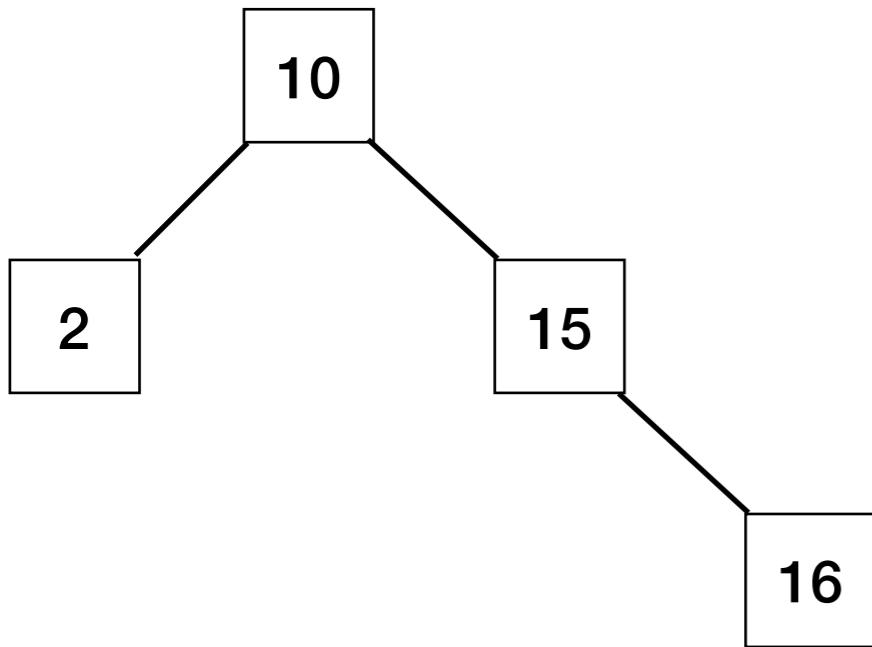


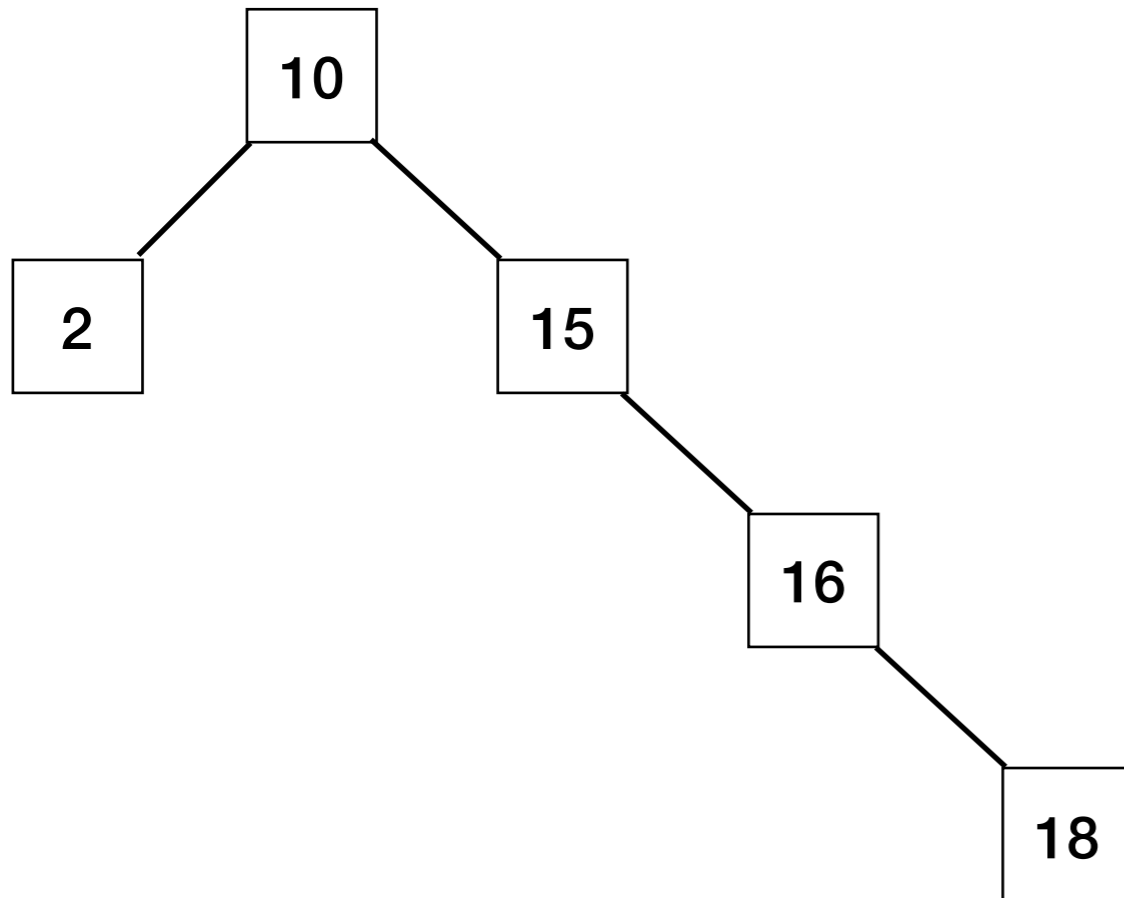
Supplemental AVL Insertion Examples

CSCI 241
Prof. Wehrwein

`avlInsert(root, 18)`

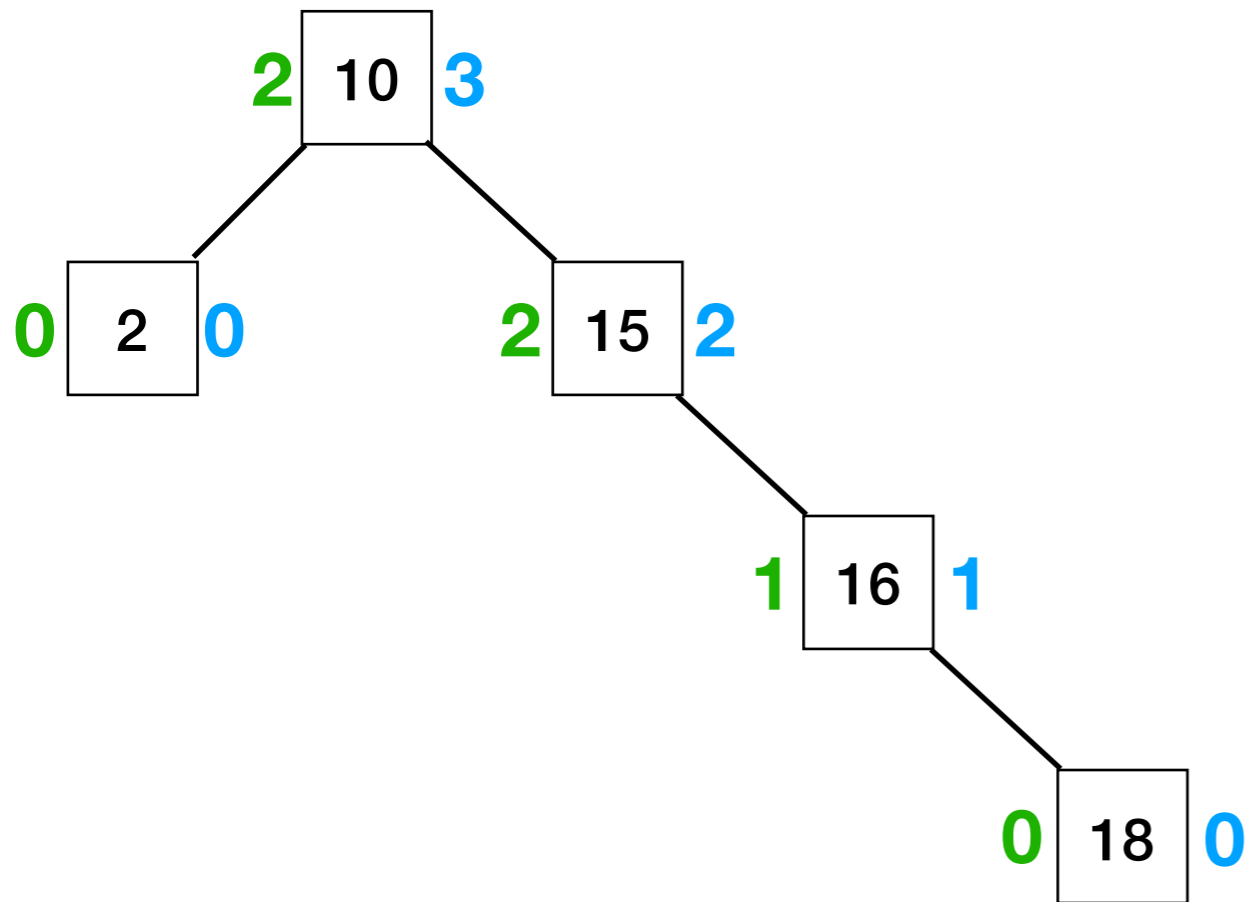


`avlInsert(root, 18)`



Exercise: Write the tree above. To the right of each node, write the height of the subtree rooted at the node. To the left, write the balance factor of the node.

avlInsert(root, 18)



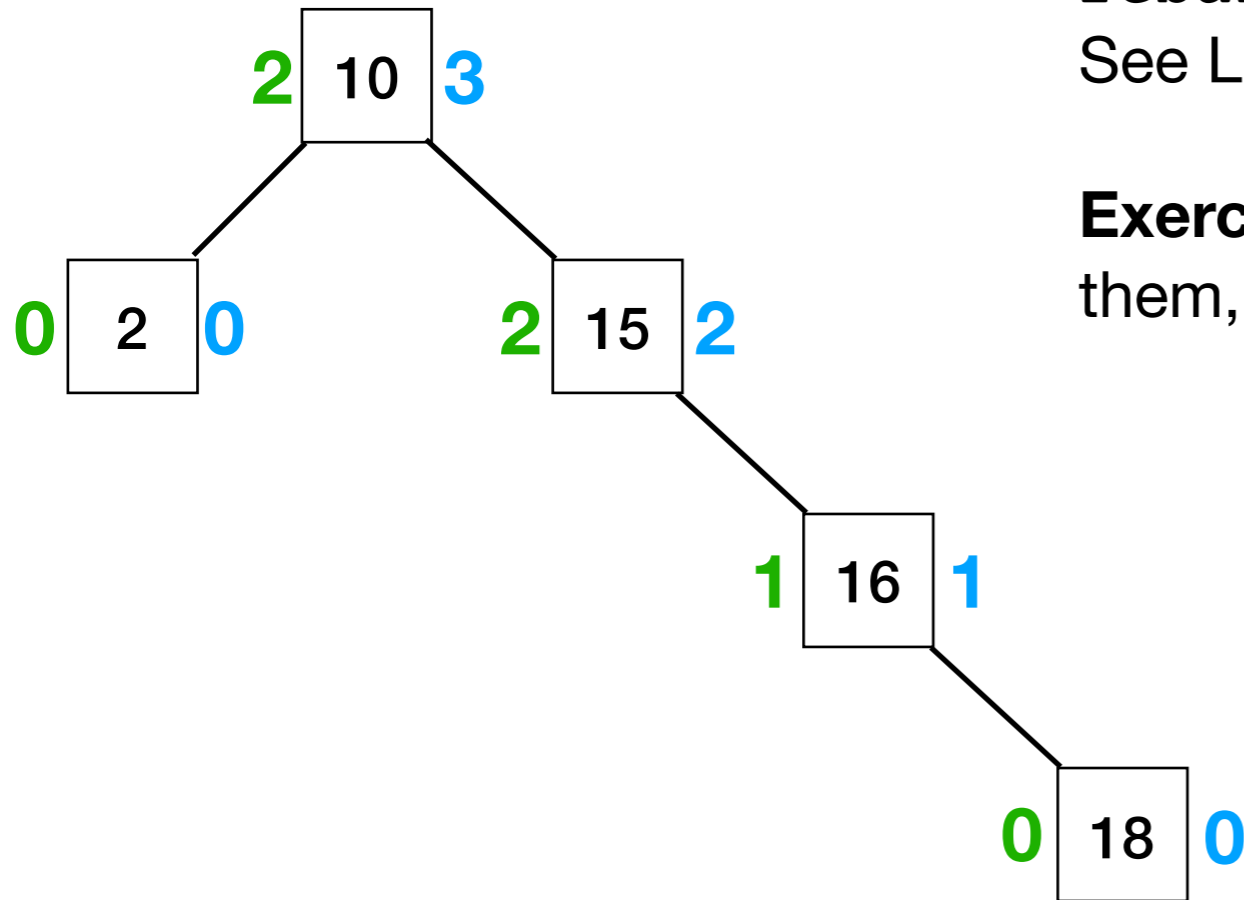
Balance(n): $\text{height}(\text{n.right}) - \text{height}(\text{n.left})$

Exercise: Write the tree above. To the right of each node, write the height of the subtree rooted at the node. To the left, write the balance factor of the node.

`avlInsert(root, 18)`

After the new node is inserted into the tree, the recursion will walk back up the tree, calling `rebalance` on each parent node in succession. See L12 slides for details.

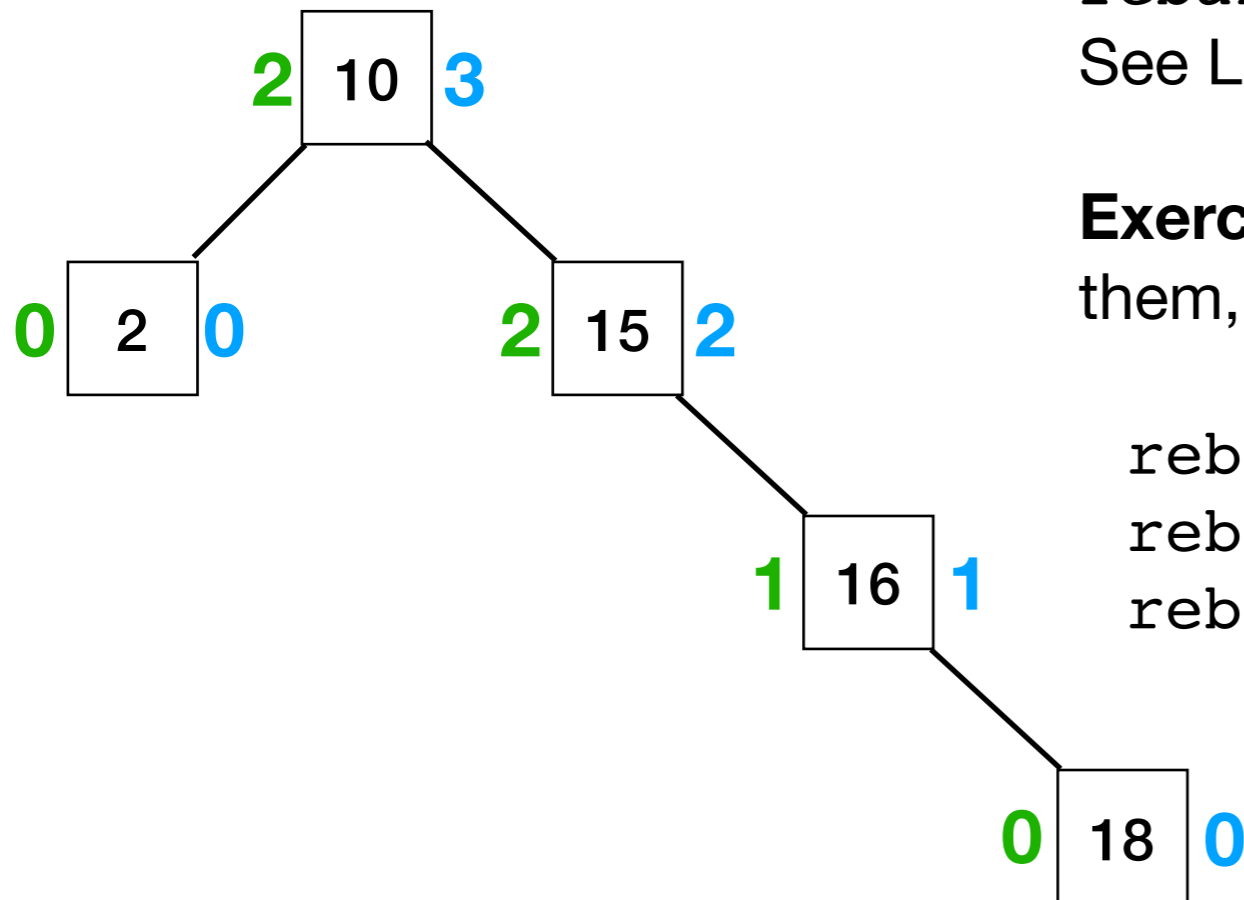
Exercise: What nodes have `rebalance` called on them, and in what order?



`avlInsert(root, 18)`

After the new node is inserted into the tree, the recursion will walk back up the tree, calling `rebalance` on each parent node in succession. See L12 slides for details.

Exercise: What nodes have `rebalance` called on them, and in what order?

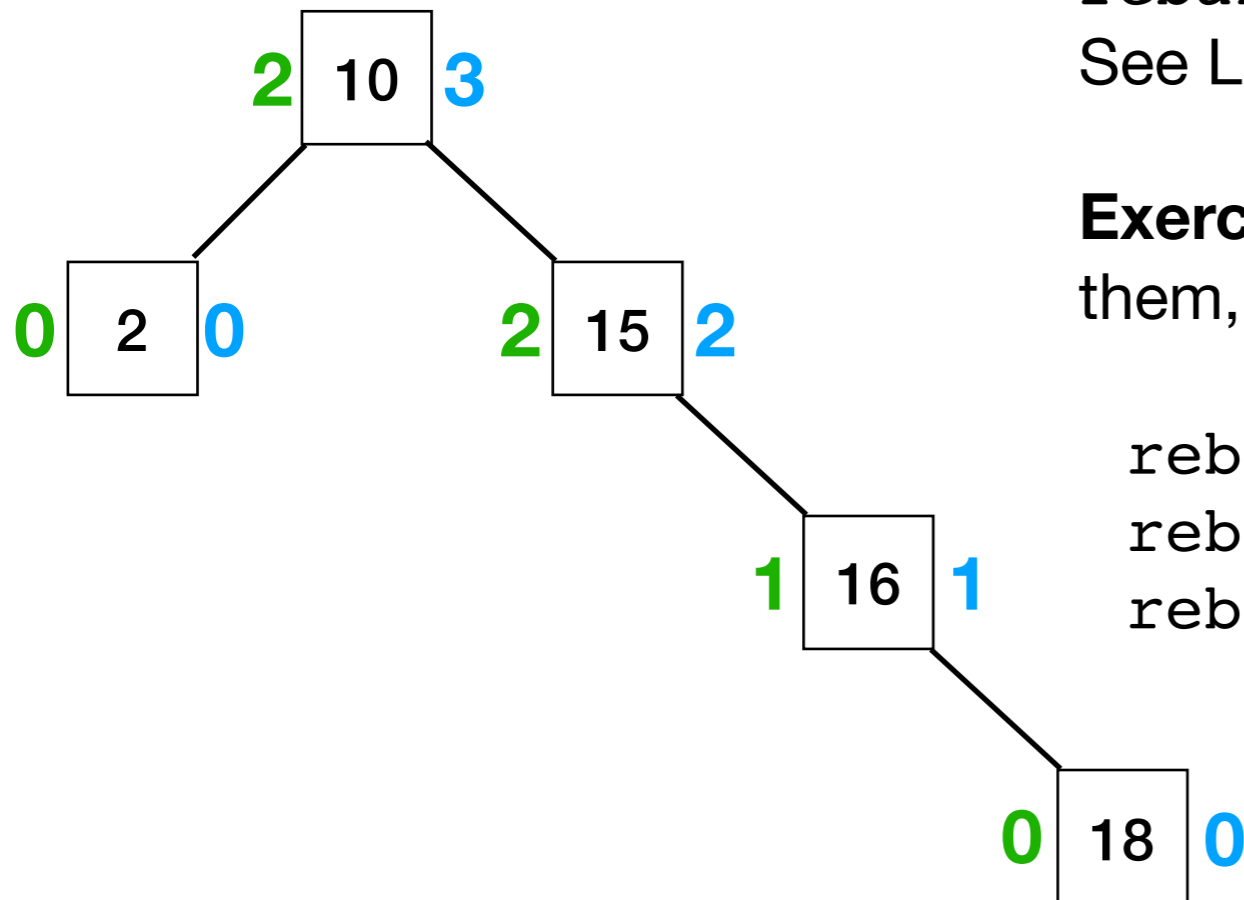


`rebalance(16)`
`rebalance(15)`
`rebalance(10)`

```
avlInsert(root, 18)
```

After the new node is inserted into the tree, the recursion will walk back up the tree, calling `rebalance` on each parent node in succession. See L12 slides for details.

Exercise: What nodes have `rebalance` called on them, and in what order?



```
rebalance(16)  
rebalance(15)  
rebalance(10)
```

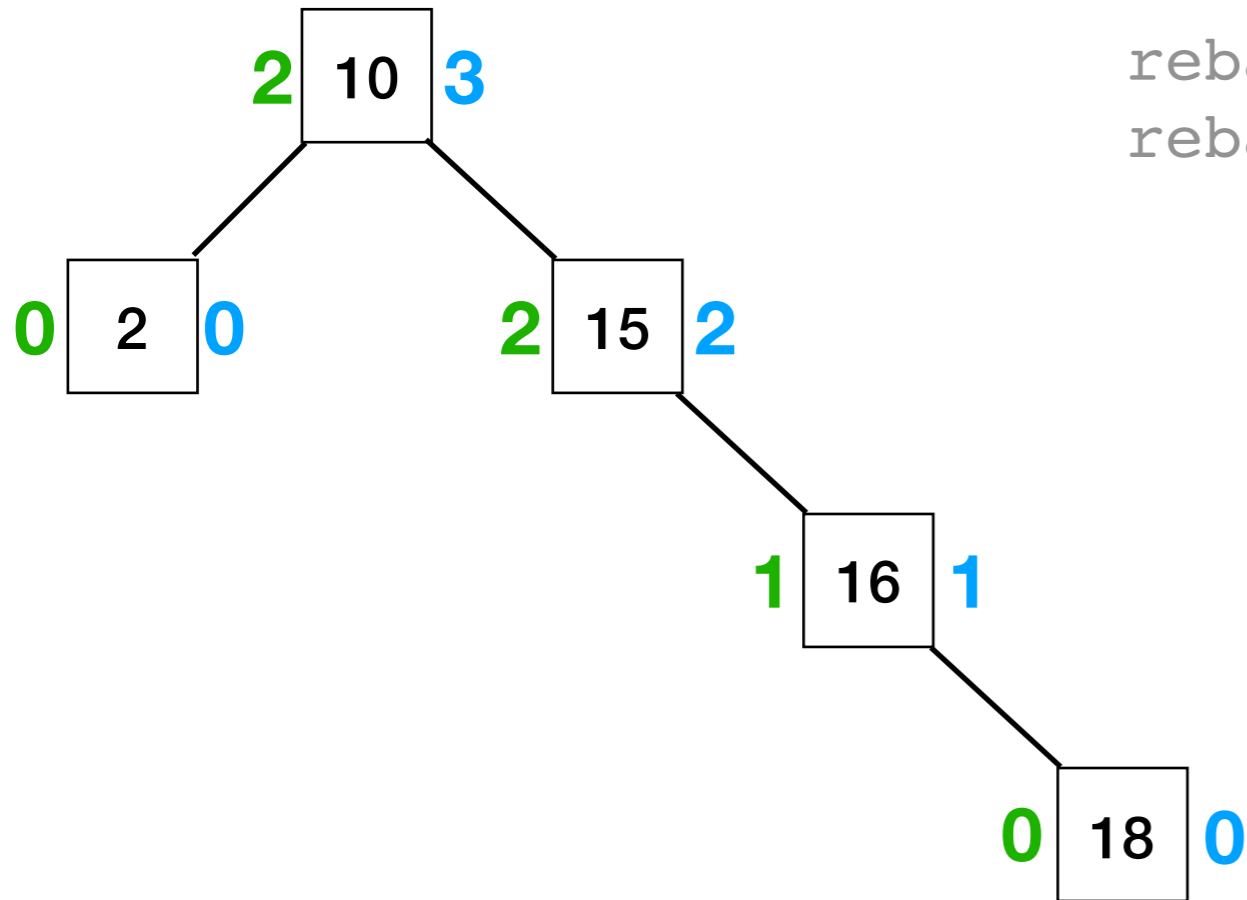
ok, let's execute this!

`avlInsert(root, 18)`

`rebalance(16)`

`rebalance(15)`

`rebalance(10)`



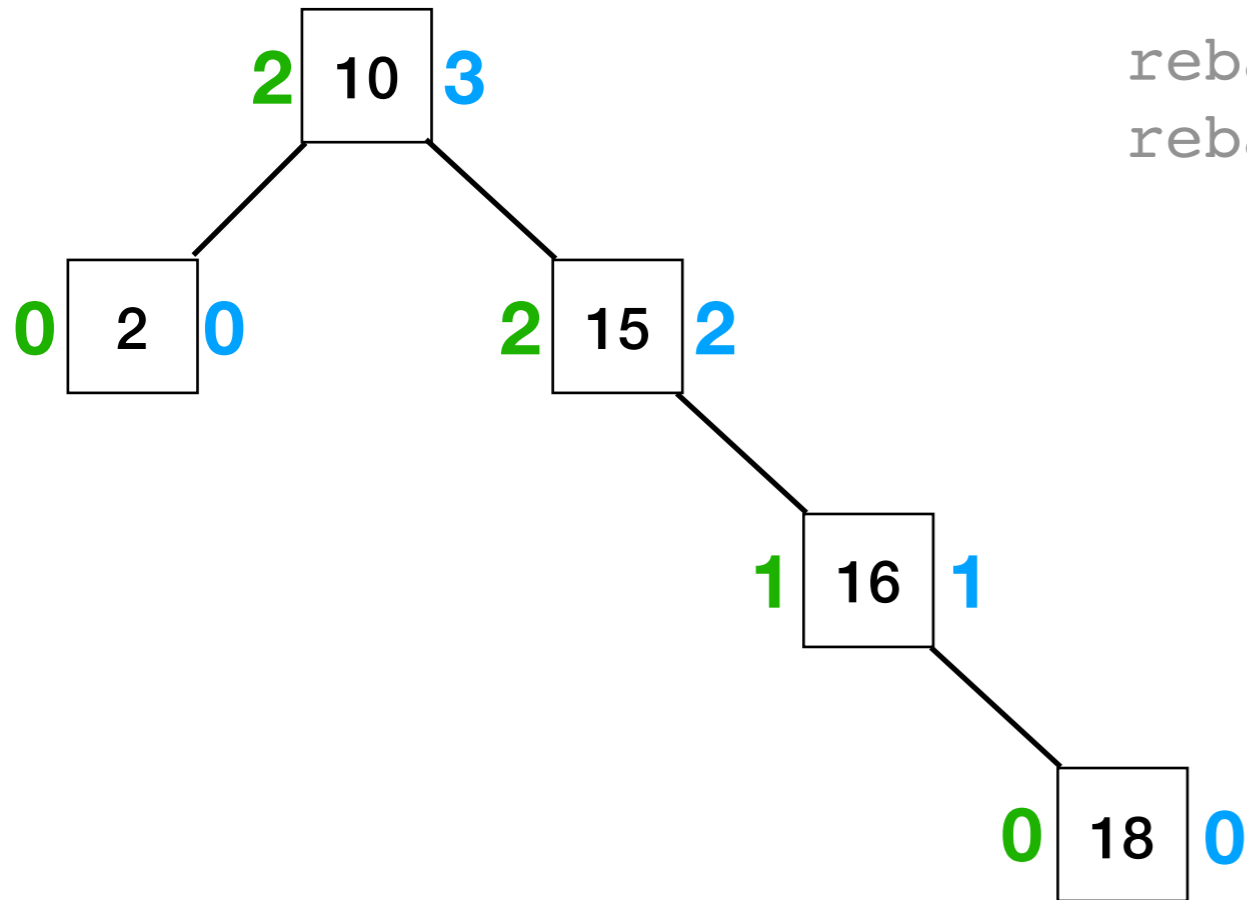
`avlInsert(root, 18)`

`rebalance(16)`

`=> bal(16) = 1; already balanced`

`rebalance(15)`

`rebalance(10)`



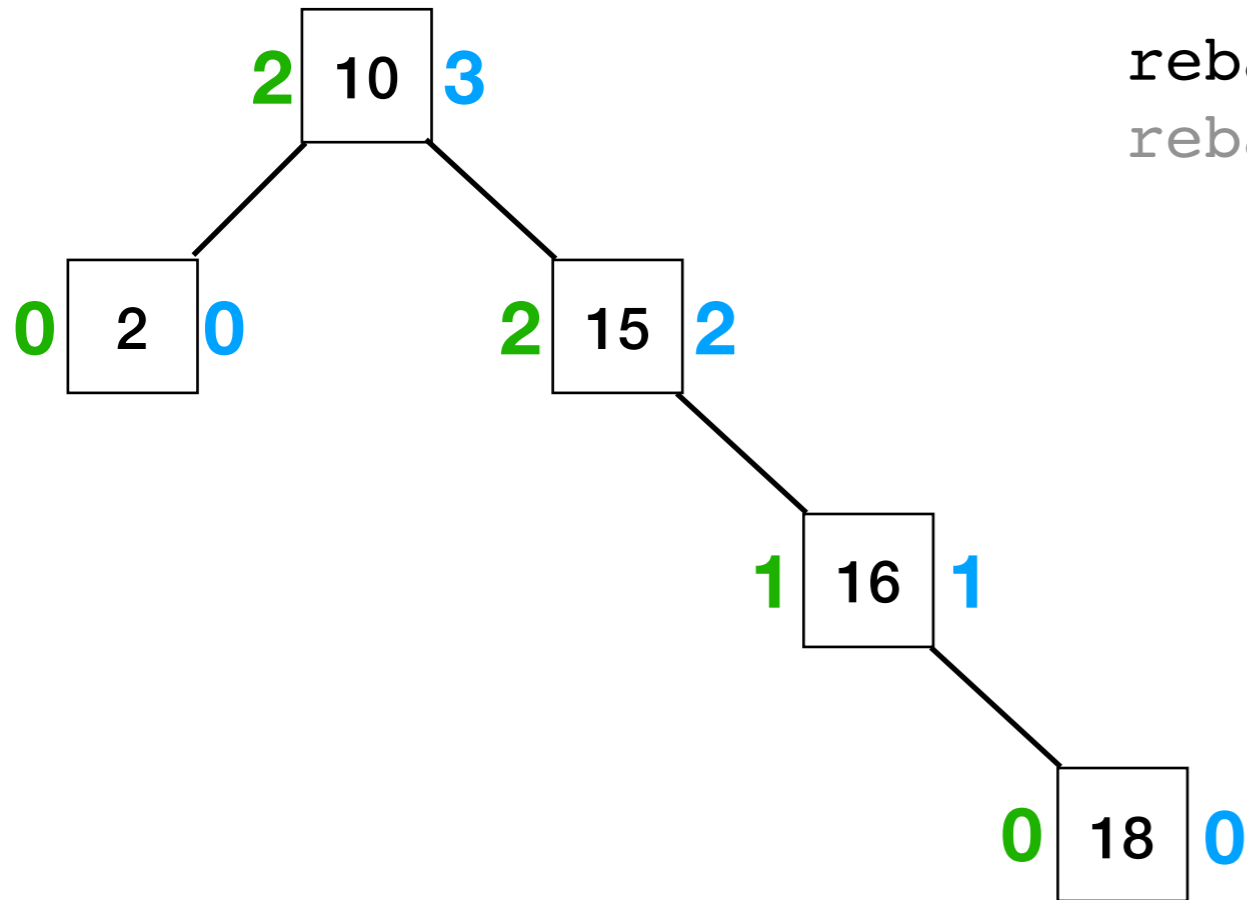
```
avlInsert(root, 18)
```

```
rebalance(16)
```

```
=> bal(16) = 1; already balanced
```

```
rebalance(15)
```

```
rebalance(10)
```



`avlInsert(root, 18)`

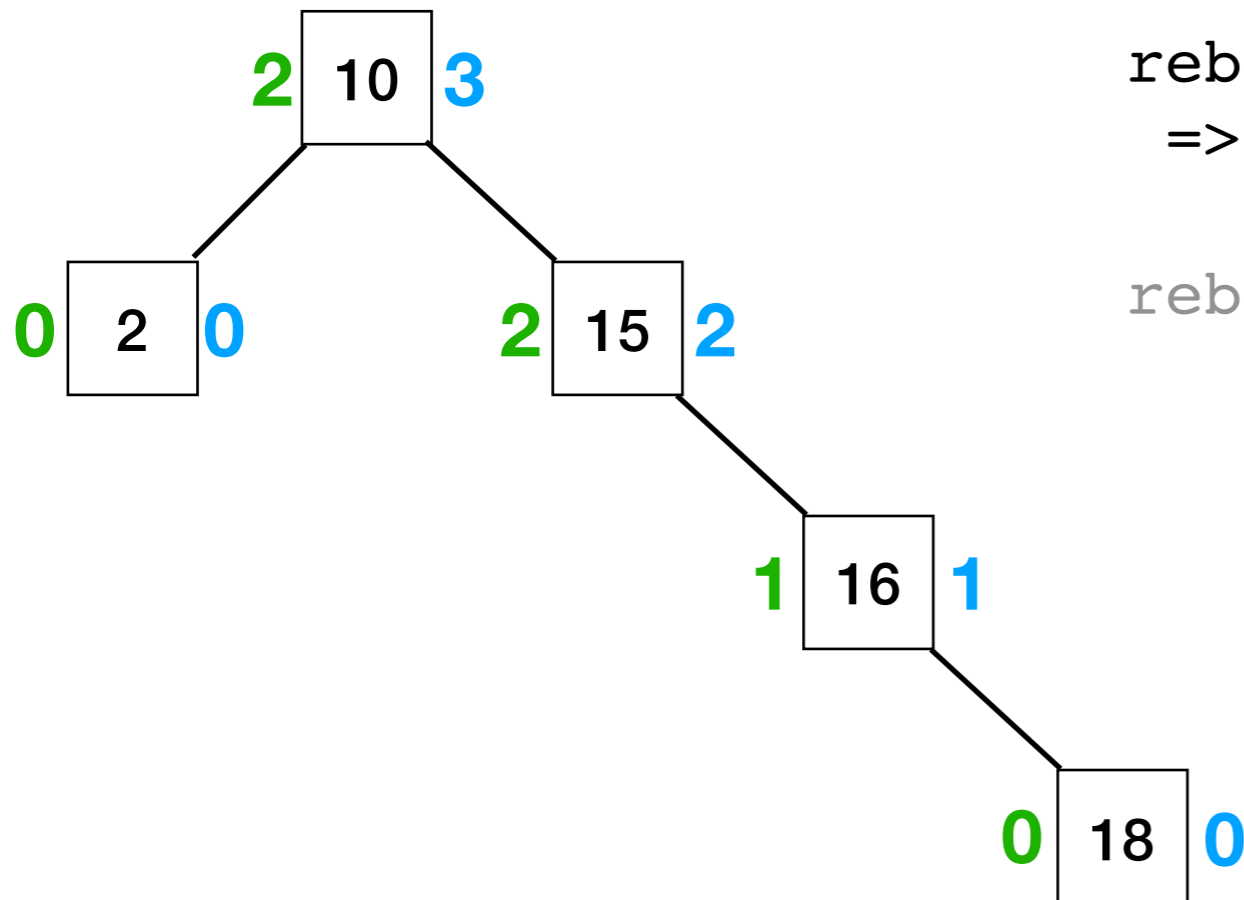
`rebalance(16)`

`=> bal(16) = 1; already balanced`

`rebalance(15)`

`=> bal(15) = 2; need to fix!`

`rebalance(10)`



```
avlInsert(root, 18)
```

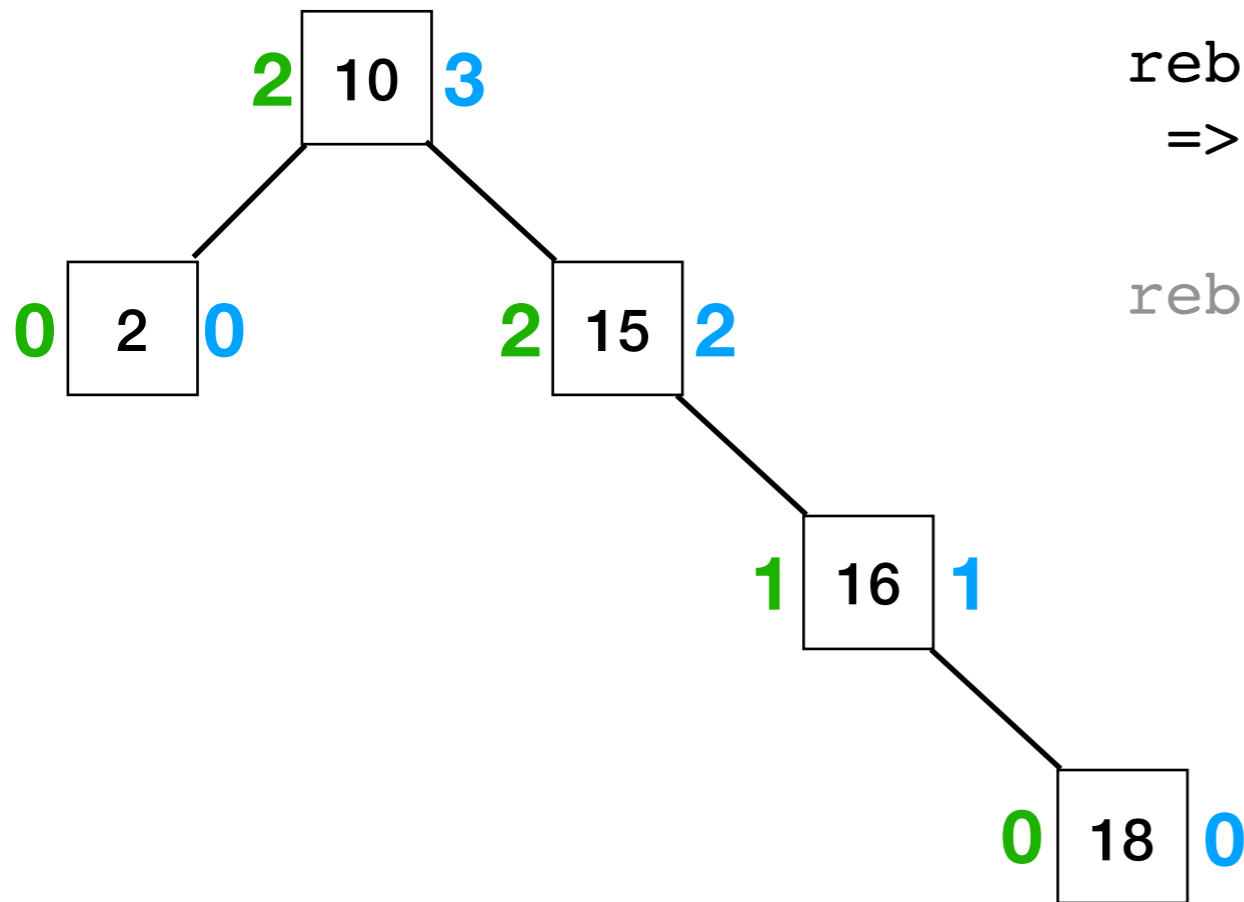
```
rebalance(16)
```

```
=> bal(16) = 1; already balanced
```

```
rebalance(15)
```

```
=> bal(15) = 2; need to fix!
```

```
rebalance(10)
```



Exercise: Step through the pseudocode for rebalance called on node 15. Which case (1, 2, 3, or 4) gets executed?

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

```
avlInsert(root, 18)
```

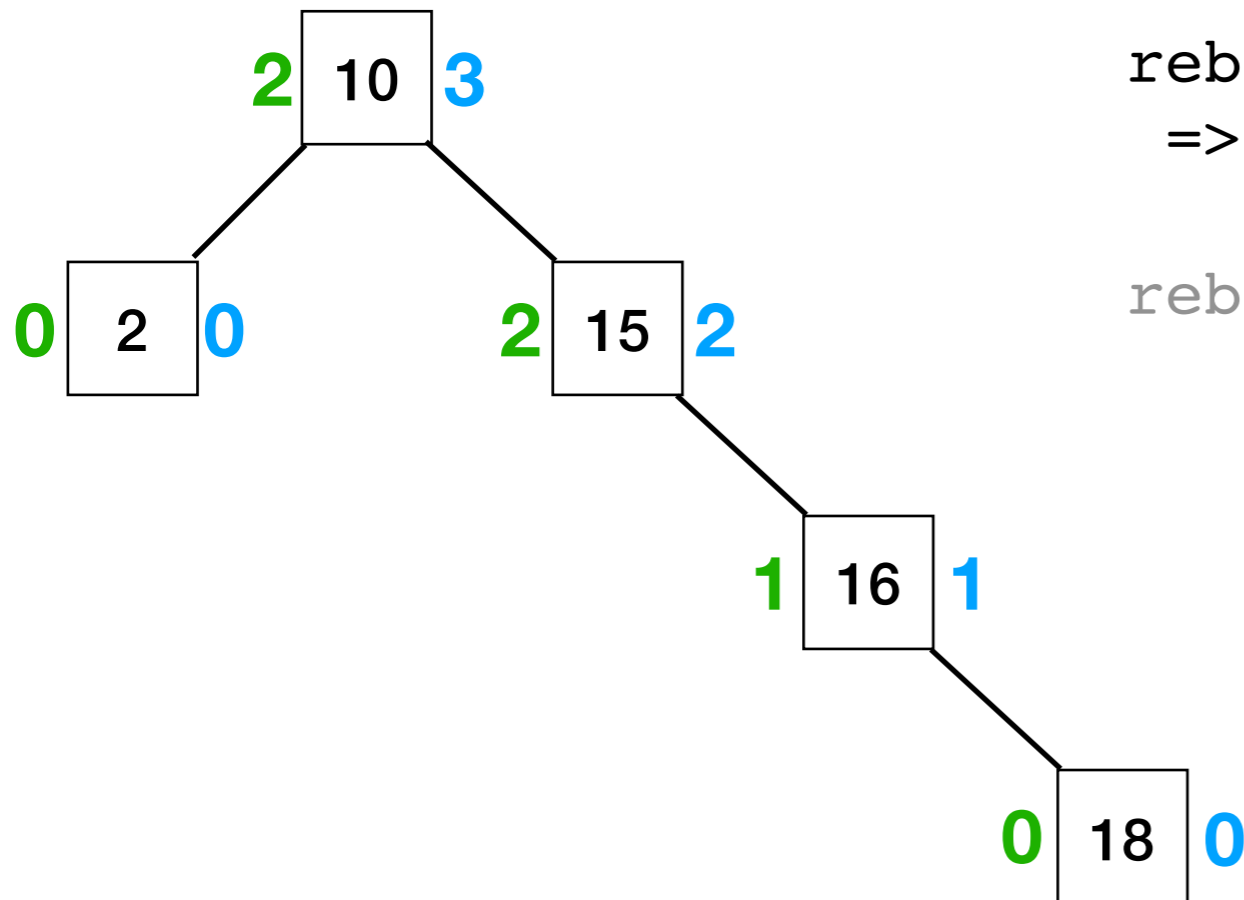
```
rebalance(16)
```

```
=> bal(16) = 1; already balanced
```

```
rebalance(15)
```

```
=> bal(15) = 2; need to fix!
```

```
rebalance(10)
```



Exercise: Step through the pseudocode for rebalance called on node 15. Which case (1, 2, 3, or 4) gets executed?

```
bal(15) > 0 (15 is R-heavy)
```

```
bal(15.right) > 0 (15's child is R-heavy)
```

```
=> Case 4 (RR)
```

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 18)

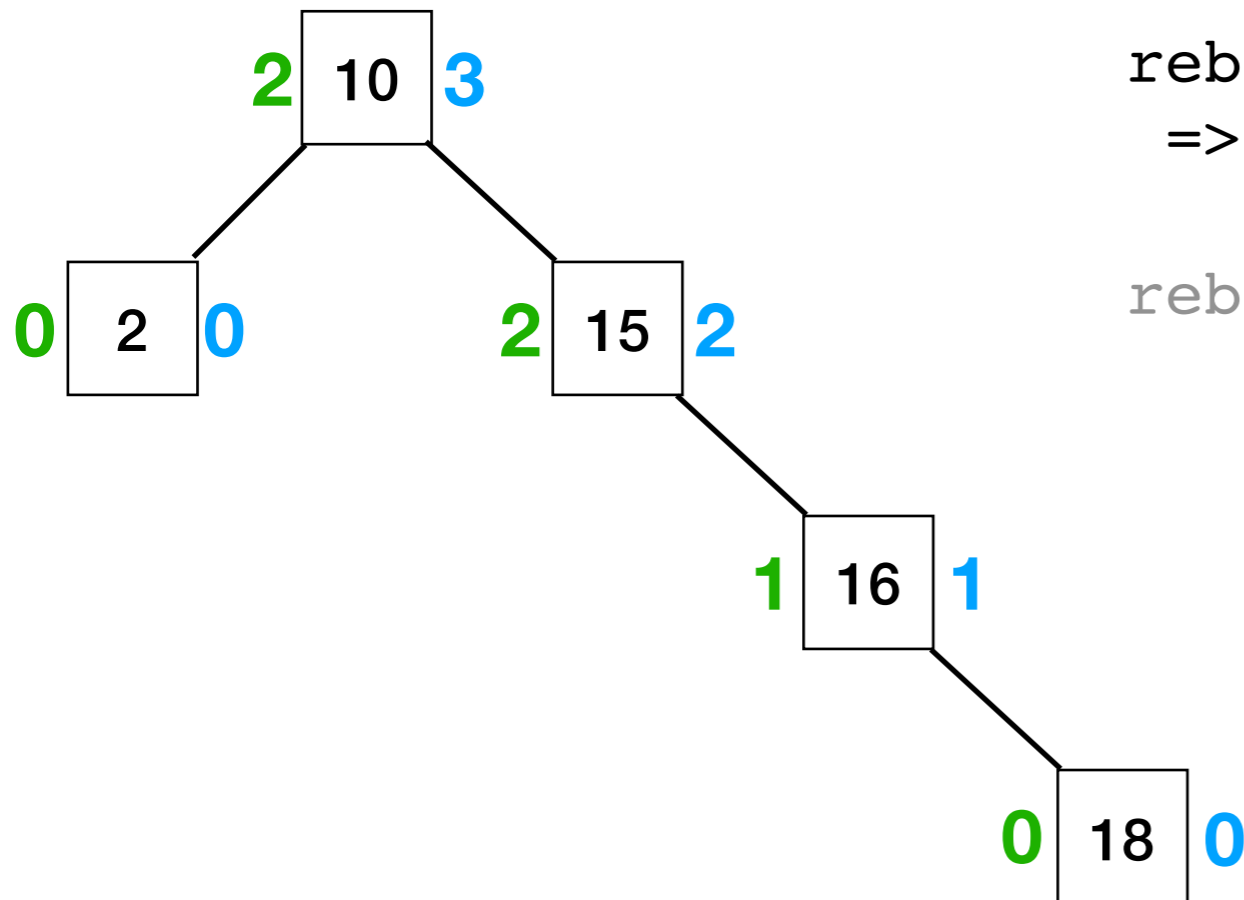
rebalance(16)

=> bal(16) = 1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

bal(15) > 0 (15 is R-heavy)

bal(15.right) > 0 (15's child is R-heavy)

=> **Case 4 (RR):** calls leftRotate(15)

avlInsert(root, 18)

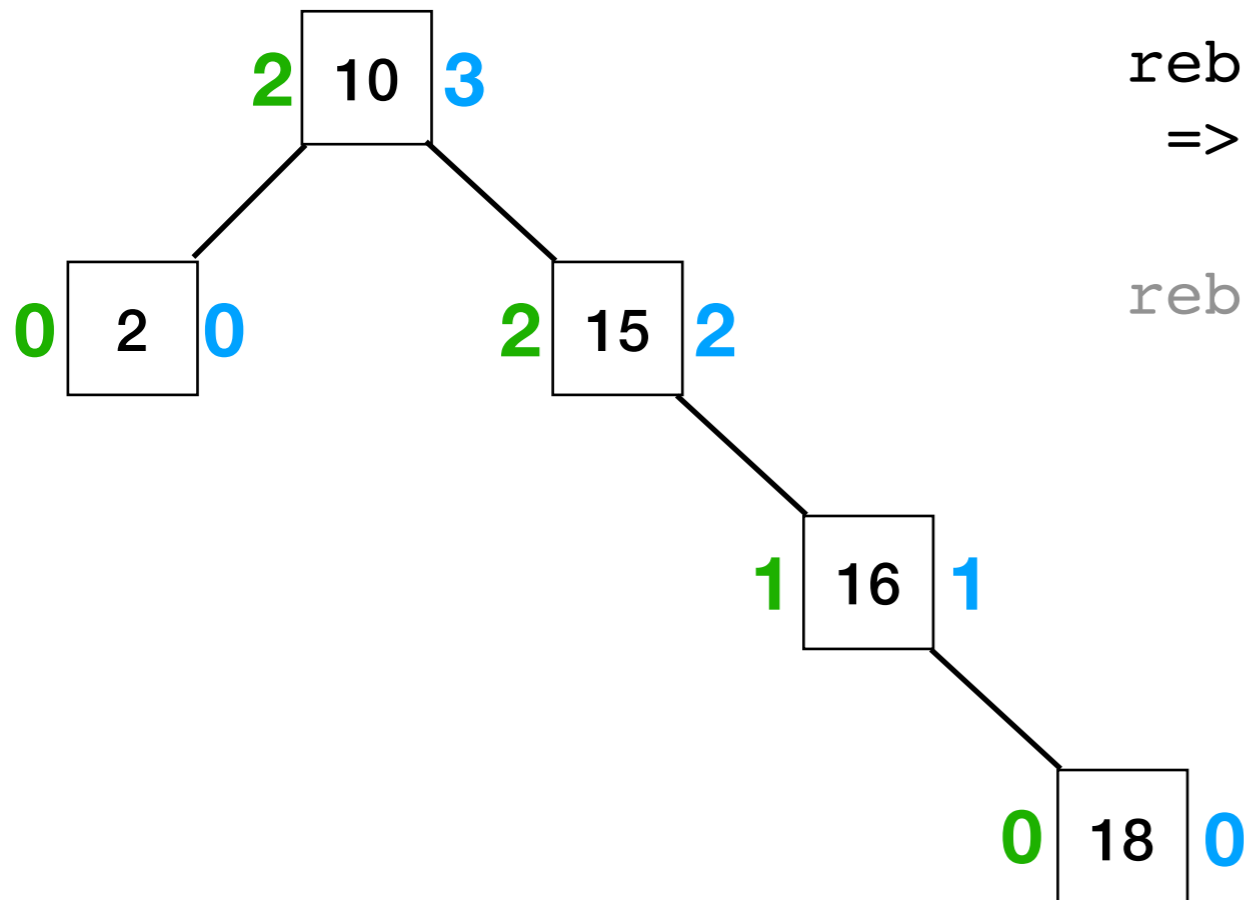
rebalance(16)

=> bal(16) = 1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Draw the tree after a left rotation on 15.

=> **Case 4 (RR):** calls leftRotate(15)

```
void rebalance(n):
    if bal(n) < -1:
        if bal(n.left) < 0
            // case 1:
            // rightRot(n)
        else:
            // case 2:
            // leftRot(n.L);
            // rightRot(n)
    else if bal(n) > 1:
        if bal(n.right) < 0:
            // case 3:
            // rightRot(n.R);
            // leftRot(n)
        else:
            // case 4:
            // leftRot(n)
```

```
avlInsert(root, 18)
```

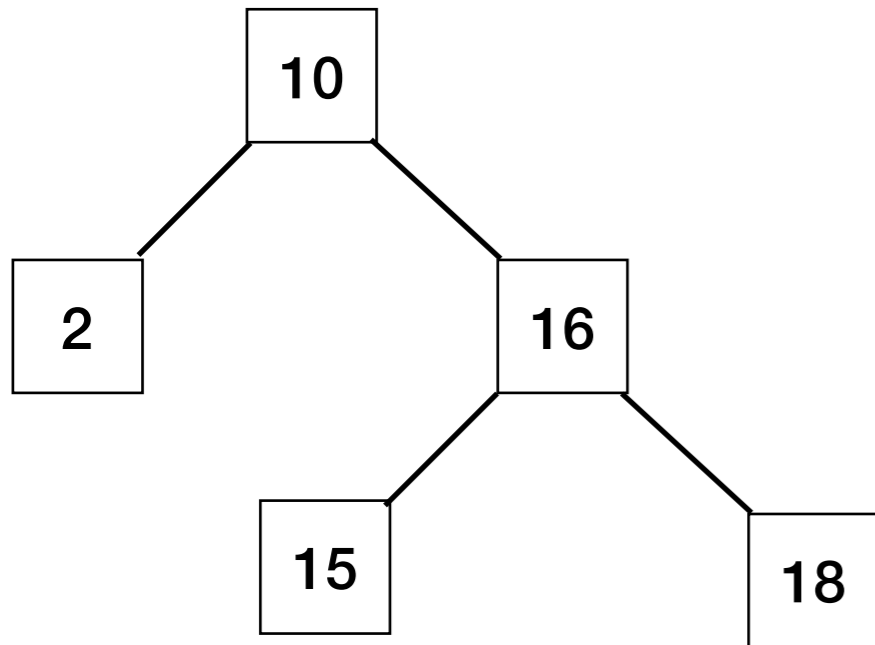
```
rebalance(16)
```

```
=> bal(16) = 1; already balanced
```

```
rebalance(15)
```

```
=> bal(15) = 2; need to fix!
```

```
rebalance(10)
```



Exercise: Draw the tree after a left rotation on 15.

```
=> Case 4 (RR): calls leftRotate(15)
```

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```


avlInsert(root, 18)

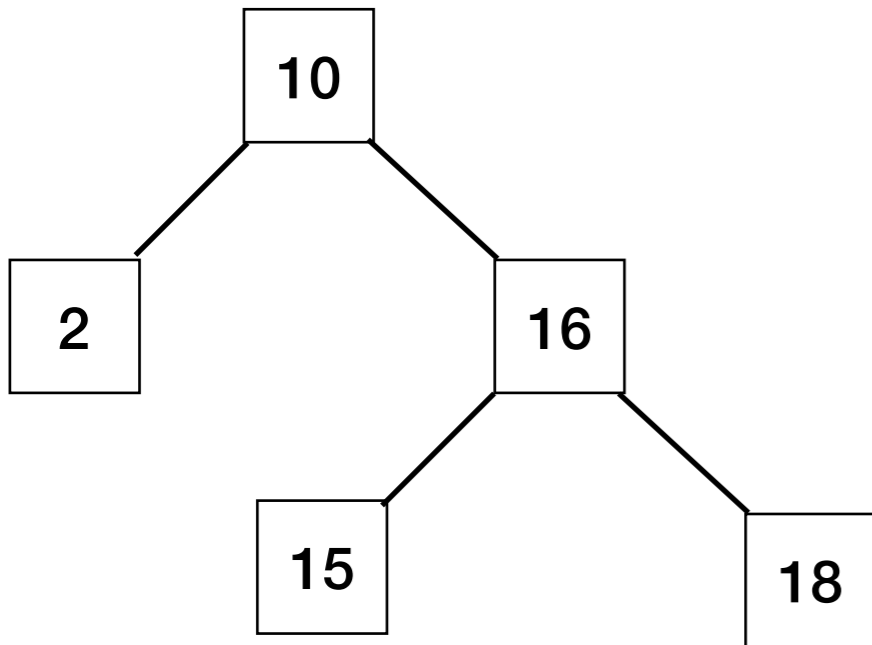
rebalance(16)

=> bal(16) = 1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Recompute heights and balance factors.

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 18)

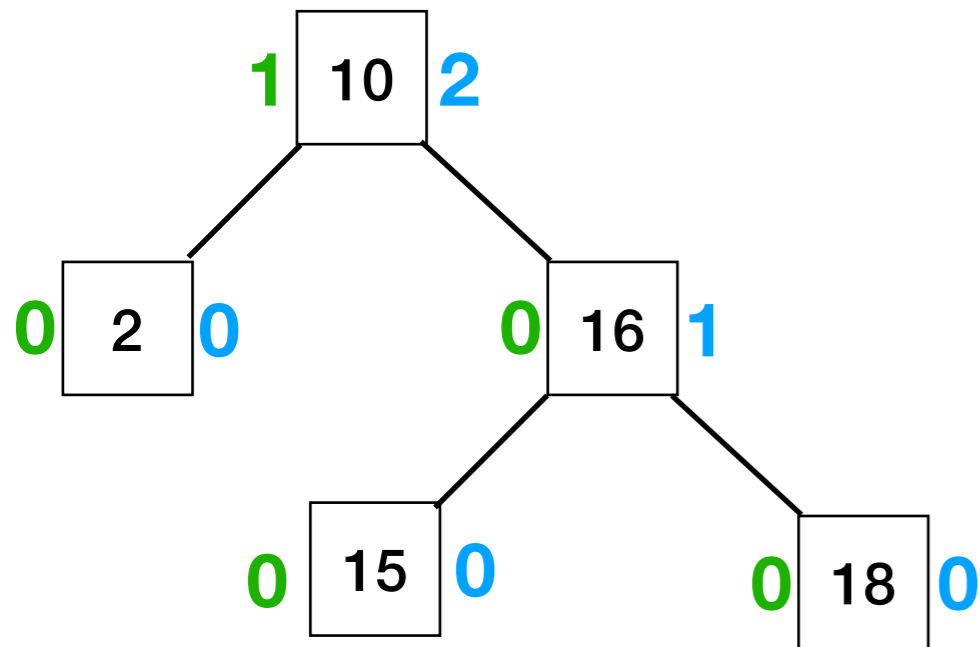
rebalance(16)

=> bal(16) = 1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



We're not done - one more rebalance operation!

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 18)

rebalance(16)

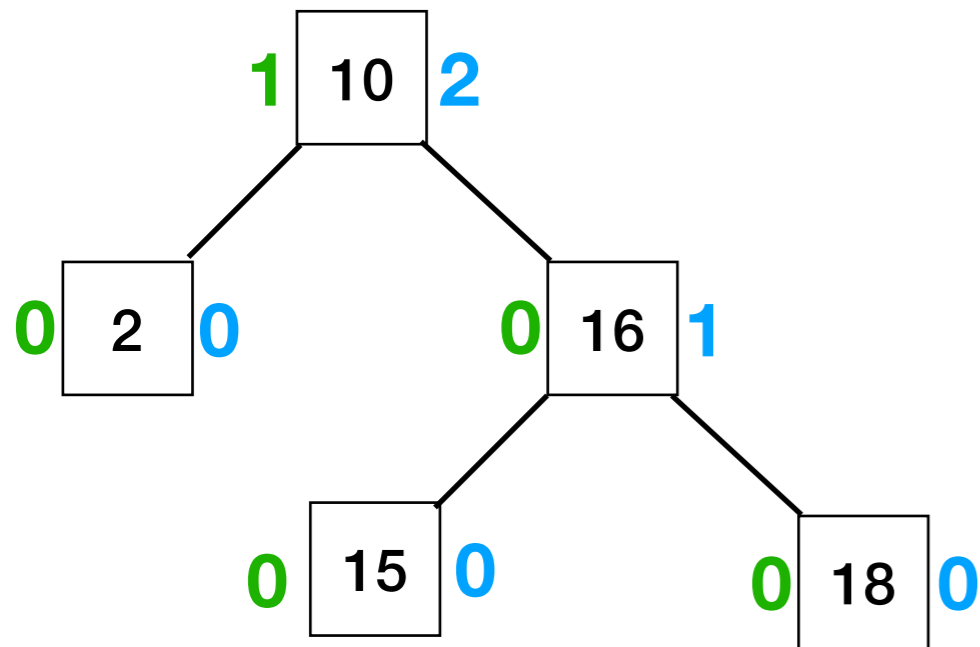
=> bal(16) = 1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)

=> bal(10) = 1; already balanced



```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 18)

rebalance(16)

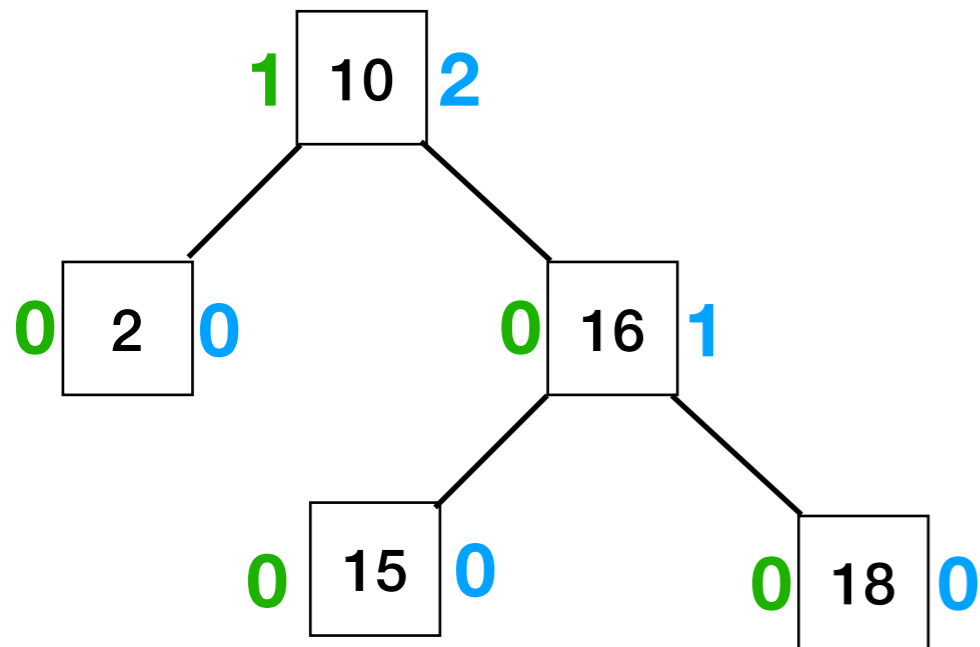
=> bal(16) = 1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)

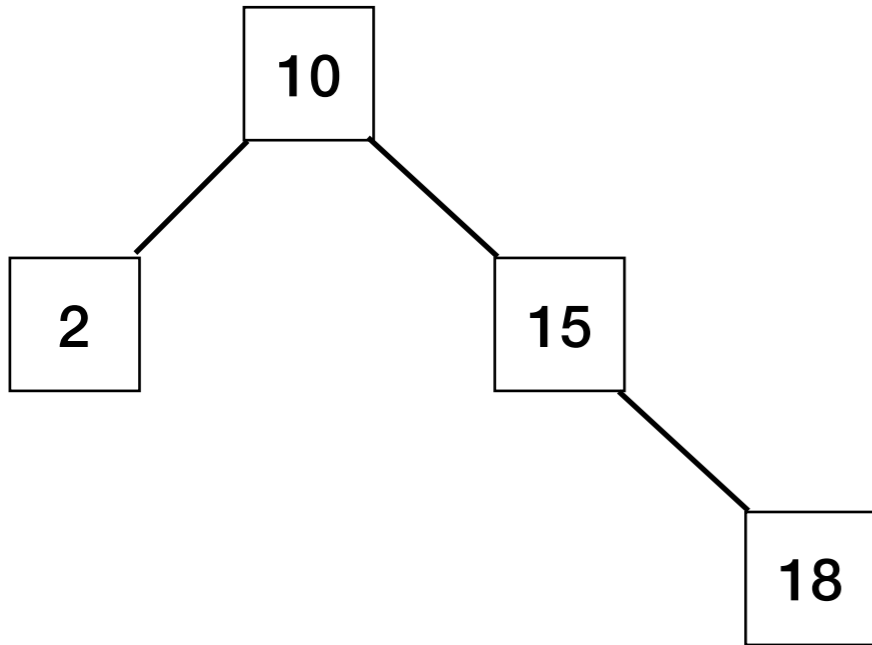
=> bal(10) = 1; already balanced



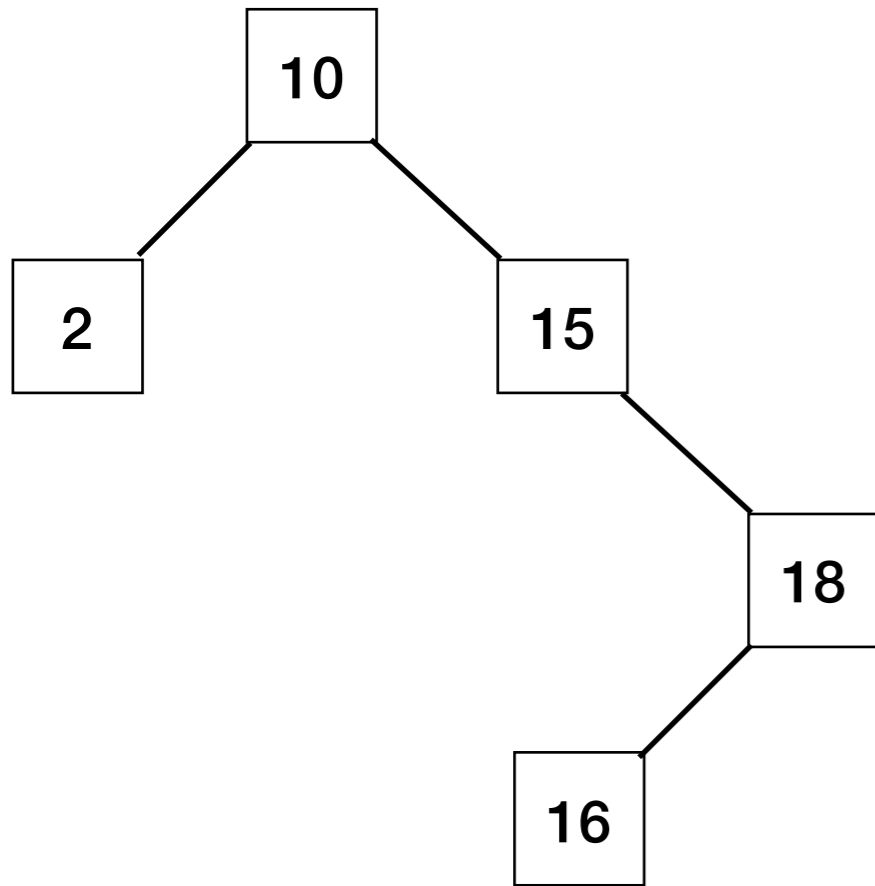
```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

Ok, now we're done. Our tree is balanced!

A different case

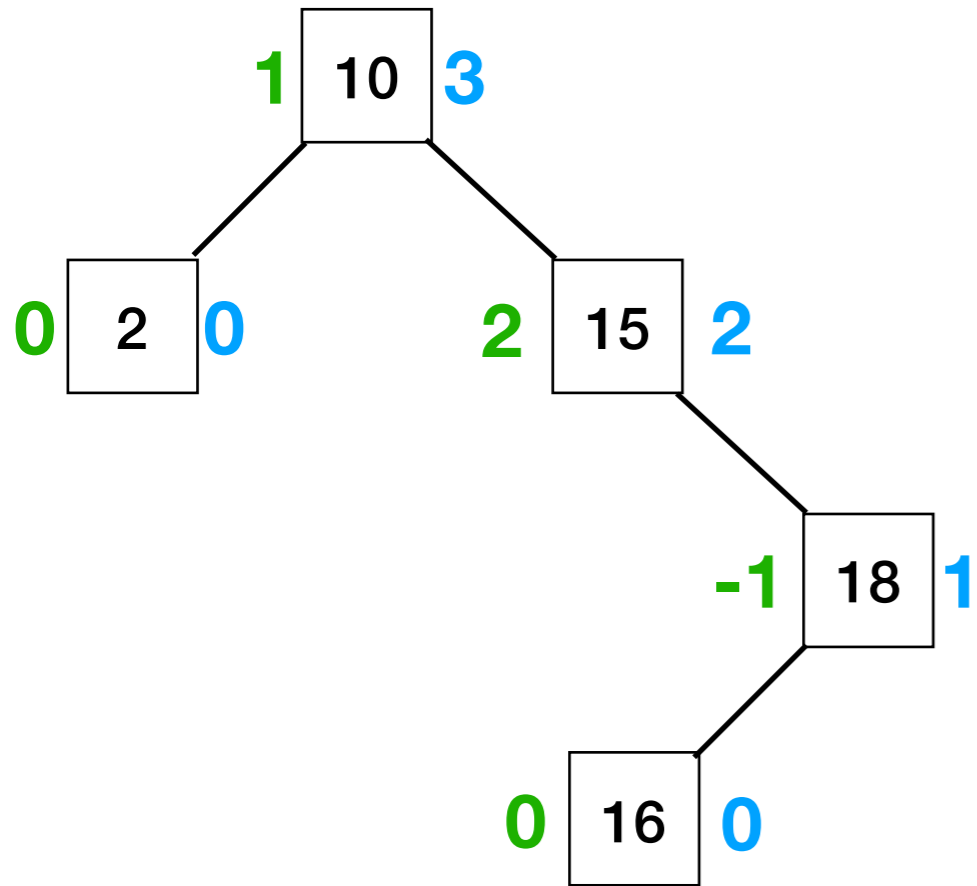


`avlInsert(root, 16)`



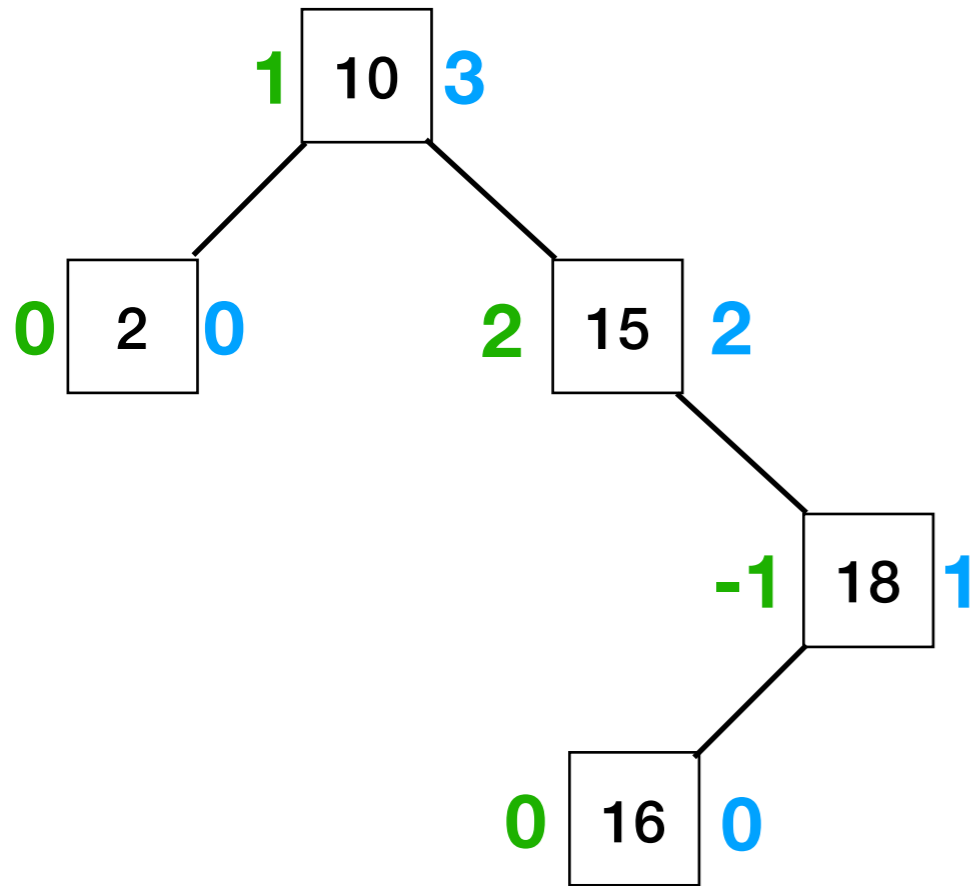
Exercise: Compute heights and balance factors.

avlInsert(root, 16)



Exercise: Compute heights and balance factors.

avlInsert(root, 16)



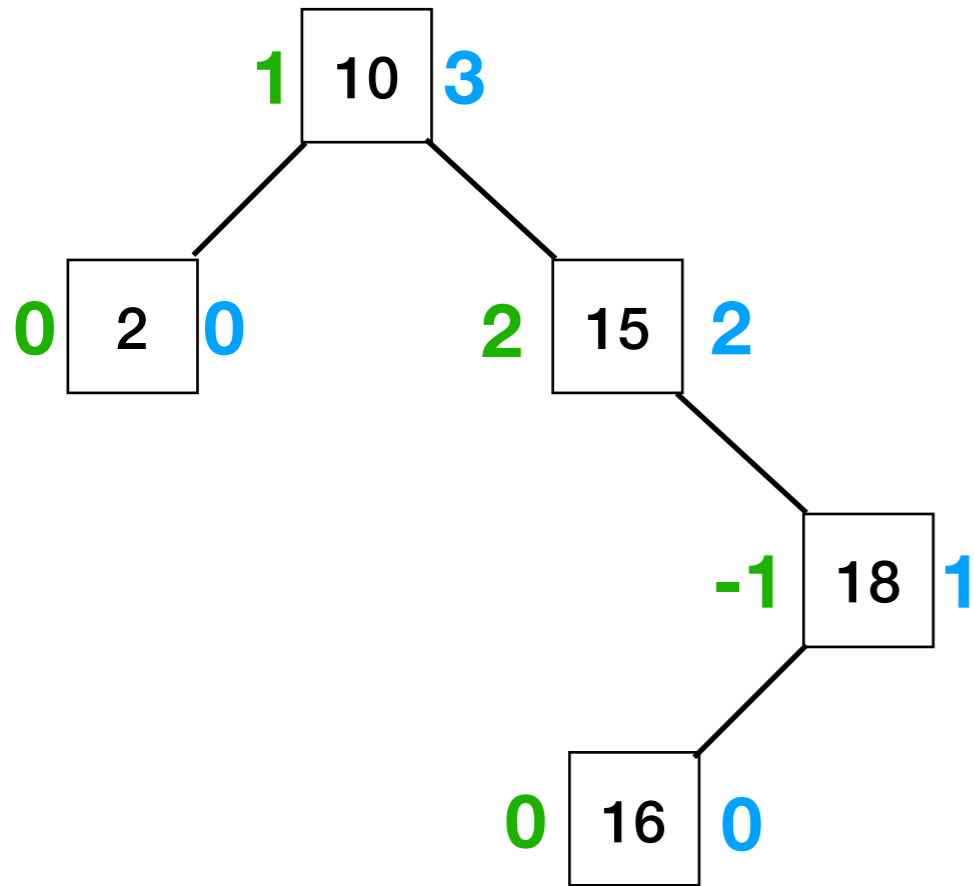
Exercise: Write the sequence of rebalance calls.

`avlInsert(root, 16)`

`rebalance(18)`

`rebalance(15)`

`rebalance(10)`



Exercise: Write the sequence of rebalance calls.

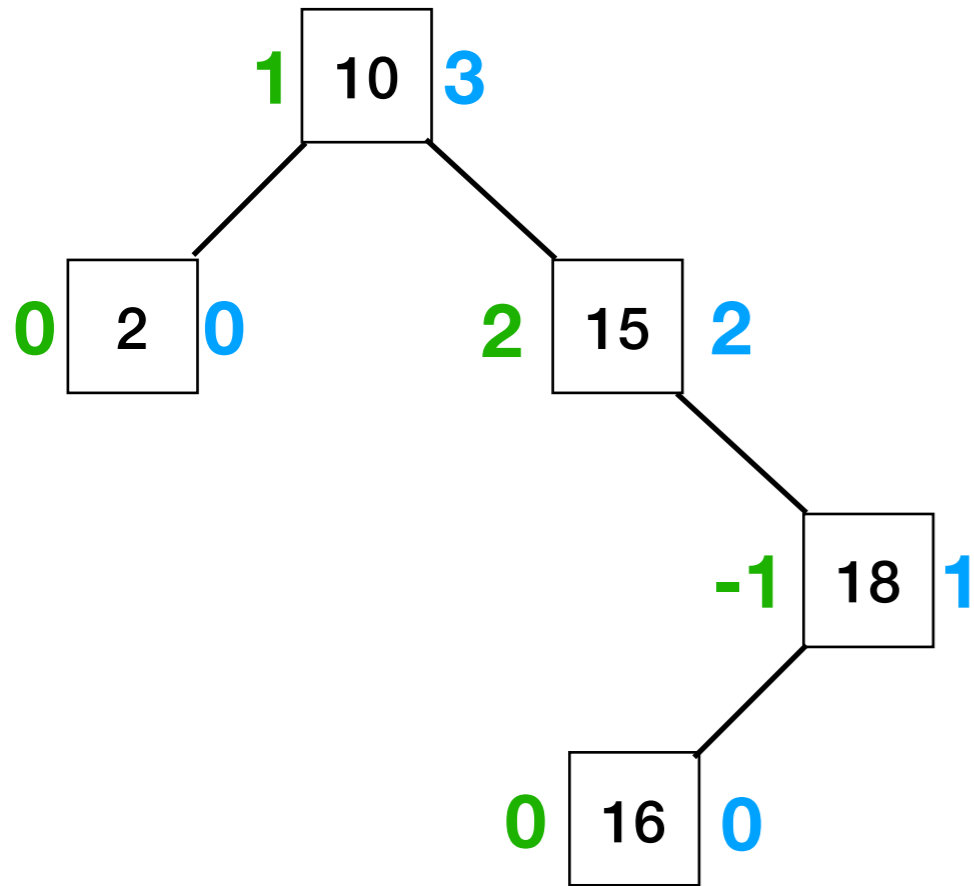
```
avlInsert(root, 16)
```

```
rebalance(18)
```

```
=> bal(18) = -1; already balanced
```

```
rebalance(15)
```

```
rebalance(10)
```



avlInsert(root, 16)

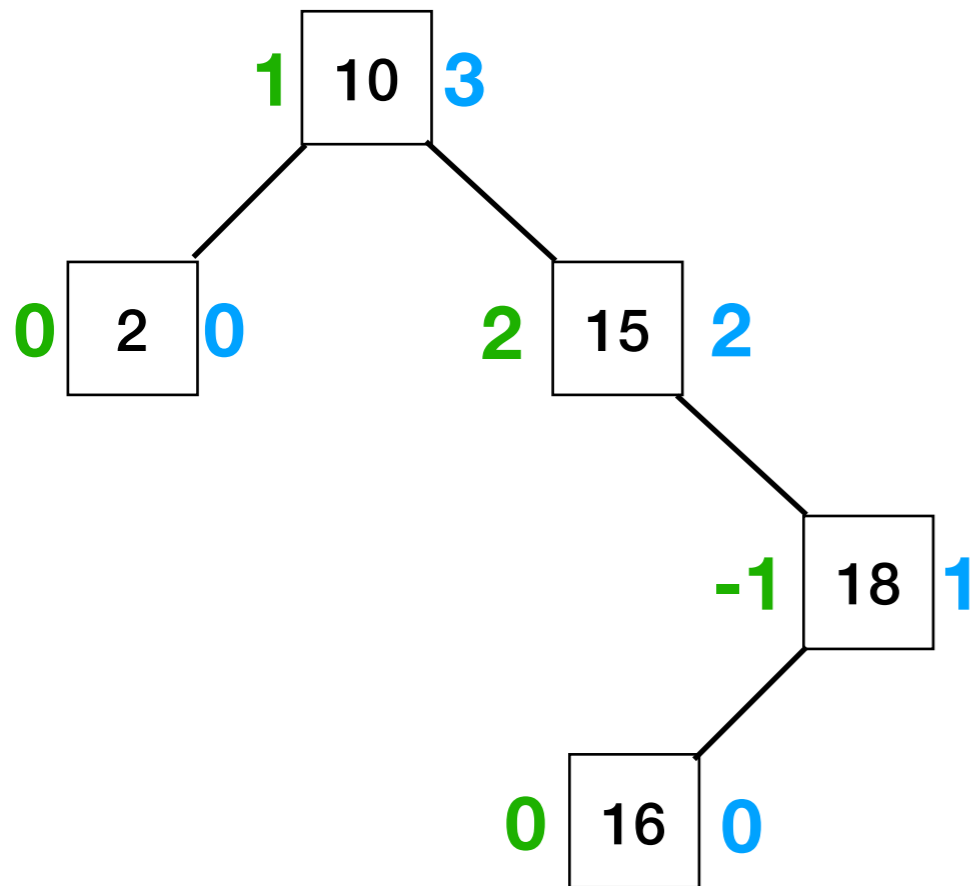
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Which case applies here?

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

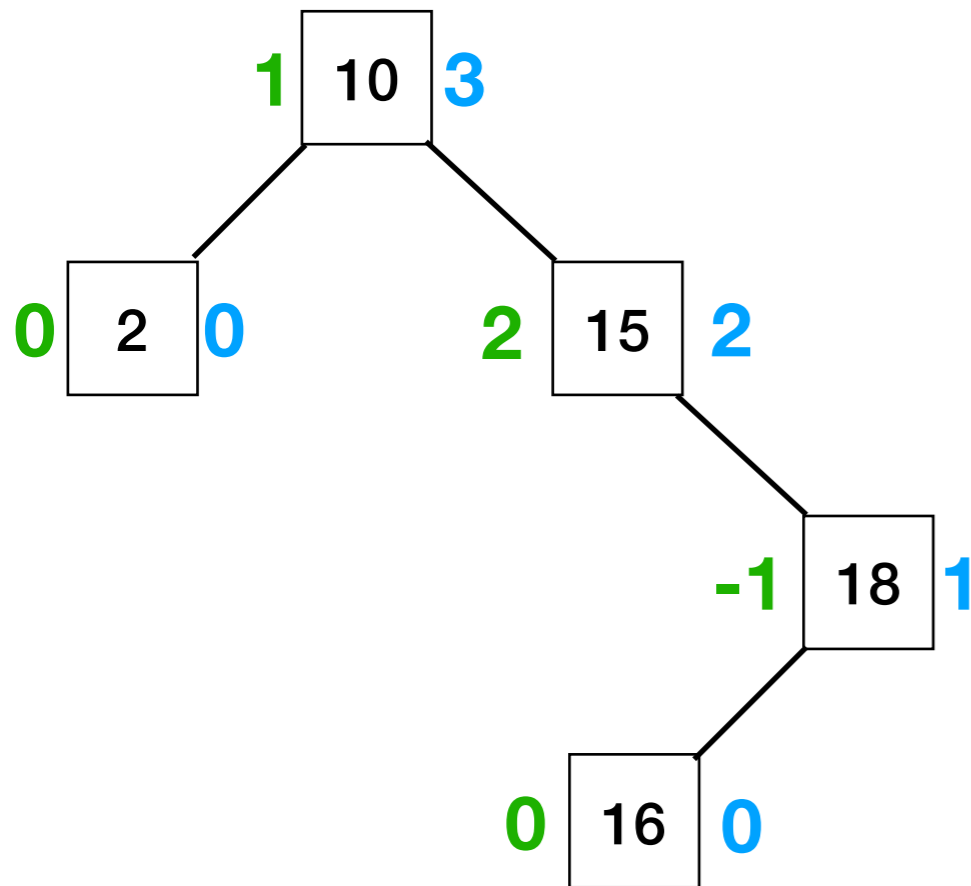
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Which case applies here?

bal(15) > 0 (15 is R-heavy)

bal(15.right) < 0 (15's child is L-heavy)

=> **Case 3 (RL):**

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

```
avlInsert(root, 16)
```

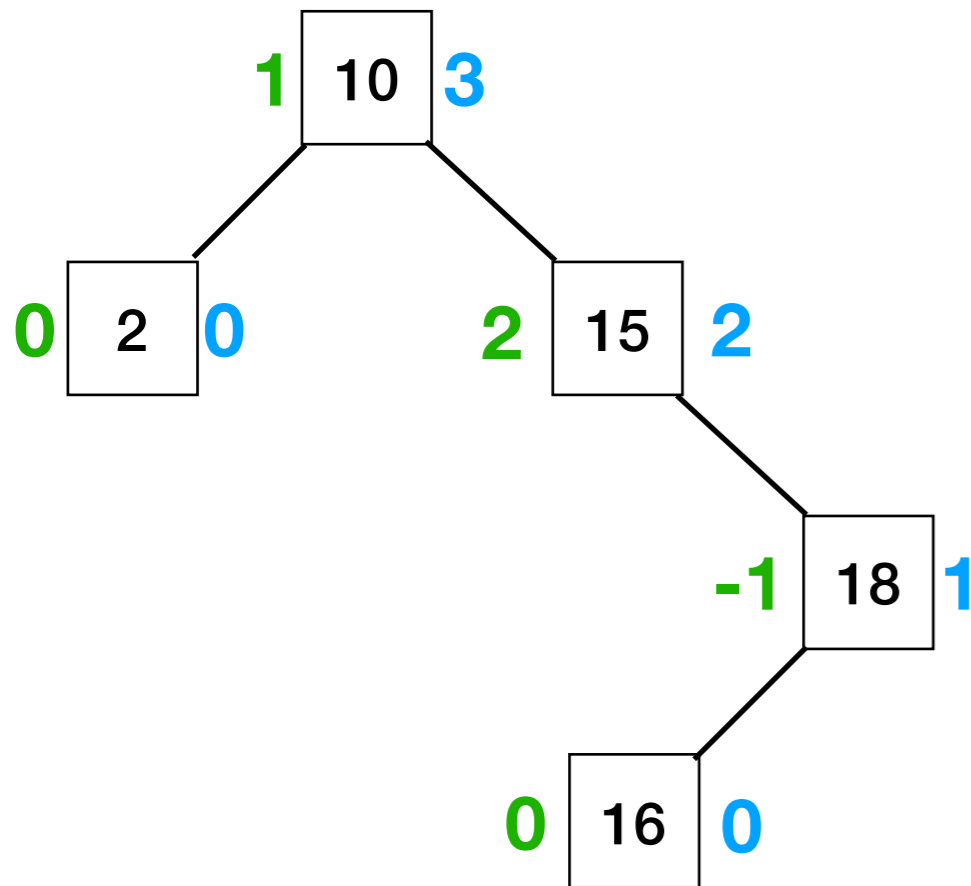
```
rebalance(18)
```

```
=> bal(18) = -1; already balanced
```

```
rebalance(15)
```

```
=> bal(15) = 2; need to fix!
```

```
rebalance(10)
```



Exercise: Which case applies here?

```
bal(15) > 0 (15 is R-heavy)
```

```
bal(15.right) < 0 (15's child is L-heavy)
```

```
=> Case 3 (RL):
```

```
rightRotate(18)
```

```
leftRotate(15)
```

```
void rebalance(n):
```

```
  if bal(n) < -1:
```

```
    if bal(n.left) < 0
```

```
      // case 1:
```

```
      // rightRot(n)
```

```
    else:
```

```
      // case 2:
```

```
      // leftRot(n.L);
```

```
      // rightRot(n)
```

```
  else if bal(n) > 1:
```

```
    if bal(n.right) < 0:
```

```
      // case 3:
```

```
      // rightRot(n.R);
```

```
      // leftRot(n)
```

```
    else:
```

```
      // case 4:
```

```
      // leftRot(n)
```

avlInsert(root, 16)

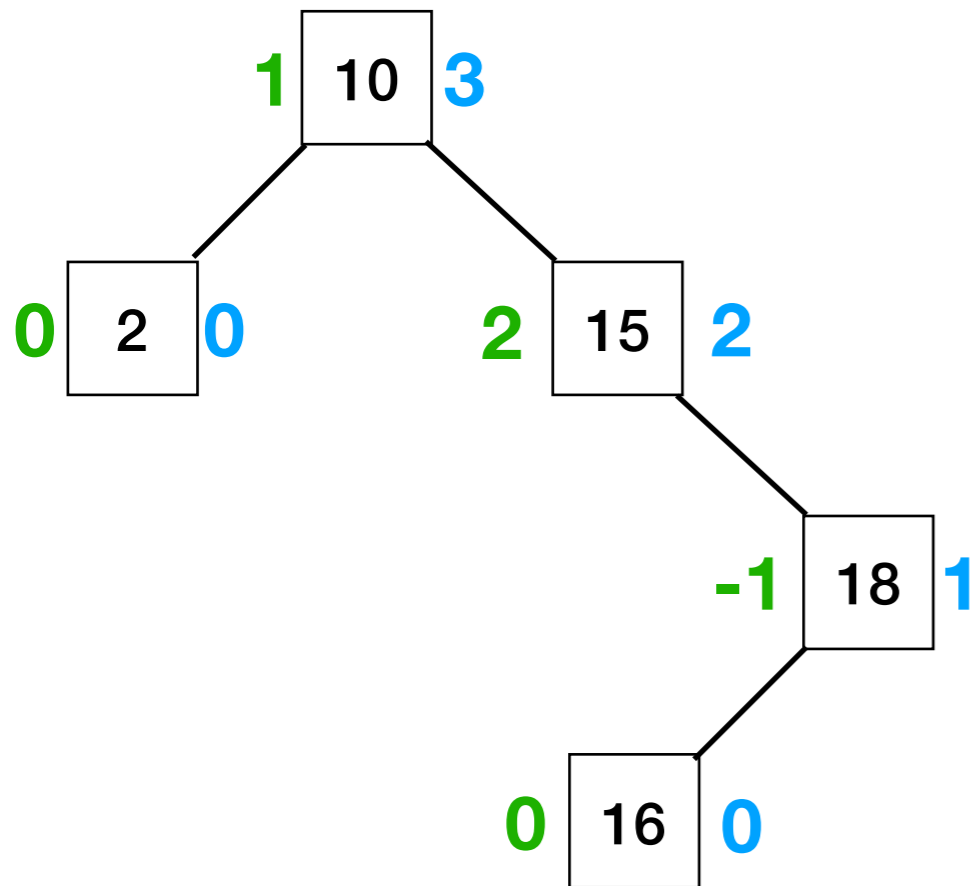
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Which case applies here?

bal(15) > 0 (15 is R-heavy)

bal(15.right) < 0 (15's child is L-heavy)

=> **Case 3 (RL):**

rightRotate(18)

leftRotate(15)

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

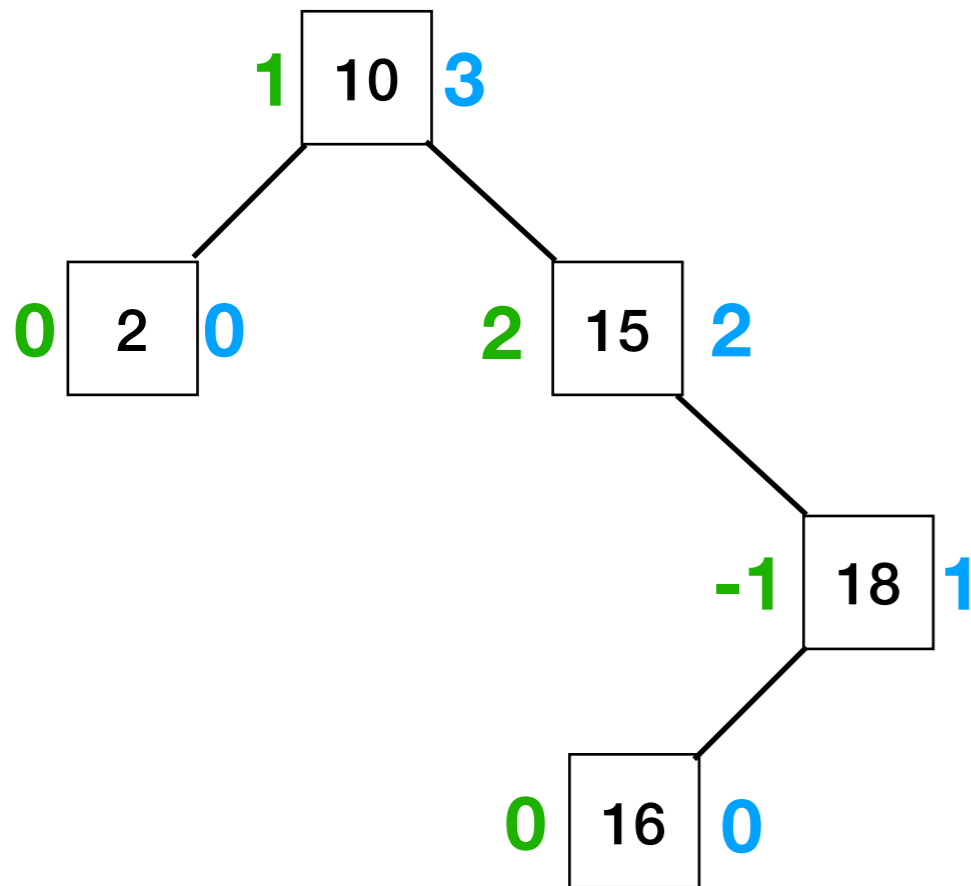
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Draw the tree after rightRotate(18).

=> **Case 3 (RL):**

rightRotate(18)

leftRotate(15)

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

```
avlInsert(root, 16)
```

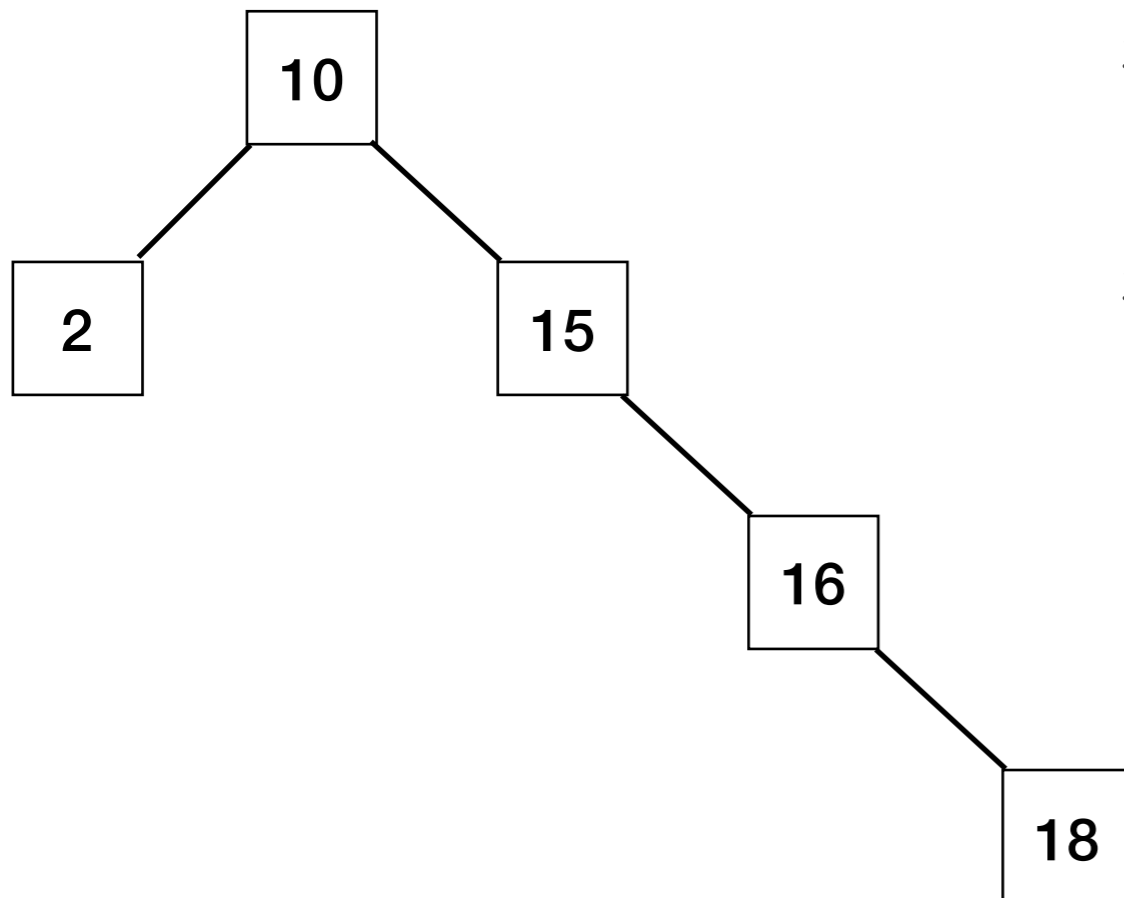
```
rebalance(18)
```

```
=> bal(18) = -1; already balanced
```

```
rebalance(15)
```

```
=> bal(15) = 2; need to fix!
```

```
rebalance(10)
```



Exercise: Draw the tree after rightRotate(18).

=> **Case 3 (RL):**

```
rightRotate(18)
```

```
leftRotate(15)
```

```
void rebalance(n):
```

```
  if bal(n) < -1:
```

```
    if bal(n.left) < 0
```

```
      // case 1:
```

```
      // rightRot(n)
```

```
    else:
```

```
      // case 2:
```

```
      // leftRot(n.L);
```

```
      // rightRot(n)
```

```
  else if bal(n) > 1:
```

```
    if bal(n.right) < 0:
```

```
      // case 3:
```

```
      // rightRot(n.R);
```

```
      // leftRot(n)
```

```
    else:
```

```
      // case 4:
```

```
      // leftRot(n)
```


avlInsert(root, 16)

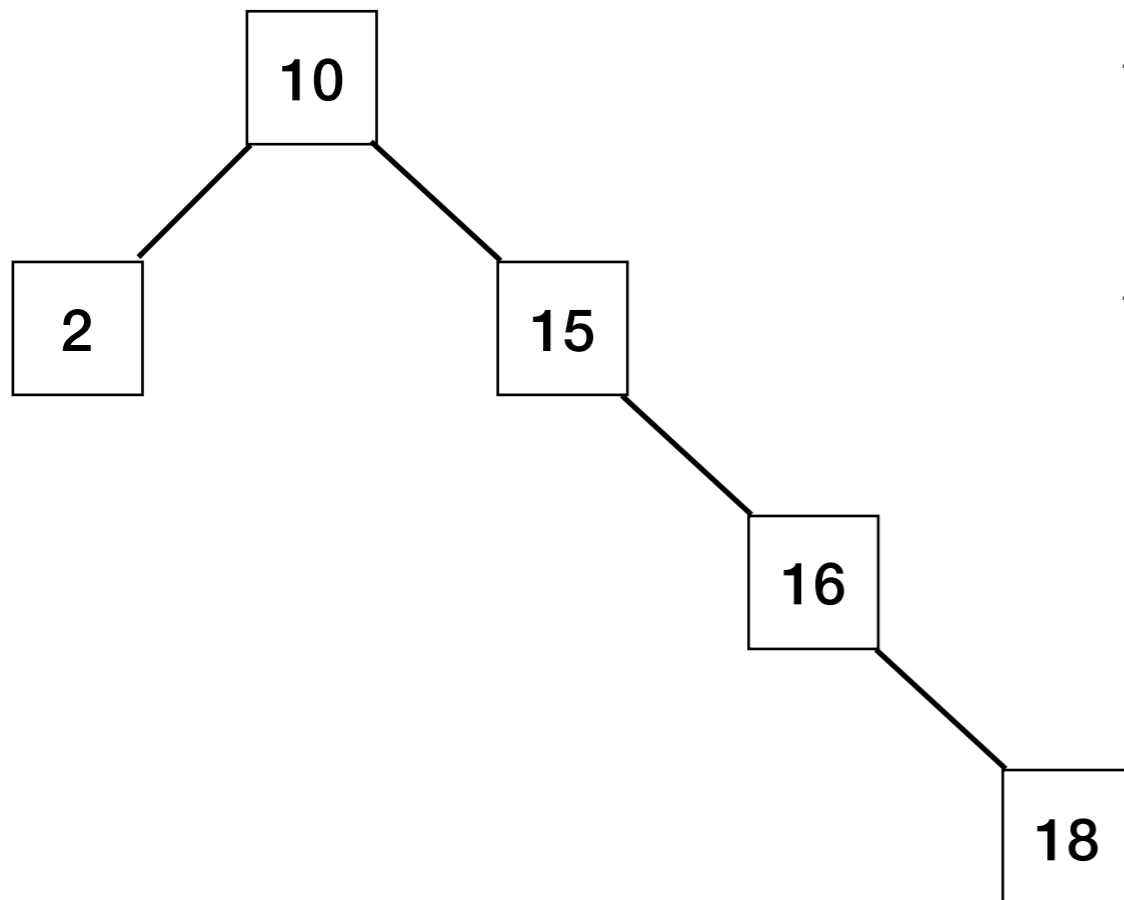
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Draw the tree after leftRotate(15).

=> **Case 3 (RL):**

rightRotate(18)

leftRotate(15)

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

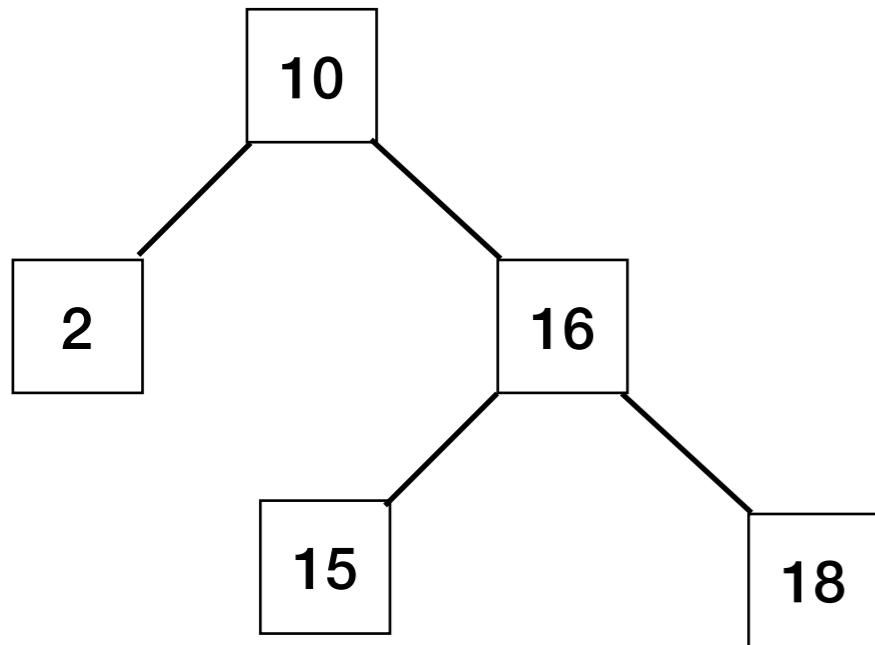
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Draw the tree after leftRotate(15).

=> **Case 3 (RL):**

rightRotate(18)

leftRotate(15)

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

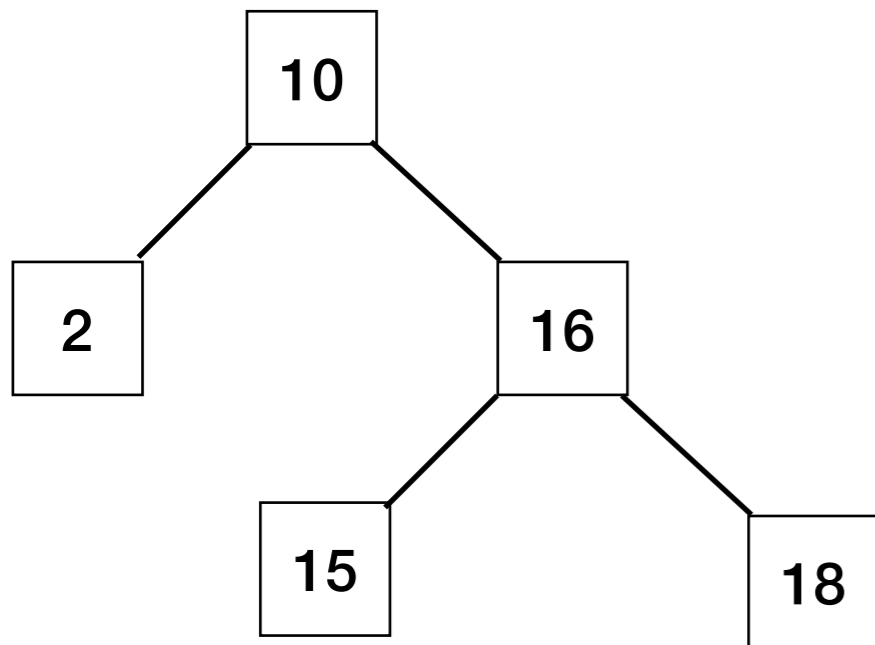
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Recompute heights.

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

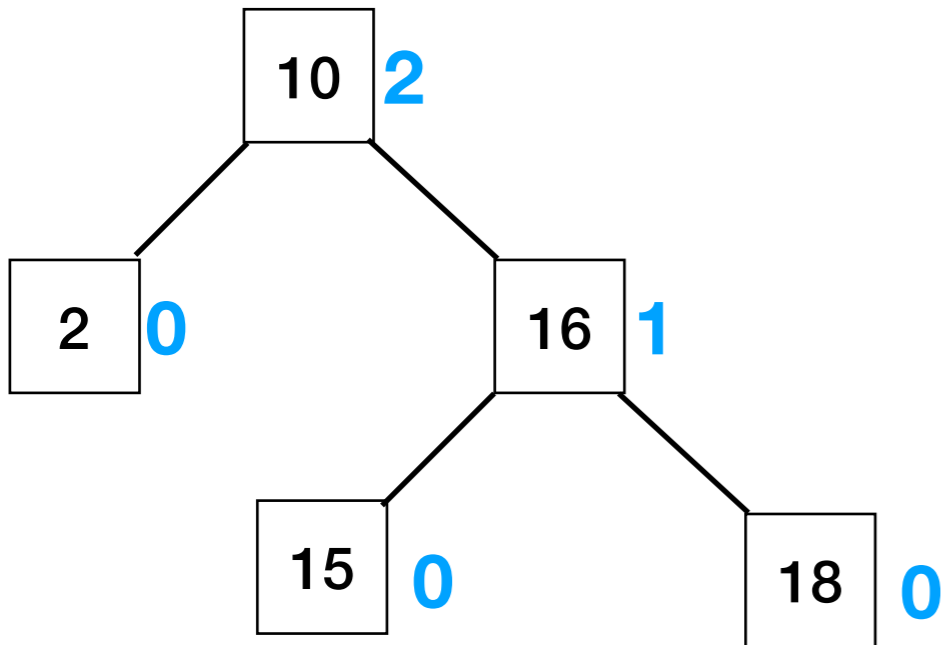
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: Recompute heights.

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

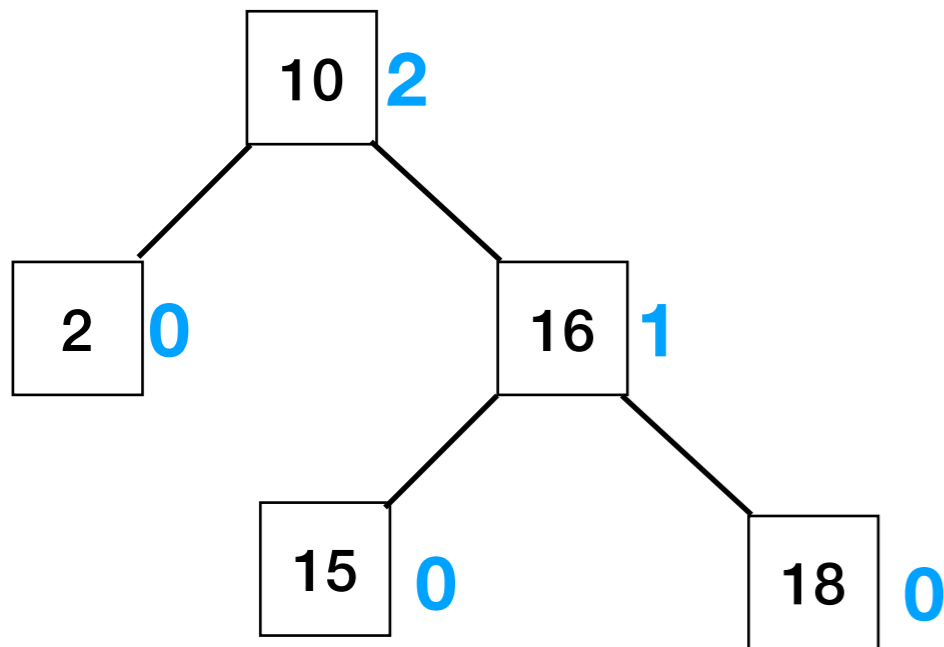
rebalance(18)

=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)



Exercise: What happens when we call rebalance(10)?

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```

avlInsert(root, 16)

rebalance(18)

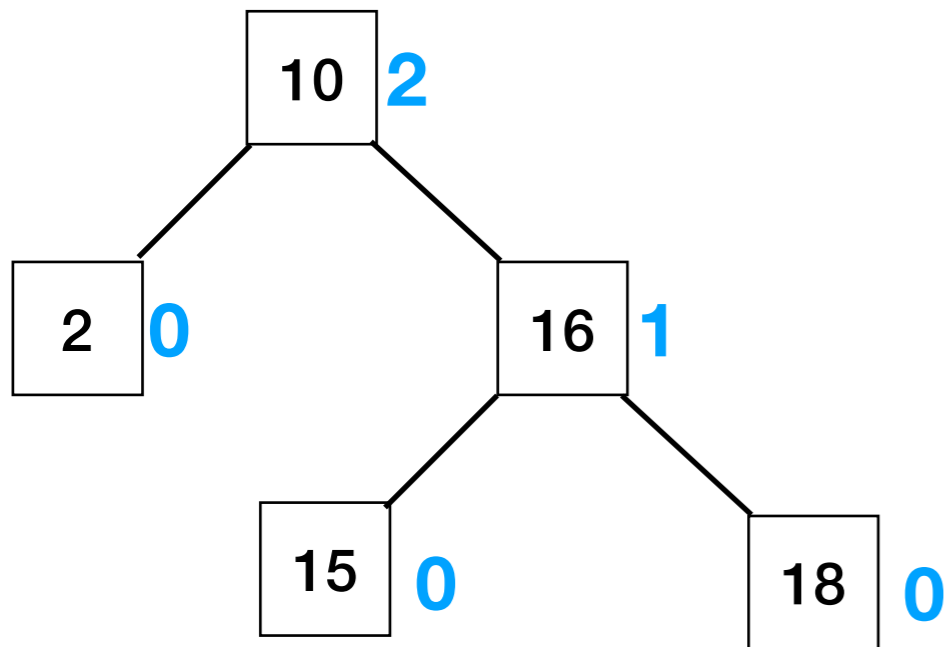
=> bal(18) = -1; already balanced

rebalance(15)

=> bal(15) = 2; need to fix!

rebalance(10)

=> bal(10) = 1; already balanced



Exercise: What happens when we call rebalance(10)?

```
void rebalance(n):  
    if bal(n) < -1:  
        if bal(n.left) < 0  
            // case 1:  
            // rightRot(n)  
        else:  
            // case 2:  
            // leftRot(n.L);  
            // rightRot(n)  
    else if bal(n) > 1:  
        if bal(n.right) < 0:  
            // case 3:  
            // rightRot(n.R);  
            // leftRot(n)  
        else:  
            // case 4:  
            // leftRot(n)
```