



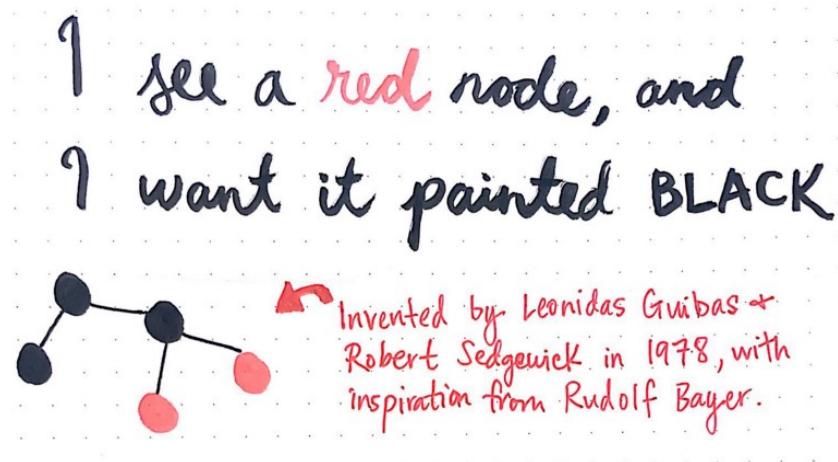
Vaidehi Joshi

[Follow](#)

Writing words, writing code. Sometimes doing both at once.

Aug 28, 2017 · 16 min read

Painting Nodes Black With Red-Black Trees



I see a red node, and I want it painted black!

There is almost always more than one way of doing something. This is especially true in the world of software. In fact, that's what all software is, really: different approaches to solving similar—or sometimes, the *exact same*—problems.

What's particularly interesting about the many approaches to problem solving within computer science is the fact that many of the solutions that we depend on as programmers daily are actually built upon more naive solutions that were initially invented by someone else. We've seen that this is the case with the algorithms that we've covered in this series. If we think back to all those sorting algorithms, many of the more obvious and efficient algorithms came about after the simpler, less efficient solutions were invented. Merge sort was derived as early as 1945, while bubble sort and the earliest iterations of insertion sort, were invented in 1956. But today, some programming languages (including Java) use a solution that is a *hybrid* of both insertion and merge sort: timsort, which was inspired by these two solutions, and only invented recently, in 2002.

But it's not just sorting algorithms that adhere to the idea of building upon a solution; this truth is scattered throughout many common computer science problems. Despite their contentious internal wars, even modern-day JavaScript frameworks do *exactly* this: each framework learns from and draws upon the solutions of others, and aims to build upon the things that work, and improve upon the things that don't. It is wise to lean on solutions that have already been created by others and build upon them to create our own.

As it turns out, programmers have known this for a long time. And they've been learning from each others' solutions for decades, from sorting algorithms, to frontend frameworks, to even the simplest of data structures. We saw an example of this most recently in the context of AVL trees, a type of self-balancing tree that build upon the ideas of binary search trees. Today's example builds upon both of these two tree structures even further, with a really fun (and maybe a little fickle) structure called a red-black tree.

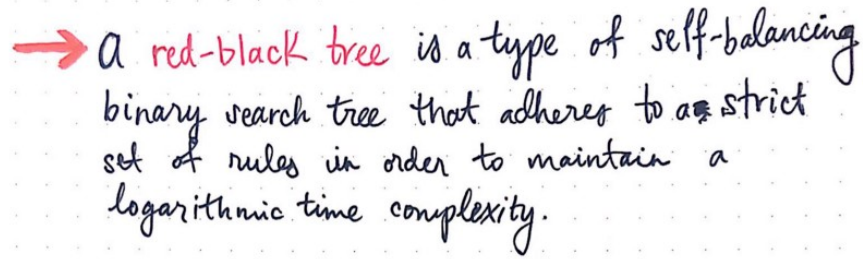
A very disciplined tree

Since we already know the basics behind a balanced binary search tree, we can jump right into red-black trees. The history behind red-black trees is pretty unique, and we'll dive into some of the idiosyncrasies of this particular structure's origins in a bit. Red-black trees were invented in 1978, by two researchers named Leonidas J. Guibas and Robert Sedgwick, at Xerox PARC, a research and development company based in Palo Alto, California. They introduced the concept for red-black trees in a paper they co-published together, called "A Dichromatic Framework for Balanced Trees".

However, they didn't invent this structure alone; they adapted the work of another German computer scientist, named Rudolf Bayer, who had already begun work on inventing this exact data structure through his research on special types of balanced trees. Both Guibas and Sedgwick attributed their work on red-black trees to Bayer's work, which had been published just years earlier, in 1972. Indeed, if it hadn't been for Bayer's groundwork and inspiration, red-black trees would have never been discovered!

Okay, so we know that red-black trees took a whole lot of brain power to come into this world. But what exactly *is* a red-black tree? Let's begin

with the simplest definition to start.

A handwritten note on a grid background. It starts with a red arrow pointing to the right, followed by the text "A red-black tree is a type of self-balancing binary search tree that adheres to a strict set of rules in order to maintain a logarithmic time complexity." The words "red-black tree" are written in red ink, while the rest is in black ink. The handwriting is cursive and slightly slanted.

→ A red-black tree is a type of self-balancing binary search tree that adheres to a strict set of rules in order to maintain a logarithmic time complexity.

Red-black trees: a definition.

A **red-black tree** is a type of self-balancing binary search tree, very similar to other self-balancing trees, such as AVL trees. However, a red-black tree is a structure that has to adhere to a very strict set of rules in order to make sure that it stays balanced; the rules of a red-black tree are exactly what enables it to maintain a guaranteed logarithmic time complexity.

On the most basic level, a red-black tree must follow four rules, no matter what. In every aspect of building and shrinking (inserting or deleting from) this tree, these four rules have to be followed; otherwise, the data structure cannot be considered a red-black tree.

Red-Black RULES:

- 1/ each node must be either **RED** or **BLACK**
- 2/ the root of the tree must ALWAYS be **BLACK**
- 3/ two **RED** nodes can never appear in a row within the tree; a **RED** node must always have a **BLACK** parent node, and **BLACK** child nodes
- 4/ every branch path from the root node in the tree to a NULL pointer passes through the exact same number of **BLACK** nodes (this is also an unsuccessful search path)

Rules of a red-black tree.

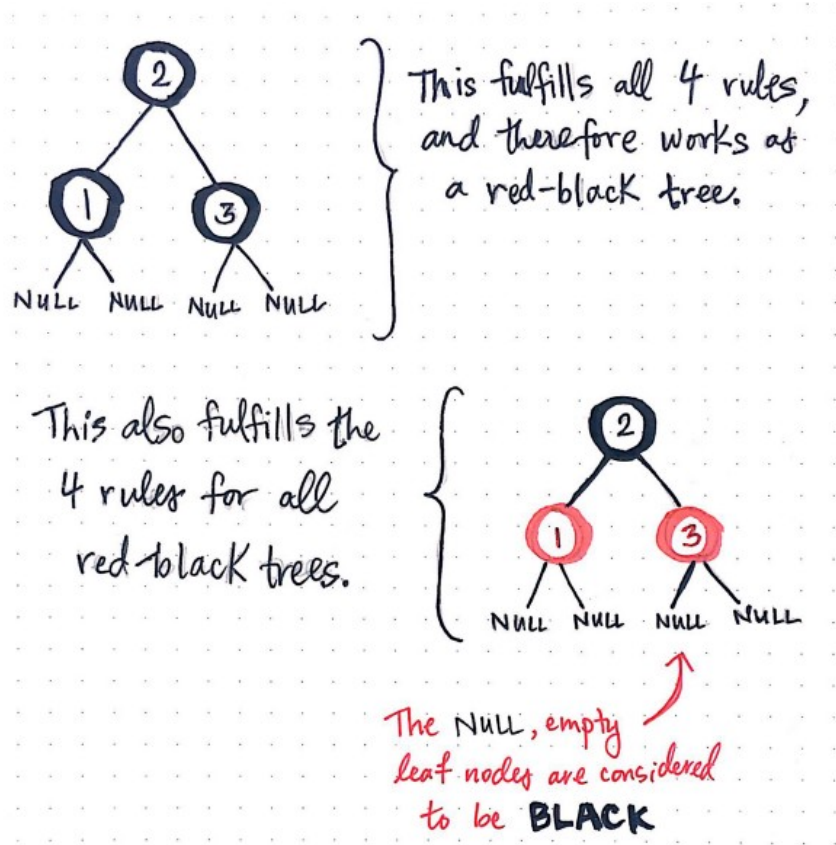
Generally speaking, the four rules of a red-black tree are always presented in the same order, as follows:

1. Every single node in the tree must be either red or black.
2. The root node of the tree must always be black.
3. Two red nodes can never appear consecutively, one after another; a red node must always be preceded by a black node (it must have a black parent node), and a red node must always have black children nodes.
4. Every branch path—the path from a root node to an empty (`null`) leaf node—must pass through the exact same number of black nodes. A branch path from the root to an empty leaf node is also known as an *unsuccessful search path*, since it represents the path we would take if we were to search for a node that didn't exist within the tree.

These rules start to make a lot more sense when we see them in action, so let's take a look at some red-black trees (and non-trees) to try to understand what's going on.

In the example shown here, the first tree has three black nodes. Notice that the two children nodes each have pointers to `null` leaf nodes. It might be obvious, but it's worth remembering this:

a `null` leaf node is always considered to be a black node, not red.



Fulfilling the rules of a red-black tree.

The first red-black tree adheres to all four rules:

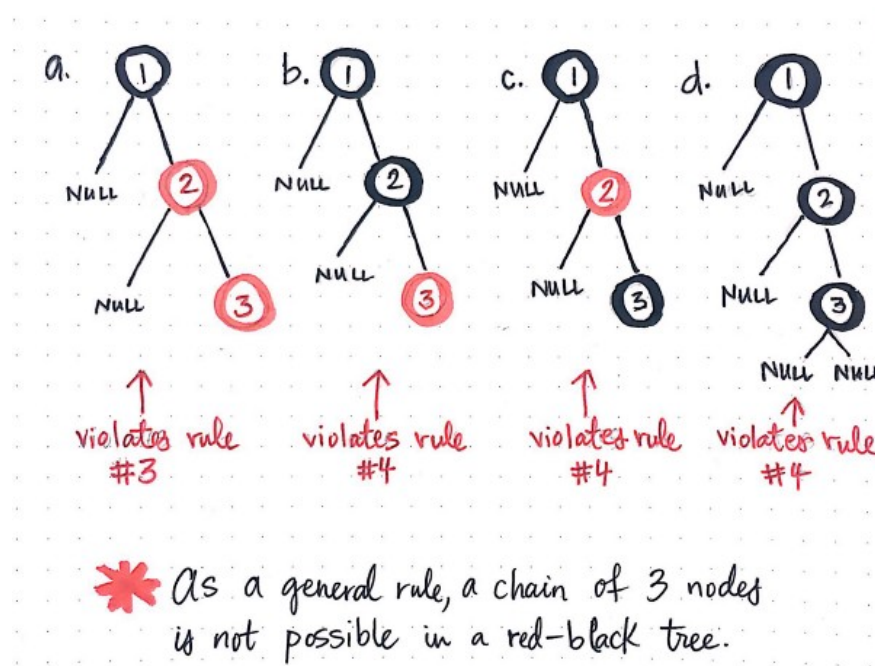
1. every node is red or black,
2. the root node is black,
3. no two red nodes appear consecutively,
4. and, finally, the path from the root node to all four `null` leaf nodes passes through two black nodes (either 2 and then 1, or 2 and then 3) on the way to the `null` leaves.

Interestingly, the second example is also a valid red-black tree. The main difference here is that the child nodes, 1 and 3 are both red. However, neither of the two red nodes appear consecutively, so this tree still doesn't violate the third rule. Also, the branch paths from the root to all the `null` leaf nodes pass through the same number of nodes in this case, too—we pass through just one black node, the root, on the path from the root to a `null` node. So, we're not violating the fourth rule, either.

At first, it might seem as though following these four rules is pretty easy to do; we created two red-black trees just now without violating any rules, right? Well, as we're about to discover, these rules are super strict and very easy to break.

The best way to demonstrate this is with an example of three chained nodes.

In the illustration shown here, we have four potential different options for how we could create a red-black tree with three nodes, 1, 2, 3. In fact, we probably have more options, but let's assume that we're not going to consider making a tree that doesn't have red and black nodes, and a root node that is black.



A chain of 3 nodes is not possible in a red-black tree.

In tree **a**, we have a black root, and two consecutive red nodes. We can tell from just looking at this tree that this is a problem; we're violating rule three!

In tree **b**, we have a black root, a black child node, and a red grandchild. This might *seem* fine, but...remember rule four? The path from the root node to every `null` leaf nodes should pass through the same number of black nodes, no matter which branch path we take. The path from **1** to its left `null` child passes through only one black node, while the path to **2**'s left child passes through two black nodes. Well, that totally breaks the fourth rule!

The same problem occurs for trees **c** and **d**; the path from the root node to each `null` node passes through a slightly different number of black nodes, which means that none of these trees are actually valid red-black trees!

As it turns out, there is a well-known proof in the realm of red-black trees that proves exactly what we just saw:

A chain of three nodes cannot possibly every be a valid red-black tree.

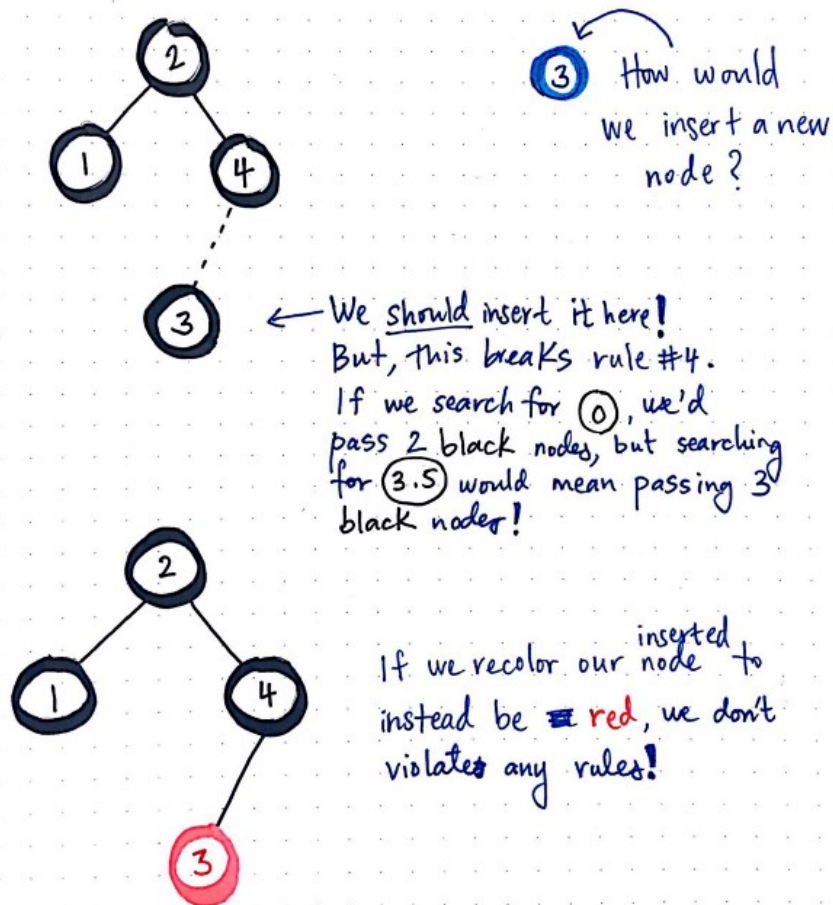
These rules are turning out to deceptively more strict than they first seemed, right? Well, even though they might seem like a total pain to follow, they're pretty important—not to mention *powerful*. Let's try to unpack why!

Following the rules of red

The trickiest time to follow the rules of red-black trees is when we're growing or shrinking the tree. A perfectly-balanced red-black tree is great, but most data structures tend to have elements and data inserted and removed from them. When it comes to red-black trees, this can be a little daunting at first.

Let's unpack how to handle a red-black tree while inserting a node, one step at a time.

* A red-black tree always has to follow its set of strict rules. The tricky thing is making sure we don't violate any of these rules when we insert or delete nodes from within the tree structure.



How would we insert a new node with a value of 3?

In the illustration shown here, we have a tree with three black nodes. How could we add a new node with a value of 3 into this tree?

The first step is basically ignoring the red-black rules, and initially just figuring out where the node would go according to the rules of a normal binary search tree. Since 3 is larger than 2 but smaller than 4, we'll make it the left child of the node 4. At first, when we insert this node, we'll break rule four, since if we search for a null leaf node from the

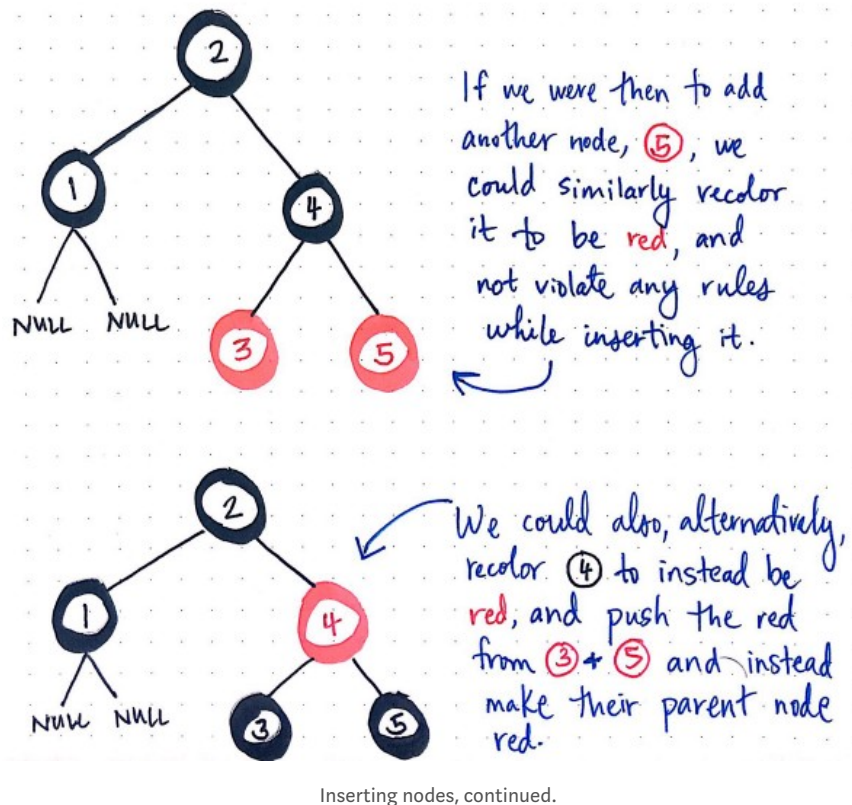
left subtree, we'll pass two black nodes, but an unsuccessful search through the right subtree passes through *three* black nodes.

Instead, if we recolor our inserted node with a value of 3 to be *red*, instead of black, we don't violate any of our four red-black tree rules.

This tends to be a good strategy when it comes to inserting into a red-black tree.

Inserting a node and immediately coloring it red makes it much easier to identify and subsequently fix any violations.

Okay, so now our tree has four nodes: 1, 2, 4, and 3. What would happen if we were to insert another node, this time with a value of 5?

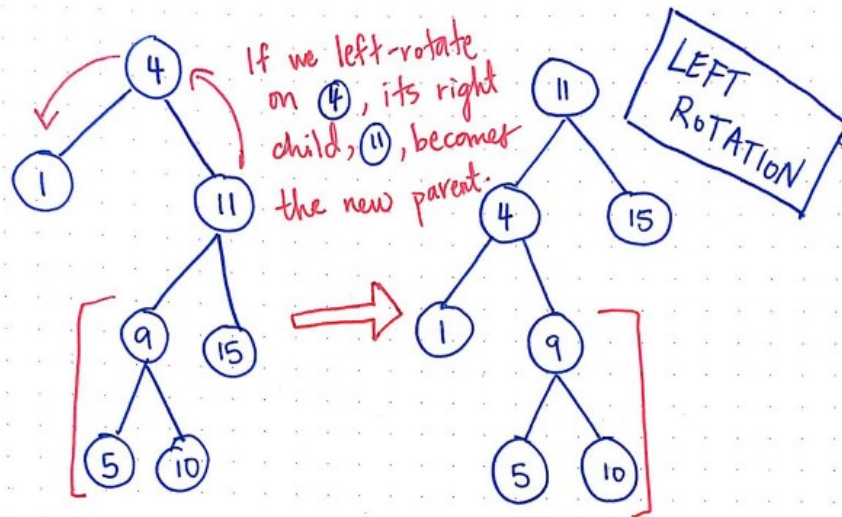


Well, we'd start by following the steps we're already familiar with: locate the position for the node and recolor it to be red.

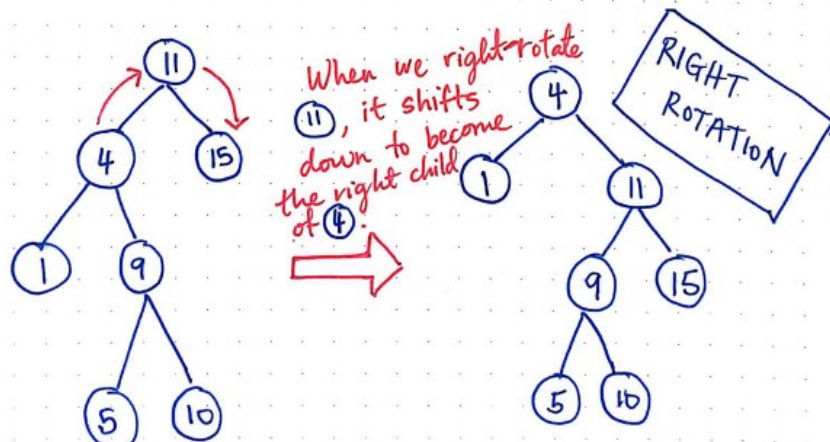
In our tree illustration, we'd be inserting our node `5` as the right child of the black node `4`. This wouldn't violate any rules, since neither of the two red nodes appear one after the other, and all paths from the root node to `null` still pass through exactly 2 black nodes.

However, we could also recolor the black parent node, `4`, to be red *instead of* black. In this scenario, we're basically "pushing up" the red from it's children, `3` and `5`, so that their parent node becomes red. Notice that we still don't violate rules three or four, and we still pass through exactly 2 black nodes in all the paths from the root to `null`.

As we've seen from this example, **recoloring** nodes is one well-used technique for handling insertions (and deletions!) into a red-black tree.



Notice how the structure of the tree has changed by virtue of the subtrees being moved up/down.



The power of left and right rotations.

Another handy trick for handling rule violations is one that we encountered earlier in this series: rotations! We learned about rotations (or “glorified swaps” as I like to think of them) when we explored AVL trees.

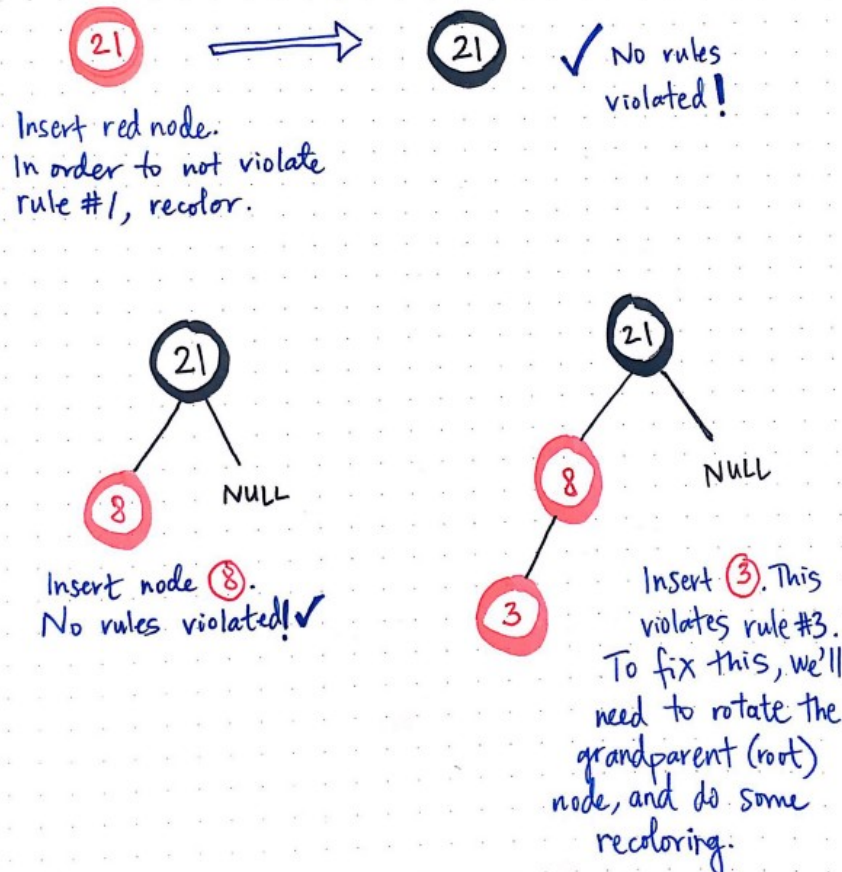
We can do something similar when it comes to red-black trees, too. In the drawing shown here, we have a red-black tree that is unbalanced;

ignoring the colors of the nodes for a moment, let's say that we need to rotate and re-balance this tree.

In the first drawing, we are left-rotating on the root node, 4, so that its right child, 11, becomes the new parent. This is called a **left-rotation**. Similarly, if we right-rotate on the new parent node, 11, and shift it down so that it once again becomes the right child of 11, we're performing a **right-rotation**.

Notice how the structure of the tree changes as we rotate in both of these cases; in our left-rotation, the subtree of 9-5-10 moved from the right subtree to the left. Conversely, in our right-rotation, the same subtree of 9-5-10 moved from the left subtree back to the right. **Rotations** tend to move around and restructure the subtrees of a larger red-black tree, which can also be helpful in preventing any rule violations.

- We can handle most insertion/deletion scenarios by **recoloring** and/or **rotating** nodes.
- It's easiest to start off by always inserting a **RED** node, and then rotating/recoloring afterwards.



Handling insertion and deletion by rotating and recoloring.

Let's look at one more example of how recoloring and rotation can help us in inserting nodes into a red-black tree.

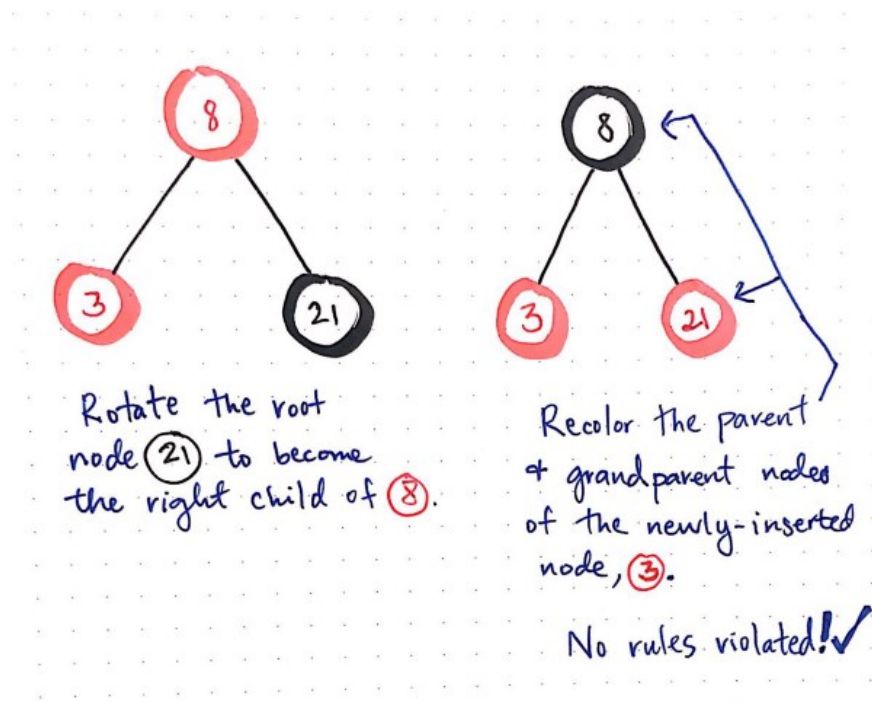
Remember: it's easiest to start off by always inserting a *red* node, and then recoloring and rotation as necessary, afterwards.

We'll start with a single root node of 21, which will be red. But, since this is the root node, and one of our rules is that the root node must

always be black, we can recolor node 21 to be black. Now, no rules are violated!

Next, we'll insert a node with a value of 8 into the tree, as the left child of the root. We can insert it as a red node and not violate any rules in the process!

Next, we'll insert a node with a value of 3. Inserting this node as the left child of 8 violates rule three, since we'll have two consecutive red nodes. In order to fix this, we'll need to rotate the grandparent node (the root) and then recolor.



Rotate and recolor, no rules are violated.

If we right-rotate the root node, and shift 21 down to become the right child of 8, we've taken one step to fixing our problem. The node 8 is our new root node, with two children, 3 and 21.

However, now we're violating our root node rule yet again!

Not to worry—we can just recolor our original parent node of the newly-inserted node (which is now the right child), 21, and the root node,

8. If we recolor 8 and 21, our root node is back to being black, and our two child nodes are both red.

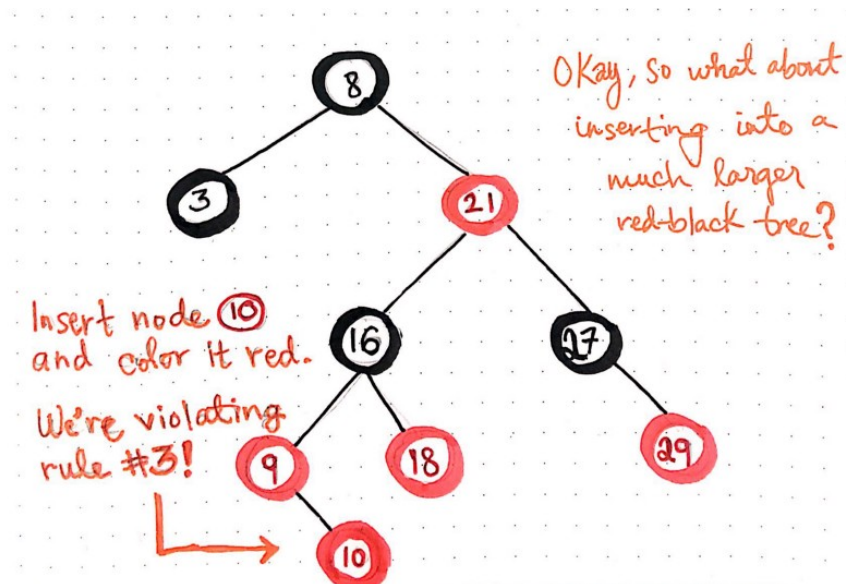
And, most important of all, no rules are violated, and we have a perfectly-balanced red-black tree!

The benefits of painting it black

Okay, so we managed to recolor and rotate enough in our first example to get things perfectly balanced and working correctly.

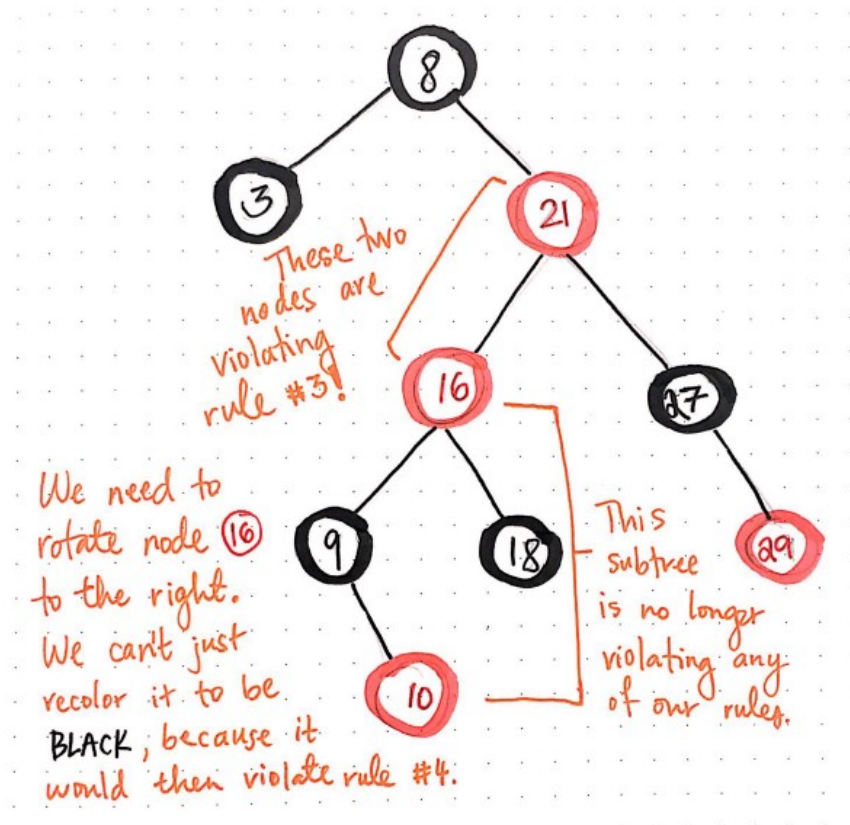
But, what about inserting elements into a much larger red-black tree? Let's walk through a more complicated example; as we'll see, the same methods apply.

The key to dealing with larger red-black trees is moving any rule violations up the tree as we go.



What about inserting into a much larger red-black tree?

In this example tree, we have eight nodes, and we're inserting node 10 into the tree. We'll insert it as the left child of 9, and color it red. Immediately, we can see that we're violating rule three—two red nodes are appearing consecutively.

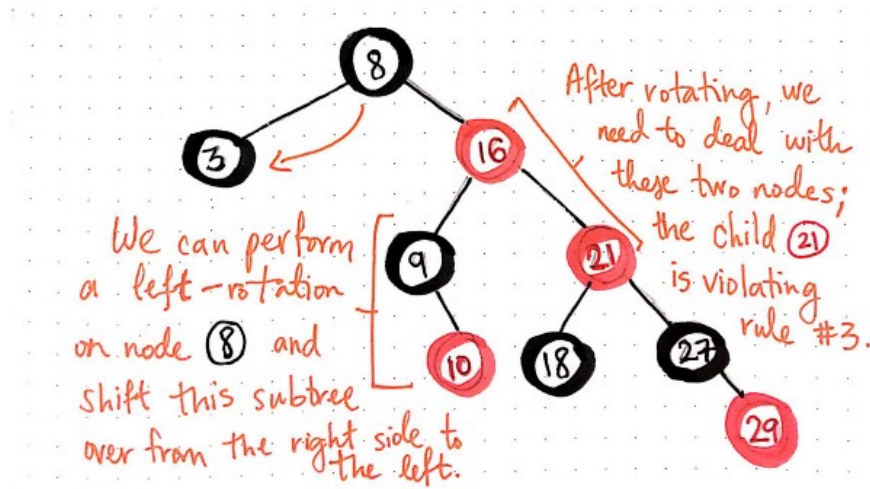


Node 16 violates the tree, so it must be rotated.

We can first start off fixing this problem by recoloring the parent nodes of our newly-inserted node, `10`. This at least solves the problem of rule three being broken!

Well...*kind of*. Actually, all we've done by recoloring here is *shifted up* our violation so that nodes `16` and `21` are now breaking rule three. We didn't get rid of this violation, we just moved it up!

We can't recolor `16` to be black, because if we did that, we'd be violating rule four, and changing the number of black nodes that we'd pass through on all paths from the root node to `null` leaves. Since there are no parent nodes left to recolor (except for the root node, which we don't want to make red!), we can now lean on rotations to help us out. Since node `16` is violating the rule of consecutive reds, we can rotate this node to the right.



Now, node 21 is the node that is violating the tree!

After rotating node 16 to the right, we've effectively pushed down nodes 21, 27, and 29.

Now, we need to deal with the two consecutive red nodes that are still breaking rule three: 16 and 21.

We could rotate the entire tree over by left-rotating node 16 and making it the new root node. However, we'll notice that both nodes 9 and 10 are going to be in the wrong spot if we make 16 the new root node. We'll need to perform a left-rotation on nodes 9 and 10 so that they both are on the left subtree rather than the right subtree.