


# BINARY SEARCH TREES

---

Problem Solving with Computers-I

<https://ucsb-cs24-sp17.github.io/>

The image shows the C++ logo in blue, with the text "C++" in a bold, sans-serif font. Below the logo is a snippet of C++ code in a monospaced font, tilted at an angle. The code is: 

```
#include <iostream>
using namespace std;
int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

# How is PA05 going?

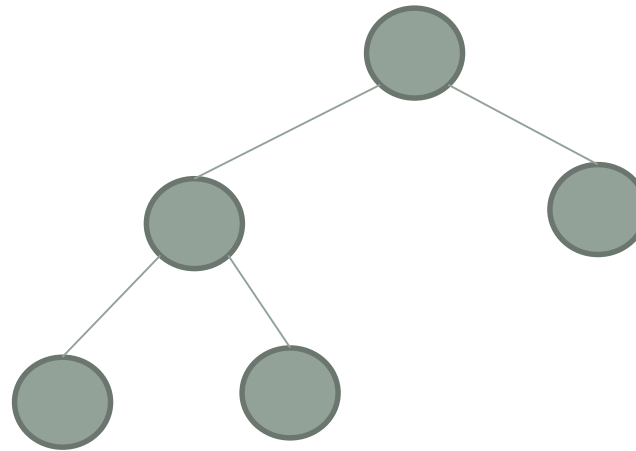
- A. Done
- B. On-track to complete
- C. Having trouble with some functions
- D. Haven't started

# How fast is BST find algorithm?

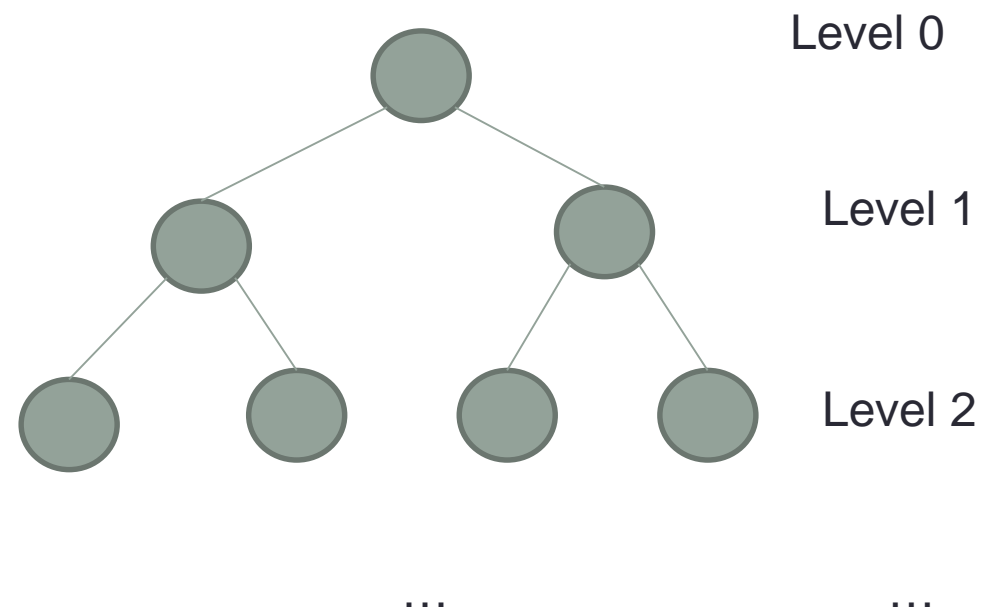
How long does it take to find an element in the tree in terms of the tree's height,  $H$ ?

*Height of a node:* the height of a node is the number of edges on the longest path from the node to a leaf

*Height of a tree:* the height of the root of the tree



Relating H (height) and N (#nodes)  
find is  $O(H)$ , we want to find a  $f(N) = H$



How many nodes are on level L in a completely filled binary search tree?

- A. 2
- B. L
- C.  $2 * L$
- D.  $2^L$

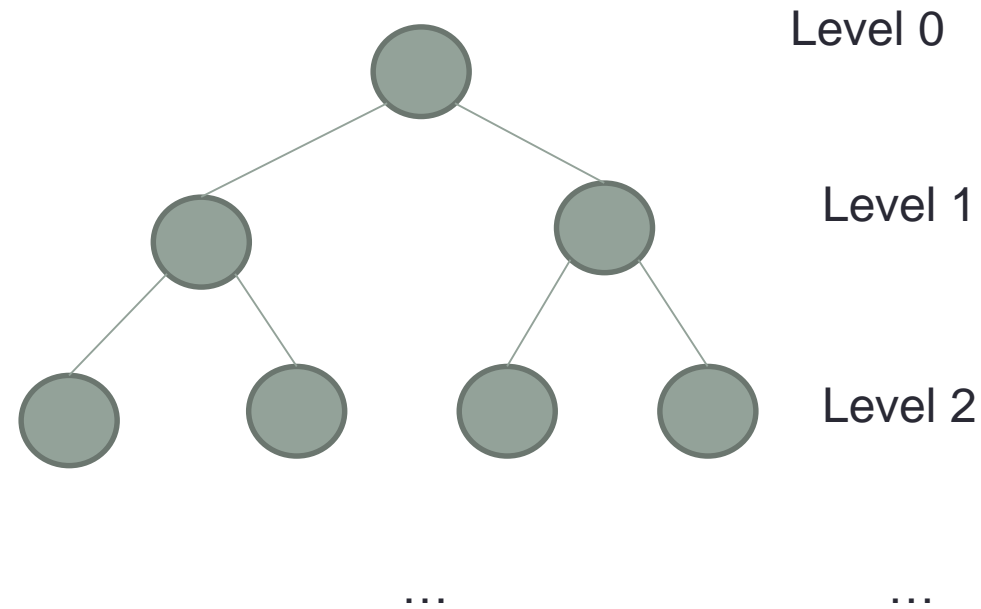
Relating H (height) and N (#nodes)  
 find is  $O(H)$ , we want to find a  $f(N) = H$

$$N = \sum_{L=0}^H 2^L = 2^{H+1} - 1$$

$$N+1 = 2^{H+1}$$

$$\log_2(N+1) = H+1$$

$$H = \log_2(N+1) - 1$$



Finally, what is the height (exactly) of the tree in terms of N?

$$H = \log_2(N + 1) - 1$$

And since we knew finding a node was  $O(H)$ , we now know it is  $O(\log_2 N)$

# Worst case analysis

Are binary search trees *really* faster than linked lists for finding elements?

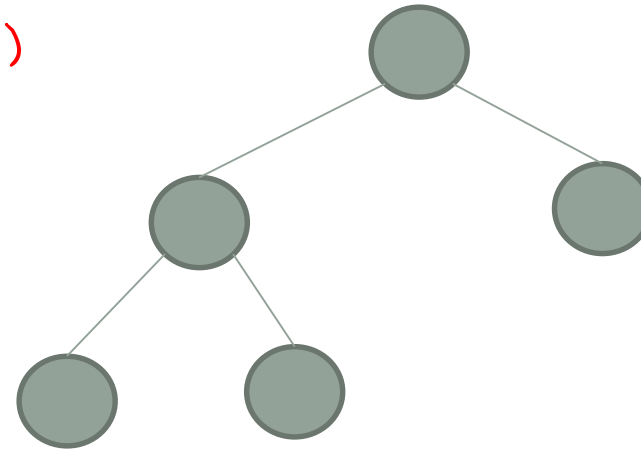
- A. Yes

- **B. No**

The running time of search in a BST is  $O(H)$   
 where  $H$  is the height of the tree.

$H$  depends on the order in which  
 elements are inserted in the BST

In the worst case  $H = N$   
 that is the BST looks like a  
 linked list

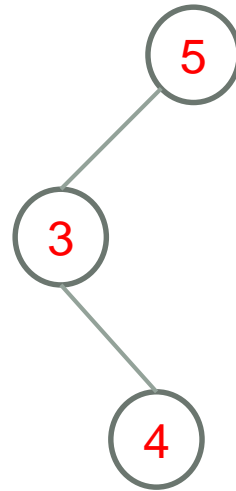


# Average case analysis of a “successful” find

Given a BST having  $N$  nodes  $x_1, \dots, x_N$ , such that  $\text{key}(x_i) = k_i$

How many compares to locate a key in the BST?

1. Worst case:  $3 \propto \text{height}$
2. Best case:  $1$
3. Average case: depends on the probability of searching each key



Here is the result! Proof is a bit involved but if you are interested in the proof, come to office hours

$$D_{avg}(N)$$

Average #comparisons to find a single item in any BST with N nodes

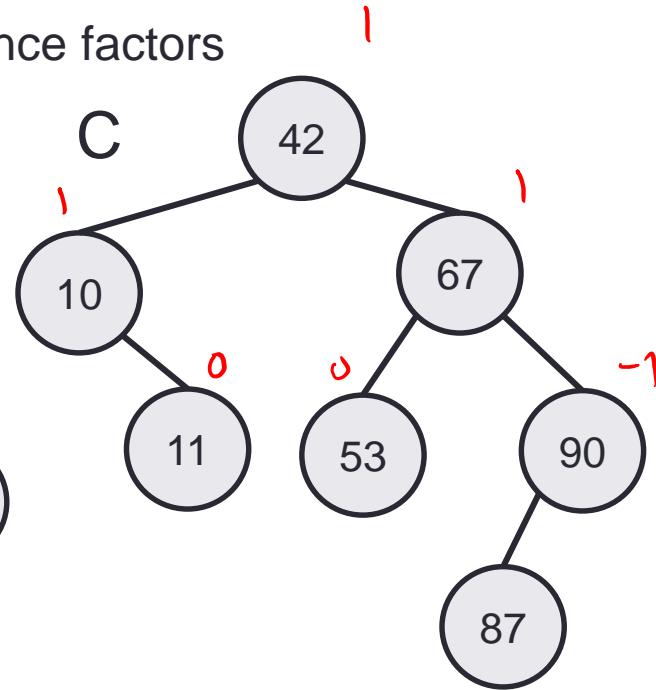
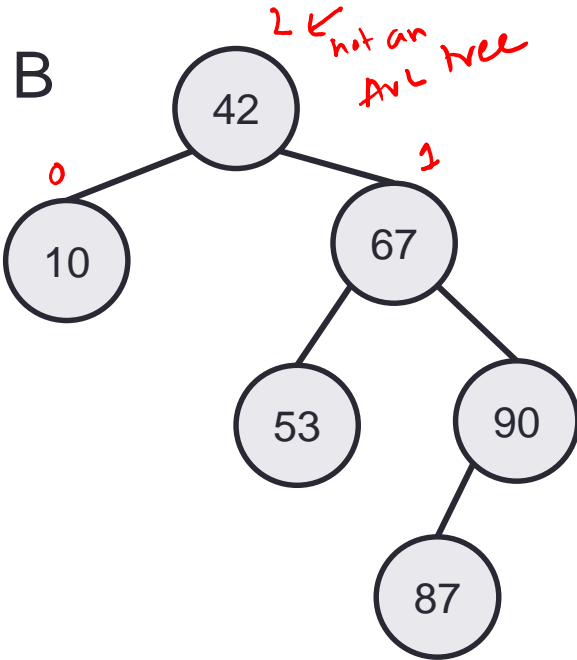
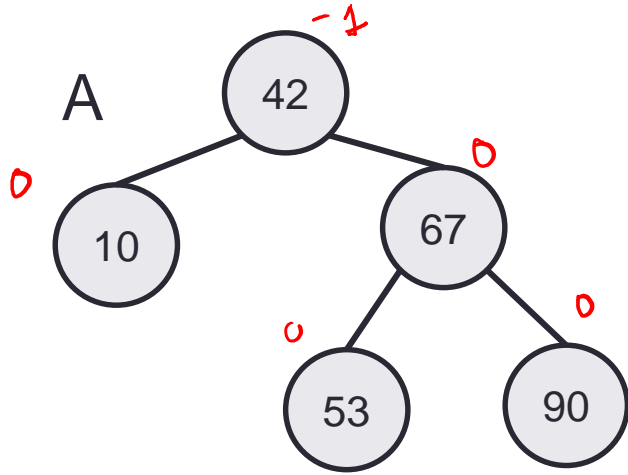
$$D_{avg}(N) \approx 1.386 \log_2 N$$

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



# Which of the following is/are balanced trees?

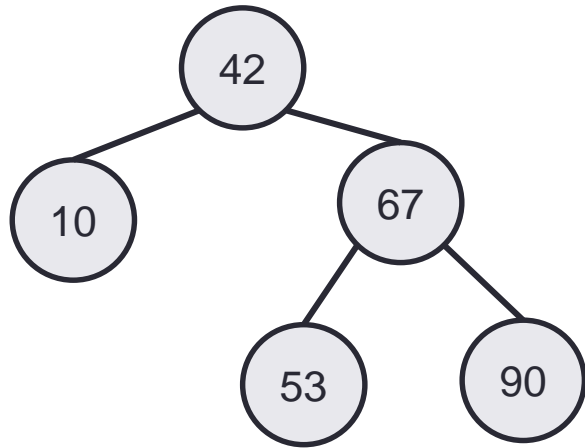
And thus can become AVL trees by adding the balance factors



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

Annotate the trees with balance factors

# AVL Tree Balance Factors



The balance factor at a node in a tree is  
 $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$

The height of tree with 1 node is 0

The height of a tree with 0 nodes is -1

# An AVL Tree is *worst case* $O(\log N)$ to find an element!

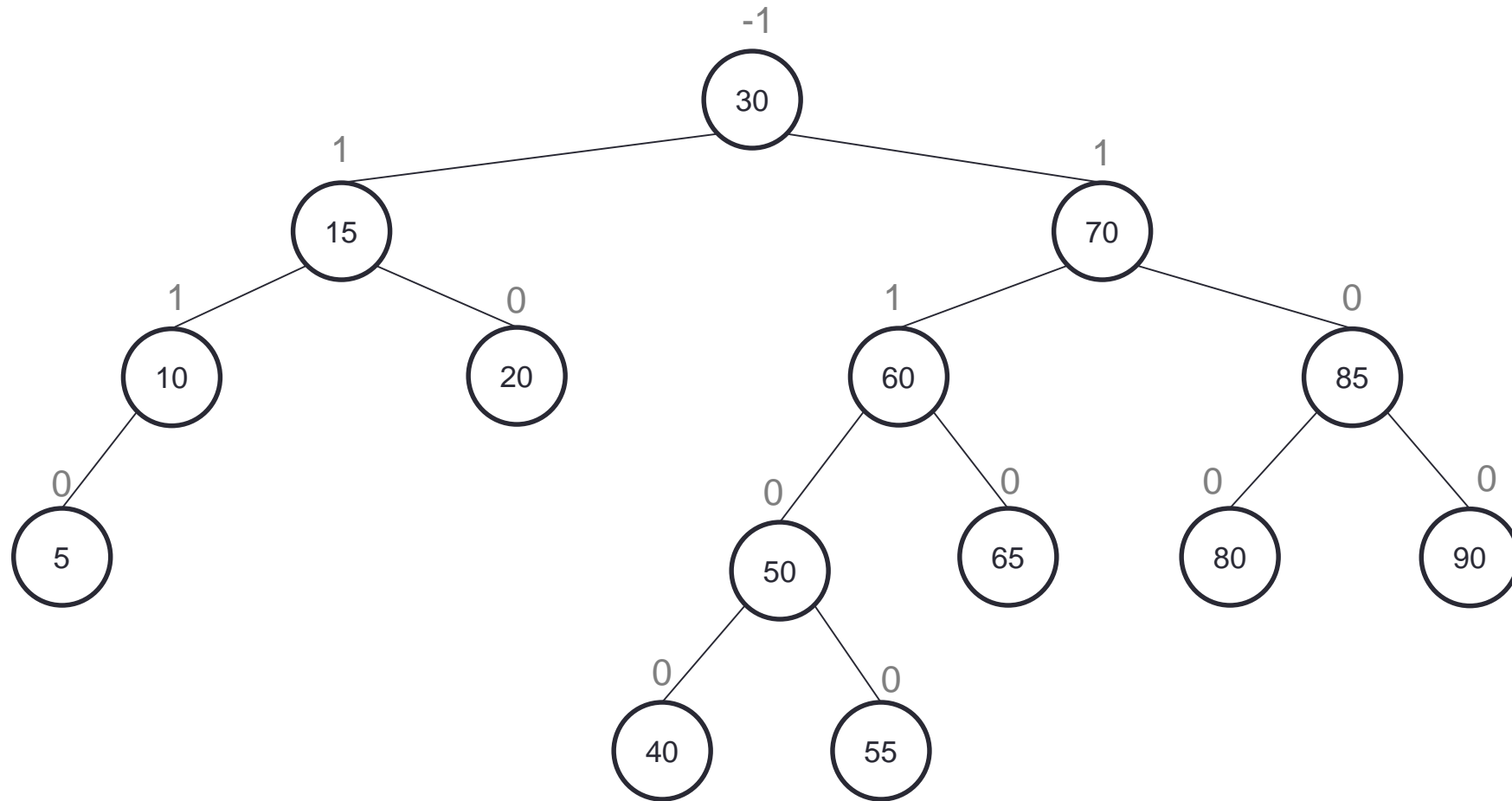
How would you prove this?

Come up with a formula that shows that the height of the tallest AVL tree with  $N$  nodes is never bigger than  $c \cdot \log N + k$ , 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.

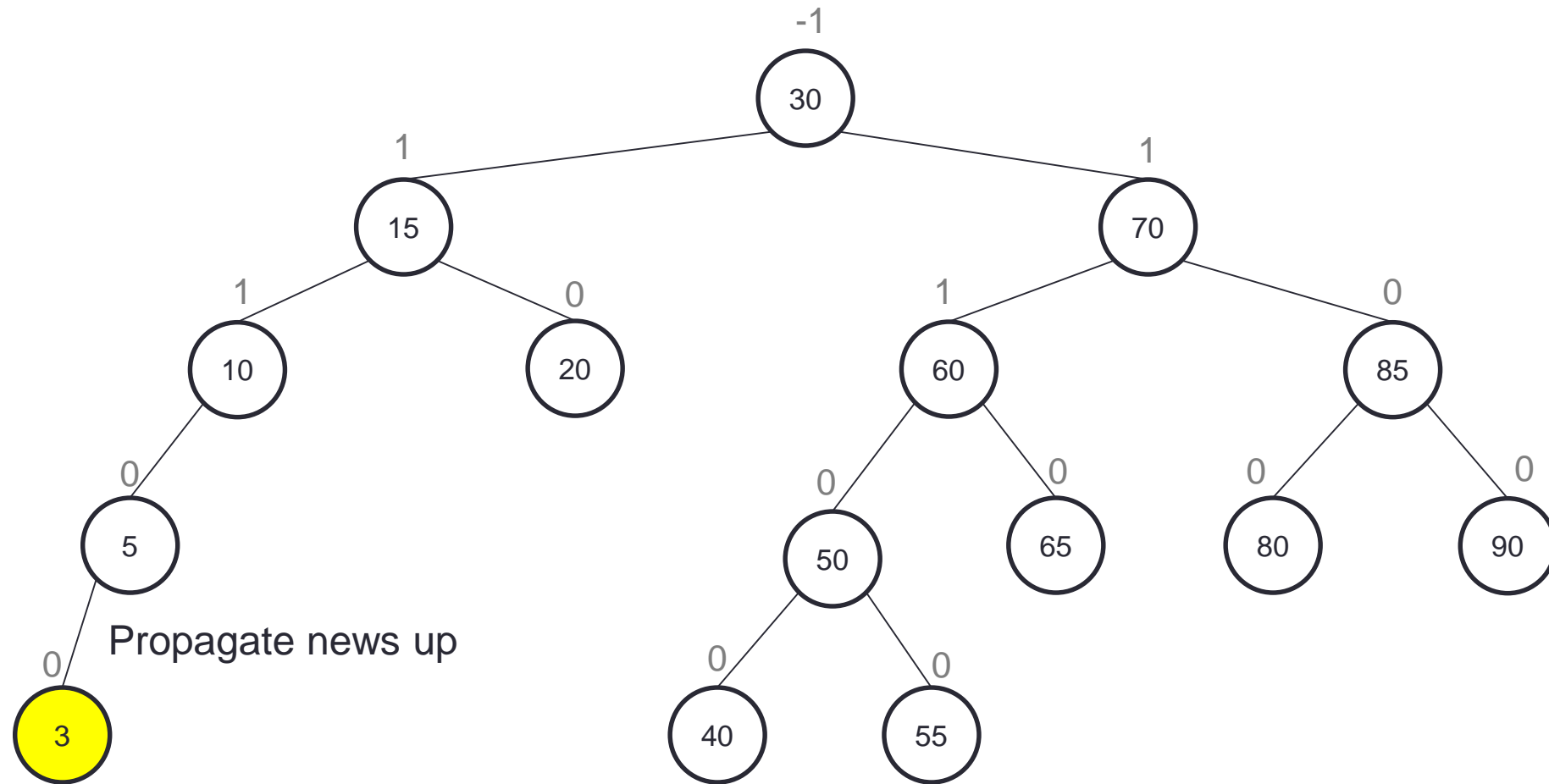
But how does the tree stay balanced??

# Inserting and rebalancing



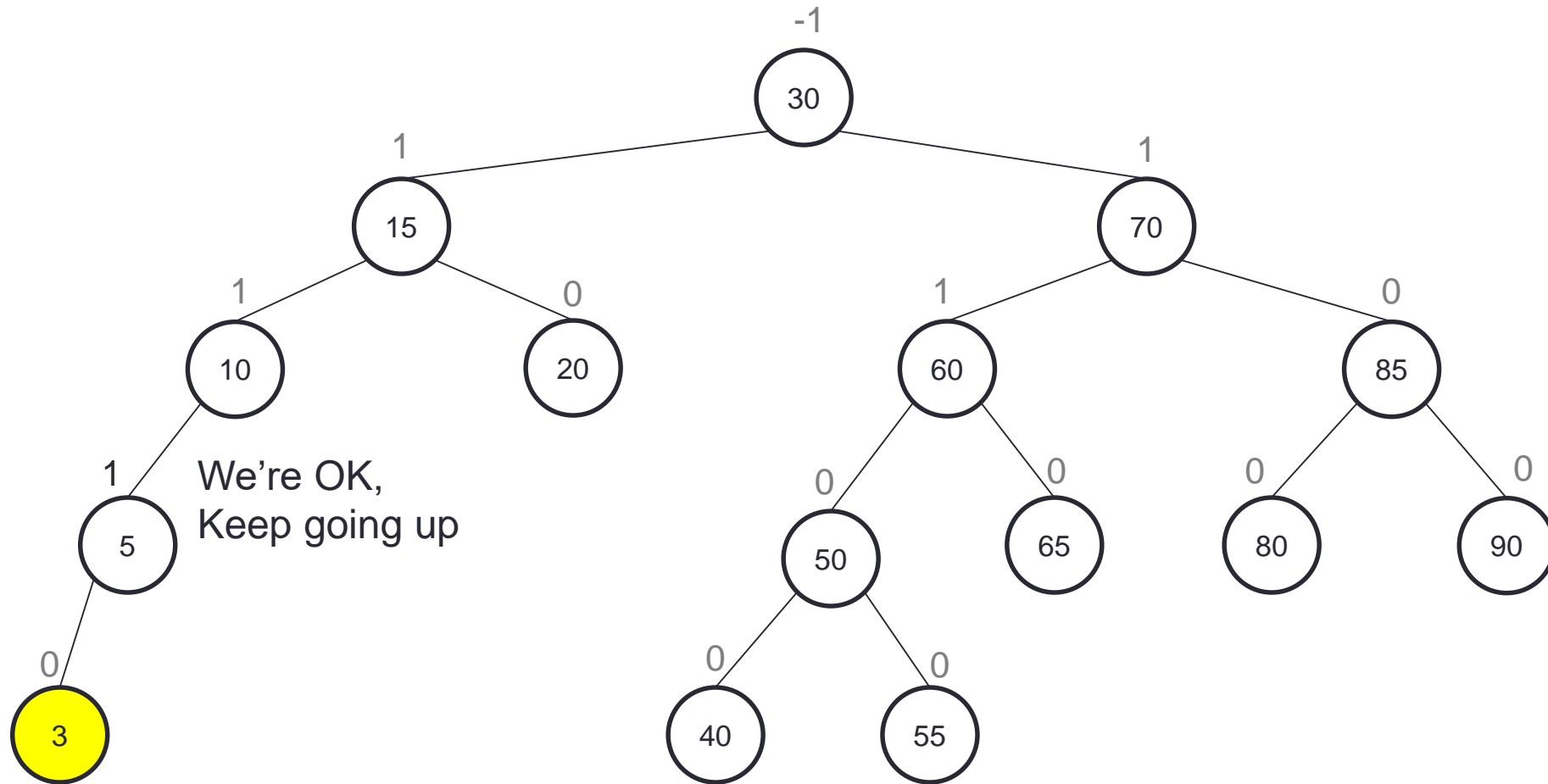
Insert 3

# Inserting and rebalancing



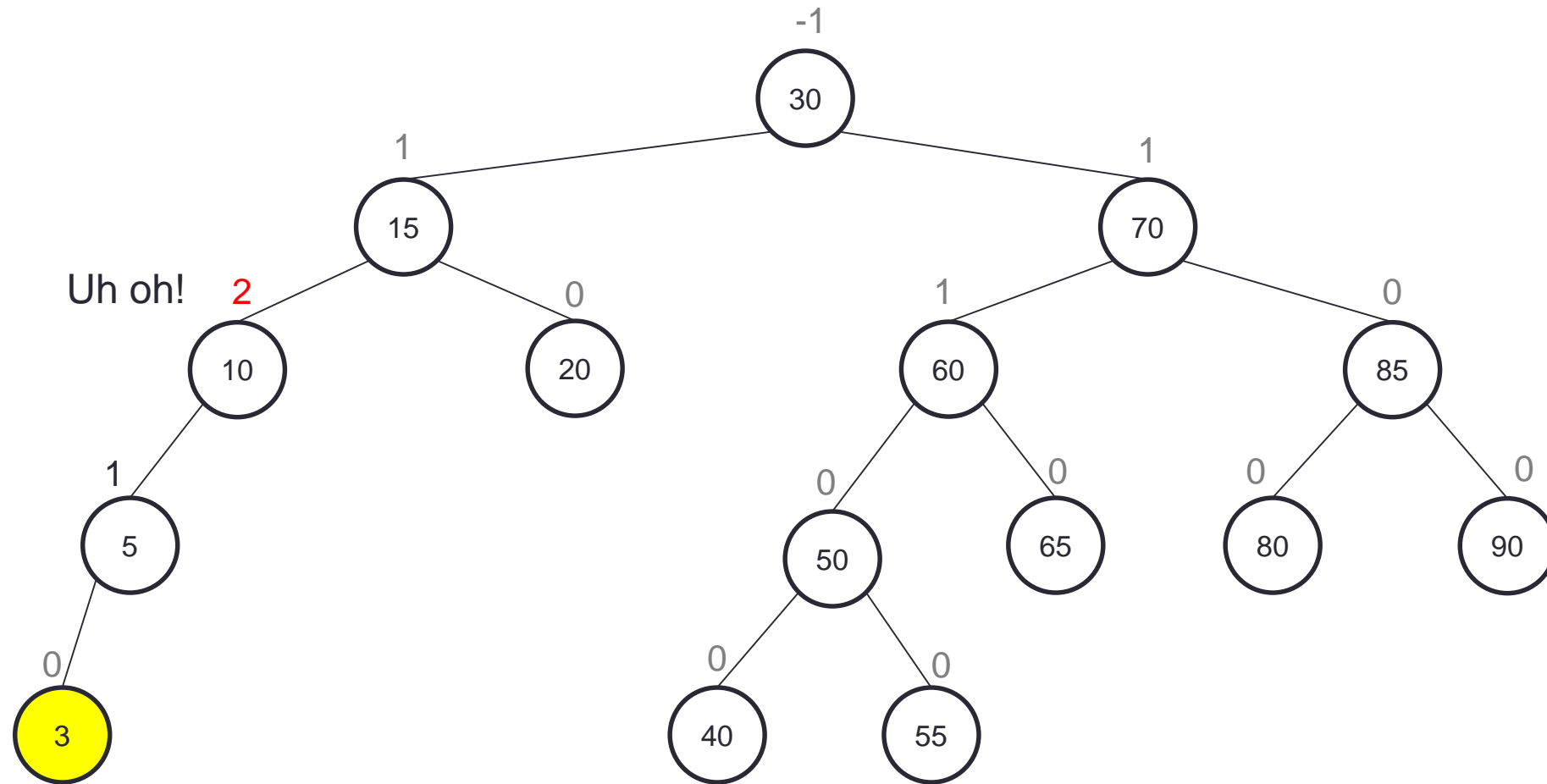
Insert 3

# Inserting and rebalancing

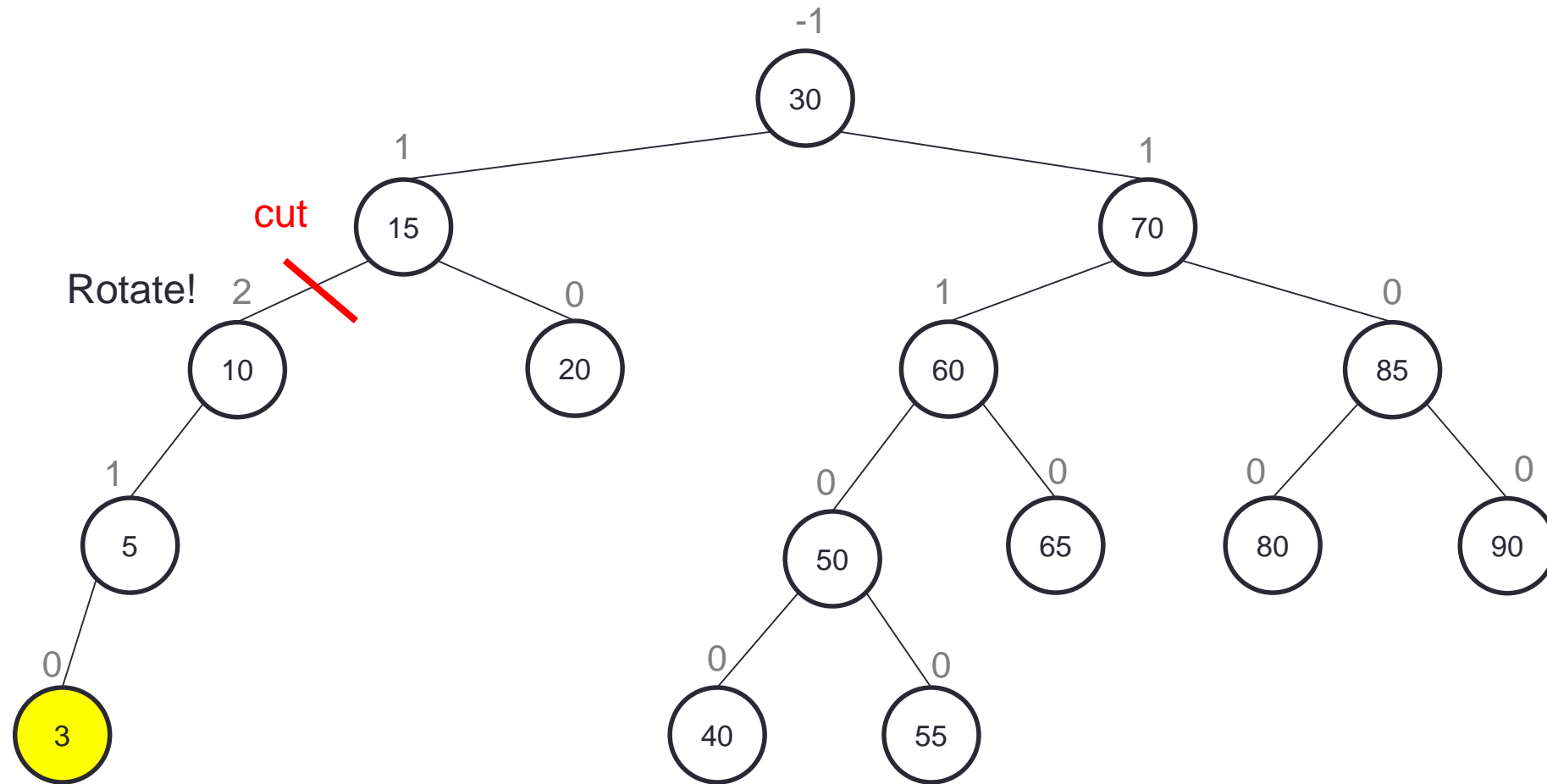


Insert 3

# Inserting and rebalancing



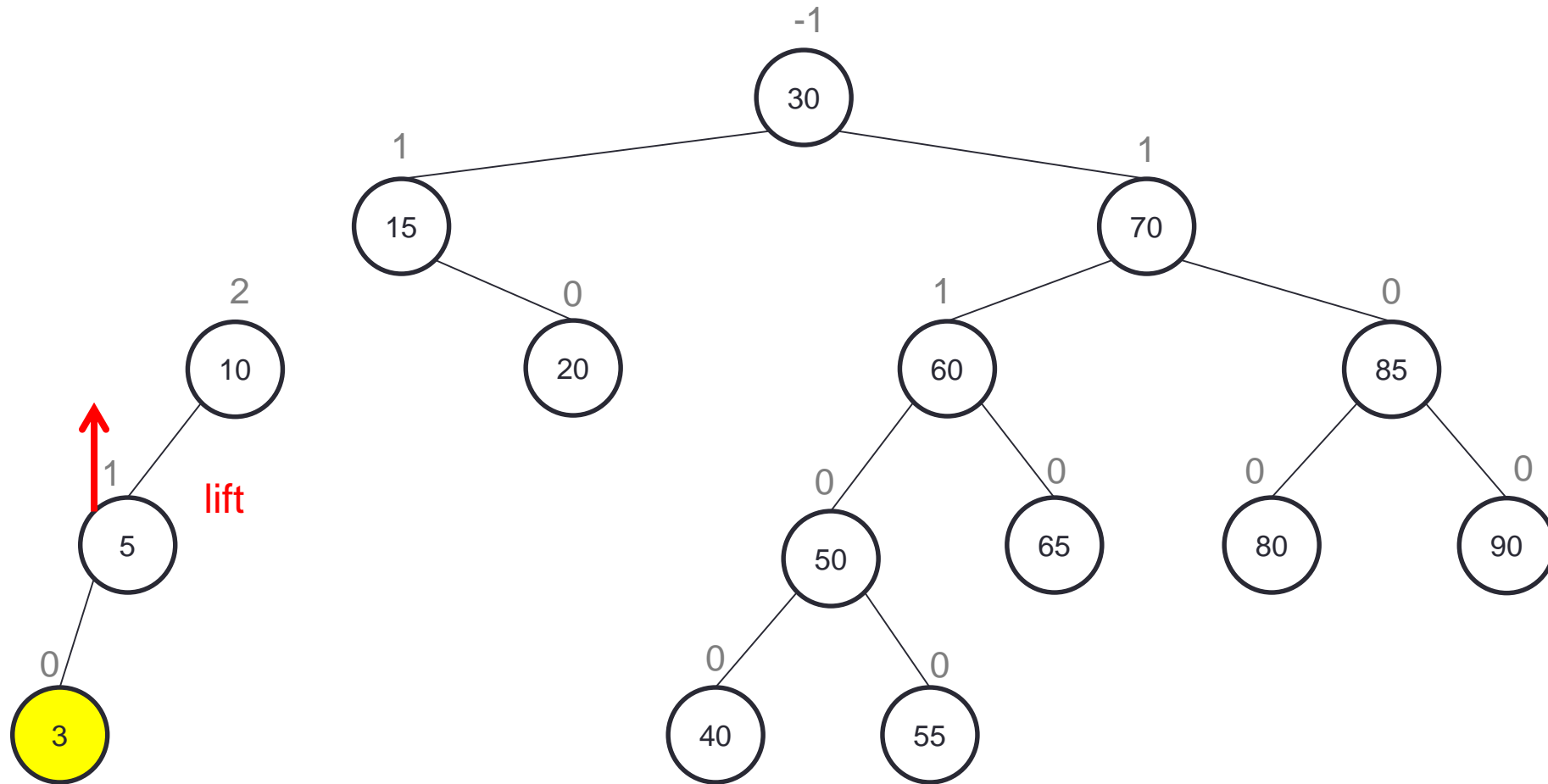
# Inserting and rebalancing



Insert 3

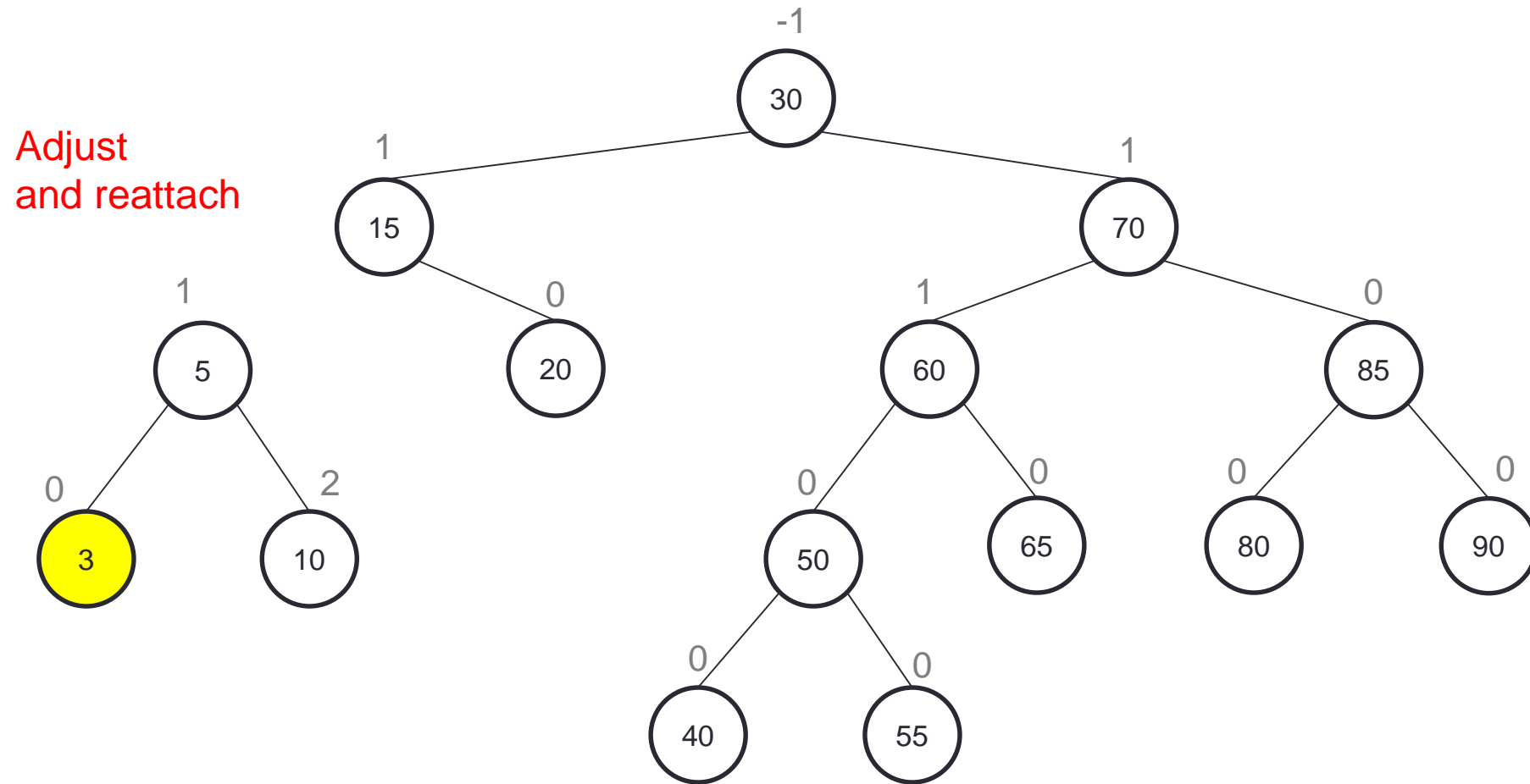


# Inserting and rebalancing



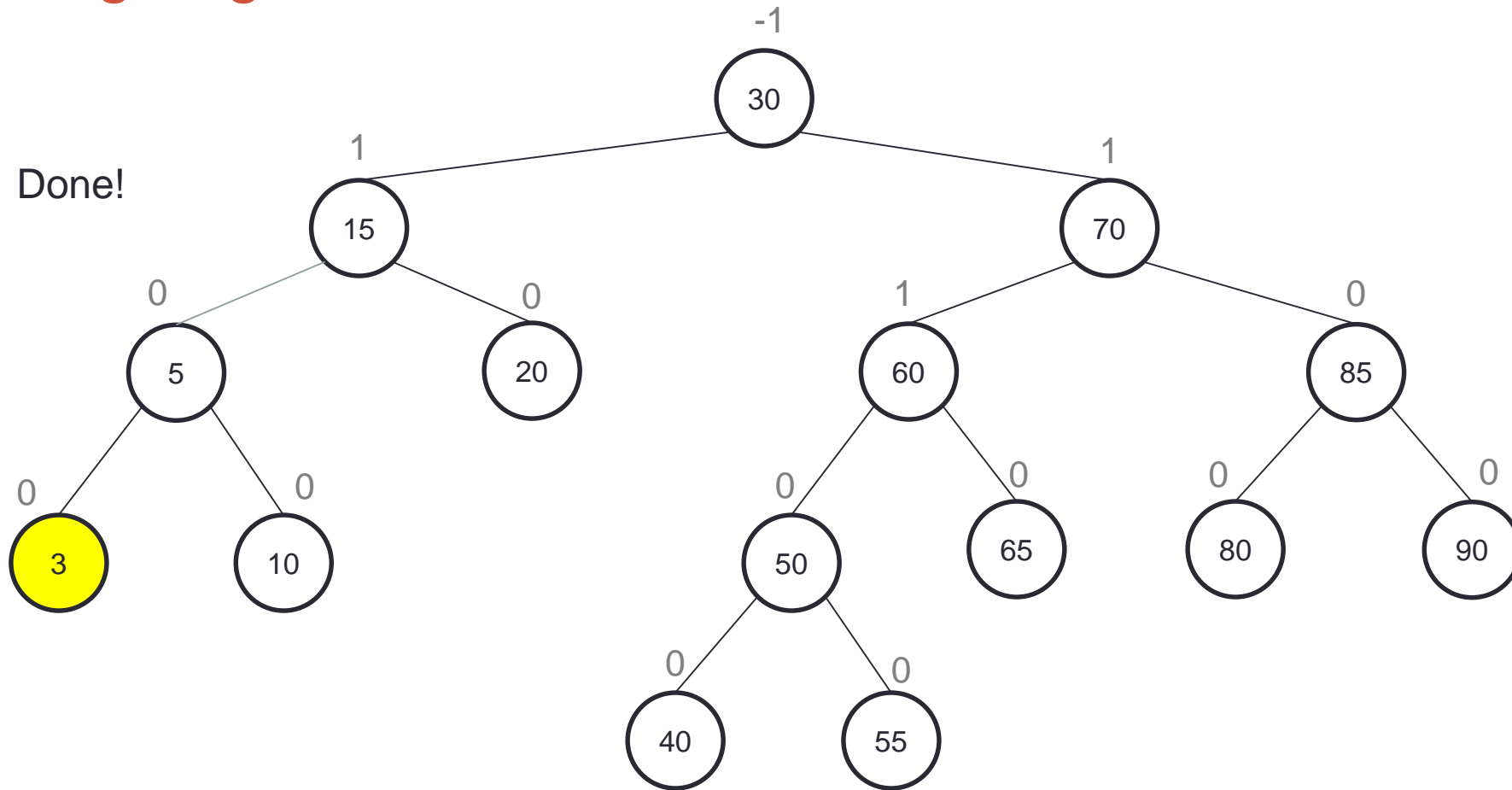
Insert 3

# Inserting and rebalancing



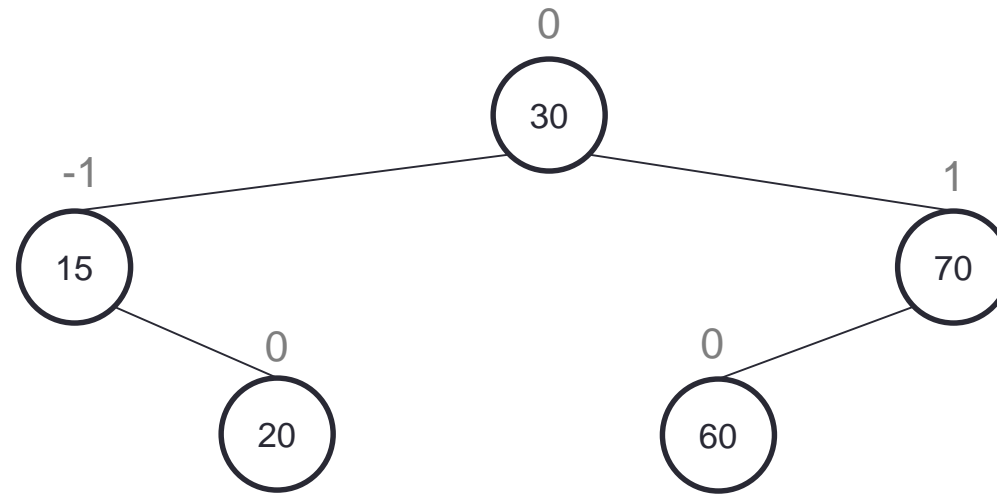
Insert 3

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



Insert 3

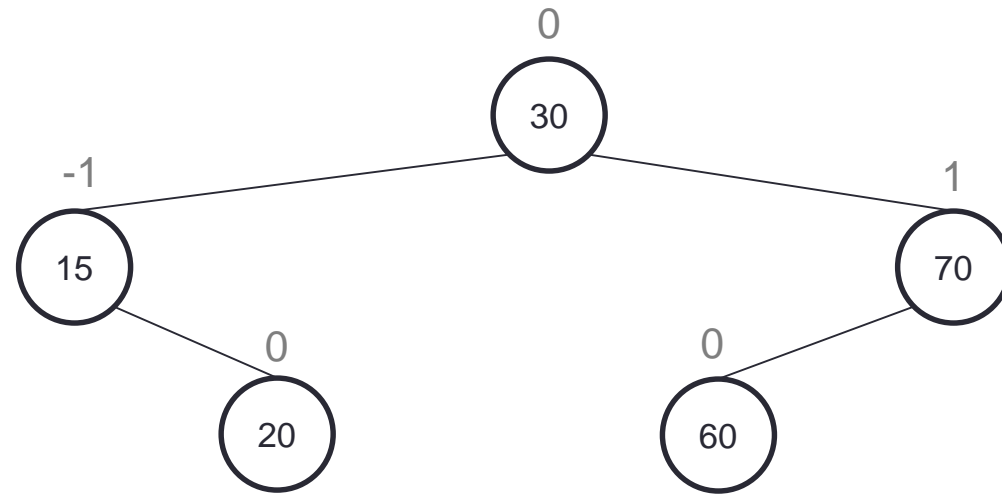
# Single rotation practice



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

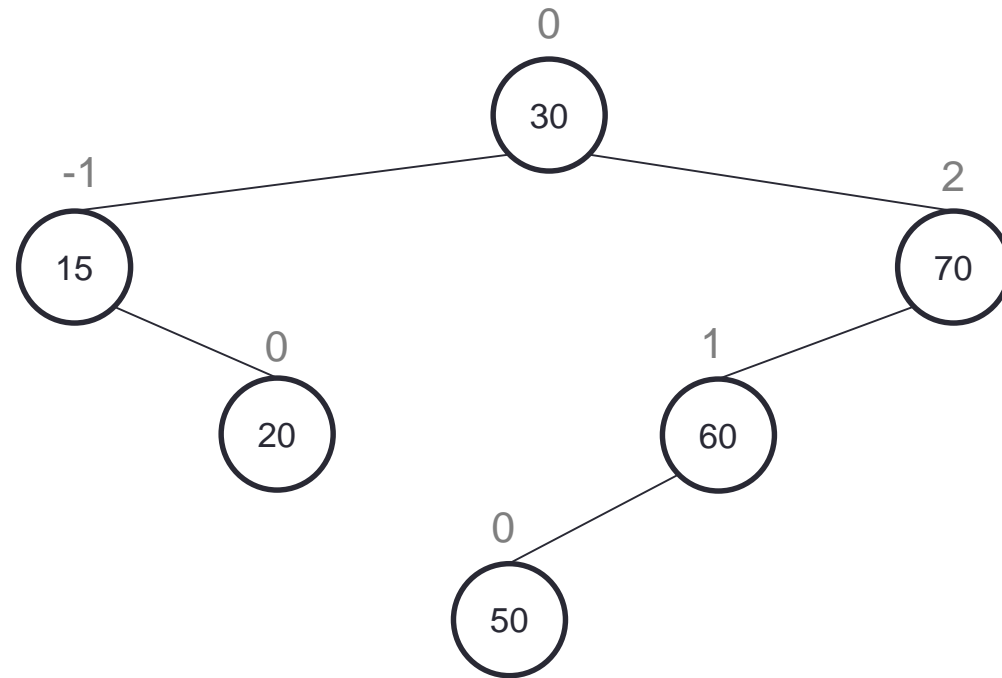
- A. 71
- B. 10
- C. 50
- D. 66

# Single rotation practice



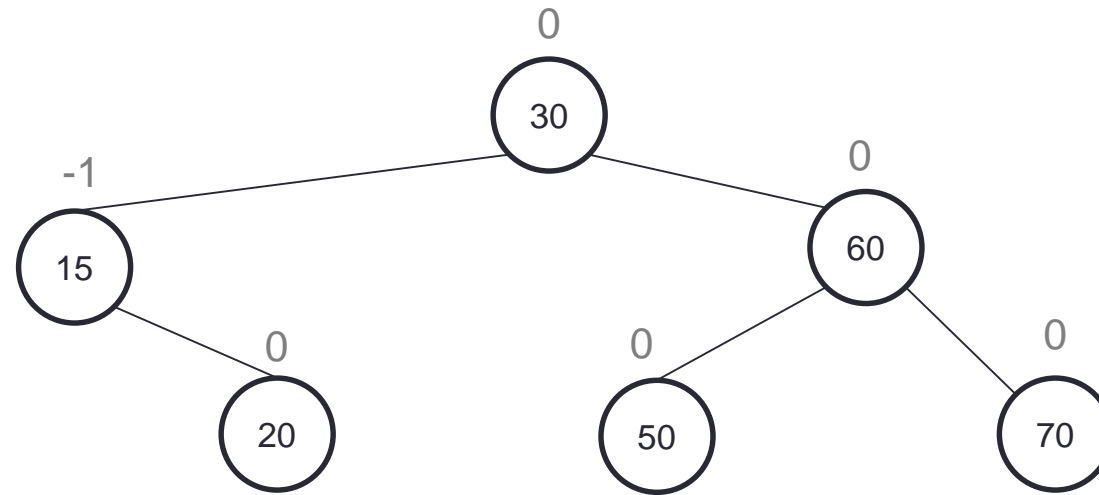
Insert 50. Draw the resulting AVL tree. (Don't peek)

# Single rotation practice



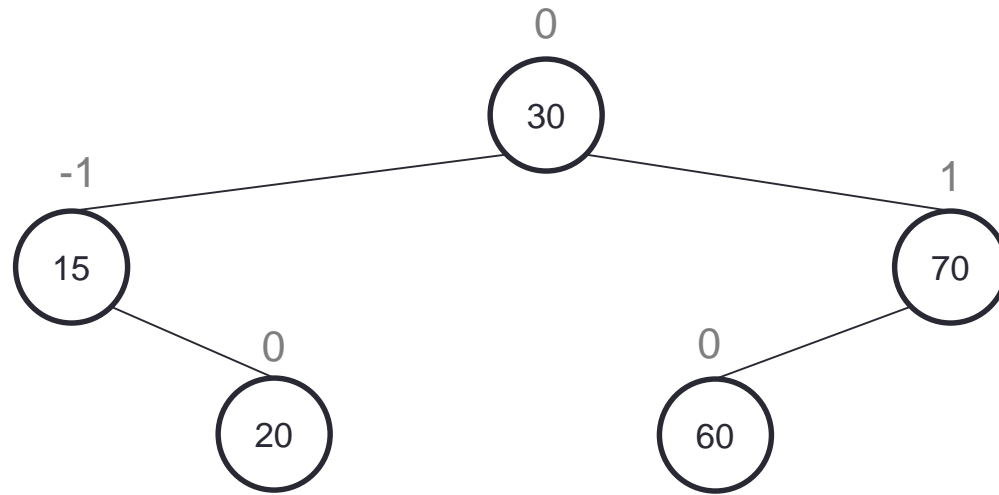
After insertion

# Single rotation practice



After rotation

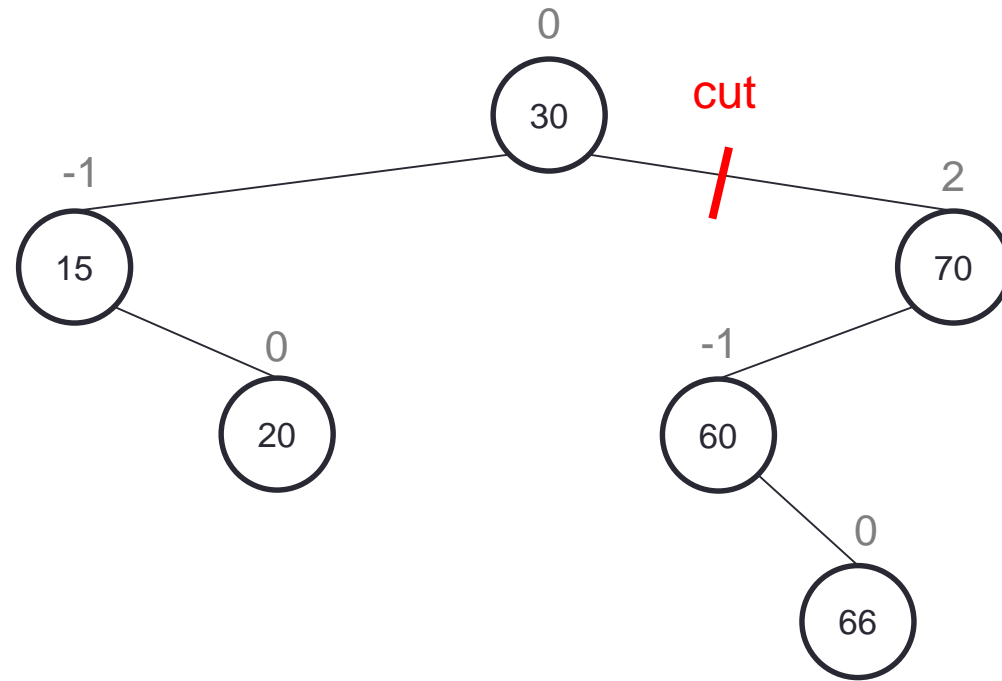
# Single rotation is not enough



What happens if we insert 66?

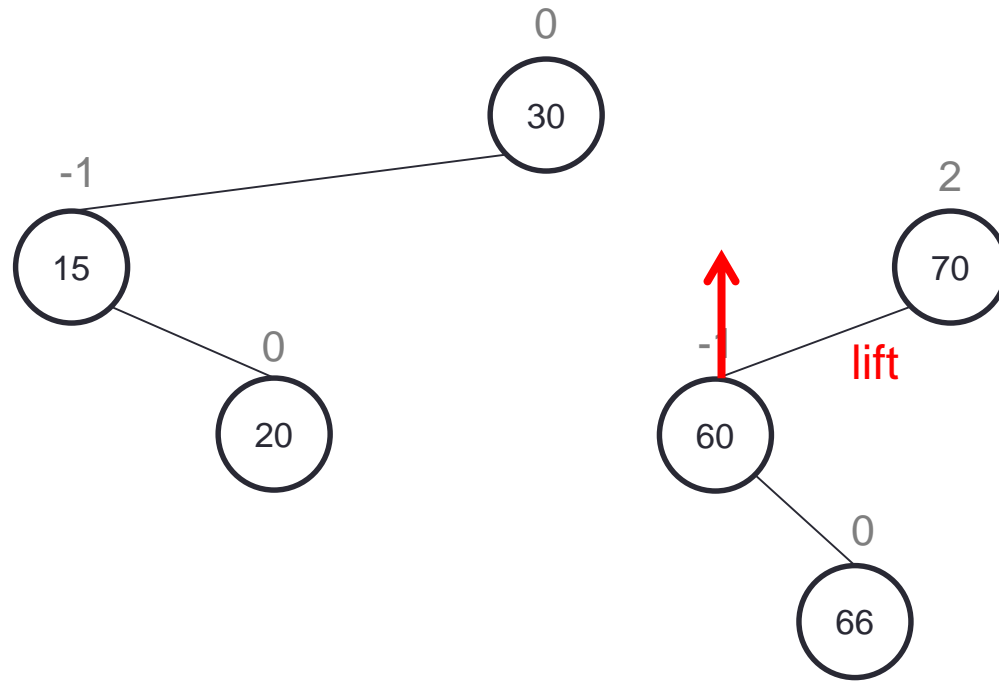


# Single rotation is not enough

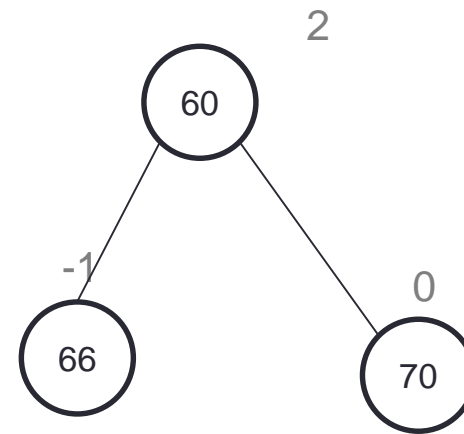
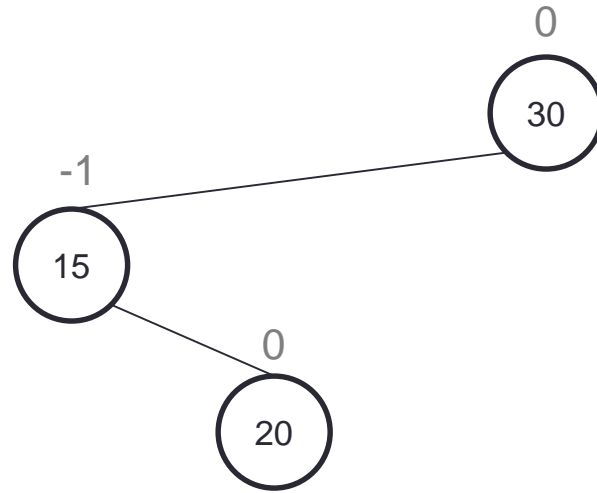


Why won't a single rotation work? Try it.

# Single rotation is not enough

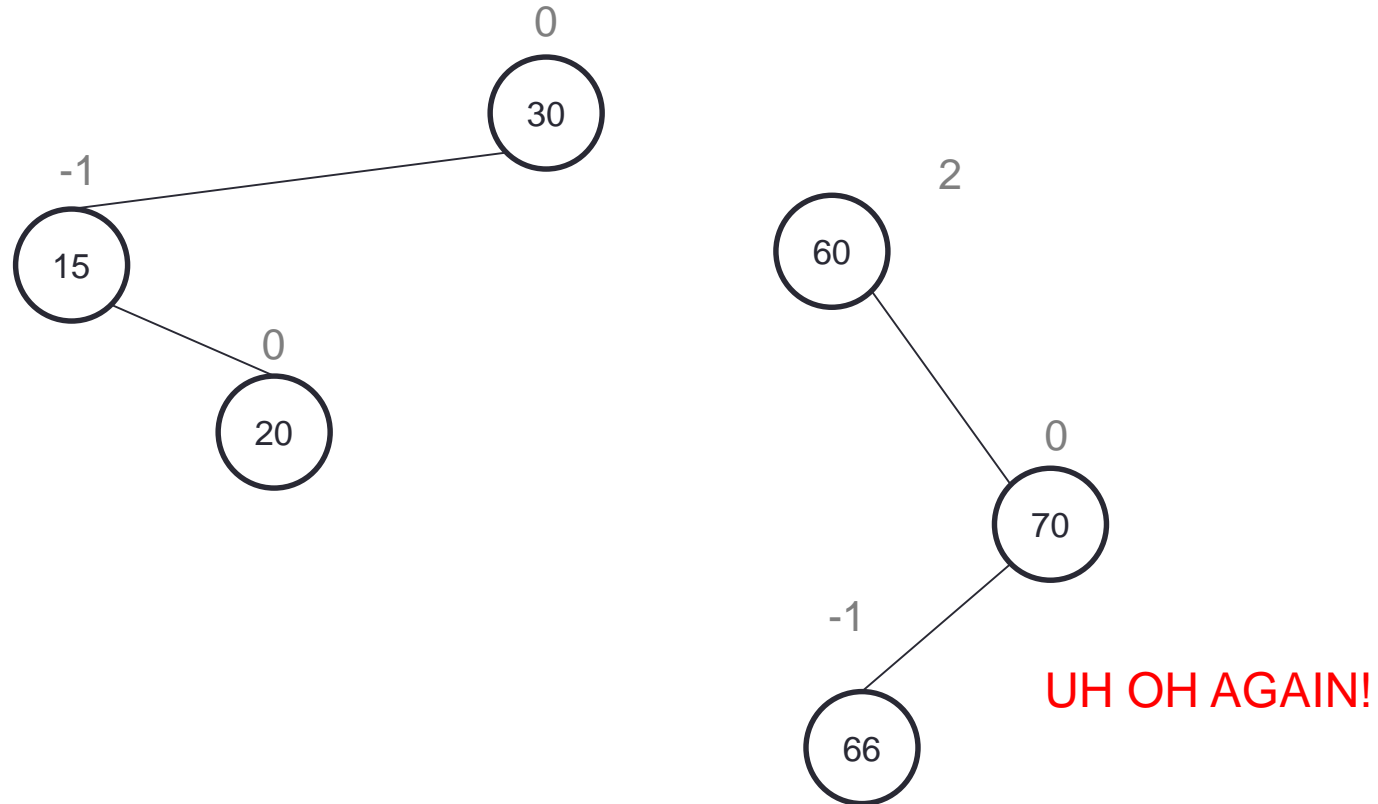


# Single rotation is not enough



UH OH!

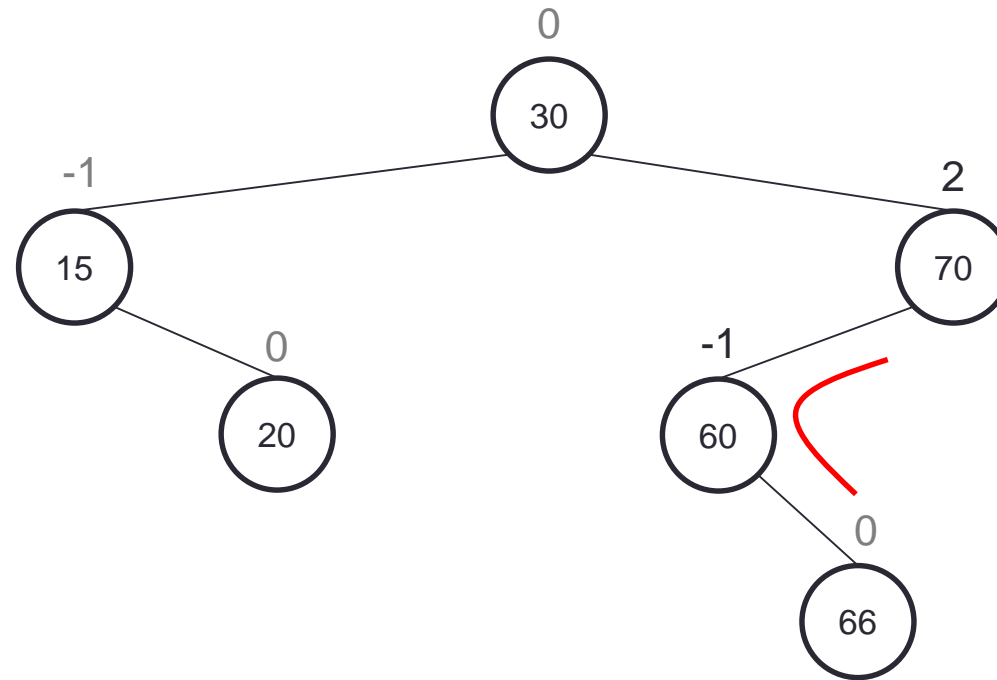
# Single rotation is not enough



Reattaching 66 here will always work with respect to the BST properties, and we know that 66 will always fit here because 60 used to be 70s left child.

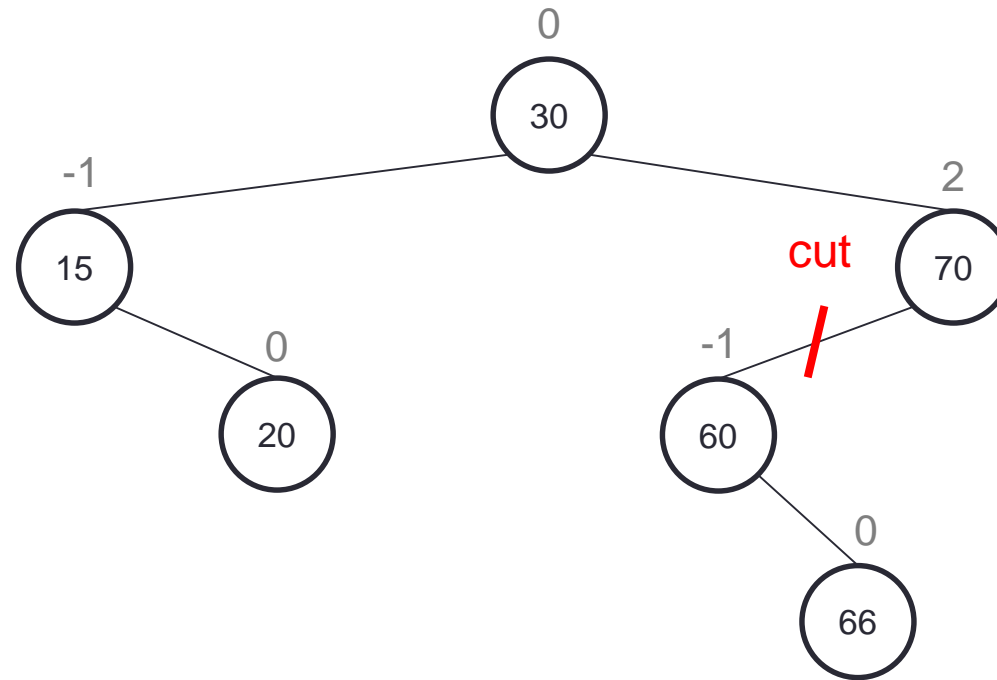
The problem is that this won't fix the balance issue!

# Double rotation to the rescue



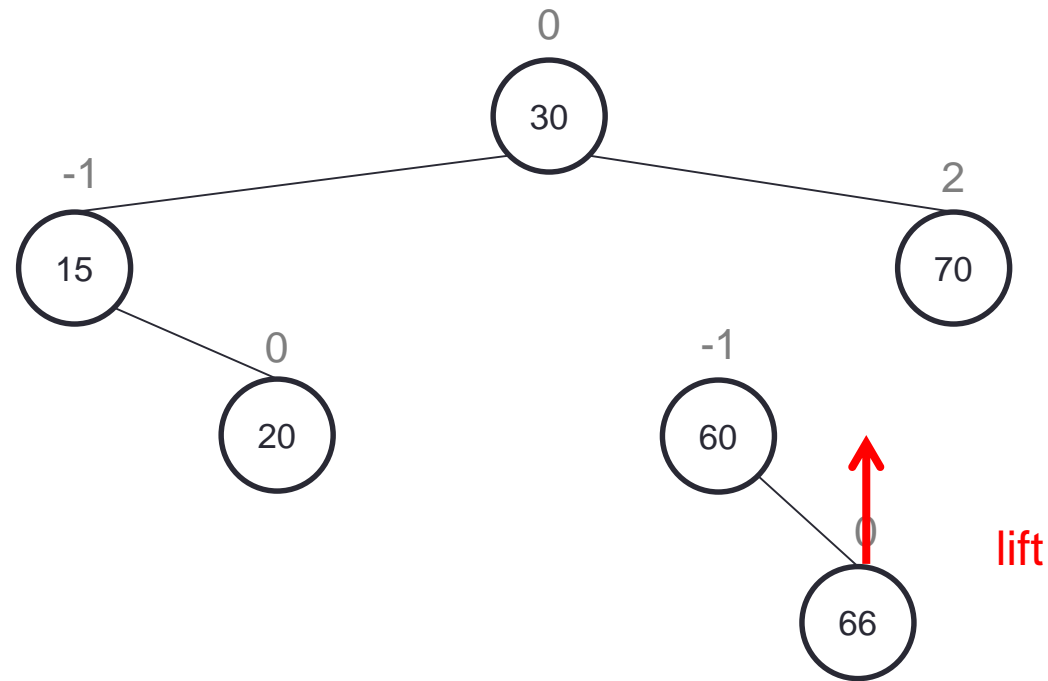
Single rotations only work to balance the tree when involved nodes are “in a line”  
This is not the case here. We want 66 to be the top node, not 60.  
So we will first rotate left at 60 to get 66 in the middle, then we can rotate right at 70.

# Double rotation to the rescue



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

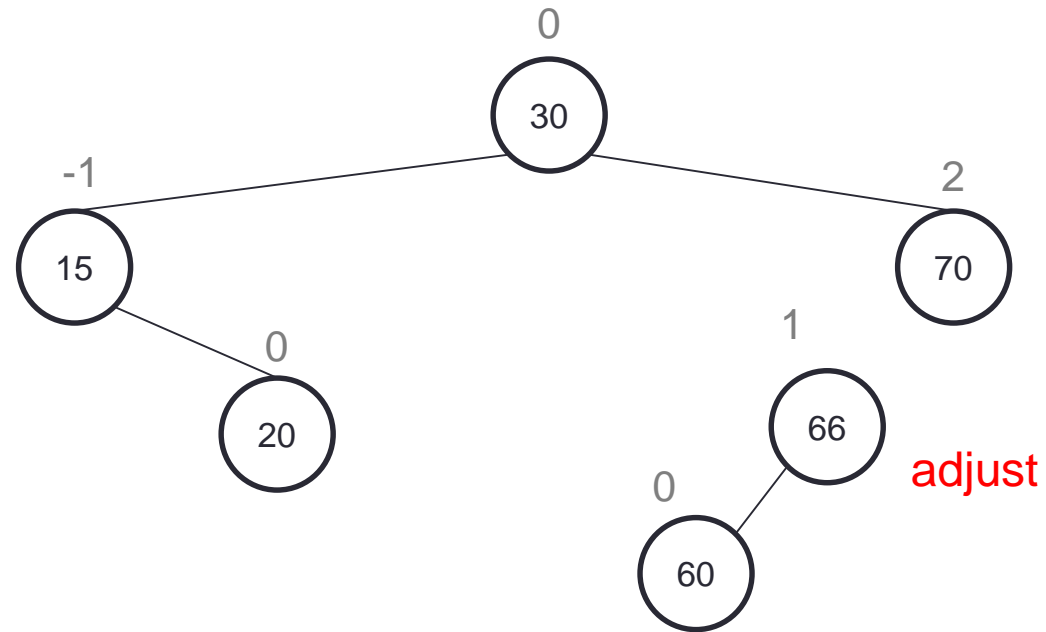
# Double rotation to the rescue



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.

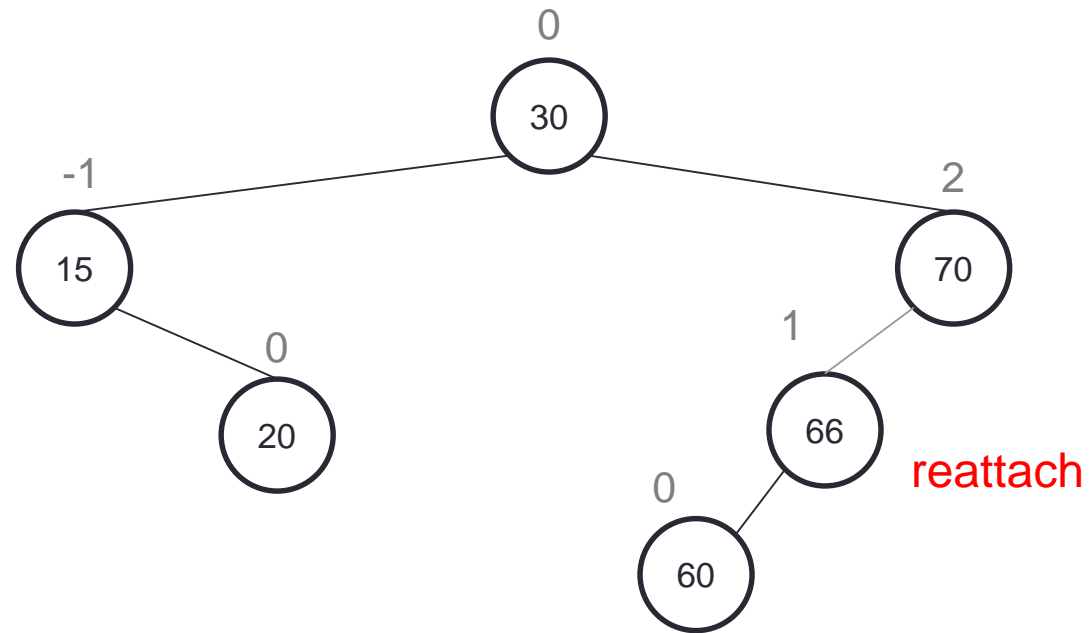
# Double rotation to the rescue



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

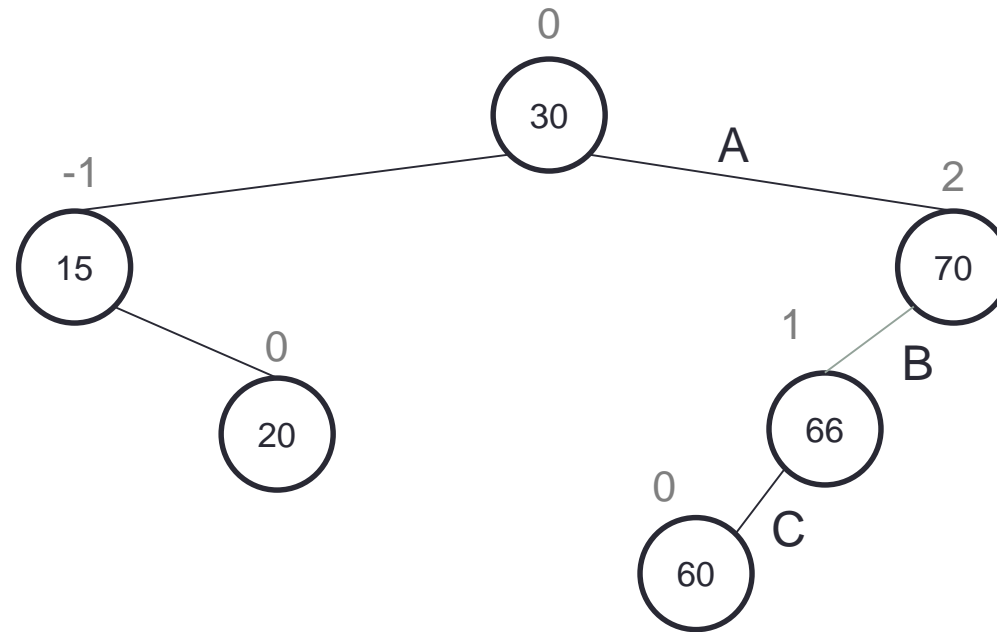


# Double rotation to the rescue



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

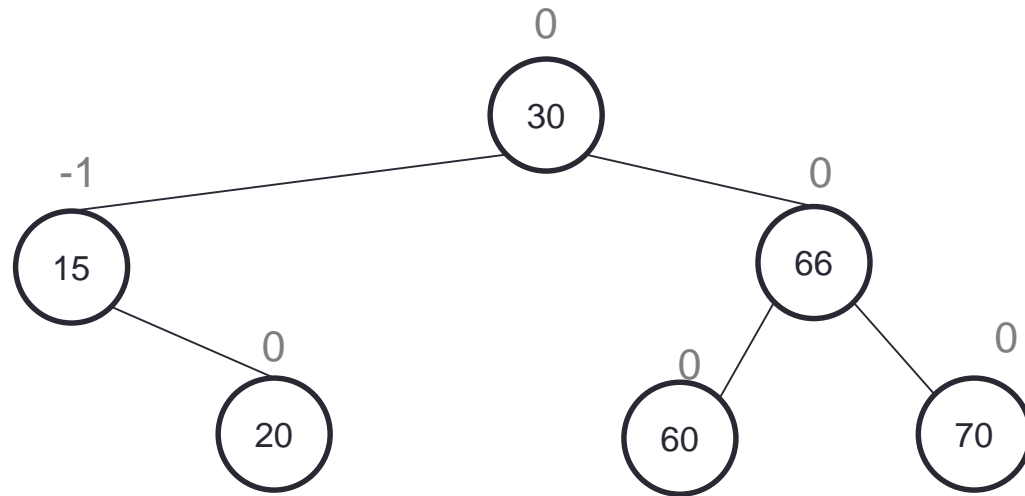
# Double rotation to the rescue



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 the second rotation?

# Double rotation to the rescue



str :: Set < >

---

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