Student Name: Mert Ekici
Student ID: 26772

Algorithms (CS301)
Assignment 3

Sabancı Üniversitesi

Problem 1 (Augmented RBT - Black Height)

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.

black-height of x: the number of black nodes along a path from a node x to a leaf under in the subtree rooted at x (denoted by bh(x)).

Purpose

To compute the black-height of a given node in a red-black tree (RBT) in a constant time.

Augmentation

Augment black-height of nodes as additional attributes within an extra field in the node.

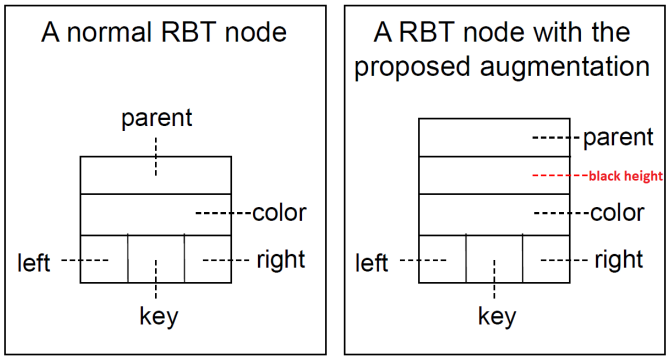

Figure 1: Black-Height Augmentation - RBT

Insert Operation

Since insert operation modifies the tree, we need to guarantee correct assignment to the black-height field after insertion is executed on the Black-Height Augmented RBT.

Additionally, assignment operation shouldn't increase the worst case running time of the original insert operation for the desired scenario.

- An insert operation will place one more element into the tree and it is colored red by default.
- Hence, there will not be an update until the fixColoring operation (red node does not increase black-height of any node).
- However, if there is a red-red coloring problem, fixColoring will be called. Note that, modifications done by fixColoring will modify black-heights of some of the nodes. Therefore, fixColoring should be analyzed case by case to identify the worst case asymptotic time complexity of inserting a node.

```
AugmentedRBT_Insert (T, x) { // T: the tree, x: a new node to be inserted
    color[x] = RED; // the new node will be red
    AugmentedBST_Insert(T, x); // insert as if we were using an ordinary BST
    AugmentedFixColoring(T, x); // fix the "red red" coloring problem if exists}
```

```
AugmentedBST_Insert (T, x) { // T: the tree, x: a new node to be inserted
    BST_Insert(T,x); // insert as if it is normal BST
    // No black-height update since the inserted node is red
}
```

inserted node x's black height = bh(x) = 1

Due to the RBT Property 3 and 4, we assign two null black children nodes to the inserted red node.

- RBT Property 1: Every node is red or black
- RBT Property 2: The root is black.
- RBT Property 3: Every leaf node is black.
- RBT Property 4: A red node has two black children.
- RBT Property 5: Every path from the root to a leaf has the same number of black nodes.

Augmented Fix Coloring

Case I

The uncle node of the child in the red-red pair is also red

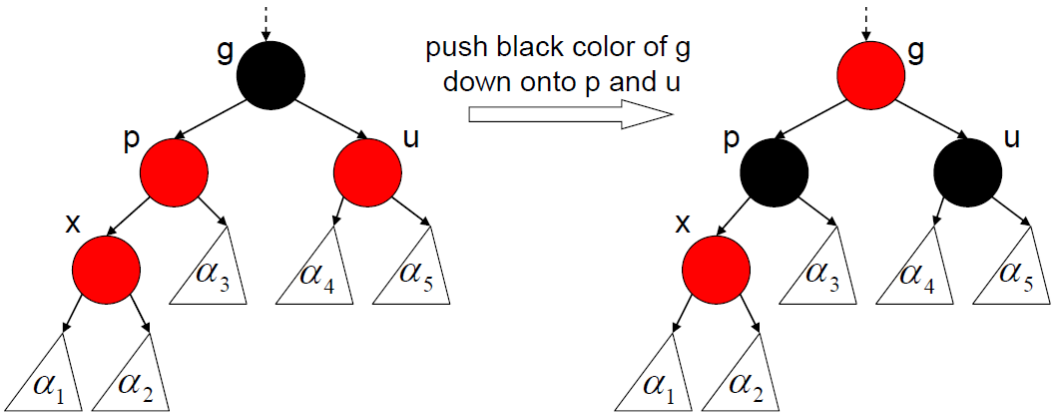

Figure 2: FixColoring - Case I

x: the child in the red-red pair
p: the parent in the red-red pair
u: the uncle of x
bh: black-height

- bh(g) increases by 1 and bh of all the other nodes stay the same.
- RBT property 5 is preserved.
- If the parent of g is also a red node, then we still have red-red pairs problem. We continue to resolve the problem until the root (if problem exists).
- These findings are also valid for the symmetric case.
- Thus, the worst case complexity of Case I is $O(lgn) = O(h)$. Repeated Case I until the root node.

Case III

x is left child of p, p is left child of g, u is black (or symmetric)

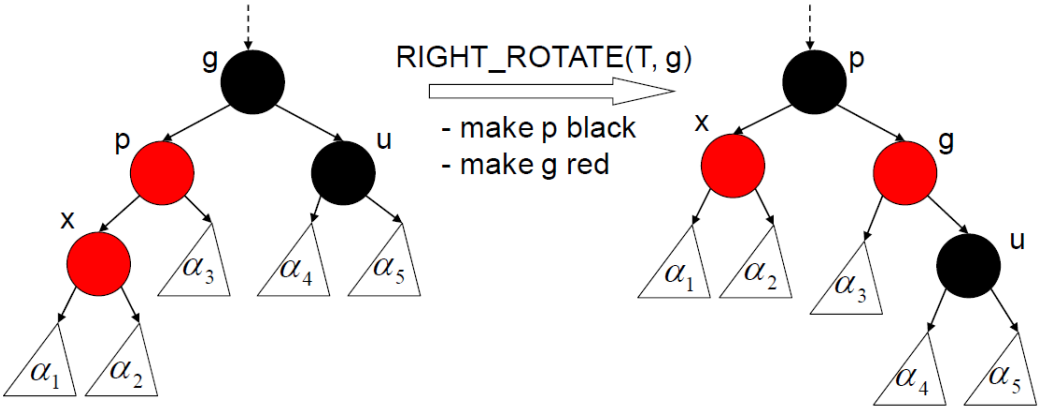

Figure 3: FixColoring - Case III

- bh(x) is still the same since it's children nodes are preserved.
- bh(p) is equal to bh(x) and it will not change as well. (RBT property 5)
- bh(u) still the same since it's children nodes are preserved.
- bh(g) is equal to bh(u)+1 and it will not change as well. (RBT property 5)
- We cannot have a red-red pair after this modification.
- The red-red problem is completely resolved.
- These findings are also valid for the symmetric case.
- No change in any of the black-heights. Since red-red problem is totally solved, no black-height modification is needed for this case at all.

Case II

x is right child of p, p is left child of g, uncle is black (or symmetric)

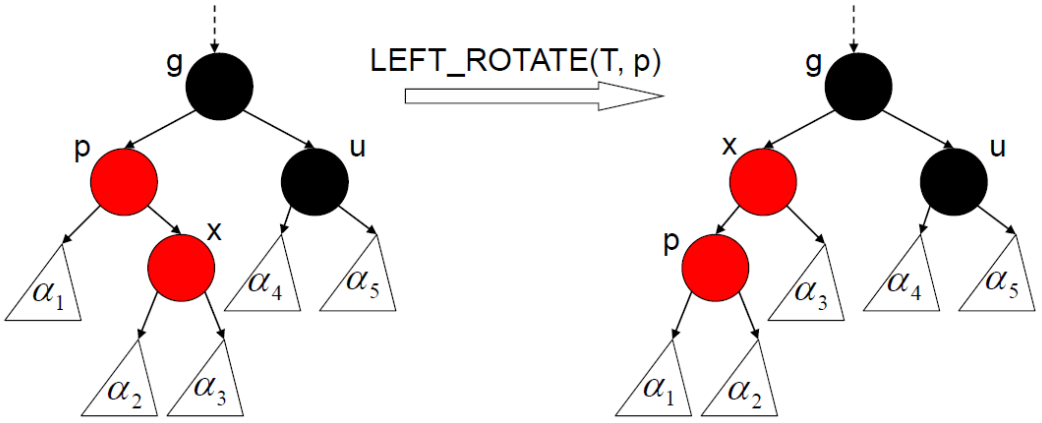

Figure 4: FixColoring - Case II First Rotation

- bh(x) and bh(p) remain the same since they both are red and they keep one of their children from the state before rotation. (RBT property 5)
- Therefore, bh(g) will not change either (bh(u) remains unchanged). (RBT property 5)
- bh(u) and black-heights of the roots for subtrees α_i ($\forall i \in [1, 5]$) stay the same since no modification is applied to them.
- It seems that this rotation did not solve anything, Augmented RBT still has the red-red pair problem.
- But now, the problem is transformed into Case III which is previously analyzed.
- Case III solution does not change any of the black-height values either.
- Hence, two rotations are done for Case II and no black height is changed (it also holds for Case III).
- Red-red problem goes away after Case II.
- These findings are also valid for the symmetric case.

Running Time

There is no additional assignment to correctly maintain new field during the insert operation until AugmentedFixColoring part (everything is the same with the regular RBT insert operation until fixColoring).

If there is a red-red pair problem;

1. If Case I applies, we should update the black-heights of constant number of nodes. Only one assignment happens for each iteration of Case I where black-height of node **g** is increased by 1. In the worst-case scenario, it starts from the deepest leaf node and goes until the root which requires at most $O(lgn)$ time (traversing a path from the root to a leaf).
2. If Case III applies, there is no change in the black-heights of the affected nodes after the rotation hence no assignment. Red-red pair problem is completely solved.
3. If Case II applies, no black-height assignment is needed for the first rotation. Then, it is transformed into Case III and no black-height change happens (no assignment) also in the second rotation. Red-red problem is completely solved.

Worst-Case Running Time of Black-Height Augmented RBT Insert Operation = $O(lgn) = O(h)$

Discussion

Additional info can be maintained efficiently during the insertion. Black-Height RBT is an efficient augmentation strategy to obtain black-height of a given node in a constant time (but only in terms of insertion operation, delete operation is also need to be checked).

Problem 2 (Augmented RBT - Depth)

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.

depth of x: the number of edges present in path from the root node to x (denoted by d(x)).
(the length of the path from the root to the node)

Purpose

To compute the depth of a given node in a red-black tree (RBT) in a constant time.

Augmentation

Augment depth of nodes as additional attributes within an extra field in the node.

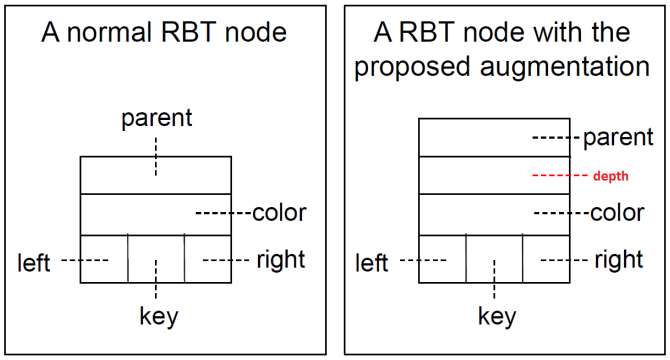

Figure 5: Depth Augmentation - RBT

Insert Operation

Since insert operation modifies the tree, we need to guarantee correct assignment to the depth field after insertion is executed on the Depth Augmented RBT.

Additionally, assignment operation shouldn't increase the worst case running time of the original insert operation for the desired scenario but it will increase for this case (depth augmentation) which is explained in the following steps.

- An insert operation will place one more element into the tree as a leaf node and it is colored red by default.
- There will not be a depth update until fixColoring since the new node is placed in a leaf (and by default two null black leaves are assigned as its children) and the depth of the inserted node becomes the depth of its parent + 1.
- However, if there is a red-red coloring problem, fixColoring will be called. Note that, modifications done by fixColoring will modify depth values of some of the nodes. Therefore, fixColoring should be analyzed case by case to identify the worst case asymptotic time complexity of inserting a node.

```
AugmentedRBT_Insert (T, x) { // T: the tree, x: a new node to be inserted
    color[x] = RED; // the new node will be red
    AugmentedBST_Insert(T, x); // insert as if we were using an ordinary BST
    AugmentedFixColoring(T, x); // fix the "red red" coloring problem if exists}
```

```
AugmentedBST_Insert (T, x) { // T: the tree, x: a new node to be inserted
    BST_Insert(T,x); // insert as if it is normal BST
    // No depth update since the inserted node does not affect any node's depth.
}
```

inserted node x's depth = d(x) = d(parent[x])+1
Due to the RBT Property 3 and 4, we assign two null black children nodes to the inserted red node.

Augmented Fix Coloring

Case I

The uncle node of the child in the red-red pair is also red

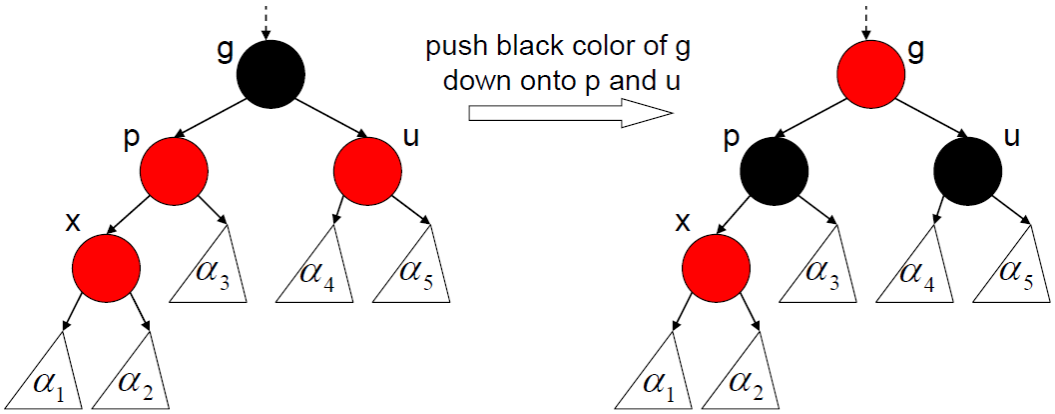

Figure 6: FixColoring - Case I

x: the child in the red-red pair
p: the parent in the red-red pair
u: the uncle of x
bh: black-height

- Only the color of the nodes are modified.
- The nodes still have the same depth fields.
- No assignment is needed for the depth fields..
- These findings are also valid for the symmetric case.
- If the parent node of g is a red node as well, then the problem will be solved again by climbing upwards to the root until its completely fixed. We need to analyze Case II and III.

Case III

x is left child of p, p is left child of g, u is black (or symmetric)

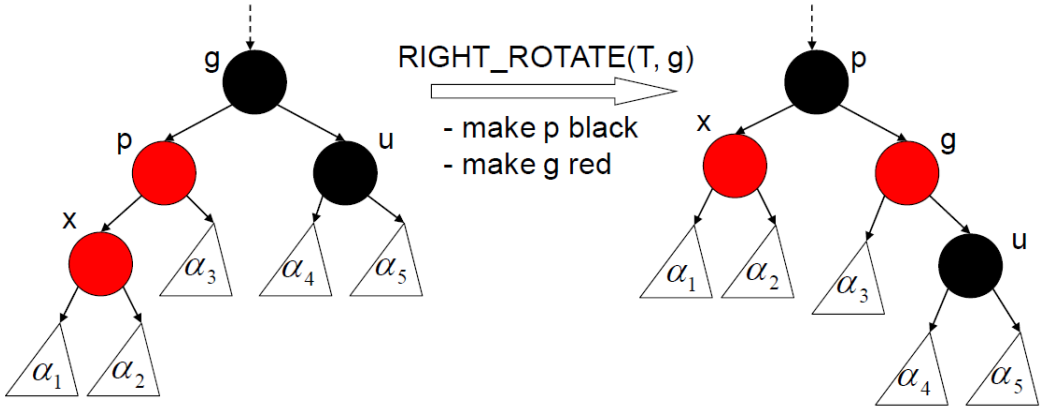

Figure 7: FixColoring - Case III

- The subtree rooted at x is moved one level up.
  Depth fields of all nodes in this subtree is increased by 1. $\forall i \in subtree(x)$, $d(i) = d(i) + 1$
- d(p) is increased by 1 since it is rotated to upper level.
- The subtree rooted at u is moved one level down.
  Depth fields of all nodes in this subtree is decreased by 1. $\forall j \in subtree(u)$, $d(j) = d(j) - 1$
- Depth values of the nodes in the subtree α_3 stay the same since the subtree preserved its level.
- We cannot have a red-red pair after this modification.
- The red-red problem is completely resolved.
- These findings are also valid for the symmetric case.

Case II

x is right child of p, p is left child of g, uncle is black (or symmetric)

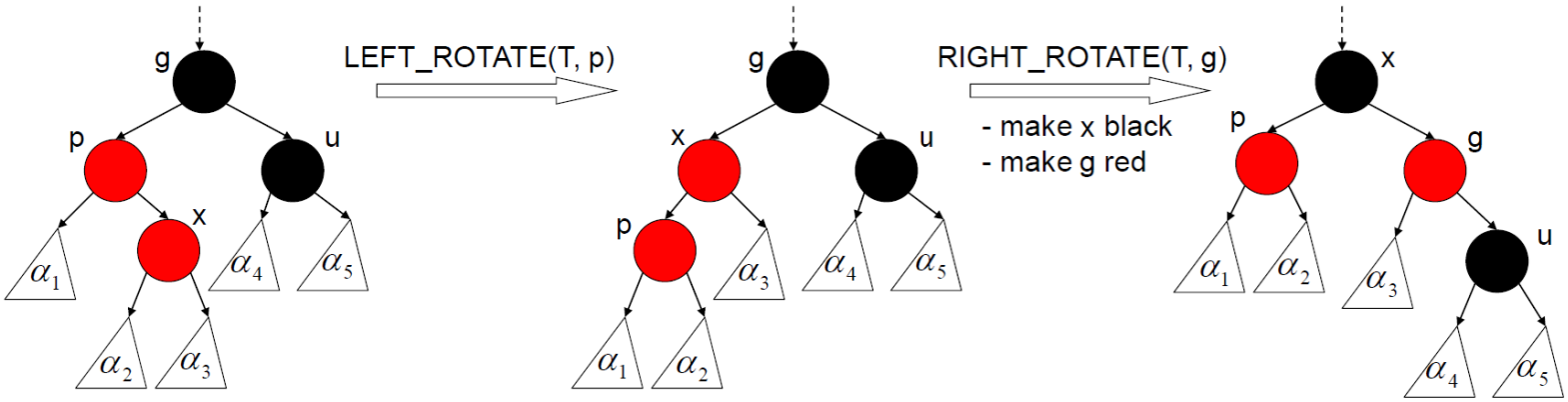

Figure 8: FixColoring - Case II

- x moved two levels up. Hence, d(x) increased by 2.
- p and its left subtree α_1 did not move after two rotations. Their depths remain the same.
- Depth values of the nodes in the subtrees α_2 and α_3 increased by 1.
- g and its right subtree rooted at u moved one level down. Depth fields of g and the nodes of g's right subtree (u and the nodes of subtrees α_4, α_5) decreased by 1.
- Red-red problem goes away after Case II.
- These findings are also valid for the symmetric case.

Both in Case II and III, many assignment operations are executed to maintain correct depth fields for the insertion. Since depth is a measure depending on the distance to the root node, rotations affect all of the nodes' depth values in the subtree rooted at the rotated nodes. Number of additional assignments increases exponentially with the height of the related subtree (linear time with the number of elements in the subtree - all elements in the subtree are traversed $O(n)$).

Let's examine this operational burden under left and right rotations.
Case 2 and Case 3 (and their symmetric cases) will perform either left or right rotations.

Right Rotation

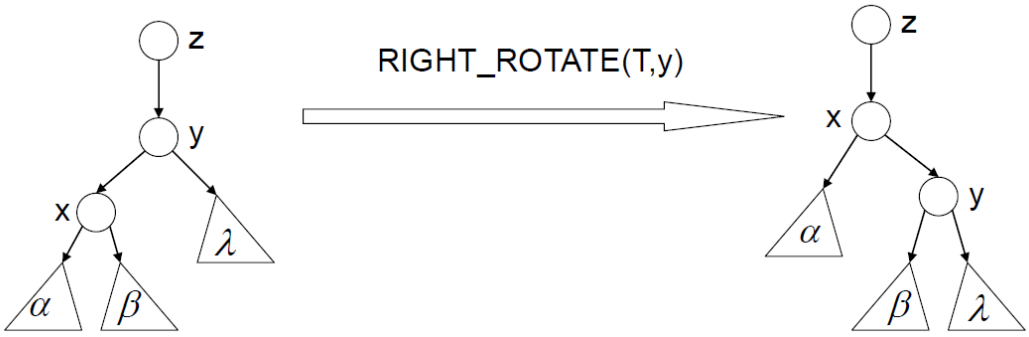

Figure 9: Right Rotation

- x and the nodes of its left subtree α are now one step closer to the root node z.
  $d(x) = d(x) + 1$ and $d(i) = d(i) + 1\ \forall i \in \alpha$
- y and the nodes of its right subtree λ are now one step further to the root node z.
  $d(y) = d(y) - 1$ and $d(j) = d(j) - 1\ \forall j \in \lambda$
- Nodes in the subtree β preserved their depth values.
- Let's assume z as the root node of a tree (with n elements) which becomes perfectly balanced after the rotation. (RBT is a type of self balancing tree itself)
- Then, subtree α has n/4 elements whereas subtree β and subtree λ have n/8 elements.
  $size(\alpha) = n/4 \quad size(\beta) = n/8 \quad size(\lambda) = n/8$
- Number of Assignments $= n/4 + n/8 + 2 = 3n/8 + 2 = O(n)$
  (+2 is for x and y)

Left Rotation

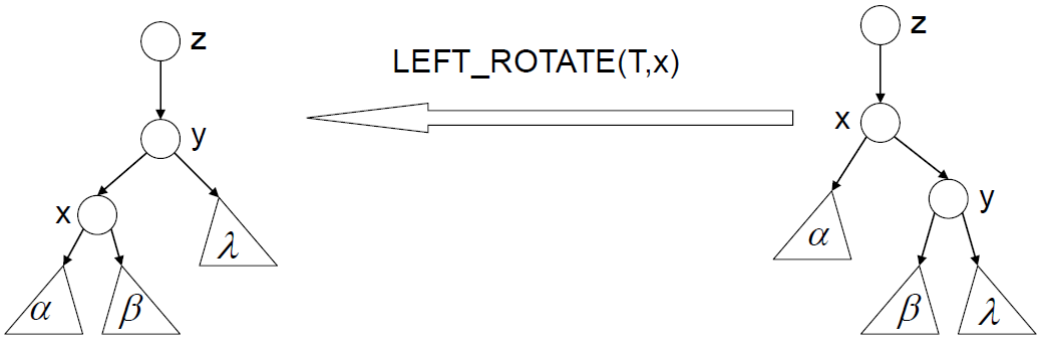

Figure 10: Left Rotation

- x and the nodes of its left subtree α are now one step further to the root node z.
  $d(x) = d(x) - 1$ and $d(i) = d(i) - 1\ \forall i \in \alpha$
- y and the nodes of its right subtree λ are now one step closer to the root node z.
  $d(y) = d(y) + 1$ and $d(j) = d(j) + 1\ \forall j \in \lambda$
- Nodes in the subtree β preserved their depth values.
- Let's assume z as the root node of a tree (with n elements) which becomes perfectly balanced after the rotation. (RBT is a type of self balancing tree itself)
- Then, subtree λ has n/4 elements whereas subtree α and subtree β have n/8 elements.
  $size(\lambda) = n/4 \quad size(\alpha) = n/8 \quad size(\beta) = n/8$
- Number of Assignments $= n/4 + n/8 + 2 = 3n/8 + 2 = O(n)$
  (+2 is for x and y)

Example

In the following example, all three cases (Case I, Case II and Case III) are observed within the insertion of a single node to a RBT.

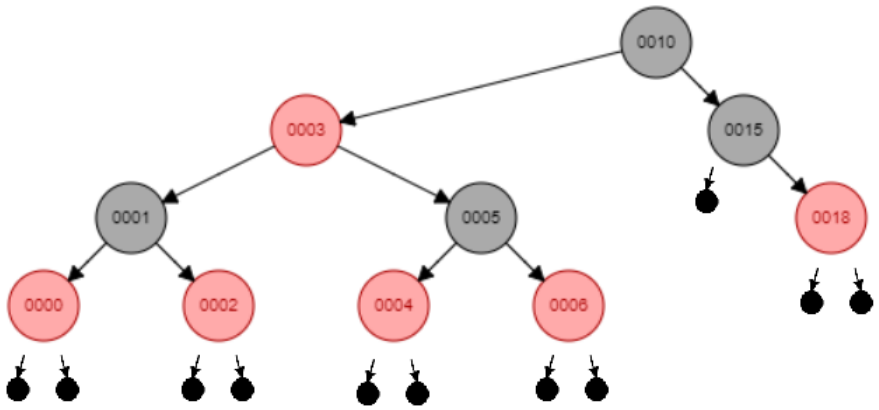

Figure 11: RBT before the insertion

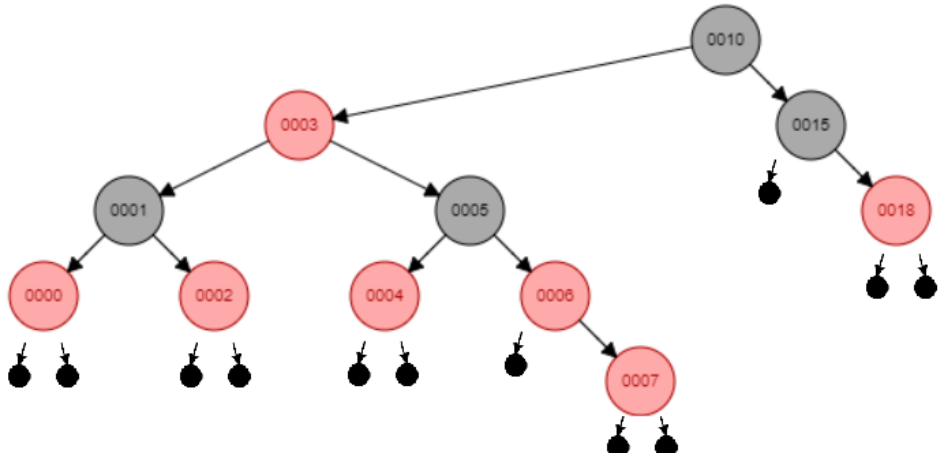

Figure 12: 7 is inserted like a normal BST - fixColoring is needed (Case I)

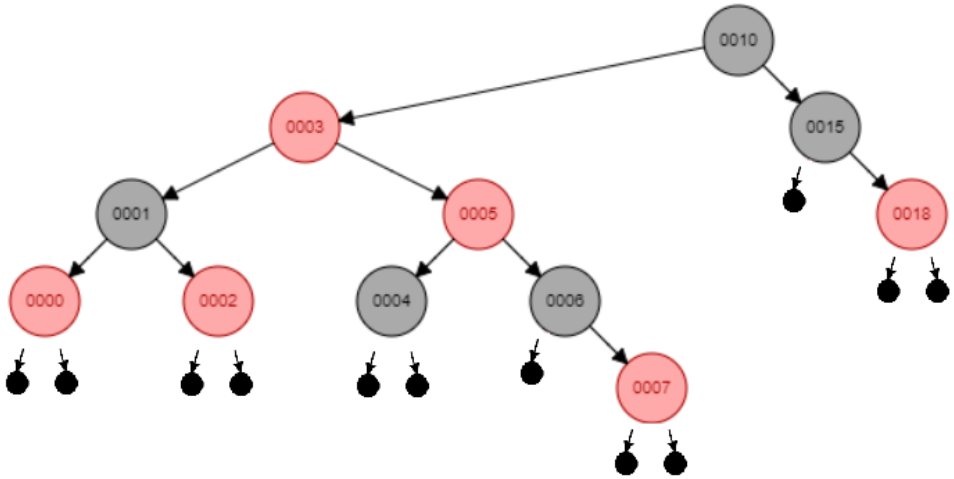

Figure 13: Case I solution is applied - Case II emerges

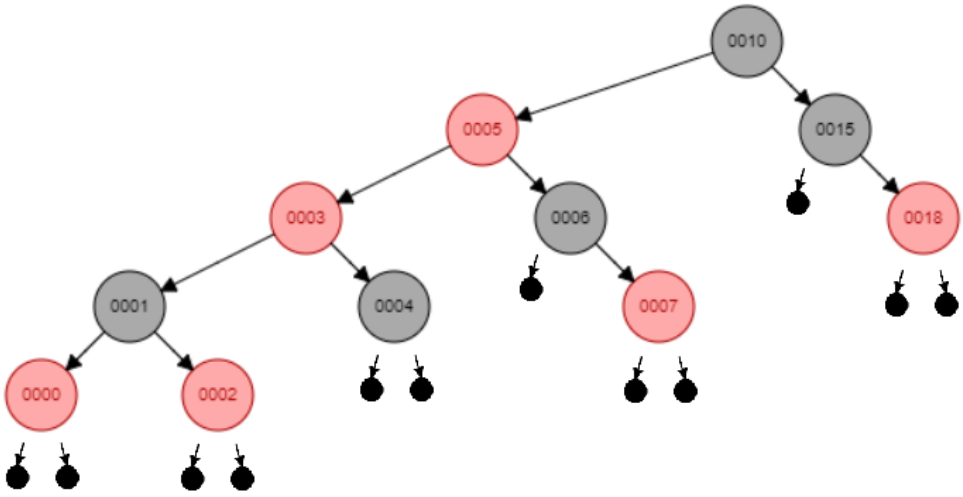

Figure 14: Left rotation - Case II is transformed into Case III

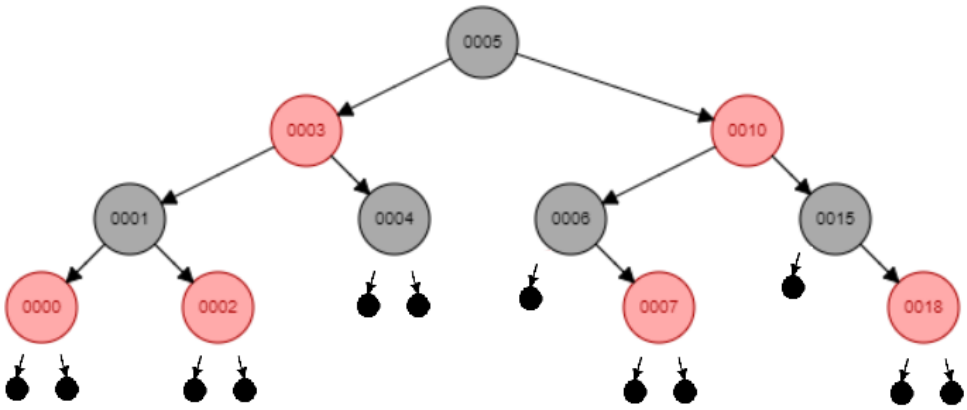

Figure 15: Case III solution is applied - RBT after the insertion

Findings

- New node with key 7 is inserted as if were using an ordinary BST. No depth update needed.
- fixColoring is called.
- Case I is detected. Black color of the node with key 6 is pushed down.
- Case II is detected. Left rotation to the node with key 5.
- Case III is detected. Left rotation to the node with key 3.
- As can be seen in Figure 14-15, even the root node with key 10 is included in the rotation. Thus, the depth values of most of the nodes are changed.
- Node 3 and its left subtree remained unchanged.
- Rest of the nodes are assigned with new depth values which is more than half of the elements.

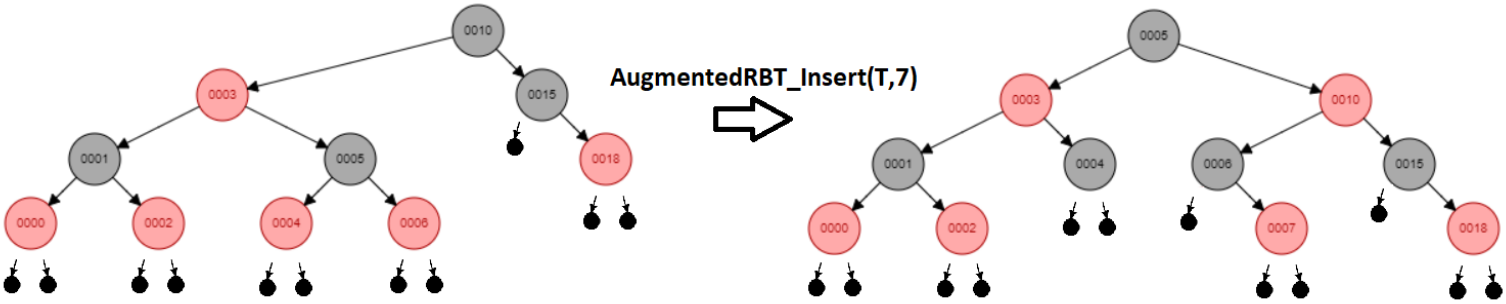

Figure 16: Before & After

Running Time

There is no additional assignment to correctly maintain new field during the insert operation until AugmentedFixColoring part (everything is the same with the regular RBT insert operation until fixColoring).

If there is a red-red pair problem;

1. If Case I applies, there is no need for any depth update unless the red-red problem is not resolved yet. In the worst-case scenario (which is previously illustrated in the example part), Case I might be used repetitively or one of Case I and Case II can happen single time.
2. If Case III applies, there is a single left or right rotation on the related node (the problem is completely resolved). Number of necessary assignments (depth updates) is linear with the size of the tree. $O(n)$
3. If Case II applies, two consecutive rotations happen (with the first rotation Case II is transformed into Case III and red-red coloring problem can be solved in the second rotation). Due to these rotations, number of additional assignments is not constant, in fact it is size dependent $O(n)$

In the worst-case, red-red problem climbs up via multiple Case I scenarios and finally left or right rotation that includes the root node happens ($O(n)$ assignments) within Case II or Case III.

Worst-Case Running Time of Depth Augmented RBT Insert Operation = $O(n)$

Discussion

Additional info cannot be maintained efficiently during the insertion. Depth RBT is not an efficient augmentation strategy to obtain the depth of a given node in a constant time.