. Red-Black Trees – By Mani Abedii

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 Properties 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. **Leaves:** All leave are considered black.
4. The **black** height of the red-black tree is the number of black nodes on a path from the root node to a leaf node. A red-black tree of height **h** has **black height >= h/2**.
5. Height of a red-black tree with **n** nodes is **h<= 2 log2(n + 1)**.

6. **Red Rule:**

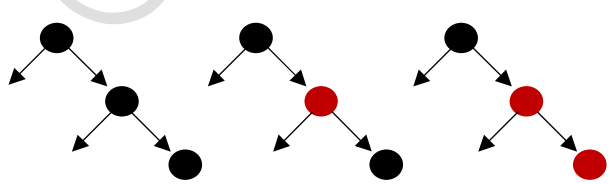
   A red node cannot have a red child (no two consecutive red nodes).
7. **Black Height Property:**

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

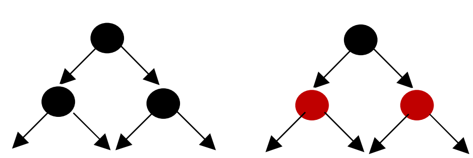
How do these properties ensure balance?

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 used rotation as a tool to do balancing after insertion. In the Red-Black tree, we use two tools to do the balancing: 1. Recoloring 2. Rotations

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:

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 "no consecutive red nodes" rule, 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. Chage the color of the grandparent as Red.
      iii. Repeat the same process for the grandparent.
   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.
      ii. RR Case:
         Fix: Perform a single left rotation.
      iii. LR Case:
         Fix: Perform a <u>left rotation</u> on the <u>left</u> child, then a <u>right rotation</u> on the node.
      iv. RL Case
         Fix: Perform a <u>right rotation</u> on the <u>right</u> child, then a <u>left rotation</u> on the node.

      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 condition.

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 x be the node to be deleted, y the node that is being actually deleted & x the node that replaces y.

- If y is red, just delete it, there won't be any violations.
- If y is Black, there will be **Double Black Violation** that needs to be fixed:

There are 4 possible cases that need to be handled:

Case 1: The sibling is red

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

Case 2: The sibling is black & its children are also black

- Recolor the sibling to red & the node to black
- If the parent is red, just make it black
- Otherwise fix the parent

Case 3: The sibling is red, its closer child to the node is red & its other child is black

- Do a right rotation on the sibling
- Now this case falls into case (4)

Case 4: The sibling is red & its farther child to the node is red

- Recolor the sibling to the color of the parent
- Recolor the parent to black
- Do a left rotation on the parent
- Recolor the node to black