CS301-Assignment 3

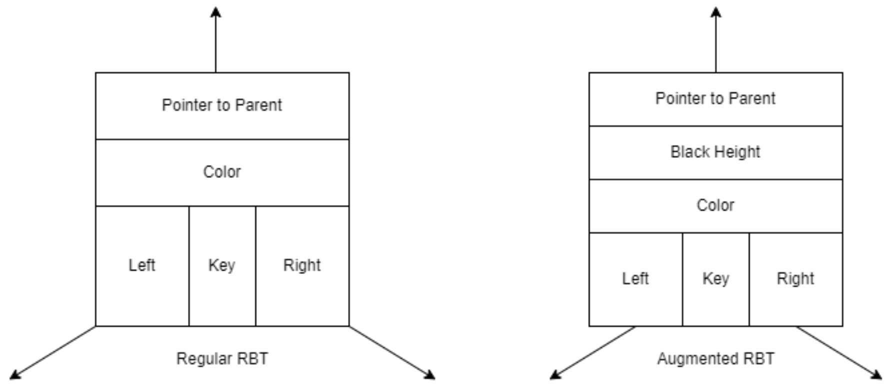
kaanalper-28147

November 2022

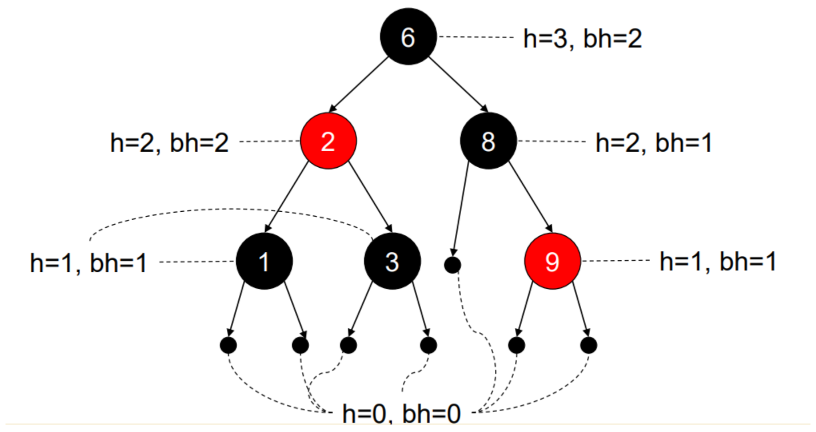
1 Question 1

1.1 Augmentation of RBT to compute black height of a given node in constant time

In the regular RBT, we keep the information about the pointer to parent, right, left nodes, color and the key value. However as it mentioned in the question, we need to create a augmented data structure to keep the information of black height in each node. It can be demonstrated as:



Augmentation of this data structure can be done in the generation step. First of all, while creating the tree, in each insertion, we need to keep the information of black height. To do this, we can directly assign black height by looking the previously inserted child's black heights. Since the child of newly inserted node has also black height property in our augmented data structure, we can inherit this information and manipulate it with respect to the child of newly inserted node. If the child is red, than we do not need to modify information. We can directly take the value of this node and assign it to out new node (parent of that node). Otherwise, if the child of our new node is black, we can take the (black height of child + 1) and assign it to our new node. If there are two children and they have different colors, than we can directly take the maximum black height across them and assign it to our new node. Each of these steps are constant time (addition, $\max(bh_{children1}, bh_{children2})$), thats why, without considering the complexity of insertion, just assigning the black heights is constant time. Example from CS301-Algorithms slides:



With this augmentation, we can directly reach to the information about the black height of given node without traversing. That's why it also takes O(1) time.

However, with the impact of the augmentation, we need to consider whether newly generated data structure alters the asymptotic time complexity of insertion operation

1.2 Insertion Operation in Augmented RBT

We can utilize the insert operation steps given in CS301 Algorithms RBT Lecture slides:

- Two phase approach while inserting
- First phase is to insert as it is inserted into BST
- The color of new node will be determined as red
- Fix the coloring to maintain the RBT property in second phase

In other words while inserting into RBT, 2 property should be preserved and considered:

1- BST property should be conserved
2- RBT properties should be preserved

1.2.1 Conservation of BST Property

BST property: all nodes in left subtree is smaller than parent and all nodes in right subtree should be bigger than parent. It is valid for all subtrees in BST. So, the insertion operation should be done in a way that this property will hold.

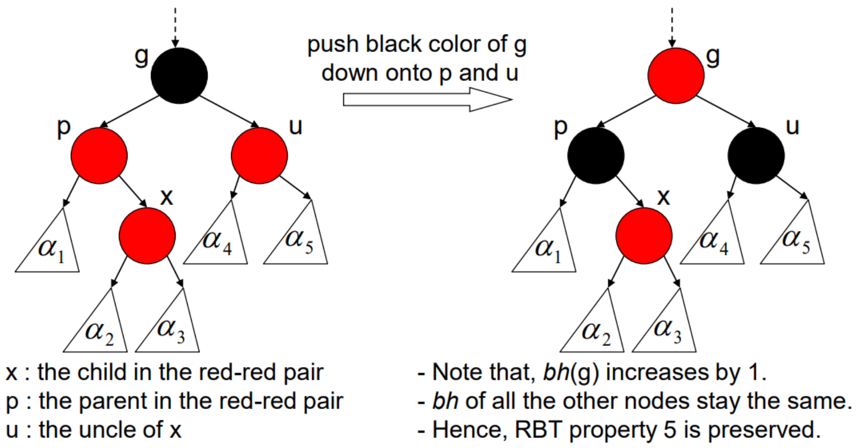
It is known that the dynamic set operations in BST costs $O(h)$ time where h is the height of the tree. Since in the worst case, binary tree's height can be n when there is n nodes, worst case time complexity becomes $O(n)$. However in RBT, in worst case, heigth become $log_2 n$. Thus, it takes $O(logn)$ times in worst case. Since the augmentation that we made is not affecting the principles of Binary search tree, it does not affect the worst case time complexity scenario. It just calculates the black height by looking at the previously inserted child nodes black heights and as it mentioned in the augmentation part it takes $O(1)$ time which does not change the overall complexity.

1.2.2 Conservation of RBT Property

As the newly inserted node determined as red, we need to consider whether there is red parent-red child conflicts and we have to fix if there is an issue like this.

In this part, we need to preserve the properties of RBT while insertion and also we need to look whether arranging the black heights affect the overall complexity. For that, we can examine 3 distinct cases of insertion for preservation of RBT properties. If we can prove that our augmentation does not affect the complexities of these 3 cases, we prove that the overall complexity is not affected by the augmentation. I will examine the cases as it was given in the CS301 Algorithms RBT Lecture slides.

Case 1: As it is mentioned in CS301-Algorithms RBT Lecture slides the uncle of the child in the red-red pair is also red, and x is the child of p.

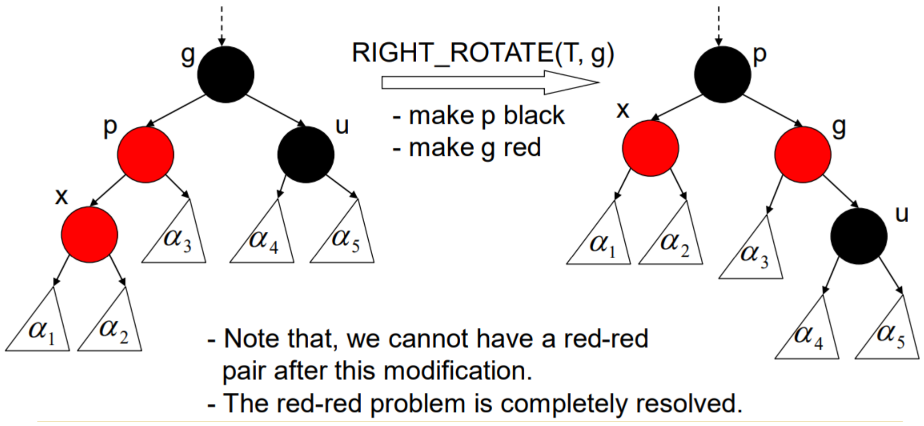

g
p
u
x
α_1
α_2
α_3
α_4
α_5
push black color of g down onto p and u
x : the child in the red-red pair
p : the parent in the red-red pair
u : the uncle of x
- Note that, *bh*(g) increases by 1.
- *bh* of all the other nodes stay the same.
- Hence, RBT property 5 is preserved.

As we found that inserting the new node in correct place for BST property to be hold is cost O(logn).

X is the node that we inserted. As we can see after the insertion, there will be a red-red pair problem exists between p and x. To solve this issue, we can change the grandparent's color to red and color of parent and uncle to black. This will solve the issue for particular subtree. However after we change the color of grandparent, it can cause another red-red pair problem between the parent of g and g. So this red-red pair problem can react up to the root where we need to fix all of these issues one by one recursively. So in each recursive step, changing the color of g, p and u takes O(1) time and maximum step count will be determined by the height of the tree. Thus, in worst case it takes $O(logn) \cdot O(1) = O(logn)$ times. So we found that in regular BST, it takes O(logn) to fix the coloring issue. However, we also need to consider the case where we add black height into our augmented data structure

Before inserting the X, all the black heights of nodes are arranged. After inserting X, first we need to consider the black height of X which is the max black height of x's childs. So $x_{bh} = max(bh(c_{a_2}, c_{a_3}))$ where c_{a_2} and c_{a_3} are childrens coming from subtrees a2 and a3. Finding the maximum of 2 numbers cost O(1). After this step, we need to look at p and u where their black height stay same since we do not add a black node. Last of all, we need to consider g and all recursive grandparent nodes. In worst case, each step of updating g cost constant time. There will be +1 increase in the g node due to the alteration of the color of p and u. But when we consider the recursive color fixing up to the root, all of the black heights will change and while changing the color we can also update the black height. Thus, it becomes $O(logn) \cdot O(1)$ as well. At the end our cumulative complexity coming from CASE 1 is $O(logn) + O(logn) = O(logn)$ where $O(logn)'s$ come from the inserting into correct place in BST and updating the colors as well as the black heights. In other words, our augmentation does not have an impact on the complexity of CASE 1. In the symmetric of case 1, the situation is same so we do not need to examine the symmetric case.

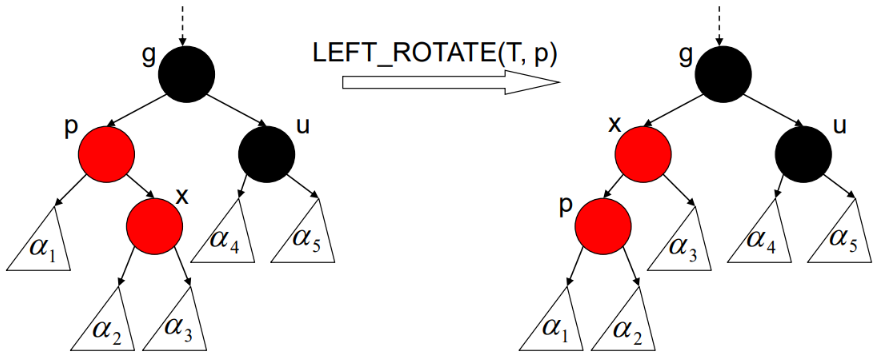
Case 3: As it is mentioned in CS301-Algorithms RBT Lecture slides newly inserted x is the left child of p and p is the left child of g.



In this case, we can rotate our tree to the right and change the parent as well as grandparent colors as shown in the figure. After this modification, there will be no existing red-red conflict. Because the grandparent of newly inserted node become black it will not led to another red-red conflict. Hence, there will be no need for recursive repair up to the root. So, fixing the coloring issue costs constant time in this example. However, we need to check black height for our augmented data structure as well.

Before inserting the X, all the black heights of nodes are arranged. After inserting X, first we need to consider the black height of X which is the max black height of x's childs. So $x_{bh} = max(bh(c_{a_1}, c_{a_2}))$ where c_{a_1} and c_{a_2} are childrens coming from subtrees a2 and a3. This step will takes constant time. Same rule is valid for the determination of black heights of x, p, g and u. So, the cost for changing the values of nodes after alterations costs constant time for x,p,g and u. However we need to consider whether these changes affect the upper nodes up to the root. Since the black heights of the a1, a2, a3, a4 and a5 do not change, the black height of the root of this subtree do not change as well after the rotation. So, we do not interfere to the upper nodes where overall complexity becomes $O(logn)$ that comes from finding the correct place for the node for preserving the BST property. At the end both our first subtree and rotated subtree has black height u+1. Algorithm result in same in symmetric scenario, so we do not need to look at the complexity of it (it will be O(logn) as well).

Case 2: Newly inserted x will be the left child of p and p is the right child of g where uncle node is black.



- It seems that we did not solve anything, we still have the red-red pair.
- Actually, we have transformed it into case 3, hence it will be immediately solved.

In this case there is zig-zag shape insertion. To ensure the RBT property we need to do 2 rotations. First of all, we need to rotate parents subtree such that we will make the x (our new node) parent. After that we continue with the process we covered in Case 3 (since the situation after rotation is same as Case 3). Since these rotations solve the issue without recursive tracking up to the root, they costs constant time. However we need to check whether they affect the black heights. In the first step there will be no alteration in black height because we just change the red-red conflicted pair locations which does not affect the black height. Second step which we cover in Case 3 does not affect the black height of overall three as well. Thus at the end computing the black height of sub-tree takes the constant time. At the end cumulative complexity for inserting a new node into our augmented data structure costs $O(logn)$ same as the Case 3.

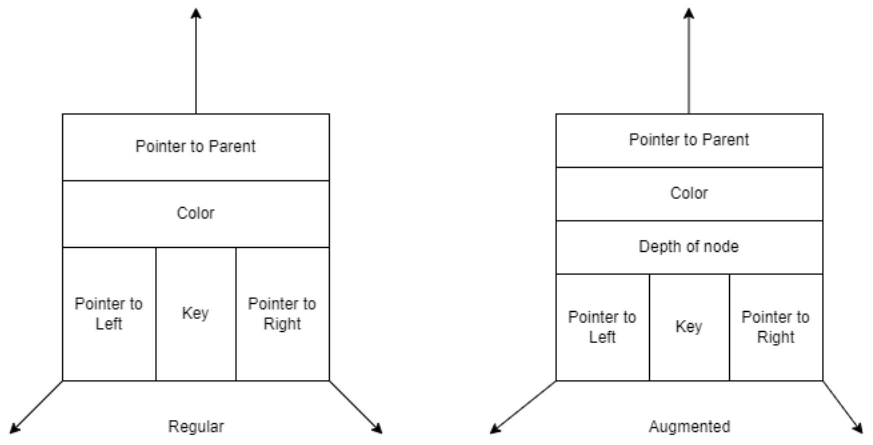
1.2.3 Overall Complexity for Inserting into Augmented RBT

We covered all three steps and as we can see, in each step our augmentation does not change the overall complexity of insertion. It will still be same as the dynamic operation of Regular RBT which is $O(logn)$.In Case 2 and Case 3 rotations does not affect the black heights. Even in the Case 1 (which recursively implement the operation up to the root), updating black heights does not exceed the $O(logn)$ complexity. Thus, our augmented data structure can be implemented without altering the complexity of insertion.

2 Question 2

2.1 Augmentation of RBT to compute depth of a given node in constant time

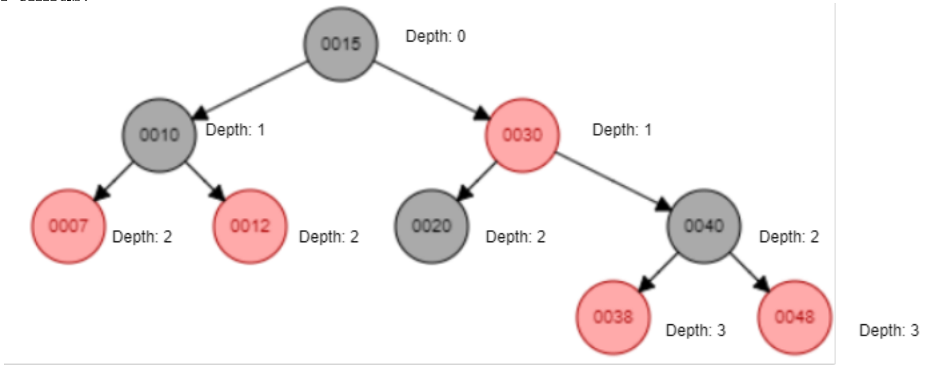
We will create an augmented data structure which holds the information about depth value of a given node. In that way we can directly access the depth value in constant time when the node is given. The regular and augmented RBTs are given as:



The depth of new node can be calculated by just looking at the parent node's depth value and add 1 to it. So newly inserted node's depth = parent's depth + 1.

The problem asks for why this augmentation increases the asymptotic time complexity of insertion operation in worst case. To illustrate this notion, we need to observe a rotation in the level of the root. The reason behind this concept is that all of the node's should be checked and updated after the root is rotated and the depth of the system changes. This phenomena led to the complete arrangement of new depths by traversing the whole tree that require the number of nodes steps (n). The situation of complete alteration mentioned can be observed with the impact of the enforcement of Red black tree properties. When a new node inserted and Case 1 happened up to the root and if it causes Case 2 or 3 at that level, this will impose the rotation of root and change of depth values. So at the end, it causes O(logn) for each insertion, O(1) for determining augmented depth value in each insertion (inheriting from parent), O(n) for changing all of the depth values after the root-level rotation mentioned. Cumulatively, it led to the O(n + logn + 1) complexity which is linear.

To illustrate this worst case notion, I will use the below tree with the insertion order of 15,30,10,7,12,20,40,38,48. Also, I assumed that all of the leaf nodes have null childs.



I used the visualization tool of San Fransisco University.

After the creation of tree we can insert our next value to observe what we encounter in the worst case scenario. To do that lets pick the new value as 35. When we add 35, there should be some cases that are considered specified in the explanation of 1^{st} question.

2.2 Insertion

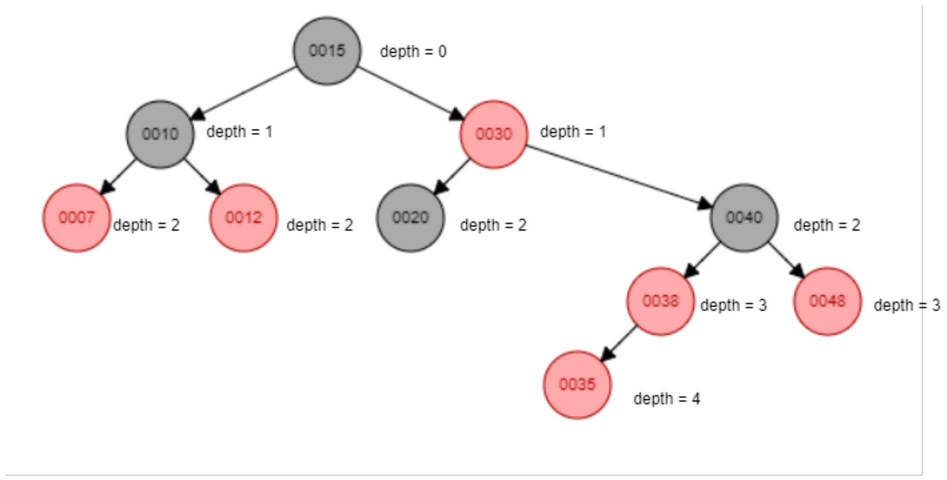
2.2.1 Finding the correct place for the value 35 and determining the value

As it was discussed in the first question, to establish the property enforced by the rules of BST, the correct place should be found. As it is mentioned, finding the correct place for insertion will take O($log(n)$).

2.2.2 Assigning corresponding depth value while inserting

While inserting, the depth value is directly assigned with inheriting the parent depth and adding 1 to it. (In the case of root it's directly 0). This step causes O(1) to compute depth and assign it to augmented data structure. In our example, 35 will inherit the depth value from 38 and add 1 to it which becomes $depth_{35}$ = 3+1 = 4

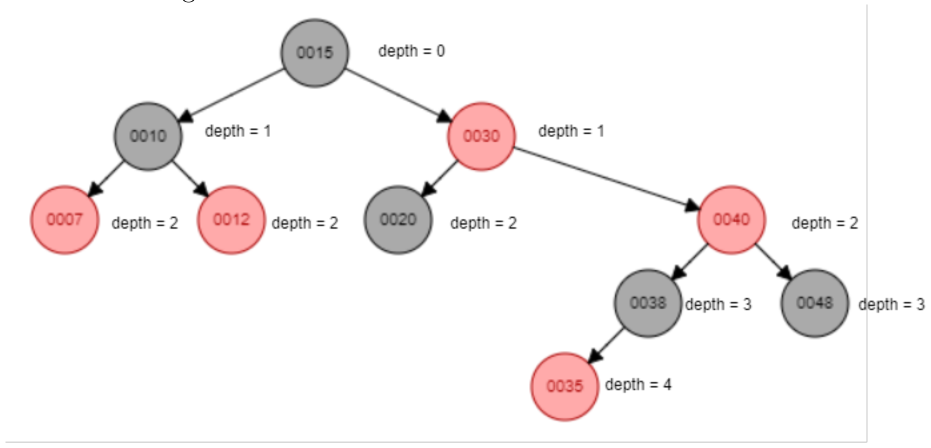
After the insertion of 35:



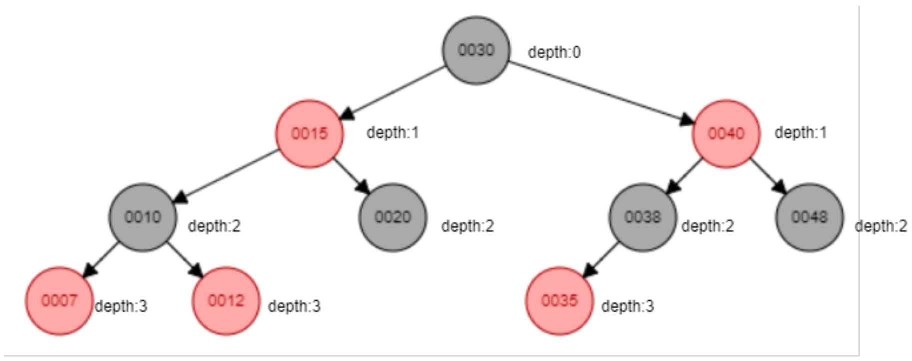
As we can see there is a red-red anomaly exist. We need to examine the correction process as a new subsection.

2.2.3 Establishment of the rules enforced by RBT properties

As in the above tree, there is a red-red conflict exist and it should be fixed with Case 1. After fixing the issue the tree become:



However as we can see the conflict cascaded up to the grandparent node level. Now there is a conflict between 30 and 40. The fact of cascading of Case 1 is encountered in this case. Now, this is the case 3 where grandparent and uncle is black (10 and 15). We need to rotate the root towards left to fix this red-red anomaly as Case 3 suggests. So the insertion of 35 led to the cascading red-red conflicts and root level rotation which we mentioned in beginning. After the left rotation on root level the tree will become:



As we can see, after the Case 3 rotation, whole structure is modified and 30 become parent. Since the parent should be black 30 the color of the 30 changes. Also the color of the 15 changes to establish the property emphasizes all routes have same black height in a path. More crucially, nearly the depth of all nodes are altered.

2.3 Overall analysis

As we discussed, finding the correct place for 35 takes O(height of tree) which is O($logn$) and assigning depth value while insertion is O(1) due to the augmentation.

However, we also need to consider the last modifications in our tree owing to the Case 1 and Case3. In the worst case case 1 led to the cascading color changes up to the root which requires O(height of tree) = O($logn$). Besides, like in case, it is a possibility to encounter rotation in root level which causes the alteration in structure of tree. While rotation, particular amount of pointer places are changed. Thus it will take O(1). Furthermore, just a particular amount of node's color is modified which again costs constant amount of time (O(1)). Up to this point, with the impact of insertion, Case 1 and Case3 modifications costs logarithmic amount of time. Now it's time to consider the depths.

As it can be seen the depth of nearly all of the nodes are changed after the modification of the RBT with the impact of rotation. So, all the nodes should be checked and updated if needed. To do that first root should be checked and updated. After that left and right subtree should be checked. This is the pre-order traversal case (Root - Left - Right). We need to implement a pre-order traverse and while traversing do the modification on embedded depth value. The change of depths in a particular step is O(1) but traversing all nodes will be O(n). Thus at the end, it requires linear complexity to assign the new depth nodes.

(Note: after the rotation, previous depth value of the left sub-tree of 30 (which is 20 in our case) will not be changed. But in any case if there is n nodes in our tree and if we do these operations, the left sub-tree of the right child of root's node count cannot exceed n/2. Thus it cannot affect the complexity even though we create an algorithm to prevent the traverse on this sub-tree. All of these notion is valid for the symmetric cases).

At the end overall analysis brought us 3 value: O(1), O($logn$) and O(n). Linear complexity is dominant amongst these. Thus the complexity of insertion become O(n) in the worst case.

So we can say that the complexity of insertion operation is increased in our augmented data structure (in RBT insertion costs logarithmic amount of time, in our augmented data structure it costs linear amount of time).