**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.

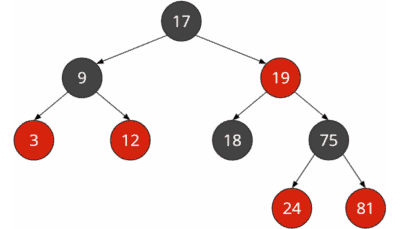
### NIL Nodes in Red-Black Trees

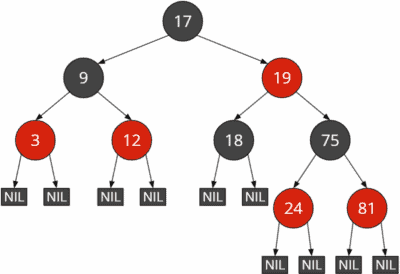
In the literature, red-black trees are depicted with and without so-called NIL nodes. A NIL node is a leaf that does not contain a value. NIL nodes become relevant for the algorithms later on, e.g., to determine colors of uncle or sibling nodes.

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.

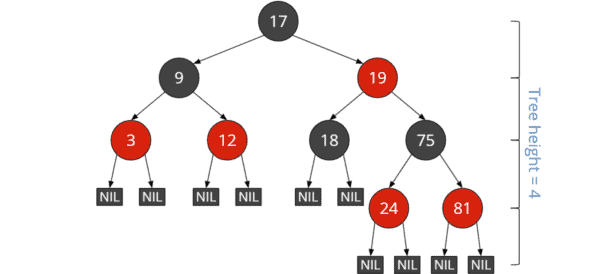
Therefore, rule 2 is often omitted in the literature.

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:

Height of red-black tree

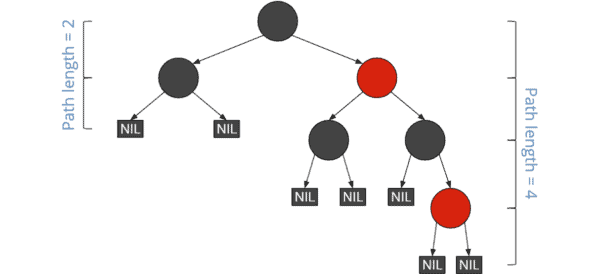
From rules 3 and 4 follows:

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](https://www.happycoders.eu/algorithms/binary-search-tree-java/#Binary_Search_Tree_in_Java) from the second part of the binary tree series.

Nodes are represented by the [Node](https://github.com/SvenWoltmann/binary-tree/blob/main/src/main/java/eu/happycoders/binarytree/Node.java) class. For simplicity, we use int primitives as the node value.