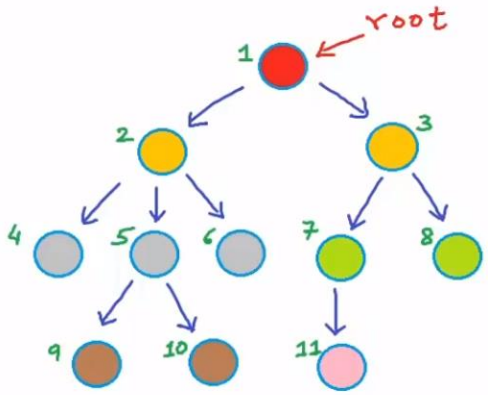
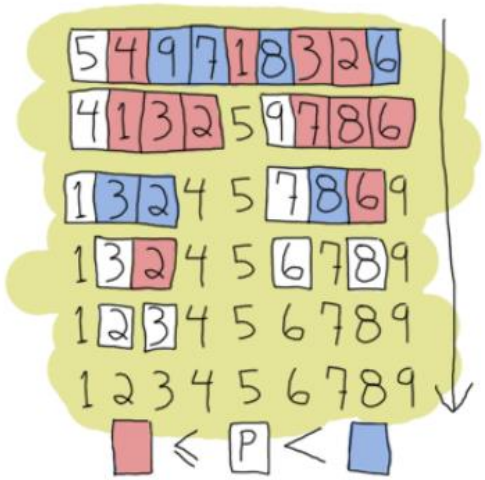
ECE 250 Data Structures & Algorithms



Red-Black Trees



Ziqiang Patrick Huang

Electrical and Computer Engineering

University of Waterloo

WATERLOO | ENGINEERING

Red-Black Trees: Another Balanced BST

- Last time: AVL trees
  - Strictly balanced
  - Balance on "way back up", through rotation
  - Pretty much requires recursion

- Another scheme: Red-black trees
  - "loosely" balanced
    - Guarantee max path length O(log(n)), but path might be longer
  - Balance on the way down, through rotation & recoloring
  - Easy to do iteratively

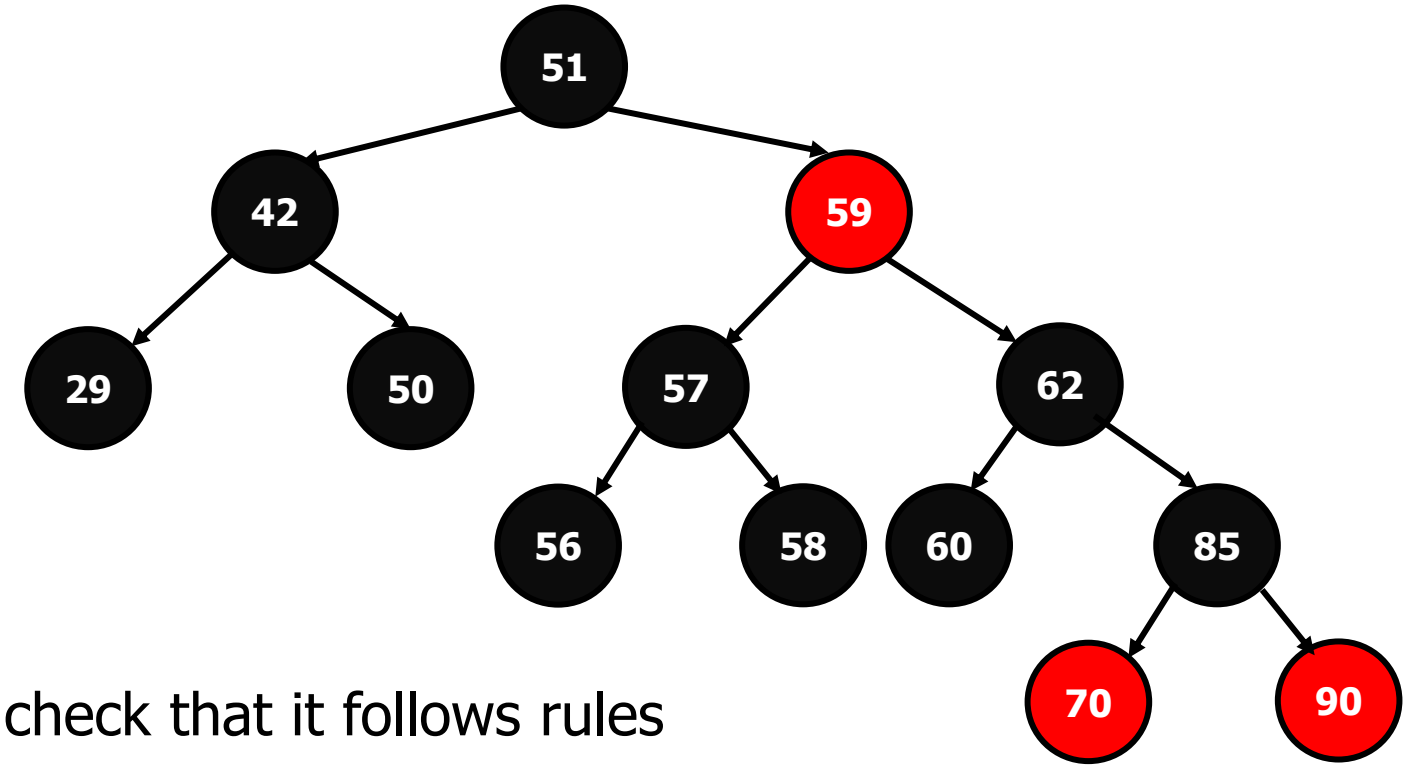
Red-Black Rules

- Red-black trees follow four rules:
  1. Every node is either red or black
  2. The root is black
  3. If a node is red, its children must be black
  4. Every path from root → NULL must have the same number of black nodes

- Why do these rules work?
  - Longest path (from root → NULL) at most 2x shortest path
    - Shortest path with N black nodes?
    - Longest path?
      - must also have exactly N black nodes(rule 4)
      - at most N red nodes(rule 3)

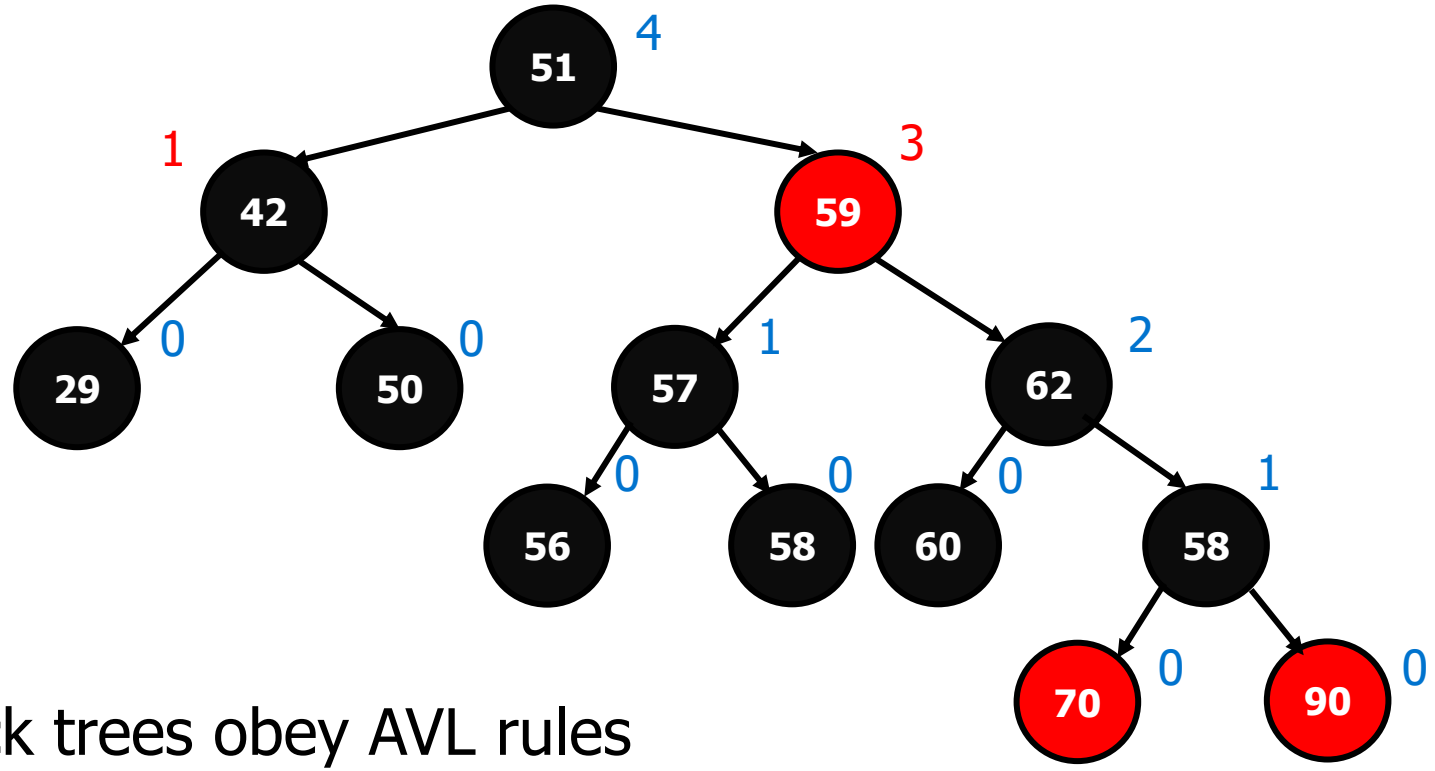
Red-Black Rules

Note: can consider
NULL nodes to be black



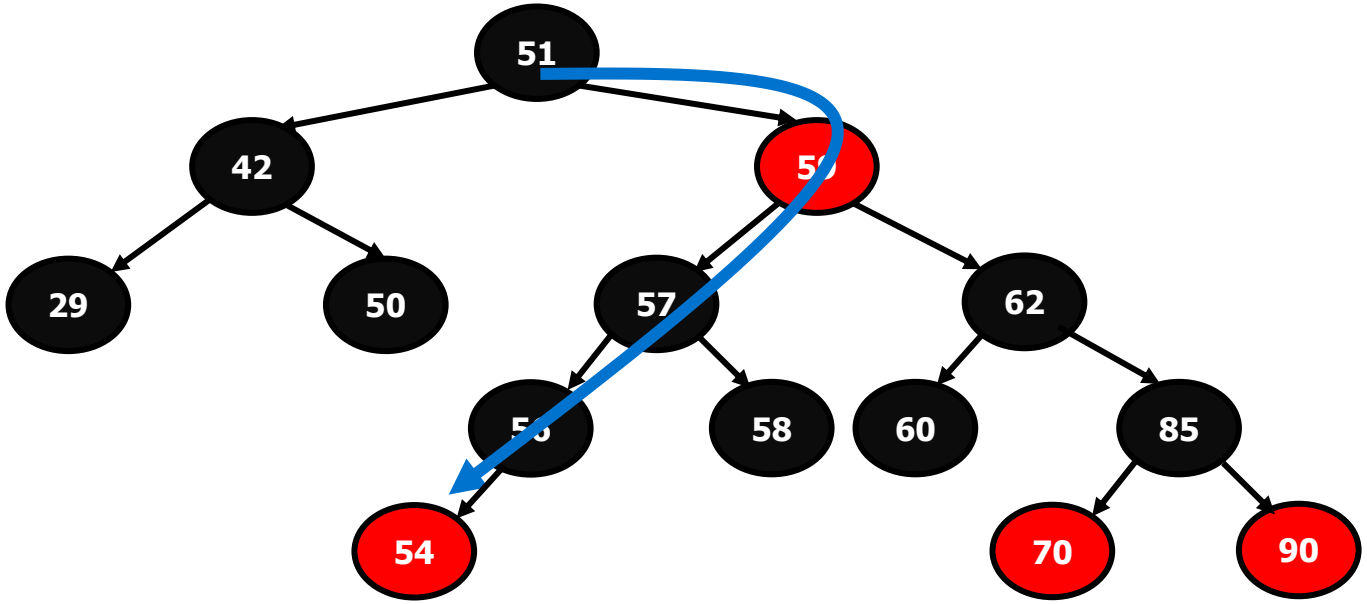
- Red Black Tree: check that it follows rules
  1. ✓ Every node is either red or black
  2. ✓ The root is black
  3. ✓ If a node is red, its children must be black
  4. ✓ Every path from root → NULL must have the same number of black nodes

Red-Black vs AVL



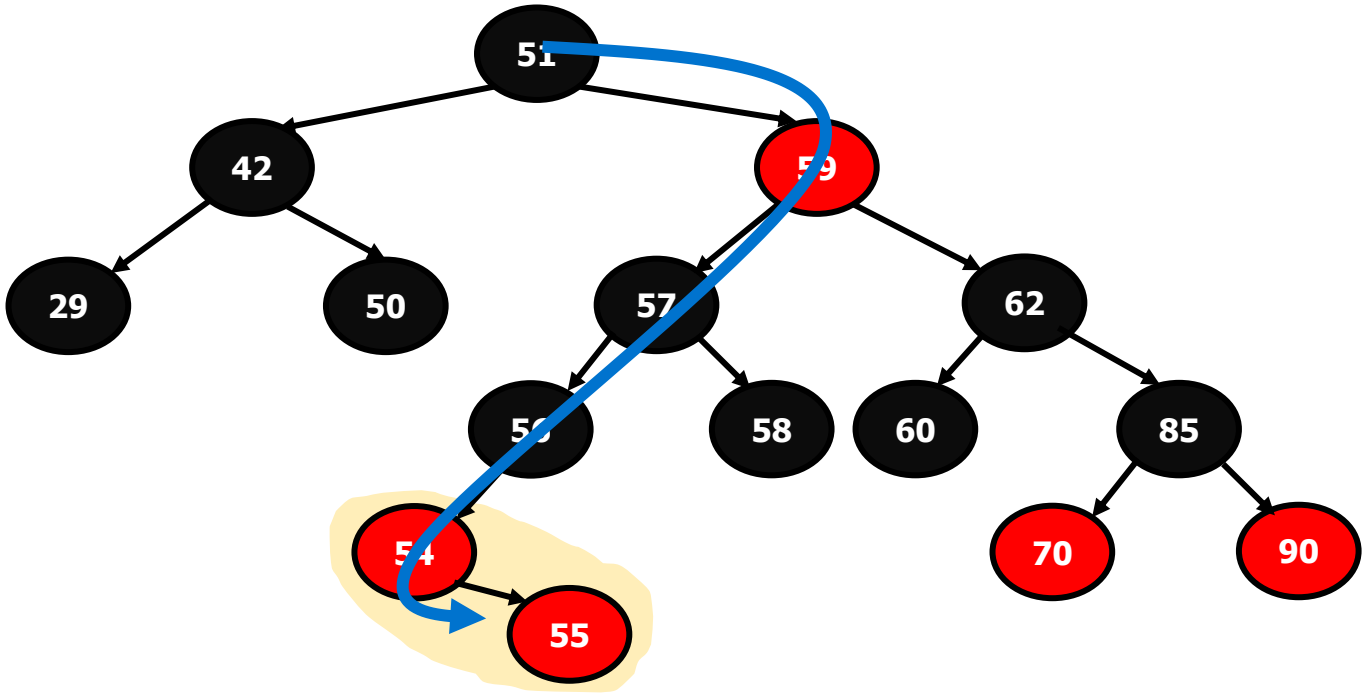
- Not all red-black trees obey AVL rules
  - The rule are weaker
  - Allows somewhat more "lopsided" tree
  - But still guarantee O(log(n)) access

Red-Black Add



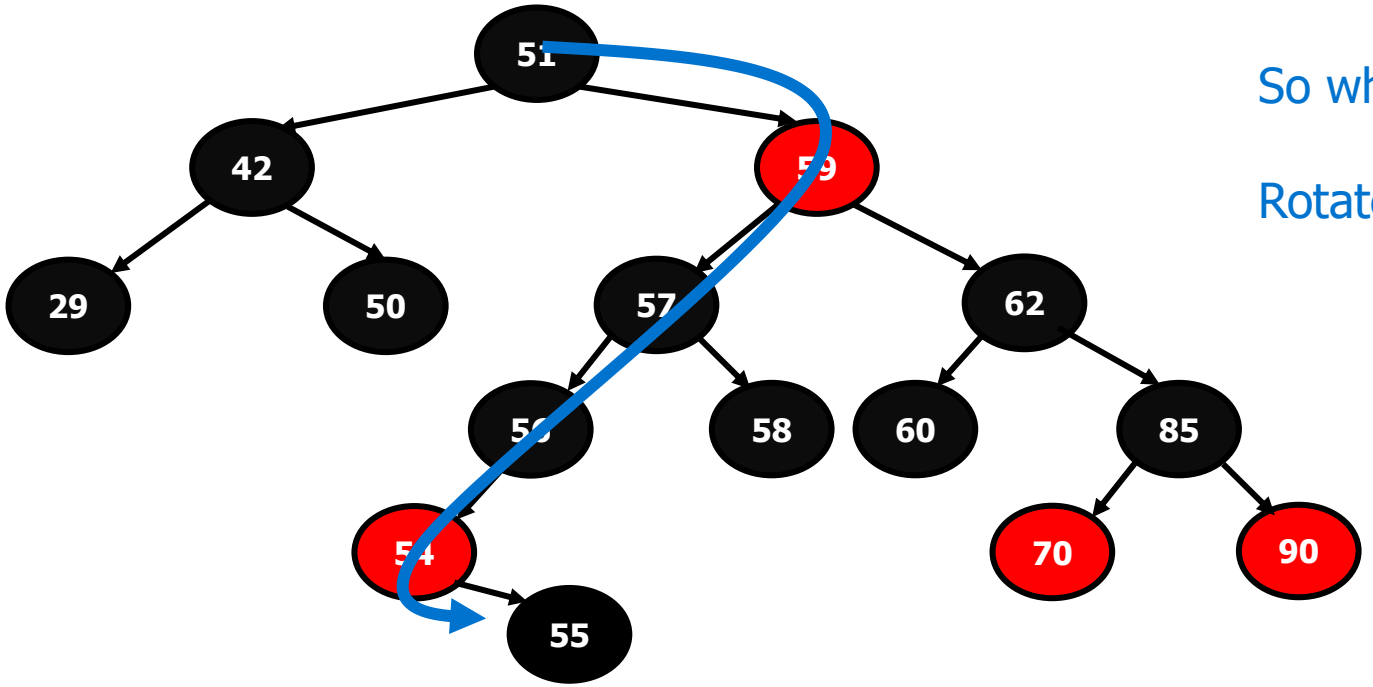
- Suppose we wanted to add 54…
  - We find the place to add "normally" (follow the rules of a BST)
  - We can add as a red node here without violating any rules
    - In fact, we ALWAYS have to add as red… why?
    - If we add as **black**, that path has more black than any other
  - So what is wrong with just always adding as red?

Red-Black Add



- Suppose we wanted to add 55…
  - We find the place to add "normally" (follow the rules of a BST)
  - But if we add 55 as red…
    - Now we have a red node with a red child (violates rule 3)

Red-Black Add

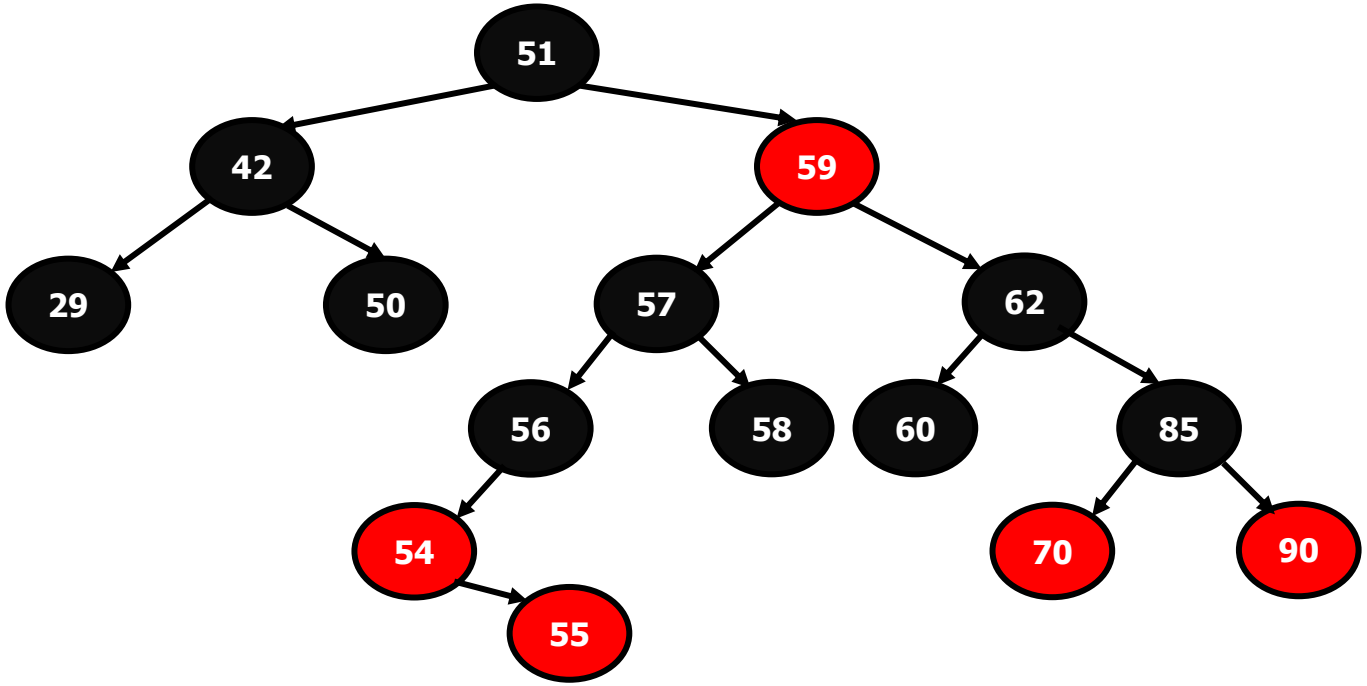


So what do we do?

Rotate and re-color!

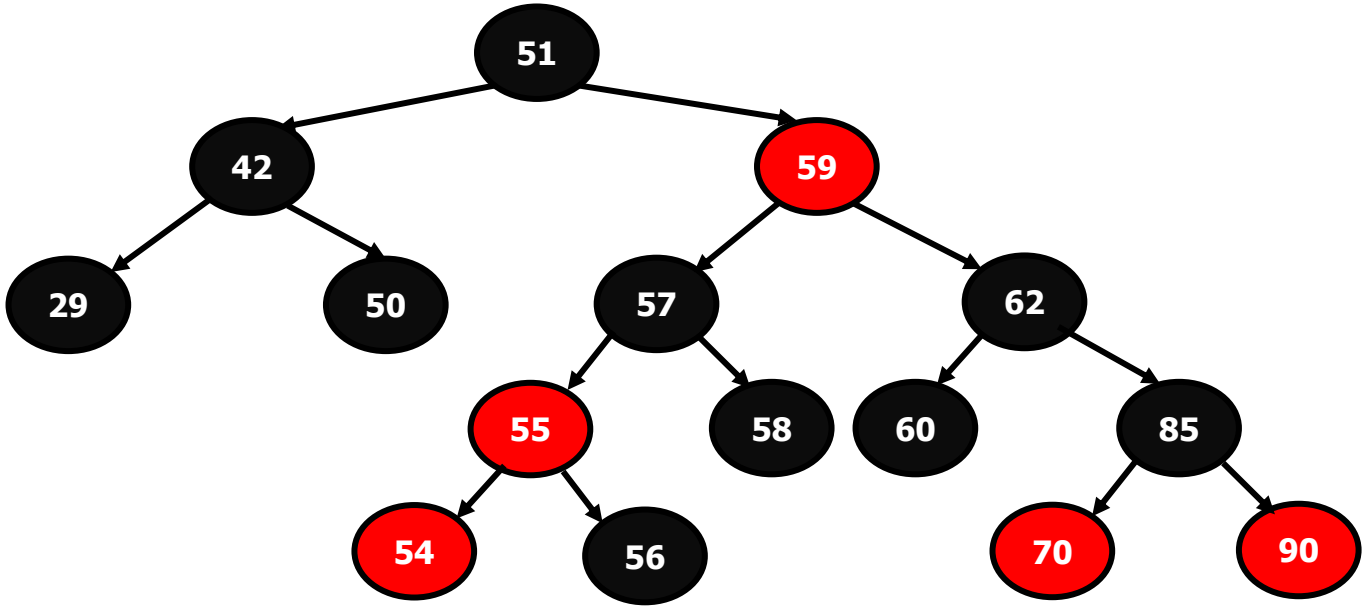
- Suppose we wanted to add 55…
  - We can't add 55 as black either… (always add as red)
    - More black nodes on that path than others…

Red-Black Add



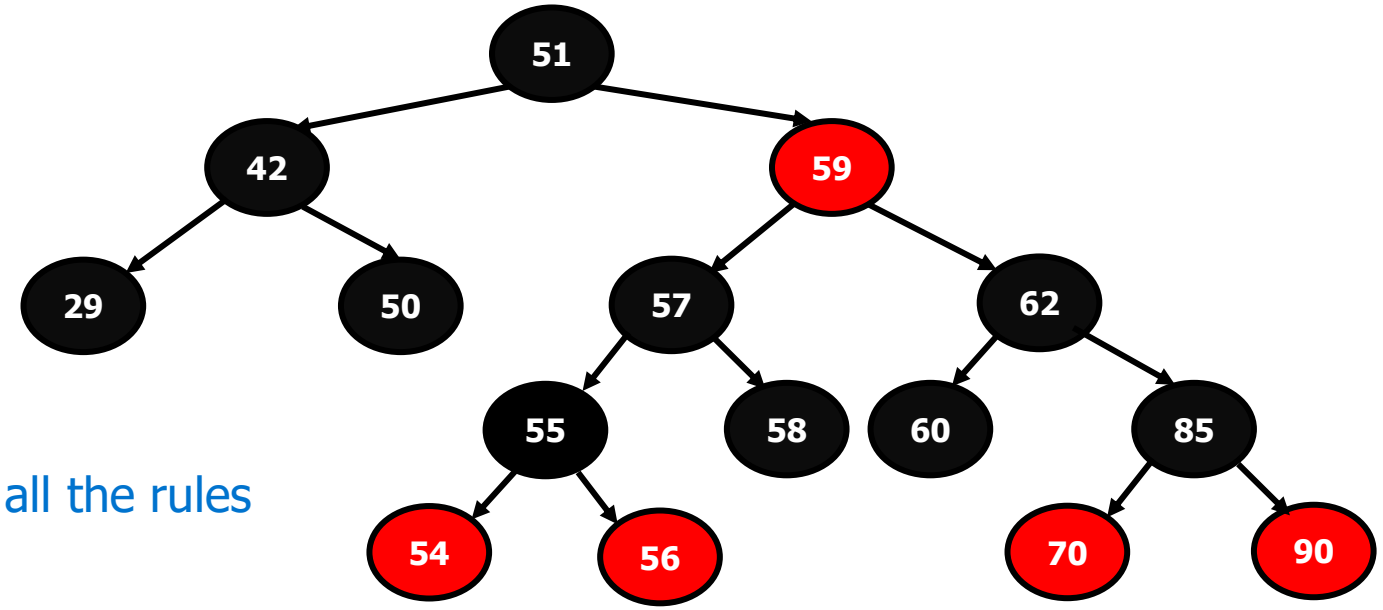
- Start with the idea of adding 55 as red (always add as red)
  - The extra red corresponds to an imbalance in the tree
  - Rotate (like AVL rotate)

Red-Black Add



- Start with the idea of adding 55 as red (always add as red)
  - The extra red corresponds to an imbalance in the tree
  - Rotate (like AVL rotate)

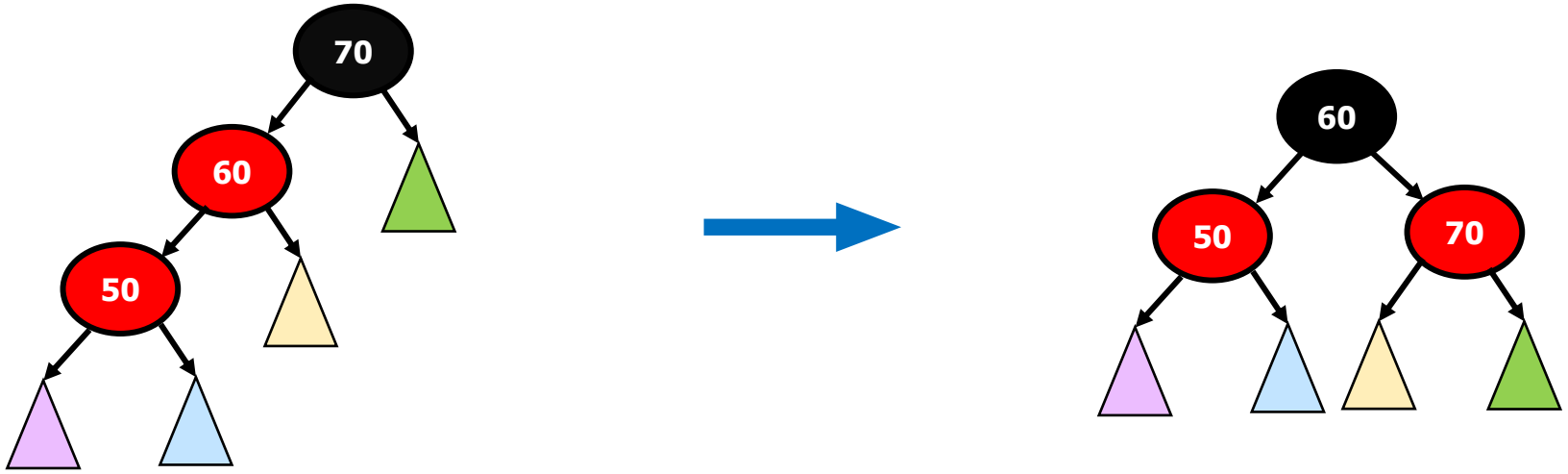
Red-Black Add



This tree now follows all the rules

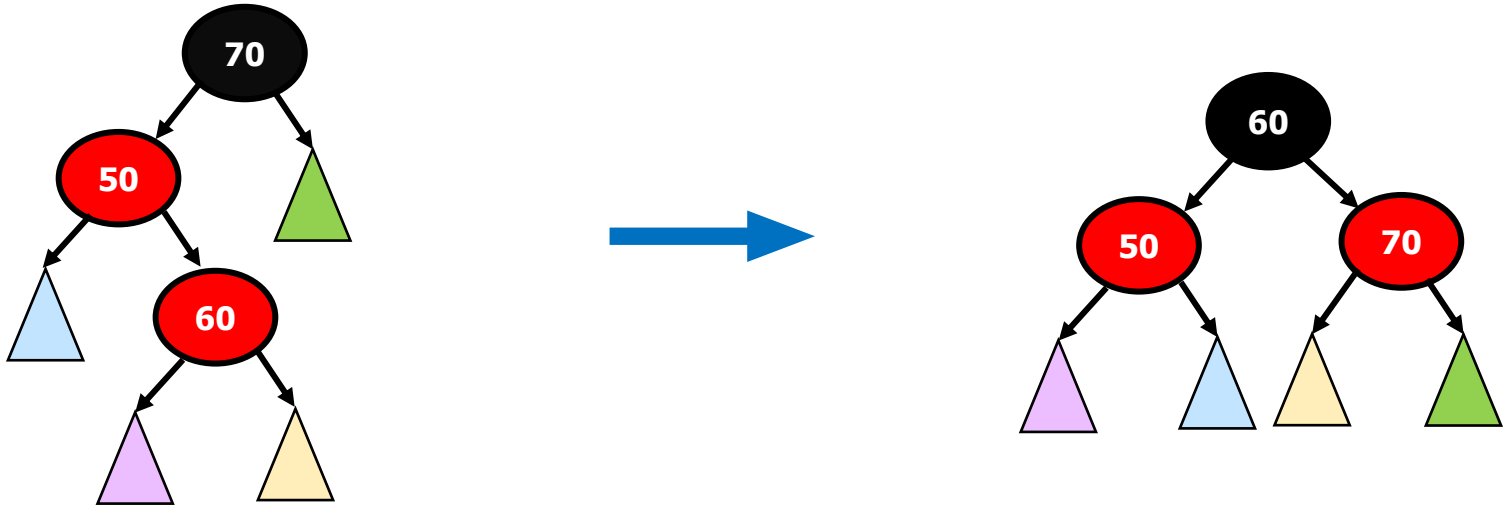
- Start with the idea of adding 55 as red (always add as red)
  - The extra red corresponds to an imbalance in the tree
  - Rotate (like AVL rotate)
  - Then re-color

Red-Black Rotations



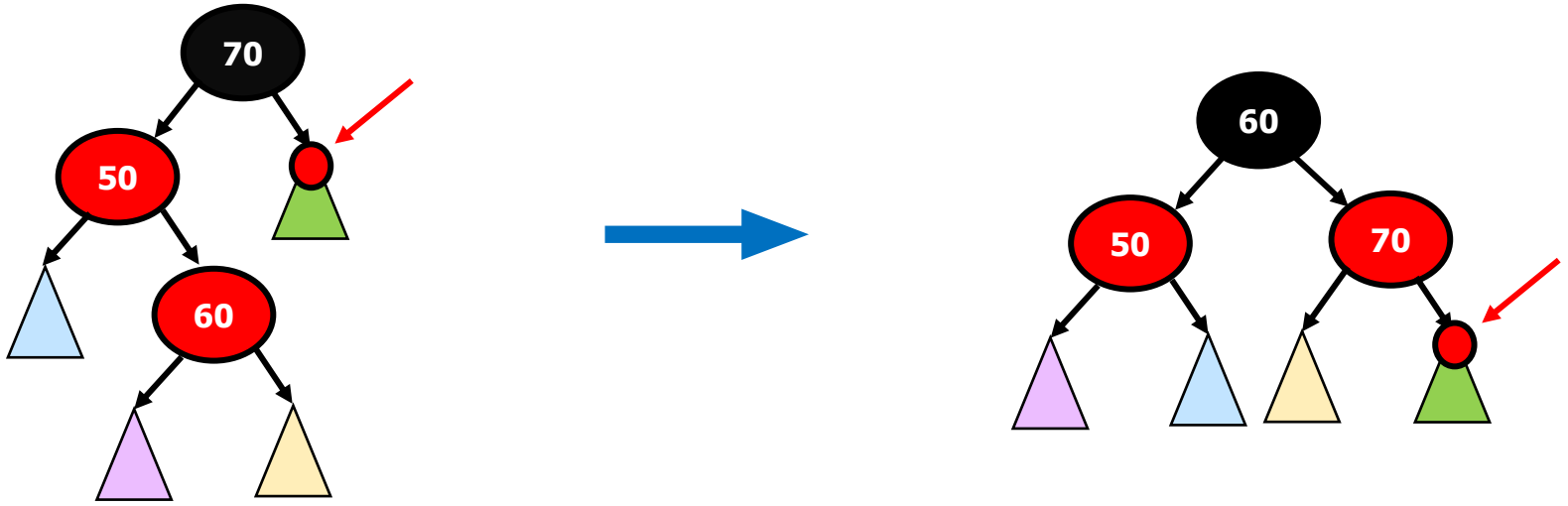
- When to rotate?
  - Anytime we have two red nodes in a row
  - Rotation just like AVL, except we change colors two

Red-Black Rotations



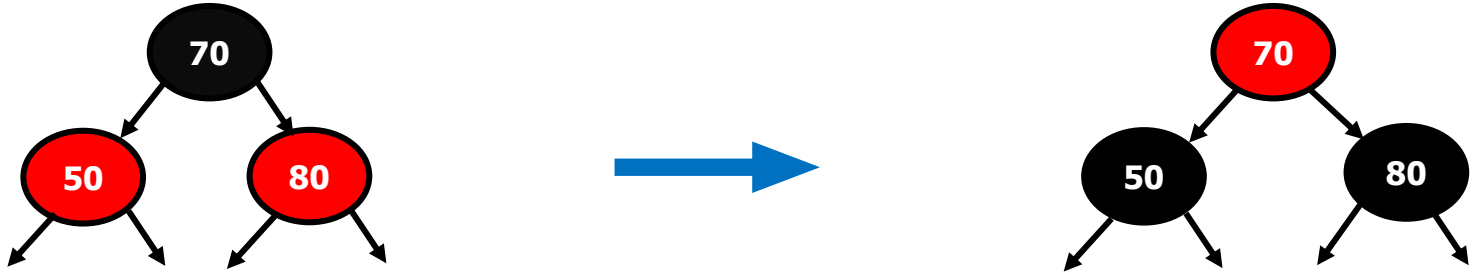
- Sometimes need to do double rotation
  - But again, just like AVL

Red-Black Rotations



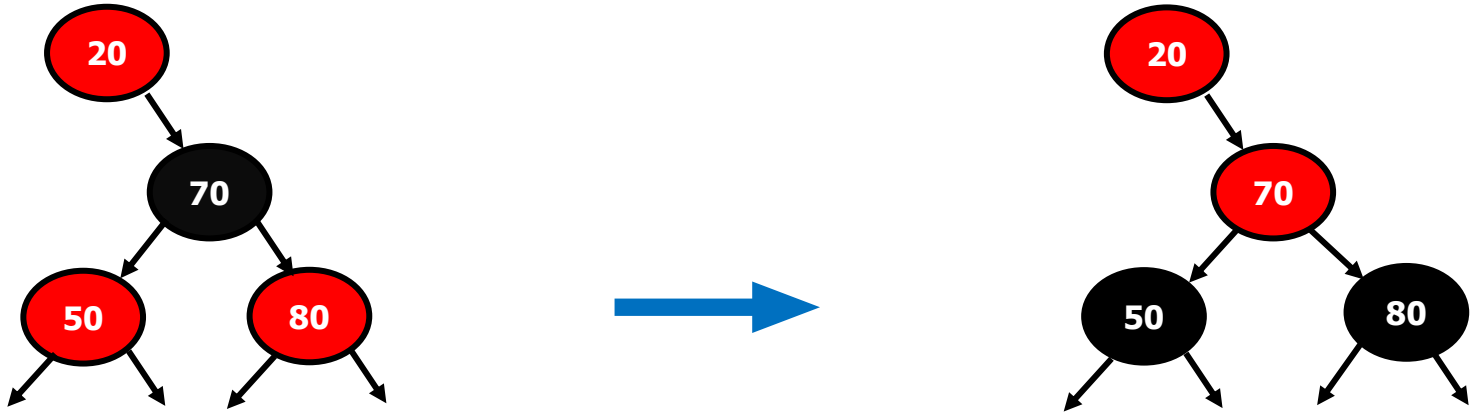
- But what happens if 70's right sub-tree has a red root
  - Now we have two red nodes in a row!

Red-Black Rotations



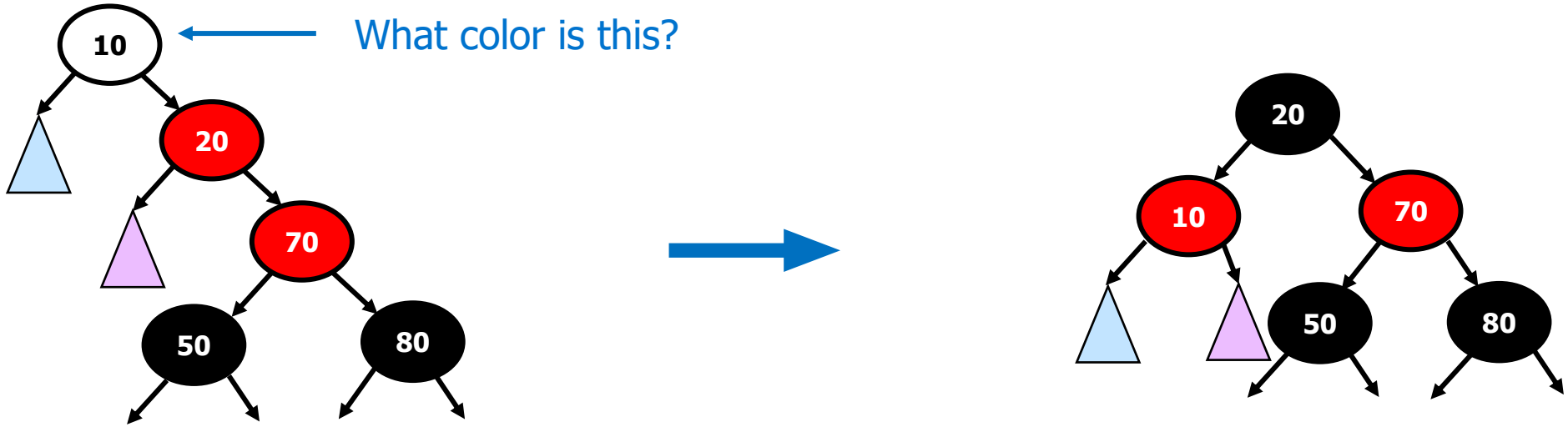
- Prevent this situation in advance
  - When we encounter a black node with 2 red children
    - Re-color the black node to red
    - And the children to black
  - Preserves the black count down the paths (maintains rule 4)

Red-Black Rotations



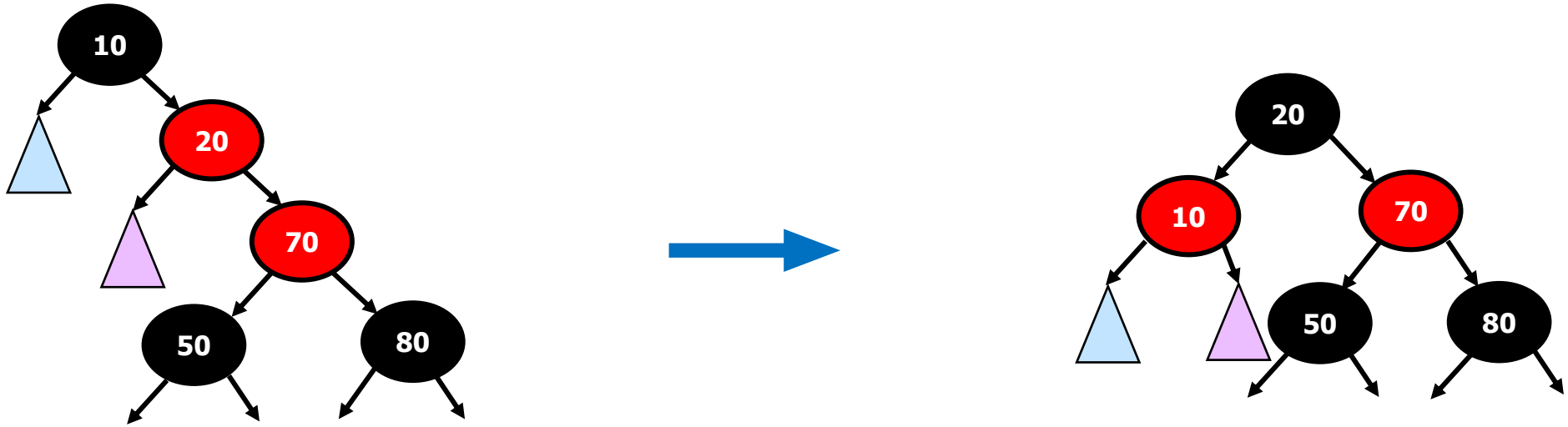
- But … We may have just introduced a rule 3 violation
  - What if the original black node has a red parent?
  - Two red nodes in a row …
  - Now what do we do? Rotate and recolor!

Red-Black Rotations



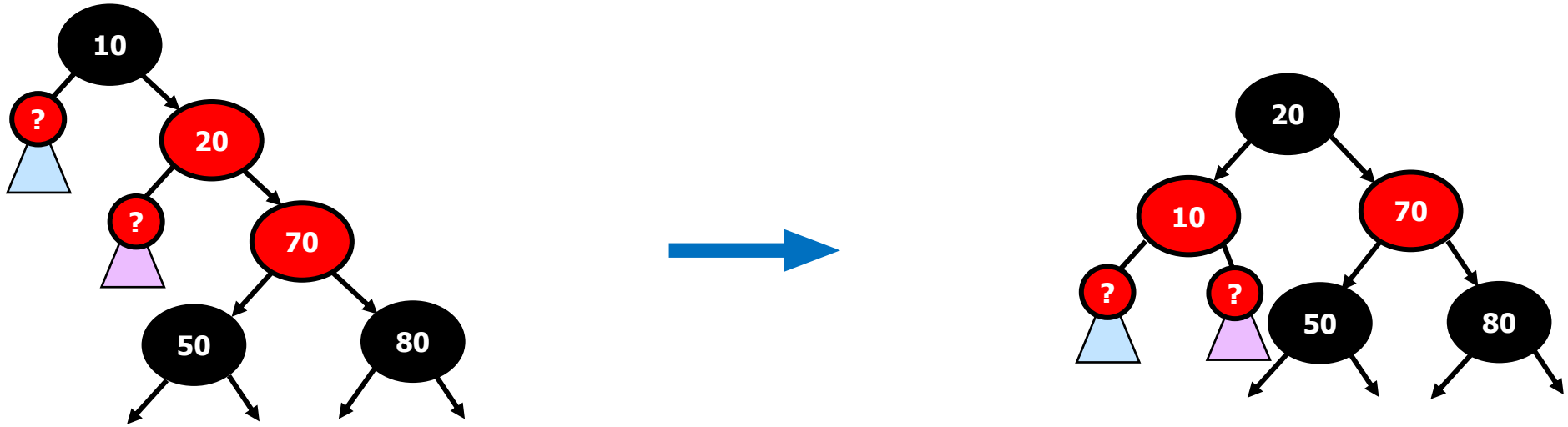
- How do we know that our rotate & recolor is legal?
  - First question: does it preserve the black node count on the paths?

Red-Black Rotations



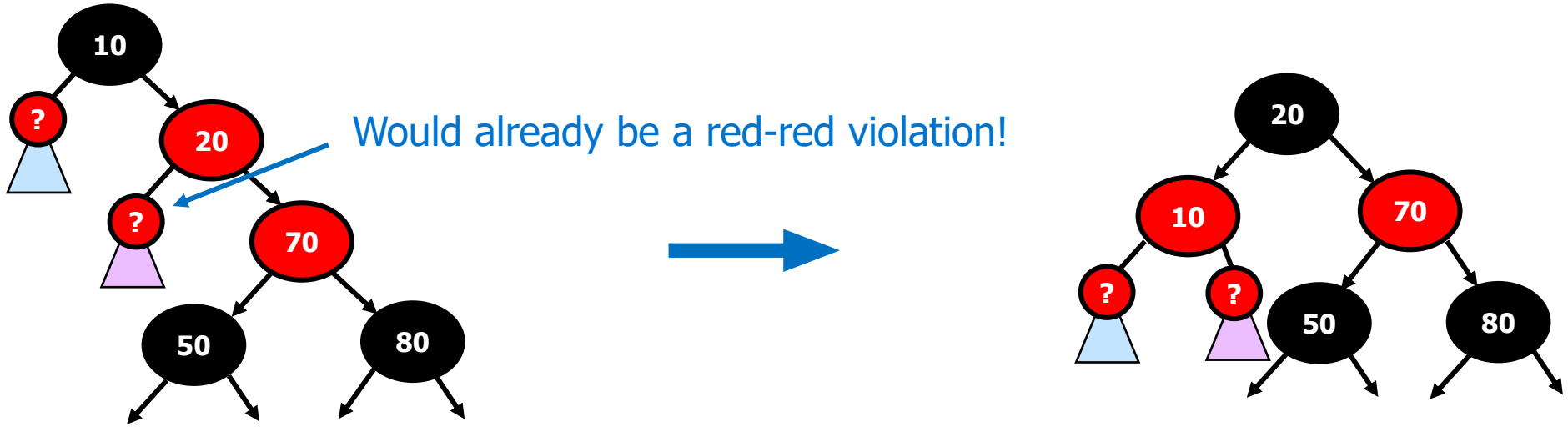
- How do we know that our rotate is legal?
  - First question: does it preserve the black node count on the paths?
  - Node at top of the rotate (10) must be black: why?
    - Node in the middle (20) was already red

Red-Black Rotations



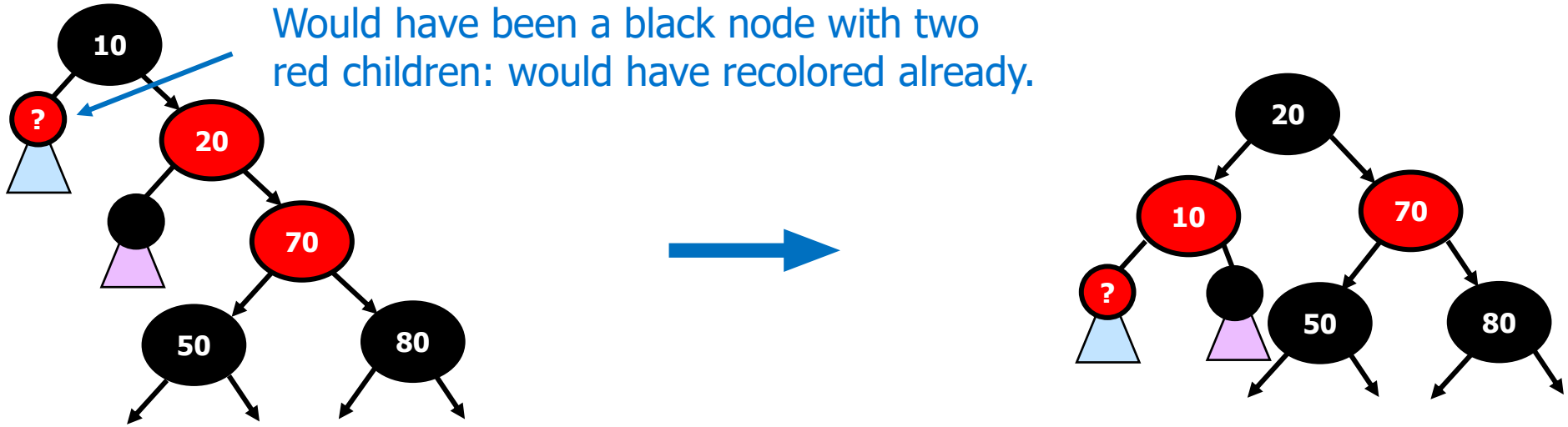
- How do we know that our rotate is legal?
  - Second question: Did we introduce other red-red violations?
    - Are either of 10's new children red?

Red-Black Rotations



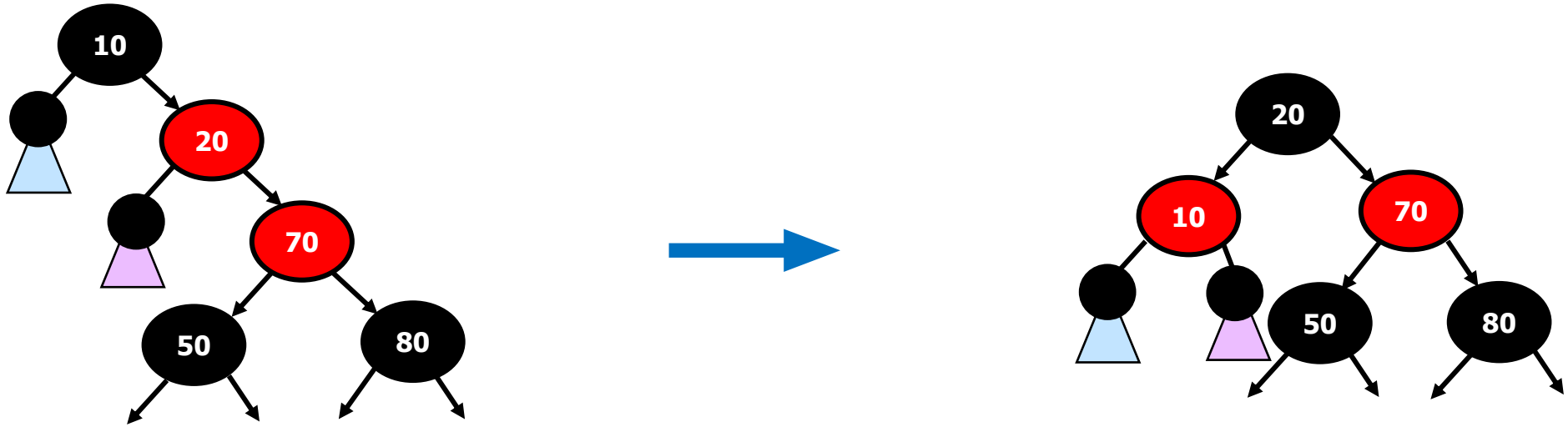
- How do we know that our rotate is legal?
  - Second question: Did we introduce other red-red violations?
    - Are either of 10's new children red?

Red-Black Rotations



- How do we know that our rotate is legal?
  - Second question: Did we introduce other red-red violations?
    - Are either of 10's new children red?

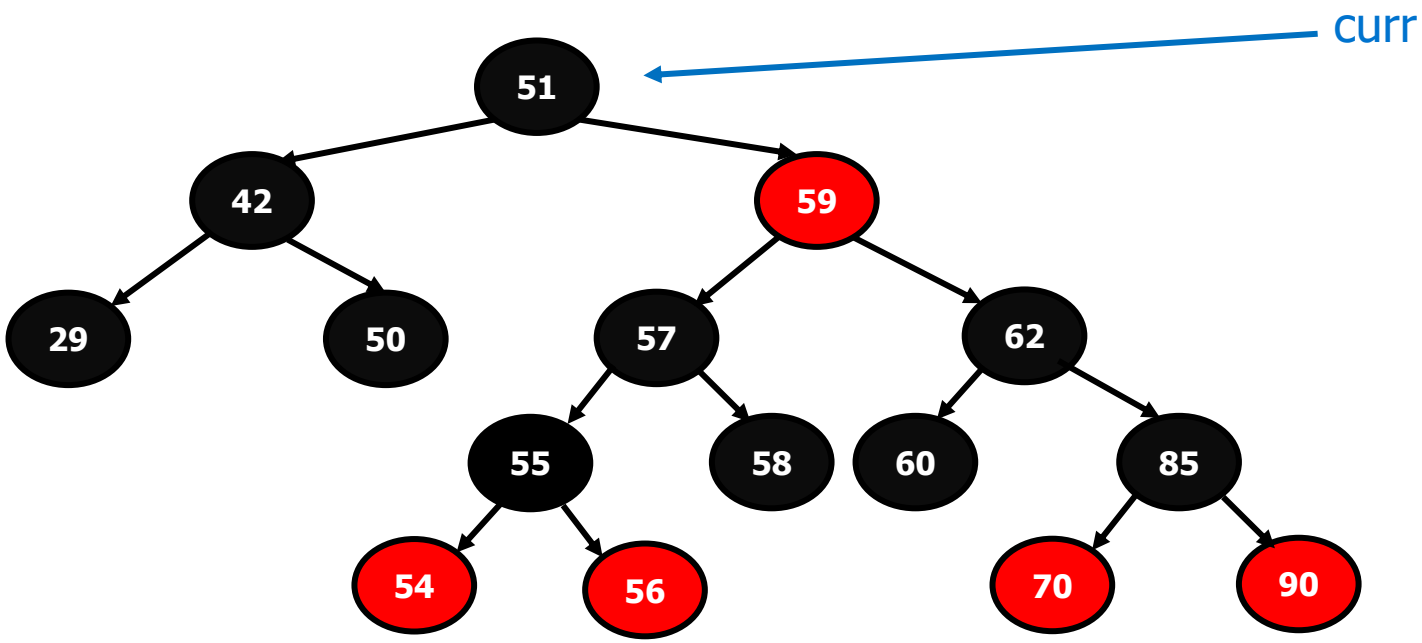
Red-Black Rotations



- How do we know that our rotate is legal?
  - So we have set things up to ensure it is legal!

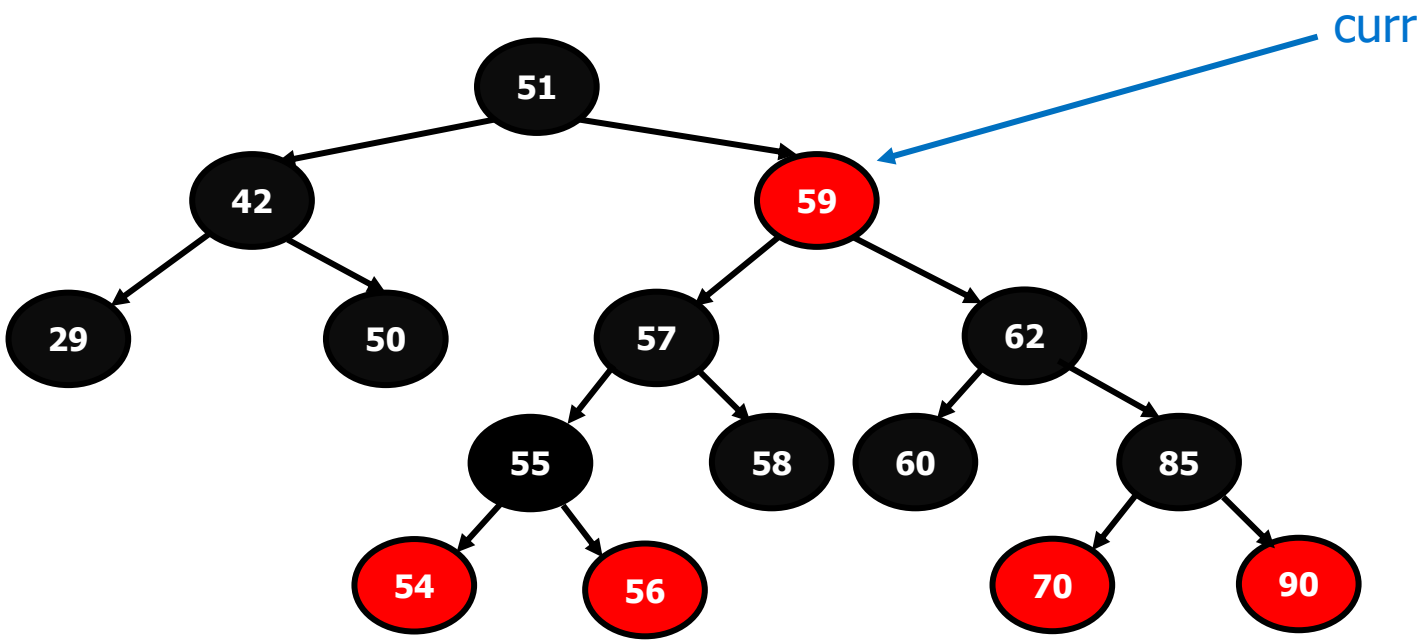
Putting It Together

Add 93



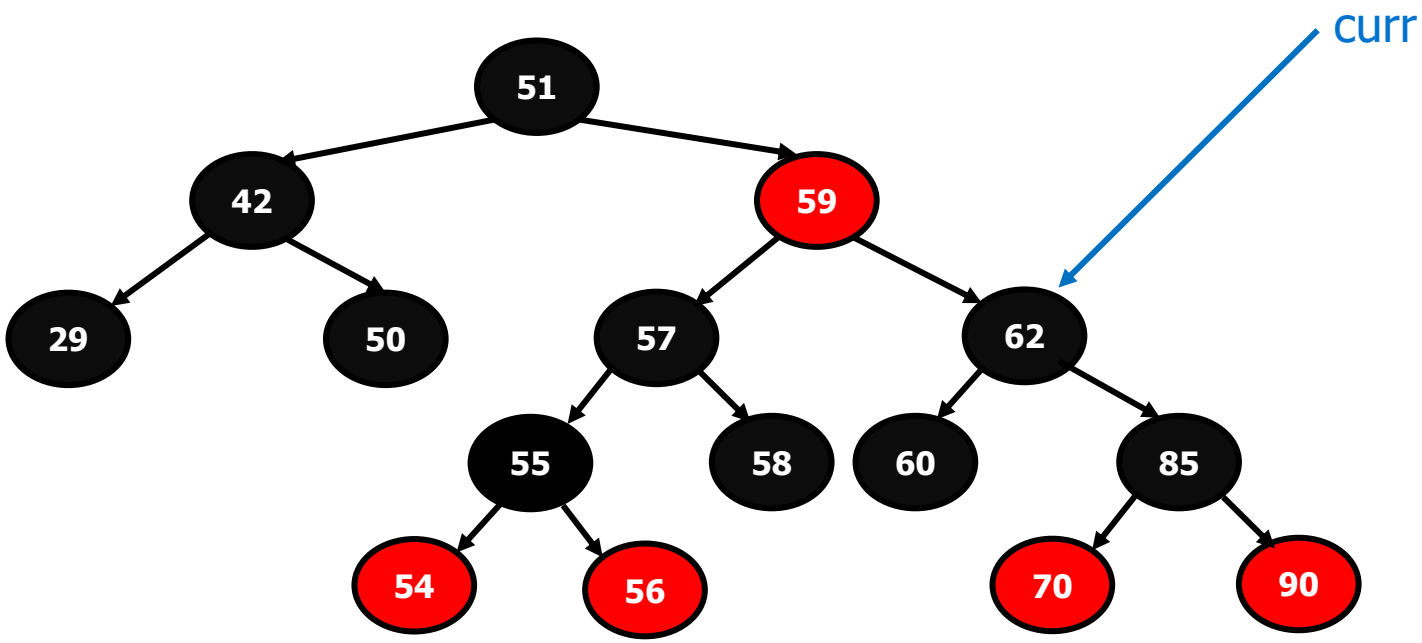
Putting It Together

Add 93



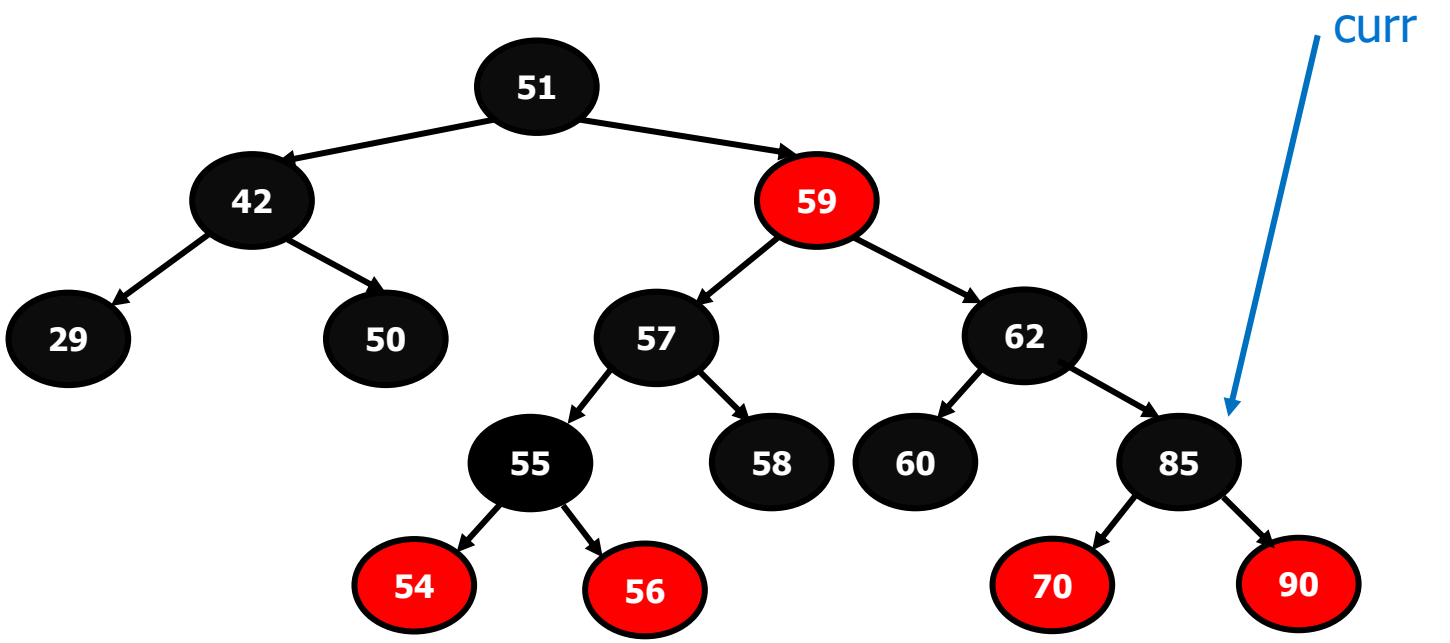
Putting It Together

Add 93



Putting It Together

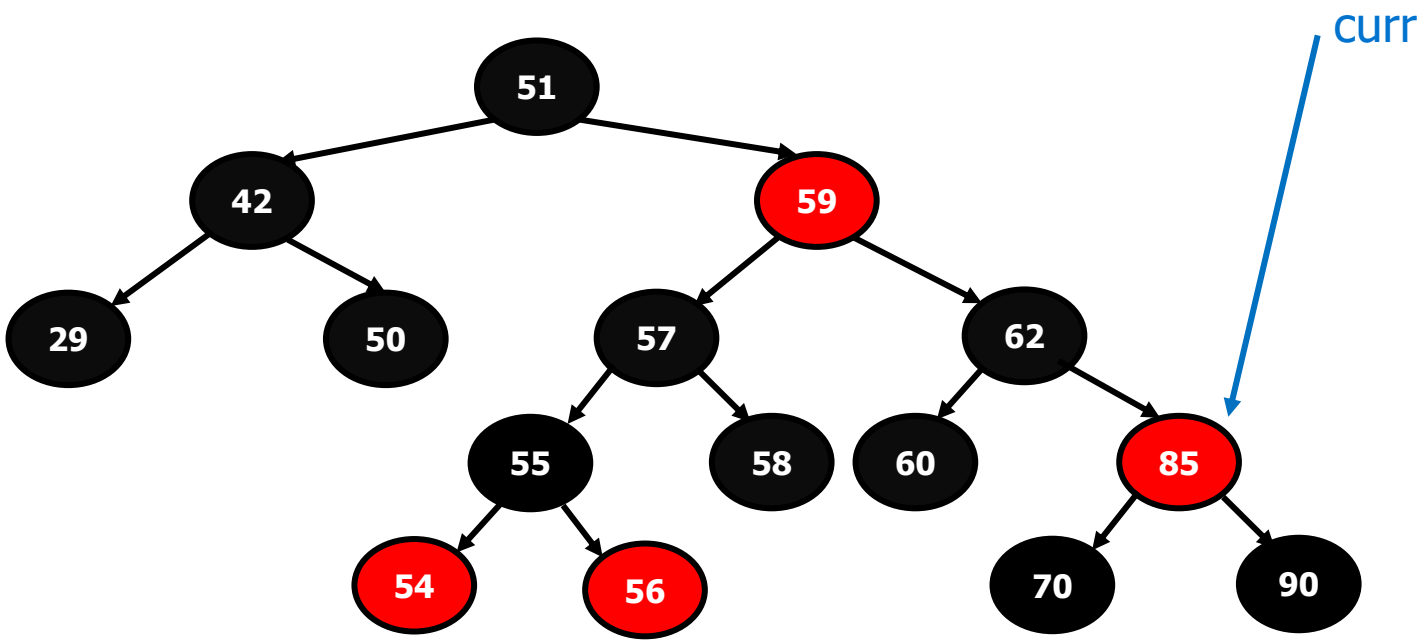
Add 93



Black node, two children!

Putting It Together

Add 93

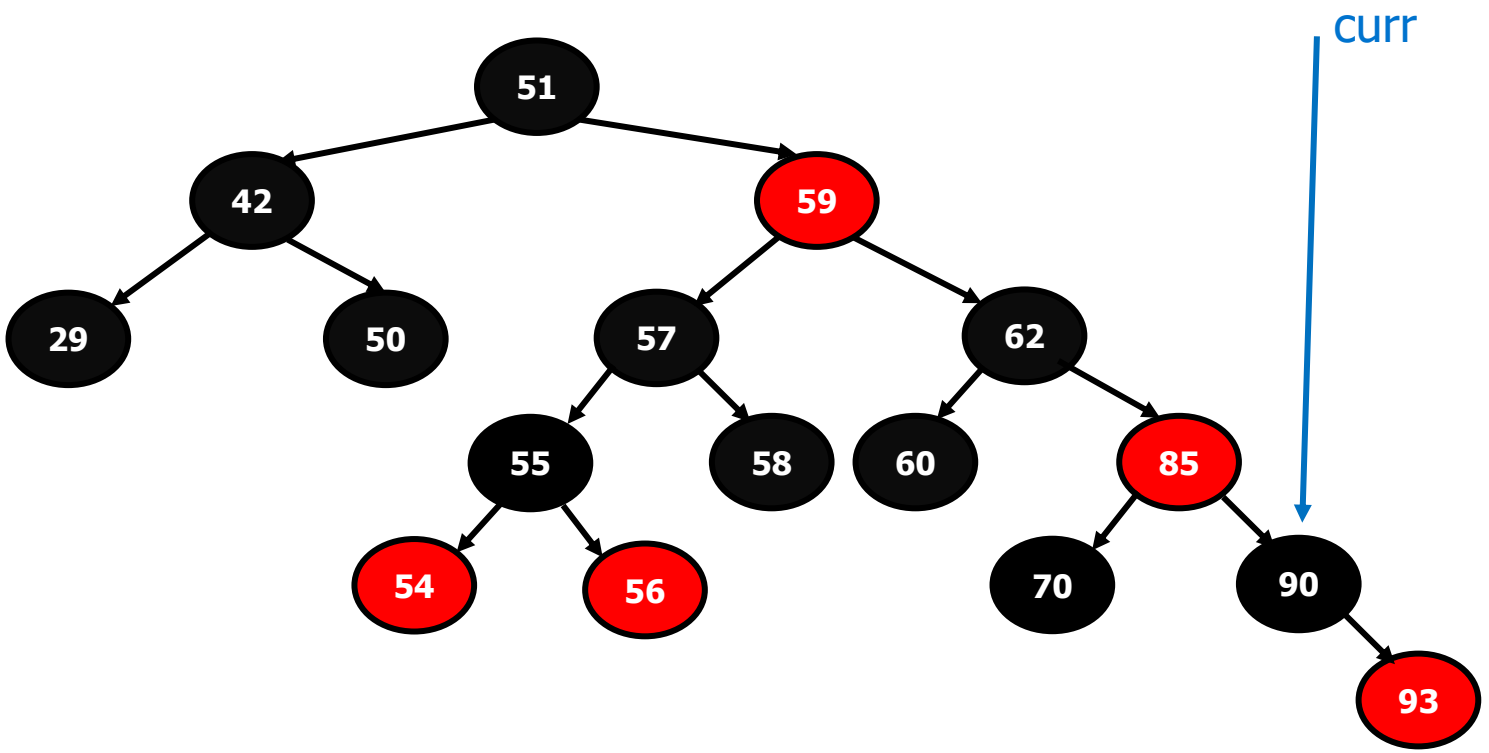


Recolor to a red with 2 black children

85's parent (62) is black, no violations
(continue)

Putting It Together

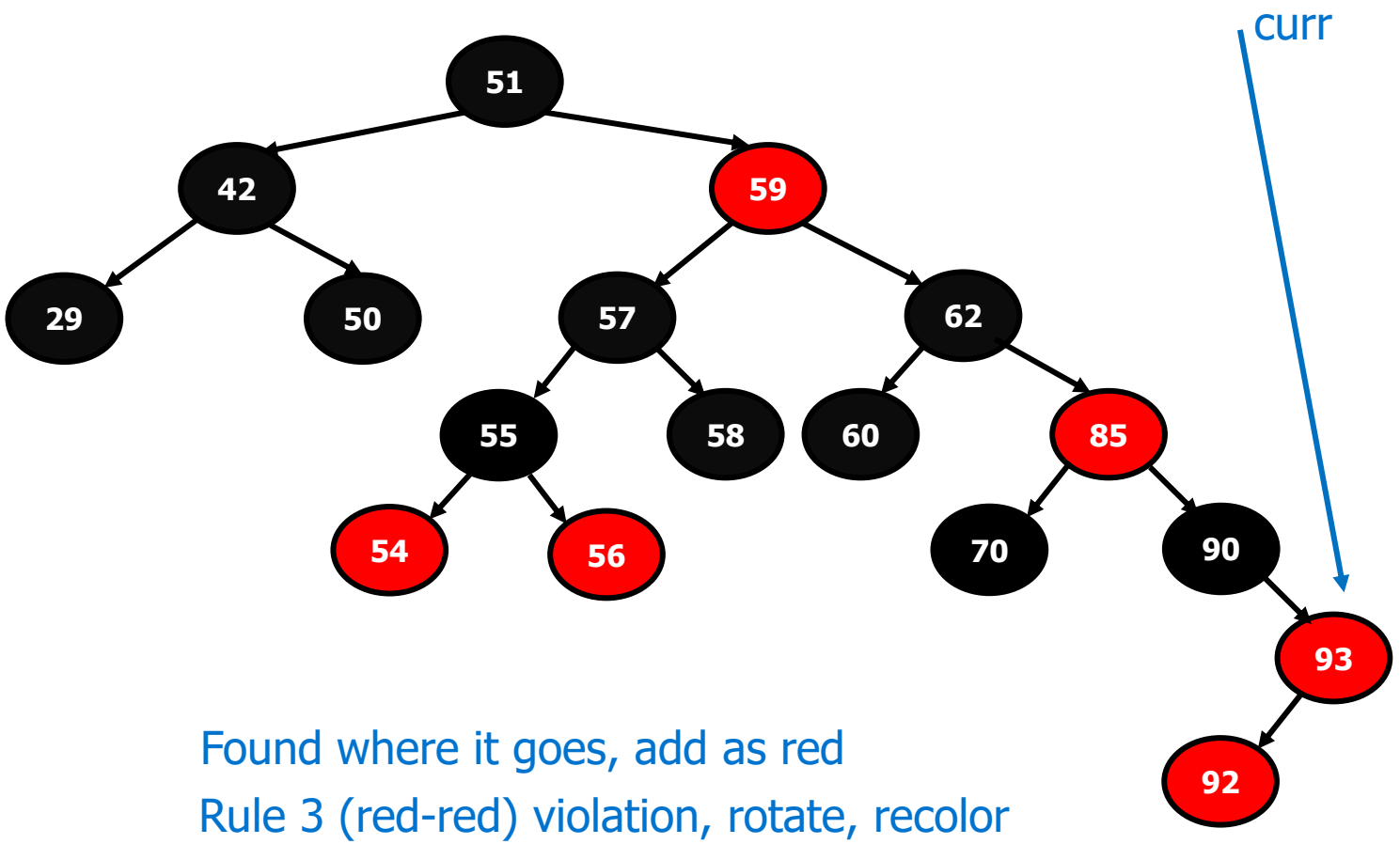
Add 93



Found where it goes, add as red

Putting It Together

Now, add 92

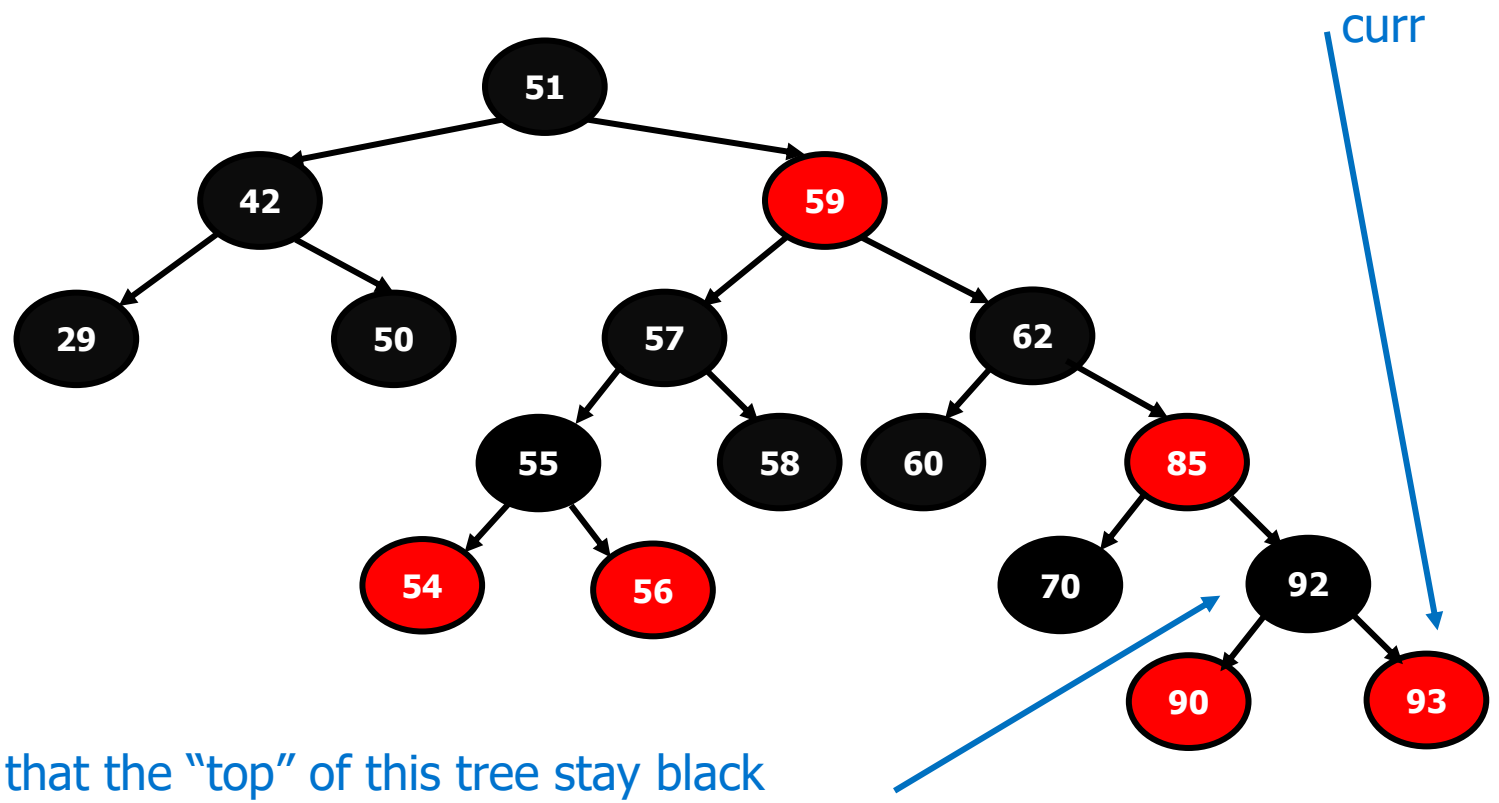


Found where it goes, add as red

Rule 3 (red-red) violation, rotate, recolor

Putting It Together

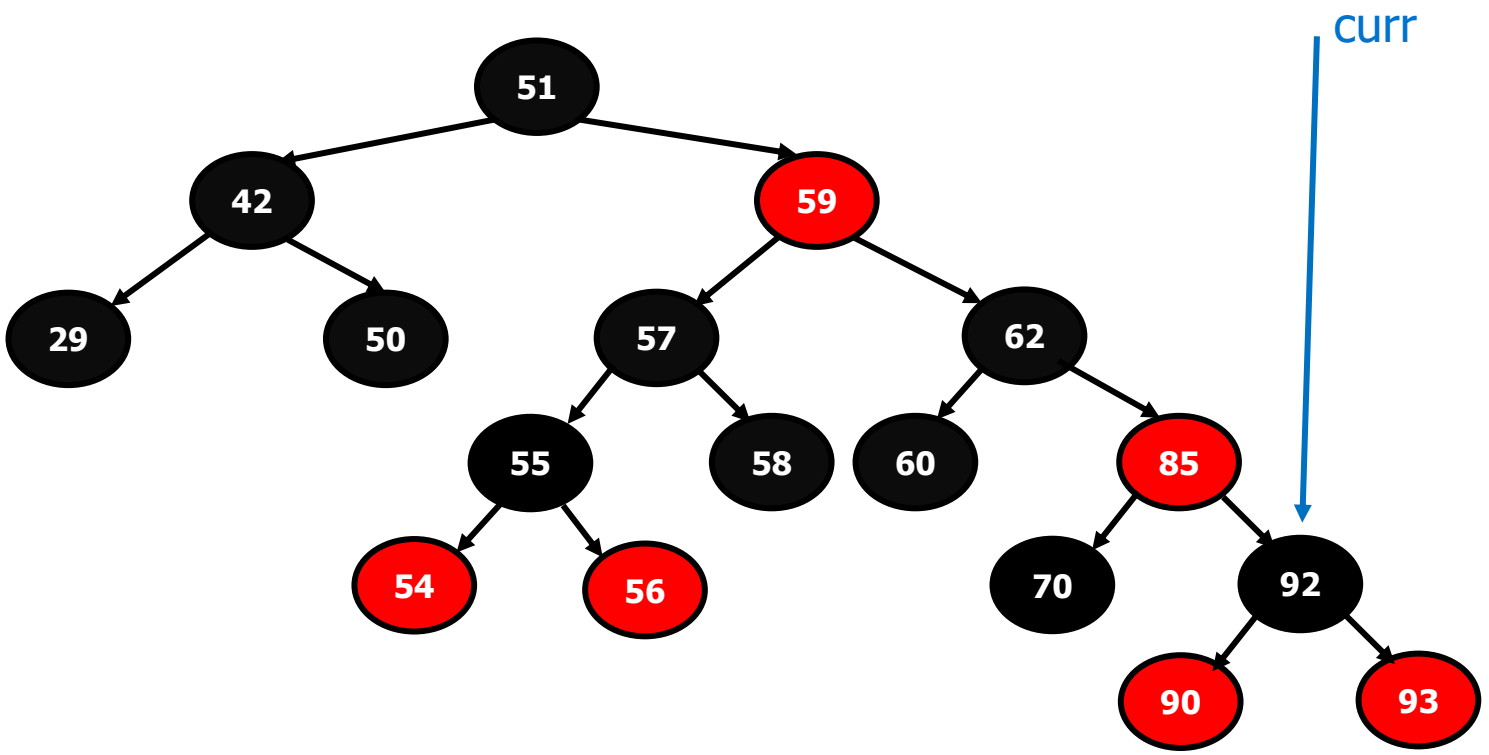
Now, add 92



Note that the "top" of this tree stay black
No possibility of needing to rotate above here

Putting It Together

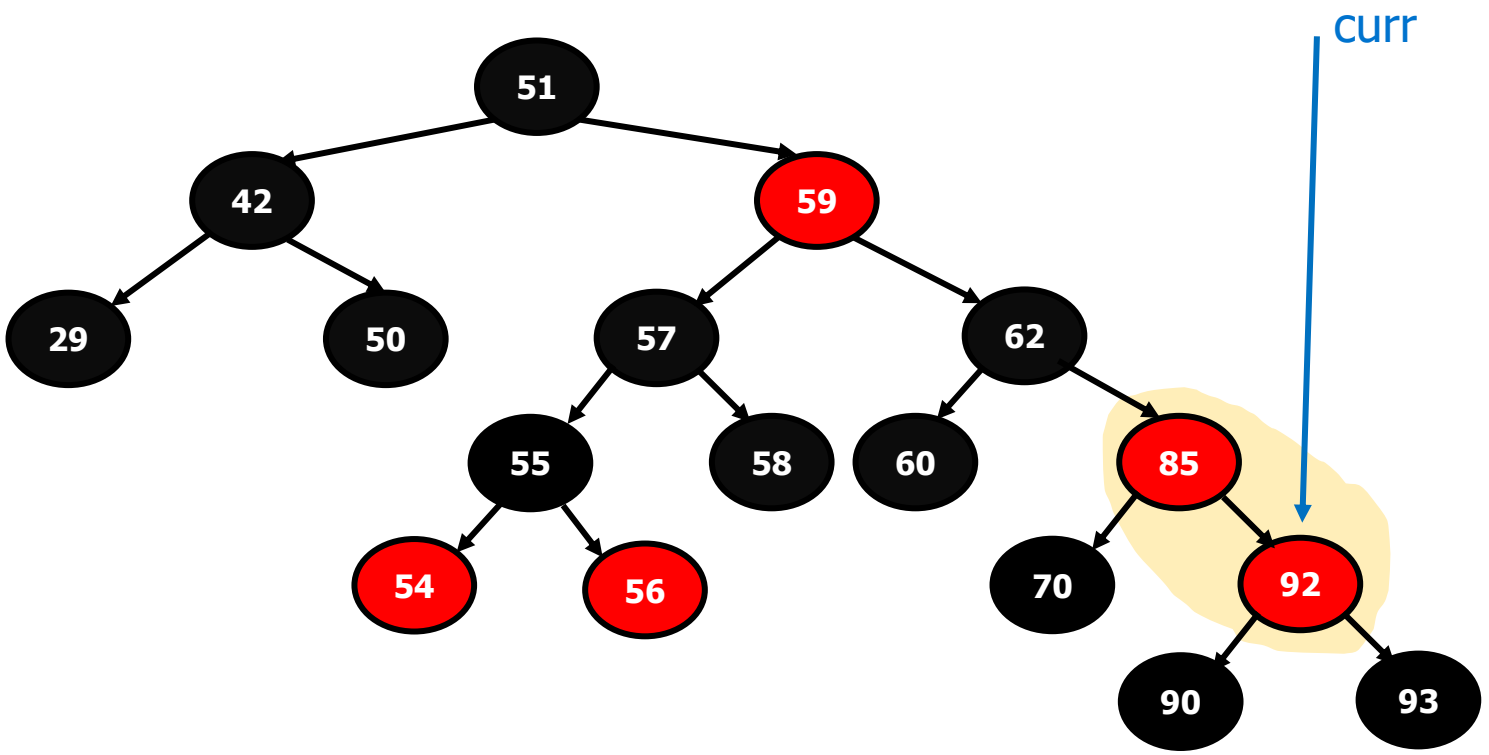
Now, add 87



Black node, two red children!

Putting It Together

Add 87



Black node, two red children!

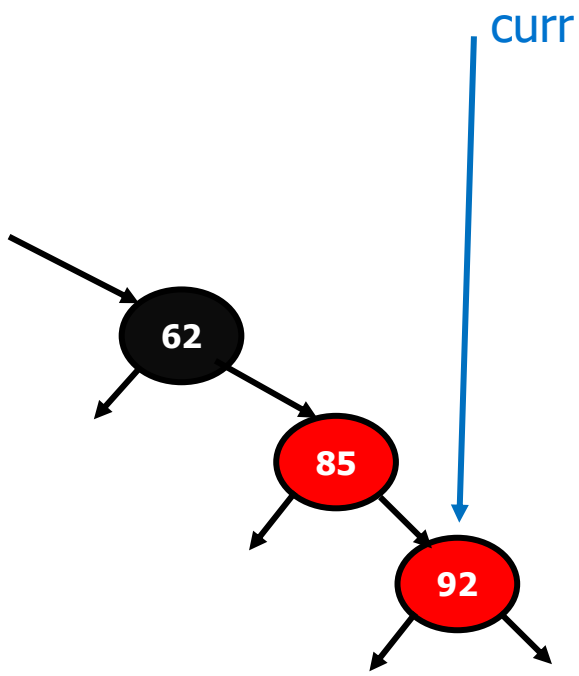
But now we have a rule 3 violation!
Need to rotate

Putting It Together

Add 87

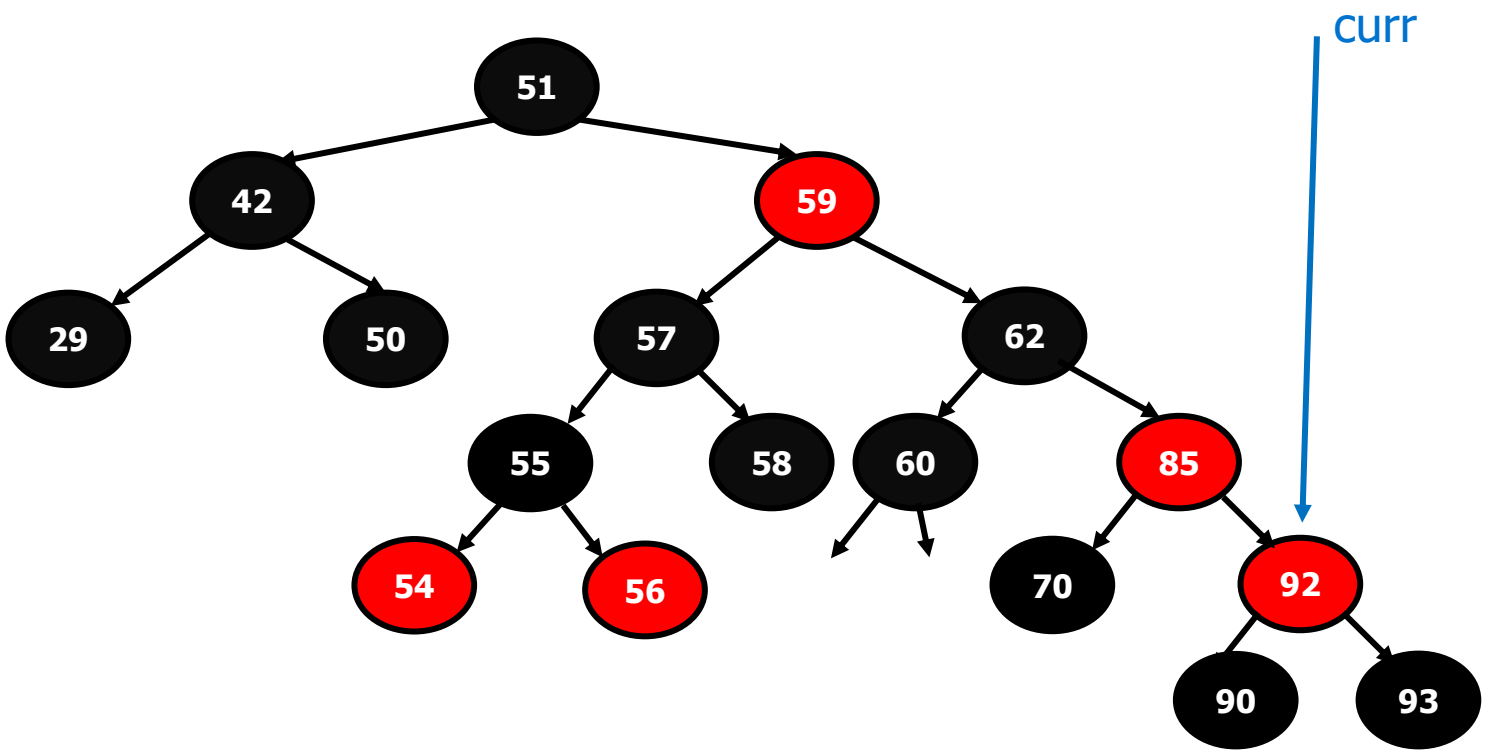
Imbalance is right -> right
- As if we added 92 in AVL

Need a single left rotation at 62



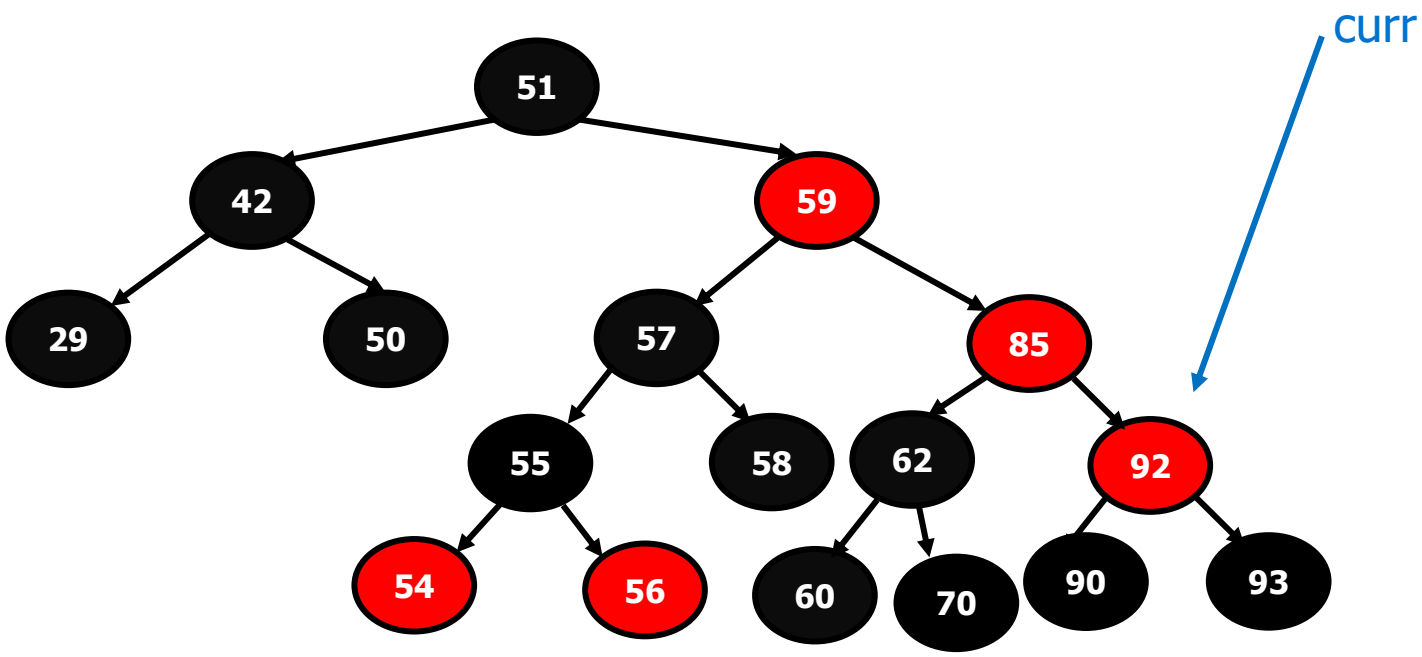
Putting It Together

Add 87



Putting It Together

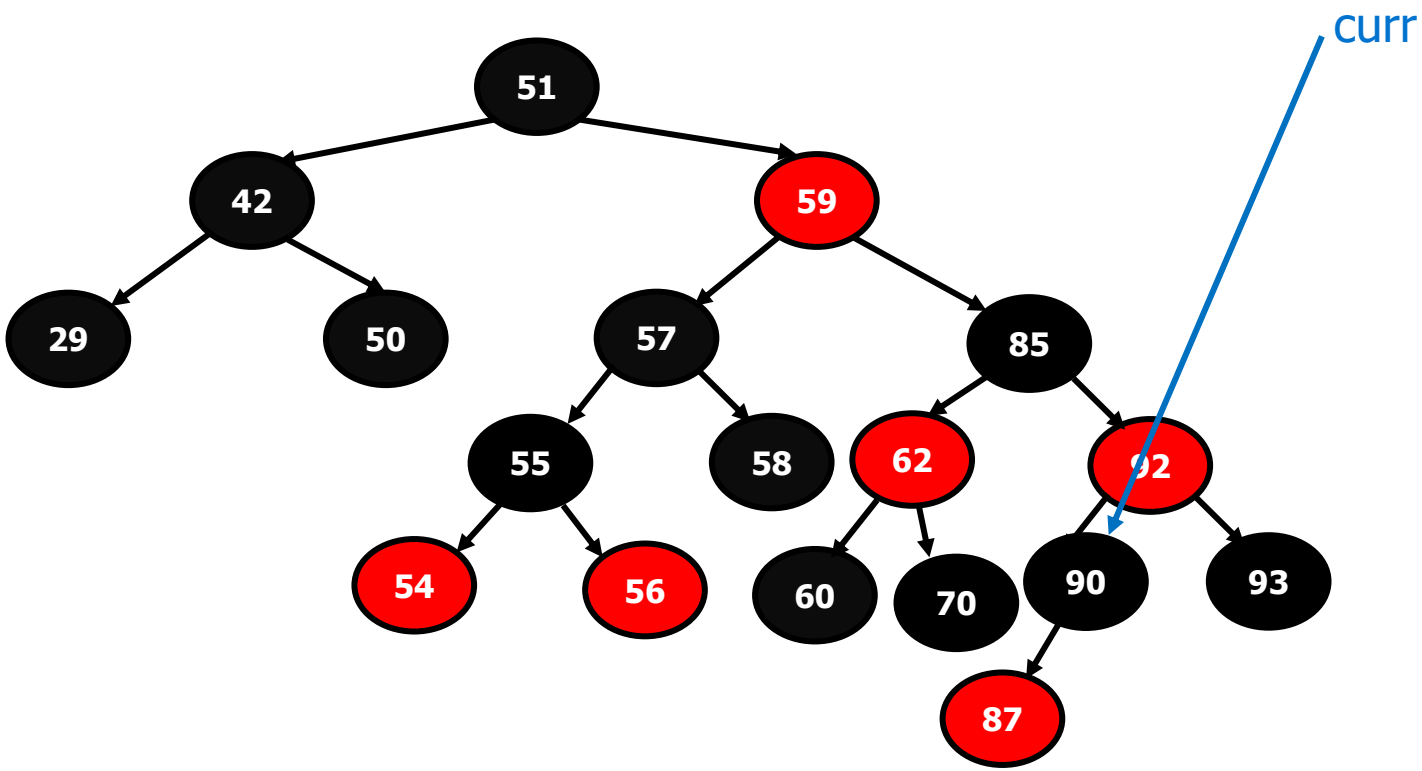
Add 87



Recolor 62 and 85

Putting It Together

Add 87

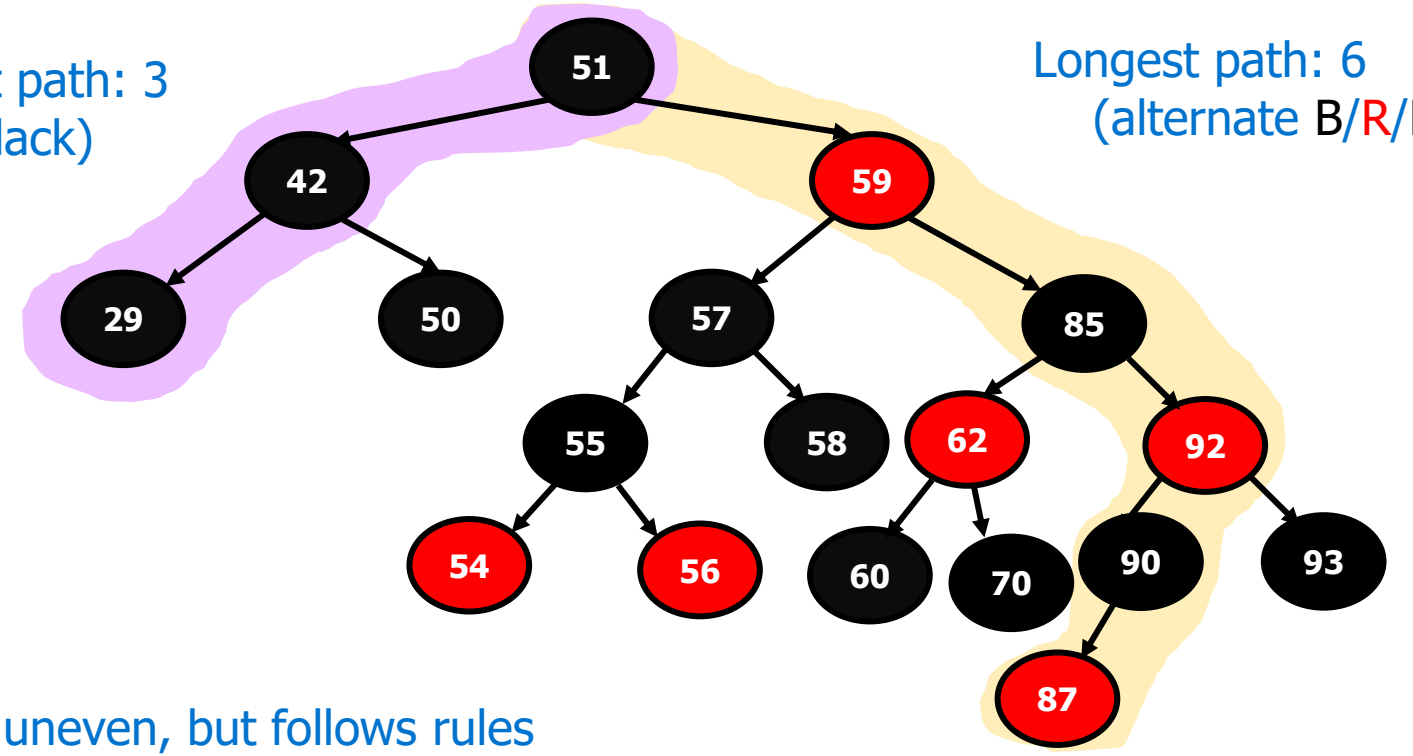


Found where it goes, add as red

Putting It Together

Shortest path: 3
(all black)

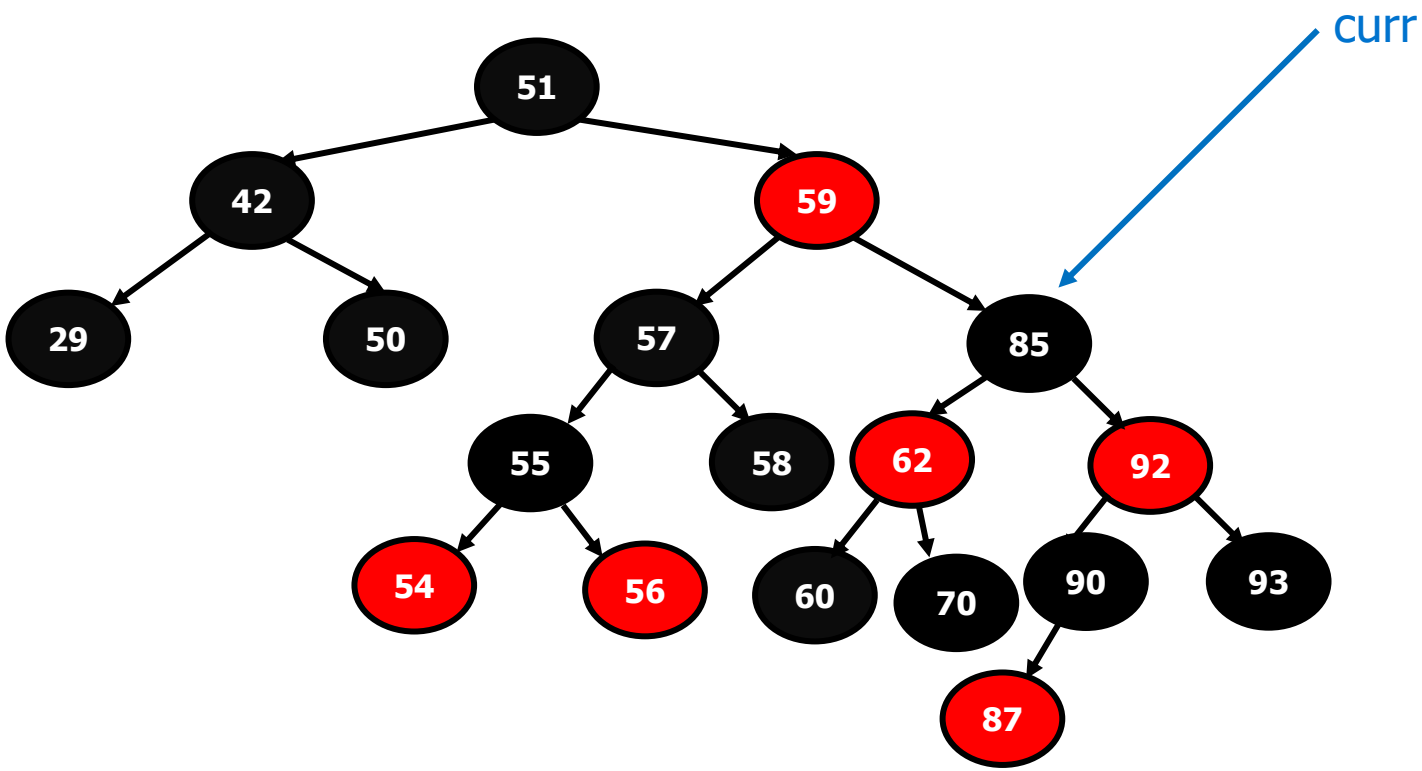
Longest path: 6
(alternate B/R/B/R/B/R)



Tree is a bit uneven, but follows rules

Putting It Together

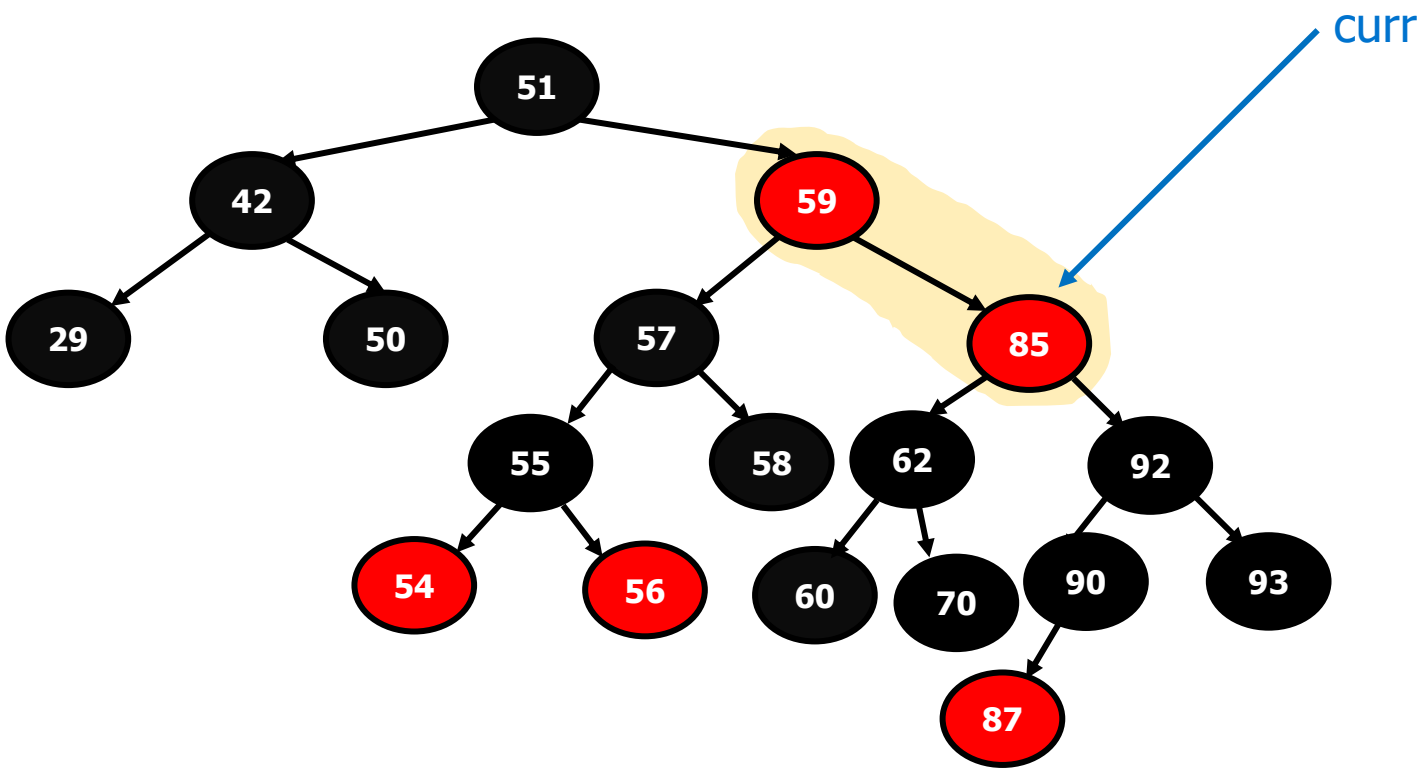
Now, add 86



Black node, two red children!

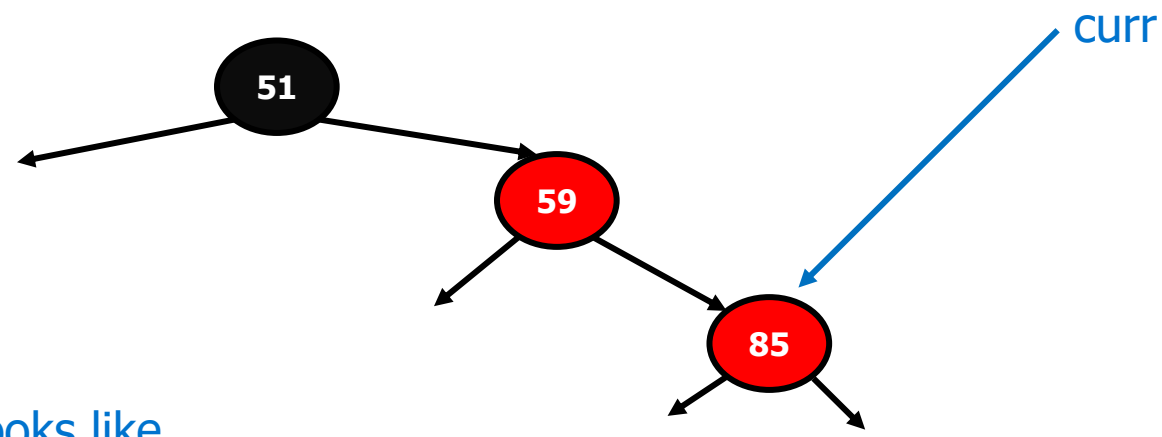
Putting It Together

Add 86



But now we have a rule 3 violation!
Need to rotate

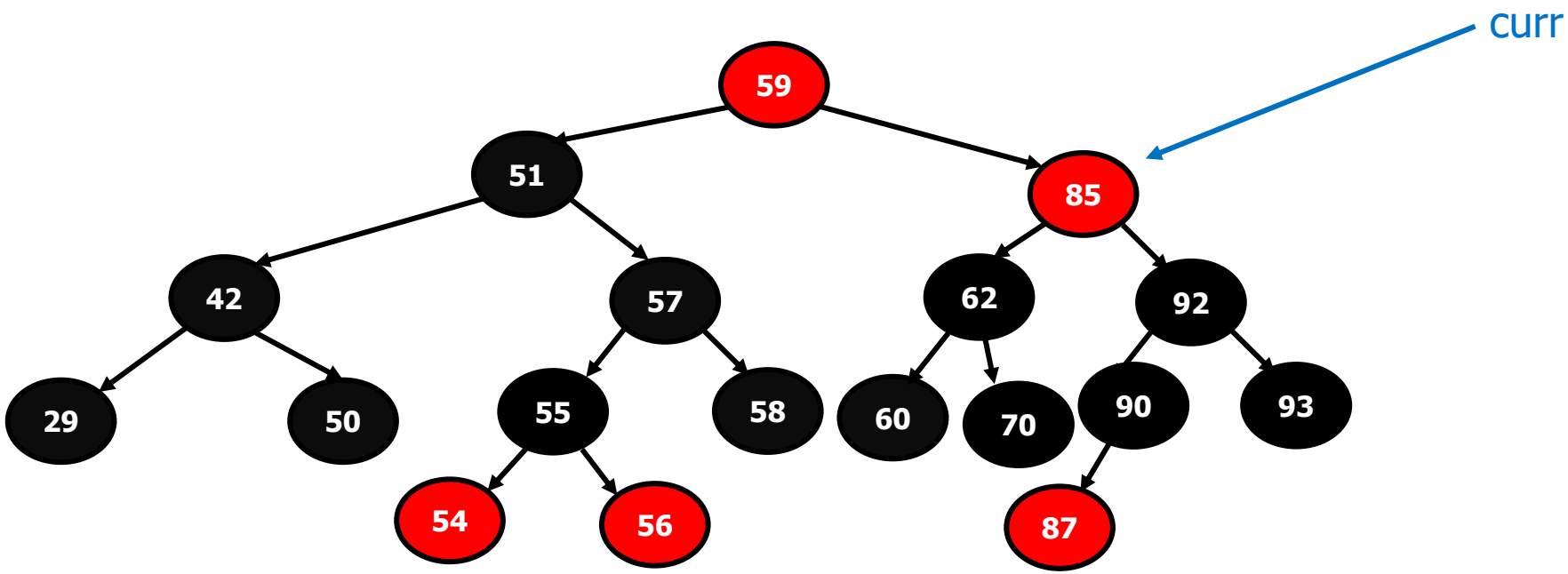
Putting It Together



This is what the rotation looks like
(as if we added 85)

Putting It Together

Add 86

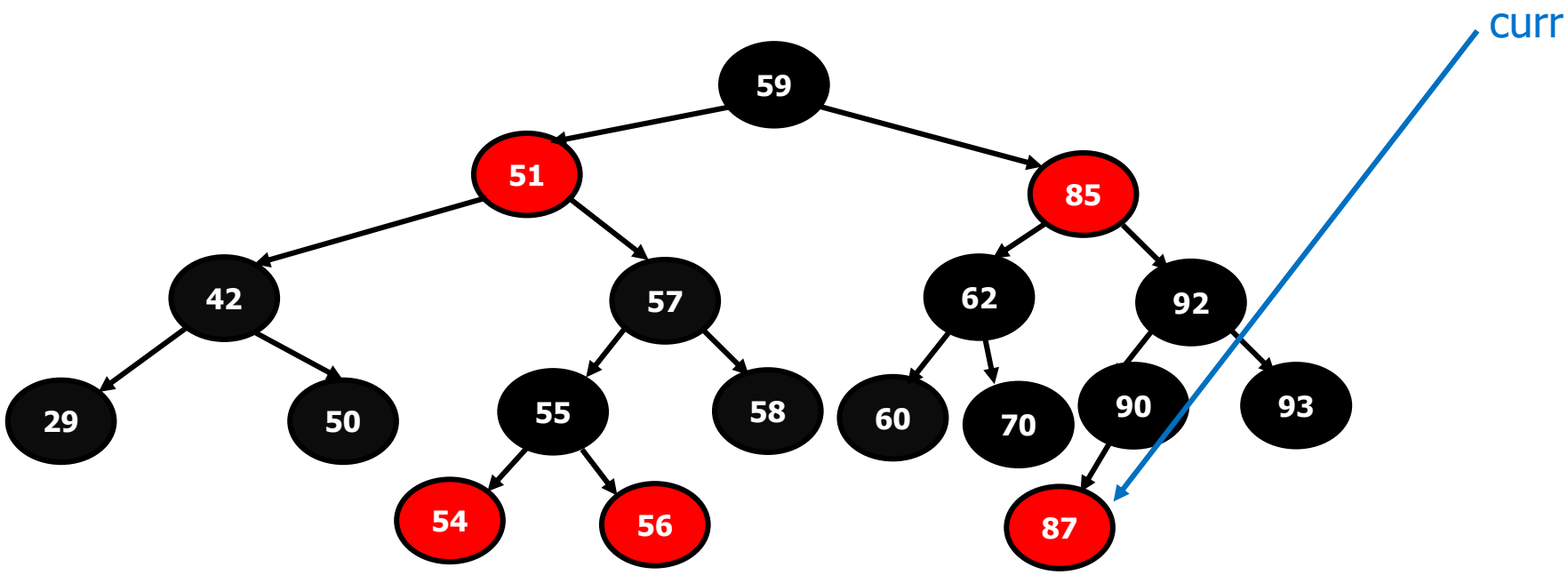


First we rotate …

Then we re-color (51 and 59)

Putting It Together

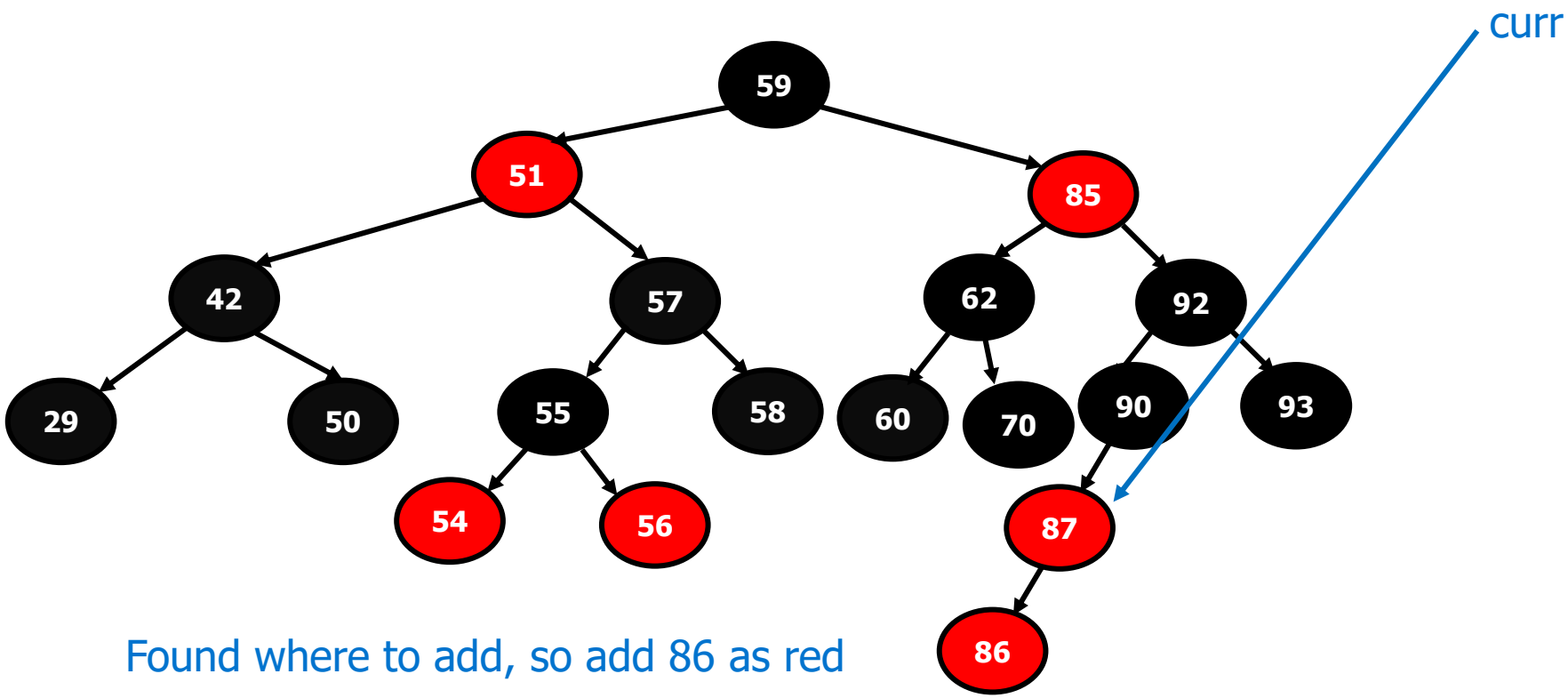
Add 86



And continue …

Putting It Together

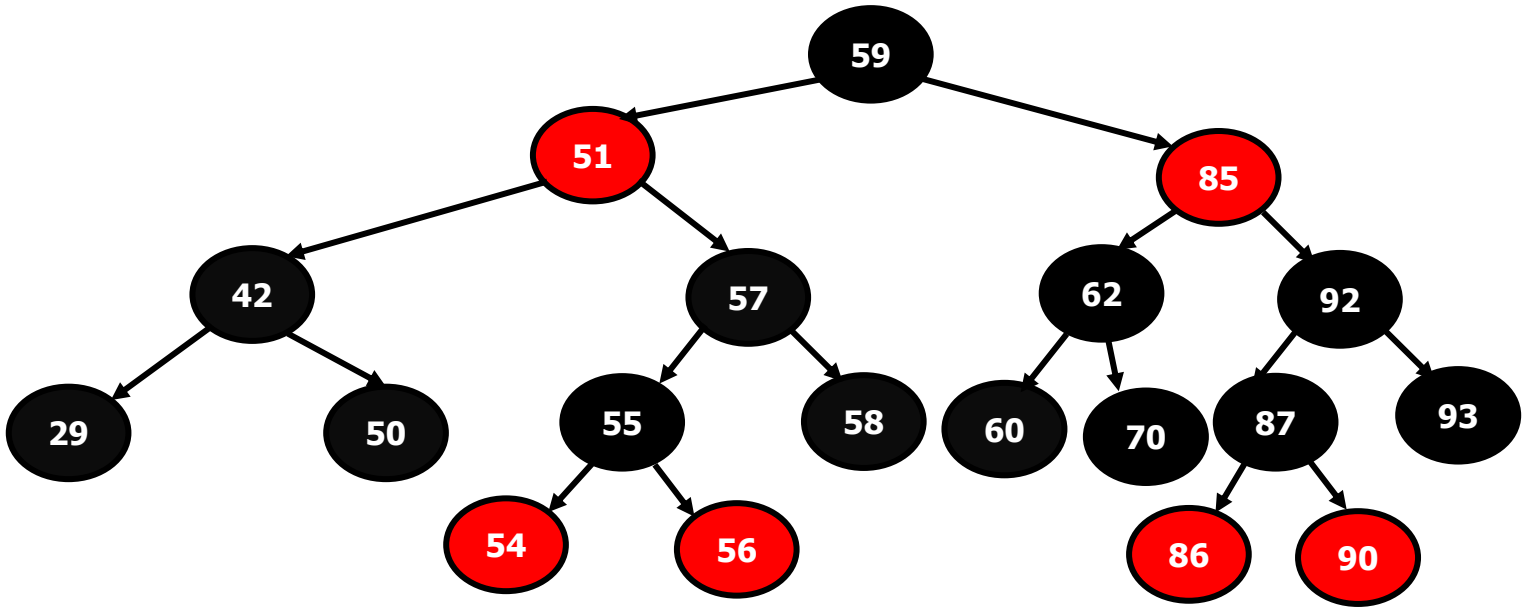
Add 86



Found where to add, so add 86 as red

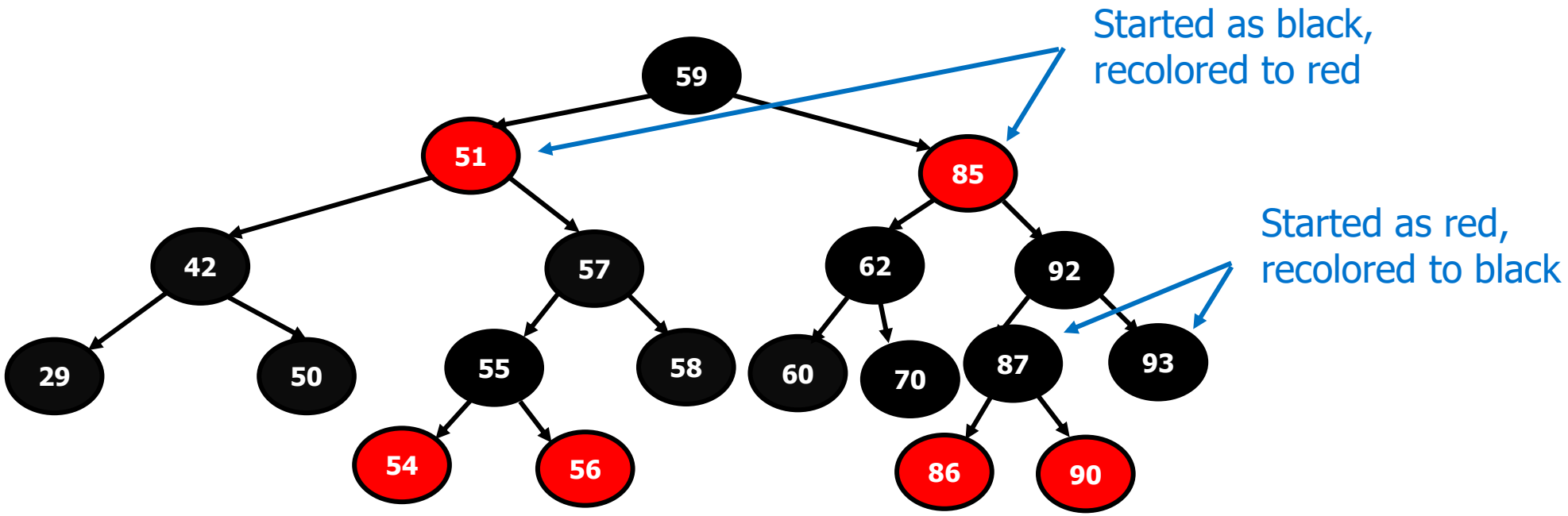
Violation: rotate, re-color

Putting It Together



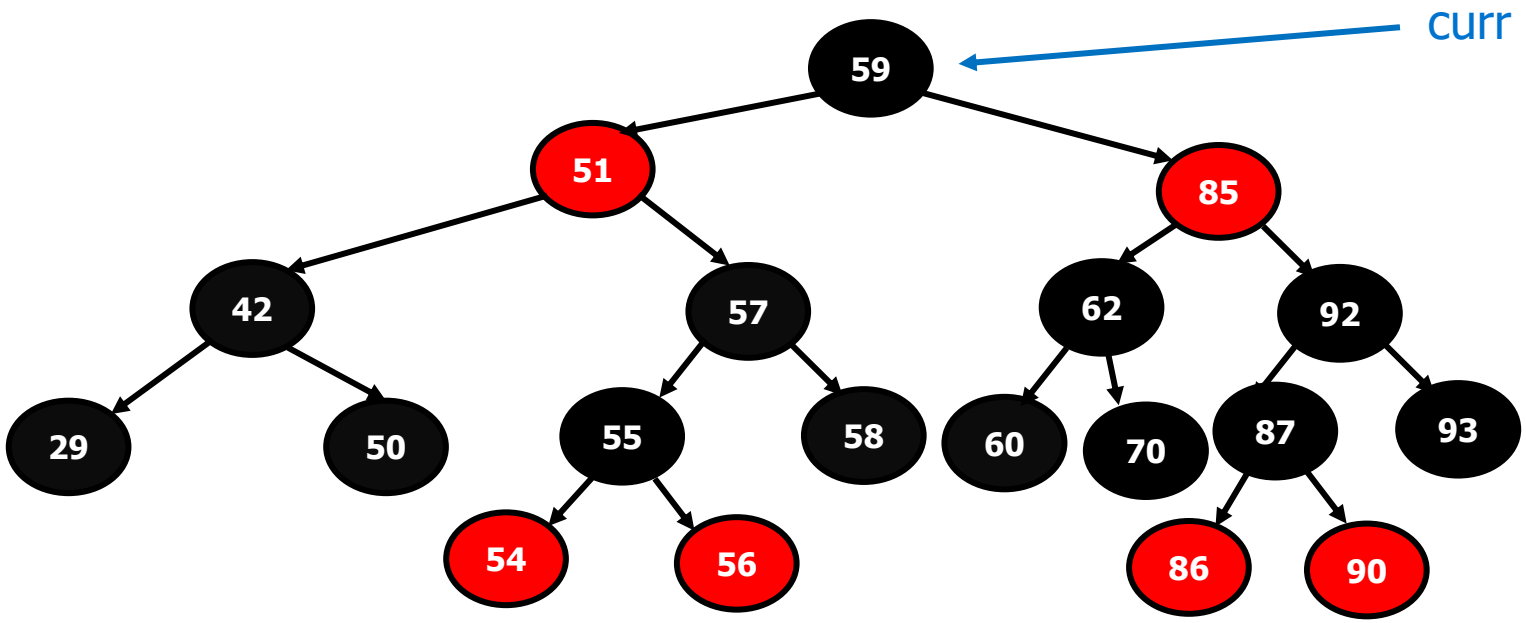
Note that we may do multiple rotates for one add
(unlike AVL)

Red Black Add



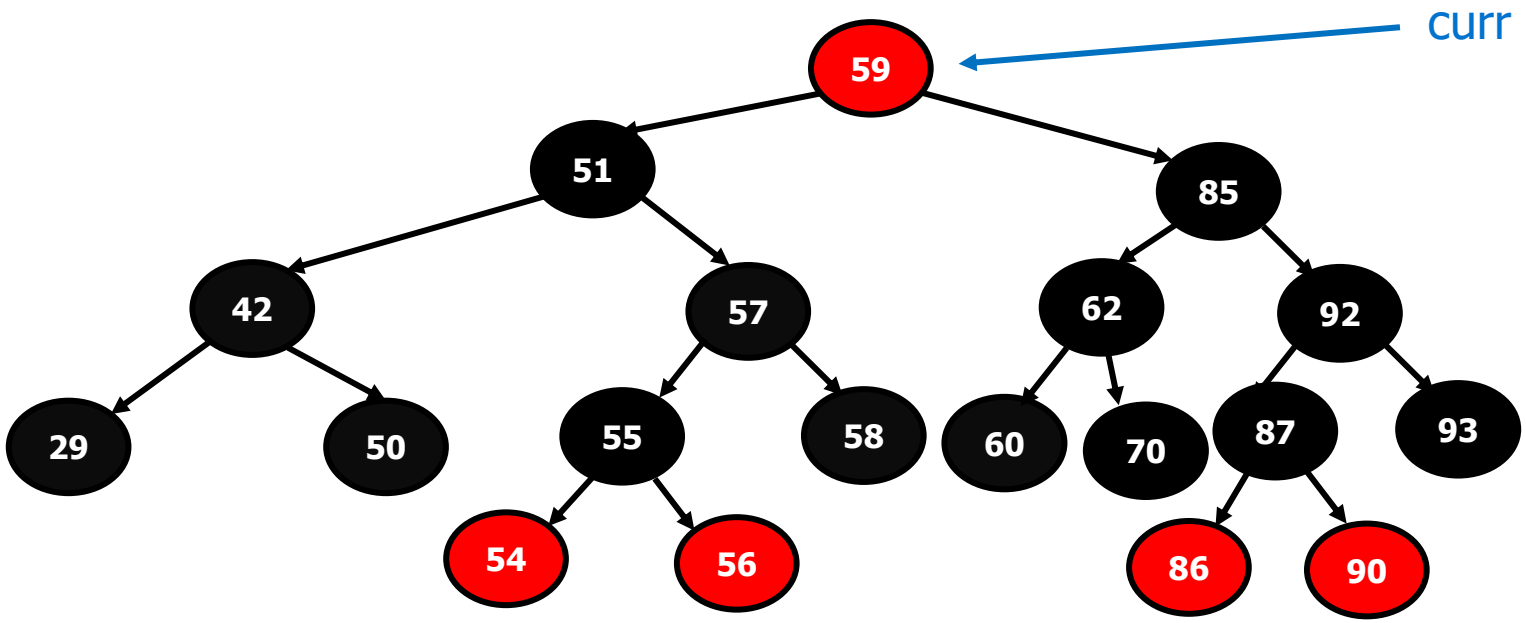
- So where do new black nodes come from ?
  - We have always added as red…
  - And the count of black nodes on a path has stayed 3…
  - … but that can't happen forever… right?

Red Black Add



- If we start to add again (doesn't matter what) …
  - The root is a black node with two red children!
  - Recolor…

Red Black Add

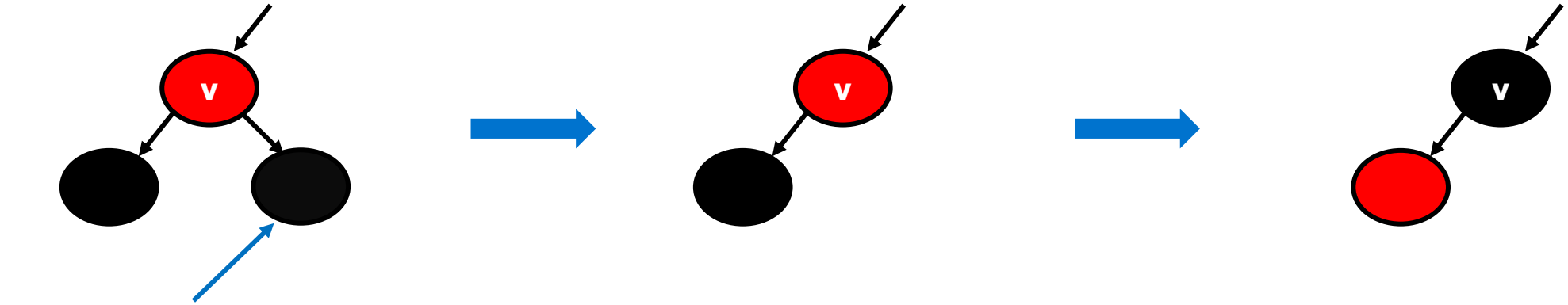


- If we start to add again (doesn't matter what) …
  - The root is a black node with two red children!
  - Recolor…
  - But rule 2 says the root is always black! Recolor root

Red-Black Delete

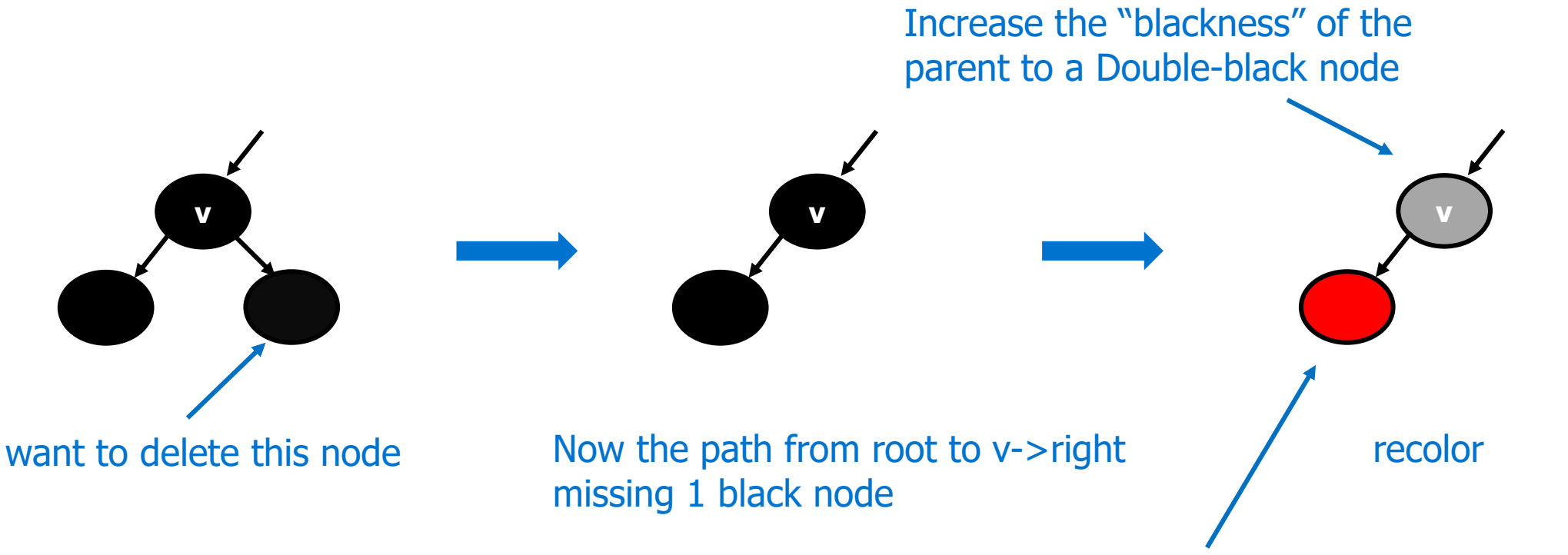
- Red-black deletion:
  - Remove red node? Easy
  - Remove black node? Now 1 short…
- Nice insight by Dr. Matt Might (Prof at U. of Utah)
  - Add temporary colors: "double black" & "negative black"
  - Allows you to keep invariants the whole time
    - Just have to "fix" colors you aren't allowed to have
  - https://matt.might.net/articles/red-black-delete/
- Side note: Matt's blog is highly recommended reading
  - Programming, grad school, HOWTOs, Productivity…

Red-Black Delete Example



- Start with an easy case
  - Delete a black leaf with a red parent

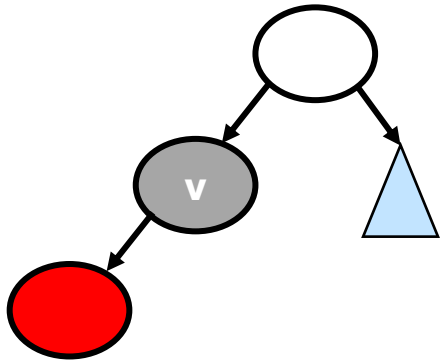
Red-Black Delete Example



- Now a slightly complicated case
  - Delete a black leaf with a black parent
  - Introduce "double-black" node to keep invariants (rule 4)

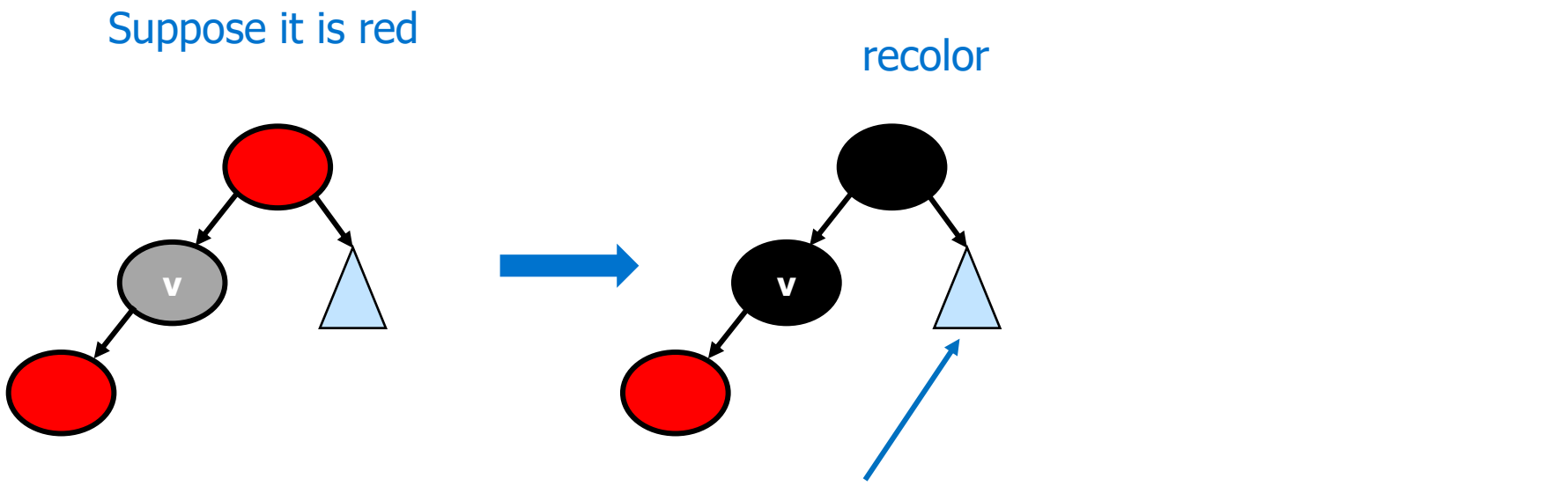
Red-Black Delete Example

Need to know the color of v's parent



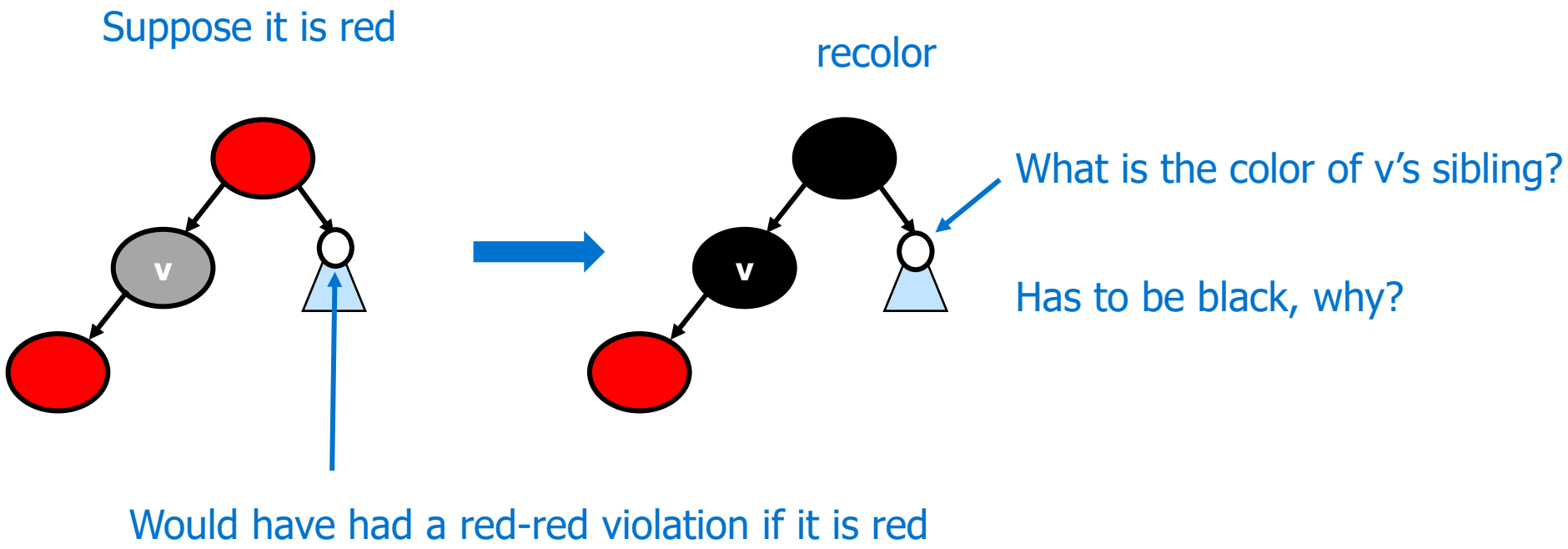
- Now just need to fix the double-black node

Red-Black Delete Example



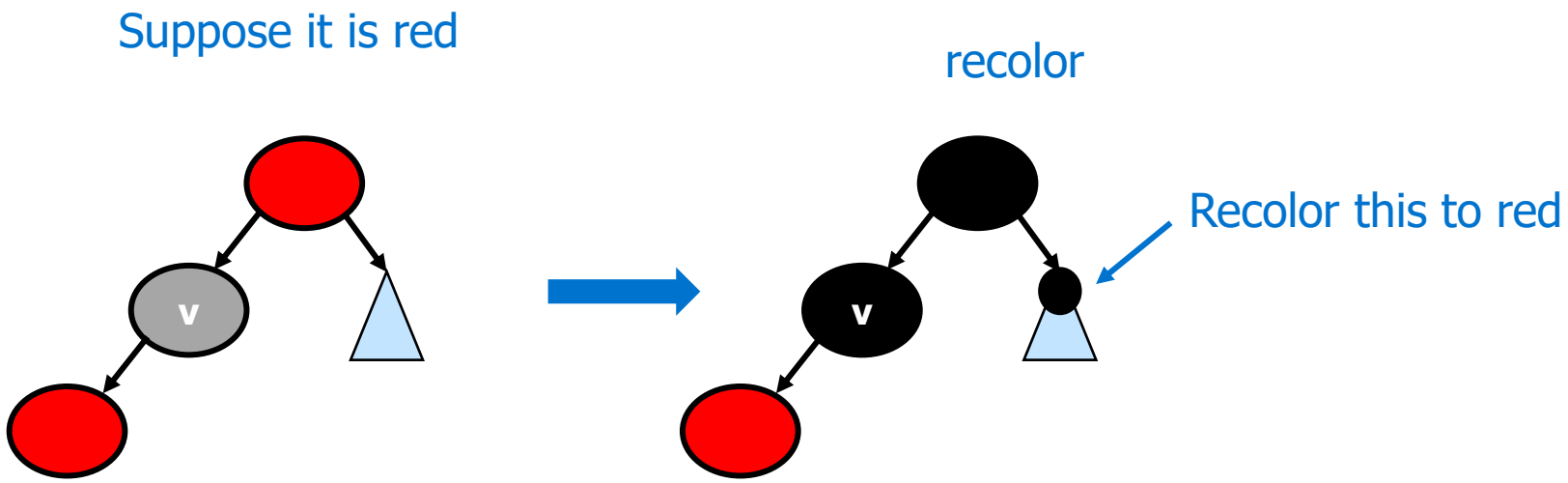
- Now just need to fix the double-black node
  - Recolor: decrease the blackness of v, increase the blackness of v's parent

Red-Black Delete Example



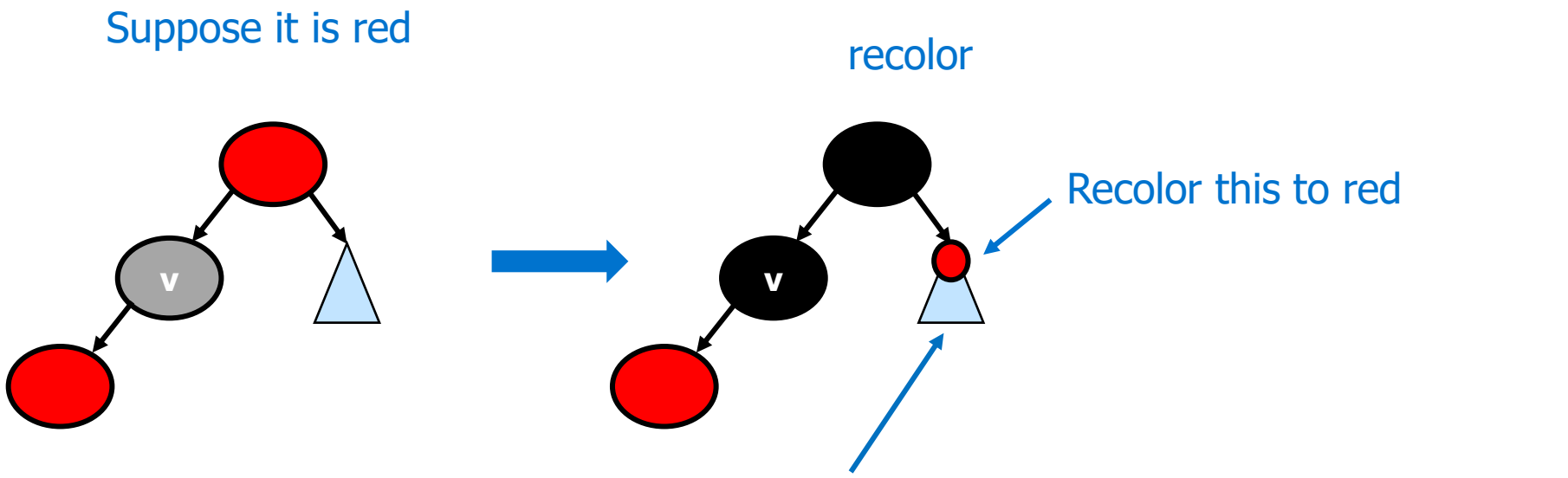
- Now just need to fix the blackness increase in v's sibling

Red-Black Delete Example



- Now just need to fix the blackness increase in v's sibling

Red-Black Delete Example

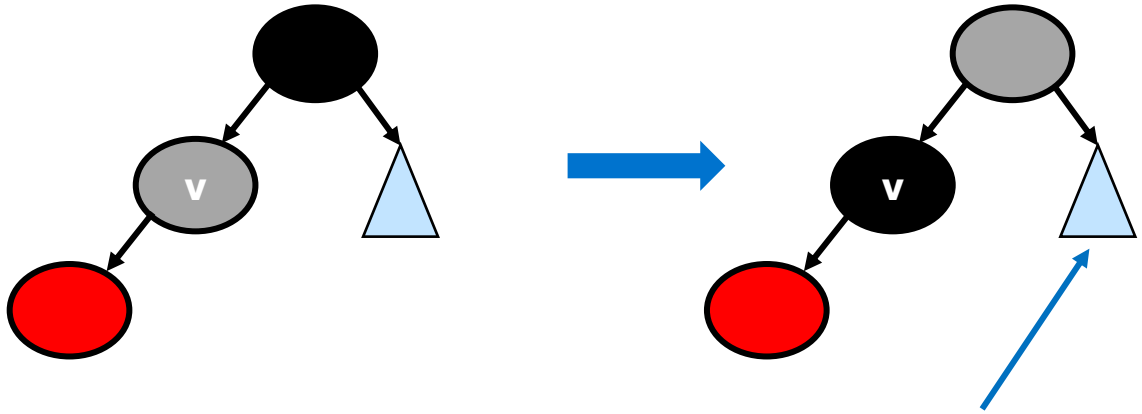


- Now just need to fix the blackness increase in v's sibling

Red-Black Delete Example

What if this is black?

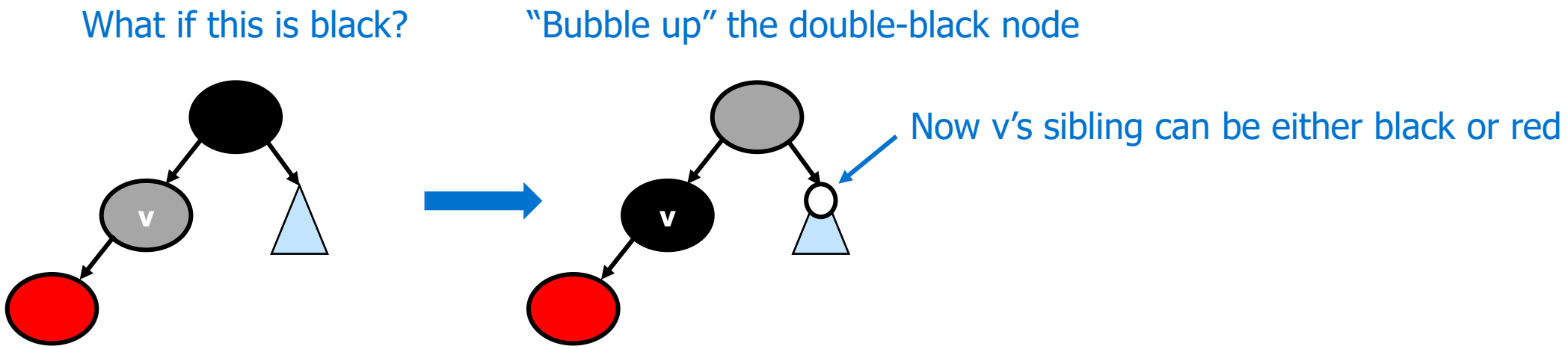
"Bubble up" the double-black node



Same problem: increased v's sibling's blackness by 1, need to fix it

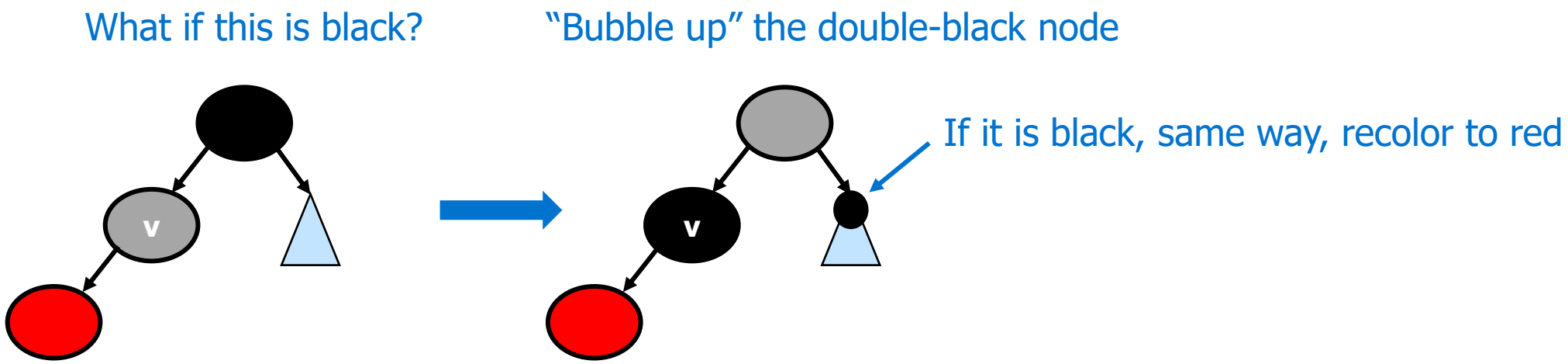
- Now just need to fix the double-black node
  - Bubble up: move double-black up to the parent node

Red-Black Delete Example



- Now just need to fix the blackness increase in v's sibling

Red-Black Delete Example

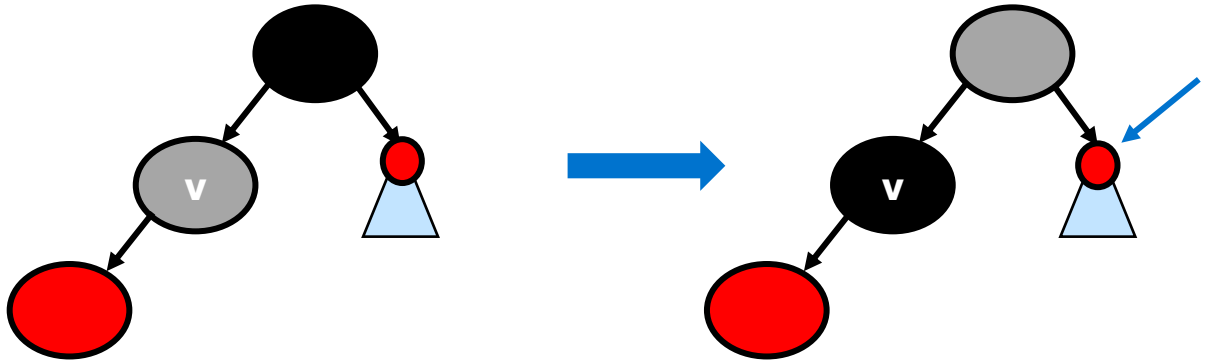


- Now just need to fix the blackness increase in v's sibling

Red-Black Delete Example

What if this is black?

"Bubble up" the double-black node

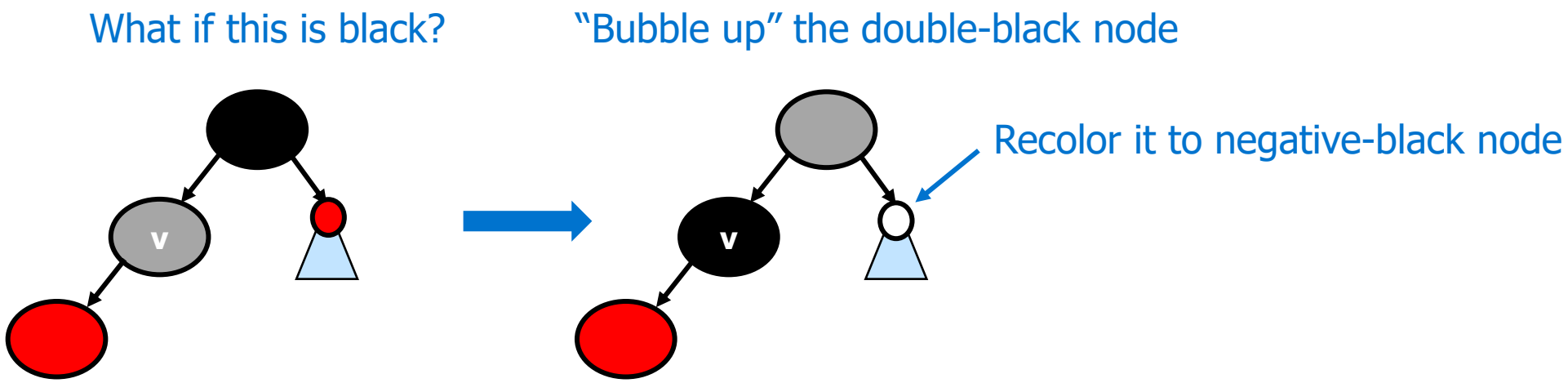


But what if it is red …

That's where the negative-black node comes in …

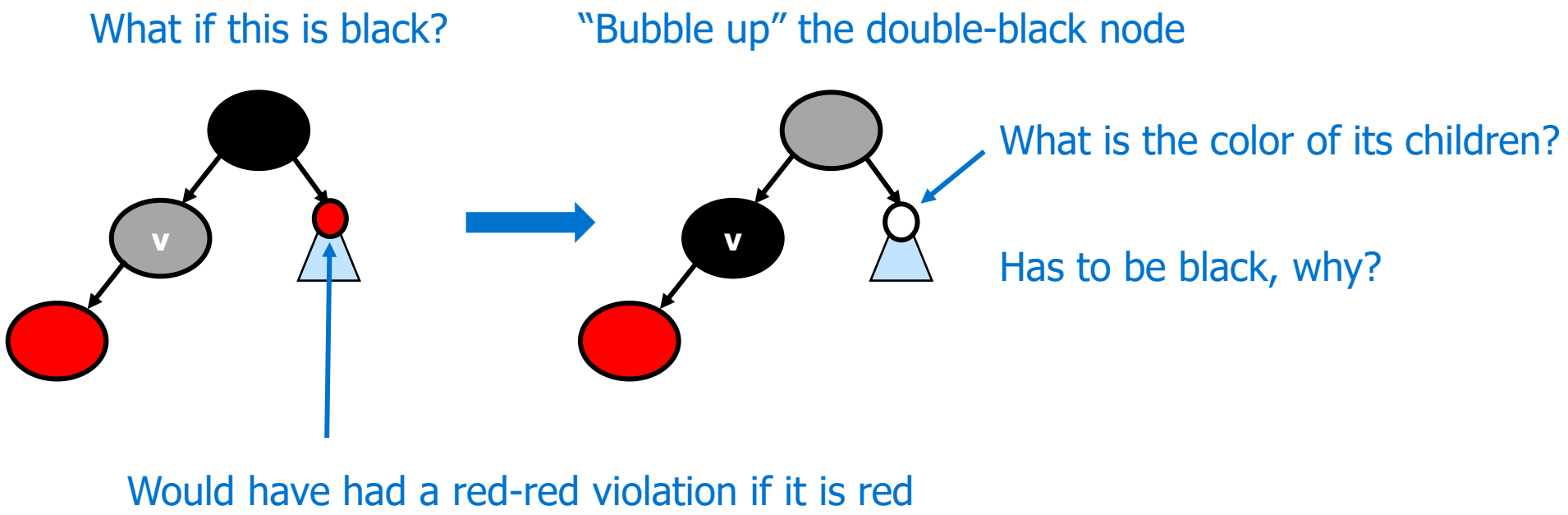
- Now just need to fix the blackness increase in v's sibling

Red-Black Delete Example



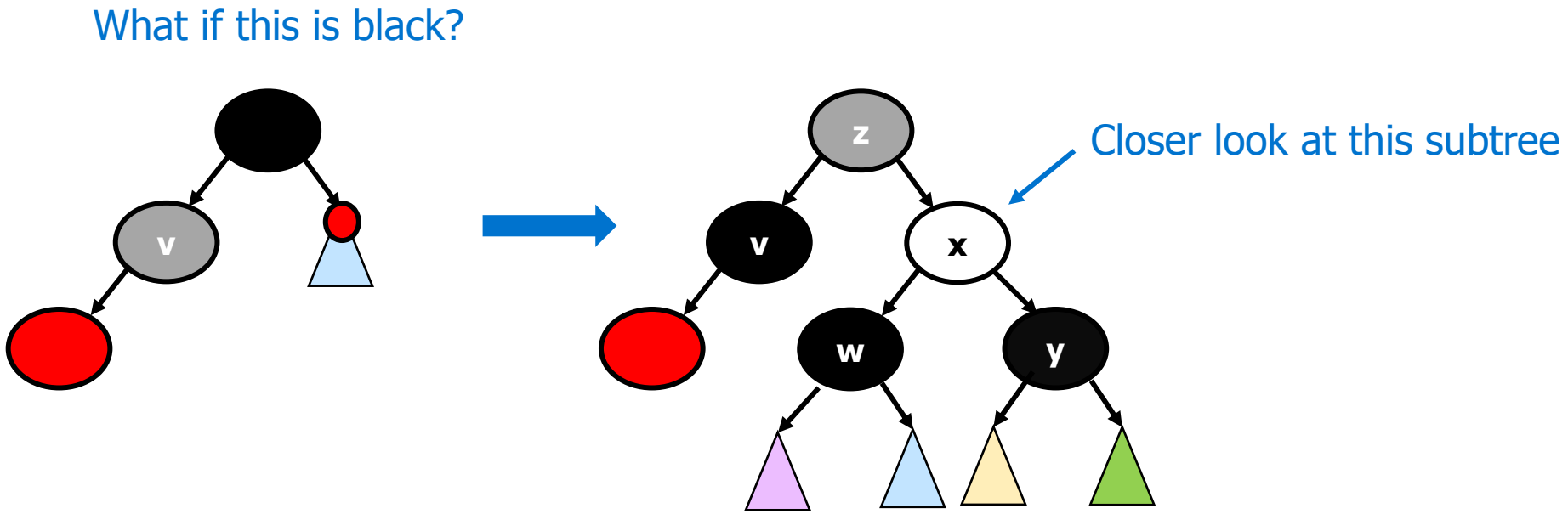
- Now just need to fix the negative-black node

Red-Black Delete Example



- Now just need to fix the negative-black node

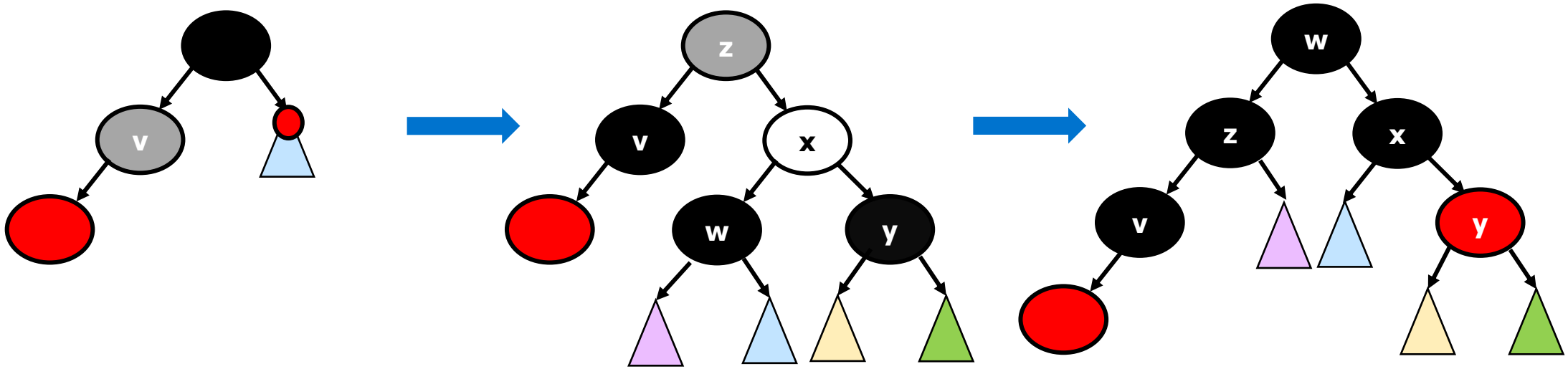
Red-Black Delete Example



- Now just need to fix the negative-black node

Red-Black Delete Example

What if this is black?



How do we get here?
Challenge for you …

- Now just to fix the negative-black node
  - Rotate and recolor

Why Red-Black?

- Advantage of red-black (vs AVL)
  - Can do all balancing on the way down for insert
  - Makes iterative algorithm easier (for people scared of recursion)
  - Faster insertion and removal
  - Less storage overhead (1-bit)

- Disadvantage of red-black (vs AVL)
  - Slightly worse guarantee on height
  - Slower lookup

Wrap Up

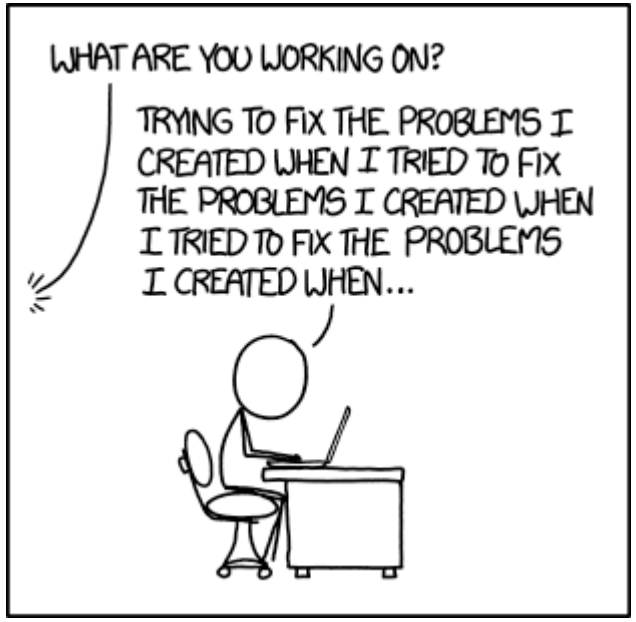
- In this lecture we talked about
  - Red-black trees
    - Another form of balanced tree
    - Weaker guarantees than AVL (pros & cons)

- Important higher-level take-away points
  - Data structure with interesting invariants
    - Key benefits guaranteed because of them
    - Operations rely on them being true at the start
  - Picture drawing: crucial
    - Would be ridiculously difficult to do without them!

- Next up
  - Heaps & Priority Queues

Suggested Complimentary Readings

- Data Structure and Algorithms in C++: Chapter 12.2
- Introduction to Algorithms: Chapter 13



Acknowledgement

- This slide builds on the hard work of the following amazing instructors:
  - Andrew Hilton (Duke)