Svantaggio di un BST

In un BST classico, le operazione sono proporzionale all'altezza dell'albero.
Nel caso peggiore, dunque, le operazioni possono avere costo **lineare**.
Per ovviare a questo problema, definiamo un nuovo tipo di albero che si bilancia da solo in maniera tale da minimizzare la sua altezza.
Questo tipo di albero viene chiamato **Albero Rosso-Nero**.

Alberi Rosso-Neri

Sono **alberi binari di ricerca** dove ogni nodo è colorato di rosso o di nero.

Proprietà

1. Ogni nodo è o rosso o nero
2. La radice è nera
3. Tutte le foglie(nil) sono nere
4. Se un nodo è rosso, i suoi figli sono neri
5. Per ogni nodo x, tutti i cammini da x ad una foglia hanno lo stesso numero di nodi neri
    - Definiamo quindi $bh(x)$ l'altezza nera di x, ovvero il numero di nodi neri su un ramo da x ad una foglia(x escluso)

Inoltre

> **L'altezza massima di un Red-Black Tree con n nodi è $2\log_2(n+1)$

Dimostrazione:

Dim. Consideriamo un albero R-N con n nodi interni.

- dimostriamo che per ogni nodo x il sottoalbero con radice in x ha almeno $2^{\mathrm{bh}(x)} - 1$ nodi, procediamo con induzione sull'altezza di x:
  - *caso base*: se $\mathrm{bh}(x) = 0$ (x è una foglia) allora è vero perché $2^0 - 1 = 0$
  - se x non è foglia allora i figli hanno altezza nera $\mathrm{bh}(x)$ oppure $\mathrm{bh}(x) - 1$ secondo il colore che hanno
  - *passo induttivo:* secondo l'ipotesi induttiva ogni figlio ha almeno $2^{(\mathrm{bh}(x)-1)} - 1$ nodi interni nel proprio sottoalbero
  - quindi almeno $2\left(2^{(\mathrm{bh}(x)-1)} - 1\right) + 1 = 2^{\mathrm{bh}(x)} - 1$ nodi interni nel sottoalbero con radice in x
- in un albero con altezza h, l'altezza nera della radice r soddisfa per via della regola del rosso

$$\mathrm{bh}(r) \geq \frac{h}{2}$$

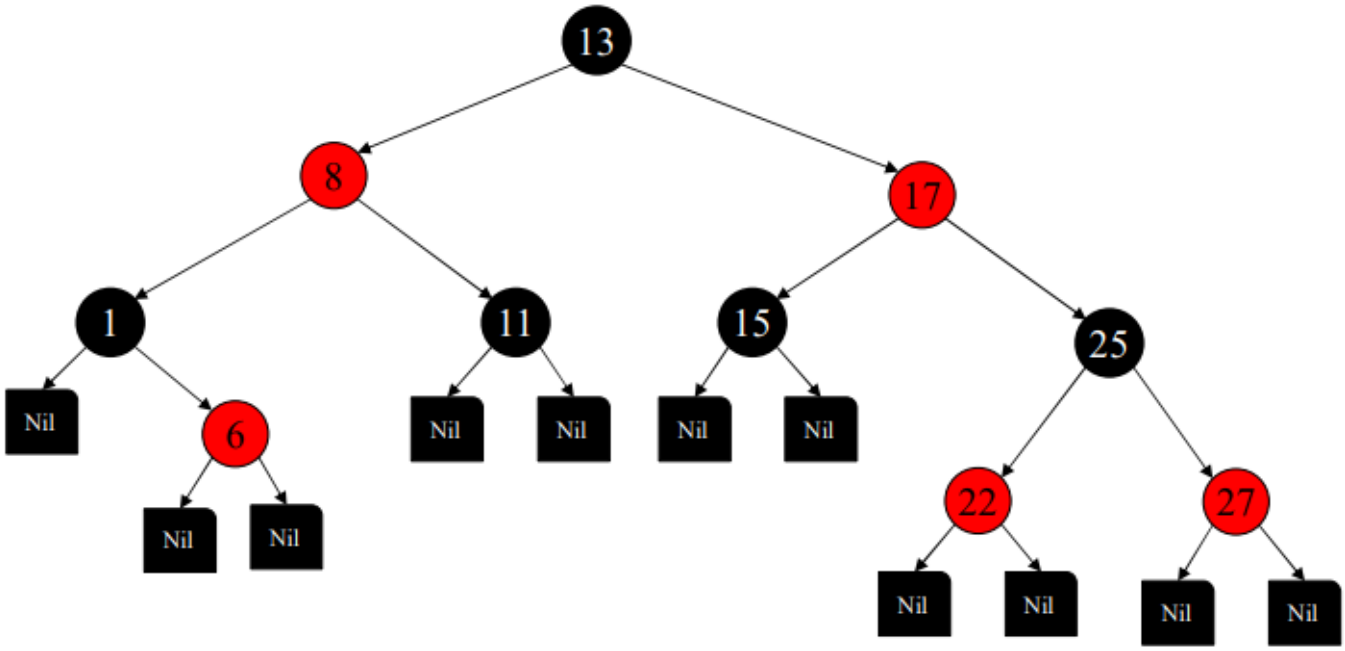
(perché lungo un cammino al massimo metà dei nodi può essere rosso)

- quindi:

$$\begin{aligned} n &\geq 2^{h/2} - 1 \\ n + 1 &\geq 2^{h/2} \\ \log_2(n+1) &\geq \log_2(2^{h/2}) = h/2 \\ 2\log_2(n+1) &\geq h \end{aligned}$$

Dal momento che in un albero, la ricerca è proporzionale all'altezza($O(h)$), possiamo dunque concludere che in un **Red-Black Tree** la ricerca è logaritmica.

Esempio di Red-Black Tree



In questa figura è chiaro come vengano rispettate le proprietà sopra menzionate

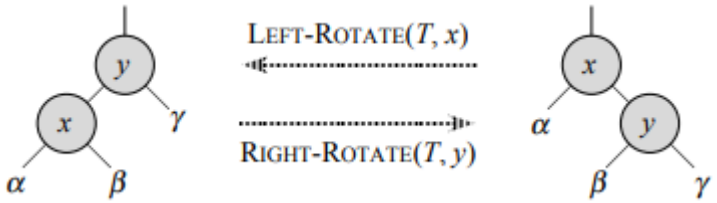
Inserimento e Cancellazione in un Red-Black Tree

Entrambe le operazioni hanno costo **logaritmico** ($O(\log_2 n)$) dal momento che scendono lungo un ramo e risalgono fino alla radice. Queste operazioni sono molto simili a quelle di un **BST** classico, tuttavia durante l'inserimento(o cancellazione) di un elemento, potrebbe venir **violata** una delle proprietà del Red-Black Tree.

Dunque bisogna gestire ulteriormente il caso in questo avvenga e dunque ripristinare tali proprietà.

Rotazione

Un'operazione utile per il ripristing delle proprietà, nel caso in cui vengano violate, è la **rotazione**.
La **rotazione** è una operazione locale in un albero che consiste nel *ruotare* a sinistra(o destra) un nodo.



Left-Rotate trasforma l'albero dalla configurazione di destra alla configurazione di sinistra. Analogamente per il Right-Rotate, si parte dalla configurazione di sinistra per arrivare a quella di destra

```
LEFT-ROTATE(T, x)
 1  y = x.right              // set y
 2  x.right = y.left         // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4      y.left.p = x
 5  y.p = x.p                // link x's parent to y
 6  if x.p == T.nil
 7      T.root = y
 8  elseif x == x.p.left
 9      x.p.left = y
10  else x.p.right = y
11  y.left = x               // put x on y's left
12  x.p = y
```

Il codice per Right-Rotate è speculare a questo.
Si noti che ha un costo costante dal momento che bisogna cambiare soltanto i puntatori per modificare la parentela

Inserimento

Procediamo con l'inserimento come se fosse un classico **BST.**
Il nuovo nodo che inseriamo sarà colorato di **rosso.**

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == T.nil
10        T.root = z
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

Tuttavia, come abbiamo già detto, durante l'inserimento potremmo violare una delle proprietà del Red-Black Tree.
Per ovviare a questo problema, alla fine dell'inserimento, chiamiamo il metodo ausiliario **RB-INSERT-FIXUP(T,z)** il cui scopo è, come suggerisce il nome, di *riparare* il Red-Black Tree affinchè valgano ancora le sue proprietà.

Le proprietà che possono essere violate sono due:

- *Proprietà 2: La radice è nera*
- *Proprietà 4: Se un nodo è rosso, i suoi figli sono neri*

La seconda proprietà viene violata nel caso in cui il nodo che inseriamo sia la radice(quando inseriamo un nuovo nodo lo coloriamo di rosso).

La quarta proprietà invece viene violata se inseriamo un nodo con parent un altro nodo rosso.

```
RB-INSERT-FIXUP(T, x)                ▷ x è il nodo che può creare problemi
  while x.p.color = red do
      if x.p = x.p.p.left then                ▷ padre di x è figlio sinistro?
          u ← x.p.p.right                          ▷ u è lo zio di x
          if u.color = red then                          ▷ caso 1?
              x.p.color ← black                          ▷ caso 1
              u.color ← black                            ▷ caso 1
              x.p.p.color ← red                          ▷ caso 1
              x ← x.p.p                                  ▷ caso 1
          else
              if x = x.p.right then                      ▷ caso 3?
                  x ← x.p                                ▷ caso 3
                  LEFT-ROTATE(T, x)                      ▷ caso 3
              x.p.color ← black                      ▷ caso 2 e 3
              x.p.p.color ← red                      ▷ caso 2 e 3
              RIGHT-ROTATE(T, x.p.p)                 ▷ caso 2 e 3
      else
          {tutto il corpo del if esterno con
          left e right scambiati}          ▷ padre di x è figlio destro
  T.root.color ← black
```

Denotiamo x come il nodo da inserire.

Caso 1

Lo zio di x è rosso(e pure il padre lo è, altrimenti nessuna proprietà verrebbe violata).
In questo caso basta colorare di **nero** il padre e lo zio.
Localmente, da x in giù, le proprietà vengono rispettate.

Caso 2

Lo zio di x è nero e x è figlio destro.
In questo caso basta ruotare a sinistra sul padre di x, colorare di nero il padre,colorare di rosso il nonno di x ed effettuare una rotazione a destra sul nonno.
Localmente, da x in giù, le proprietà vengono rispettate.
Se si verifica questo caso, il ciclo loop non reitera

Caso 3

Lo zio di x è nero e x è figlio sinistro.
In questo caso basta colorare di nero il padre,colorare di rosso il nonno di x ed effettuare una rotazione a destra sul nonno.
Localmente, da x in giù, le proprietà vengono rispettate.
Se si verifica questo caso, il ciclo loop non reitera

Cancellazione

Procediamo con la rimozione dell'elemento come se fosse un classico **BST.**
Tuttavia la rimozione dell'elemento potrebbe violare le proprietà del **Red-Black Tree**(le proprietà violate sono le stesse dell'**insert**).
Dunque procediamo a ripristinarle.

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

```
RB-TRANSPLANT(T, u, v)
1  if u.p == T.nil
2       T.root = v
3  elseif u == u.p.left
4       u.p.left = v
5  else u.p.right = v
6  v.p = u.p
```

Denotiamo con z il nodo da eliminare.
y come il successore di z
e x come

1. figlio di z se z ha un solo figlio
    - in questo caso x prende il posto di z
2. figlio di y di z se z ha due figli.

- il successore di z prende il posto di z e x prende il posto di y(il successore)

x mantiene il suo colore mentre y prende il colore di z.
Ci salviamo inoltre il colore del nodo perso.
Nel caso 1 sarà il colore di z, nel caso 2 quello di y

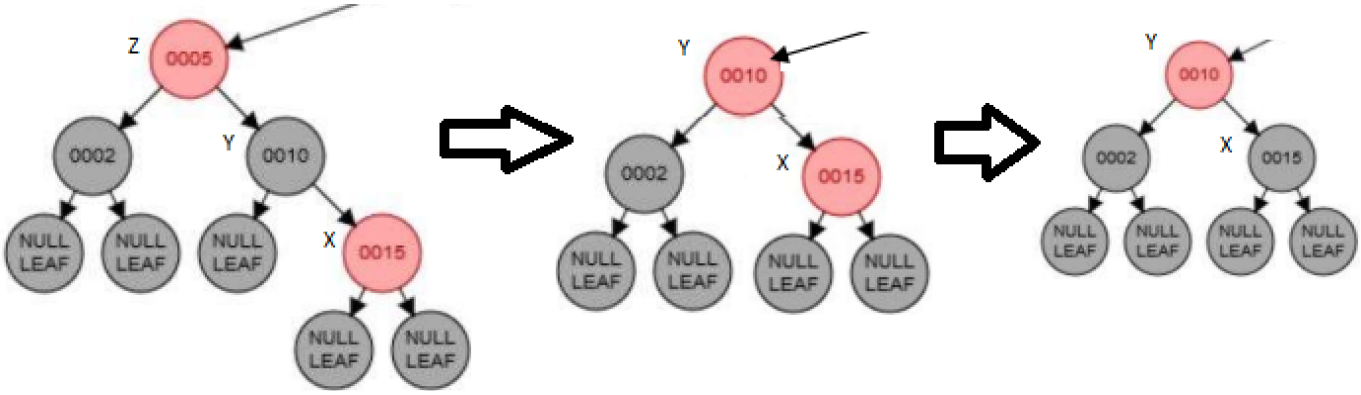
Per ripristinare le proprietà violate poi ci appoggiamo al metodo **RB-DELETE-FIXUP.**

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

Denotiamo ulteriormente w come fratello di x

Caso 0

Se x diventa il parent di y ed entrambi sono rossi, coloro x di nero.



Z è il nodo da eliminare

Caso 1

Caso 2

Caso 3

Caso 4

Edu mi ha convinto a non capire il delete del red-black-tree visto che non è in programma ed è complicato