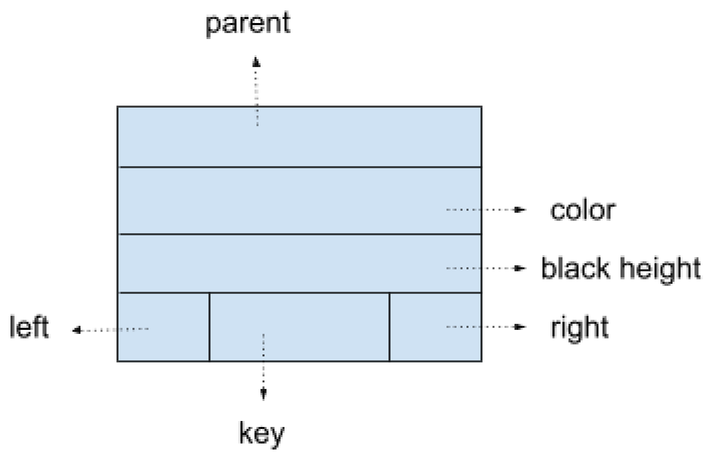
CEREN ARKAÇ
CS 301 - Homework 3
November 2022

Problem 1: To compute the black-height of a given node in a red-black tree (RBT) in constant time, consider augmenting the black-heights of nodes as additional attributes in the nodes of the RBT. Please explain why this augmentation does not increase the asymptotic time complexity of inserting a node into an RBT in the worst case.

Solution:

Step1:
Augment the black height information to the structure of the nodes. Here is what it should look like:



The black height of a node x is determined by its color and the black height of the left child of x and the right child of x.
For example, if the black height of a node x is K, then the black height of x.p is

a) K+1, if x.p is black
b) K, if x.p is red.

This means that a change in bh(x) will affect only the ancestors of x. For instance, updating x.bh might cause updating x.p.bh, and this may cause updating x.p.p.bh and so on. This process may continue until updating the bh of the root. Thus, this update process depends on the height of the tree in the worst case, since the black heights of all the nodes in the path should be updated. In the lecture, we showed that h = O(logn) in a red-black tree. Because the cost for updating an information takes constant time, the worst case to update the black height information of the whole tree is O(logn).

Step2:

Now, I will explain how insertion occurs and it is affected by augmenting bh attribute to the tree. Normally, insertion is followed by 2 phases.

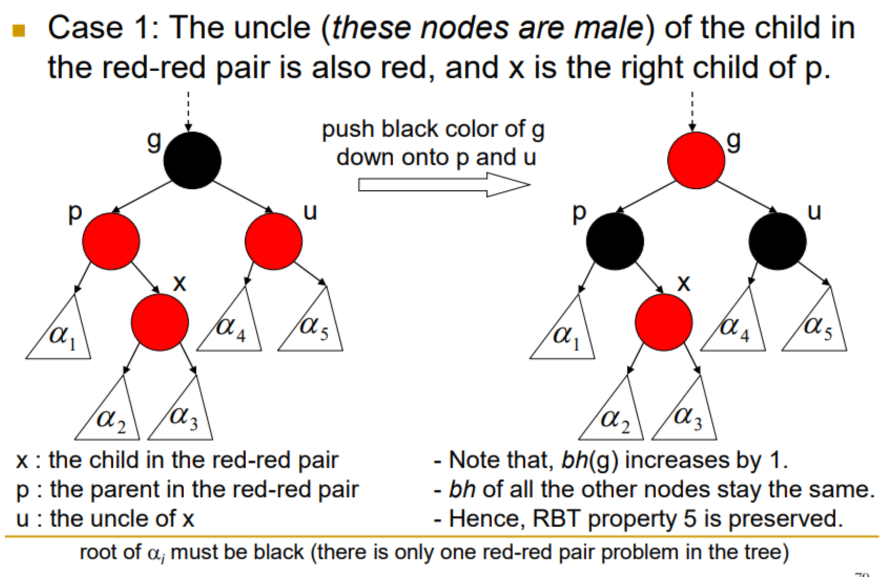
1) Inserting the new element as if the tree is a BST.

   In the worst case, the node x had to be inserted as a leaf node, that is, we may need to traverse the whole RBT from the root to the deepest leaf node. The height of a RBT is O(logn). Thus, insertion will take O(logn), where n is the initial size of the RBT.

2) Setting the color of the new node as red and fixing coloring to comply with the RBT properties, if necessary. (if the inserted node needs to be inserted as a child of a black node, no color fixing is needed. Red-red parent and child makes a problem.)

   There are 3 cases we can encounter while fixing the red-red color problems which we discussed in the lecture.

In case 1, 1) takes O(logn). 2) takes again O(logn).


▪ Case 1: The uncle (*these nodes are male*) of the child in the red-red pair is also red, and x is the right child of p.

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.

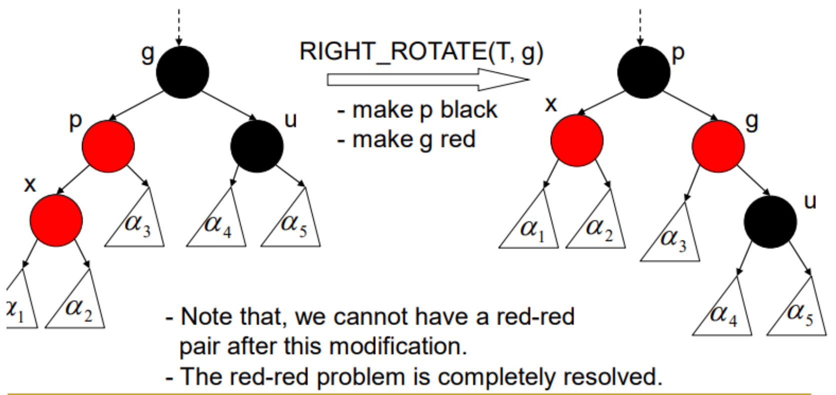
root of $\alpha_i$ must be black (there is only one red-red pair problem in the tree)

Here is the reason: There will be no nodes to be rotated in this case (in contrast to case 2 and 3) and the only works we have to do are updating the black heights and changing the color of the root node, which take constant time. However, we may have to keep applying the black height updates up to the root node. This means, fixing coloring for this case takes O(h) time. That's why fixing coloring may take O(logn). By summing 1) and 2) up, the time complexity for this case is O(logn + logn) = O(logn). Also, the symmetry of case 1 has the same time complexity as itself.

In case 3,

1) takes O(logn). I need to show the tree here to explain the running time complexity for 2) phase.

- Case 3: x is left child of p, p is left child of g, u is black.



- Note that, we cannot have a red-red pair after this modification.
- The red-red problem is completely resolved.

root of $\alpha_1$, $\alpha_2$, and $\alpha_3$ must be black (there is only one red-red pair problem in the tree)

81

Here, there are some rotations. So, we need to check the black heights carefully.

bh(g) =

i) After the insertion: bh(u)

ii) After the rotation: Again bh(u)

bh(p) =

i) After the insertion: bh(x)

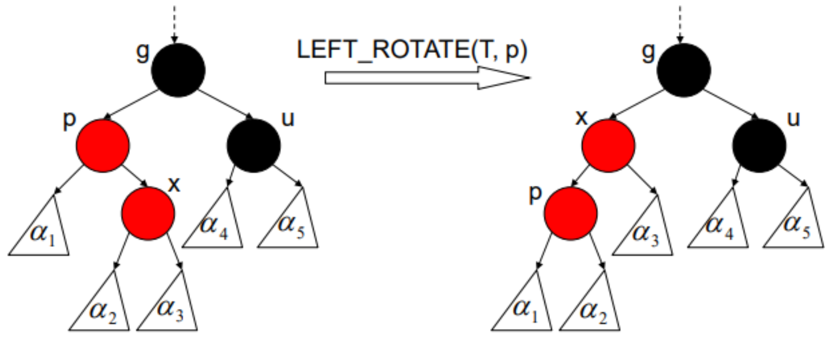
ii) After the rotation: Again bh(x)

Computing will not increase the time complexity. These computations will take constant time. Also, changing the colors will take constant time.

So, we should just consider the propagation of the updates of the black heights up to the root of the tree. Since it is h = O(logn), case 3 takes O(log n + log n) = O(logn) time. Symmetric of case 3 will be the same as this in terms of the time complexity.

In case 2, 1) takes O(logn). I need to show the tree here to explain the running time complexity for 2) phase.

- Case 2: x is right child of p, p is left child of g, uncle 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.

root of $\alpha_1$, $\alpha_2$, and $\alpha_3$ must be black (there is only one red-red pair problem in the tree)

(image is retrieved from the lecture slides prepared by Hüsnü Yenigün)

Here, there are some rotations. So, we need to check the black heights carefully.

bh(g) =

i) After the insertion: bh(u)
ii) After the rotation: Again bh(u)

bh(p) =

i) After the insertion: bh(x)
ii) After the rotation: Again bh(x)

Here, case2 is transformed into case 3, which I explained above. This means the time complexity will be the same after solving case 3 as well.

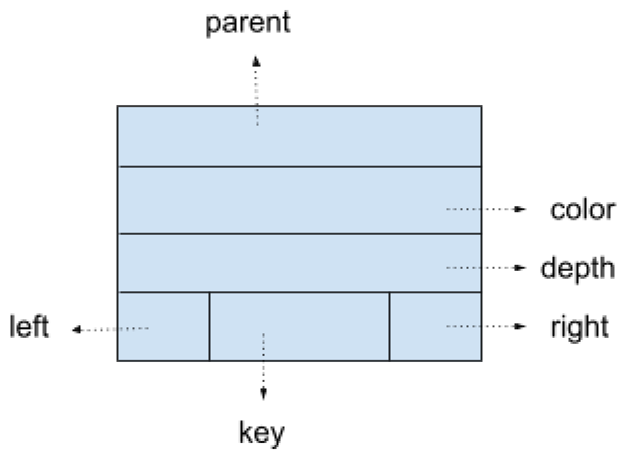
Computing will not increase the time complexity. These computations will take constant time. Also, changing the colors will take constant time.

So, we should just consider the propagation of the updates of the black heights up to the root of the tree. Since it is h = O(logn), case 2 takes O(log n + log n) = O(logn) time. Symmetric of case 2 will be the same as this in terms of the time complexity.

All possibilities take O(logn) time. Even the worst case, case2, is O(logn). Therefore, augmenting the black heights does not increase the asymptotic time complexity of insertion operation.

Problem 2: To compute the depth of a given node in an RBT in constant time, consider augmenting the depths of nodes as additional attributes in the nodes of the RBT. Please explain by an example why this augmentation increases the asymptotic time complexity of inserting a node into an RBT in the worst case.

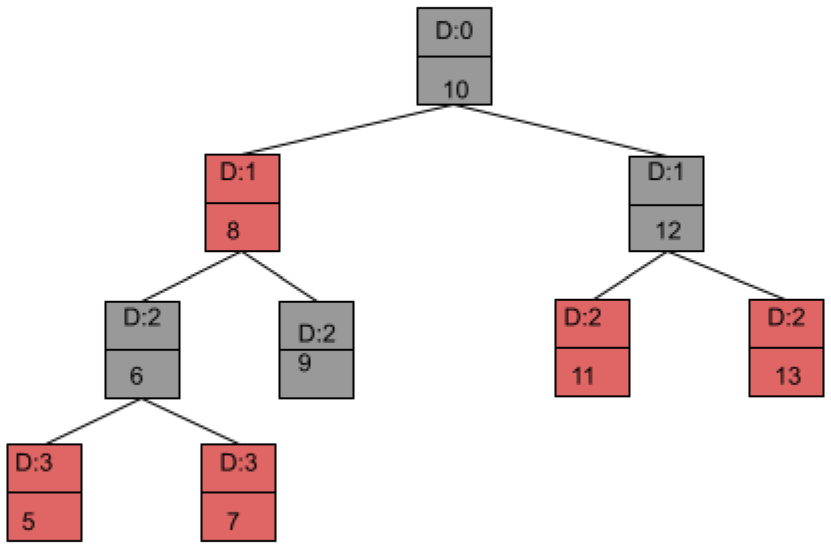
Solution:



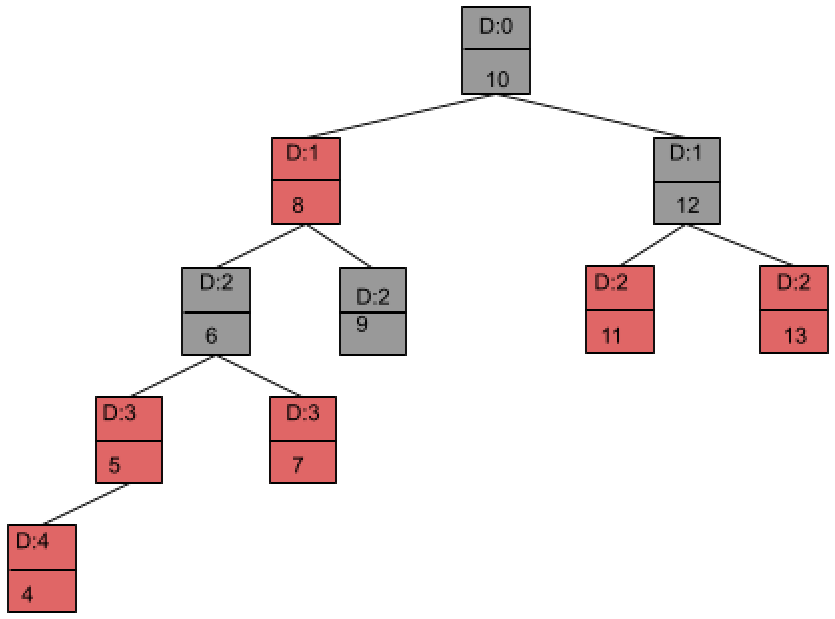
Because the RBT has to comply with the RBT requirements after the insertion, there may be some rotations to be done. Moreover, the depth of the newly inserted node can be computed during finding the place for it to locate. Therefore, the time complexity for insertion will affected by two steps:

1) Inserting the node as if the tree is a binary search tree and computing the depth of node x while inserting. (It is O(logn) + O(logn). I will explain it again.)
2) Fixing coloring after the insertion.

Let’s say this is my RBT:

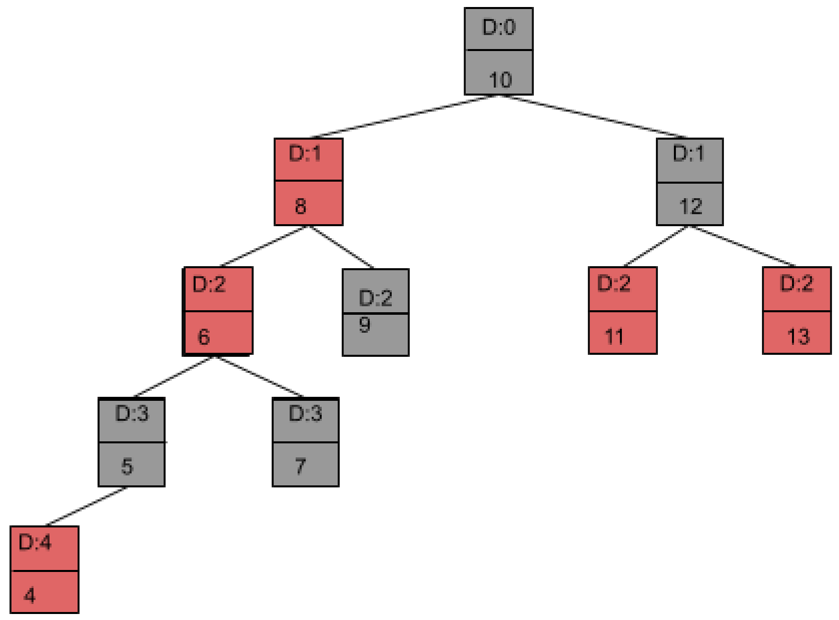


Now, I will insert 4. 4 should be inserted into the left child of 5. While finding the right place for 4, count the number of levels passed and when the right place is reached, set the count as depth. Updating count takes constant time. In the worst case, the node to be inserted should be a leaf node. Then computing the depth of the new inserted node can be done in O(logn).

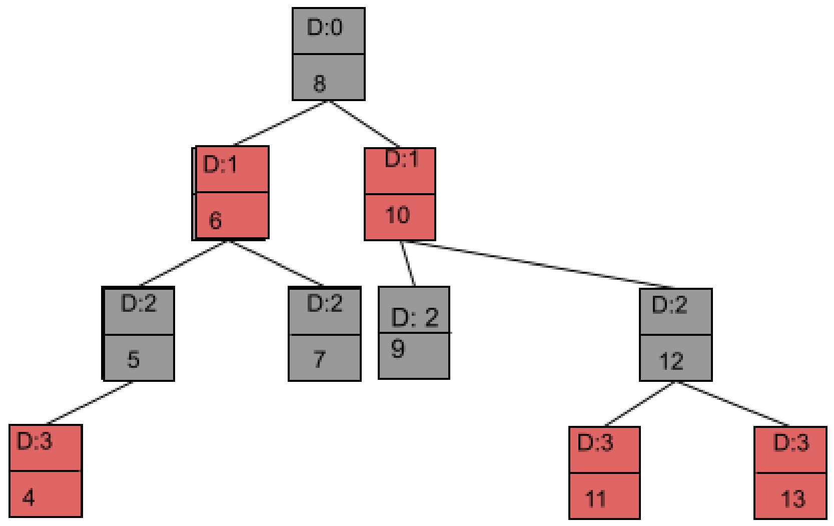


There is a red-red color situation. The coloring should be fixed. The above figure fits into case 1.

Push back color of 6 into 5 and 7.



Now, case 1 situation propagated to the above level. The above figure fits into case 3. Fix the coloring.



Now, everything is OK.

After rotation, The depths of almost all of the nodes have to be changed. Only the depth of the node with key 9 remained the same. As a result, 9 out of 10 nodes have to be updated.
To do this, after the insertion, almost all of the nodes have to be traversed one by one. There are many nodes in many levels to be updated. So, this cannot be bounded by the height of the tree, which is h = O(logn). This updating (traversing) process may even be close to Theta(n). I

cannot compute it easily. But, it is certain that insertion to the tree after augmentation of depth takes bigger than O(logn) time, when n is the input size. And it dominates O(logn) and, which is coming from the first phase of the insertion (inserting the new node as if the tree is a BST and computing the depth of the newly inserted node.