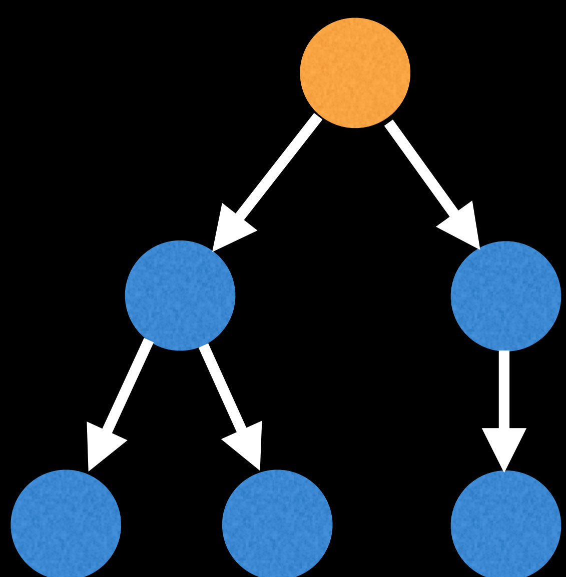
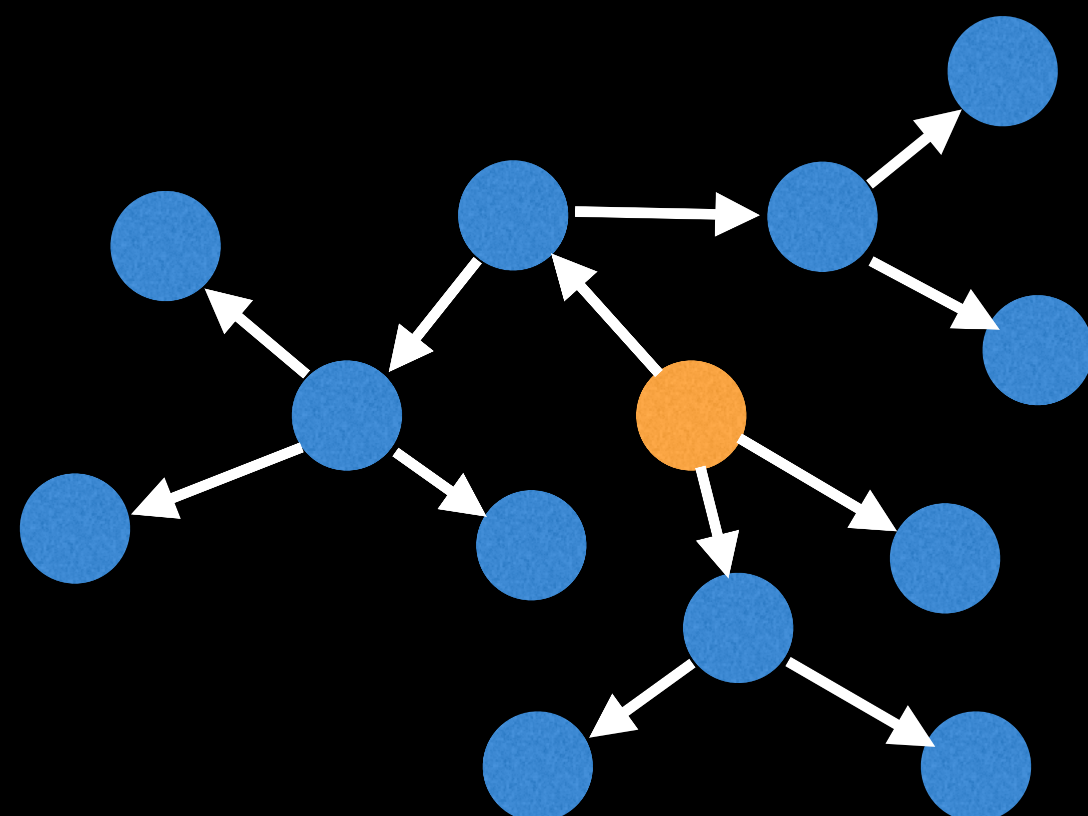
Rooted Trees!

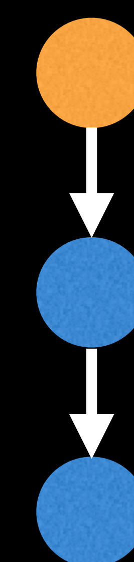
One of the more interesting types of trees is a **rooted tree** which is a tree with a designated **root node**.



Rooted tree



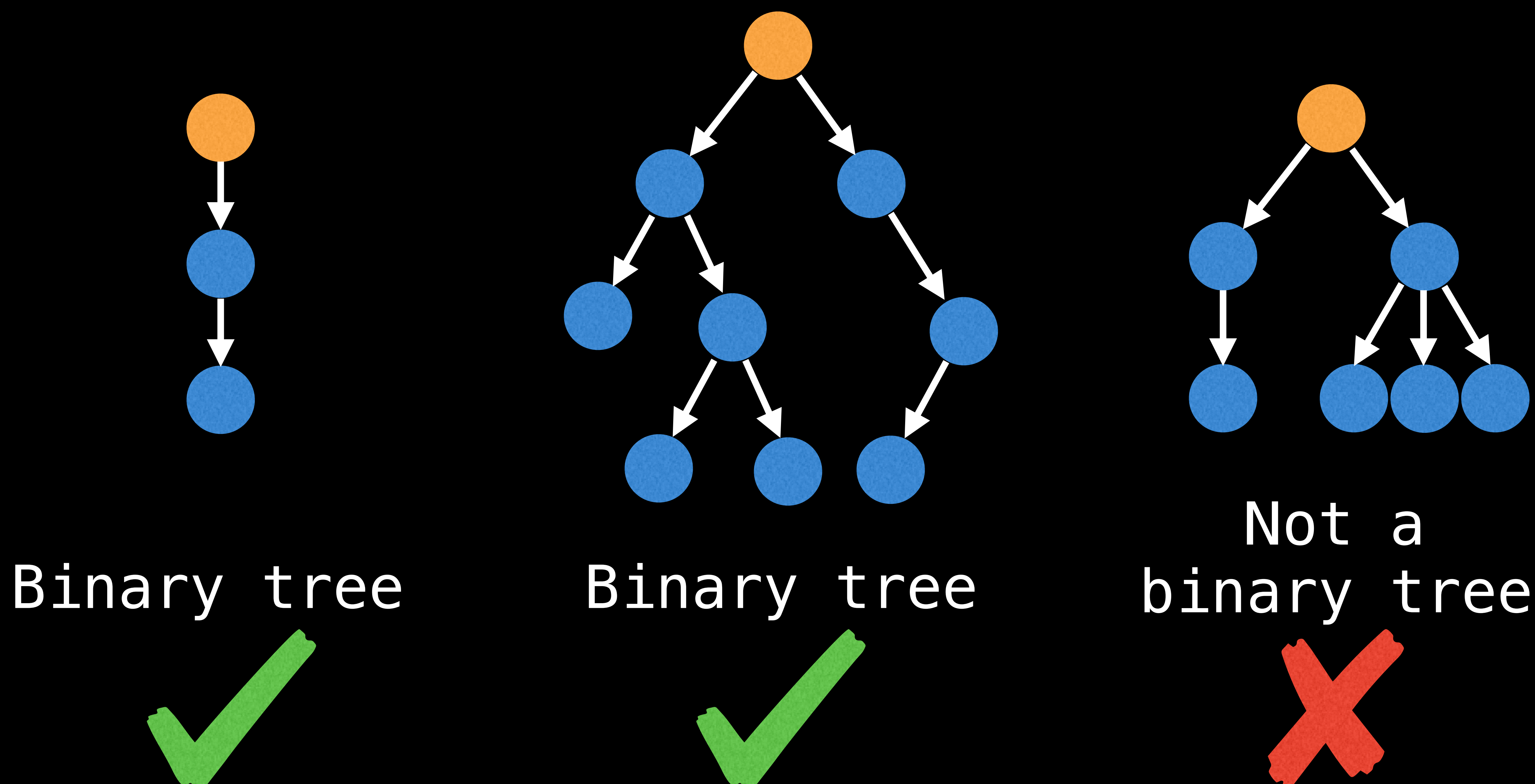
Rooted tree



Rooted tree

Binary Tree (BT)

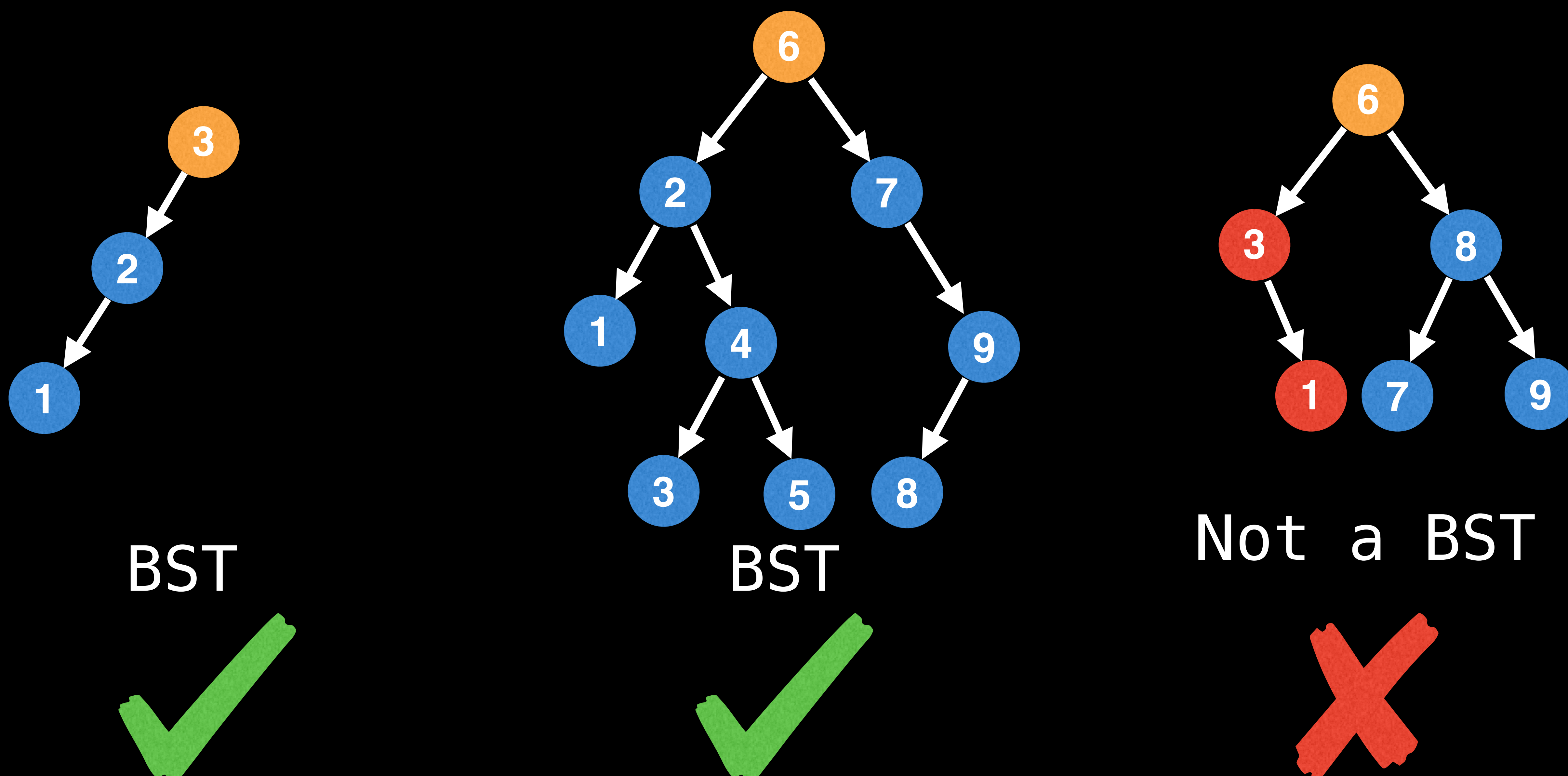
Related to rooted trees are **binary trees** which are trees for which every node has **at most two child nodes**.



Binary Search Trees (BST)

Related to binary trees are **binary search trees** which are trees which satisfy the BST invariant which states that for every node x:

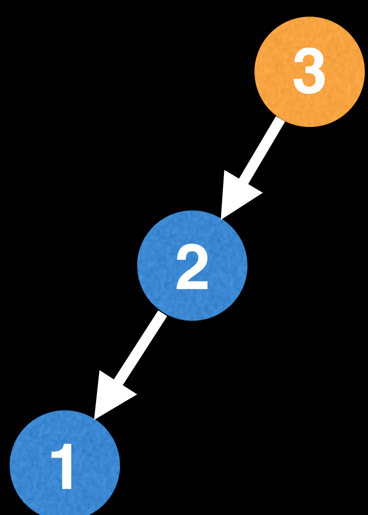
x.left.value ≤ x.value ≤ x.right.value



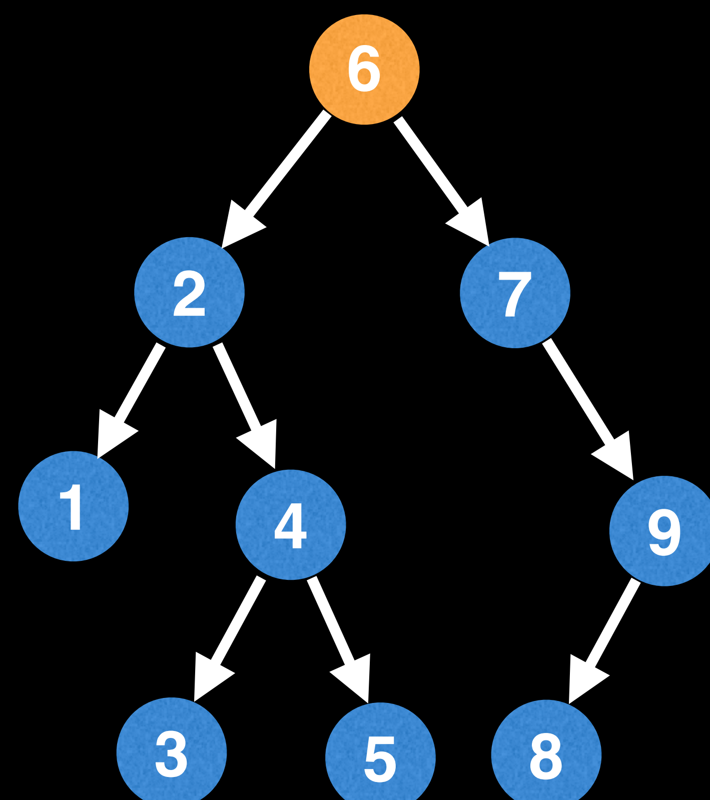
Binary Search Trees (BST)

It’s often useful to **require uniqueness** on the node values in your tree. Change the invariant to be strictly < rather than ≤:

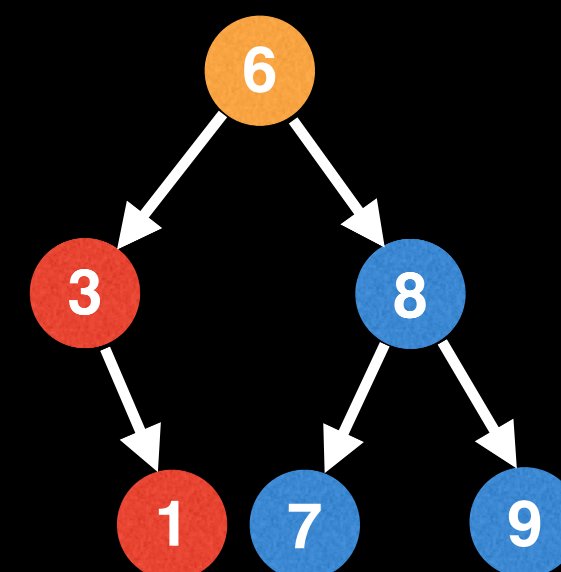
x.left.value < x.value < x.right.value



BST



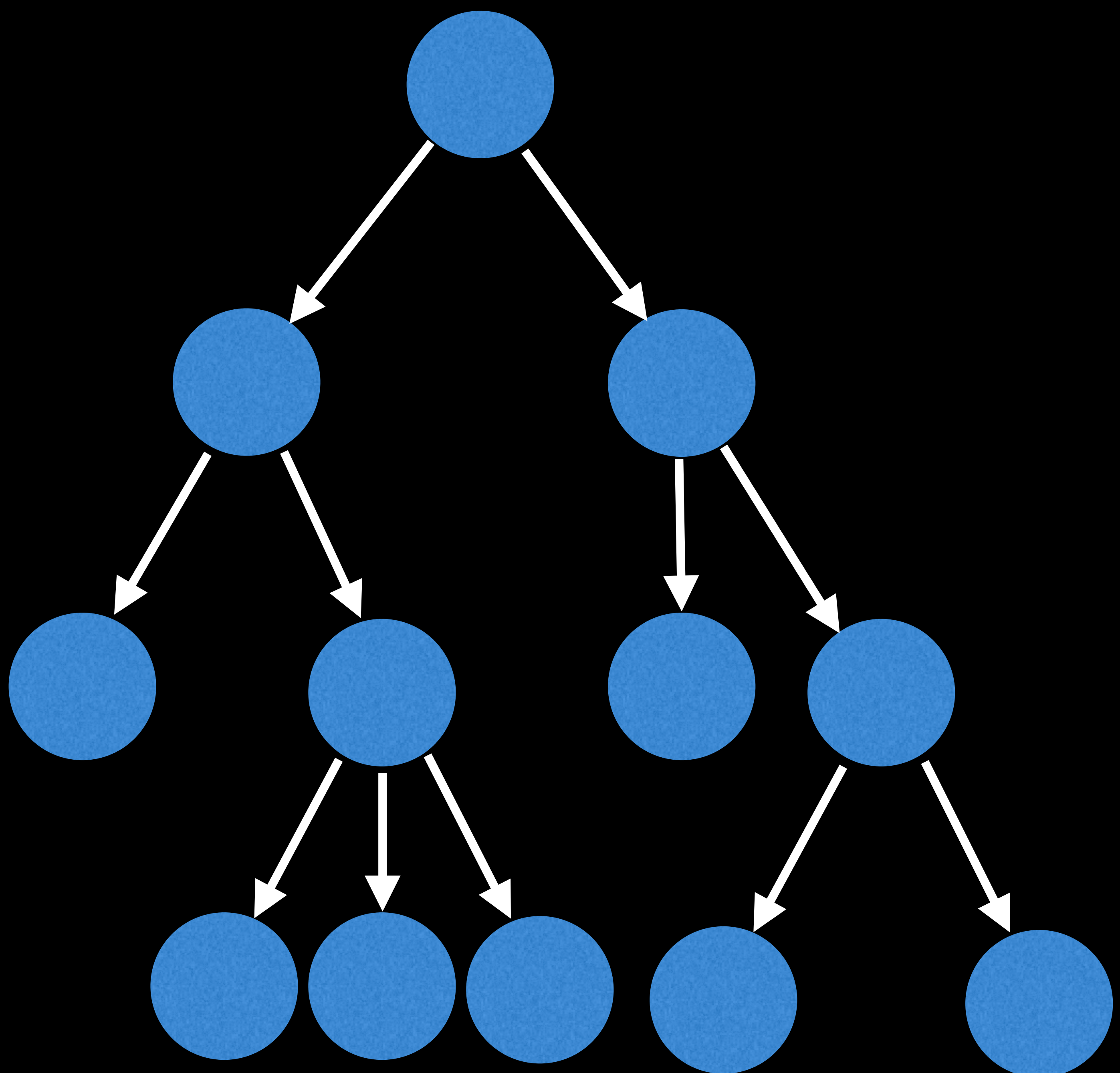
BST



Not a BST

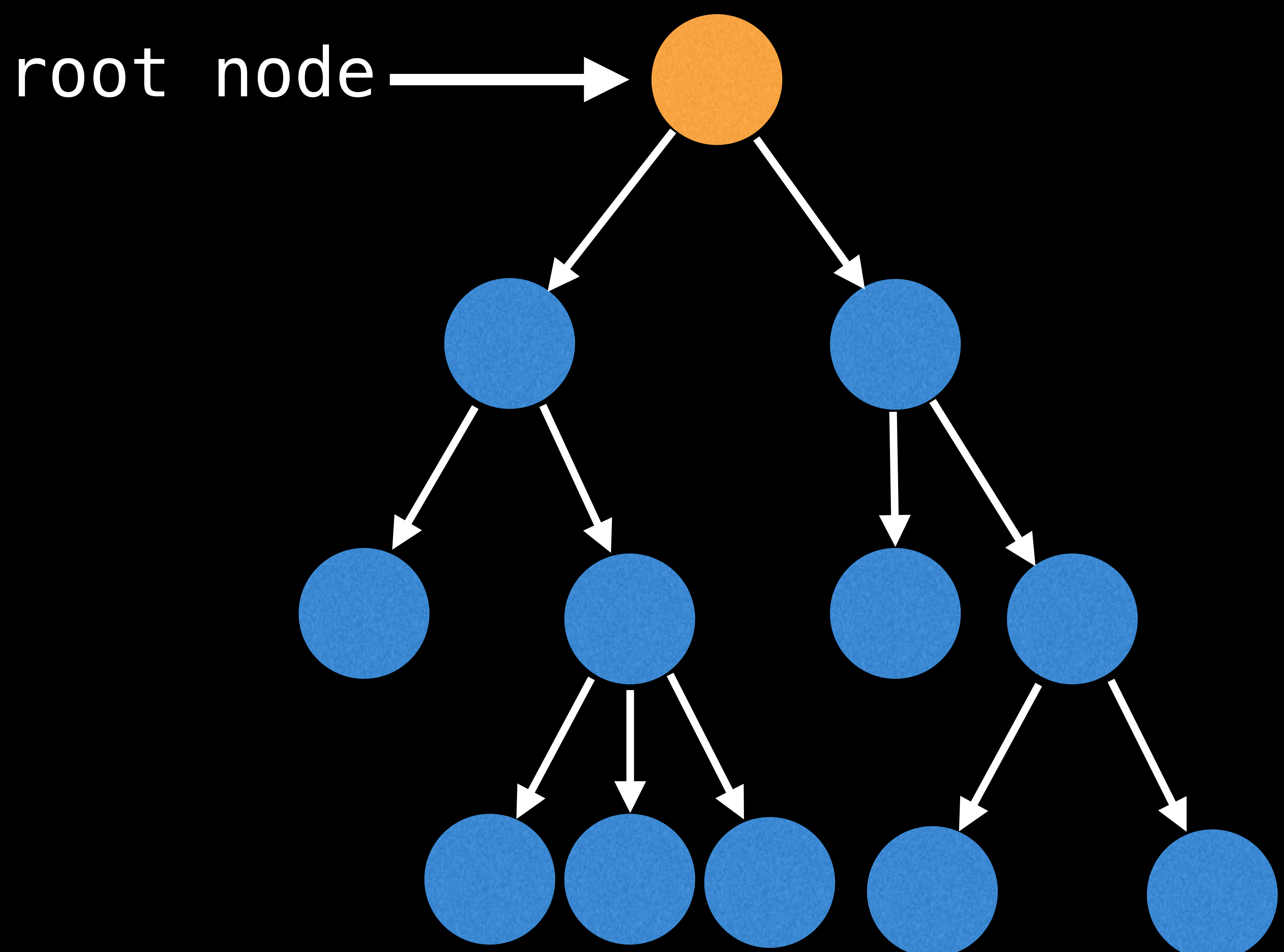
Storing rooted trees

Rooted trees are most naturally defined recursively in a top-down manner.



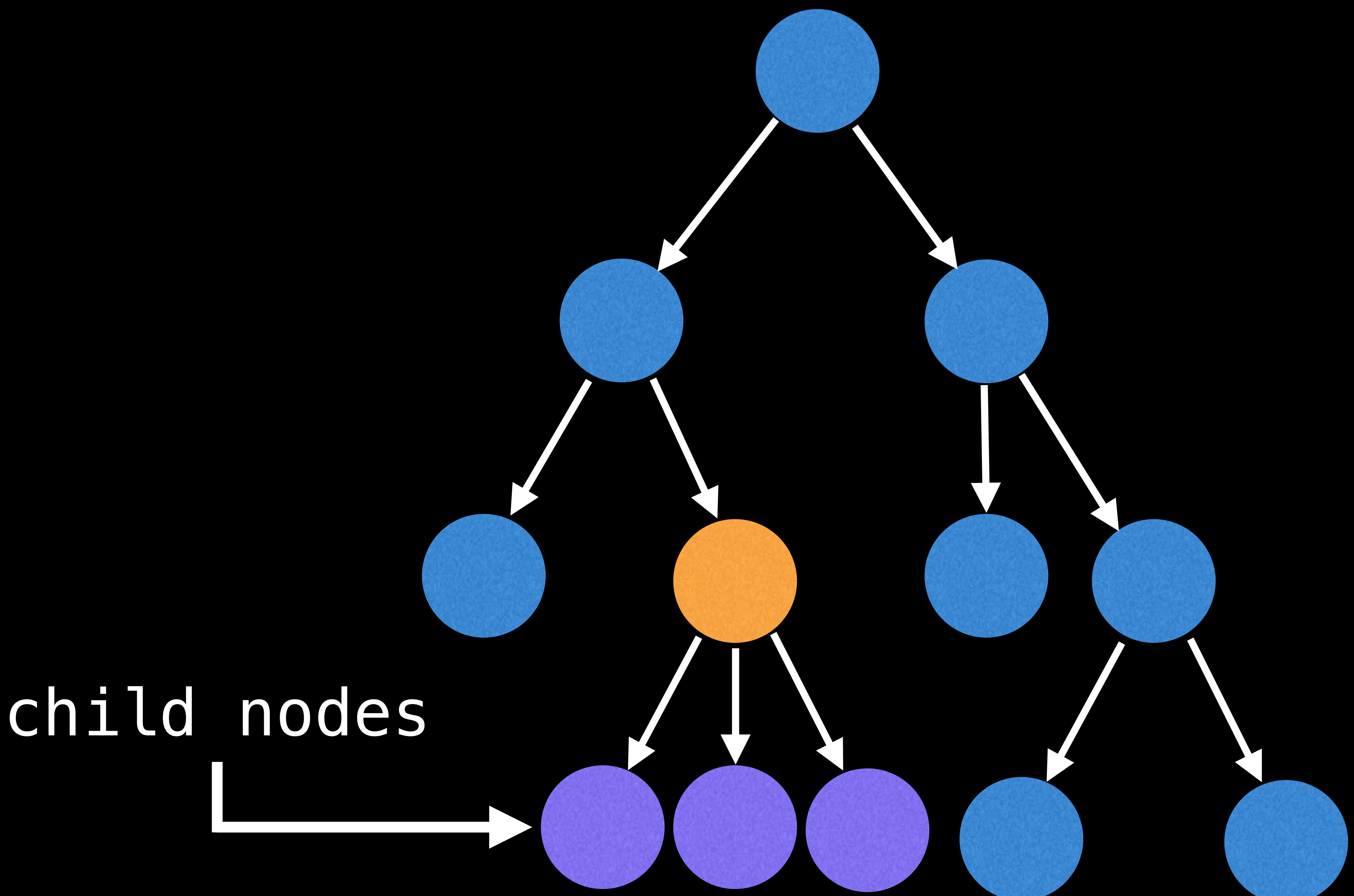
Storing rooted trees

In practice, you always maintain a pointer reference to the **root node** so that you can access the tree and its contents.



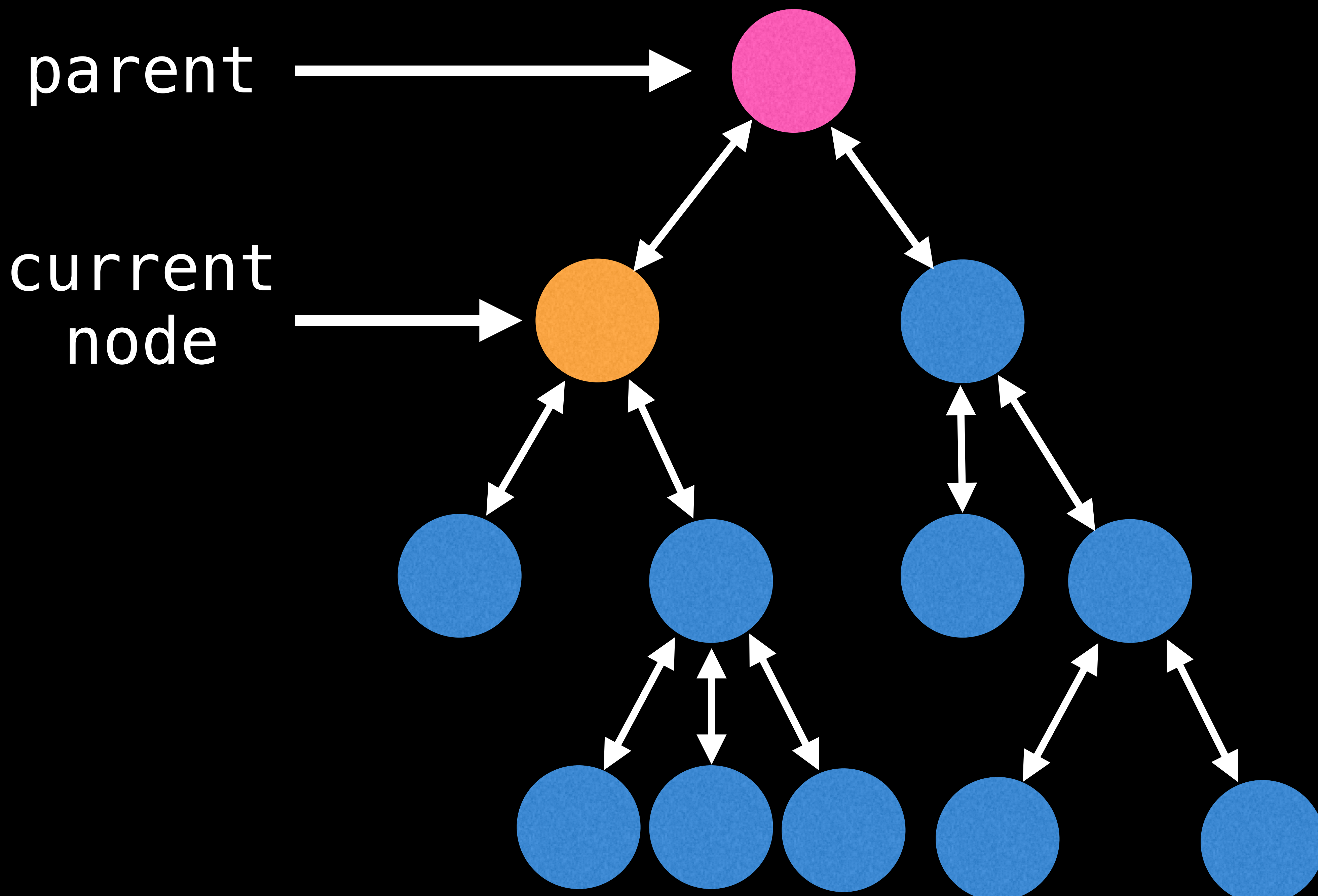
Storing rooted trees

Each node also has access to a list of all its **children**.



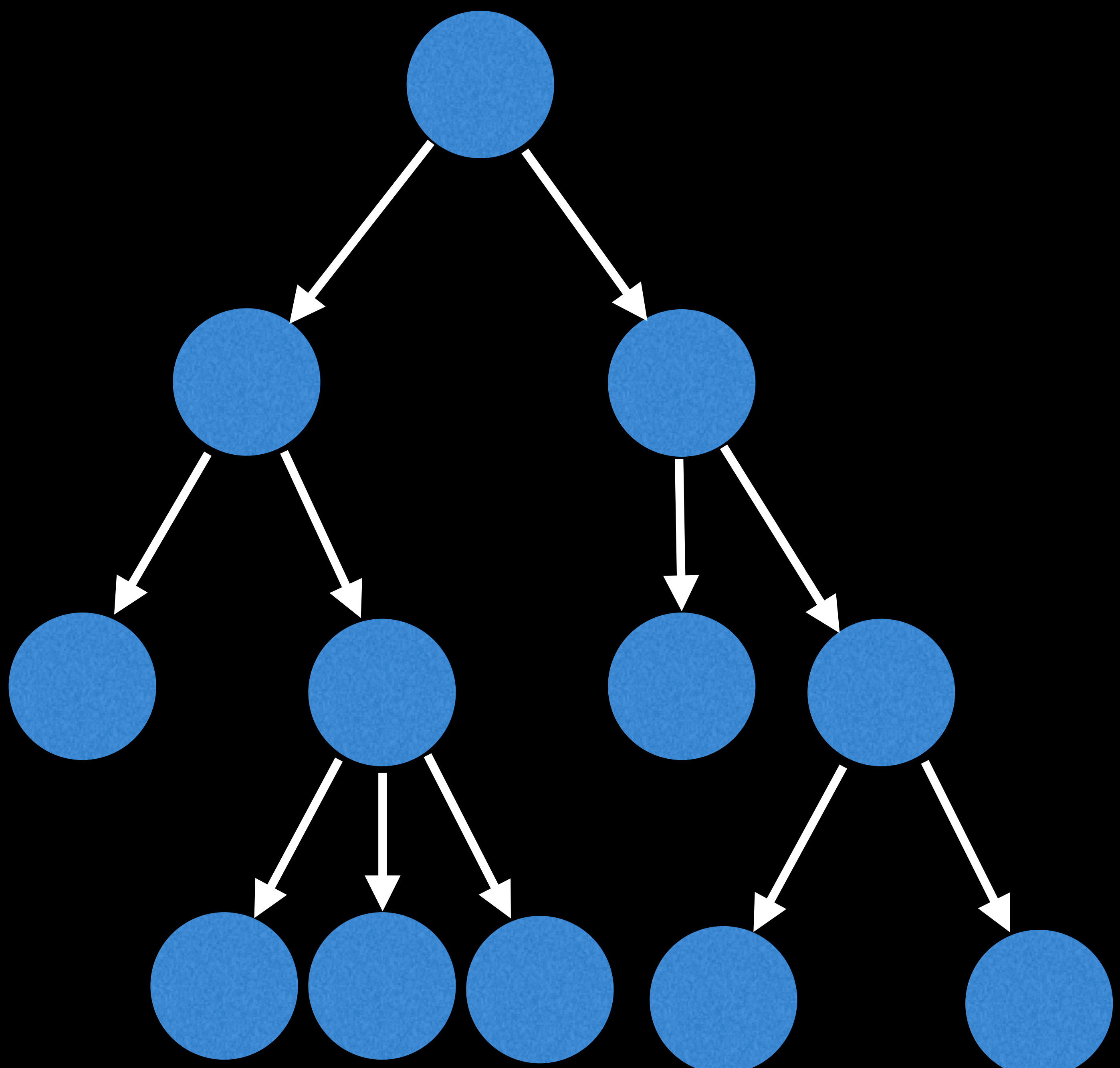
Storing rooted trees

Sometimes it's also useful to maintain a pointer to a node's **parent node** effectively making edges **bidirectional**.



Storing rooted trees

However, this isn't *usually* necessary because you can access a node's parent on a recursive function's callback.



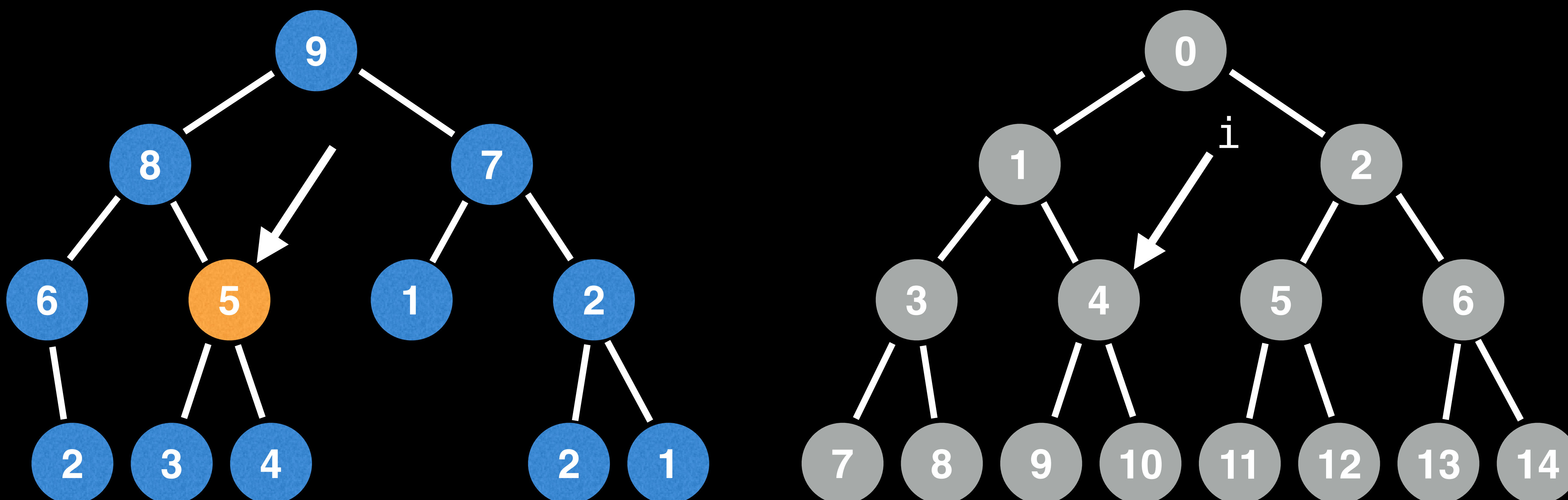
Storing rooted trees

If your tree is a **binary tree**, you can store it in a **flattened array**.

This trick also works for any n-ary tree

Storing rooted trees

In this flattened array representation, each node has an assigned index position based on where it is in the tree.

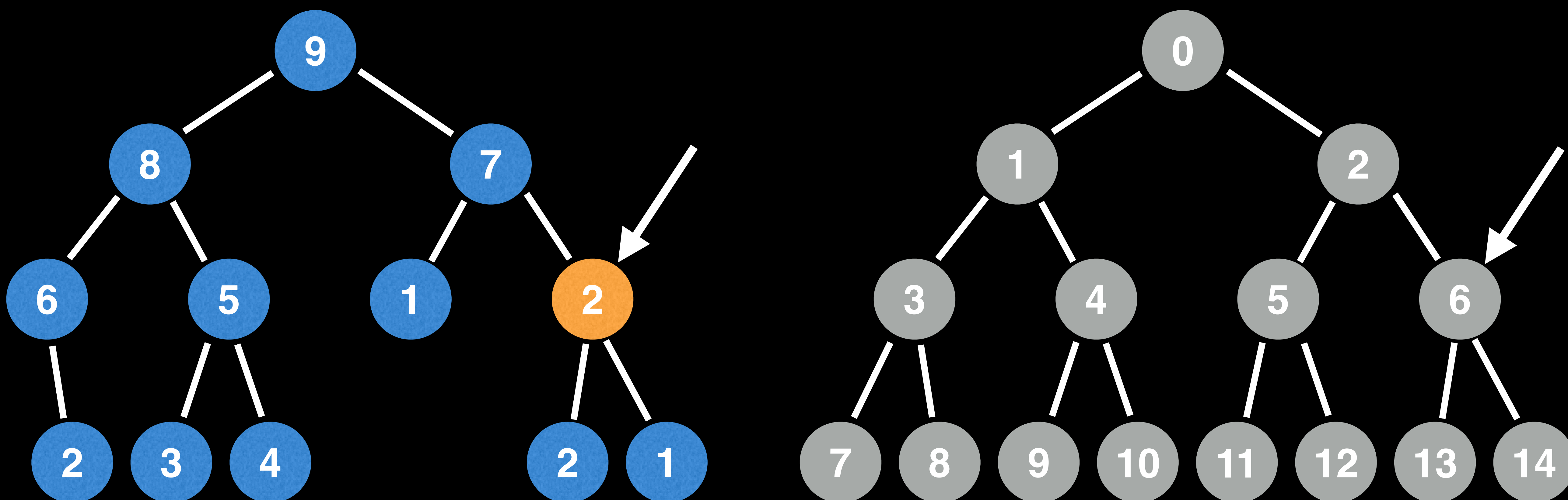


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

In this flattened array representation, each node has an assigned index position based on where it is in the tree.

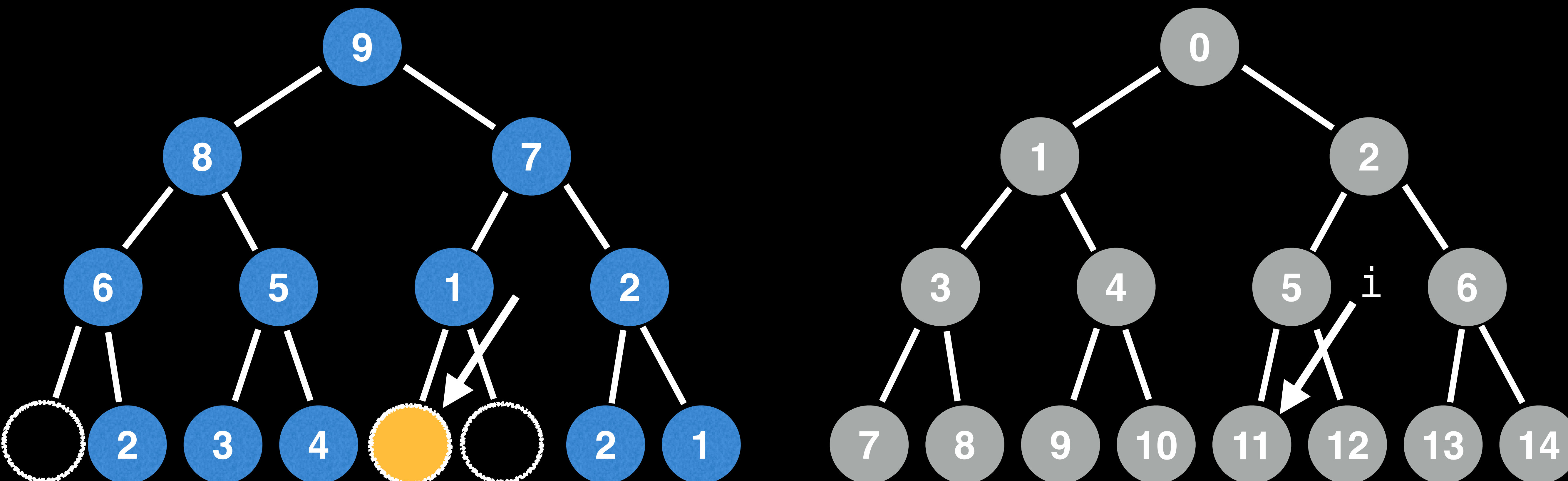


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

Even nodes which aren't currently present have an index because they can be mapped back to a unique position in the "index tree" (gray tree).



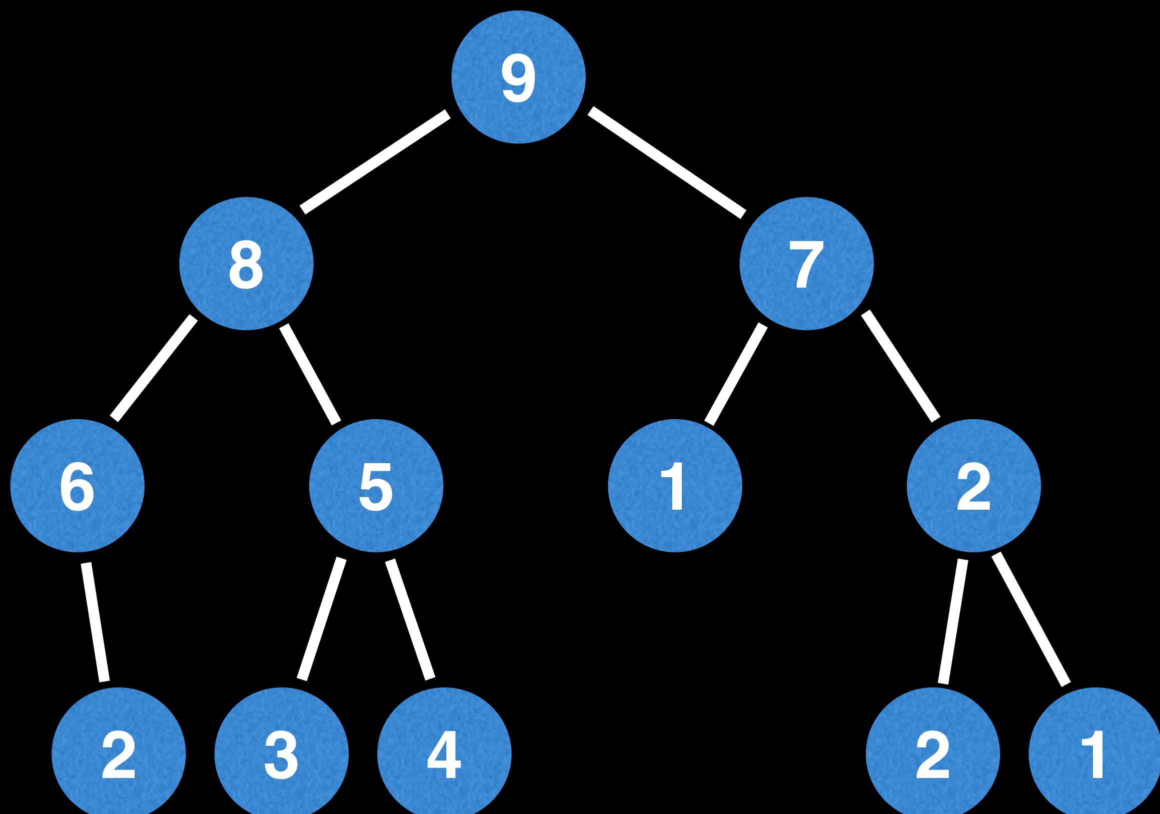
index tree

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

The root node is always at index 0 and the children of the current node *i* are accessed relative to position *i*.



Let *i* be the index of the current node

left node: 2*i + 1
right node: 2*i + 2

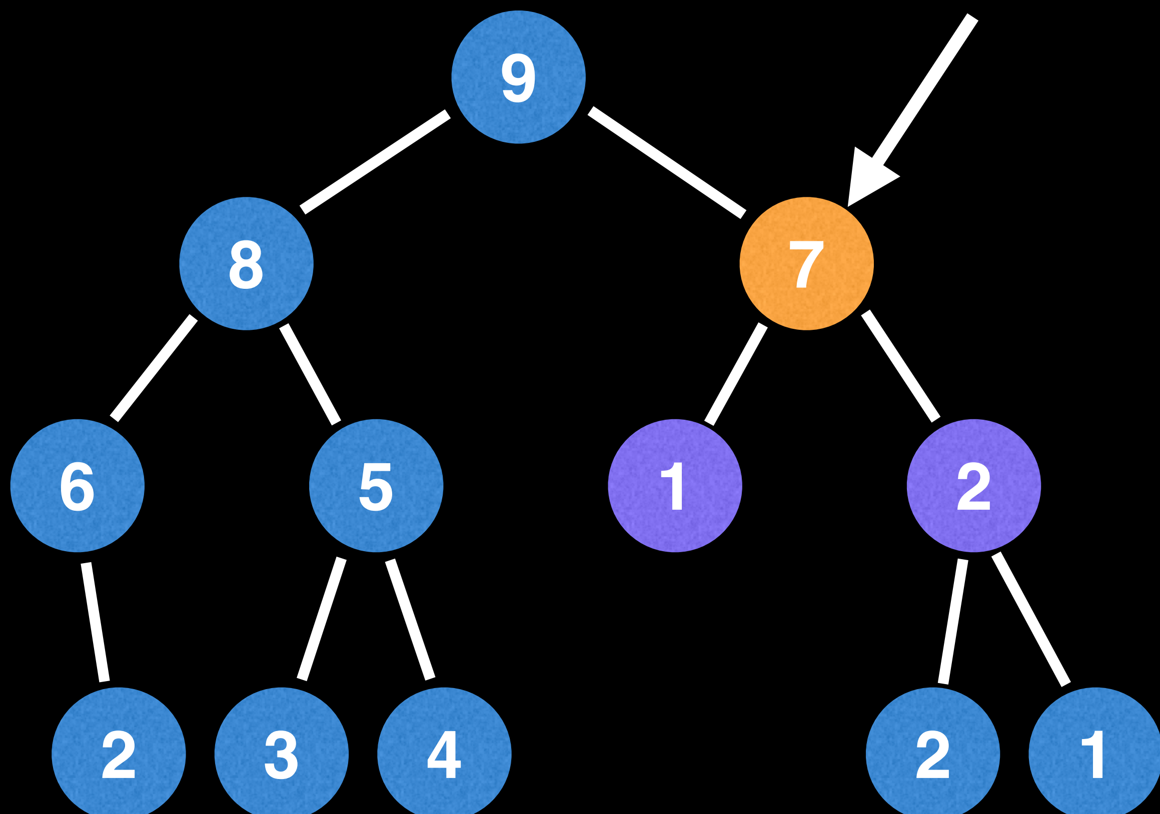
Reciprocally, the parent of node i is: floor((i-1)/2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

This trick also works for any n-ary tree

Storing rooted trees

The root node is always at index 0 and the children of the current node *i* are accessed relative to position *i*.



Let *i* be the index of the current node

left node: 2*i + 1
right node: 2*i + 2

Reciprocally, the parent of node i is: floor((i-1)/2)

i

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
9	8	7	6	5	1	2	∅	2	3	4	∅	∅	2	1

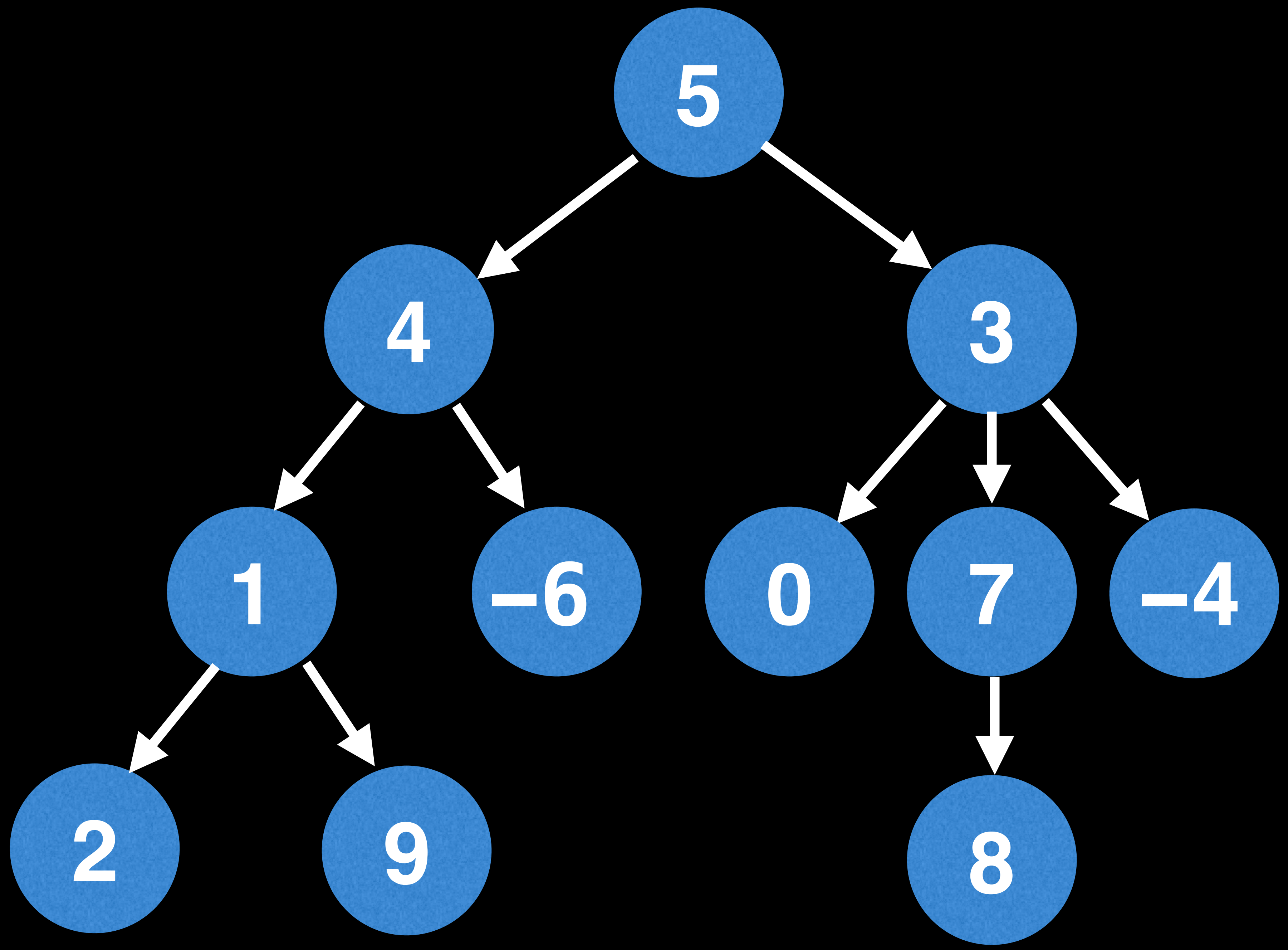
This trick also works for any n-ary tree

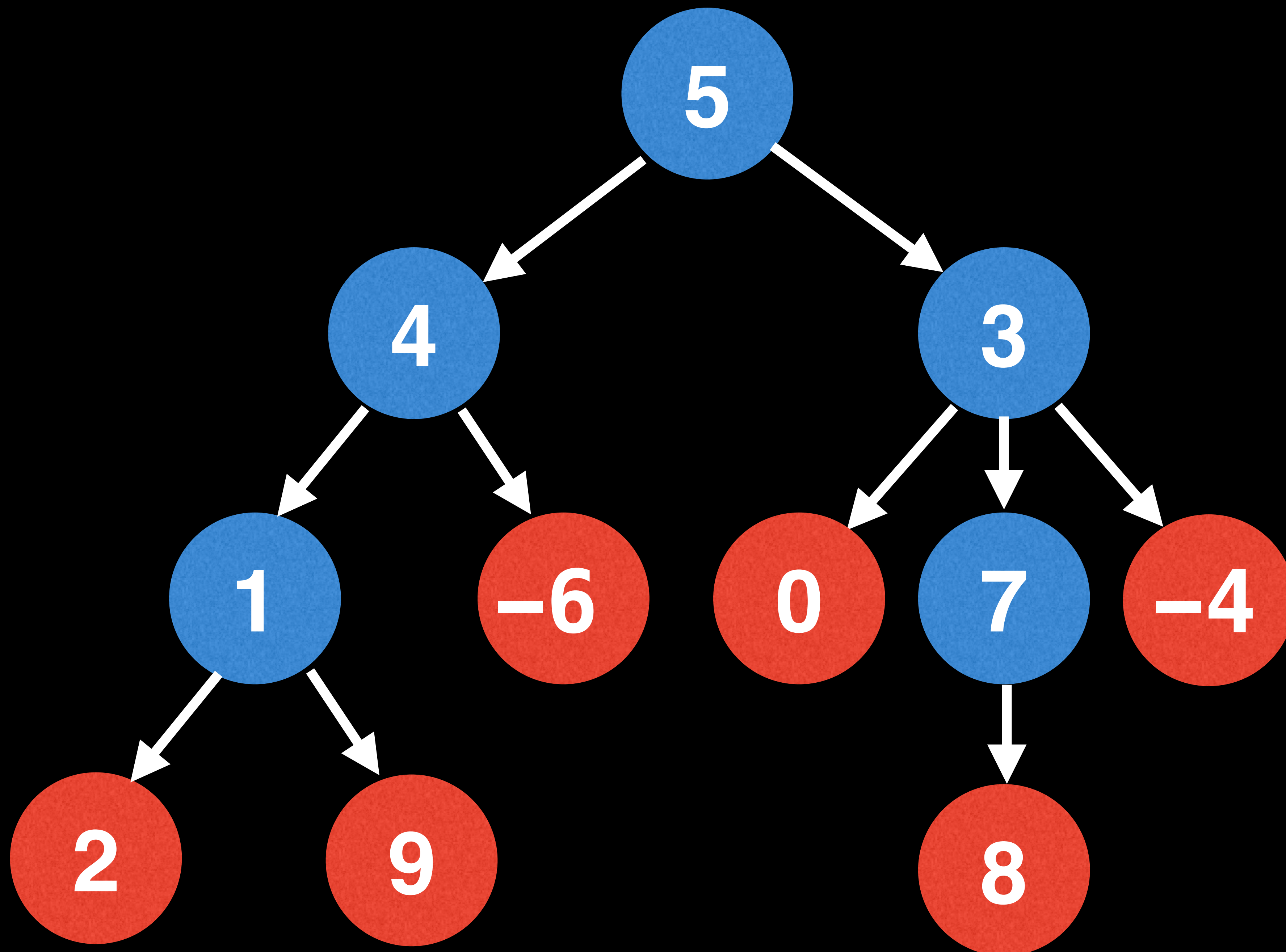
Beginner tree algorithms

William Fiset

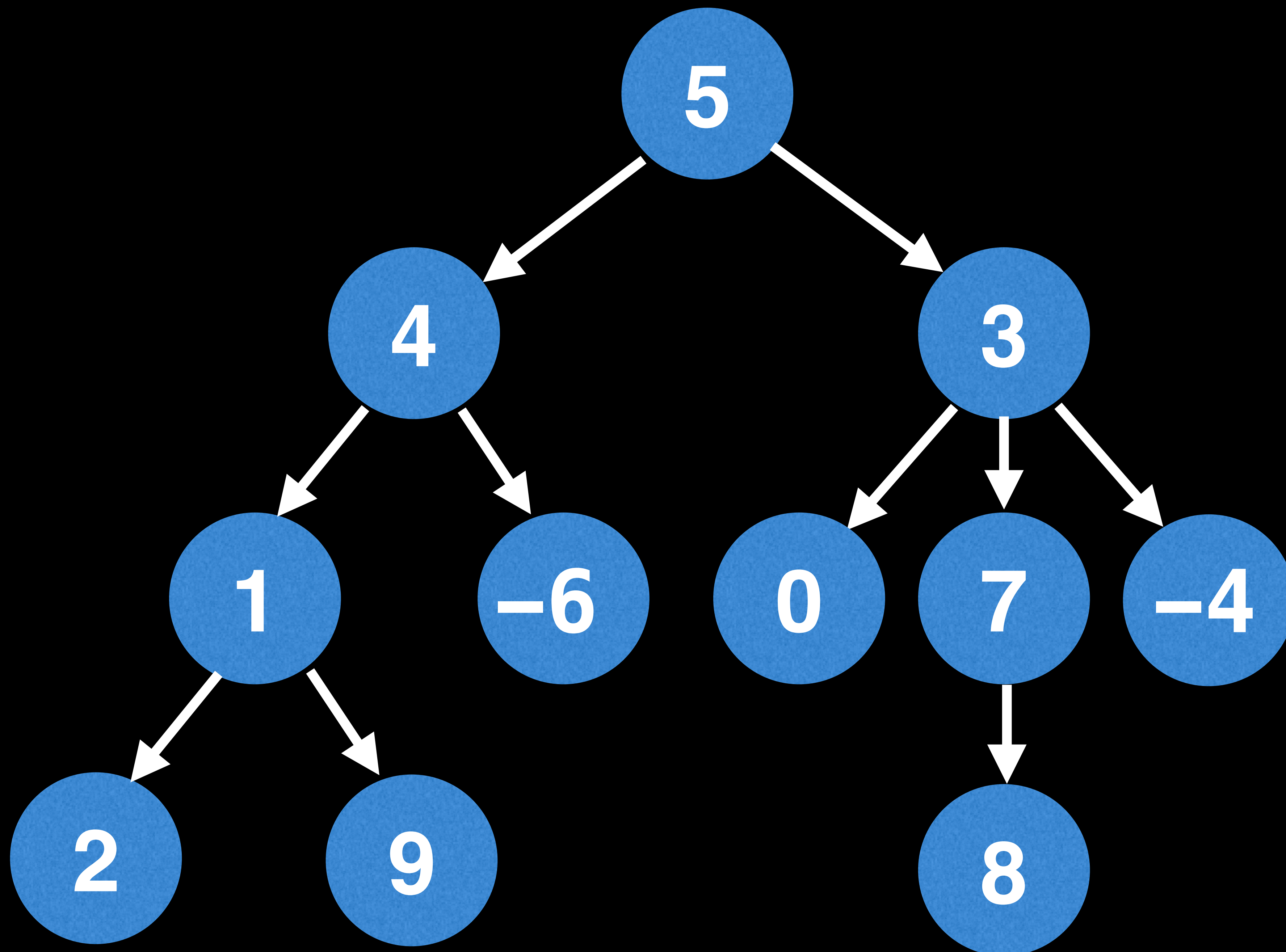
Problem 1: leaf node sum

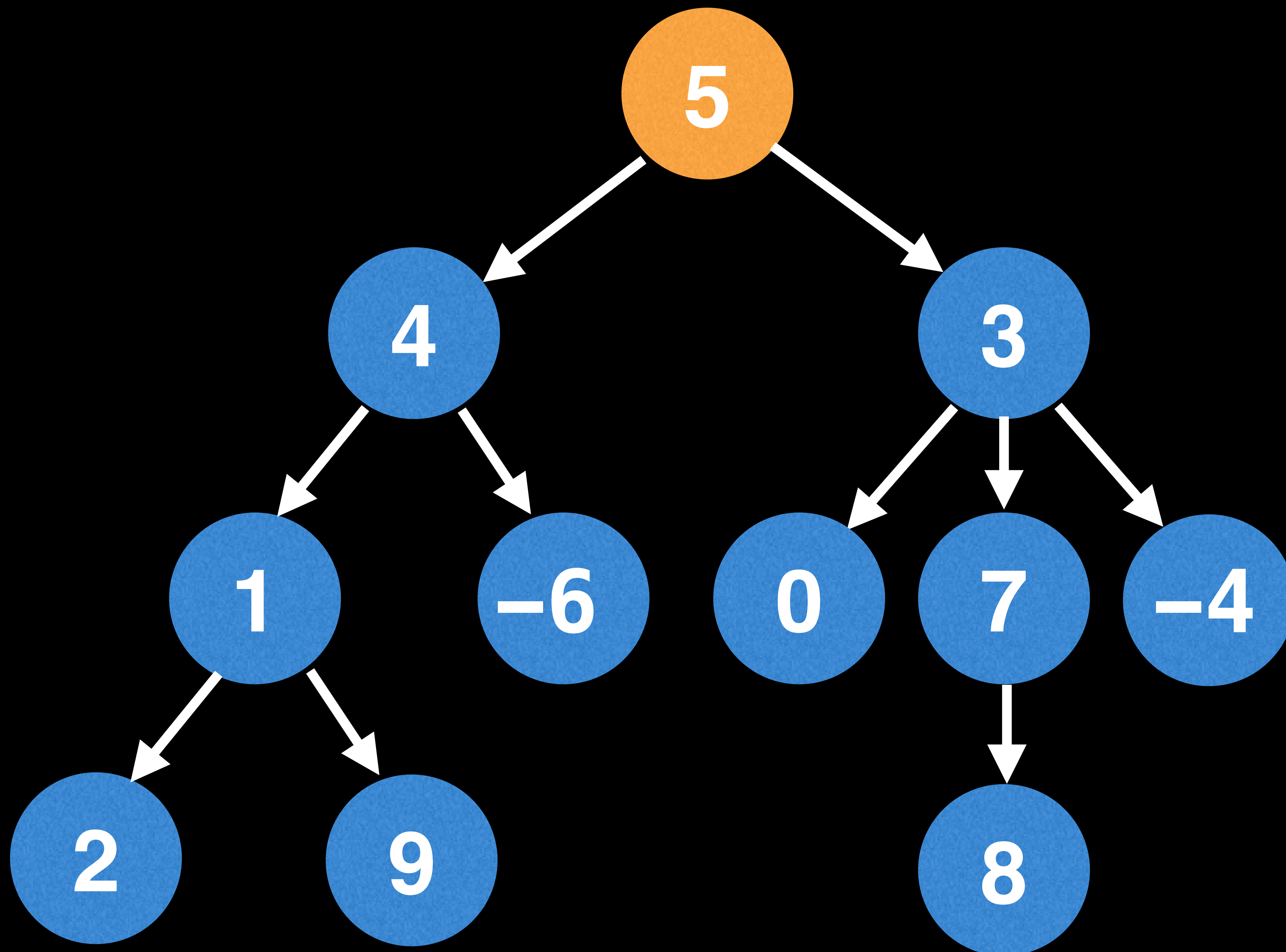
What is the sum of all the leaf node values in a tree?


5
4
3
1
−6
0
7
−4
2
9
8

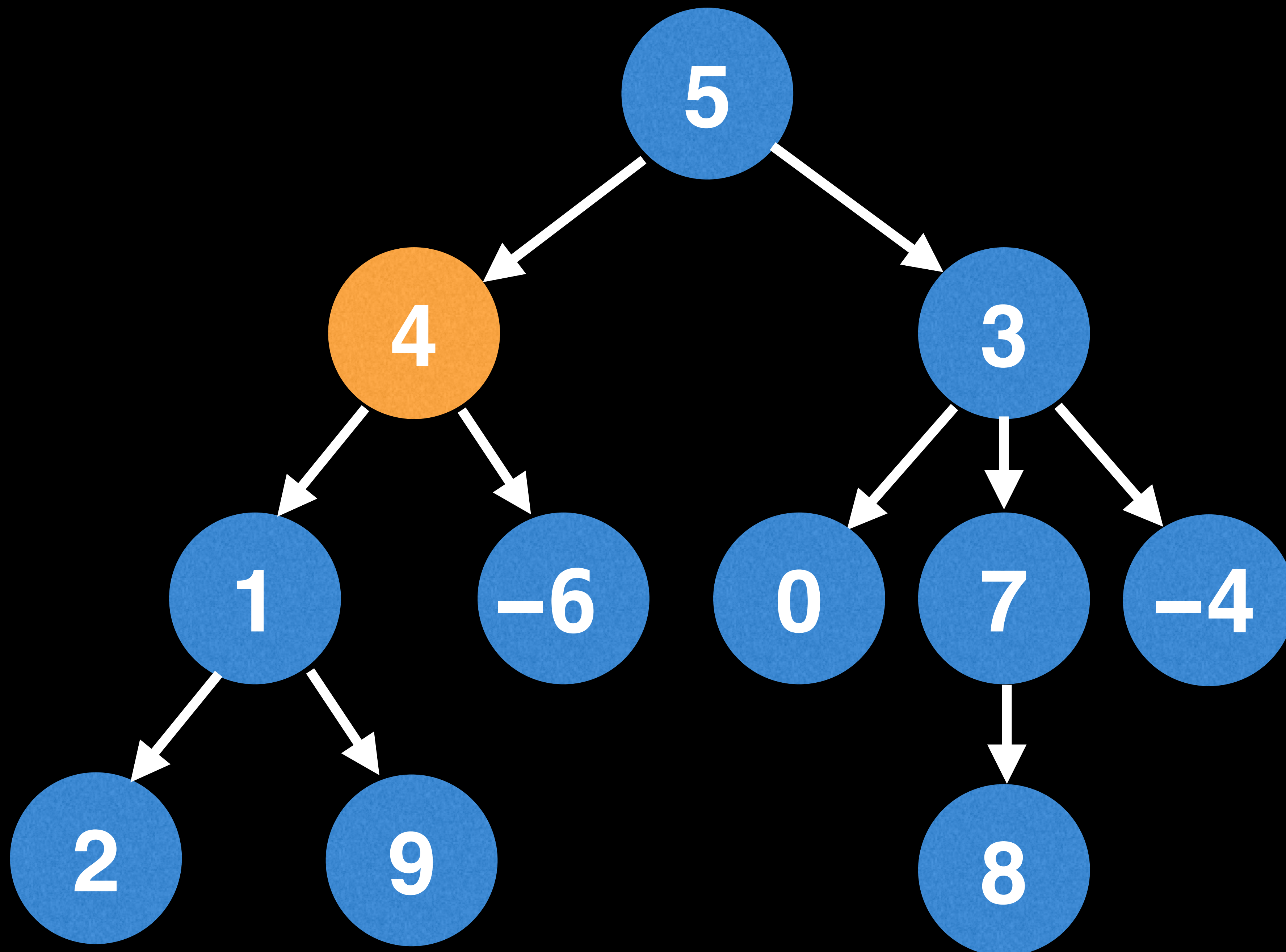


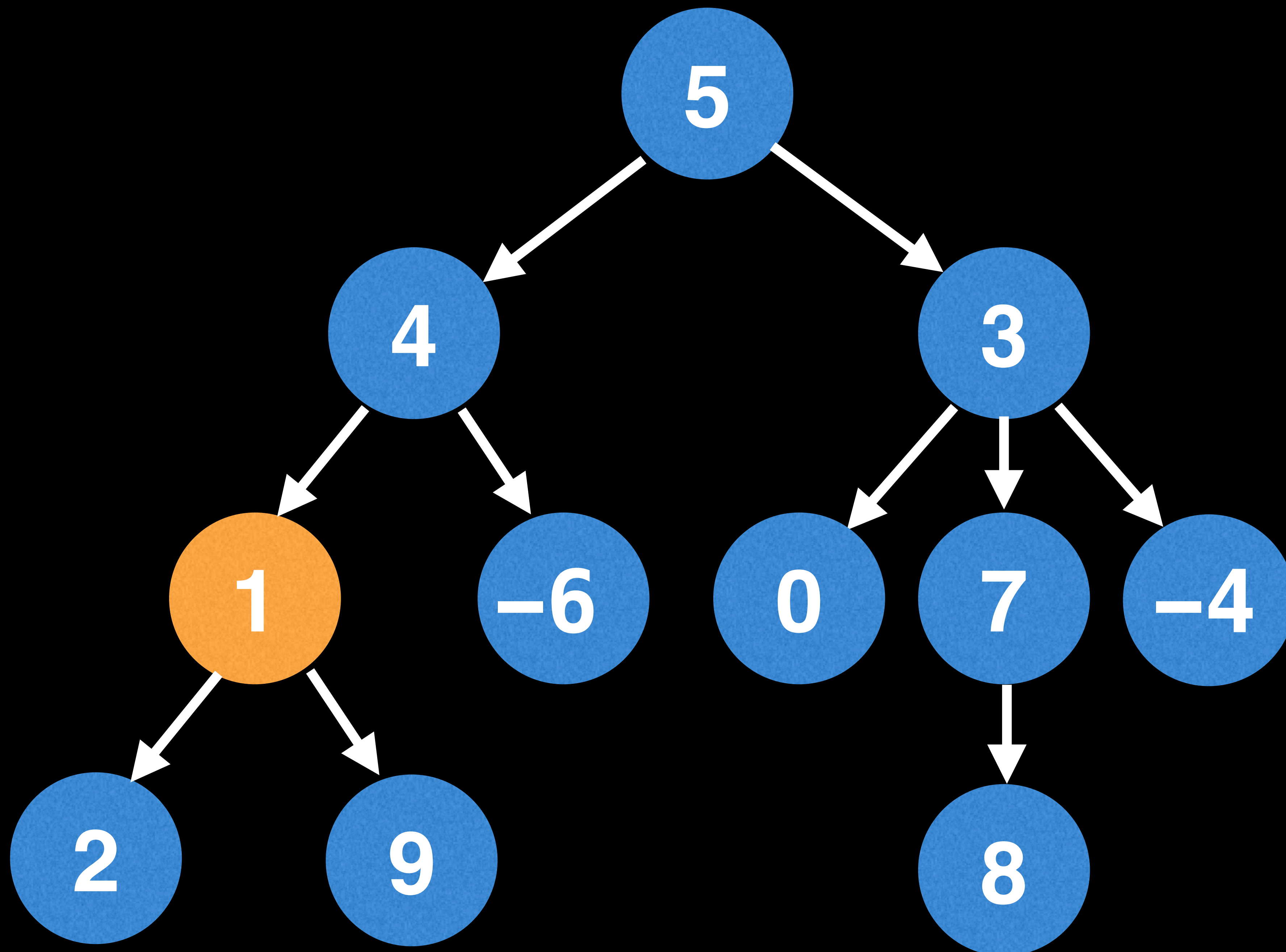
$$2 + 9 - 6 + 0 + 8 - 4 = 9$$

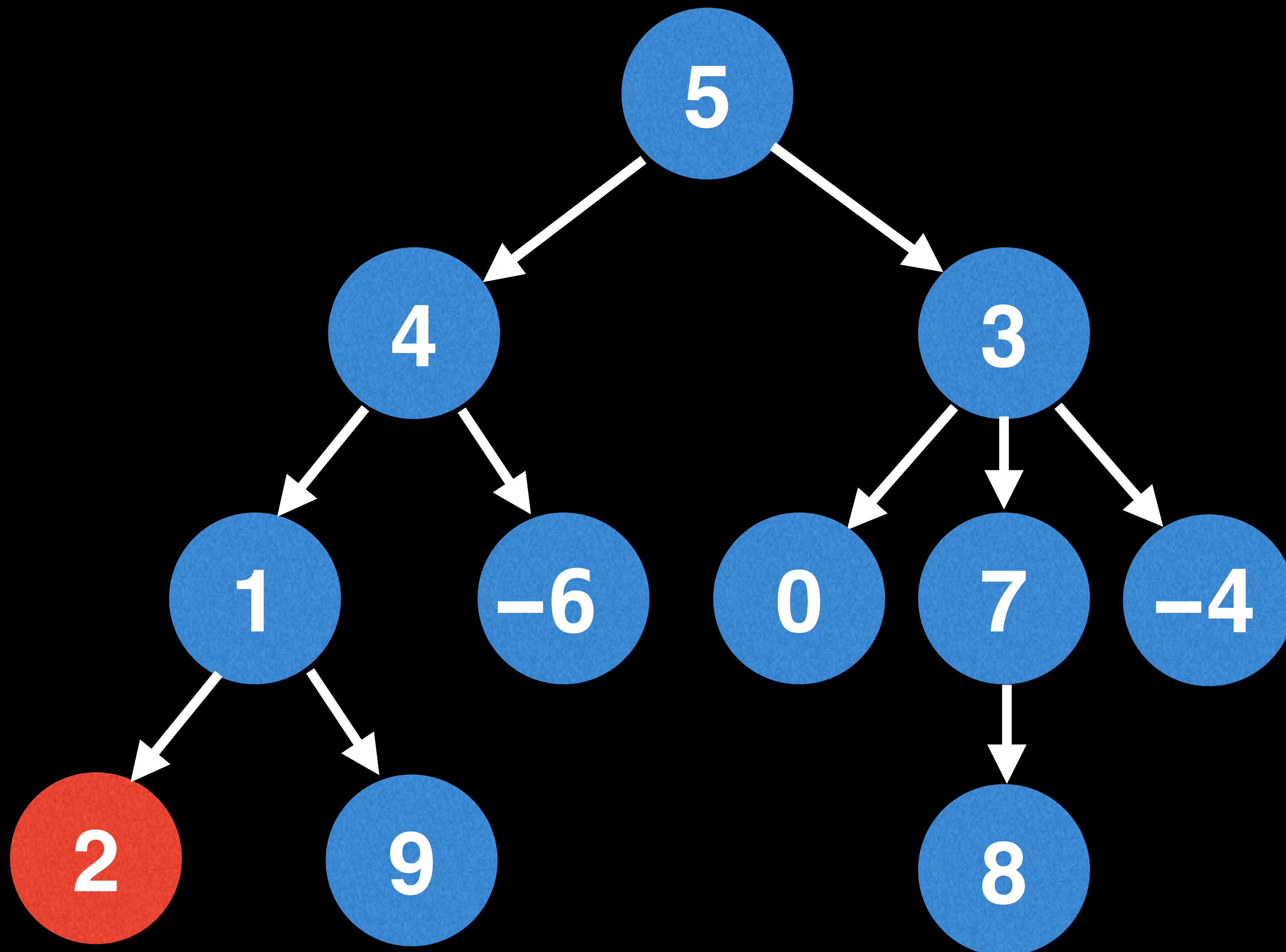

5
4
3
1
−6
0
7
−4
2
9
8

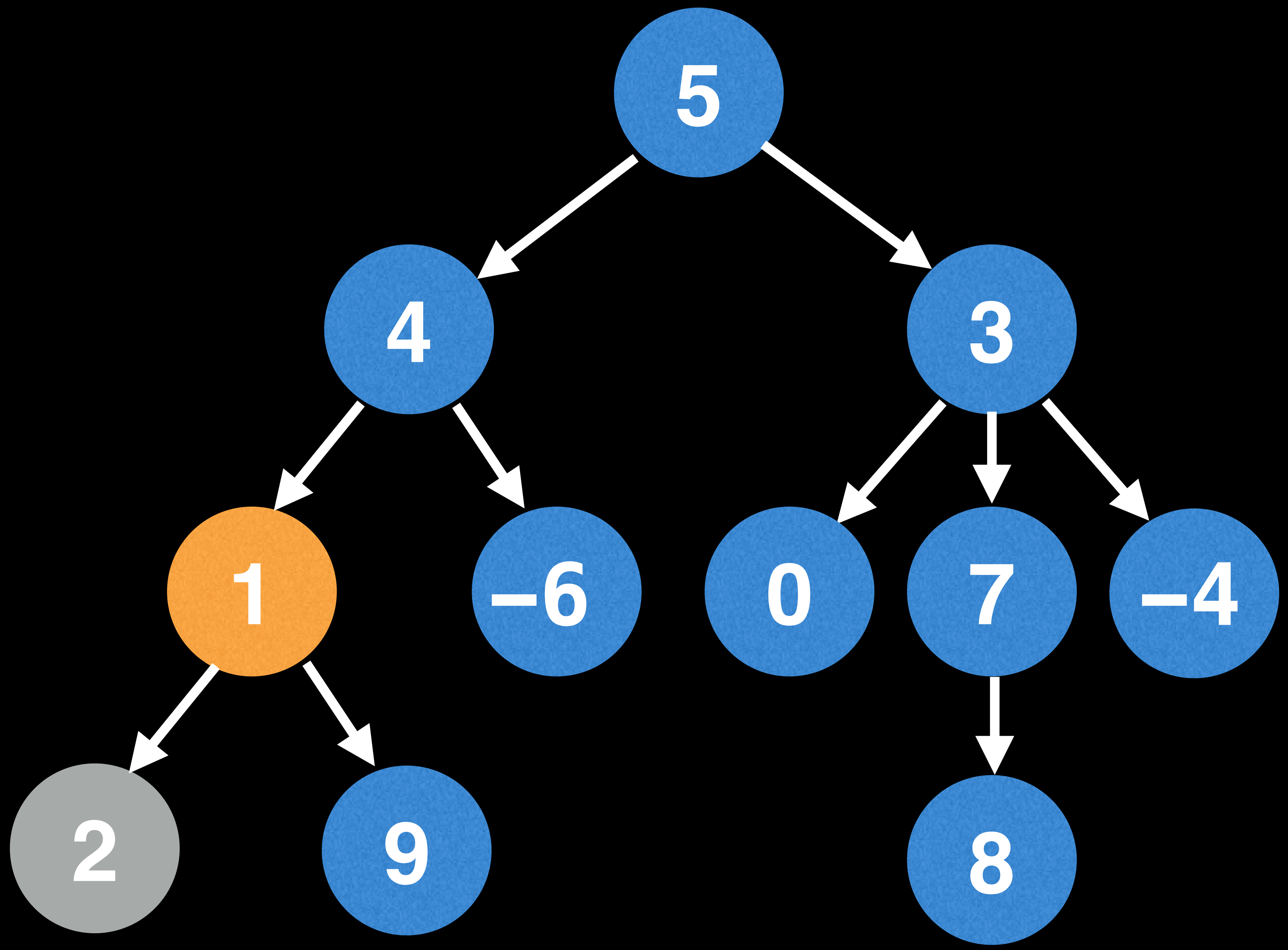


When dealing with rooted trees you begin with having a reference to the root node as a starting point for most algorithms.

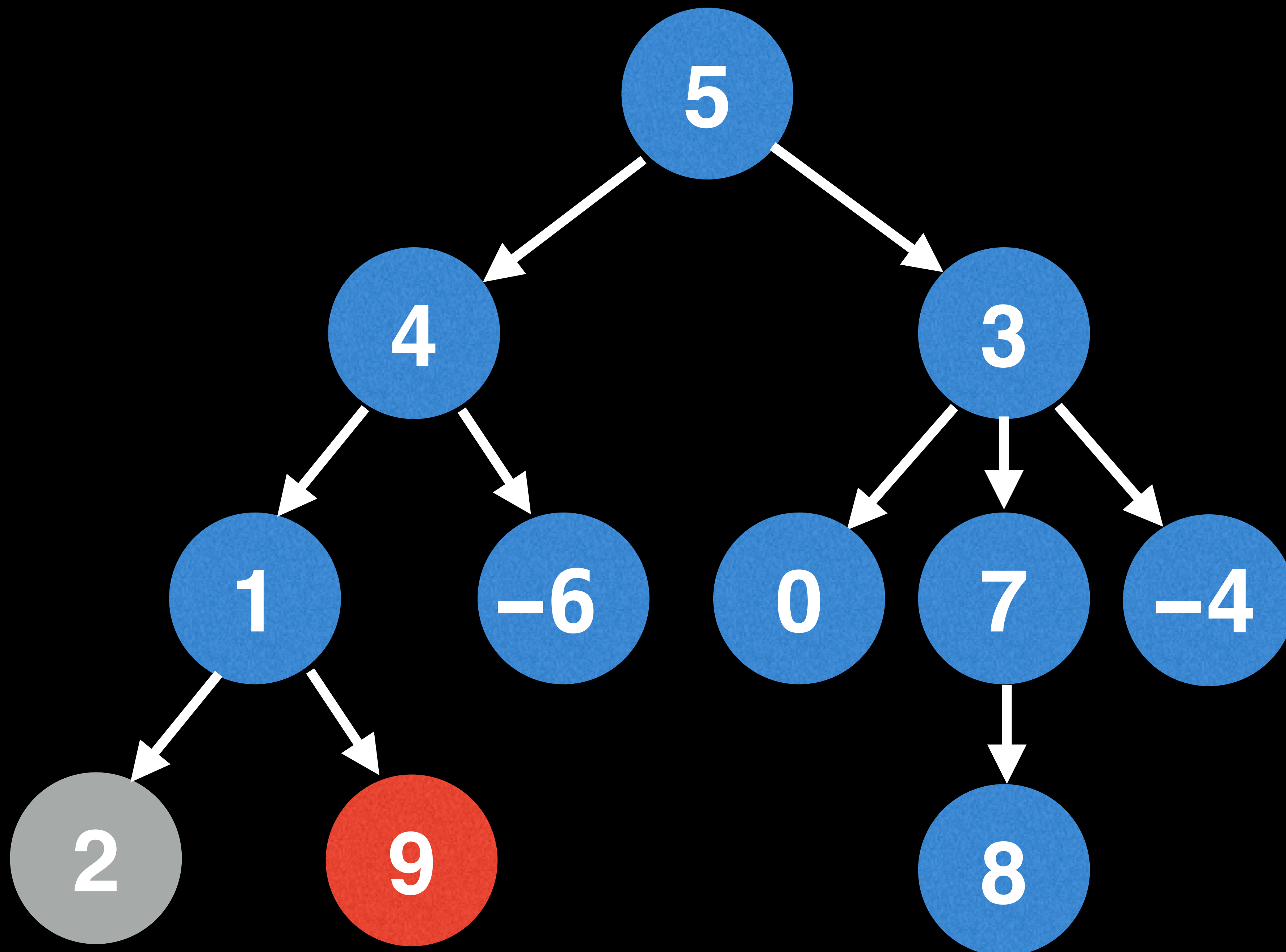

5
4
3
1
-6
0
7
-4
2
9
8


5
4
3
1
-6
0
7
-4
2
9
8

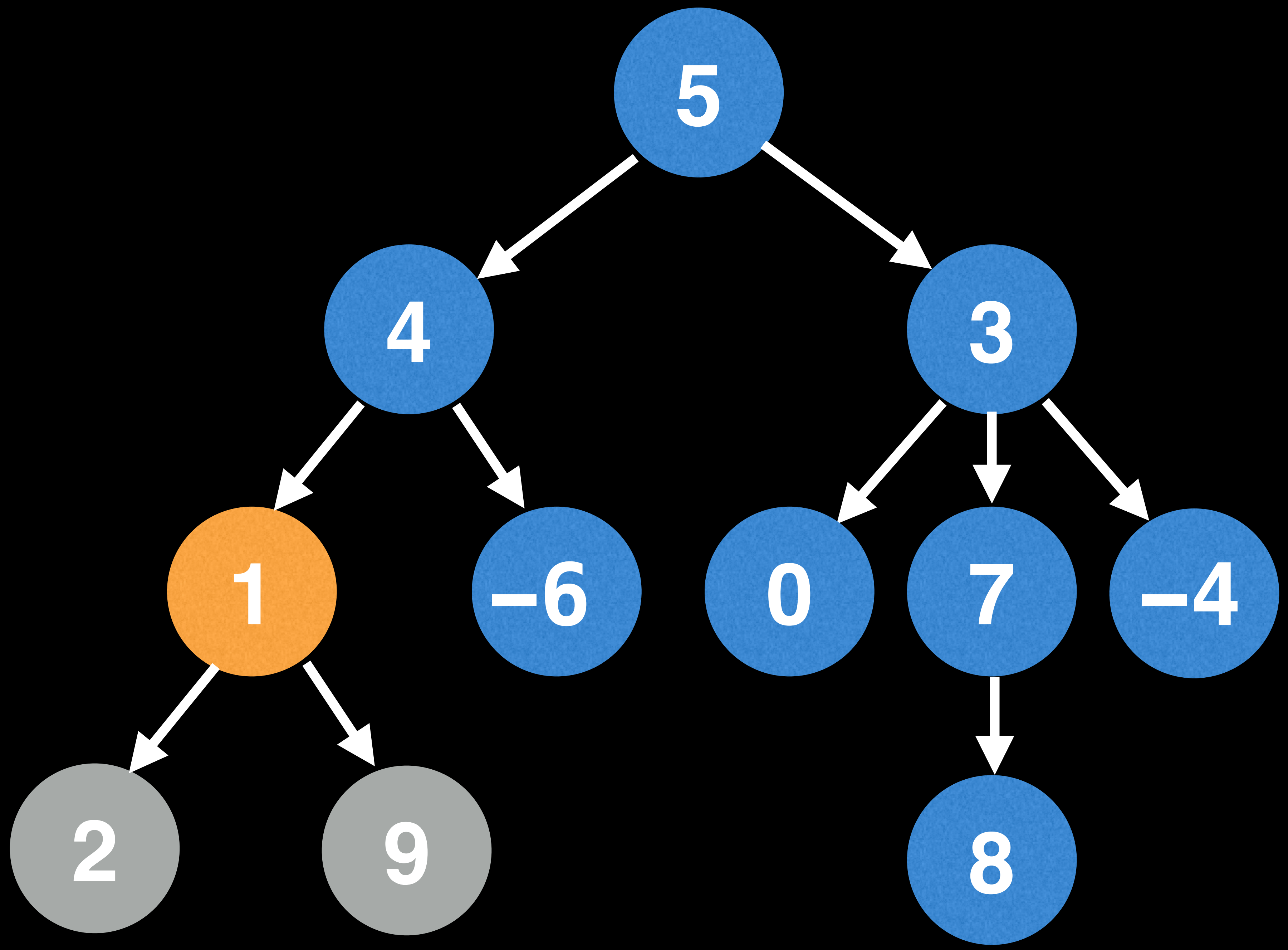

5
4
3
1
−6
0
7
−4
2
9
8



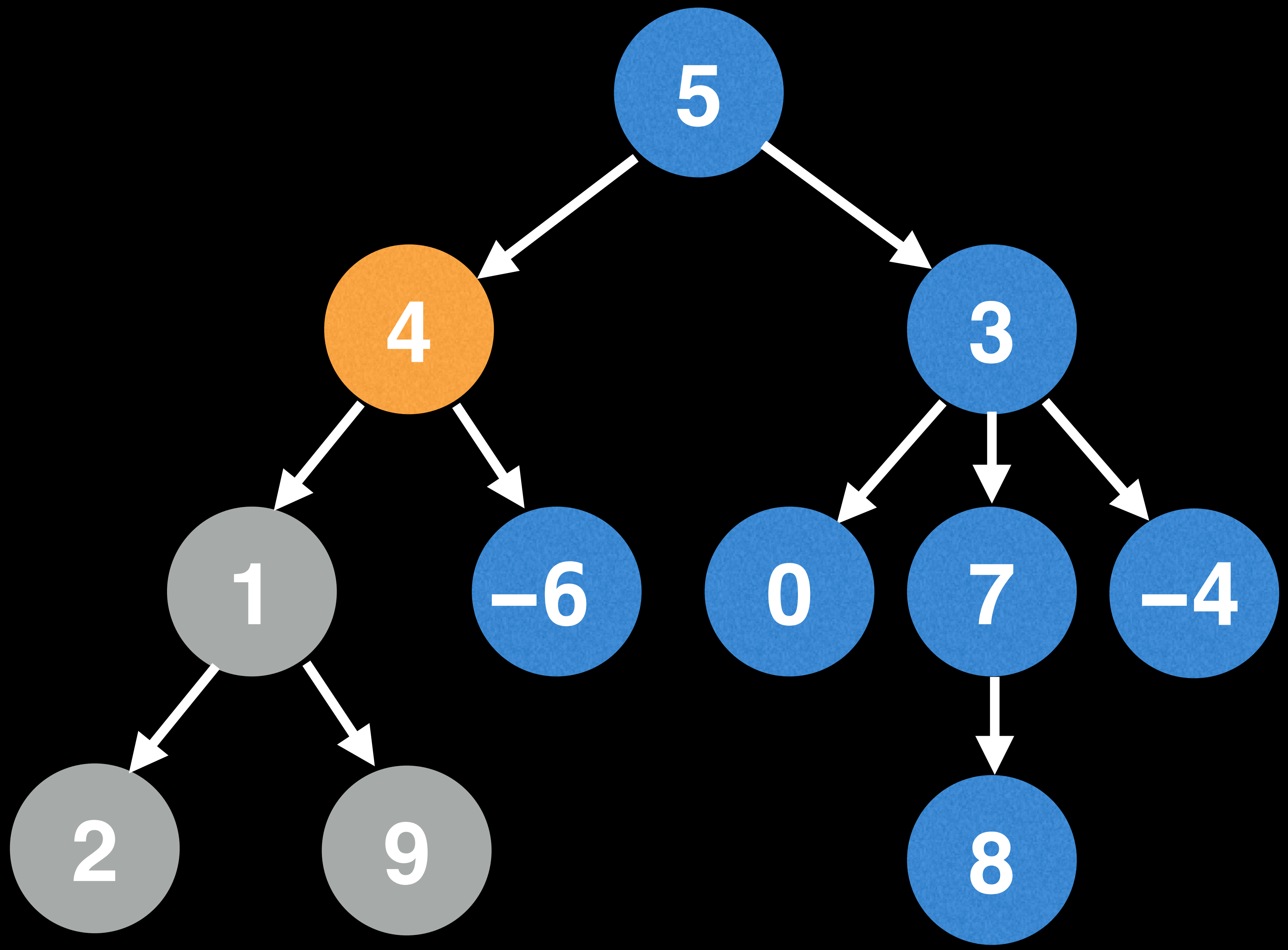
2



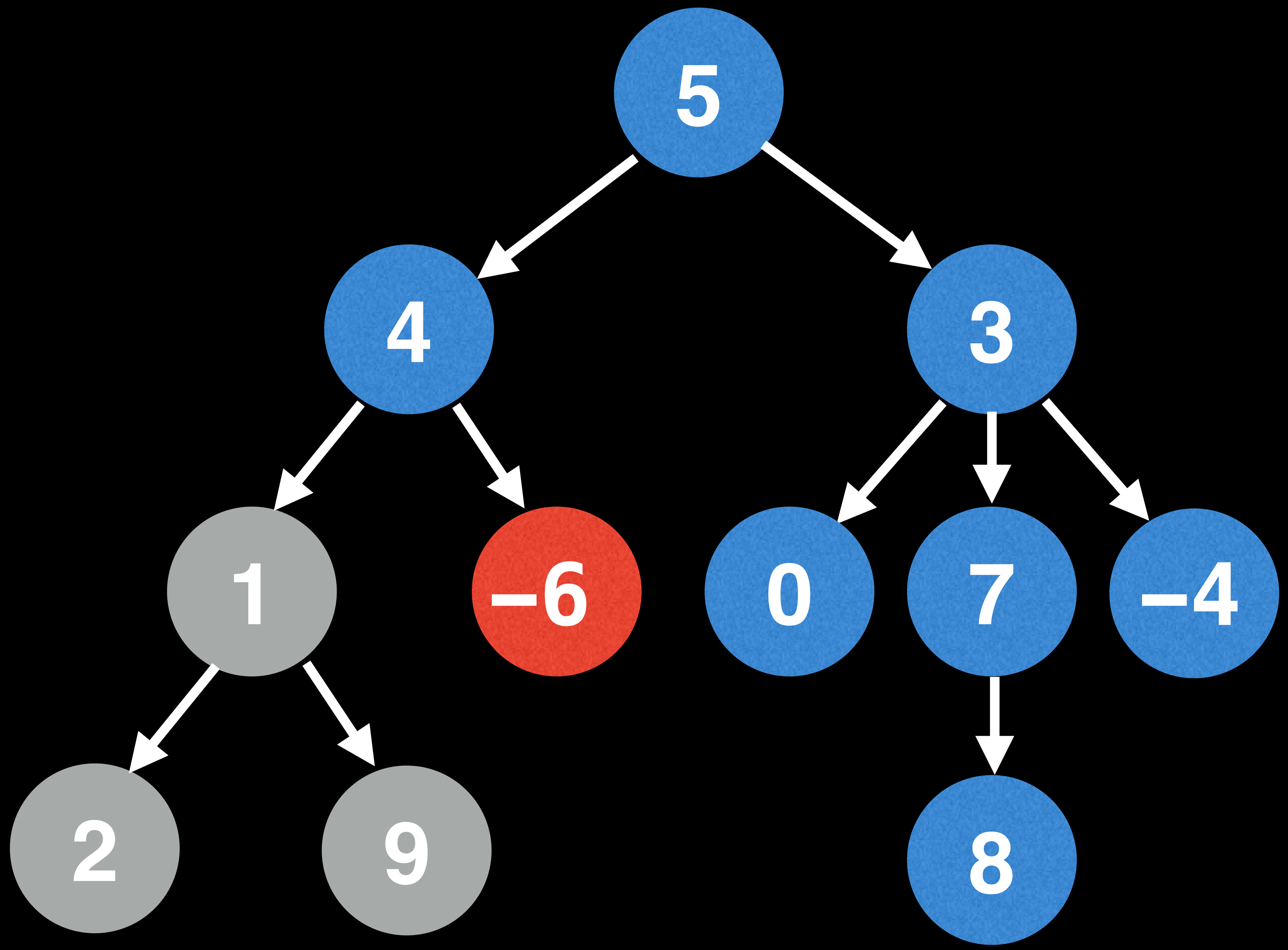
2



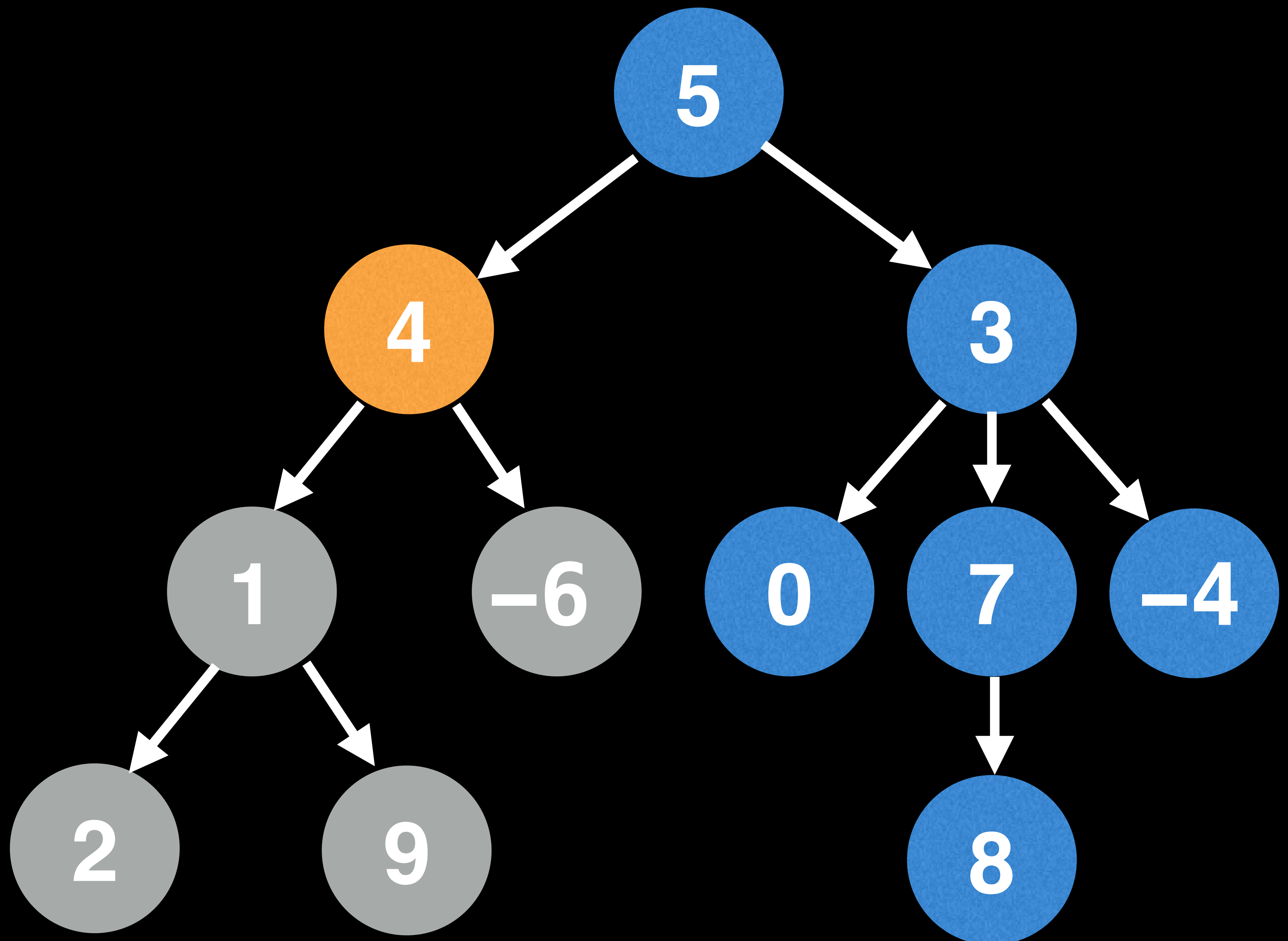
2 + 9



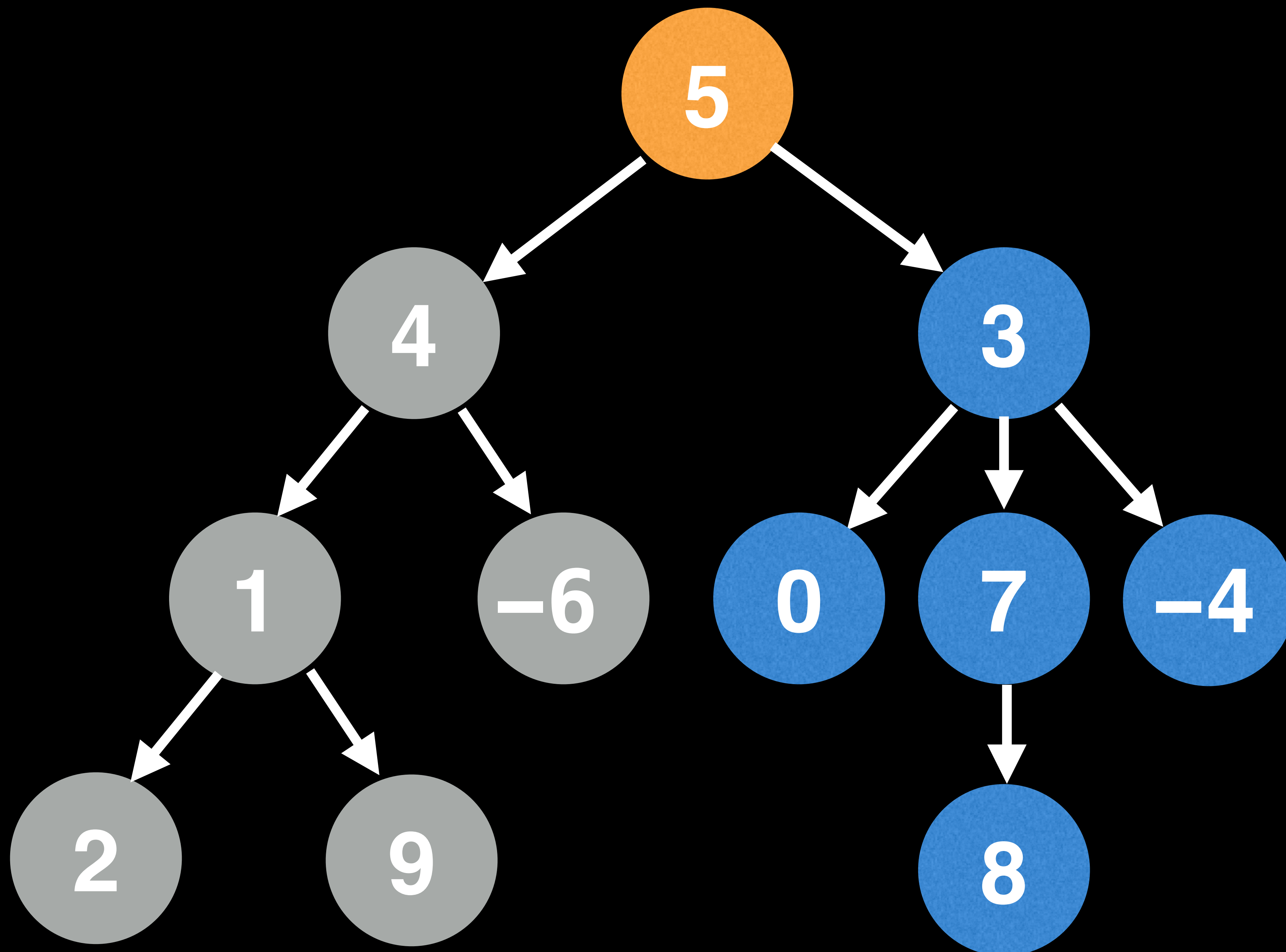
2 + 9



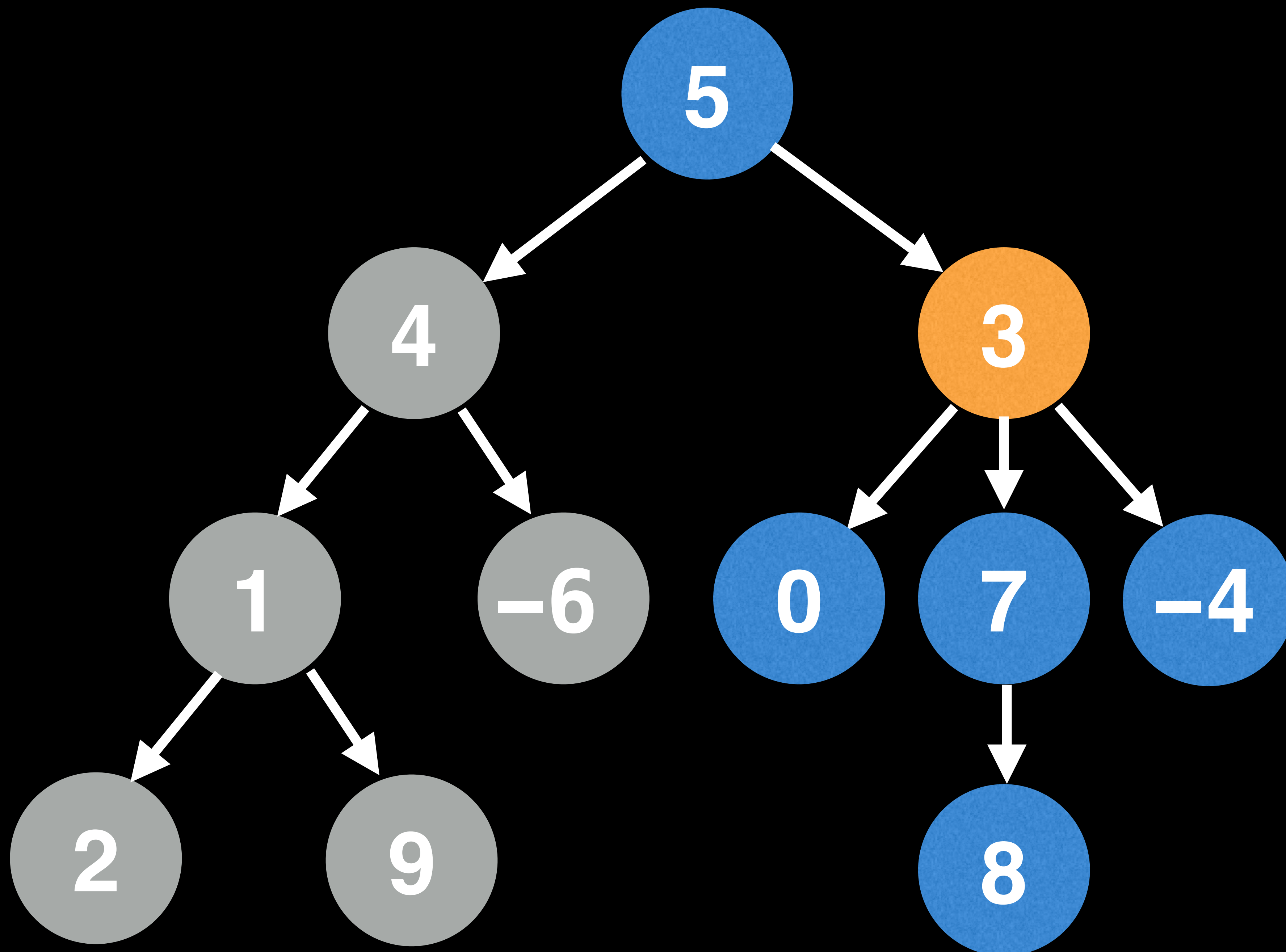
2 + 9



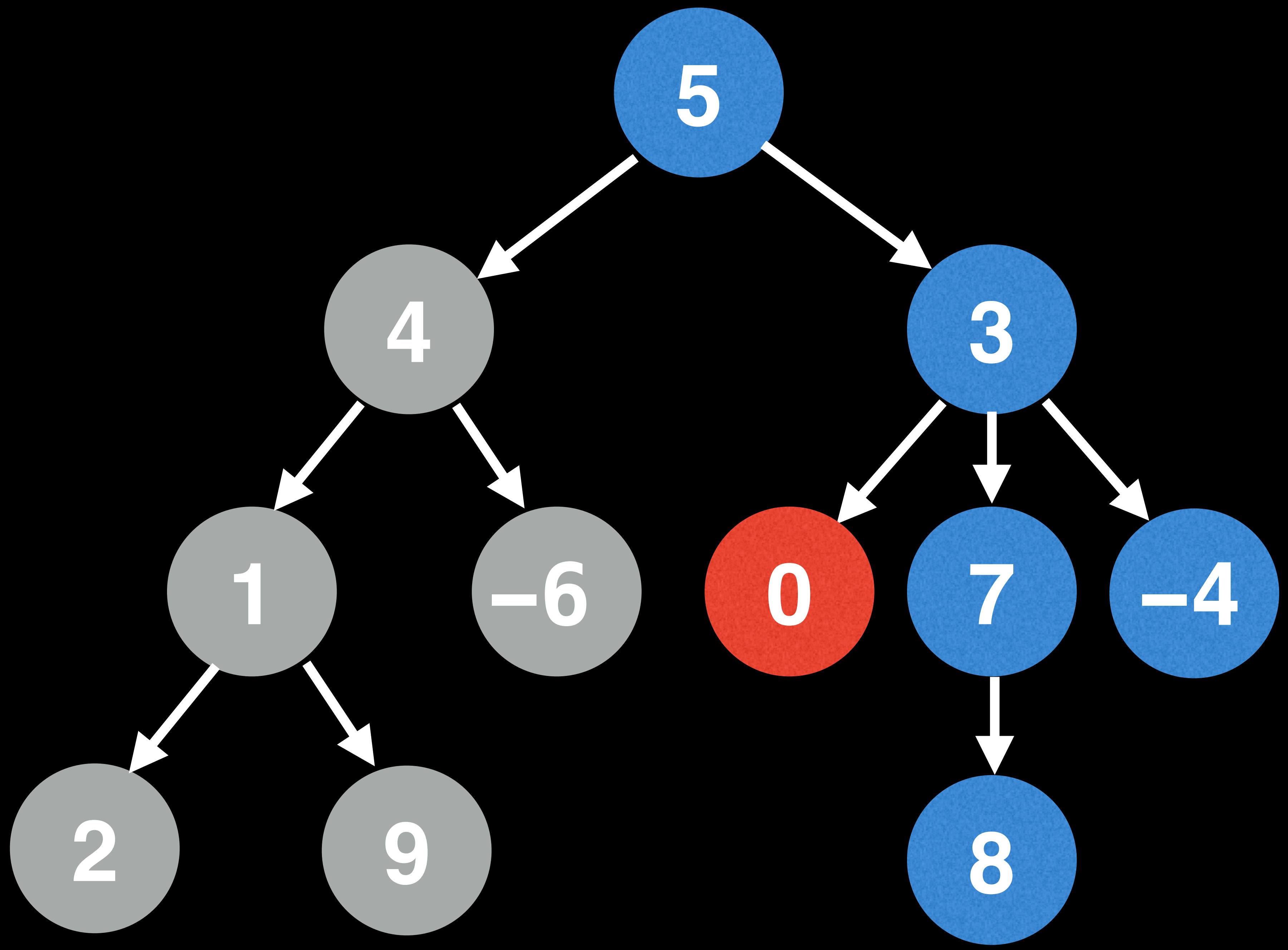
2 + 9 − 6



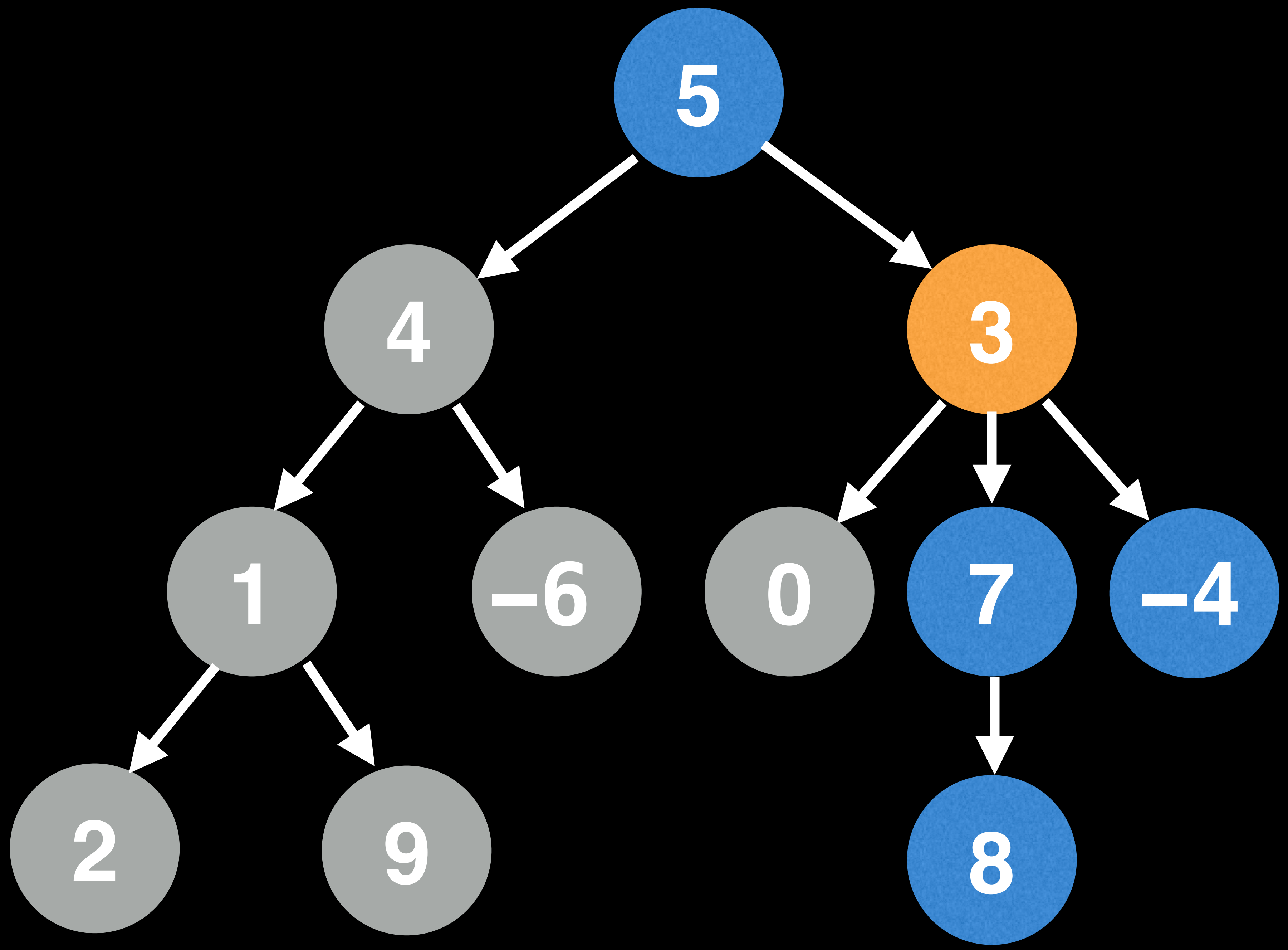
2 + 9 − 6



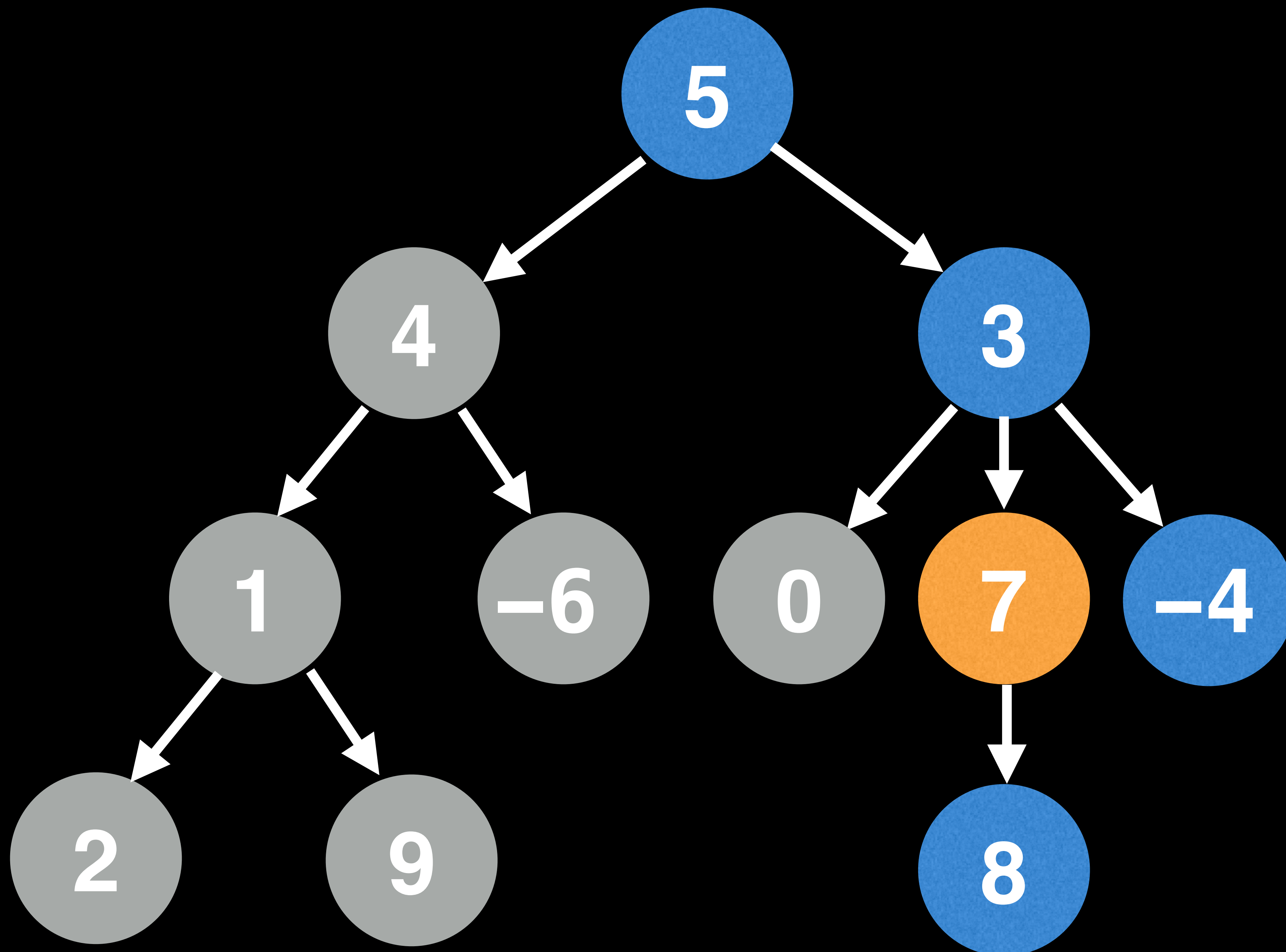
$2 + 9 - 6$



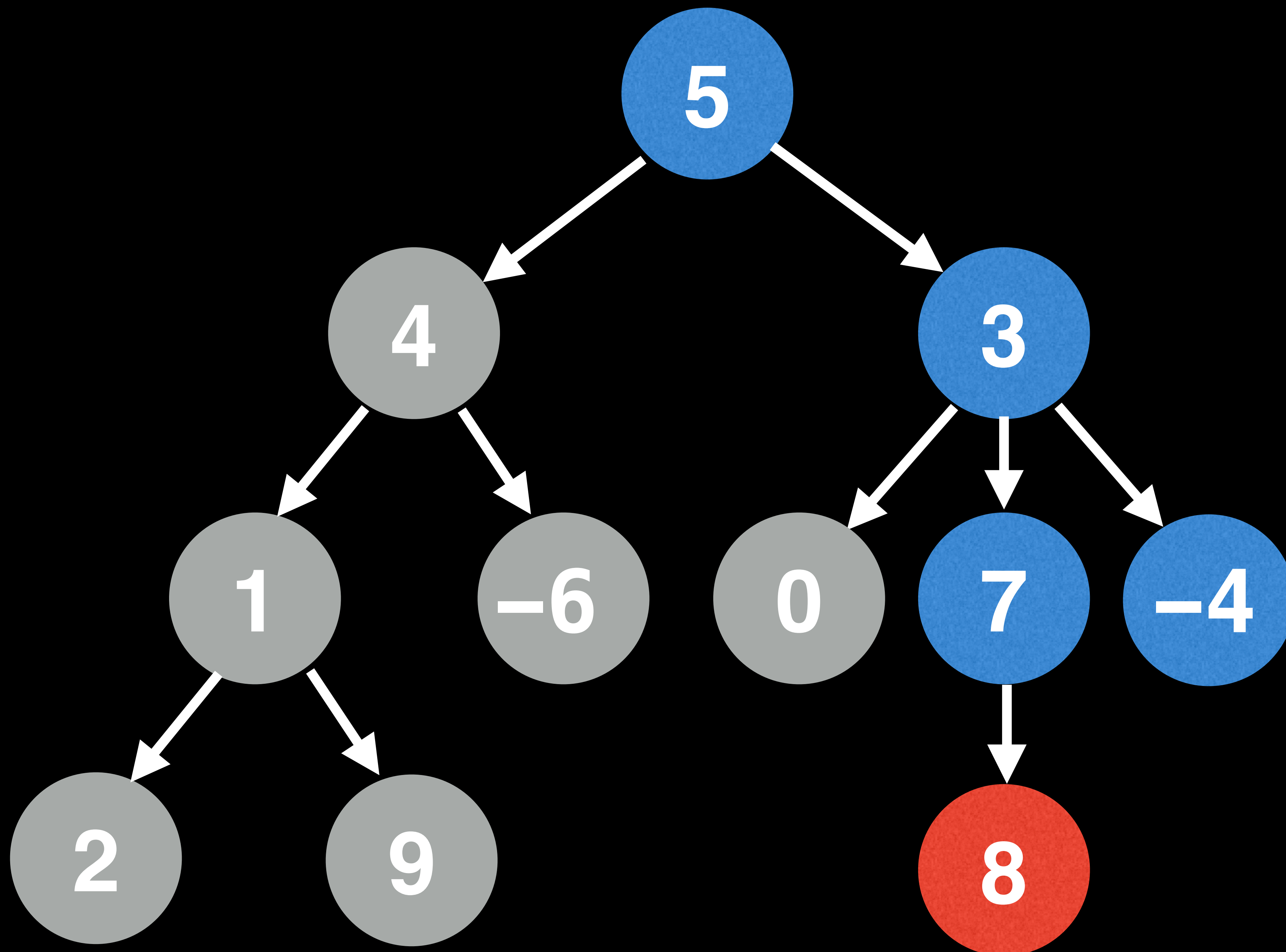
$2 + 9 - 6$



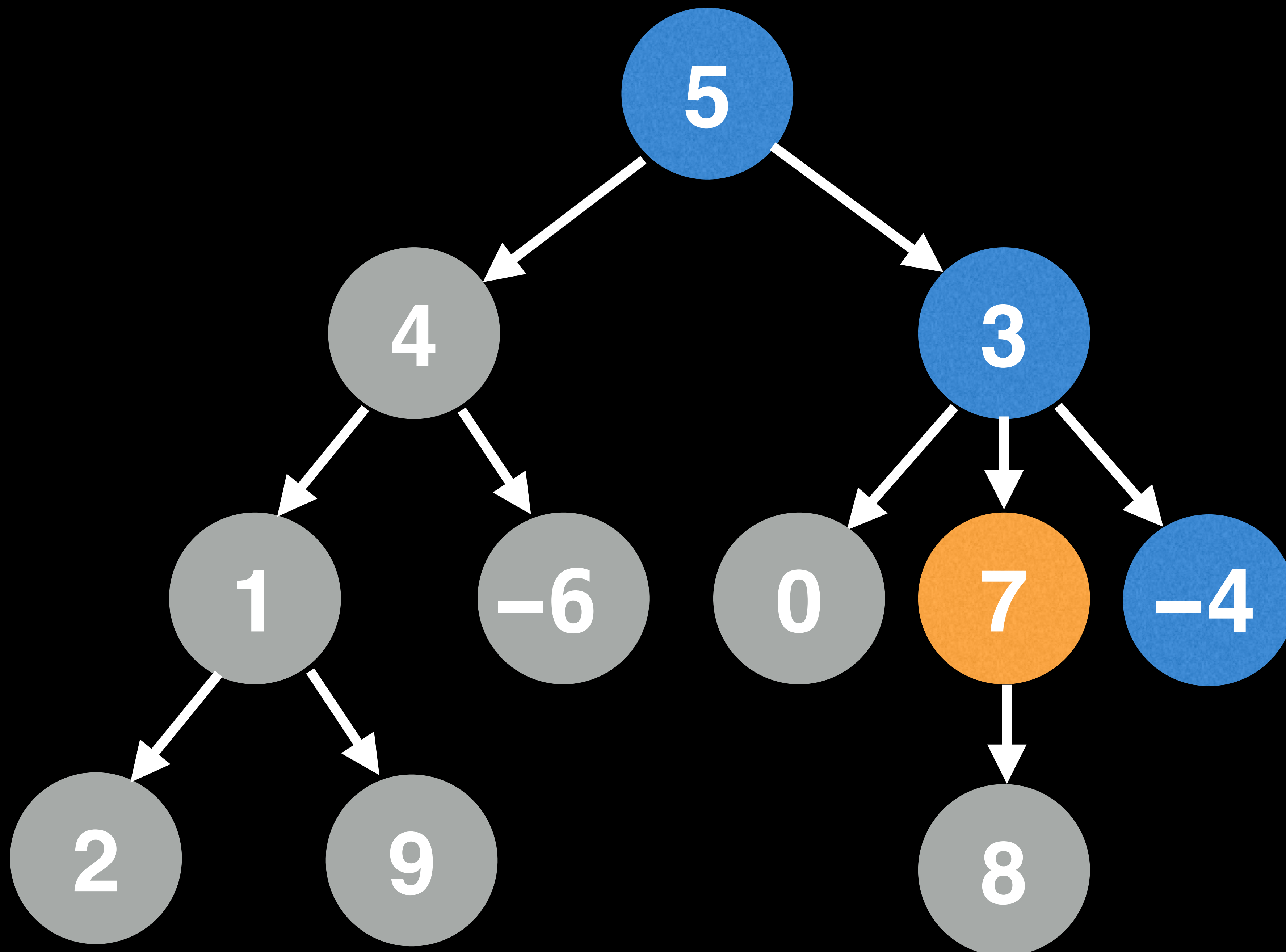
2 + 9 − 6 + 0



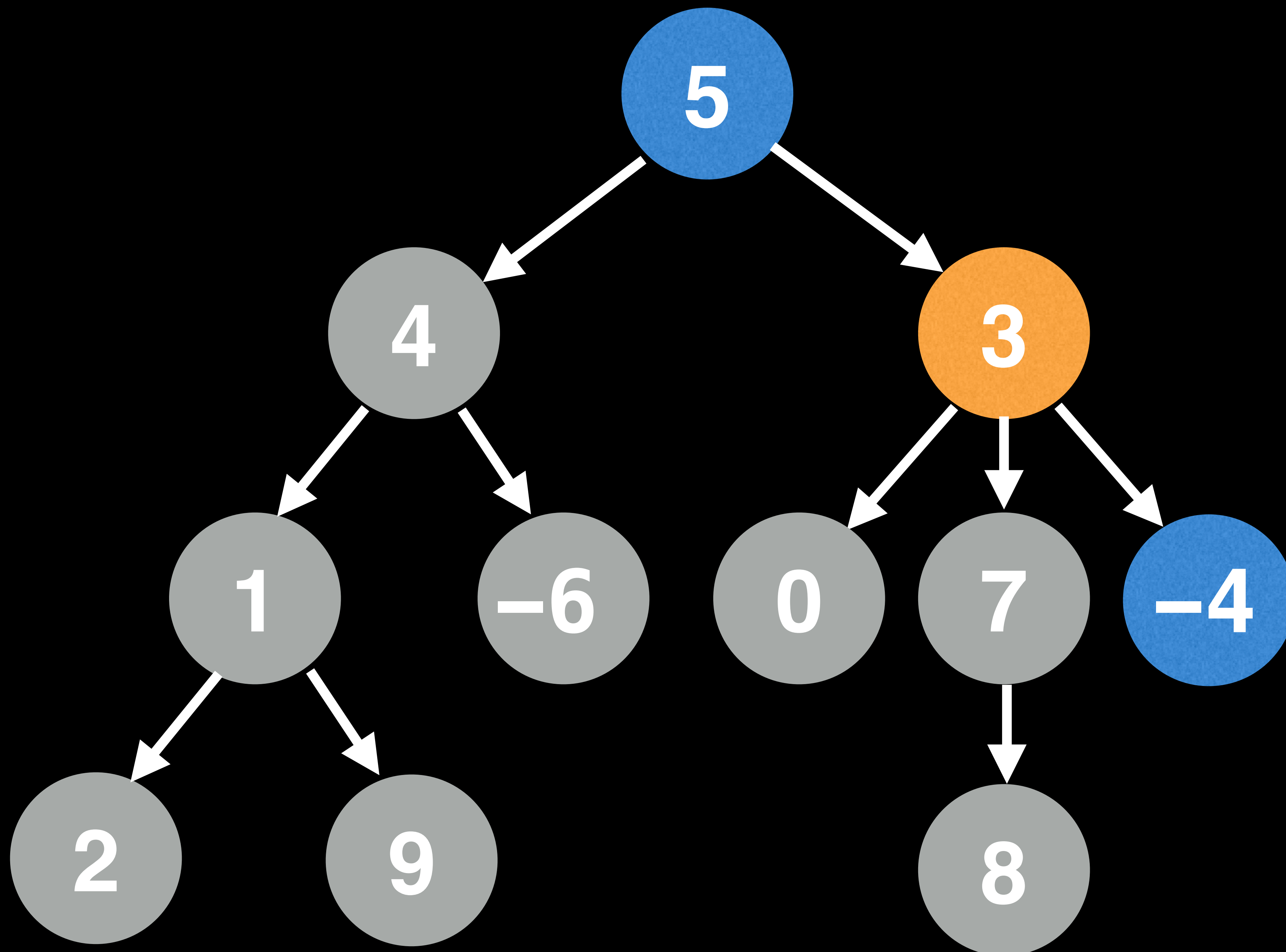
2 + 9 − 6 + 0



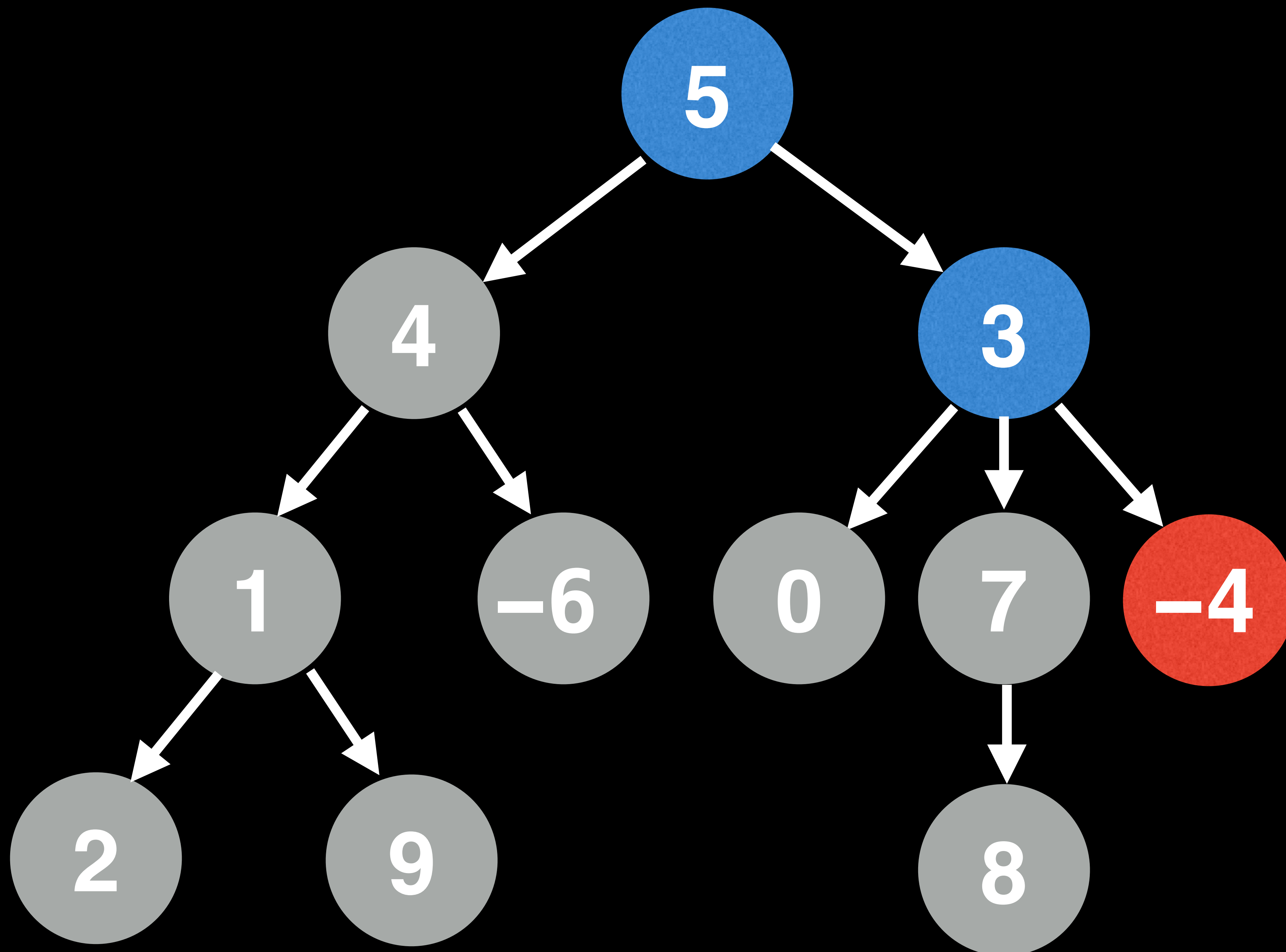
2 + 9 − 6 + 0



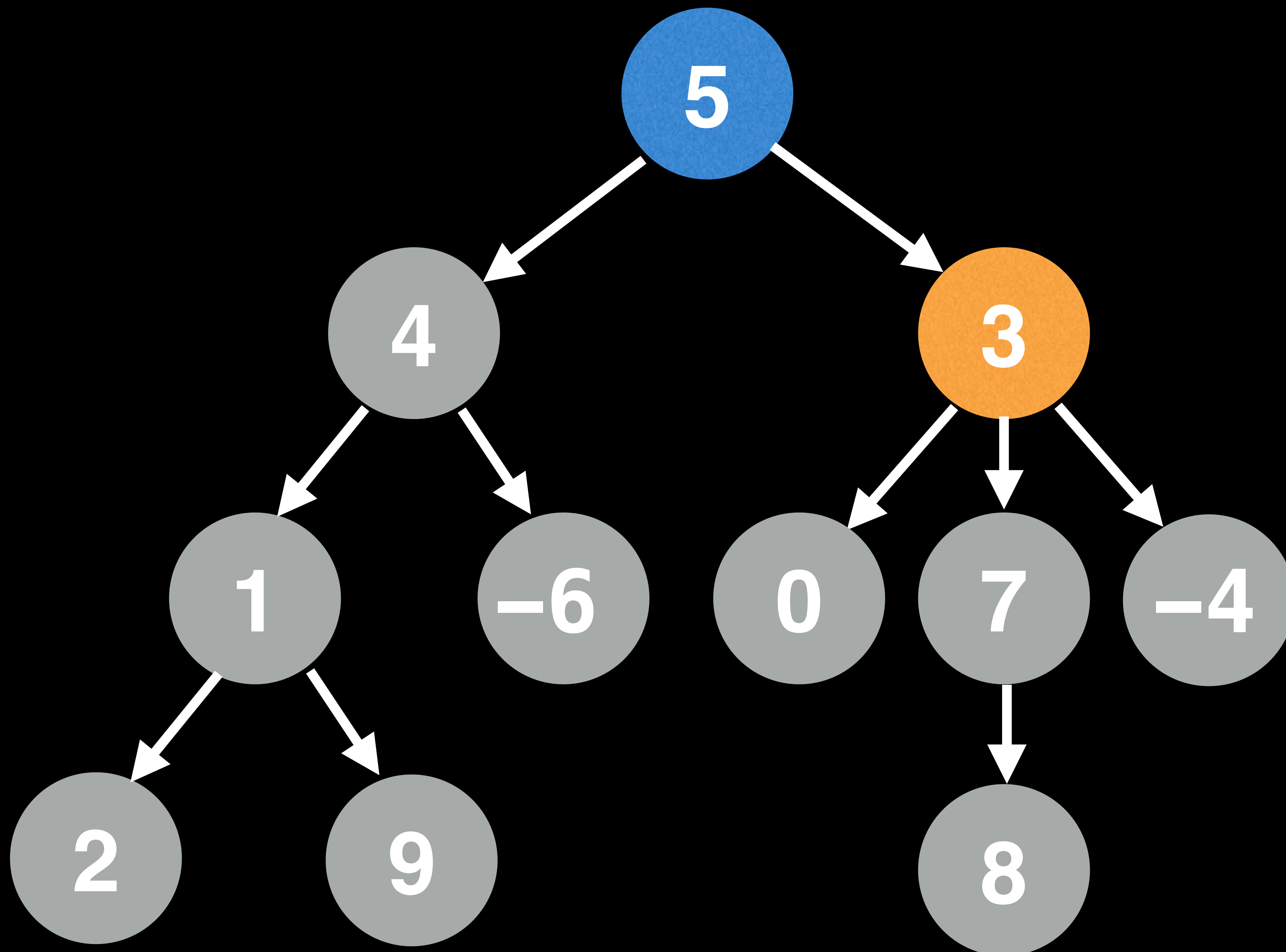
2 + 9 − 6 + 0 + 8



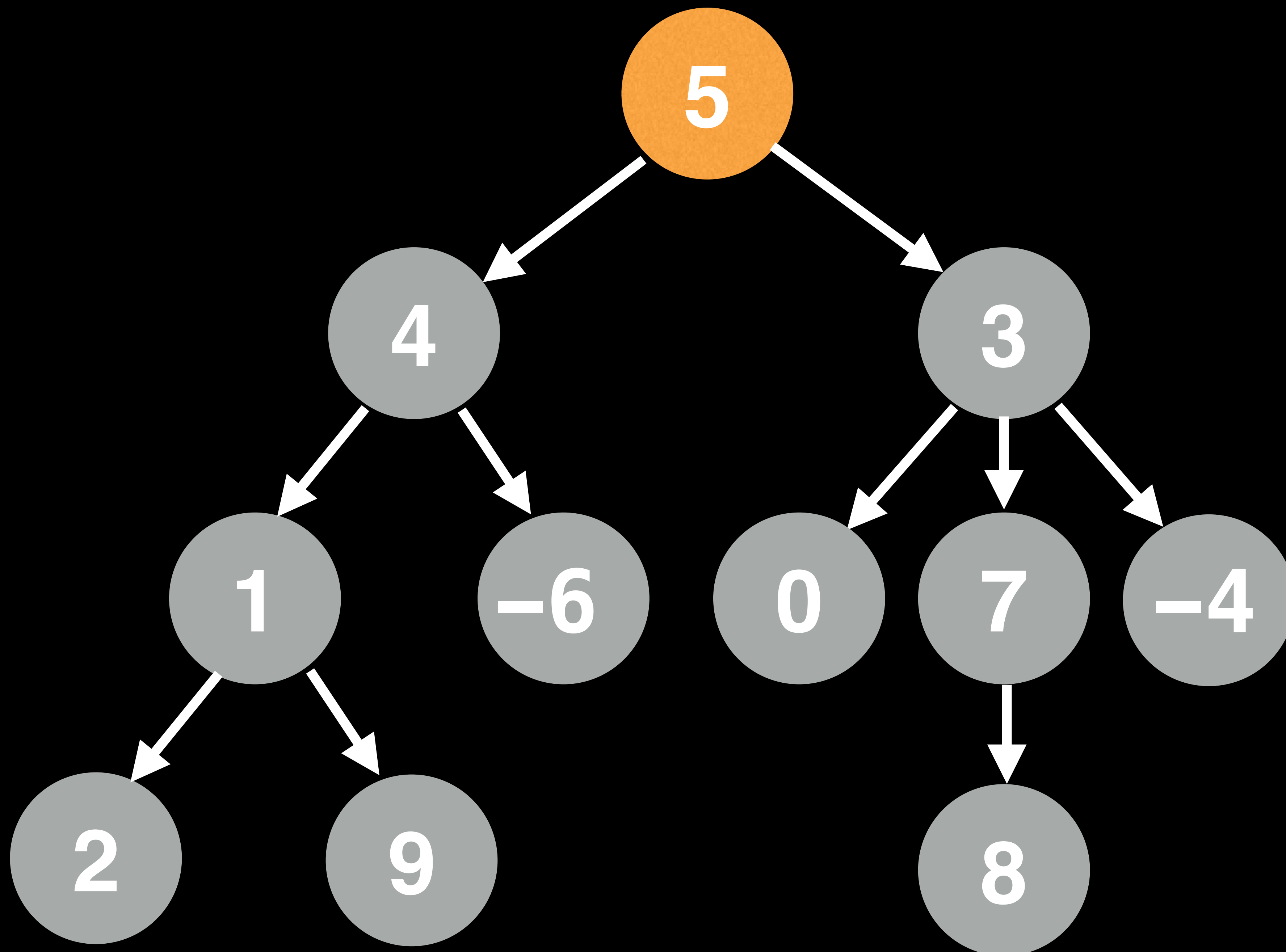
2 + 9 − 6 + 0 + 8



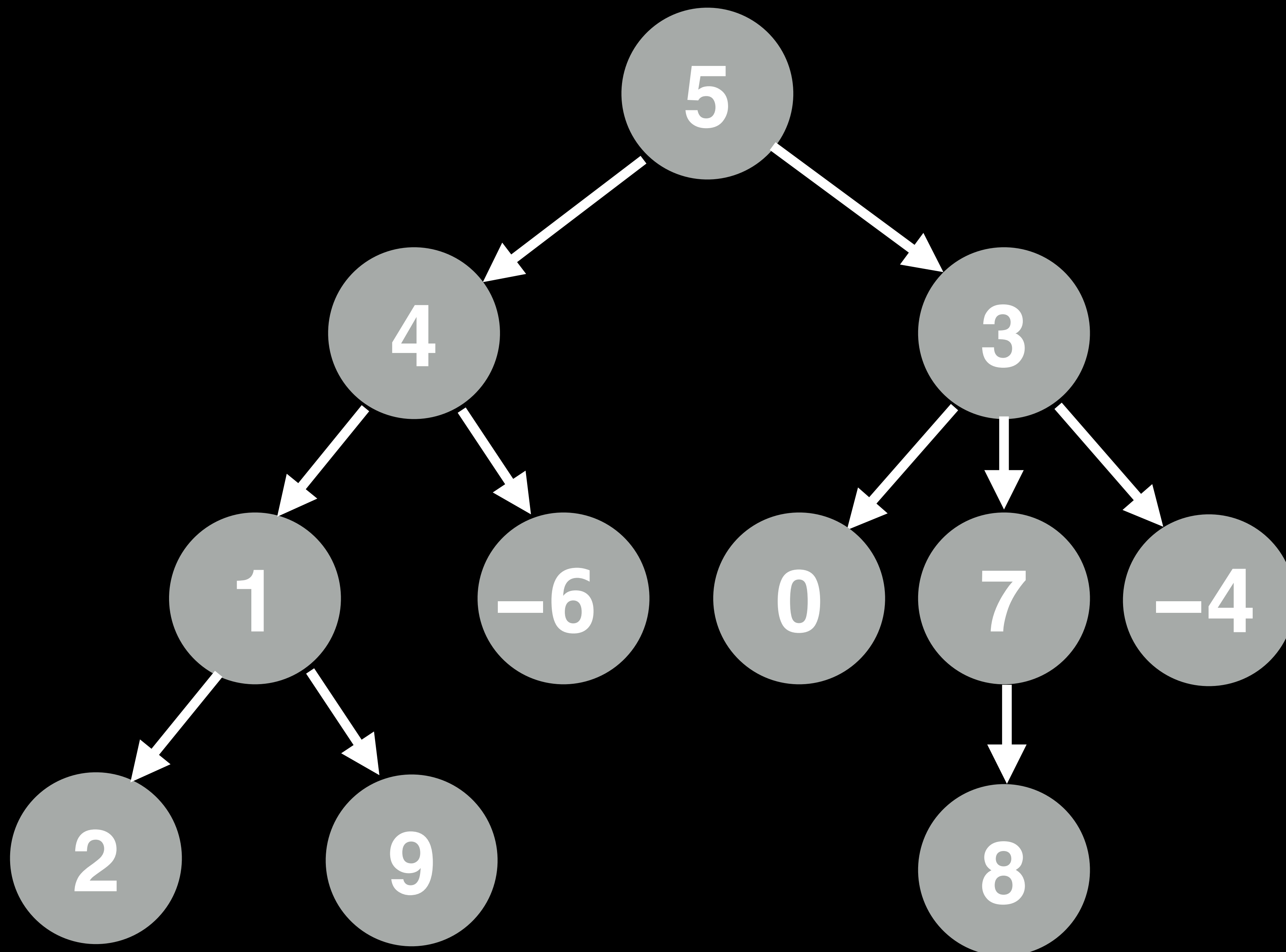
2 + 9 − 6 + 0 + 8



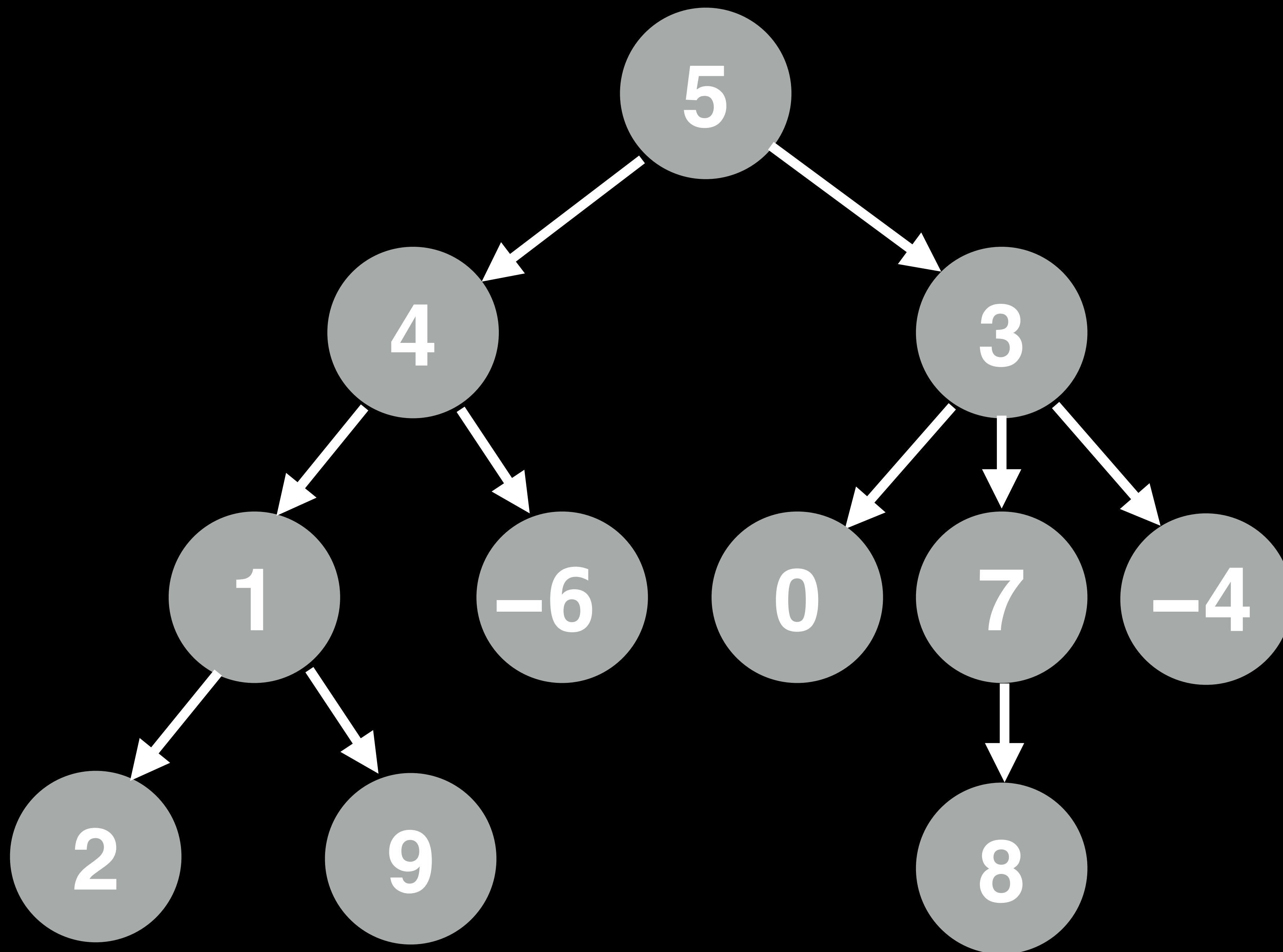
2 + 9 − 6 + 0 + 8 − 4



2 + 9 − 6 + 0 + 8 − 4



$2 + 9 - 6 + 0 + 8 - 4$



$$2 + 9 - 6 + 0 + 8 - 4 = 9$$

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

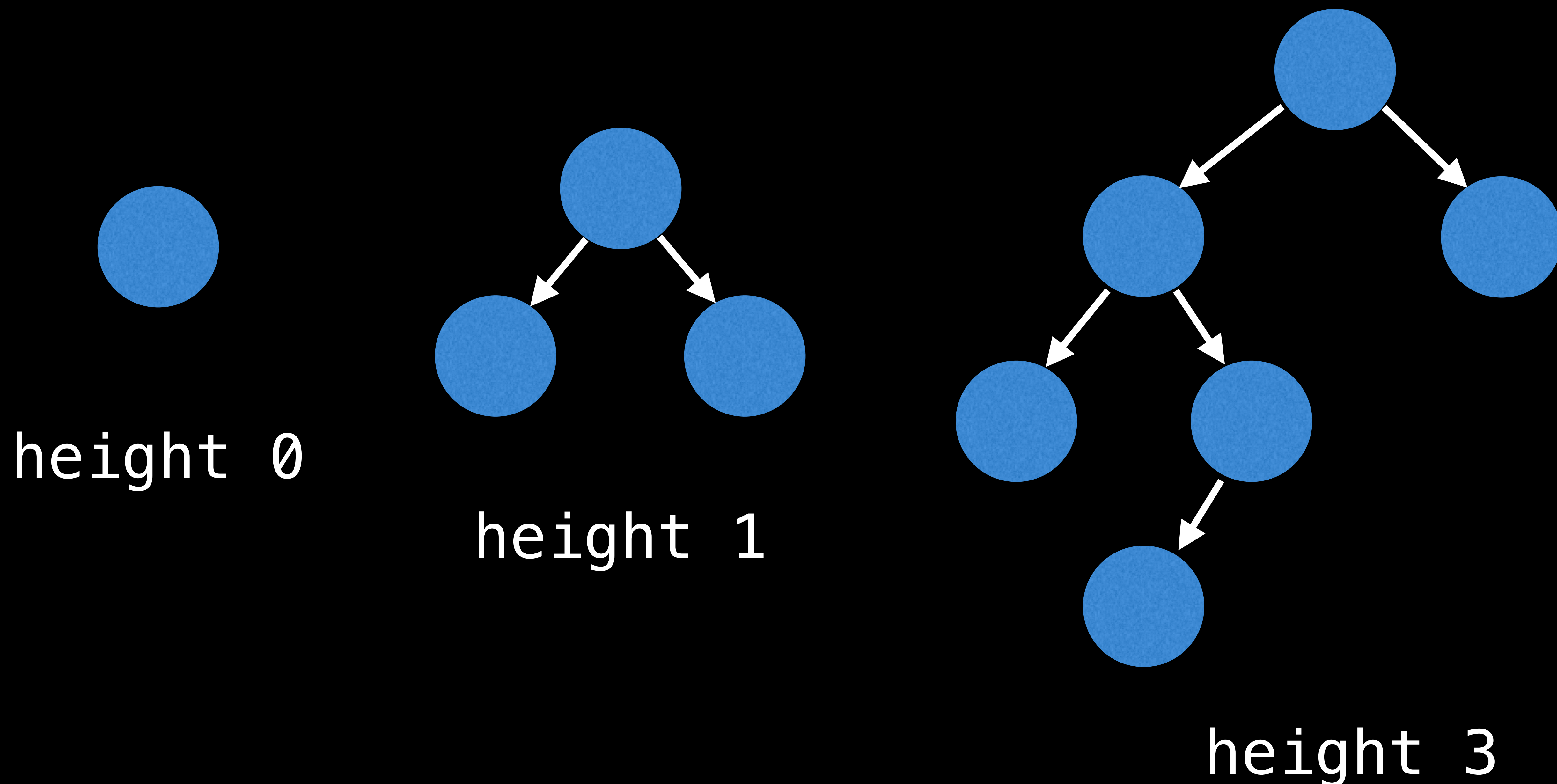
```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

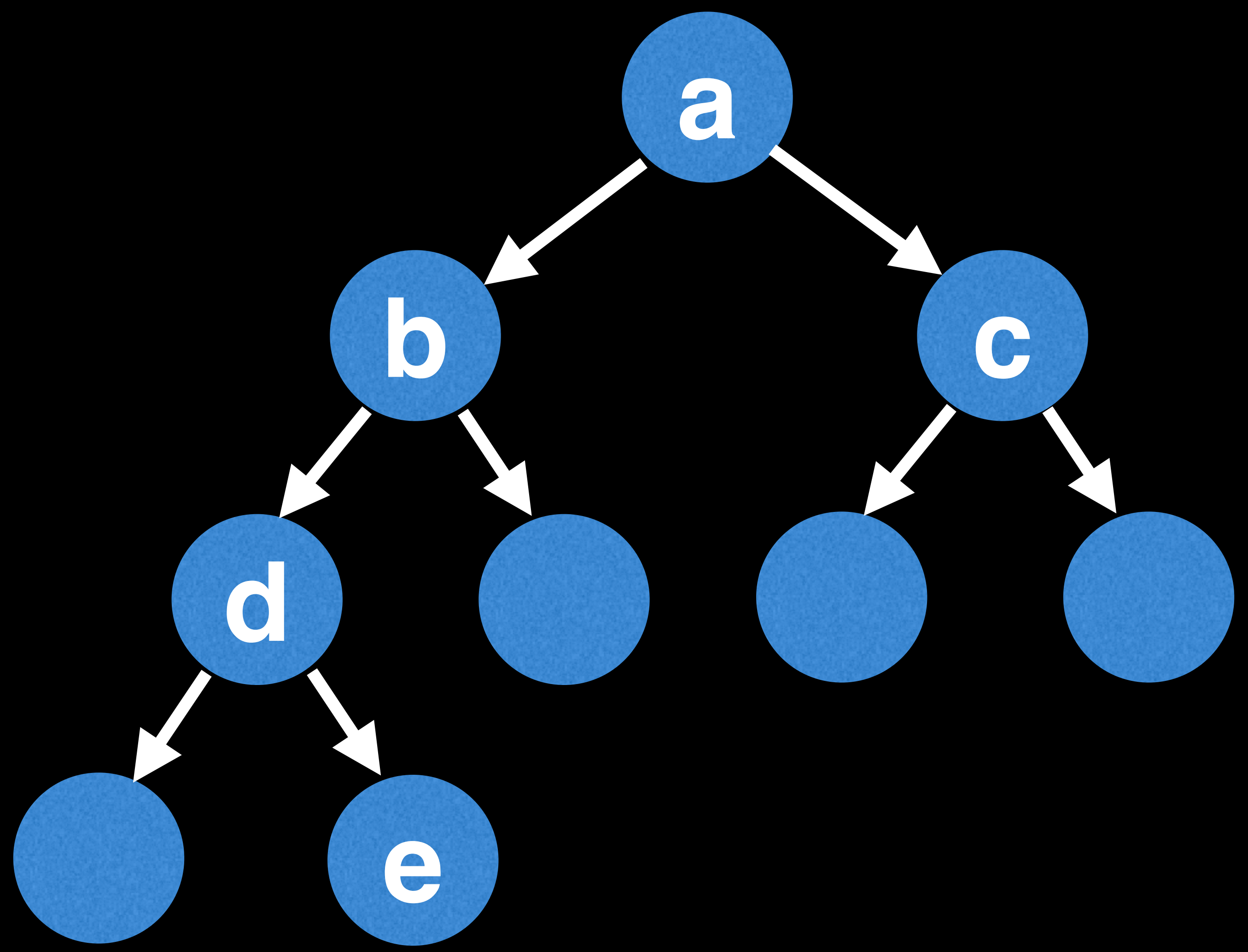
```
# Sums up leaf node values in a tree.
# Call function like: leafSum(root)
function leafSum(node):
  # Handle empty tree case
  if node == null:
    return 0
  if isLeaf(node):
    return node.getValue()
  total = 0
  for child in node.getChildNodes():
    total += leafSum(child)
  return total

function isLeaf(node):
  return node.getChildNodes().size() == 0
```

Problem 2: Tree Height

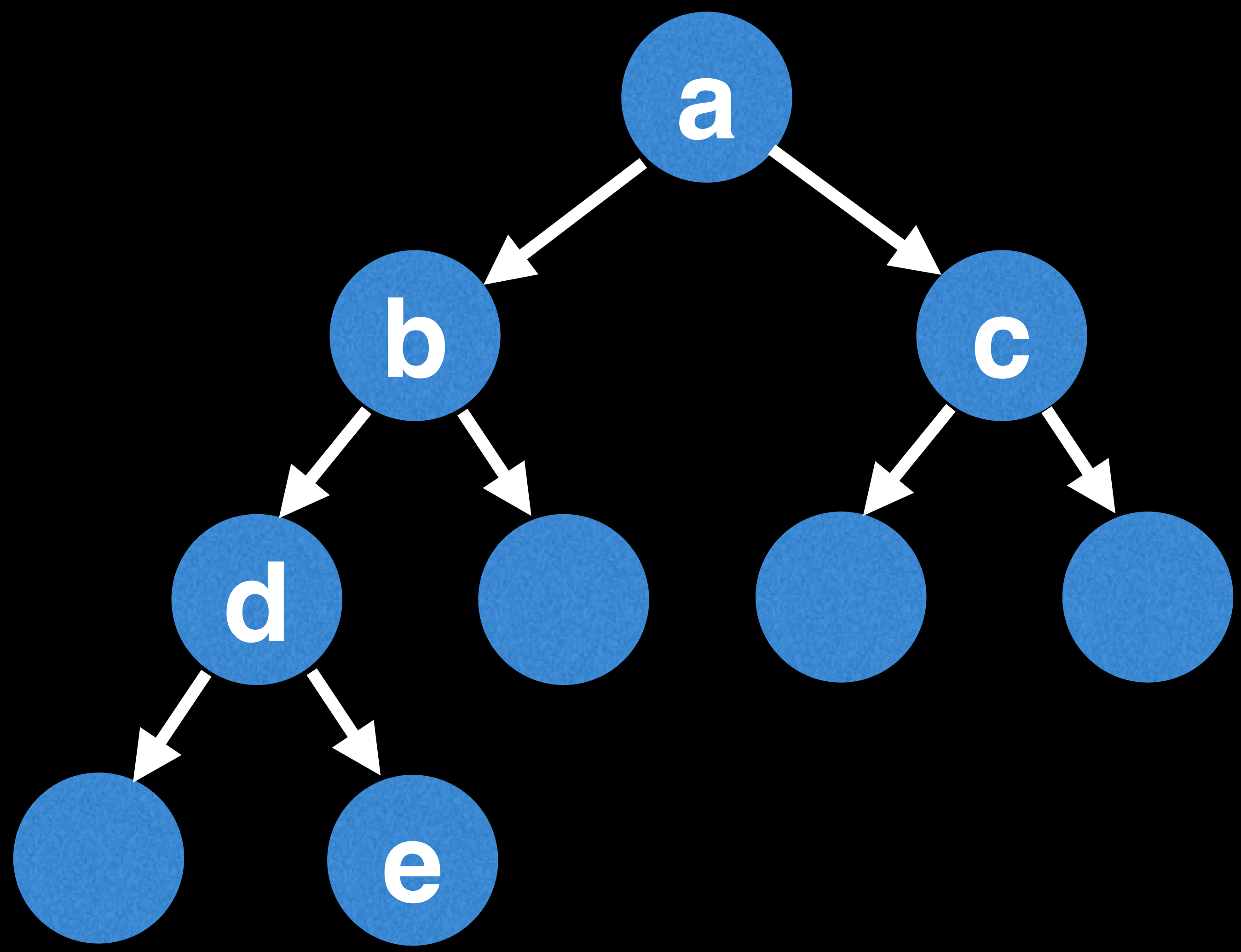
Find the **height** of a **binary tree**. The **height** of a tree is the number of edges from the root to the lowest leaf.



Let **h**(x) be the height of the subtree rooted at node x.

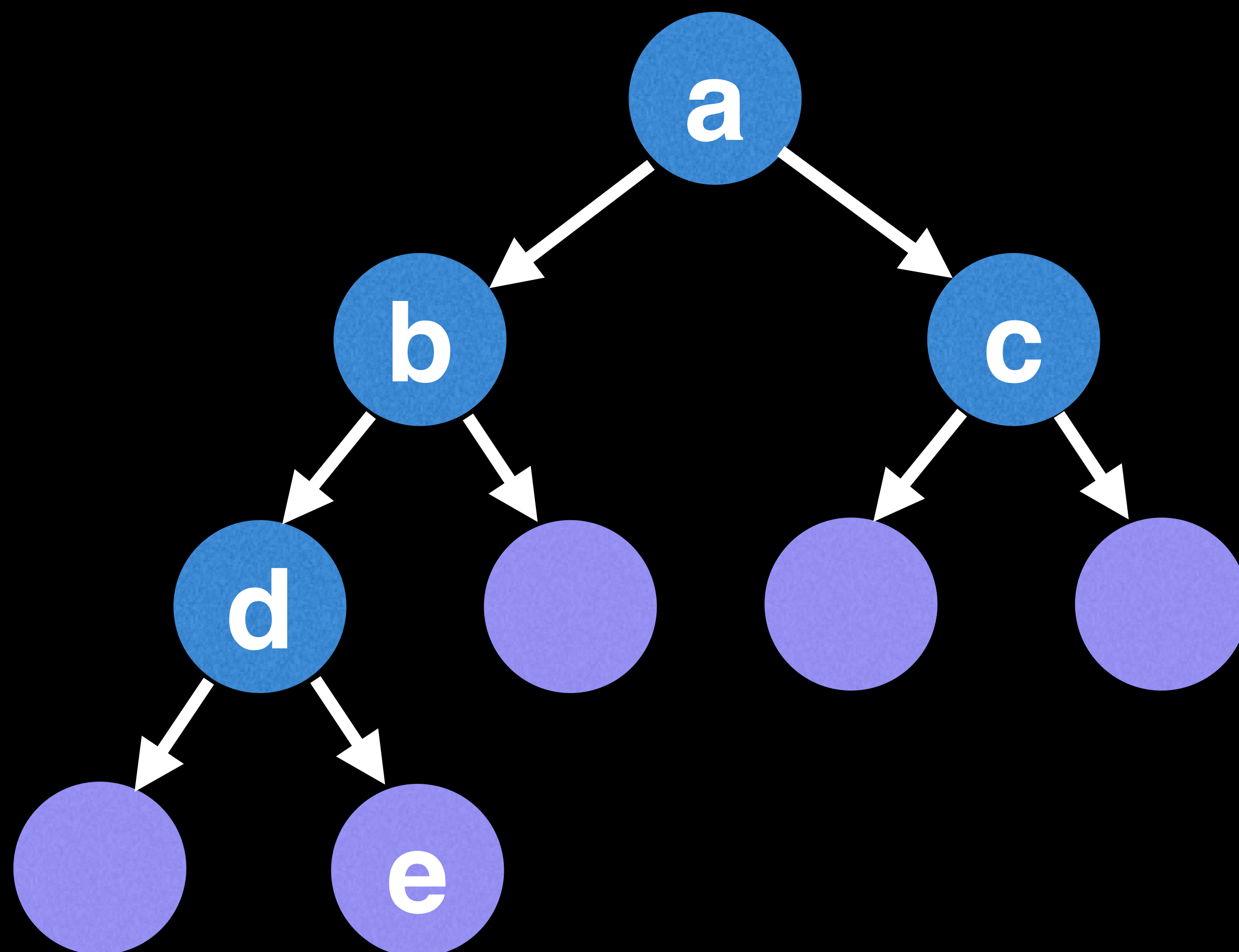


Let **h**(x) be the height of the subtree rooted at node x.



h(a) = 3, **h**(b) = 2, **h**(c) = 1, **h**(d) = 1, **h**(e) = 0

By themselves, leaf nodes such as node **e** don't have children, so they don’t add any additional height to the tree.

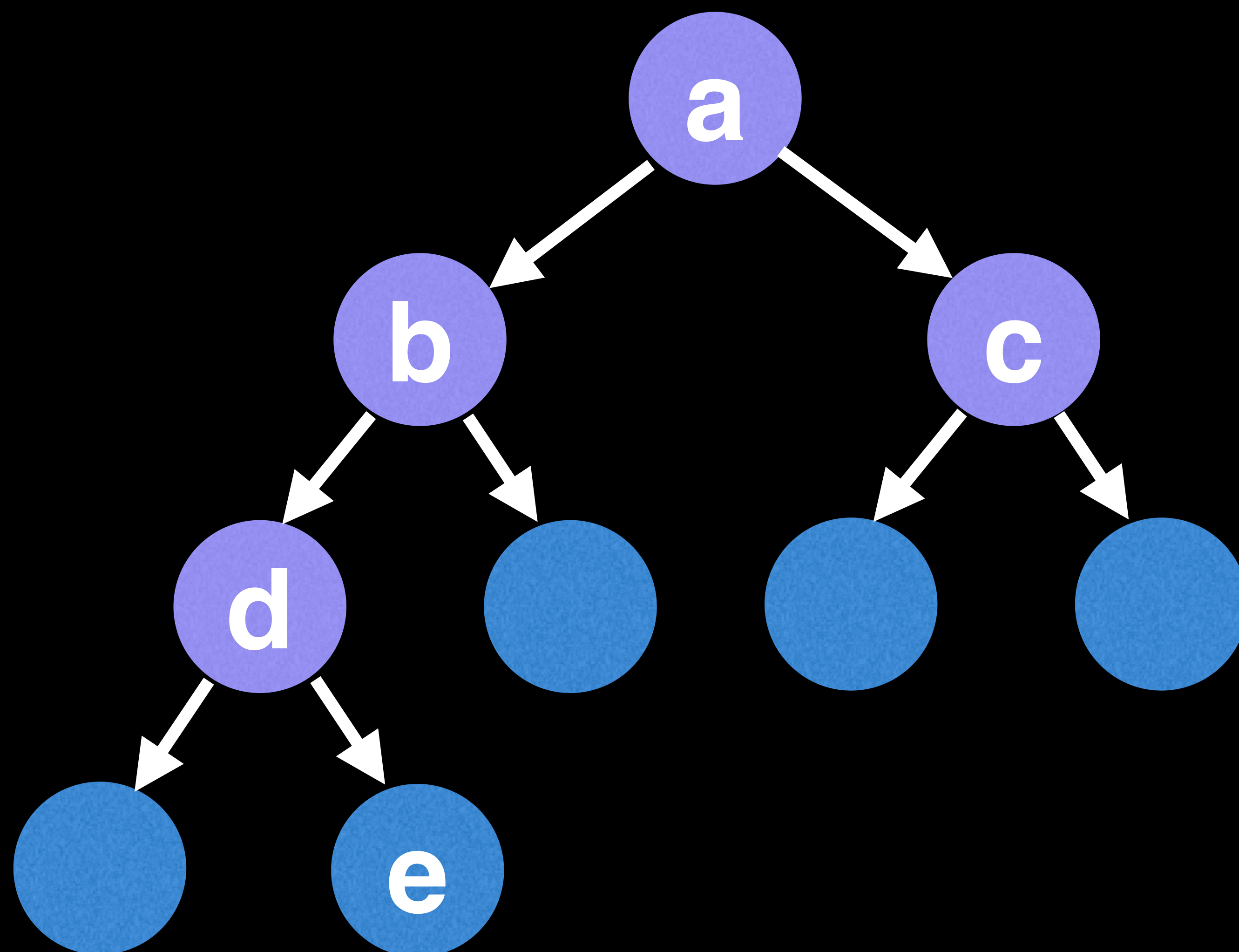


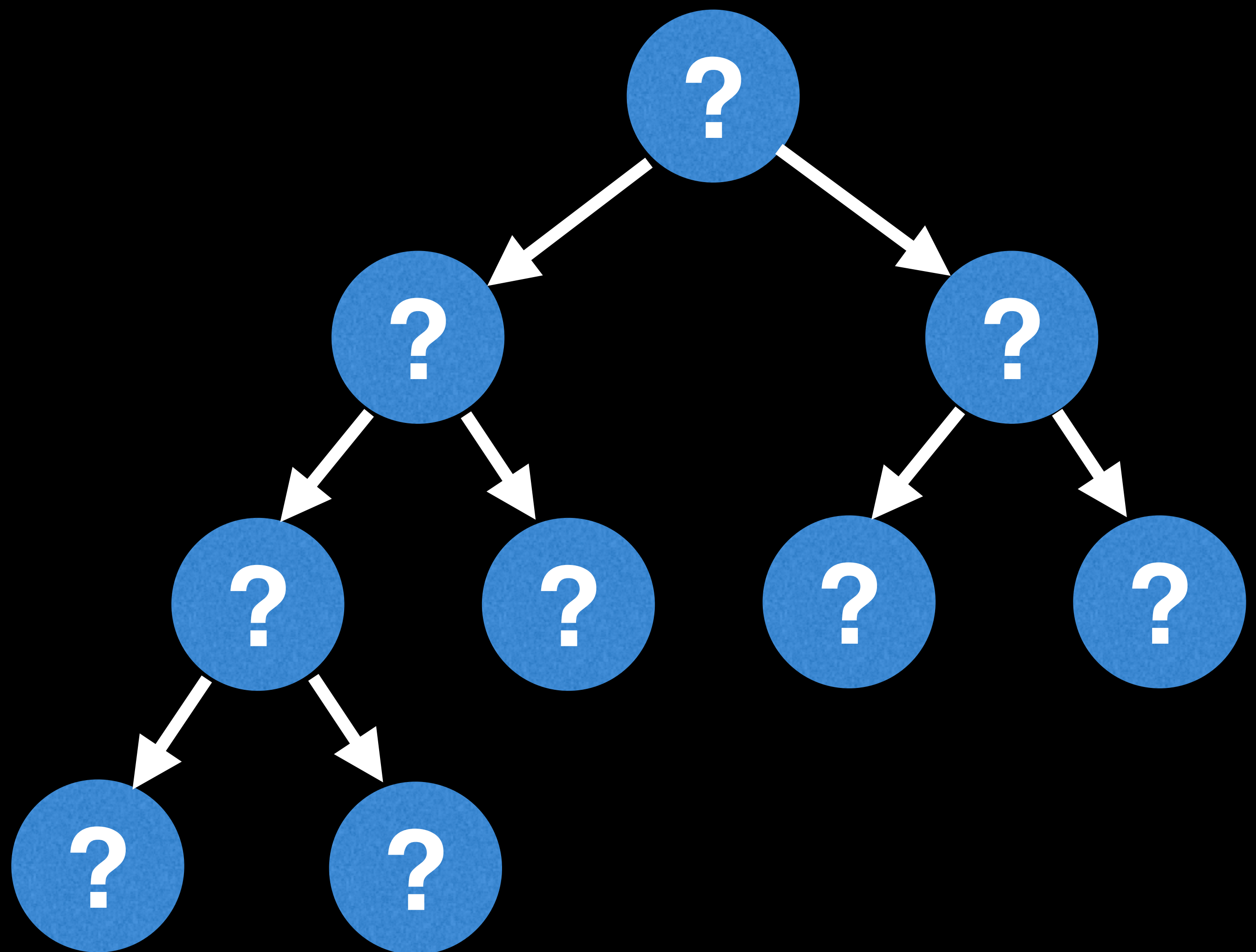
As a base case we can conclude that:

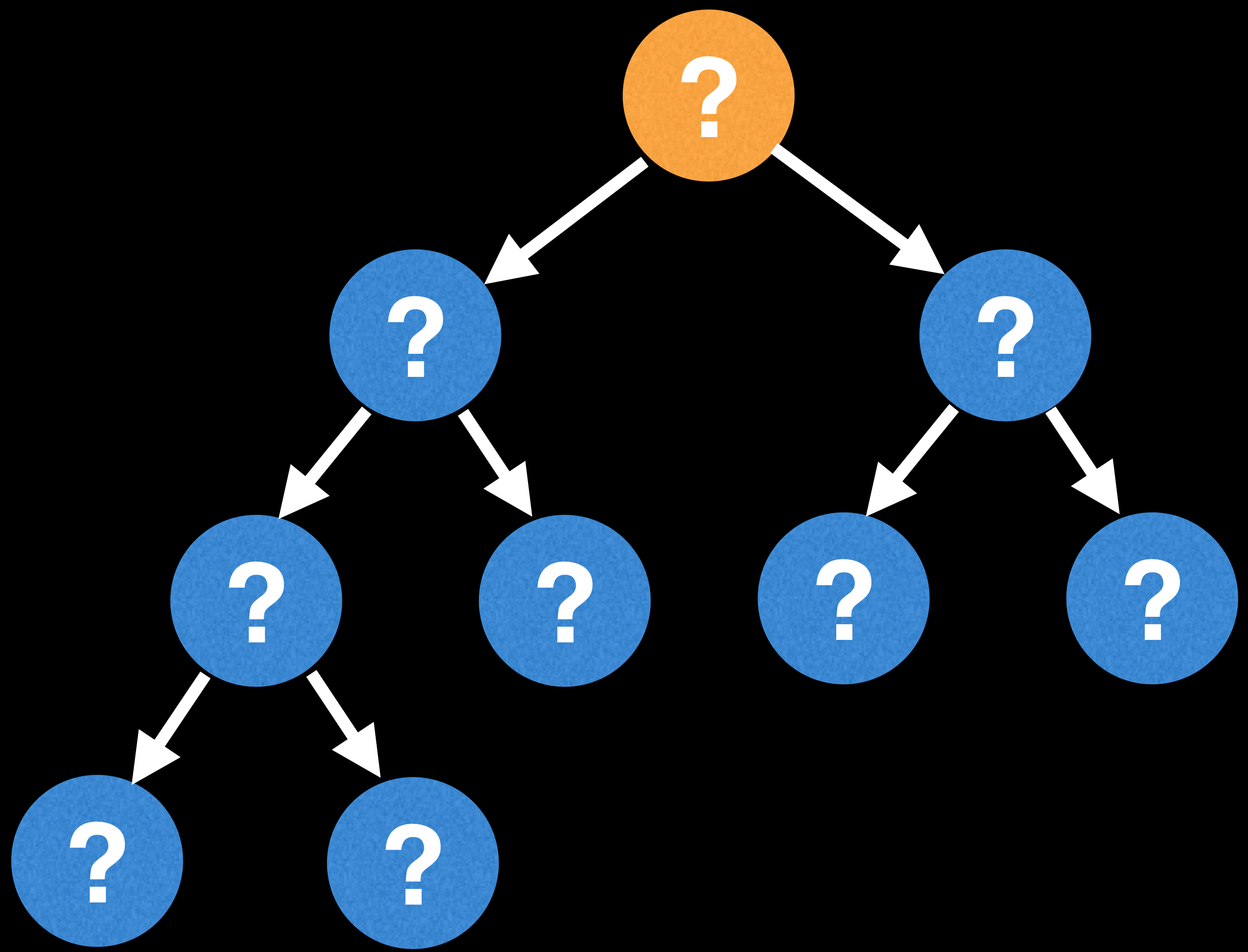
h(leaf node) = 0

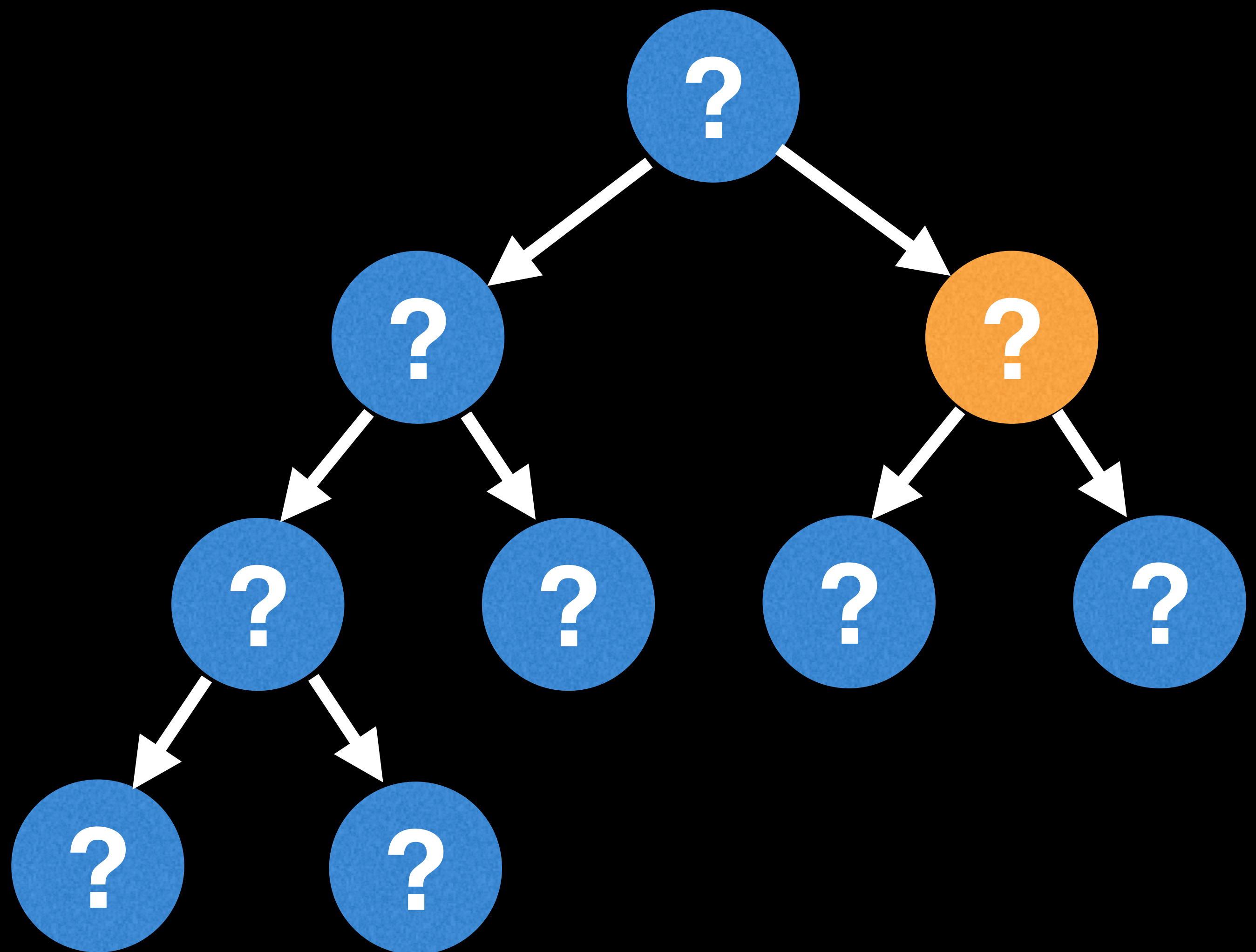
Assuming node x is not a leaf node, we're able to formulate a recurrence for the height:

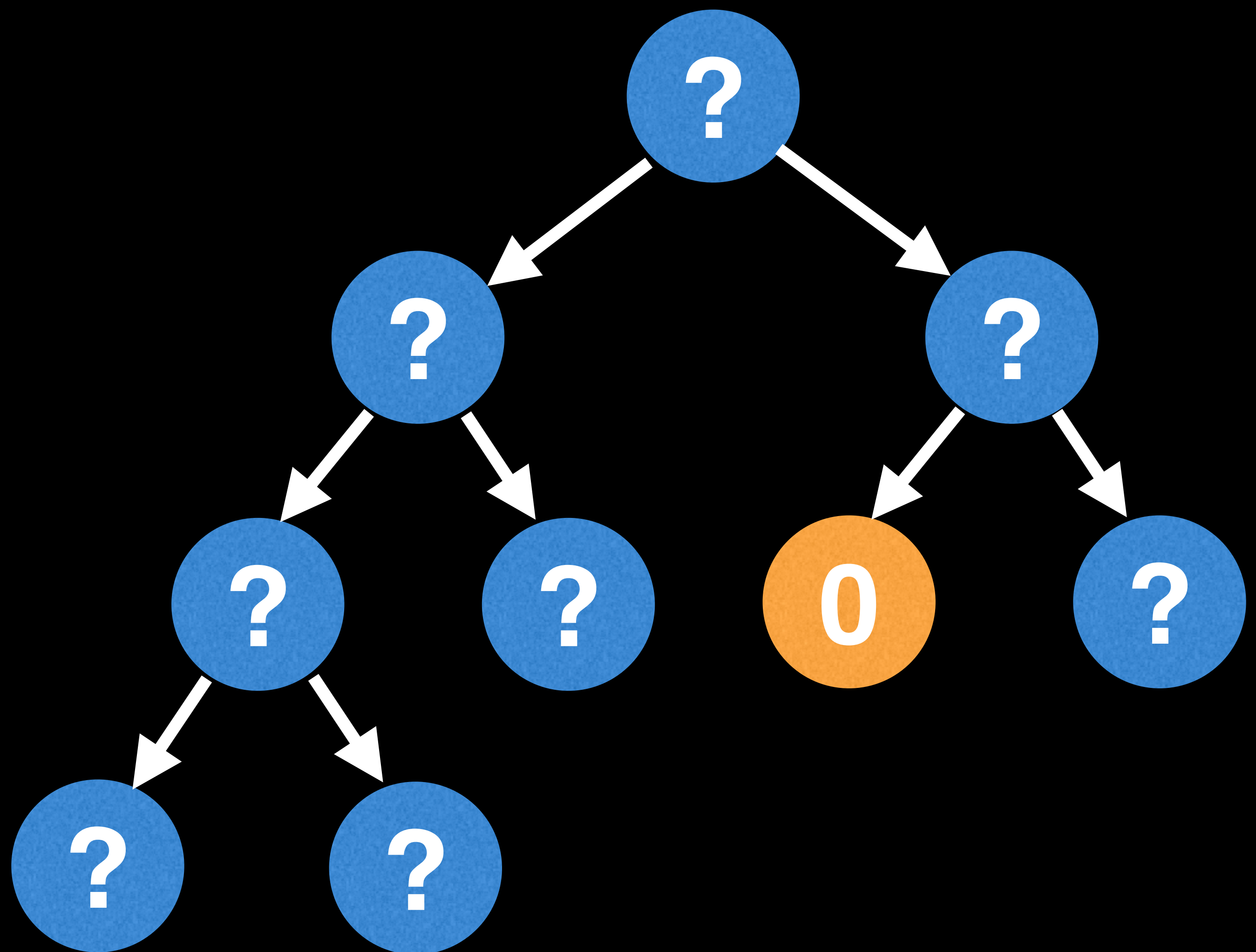
h(x) = **max**(**h**(x.left), **h**(x.right)) + 1



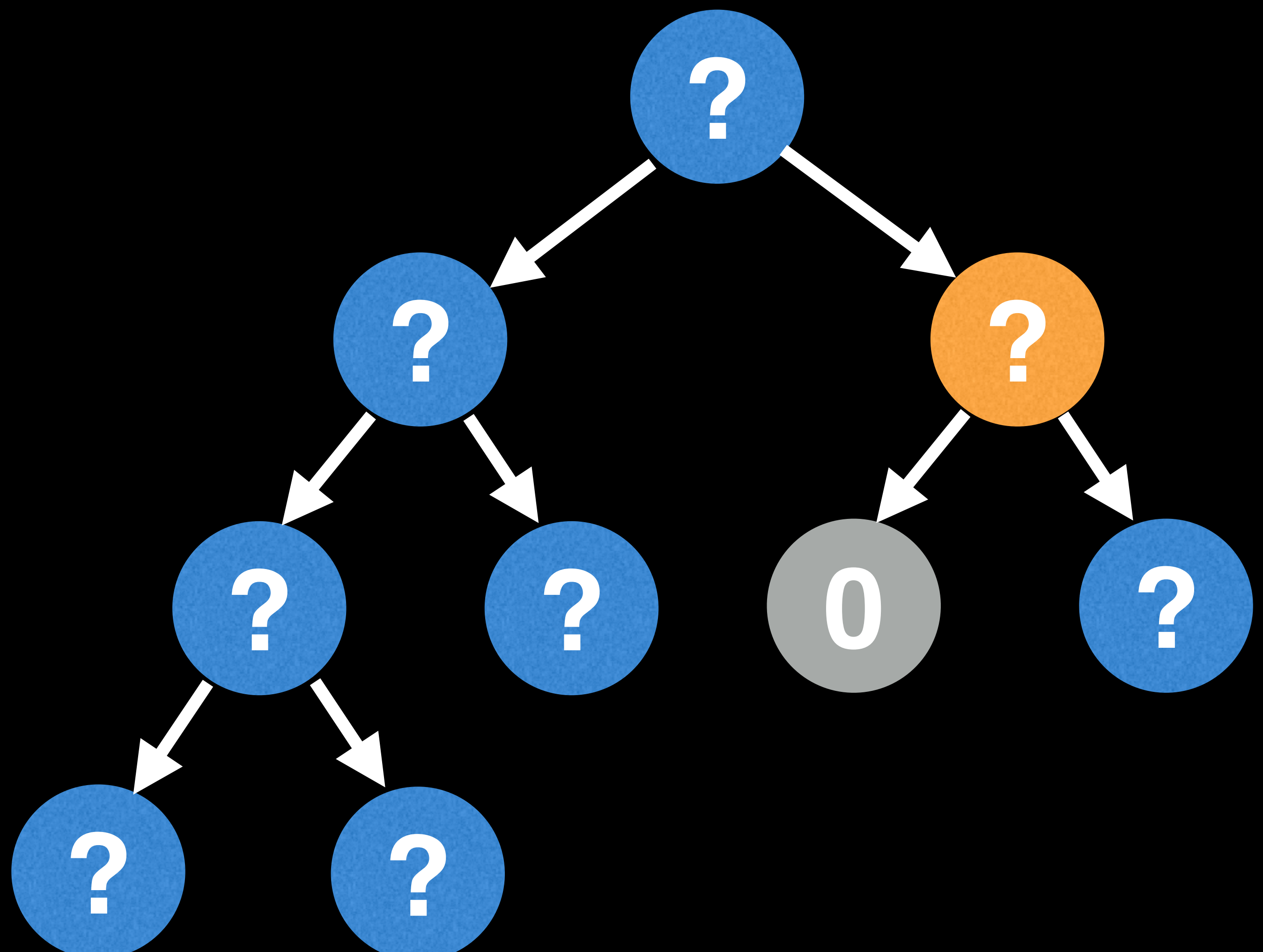


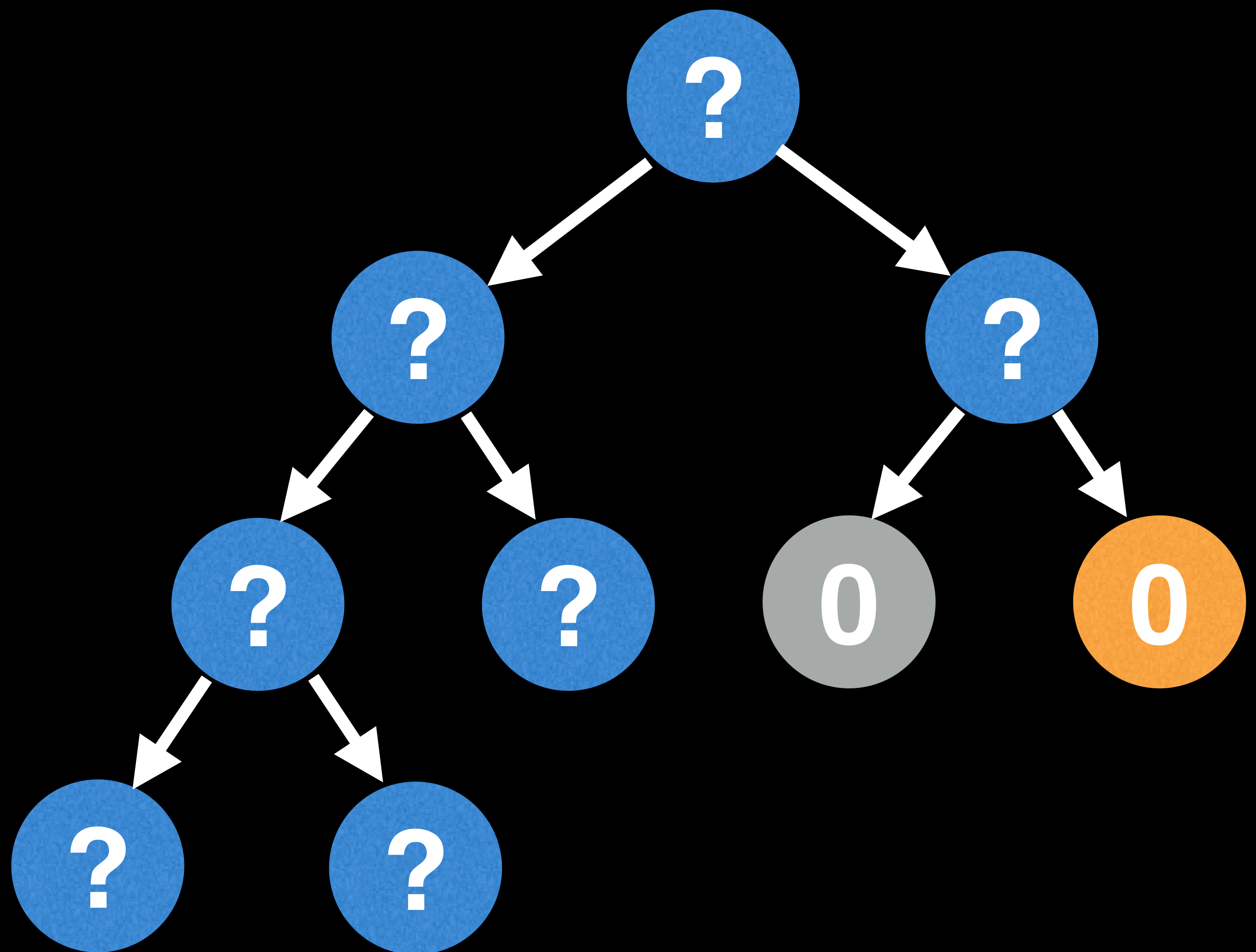




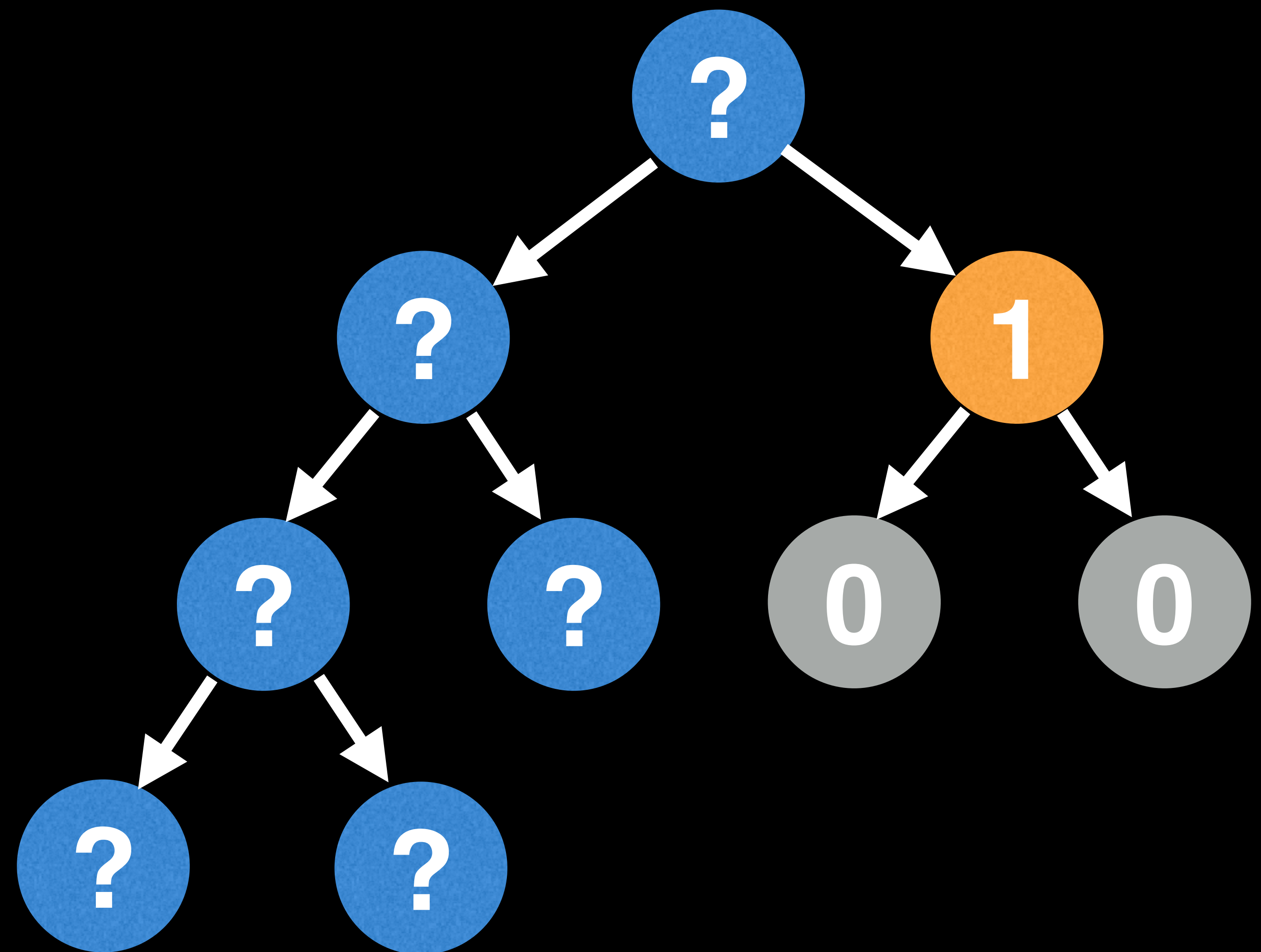


Leaf node has a height of 0

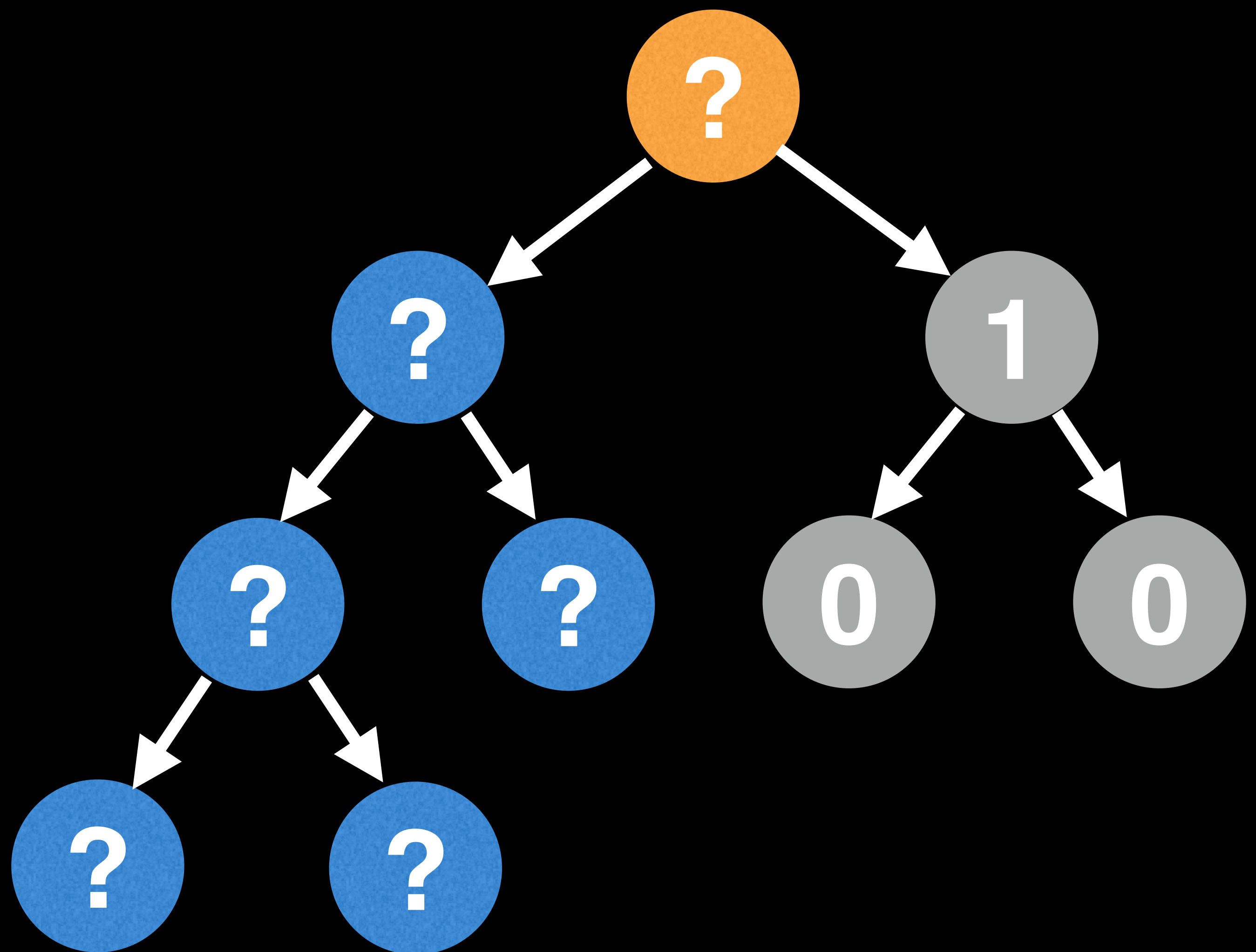

?
?
?
?
?
0
?
?
?

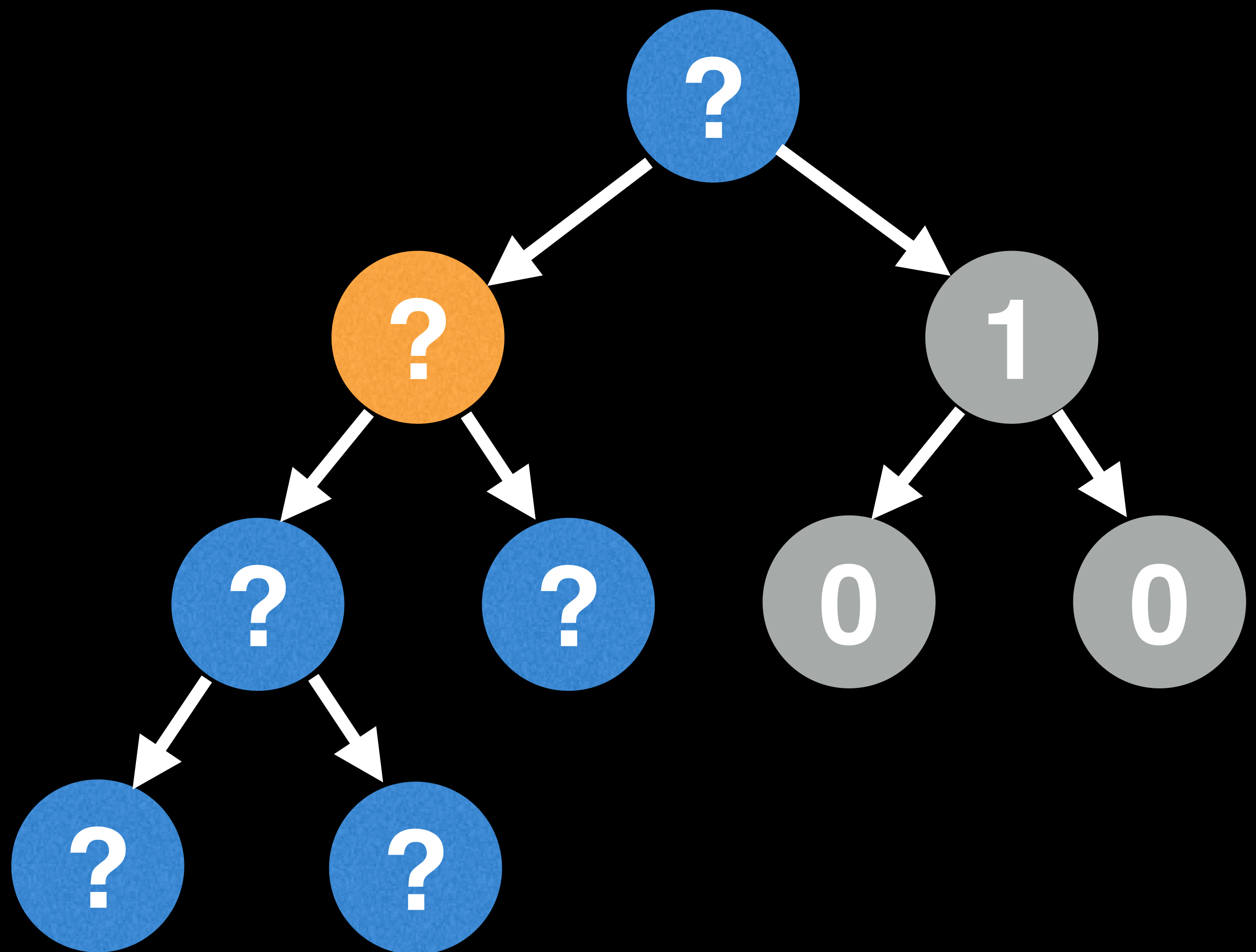


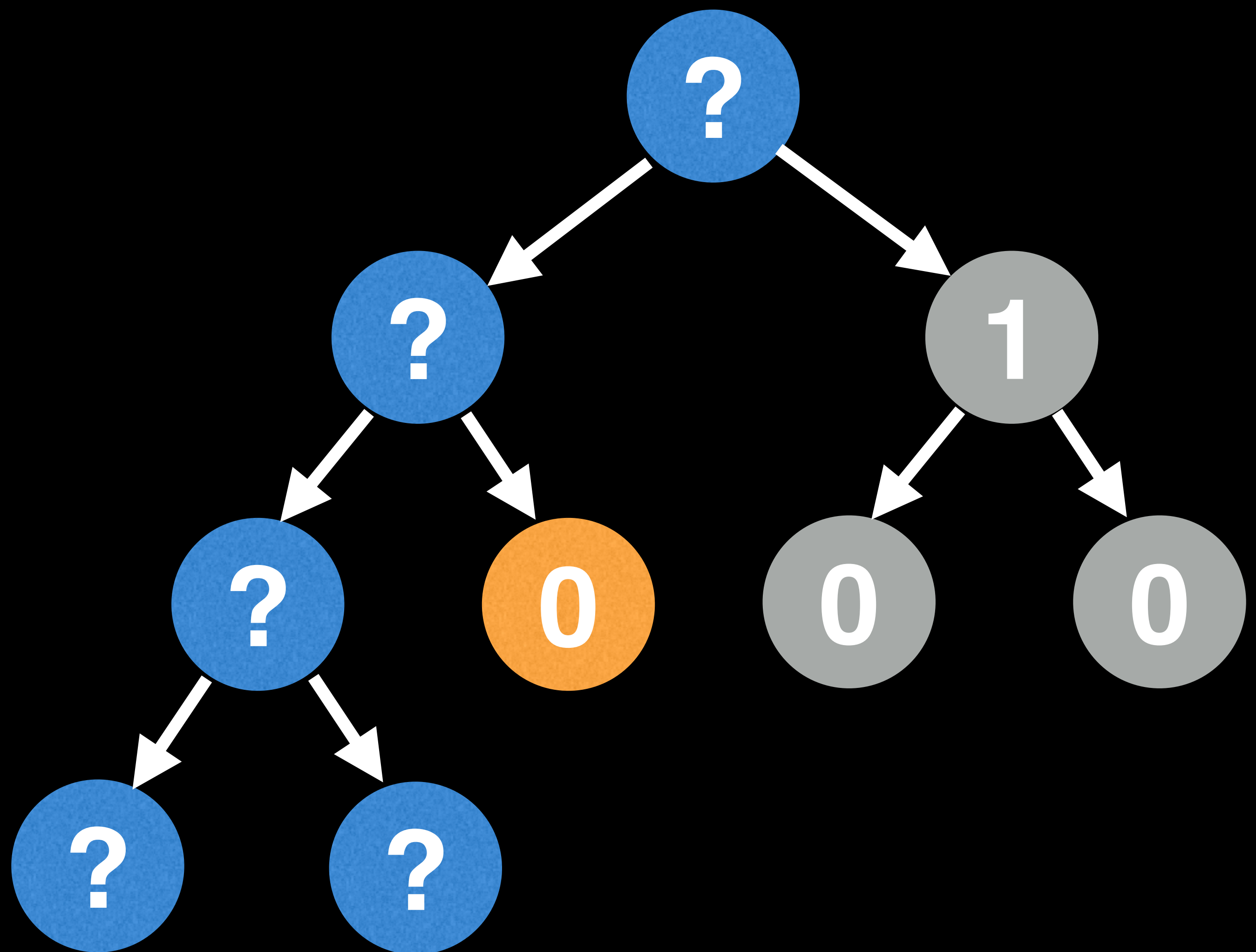
Leaf node has a height of 0



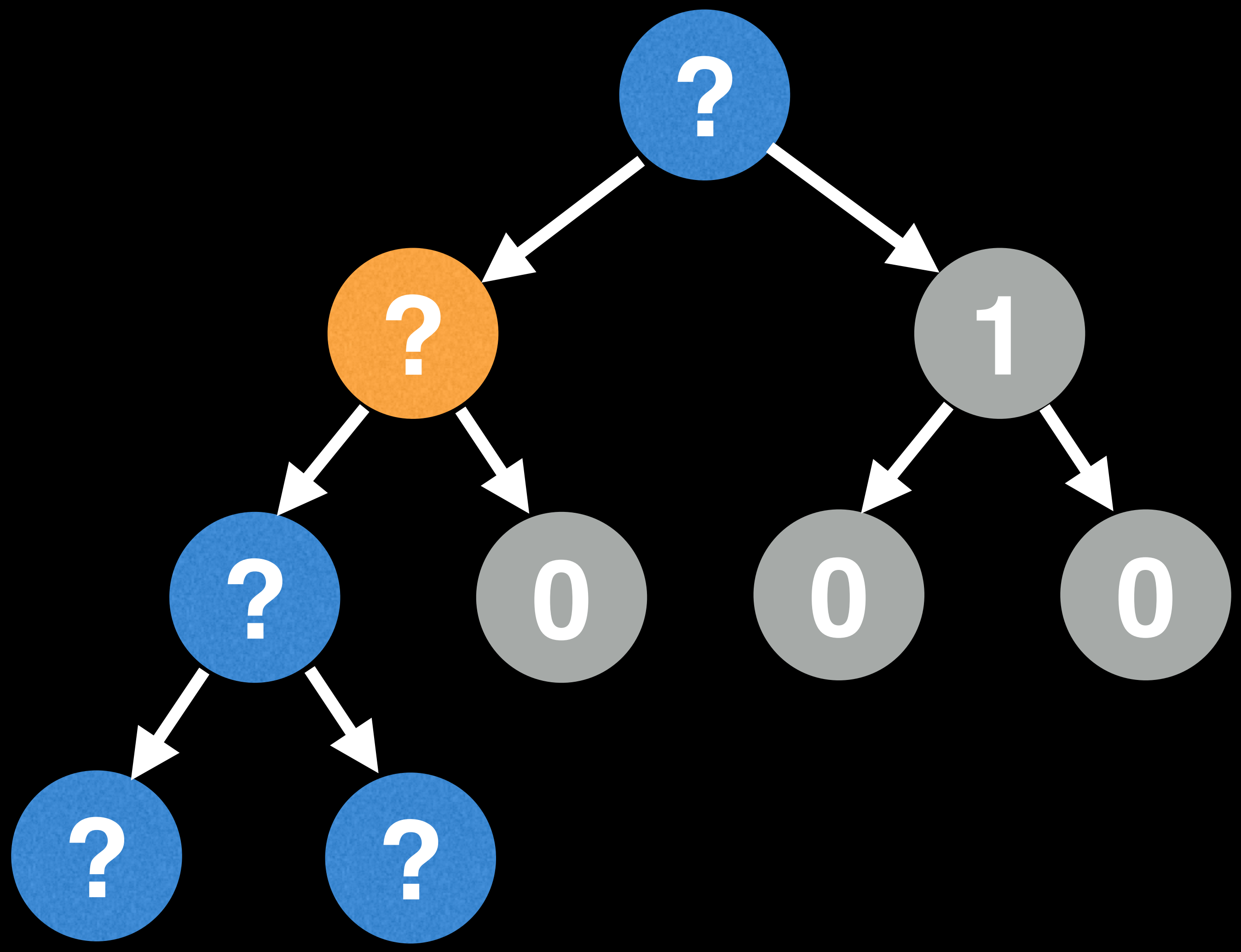
height = max(0, 0) + 1 = 1

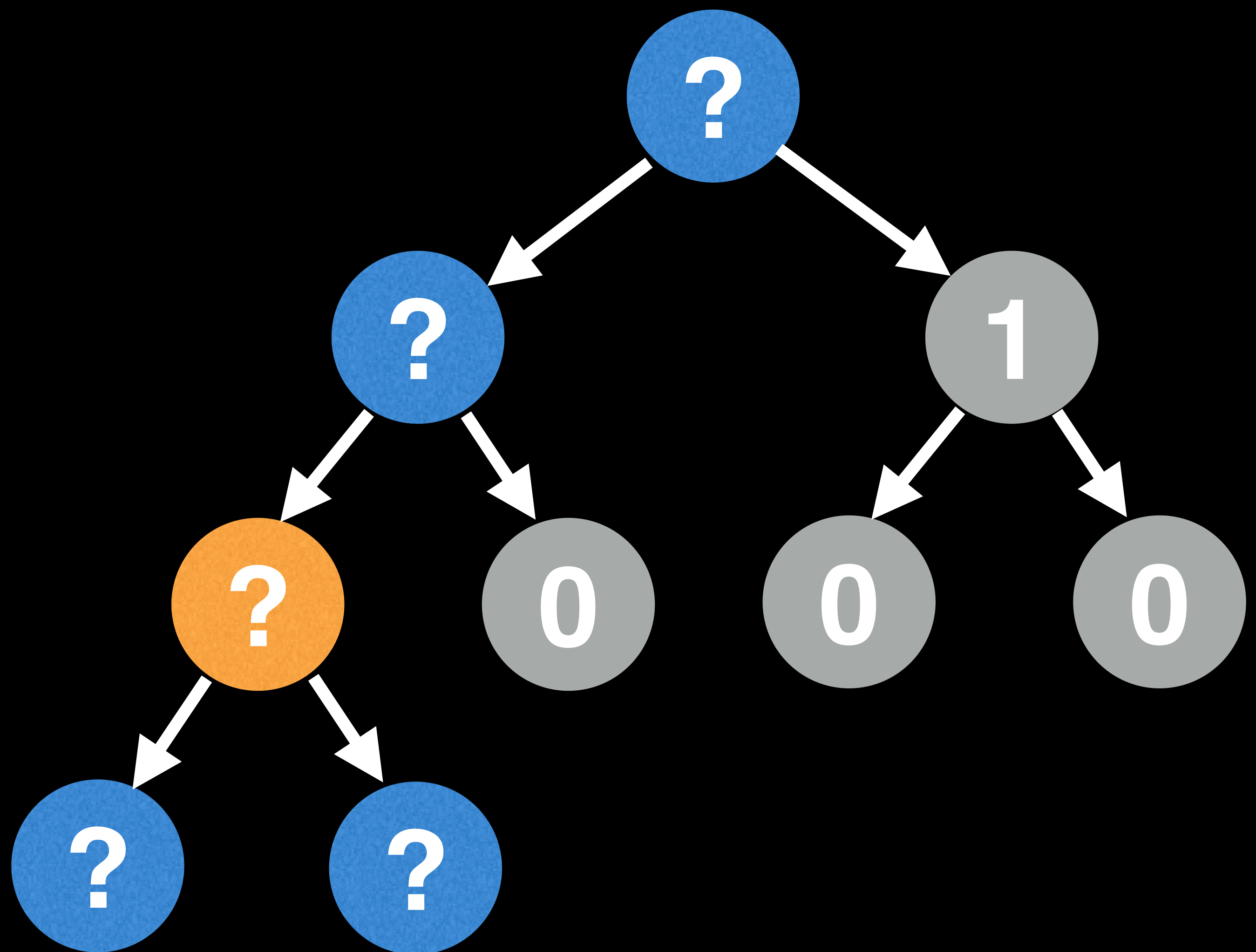

?
?
1
?
?
0
0
?
?

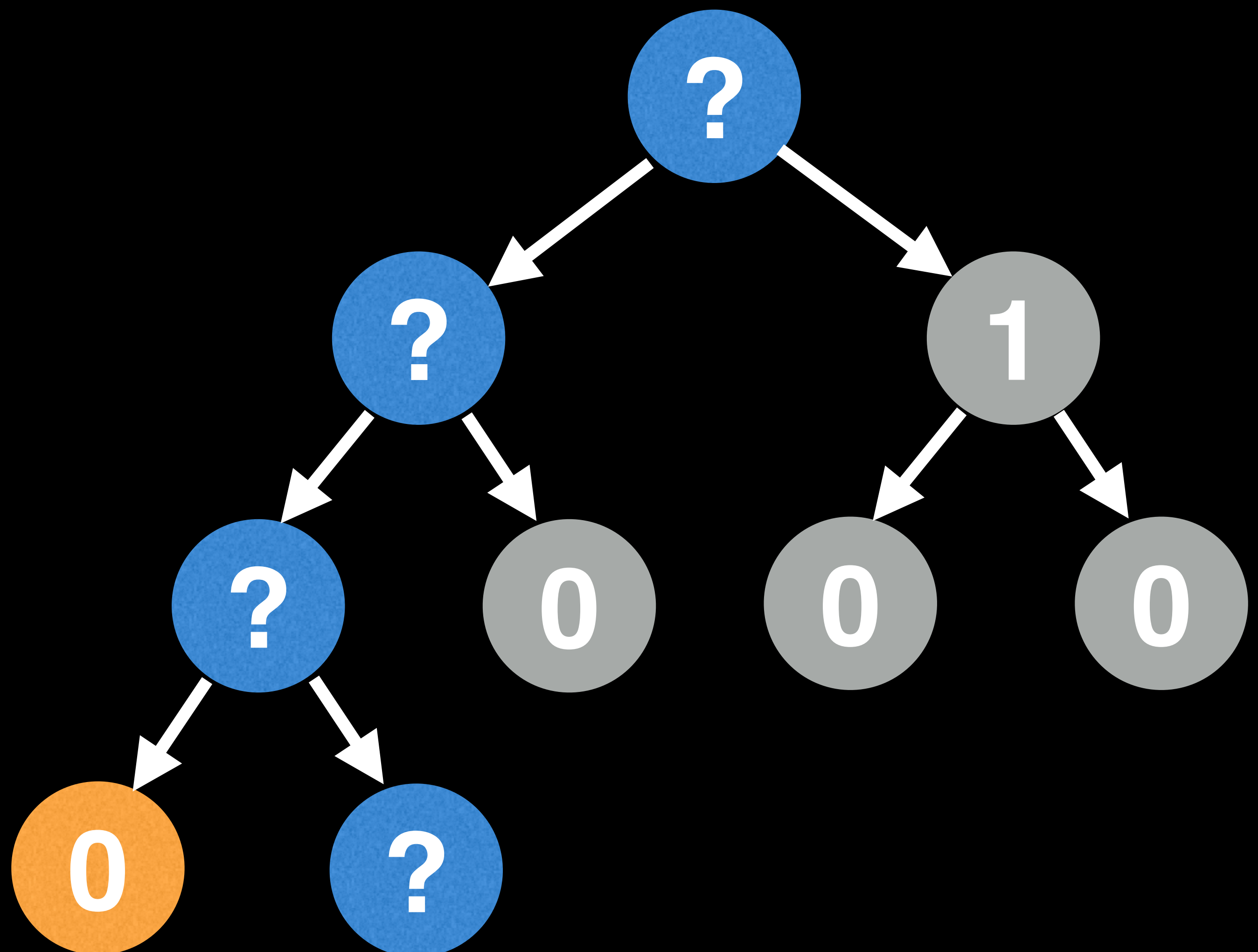

?
?
1
?
?
0
0
?
?



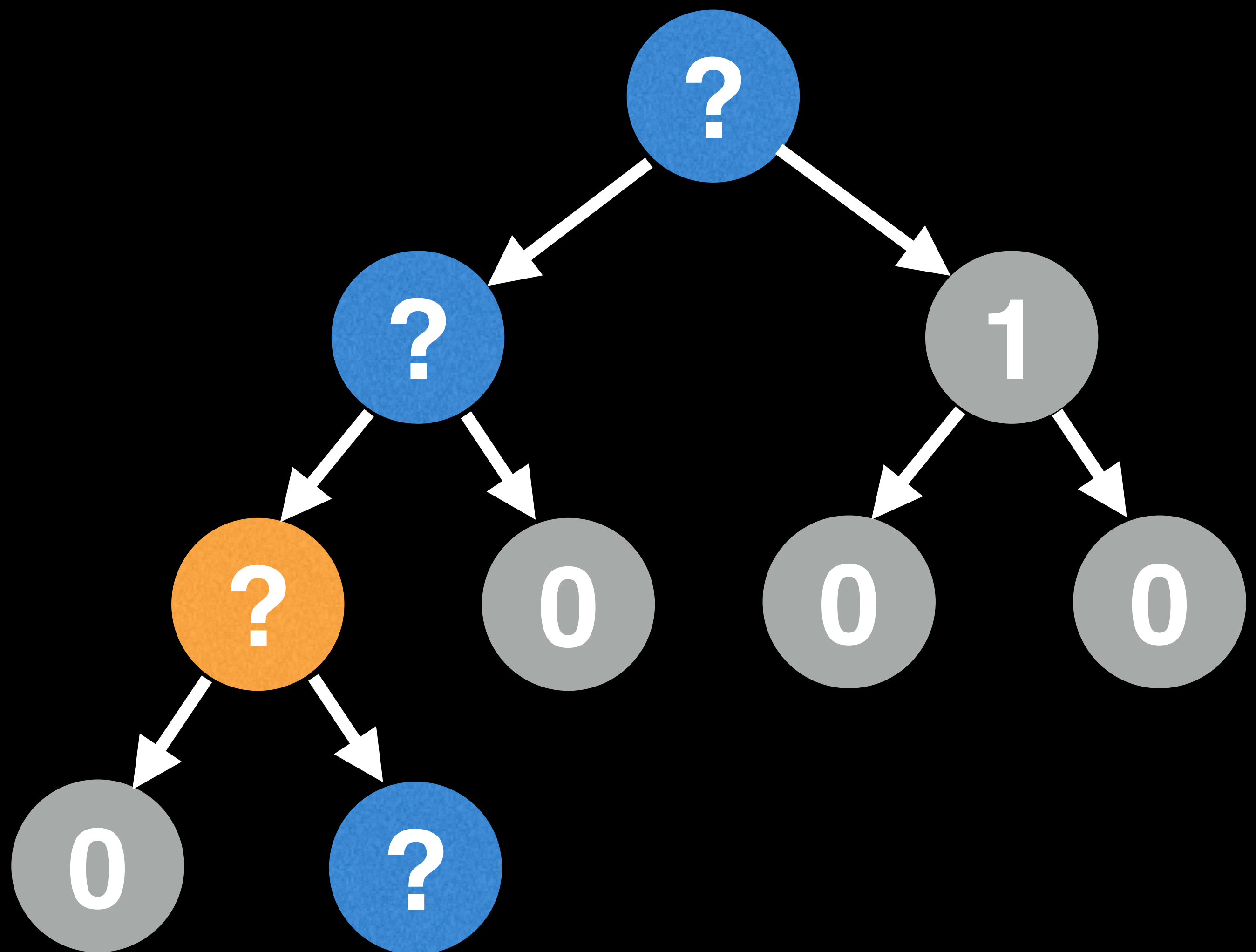
Leaf node has a height of 0

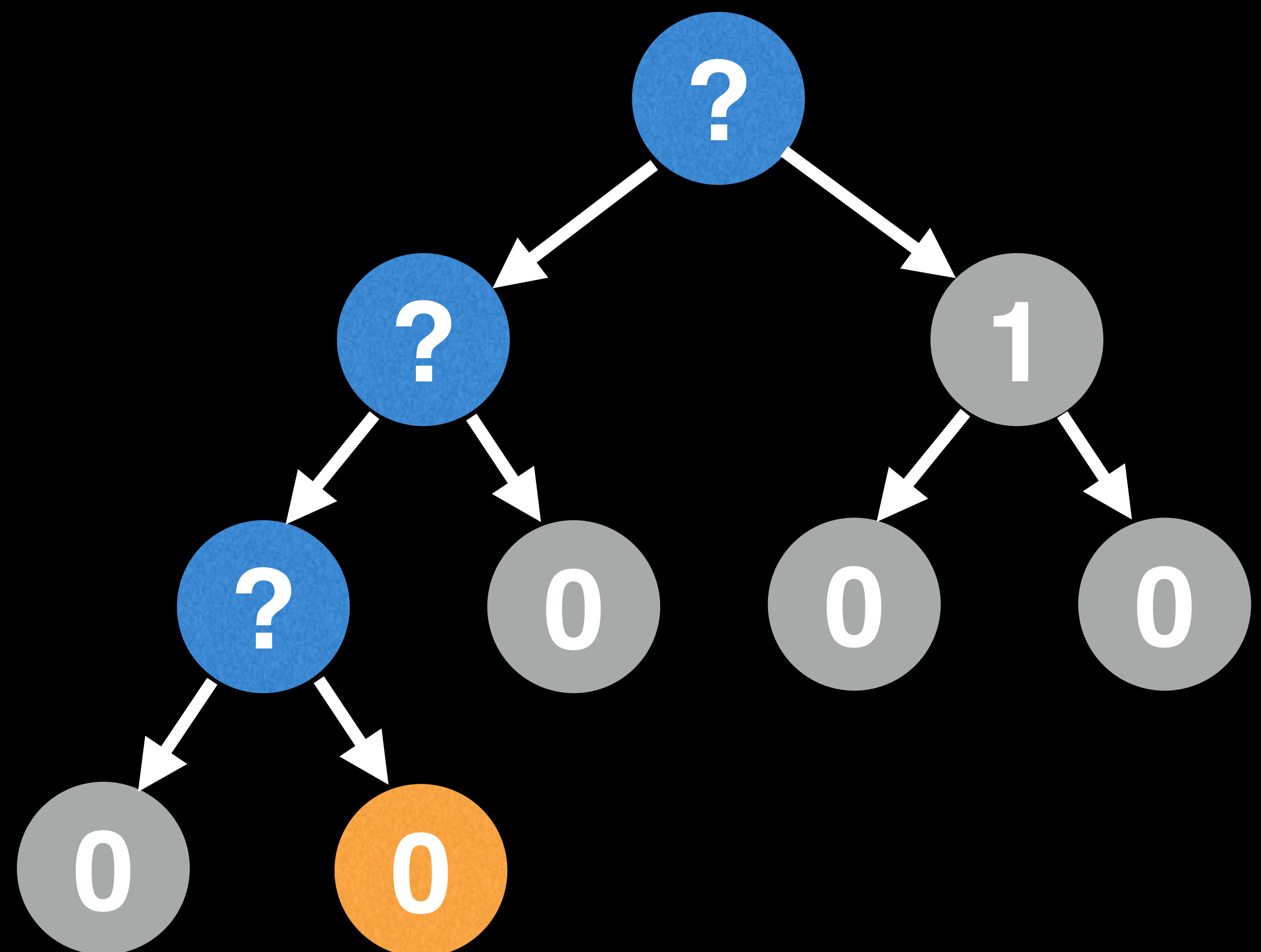

?
?
1
?
0
0
0
?
?


?
?
1
?
0
0
0
?
?

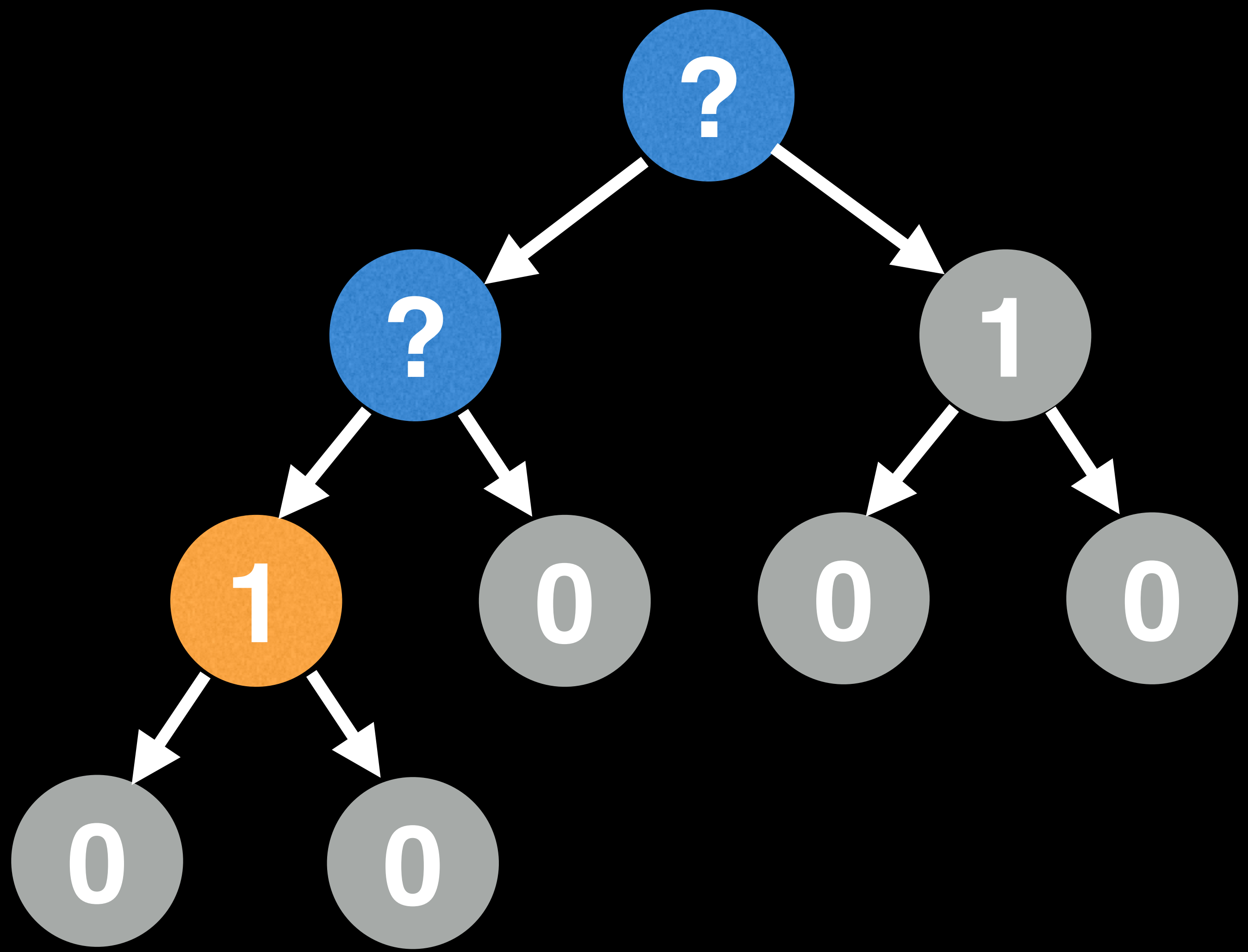


Leaf node has a height of 0

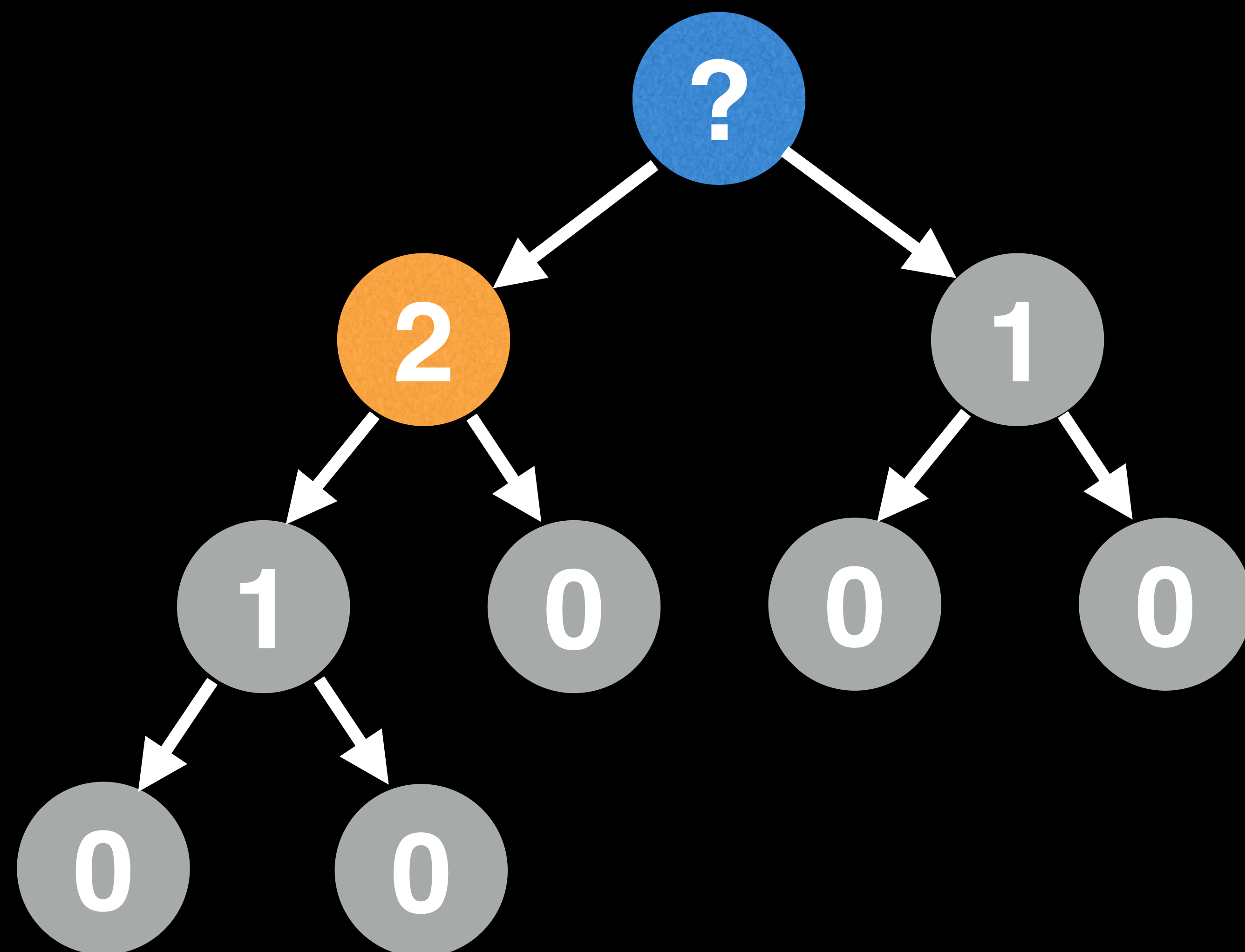

?
?
1
?
0
0
0
0
?



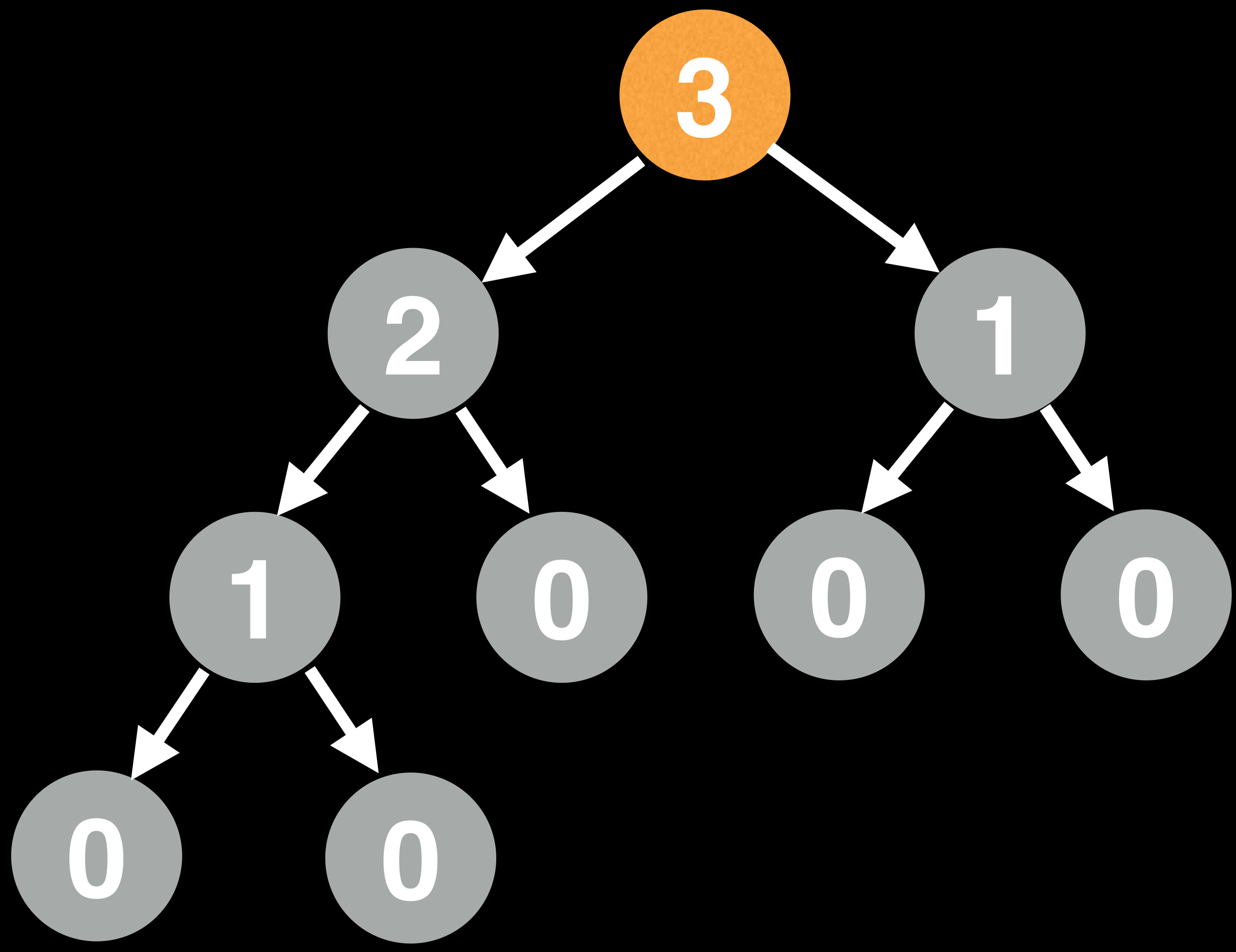
Leaf node has a height of 0



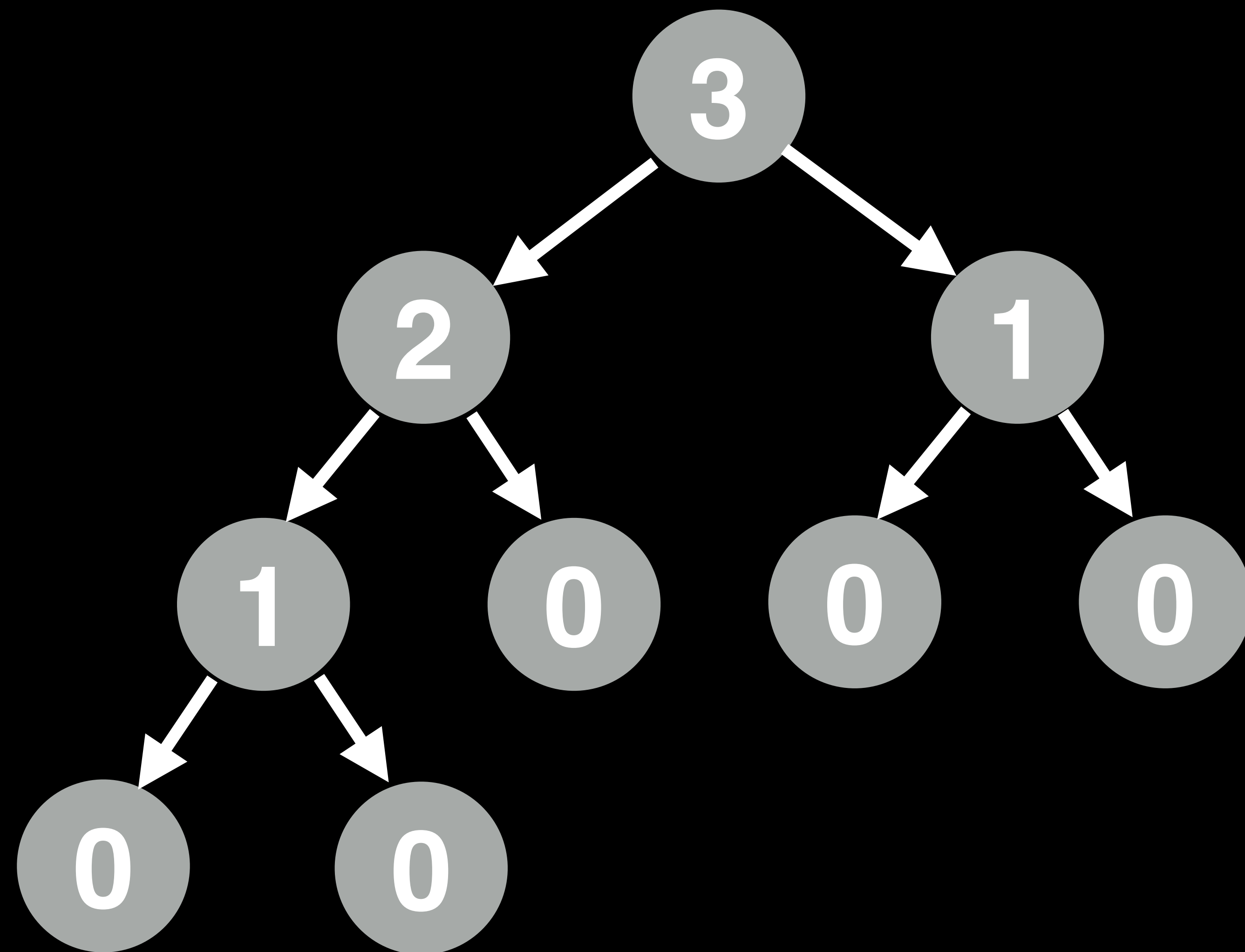
height = max(0, 0) + 1 = 1



height = max(1, 0) + 1 = 2



height = max(2, 1) + 1 = 3


3
2
1
1
0
0
0
0
0

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

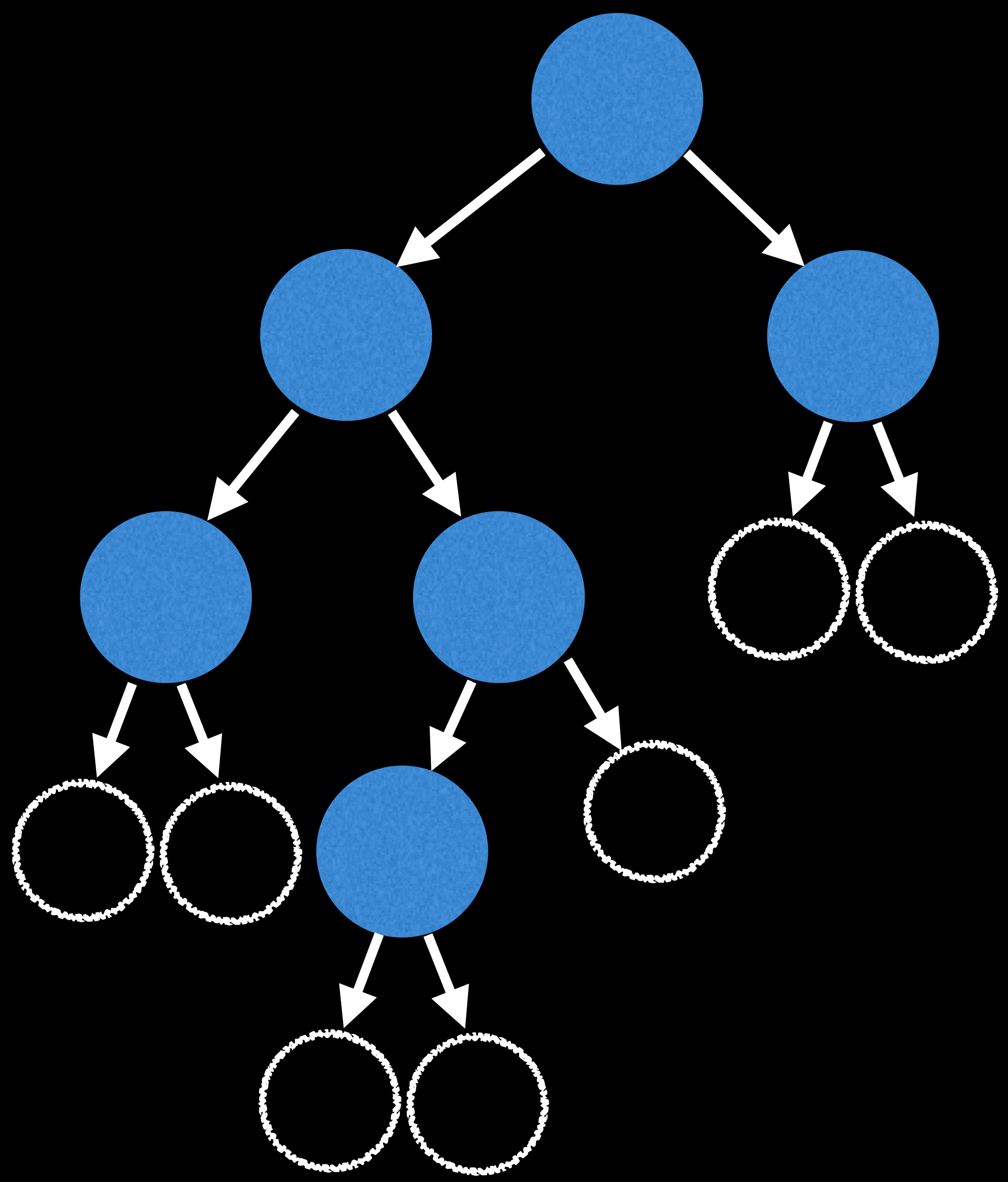
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

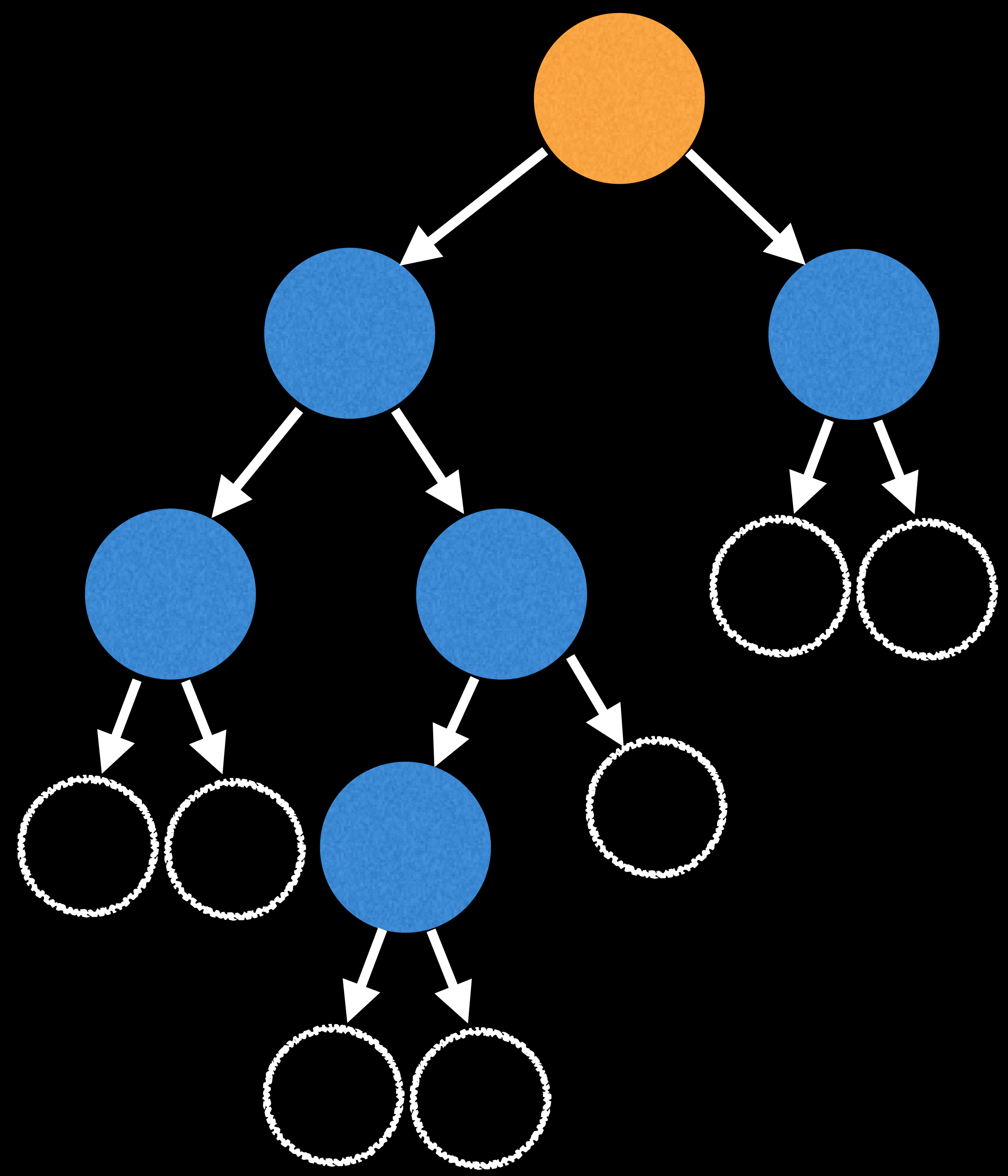
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Handle empty tree case
  if node == null:
     return -1

  # Identify leaf nodes and return zero
  if node.left == null and node.right == null:
    return 0

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

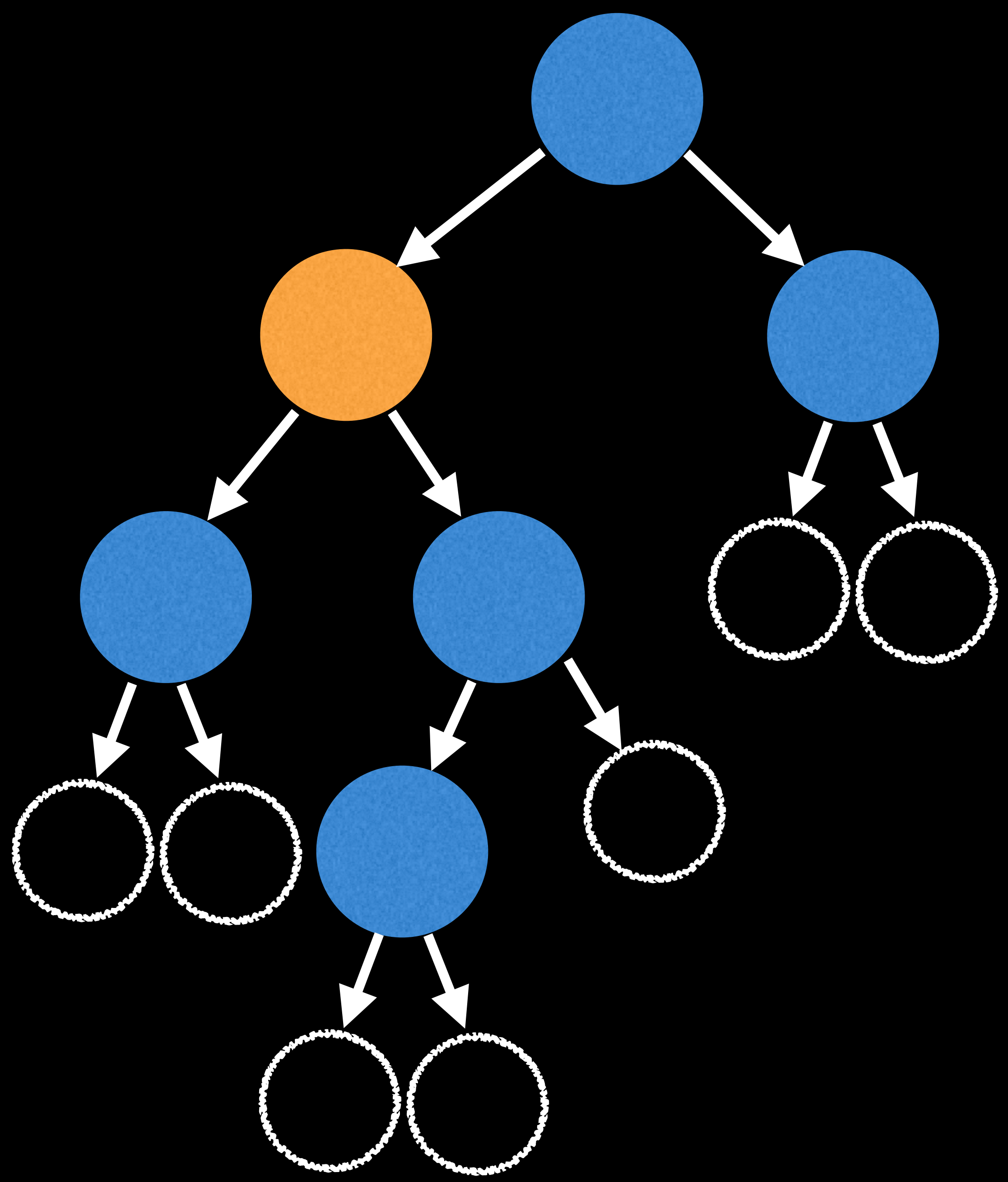
```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Return -1 when we hit a null node
  # to correct for the right height.
  if node == null:
     return -1

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```

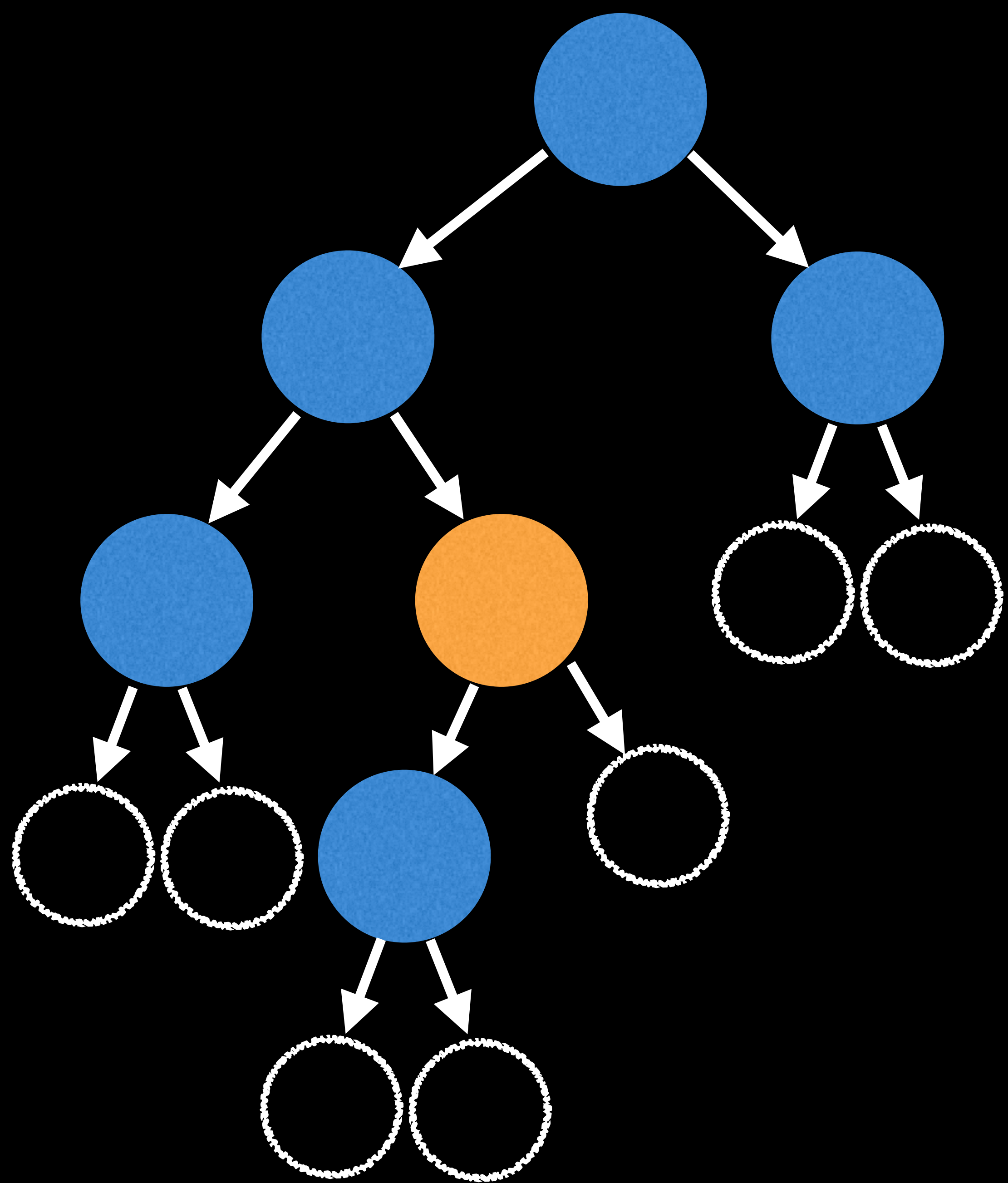
Notice that if we visit the null nodes our tree is one unit taller.

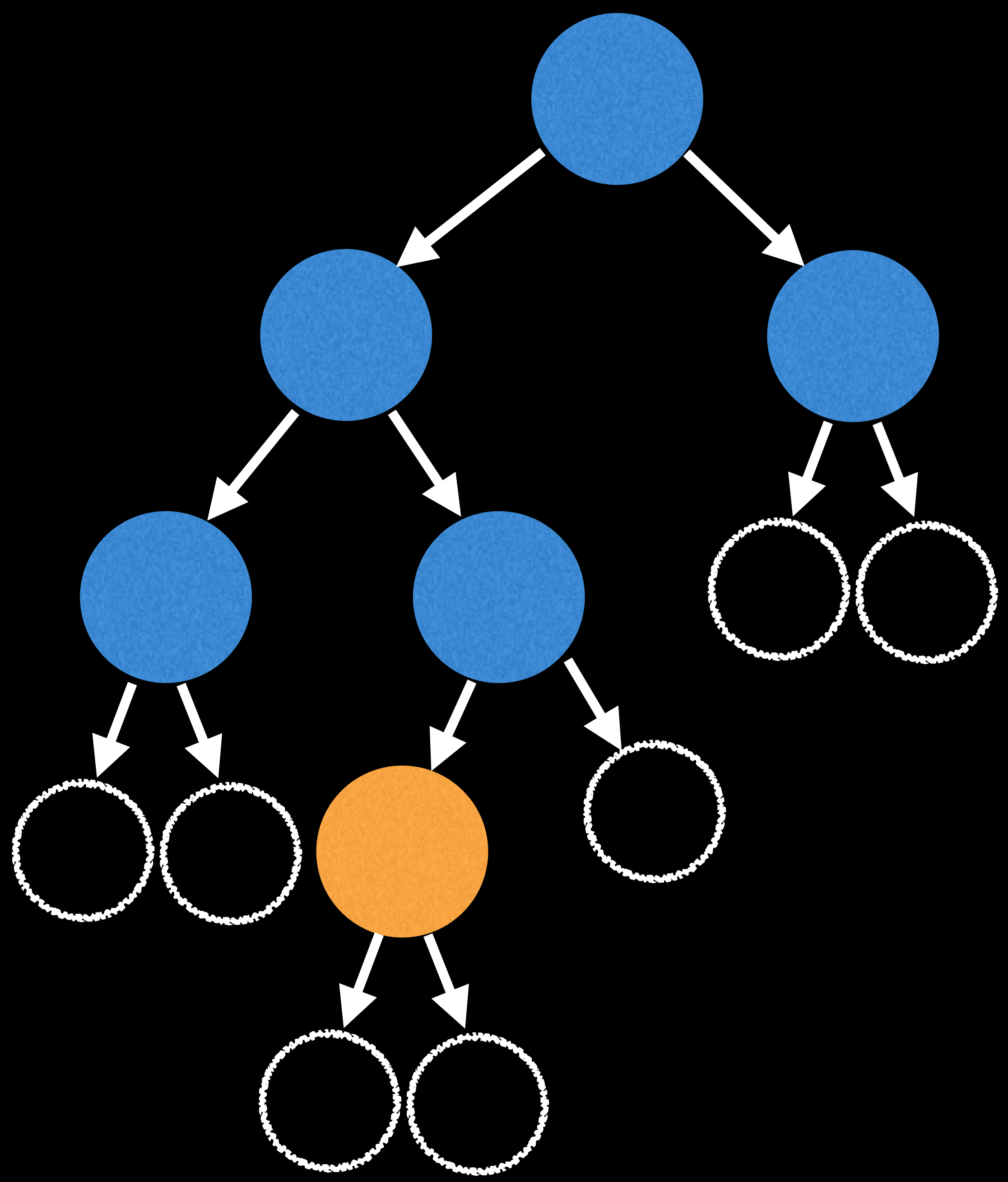


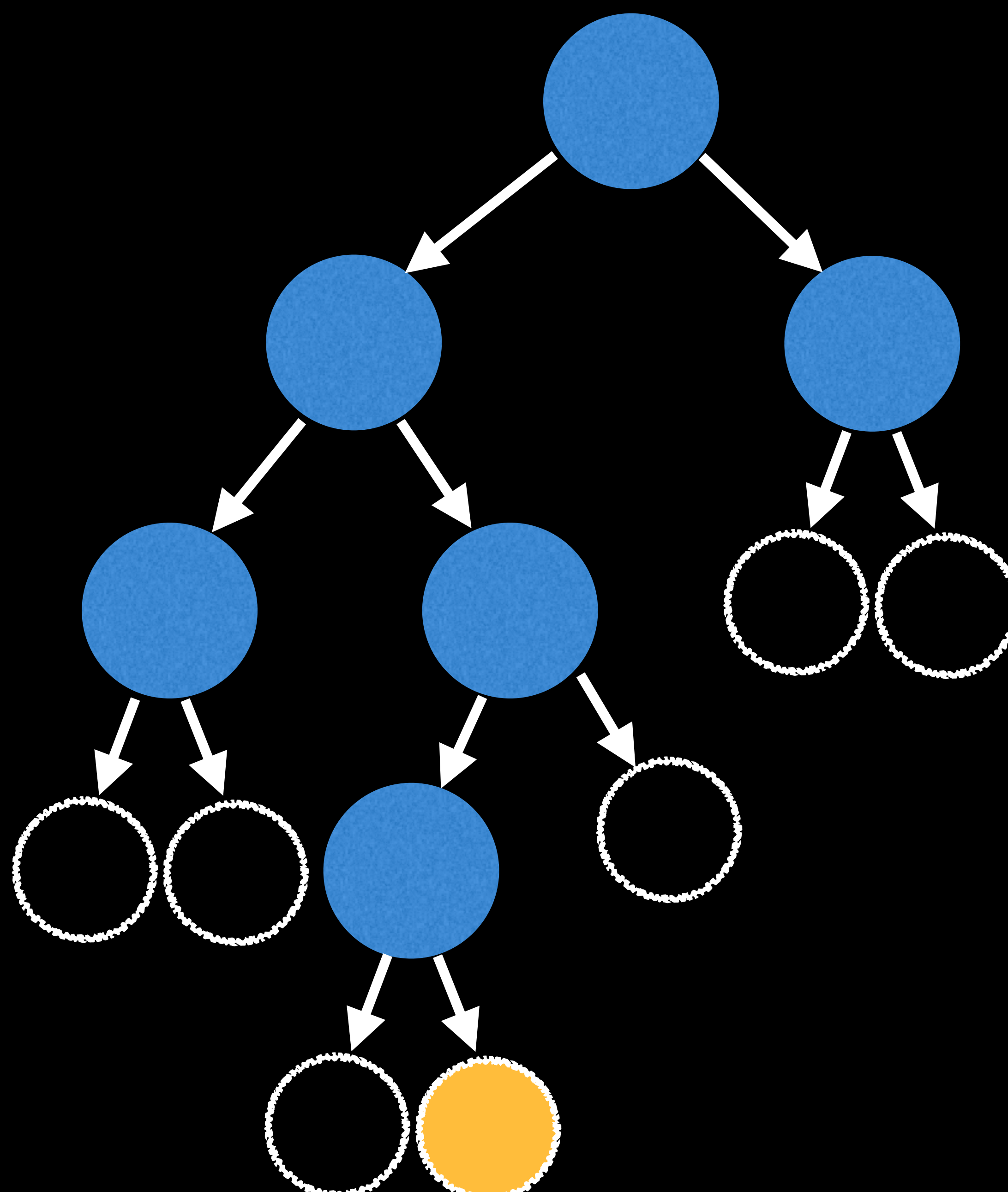
When we go down the tree we need to correct for the height added by the null nodes.

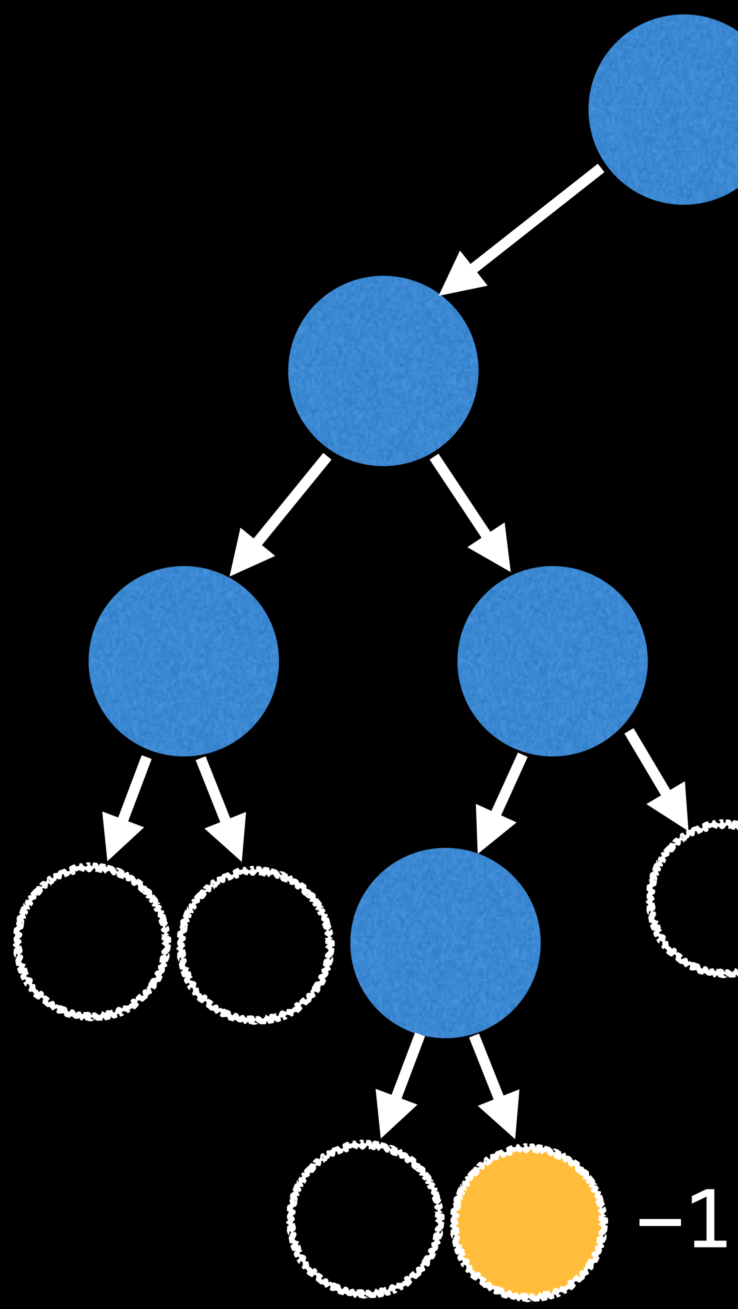


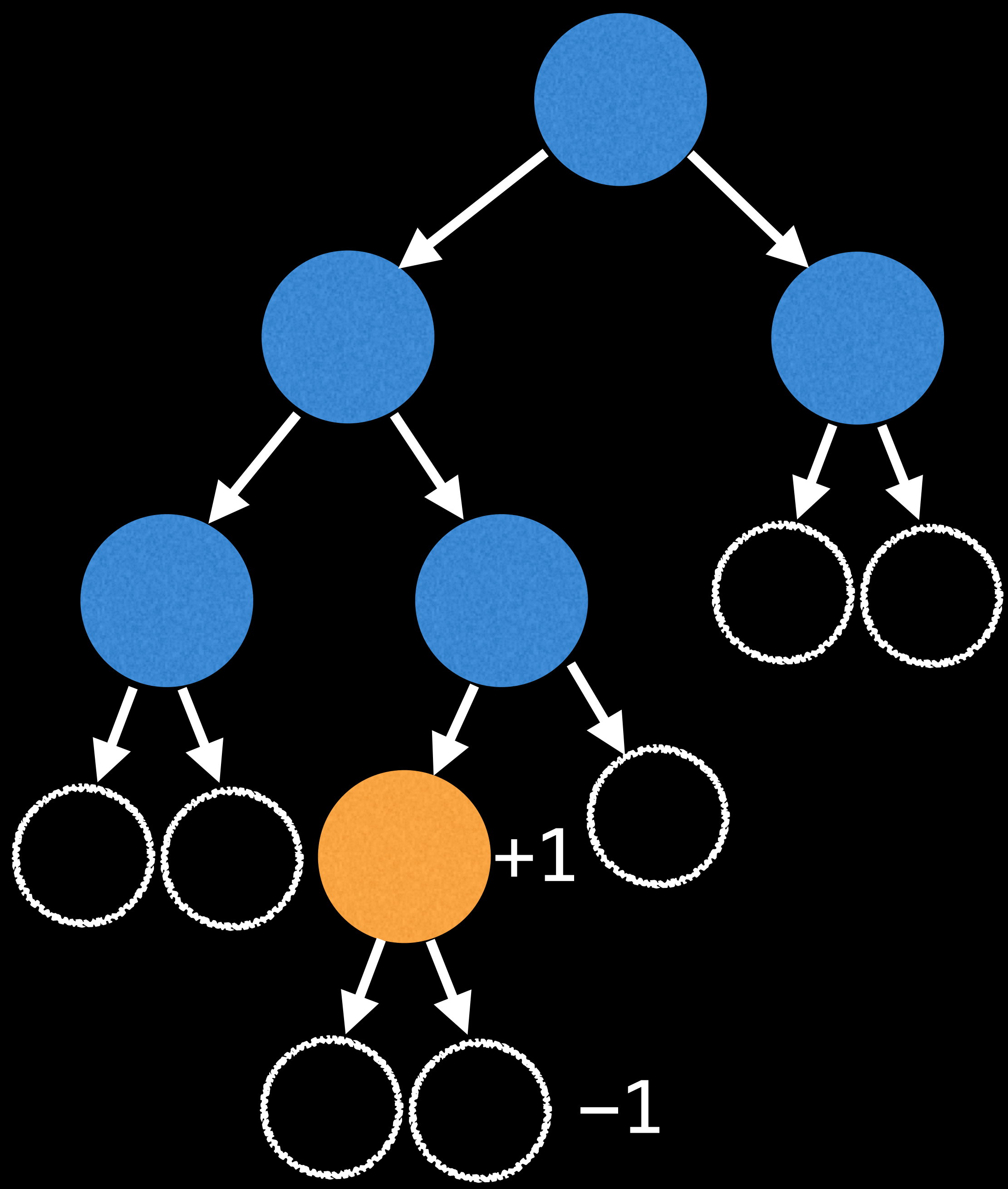


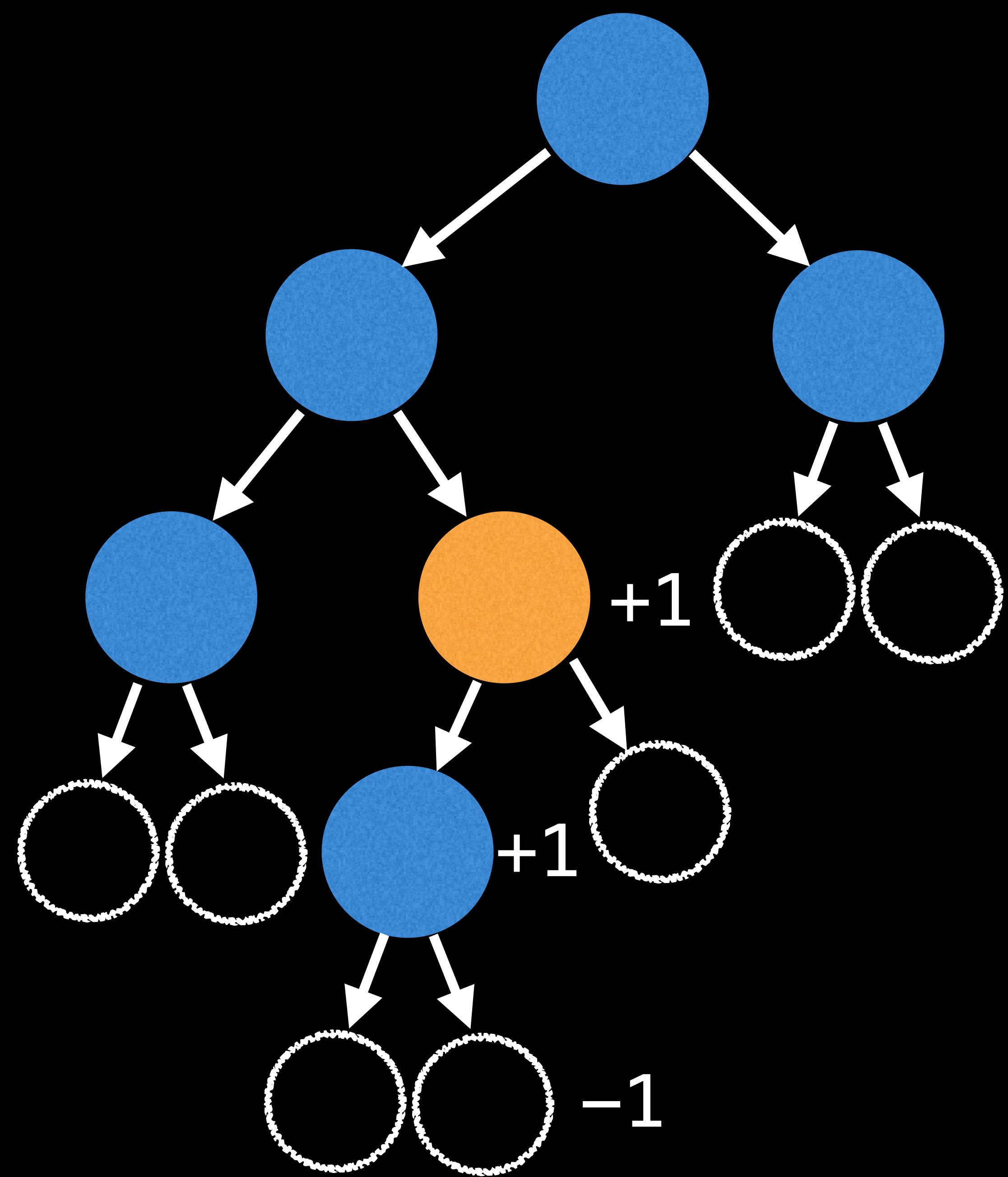


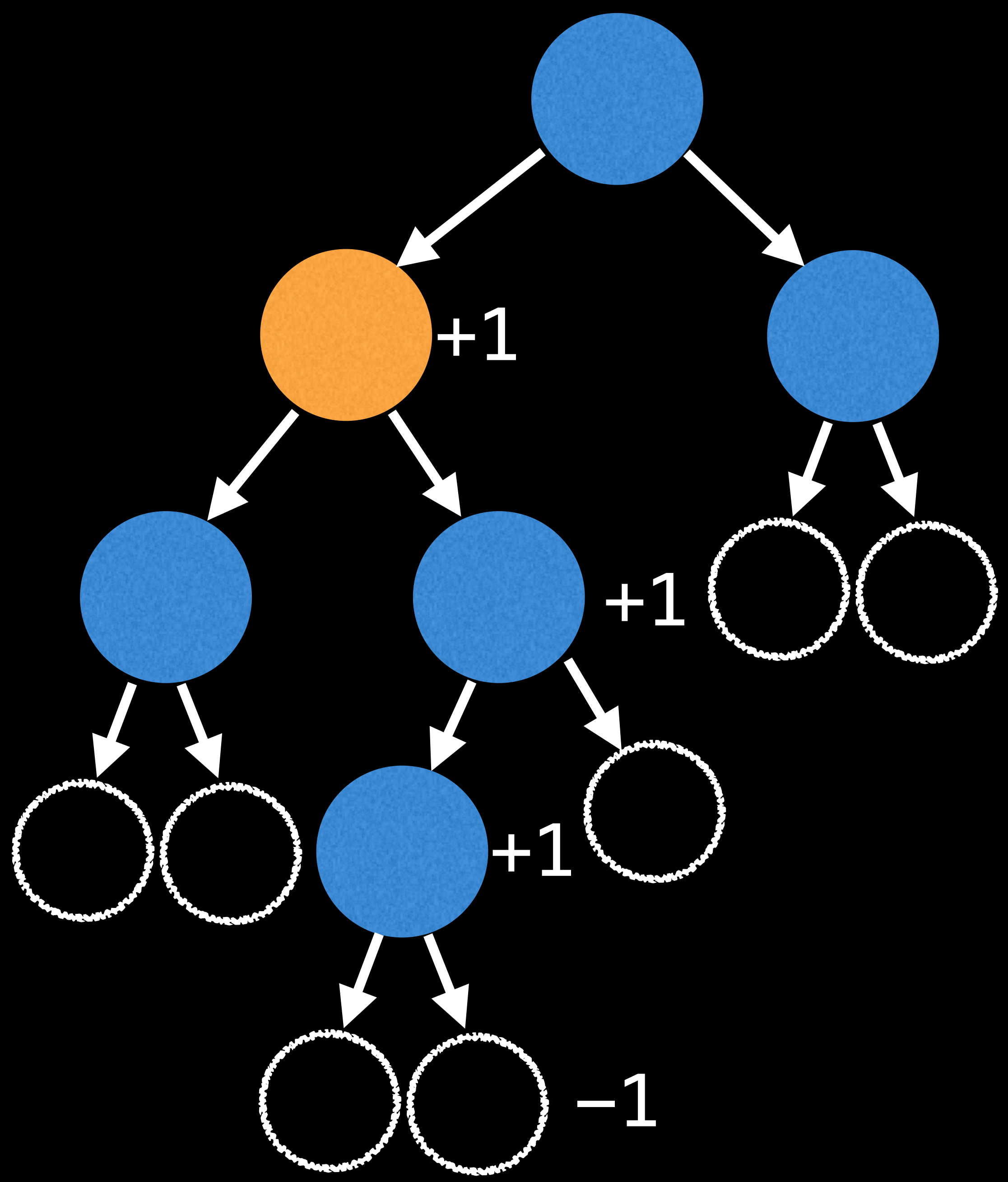


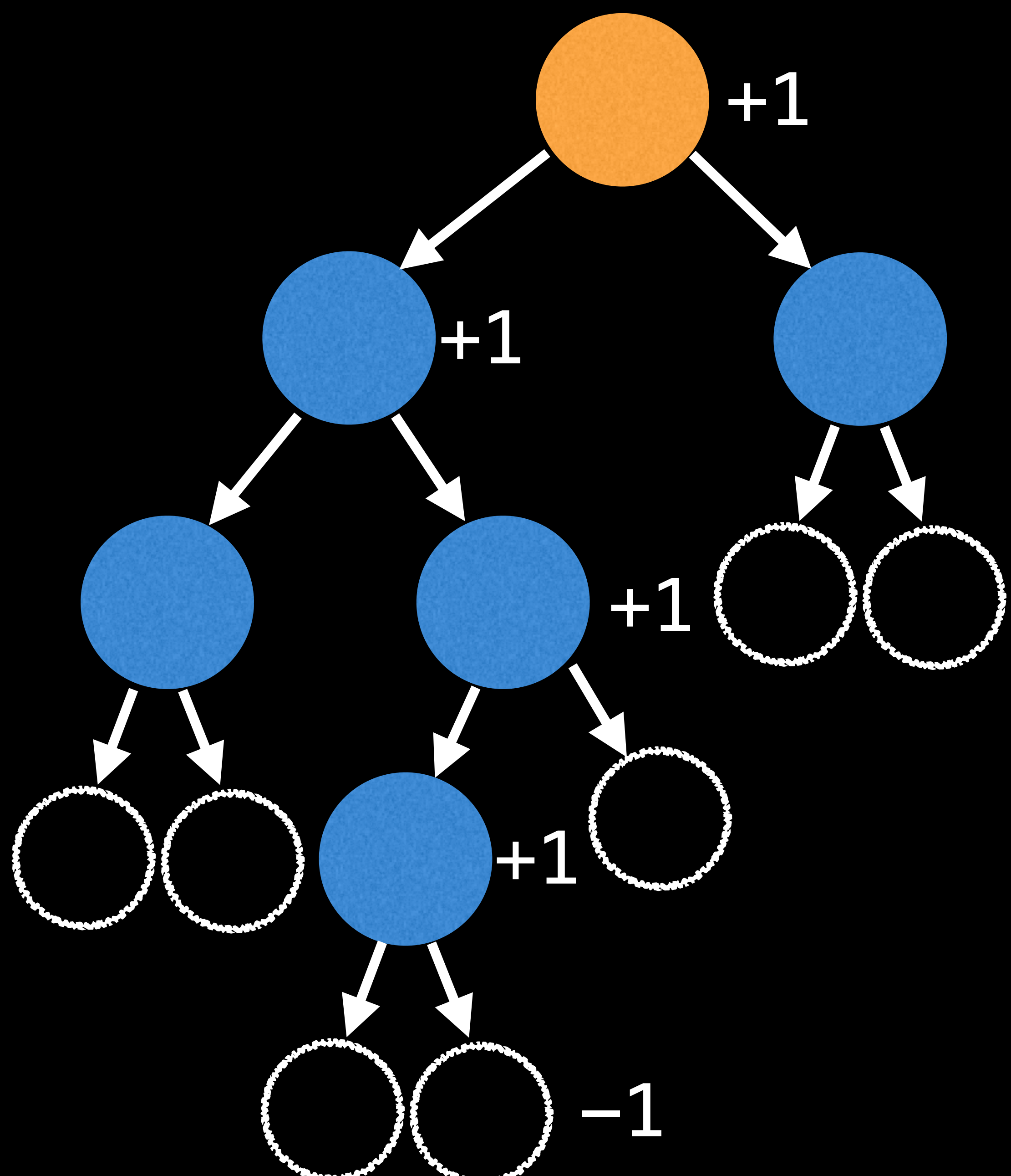


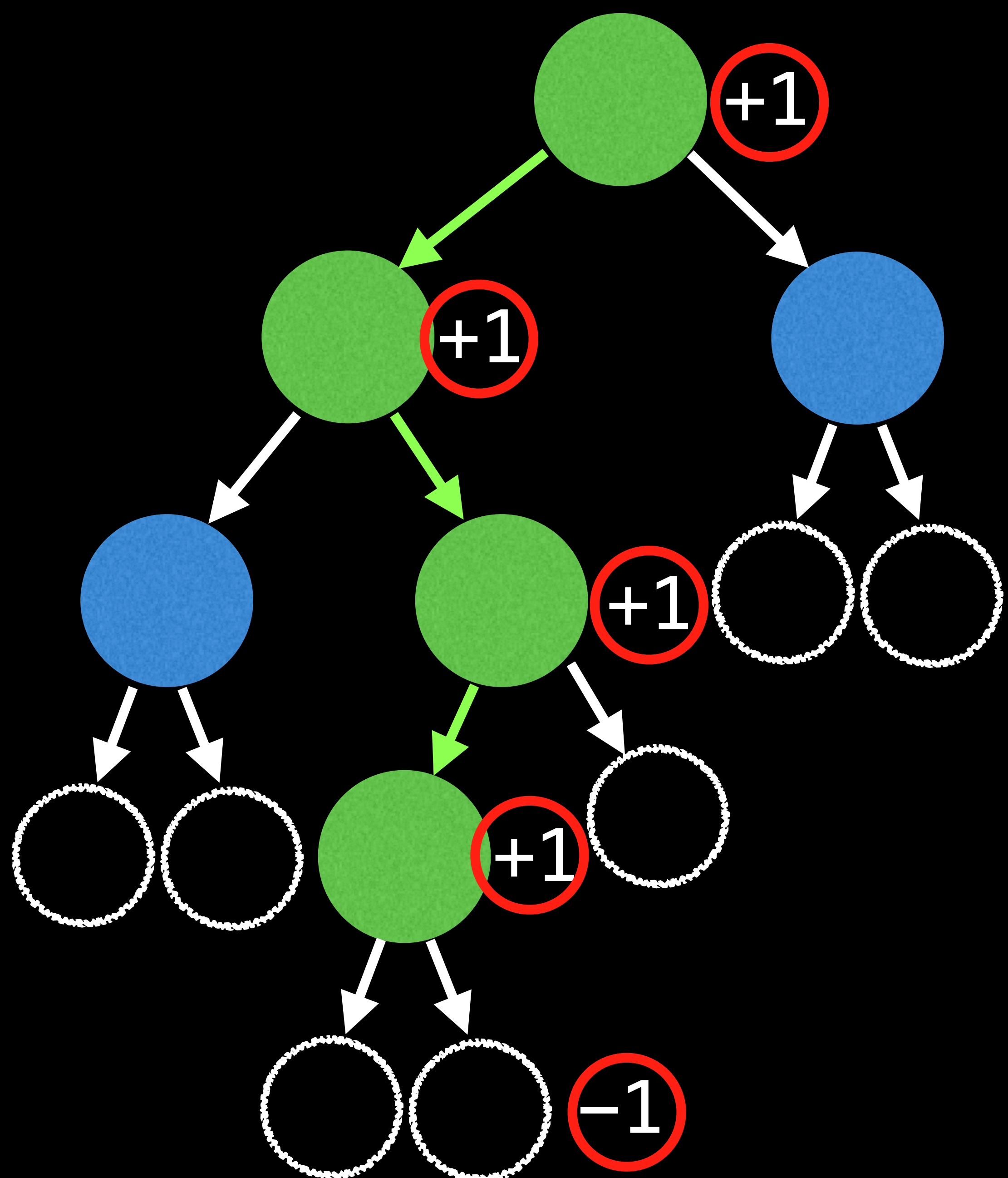

−1


+1
−1


+1
+1
−1


+1
+1
+1
−1


+1
+1
+1
+1
−1



1 + 1 + 1 + 1 − 1 = 3

```
# The height of a tree is the number of
# edges from the root to the lowest leaf.
function treeHeight(node):
  # Return -1 when we hit a null node
  # to correct for the right height.
  if node == null:
     return -1

  return max(treeHeight(node.left),
             treeHeight(node.right)) + 1
```