 (Sinistra)

Ricerca – esempio



Valore cercato: 3

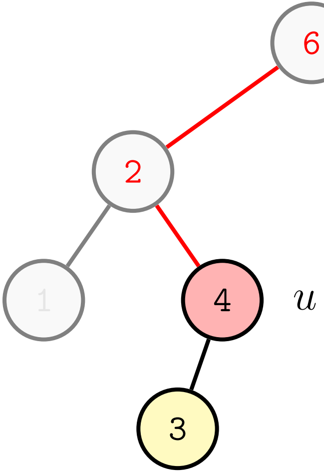
- $u =$ 6
- $3 < 6$; $u =$ 2 (Sinistra)
- $3 > 2$; $u =$ 4 (Destra)
- $3 < 4$; $u =$ 3 (Sinistra)
- $3 = 3$; Trovato

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



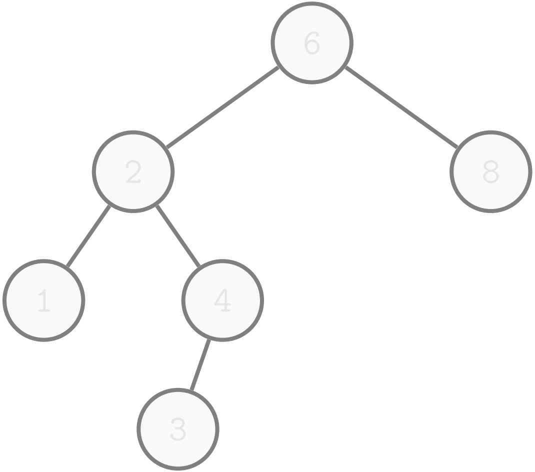
Tree lookupNode(Tree T, Item k)

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

Valore cercato: 5

- $u = 6$
- $5 < 6;\ u = 2$
- $5 > 2;\ u = 4$

Ricerca – esempio



Tree lookupNode($\text{Tree}\ T,\ \text{Item}\ k$)

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

Valore cercato: 5

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

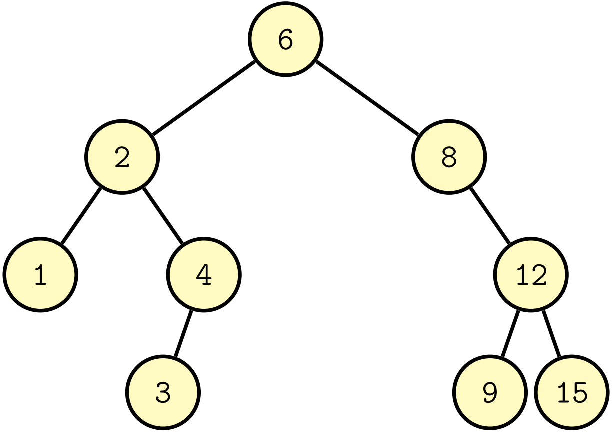
Ricerca – Implementazione

Ricorsiva

TREE lookupNode(TREE T, ITEM k)

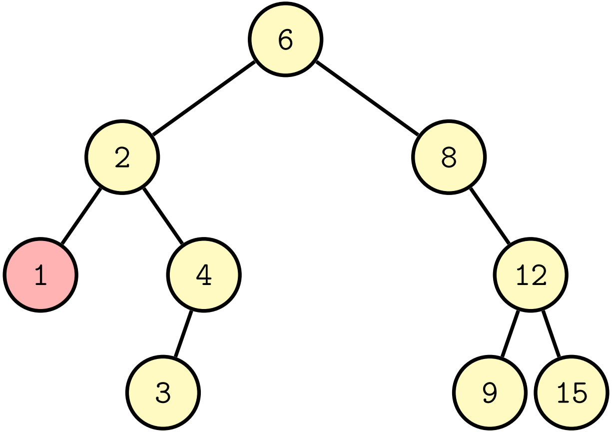
if $T == \textbf{nil}$ **or** $T.key == k$ **then**
  return T
else
  return lookupNode(iif($k < T.key$, $T.left$, $T.right$), k)

Minimo-massimo



- `min` albero radice 6 ?

Minimo-massimo

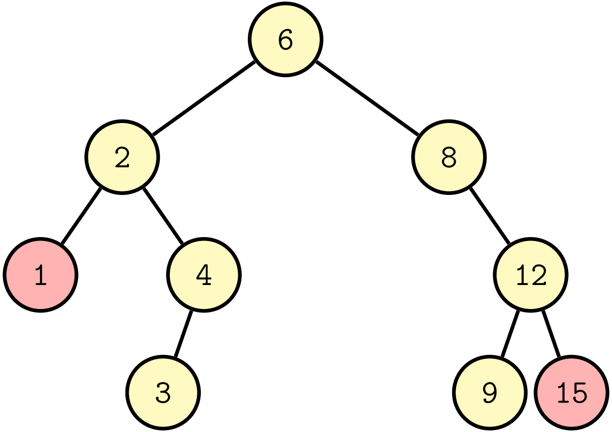


- `min` albero radice 6 ?

  1

- `max` albero radice 6 ?

Minimo-massimo



- `min` albero radice 6 ?

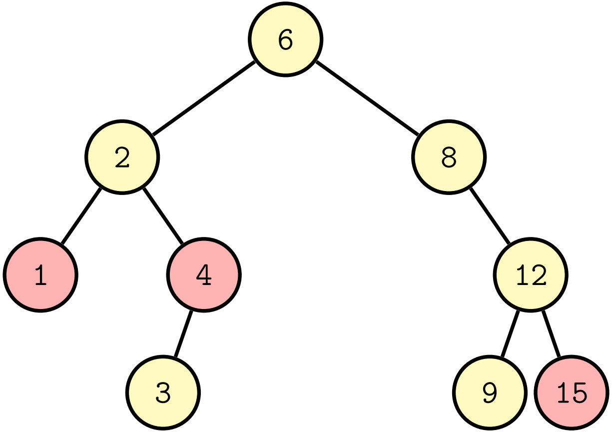
  1

- `max` albero radice 6 ?

  15

- `max` albero radice 2 ?

Minimo-massimo



- `min` albero radice 6 ?

  1

- `max` albero radice 6 ?

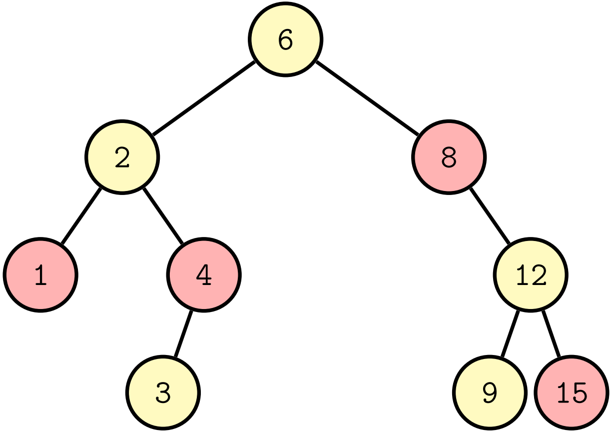
  15

- `max` albero radice 2 ?

  4

- `min` albero radice 8 ?

Minimo-massimo



- `min` albero radice 6 ?

  1

- `max` albero radice 6 ?

  15

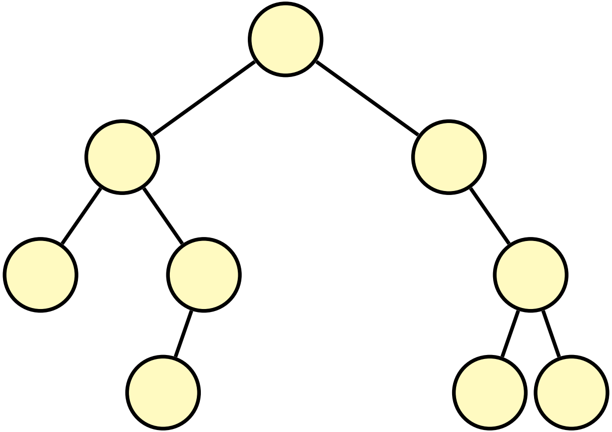
- `max` albero radice 2 ?

  4

- `min` albero radice 8 ?

  8

Minimo-massimo



Tree min(**Tree** T)

Tree $u = T$
while $u.\mathit{left} \neq$ **nil do**
    $u = u.\mathit{left}$
return u

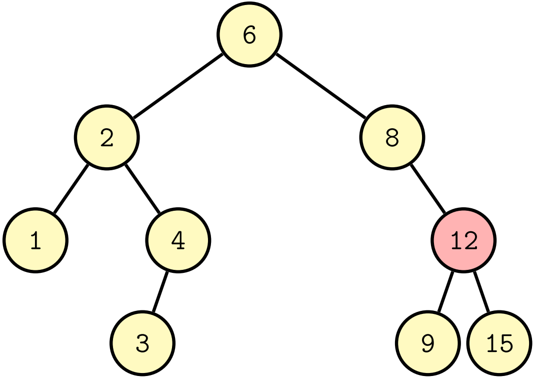
Tree max(**Tree** T)

Tree $u = T$
while $u.\mathit{right} \neq$ **nil do**
    $u = u.\mathit{right}$
return u

Successore-predecessore – Esempio 1

Definizione

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

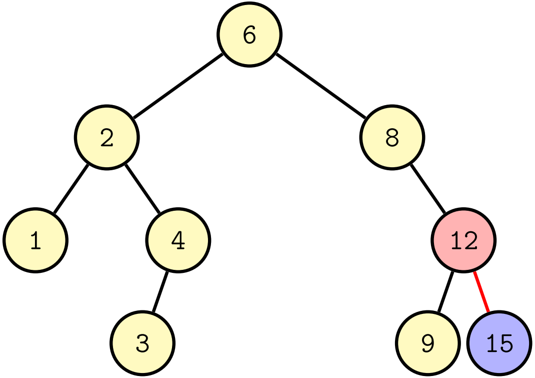


Successore di 12 ?

Successore-predecessore – Esempio 1

Definizione

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

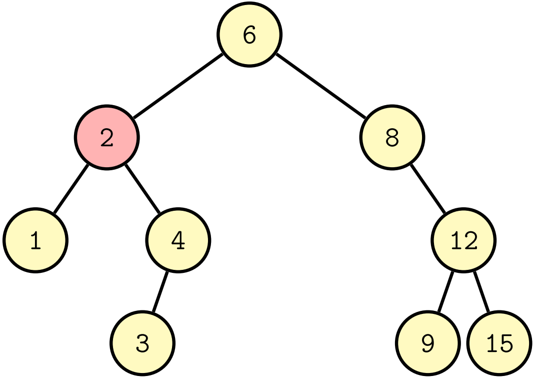


Successore di 12 ? 15

Successore-predecessore – Esempio 2

Definizione

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

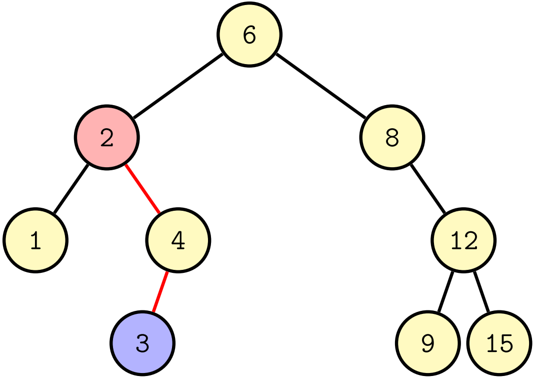


Successore di 2 ?

Successore-predecessore – Esempio 2

Definizione

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

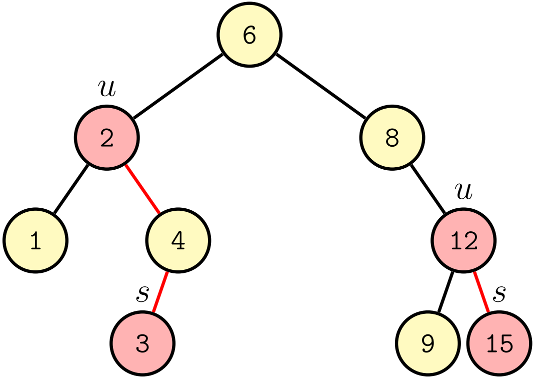


Successore di 2 ? 3

Successore-predecessore – Caso 1

Definizione

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

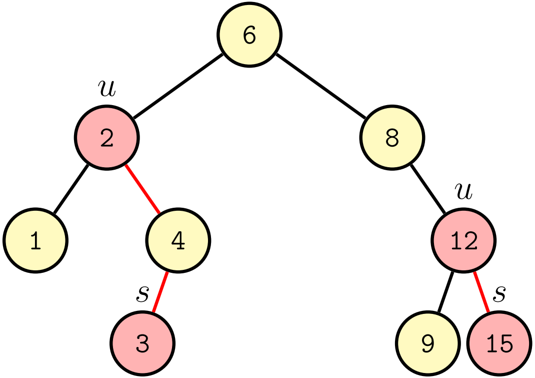


Successore di u?

Successore-predecessore – Caso 1

Definizione

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



Successore di u?

Caso 1

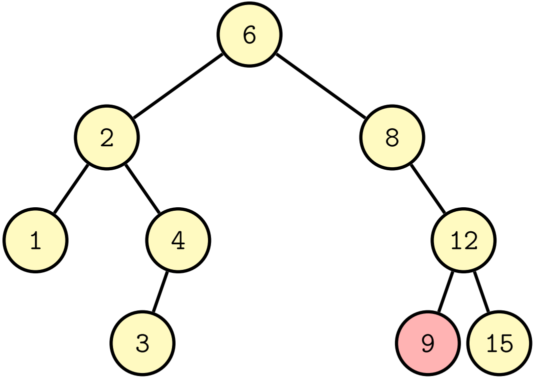
u ha figlio destro

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

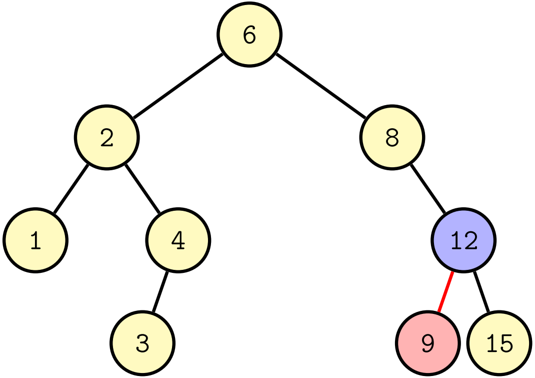


Successore di 9 ?

Successore-predecessore – Esempio 3

Definizione

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

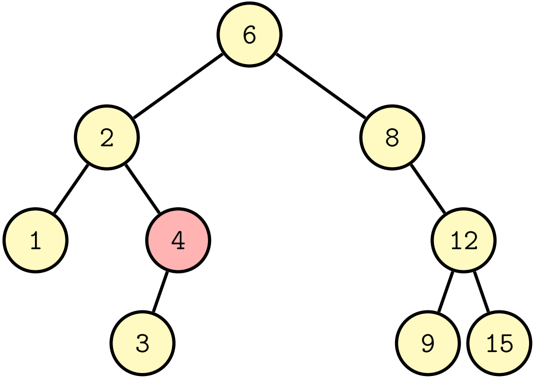


Successore di 9 ? 12

Successore-predecessore – nodo 4

Definizione

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

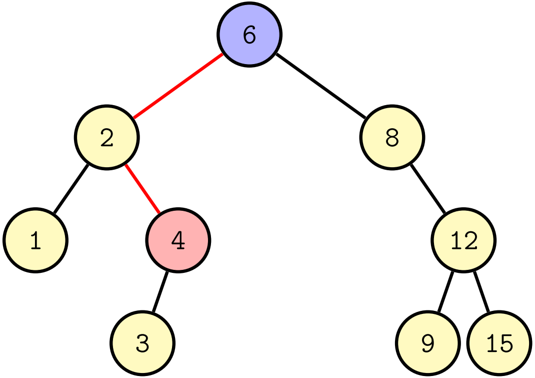


Successore di 4 ?

Successore-predecessore – nodo 4

Definizione

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

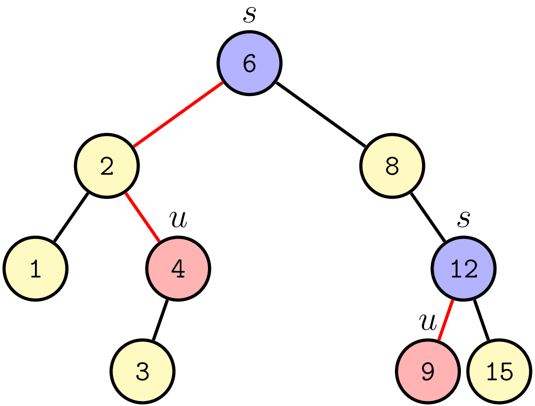


Successore di 4 ? 6

Successore-predecessore – Caso 2

Definizione

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

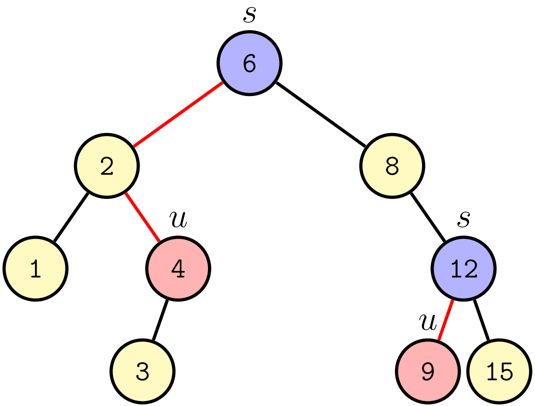


Successore di u?

Successore-predecessore – Caso 2

Definizione

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



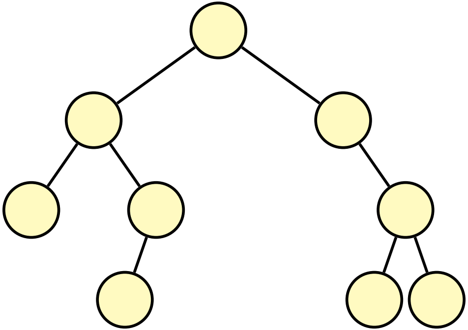
Successore di u?

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 == nil then
    return t
if t.right ≠ nil then                          % Caso 1
    return min(t.right)
else                                           % Caso 2
    TREE p = t.parent
    while p ≠ nil and t == p.right do
        t = p
        p = p.parent
    return p
```

Successore-predecessore – Implementazione

Tree predecessorNode(Tree t)

```
if t == nil then
    return t
if t.left ≠ nil then                          % Caso 1
    return max(t.left)
else                                          % Caso 2
    Tree p = t.parent
    while p ≠ nil and t == p.left do
        t = p
        p = p.parent
    return p
```

Tree successorNode(Tree t)

```
if t == nil then
    return t
if t.right ≠ nil then                         % Caso 1
    return min(t.right)
else                                          % Caso 2
    Tree p = t.parent
    while p ≠ nil and t == p.right do
        t = p
        p = p.parent
    return p
```

Per passare da successore a predecessore

- *right* diventa *left*
- min diventa max

Inserimento – insertNode()

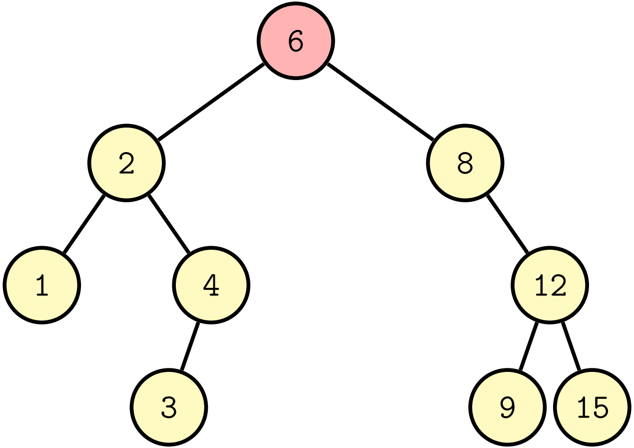
Tree insertNode(Tree T, Item k, Item v)

- 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 $T ==$ **nil**, restituisce il primo nodo dell'albero.
- Altrimenti, restituisce T inalterato

Implementazione dizionario

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

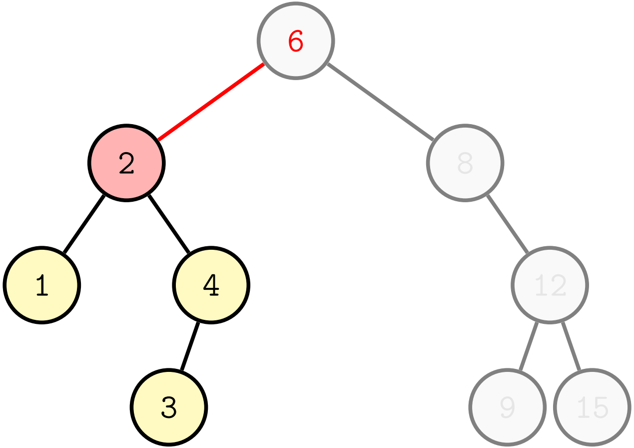
Inserimento – esempio



Valore da inserire: 5

- $u =$ 6

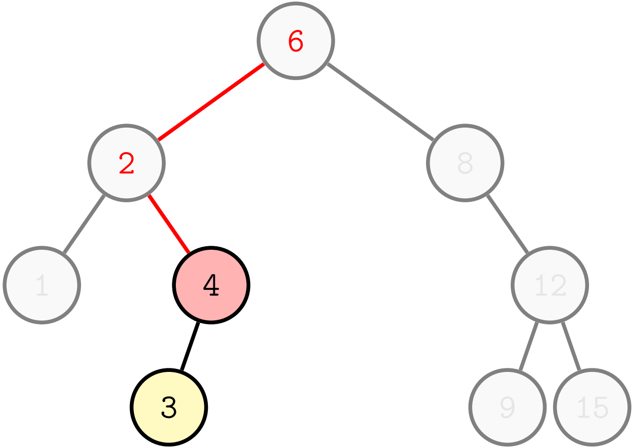
Inserimento – esempio



Valore da inserire: 5

- $u =$ 6
- $5 < 6$; $u =$ 2 (Sinistra)

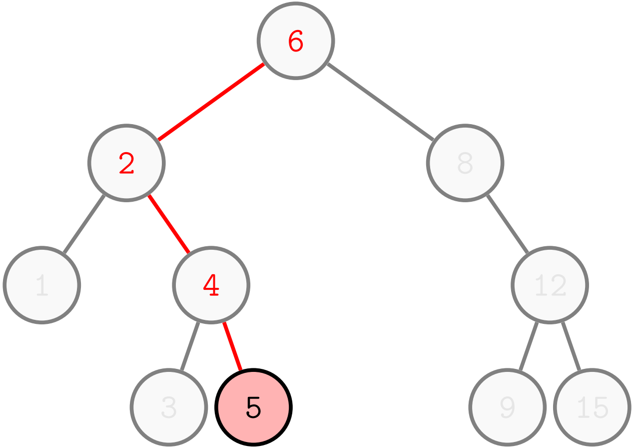
Inserimento – esempio



Valore da inserire: 5

- $u =$ 6
- $5 < 6$; $u =$ 2 (Sinistra)
- $5 > 2$; $u =$ 4 (Destra)

Inserimento – esempio



Valore da inserire: 5

- u = 6
- $5 < 6$; u = 2 (Sinistra)
- $5 > 2$; u = 4 (Destra)
- $5 > 4$; u = **nil** (Destra)
  Inserito

Inserimento – implementazione

```
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
    u.value = v                                       % Chiave già presente
else
    TREE new = Tree(k, v)                             % Crea un nodo coppia chiave-valore
    link(p, new, k)
    if p == 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 ≠ nil then
    u.parent = p                              % Registrazione padre
if p ≠ nil then
    if k < p.key then  p.left = u             % Attaccato come figlio sinistro
                 else  p.right = u            % Attaccato come figlio destro
```

Cancellazione

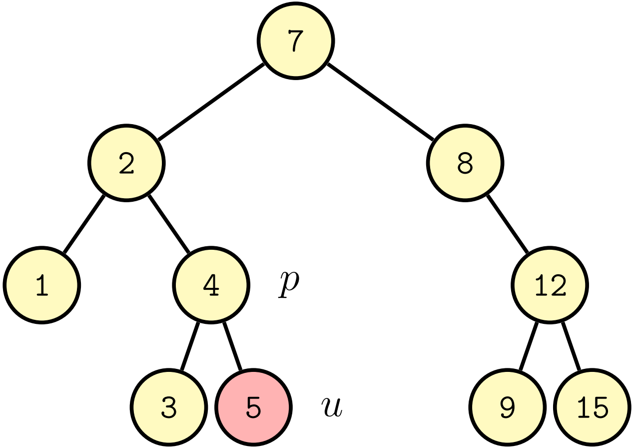
Tree removeNode(Tree T, Item k)

- 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

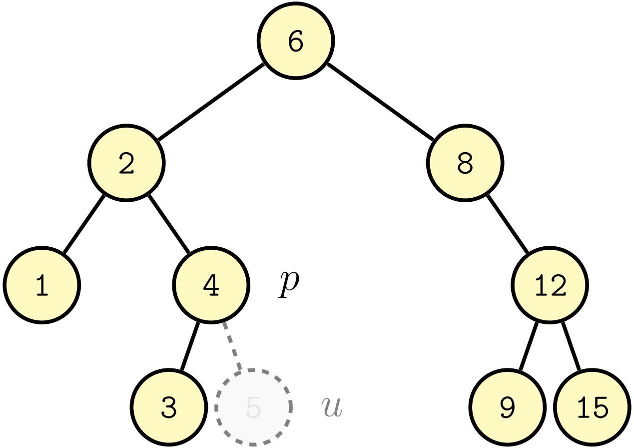
Il nodo da eliminare u non ha figli

Semplicemente si elimina!

Esempio

- Eliminazione 5

Cancellazione



Caso 1

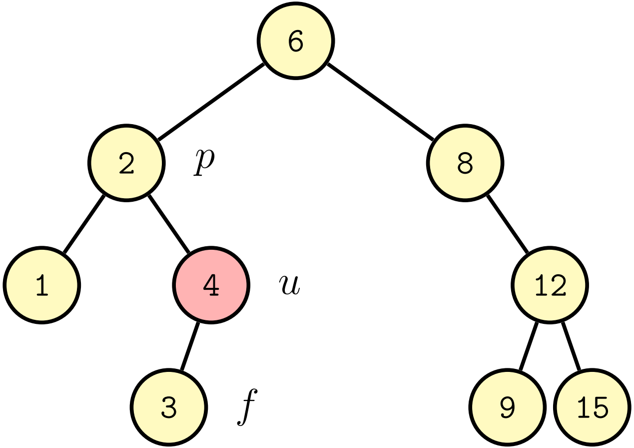
Il nodo da eliminare u non ha figli

Semplicemente si elimina!

Esempio

- Eliminazione 5

Cancellazione



Caso 2

Il nodo da eliminare u ha un solo figlio f

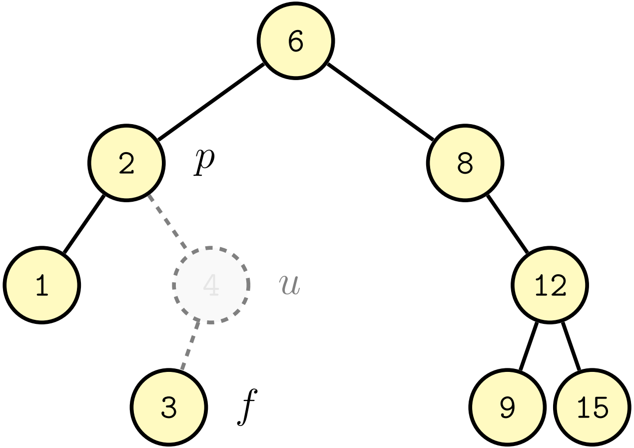
Si elimina u

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

Esempio

- Eliminazione 4

Cancellazione



Caso 2

Il nodo da eliminare u ha un solo figlio f

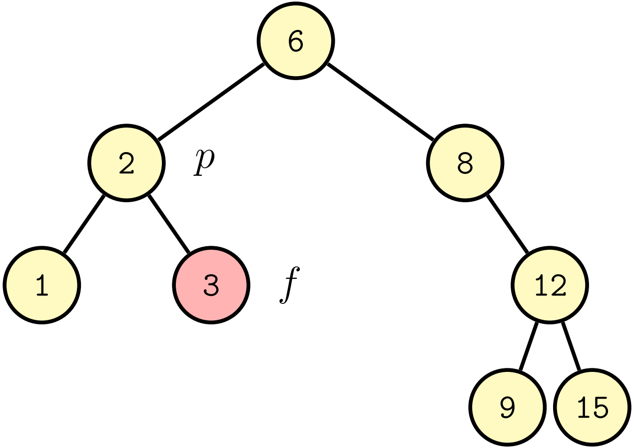
Si elimina u

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

Esempio

- Eliminazione 4

Cancellazione



Caso 2

Il nodo da eliminare u ha un solo figlio f

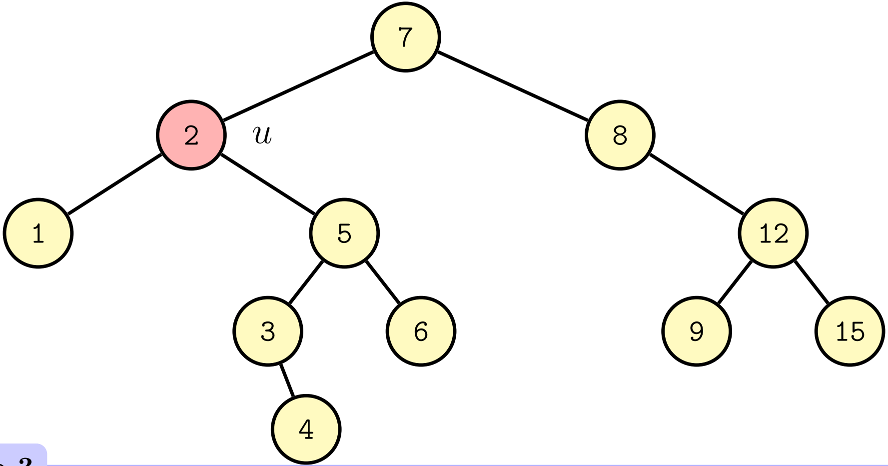
Si elimina u

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

Esempio

- Eliminazione 4

Cancellazione

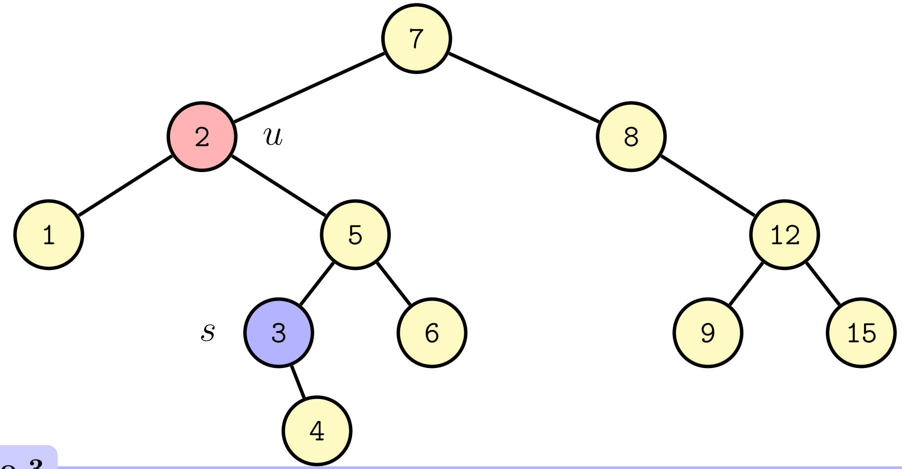


Caso 3

Il nodo da eliminare u ha due figli

- Eliminazione 2

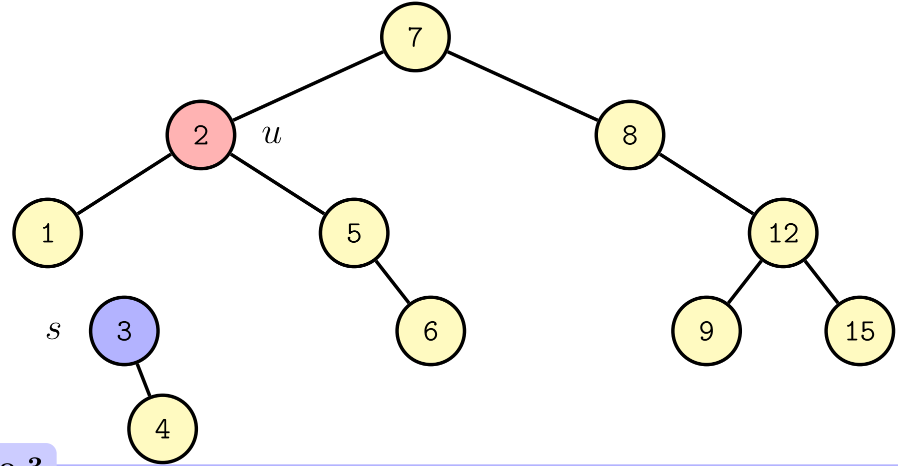
Cancellazione



Caso 3

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

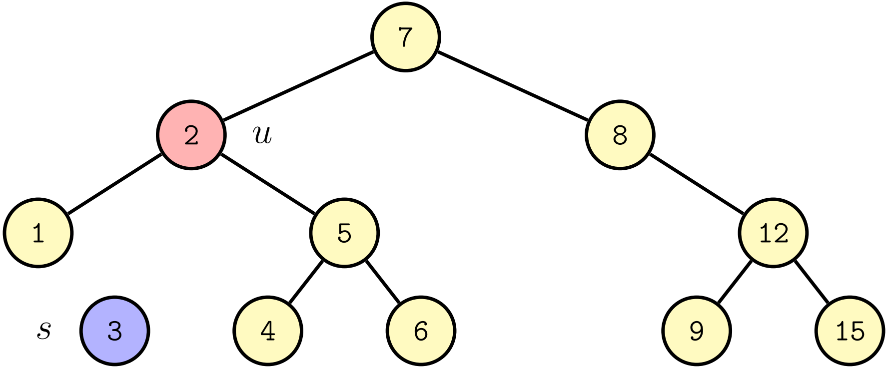
Cancellazione



Caso 3

- Si "stacca" il successore

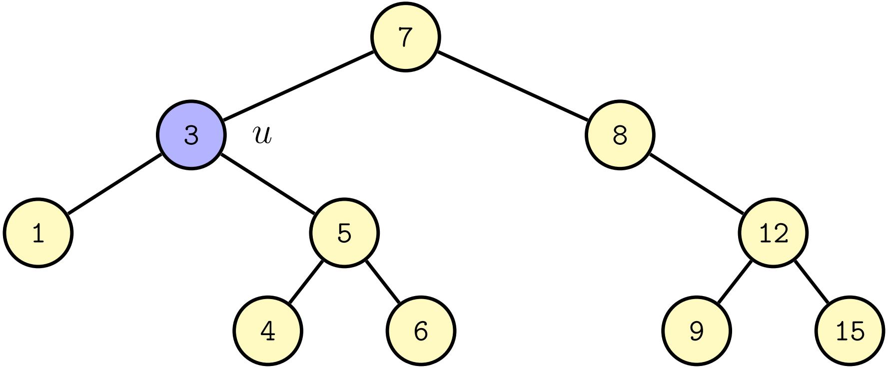
Cancellazione



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 = lookupNode(T, k)
if u ≠ nil then
    if u.left == nil and u.right == nil then          % Caso 1
        link(u.parent, nil, k)
        delete u
    else if u.left ≠ nil and u.right ≠ nil then       % Caso 3
        [...]
    else                                              % Caso 2
        [...]
return T
```

Cancellazione – Implementazione

```
TREE removeNode(TREE T, ITEM k)
TREE t
TREE u = lookupNode(T, k)
if u ≠ nil then
    if u.left == nil and u.right == nil then          % Caso 1
        [...]
    else if u.left ≠ nil and u.right ≠ nil then       % Caso 3
        TREE s = successorNode()
        link(s.parent, s.right, s.key)
        u.key = s.key
        u.value = s.value
        delete s
    else                                              % Caso 2
        [...]
return T
```

Cancellazione – Implementazione

```
TREE removeNode(TREE T, ITEM k)
TREE t
TREE u = lookupNode(T, k)
if u ≠ nil then
    if u.left == nil and u.right == nil then          % Caso 1
        [...]
    else if u.left ≠ nil and u.right ≠ nil then       % Caso 3
        [...]
    else if u.left ≠ nil and u.right == nil then      % Caso 2
        link(u.parent, u.left, k)
        if u.parent = nil then
            T = u.left
    else
        link(u.parent, u.right, k)
        if u.parent = nil then
            T = u.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

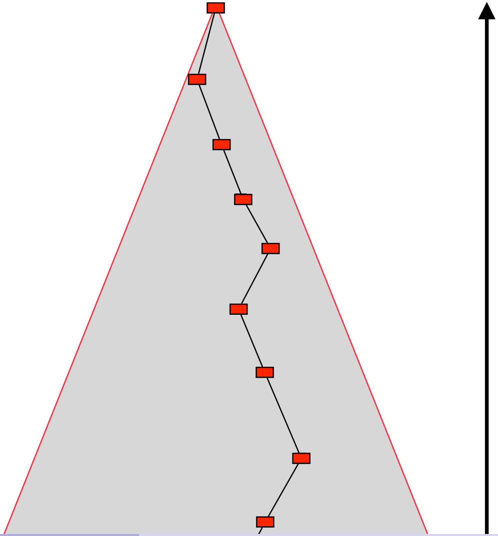
Tutte le operazioni sono confinate ai nodi posizionati lungo un cammino semplice dalla radice ad una foglia

h = 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

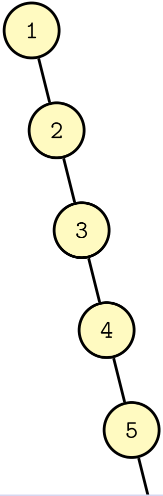
h = Altezza dell'albero

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

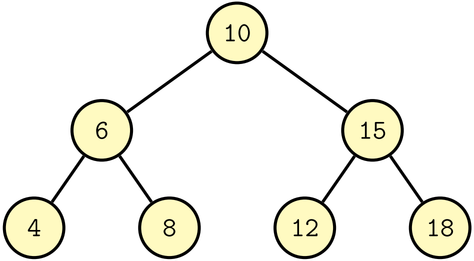
h = Altezza dell'albero

Tempo di ricerca: $O(h)$

Domande

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

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



Nothing to see here, move along!

La seconda parte di queste slide verrà revisionata prima della prossima lezione.

Algoritmi e Strutture Dati

Alberi binari di ricerca bilanciati

Alberto Montresor

Università di Trento

2020/10/22

Altezza degli alberi binari di ricerca

Altezza ABR, caso pessimo

- $O(n)$

Altezza ABR, caso medio

- Caso "semplice": inserimenti in ordine casuale
  - È 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

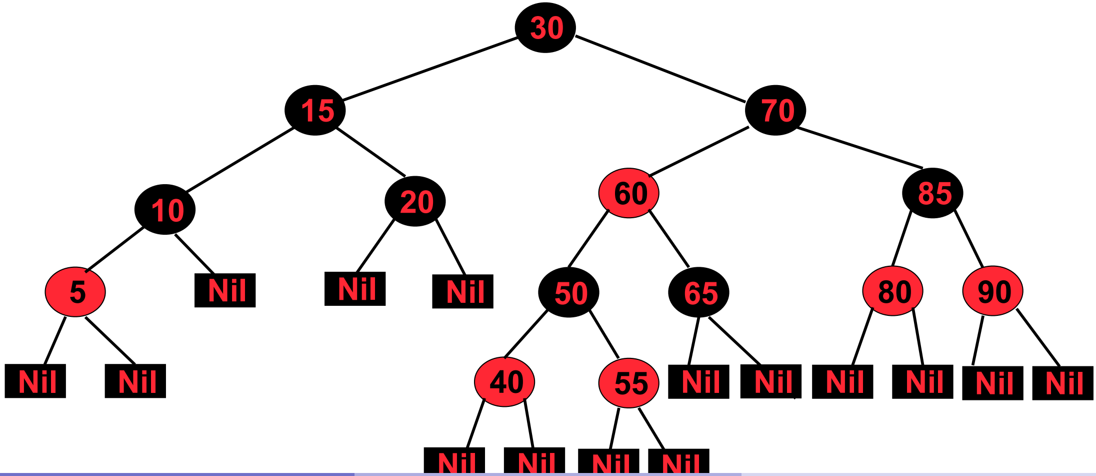
Alberi Red-Black (Guibas and Sedgewick, 1978)

Sono alberi binari 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**
- Vengono rispettati i seguenti 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 suo sottoalbero ha lo stesso numero di nodi neri

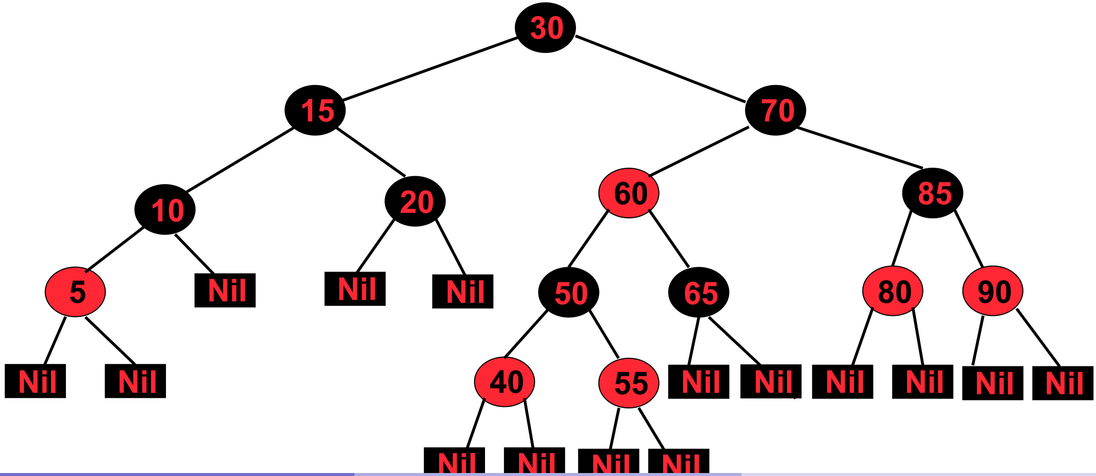
Esempi

1. La radice è nera



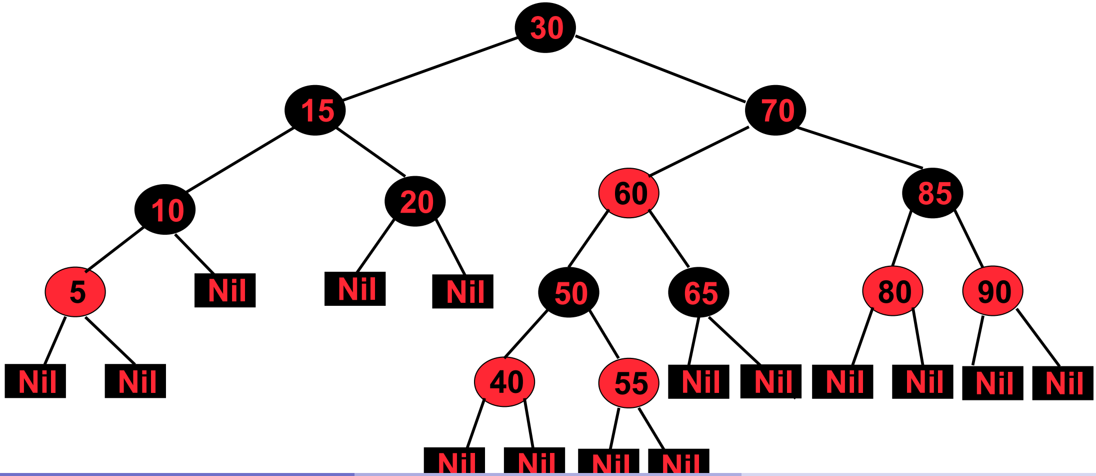
Esempi

2. Tutte le foglie sono nere



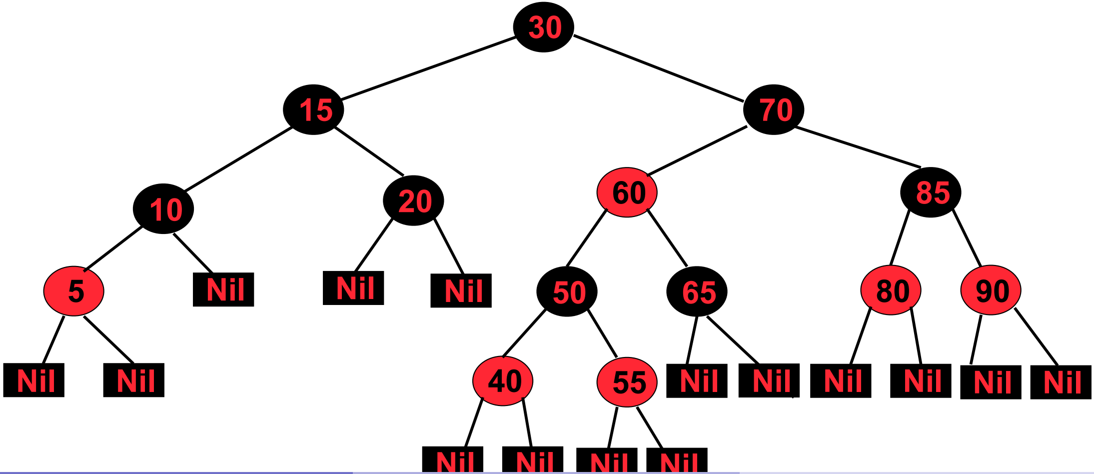
Esempi

3. Entrambi i figli di un nodo rosso sono neri



Esempi

❹ Ogni cammino semplice da un nodo u ad una delle foglie contenute nel suo sottoalbero ha lo stesso numero di nodi neri



Reality check

Java TreeMap, Java TreeSet

https://docs.oracle.com/javase/8/docs/api/java/util/TreeMap.html

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

C++ STL

https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.1/stl__tree_8h-source.html

```
namespace std
{
  // Red-black tree class, designed for use in implementing STL
  // associative containers (set, multiset, map, and multimap). The
  // insertion and deletion algorithms are based on those in Cormen,
  // Leiserson, and Rivest, Introduction to Algorithms (MIT Press,
  // 1990), except that
```

Reality check

Linux

https://github.com/torvalds/linux/blob/master/Documentation/core-api/rbtree.rst

To quote Linux Weekly News:

> There are a number of red-black trees in use in the kernel. The deadline and CFQ I/O schedulers employ rbtrees to track requests; the packet CD/DVD driver does the same. The high-resolution timer code uses an rbtree to organize outstanding timer requests. The ext3 filesystem tracks directory entries in a red-black tree. Virtual memory areas (VMAs) are tracked with red-black trees, as are epoll file descriptors, cryptographic keys, and network packets in the "hierarchical token bucket" scheduler.

https://github.com/torvalds/linux/blob/master/include/linux/rbtree.h

master | linux / include / linux / **rbtree.h**

161 lines (130 sloc) | 5.1 KB

```
/* SPDX-License-Identifier: GPL-2.0-or-later */
/*
  Red Black Trees
```

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 avere accesso al colore di entrambi i figli, evitare di dover gestire casi particolari quando uno dei due è **nil**.
- Al posto di un puntatore **nil**, si usa un puntatore ad un nodo **Nil** con colore nero
- Ne esiste solo uno, per risparmiare memoria

Altezza nera

Altezza nera di un nodo v

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

Altezza nera di un albero Red-Black

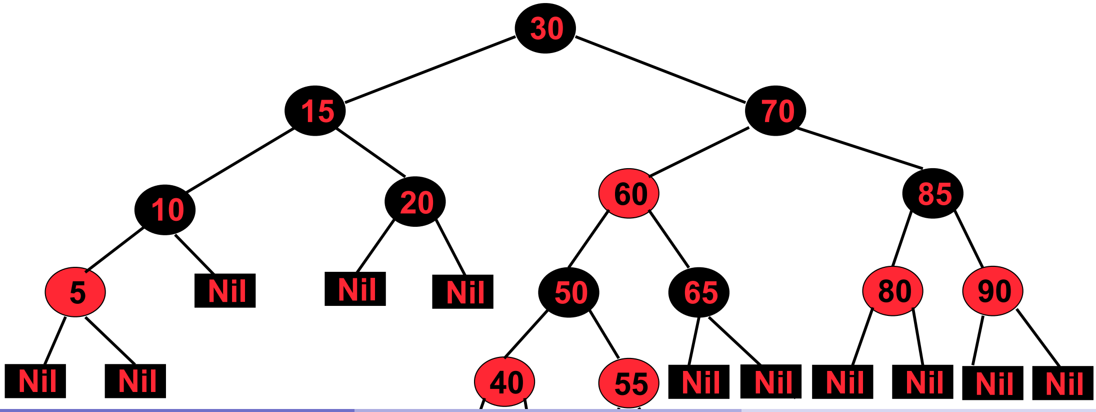
L'altezza nera di un albero Red-Black è pari all'altezza nera della sua radice

---

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

Esempi

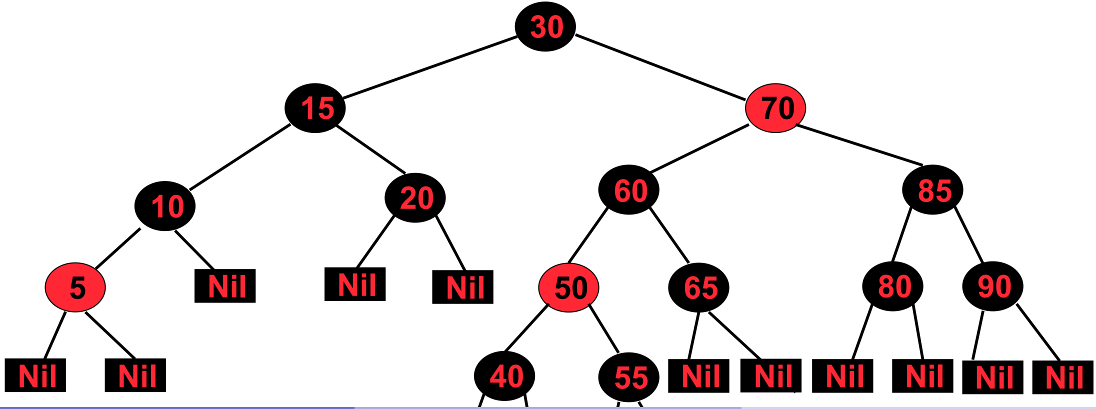
Più colorazioni sono possibili – Versione 1
Altezza nera: $bh(r) = 3$



Esempi

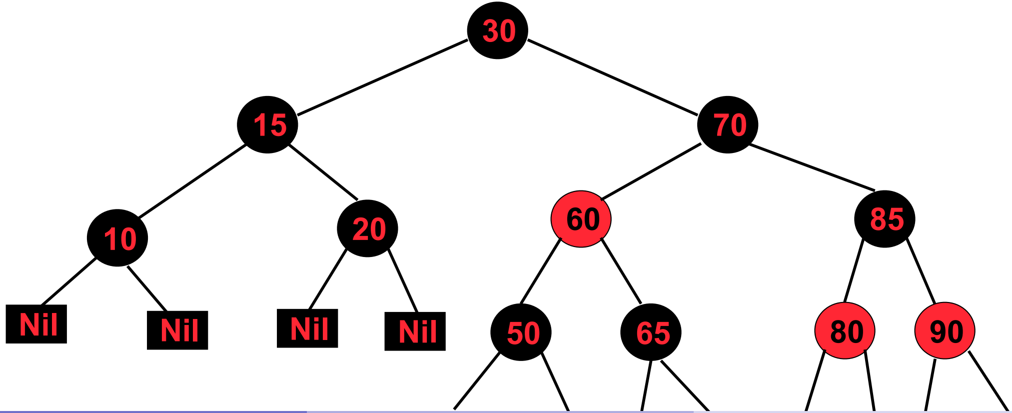
Più colorazioni sono possibili – Versione 2

Altezza nera: $bh(r) = 3$



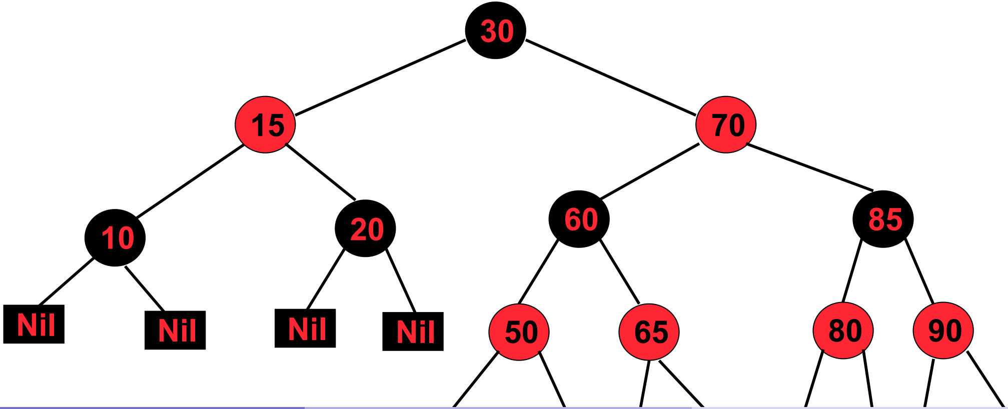
Esempi

Cambiare colorazione può cambiare l'altezza nera
Altezza nera: $bh(r) = 3$



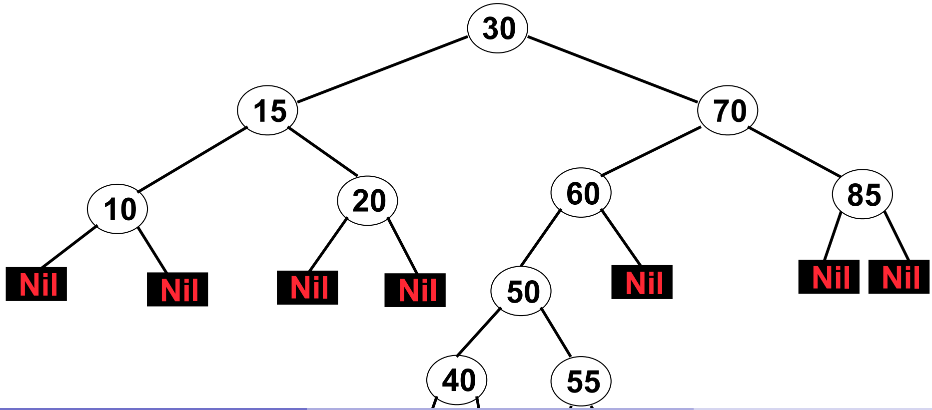
Esempi

Cambiare colorazione può cambiare l'altezza nera
Altezza nera: $bh(r) = 2$



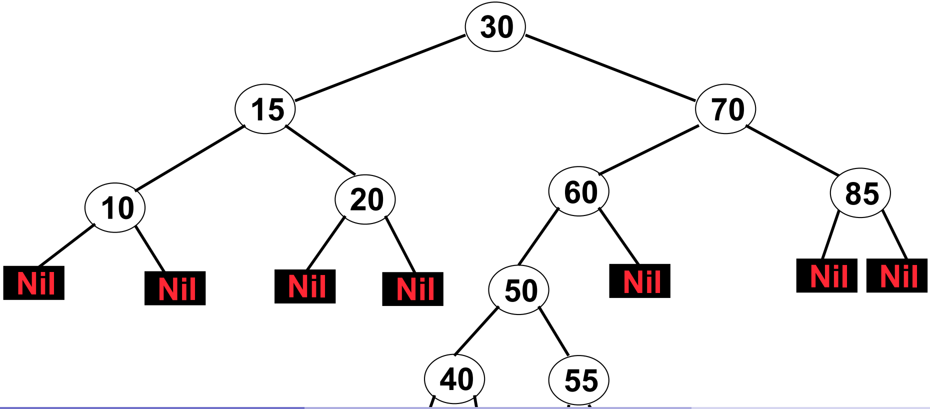
Esempi

Questo albero può essere un albero Red-Black?



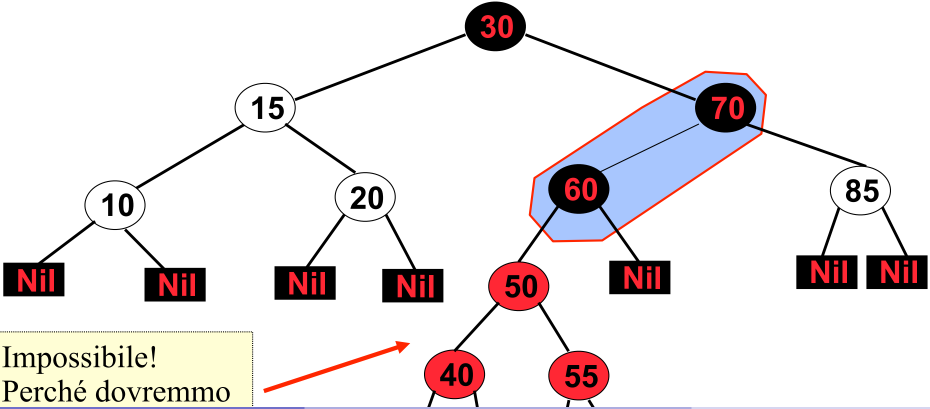
Esempi

Guardando il lato sinistro, $2 \leq bh(r) \leq 3$
Guardando il lato destro, $bh(r) \geq 3$
Quindi bh(r)=3.



Esempi

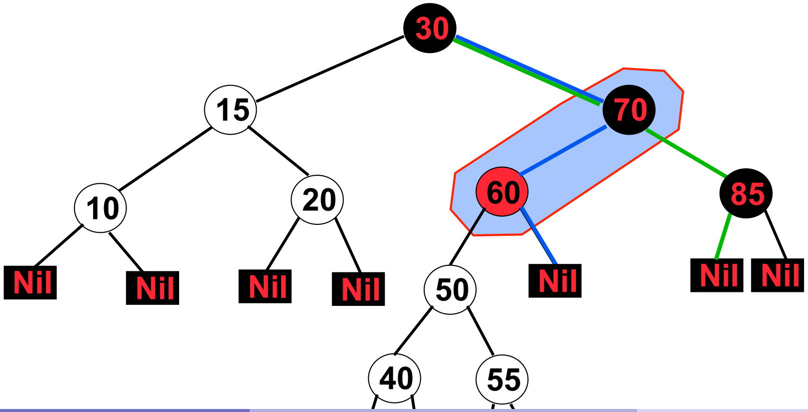
Supponiamo che 60 e 70 siano entrambi neri.
Per rispettare il vincolo ❹, dobbiamo infrangere il vincolo ❸



Impossibile!
Perché dovremmo

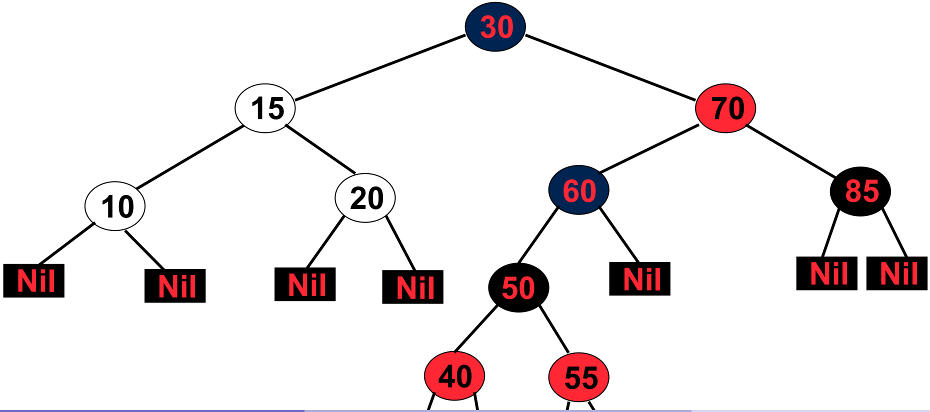
Esempi

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



Esempi

Proviamo a colorare di rosso il nodo 70.
Esistono cammini con 2 nodi neri
Impossibile per il vincolo 4

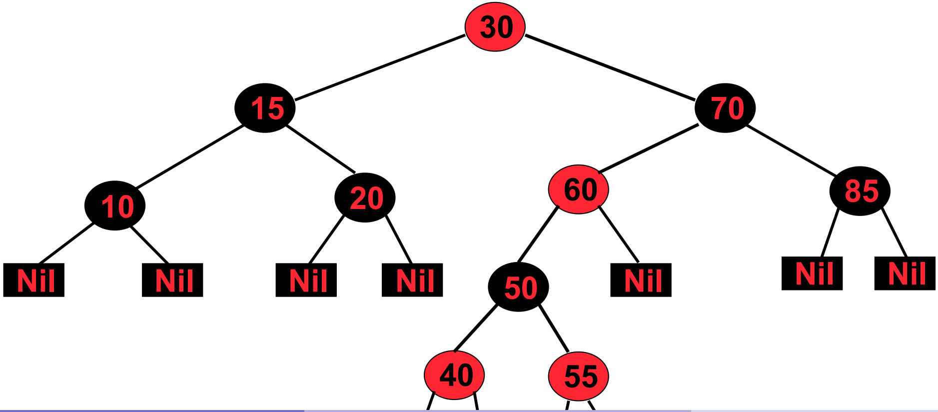


Esempi

Questa è l'ultima possibilità.

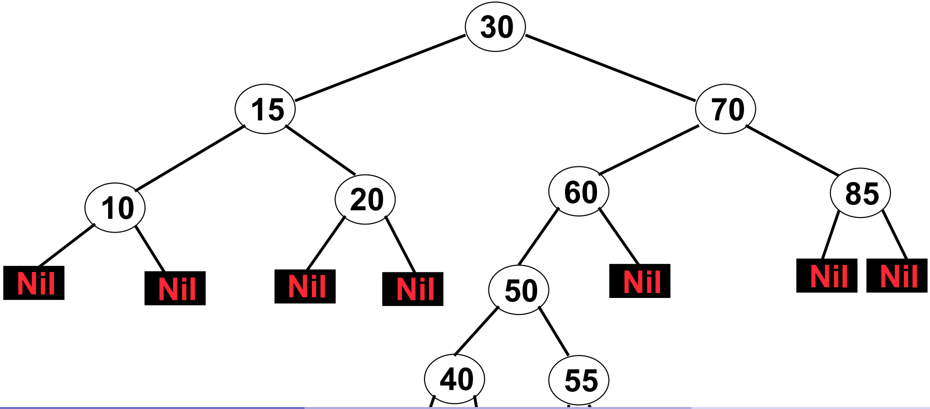
Impossibile perchè non rispetta il vincolo 1

Impossibile perchè non rispetta il vincolo 4



Esempi

Questo albero non può essere un albero Red-Black!



Inserimento

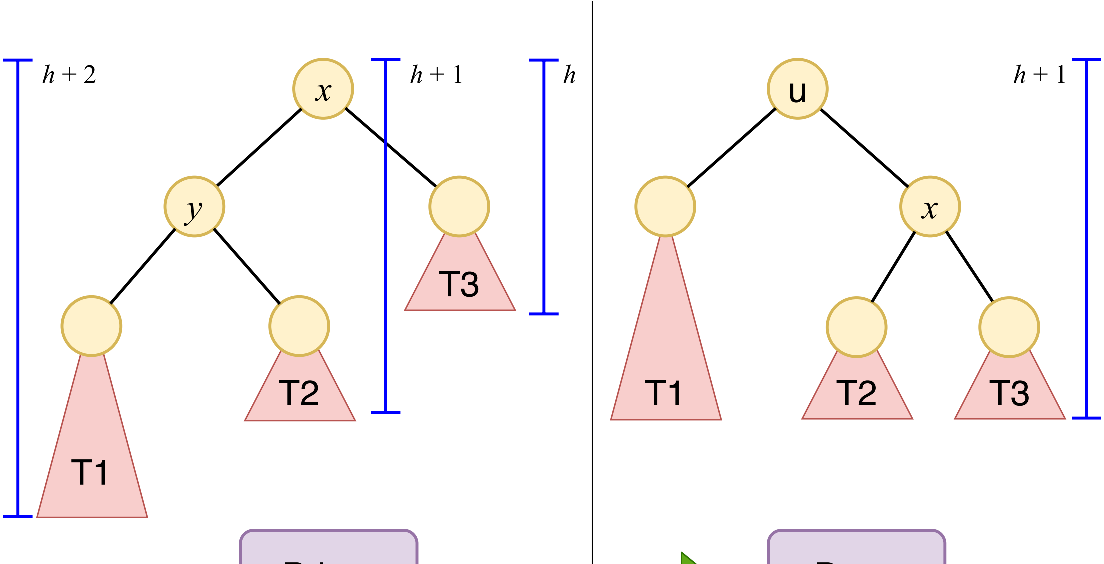
Durante la modifica di un albero Red-Black

- È possibile che le condizioni di bilanciamento risultino violate

Quando i vincoli Red-Black vengono violati si può agire:

- Modificando i colori nella zona della violazione
- Operando dei ribilanciamenti dell'albero tramite rotazioni
  - Rotazione destra
  - Rotazione sinistra

Rotazione



Rotazione a 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** $p.right \leftarrow y$

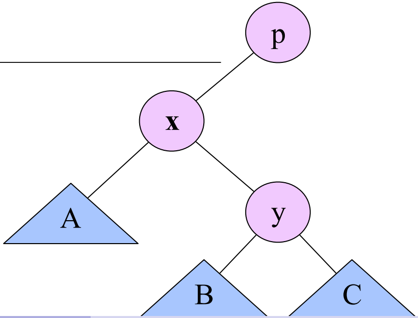
return y

- ## Operazioni

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

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

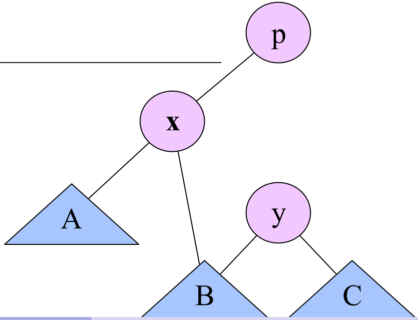
  (3) far diventare *y* figlio di *p*, il vecchio padre di *x*



Rotazione a sinistra

```
TREE rotateLeft(TREE x)
------------------------------------------------------------------
    TREE y ← x.right
    TREE p ← x.parent
(1) x.right ← y.left            % Il sottoalbero B diventa figlio destro di x
(1) if y.left ≠ nil then y.left.parent ← x
(2) y.left ← x                  % x diventa figlio sinistro di y
(2) x.parent ← y
(3) y.parent ← p                % y diventa figlio di p
(3) if p ≠ nil then
        if p.left = x then p.left ← y else p.right ← 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

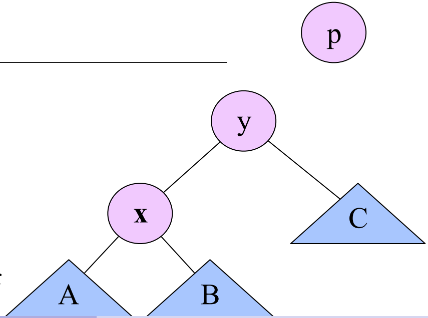


Rotazione a sinistra

TREE rotateLeft(TREE x)

TREE $y \leftarrow x.right$
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

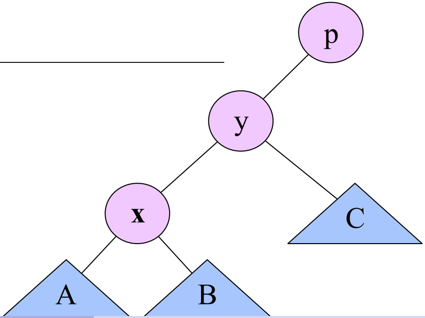


Rotazione a sinistra

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



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?

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

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
    u.value = v                                     % Chiave già presente
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

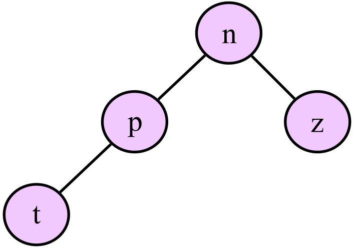
- Ci spostiamo verso l'alto lungo il percorso di inserimento
- Ripristinare il vincolo 3 (figli neri di nodo rossso)
- Spostiamo le violazioni verso l'alto rispettando il vincolo (4) (mantenendo l'altezza nera dell'albero)
- Al termine, coloriamo la radice di nero (vincolo 1)

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

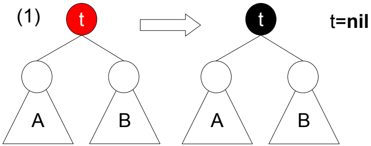


balanceInsert(TREE t)

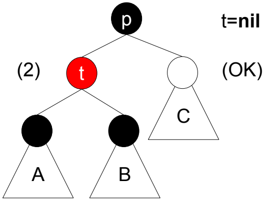
$t.color \leftarrow$ RED
while $t \neq$ **nil do**
  TREE $p \leftarrow t.parent$ % Padre
  TREE $n \leftarrow$ iif($p \neq$ **nil**, $p.parent$, **nil**) % Nonno
  TREE $z \leftarrow$ iif($n =$ **nil**, **nil**, iif($n.left = p$, $n.right$, $n.left$)) % Zio

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

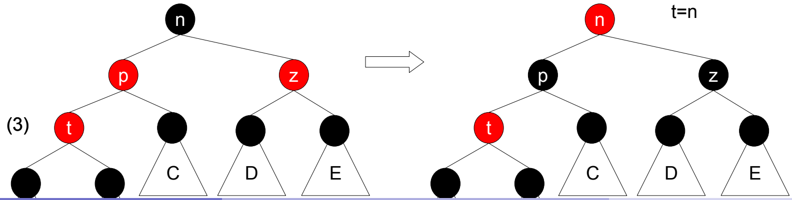
- **Caso 1:**
  - Nuovo nodo *t* non ha padre
  - Primo nodo ad essere inserito o siamo risaliti fino alla radice
  - Si colora *t* di nero
- **Caso 2**
  - Padre *p* di *t* è nero
  - Nessun vincolo violato

$$
\begin{array}{ll}
\textbf{while } t \neq \textbf{nil do} & \\
\quad \textsc{Tree}\ p \leftarrow t.parent & \%\ \text{Padre} \\
\quad \textsc{Tree}\ n \leftarrow \mathsf{iif}(p.parent \neq \textbf{nil}, p.parent, \textbf{nil}) & \%\ \text{Nonno} \\
\quad \textsc{Tree}\ z \leftarrow \mathsf{iif}(n = \textbf{nil}, \textbf{nil}, \mathsf{iif}(n.left = p, right, left)) & \%\ \text{Zio} \\
\quad \textbf{if } p = \textbf{nil then} & \%\ \text{Caso (1)} \\
\quad\quad t.color \leftarrow \text{BLACK} & \\
\quad\quad t \leftarrow \textbf{nil} & \\
\quad \textbf{else if } p.color = \text{BLACK} \textbf{ then} & \%\ \text{Caso (2)}
\end{array}
$$

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

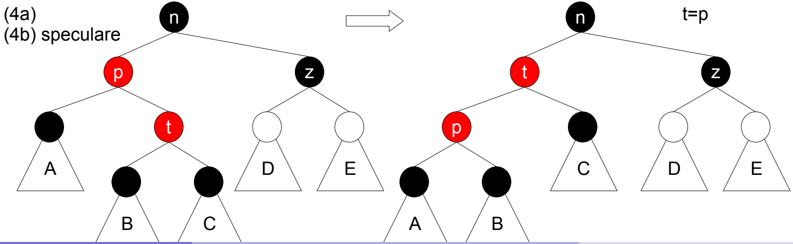
- Se z è rosso, è possibile colorare di nero p, z, e di rosso n.
- Poiché tutti i cammini che passano per z e p passano per n, la lunghezza dei cammini neri non è cambiata.
- Il problema può essere ora sul nonno:
  - violato vincolo (1), ovvero n può essere una radice rossa
  - violato vincolo (3), ovvero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

```
else if z.color = RED then                 % Caso (3)
    p.color ← z.color ← BLACK
    n.color ← RED
    t ← n
```

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

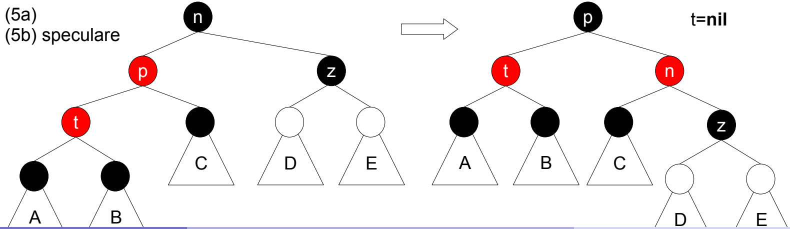
- Si assuma che t sia figlio destro di p e che p sia figlio sinistro di n
- Una rotazione a sinistra a partire dal nodo p scambia i ruoli di t e p ottenendo il caso (5a), dove i nodi rossi in conflitto sul vincolo (3) sono entrambi figli sinistri dei loro padri
- I nodi coinvolti nel cambiamento sono 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 ← rotateLeft(p)
        t ← p
    else if (t = p.left) and (p = n.right) then           % Caso (4.b)
        n.right ← rotateRight(p)
        t ← p
```

Inserimento – 7 casi possibili

- **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



Inserimento – 7 casi possibili

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

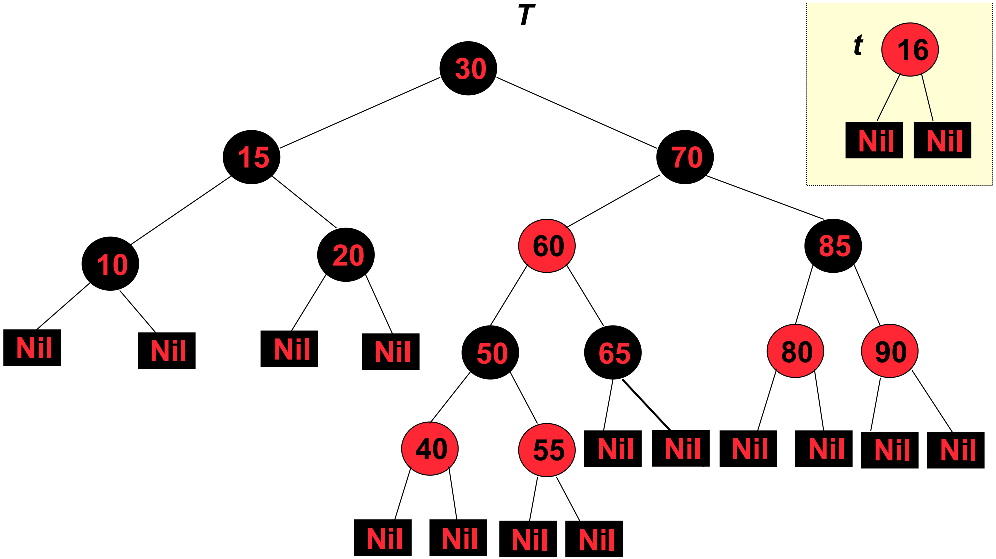
- Si assuma che t sia figlio sinistro di p e p sia figlio sinistro di n
- Una rotazione a destra a partire da n ci porta ad una situazione in cui t e n sono figli di p
- Colorando n di rosso e p di nero ci troviamo in una situazione in cui tutti i vincoli Red-Black sono rispettati
- in particolare, la 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
```

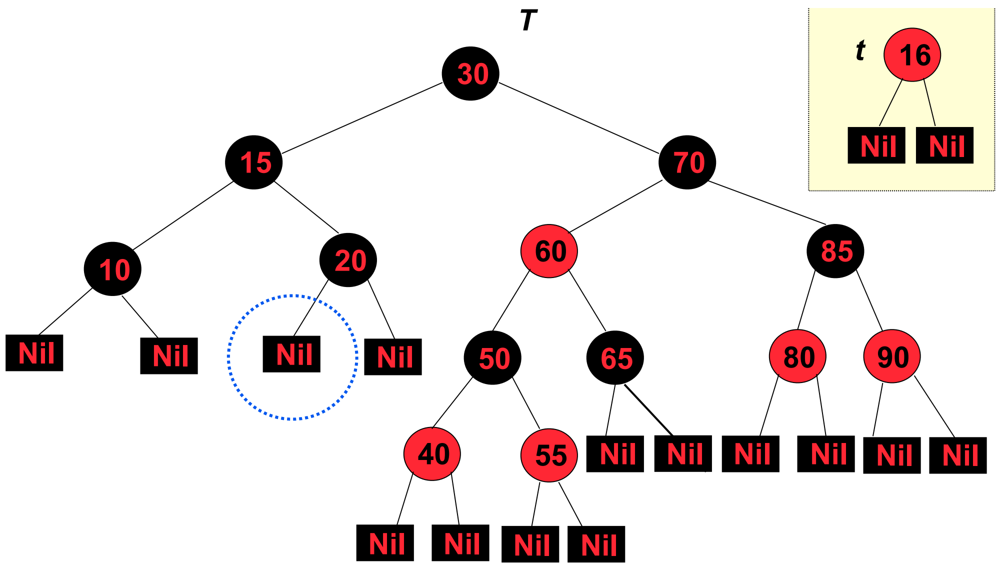
All together, now!

```
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)
```

Inserimento – Esempio

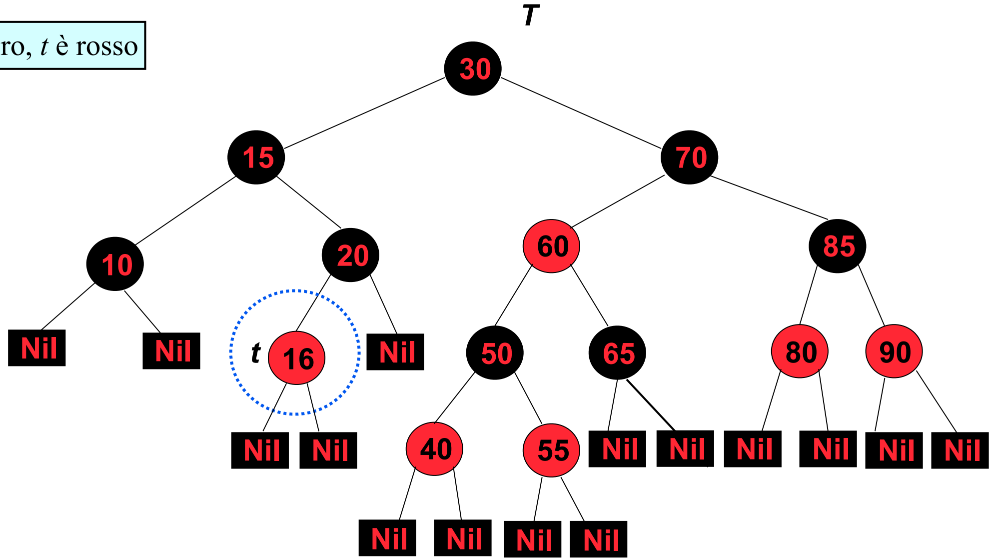


Inserimento – Esempio



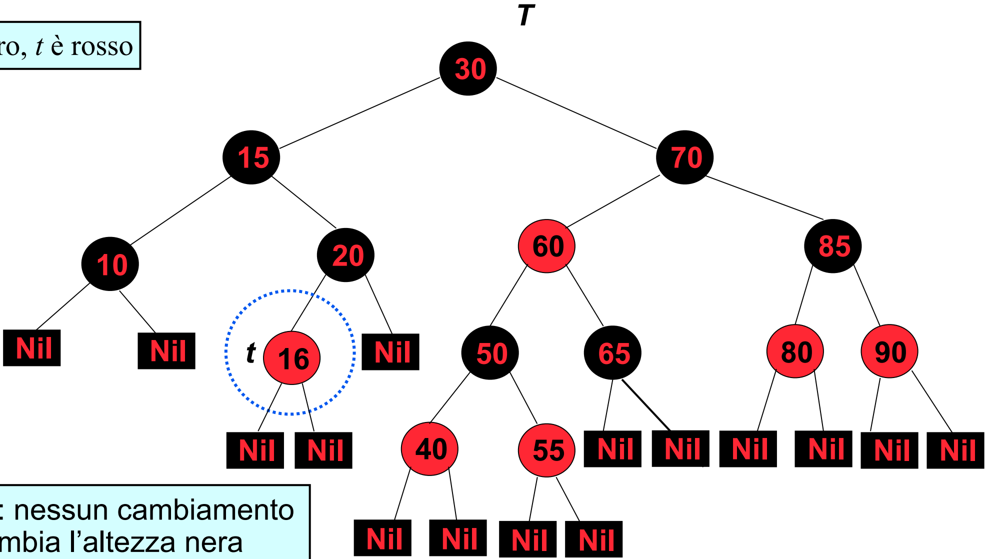
Inserimento – Esempio

p è nero, t è rosso



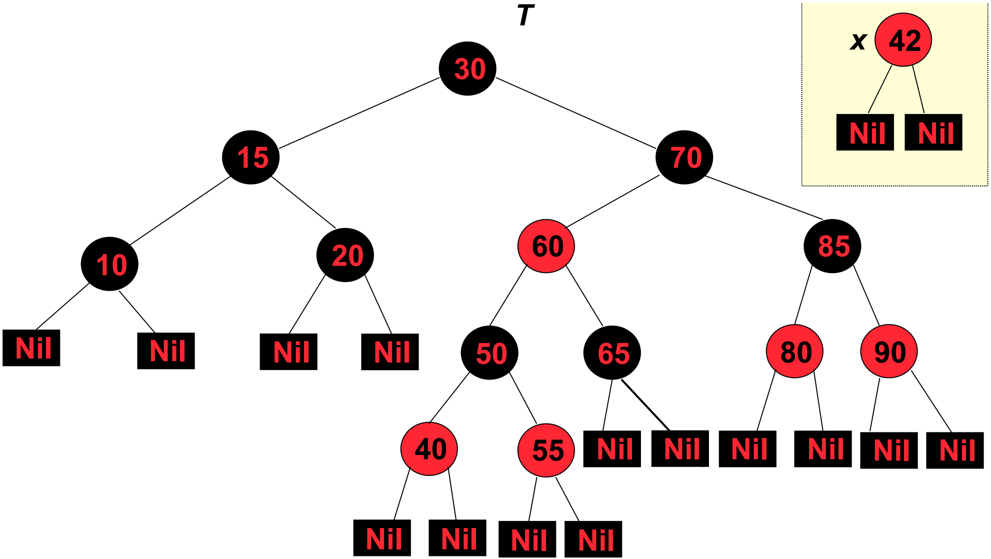
Inserimento – Esempio

p è nero, *t* è rosso

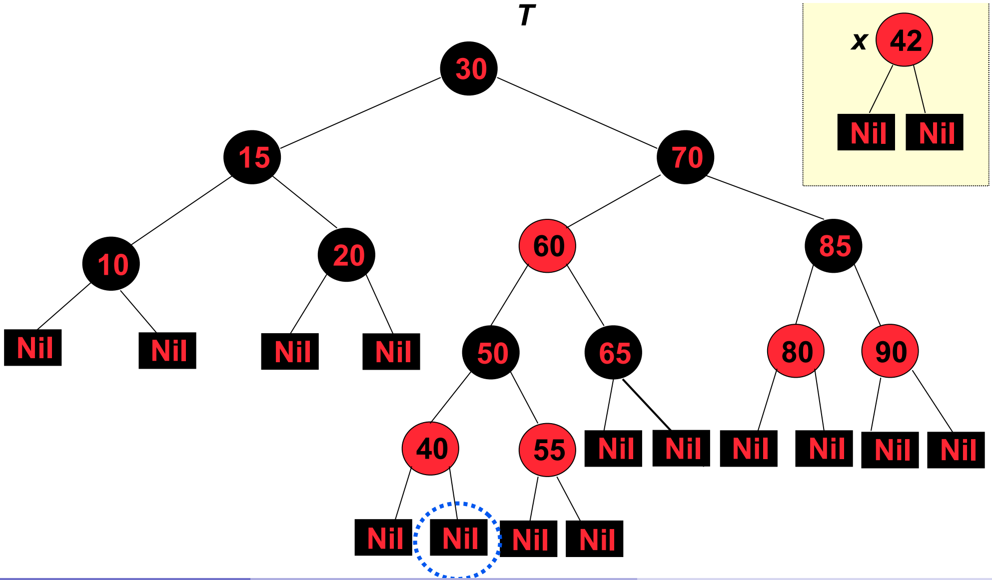


Caso 2: nessun cambiamento
Non cambia l'altezza nera
di nessun nodo!

Inserimento – Esempio

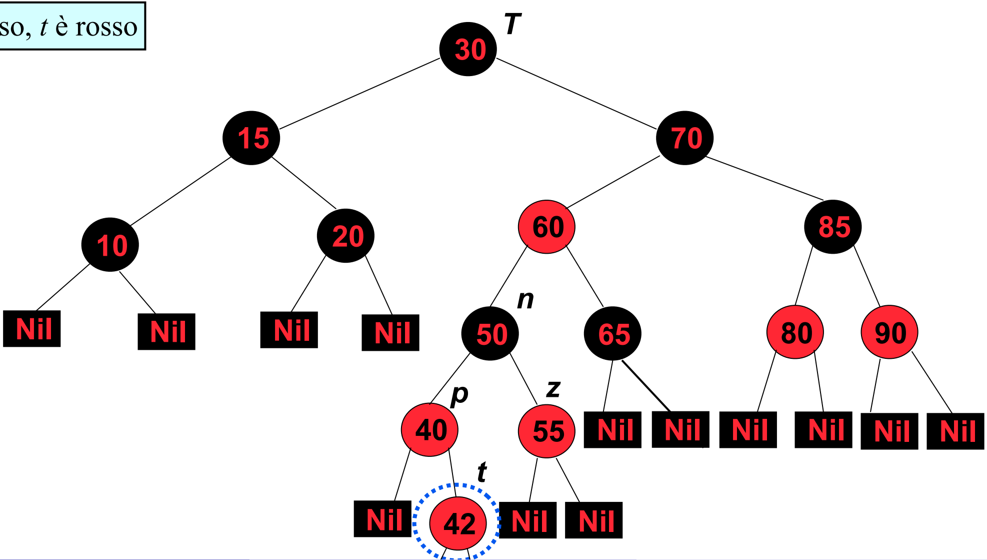


Inserimento – Esempio



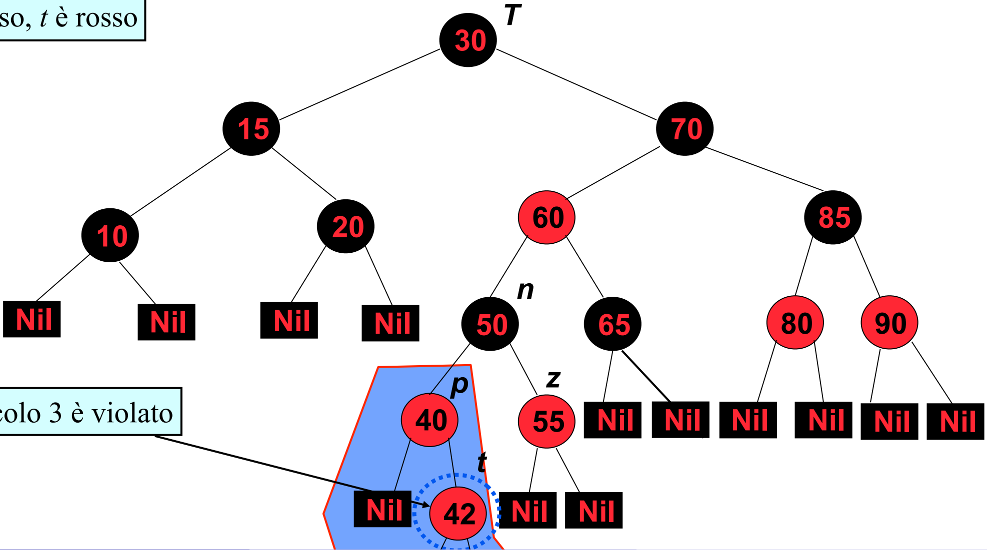
Inserimento – Esempio

p è rosso, t è rosso



Inserimento – Esempio

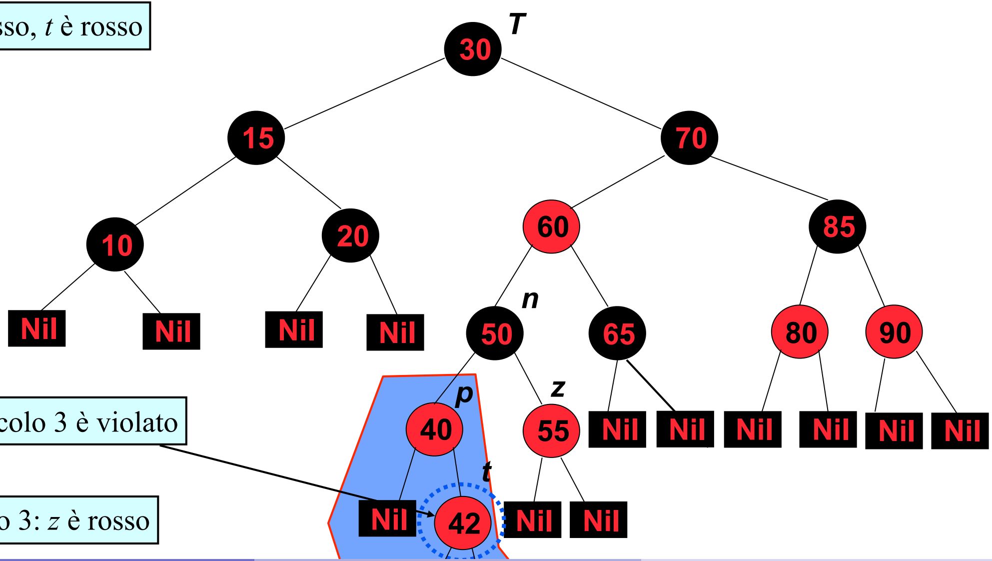
p è rosso, t è rosso



Vincolo 3 è violato

Inserimento – Esempio

p è rosso, t è rosso

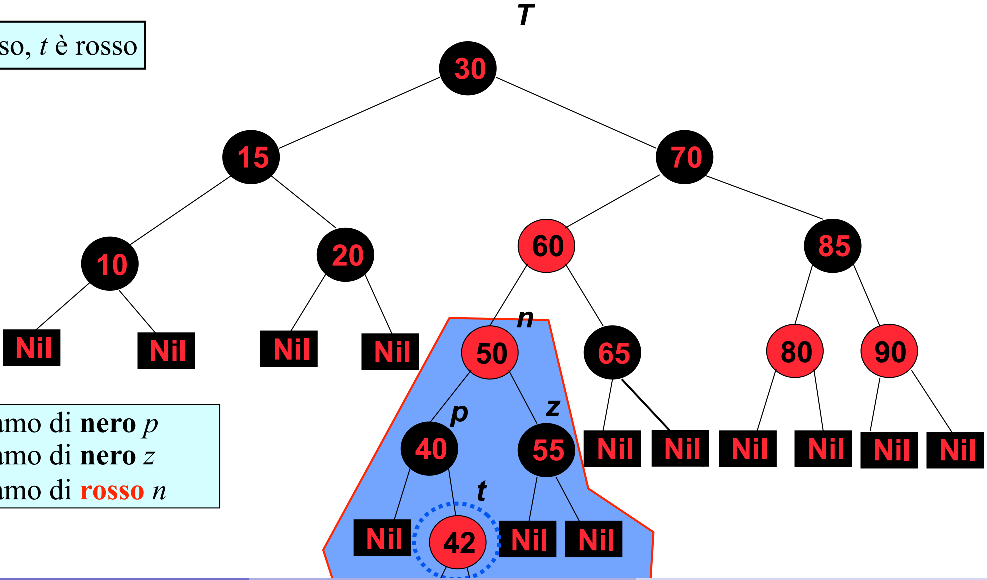


Vincolo 3 è violato

Caso 3: z è rosso

Inserimento – Esempio

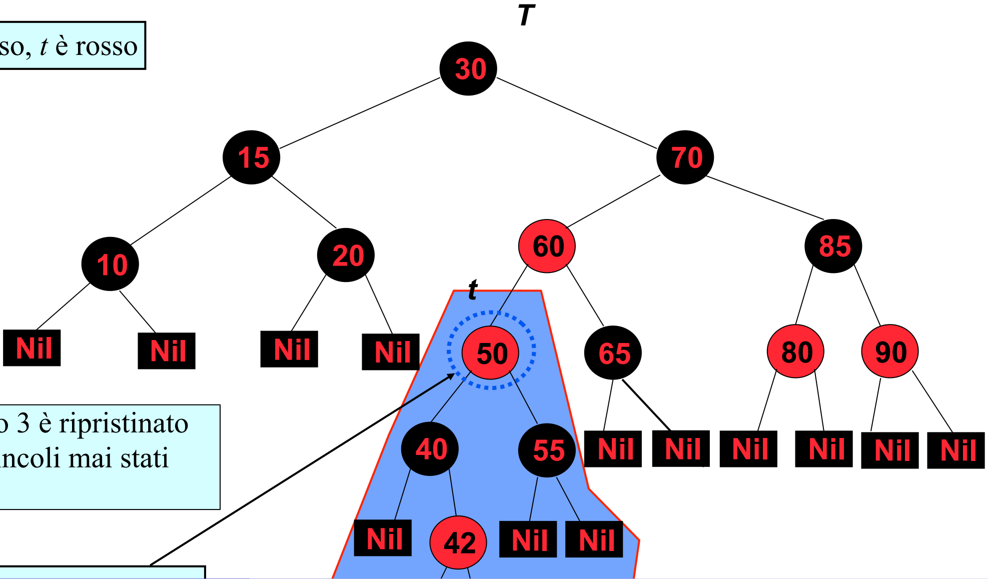
p è rosso, t è rosso



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

Inserimento – Esempio

p è rosso, *t* è rosso

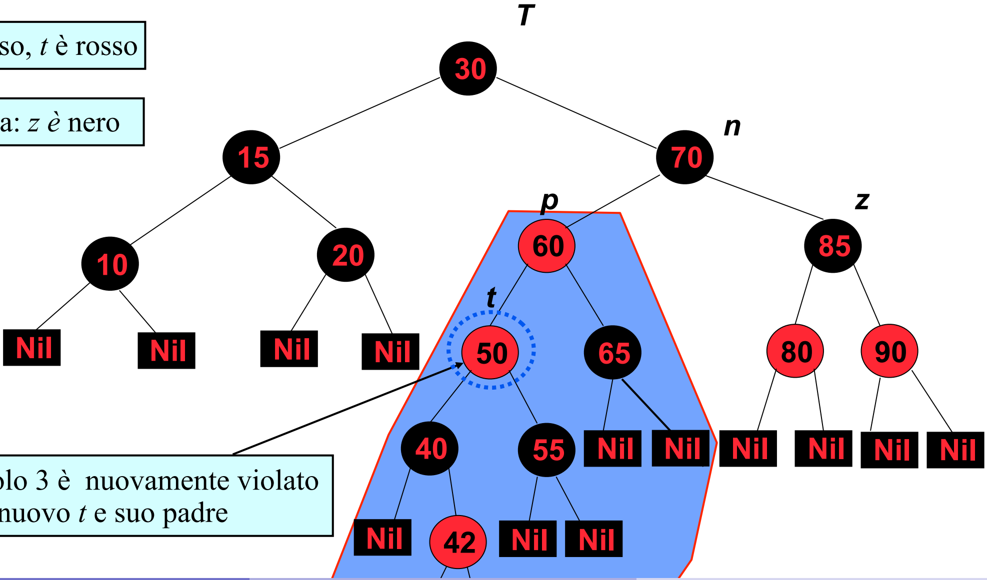


Vincolo 3 è ripristinato
Altri vincoli mai stati violati

Inserimento – Esempio

p è rosso, t è rosso

Caso 5a: z è nero

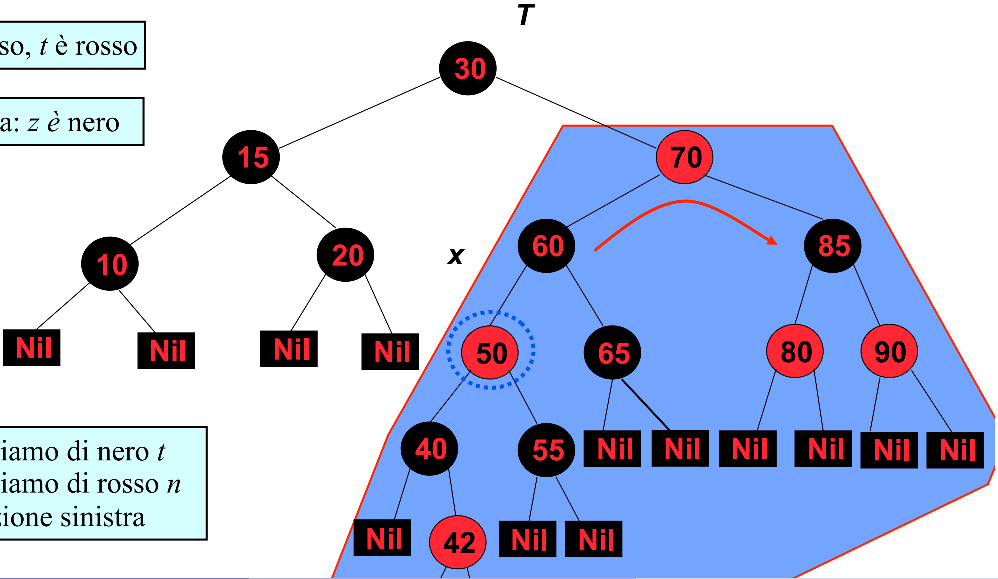


Vincolo 3 è nuovamente violato tra il nuovo t e suo padre

Inserimento – Esempio

p è rosso, t è rosso

Caso 5a: z è nero

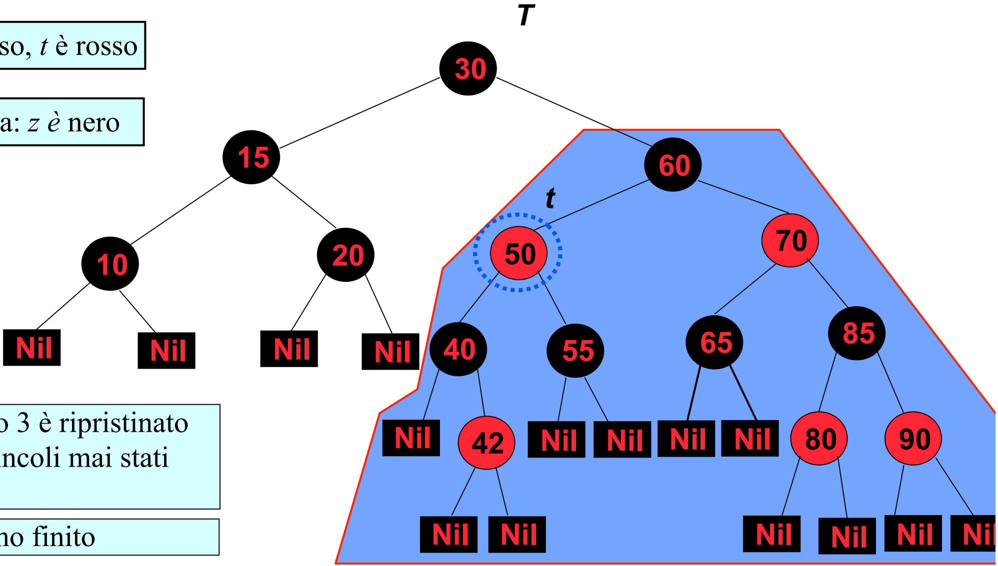


Coloriamo di nero t
Coloriamo di rosso n
Rotazione sinistra

Inserimento – Esempio

p è rosso, t è rosso

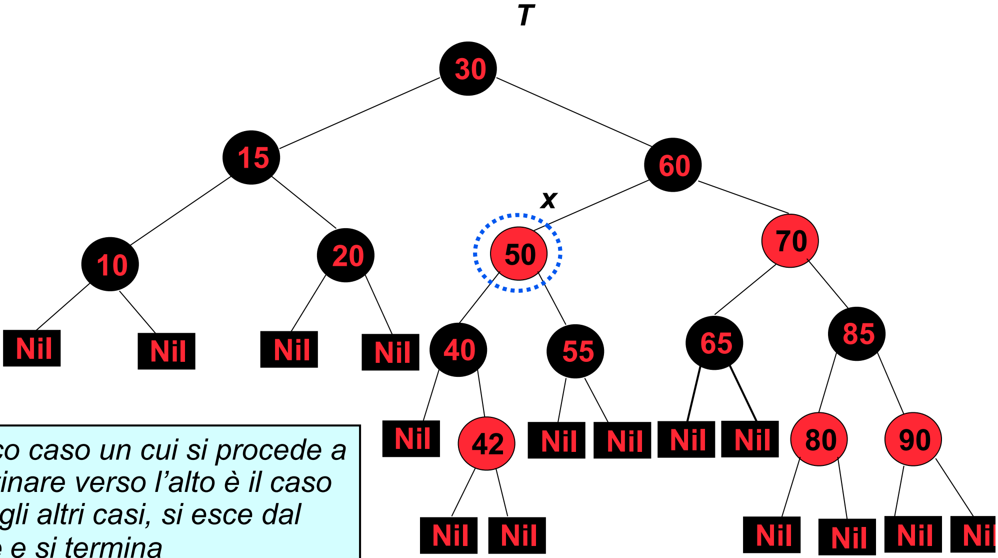
Caso 5a: z è nero



Vincolo 3 è ripristinato
Altri vincoli mai stati violati

Abbiamo finito

Inserimento – Esempio



*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*

Altezza albero Red-Black

Teorema

In un albero RB, un sottoalbero di radice u contiene $n \geq 2^{bh(u)} - 1$ nodi interni (nodi non foglie **Nil**).

Dimostrazione

Caso base $h = 0$:

- Se $h = 0$, u è una foglia **Nil**
- il sottoalbero con radice u contiene $n \geq 2^{bh(u)} - 1 = 2^0 - 1 = 0$ nodi interni

Altezza albero Red-Black

Teorema

In un albero RB, un sottoalbero di radice u contiene $n \geq 2^{bh(u)} - 1$ nodi interni (nodi non foglie **Nil**).

Dimostrazione

Passo induttivo $h > 1$:

- Allora u è un nodo interno con due figli
- Ogni figlio v di u ha un'altezza nera $bh(v)$ pari a:
  - Se rosso: $bh(u)$
  - Se nero: $bh(u) - 1$
- Per ip. induttiva, ogni figlio ha $\geq 2^{bh(u)-1} - 1$ nodi interni
- Quindi, il n. di nodi interni del sottoalbero con radice u è:

$$n \geq 2 \cdot \left(2^{bh(u)-1} - 1\right) + 1 = 2^{bh(u)} - 2 + 1 = 2^{bh(u)} - 1$$

Altezza albero Red-Black

Teorema

In un albero RB, almeno la metà dei nodi dalla radice ad una foglia deve essere nera.

Dimostrazione

- Per il vincolo 2, se un nodo è rosso, i suoi figli devono essere neri.
- La situazione in cui sono presenti il minor numero di nodi neri è il caso in cui rossi e neri sono alternati
- Quindi, almeno la metà dei nodi deve essere nera.

Altezza albero Red-Black

Teorema

In un albero RB, dati due cammini dalla radice a due foglie, non è possibile che uno sia più lungo del doppio dell'altro.

Dimostrazione

- Per il vincolo 4, ogni cammino da un nodo ad una qualsiasi foglia contiene lo stesso numero di nodi neri.
- Per il Lemma precedente, almeno metà dei nodi in ognuno di questi cammini sono neri.
- Quindi, al limite, uno dei due cammini è costituito da solo nodi neri, mentre l'altro è costituito da nodi neri e rossi alternati.

Altezza albero Red-Black

Teorema

L'altezza massima di un albero rosso-nero con n nodi interni è al più $2\log(n+1)$.

Dimostrazione

$$
\begin{aligned}
n \geq 2^{bh(r)} - 1 \quad &\Leftrightarrow \quad n \geq 2^{h/2} - 1 \\
&\Leftrightarrow \quad n + 1 \geq 2^{h/2} \\
&\Leftrightarrow \quad \log(n+1) \geq h/2 \\
&\Leftrightarrow \quad h \leq 2\log(n+1)
\end{aligned}
$$

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

- È 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 ≠ 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 = 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 $O(1)$

Alberi Red-Black in Popular Culture

Gli alberi RB sono menzionati (correttamente) in un episodio della serie TV canadese "Missing"

Jess: "It was the red door again."
Pollock: "I thought the red door was the storage container."
Jess: "But it wasn't red anymore, it was black."
Antonio: "So red turning to black means what?"
Pollock: "Budget deficits, red ink, black ink."
Antonio: "It could be from a binary search tree. The red-black tree tracks every simple path from a node to a descendant leaf that has the same number of black nodes."
Jess: "Does that help you with the ladies?"

https://en.wikipedia.org/wiki/Red%E2%80%93black_tree#Popular_culture