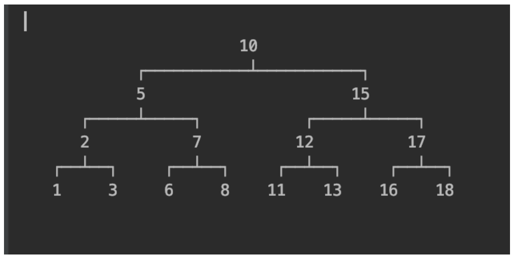
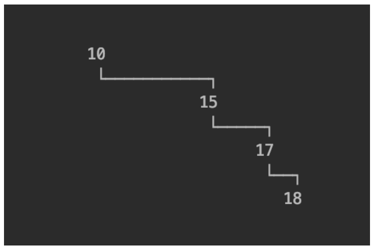
A9: Binary Search Trees, AVL Trees, and Red-Black Trees

1 Binary Search Trees

These are binary trees that satisfy the following *order property*: all the nodes to the left of a root are smaller than it, and all the nodes to the right of a root are larger than or equal to it. Here is an ideal case:



In this case, we can insert, delete, and search the tree for any item using a maximum of four comparisons, i.e., using $O(\log n)$ comparisons. But binary search trees make no guarantees about being balance. The following is also a binary search tree:



In the latter tree it might take $O(n)$ comparisons to insert, delete, or search for an element.

2 AVL Trees

AVL trees maintain balance with a strict property: throughout the tree, the height of any two siblings never differs by more than 1.

2.1 Insertion

In order to maintain the balanced condition, we reason as follows. Assume the tree is balanced before an insertion, i.e., no two siblings differ in height by more than 1. A single insertion might increase the height of one particular subtree. If we are unlucky, that tree was already 1-higher than its sibling and the new insertion causes to be 2-higher. For example, if we insert 15 in the balanced tree displayed on the left, we get the unbalanced tree displayed on the right:

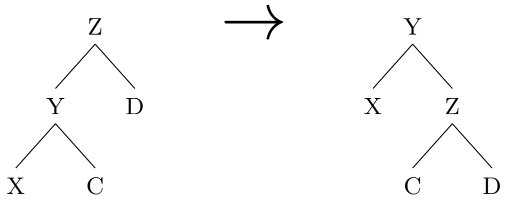
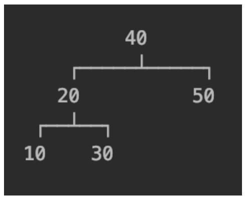
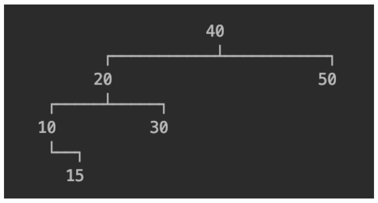
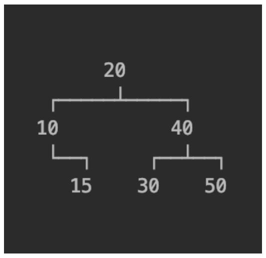


Figure 1: Right Rotation (Balancing Left-Heavy Tree)

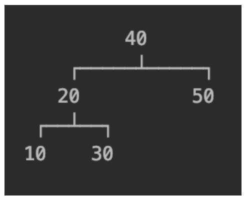


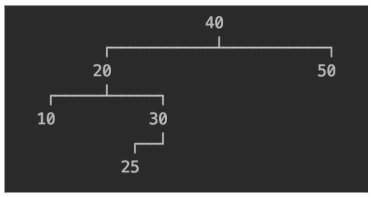


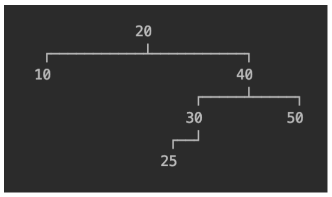
It is quite easy to detect the unbalance by comparing heights. It is also straightforward to restore the balance by a right rotation which takes the “heavy” tree on the left and slides it to the root. The general pattern of a right rotation is illustrated in Fig. 1. In our case, the rotation produces the tree below:



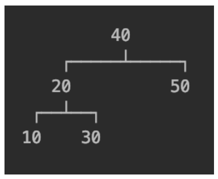
There is actually an even more unlucky case. Consider the same initial tree as before, and let’s insert 25, and follow it with a right rotation:

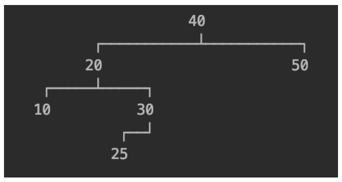


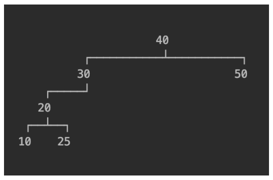


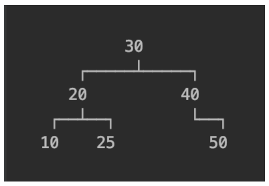


The result is however unbalanced. The reason is that the tree rooted at 30 is taller than the tree rooted at 10 and the rotation causes the tall tree to move to the other side, shifting the unbalance from one side to the other. To fix this, before doing the right rotation on the tree rooted at 40, we need to do a left rotation on tree rooted at 20:





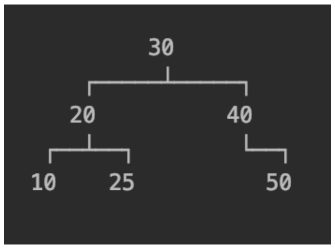




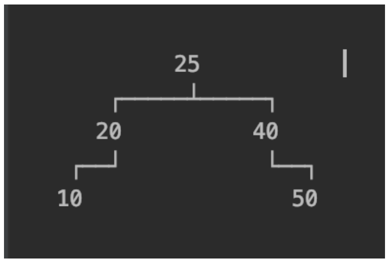
It is remarkable that this is all that is needed and that this is guaranteed to maintain balance while inserting.

2.2 Deletion

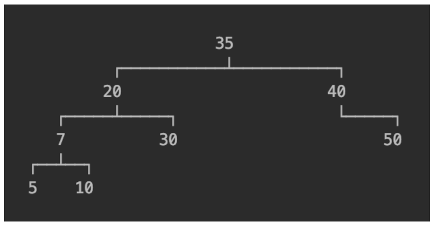
Deleting a node from an AVL has two easy cases and one difficult case. First the easy cases. Assume we want to delete a node that happens to be in the left subtree (the other situation is symmetric). We simply make a recursive call on the left subtree, check if the new height is too short, and do the appropriate rotation. The difficult case is when we need to delete the root of a tree. For example, deleting 30 from the following tree:



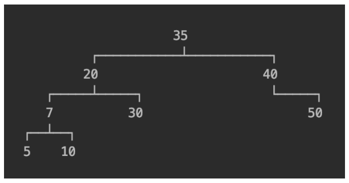
We can replace the root but the smallest node in the right subtree (40) or by the largest node in the left subtree (25). We will demonstrate the second case:

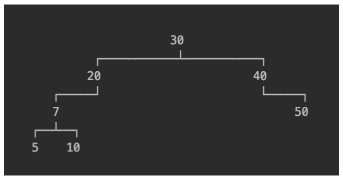


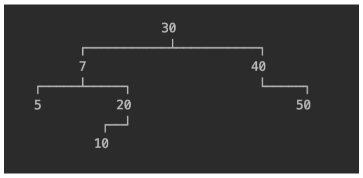
But this may not work so well. Consider deleting 35 from the following tree:



As before, we find the largest node in the left subtree (30), delete it from its position, and put it at the root. However, remove 30 from its position leaves the tree rooted at 20 unbalanced. So we must rotate it.







3 Red-Black Trees

Despite the simplicity of AVL trees, they are not the preferred approach in production software compared to red-black trees. Both kinds of trees guarantee $O(\log n)$ insertions, deletions, and search but red-black trees are often more efficient as they are less strict about the definition of balance. Technically, red-black trees maintain several invariants:

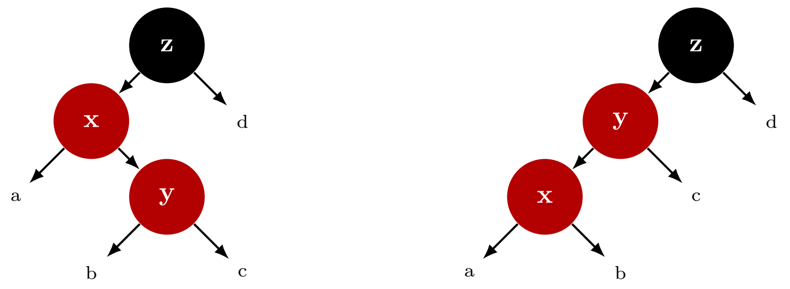


Figure 2: Red-red patterns in left subtree

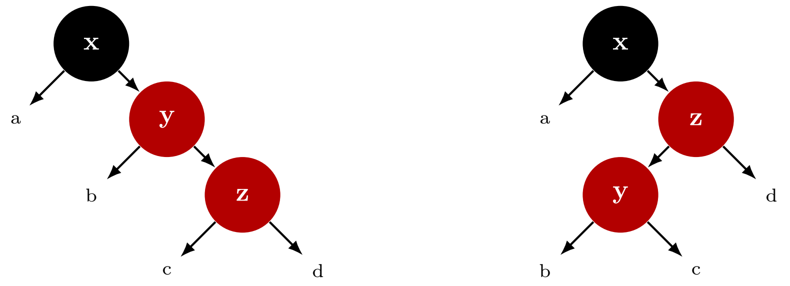


Figure 3: Red-red patterns in right subtree

1. Every node is either red or black.
2. The root is black.
3. All empty nodes are black.
4. Red nodes cannot have red children.
5. Every path from a given node to any of the empty nodes contains the same number of black nodes.

These properties ensure that the longest path in a red-black tree is at most twice the length of the shortest path, maintaining logarithmic height. Indeed since the number of black nodes from the root to any empty node is guaranteed to be the same, the only difference in height must be due to the presence of red nodes. But the constraints ensure there will never be any two consecutive red nodes along any path.

The presentation below is simplified and slightly less efficient than the production quality implementation. In this approach, insertions in red-black trees are essentially identical to insertions in AVL trees. Deletions are quite involved but simpler than the usual presentation.

3.1 Insertions

To insert a node, we navigate following the BST order until we find an empty node. There we create a new *red* node with two black empty children. If the parent of this red node is black, the tree is still well-formed and we are done. However if the parent is red, we must eliminate the red-red combinations.
Figs. 2 and 3 give the four possible arrangements of red-red violations. Each of these trees is corrected by transforming it to the tree in Fig. 4. You may recognize the correction as essentially the four AVL rotations: right, left, leftRight, and rightLeft.

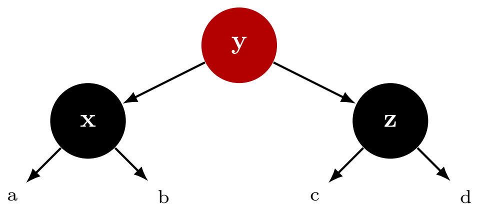


Figure 4: Result of balancing all four patterns in Figs. 2 and 3.

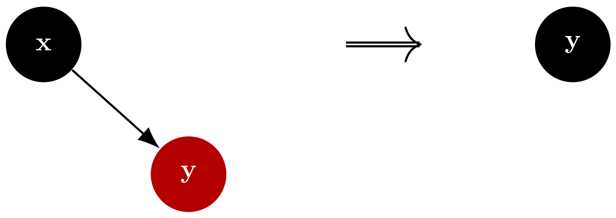
3.2 Deletion

Deleting a node in a red-black tree is more involved than insertion and requires multiple steps. The first step is familiar and similar to the deletion process for AVL trees: (i) if the value to be deleted is smaller than the current root, recur to the left and rebalance; (ii) if it is greater, recur to the right and rebalance; and (iii) if it is the current root, call a specialized method for merging the two subtrees.

The details of these merging subtrees and rebalancing are more complex in the red-black trees, however. The main issue is that deleting a black node necessarily disrupts the uniform black-height property. There are many approaches to restore the black height property. We present here an incremental solution that is relatively straightforward. The idea is to introduce two additional "colors" (or states) during deletion: *double-black* and *negative-black*. These are not actual colors but conceptual markers used to track black-height deficiencies or surpluses during rebalancing. A double-black node is a placeholder node that counts as two black nodes (one more black than a normal black node), and a negative-black node (equivalent to a "double red" node in some formulations) is a red node that is treated as if it has one fewer black than normal. These temporary states "bubble up" the tree until the discrepancy is resolved which, in the worst case, only happens at the root.

Merging subtrees. Given this setup, let us first examine the process of merging two subtrees which happens when we delete the root of a given tree. Just like for AVL trees, the idea is to extract the minimum from the right subtree, replace the deleted node with that minimum, and rebalance. There are however a few base cases that need to be treated specially:

- the deleted value was black and had only one red child: in that case, we simply color that child as black;



- the deleted value was a red leaf with two empty (black) children: in that case, we return an empty (black) node.
- the deleted value was a black leaf with two empty (black) children. In other words, the deleted node accounted for *two* black nodes in the height calculation. For this reason, we return a special empty node a double-black color. As explained below, we will then "bubble" up this double-black node up the tree to find an appropriate point where it can be eliminated.

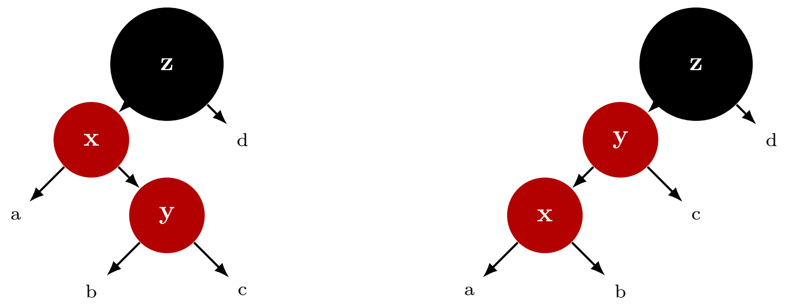


Figure 5: Doubleblack-red patterns in left subtree

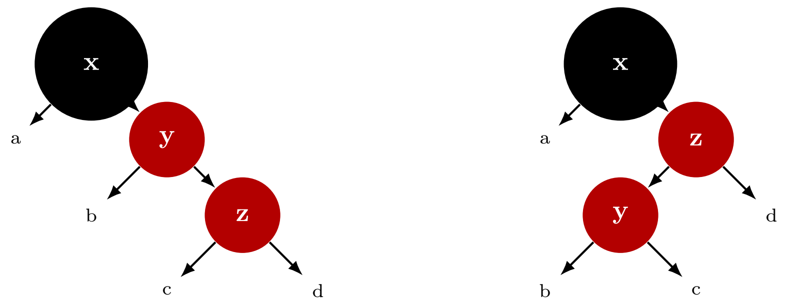
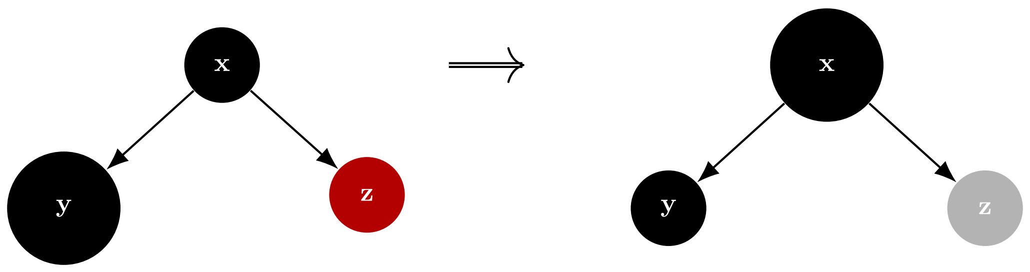


Figure 6: Doubleblack-red patterns in right subtree

Bubbling up. The bubble-up process uses just one rule: when the double-black node reaches a parent p, we absorb the extra blackness into p and reduce the blackness of the other child. For example, if the parent was black and the other child was red, we get the transformation below (where double-black is a large black circle and negative-black is a grey circle):



This transformation resolves the black-height issue at that level. This process continues up the tree until we either reach a red node or the root. If the double-black node reaches a red parent, the extra black is absorbed in the red parent and the bubbling stops. If the double-black node reaches the root, we simply color the root black, thus uniformly decreasing the black height of every node.

Balancing. As illustrated in Figs. 2, 3, and 4, four possible rotations are needed to rebalance the tree after insertion. Deletion will require six more possible rotations but fortunately four of them are identical to the ones used for insertions. These are listed in Figs. 5, 6, and 7. The remaining two balancing operations involve a negative-black child under a double-black parent. We show one case in Fig. 8 and the other is symmetric. One good intuition to remember the transformation is to think of an AVL leftRight rotation: do a left rotation on the subtree rooted at x and then a right rotation on the tree rooted at z:

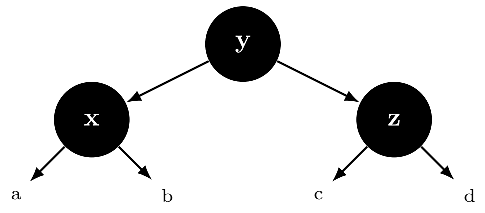


Figure 7: Result of balancing all four patterns in Figs. 5 and 6.

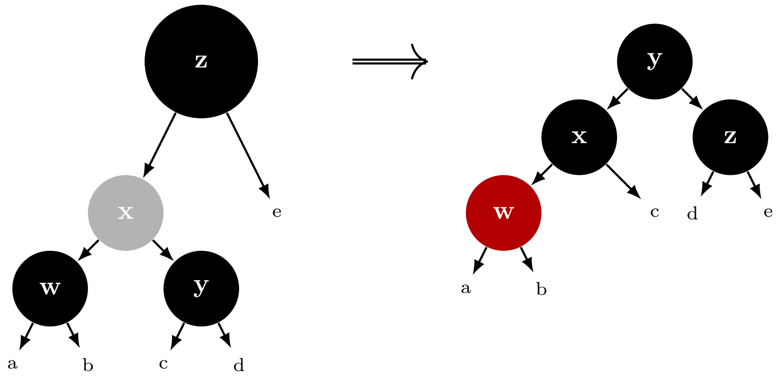


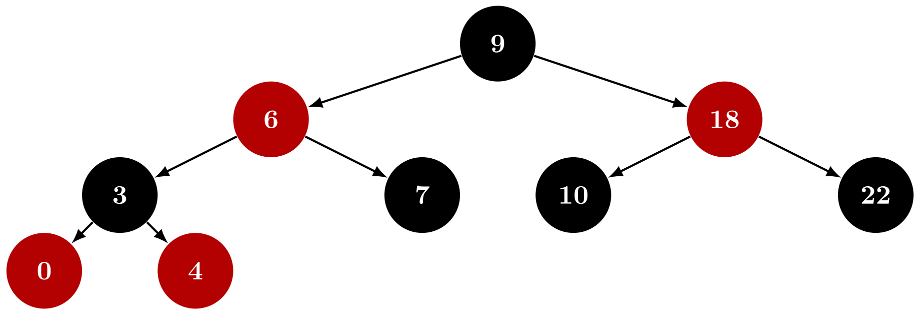
Figure 8: DoubleBlack-negativeBlack balancing

4 Examples

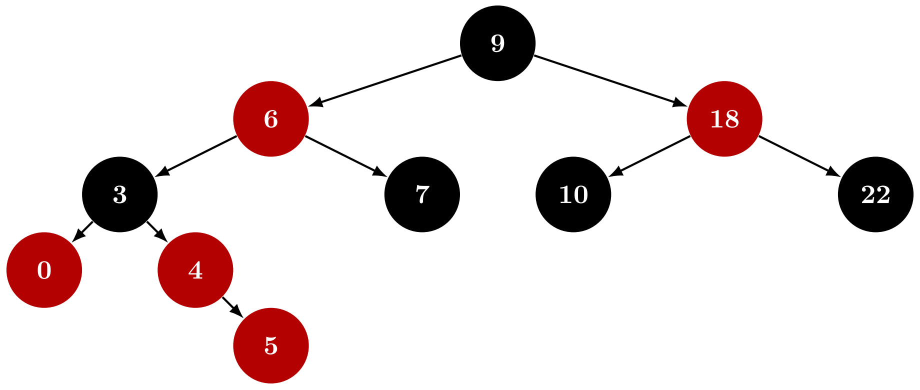
We present three examples in detail: one insertion, and two deletions.

4.1 Insert

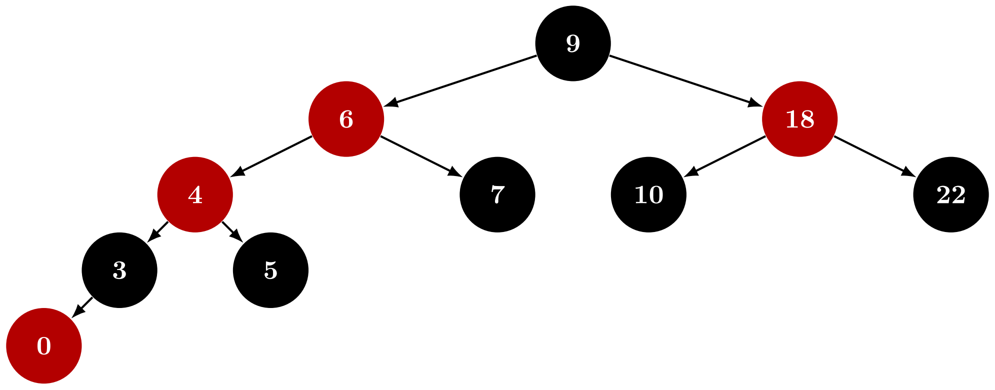
We start with this tree:



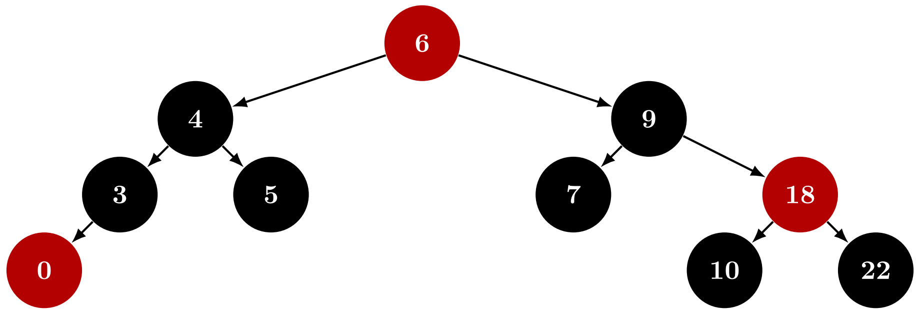
We want to insert 5. This initially creates a red node at the expected position:



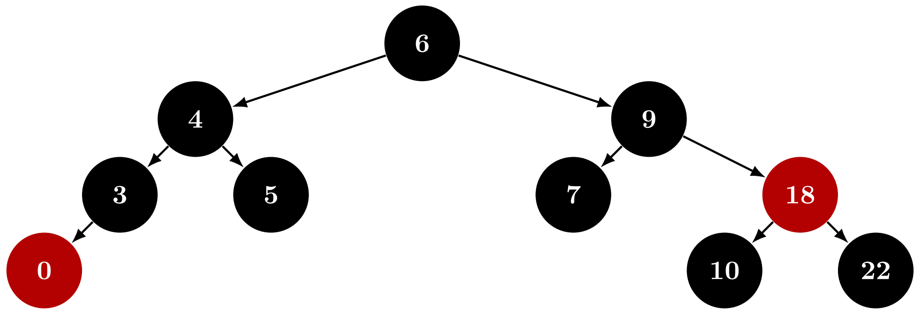
The tree has a red-red violation that is corrected by doing a left rotation at the tree rooted at 3, coloring the new root (2) as red, and coloring its children (3 and 4) as black:



This creates another red-red violation that is corrected by doing a right rotation at the root of tree (9), coloring the new root (5) as red, and coloring its children (2 and 9) black:

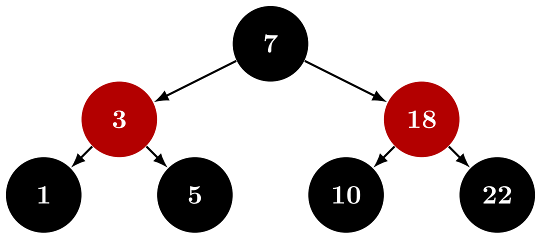


The only thing remaining is to color the root black which does not affect anything else. The final tree is below:

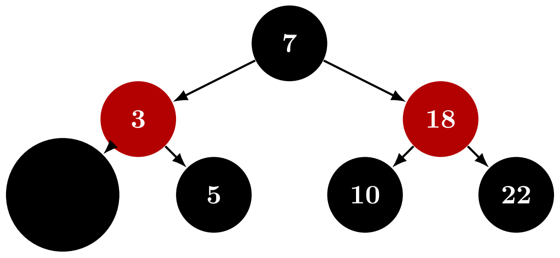


4.2 Easy delete

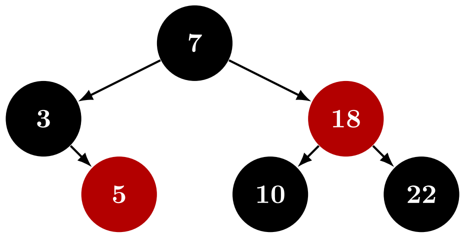
We start with this tree:



We want to delete 1. This requires us to merge the two empty subtrees under the node 1. Since 1 was colored black, the result is a double-black node:



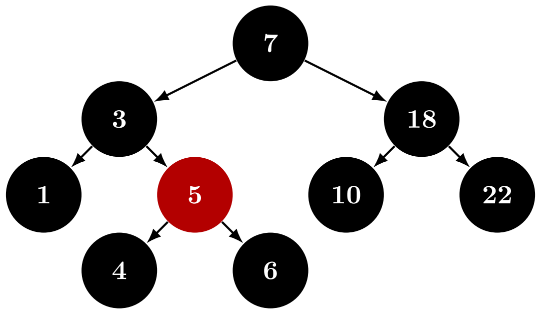
We then start the process of bubbling this double-black node up. Luckily, the parent is a red node so the situation is resolved immediately by increasing the blackness of 3 and decreasing the blackness of 5:



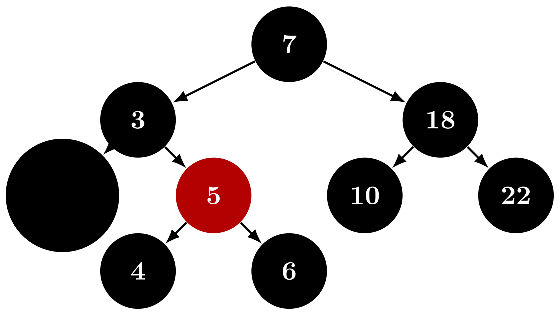
The root is black so this is the final result.

4.3 Harder delete

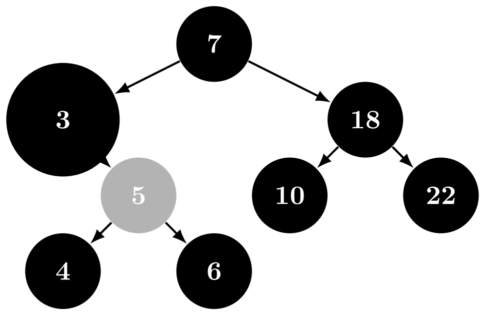
In our final example, we start with the following tree:



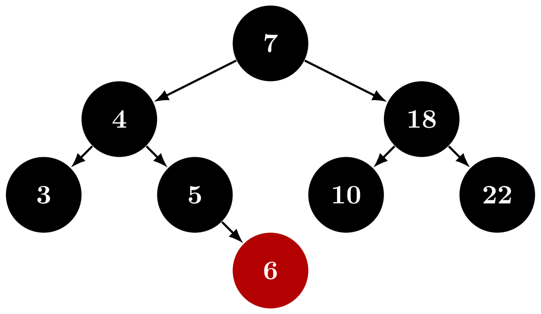
The goal is to remove 1 again. Just like in the previous example, the deletion results in a double-black node:



We now start the process of bubbling the double-black node up. Unfortunately, the parent is also black. So increasing the blackness of the parent (3) makes the parent double-black, and decreasing the blackness of the other child (5) makes the child negative-black:



The tree rooted at 3 now matches one of the cases for balancing and is transformed by doing a right rotation on the tree rooted at 5 and then a left rotation on the tree rooted at 3:


7
4
18
3
5
10
22
6