Red Black Trees

**What is a Red-Black Tree?**

**Introduction**

Red-Black Trees are another type of self-balancing Binary Search Tree, but with some additions: the nodes in Red-Black Trees are colored either red or black. Colored nodes help with re-balancing the tree after insertions or deletions. We will go through the insertion and deletion functions of Red-Black trees just like we did with AVL Trees previously.

**Properties of Red-Black Trees**

* Every node is either **Red** or **Black** in color
* The root is always colored, **Black**
* Two **Red** nodes cannot be adjacent, i.e., No red parent can have a red child and vice versa
* Each path from the root to None contains the same number of **Black** colored nodes
* The color of *NULL* nodes is considered **Black**

From the perspective of implementation, our node class will contain the addition of a *boolean* variable that will store the information about the color of a node. Here is a basic structure of a Node, which will be used to build a Red-Black tree.

class Node

{

    int value;

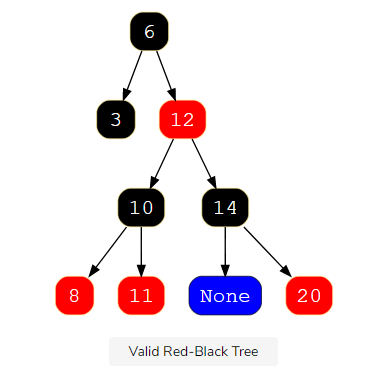
    Node\* leftChild;

    Node\* rightChild;

    bool isRed;

};

Here is an example of a valid Red-Black Tree:



**Time Complexity**

Balancing the tree doesn’t result in a tree being perfectly balanced but it is good enough to make time complexity of basic operations like searching, deletion, and insertion to be around O(log n)*O*(*logn*).

**AVL vs. Red-Black Trees**

Although AVL Trees are technically more ‘balanced’ than Red-Black Trees, AVL Trees take more rotations during insertion and deletion operations than Red-Black Trees. So, if you have search-intensive applications where insertion and deletion are not that frequent, use AVL Trees, otherwise use Red-Black Trees.

As the above operations involve a series of steps and cases to follow to fulfill the property of Red-Black Trees and to keep the Trees balanced, we will look into each operation of insertion and individually.

# Red-Black Tree Insertion

## Insertion in Red-Black Tree

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

1. Insert the given node using the standard BST Insertion technique that we studied earlier and color it **Red**.
2. If the given node is the root, then change its color to **Black**
3. If the given node is not the root, then we will have to perform some operations to make the Tree follow the Red-Black property.

## Rebalancing the Tree

There are two ways to balance an unbalanced tree:

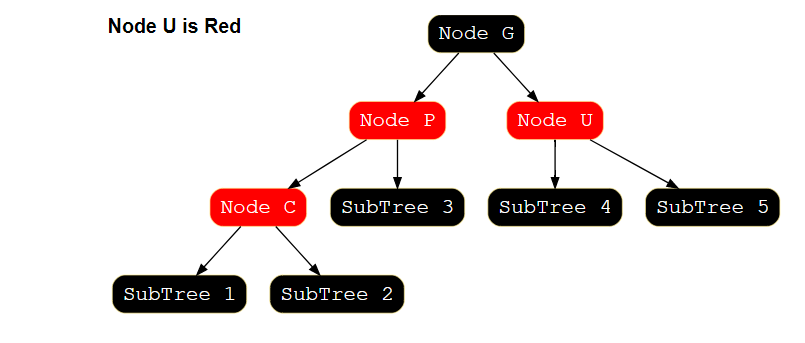
1. Recoloring Nodes
2. Rotating Nodes (left or right)

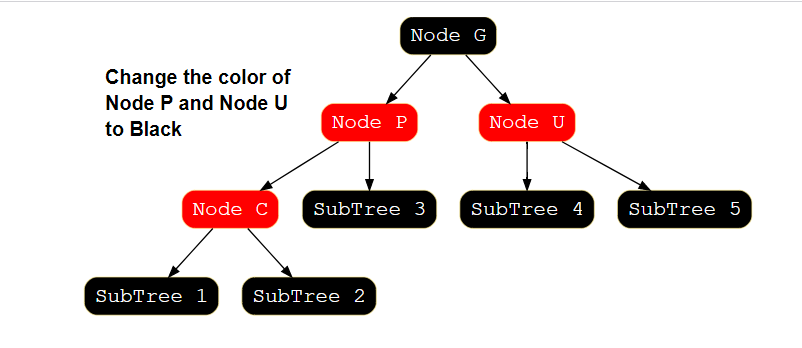
But before the details are explained, let’s define the structure of the Red-Black Tree and some nodes relative to the given node, which is the node that we inserted in the Tree.

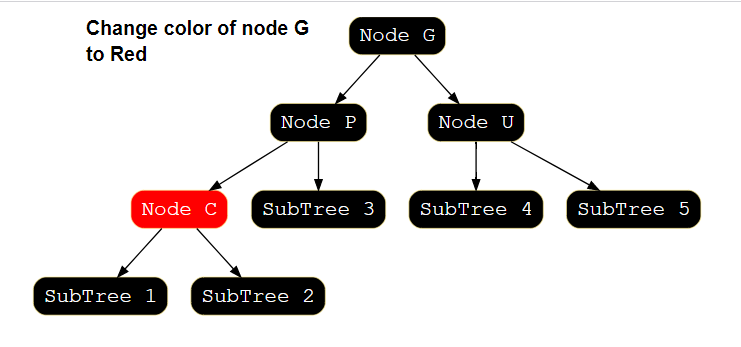
* Node C – the newly inserted node
* Node P – the parent of the newly inserted node
* Node G – the grandparent of the newly inserted node
* Node U – the sibling of the parent of the newly inserted node, i.e., the sibling of Node P / child of Node G

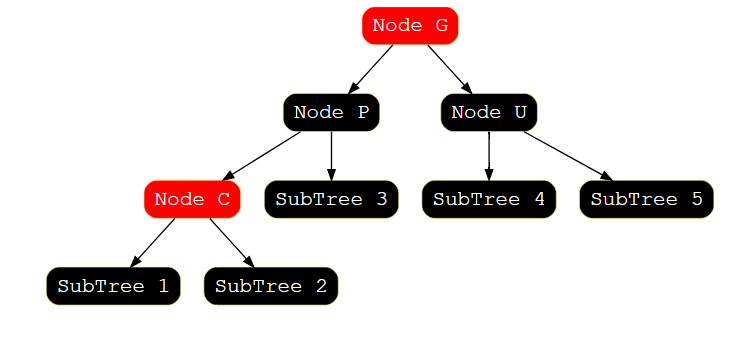
If the newly inserted node is not a root and the parent of the newly inserted node is not **Black**, first, we will check Node U, which is based on Node U’s color, we balance the Tree. If Node U is **Red**, do the following:

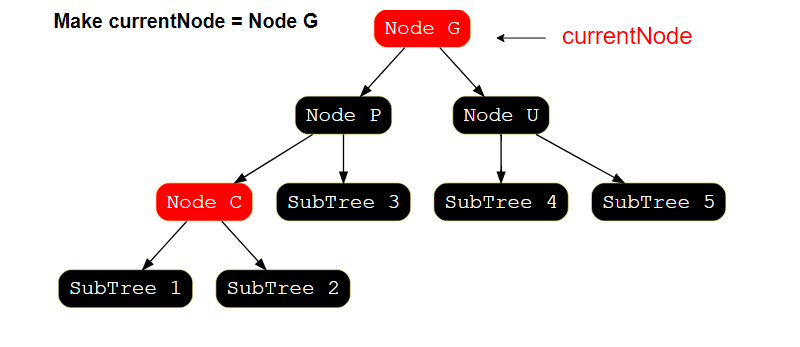
1. Change the color of Node P and U to **Black**
2. Change the color of Node G to **Red**
3. Make Node G the new currentNode and repeat the same process from step two

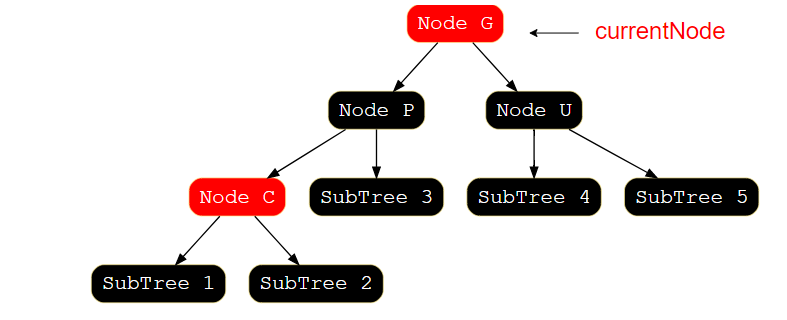












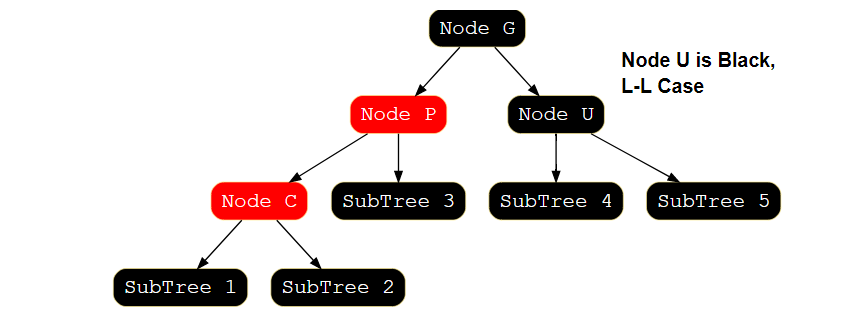
If Node U is **Black**, then we come across four different scenarios based on the arrangements of Node P and G, just like we did in AVL trees. We will cover each of these scenarios and try to help you understand through illustrations.

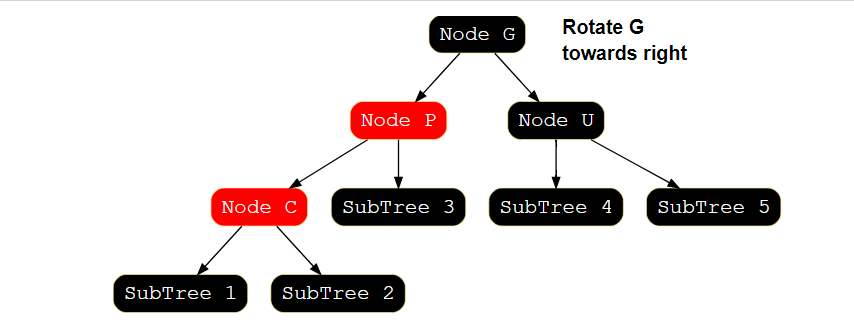
* Left-Left: Node P is the leftChild of Node G, and currentNode is the leftChild of Node P
* Left-Right: Node P is the leftChild of Node G, and currentNode is the rightChild of Node P
* Right-Right: Node P is the rightChild of Node G, and currentNode is the rightChild of Node P
* Right-Left: Node P is the rightChild of Node G, and currentNode is the leftChild of Node P

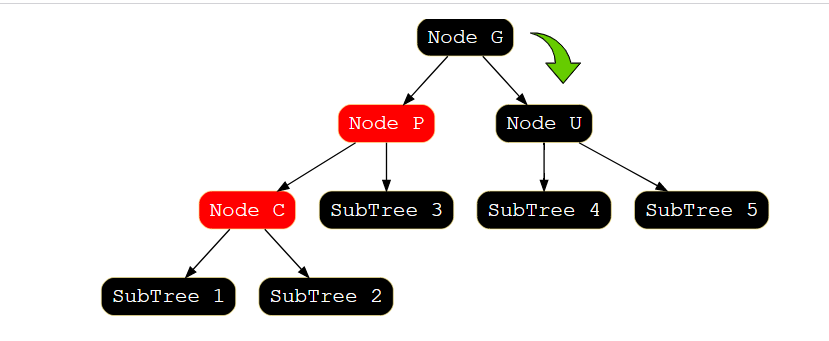
### Case 1: Left-Left

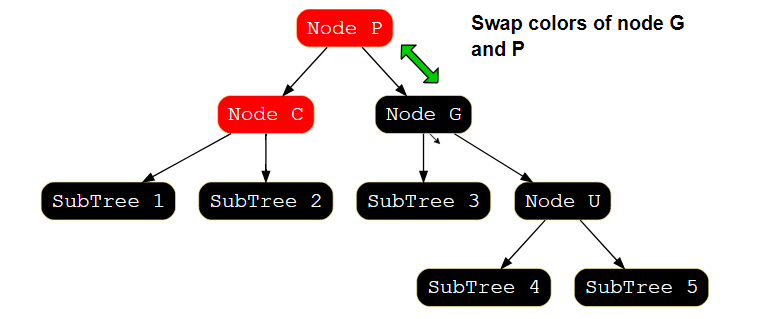
In case the Node P is the leftChild of Node G, and currentNode is the leftChild of Node P, we perform the following steps. Look at the illustration below for a better understanding.

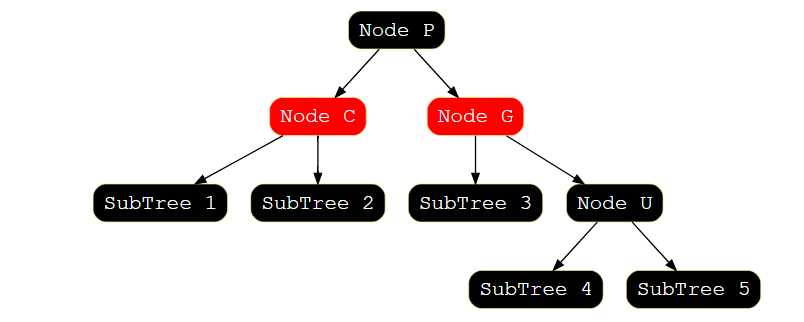
1. Right Rotate Node G
2. Swap the colors of Nodes G and P







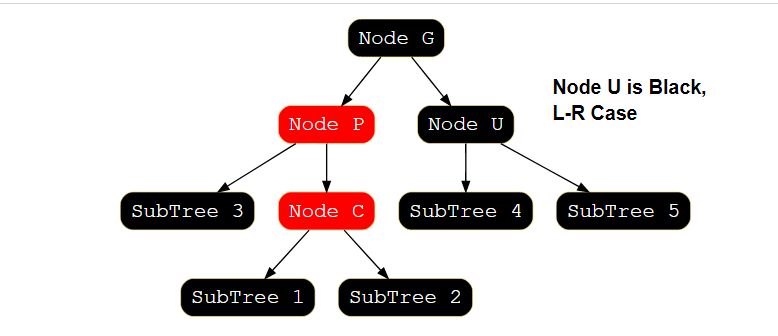


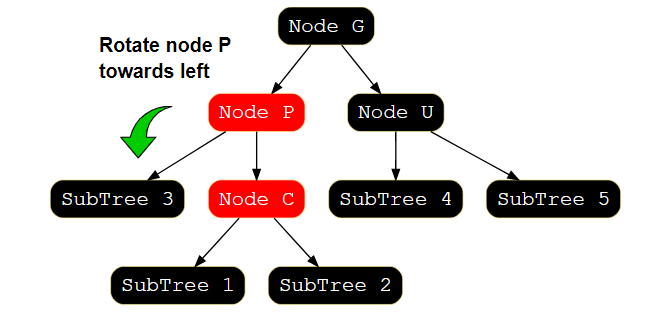


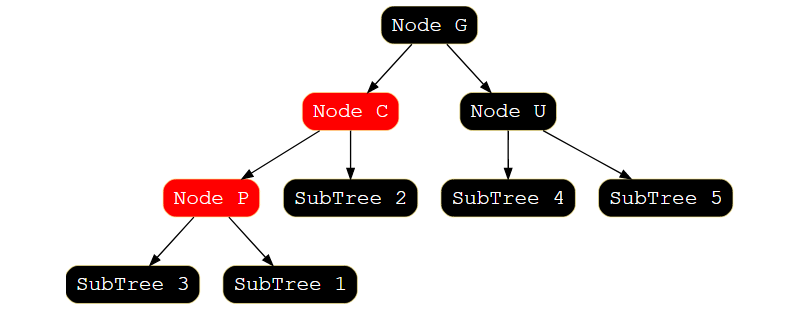
### Case 2: Left-Right

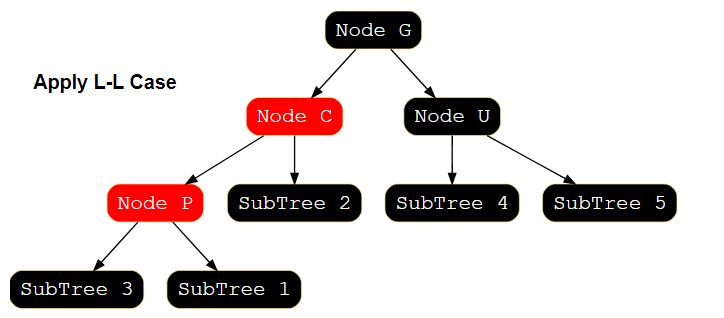
In the case when the Node P is the leftChild of Node G, and the currentNode is the rightChild of Node P, we perform the following steps. Look at the illustration below for better understanding:

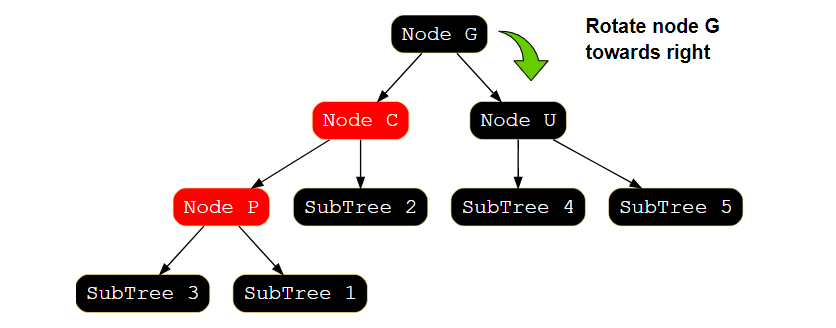
1. Left Rotate Node P
2. After that, repeat the steps that we covered in the Left-Left case

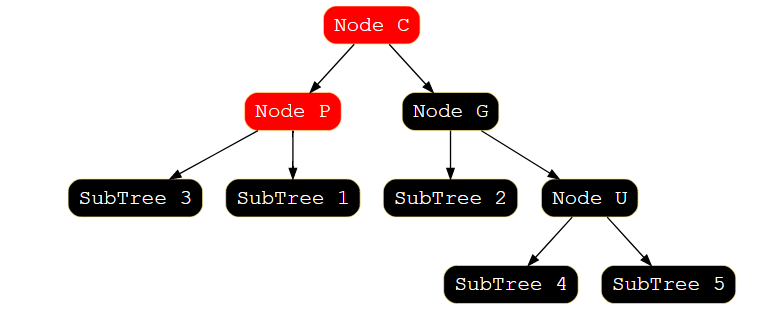


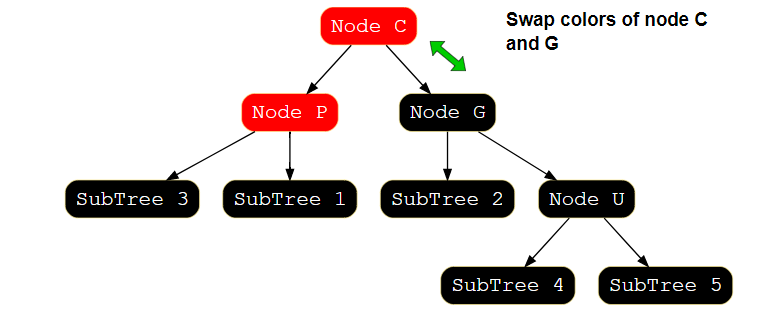


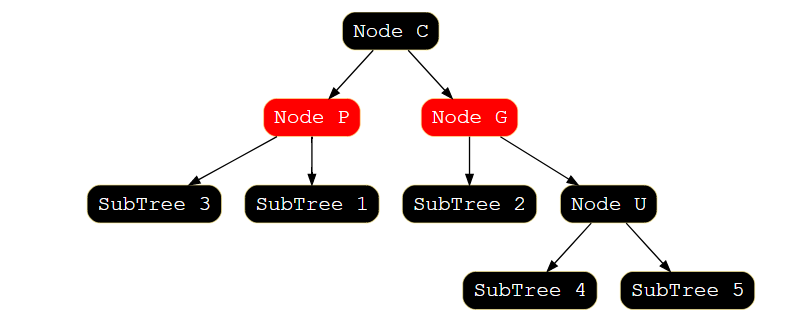








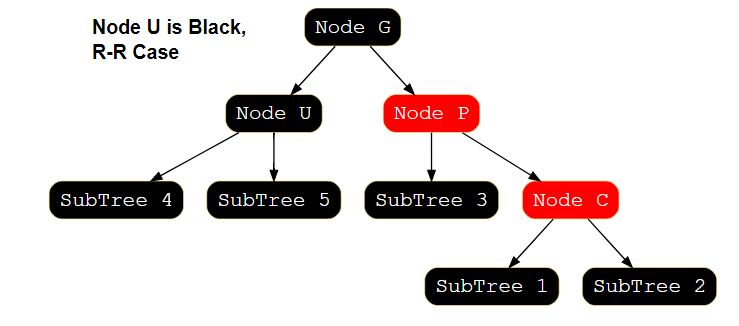


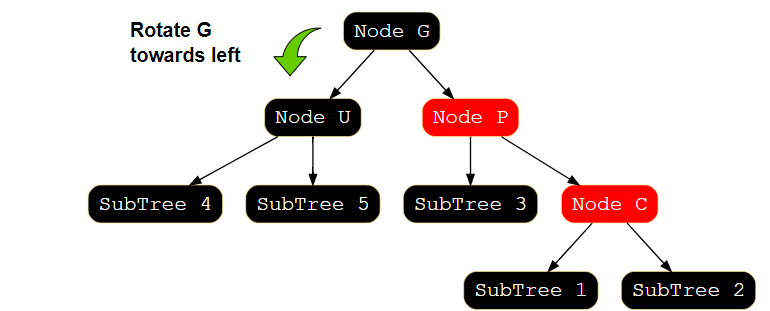


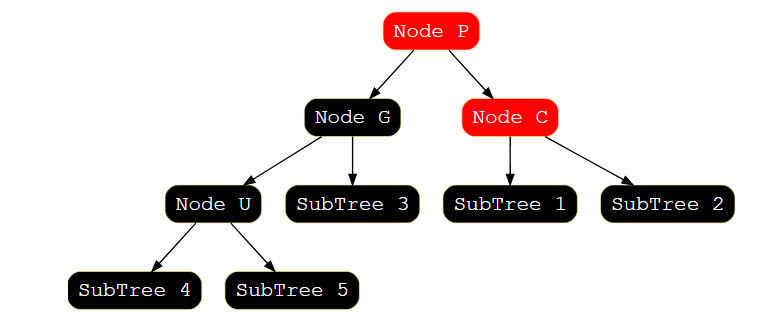
### Case 3: Right-Right

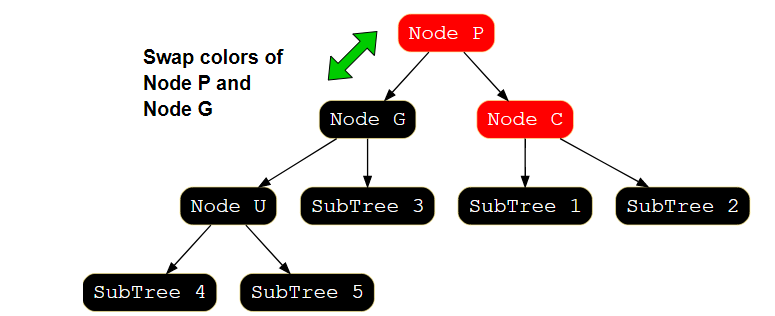
In the case when Node P is the rightChild of Node G, and the currentNode is the rightChild of Node P, we perform the following steps. Look at the illustration below for a better understanding.

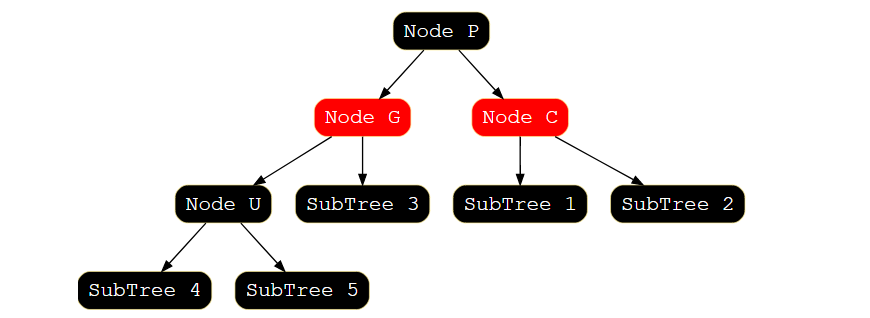
1. Left Rotate Node G
2. Swap colors of Node G and P







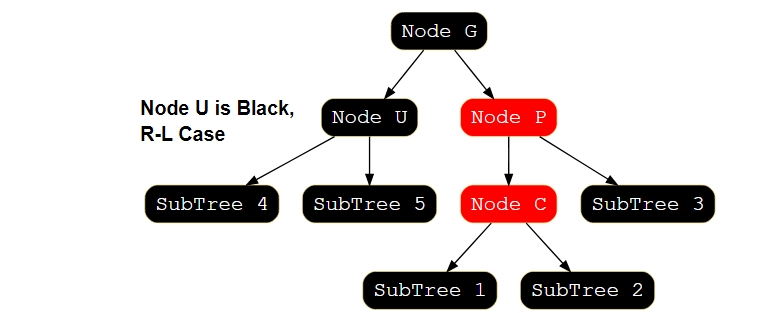


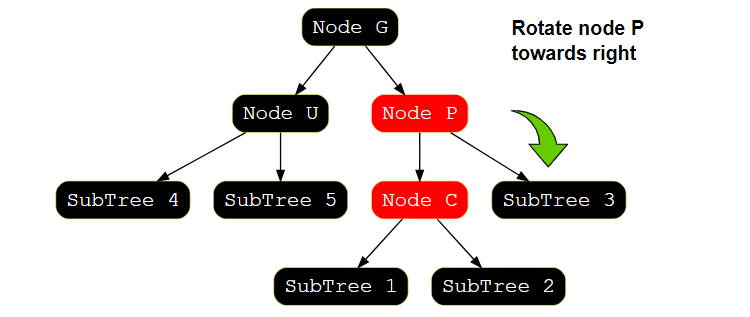


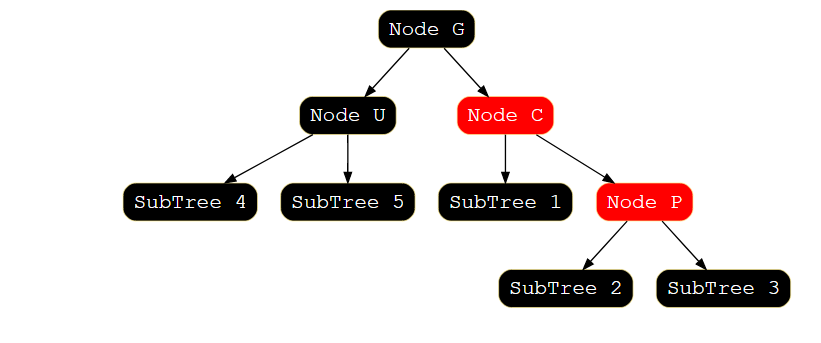
### Case 4: Right-Left

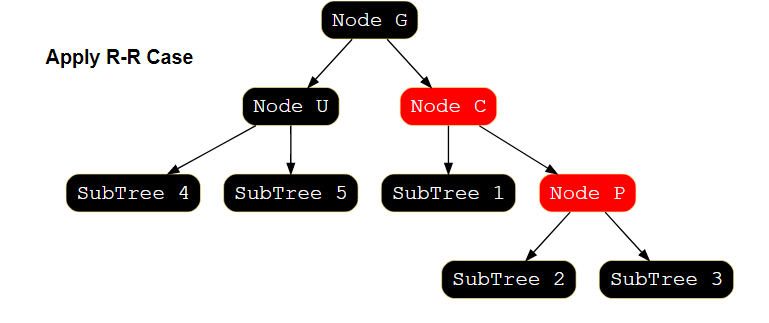
In the case when Node P is the rightChild of Node G, and the currentNode is the leftChild of Node P, we perform the following steps. Look at the illustration below for a better understanding.

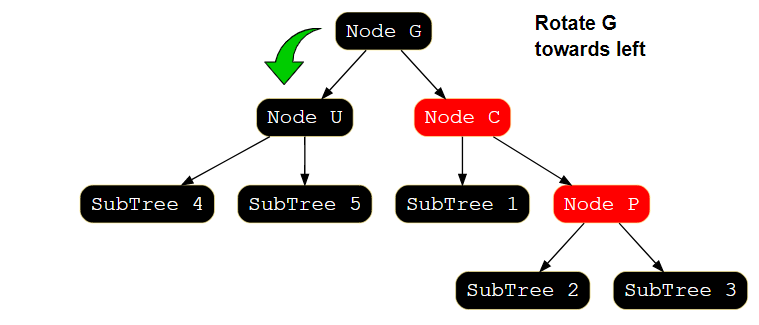
1. Right Rotate Node P
2. After that, repeat the steps that we covered in Right-Right case

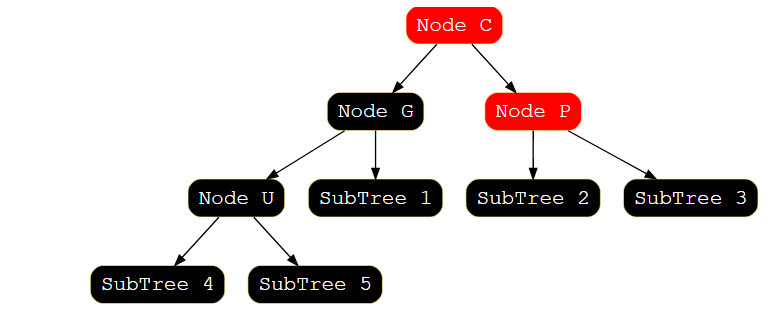


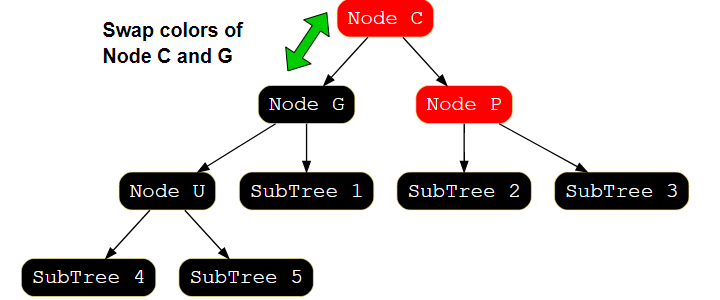


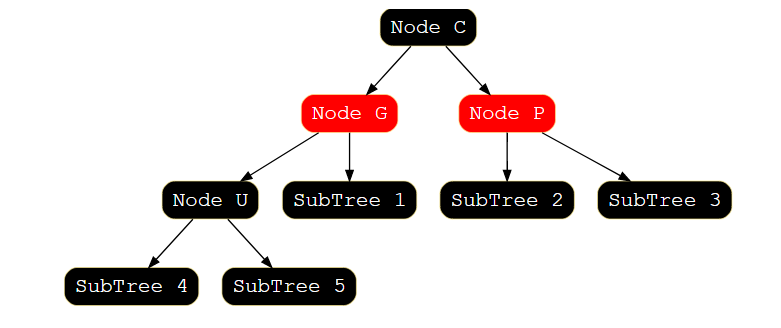












### Time Complexity

The time complexity of the insert operation is O(log(n))*O*(*log*(*n*)).

# Red-Black Tree Deletion

## 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 colors 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 the 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 the 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.

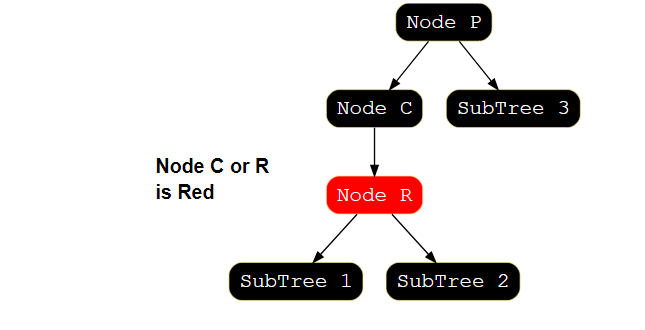
* In case of the leaf node deletion, delete the node and make the child of the parent of the node to be deleted None.
* In the case of a node with one child only, link the parent of the node to be deleted with that one child.

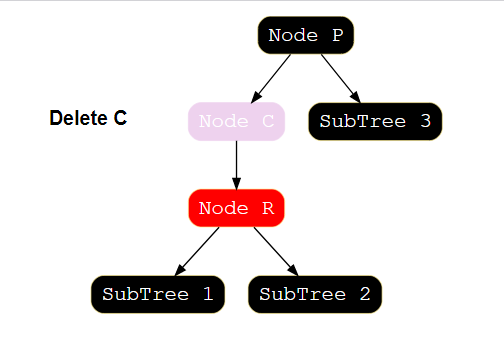
Let’s name some nodes relative to Node C, which is the node that we want to delete:

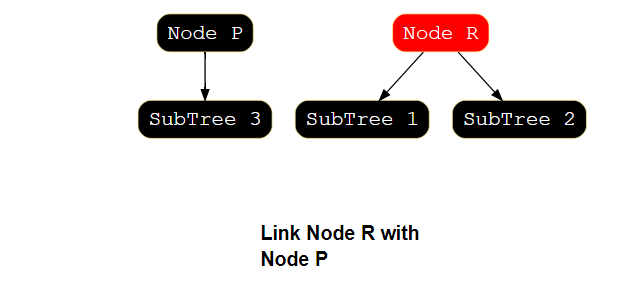
* Node C – The node to be deleted (let’s call it currentNode)
* Node P – The parent node of the currentNode
* Node S – The sibling node (once we rotate the tree, Node R will have a sibling node which we name Node S)
* Node SC – child node of Node S
* Node R – node to be Replaced with the currentNode and linked with Node P (Node R is the single child of Node C)

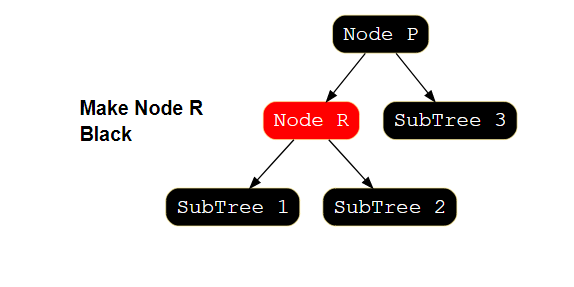
## Deletion Cases

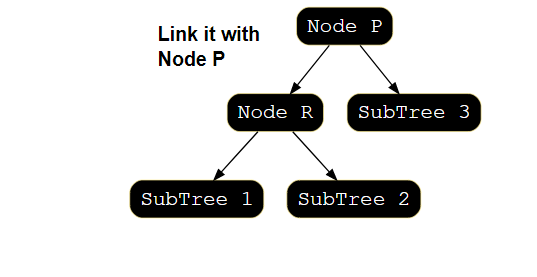
Let’s study the deletion cases and the steps performed in each of these cases to make the tree balanced again. Given below is the first case in which Node C or Node R is red. In this type of scenario, we make Node R black and link it to Node P.



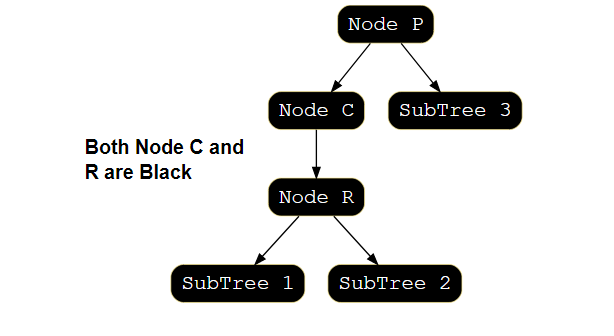


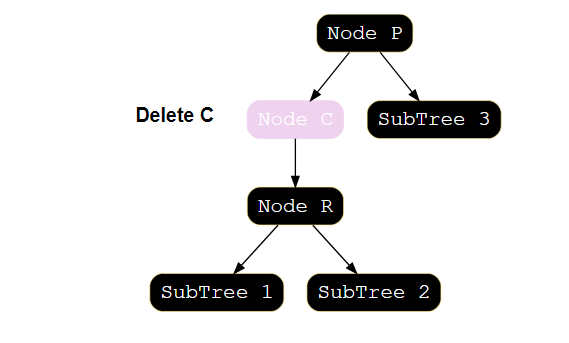


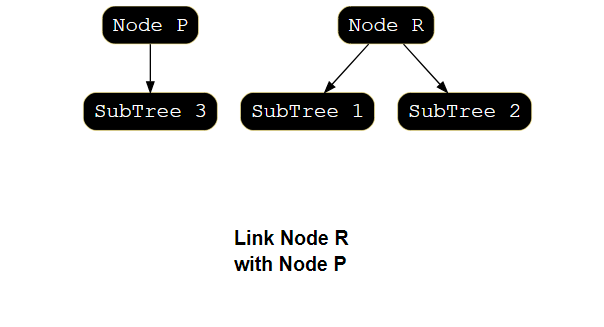


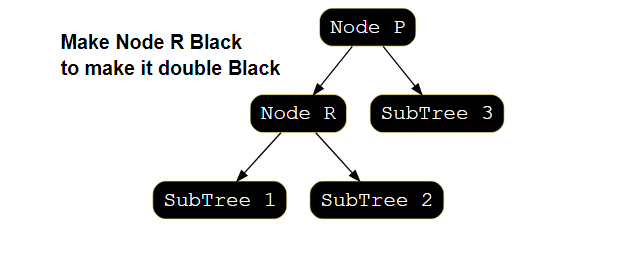


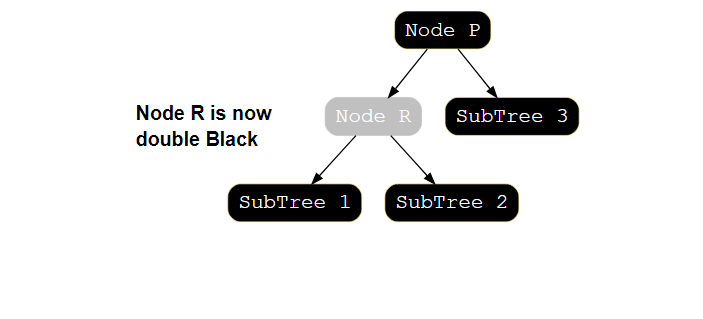
The second case is if both Node C and Node R are black, then make Node R black. Now Node R is double black, i.e., it was already black, and when we found both Node C and Node R black, then we again make Node R black. Remember that the “None” node is always black. Let’s now convert Node R from double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











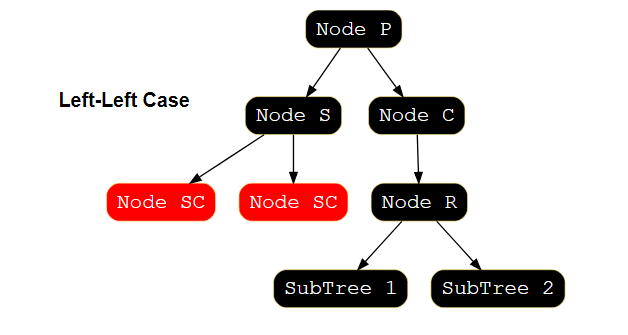
We need to perform the following steps while Node R is double black, and it is not the root of the Tree. If Node S (sibling of Node R) is Black and one or both of Node S children are Red:

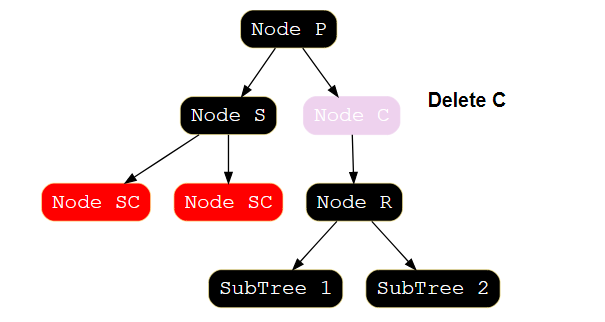
* Left-Left: Node S is *leftChild* of Node P, and Node SC (red) is *leftChild* of S, or both children of S are red
* Right-Right: Node S is *rightChild* of Node P, and Node SC (red) is *rightChild* of S, or both children of S are red
* Left-Right: Node S is *leftChild* of Node P and Node SC (red) is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

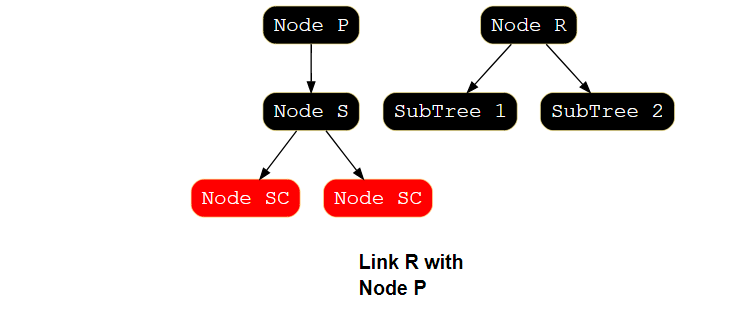
### Case 1: Left-Left

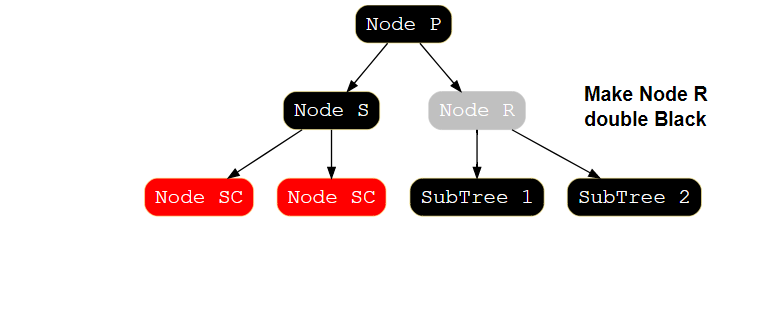
In case the Node S is leftChild of Node P, and Node SC (red) is leftChild of S, or both children of S are red, we perform the following steps. Look at the illustration below for better understanding:

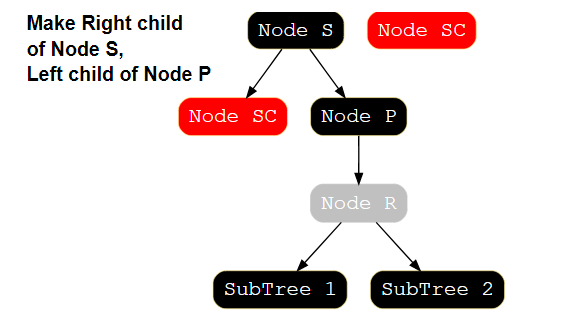
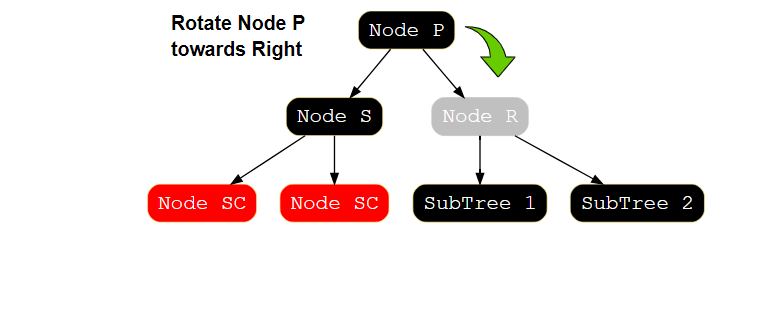
1. Rotate Node P towards the right
2. Make the right child of Node S the left child of Node P











### 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

### 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

### 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 the right
2. Rotate Node P towards left

### Time Complexity

The time complexity of the delete operation is O(log(n))*O*(*log*(*n*)).