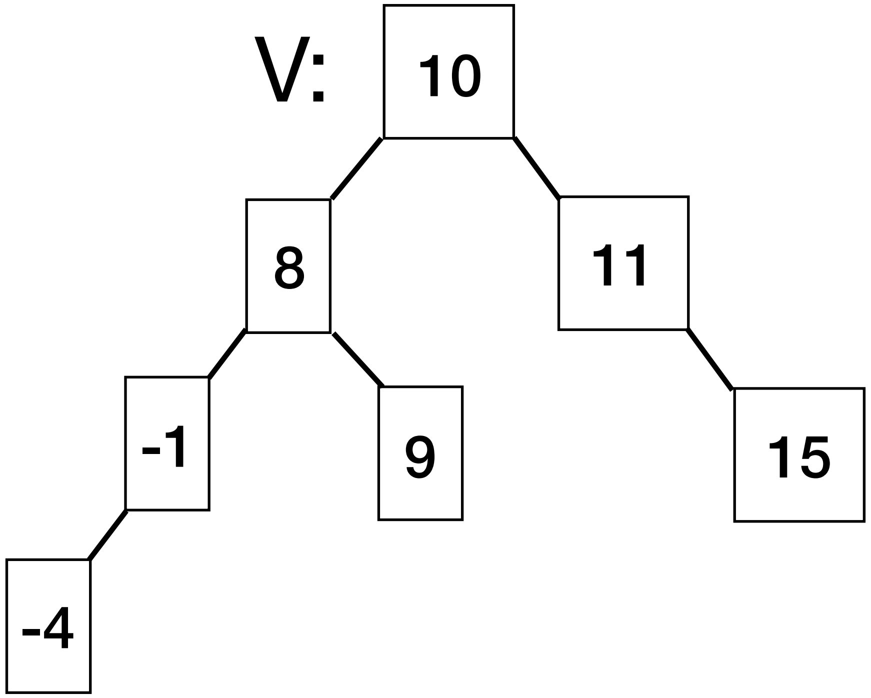
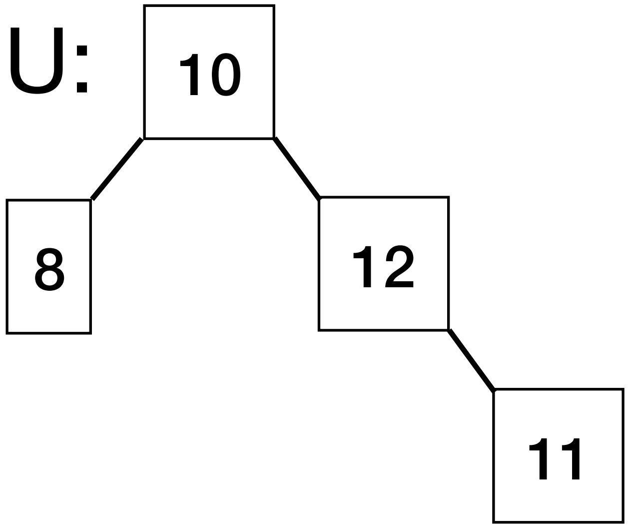
Lecture 14 Exercises:
AVL insertion and rebalancing

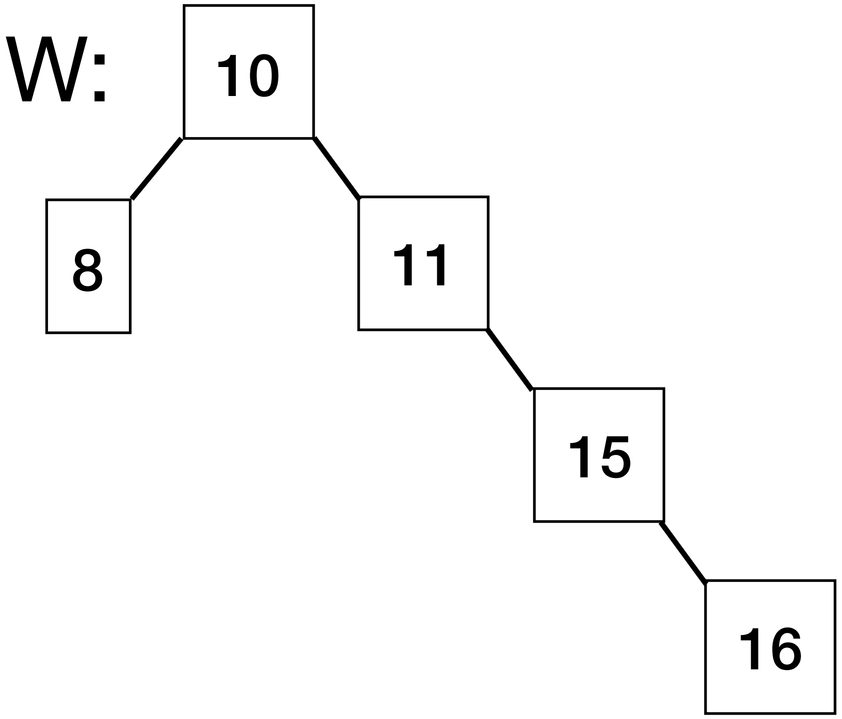
V:



U:



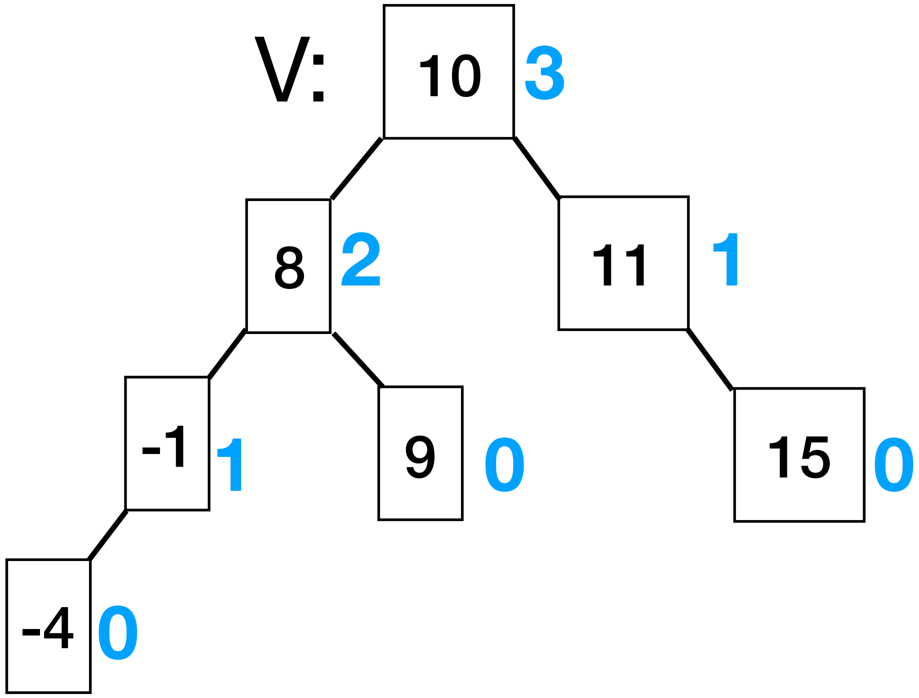
W:



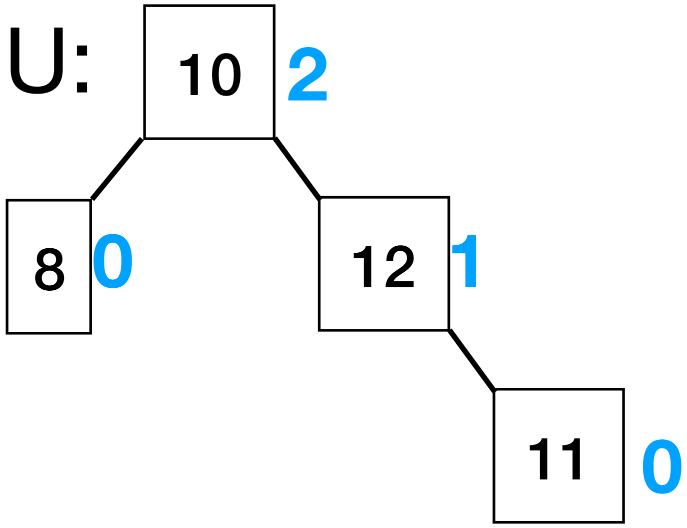
ABCD: Which of these is/are **not** AVL trees?

A. U
B. W
C. V and W
D. U and W

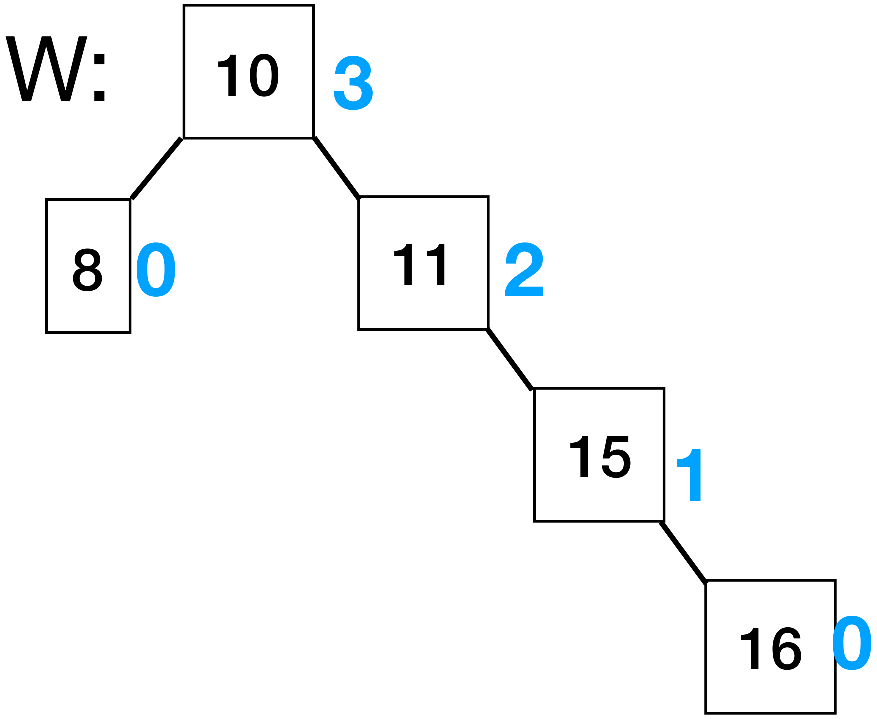
V:



U:



W:



ABCD: Which of these is/are **not** AVL trees?

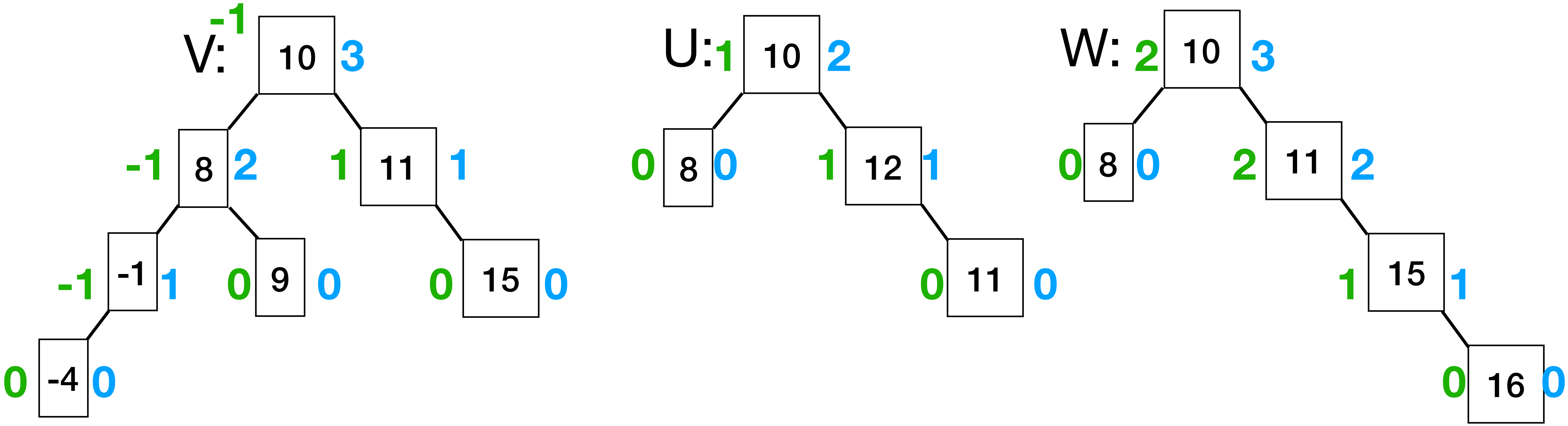
A. U

B. W

C. V and W

D. U and W

Heights in blue.



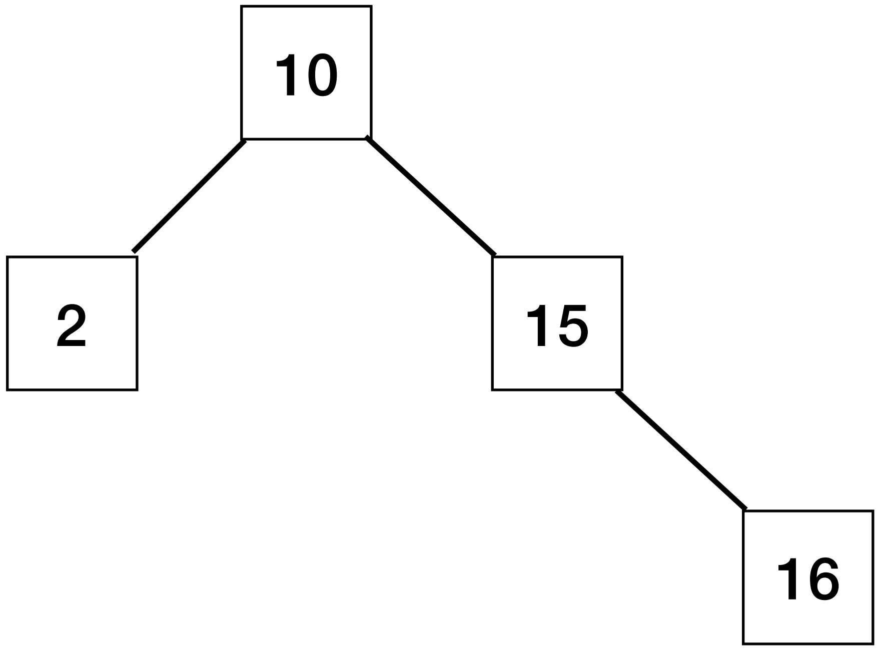
ABCD: Which of these is/are **not** AVL trees?

A. U
B. W
C. V and W
D. U and W

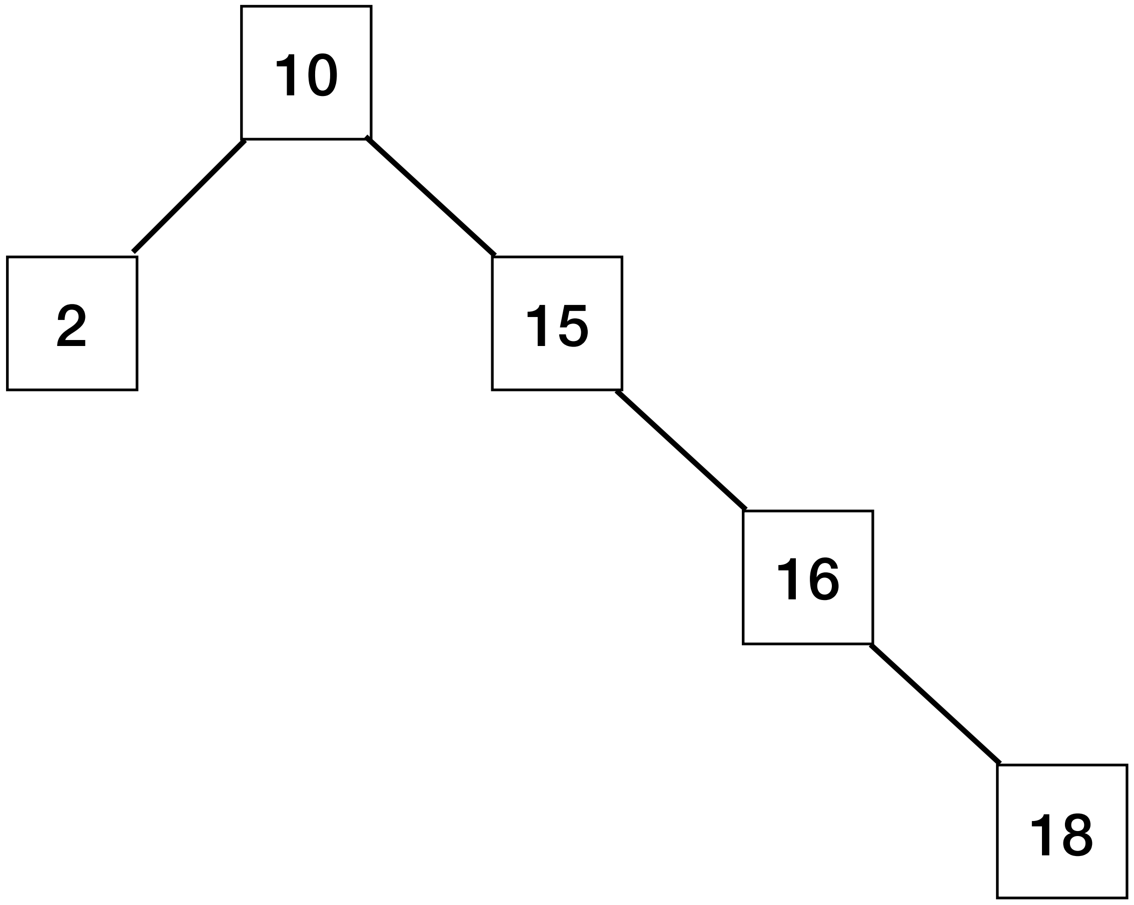
Heights in blue.

Balance factors in green.

Exercise: Do the BST insertion part of `avlInsert(root, 18)`

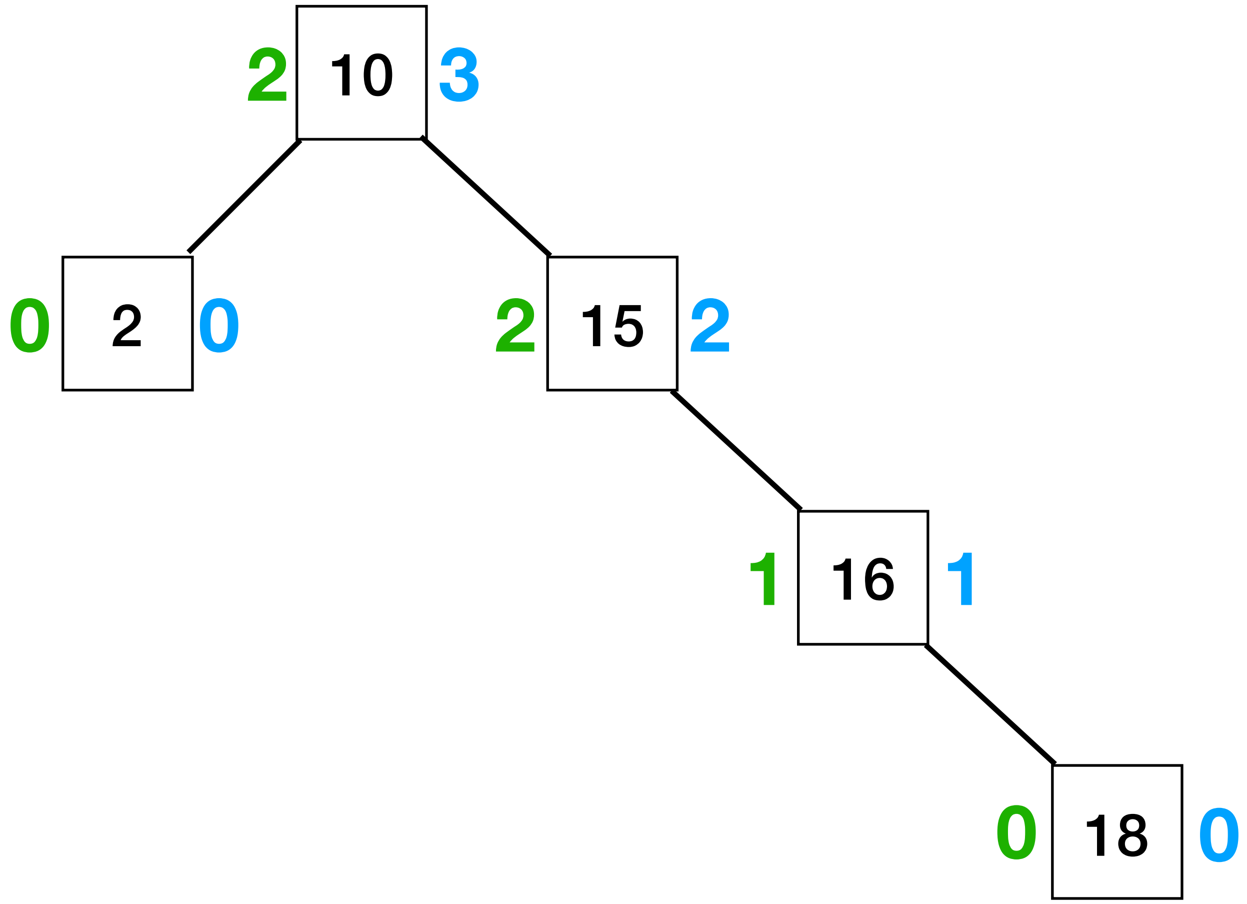


`avlInsert(root, 18)`



Exercise: 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.

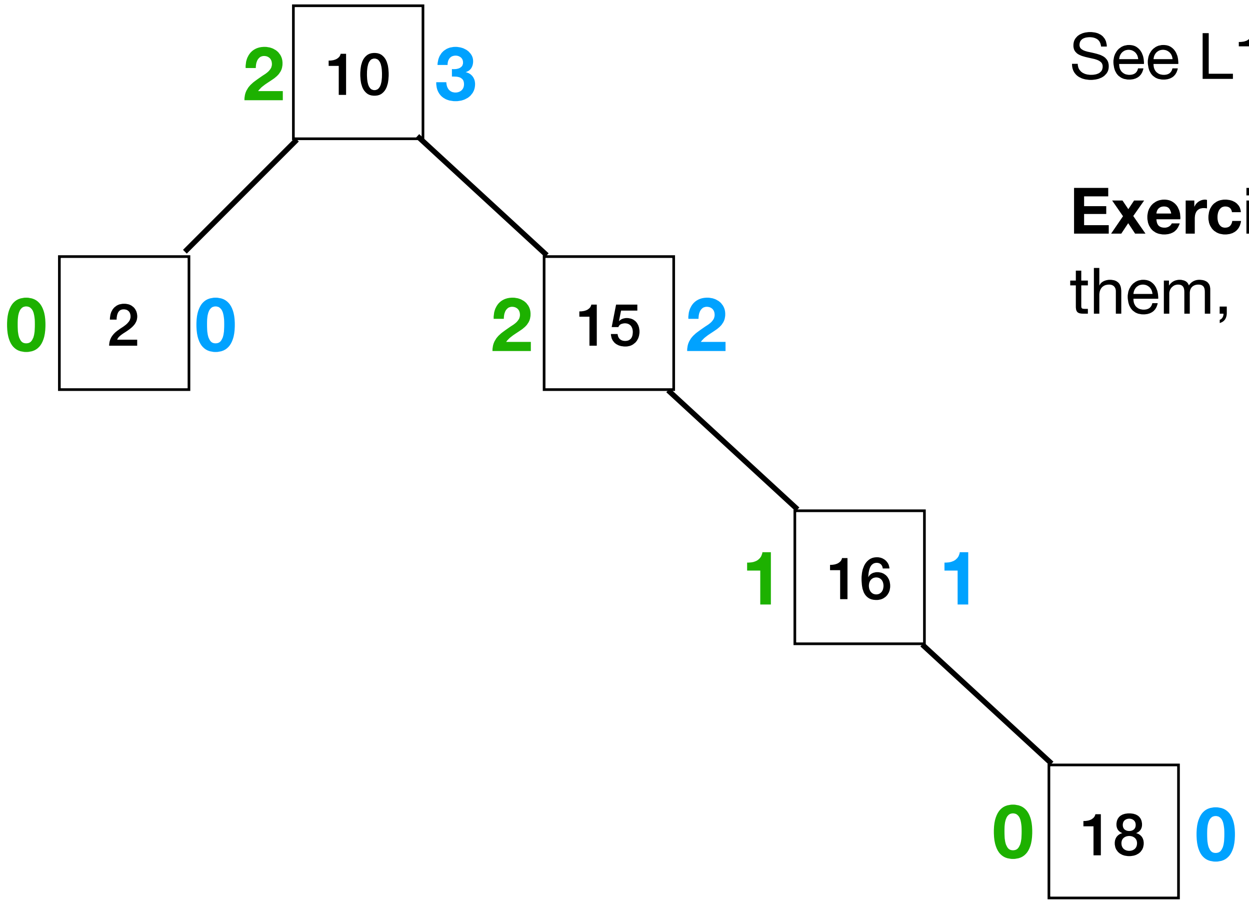
`avlInsert(root, 18)`



Balance(n): height(n.right) - height(n.left)

Exercise: 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 L14 slides for details.

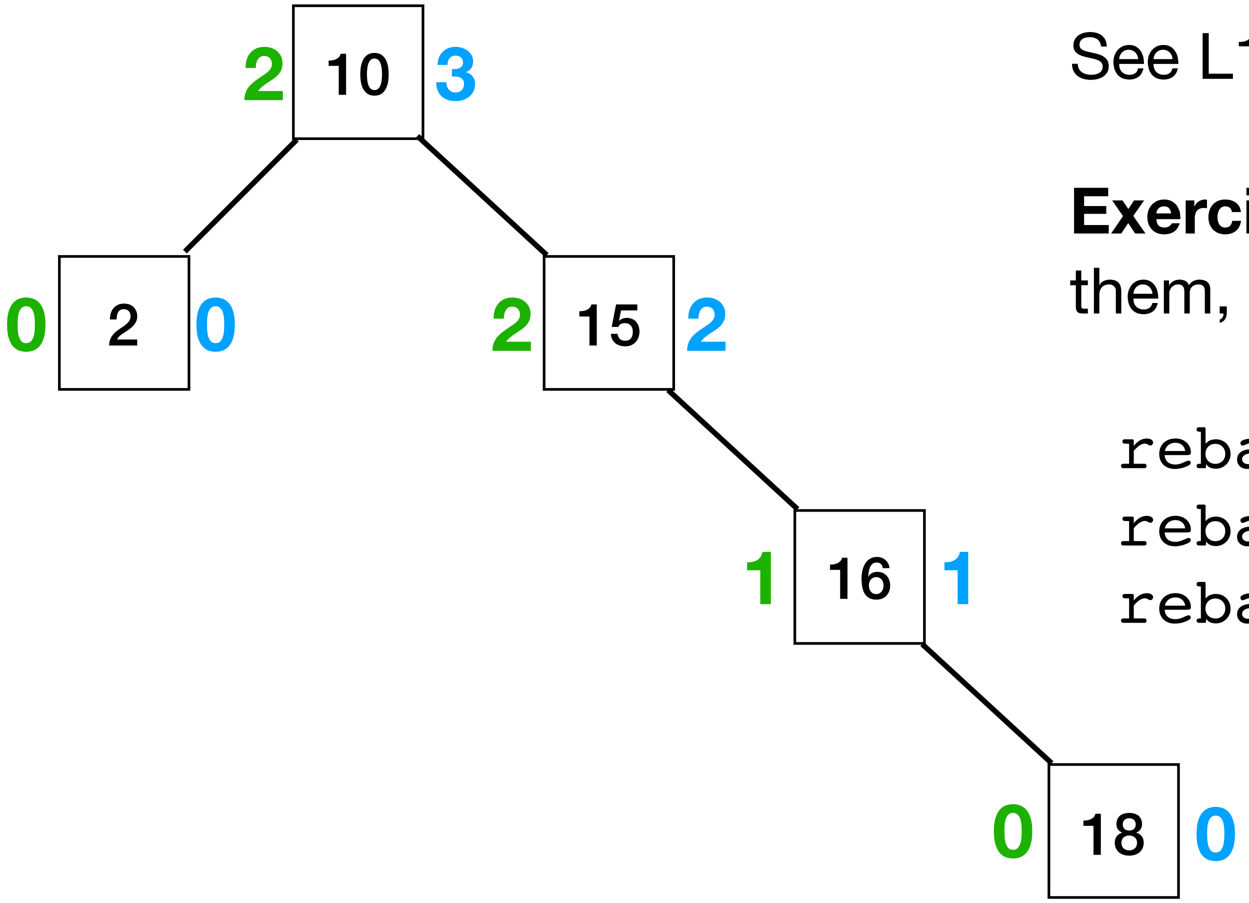
Exercise: What nodes have `rebalance` called on them, and in what order?

`avlInsert(root, 18)`

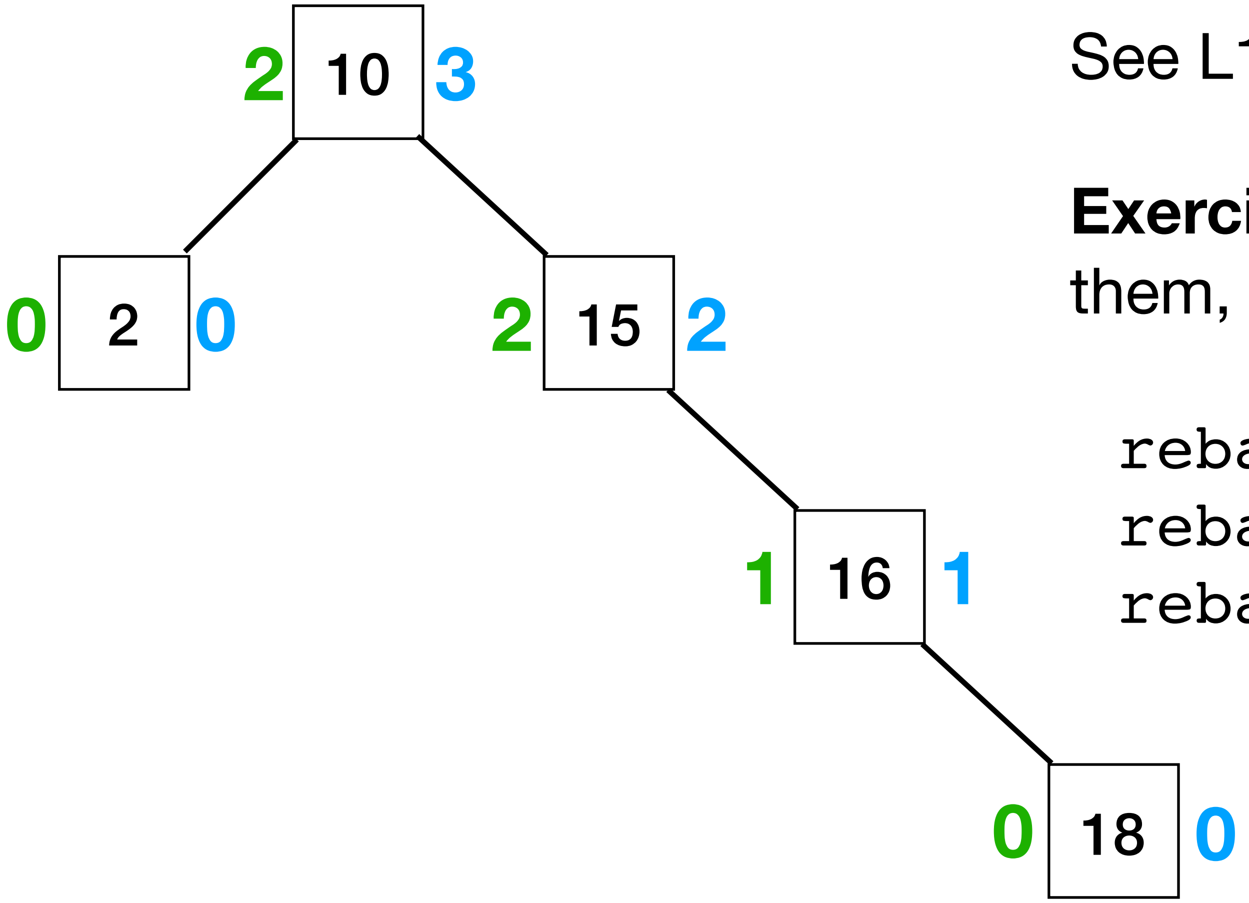
After the new node is inserted into the tree, the recursion will walk back up the tree, calling `rebalance` on each parent node in succession. See L14 slides for details.

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 L14 slides for details.

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

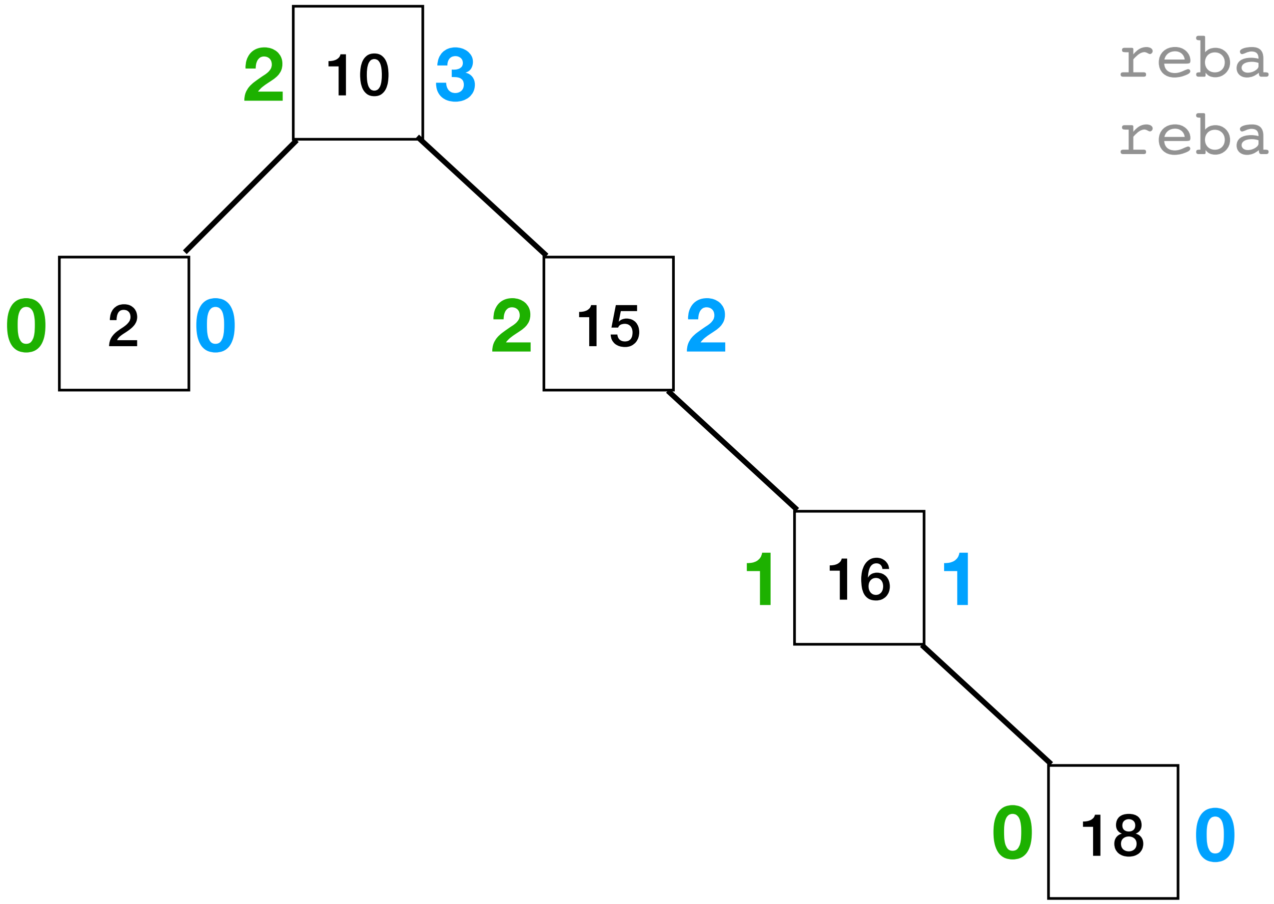
```
rebalance(16)
rebalance(15)
rebalance(10)
```

ok, let’s execute this!

`avlInsert(root, 18)`

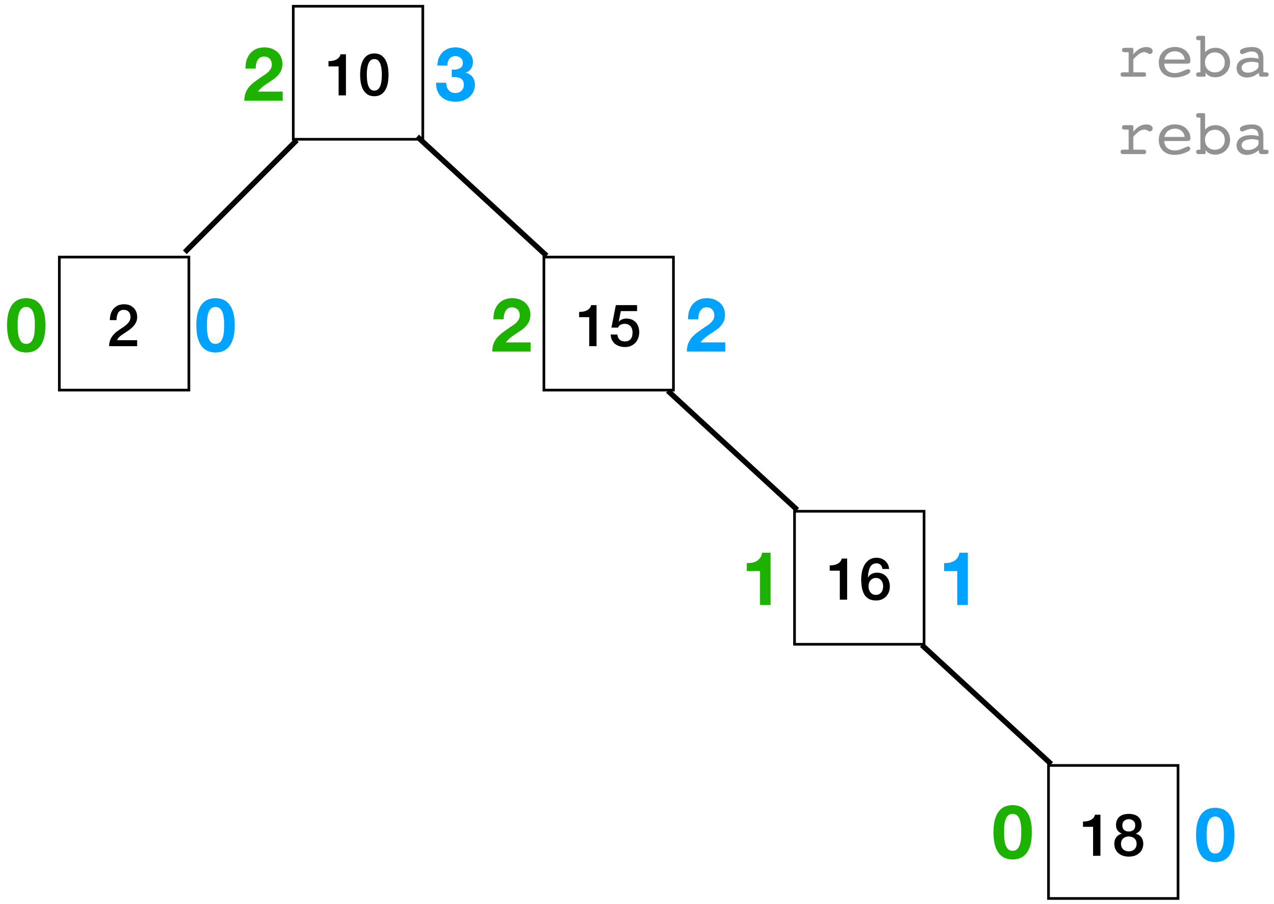
`rebalance(16)`

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



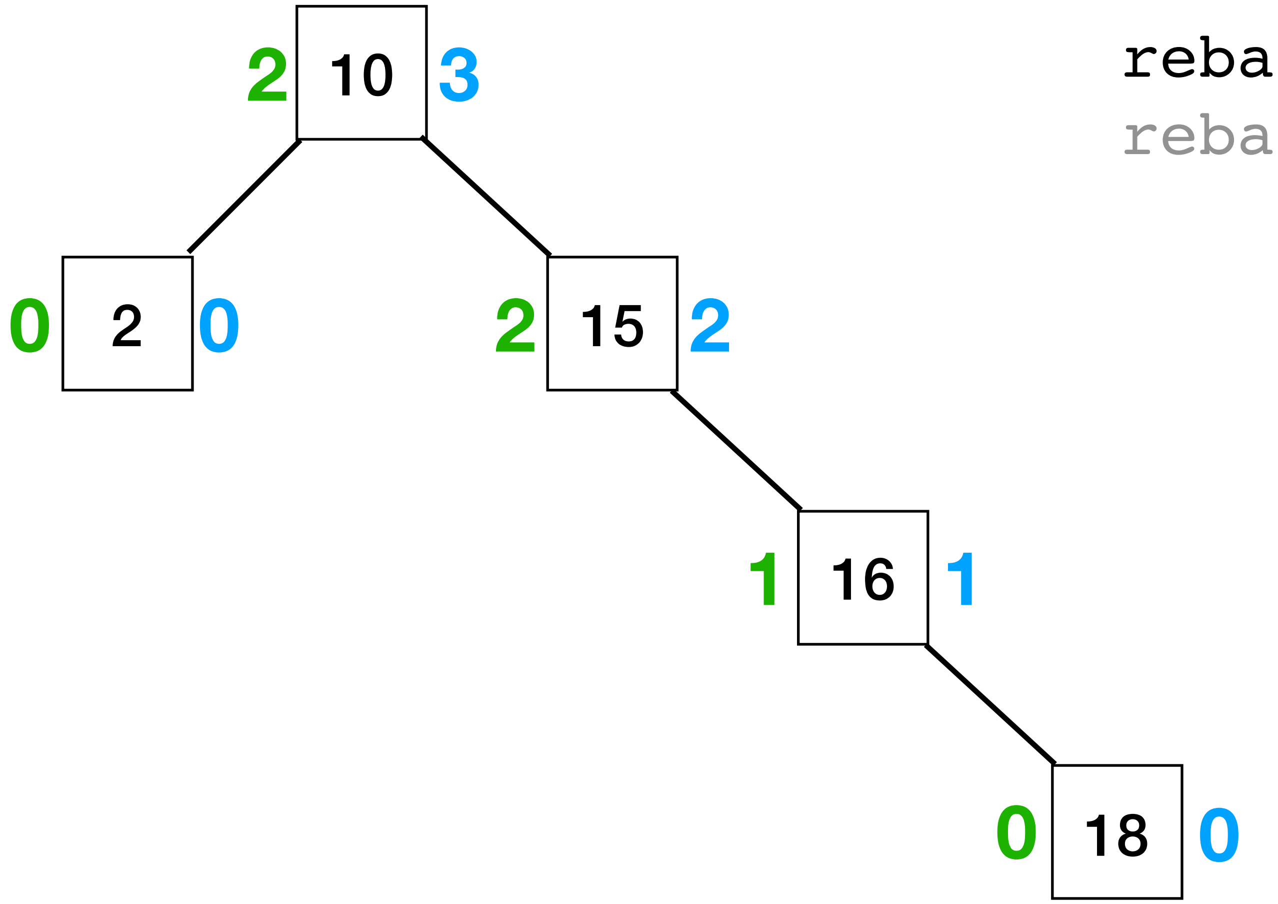
```
avlInsert(root, 18)
```

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

 rebalance(15)
 rebalance(10)
```

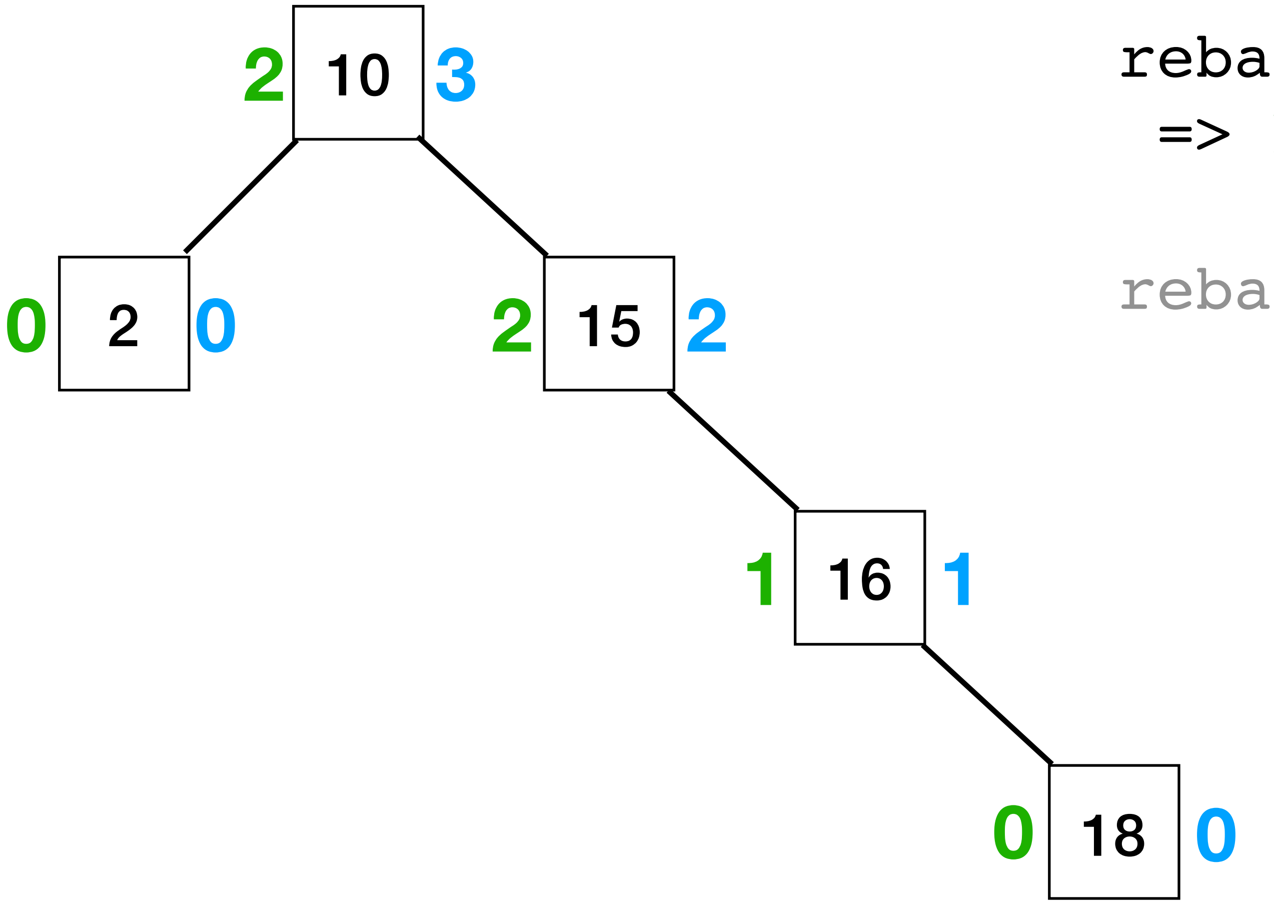


`avlInsert(root, 18)`

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

rebalance(15)
rebalance(10)
```



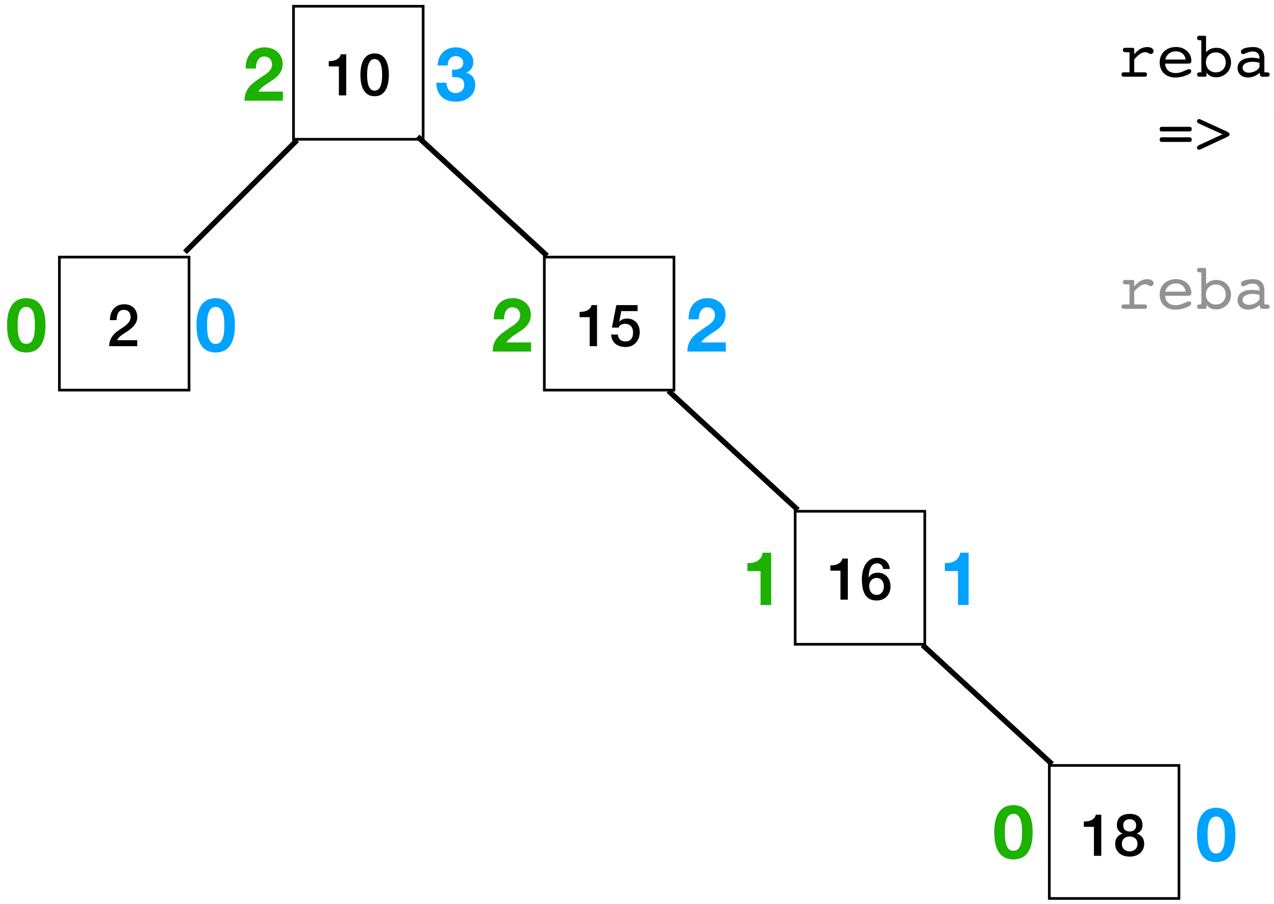
`avlInsert(root, 18)`



```
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)
```

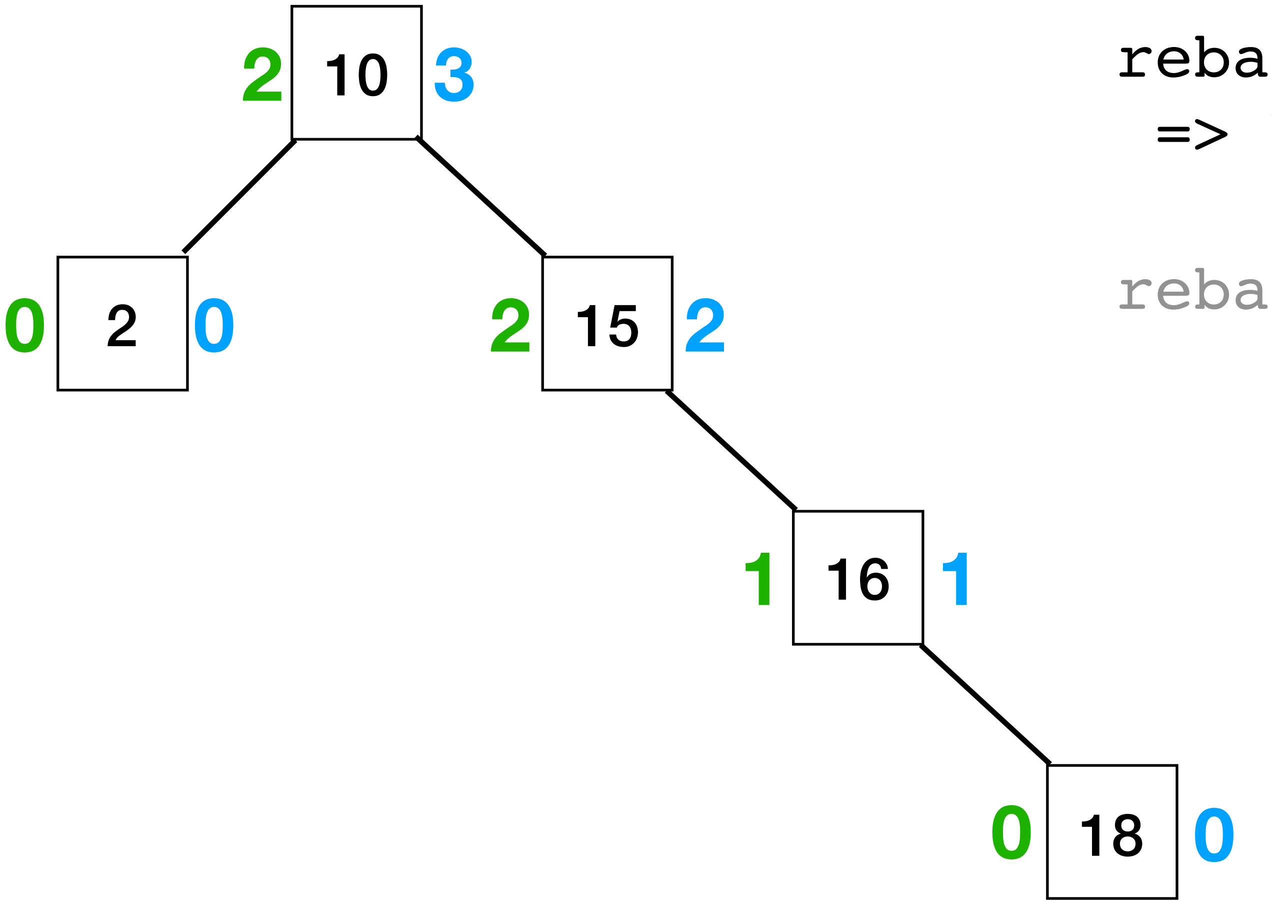


```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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

`avlInsert(root, 18)`

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

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

rebalance(10)
```



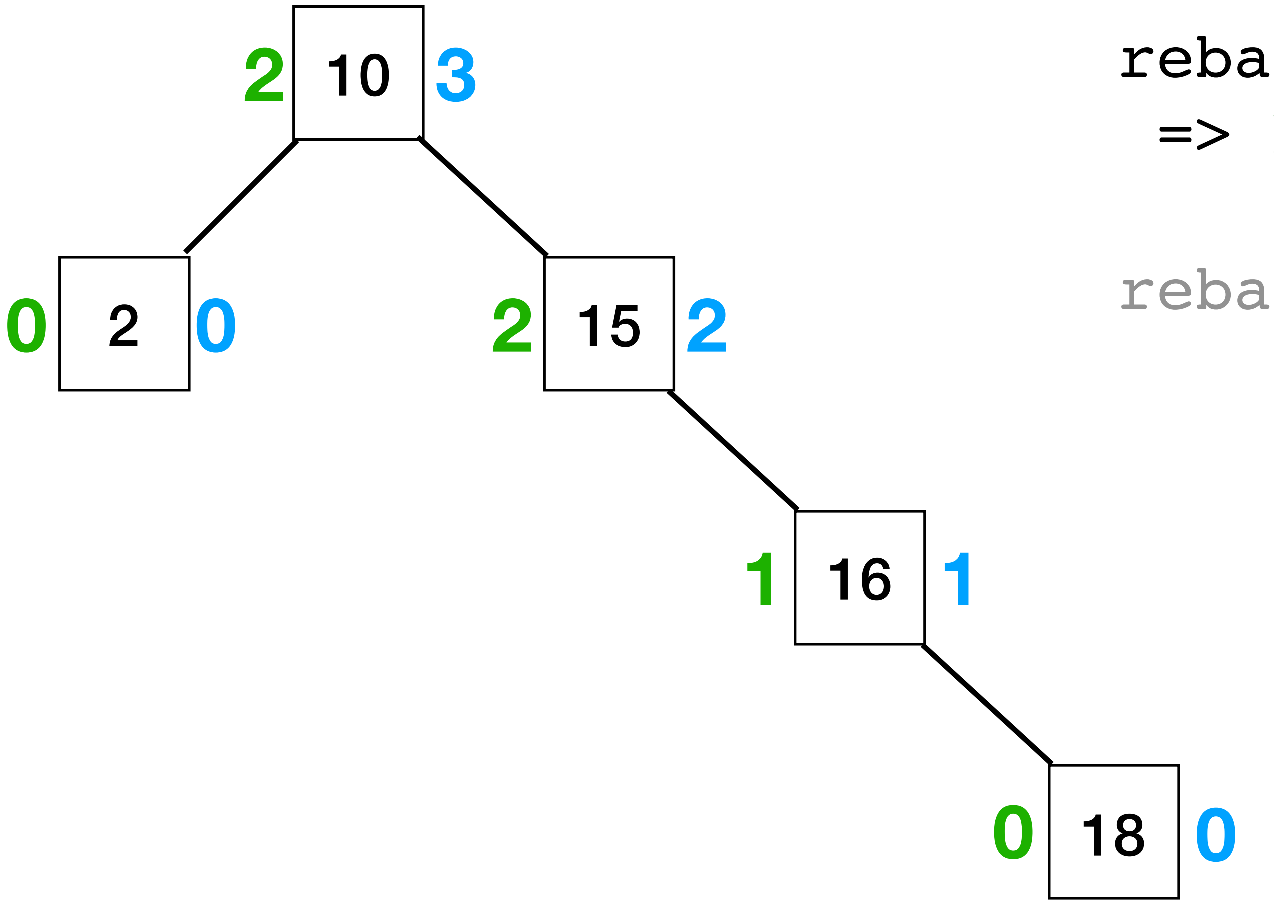
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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)
```

`avlInsert(root, 18)`

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

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

rebalance(10)
```

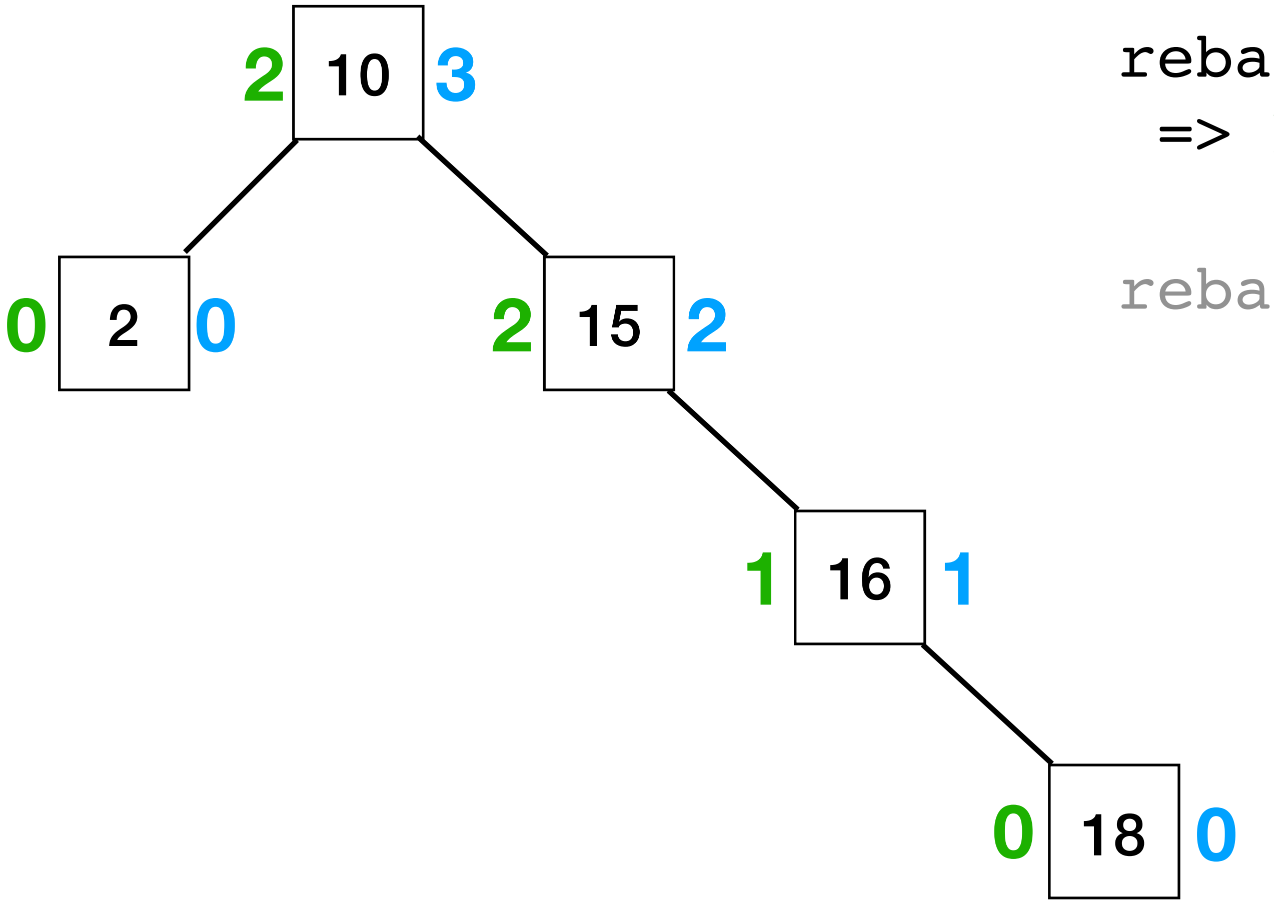


```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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



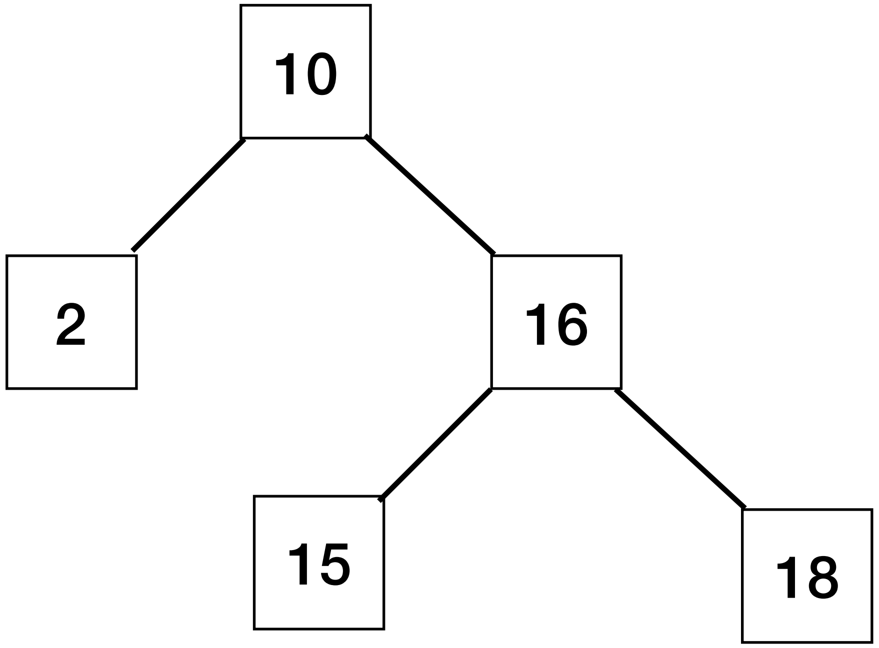
Exercise: Draw the tree after a left rotation on 15.

`=>` **`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)
```



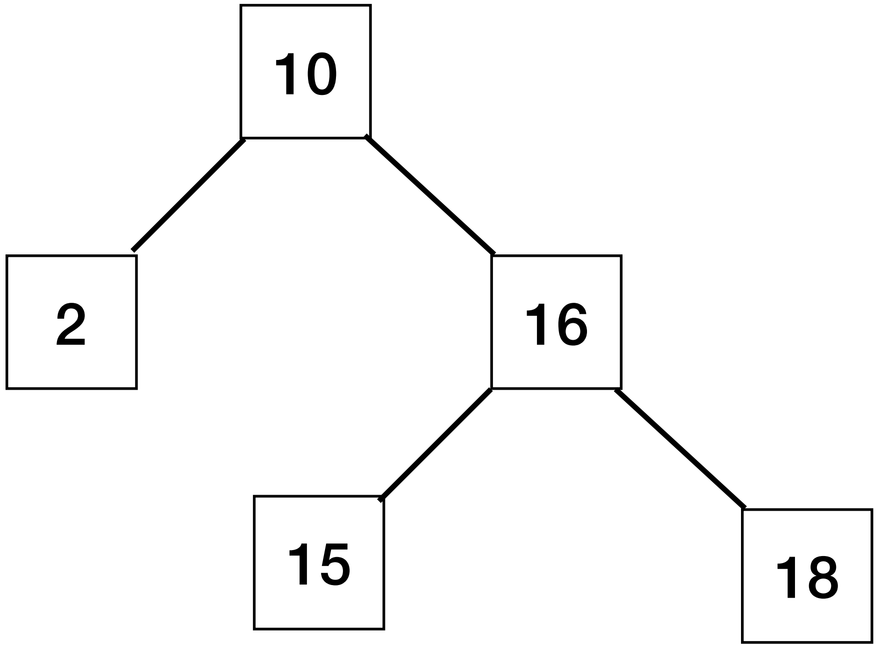
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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

=> **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)
```

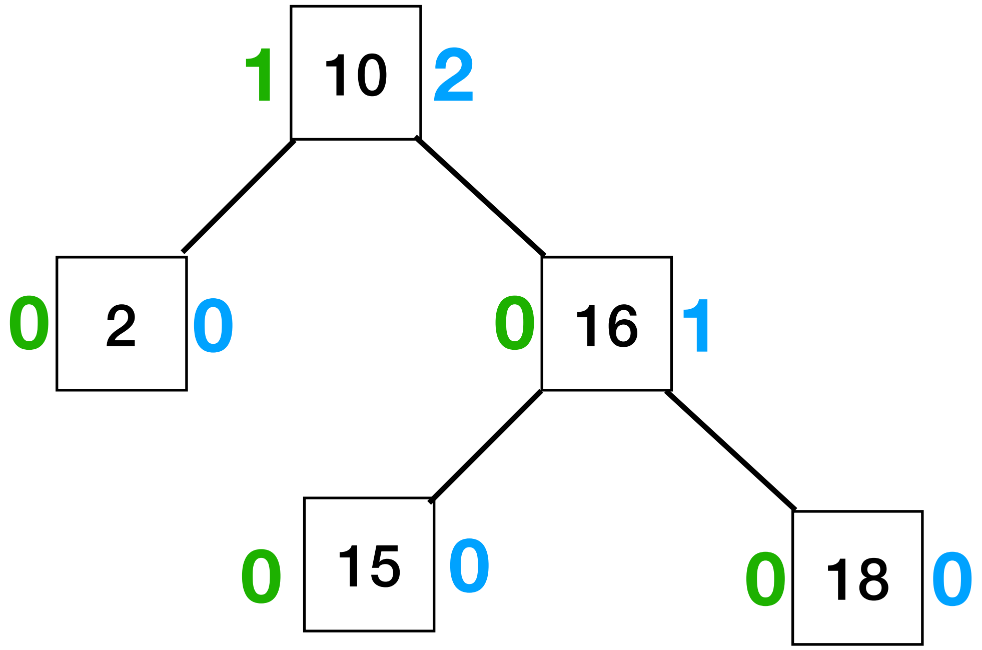


```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

Exercise: Recompute heights and balance factors.

avlInsert(root, 18)

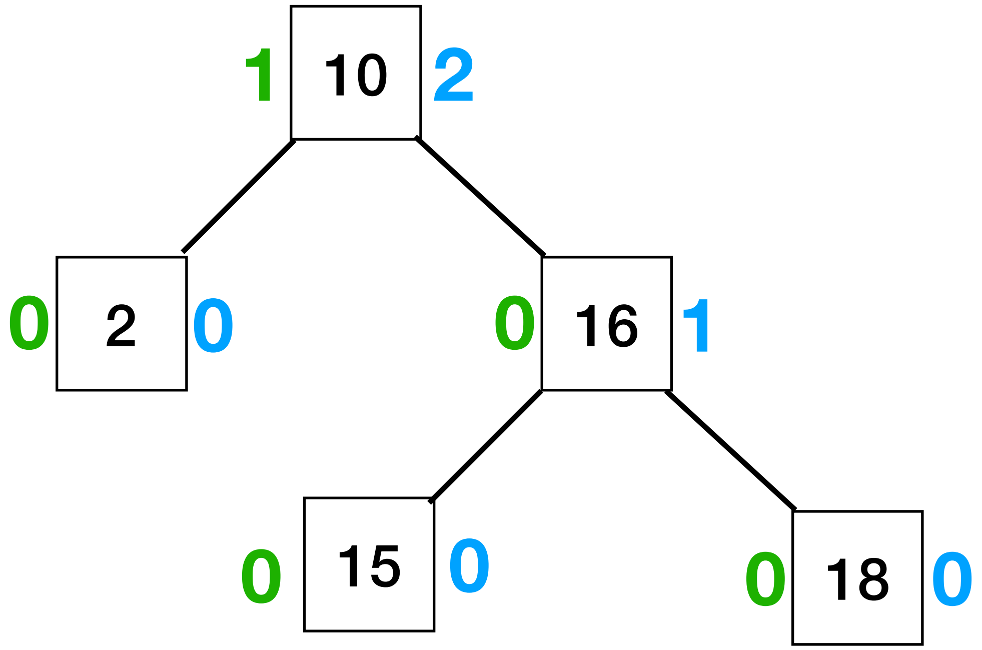
```
rebalance(16)
 => bal(16) = 1; already balanced

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)

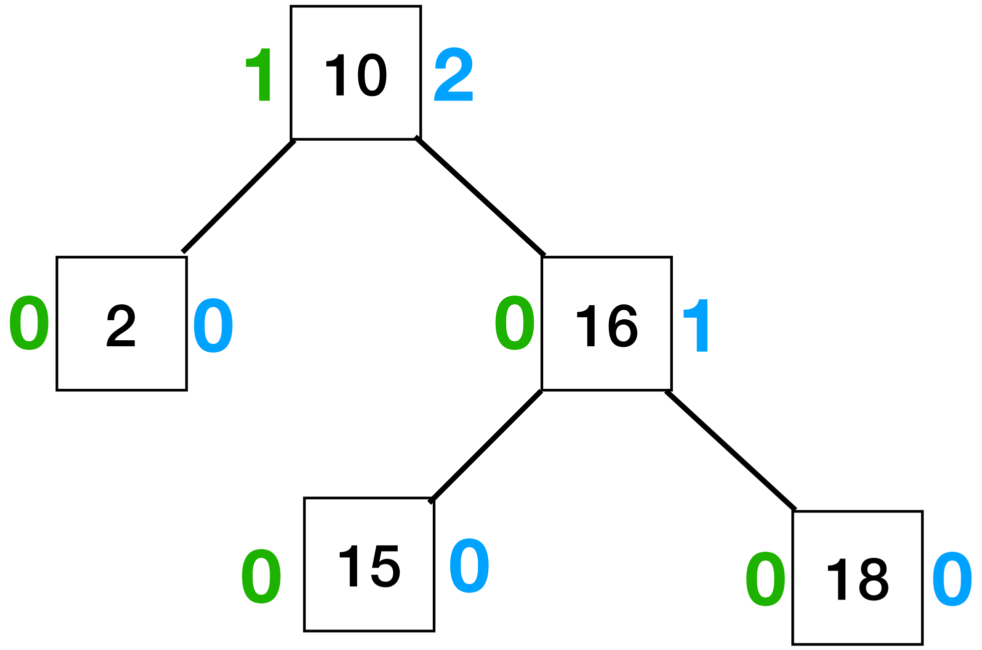


```
rebalance(16)
 => bal(16) = 1; already balanced

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)

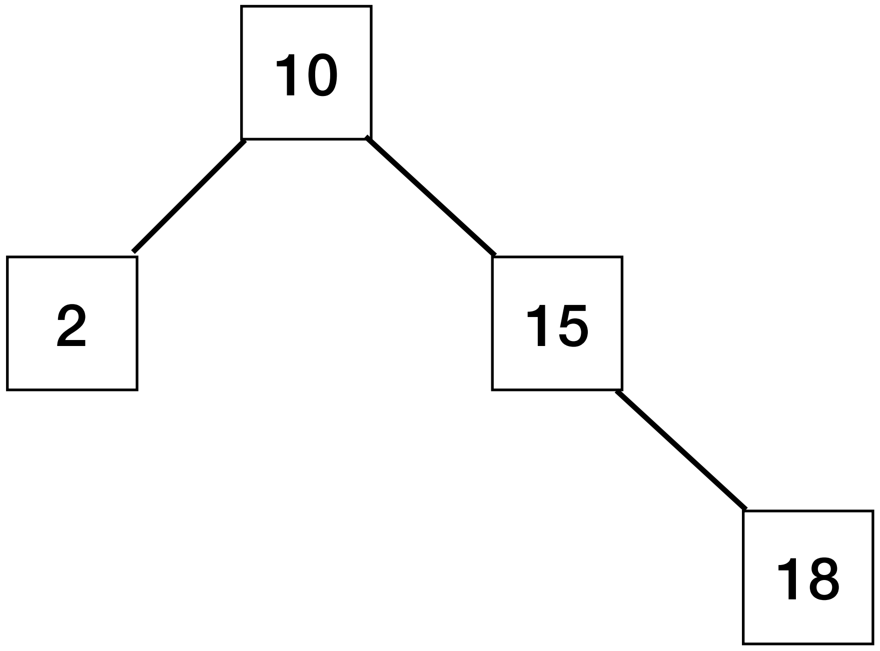
```
rebalance(16)
 => bal(16) = 1; already balanced

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

rebalance(10)
 => bal(10) = 1; already balanced
```



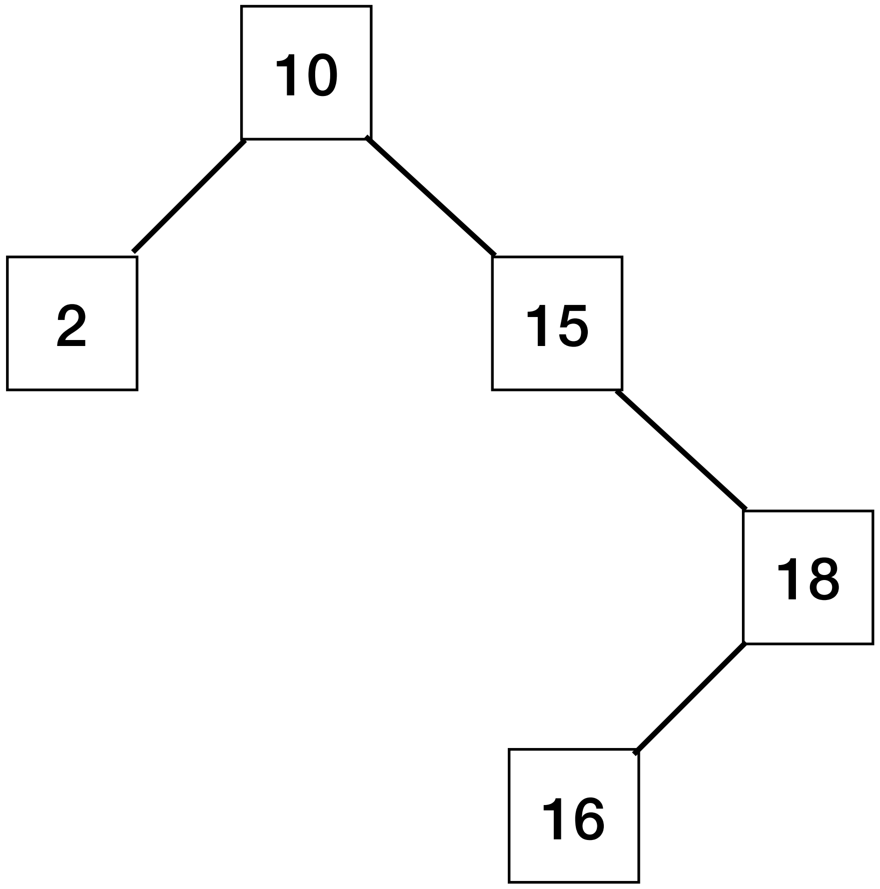
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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

A different case

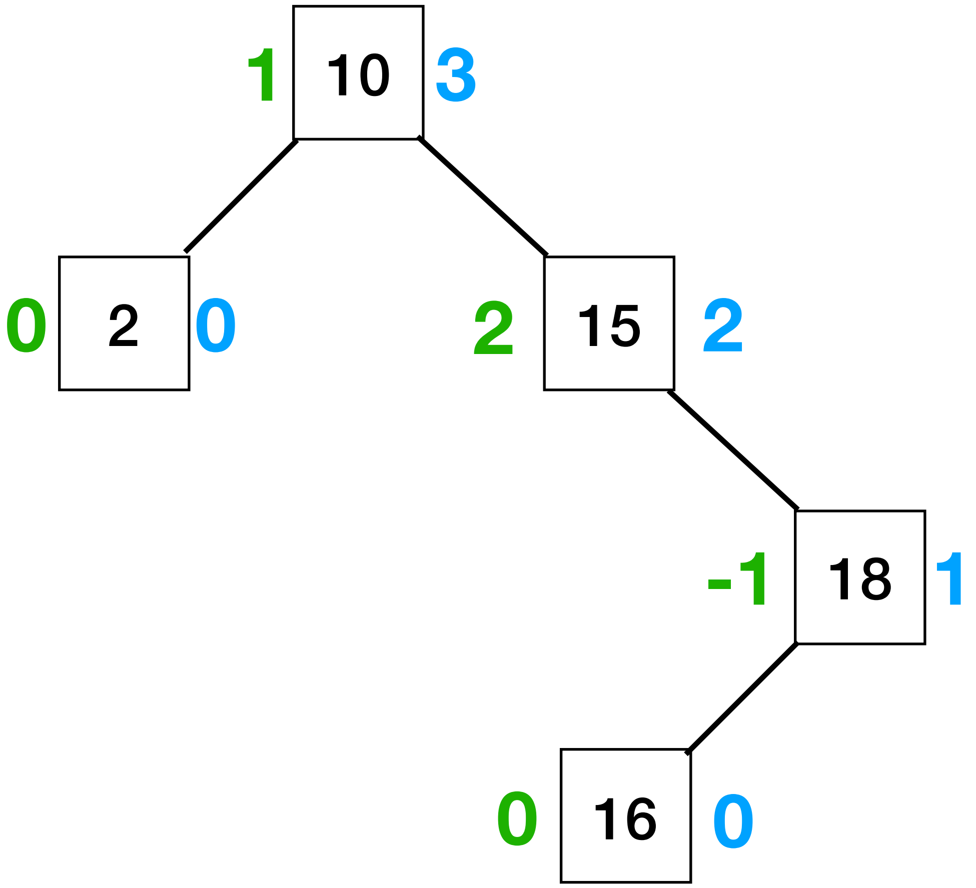


`avlInsert(root, 16)`



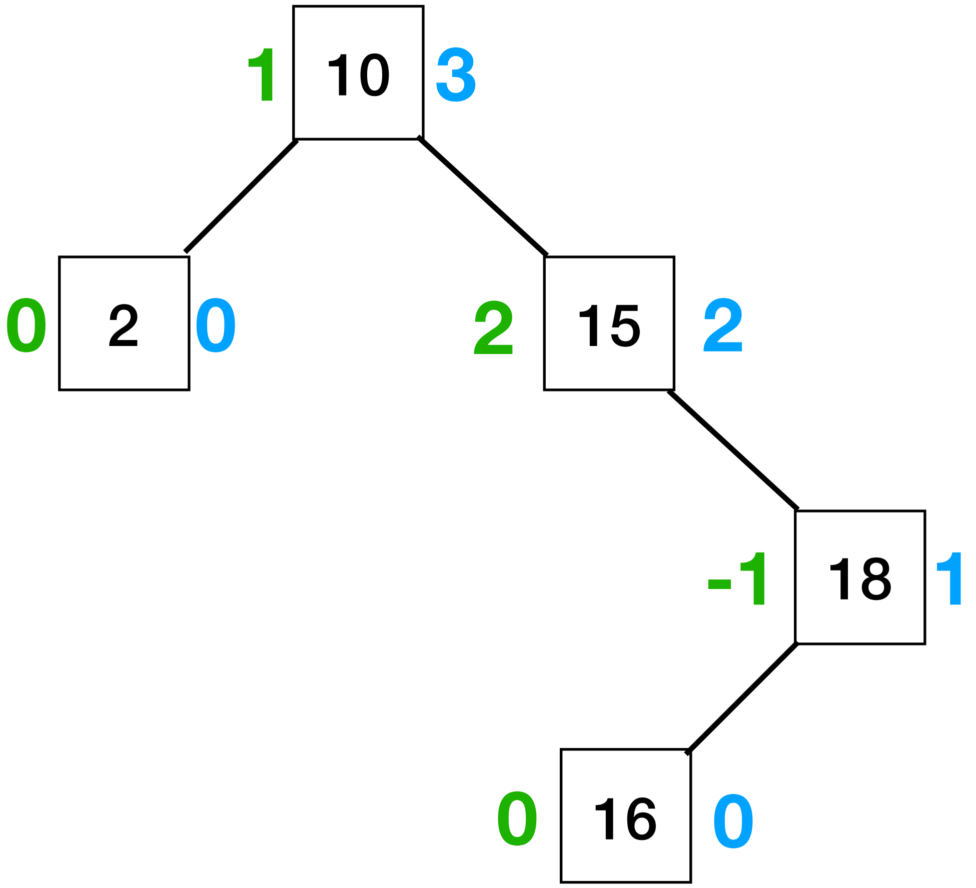
Exercise: Compute heights and balance factors.

`avlInsert(root, 16)`



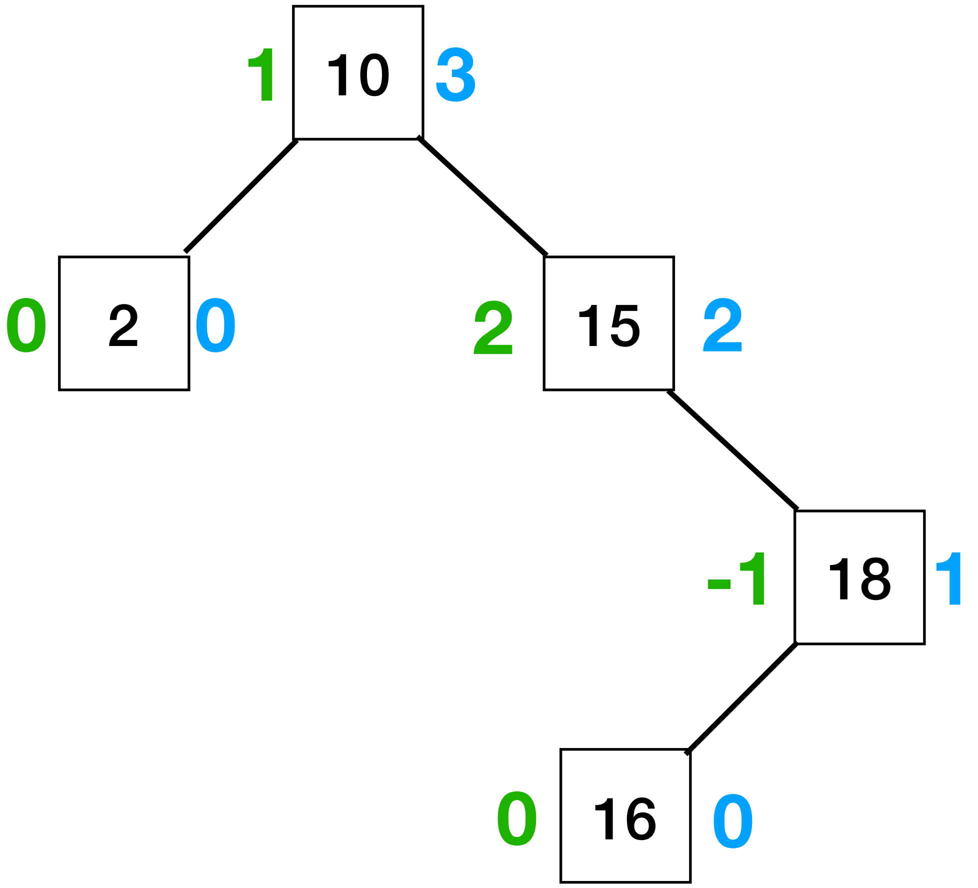
Exercise: Compute heights and balance factors.

`avlInsert(root, 16)`

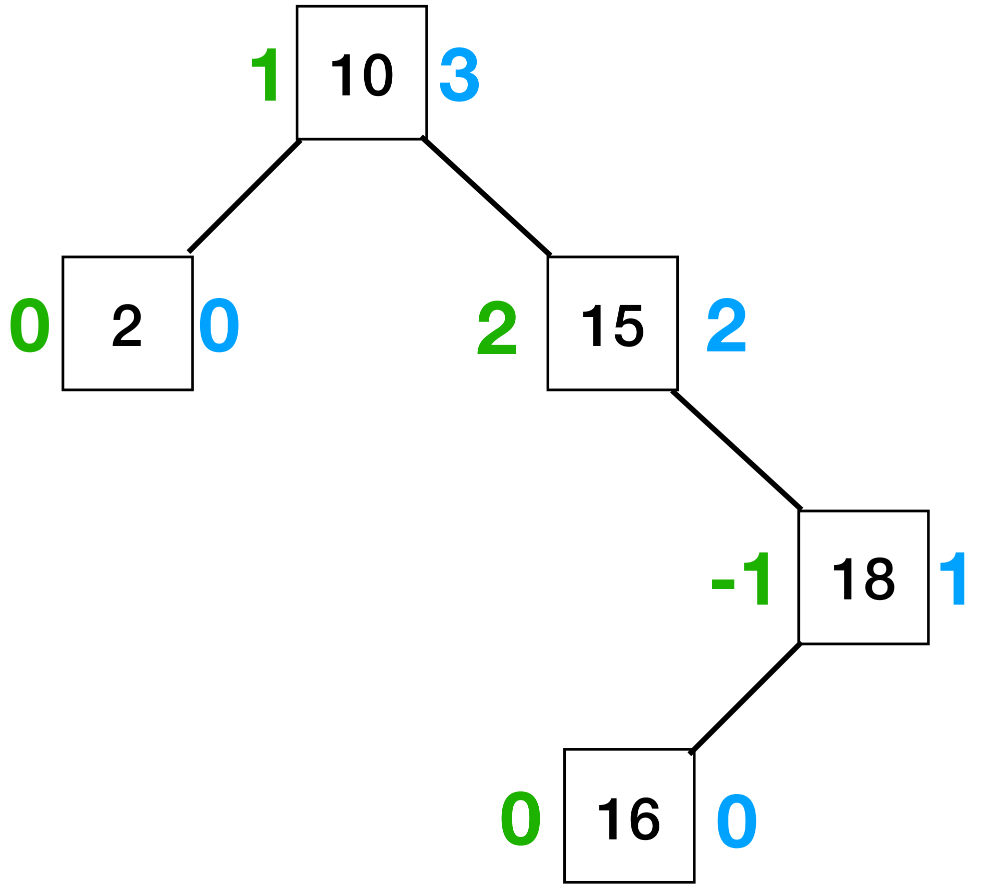


Exercise: Write the sequence of `rebalance` calls.

```
avlInsert(root, 16)
  rebalance(18)

  rebalance(15)

  rebalance(10)
```

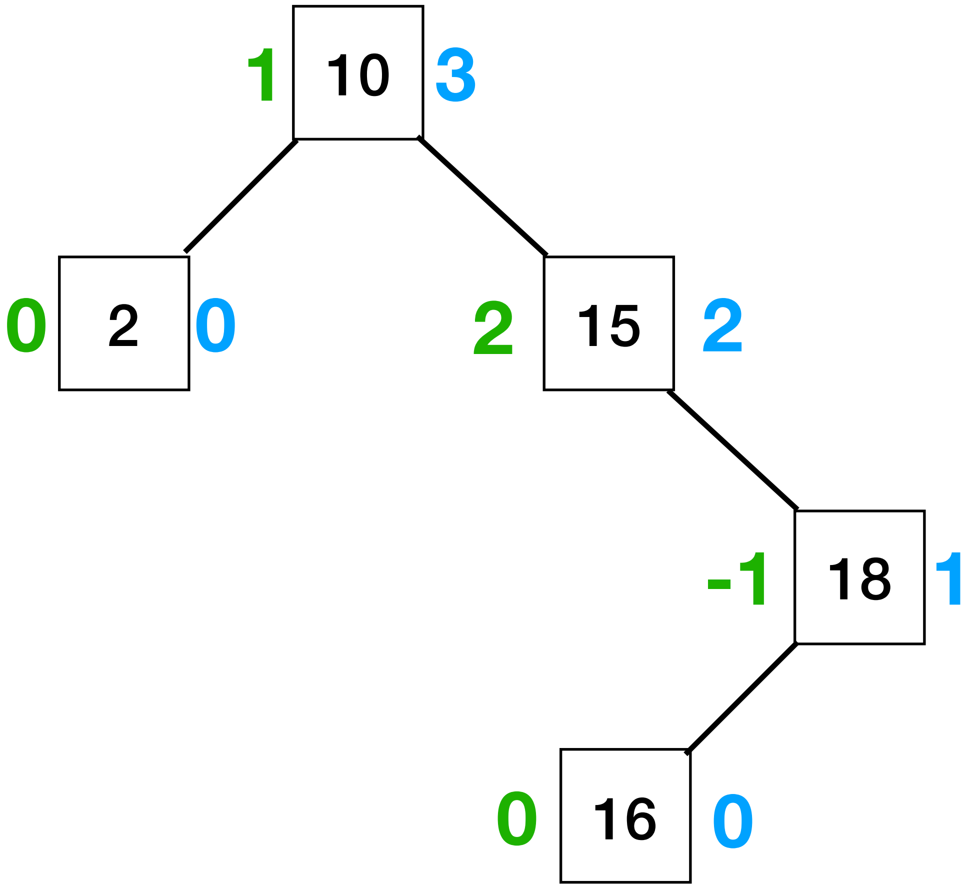


Exercise: Write the sequence of `rebalance` calls.



```
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)
```

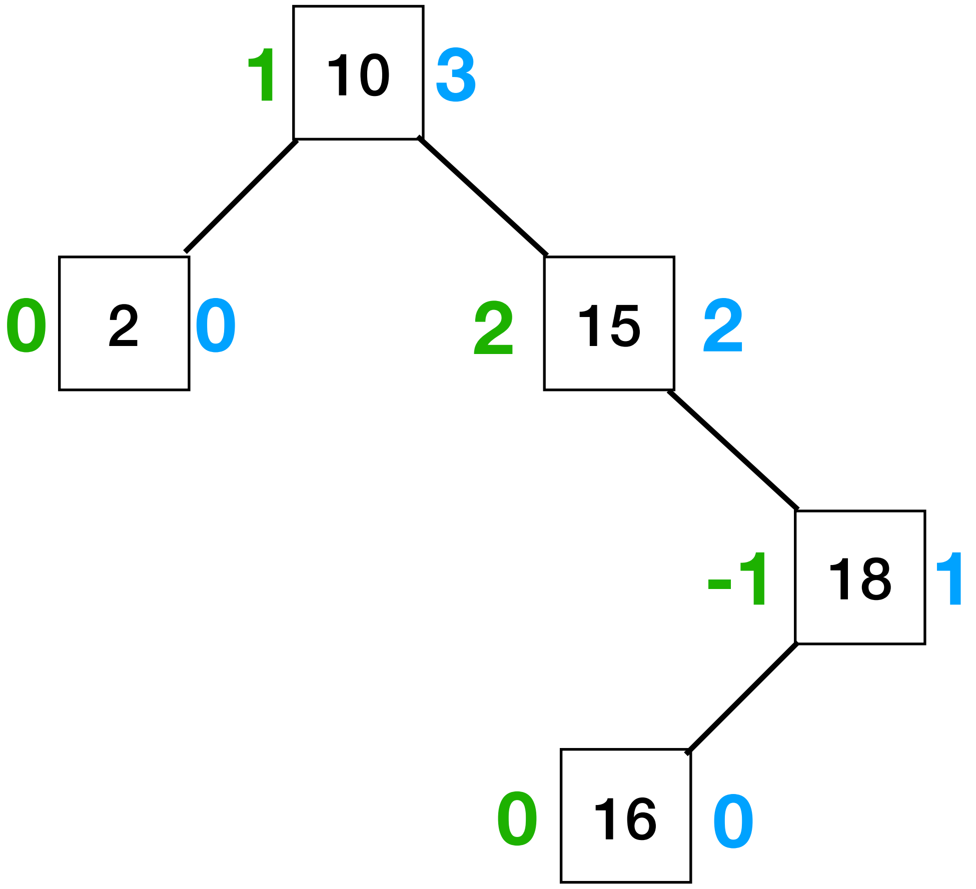


Exercise: Which case applies here?

```
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)
```



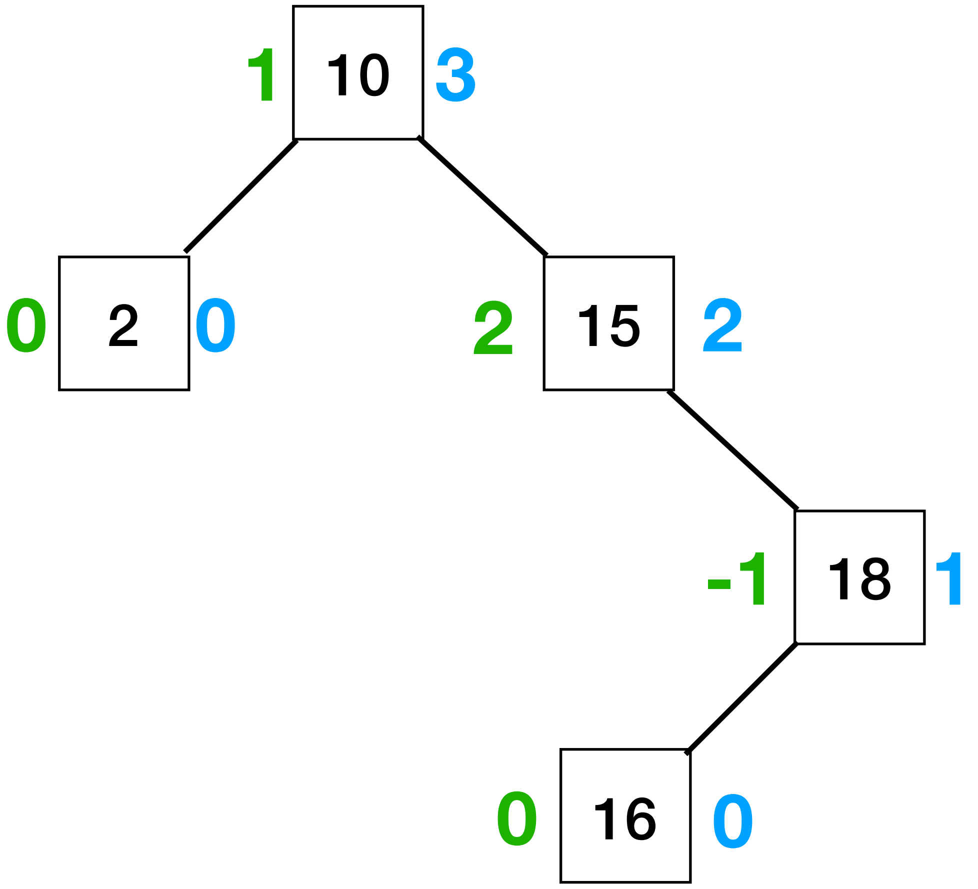
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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):
```

`avlInsert(root, 16)`

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

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

rebalance(10)
```



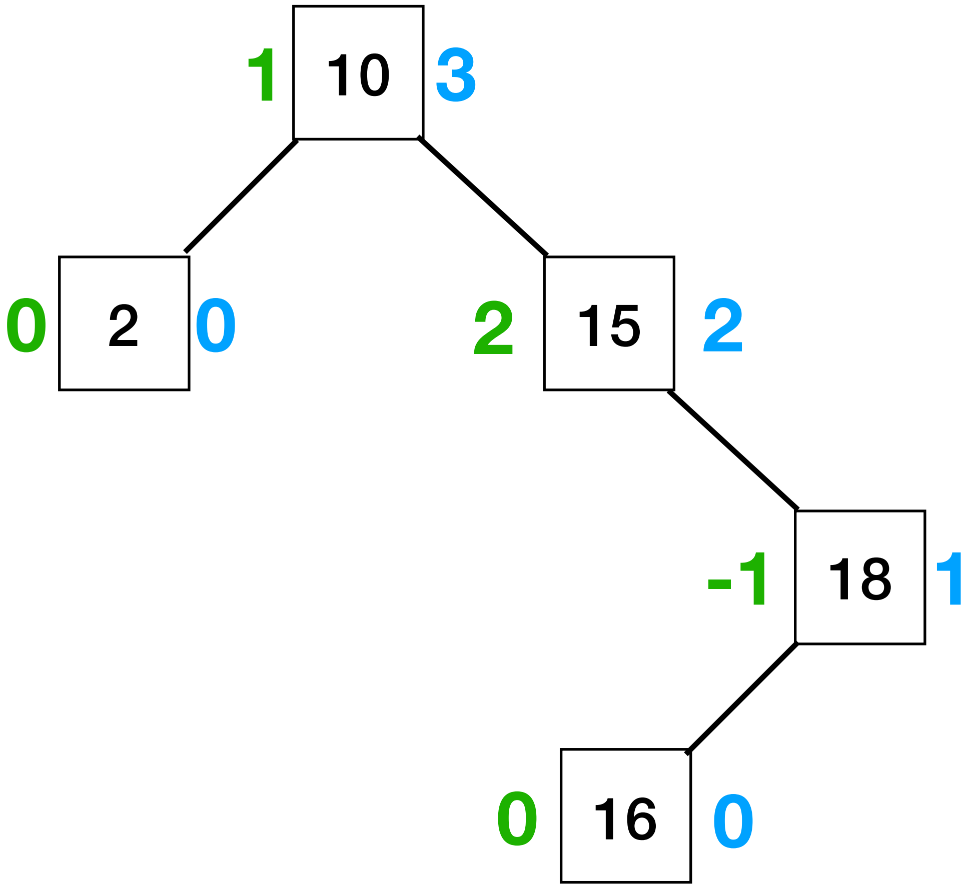
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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)
```

avlInsert(root, 16)

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

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

rebalance(10)
```



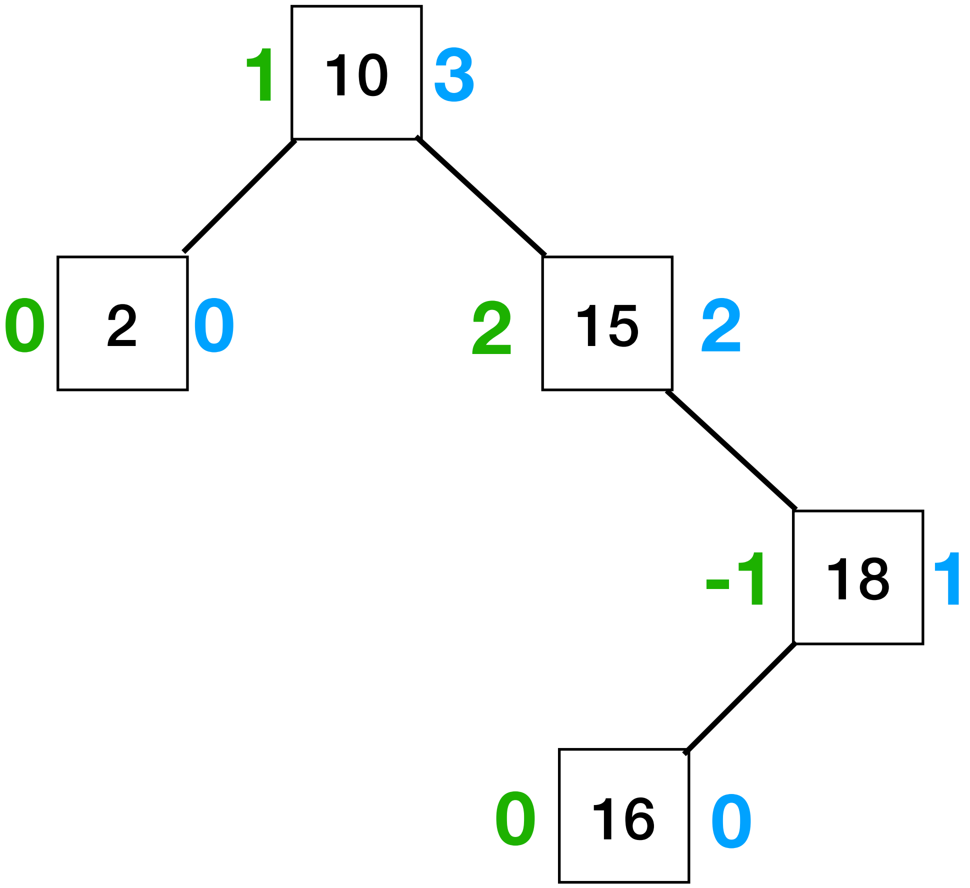
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

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)
```

avlInsert(root, 16)

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

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

rebalance(10)
```



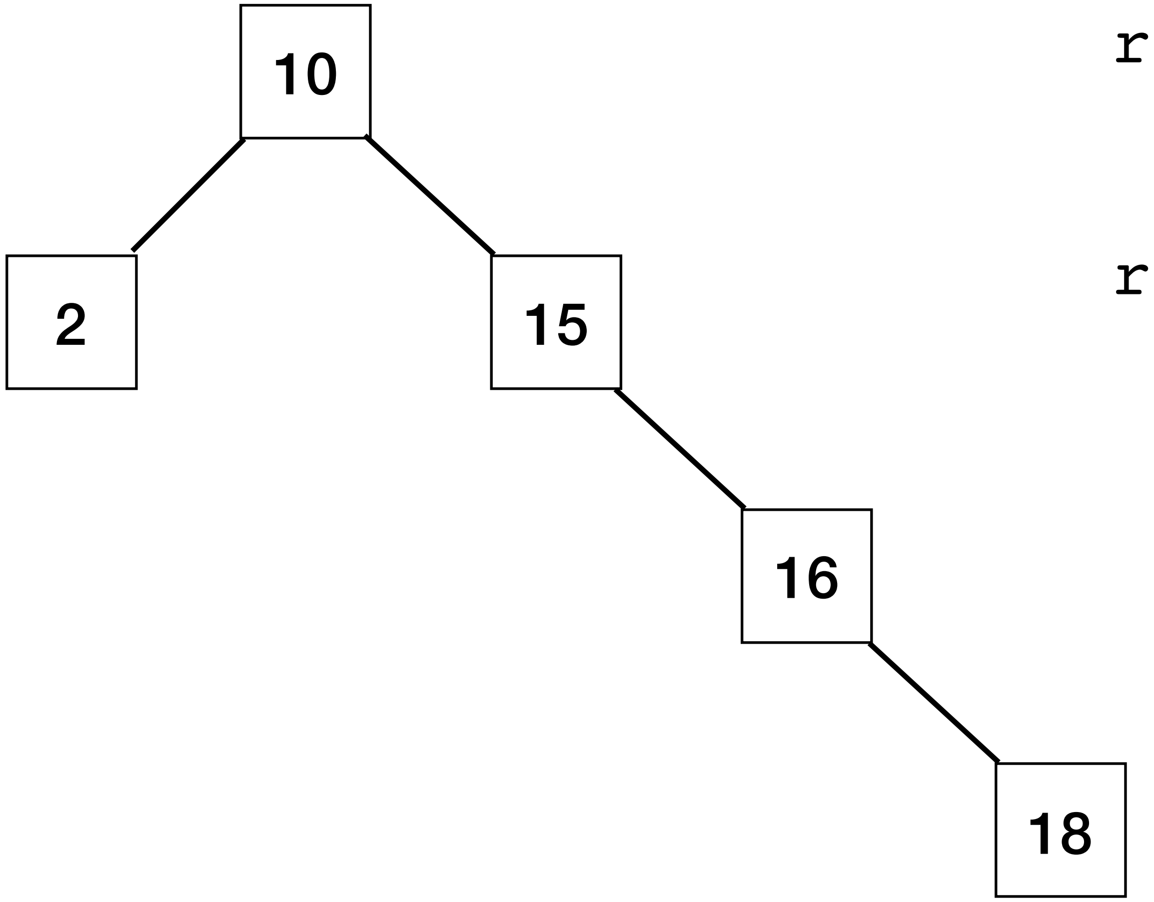
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

Exercise: Draw the tree after rightRotate(18).

```
=> Case 3 (RL):
     rightRotate(18)
     leftRotate(15)
```

avlInsert(root, 16)

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

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

rebalance(10)
```

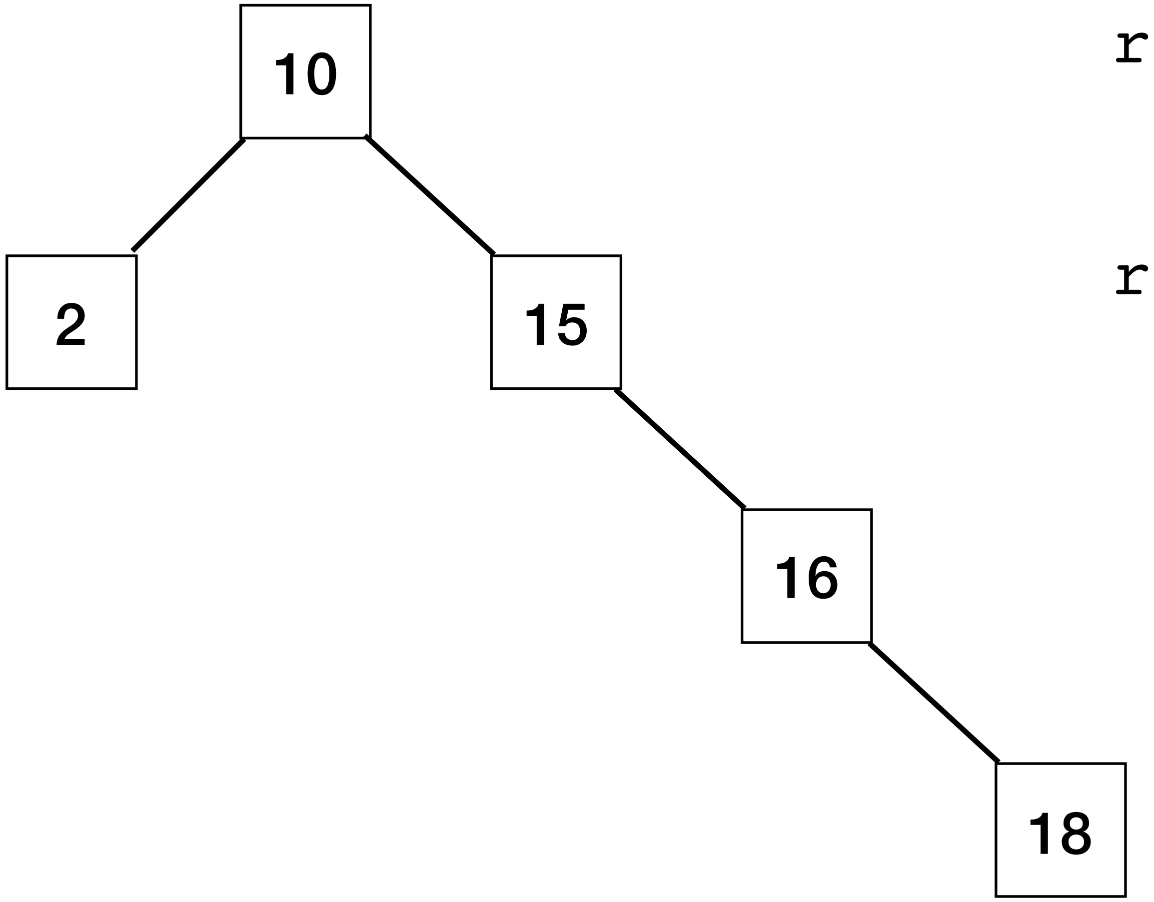


Exercise: Draw the tree after rightRotate(18).

```
=> 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)
```



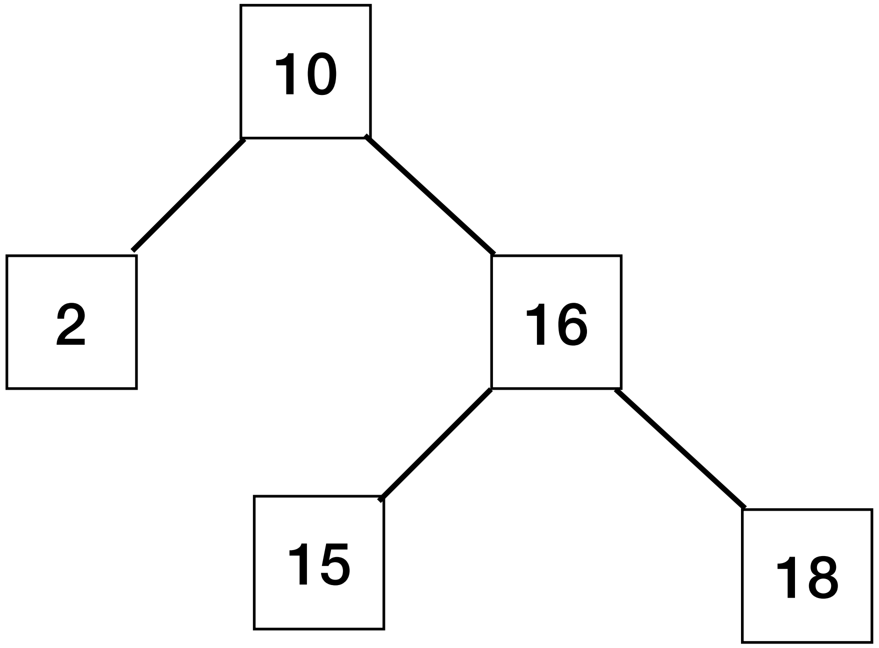
Exercise: Draw the tree after leftRotate(15).

```
=> 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)
```



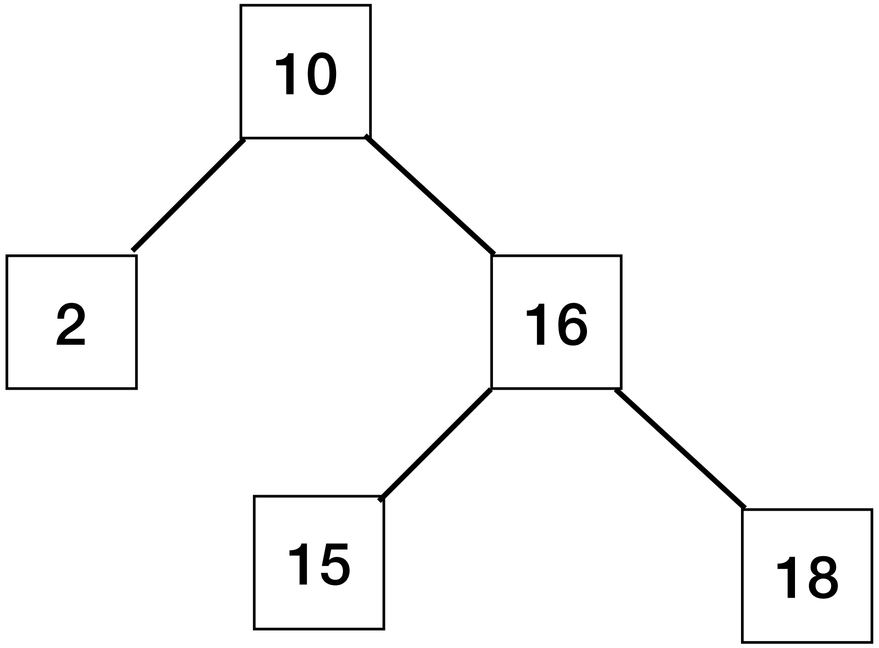
```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

Exercise: Draw the tree after leftRotate(15).

```
=> Case 3 (RL):
     rightRotate(18)
     leftRotate(15)
```

avlInsert(root, 16)

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

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

rebalance(10)
```

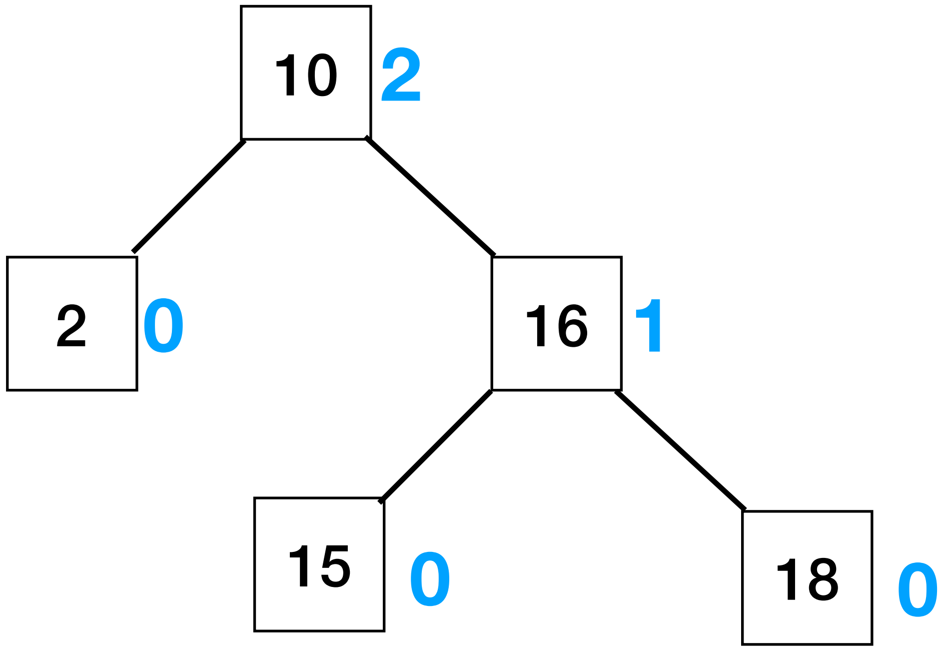


```
void rebalance(n):
  if bal(n) < -1:
    if bal(n.left) < 0
      // case 1:
      // rightRot(n)
    else:
      // case 2:
      // leftRot(n.L);
      // rightRot(n)
  else if bal(n) > 1:
    if bal(n.right) < 0:
      // case 3:
      // rightRot(n.R);
      // leftRot(n)
    else:
      // case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

Exercise: Recompute heights.

avlInsert(root, 16)

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

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

rebalance(10)
```

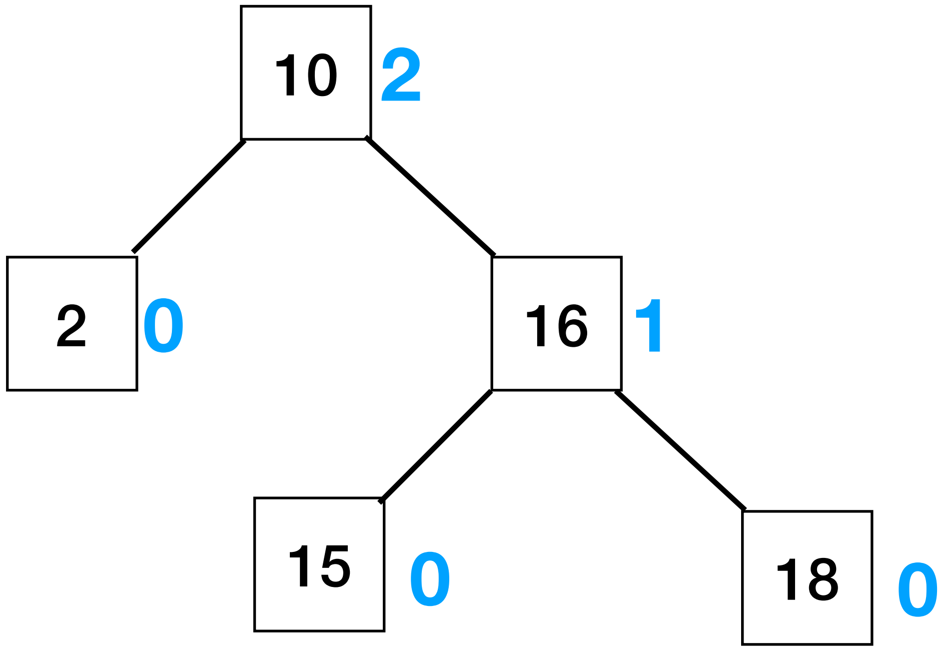


Exercise: Recompute heights.

```
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)
```

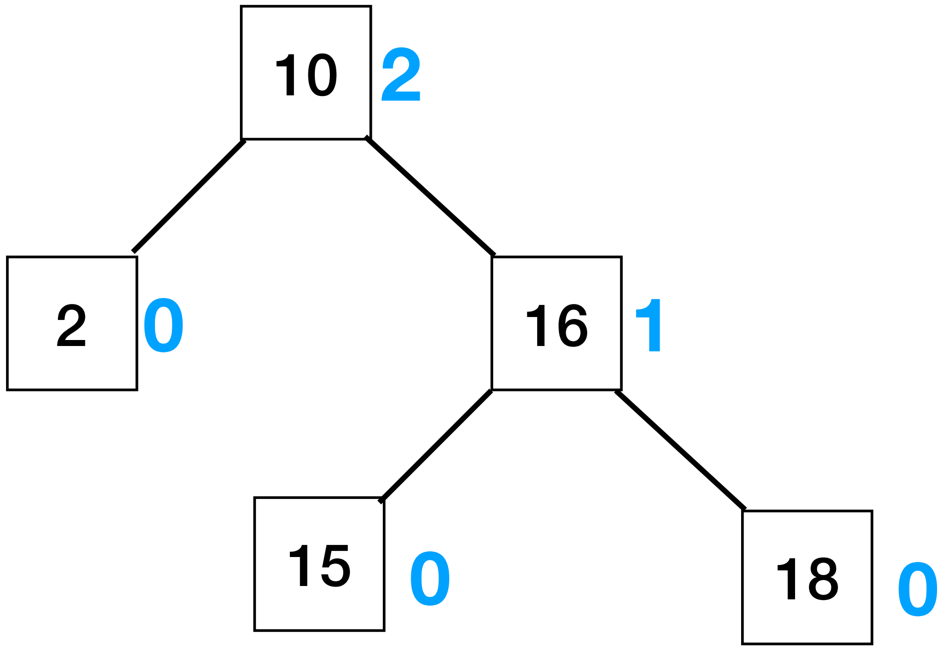


Exercise: What happens when we call rebalance(10)?

```
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
```



```
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)
```

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

Maintaining Height

- Computing height by recursively walking the tree is O(n).
- Doing this every time would ruin our O(log n) runtime of AVL insertion!
- Each node needs to keep track of its height.

Maintaining Height

- Each node needs to keep track of its height.
  - **Exercise**: When can a node’s height change?

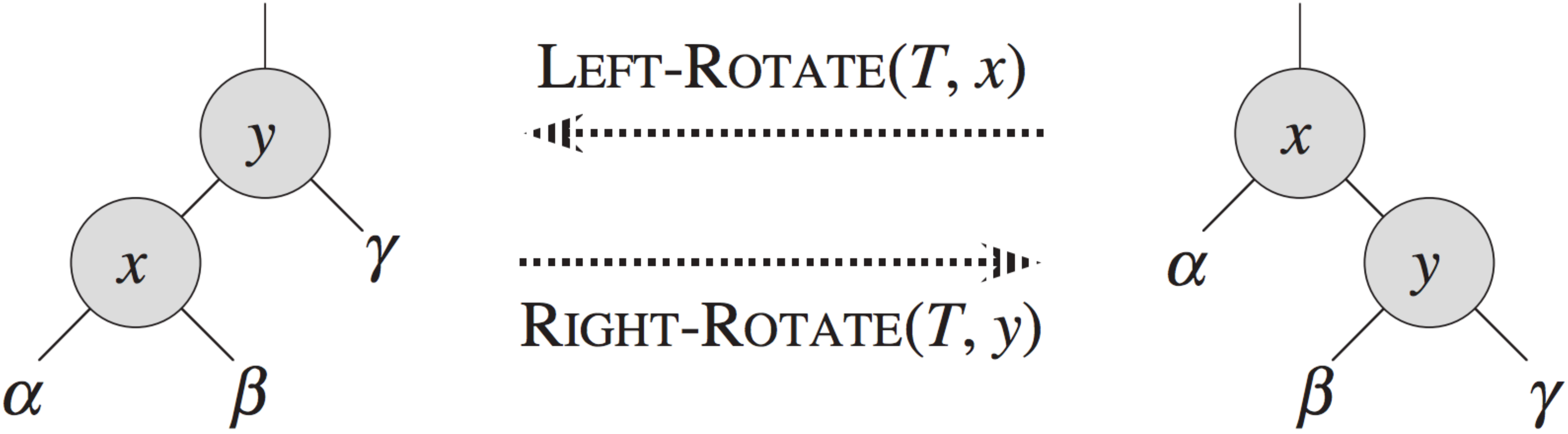
Maintaining Height

- Each node needs to keep track of its height.
  - **Exercise**: When can a node’s height change?
    - After an insertion
    - After a rotation

Maintaining Height

- Each node needs to keep track of its height.
  - **Exercise**: When can a node's height change?
    - After an insertion
    - After a rotation

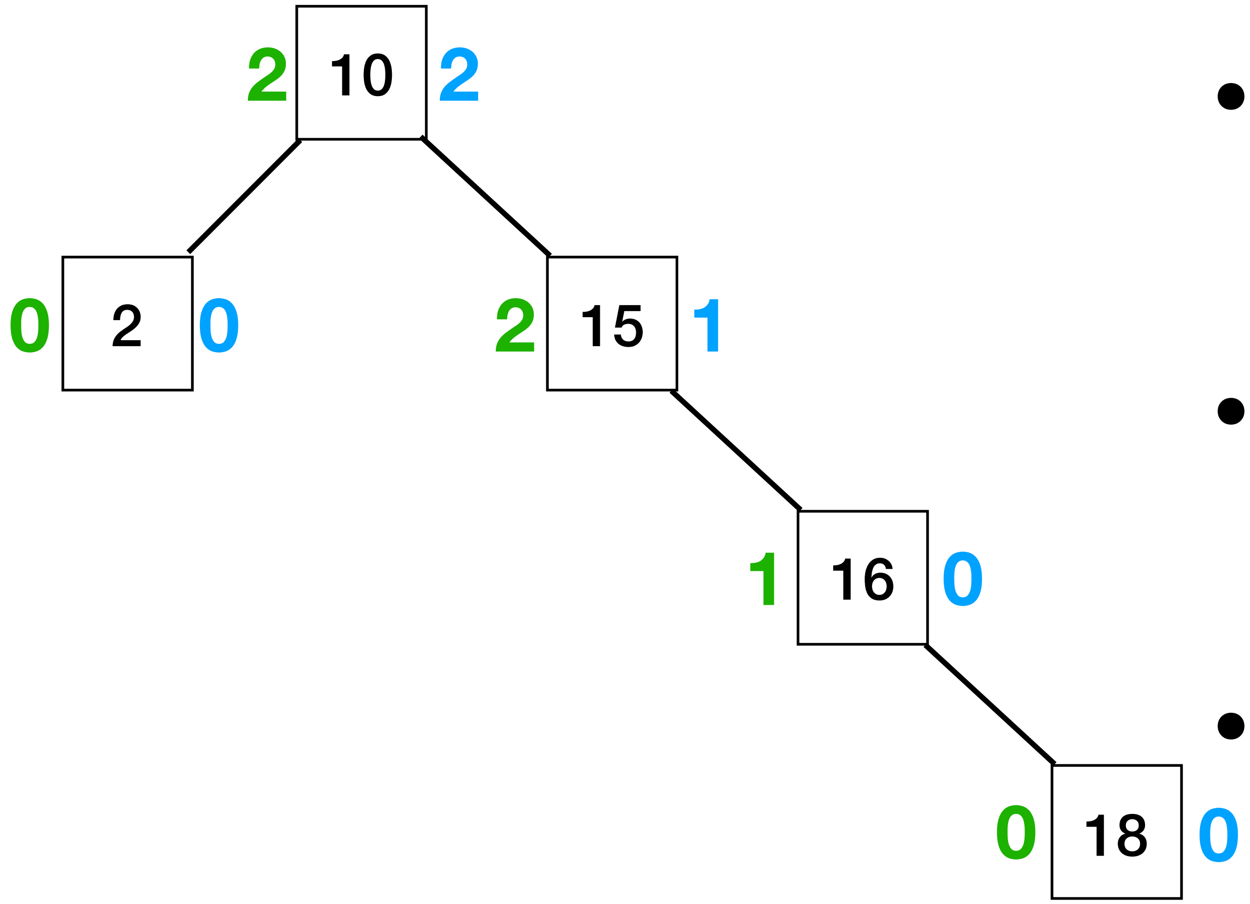
Height after rotations



- Heights of child subtrees (alpha, beta, gamma) can't change.

- Heights of x and y change, but can be calculated **directly from heights of children**.

Height after insertion



- After insertion, all nodes on path to root need to have height updated.

- Fortunately, we're calling rebalance on exactly those nodes!

- Because we're walking up from a leaf, the child's height is already up-to-date.

- A node's height update can safely be computed from the child heights.

Height after insertion