Red Black Tree

We've already explored Binary Search Trees (BSTs) and their self-balancing cousins, AVL Trees. You know that regular BSTs can become skewed, leading to O(N) worst-case performance, and AVL trees solve this by strictly maintaining a balance factor of -1, 0, or 1 using rotations.

Now, let's meet another self-balancing superstar: the **Red-Black Tree**!

⚫🔴 Red-Black Trees: A Different Approach to Balance!

A **Red-Black Tree** is another type of **self-balancing Binary Search Tree**. Like AVL trees, it guarantees O(logN) time complexity for search, insert, and delete operations in the worst case. However, instead of strictly enforcing height balance with a balance factor, Red-Black Trees maintain balance by enforcing a set of **coloring rules** on its nodes.

The "balance" in a Red-Black Tree isn't about height differences being at most 1. Instead, it's about ensuring that the longest path from the root to any leaf is no more than twice as long as the shortest path. This looser balance criterion means Red-Black Trees perform fewer rotations than AVL trees on average, making them a good choice for applications with frequent insertions and deletions.

The Core Idea: Coloring Rules

Every node in a Red-Black Tree is colored either **Red** or **Black**. These colors, combined with five strict properties, ensure the tree remains approximately balanced.

📜 The Five Fundamental Properties of a Red-Black Tree

For a binary search tree to be a valid Red-Black Tree, it *must* satisfy these five properties:

1. **Node Color Property:** Every node is either **Red** or **Black**.
2. **Root Property:** The **root** of the tree is always **Black**.
3. **Red-Child Property (No Double Reds):** If a node is **Red**, then both of its children must be **Black**. (This prevents consecutive Red nodes along any path from the root to a leaf).
4. **Black-Height Property:** For every node, all simple paths from the node to descendant leaf nodes contain the **same number of Black nodes**. (This is the property that guarantees balance).
5. **Null Leaf Property:** All null leaves (often represented as special "NIL" nodes or

simply null pointers) are considered **Black**.

If any of these properties are violated after an insertion or deletion, the tree must be rebalanced through a combination of **recoloring** and **rotations**.

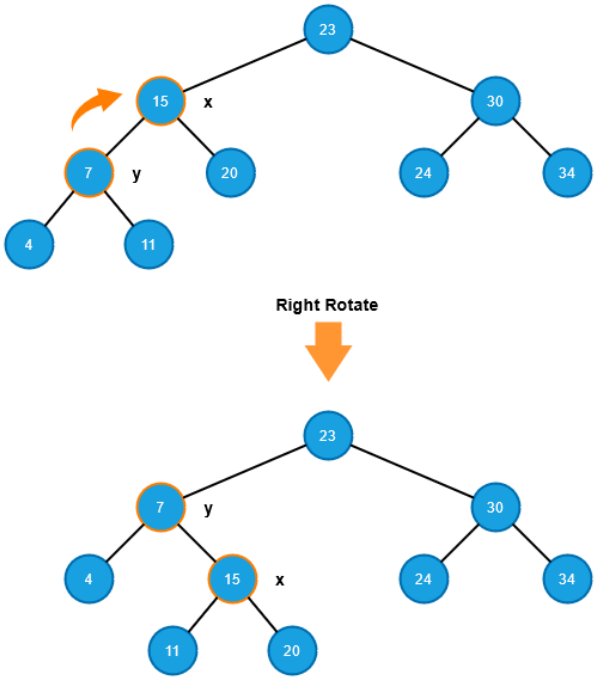
🔄 The Magic of Rotations (Revisited)

Rotations in Red-Black Trees are the same fundamental operations as in AVL trees: **Left Rotation** and **Right Rotation**. They rearrange nodes to restore the BST property and help satisfy the Red-Black properties, particularly the Black-Height property.

Let's denote the node where the rotation is performed as X or Y, and T1, T2, T3 are subtrees.

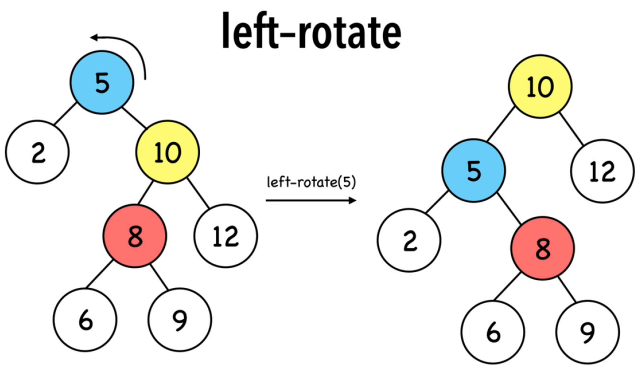
1. Right Rotation

- Used to fix a "left-heavy" imbalance.
    - The parent (Y) moves down to become the right child of its left child (X).



2. Left Rotation

- Used to fix a "right-heavy" imbalance.
    - The parent (X) moves down to become the left child of its right child (Y).



🚀 Basic Operations in a Red-Black Tree (with Balancing)

The core operations (insert, delete, search) are similar to a regular BST, but they require additional logic to maintain the Red-Black properties.

1. insert(data): Adding a New Node

Insertion in a Red-Black Tree is a multi-step process to ensure properties are maintained:

1. **Standard BST Insertion:** Insert the new node N as you would in a normal BST (recursively traverse, go left for smaller, right for larger, insert at null spot).
2. **Color the New Node Red:** Always color the newly inserted node N as **Red**. (Why? Because inserting a Black node might violate the Black-Height property on many paths. Inserting a Red node only potentially violates the Red-Child property.)
3. **Handle Violations (Fix-up):** Now, check for violations of Red-Black properties, especially the **Red-Child Property** (two consecutive Red nodes: parent P and child N are both Red). This is the main focus of the "fix-up" process.
    - **Case 1: N is the Root.** If N is the root, color it **Black** (Property 2). This is the simplest fix.
    - **Case 2: N's Parent P is Black.** No violation of Red-Child property. The tree remains valid.
    - **Case 3: N's Parent P is Red.** This is a violation! Now we look at P's sibling, U (the **uncle** of N).
        - **Subcase 3a: U is Red.** (Parent P is Red, Uncle U is Red).
            - **Recolor:** Change P to Black, U to Black, and their grandparent G to Red.
            - **Recurse:** Now, the grandparent G might have a Red parent, so repeat

the fix-up process starting from G.

- **Subcase 3b: U is Black.** (Parent P is Red, Uncle U is Black).
  - This requires **rotations** and **recoloring**. There are two sub-subcases here (which correspond to LR and RL cases from AVL, but with specific coloring):
    - **LR (Left-Right) Case:** N is a right child, P is a left child. Perform a **Left Rotation** on P (making N the new P), then proceed as if it were an LL case.
    - **RL (Right-Left) Case:** N is a left child, P is a right child. Perform a **Right Rotation** on P (making N the new P), then proceed as if it were an RR case.
    - **LL (Left-Left) Case:** N is a left child, P is a left child. Perform a **Right Rotation** on G. Recolor P to Black, G to Red.
    - **RR (Right-Right) Case:** N is a right child, P is a right child. Perform a **Left Rotation** on G. Recolor P to Black, G to Red.

This fix-up process might involve one or two rotations and some recoloring, ensuring the tree remains balanced.

2. delete(data): Removing a Node

Deletion in Red-Black Trees is significantly more complex than insertion. It aims to remove a node while preserving the five properties.

1. **Standard BST Deletion:** First, find the node to delete. If it has two children, find its inorder successor (or predecessor), copy its data to the node to be deleted, and then delete the successor (which will have at most one child). This means the actual node being removed (X) will always have at most one child.
2. **Identify the "Problem Node":** The node X that is actually removed from the tree. If X is Red, its removal is usually straightforward (just remove it). If X is Black, its removal can violate the Black-Height property (a path loses a Black node). This violation needs to be fixed.
3. **Fix-up (Restoring Properties):** This is the most intricate part, involving many cases based on the colors of X's sibling S, S's children, and X's parent P. The goal is to restore the Black-Height property, often by:
   - **Recoloring:** Changing node colors.
   - **Rotations:** Performing rotations to redistribute Black nodes.
   - **Moving the "Double Black" problem:** The fix-up process often involves moving a "double black" violation up the tree until it's resolved or reaches the root.

Due to its complexity, implementing deletion correctly from scratch is a significant

challenge. For DSA learning, focus on understanding the concepts and insertion first.

3. search(data): Finding a Node

- **Concept:** This operation is identical to searching in a regular BST. No balancing is needed during a search.
- **Time Complexity:** O(logN) in both average and worst cases (because the tree is always balanced!). This is the main advantage of AVL and Red-Black trees over unbalanced BSTs.