# Red-Black Trees

→ A BST with 1 extra bit of storage per node, its color (red or black)

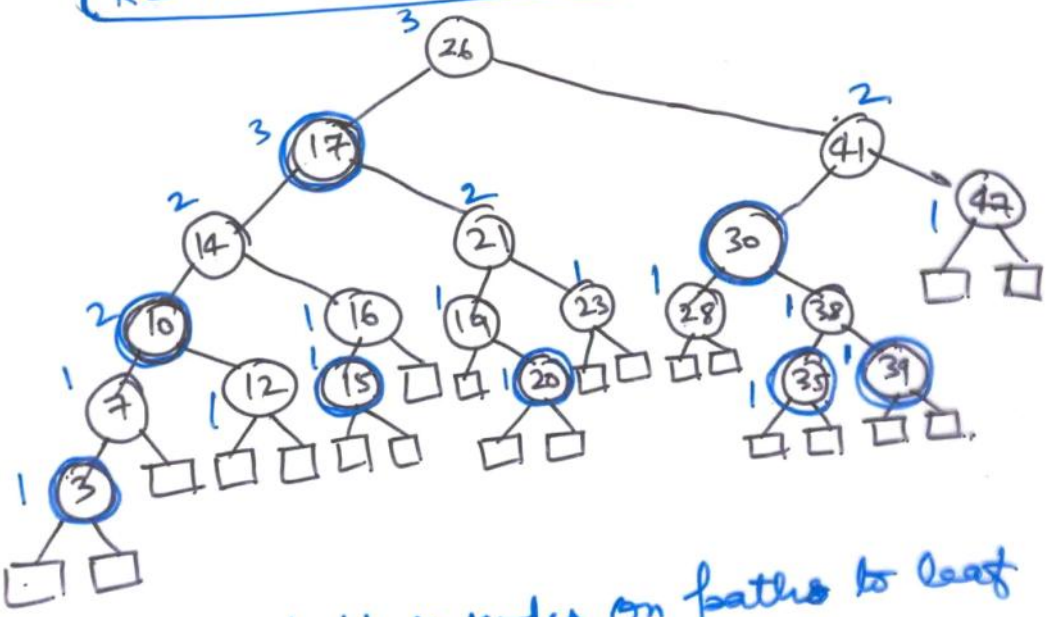
→ By constraining the way nodes can be colored on any path from the root to a leaf, it ensures that no path is more than twice as long as any other

→ Ensures the worst-case performance of the basic BST ops on red black trees is $O(h) = O(\lg n)$

## RB properties

→ (1) Every node is either red or black
(2) The root is black
(3) Every leaf is black.
(4) If a node is red, then both of it's children are black
(5) Every simple path from a node to a descendant leaf contains the same number of black nodes

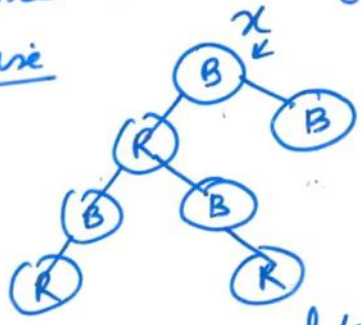
(Red are marked with blue circles)




bh(x) ⇒ No. of black nodes on paths to leaf not counting x

→ If T is an R-B Tree, we define bh(T) as the black-height of its root

Exercise




bh(x) = 2 h(x) = 4 (or 3)

– Lemma : A red-black Tree with $n$ internal nodes has height at most $2\lg(n+1)$.

Proof :

→ first show that any subtree rooted at node $x$ contains at least $2^{bh(x)} - 1$ internal nodes

→ if height of $x = 0$

$\Rightarrow$ $x$ must be a leaf and subtree rooted at $x$ contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes

→ positive height


→ depending on color

By induction : each child has $\geq 2^{bh(x)-1} - 1$ internal nodes.

so subtree rooted at $x$ has

$$\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1$$

$$= 2^{bh(x)} - 1$$

→ Let h be the height of the tree. According to R-B. tree property, at least half of the nodes on any simple path from the root to a leaf, not including the root, must be black.

Consequently the black height of the root must be at least $h/2$;

⇒ for x as root

$$n \geq 2^{bh(x)} - 1 \geq 2^{h/2} - 1$$

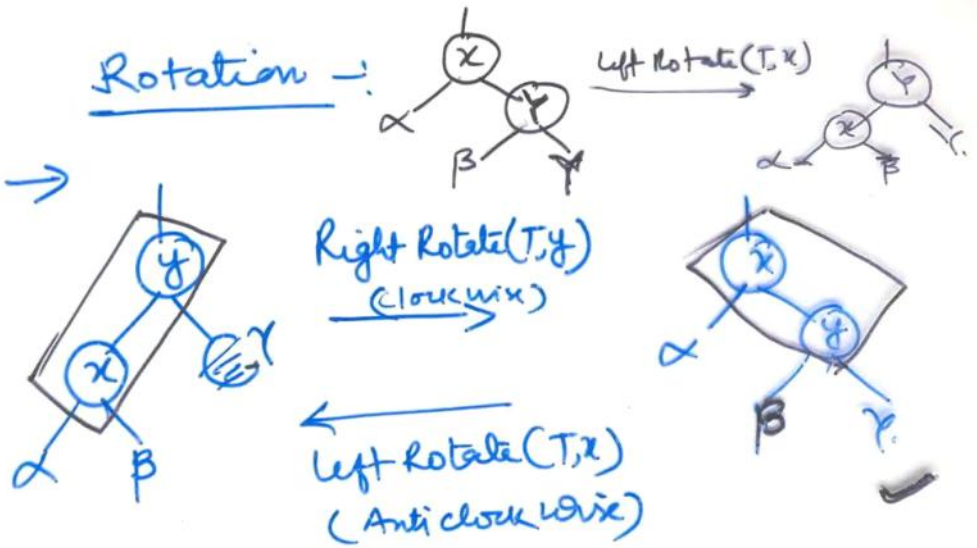
$$\Rightarrow n+1 \geq 2^{h/2}$$

$$\Rightarrow \lg(n+1) \geq h/2$$

$$\Rightarrow h \leq 2\lg(n+1)$$

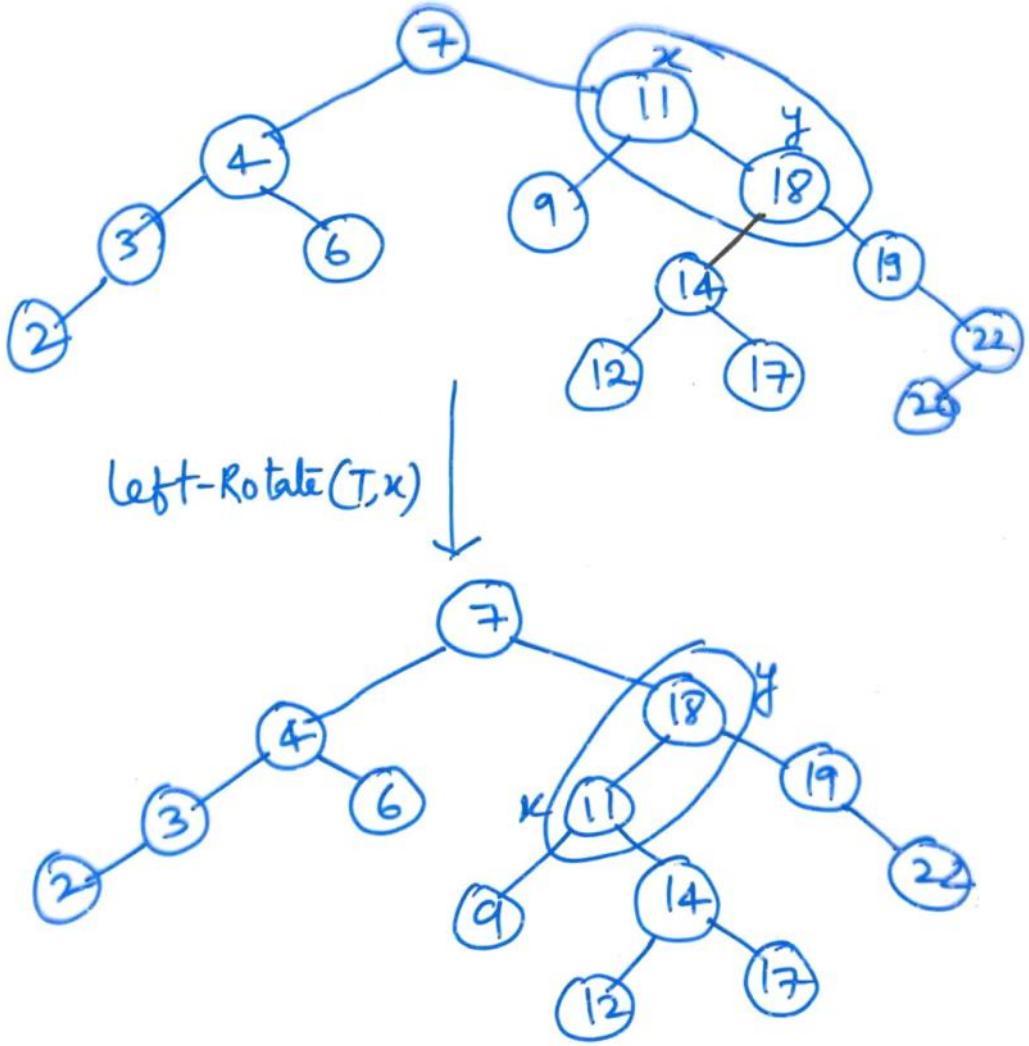
⇒ All red-black operations runs is $O(\lg n)$ worst case time

## Rotation :-




## Left ROTATE(T, x)

```
//1   y ← right[x]          // set y
//2   right[x] ← left[y]    // turn y's left subtree into x's right subtree
//3   p[left[y]] ← x
//4   p[y] ← p[x]
//5   if p[x] = nil[T]
//6     then root[T] ← y
//7     else if x = left[p[x]]
//8       then left[p[x]] ← y
//9       else right[p[x]] ← y
//10  left[y] ← x
//11  p[x] ← y
```




left-Rotate(T, x)




→

# Insertion in R-B Trees

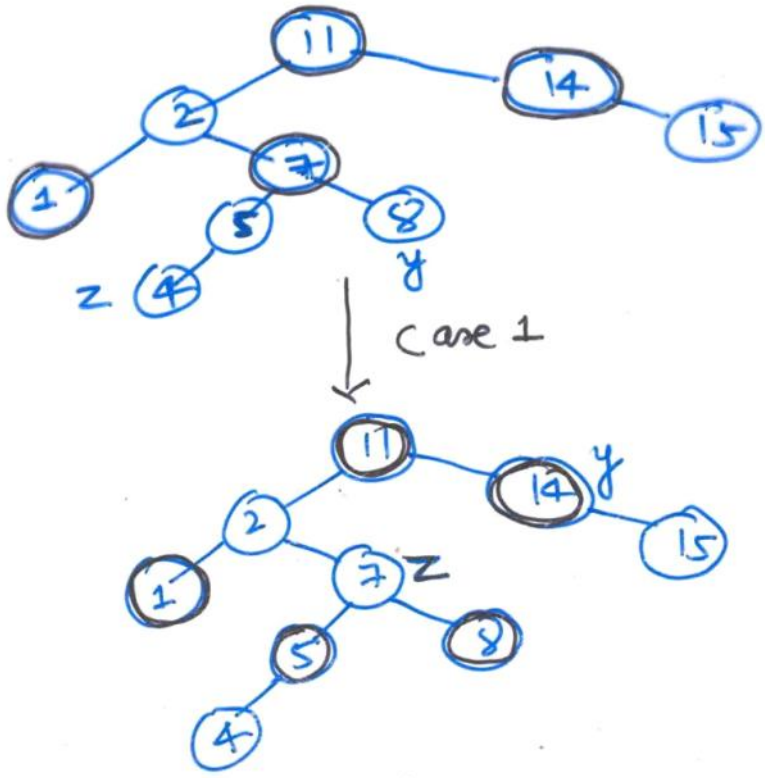
→ Insertion can be done in $O(\lg n)$ time

→ Basic steps

* Use Tree-insert (for BST's) to insert a node, Z, into T.
* Color node Z red
* fix-up the modified tree by re-coloring nodes and performing rotations to preserve the RB tree property. This is done by calling RB-Insert-fixup

## RB-Insert Fixup (T, z)

* Case 1 :- The child z, it's parent and it's uncle y are red
  - → change parent and uncle to black
  - → change grandparent to red (preserves black height)
  - → Make grandparent new z, grandparent's uncle new y



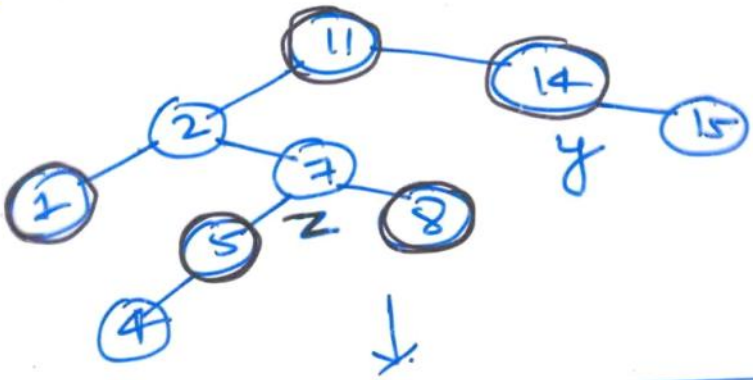
Case 2 :

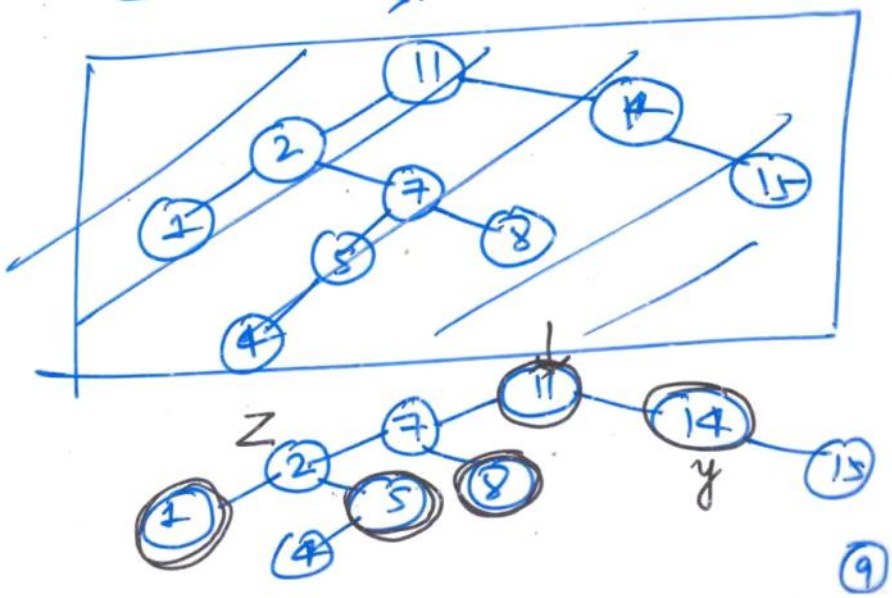
z and it's parent are red, it's uncle y is black, and z is the right child. Converts case 2 into Case 3

→ Rotate left on z's parent

→ former parent becomes new z

→ old z becomes parent.



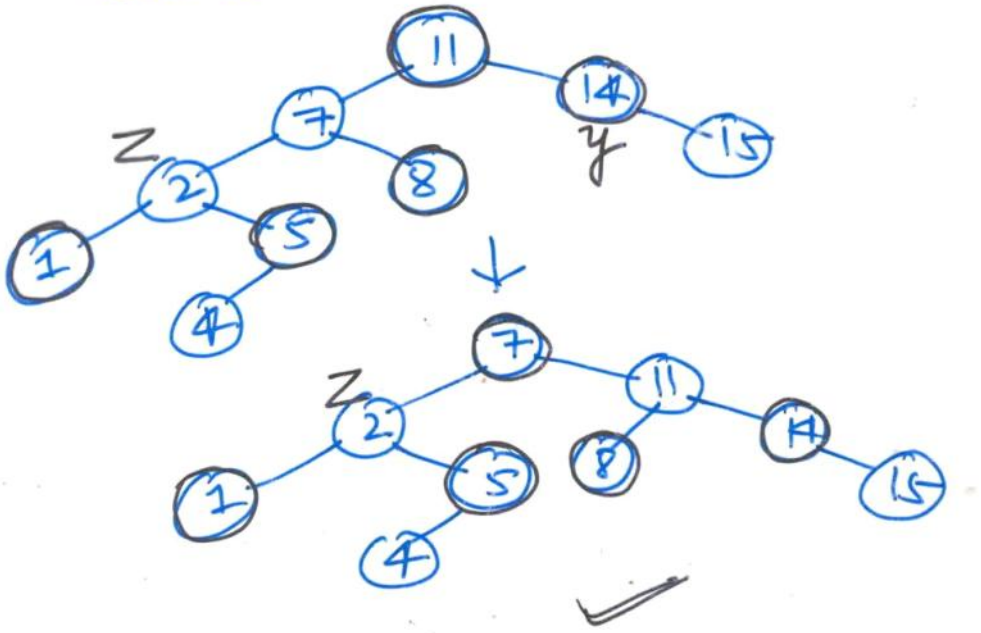
Case 3 :

z and it's parent are red, it's uncle y is black and z is a left child

→ Rotate right, away from z, on z's grandparent

→ Color z's parent black.

→ Now z's old grandparent is z's sibling; color it red to maintain the right number of black nodes down each path

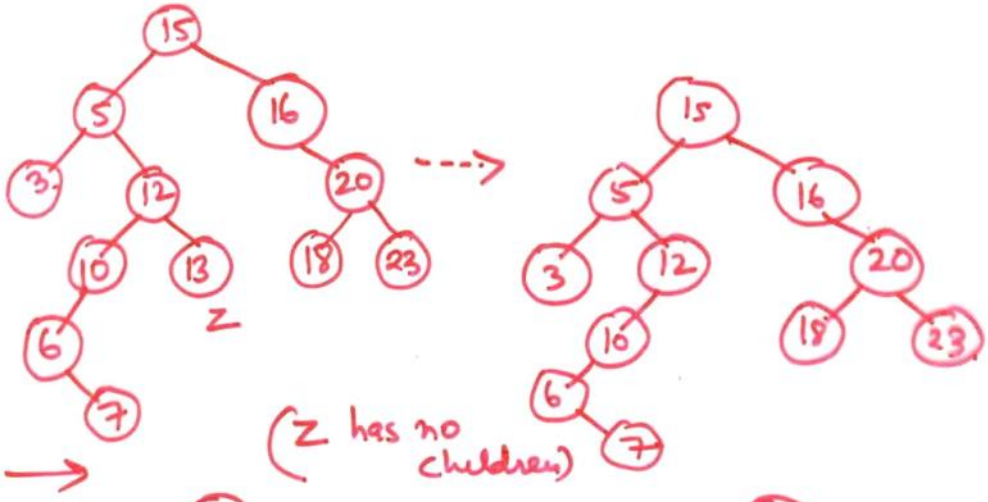


## RB-INSERT(T, z)

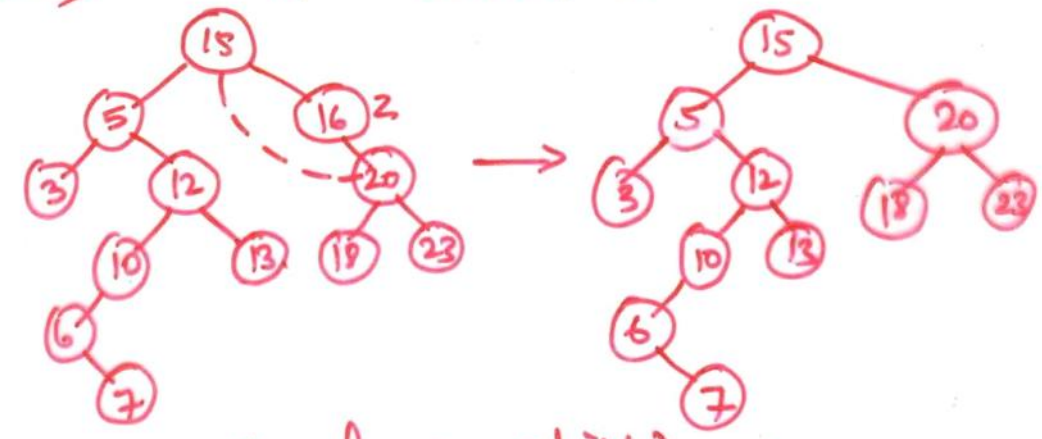
```
//1  y ← nil[T]
//2  x ← root[T]
//3  while x ≠ nil[T]
//4    do y ← x
//5       if key[z] < key[x]
//6         then x ← left[x]
//7         else x ← right[x]
//8  p[z] ← y
//9  if y = nil[T]
//10    then root[T] ← z
//11    else if key[z] < key[y]
//12           then left[y] ← z
//13           else right[y] ← z
//14 left[z] ← nil[T]
//15 right[z] ← nil[T]
//16 color[z] ← RED
//17 RB-INSERT-FIXUP(T, z)
```

## RB-INSERT-FIXUP(T,Z)

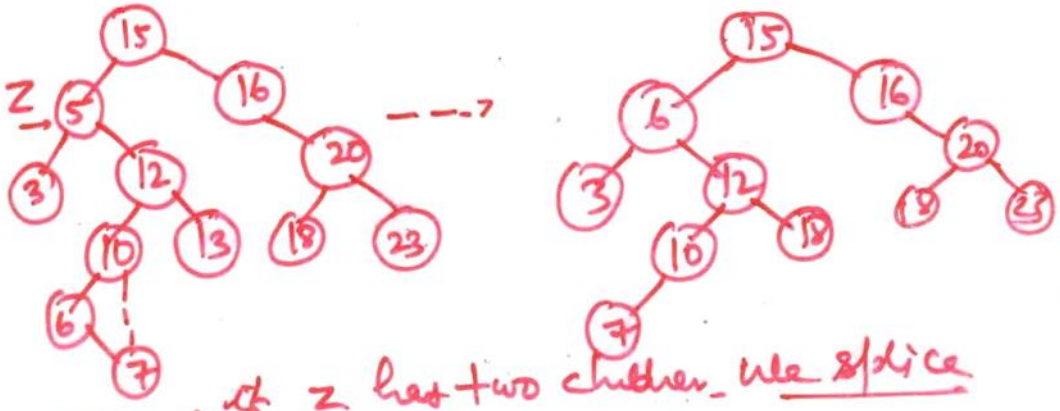
```
//1.  while color[p[z]] = RED
//2      do if p[z] = left[p[p[z]]]
//3           then y ← right[p[p[z]]]
//4                if color[y] = RED
//5                then color[p[z]] ← BLACK          ┐
//6                     color[y] ← BLACK             │ case 1
//7                     color[p[p[z]]] ← RED         │
//8                     z ← p[p[z]]                  ┘
//9                else if z = right[p[z]]           ┐
//10                    then z ← p[z]                │ case 2
//11                         LEFT-ROTATE(T,z)        ┘
//12                    color[p[z]] ← BLACK          ┐
//13                    color[p[p[z]]] ← RED         │ case 3
//14                    RIGHT-ROTATE(T, p[p[z]])     ┘
//15           else (same as then clause
                   with "right" and "left"
                   exchanged)
//16 color[root[T]] ← BLACK
```




(z has no children)




(z has one child) → splice out z



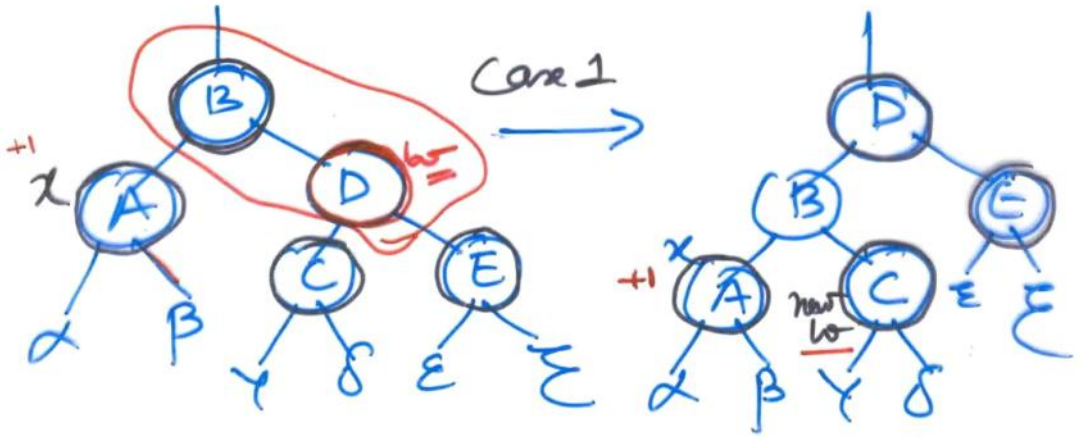

if z has two children, use splice out it's successor y.

Case-1 (w is red)

x's sibling w is red; (x is Doubly black)

→ Rotate (left) towards x around x's parent and change colors.

→ Now x's sibling is black and.
We are in Case 2, 3 or 4
x remains doubly black



Case 2 (w is black, both w's children are black)

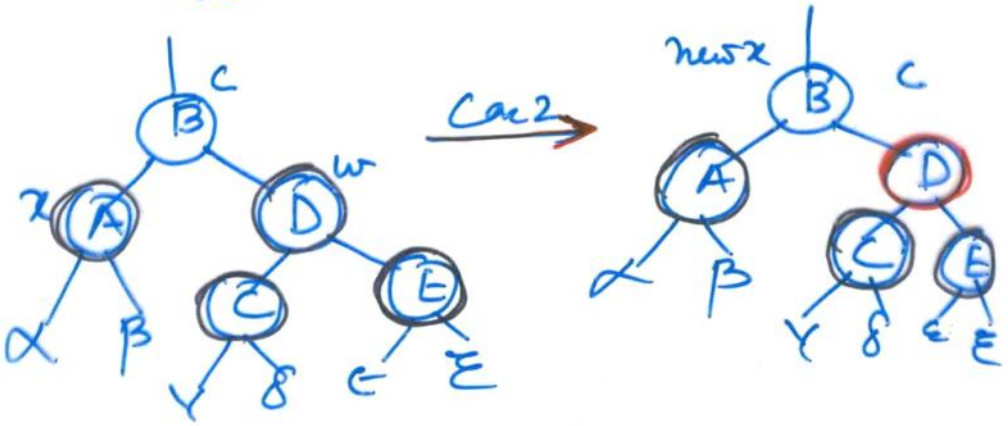
→ w is black, and are both w's children; x is doubly black.

→ Make w red, make x point to parent

→ preserves consistency of black height, counting x.

→ Moves problem closer to root; extra black now applies to C

→ Exists while loop and colors new x Black, it came from case 1



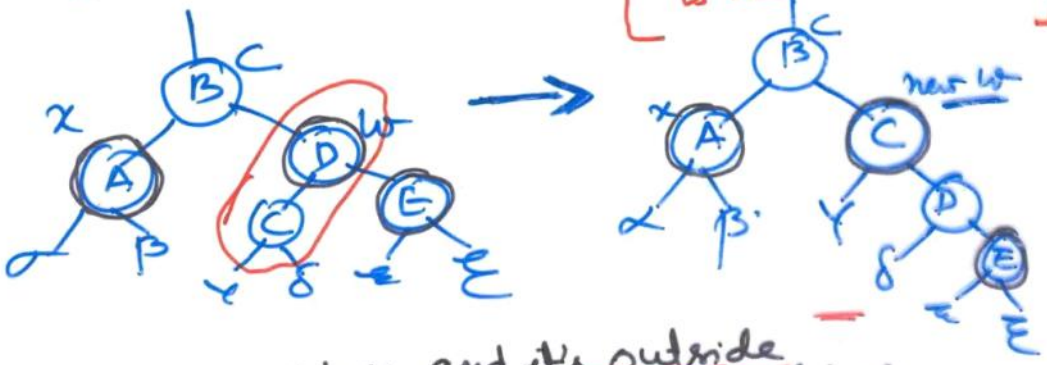

→ We have not solved the problem. but we have pushed it up towards the root one step.

Case 3: (w is black and it's inside child (towards x) is red; x is doubly black)

→ Rotate outward around w, x remains doubly black. and change colors. and other child is black.

→ Make x's new sibling black, siblings new right child red.

[far nephew is black, near nephew is red]

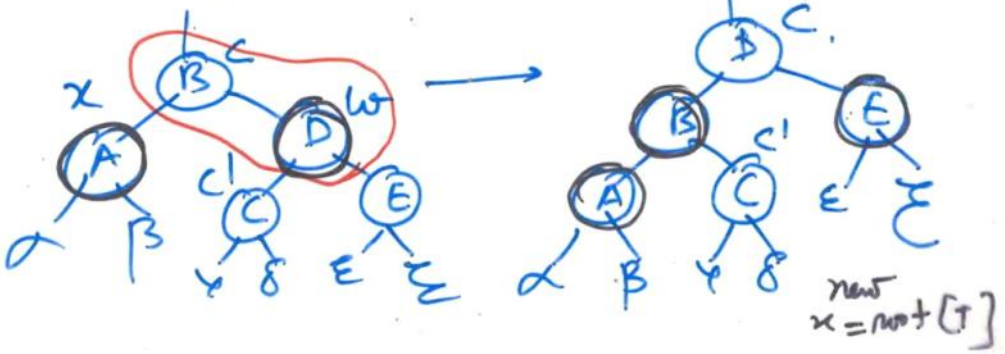



Case 4: w is black, and it's outside child is red; x is Doubly black

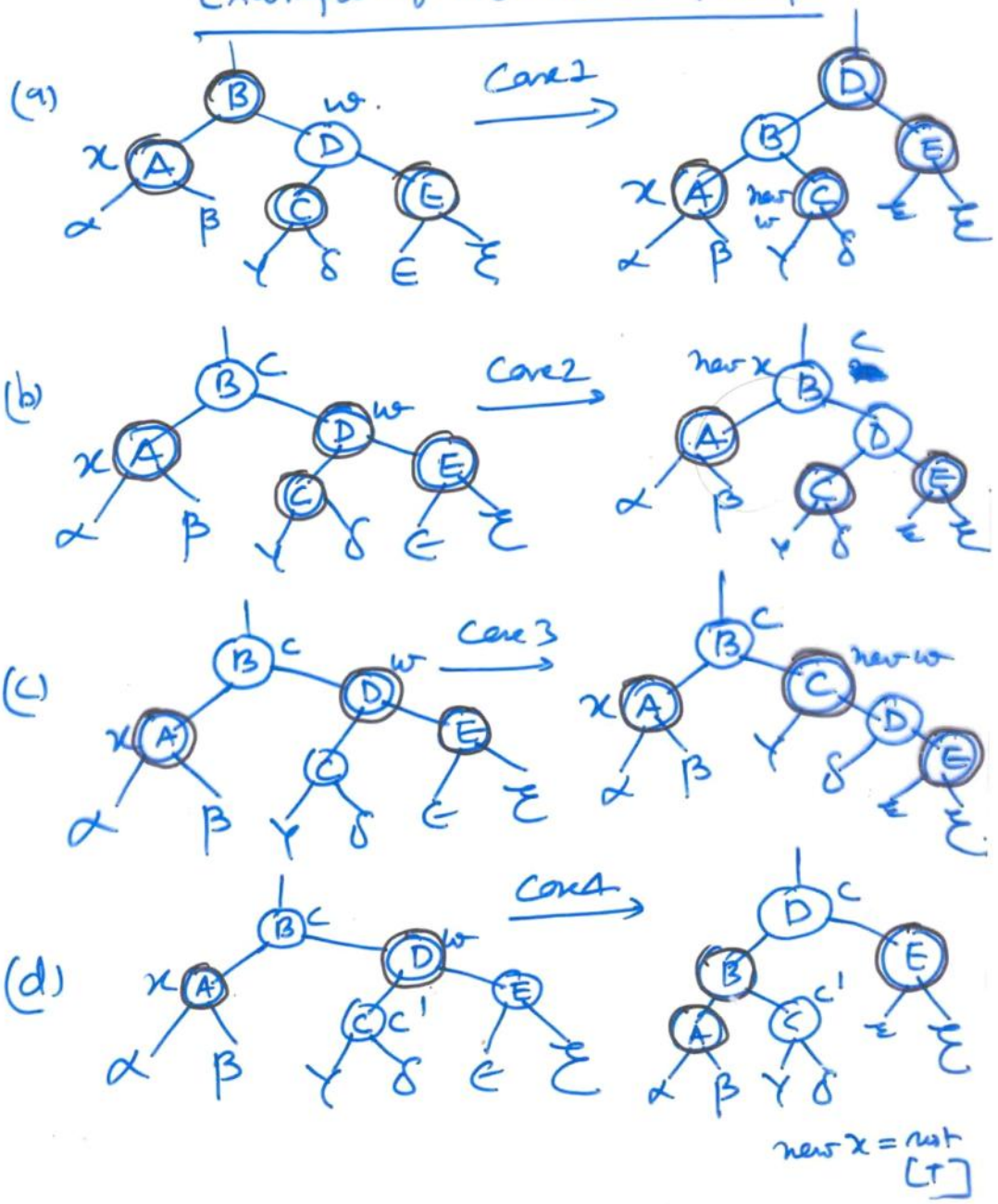
* Move E towards root of right side color it black. (rotate and change color).

→ Do this by rotating towards x.

→ setting x to the root terminate while loop



# Example of RB-Delete fixup.



RB-DELETE(T,z)

```
if left[z] = nil[T] or right[z] = nil[T]
  then y ← z
  else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
   then x ← left[y]
   else x ← right[y]
p[x] ← p[y]
if p[y] = nil[T]
  then root[T] ← x
  else if y = left[p[y]]
       then left[p[y]] ← x
       else right[p[y]] ← x
if y ≠ z
  then key[z] ← key[y]
       Copy y's satellite data into z
if Color[y] = BLACK
   then RB-DELETE-FIXUP(T,x)
return y.
```

## RB-DELETE-FIXUP (T, x)

```
//1  while x ≠ root[T] and color[x] = BLACK
//2      do if x = left[p[x]]
//3          then w ← right[p[x]]
//4               if color[w] = RED
//5                  then color[w] ← BLACK.
//6                       color[p[x]] ← RED.
//7                       LEFT-ROTATE(T, p[x])          } Case 1
//8                       w ← right[p[x]]
//9               if color[left[w]] = BLACK and color[right[w]]
                                                        = BLACK
//10                 then color[w] ← RED                } Case 2.
//11                      x ← p[x]
//12              else if color[right[w]] = BLACK
//13                   then color[left[w]] ← BLACK       } Case 3
//14                        color[w] ← RED.
//15                        RIGHT-ROTATE(T, w)
//16                        w ← right[p[x]]
//17                   color[w] ← color[p[x]]            } Case 4
//18                   color[p[x]] ← BLACK.
//19                   color[right[w]] ← BLACK
//20                   LEFT-ROTATE(T, p[x])
//21                   x ← root[T]
//22      else (same as then clause with "right"
                and "left" exchanged).
//23 color[x] ← BLACK.
```

# The Search problem.

→ Find items with keys matching a given search key.

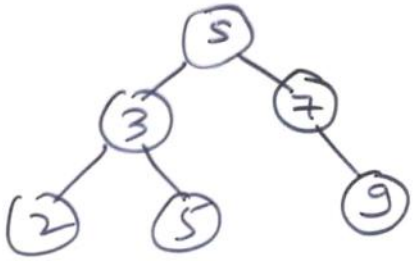
→ Applications:

* Keeping track of customer account info in a bank
* Keep track of the reservations on flight

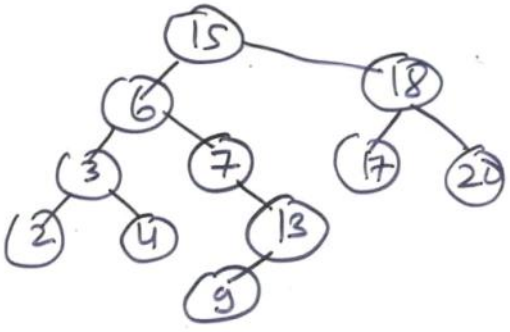
→ { Binary search tree }

→ if $y$ is in left subtree of $x$.
$Key[y] \leq Key[x]$

→ if $y$ is in right subtree of $x$
Then $Key[y] \geq Key[x]$




→ successor$(x) = y$, such that $Key[y]$ is the smallest $Key > Key[x]$




successor(15) = 17
successor(13) = 15
successor(9) = 13

# Insertion in BST.

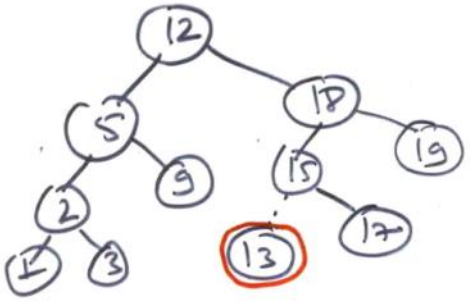
→ Beginning at the root, go down the tree and maintain
1. pointer x: traces the downward path
2. pointer y: parent of x ("trailing pointer").

→ if Key[x] < v, move to the right child of x.
else move to the left child of x.

→ when x is NIL, we found the correct position

→ if v < Key[y] insert the new node as y's left child else insert it as y's right child

(insert 13)



## Deletion in BST.

→ Goal: Delete a given node z from a BST.

→ Case 0: z has no children
Soln: make the parent of z point to Nil.

→ Case 1: z has one child
Soln: Delete z by making the parent of z points to z's child instead of to z

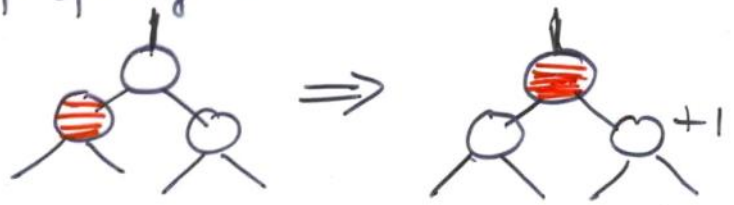
→ Case 3: z has two children
Soln: z's successor (y) is the minimum node in z's right subtree
→ y has either no children or one right child (but no left child)
→ Delete y from the tree (via case 0 or 1)

## Deletion algorithm.

1. Remove v
2. if V.Color = red, we are done!
   Else assume that u (v's non-nil child or nil) gets additional black Color
3. if u.Color () = red then u.setColor (black) and we are done
4. Else u's color is double black.

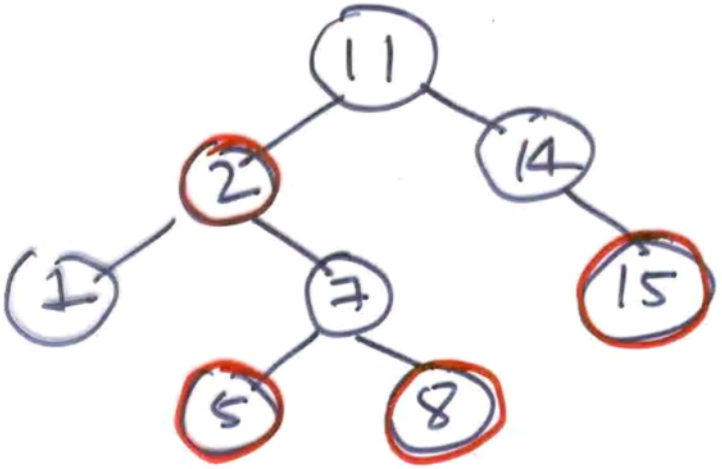
## Double-black Node.

Doubly black node is a node which has Color of two black, it violates property 1

→ 


(+1 means the node need another black to maintain the invariant of property)

Example :




insert 4