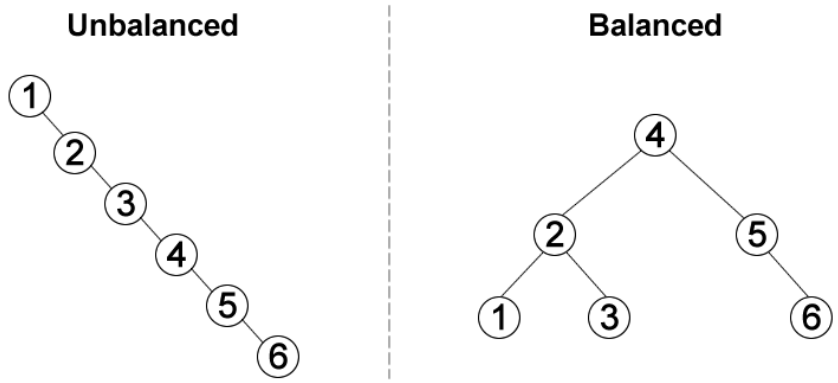
CSC 325 Adv Data Structures

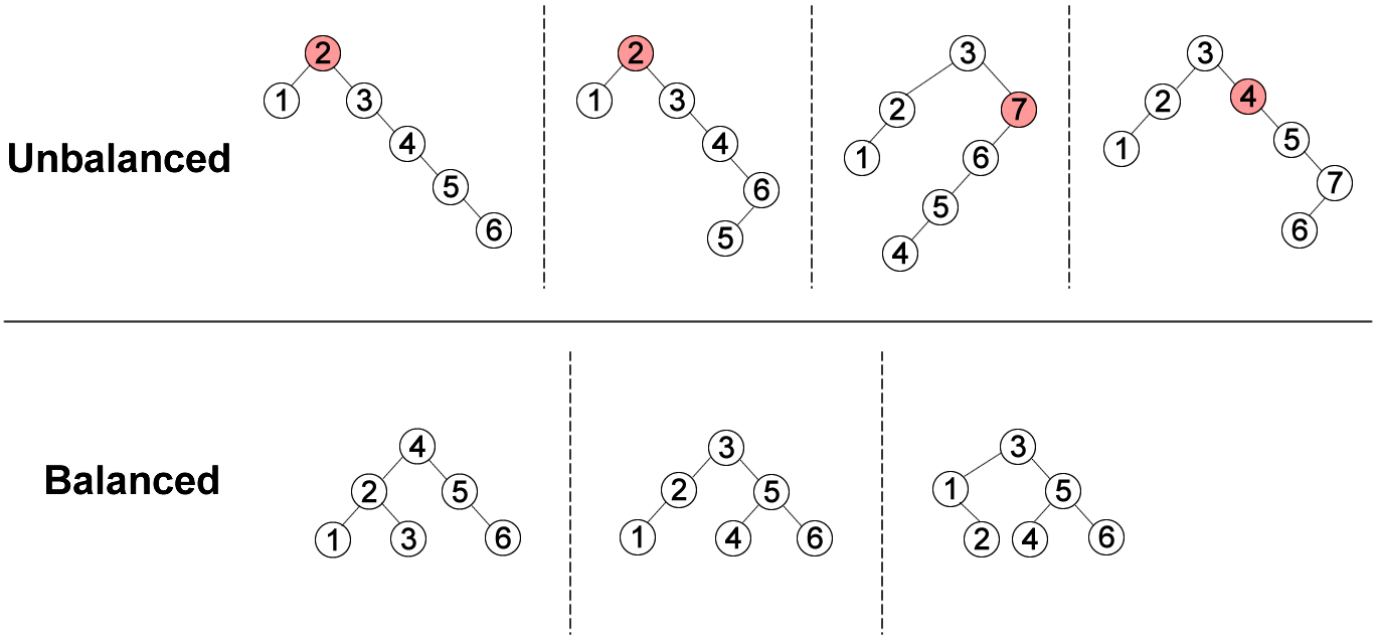
Lecture 11

Red Black Trees

In CSC 220, you learned about 2-3 trees, which is a type of tree that remains balanced. Recall that a balanced tree is much better in real time performance than an unbalanced tree. Consider the following diagram:



If you were to search for the number 6 in both of these binary search trees, which one would take longer? Obviously the unbalanced tree. In this tree, we must search 1, 2, 3, 4, 5 and then finally find 6 at the very bottom. On the other hand, searching the balanced tree just requires searching 4, 5 and then finding 6. This different in time is small for this example (searching 6 nodes instead of 3 nodes), but this gets HUGE as the tree grows. Essentially, this is the difference between O(n) in the unbalanced case and O(lg n) in the balanced case. Think about if n was 1,000,000. We would search 20 nodes in the balanced tree and all 1,000,000 nodes in the unbalanced tree. This means that the unbalanced tree would take 50,000 times longer! If we consider that, for example, each node we check takes 1 millisecond, we would have 20 milliseconds (or 1/50th of a second) in the balanced tree and 1,000 seconds (or ~16.5 minutes) in the unbalanced tree. Now, this is an extreme example and not all unbalanced trees are this bad. Consider the following trees:



There are many different ways a tree can present as unbalanced. It is important to define this so that we can definitively say whether a tree is balanced or not. I'm going to use the following definition: If for any node, the absolute difference between the height of the shortest path to a null node and its longest path to a null node is more than 2 times, the tree is unbalanced. We will see this definition again when talking about Red Black trees. In the unbalanced trees above, the node colored in red has a difference in the height of its left and right subtrees of more than 2x. For example, the third tree has node with value 7 highlighted in red. This node has a height of 4 for its left subtree (if we count the null children of node 4), but a height of 1 for its right subtree (i.e. again counting a null child). Clearly 4 is more than two times 1, so this tree is unbalanced. Note that height doesn't just mean the absolute number of nodes. It means the number of levels we can descend from the node. So, for example, if node 6 had a right child, this wouldn't affect our count since it isn't on the path we are considering.

We can see that, although a tree is either balanced or unbalanced, there is a good bit of difference in speed for unbalanced trees. Not all of them are so horrible in performance. However, a balanced tree will always outperform an unbalanced tree in general and the time complexity still shows a difference of linear vs logarithmic time. So, the motivation behind having a balanced tree should be clear.

Naming Convention

Before we introduce our data type, I want to talk about naming conventions. I still see some students referring to all trees as binary search trees or all heaps as a min heap, etc. This is not only bad practice, it is outright incorrect. A "binary search tree" is called that because it is a "tree", which has a parent child relationship among the nodes that forms a hierarchy. It is a "binary tree", which has a limit of at most 2 child nodes per parent node. And it is a "search tree", because there is a search property that imposes order on the values in the nodes, allowing for easy and efficient searching (in this case lesser values go to the left and greater values go to the right). These properties when put together form what is known as a "binary search tree". Not every tree meets these requirements. For example, consider a 2-3 tree. This is definitely not a binary tree, as nodes can have either 2 or 3 children. In fact, the name "2-3 tree" implies the number of children a node can have. A 2-3-4 tree (and yes this is a thing) contains nodes that have either 2, 3 or 4 children (and, in turn, can have 1, 2 or 3 keys in the node). Both of these (the 2-3 tree and the 2-3-4 tree) are types of "B trees" which are characterized by allowing more than 1 key inside of a node, a property that is not present in all trees (e.g. other trees, such as a binary search tree, only allows a single key per node). A data structure's name is a very important indication of the capabilities, restrictions, and properties that it honors. It is important to get the name right so others can understand exactly what you are talking about.

Red-Black Trees

A red-black tree builds upon the properties of a binary search tree and adds an extra characteristic to all nodes. Each node is either 'red' or 'black'. This property allows the tree to balance itself after insertion or deletion operations. This isn't an easy tree to understand, so let's take things one at a time. First, here are the operations we will go over:

Abstract Data Type: Red-Black Tree

`insert(value)`	Inserts the given value into the tree
`search(value)`	Searches for the given value in the tree. Returns true if found, false otherwise
`min()`	Returns the minimum value in the tree
`max()`	Returns the maximum value in the tree
`size()`	Returns the number of nodes in the tree
`inorder()`	Inorder traversal through our tree

A search tree, such as a binary search tree, has a "search property" (i.e. lesser values go to the left, greater values to go the right). A max heap has a "max heap property" (i.e. value of parent is greater than both children). A min heap has a "min heap property". A red-black tree has, you guessed it, a "red-black property". Actually, it has several properties, so more like "red-black properties". So, in addition to all the properties of a binary search tree, the red-black tree adds the following properties:

1. "**Balanced property**": The height of the tree is at most 2 * lg (n + 1). A faster way to tell if a Red Black tree is balanced is to see if the height of the longest path from any node to a leaf is 2x or less the height of the shortest path from node to leaf. This is the same definition we used previously. As an example, if the closest leaf to a node is 2 nodes away, the furthest leaf to that same node is no more than 4 nodes away.
2. "**Color property**": Every node is colored as either 'red' or 'black'
3. "**Root property**": The root is always black
4. "**External property**": All leaf (aka "external") nodes are null and all null nodes are consider black. This means that if there is no root, it is still considered black (as a null node) and the root property still holds. This also means that we must consider the null children to be black of a node that has no actual child nodes. This doesn't really change any memory allocation (a null node doesn't really exist), it just changes how we perceive the color of a null node which you will see an example of later.
5. "**Depth property**": Every path from the root to any null node (aka a leaf) goes through the same number of black nodes. So, for example, if the shortest path from root to a null node touches 3 black nodes, then the longest path will also touch exactly 3 black nodes (it may have more red nodes to make up the length difference and it may not be the same black nodes that it encounters, but there will be 3 of them). Also, any path in between the longest and shortest will also touch exactly 3 black nodes. Whether you consider the root node in this count (since it is black) or not really doesn't matter since all paths start at the root.
6. "**Red property**": There are no two adjacent red nodes (i.e. a red node cannot have a red parent or a red child)

Notice something interesting about the relationship between the depth property and red property? They seem to prove the balanced property. Think about this. Given the depth property, if the shortest path from root to null touches 3 black nodes, then the longest path must also touch 3 black nodes. Given the red property, there cannot be two consecutive red nodes (i.e. a parent and its child both being red is not allowed). This means that the longest path would have red nodes in between the black nodes making it longer. The absolute shortest a path can be would be to have all black nodes. The absolute longest a path can be would be to have alternating red and black nodes. This means the number of black nodes would equal the number of

red nodes in the alternating path. If the number of black nodes is k, then the number of red nodes would also be k, leading to a path length of 2k, which is exactly the max given to us by the balanced property! This is very interesting and follows quite logically. You may say, well what about if the root is red and the leaf is red, then we would have 2k + 1 red nodes (still alternating) and that would break the balanced property. Actually, due to the root property and the external property, this can NOT happen. So red nodes must be internal. This perfectly caps the number of red nodes to be within bounds. It seems as though these properties all come together to prove the balanced property of the tree.

Search, Min, Max and Size

The search, min, max and size operations all function exactly the same as they do in a regular BST. So these are not worth going over. You should remember exactly how these work from CSC 220. Note that for the size operation, you can simply keep track of a counter while doing other operations. So instead of traversing the tree and counting the nodes (which would take linear time), we can simply keep track of the counter and return it (which would take constant time).

Inorder

Recall that an inorder traversal follows the pattern of:

- L - Visiting the left child
- N - Processing the node's value
- R - Visiting the right child

This is done recursively and starts at the root. The "processing" part depends on what you want to do (e.g. printing the value or concatenating it to a string). In the end, an inorder traversal will process the nodes in sorted order. This is the same for a Red Black tree as it is for a Binary Search tree.

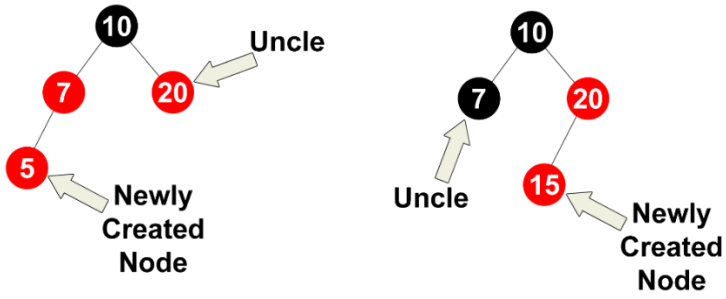
Insert

The insertion operation is quite interesting. Before we get into this operation, it is important to talk about an addition to the nodes in this tree. For proper implementation, you will absolutely need to store a pointer to the parent for each node. This means a node will have its data, its color, pointer to left child, pointer to right child, and pointer to parent. This also means you must update the parent pointer any time the node moves around the tree. Failure to do so will result in the next operation possibly failing.

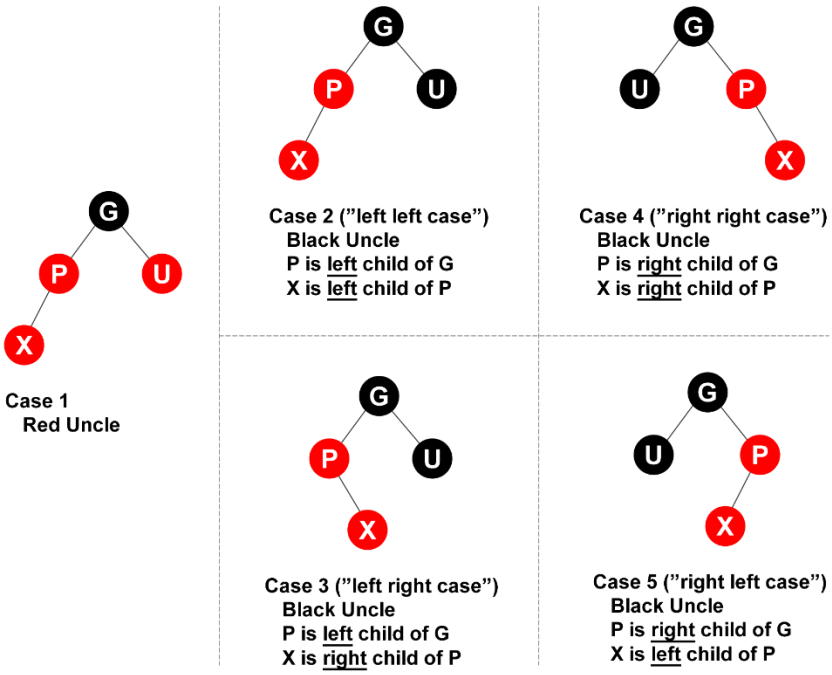
For insertion, first you insert the value just like you would in a normal binary search tree. You then color the newly inserted node red. By coloring the node red, two properties could now be violated, so we must check:

- If the root is red, i.e. we just created our root -> violation of the root property
- If the parent of the newly created node is red -> violation of the red property

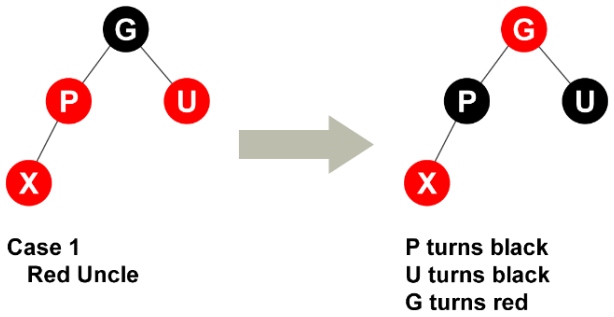
If the root property is violated, simply change the root's color to black (we can consider this to be "case 0"). However, if the red property is violated, there are 5 cases we can be in. In each of these cases, we need to look at the uncle's color. Recall that, as in real life, an uncle is the sibling of your parent. For example:



Depending on the uncle's color and the relationship between the newly created node, it's parent and it's grandparent, we can be in one of 5 cases. If we consider the newly created node to be node X, it's parent to be node P, it's grandparent to be node G and it's uncle to be node U, then we have the following five cases:



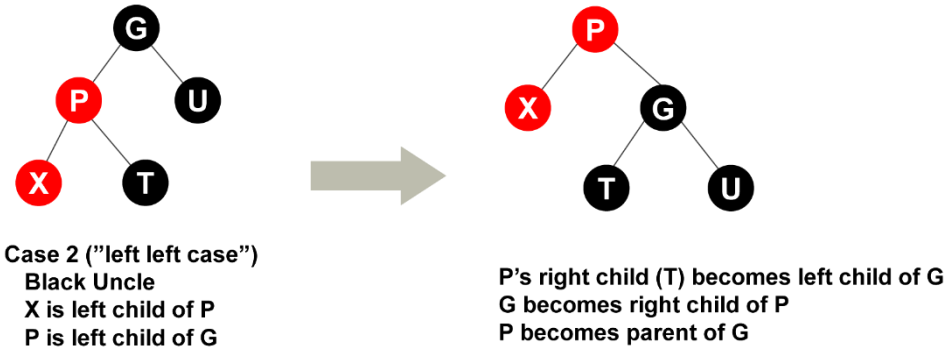
Case 1. For this case it doesn't matter if we are the left or right child, the solution is to recolor. The tree isn't unbalanced enough if the uncle is red, so a simple recoloring will suffice. If this situation happens again, the uncle will be black and re-balancing will have to occur. To recolor, do the following:



Simply change the parent and uncle to black and the grandparent to red. Note that this won't cause issues with the depth property, as before the recoloring, any path down this part of the tree would have touched G which is a black node, adding 1 to the black count. After recoloring, any path must now touch either P or U, also adding 1 to the black count. So the black count remains the same as before. For this reason, we don't have to explicitly check the depth property.

After recoloring, you will need to recursively check for violations with the grandparent, G. Note that if the grandparent is the root, the violation check should spot a case 0 violation (i.e. root property violation) and change the root to black. However, even if G is not the root, there may still be violations if G's parent is also red. So we must recursively check for violations as we work our way up the tree.

Case 2. This is also known as the "left left case" since P is left of G and X is left of P. To fix this we have to rotate some things:



Case 2 ("left left case")
Black Uncle
X is left child of P
P is left child of G

P's right child (T) becomes left child of G
G becomes right child of P
P becomes parent of G

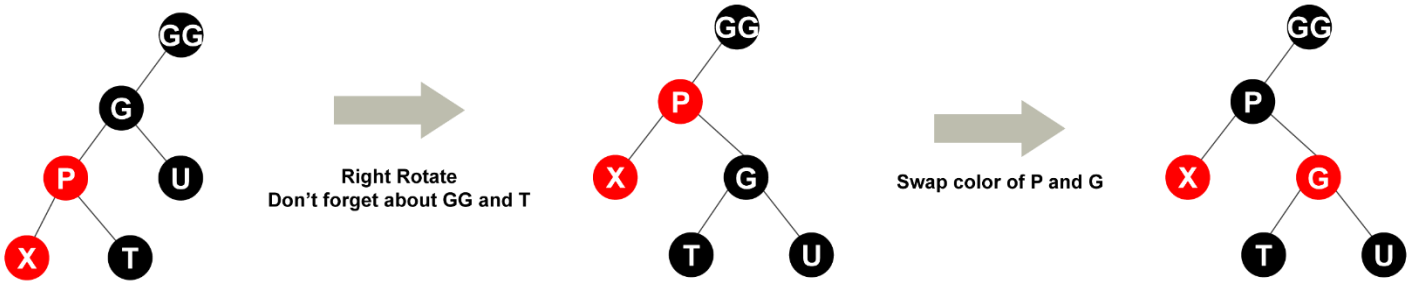
T is a potential right child that the parent P might have. If the parent does not have a right child, then this part can be ignored. As you can see, P and G move around in the tree. Their child pointers and parent pointer must change accordingly. This is known as a "right rotate" as it appears as though the tree has rotated clockwise. This seems difficult to implement at first, but really it is just about reassigning the right pointers (which is just calling setLeft, setRight and setParent on the right nodes). It is really important to remember that the parent pointers of P, G and T must change! For example, P's parent pointer is now whatever G's parent pointer used to be. Think about this for a second. Let's call this node GG (for "great grandparent" since this used to be G's parent). If G was a left child of GG, then P should now be the left child of GG. Same for right child. In other words:

Left child of GG becomes P

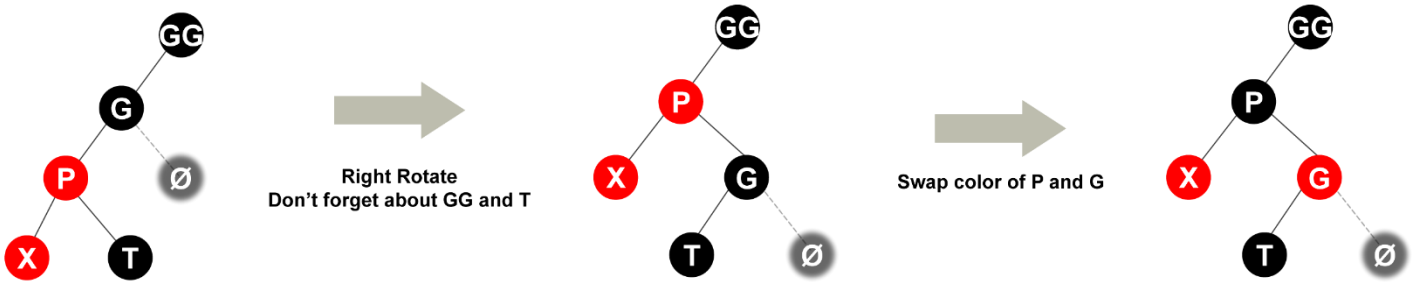
Right child of GG becomes P

So this means that pointers associated with GG, G, P, and T must change (pointers for U and X aren't affected). This includes left, right and parent pointers where applicable. Do note that some pointers will not change, such as G's right child, U. Before the rotation, G's right child is the Uncle U and after the rotation it is still U. So only change the pointers that actually change during the rotation. Also, note that GG may actually be null. In this case, do NOT forget to make P the new root! Failure to do so will corrupt your tree as the root will still be pointing to G even after the rotation.

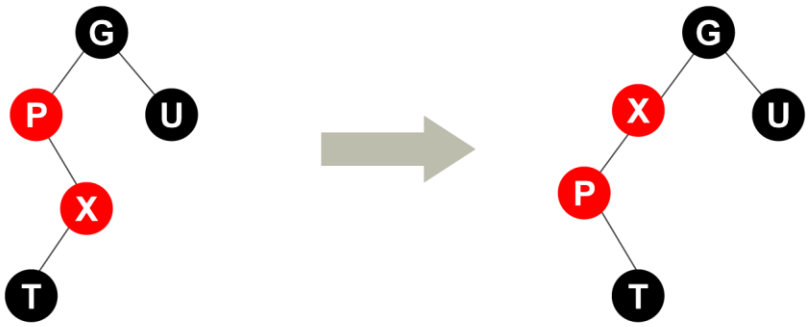
The final change, after the rotation, is to swap the colors of P and G. So, to summarize, we have the following:



Now, to be clear, think about the external property. All null nodes are to be considered black. This means that if the right child of G is null, it is still to be considered a black node. Meaning that the uncle would still be black, putting us into case 2:



Case 3. This is the "left right case". We need to do a left rotate:



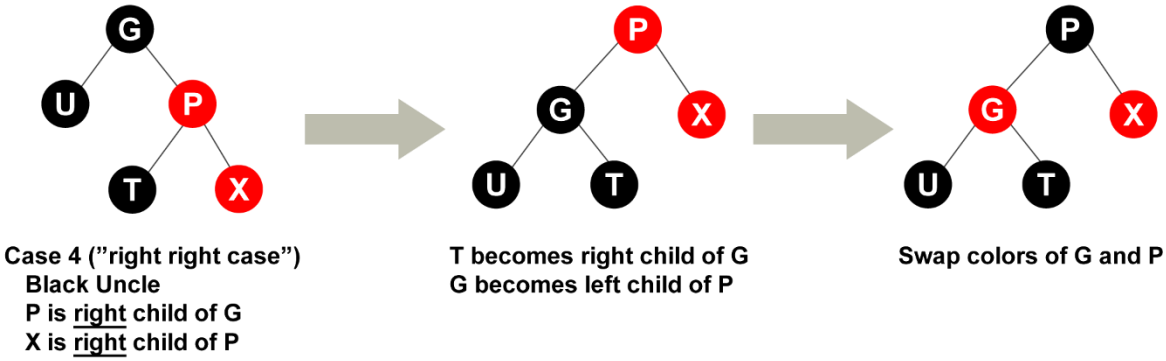
Case 3 ("left right case")
Black Uncle
P is left child of G
X is right child of P

T becomes right child of P
P becomes left child of X
X becomes left child of G

This case ends up being a tad bit easier. Since G doesn't change, we don't have to worry about GG. We also don't even have to check to see if T is null, since if it is null, then P's right child becomes null and we still have effectively transferred T. Notice though that there is still a violation. To be specific, X and P are parent and child, but both are red. This violates the red property. We could fix it directly in code, but I find it is much easier to just recursively call the function sending it P to check for violations. It will then see that P is a Case 2 (left left) violation and fix it accordingly, reusing the code we wrote for case 2. If you can't tell that this is a case 2 violation, just imagine X as the parent P and P as the newly created node X. If you swap the letters, it is exactly case 2. If you call the function with P as the X node, the code will happily treat P as X and X as P, making the necessary adjustments in the rotation. Note that the labelling of node T would change, however whether X or P has extra children doesn't change the violation type

Speaking of T, why do we have to consider a potential child node (labelled T in case 3) when X is the newly created node? Note that we signify X as the node we are ultimately examining but case 3 could have been spotted after other violations had been fixed and we could be working our way up the tree. So if this isn't the very first violation after an insert, X may have a child or two by now.

Case 4. The right right case.



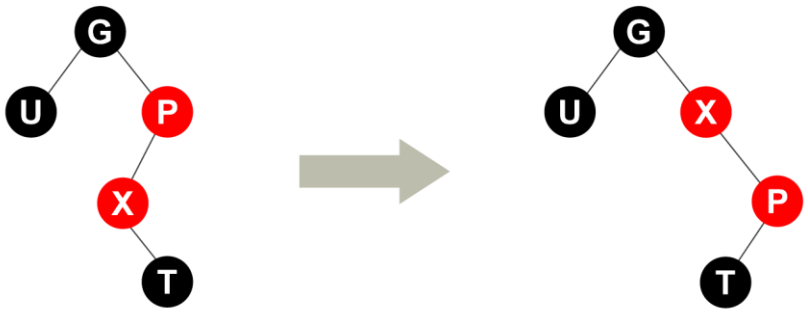
Case 4 ("right right case")
Black Uncle
P is right child of G
X is right child of P

T becomes right child of G
G becomes left child of P

Swap colors of G and P

Here since P moves into G's spot, we must consider GG like we did with the left left case.

Case 5. The right left case.



Case 5 ("right left case")
Black Uncle
P is right child of G
X is left child of P

T becomes left child of P
P becomes right child of X
X becomes right child of G

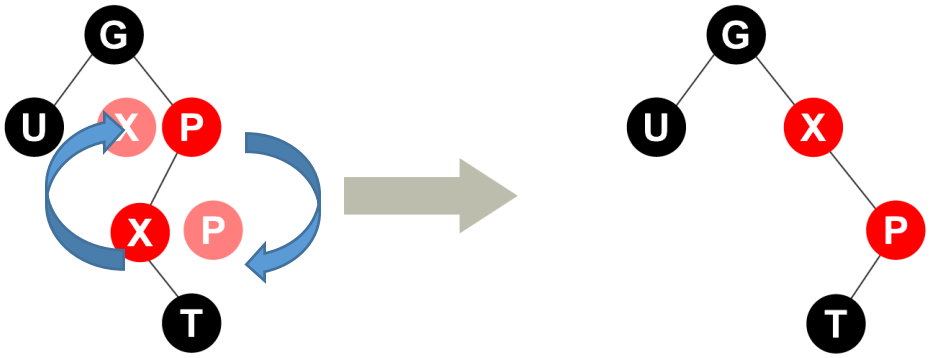
Just like with the left right case, we don't have to worry about GG, however we do need to recursively call the function with P as the node to examine. The function will then detect a right right case with P and move things around accordingly (followed by a recoloring).

Another way you can look at these cases is the following:

Case		Indication	Resolution
Case 0		Red Root	Recolor
Case 1		Red Uncle	Recolor
Case 2	Left Left	P and X form a line	Right Rotate Don't forget about GG Swap colors of P and G
Case 3	Left Right	P and X form part of a triangle	Left Rotate Recursively call with node P
Case 4	Right Right	P and X form a line	Left Rotate Don't forget about GG Swap colors of P and G
Case 5	Right Left	P and X form part of a triangle	Right Rotate Recursively call with node P

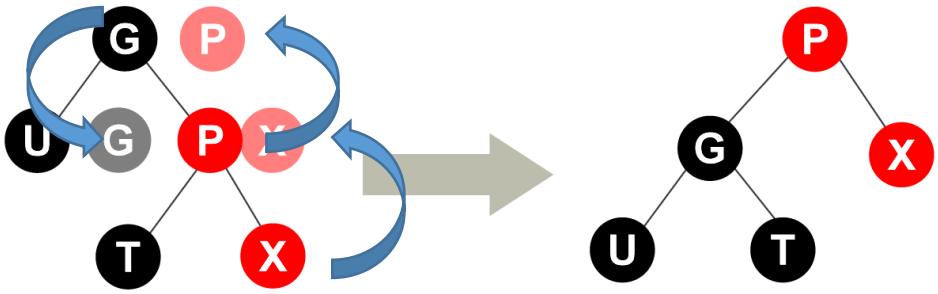
Rotations

In case you are having trouble seeing why we call it a "left rotate" or a "right rotate", consider the images below. These show an example of a right rotate and a left rotate. The direction is important here. Recall that when you touch a nozzle (such as a handle on a faucet or garden hose) and you turn it to the right, this forms a clockwise (aka CW) rotation, which is exactly what the nodes do in a right rotate:



Right Rotate - CW

When you turn it to the left, this forms a counter-clockwise (CCW) rotation, which is exactly what the nodes do in a left rotate:



Left Rotate - CCW

Now go back and look at the cases above. You should be able to pick out when we are doing a left rotate and when we are doing a right rotate. Remember, while this may look difficult to do in code, it is really just re-assigning pointers, which is simply just setting the left, right and/or parent pointers in a node to point to different nodes, thereby changing where that node is in the tree. Do take special care to get the order right though! Assignment pointers out of order can corrupt the tree. I HIGHLY recommend drawing this stuff (and playing through a few scenarios) to help you figure out the code.

References:

- https://www.youtube.com/watch?v=YCo2-H2CL6Q
- https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/
- https://www.cs.usfca.edu/~galles/visualization/RedBlack.html