Taner Giray Sönmez
2023, Fall

CS 301: Algorithms
Assignment 3

Problem 1

First of all, we need to modify the normal RBT node in order to get augmented version of RBT node. For this purpose, we just need to add additional integer field to the regular RBT node and get RBT node with the proposed augmentation.Therefore, new instance of RBT node includes parent,left, right pointers, color field and newly proposed integer black-height field which we can simply call it bh.

Since we augment new field to the RBT, we want to ensure the followings:

- The additional fields help us to compute more efficiently
- The additional fields can be maintained within a reasonable time

In other words, "since insert and delete operations modify the tree, we need to guarantee correct assignment to the new fields after these operations are executed on the augmented RBT". Moreover, "we need to be able to perform the assignments in a time that would not increase the worst case running time of the original insert and delete operations". Question wants us to explain insertion operation, hence we do not consider the deletion operation here. Regular RBT has an insertion complexity of O(h) which is O(logn). In order to maintain O(logn) worst complexity in augmented version, we need the consider three different cases for insertion(regular insertion of RBT) which are given in lecture slides:

First case:

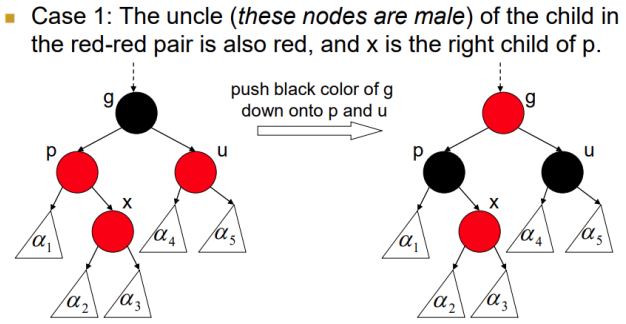

Figure 1: First Case

This happens when newly inserted node x's parent is red also x's uncle is red. Besides, x is right child of the parent. In order to maintain RBT's properties(a red node always has two black children) we need color tree again. For this purpose, we give grandfather's color to parent and uncle, and change grandfather's color to opposite. After coloring, the black height of a_1 to a_5 nodes, as well as the black heights of the parent and uncle, does not change. Because the number of black nodes from those nodes (excluding the starting nodes) to leaves does not change. On the other hand, grandfather's black height should be incremented as 1 since p and u nodes are black now. The complexity of this operation is constant because essentially we just increment the black height field(bh) of node g and change the color of it as opposite. After that, there can be still red-red pair in upper nodes of g.Hence we need to fix coloring of upper nodes as well. In the worst case scnerio, this fixing process may extend up to the root, ensuring that the root remains black. In this case, complexity is O(height of the tree) which is O(logn).Since adjustment of black height(b)

field of the nodes takes constant time, when we go up until root, it still remain as constant. Still we can encounter second and third case when proceeding to upper nodes, but these cases also take constant time(i wanted to say that overall complexity of them stays as O(logn)) which will be explained in followings parts.By considering all of these, we can simply say that in the situation of first case, augmented RBT's insertion operation cost does not increase, it stays as O(logn). Since insertion is O(logn) and let's say increment take constant time c. Then we will have (height of the tree * c) as cost of increment in the worst case. This is nothing but O(logn), insertion is also 0(logn) hence combining these two again will be O(logn). Lastly, the symmetry of the first case (where x is the left child) does not alter anything; hence, the complexity remains unchanged.

Third case:

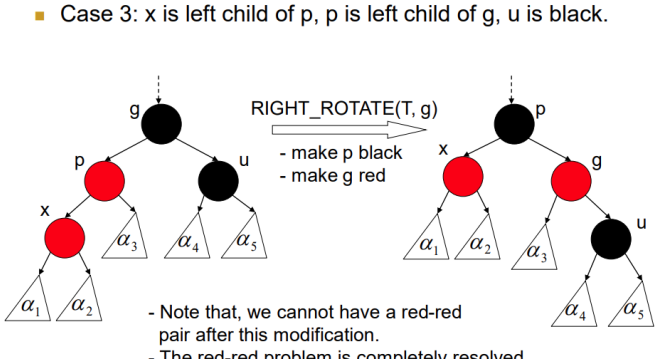

Figure 2: Third Case

This case happens when newly inserted node x is left child of parent, parent(red) is also left child of grandfather and uncle is black. Parent has red child which cannot be happened hence we need the solve. In order to solve this we will do right rotation. We will rotate around g(pivot). After that we will make parent black and grandfather(grandparent) red. If we carefully inspects the nodes after rotation and recoloring, the black height of the parent is increased by 1 since black uncle becomes the child of the parent. Simultaneously, the uncle, x, and grandfather do not experience any change in black height (bh) because the trees rooted at these nodes do not possess any additional black nodes. Again the operations we did here are constant as well encompassing rotations and the updating of a single bh value. Moreover in this case we do not need to bother ourselves with upper nodes because "If case 3 applies, then a single rotation (constant time operation) will terminate the algorithm". Hence again, insertion of a node O(logn) with constant time rotation and constant time incrementation of bh value. Eventually, we got O(logn) complexity again. Lastly, the symmetry of the third case does not alter anything; hence, the complexity remains unchanged.

Second case:

This case happens when newly inserted node x is right child of parent, parent(red) is left child of grandfather and uncle is black. Again, parent has red child which cannot be happened hence we need the solve. The approach for this case, as explained in the lecture, involves attempting to transform it into the third case. Once converted, we can apply the solution for the third case, which we already know. We need to left rotate around parent(pivot) and then it becomes case three which we know how to solve. When we left rotate around p, the black heights of the nodes do not change(no new black nodes). Moreover, we know that this left rotation takes constant time. After that this will be case three which has the complexity of O(logn). Overall, the complexity of third case will surpass the constant time left rotation and final complexity becomes again O(logn).Lastly, the symmetry of the second case does not alter anything; hence, the complexity remains unchanged.

Considering all these steps, we showed that complexity remains O(logn) in augmented RBT in all cases.Therefore, we can conclude that this augmentation does not increase the time complexity when inserting a new node in the worst-case scenario.

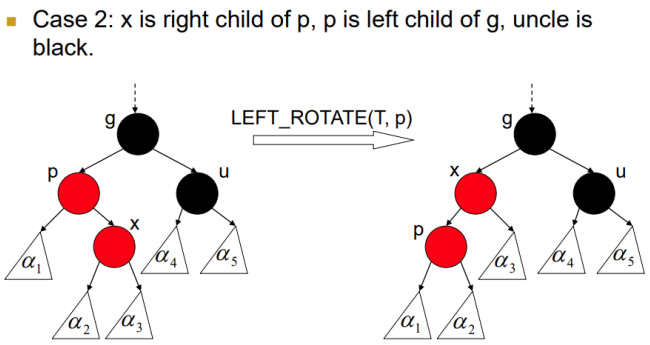

Figure 3: Second Case

Problem 2

Now let's considering RBT with augmented integer depth field.First of all, we need to modify the normal RBT node in order to get augmented version of RBT node. For this purpose, we just need to add new integer field to the regular RBT node and get RBT node with the proposed augmentation.Therefore, new instance of RBT node includes parent,left, right pointers, color field and newly proposed integer depth field which we can simply call it dp. Again we need to consider three cases of insertion.

First case:

If we consider to image above for the first case, we can see that only colors of nodes are changed. In other words, position of nodes do not change. This implies that there is no alteration in the depth of the nodes. Even if we need to propagate until the root in the worst case, it does not change the depth of any nodes. This is because we are only changing colors, and nothing will alter the positions of the nodes. However, if we encounter case two or case three(while going upper nodes), the complexity will increase. We will inspect those cases in the following parts. In any case, in terms of the first case, complexity remains the same(if we do not encounter any case2 or case3), which is O(logn) for insertion. Put simply, it is the same as regular insertion.Lastly, the symmetry of the first case does not alter anything; hence, the complexity remains unchanged.

Second and Third Cases:

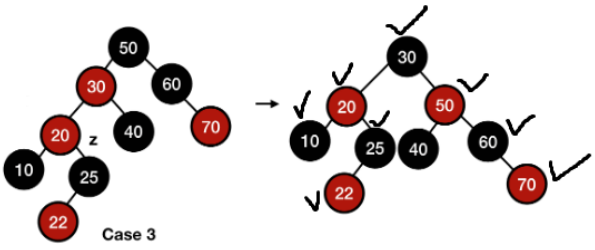

Figure 4: Case 3

For the second and third cases, I will illustrate how rotation increases the complexity. Suppose we have a situation (case 3) as shown in Figure 4 where we need to perform a right rotation/recoloring to satisfy the Red-Black Tree properties. I have marked the nodes for which we need to change the depth when we rotate the tree. As seen in the figure, 8 out of 9 nodes require a change in depth. This implies that we need to change the depth of 8n/9 nodes, which is greater than O(logn). Updating a node takes constant time c then total time O(c*8n/9) which is O(8n/9).Since O(8n/9) represents linear complexity, it is larger than O(logn), which signifies logarithmic complexity.

As a result, the overall complexity becomes linear O(n) since linear complexity dominates logarithmic complexity. Even though we may not encounter a complexity increase in case 1, Cases 2 and 3 would elevate the complexity, making it linear. In the worst case, rotations will evidently cause a significant change in depth, resulting in linear insertion. Moreover, the subtree's size cannot become significantly larger (after insertion) to achieve O(logn) complexity. Because RBTs balance themselves after insertion/deletion, preventing such occurrences. In other words,even in the worst-case scenario, the size of that tree is smaller than half of the total number of nodes(0(c*n/2)=O(n)) because RBT balances itself. This is the reason why we cannot maintain the complexity of O(logn). Hence, when we rotate the root, getting O(logn) complexity with this new augmentation becomes unachievable. To conclude, in the worst-case scenario, our system's complexity is O(n) for Case 1, which encounters either Case 2 or Case 3 in upper nodes. Both Case 2 and Case 3, impacted by rotations, also maintain a complexity of O(n). In summary, our system consistently operates with linear time complexity (O(n)).Lastly, for the symmetric cases, nothing will change; hence the complexity will increase as well.