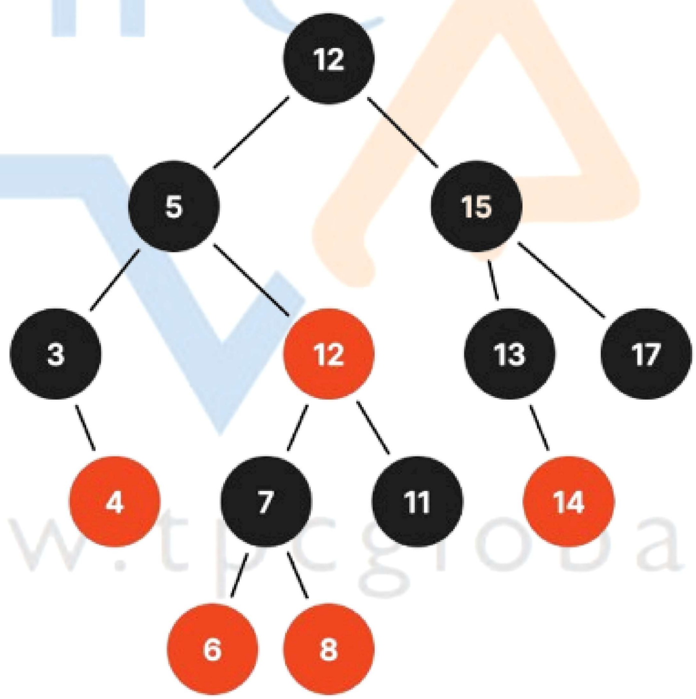
Red-Black Tree (RBT)

Introduction

A **Red-Black Tree (RBT)** is a **self-balancing Binary Search Tree (BST)** that ensures the height of the tree is **O(log n)**. It is widely used in **real-world applications** such as memory management, STL maps/sets, and scheduling systems.

- Unlike AVL trees, RBTs allow **some imbalance**, but enforce rules using **node colors** to guarantee approximate balance.
- Each node in RBT is either **Red** or **Black**.



Properties of Red-Black Tree

A Red-Black Tree must satisfy the following **five properties**:

1. **Node Color:** Each node is either **red** or **black**.
2. **Root Property:** The root node is always **black**.
3. **Leaf Property:** All leaves (**NIL nodes**) are **black**.
4. **Red Property:** If a node is **red**, then **both its children are black**.
5. **Black Height Property:** Every path from a node to its descendant leaves has the **same number of black nodes**.

Balance Guarantee: The longest path from root to leaf is at most **twice** the shortest path, ensuring **O(log n)** operations.

Advantages of Red-Black Tree

- Guarantees **O(log n)** search, insertion, and deletion.
- Less strict balancing than AVL, so **faster insertion and deletion**.
- Commonly used in:
  - **Linux kernel process scheduler**
  - **C++ STL `map` and `set`**
  - **Java `TreeMap` and `TreeSet`**

Node Structure

Each node stores:

```
class Node {
    int data;          // Node value
    Node* left;        // Left child
    Node* right;       // Right child
    Node* parent;      // Parent node
    bool color;        // RED or BLACK
};
```

Rotations in Red-Black Tree

Rotations are used to **maintain balance**:

1. Left Rotation (LL)

- Moves the right child up and the current node down to the left.
- Fixes **right-heavy** imbalances.

2. Right Rotation (RR)

- Moves the left child up and the current node down to the right.
- Fixes **left-heavy** imbalances.

> Rotations are performed during **insertions and deletions** to restore RBT properties.

Insertion in Red-Black Tree

Steps:

1. Insert the node **like in a BST** and color it **RED**.
2. Fix violations of RBT properties using **recoloring and rotations**.

Cases for Red-Black Tree Insertion Fix-Up:

Case 1: New node is root → Color it black
Case 2: Parent is black → Tree remains valid
Case 3: Parent & uncle are red → Recolor parent & uncle black, grandparent red, continue upward
Case 4: Parent red, uncle black, triangle shape → Rotate + recolor
Case 5: Parent red, uncle black, line shape → Rotate + recolor

Example:
Insert `10, 20, 30` into an empty RBT.

- Insert `10` → root, color black
- Insert `20` → red, no violation
- Insert `30` → red, parent `20` red → **RR rotation** at root → `20` becomes new root

Deletion in Red-Black Tree

Steps:

1. Delete the node **like in BST deletion**.
2. If a **black node is deleted**, it may violate RBT properties.
3. Fix violations using **rotations and recoloring**.

Cases for Deletion Fix-Up

- Sibling red → rotate + recolor

- Sibling black with black children → recolor parent & sibling
- Sibling black with at least one red child → rotate + recolor

Note: Deletion fix-up is **more complex than insertion**, but ensures **O(log n)** height.

Comparison: AVL Tree vs Red-Black Tree

Feature	AVL Tree	Red-Black Tree
Balance	Strict height balance	Loose balance using colors
Insert/Delete	Slower (more rotations)	Faster (fewer rotations)
Search	Slightly faster	Slightly slower
Use Cases	Databases, memory-limited	OS kernels, STL, Java collections

Applications of Red-Black Tree

1. **Associative arrays** (`TreeMap`, `TreeSet`)
2. **Priority queues**
3. **Memory management** in OS (Linux kernel)
4. **Multiset and map operations** in C++ STL