***General information about the Red-Black tree***

A red-black tree is a type of Binary Search Tree (BST). This tree can balance itself automatically. Each node has a color. It can be red or black. It is complex, but has a good worst-case running time for its operations and is efficient in practice: it can search, insert, and delete in *O*(log *n*) time, where *n* is the total number of elements in the tree. After inserting and deleting nodes, they can change their color. These colors are used to show that the tree remains balanced during insertions and deletions. Red-black trees are described with and without NIL nodes. We can call them leaves. Also, in programming, we can symbolize them by NULL references.

There are some rules for RBT:

1)Each node is red or black

2) All null(NIL) leaves are black

3) The root must be black

***Why do we use red-black trees?***

As I said, red-black trees have *O*(log *n*) time complexity(search, insert, delete). N is number of the elements in tree. The Red-Black trees guarantee a O(log(n)) in insert, delete (even in worst case). They are balanced search trees and therefore balance themselves to always maintain a height of log(n).

Consider inserting 1,2,3,4,5 into a binary tree. It’ll make 1 as the root and all the following elements would keep going to the right thus forming a linked list in principle (and each operation thus taking O(n) time).

Average time complexity may be the same, but if we consider the worst-case (which we always do), the time complexity of red-black trees is better than binary search trees.

***So let’s explain the code.***

Firstly, we must define Node. Nodes are symbolized by the Node class. We create a **Node class** and then its **value, parent, left and right child** of the Node**. \_\_init\_\_\_** method means to initialize and we call them when attributes of this class are created. **Self.color= 1 means** that new node is always inserted as Red Node. Then, we define red-black tree in RBT class. As I said before null is always black, so null color is 0. In tree, we insert a new node that must be red. We set its color to 1(red). Then we find place for the new node. If its value is smaller than parent node or root, we place it left, else, right. We set parent of node.

.

Suppose there is a tree and we have to count the number of leaves on that tree. We use recursion to know how long we will go. Due to a large number of branches, recursion simplifies this calculation.

There are rotation operations. The positions of the nodes of a subtree are changed during operation.

When other processes, such as insertion and deletion, violate the attributes of a red-black tree, the rotation operation is employed to restore them

Rotations are divided into two categories:

Left Rotate In the left rotation, the right-hand node arrangement is turned into the left-hand node configuration.

The arrangement of the nodes on the left is changed into the configurations on the right node right rotation.

Left rotation starts on line 54. “a” is the right child of “b”. There I changed right child of b to left child of a and parent of an as parent of b.

Code for right rotation starts on line 71. Change left child of b to right child of a and parent of b as parent of a.

***Insertion and Deletion:***

To add a new node, follow the instructions in the corresponding article's "binary search tree insertion" section. To put it another way, we start at the root and work our way down, attaching the new node to a leaf or half-leaf.

The code can be found line 101.

Inserting an Element in a Red Black Tree: Steps

Check to see if the tree is empty. Insert a new node and color it Black if the tree is empty. Because the color of the Root Node must always be black

Otherwise, if the Tree is not empty, add the new node at the end as a leaf node and color it Red.

If the new node's parent is Red, and its parent's node is also Red, then the color of the new node's neighbor, Parent, and Grandparents is flipped to Black.

If the new node's parent is Red and its parent's node is empty or NULL, the new node and parent should be rotated.

Output: R---- 19(BLACK)

L---- 3(RED)

| R---- 4(RED)

| R---- 8(RED)

R---- 24(RED)

R---- 37(RED)

After deleting:

R---- 19(BLACK)

L---- 3(RED)

| R---- 4(RED)

| R---- 8(RED)

R---- 24(RED)

>