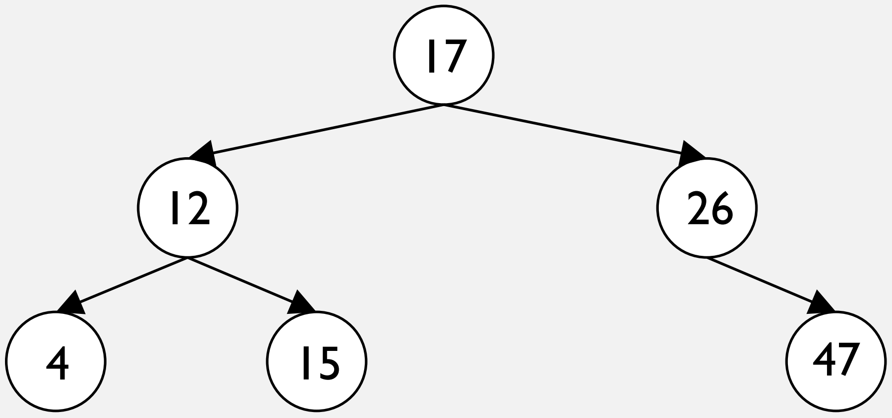
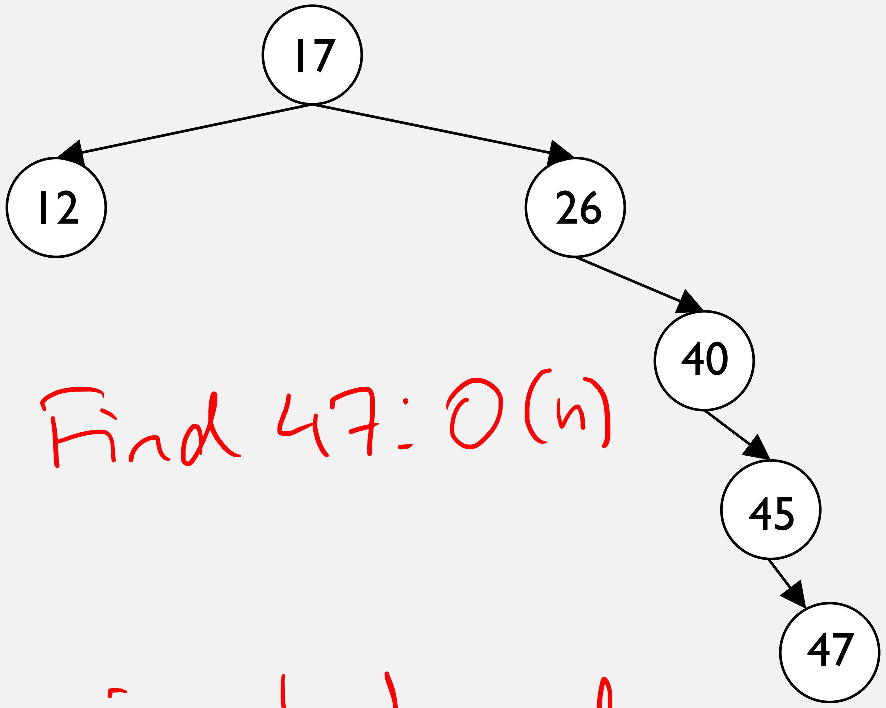
AVL TREES AND RED-BLACK TREES

ADS1, S2023

BALANCED TREES



Find 47: $O(\log n)$



Find 47: $O(n)$

log n time only if tree is balanced

HOW DO WE BALANCE TREES?

Keep restructuring the tree to ensure $O(\log n)$ access

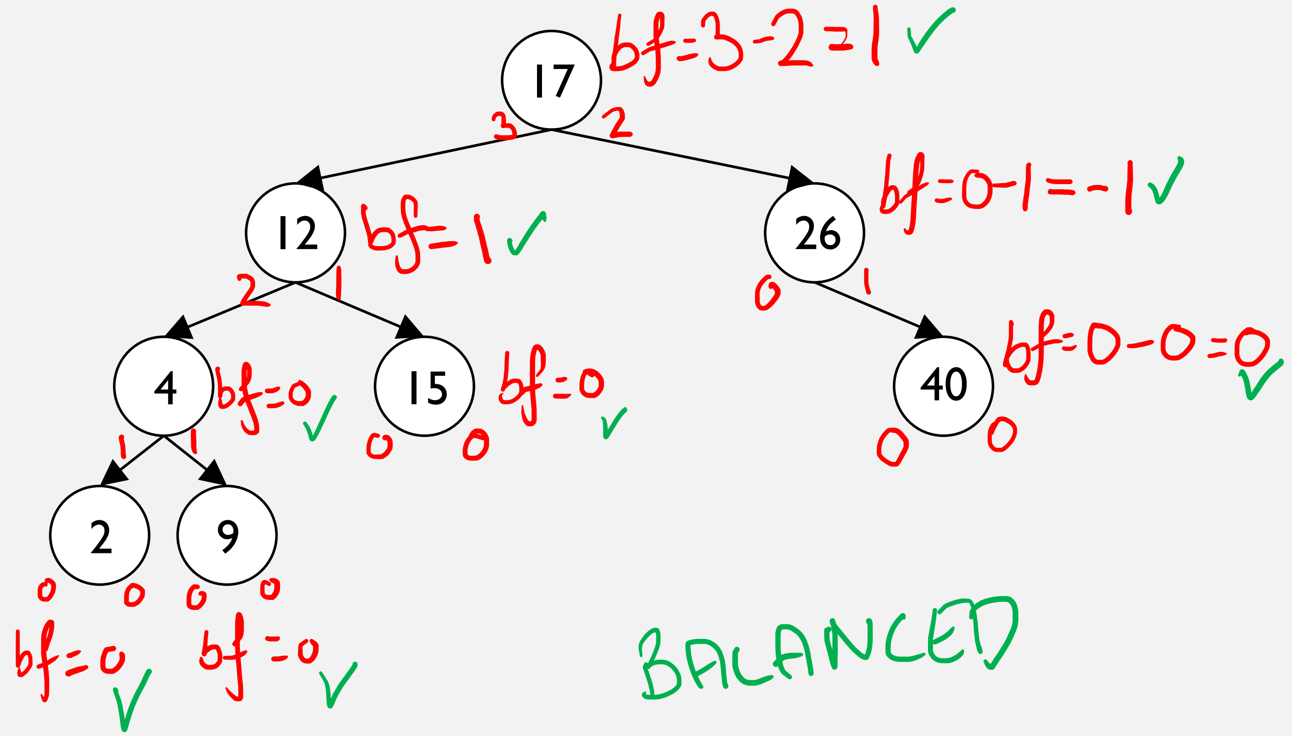
- (2,4)-trees
- AVL trees
- B-trees
- Randomized trees
- Red-black trees
- Splay trees
- … many more

AVL TREES

For each node:

The height difference of left and right subtree at most one.

AVL TREES



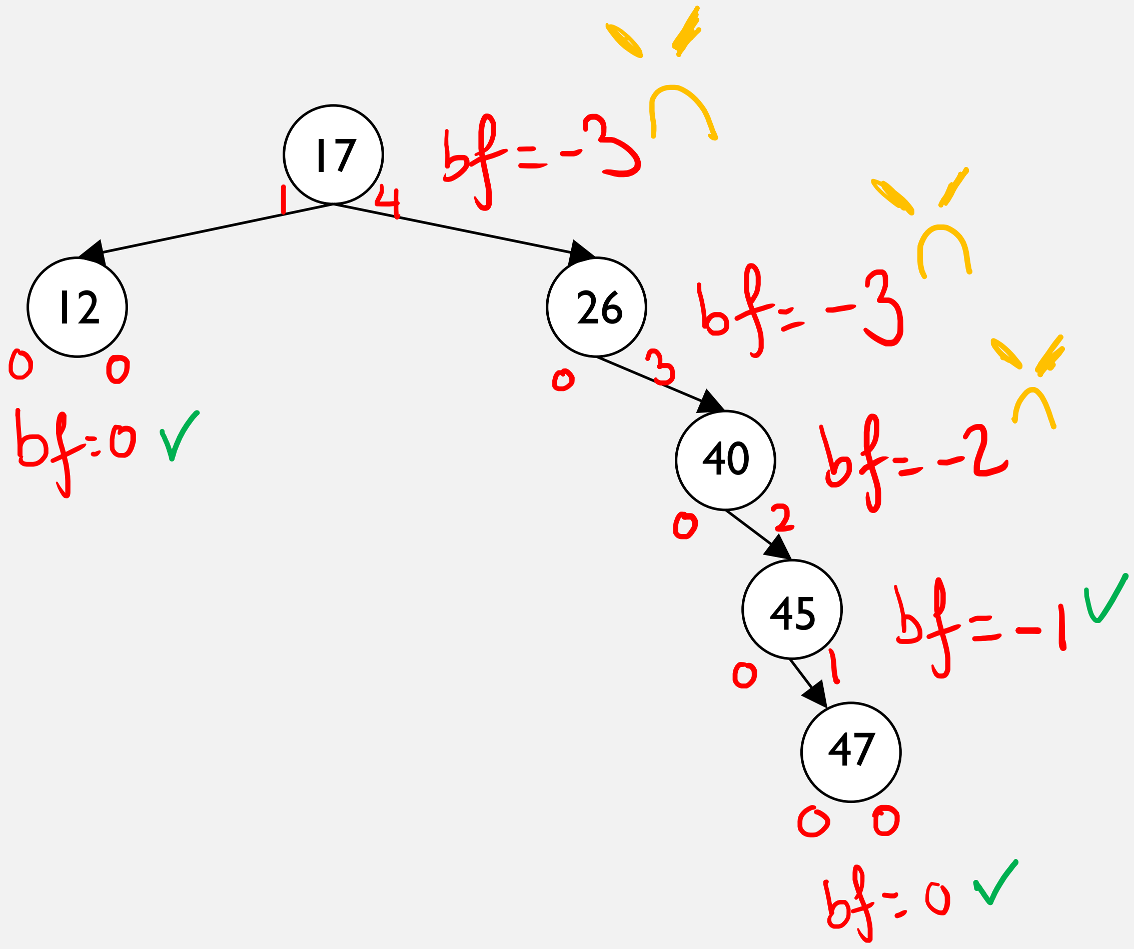
balance factor

↓

$$bf = h_l - h_r$$

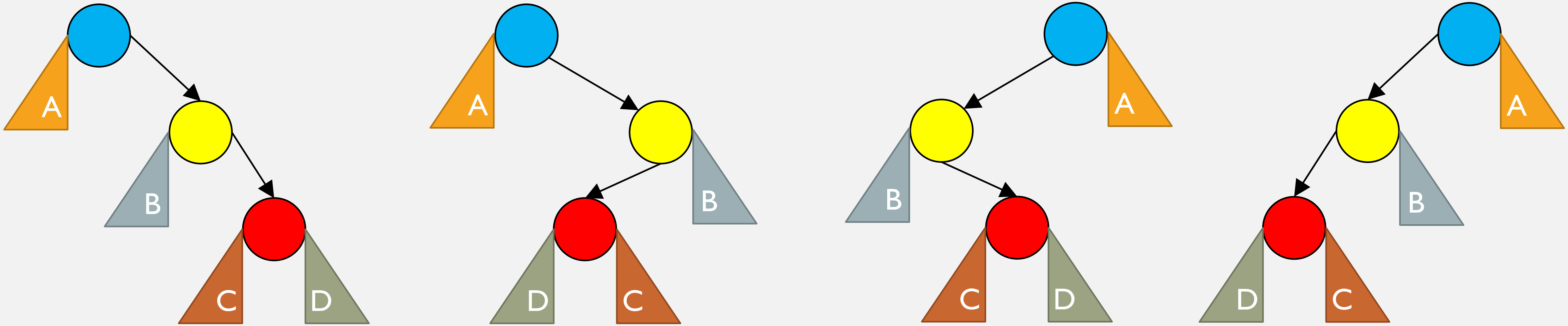
$$|bf| \leq 1$$

AVL TREES

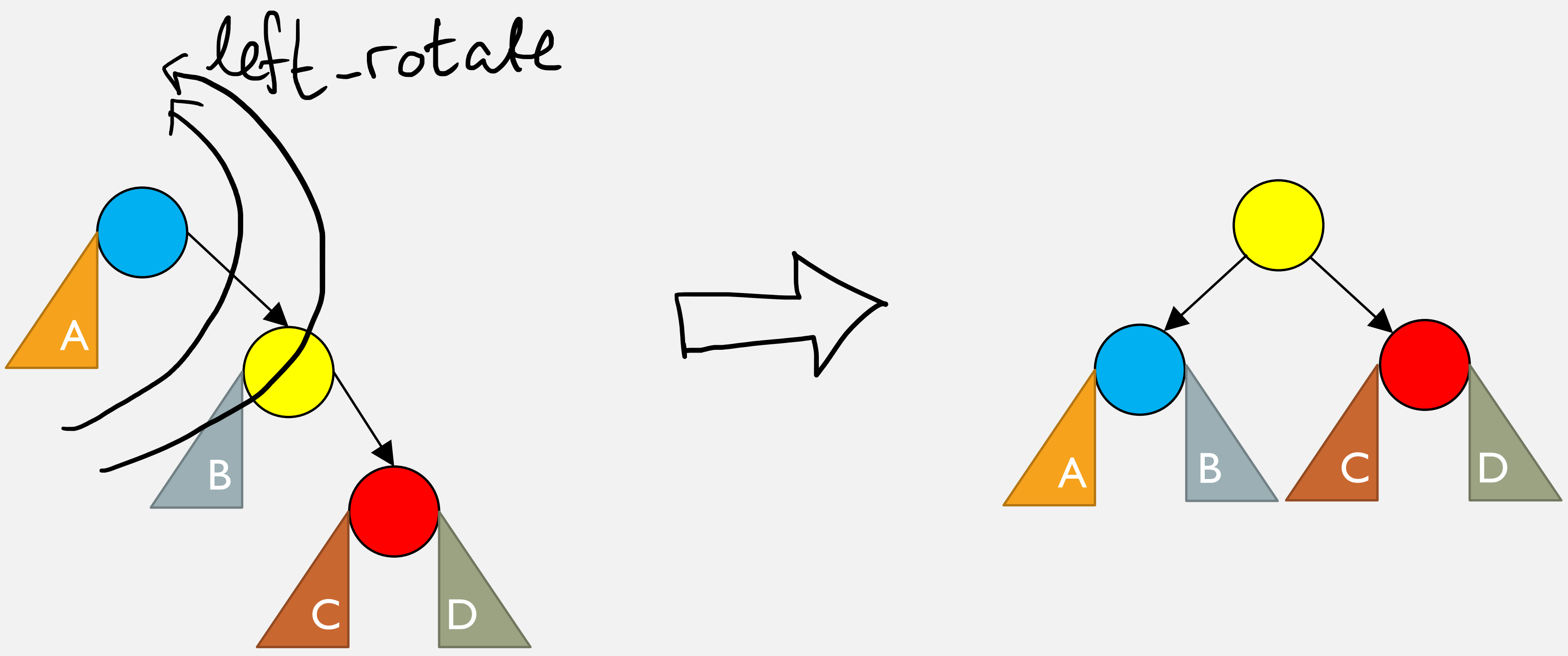


UNBALANCED

THE FOUR WAYS OF BEING UNBALANCED

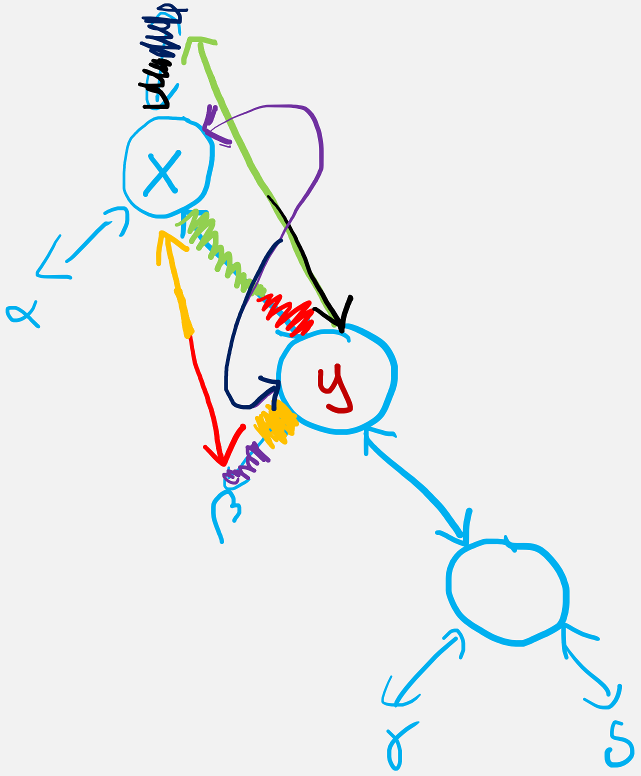


AVL-BALANCING OF RIGHT-RIGHT TREES



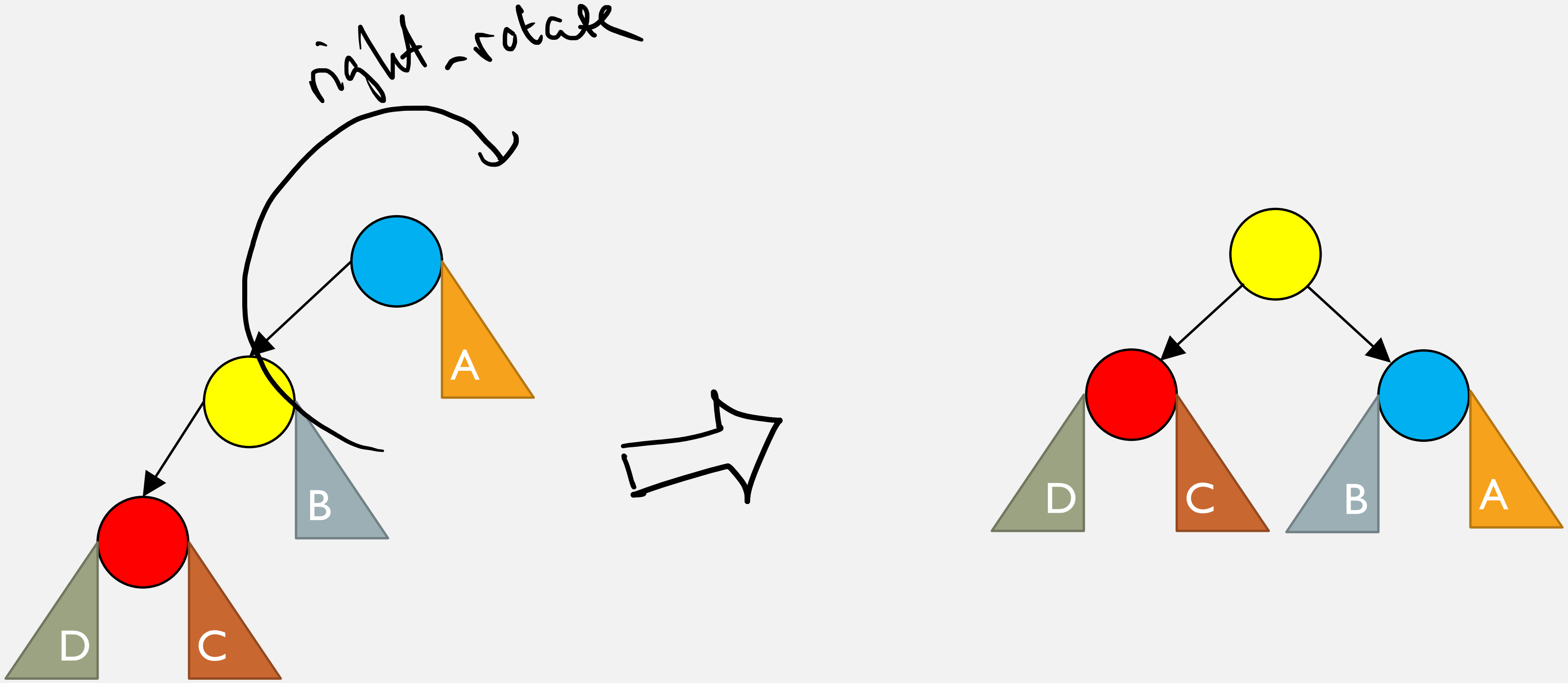
another way of thinking about it: pull up the middle one

BUT WHAT IS A ROTATION?



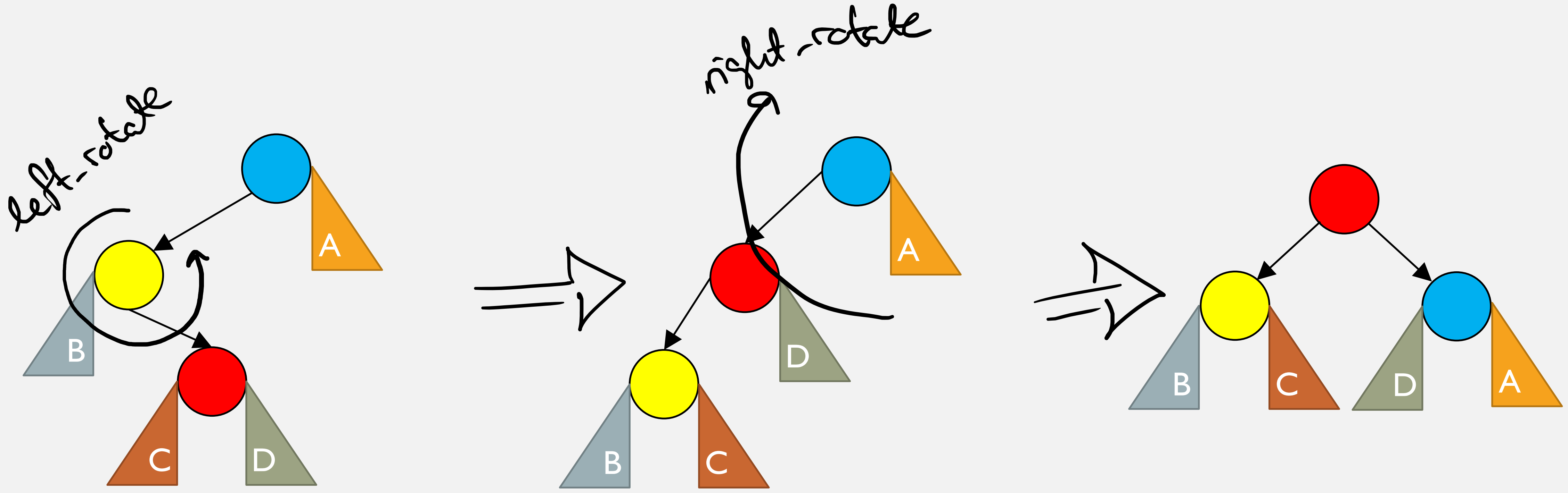

```
left_rotate(x):
    y = x.right
    x.right = y.left
    if y.left ≠ null:
        y.left.parent = x
    y.parent = x.parent
    if x.parent == null :
        T.root = y
    else if x == x.parent.right :
        x.parent.right = y
    else:
        x.parent.left = y
    y.left = x
    x.parent = y
```

AVL-BALANCING OF LEFT-LEFT TREES



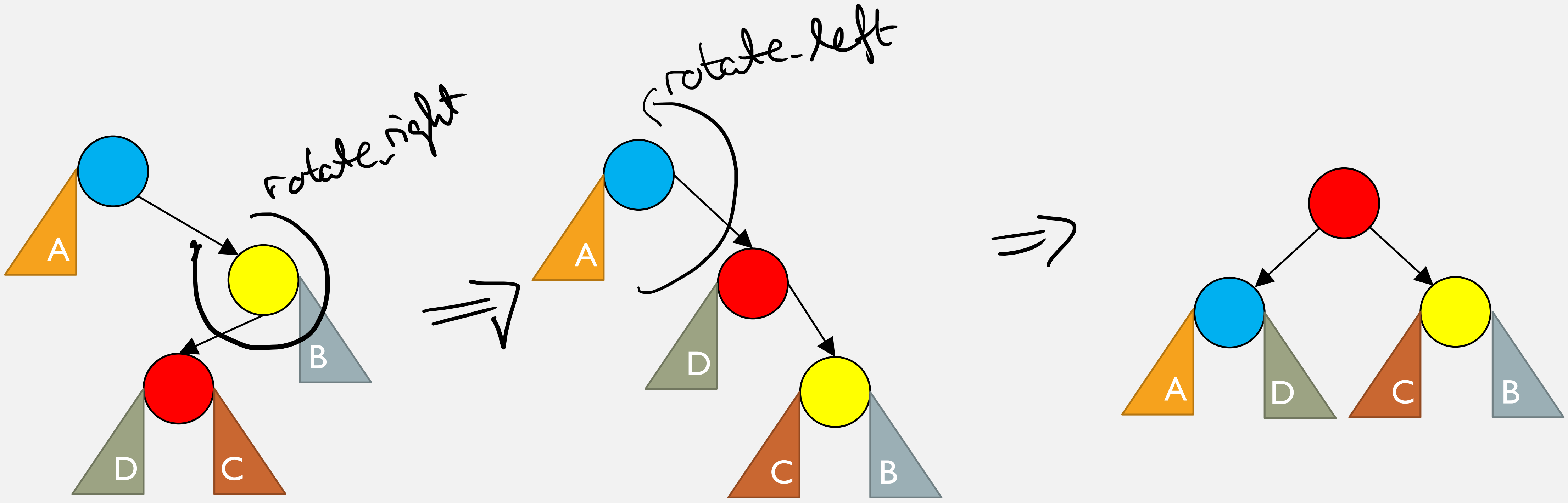
another way of thinking about it: pull up the middle one

AVL-BALANCING OF LEFT-RIGHT TREES



another way of thinking about it: pull up the middle one

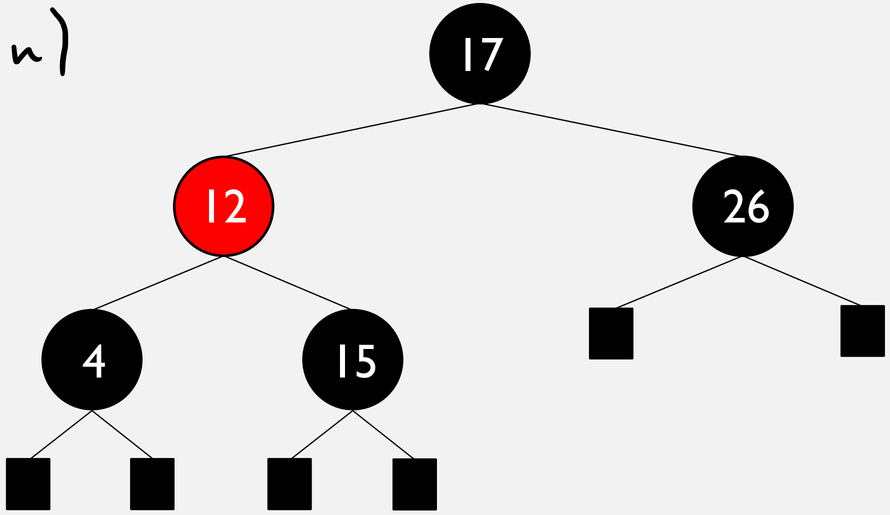
AVL-BALANCING OF RIGHT-LEFT TREES



another way of thinking about it: pull up the middle one

RED-BLACK TREES

Guarantees height $\leq 2\log(n+1) = O(\log n)$

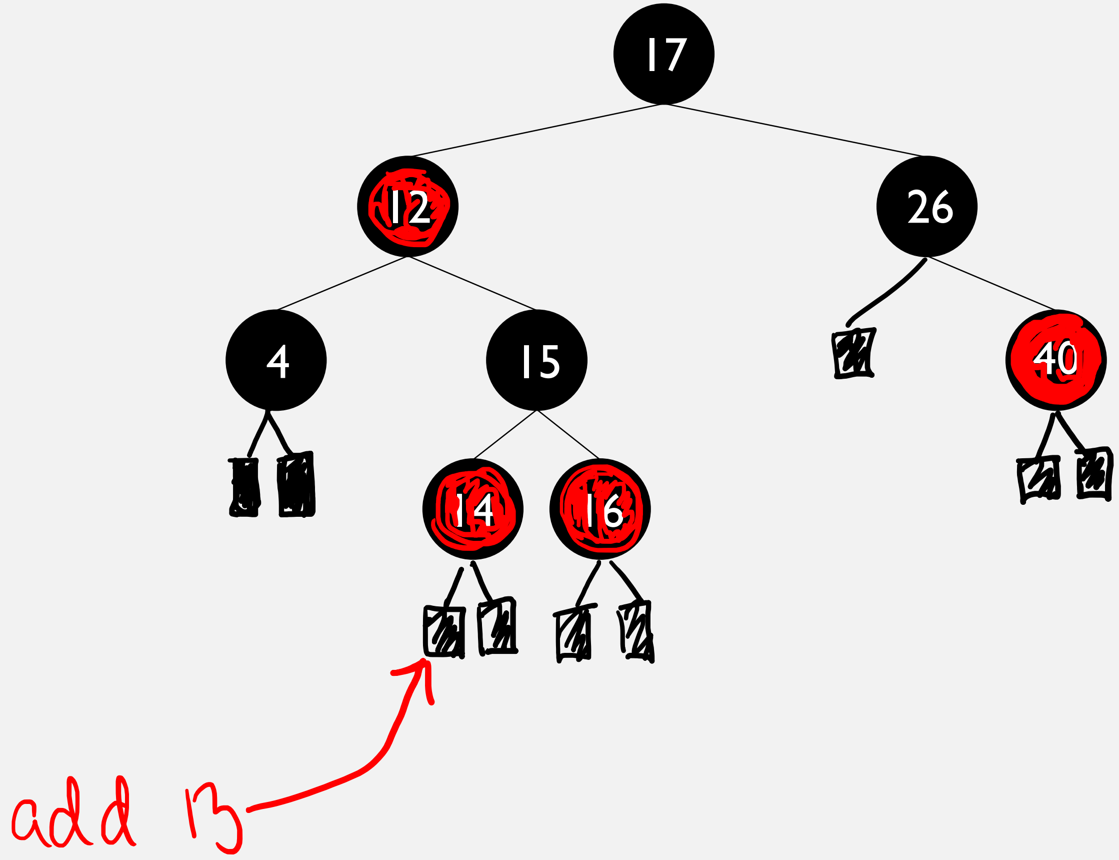


- Every node is either red or black
- The root is black
- Every red node has a black parent

null reference

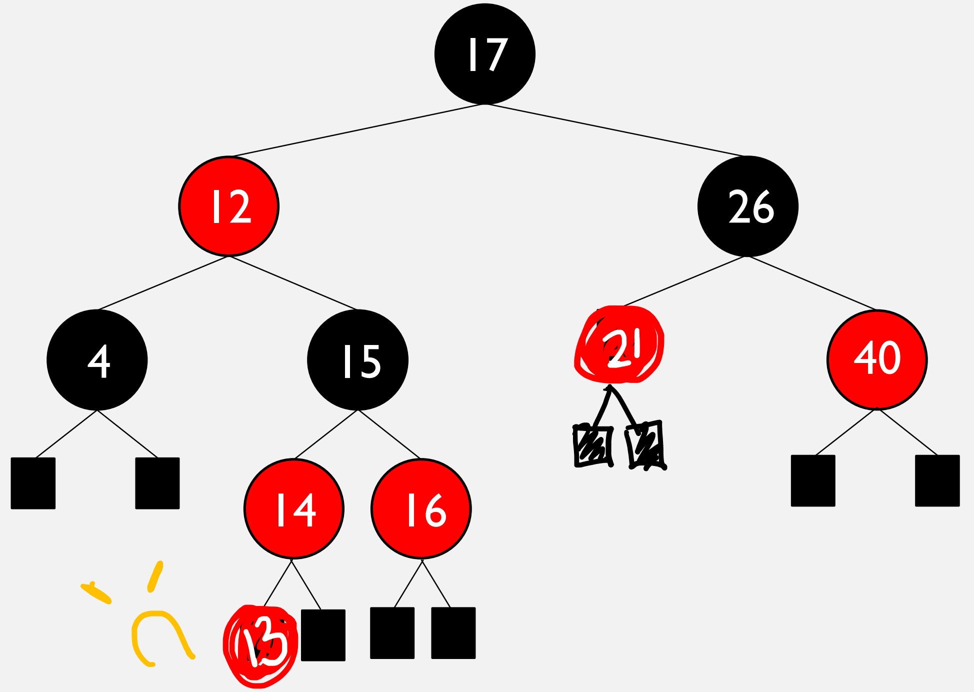
- All paths from the root to an external leaf has the same number of black nodes

A RED-BLACK TREE



INSERTION IN RED-BLACK TREES

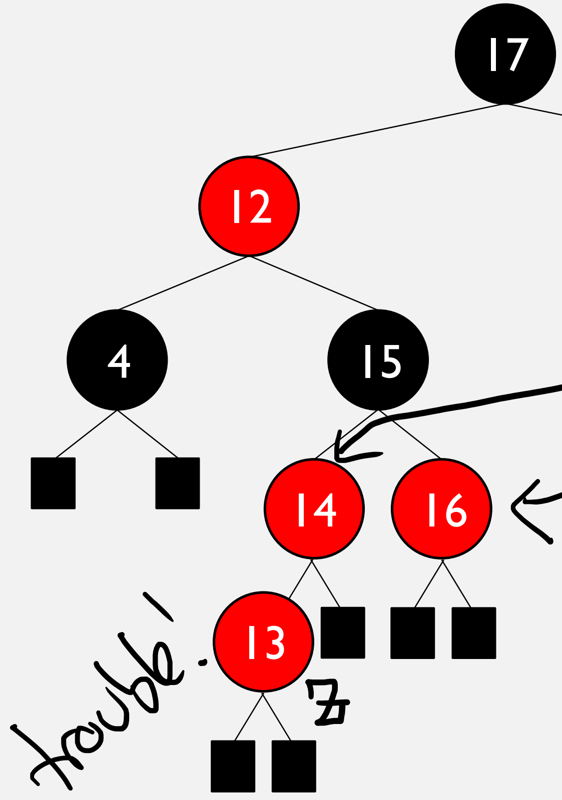
Whenever we insert, color the new node red.



insert 21

insert 13
→ trouble

INSERTION IN RED-BLACK TREES

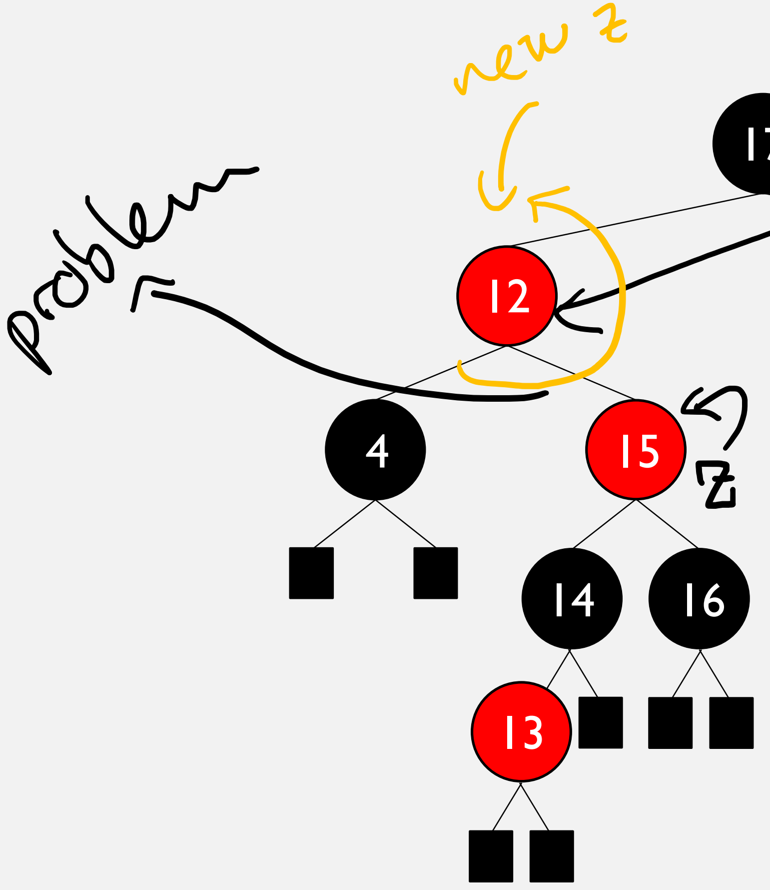


Case I

Parent is red
Uncle is red

Then
Color uncle + parent black
Color grandparent red
Set z to grandparent

INSERTION IN RED-BLACK TREES

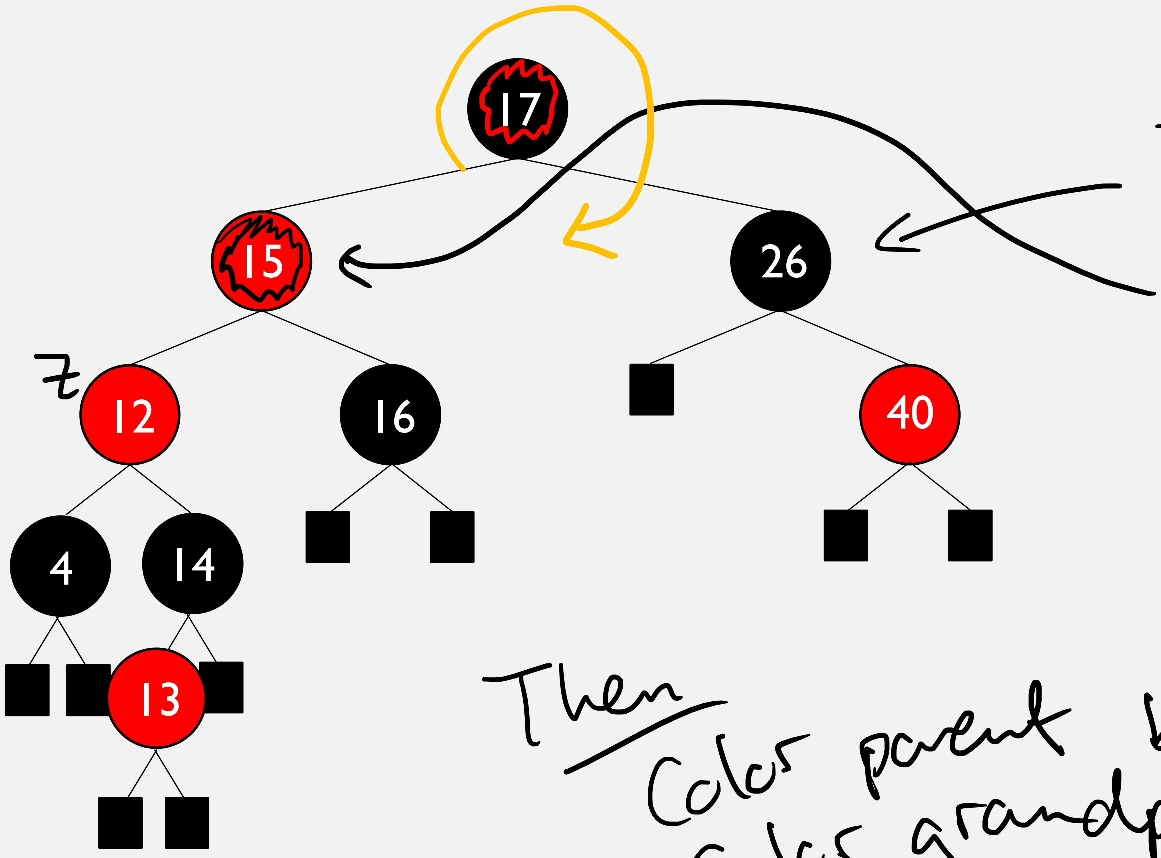


Case IIa

Parent red
Uncle black
z right child of parent

Then
Set z to parent
left rotate around z

INSERTION IN RED-BLACK TREES

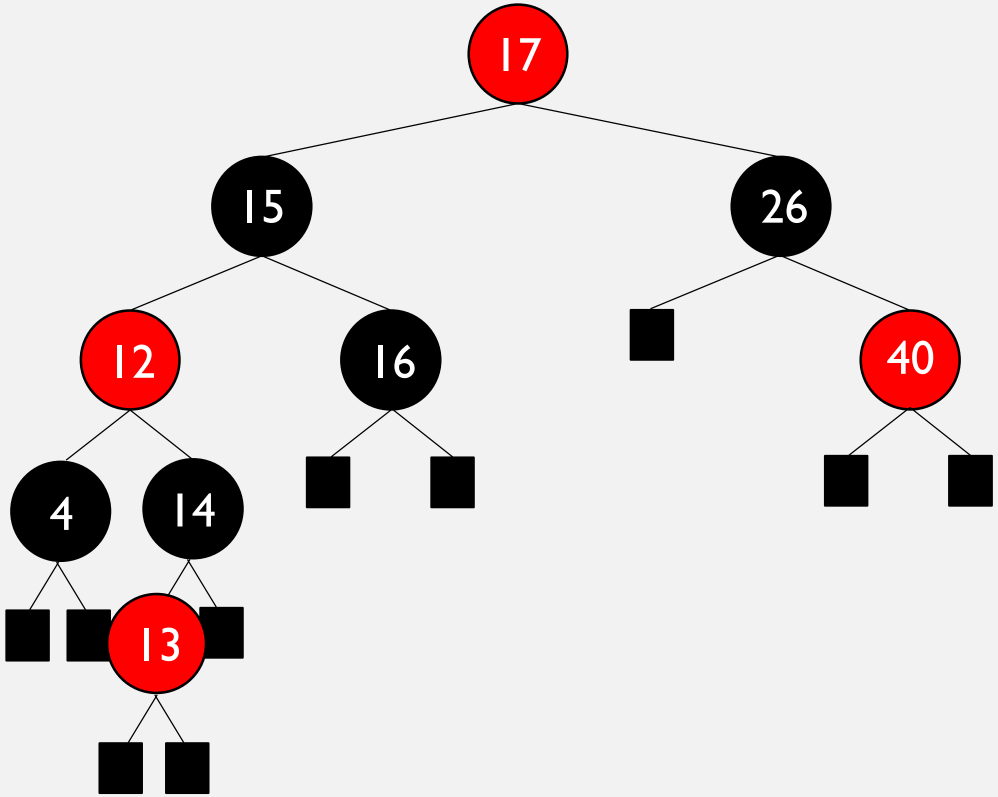


Case IIIa

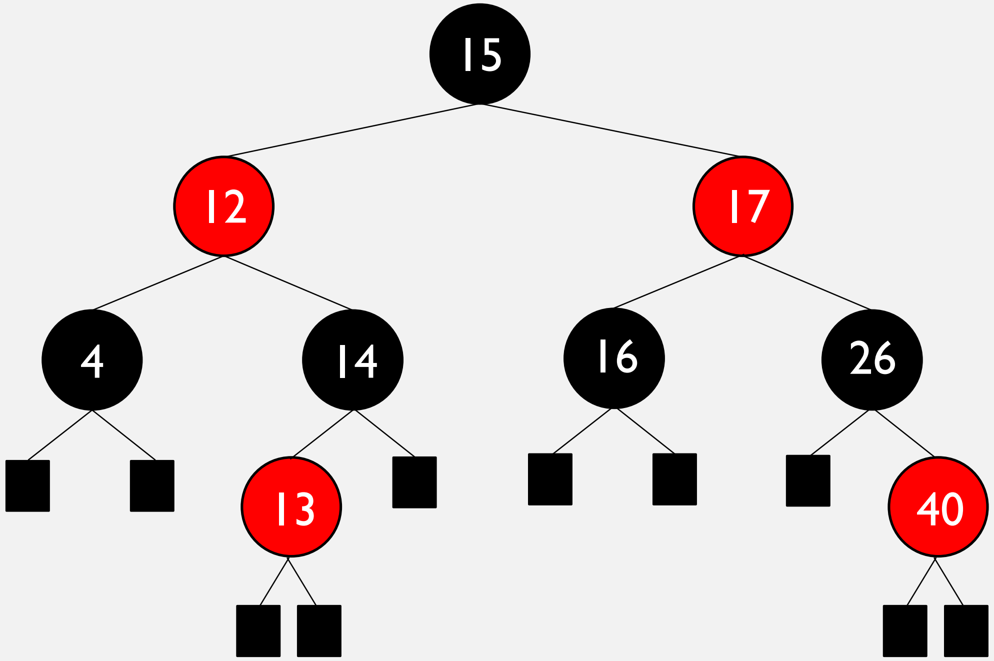
Uncle black
Parent red
z is left child of parent

Then
Color parent black
Color grandparent red
Right rotate around grandparent

INSERTION IN RED-BLACK TREES



INSERTION IN RED-BLACK TREES

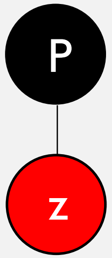


valid red-black tree

SIX CASES

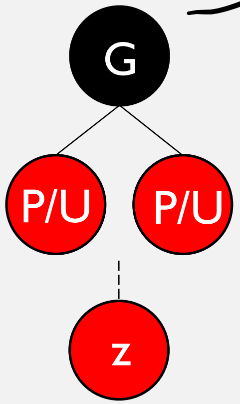
P=parent
U=uncle
G=grandparent

Case 0



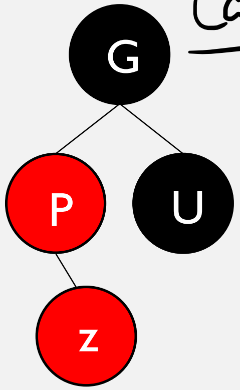
do nothing

Case I



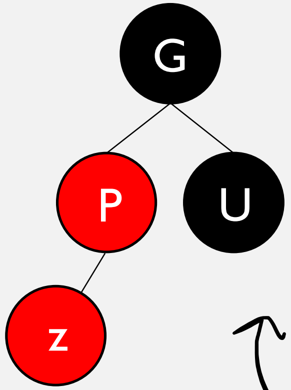
Color P+U black
Color G red
Set z to G

Case IIa



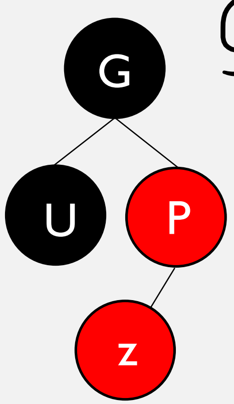
Set z to P
Left rotate
around z

Case IIIa



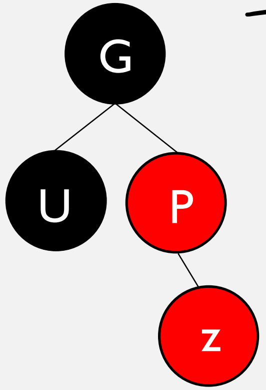
Color P black,
G red
Right rotate
around G

Case IIb



Set z to P
Right rotate
around z

Case IIIb



Color P black,
G red
Left rotate
around G

```
while z.p.color == RED:            // while we are in trouble
    if z.p == z.p.p left:          // "a" cases
        u = z.p.p.right
        if u.color == RED:
            z.p.color = BLACK      }
            u.color = BLACK        }  Case I
            z.p.p = RED            }
            z = z.p.p              }
        else:
            if z = z.p.right:      }
                z = z.p            }  Case II
                left_rotate(z)     }
            z.p.color = BLACK      }
            z.p.p.color = RED      }  Case III
            right_rotate(z.p.p)    }
    else:
        as above but left <-> right   // "b" cases
root.color = BLACK
```

DELETION

Can be done, but is quite complex

AVL TREES VS RED-BLACK TREES

	Red-black	AVL
Look-up	Slower	Faster
Insert	Faster	Slower
	Roughly balanced	Strictly balanced