Balanced Binary Search Trees (BBSTs)

William Fiset

What is a BBST?

A **Balanced Binary Search Tree (BBST)** is a **self-balancing** binary search tree. This type of tree will adjust itself in order to maintain a low (logarithmic) height allowing for faster operations such as insertions and deletions.

Complexity of Binary Search Trees

Operation	Average	Worst
Insert	O(log(n))	O(n)
Delete	O(log(n))	O(n)
Remove	O(log(n))	O(n)
Search	O(log(n))	O(n)

Complexity of Balanced Binary Search Trees

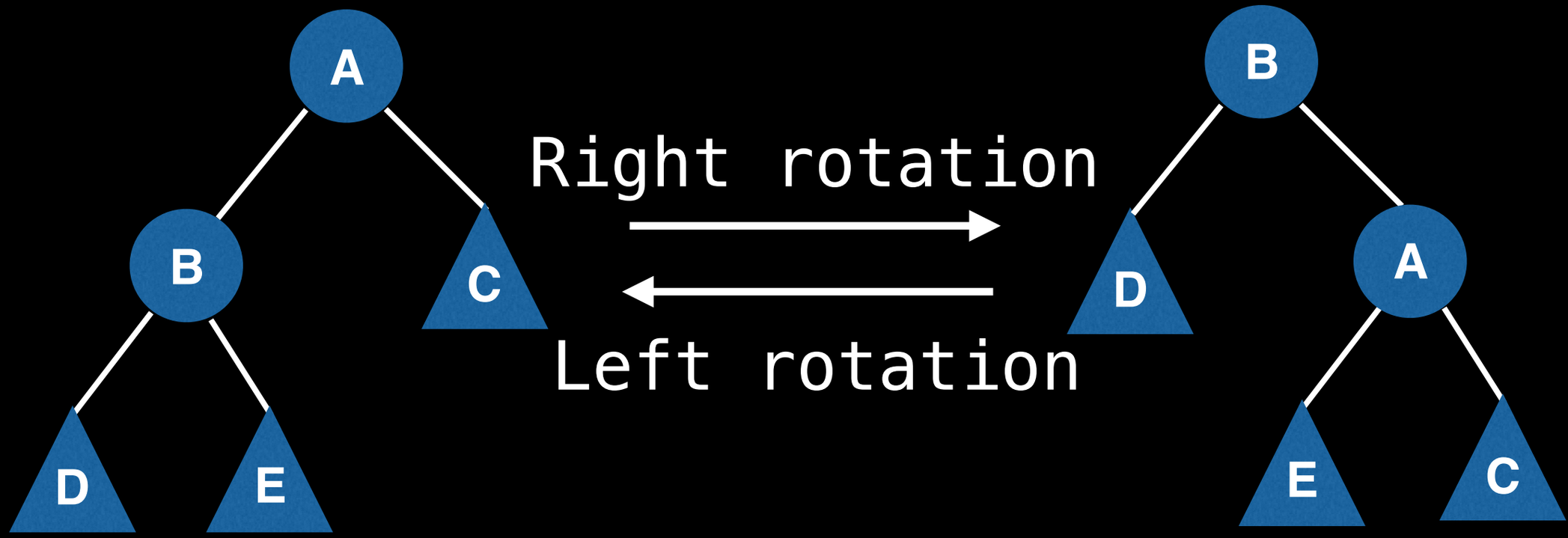
Operation	Average	Worst
Insert	O(log(n))	O(log(n))
Delete	O(log(n))	O(log(n))
Remove	O(log(n))	O(log(n))
Search	O(log(n))	O(log(n))

Tree Rotations!

Tree rotations

The secret ingredient to most BBST algorithms is the clever usage of a **tree invariant** and **tree rotations**.

A tree invariant is a property/rule you impose on your tree that it must meet after every operation. To ensure that the invariant is always satisfied a series of tree rotations are normally applied.



Q: Why does this work? Why are you allowed to change the structure of a tree like this?

Short answer: In the left tree we know that D < B < E < A < C and this remains true for the right subtree, so we didn’t break the BST invariant and, therefore, this is a valid transformation.

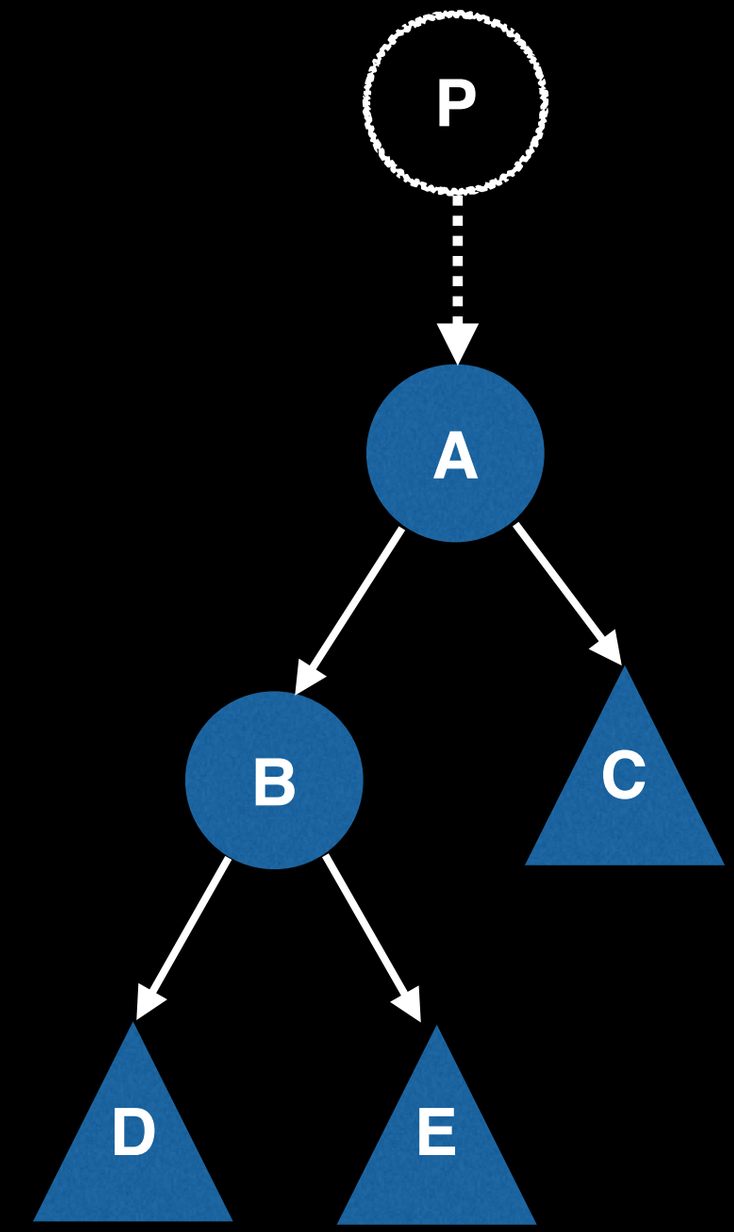
Long answer

Recall that all BBSTs are BSTs so the BST invariant holds. This means that for every node *n*, *n.left* < *n* and *n* < *n.right*.

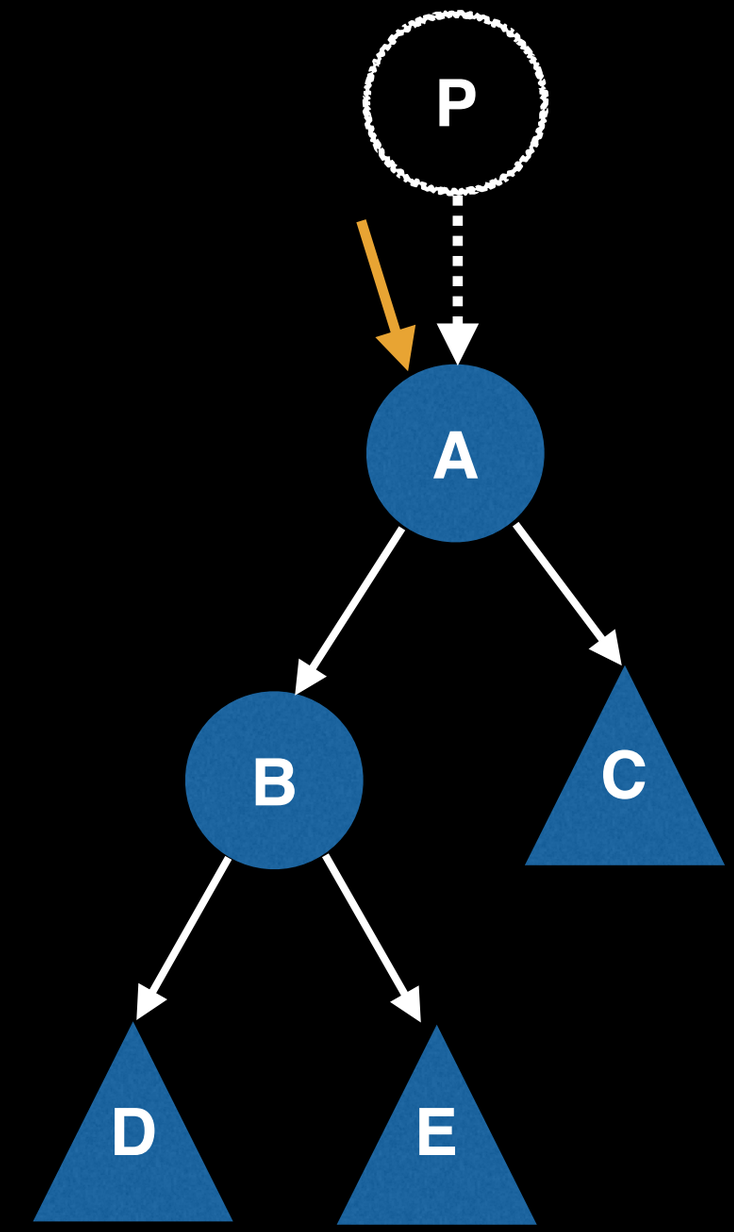
NOTE: The above assumes we only have unique values, otherwise we'd have to consider the case where n.left ≤ n and n ≤ n.right

It does not matter what the structure of the tree looks; all we care about is that the BST invariant holds. This means we can shuffle/ transform/rotate the values and nodes in the tree as we please as long as the BST invariant remains satisfied!

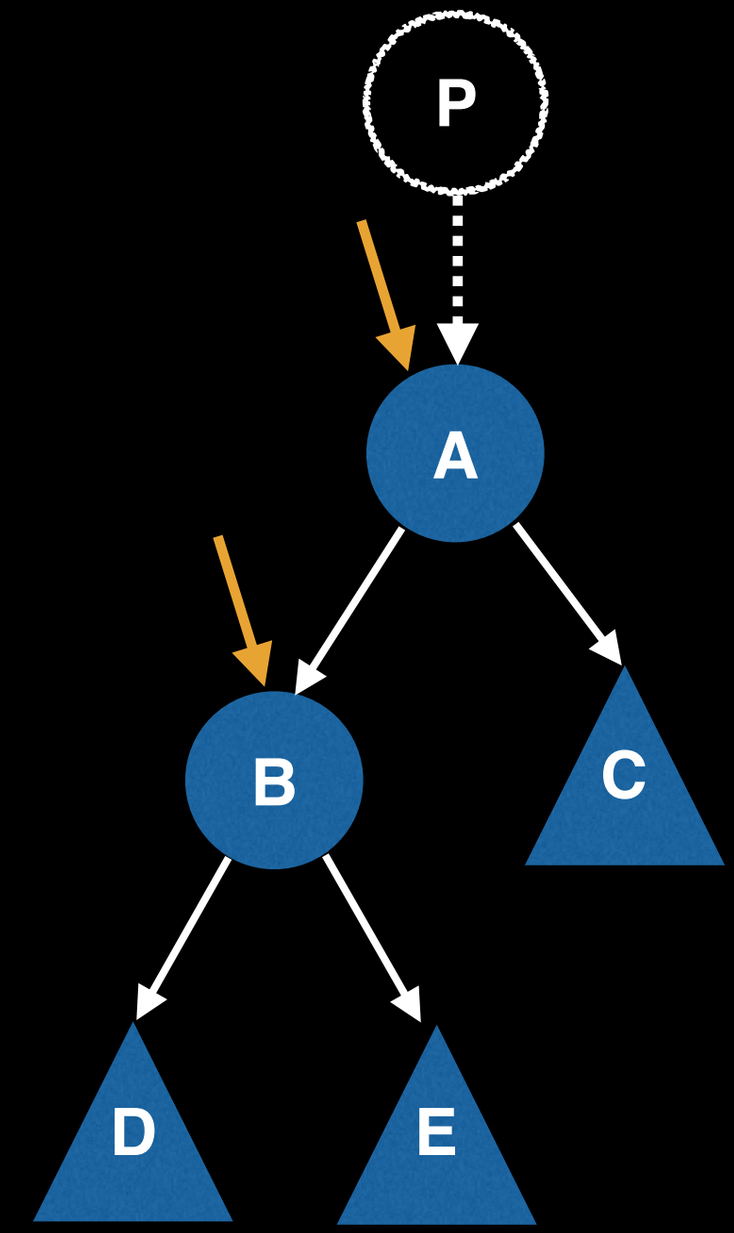
```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```



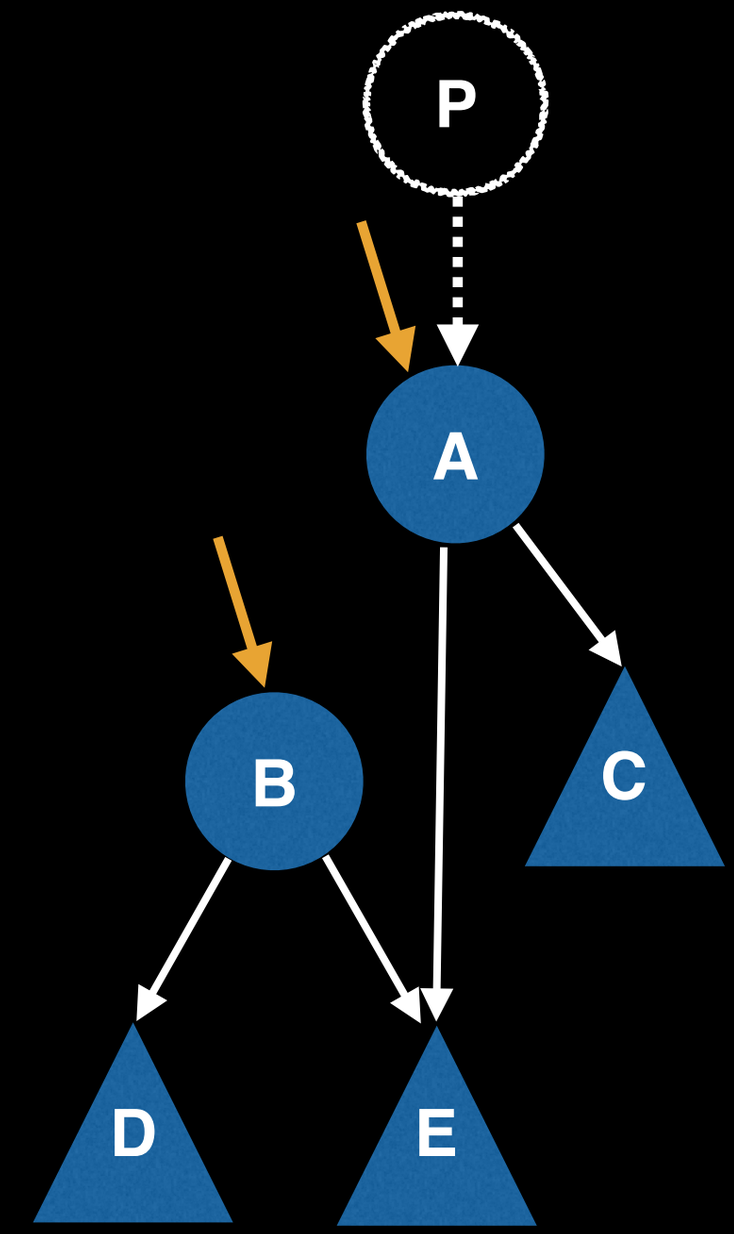
```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```



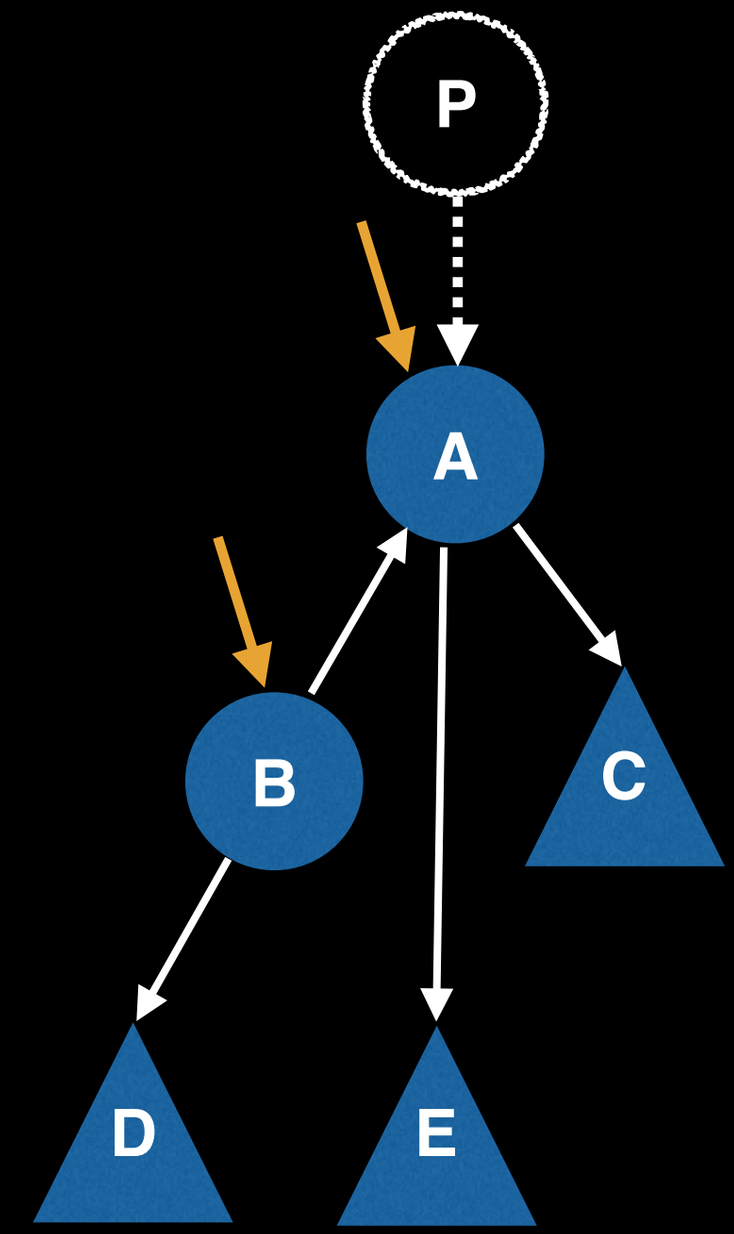
```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```



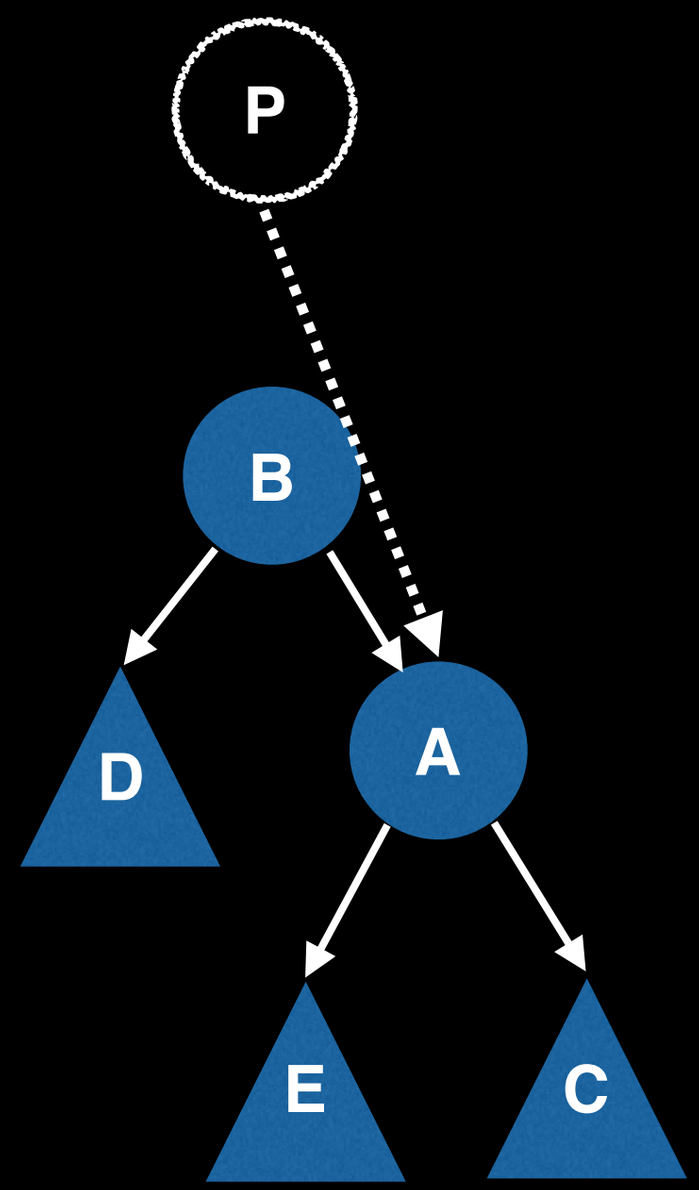
```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```



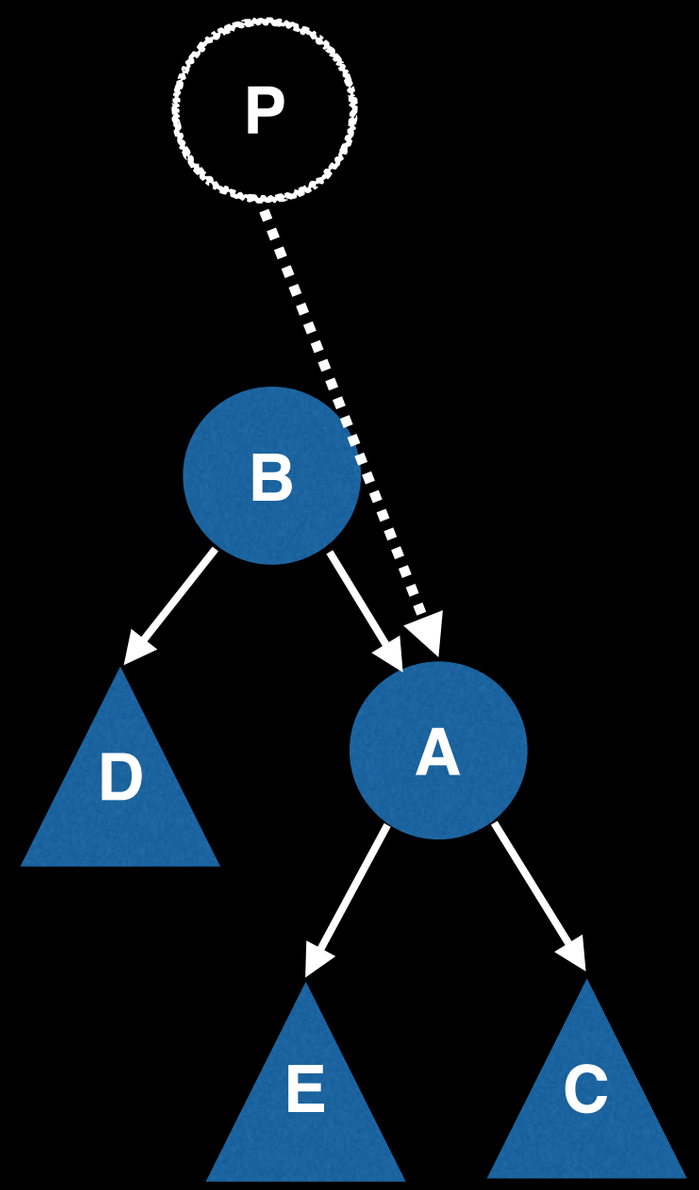
```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```



```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```

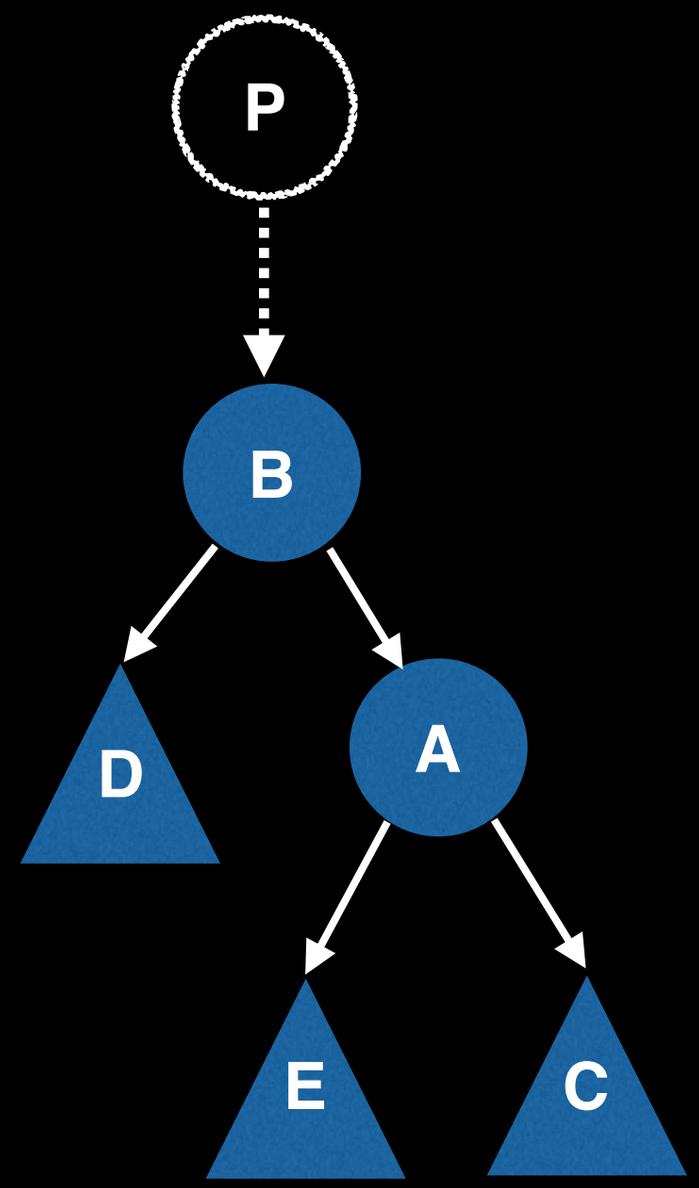


```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```



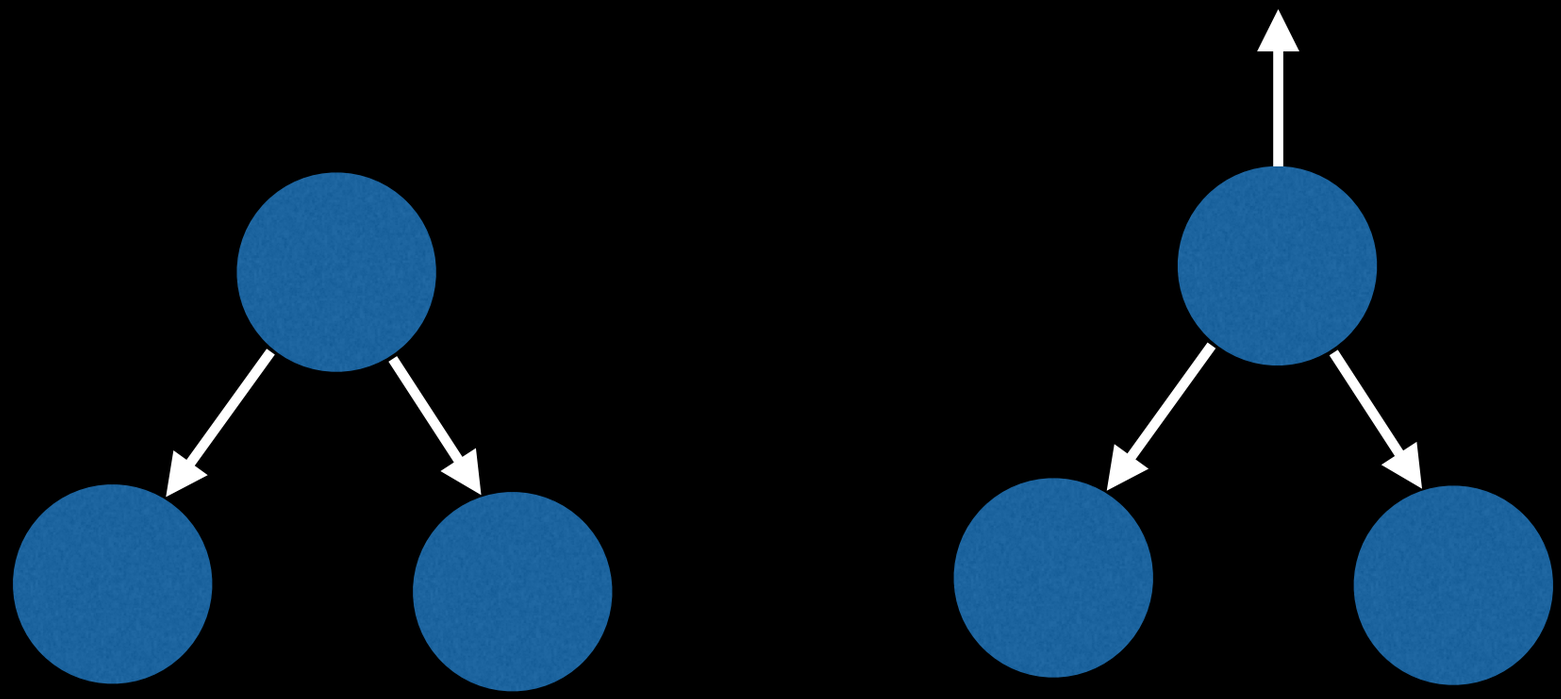
NOTE: It’s possible that before the rotation node A had a parent whose left/right pointer referenced it. It’s very important that this link be updated to reference B. This is usually done on the recursive callback using the return value of *rotateRight*.

```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  return B
```

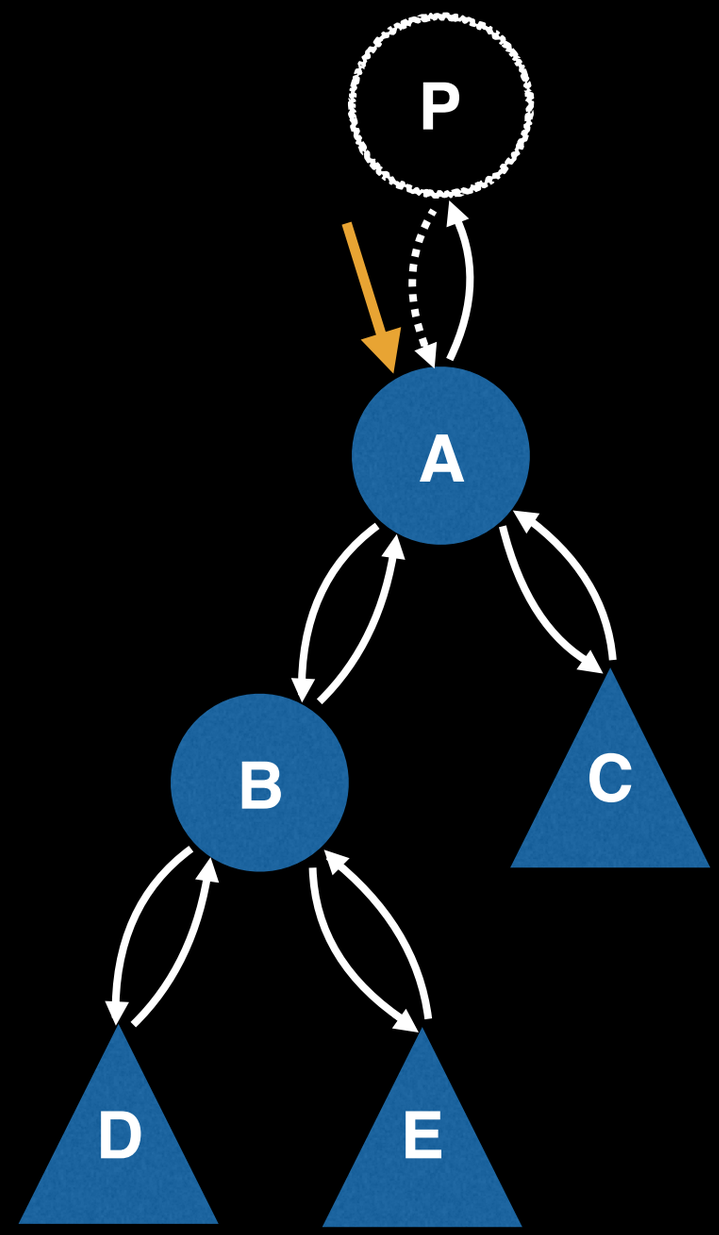


NOTE: It’s possible that before the rotation node A had a parent whose left/right pointer referenced it. It’s very important that this link be updated to reference B. This is usually done on the recursive callback using the return value of *rotateRight*.

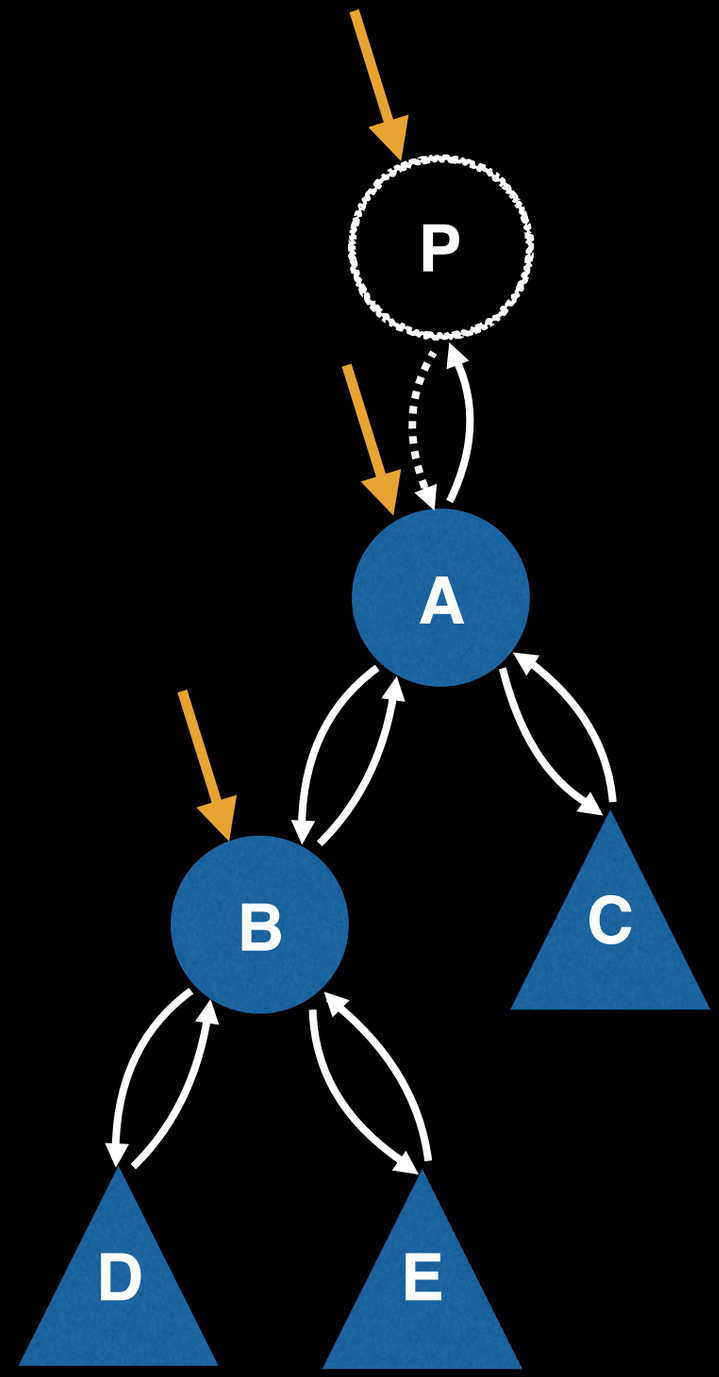
In some BBST implementations where you often need to access the parent/uncle nodes (such as RB trees), it’s convenient for nodes to not only have a reference to the left and the right child nodes but also the parent node. This can complicate tree rotations because instead of updating **three** pointers, now you have to update **six**!



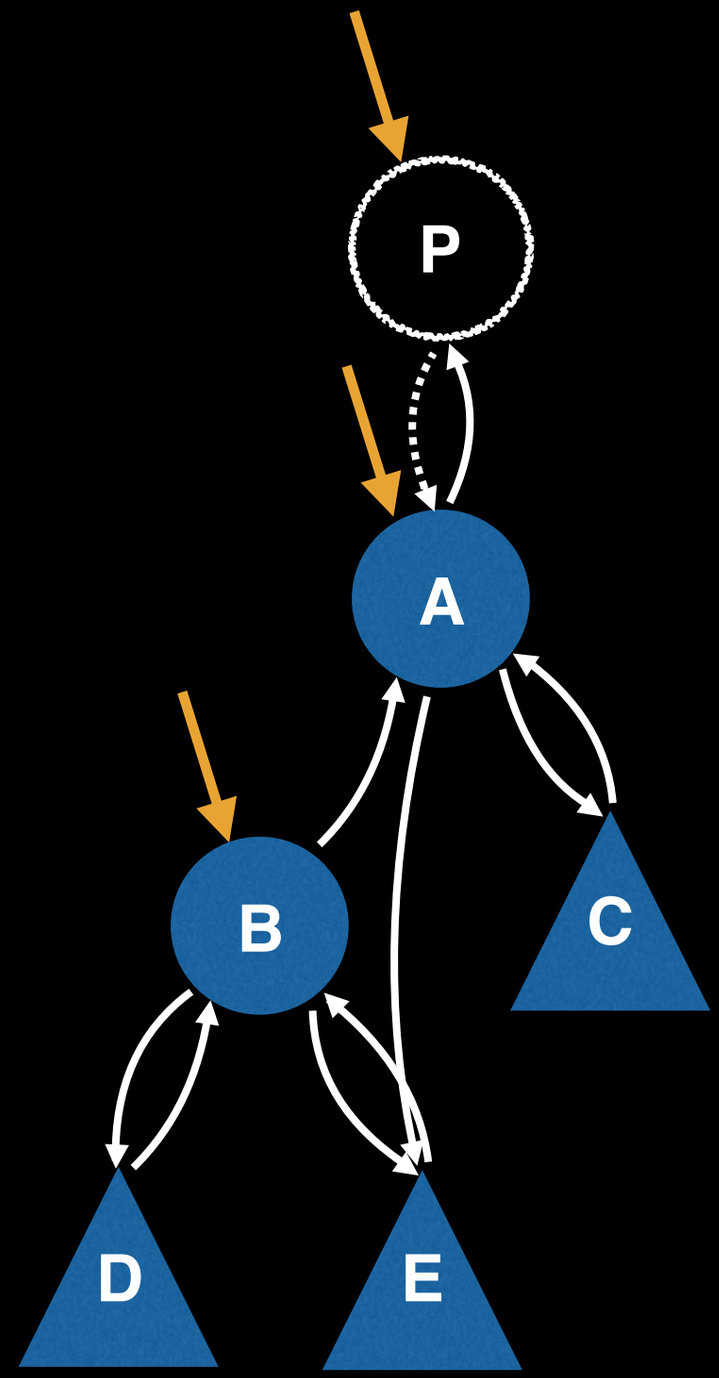
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



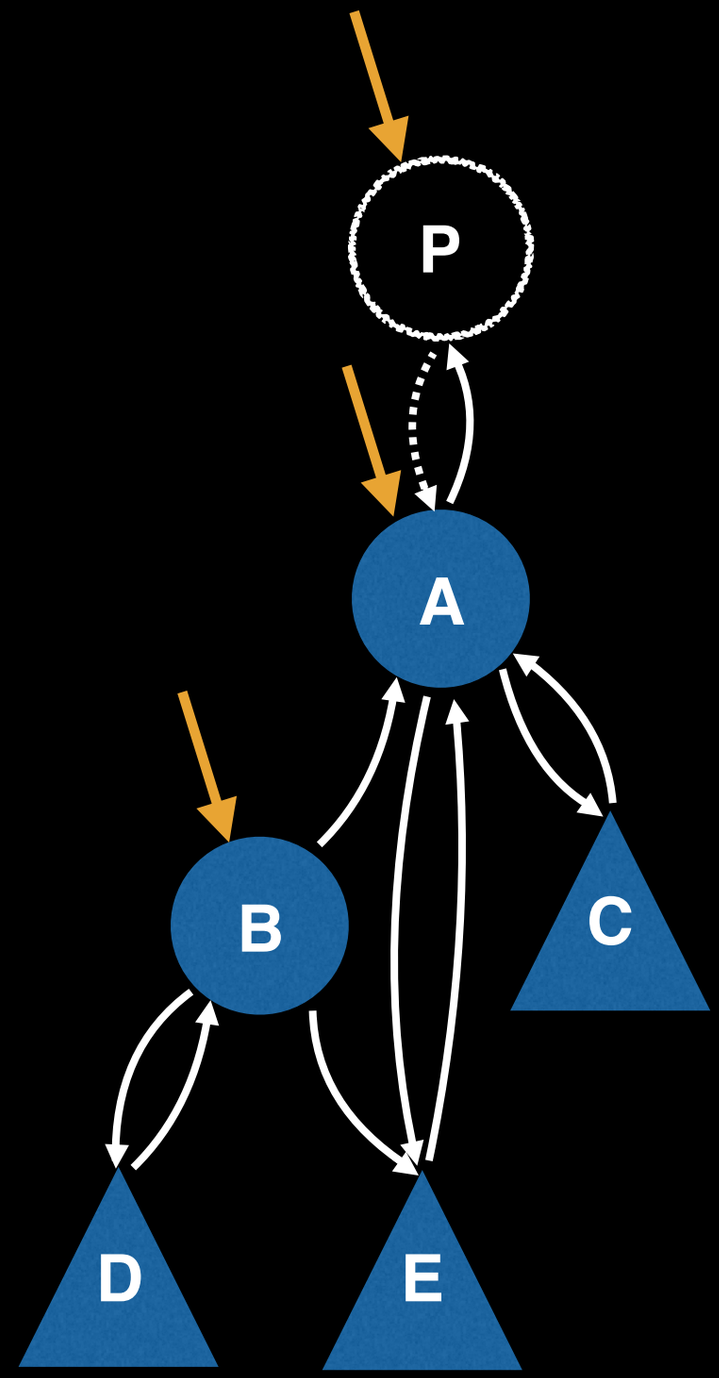
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



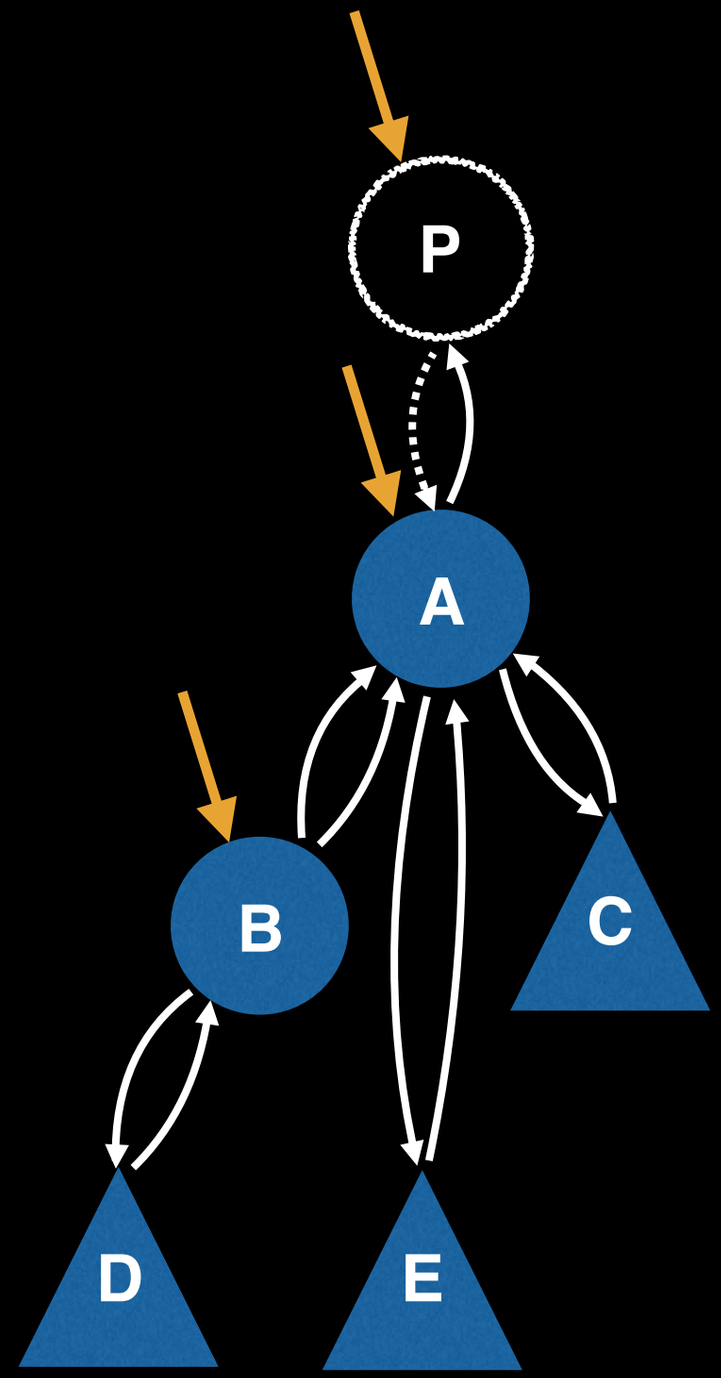
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



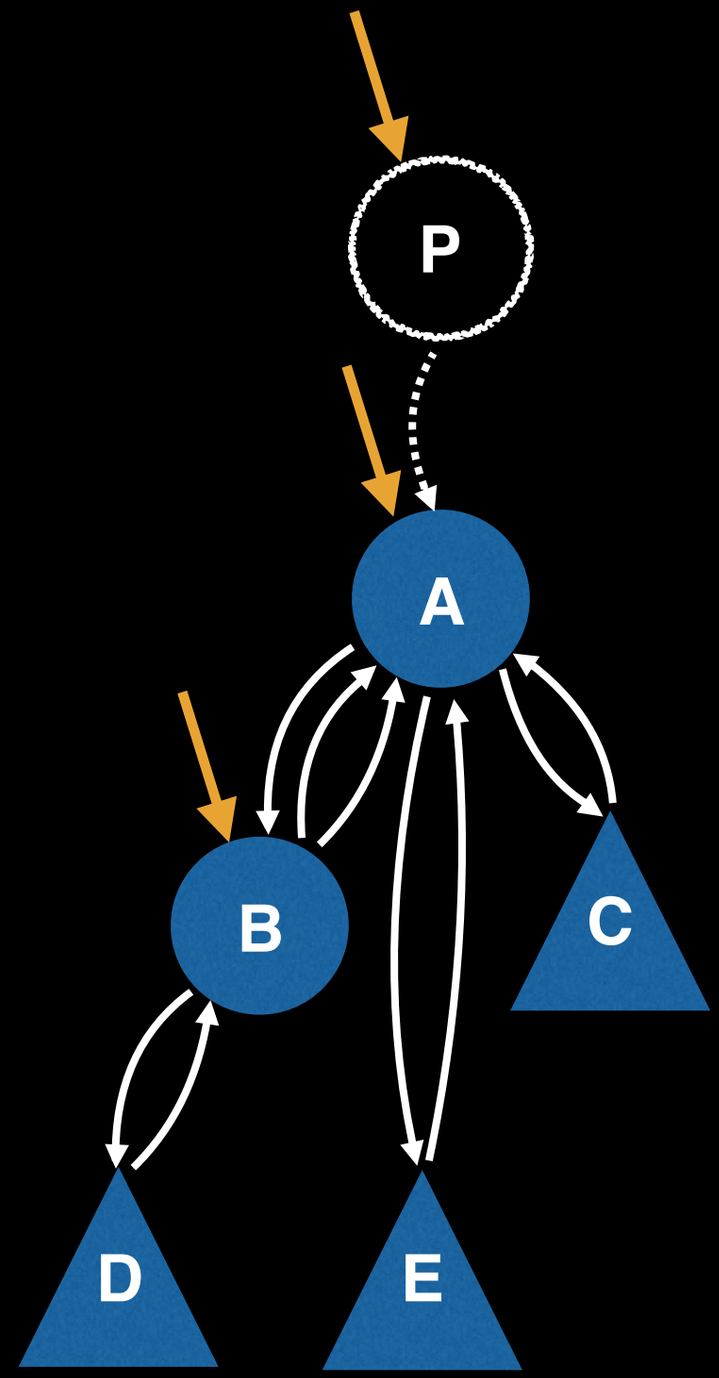
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



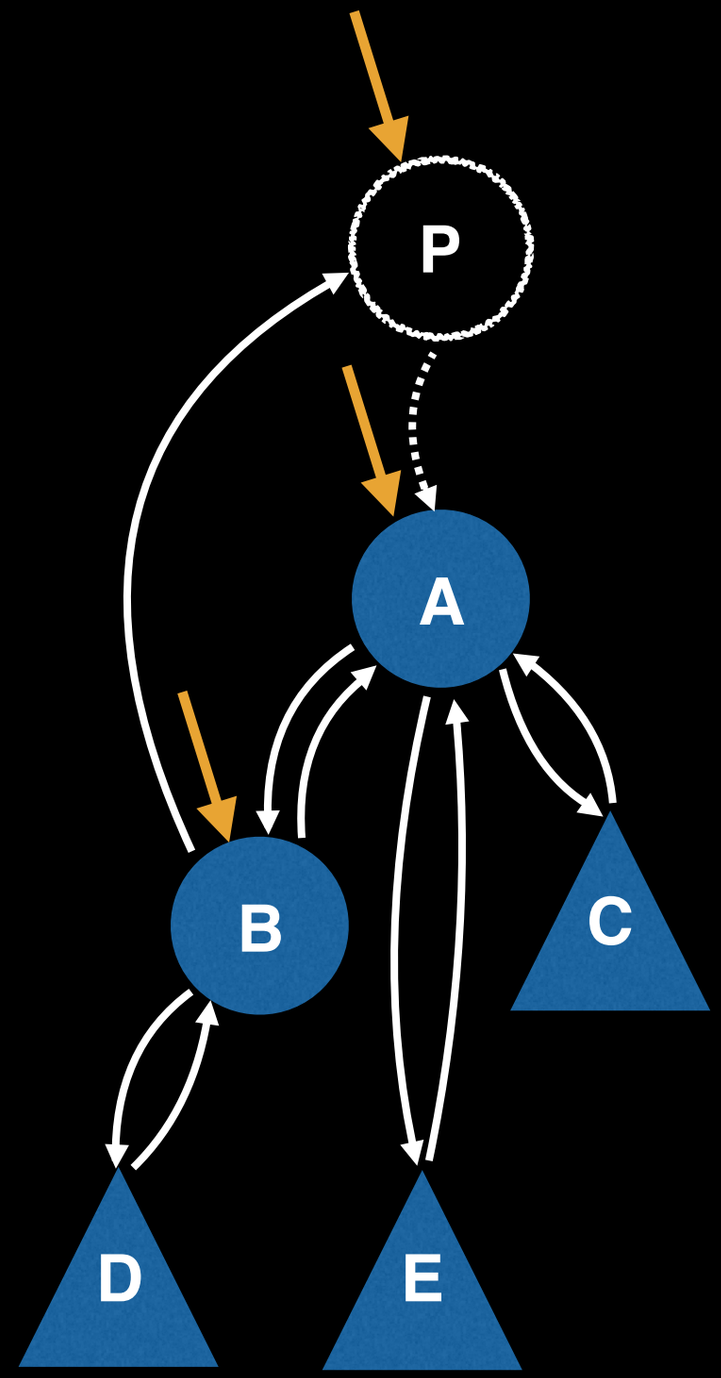
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



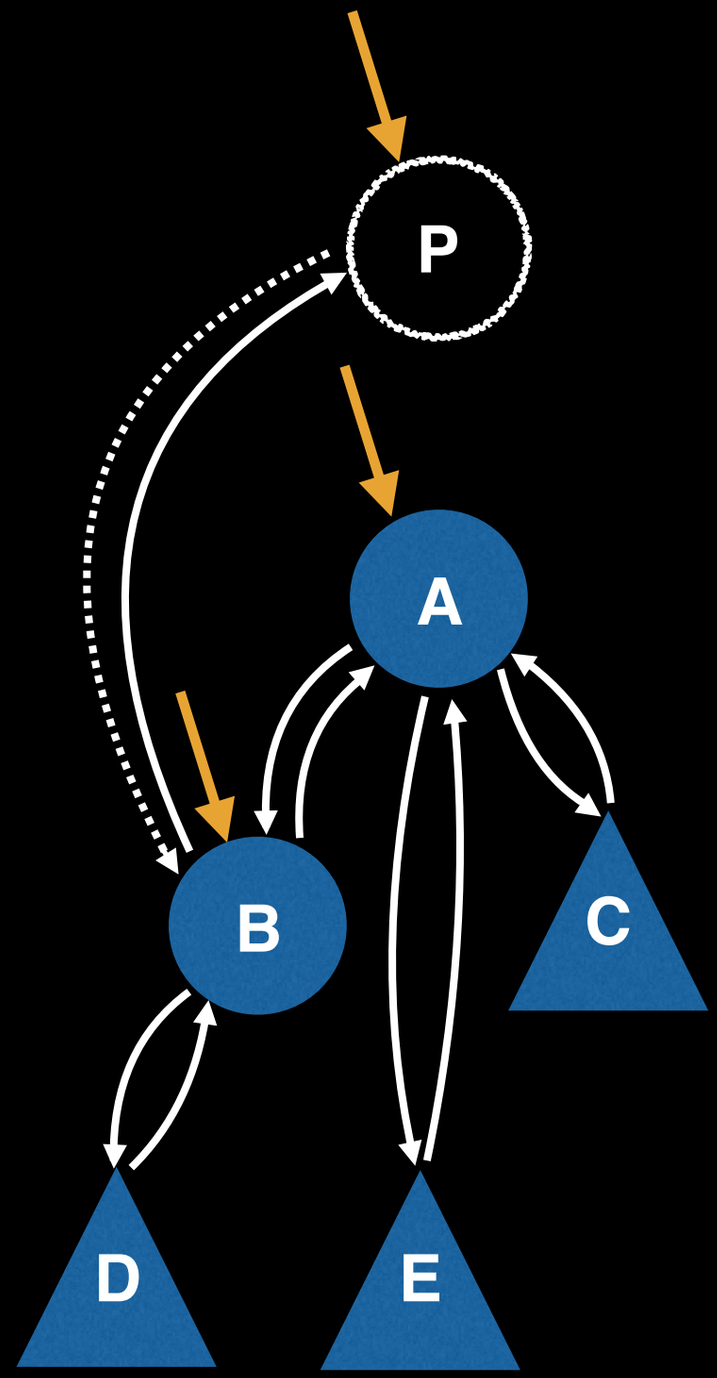
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



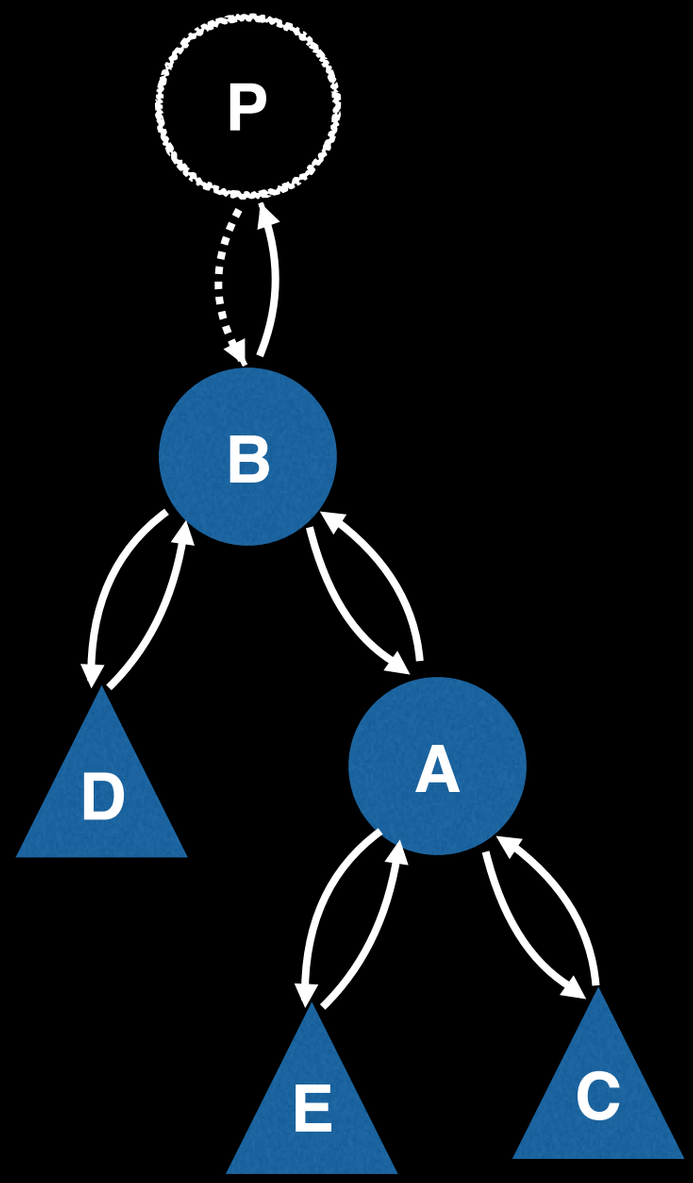
```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



```
function rightRotate(A):
  P := A.parent
  B := A.left
  A.left = B.right
  if B.right != null:
    B.right.parent = A
  B.right = A
  A.parent = B
  B.parent = P
  # Update parent down link.
  if P != null:
    if P.left == A:
      P.left = B
    else:
      P.right = B
  return B
```



Next Video:
AVL Tree Insertion

Summary: BBSTs remain balanced by performing a series of left/right tree rotations when their invariant is not satisfied.

Inserting Elements into an AVL Tree

William Fiset

AVL Tree Introduction

An **AVL tree** is one of many types of **Balanced Binary Search Trees (BBSTs)** which allow for logarithmic **O(log(n))** insertion, deletion and search operations.

In fact, it was the first type of BBST to be discovered. Soon after, many other types of BBSTs started to emerge including the 2–3 tree, the AA tree, the scapegoat tree, and its main rival, the red–black tree.

AVL Tree Invariant

The property which keeps an AVL tree balanced is called the **Balanced Factor (BF)**.

BF(node) = **H**(node.right) - **H**(node.left)

Where **H**(x) is the height of node x. Recall that **H**(x) is calculated as the **number of edges** between x and the furthest leaf.

The invariant in the AVL which forces it to remain balanced is the requirement that the balance factor is always either -1, 0 or +1.

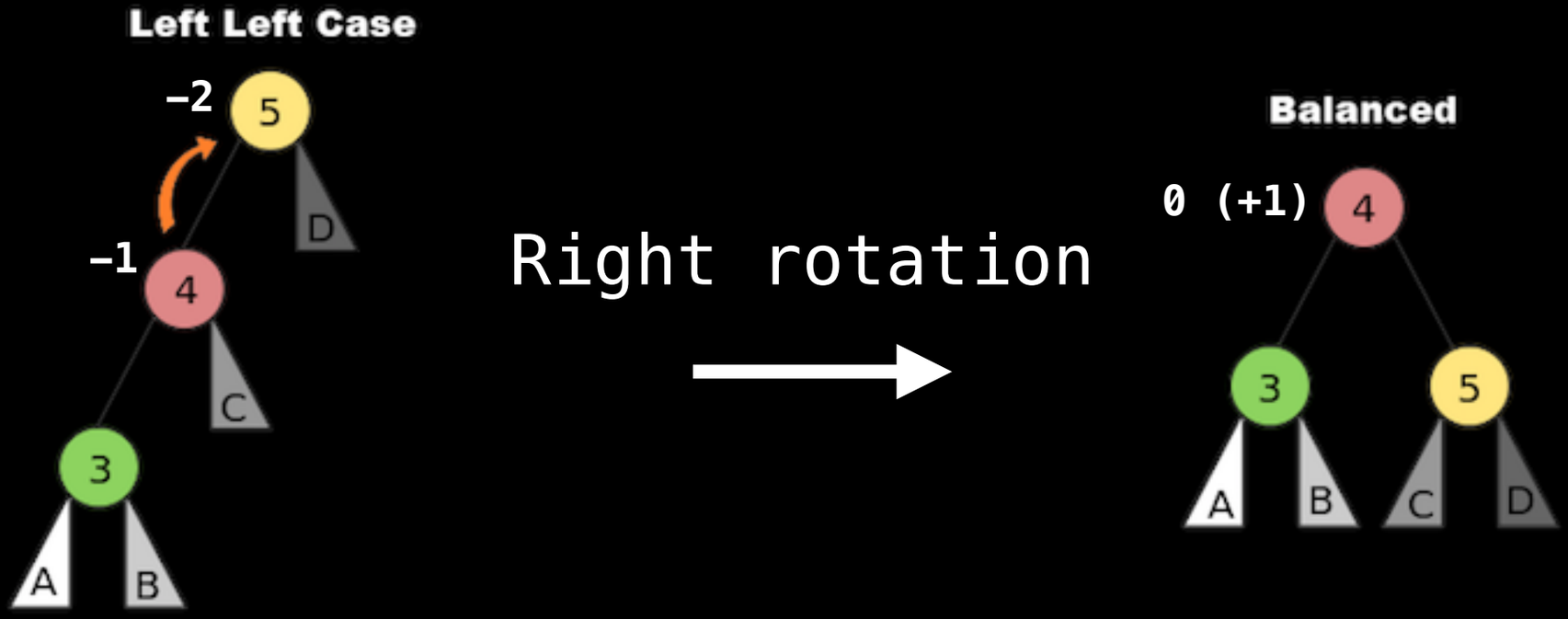
Node Information to Store

- The actual value we're storing in the node. **NOTE:** This value must be comparable so we know how to insert it.
- A value storing this node's **balance factor**.
- The **height** of this node in the tree.
- Pointers to the **left/right child nodes**.

Q: What if the **BF** of a node is ∉ {−1, 0, +1}?
How do we restore the AVL tree invariant?

A: If a node's **BF** ∉ {−1, 0, +1} then the **BF** of that node is ±2 which can be adjusted using **tree rotations**.

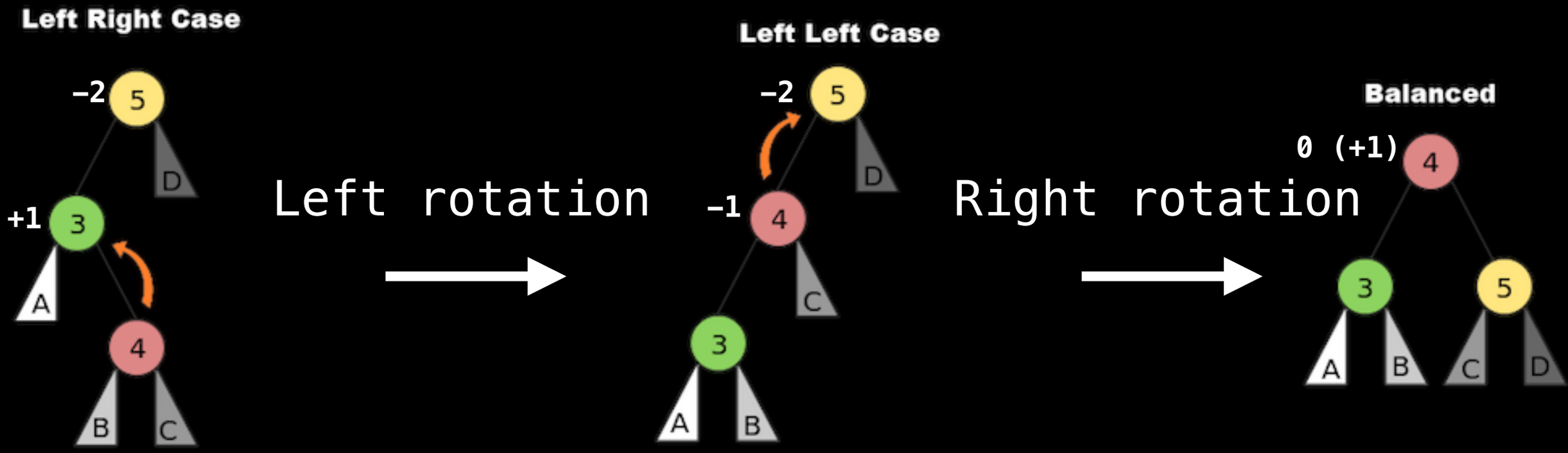
Recall: BF(node) = **H**(node.right) − **H**(node.left)



Q: What if the **BF** of a node is ∉ {−1, 0, +1}?
How do we restore the AVL tree invariant?

A: If a node's **BF** ∉ {−1, 0, +1} then the **BF** of that node is ±2 which can be adjusted using **tree rotations**.

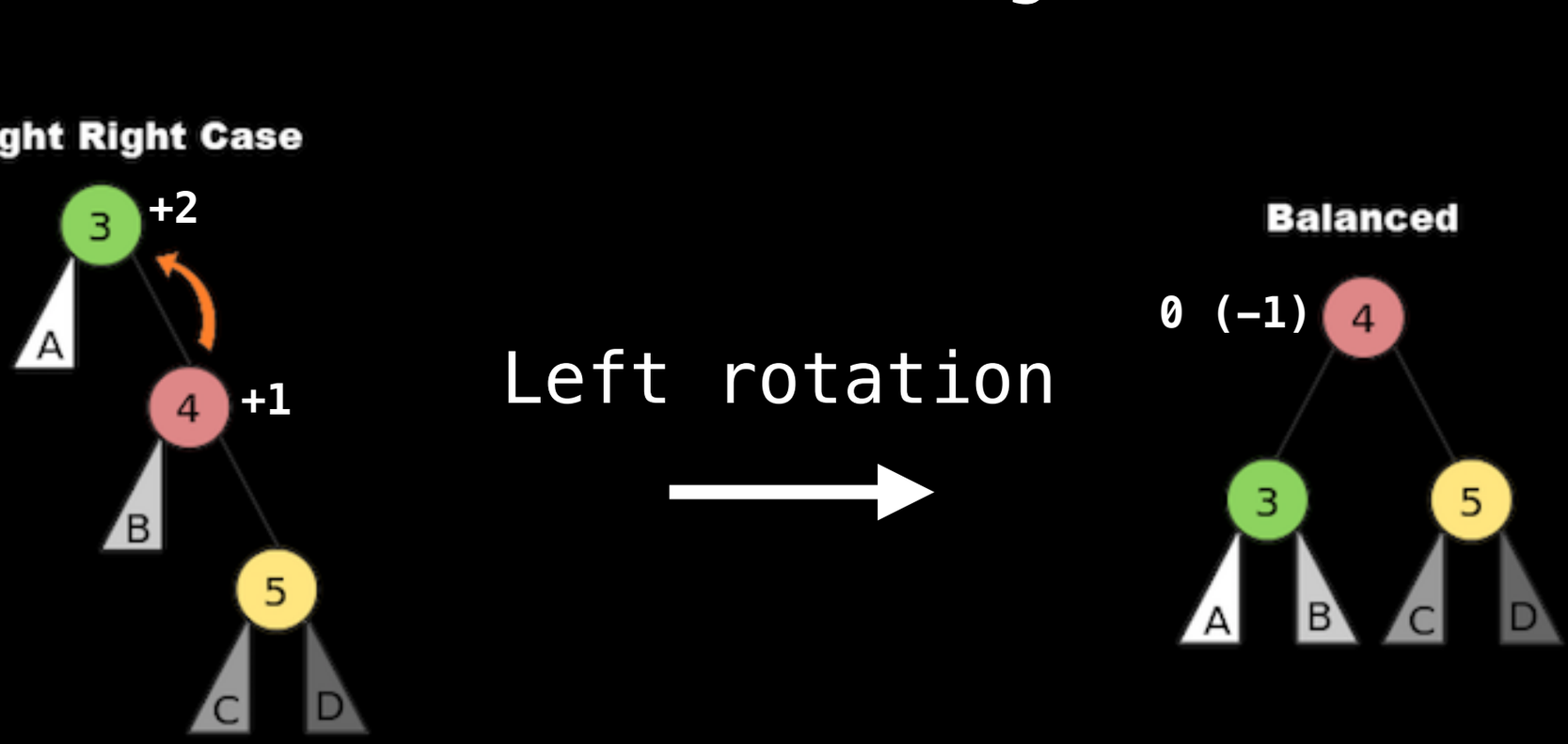
Recall: BF(node) = **H**(node.right) − **H**(node.left)



Q: What if the **BF** of a node is ∉ {−1, 0, +1}?
How do we restore the AVL tree invariant?

A: If a node's **BF** ∉ {−1, 0, +1} then the **BF** of that node is ±2 which can be adjusted using **tree rotations**.

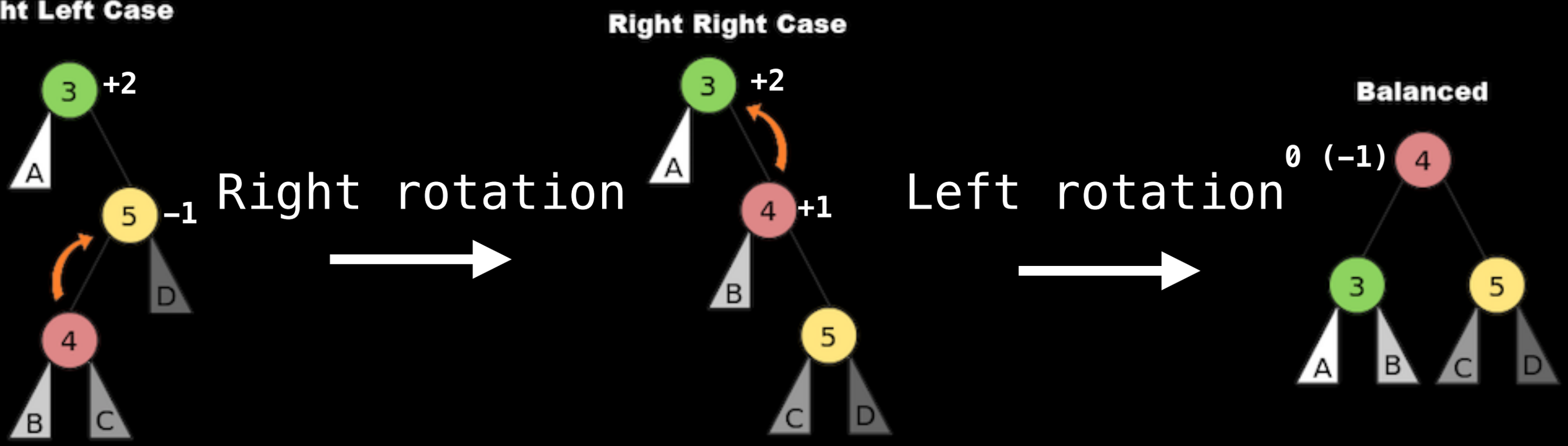
Recall: BF(node) = **H**(node.right) − **H**(node.left)



Q: What if the **BF** of a node is ∉ {−1, 0, +1}?
How do we restore the AVL tree invariant?

A: If a node's **BF** ∉ {−1, 0, +1} then the **BF** of that node is ±2 which can be adjusted using **tree rotations**.

Recall: BF(node) = **H**(node.right) − **H**(node.left)



```
# Public facing insert method. Returns true
# on successful insert and false otherwise.
function insert(value):

    if value == null:
        return false

    # Only insert unique values
    if !contains(root, value):
        root = insert(root, value)
        nodeCount = nodeCount + 1
        return true

    # Value already exists in tree.
    return false
```

```
function insert(node, value):
    if node == null: return Node(value)

    # Invoke the comparator function in whatever
    # programming language you're using.
    cmp := compare(value, node.value)

    if cmp < 0:
        node.left = insert(node.left, value)
    else:
        node.right = insert(node.right, value)

    # Update balance factor and height values.
    update(node)

    # Rebalance tree
    return balance(node)
```

```
function update(node):

  # Variables for left/right subtree heights
  lh := -1
  rh := -1
  if node.left  != null: lh = node.left.height
  if node.right != null: rh = node.right.height

  # Update this node's height.
  node.height = 1 + max(lh, rh)

  # Update balance factor.
  node.bf = rh - lh
```

```
function balance(node):
  # Left heavy subtree.
  if node.bf == -2:
    if node.left.bf <= 0:
      return leftLeftCase(node)
    else:
      return leftRightCase(node)

  # Right heavy subtree.
  else if node.bf == +2:
    if node.right.bf >= 0:
      return rightRightCase(node)
    else:
      return rightLeftCase(node)

  # Node has balance factor of -1, 0 or +1
  # which we do not need to balance.
  return node
```

```
function leftLeftCase(node):
  return rightRotation(node)


function leftRightCase(node):
  node.left = leftRotation(node.left)
  return leftLeftCase(node)
```

```
function rightRightCase(node):
  return leftRotation(node)


function rightLeftCase(node):
  node.right = rightRotation(node.right)
  return rightRightCase(node)
```

AVL Tree Rotation Method

```
function rightRotate(A):
  B := A.left
  A.left = B.right
  B.right = A
  # After rotation update balance
  # factor and height values.
  update(A)
  update(B)
  return B
```

AVL tree rotations require you to call the **update method**! The left rotation is symmetric.

Next Video: AVL Tree Removals

Source code for the AVL tree can be found at:

https://github.com/williamfiset/data-structures

Removing Elements from an AVL Tree

William Fiset

Removing Elements from a BST

Removing elements from a Binary Search Tree (BST) can be seen as a two-step process:

1) **Find** the element we wish to remove (if it exists).

2) **Replace** the node we want to remove with its successor (if any) to maintain the BST invariant.

*Recall the **BST invariant**: left subtree has smaller elements and right subtree has larger elements.*

Find Phase

When searching our BST for a node with a particular value, one of four things will happen:

1) We hit a **null node** at which point we know the value does not exist within our BST

2) Comparator value **equal to 0** (found it!)

3) Comparator value **less than 0** (the value, if it exists, is in the left subtree)

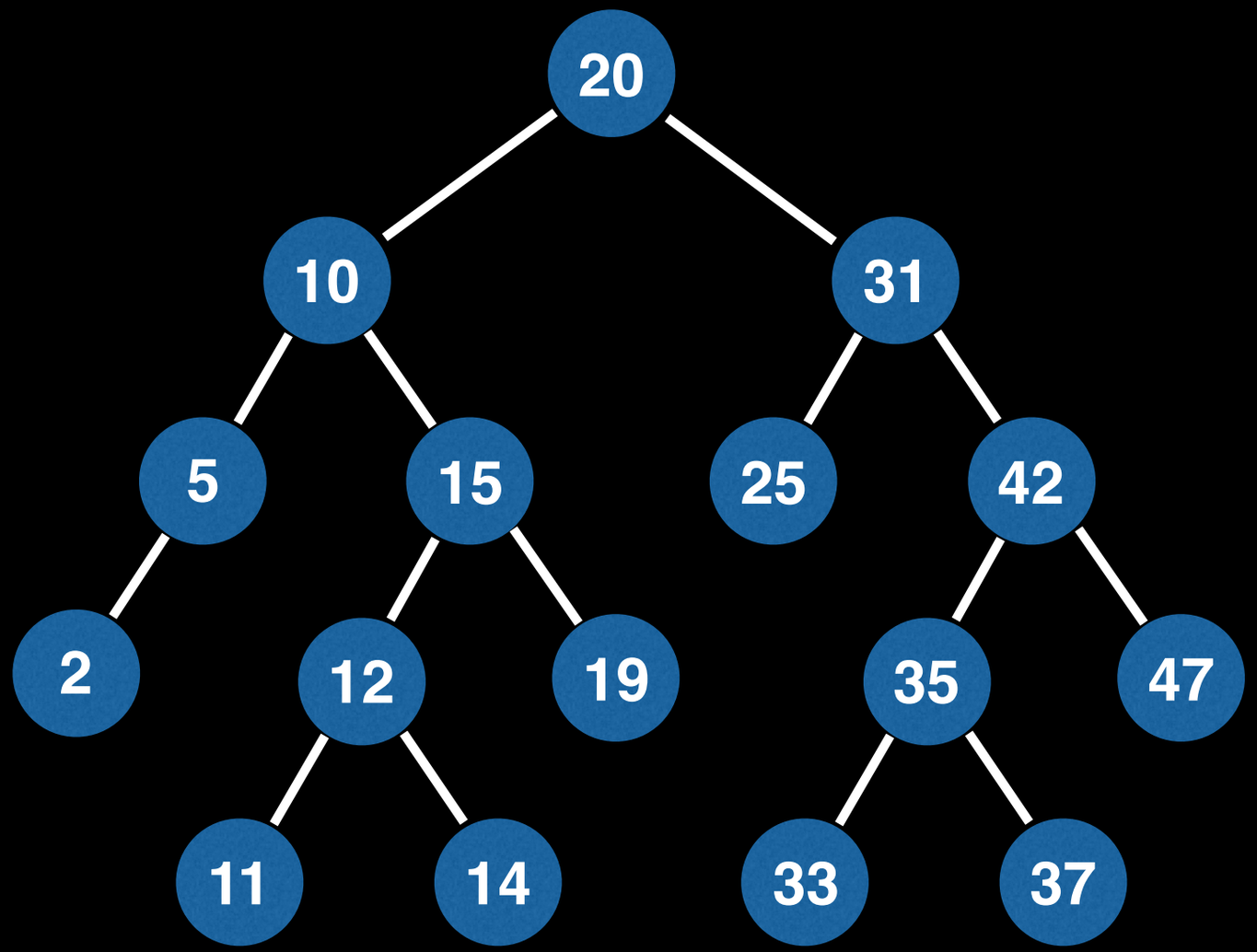
4) Comparator value **greater than 0** (the value, if it exists, is in the right subtree)

Find Phase

Find queries:

find(14) ←

find(26)

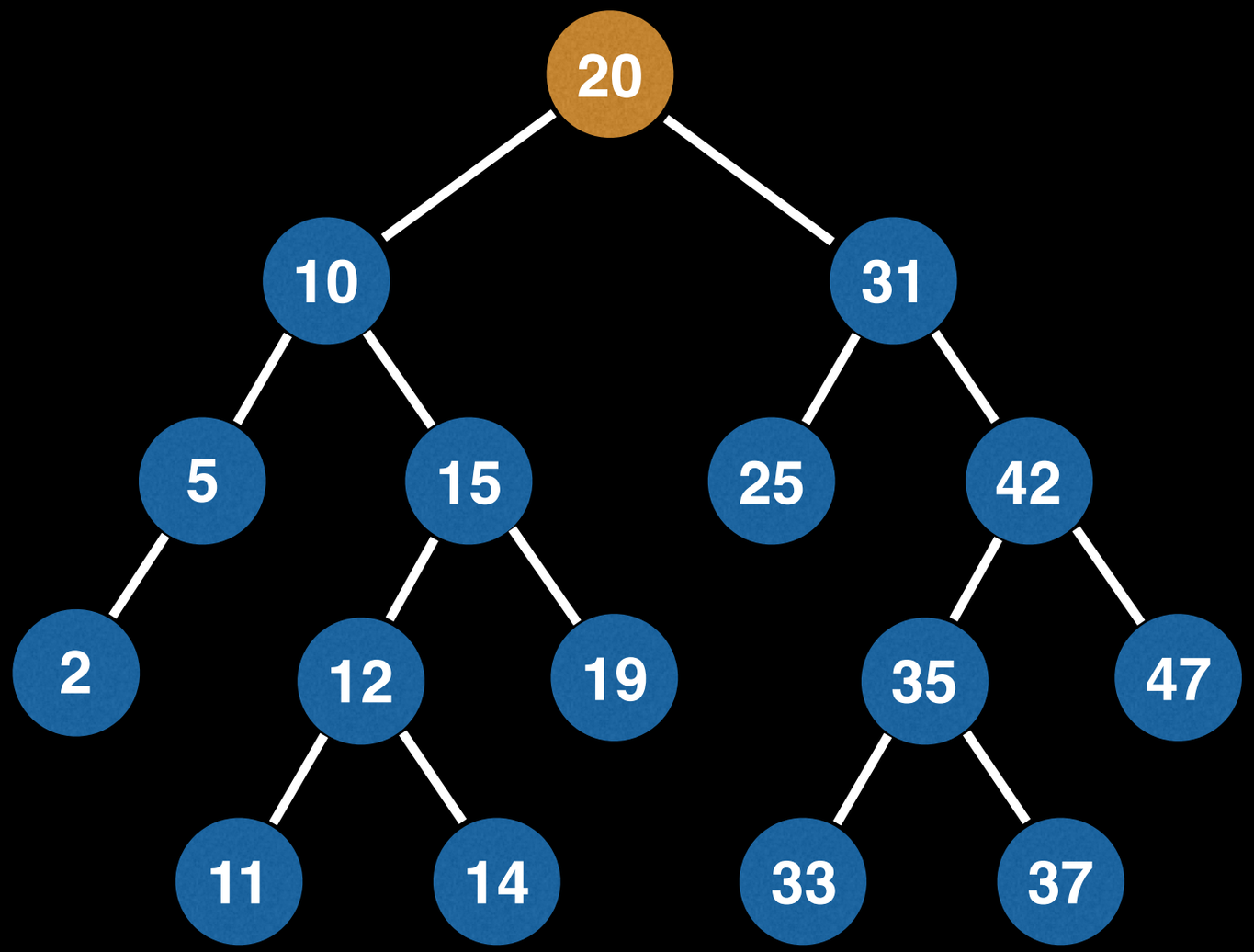


Find Phase

Find queries:

find(14) ←

find(26)

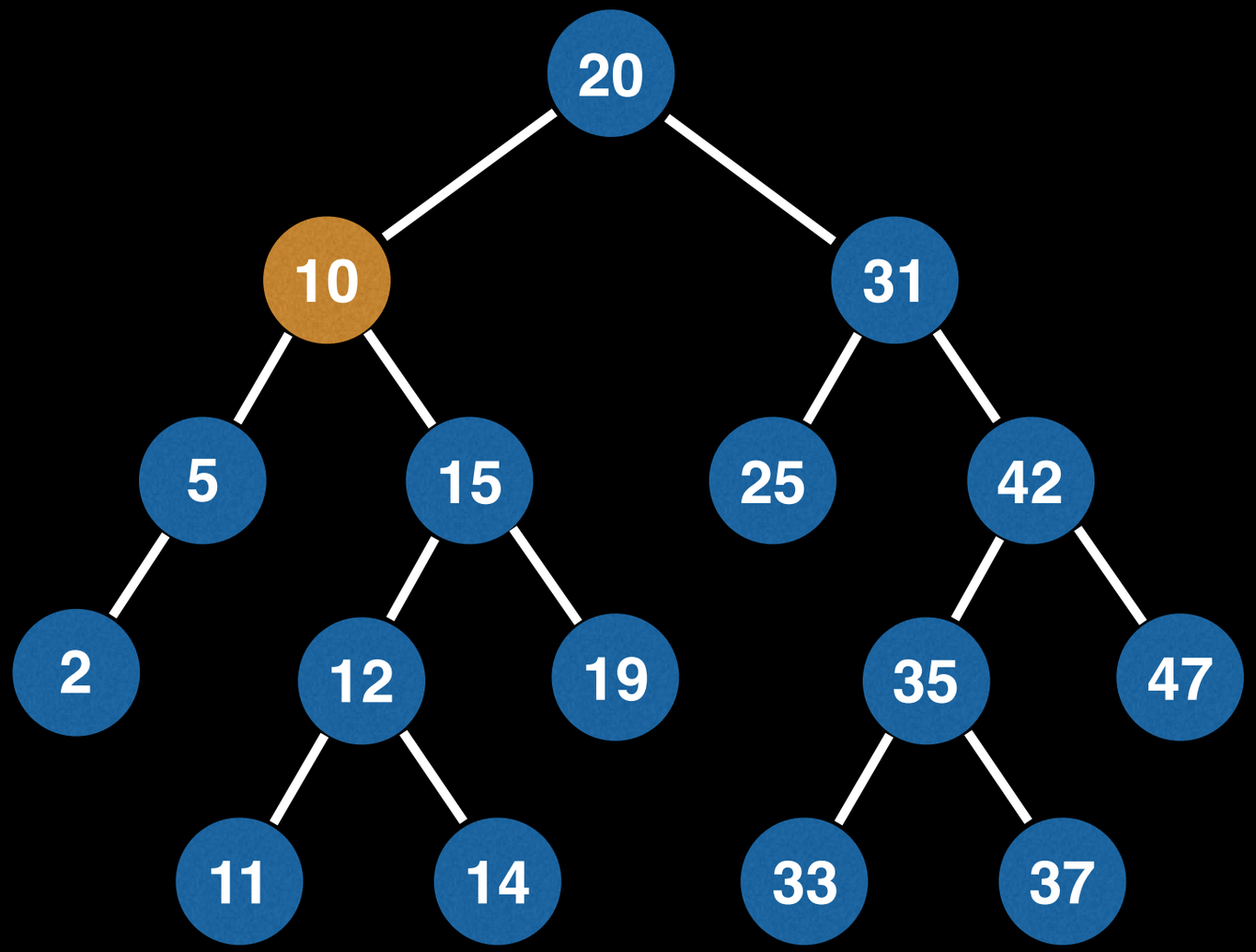


Find Phase

Find queries:

find(14) ←

find(26)

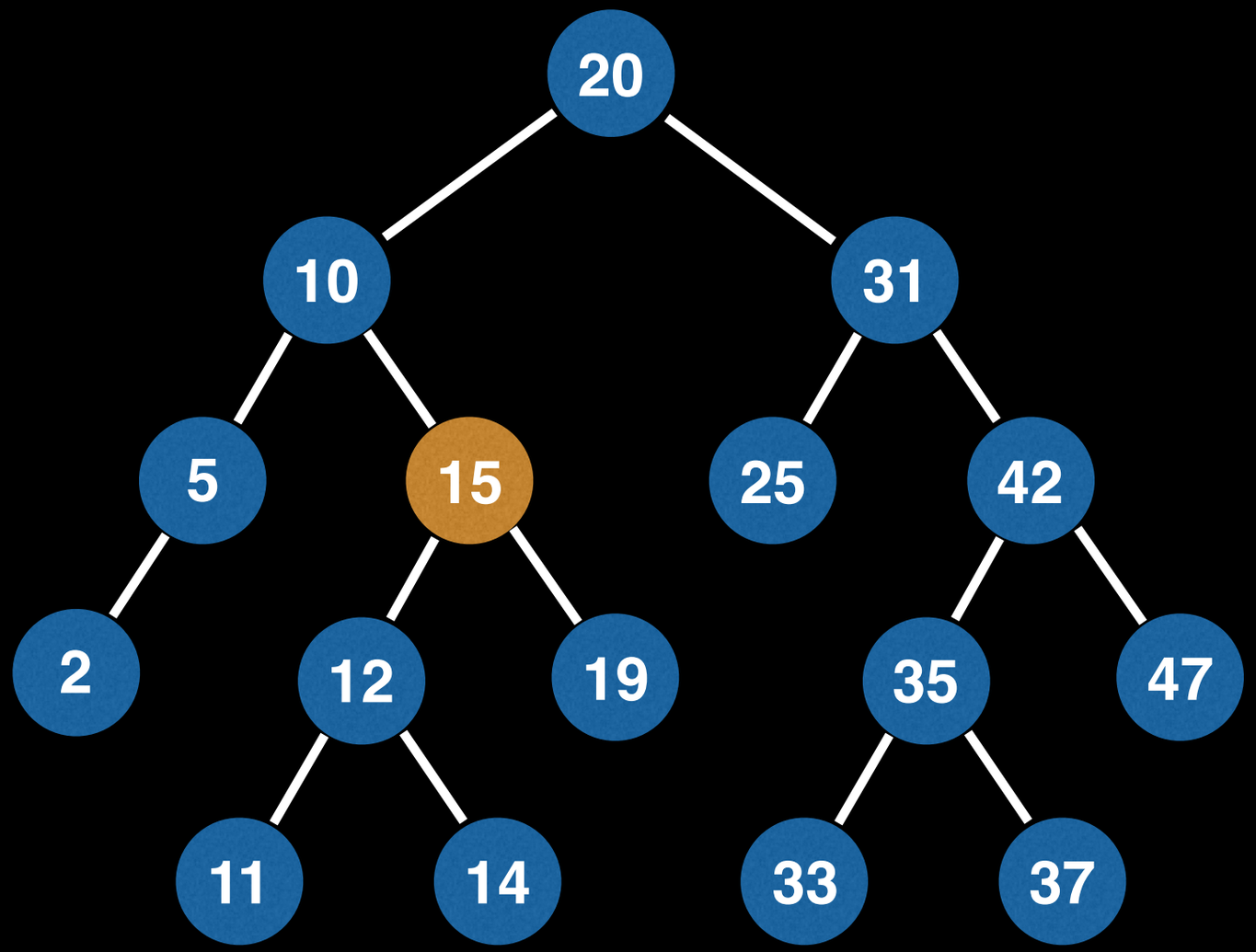


Find Phase

Find queries:

find(14) ←

find(26)

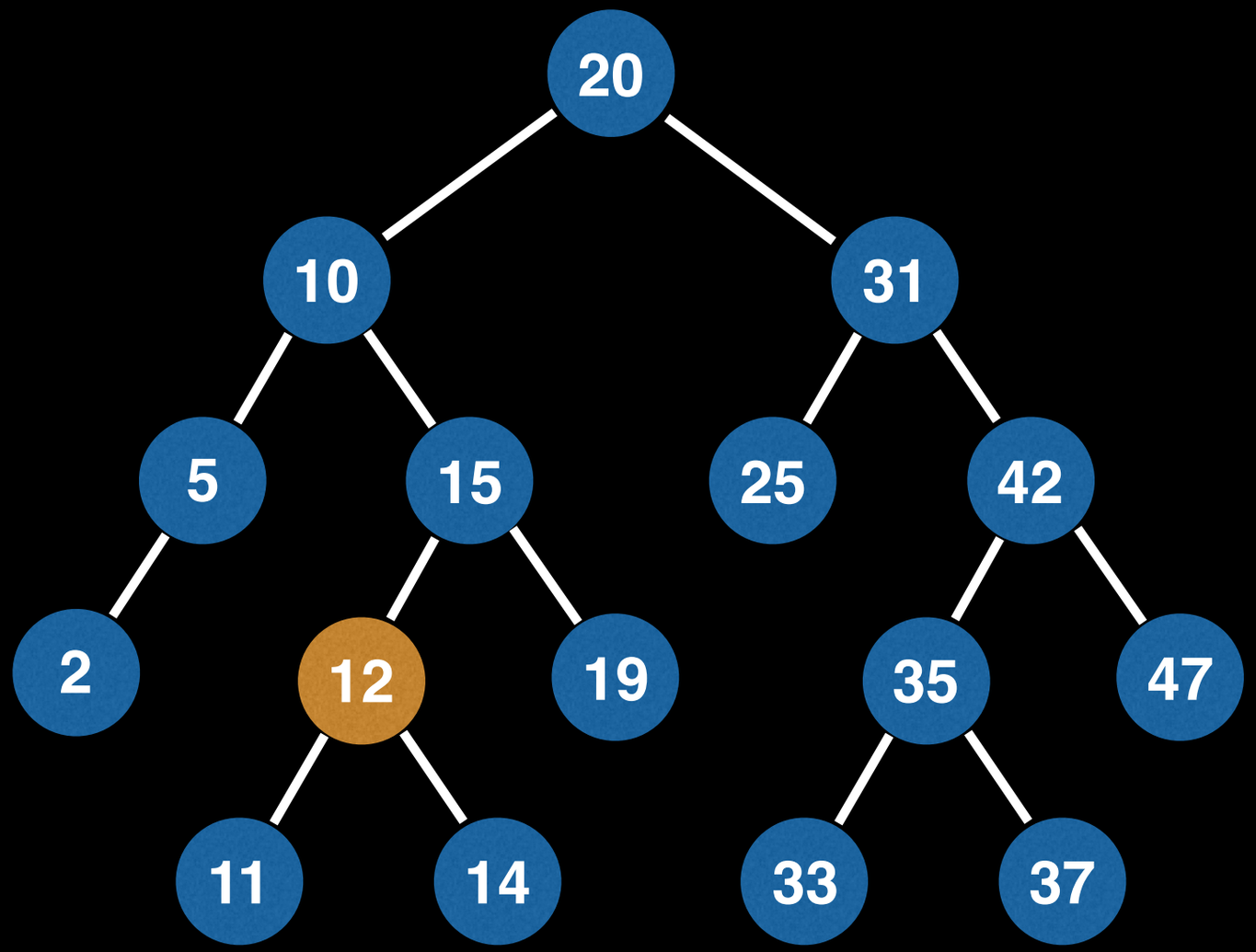


Find Phase

Find queries:

find(14) ←

find(26)

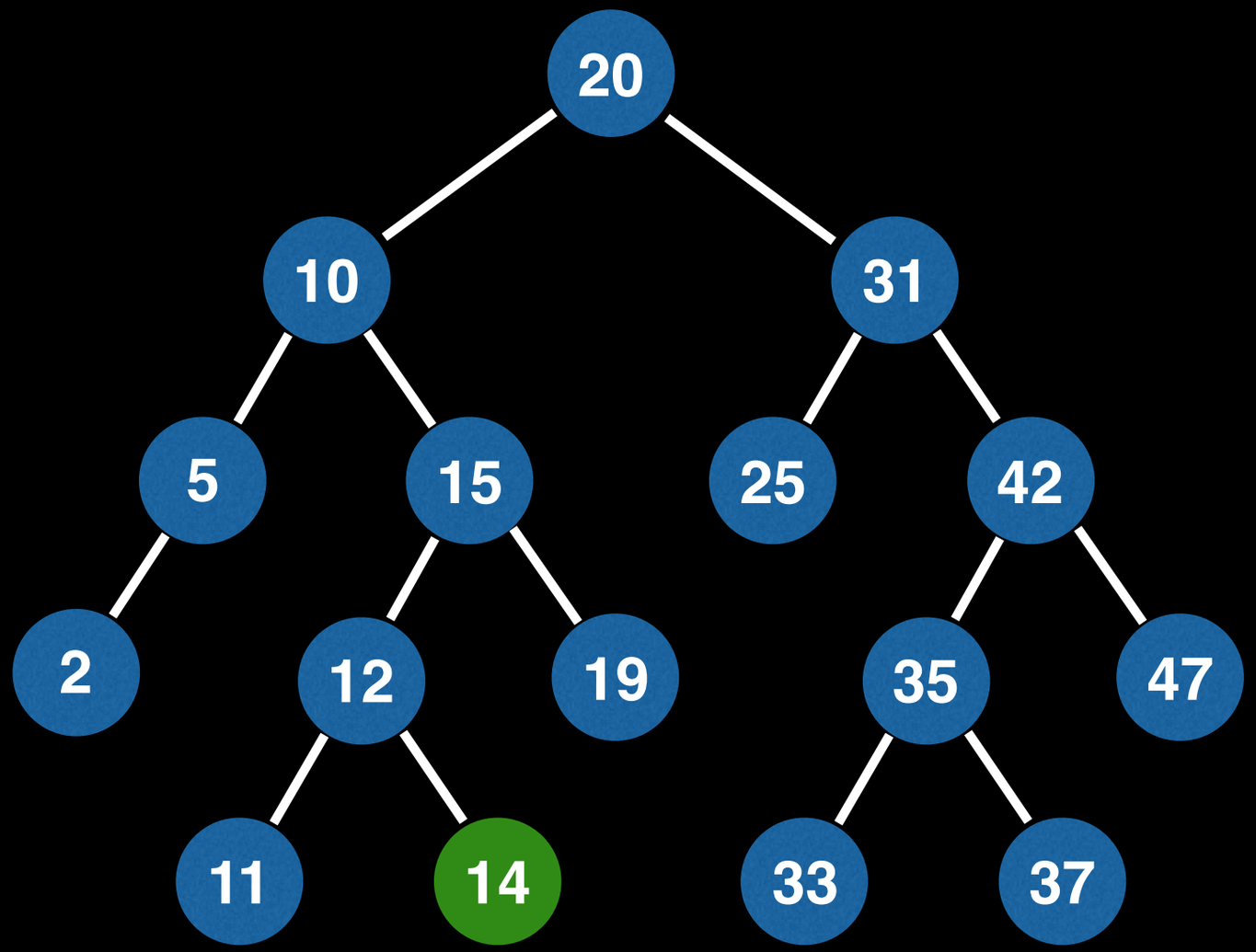


Find Phase

Find queries:

find(14) ←

find(26)

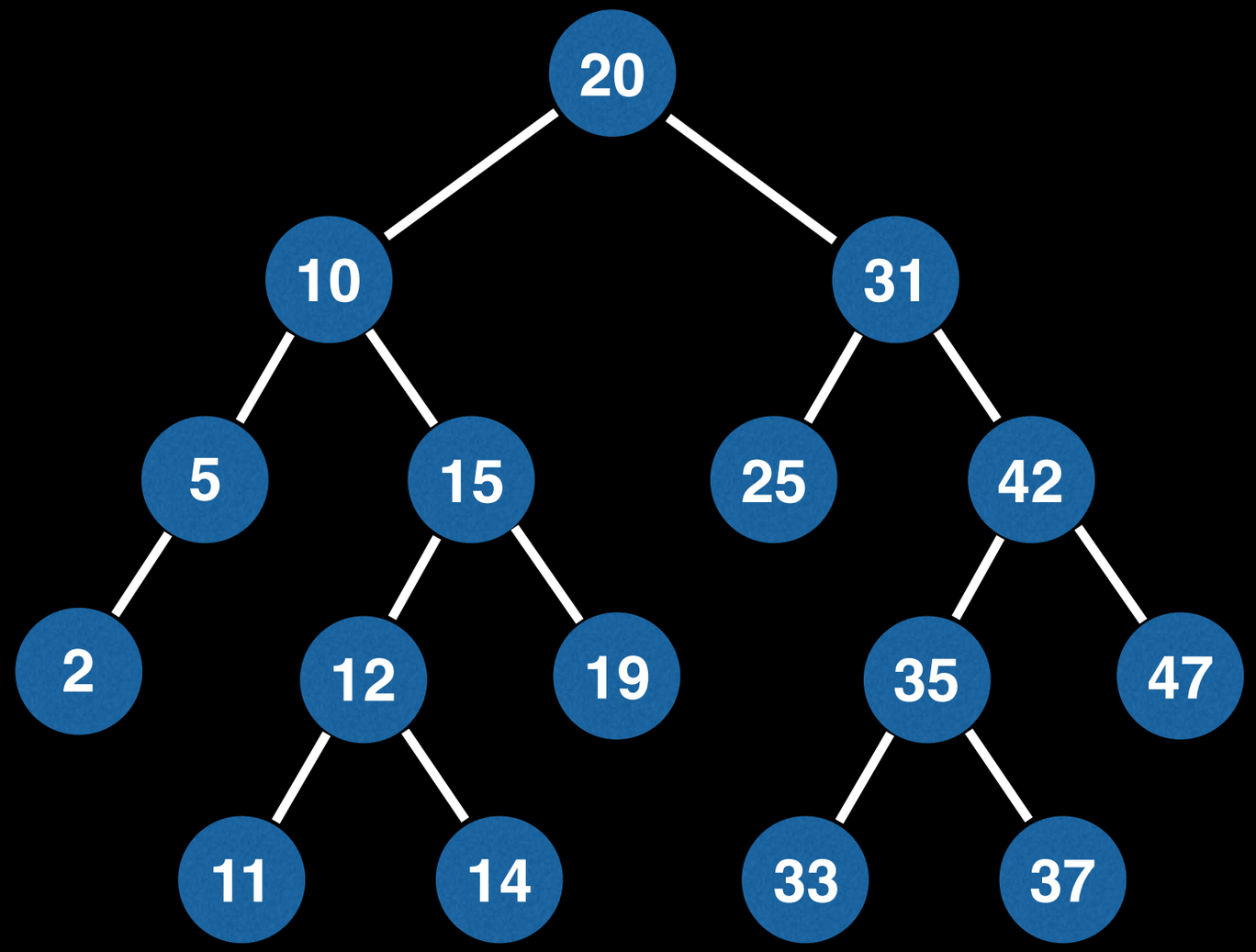


Find Phase

Find queries:

find(14)

find(26) ←

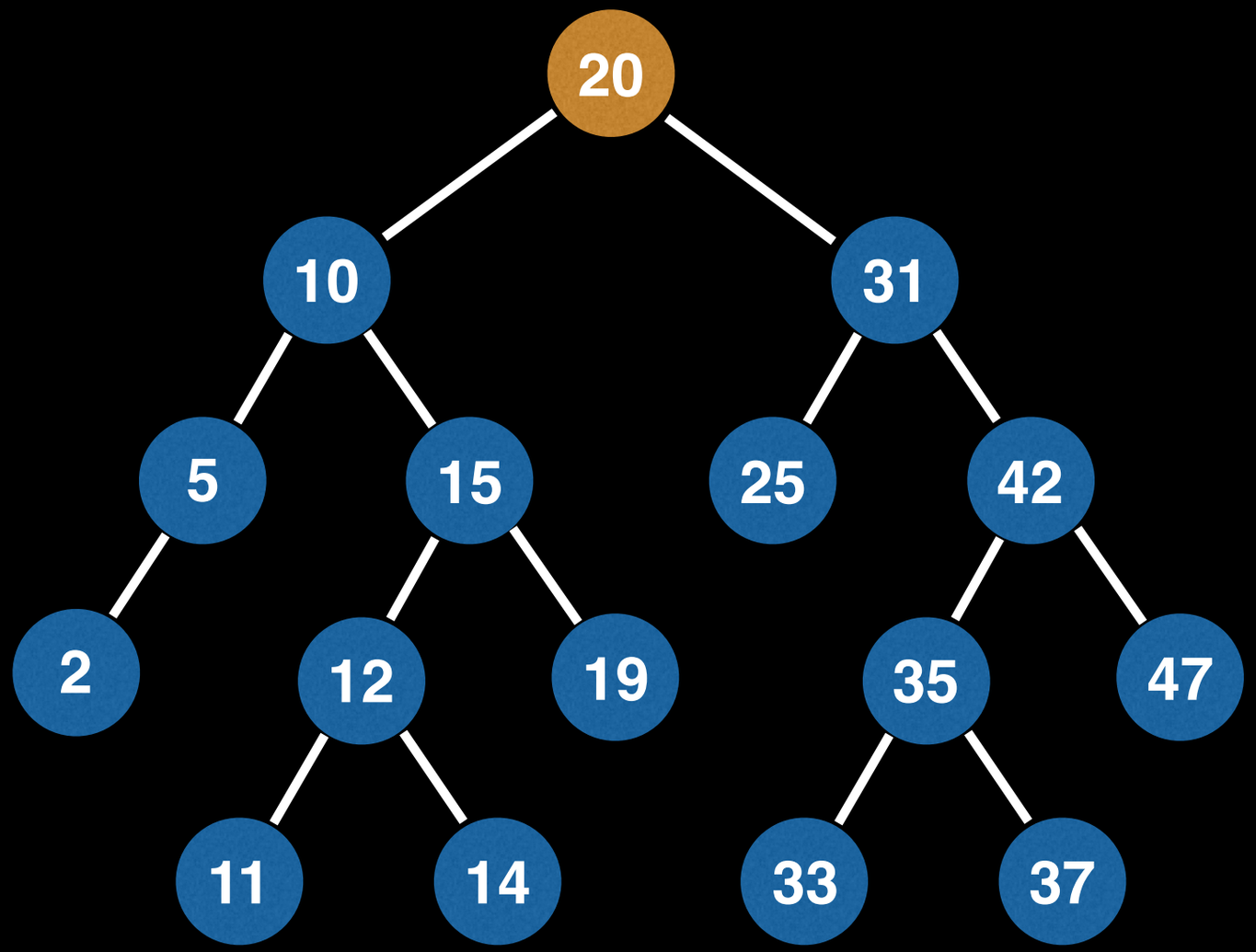


Find Phase

Find queries:

find(14)

find(26) ←

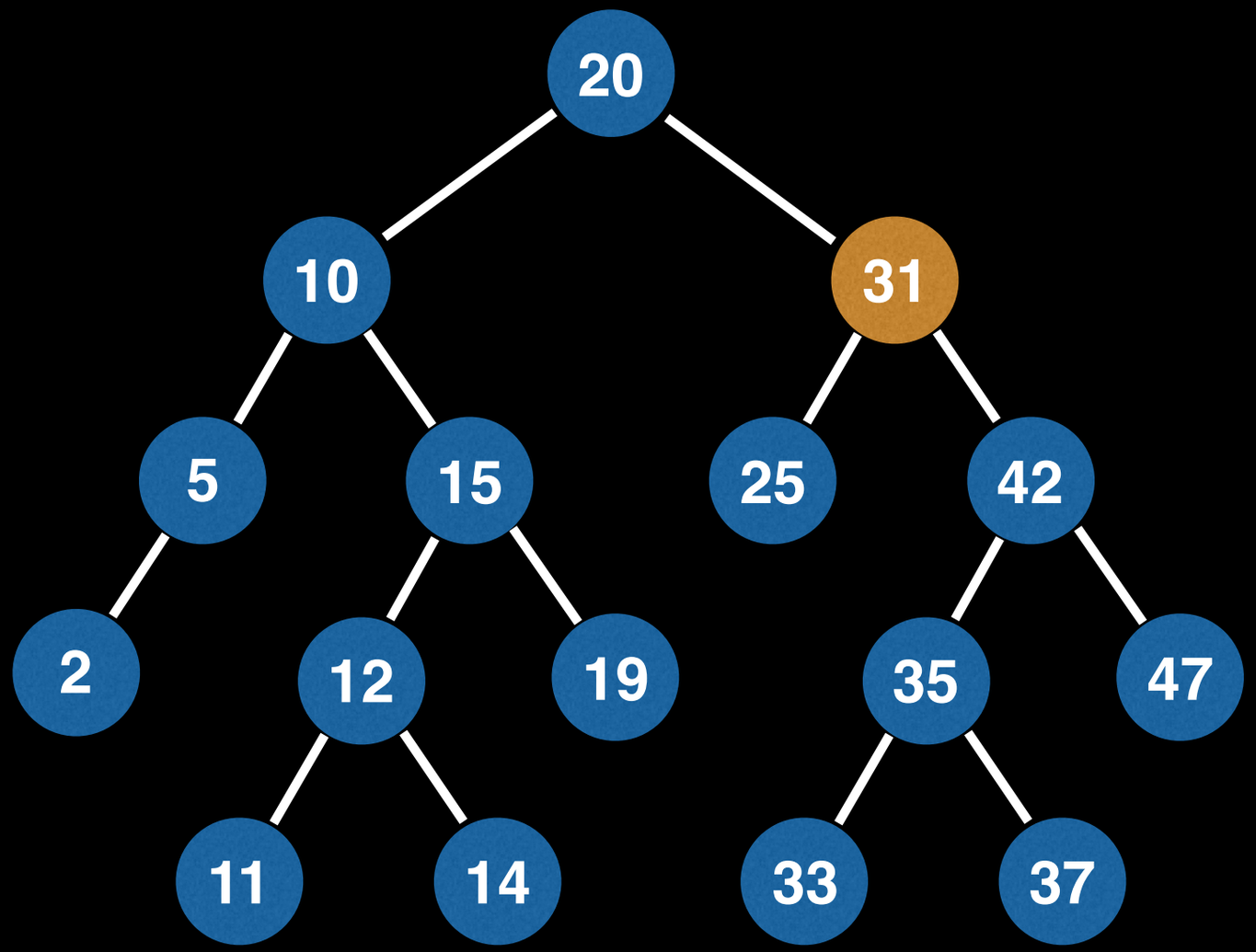


Find Phase

Find queries:

find(14)

find(26) ←

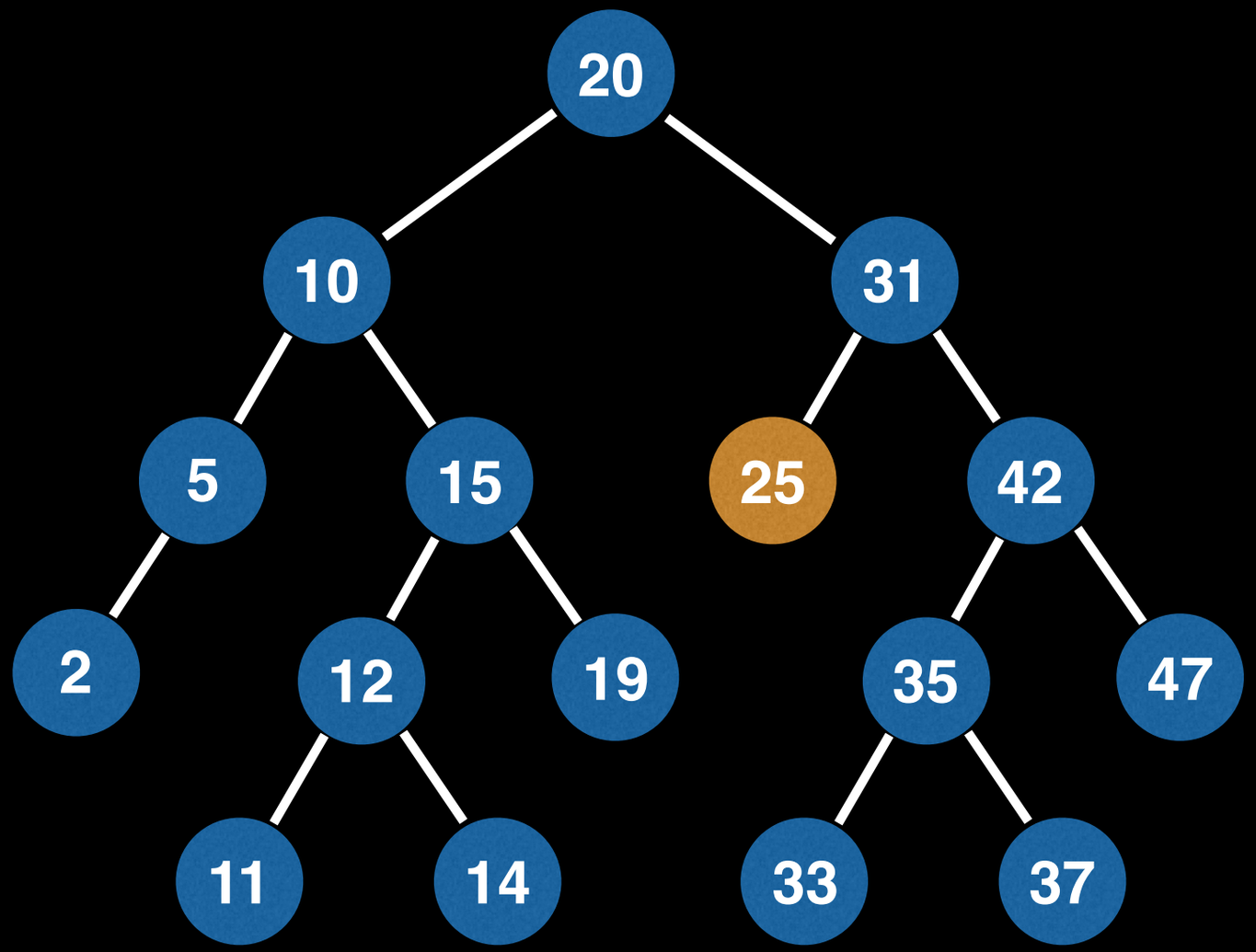


Find Phase

Find queries:

find(14)

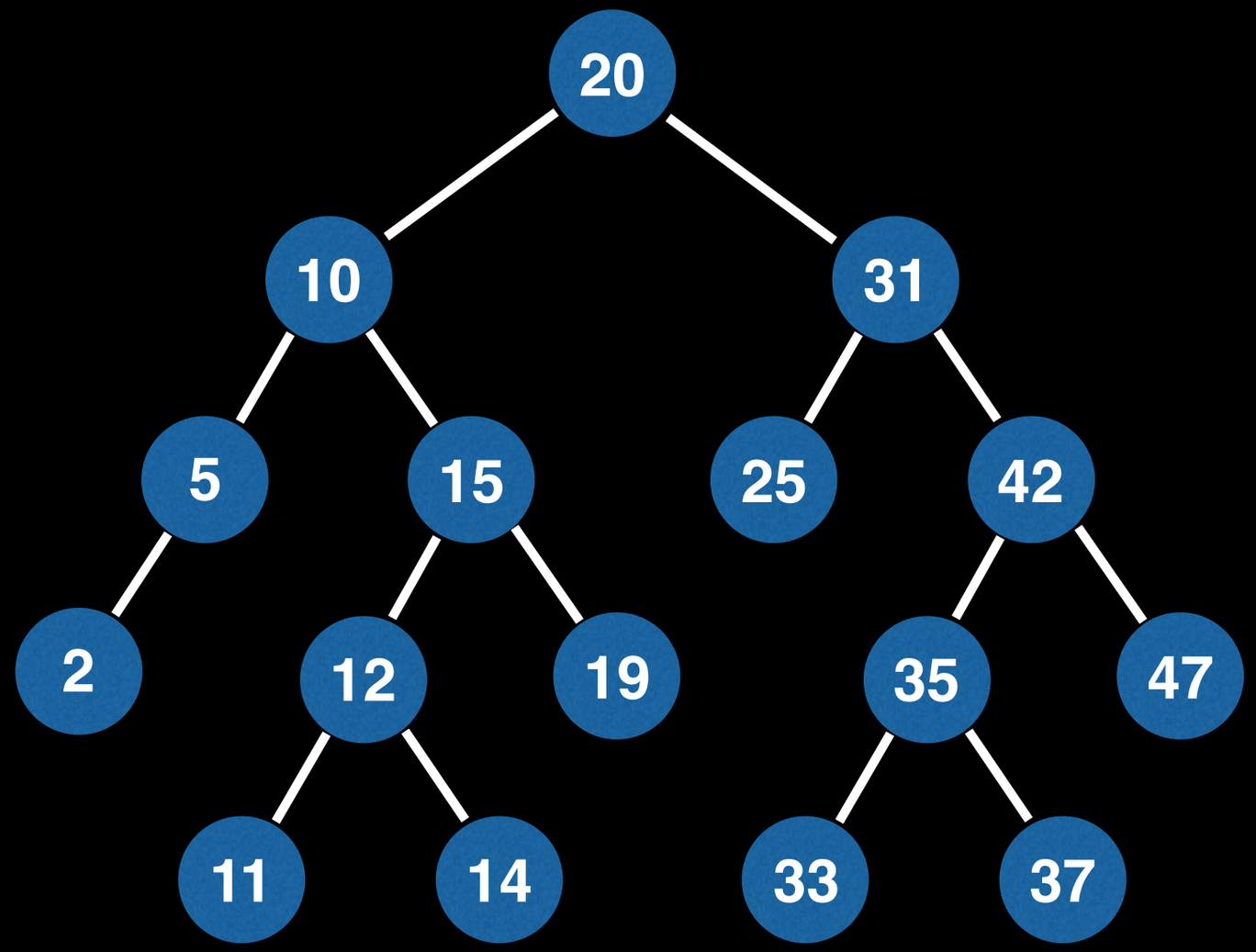
find(26) ←



Find Phase

Find queries:

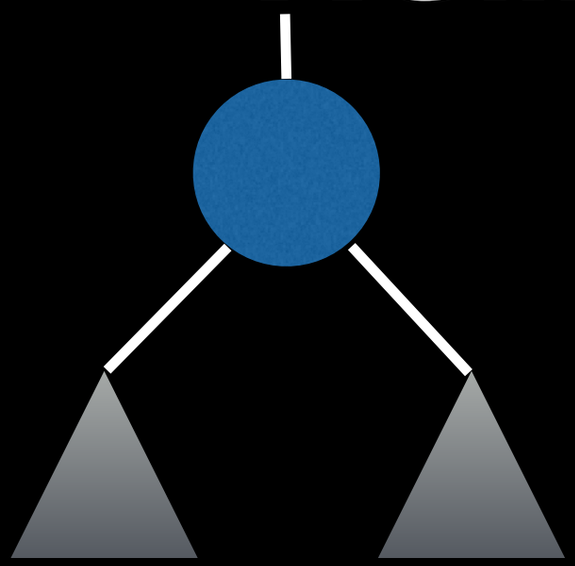
find(14)
find(26) ←



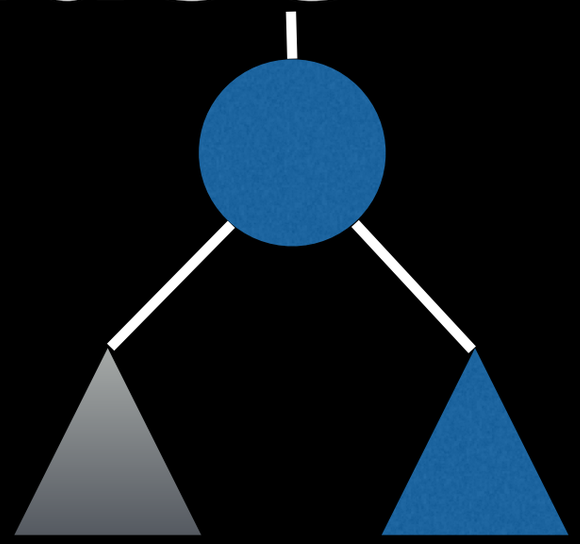
26 was not found :/

Remove phase

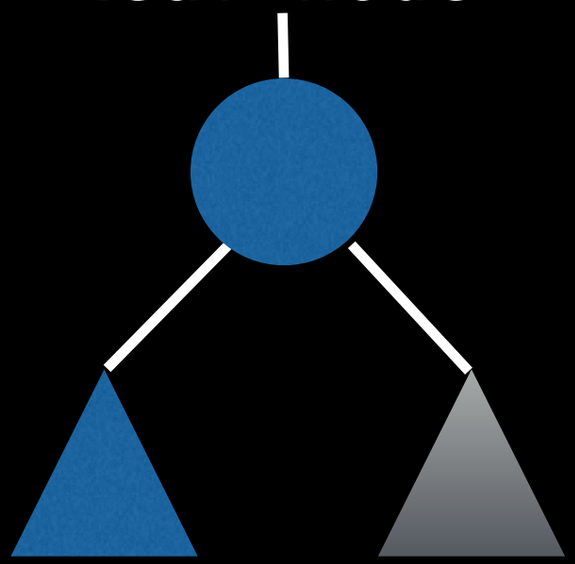
Four Cases



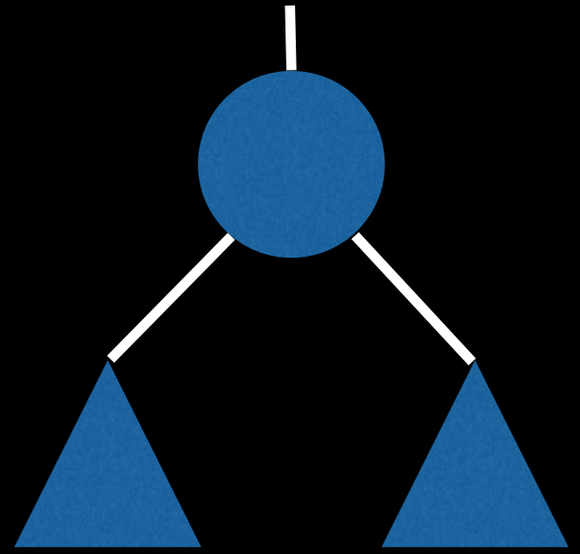
Node to remove is a leaf node



Node to remove has a right subtree but no left subtree



Node to remove has a left subtree but no right subtree

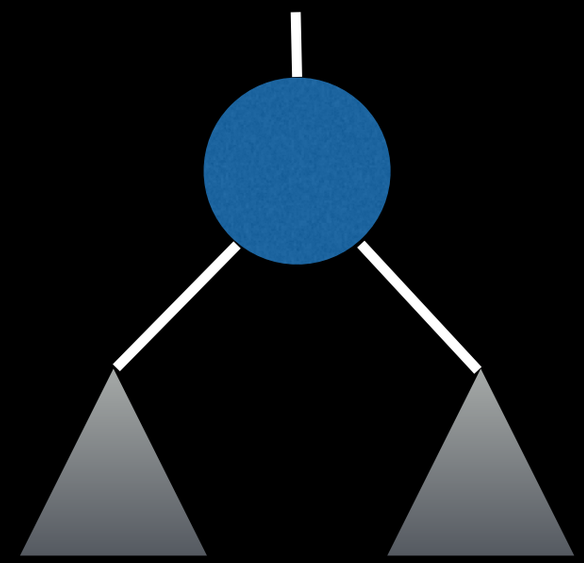


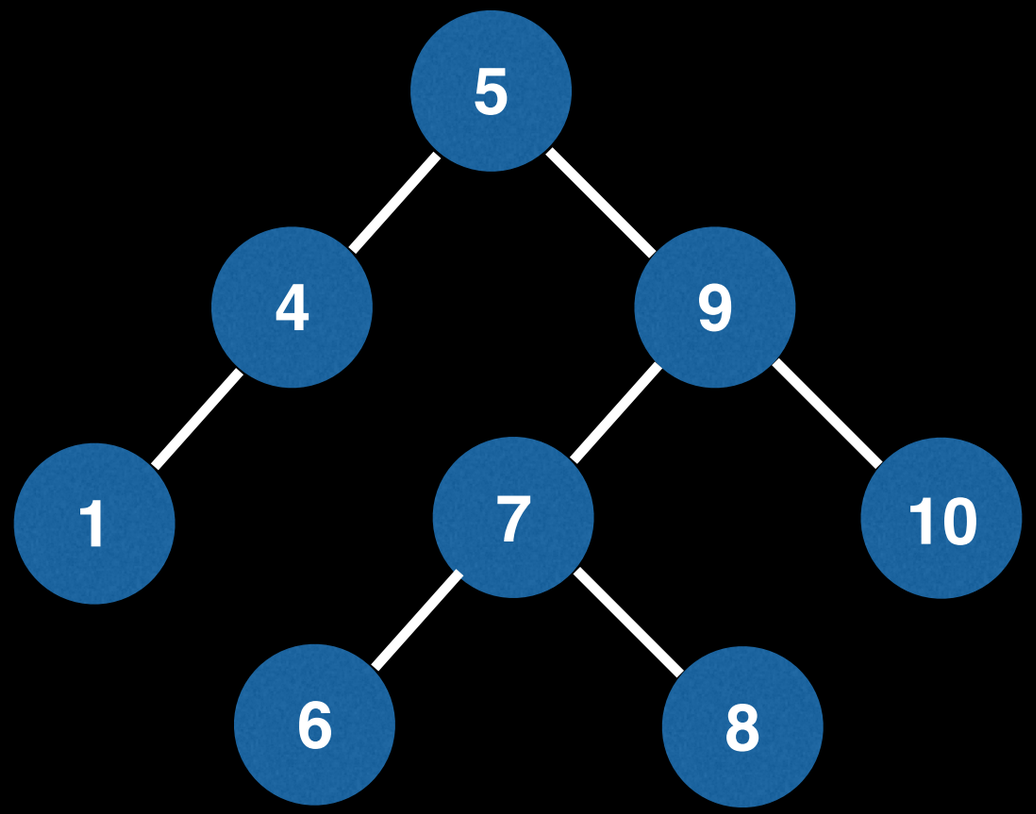
Node to remove has a both a left subtree and a right subtree

Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)

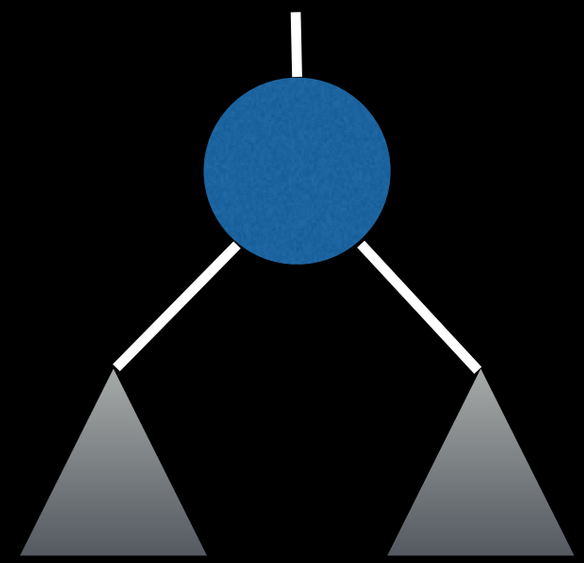




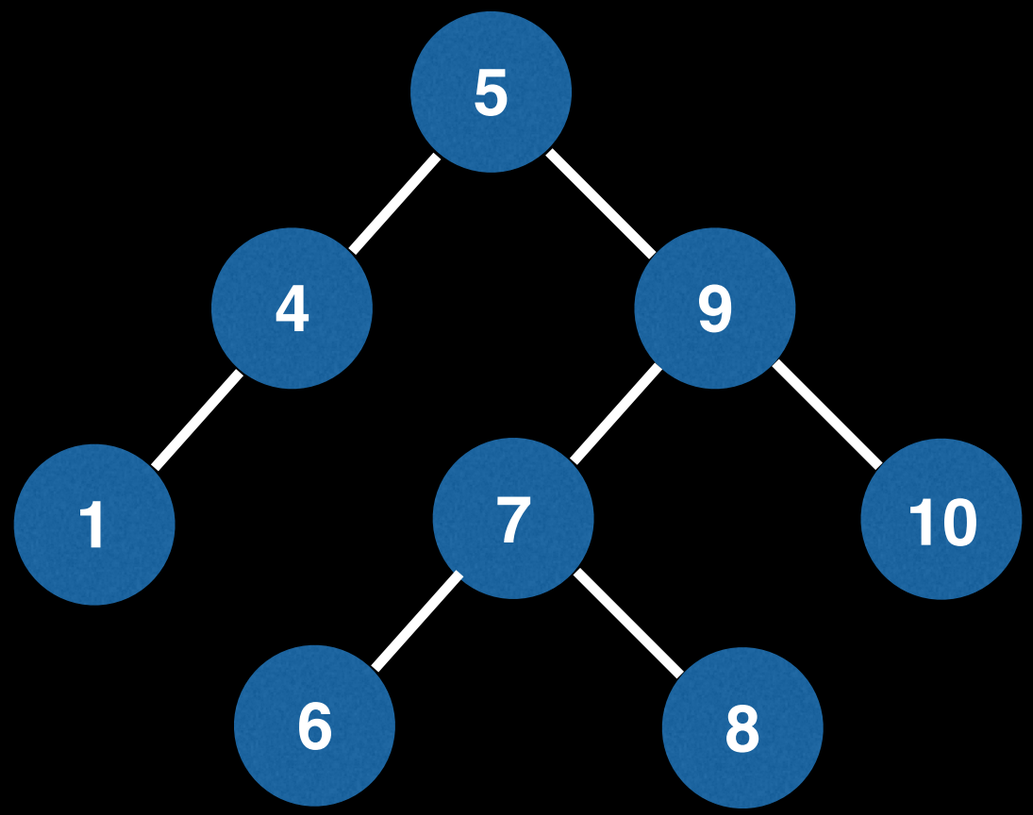
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)



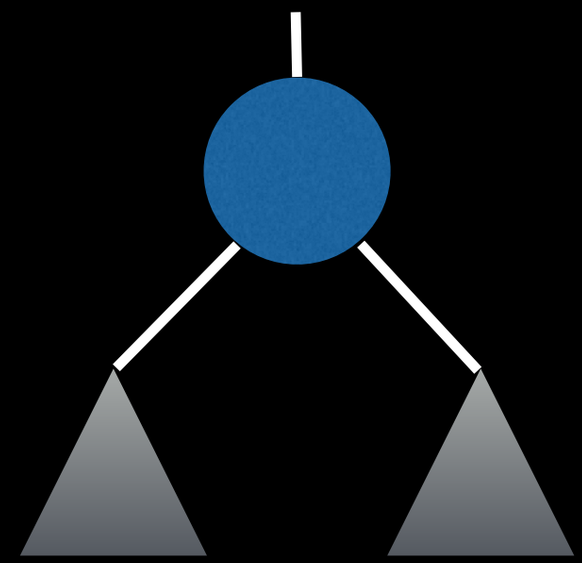
Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



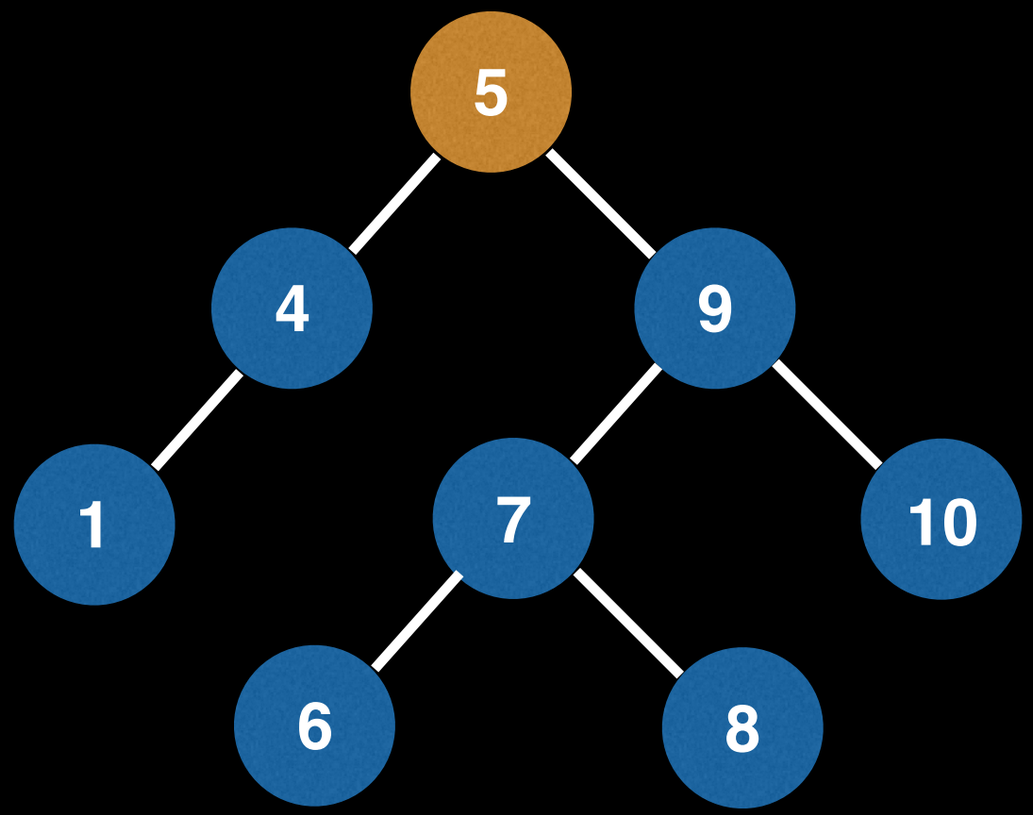
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)



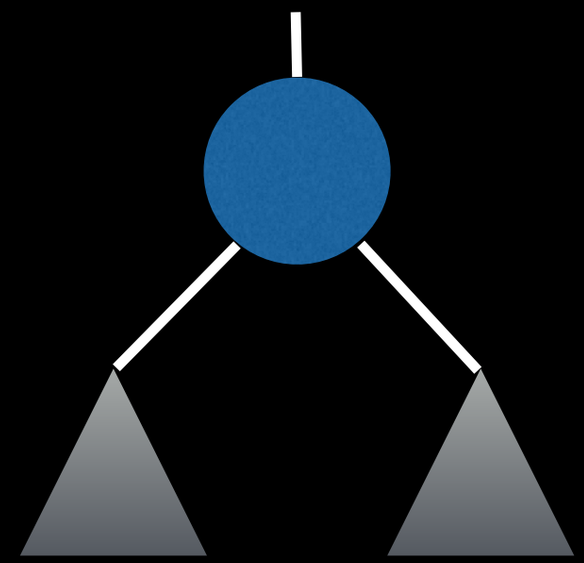
Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



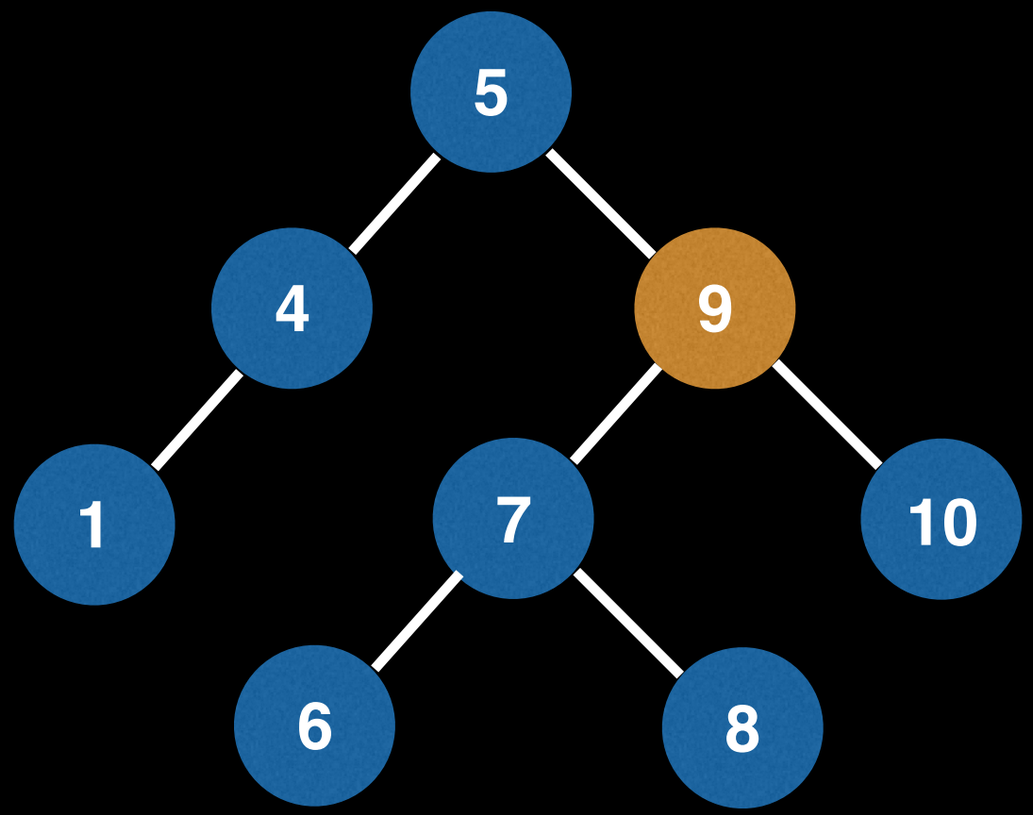
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)



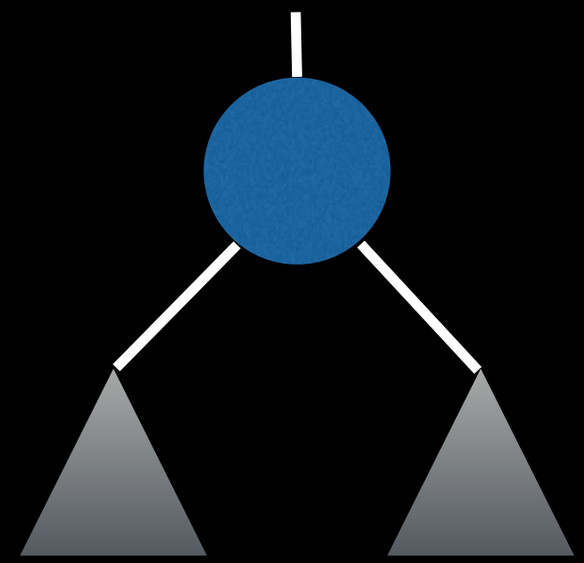
Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



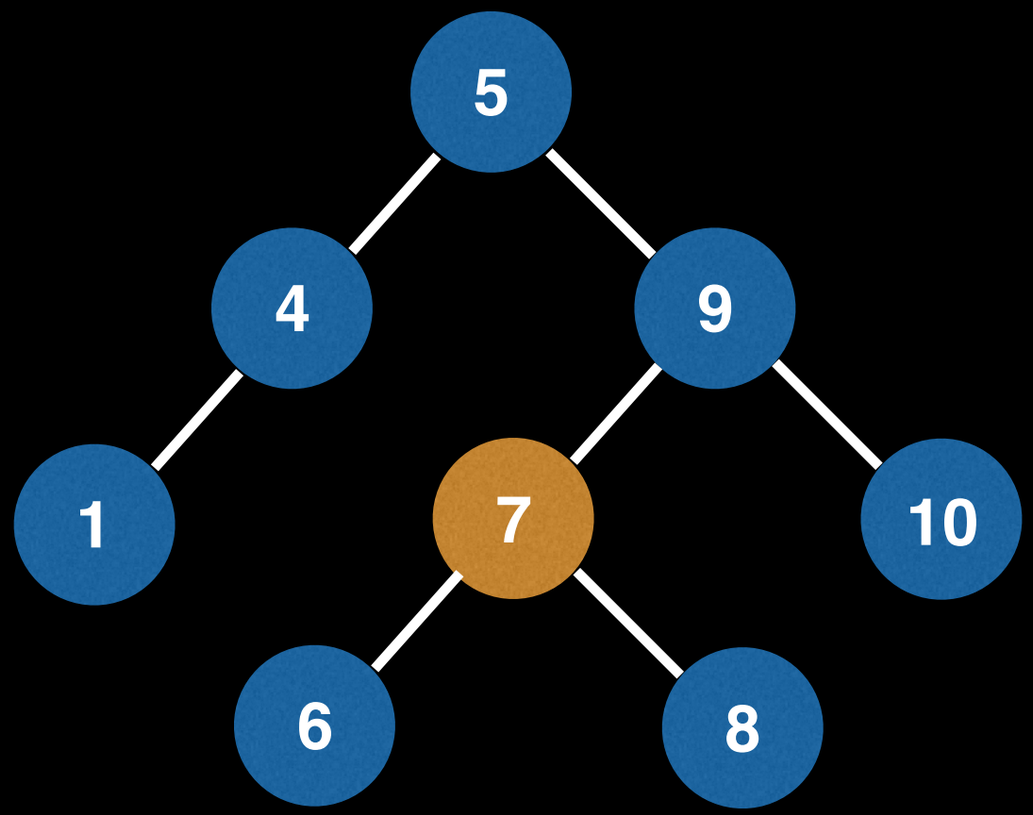
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)



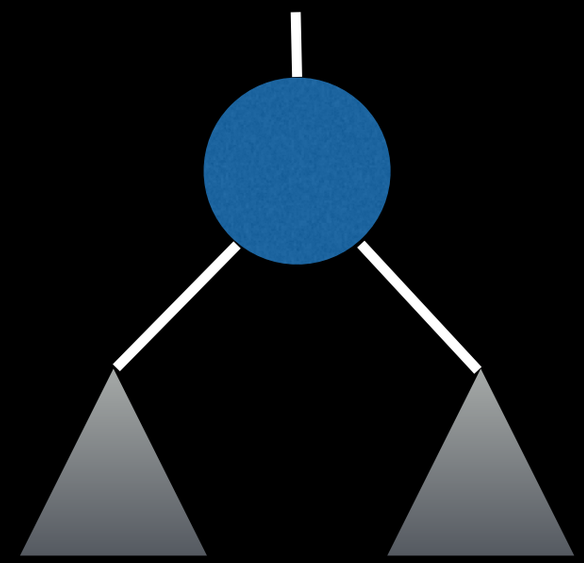
Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



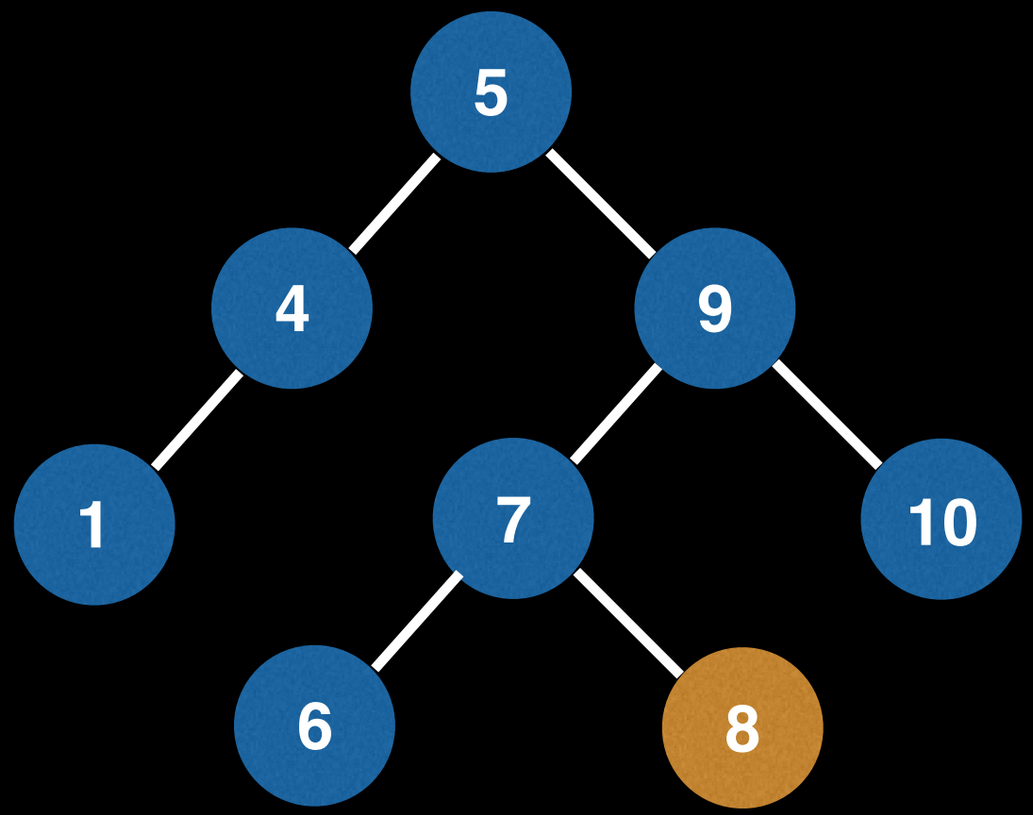
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)



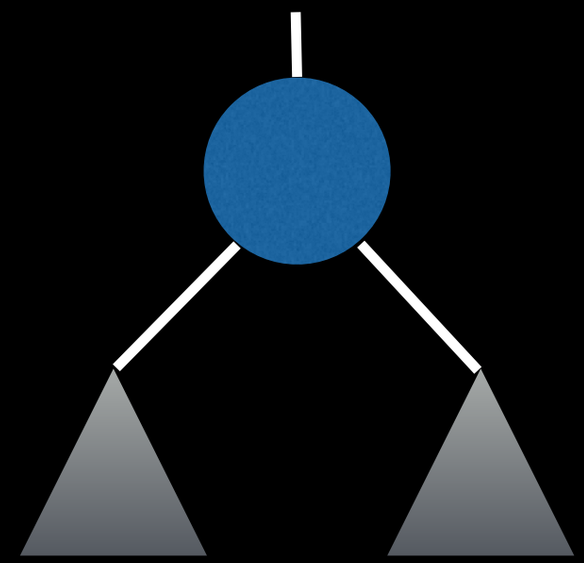
Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



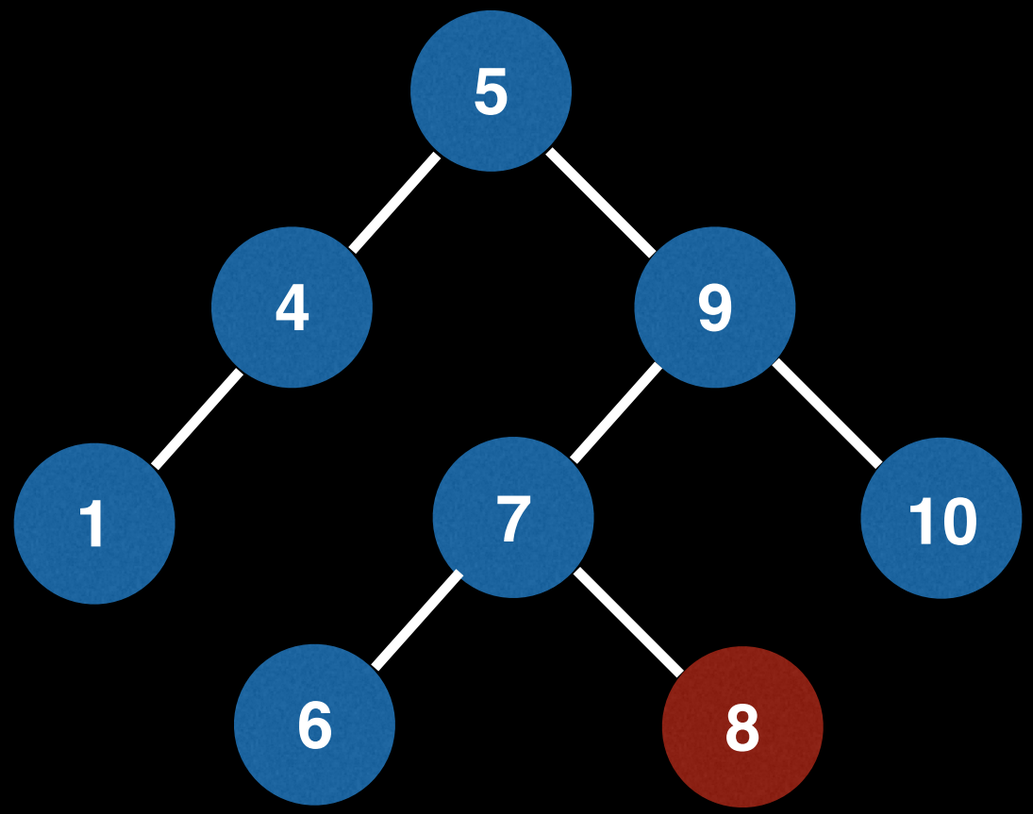
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)



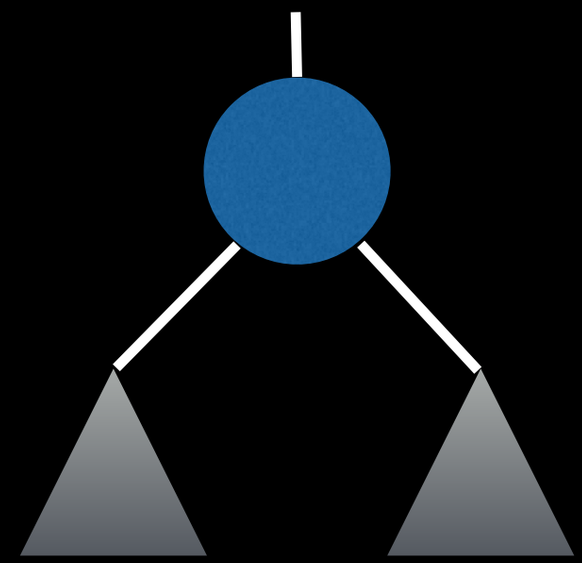
Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



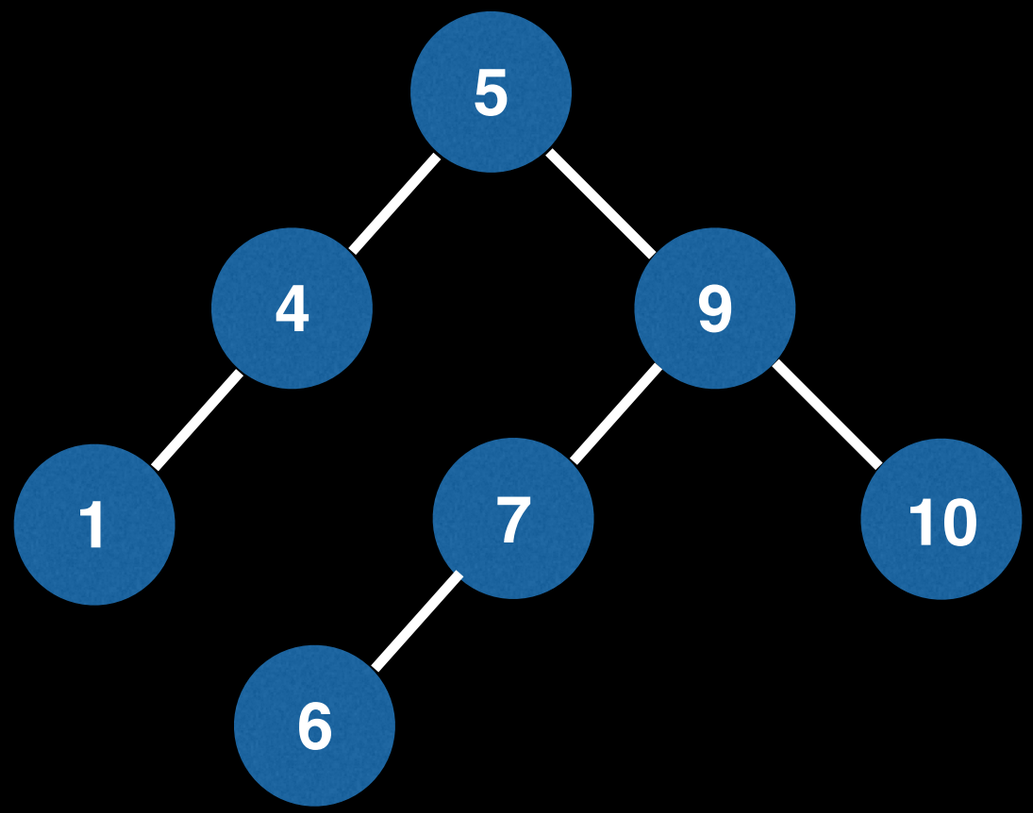
Remove phase

Case I: Leaf node

If the node we wish to remove is a leaf node then we may do so without side effect :)

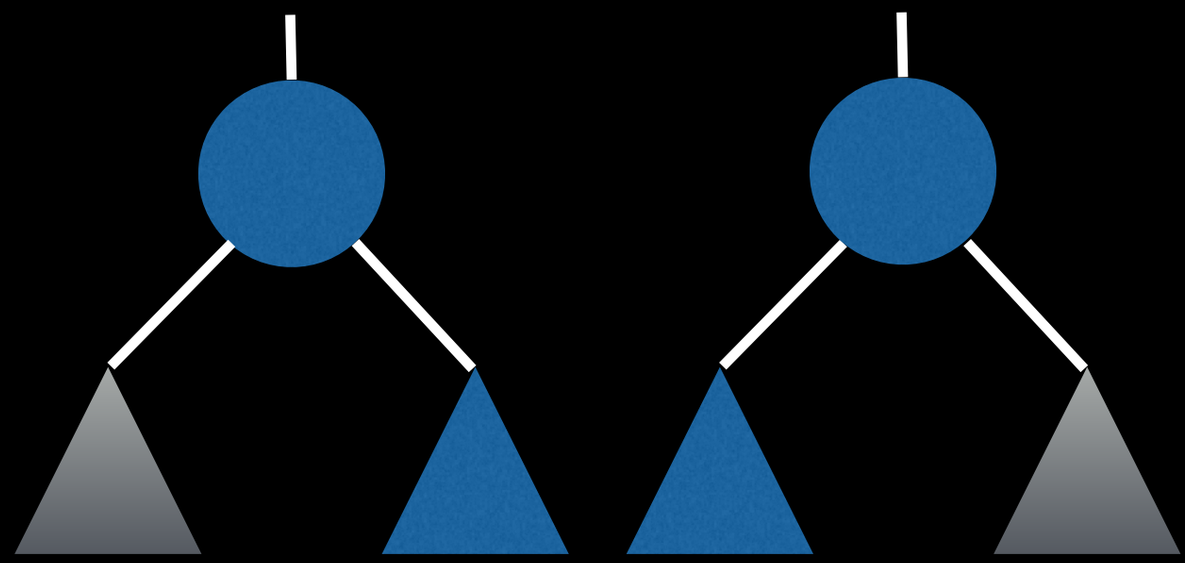


Suppose we want to remove 8 from the BST on the right. First we would find 8 then remove it immediately since it’s a leaf node



Remove phase

Cases II & III: either the left/right child node is a subtree

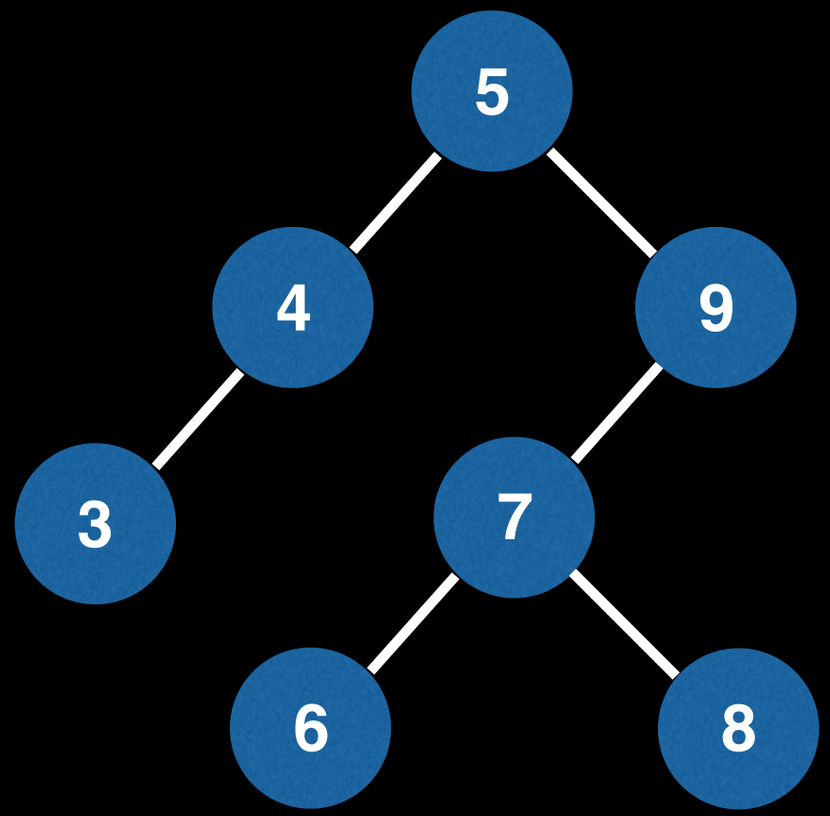


The successor of the node we are trying to remove in these cases will be the **immediate node down from the left/right subtree**.

It may be the case that we are removing the root node of the BST, in which case its immediate child becomes the new root, as you would expect.

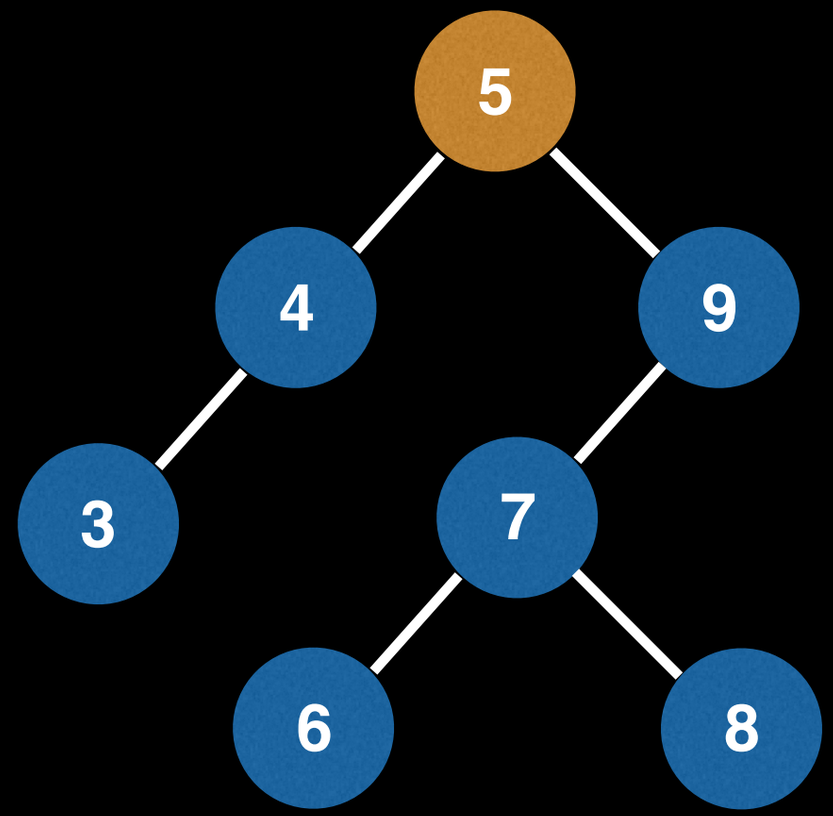
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



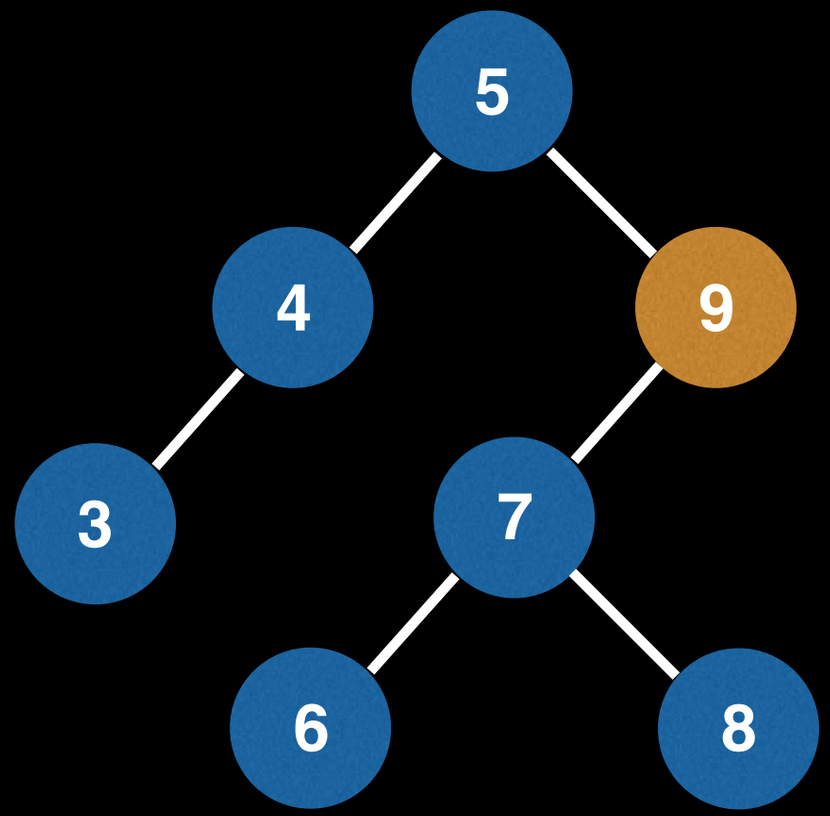
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



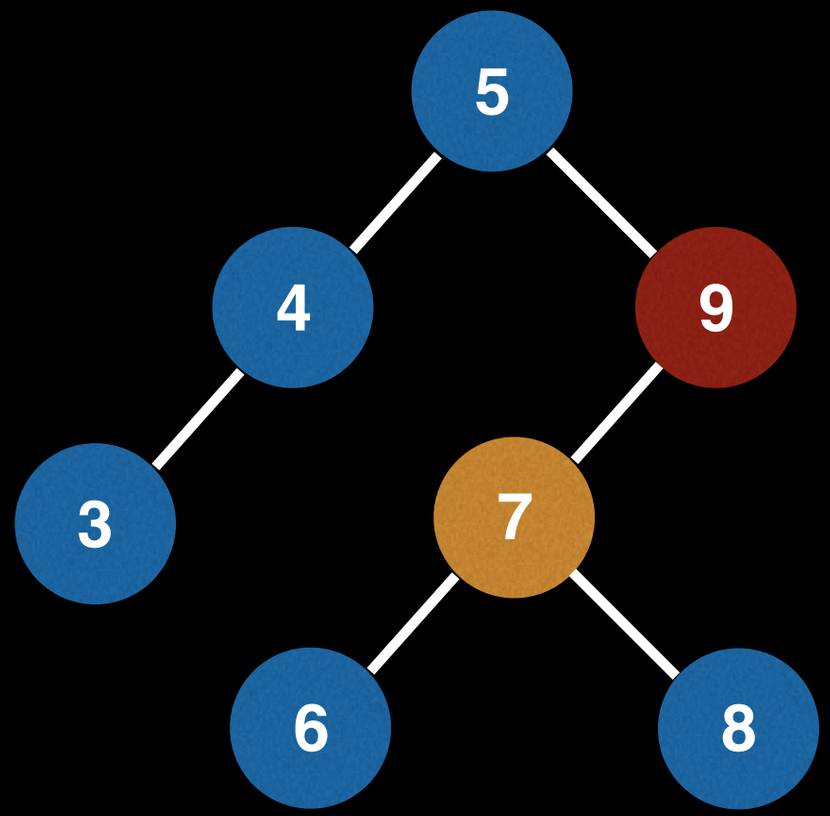
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



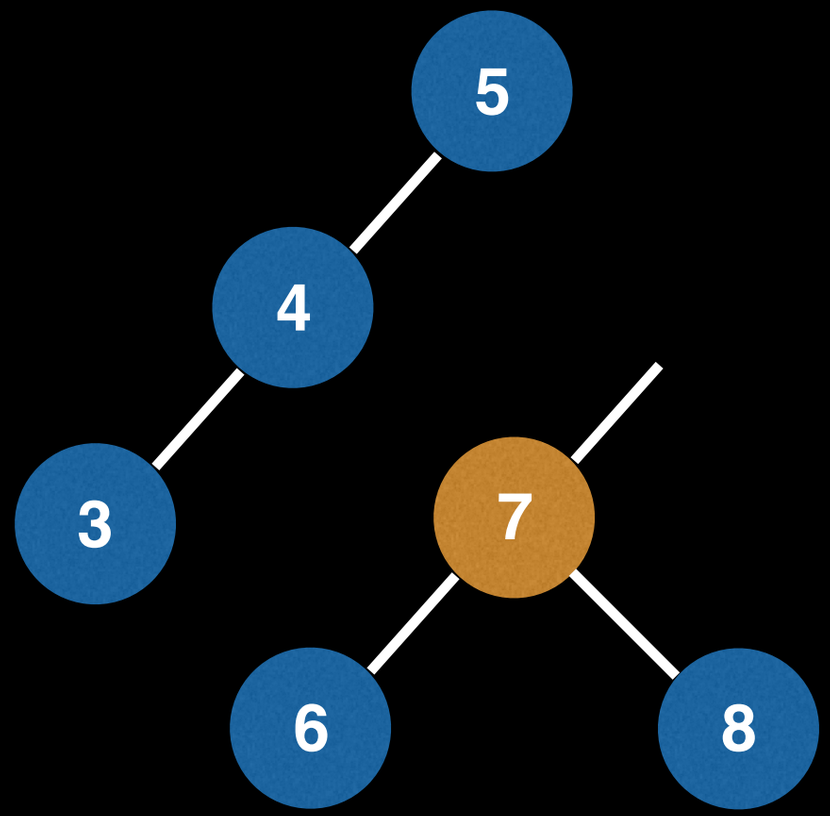
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



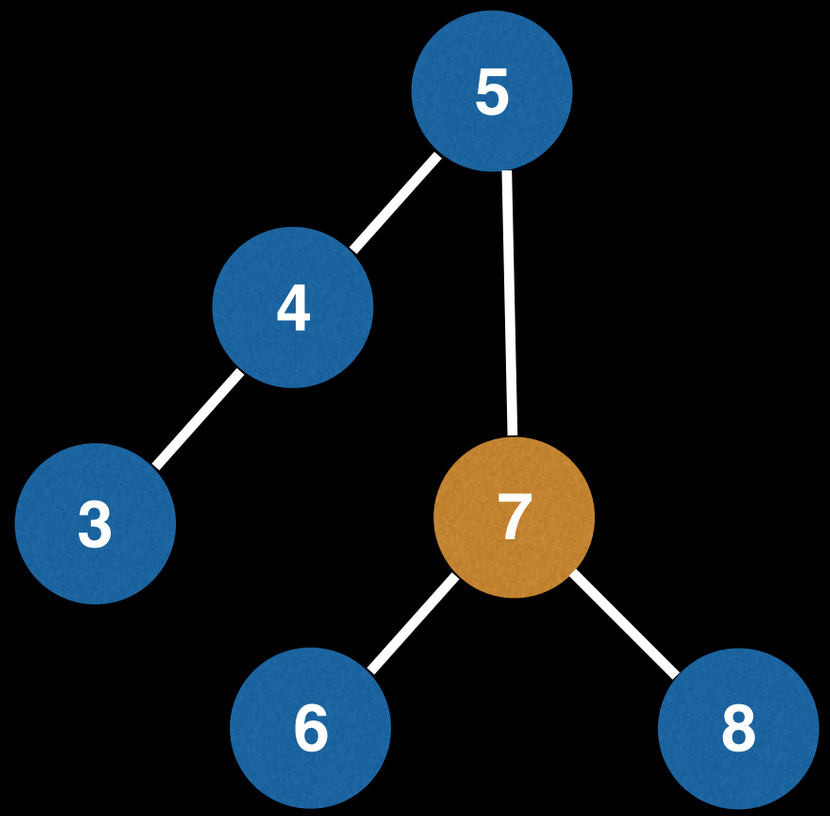
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



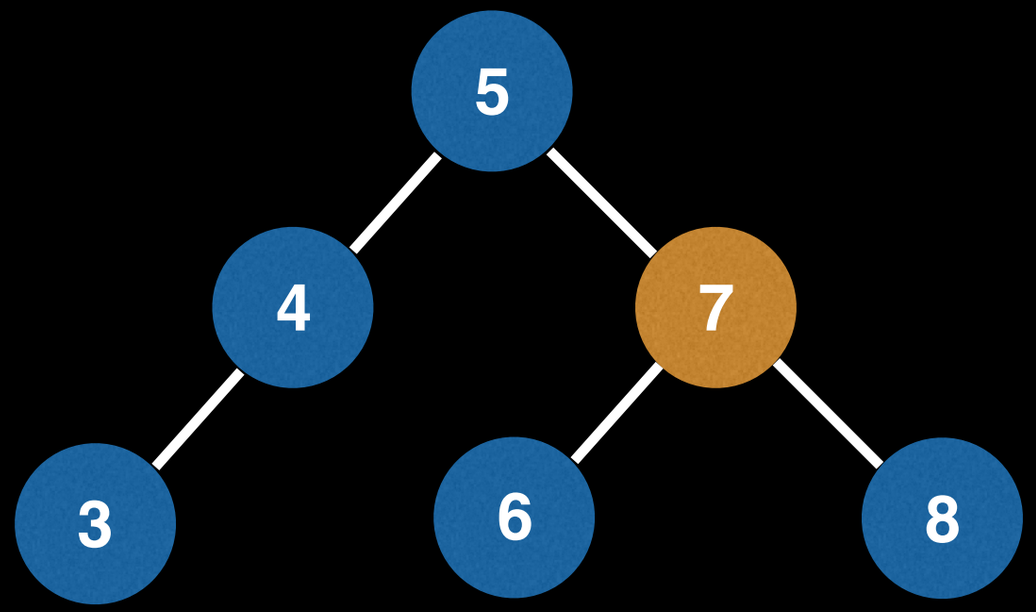
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



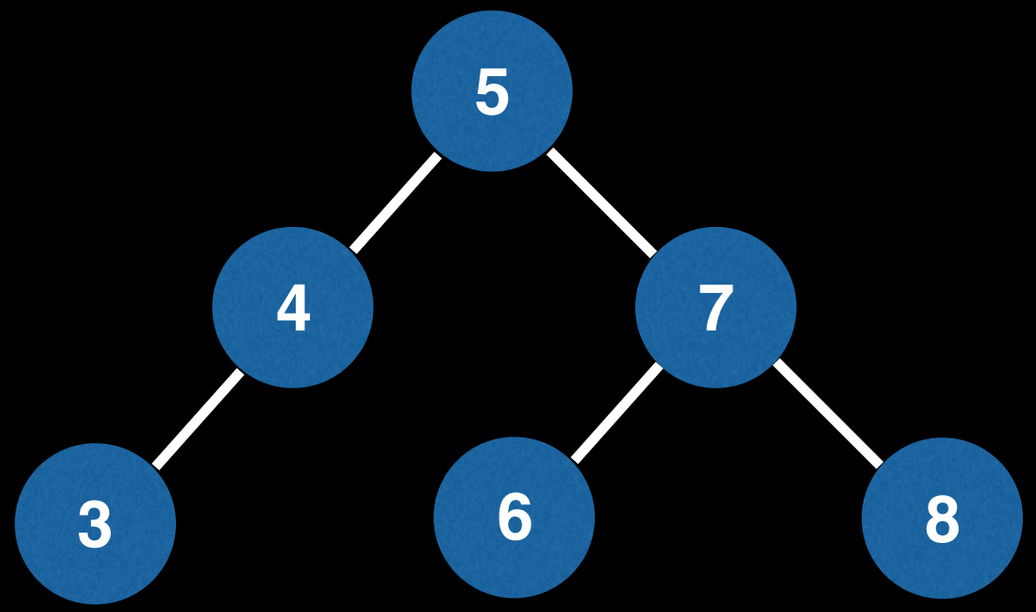
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



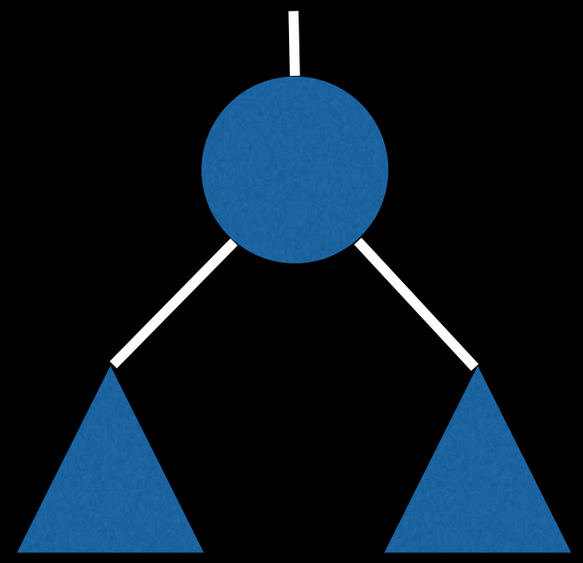
Remove phase

Suppose we wish to remove 9,
then we encounter case II
with a left subtree



Remove phase

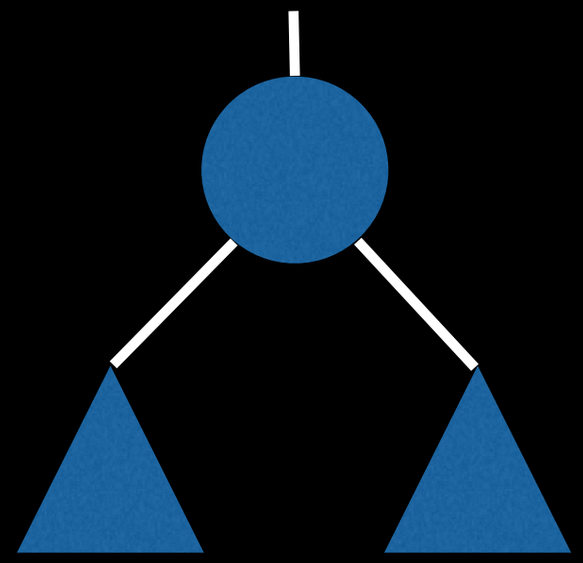
Case IV: Node to remove has both a left subtree and a right subtree



Q: In which subtree will the successor of the node we are trying to remove be?

Remove phase

Case IV: Node to remove has both a left subtree and a right subtree



Q: In which subtree will the successor of the node we are trying to remove be?

A: The answer is both! The successor can either be the **largest value** in the **left subtree** OR the **smallest value** in the **right subtree**.

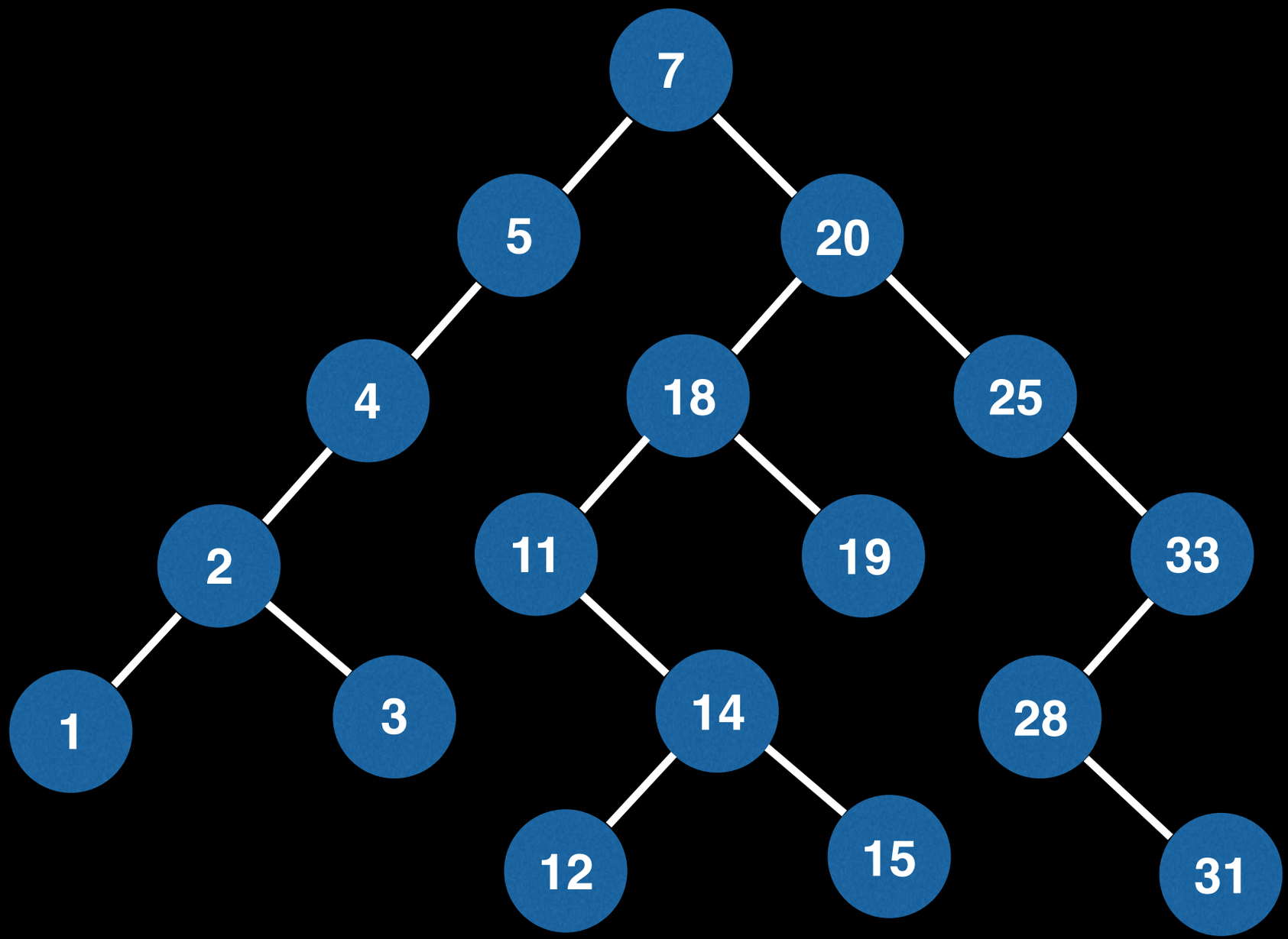
Remove phase

Once the successor node has been identified (if it exists), replace the value of the node to remove with the value in the successor node.

NOTE: Don’t forget to remove the duplicate value of the successor node that still exists in the tree at this point! One strategy to resolve this is by calling the function again recursively but with the value to remove as the value in the successor node.

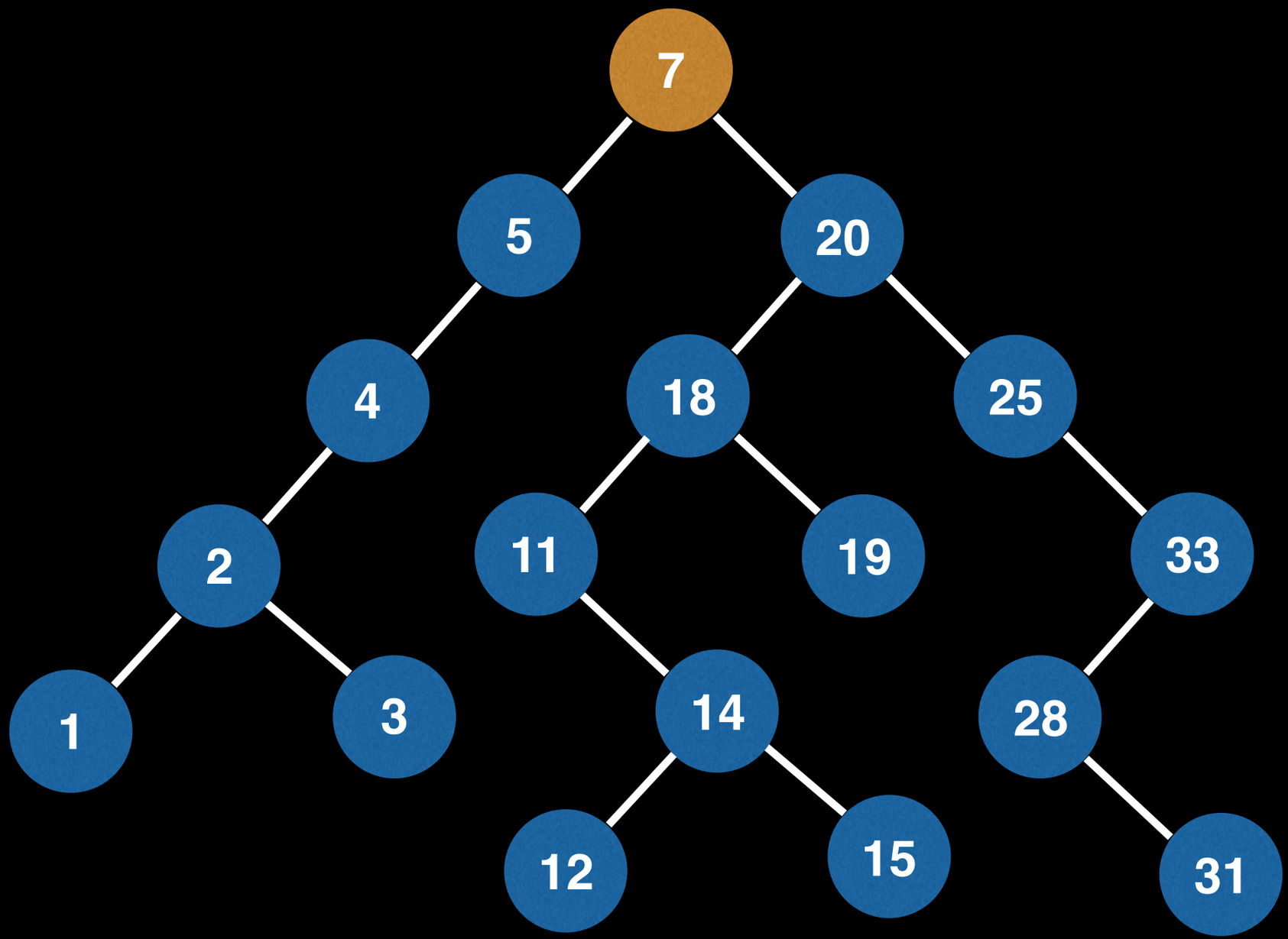
Let’s remove node 7. This is a case IV removal.

NOTE: This is a removal example for BSTs in general, not an AVL tree per se.

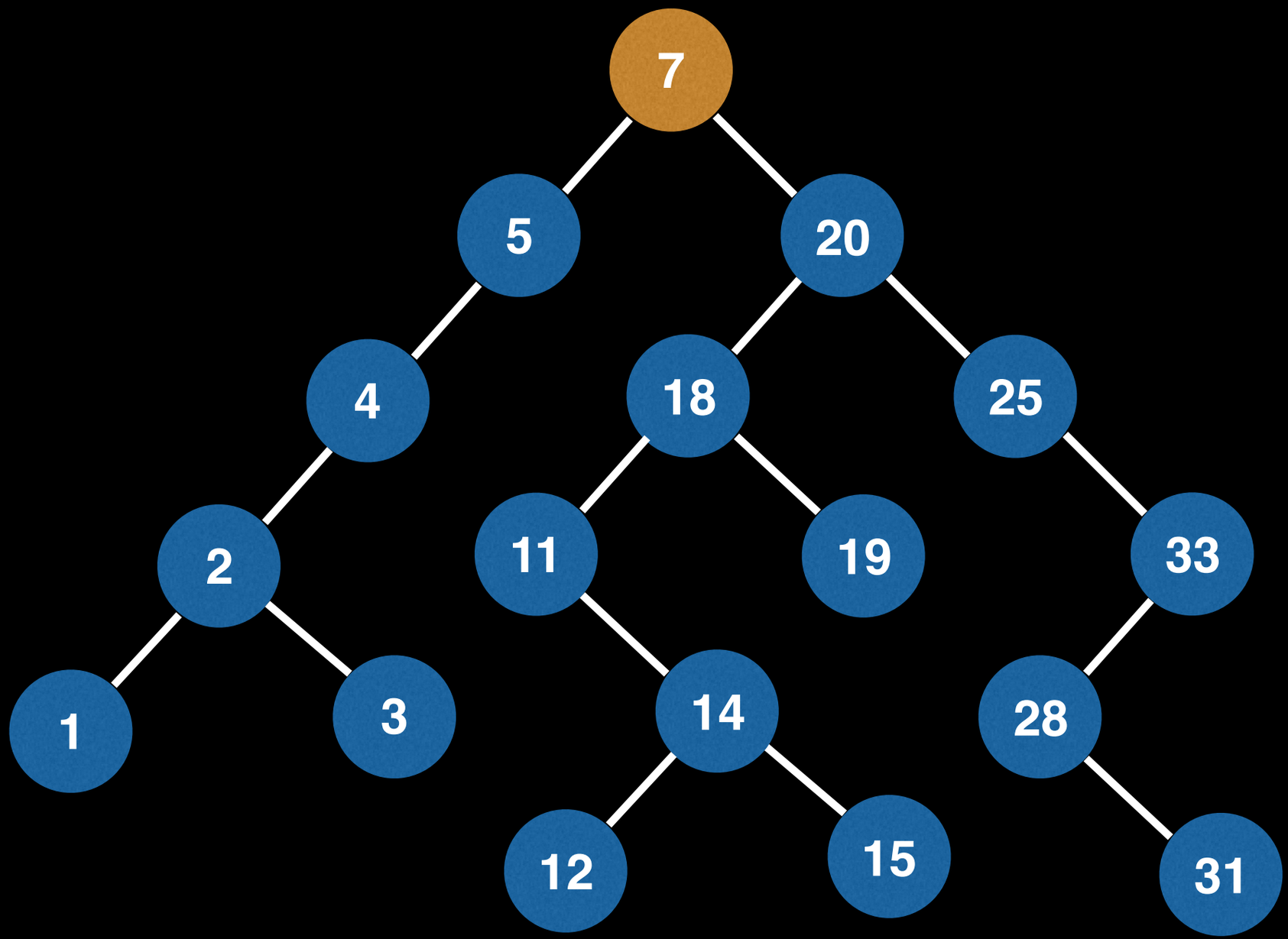


Let’s remove node 7. This is a case IV removal.

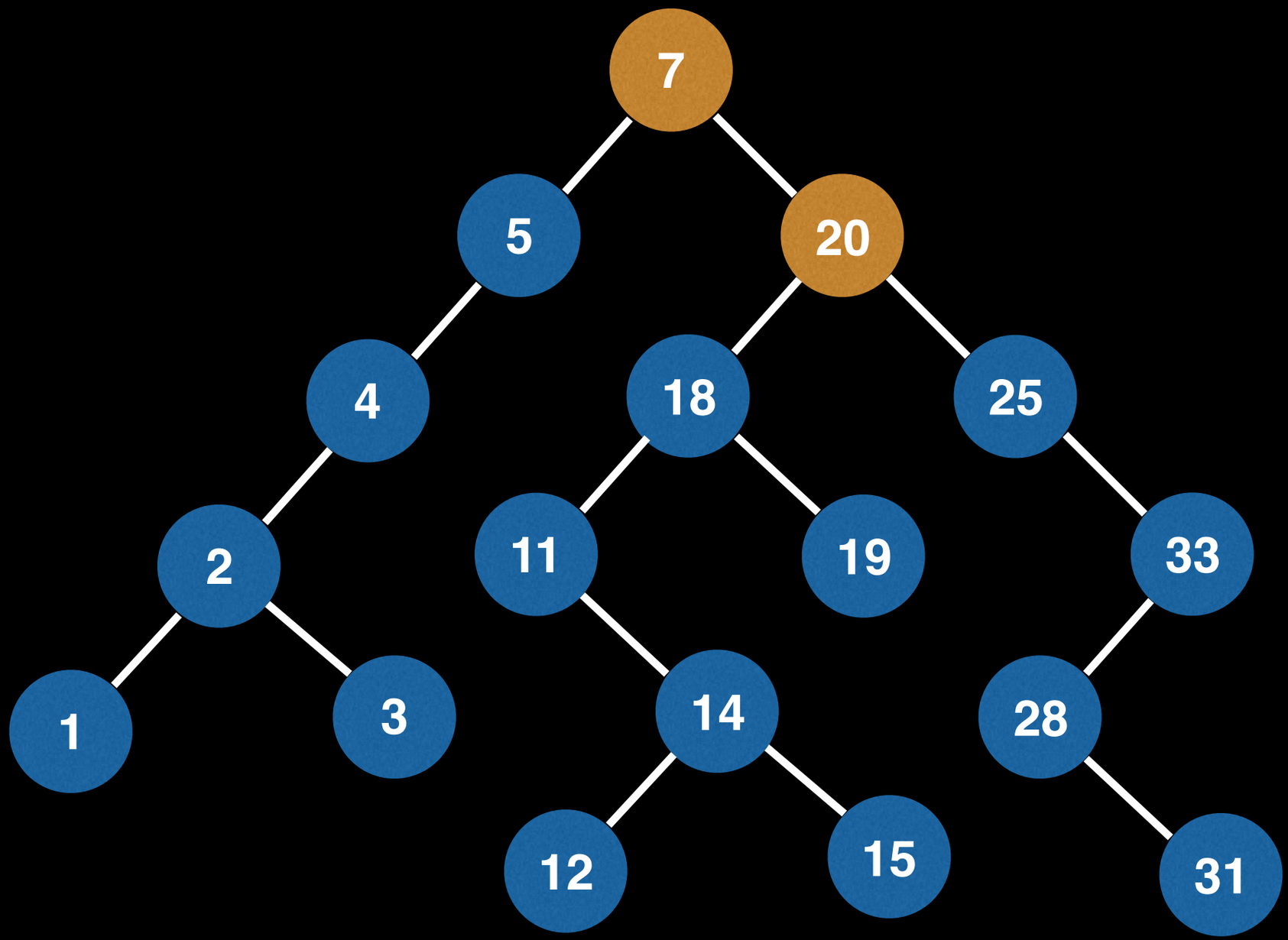
NOTE: This is a removal example for BSTs in general, not an AVL tree per se.



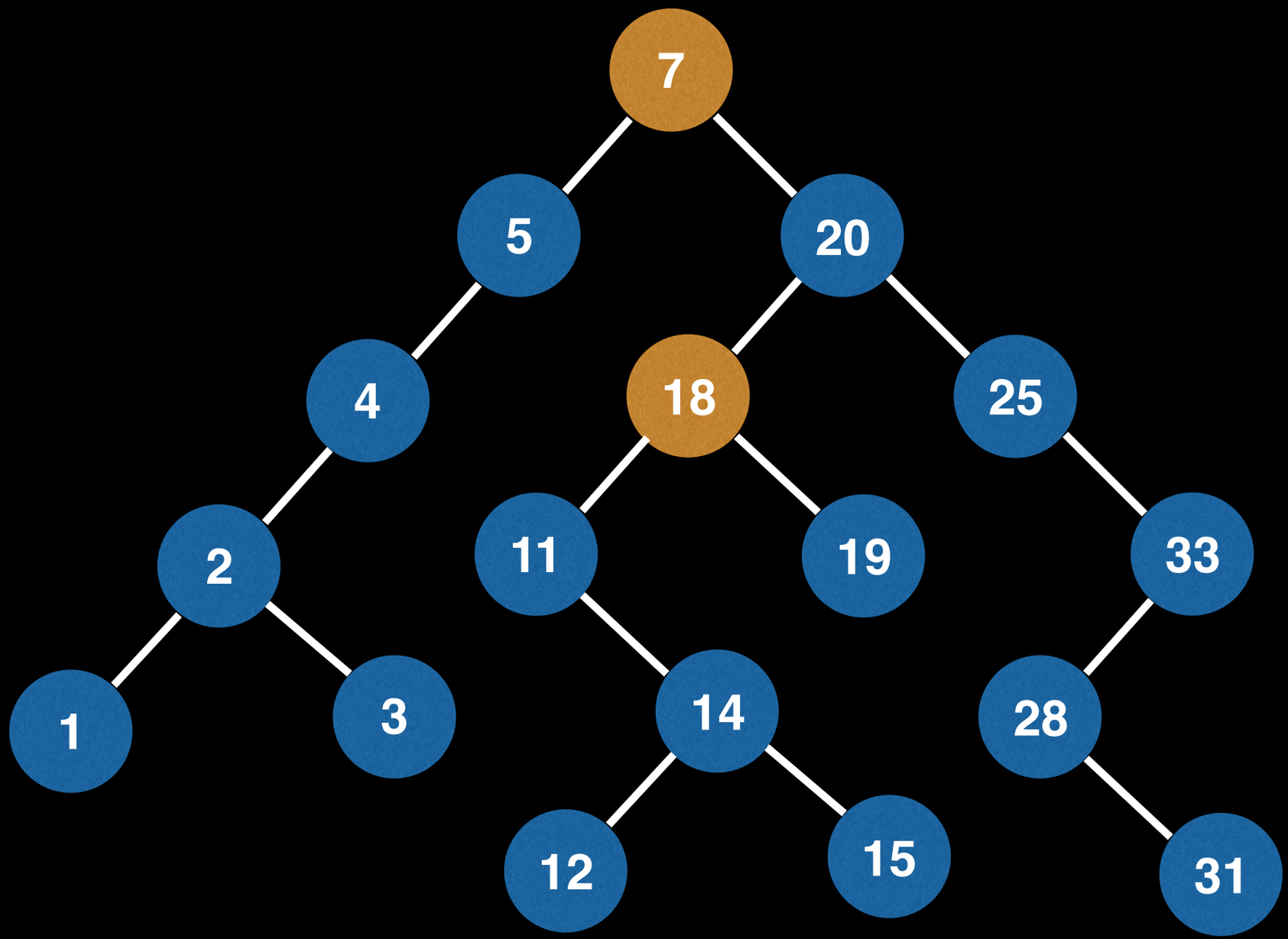
Now choose the successor to be either the smallest value in the right subtree or the largest in the left subtree. Let’s do the former. To do this, dig as far left as possible in the right subtree.



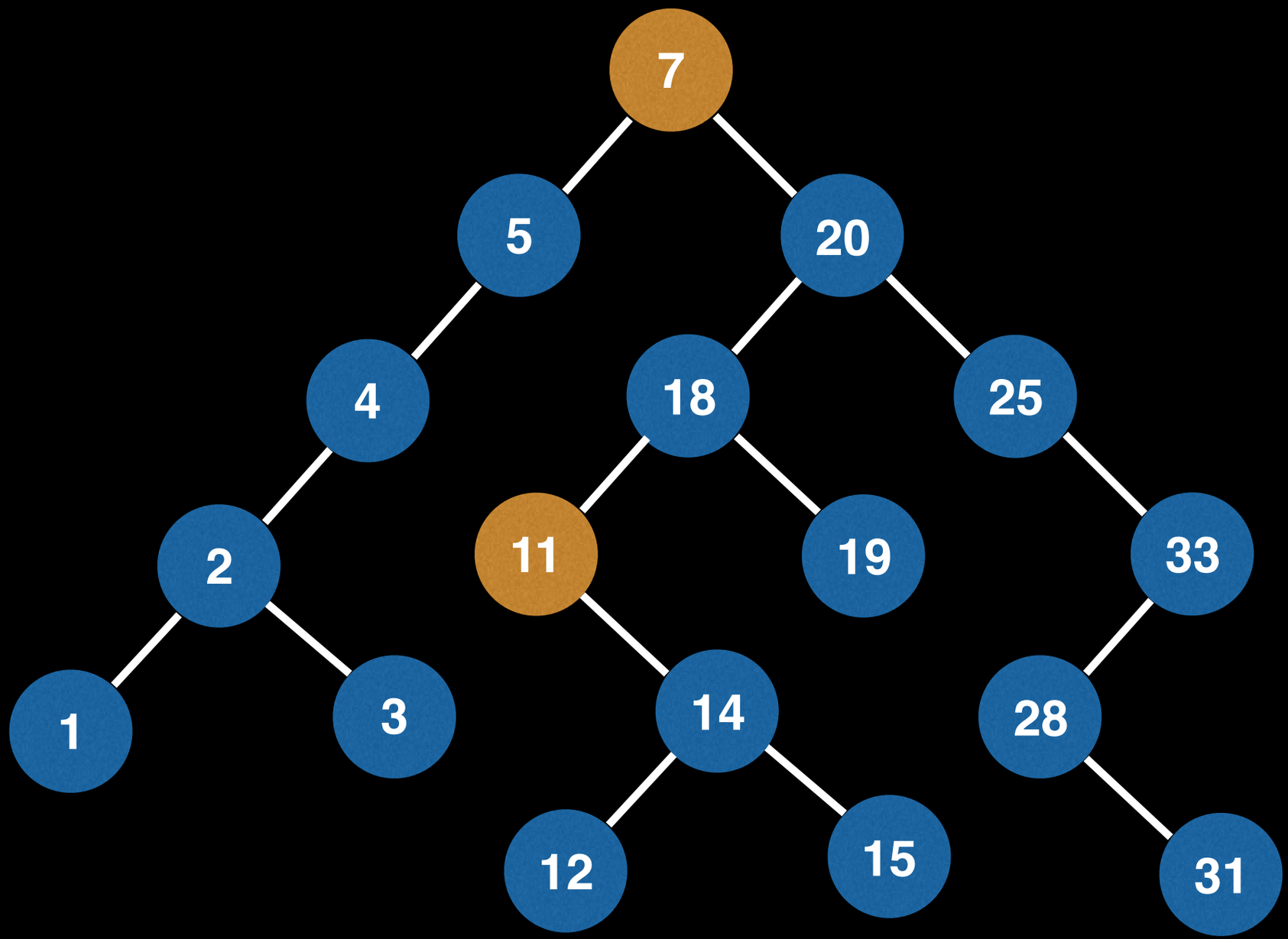
Now choose the successor to be either the smallest value in the right subtree or the largest in the left subtree. Let’s do the former. To do this, dig as far left as possible in the right subtree.



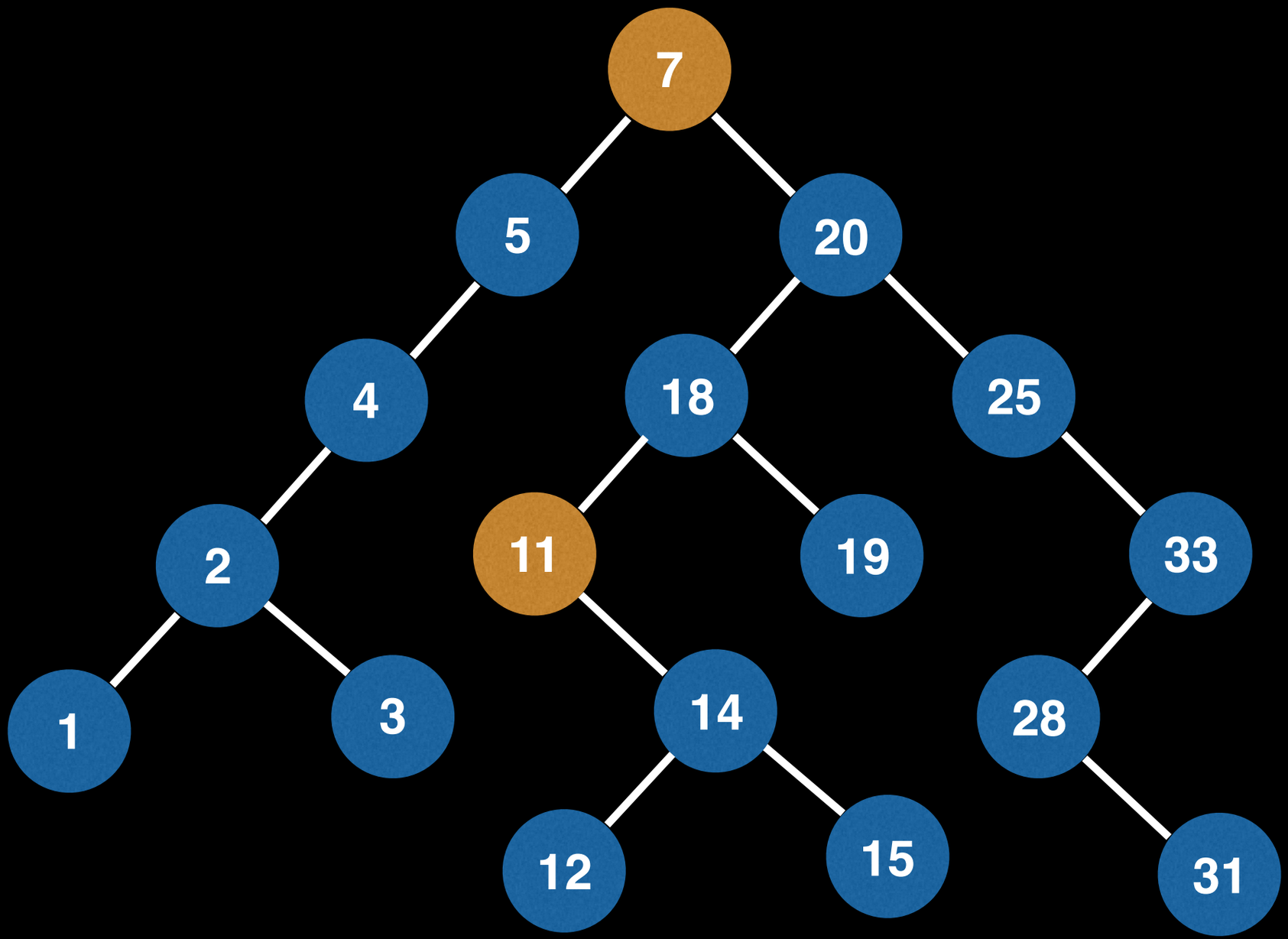
Now choose the successor to be either the smallest value in the right subtree or the largest in the left subtree. Let’s do the former. To do this, dig as far left as possible in the right subtree.



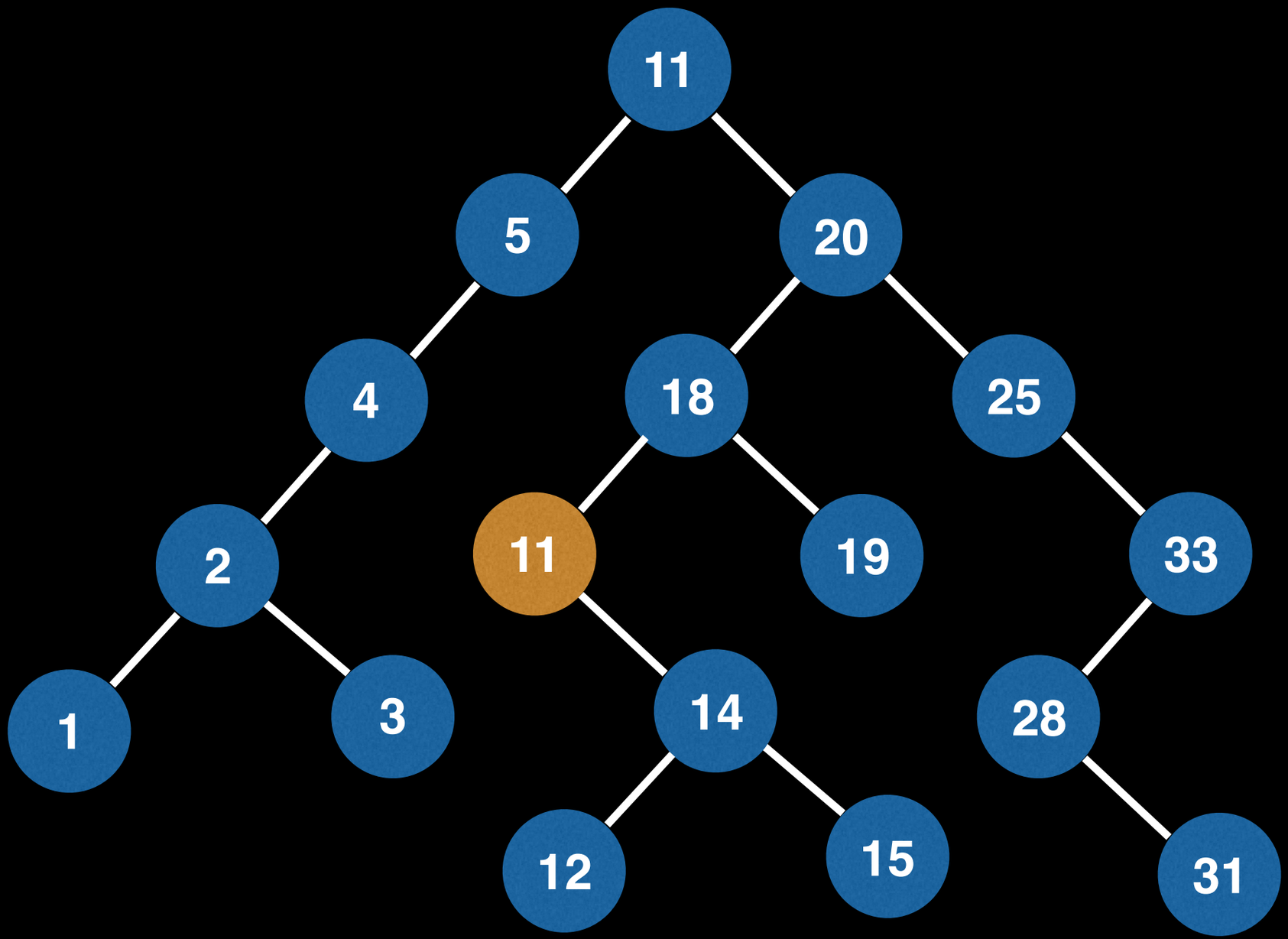
Now choose the successor to be either the smallest value in the right subtree or the largest in the left subtree. Let’s do the former. To do this, dig as far left as possible in the right subtree.



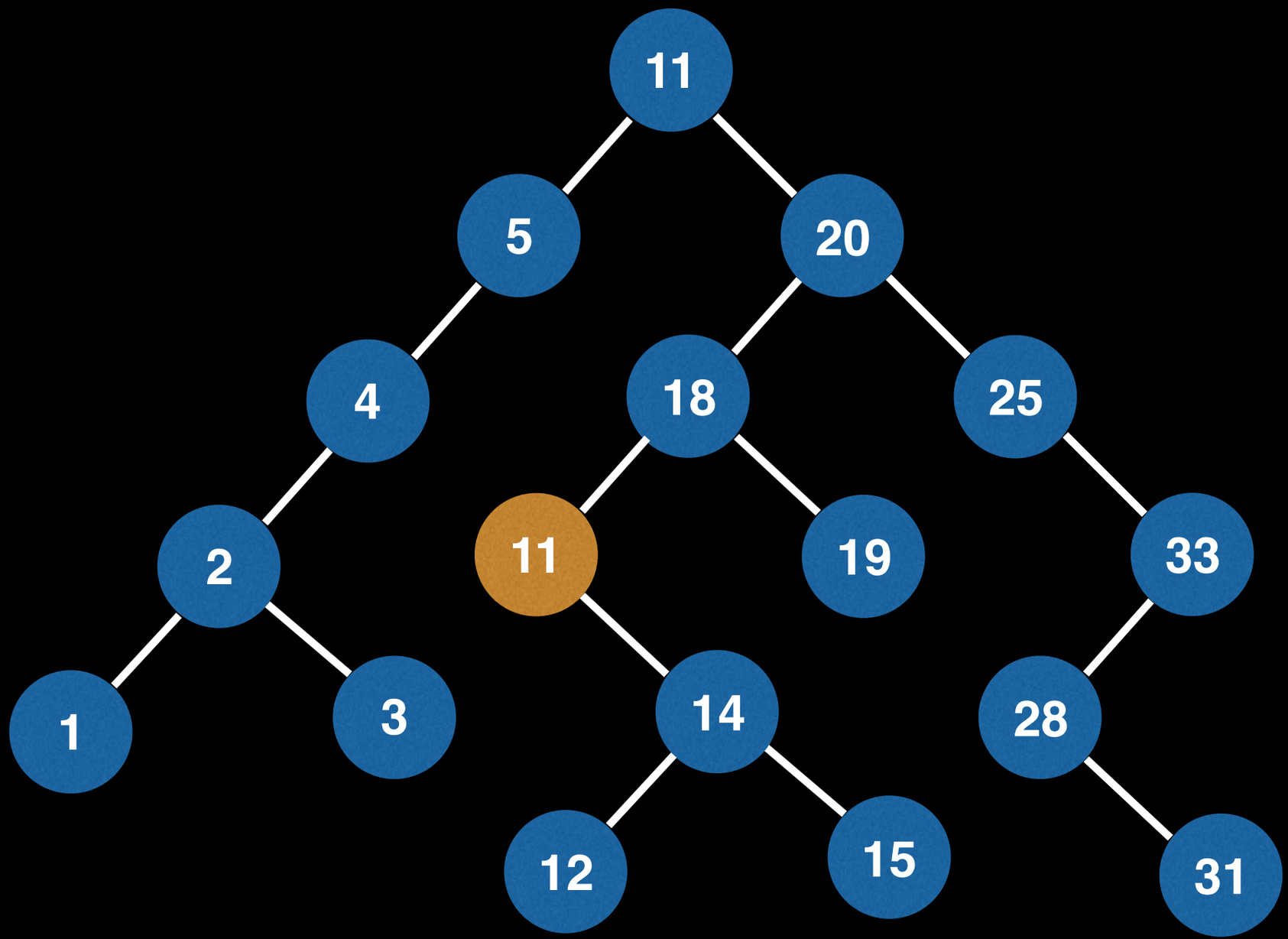
Copy the value from the node found in right subtree (11) to the node we want to remove.



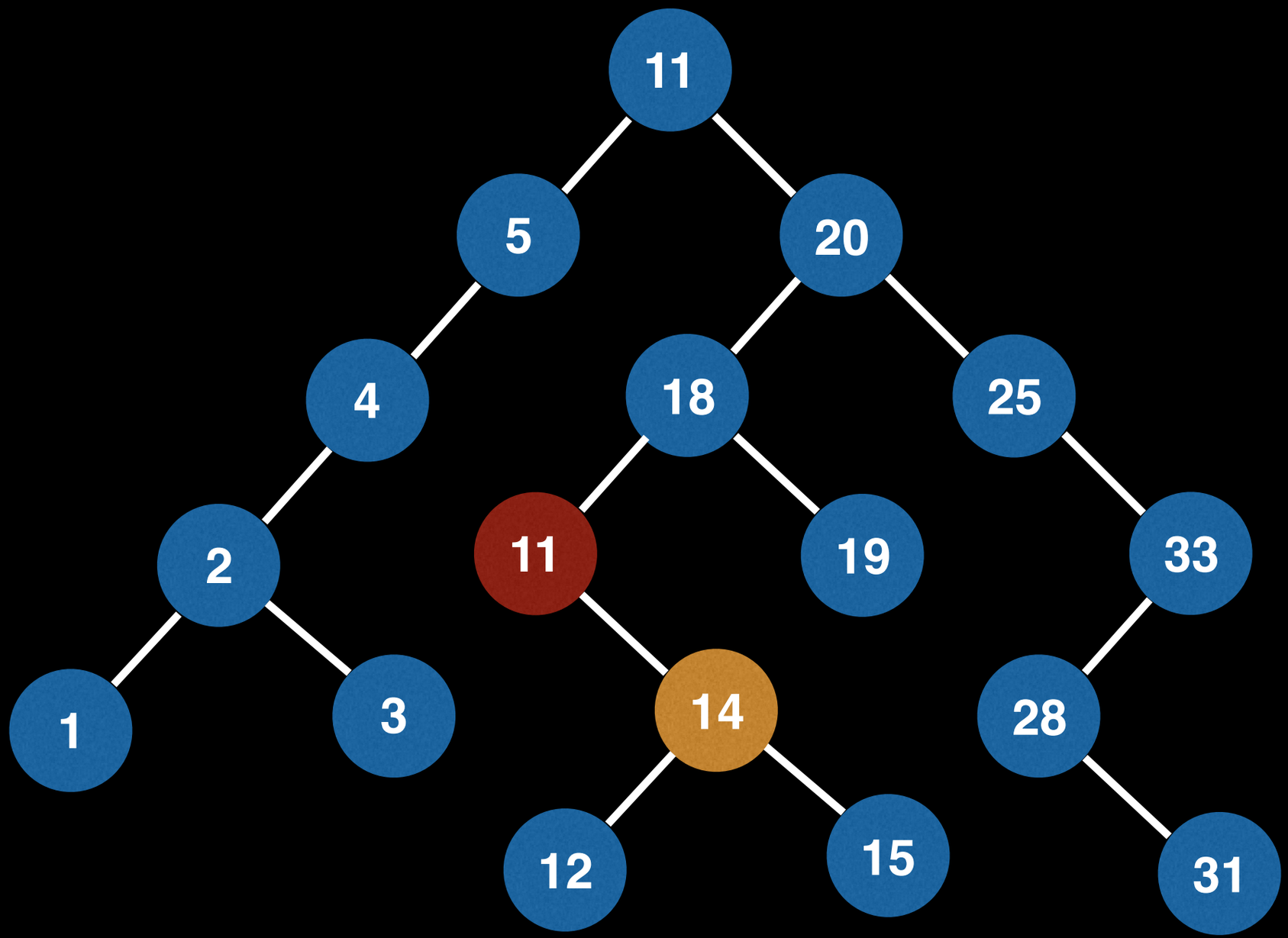
Copy the value from the node found in right subtree (11) to the node we want to remove.



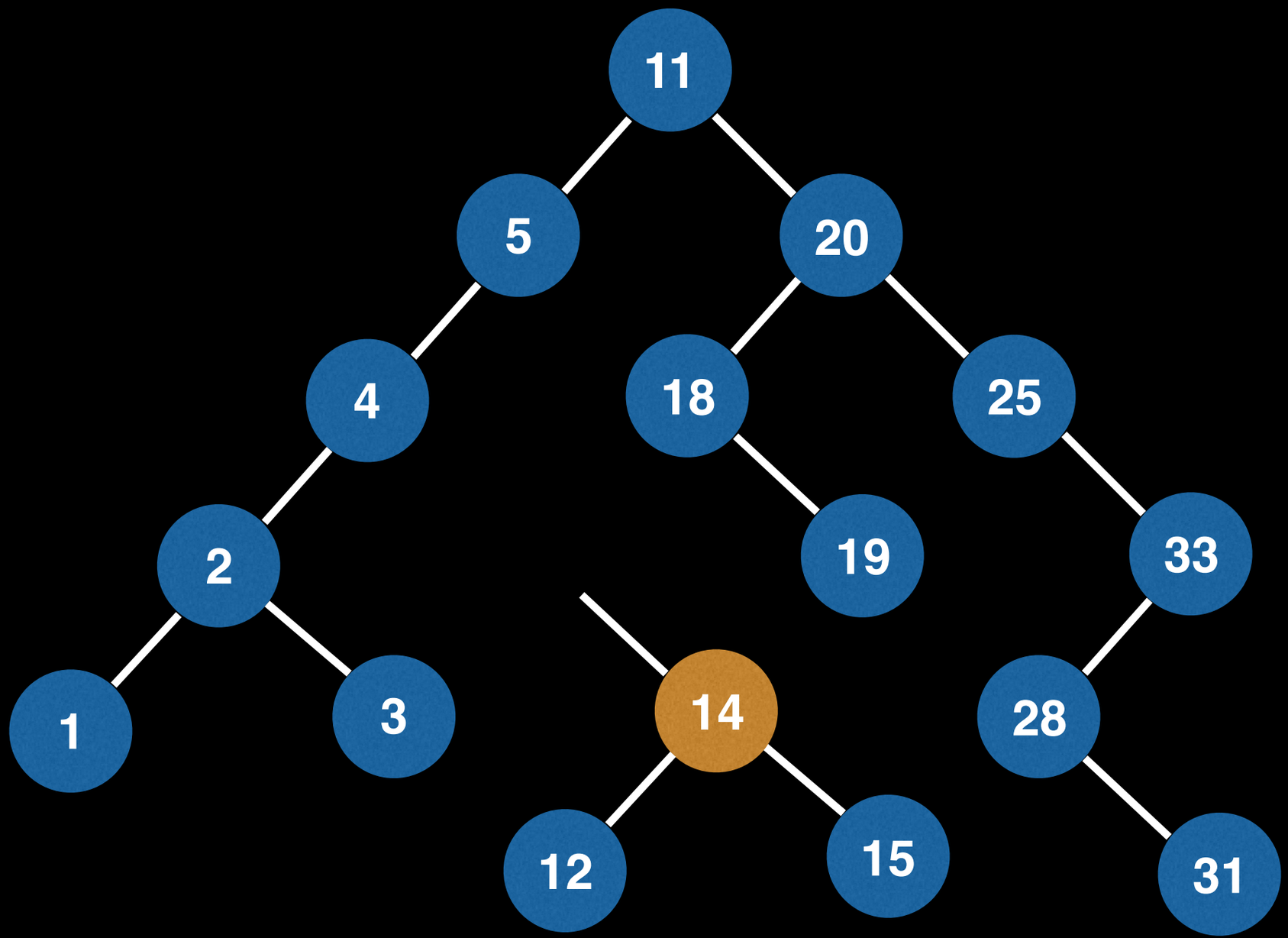
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal.



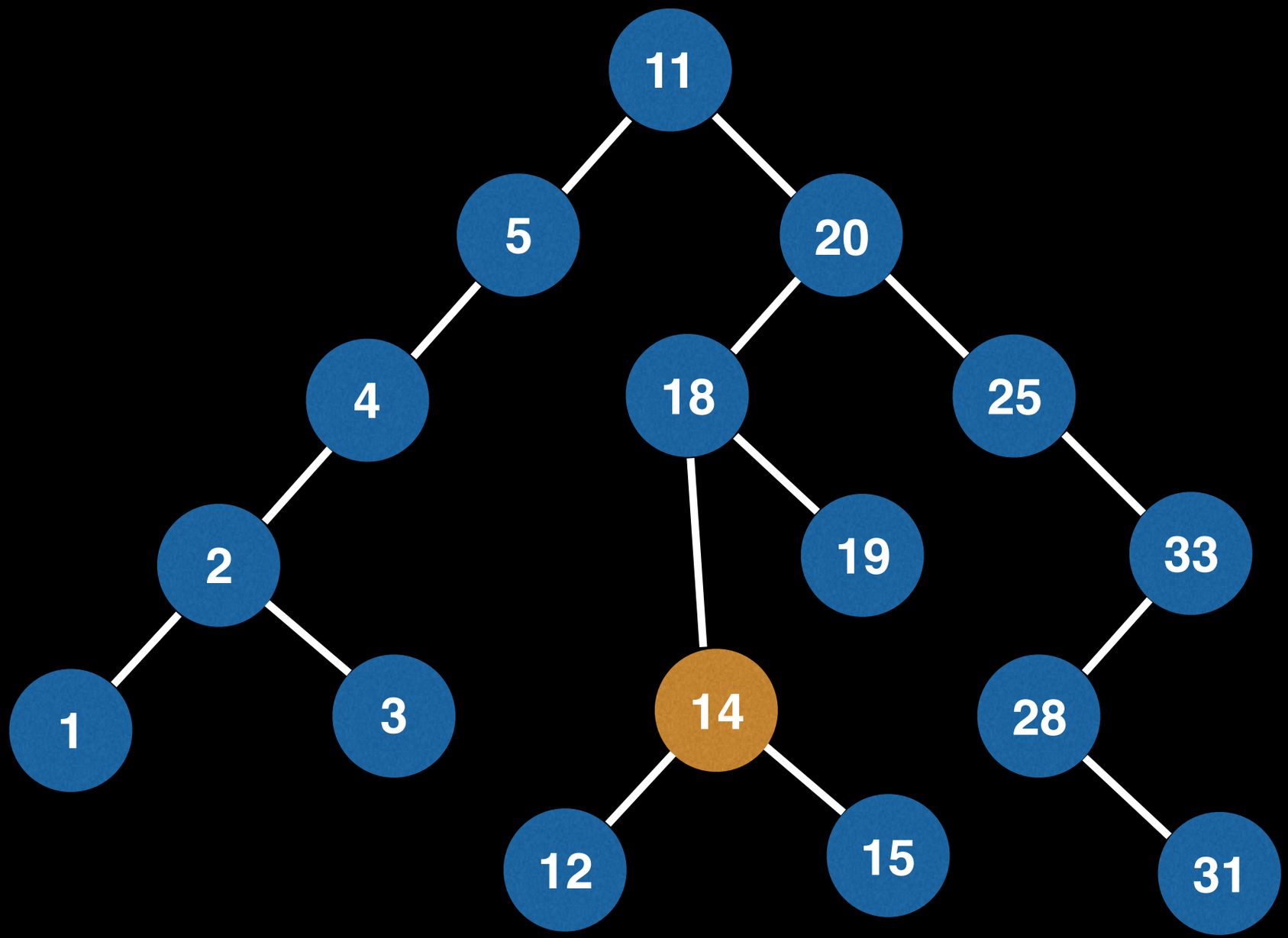
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal.



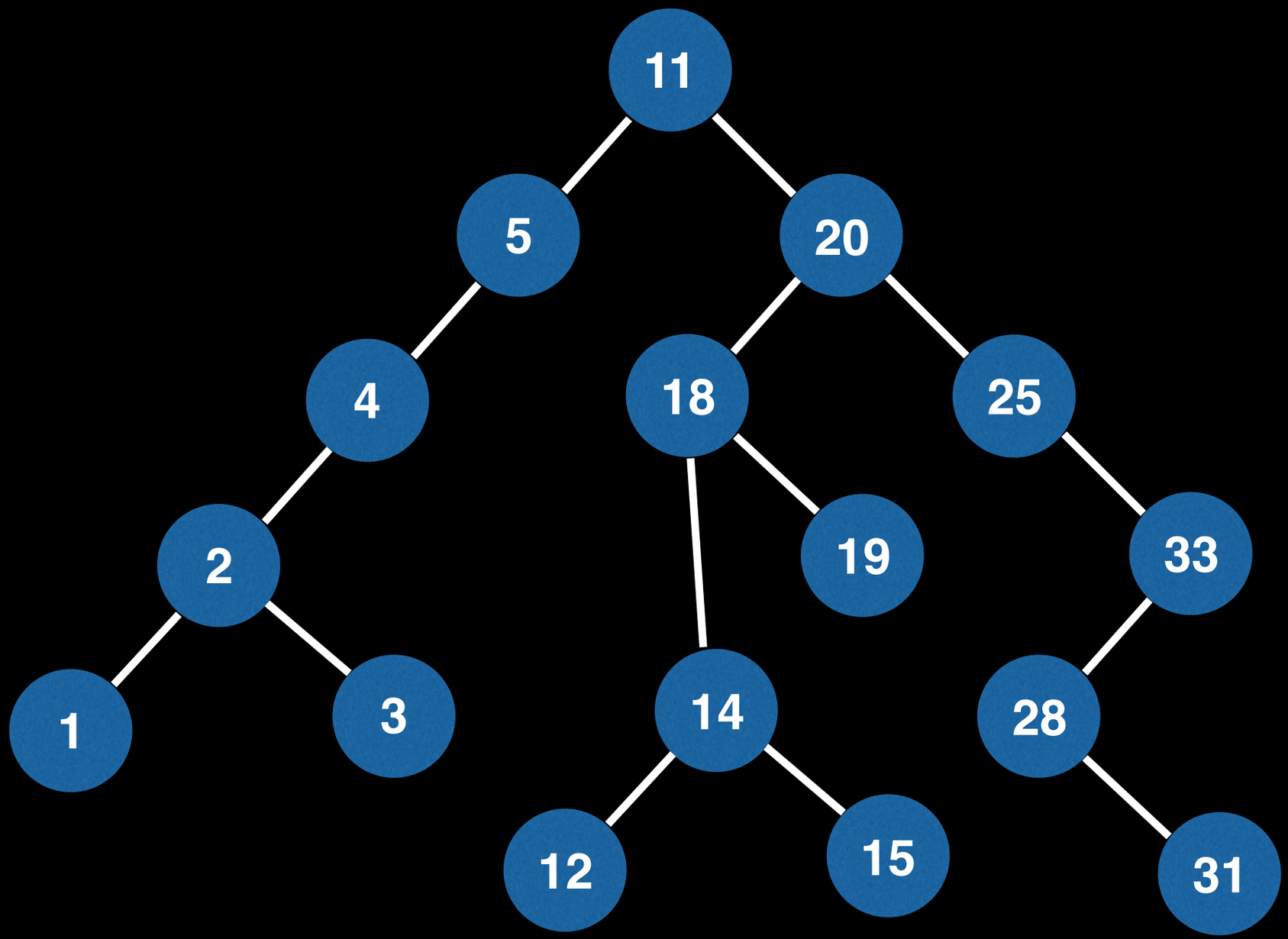
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal.



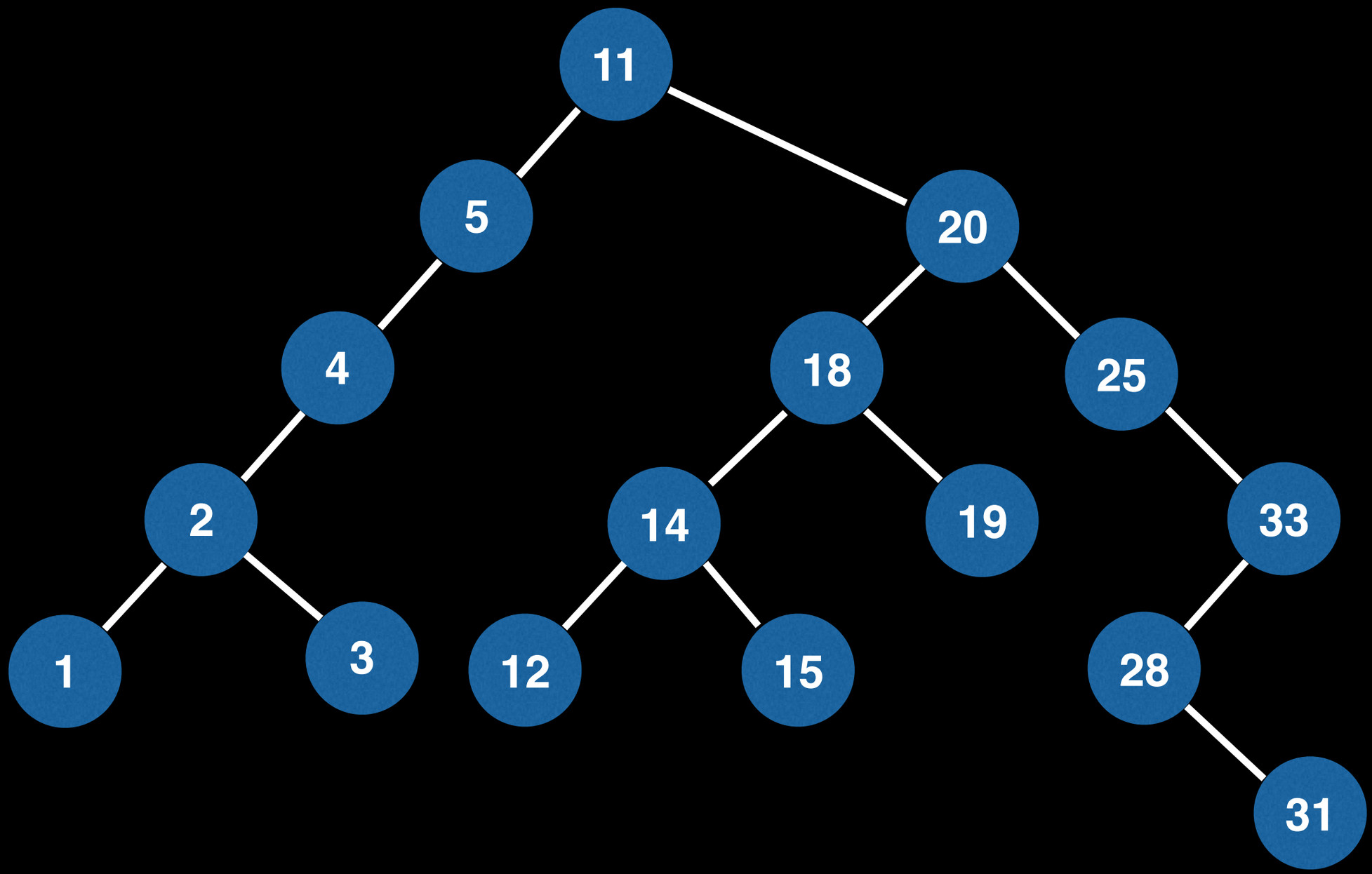
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal.



Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal.



Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal.



Augmenting BST Removal Algorithm for AVL Tree

Augmenting the removal algorithm from a plain BST implementation to an AVL tree is just as easy as adding two lines of code:

```
function remove(node, value):
    …
    # Code for BST item removal here
    …

    # Update balance factor
    update(node)

    # Rebalance tree
    return balance(node)
```

Next Video: AVL Tree Source Code

Source code for the AVL tree can be found at:

https://github.com/williamfiset/data-structures

AVL Tree Source Code

William Fiset

Source Code Link

Implementation source code and tests can all be found at the following link:

github.com/williamfiset/data-structures

NOTE: Make sure you have understood the previous video sections explaining how a AVL works before continuing!