Red-Black Trees

Step 1

In the previous section, we learned about the **AVL Tree**, which is a special self-balancing **Binary Search Tree** that guarantees a **worst-case O(log *n*)** time complexity for finding, inserting, and removing elements. However, the **AVL Tree** required to make two passes through the tree for inserting or removing elements: one pass down the tree to actually perform the insertion or removal, and then another pass up the tree to maintain the tree's balance. Can we somehow avoid this second pass to speed things up a bit further?

In 1972, Rudolf Bayer, a German computer scientist, invented a data structure called the **2-4 Tree** in which the length of any path from the root to a leaf was guaranteed to be equal, meaning that the tree was guaranteed to be perfectly balanced. However, the **2-4 Tree** was not a **Binary Search Tree**.

In 1978, Leonidas J. Guibas and Robert Sedgewick, two computer scientists who were Ph.D. students advised by the famous computer scientist Donald Knuth at Stanford University, derived the **Red-Black Tree** from the **2-4 Tree**. Some say that the color red was chosen because it was the best-looking color produced by the color laser printer available to the authors, and others say it was chosen because red and black were the colors of pens available to them to draw the trees.

In this section, we will discuss the **Red-Black Tree**, which is a self-balancing **Binary Search Tree**, resulting in a **worst-case O(log *n*)** time complexity, that is able to insert and remove elements by doing just a *single* pass through the tree, unlike an **AVL Tree**, which requires *two* passes through the tree.

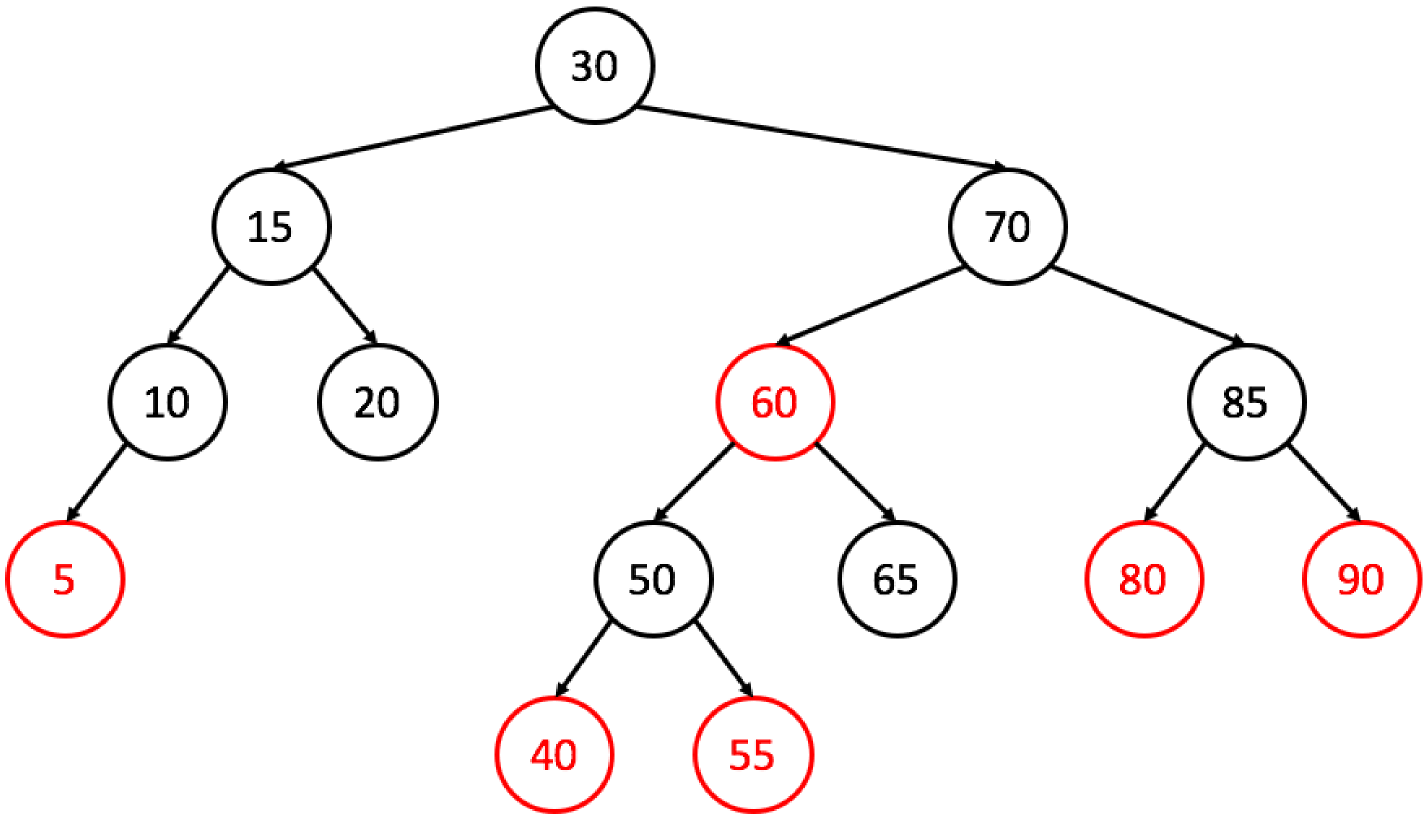
Figure: Leonidas Guibas (left) and Robert Sedgewick (right)

Step 2

A **Red-Black Tree** is a **Binary Search Tree** in which the following four properties must hold:

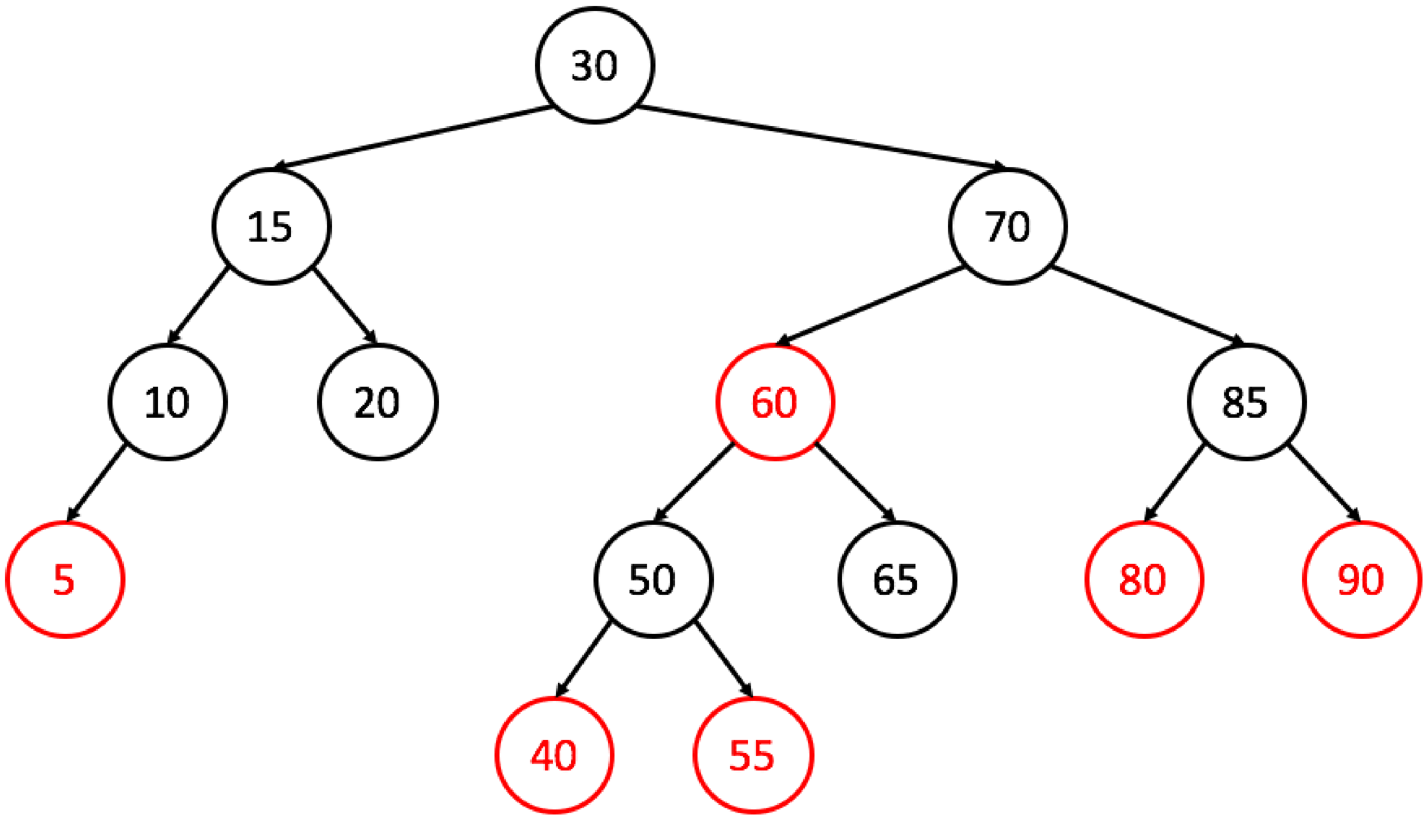
1. All nodes must be "colored" either **red** or **black**
2. The root of the tree must be **black**
3. If a node is **red**, all of its children must be **black** (i.e., we can’t have a **red** node with a **red** child)
4. For any given node *u*, every possible path from *u* to a "null reference" (i.e., an empty left or right child) must contain the same number of **black** nodes

Also, not quite a property but more of a definition, "null references" are colored **black**. In other words, if some node has a right child but no left child, we assume this "phantom" left child is colored **black**. Below is an example of a **Red-Black Tree**:



Step 3

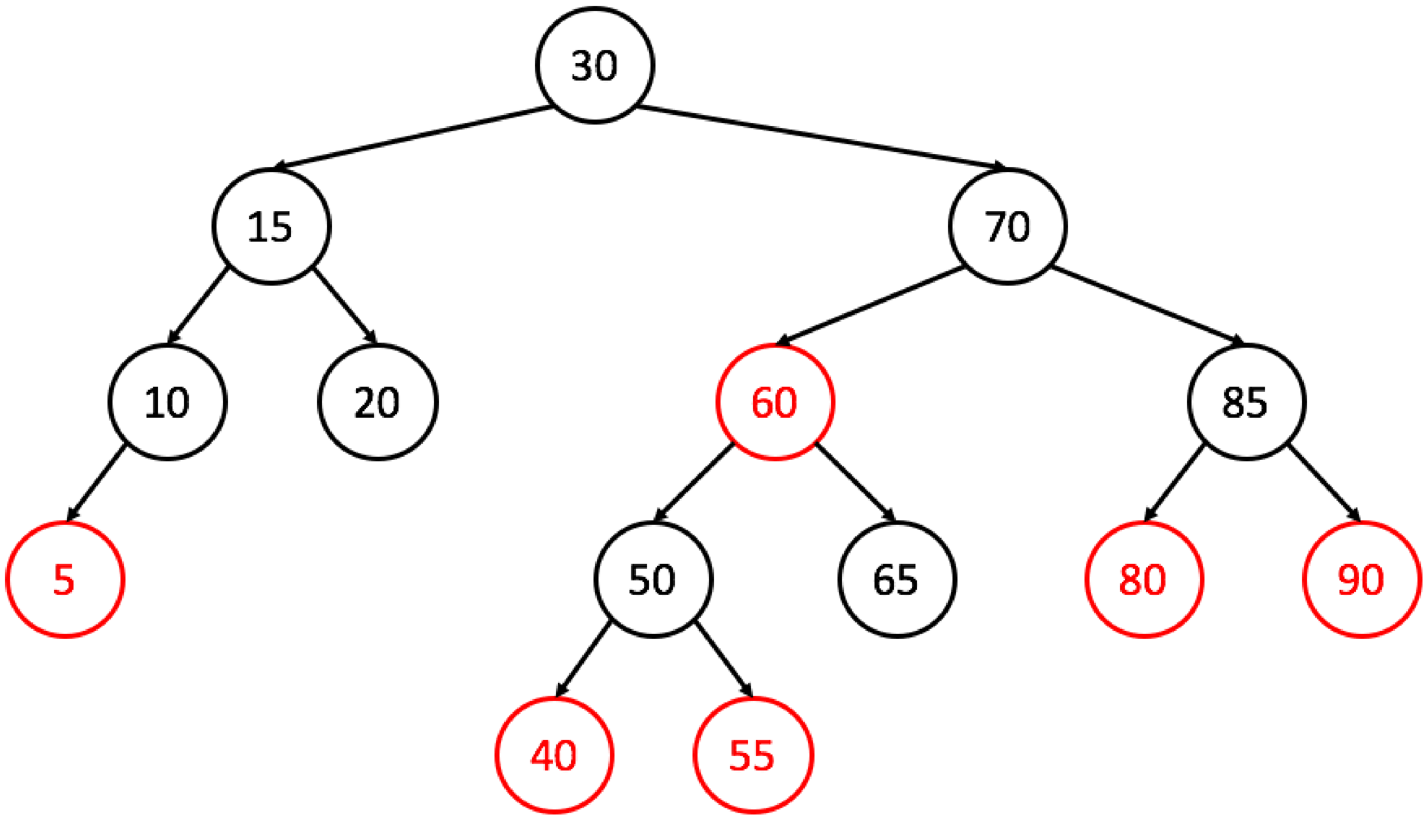
EXERCISE BREAK: The **Red-Black Tree** from the previous step has been reproduced below. How many **black** nodes are in the tree? Include null references (i.e., "phantom" children) in your count. Remember, both **red** and **black** nodes can have null references!



To solve this problem please visit https://stepik.org/lesson/28866/step/3

Step 4

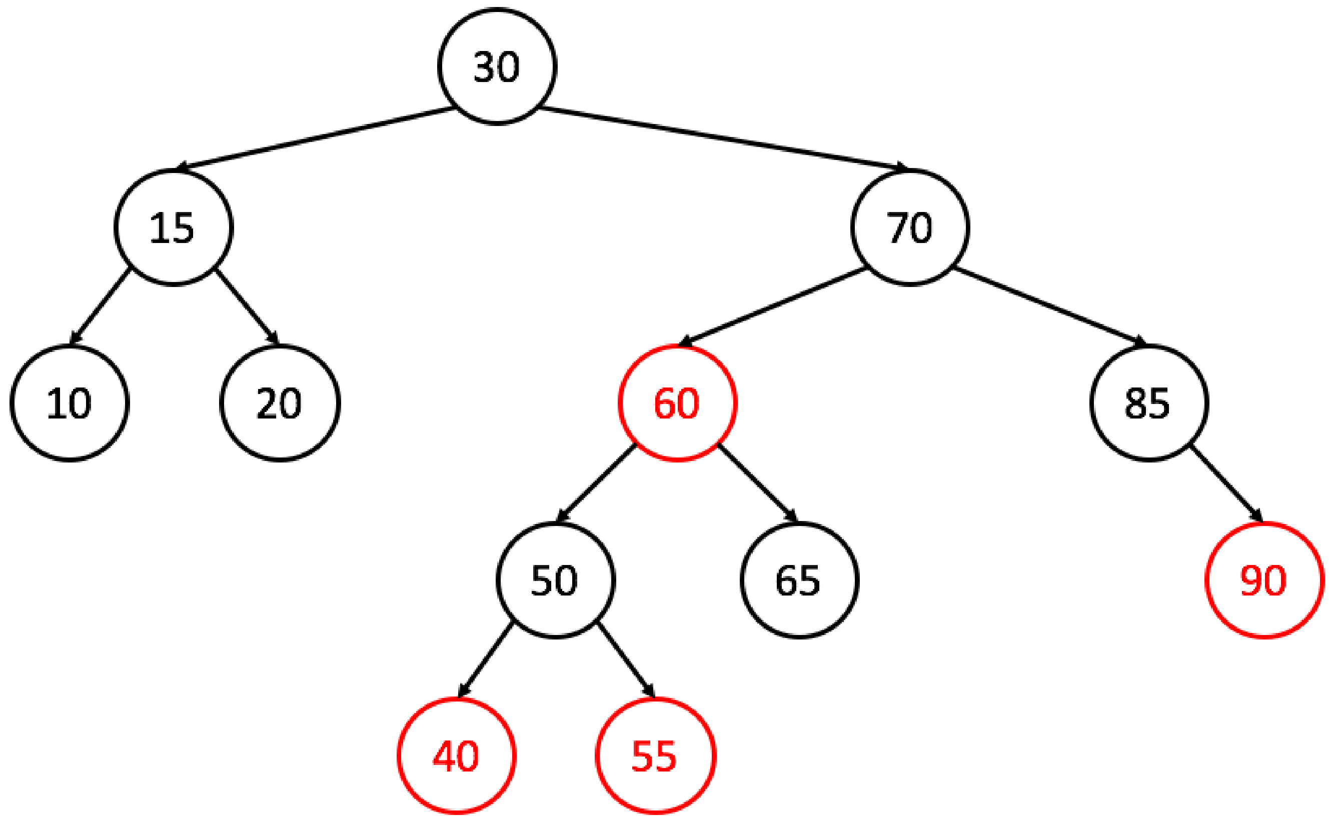
EXERCISE BREAK: The **Red-Black Tree** from the previous step has been reproduced below. If we were to ignore node colors, is the tree *also* an **AVL Tree**?



To solve this problem please visit https://stepik.org/lesson/28866/step/4

Step 5

EXERCISE BREAK: The following tree is a slightly modified version of the tree from the previous step. Which of the following statements are true? (Select all that apply)



To solve this problem please visit https://stepik.org/lesson/28866/step/5

Step 6

As you hopefully noticed, although *some* **Red-Black Trees** are *also* **AVL Trees**, *not all* **Red-Black Trees** are **AVL Trees**. Specifically, based on the example in the previous step, **AVL Trees** are actually *more balanced* than **Red-Black Trees**, as the **AVL Tree** balance restrictions are stricter than the **Red-Black Tree** balance restrictions. Nevertheless, we can formally prove that **Red-Black Trees** do indeed also have a **O(log *n*) worst-case** time complexity.

First, let's define $bh(x)$ to be the *black height* of node *x*. In other words, $bh(x)$ is the number of **black** nodes on the path from *x* to a leaf, excluding itself. We claim that any subtree rooted at *x* has at least $2^{bh(x)} - 1$ internal nodes, which we can prove by induction.

Our base case is when $bh(x) = 0$, which only occurs when *x* is a leaf. The subtree rooted at *x* only contains node *x*, meaning the subtree has 0 internal nodes (because *x* is a leaf, and is thus not an internal node by definition). We see that our claim holds true on the base case: $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes in the subtree rooted on *x*.

Now, let's assume this claim holds true for all trees with a *black height* less than $bh(x)$. If *x* is **black**, then both subtrees of *x* have a *black height* of $bh(x) - 1$. If *x* is **red**, then both subtrees of *x* have a *black height* of $bh(x)$. Therefore, the number of internal nodes in any subtree of *x* is $n \geq 2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 \geq 2^{bh(x)} - 1$.

Now, let's define *h* to be the **height** of our tree. At least half of the nodes on any path from the root to a leaf must be **black** if we ignore the root (because we are not allowed to have two **red** nodes in a row, so in the worst case, we'll have a **black-red-black-red-black-...** pattern along the path). Therefore, $bh(x) \geq \frac{h}{2}$ and $n \geq 2^{\frac{h}{2}} - 1$, so $n + 1 \geq 2^{\frac{h}{2}}$.

This implies that $\log(n+1) \geq \frac{h}{2}$, so $h \leq 2\log(n+1)$. In other words, we have formally proven that the **height of a Red-Black Tree is O(log *n*)**.

Step 7

EXERCISE BREAK: Which of the following statements are true? (Select all that apply)

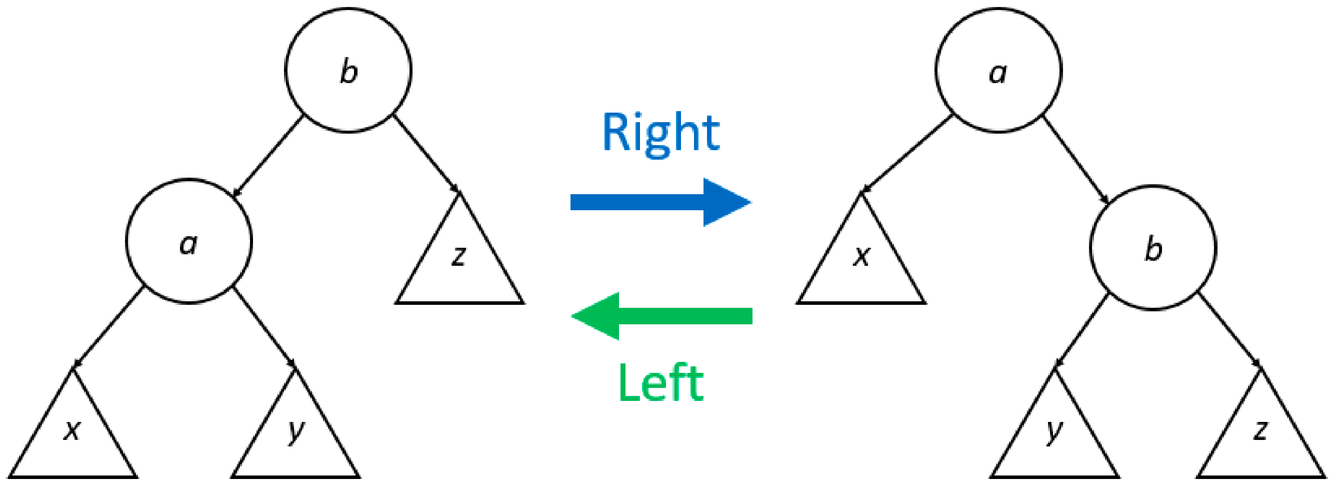
To solve this problem please visit https://stepik.org/lesson/28866/step/7

Step 8

The insertion and removal algorithms for the **Red-Black Tree** are a bit tricky because there are so many properties we need to keep track of. With the **AVL Tree**, we had one restructuring tool at our disposal: **AVL rotations**. Now, we have two tools at our disposal: **AVL rotations** *and* **node recoloring** (i.e., changing a **black** node's color to **red**, or a **red** node's color to **black**).

Before continuing with the **Red-Black Tree** algorithms, we want to first refresh your memory with regard to **AVL Rotations** because they are somewhat non-trivial.

Recall that **AVL Rotations** can be done in two directions: **right** or **left**. Below is a diagram generalizing both right and left AVL Rotations. In the diagram, the triangles represent arbitrary subtrees of any shape: they can be empty, small, large, etc. The circles are the "important" nodes upon which we are performing the rotation.

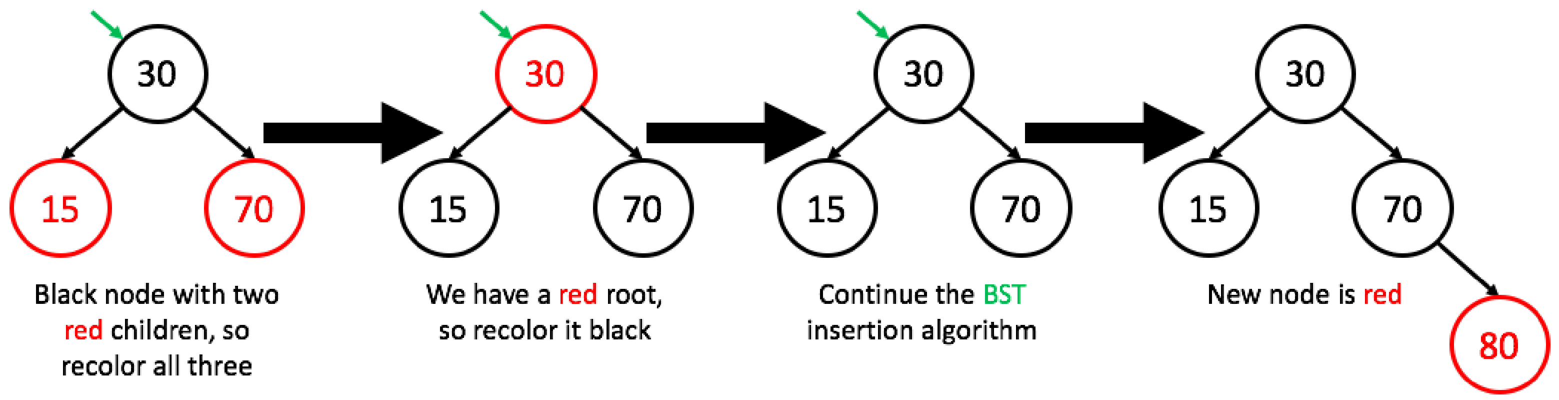


Step 9

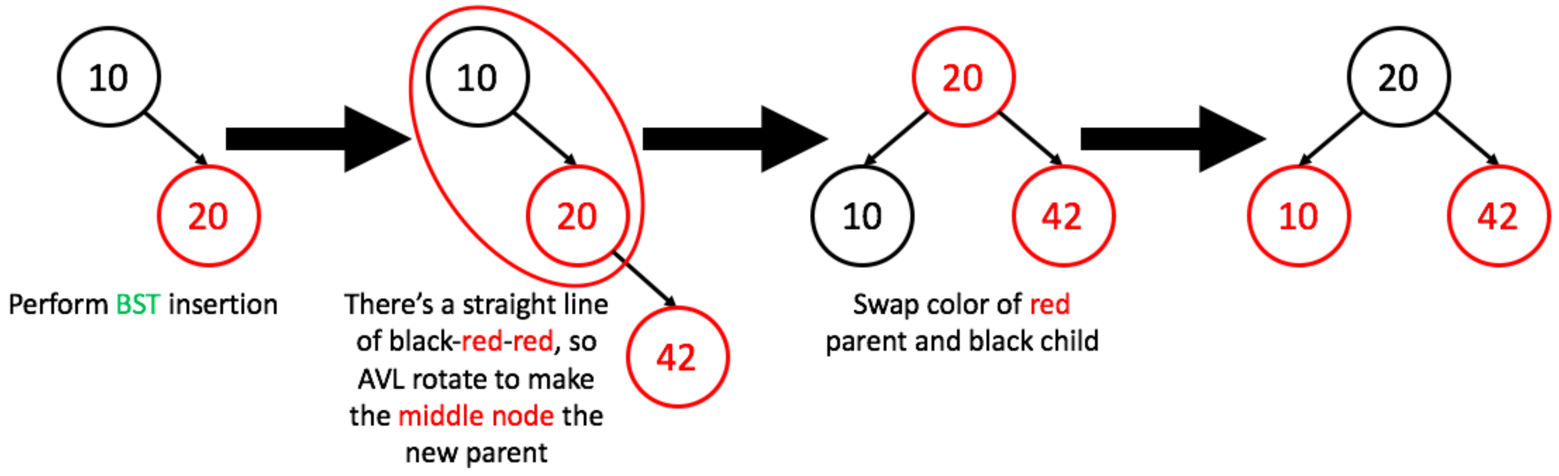
Before even going into the steps of the insertion algorithm, recall that the root of a **Red-Black Tree** *must* be colored **black**. As such, if we ever have a **red** root after an insertion, assuming it has no **red** children (which it shouldn't, assuming we did the insertion algorithm correctly), we can simply recolor the root **black**. Keep this in the back of your mind as you read the following paragraphs.

Also, our default color for newly-inserted nodes is always **red**. The motivation behind this is that, if we were to automatically color a newly-inserted node **black**, we would probably break the "same number of **black** nodes along paths to null references" restriction (Property 4) of the **Red-Black Tree**, and we would need to do potentially complicated restructuring operations to fix this property. By making newly-inserted nodes **red**, the number of **black** nodes in the tree is unchanged, thus maintaining the "**black** nodes along paths to null references" restriction (Property 4). Also keep this in the back of your mind as you read the following paragraphs.

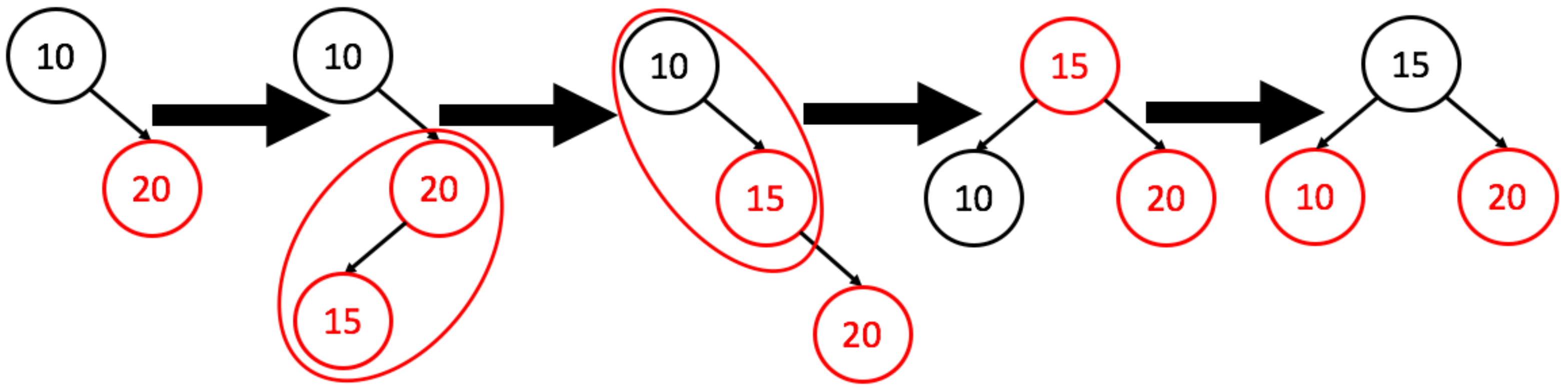
The first step to removing an element from a **Red-Black Tree** is to simply perform the regular **Binary Search Tree** insertion algorithm, with the newly-inserted node being colored **red**. If this new node is the first one in the tree (i.e., it is the root), simply recolor it **black**. Otherwise, during your traversal down the tree in the **Binary Search Tree** insertion algorithm, if you ever run into a **black** node with two **red** children, recolor all three nodes (i.e., make the **black** node **red**, and make its two **red** children **black**). Below is a simple example of this process, where we insert the number 80 into the following tree:


30
15
70
Black node with two red children, so recolor all three
30
15
70
We have a red root, so recolor it black
30
15
70
Continue the BST insertion algorithm
30
15
70
80
New node is red

In the example above, the newly-inserted node (80) happened to be the child of a **black** node, meaning we didn't violate the "**red** nodes cannot have **red** children" property by chance. However, what if we weren't so lucky, and the newly-inserted **red** node became the child of a **red** node? It turns out that we can fairly simply fix this problem using AVL rotations and recoloring. Below is an example of this process, where we insert the number 42 into the following tree:


10
20
Perform BST insertion
10
20
42
There's a straight line of black-red-red, so AVL rotate to make the middle node the new parent
20
10
42
Swap color of red parent and black child
20
10
42

That was pretty easy, right? We're hoping you're a bit skeptical because of what you remember seeing in the previous **AVL Tree** section. Specifically, this single AVL rotation was the solution because our nodes of interest were in a *straight line* (i.e., the **black-red-red** path was a straight line), but what about if they were in a *kink* shape (i.e., the **black-red-red** path had a kink)? Just like in the **AVL Tree** section, our simple solution is to just perform a first AVL rotation to *transform* our kink shape *into* a straight line, which we're comfortable with dealing with, and then just dealing with the straight line using a second AVL rotation. In other words, in the case of a kink, we need to perform a **double rotation**. Below is an example of this process, where we insert the number 15 into the same tree as before:

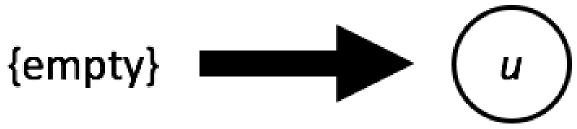


And that's it! That is the entirety of **Red-Black Tree** insertion. It would be a good idea to look at the "before" and "after" trees for each of the possible cases described above and verify in your own mind that the **Red-Black Tree** properties we mentioned at the beginning of this section remain unviolated after the new insertion.

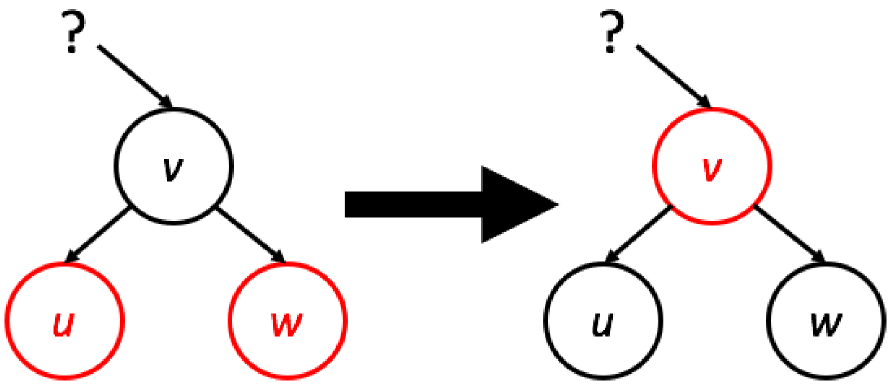
Step 10

To summarize, there are a handful of cases we must consider when inserting into a **Red-Black Tree**. In all of the following cases, the new node will be denoted as *u*. Also, a question mark denotes the rest of the tree outside of what is explicitly shown with nodes.

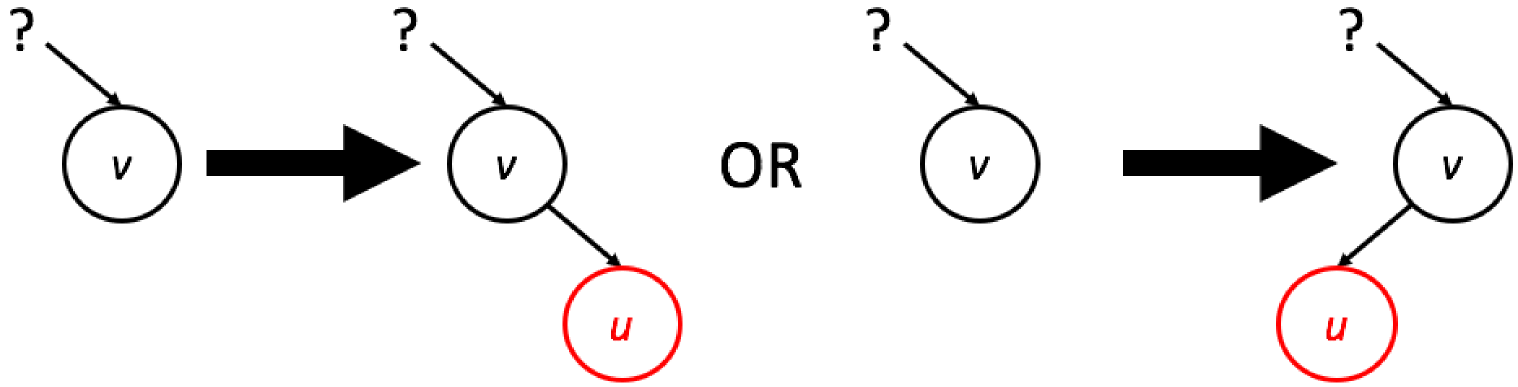
The most trivial case: if the tree is empty, insert the new node as the root and color it **black**:



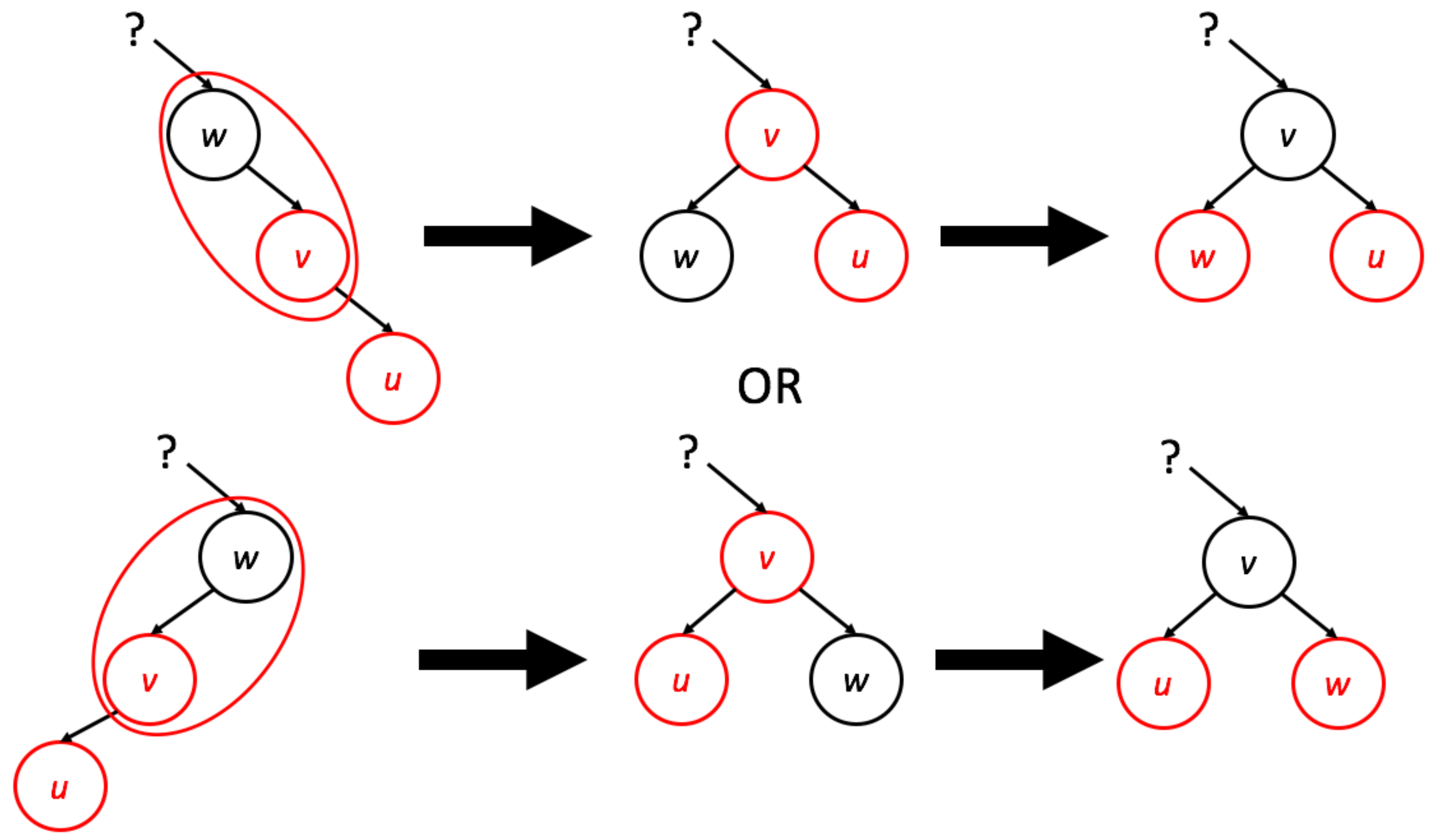
If the tree is not empty, insert the new node via the **Binary Search Tree** insertion algorithm, and color the new node **red**. While you're performing the **Binary Search Tree** insertion algorithm, if you run into a **black** node with two **red** children, recolor all three (and if the parent was the root, recolor it back to **black**):



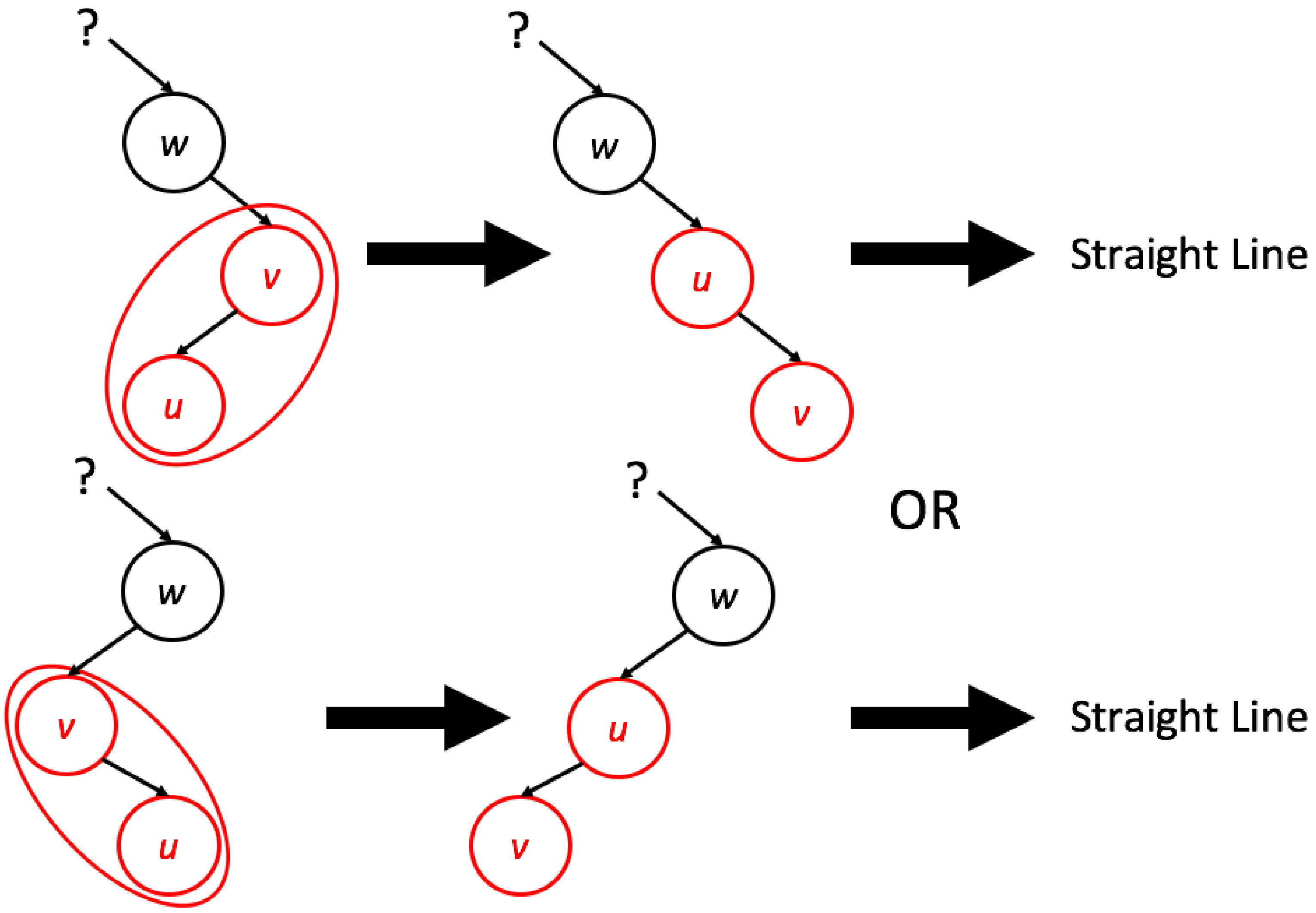
If the newly-inserted node is the child of a **black** node, we are done:



If the new node is the child of a **red** node, if the nodes are in a **straight line**, perform a **single rotation** and a **recoloring**:

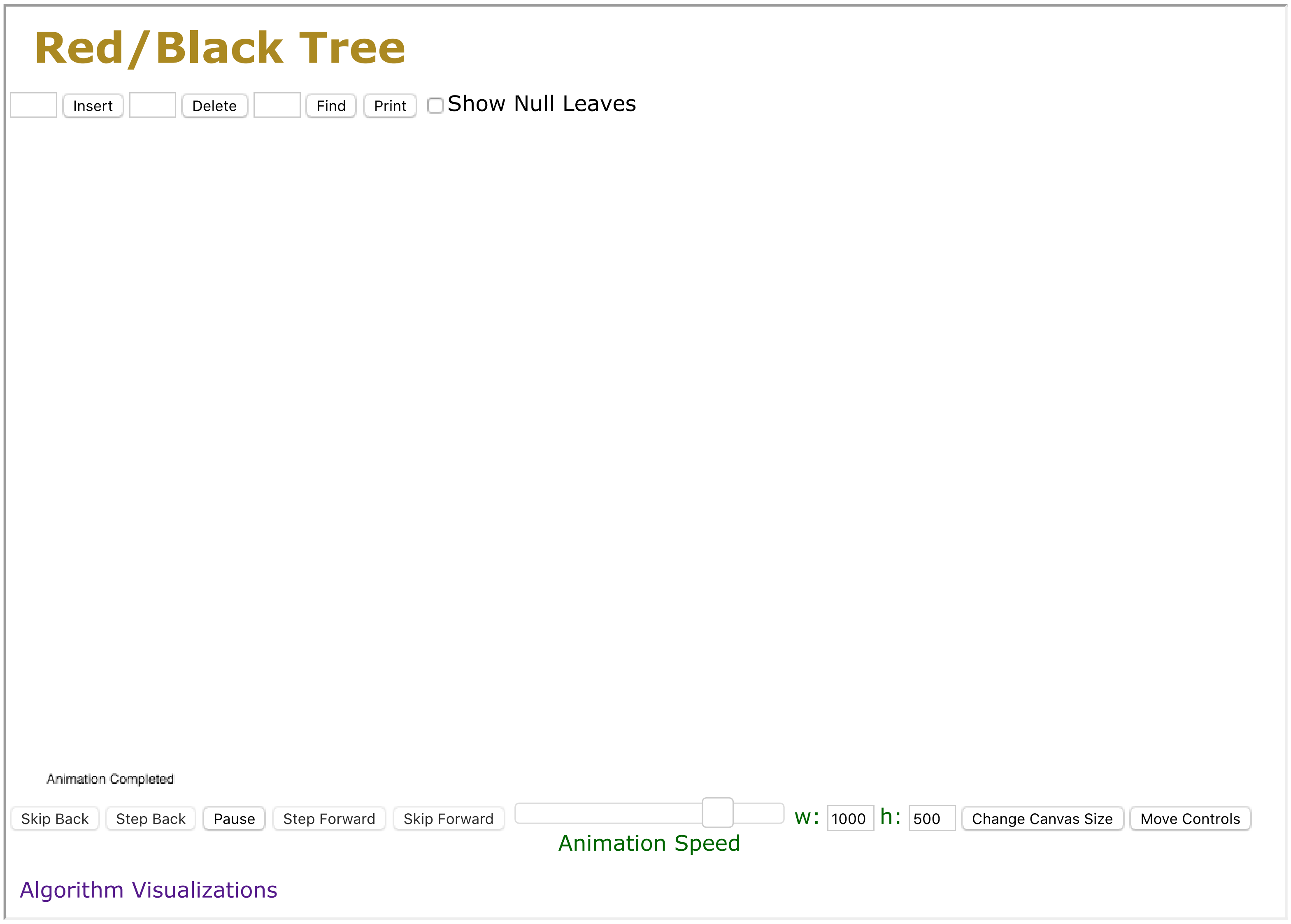


If the new node is the child of a **red** node, if the nodes are in a **kink**, perform a **single rotation** to transform them into a **straight line**, and then perform the **straight line** method above:



Step 11

Below is a visualization of a **Red-Black Tree**, created by David Galles at the University of San Francisco. Note that we did not cover the "remove" algorithm of a **Red-Black Tree** because the numerous cases that must be taken into account make it a bit too complex for us to want to dive into, but if you are interested, you can find a good summary here.



Step 12

We hope that you may be a bit confused at this point. We started this entire discussion with the motivation that we would be deviating from the **AVL Tree**, which is always pretty close to the optimal height of a **Binary Search Tree** (i.e., around **1 log *n***) and towards the **Red-Black Tree** for the sole intention of getting even better speed in-practice, yet we just formally proved that a **Red-Black Tree** can potentially be so out-of-balance that it can hit roughly twice the optimal height of a **Binary Search Tree** (i.e., around **2 log *n***), which would be *slower* in practice!

Our response to you is: fair enough. You are indeed correct that, given a **Red-Black Tree** and an **AVL Tree** containing the same elements, the **AVL Tree** will probably be able to perform slightly faster *find* operations in-practice. Specifically, in the absolute worst case (where a **Red-Black Tree** has roughly twice the height of a corresponding **AVL Tree**), the **Red-Black Tree** will take roughly twice as long to find an element in comparison to the corresponding **AVL Tree** (around **2 log *n*** vs. around **log *n*** operations). If this logic wasn't clear, just note that a find operation on any **Binary Search Tree** takes steps proportional to the height of the tree in the worst-case because we simply traverse down the tree, so since the **AVL Tree** has roughly half the height of the **Red-Black Tree** in the absolute worst case, it performs roughly half the operations to perform a find.

However, what about *insert* and *remove* operations? Using the same exact logic, if a **Red-Black Tree** is extremely out of balance, it will take roughly the same amount of time to remove or insert an element in comparison to the corresponding **AVL Tree** (around **2 log *n*** vs. around **2 log *n*** operations). However, if a **Red-Black Tree** is pretty balanced, it will take roughly *half* the time to remove or insert an element in comparison to the corresponding **AVL Tree** (around **log *n*** vs. around **2 log *n*** operations). If this logic wasn't clear, just note that an insertion or a removal operation on a **Red-Black Tree** only takes one pass down the tree, whereas an insertion or removal operation on an **AVL Tree** takes two passes (one down, and one back up). So, if the two trees are roughly equally balanced, the **Red-Black Tree** performs the insertion or removal using roughly half as many operations.

In short, we typically see **AVL Trees** perform better with find operations, whereas we typically see **Red-Black Trees** perform better with insert and remove operations. In practice, most data structures we use will be updated frequently, as we rarely know all of the elements we need in advance. As a result, because insertions and removals are actually very frequent in practice, it turns out that **Red-Black Trees** are the **Binary Search Tree** of choice for ordered data structures in many programming languages (e.g. the C++ `map` or `set`).

Step 13

With this, we have concluded our discussion regarding the various types of **Binary Search Trees**. It was quite a lengthy discussion, so below is a brief recap of what we covered.

We first introduced the **Binary Search Tree**, a binary tree in which, for any node *u*, all nodes in *u*'s left subtree are smaller than *u* and all nodes in *u*'s right subtree are larger than *u*. With this property, for balanced trees, we would be able to achieve an O(log *n*) time complexity for finding, inserting, and removing elements. We were able to formally prove that, although the **worst-case** time complexity of the three operations of a **Binary Search Tree** is **O(*n*)**, under some assumptions of randomness, the **average-case** time complexity is **O(log *n*)**. However, we also noted that the assumptions of randomness we made were somewhat unreasonable with real data.

Then, to remedy the unreasonable nature of our assumptions we made when we proved the O(log *n*) average-case time complexity of a **Binary Search Tree**, we introduced the **Randomized Search Tree**, a special kind of **Treap** in which *keys* are the elements we would have inserted into a regular **Binary Search Tree** and *priorities* are randomly-generated integers. Even though we didn't explicitly go over the formal proof in this text, it can be proven that a **Randomized Search Tree** does indeed simulate the same random distribution of tree shapes that came about from the assumptions of randomness we made in our formal proof for the O(log *n*) average-case time complexity. However, even though we explored a data structure that could actually obtain the **average-case O(log *n*)** time complexity we wanted, we were still stuck with the original **O(*n*) worst-case** time complexity.

To speed things up even further, we introduced the **AVL Tree** and the **Red-Black Tree**, two self-balancing **Binary Search Tree** structures that were guaranteed to have a **worst-case** time complexity of **O(log *n*)**. The balance restrictions of the **AVL Tree** are stricter than those of the **Red-Black Tree**, so even though the two data structures have the same *time complexities*, in practice, **AVL Trees** are typically faster with find operations (because they are forced to be more balanced, so traversing down the tree without modifying it is faster) and **Red-Black Trees** are typically faster with insertion and removal operations (because they are not required to be as balanced, so they perform less operations to maintain balance after modification).