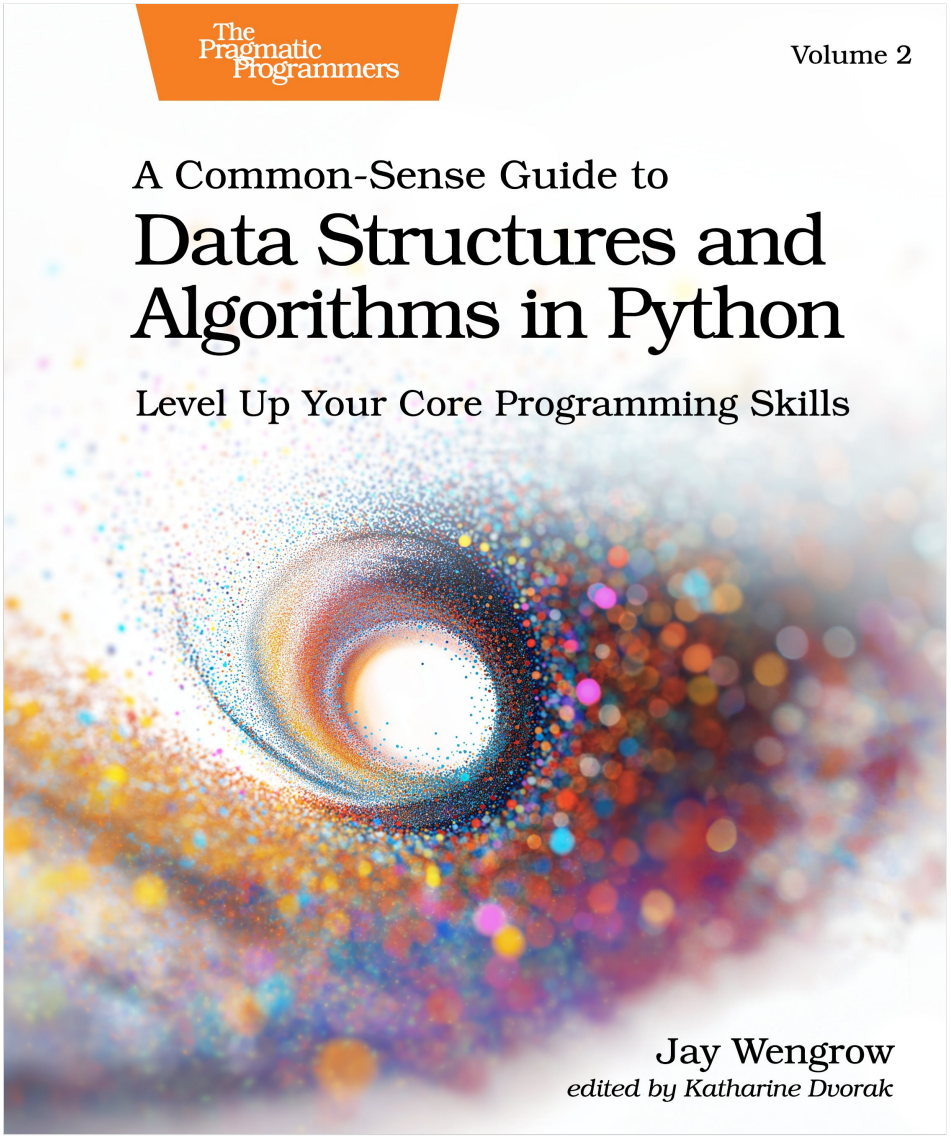


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

In Volume 1, Chapter 15, I covered binary-search trees (BSTs), including the BST deletion operation. Being that red-black trees are a type of BST, the red-black tree deletion algorithm starts the same as regular BST deletion.

However, after we use the BST deletion algorithm to delete a node from the red-black tree, this may introduce a violation of the Red-Black Rules. As such, just as with red-black tree insertion, red-black tree deletion undergoes a second "Fixing Phase" in which we modify the tree to correct any red-black violations.

In short, red-black tree deletion has two phases:

- Phase 1: We delete a node with the regular BST deletion algorithm.
- Phase 2: We modify the tree to bring it back into compliance with the Red-Black Rules. This is the Fixing Phase.

Phase 1: Deleting a Node

Phase 1 of red-black tree deletion begins by deleting a node with the BST deletion algorithm. As a brief refresher, following is the gist of BST deletion. It's only a summary, though. To understand the details of the BST deletion algorithm more clearly, you may want to refresh your memory by taking a look at Volume 1, Chapter 15.

Any deletion of a node within a BST falls under one of three general cases:

- The deleted node has no children.
- The deleted node has one child.
- The deleted node has two children.

However, the case of a deleted node having two children can be broken down into further subcases, based on where the *successor node* is. Again, this is covered at length in Volume 1, but as a quick reminder, the successor node is the node with the next-greatest value in the tree after the value of the deleted node. So if a tree contained all the values from 1 to 100, and we deleted the node whose value is 48, the successor node is the node with the value 49. In Volume 1, I also explain how we locate the successor node.

All in all, we can describe all the cases as follows:

- Case A: The deleted node has no children.
- Case B: The deleted node has only one child.
- Case C: The deleted node has two children, and the successor node is the deleted node's right child.

- Case D: The deleted node has two children, and the successor node is *not* the deleted node's right child, but some node further down the tree.

Even Case D can be broken down into two further subcases. One subcase is where the successor node has a right child, and the other is where the successor node does *not* have a right child. However, to streamline my explanation, I'm going to lump both of these subcases into Case D.

In all cases, a node is deleted from the tree by taking its parent and modifying its appropriate child attribute. That is, if the deleted node is a left child, we change the parent's `left_child` attribute to be `None` or some other node in the tree. If, on the other hand, the deleted node is a right child, we modify the parent's `right_child` attribute.

Here's the algorithm that describes what we do in Cases A through D:

- Case A (the deleted node has no children): We simply remove the node from the tree, without the need for any further action.
- Case B (the deleted node has only one child): We change the parent of the deleted node so that its child is no longer the deleted node, but instead the deleted node's own child.
- Case C (the deleted node has two children, and the successor node is the deleted node's right child): We change the parent of the deleted node so that its child is no longer the deleted node, but instead the successor node. In this case, the successor node is the deleted node's right child.
- Case D (the deleted node has two children, and the successor node is *not* the deleted node's right child): We take the successor node's value and use that value to overwrite the deleted node's value. We then delete the successor node from the tree in the following way: if the successor node has a right child, we modify the successor node's parent so that its left child is now the successor node's right child. If the successor node does *not* have a right child, then the successor node's left child can simply point to `None`.

If this doesn't make much sense to you, don't worry; you can check out Chapter 15 in Volume 1 for a full discussion.

Additions to Phase 1

Before we look at the code implementation, it's important that I point out a few notable differences between the code implementation for RBT deletion and the code implementation in Volume 1.

First, in Volume 1, we only dealt with a `Node` class; we didn't have a dedicated tree class. Here, we have a dedicated `RedBlackTree` class that, in turn, works with a `Node` class. This difference isn't terribly important, but it does mean that I couldn't simply copy the Volume 1 implementation and paste it here.

Another difference is that in our Volume 1 BST implementation, there was no need for a node to keep track of who its parent is. Nodes simply knew who their children were. In our red-black tree implementation, on the other hand, nodes are aware of their children *and* their parents. And so, when we modify the red-black tree upon deletion, we can't only update `left_child` and `right_child` attributes; we have to modify `parent` attributes as well.

These implementation differences are minor, although necessary. However, there are some other additions to Phase 1 of the red-black tree deletion algorithm that are more noteworthy.

Color Change

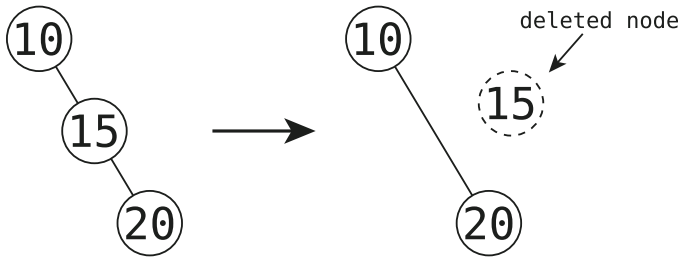
Deleting a node from a red-black tree often means that some other node will move into the spot where the deleted node was plucked from. I will call this other node the "plugged-in node." That is, we plug this node into the spot from which we took away the deleted node. A new step of red-black tree deletion that didn't exist in regular BST deletion is that *we declare the plugged-in node's color to be the same color as the deleted node.*

Check the Color of the "Deleted Node"

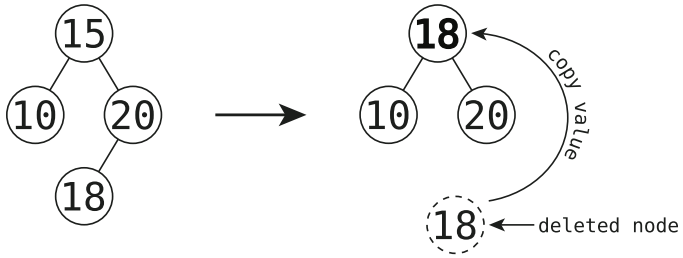
Another key addition to the algorithm is that at the end of Phase 1, we check to see whether the node we deleted is red or black; and if it is black, we launch Phase 2, The Fixing Phase.

Before I elaborate on this point, it's critical that I explain a tricky nuance when it comes to the "deleted node," which, going forward, I will refer to by its variable name, `deleted_node`. `deleted_node` refers to the actual `Node` instance that was disconnected from the tree. However—and this is the tricky part—the `deleted_node` isn't always the node *whose value* we remove from the tree. Let me explain with some examples, using regular BSTs.

In the following BST, when we delete the 15 node, the `deleted_node` is, indeed, the `Node` instance we remove from the tree:



However, in the next BST, when we delete the 15, we don't actually remove the 15 `Node` instance from the tree. Rather, we copy the value of the successor node (the 18) into the 15 `Node`, and it's the *successor node* whose `Node` we actually disconnect from the tree:



And so in this tree, the `deleted_node` was the original `Node` that contained the 18, and not the root node. In other words, although the *value* we deleted from the tree was 15, the actual `Node` *instance* that was disconnected from the tree was the `Node` that originally contained the value 18.

It emerges that in some cases, the `deleted_node` is, in fact, the node whose value we deleted. But in other cases, the `deleted_node` is a different node entirely. In these latter cases, the `deleted_node` is the successor node, as that is the `Node` instance that is removed from the tree.

In any case, to determine whether we need to even launch the Fixing Phase, our code checks to see whether the `deleted_node` is red or black. If the `deleted_node` is red, there is no Fixing Phase because it's *impossible* that our deletion could have introduced a Red-Black Rule violation. I'll explain why.

There are only two potential violations: one that creates red enemies, and one that makes one tree path have a greater black height than another path. Deleting a red node can't possibly break either rule.

It can't cause red enemies, because an existing red node will only have a black parent and black children. If this weren't the case, then the tree would *already* be in violation of having red enemies. And so, when we remove a red node

instance from the tree, all this does is bring black nodes closer together, which is certainly no problem of red enemies.

At the same time, removing a red node instance from the tree will also not violate the Black Height Rule, since deleting a red node doesn't affect the black height of a path. After all, the black height is determined by the number of *black* nodes in a path. The number of red nodes is completely irrelevant to black height.

On the flip side, when we remove a black node, we can easily violate one or both of the Red-Black Rules.

First, if the deleted_node is black, and it has a red parent and a red child, when we remove the deleted_node from the tree, the red parent and red child become neighbors, and this causes a Red Enemies violation.

Second, by removing a black node from the tree, we, by definition, decrease the black height of its path by 1. Accordingly, this path will now have a smaller black height than other paths in the tree, since before we removed this node, all paths had the same black height.

And so, if—and only if—the deleted_node is black, it is necessary to launch the Fixing Phase.

Code Implementation: Phase 1 of Red-Black Tree Deletion

Before we dive into the Fixing Phase, let me drop the code for Phase 1 here. In addition to the actual delete method, there's also a search method (which is important for the tree to have in any case) as well as a replace_with_successor_node helper method:

```
def search(self, value):
    current_node = self.root
    while current_node:
        if current_node.value == value:
            return current_node

        if value < current_node.value:
            current_node = current_node.left_child
        else:
            current_node = current_node.right_child

    return None

def delete(self, value_to_delete):
    node_with_value_to_delete = self.search(value_to_delete)

    if not node_with_value_to_delete:
        return None
```

```
    if node_with_value_to_delete.left_child and \
    node_with_value_to_delete.right_child:
        deleted_node = self.replace_with_successor_node(node_with_value_to_delete)
        # When the deleted node has 2 children, the fixup_node is the right
        # child of the successor node. The successor node is the node that
        # moved into the place of the deleted node:
        fixup_node = deleted_node.right_child

        # If the successor node has no right child, we set the fixup_node to be
        # a phantom black node whose parent is the successor node. (This phantom
        # node isn't actually set as a child of any other node, though.)
        if not fixup_node:
            fixup_node = rbt_node.Node(None, "black")
            fixup_node.parent = deleted_node.parent

        deleted_color = deleted_node.color
    else:  # deleted node has 0 or 1 children
        deleted_color = node_with_value_to_delete.color
        fixup_node = child_of_deleted_node = \
            (node_with_value_to_delete.left_child or
             node_with_value_to_delete.right_child)

        # If the deleted node has 0 children, the fixup_node will be
        # a phantom black node whose parent is the parent of the deleted node:
        if not fixup_node:
            fixup_node = rbt_node.Node(None, "black")
            fixup_node.parent = node_with_value_to_delete.parent

        if not node_with_value_to_delete.parent:
            self.root = child_of_deleted_node
            child_of_deleted_node.parent = None
        else:
            if self.is_a_left_child(node_with_value_to_delete):
                node_with_value_to_delete.parent.left_child = \
                    child_of_deleted_node
            elif self.is_a_right_child(node_with_value_to_delete):
                node_with_value_to_delete.parent.right_child = \
                    child_of_deleted_node

            if child_of_deleted_node:
                child_of_deleted_node.parent = \
                    node_with_value_to_delete.parent

    if deleted_color == "black":
        self.fix_delete(fixup_node)

    return node_with_value_to_delete

def replace_with_successor_node(self, node):
    successor_node = node.right_child

    if not successor_node.left_child:
        node.value = successor_node.value
        node.right_child = successor_node.right_child
        if successor_node.right_child:
```

```
            successor_node.right_child.parent = node
        return successor_node

    while successor_node.left_child:
        parent_of_successor_node = successor_node
        successor_node = successor_node.left_child

    if successor_node.right_child:
        parent_of_successor_node.left_child = successor_node.right_child
        successor_node.right_child.parent = parent_of_successor_node
    else:
        parent_of_successor_node.left_child = None

    node.value = successor_node.value

    return successor_node
```

I'll let you peruse and enjoy reading through the code; much of it was covered in Volume 1. I am, however, going to zero in on one particular aspect of this code—the fixup_node—because it's the preparatory step for the Fixing Phase, which we'll explore next.

As noted, we launch the Fixing Phase only if the deleted_node is black. As you can see in the previous code, we do this by calling the fix_delete method. The fix_delete method is the Fixing Phase, and I'll share that code in a little bit. For now, note that when we call the fix_delete method, we pass in an argument called fixup_node. Let's talk about that fixup_node for a bit, since it's the focal point of the Fixing Phase.

The fixup_node is *not* the deleted_node, nor is it necessarily the node causing a violation. Rather, it's simply the starting point for analyzing what modifications need to be made in the tree to fix the violation. That is, the algorithm inspects the fixup_node in relation to its neighbors, and based on which nodes are red and which nodes are black, decides what approach to take to fix the tree. In short, the fixup_node is the focus of the Fixing Phase.

However, we choose which node should be the fixup_node in Phase 1. The rules of choosing the fixup_node revolve around the node *whose value* we're removing from the tree. This gets back to that important nuance I discussed earlier, which is that the *value* we're removing from the tree doesn't always belong to the *node instance* we're removing from the tree. The rules of choosing the fixup_node are based on the node that contains the *value* we're deleting. In my implementation, I called this the node_with_value_to_delete.

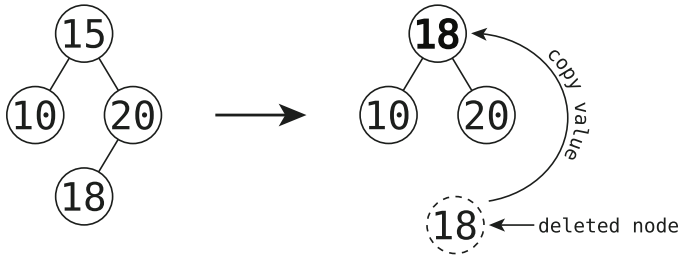
To reiterate: the deleted_node is the node *instance* we disconnect from the tree. Here, though, we focus on the node_with_value_to_delete, which is the node that contains the *value* we're deleting from the tree.

In any case, the rules for choosing the fixup_node are as follows:

- If the node_with_value_to_delete has *two* children, then the fixup_node is the right child of the successor node. The successor node is, in fact, the deleted_node instance we removed from the tree, and the fixup_node is its right child.
- If the node_with_value_to_delete has *one* child, then we designate that very child to be the fixup_node.
- If the node_with_value_to_delete has *zero* children, then we create a new, *temporary fake node* to serve as the fixup_node. We don't actually connect this fake node by making it a child of any other node in the tree. However, we do set the fake node's parent attribute to be the node_with_value_to_delete.

Although that last point about making a fake node may seem strange, it makes a little more sense when you recall the idea of the "phantom nodes" we discussed in Chapter 5. That is, every leaf node really has phantom nodes that affect the true black height of the tree. Here, then, we bring that phantom node to life (yikes!) by making an actual Node instance to represent that phantom node. In this particular case, this phantom node serves as the fixup_node.

In truth, in some cases, we have to make a fake node even when the node_with_value_to_delete has two children. That is, I stated that when the node_with_value_to_delete has two children, the fixup_node is the right child of the successor node. But sometimes, the successor node doesn't *have* a right child, such as the node instance (18) being deleted in the following scenario:



Here, too, because the successor node doesn't have a right child, we create a fake child of the successor node to serve as the fixup_node.

I know that this all seems weird and arbitrary, but stay with me on this. Next, we'll see how the fixup_node plays a starring role in the Fixing Phase.

Phase 2: The Fixing Phase

Okay, now for the fun part! It's time to delve into the Fixing Phase.

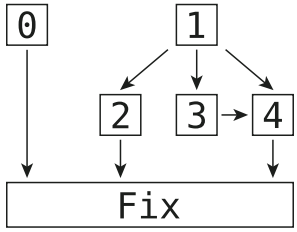
The gist of the Fixing Phase is that it looks at the `fixup_node` and its surrounding nodes, and takes stock of which nodes have which colors. There are basically *five* different scenarios that we can encounter, and for each of these five cases, the Fixing Phase algorithm performs a different set of steps to, well, fix things up. These steps typically involve rotations and color flips.

I will refer to these five scenarios as Cases 0, 1, 2, 3, and 4. Note that these have nothing to do with Cases A through D from Phase 1. (That's why I used numbers for this set of cases and letters for the Phase 1 set of cases; I'm trying to head off any potential confusion.)

The Five Cases

Before I describe Cases 0 through 4, I will first note that the "fix" for some of these cases is to transform them into *another case*. When this happens, the transformed case will still require additional fixing. For example, Case 2, like all cases, needs to be fixed. So if we transform a Case 1 into a Case 2, we'll still need to perform the Case 2 fixing algorithm to resolve the remaining Red-Black Rule violations.

Here is a visual that shows which cases we can fix directly, and which cases first need to be transformed into another case:



Let me put this diagram into words. Cases 0, 2, and 4 can be fixed directly (although a different set of steps is needed to fix each case). Case 1 needs to first be transformed into either Case 2, 3, or 4, depending on various *subcases* of Case 1. Case 3 needs to first be transformed into Case 4. I know, it's all so much fun!

Now, there's actually a little more complexity to this than I previously described, but I'm laying things out one step at a time so that this is not more overwhelming than it has to be. As we proceed, I'll reveal each layer of complexity.

We're now ready to look at our first scenario, Case 0. As I present each case and the steps for fixing it, I will not always get into the nitty-gritty of how

each fixing step resolves every violation—because, hey, this is just an article after all.

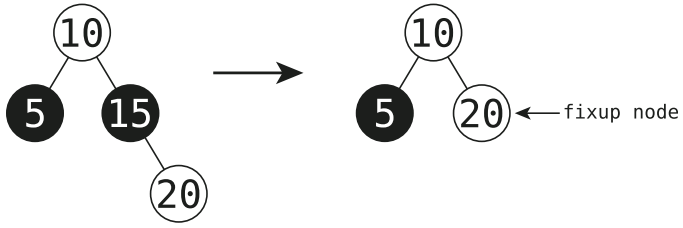
Case 0

Case 0 really consists of two different possible subcases, but because the fix for both subcases is identical, I just lump them all into one case.

In short, Case 0 is when the `fixup_node` is either red or the root of the tree. The fix for this is simply to color the `fixup_node` black. Again, without going too deep into the weeds, here's the gist of why.

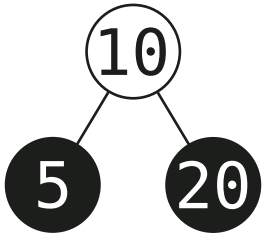
If we're dealing with a root `fixup_node`, whether the root is red or black doesn't matter for violations of the Black Height Rule. For if we turn a red root black, we're increasing the black height of *all* the paths equally—namely, by 1. However, the root being red may be a violation of the Red Enemies Rule if the root happens to have a red child. And so, coloring the root black resolves the Red Enemies violation without causing any possible Black Height violation.

In the case where the `fixup_node` is *not* the root but is red, this will only have occurred because we removed its black parent. (It certainly didn't have a red parent, because that would have been a Red Enemies problem.) For example, look what happens when we delete the 15 node from this tree:



The 20 node is now the `fixup_node`, and we now have *two* violations on our hands. That is, the 10 and 20 form Red Enemies, and the tree's left path has a black height of 1 while the tree's right path has a black height of 0.

We can fix all of this by simply coloring the `fixup_node` (20) black:



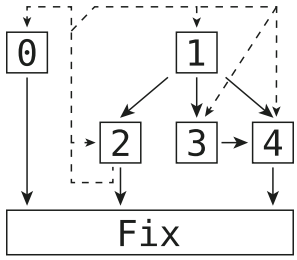
Okay, that takes care of Case 0, which, admittedly, is the easiest case and has the easiest fix. We're about ready to move on to Cases 1 through 4, but first, one more small preamble is in order.

The Fixing Phase Loop

In truth, if we have any of the Cases 1 through 4 on our hands, the Fixing Phase initiates a loop. That is, in our implementation, the loop is a `while` loop that's set to run as long as we're not dealing with Case 0. However, in practice, the loop can really only run more than once if we encounter Case 2, as I'll now explain.

The reason for the loop is that when we fix Case 2, although this does fix the nodes surrounding the `fixup_node`, this fix may cause new violations further up the tree. When this happens, we update the `fixup_node` to be some node higher up the tree, and this new `fixup_node` may turn out to be any Case from 0 through 4. If it's Case 0, we resolve it by turning the `fixup_node` black, but if it's any of Cases 1 through 4, the loop begins again. We then resolve whatever case we have on our hands in the appropriate way. If the new case happens to be Case 2, then the loop may repeat yet again.

The following illustration is an update to our case "flow" diagram that reflects this loop. The dashed lines indicate that when we resolve Case 2, this will cause a new case farther up the tree:



We may end up designating new `fixup_nodes` multiple times, but eventually the tree will be fully resolved.

We're now ready to look at the details of Cases 1 through 4. Now, in our implementation, our code handles the cases in the order of Case 1, Case 2, Case 3, and Case 4, as you might expect. However, in presenting these cases to you, I'm going to go in *backward* order. This is because it's easier to grasp things if you learn about the cases in the order of Case 4, Case 3, Case 2, and Case 1. (The reason I don't simply re-number the cases is because I want to stay consistent with the literature out there that labels the cases in this way.)

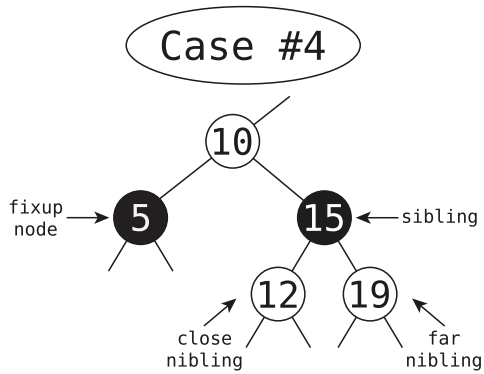
Let's do it!

Case 4

Case 4 is when the fixup_node's sibling is black, and its "far nibling" is red. Here's what that means.

First, the word "nibling" is a real (and awesome) English word that is the gender-neutral word for niece or nephew. Just as the word *sibling* is the gender-neutral term for a brother or sister; the word *nibling* does the same for nieces and nephews. As such, a node's niblings are its siblings' children.

Second, if a node has two niblings, one nibling is closer to it, and one is farther:



Here, you can see that the 5 node has the sibling 15 and niblings 12 and 19. Since 12 is closer to the 5 than the 19, I call the 12 the "close nibling" and the 19 the "far nibling."

This diagram also represents Case #4, given that the 5 is the fixup_node. The fixup_node's sibling (15) is black, and the fixup_node's far nibling is red. It's Case #4 at its best.

Note that the previous visual is merely a section of a larger tree. I didn't want to complicate things by drawing a large, complex tree; instead, I'm only focusing on the section of the tree that's relevant to describing Case 4.

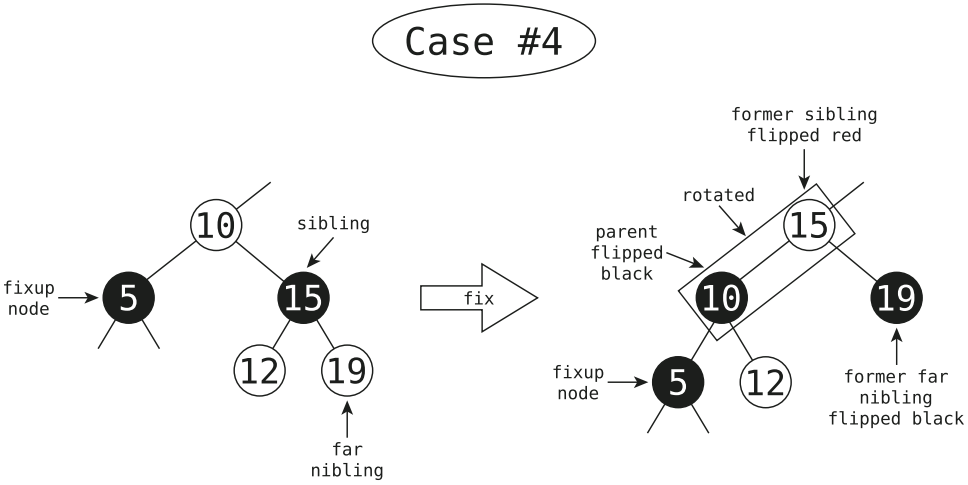
Now, recall that we only initiated the Fixing Phase in the first place because we deleted a black node. (Again, if the deleted_node was red, we skip the Fixing Phase altogether.) This means that we definitely have a Black Height Rule violation on our hands. Although this isn't immediately apparent in the previous diagram, again, I'm only showing part of a larger tree. The 12 and 19 nodes may each have black children that aren't displayed here. In any case, because this Fixing Phase was initiated because we deleted a black node, it's

certain that the path with the fixup node has a smaller black height than other paths in the tree.

The algorithm for fixing Case #4 is:

1. Make the fixup_node's sibling color identical to the fixup_node's parent's color.
2. Turn the fixup_node's parent black.
3. Turn the fixup_node's far nibling black.
4. Rotate the fixup_node's parent and sibling.

Here's the before and after of this fix:



Although these steps may seem a bit wonky, they effectively add a black node to the fixup_node's branch. This is the fix we need, since our whole problem was that the fixup_node's branch was missing one black node.

Also, had the fixup_node (5) been red, we also would have had a Red Enemies violation between the 5 and the 10. Coloring the 10 black solves this problem as well.

After these moves are made, we then update the fixup_node variable to point to the tree's root. This will ensure that our Fixing Phase loop doesn't run again, since the loop doesn't run when we have Case 0 on our hands, which is the case when the root is the fixup_node.

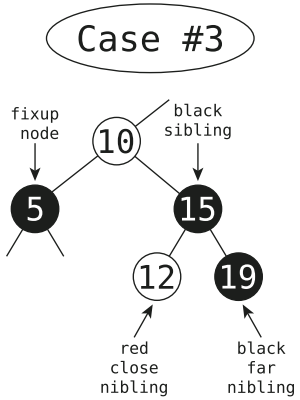
Once the loop is done, there's one final step, which is actually executed at the end of the fix for *all* cases—and this is to color the fixup_node black. Now, in our example, the fixup_node happened to already be black. However, in the case that it was red, we'd flip it to black at this point. This is mainly done to execute the fix for Case 0, which essentially skips to this point since it bypasses the Fixing Phase loop altogether.

Case 3

Case 3 is quite specific—it occurs when the following three conditions all hold true:

1. The fixup_node's sibling is black.
2. The fixup_node's close nibling is red.
3. The fixup_node far nibling is black.

Here's an example:

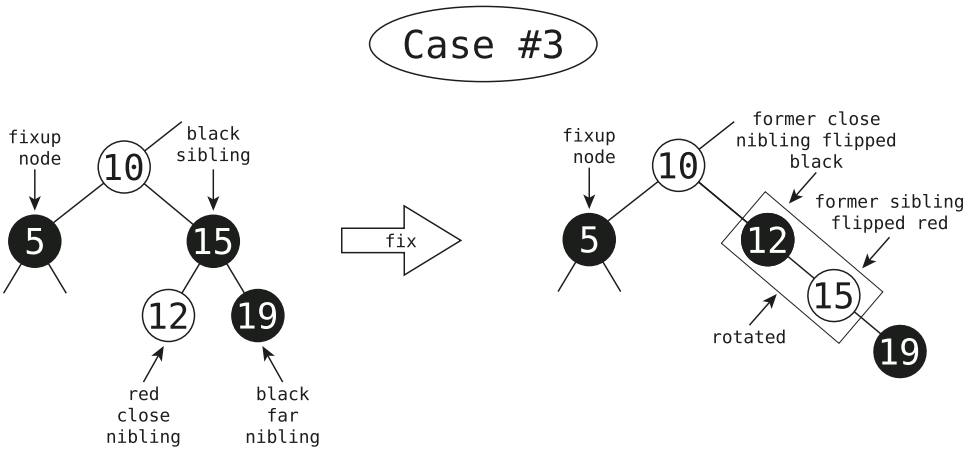


As with Case 4, the problem here is that the fixup_node's branch just lost a black node, and so has a smaller black height than the other branches.

The trick to fixing Case 3 is to turn it into Case 4. From there, we apply the Case 4 fix as I detailed in the previous section. Here's the algorithm for converting Case 3 into Case 4:

1. Color the close nibling black.
2. Color the sibling red.
3. Rotate the sibling and close nibling.

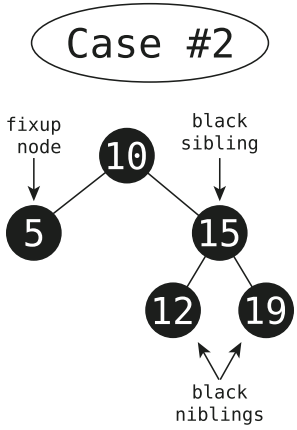
Here's a visual of this conversion:



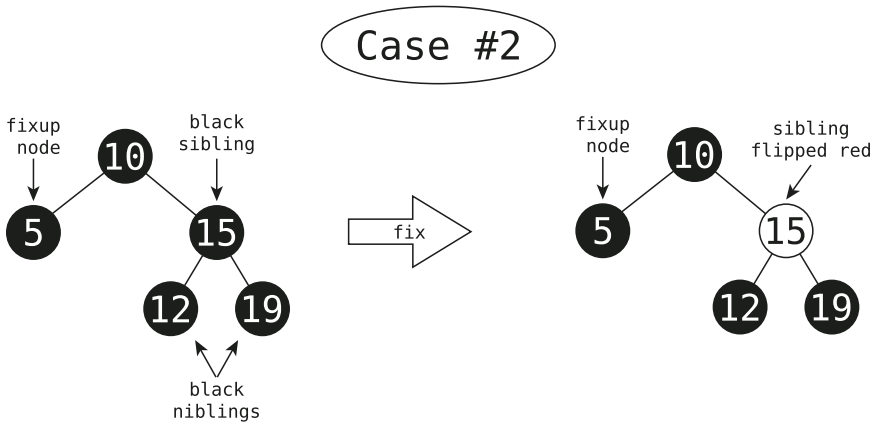
The tree has now been transformed into Case 4, as the fixup_node's new sibling is black, and the new far nibling is red. As such, the code proceeds with the Case 4 fixup algorithm.

Case 2

Case 2 is when the fixup_node's sibling and niblings are all black. For example:

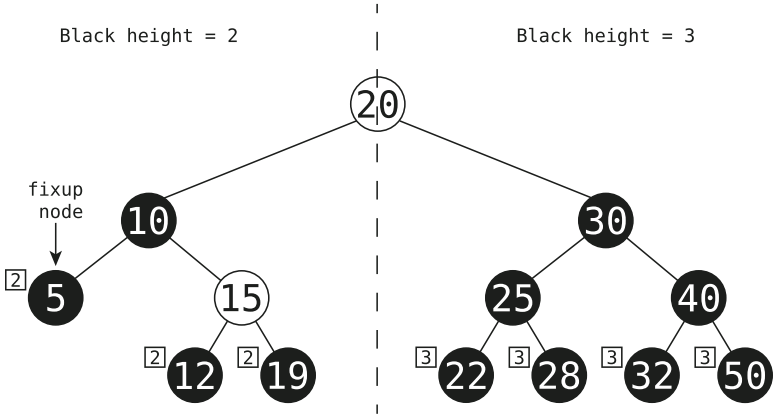


The problem here (just like Cases 3 and 4) is that the fixup_node's branch just lost a black node, and so has a smaller black height than the other branches. The initial fix is to color the sibling *red*:



This helps things a bit, because when we encounter Case 2, the fixup_node's branch has one less black node than the sibling's branch. By coloring the sibling red, we thereby subtract a black node from the sibling's branch as well, and so the black heights of both branches become the same once again.

However, this doesn't necessarily fix the *entire* tree, because the fixup_node and its sibling may both be part of one branch of a larger tree. For example:



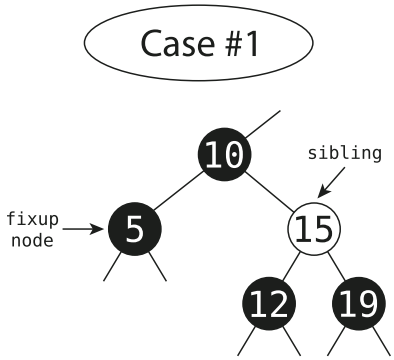
As you can see, the fixup_node and its sibling live in the left half of the tree. They each lost a black node, and so each path in the left half of the tree has a smaller black height than any path in the right half of the tree.

And so, what we do next is to reassign the fixup_node to a node *farther up the tree*. Specifically, the fixup_node's *parent* becomes the new fixup_node.

Once we do this, we begin the Fixing Phase loop once again and analyze the new fixup_node and its local nodes. Once again, we'll find ourselves in any one of Cases 0 through 4, and we'll then apply the appropriate fix to the case at hand.

Case 1

Case 1 is the easiest to describe: it occurs whenever the fixup_node's sibling is *red*, period. Here's an example:

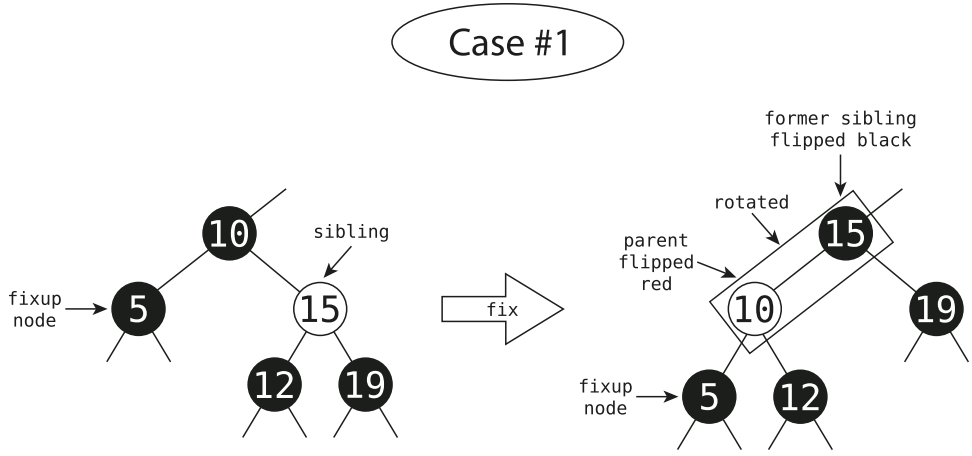


Once again, the problem at hand is that the fixup_node's branch lost a black node, and so currently has a smaller black height than the other branches of the tree.

The immediate fix is to make the following changes to the tree:

1. Color the fixup_node's sibling black.
2. Color the fixup_node's parent red.
3. Rotate the fixup_node's parent and sibling.

Here's what this fix looks like:



Although it's not necessarily intuitive, this converts Case 1 into either Case 2, Case 3, or Case 4. That is, it's inevitable that these color flips and rotations will turn Case 1 into one of the other cases. From there, we can apply the appropriate fix for whatever the new case turns out to be.

Okay, we did it! That wraps up all of the cases, as well as the entire deletion algorithm.

Code Implementation: Phase 2 of Red-Black Tree Deletion (The Fixing Phase)

Here is the code for the Fixing Phase of red-black tree deletion:

```
def fix_delete(self, fixup_node):
    while fixup_node is not self.root and self.is_black_or_blank(fixup_node):
        # When fixup_node, whether real or "phantom", is a left child:
        if fixup_node.parent.right_child and \
        fixup_node.parent.right_child is not fixup_node:
            sibling = fixup_node.parent.right_child
            # Case 1:
            if sibling.color == "red":
                # print("Left, Case #1")
                sibling.color = "black"
                fixup_node.parent.color = "red"
                self.rotate_counterclockwise(fixup_node.parent,
                                             fixup_node.parent.right_child)
                sibling = fixup_node.parent.right_child
            # Case 2:
            if self.is_black_or_blank(sibling.left_child) and \
            self.is_black_or_blank(sibling.right_child):
                # print("Left, Case #2")
                sibling.color = "red"
                fixup_node = fixup_node.parent
            else:
            # Case 3:
                if self.is_black_or_blank(sibling.right_child):
                    # print("Left, Case #3")
                    sibling.left_child.color = "black"
                    sibling.color = "red"
                    self.rotate_clockwise(sibling,
                                          sibling.left_child)
                    sibling = fixup_node.parent.right_child
            # Case 4:
                # print("Left, Case #4")
                sibling.color = fixup_node.parent.color
                fixup_node.parent.color = "black"
                sibling.right_child.color = "black"
                self.rotate_counterclockwise(fixup_node.parent,
                                             fixup_node.parent.right_child)
                fixup_node = self.root
        # When fixup_node is a right child:
        else:
            sibling = fixup_node.parent.left_child
            # Case 1:
            if sibling.color == "red":
```

```
                # print("Right, Case #1")
                sibling.color = "black"
                fixup_node.parent.color = "red"
                self.rotate_clockwise(fixup_node.parent,
                                      fixup_node.parent.left_child)
                sibling = fixup_node.parent.left_child
            # Case 2:
            if self.is_black_or_blank(sibling.left_child) and \
            self.is_black_or_blank(sibling.right_child):
                # print("Right, Case #2")
                sibling.color = "red"
                fixup_node = fixup_node.parent
            else:
            # Case 3:
                if self.is_black_or_blank(sibling.left_child):
                    # print("Right, Case #3")
                    sibling.right_child.color = "black"
                    sibling.color = "red"
                    self.rotate_counterclockwise(sibling,
                                                 sibling.right_child)
                    sibling = fixup_node.parent.left_child
            # Case 4:
                # print("Right, Case #4")
                sibling.color = fixup_node.parent.color
                fixup_node.parent.color = "black"
                sibling.left_child.color = "black"
                self.rotate_clockwise(fixup_node.parent,
                                      fixup_node.parent.left_child)
                fixup_node = self.root

    fixup_node.color = "black"
    return fixup_node

def is_black_or_blank(self, node):
    return (not node) or node.color == "black"
```

There's a lot here, but I've used comments to delineate which lines of code address which case. I didn't highlight Case 0, though, and that's because the while loop doesn't run at all when Case 0 occurs. That is, Case 0, which is when the fixup_node is red or the root, is specifically excluded by the while loop. Again, the fix for Case 0 occurs *after* the loop, where we color the fixup_node black.

One important thing you'll notice is that the code inside the while loop is wrapped in a giant if statement. This statement deals with two paths, namely, whether the fixup_node is a left or right child. These two paths are doing the same thing, but are the mirror opposite of each other.

For example, if the fixup_node is a left child, then the code looks to its parent's *right* child to locate the sibling. But if the fixup_node is a right child, then the

sibling will be the parent's *left* child. Along the same lines, the close and far niblings and the direction of the rotations will all depend on whether the `fix-up_node` is a left or right child. Otherwise, the two paths are effectively making the same moves.

Well, that wraps up the exciting world that is red-black tree deletion. It's complicated, but it works. If you've read this until the end, you're a trooper! Nice job.