Red-Black Trees

August 20, 2014

1 Introduction

OK!!! we have seen that Binary Search Trees (BST) have a nice property the, insertion, deletions, minimum and maximum depends on the height of the tree. However, BST do not have the ability to keep a well balanced tree unless you are willing to randomize the initial input and have only insertions (Cormen's Chapter 12.4). This is clearly not the case for the average application that uses Binary Trees (Look at "Linux Kernel Development"). Therefore, we require an auto-balancing data structure. This is accomplished by extending the BST into Red-Black Trees (RBT).

2 Defining the Data Structure

First, each node of the three now contains the attributes: Color, key, left, right and p. In addition, a red-black tree is a binary search tree that satisfies the following red-black properties:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

An example of this type of data structures can be seen in (fig 1) . These properties insure that the tree is going to be balanced.

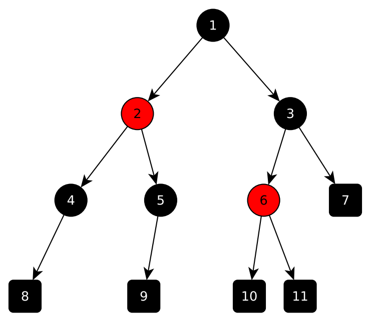


Figure 1: Red-Black Tree Example

3 Properties About the Height Of RBT

The Basic Lemma for the height of the RBT is :

Lemma 13.1

A red-black tree with n internal nodes has height at most 2 lg(n+1).

Proof:

We will show, first, that the subtree rooted at any node x contains $2^{bh(x)}-1$ internal nodes.

- If $bh(x) = 0$, then $2^{bh(x)} - 1 = 2^0 - 1 = 0$.
- If $bh(x) > 0$, then $child[x]$ has height $bh(x)$ or $bh(x) - 1$ depending on if x is black or red. Since the height of a child of x is lees than the height of x itself, we can conclude that each child has at least $2^{bh(x)-1} - 1$.
- Then, each subtree rooted at x contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes.

Finally, because of property 4, at least half of the nodes in a simple path must be black. Then, $bh(x) \geq \frac{h}{2}$.

An immediate consequence of the lemma is that SEARCH, MINIMUM, MAXIMUM, SUCCESSOR AND PREDECESSOR have a complexity of $O(\log n)$.

4 Operations: TREE-INSERT and TREE-DELETE

In order to be able to keep the structure of a red black tree we require the following operations implemented for the RBT data structure

4.1 Rotations

Basically, we use a mirror operations to be able to perform the insertion and deletion operations to recover the RBT properties. These operations are the

- Left-Rotate(T, x).
- Right-Rotate(T, x)

These can be seen in (fig. 2).

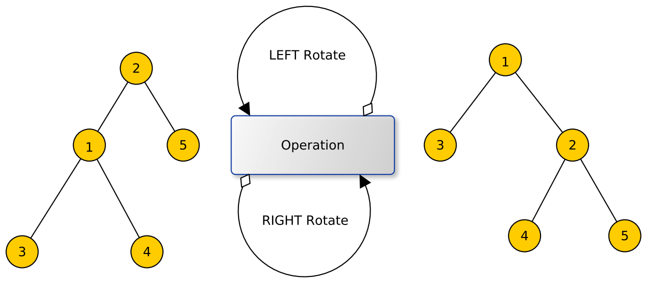


Figure 2: Rotations

4.2 Tree-Insert Invariance

Here, we will prove the tree invariance.

Initialization: Prior to the first iteration of the loop, we started with a RBT with no violations. Then, the algorithm insert the red node z at the bottom of the RBT, and this does not violate properties 1,3 and 5. The RB-INSERT-FIXUP is called:

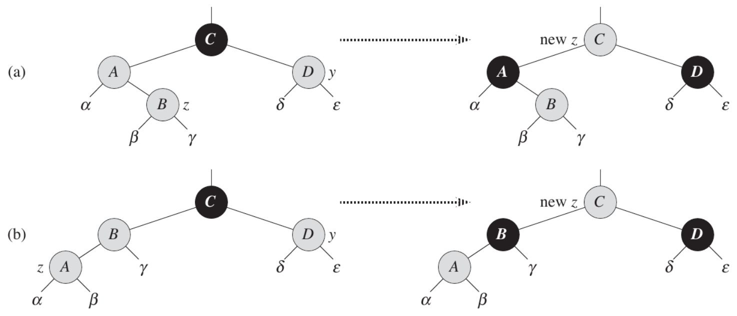
1. If z is the first node to be inserted, you violate the property 2. Then, it is the only violation on the entire RBT
2. If z is not the first node to be inserted than you could be violating property 4.

Maintenance

Six cases need to be considered in the while loop, but three are symmetric to the other tree depending if $z.p$ is a right or left child (line 2) of $z.p.p$. The code only contains the part when $z.p$ is a left child. Then, we distinguish each of the following cases by the color of $z's$ uncle $(z.p.p.right)$.

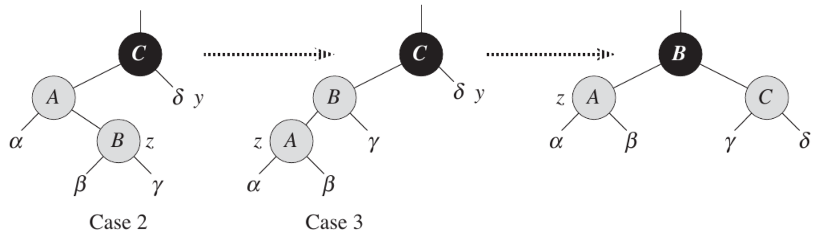
Case 1

If $z's$ uncle y is red



Case 2: $z's$ uncle y is black and z is a right child.

Case 3: $z's$ uncle y is black and z is a left child.



Termination:

When the loop terminates, it does because $z.p$ is black. Thus, at loop termination, the property 4 is not violated. The only property that could be violated is the property 2. The last lines in the RB-INSERT-FIXUP fixes that problem.

4.3 Tree-Deletion

In this code (fig. 3), we have the following:

- y is a node to be removed or moved around the tree (Look at lines 1 or 9 for this).
  - In line 1, y points to z when it has less than two children and it is removed.
  - In line 9, when z has two children then $y = Tree{-}Minimum(z.right)$. Then, y will move to $z's$ position.
- Now, the color of $y's$ can change therefore it gets stored in $y - original - color$ (Lines 2, 10).
  - Then, when z has two children $y \neq z$ then y moves to $z's$ position and y gets the same color than z. This can produce a violation.

- In lines 4, 7, and 11, x is set to point to
  - to $y's$ only child or
  - T.nil
- Since x moves to $y's$ original position, the $x.p$ is pointed to $y's$ parent. Then,
  - if z is not the original $y's$ parent $x.p$ takes place in line 6 of RB-Transplant (fig. 4)
  - if z is the the original $y's$ parent, we do not want $x.p$ to point to it since we are going to remove it. Then, in line 13 of RB-Delete, $x.p$ is set to point to y then that will take the position of z in line 17.
- If y was originally black after taking the $z.color$ can produce a violation, then RB-DELET-FIXUP is called.
- If y was originally red the RBT properties still hold.

```
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)
```

Figure 3: Code for Deletion

```
RB-TRANSPLANT($T, u, v$)
if $u.p == T.nil$
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
$v.p = u.p$
```

Figure 4: RB-Transplant

```
RB-DELETE-FIXUP($T, x$)
while $x \neq T.root$ 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
```

Figure 5: The final fix up for deleting a node in a RBT

Now, if y was black, three problems can arise:

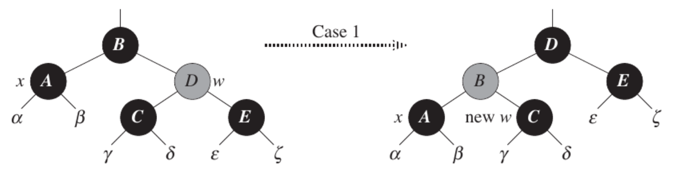
1. If y was a root and a red child becomes a the new root, we have violated property 2.

2. If both x and $x.p$ are red, then we have violated property 4.

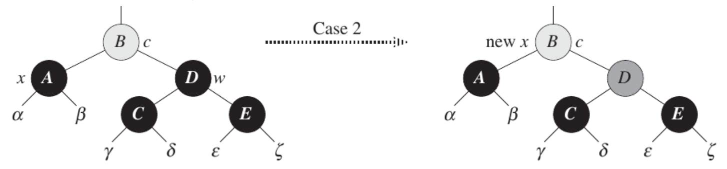
3. Moving y around decreases the bh of a single simple path. Property 5 is violated.

This could be fixed assuming that x has an "extra black." This means that the node is "doubly black" or "red-and-black." This can give us the the key issue to understand the code in (fig. 5). The procedure RB-DELETE -FIXUP restores properties 1, 2, and 4 by using the while loop to push the extra BLACK node up the tree until:

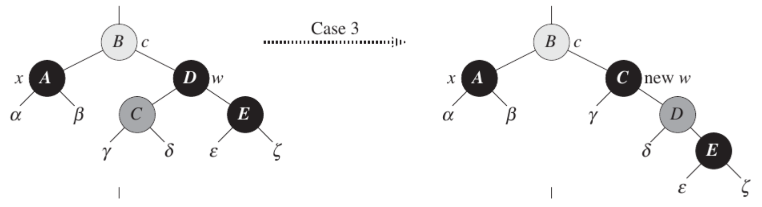
- If we have that x is a red-and-black, we simply need to change the color of the node to BLACK (Line 23).
- If we have that x is pointing to the root, remove "extra node."
- Use suitable rotations and re-colorings until x stops to be a doubly black node. They are:
  - Case 1: x's sibling w is red.



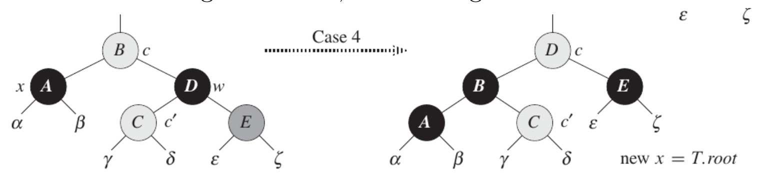
  - Case 2: x's sibling w is black, and both of w's children are black.



  - Case 3: x's sibling w is black, w's left child is red, and w's right child is black.



  - Case 4: x's sibling w is black, and w's right child is red.


Case 4
new $x = T.root$