Notes on Red-black Trees

Geoffrey Matthews

May 21, 2018

Red-black trees

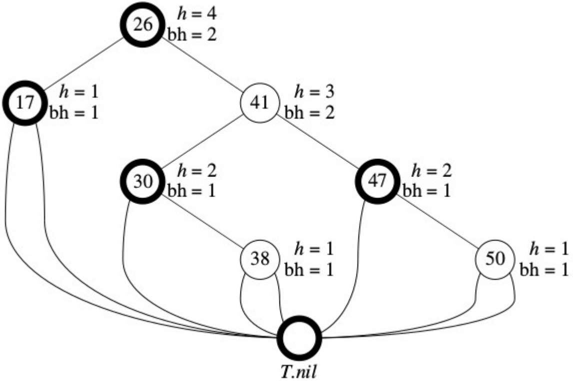
- A variation of binary search trees.
- **Balanced:** height is $O(\lg n)$, where n is number of nodes.
- Operations will take $O(\lg n)$ in worst case.

Red-black trees

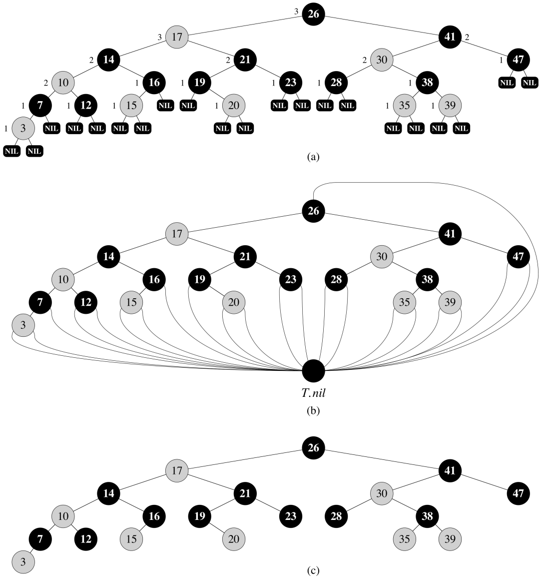
- A **red-black tree** is a binary search tree.
- One bit per node stores an attribute *color*, red or black.
- All leaves are empty (nil) and colored black.
- We use a sentinel $T.nil$ for all the leaves of a red-black tree T.
- $T.nil.color$ is black.
- The root's parent is also $T.nil$.
- All other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

Red-black tree properties (remember!)

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.
5. All paths from a node to descendent leaves have same number of black nodes, called **black height**.



Red-black tree



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: $\text{bh}(x)$ 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

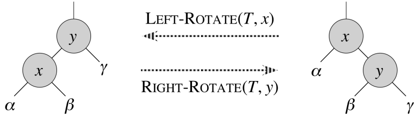
- MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR and SEARCH all run in $O(h) = O(\lg n)$ time.
- INSERT, what color to make the new node?
  - Red?
    - Might violate property 4.
  - Black?
    - Might violate property 5.
- DELETE, what color was the old node?
  - Red?
    - Successor might be black.
  - Black?
    - 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
```

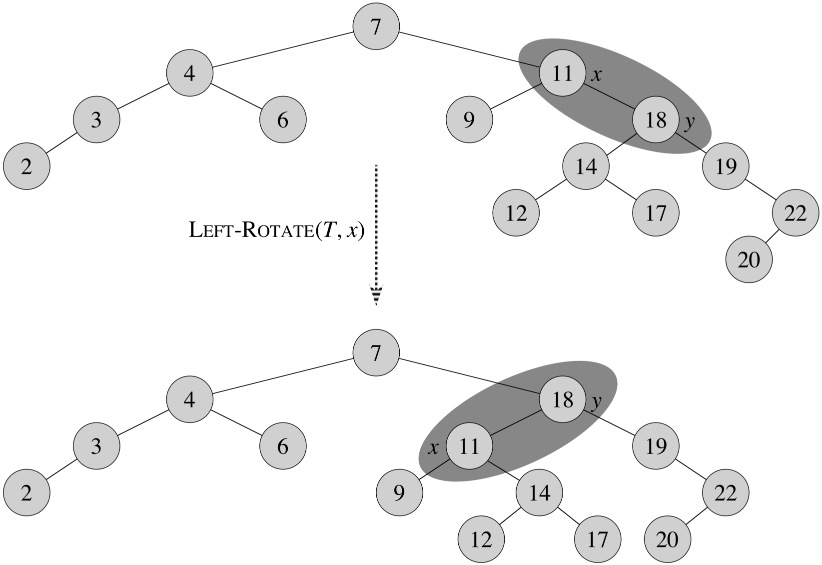


Assumes

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

$\Theta(1)$

Rotations


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

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == NIL
    T.root = z          // tree T wa
elseif z.key < y.key
    y.left = z
else y.right = z
```

RB-INSERT(T, z)

```
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

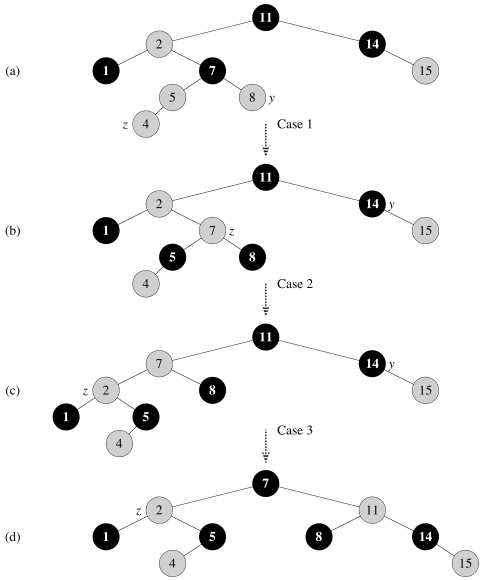
Insertion fixup

- Start by doing regular binary-tree insertion.
- Color new node red.
- May violate red-black tree properties 2 or 4:

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

Insertion fixup



Case 1, uncle red:

- make gramma's children black
- make gramma red
- make gramma new z
- loop

Case 2 (optional):

- rotate left

Case 3:

- rotate right
- make parent black

```
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
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else if z == z.p.right
                z = z.p
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else
        ("right" ⇌ "left")
T.root.color = BLACK
```

Loop Invariant: at the start of each iteration of the **while** loop:

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

Loop invariant

Loop Invariant: at the start of each iteration of the **while** loop:

1. z is red
2. If $z.p$ is the root, then $z.p$ is black.
3. There is at most one red-black violation:
   3.1 z is a red root.
   3.2 z and $z.p$ are both red.

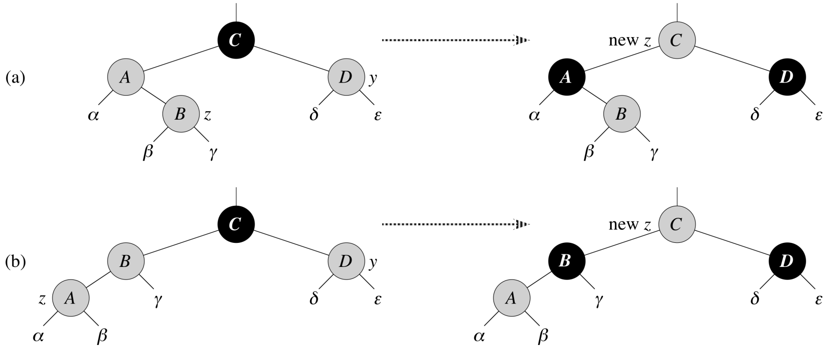
Initialization:
- 1 because we set it that way.
- 2 because it was root to begin with.
- 3.1 possible if z is new root.
- 3.2 possible if $z.p$ was red.

Termination:
- 3.1 last line fixes red root.
- 3.2 loop only terminates when $z.p$ is black.

Maintenance, Case 1, uncle is red

Loop Invariant: at the start of each iteration of the **while** loop:

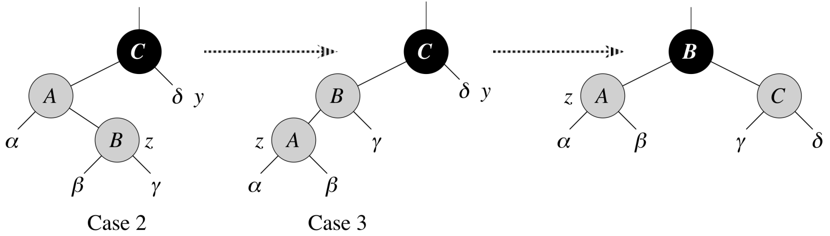
1. z is red
2. If $z.p$ is the root, then $z.p$ is black.
3. There is at most one red-black violation:
   3.1 z is a red root.
   3.2 z and $z.p$ are both red.



Maintenance, Case 2&3, uncle is black:

Loop Invariant: at the start of each iteration of the **while** loop:

1. z is red
2. If $z.p$ is the root, then $z.p$ is black.
3. There is at most one red-black violation:
    3.1 z is a red root.
    3.2 z and $z.p$ are both red.



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 or stops.
  - $O(\lg n)$ levels.
  - Also note that there are at most 2 rotations overall.
- Insertion into red-black tree is $O(\lg n)$.

RB-TRANSPLANT

TRANSPLANT(T, u, v)

if $u.p$ == NIL
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$

RB-TRANSPLANT(T, u, v)

if $u.p == T.nil$
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
$v.p = u.p$

```
TREE-DELETE(T, z)
if z.left == NIL
    TRANSPLANT(T, z, z.right)
elseif z.right == NIL
    TRANSPLANT(T, z, z.left)
else
    y = TREE-MINIMUM(z.right)
    if y.p ≠= z
        TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
```

Differences:

- y is either z or the node moved
- save y's color
- x is node that moves into y's original position
- set $x.p$ to original position of $y.p$
  - in $x.p = y$, or
  - in RB-TRANSPLANT
- RB-DELETE-FIXUP if y was black

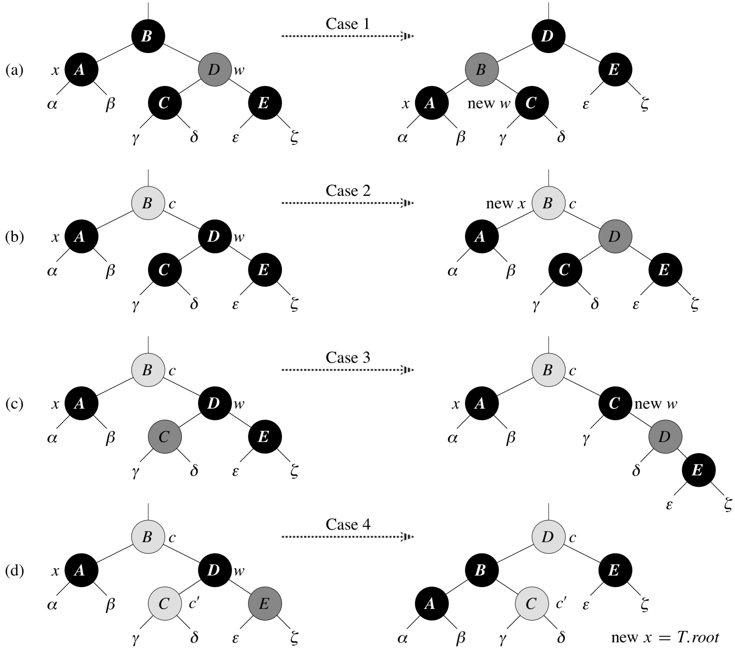
```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

What violations could occur if y was black?

1. Every node is either red or black. Still OK.
2. The root is black.
   Violation if y is the root and x is red.
3. Every leaf ($T.nil$) is black. Still OK.
4. If a node is red, then both its children are black.
   Violation if $x.p$ and x are both red.
5. All paths from a node to descendent leaves have same number of black nodes.
   Any paths containing y now have 1 fewer black.
   - Correct by giving x an "extra black."
   - Add 1 to count of black nodes on paths containing x.
   - Now 5 is OK, but 1 is not.
   - x is either **black&black** or **red&black**
   - $x.color$ is still either RED or BLACK
   - x just marks the node with a black to move somewhere.

RB-Delete-Fixup

- Idea: move the extra black up the tree until
  - x points to a **red&black** node $\Longrightarrow$ turn it black,
  - x points to the root $\Longrightarrow$ just remove the extra black, or
  - we can do rotations and recolorings to finish.
- Within the **while** loop:
  - x always points to a nonroot doubly black node
  - w is x's sibling
  - w cannot be $T.nil$ since that would violate 5 at $x.p$.
- Four cases when x is a left child:
  - Case 1: w is red
  - Case 2: w's children both black
  - Case 3: w's left child red, right child black
  - Case 4: x's right child red


(a)
Case 1
(b)
Case 2
new x
(c)
Case 3
new w
(d)
Case 4
new x = T.root

```
RB-DELETE-FIXUP(T, x)
  while x ≠ T.root and x.color == BLACK
      if x == x.p.left
          w = x.p.right
          if w.color == RED
              w.color = BLACK                                    // case 1
              x.p.color = RED                                    // case 1
              LEFT-ROTATE(T, x.p)                                // case 1
              w = x.p.right                                      // case 1
          if w.left.color == BLACK and w.right.color == BLACK
              w.color = RED                                      // case 2
              x = x.p                                            // case 2
          else if w.right.color == BLACK
                  w.left.color = BLACK                           // case 3
                  w.color = RED                                  // case 3
                  RIGHT-ROTATE(T, w)                             // case 3
                  w = x.p.right                                  // case 3
              w.color = x.p.color                                // case 4
              x.p.color = BLACK                                  // case 4
              w.right.color = BLACK                              // case 4
              LEFT-ROTATE(T, x.p)                                // case 4
              x = T.root                                         // case 4
      else (same as then clause with "right" and "left" exchanged)
  x.color = BLACK
```

Analysis

- $O(\lg n)$ time for RB-DELETE up to the call of RB-DELETE-FIXUP
- Within RB-DELETE-FIXUP:
  - Case 2 is the only case in which more iterations occur:
    - x moves up 1 level
    - Hence $O(\lg n)$ iterations
  - Each of cases 1, 3, and 4 has 1 rotation $\implies \leq 3$ rotations in all
  - Hence, $O(\lg n)$ time.
- Note:
- Red-black trees use at most a constant number of rotations.
- AVL trees in worst case use $\Omega(\lg n)$ rotations.