CS 301 - Assignment 3

Rebah Özkoç
29207

April 17, 2022

Problem 1 (Black Height)

To compute the black-height of a given node in a red-black tree in a constant time, we can store the black-heights of nodes as additional attributes in the nodes of the RBT without increasing the asymptotic time complexity of inserting a node in the worst case. When we make an insertion to the red-black tree we need to update the black heights of the nodes that are affected.

Black height of a node is equal to the number of black nodes along the path from the node to a leaf under in the sub-tree rooted at that node. As a property of the red black trees, every path from a node to its leaves have to have same number of black nodes. Thus, every node has only one black height as an attribute and color fixing operation of the red black tree ensures that this property holds.

How to Compute the Black Height of a Node?

Let node x has two children x1, x2 and x1 is black, x2 is red. Let black height of the node x1 is $BH(x1)$ and black height of the node x2 is $BH(x2)$. We can compute the $BH(x)$ by two ways.

$$BH(x) = BH(x1) + 1 = BH(x2)$$

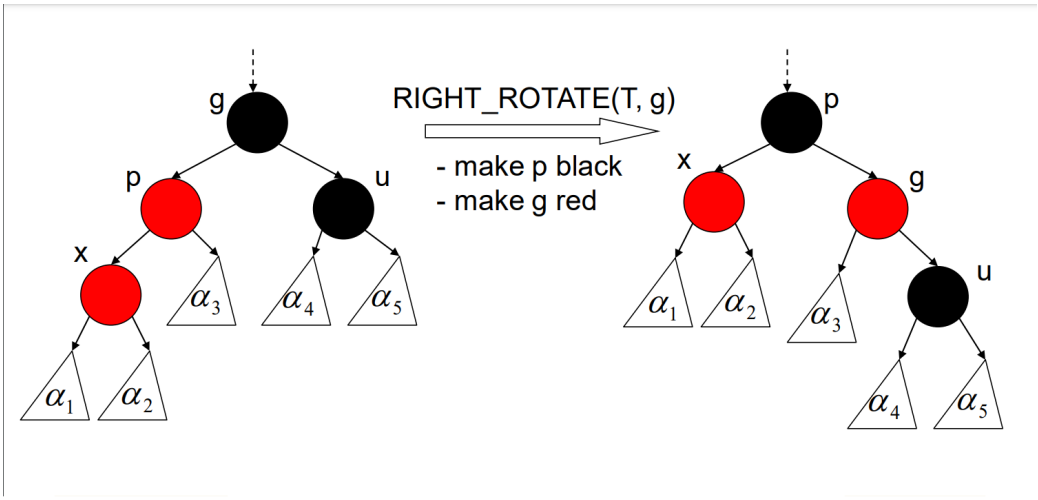
These two ways give the same result always. We can conclude that if we now the black heights of the children of a node, we can compute the black height of the node without any further information.

Case-by-Case Analysis

When inserting a new node to the RBT, firstly we need to find the position of the new node as if we are using a BST like a regular RBT. The color of the new node will be red. We can find the black height of the new node by just checking the children of the new node and applying the explained method in constant time.

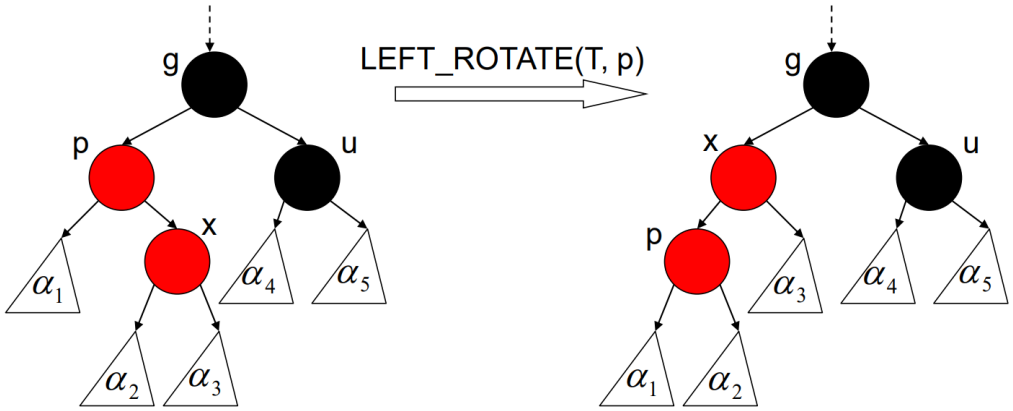
After this insertion we need to fix the coloring if any of the properties of the RBT are violated. While fixing the colors, we can fix the black height attributes of nodes if there is a problem. After we add the new node there are 3 cases (and their symmetric cases) that violates the RBT property of the tree.

Case 1:
New node x is left child of p, p is left child of g, u is black.



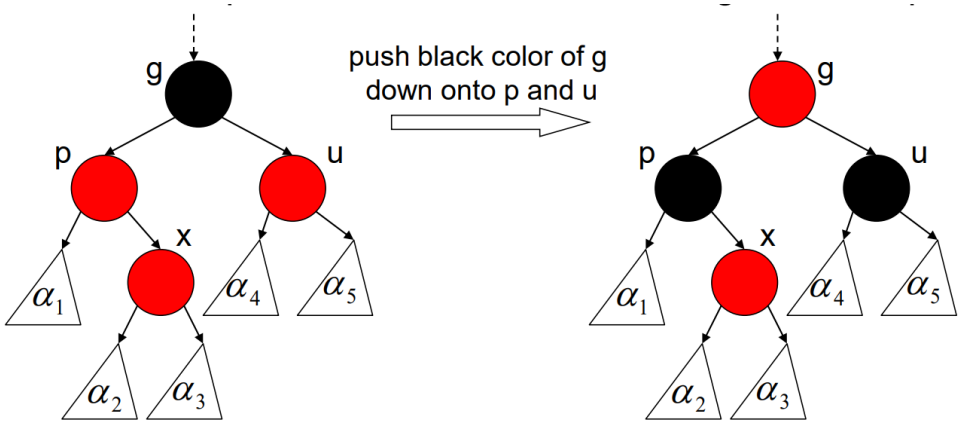
When this case happens, as you can see from the diagram we make right rotation, and p becomes the new root the sub-tree. In this case the black height of the sub-trees a1, a2, a3, a4, a5 stays same. Black height of u also do not change because black height of their children do not change. Black height of the g also do not change because one of the children of the g is still u and BH(u) did not change. After we add the node x black height of p did not change because the new node is red. Thus none of the heights of the did not change in the graph and we don't need to make any modification to the upper nodes in the graph in this case. Overall, this augmentation does not increase the asymptotic time complexity of inserting a node into an RBT in the worst case in the first case.

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



When this case happens, as you can see from the diagram we make a left rotation, and we get the case 1. In this case the new node is red hence the BH(p) and BH(g) do not change. Also BH(u) did not change because we did not make any change on u and its children. After that point we implement the same algorithm that we used in the case 1. Overall, this augmentation does not increase the asymptotic time complexity of inserting a node into an RBT in the worst case in the second case.

Case 3:
The uncle/aunt of the child in the red-red pair is also red, and the new node x is the right child of p.



When this case happens unlike the other two cases we increase the number of black nodes by 1. As you can see from the diagram we make the parent and uncle nodes of the new node x black and their parent g becomes red. In this case black heights of a1, a2, a3, a4, and a5 stays same. Also black heights of p, u, and x stays same because the black heights of their children stayed same. But the black of the node g increased by 1. We can maintain the black heights of nodes in the diagram in constant time. But since we change the color of node g, another red-red violation can happen with g and its parent. In this case we can solve the problem like we did in the first, second, or third case. In the worst case case 3 happens again and again and goes until the root. If this happens we need to change as many as the height of the tree node's black height attribute. In the worst case we need to change black height of $O(height) = O(lg(n))$ nodes. Overall, this augmentation does not increase the asymptotic time complexity of inserting a node into an RBT in the worst case in the third case.

Symmetric Cases:

Besides these three cases, there are three more cases that is the symmetric versions of these cases.

In the first case new node x is right child of p, p is right child of g, u is black. In this case the analysis works same with the first case.

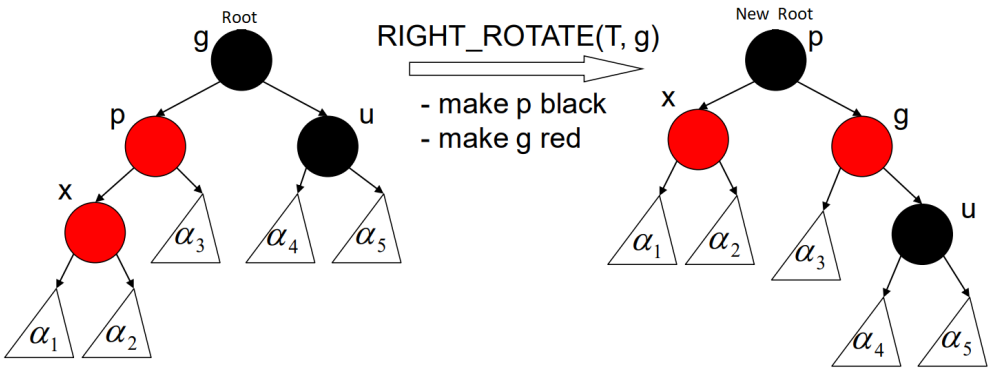
In the second case new node x is left child of p, p is right child of g, uncle is black. In this case the analysis works same with the second case.

In the third case the uncle/aunt of the child in the red-red pair is also red, and the new node x is the left child of p. In this case the analysis works same with the third case.

Problem 2 (Node Depth)

To compute the depth of a given node in an RBT in constant time, we can augment the depths of nodes as additional attributes in the nodes of the RBT. But this augmentation increases the asymptotic time complexity of inserting a node into an RBT in the worst case. We can show that by an example. The depth of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0 in RBT. We can compute the depth of a node by adding 1 to the depth of its parent.

If we can find a case that depth of every node in the tree change than we can say that we can not preserve the $O(lg(n))$ time complexity in the worst case. Consider the RBT in the diagram.



In this case we add the new node x as the left child of p. In this case before the rotation depths are like that:

$Depth(g) = 0$
$Depth(p) = 1$
$Depth(u) = 1$
$Depth(x) = 2$
$Depth(a1)'s = 2 + localdepth(e.g. 1, 2, ...)$
$Depth(a2)'s = 2 + localdepth(e.g. 1, 2, ...)$
$Depth(a3)'s = 1 + localdepth(e.g. 1, 2, ...)$
$Depth(a4)'s = 1 + localdepth(e.g. 1, 2, ...)$
$Depth(a5)'s = 1 + localdepth(e.g. 1, 2, ...)$

After the rotation the depth of the all nodes on the left side of the g decreases by 1 and the depth of the all nodes on the right side of the g and g itself increases by 1. New depths are like this:

$Depth(g) = 1$
$Depth(p) = 0$
$Depth(u) = 2$
$Depth(x) = 1$
$Depth(a1)'s = 1 + localdepth(e.g. 1, 2, ...)$
$Depth(a2)'s = 1 + localdepth(e.g. 1, 2, ...)$
$Depth(a3)'s = 1 + localdepth(e.g. 1, 2, ...)$
$Depth(a4)'s = 2 + localdepth(e.g. 1, 2, ...)$
$Depth(a5)'s = 2 + localdepth(e.g. 1, 2, ...)$

In this case we need to change the depth of every node in the tree except a3 and its children nodes. Since we want to find a counter example let's assume that a3 has same or lower number of nodes than a1, a2, a4, and a5. Assume that

number of nodes in a1 = # nodes in a2 = # nodes in a3 = # nodes in a4 = # nodes in a5 = k

$$n = 5k + 4$$

In this case we need to change the depth of $(4/5) * n + 4$ nodes. This is clearly exceeds the $O(log(n))$ asymptotic time complexity bound and it belongs $O(n)$ asymptotic time complexity. Overall, this augmentation increases the asymptotic time complexity of inserting a node into an RBT.

References

Lecture Slides of CS 301 Algorithm Course