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

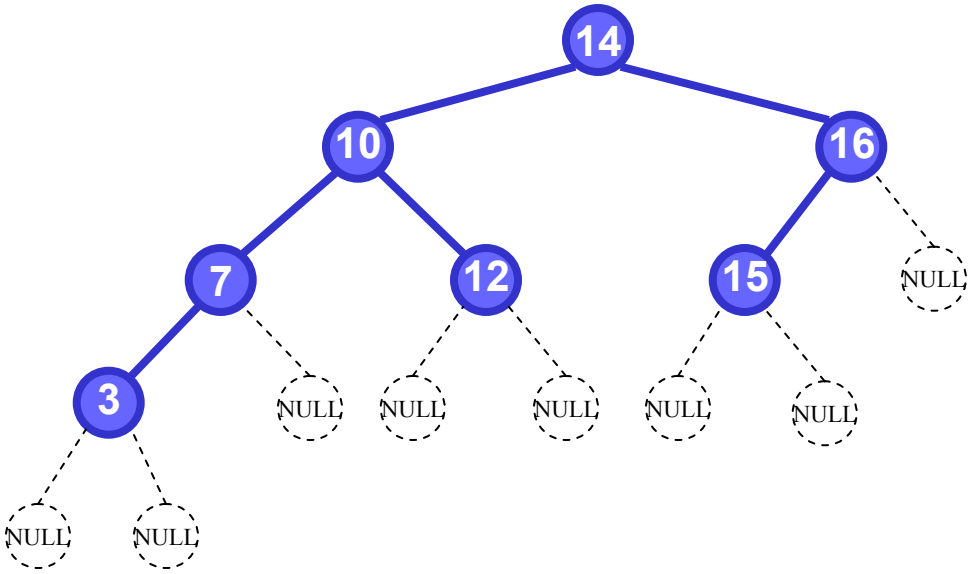
- Un dizionario realizzato con un albero binario di ricerca consente operazioni efficienti quando l'albero è bilanciato

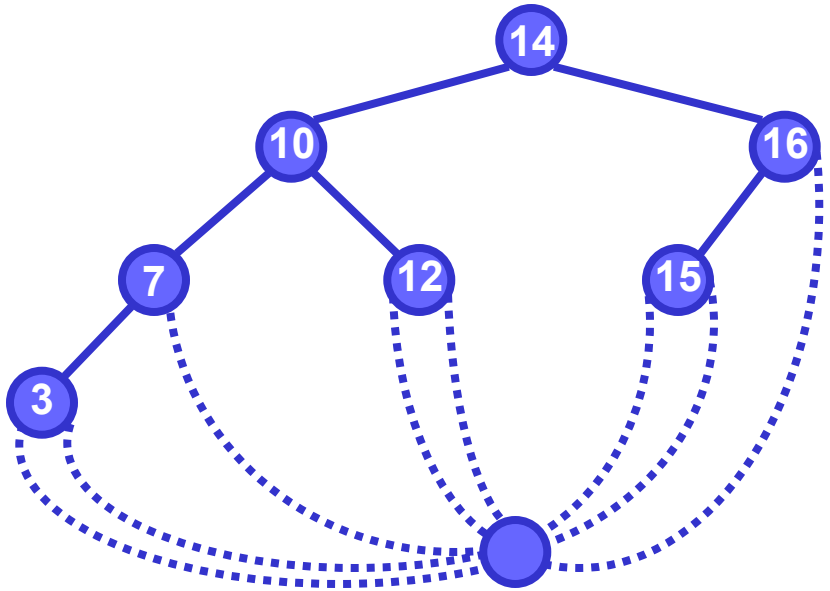
alberi binari di ricerca (complessità nel caso peggiore)		
operazione	**sbilanciati**	**bilanciati**
ricerca	$\Theta(n)$	$\Theta(\log n)$
inserimento	$\Theta(n)$	$\Theta(\log n)$
cancellazione	$\Theta(n)$	$\Theta(\log n)$

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

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`





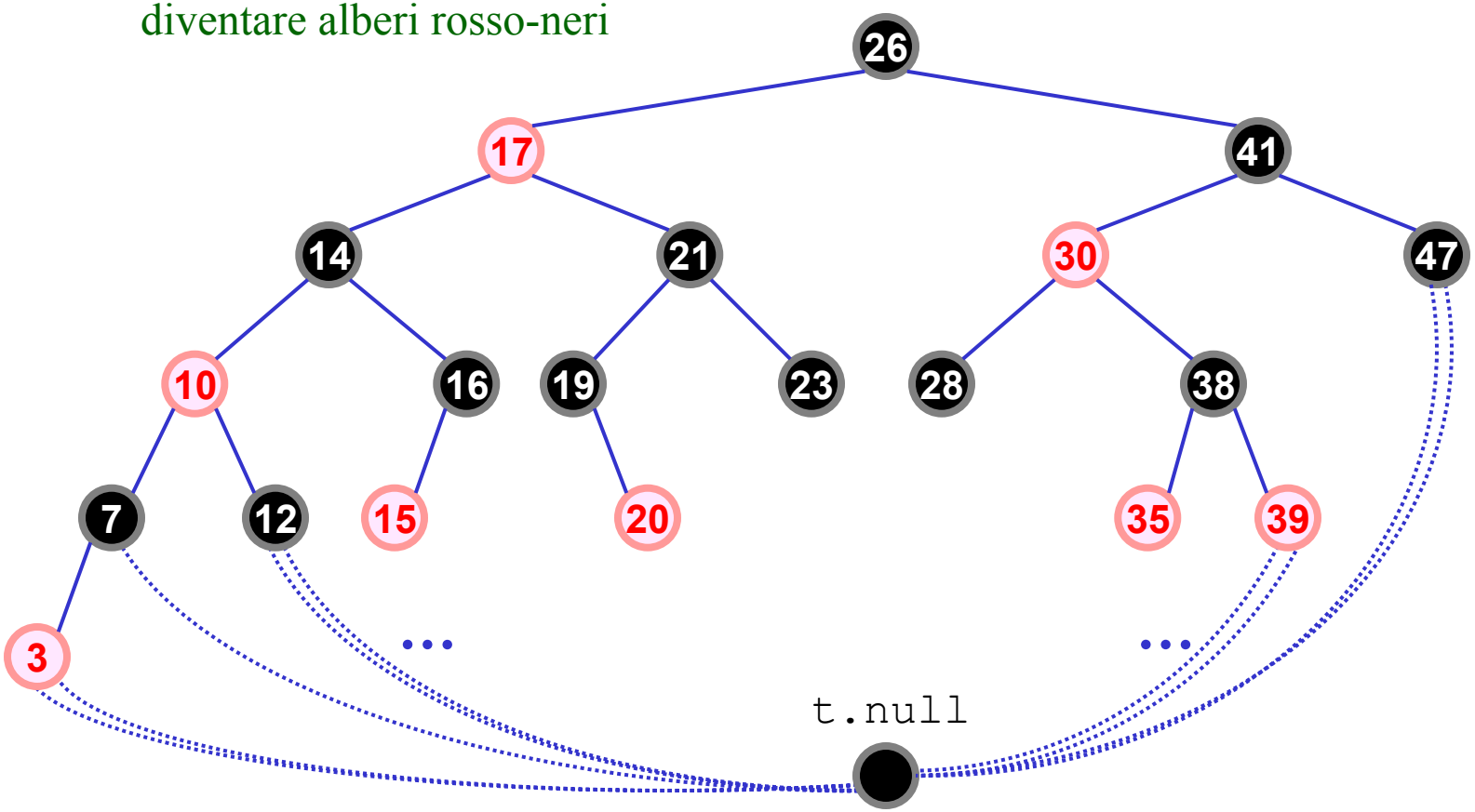
`t.null`

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 `t.root` e la sentinella `t.null` sono nere
  3. se un nodo è rosso entrambi i suoi figli sono neri
  4. tutti i cammini che vanno dalla radice a `t.null` contengono lo stesso numero di nodi neri
- Convenzionalmente chiamiamo "altezza" dell'albero rosso-nero la lunghezza del cammino più lungo tra la radice e `t.null`
  - corrisponde in realtà all'altezza + 1

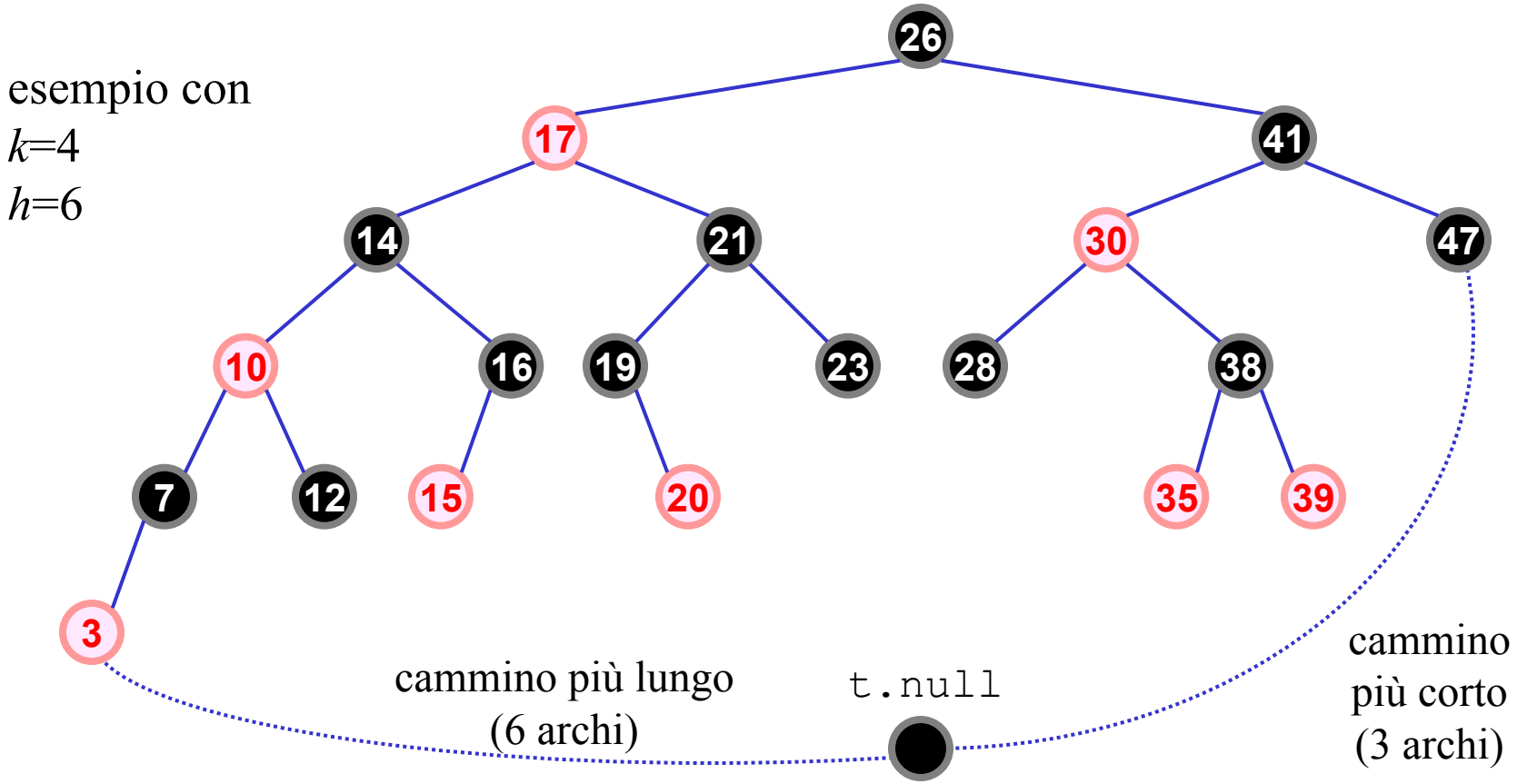
Esempio di albero rosso-nero

- 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



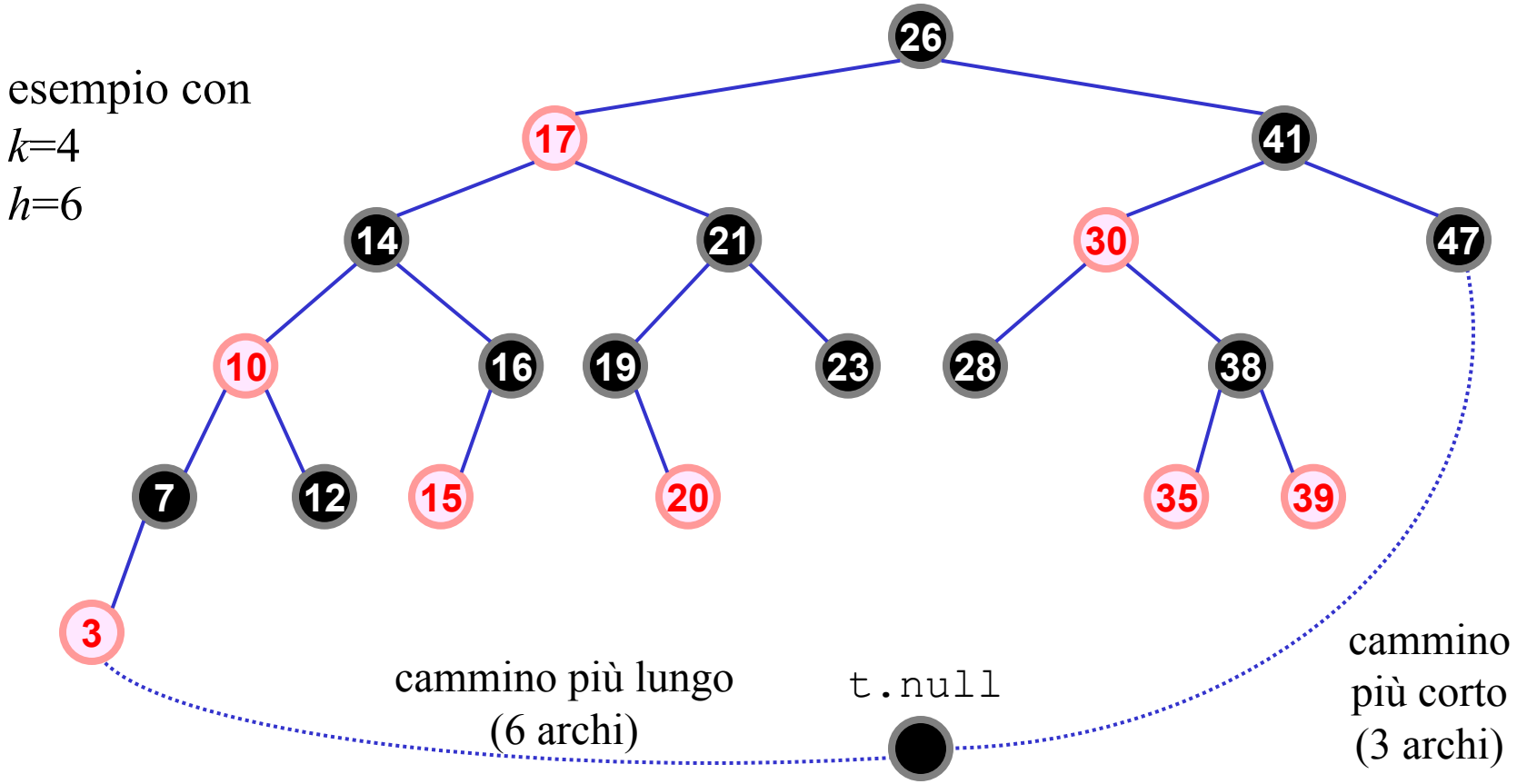
Alberi rosso-neri e bilanciamento

- Tutti i cammini dalla radice a `t.null` hanno k nodi neri (nell'esempio k=4)
  - ogni cammino ha almeno 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)



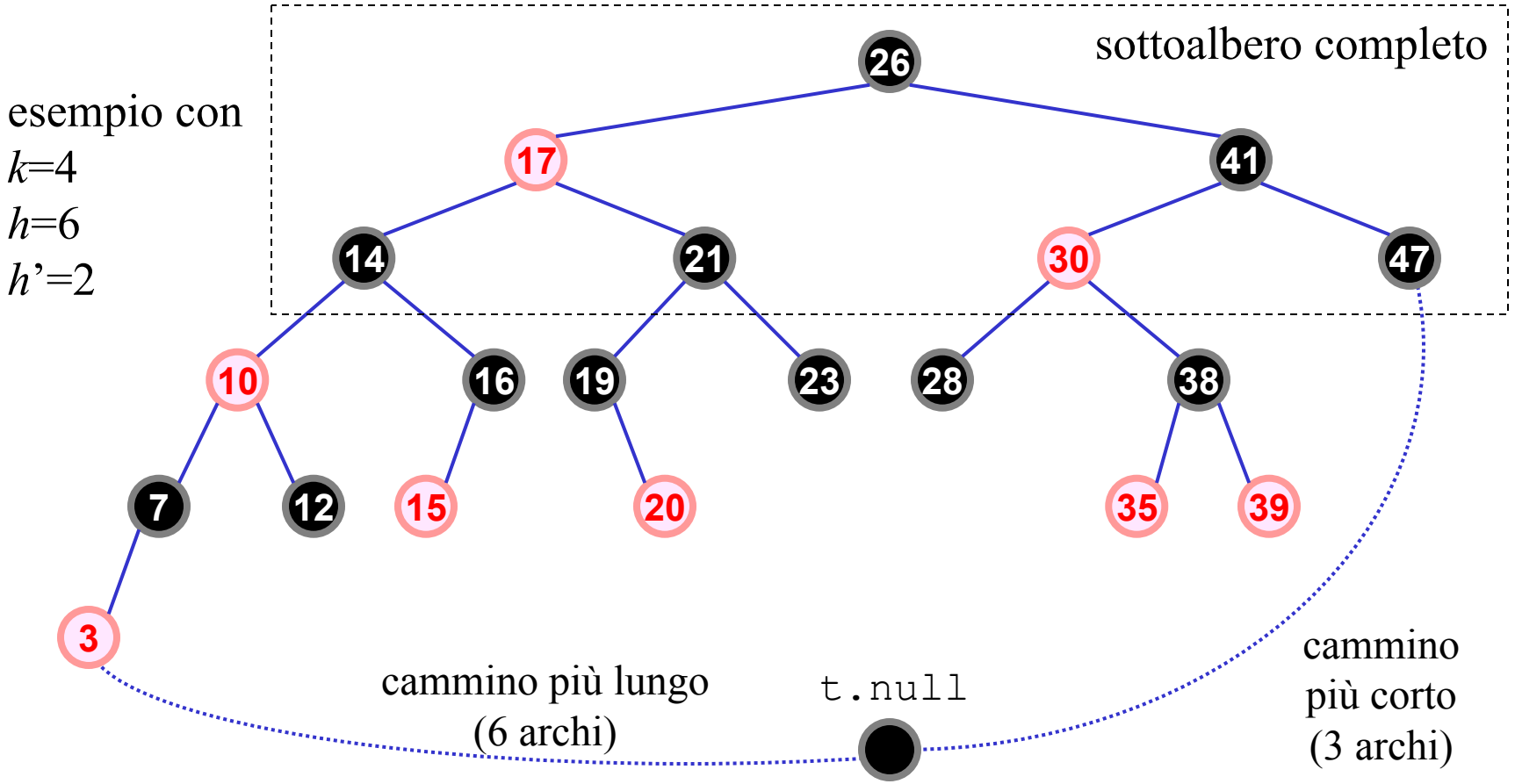
Alberi rosso-neri e bilanciamento

- Tutti i cammini dalla radice a `t.null` hanno k nodi neri (nell'esempio k=4)
  - la lunghezza del cammino più lungo ($2(k\text{-}1)$) è al massimo due volte la lunghezza del cammino più corto (k-1)



Alberi rosso-neri e profondità

- Tutti i cammini dalla radice a `t.null` hanno k nodi neri (nell'esempio k=4)
  - l'albero contiene un sottoalbero completo di profondità $h' = h/2 - 1$



Alberi rosso-neri e numero dei nodi

- Tutti i cammini dalla radice a `t.null` hanno k nodi neri (nell'esempio k=4)
  - l'albero ha profondità massima $h = 2(k\text{-}1)$
  - l'albero contiene un sottoalbero completo di profondità $h' = h/2 - 1$
- I nodi dell'albero sono almeno quelli del sottoalbero completo
  - ricorda che un albero completo di altezza x ha 2^{x+1}-1 nodi

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

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

$$\frac{h}{2} \leq \log(n+1)$$

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

- Dunque $h \in O(\log n)$

Alberi rosso-neri e numero dei nodi

- Abbiamo appena dimostrato che in un albero rosso-nero $h \in O(\log n)$
- Sappiamo però che in un albero binario h è almeno l'altezza di un albero completo con n nodi, cioè $h \in \Omega(\log n)$
- Dunque in un albero rosso-nero $h \in \Theta(\log n)$

Operazioni sugli alberi rosso-neri

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

Operazioni `INSERT` e `DELETE`

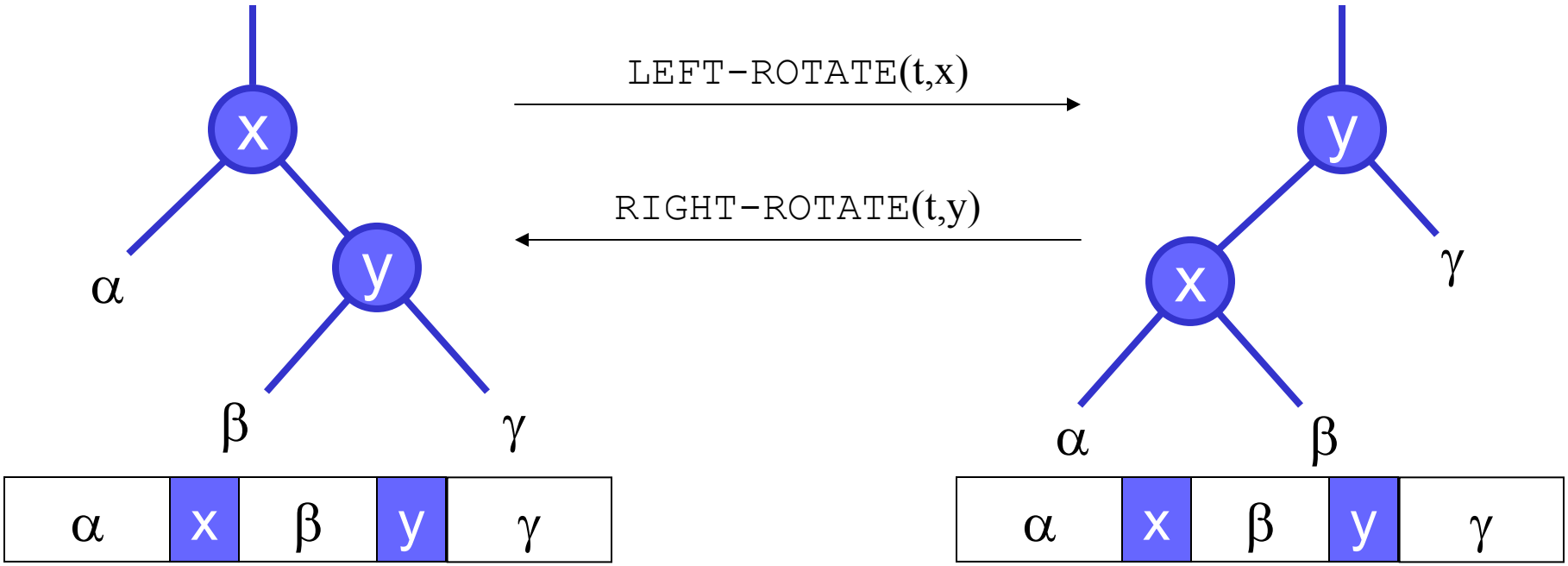
- Le operazioni `INSERT` e `DELETE` possono ugualmente essere eseguite in $\Theta(\log n)$
- `TREE_INSERT` e `TREE_DELETE`, però, non garantiscono la conservazione delle proprietà degli alberi rosso-neri
  - a valle delle operazioni di inserimento e cancellazione devono essere lanciate delle procedure che ripristinano tali proprietà in $\Theta(\log n)$
- Nel seguito vedremo a titolo di esempio la sola procedura `RB_INSERT` per l'inserimento di un nodo

Procedura RB_INSERT

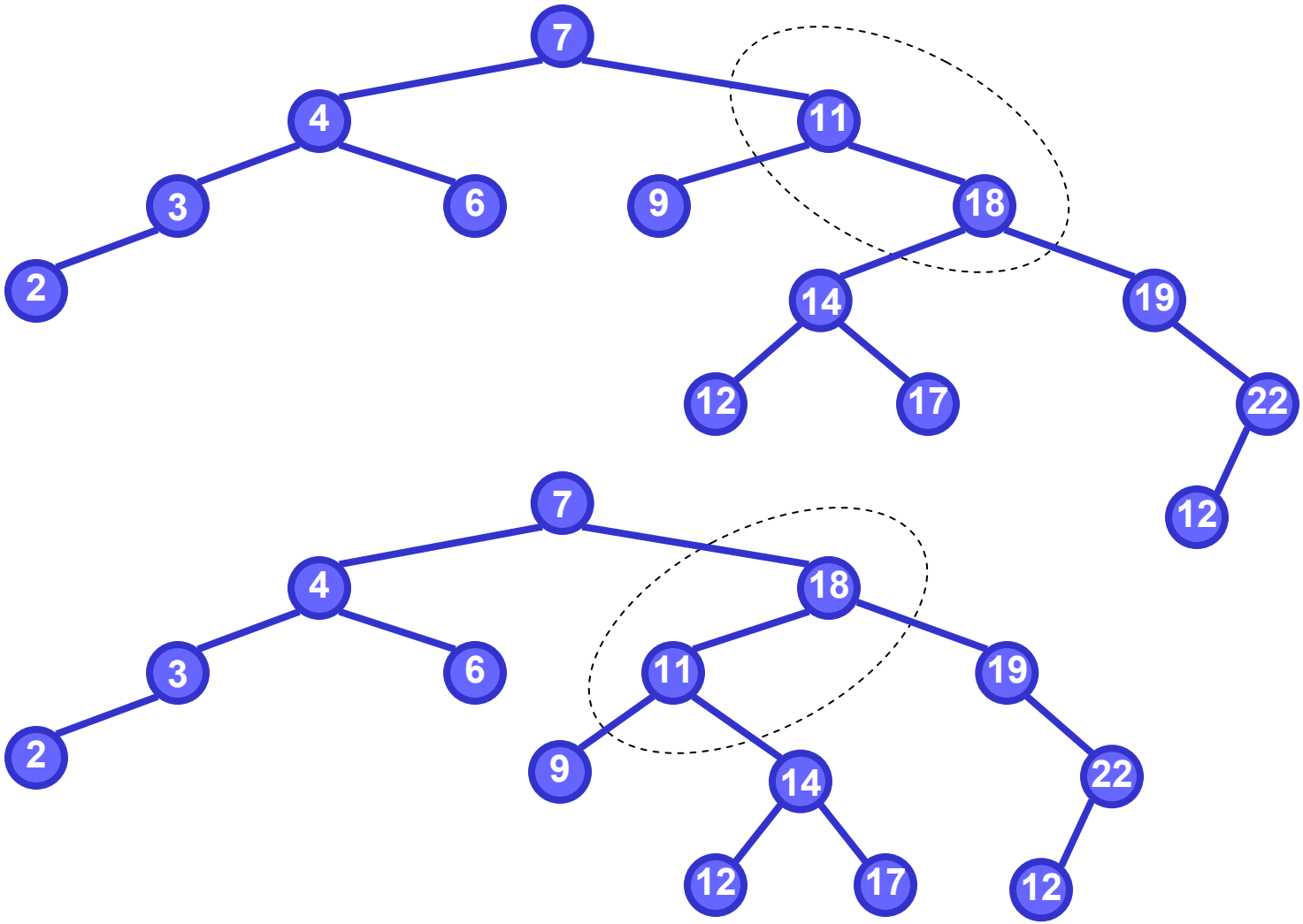
```
RB_INSERT(t,new)        /* inserisco il nodo new nell'albero t */
1.  y = t.null
2.  x = t.root
3.  while x != t.null       // finché non sono arrivato a t.null
4.      y = x               // cerco il padre y a cui appendere new
5.      if new.key < x.key
6.          x = x.left
7.      else x = x.right
8.  new.p = y                   // aggiorno il genitore di new
9.  if y == t.null              // se new deve diventare la radice…
10.     t.root = new            // …aggiorno t.root
11.  else if new.key < y.key
12.     y.left = new
13.  else y.right = new
14.  new.left = new.right = t.null
15.  new.color = RED            // i nuovi nodi sono sempre rossi
16.  RB_INSERT_FIXUP(t,new)     // ripristina le proprietà
```

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 $\Theta(1)$

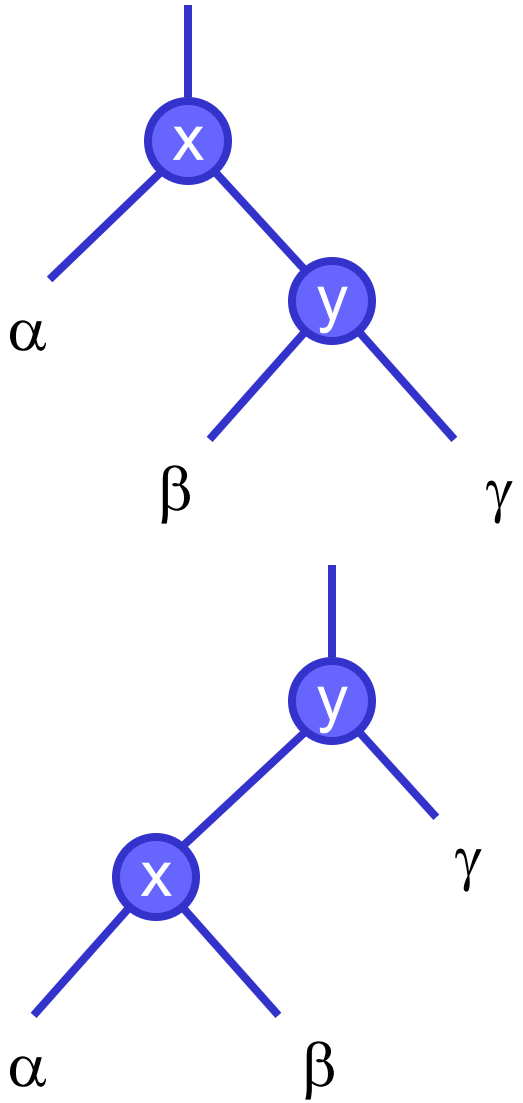


Esempio di rotazione a sinistra



Procedura LEFT_ROTATE

```
LEFT_ROTATE(t,x)
1.  y = x.right                 // trovo y
2.  x.right = y.left            // sposto β
3.  if y.left != t.null
4.      y.left.p = x
5.  y.p = x.p
6.  if x.p == t.null
7.      t.root = y
8.  else if x == x.p.left
9.      x.p.left = y
10.     else x.p.right = y
11. y.left = x
12. x.p = y
```

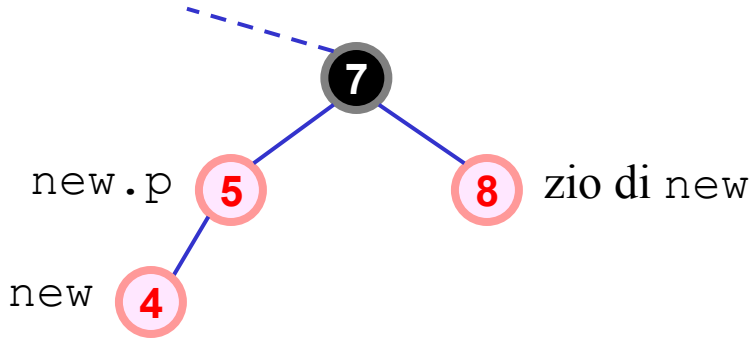


Ripristino dell'albero rosso-nero

- 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. tutti i cammini che vanno dalla radice a `t.null` 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: new è un figlio sinistro e lo zio è nero e figlio destro

caso 1': new è un figlio destro e lo zio è nero e figlio sinistro

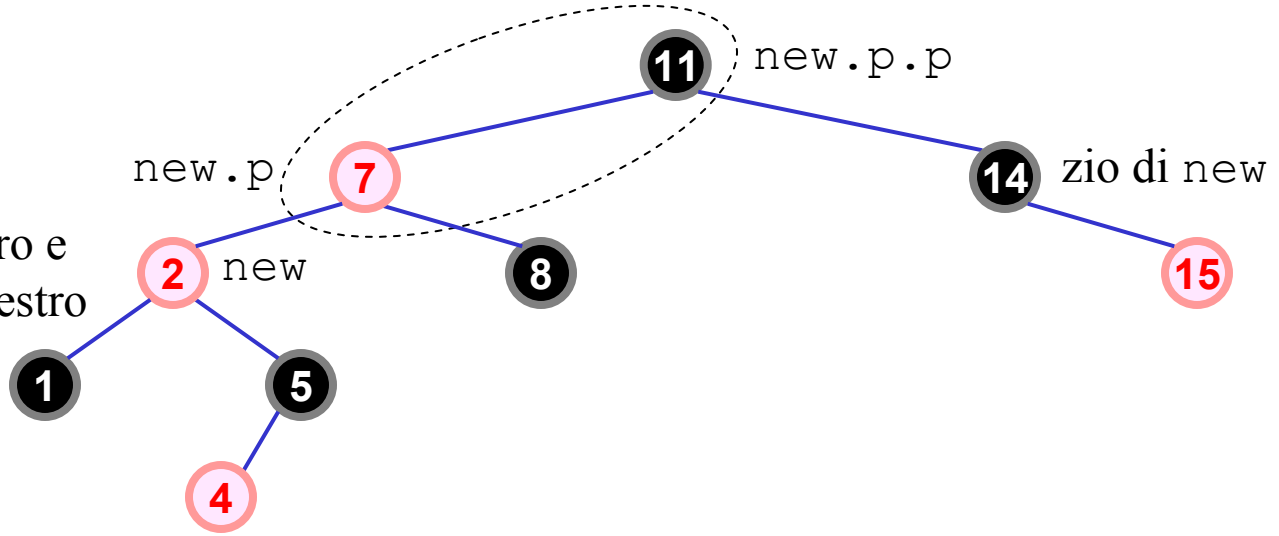
caso 2: new è un figlio destro e lo zio è nero e figlio destro

caso 2': new è un figlio sinistro e lo zio è nero e figlio sinistro

caso 3: lo zio di `new` è rosso

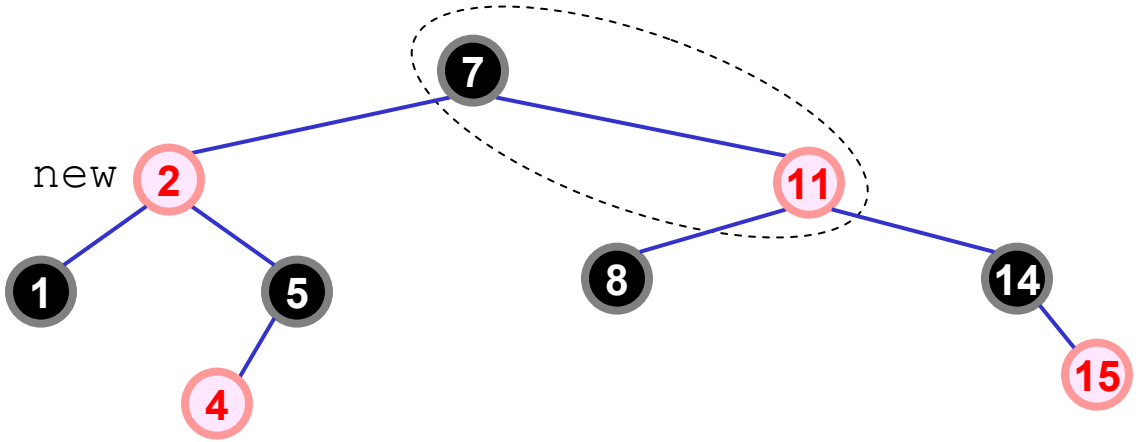
Violazione: caso 1

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



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

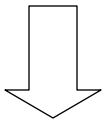
rotazione destra su
`new.p.p`



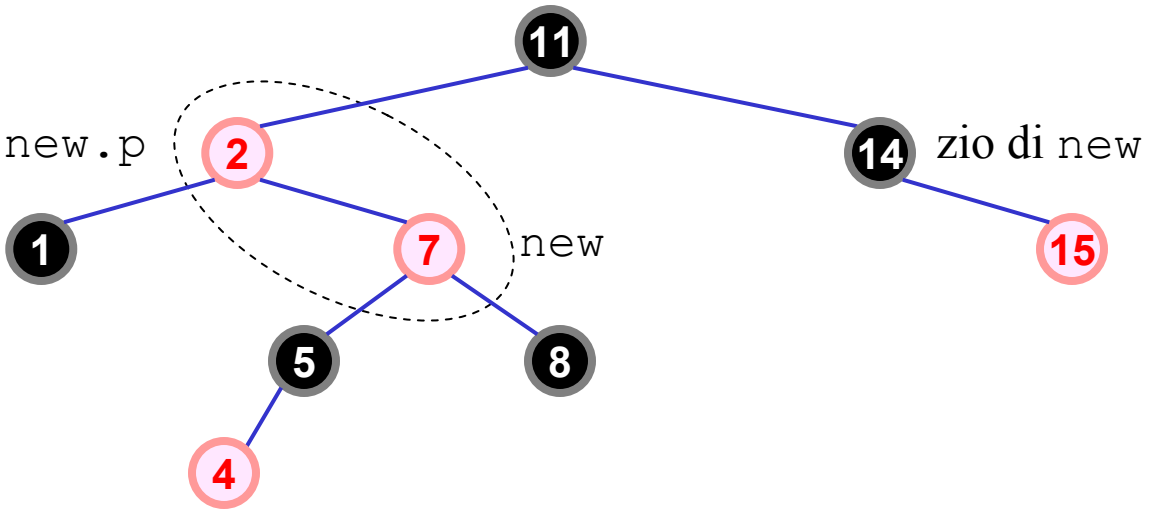
ora l'albero è rosso-nero

Violazione: caso 2

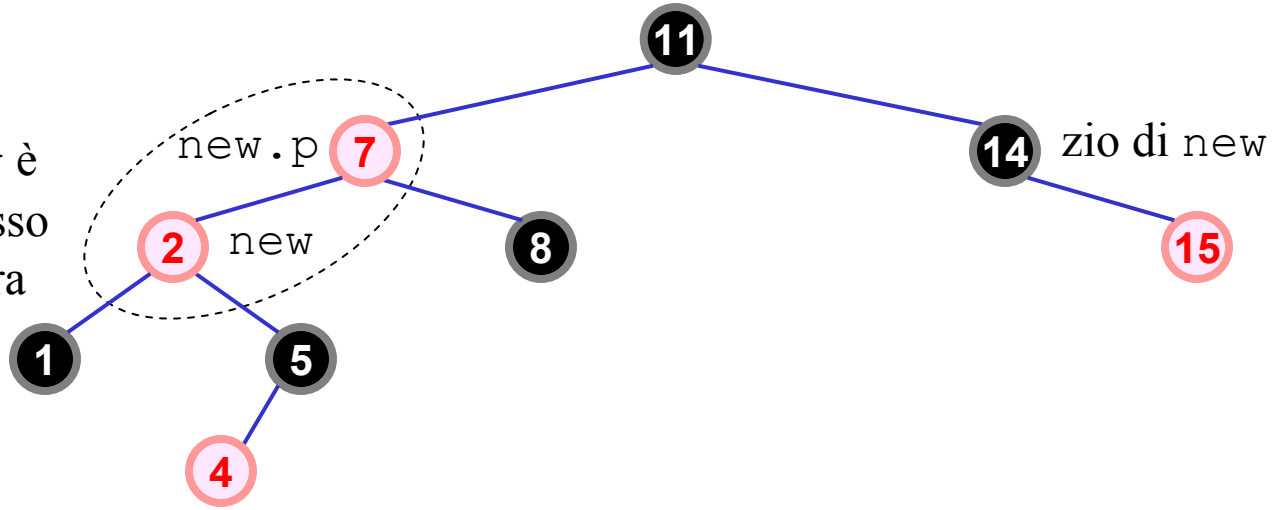
caso 2:
new è un figlio destro e lo zio è nero e figlio destro



rotazione sinistra su `new.p`

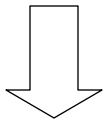


i due nodi violano ancora la regola 3 (ma questa volta `new` è un figlio sinistro e posso applicare la procedura del caso 1)

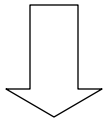


Violazione: caso 3

caso 3:
lo zio di `new` è rosso

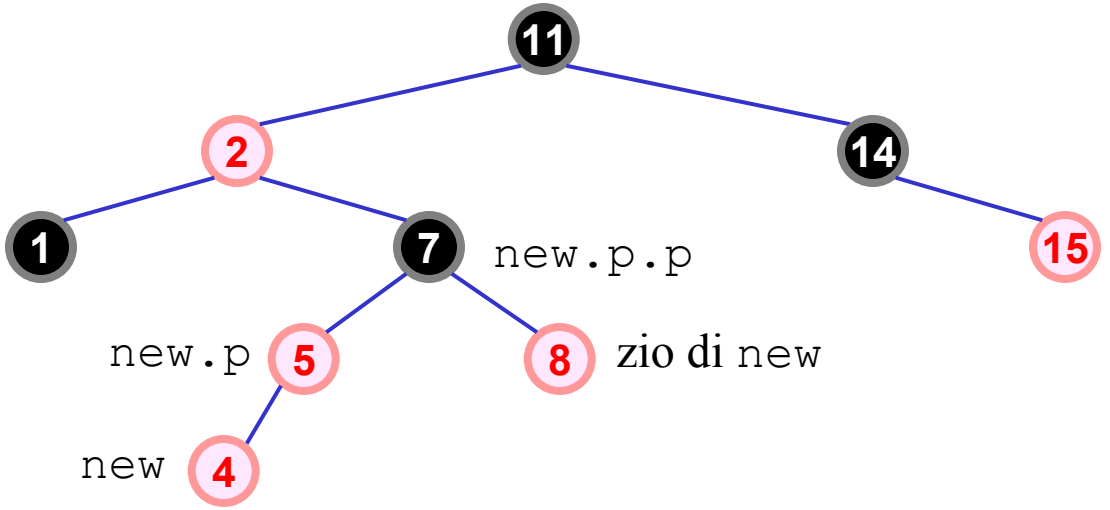


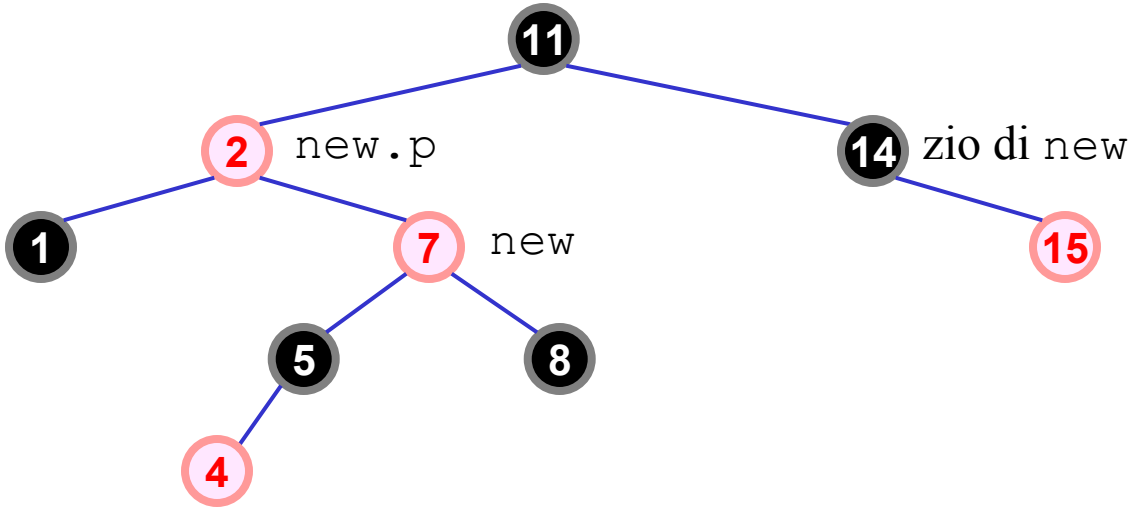
ricolorazione
di `new.p`, dello zio di
`new` e di `new.p.p`



iterazione:
`new = new.p.p`

ora `new` e `new.p`
potrebbero ancora
violare la regola 3
(ma `new` è più
vicino alla radice)





Violazione: caso 3

- Nel caso 3 `new` è più vicino alla radice ma potrebbe violare la regola 3 con `new.p`
- Occorre rilanciare la procedura con il nuovo `new`
- Il caso peggiore è quando si ha una sequenza di casi 3 fino a che non si risale alla radice
- Quando arriviamo alla radice questa diventa rossa
  - in questo caso è sufficiente ricolorare la radice di nero
  - questo equivale ad incrementare di uno il numero dei nodi in ogni cammino dalla radice al nodo `t.null`

Complessità di `RB_INSERT_FIXUP`

- Le violazioni nel caso 1 e 2 vengono risolte in tempo $\Theta(1)$
- Poiché l'albero è alto $\Theta(\log n)$, la procedura per risolvere una violazione nel caso 3 può essere rilanciata al massimo $\Theta(\log n)$ volte
- La complessità di `RB_INSERT_FIXUP`, e dunque di `RB_INSERT`, è $\Theta(\log n)$

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

1. 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?
2. 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

Esercizi

3. Mostra come un albero rosso-nero possa essere utilizzato per costruire una coda di priorità
   - come si può fare per accedere all'elemento minimo/massimo della coda?
   - qual è il costo delle operazioni di accesso, di cancellazione e di inserimento?