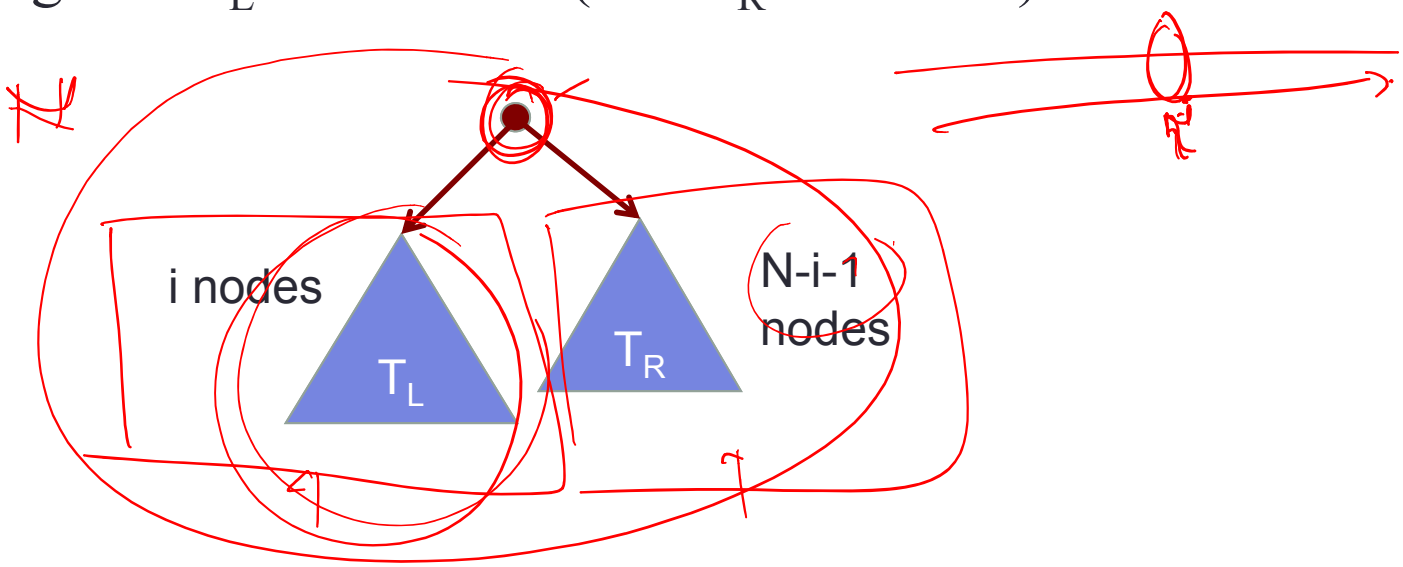
CSE 100: RUNTIME ANALYSIS: BST FIND

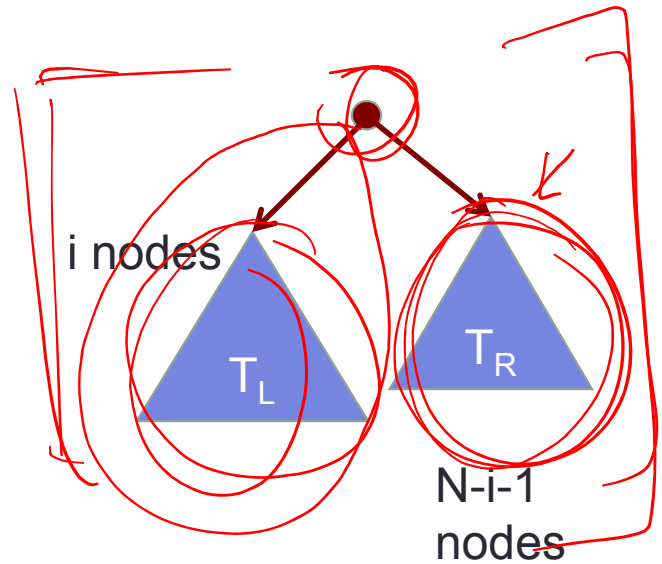
Towards a recurrence relation for average BST total depth

- Define D(N|i) as expected total depth of a BST with N nodes, assuming that T_L has *i* nodes (and T_R has *N-i-1*)



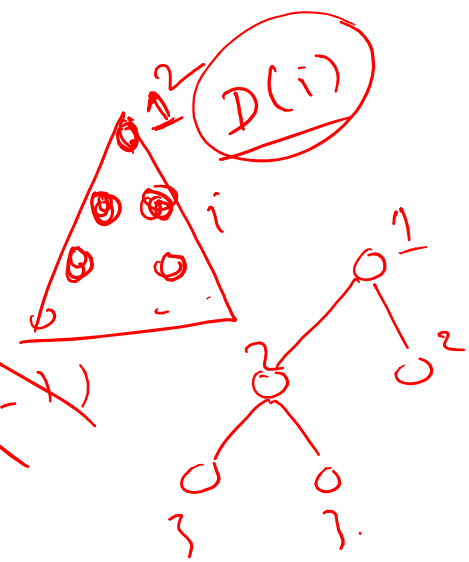
Towards a recurrence relation for average BST total depth

- We defined D(N|i) as the expected total depth of a BST with N nodes, assuming that T_L has *i* nodes (and T_R has *N-i-1*)



What is D(N|i) in terms of D(i) & D(N-i-1)?
Hint: all nodes in each subtree are 1 deeper in tree T

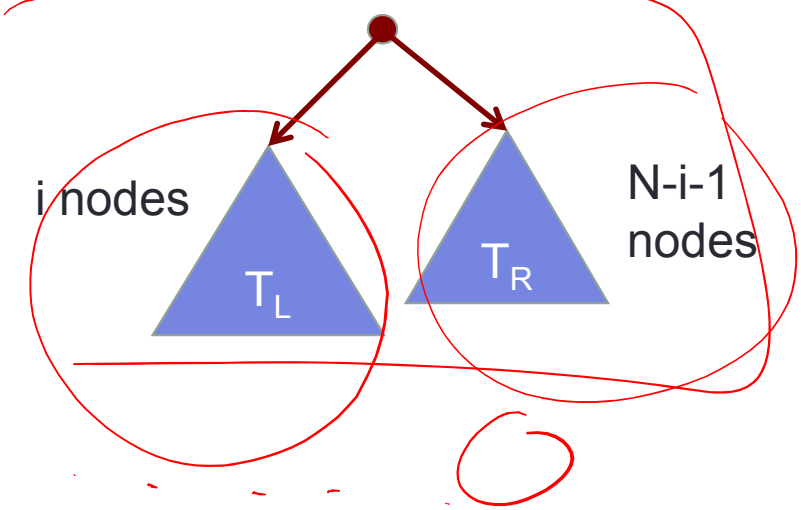
A. D(i) + D(N-i-1)
B. D(i) + D(N-i-1) + 1
C. D(i) + D(N-i-1) + N



$+ \boxed{D(i)} + \cancel{i}$
$+ D(N-i-1) + \cancel{(N-i-1)}$
$+ \cancel{1}$

Average case analysis of find in BST

- Given N nodes, how many such subsets of trees are possible as i is varied?



i

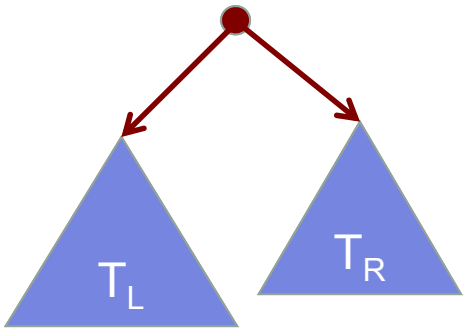
i=0 i=1 i=(N-1)

N

A. N
B. N!
C. $\log_2 N$
D. (N-1)!

Probability of subtree sizes

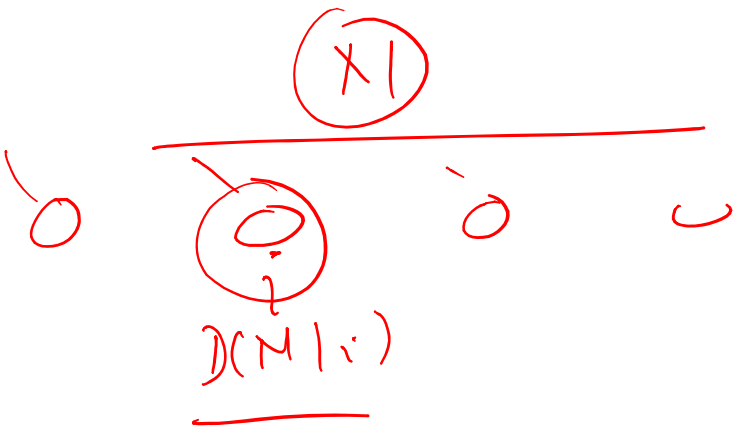
- Let $P_N(i)$ = the probability that T_L has *i* nodes
- It follows that D(N) is given by the following equation



i nodes

N-i-1 nodes

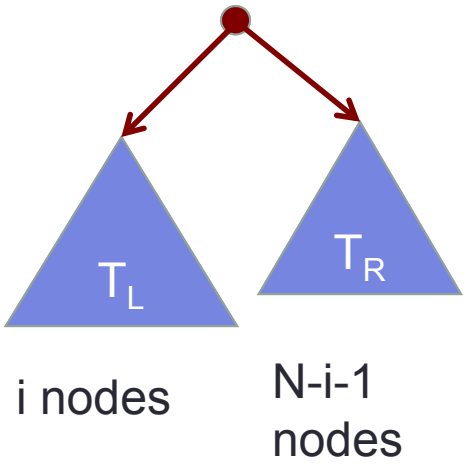
$$D(N) = \sum_{i=0}^{N-1} P_N(i) D(N \mid i)$$



Probability of subtree sizes

- Let $P_N(i)$ = the probability that T_L has *i* nodes
- It follows that D(N) is given by the following equation

$$D(N) = \sum_{i=0}^{N-1} P_N(i) D(N \mid i)$$

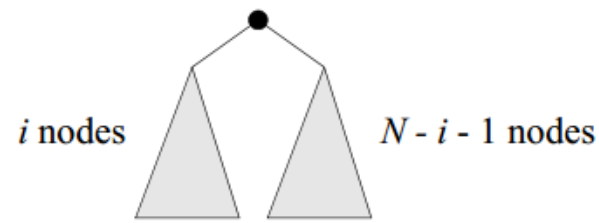


What is the value of $P_N(i)$?
Hint: use assumption #2, any of the N keys are equally likely to be inserted first

A. *1/N*
B. It depends on i and N

Average total depth of a BST with N nodes

$$D(N) = \sum_{i=0}^{N-1} P_N(i) D(N \mid i)$$



$$D(N) = \sum_{i=0}^{N-1} \frac{1}{N}[D(i) + D(N-i-1) + N]$$

$j = (N - i - 1)$

$$= \boxed{\frac{1}{N}\sum_{i=0}^{N-1} D(i)} + \boxed{\frac{1}{N}\sum_{i=0}^{N-1} D(N-i-1)} + N$$

True or false: The term in the blue box is equal to the term in the red box

A. True
B. False

- Note that those two summations just add the same terms in different order; so

$$N \times \left[D(N) = \frac{2}{N} \sum_{i=0}^{N-1} D(i) + N \right]$$

- ... and multiplying by N,

$$\left[ND(N) = 2 \sum_{i=0}^{N-1} D(i) + N^2 \right]$$

- Now substituting N-1 for N,

$$(N-1)D(N-1) = 2 \sum_{i=0}^{N-2} D(i) + (N-1)^2$$

$a^2 - b^2$

$(a+b)(a-b)$

- Subtracting that equation from the one before it gives

$$ND(N) - (N-1)D(N-1) = 2D(N-1) + N^2 - (N-1)^2$$

- ... and collecting terms finally gives this recurrence relation on $D(N)$:

$$ND(N) = (N+1)D(N-1) + 2N - 1$$

$$N*D(N) = (N+1)*D(N-1) + 2N - 1$$

How does this help us, again?

A. We can solve it to yield a formula for D(N) that does not involve N!

B. We can use it to compute D(N) directly

C. I have no idea, I'm totally lost

$$ND(N) = (N+1)D(N-1) + 2N - 1$$

Through unwinding and some not-so-complicated algebra (which you can find in your reading, a.k.a. Paul's slides) we arrive at:

$$D(N) = 2(N+1)\sum_{i=1}^{N} \frac{1}{i} - 3N$$

No N! to be seen! Yay!

And with a little more algebra, we can even show an approximation:

$$\frac{D(N)}{N} = D_{avg}(N) \approx 1.386 \log_2 N$$

Conclusion: The average time to find an element in a BST with no restrictions on shape is Θ(log N).

The importance of being balanced

- A binary search tree has average-case time cost for Find = Θ (log N):

What does this analysis tell us:

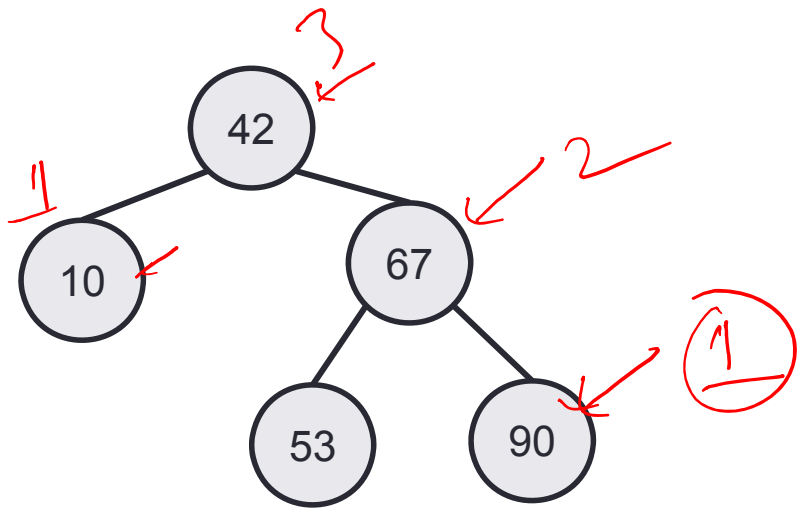
- On an average things are not so bad provided assumptions 1 and 2 hold
- But the probabilistic assumptions we made often don't hold in practice
  - Assumption #1 may not hold: we may search some keys many more times than others
  - Assumption #2 may not hold: approximately sorted input is actually quite likely, leading to unbalanced trees with worst-case cost closer to O(N) when N is large
- We would like our search trees to be balanced

The importance of being balanced

- We would like our search trees to be balanced
- Two kinds of approaches
  - Deterministic methods guarantee balance, but operations are somewhat complicated to implement (AVL trees, red black trees)
  - Randomized methods (treaps, skip lists) (insight from our result) – deliberate randomness in constructing the tree helps!!
    - Operations are simpler to implement
    - Balance not absolutely guaranteed, but achieved with high probability
  - We will return to this topic later in the course

Local computation of height of a node

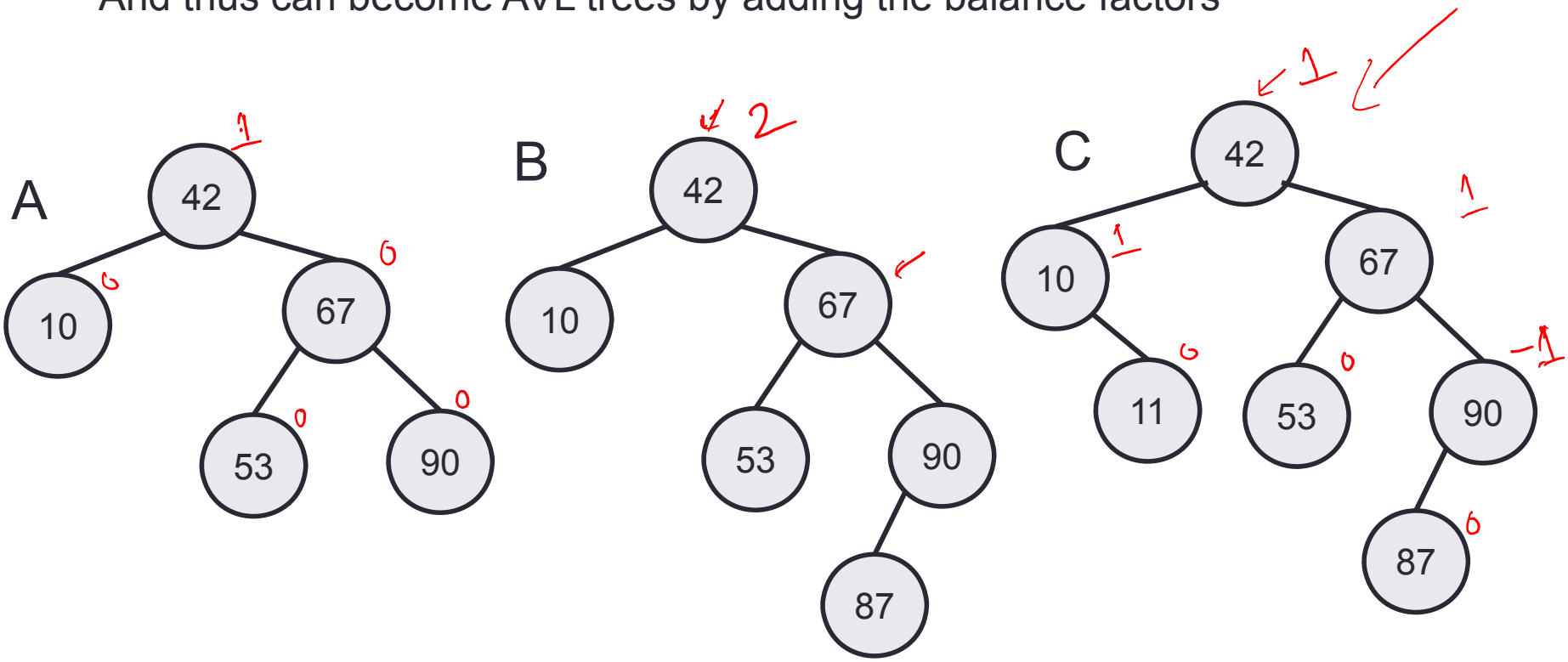
Height of a node= max{ Height of left child, Height of right child} +1



- Goal of AVL trees: For each node, maintain the difference between height of left and right children to within +/-1
- Each node maintains then a balance factor:
- Balance factor= height of the right child – height of left child

Which of the following is/are balanced trees?

And thus can become AVL trees by adding the balance factors



D. A&C
E. A&B&C

Annotate the trees with balance factors
(for those that are balanced)

An AVL Tree is *worst case* O(logN) to find an element! (We might prove this later)

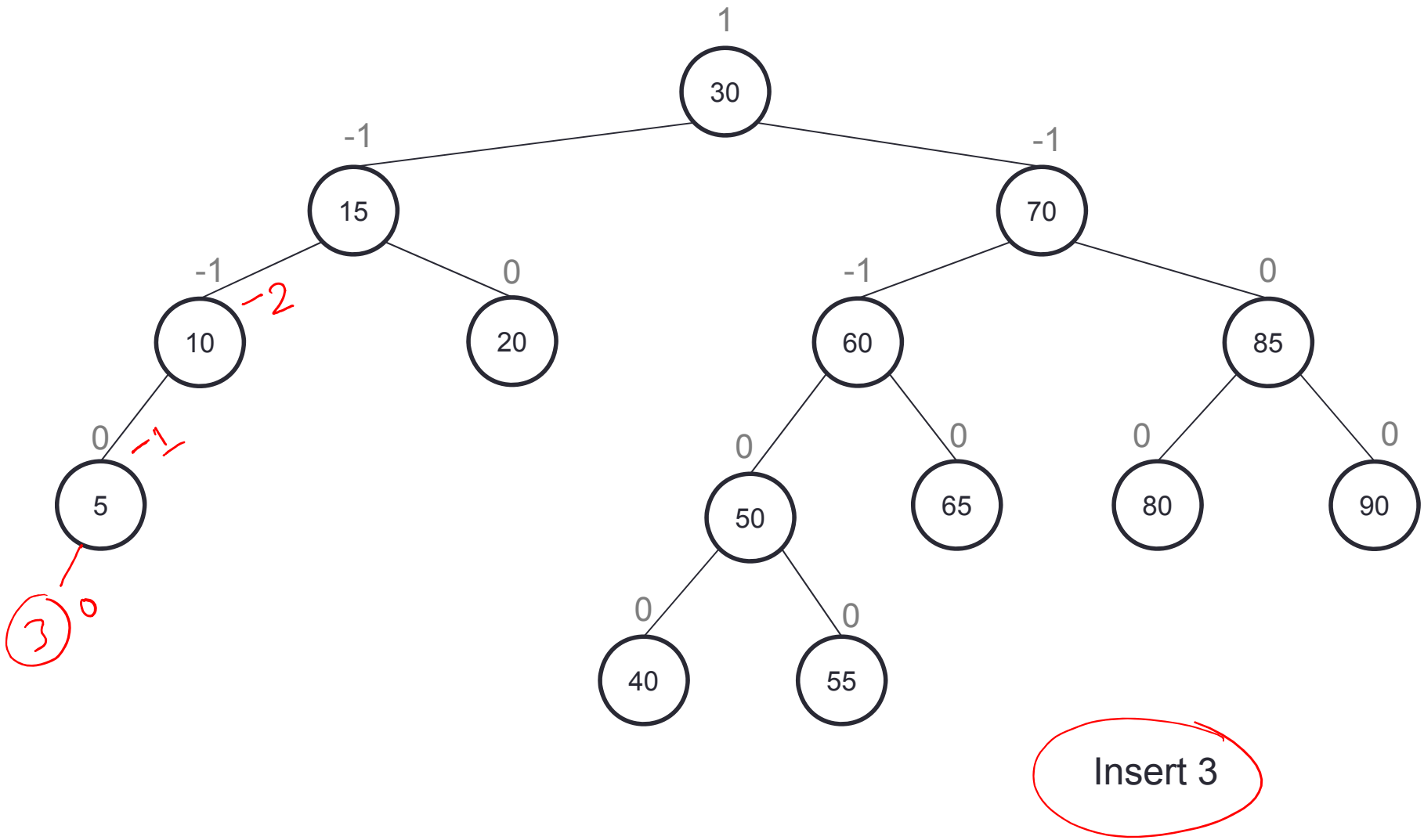
How would you prove this?

A. Come up with a formula that shows that the height of the tallest AVL tree with N nodes is never bigger than c*logN + k, for some c and k (assuming large N).

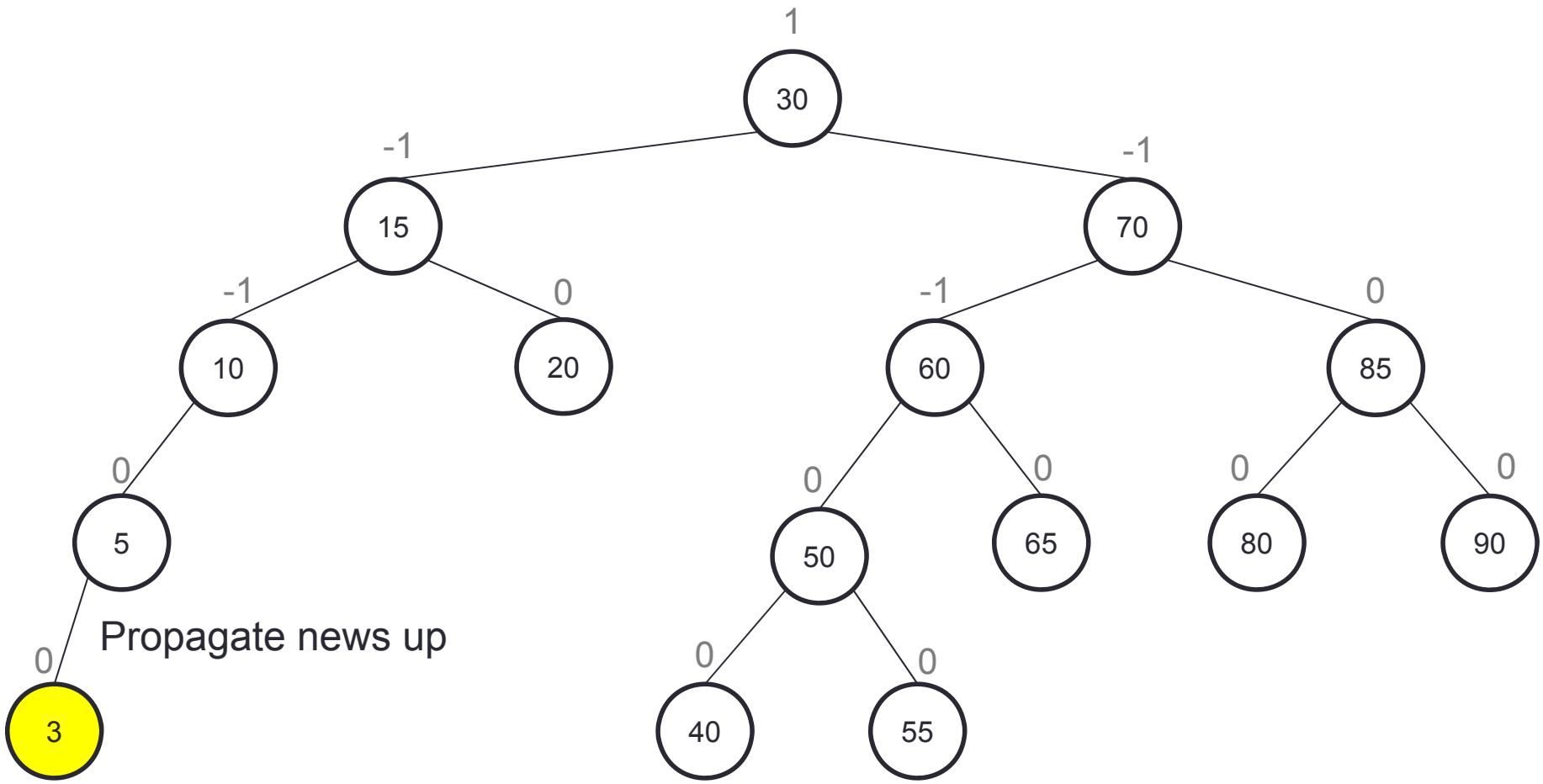
The key to this proof is showing that the height stays “small”, no matter how legally “unbalanced” the tree is.

But how does the tree stay balanced as new nodes are inserted??

Inserting and rebalancing

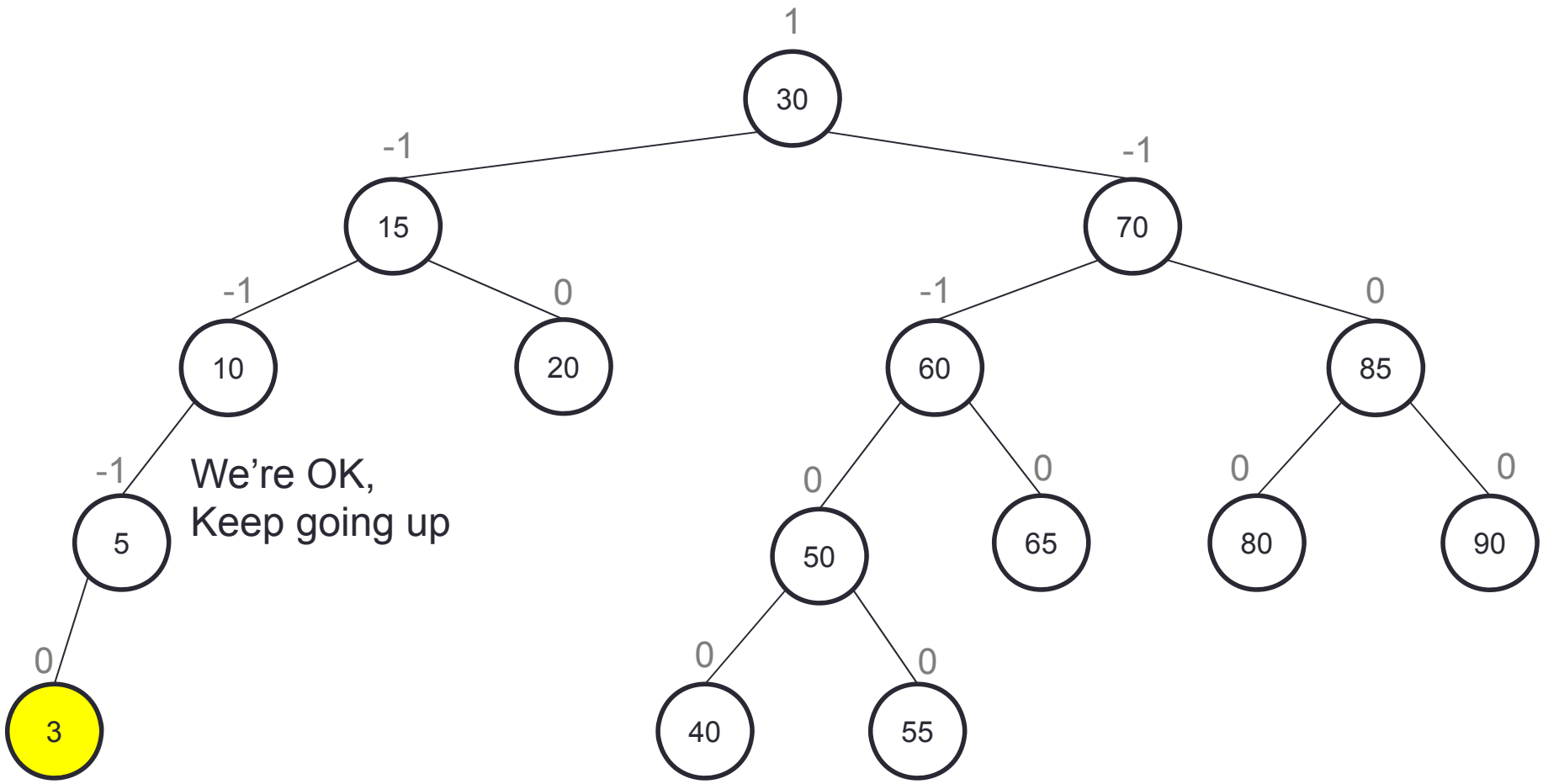


Inserting and rebalancing



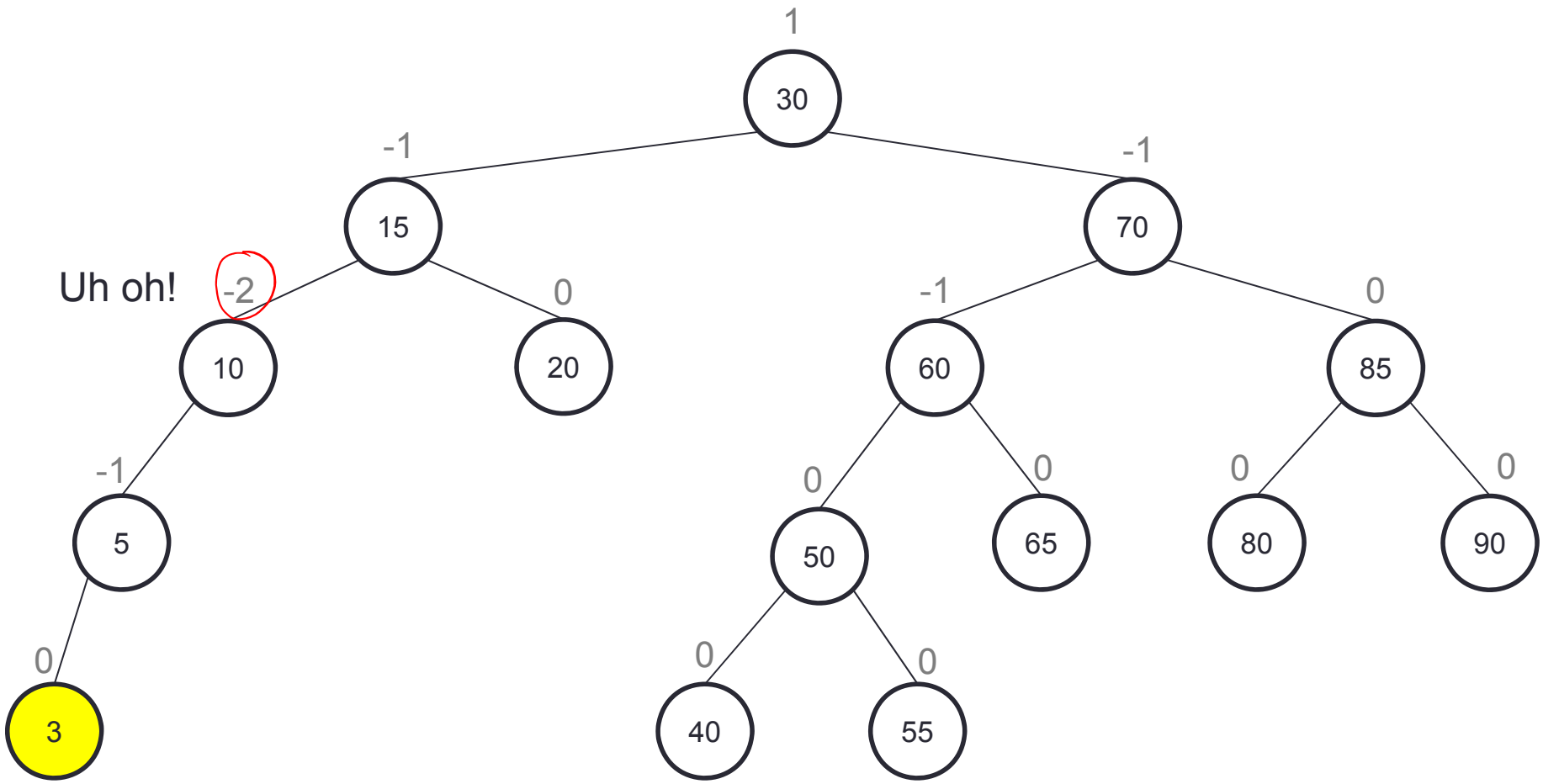
Insert 3

Inserting and rebalancing



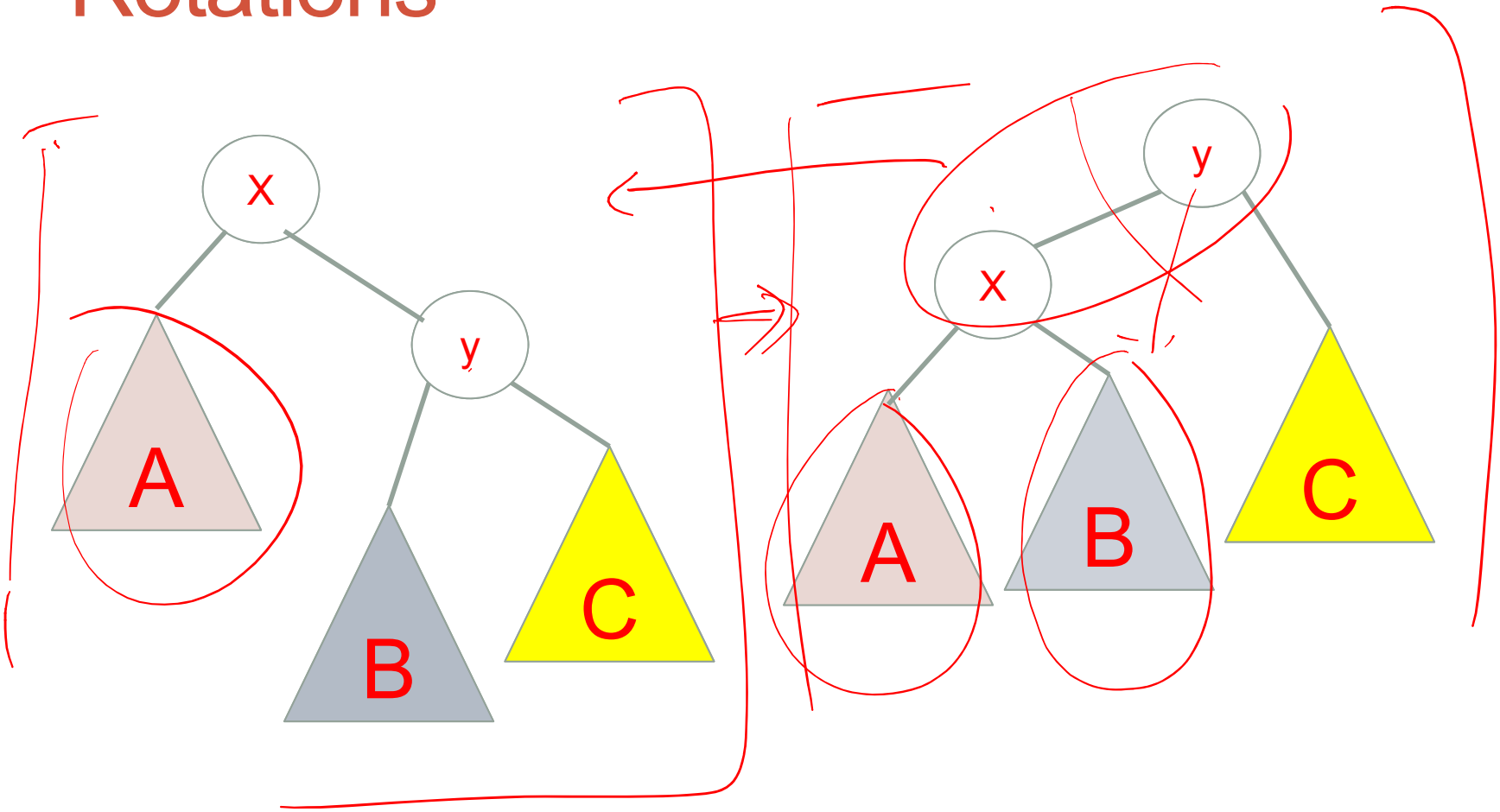
Insert 3

Inserting and rebalancing



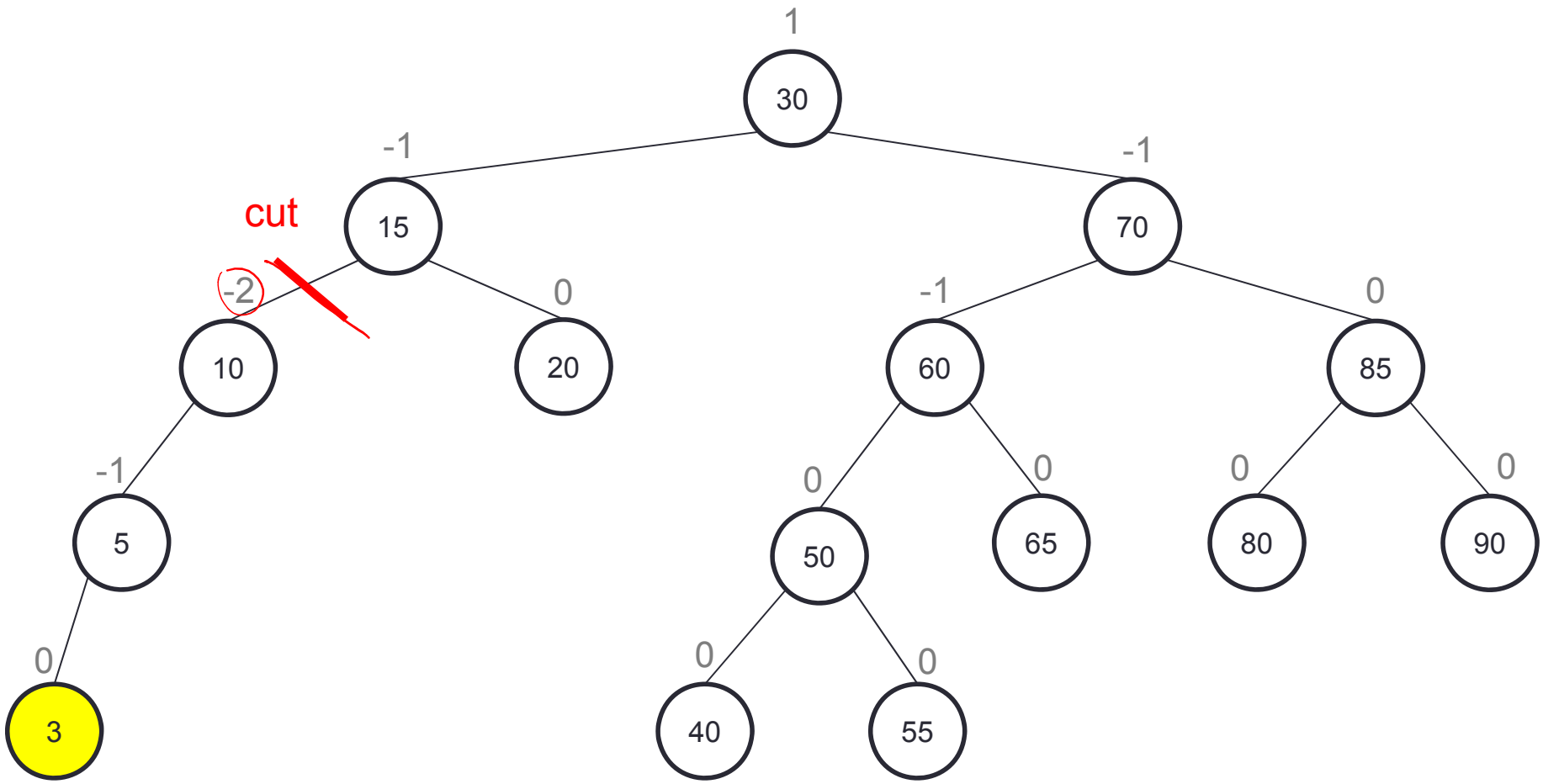
Insert 3

Rotations



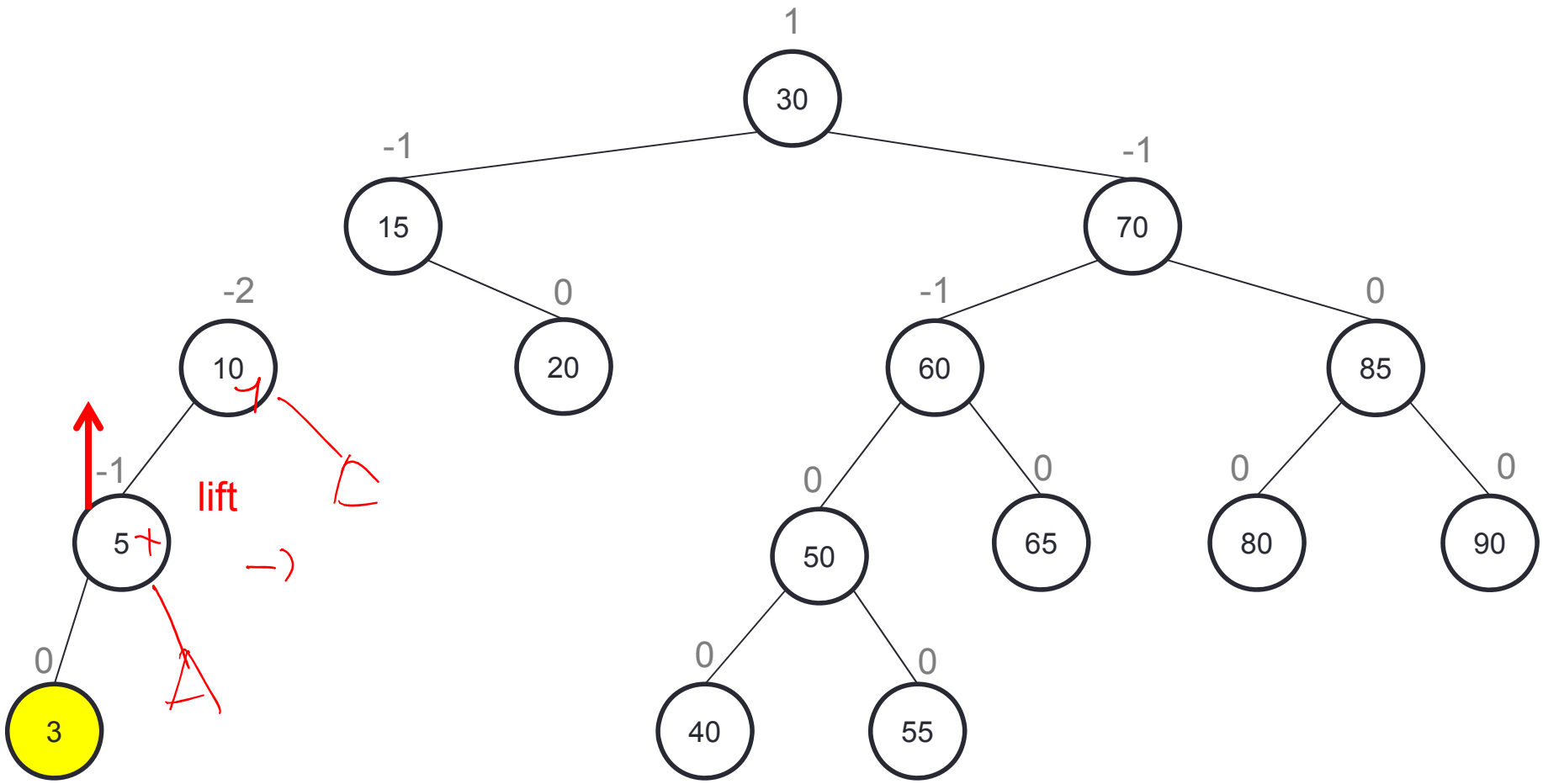
Inserting and rebalancing

Rotate! Left or right?



Insert 3

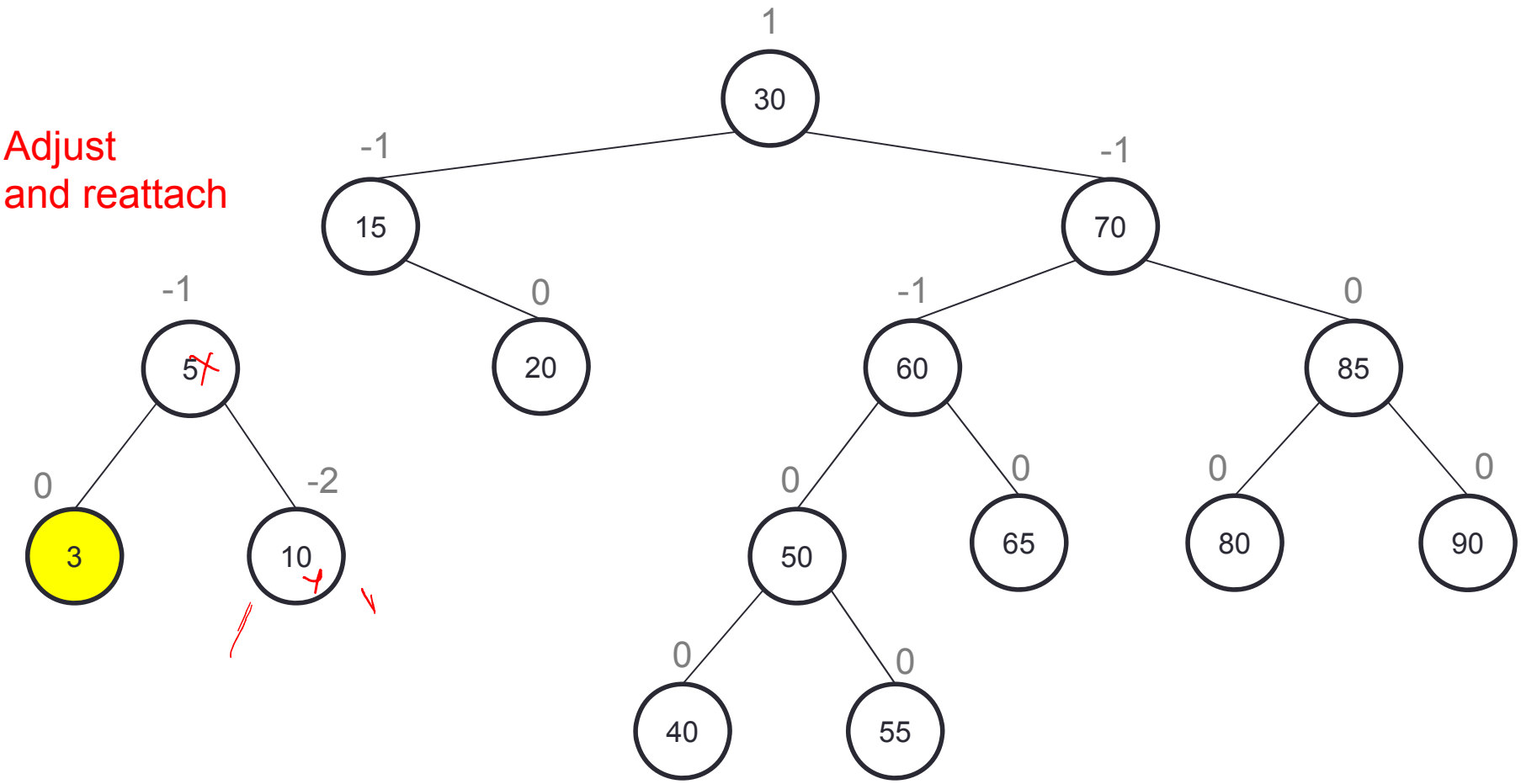
Inserting and rebalancing



Insert 3

Inserting and rebalancing

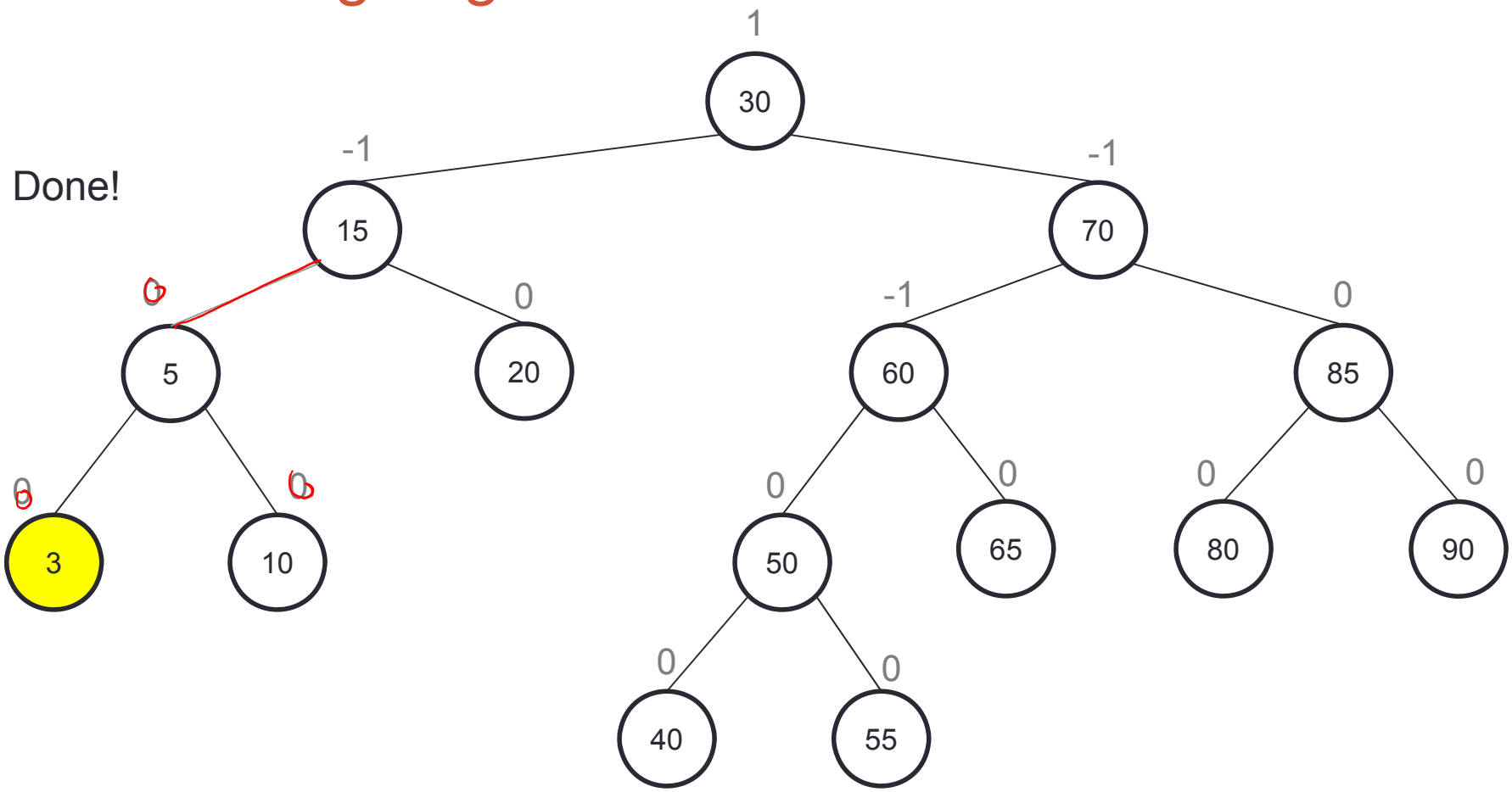
Adjust
and reattach



Insert 3

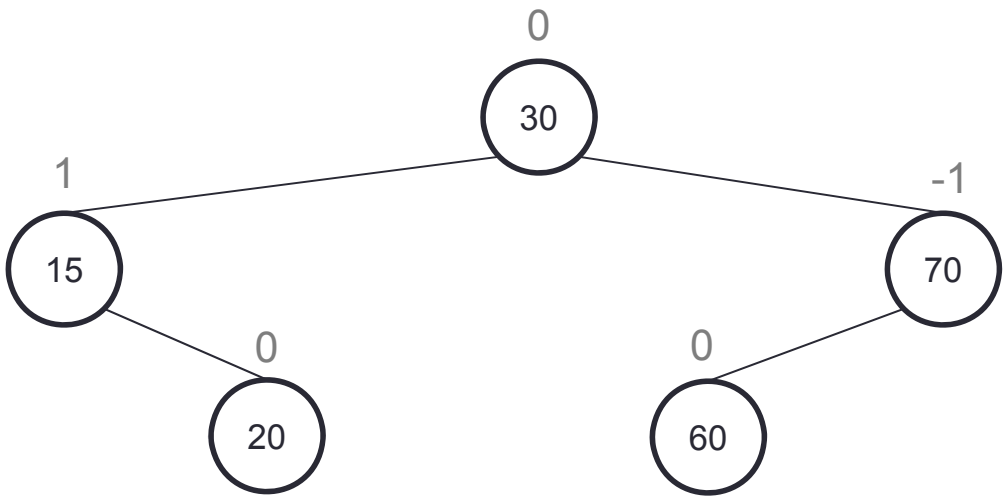
We just did a single rotation of 5 around 10
A.k.a. a single right rotation

Done!



Insert 3

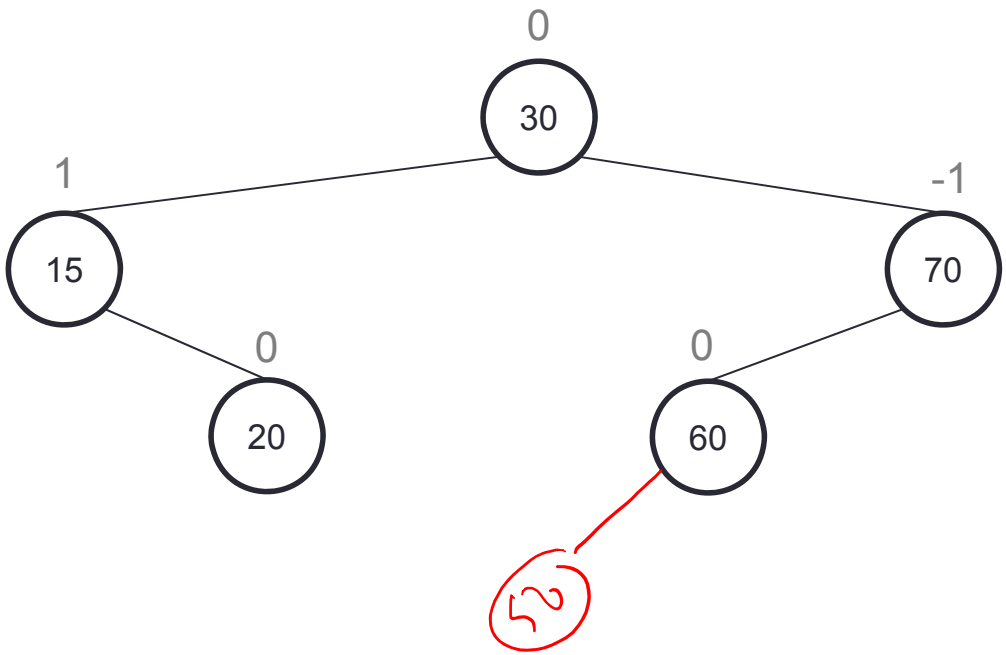
Single rotation practice



What could you insert into this AVL tree that would result in a single right rotation?

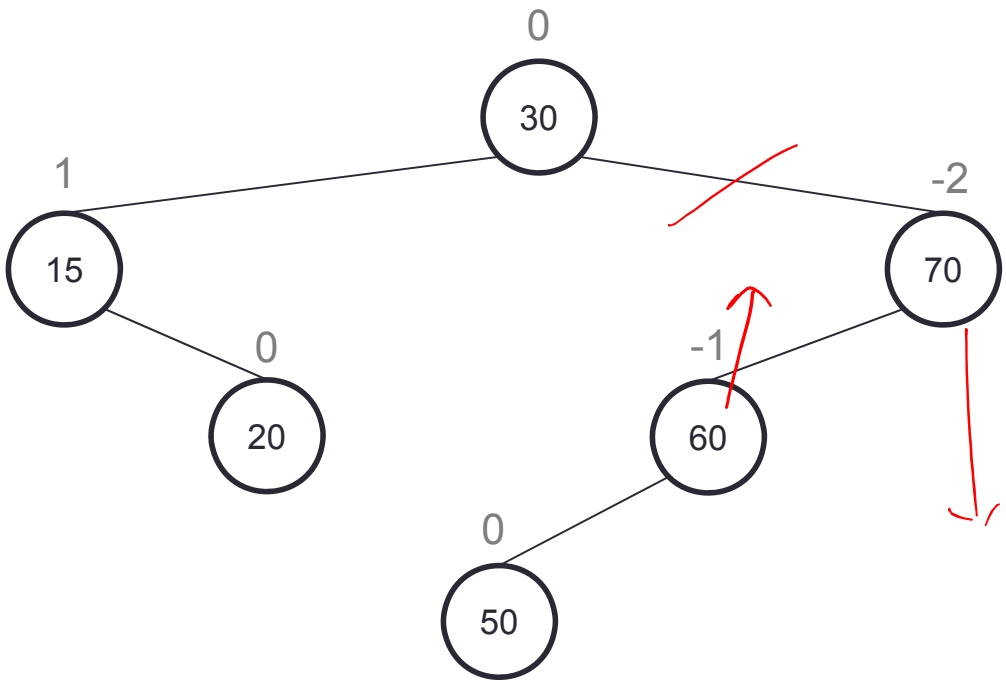
A. 71
B. 10
C. 50
D. 66

Single rotation practice



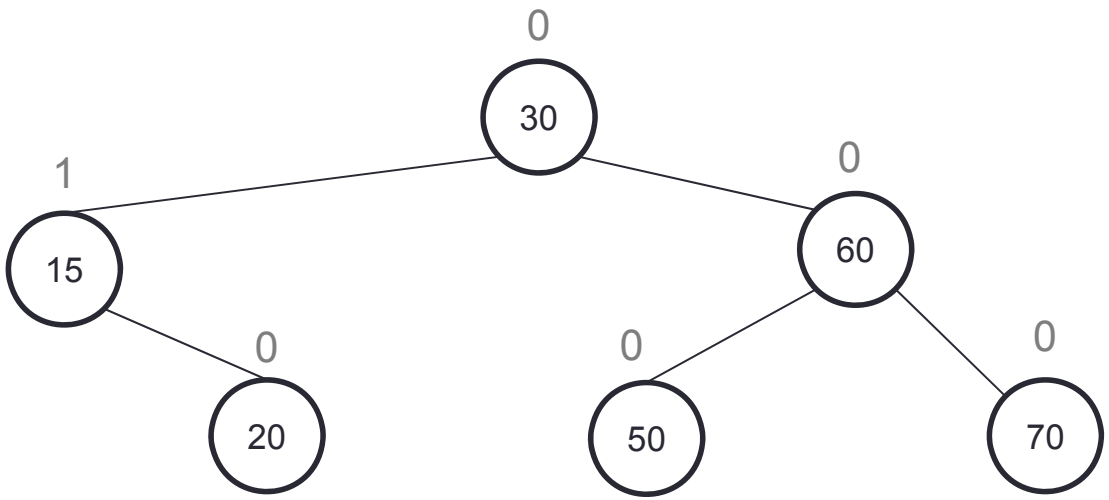
Insert 50. Draw the resulting AVL tree. (Don't peek)

Single rotation practice



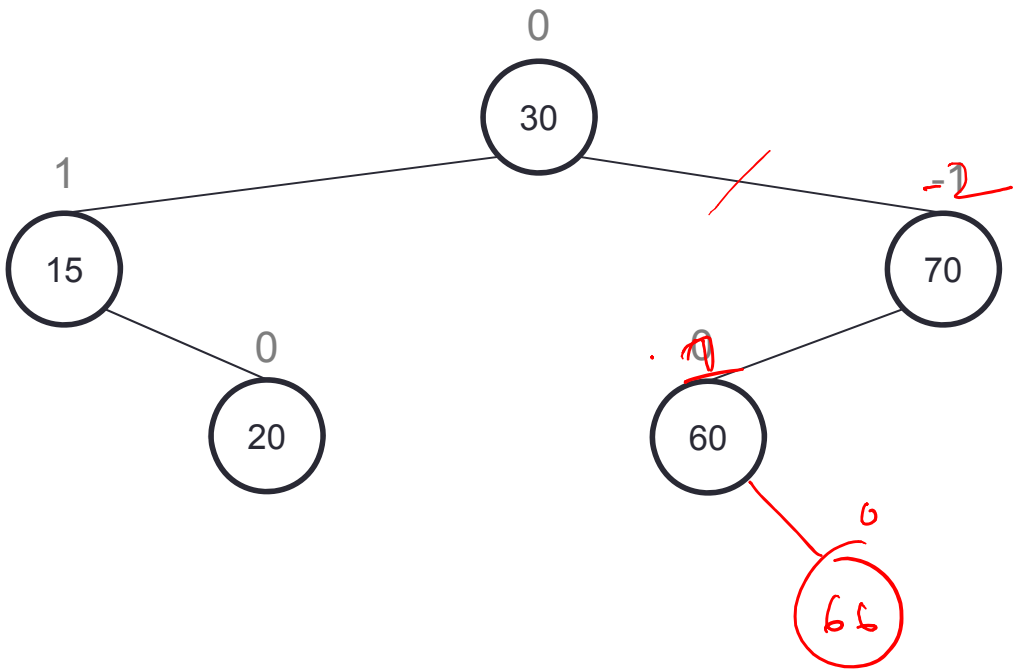
After insertion

Single rotation practice



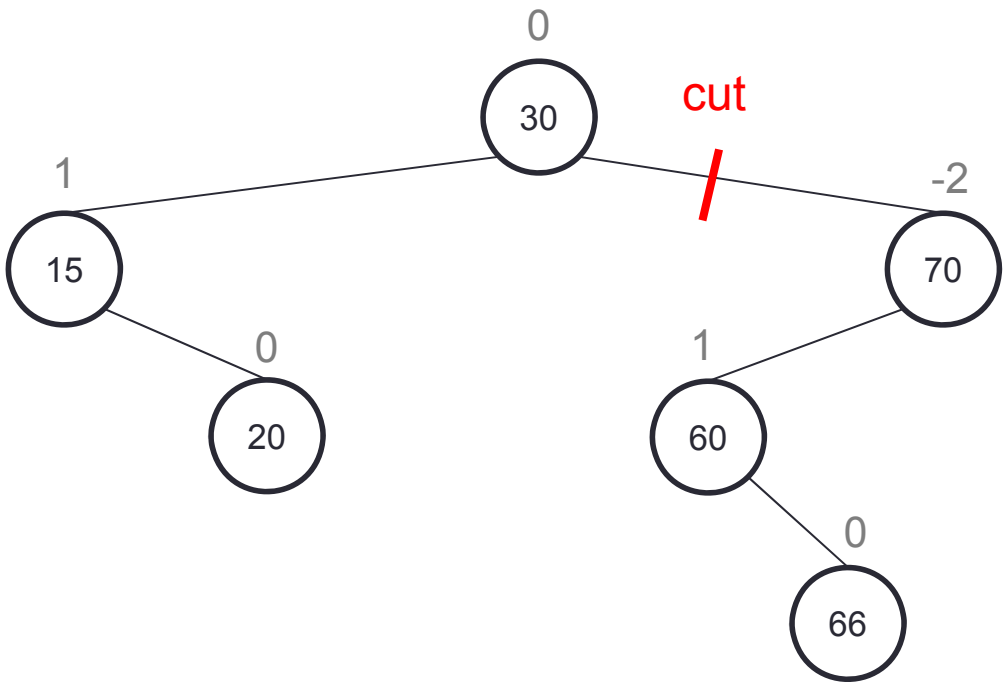
After rotation

Single rotation is not enough



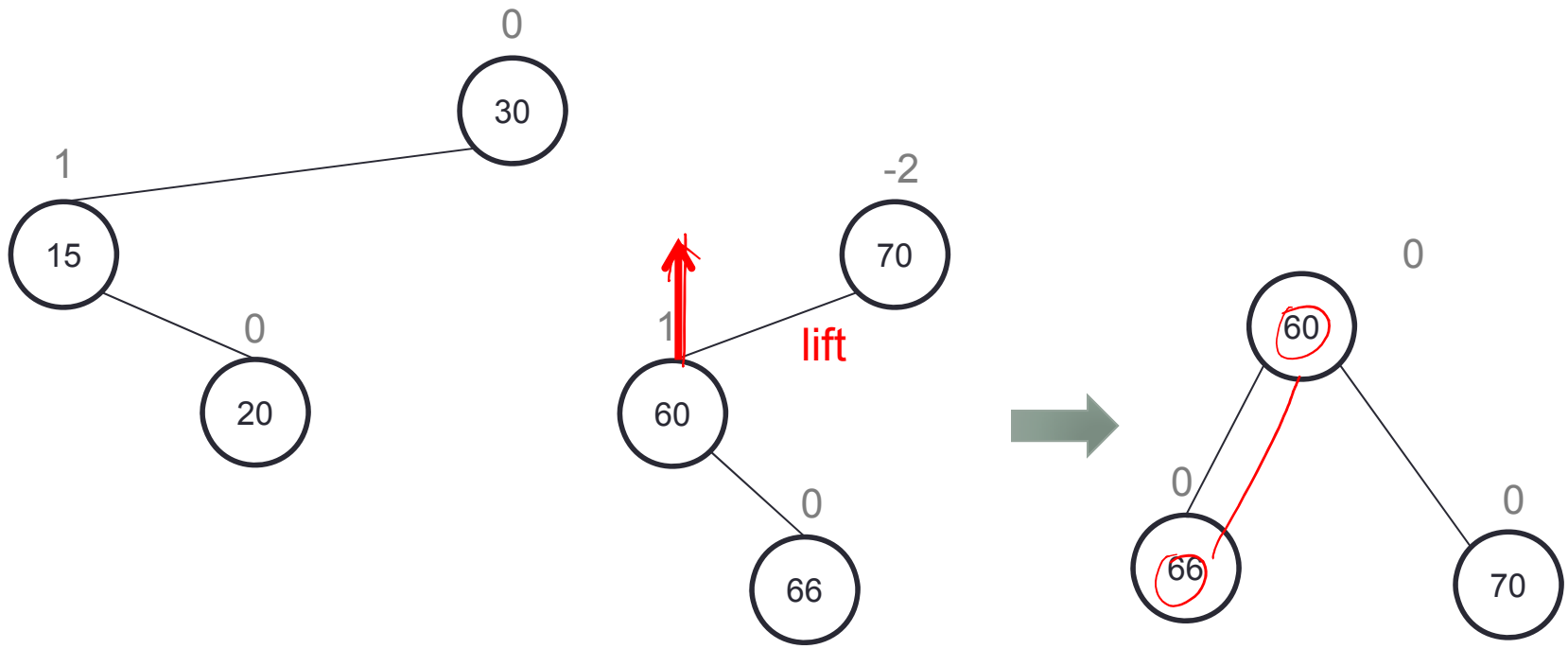
What happens if we insert 66?

Single rotation is not enough

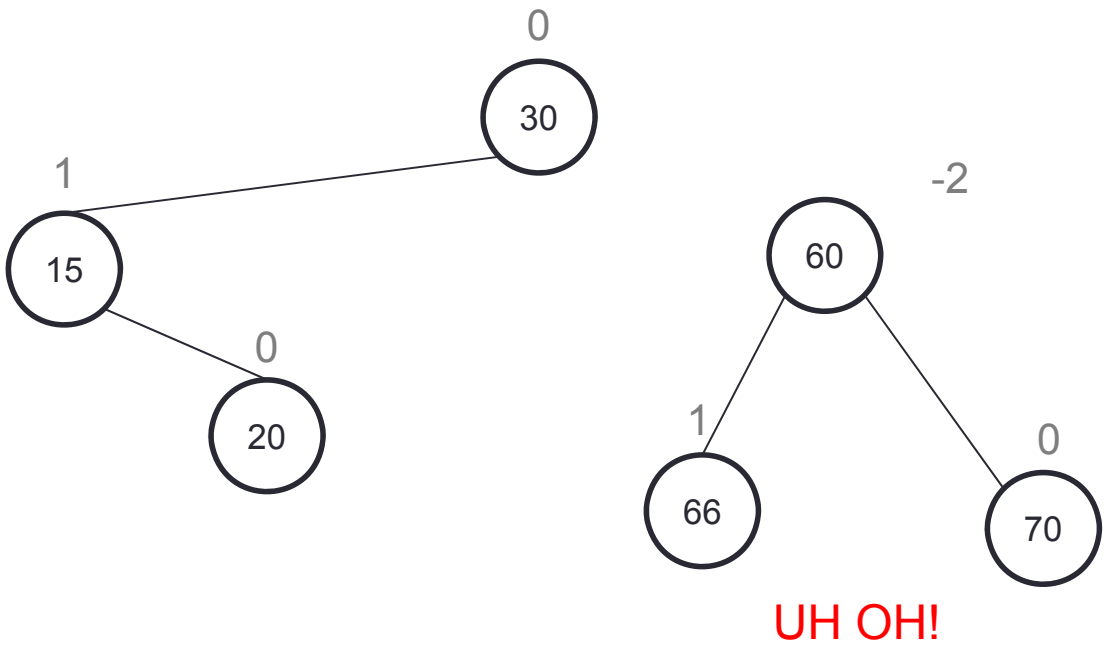


Is this a valid rotation?

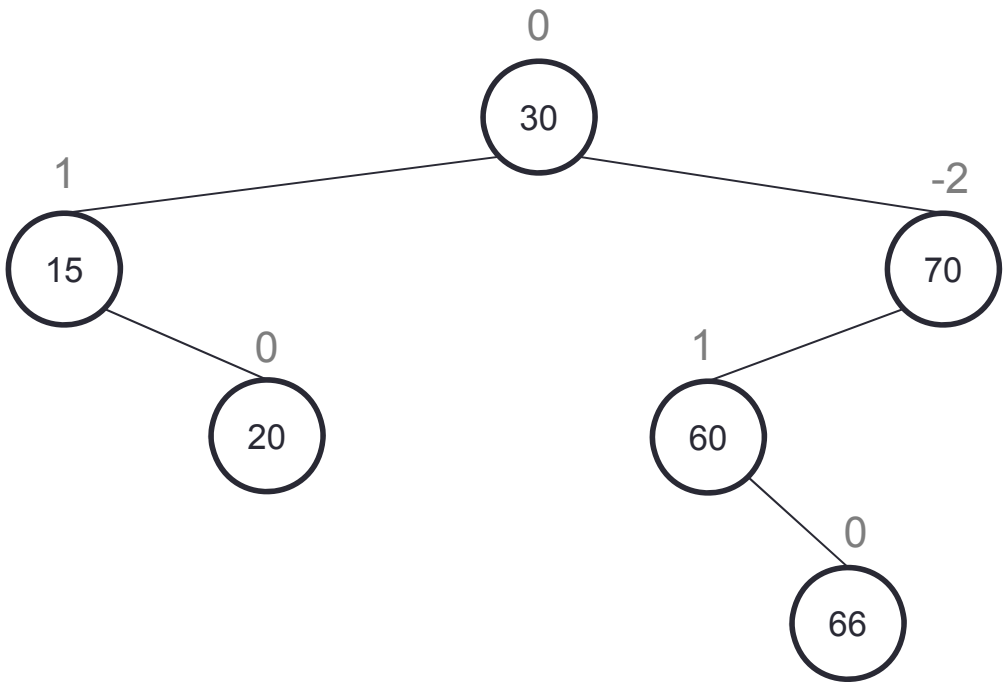
A. Yes
B. No



Invalid rotation!

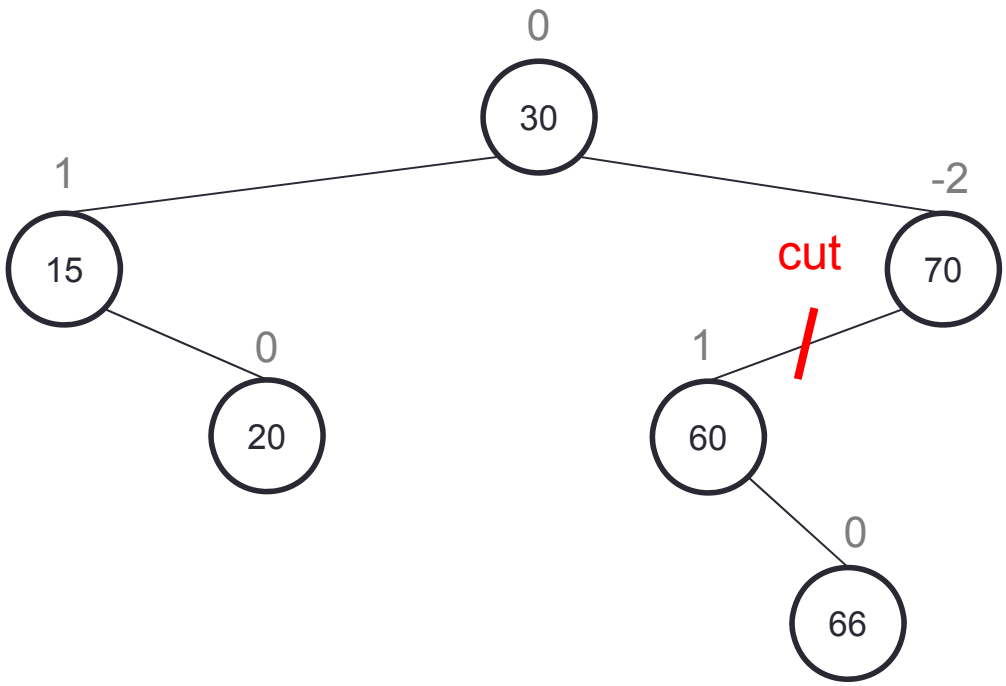


Double rotation to the rescue



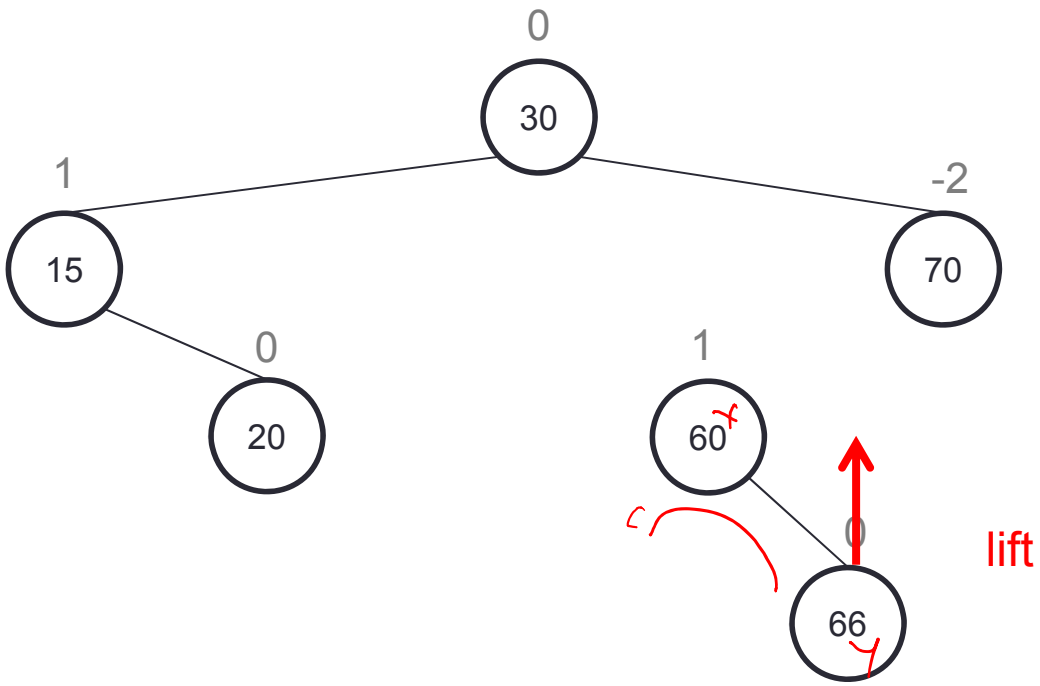
Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

Double rotation to the rescue



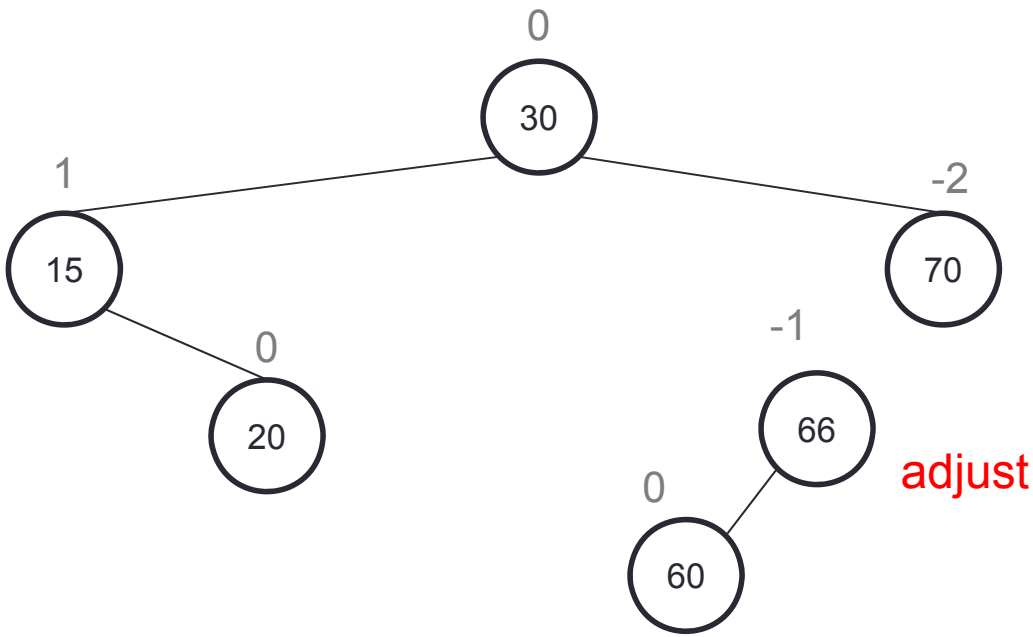
Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

Double rotation to the rescue



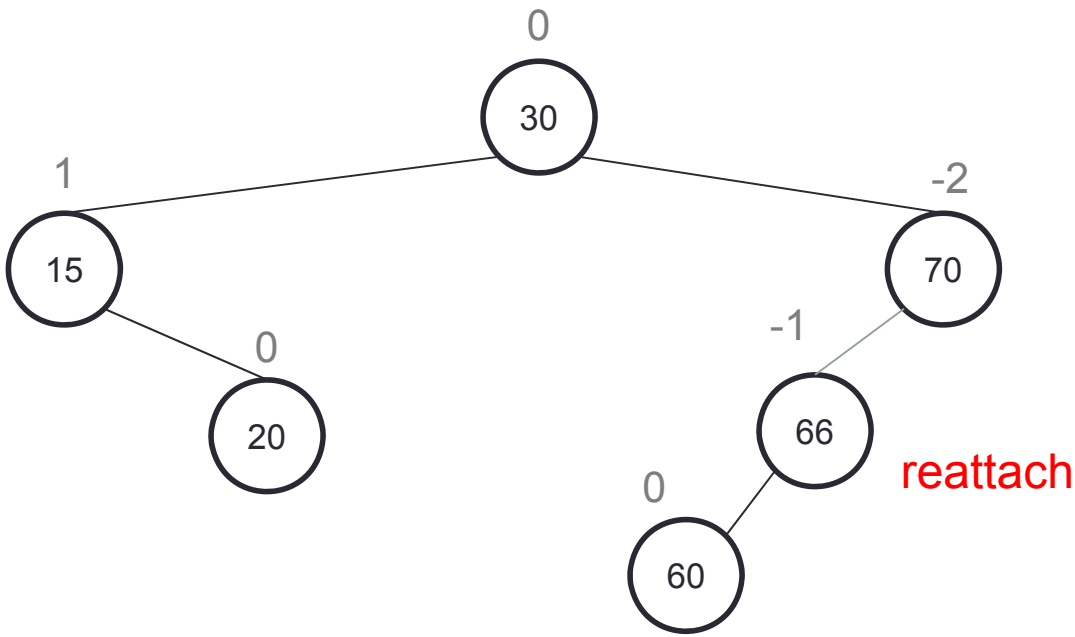
Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

Double rotation to the rescue



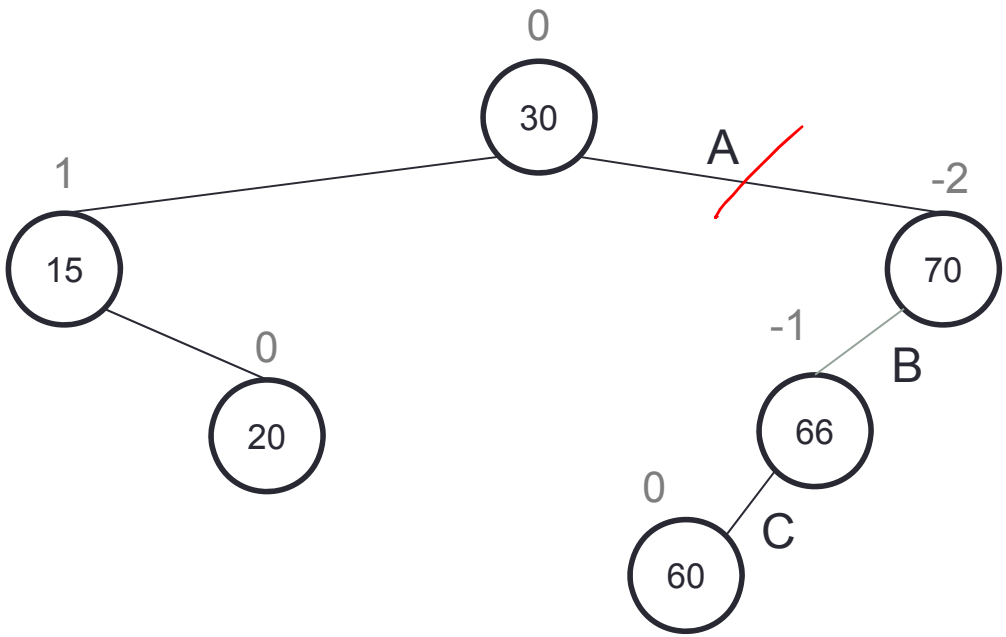
Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

Double rotation to the rescue



Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

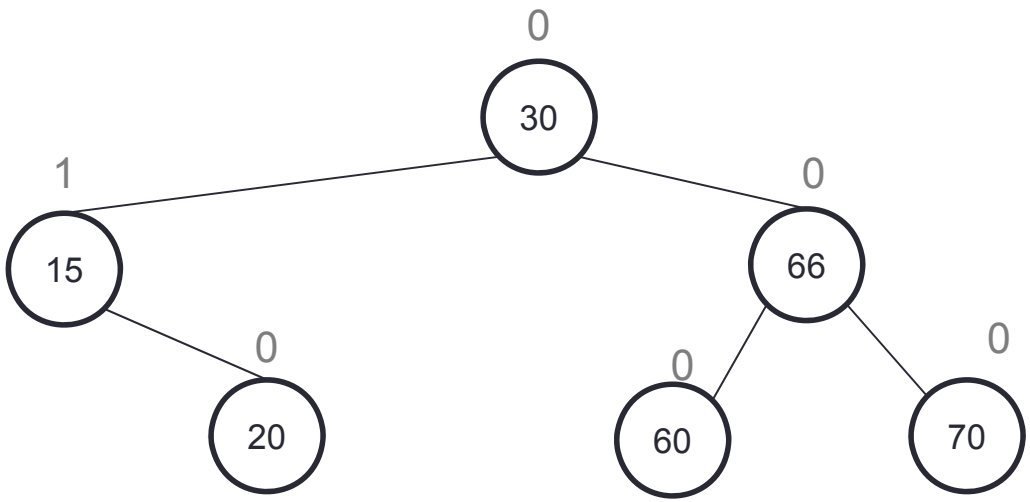
Double rotation to the rescue



Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

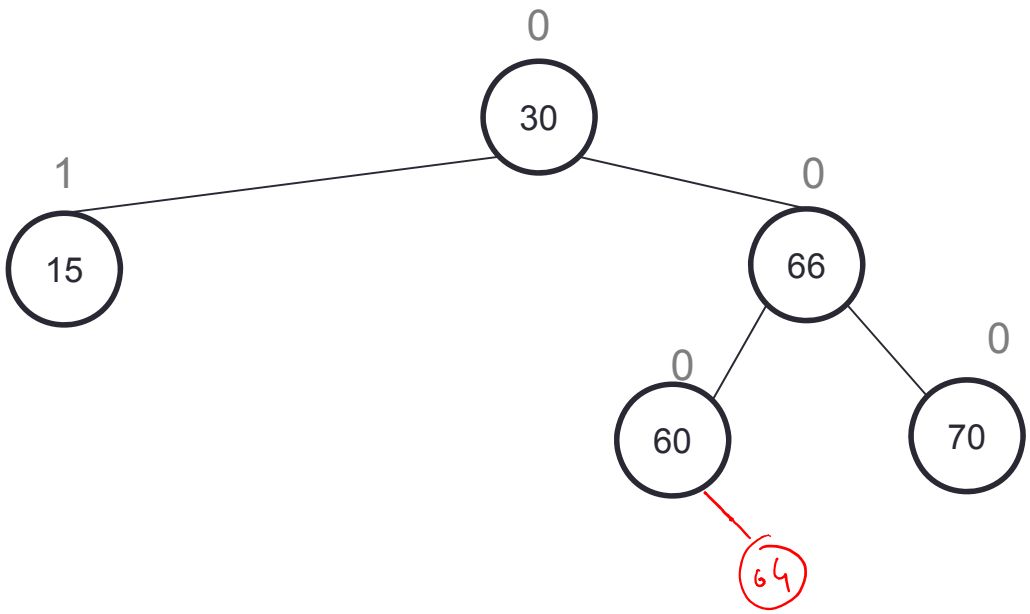
Where in the tree above should I cut to start this rotation?

Double rotation to the rescue



Single rotations only work when involved nodes are “in a line”
So we will first rotate left around 60, then we can rotate right around 70.

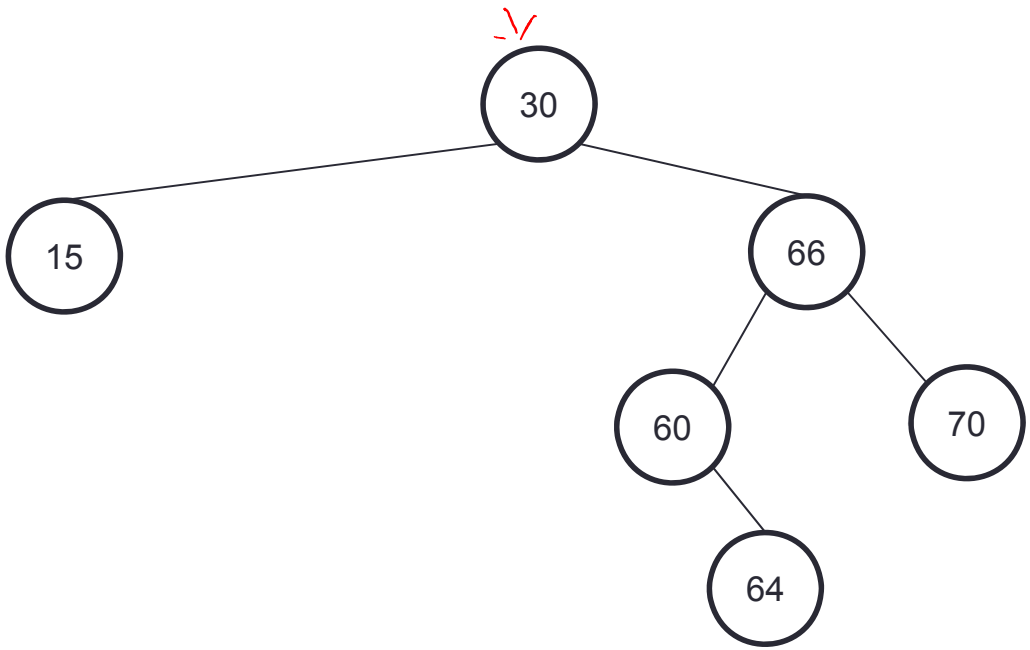
It's sometimes even more complicated



Insert 64… do we need a double or a single rotation?

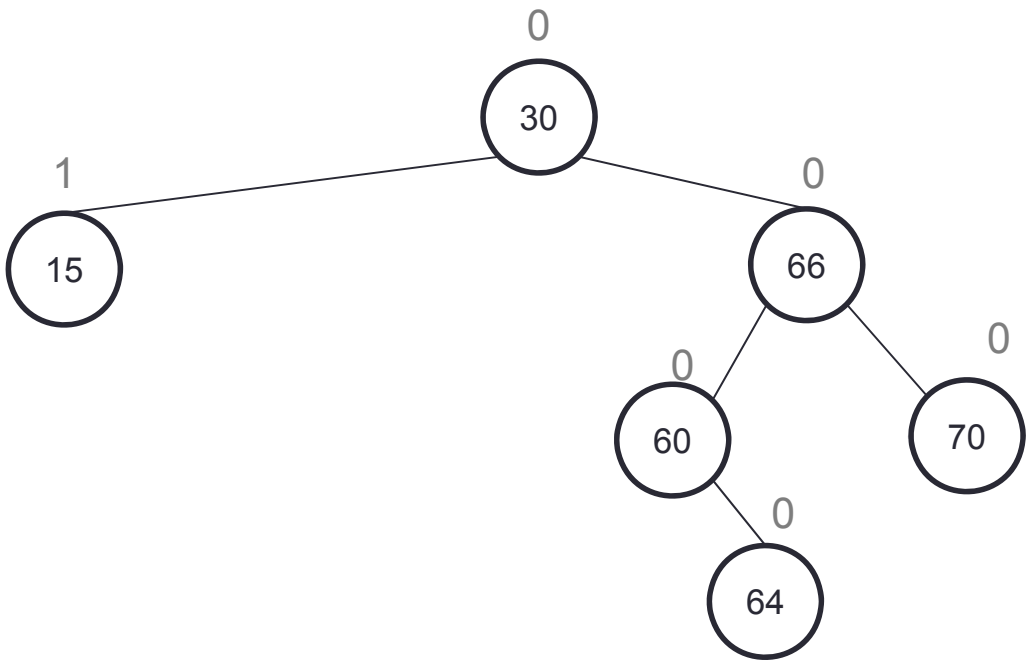
A. Double
B. Single
C. No rotation needed

Where is the tree out of balance?

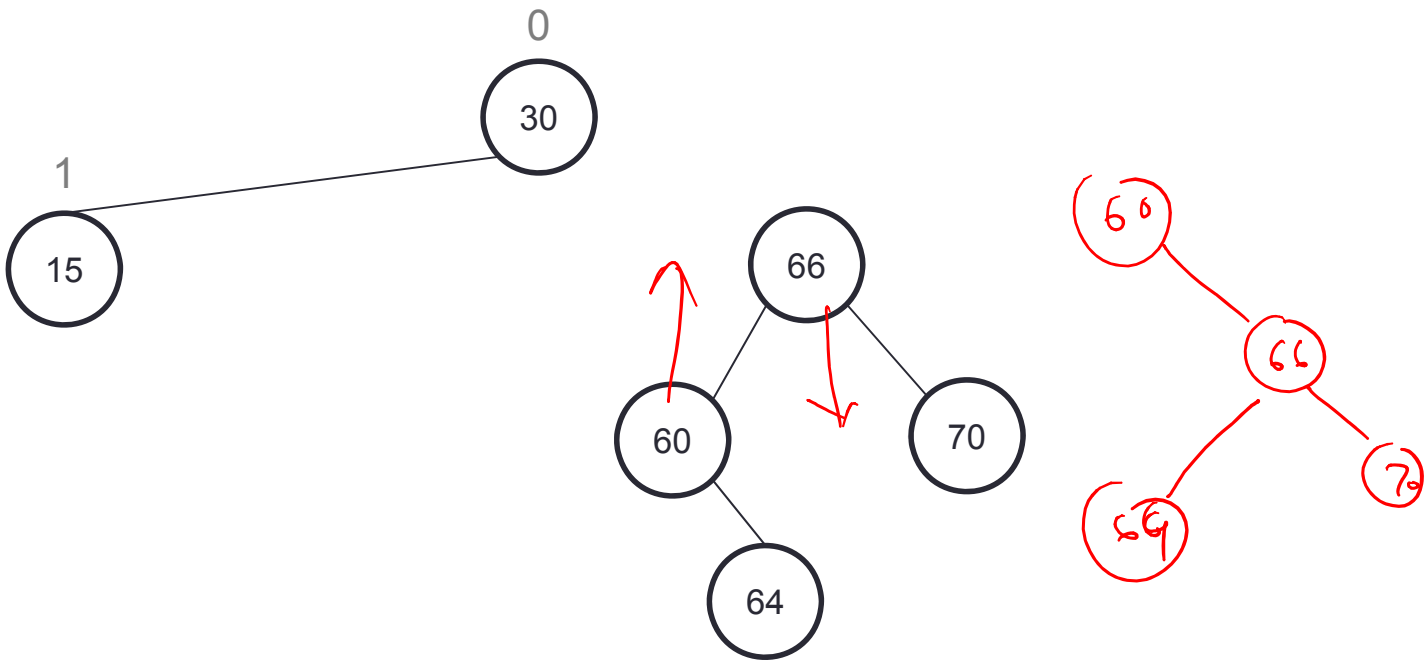


A. Node 30
B. Node 66
C. Node 60
D. Node 64

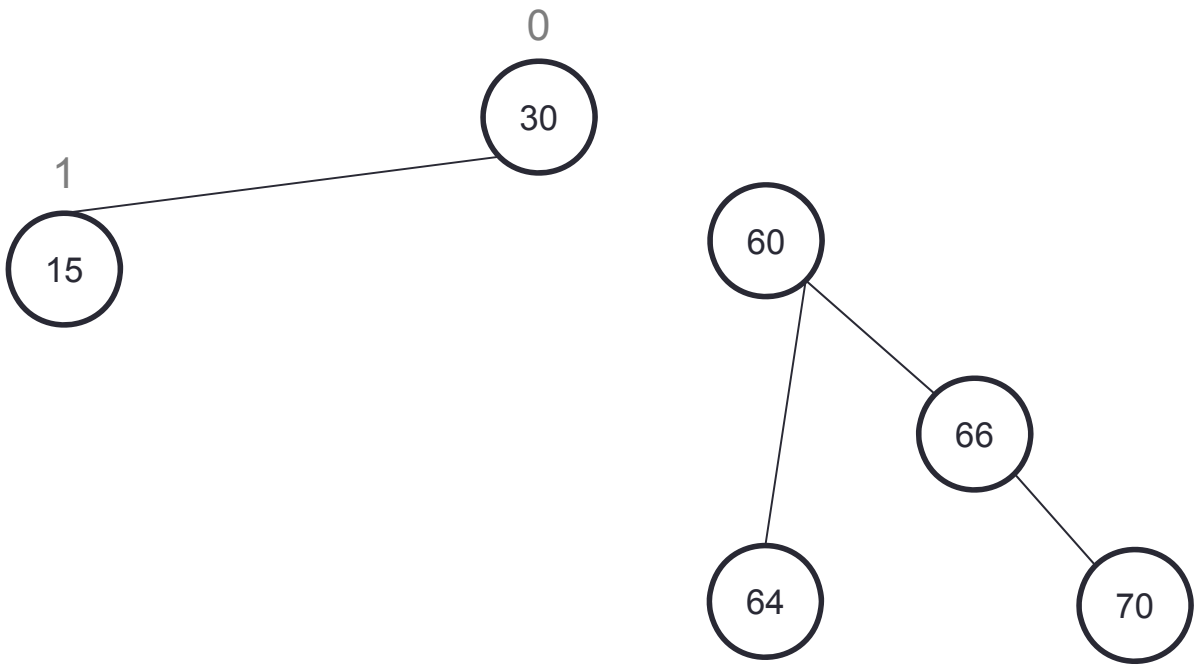
Rotate right around 66 to make a straight line



Rotate right around 66 to make a straight line

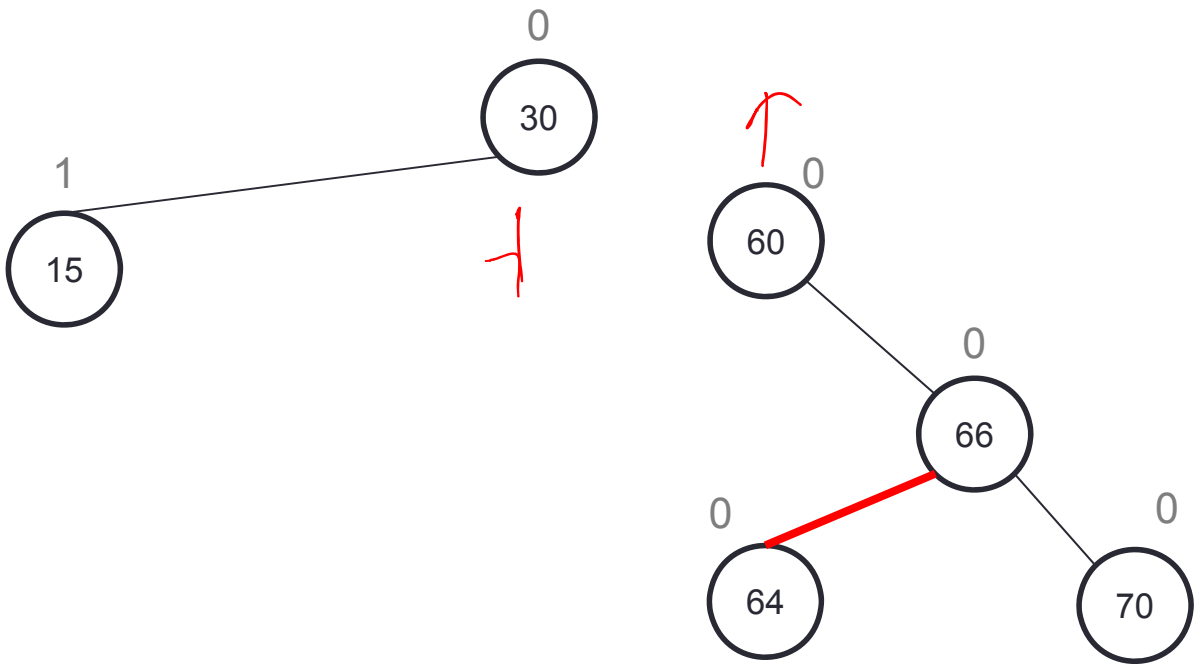


Rotate right around 66 to make a straight line

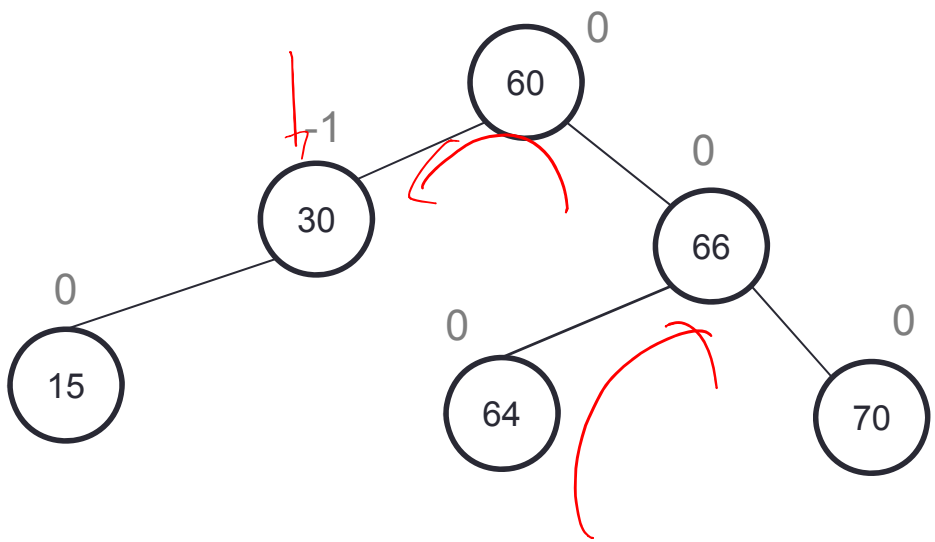


UH OH! Where do we put 64??
Are we stuck?

Rotate right around 66 to make a straight line



Will 64 always reattach there?

A. No, sometimes this doesn't work
B. Yes, this will always work

Finishing the rotation to balance the tree