Notes on Red-black Trees

Geoffrey Matthews

May 25, 2016

Red-black trees

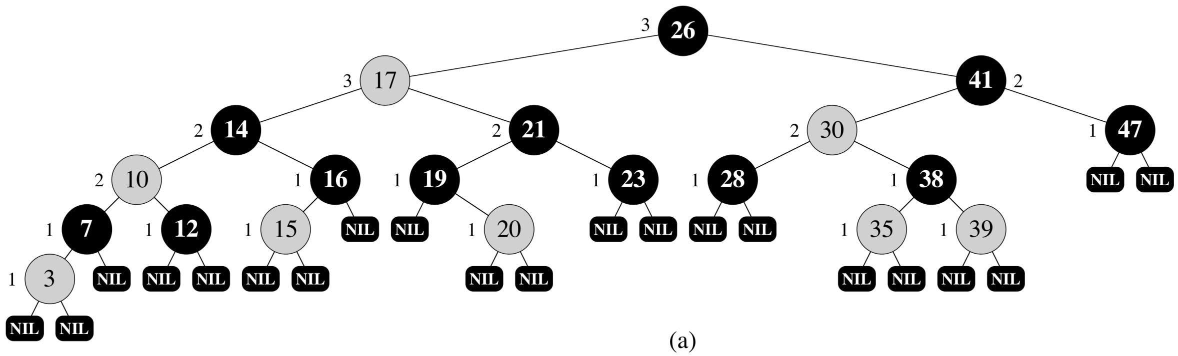
- A variation of binary search trees.
- **Balanced:** height is $O(\lg n)$, where n is number of nodes.
- Operations will take $O(\lg n)$ in worst case.

Red-black trees

- A **red-black tree** is a binary search tree.
- One bit per node stores an attribute *color*, red or black.
- All leaves are empty (nil) and colored black.
- We use a sentinel $T.nil$ for all the leaves of a red-black tree T.
- $T.nil.color$ is black.
- The root's parent is also $T.nil$.

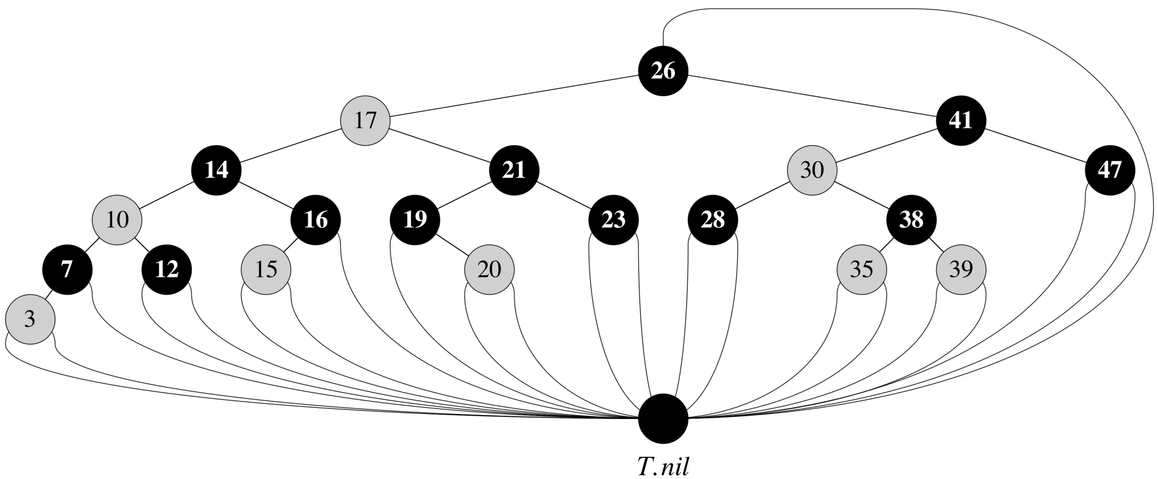
Red-black tree properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf ($T.nil$) is black.
4. If a node is red, then both its children are black
   - Hence no two reds in a row.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.



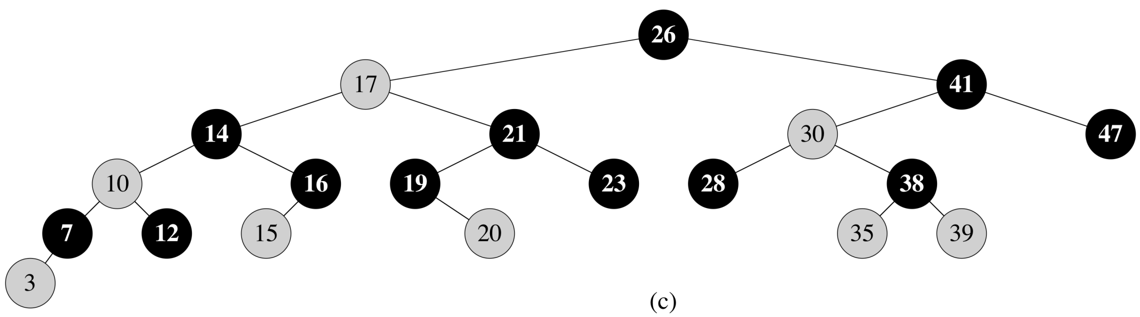
3 26
3 17 41 2
2 14 2 21 2 30 1 47
2 10 1 16 1 19 1 23 1 28 1 38 NIL NIL
1 7 1 12 1 15 NIL NIL 1 20 NIL NIL NIL NIL 1 35 1 39
1 3 NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL
NIL NIL

(a)



26
17 41
14 21 30 47
10 16 19 23 28 38
7 12 15 20 35 39
3
T.nil

(b)



26
17 41
14 21 30 47
10 16 19 23 28 38
7 12 15 20 35 39
3

(c)

Height of a red-black tree

- **Height of a node** is the number of edges in longest path to leaf.
- **Black-height** of a node x: $\mathrm{bh}(x)$ is the number of black nodes (including $T.nil$) on a path from x to a leaf, not counting x.
  - By property 5, black-height is well defined.
  - Changing the color of a node does not change its black-height.
  - Changing the color of a node will change the black-height of its ancestors.

Claim 1: Any node with height h has black-height $\geq h/2$.

Proof

- By property 4, $\leq h/2$ nodes on the path from node to a leaf are red.
- Hence $\geq h/2$ are black.

Claim 2: The subtree rooted at x contains $\geq 2^{bh(x)} - 1$ internal nodes.

Proof. By induction on height of x.
Basis: Height of $x = 0 \Rightarrow x$ is a leaf and so $bh(x) = 0$, $2^0 - 1 = 0$.
Inductive step:

- Let the height of x be h.
- Any child of x has height $h - 1$ and black-height either $bh(x)$ (if the child is red) or $bh(x) - 1$ (if the child is black).
- By inductive hypothesis, each child has $\geq 2^{bh(x)-1} - 1$ internal nodes.
- Thus, the subtree rooted at x contains $\geq 2 \cdot (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes.

Lemma: A red-black tree with n internal nodes and height h has

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

- Recall proven claims:
  - Any node with height h has black-height $\geq h/2$.
  - The subtree rooted at any node x contains $\geq 2^{bh(x)} - 1$ internal nodes.

Proof

Let h and b be the height and black-height of the root, respectively.
By the above two claims,

$$n \geq 2^b - 1 \geq 2^{h/2} - 1$$

Adding 1 to both sides and then taking logs gives

$$\lg(n+1) \geq h/2$$

which implies that

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

Operations on red-black trees

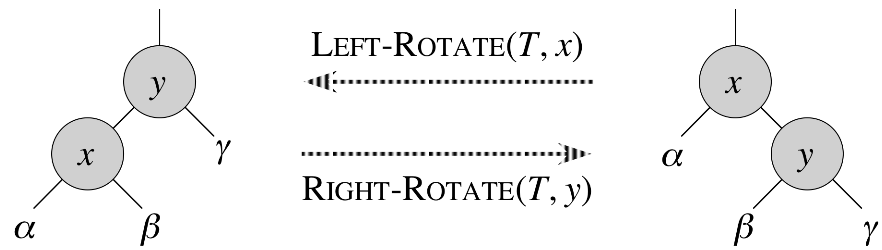
- MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR and SEARCH all run in $O(h) = O(\lg n)$ time.
- INSERT, what color to make the new node?
  - Red? Might violate property 4.
  - Black? Might violate property 5.
- DELETE, what color was the old node?
  - Red? OK.
    - Unless successor is black?
  - Black? Could cause two reds in a row, and violate properties 2 and 5.

Rotations

- Only pointers are changed.
- Won't upset binary-search-tree property.
- Doesn't care about red-black.

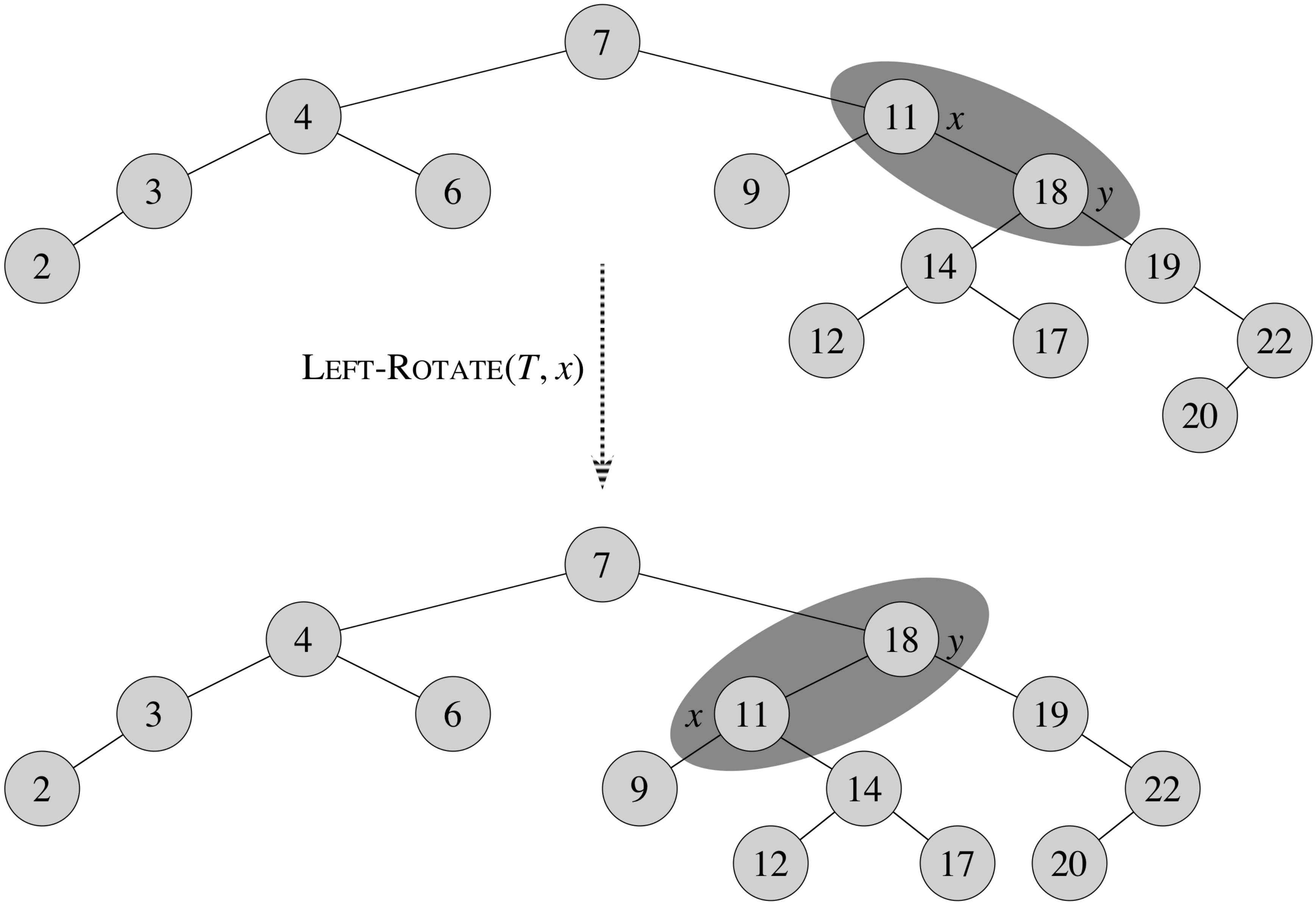
LEFT-ROTATE(T, x)

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
        y.left.p = x
y.p = x.p
if x.p == T.nil
        T.root = y
elseif x == x.p.left
        x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```



Assumes

- $x.right \neq T.nil$
- root's parent is $T.nil$

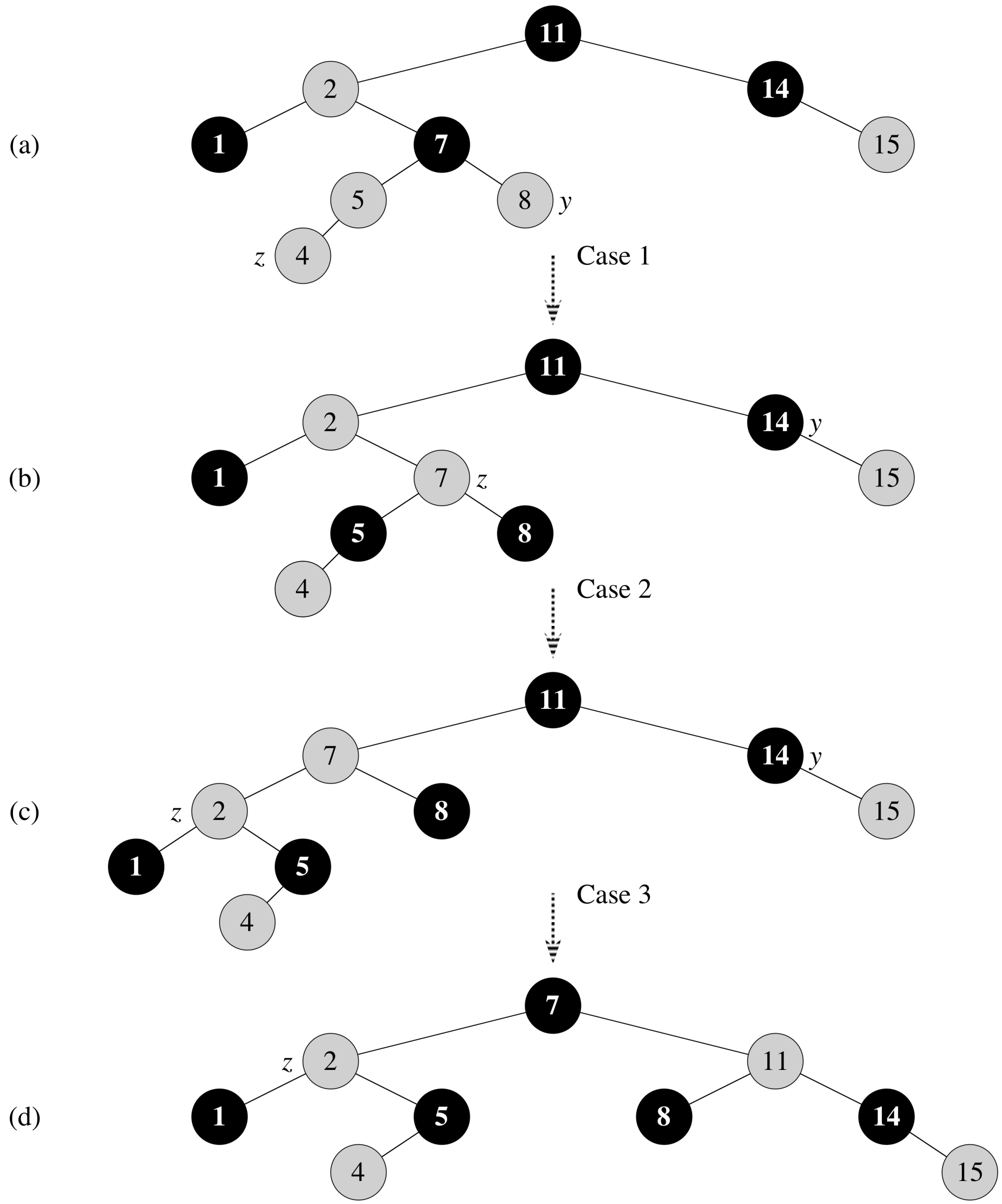

7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

Insertions

- Start by doing regular binary-tree insertion.
- Color new node red.
- May violate red-black tree properties:

1. Every node is either red or black.
   - OK.
2. The root is black.
   - New node might be root.
3. Every leaf ($T.nil$) is black.
   - OK.
4. If a node is red, then both its children are black.
   - New node's parent might be red.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
   - OK.

```
RB-INSERT-FIXUP(T, z)
while z.p.color == RED
    if z.p == z.p.p.left
        y = z.p.p.right
        if y.color == RED
            z.p.color = BLACK // case 1
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p //  case 2
                LEFT-ROTATE(T, z)
            z.p.color = BLACK // case 3
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then with "right" and "left" exchanged)
T.root.color = BLACK
```

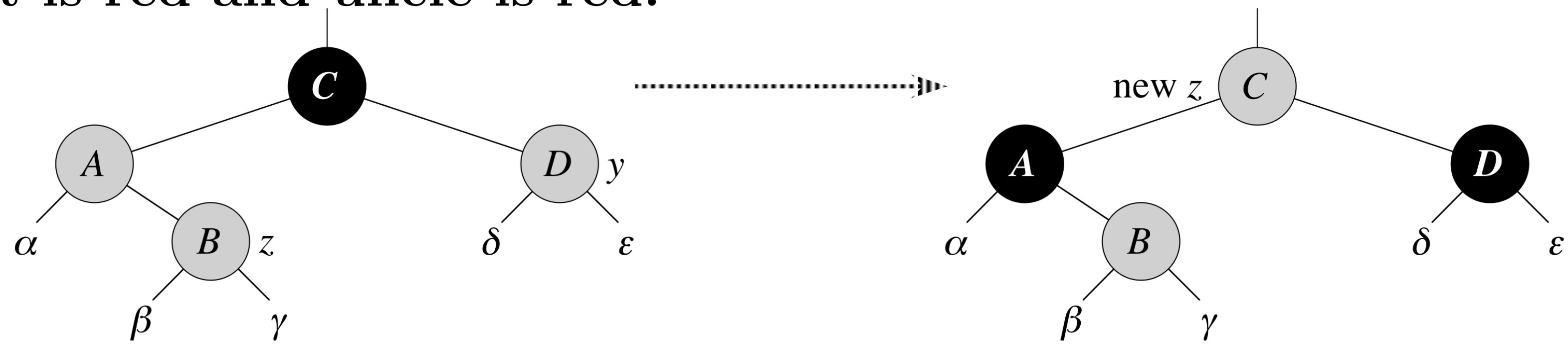

(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15

Insert fixup loop invariant.

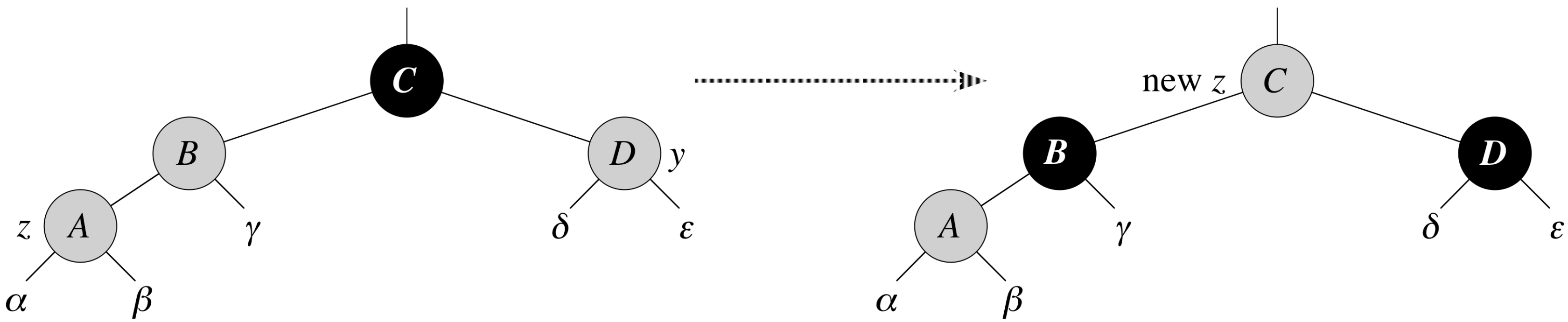
- z is red
- There is at most one red-black violation:
  - z is a red root.
  - z and $z.p$ are both red.

Parent is red and uncle is red:

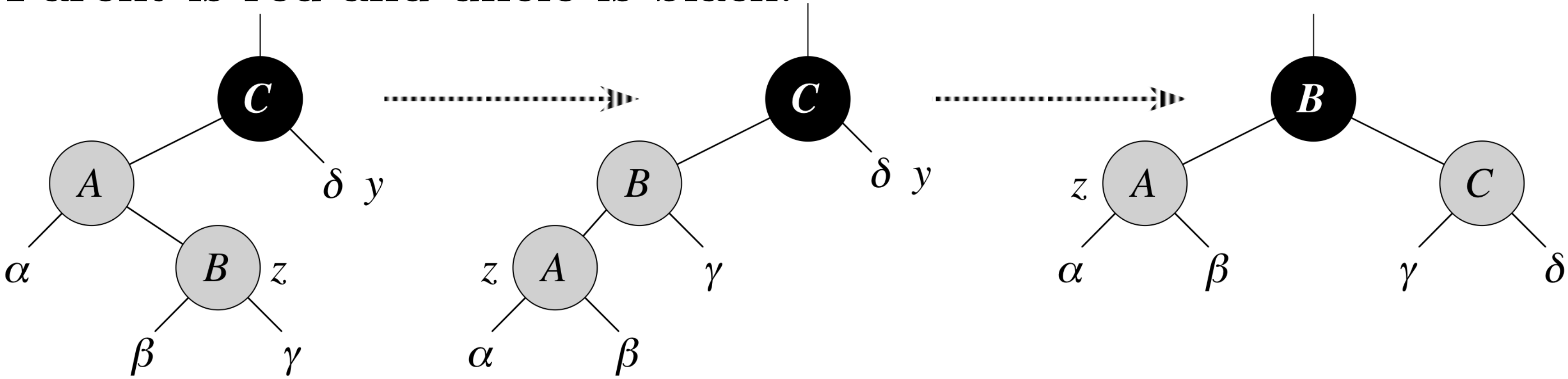
(a)



(b)



Parent is red and uncle is black:



Case 2

Case 3

Analysis

- $O(\lg n)$ time to insert into binary tree.
- Fixup also $O(\lg n)$:
  - Each pass through the loop takes $O(1)$ time.
  - Each iteration moves z up two levels.
  - $O(\lg n)$ levels.
  - Also note that there are at most 2 rotations overall.
- Insertion into red-black tree is $O(\lg n)$.

Deletion

- Not covered here.
- But also $O(\lg n)$.