**CSC 306: Introduction to Algorithm**

**Project**

Name : Victor Francis Halder

ID : 1610026

Section : 2

Faculty : Rajib Hossain Pavel & Shariful Islam

Topic: Red Black Tree

**Data Structures of Red Black Tree**

Red-Black Tree is a **self-balancing Binary Search Tree (BST)**, where every node follows following rules.  
[](https://www.geeksforgeeks.org/wp-content/uploads/RedBlackTree.png)

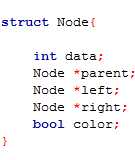
**1)** Every node has a color either red or black.

**2)** Root of tree is always black.

**3)** There are no two adjacent red nodes (A red node cannot have a red parent or red child).

**4)** All simple paths from any node x to a descendant leaf have the same number of black nodes = **black-height(x)**

In our Red Black Tree, it has an extra bit of storage per node, its **color**. We also need to keep track of the **parent** of each node, so that a red-black tree's node structure would be



All Red Black Tree has some 3 main operations:

* Search
* Insertion
* Deletion

**Data Structures & Algorithms of Red Black Tree**

Insertion

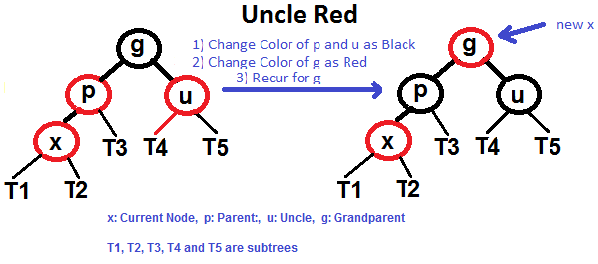
In Red Black Tree, we use rotation as a tool to do balancing after insertion caused imbalance. We use two tools to do balancing.

* Recoloring
* Rotation

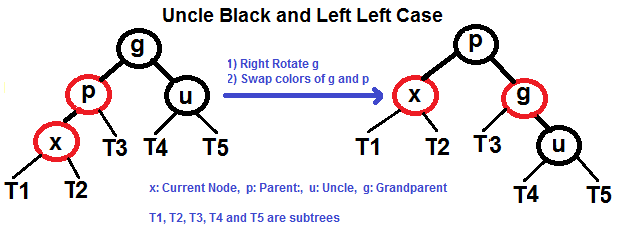
Our first approach is to try recoloring first, if recoloring doesn’t work, then we go for rotation.

The algorithms has mainly two cases depending upon the color of uncle. If uncle is RED, we do recoloring. If uncle is **BLACK**, we do rotations and/or recoloring.

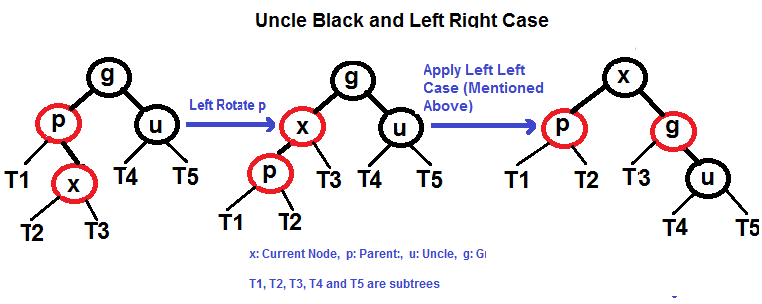
* Color of a NULL node is considered as **BLACK**.
* Let x be the newly inserted node.
* Perform standard BST insertion and make the color of newly inserted nodes as RED.
* If x is root, change color of x as **BLACK** (Black height of complete tree increases by 1).
* Do following if color of x’s parent is not **BLACK** or x is not root.
* **If x’s uncle is** **RED** (Grand parent must have been black from property 4)
* Change color of parent and uncle as **BLACK**.
* Color of grand parent as RED.
* Change x = x’s grandparent



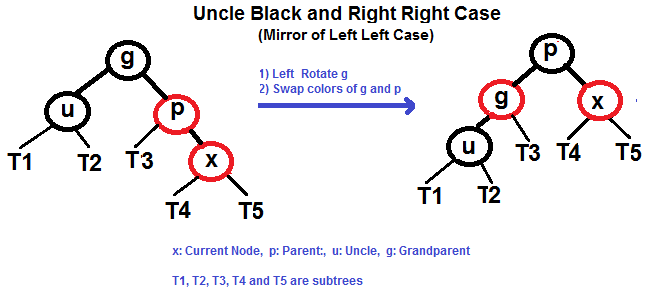
* **if x’s uncle is BLACK**, then there can be four configurations for x, x’s parent (**p**) and x’s grandparent (**g**)
* **Left Left Case** (p is left child of g and x is left child of p)
* Right Rotate (g)
* Swap colors of (g) and (p)



* **Left Right Case** (p is left child of g and x is right child of p)
* Left Rotate (p)
* Right Rotate(g)
* Swap Colors (g) and (p)



* **Right Right Case** (Mirror of Left Left Case)
* Left Rotate (g)
* Swap Colors (g) and (p)



* **Right Left Case (Mirror of Left Right Case)**
* Right Rotate(p)
* Left Rotate(g)
* Swap Colors (g) and (p)

**Algorithm of Red Black Tree**

Insertion:

* Let **x** be the newly inserted node.
* Perform standard **BST insertion** and make the color of newly inserted nodes as RED.
* If x is root, change color of x as **BLACK** (Black height of complete tree increases by 1).
* If color of x’s parent is not **BLACK** or x is not root
* **if x’s uncle is RED**
* Change color of parent and uncle as **BLACK**.
* Color of grand parent as RED.
* Change x = x’s grandparent
* **If x’s uncle is BLACK**
* **Left Left Case** (p is left child of g and x is left child of p)
* Right Rotate (g)
* Swap colors of (g) and (p)
* **Left Right Case** (p is left child of g and x is right child of p)
* Left Rotate (p)
* Right Rotate(g)
* Swap Colors (g) and (p)
* **Right Right Case** (Mirror of Left Left Case)
* Left Rotate (g)
* Swap colors of (g) and (p)
* **Right Left Case (Mirror of Left Right Case)**
* Right Rotate(p)
* Left Rotate(g)
* Swap Colors (g) and (p)

**Proof of Correctness of Red Black Tree**

**Black Height of a Red-Black Tree**

Black height is number of black nodes on a path from a node to a leaf. Leaf nodes are also counted black nodes. From above properties 3 and 4, we can derive, **a node of height h has black-height >= h/2**.

**Every Red Black Tree with n nodes has height <=** 2Log2(n+1)

**Proof:**

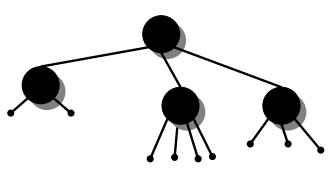
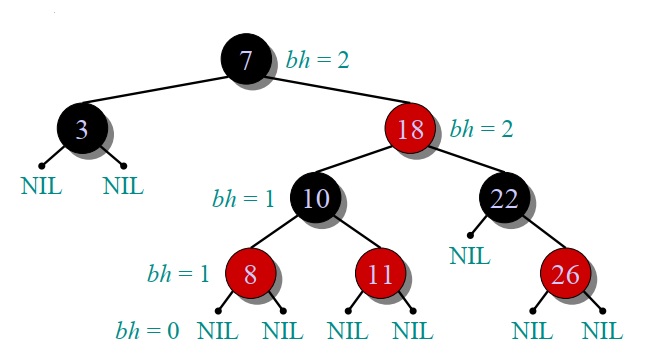
A Red Black Tree with *n* internal nodes has height at most 2lg(n+1)

For a general Binary Tree, let **k** be the minimum number of nodes on all root to NULL paths, then n >= 2k – 1 (Ex. If k is 3, then n is atleast 7). This expression can also be written as k <= 2Log2(n+1)

From property 4 of Red-Black trees and above claim, we can say in a Red-Black Tree with n nodes, there is a root to leaf path with at-most Log2(n+1) black nodes.

From property 3 of Red-Black trees, we can claim that the number black nodes in a Red-Black tree is at least ⌊ n/2 ⌋ where n is the total number of nodes.

From above 2 points, we can conclude the fact that Red Black Tree with **n** nodes has height <= 2Log2(n+1)



**Complexity Analysis of Red Black Tree**

The running time of Red Black Tree – **Insertion** takes at most **O(lg n)** time,

Since the height of a red-black tree on ***n***node is **lg n**.

In Red Black-Insert - Tree - **Fix Violations**, the **while** loop repeats only if case 1 occurs, and then the pointer of current node moves two levels up the tree.

The total number of times the **while** loop can be executed is therefore **O(lg n)**.

Therefore **Red Black Tree – Insertion** takes a total of O(lg n) time. It usually doesn’t perform more than two rotations, since the **while** loop terminates if case 2 or case 3 is executed.