Red-Black Tree

By Mani Abedii

View Full DSA Repository

A clear, compact guide covering essential data structures and algorithms, designed for easy understanding and quick learning.

Red-Black Trees are a type of self-balancing binary search tree. They are widely used in computer science because they maintain balance dynamically, ensuring efficient operations such as insertion, deletion, and lookup.

A Red-Black Tree has a special color-based structure that makes it efficient. Each node in the tree is either **red** or **black**, and these colors help the tree stay balanced automatically.

Thanks to this unique structure, Red-Black Tree guarantees logarithmic time complexity O(log n) for search, insert & delete operations.

The AVL trees are more balanced compared to Red-Black Trees, but they may cause more rotations during insertion and deletion. So if your application involves frequent insertions and deletions, then Red-Black trees should be preferred.

Key Characteristics of Red-Black Trees:

1. **Node Colors:** Each node is either red or black.
2. **Root:** The root of the tree is always black.
3. **NIL Nodes:** A nil (null) node is an always-black placeholder node for a missing child.
4. **Black Height:** the black height of the red-black tree is the number of black nodes on a path from the root node to a nil node.
5. A red–black tree with n nodes has height $h \leq 2\lg(n+1)$ because its black height is at most and the total height can be at most twice the black height.

6. **Red Rule:**

   A red node cannot have a red child (no two consecutive red nodes).

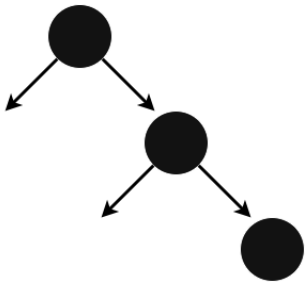
7. **Black Height Property:**

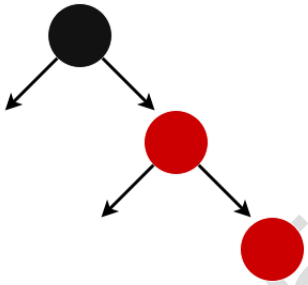
   The black height is the same for all paths. This property ensures that the longest path from the root to any nil is no more than twice as long as the shortest path, maintaining the tree's balance and efficient performance.

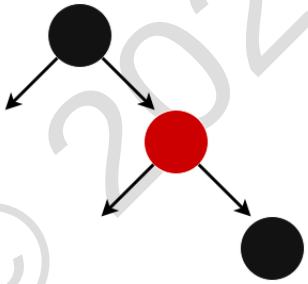
How do these properties ensure balance?

A simple example to understand balancing is, that a **chain** of 3 nodes is not possible in the Red-Black tree. We can try any combination of colors and see if all of them violate the Red-Black tree property.

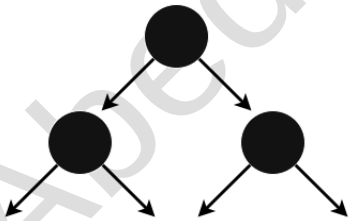
Not possible:

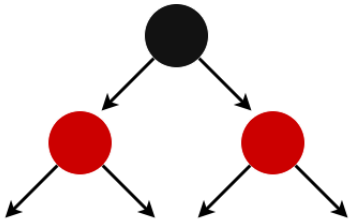






Possible:





Key Operations & Methods:

1. **Rotations:**

   Rotations are fundamental operations in maintaining the balanced structure of a Red-Black Tree. They help to preserve the properties of the tree, ensuring that the longest path from the root to any leaf is no more than twice the length of the shortest path.

   Left Rotation: A left rotation at node x, moves x down to the left and its right child y up to take x's place.

   Right Rotation: A right rotation at node x moves x down to the right and its left child y up to take x's place.

   Rotations in Red-Black Trees are typically performed during insertions and deletions to maintain the properties of the tree.

2. **Insertion:**

   Let's call the new node x.

   First, perform a standard BST insertion & assign a red color to x.
   When a new node is inserted, it can create violations to Red-Black Tree properties because:

   - The root must be black.
   - Red nodes cannot have red children.

   In AVL tree insertion, we only used rotation as a tool to do balancing after insertion. In the Red-Black tree, we also use **recoloring** to do the balancing:

   Fix algorithms have mainly two cases depending upon the color of the **uncle**. If the uncle is red, we do recolor, but if the uncle is black, we do rotations and/or recoloring.

After standard BST insertion, the newly inserted node "x" is colored red:

1. If **x is the root**, change the color of x as black**. If x is not the root & the color of x's parent is black**, no adjustments are needed because inserting a red node as the child of a black node won't violate the "no consecutive red nodes" rule, and also won't change the number of black nodes.

2. **If x is not the root and the color of x's parent is not black**:
    a. **If x's uncle is RED** → Grandparent must have been black, otherwise uncle couldn't have been red so:
        i. Change the color of parent & uncle as Black.
        ii. Change the color of the grandparent as Red.
        iii. Repeat the same process for the grandparent.

        Ensure the root stays black.

    b. **If x's uncle is BLACK (null also counts as black)** → then there can be 4 configurations for x, parent & grandparent (Similar to AVL Tree):

        i. Left-Left (LL) Case

            parent is the left child of the grandparent & x is the left child of the parent.

            Fix: Perform a single right rotation on the grandparent.

        ii. RR Case:

            Fix: Perform a single left rotation on the grandparent.

        iii. LR Case:

            Fix: Perform a left rotation on the parent, then a right rotation on the grandparent.

        iv. RL Case

            Fix: Perform a right rotation on the parent, then a left rotation on the grandparent.

    For (i) & (ii), swap the colors of grandparent & parent after rotations.

    For (iii) & (iv), swap color of grandparent & x after rotations.

3. Deletion:

Deletion in a red-black tree is a bit more complicated than insertion. When a node is to be deleted it can have no children, one child or two children.

After the node is deleted, the red-black properties might be violated. To restore these properties, some color changes and rotations are performed on the node in the tree. The changes are similar to those performed during insertion, but with different conditions.

In the insert operation, we check the color of the uncle to decide the appropriate case. In the delete operation, we check the color of the **sibling** to decide the appropriate case.

The main property that violates after insertion is **two consecutive reds**. In delete, the main violated property is, **change of black height in subtrees** as deletion of a black node may cause reduced black height in one root to leaf path.

To understand deletion, the notion of double black is used. When a black node is deleted and replaced by a black child, the child is marked as **double black**. The main task now becomes to convert this double black to single black.

First, perform a standard BST delete:

When we perform a standard delete operation in BST, we always end up deleting a node which is a leaf. So, we somehow move down the node to be deleted, then we delete it.

Let **z** be the node to be deleted (has the deleting value), **y** the node that is actually removed, and **x** the y's child (or NIL node) that moves into y's position and may carry the double-black violation.

1. If y is red, simply delete it; no red–black properties are violated.
2. If y is **black** & x is red, again simply delete y, x will replace it and no properties are violated.
3. If y is **black** & x is **black**, deleting it causes a double black violation at x, which must be fixed using the red–black tree deletion fix-up procedure.

There are 5 possible cases for double black x that need to be handled:

Case 3.1: The double black x is the root of the tree

- Simply ignore it (double black root doesn't violate any red-black tree properties)

Case 3.2: The sibling is red

- Swap the color of the sibling & the parent (recolor the parent to red & the sibling to black)
- Perform a rotation on the parent toward x.
- The new sibling is always black, so it falls into one of the cases below

Case 3.3: The sibling is black & its children are also black

- Recolor the sibling to red.
- Give the extra black to the parent:
    - If the parent is red, just make it black.
    - If the parent is already black, it will become double black → Fix the parent (reapply cases).

Case 3.4: The sibling is black, its farther child to x is black & the closer child is red

- Swap the color of the sibling & its closer child (recolor the sibling to red & its closer child to black)
- Perform a rotation on the sibling away from x.
- Now this case falls into case (4)

Case 3.5: The sibling is black & its farther child to x is red

- Recolor sibling to parent's color
- Recolor parent to black
- Recolor sibling's far child to black
- Perform a rotation on the parent toward x,