



Red-Black Tree Deletion

This lesson will cover the deletion function in Red-Black trees and will discuss all four deletion cases.

We'll cover the following



- Deletion in Red-Black Tree
- Algorithm for Deletion
- Deletion Cases
 - Case 1: Left-Left
 - Case 2: Right-Right
 - Case 3: Left-Right
 - Case 4: Right-Left

Deletion in Red-Black Tree

Before we start discussing how deletion works in a Red-Black tree, let's discuss the main difference between the Insertion and Deletion operations in Red-Black trees.

In insertion, we may violate the alternating parent-child color property, i.e., a red parent may have a red child. But in the deletion function, we may end up deleting a black node that could violate the property that “the same number of black nodes must exist from the root to the None node for every path.”



In insertion, we check the color of the *sibling of the parent* of *currentNode* and based on the color we perform appropriate operations to balance the tree. But now in the deletion operation, we will check the color of the *sibling* node of the *currentNode* and based on its color, we will perform some actions to balance the tree again.

Algorithm for Deletion

Here is a high-level description of the algorithm to remove a value in a Red-Black Tree:

1. Search for a node with the given value to remove. We will call it *currentNode*
2. Remove the *currentNode* using the standard BST deletion operation that we studied earlier

When deleting in a BST, we always end up deleting either a leaf node or a node with only one child because if we want to delete an internal node, we always swap it with a leaf node or a node with at most one child.

- In case of the leaf node deletion, delete the node and make the child of the parent of the node to be deleted **None**
- In case of a node with one child only, link the parent of the node to be deleted with that one child.

Let's name some nodes relative to Node C, which is the node that we want to delete:

- Node C – The node to be deleted (let's call it *currentNode*)
- Node P – The parent node of the *currentNode*
- Node S – The sibling node (once we rotate the tree, Node R will have a sibling node which we name Node S)
- Node SC – The child node of Node S

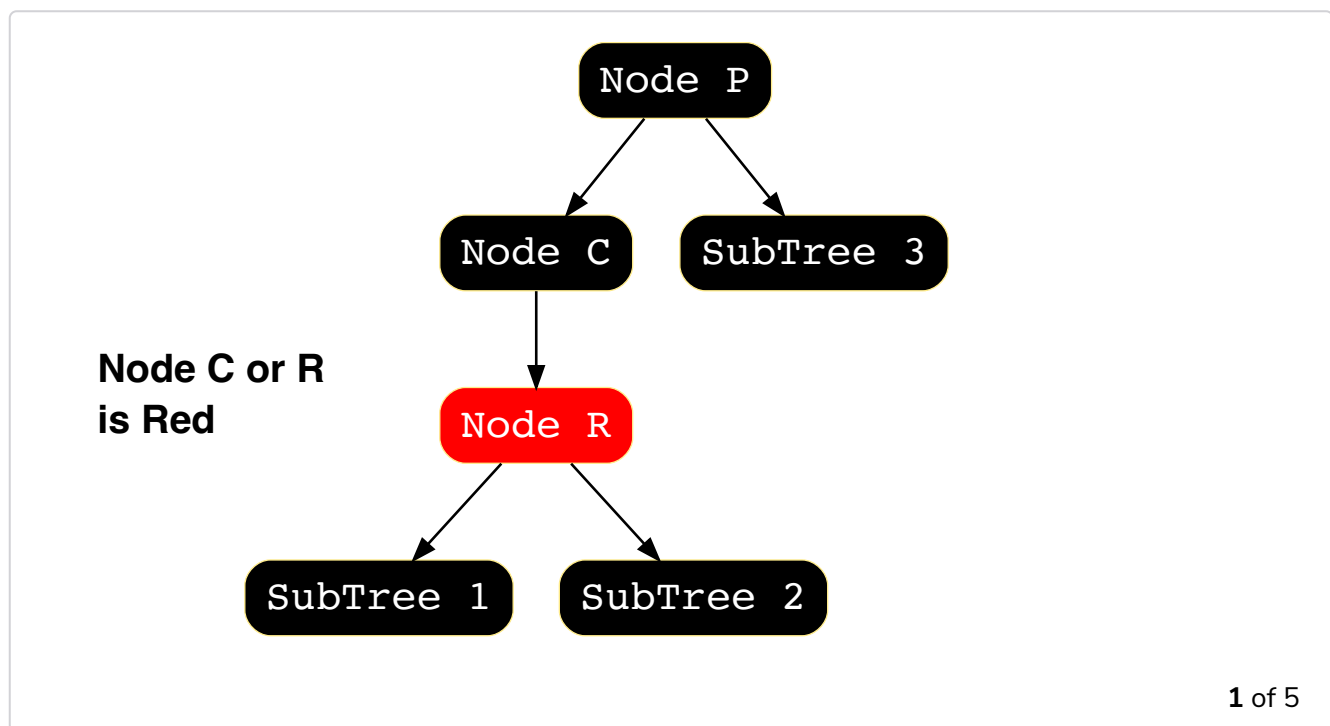


- Node R – The node to be Replaced with the *currentNode*
Node P (Node R is the single child of Node C)



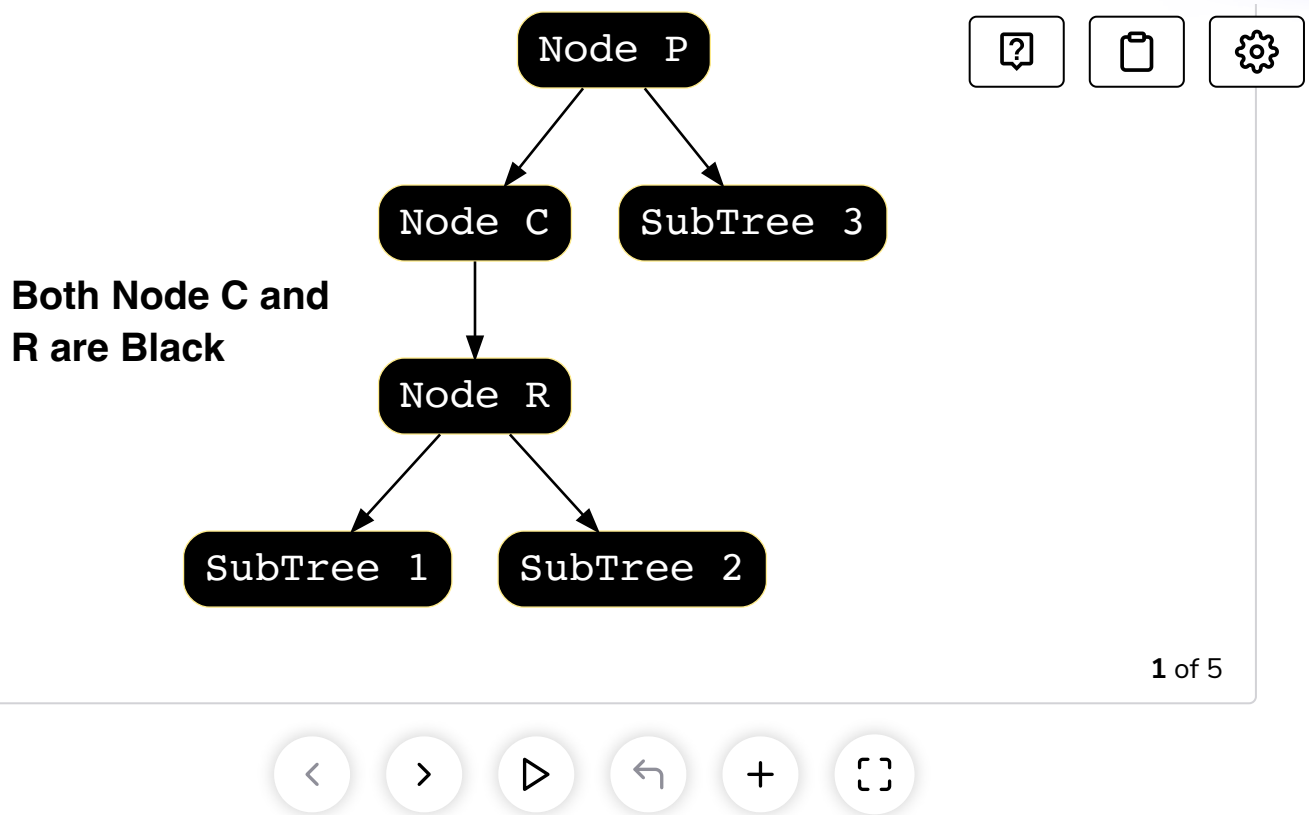
Deletion Cases

Let's study the deletion cases and the steps performed in each of these cases to make the tree balanced again. Given below is the first case in which Node C or Node R is red. In this type of scenario, we make Node R black and link it to Node P.



The second case is if both Node C and Node R are black, then make Node R black. Now Node R is double black, i.e., it was already black and, when we found both Node C and Node R black, then we again make Node R black. Remember that “None” node is always black. Let's now convert Node R from double to single black.





We need to perform the following steps while Node R is double black and it is not the root of the Tree. If Node S (sibling of Node R) is Black and one or both of Node S children are Red:

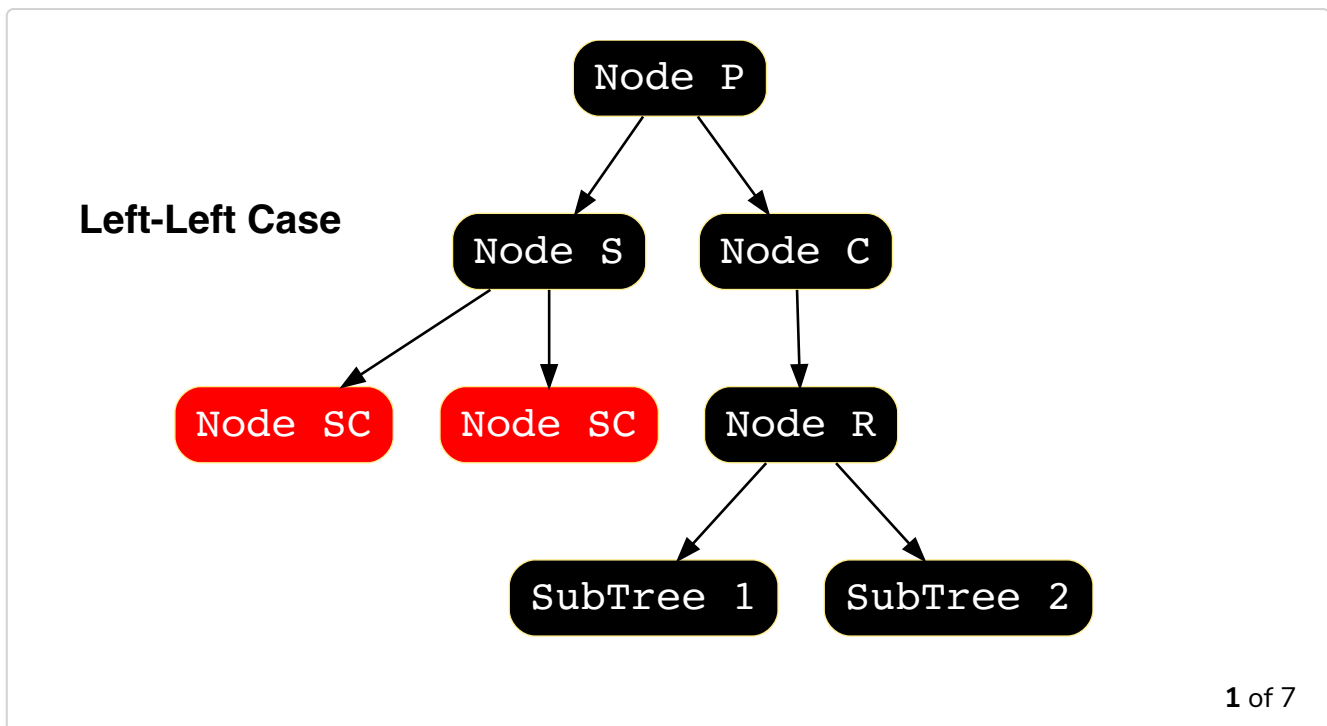
- Left-Left: Node S is *leftChild* of Node P and Node SC (red) is *leftChild* of S, or both children of S are red
- Right-Right: Node S is *rightChild* of Node P and Node SC (red) is *rightChild* of S, or both children of S are red
- Left-Right: Node S is *leftChild* of Node P and Node SC (red) is *rightChild* of S
- Right-Left: Node S is *rightChild* of Node P and Node SC (red) is *leftChild* of S

Case 1: Left-Left#



In case the Node S is *leftChild* of Node P and Node SC (red) is both children of S are red, we perform the following steps. Look at the illustration below for better understanding:

1. Rotate Node P towards right
2. Make right child of Node S the left child of Node P

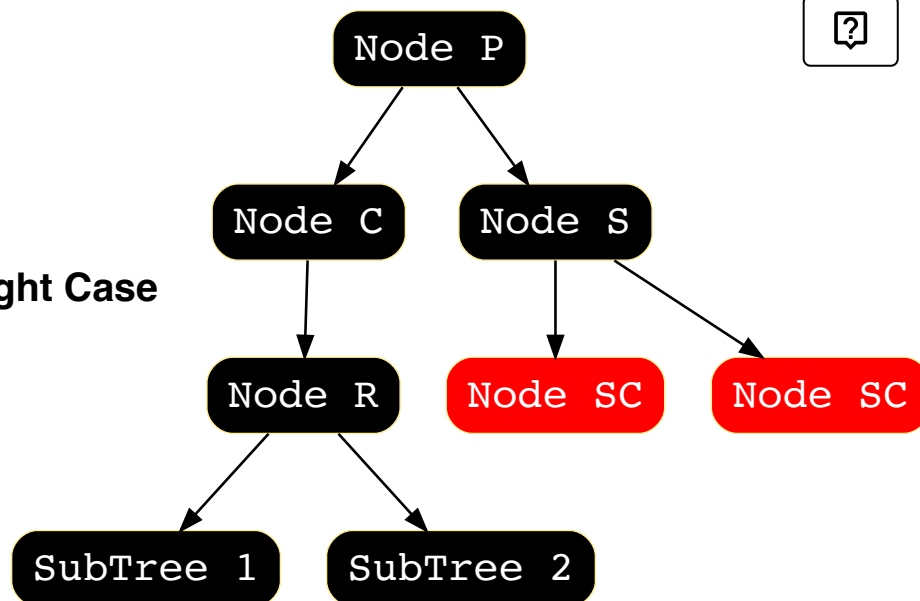


Case 2: Right-Right

In case the Node S is *rightChild* of Node P and Node SC (red) is *rightChild* of S, or both children of S are red, we perform the following steps. Look at the illustration below for better understanding:

1. Rotate Node P towards left
2. Make left child of Node S the right child of Node P



Right-Right Case

1 of 8

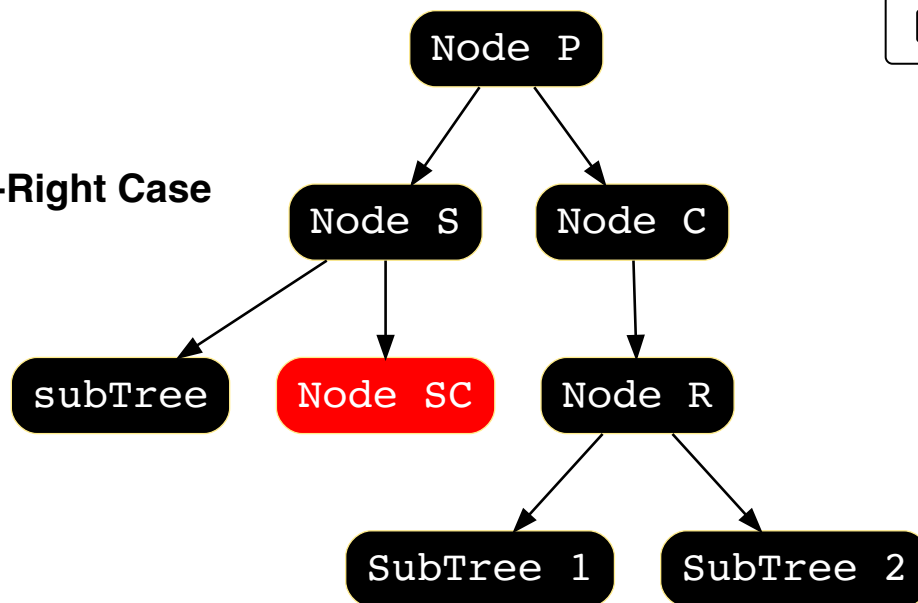


Case 3: Left-Right

In case the Node S is *leftChild* of Node P and Node SC (red) is *rightChild* of S, we perform the following steps. Look at the illustration below for better understanding:

1. Rotate Node S towards left
2. Rotate Node P towards right



Left-Right Case

1 of 9

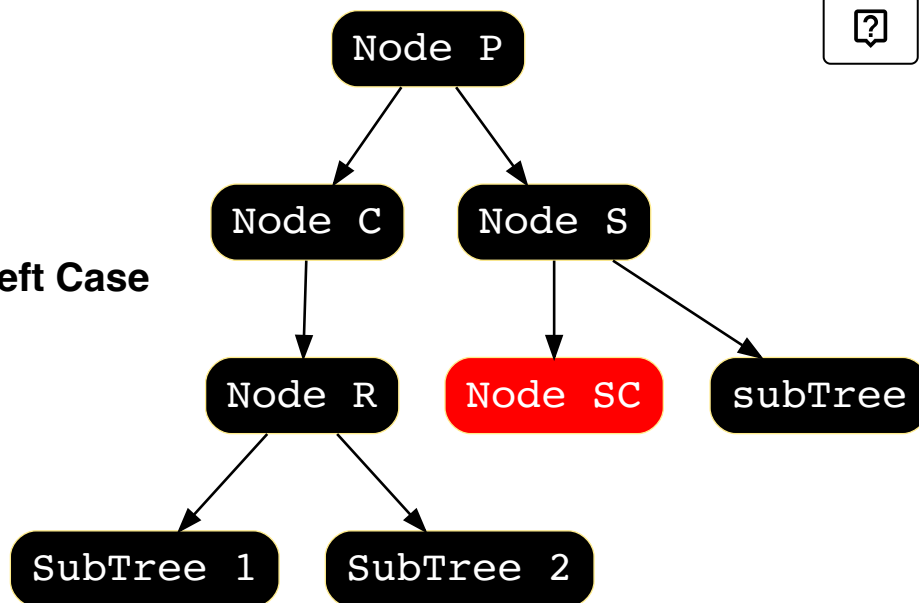


Case 4: Right-Left

In case the Node S is *rightChild* of Node P and Node SC (red) is *leftChild* of S, we perform the following steps. Look at the illustration below for better understanding:

1. Rotate Node S towards right
2. Rotate Node P towards left



Right-Left Case

1 of 10




Till this point, we have studied a number of trees from Binary Tree to different types of Binary Trees like BST and then even further types of BST like AVL and Red-Black Trees. We have covered both basic deletion and insertion operations of these trees along with Python Implementation of BST. We are now left with only one important tree data structure known as 2-3 Trees. So, in the next lesson, we will cover it in detail just like we did with the rest of tree structures.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. [See how](#) ⓘ


[← Back](#)
[Next →](#)




 Report an Issue

