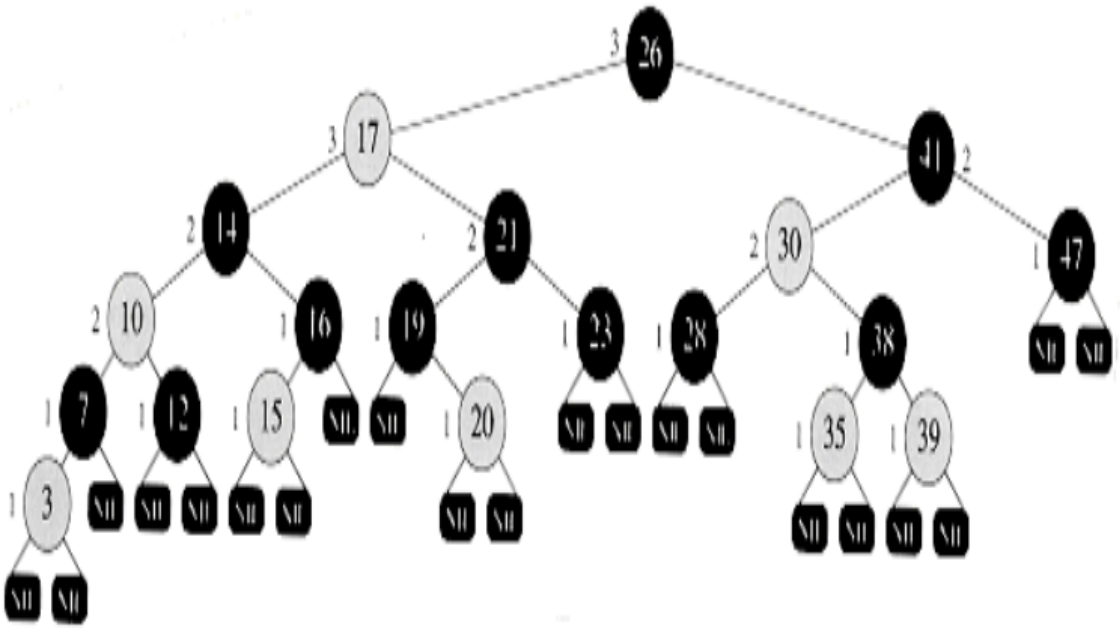
Red Black Tree:

A binary search tree is a red-black tree if it satisfies the following ***red-black properties***:

1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Example:



Red Black Tree

Red black tree is another variant of binary search tree in which every node is colored either red or black we can define a red black tree as follows:

Red black tree is a binary search tree in which every node is colored either red or black.

A red-black tree's node structure would be:

```
struct t_red_black_node {
    enum { red, black } colour;
    void *item;
    struct t_red_black_node *left,
                    *right,
                    *parent;
    }
```

In red black tree the color of node is decided based on the properties of red black tree. Every red black tree has the following properties:

1. Red black tree must be a binary search tree.

2.The root node must be colored black.

3. The children of red color node must be colored black. There should not be two consecutive red nodes.

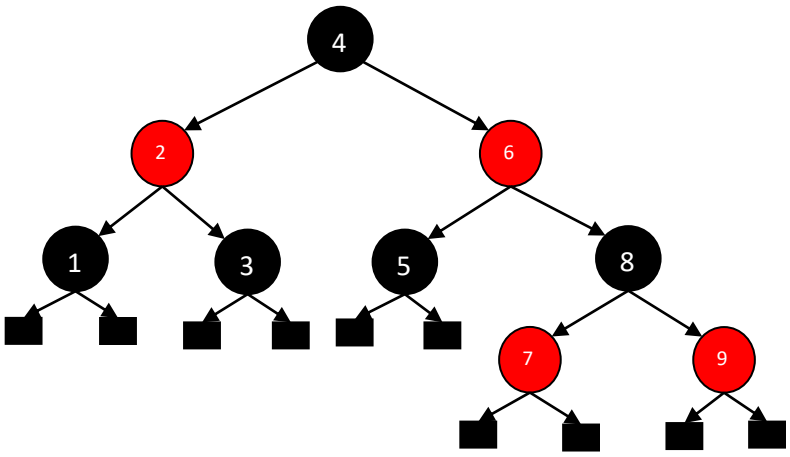
4. In all the paths of the tree there should be same number of black color nodes.

5. Every new node must be inserted with red color.

6. Every leaf(i.e null node) must be colored black.

Example:

Following is a red black tree which is created by inserting number from 1 to 9.



The above tree is a red black tree where every node is satisfying all the properties of red black tree.

Insertion into red black tree:

In a red black tree, every new node must be inserted with the color red. The insertion operation in red black tree is similar to insertion operation in binary search tree. But it is inserted with a color property. After every insertion operation, we need to check all the properties of red black tree. If all the properties are satisfied then we go to next operation otherwise we perform the following operation to make it red black tree.

The operations are

1. Recolor

2. Rotation

3. Rotation followed by recolor.

The insertion operation in red black tree is performed using the following steps:

Step 1: Check whether tree is empty.

Step2: If tree is empty when insert the **newnode** as root node with color black an exit from the operation.

Step3: If tree is not empty then insert the **newnode** as leaf node with color red.

Step4: If the parent of **newnode** is black then exit from the operation.

Step5: If the parent of **newnode** is r red then change the color of parent node's sibling of **newnode**.

Step6: If it is colored black or null then make suitable rotation and recolor it.

Step7: If it is colored red then perform recolor.

Repeat the same until tree becomes red black tree.

Example:

Create a red black tree by inserting following sequence of number :-

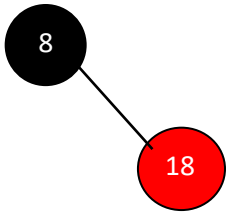
8, 18, 5, 15, 17, 25, 40, and 80

Insert (8)

Tree is empty. So insert newnode as root node with black color.

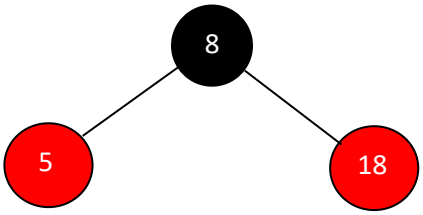
Insert (18)

Tree is not empty. So insert newnode with red color.



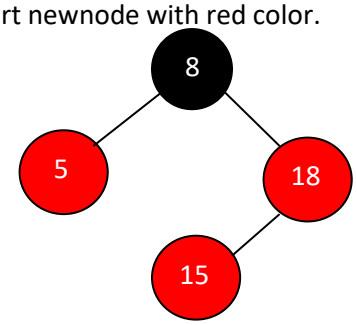
Insert (5)

Tree is not empty. So insert newnode with red color.



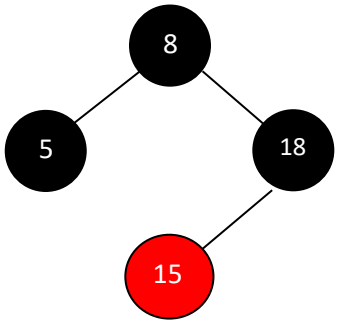
Insert (15)

Tree is not empty. So insert newnode with red color.



Here there are two consecutive red nodes 18 and 15. The newnode's parent sibling color is red and parent's parent is root node. So we use recolor to make it red black tree.

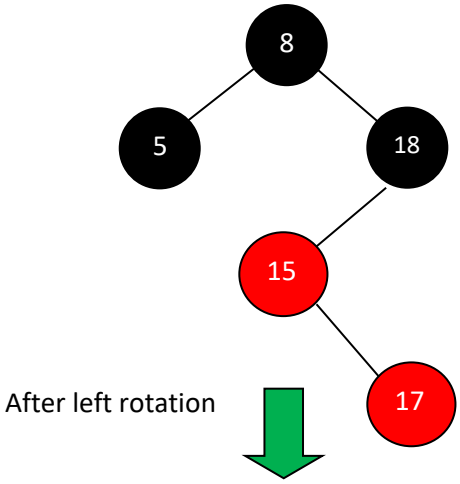
After recolor



After recolor operation, the trees satisfying all red black tree properties.

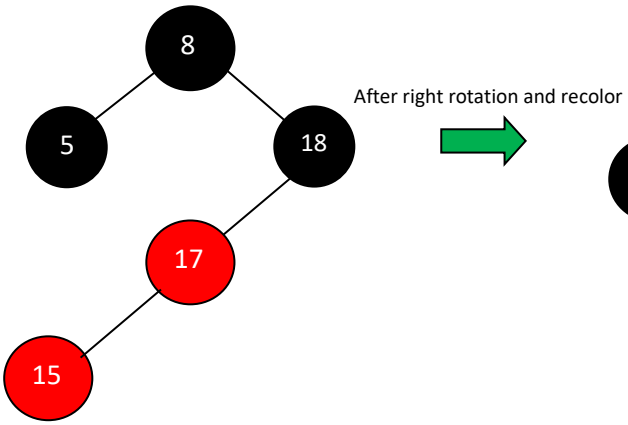
Insert (17)

The tree is not empty. So insert newnode with red color.

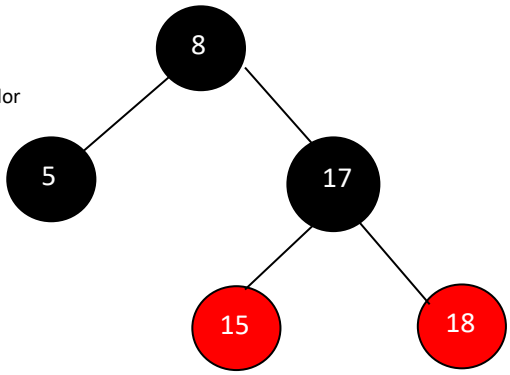


Here there two consecutive red nodes 15 and 17.The newnode's parent sibling is null. So we need rotation. Here we need LR rotation and recolor.

After left rotation

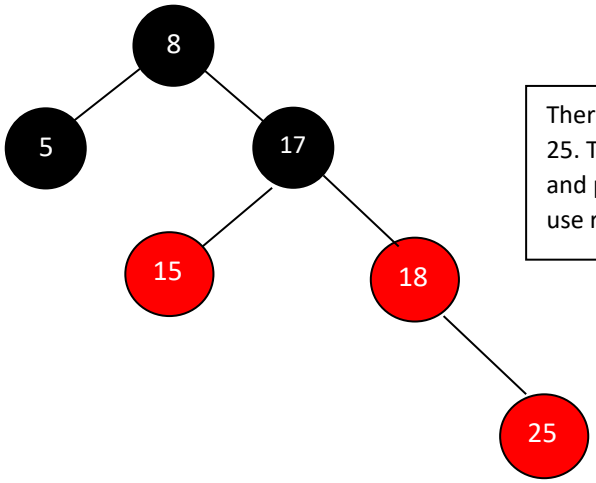


After right rotation and recolor



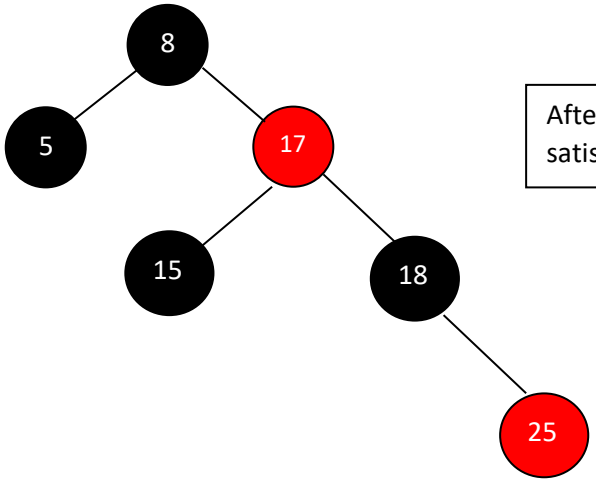
Insert (25)

Tree is not empty. So insert newnode with red color.



There are two consecutive red nodes 18 and 25. The newnode's parent sibling color is red and parent's parent is not root node. So we use recolor and recheck.

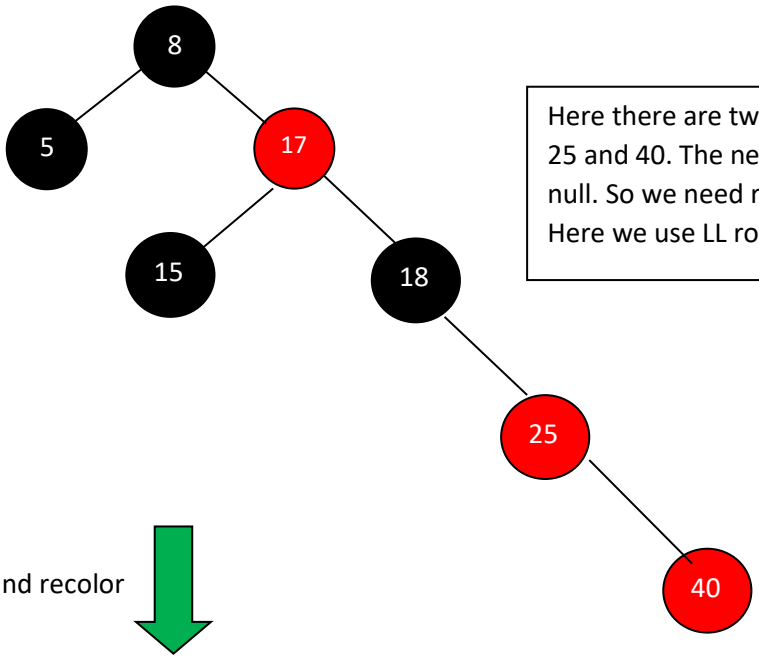
After recolor



After recolor operation, the tree is satisfying all red black tree properties.

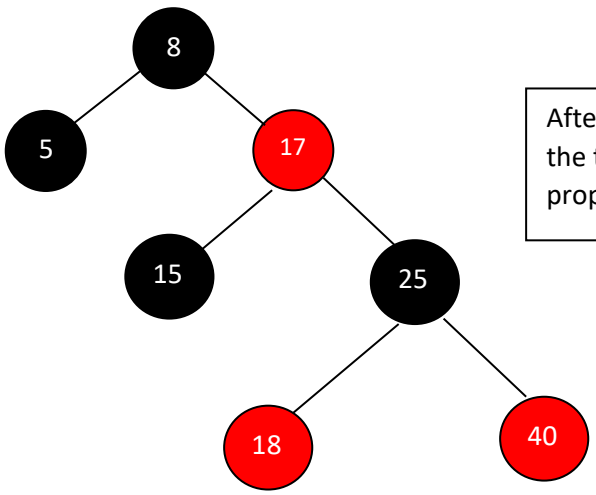
Insert(40)

Tree is not empty. So insert newnode with red color.



Here there are two consecutive red nodes 25 and 40. The newnode's parent sibling is null. So we need rotation and recolor. Here we use LL rotation and recheck.

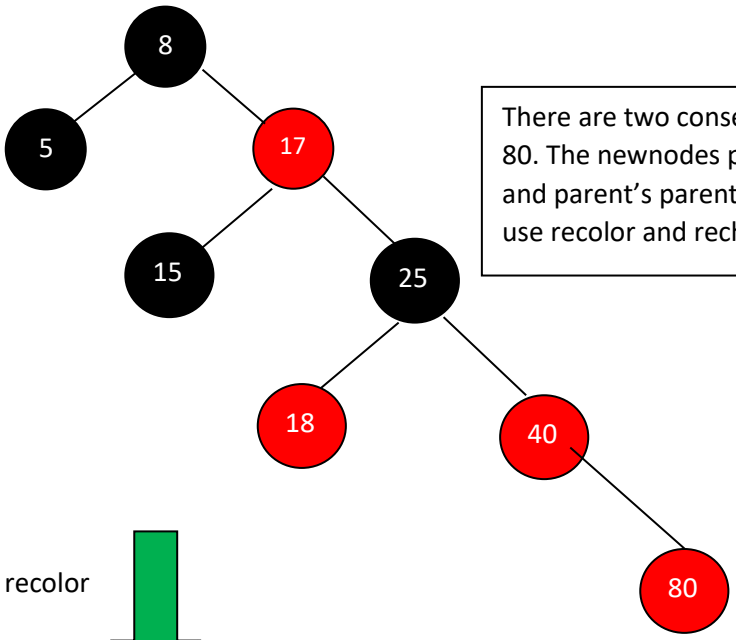
After LL rotation and recolor



After LL rotation and recolor operation, the tree satisfying all red black tree properties.

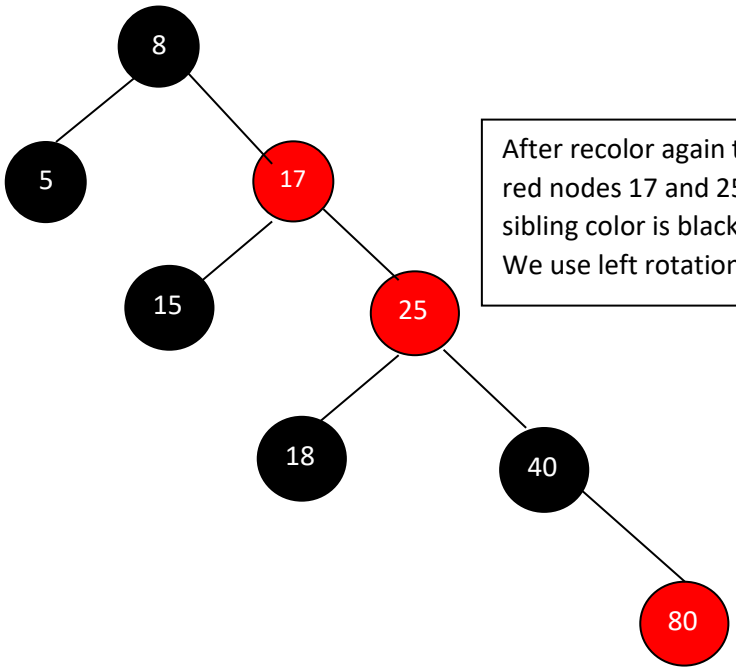
Insert(80)

Tree is not empty. So insert newnode with red color.



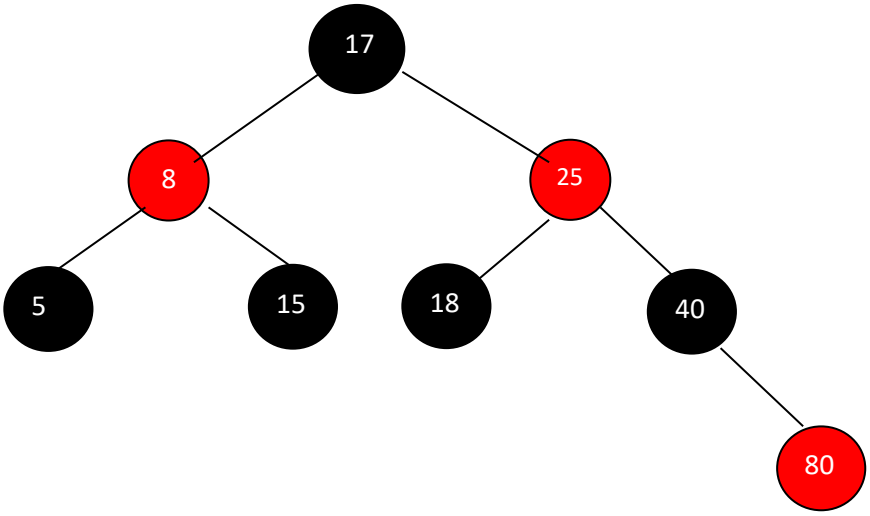
There are two consecutive red nodes 40 and 80. The newnodes parent sibling color is red and parent's parent is not root node. So we use recolor and recheck.

After recolor



After recolor again there are two consecutive red nodes 17 and 25. The newnode's parent sibling color is black. So we need rotation. We use left rotation And recolor.

After left rotation And recolor



Finally above tree is satisfying all the properties of red black tree and it is a perfect red black tree.

Deletion operation in red black tree

The deletion operation in red black tree is similar to deletion operation in BST. But after every deletion operation we need to check with the red black tree properties. If any of the properties violated then make suitable operations like recolor, rotation and rotation followed by recolor to make it red black tree.

Operations on Red Black Tree in details:

Rotations:

A rotation is a local operation in a search tree that preserves *in-order* traversal key ordering.

Note that in both trees, an in-order traversal yields:

A x B y C

The left_rotate operation may be encoded:

```
left_rotate( Tree T, node x ) {
    node y;
    y = x->right;
    /* Turn y's left sub-tree into x's right sub-tree */
    x->right = y->left;
    if ( y->left != NULL )
        y->left->parent = x;
    /* y's new parent was x's parent */
    y->parent = x->parent;
```

```
/* Set the parent to point to y instead of x */
/* First see whether we're at the root */
if ( x->parent == NULL ) T->root = y;
else
    if ( x == (x->parent)->left )
        /* x was on the left of its parent */
        x->parent->left = y;
    else
        /* x must have been on the right */
        x->parent->right = y;
/* Finally, put x on y's left */
y->left = x;
x->parent = y;
}
```

Insertion:

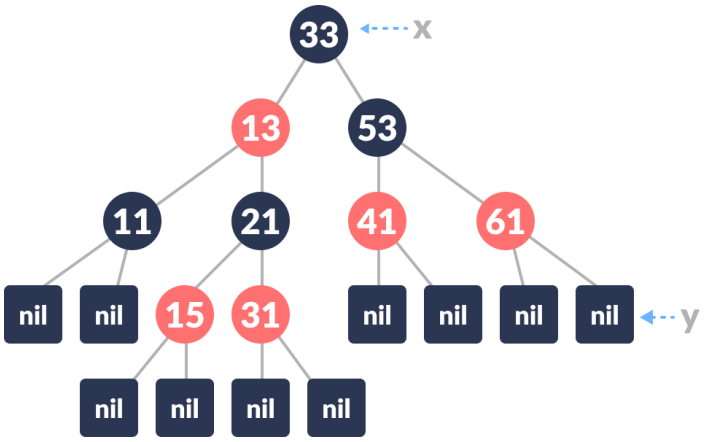
Insertion is somewhat complex and involves a number of cases. Note that we start by inserting the new node, `x`, in the tree just as we would for any other binary tree, using the `tree_insert` function. This new node is labelled red, and possibly destroys the red-black property. The main loop moves up the tree, restoring the red-black property.

Algorithm to Insert a New Node

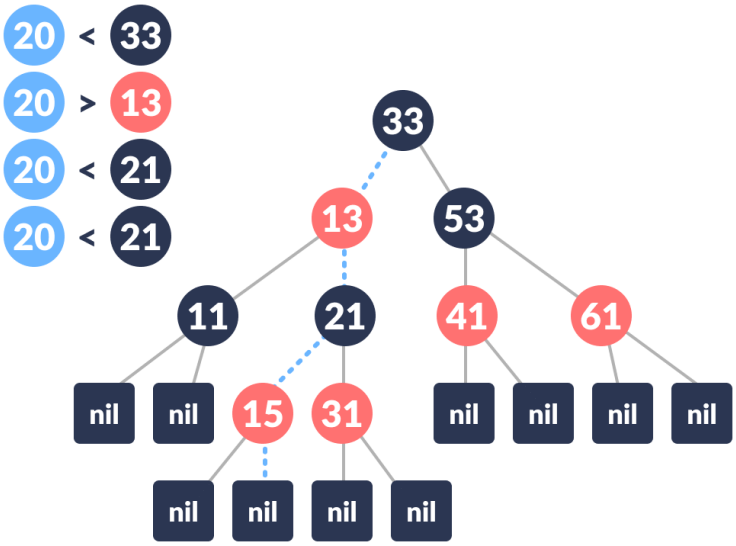
Following steps are followed for inserting a new element into a red-black tree:

1. The `newNode` be:

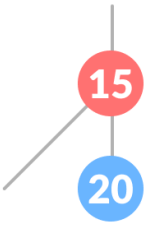
2. Let y be the leaf (ie. `NIL`) and `x` be the root of the tree. The new node is inserted in the following tree.



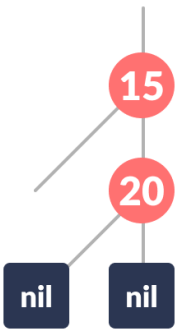
3. Check if the tree is empty (ie. whether `x` is `NIL`). If yes, insert `newNode` as a root node and color it black.
4. Else, repeat steps following steps until leaf (`NIL`) is reached.
    a. Compare `newKey` with `rootKey`.
    b. If `newKey` is greater than `rootKey`, traverse through the right subtree.
    c. Else traverse through the left subtree.



5. Assign the parent of the leaf as parent of `newNode`.
6. If `leafKey` is greater than `newKey`, make `newNode` as `rightChild`.
7. Else, make `newNode` as `leftChild`.



8. Assign `NULL` to the left and `rightChild` of `newNode`.
9. Assign RED color to `newNode`.



10. Call InsertFix-algorithm to maintain the property of red-black tree if violated.

Why newly inserted nodes are always red in a red-black tree?

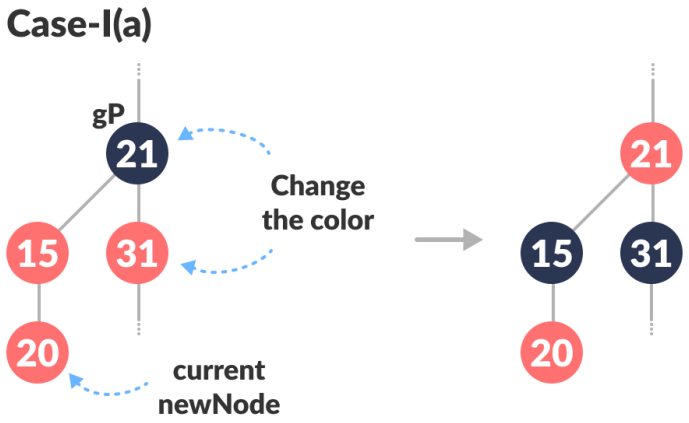
This is because inserting a red node does not violate the depth property of a red-black tree.

If you attach a red node to a red node, then the rule is violated but it is easier to fix this problem than the problem introduced by violating the depth property.

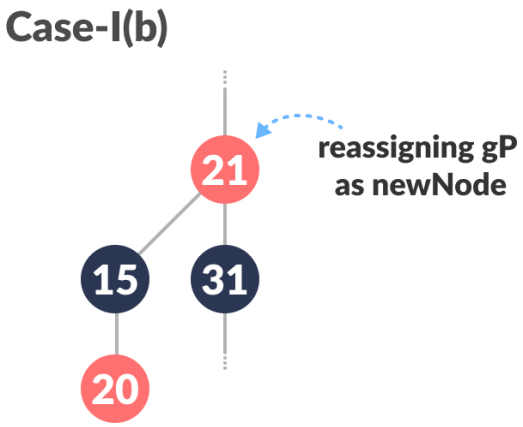
Algorithm to Maintain Red-Black Property After Insertion

This algorithm is used for maintaining the property of a red-black tree if insertion of a newNode violates this property.

1. Do the following until the parent of `newNode` `p` is RED.
2. If `p` is the left child of `grandParent` `gP` of `newNode`, do the following.
   Case-I:
   a. If the color of the right child of `gP` of `newNode` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.

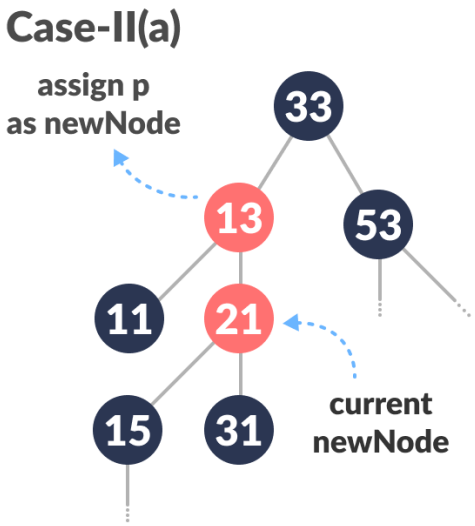


b. Assign `gP` to `newNode`.

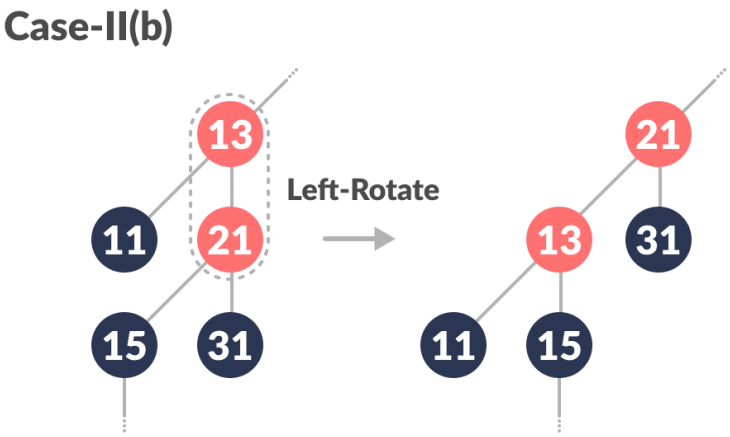


Case-II:

c. (Before moving on to this step, while loop is checked. If conditions are not satisfied, it the loop is broken.)
Else if `newNode` is the right child of `p` then, assign `p` to `newNode`.

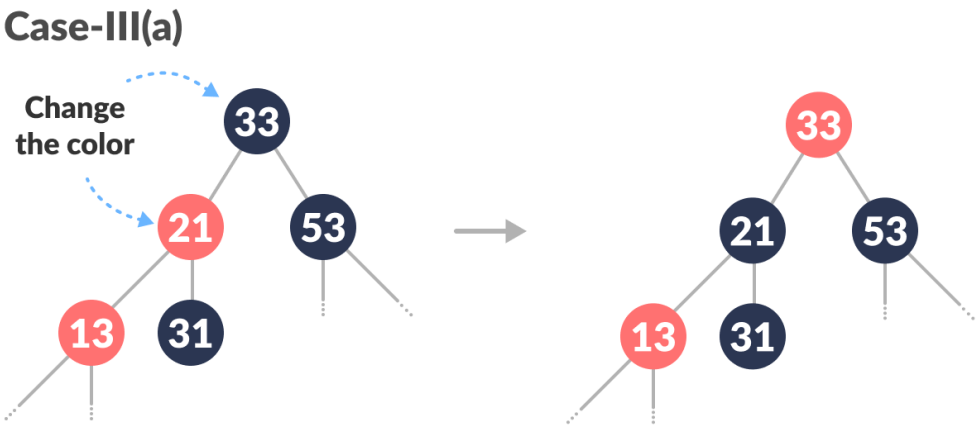


d. Left-Rotate `newNode`.

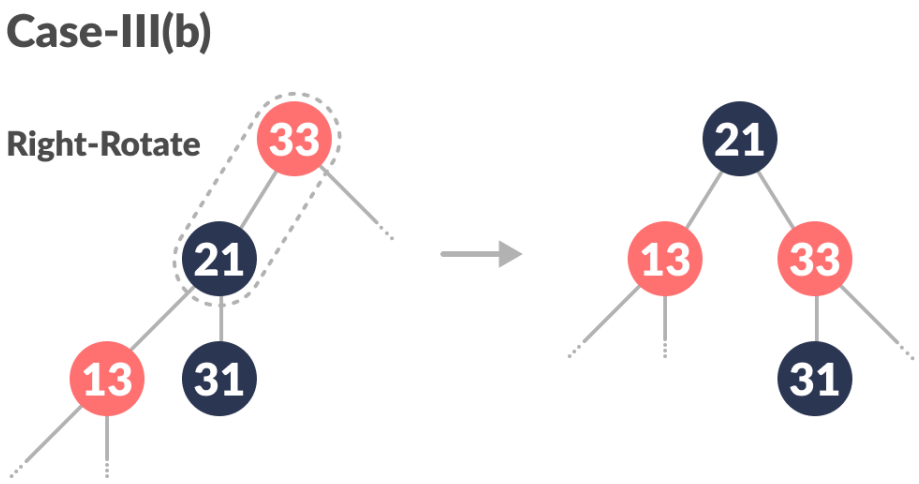


Case-III:

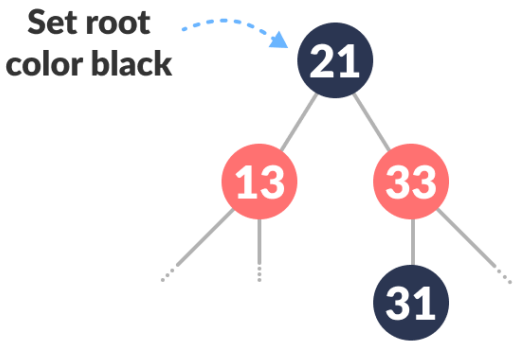
e. (Before moving on to this step, while loop is checked. If conditions are not satisfied, it the loop is broken.)
Set color of `p` as BLACK and color of `gP` as RED.



f. Right-Rotate `gP`.

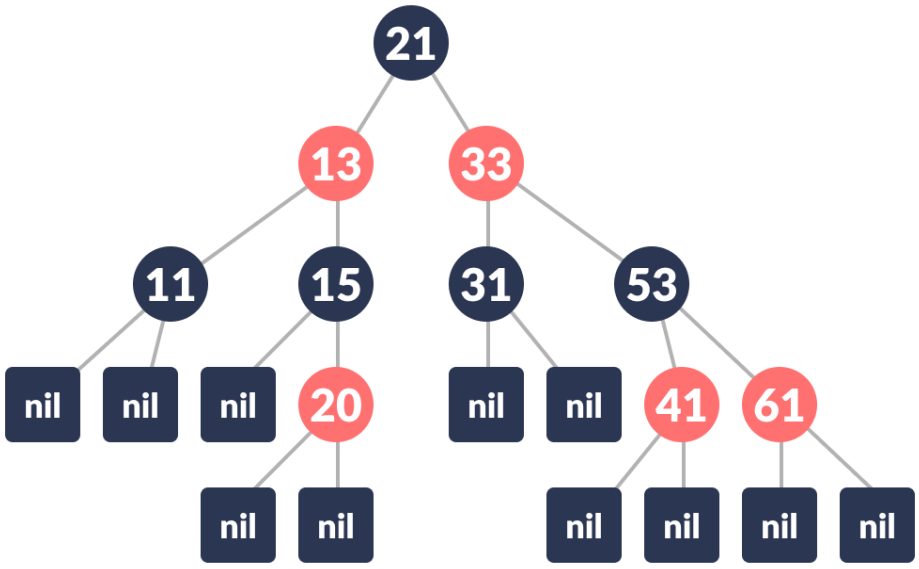


3. Else, do the following.
    a. If the color of the left child of `gP` of `z` is RED, set the color of both the children of `gP` as BLACK and the color of `gP` as RED.
    b. Assign `gP` to `newNode`.
    c. Else if `newNode` is the left child of `p` then, assign `p` to `newNode` and Right-Rotate `newNode`.
    d. Set color of `p` as BLACK and color of `gP` as RED.
    e. Left-Rotate `gP`.
4. (This step is perfomed after coming out of the while loop.)
   Set the root of the tree as BLACK.



The final tree look like this:

Final Tree



The insertion operation is encoded as:

```
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->colour = red;
    while ( (x != T->root) && (x->parent->colour == red) ) {
       if ( x->parent == x->parent->parent->left ) {
           /* If x's parent is a left, y is x's right 'uncle' */
           y = x->parent->parent->right;
           if ( y->colour == red ) {
               /* case 1 - change the colours */
               x->parent->colour = black;
               y->colour = black;
               x->parent->parent->colour = red;
               /* Move x up the tree */
               x = x->parent->parent;
               }
           else {
               /* y is a black node */
               if ( x == x->parent->right ) {
                   /* and x is to the right */
                   /* case 2 - move x up and rotate */
                   x = x->parent;
                   left_rotate( T, x );
                   }
```

```
            /* case 3 */
            x->parent->colour = black;
            x->parent->parent->colour = red;
            right_rotate( T, x->parent->parent );
            }
        }
    else {
        /* repeat the "if" part with right and left
           exchanged */
        }
    }
/* Colour the root black */
T->root->colour = black;
}
```

Examination of the code reveals only one loop. In that loop, the node at the root of the sub-tree whose red-black property we are trying to restore, x, may be moved up the tree *at least one level* in each iteration of the loop. Since the tree originally has **O(log n)** height, there are **O(log n)** iterations. The tree_insert routine also has **O(log n)** complexity, so overall the rb_insert routine also has **O(log n)** complexity.

Red-black trees:

Trees which remain **balanced** - and thus guarantee **O(logn)** search times - in a dynamic environment. Or more importantly, since any tree can be re-balanced - but at considerable cost - can be re-balanced in **O(logn)** time.

Time complexity in big O notation

Algorithm	Average	Worst case
Space	O(*n*)	O(*n*)
Search	O(log *n*)[1]	O(log *n*)[1]
Insert	O(log *n*)[1]	O(log *n*)[1]
Delete	O(log *n*)[1]	O(log *n*)[1]

Applications:

Red black tree offer worst case guarantee for insertion time, deletion time and search time. Not only does this make them valuable in time sensitive applications such as real time applications but it makes them valuable building blocks in other data structures which provide worst case guarantees.

1. Most of the self-balancing BST library functions like map and set in C++ (OR TreeSet and TreeMap in Java) use Red Black Tree
2. It is used to implement CPU Scheduling Linux. Completely Fair Scheduler uses it.