e-PGPathshala

Subject: Computer Science

Paper: Data Structures

Module: Red Black Trees - I

Module No: CS/DS/25

Quadrant 1- e-text

Welcome to the e-PG Pathshala Lecture Series on Data Structures. We have understood the basic concept of balanced binary search trees and discussed one such tree – the AVL tree. In this module we will discuss another balanced binary search tree – the Red Black tree.

Learning Objectives

The learning objectives of the module are as follows:

- To understand the concept of Red Black Trees
- To discuss the properties of Red Black Trees
- To describe rotation of Red Black Trees
- To explain the Insertion into Red Black Trees

25.1 Introduction to Red Black Trees

Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***. In this case as in balanced trees the height is of the $O(\log n)$, where n is the number of nodes. Hence the operations associated with Red-black trees take $O(\log n)$ time in the worst case. However in the case of Red-black trees each node requires an extra bit when compared to binary search trees since each node is now associated with a bit indicating the color attribute which can be either **red** or **black**. The nodes of Red-black trees inherit all the other attributes associated with a node of the binary search tree such as key, and pointers to the left sub-tree, right sub-tree and the parent. Now in the case of Red-black trees all empty trees (leaves) are colored black. This is normally carried out by setting the single sentinel associated with color as *nil*, for all the leaves of red-black tree T, with $color[nil]$ = black. Here we assume that the leaf nodes null nodes are set as nil. The parent of the root is also set to nil.

25.2 Red-black Properties

As we have already discussed the Red-black tree data structure requires an extra one-bit color field in each node. Every node is either red or black. Some of the properties associated with the Red-black tree are:

1. Every node in a Red-black tree is either red or black

2. The root is **black** and every leaf (*nil*) is **black.** We assume that every node that is not a leaf has 2 children (one or both of which may be nil nodes)
3. If a node is **red**, then its parent is **black**. In other words, if a node is **red**, then both of its children are **black.** This means that Red-black trees do not allow 2 consecutive red nodes on a path
4. All simple paths from any node *x* to a descendant leaf have the same number of black nodes = black-height(*x*).

25.2.1 Red-black Tree – Example

Now let us understand the properties of Red-black trees using an example (Figure 25.1). For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs. NIL[T] has the same fields as an ordinary node. As we have discussed all leaf nodes are colored black.

Color[NIL[T]] = BLACK

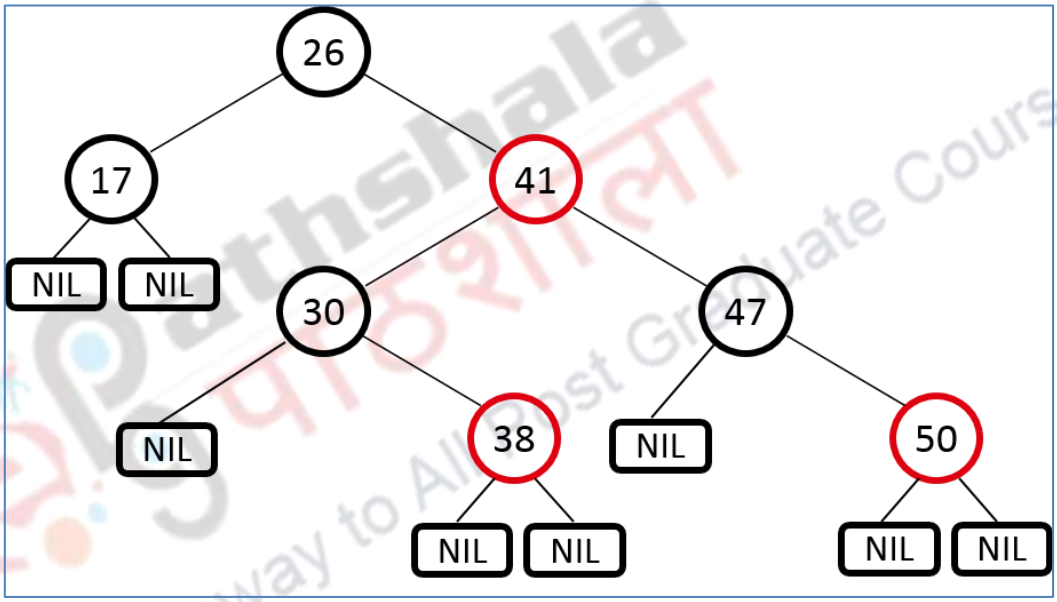


Figure 25.1 Example of a Red-black Tree

Figure 25.2 shows the properties for a given Red-black tree. As you can see all nodes of the tree are colored as either red or black (property 1). The root node (**10**) and all leaf nodes are colored black (property 2). All the red nodes **1, 5, 3, 25 and 60** have black parents (property 3). In addition as per property 4 all these red nodes have both

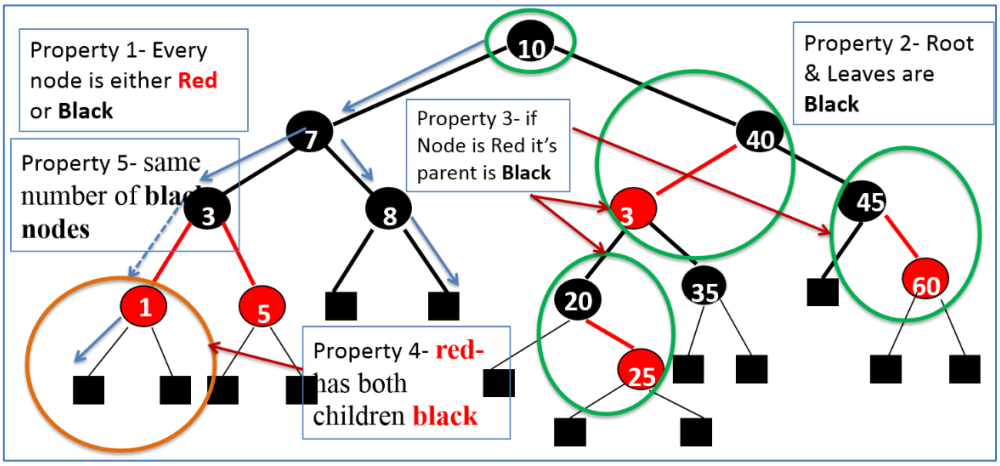


Figure 25.2 Example of the Red-black Tree showing the Properties

children black and these red nodes cannot occur consequently on a path in the Red-black tree. Moreover the number of black nodes on a simple path from any black node to a descendent leaf node is the same (property 5). Example the number of black nodes from **10** to leaf node through any path **(10-7-3-1-Nil, 10-7-8-Nil, 10-7-3-5-Nil)** is 3 where the count does not include the node itself.

25.2.3 Black-Height of a Node

Based on property 5, we can also define the black-height of a node. In general the height of a nodeis defined as the number of edges in the longest path to a leaf.

Black-height of a node x: bh(x) is the number of black nodes (including NIL) on the path from x to a leaf, without counting node x. In the example shown in Figure 25.3 the red node 50 has both height and black-height as 1 but 47 has height 2 and black height as 1 since there is only one black node (Nil) in the path from **47 (47-50-Nil)** to the leaf excluding **47.**

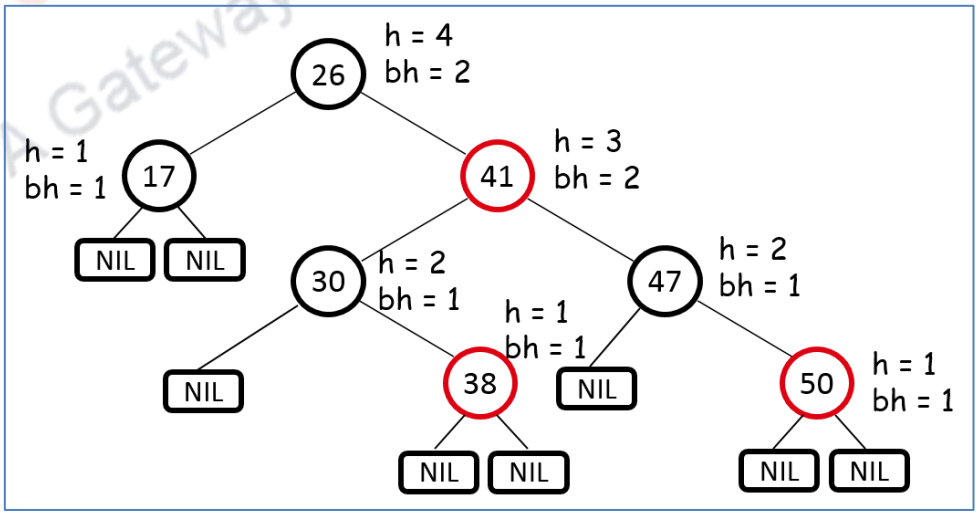


Figure 25.3 Height and Black-height of Nodes

25.3 Implementing Red-black Operations

As in any data structure we can classify the operations associated with it as access operations and update operations. Access operations include searching for an element, finding the minimum or maximum element in the red-black tree, finding the predecessor, successor etc. Update operations include inserting elements into and deleting elements from the red-black tree. Remember that a red-black tree is basically a binary search tree (BST) and we can use all the BST algorithms without change by simply ignoring the colors associated with each node. The worst-case time complexity depends on h, the height of the tree O(h) = O(log n). Now after performing the BST operations, we have to ensure that the modified tree will still obey the red-black tree properties. In order to maintain the red-black tree properties we need to carry out some rotations of the nodes of the tree.

25.4 **Rotations** and **Red-Black-Trees**

Rotations are the basic tree-restructuring operation for almost all *balanced* search trees for the purpose of rebalancing a balanced tree. These operations are necessary for re-structuring the red-black tree after insert and delete operations on red-black trees. The local operation in a search tree should ensure the binary search tree property.

Rotations take a red-black-tree and a node within the tree and together with some node re-coloring they help restore the red-black-tree property. These rotations change some of the connections between nodes and also ensure that the binary-search tree property is maintained. Rotation takes a red-black-tree and a node, appropriately changes pointers to change the local structure, and ensures that the binary-search-tree property is not violated. There are two types of rotations left rotation and right rotation which are inverse of each other. As is shown in Figure 25.4 during Left-Rotate(T,x), x is rotated to the left and now becomes the left sub-tree of y which now becomes the root. The left sub-tree β of y becomes the new right sub-tree of x. Now the inverse of the left rotate is right rotate where Right-Rotate (T,y) right rotates about the node y. Rotate operations are used for restructuring the tree after insert and delete operations on red-black trees. These rotations are general rotations discussed in earlier modules.

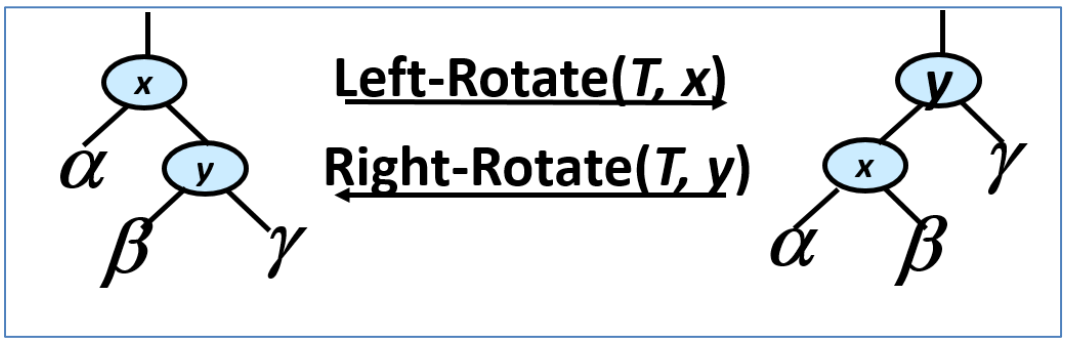


Figure 25.4 Left and Right Rotate

25.4.1 Left Rotations

When performing left rotation on a node x we assume that the right child of x (y) is not NIL. The basic idea is to pivot around the link from x to y where y is the root of the right sub-tree of the node x. We make y the new root of the sub-tree and x becomes y's left child while y's left child becomes x's right child.

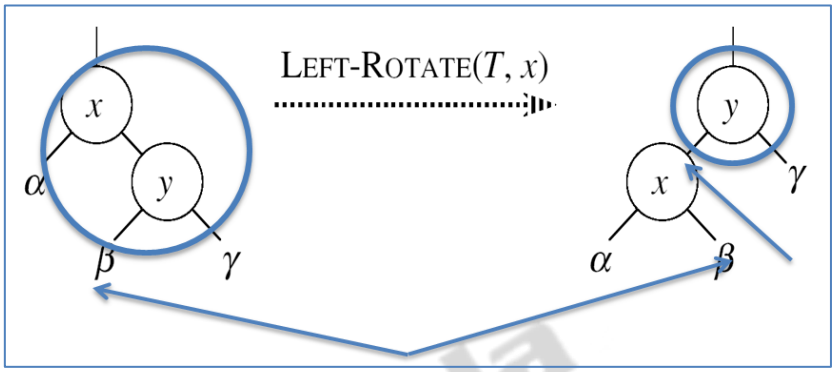


Figure 25.6 Left Rotation

25.4.1.1 Example of Left Rotation

Figure 25.6 shows an example of left rotation of 12 with 9 as the pivot. Now 12 is rotated left and now becomes the left sub-tree of 9's parent 7. The left sub-tree of 12 rooted at 11 now becomes the right sub-tree of 9.

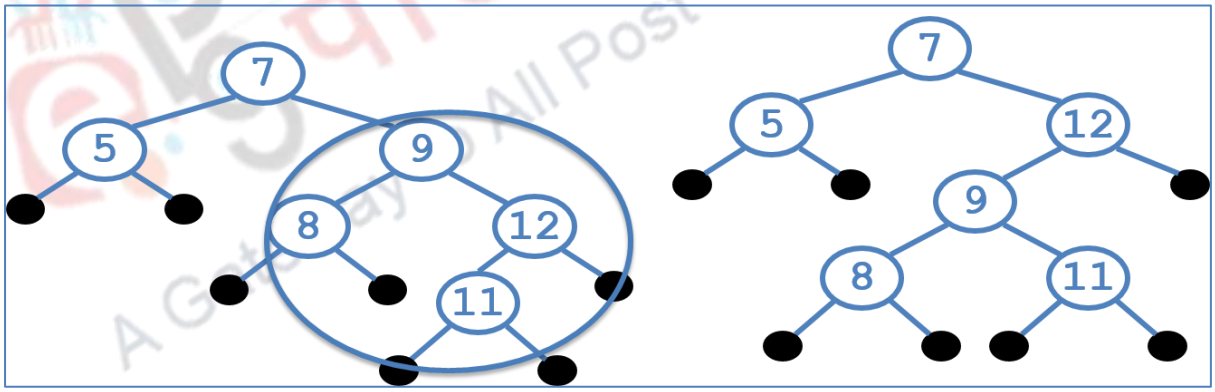


Figure 25.6 Example for Left Rotation

25.4.2 Right Rotation

In general, rotation of a node for re-balancing a balanced tree involves pointer manipulation. In the case of right rotation of node x about y where x is left sub-tree of y (Figure 25.5). We assume that for a right rotation on a node x, the left child of x of y is not NIL. When we pivot around the link from y to x, we make x the new root of the sub-tree and y becomes x's right child and x's right child becomes y's left child.

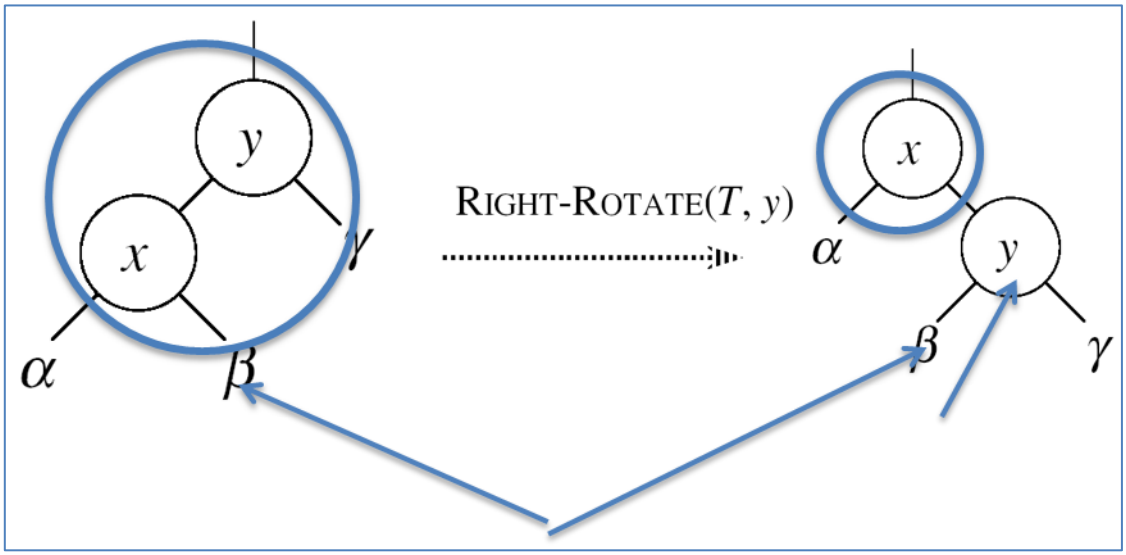


Figure 25.7 Right Rotation

25.5 Modifying Operations

As is generally the case the operations INSERT and DELETE cause modifications to the red-black tree. After we carry out the operation, we need to change color and restructure the links of the tree via ***"rotations"*** we have discussed in the previous section.

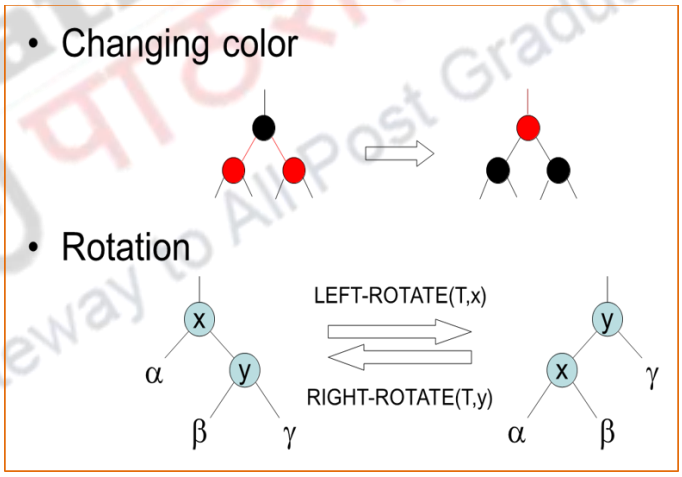


Figure 25.8 Changing of Color and Rotation

25.6 Insertion – Red-Black Tree

We need to insert a new node z into a red-black-tree. The first step is the insertion of the node z into the tree as is done for an ordinary binary search tree. We color the new node as red and then go on to restore the red-black-tree properties. Now let us consider how to color the new node. Let us consider the example given in Figure 25.9. Let us assume that we have inserted 35 and colored it red . We see that property 4 is violated. As we know property 4 states that If a node is **red**, then both of its children should be **black** and Red-black trees do not allow 2 consecutive red nodes on a path – 38 and 35 two consecutive red nodes. Let us assume that we have inserted 14 and colored it black, in this case property 5 which states that all

paths from a node to its leaves contain the same number of black nodes is violated. Hence we need to carry out modifications to restore the red-black tree property.

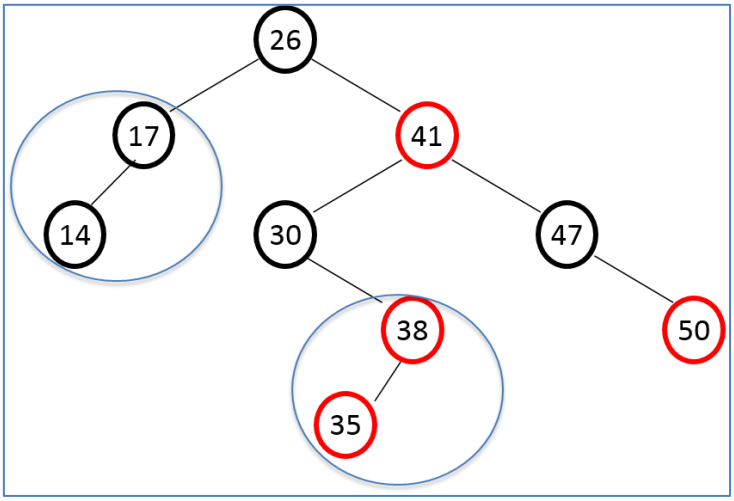


Figure 25.9 Example for Insertion

25.6.1 Insertion in Detail

The first step of insertion is creation of a new node with the data element and insertion of the new node into the Red-black tree using the binary search tree insertion module. Next the new node needs to be assigned a color. If we make the node black there may be the case of one root-to-external-node path has an extra black node (black pointer). This particular type of problem is hard to rectify. We could make the node red there may be the case of one root-to-external-node path may have two consecutive red nodes (pointers). This problem can be remedied by color flips and/or a rotation. In case there is a violation of property, we move this violation of property 3 up the tree, by recoloring until it can be fixed with rotations and recoloring.

25.6.2 Red-Black Trees: The Problem with Insertion

Let us assume that we are going to insert 8 (Figure 25.10), the place where it is inserted is decided by the binary search property and let us color it red.

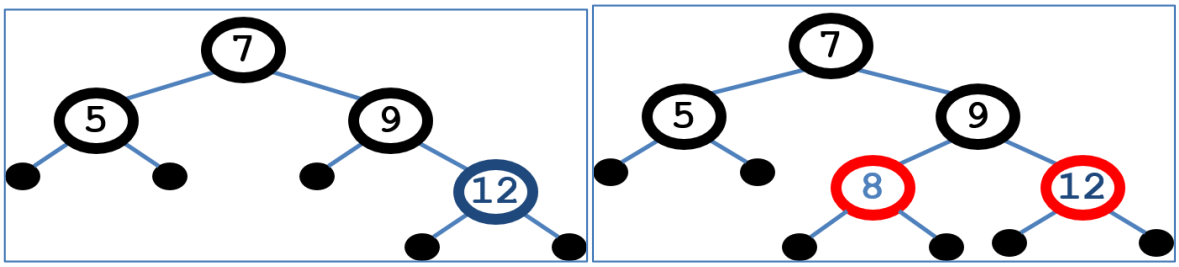


Figure 25.10 Example of Inserting 8

Let us assume that now we want to insert 11, it cannot be colored red (violation of property 3) and cannot be colored black (violation of property 4). The solution is to rotate and recolor the node black as shown in Figure 25.11. In the following section we will discuss the different methods needed for rebalancing.

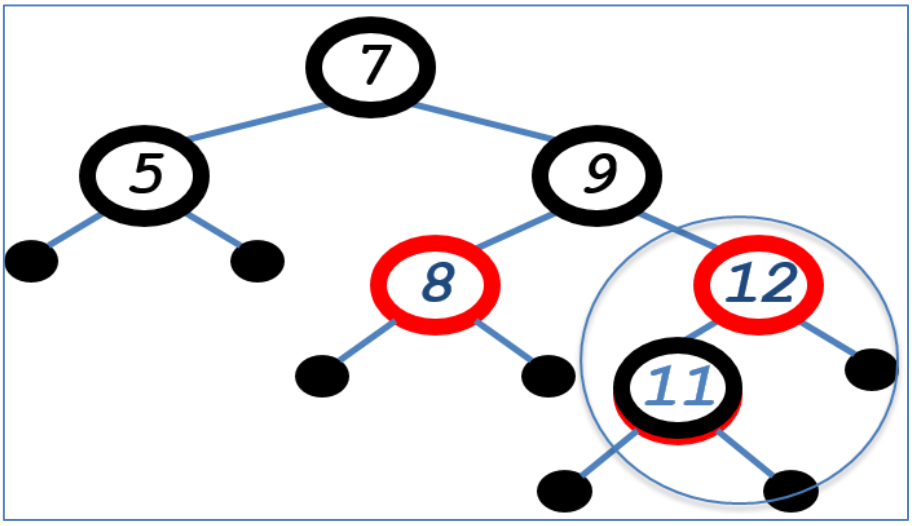


Figure 25.11 Insertion of 11

25.7 Red-Black Tree – Rebalancing after Insertion

Now in order to explain the cases of rebalancing we will define the concept of uncle of a node x is the sibling of the parent of x. Now there are 3 cases where rebalancing will be carried out

25.7.1 Case 1

Conditions for case 1 (Figure 25.12 (a))

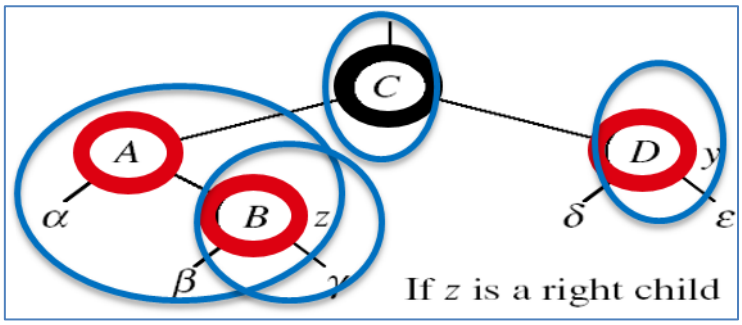
- Now the let check whether the node that is inserted z's (B) "uncle" y – (D) is red.
- We check whether z is a right child and
- Whether z's grandparent that p[p[z]] is (here c) black where p[z] is the parent of z. This will be the case when p[z] is red.
- We also check if both z and its parent p[z] (here B and A) are red

Only if all these conditions are true, do we carry out the operations associated with Case1.

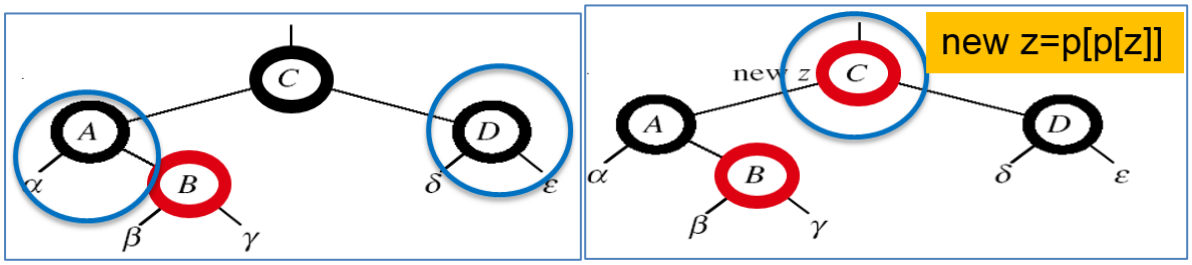
Steps for Case 1

If all the above conditions are true we essentially need to push the "red" violation up the tree. The same action will be taken whether z is a left or a right child. The following steps need to be carry out the following steps (Figure 25.12 (b) & (c)):

- We color p[z] black,
- We also color the uncle y of node z as black
- We then color the grandparent of z that is p[p[z]] red
- We then set new z = p[p[z]]



(a)



(b) (c)

Figure 25.12 Case 1 – Rebalancing after Insertion

Now we will discuss Case 3 before going onto to Case 2

25.7.2 Case 3

Conditions for case 3 (Figure 25.13)

- Now we check whether the node that is inserted z's (B) "uncle" y –is black and
- We check whether z is a left child of B

Steps for Case 3

If all the above conditions are true we essentially need to push the p[z] up the tree. The action will be taken when z is a left child. The following steps need to be carried out (Figure 25.13):

- We color p[z] black,
- We then color the grandparent of z that is p[p[z]] (C) red
- We then RIGHT-ROTATE(T, p[p[z]] (C))
- Now p[z] (B) is black

No longer are 2 reds in a row. We have performed a right rotation that preserves property 4 that is all downward paths contain same number of black nodes

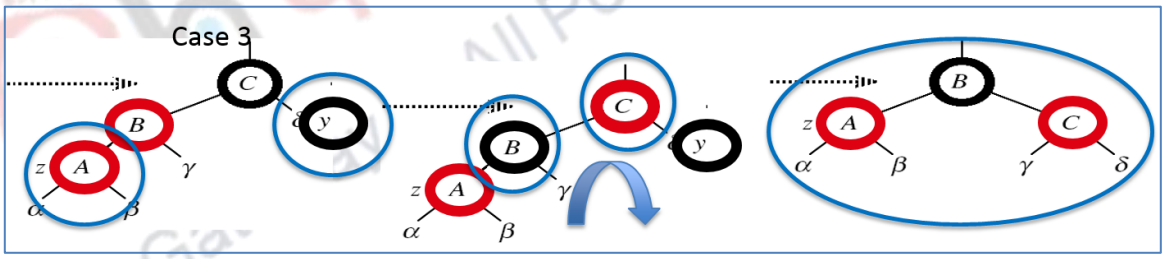


Figure 25.13 Case 3 – Rebalancing after Insertion

25.7.2 Case 2

Conditions for case 2 (Figure 25.14)

- Now the let check whether the node that is inserted z's (B) "uncle" y –is black and
- We then check whether z is a right child

Steps for Case 2

If all the above conditions are true we essentially need to push the p[z] up the tree. The following action will be taken when z is a right child. The following steps need to be carry out the following steps (Figure 25.13):

- We make z to be the parent of z (p[z]) that is z is now A.
- We then LEFT-ROTATE(T, B) about A- Now z is a left child

Now both z (A) and p[z] (B) are red and we have a case 3 which we rebalance using the steps outlined in Section 25.7.2.

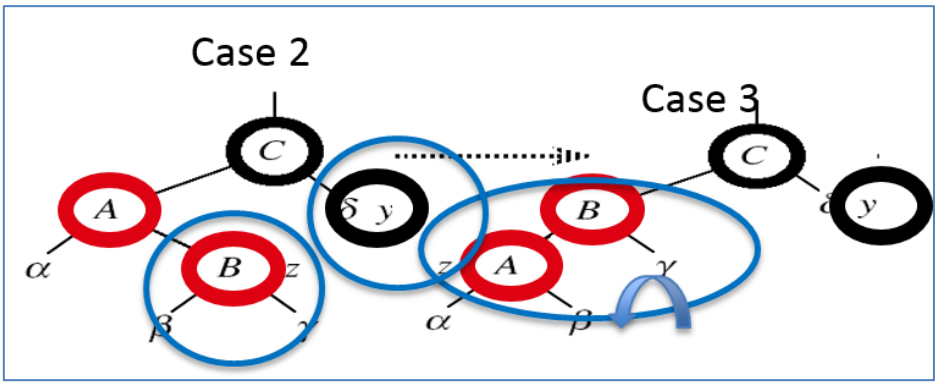


Figure 25.14 Case 2 followed by Case 3 – Rebalancing after Insertion

25.7 An Example showing all Three Cases

Figure 25.15 shows an example for the insertion of 4 into the Red-black tree.

4 is a right child. Here both 4 and it's parent 5 are red and the uncle of 4 (8) is also red. Moreover 4's grandparent is black. All these conditions gives rise to **Case 1 rebalancing**. Accordingly we color the parent of 4 as well as it's uncle 8 as black. We then color the grandparent of 4 that is 7 as red and make 7 as the node which leads to rebalancing.

Now in the rebalanced tree the node 7 and it's parent 2 are red, 7 is a right child and 7's uncle is black, all conditions that lead to a **Case 2 rebalancing**. Accordingly we make the node for rebalancing to be 7's parent 2, and then carry out a LEFT-ROTATE of the tree with 2 as the pivot.

Now 2 and it's parent 7 are red, 2 is a left child and 2's uncle 14 is black. All these conditions gives rise to **Case 3 rebalancing**. Accordingly we color parent of 2 that is 7 black, and it's grandparent 11 red. We then RIGHT-ROTATE the tree with 11 as the pivot. Now we end up with the parent of 2 being black. Thus the insertion of 4 lead to three cases of rebalancing, case 1 with 4 as the main node, case 2 with 7 as the main node and finally case 3 with 2 as the main node. Finally we have a binary search tree that satisfies all the properties of the Red-Black tree.

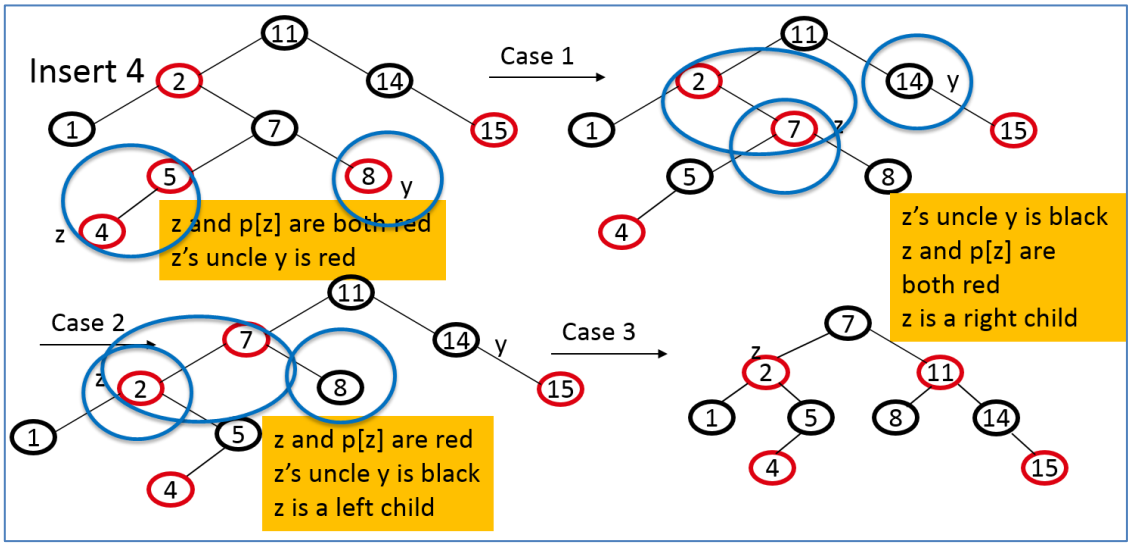


Figure 25.15 Example showing the 3 Cases

Summary

- Explained the concept of Red Black Trees
- Discussed the properties of Red Black Trees
- Described rotation of Red Black Trees
- Explained the Insertion into Red Black Trees