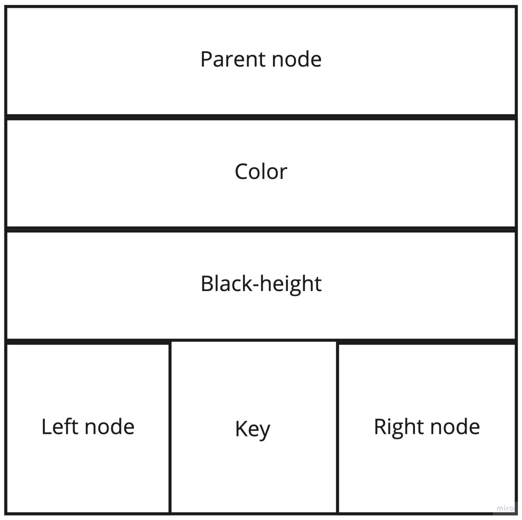
CS301-A3

Yasemin Özkut (28363)

November 2022

1 Problem 1

If we augment the black-heights of nodes as additional attributes in the nodes of the RBT, our nodes becomes as below image.



According to the property of RBT, each leaf node (null nodes) is black. To calculate every other nodes' black height, we have to look at their child's color. According to the property of RBT, each sibling bust have the same black height because each path from node x to leaf should have same number of black nodes.

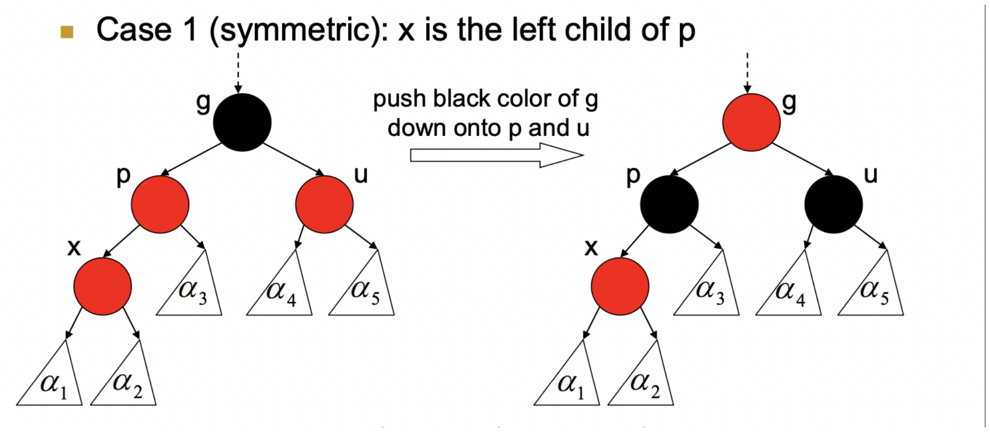
If the node x has a black colored child y, its black height will be black height of y + 1. However, if the node x has no black child and has a red colored child y, its black height will be equal to black height of y.

Examining insertion time complexity:

Since RBT has the properties of BST, all dynamic operations will take O(h) where h is the height of the tree. RBT has height of O(lgn). So, all dynamic operations take O(lgn) times. To insert, we have to find the position so we have to choose either right or left node. This provides us to eliminate half part of the nodes in each iterative step. So, the time complexity becomes O(lgn).
To protect the black red properties of the tree, we have to perform rotation and recoloring. Rotation operations will take O(c)=constant amount of time since we have to perform it on constant amount of nodes. We can also change the black height property while changing the colors of the nodes.

THERE ARE 3 CASES FOR COLORING:

Case 1:


Case 1 (symmetric): x is the left child of p
push black color of g down onto p and u
g
p
u
x
α_1
α_2
α_3
α_4
α_5

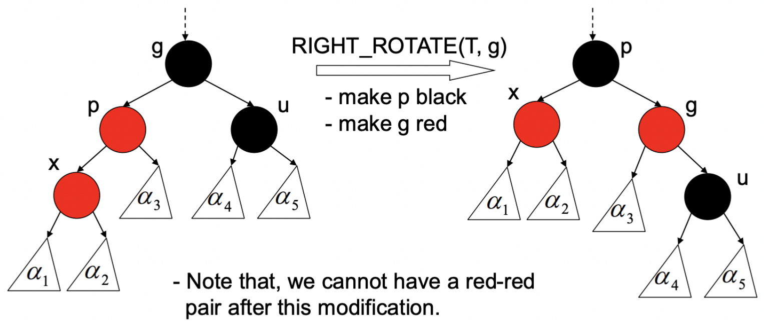
Before inserting x:
p: black height = black height of x
u: black height = black height of $\alpha4$ or black height of $\alpha5$
g: black height of p or black height of u

After inserting x:
p: black height = 1
u: black height = 1
g: black height of p + 1 or black height of u + 1

The only black height change will be on g node and only color changes will be on nodes p, u, and g. Since we are changing color and black heights at the same step, and we do not scan all tree nodes, this step will take constant amount of time. The time complexity is O(c) constant. However, since g node became red after insertion, the colors may conflict with g node's parent since red parent node cannot have red chşld node. So, in the worst case Case 1 may keep happening until the root. So, in the worst case, we may have to cascade up until the root and this causes O(h) = O(lgn) amount of time.

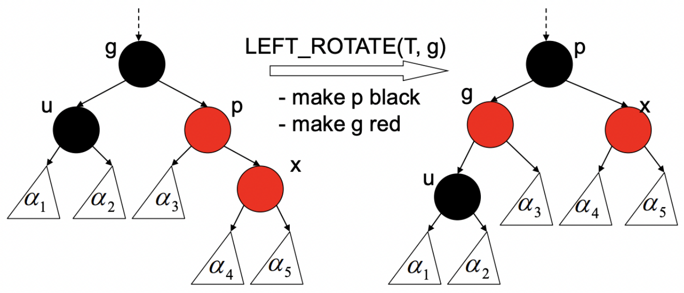
Case 3 (Symmetric):

- Case 3: x is left child of p, p is left child of g, u is black.



- Note that, we cannot have a red-red pair after this modification.
- The red-red problem is completely resolved.

- Case 3 (symmetric): x is right child of p, p is right child of g, u is black.



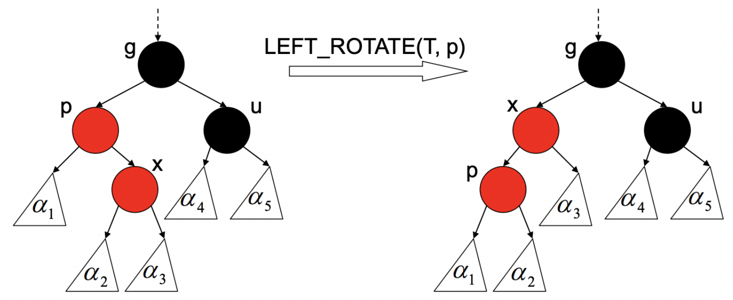
Before inserting x:
p: black height = black height of x
u: black height = black height of $\alpha 4$ or black height of $\alpha 5$ (or for the symmetric case, black height of $\alpha 1$ or black height of $\alpha 2$)
g: black height of u + 1

After inserting x:
p: black height = black height of g
u: black height = black height of $\alpha 4$ or black height of $\alpha 5$ (or for the symmetric case, black height of $\alpha 1$ or black height of $\alpha 2$)
g: black height of u + 1

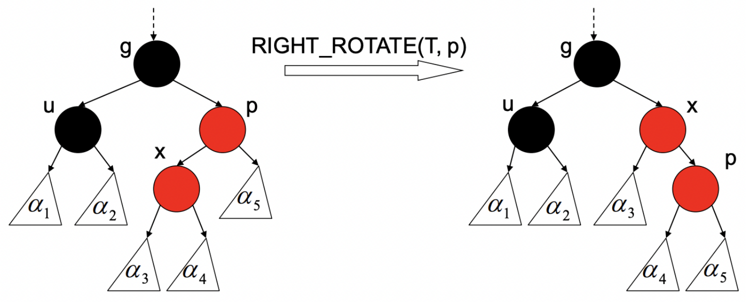
Before inserting x, g was the parent of p. After inserting x, p became the parent of g. However, the parent's black height did not change. It was black height of u+1 before insertion. It became black height of g which is equal to black height of u + 1. The only black height change will be on p node. There would not be any color change. So, this step will take constant amount of time. The time complexity is O(c) constant.

<u>**Case 2:**</u>

- Case 2: x is right child of p, p is left child of g, uncle is black.



- Case 2 (symmetric): x is left child of p, p is right child of g, uncle is black.



Before inserting x:
p: black height = black height of x
u: black height = black height of $\alpha 4$ or black height of $\alpha 5$ (or for the symmetric case, black height of $\alpha 1$ or black height of $\alpha 2$)
g: black height of u + 1

After inserting x:
p: black height = black height of x
u: black height = black height of $\alpha 4$ or black height of $\alpha 5$ (or for the symmetric case, black height of $\alpha 1$ or black height of $\alpha 2$)
g: black height of u + 1

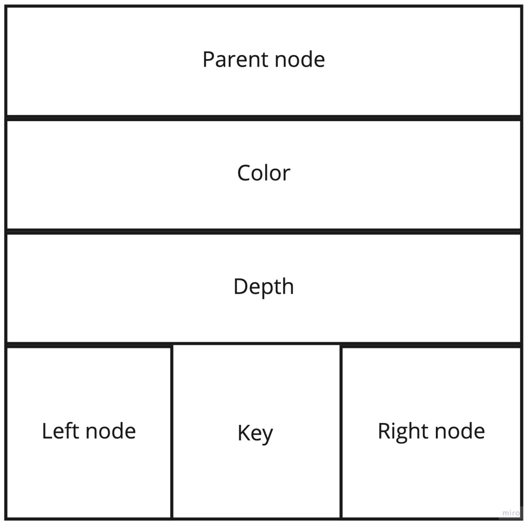
The black heights and colors will not change in Case 2 after rotation. So, if Case 2 happens, it will take constant amount of time O(c). After Case 2, we have to perform Case 3 because of red-red node conflict. Since Case 3 also takes constant amount of time O(c) that we found above, Case 2 still takes O(c) amount of time.

Examining updating black heights and colors time complexity:

If Case 2 and 3 happens, it will take constant amount of time according to our analysis. However, if Case 1 happens, it may keep happening because of the conflicts of the nodes that I explained in my analysis. It will take O(lgn) amount of time. So, in the worst case, we have O(lgn) for inserting the node and O(lgn) for updating the black heights and colors of the node. So, in total we have O(lgn) + O(lgn) time complexity. Which is O(lgn). So, this augmentation does not increase the asymptotic time complexity of inserting a node into an RBT in the worst case.

2 Problem 2

If we augment the depths of nodes as additional attributes in the nodes of the RBT, our nodes becomes as below image



The root node of the RBT has a depth of 0. To calculate the given node y's depth, we have to look at its parent x node. Y node will have have a depth of (depth of x + 1).

Examining insertion time complexity:

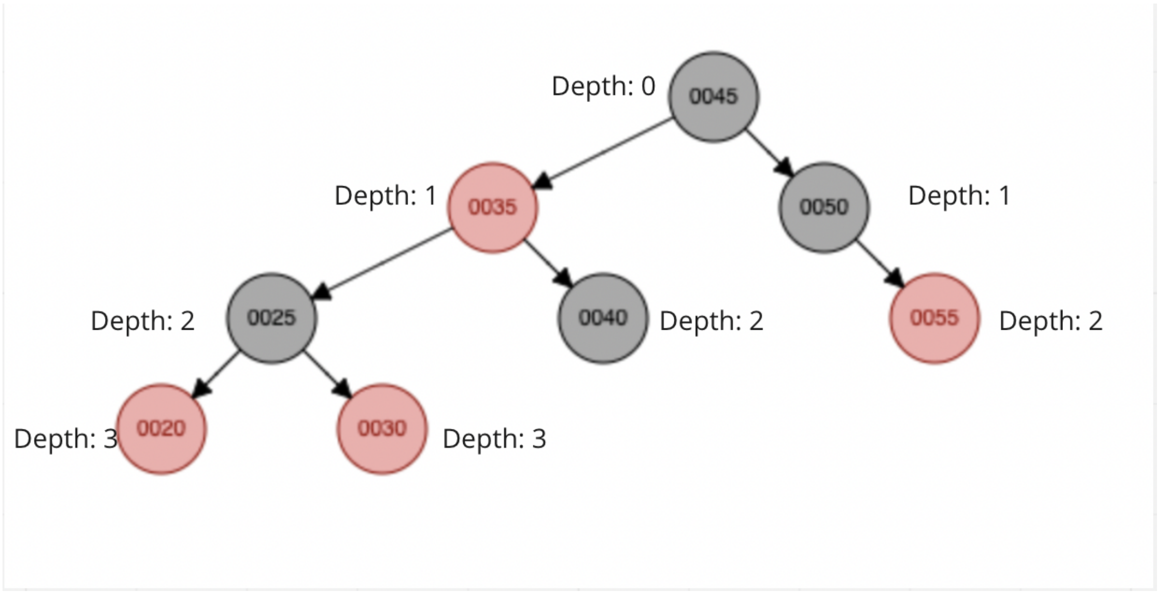
Like I explained in the Problem 1 inserting a node takes O(lgn) amount of time. Changing colors take O(c) constant amount of time. However, if we want to update the depths of the nodes after insertion, the time complexity will change according to the Cases.

If we insert a node and only case 1, 2 or 3 happens the time complexity will be O(lgn) because insertion will take O(lgn), updating color and updating the depths will take constant O(c) amount of time because we only have to change constant amount of nodes' depths and colors. Different from Problem 1, instead of black-height property of the nodes we have to change the depths of the nodes and their subtree's depths if any rotation happens. Updating the depths will be performed on constant amount of nodes.

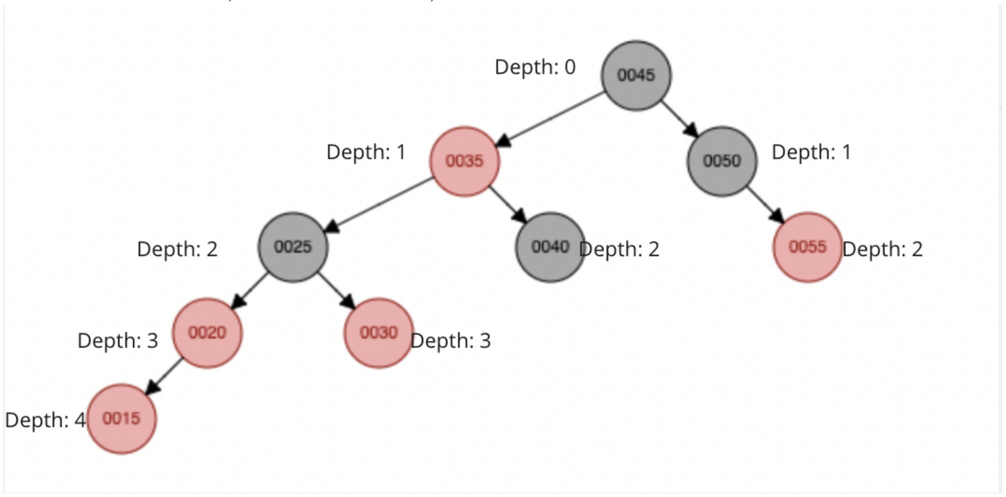
However, as I explained in the Problem 1, if Case 1 happens, it may cause red-red conflict with the parent node and it may cause Case 2, Case 3 or Case 3 to happen in the upper level nodes. If any path from root to leaf has a different amount of black nodes, or we have a red-red conflict, we have to re-arrange the root because of to conserve the balance for RBT, and the tree should be left or right rotate with Case 2 or 3 according to the problem we have after Case 1. Since the root changes, all nodes' depths should be updated. So, we have to traverse all nodes and update them accordingly. This action will take O(n) amount of time. Therefore, our total time complexity of inserting and updating depths takes O(lgn + n) times which is O(n) linear time. In conclusion, in the worst case, our time complexity increases from O(lgn) to O(n).

Example:

Let a tree with insertion order be {45, 50, 35, 55, 40, 25, 20, 30}:

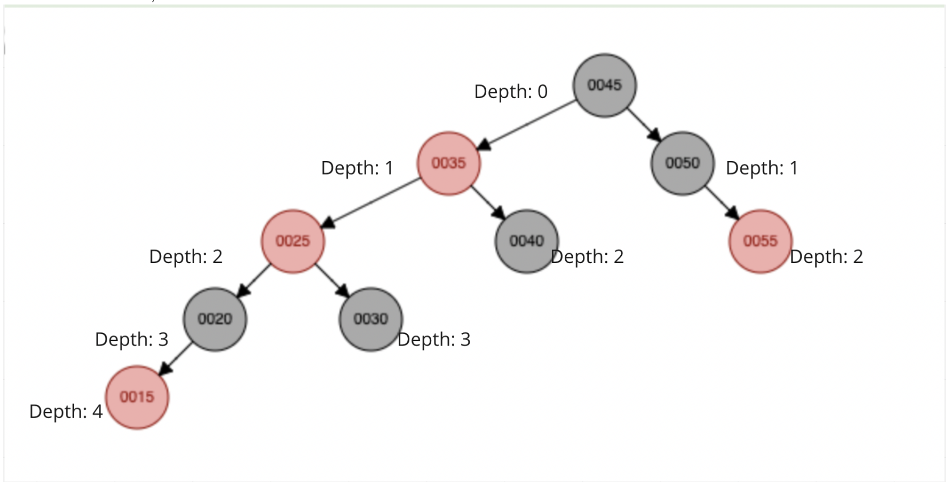


Now lets insert 15 (our tree becomes):



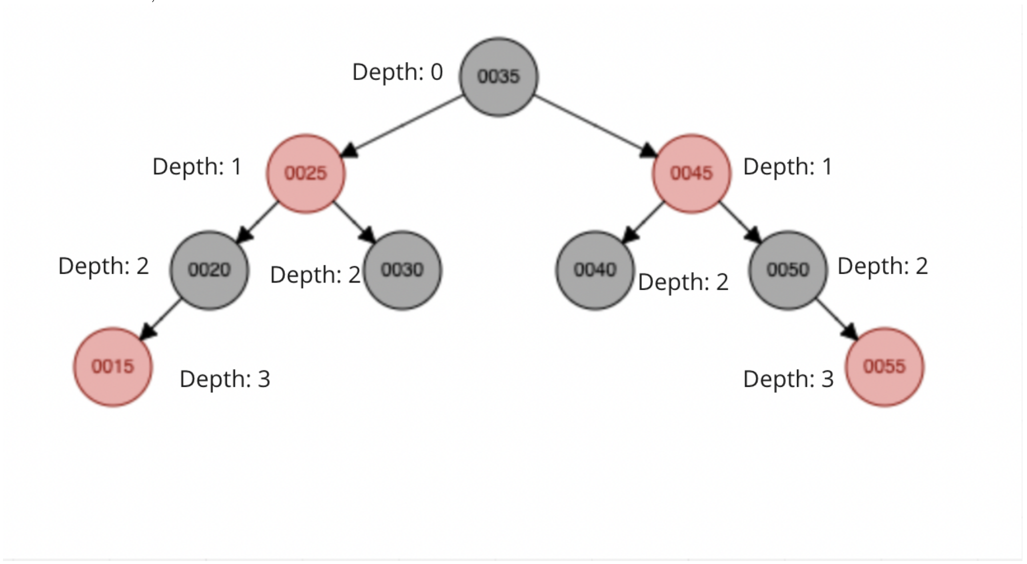
Inserting 15 will take O(h) where h is the height of the tree amount of time which is O(lgn). Since we are not changing any location of other nodes, determining node 15's depth takes only O(1) amount of time because e only have to look at its parent depth and add 1. However, red-red conflict happens between the new node 15 and 20. 15's parent is red, its uncle node 30 is red and its grandparent node 25 is black. So, Case 1 need to happen.

After Case 1, our tree becomes:



The depths of the nodes are still the same but the colors have changed. Changing colors of the nodes will take O(1) amount o time as I explained in Problem 1. So, again re-red conflict happened between 25 and its parent 35. Node 25's uncle node 50 is black and node 25's grandparent node 45 is also black. So, Case 3 needs to happen. There has to be single right rotation.

After Case 3, our tree becomes:



If we compare our depths of each node of our tree after Case 1 and Case 3, we can see that every node except node node 40's depth has changed. The reason is we changed the root of the tree with this rotation. So, our time complexity becomes O(n) since wee change each node's depths in the worst case.

<u>Conclusion:</u>

In conclusion, in the worst case scenario, after inserting a node which takes O(lgn) amount of time, Case 1 can happens and there might be red-red conflicts again and we may have to perform Case 1, 2, or 3 for the upper nodes to preserve the RBT properties. So, these conflicts may cascade up close to the root and we may have to left or right rotate. Therefore, root can be changed. If the root changes, we have to update all of the depths of the nodes. This takes O(n) amount of time because we have to traverse all nodes. Therefore, as I explained above our total time complexity of inserting and updating depths takes O(lgn + n) = O(n) linear time. In conclusion, in the worst case, our time complexity increases from O(lgn) to O(n).

<u>References:</u>

Lecture slides and https://www.cs.usfca.edu/ galles/visualization/RedBlack.html for visualization