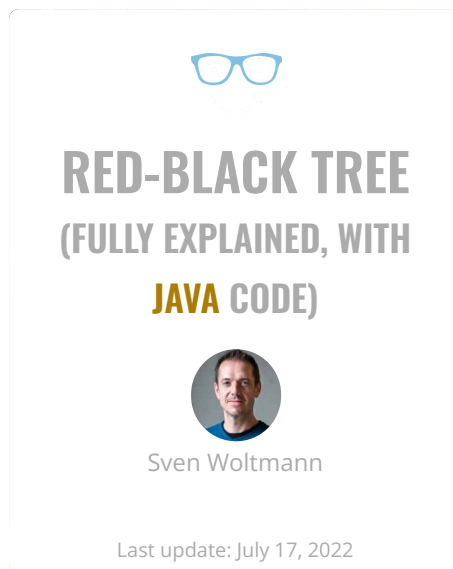




eu



The red-black tree is a widely used concrete implementation of a [self-balancing binary search tree](#). In the JDK, it is used in [TreeMap](#), and since Java 8, it is also used for bucket collisions in [HashMap](#). How does it work?

In this article, you will learn:

- What is a red-black tree?
- How do you insert elements into a red-black tree? How do you remove them?
- What are the rules for balancing a red-black tree?
- How to implement a red-black tree in Java?
- How to determine its time complexity?
- What distinguishes a red-black tree from other data structures?

You can find the source code for the article in this [GitHub repository](#).

What Is a Red-Black Tree?

A red-black tree is a self-balancing binary search tree, that is, a binary search tree that automatically maintains some balance.

Each node is assigned a color (red or black). A set of rules specifies how these colors must be arranged (e.g., a red node may not have red children). This arrangement ensures that the tree maintains a certain balance.

After inserting and deleting nodes, quite complex algorithms are applied to check compliance with the rules – and, in case of deviations, to restore the prescribed properties by recoloring nodes and rotations.

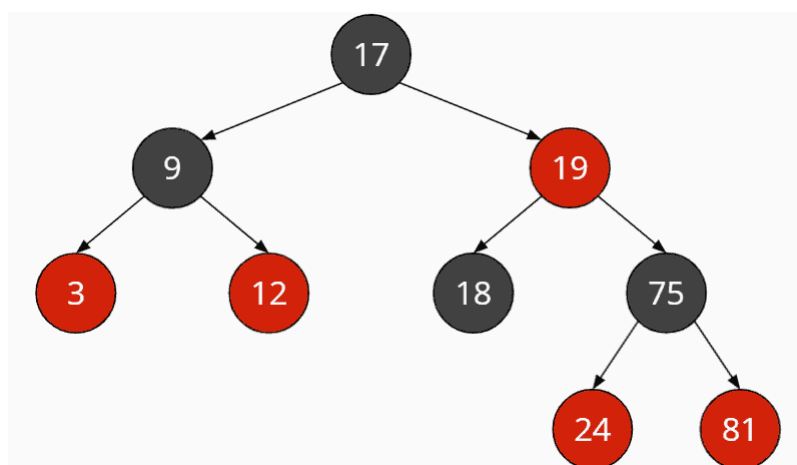
NIL Nodes in Red-Black Trees

In the literature, red-black trees are depicted with and without so-called NIL nodes. A NIL node is a leaf that does not contain a value. NIL nodes become relevant for the algorithms later on, e.g., to determine colors of uncle or sibling nodes.

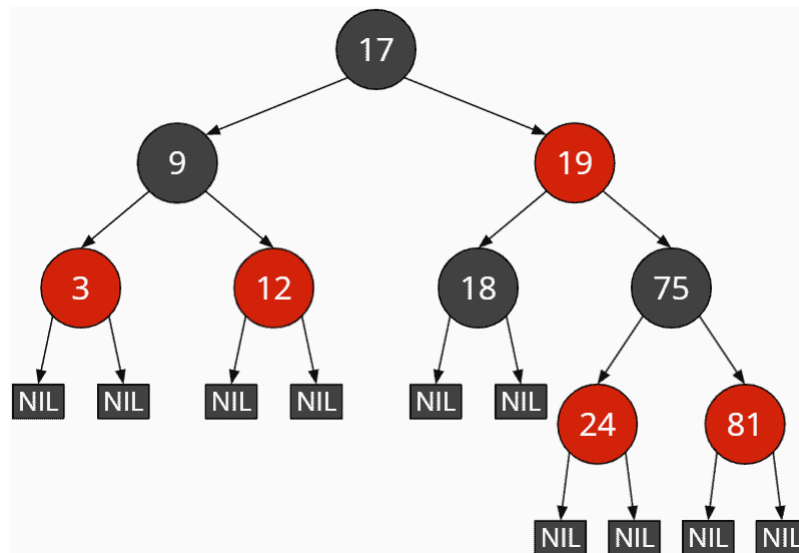
In Java, NIL nodes can be represented simply by `null` references; more on this later.

Red-Black Tree Example

The following example shows two possible representations of a red-black tree. The first image shows the tree without (i.e., with implicit) NIL leaves; the second image shows the tree with explicit NIL leaves.



Red-black tree with implicit NIL leaves



Red-black tree with explicit NIL leaves

In the course of this tutorial, I will generally refrain from showing the NIL leaves. When explaining the insert and delete operations, I will show them sporadically if it facilitates understanding the respective algorithm.

Red-Black Tree Properties

The following rules enforce the red-black tree balance:

1. Each node is either red or black.
2. (The root is black.)
3. All NIL leaves are black.
4. A red node must not have red children.
5. All paths from a node to the leaves below contain the same number of black nodes.

Rule 2 is in parentheses because it does not affect the tree's balance. If a child of a red root is also red, the root must be colored black according to rule 4. However, if a red root has only black children, there is no advantage in coloring the root black.

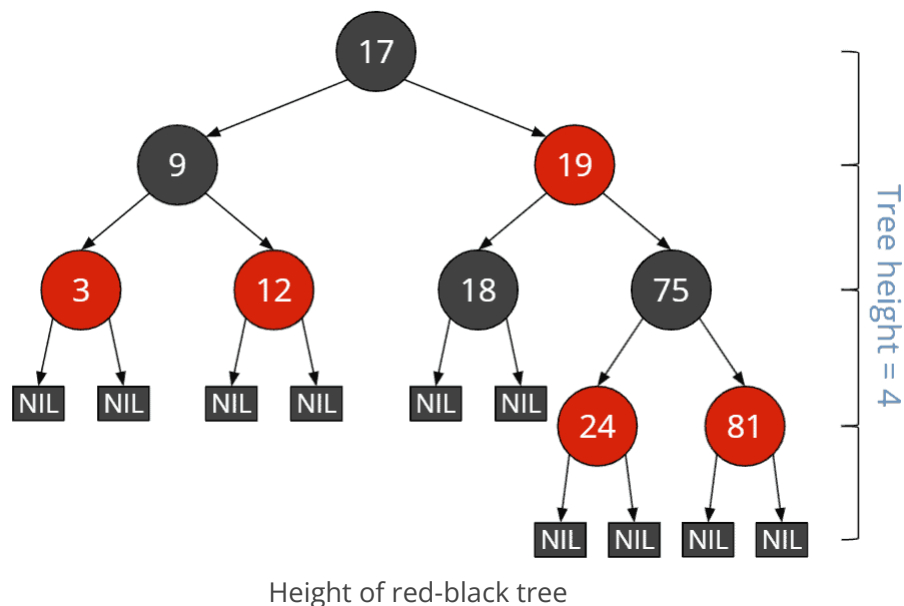
Therefore, rule 2 is often omitted in the literature.

When explaining the insert and delete operations and in the Java code, I will point out where there would be differences if we would also implement rule 2. So much in advance: The difference is only one line of code per operation :)

By the way, from rules 4 and 5 follows that a red node always has either two NIL leaves or two black child nodes with values. If it had one NIL leaf and one black child with value, then the paths through this child would have at least one more black node than the path to the NIL leaf, which would violate rule 5.

Height of a Red-Black Tree

We refer to the height of the red-black tree as the maximum number of nodes from the root to a NIL leaf, not including the root. The height of the red-black tree in the example above is 4:



From rules 3 and 4 follows:

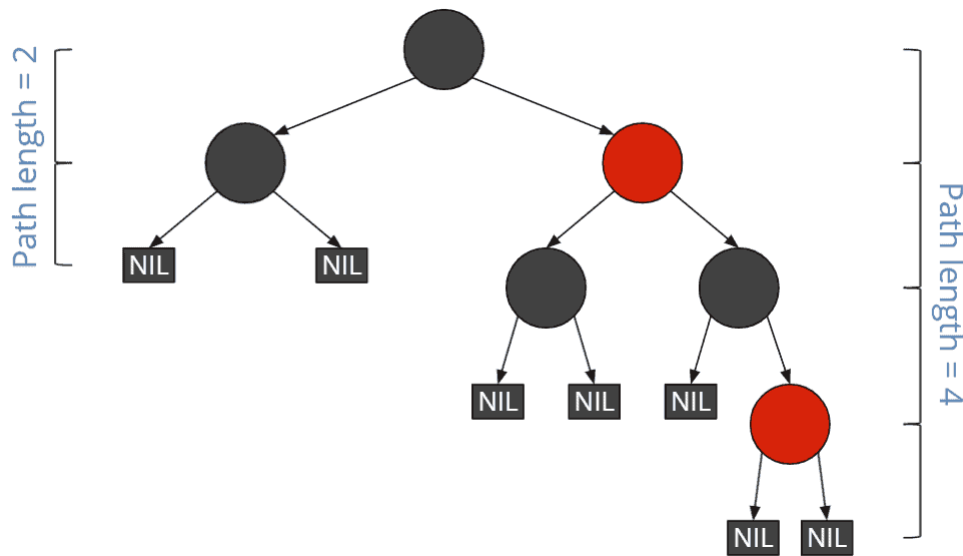
The *longest* path from the root to a leaf (not counting the root) is at most twice as long as the *shortest* path from the root to a leaf.

That is easily explained:

Let's assume that the shortest path has (in addition to the root) n black nodes and no red nodes. Then we could add another n red nodes before each black node without

breaking rule 3 (which we could reword to: no two red nodes may follow each other).

The following example shows the shortest possible path through a red-black tree of height four on the left and the longest possible path on the right:



Shortest and longest path in a red-black tree

The paths to the NIL leaves on the left have a length (excluding the root) of 2. The paths to the NIL leaves on the bottom right have a length of 4.

Black Height of a Red-Black Tree

Black height is the number of black nodes from a given node to its leaves. The black NIL leaves are counted, the start node is not.

The black height of the entire tree is the number of black nodes from the root (this is not counted) to the NIL leaves.

The black height of all red-black trees shown so far is 2.

Red-Black Tree Java Implementation

As a starting point for implementing the red-black tree in Java, I use the [Java source code for the binary search tree](#) from the second part of the binary tree series.

Nodes are represented by the `Node` class. For simplicity, we use `int` primitives as the node value.

To implement the red-black tree, besides the child nodes `left` and `right`, we need a reference to the parent node and the node's color. We store the color in a `boolean`, defining red as `false` and black as `true`.

```
public class Node {  
    int data;  
  
    Node left;  
    Node right;  
    Node parent;  
  
    boolean color;  
  
    public Node(int data) {  
        this.data = data;  
    }  
}
```

We implement the red-black tree in the `RedBlackTree` class. This class extends the `BaseBinaryTree` class presented in the second part of the series (which essentially provides a `getRoot()` function).

We will add the operations (insert, search, delete) in the following sections, step by step.

But first, we have to define some helper functions.

Red Black Tree Rotation

`Insertion` and `deletion` work basically as described in the article about binary search trees.

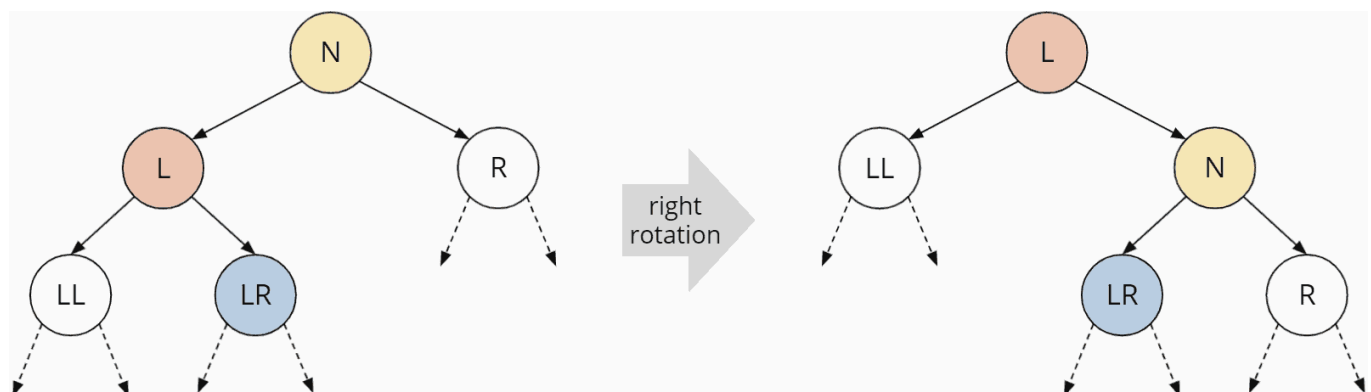
After insertion and deletion, the red-black rules ([see above](#)) are reviewed. If they have been violated, they must be restored. That happens by recoloring nodes and by rotations.

The rotation works precisely like with [AVL trees](#), which I described in the previous tutorial. I'll show you the corresponding diagrams again here. You can find detailed explanations in the section "[AVL tree rotation](#)" of the article just mentioned.

Right Rotation

The following graphic shows a right rotation. The colors have no relation to those of the red-black tree. They are only used to track the node movements better.

The left node *L* becomes the new root; the root *N* becomes its right child. The right child *LR* of the pre-rotation left node *L* becomes the left child of the post-rotation right node *N*. The two white nodes *LL* and *R* do not change their relative position.



Right rotation in a red-black tree

The Java code is slightly longer than in the AVL tree – for the following two reasons:

1. We also need to update the parent references of the nodes (in the AVL tree, we worked without parent references).
2. We need to update the references to and from the pre-rotation top node's parent (*N* in the graphic). For the AVL tree, we did that indirectly by returning the new root of the rotated subtree and "hooking" the rotation into the recursive call of the insert and delete operations.

You can find the implementation of the right rotation in the [source code starting at line 358](#):

```
private void rotateRight(Node node) {
    Node parent = node.parent;
    Node leftChild = node.left;

    node.left = leftChild.right;
    if (leftChild.right != null) {
        leftChild.right.parent = node;
    }

    leftChild.right = node;
    node.parent = leftChild;

    replaceParentsChild(parent, node, leftChild);
}
```

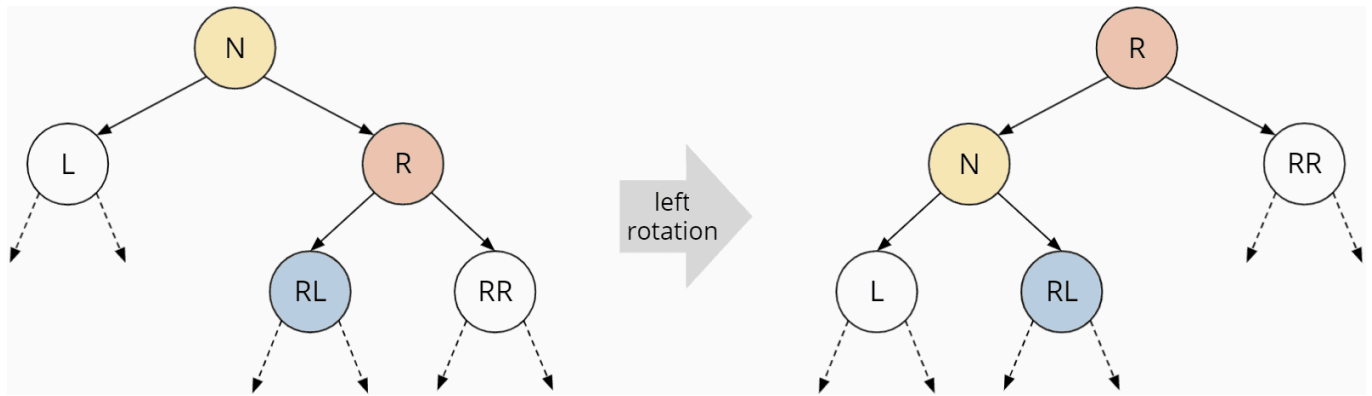
The `replaceParentsChild()` method called at the end sets the parent-child relationship between the parent node of the former root node N of the rotated subtree and its new root node L . You can find it [in the code starting at line 388](#):

```
private void replaceParentsChild(Node parent, Node oldChild, Node new
    if (parent == null) {
        root = newChild;
    } else if (parent.left == oldChild) {
        parent.left = newChild;
    } else if (parent.right == oldChild) {
        parent.right = newChild;
    } else {
        throw new IllegalStateException("Node is not a child of its paren
    }

    if (newChild != null) {
        newChild.parent = parent;
    }
}
```


Left Rotation

Left rotation works analogously: The right node R moves up to the top. The root N becomes the left child of R . The left child RL of the formerly right node R becomes the right child of the post-rotation left node N . L and RR do not change their relative position.



Left rotation in a red-black tree

Here is the Java code for the left rotation ([source code, starting at line 373](#)):

```
private void rotateLeft(Node node) {
    Node parent = node.parent;
    Node rightChild = node.right;

    node.right = rightChild.left;
    if (rightChild.left != null) {
        rightChild.left.parent = node;
    }

    rightChild.left = node;
    node.parent = rightChild;

    replaceParentsChild(parent, node, rightChild);
}
```

Red-Black Tree Operations

Like any binary tree, the red-black tree provides operations to find, insert, and delete nodes. We will go through these operations step by step in the following sections.

At this point, I would like to recommend the [red-black tree simulator by David Galles](#). It allows you to animate any insert, delete and search operations graphically.

Red-Black Tree Search

The search works like in any binary search tree: We first compare the search key with the root. If the search key is smaller, we continue the search in the left subtree; if the search key is larger, we continue the search in the right subtree.

We repeat this until we either find the node we are looking for – or until we reach a NIL leaf (in Java code: a `null` reference). Reaching a NIL leaf would mean that the key we are looking for does not exist in the tree.

For a graphical representation of the search, see the example in the [article about binary search trees](#).

For the red-black tree, we implement an iterative variant of the search. You can find it in the [source code starting at line 14](#):

```
public Node searchNode(int key) {  
    Node node = root;  
    while (node != null) {  
        if (key == node.data) {  
            return node;  
        } else if (key < node.data) {  
            node = node.left;  
        } else {  
            node = node.right;  
        }  
    }  
  
    return null;  
}
```

This code should be self-explanatory.

In the ["Searching" section](#) of the article mentioned above, you can also find a recursive version of the search.

Red-Black Tree Insertion

To insert a new node, we first proceed as described in the ["binary search tree insertion"](#) section of the corresponding article. I.e., we search for the insertion position from the root downwards and attach the new node to a leaf or half-leaf.

You can find the code in the [RedBlackTree class](#), starting at line 29:

```
public void insertNode(int key) {
    Node node = root;
    Node parent = null;

    // Traverse the tree to the left or right depending on the key
    while (node != null) {
        parent = node;
        if (key < node.data) {
            node = node.left;
        } else if (key > node.data) {
            node = node.right;
        } else {
            throw new IllegalArgumentException("BST already contains a node");
        }
    }

    // Insert new node
    Node newNode = new Node(key);
    newNode.color = RED;
    if (parent == null) {
        root = newNode;
    } else if (key < parent.data) {
        parent.left = newNode;
    } else {
        parent.right = newNode;
    }
    newNode.parent = parent;
}
```

```
    fixRedBlackPropertiesAfterInsert(newNode);  
}
```

We initially color the new node red so that rule 5 is satisfied, i.e., all paths have the same number of black nodes after insertion.

However, if the parent node of the inserted node is also red, we have violated rule 4. We then have to repair the tree by recoloring and/or rotating it so that all rules are satisfied again. That is done in the `fixRedBlackPropertiesAfterInsert()` method, which is called in the last line of the `insertNode()` method.

During the repair, we have to deal with five different cases:

- Case 1: New node is the root
- Case 2: Parent node is red and the root
- Case 3: Parent and uncle nodes are red
- Case 4: Parent node is red, uncle node is black, inserted node is "inner grandchild"
- Case 5: Parent node is red, uncle node is black, inserted node is "outer grandchild"

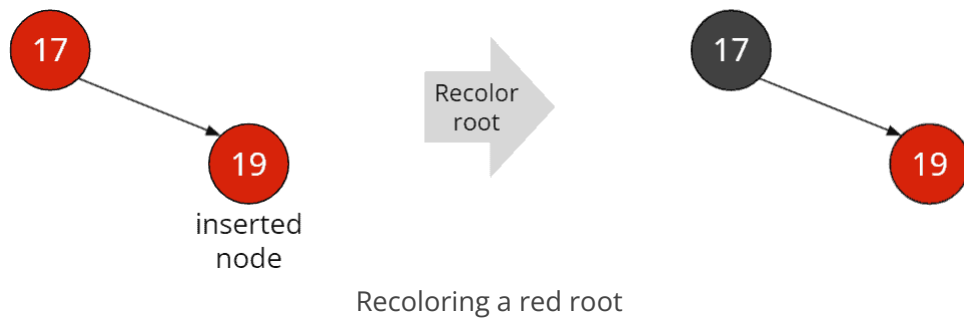
The five cases are described below.

Case 1: New Node Is the Root

If the new node is the root, we don't have to do anything else. Unless we work with rule 2 ("the root is always black"). In that case, we would have to color the root black.

Case 2: Parent Node Is Red and the Root

In this case, rule 4 ("no red-red!") is violated. All we have to do now is to color the root black. That leads to rule 4 being complied with again.



And rule 5? Since the root is not counted in this rule, all paths still have one black node (the NIL leaves not displayed in the graphic). And if we would count the root, then all paths would now have two black nodes instead of one – that would also be allowed.

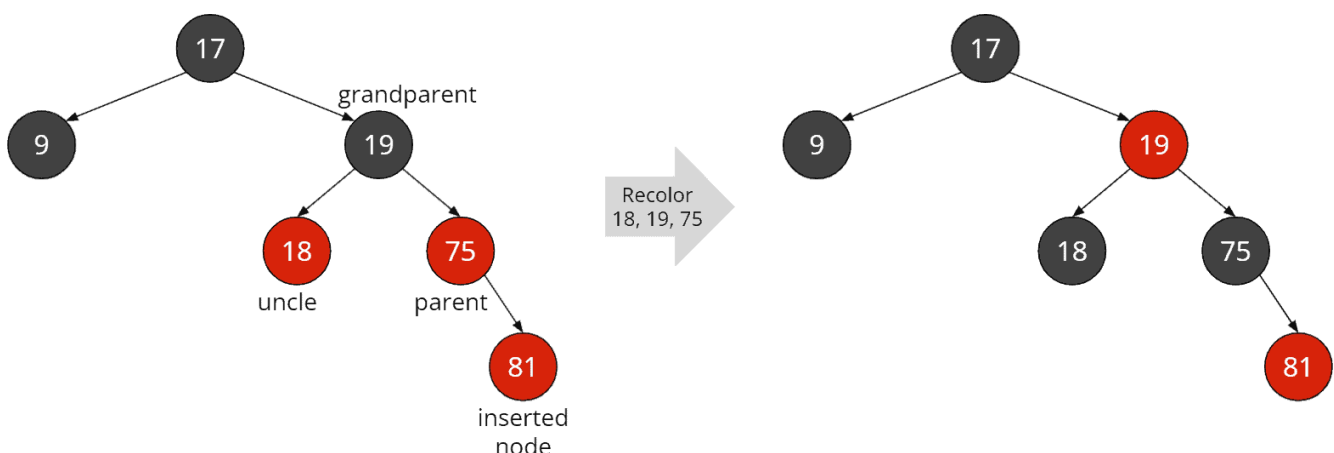
If we work with rule 2 ("the root is always black"), we have already colored the root black in case 1, and case 2 can no longer occur.

Case 3: Parent and Uncle Nodes Are Red

We use the term "uncle node" to refer to the sibling of the parent node; that is, the second child of the grandparent node next to the parent node. The following graphic should make this understandable: Inserted was the 81; its parent is the 75, the grandparent is the 19, and the uncle is the 18.

Both the parent and the uncle are red. In this case, we do the following:

We recolor parent and uncle nodes (18 and 75 in the example) black and the grandparent (19) red. Thus rule 4 ("no red-red!") is satisfied again at the inserted node. The number of black nodes per path does not change (in the example, it remains at 2).



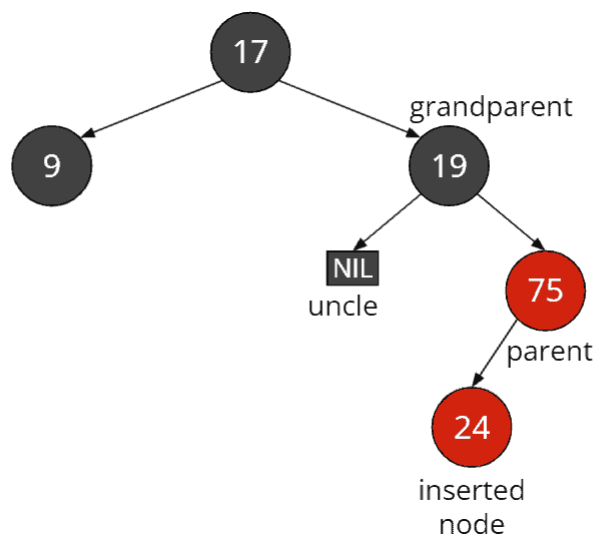
Recoloring parent, grandparent, and uncle

However, there could now be two red nodes in a row at the grandparent node – namely, if the great-grandparent node (17 in the example) were also red. In this case, we would have to make further repairs. We would do this by calling the repair function recursively on the grandparent node.

Case 4: Parent Node Is Red, Uncle Node Is Black, Inserted Node Is "Inner Grandchild"

I must first explain this case: "inner grandchild" means that the path from the grandparent node to the inserted node forms a triangle, as shown in the following graphic using 19, 75, and 24. In this example, you can see that a NIL leaf is also considered a black uncle (according to rule 3).

(For the sake of clarity, I have not drawn the two NIL leaves of the 9 and the 24, as well as the right NIL leaf of the 75.)



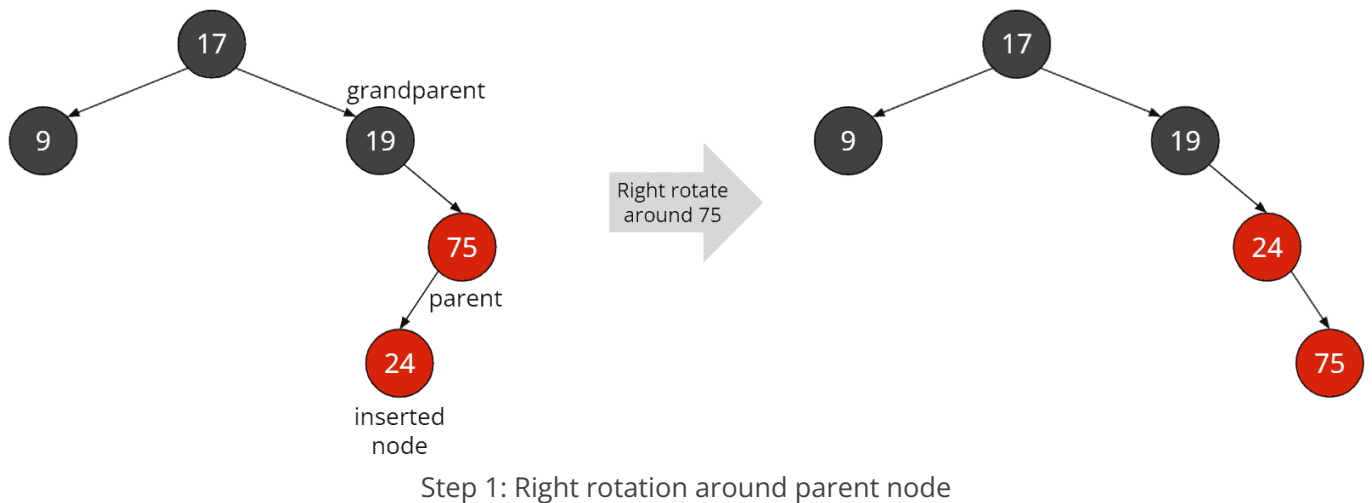
Case 4: Black uncle, inserted node is "inner" grandchild

In this case, we first rotate at the parent node in the *opposite direction of the inserted node*.

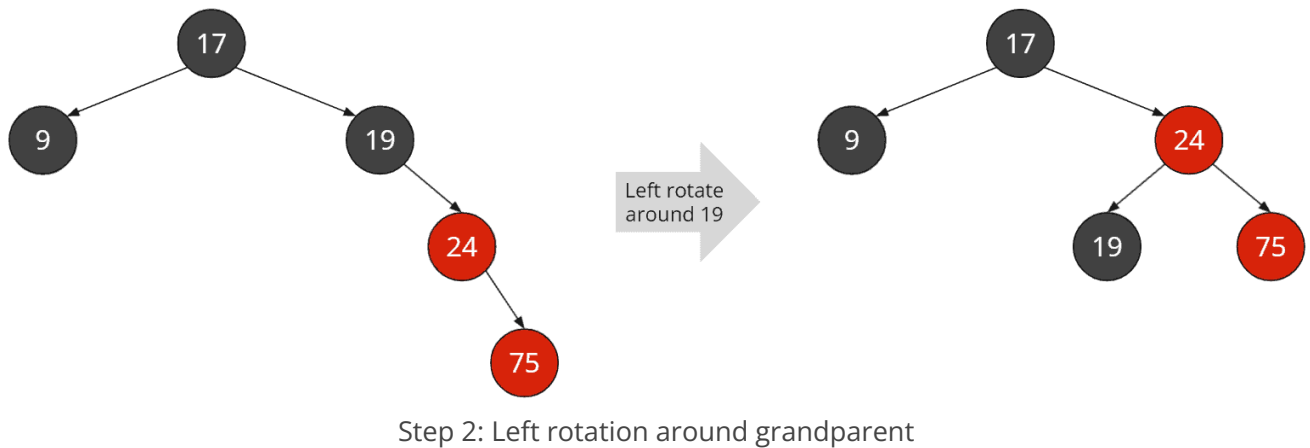
What does that mean?

If the inserted node is the *left* child of its parent node, we rotate to the *right* at the parent node. If the inserted node is the *right* child, we rotate to the *left*.

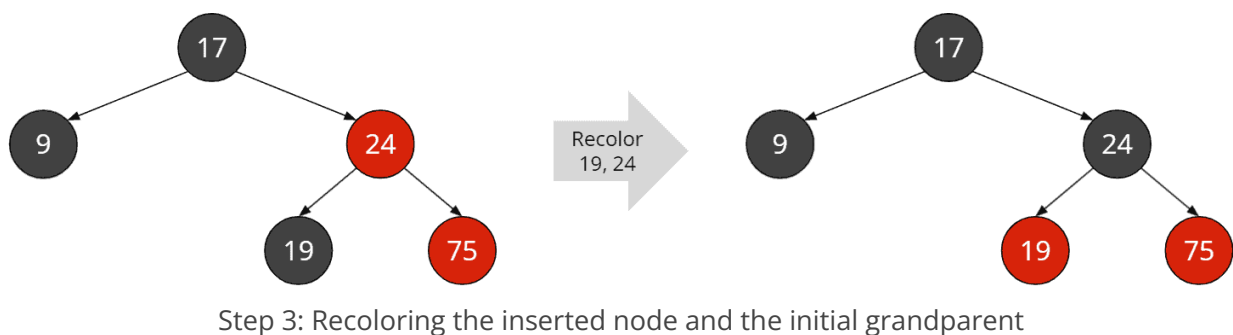
In the example, the inserted node (the 24) is a left child, so we rotate to the right at the parent node (75 in the example):



Second, we rotate at the grandparent node in the *opposite direction* to the previous rotation. In the example, we rotate left around the 19:



Finally, we color the node we just inserted (the 24 in the example) black and the original grandparent (the 19 in the example) red:



Since there is now a black node at the top of the last rotated subtree, there cannot be a violation of rule 4 ("no red-red!") at that position.

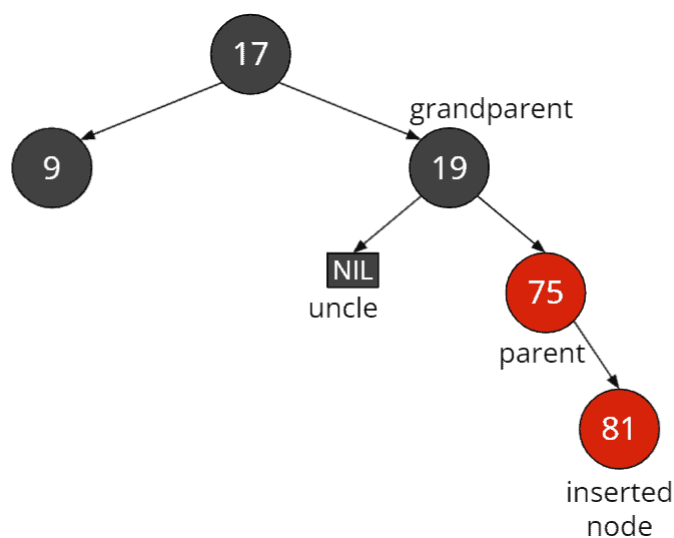
Also, recoloring the original grandparent (19) red cannot violate rule 4. Its left child is the uncle, which is black by definition of this case. And the right child, as a result of the second rotation, is the left child of the inserted node, thus a black NIL leaf.

The inserted red 75 has two NIL leaves as children, so there is no violation of rule 4 here either.

The repair is now complete; a recursive call of the repair function is not necessary.

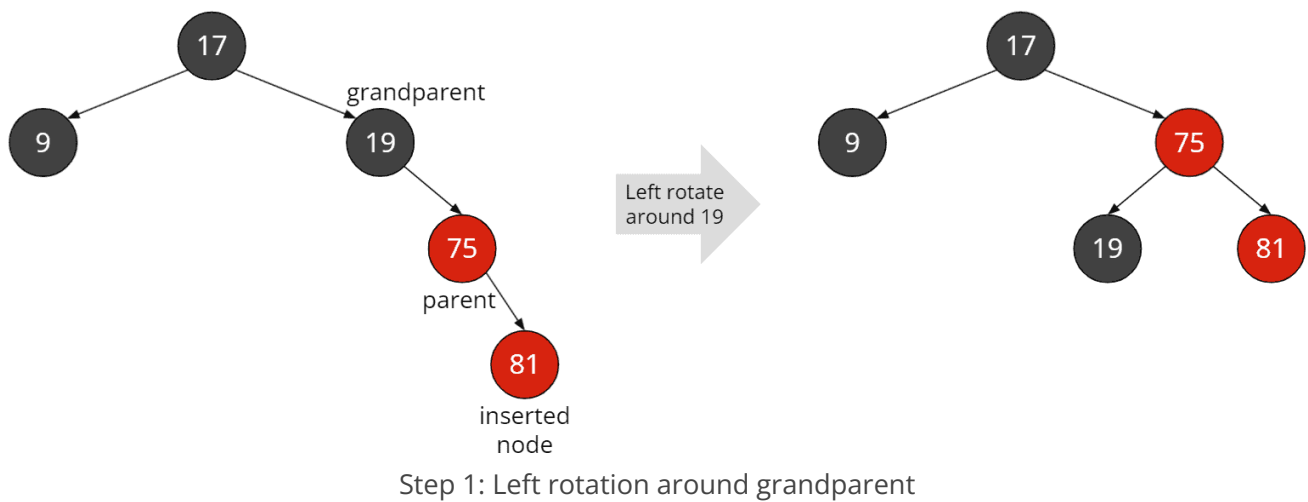
Case 5: Parent Node Is Red, Uncle Node Is Black, Inserted Node Is "Outer Grandchild"

"Outer grandchild" means that the path from grandparent to inserted node forms a line, such as the 19, 75, and 81 in the following example:

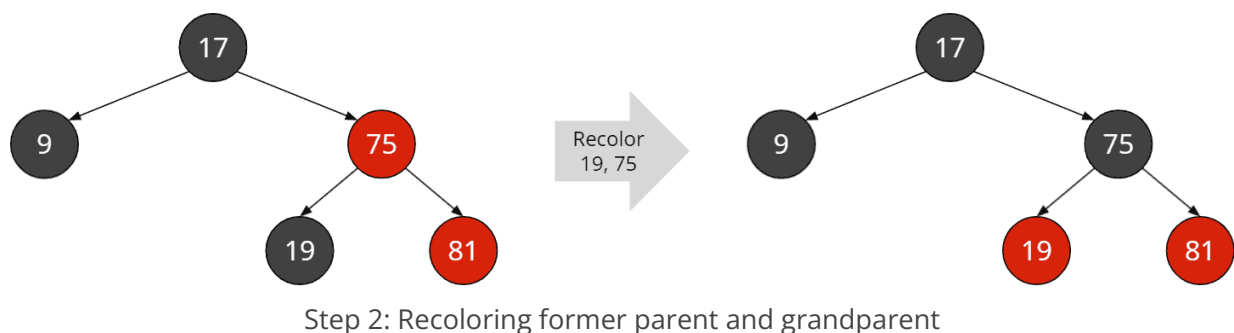


Case 5: Black uncle, inserted node is "outer" grandson

In this case, we rotate at the grandparent (19 in the example) in the *opposite direction of the parent and inserted node* (after all, both go in the same direction in this case). In the example, the parent and inserted nodes are both *right* children, so we rotate *left* at the grandparent:



Then we recolor the former parent (75 in the example) black and the former grandparent (19) red:



As at the end of case 4, we have a black node at the top of the rotation, so there can be no violation of rule 4 ("no red-red!") there.

The left child of the 19 is the original uncle after rotation, so it is black by case definition. The right child of the 19 is the original left child of the parent node (75), which must also be a black NIL leaf; otherwise, the right place where we inserted the 81 would not have been free (because a red node always has either two black children with value or two black NIL children).

The red 81 is the inserted node and, therefore, also has two black NIL leaves.

At this point, we've completed the repair of the red-black tree.

If you have paid close attention, you will notice that case 5 corresponds precisely to the second rotation from case 4. In the code, this will be shown by the fact that only

the first rotation is implemented for case 4, and then the program jumps to the code for case 5.

Implementation of the Post-Insertion Repair Method

You can find the complete repair function in [RedBlackTree starting at line 64](#). I have marked cases 1 to 5 by comments. Cases 4 and 5 are split into 4a/4b and 5a/5b depending on whether the parent node is left (4a/5a) or right child (4b/5b) of the grandparent node.

```
private void fixRedBlackPropertiesAfterInsert(Node node) {
    Node parent = node.parent;

    // Case 1: Parent is null, we've reached the root, the end of the r
    if (parent == null) {
        // Uncomment the following line if you want to enforce black root
        // node.color = BLACK;
        return;
    }

    // Parent is black --> nothing to do
    if (parent.color == BLACK) {
        return;
    }

    // From here on, parent is red
    Node grandparent = parent.parent;

    // Case 2:
    // Not having a grandparent means that parent is the root. If we en
    // (rule 2), grandparent will never be null, and the following if-t
    // removed.
    if (grandparent == null) {
        // As this method is only called on red nodes (either on newly in
        // recursively on red grandparents), all we have to do is to reco
        parent.color = BLACK;
        return;
    }

    // Get the uncle (may be null/nil, in which case its color is BLACK
    Node uncle = getUncle(parent);
```

```
// Case 3: Uncle is red -> recolor parent, grandparent and uncle
if (uncle != null && uncle.color == RED) {
    parent.color = BLACK;
    grandparent.color = RED;
    uncle.color = BLACK;

    // Call recursively for grandparent, which is now red.
    // It might be root or have a red parent, in which case we need to
    fixRedBlackPropertiesAfterInsert(grandparent);
}

// Parent is left child of grandparent
else if (parent == grandparent.left) {
    // Case 4a: Uncle is black and node is left->right "inner child"
    if (node == parent.right) {
        rotateLeft(parent);

        // Let "parent" point to the new root node of the rotated subtree
        // It will be recolored in the next step, which we're going to
        parent = node;
    }

    // Case 5a: Uncle is black and node is left->left "outer child" of
    rotateRight(grandparent);

    // Recolor original parent and grandparent
    parent.color = BLACK;
    grandparent.color = RED;
}

// Parent is right child of grandparent
else {
    // Case 4b: Uncle is black and node is right->left "inner child"
    if (node == parent.left) {
        rotateRight(parent);

        // Let "parent" point to the new root node of the rotated subtree
        // It will be recolored in the next step, which we're going to
        parent = node;
    }

    // Case 5b: Uncle is black and node is right->right "outer child"
```

```

        rotateLeft(grandparent);

        // Recolor original parent and grandparent
        parent.color = BLACK;
        grandparent.color = RED;
    }
}

```

You will find the helper function `getUncle()` [starting at line 152](#):

```

private Node getUncle(Node parent) {
    Node grandparent = parent.parent;
    if (grandparent.left == parent) {
        return grandparent.right;
    } else if (grandparent.right == parent) {
        return grandparent.left;
    } else {
        throw new IllegalStateException("Parent is not a child of its gra
    }
}

```

Implementation Notes

Unlike the AVL tree, we cannot easily hook the repair function of the red-black tree into the existing recursion from [BinarySearchTreeRecursive](#). That is because we need to rotate not only at the node under which we inserted the new node but also at the grandparent if necessary (cases 3 and 4).

You will find numerous alternative implementations in the literature. These are sometimes minimally more performant than the way presented here since they combine multiple steps. That doesn't change the order of magnitude of the performance, but it can gain a few percent. It was important for me to implement the algorithm in a comprehensible way. The more performant algorithms are always more complex, too.

I implemented the iterative insertion in two steps – search first, then insertion – unlike [BinarySearchTreeIterative](#), where I combined the two. That makes

reading the code a bit easier but requires an additional `"if (key < parent.data)"` check to determine whether the new node needs to be inserted as a left or right child under its parent.

Red-Black Tree Deletion

If you have just finished reading the chapter on inserting, you might want to take a short break. After all, deleting is even more complex.

First, we proceed as described in the ["Binary Search Tree Deletion" section](#) of the article on binary search trees in general.

Here is a summary:

1. If the node to be deleted has *no* children, we simply remove it.
2. If the node to be deleted has *one* child, we remove the node and let its single child move up to its position.
3. If the node to be deleted has *two* children, we copy the content (not the color!) of the in-order successor of the right child into the node to be deleted and then delete the in-order successor according to rule 1 or 2 (the in-order successor has at most one child by definition).

After that, we need to check the rules of the tree and repair it if necessary. To do this, we need to remember the deleted node's color and which node we have moved up.

- If the deleted node is red, we cannot have violated any rule: Neither can it result in two consecutive red nodes (rule 4), nor does it change the number of black nodes on any path (rule 5).
- However, if the deleted node is black, we are guaranteed to have violated rule 5 (unless the tree contained nothing but a black root), and rule 4 may also have been violated – namely if both parent nodes and the moved-up child of the deleted node were red.

First, here is the code for the actual deletion of a node ([class RedBlackTree, line 163](#)). Underneath the code, I will explain its parts:

```
public void deleteNode(int key) {
    Node node = root;

    // Find the node to be deleted
    while (node != null && node.data != key) {
        // Traverse the tree to the left or right depending on the key
        if (key < node.data) {
            node = node.left;
        } else {
            node = node.right;
        }
    }

    // Node not found?
    if (node == null) {
        return;
    }

    // At this point, "node" is the node to be deleted

    // In this variable, we'll store the node at which we're going to s
    // properties after deleting a node.
    Node movedUpNode;
    boolean deletedNodeColor;

    // Node has zero or one child
    if (node.left == null || node.right == null) {
        movedUpNode = deleteNodeWithZeroOrOneChild(node);
        deletedNodeColor = node.color;
    }

    // Node has two children
    else {
        // Find minimum node of right subtree ("inorder successor" of cur
        Node inOrderSuccessor = findMinimum(node.right);

        // Copy inorder successor's data to current node (keep its color!
        node.data = inOrderSuccessor.data;
```

```

        // Delete inorder successor just as we would delete a node with 0
        movedUpNode = deleteNodeWithZeroOrOneChild(inOrderSuccessor);
        deletedNodeColor = inOrderSuccessor.color;
    }

    if (deletedNodeColor == BLACK) {
        fixRedBlackPropertiesAfterDelete(movedUpNode);

        // Remove the temporary NIL node
        if (movedUpNode.getClass() == NilNode.class) {
            replaceParentsChild(movedUpNode.parent, movedUpNode, null);
        }
    }
}

```

The first lines of code search for the node to be deleted; the method terminates if that node can't be found.

How to proceed depends on the number of children nodes to be deleted.

Deleting a Node With Zero or One Child

If the deleted node has at most one child, we call the method `deleteNodeWithZeroOrOneChild()`. You can find it in the [source code starting at line 221](#):

```

private Node deleteNodeWithZeroOrOneChild(Node node) {
    // Node has ONLY a left child --> replace by its left child
    if (node.left != null) {
        replaceParentsChild(node.parent, node, node.left);
        return node.left; // moved-up node
    }

    // Node has ONLY a right child --> replace by its right child
    else if (node.right != null) {
        replaceParentsChild(node.parent, node, node.right);
        return node.right; // moved-up node
    }

    // Node has no children -->
}

```

```

// * node is red --> just remove it
// * node is black --> replace it by a temporary NIL node (needed t
else {
    Node newChild = node.color == BLACK ? new NilNode() : null;
    replaceParentsChild(node.parent, node, newChild);
    return newChild;
}
}

```

I have already introduced you to the `replaceParentsChild()` method (which is called several times here) in the rotation.

The case where the deleted node is black and has no children is a special case. That is dealt with in the last `else` block:

We have seen above that deleting a black node results in the number of black nodes no longer being the same on all paths. That is, we will have to repair the tree. The tree repair always starts (as you will see shortly) at the moved-up node.

If the deleted node has no children, one of its NIL leaves virtually moves up to its position. To be able to navigate from this NIL leaf to its parent node later, we need a special placeholder. I've implemented one in the class `NilNode`, which you can find in the [source code starting at line 349](#):

```

private static class NilNode extends Node {
    private NilNode() {
        super(0);
        this.color = BLACK;
    }
}

```

Finally, the `deleteNodeWithZeroOrOneChild()` method returns the moved-up node that the calling `deleteNode()` method stores in the `movedUpNode` variable.

Deleting a Node With Two Children

If the node to be deleted has two children, we first use the `findMinimum()` method (line 244) to find the in-order successor of the subtree that starts at the right child:

```
private Node findMinimum(Node node) {
    while (node.left != null) {
        node = node.left;
    }
    return node;
}
```

We then copy the data of the in-order successor into the node to be deleted and call the `deleteNodeWithZeroOrOneChild()` method introduced above to remove the in-order successor from the tree. Again, we remember the moved-up node in `movedUpNode`.

Repairing the Tree

Here is once more the last `if`-block of the `deleteNode()` method:

```
if (deletedNodeColor == BLACK) {
    fixRedBlackPropertiesAfterDelete(movedUpNode);

    // Remove the temporary NIL node
    if (movedUpNode.getClass() == NilNode.class) {
        replaceParentsChild(movedUpNode.parent, movedUpNode, null);
    }
}
```

As stated above, deleting a *red* node does not violate any rules. If, however, the deleted node is *black*, we call the repair method `fixRedBlackPropertiesAfterDelete()`.

If any, we've needed the temporary `NilNode` placeholder created in `deleteNodeWithZeroOrOneChild()` only for calling the repair function. We can therefore remove it afterward.

When deleting, we have to consider one more case than when inserting. In contrast to the insertion, the color of the uncle is not relevant here but that of the deleted node's sibling.

- Case 1: Deleted node is the root
- Case 2: Sibling is red
- Case 3: Sibling is black and has two black children, parent is red
- Case 4: Sibling is black and has two black children, parent is black
- Case 5: Sibling is black and has at least one red child, "outer nephew" is black
- Case 6: Sibling is black and has at least one red child, "outer nephew" is red

The following sections describe the six cases in detail:

Case 1: Deleted Node Is the Root

If we removed the root, another node moved up to its position. That could only happen if the root had zero or only one child. If the root had had two children, it would have been the in-order successor that would have been removed in the end and not the root node.

If the root had *no* child, the new root is a black NIL node. Thus the tree is empty and valid:



Case 1a: Removing a root without a child

If the root had *one* child, then this had to be red and have no other children.

Explanation: If the red child had another red child, rule 4 ("no red-red!") would have been violated. If the red child had a black child, then the paths through the red node would have at least one more black node than the NIL subtree of the root, and thus rule 5 would have been violated.

Thus, the tree consists of only one red root and is therefore also valid.

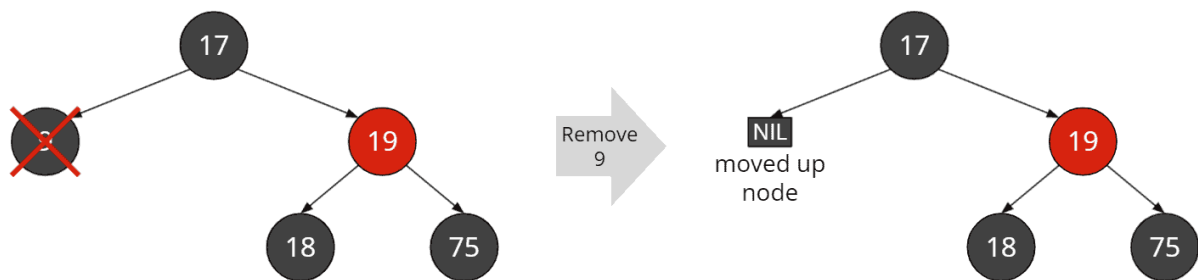


Case 1b: Removing a root with one child

Should we work with rule 2 ("the root is always black"), we would now recolor the root.

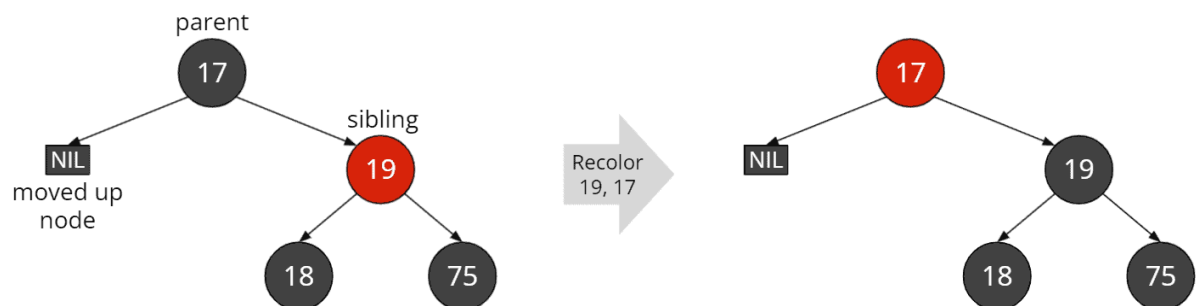
Case 2: Sibling Is Red

For all other cases, we first check the color of the sibling. That is the second child of the parent of the deleted node. In the following example, we delete the 9; its sibling is the red 19:



Case 2: Red sibling

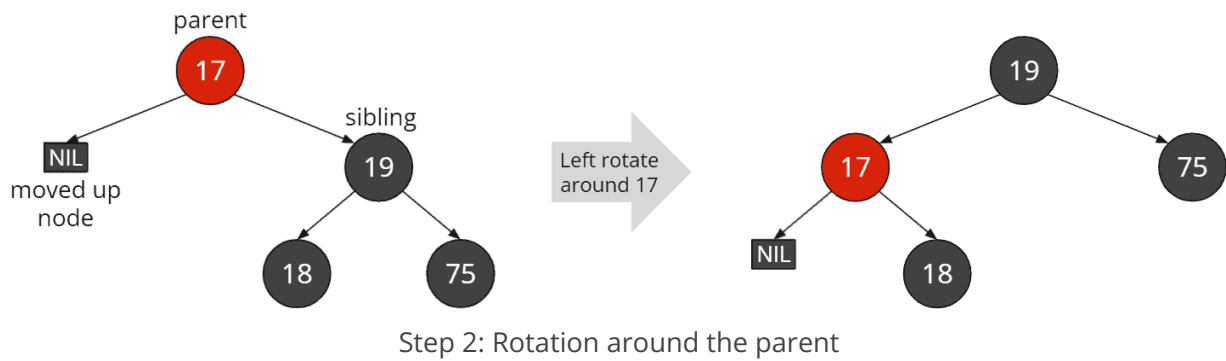
In this case, we first color the sibling black and the parent red:



Step 1: Recoloring sibling and parent

That obviously violated rule 5: The paths in the right subtree of the parent each have two more black nodes than those in the left subtree. We fix this by rotating around the parent in the direction of the deleted node.

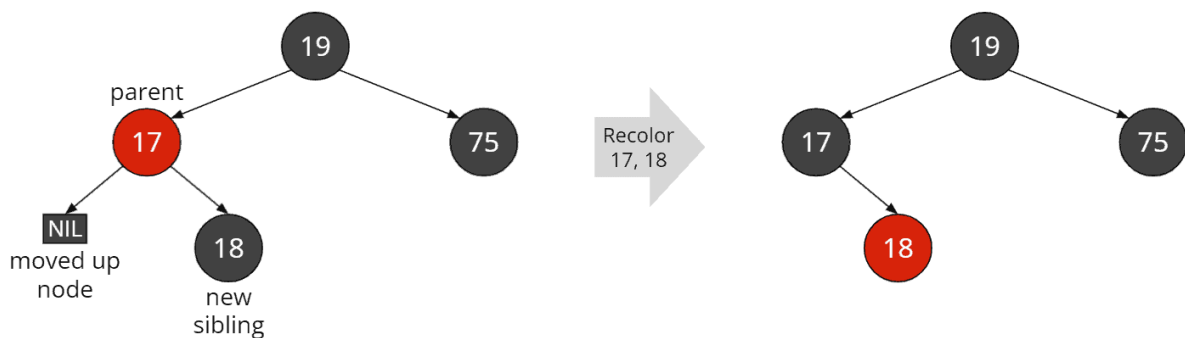
In the example, we have deleted the *left* node of the parent node – we, therefore, perform a *left* rotation:



Now we have two black nodes on the right path and two on the path to the 18. However, we have *only one* black node on the path to the left NIL leaf of 17 (remember: the root does not count, the NIL nodes do – even the ones not drawn in the graphic).

We look at the new sibling of the deleted node (18 in the example). That new sibling is now definitely black because it is an original child of the red sibling from the beginning of the case.

Also, the new sibling has black children. Therefore, we color the sibling (the 18) red and the parent (the 17) black:



(Step 3: Recoloring parent and new sibling)

Now all paths have two black nodes; we have a valid red-black tree again.

Case 2 - Fall-Through

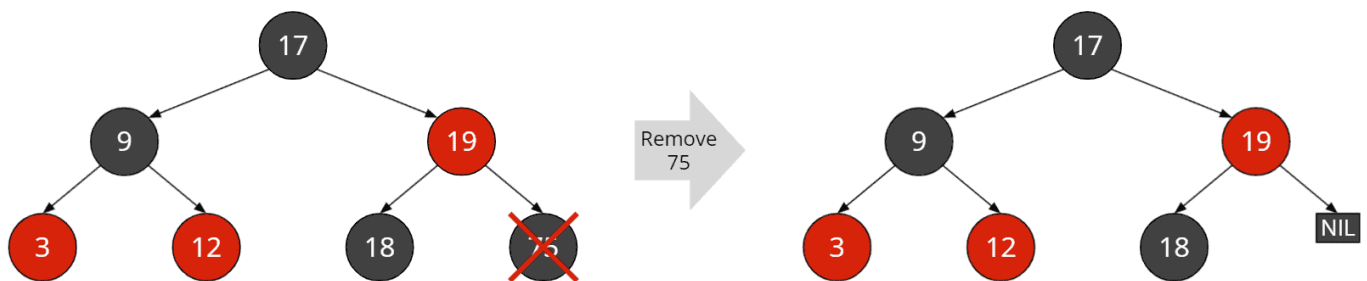
In fact, I have anticipated something in this last step. Namely, we have executed the rules of case 3 (that's why the image subtitle is in parentheses).

In this last step of case 2, we always have a black sibling. The fact that the black sibling had two black children, as required for case 3, was a coincidence. In fact, at the end of case 2, any of the cases 3 to 6 can occur and must be treated according to the following sections.

Case 3: Sibling Is Black and Has Two Black Children, Parent Is Red

In the following example, we delete the 75 and let one of its black NIL leaves move up.

(Again, as a reminder: I only show NIL nodes in the graphics when they are relevant for understanding.)

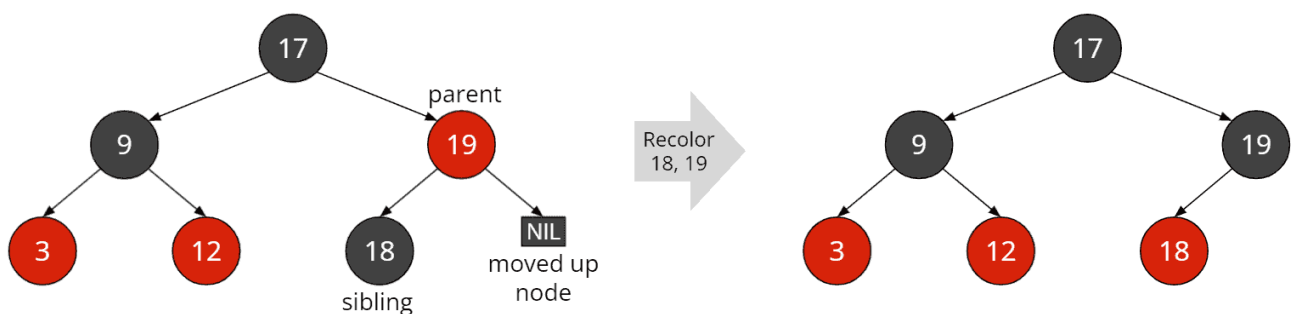


Case 3: Black sibling with black children and red parent

The deletion violates rule 5: In the rightmost path, we now have one black node less than in all others.

The sibling (the 18 in the example) is black and has two black children (the NIL leaves not shown). The parent (the 19) is red. In this case, we repair the tree as follows:

We recolor the sibling (the 18) red and the parent (the 19) black:

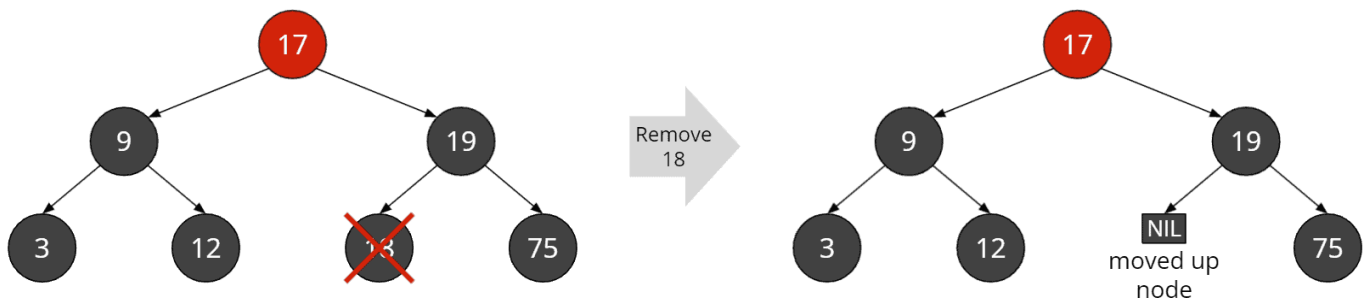


Recoloring parent and sibling

Thus we have a valid red-black tree again. The number of black nodes is the same on all paths (as required by rule 5). And since the sibling has only black children, coloring it red cannot violate rule 4 ("no red-red!").

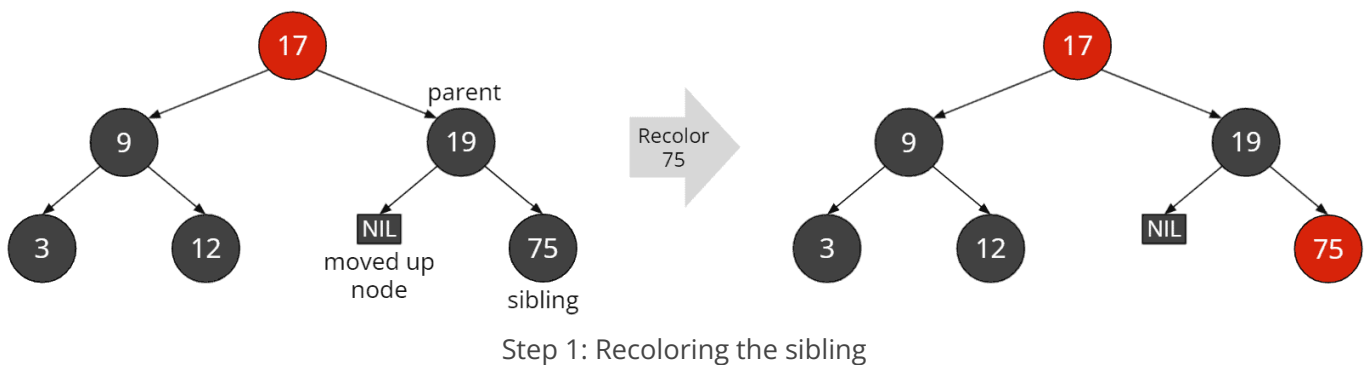
Case 4: Sibling Is Black and Has Two Black Children, Parent Is Black

In the following example, we delete the 18:



This leads (just like in case 3) to a violation of rule 5: On the path to the deleted node, we now have one black node less than on all other paths.

In contrast to case 3, in this case, the parent node of the deleted node is black. We first color the sibling red:



That means that the black height in the subtree that starts at the parent node is again uniform (2). In the left subtree, however, it is one higher (3). Rule 5 is therefore still violated.

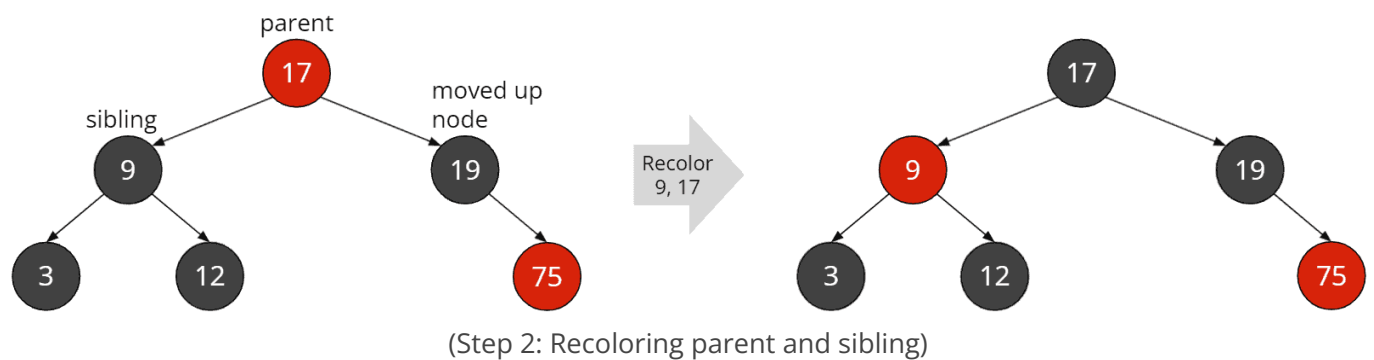
Case 4 - Recursion

We solve this problem by pretending that we deleted a black node between nodes 17 and 19 (which would have had the same effect). Accordingly, we call the repair

function recursively on the parent node, i.e., the 19 (which would have been the moved-up node in this case).

The 19 has a black sibling (the 9) with two black children (3 and 12) and a red parent (17). Accordingly, we are now back to case 3.

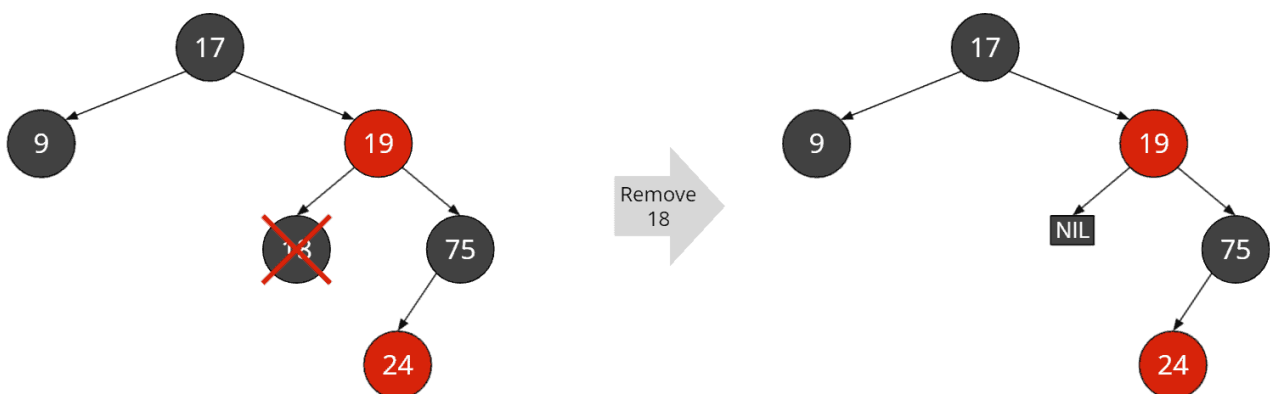
We solve case 3 by coloring the parent black and the sibling red:



The black height is now two on all paths, so our red-black tree is valid again.

Case 5: Sibling is black and has at least one red child, "outer nephew" is black

In this example, we delete the 18:



Case 5: Black sibling with at least one red child and a black "outer nephew"

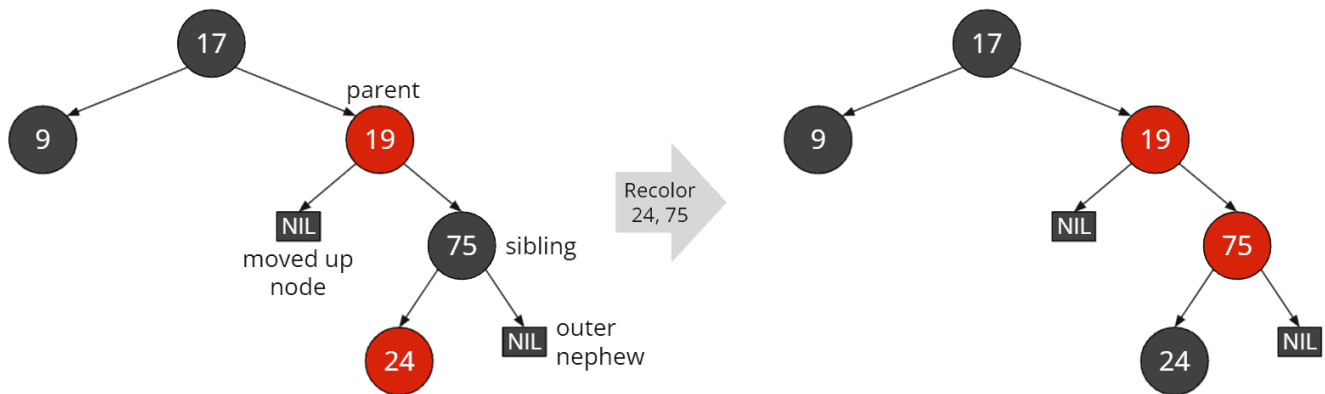
As a result, we again violated rule 5 since the subtree starting at the sibling now has a black height greater by one.

We examine the "outer nephew" of the deleted node. "Outer nephew" means the child of the sibling that is opposite the deleted node. In the example, this is the right

(and by definition black) NIL leaf under the 75.

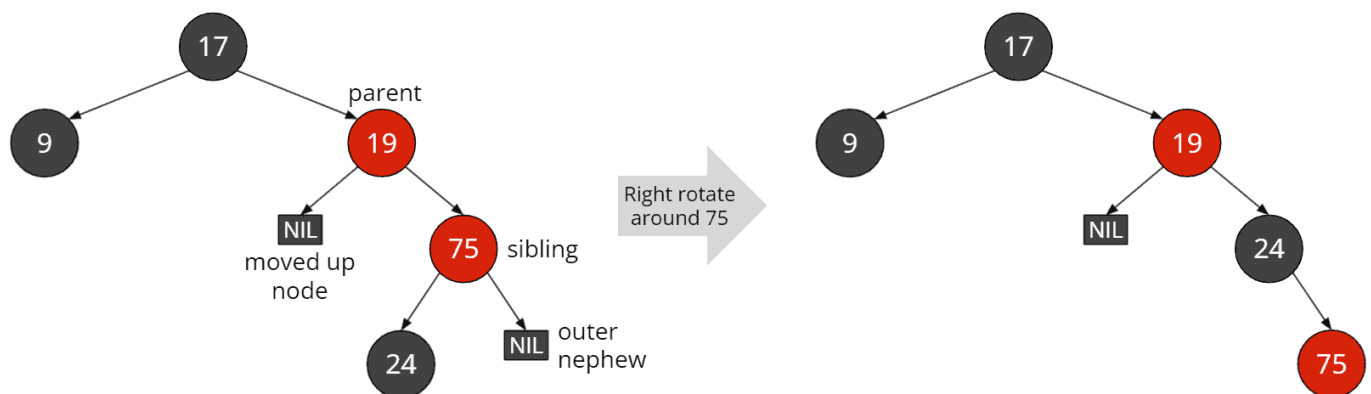
In the following graphic, you can see that parent, sibling and nephew together form a line (in the example: 19, 75, and its right NIL child).

We start the repair by coloring the inner nephew (the 24 in the example) black and the sibling (the 75) red:



Step 1: Recoloring sibling and inner nephew

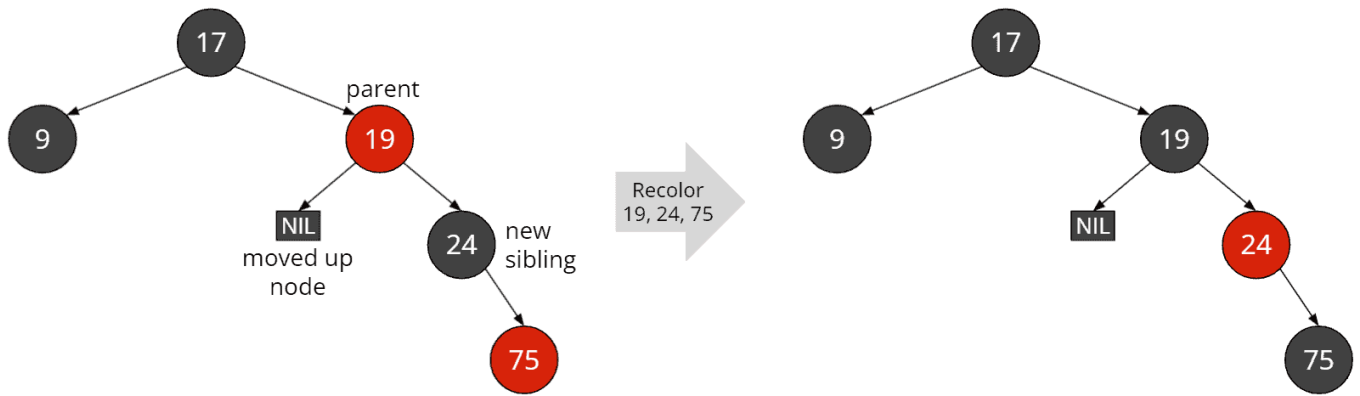
Then we perform a rotation at the sibling node *in the opposite direction of the deleted node*. In the example, we've deleted the parent's *left* child, so we perform a *right* rotation at the sibling (the 75):



Step 2: Rotation around sibling

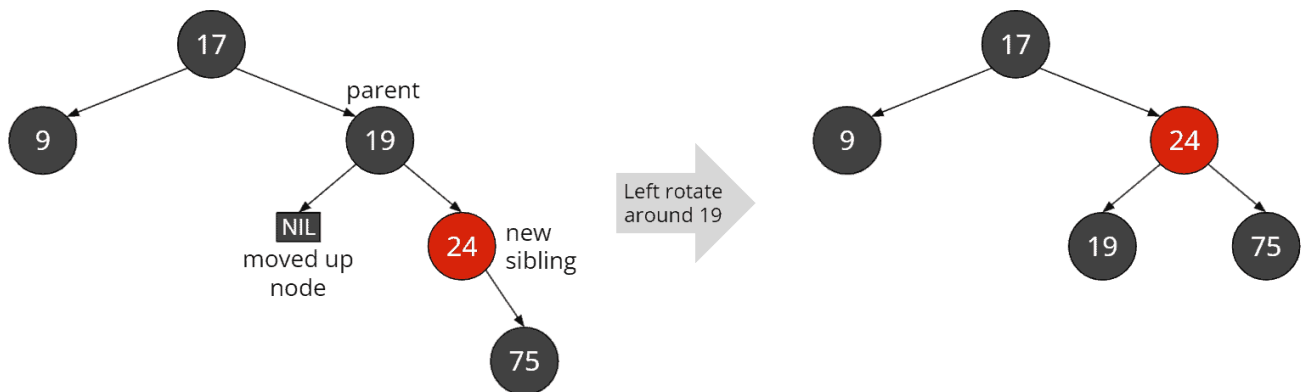
We are doing some recoloring again:

- We recolor the sibling in the color of its parent (in the example, the 24 red).
- Then we recolor the parent (the 19) and the outer nephew of the deleted node, i.e., the right child of the new sibling (the 75 in the example) black:



Step 3: Recoloring parent, sibling, and nephew

Finally, we perform a rotation on the parent node *in the direction of the deleted node*. In the example, the deleted node was a *left* child, so we perform a *left* rotation accordingly (at 19 in the example):

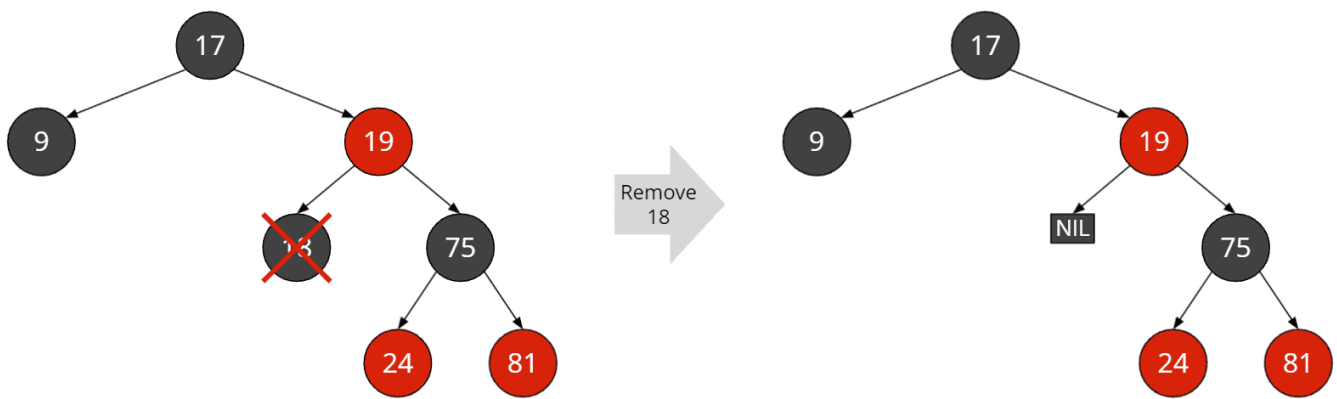


Step 4: Rotation around the parent

This last step restores compliance with all red-black rules. There are no two consecutive red nodes, and the number of black nodes is uniformly two on all paths. We've thus completed the repair of the tree.

Case 6: Sibling is black and has at least one red child, "outer nephew" is red

In the last example, which is very similar to case 5, we also delete the 18:

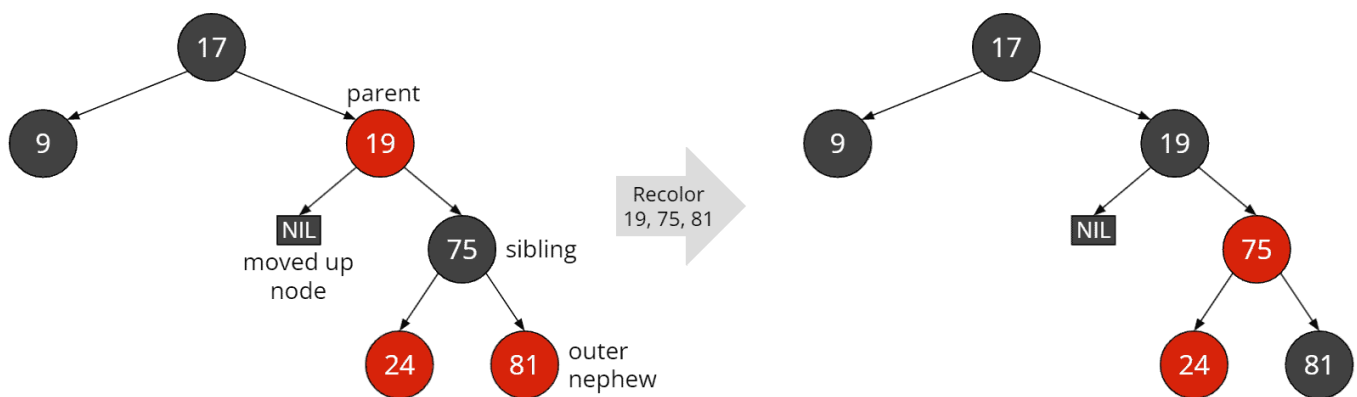


Case 6: Black sibling with at least one red child and a red "outer nephew"

As a result, as in case 5, we violated rule 5 because the path to the deleted node now contains one less black node.

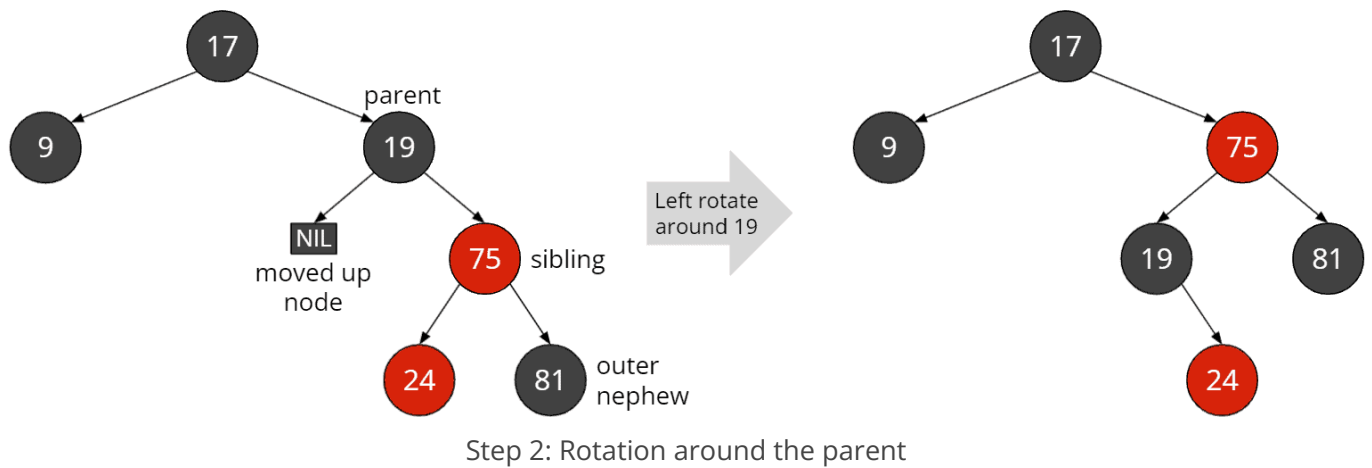
In case 6, unlike case 5, the outer nephew (81 in the example) is red and not black.

We first recolor the sibling in the parent's color (in the example, the 75 red). Then we recolor the parent (the 19 in the example) and the outer nephew (the 81) black:



Step 1: Recoloring parent, sibling, and nephew

Second, we perform a rotation at the parent node *in the direction of the deleted node*. In the example, we've deleted a *left* child; accordingly, we perform a *left* rotation around the 19:



This rotation restores the red-black rules. No two red nodes follow each other, and the number of black nodes is the same on all paths (namely 2).

The rules in this last case are similar to the final two steps of case 5. In the source code, you will see that for case 5, only its first two steps are implemented, and the program then goes to case 6 to execute the last two steps.

With this, we have studied all six cases. Let's move on to the implementation of the repair function in Java.

Implementation of the Post-Deletion Repair Method

You can find the `fixRedBlackPropertiesAfterDelete()` method [in the source code starting at line 252](#). I have marked cases 1 to 6 with comments.

```
private void fixRedBlackPropertiesAfterDelete(Node node) {
    // Case 1: Examined node is root, end of recursion
    if (node == root) {
        // Uncomment the following line if you want to enforce black root
        // node.color = BLACK;
        return;
    }

    Node sibling = getSibling(node);

    // Case 2: Red sibling
    if (sibling.color == RED) {
        handleRedSibling(node, sibling);
        sibling = getSibling(node); // Get new sibling for fall-through t
    }
}
```

```

    }

    // Cases 3+4: Black sibling with two black children
    if (isBlack(sibling.left) && isBlack(sibling.right)) {
        sibling.color = RED;

        // Case 3: Black sibling with two black children + red parent
        if (node.parent.color == RED) {
            node.parent.color = BLACK;
        }

        // Case 4: Black sibling with two black children + black parent
        else {
            fixRedBlackPropertiesAfterDelete(node.parent);
        }
    }

    // Case 5+6: Black sibling with at least one red child
    else {
        handleBlackSiblingWithAtLeastOneRedChild(node, sibling);
    }
}

```

You will find the helper methods `getSibling()` and `isBlack()` [starting at line 334](#):

```

private Node getSibling(Node node) {
    Node parent = node.parent;
    if (node == parent.left) {
        return parent.right;
    } else if (node == parent.right) {
        return parent.left;
    } else {
        throw new IllegalStateException("Parent is not a child of its gra
    }
}

private boolean isBlack(Node node) {
    return node == null || node.color == BLACK;
}

```

Handling a red sibling (case 2) [starts at line 289](#):

```

private void handleRedSibling(Node node, Node sibling) {
    // Recolor...
    sibling.color = BLACK;
    node.parent.color = RED;

    // ... and rotate
    if (node == node.parent.left) {
        rotateLeft(node.parent);
    } else {
        rotateRight(node.parent);
    }
}

```

You can find the implementation for a black sibling knot with at least one red child (cases 5 and 6) [starting at line 302](#):

```

private void handleBlackSiblingWithAtLeastOneRedChild(Node node, Node
    boolean nodeIsLeftChild = node == node.parent.left;

    // Case 5: Black sibling with at least one red child + "outer nephe
    // --> Recolor sibling and its child, and rotate around sibling
    if (nodeIsLeftChild && isBlack(sibling.right)) {
        sibling.left.color = BLACK;
        sibling.color = RED;
        rotateRight(sibling);
        sibling = node.parent.right;
    } else if (!nodeIsLeftChild && isBlack(sibling.left)) {
        sibling.right.color = BLACK;
        sibling.color = RED;
        rotateLeft(sibling);
        sibling = node.parent.left;
    }

    // Fall-through to case 6...

    // Case 6: Black sibling with at least one red child + "outer nephe
    // --> Recolor sibling + parent + sibling's child, and rotate around
    sibling.color = node.parent.color;
    node.parent.color = BLACK;
    if (nodeIsLeftChild) {
        sibling.right.color = BLACK;

```

```
        rotateLeft(node.parent);  
    } else {  
        sibling.left.color = BLACK;  
        rotateRight(node.parent);  
    }  
}
```

Just as for inserting, you will find numerous alternative approaches for deleting in the literature. I have tried to structure the code so that you can follow the code flow as well as possible.

Traversing the Red-Black Tree

Like any binary tree, we can traverse the red-black tree in pre-order, post-order, in-order, reverse-in-order, and level-order. In the ["Binary Tree Traversal"](#) section of the introductory article on binary trees, I have described traversal in detail.

In that section, you will also find the corresponding Java source code, implemented in the classes [DepthFirstTraversal](#), [DepthFirstTraversalIterative](#), and [DepthFirstTraversalRecursive](#).

The traversal methods work on the [BinaryTree](#) interface. Since [RedBlackTree](#) also implements this interface, we can easily apply the traversal methods to it as well.

Red-Black Tree Time Complexity

For an introduction to the topic of time complexity and O-notation, [see this article](#).

Free Bonus:

Big O Cheat Sheet

[7 Time Complexity Classes on 1 Page]

Use this 1-page PDF cheat sheet as a reference to quickly look up the seven most important time complexity classes (with descriptions and examples).

First name...

Email address...

Send Me the Cheat Sheet Now!

You get access to this PDF by signing up for my newsletter. I won't send any spam, and you can opt-out at any time.

We can determine the cost of searching, inserting, and deleting a node in the binary tree as follows:

Search Time

We follow a path from the root to the searched node (or to a NIL leaf). At each level, we perform a comparison. The effort for the comparison is constant.

The search cost is thus proportional to the tree height.

We denote by n the number of tree nodes. In the "[Height of a Red-Black Tree](#)" section, we have recognized that the longest path is at most twice as long as the shortest path. It follows that the height of the tree is bounded by $O(\log n)$.

A formal proof is beyond the scope of this article. You can read the proof on [Wikipedia](#).

Thus, the time complexity for finding a node in a red-black tree is: $O(\log n)$

Insertion Time

When inserting, we first perform a search. We have just determined the search cost as $O(\log n)$.

Next, we insert a node. The cost of this is constant regardless of the tree size, so $O(1)$.

Then we check the red-black rules and restore them if necessary. We do this starting at the inserted node and ascending to the root. At each level, we perform one or more of the following operations:

- Checking the color of the parent node
- Determination of the uncle node and checking its color
- Recoloring one up to three nodes
- Performing one or two rotations

Each of these operations has constant time, $O(1)$, in itself. The total time for checking and repairing the tree is therefore also proportional to its height.

So the time complexity for inserting into a red-black tree is also: $O(\log n)$

Deletion Time

Just as with insertion, we first search for the node to be deleted in time $O(\log n)$.

Also, the deletion cost is independent of the tree size, so it is constant $O(1)$.

For checking the rules and repairing the tree, one or more of the following operations occur – at most once per level:

- Checking the color of the deleted node
- Determining the sibling and examining its color
- Checking the colors of the sibling's children
- Recoloring the parent node
- Recoloring the sibling node and one of its children
- Performing one or two rotations

These operations also all have a constant complexity in themselves. Thus, the total effort for checking and restoring the rules after deleting a node is also proportional to the tree height.

So the time complexity for deleting from a red-black tree is also: $O(\log n)$

Red-Black Tree Compared With Other Data Structures

The following sections describe the differences and the advantages and disadvantages of the red-black tree compared to alternative data structures.

Red-Black Tree vs. AVL Tree

The red-black tree, as well as the [AVL tree](#), are self-balancing binary search trees.

In the red-black tree, the longest path to the root is at most twice as long as the shortest path to the root. On the other hand, in the AVL tree, the depth of no two subtrees differs by more than 1.

In the red-black tree, balance is maintained by the node colors, a set of rules, and by rotating and recoloring nodes. In the AVL tree, the heights of the subtrees are compared, and rotations are performed when necessary.

These differences in the characteristics of the two types of trees lead to the following differences in performance and memory requirements:

- Due to the more even balancing of the AVL tree, search in an AVL tree is usually faster. In terms of magnitude, however, both are in the range $O(\log n)$.
- For insertion and deletion, the time complexity in both trees is $O(\log n)$. In a direct comparison, however, the red-black tree is faster because it rebalances less frequently.
- Both trees require additional memory: the AVL tree one byte per node for the height of the subtree starting at a node; the red-black tree one bit per node for the color information. This rarely makes a difference in practice since a single bit usually occupies at least one byte.

If you expect many insert/delete operations, then you should use a red-black tree. If, on the other hand, you expect more search operations, then you should choose the AVL tree.

Red-Black Tree vs. Binary Search Tree

The red-black tree is a concrete implementation of a self-balancing binary search tree. So every red-black tree is also a binary search tree.

There are also other types of binary search trees, such as the AVL tree mentioned above – or trivial non-balanced implementations. Thus, not every binary search tree is also a red-black tree.

Summary

This tutorial taught you what a red-black tree is, which rules govern it and how these rules are evaluated and restored if necessary after inserting and deleting nodes. I also introduced you to a Java implementation that is as easy to understand as possible.

The JDK uses red-black trees in [TreeMap](#) (here is the [source code](#) on GitHub) and in bucket collisions in [HashMap](#) (here is the [source code](#)).

With this, I conclude the tutorial series on binary trees.

If I could help you better understand [binary trees in general](#), [binary search trees](#), [AVL trees](#), and – in this article – red-black trees, I'm happy about a comment. Also, feel free to share the article using one of the share buttons at the end.

Do you want to be informed when the next article is published on HappyCoders.eu? Then [click here](#) to sign up for the HappyCoders newsletter.



[Previous:](#)
AVL Tree



Free Bonus:

Big O Cheat Sheet

[7 Time Complexity Classes on 1 Page]

Use this 1-page PDF cheat sheet as a reference to quickly look up the seven most important time complexity classes (with descriptions and examples).

Send Me the Cheat Sheet Now!

You get access to this PDF by signing up for my newsletter. I won't send any spam, and you can opt-out at any time.

About the Author

I'm a freelance software developer with more than two decades of experience in scalable Java enterprise applications. My focus is on optimizing complex algorithms and on advanced topics such as concurrency, the Java memory model, and garbage collection. Here on HappyCoders.eu, I want to help you become a better Java programmer. [Read more about me here.](#)



Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

Name *

Email *

Post Comment

12 comments on “Red-Black Tree (Fully Explained, with Java Code)”

**Sangita Elango**

In line 102, you have stated "Call recursively for grandparent, which is now red" and in line 104, you have called "fixRedBlackPropertiesAfterInsert(grandparent);"

My question is - If Uncle is black we are doing some step to get a result that changes grandparent to red too. so why you are not calling the "fixRedBlackPropertiesAfterInsert(grandparent);" in line 130 and 149. please clarify.

REPLY

**Sven Woltmann**

Thanks for this good question. You're talking about cases 4 and 5.

The answer is: we color the *old* grandparent red and the *old* parent black. On rotation, the red grandparent becomes a child and the black parent becomes the new grandparent. So after the

rotation, the grandparent is black. This means there can't be two subsequent red nodes, so there's no need to continue fixing by calling `fixRedBlackPropertiesAfterInsert()` again.

It is confusing that the code first rotates and then recolors. It probably would have been more understandable the other way around.

REPLY



Sangita Elango

Now I get it. Thankyou so much!



Dorian

I have used your implementation, but when I delete some items from the tree and add them back it gives me an error that the node already exists. Only thing I have changed is the `node.data` to `node.key` for a string key.

REPLY



Sven Woltmann

Deleting and then adding the same values works with my implementation.

Maybe you broke something when changing `int` to `String`. Did you make sure you're comparing the Strings correctly? With `'equals()'` instad of `'=='`? And with `'compareTo()'` instead of `'<'` and `'>'`?

If you want, you can send me your code with a test that shows the failing insert, and I'll have a look.

REPLY



Dorian

It is fixed. Still can't find where I went wrong though.



Azhat

I was able to understand the concept of RB Tree, but when it comes to implementing in code, I was not able to understand even after going through so many sites and references.

Your explanation of implementing RB Tree in java is one of the best. It's very easy to understand and code. Appreciate your effort in writing such a detailed and nice article.

One minor correction, In Case 4: Sibling Is Black and Has Two Black Children, Parent Is Black, of deleting a node, in the second figure, sibling node should be 75 instead of 18

REPLY



Sven Woltmann

Hi Azhat,

thanks so much for your message; I'm happy that my explanations helped you understand the implementation.

And thanks again for pointing out the error in case 4. I've fixed it now.

Best wishes

Sven

REPLY



IanTsai

That is a clear explanation for the RB tree, Thank you for sharing.

REPLY



Nex

"The following example shows the shortest possible path through a red-black tree of height four on the left and the longest possible path on the right:"

Shouldn't be "height two on the left"?

REPLY



Sven Woltmann

No. But I admit that the sentence may be hard to read. Maybe better with these parentheses:

"The following example shows the shortest possible path (through a red-black tree of height four) on the left and the longest possible path on the right"

- The tree's height is four.
- The shortest possible path is on the left.
- The longest possible path is on the right.

REPLY



Ra

Thanks a lot. Could you do a very short article on PostgreSQL B-trees?

REPLY

You might also like the following articles



ADVENT **OF** CODE 2022 – OBJECT-ORIENTED SOLUTIONS **IN** **JAVA**

Sven Woltmann

December 1, 2022



RADIX SORT – ALGORITHM, SOURCE CODE, TIME COMPLEXITY

Sven Woltmann

July 19, 2022



STACK **VS.** QUEUE

Sven Woltmann

June 8, 2022



JAVA QUEUE **VS.** DEQUE

Sven Woltmann

June 7, 2022



Advanced Java topics, algorithms and data structures.

JOIN OUR FREE NEWSLETTER

Boost your skills: Become a better Java programmer.

CLICK HERE TO SUBSCRIBE!

Blog

Java

Algorithms and Data Structures

DevOps

Software Craftsmanship

Book Recommendations

Resources

Java Versions Cheat Sheet

Big O Cheat Sheet

Newsletter

Publications

Conference Talks

About

About Sven Woltmann

HappyCoders Manifesto

Follow us



[Legal Notice](#) [Privacy Policy](#)

Copyright © 2018–2024 Sven Woltmann



[8 Bewertungen auf ProvenExpert.com](#)