CS 301 A 3

Oytun Kuday Duran / 28357

November 2022

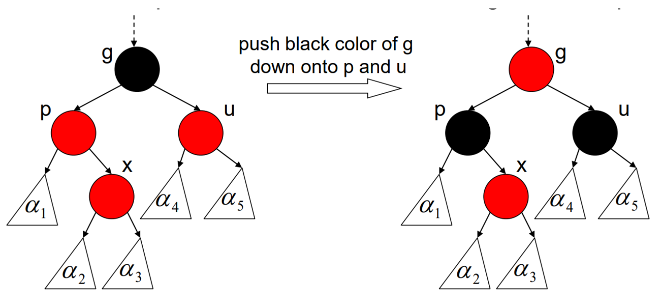
1 Problem 1

To get the black height of a RBT node in O(1) time, it is needed to add an extra information field to the node to store black height information. Moreover, since all siblings have the same black height, to calculate the black height it is only needed to check a child node. If the child of the node is black, then black height of the parent is 1 more than black height of the child. If it is red, then the black height of the parent is same as child's.

To perform the insertion operation on RBTs it is needed to insert the node just like it is done in Binary Search Trees, and then perform the recoloring operation to preserve RBT properties. Insertion on a BST takes O(h) time, since it is a RBT, h=log n and it takes O(lg n) time. Moreover, new node will have a black height of 1 due to the black nodes at the leaves.

After inserting the node, we need to do recoloring and rotation operations. There are 3 different cases as discussed in lectures:

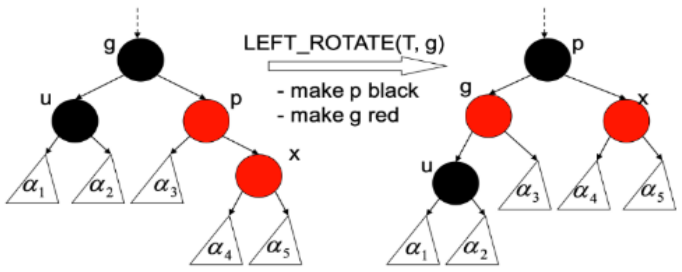
1.1 Case 1



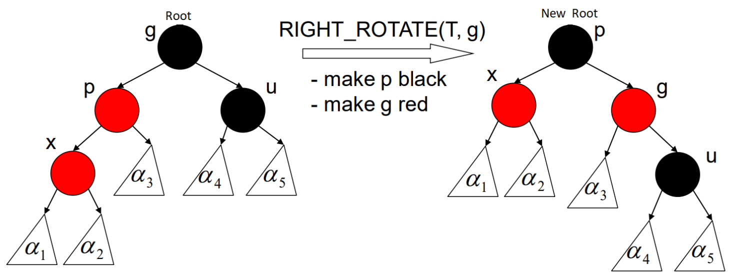
In case 1, we insert x in the figure above. There will be a conflict because p and x are both red, one red node must have black child. We need to recolor and push black color from g to p and u. The black heights of p and u nodes in figure does not change since we don't change color of any nodes below them.

It is not needed to update (black heights of a1,a2,a3,a4,a5 does not change). However, for g node, since p and u became black, black height of g will change. Therefore, we will update it as p->bh +1 or u->bh +1 since both siblings have same black height. Shortly, there is more black nodes below g now. Updating height of g which is the only one affected in figure takes O(1) time since we will only make 1 call to add it 1 or assign it as 1 more than p or u. Overall black height of the subtree also does not change since it is still a1->bh + 1 as can be seen in the figure. In this case, upper node of g also may be red and red black red child conflict property may be violated again, so we may also need to iterate to upper RBT. In worst case, it may cascade till the root. In case 1, solving red black conflict and black height takes O(log n) time in worst case. So, in case 1 insertion, worst case asymptotic complexity would be O(log n) for finding correct place for node insertion and O(log n) for resolving conflicts, and O(log n) + O(log n) = O(log n). As can be seen, as it is a constant time operation, and black height of the other nodes in the tree such as nodes below p/u won't change, it doesn't change the complexity of case 1 insertion. In addition, symmetric of case 1 is same as fundamental operations are same, and cascading may affect root, so we also may need to update root as color black, in this case, black height of the tree may get increased by 1. Still, as it takes O(1) time to update a single element, the overall worst-case complexity still will be O(log n).

1.2 Case 3



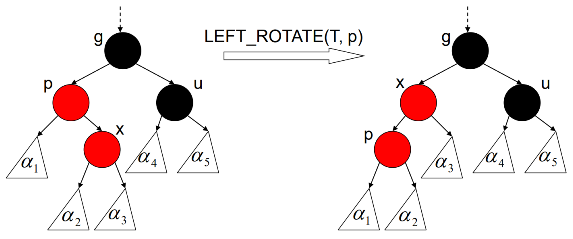
In case 3, when we insert x, there will again be a red parent red child violation which can be solved using rotations and recoloring. Moreover, case 3 might occur from case 1 while going to upper layers while cascading. In this case, after insertion, black height of node g which is equal to u->bh + 1, won't change as can be seen in figure since black nodes under g won't increase. For p which is equal to x->bh, it is similar. We won't need to update the black height of p and g nodes as a result. Black heights of other nodes than p,g,u and x won't change as a result of recoloring and rotation operation in case 3 since there aren't any additional rotation and recoloring operations afterwards. As a

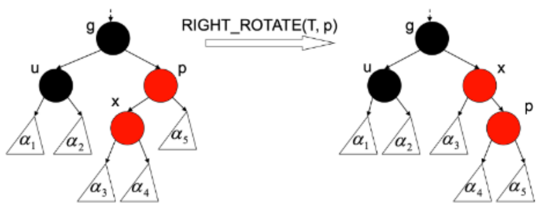


result, worst case asymptotic complexity of the insertion still will be O(log n). For the symmetric case in the figure below, since algorithm is same, complexity and the interpretation of black heights will be same. Since case 3 won't cascade as it did in case 1, total black height of the subtree in figure will still be the same after insertion which is u->bh +1 . As a result, since there is no additional operations needed and updating black height of a node takes only O(1) time, case 3 insertion operation worst case asymptotic complexity will still be O(log n). Black height attribute won't change the worst case complexity in case 3 too.

1.3 Case 2

Case 2 is similar to case 3. Only difference is the first rotation operation which takes constant amount of time (O(1)) in BST/RBT, so the worst case asymptotic time complexity is still O(log n). The case 2 operations and the operations for the symmetric case can be seen in figures below.





In both symmetries of case 2, rotations between p and inserted node x black heights of the nodes still stay same because both nodes are red. Case 3 will apply here again. Notice that black heights of any nodes haven't changed from rotations since any black heights below them didn't change.

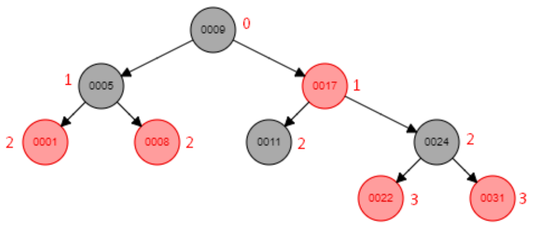
1.4 Conclusions for all 3 cases

For case 1, 2 and 3, the worst case asymptotic complexity of the insertion operation didn't change after the addition of the extra black-height attribute. In the case 1 which may have needed the most amount of iterations when we may need cascading update operations, since updating black height of a node takes constant time, the worst case complexity did not exceed O(log n). We can safely add another attribute namely black height to a RBT without changing the worst case complexity of the insertion operation because the rotation operation does not affect all nodes and black height structure of the RBT and it does not cascade to other levels in the RBT for rotation operations (Also, in case 1, at worst it is still O(log n)). Moreover, since in symmetric cases, since we are doing the identical operations in different directions, it is the same aswell.

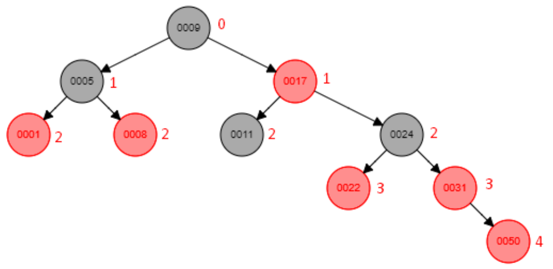
2 Problem 2

In this situation, it is needed to store depth of a node similar to previous question as an attribute to RBT node struct in order to get a node's depth in constant amount of time. In previous question, insertion rotation operations didn't change overall black height structure of the tree so it did not increase the worst case asymptotic complexity of insertion operation with extra attribute. However, in the depth case, after insertion operations, depth of some nodes will get changed. It is needed to understand whether updating depths of affected nodes will affect worst case complexity or not.

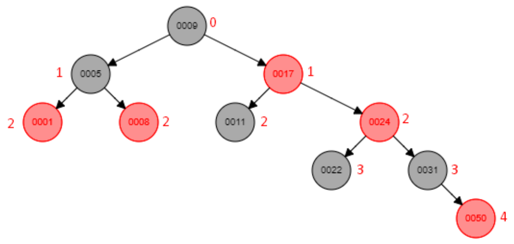
One can divide the inspection of this operation in 2 seperate parts namely calculation of depth for the new node to be inserted and updating the depths of affected nodes after the insertion rotations. The first part of insertion of a node and calculating its depth will take O(log n) which is same time as insertion since one can calculate depth of new node while finding a place for inserting it to the RBT in constant amount of time.
For illustrating with an example, I will use a tree that I generated using RBT generator of University of San Francisco (You can access it from references.), and red values next to a node denotes it's depth. The initial tree will be as figure below which has orderly inserted values 5, 17, 9, 24, 8, 11, 22, 1 and 31:



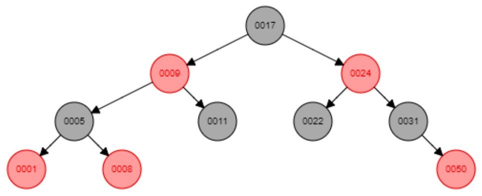
If we wanted to insert a very big value to this tree such as 50, firstly, it would be the right child of 31 since it is the largest before any operations. Moreover, the initial tree would be as the figure below with new node having a depth of 4 (previous node +1), also there will be a red-red conflict since initial coloring of node with 50 will be red:



Because of red parent red child conflict (Case 1), after the first recoloring operation initial RBT would be as:



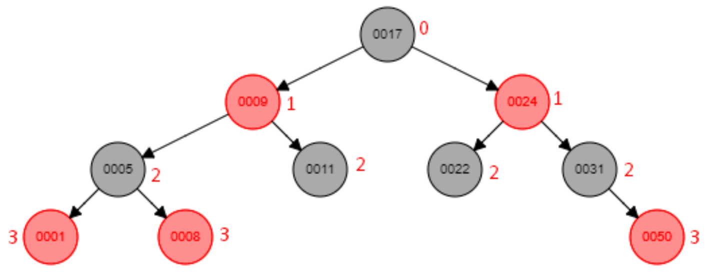
As can be seen, in this part, before any rotations are performed, only thing needed is to calculate the depth of new node which can be done while finding the correct place to insert it. Therefore, the depth calculation and finding correct place takes O(log n) time. From now on, after case 1 conflict is solved, since red-red property violation is transmitted the upper levels of RBT, we will use case 3 operation as you can see in the figure above. It can be noticed that due to rotation operations, there will be more work to done because the structure of the RBT will change. The initial RBT after first rotation will be as figure below:



As can be seen, new root is 17, new left child of root is 9 and right child is 24. As you can see now in the figure, depths of nodes needs to be recalculated since the structure of RBT changed. As root changed, in worst case, depth of at least n/2 nodes has changed (n is the number of nodes), except for 11, all nodes at the left subtree of the root is incremented by 1. Every node in the right subtree of root is decreased by 1. To update all of those nodes, we need to iterate through new root and update respective depths. We need to go over at least n/2 nodes. Since update of a node takes constant amount of time c and there are at least n/2 nodes, the total algorithmic complexity of update operation would be O(c*n/2) = O(n) time. These update operations will dominate the insertion to RBT that is O(log n). If we add an attribute as depth to RBTs, we affect the algorithmic complexity of insertion operations.

Here, considering worst case and it can be observed than root− >left− > right (Let's call s) and it's subtree's depth didn't change, it can be assumed that subtree of s is less or equal to all other nodes except this subtree. so number of nodes at subtree of s is <= n/2. Therefore, our claim is at least n/2 nodes to change since depths of subtree of s won't change.

Furthermore, RBT after calculation of depths in O(n/2)=O(n) and updating accordingly, can be seen in the figure below:



3 References:

Lecture slides of CS 301 Fall 2022

https://www.cs.usfca.edu/ galles/visualization/RedBlack.html