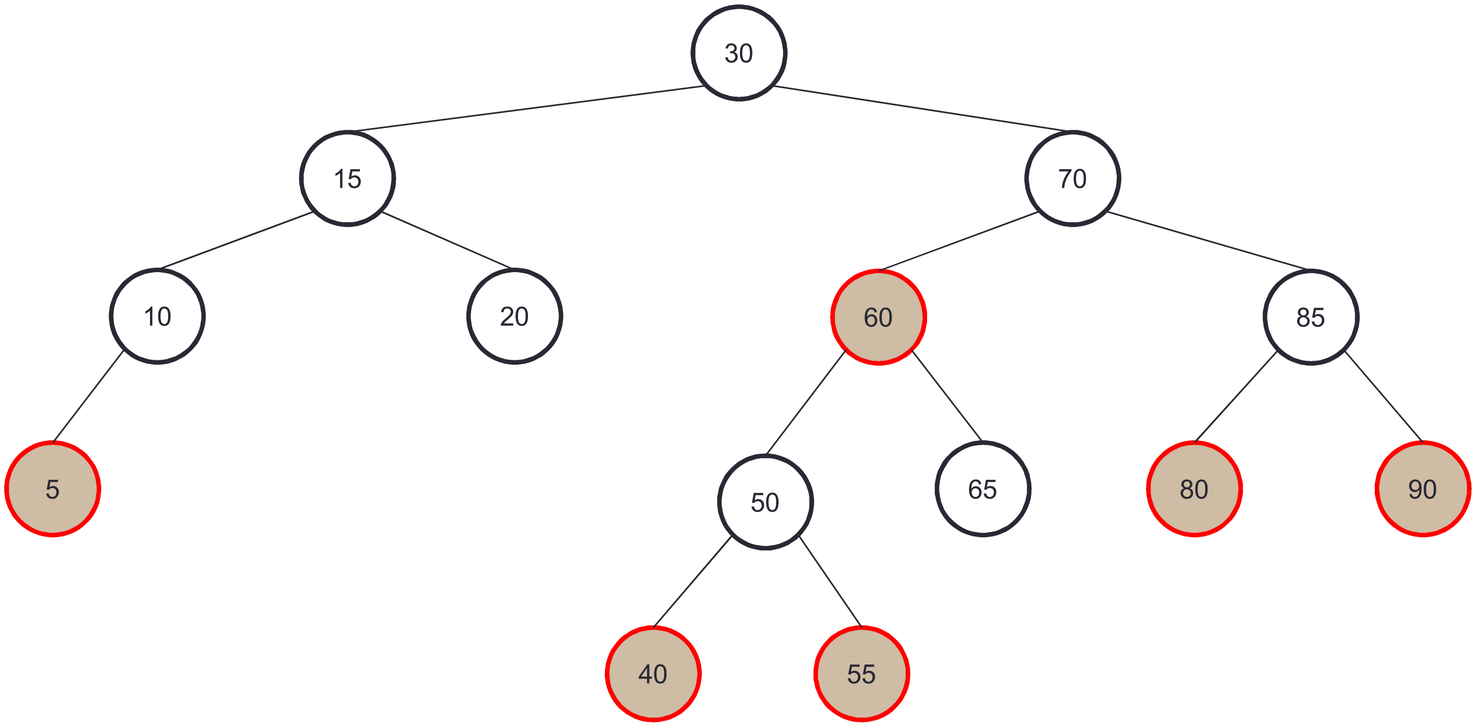
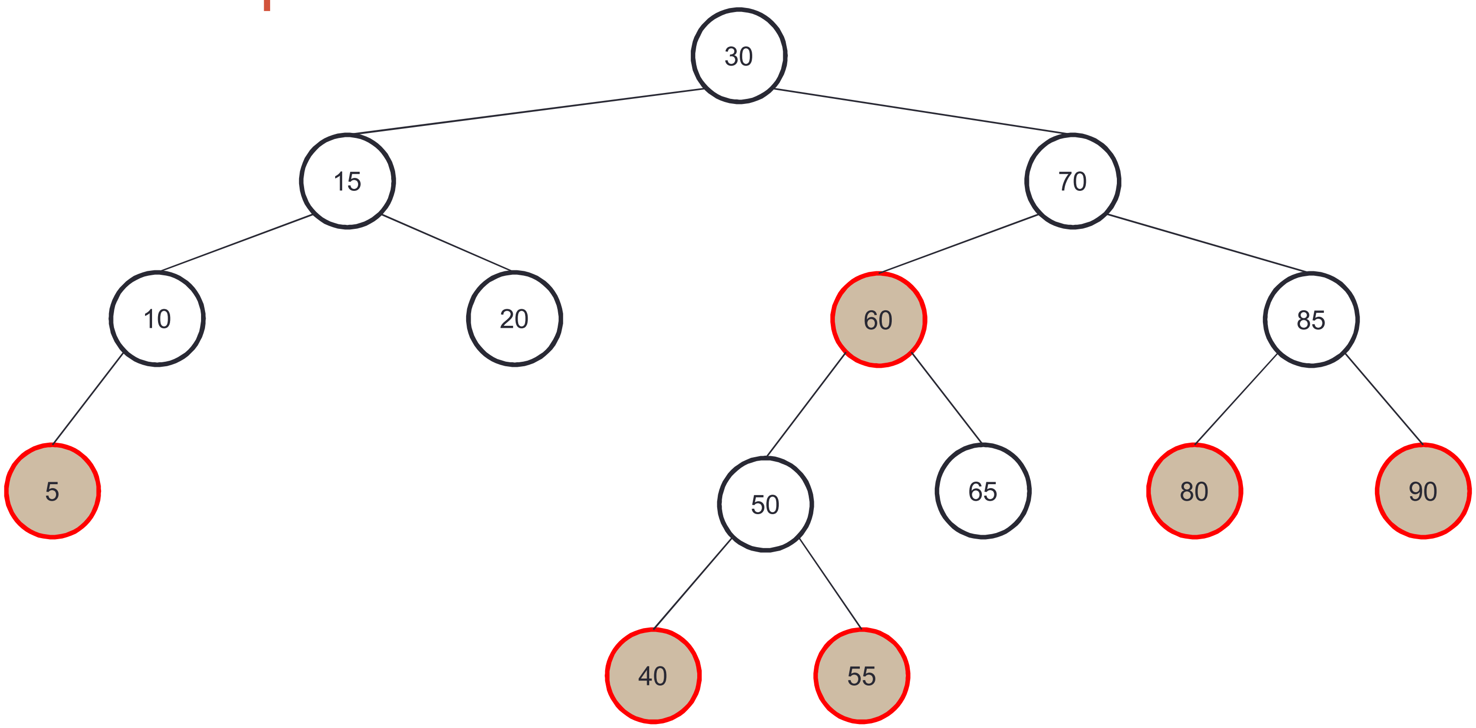
CSE 100: RBT

Red-Black trees have height O(logN)



Intuition for proof – remove all red nodes all black nodes are at the same height. Can then reason about relationship between n and the height. Height of just black is will be O(log n). Add back red and you get just 2*height black.

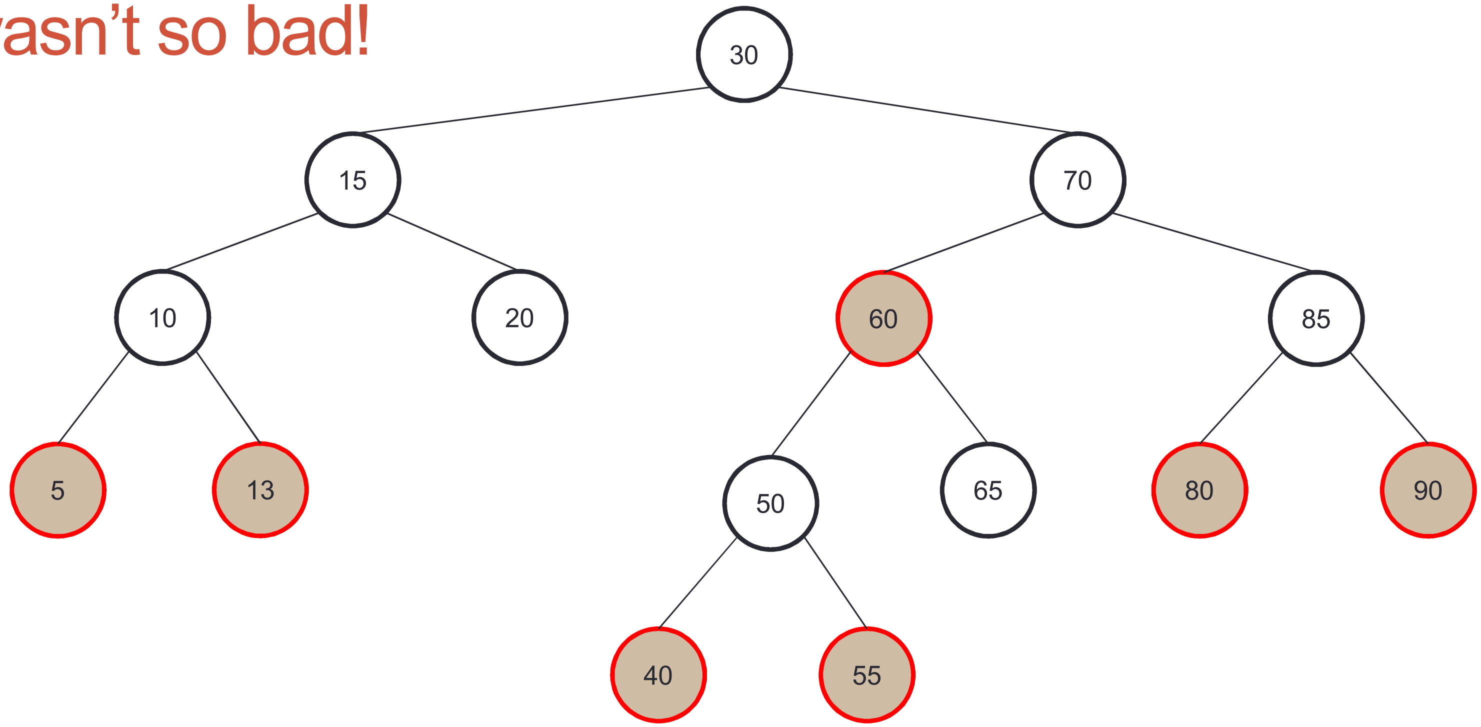
Now for the fun part… insertions



Non-root insertions will always be red

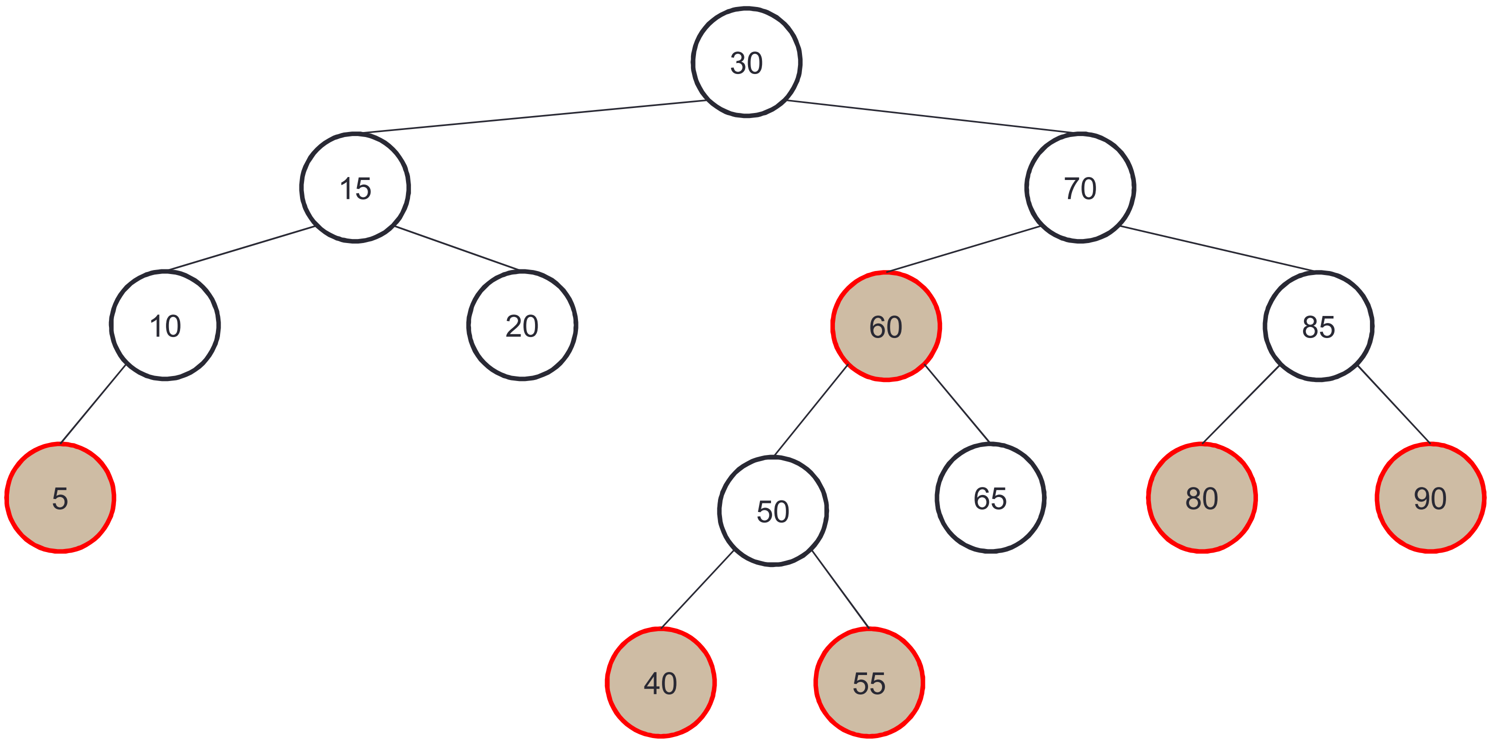
Try inserting 13

That wasn't so bad!



Case 0: Parent was black. Insert new leaf node (red) and you're done.

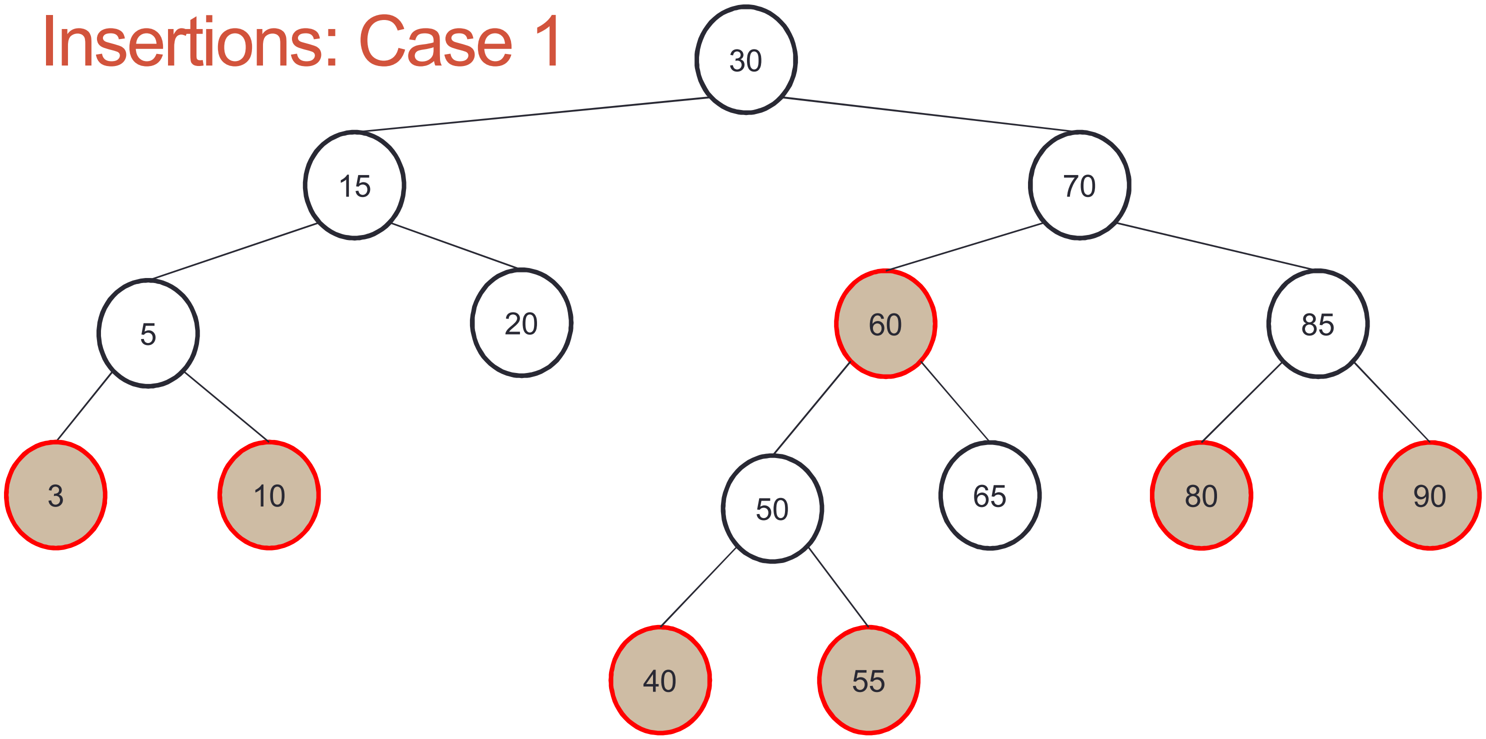
Insertions: More complicated case



Try inserting 3

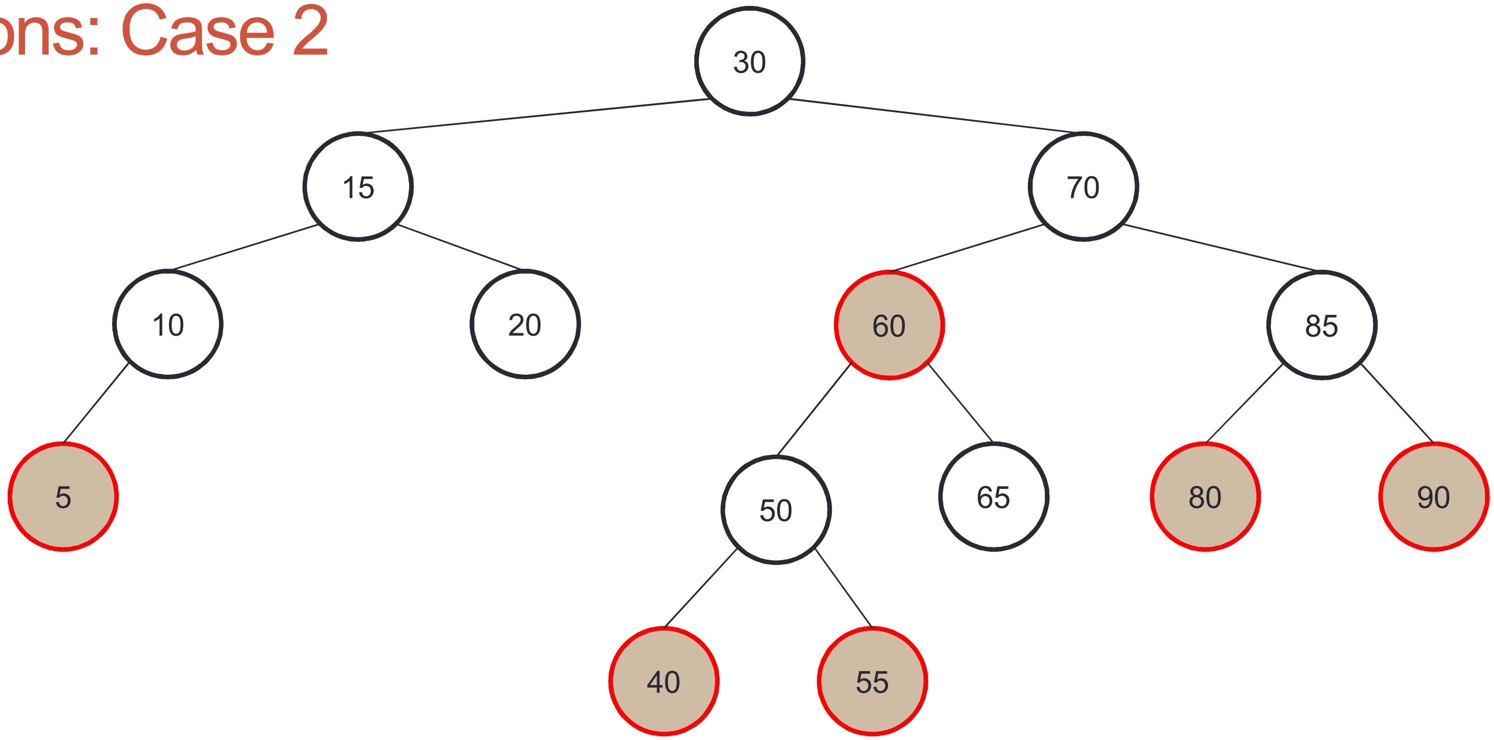
Case 1: Parent of leaf is red, parent is left child of grandparent,
leaf is left child of parent, (& sibling of parent is black)

Insertions: Case 1



Right AVL rotation, and recolor

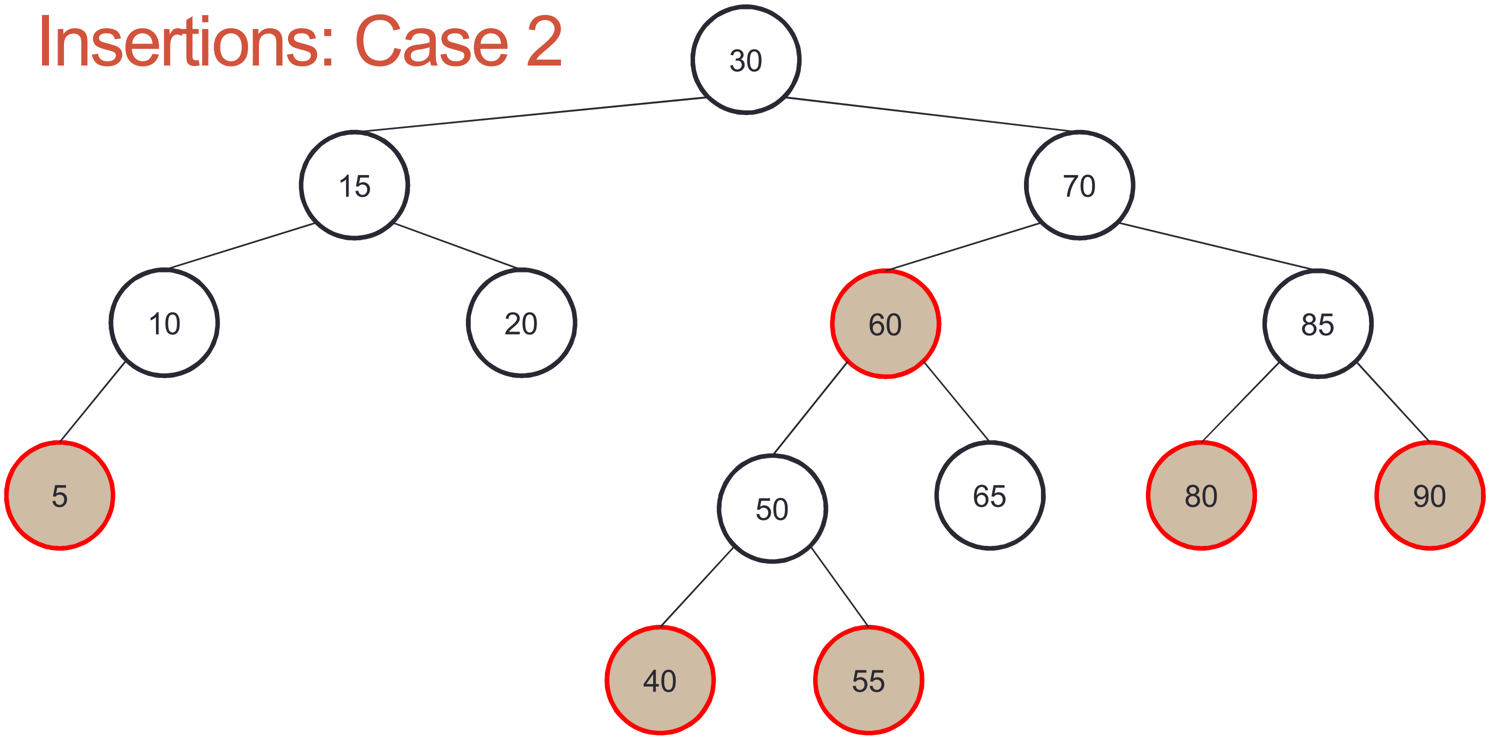
Insertions: Case 2



Which insertion can we not handle with the cases we've seen so far?

A. 1 B. 7 C. 12 D. 25

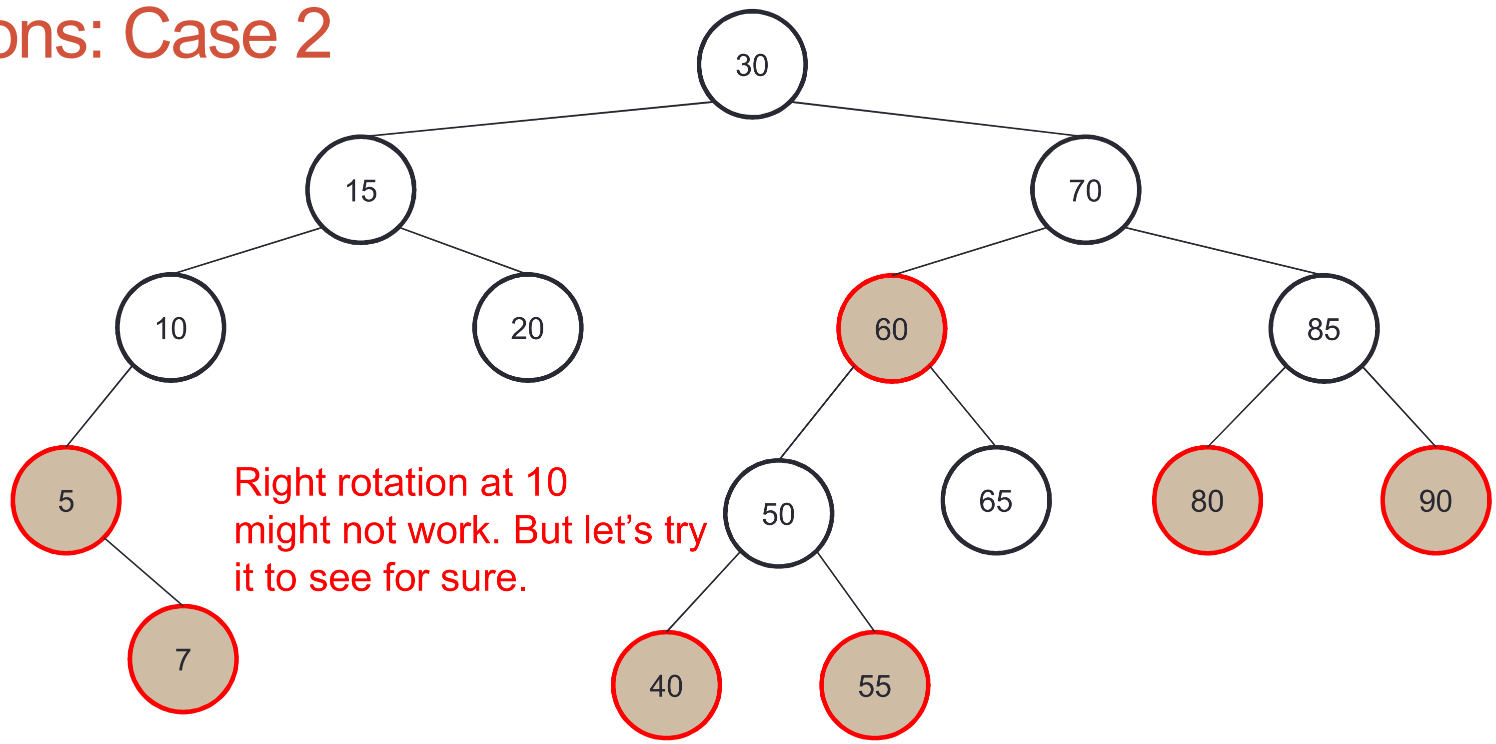
Insertions: Case 2



Insert 7

Case 2: Parent of leaf is red, parent is a left child of grandparent,
leaf is right child of parent, (& sibling of parent is black)

Insertions: Case 2

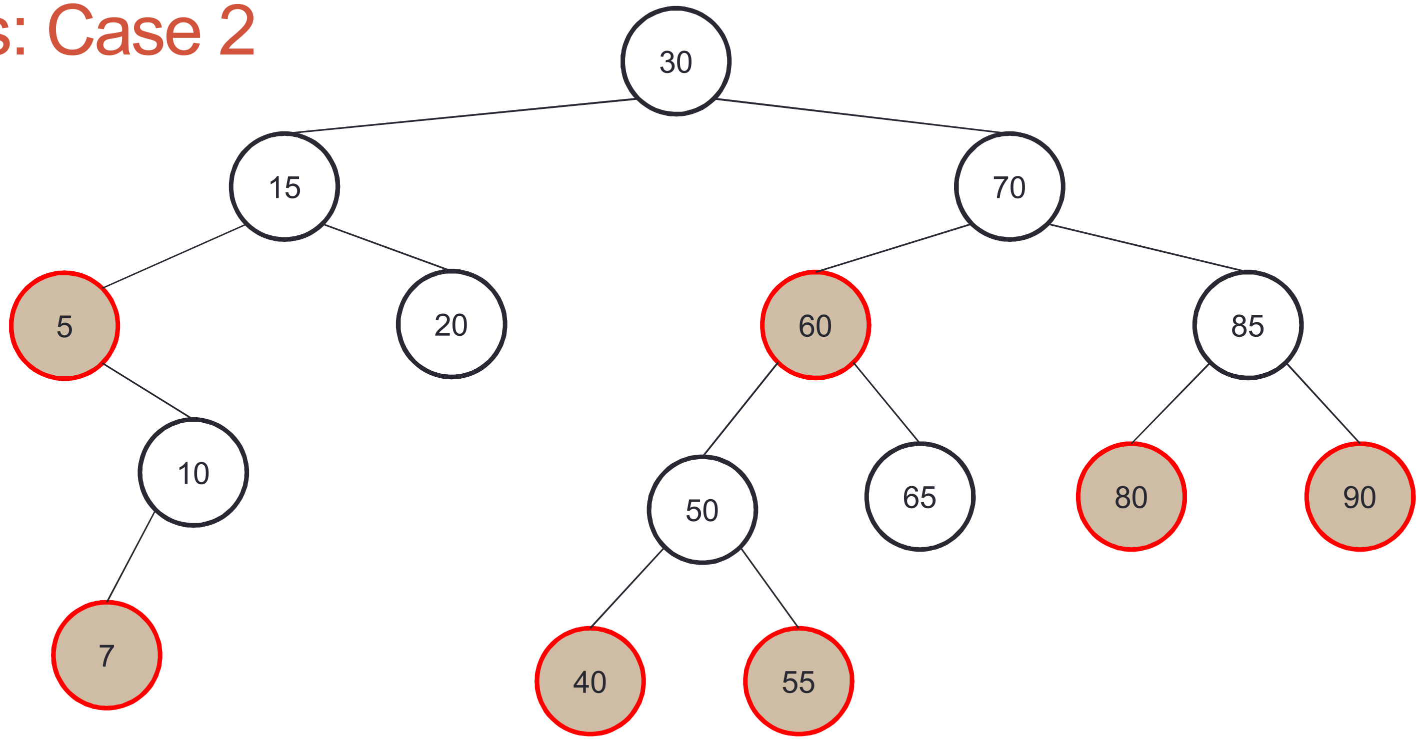


Right rotation at 10 might not work. But let's try it to see for sure.

Insert 7

Case 2: Parent of leaf is red, parent is a left child of grandparent, leaf is right child of parent, (& sibling of parent is black)

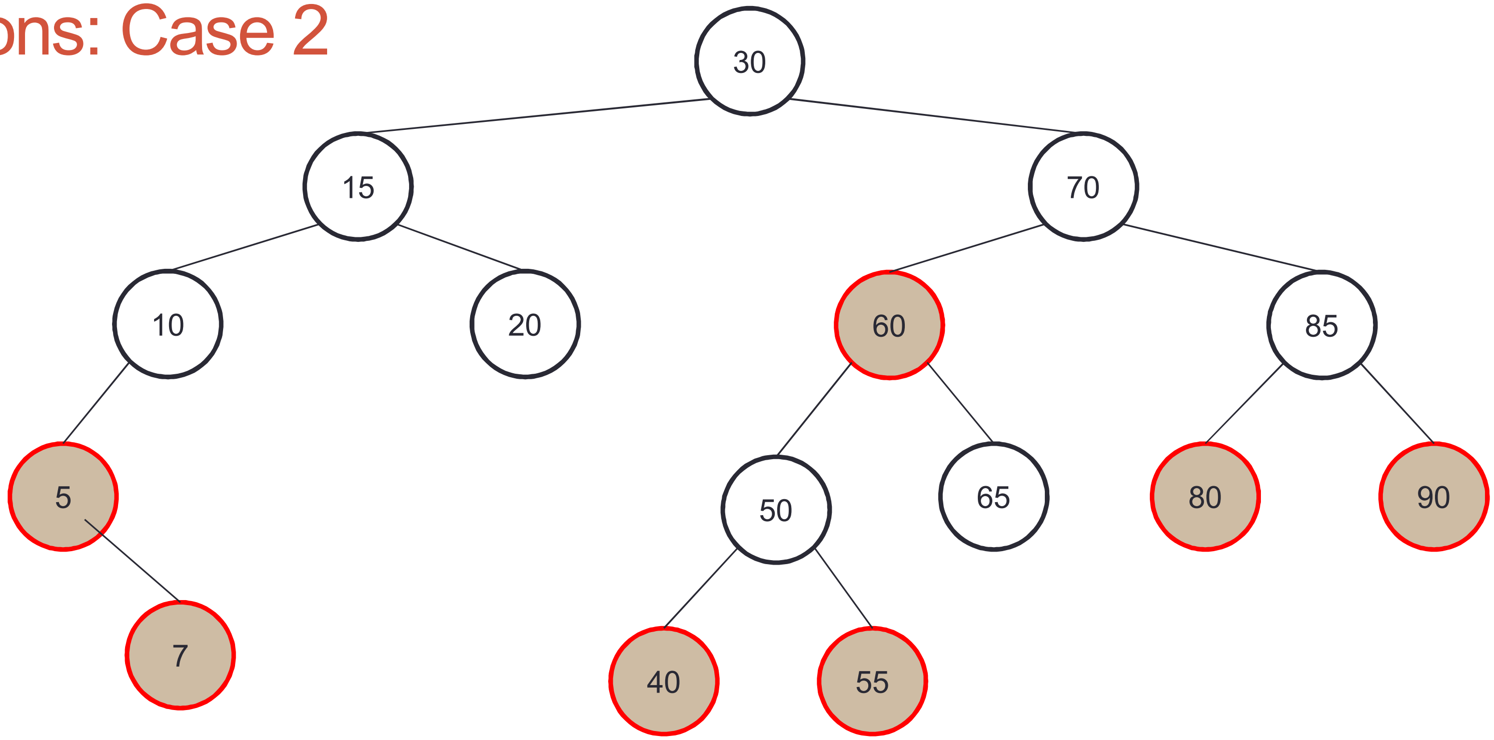
Insertions: Case 2



Why didn't this work?

A. It did! We're done!
B. The property about red nodes having only black children is violated.
C. The property about having the same number of black nodes on any path from the root through a null reference is violated.

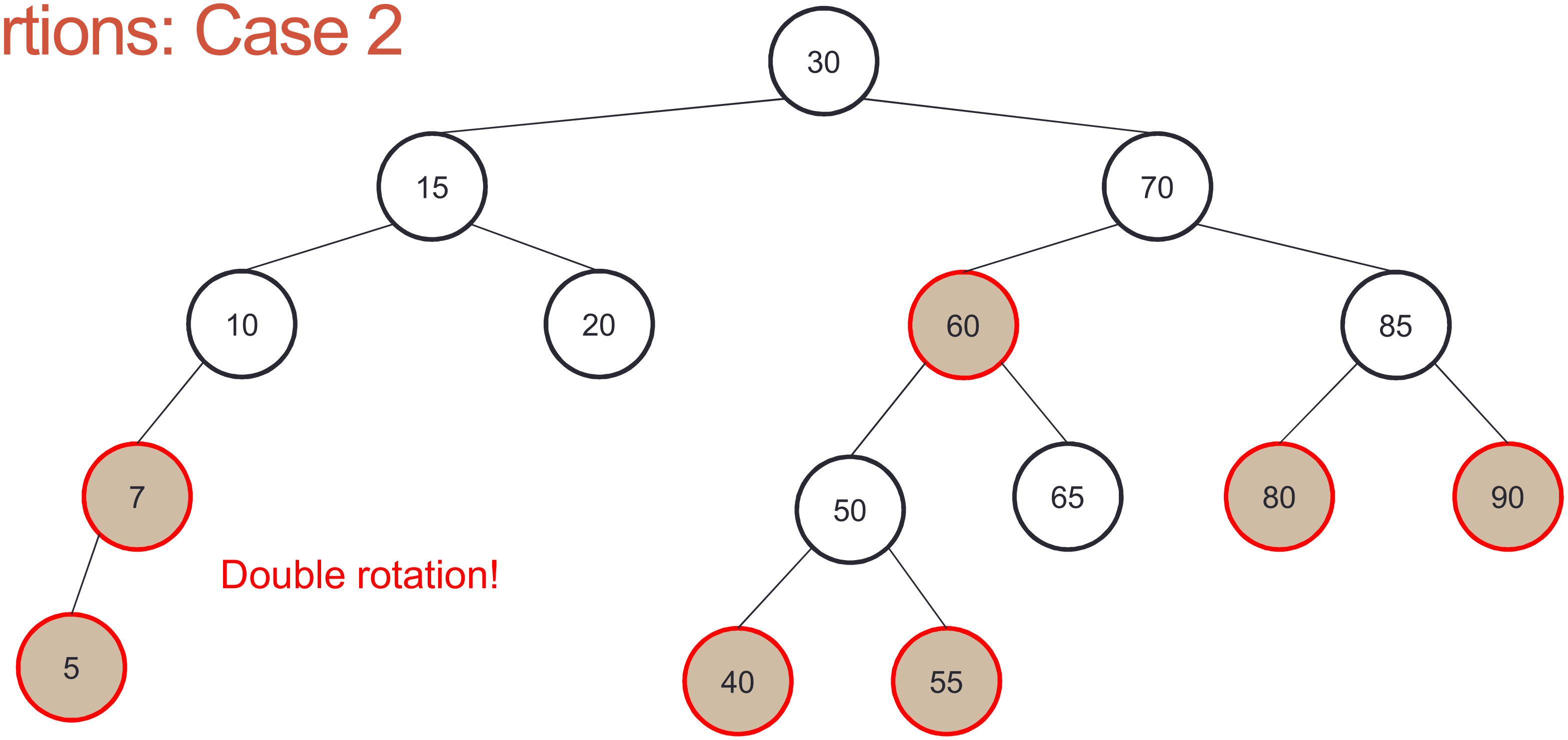
Insertions: Case 2



Insert 7

Case 2: Parent of leaf is red, parent is a left child of grandparent, leaf is right child of parent, (& sibling of parent is black)

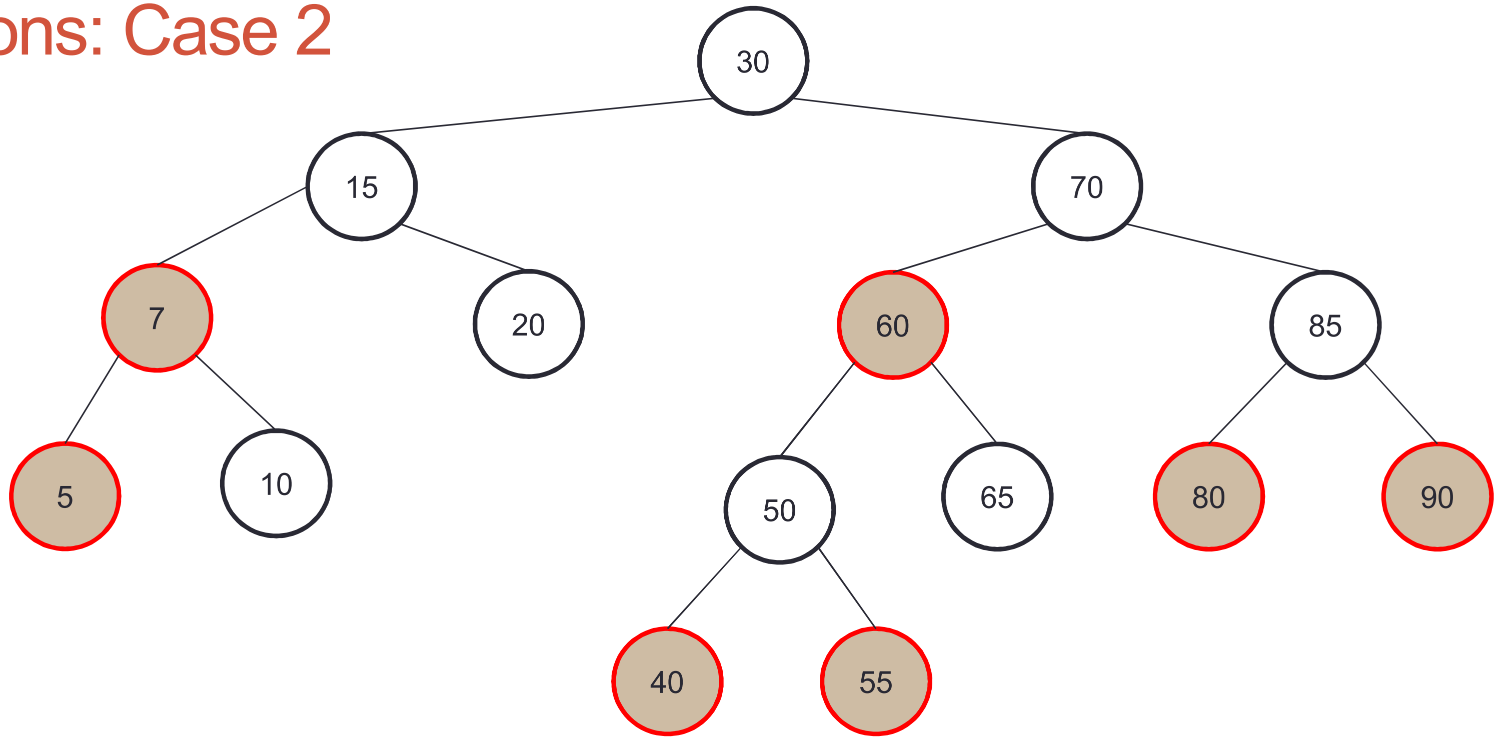
Insertions: Case 2



Insert 7

Case 2: Parent of leaf is red, parent is a left child of grandparent,
leaf is right child of parent, (& sibling of parent is black)

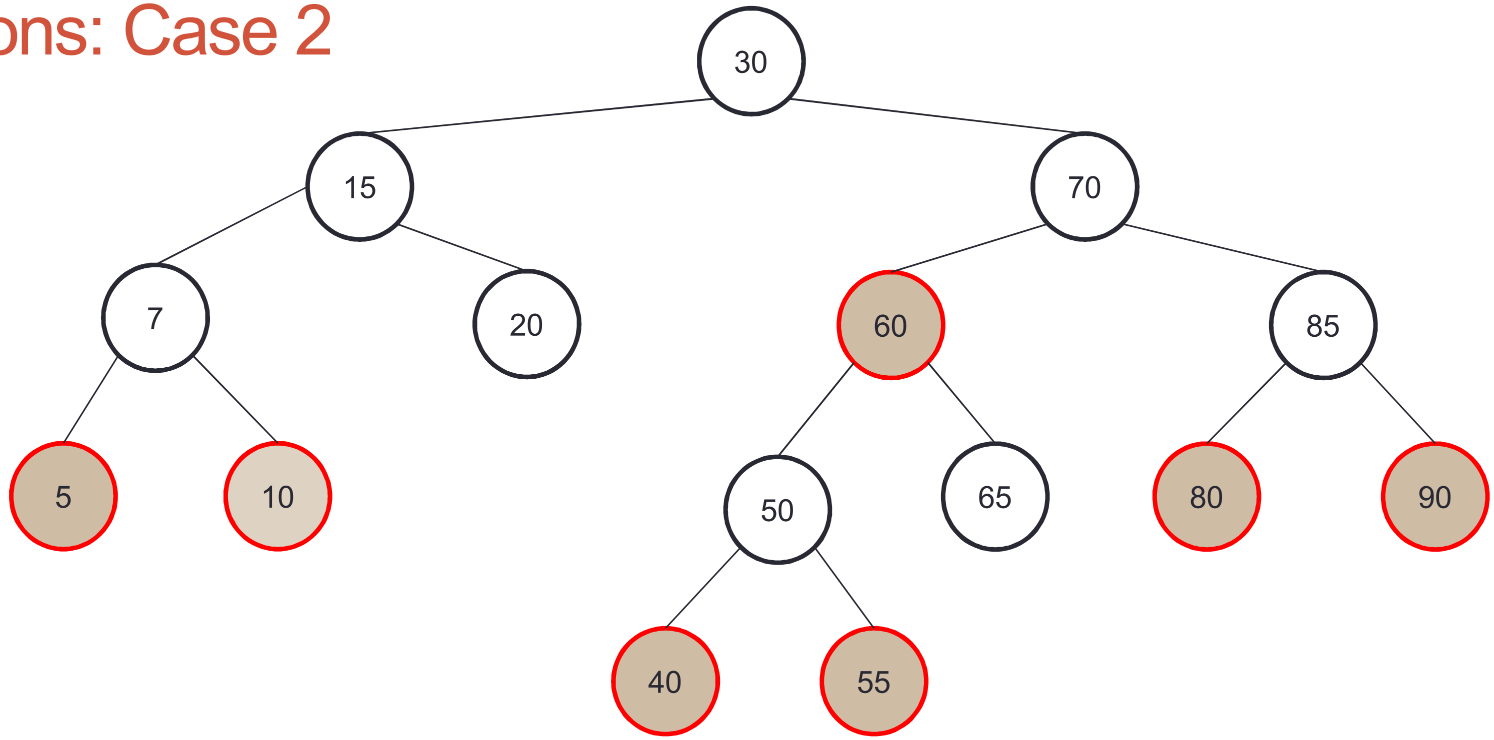
Insertions: Case 2



Insert 7 Recolor!

Case 2: Parent of leaf is red, parent is a left child of grandparent,
leaf is right child of parent, (& sibling of parent is black)

Insertions: Case 2



Insert 7

Case 2: Parent of leaf is red, parent is a left child of grandparent,
leaf is right child of parent, (& sibling of parent is black)

Insertions: Summary, so far

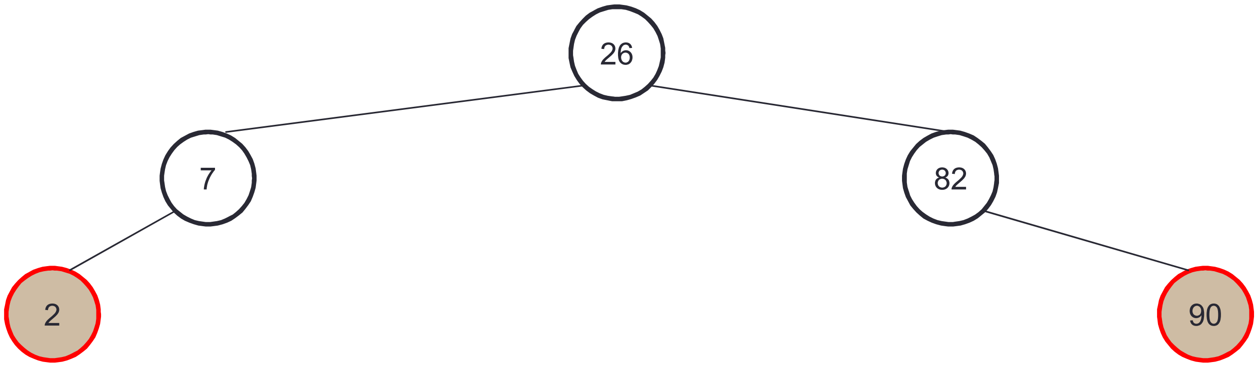
Case 0: The parent of the node you are inserting is black. Insert and you're done

For the remaining cases, the parent of the node is red, the sibling of the parent is black:

Case 1: P is left child of G, X is left child of P (single rotate then recolor)
Case 2: P is left child of G, X is right child of P (double rotate then recolor)

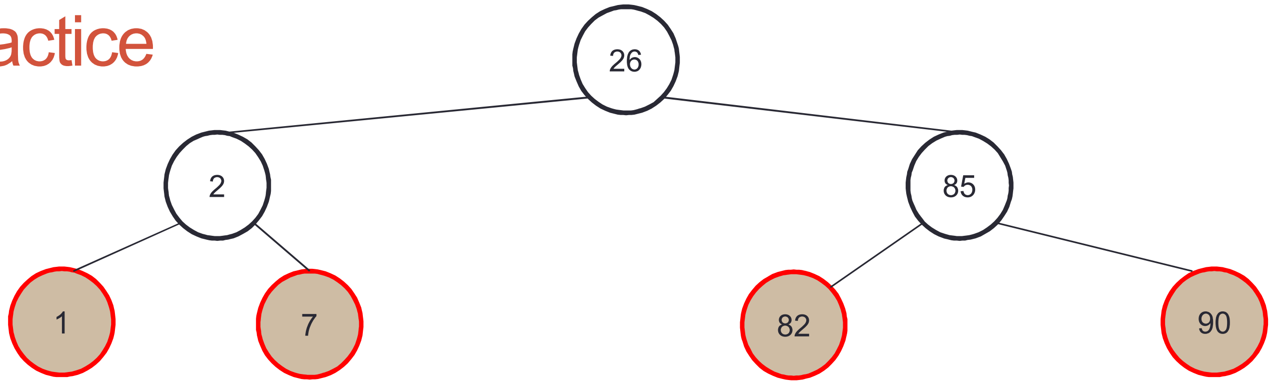
Case 3: P is right child of G, X is right child of P
Case 4: P is right child of G, X is left child of P

Practice



Insert 1 and then insert 85. Draw the resulting tree.

Practice



The final tree

Insertions: Summary, so far

Case 0: The parent of the node you are inserting is black. Insert and you're done

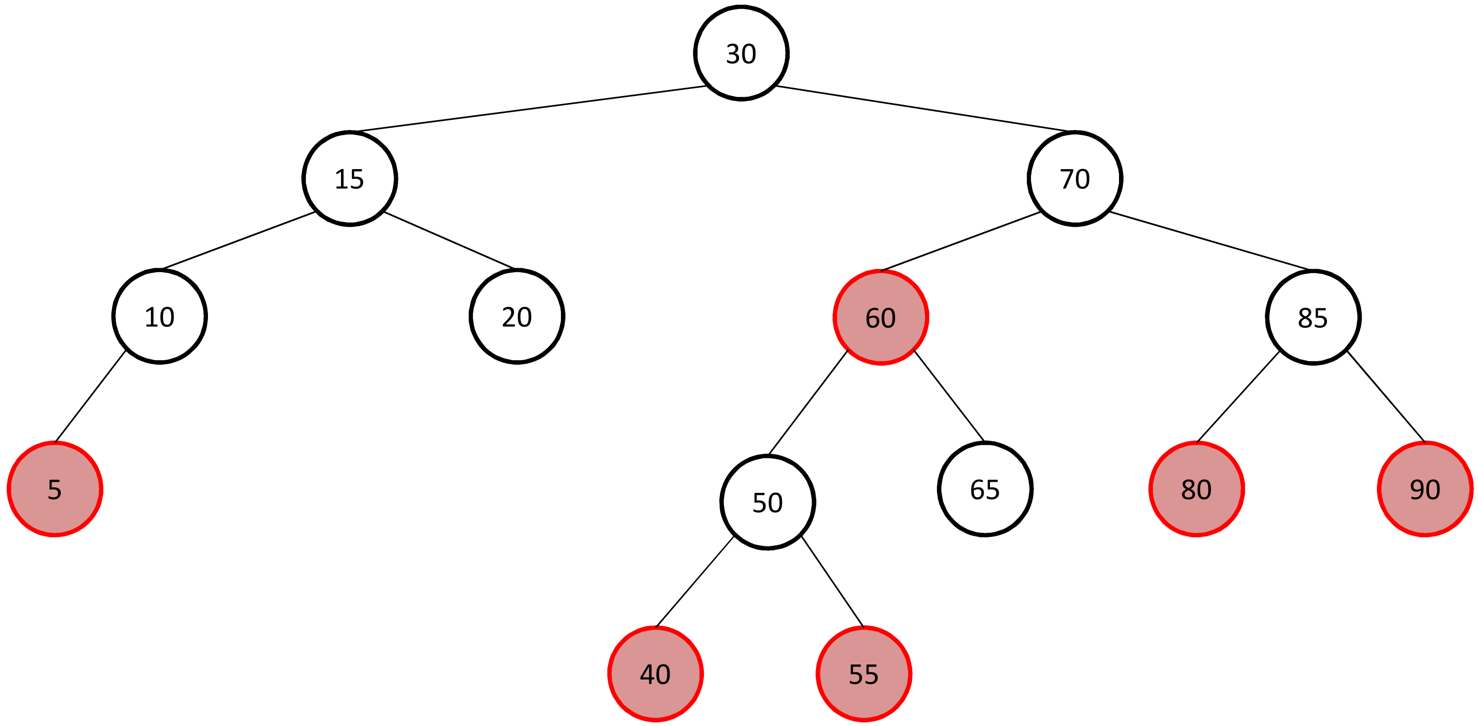
For the remaining cases, the parent of the node is red, **the sibling of the parent is black**:

Case 1: P is left child of G, X is left child of P (single rotate then recolor)
Case 2: P is left child of G, X is right child of P (double rotate then recolor)

Case 3: P is right child of G, X is right child of P
Case 4: P is right child of G, X is left child of P

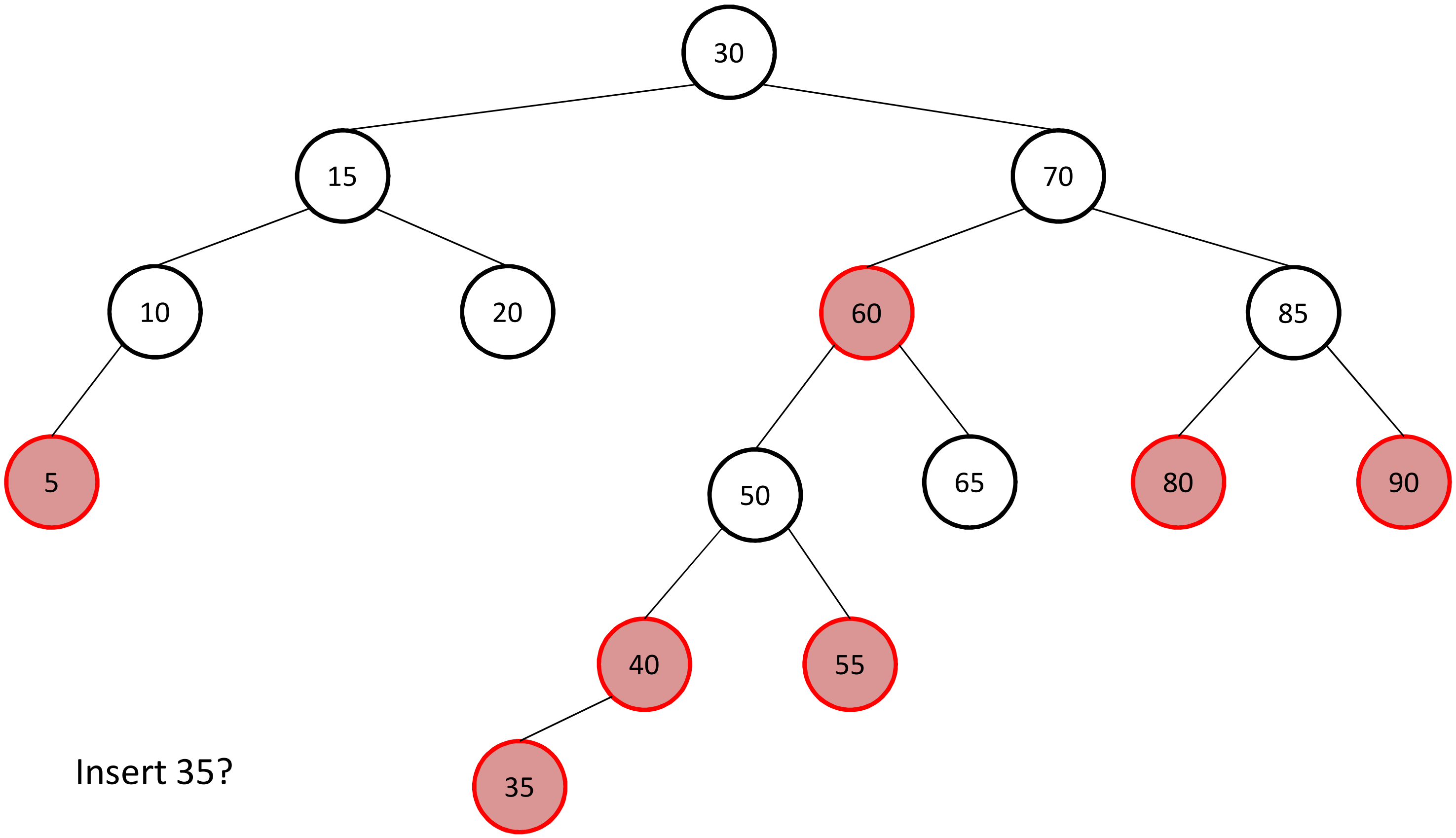
What if the sibling of the parent is red??

Insertions: Parent's sibling is red



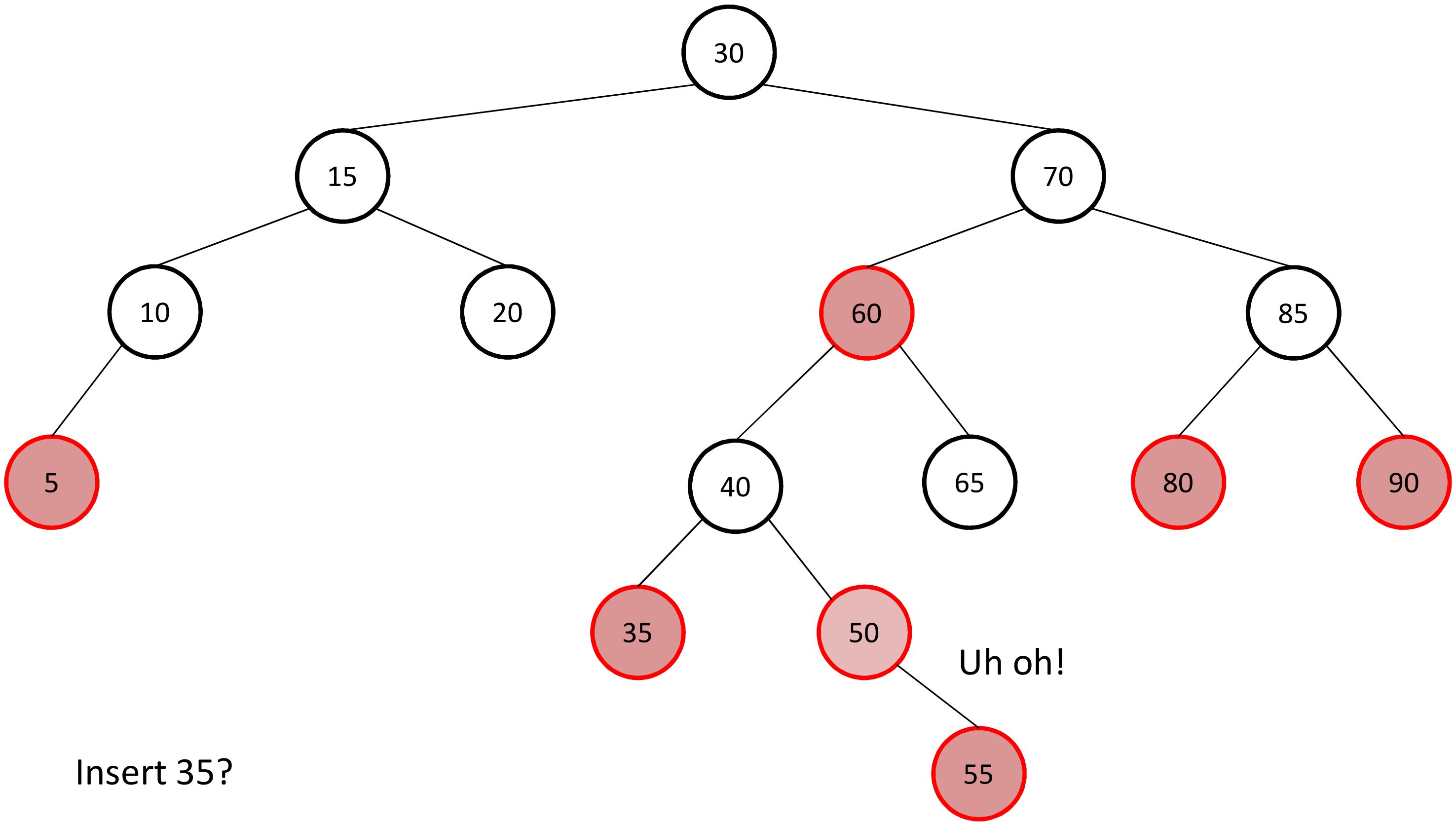
Insert 35?

Insertions: Parent's sibling is red



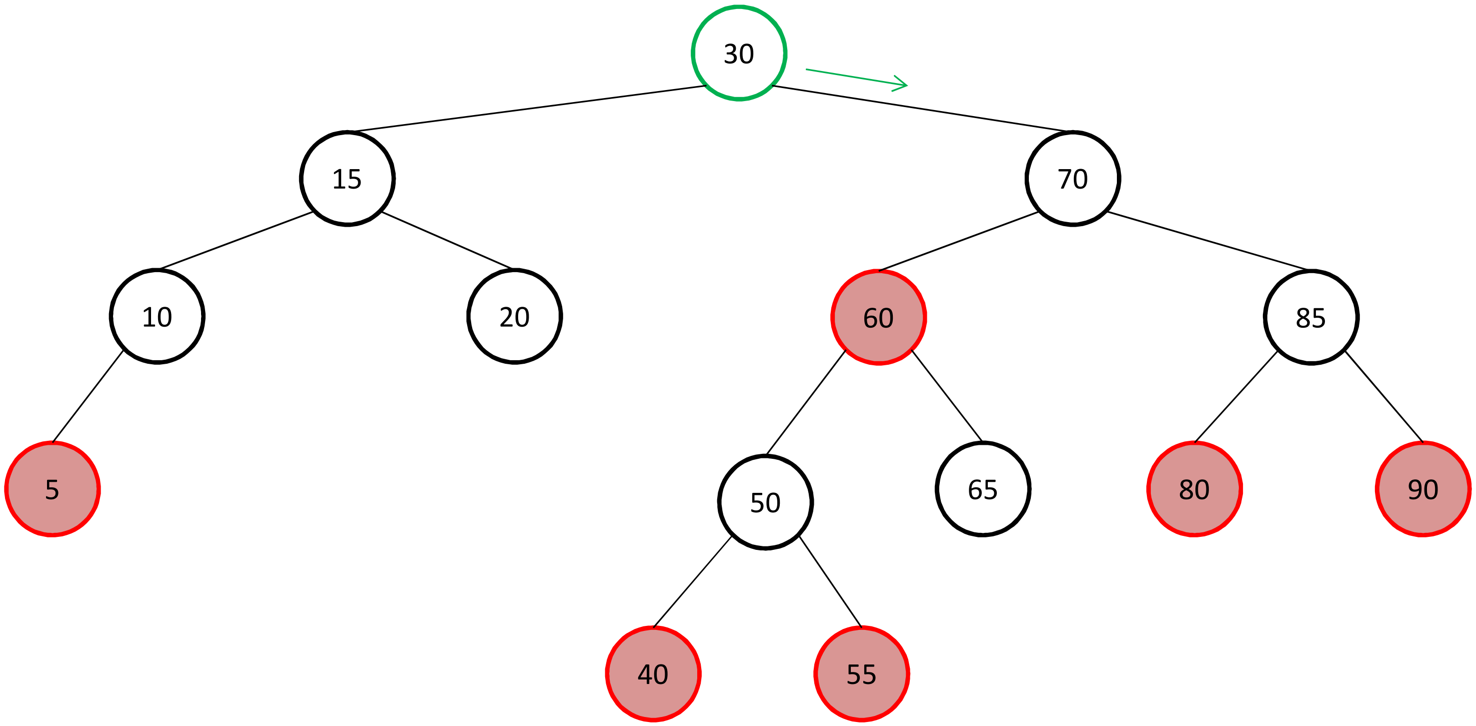
Insert 35?

Insertions: Parent's sibling is red



Insert 35?

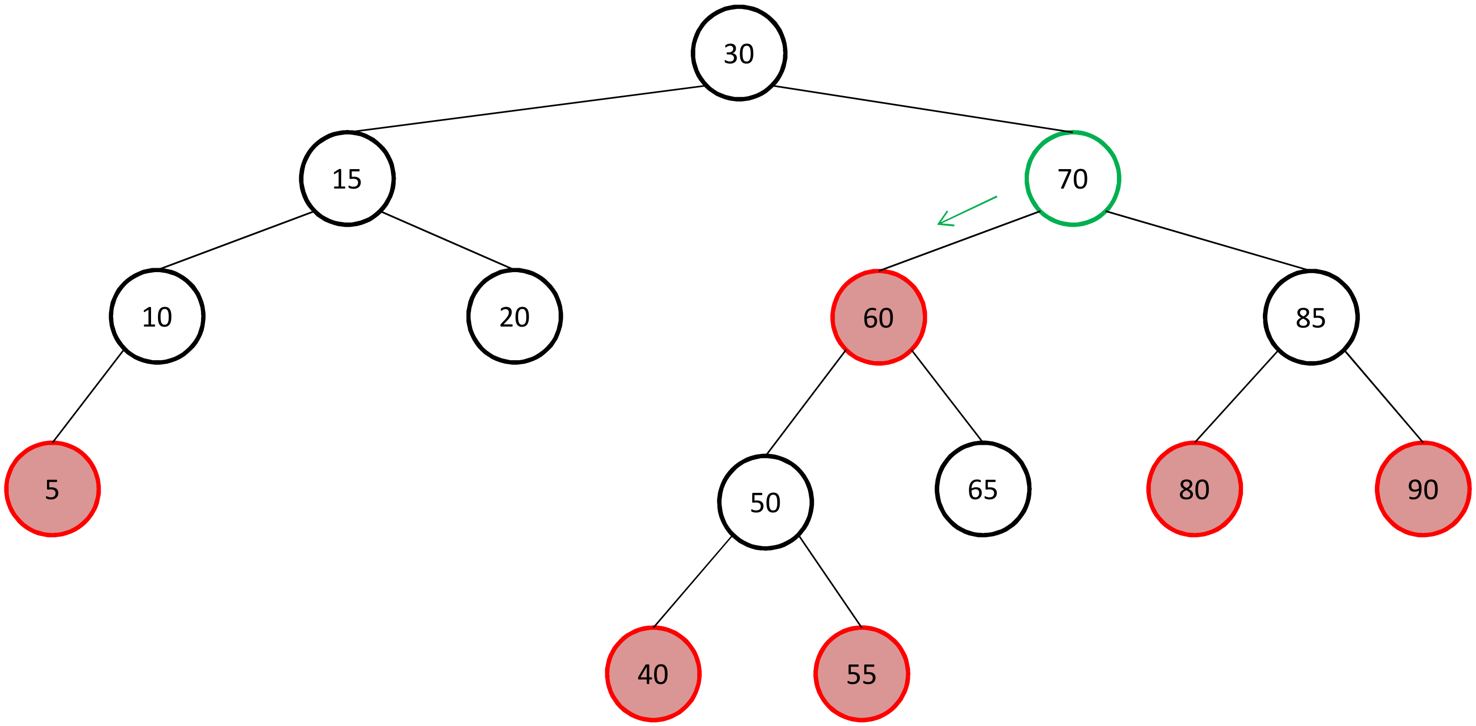
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

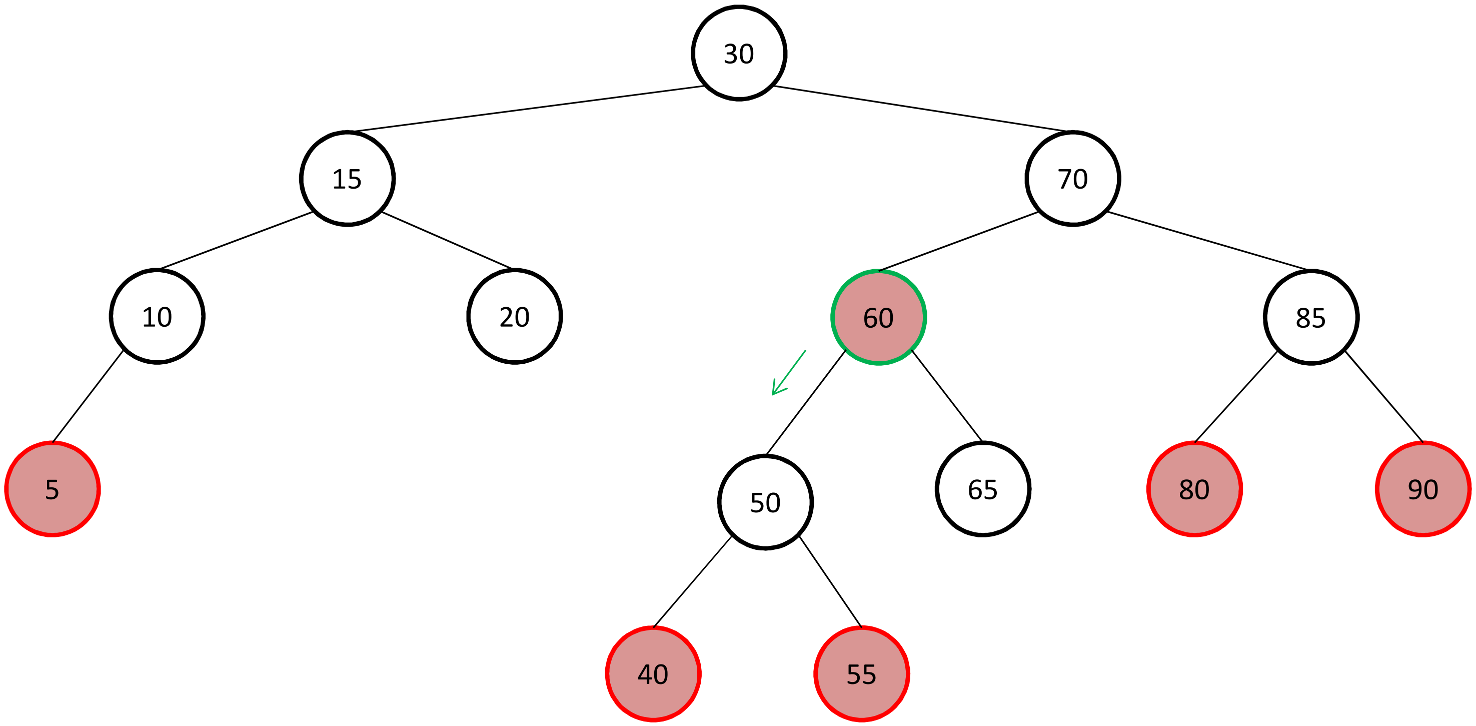
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

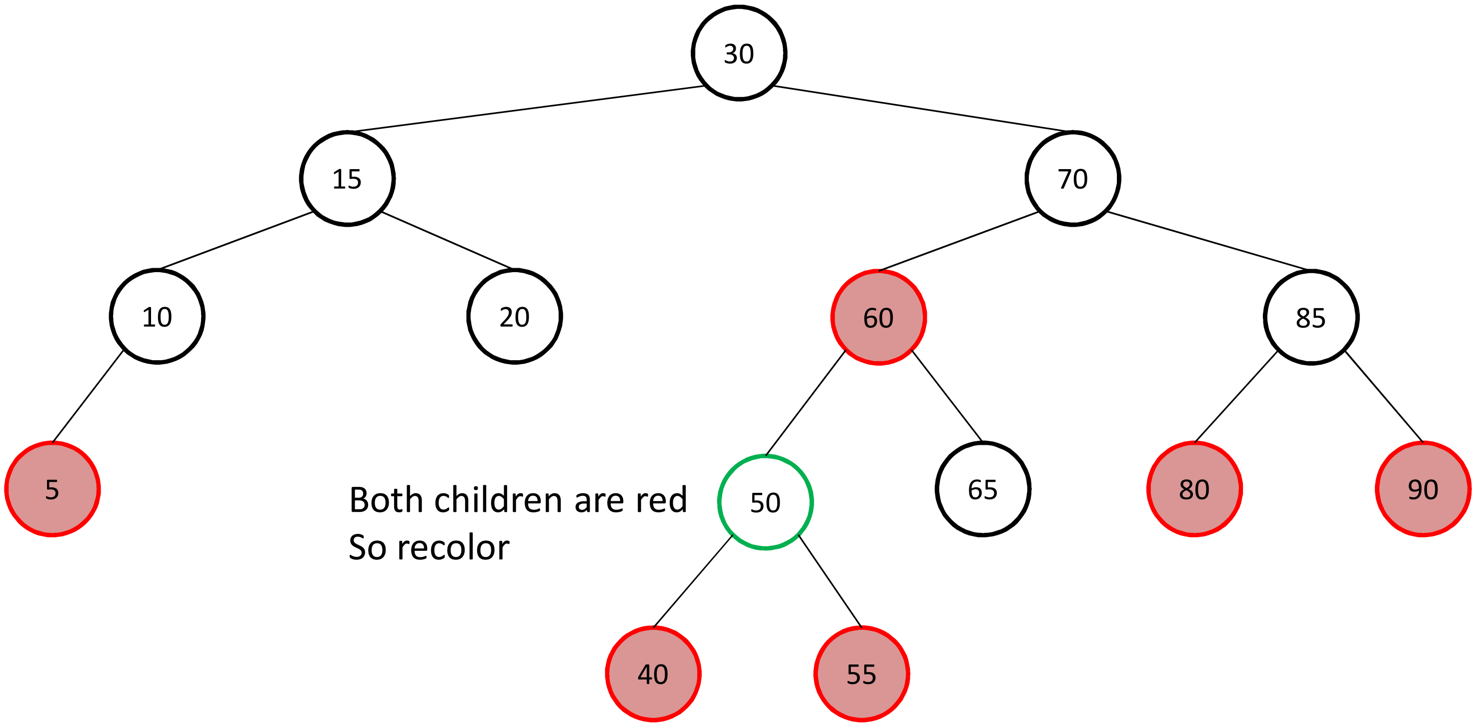
Insertions: Parent’s sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it’s children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Insertions: Parent's sibling is red

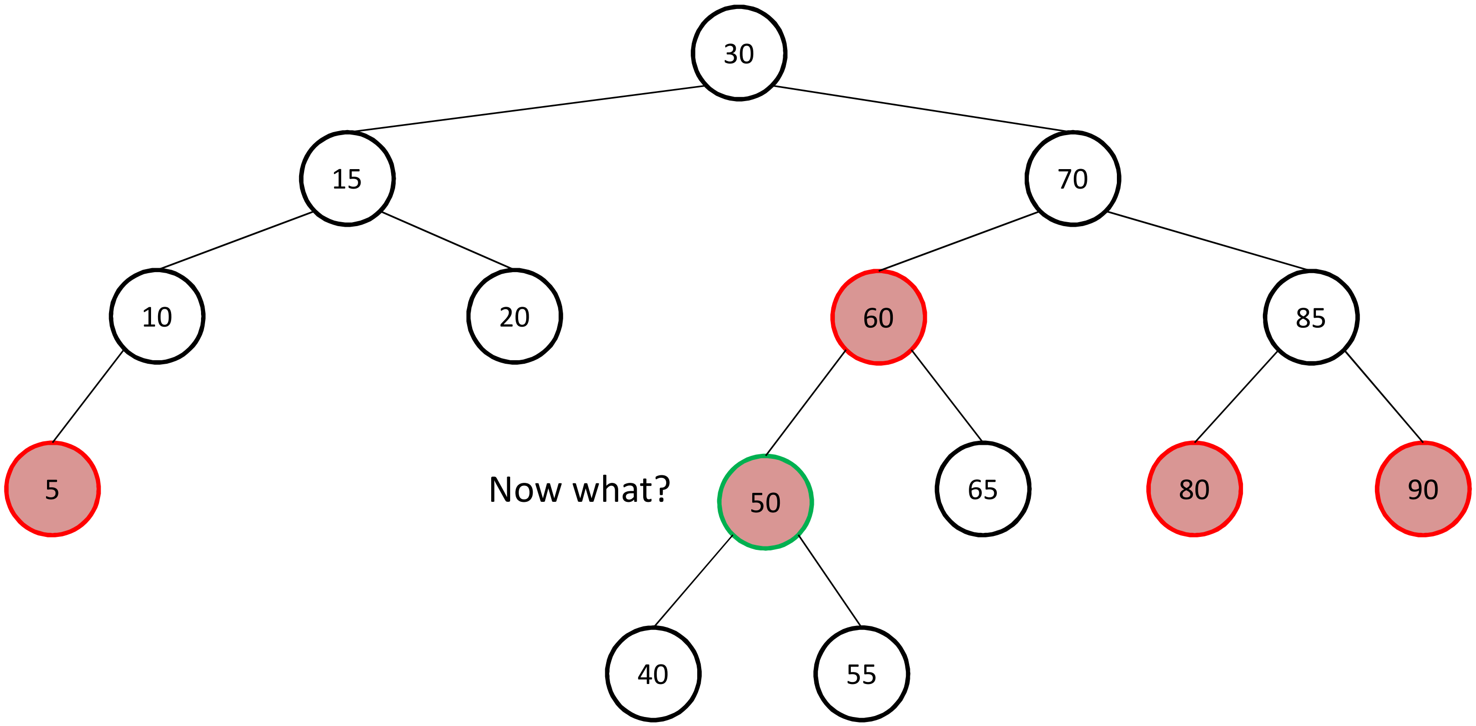


Both children are red
So recolor

Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Insertions: Parent's sibling is red

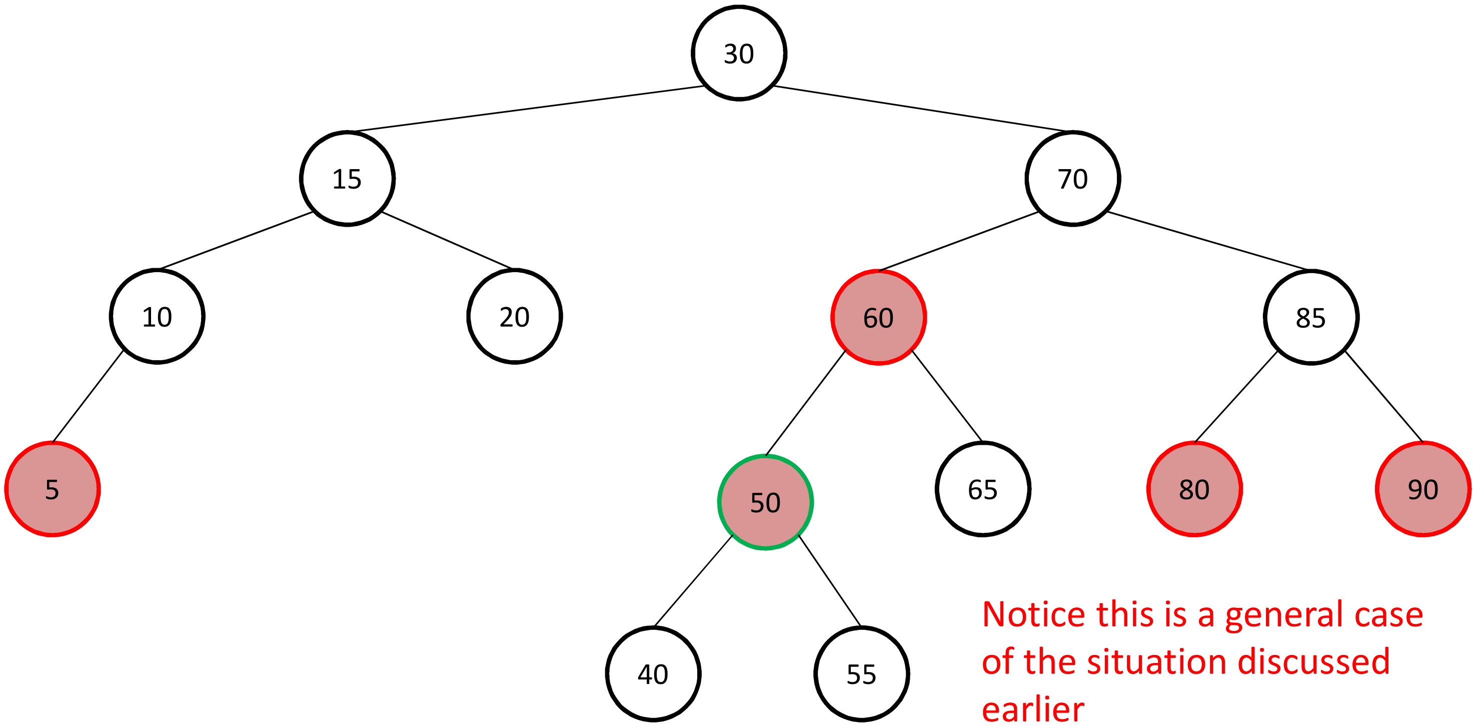


Insert 35?

Property 4 is okay, but 3 is not.
But wait – we've seen this before!

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Insertions: Parent's sibling is red



Notice this is a general case of the situation discussed earlier

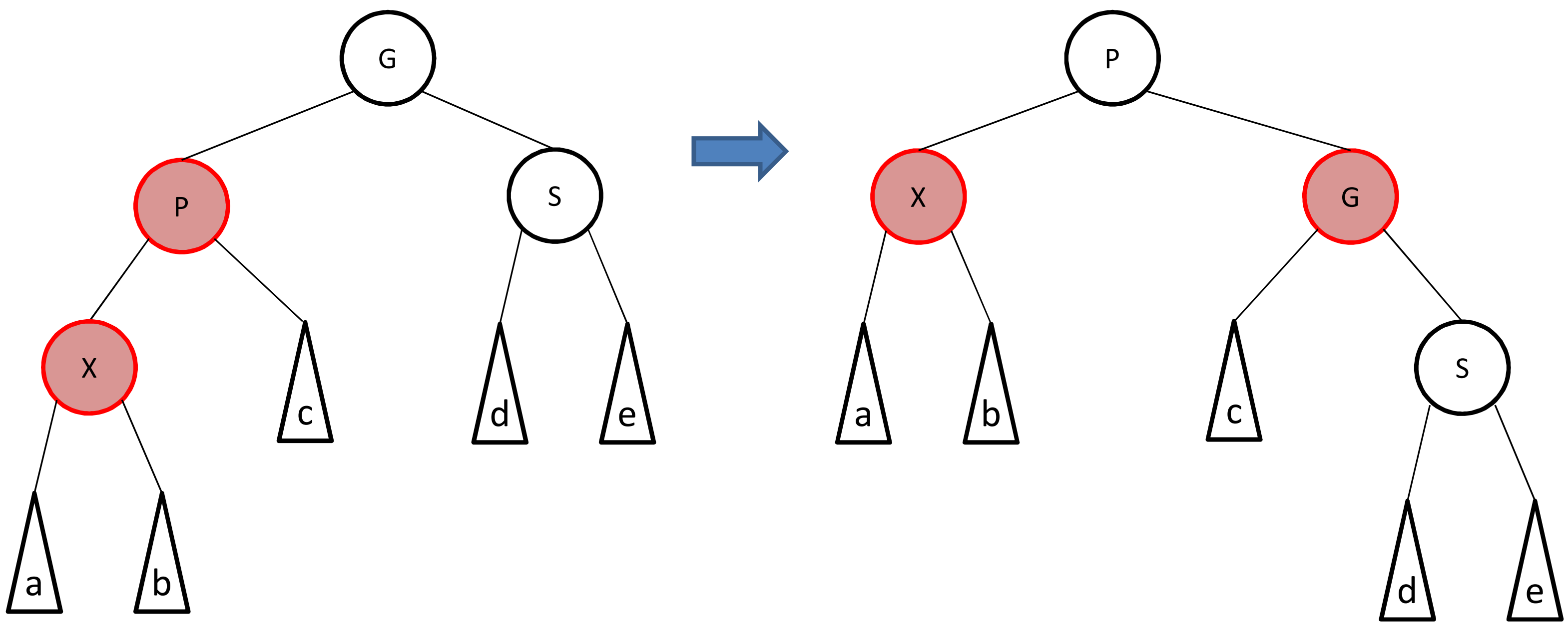
Insert 35?

We've seen this case, must Rotate at 70.

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

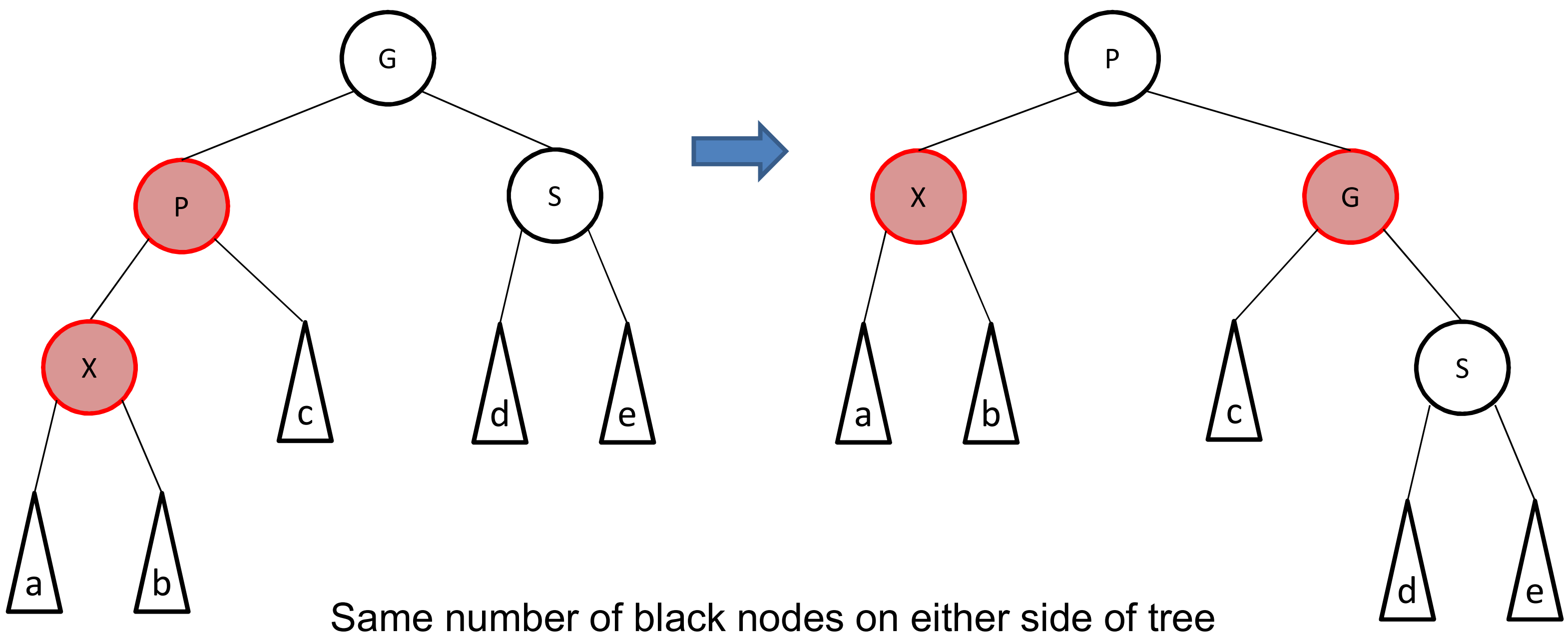
Case 1 in general

(assume this is a legal red-black tree, i.e. there are black nodes hidden in the subtrees)



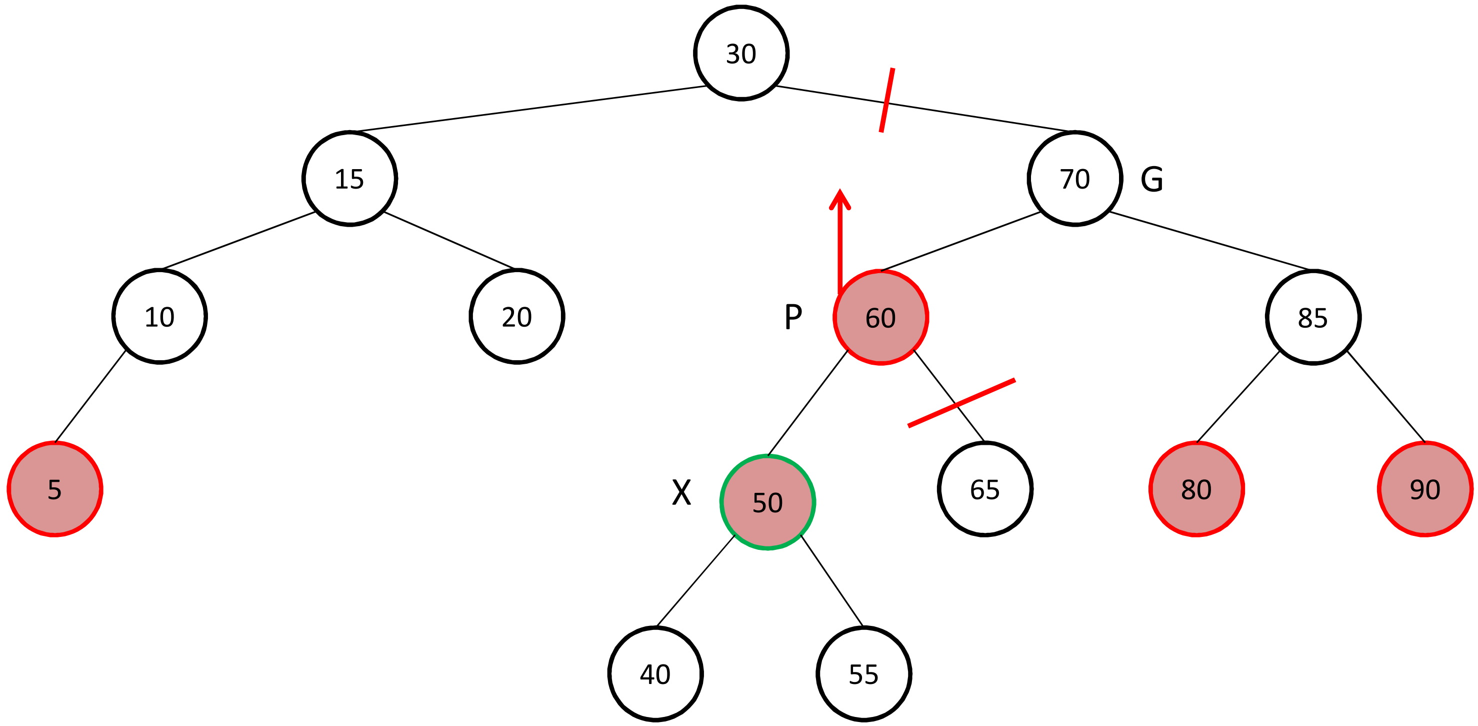
If X's Parent (P) is red, P is a left child of G, X is a left child of P, (and P's sibling (S) is black), then
Rotate P right, flip colors of P and G

Case 1 in general



Same number of black nodes on either side of tree
Roots of subtrees a, b and c (and node S) must be black
X's and G's parent is now guaranteed to be black
BST property preserved through AVL rotations

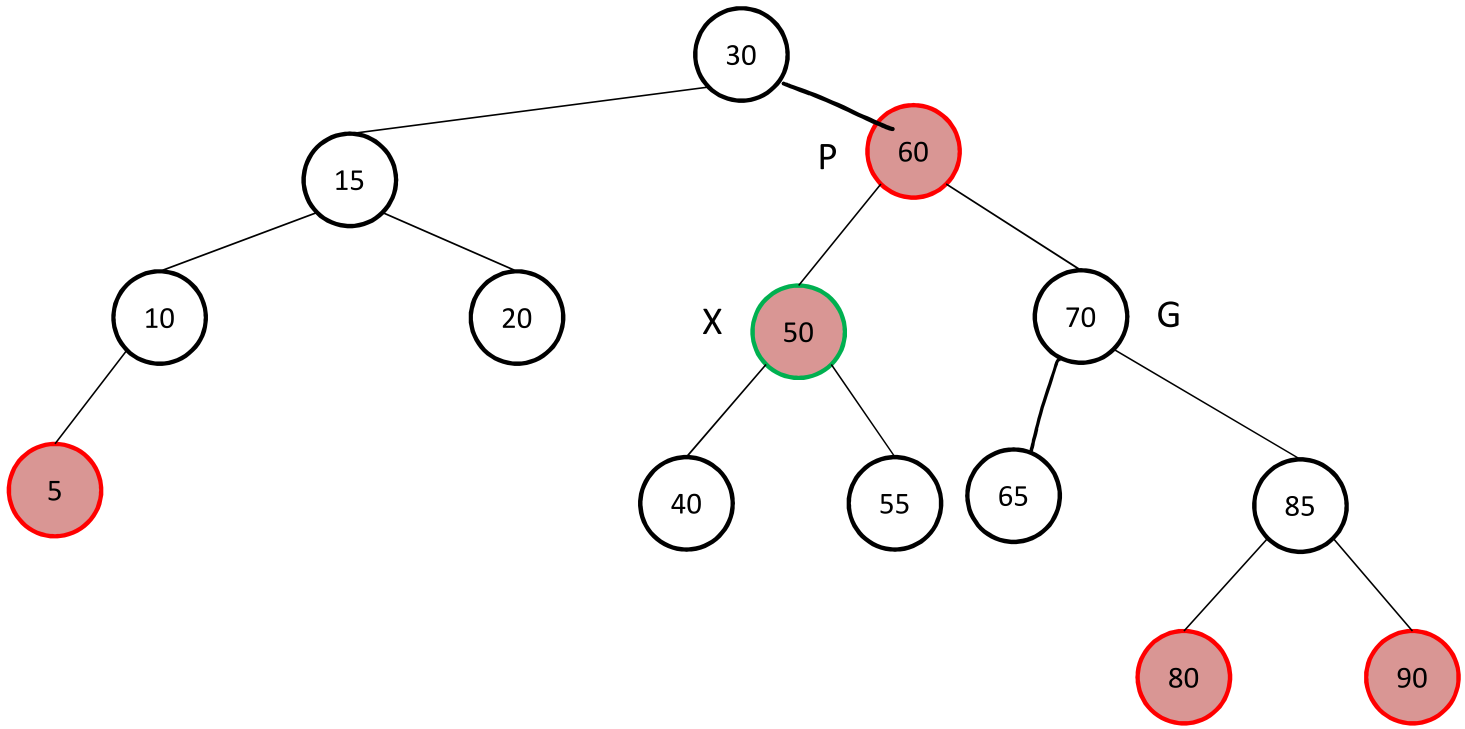
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

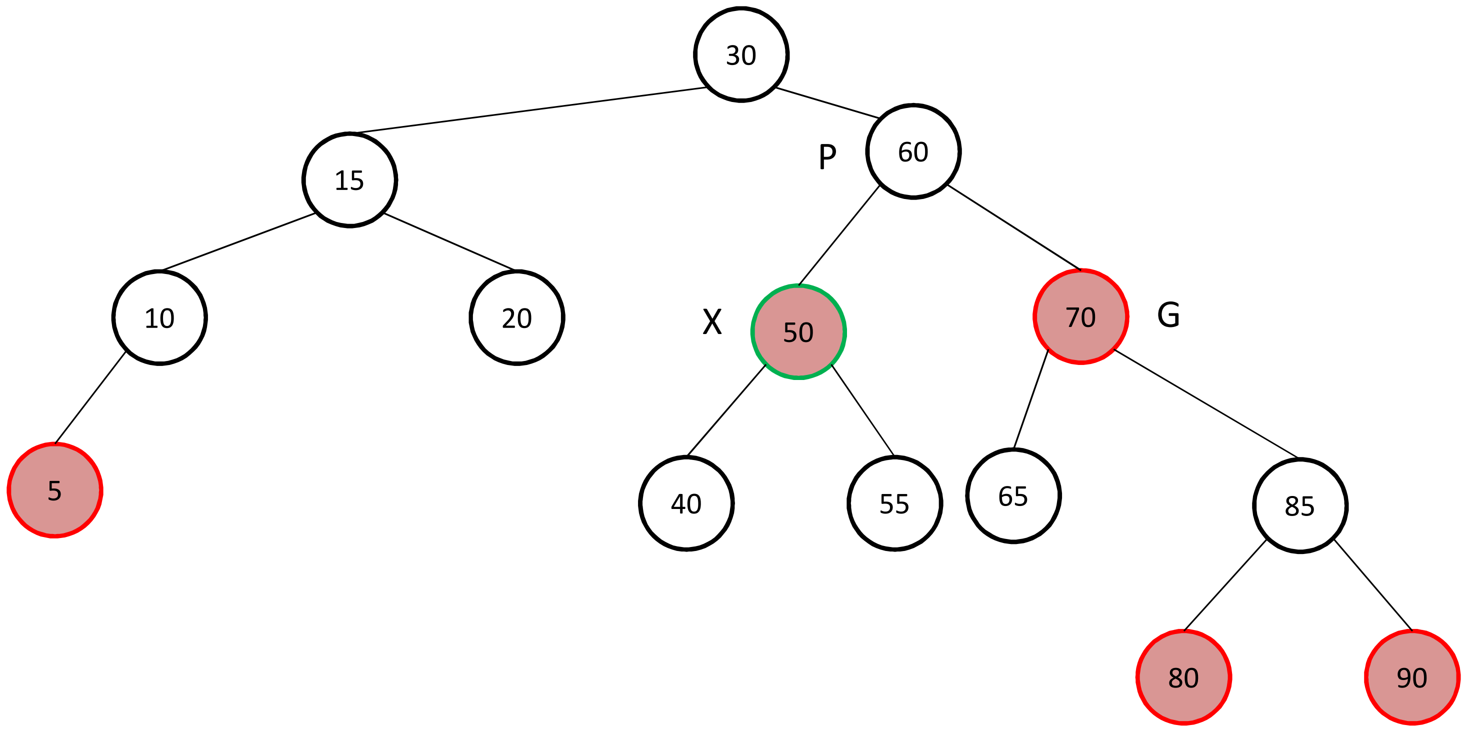
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

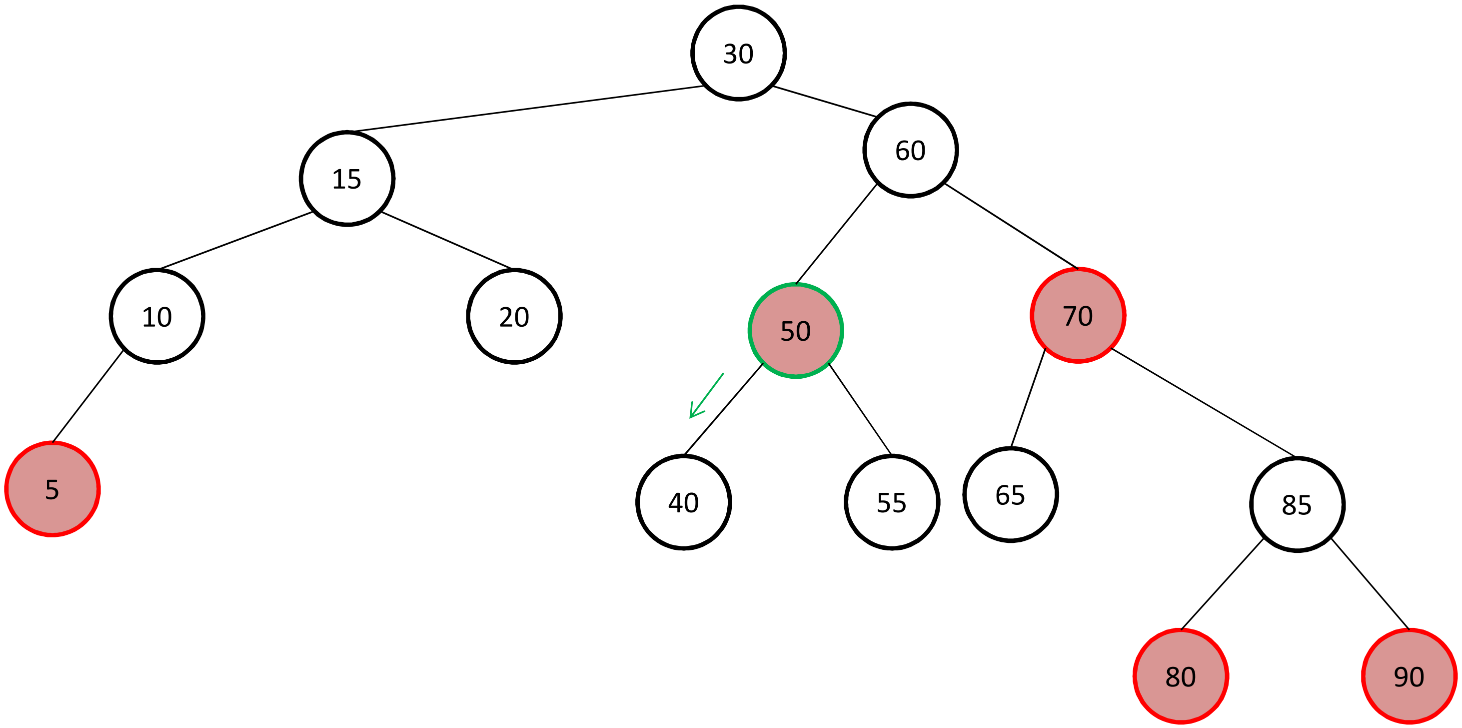
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

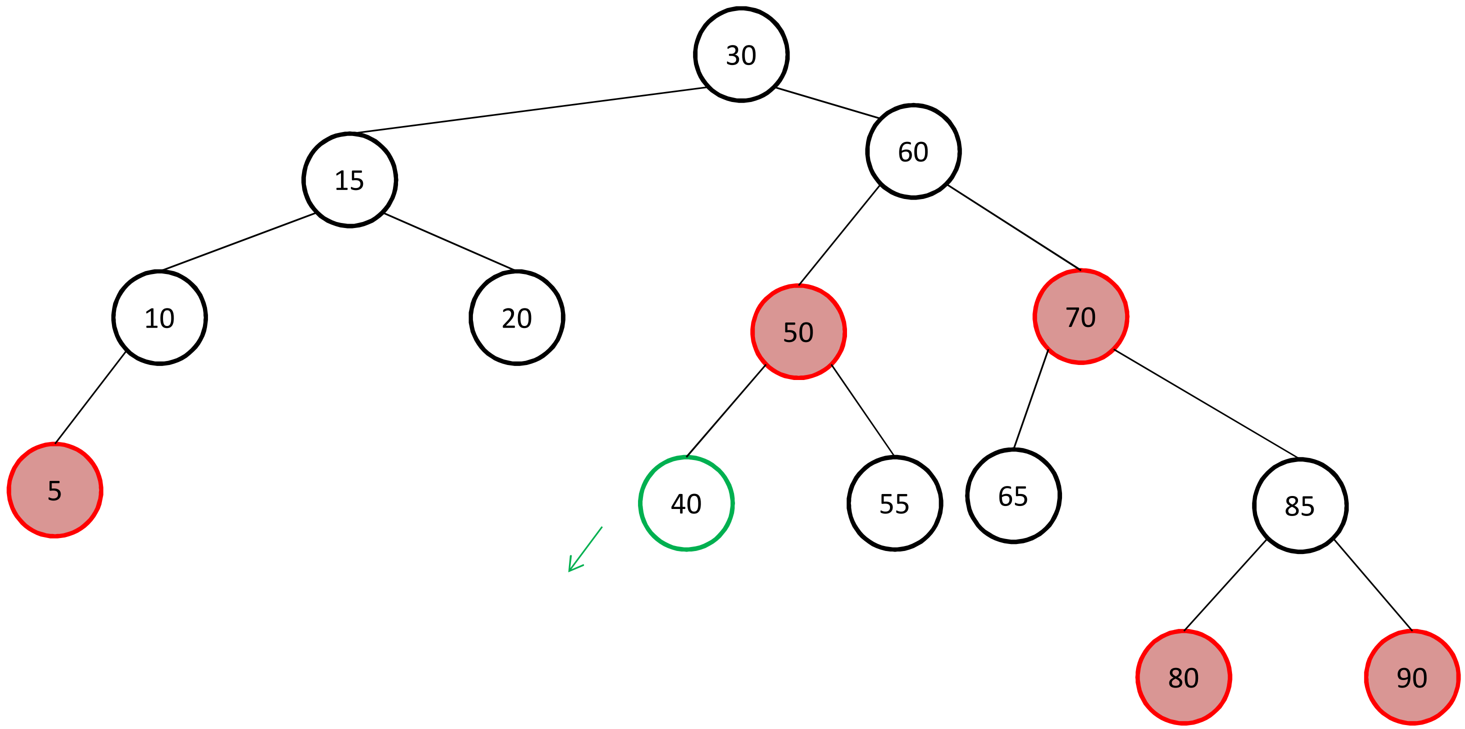
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

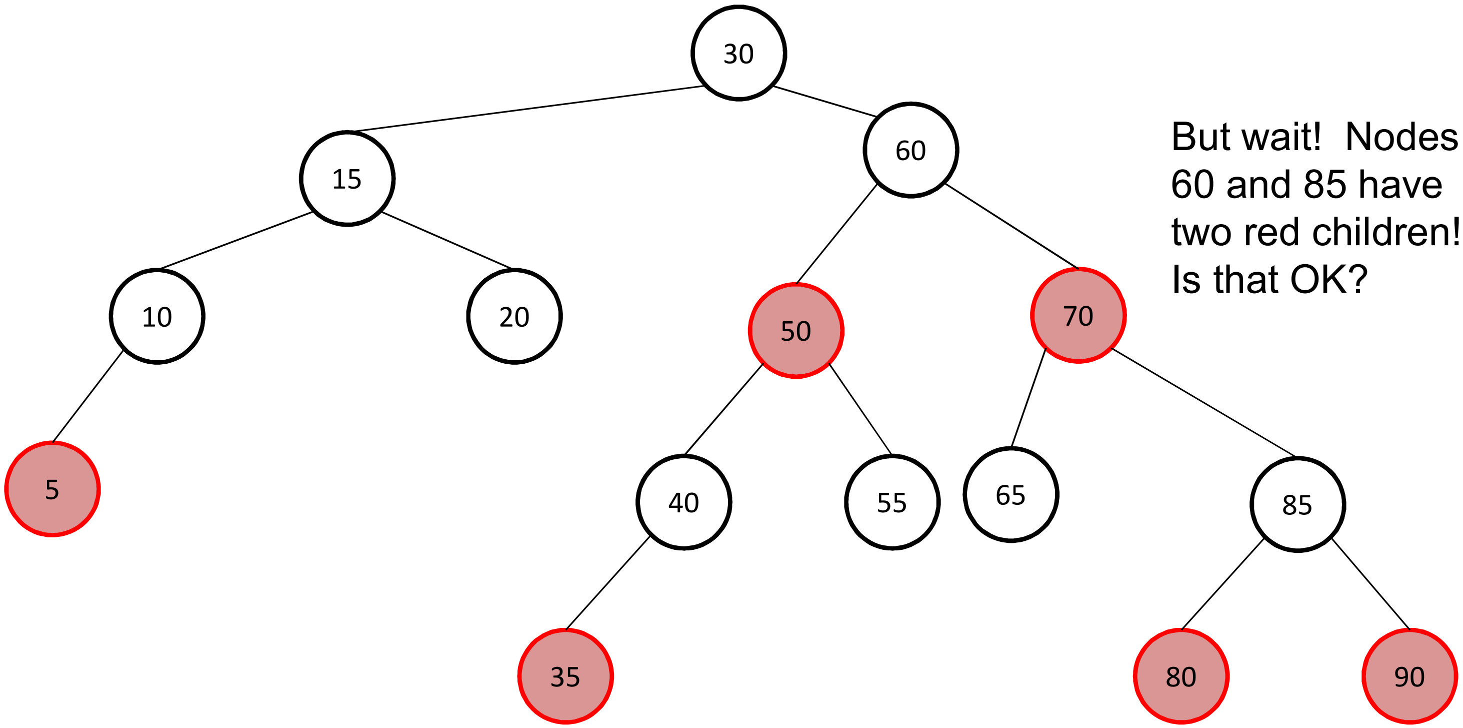
Insertions: Parent's sibling is red



Insert 35?

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Insertions: Parent's sibling is red



But wait! Nodes 60 and 85 have two red children! Is that OK?

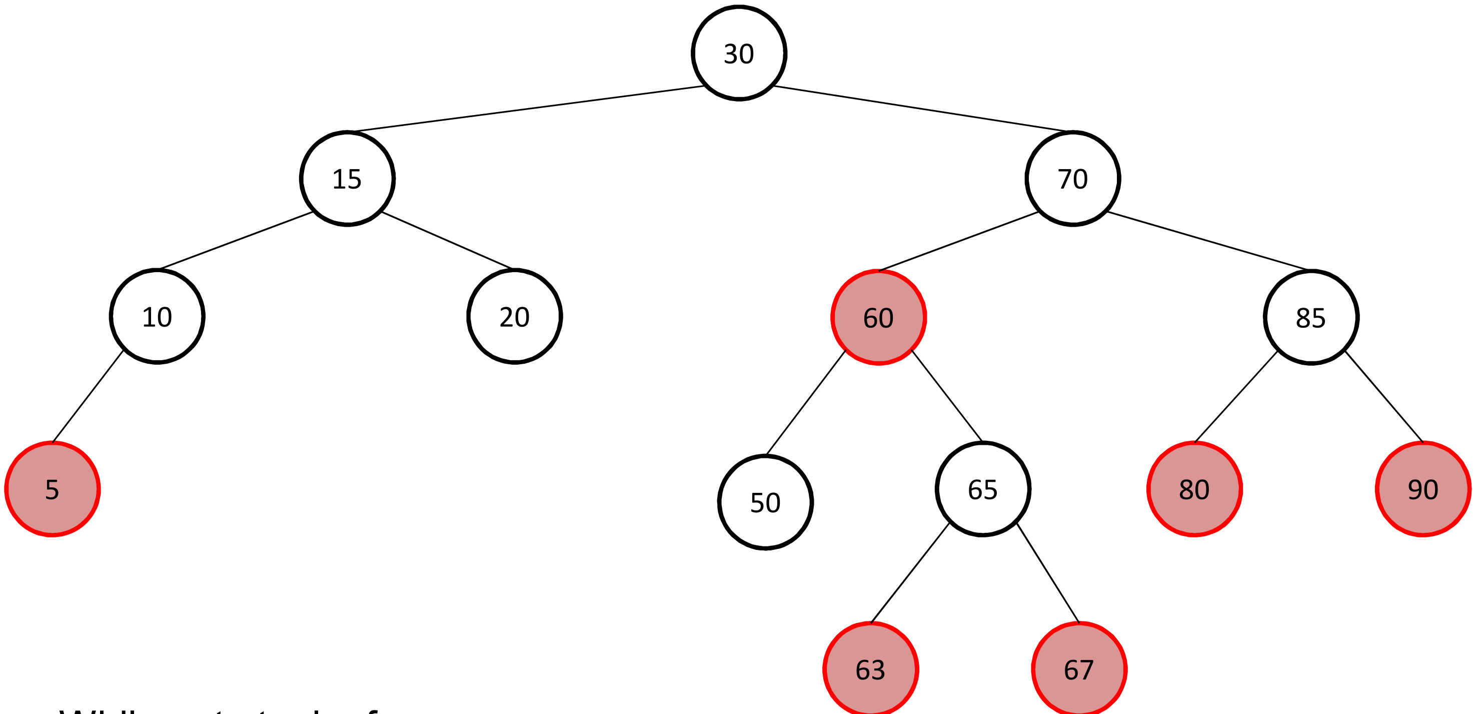
Insert 35? DONE!

1. Nodes are either red or black
2. Root is always black
3. If a node is red, all it's children must be black
4. For every node X, every path from X to a null reference must contain the same number of black nodes

Can any node have 2 red children?

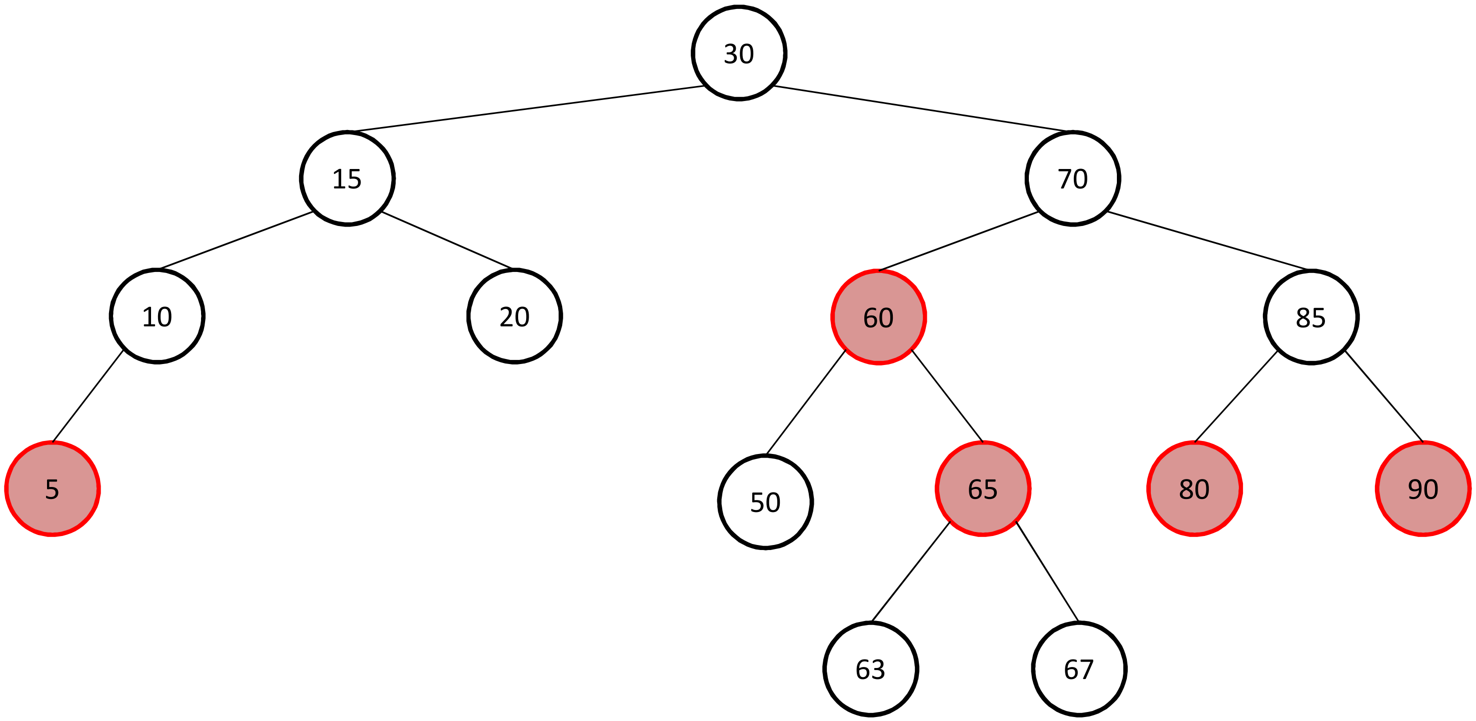
- As we descend the tree, we detect if a node X has 2 red children, and if so we do an operation to change the situation
- Note that in doing so:
  - we may change things so that a node above X now has 2 red children, where it didn't before! (example: node 60 after we insert 35)
  - if we have to do a double rotation, we will move X up and recolor it so that it becomes black, and has 2 red children itself! (example: work through inserting 64 in the tree on the following page)
- But neither of these is a problem, because
  - it never violates any of the properties of red-black trees (those 2 red nodes will always have a black parent, for example),
  - and the 2 red siblings will be too "high" in the tree for either of them to be the sibling of the parent of any red node that we find or create when we continue this descent of the tree

Exercise: Insert 64 into this tree



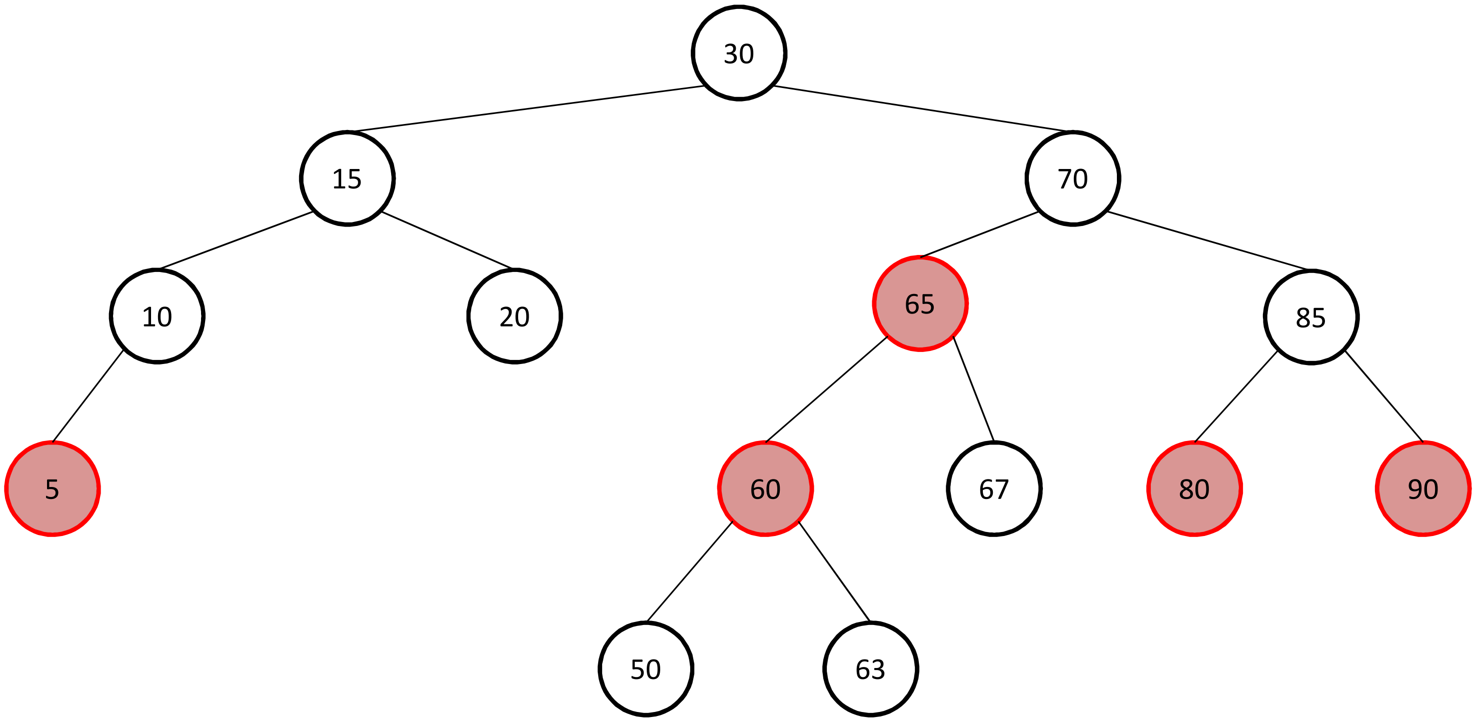
- While not at a leaf:
  - Move down the tree to where node should be placed
  - If you encounter a node with two red children, recolor, then perform any necessary rotations to fix the tree
- Insert the node
- Perform any necessary rotations to fix the tree

Exercise: Insert 64 into this tree



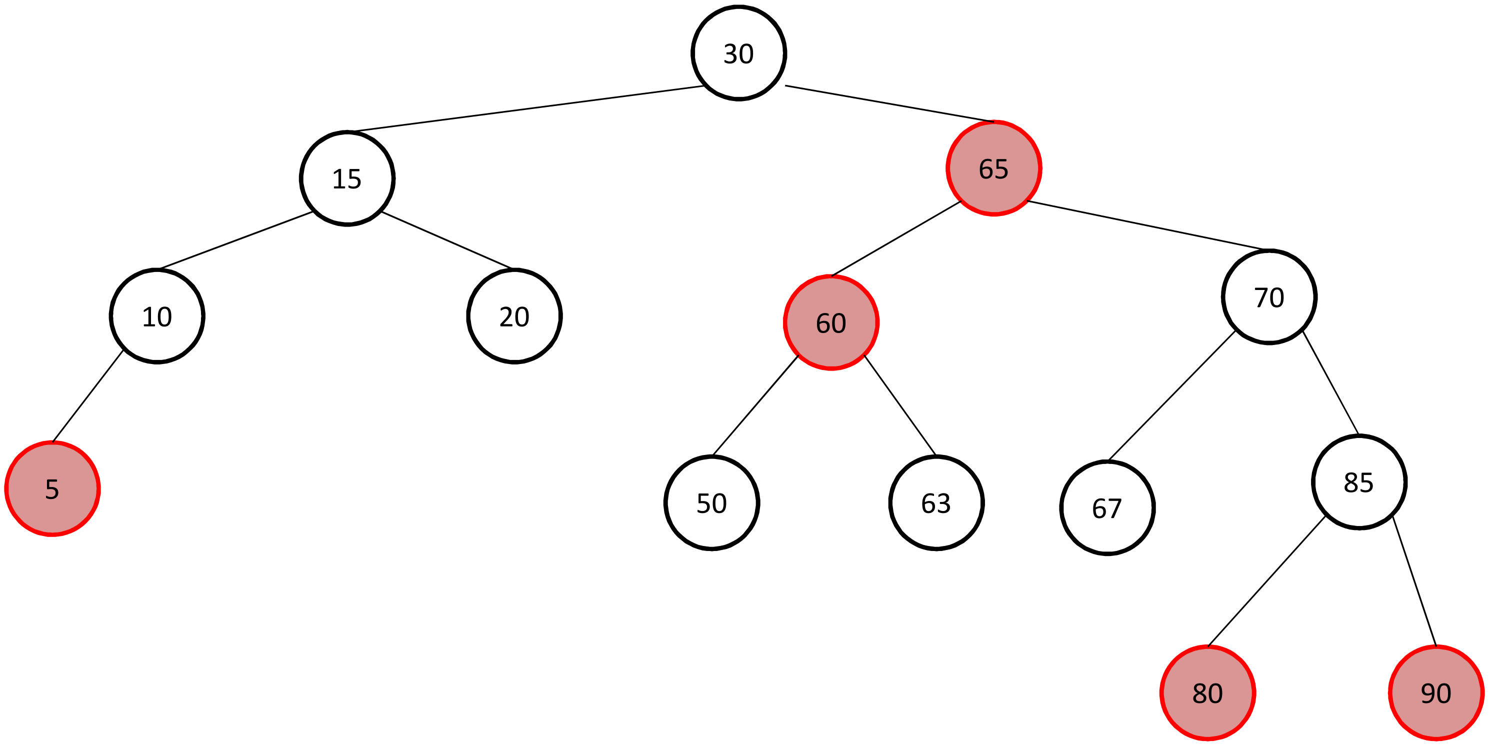
Recolor

Exercise: Insert 64 into this tree



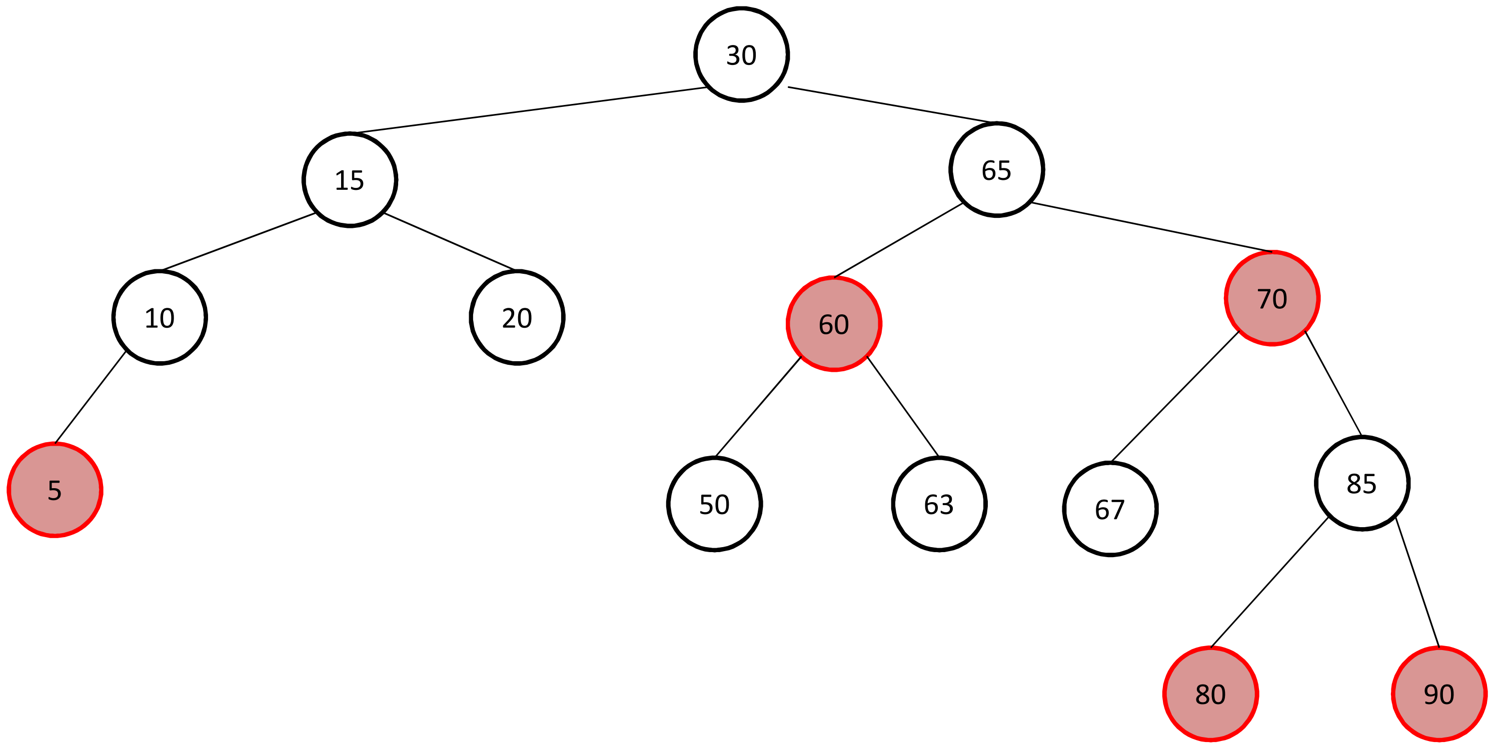
Double rotation (rotation 1)

Exercise: Insert 64 into this tree



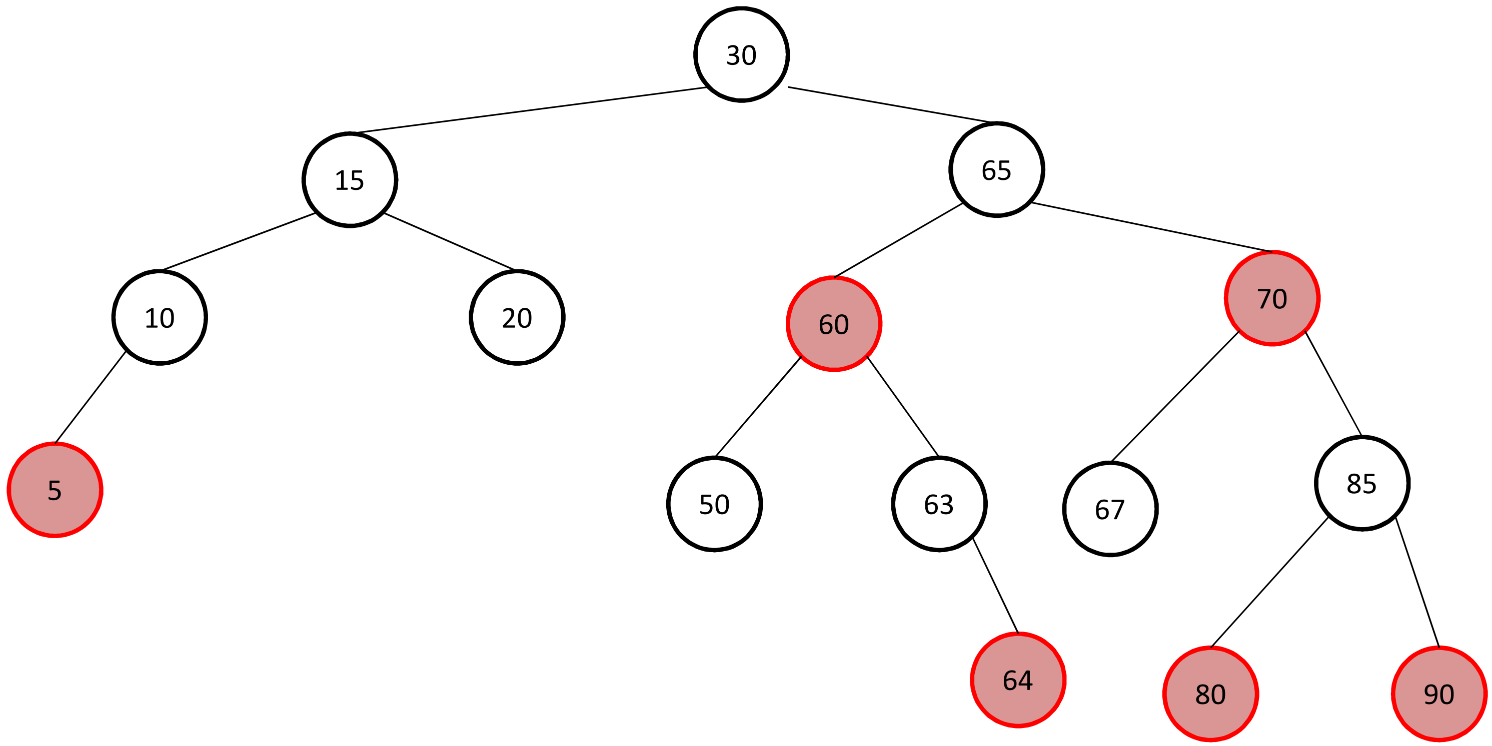
Double rotation (rotation 2)

Exercise: Insert 64 into this tree



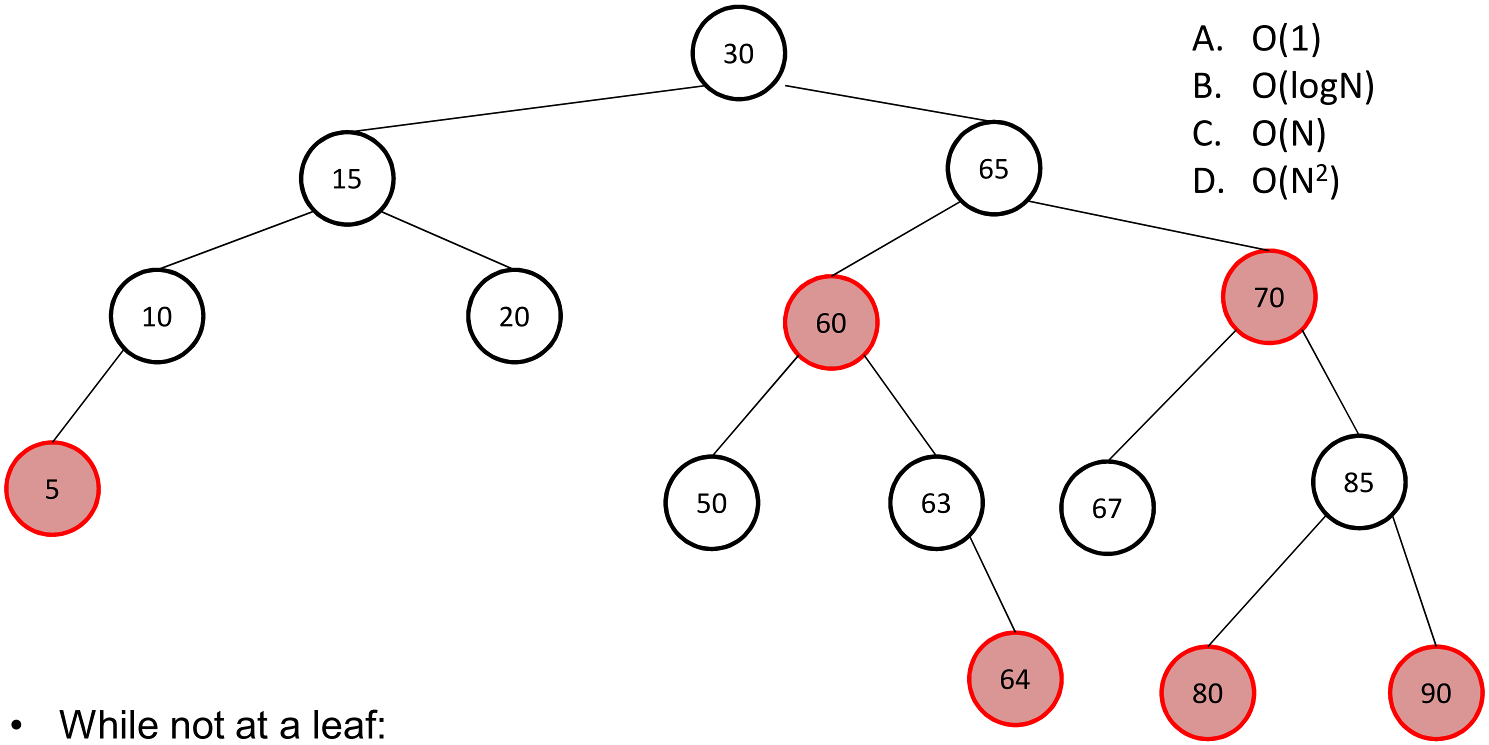
Recolor

Exercise: Insert 64 into this tree



Insert

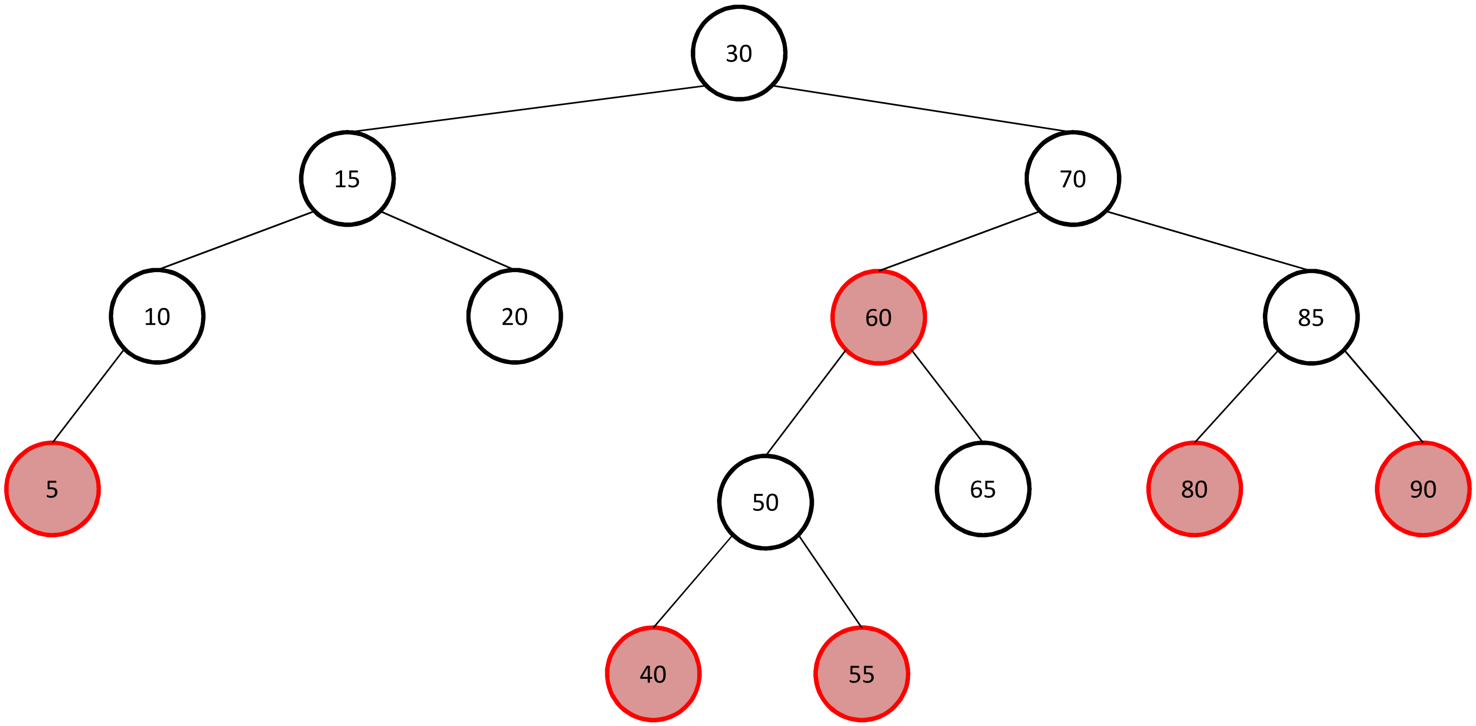
What is the Big-O running time of insert?



A. O(1)
B. O(logN)
C. O(N)
D. $O(N^2)$

- While not at a leaf:
  - Move down the tree to where node should be placed
  - If you encounter a node with two red children, recolor, then perform any necessary rotations to fix the tree
- Insert the node
- Perform any necessary rotations to fix the tree

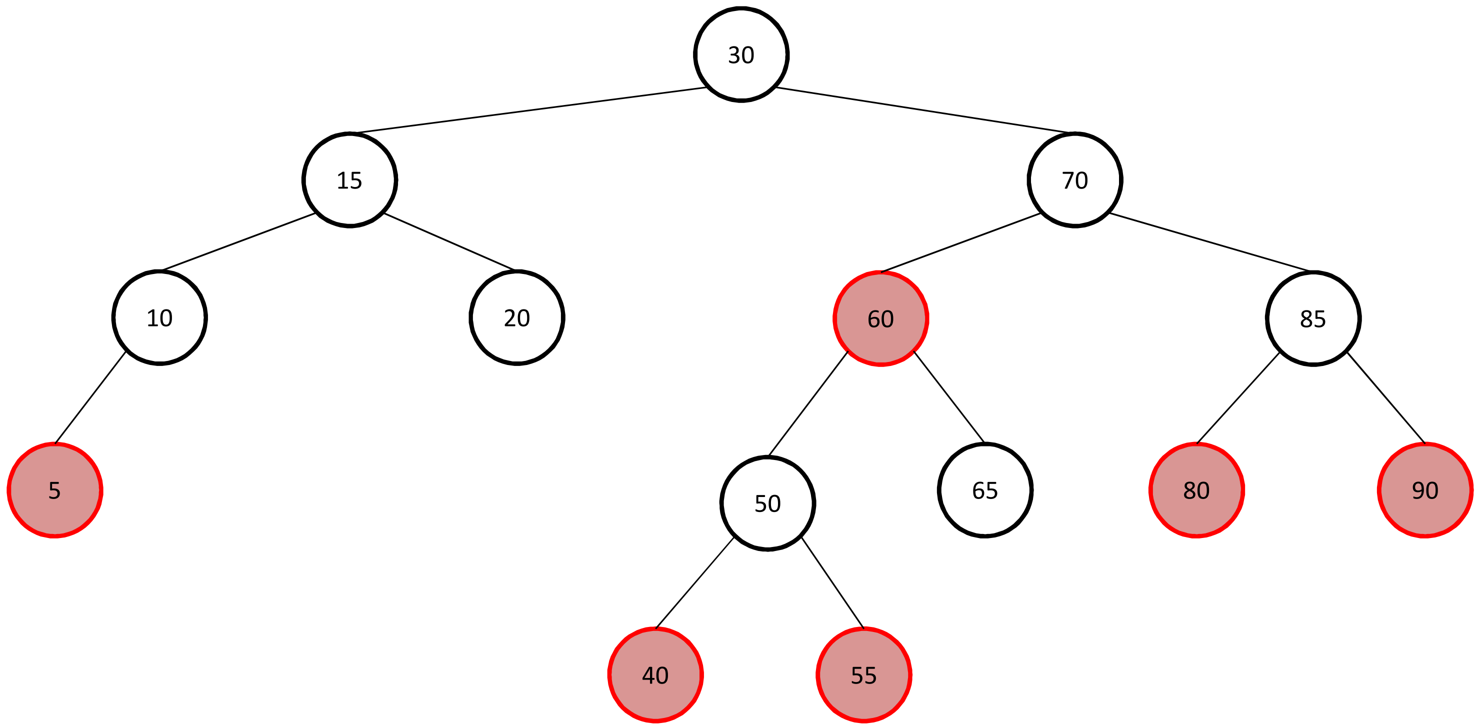
Red-Black Trees vs. AVL trees



Is this an AVL tree?

A. Yes
B. No

Red-Black Trees vs. AVL trees

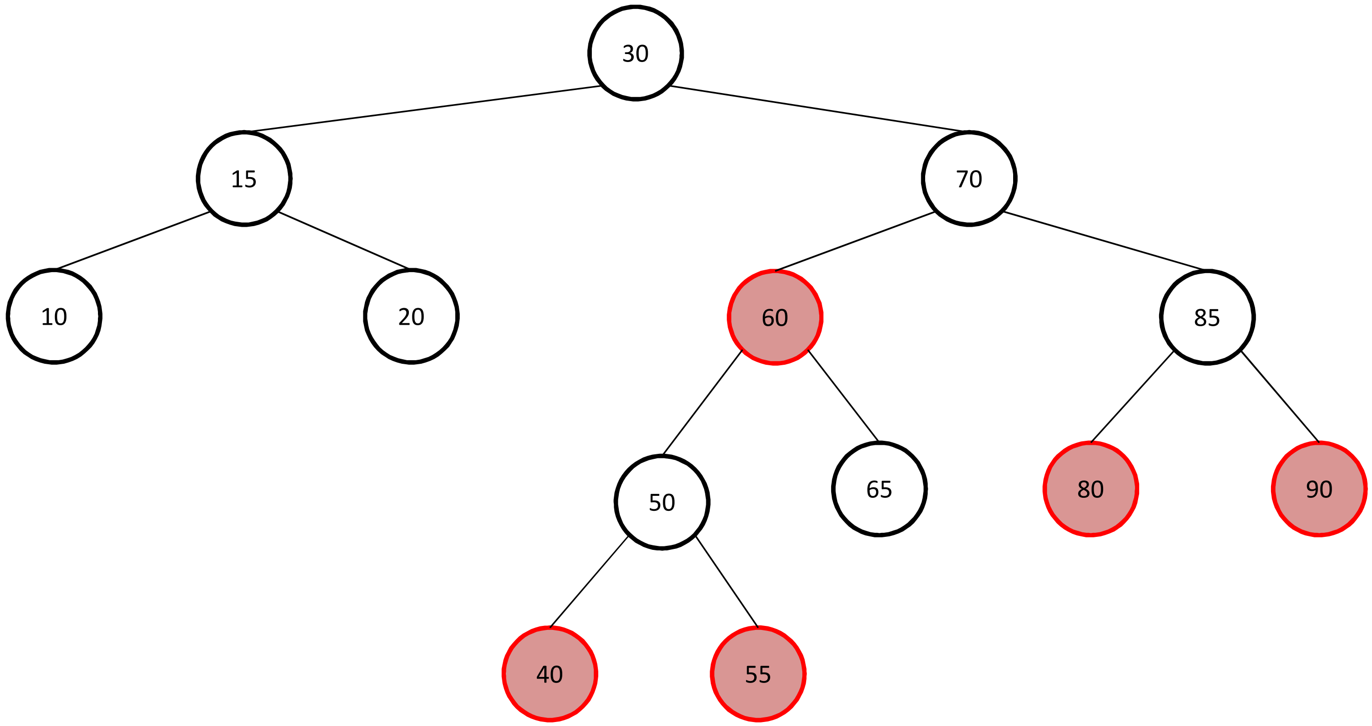


Is this an AVL tree?
Yes

Are all red black trees AVL trees?

A. Yes
B. No

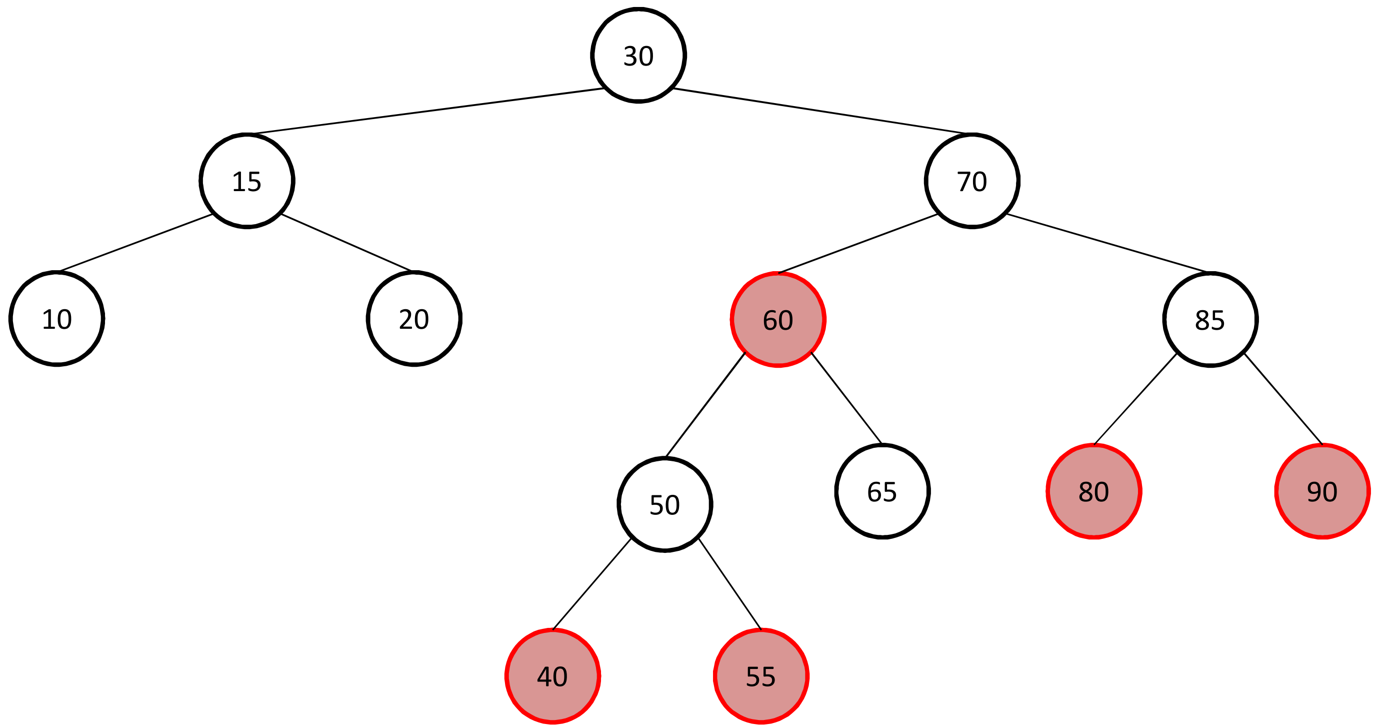
Red-Black Trees



Is this an AVL tree?
(Not anymore)

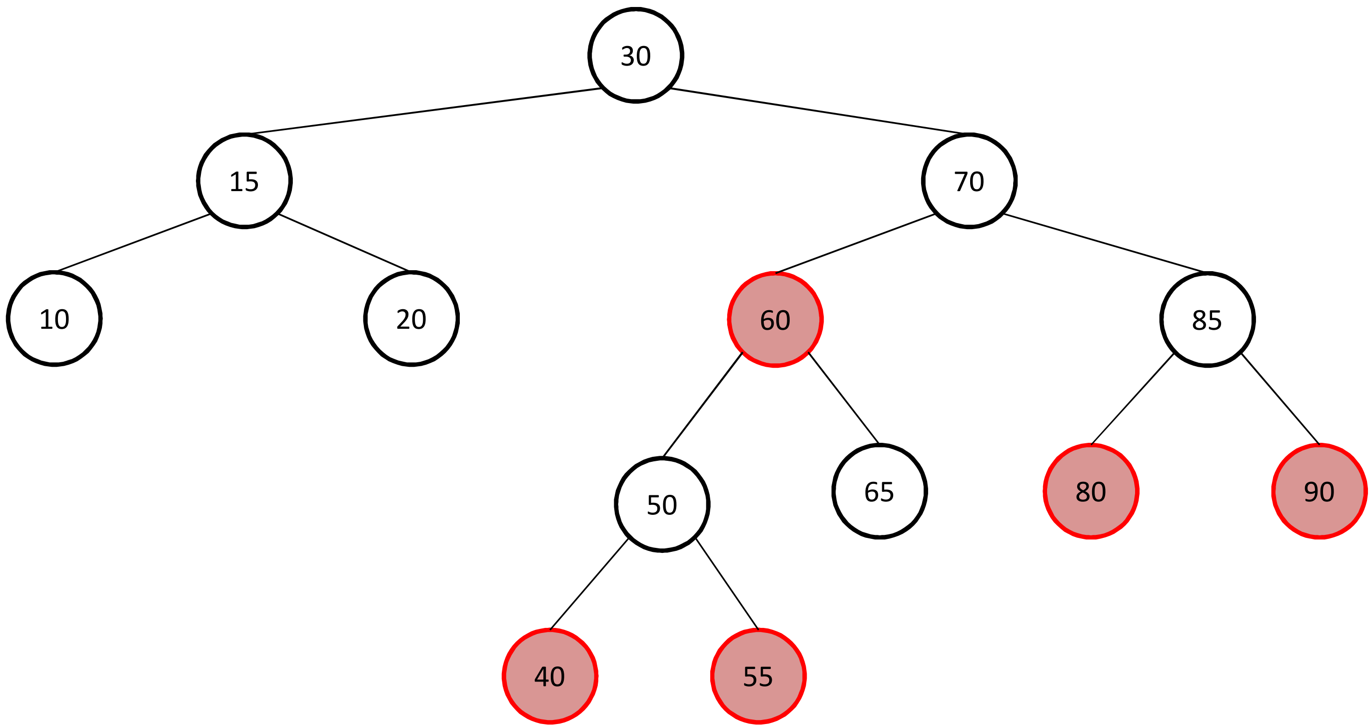
Are all red black trees AVL trees?
No

Why use Red-Black Trees



Fast to insert, slightly longer to find (but still guaranteed O(log(N)))

Why use Red-Black Trees



Faster to insert (than AVL): RBT insertion traverses the tree once instead of twice
Slower to find (that AVL): RBTs are generally slightly taller than AVL trees