Red-Black Tree

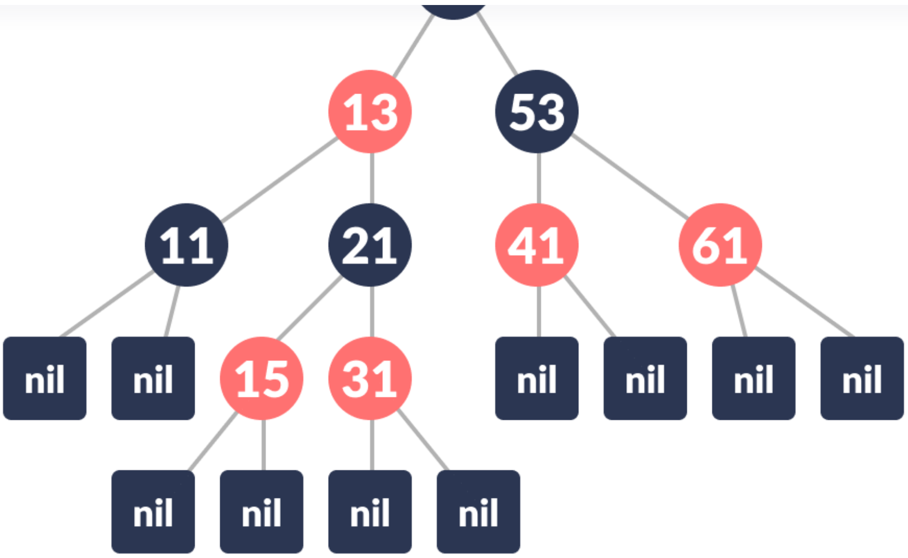
In this tutorial, you will learn what a red-black tree is. Also, you will find working examples of various operations performed on a red-black tree in C, C++, Java and Python.

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property:** The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** For each node, any simple path from this node to any of its descendant leaf has the same black-depth (the number of black nodes).

An example of a red-black tree is:



Each node has the following attributes:

- color
- key
- leftChild
- rightChild
- parent (except root node)

How the red-black tree maintains the property of self-balancing?

The red-black color is meant for balancing the tree.

The limitations put on the node colors ensure that any simple path from the root to a leaf is not more than twice as long as any other such path. It helps in maintaining the self-balancing property of the red-black tree.

Operations on a Red-Black Tree

Rotating the subtrees in a Red-Black Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

Rotation operation is used for maintaining the properties of a red-black tree when they are violated by other operations such as insertion and deletion.

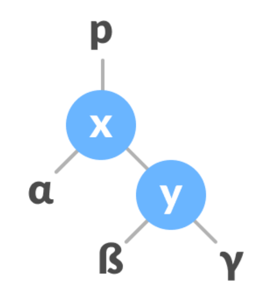
There are two types of rotations:

Left Rotate

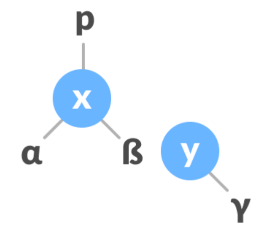
In left-rotation, the arrangement of the nodes on the right is transformed into the arrangements on the left node.

Algorithm

1. Let the initial tree be:

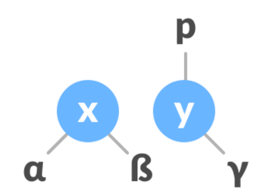


2. If `y` has a left subtree, assign `x` as the parent of the left subtree of `y`.

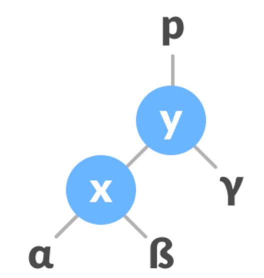


3. If the parent of `x` `p` is `NULL`, make `y` as the root of the tree.

5. Else assign `y` as the right child of `p`.



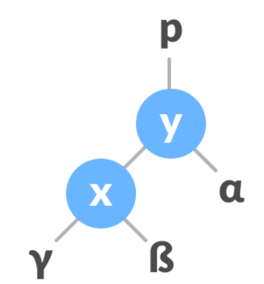
6. Make `y` as the parent of `x`.



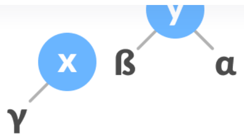
Right Rotate

In right-rotation, the arrangement of the nodes on the left is transformed into the arrangements on the right node.

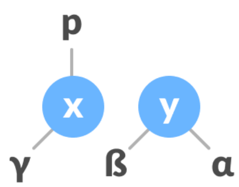
1. Let the initial tree be:



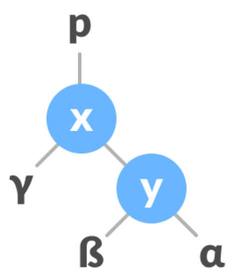
2. If `x` has a right subtree, assign y as the parent of the right subtree of `x`.



3. If the parent of `y` is `NULL`, make `x` as the root of the tree.
4. Else if `y` is the right child of its parent `p`, make `x` as the right child of `p`.
5. Else assign `x` as the left child of `p`.



6. Make `x` as the parent of `y`.

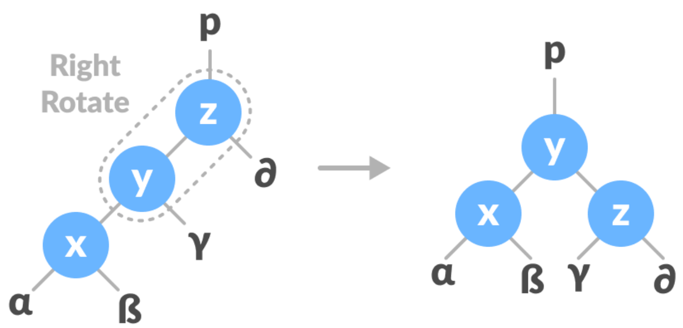


Left-Right and Right-Left Rotate

In left-right rotation, the arrangements are first shifted to the left and then to the right.

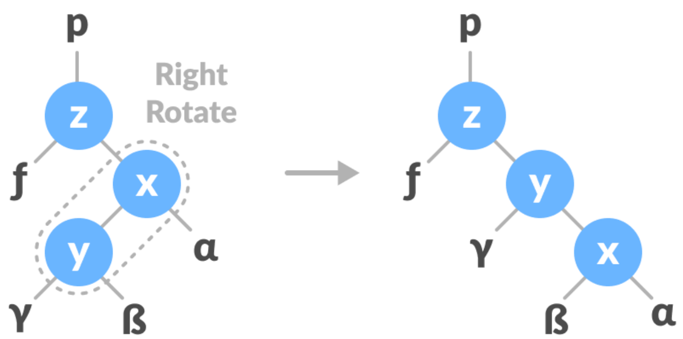
1. Do left rotation on x-y.

2. Do right rotation on y-z.

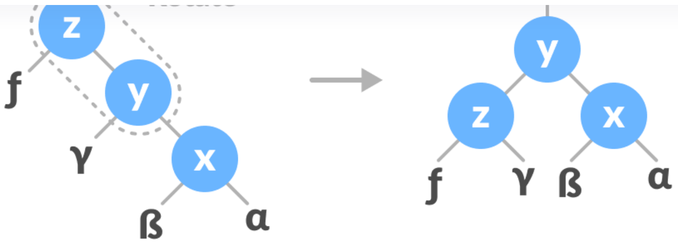


In right-left rotation, the arrangements are first shifted to the right and then to the left.

1. Do right rotation on x-y.



2. Do left rotation on z-y.



Inserting an element into a Red-Black Tree

While inserting a new node, the new node is always inserted as a RED node. After insertion of a new node, if the tree is violating the properties of the red-black tree then, we do the following operations.

1. Recolor
2. Rotation

Algorithm to insert a node

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

1. Let y be the leaf (ie. `NIL`) and x be the root of the tree.
2. Check if the tree is empty (ie. whether `x` is `NIL`). If yes, insert `newNode` as a root node and color it black.
3. 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.
4. Assign the parent of the leaf as parent of `newNode`.
5. If `leafKey` is greater than `newKey`, make `newNode` as `rightChild`.

7. Assign `NULL` to the left and `rightChild` of `newNode`.
8. Assign RED color to `newNode`.
9. 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 `z`, do the following.
   Case-I:
   a. If the color of the right 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`.
   Case-II:
   c. Else if `newNode` is the right child of `p` then, assign `p` to `newNode`.
   d. Left-Rotate `newNode`.
   Case-III:
   e. Set color of `p` as BLACK and color of `gP` as RED.

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. Set the root of the tree as BLACK.

Deleting an element from a Red-Black Tree

This operation removes a node from the tree. After deleting a node, the red-black property is maintained again.

Algorithm to delete a node

1. Save the color of `nodeToBeDeleted` in `origrinalColor`.
2. If the left child of `nodeToBeDeleted` is `NULL`
   a. Assign the right child of `nodeToBeDeleted` to x.
   b. Transplant `nodeToBeDeleted` with `x`.
3. Else if the right child of `nodeToBeDeleted` is `NULL`
   a. Assign the left child of `nodeToBeDeleted` into `x`.
   b. Transplant `nodeToBeDeleted` with `x`.
4. Else
   a. Assign the minimum of right subtree of `noteToBeDeleted` into `y`.

c. Assign the `rightChild` of `y` into `x`.

d. If `y` is a child of `nodeToBeDeleted`, then set the parent of `x` as `y`.

e. Else, transplant `y` with `rightChild` of `y`.

f. Transplant `nodeToBeDeleted` with `y`.

g. Set the color of y with originalColor.

5. If the `originalColor` is BLACK, call DeleteFix(x).

Algorithm to maintain Red-Black property after deletion

This algorithm is implemented when a black node is deleted because it violates the black depth property of the red-black tree.

This violation is corrected by assuming that node `x` (which is occupying `y`'s original position) has an extra black. This makes node `x` neither red nor black. It is either doubly black or black-and-red. This violates the red-black properties.

However, the color attribute of `x` is not changed rather the extra black is represented in `x`'s pointing to the node.

The extra black can be removed if

1. It reaches the root node.
2. If `x` points to a red-black node. In this case, `x` is colored black.
3. Suitable rotations and recolorings are performed.

Following algorithm retains the properties of a red-black tree.

1. Do the following until the `x` is not the root of the tree and the color of `x` is BLACK
2. If `x` is the left child of its parent then,

   a. Assign w to the sibling of x.

a. Set the color of the right child of the parent of `x` as BLACK.

b. Set the color of the parent of `x` as RED.

c. Left-Rotate the parent of `x`.

d. Assign the `rightChild` of the parent of `x` to `w`.

c. If the color of both the right and the `leftChild` of `w` is BLACK,

Case-II:

a. Set the color of `w` as RED

b. Assign the parent of `x` to `x`.

d. Else if the color of the `rightChild` of `w` is BLACK

Case-III:

a. Set the color of the `leftChild` of `w` as BLACK

b. Set the color of `w` as RED

c. Right-Rotate `w`.

d. Assign the `rightChild` of the parent of `x` to `w.`

e. If any of the above cases do not occur, then do the following.

Case-IV:

a. Set the color of `w` as the color of the parent of `x`.

b. Set the color of the parent of parent of `x` as BLACK.

c. Set the color of the right child of `w` as BLACK.

d. Left-Rotate the parent of `x`.

e. Set `x` as the root of the tree.

3. Else the same as above with right changed to left and vice versa.

4. Set the color of `x` as BLACK.

examples.

Python, Java and C/C++ Examples

Python Java C C++

```
# Implementing Red-Black Tree in Python

import sys

class Node():
    def __init__(self, data):
        self.data = data
        self.parent = None
        self.left = None
        self.right = None
        self.color = 1

class RedBlackTree():
    def __init__(self):
        self.TNULL = Node(0)
        self.TNULL.color = 0
        self.TNULL.left = None
        self.TNULL.right = None
        self.root = self.TNULL
```

Red-Black Tree Applications

1. To implement finite maps
2. To implement Java packages: `java.util.TreeMap` and `java.util.TreeSet`
3. To implement Standard Template Libraries (STL) in C++: multiset, map, multimap
4. In Linux Kernel

Next Tutorial:
Red-Black Tree Insertion (/dsa/insertion-in-a-red-black-tree)

Previous Tutorial:
(/dsa/binary-search-tree)

(/dsa) > Red-Black Tree

Related Tutorials

DS & Algorithms

Greedy Algorithm

(/dsa/greedy-algorithm)

DS & Algorithms

Types of Queue

(/dsa/types-of-queue)

DS & Algorithms

Deque

(/dsa/deque)

DS & Algorithms

Hashing

(/dsa/hashing)