Algoritmi e strutture dati

Alberi red-black (RBT)

Menú di questa lezione

In questa lezione introduciamo una prima specializzazione degli alberi, chiamati alberi red-black, e studiamo le complessità delle operazioni ad essi associate.

Alberi red-black: introduzione

Un **albero red-black** (RBT) è un albero binario di ricerca (BST) **bilanciato** per costruzione. Possiede tutte le caratteristiche di un BST, ma la sua altezza è sempre $\Theta(log(n))$, dove n è il numero di elementi dell'albero. Un RBT, come un BST, è una struttura dati dinamica, basata sull'ordinamento, e sparsa. È ovvio che tutte le operazioni, ed in particolare la ricerca di un elemento, che funzionano in tempo proporzionale all'altezza diventano esponenzialmente piú efficienti su un RBT. La caratteristica principale di queste strutture è che l'inserimento e l'eliminazione mantengono la proprietá di bilanciamento; questa è a sua volta implicata da una serie di proprietá che andremo a dettagliare e che dovremo mantenere.

Alberi red-black: introduzione

Rudolf Bayer è noto per l'introduzione degli alberi red-black, ma anche degli alberi B, che vedremo nel prossimo blocco. L'introduzione formale risale al 1972.

Alberi red-black: introduzione

Ogni nodo in un RBT ha un'informazione in piú rispetto a un nodo in un BST: oltre a un puntatore al padre, i puntatori ai due figli, e la chiave, abbiamo il **colore** ($x.color$), che per convenzione è rosso o nero . Inoltre ogni foglia (ogni nodo senza figli) possiede due figli **virtuali**, che non contengono chiave e sono sempre di colore nero. Il padre della radice, per convenzione, è anche lui un nodo virtuale senza chiave, senza figli, e di colore nero.

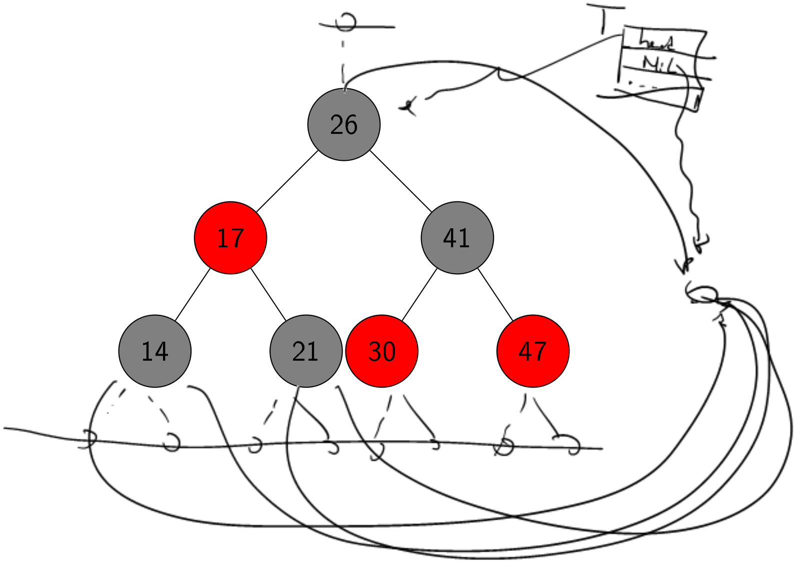
Alberi red-black: introduzione

Le regole che ogni albero rosso-nero deve rispettare, in aggiunta alla proprietá di ordinamento dei BST, sono:

1. Ogni nodo é rosso o nero;
2. La radice è nera; VIRTUALE
3. Ogni foglia (esterna, **nil**) è nera;
4. Se un nodo è rosso, entrambi i suoi figli sono neri;
5. Per ogni nodo, tutti i percorsi semplici da lui alle sue foglie, contengono lo stesso numero di nodi neri.

Chiameremo i nodi di un albero RB **interni**, per distinguerli dai nodi **esterni** che aggiungiamo in maniera artificiale a ogni albero RB. Una foglia esterna è un nodo che ha tutte le propietà di ogni altro nodo ma non porta alcuna chiave, ed è sempre di colore nero. Quindi ogni nodo interno ha di un RBT ha sempre due figli (che possono essere entrambi esterni o uno solo dei due), ed ogni nodo esterno non ha figli. Dal punto di vista implementativo, definiamo una sentinella $T.Nil$ (un campo aggiuntivo di T) come un nodo con tutte le proprietá di un nodo di T, e colore fissato a nero, per il ruolo di foglia esterna.

Alberi red-black: introduzione

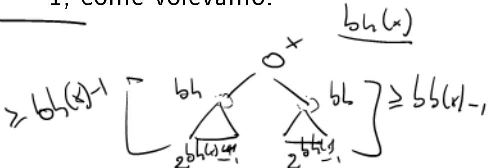


Alberi red-black: introduzione

Osserviamo che nell'esempio anteriore le foglie esterne non sono state visualizzate. Il principio fondamentale dei RBT è che le proprietá sono valide quando l'albero è vuoto e vengono mantentute tali dopo ogni inserimento e eliminazione. Dobbiamo ancora dimostrare che esse garantiscono il bilanciamento dell'albero - a meno di una costante. Cominciamo definendo l'**altezza nera** ($bh(x)$) di un nodo x in T come il numero di nodi neri su qualsiasi percorso semplice da x (senza contare x) a una foglia esterna (contandola). Si noti che è una buona definizione, grazie alla proprietá 5 ($bh(x)$ è sempre la stessa considerando qualsiasi percorso semplice). L'**altezza nera** di T è $bh(T.root)$.

$$2^{bh(x)-1}-1+2^{bh(x)-1}-1+1=2^{bh(x)}-1$$

Adesso dimostriamo che se T è un RBT con n nodi interni (quindi escludendo le foglie esterne), allora la sua altezza massima è $2 \cdot log(n+1)$. A questo fine, mostriamo, prima, che il sotto-albero radicato in x contiene almeno $2^{bh(x)}-1$ nodi interni, per induzione. Quando $bh(x)$ è 0, allora, per definizione, $x = T.Nil$, e il sotto-albero radicato in x non ha nodi interni; l'altezza nera di x è 0 (perchè non si include il nodo stesso), ed abbiamo che $2^{bh(x)}-1=1-1=0$, come volevamo. Se $bh(x)$ è positiva, allora l'altezza nera di entrambi i suoi figli è almeno $bh(x)-1$. Per ipotesi induttiva ognuno dei due sotto-alberi ha almeno $2^{bh(x)-1}-1$ nodi interni. Quindi il sotto-albero radicato in x ha almeno $2\cdot(2^{bh(x)-1}-1)+1$ nodi interni, che è esattamente $2^{bh(x)}-1$, come volevamo.



Consideriamo adesso T di altezza h. Per la proprietá 4, almeno la metá dei nodi dalla radice (esclusa) ad una foglia su qualsiasi ramo è nera. Quindi $bh(T.root) \geq \frac{h}{2}$. Dalla proprietá precedente, il numero n di nodi in T è $n \geq 2^{bh(T.root)} - 1$, cioè $n \geq 2^{\frac{h}{2}} - 1$.

Quindi:

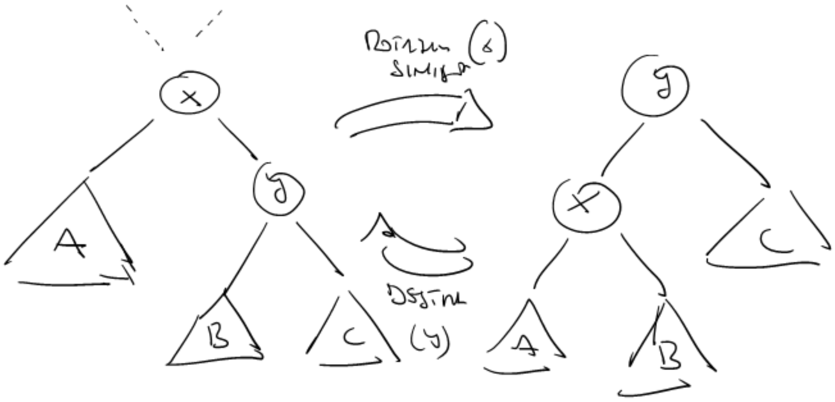
n	$\geq 2^{\frac{h}{2}} - 1$	risultato precedente
$n+1$	$\geq 2^{\frac{h}{2}}$	calcolo algebrico
$log(n+1)$	$\geq \frac{h}{2}$	prop. logaritmi
h	$\leq 2 \cdot log(n+1)$	tesi

Un albero binario completo (guardiamo solo i nodi interni) ha altezza h sempre maggiore o uguale a $log(n) - 1$, dove n è il numero di nodi totali. Pertanto, $log(n) - 1 \leq h \leq 2 \cdot log(n+1)$, cioè $h = \Theta(log(n))$.

Alberi red-black: rotazioni

Abbiamo giá capito che inserimento ed eliminazioni in un RBT possono violare le proprietá, e che la maggiore difficoltá nell'implementare queste procedure consiste precisamente nel modificare la struttura dell'albero per ripristinare queste proprietá. Un passo intermedio fondamentale per questa riparazione è la **rotazione**, che puó essere destra o sinistra, e che preserva la proprietá BST (non le proprietá RBT). L'idea è che possiamo ribilanciare l'albero e poi preoccuparci dei colori. Risolviamo il problema della **rotazione sinistra**: dato un RBT T, ed un nodo x in T, con figlio destro y, ottenere un nuovo albero T', dove y ha come figlio sinistro x. Simmetricamente, potremo definire il problema della rotazione destra. In entrambi i casi la complessitá é $\Theta(1)$.

IDEA INTUITIVA DELLA ROTAZIONE



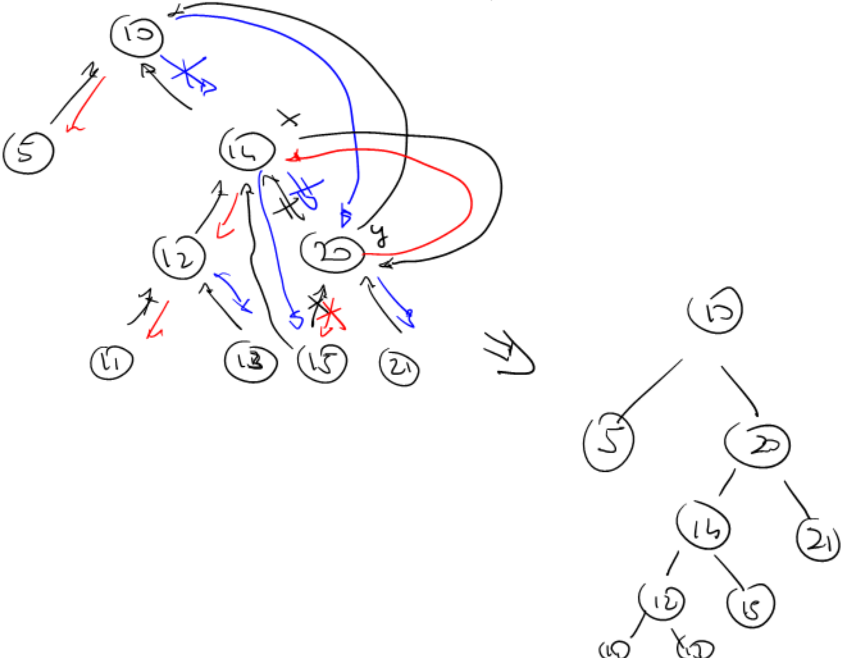
$A \leq x \leq B \leq y \leq C$

proc ***BSTTreeLeftRotate*** (T, x)

$$
\begin{cases}
y = x.right \\
x.right = y.left \\
\textbf{if } (y.left \neq T.Nil) \\
\quad \textbf{then } y.left.p = x \\
y.p = x.p \\
\textbf{if } (x.p = T.Nil) \\
\quad \textbf{then } T.root = y \\
\textbf{if } ((x.p \neq T.Nil) \textbf{ and } (x = x.p.left)) \\
\quad \textbf{then } x.p.left = y \\
\textbf{if } ((x.p \neq T.Nil) \textbf{ and } (x = x.p.right)) \\
\quad \textbf{then } x.p.right = y \\
y.left = x \\
x.p = y
\end{cases}
$$

Una volta capito come funzionano i cambi di puntatori, mostrare la **correttezza** delle rotazioni è immediato. Inoltre, si vede subito che la **complessità** è costante in entrambi i casi destro e sinistro.

LEFT ROTATE (T, x)



Alberi red-black: inserimento

Risolviamo adesso il problema di inserire un nodo z in un RBT T in maniera da mantenere tutte le proprietá di T. Chiaramente se usiamo *BSTTreeInsert* cosí com'é, abbiamo la garanzia che la proprietá BST sia rispettata. Se il nodo inserito è colorato rosso, allora anche la proprietá 5 è rispettata; inoltre, poiché z sarà sempre una nuova foglia, inserendo correttamente le sue foglie esterne, garantiamo anche la proprietá 3. La proprietá 1 è rispettata semplicemente assegnando il colore (rosso) a z. Quindi, solo due proprietá possono essere violate: se z diventa la radice, allora **violiamo 2**, se, invece, z diventa figlio di un nodo rosso, allora **violiamo 4**.

Alberi red-black: inserimento

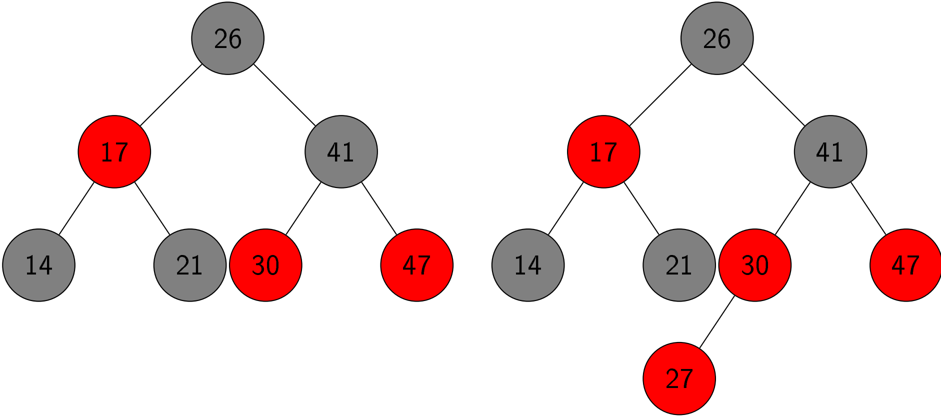
BST Insert *

```
proc RBTreeInsert (T, z)
  y = T.Nil
  x = T.root
  while (x ≠ T.Nil)
    y = x
    if (z.key < x.key)
      then x = x.left
      else x = x.right
  z.p = y
  if (y = T.Nil)
    then T.root = z
  if ((y ≠ T.Nil) and (z.key < y.key)
    then y.left = z
  if ((y ≠ T.Nil) and (z.key ≥ y.key)
    then y.right = z
  z.left = T.Nil
  z.right = T.Nil
  z.color = RED
  RBTreeInsertFixup(T, z)
```

* Due e Nil si vede anche T.Nil

Alberi red-black: inserimento

Nell'albero di esempio, inseriamo z con chiave 27, ottenendo (prima dell'esecuzione di *RBTreeInsertFixup*) una violazione della propietá 4:



Alberi red-black: inserimento

RISULTATO PADRE-ROSSO di z

proc ***RBTreeInsertFixup*** (T, z)

- **while** $(z.p.color = RED)$
  - **if** $(z.p = z.p.p.left)$
    - **then** $RBTreeInsertFixUpLeft(T, z)$
    - **else** $RBTreeInsertFixUpRight(T, z)$
- $T.root.color = BLACK$

6

PATCH

Alberi red-black: inserimento

[illegible]

```
proc RBTreeInsertFixupLeft (T, z)
  y = z.p.p.right
  if (y.color = RED)
    then
      z.p.color = BLACK
      y.color = BLACK
      z.p.p.color = RED
      z = z.p.p
    else
      if (z = z.p.right)
        then
          z = z.p
          TreeLeftRotate(T, z)
      z.p.color = BLACK
      z.p.p.color = RED
      TreeRightRotate(T, z.p.p)
```

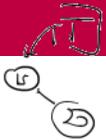
caso 1

caso 2

caso 3

```
proc RBTreeInsertFixupRight (T, z)
  y = z.p.p.left
  if (y.color = RED)
    then
      z.p.color = BLACK
      y.color = BLACK
      z.p.p.color = RED
      z = z.p.p
    else
      if (z = z.p.left)
        then
          z = z.p
          TreeRightRotate(T, z)
      z.p.color = BLACK
      z.p.p.color = RED
      TreeLeftRotate(T, z.p.p)
```

Alberi red-black: correttezza dell'inserimento



La scelta che si fa all'inizio di *RBTreeInsertFixup* genera 2 casi, che dipendono dal fatto che $z.p$ sia figlio destro o sinistro di $z.p.p$. All'interno di ogni caso vi sono tre sotto-casi, che si distinguono dal colore di y (lo **zio** di z): se è rosso, è un caso, e se è nero, allora, se z è figlio destro è un secondo caso, e se è figlio sinistro è un terzo caso. Il totale è quindi di 6 casi, i primi tre completamente simmetrici ai secondi tre. Osserviamo che se z è la radice (abbiamo inserito un nodo in un albero vuoto), allora $z.p = T.Nil$, e $T.Nil.color = BLACK$: quindi la condizione del ciclo **while** è corretta e determina un corretto caso di terminazione. Similmente, se z è un figlio diretto della radice, allora $z.p$ è la radice, e quindi $z.p.p = T.Nil$, e pertanto $z.p.p.left$ e $z.p.p.right$ sono entrambi Nil e diversi da $z.p$: quindi tutte le condizioni **if** sono ben definite.

Alberi red-black: correttezza dell'inserimento

Analizziamo adesso il codice. L'idea di fondo è: se esiste un problema dopo l'inserimento (violazione della proprietá 2 o della proprietá 4), questo si **spinge verso l'alto** con il caso 1, finchè è possibile. Quando non é piú possibile, si salta al caso 2 (immediatamente convertito al caso 3) o al caso 3: una rotazione risolve il problema in forma definitiva e garantisce l'uscita dal ciclo (**terminazione**). Per mostrare la **correttezza**, usiamo la seguente **invariante**: z é rosso, se $z.p$ è la radice, allora è nera, e se T víola qualche proprietá, allora ne víola esattamente una, che è la 2 o la 4 (se é la 2, è perché z è la radice ed è rossa, se é la 4, è perché z e $z.p$ sono entrambi rossi). La condizione di uscita (nei tre casi) è che $z.p$ è di colore nero; quindi l'invariante sommata alla condizione di uscita piú l'ultima istruzione di *RBTreeInsertFixup* ci dá la correttezza.

Alberi red-black: correttezza dell'inserimento

Sappiamo che T è un RBT legale prima di chiamare *RBTreeInsertFixup*. Per quanto riguarda l'**inizializzazione**, dobbiamo mostrare che l'inviariante è vera prima di chiamare *RBTreeInsertFixup*. Osserviamo, prima di tutto, che z viene inserito rosso. Inoltre, se $z.p$ è la radice, allora $z.p$ era nera (perché T è legale) e prima di chiamare *RBTreeInsertFixup* questo non è cambiato. Infine, giá sappiamo che le proprietá 1,3, e 5 non sono violate alla chiamata di *RBTreeInsertFixup*. Se T víola 2, deve essere perché z è la radice (e T era vuoto prima dell'inserimento); ma in questo caso $z.p = z.left = z.right = T.Nil$ sono tutti nodi neri, perció la proprietá 4 non è violata e la violazione della 2 è l'unica. Se invece T víola 4, poiché $z.left = z.right = T.Nil$ sono neri, e il resto di T non ha violazioni, deve essere perché $z.p$ è rosso come z.

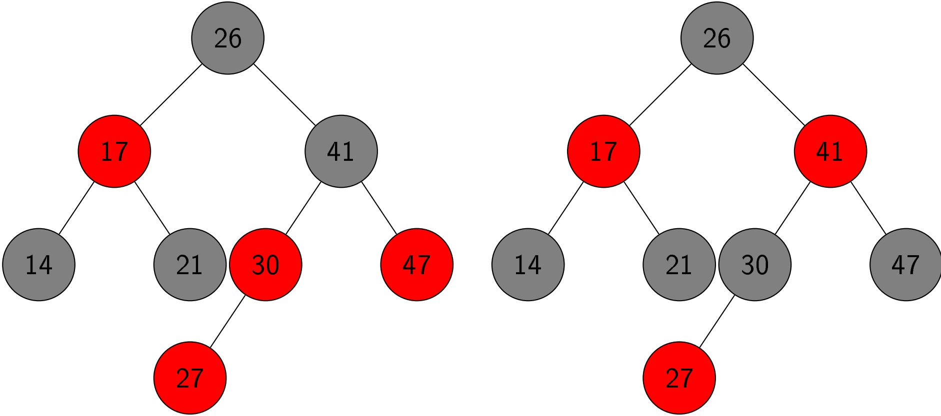
Alberi red-black: correttezza dell'inserimento

Cosa accade dopo la fine dell'inserimento? Come abbiamo detto, al termine della procedura $z.p$ è nero. Quindi la proprietá 4 è rispettata al termine. Se al termine del ciclo la proprietá 2 è violata, l'ultima linea del codice la ripristina. Ci rimane da dimostrare che l'invariante è mantentuta da un ciclo al seguente. Come abbiamo visto ci sono sei casi da analizzare. Ne analizziamo tre, assumendo che $z.p$ è figlio sinistro di $z.p.p$ (che esiste: infatti, se z è la radice, allora $z.p = T.Nil$ è nero, e il ciclo non si esegue). Stiamo quindi assumendo che si esegue *RBTreeInsertFixupLeft*.

Alberi red-black: correttezza dell'inserimento

Caso 1: lo zio y di z è rosso. Poiché $z.p.p$ è nero (per ipotesi), coloriamo di nero sia $z.p$ che y, e coloriamo di rosso $z.p.p$, per mantenere la propietá 5. Adesso $z.p.p$ diventa z (quindi spositamo il potenziale problema un passo piú in alto). Dobbiamo mostrare che il nuovo z è tale che l'inviariante è mantenuta. Prima di tutto, z è rosso (era $z.p.p$ prima, e lo coloriamo rosso); poi, $z.p$ (vecchio $z.p.p.p$) non cambia colore, quindi, se è la radice, è rimasta nera; infine, le proprietá 1 e 3 non sono a rischio, e sappiamo giá che 5 è mantenuta: se (il nuovo) z è la radice, allora è rossa, e si víola 2, ma solo 2, giacché $z.p = T.Nil$ è nero, se (il nuovo) z non è la radice allora solo 4 puó essere ancora violata e grazie alle altre ipotesi ed alla correzione nel ciclo eseguito, questa violazione è dovuta a che (il nuovo) $z.p$ è rosso.

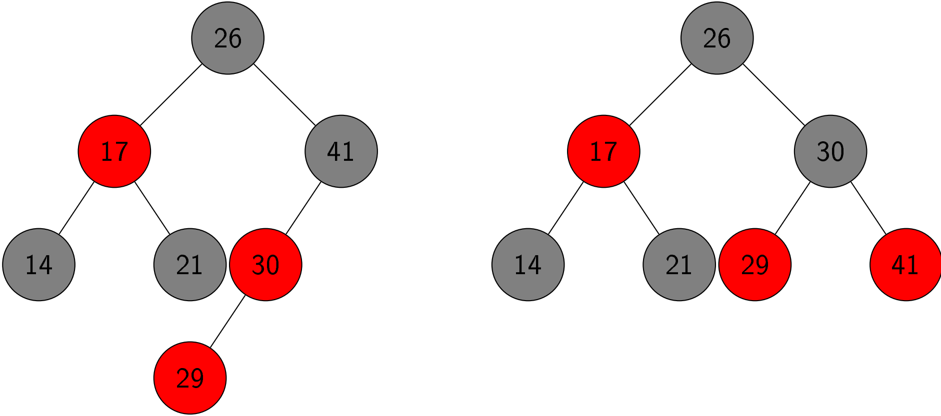
Esempio di esecuzione di *RBTreeInsertFixup*, caso 1:



Caso 3: lo zio y di z è nero e z è figlio sinistro di suo padre. Il caso 2 (z è figlio destro di suo padre) si riporta immediatamente al caso 3 attraverso una rotazione ed un ricoloramento. Il nodo $z.p$ diventa nero, ed il nodo $z.p.p$ diventa rosso. Sappiamo giá che entrambi i nodi esistono. La rotazione a destra su $z.p.p$ ripristina la proprietá 5. Ci rimande da mostrare che z (che non è cambiato) è tale che l'inviariante è mantenuta: prima di tutto, z è (ancora) rosso; poi, se $z.p$ è la radice, è diventata nera (se non lo era gía); infine, le proprietá 1 e 3 non sono a rischio, e sappiamo giá che 5 è mantenuta; inoltre in questo caso la proprietá 2 non si puó violare. La unica violazione alla proprietá 4 (z e $z.p$ entrambi rossi) viene corretta, e non ci sono altre violazioni.

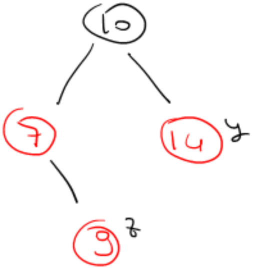
Alberi red-black: inserimento

Esempio di esecuzione di *RBTreeInsertFixup*, caso 3:



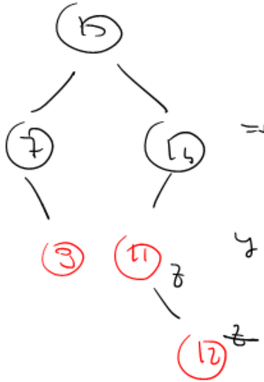
Esercizio 10

10, 7, 14, 9, 11, 12



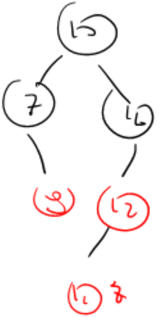
FixUpLeft

caso 1



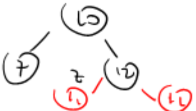
FixUpLeft

caso 2



caso 3

y



11

Alberi red-black: correttezza dell'inserimento

Concludendo l'analisi dell'inserimento (che ha **complessità**, nel caso peggiore, $\Theta(h) = \Theta(log(n))$: nel peggior caso si esegue tutto il ciclo **while** seguendo il caso 1, e si percorre un ramo intero), osserviamo che il caso 1 si verifica in un RBT previamente bilanciato, e si sistemano i colori per mantenere, al peggio, un leggero sbilanciamento. I casi 2 e 3, invece operano su un RBT giá leggermente sbilanciato: ma questo si rivela facile da sistemare grazie a, al massimo, due rotazioni, e poi un ricoloramento sistematico. Questo complesso sistema ci permette di mantenere una struttura bilanciata anche quando si opera un inserimento di elementi in ordine, che, invece, genererebbe un BST molto sbilanciato. Come si puó intuire, l'eliminazione di un nodo da un RBT è **molto** complessa, e non la vediamo. Anche questa operazione ha costo $\Theta(h) = \Theta(log(n))$.

Alberi binari di ricerca, red-black e liste: confronto

	Liste	BST c. medio	BST c. peggiore	RBT c. peggiore
Inserimento	$\Theta(1)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$
Cancellazione	$\Theta(1)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$
Visita	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Ricerca	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$
Successore	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$
Predecessore	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$
Massimo	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$
Minimo	$\Theta(n)$	$\Theta(log(n))$	$\Theta(n)$	$\Theta(log(n))$

Conclusione

Gli alberi red-black sono una struttura complessa. Non sono gli unici alberi bilanciati; altri esempi includono gli alberi di Fibonacci e gli alberi AVL. Tutte le strutture bilanciate sono simili tra loro, e l'uso di una o dell'altra dipende da dettagli che noi non riusciamo ad evidenziare in questa sede.