CS301-A3

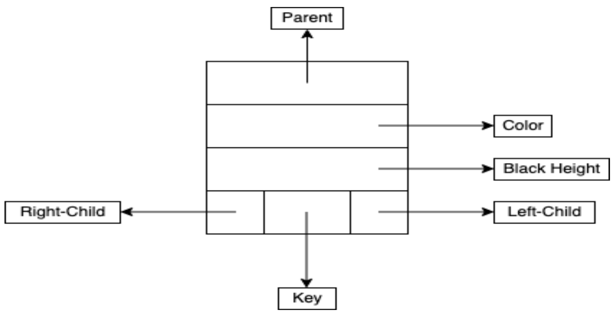
Görkem Yar (Student)

March 2022

1 Problem-1

To find a black height of a given node in constant time, we need to add black height of that node as an additional attribute. This new attribute required to be preserved in the insertion and deletion operations without changing the algorithmic complexity of these operations in the Red-Black-Tree.

Firstly, after the modification in the Red-Black-Tree node it will be shown as:



To calculate the Black height of a node one need to look at the children of this node. Because of the Red Black Tree property siblings need to have same black height. Therefore, they generate two different cases.

Consider x is the node that which we are trying to calculate its black height (consider y is the child of x):

1- If the child of x is a black node then: Black height of X = Black height of y + 1

1- If the child of x is a red node then: Black height of X = Black height of y

The second part of the question is showing the insertion operation. The insertion operation can be examined in two parts:

1- Insertion to the correct place (like the Binary Search Tree)

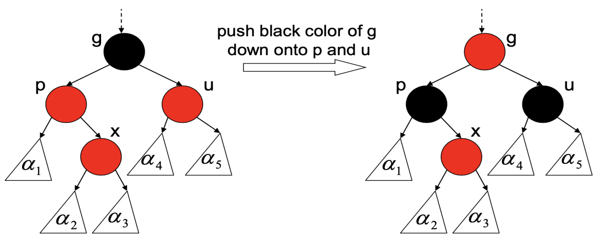
2- Fixing Coloring conflicts by making recoloring and rotations. In this step, while we fixing coloring conflicts, we will update the respective black height of the respective tree nodes.

The first step is similar to the Binary Search Tree, so it will take O(height of the Tree) time. By the Red Black Tree properties we know that Height of the Tree is in O($logn$). In this step, when a new node inserted to a tree, the black height of that new node will be 1 because of the arbitrarily inserted null black nodes.

After finding the correct place for the new node, we need to reorganize coloring of the tree when we do the recoloring and rotation operations the black height of the respective nodes will be updated. There are 3 different cases for fixing coloring conflicts as well as reorganizing the black height of nodes in a tree.

Case I:

In this case the node that is going to be inserted is x. When we insert x, there will be a red parent red child conflict and the uncle/aunt of the newly inserted node is red as well.



To solve this issue we will only make recoloring. In this case we will update black heights as:

For node p and u:

The black height of those nodes have not changed since black height of their children does not change. As a result, we do not need to update them.

For node g:

Black height of g = black height of p + 1

OR

Black height of g = black height of u + 1

Since heights black height of a leaf nodes to a common ancestor can not be differ. Both of these operations are possible.

Other nodes: Black height of the other nodes did not change.

Updating the height of G takes constant time since we just called a function that take constant time (return black height of a tree from p or u). Only the black height of the g has changed additional to the normal Red Black Tree insertion.

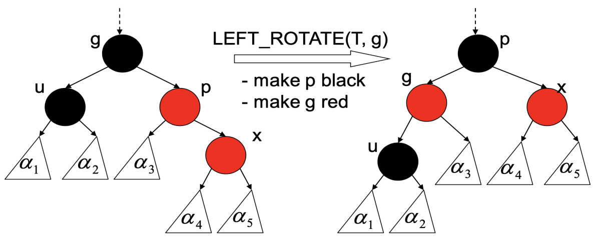
When we examined overall black-height of the subtree, it does not change because the black height of the tree was equal to Black height of a4 +1. And after these operations it is still equal to Black height of a4 + 1. Apart from this, the Case I may cascade to upper levels since g was a black node before and it was not conflict with its parent for sure. However, after insertion g became a red node and it may conflict with its parent. Therefore this Case may cascade to upper levels.

Since recoloring and updating black height of the node G takes constant time. Case I takes constant time in a single level. However, it may cascade to upper levels. So the overall worst case algorithmic complexity for the Case I to solve black height and color conflicts is O(height of tree)= O($logn$).

The total algorithmic complexity of the insertion operation in the Case I would be O($logn$)+O($logn$)= O($logn$). (The first O($logn$) cames from the finding the correct place for the new element.) So, adding additional black-height attribute does not change the algorithmic complexity of the insertion operation in Case I.

The symmetric case for Case I, The algorithm is same, just the situation is symmetric complexities are same.

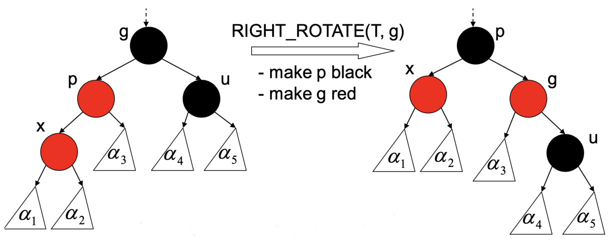
Important Note: Since Case I cascading a red node to upper levels, this situation may occur in the actual root of the Red Black Tree. In that case, we need to reupdate the color of the root as black which increases the



black height of the tree. Since updating a single node takes constant time this situation does not change the algorithmic complexity of the insertion operation.

Case III:

In this case the node that is going to be inserted is x. When we inserted x, there will be a red parent red child conflict. Also, we need to consider that this conflict situation might be transmitted from case I because case I may transfer the conflict to upper levels.



To solve this issue we will make rotations and recoloring. In this case there is not any update in the black heights of the nodes because of the followings:

For node g:

Before insertion: Black height of g = black height of u

After insertion: Black height of g = black height of u

As a result, we do not need to make update for the black height of node g.

For node P:

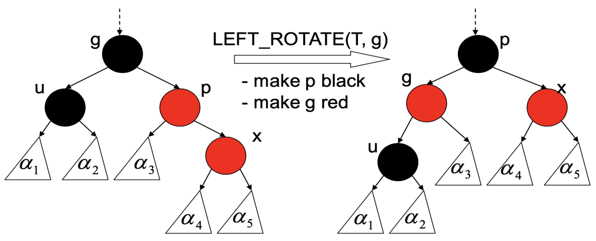
Before insertion: Black height of P = black height of x

After insertion: Black height of P = black height of x

As a result, we do not need to make update for the black height of node p.

For the other nodes: Black height of the other nodes did not change because of the recoloring and rotations. Since we did not make additional updates-operations in case III there is not any additional cost of this operation.

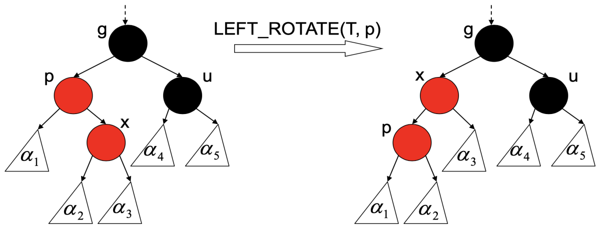
The symmetric case for Case III, The algorithm is same, just the situation is symmetric complexities are same.



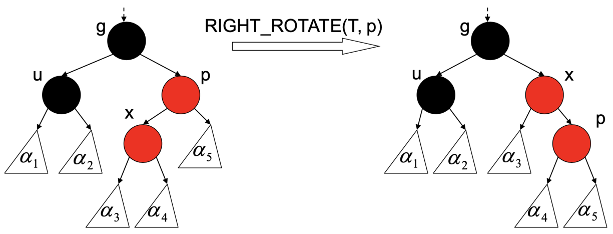
Apart from these, the third Case does not cascade to upper levels since the black height of the given subtree did not change. Before the rotations and insertion the black height of this subtree was equal to Black height of u + 1. And after these operations it is still equal to Black height of u + 1.

Since there is not any additional operations in Case III the algorithmic complexity of the insertion operation does not change. It is still equal to O($logn$). So, adding additional black-height attribute does not change the algorithmic complexity of the insertion operation in Case III.

Case II: This case is actually a copy of Case III except for the first rotation operation. And that rotation operation takes O(1) time so this Case takes the have same time complexity as Case II.
Additional Rotations are as follows:
Main Case II:



Symmetry II:



In the rotations between P and X the black height of the nodes does not change because both of them are red.
Overall Evaluation: All of the three cases did not change the algorithmic complexity of the insertion operation. In the worst case (Case I) we need to update the nodes in a path from root to a leaf. Even in that case since the update operation takes constant time the Algorithmic Complexity does not exceed O($logn$). As a result, we can add black height of the tree as an additional attribute without changing the Algorithmic Complexity. The main reason that lays behind why the algorithmic complexity does not change is rotation operations do not effect the whole black height structure of the tree. It does not cascade to upper or lower levels in rotation operations.

2 Problem-2

In this question, we are required to access the depth of a node in constant time to do that depth of a node will be added as an additional variable to the node. In each insertion and deletion operation, depth of the some of nodes will change. What will be examined in this question is whether keeping depths of nodes updated does change the algorithmic complexity in insertion operation.

The updates in depths in a insertion operation can be examined two parts separately:

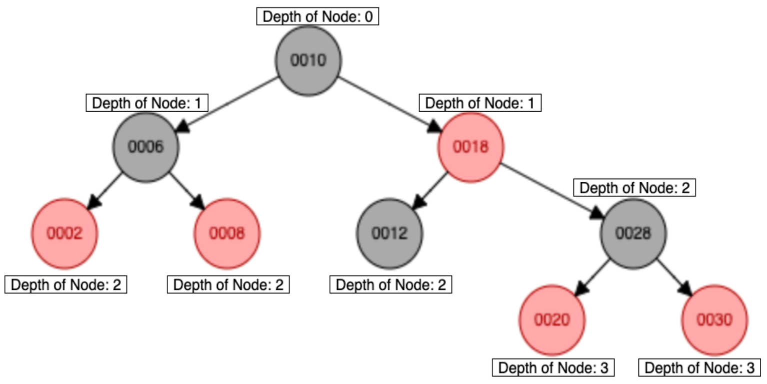
1- The calculation of depth for the newly inserted node.

2- Rearranging the depths of nodes that changed due to rotations in the insertion operation.

The first step will take O($logn$) since the depth of a newly inserted node can be calculated while finding a correct place for the insertion.

For this question, a Red Black Tree of key values 6, 18, 10, 28, 8, 12, 20, 2, 30 (inserted in this order) will be exercised to illustrate the depth of nodes in insertion operation.

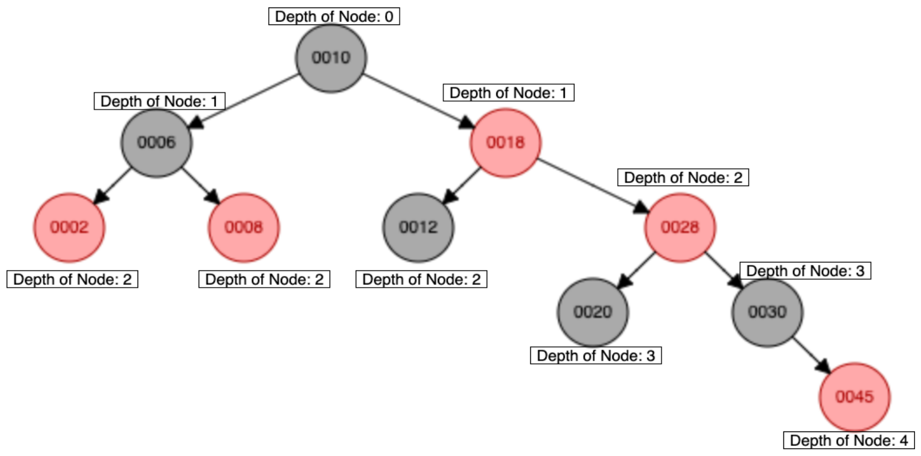
The initial version of this tree (after adding these elements to make a better illustration):



Now, if 45 is added to the tree the insertion steps would be: It is going to be added as a right child of 30 and since the added node is a red node there will be a conflict; as a result, the Case I of insertion operation will be executed.

In this part of the problem, the only thing that is needed to be calculated is the depth of newly added node. And while finding a correct place for that node, calculation for the depth can be made. Therefore, the calculation of the depth for a new node takes O($logn$).

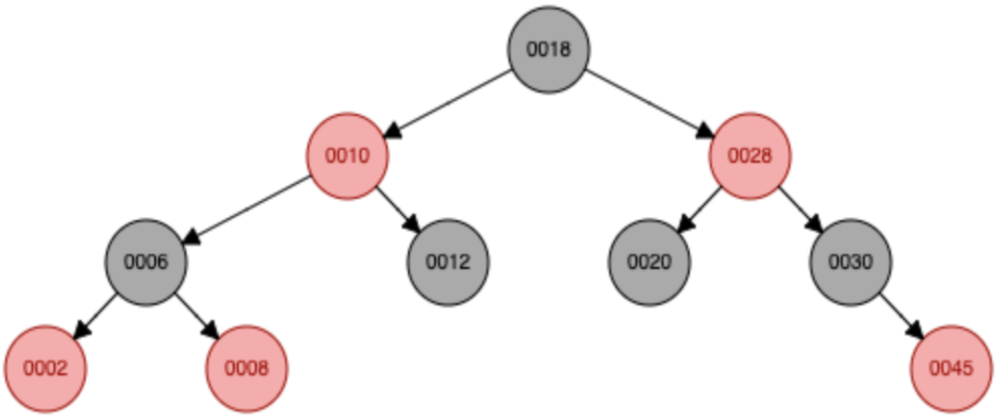
The following figure that represents the first part of the insertion, is in the next page. (After the first conflict Case (Case I) is solved)



For this step specifically, the algorithmic complexity is recoloring and calculation of depth for new node and finding the correct place for the new node. In total it is in O($logn$).

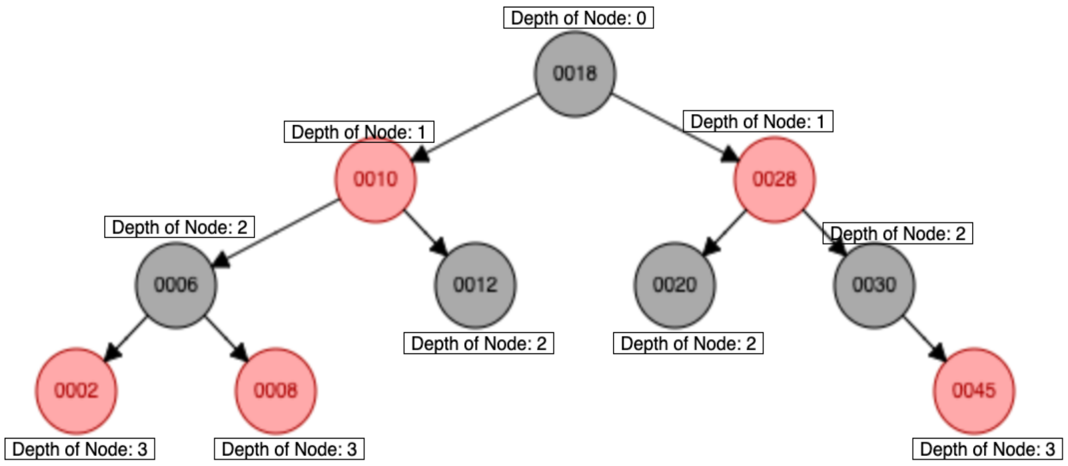
Since the Case I is used in the first step of the algorithm, the red-red conflict is transmitted into the upper levels of the tree. To solve this, the properties of the Case III will be used. Since, there is a rotation in Case III everything will be change. The new root of the tree will be 18, and 10 will be a left child of the 18. Also, 12 will be the right child of 10.

!!!Important Note!!!: When the root is rotated, depth of every node except for the root's right child's left child has changed. In this example the respective node is 12. The depth of 12 and depth of 12's children have not changed. Apart from 12 there are other 3 nodes which are 2, 8 and 28 in the same level. So in the worst case running time, we can assume that number of nodes in the subtree of 12 is less then the number of nodes that is not belong to subtree 12. Therefore, we can say the number of nodes in the subtree of 12 is $\leq$ n/2. Which leads us to claim there is at least n/2 nodes that change their depths <u>in the worst case</u>.



Now the depth of at least n/2 nodes has changed because the root is changed. For the nodes that is left part of 18 the depth has increased by 1 (except for 12). On the other hand, for the nodes that is remaining at the right part of the 18 the depth decreased by 1. To update these nodes we need to iterate and update their depths. Since we are going over at least n/2 nodes. Even though updating operation take constant time for a node, there are at least n/2 nodes so the total algorithmic complexity of this update operation will take n/2* O(1)= O(n) time. O(n) dominates the normal insertion algorithmic complexity which is O($logn$). As a result we cannot hold a depth of node as a attribute without effecting the algorithmic complexity of the insert operation.

After updating the heights the tree will be shown as:



As it illustrated, in the worst case running time, to keep depths of the nodes up to date, at least half of the nodes revisited again (In this specific example every node except 12 revisited again). Therefore the worst running time of the insertion became O(n) which increased the algorithmic complexity of the insert.

3 References:

-Lecture Slides