# algoritmi e strutture di dati

## alberi rosso-neri

*m.patrignani*

## contenuto

- definizione di alberi rosso-neri
- proprietà degli alberi rosso-neri
- complessità delle operazioni elementari
- rotazioni
- inserimenti e cancellazioni

# motivazioni

- un dizionario realizzato con un albero binario di ricerca consente operazioni efficienti quando l'albero è bilanciato

| alberi binari di ricerca (complessità nel caso peggiore) | | |
|---|---|---|
| **operazione** | **sbilanciati** | **bilanciati** |
| ricerca | $\Theta(n)$ | $\Theta(\lg n)$ |
| inserimento | $\Theta(n)$ | $\Theta(\lg n)$ |
| cancellazione | $\Theta(n)$ | $\Theta(\lg n)$ |

- ha senso investire delle risorse per mantenere l'albero bilanciato

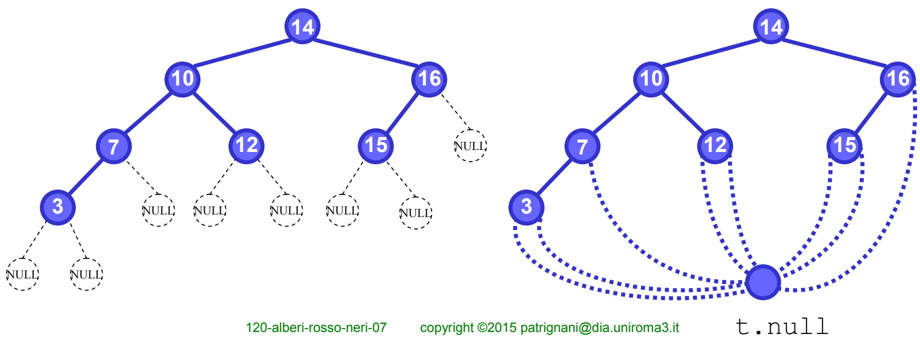


# albero bilanciato

- definiamo come *bilanciato* un albero in cui la lunghezza del cammino più lungo tra la radice e una foglia è al massimo due volte la lunghezza del cammino più corto
- per gestire il bilanciamento ad ogni nodo viene associato un campo aggiuntivo che specifica il colore: rosso o nero

# albero con sentinelle

- gestire il bilanciamento di un albero è un obiettivo complesso
- per semplicità vorremmo che non ci siano nodi con un solo figlio destro o un solo figlio sinistro
  - questo può essere realizzato aggiungendo all'albero `t` un nodo "sentinella" `t.null` e sostituendo con un puntatore a `t.null` ogni valore NULL del puntatore `x.left` o `x.right` di un nodo `x`



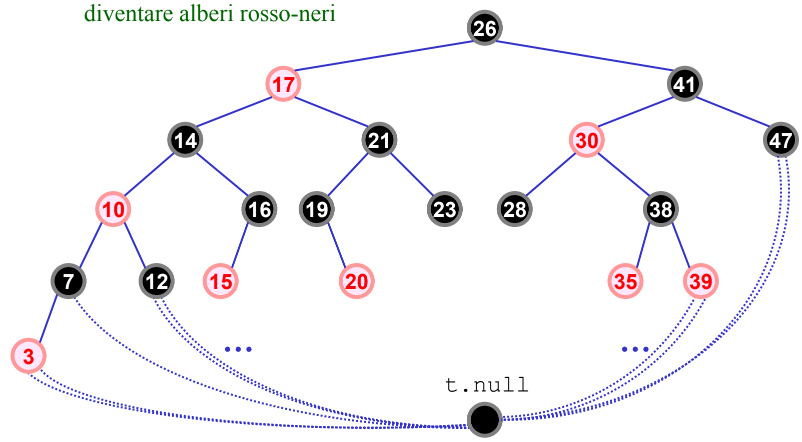



# definizione di alberi rosso-neri

- un albero rosso-nero è un albero binario di ricerca nel quale
  1. ogni nodo è rosso o nero
  2. la radice e la sentinella `t.null` sono nere
  3. se un nodo è rosso entrambi i suoi figli sono neri
  4. per ogni nodo, tutti i percorsi che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri

# esempio di albero rosso-nero

- attenzione
  - l'albero deve essere un albero binario di ricerca
  - non tutti gli alberi binari di ricerca possono essere colorati in maniera da diventare alberi rosso-neri

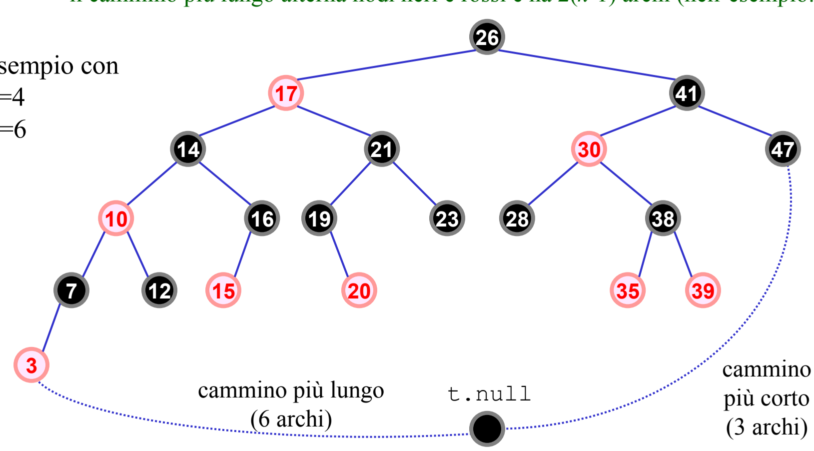


# alberi rosso-neri e bilanciamento

- supponiamo che tutti i cammini dalla radice ad una foglia abbiano *k* nodi neri
  - ogni cammino ha almento *k*-1 archi (nell'esempio: 3 archi)
  - il cammino più lungo alterna nodi neri e rossi e ha 2(*k*-1) archi (nell'esempio: 6 archi)

esempio con
*k*=4
*h*=6

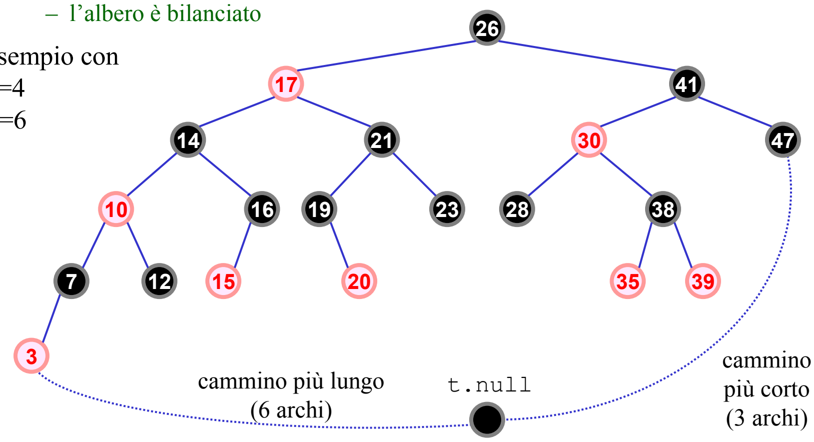


# alberi rosso-neri e bilanciamento

- supponiamo che tutti i cammini dalla radice ad una foglia abbiano *k* nodi neri
  - la lunghezza del cammino più lungo (2(*k*-1)) è al massimo due volte la lunghezza del cammino più corto (*k*-1)
  - l'albero è bilanciato

esempio con
*k*=4
*h*=6

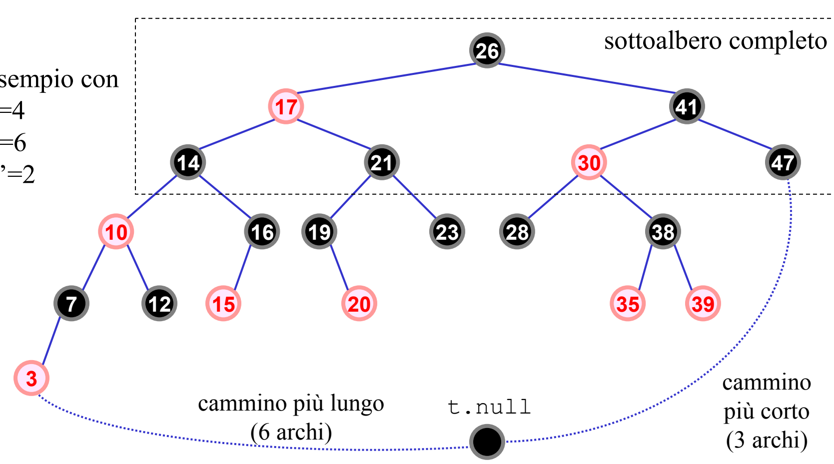


# alberi rosso-neri e profondità

- supponiamo che tutti i cammini dalla radice ad una foglia abbiano *k* nodi neri
  - l'albero contiene un sottoalbero completo di profondità $h' = h/2 - 1$

esempio con
*k*=4
*h*=6
*h*'=2



# alberi rosso-neri e numero dei nodi

- supponiamo che tutti i cammini dalla radice ad una foglia abbiano $k$ nodi neri
  - l'albero ha profondità massima $h = 2(k-1)$
  - l'albero contiene un sottoalbero completo di profondità $h' = h/2 - 1$
- i nodi interni dell'albero sono almeno quelli del sottoalbero completo
  - ricorda che un albero completo di altezza $x$ ha $2^{x+1}-1$ nodi

$$n \geq 2^{h'+1} - 1 = 2^{\left(\frac{h}{2}-1\right)+1} - 1 = 2^{\frac{h}{2}} - 1$$

$$n + 1 \geq 2^{\frac{h}{2}}$$

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

- sappiamo però che $h$ è almeno l'altezza di un albero completo con $n$ nodi, cioè $h \in \Omega(\lg(n))$
- dunque $h \in \Theta(\lg(n))$



# operazioni sugli alberi rosso-neri

- l'altezza dell'albero è logaritmica nel numero dei nodi ($h \in \Theta(\lg n)$)
- tutte le operazioni di consultazione eseguibili in tempo $\Theta(h)$ su un albero binario di ricerca sono eseguibili in tempo $\Theta(\lg n)$ su un albero rosso-nero:
  - SEARCH
  - MINIMUM
  - MAXIMUM
  - SUCCESSOR
  - PREDECESSOR

## operazioni INSERT e DELETE

- le operazioni INSERT e DELETE possono ugualmente essere eseguite in $\Theta(\ln n)$
- TREE-INSERT e TREE-DELETE, però, non garantiscono la conservazione delle proprietà degli alberi rosso-neri
  - a valle delle operazioni di inserimento e cancellazione vengono lanciate delle procedure che ripristinano tali proprietà in $\Theta(\ln n)$
- nel seguito vedremo a titolo di esempio la sola procedura RB-INSERT per l'inserimento di un nodo

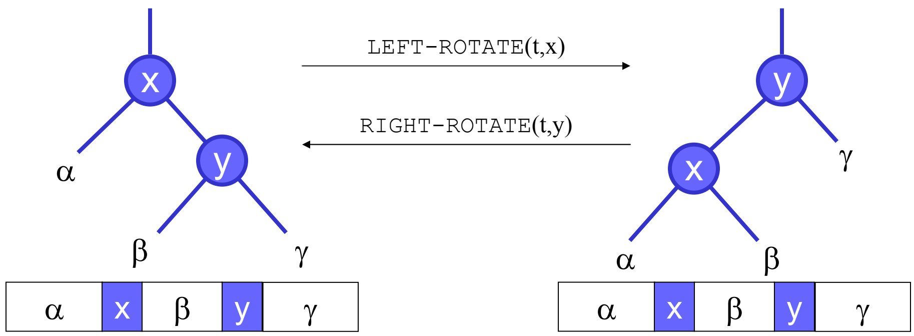


## procedura RB-INSERT

```
RB-INSERT(t,new)          ▷ inserisco il nodo new nell'albero t
    y = t.null
    x = t.root
    while x != t.null       ▷ finché non sono arrivato a t.null
        y = x               ▷ cerco il padre y a cui appendere new
        if new.key < x.key
            x = x.left
        else x = x.right
    new.p = y                   ▷ aggiorno il genitore di new
    if y == t.null              ▷ se new deve diventare la radice…
        t.root = new            ▷ …aggiorno t.root
    else if new.key < y.key
        y.left = new
    else y.right = new
    new.left = new.right = t.null
    new.color = RED             ▷ i nuovi nodi sono sempre rossi
    RB-INSERT-FIXUP(t,new)      ▷ ripristina le proprietà dell'albero
```

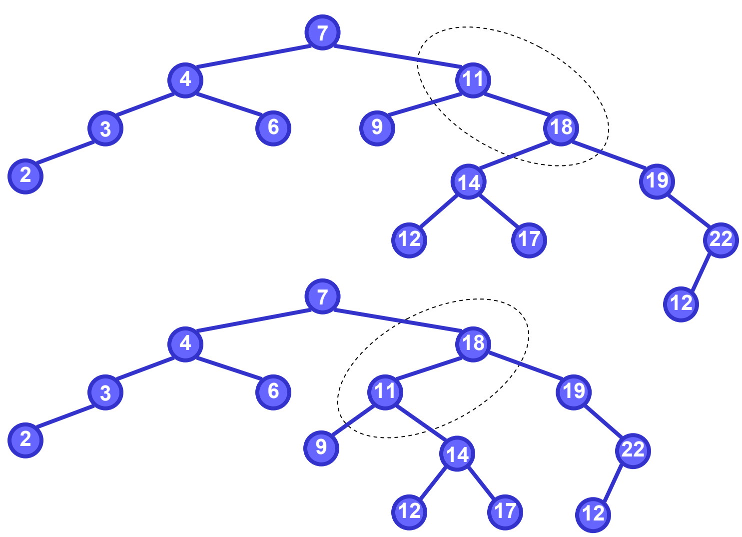
# rotazioni

- l'operazione base che viene utilizzata per ripristinare le proprietà dell'albero rosso-nero è la rotazione
  - le rotazioni non alterano i colori dei nodi
  - l'albero rimane un albero binario di ricerca
  - l'operazione può essere eseguita in tempo $\Theta(1)$



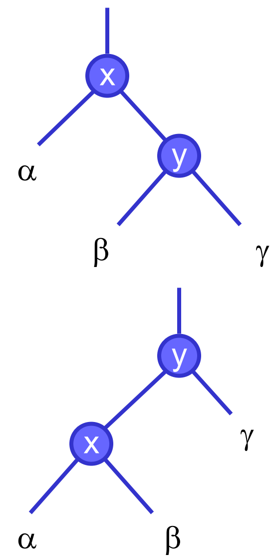
# esempio di rotazione a sinistra



# procedura LEFT-ROTATE

```
LEFT-ROTATE(t,x)
y = x.right              ▷ trovo y
x.right = y.left         ▷ sposto β
if y.left != t.null
    y.left.p = x
y.p = x.p
if x.p == t.null
    t.root = y
else if x == x.p.left
    x.p.left = y
    else x.p.right = y
y.left = x
x.p = y
```



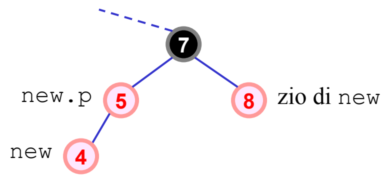



# ripristino dell'albero rosso-nero

- il nuovo nodo aggiunto è una foglia e ha colore rosso
- ricordiamo i vincoli di un albero rosso-nero
    1. ogni nodo è rosso o nero
    2. la radice e la sentinella `t.null` sono nere
    3. se un nodo è rosso entrambi i suoi figli sono neri
    4. per ogni nodo, tutti i percorsi che vanno dal nodo alle foglie sue discendenti contengono lo stesso numero di nodi neri
- se l'albero era vuoto la proprietà 2 è violata
    - in questo caso è sufficiente colorare la radice di nero
- altrimenti solo la proprietà 3 potrebbe essere violata
    - situazione più complicata

## violazione: nodo rosso con un figlio rosso

- se `RB-INSERT` ha appeso il nuovo nodo `new` (che è sempre rosso) ad un genitore rosso
  - chiamiamo “zio di `new`” il nodo fratello del genitore di `new`
    - lo zio di `new` esiste sempre, eventualmente è `t.null`
  - sono possibili tre casi




caso 1: lo zio di `new` è nero e `new` è un figlio sinistro

caso 2: lo zio di `new` è nero e `new` è un figlio destro

caso 3: lo zio di `new` è rosso



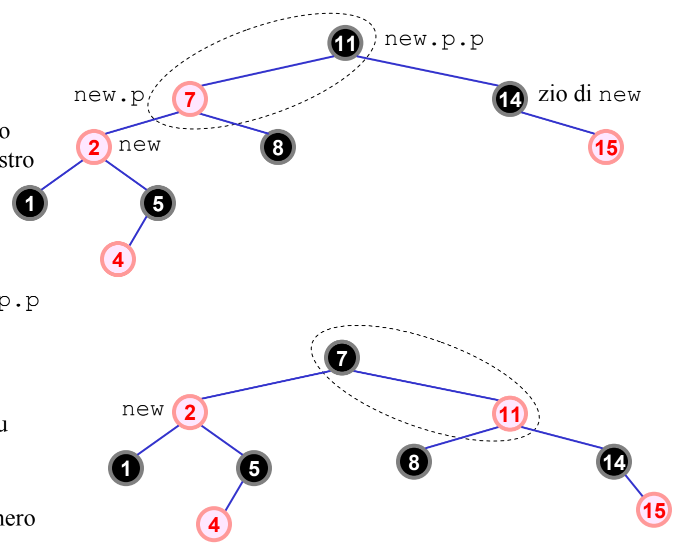
## violazione: caso 1

caso 1:
lo zio di `new` è nero
e `new` è un figlio sinistro

ricolorazione
di `new.p` e di `new.p.p`

rotazione destra su
`new.p.p`

ora l’albero è rosso-nero

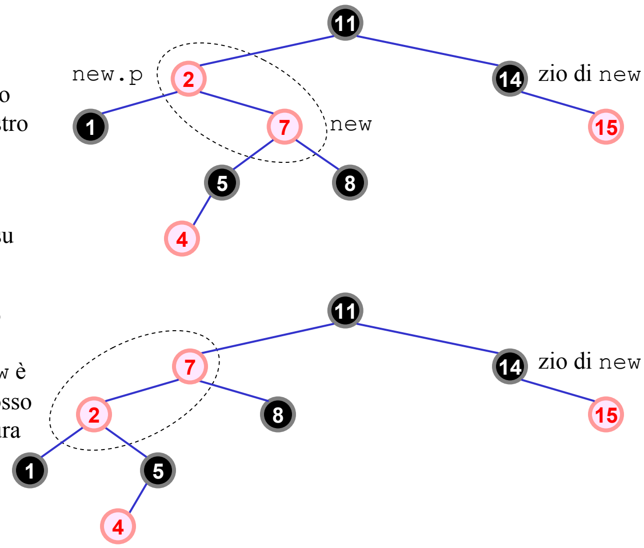


# violazione: caso 2

caso 2:
lo zio di `new` è nero
e `new` è un figlio destro

rotazione sinistra su
`new.p`

i due nodi violano
ancora la regola 4
(ma questa volta `new` è
un figlio sinistro e posso
applicare la procedura
del caso 1)





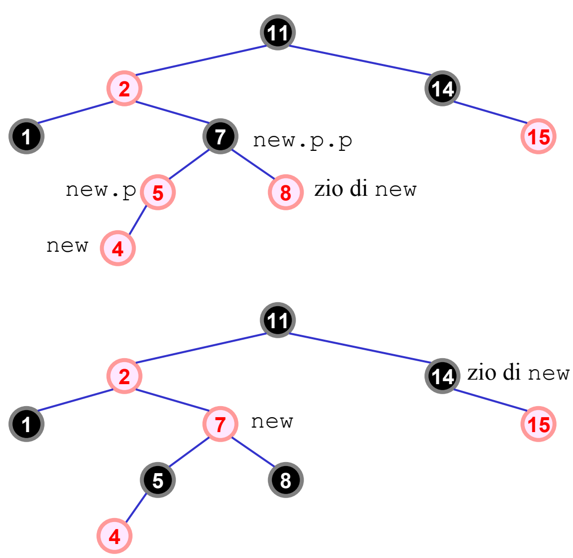

# violazione: caso 3

caso 3:
lo zio di `new` è rosso

ricolorazione
di `new.p` e `new.p.p`

iterazione:
`new = new.p.p`

ora `new` e `new.p`
potrebbero ancora
violare la regola 4
(ma `new` è più
vicino alla radice)



## violazione: caso 3

- nel caso 3 `new` è più vicino alla radice ma potrebbe violare la regola 4 con `new.p`
- occorre rilanciare la procedura con il nuovo `new`
- l'intera procedura può essere rilanciata al massimo $\Theta(\ln n)$ volte
  - il caso peggiore è quando si ha una sequenza di casi 3 fino a che non si risale alla radice

## cancellazioni in un albero rosso-nero

- analogamente ad `RB-INSERT`, la procedura `RB-DELETE`
  - prima cancella un nodo con la stessa strategia di `TREE-DELETE` degli alberi binari di ricerca
  - poi ripristina le proprietà degli alberi rosso-neri chiamando una opportuna procedura `RB-DELETE-FIXUP`
    - `RB-DELETE-FIXUP` utilizza rotazioni e ricolorazioni

## conclusioni

- complessivamente gli alberi rosso-neri offrono una realizzazione di alberi binari di ricerca con le seguenti complessità nel caso peggiore
  - inserimento in $\Theta(\log n)$
  - cancellazione in $\Theta(\log n)$
  - ricerca in $\Theta(\log n)$

## esercizi

- qual è la complessità dell'algoritmo `TREE-SORT`, che utilizza un albero binario di ricerca per ordinare un array, nel caso in cui l'albero sia un albero rosso-nero?
- data una realizzazione del tipo astratto di dato "insieme" tramite un albero rosso-nero con le seguenti funzioni
  - INSERT(t,k) in $\Theta(\log n)$
  - REMOVE(t,k) in $\Theta(\log n)$
  - SEARCH(t,k) in $\Theta(\log n)$

  realizza la funzione `UNIONE`($t_1$,$t_2$) che calcola l'unione di due insiemi $t_1$ e $t_2$ e discutine la complessità