How to easily remember Red-Black Tree insert and delete?

It is quite easy to fully understand standard Binary Search Tree and its operations. Because of that understanding, I even don't need to remember the implementations of those insert, delete, search operations.

I am learning Red-Black Tree now and I understand its properties for keeping the tree balanced. However I feel very hard to understand its insert and delete procedures.

I understand when inserting a new node, we mark the node as red (because red is the best we can do to avoid breaking less Red-Black tree laws). The new red node may still break the "no continuous red nodes law". Then we fix it via:

1. check its uncle's colour, if red, then mark its parent and uncle as black, and go to grandparent.
2. if it is right child, left rotate its parent
3. mark its parent as black and its grandparent as red, then right rotate its grandparent.

done (basically like above).

Many places describes Red-Black tree's insert like above. They just tell you how to do it. But why those steps can fix the tree? Why first left rotate, and then right rotate?

Can anyone explains why to me more clearly, even more clear than CLRS? What's the magic of rotation?

I really wish to understand so after 1 year, I can implement Red-Black tree by myself without review a book.

Thanks

algorithm data-structures tree red-black-tree

asked Feb 27 '12 at 18:02

Jackson Tale
8,487 13 62 151

You could consider converting red-black trees to the corresponding 2-3 btree representation. – hugomg Feb 27 '12 at 18:06

@missingno could you please give me more hints? – Jackson Tale Feb 27 '12 at 21:06

This is my favorite video on the subject: video.google.com/videoplay?docid=6787296543123258092 – perreal Feb 28 '12 at 0:12

3 @perreal can you give any other mirror of that video? it was deleted. – Mustafa May 29 '13 at 15:42

Can you guys take a look at my RedBlackTree remove method? stackoverflow.com/questions/28705454/… – committedandroider Feb 25 at 19:04

6 Answers

ignore my (now deleted) comment - i think okasaki's code is going to help you. if you have the book ("purely functional data structures"), look at the text on page 26 and figure 3.5 (facing, p 27). it's hard to get clearer than that.

unfortunately the thesis available on-line doesn't have that part.

i'm not going to copy it out because the diagram is important, but it shows that all the different cases are basically the same thing, and it gives some very simple ML code that hammers that home.

[update] it looks like you may be able to see this on amazon. go to the book's page, mouse over the image and enter "red black" in the search box. that gives you results that include pages 25 and 26, but you need to be logged on to see them (apparently - i haven't tried logging in to check).

edited Apr 19 '12 at 14:22 answered Feb 28 '12 at 0:01

andrew cooke
25.2k 3 48 92

For the benefit of anybody else reading this thread who doesn't have access to the book mentioned in the accepted answer, here is what I hope will be an acceptable descriptive answer.

Rotating puts the tree in a state where it meets the criteria to recolor (the child node has a red uncle). There are two key differences:

- which node is the "child" and which node is the "uncle" has changed;
- instead of recoloring the parent and uncle to black and the grandparent to red, you recolor the parent to red, and the grandparent to black.

When the child node doesn't have a red uncle, you have to rotate because if the uncle node is already black, then making the parent black would increase the black height by 1 on only one side of the grandparent. This would violate the height invariant property of red-black trees and make the tree unbalanced.

Now let's look at how the rotation transforms the tree so that we have a child node with a red uncle and can use recoloring. I recommend drawing this out to fully understand it.

- Let x be the current red node with a red parent.
- Let p be the red parent of x before the rotation (if the parent was black, we'd be done already).
- Let y be the black uncle of x before the rotation (if the uncle was red, we wouldn't need a rotation. We'd simply recolor the parent and uncle to black and the grandparent to red).
- Let g be the black grandparent of x before the rotation (since the parent is red, the grandparent must be black; otherwise this was not a red-black tree to begin with.)
- When you have a left-left (LL) or right-right (RR) case (that is, x is the left child of p and p is the left child of g OR x is the right child of p and p is the right child of g), after a single rotation (right if LL, left if RR), y becomes the child and x its uncle. Since x is a red uncle, you now have a case where you can recolor. So, recolor the parent of the child (since the child is now y, its parent is g) to red, and the child's grandparent (which is now p) to black.
- When you have an LR (x is the left child or p and p is the right child of g) or RL case (x is the right child of p and p is the left child of g), after a double rotation (right then left if LR, left then right if RL), y becomes the child and p its uncle. Since p is a red uncle, you again have a case where you can recolor. So, recolor the parent (since the child is now y, its parent is g) to red, and the child's grandparent (which is now x) to black.

Before the rotation and recoloring, you had a black grandparent with 2 red nodes and 0 black nodes on side A (left or right) and 0 red nodes and 1 black node on side B (the opposite of side A). After the rotation and recoloring, you have a black grandparent with 1 red node and 0 black nodes on side A and 1 red node and 1 black node on side B. So you essentially moved one of the red nodes to the other sub-tree of the grandparent without increasing the black height of either sub-tree.

That's the magic of rotation. It allows you to move the extra red node to another branch without changing the black height, and still preserving the sorted traversal property of a binary search tree.

edited Feb 22 '13 at 12:01

answered Feb 22 '13 at 2:20

Serge Binette
101 1 3

The logic is fairly simple. Suppose z is red and z's parent is red: If z's uncle is red, do step 1 to push the problematic node upwards until either (1) the parent becomes the root. Then simply mark the root black. Done or (2) z's uncle is black.

In case (2) either (a) z is the left child of its parent, then step 3 will be the last step as all properties of BST are fulfilled. Done. or (b) z is the right child of its parent. Step 2 will convert the problem to case (a). Then do step3. Done.

Thus the logic is to try to reach case (1) and (2a), whichever comes first. Those are the situations we know the solutions.

answered Sep 27 '12 at 0:15


blackgreenmac
41 1 3

Can you take a look at my RedBlackTree remove method? stackoverflow.com/questions/28705454/… – committedandroider Feb 25 at 5:25

sorry, haven't touched rb-tree for long. – blackgreenmac Feb 25 at 22:24

thanks for your response anyways:) The real question is why are we learning the implementation of this..... – committedandroider Feb 26 at 8:13

Perhaps it is worth starting with left-leaning red black trees. They offer an interesting simplified implementation.

answered Sep 27 '12 at 5:49


Peteris
1,676 1 9 21

Any 2-4 (2-3-4) tree can be converted into a Red-black tree. And understanding of 2-4 trees is much easier. If you just go through the insert and delete operations in 2-4 trees, you will feel that there is no need to remember any rules to achieve the same. You will see a clear simple logic which enables you to come up with the solutions to take care of different insertion and deletion scenarios.

Once you have a clear understanding of 2-4 trees, when ever you deal with Red-black trees, you can mentally map this Red-black trees to 2-4 trees and come up with a logic by yourself.

I found following couple of videos which are extremely useful in understanding 2-4 trees, Red-black trees and mapping of 2-4 trees to Red-black trees. I would recommend to go through these videos.

1) For 2-4 trees : http://www.youtube.com/watch?v=JZhdUb5F7oY&list=PLBF3763AF2E1C572F&index=13

2) For Red-black trees : http://www.youtube.com/watch?v=JRsN4Oz36QU&list=PLBF3763AF2E1C572F&index=14

Even though they are one hour long videos each, I felt it was worth going through them.

answered Oct 22 '14 at 7:10


Nagakishore Sidde
381 3 8

you really make good conclusion that three cases.

after insertion in RB-Tree, there left a main problem to solve if there is. there two continuous red nodes!! how could we make that two continuous red nodes disappear without violate that rule (every path have same count black node) so we see the two node, there only exists 3 circurm...

I am sorry, you could see the text book of Instruction to Algorithms

no body can help you think through rb-tree. they can only guide you at some key point.

answered Jan 21 at 10:37


surfree
21 1