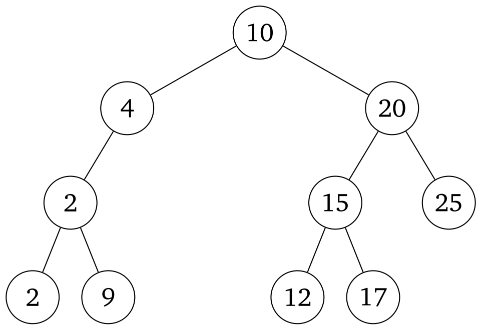
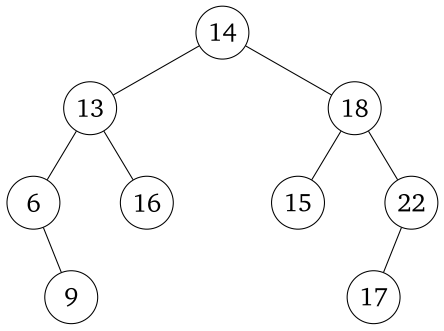
Red-Black Trees

Binary search trees

You hopefully remember a bit about binary search trees from Data Structures. A binary search tree (BST) is a binary tree with the following property:

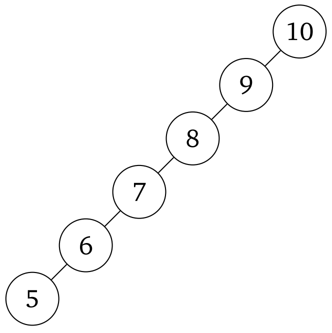
> For any given node N, each element in its left subtree is less than or equal to the value stored in N and each element in its right subtree is greater than the value stored in N.

The tree below on the left is a BST. The tree on the right is not. The value 17 at the bottom right is larger than 14 and is to the right of 14. That can't happen in a BST. It's also to the right of 18, which can't happen. Further, there's a 16, which is greater than 14, to the left of 14, which also isn't allowed.

The purpose of a BST is to store data in a kind of sorted order that makes it quick determine if something is in the tree, quick to find the max and min, and quick to return the elements in order. It is also quick to add and remove elements. All of these operations run in $O(\log n)$ time, provided the tree is balanced.

That's where we left things in Data Structures. But if we're not careful, the BST can quickly become unbalanced. This is where one side of the tree is significantly longer than the other. For instance, suppose we add the values 10, 9, 8, 7, 6, 5 to a BST in that order. The tree ends up not looking very treelike, as shown below.



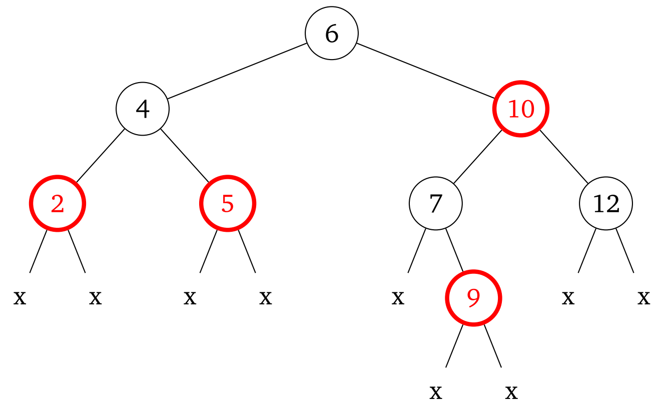
It is actually quite common to have real-life data initially be in sorted or reversed order like this. Since there's no branching structure here, the running times will degenerate from $O(\log n)$ to $O(n)$. There are a few tricks people have come up with for preventing this. The one we will cover is called a *red-black tree*.

Red-black trees

A red-black tree is a BST where each node is colored either red or black. The nodes have to meet the following additional properties:

1. The root and the null leaves at the bottom of the tree are colored black.
2. A red node cannot have a red child.
3. Every path from the root to the null leaves at the end of the tree has the same number of black nodes on it.

Shown below is a red-black tree. In case you're viewing this in black-and-white, the red nodes have thicker boundaries. The null leaves are shown with x's. In a BST, the null leaves are used to indicate the end of the tree.



Notice that the three rules are satisfied because (1) the root and the null leaves are all black; (2) no red node has a red child (though red grandchildren are okay); and (3) every path from the root to a null leaf contains the same number of black nodes (namely exactly 3 black nodes, 2 if you don't count the null nodes). We will not show the null nodes in future figures, but know they are there.

As we add new nodes to the tree, we will have to do certain operations to make sure the three red-black tree properties are preserved. Here is the procedure to add a new node:

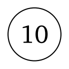
1. The very first node added to a red-black tree is black.
2. Every other node is red when we first add it. We follow the ordinary BST rules when placing that node (go left if less than or equal, right otherwise).
3. If the new node ends up having a black parent, then nothing needs to be done.
4. If the new node has a red parent, then we will have to do some recolorings and rotations of elements in order to keep the properties of the tree satisfied.

Following this process in order to satisfy the properties of a red-black tree will have the effect of keeping the tree balanced so that one side never gets very much deeper than the other side. Below we will try out an example where we add the values 10, 9, 8, . . . , 1 in order. For an ordinary BST, this would produce a long string of nodes instead of a proper tree, and we'll see how the red-black properties prevent that from happening.

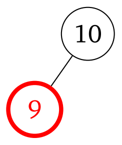
At most of the steps, we will run into problems where the red-black properties are broken. We will fix them by doing recolorings and rotations. The rotations, in particular, will keep the tree balanced. After this example, we will give the rules for when to do a rotation and when to do a recoloring. Please read through this example, but don't expect to understand everything on the first pass. If you want to play along with this example, the site below has really nice animations of how red-black trees work:

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html

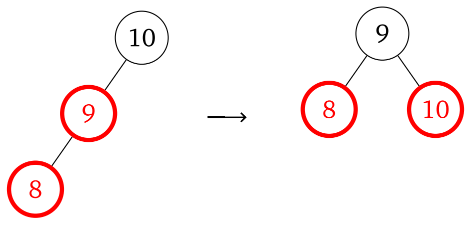
Add 10. The first node added is always black.



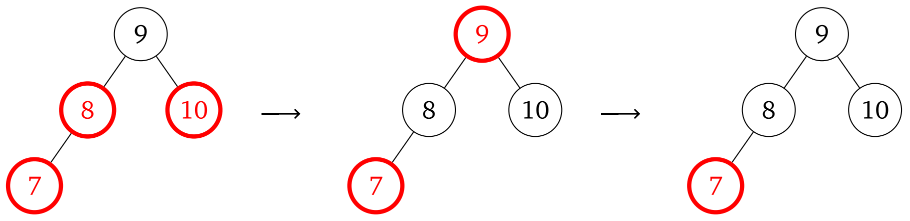
Add 9. New nodes are always red. Since 9 is less than 10, it goes to the left of 10, like below.



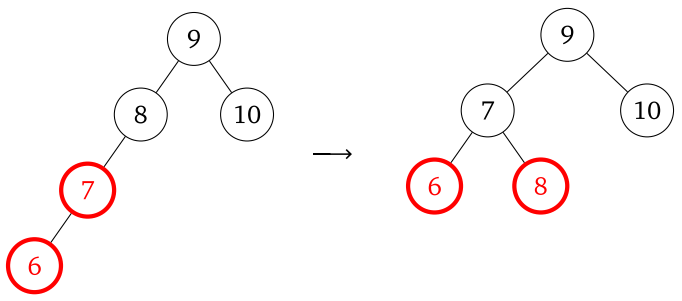
Add 8. It's a new node, so it's red, and it goes to the left of 9. Now we have a problem because we have two red nodes in a row. The solution to this problem is to do a rotation and recoloring. We rotate the three nodes clockwise as shown. In particular, 10 shifts down and right, 9 shifts up to the root, and 8 shifts up. Remember that the root must be black, so we have to recolor 9 to black. This means we have to color 10 red in order to satisfy the rule about every path from the root to the end of the tree having the same number of black leaves.



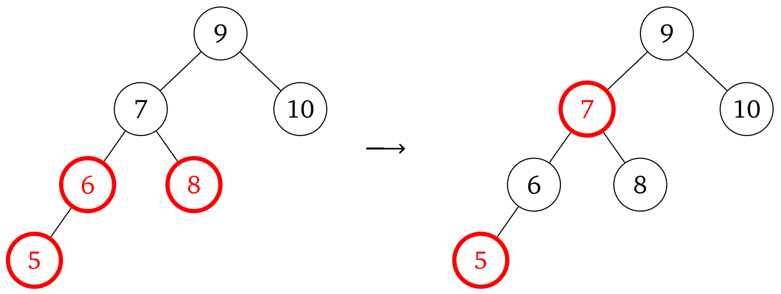
Add 7. It goes left of 8. This causes another problem with two reds in a row. When both the parent of the new node and its uncle (the sibling of its parent) are red, the solution is to recolor both the parent and uncle black and color the grandparent red. This is shown in the middle below. However, this breaks the rule about the root needing to be black, so we recolor the grandparent black, as shown on the right.



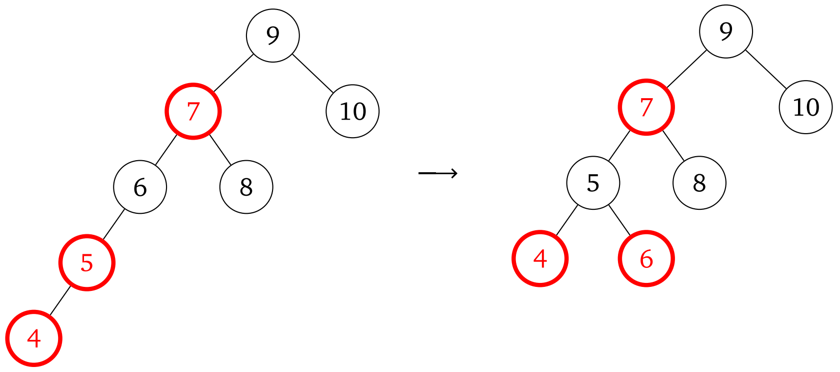
Add 6. Again we get an issue with two reds next to each other. The solution here is a rotation. Rotations are the solution whenever the new node's parent is red and its uncle is black (or null). The rotation involves the new node, its parent, and its grandparent all rotating clockwise. We also recolor things so that the parent and grandparent switch colors. This is shown on the right below. Notice that this rotation/recoloring preserves all of the properties of red-black trees.



Add 5. The new node goes left of 6. We end up with two red nodes next to each other. To fix this issue, since both the parent and uncle are red, we can change them to black and the grandparent to red, and then the red-black rules are all satisfied.



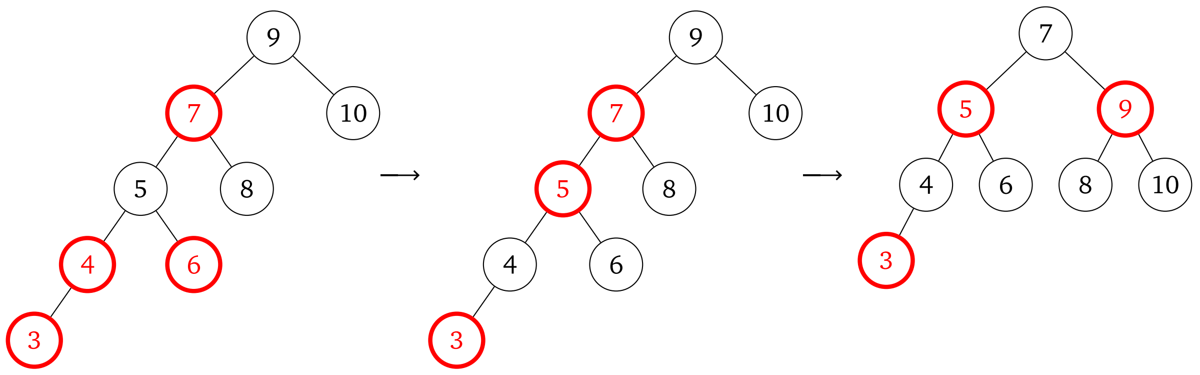
Add 4. The new node goes left of 5. We will need to do a rotation/recoloring to fix the problem.



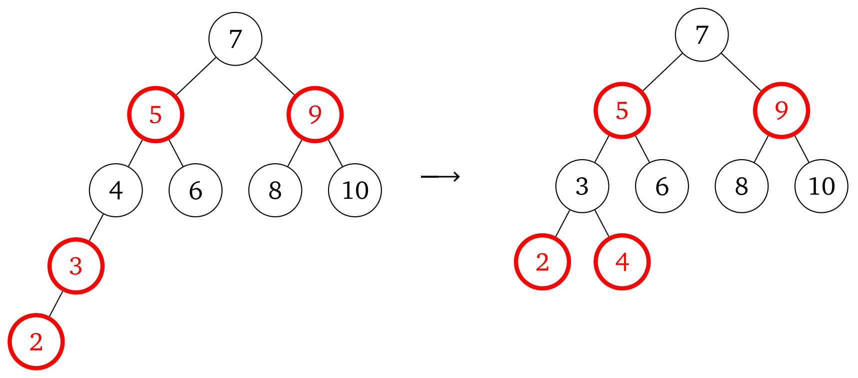
Add 3. The new node goes left of 4. As both its parent and uncle are red, we swap their colors with the grandparent's color. But in fixing the problem we introduce a new problem farther up. The 5 and 7 are now two red nodes in a row. To fix this, we do another rotation/recoloring. This is a little more sophisticated as it involves moving parts of the left half of the tree over to the right.

In this rotation, 5 is playing the role of the new node, 7 is playing the role of the parent, 9 is the grandparent, and 10 is the uncle. The rotation always involves moving the new node into the parent's place, moving the parent into the grandparent's place, and moving the grandparent into the uncle's place. This has the effect of making 9 the new right neighbor of 7, but 7 already had a right neighbor, namely node 8. We can fix this by moving 8 over to the right half of the tree as shown.

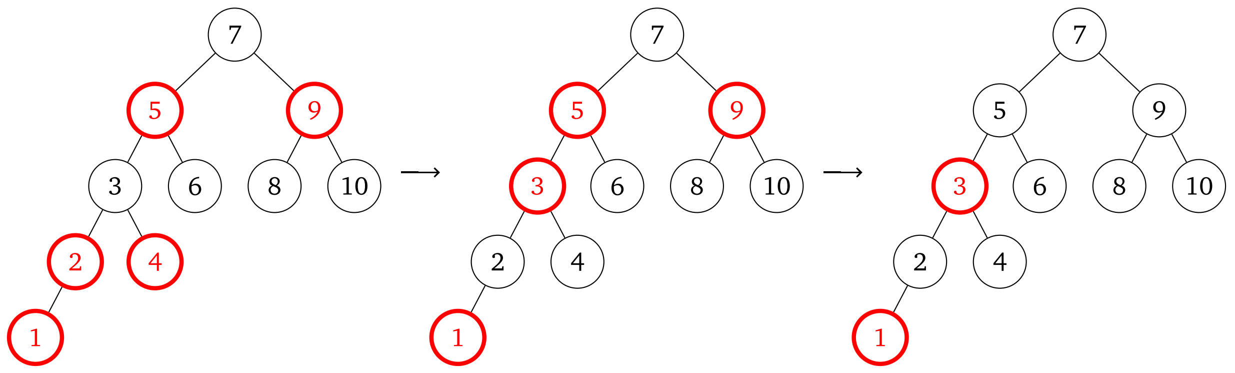
Finally, these rotations always have a recoloring component to them, where we swap the colors of the parent and grandparent (7 and 9 in this case).



Add 2. Things are a little simpler here. We have a rotation to do, but it only involves the bottom left of the tree.



Add 1. After adding 1, since both its parent and uncle are red, we do a color swap where the parent and uncle become black and the grandparent becomes red. This causes an issue further up the tree, where 3 and 5 are now red and next to each other. Because both the parent and the uncle of 3 are red, to fix this problem, we swap the parent's and uncle's color with the grandparent. This turns the grandparent red. However, since the grandparent is the root, it is not allowed to be red, so we turn it black.



Rules for insertion

We'll restate the rules again:

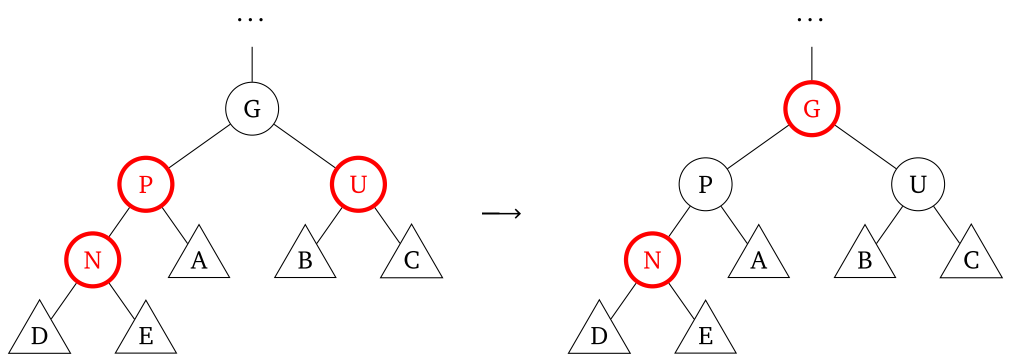
1. The very first node added to a red-black tree is black.
2. Every other node is red when we first add it. We follow the ordinary BST rules when placing that node (go left if less than or equal, right otherwise).
3. If the new node ends up having a black parent, then nothing needs to be done.
4. If the new node has a red parent, then we will have to do some recolorings and rotations of elements in order to keep the properties of the tree satisfied.

For recolorings and rotations, there are several cases. In each of the cases below, N is the new node. Node P is the parent of N. Node G is the grandparent of N, and U is the uncle of N (namely the sibling of P and the other child of G). What is shown in the figures below is just a portion of the tree. Assume the tree continues above G. Assume also that A, B, C, D, and E in the figures are not single nodes, but entire subtrees (which is why they are shown with triangles instead of circles).

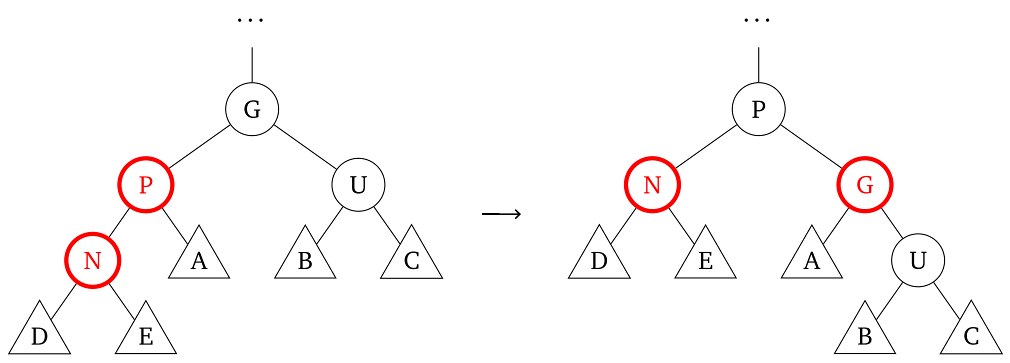
Note that we call N the "new node". Usually it's the node that was just added, but it may be an old node that got colored red as a result of some other recoloring. It will always be the bottom node of a pair of adjacent red nodes.

Case 1. Both P and U are red The solution here is to swap the colors of P and U with the color of G. Specifically P and U will turn black, and G will turn red. This may cause problems farther up the tree. If so, repeat the process at that point in the tree, going through all the possible cases. Note also that if G is the root of the tree, then G will need to be turned black because the root must always be black.

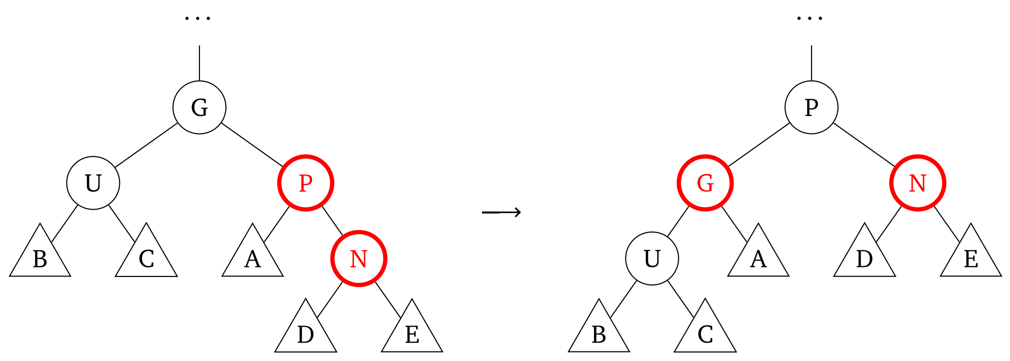
This case is shown in the figure below. This rule works no matter where N is being inserted, whether it's on the left, in the middle or on the right.



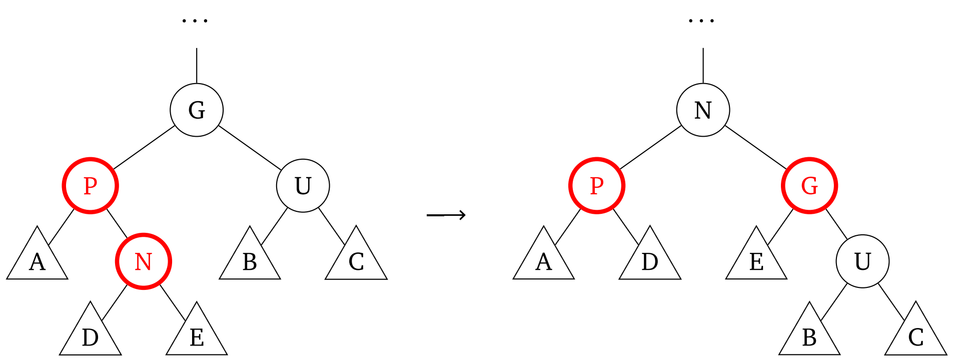
Case 2a. (left-left) We do a clockwise rotation, where N moves to P's position, P moves to G's position, and G moves to U's position. By doing this, the subtree A loses its home, but we can move it over to be left of *G*. After the rotation, we recolor P to black and G to red.



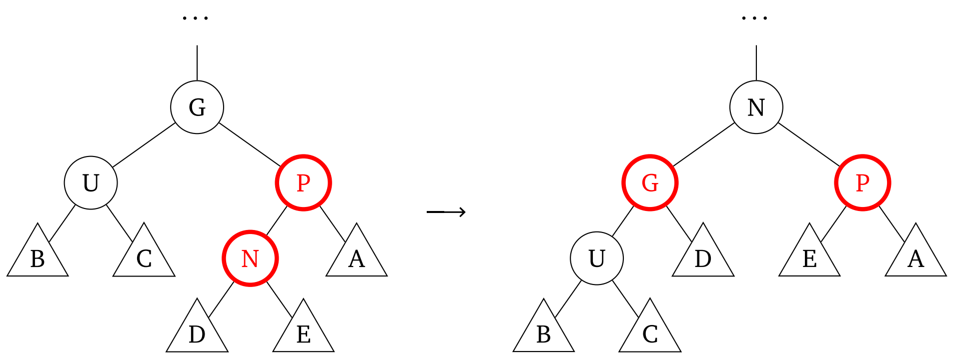
Case 2b. (right-right) This a the same as Case 2a, but with the rotation going counterclockwise.



Case 3a. (left-right) What happens here is actually a combination of two rotations. The end result is that N goes into G's spot, and G and U shift down. N's children need new homes, and they become children of P and G. The order is important here. Because D is a left child of N, it needs to become P's child and not G's in order to stay left of N in the new tree. Finally, we recolor N to black and G to red.



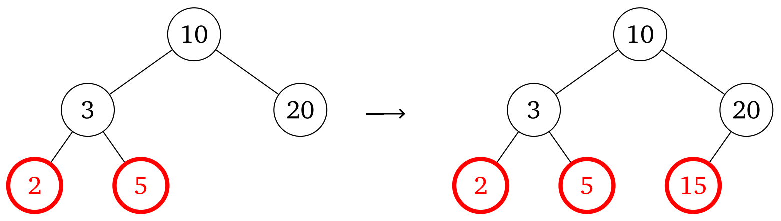
Case 3b. (right-left) This is the symmetric case from 3a. It is likewise a combination of two rotations. The net result is that N moves into G's place. G and U slide down, and we have to find new homes for N's children, which now become children of G and P. And we recolor G to red and N to black.



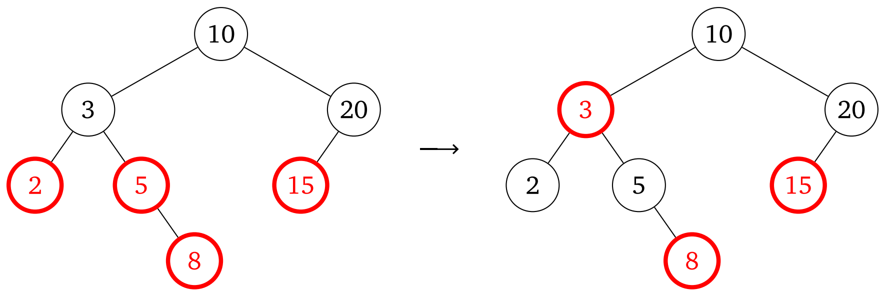
To summarize, all of these rules are used when we have to fix the situation of two red nodes in a row. If both the parent and uncle are red, then use Case 1. Otherwise, the other cases correspond to four possible configurations of P and N in relation to G: left-left (Case 2a), right-right (Case 2b), left-right (Case 3a), and right-left (Case 3b). The trick is to identify each of the parts N, P, U, G, A, B, C, D, and E (some of which may be null), and apply to appropriate operation. Below we will try a few more examples.

A few more examples

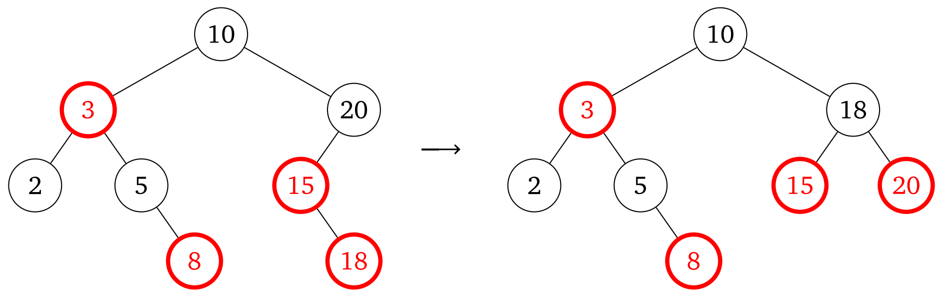
Suppose the tree looks like below on the left, where we are inserting 15. Following BST rules (left if less, right if greater), we find the right place to insert it is to the left of 20. New nodes are always inserted red, and inserting it causes no adjacent red nodes, so there's nothing else to do.



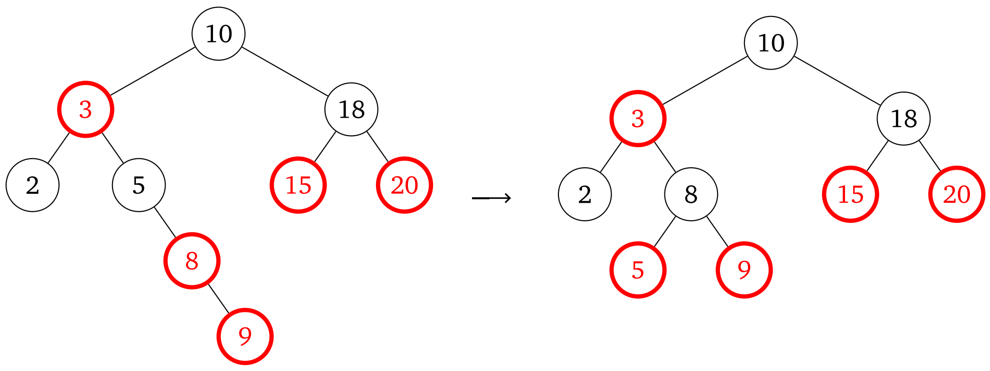
Let’s now add 8 to the tree. It will go to the right of 5, like below on the left. We now have two reds in a row. Both the parent and the uncle of the inserted node are red, so to fix the problem, we use Case 1, which is to change the parent and the uncle to black and the grandparent to red.



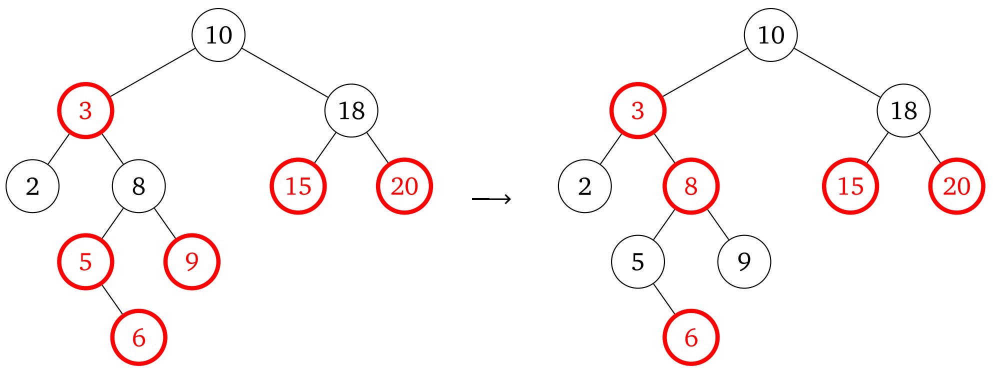
Now let’s say we want to add 18 into the tree. It will go to the right of 15, as shown below on the left. We have two adjacent red nodes in a left-right orientation, so this is Case 3a. Here N is 8, P is 15, G is 20, and the U is a null leaf. According to Case 3a, N will move up to G’s place and G will slide down. If you look at the figure for Case 3, there are various subtrees A, B, C, D, and E. They are all null in this case, so after moving N and G there’s not much else to do, other than recolor G and N as required by Case 3a.



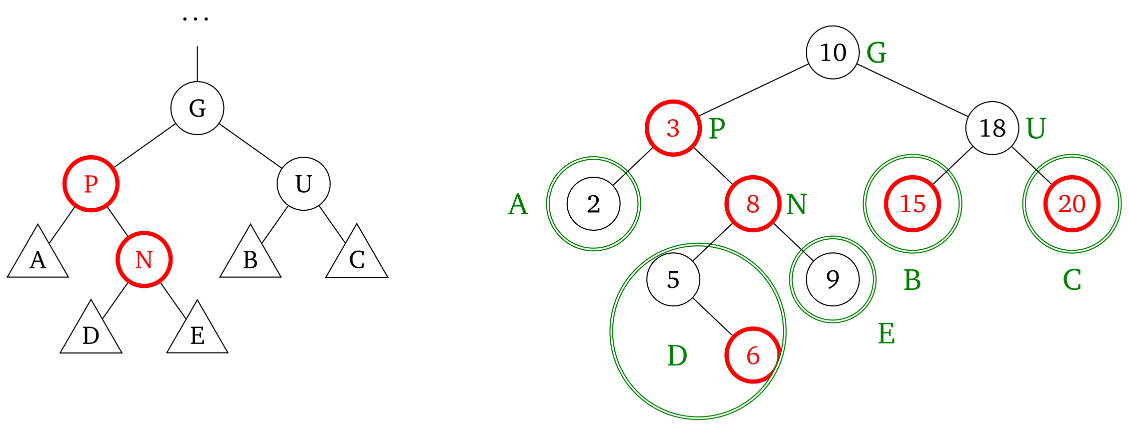
Now let’s insert 9. It goes to the right of 8. This gives us two red nodes in a row. These are in a right-right orientation, so it’s Case 2b. We have N=9, P=8, and G=5. The subtrees A, B, C, D, and E are all null, as is U, so all we have to do is rotate N, P, and G counterclockwise and recolor, as shown on the right.



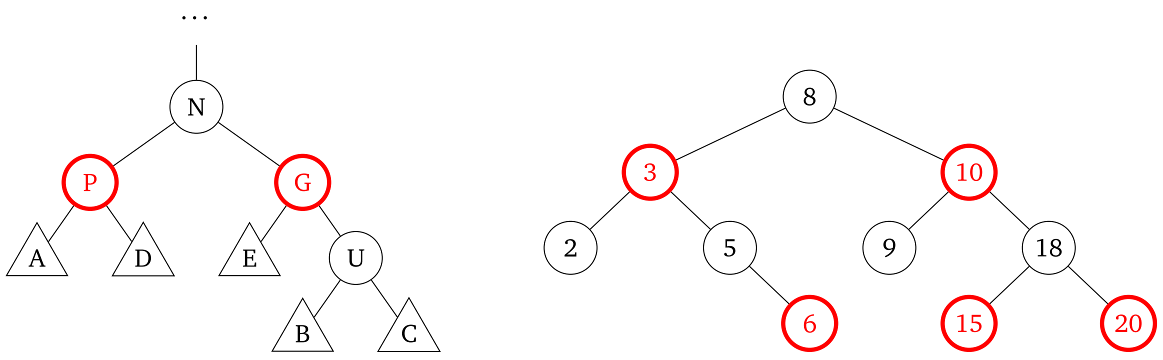
Finally, let’s try adding 6. This is more complicated than the previous steps. First, 6 will go to the right of 5, as shown below on the left. This gives us two red nodes in a row, which we can solve with Case 1 since both the parent and the uncle of the new node are red.



However, this creates a new problem farther up the tree as now 8 and 3 are adjacent red nodes. This is Case 3a, as the orientation is left-right. The figure for Case 3a is shown on the left, and the various part of the tree are labeled on the right according to Case 3a on the right.



Below on the left is the operation we are supposed to do for Case 3a, and on the right is the result of doing that.



Conclusion

Why cover this? It's a well-known data structure that's good to know a little about. We won't go any more in depth than this. In particular, we won't talk about deleting vertices (which is similar, with various cases). We also won't worry about how to code it.

Once more, the site below is nice for playing around with red-black trees to help understand how they work.

https://www.cs.usfca.edu/~galles/visualization/RedBlack.html