

cs301 Assignment-3

Hasan Berkay Kürkçü - 27851

November 2022

Problem 1 – Augmenting Black Height

So, first of all node structure will not be that different from original RBT node where we add only one additional field called bheight (representing black height of that current instance) through which we can find black height of any node in constant time at the end. Black height means number of black nodes on the path from that node to leaf. Here is the node structure taken from lecture and modified:

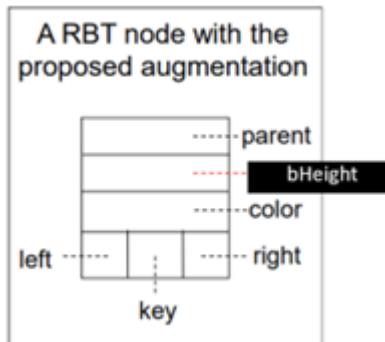


Figure 1: Fig1

Now, as we usually do with augmenting data structures we need to check if that added extra field can be maintained or not after insertion, deletion, etc. Here we will compare insertion case with previous complexity which is $O(\log n)$. Then, when it comes to insertion in RBTs we will first insert it as BST (because RBTs are just augmented version of BSTs at the end), then to keep it balanced we'll do required rotations or shifts or fix it in a sense.

Inserting it as BST is $O(h)$ which is $O(\log n)$ for RBTs because what we do is based on the key value go left or right and go 1 level deeper at each step. Here I took example of insertion from the slides and wrote down the bh values for

that tree to observe.

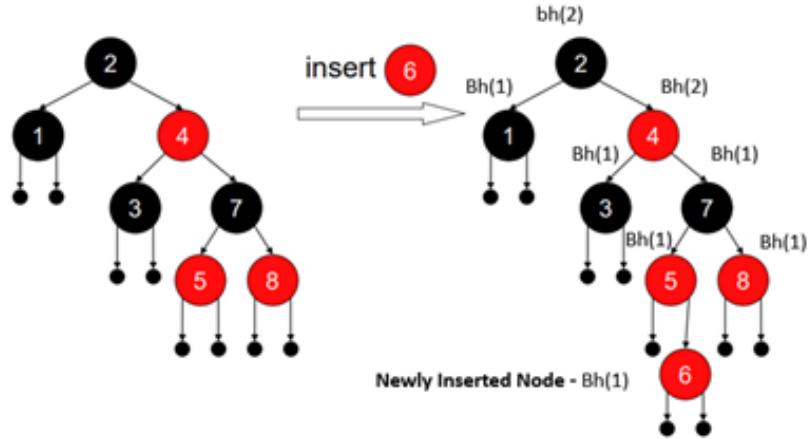


Figure 2: Fig2

Now, for fix part we'll have 3 different cases like non-augmented RBT version. I'll follow lecture slide order (Case 1 Case 3 and Case 2 which is more reasonable)

Case 1) This case happens when parent of inserted and node and sibling of the parent is red (red-red pair) and inserted node is the right child of the parent node. To solve that problem (because we have red child of red parent which violates RBT) we will move red color of parents to black grandparent and vice versa (like switching). Illustration below:

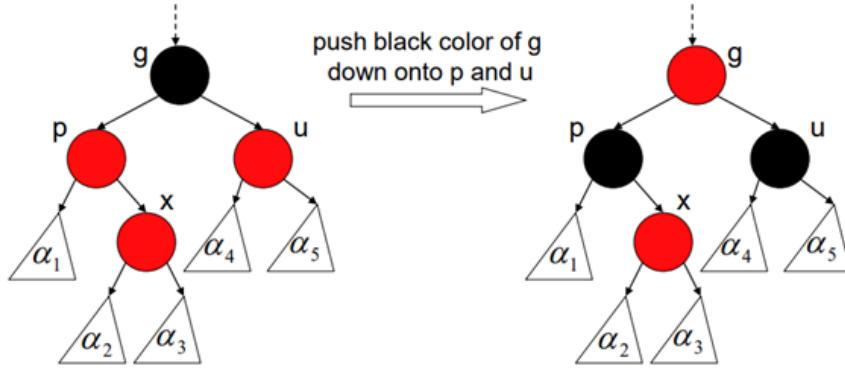


Figure 3: Fig3

So, reviewing this illustration we see that inserting node with key x does not change bh of $\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, p$ or u because for bh to change there should be an increment on the number of black nodes from that node to leaf and for those this is not the case (note that inserted node is red). However, for g , there is a change after swapping colors such as $bh(g) = bh(g) + 1$ because now its children are black nodes. Then, we can say that this operation's complexity is constant because all we do is taking blackHeight of g and increment it by 1 and swapping colors. Now, it is still not done because note that after that fix still there might be red-red pair on the upper layer (now we treat grandparent as inserted, parent of it as new parent... etc.) and fix that layer, that is why in the worst case we can go up until root (in that case root should be left as black as a rule) and complexity becomes $O(h) = O(\log n)$. Note that at each level it again takes constant time complexity. In other case while going above layers at some instant we can encounter case 2 or case 3 which are constant time complexity operations as explained below, so again it is not a problem.

Then, here for Case 1 we see that augmentation does not increase the previous time complexity which is $O(\log n)$ because still we have $O(\log n)$ from insertion + $O(\log n)$ from fixing part = $O(\log n)$. Note that if x were to be left child of the parent (symmetric case) everything would be the same and complexity would not increase.

Case 3) This case happens when we have inserted node x as left child of its parent (for symmetric It is right) and its parent is red. However, this time sibling of parent or uncle lets say is black. This time, to fix the problem (because red node cannot have red child which is case at start) we will do rotation to right hand side, make grandparent red and parent black after rotation. Illustration taken from lecture slides:

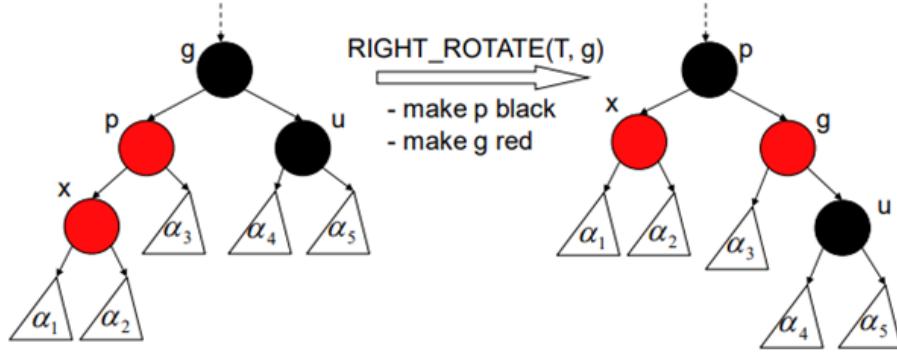


Figure 4: Fig4

So, again thinking if augmentation increase the time complexity, we will review those nodes' cases. For parent bh increase by 1. For x, g or u we do not see any bh change which means they still have same number of black nodes on the path through leaves (only heights of nodes change due to the rotation). Then we just do constant operation here by doing rotation steps and updating one bh value. Here we don't have to think upper layers because as we said in the lecture only red-red problem is fixed at that layer and it cannot repeat above. Then we have constant complexity for rotation and as stated above $O(\log n)$ for the insertion of the node which gives us $O(\log n)$ at the end. We do not see any increase because of the augmentation in case 3 neither. As in the case 1 considering symmetric does not change anything other than rotation side.

Case 2) In that case, though it seems same, there is a subtle difference with case 3 which is now inserted node is right child of the parent (for symmetric it is left), or in other words child is inner side now which is more problematic. To solve it, we will convert scenario to case 3 and then solve it as case 3. Illustrated picture below is converting step:

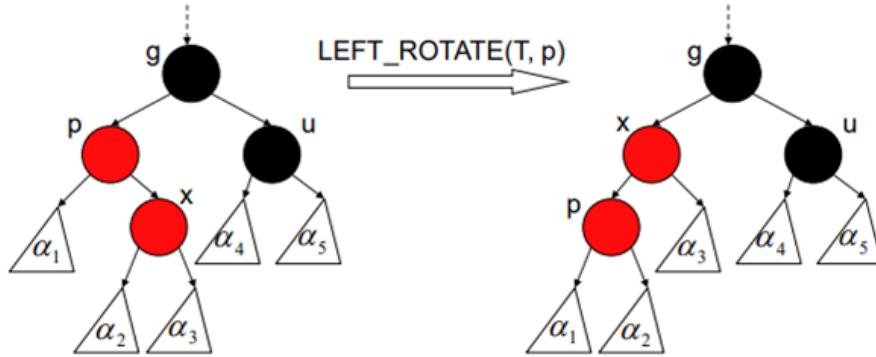


Figure 5: Fig5

Reviewing it, we see that all we do is moving red nodes (x and p and their subtrees) which means none of the bh values will change there, all same and for the complexity, it is just constant since all we do is rotation. Then we said that rest is the same as case 3 and we showed above that case 3 does not add anything other than constant complexity. Hence, for case 2 we have dominating step as insertion of new node which gives us $O(\log n)$ at the end.

As a result, we took into consideration all three cases and for case 2 and case 3 we see that upon insertion of new node which is $O(\log n)$ we just add constant times and augmentation does not make difference. For the case 1 we said that it may apply for all levels above up until the root but again it just adds another $O(\log n)$ to $O(\log n)$. Hence, we can say that this augmentation does not increase time complexity of inserting new node in the worst case.

Problem 2 – Augmenting Depth

Now, we wont do very different things from problem 1 and again review if insertion of this augmentation can be maintained without increasing time complexity. As in the problem 1 finding first values for augmentation is easy because here while inserting as we go below, we increase the depth by 1 and this is depth for that node. Here is the node structure:

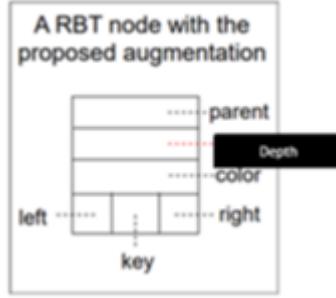


Figure 6: Fig6

When it comes to insertion, for case 1 we know that only color changes and structure or placement of nodes does not change and because of that none of the depth values change. Again, it can go up until to the root; however, nothing changes because at every level we just change colors. Below, we will be considering cases when case 1 stops at some level and transforms into case 2 or case 3 and we review those cases separately but for the case 1 insertion complexity is equals to inserting the node (also we can find while inserting in constant time) $O(\log n)$ + considering worst case where it goes to root $O(\log n) = O(\log n)$

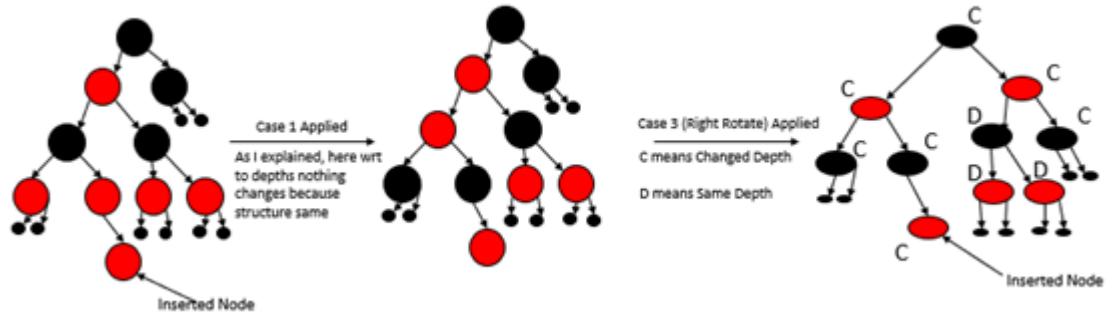


Figure 7: Fig7

In the first step, case 1 did not change anything on depth wise. However, when it comes to rotation (here I tried to illustrate case where we need to rotate root), I put C for nodes when rotation changed depth and D when it did not change. As we see here more than $7n/10$ of the nodes' depth need to be updated which is more than $O(\log n)$. Because of that, we have $O(\log n)$ from inserting the new node and $O(7n/10)$ for that case for rotation and $O(n)$ at the end. Hence, this augmentation increase time complexity and it can not be maintained in

$O(\log n)$ time. When we focus on nodes that have same depth, they are right child of rotated node and they are put at same level, that is why they have the same depth. Then we can ask if that subtree become much bigger and again if we get $O(\log n)$ in some cases; however, this is impossible because RBTs are self-balanced and that subtree cannot have that many number of nodes, so whenever root is rotated we can not maintain that complexity with depth augmentation. In the symmetric case it will not be different we will just consider the subtree of rotated node's left child as having same depth. As I explained, even in the worst case size of that tree is smaller than half of the total number of nodes because it is self balancing, that is why we can not maintain same complexity.