Red Black Trees

Considerações gerais sobre árvores de busca e o seu balanceamento

1. Busca significa começar na raiz, e “descer” até o node correto.
    a. O(log n) se árvore é balanceada
2. Inserção deve buscar o novo dado e incluir nó na posição onde ele estaria.
    a. Caso contrário, o nó inserido não poderia ser buscado depois!
    b. E remoção precisa primeiro buscar o dado de qualquer forma!
3. Mas a inserção (remoção) pode desbalancear a árvore!
4. Portanto todo método de árvore balanceada deve:
    a. Definir propriedades que garantam o balanceamento
    b. Verificar eventuais violações destas propr. depois da inserção (remoção)
    c. Corrigir as violações
    d. Executar (b + c) em O (log n) tempo
        i. Pois inserções (remoções) devem executar em O (log n)

Propriedades de Red-Black Trees

1. Nodes are red or black.
2. Root ← black.
3. NIL node ← black.
4. A red node has 2 black children (The NIL node counts)
    a. Or, a red node can not have red children
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes (The NIL node counts).
    a. Definition of Black Height or bh()

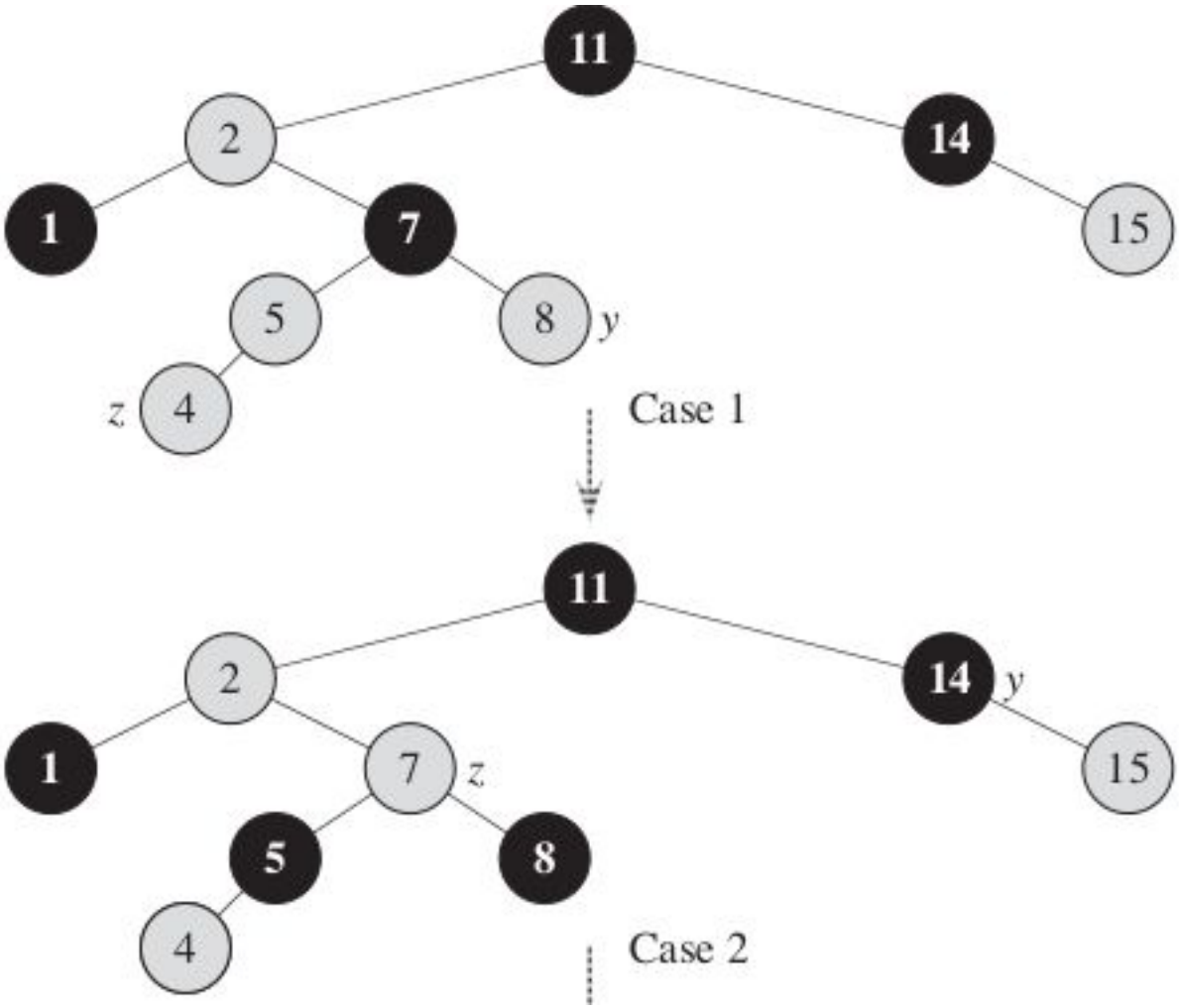
Comentarios informais

1. A maioria dos nós são Black.
   a. NIL, root, e subárvores não muito desbalanceadas
2. Nós Red “esticam” a subárvore, permitindo desbalanceamento até 2x
   a. Ou seja, ADIAM a correção do desbalanceamento, para baratear a correção.
3. Como Nó Red tem filho Black, um ramo é no max 2x o menor
   a. Aqui está a “mágica” que mantém O(log n) com constante 2
4. 1 rotação (simples ou dupla) por inserção
   a. propriedades são sempre mantidas antes da inserção, única violação é o novo nó
   b. Correções locais de cor, e talvez uma rotação p/ balancear sub-arvores
   c. AVL pode ter O(log n) rotações por inserção. Aqui está a vantagem da RB-Tree
5. “NIL nodes” existem e contam para facilitar ambos código **e** provas
   a. Menos ‘ifs’ e casos se ponteiros p/ filhos nulos existem (e pai da raiz é NIL também)
   b. Contas um pouco mais fáceis nas provas. Pelo menos foi assim que o Sedgevick fez.

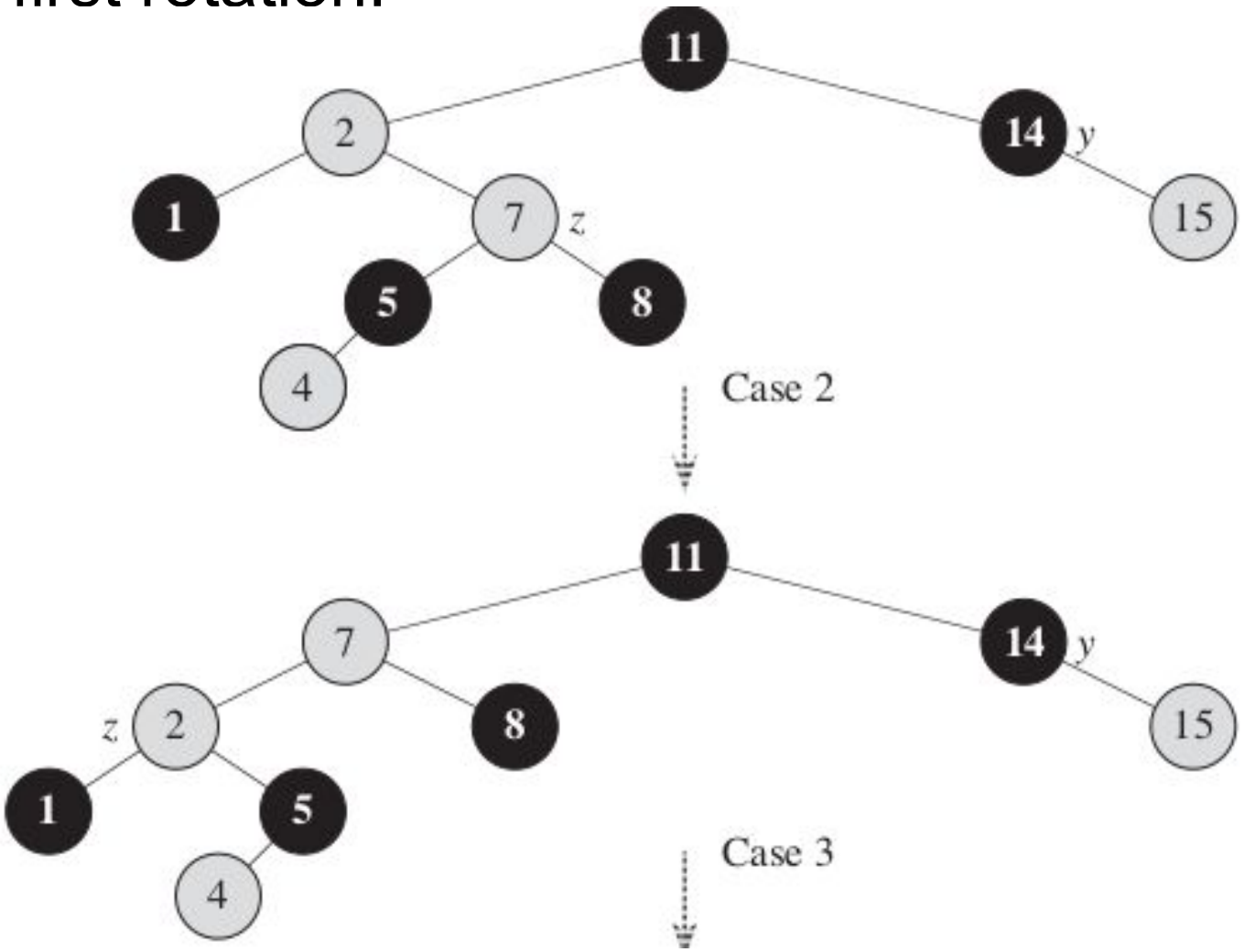
```
RB-INSERT-FIXUP(T,z) // z is the newly inserted node

 1.  while z.p.color == RED             // z red, father red: BAD
 2.     if z.p == z.p.p.left            // if father is left child
 3.        y = z.p.p.right              // uncle y is right child
 4.        if y.color == RED            // case 1 uncle y is red
 5.           z.p.color = BLACK         // case 1 father ← B
 6.           y.color   = BLACK         // case 1 uncle y ← B
 7.           z.p.p.color = RED         // case 1 grandpa ← R
 8.           z = z.p.p                 // loop with grandpa as new z
 9.        else if z == z.p.right       // z is right child
10.                z = z.p              // case 2 (2 stage rotation)
11.                LEFT-ROTATE(T,z)     // case 2
12.           z.p.color = BLACK         // case 3 (single rotation)
13.           z.p.p.color = RED         // case 3
14.           RIGHT-ROTATE(T,z.p.p)     // case 3 (loop ends here)
15.     else (same as then clause with “right” and “left” exchanged)
16.  T.root.color = BLACK // last step may leave RED root (BAD)
```

Case 1: Easy, uncle y (8) is red
Next loop, z is grandpa 7, and and uncle 14 is black



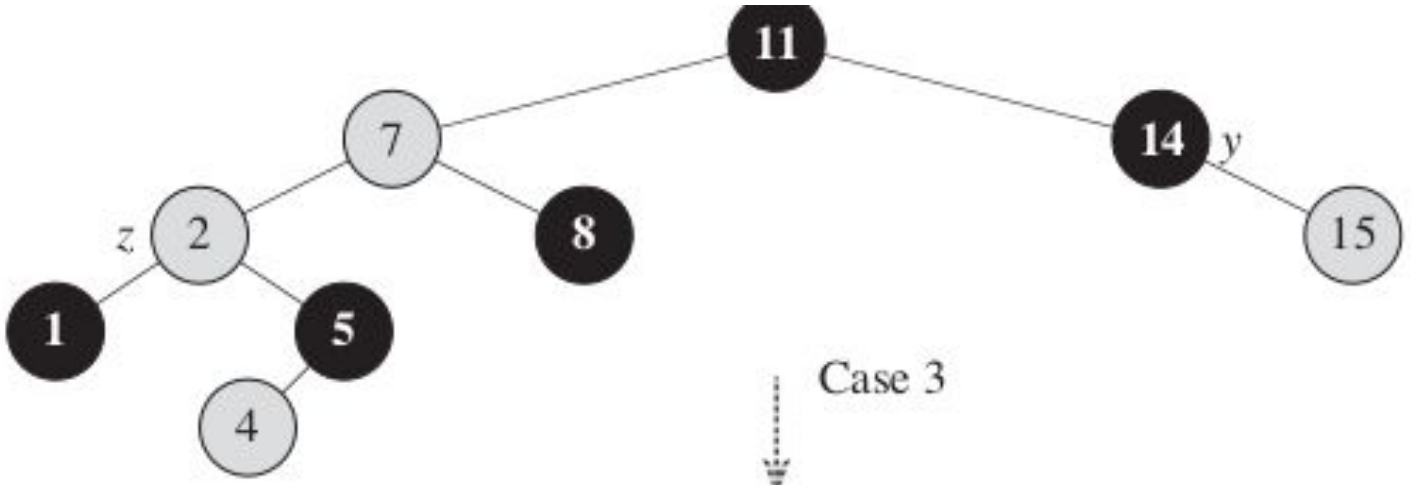
Uncle y is black, no easy fix: needs to rotate
Z is right child, it is double rotation
This is the first rotation:



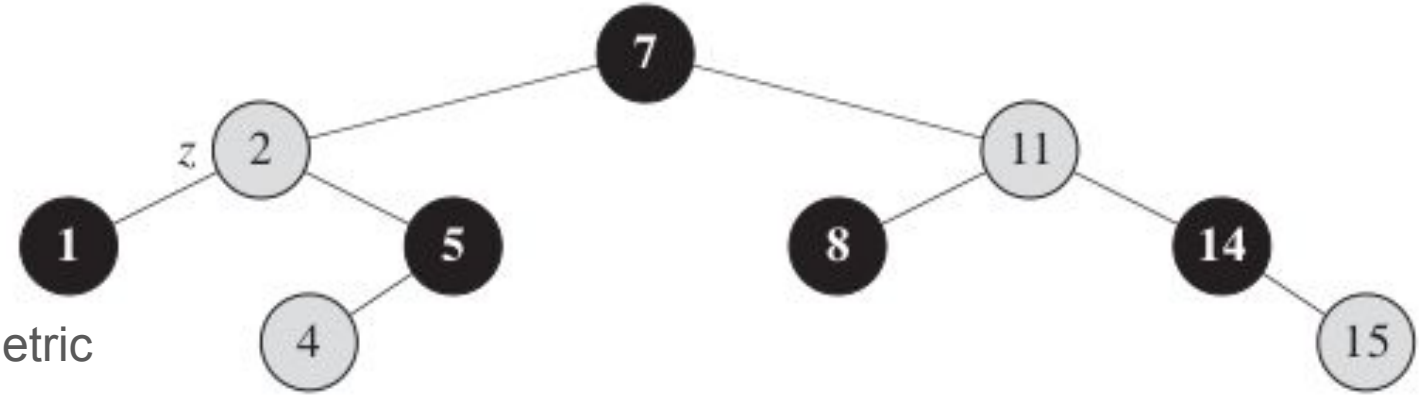
2 and 7 are still
RED father-child
(BAD)

This rotation fixes the RED father-son issue
Which was the lone violation of RB properties
The loop ends now because z.p is black!

(The 1st rotation is sometimes needed to setup this one).



Case 3



This is only half!

The other half is symetric

Loop invariants

1. Z is red
2. If z.p is root, then z.p is black
3. T may violate at most 1 property of RB-Tree (XOR)
   a. z is root and z is red
   b. z and z.p are red

- Importante: só há **uma** violação, que será corrigida por **uma** rotação .
- O(log n) pois loop no caso 1 pode chegar até a raiz
- O(log n) rotações são o custo principal das operações da árvore AVL.
- Note como NIL node facilita codigo:
  - linhas 1 e 2 , pai da raiz é NIL, e é Black
  - Se uncle y for NIL, o ponteiro não dá segfault e ele é Black

Links

- Visualization https://www.cs.usfca.edu/~galles/visualization/RedBlack.html
- Outras árvores para comparar
  - AVL https://www.cs.usfca.edu/~galles/visualization/AVLtree.html
  - B https://www.cs.usfca.edu/~galles/visualization/BTree.html
  - B+ https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html