CS300 Assignment

Alpay Naçar

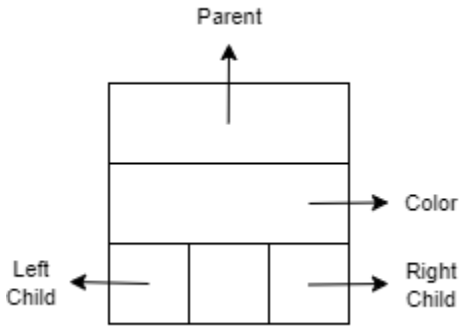
31133

November 2023

Purpose of RBT is to limit height with O(logn), where n is node count. So that dynamic set operations such as insert, delete, are going to be done in logarithmic time.

Normal RBT properties:

RBT is a specialized binary tree, and all binary tree properties apply to RBT, but there is more.



Every node has a color (it is either red or black), and the root is always black.

Every null child pointer can be thought of as an invisible black node, so that every leaf node is black.

A red colored node has two black colored children. (a red colored node can't have a red colored child, this is how we understand if we need to do a rotation or not) (There can't be more than h/2 red colored nodes in a path between root and leaf)

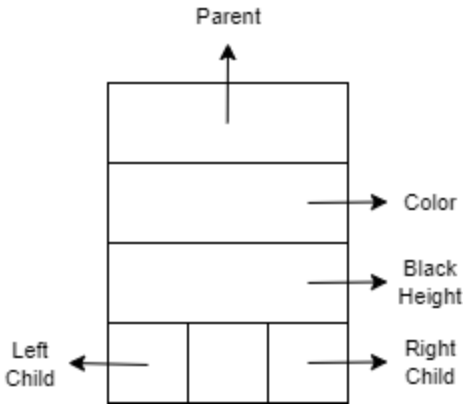
Every path from the root to a leaf has the same number of black nodes. (Black height of siblings are the same)

Using the last two properties, we can say that distances from root to all leaves is between black node count of a path from root to leaf, and 2 times of this count (included red colored nodes to the counter). Therefore if we can do all operations without violating these rules, we can say that the tree is self-balanced.

Problem 1

RBT with Black Height Augmentation:

To augment black heights into the RBT. We need to store them as a new attribute in our nodes. Space complexity is the same because when there are n nodes, total space was (O(n)) c1*n (n is amount of nodes in RBT). With the black height attribute, space complexity becomes (c2 = c1 + size of black height integer) c2*n, and only constant changes, therefore asymptotic space complexity is same. You can see the old RBT, and new RBT (with black height as another attribute of the node) in the drawings. We can access to black height in constant time if we store it in our RBT and update it accordingly after every operation made in RBT accordingly. Black height change after all possible operations will be shown below.



What is black height:

Black height is the amount of black colored nodes from current node to the leaf of current subtree, and it is same for all paths between current node and the leaves (in this subtree) that you can reach from child nodes (not parent nodes). Black height of a node is same as its childs black height and add one if current node is colored black.

In RBT the number of black node count is always the same when we count them from root to leaves. This characteristic of RBT is protected from rotations by some control mechanisms. (rotations will be explained later) We can use this fact to prove that both child's should have the same black height always. We know that ancestors of both these childs are the same, therefore upper part of the root to leaf path is the same. And using the fact that in RBT root to leaf path's black node counter are the same, we can say that in both childs, there should be the same number of black nodes till leaves. Therefore, the black heights are the same. One other important feature of RBT is you can't have a red colored node with a red colored child, this is cause of balancing mechanism.

There are two important things I need to check before I can say that this black height can be added, and it can be used in constant time. This black height attribute should be maintainable, which means that no other operation such as insert or delete should break it. And it shouldn't increase my asymptotic worst case time complexity to maintain this attribute.

How does self-balancing work in RBT:

When a new node is inserted or a node is deleted, from the point you made the change, you go up until you cannot go up further (root has no parent), and do recoloring if needed to preserve RBT properties,

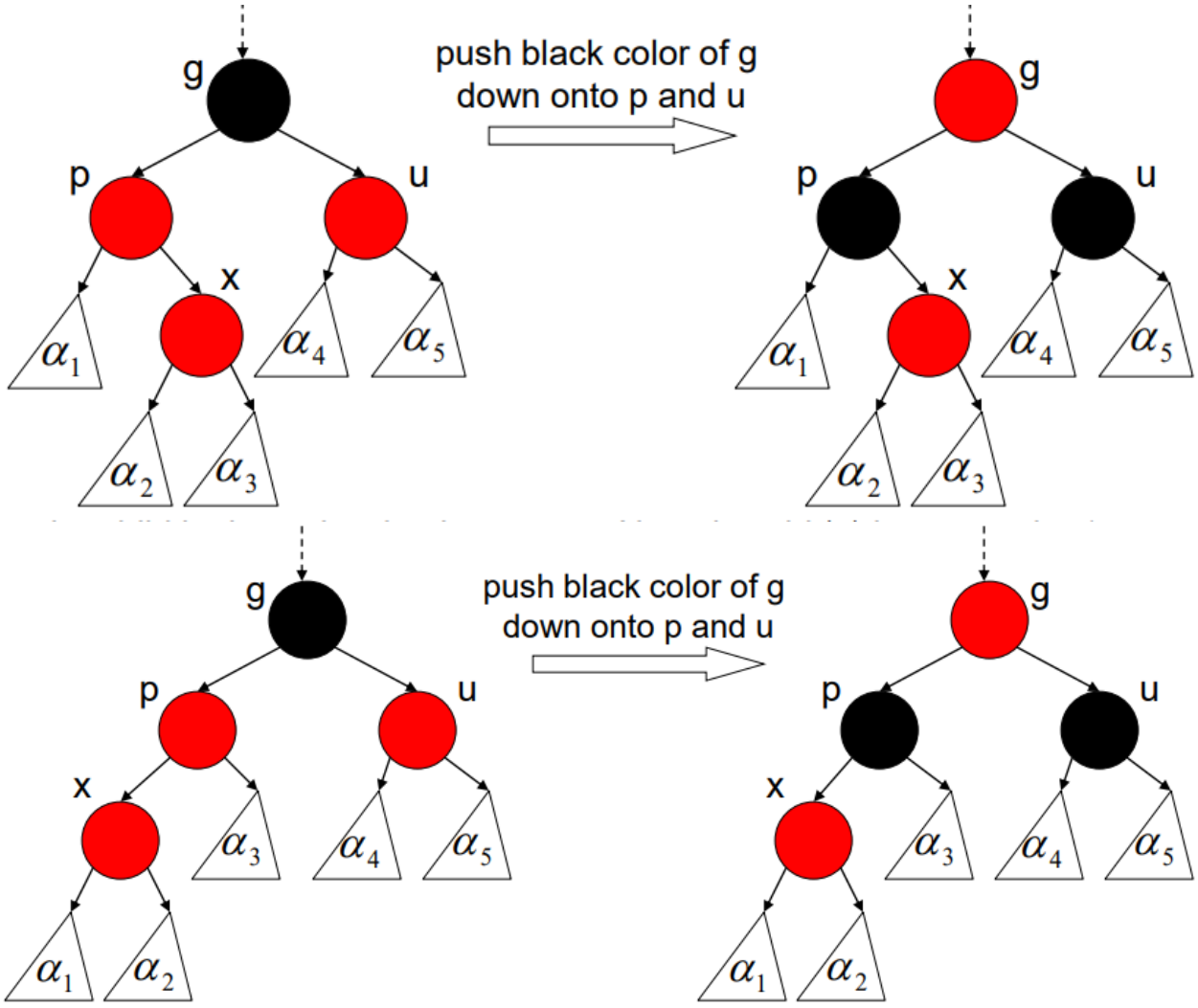
and sometimes structure of the tree will be modified (rotations) in order to protect these properties of RBT. For example, when there are consecutive red colored nodes in the tree, some operations on the tree should be done in order to fix this. This operations should be done in O(1) time, so that there is maximum O(logn) operations, from the changed node to the root, therefore total time complexity of these changes is going to be O(logn).

Insertion

RBT add operation starts just like a normal Binary Tree insert operation. We add the node we want to add to the location that it should be, the color of this node is initially red, but if it's the root, we change it to black (because of the property of RBT, root can't be colored red). Till this moment black height of any node not changed because we only added a red node and are not counting red nodes, we are counting black nodes. After insertion of this node, some recoloring and self-balancing operations need to be done in order to not violate any rules of RBT. Starting from the node that we inserted, we need to go up till the root and check if there is any consecutive red nodes. There can't be consecutive red nodes in an RBT, but this rule can be breached when insertion made, so we need to control it and fix it if necessary. (Consecutive red nodes indicate that the part having consecutive red nodes might be outweighing other part)

Here are the four cases where there is a red colored node as a child of another red colored node. In the images, x is the current node, p is the parent node, g is the grandparent node and u is the uncle node (grandparents other child than parent). Also, we know that grandparent exists because parent is red and red colored node can't be the root therefore there is upper nodes, and we know that uncle exists (at least there is invisible black node that we put to null childs) and we can rotate to that side if needed. All of the rotation operations, we only need to update black height of nodes under the operation (and the nodes we did the operation too), black height of g will be the same and we upper nodes will have black height calculated using g as I explained in properties of RBT.

Case 1:

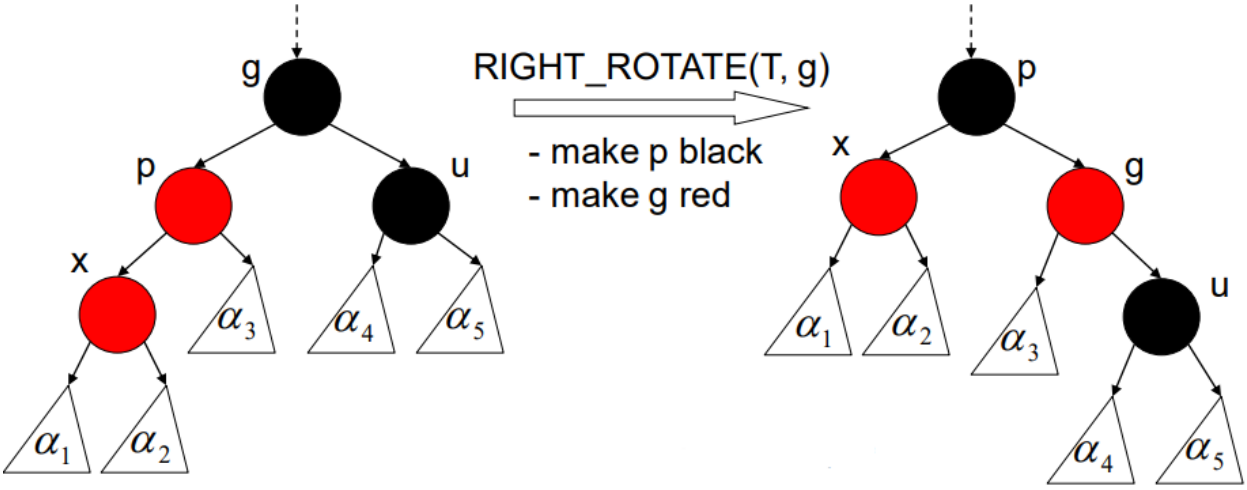


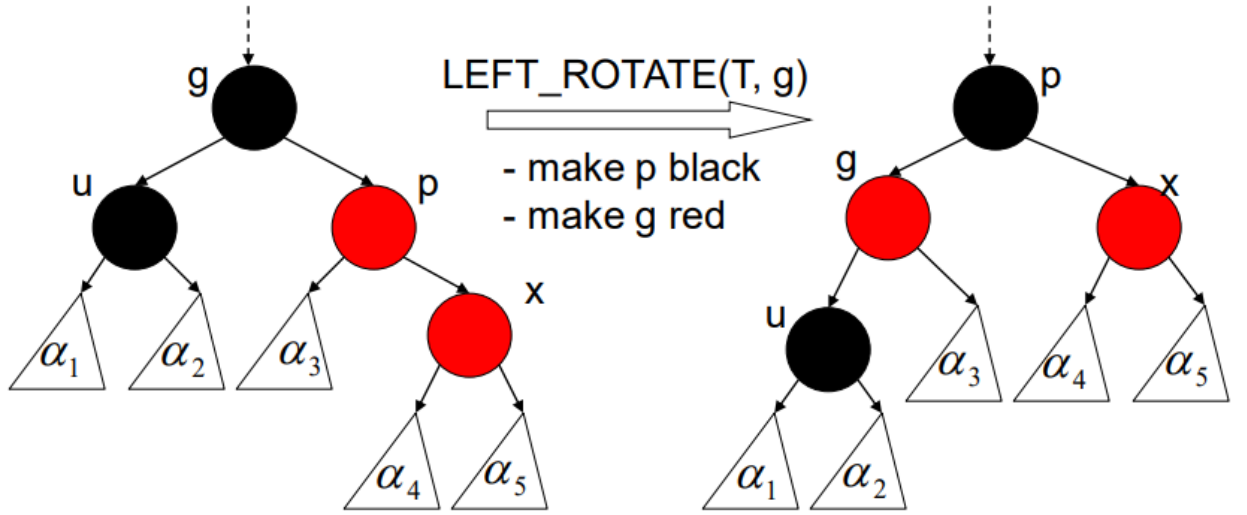
If the uncle of the child in the red-red pair is also red. Child being left or right child of parent doesn't matter, so it applies for both situations. As you can see from the images, the black color of g can be pushed down to p and u, so that p and u will become black, and g will become red. Making g red can cause some other problems on the upper part, so we will continue going upwards to the root to check if ins later iterations. We might need to do another recoloring and or rotation.

We handled the case where uncle is red colored. In case 2 and case 3, uncle is black. We need to do rotations in order to fix the violation in case 2 and case 3.

Black height of nodes in a1, a2, a3, a4, a5 was same before we start recoloring and rotation, and we don't need to update them because we only move a black color from their upper node to one level down, but number of black nodes till the root is still the same. Node g's black color is still the same as well, even though it lost black color, child having black color will balance it. Nodes p and u's black height will be incremented by 1. As you can see black height operation can be done in constant time in case1.

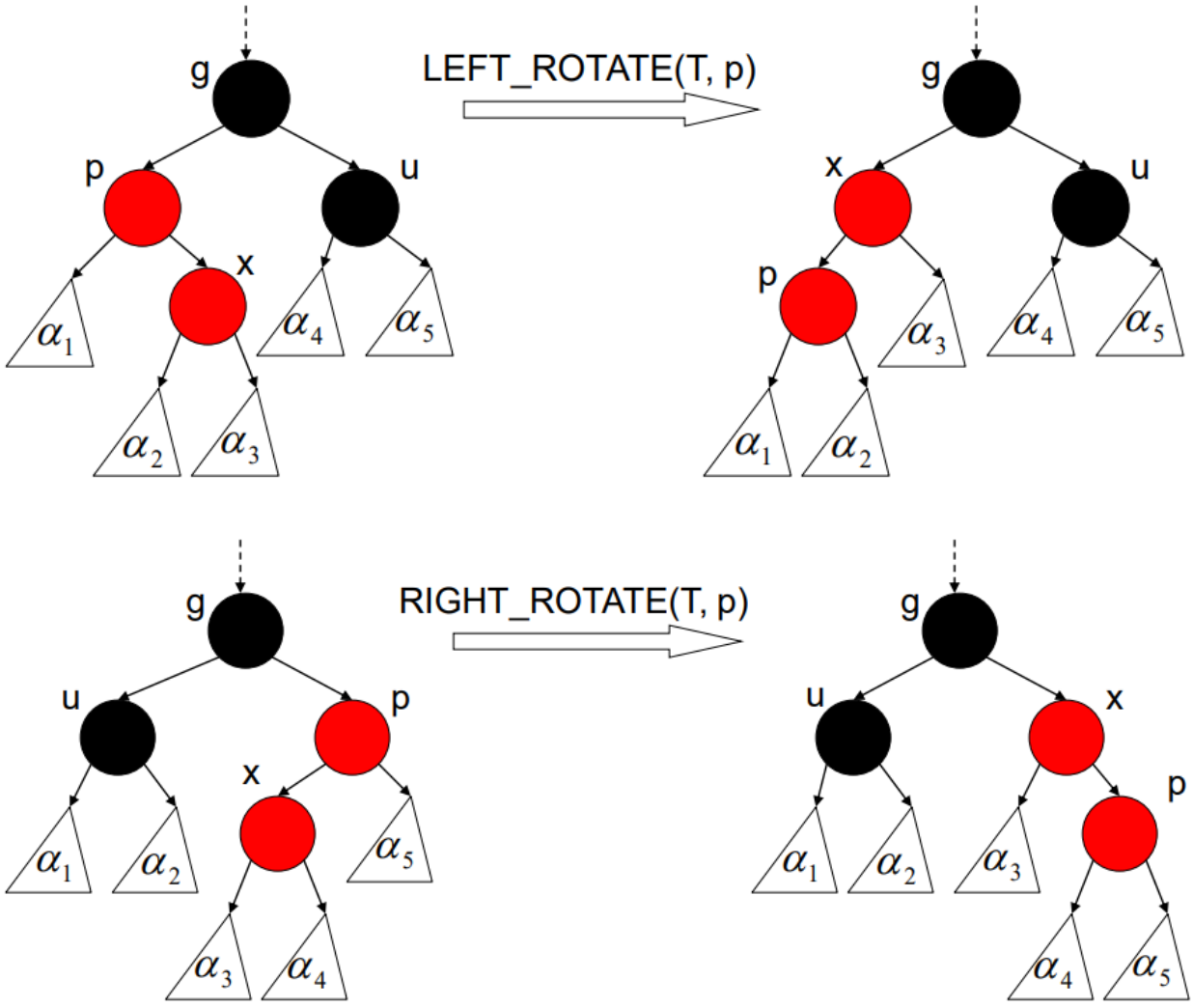
Case 3:





We know that uncle is not black. If x is left child of p and p is left child of g, or x is right child of p and p is right child of g (parent and child is leaning to the same side), rotate to the other side (the one that childs are not leaning). We can do this rotate operation in O(1) time. As you can see from the images, in all subtrees a1, a2, a3, a4, a5, we don't need to change any node's black height. Also we don't need to change x, p, g, u's black height's because black nodes before and after the rotate operation both under them and upper side of them are the same. As you can see, black height operation can be done in constant time in case3.

Case 2:



We know that uncle is black, and p and x are leaning to different sides (explained meaning of leaning above). Node x will rotate to side of p in this case. The side of uncle won't affected with this operation, and because both p and x is red, we don't need to update p, x, or any node under these two. After we make this operation, we can use case3 so that we can fix red-red pairing. As you can see, black height operation can be done in constant time in case2.

Problem 2

We can add depth as an attribute to the nodes just like we did in previous problem to black height. But to determine if this can be maintainable without changing time complexity, we should consider some cases. Finding one case where depth cannot be preserved is enough to say that depth cannot be added to reach depth of a node in constant time.

Insertion

After we insert a node, first we need to determine its depth. In order to do that, we can count it from root to the inserted node, or we can increase this counter while coming from root, instead of doing this same traversal again. In both ways, we can find this nodes depth in O(logn). Our previous time complexity was also same, so this operation is not increasing asymptotic time complexity of inserting a node to RBT.

Then we need to check if we need to update any other nodes depth. We know that depth is calculated using nodes above, so we only need to change nodes under this inserted node, and because inserted node is a leaf of this tree, there will be no nodes which should be updated after this insertion.

Then we will do recoloring and rotations if needed to preserve RBT properties. I explained them in previous problem with details. Here is depth specialized explanation.

Case1:

We are not rotating in case1, therefore we do not have anything more than the normal implementation. Depth doesn't cause any problems in this case.

Case3:

With this rotation, all nodes in that subtree of g should change their depths, and we can't do that in O(logn) time. Best we can do is to traverse all subtree and redetermine each nodes depth. This can be done in O(n) time, but it will increase our time complexity to O(n) so we cannot do this operation without an increase in asymptotic time complexity.

Showing one case where we cannot preserve is enough but let me show Case2 just in case.

Case2:

After the operation a1 and a3 in left rotate, a3 and a5 in right rotate should be updated, this cannot be done in O(logn) time just like case2. Furthermore, case3 will be done after this case and case3 runs in O(n) time.

Best we can do is calculating depth of all nodes after all rotations done, so that we won't update depths of same nodes again and again, but this still requires O(n) time.

In conclusion, depth cannot be implemented without an increase in asymptotic time complexity.