Data Structures
Red Black Tree Insert and Delete

Cain Susko

Queen's University
School of Computing

February 28, 2022

Intro to RBTree Insert and Delete

the insert and delete operations modify the tree such that we must fix the following in order for it to remain a RBTree.

1. colour changes
2. restructuring the links of the tree via. rotation

Rotations

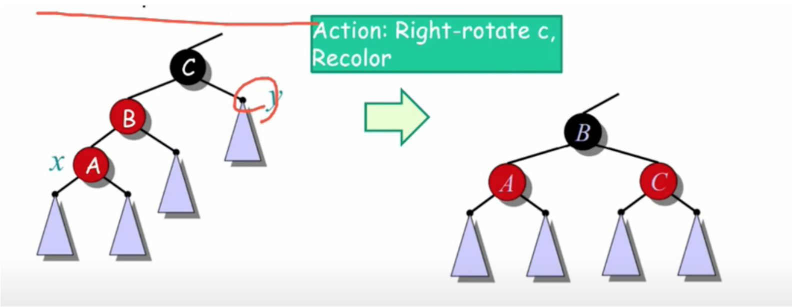
rotations are used to maintain the ordering of the keys. A rotation can be performed in $O(n)$

Insertion

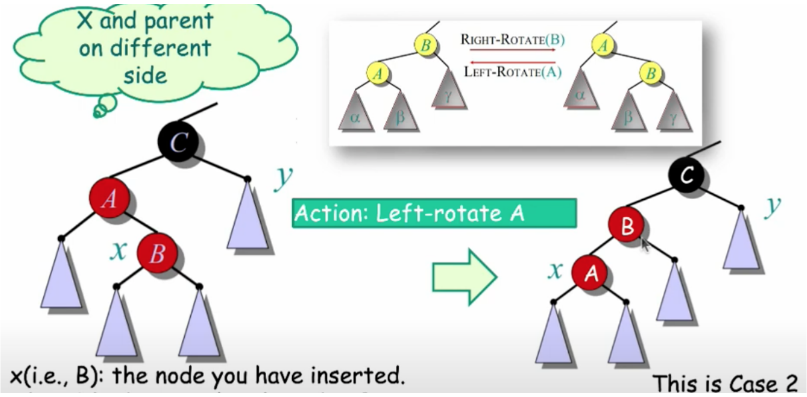
When doing an insertion, the only property that is violated is property 4 (see lesson 5-3-5). This is because nodes are only added as a leaf of the tree (as is the normal). There are 3 cases for this violation given x, the target node:

1. if the Aunt or Uncle node y of x is red then:
   recolour y, the parent of y, and the parent of x

2. if the Aunt or Uncle node is black, and x and it's parent are on the same side, then:
   right-rotate y and recolour the parent of x and y



3. if the Aunt or Uncle node is black, and x and it's parent are on different sides, then:
   left rotate x



X and parent on different side

RIGHT-ROTATE(B)

LEFT-ROTATE(A)

Action: Left-rotate A

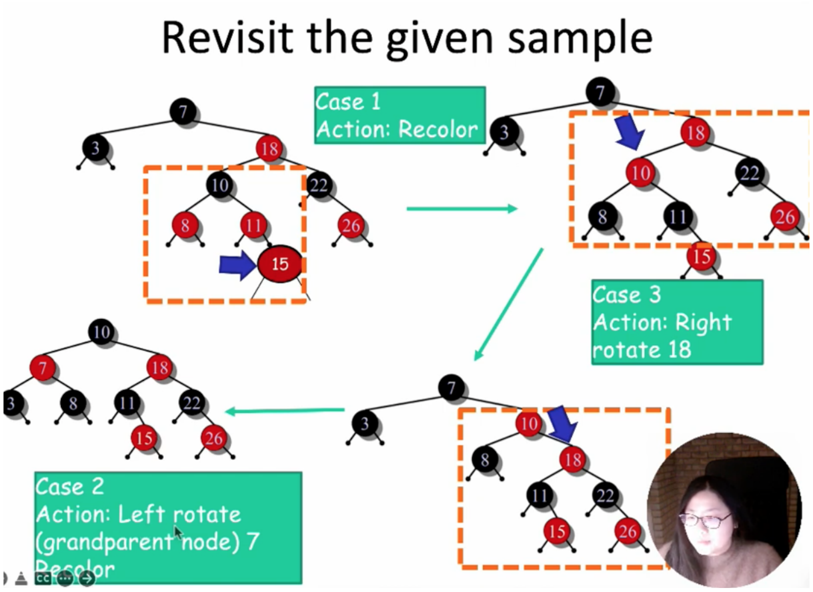
x(i.e., B): the node you have inserted.

This is Case 2

The time complexity of all 3 of these cases are:

$$O(\log n)$$

to see each of these cases in the context of eachother in a real example, observe the following:



Revisit the given sample

Case 1
Action: Recolor

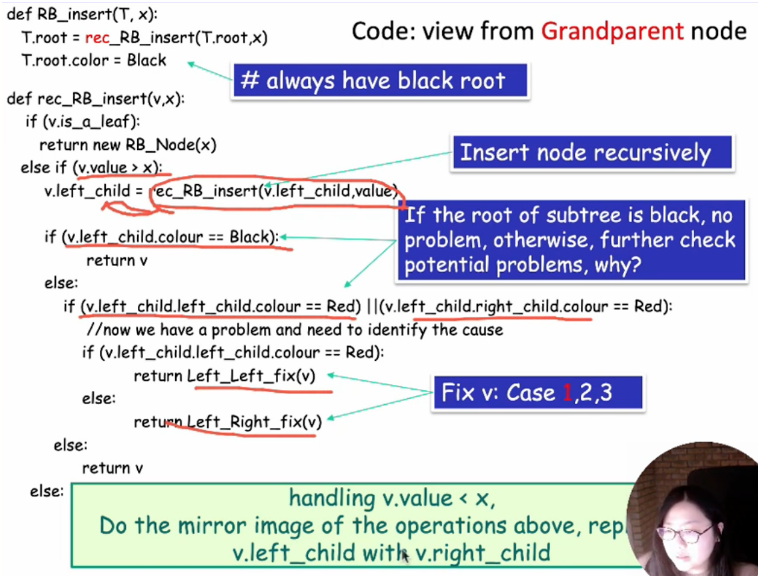
Case 3
Action: Right rotate 18

Case 2
Action: Left rotate (grandparent node) 7
Recolor

Algorith for Insertion from all this information, we can generalize the process for insertion as the following algorithm:

```
if current and parent are both red:
        if grandparent's other child (aunt or uncle) is red:
                colour grandpa red
        else if the current and parent are both on the same side:
                do a single rotation and recolour
        else:
                do a double rotation and recolour
```

An implementation can also be seen in the following figure



Code: view from Grandparent node

```
def RB_insert(T, x):
  T.root = rec_RB_insert(T.root,x)
  T.root.color = Black

def rec_RB_insert(v,x):
   if (v.is_a_leaf):
     return new RB_Node(x)
  else if (v.value > x):
     v.left_child = rec_RB_insert(v.left_child,value)

     if (v.left_child.colour == Black):
           return v
     else:
       if (v.left_child.left_child.colour == Red) ||(v.left_child.right_child.colour == Red):
         //now we have a problem and need to identify the cause
         if (v.left_child.left_child.colour == Red):
               return Left_Left_fix(v)
         else:
               return Left_Right_fix(v)
    else:
         return v
 else:
```

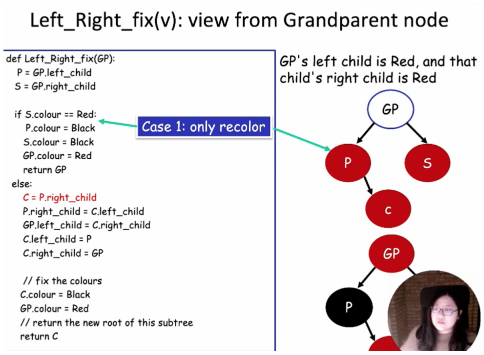
always have black root

Insert node recursively

If the root of subtree is black, no problem, otherwise, further check potential problems, why?

Fix v: Case 1,2,3

handling v.value < x,
Do the mirror image of the operations above, rep
v.left_child with v.right_child

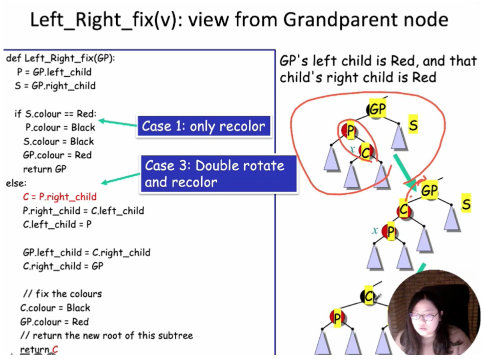


Left_Right_fix(v): view from Grandparent node

```
def Left_Right_fix(GP):
   P = GP.left_child
   S = GP.right_child

   if S.colour == Red:
      P.colour = Black
      S.colour = Black
      GP.colour = Red
      return GP
   else:
      C = P.right_child
      P.right_child = C.left_child
      GP.left_child = C.right_child
      C.left_child = P
      C.right_child = GP

      // fix the colours
      C.colour = Black
      GP.colour = Red
      // return the new root of this subtree
      return C
```

Case 1: only recolor

GP's left child is Red, and that child's right child is Red



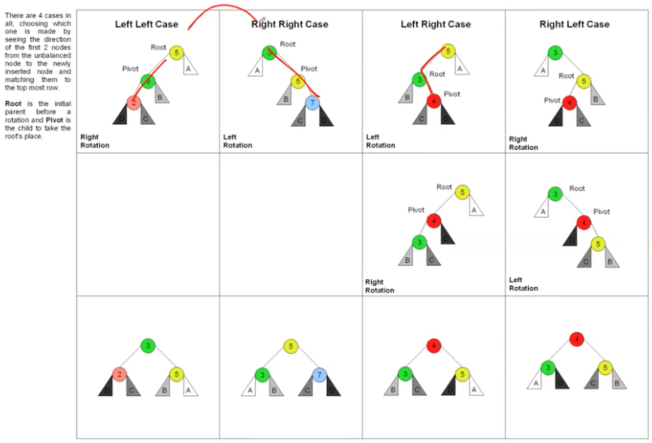
Left_Right_fix(v): view from Grandparent node

```
def Left_Right_fix(GP):
   P = GP.left_child
   S = GP.right_child

   if S.colour == Red:
      P.colour = Black
      S.colour = Black
      GP.colour = Red
      return GP
else:
      C = P.right_child
      P.right_child = C.left_child
      C.left_child = P

      GP.left_child = C.right_child
      C.right_child = GP

      // fix the colours
      C.colour = Black
      GP.colour = Red
      // return the new root of this subtree
      return C
```

Case 1: only recolor

Case 3: Double rotate and recolor

GP's left child is Red, and that child's right child is Red



There are 4 cases in all, choosing which one is made by seeing the direction of the first 2 nodes from the unbalanced node to the newly inserted node and matching them to the top most row.

Root is the initial parent before a rotation and **Pivot** is the child to take the root's place.