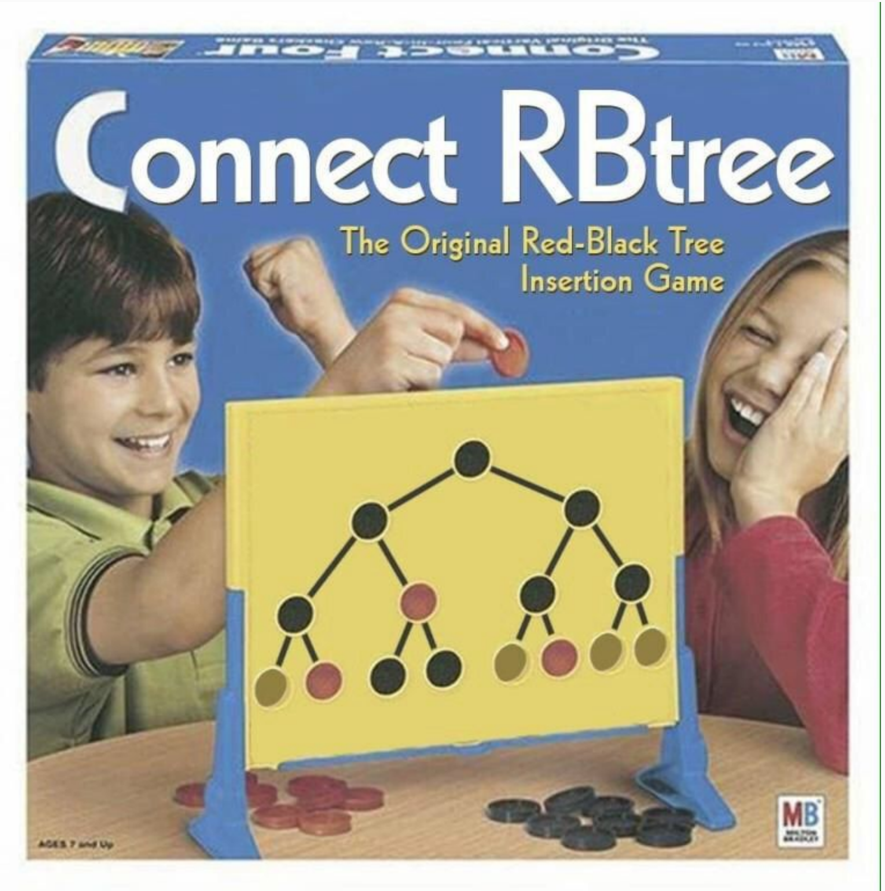
Red-Black Trees

CS2040S, AY19/20 Sem 1

Eldon Chung | eldon.chung@u.nus.edu
Wang Zhi Jian | wzhijian@u.nus.edu

Red-Black Trees are fun!

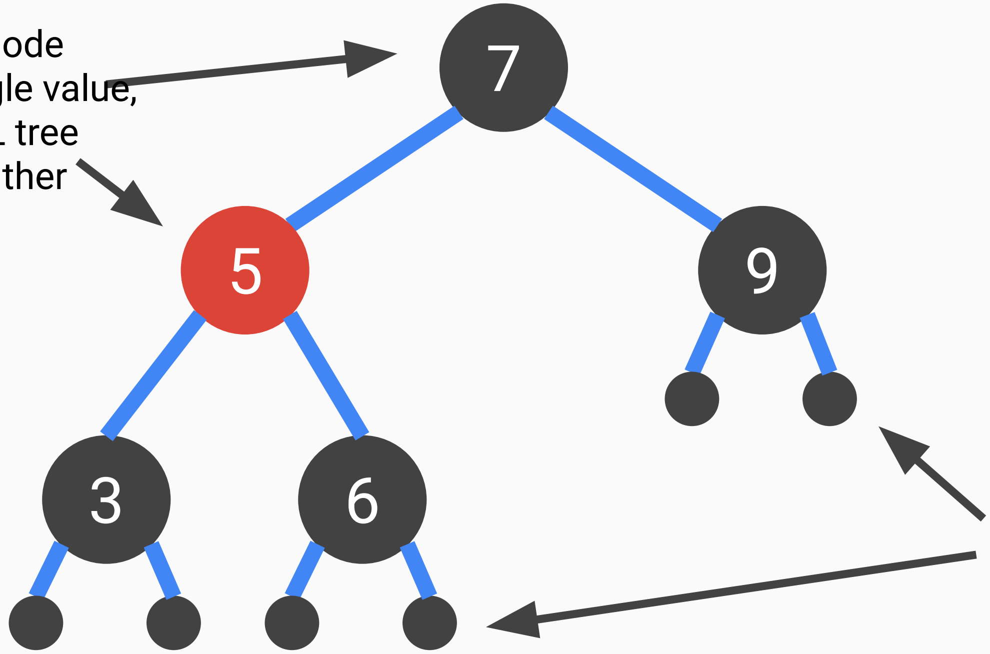


Motivation

1. Low constant factor, faster than AVLs in practice!
2. Less amount of rebalancing that needs to be done. (Constant amount) Still needs potentially O(log n) recolours though, but this is a cheap operation!
3. Rebalancing algorithm after insert and deletes very similar!

Structure of a Red-Black Tree

- Each **internal** node contains a single value, just like an AVL tree
- Each node is either **red** or **black**

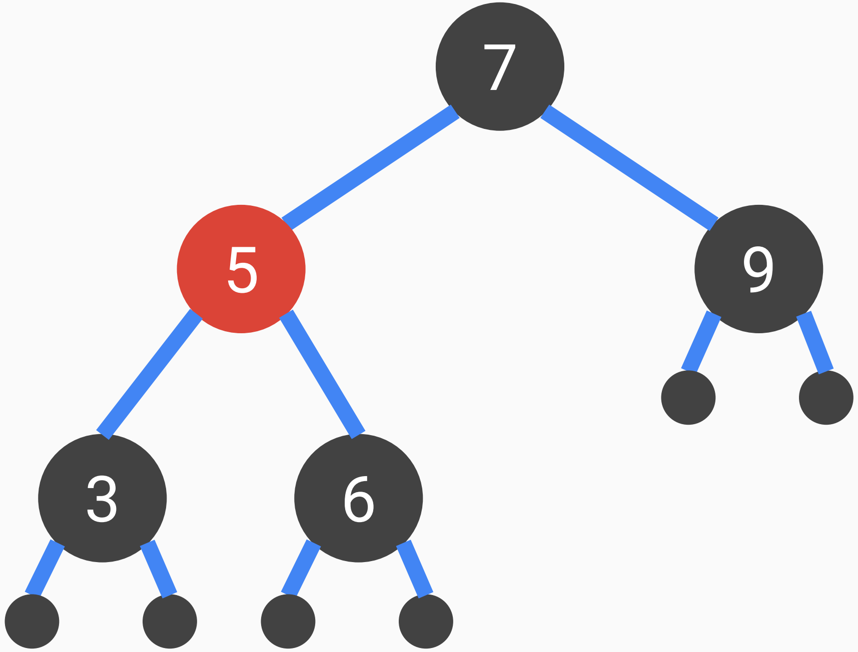


- All leaf nodes contain null values (for convenience later on)

Properties of a Red-Black Tree

Property 0

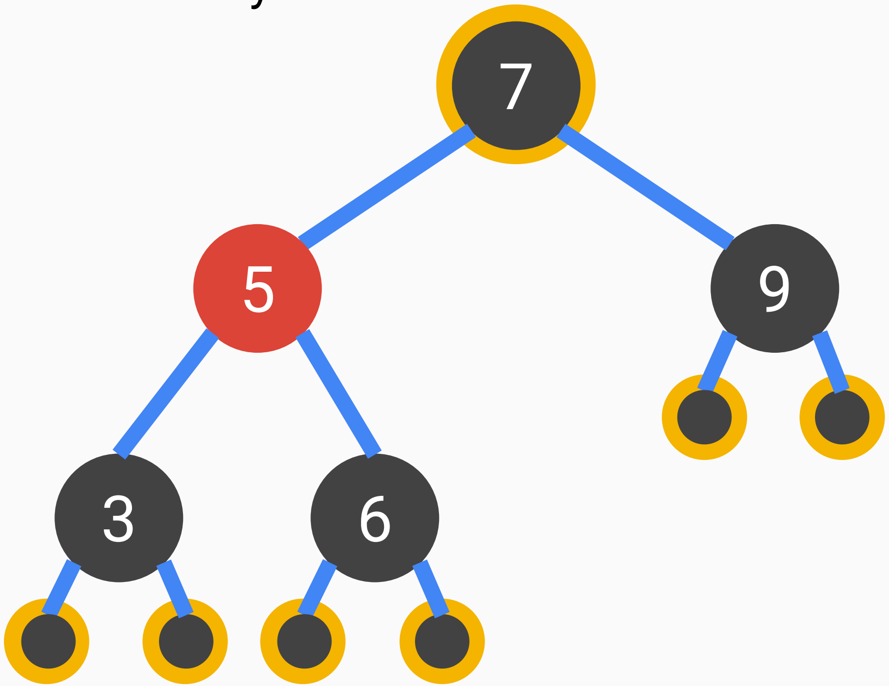
The BST property is maintained for all nodes in the Red-Black Tree.



Properties of a Red-Black Tree

Property 1

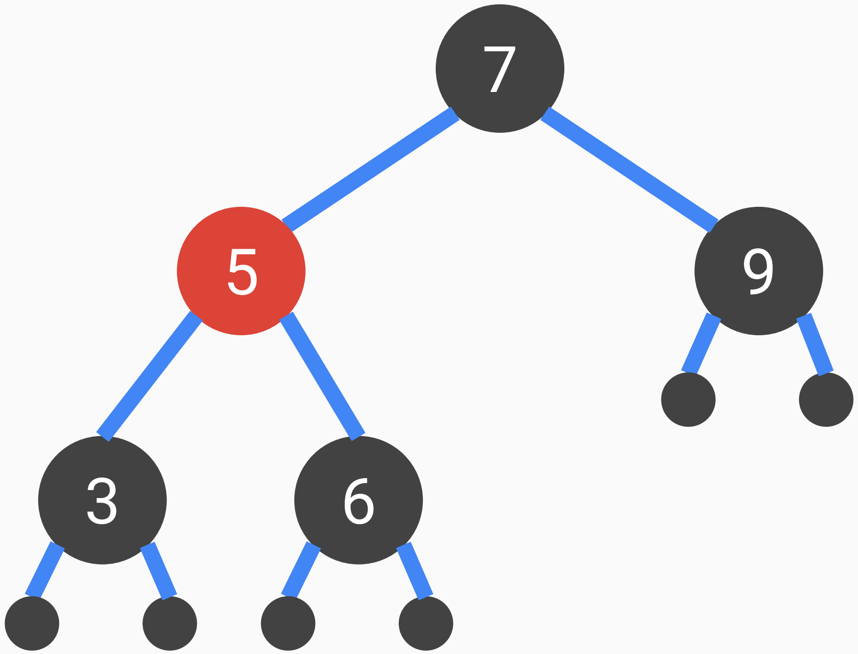
The root and leaves are always black.



Properties of a Red-Black Tree

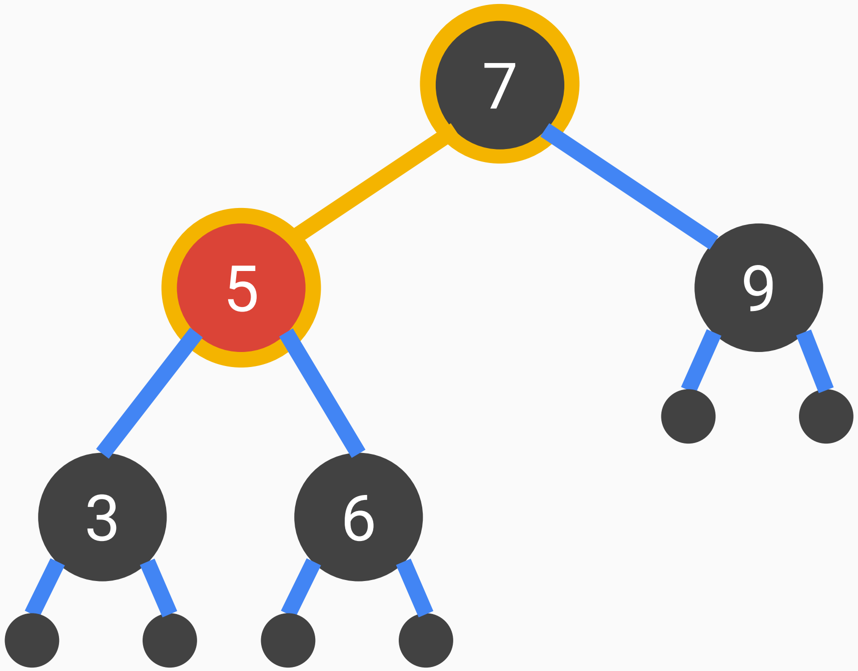
Property 2

Every node is either red or black.



Property 3

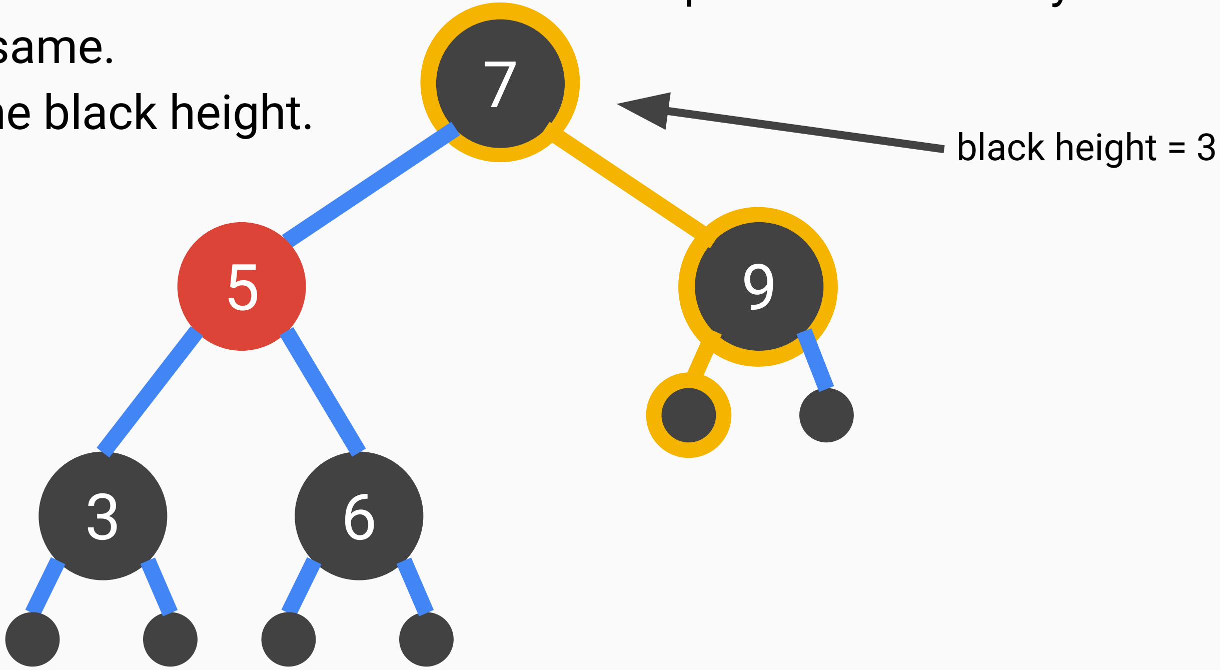
The parent of a red node must be black.



Properties of a Red-Black Tree

Property 4

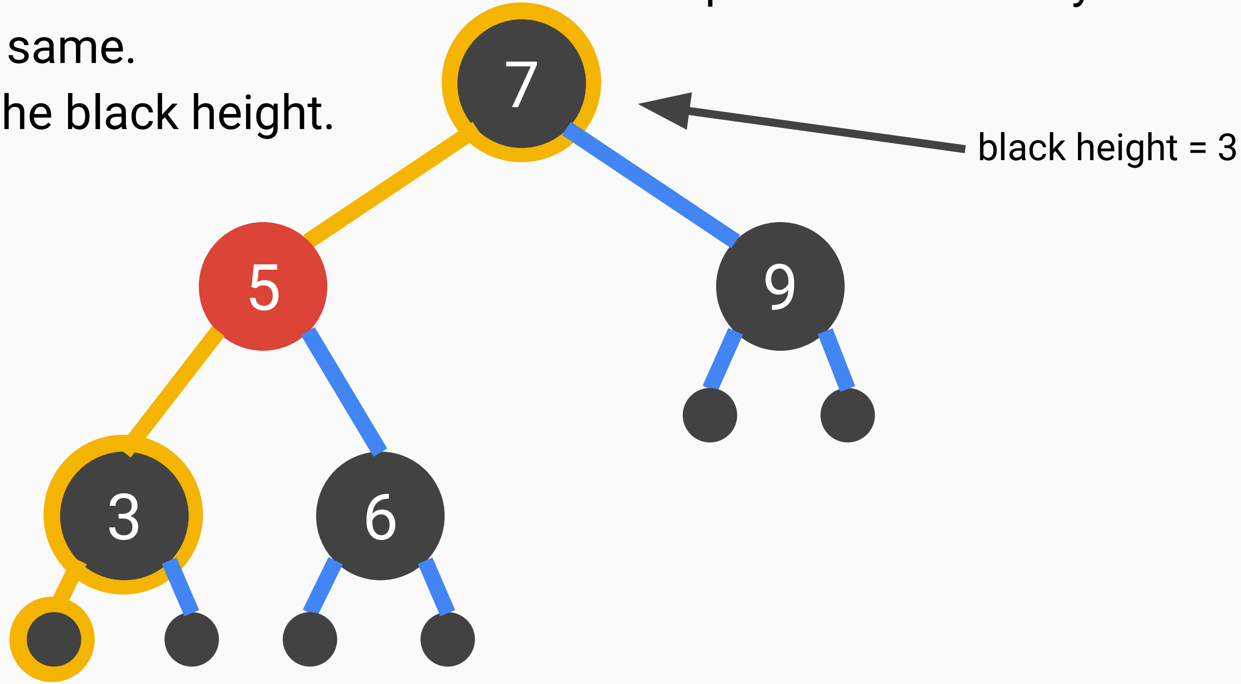
For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Properties of a Red-Black Tree

Property 4

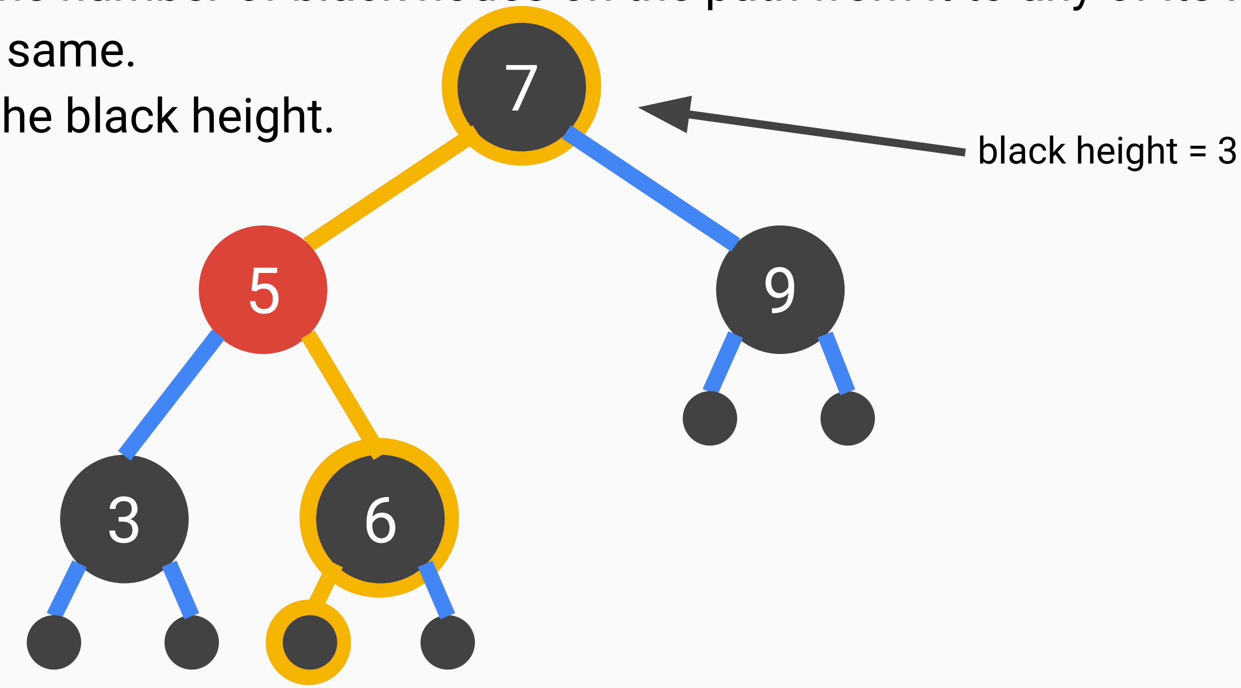
For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Properties of a Red-Black Tree

Property 4

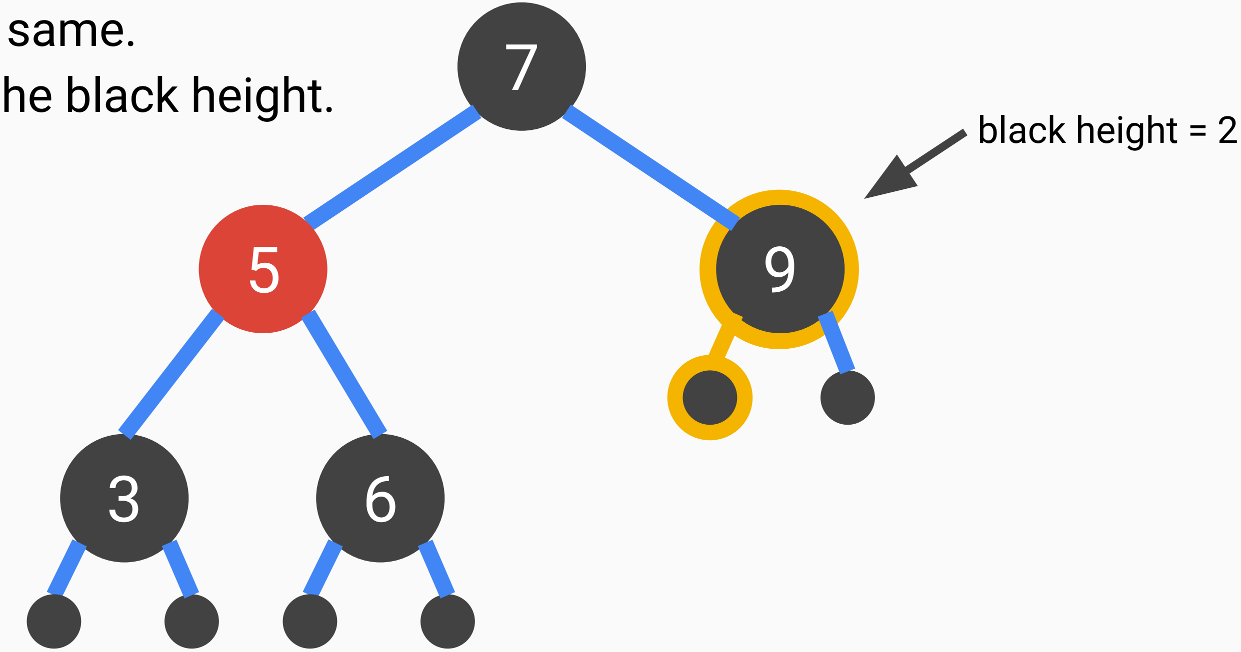
For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Properties of a Red-Black Tree

Property 4

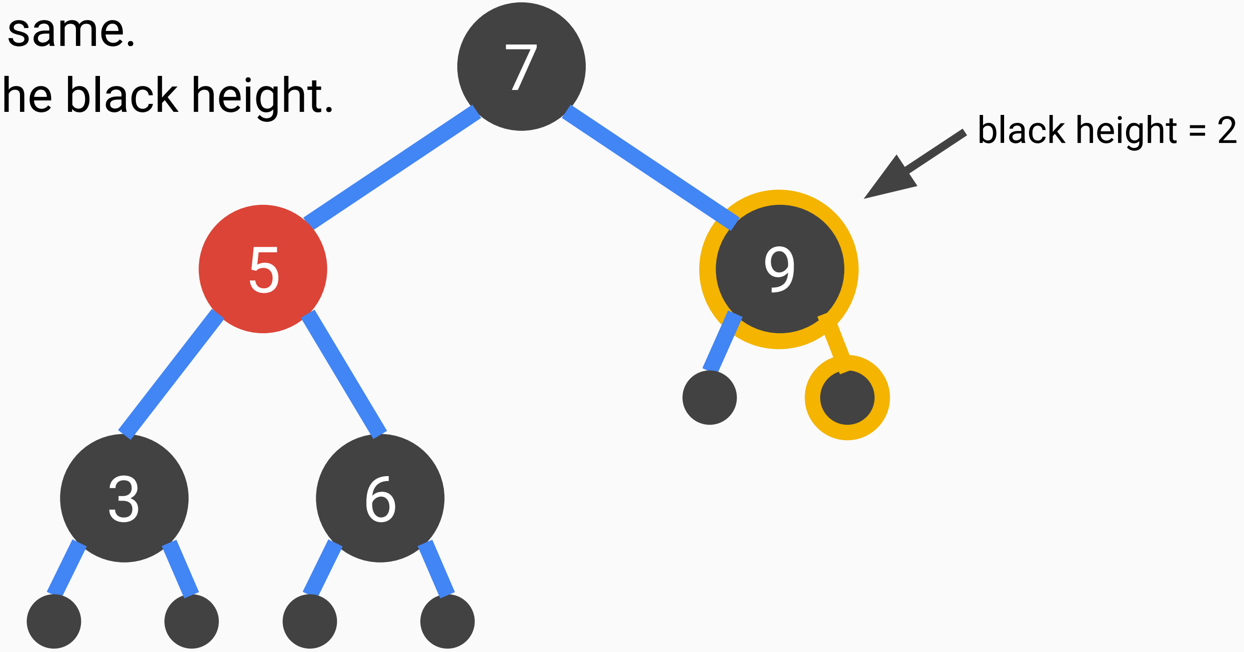
For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Properties of a Red-Black Tree

Property 4

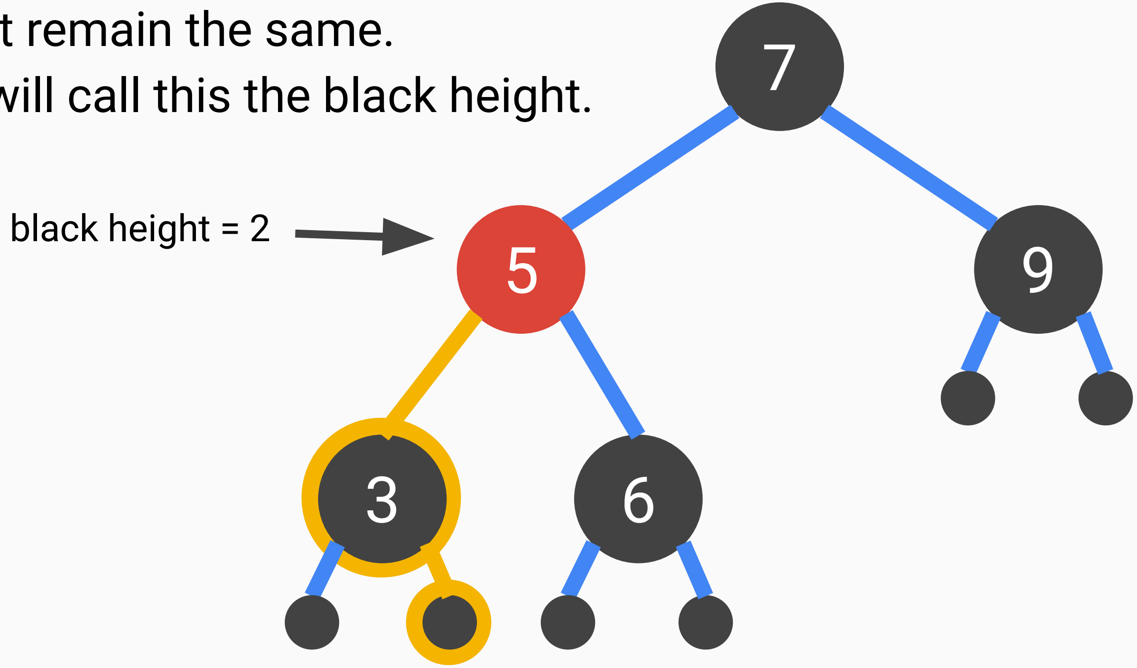
For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Properties of a Red-Black Tree

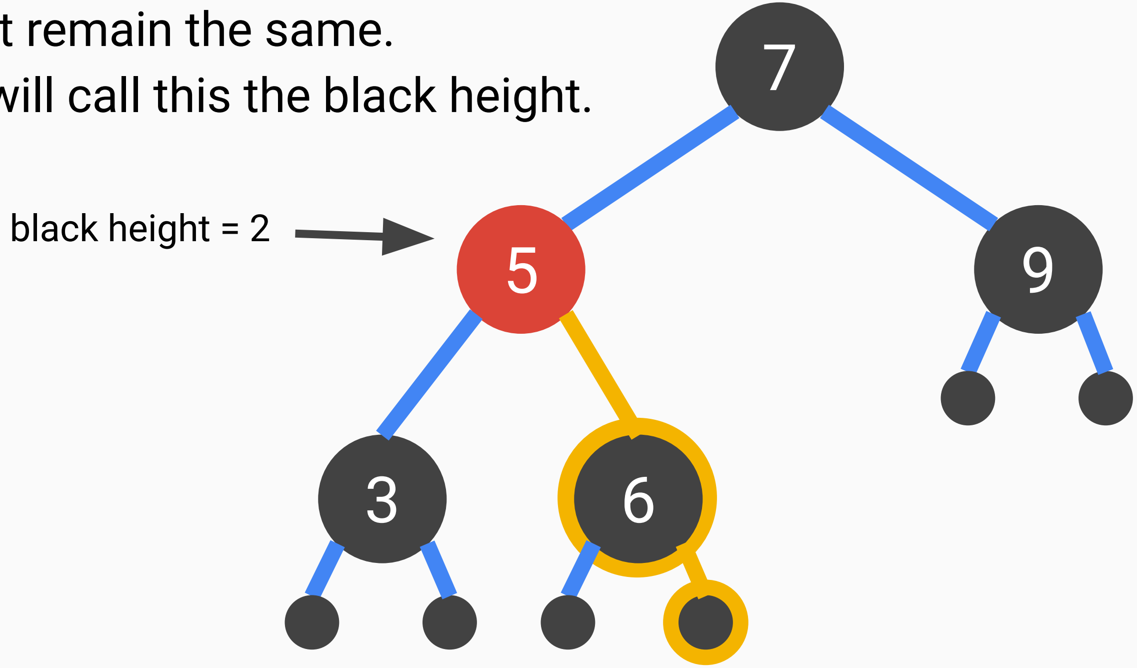
Property 4

For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Property 4

For every node, the number of black nodes on the path from it to any of its leaves must remain the same.
We will call this the black height.



Properties of a Red-Black Tree: Summary

Properties of a Red-Black Tree

1. The root and leaves are always **black**.
2. Every node is either **red** or **black**.
3. The parent of a red node must be **black**.
4. For every node, the number of **black** nodes on the path from it to any of its leaves must remain the same. We will call this the **black height**.

How “balanced” is this tree?

For this tree to be “good” we want that the height *h* is at most O(log *n*), since our operations will take time proportional to the height.

Recall that for the root, any path from it to any of its leaves needs to have the same number of black nodes, and the longest path intuitively alternates red and black nodes, whereas the shortest path is all black nodes.

e.g. black height = 3

shortest:  longest: 

So this means that the left subtree and right subtree can only differ by at most a factor of 2.

How “balanced” is this tree?

Using this, we want to show that the height of the tree should be at most O(log *n*).

To do this, let’s say that a size *n* Red-Black Tree has a black height of at least $h/2$. Since the tree has black height at least $h/2$, it will have at least $2^{h/2} - 1$ nodes. So we get that:

$$n \geq 2^{h/2} - 1$$

$$\log(n + 1) \geq h / 2$$

$$2 \log(n + 1) \geq h$$

Thus the height of a Red-Black Tree is at most O(log *n*).

Red-Black Tree Operations

Operations

Since the Red-Black Tree is an implementation of the Ordered Dictionary ADT, it supports all the operations of an Ordered Dictionary ADT.

Most of the operations work in the same way as for BSTs, so we will focus on the two operations that work differently for Red-Black Trees: **Insert** and **Delete**.

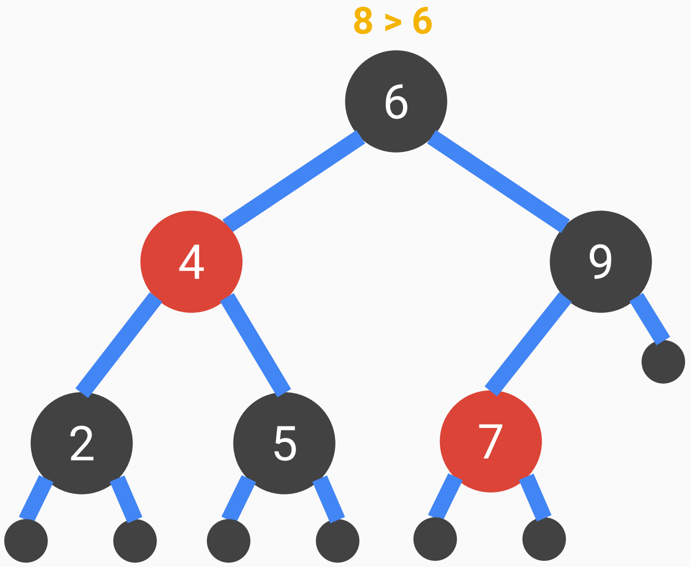
Insert and **Delete** make use of a very special process: **Rebalance and recolouring**.

Example of an Insertion

Red-Black Tree: Insert

First, we find the location to insert the new node, using a similar procedure as in BSTs.

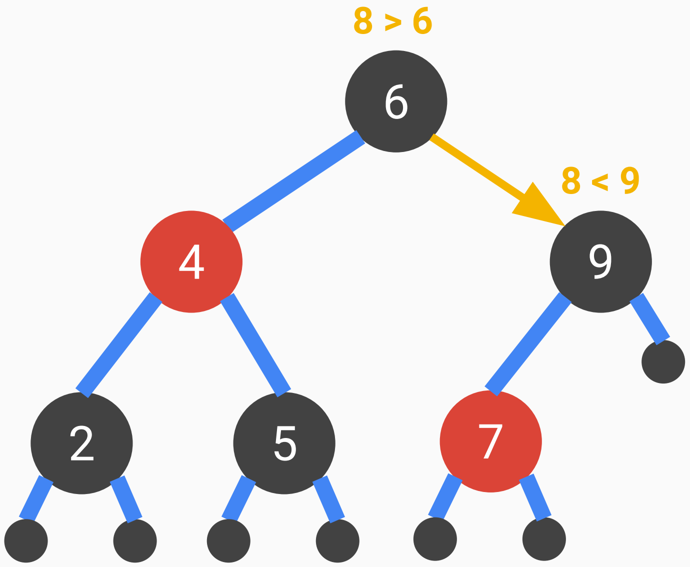
e.g. Insert 8



Red-Black Tree: Insert

First, we find the location to insert the new node, using a similar procedure as in BSTs.

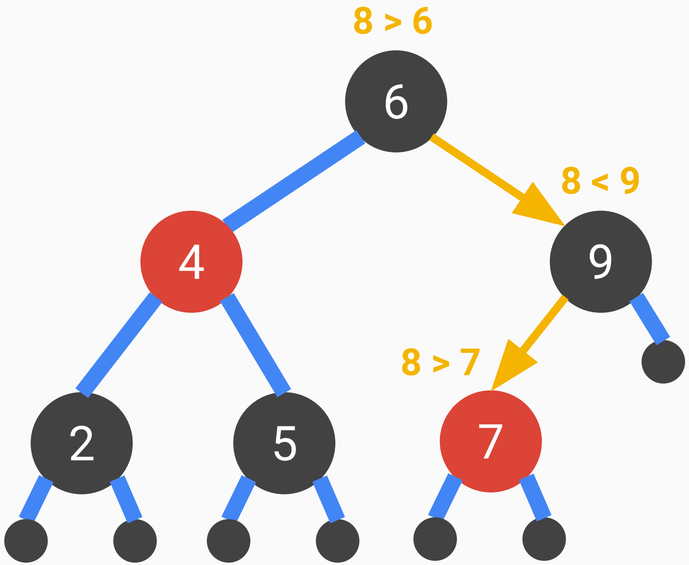
e.g. Insert 8



Red-Black Tree: Insert

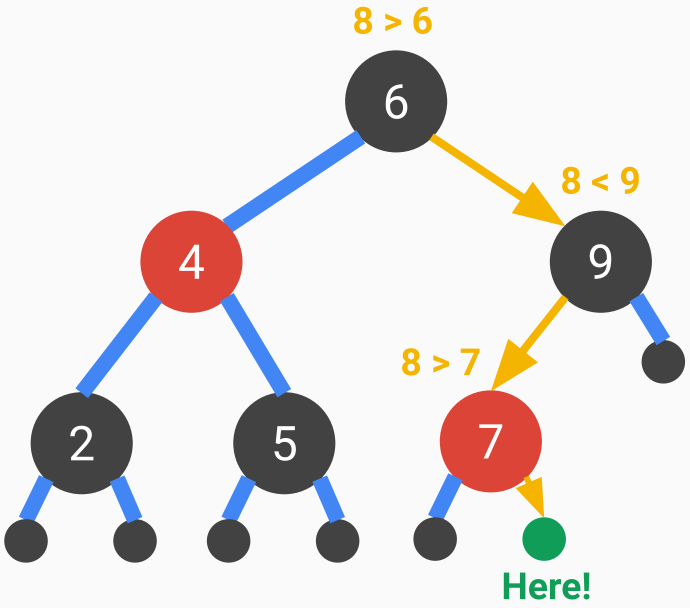
First, we find the location to insert the new node, using a similar procedure as in BSTs.

e.g. Insert 8



Red-Black Tree: Insert

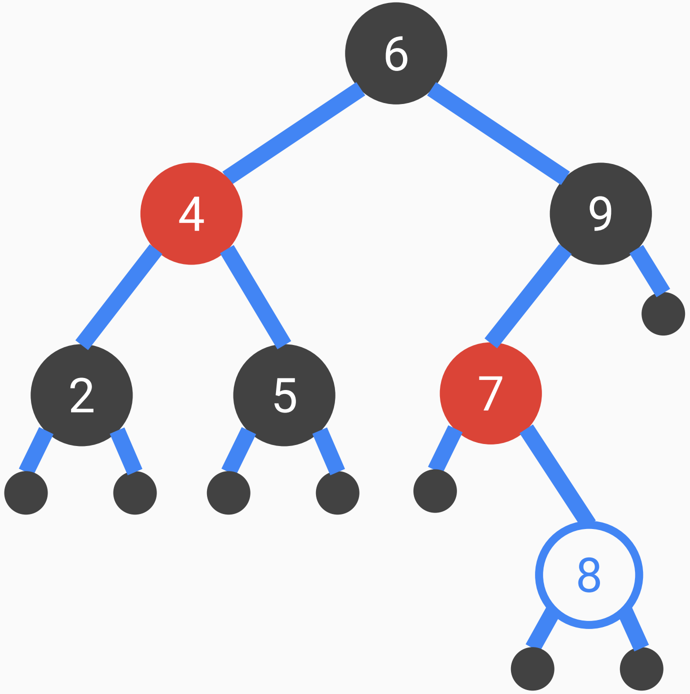
First, we find the location to insert the new node, using a similar procedure as in BSTs.
e.g. Insert 8



Red-Black Tree: Insert

Then, replace the null leaf with the value to be inserted into the tree, and add two null leaves.

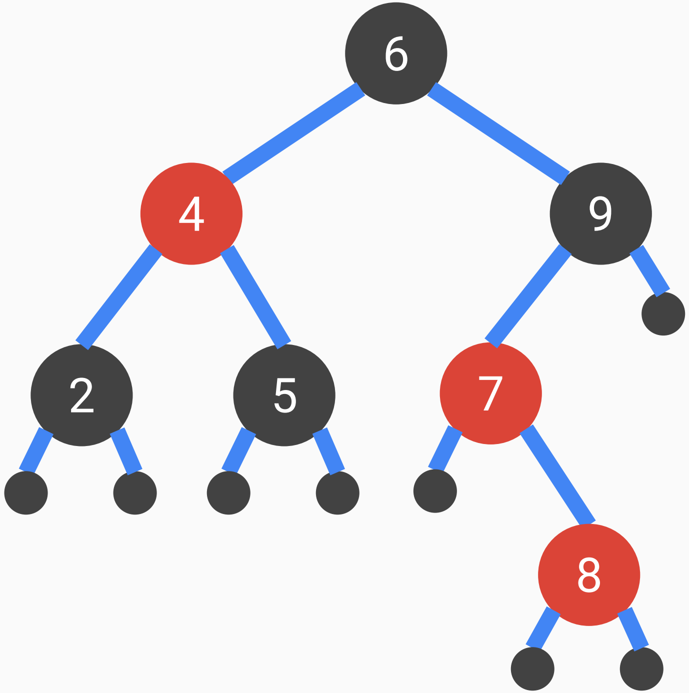
e.g. Insert 8



Red-Black Tree: Insert

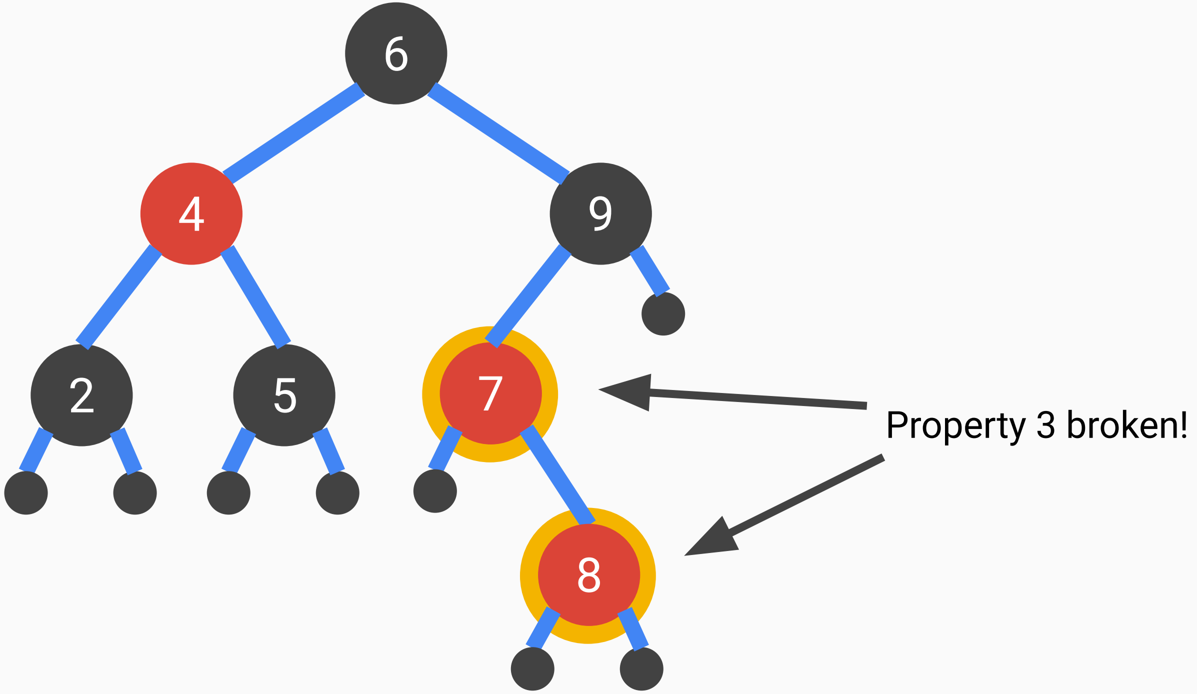
Finally, colour the node **red**.

e.g. Insert 8



Red-Black Tree: Insert

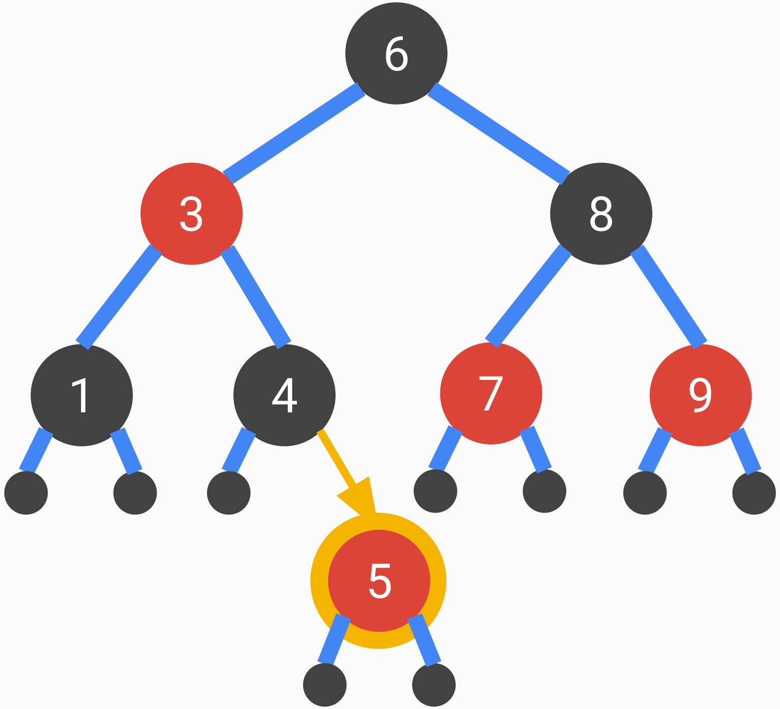
Notice that insertions may break property 3 (The parent of a red node must be black).
We need to **rebalance** the tree next. (Refer to the section on rebalancing).
e.g. Insert 8



Example of a Deletion

Red-Black Tree: Delete

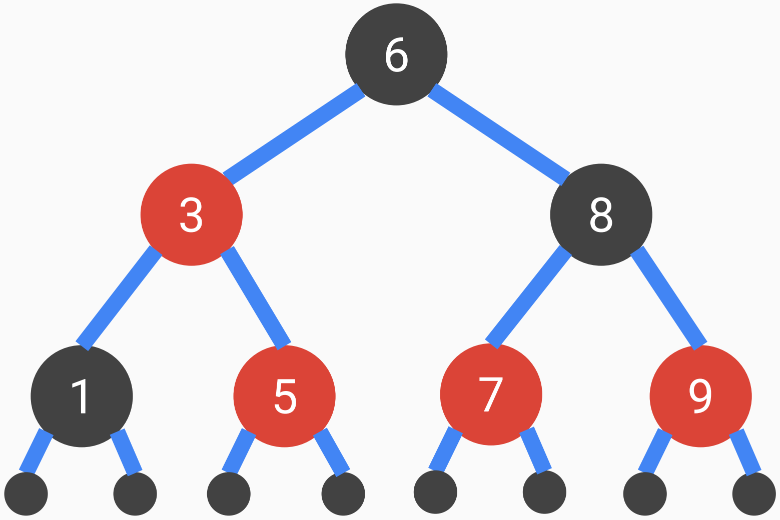
First, we find replacement for the deleted node, using a similar procedure as in BSTs.
e.g. Delete 4



Red-Black Tree: Delete

Then, perform the deletion.

e.g. Delete 4

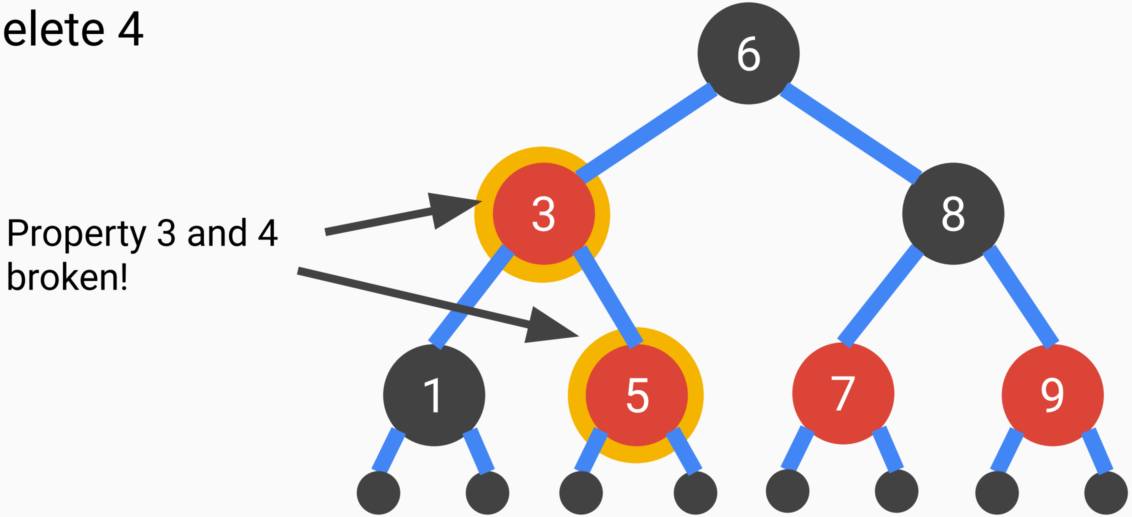


Red-Black Tree: Delete

Notice that deletions may break properties 1, 3 and 4.

We need to **rebalance** the tree next. (Refer to the section on rebalancing).

e.g. Delete 4

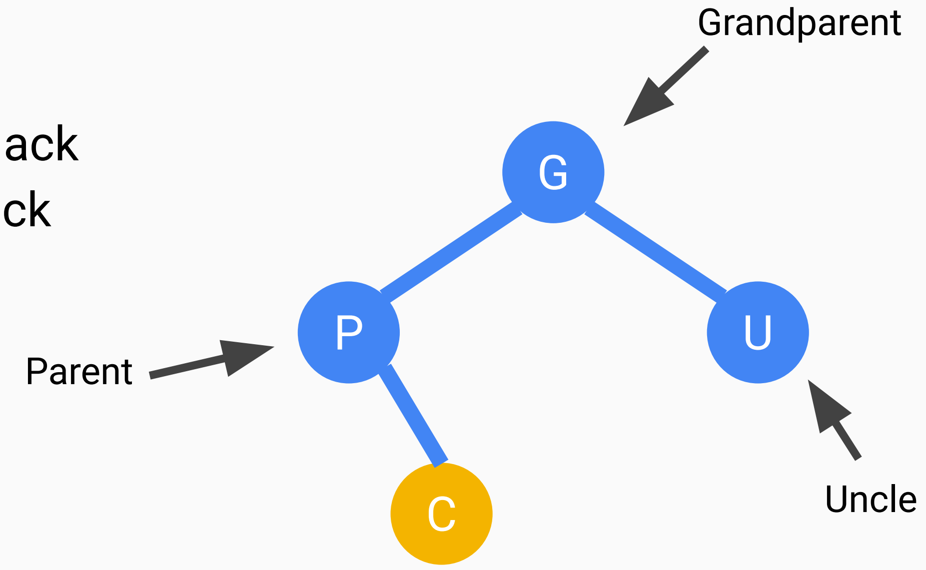


The Rebalancing Algorithm (For Insertion*)

Red-Black Tree: Rebalance

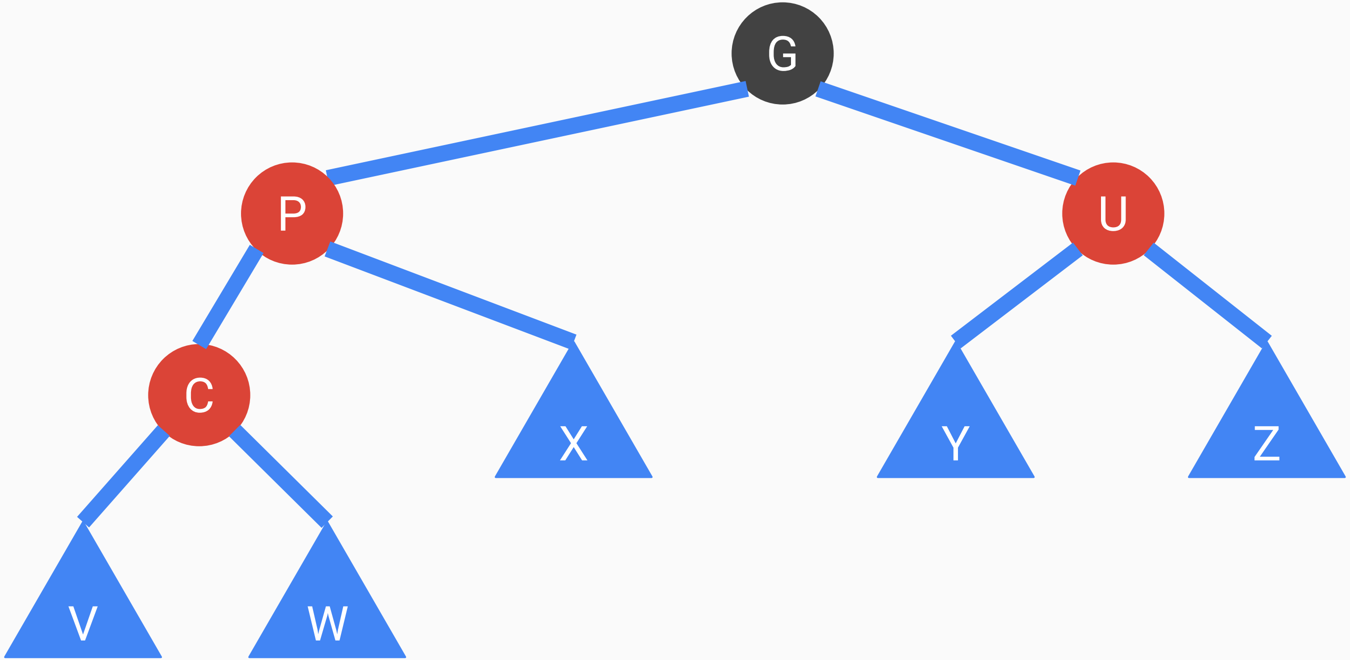
Inserting a node may cause violations of the properties of a Red-Black Tree. There are 3 possible "shapes" the tree can look like:

1. Uncle is red
2. Node is right child and uncle is black
3. Node is left child and uncle is black



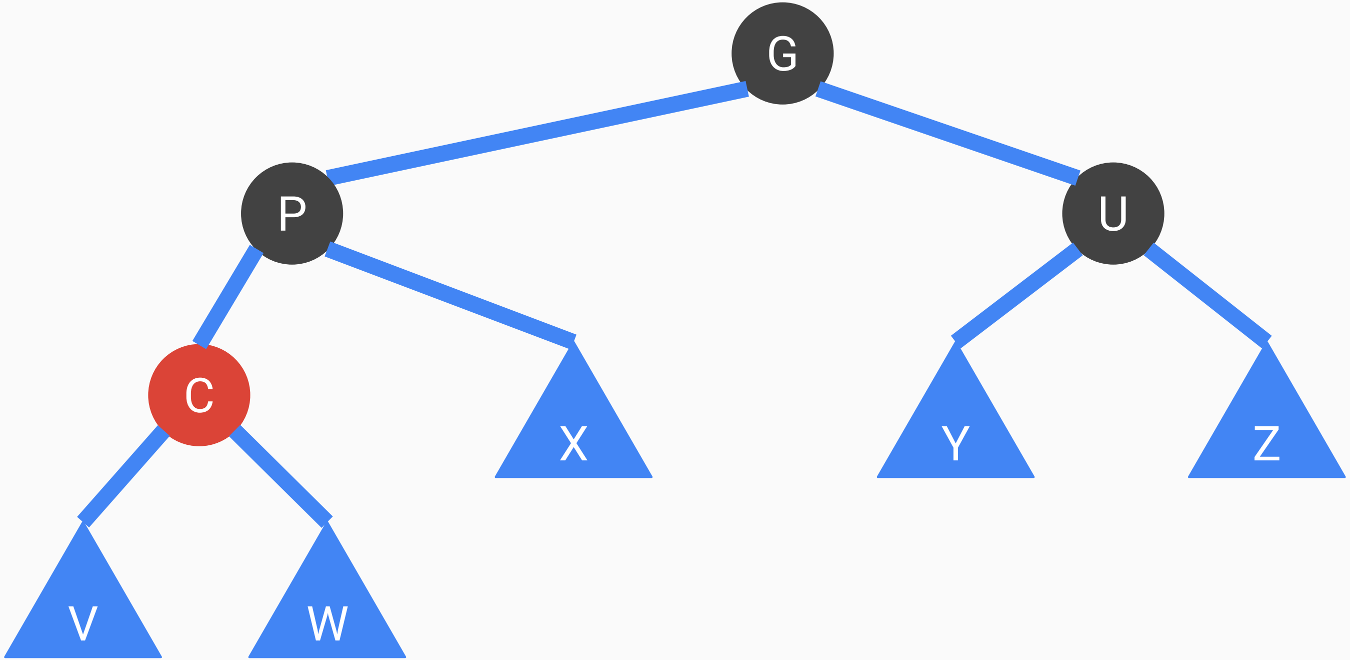
Red-Black Tree: Rebalance

1. Uncle is red.



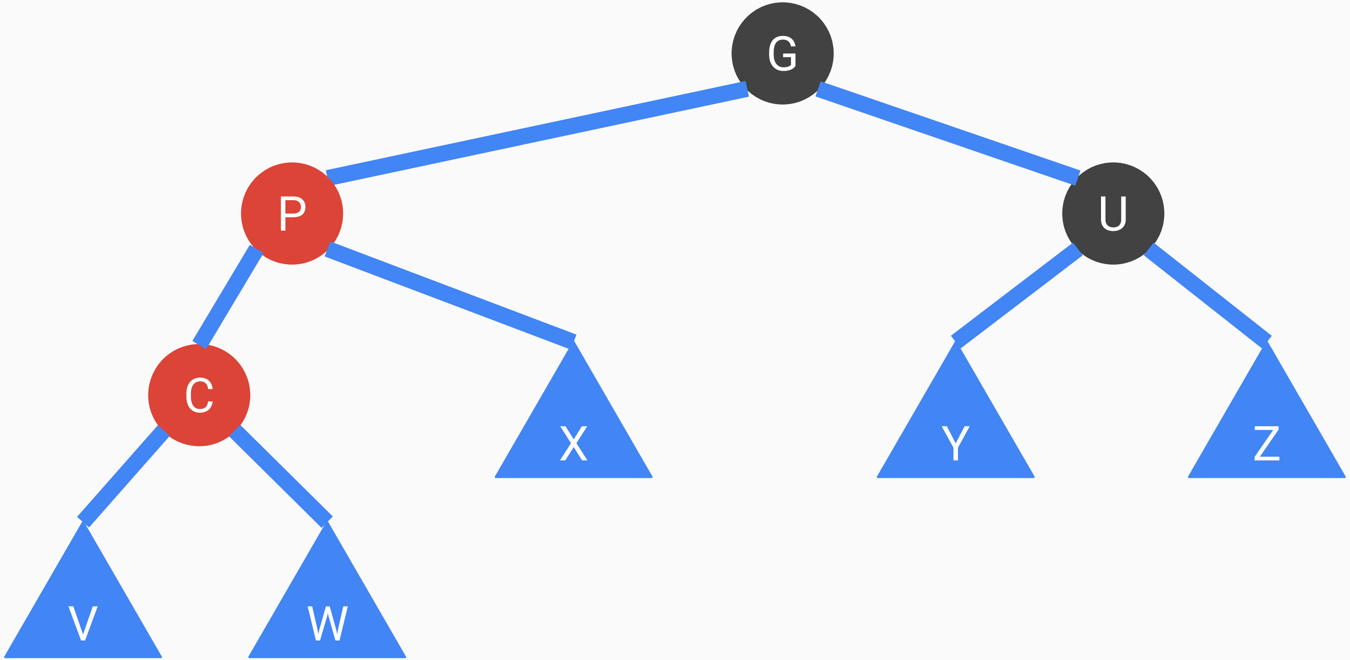
Red-Black Tree: Rebalance

1. Uncle is red.
   To fix, recolour the parent and the uncle.



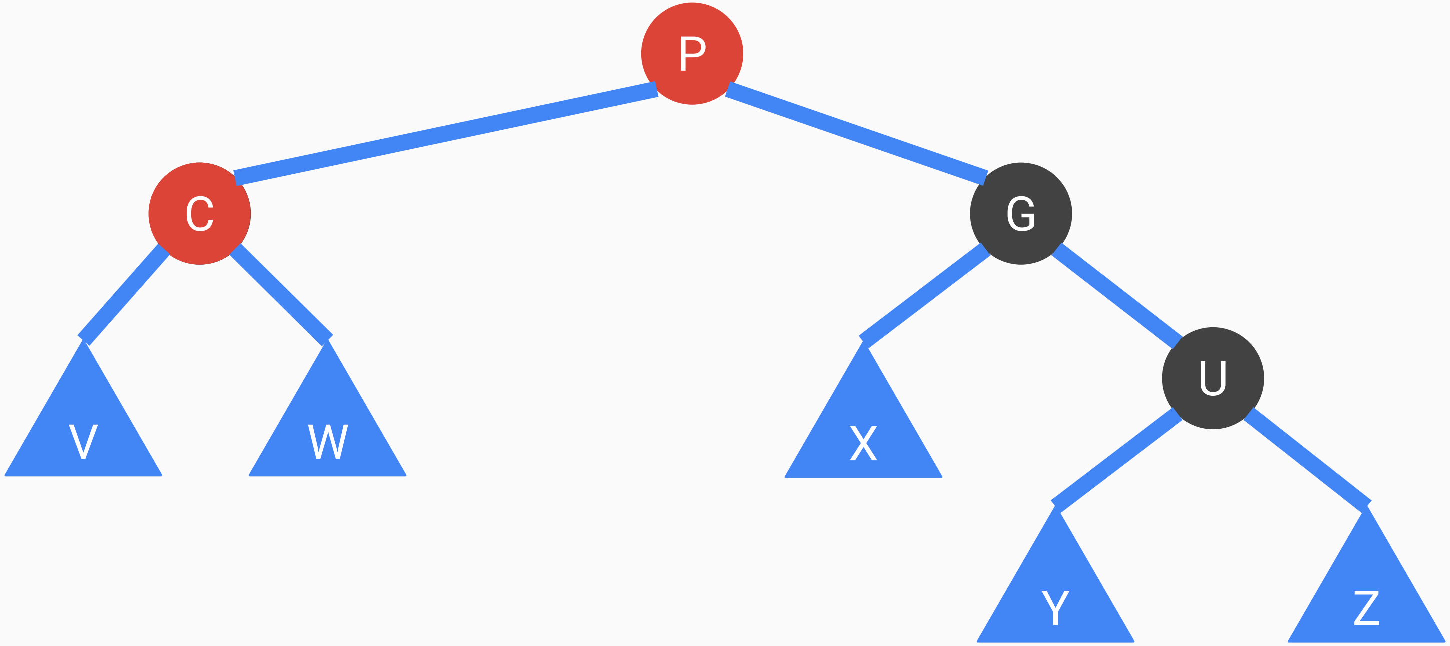
Red-Black Tree: Rebalance

2. Node is left child and uncle is black.



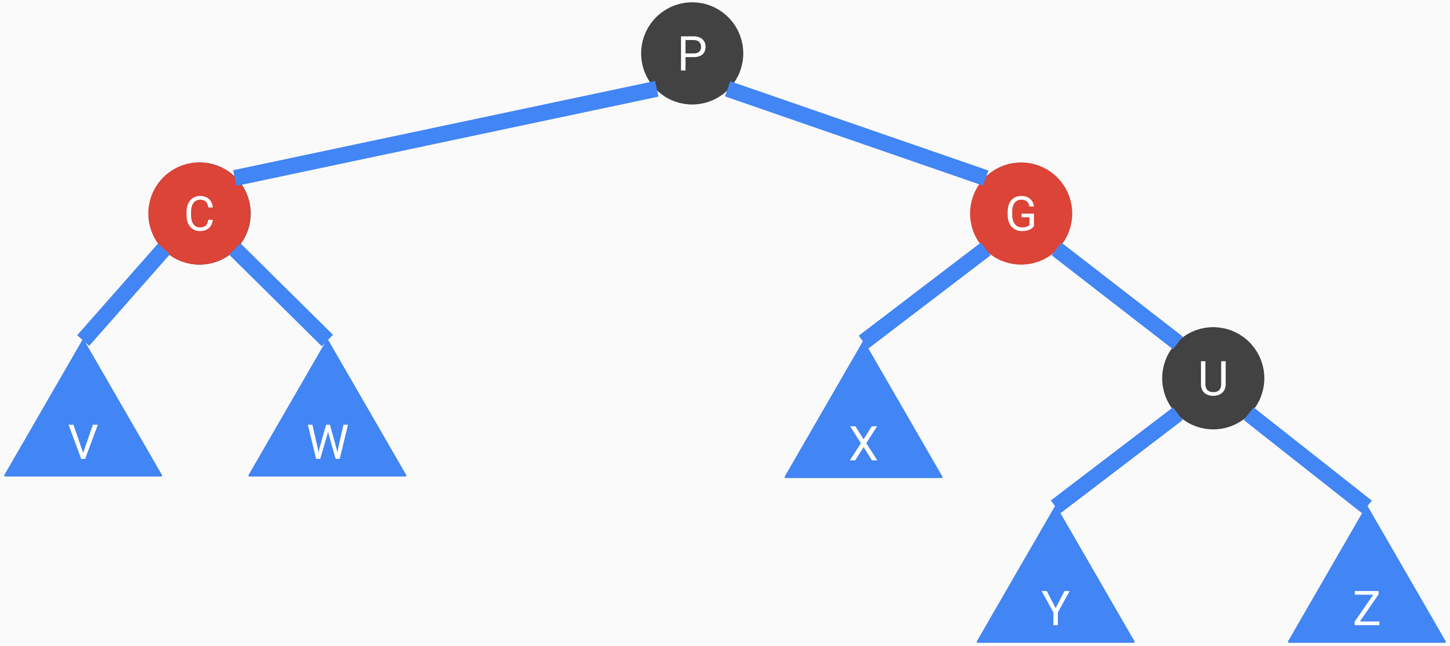
Red-Black Tree: Rebalance

2. Node is left child and uncle is black.
   First, perform a rotation on the grandparent and the parent.



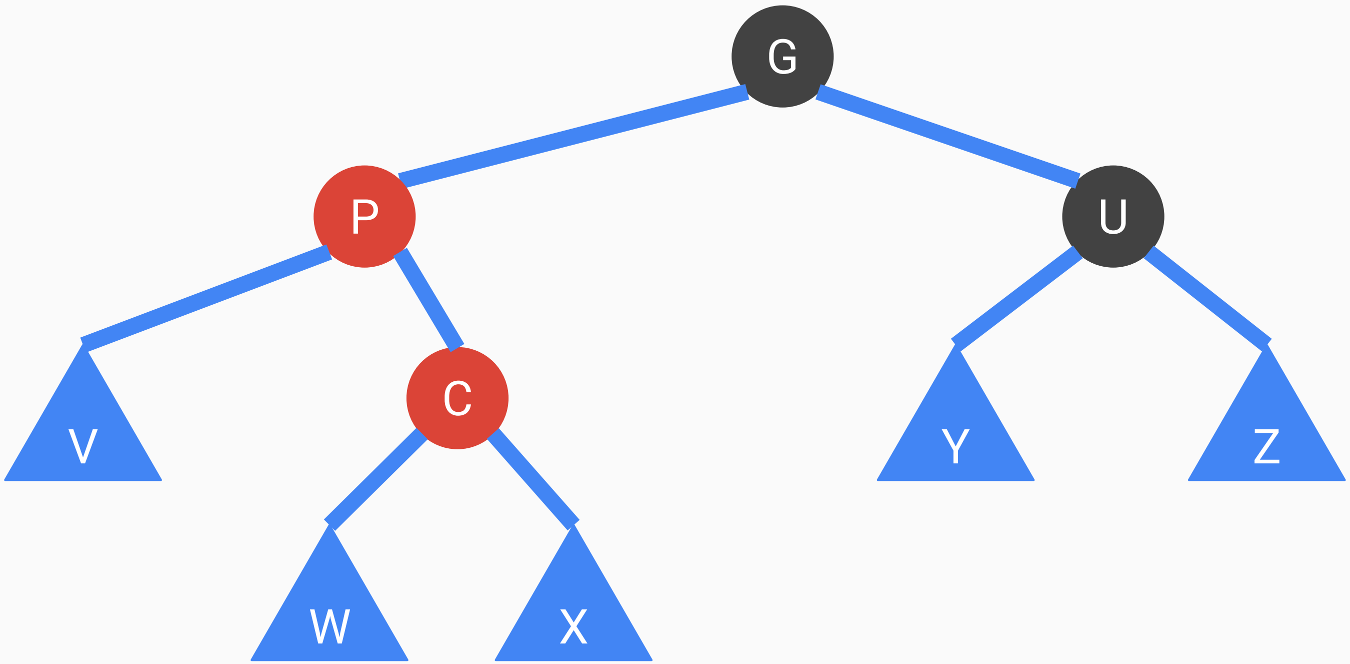
Red-Black Tree: Rebalance

2. Node is left child and uncle is black.
   Then, colour the parent black and the grandparent red.



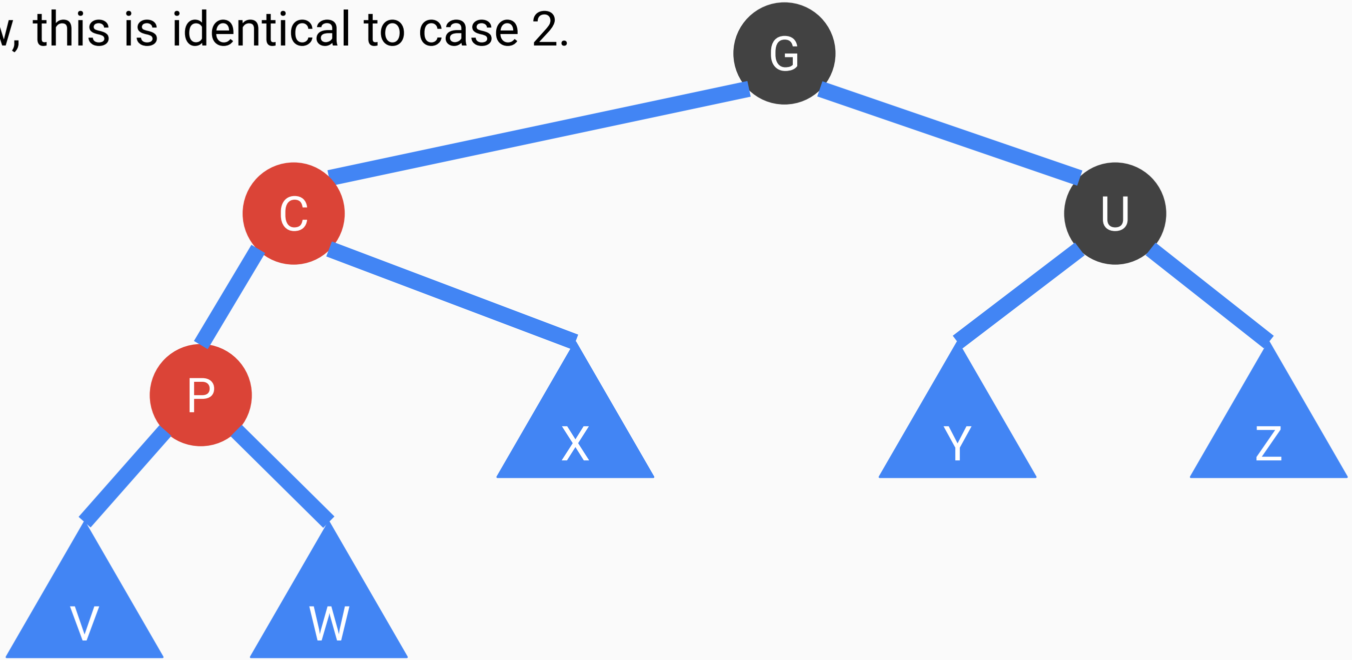
Red-Black Tree: Rebalance

3. Node is right child and uncle is black.



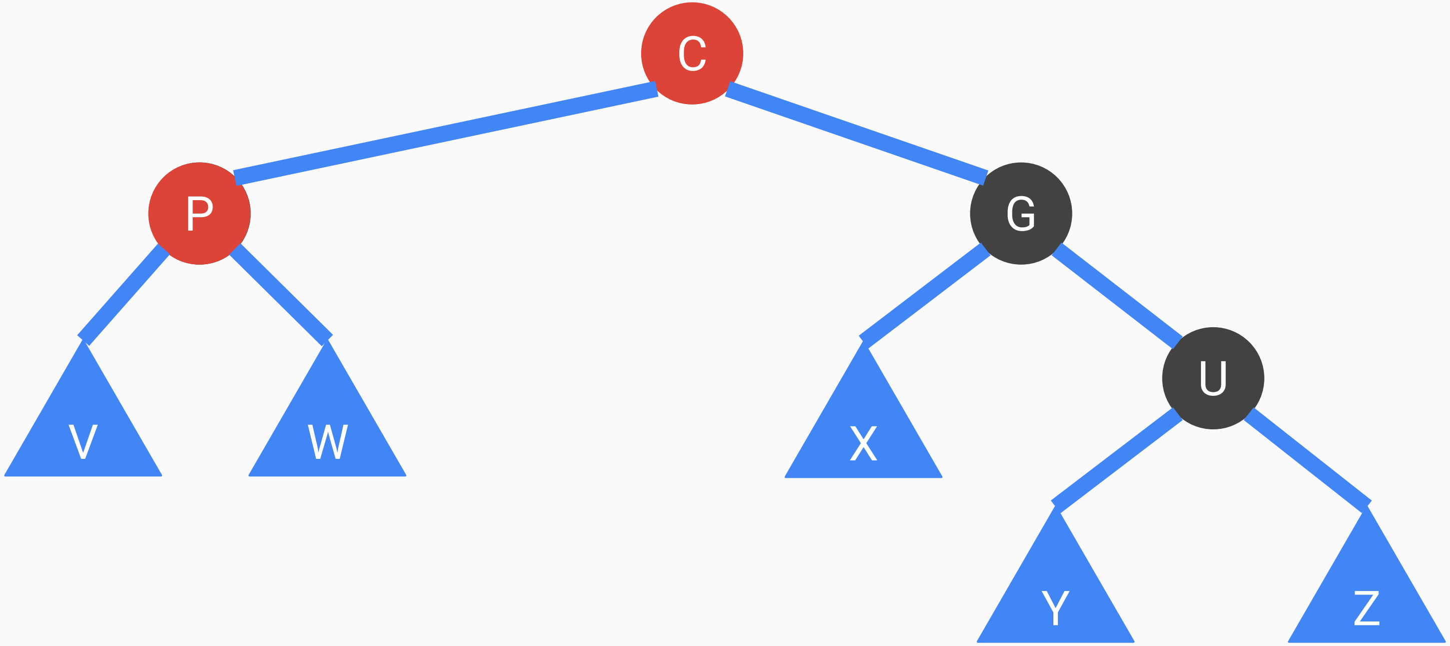
Red-Black Tree: Rebalance

3. Node is right child and uncle is black.
   First, perform a rotation on the parent and the child.
   Now, this is identical to case 2.



Red-Black Tree: Rebalance

3. Node is right child and uncle is black.
   Then, perform a rotation on the grandparent and the child.



Red-Black Tree: Rebalance

3. Node is right child and uncle is black.
   Then, colour the child black and the grandparent red.