Homework 3

Ege Aktemur

November 2022

Question 1

To show this augmentation does not increase the asymptotic time complexity one must observe and analyze the changes in the original Red Black Tree insertion. As a remainder this procedure has two parts first one is insertion and the second one is fixing the colors.

Question 1.a Insertion

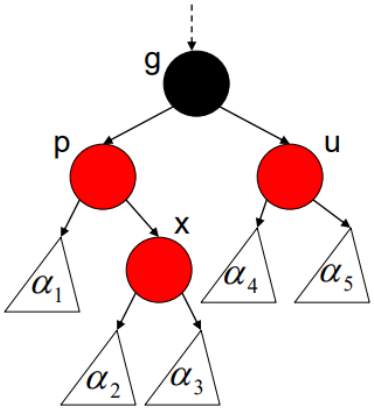
We will insert the node the same way as we do in the normal Red Black Tree. Which is done by finding the appropriate place (worst case O(logn) since it is limited with height) and inserting there (O(1)). We will insert the node as red node and fix the coloring in the next step. To sum up, this step costs O(logn).

Question 1.b Fix Coloring

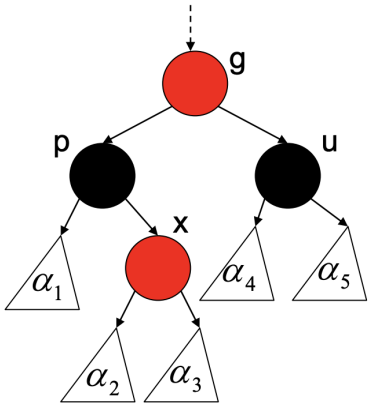
If the insertion step creates red parent with red child problem we should fix it. As we seen in the lecture there can be 3 cases that can occur. Now, I will observe them. To visualize my analyses I will uses Red Black Tree examples from lecture slides and create my own Red Black Tree using
https://www.cs.usfca.edu/ galles/visualization/RedBlack.html.

Question 1.b.1 Case 1

This problem occurs when the uncle of the child in the red-red pair is also red, and x is the right child of p or x is the left child of p and this problem can be solved by recoloring. However there is a catch, if case 2 or case 3 applies, the problem will get fixed, however instead of that, case 1 may keep applying, and we may reach to the root of the tree.



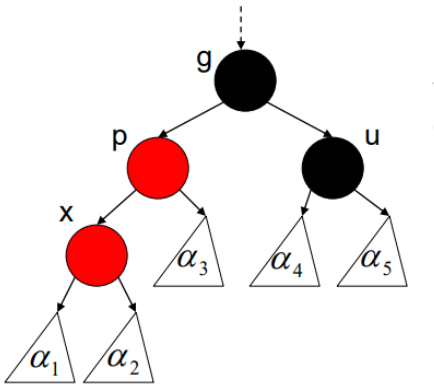
To calculate algorithmic complexity of this case we should observe the changes in black heights of each node. Since both cases of case 1 gives the same results I will observe only one.



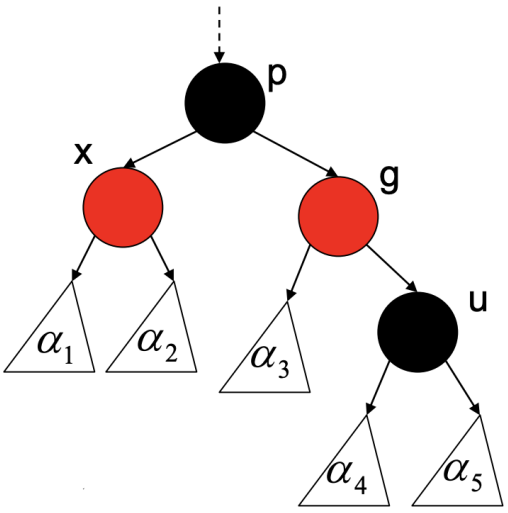
As we seen in the lecture we will fix this problem by recoloring. Only black height change in this case is in the node g (increases by 1). So in each step we only do a O(1) more operations. However since this problem may continue to the root and since we will repeat this operation at most O(logn) times. To sum up, we can say that adding black height property does not increase the complexity in case 1 since complexity is already O(logn).

Question 1.b.2 Case 3

This problem happens when x is left child of p, p is left child of g, u is black or x is right child of p, p is right child of g, u is black.



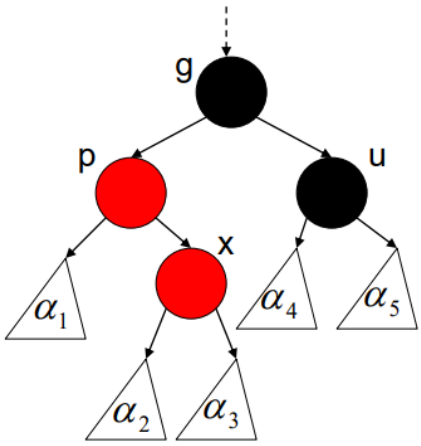
To calculate algorithmic complexity of this case we should again observe the changes in black heights of each node. Since both cases of case 3 gives the same results I will observe only one.



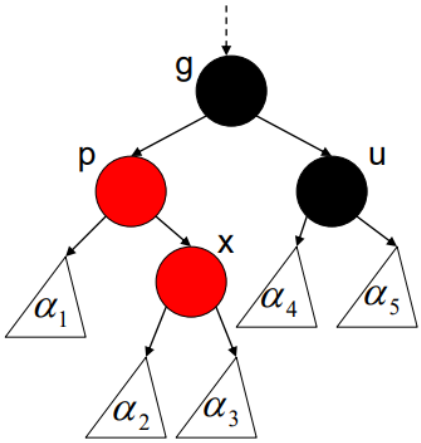
As we seen in the lecture we will fix this problem by rotating. When we observe the changes we made to fix coloring in the figure above, will see that black height of x, p, g and u will not change. Therefore, adding black height property will not add any complexity to the normal insertion (O(logn)) since it runs in constant time because there will be no updates in the any black height properties.

Question 1.b.3 Case 2

This problem happens when x is right child of p, p is left child of g, uncle is black or x is left child of p, p is right child of g, uncle is black.



To calculate algorithmic complexity of this case we should again observe the changes in black heights of each node. Since both cases of case 2 gives the same results I will observe only one.



As we seen in the lecture we will fix this problem by rotating once which turns it to a Case 3 problem. When we observe the changes we made to fix coloring in the figure above, will see that black height of x, p, g and u will not change in the first rotate. Then as we proved Case 3 also does not change any black height. Therefore, adding black height property will not add any complexity to the normal insertion (O(logn)) since it runs in constant time.

Question 1.b.4 Conclusion

We observed that in all possibilities worst case insertion is O(logn) which proves the statement that adding black height property to Red Black Trees does not increase the complexity because it is still O(logn).

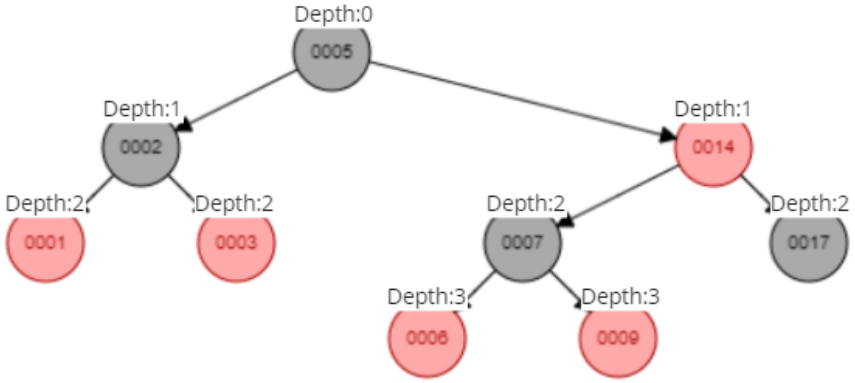
Question 2

In this section I will try to prove that adding a depth property to nodes would increase the complexity of normal Red Black Tree implementation. To explain the morality of the proof when we added black height property we were not expected to update much nodes compared to depth property. Reason behind this is we were only updating the depth field of affected nodes. Moreover, base of this proof will be proving that depth field calculation does updates more fields than O(logn).
The problem can be separated to 2 parts. First part is calculating the depth property of the inserted node and the second part is updating the depth fields of the affected nodes when we make the color fixing. Furthermore, we can see that the best case is no color fixing and the worst case is most color fixing.

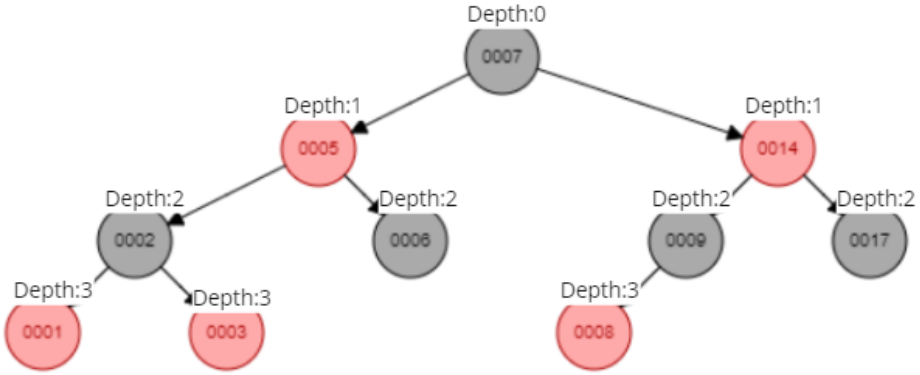
There is 2 approaches we can use:
First approach is checking and updating each node's depth field after insertion completes (O(n)).
Second approach is updating nodes during color fixing. While second approach can perform better when we don't need color fixing we will see that it also runs in O(n) in worst case.

To explain myself better I will show a visualisation I prepared over
https://www.cs.usfca.edu/ galles/visualization/RedBlack.html.

1) BEFORE INSERTION



2) AFTER INSERTION OF 8



	Start	Updated/Inserted	End
1	2	+	3
2	1	+	2
3	2	+	3
5	0	+	1
6	3	+	2
7	2	+	0
8	-	+	3
9	3	+	2
14	1	-	1
17	2	-	2

Steps:

1. Insert 8 as left child of 9. Creates Case 1.
2. Recolor
3. Rotate right
4. Rotate left
5. Recolor

If we trace the steps we see that we had to do a rotation in the root in the step 4 which results at least n/2 updates in depth properties. As I showed in the table above we had to do 8 updates in this 10 node tree. therefore we can say that running time of this method is O(n/2) = O(n).