CS301-HW3

Ahmet Furkan Ün

November 19, 2023

Problem 1

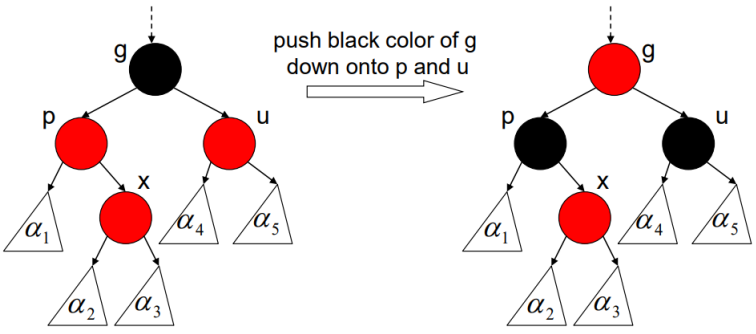
Initially, the node structure isn't significantly different from the original Red-Black Tree (RBT) node. We introduce a single additional field called "bheight," representing the black height of the current instance. This allows us to determine the black height of any node promptly—at the end of the process. Black height refers to the count of black nodes on the path from that node to a leaf.

In the customary fashion of augmenting data structures, we assess whether this extra field can be maintained post-insertion, deletion, and other operations. In this context, we'll compare the insertion case with the previous complexity, which was O(logn). Concerning insertion in RBTs, our approach involves initially inserting it as a Binary Search Tree (BST). RBTs are essentially augmented versions of BSTs. To maintain balance, we perform necessary rotations, shifts, or fixes afterward. The time complexity of inserting it as a BST is O(h), effectively O(logn) for RBTs. This is because the process, involves going left or right and descending one level deeper at each step.

Now, for the fix part, we encounter three different cases, mirroring the structure of the non-augmented RBT version.

(a) Case 1

In this scenario, Case 1 unfolds when the parent of the inserted node and the sibling of the parent are both red (forming a red-red pair), and the inserted node is the right child of the parent. To address this issue, as having a red child of a red parent violates the Red-Black Tree (RBT) property, we implement a solution. The remedy involves a color exchange, specifically, transferring the red color of the parents to the black grandparent and vice versa—essentially performing a switch. The illustration below demonstrates this process:



Upon reviewing the illustration, it becomes apparent that inserting a node with key x doesn't alter the black height (bh) of a1, a2, a3, a4, a5, p, or u. Since the inserted node is red, there is no increment in the number of black nodes from these nodes to the leaf. However, for g, a change occurs after the color swap, resulting in bh(g) = bh(g) + 1 because its children are now black nodes.

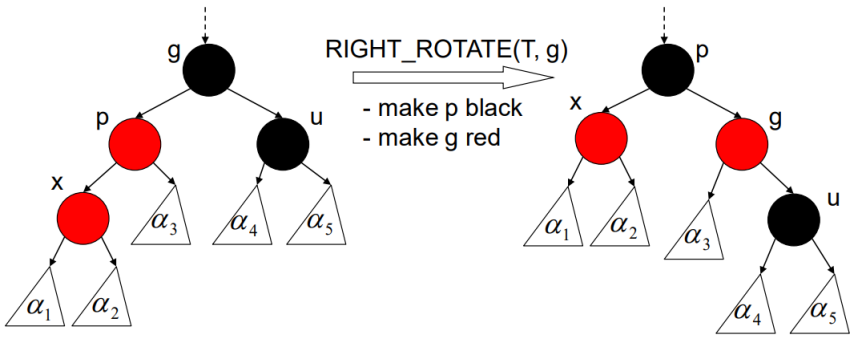
The complexity of this operation is deemed constant because it involves taking the black height of g, incrementing it by 1, and performing color swaps. Yet, the process isn't complete, as even after this fix, there may still be a red-red pair on the upper layer. At this point, we treat the grandparent as the newly inserted node, and the parent of it becomes the new parent, and so on. This process continues until we potentially reach the root (which should be left as black, following the rule). Consequently, the overall complexity becomes O(h) = O(logn) in the worst case, considering that at each level, it again takes constant time complexity.

It's worth noting that while ascending through the layers, we may encounter case 2 or case 3 at some point. However, these operations are constant time complexity, as explained below, so they don't pose a problem in the overall complexity analysis.

In the context of Case 1, it's evident that the augmentation doesn't increase the time complexity, which remains O(logn). This is because we still have O(logn) from the insertion phase plus O(logn) from the fixing part, resulting in an overall complexity of O(logn). It's essential to note that if x were the left child of the parent (symmetric case), the situation would be same, and the complexity wouldn't experience an increase.

(b) Case 3

This situation arises when the inserted node x becomes the left child of its parent (for the symmetric case, it would be the right child), and the parent is red. However, in this instance, the sibling of the parent or the uncle is black. To rectify this issue, as a red node cannot have a red child initially, a right-hand side rotation is performed. This results in making the grandparent red and the parent black after the rotation. The illustration is extracted from the lecture slides:



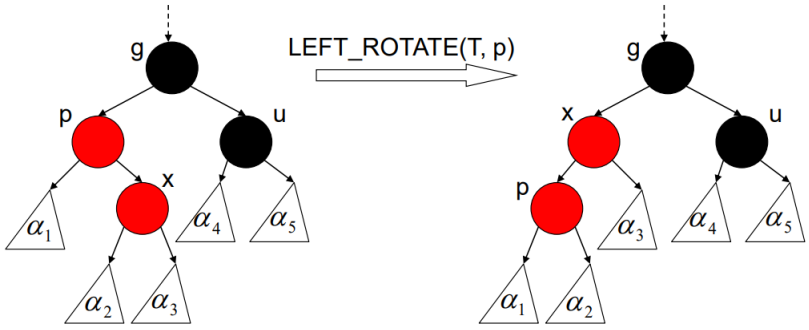
Considering whether augmentation impacts the time complexity, we scrutinize the cases of the affected nodes. For the parent, the black height (bh) increases by 1. As for x, g, or u, there is no observed change in bh, indicating that they still have the same number of black nodes on the path through leaves (though the heights of nodes alter due to the rotation). The operation here involves constant steps through the rotation and updating one bh value.

No consideration needs to be given to upper layers since, as discussed in the lecture, the red-red problem is resolved at that layer and does not repeat above. Consequently, the rotation incurs constant complexity, and when combined with the O(logn) complexity for the insertion of the node,

the overall complexity remains O(logn). Similar to Case 1, the augmentation does not introduce an increase in complexity in Case 3. Symmetry, in this context, only affects the rotation side without altering the overall analysis.

(c) Case 2

Now, let's delve into Case 2. Although it may appear similar to Case 3, a subtle difference exists: the inserted node is now the right child of the parent (for the symmetric case, it would be the left child), or, in other words, the child is on the inner side, which introduces a more problematic scenario. To resolve this, we transform the situation into Case 3 and subsequently address it as such. The illustrated picture below demonstrates the conversion step:



Upon reviewing the process, it becomes apparent that all we are doing is relocating the red nodes (x and p and their subtrees). This means none of the black height (bh) values will change; they all remain the same. In terms of complexity, this operation is constant, as it involves only a rotation. As discussed earlier, the remainder of the process aligns with Case 3, and as demonstrated, Case 3 introduces only constant complexity. Consequently, for Case 2, the dominant step is the insertion of the new node, which incurs a time complexity of O(logn).

In summary, considering all three cases, for Case 2 and Case 3, the insertion of the new node, with a time complexity of O(logn), is augmented by constant steps. As for Case 1, it may apply to all levels above up until the root, but again, it merely adds another O(logn) to the existing complexity. Therefore, we can confidently assert that this augmentation does not elevate the time complexity of inserting a new node in the worst case.

Problem 2

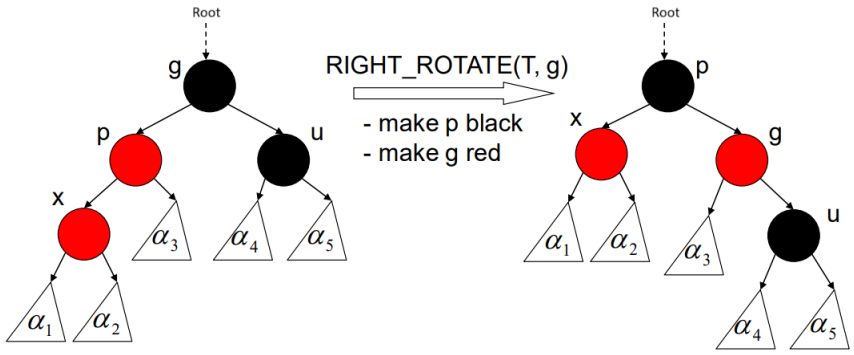
In the context of finding initial depth values for augmentation, it is relatively straightforward here. While inserting, as we descend, the depth increases by 1, and this depth corresponds to that node.

Regarding insertion, for Case 1, as explained in Problem 1, we recognize that only color changes occur, and the structure or placement of nodes remains unchanged. Consequently, none of the depth values change during this process. Although Case 1 can extend up to the root, no alterations occur because, at every level, only color changes are implemented. Below, we will explore cases where Case 1 stops at a certain level and transforms into Case 2 or Case 3. However, for Case 1 insertion, the complexity is equivalent to inserting the node (which can be determined in constant time) O(logn) + accounting for the worst-case scenario where it goes up to the root O(logn) = O(logn).

However we need to be aware of the situations where the case 1 problem turns into a case 2 or case 3 problem. We know that case 1 problem may go up to the root of the tree. This means

that we can also carry the problem to the upper nodes of the tree and we may encounter case 2 or case 3 cases near the root. As we explained in the Problem 1, case 2 and case 3 problems need rotations to be resolved. However, rotations means change in the depth values in the context of this depth augmented RBT.

In the worst case, we may solve the case 1 problems one after each other and very close to the root we may encounter case 2 or case 3. Let's consider an example where the case 1 problem carried near to the root and converted into a Case 3 problem.

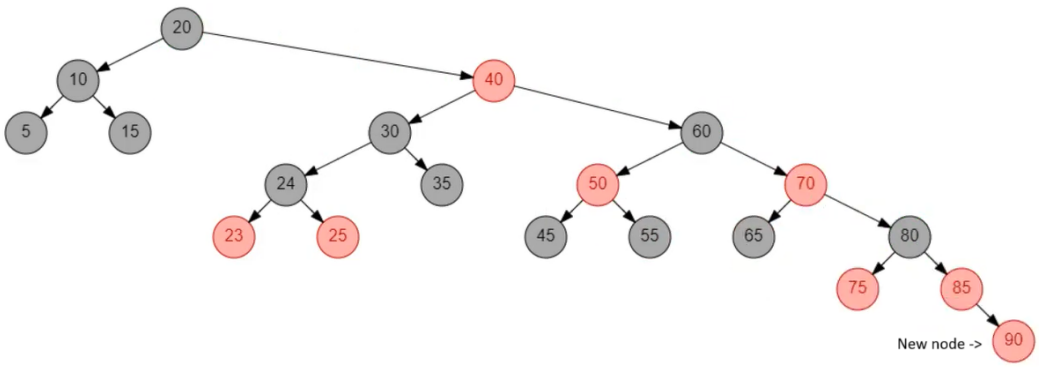


If you look at the figure, you can see that only the depths of subtree $\alpha 3$ remains same, and the depth of g, u and elements of subtrees $\alpha 1$, $\alpha 2$ should be increased by 1. Similarly the depth of p, x and elements of subtrees $\alpha 4$, $\alpha 5$ should decreased by 1.

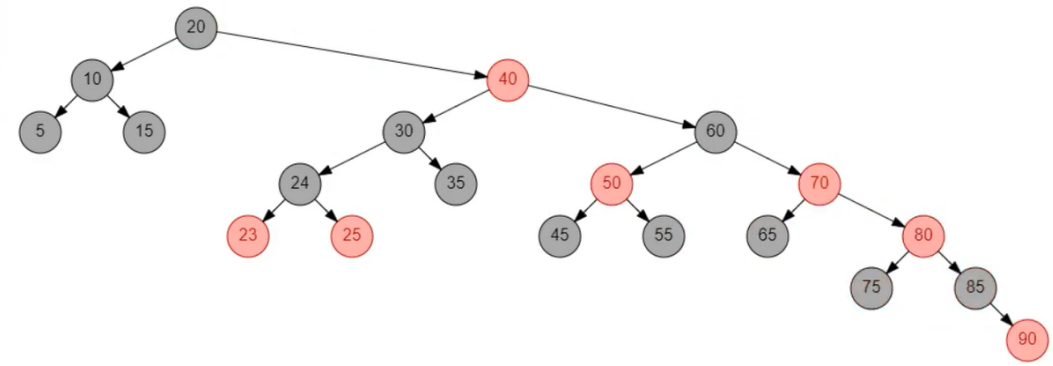
This means that we changed all the depth values except the subtree $\alpha 3$. We know that the height of the tree is at most 2log(n+1), then the maximum number of elements can be contained in subtree $\alpha 3$ is 2log(n+1)-2. Therefore if there are n nodes in the tree, we changed the depth of $n - (2log(n+1) - 2) = n + 2 - 2log(n+1)$ nodes.

Changing the depth is a constant time operation but doing this many times cost $O(n + 2 - 2log(n+1)) = O(n)$ which dominates the O(logn).

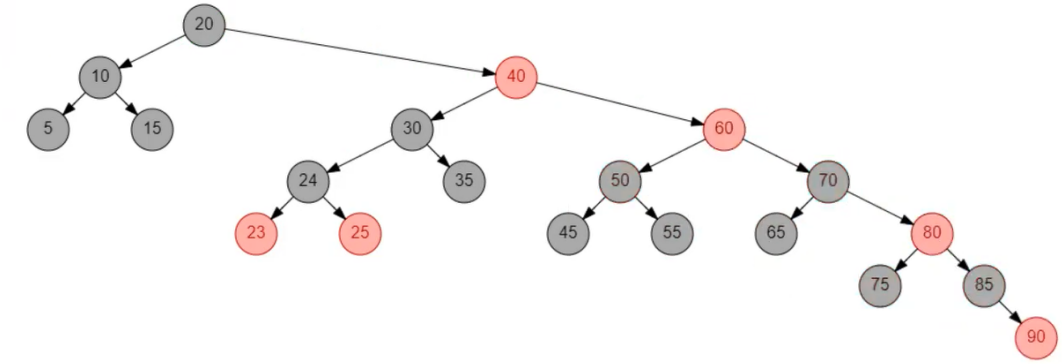
Let s consider a concrete example:



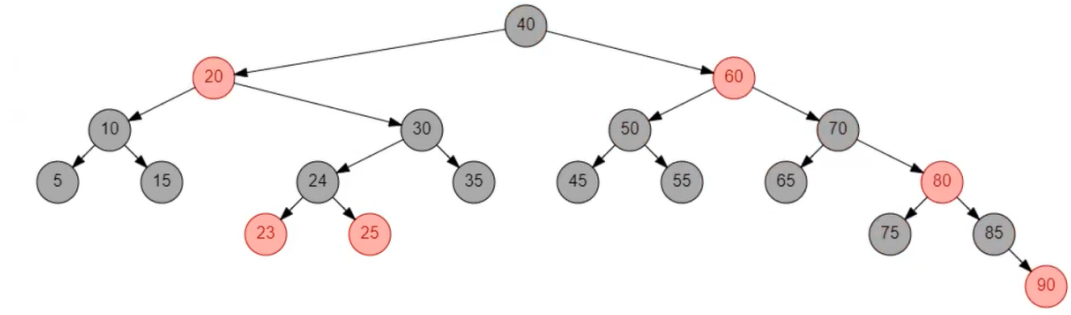
Then we need to apply Case 1:



Then we need to apply Case 1 again:



Then we need to apply Case 3:



From this example, we can see the carriage of the problem by the case1 to the root. The first 2 application of the Case 1 did not changed any depth values at all. However after the application of Case 3, we can see that only the depth of the elements of subtree whose root is 30 did not changed. This means that in a tree with 20 elements, only the depth of 5 elements remained the same and the depth of 15 elements changed which is 3n/4.

As a conclusion we can say that augmenting the depth value in the RBT increases the asymptotic time complexity of inserting a node into an RBT in the worst case because of the rotations performed in the Cases 2 and 3.