**Description**:

**Link to previous article** **on** **insertion** : https://cppsecrets.com/users/80971121111111141184611297116110101494864103109971051084699111109/Python-program-to-understand-Red-Black-trees-and-insert-nodes-to-it.php

**Link to source code** : https://github.com/apoorvpatne10/Red-Black-Tree

This python article involves deletion of nodes from a Red-Black tree. According to stuff I covered in the previous article about insertion in a red-black tree, we check color of the uncle (parent's sibling) node to decide the appropriate case. In delete operations, we also check the color of sibling to decide the case it is applicable to. As we know, the main property that violates after insertion is 2 consecutive reds.

In delete, the main course of action that may violate the property of red black trees is the change of black height in sub-trees. This is because deletion of a black node may cause reduced black height in one root to leaf path. To perform deletion in some cases, a term called as **black node** is used. When a black node is deleted and replaced by a black child, the child is marked as double black. The objective is to convert this double black to single black so that all the properties of Red-Black trees are preserved.

Following are the properties of Red-Black trees just for our reference:

**1.** Each node is either red or black.

**2**. The root of the tree is always black.

**3.** All leaves are null and they are black.

**4.** If a node is red, then its parent is black.

**5.** Any path from a given node to any of its descendant leaves contain the same number of black nodes.

**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

Now I'll go through all the possible cases of deletion of a node in a red-black trees:

Initially, we'll delete a node just like we delete a node in a normal binary search tree. Perform deletion operation is simple. The goal is to delete a node which is either leaf or has only one child. When we encounter an internal node, we try to find the in-order successor of the node and replace it with its in-order successor. Then we delete the in-order successor.  Successor is ALWAYS a leaf-node or a node with one child.

Here are the cases that we'll be looking at:

**1.** Red leaf node

**2.** Black node, with one red child node

**3.** Black leaf node

**->** In the first case, the normal binary search tree delete is sufficient. Removing a red node doesn't change the black depths of any node, nor does it create a red child for any red node.

**->** In the second case, after we complete the binary search tree delete, we must simply recolour the child node of the deleted node to black. Changing this colour adds one to the black depths of each node in the sub-tree of the deleted node, thus restoring the equality of the black depths of all external nodes. This also makes sure that changing a node to black doesn't violate any of other red-black tree specifications.

**->** The third case is pretty different and neither of the above strategies are sufficient for dealing with the third case. When we patch in the child of the deleted node in this case, in order to temporarily preserve the black depth property, we will colour this child node a fictitious "double black" colour.

Before I go further with the "double black" terminology, I'd like justify why the cases above are the only cases we deal with. Firstly, we'll only delete nodes with 0 or 1 child. Neither coloured node can have one black child. IF it did, the black height of the node's null child wouldn't be proper. Also, a red not CAN NOT have a red child. That'd be a red-red violation and if it exists, the given red-black tree is invalid. These observations narrow the cases to the situations listed above.

**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

Now, the remaining cases with this "double black" node can be categorised as follows:

1. The sibling of double-black node is black and has a red child

2. The sibling of double-black node is black and both children are black.

3. The sibling of the double-black node is red.

It should be noted that even though the double black node is a null node, after starting the recolouring/restructuring processes, we may create double black node that isn't null.

To deal with each of the situations, let's set up names for the important nodes we'll be considering most often:

1. Let the child of the deleted node which is coloured "double black" be N.

2. Let Y be the sibling of N.

3. Let Z be child of Y. The specified child will be designated in each case.

4. Let X be the parent of Y.

**Case 1 : Y is black and has a red child Z**

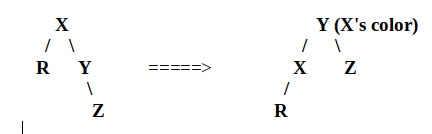
-> This is one of the terminal cases. If the root of tree is a double black node, we simply change its colour to black.

**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

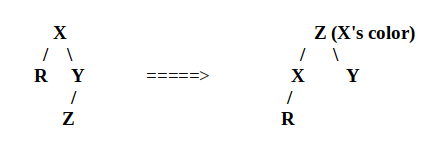
**Case 2 : Y is black and has a red child Z**

-> Here, we take nodes X, Y and Z and relabel them a, b and c respectively in their in-order sequence. We place b  where X used to be, and then have a and c be the left and right children of X respectively. Then we colour a and c black, and colour b whatever the colour X had before. This is also applicable to all the mirror cases.This will eliminate the double black problem, so we stop here. Here's a simple demonstration:

Here, Z is Y's right child



Here, Z is Y's right child

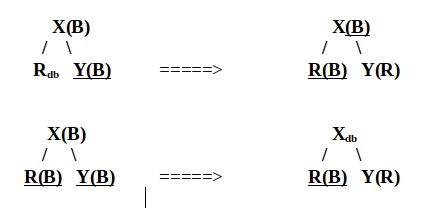


Notice how in each of these situations I've been able to eliminate the double black node, but maintain the "black depth" of each external node in the tree.

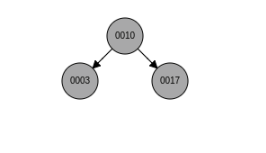
**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Case 3 : Y is black and has 2 black children**

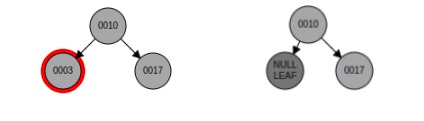
We deal with this case by just recolouring, instead of making any structuring changes to the tree. In particular we'll colour the R node black (changing it from "double black") and then colour Y red. What this does is we subtract one from the black depth of every external node in the sub-tree of X. To compensate for this, we must change X from red to black. But, this ONLY works if X was red to begin with. If it's not, to maintain the "black depth" at the external nodes in the sub-tree rooted at X, we must colour X "double black". If this occurs, we've pushed the "double black" node up the tree.



Let's understand this with a very simple example. Here I want to delete node 3 which has two null children. 



We find this node, delete it and turn it into a double-black null node.



According to case 3 which we're currently studying, in this situation the sibling node is turned into a red node. That is we decrease the count of black nodes in the right sub-tree so that all properties of red-black trees are satisfied.

The root node 10 becomes a double-black node now. And according to case 1, a double-black node at root is converted to black node. The double black node is turned into a single black null node. Here's the result:

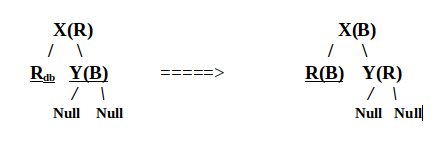


In case 3, we're basically pushing the problem upwards and we start comparing the current case from case 1 onwards.

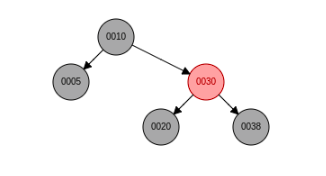
**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

**Case 4 : Y is black, X is red and Y has 2 black children**

This case is applicable when the "double black" node has a red parent and a black sibling with 2 children. To understand it let's look at general picture:

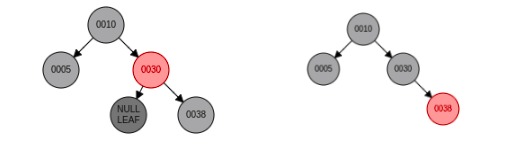


According to this, we turn the double-black node to a black node. Change the parent's colour to black if it is red as shown above, and after it turn the sibling's colour from black to red. Let's look at real example to understand this better:



Consider the above tree. Suppose we want to delete 20 from this red-black tree. We'll first find 20, delete it and then replace it with one of its null children. This deleted node is now a double-black node to compensate the loss of one black node, as I've mentioned in the above sections.

The double black node's parent becomes black and the sibling becomes red. Here's how the red-black tree looks after deletion of node 20:

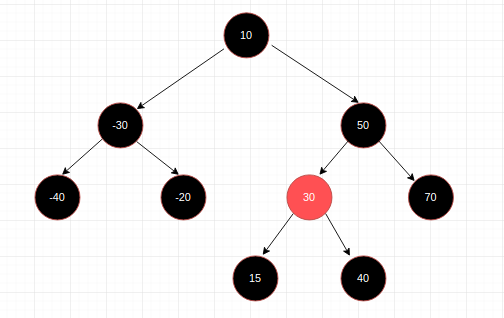


**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

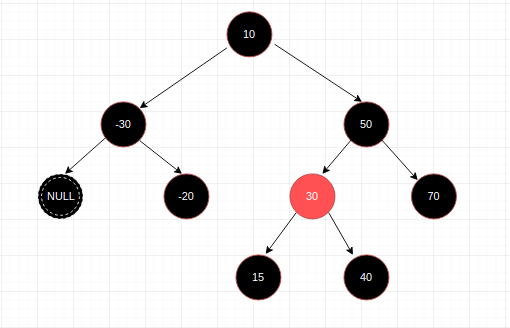
**Case 5 : Y is black, X is black. X's left child is red and right child is black**

Let's just understand this case with an example because that would be much simpler to understand.

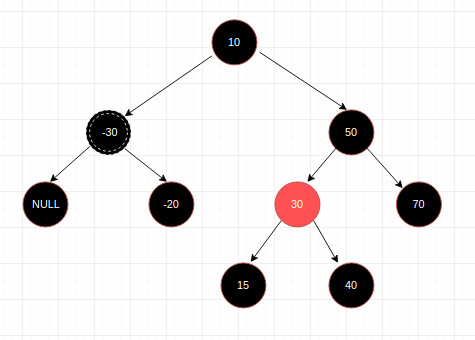
Consider the following red-black tree:



Suppose, we want to delete node -40 from this red-black tree. We find node -40 and delete it. This results in the formation of a double-black node:

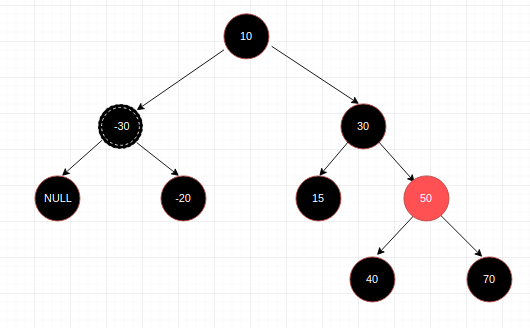


This is an instance of case 3 where the sibling of the double-black node is black with black null children, and their parent is also black. So according to case 3, we simply change the colour of the sibling node to red. Also we push the problem of this double-black node one upwards in this red-black tree. Now -30 will become our double-black node. Here's our current situation:



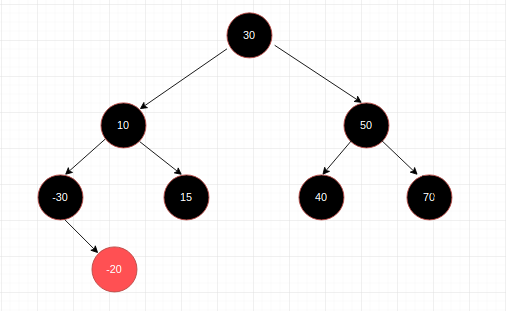
NOW, we finally are onto our case 5 where sibling of -30 (double-black node) is black (50) and, 50's left child is red and right child is black.

So we resolve this we do a right rotation around node 50. After rotation 30 becomes the root of this sub-tree. 50 becomes its right child. 70 is 50's right child (unchanged). 40 which was the right child of 30 becomes the left child of 50. And 15 stays the same with black null children. Finally we swap the colour of node 30 and 50 so that all the properties of red-black trees are preserved.



This condition is applicable to case 6 which is a terminal case (Y is black, X is black/red. X's right child is red and left child can be red/black). Check out case 6 below for more details. The sibling of current double-black node is black with one red child and the colour of other child and parent doesn't matter.

For case 6 we left-rotate the entire tree around its root node. Here's the result:



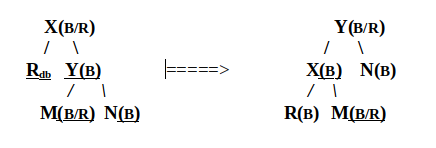
So after doing this transformation, we can see the count of black nodes is same for each path from root to leaf node and there are no red-red violations. So we started with case 3 condition, pushed the problem up and it resulted in case 5. And once we hit case 5, all we had to do was one rotation to find the valid and correct form of red-black tree.

**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

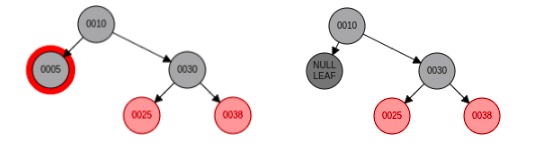
**Case 6 : Y is black, X is black/red. X's right child is red and left child can be red/black**

Here's a graphical representation of this case. The (db) sub-script denotes that it is a "double-black" null node. It has a black node as its sibling. (B/R) denotes that the colour of these nodes can be either black or red. To counter this double-black node, we rotate the tree in an anticlockwise direction and some re-colouring. 

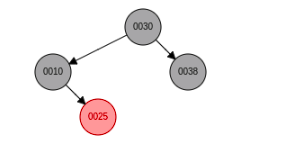
This case is also a terminating case. Once applied, the red-black tree obtained is a valid tree with all of its properties satisfied.



Here's a simple example. Consider the below red-black tree in which we want to delete node 5. This node is first converted to double-black null node:



After this, we do a left rotation to balance this red-black tree. 30 takes the previous root node's (10) colour. 10's colour remains the same, that is, black. And node 38's colour changes from red to black. Here's the final result:



**-------------------------------------------------------------------------------------------------------------------------------------------------------------**

Here's the code in python which performs removal operation for nodes along with insertion:

Code: