dynamic BALANCED SEARCH TREES (NON-RANDOM)

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

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

RED-BLACK trees

Structure: 1) nodes are colored red or black.

RED-BLACK trees

Structure:

1) nodes are colored red or black.
2) root is always black.

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.

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.
4) every red node has a black parent.

RED-BLACK trees

Structure:

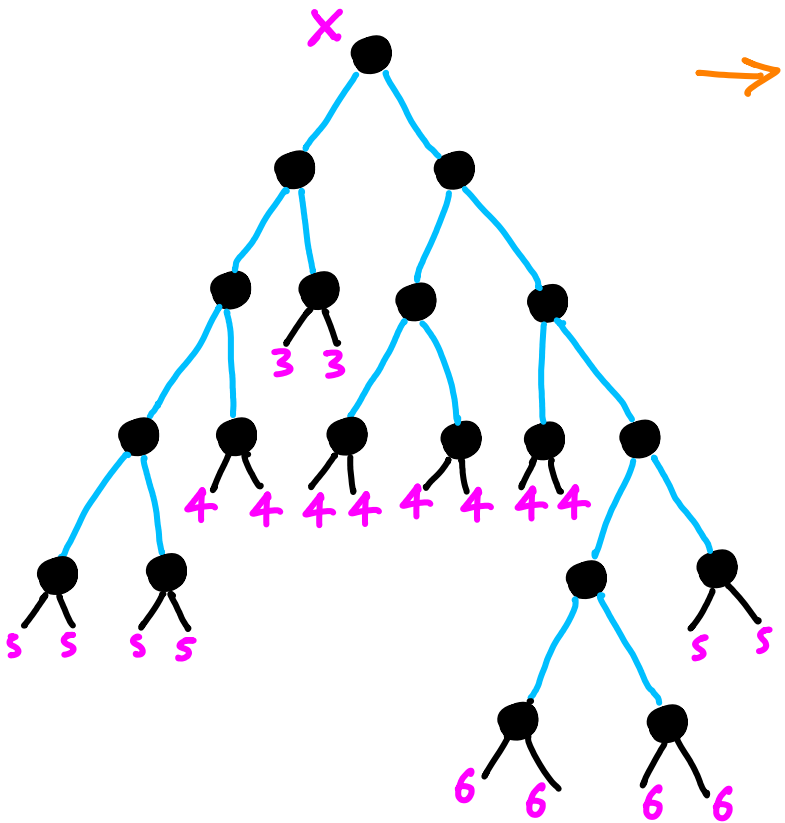
1) nodes are colored red or black.
2) root is always black.
3) add black "dummy" leaves so every "real" node has 2 children.

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 = black-height[x] (not including x)

4) every red node has a black parent.

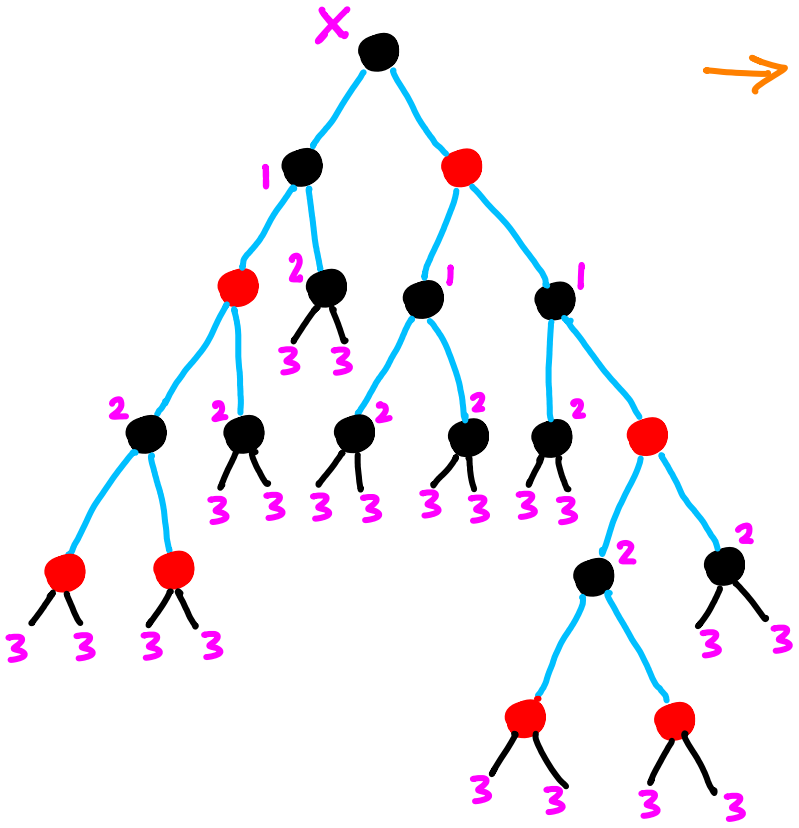
5) for any node x : all paths down to leaves contain equal number of black nodes = black-height[x]



→ fails rule 5

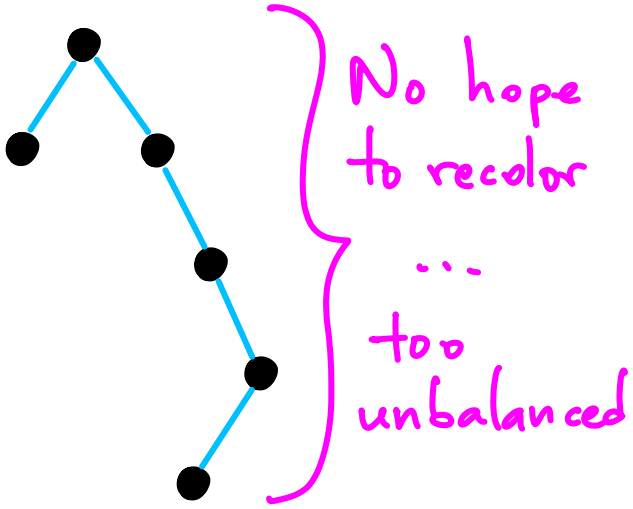
4) every red node has a black parent.

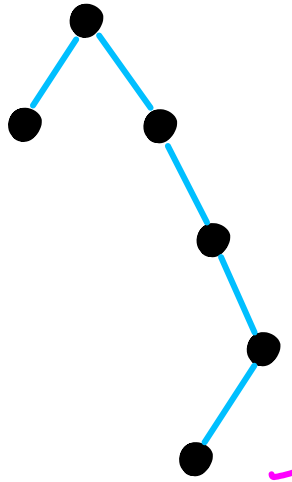
5) for any node x : all paths down to leaves contain equal number of black nodes = black-height[x]



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

black-height difference : 6 vs 3


No hope
to recolor
...
too
unbalanced

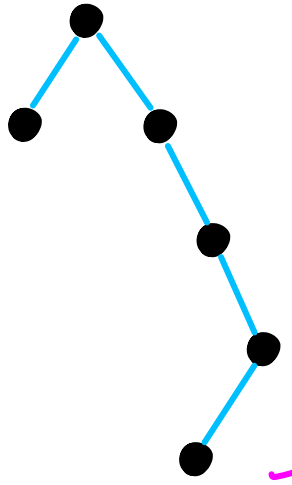


No hope to recolor

...

too unbalanced

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



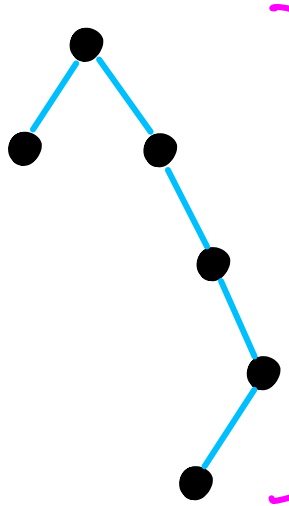
No hope to recolor

...

too unbalanced

Any root→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.



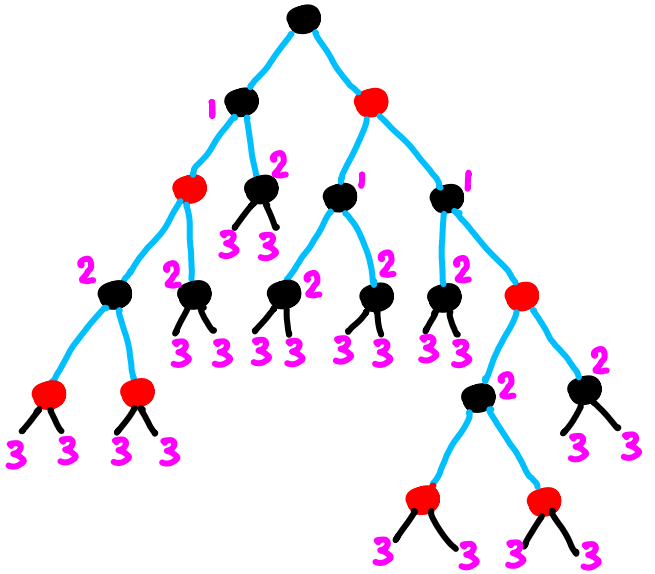
No hope to recolor

...

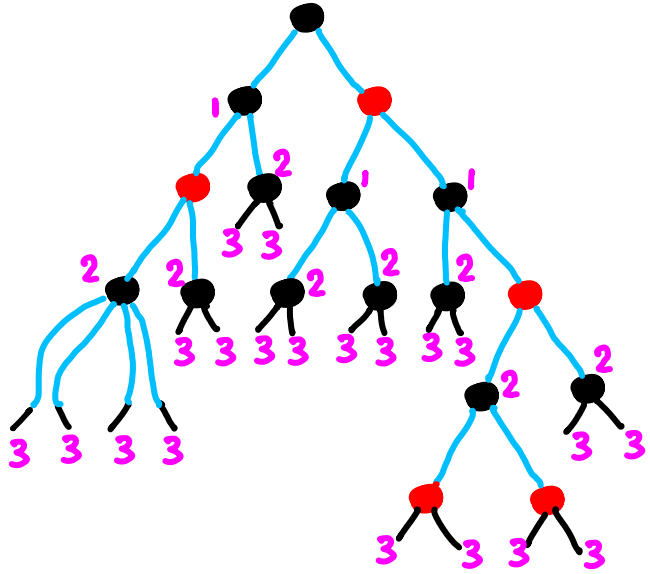
too unbalanced

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

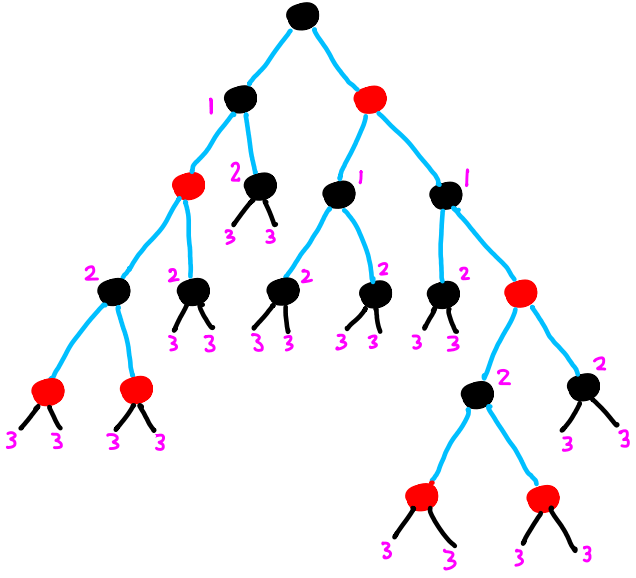
So if any path is >2 times longer than another, we can't make it RB.

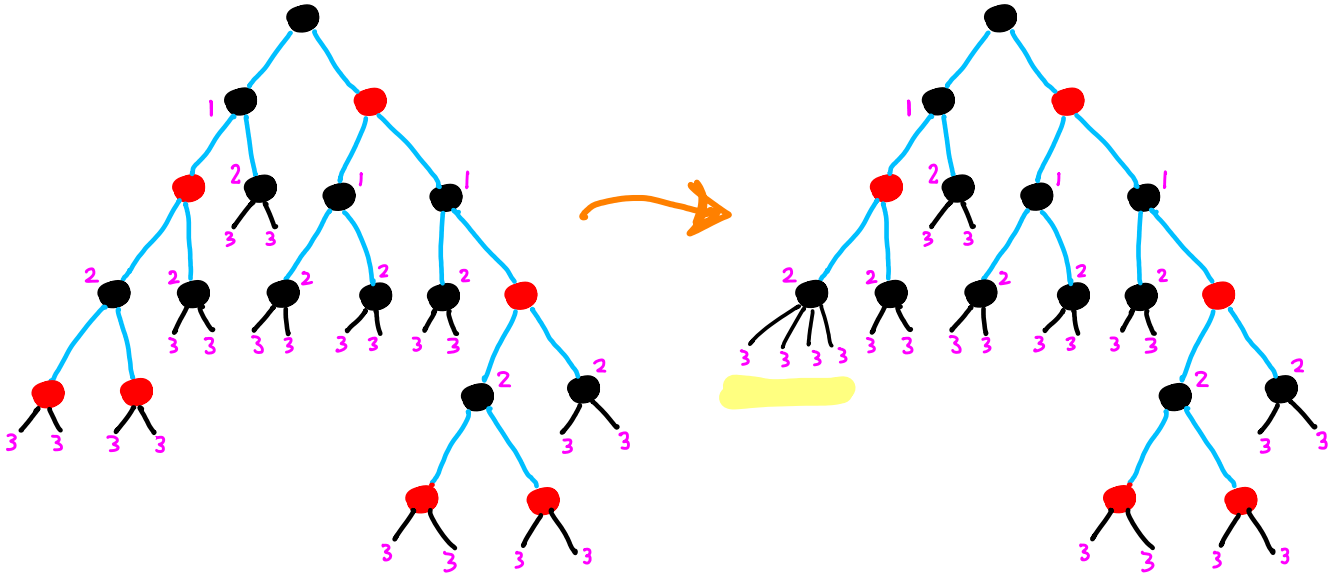


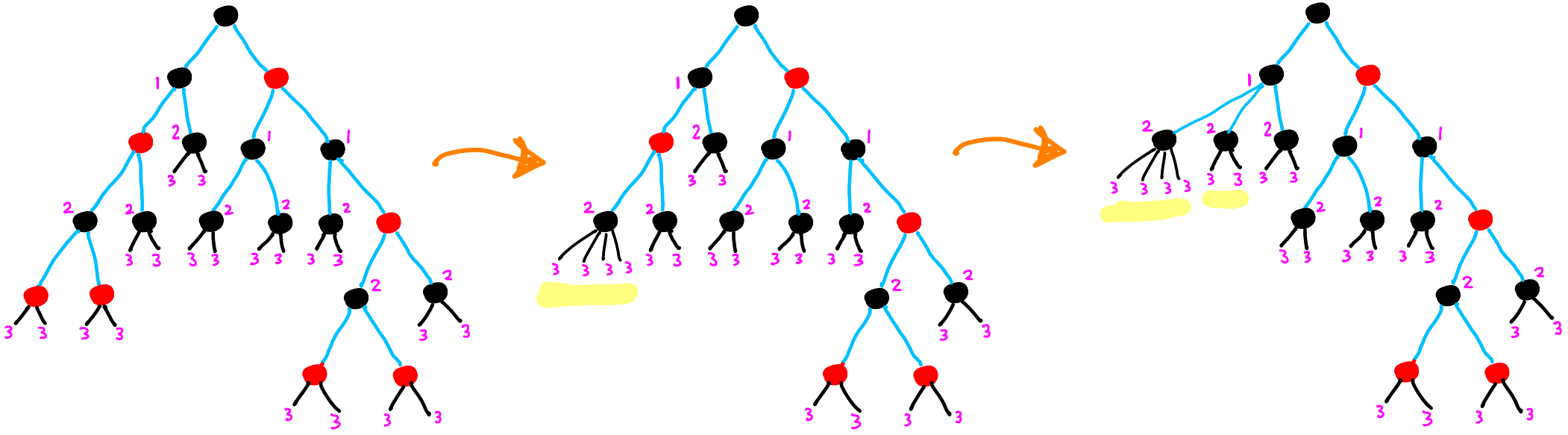
=

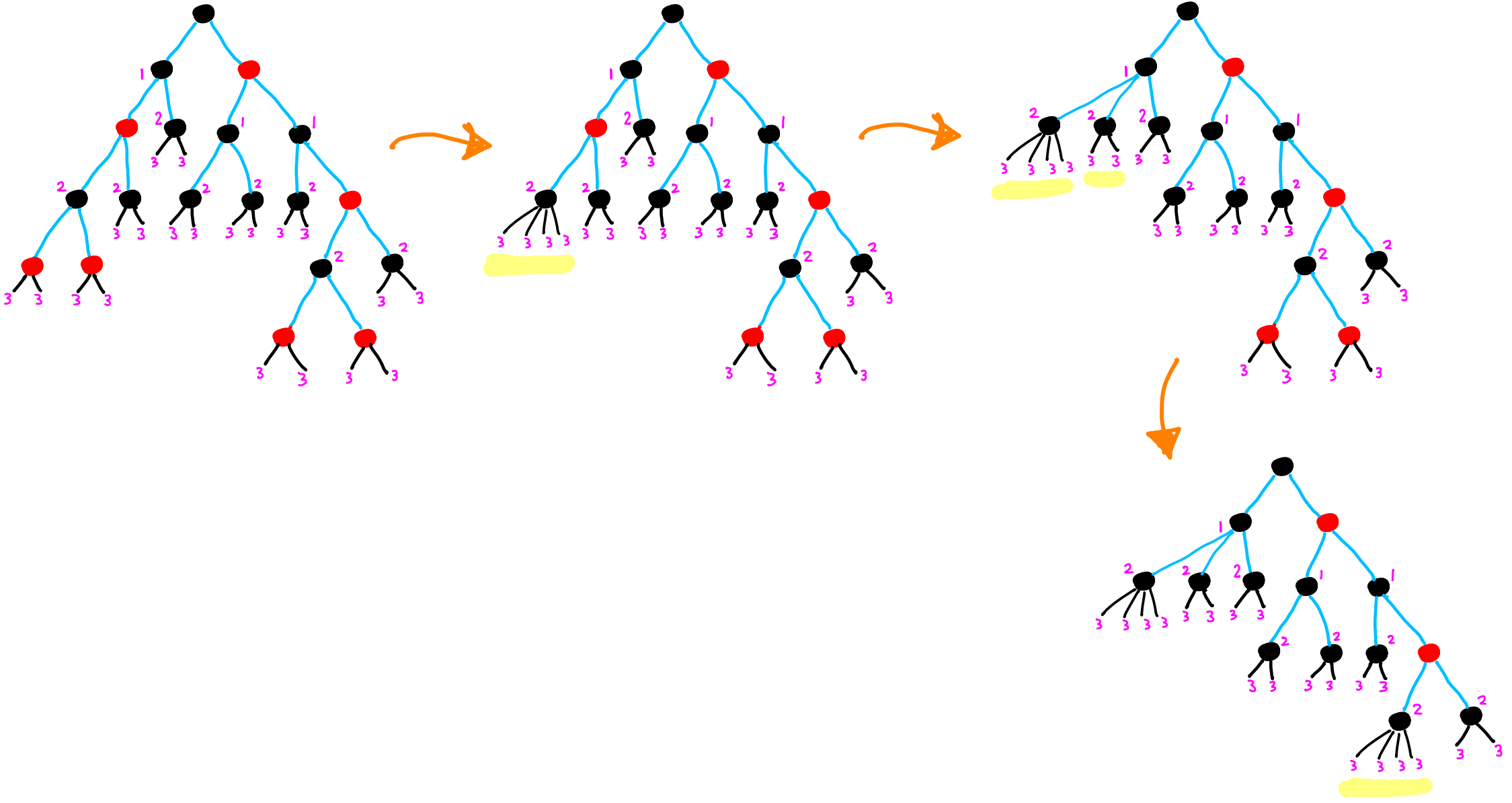


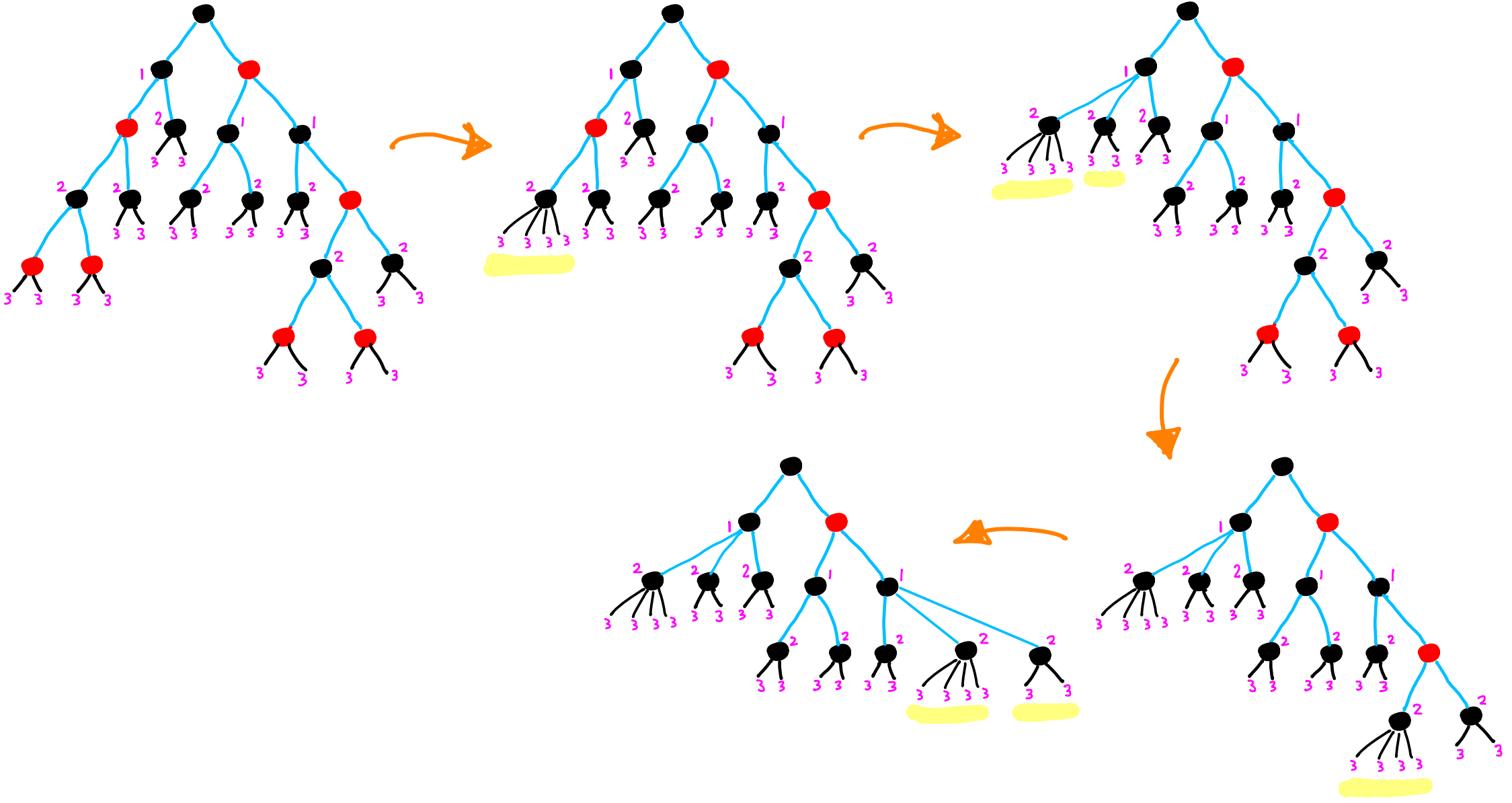
CONTRACTION

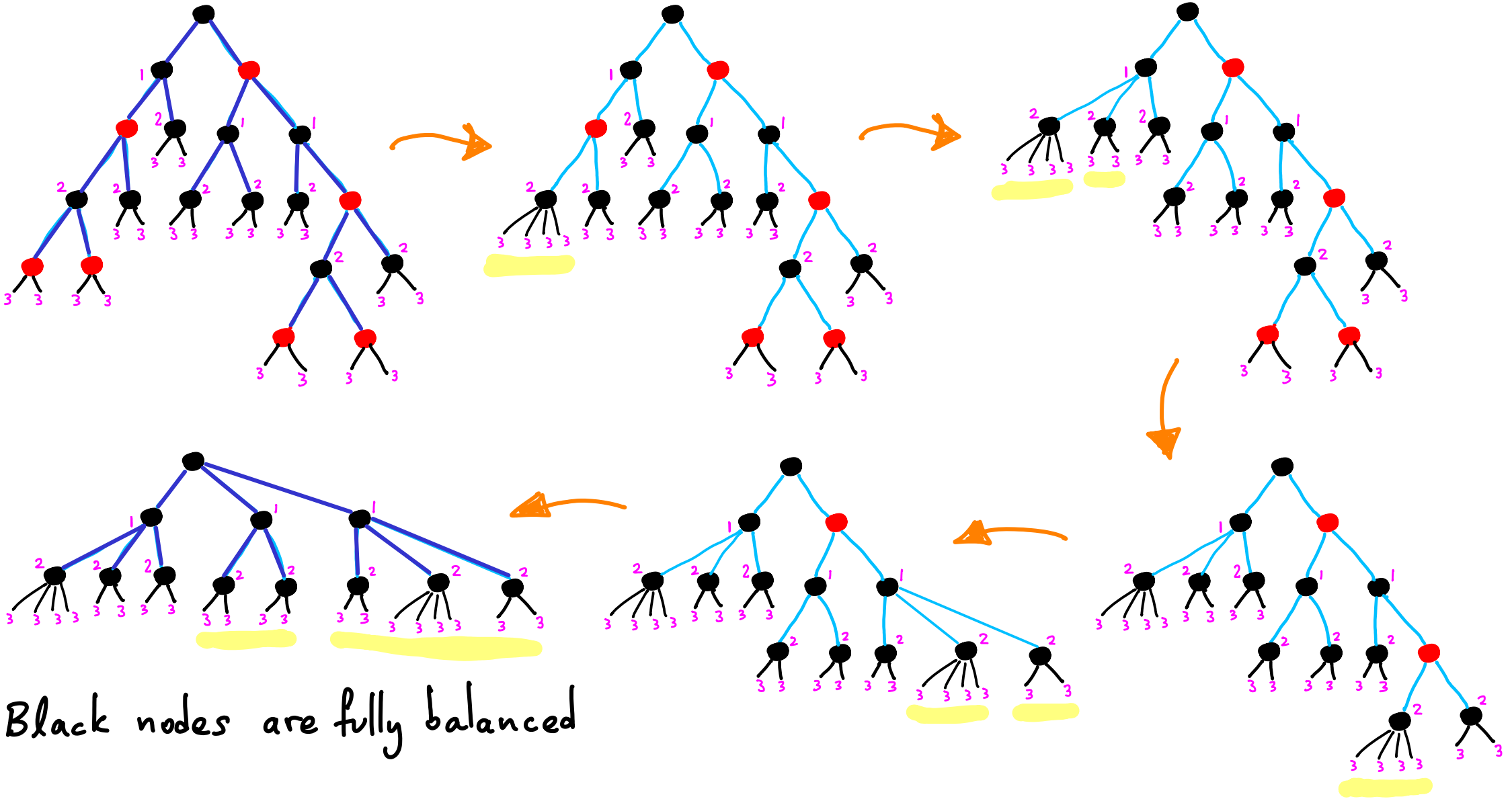

1
2
1
1
2
2
2
2
2
2
2
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3
3

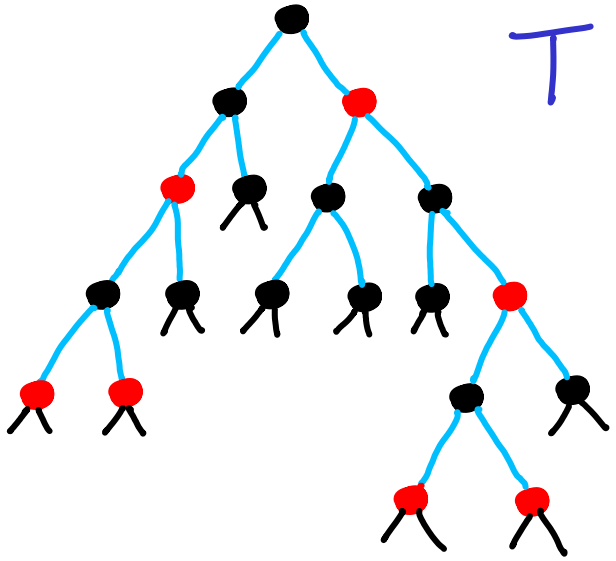


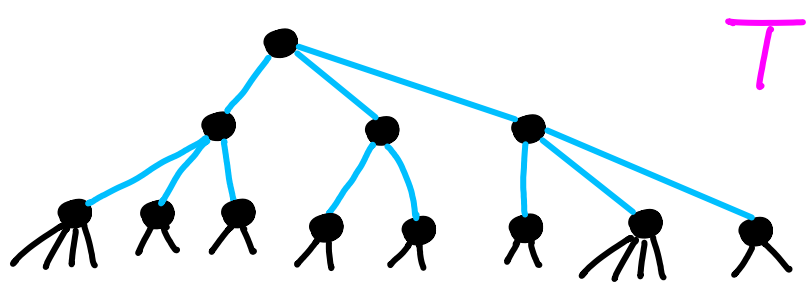




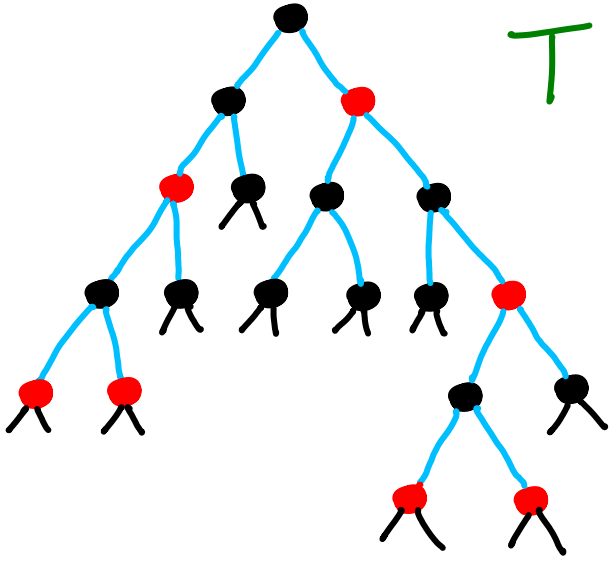


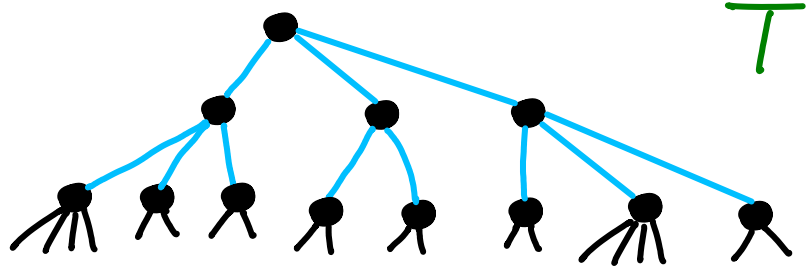

Black nodes are fully balanced





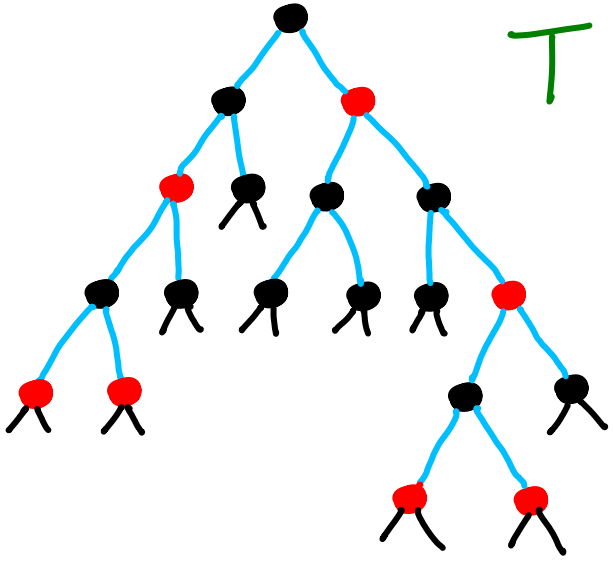
If root(T) has same black-height on all paths
then height(T') is perfectly balanced

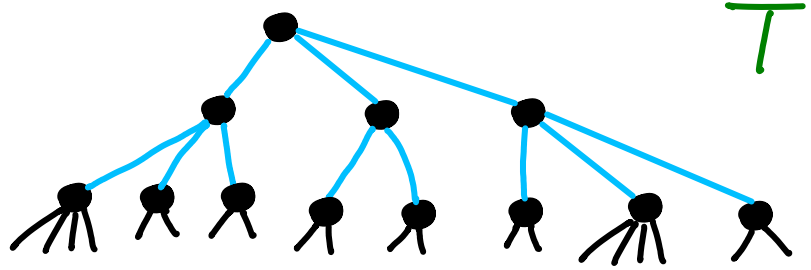




If root(T) has same black-height on all paths
then height(T') is perfectly balanced

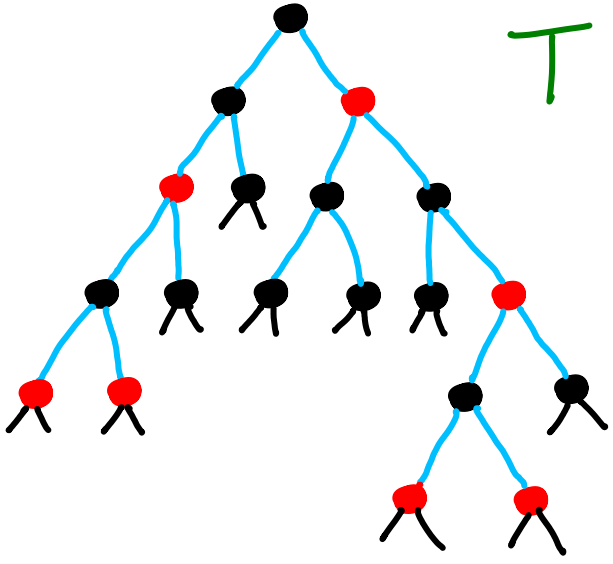
Q: what is max. degree of T' ?

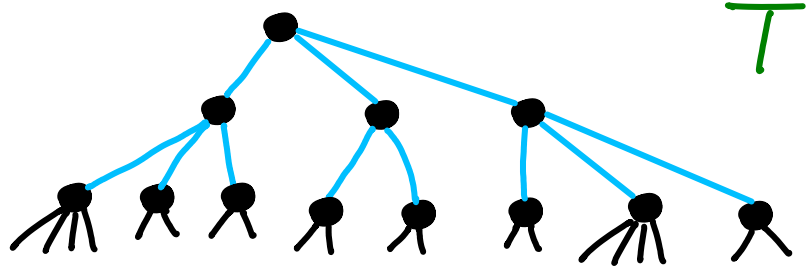




If root(T) has same black-height on all paths
then height(T') is perfectly balanced

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



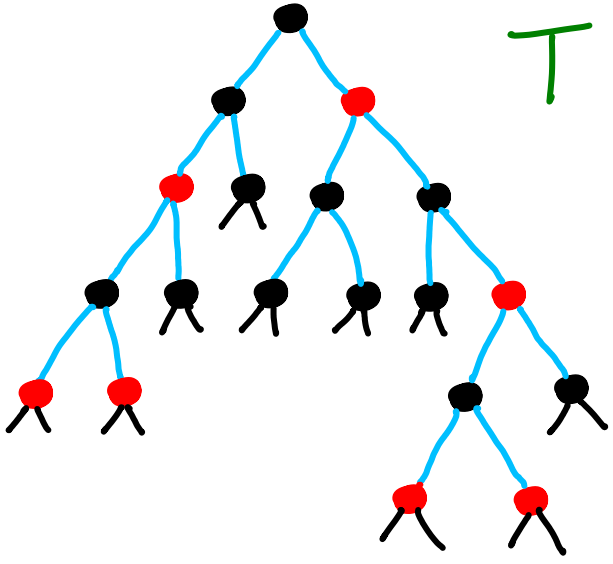


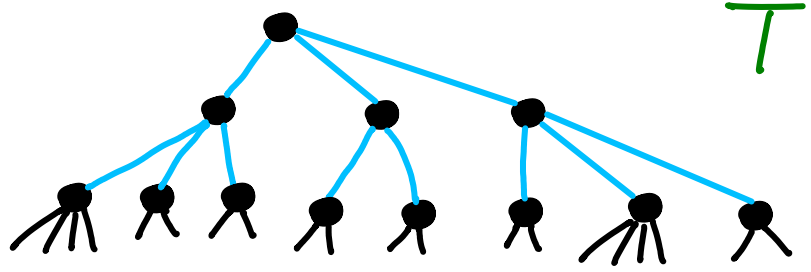
If root(T) has same black-height on all paths
then height(T') is perfectly balanced

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

#leaves in T : n+1 = size(T)+1

#leaves in T' : same





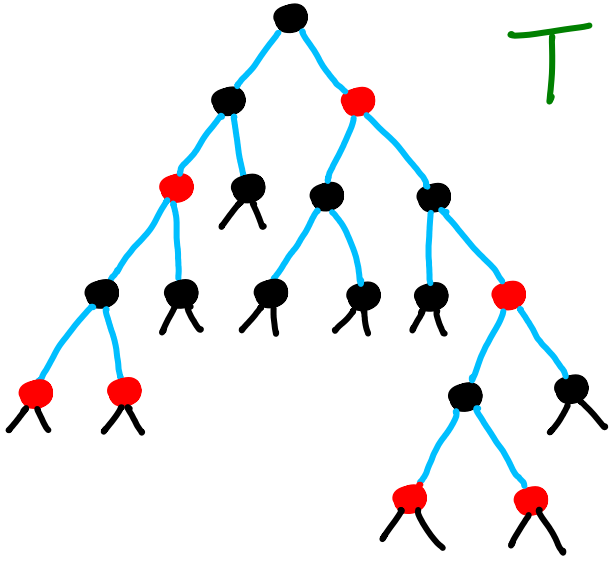
If root(T) has same black-height on all paths
then height(T') is perfectly balanced

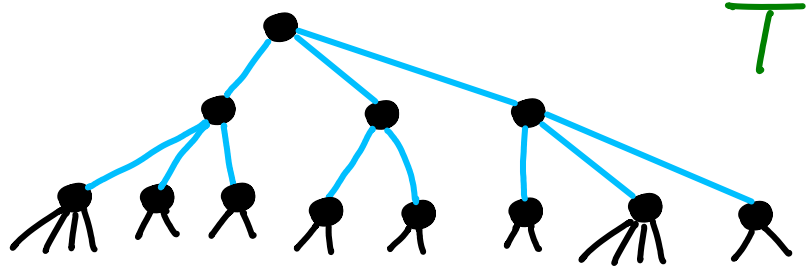
Q: what is max.degree of T' ? (2-3-4)

#leaves in T : n+1 = size(T)+1

#leaves in T' : same

height(T') ≤ log(n+1) [higher degree ⇒ smaller height ; worst-case : binary]





If root(T) has same black-height on all paths
then height(T') is perfectly balanced

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

#leaves in T : n+1 = size(T)+1

#leaves in T' : same

height(T') ≤ log(n+1) [higher degree ⇒ smaller height ; worst-case : binary]

Re-inserting red nodes : at most doubles height → height(T) ≤ 2 log(n+1)

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

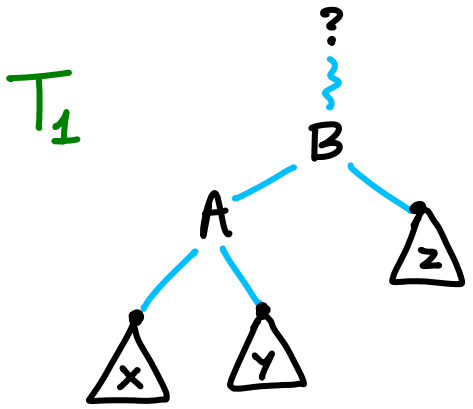
↳ search, min, max, next, prev : $O(\log n)$ time.

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

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)



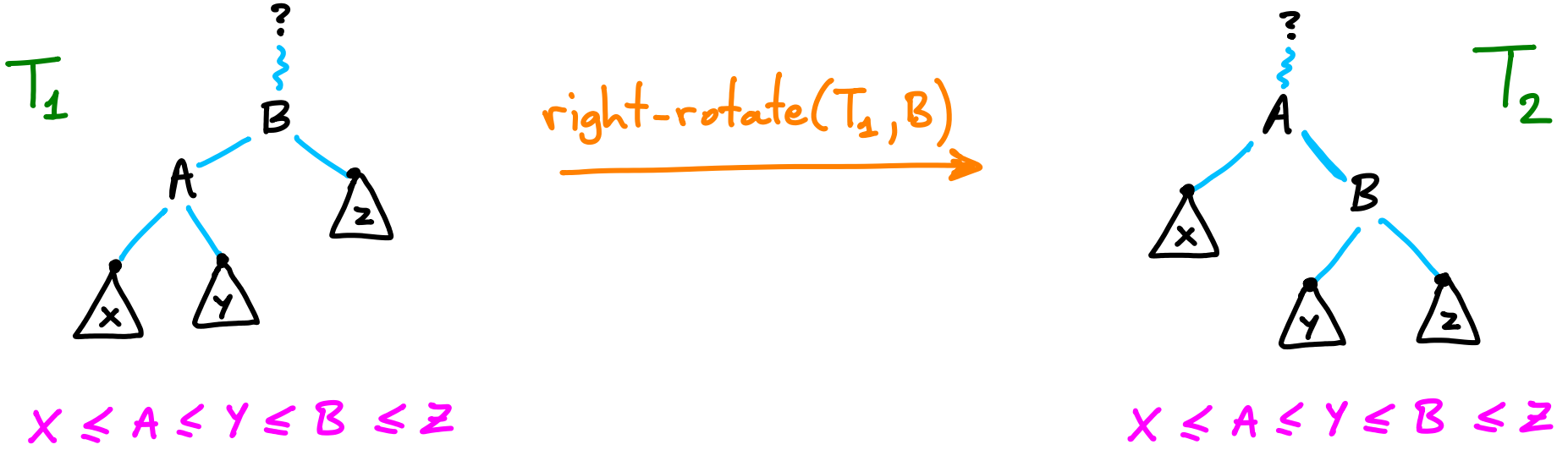
$X \leq A \leq Y \leq B \leq Z$

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



$X \leq A \leq Y \leq B \leq Z$

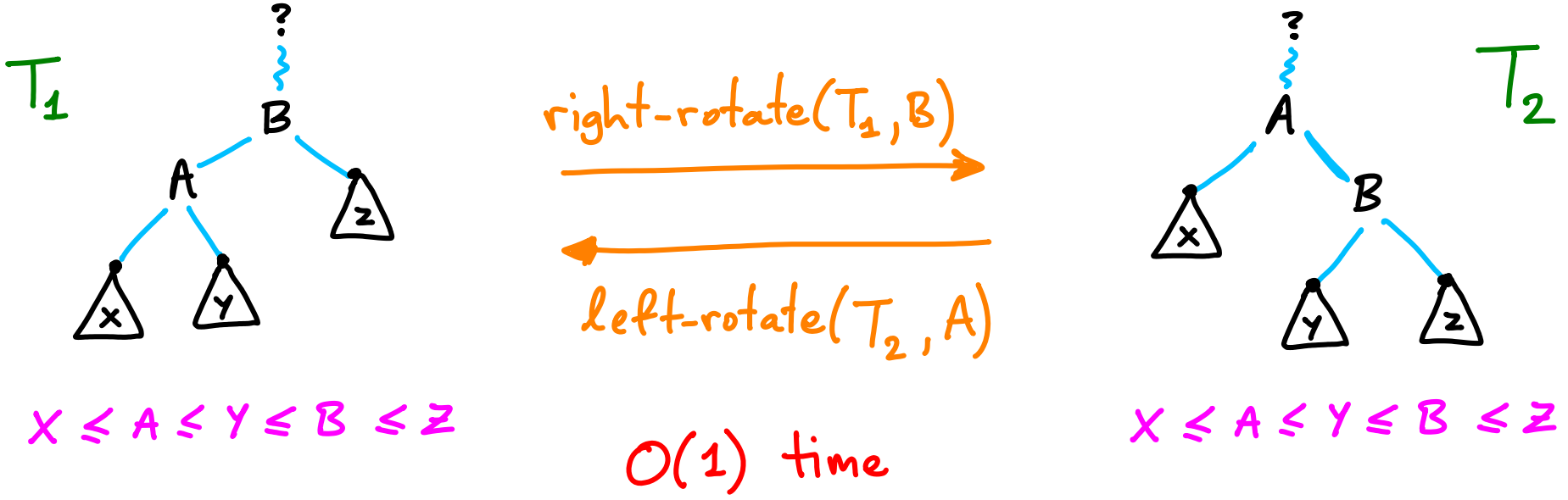
$X \leq A \leq Y \leq B \leq Z$

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



$x \le A \le y \le B \le z$

$O(1)$ time

$x \le A \le y \le B \le z$

INSERT IN RED BLACK TREES

greedy (optimistic) start : insert as in any BST

INSERT IN RED BLACK TREES

greedy (optimistic) start : { 1) insert as in any BST
2) color new node red (keeps black-height ok)

INSERT IN RED BLACK TREES

greedy (optimistic) start :
1) insert as in any BST
2) color new node red (keeps black-height ok)
3) if parent is also red (violate parent rule 4)

INSERT IN RED BLACK TREES

greedy (optimistic) start :
1) insert as in any BST
2) color new node red (keeps black-height ok)
3) if parent is also red (violate parent rule 4)
then color parent black

INSERT IN RED BLACK TREES

greedy (optimistic) start :

1) insert as in any BST
2) color new node red (keeps black-height ok)
3) if parent is also red (violate parent rule 4)
   then color parent black and
   look for problems further up

begins an error-correcting
trail up to root,
involving O(1) recolorings
and rotations per level

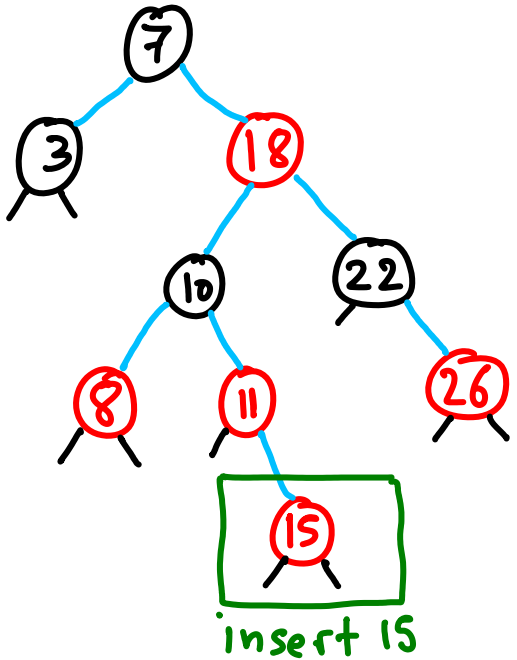
INSERT IN RED BLACK TREES

greedy (optimistic) start :
1) insert as in any BST
2) color new node red (keeps black-height ok)
3) if parent is also red (violate parent rule 4)
then color parent black and
look for problems further up

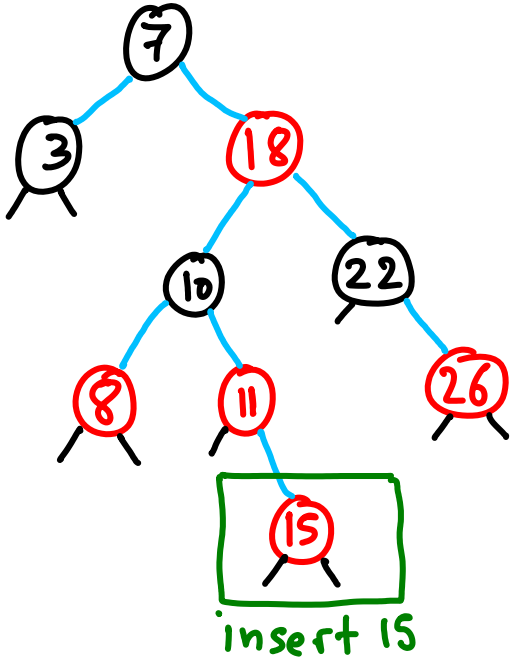
begins an error-correcting
trail up to root,
involving O(1) recolorings
and rotations per level

→ O(log n) time

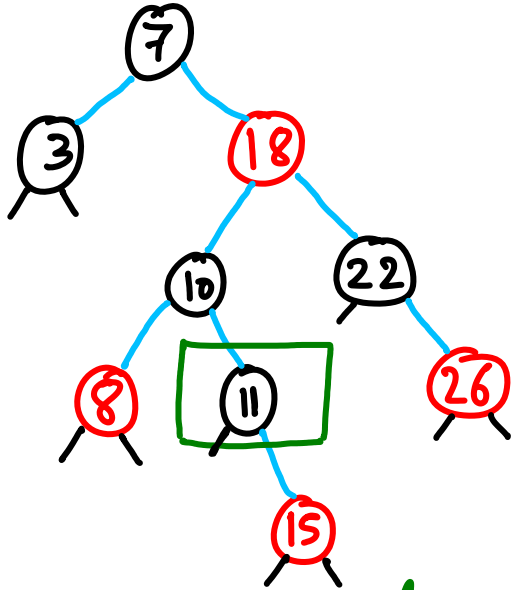
example of O(1)-time error-corrections (per level)



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



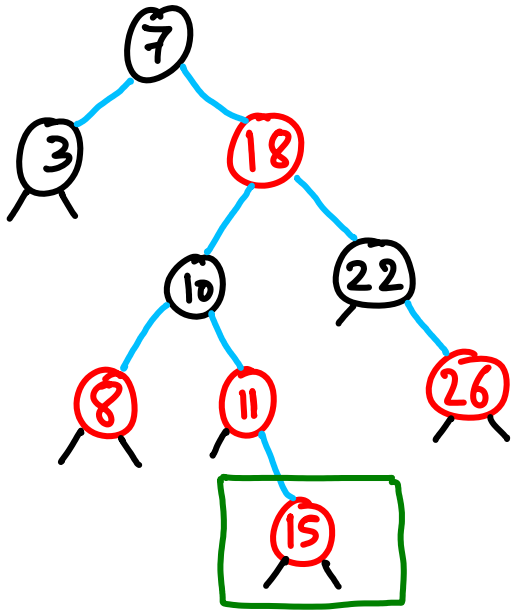
insert 15



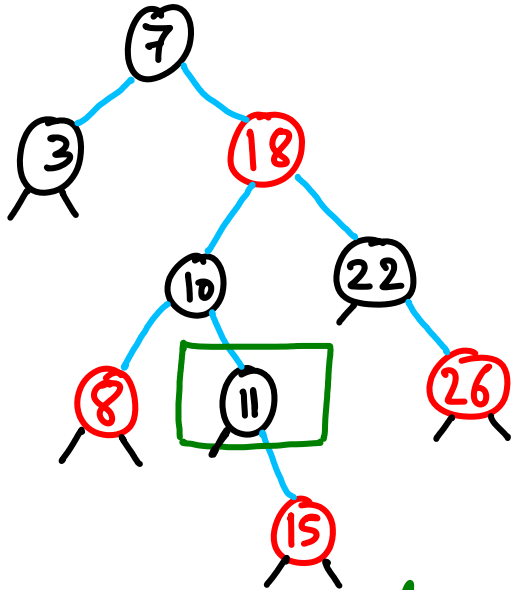
recolor parent of 15

↳ create black-height problem

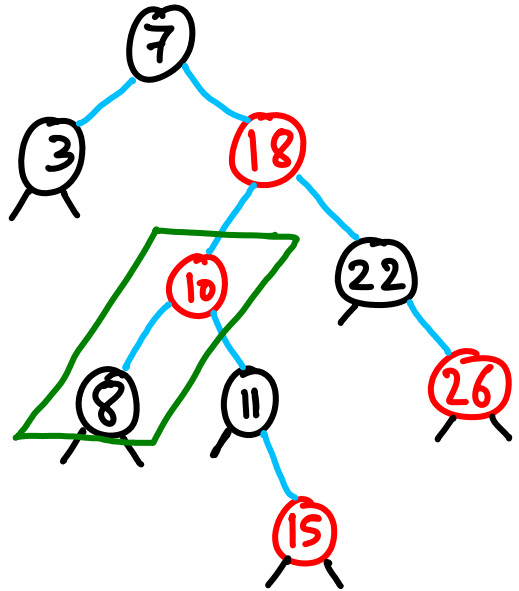
example of O(1)-time error-corrections (per level)



insert 15

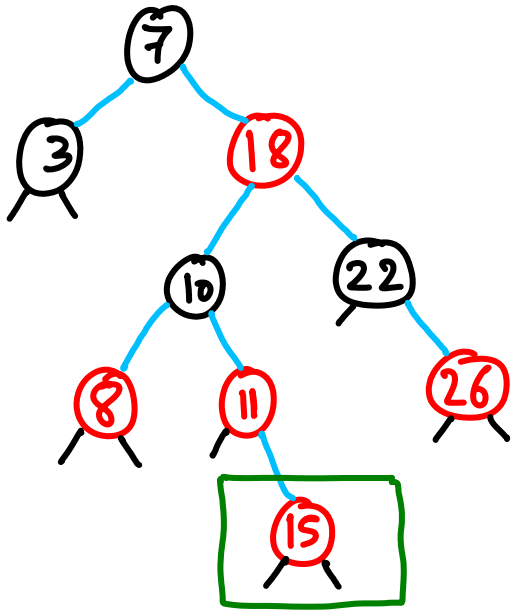


recolor parent of 15
↳ create black-height problem

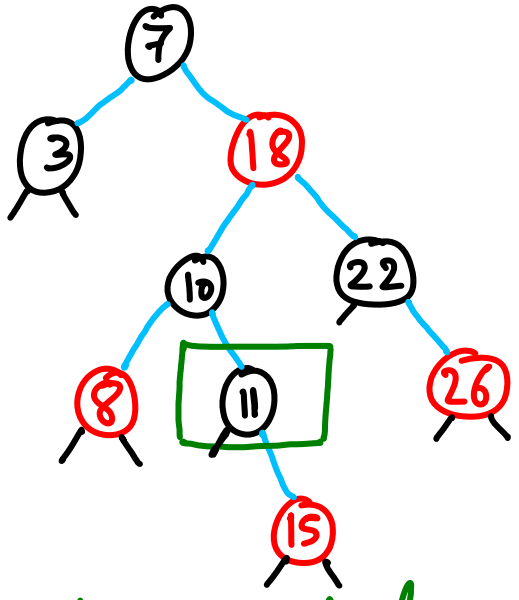


black height problem fixed.
↳ new consecutive-red problem

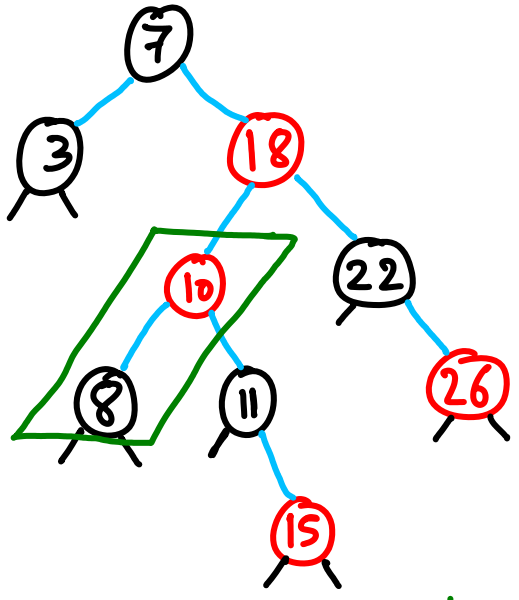
example of O(1)-time error-corrections (per level)



insert 15

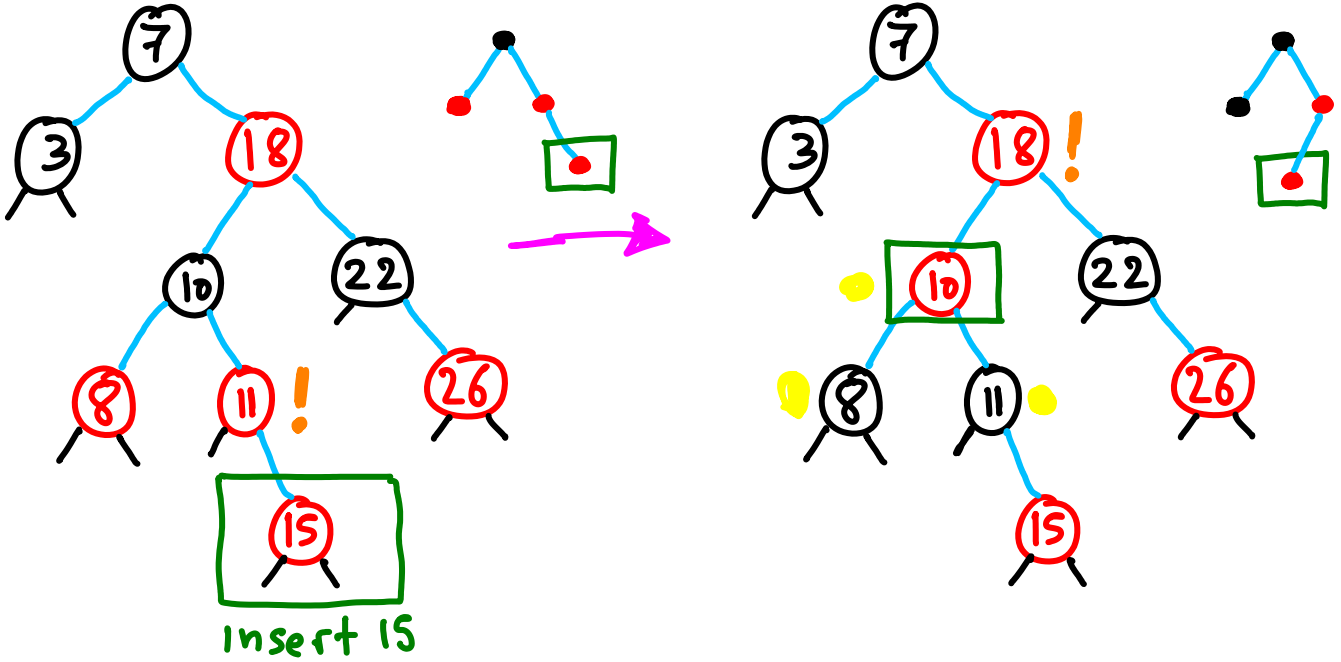


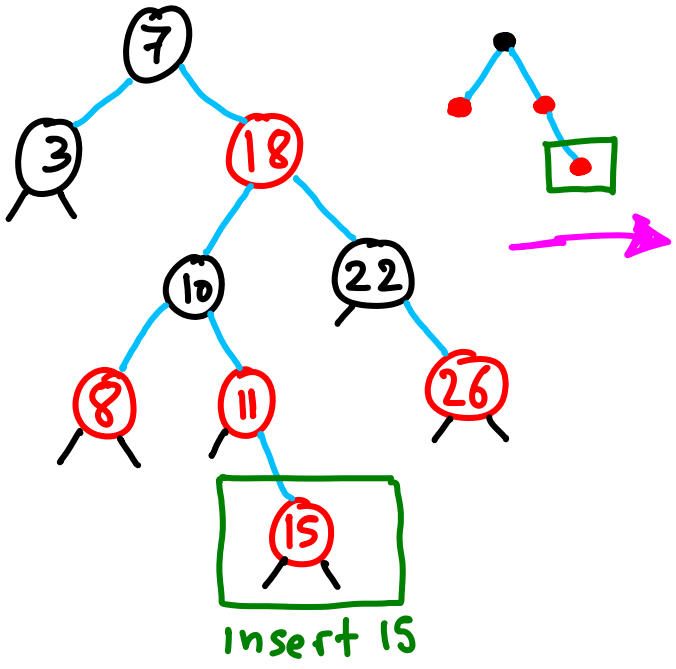
recolor parent of 15
↳ create black-height problem

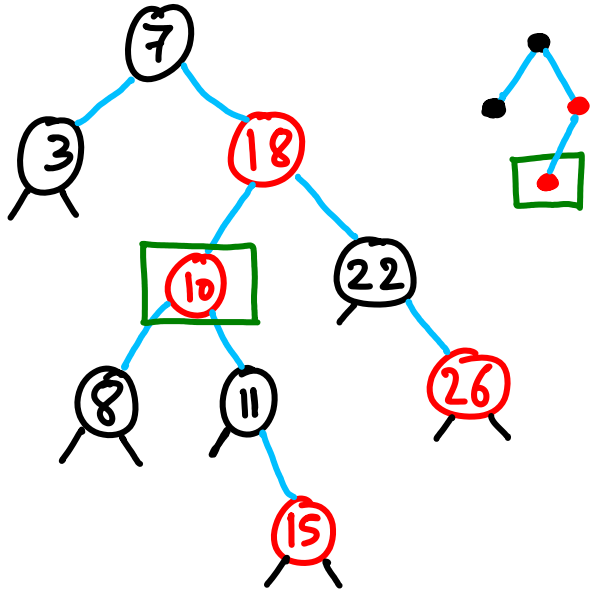


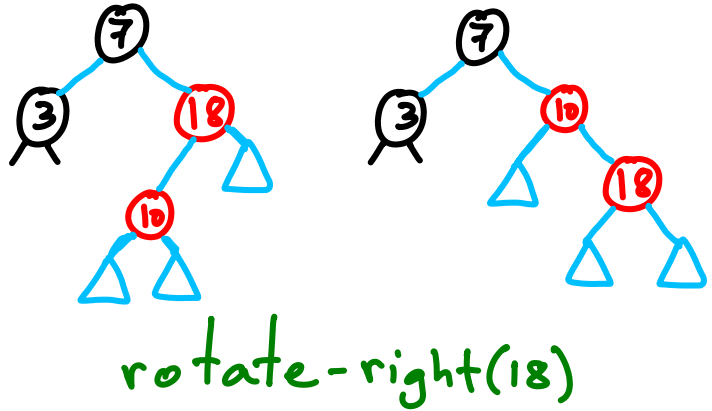
black height problem fixed.
↳ new consecutive-red problem

problem transferred from new node to its grandparent

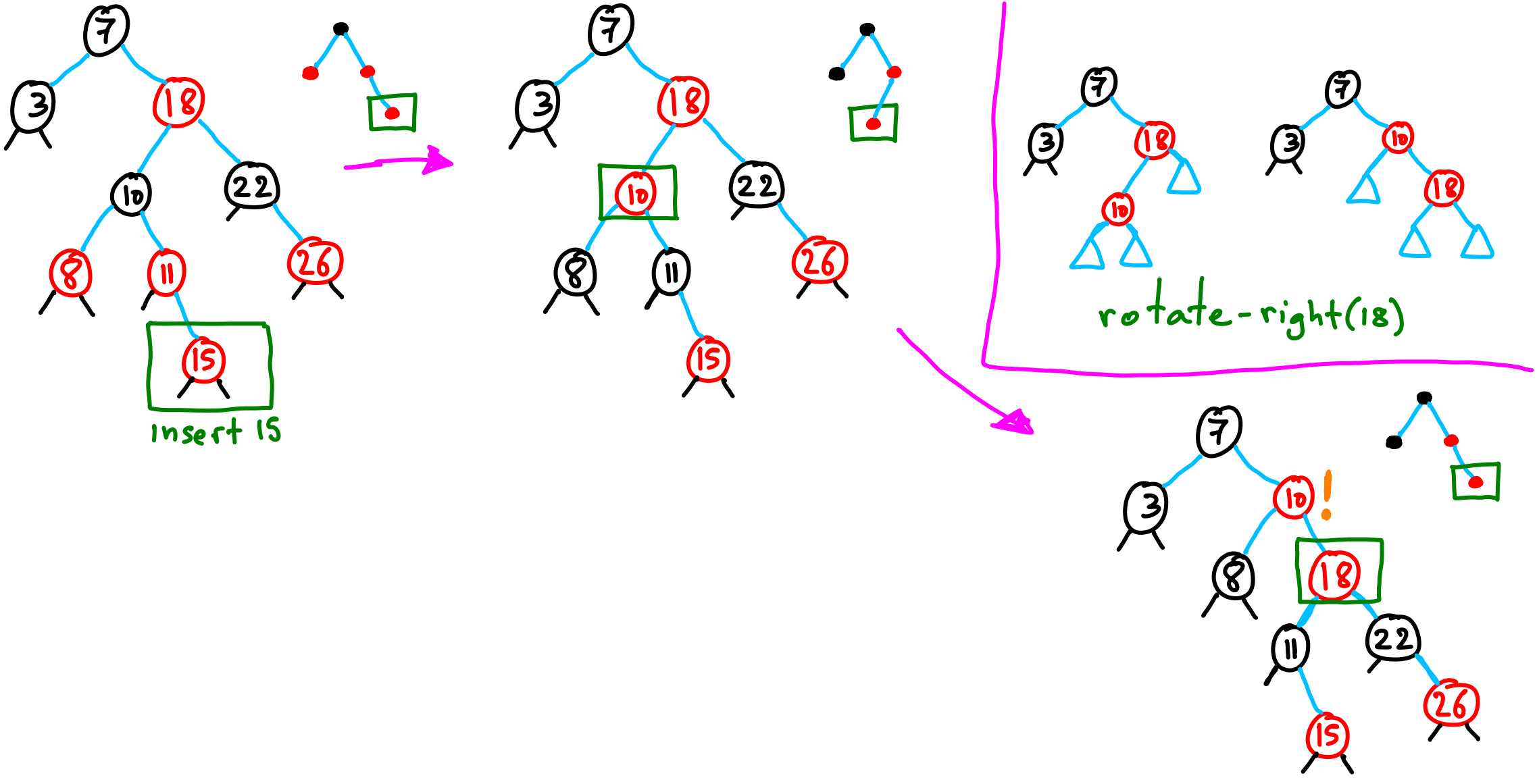

7
3
18
10
22
8
11
!
26
15
insert 15
7
3
18
!
10
22
8
11
26
15

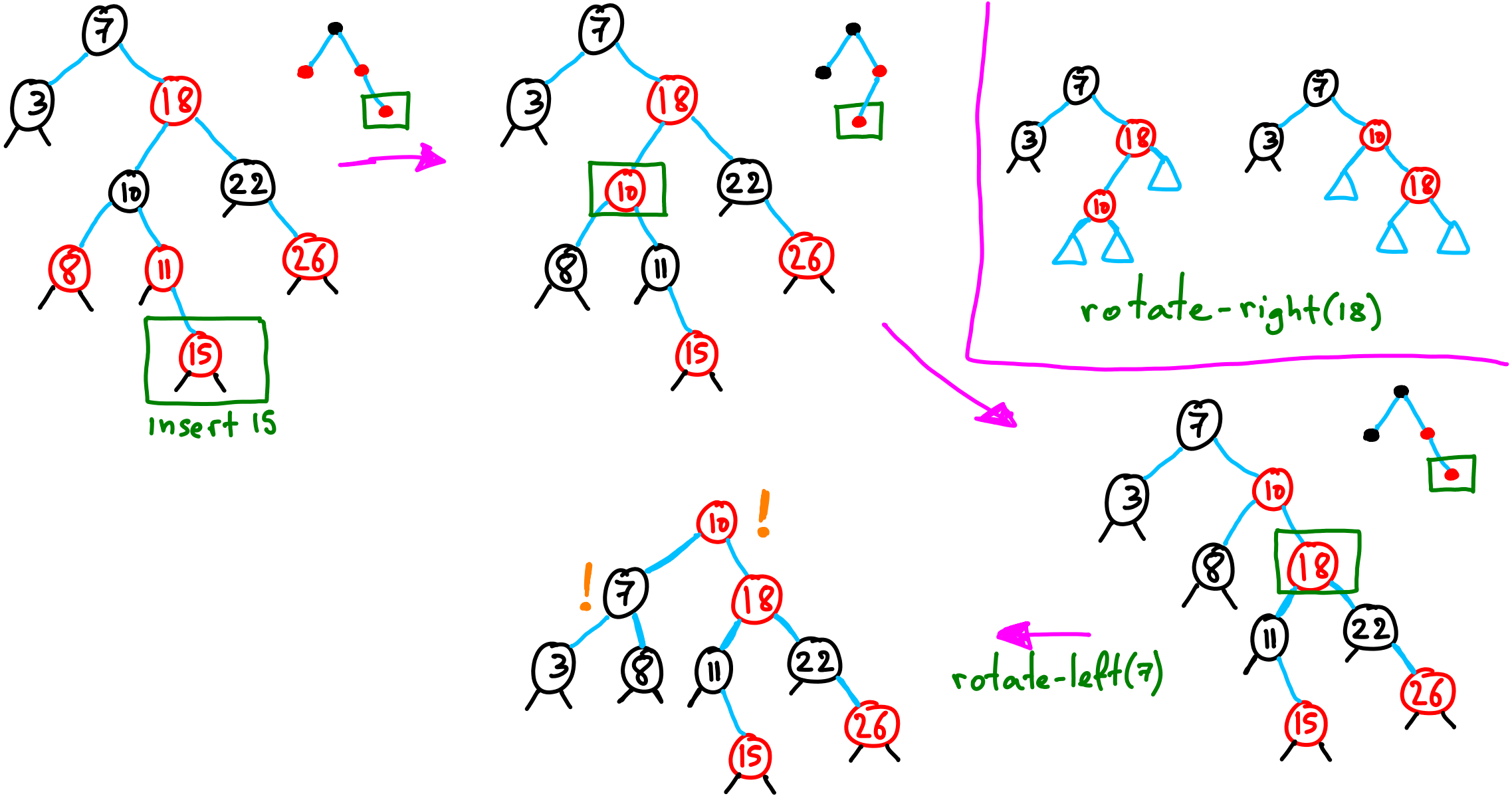


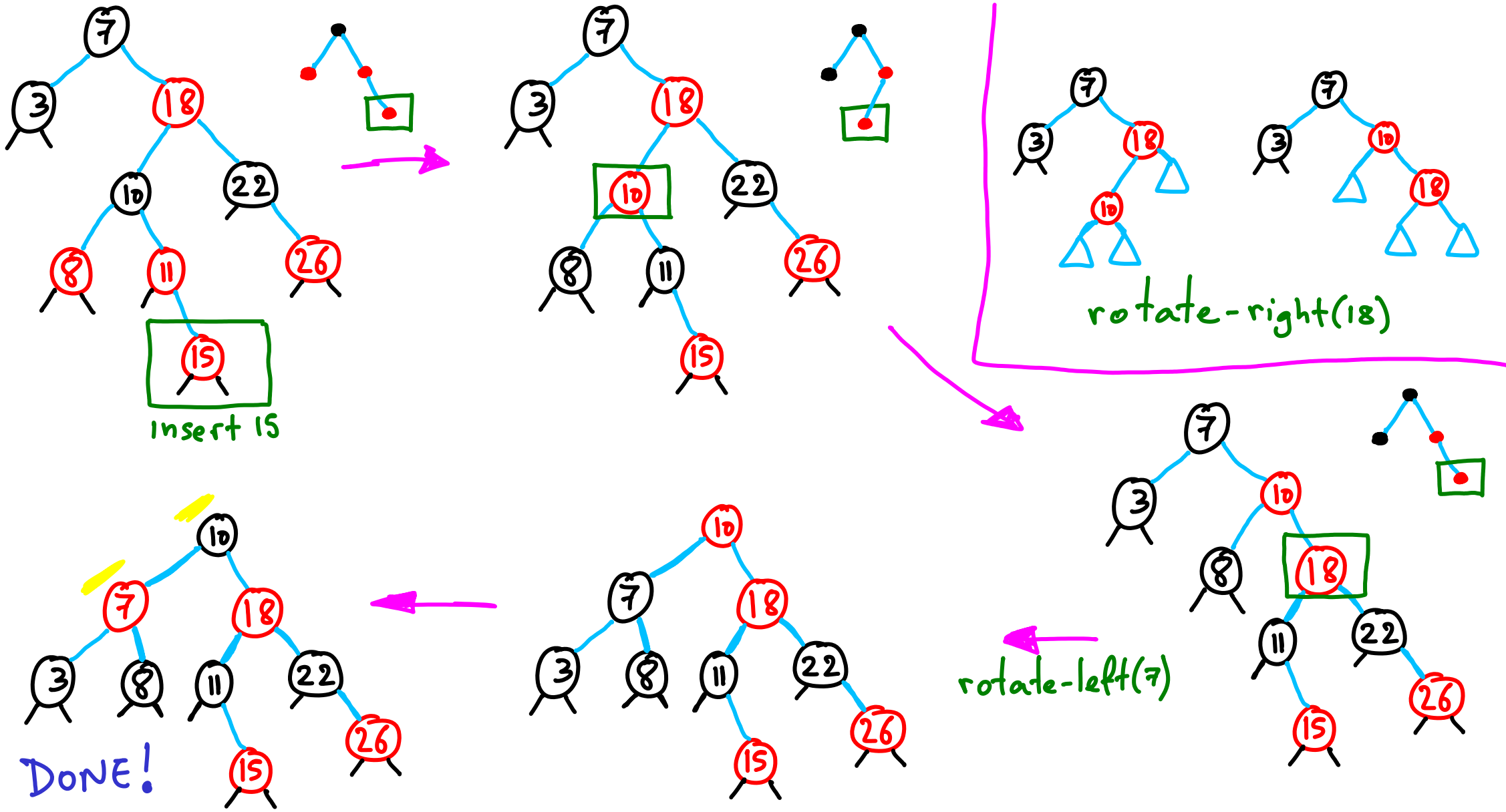




(doesn't affect BH in this case)


7
3
18
10
22
8
11
26
15
insert 15
rotate-right(18)
!


7
3
18
10
22
8
11
26
15
insert 15
rotate-right(18)
rotate-left(7)
!
!


7
3
18
10
22
8
11
26
15
insert 15
rotate-right(18)
rotate-left(7)
DONE!

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

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

RB-insert(x)

- initial step: regular insert & color x red. If x = root, trivial: x → black

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

RB-insert(x)

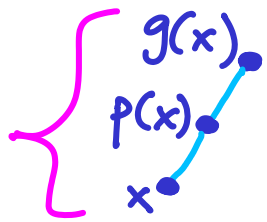
- initial step: regular insert & color x red.
- while $x \neq$ root and $p(x)$ is red

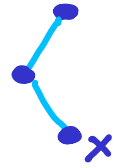
↘ implies x will be propagating up

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

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and $p(x)$ is red

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

or 

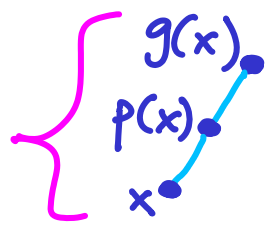
} don't care about colors

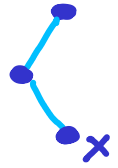
↳ why does $g(x)$ exist?

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

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and $p(x)$ is red

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

or 

} don't care about colors

↳ why does $g(x)$ exist?

$p(x)$ = red so it isn't the root.

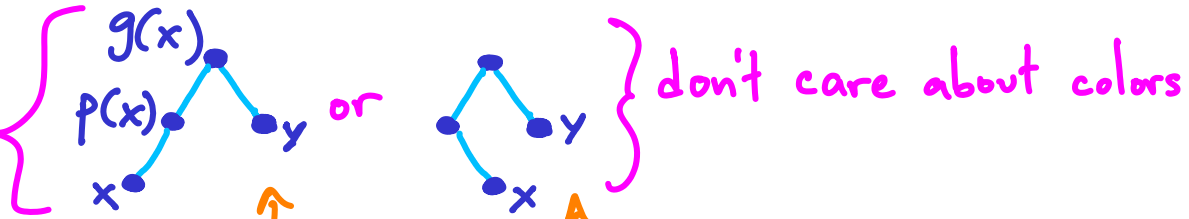
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and $p(x)$ is red

  if $p(x) = left(g(x))$

  $y \leftarrow right(g(x))$

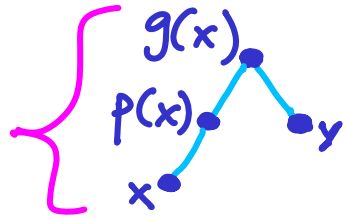


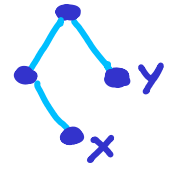
why can we assume y exists?

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

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and $p(x)$ is red

if $p(x) = left(g(x))$ 

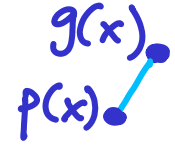
or 

} don't care about colors

$y \leftarrow right(g(x))$

↳ why can we assume y exists?

It is at least a dummy leaf.

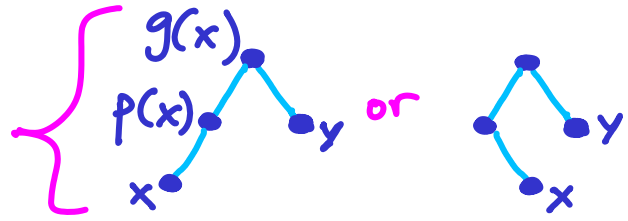
in which case the subtree of $g(x)$ is 

before inserting x.

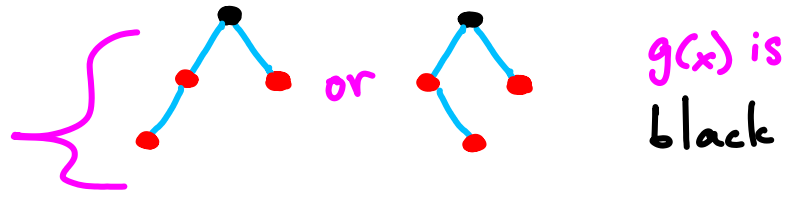
The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and $p(x)$ is red

if $p(x) = left(g(x))$ 

$y \leftarrow right(g(x))$

if y is red then run CASE 1 

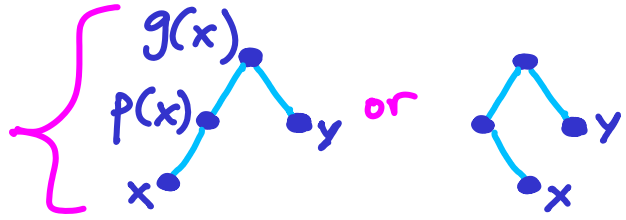
g(x) is black

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

RB-insert(x)

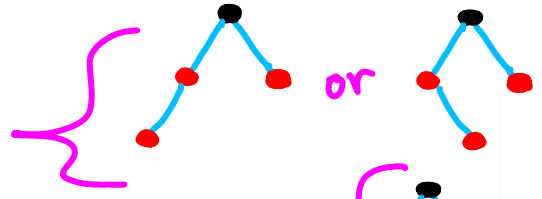
- initial step: regular insert & color x red.
- while $x \neq$ root and p(x) is red

'if p(x) = left(g(x))

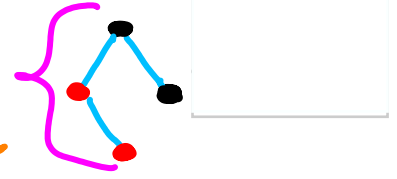


y ← right(g(x))

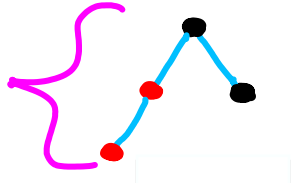
if y is red then run CASE 1



else if x = right(p(x)) then run CASE 2.



Run CASE 3

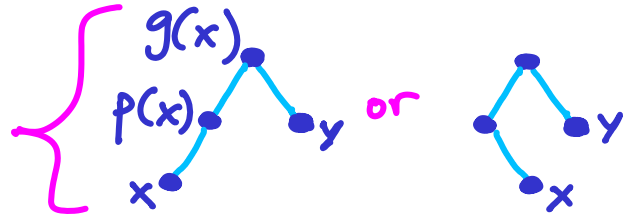


The algorithm is basically: recolor upwards until ineffective, then do 2 rotations

RB-insert(x)

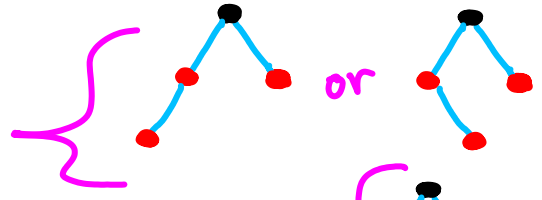
- initial step: regular insert & color x red.
- while $x \neq$ root and p(x) is red

'if p(x) = left(g(x))

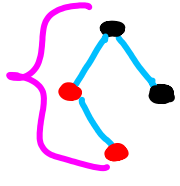


y ← right(g(x))

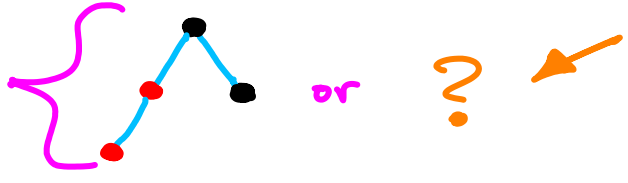
if y is red then run CASE 1



else if x = right(p(x)) then run CASE 2.



Run CASE 3



else \\ p(x) = right(g(x))

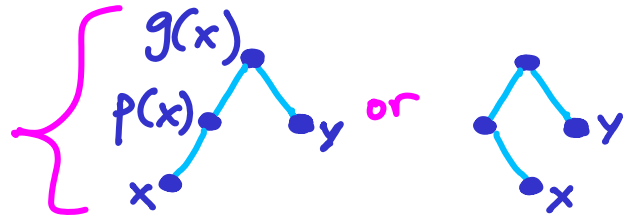
do as before but with "left" & "right" switched \\symmetric

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

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and p(x) is red

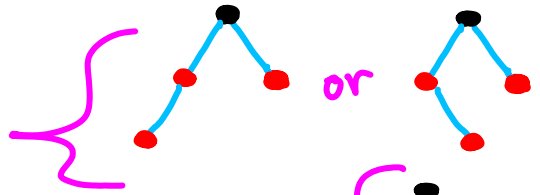
  if p(x) = left(g(x))

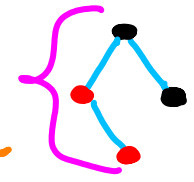
  these will move x up (redefine) (and it will be red)

  y ← right(g(x))

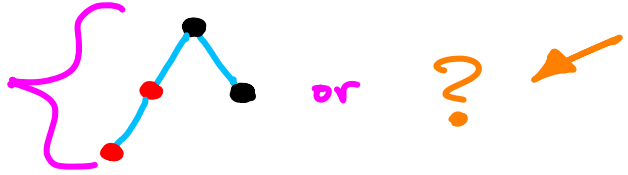
  if y is red then run CASE 1

  else if x = right(p(x)) then run CASE 2.

  Run CASE 3

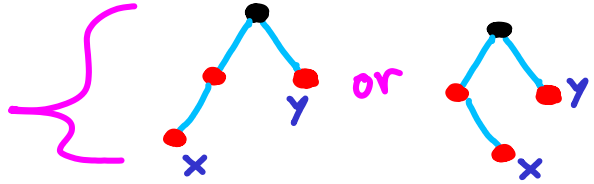
  

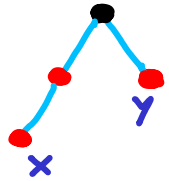
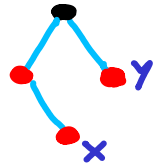
  else // p(x) = right(g(x))

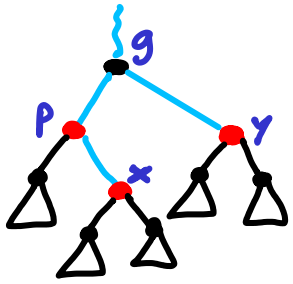
  do as before but with "left" & "right" switched // symmetric

- color the root black

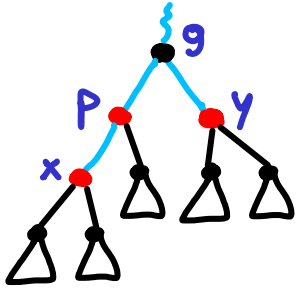
if y is red then run CASE 1

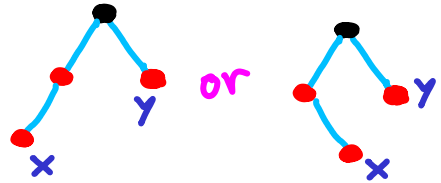


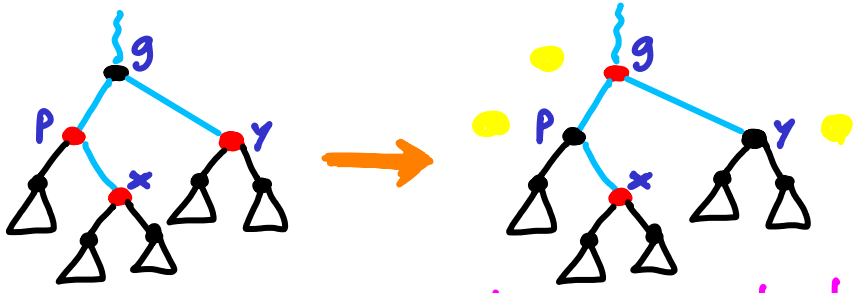
if y is red then run CASE 1 {  or 



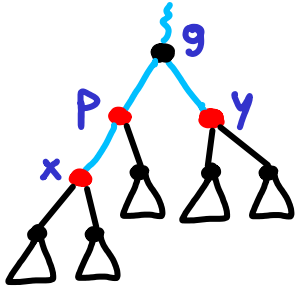
Every △ contributes same to black-height.

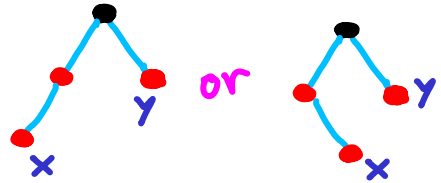


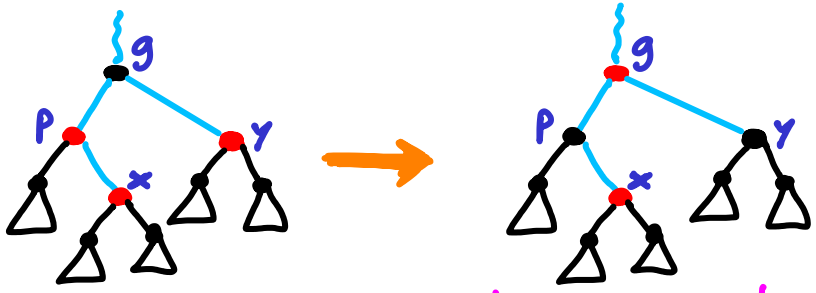
if y is red then run CASE 1 {  or



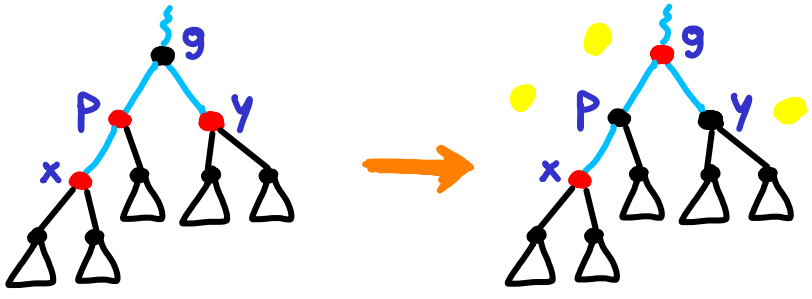
Every △ contributes same to black-height.



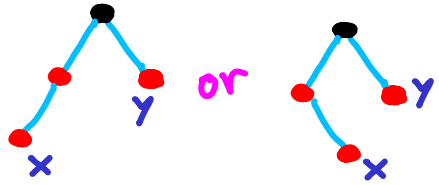
if y is red then run CASE 1 {  or

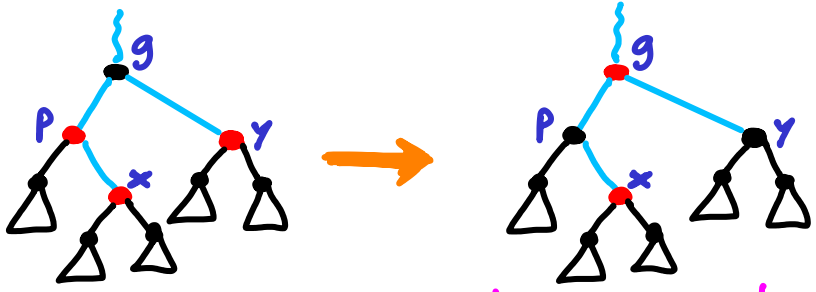


Every △ contributes same to black-height.

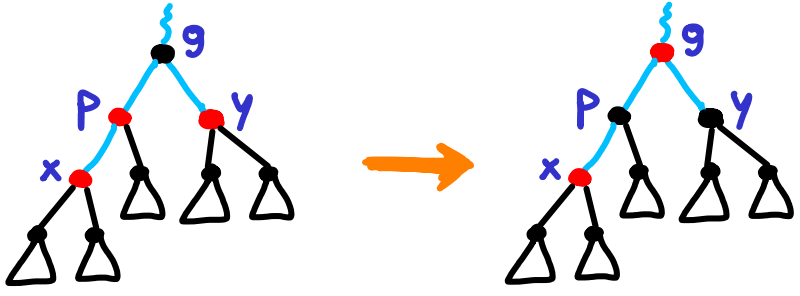


if y is red then run CASE 1



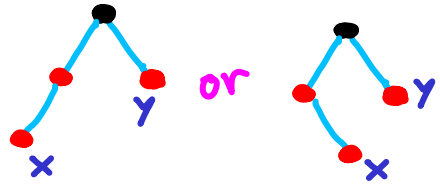


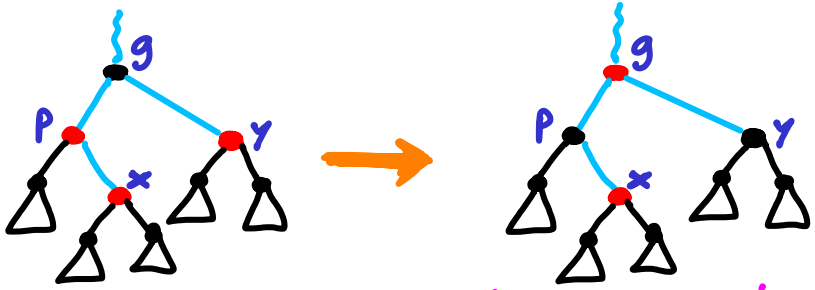
Every △ contributes same to black-height.



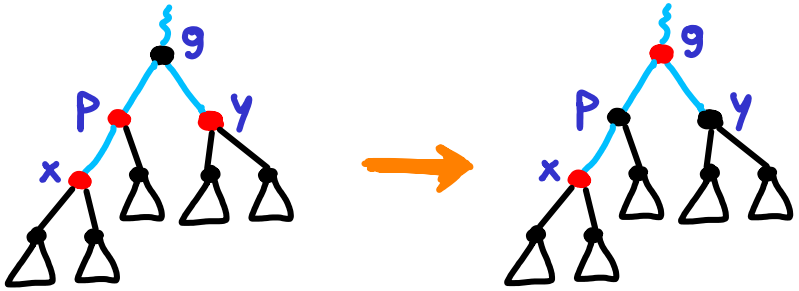
Recolor y, p(x), g(x)
Let $x \leftarrow g(x)$

if y is red then run CASE 1



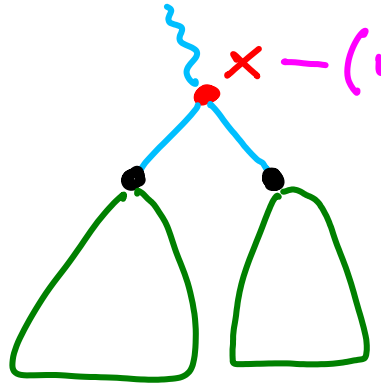


Every △ contributes same to black-height.



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

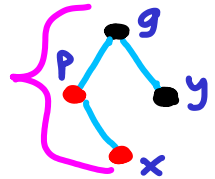
blackheight(p(x)) preserved



return to while loop.

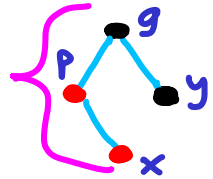
(repeat if new p(x) is red)

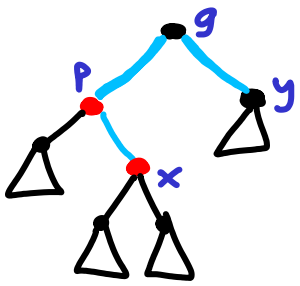
else if $x = \text{right}(p(x))$ then run CASE 2.



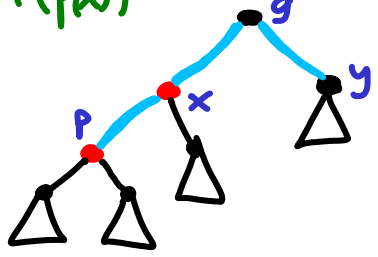
(y is black)

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

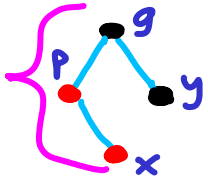


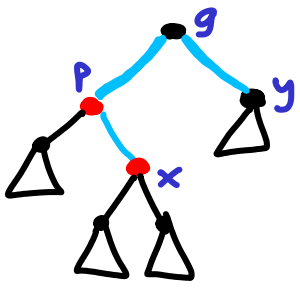


rotate-left(p(x))

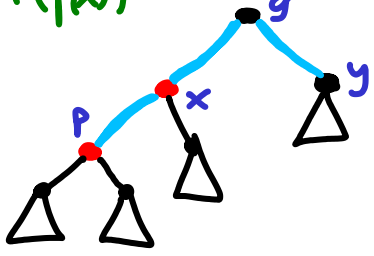


else if $x = \text{right}(p(x))$ then run CASE 2.

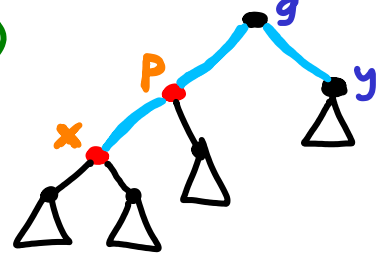




rotate-left(p(x))

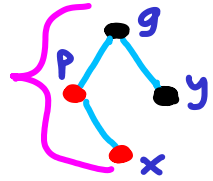


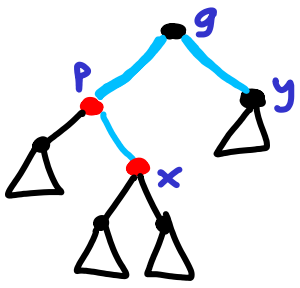
switch labels
$x \leftrightarrow p(x)$



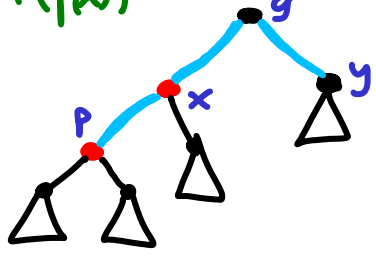
conditions of case 3

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

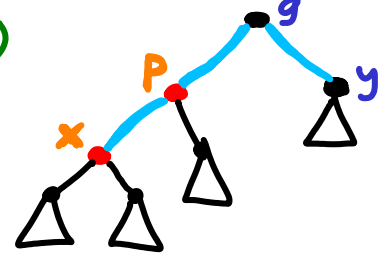




rotate-left(p(x))

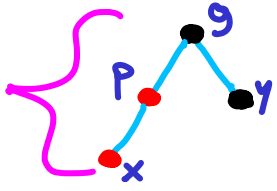


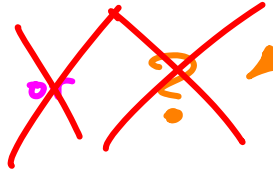
switch labels
$x \leftrightarrow p(x)$



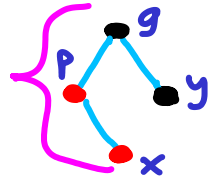
conditions of case 3

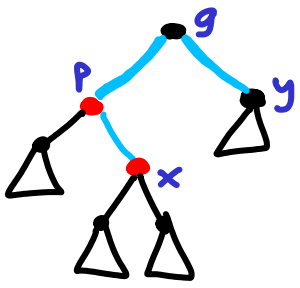
Run CASE 3



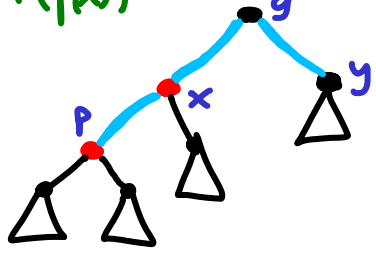


else if $x = \text{right}(p(x))$ then run CASE 2.

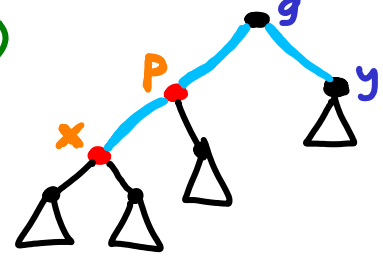




rotate-left(p(x))

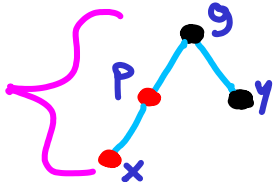


switch labels
$x \leftrightarrow p(x)$

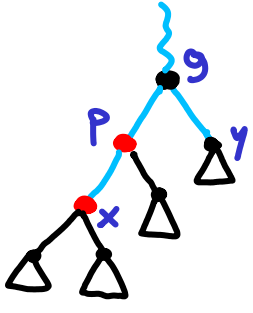


conditions of case 3

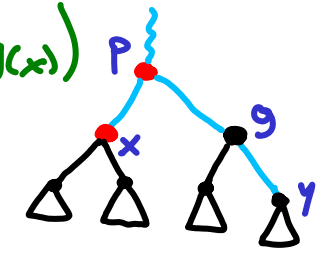
Run CASE 3



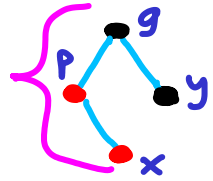
or ?

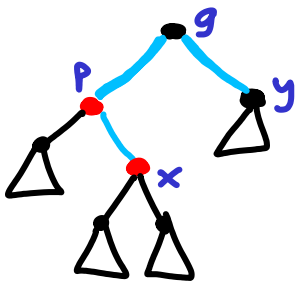


rotate-right(g(x))

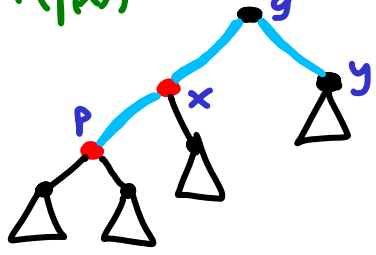


else if $x = \text{right}(p(x))$ then run CASE 2.

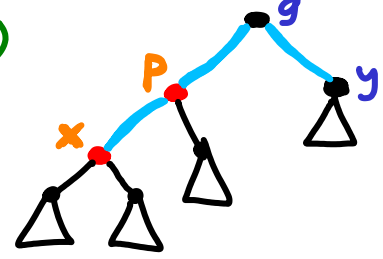




rotate-left(p(x))

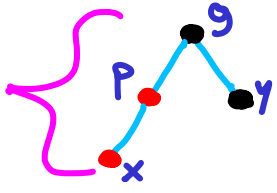


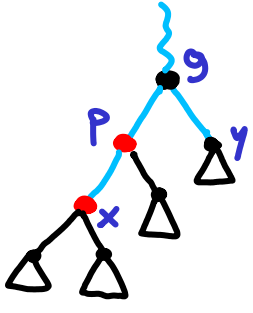
switch labels
$x \leftrightarrow p(x)$



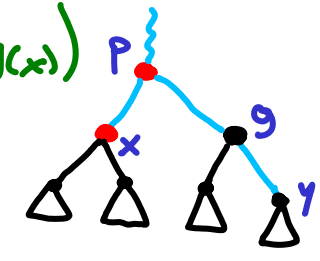
conditions of case 3

Run CASE 3

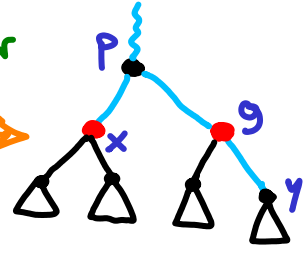




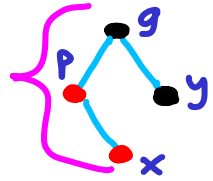
rotate-right(g(x))

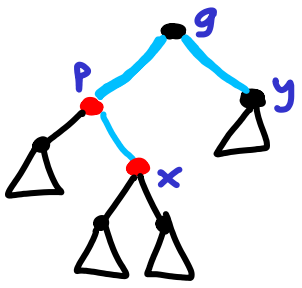


re-color p,g

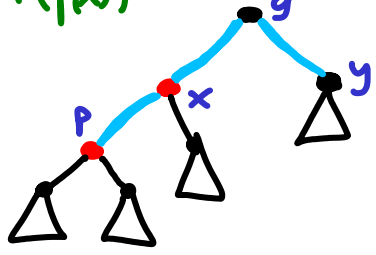


else if $x = \text{right}(p(x))$ then run CASE 2.

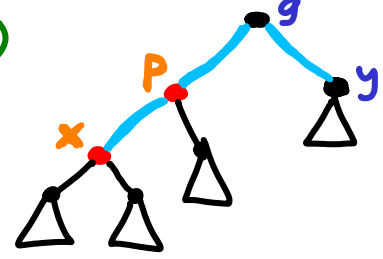




rotate-left(p(x))

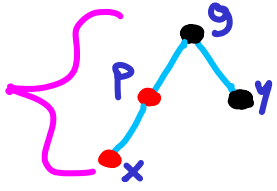


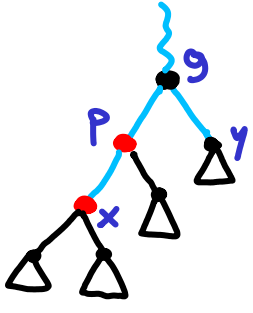
switch labels
$x \leftrightarrow p(x)$



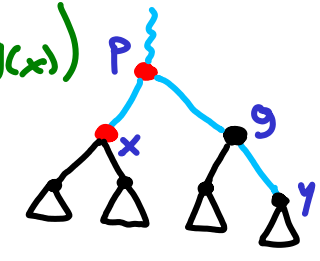
conditions of case 3

Run CASE 3

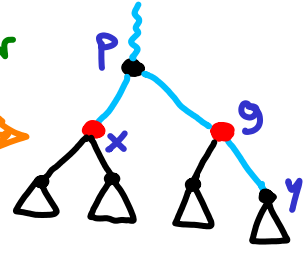




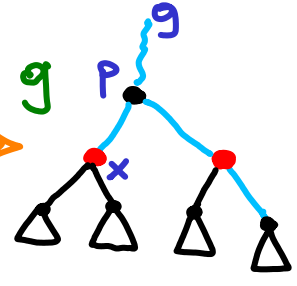
rotate-right(g(x))



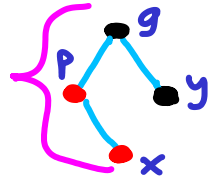
re-color p, g

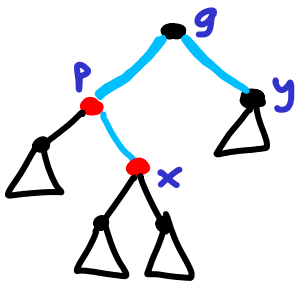


relabel g

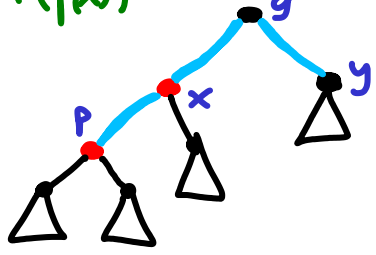


else if $x = \text{right}(p(x))$ then run CASE 2.

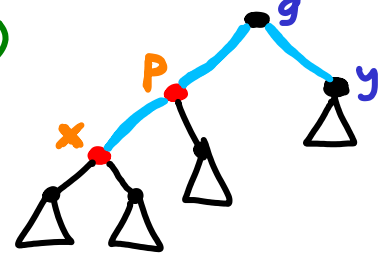




rotate-left(p(x))

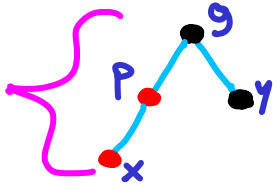


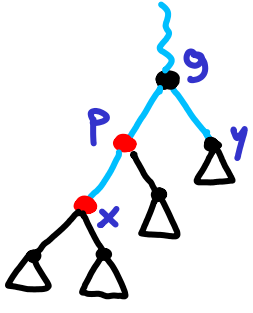
switch labels
$x \leftrightarrow p(x)$



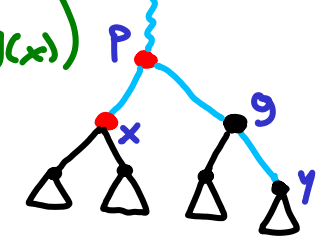
conditions of case 3

Run CASE 3

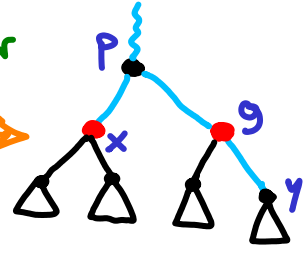




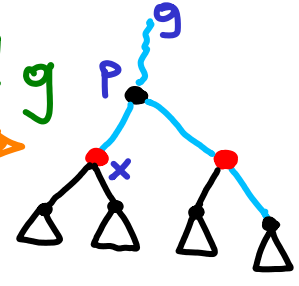
rotate-right(g(x))



re-color p, g

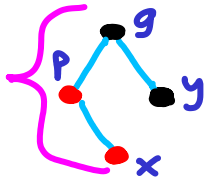


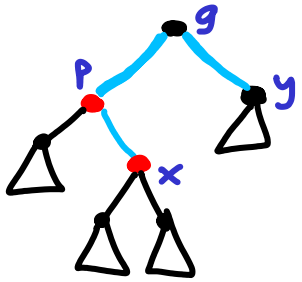
relabel g



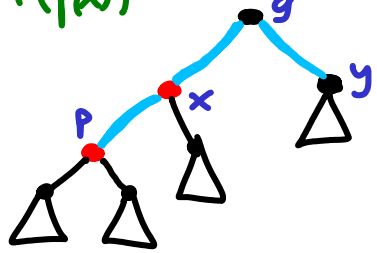
x moves up in tree

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

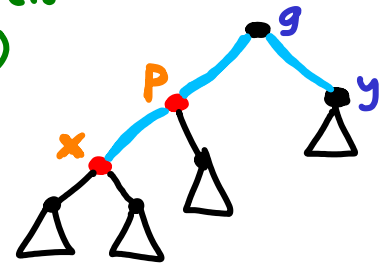




rotate-left(p(x))

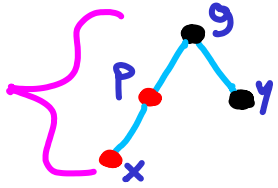


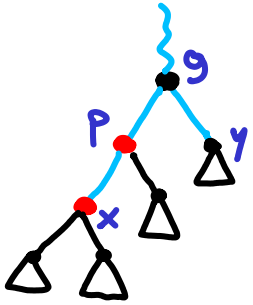
switch labels
$x \leftrightarrow p(x)$



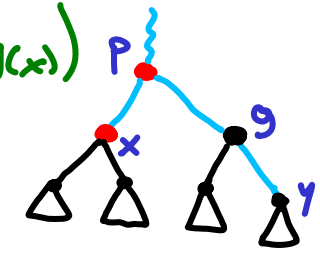
conditions of case 3

Run CASE 3

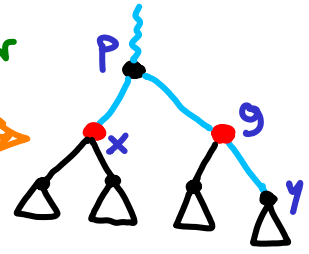




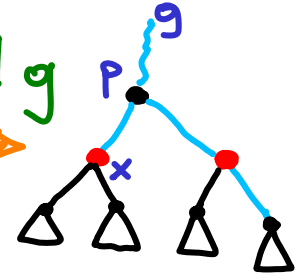
rotate-right(g(x))



re-color p, g



relabel g



x moves up in tree

Now p(x) is black.

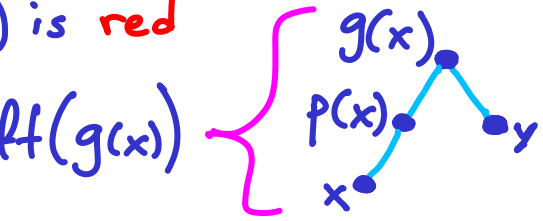
Exit (while)

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

RB-insert(x)

- initial step: regular insert & color x red.
- while $x \neq$ root and p(x) is red

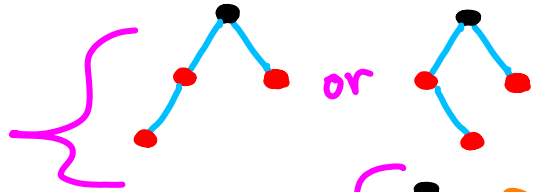
  if p(x) = left(g(x))

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

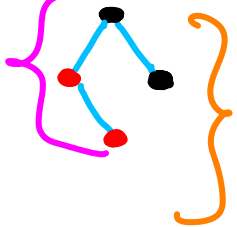
  y ← right(g(x))

  if y is red then run CASE 1

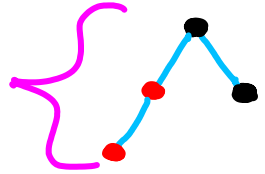
  recolor, repeat WHILE

  else if x = right(p(x)) then run CASE 2.

  rotate, exit WHILE

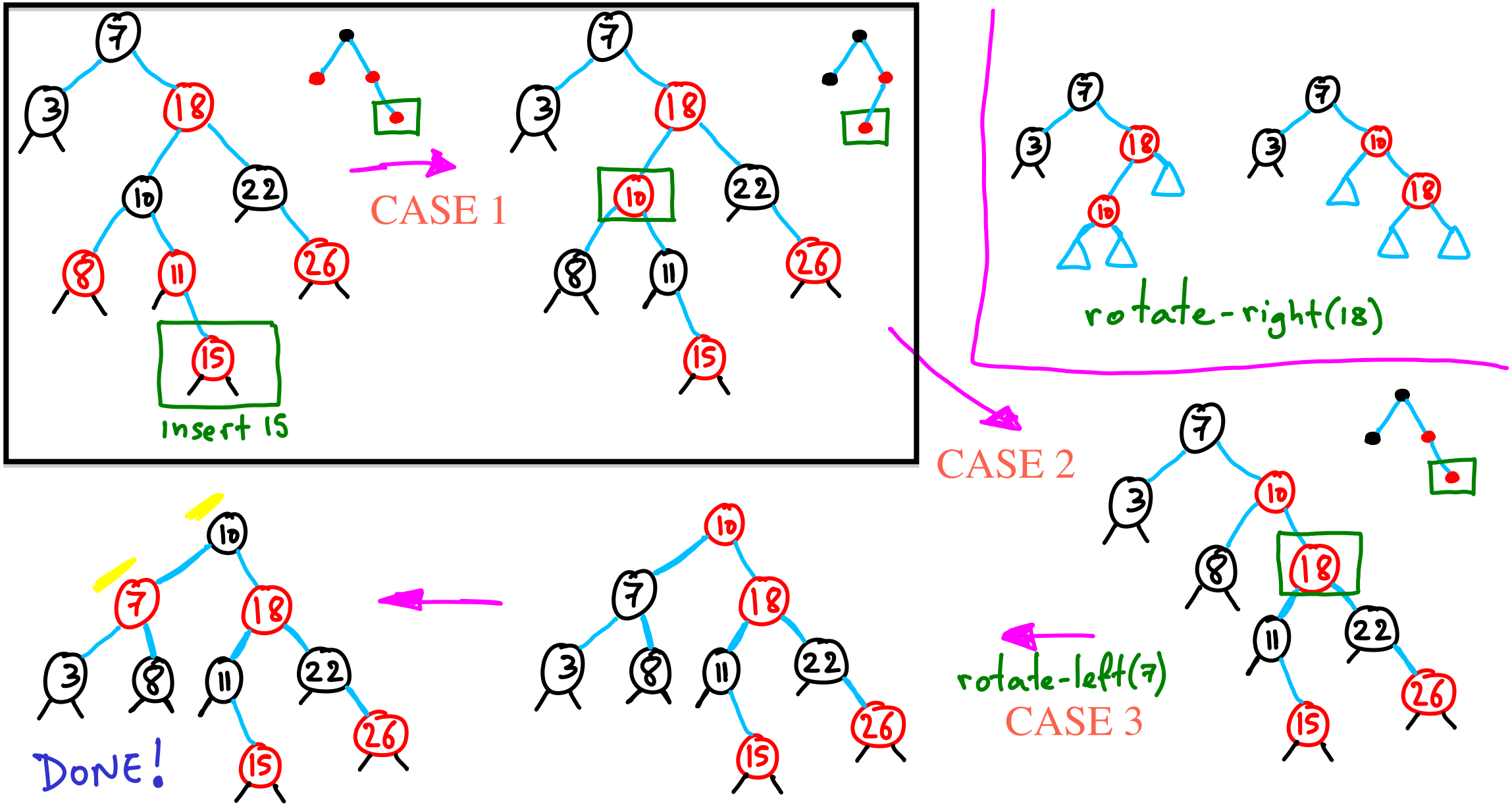
  Run CASE 3

  else \\ p(x) = right(g(x))

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

- color the root black


7
3
18
10
22
8
11
26
15
insert 15
CASE 1
rotate-right(18)
CASE 2
rotate-left(7)
CASE 3
DONE!