

dynamic BALANCED SEARCH TREES (NON-RANDOM)

Objectives: search, insert, delete in $O(\log n)$ time

↳ always maintain $\Theta(\log n)$ height & update in $O(\log n)$ time

Variations: AVL, 2-3, 2-3-4, B-trees, red-black, skip lists, treaps
today

RED-BLACK trees

Structure: 1) nodes are colored red or black.

2) root is always black.

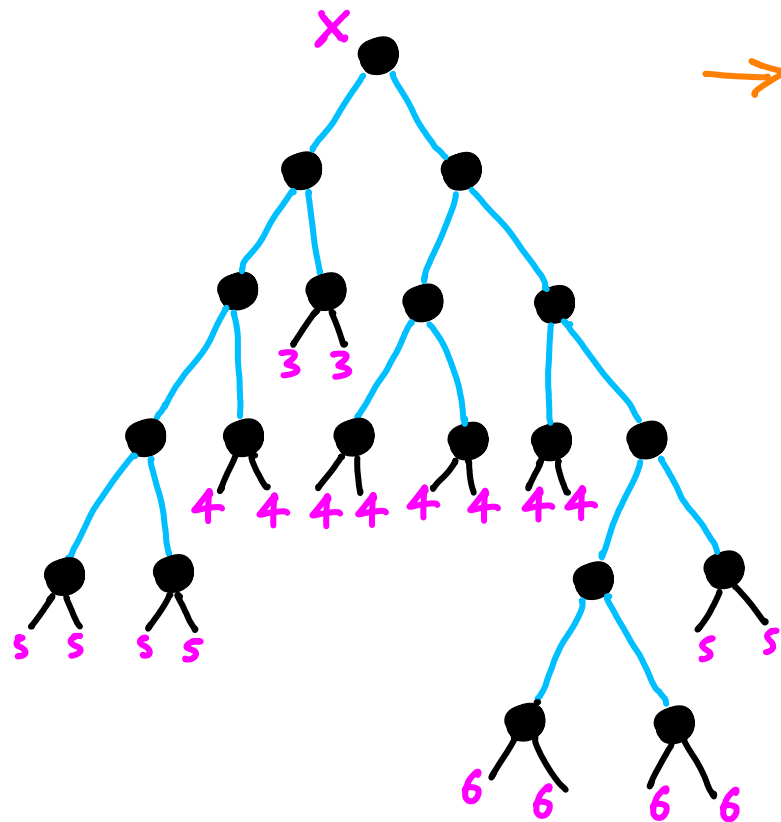
3) add black "dummy" leaves so every "real" node has 2 children.

the important rules { 4) every red node has a black parent.

5) for any node x : all paths down to leaves contain equal number of black nodes = $\underbrace{\text{black-height}[x]}_{\text{not including } x}$

4) every red node has a black parent.

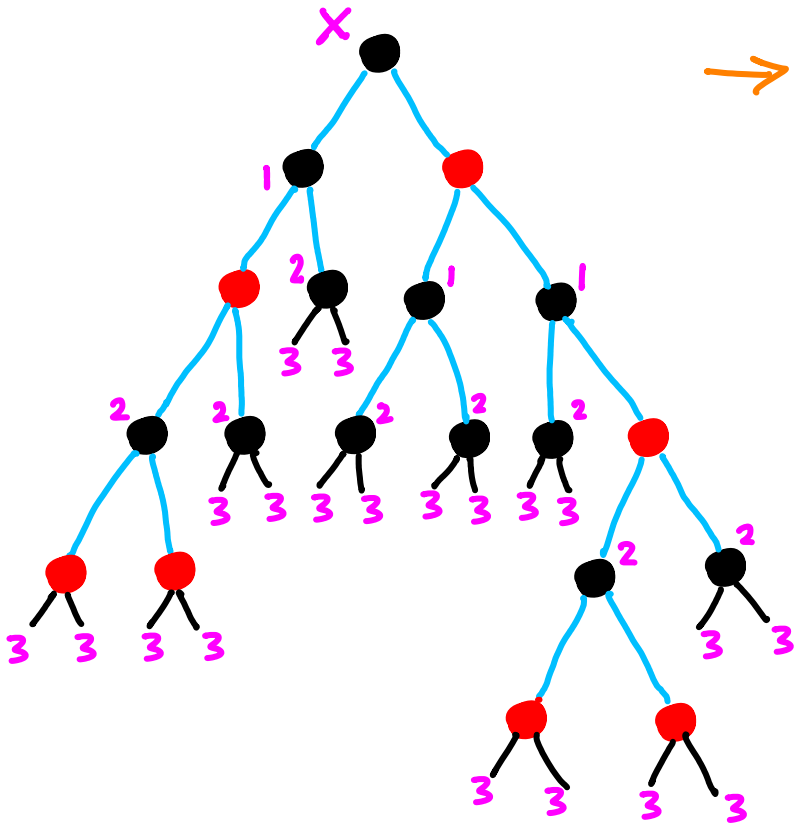
5) for any node x : all paths down to leaves contain equal number of black nodes = $\text{black-height}[x]$



→ Fails rule 5

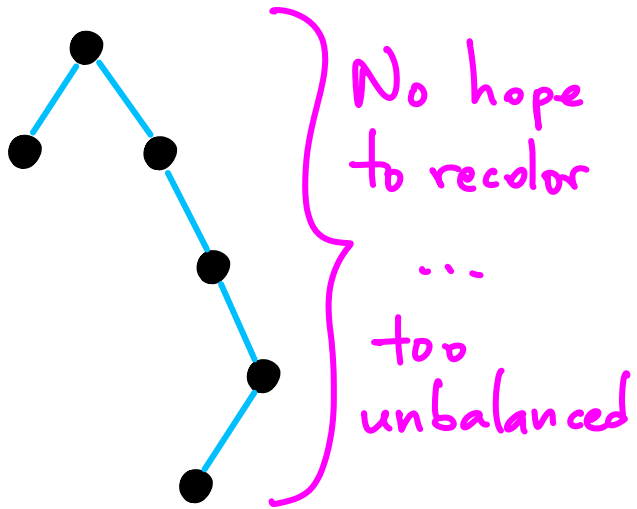
4) every red node has a black parent.

5) for any node x : all paths down to leaves contain equal number of black nodes = $\text{black-height}[x]$



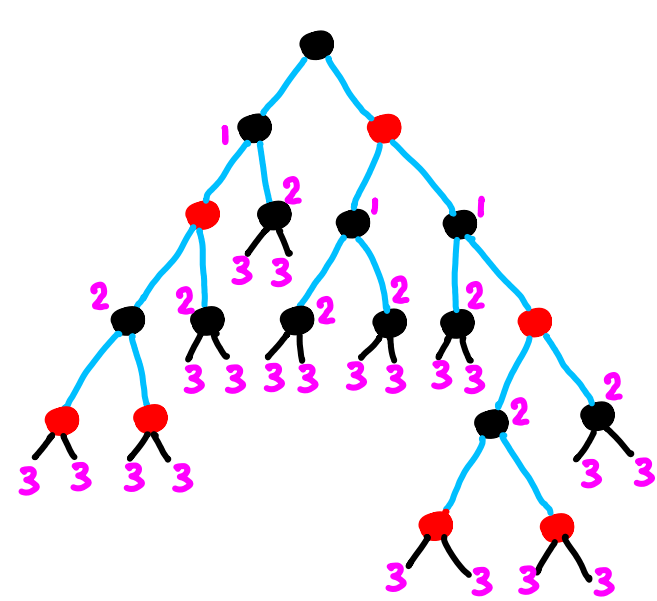
→ Fails rule 5 ⇒ fix by making some nodes red.

black-height difference : 6 vs 3

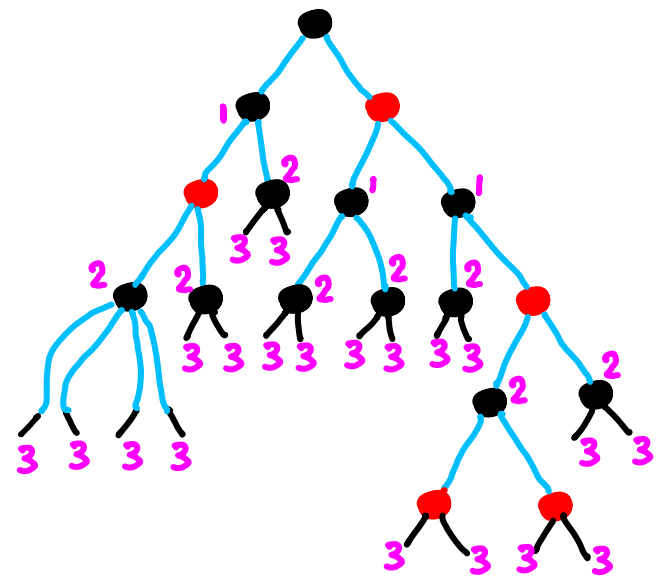


Any root \rightarrow leaf path of size k must have $\geq \frac{k}{2}$ black nodes. } Rule 4

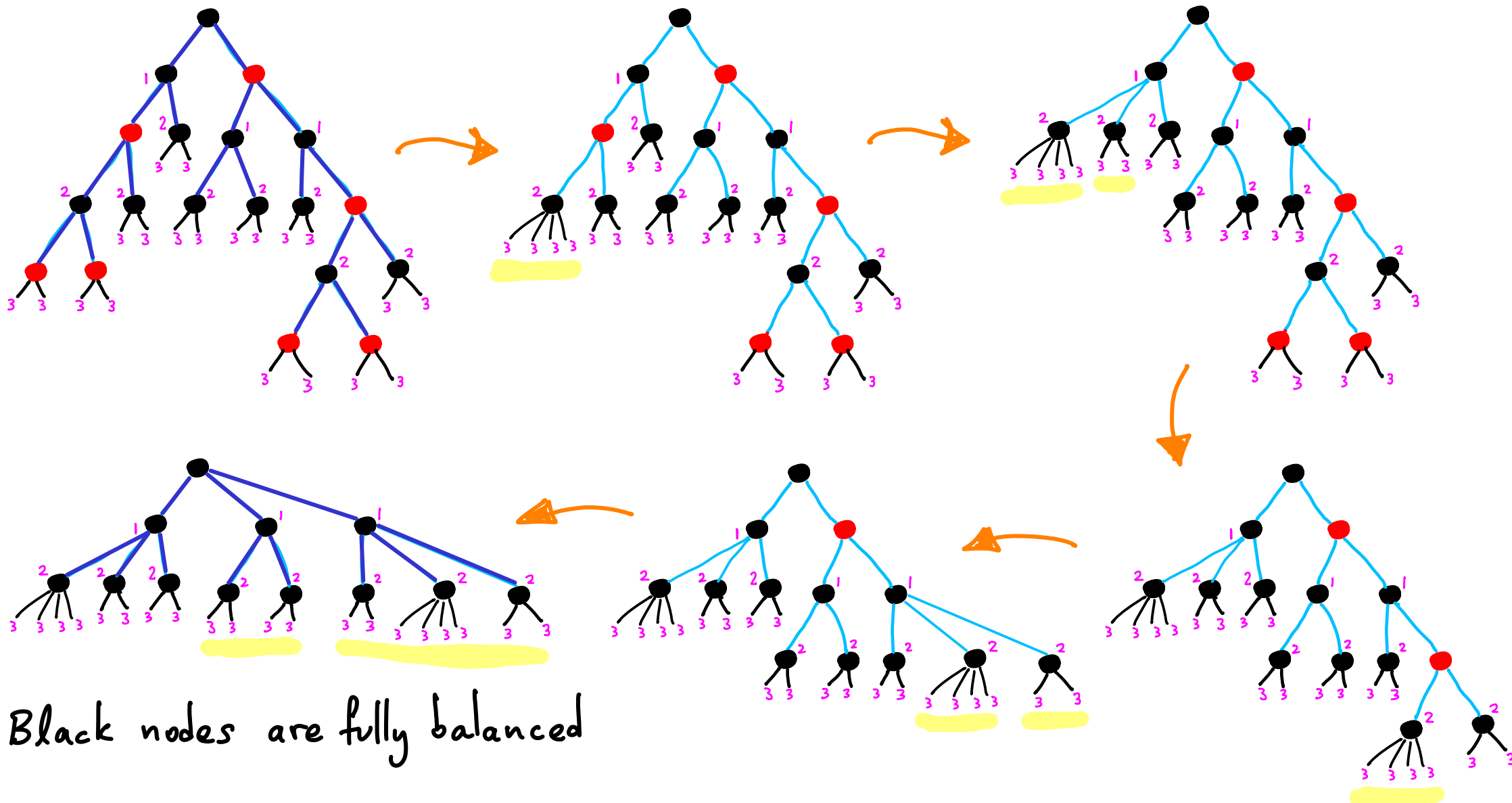
So if any path is >2 times longer than another, we can't make it RB.



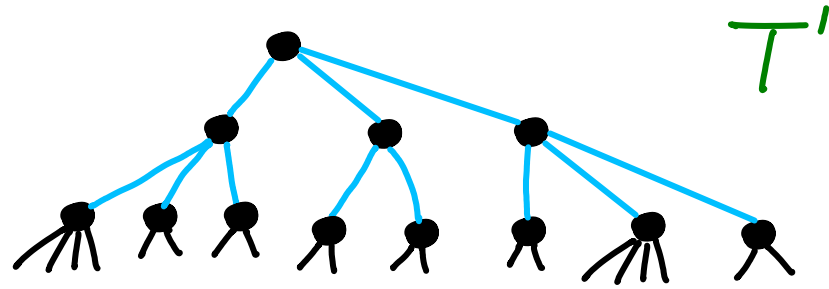
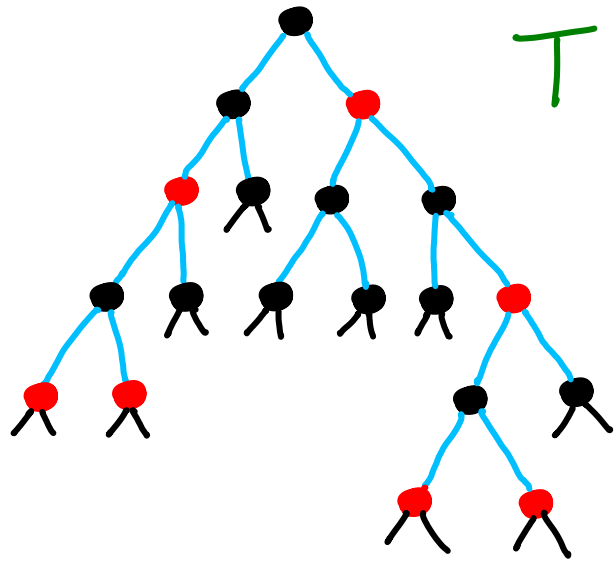
==



CONTRACTION



Black nodes are fully balanced



If $\text{root}(T)$ has same black-height on all paths
then $\text{height}(T')$ is perfectly balanced

Q: what is max.degree of T' ? (2-3-4)

#leaves in T : $n+1 = \text{size}(T)+1$

#leaves in T' : same

$\text{height}(T') \leq \log(n+1)$ [higher degree \Rightarrow smaller height ; worst-case: binary]

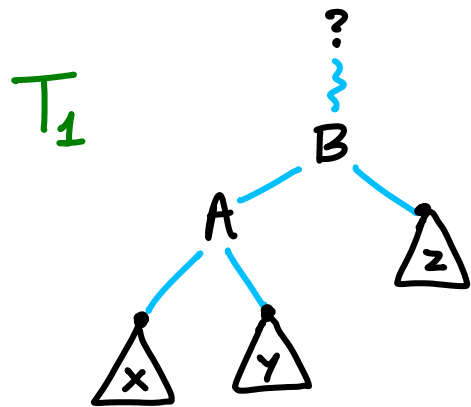
Re-inserting red nodes : at most doubles height $\rightarrow \text{height}(T) \leq 2 \log(n+1)$

We have seen that RB trees are reasonably balanced : $\sim 2 \log n$

↳ search, min, max, next, prev : $O(\log n)$ time.

Next: how to update RB trees (insert, delete)

ROTATIONS in arbitrary BSTs

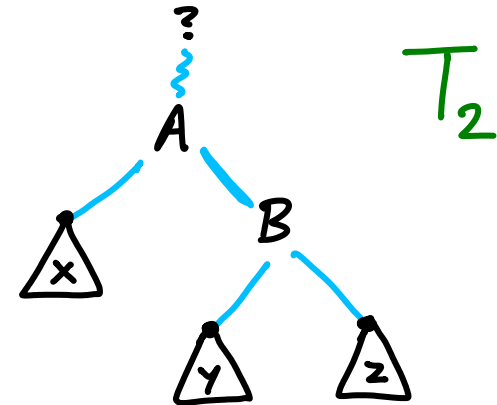


right-rotate(T_1, B)



left-rotate(T_2, A)

$O(1)$ time




$x \leq A \leq y \leq B \leq z$

$x \leq A \leq y \leq B \leq z$

INSERT IN RED BLACK TREES

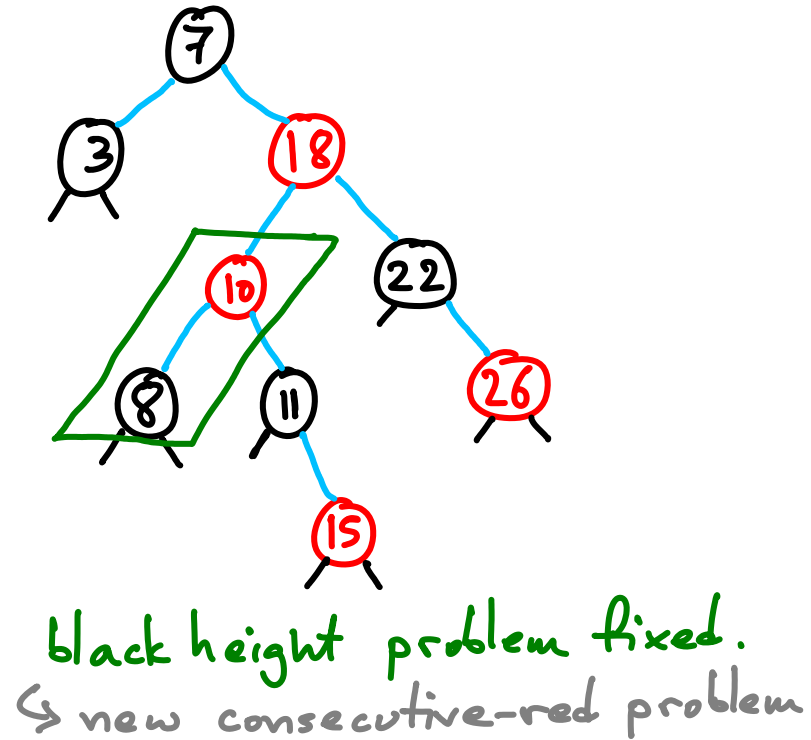
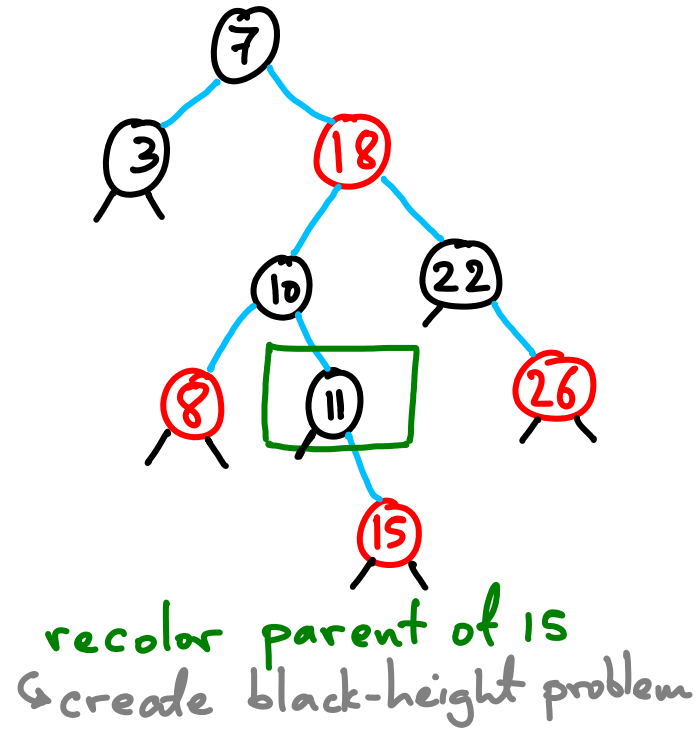
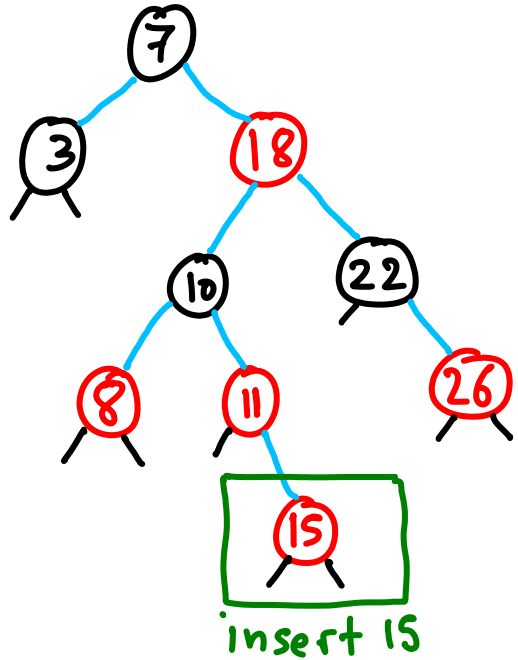
greedy (optimistic) start : $\left\{ \begin{array}{l} 1) \text{ insert as in any BST} \\ 2) \text{ color new node red} \quad (\text{keeps black-height ok}) \\ 3) \text{ if parent is also red} \quad (\text{violate parent rule 4}) \end{array} \right.$

begins an error-correcting trail up to root,  look for problems further up

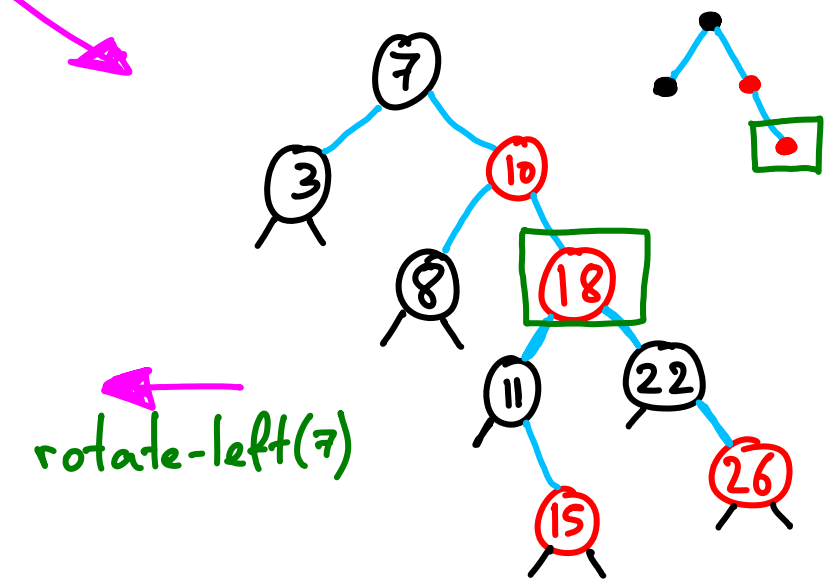
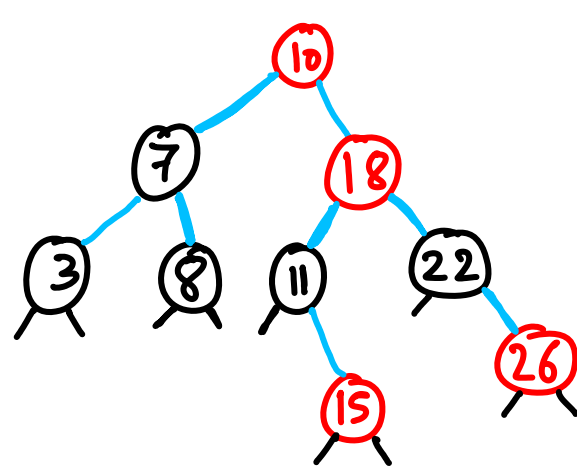
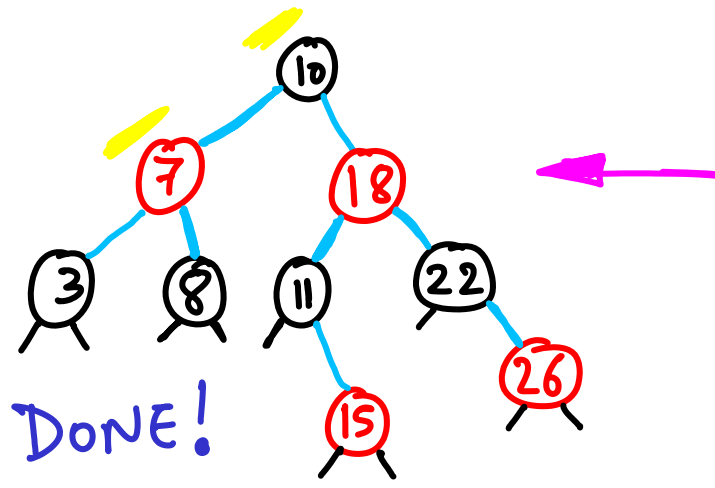
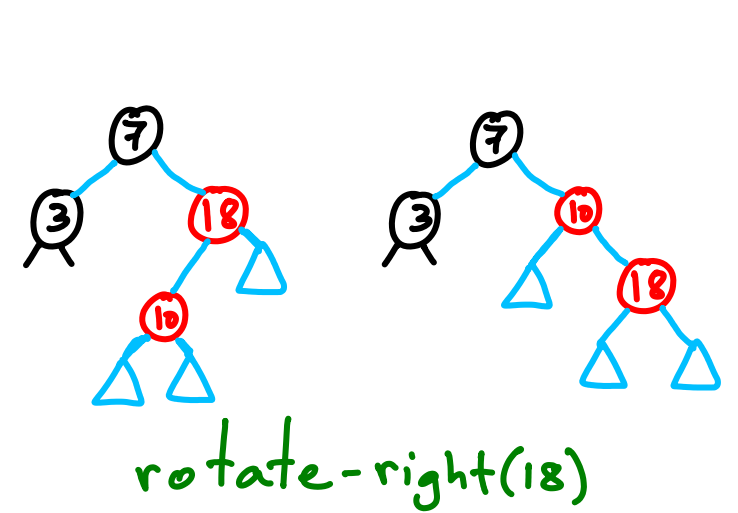
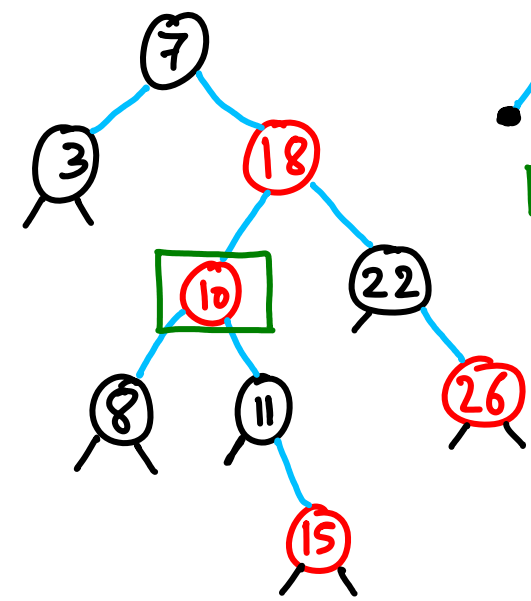
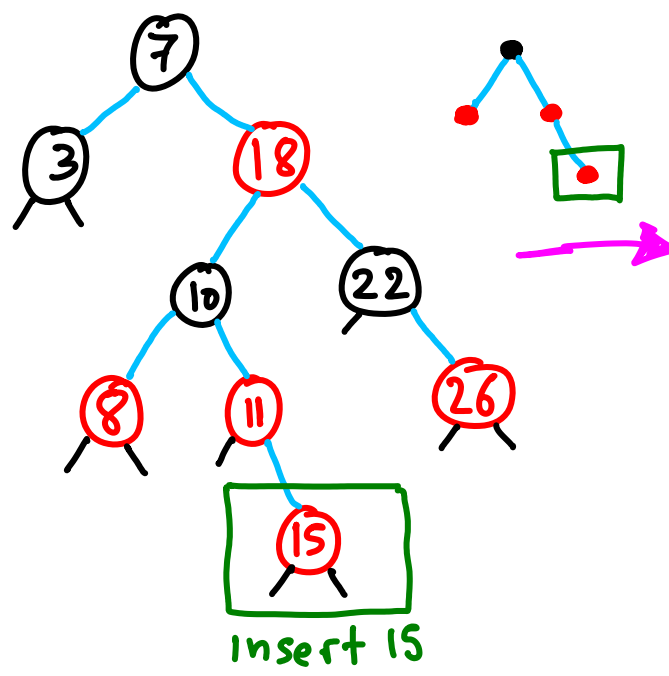
involving $O(1)$ recolorings
and rotations per level

$\rightarrow O(\log n)$ time

example of $O(1)$ -time error-corrections (per level)



problem transferred from new node to its grandparent



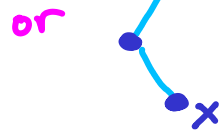
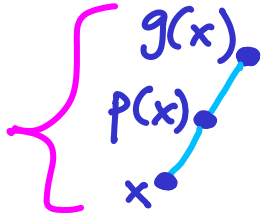
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

RB-insert(x)

- initial step: regular insert & color x red.

- while $x \neq \text{root}$ and $p(x)$ is red

if $p(x) = \text{left}(g(x))$



} don't care about colors

→ why does $g(x)$ exist?

$p(x) = \text{red}$ so it isn't the root.

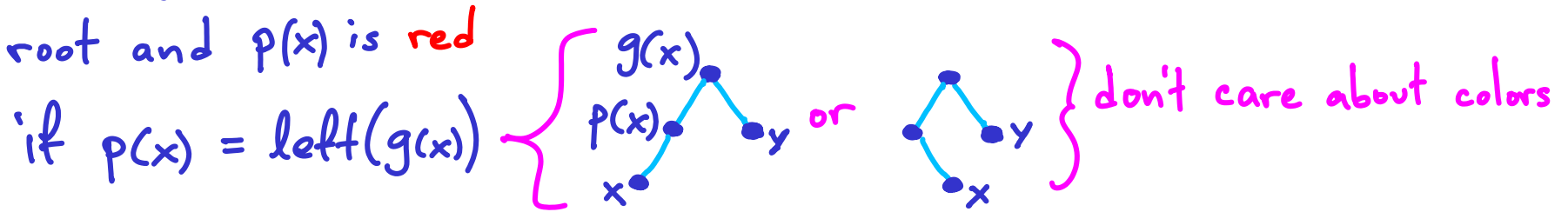
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

RB-insert(x)

- initial step: regular insert & color x red.

- while $x \neq \text{root}$ and $p(x)$ is red

if $p(x) = \text{left}(g(x))$



$y \leftarrow \text{right}(g(x))$

→ why can we assume y exists?

It is at least a dummy leaf.

in which case the subtree of $g(x)$

is  before inserting x.

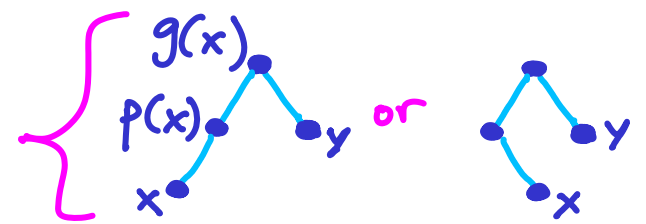
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

RB-insert(x)

- initial step: regular insert & color x red.

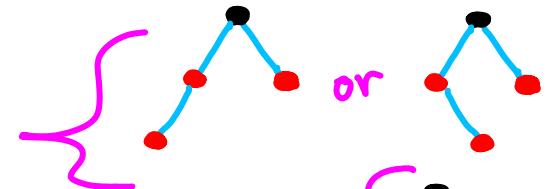
- while $x \neq \text{root}$ and $p(x)$ is red

if $p(x) = \text{left}(g(x))$

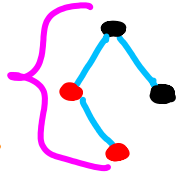


$y \leftarrow \text{right}(g(x))$

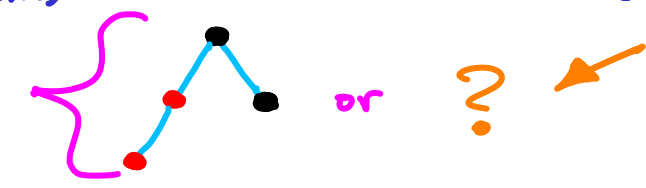
if y is red then run CASE 1



else if $x = \text{right}(p(x))$ then run CASE 2.



Run CASE 3



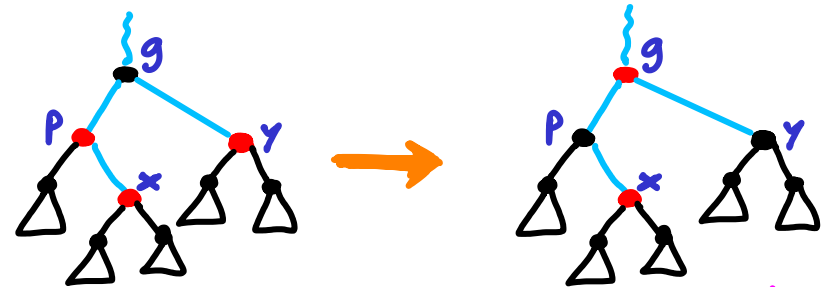
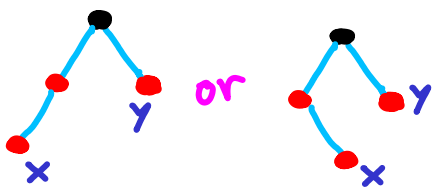
else $\parallel p(x) = \text{right}(g(x))$

do as before but with "left" & "right" switched \parallel symmetric

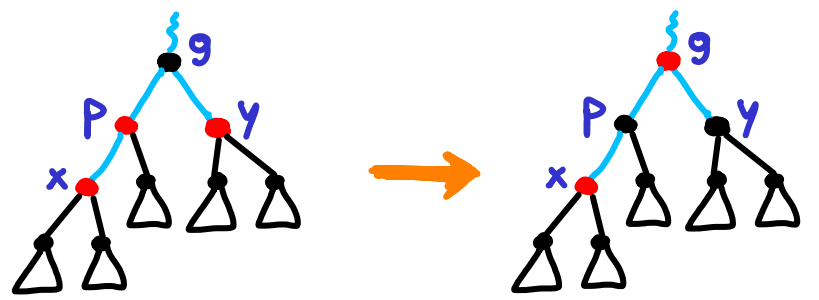
color the root black

these will
move
x up
(redefine)
(and it will
be red)

if y is red then run CASE 1

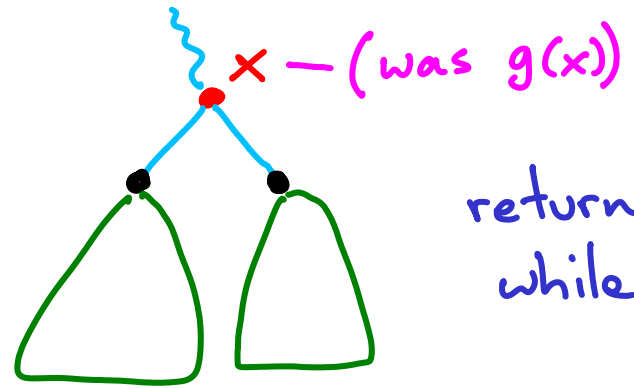


Every \triangle contributes same to black-height.



Recolor $y, p(x), g(x)$
Let $x \leftarrow g(x)$

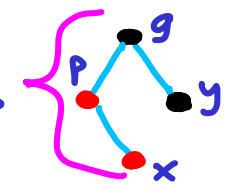
blackheight($p(x)$) preserved



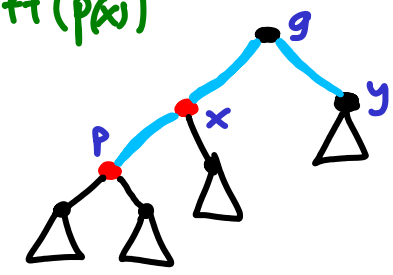
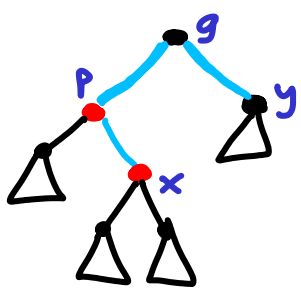
return to while loop.

(repeat if new $p(x)$ is red)

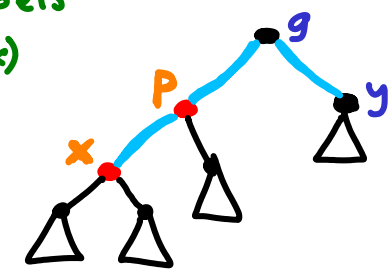
else if $x = \text{right}(p(x))$ then run CASE 2.



rotate-left($p(x)$)

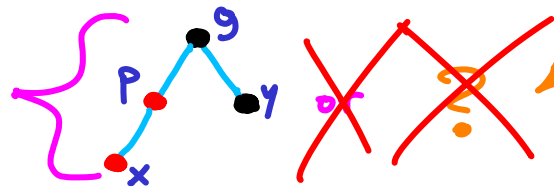


switch labels
 $x \leftrightarrow p(x)$

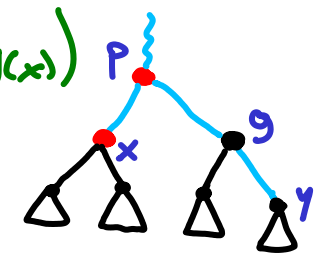
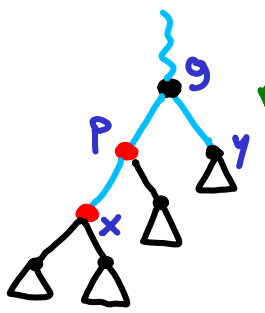


conditions of case 3

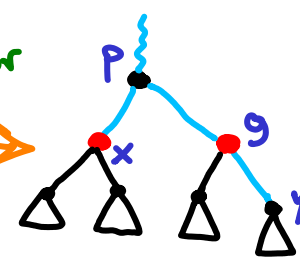
Run CASE 3



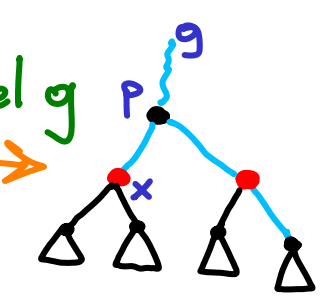
rotate-right($g(x)$)



re-color
 p, g



relabel g



x moves up in tree
Now $p(x)$ is black.
Exit while

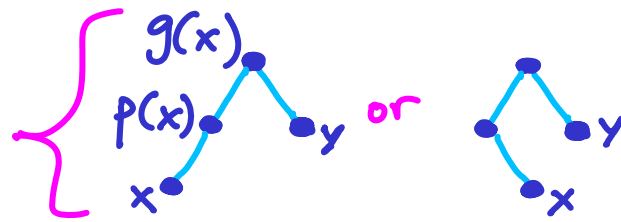
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations;

RB-insert(x)

- initial step: regular insert & color x red.

- while $x \neq \text{root}$ and $p(x)$ is red

if $p(x) = \text{left}(g(x))$

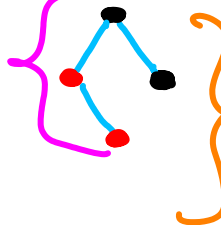
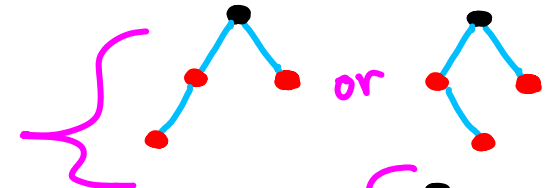
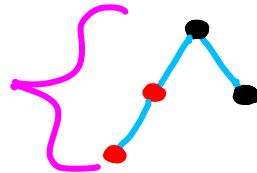


$y \leftarrow \text{right}(g(x))$

if y is red then run CASE 1

else if $x = \text{right}(p(x))$ then run CASE 2.

Run CASE 3



these will move x up (redefine) (and it will be red)

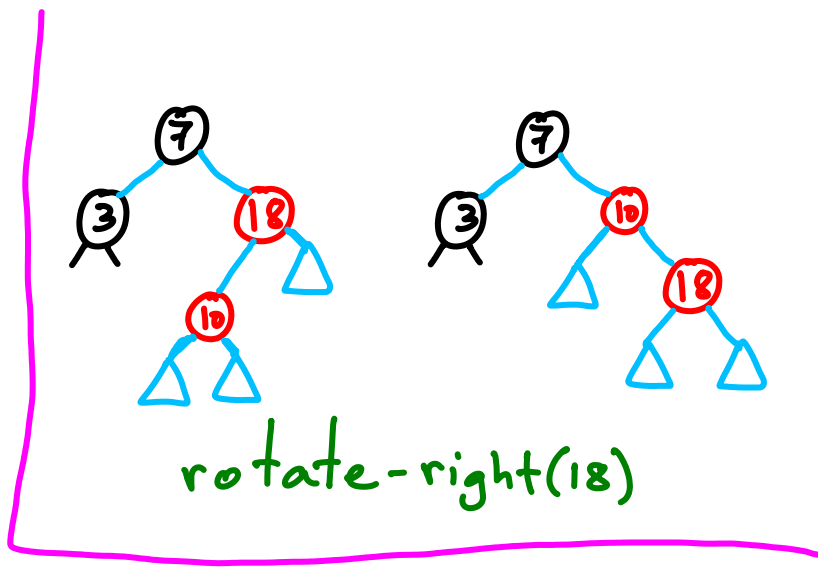
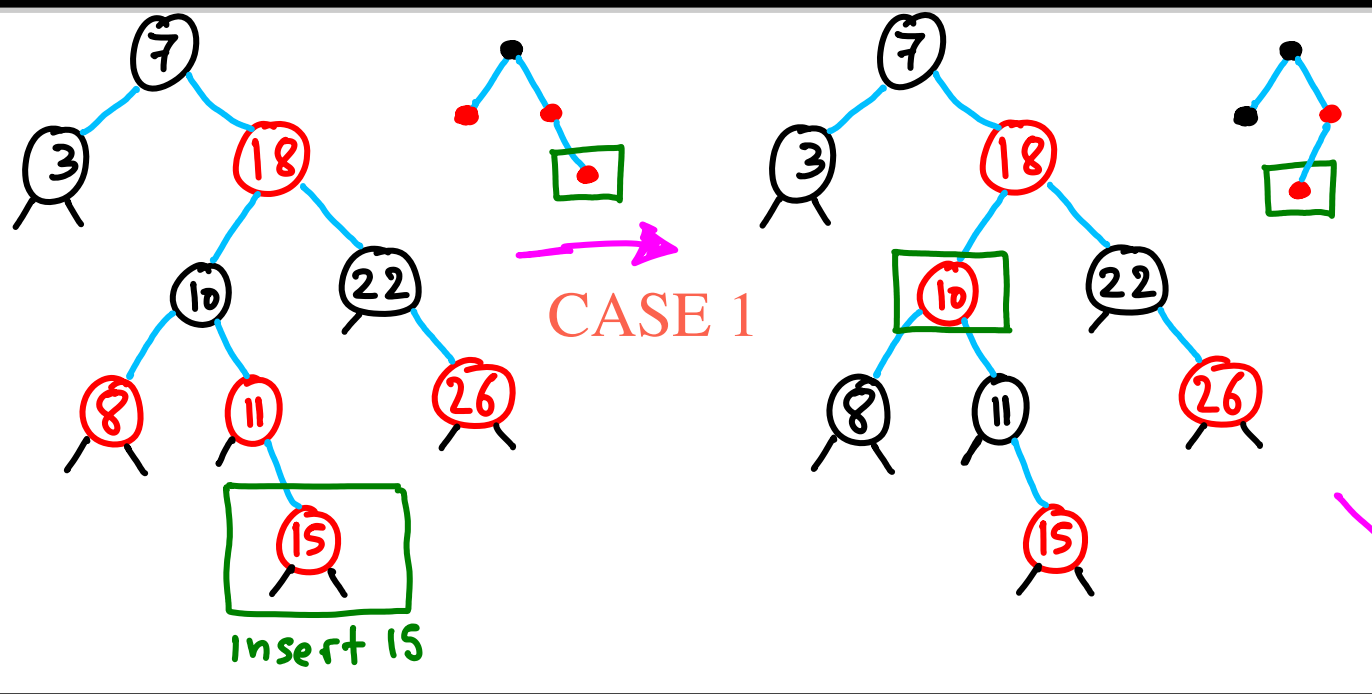
recolor, repeat WHILE

rotate, exit WHILE

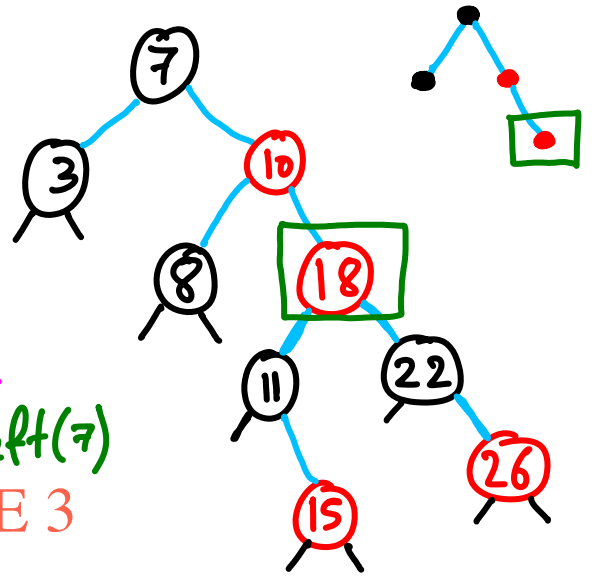
else $\parallel p(x) = \text{right}(g(x))$

do as before but with "left" & "right" switched \parallel symmetric

color the root black



CASE 2



CASE 3

