# CS24 Week 8 Lecture 1

Kyle Dewey

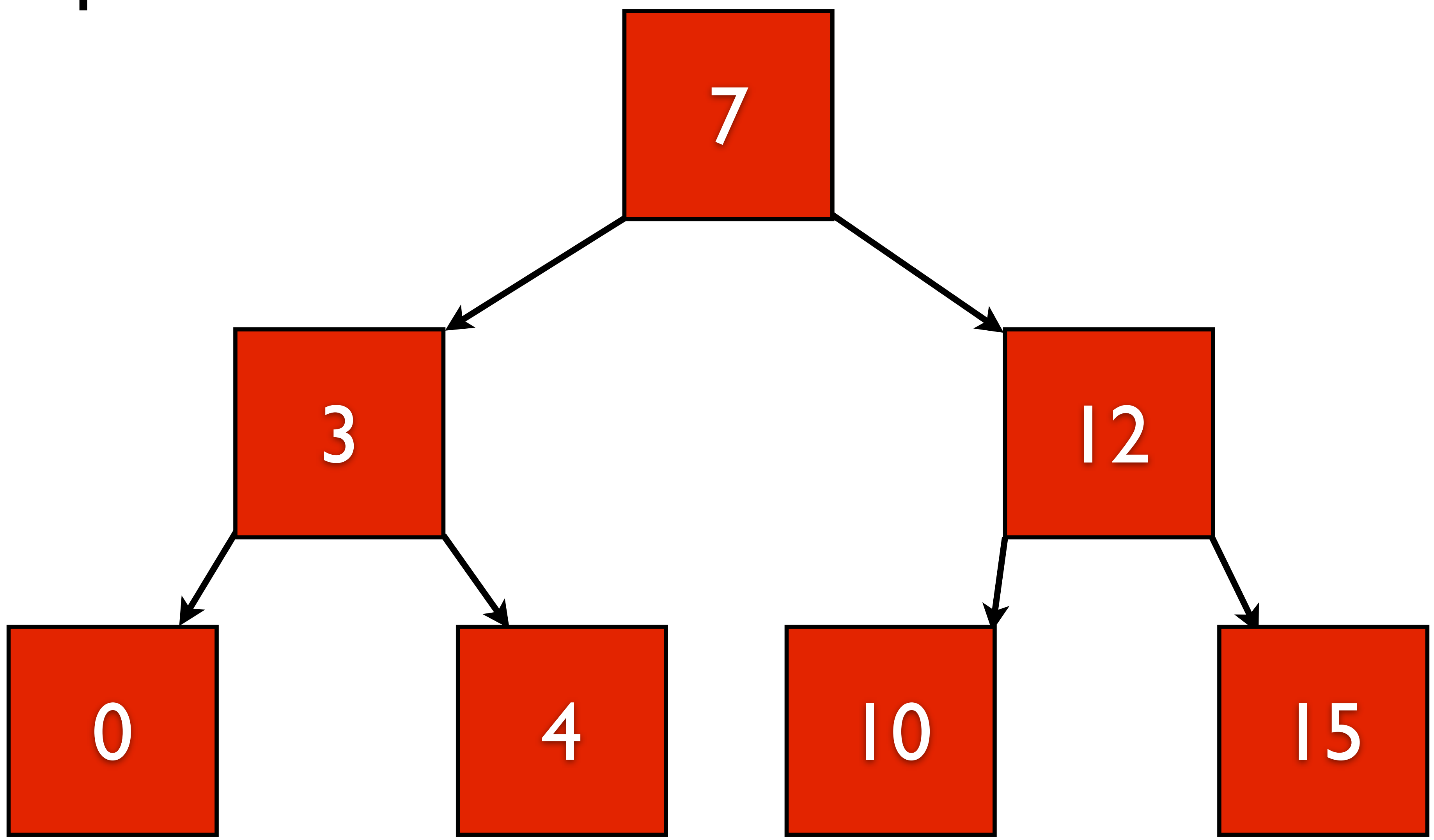
# Overview

- Tree terminology
- Tree traversals
- Implementation (if time)

# Terminology

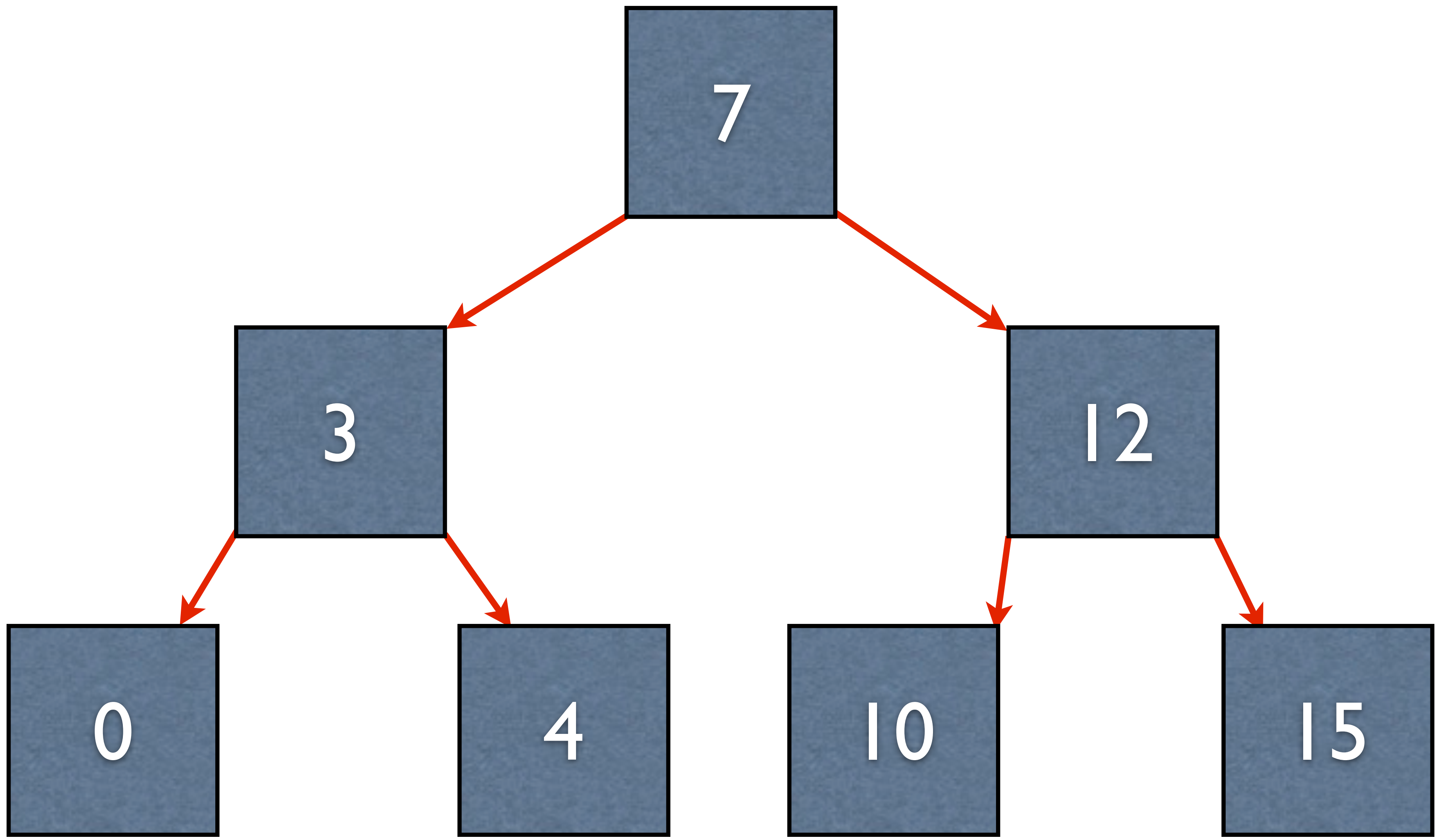
# Node

- The most basic component of a tree - the squares



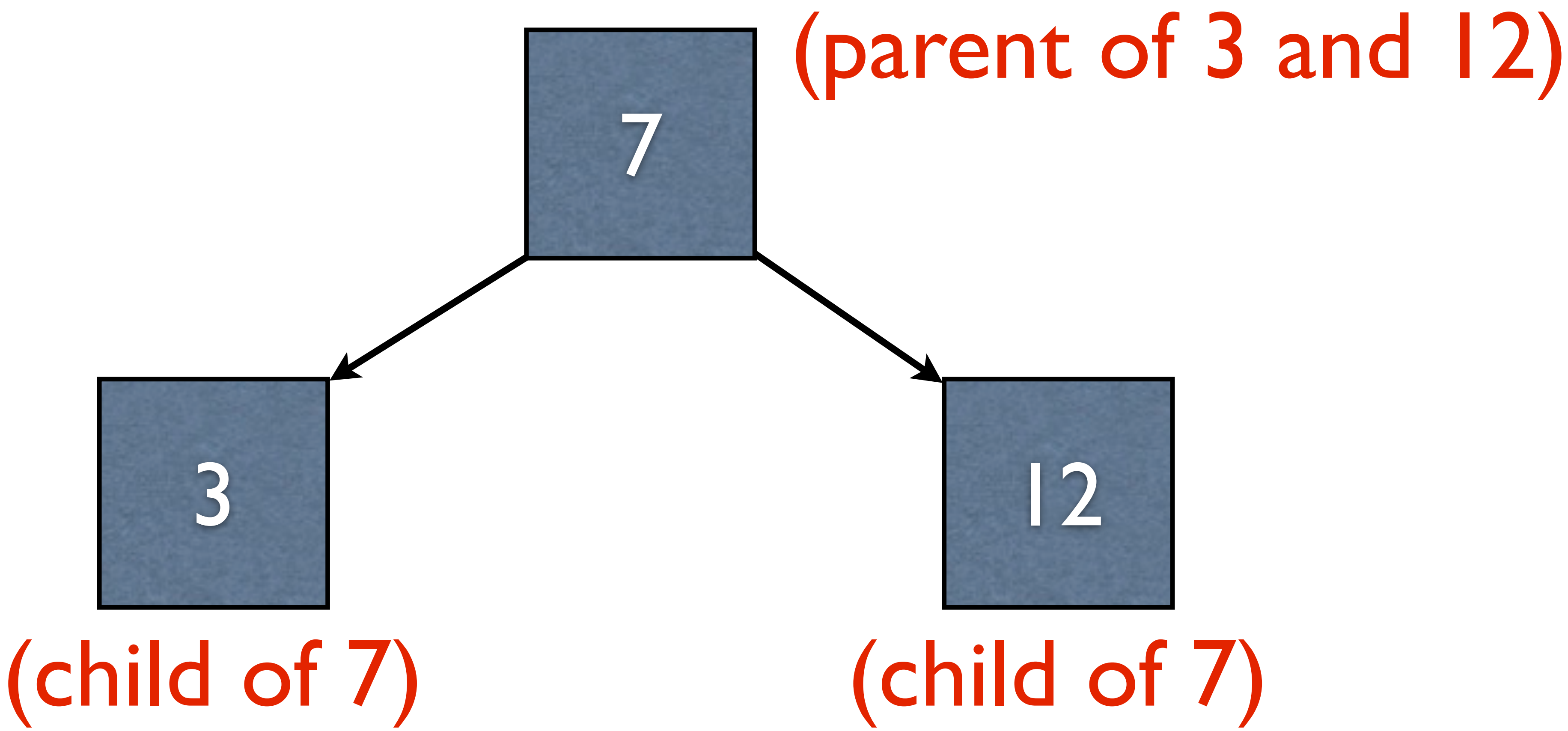
# Edge

- The connections between nodes - the arrows



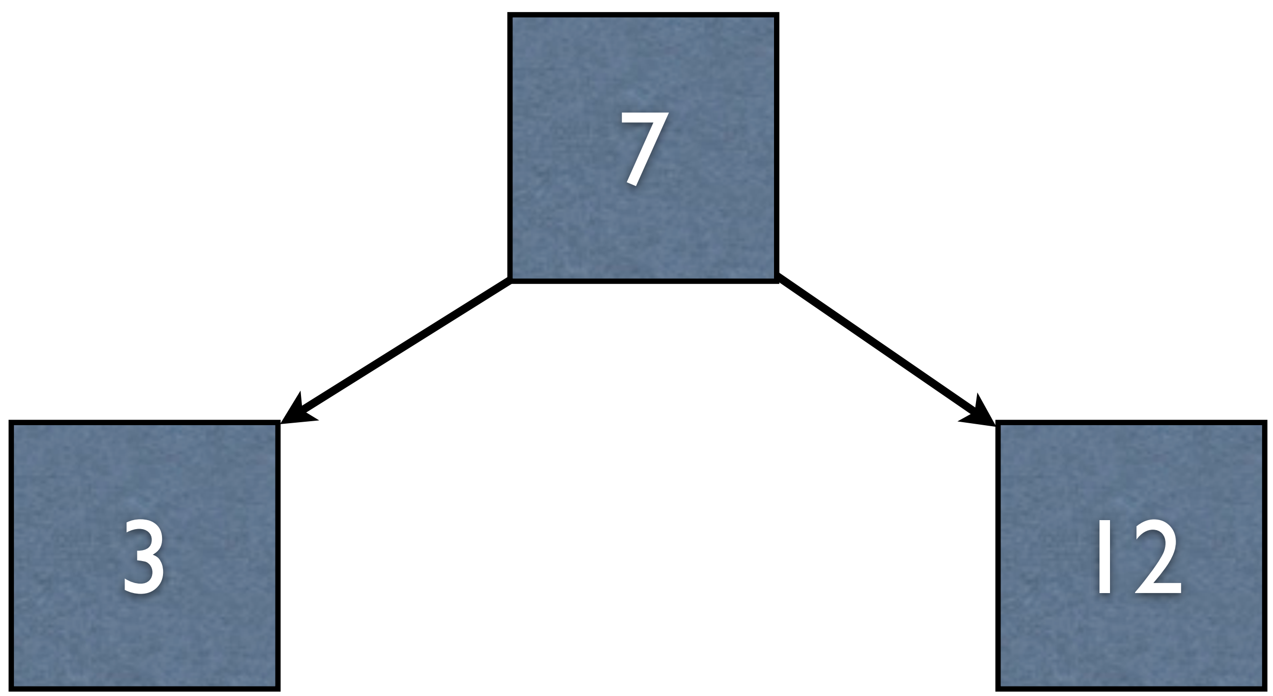
# Parent / Child

- A parent is the predecessor of a node
- A child is the successor of a node
- Not all nodes have parents
- Not all nodes have children



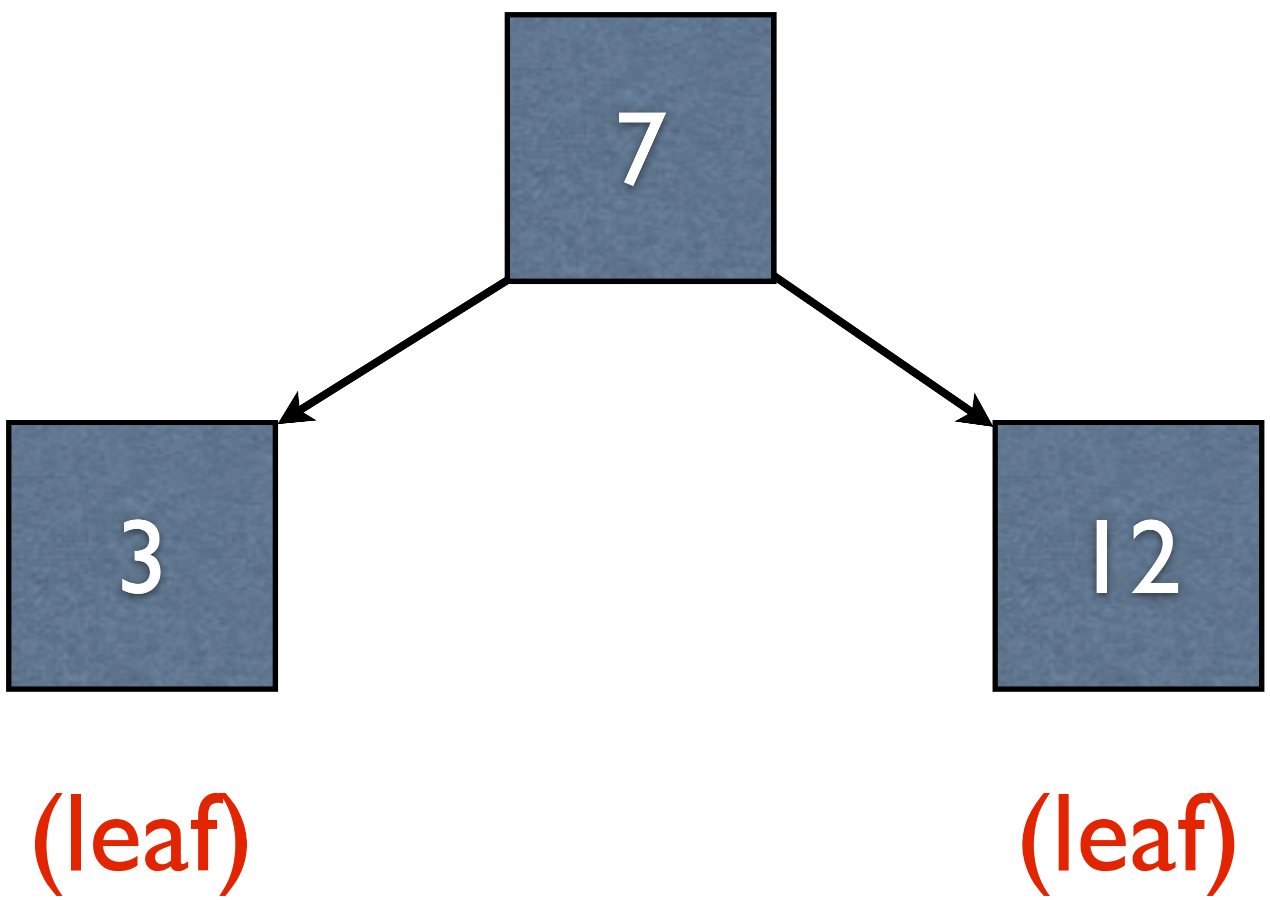
# Leaf / Terminal Node

- A node without any children



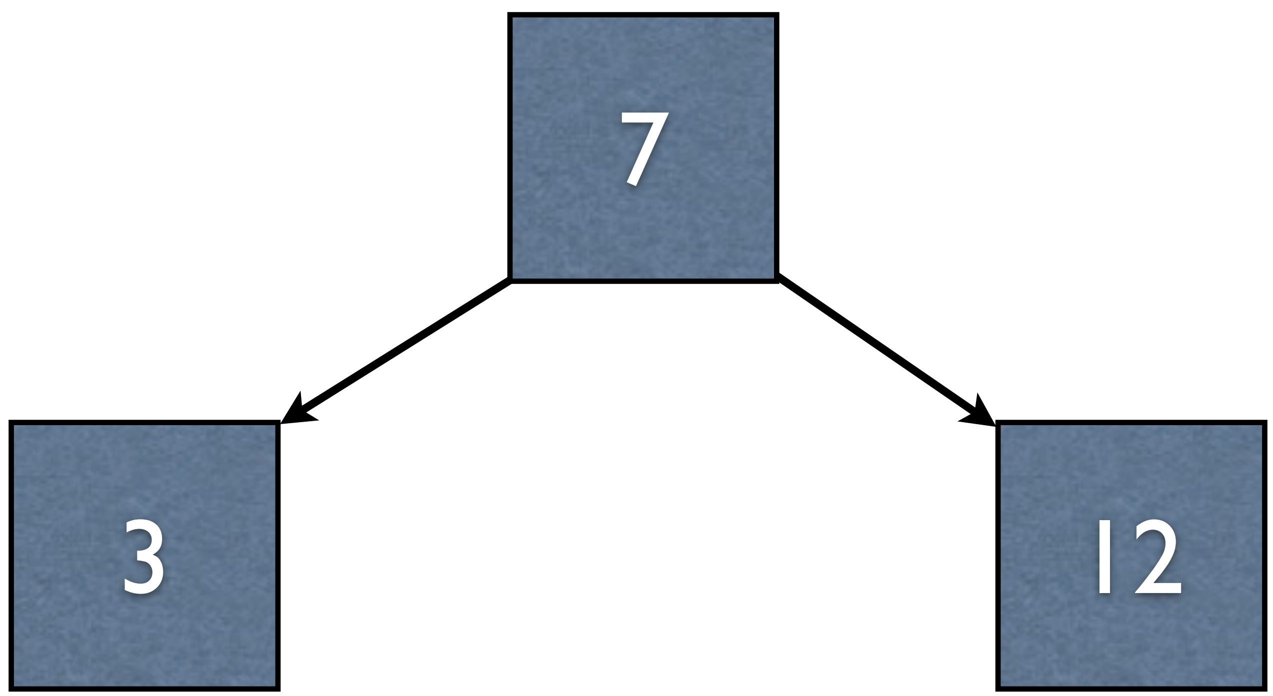
# Leaf / Terminal Node

- A node without any children



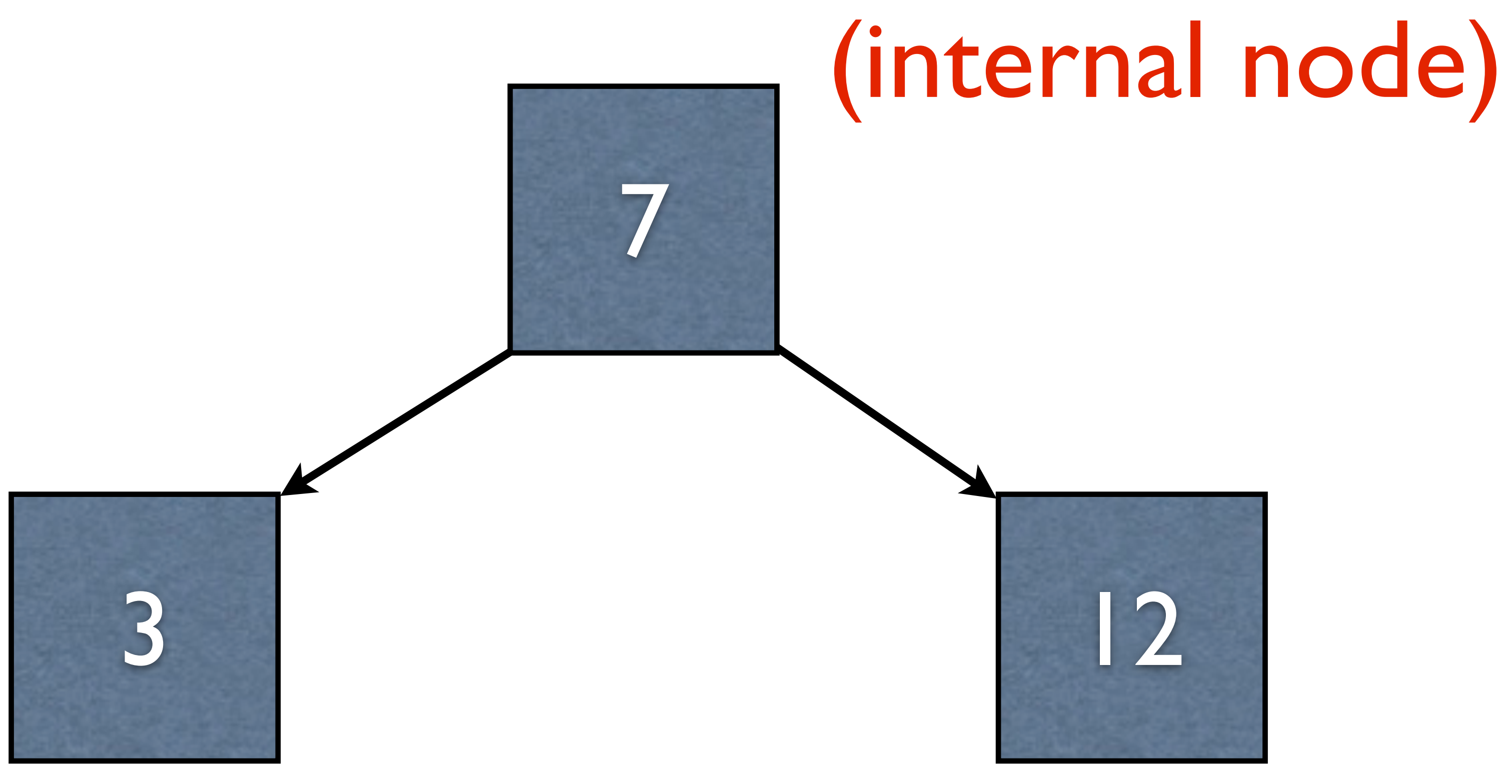
# Internal Node

- A node with at least one child



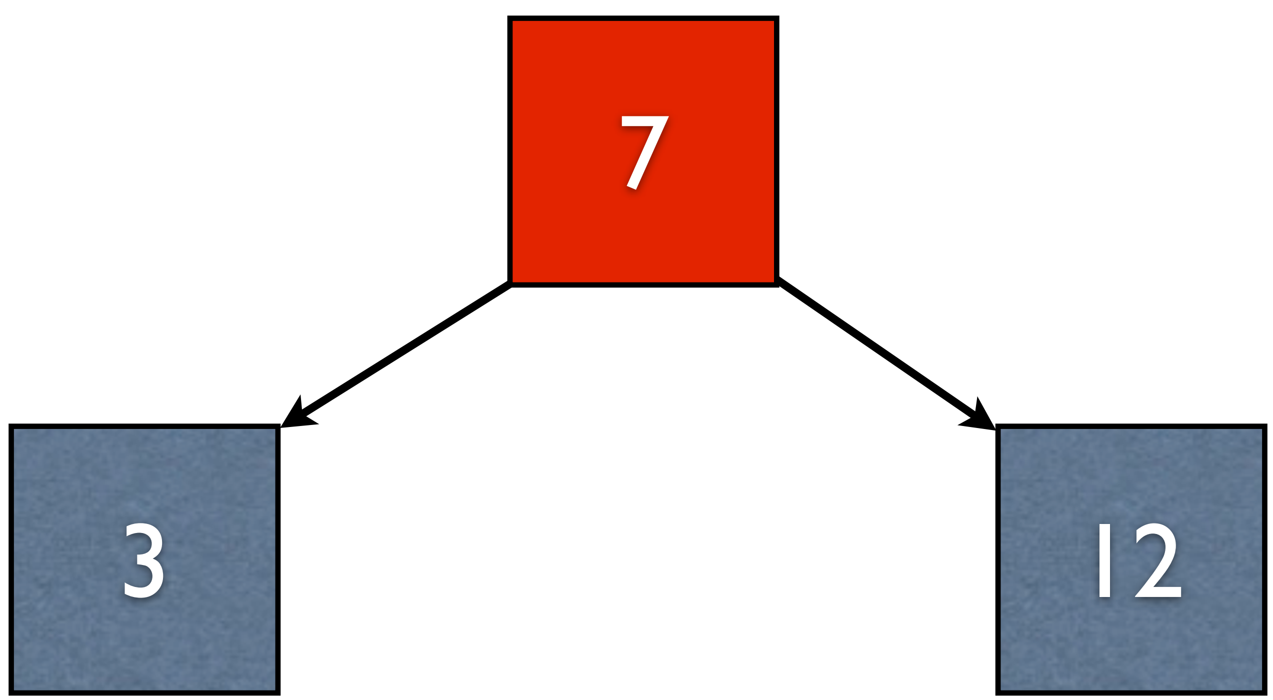
# Internal Node

- A node with at least one child



# Root Node

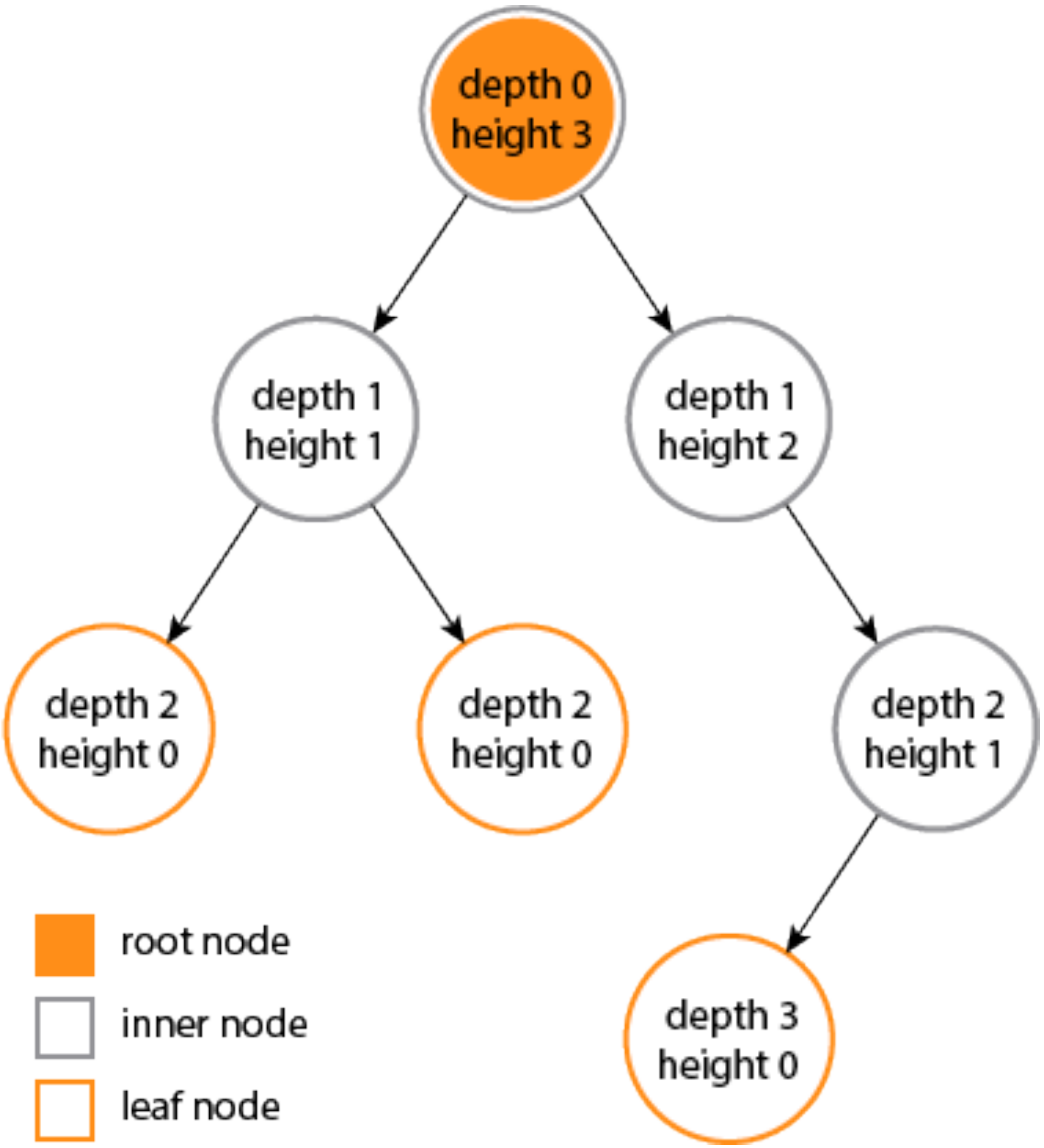
- Node without any parent
- Often drawn as the topmost node



# Height and Depth

- Height: The number of edges on the *longest* path from a node to a leaf
- Depth: the number of edges between a node and the root node

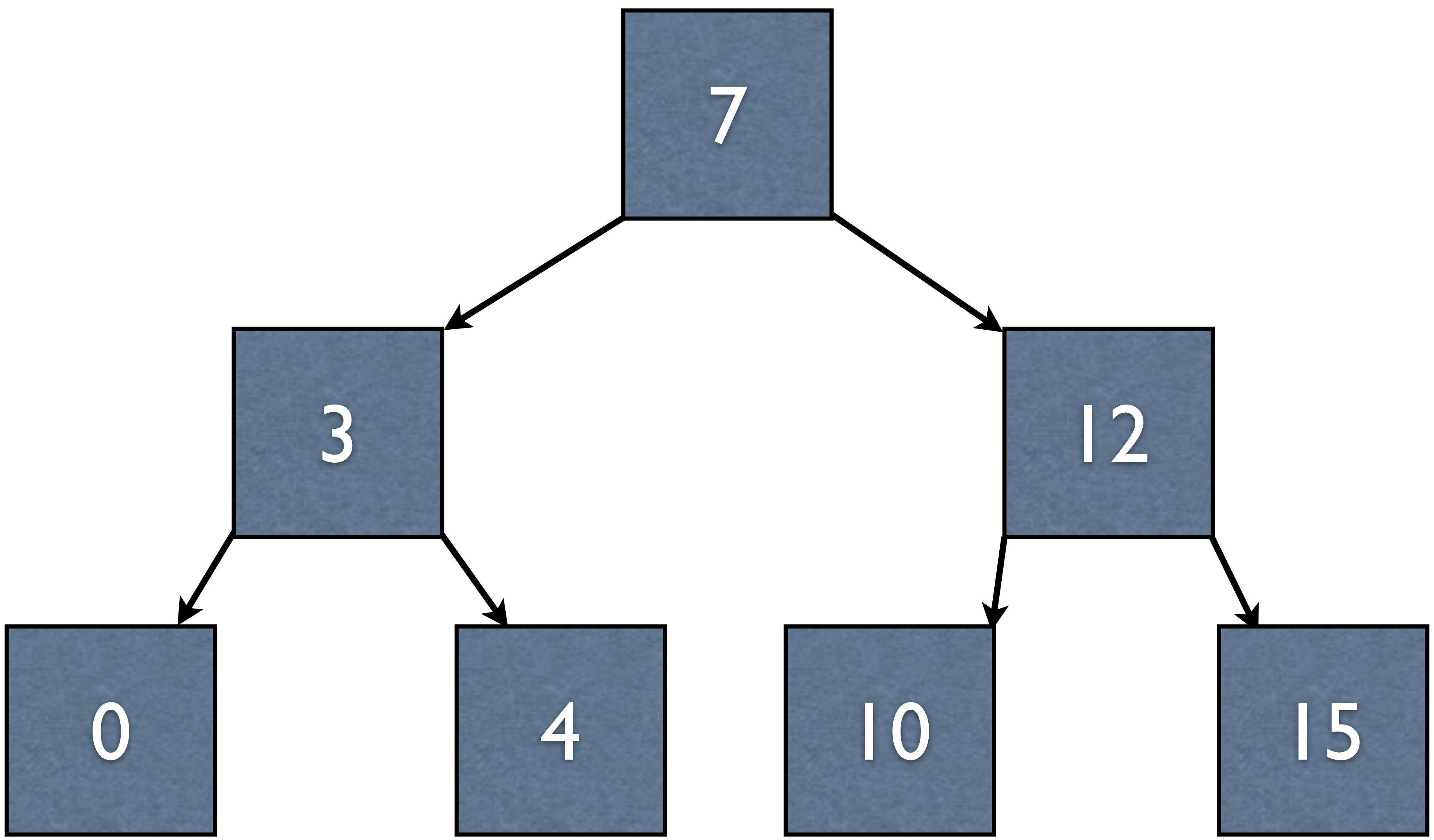
# Height and Depth




http://stackoverflow.com/questions/2603692/what-is-the-difference-between-tree-depth-and-height

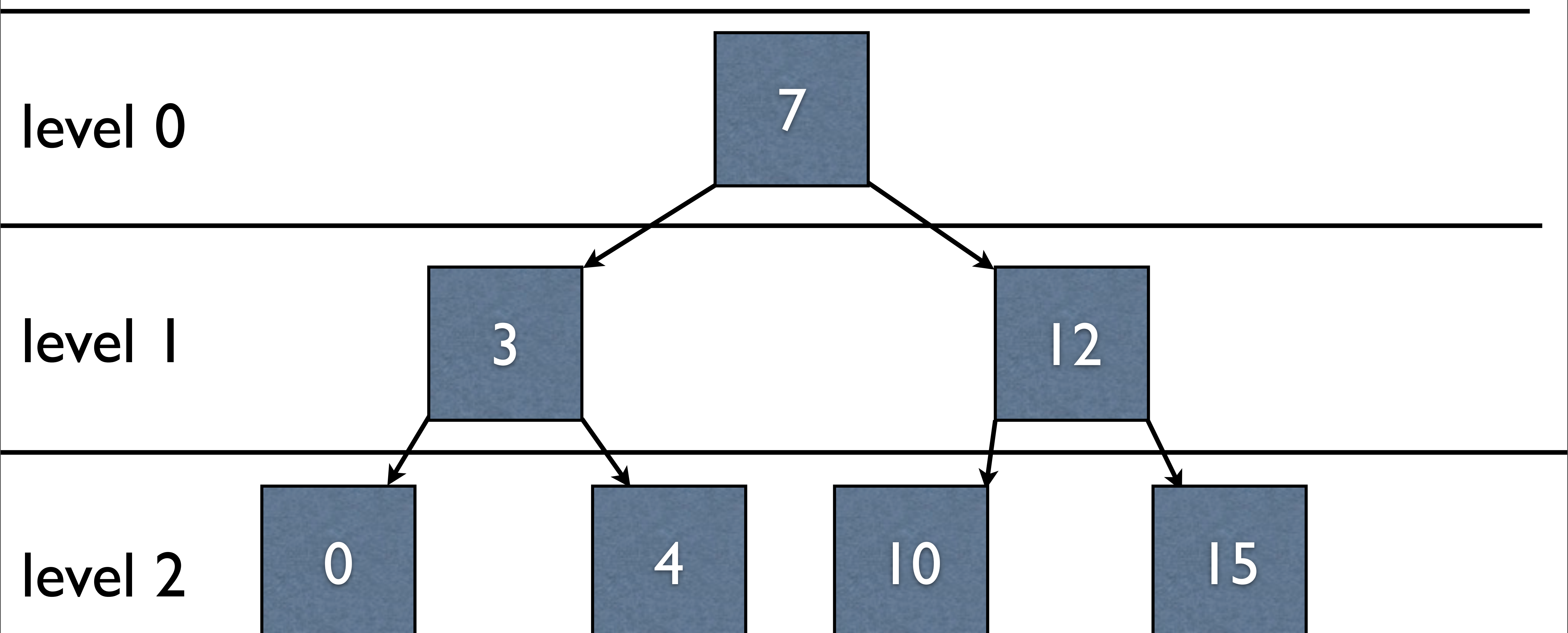
# Level

- All the nodes of a tree which have the same depth



# Level

- All the nodes of a tree which have the same depth



# `k`-ary Tree

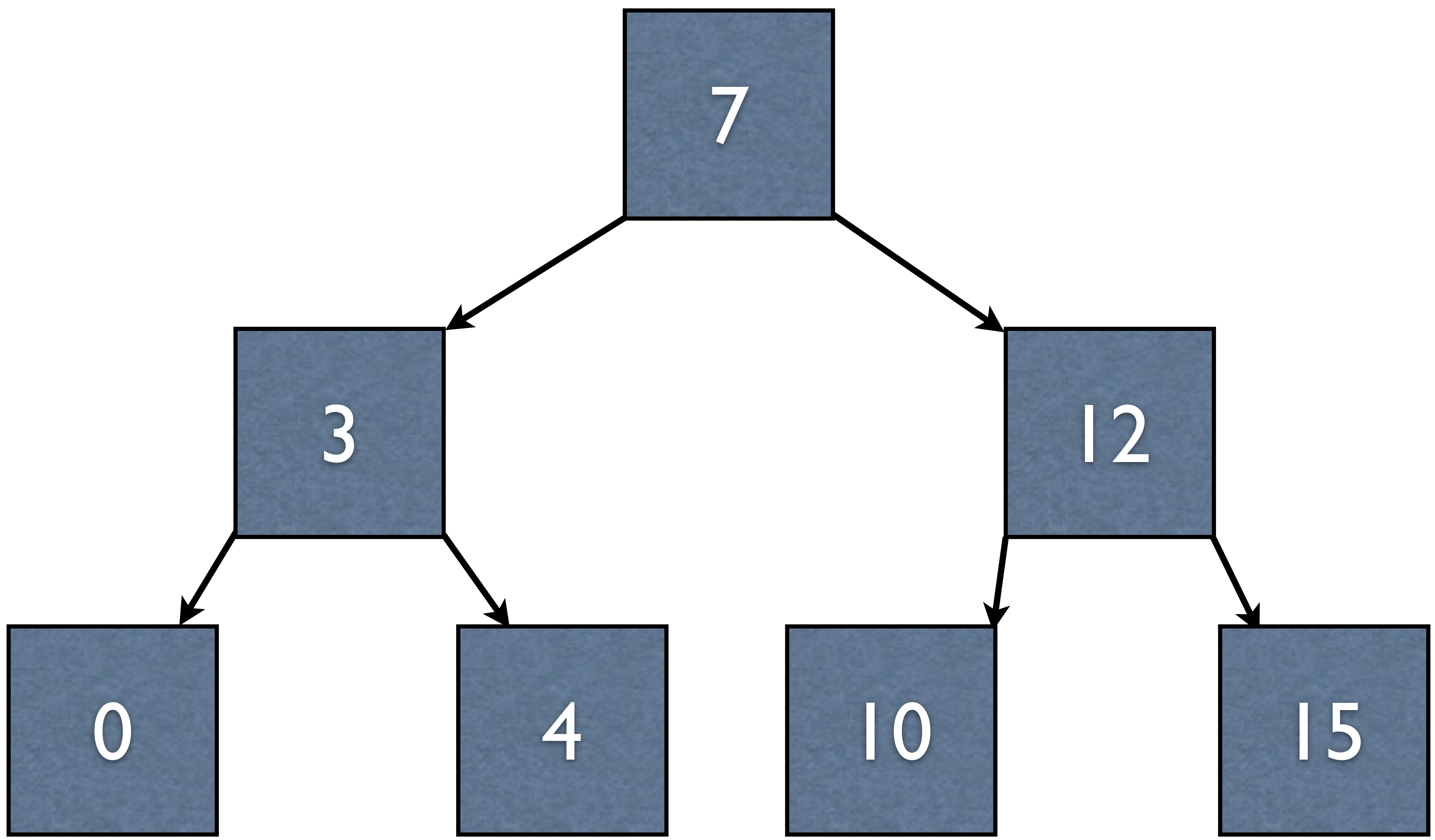
- A tree where each node can have between 0 and k children
  - What is `k` for a binary tree?

# `k`-ary Tree

- A tree where each node can have between 0 and k children
  - What is `k` for a binary tree? - 2

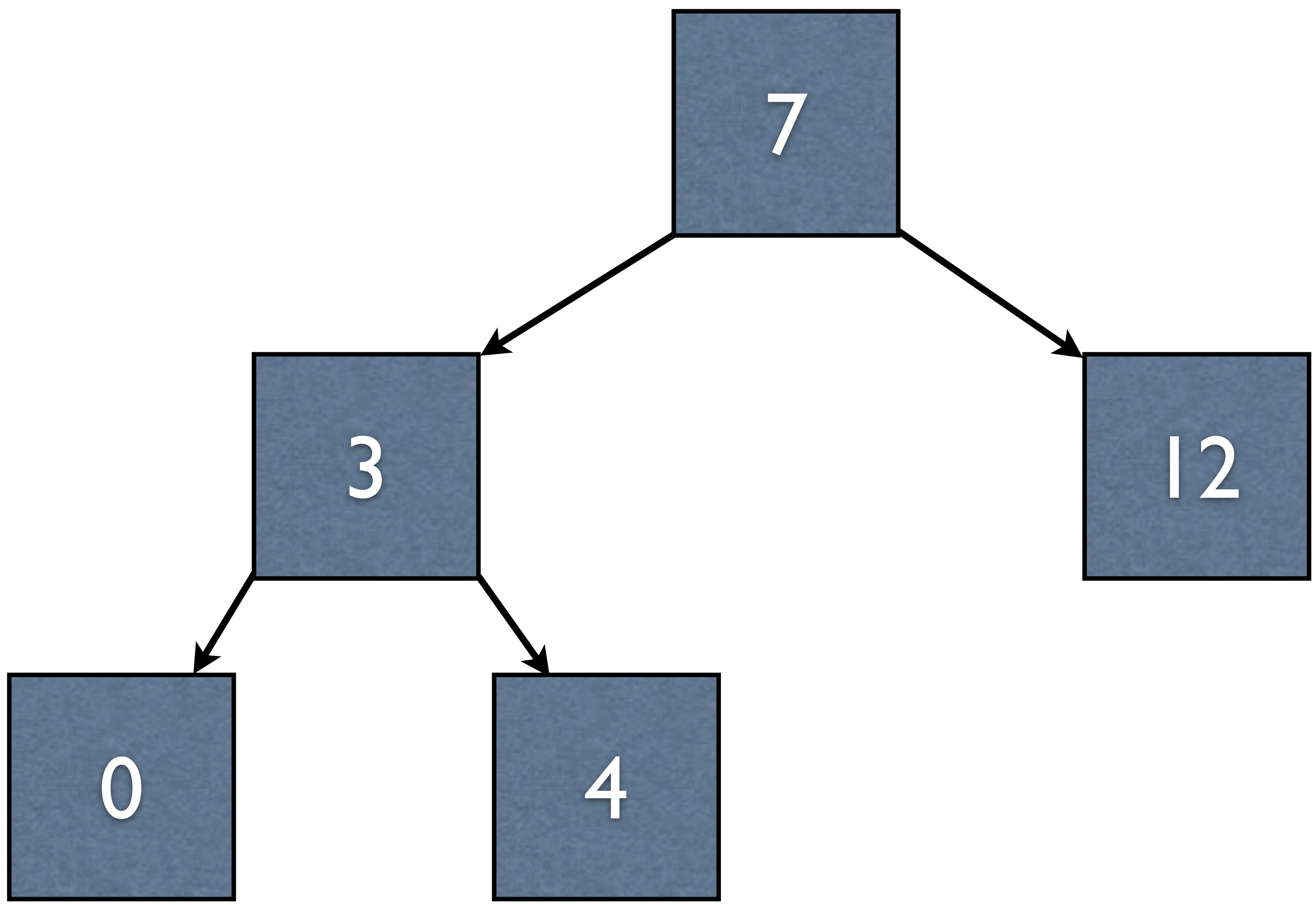
# Full `k`-ary Tree

- All nodes have either 0 or `k` children



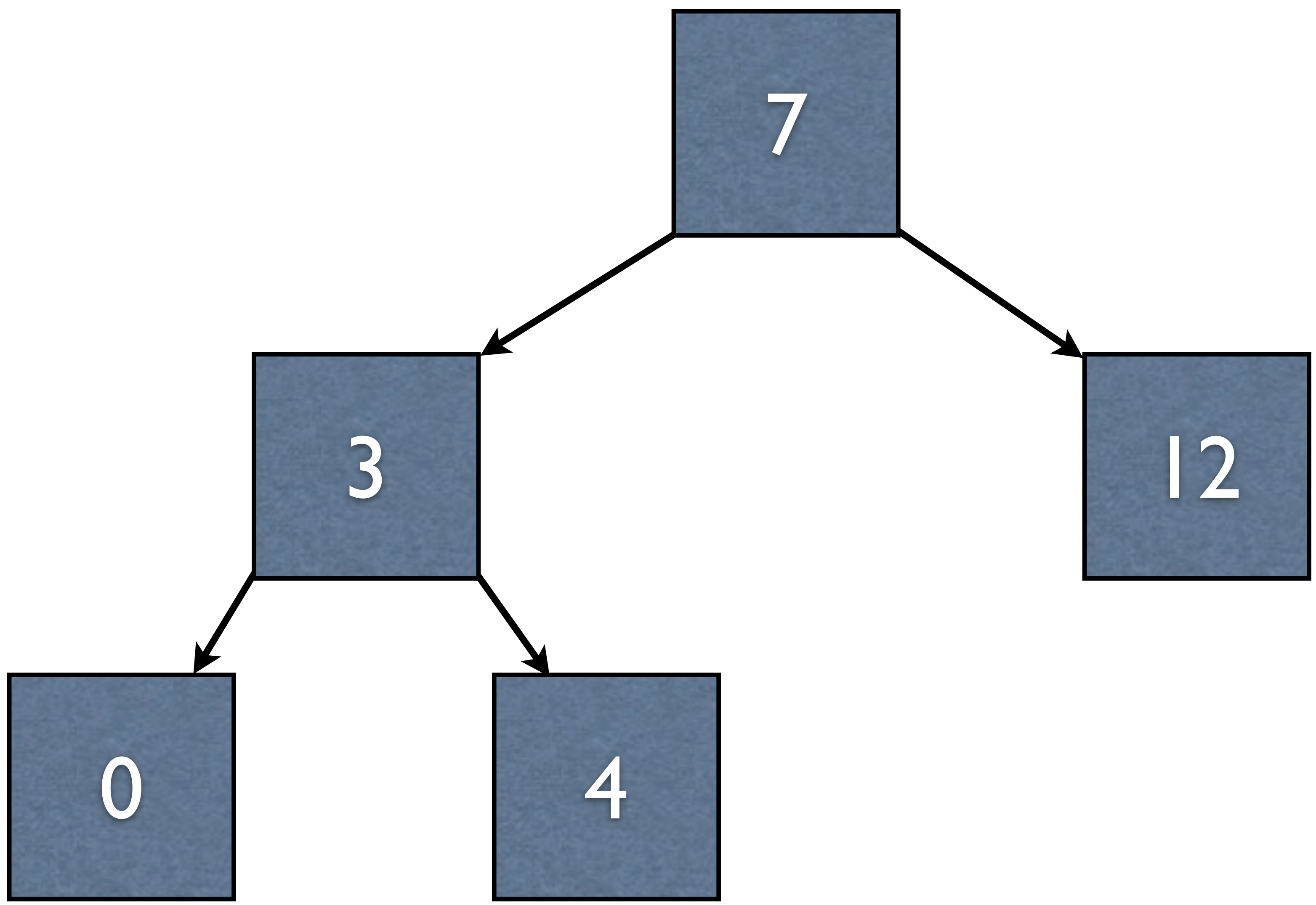
# Complete `k`-ary Tree

- Like a full `k`-ary tree, except the last level is permitted to be missing nodes, but **only on the right**



# Complete `k`-ary Tree

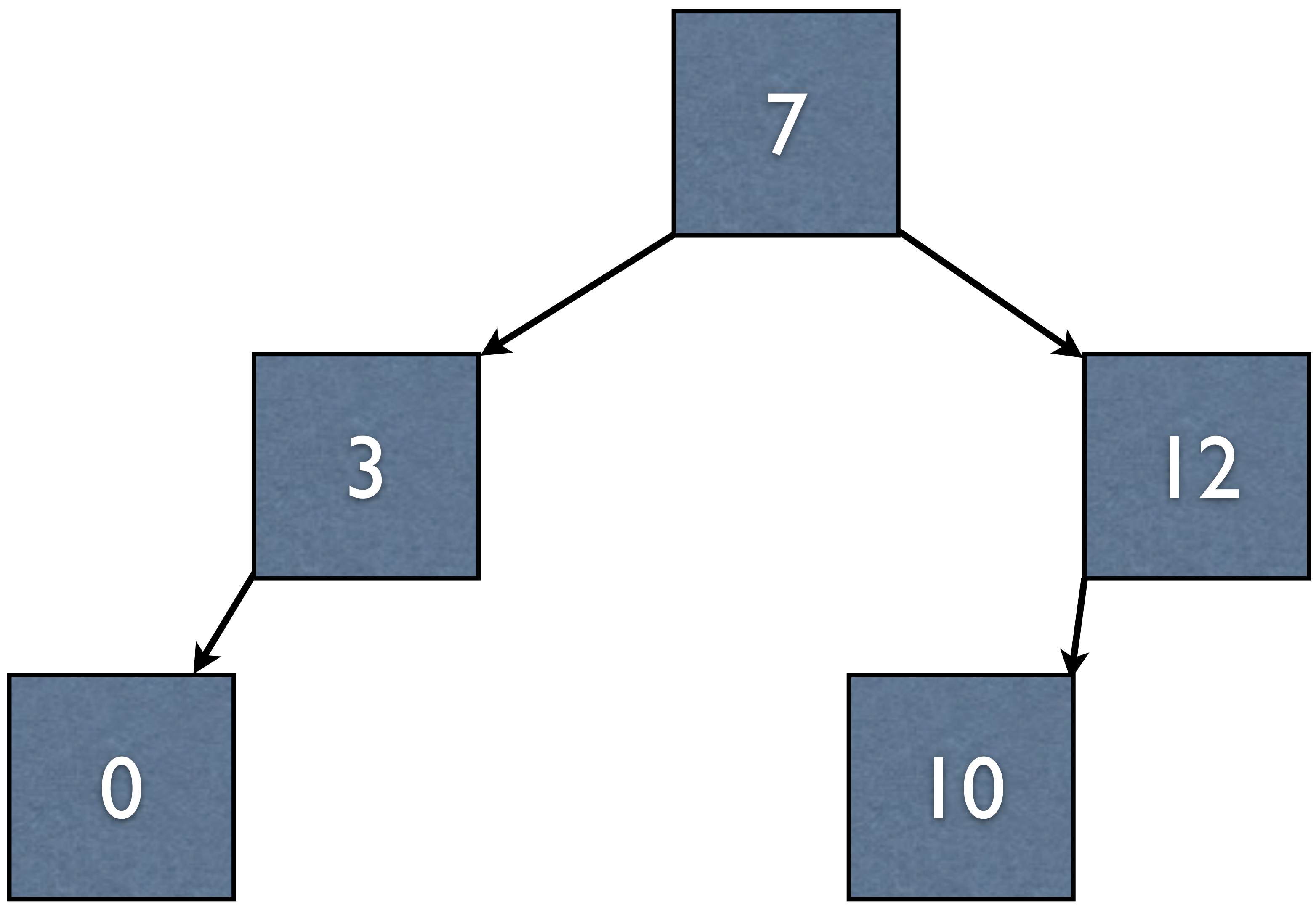
- Like a full `k`-ary tree, except the last level is permitted to be missing nodes, but **only on the right**




ok

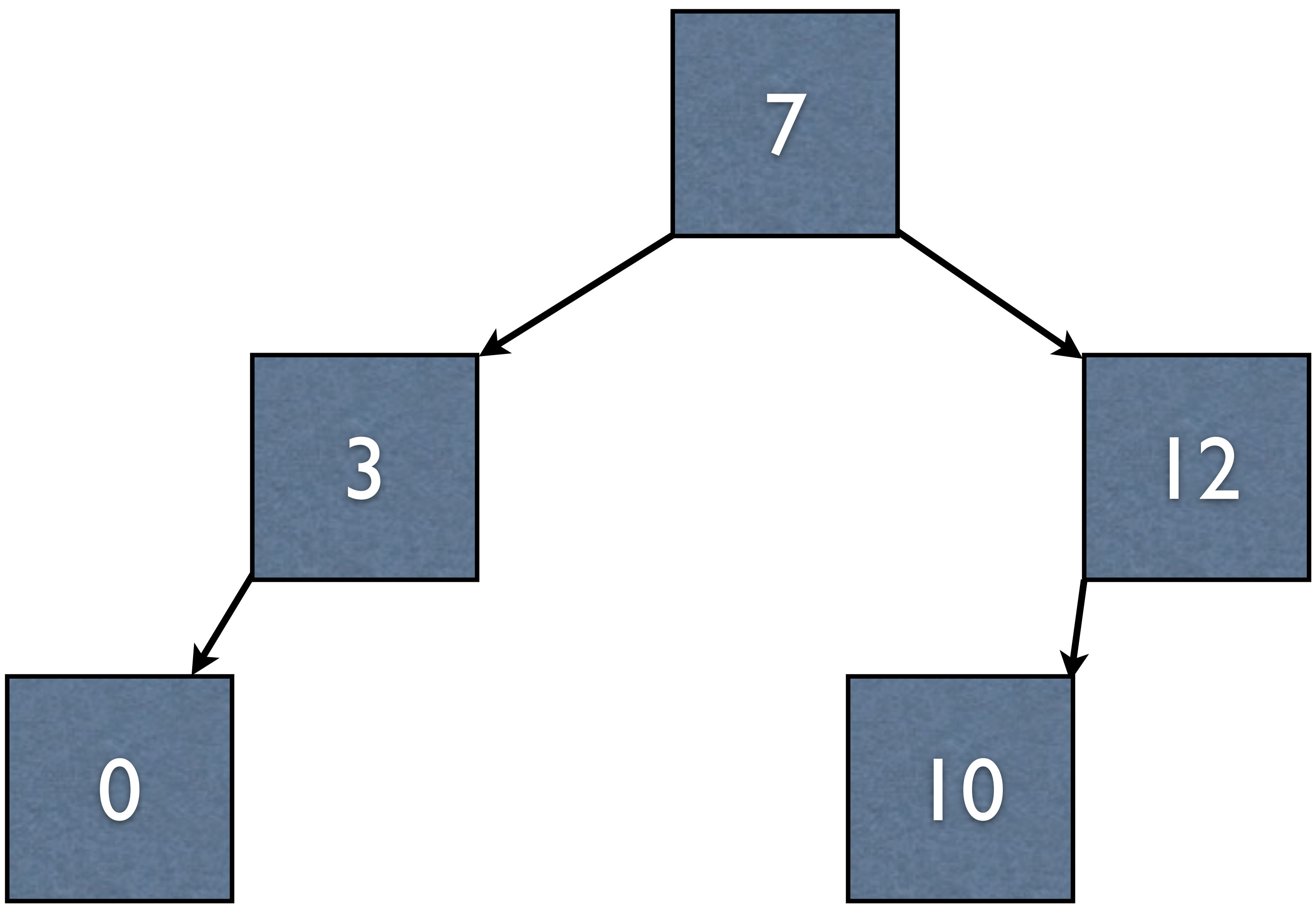
# Complete `k`-ary Tree

- Like a full `k`-ary tree, except the last level is permitted to be missing nodes, but **only on the right**



# Complete `k`-ary Tree

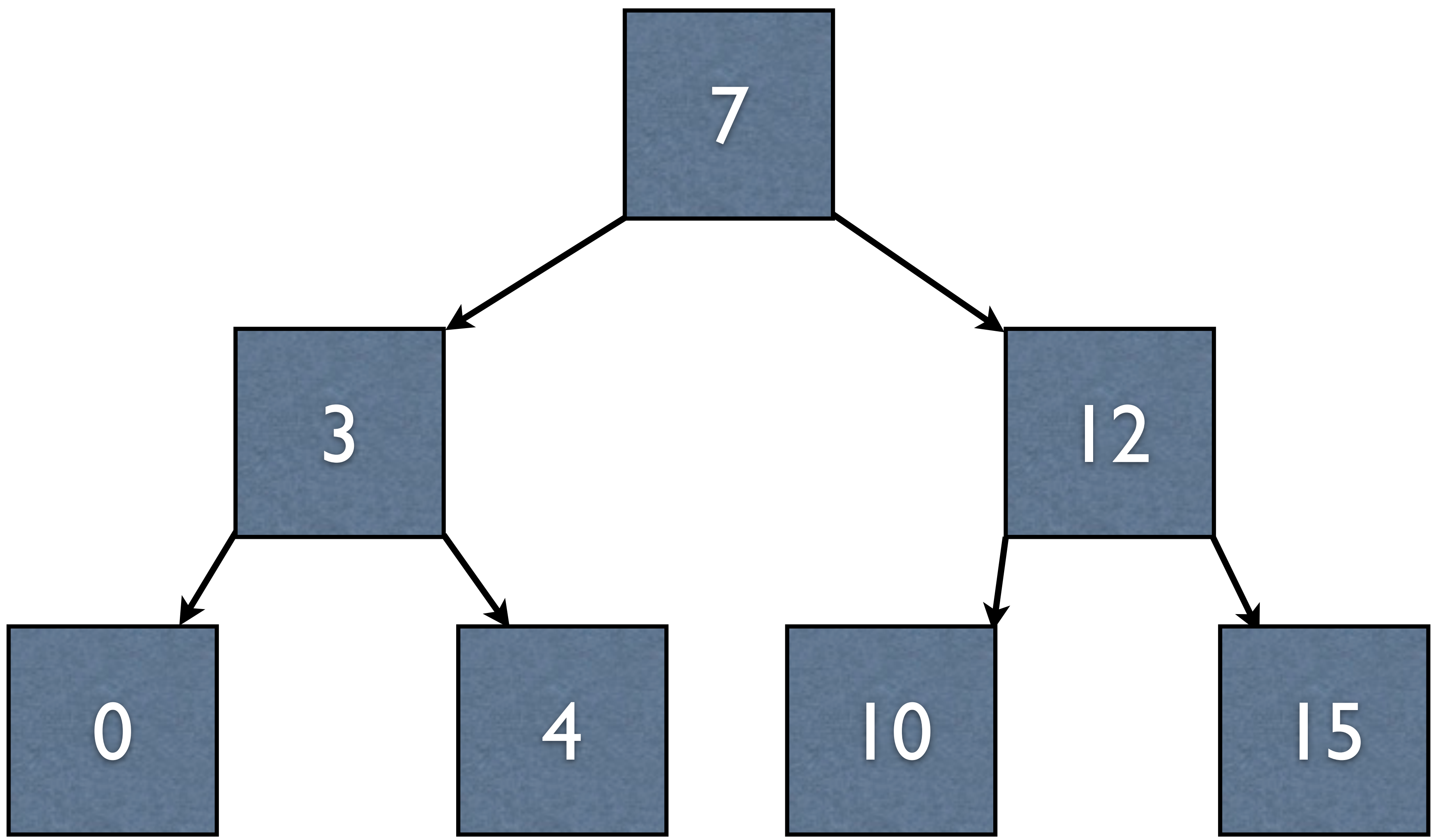
- Like a full `k`-ary tree, except the last level is permitted to be missing nodes, but **only on the right**




**not** ok

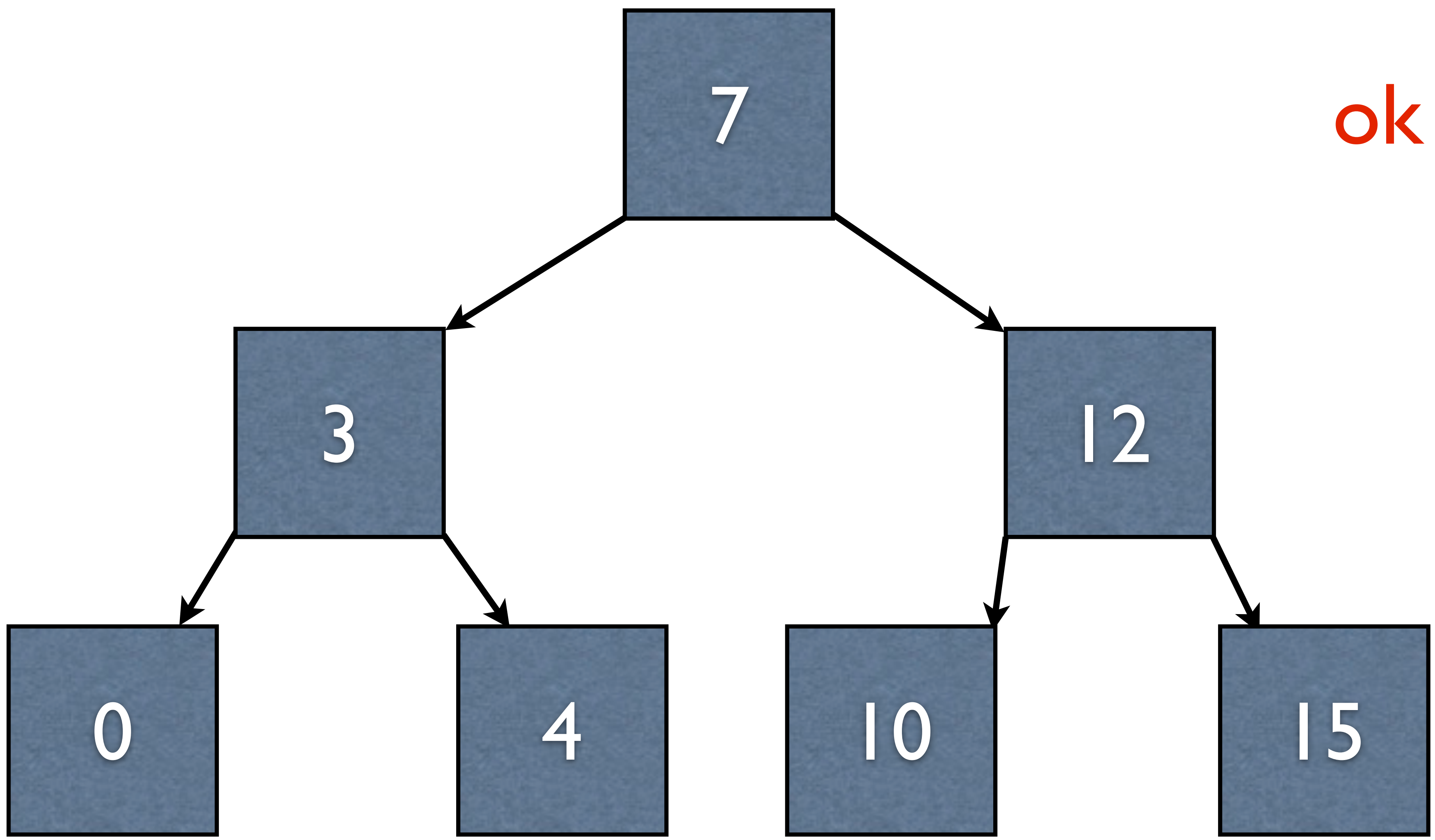
# Balanced Tree

- For all nodes, the height of the left and right subtrees differ by no more than one



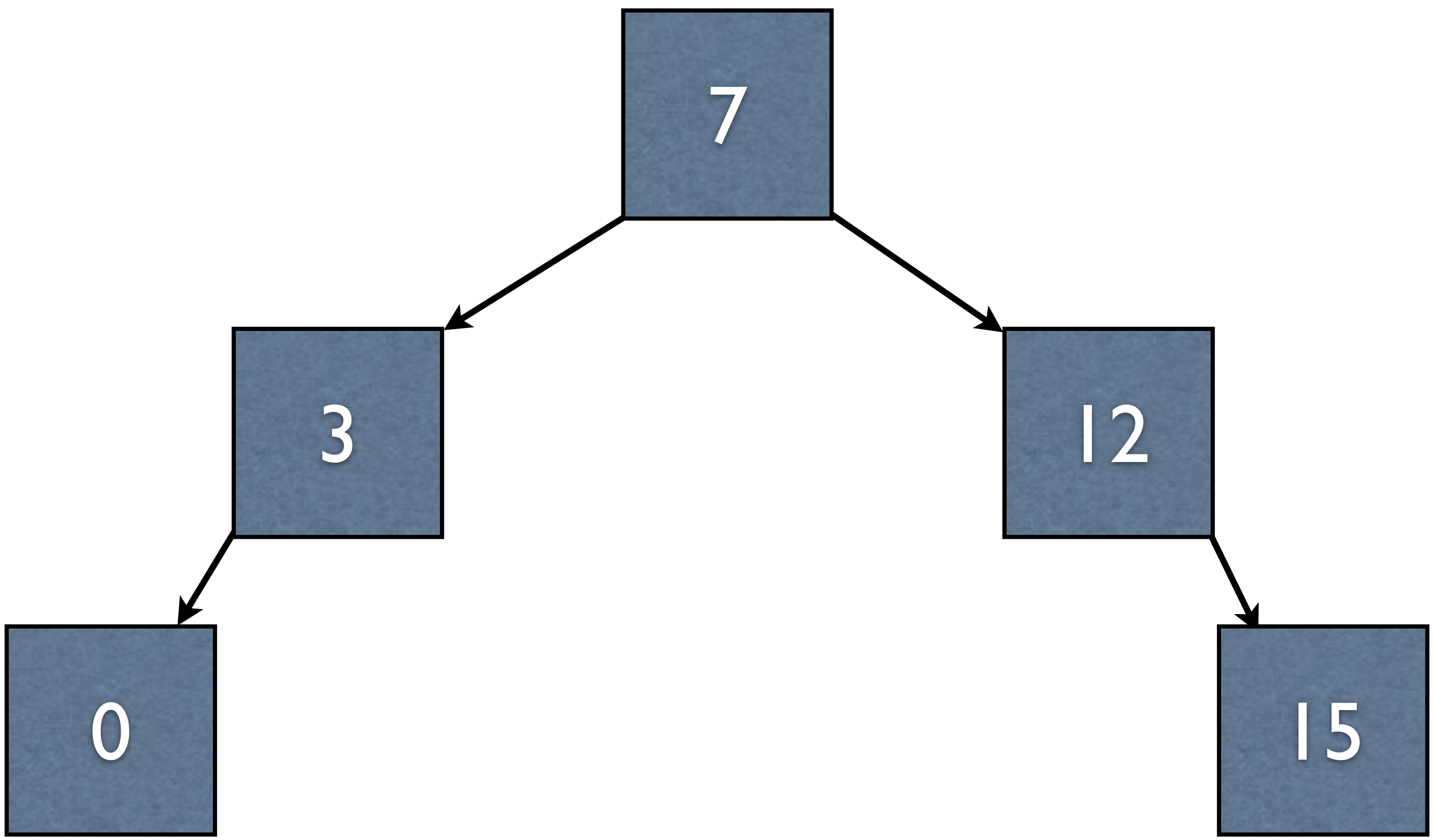
# Balanced Tree

- For all nodes, the height of the left and right subtrees differ by no more than one



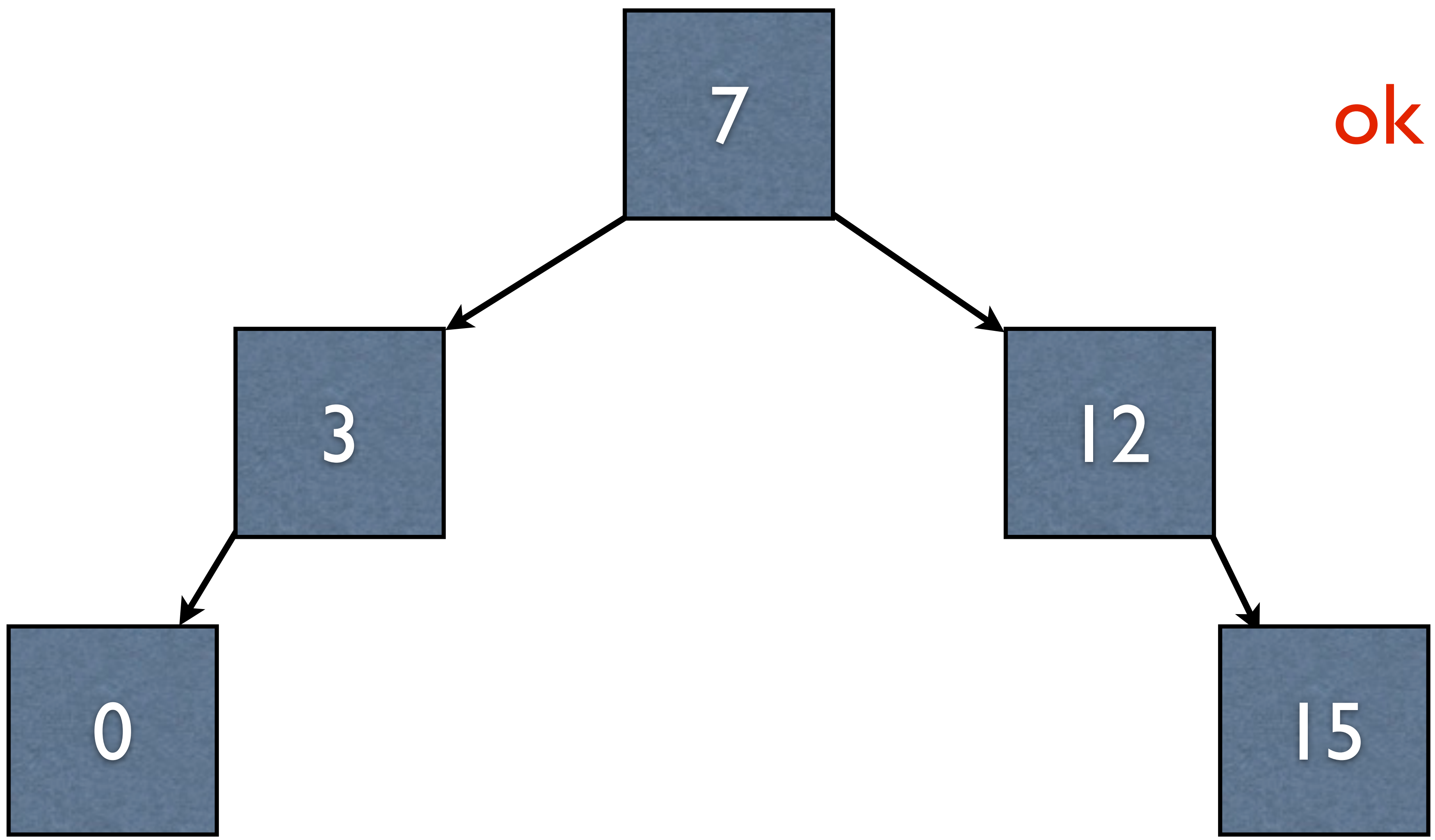
# Balanced Tree

- For all nodes, the height of the left and right subtrees differ by no more than one



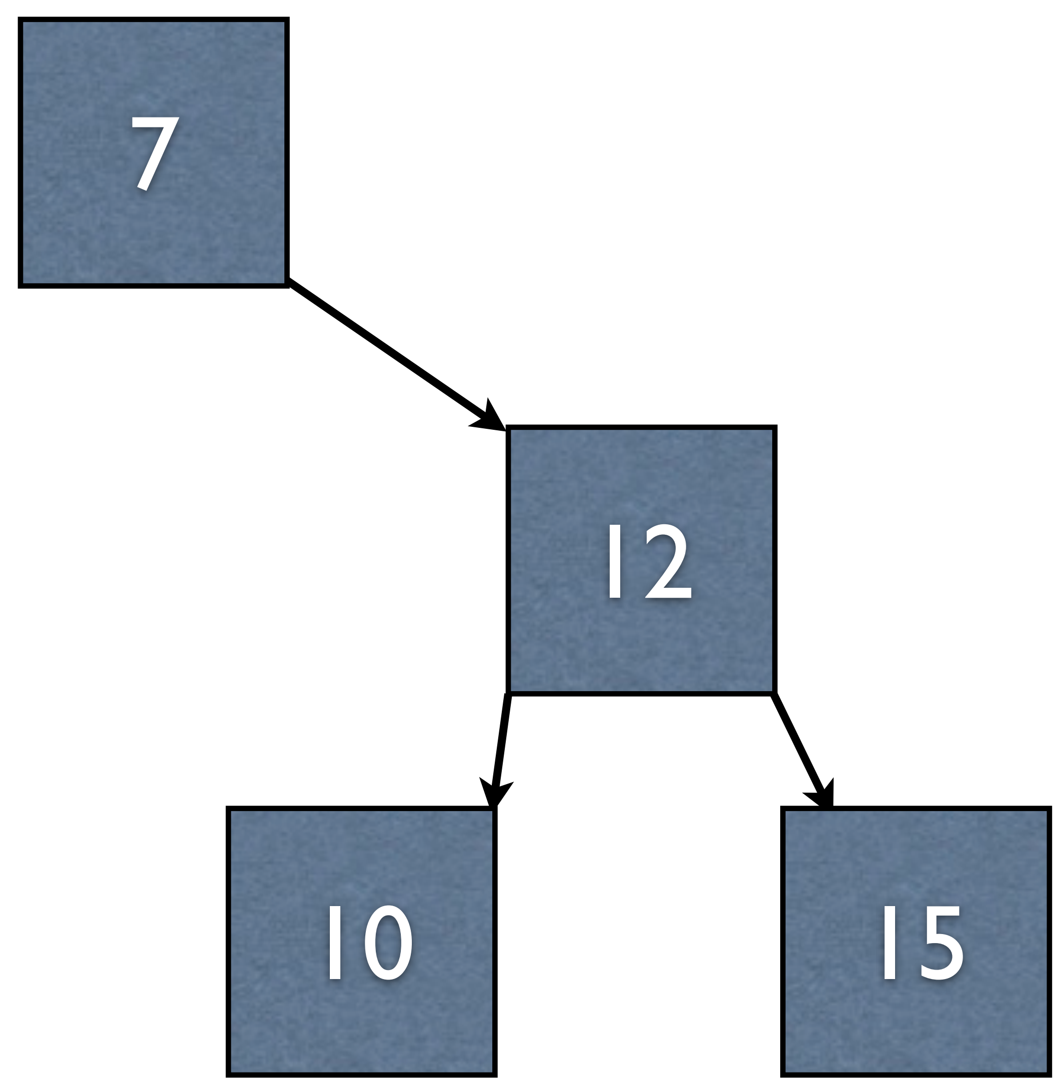
# Balanced Tree

- For all nodes, the height of the left and right subtrees differ by no more than one



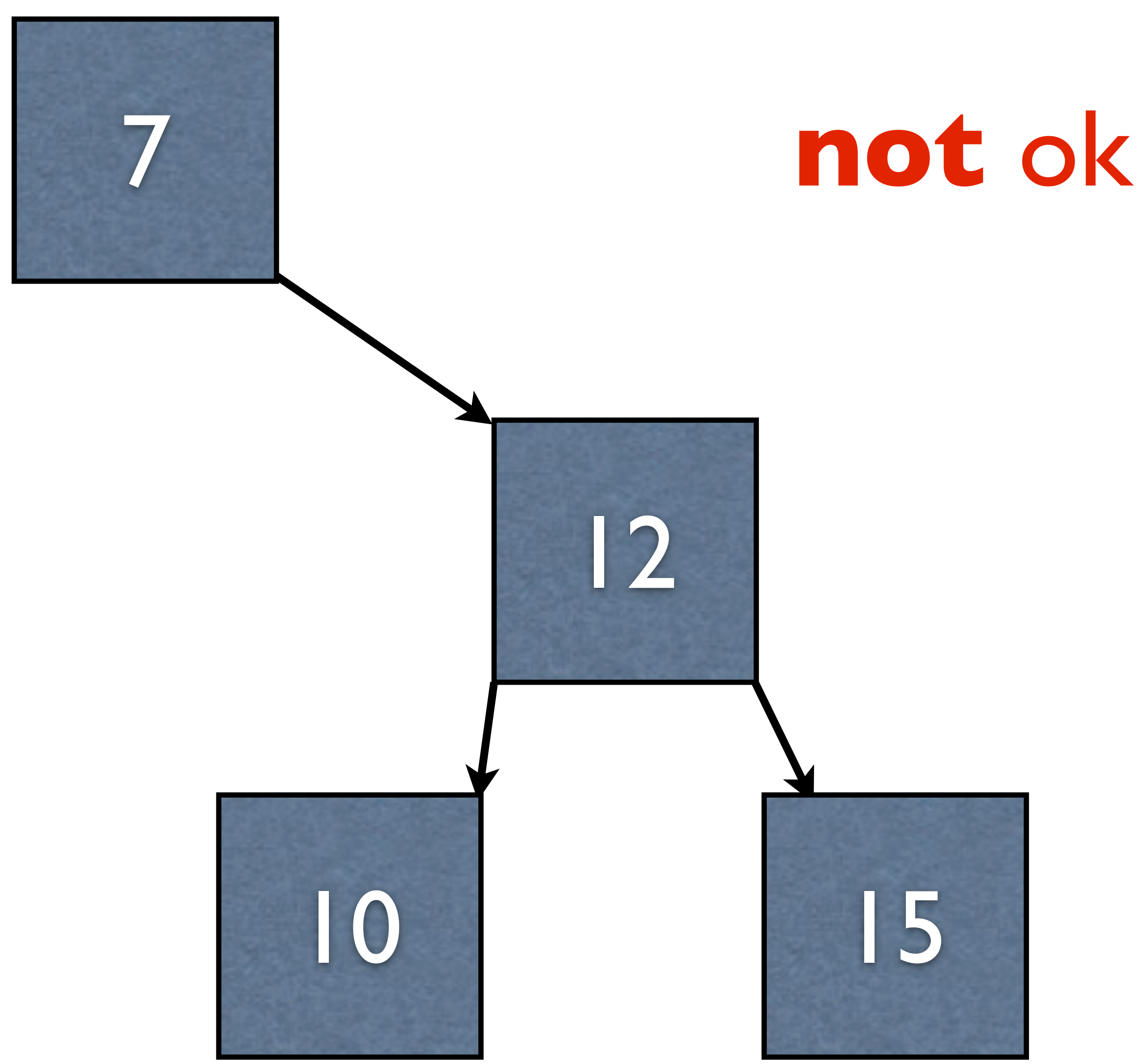
# Balanced Tree

- For all nodes, the height of the left and right subtrees differ by no more than one



# Balanced Tree

- For all nodes, the height of the left and right subtrees differ by no more than one

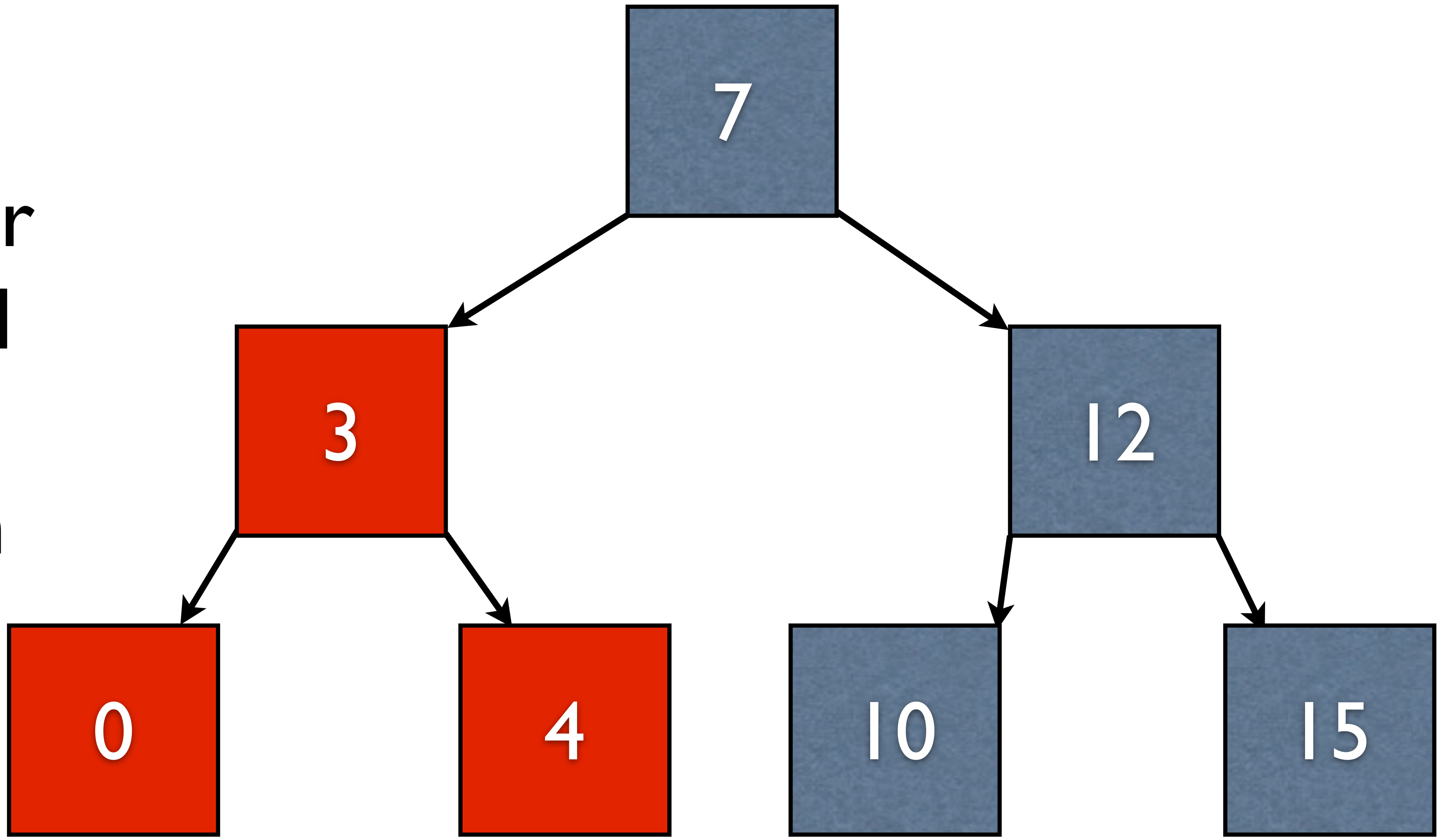


# Subtree

- Nearly synonymous with node
  - We recursively defined the tree to be either a node with an element and two children, or an empty tree (`NULL`)
  - Generally refers to some subcomponent of a larger tree, **including** recursive subcomponents

# Subtree Example

Can refer to 3 and its children




**Cannot** just refer to 3

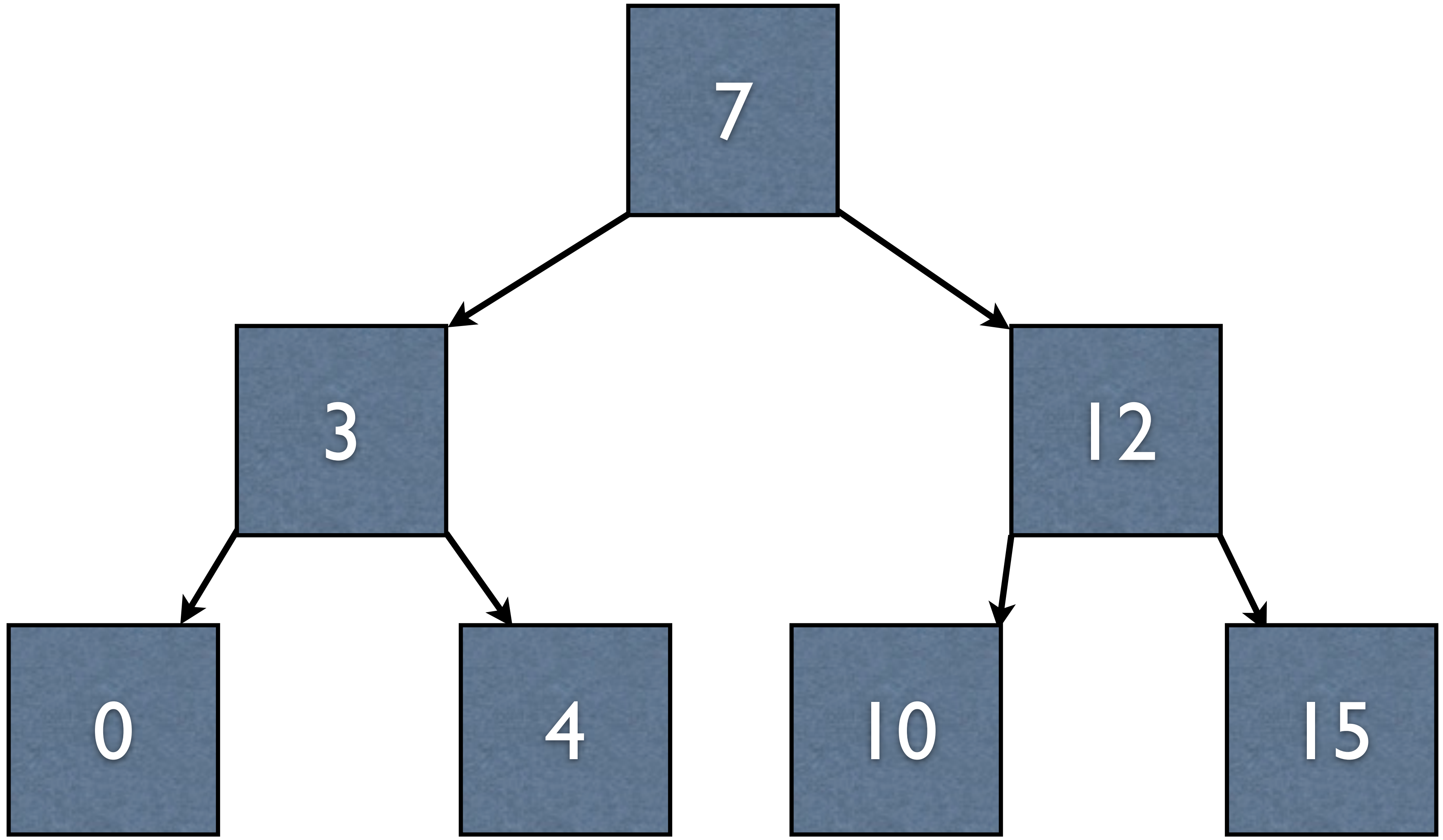
# Traversals

# Traversals

- For many tree-related problems, the order in which nodes are processed can have a huge impact
- Two basic kinds: breadth-first search and depth-first search

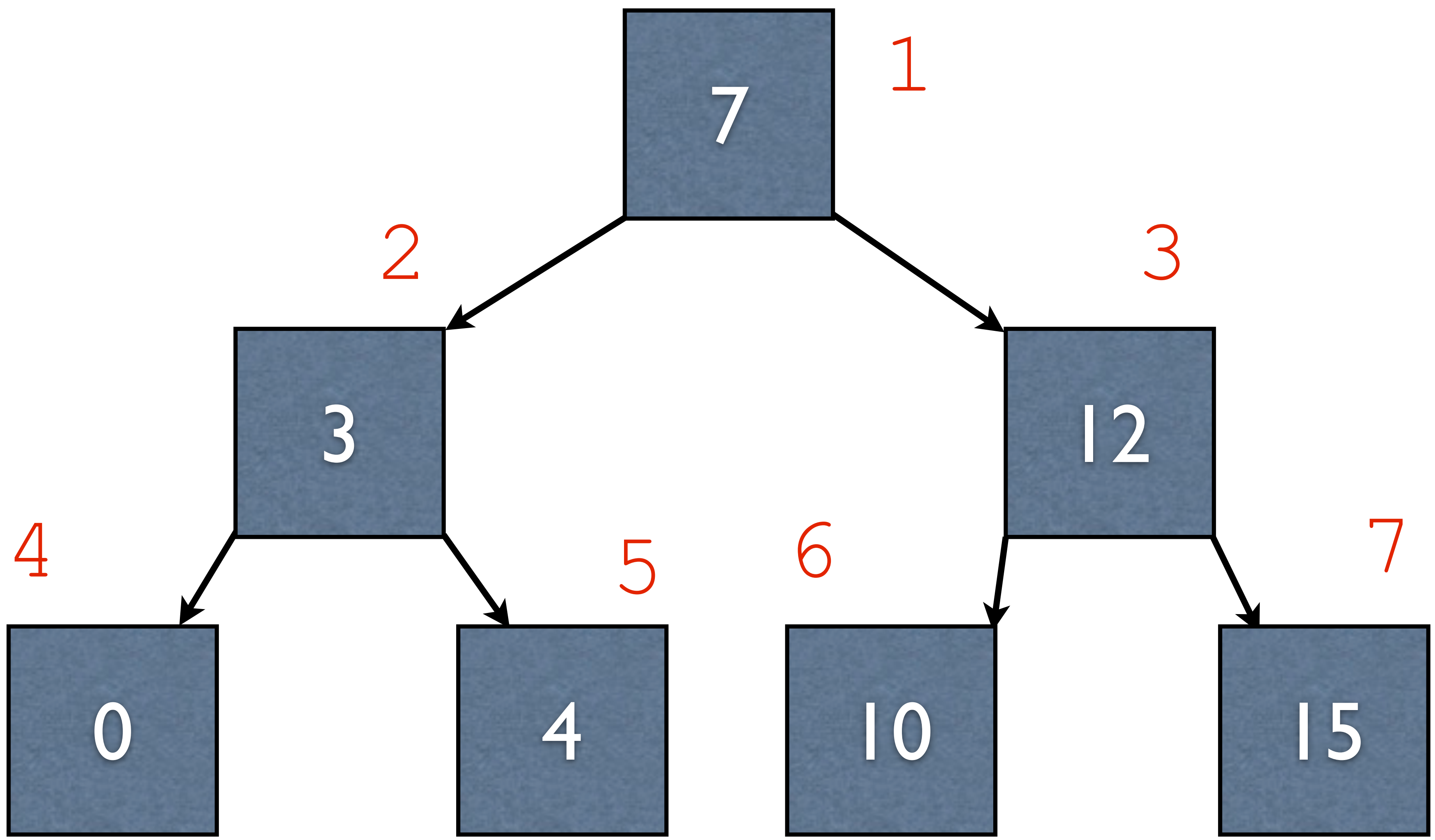
# Breath-First Search (BFS)

- Tree is traversed as if nodes were words on a page (top to bottom, left to right)



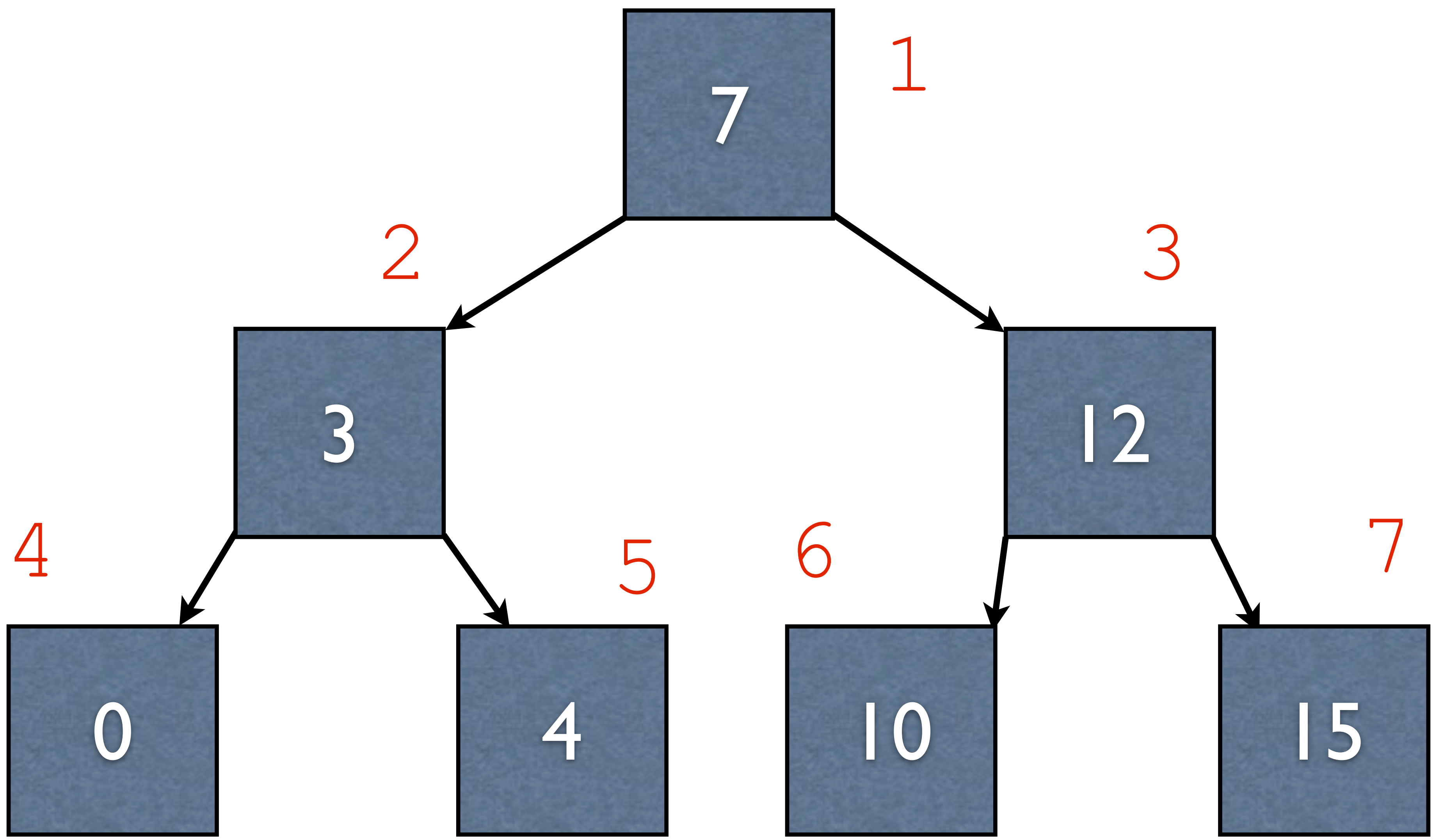
# Breath-First Search (BFS)

- Tree is traversed as if nodes were words on a page (top to bottom, left to right)



# Implementing BFS

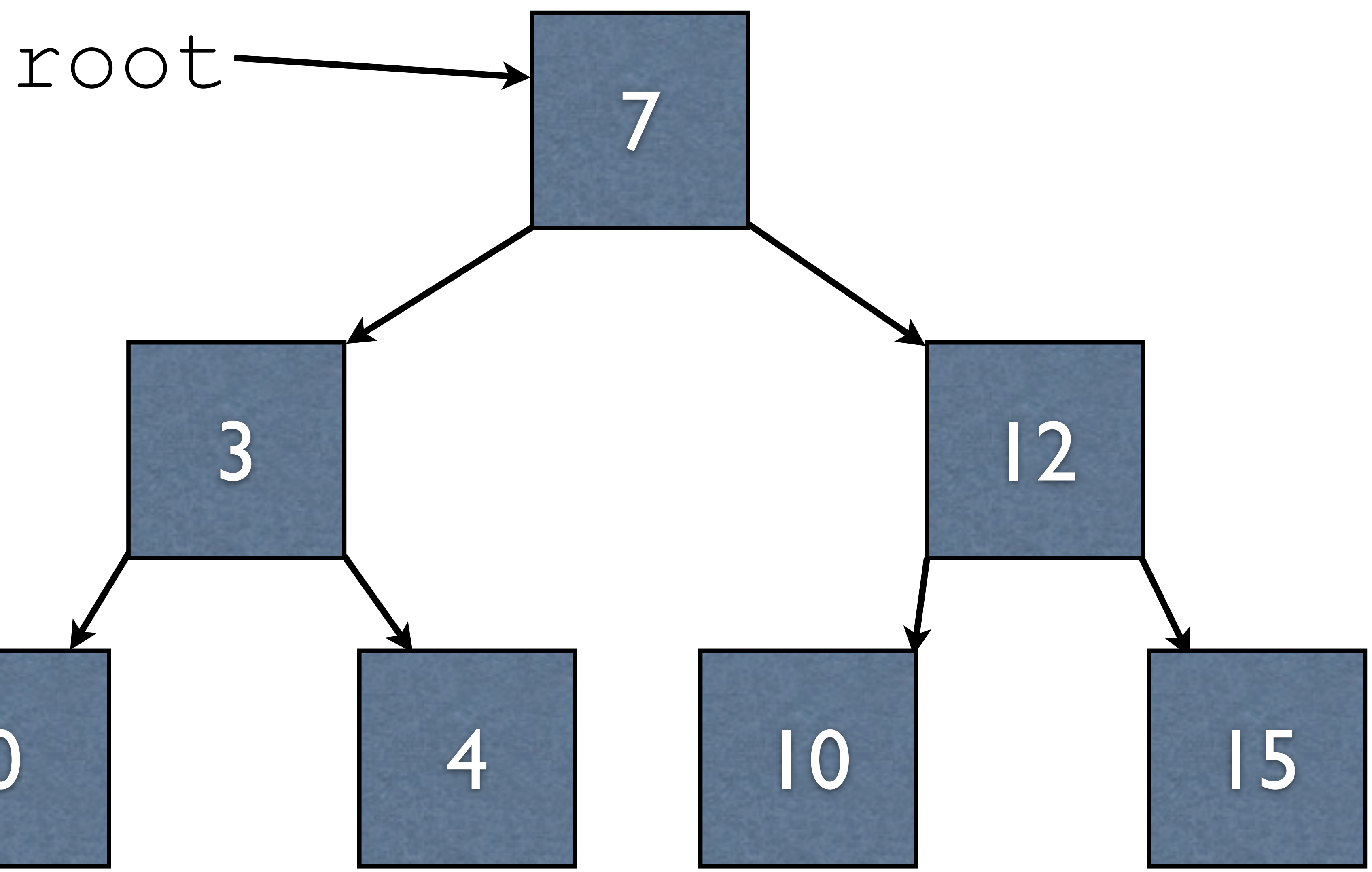
- Question: how might we implement BFS?
  - Hint: you'll need a data structure you've implemented before



# Implementing BFS

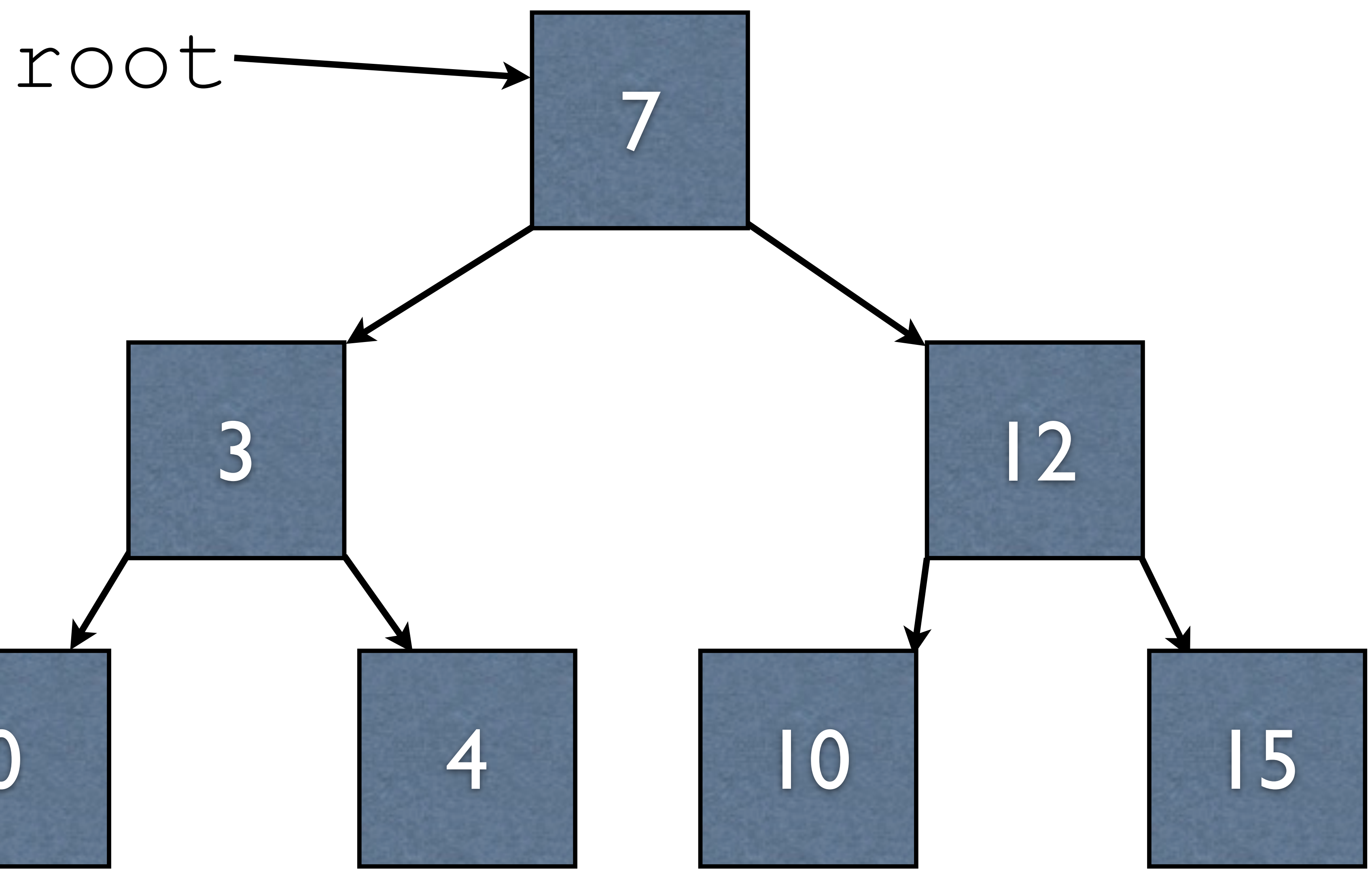
- Idea: put nodes on a queue
- Visit nodes according to the queue order
- When we are done with a node, put its children onto the end of the queue

# Implementing BFS




Queue: <<empty>>

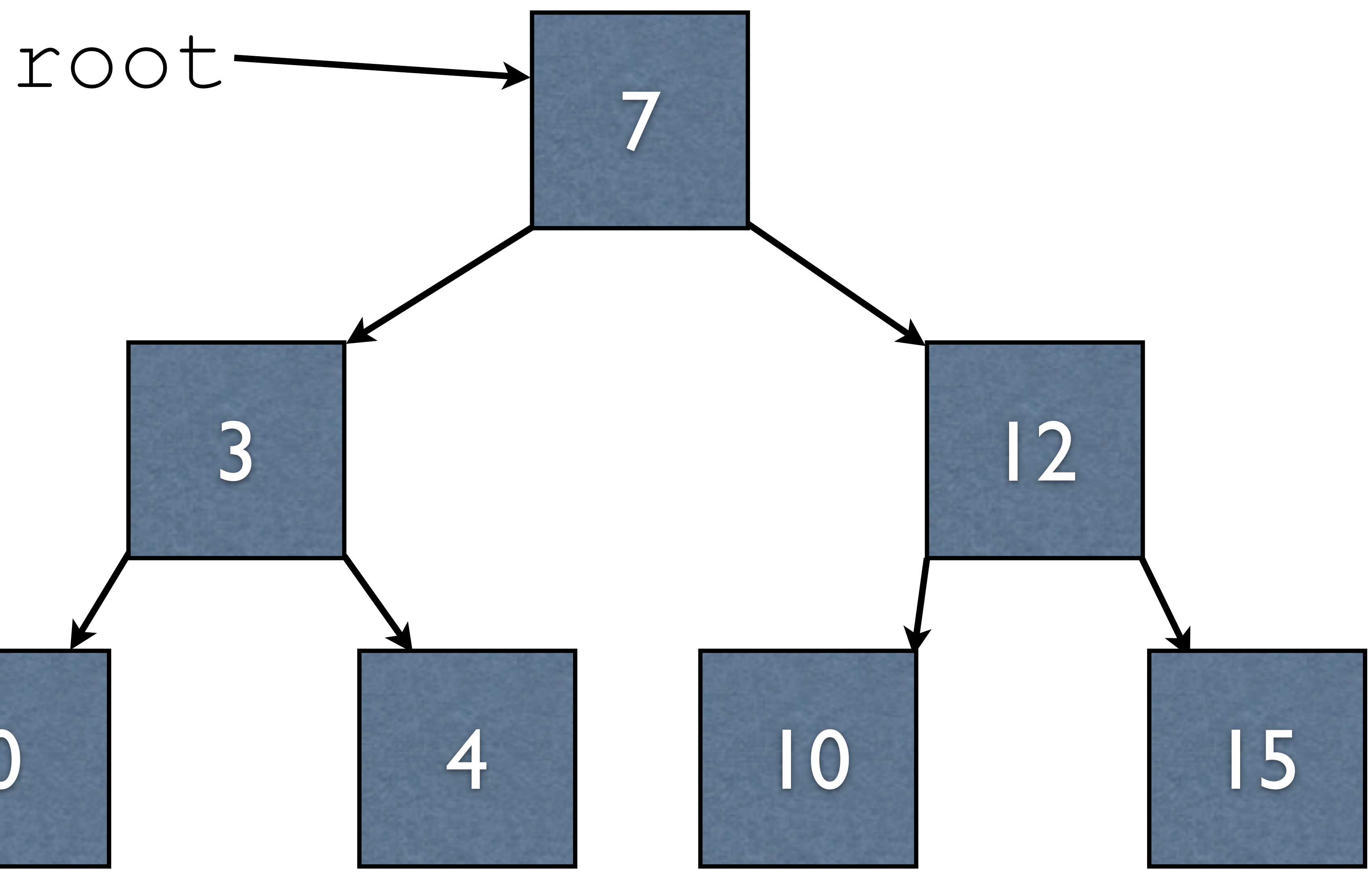
# Implementing BFS




Put `root` on the queue first (this is the node, not just the number)

Queue: 7

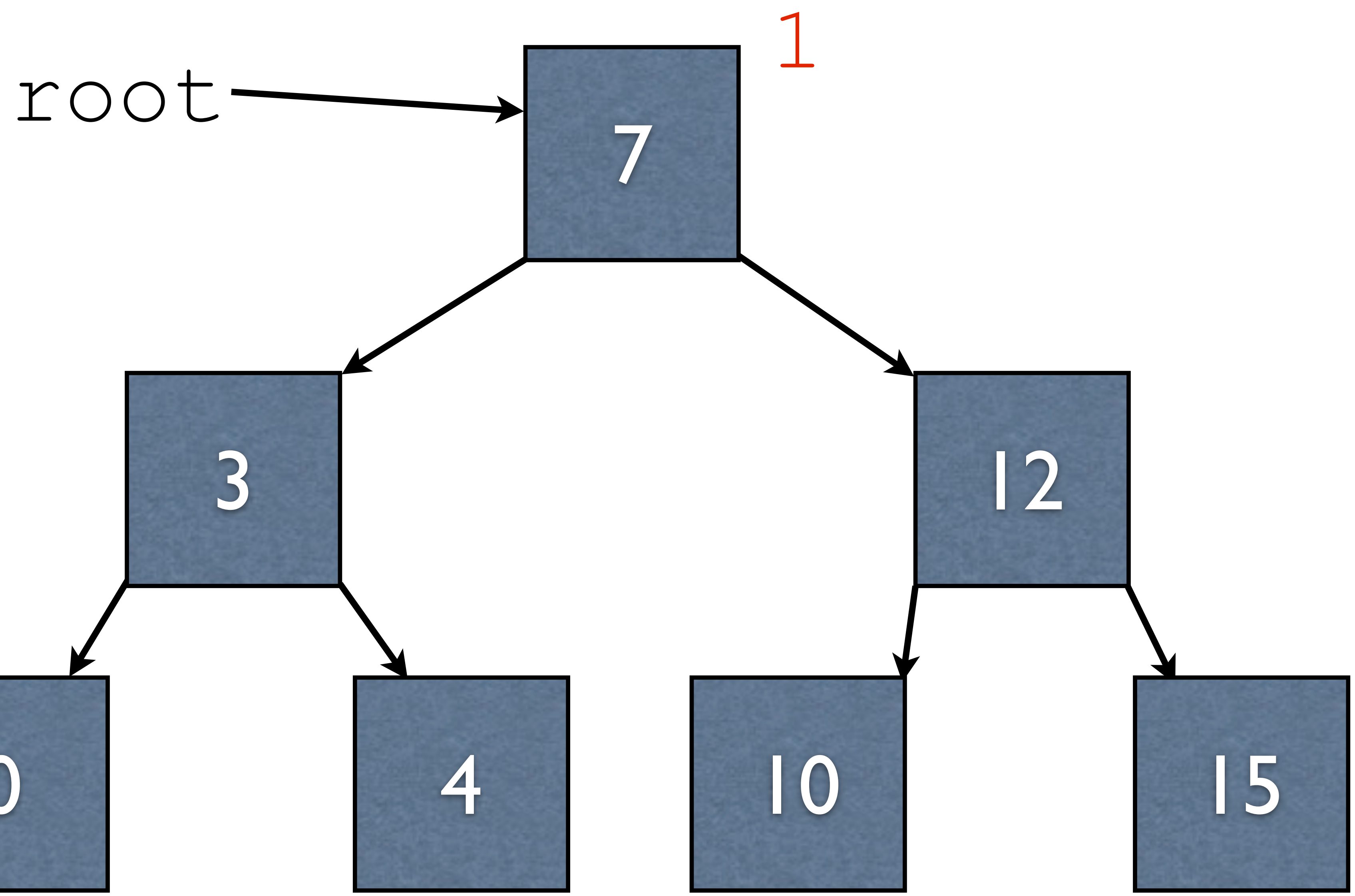
# Implementing BFS




Now dequeue

Queue: `7`

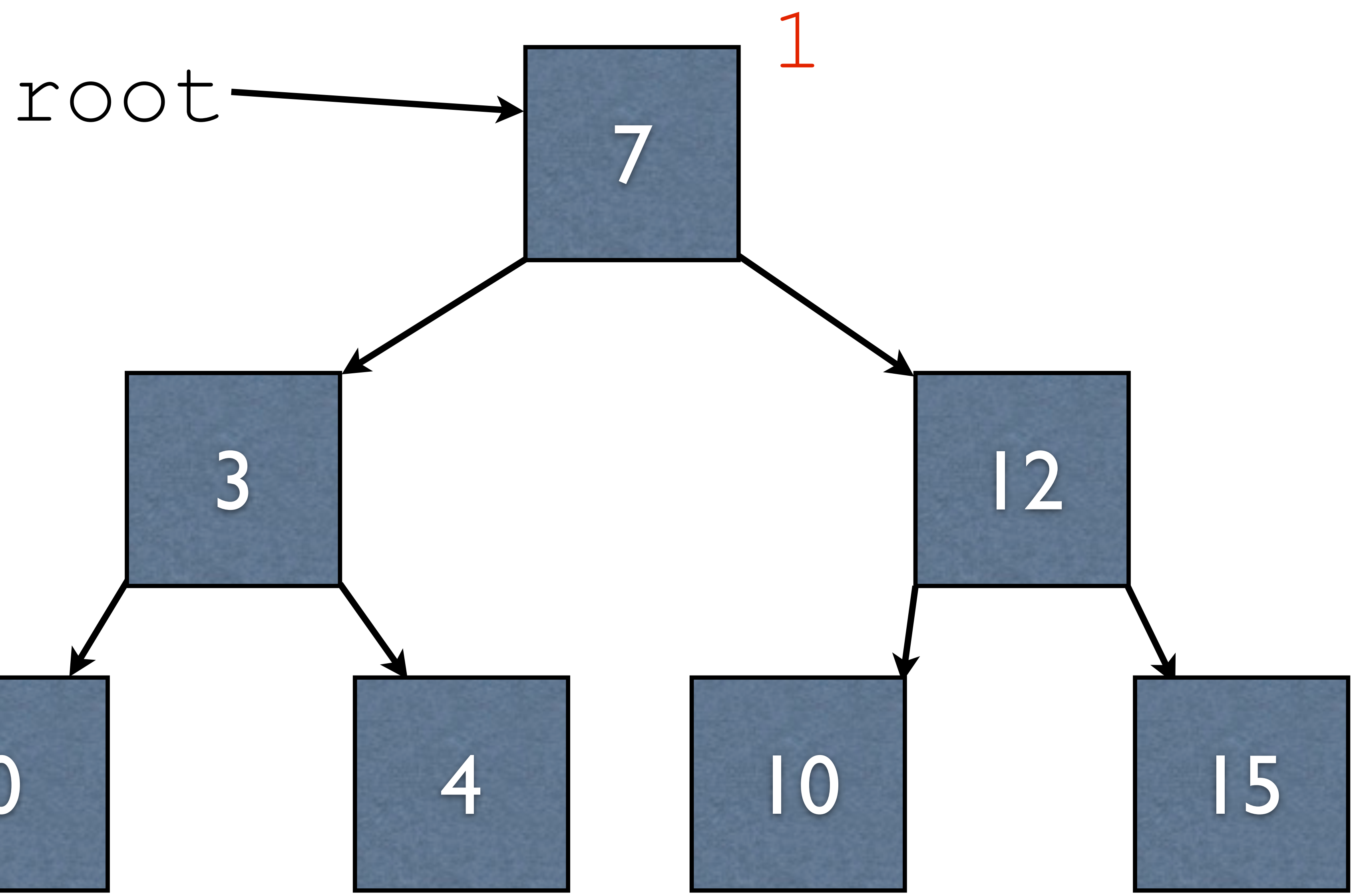
# Implementing BFS




Now dequeue

Queue: 7

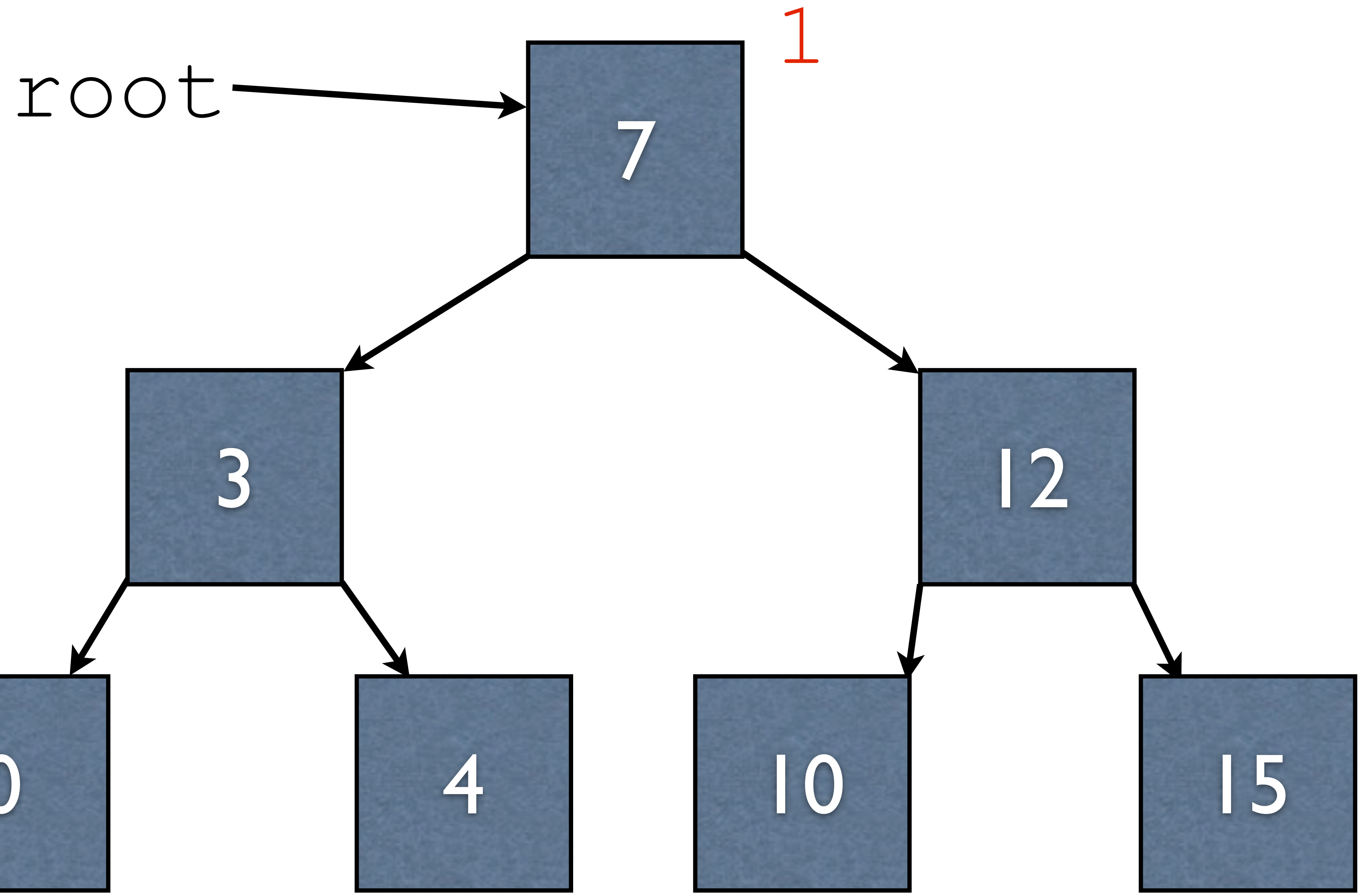
# Implementing BFS




Now dequeue

Queue: <<empty>>

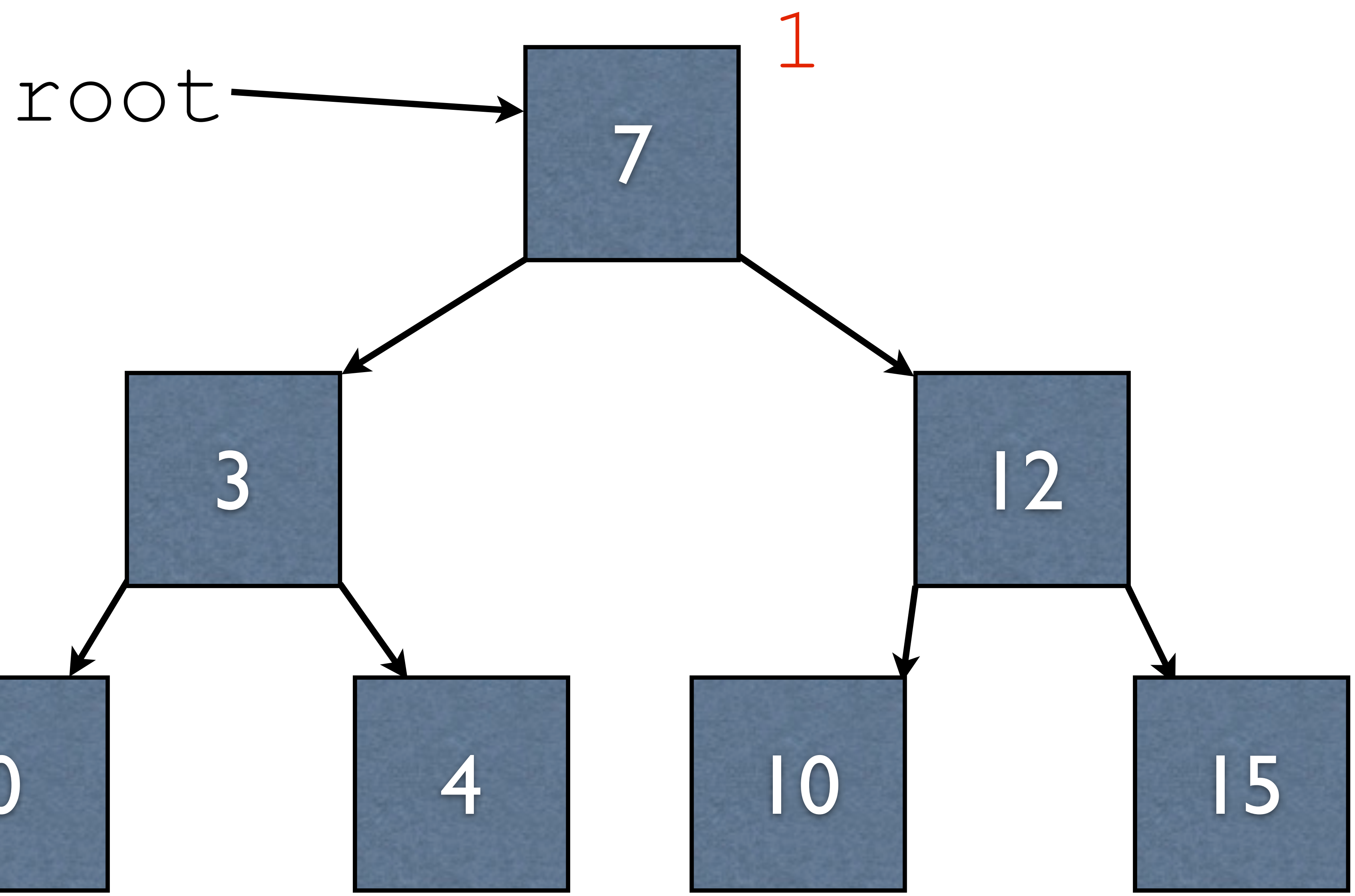
# Implementing BFS




Now put on the child nodes

Queue: `<<empty>>`

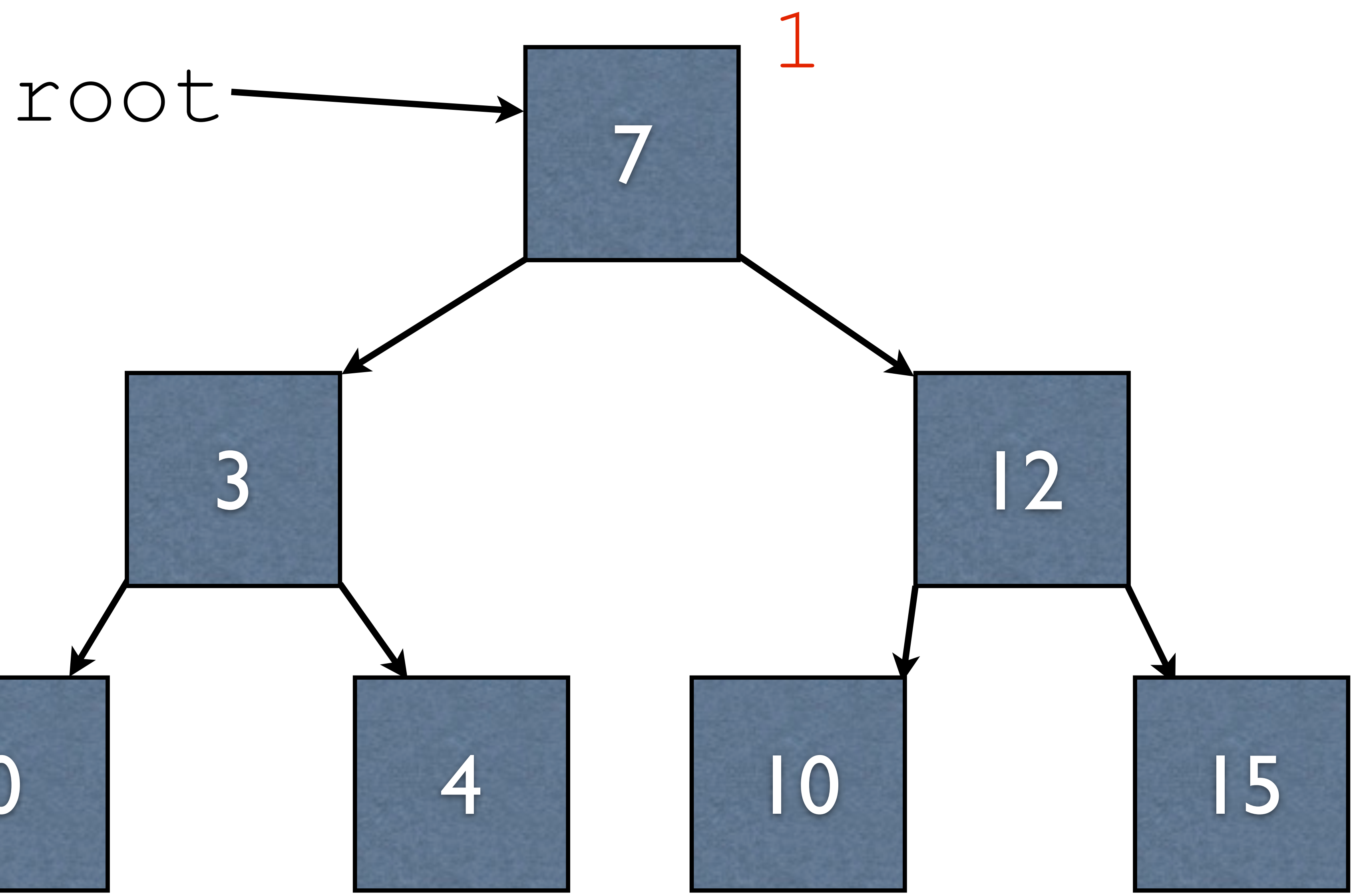
# Implementing BFS




Now put on the child nodes

Queue: `3, 12`

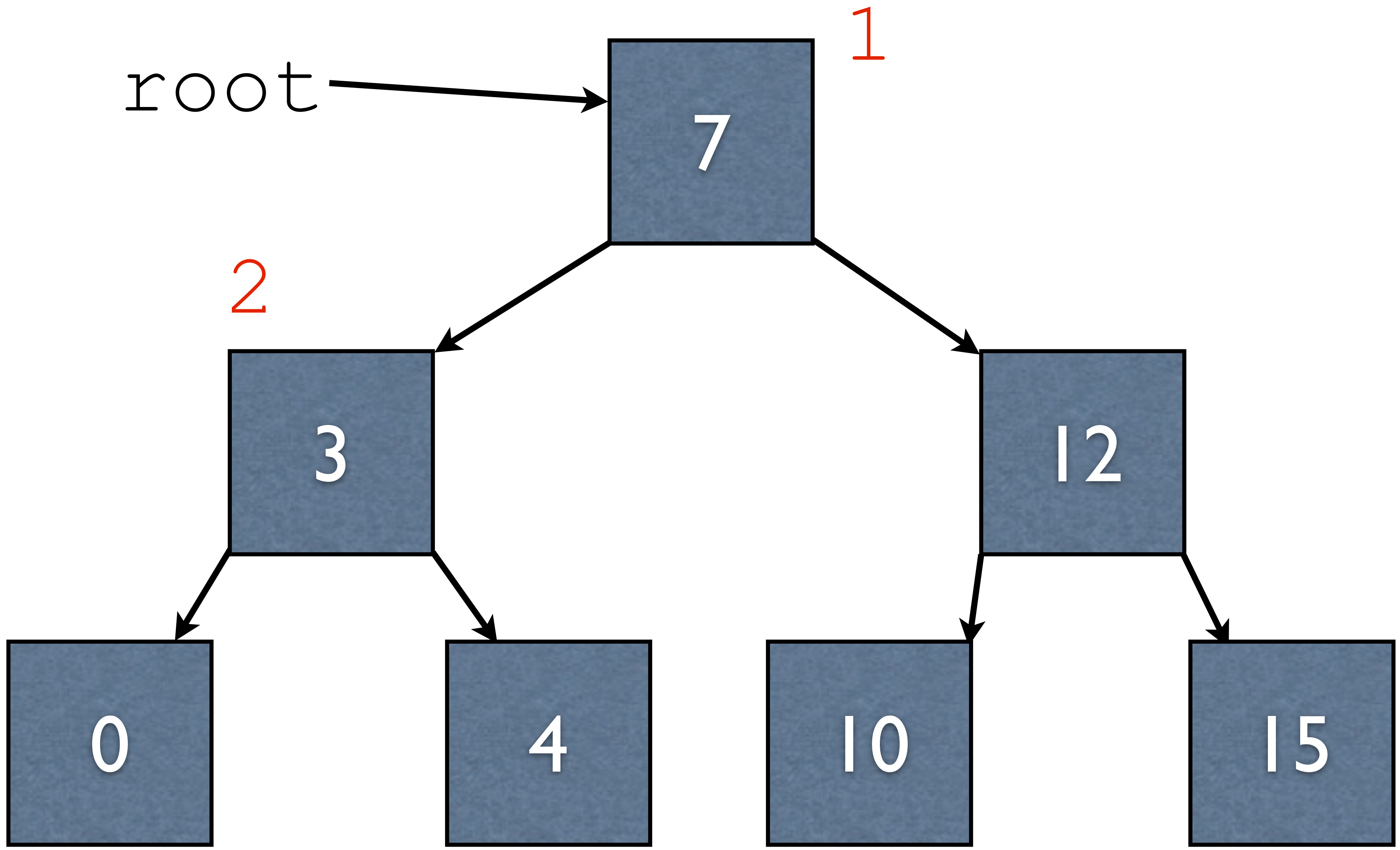
# Implementing BFS




Repeat

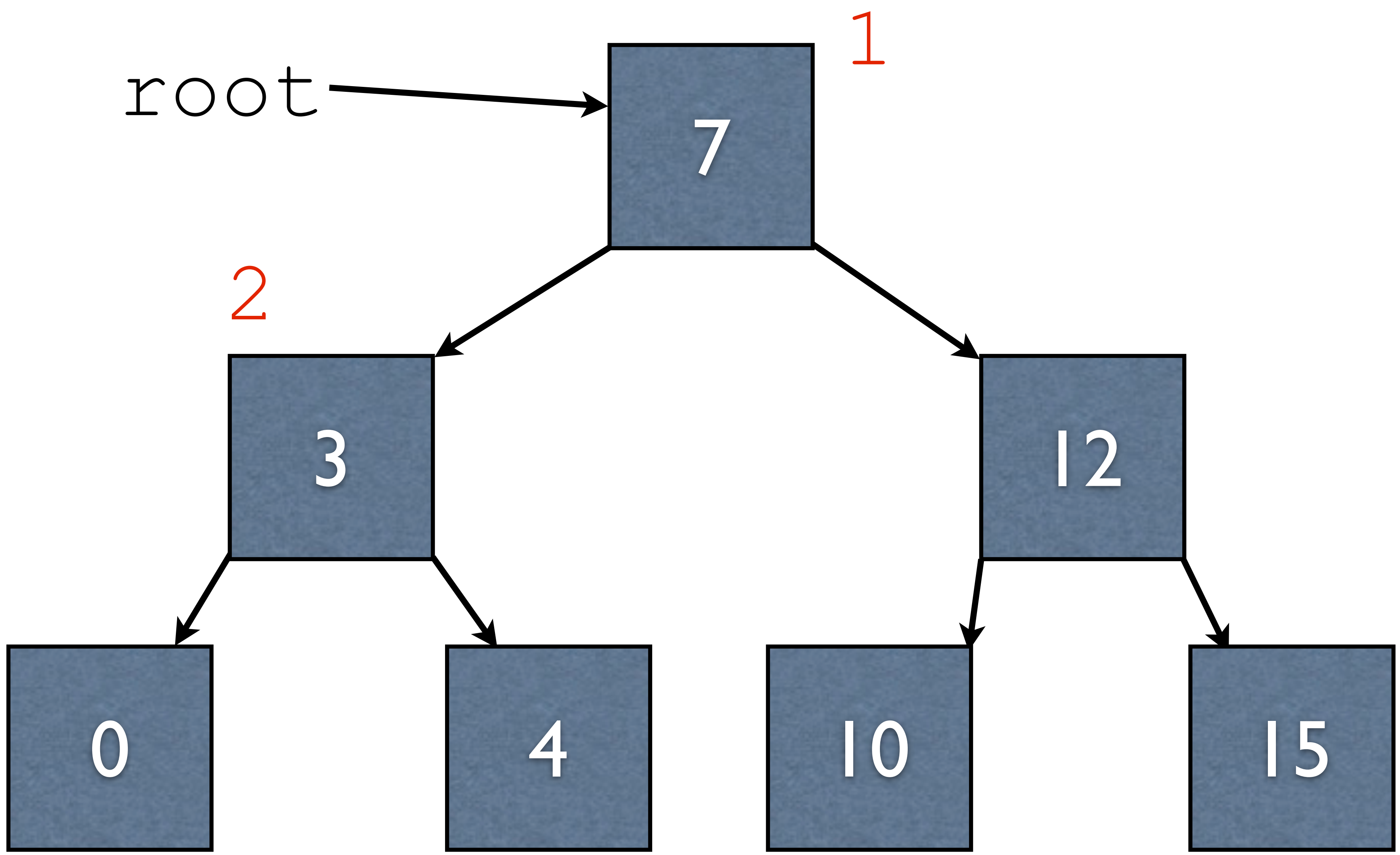
Queue: `3, 12`

# Implementing BFS



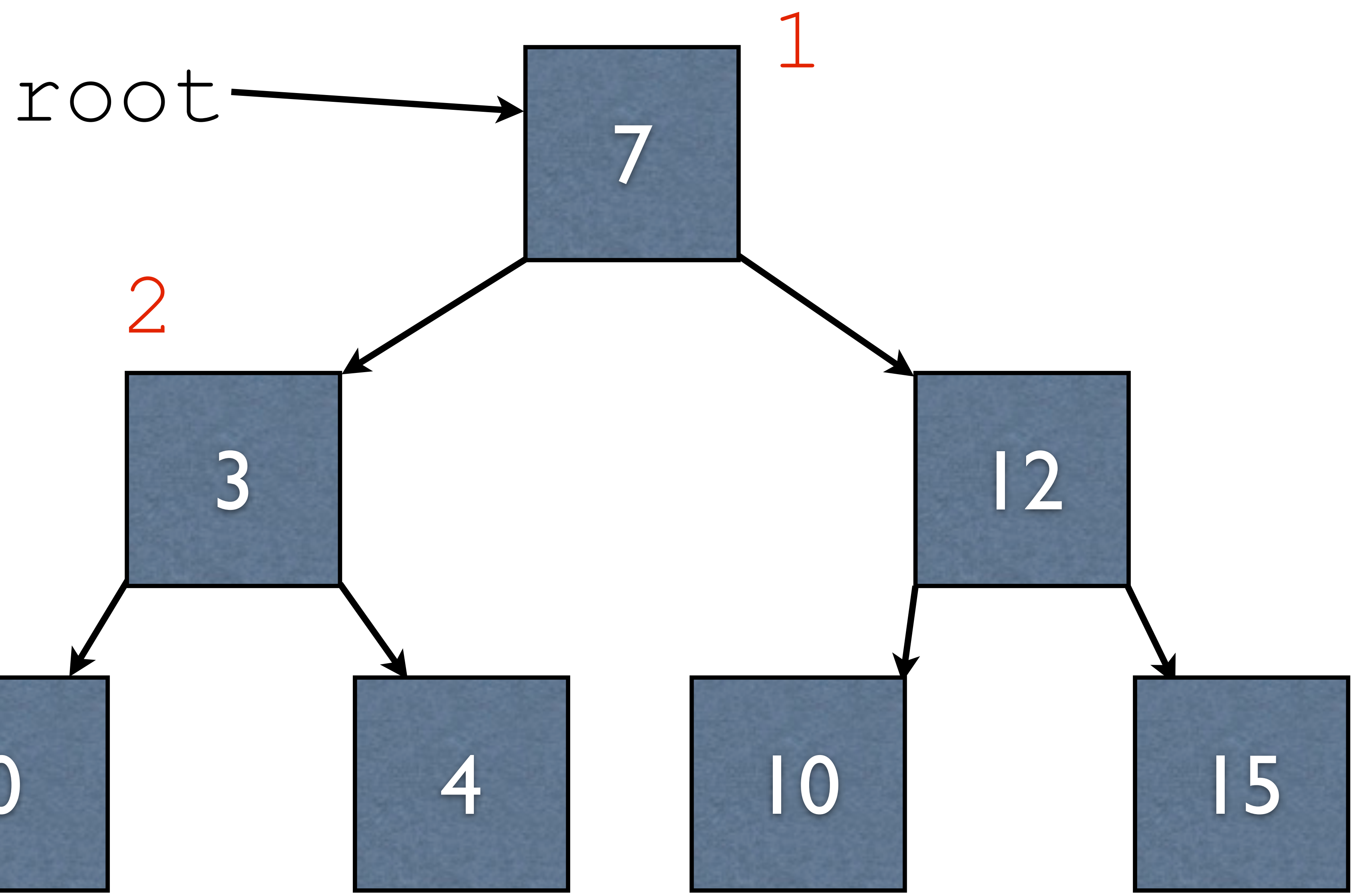

Queue: 3, 12

# Implementing BFS



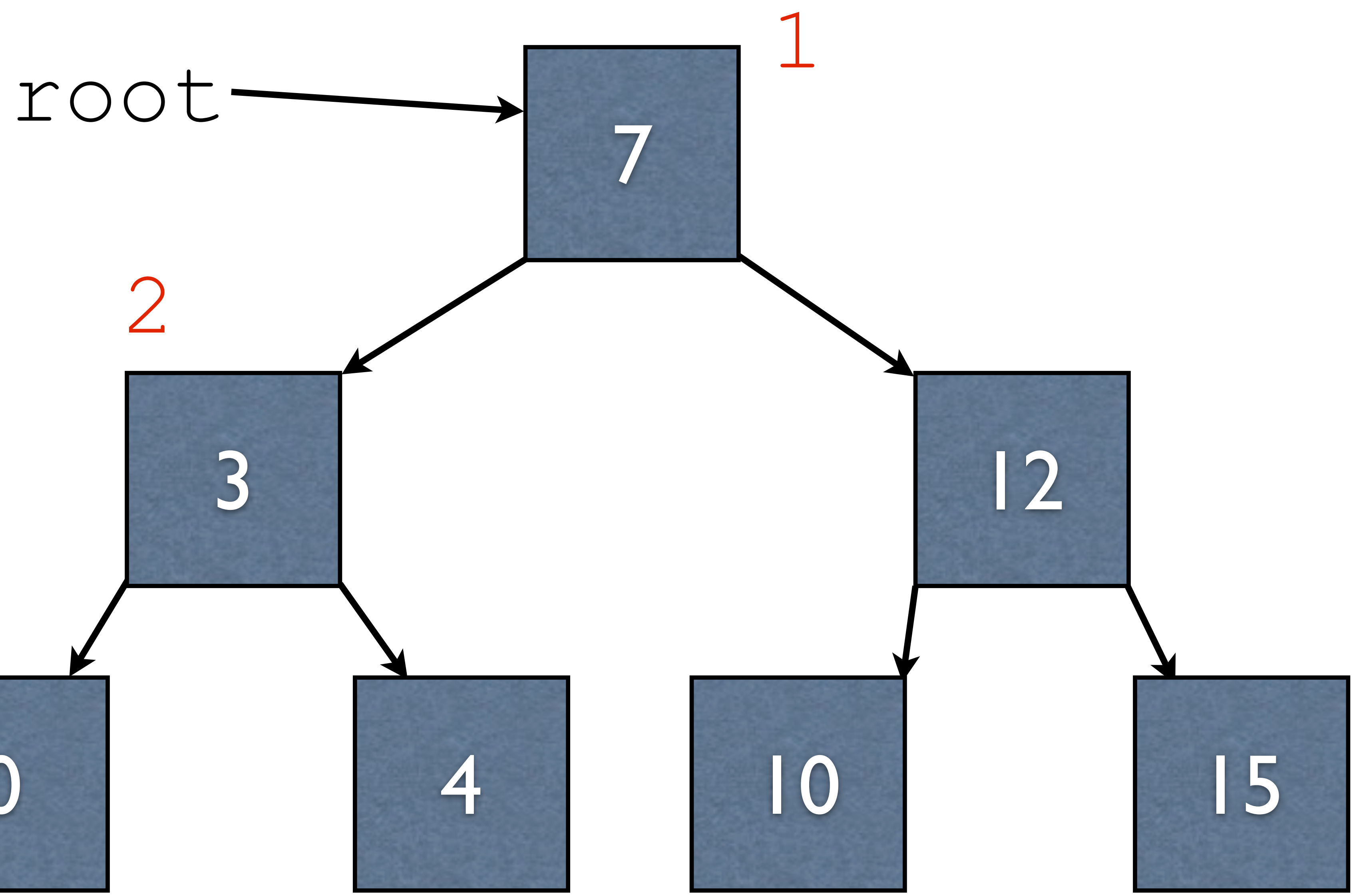

Queue: 12

# Implementing BFS



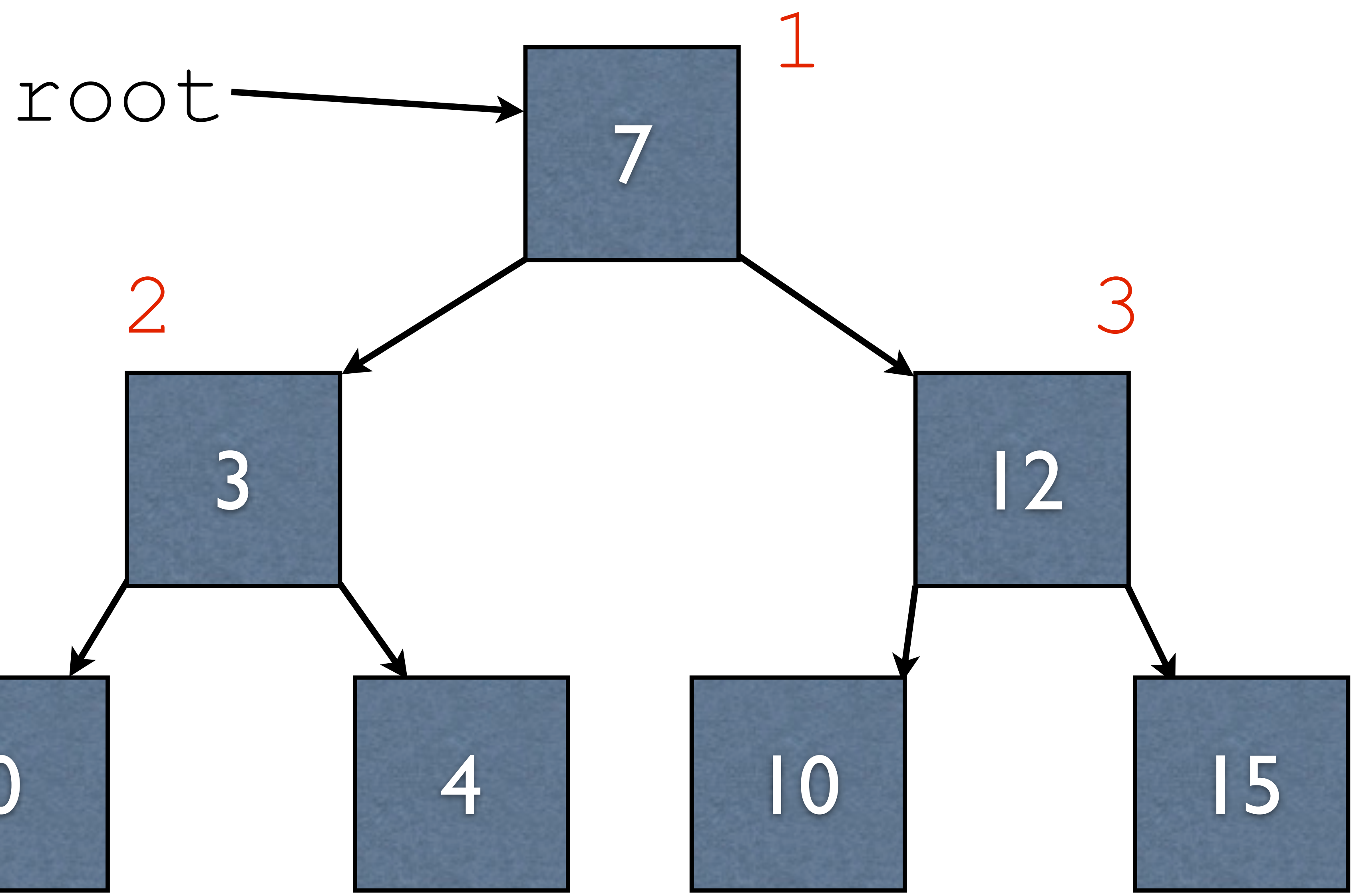

Queue: 12, 0, 4

# Implementing BFS



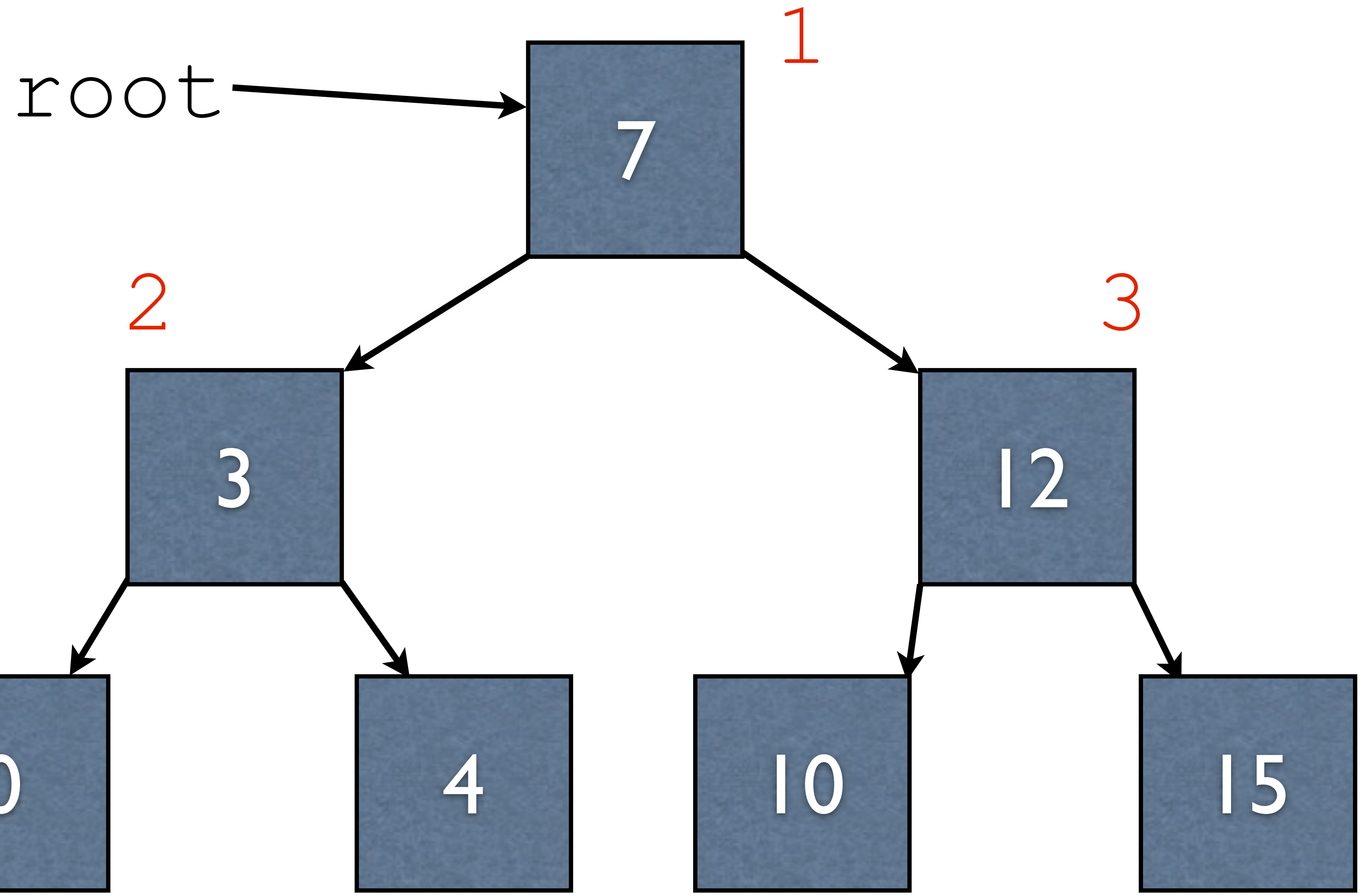

**Queue:** `12, 0, 4`

# Implementing BFS



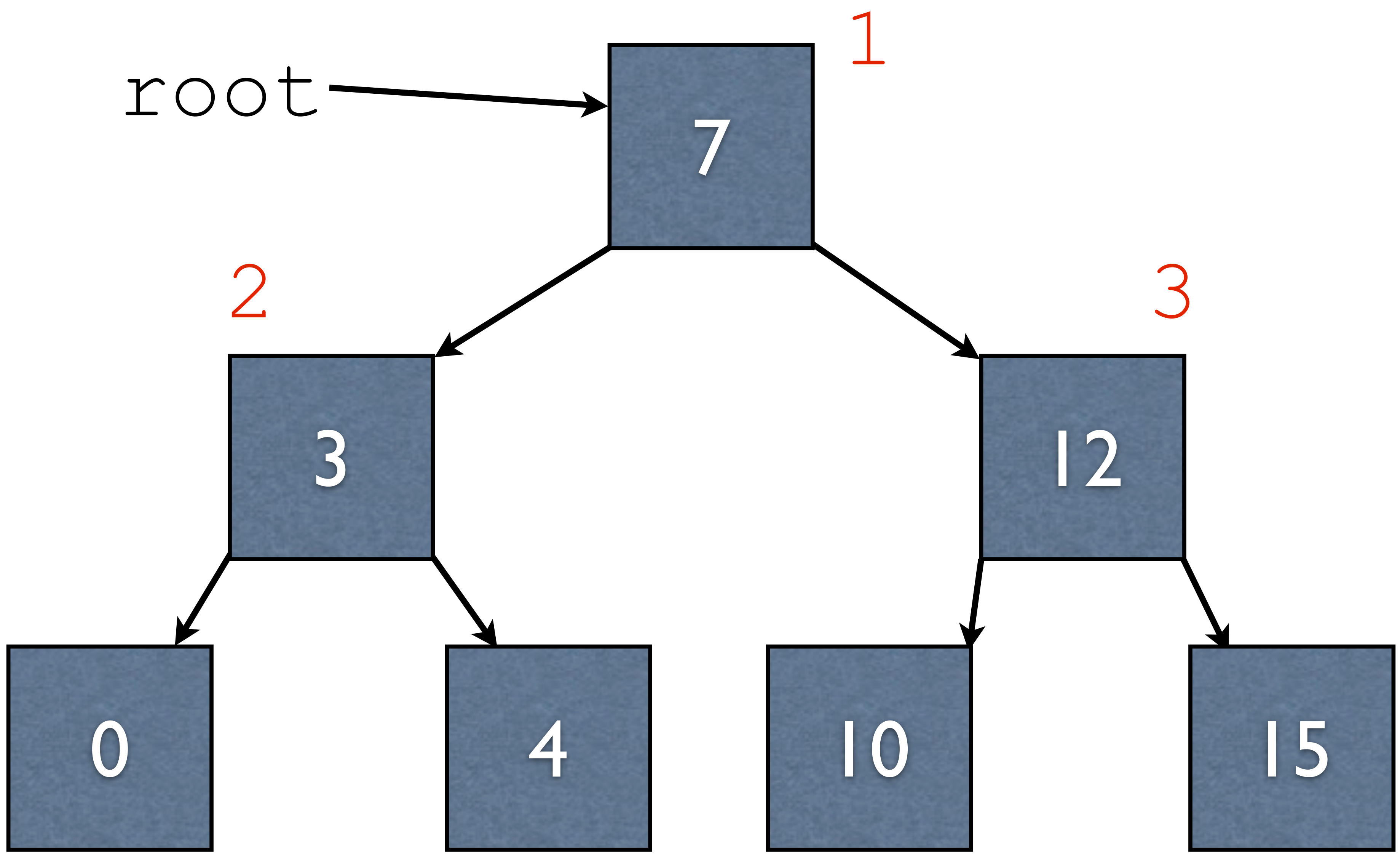

Queue: 12, 0, 4

# Implementing BFS



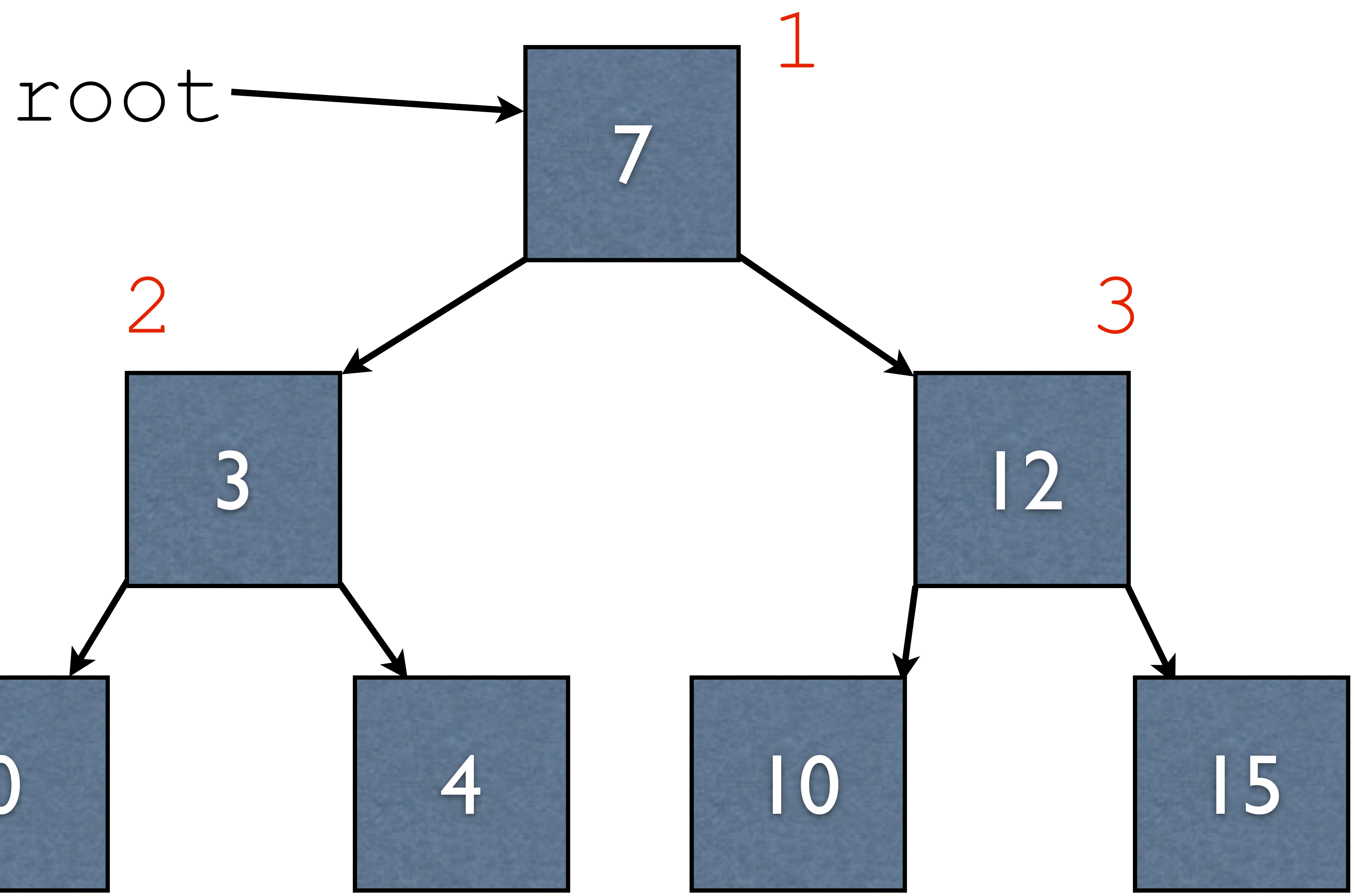

**Queue:** 0, 4

# Implementing BFS



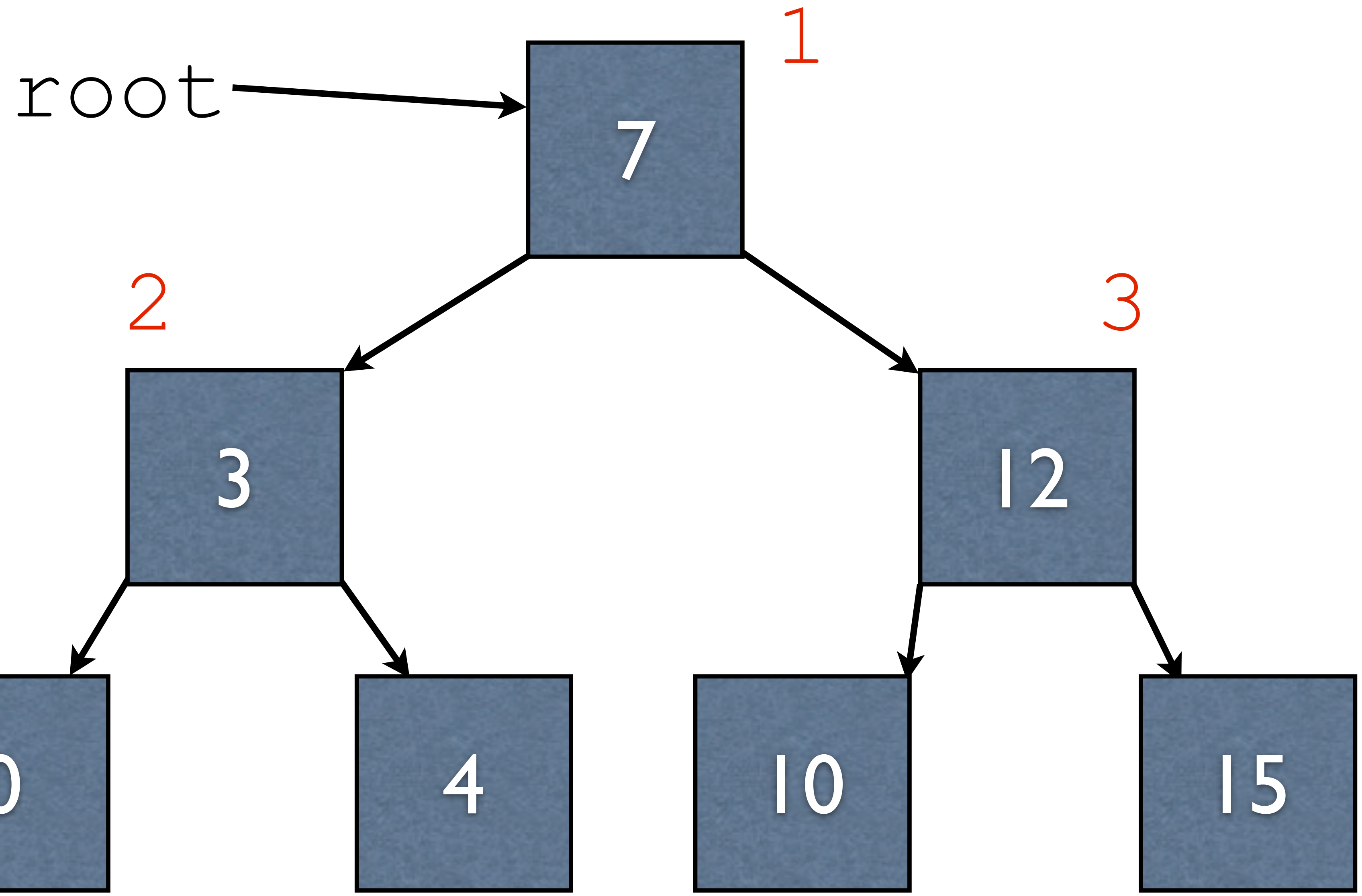

**Queue:** `0, 4, 10, 15`

# Implementing BFS



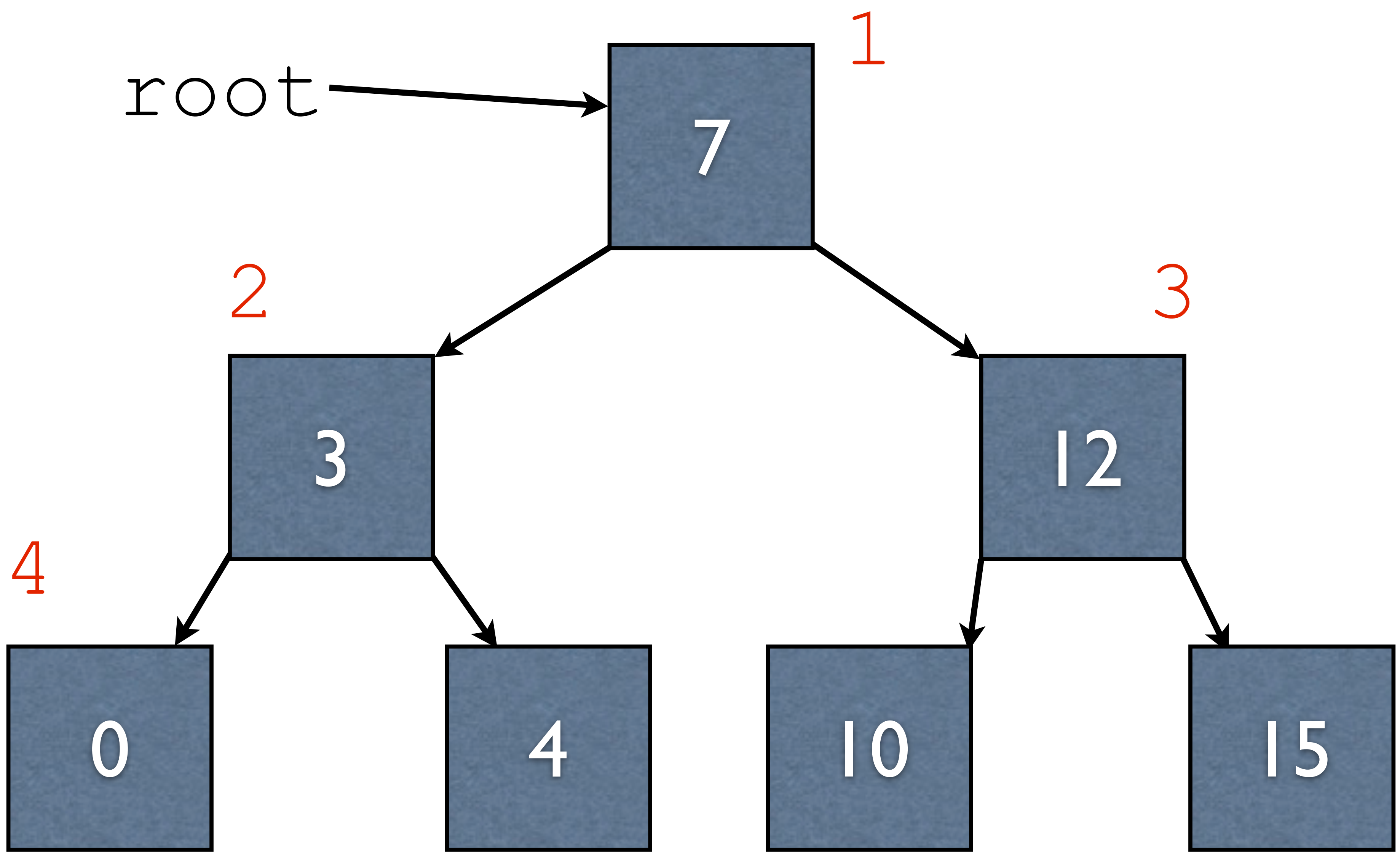

**Queue:** `0, 4, 10, 15`

# Implementing BFS



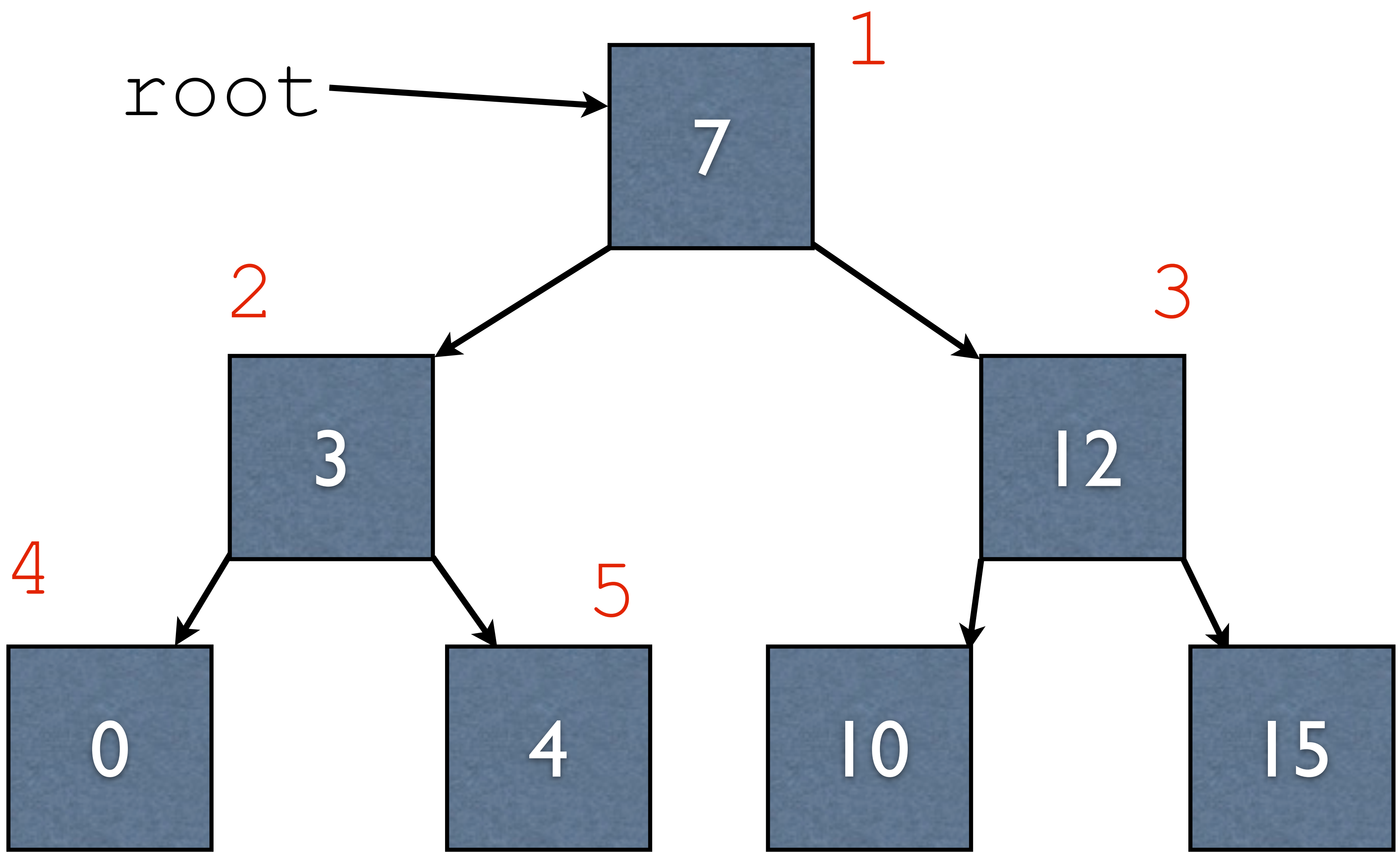

Queue: 0, 4, 10, 15

# Implementing BFS



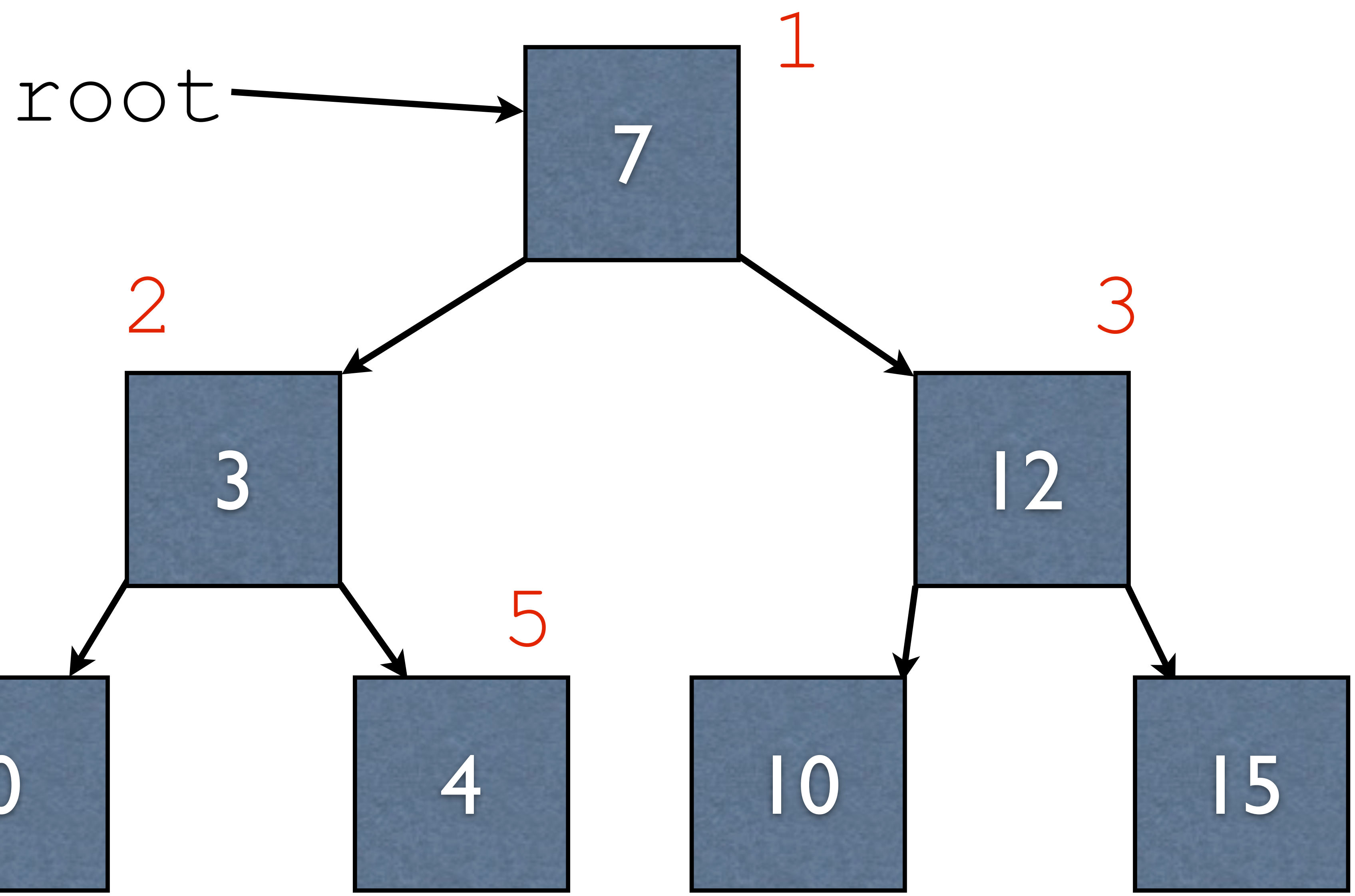

**Queue:** `4, 10, 15`

# Implementing BFS



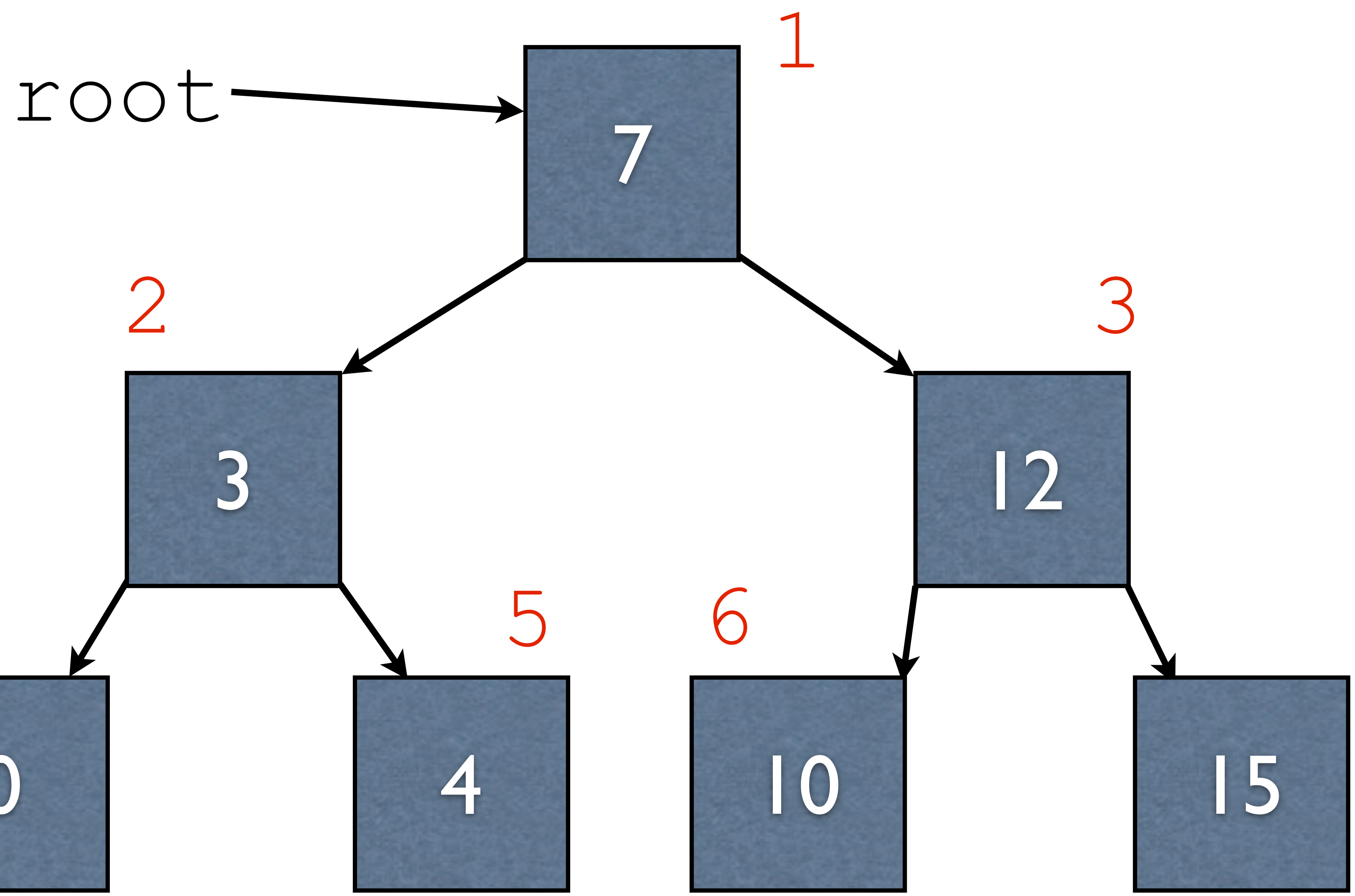

Queue: 4, 10, 15

# Implementing BFS



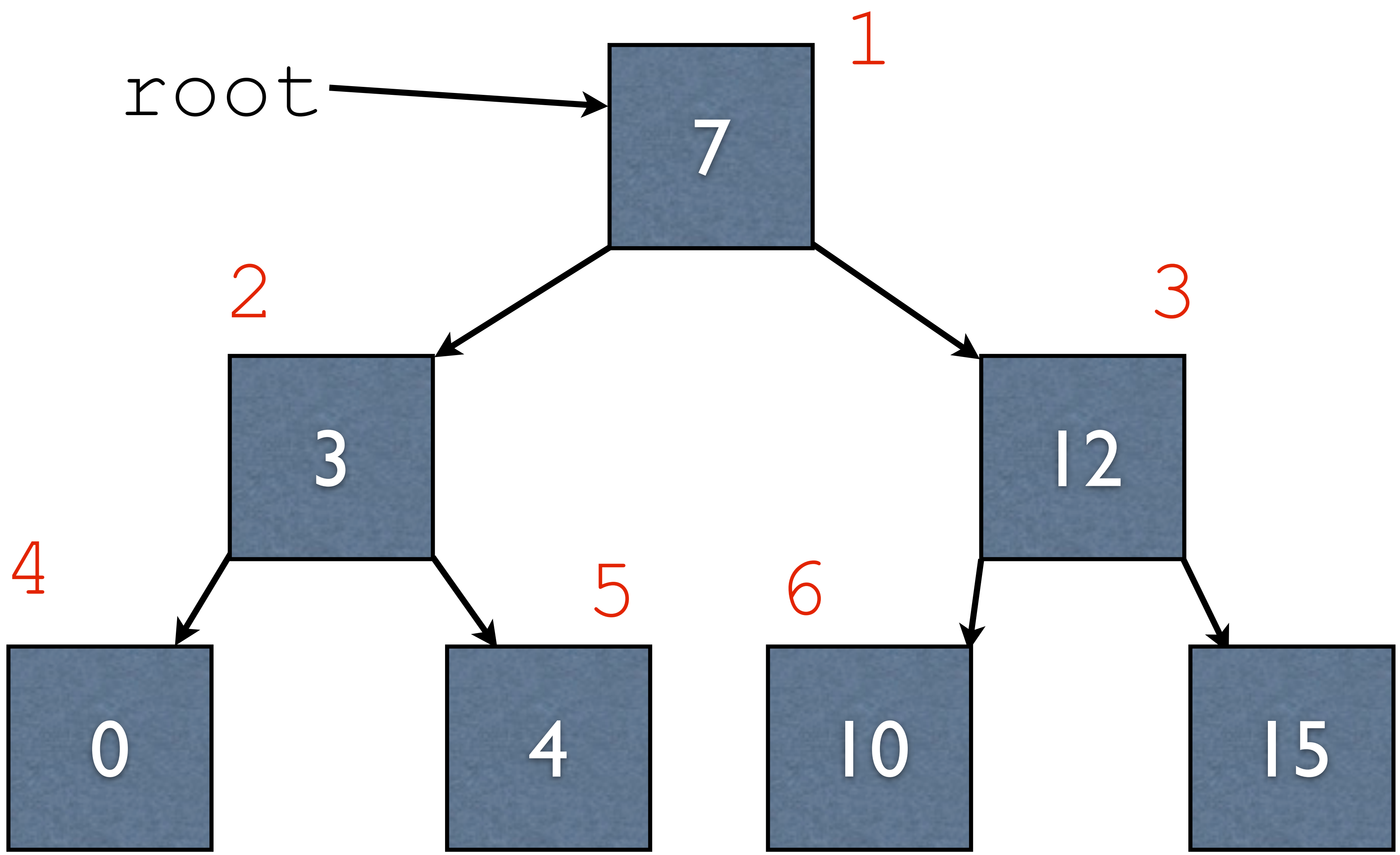

**Queue:** `10, 15`

# Implementing BFS



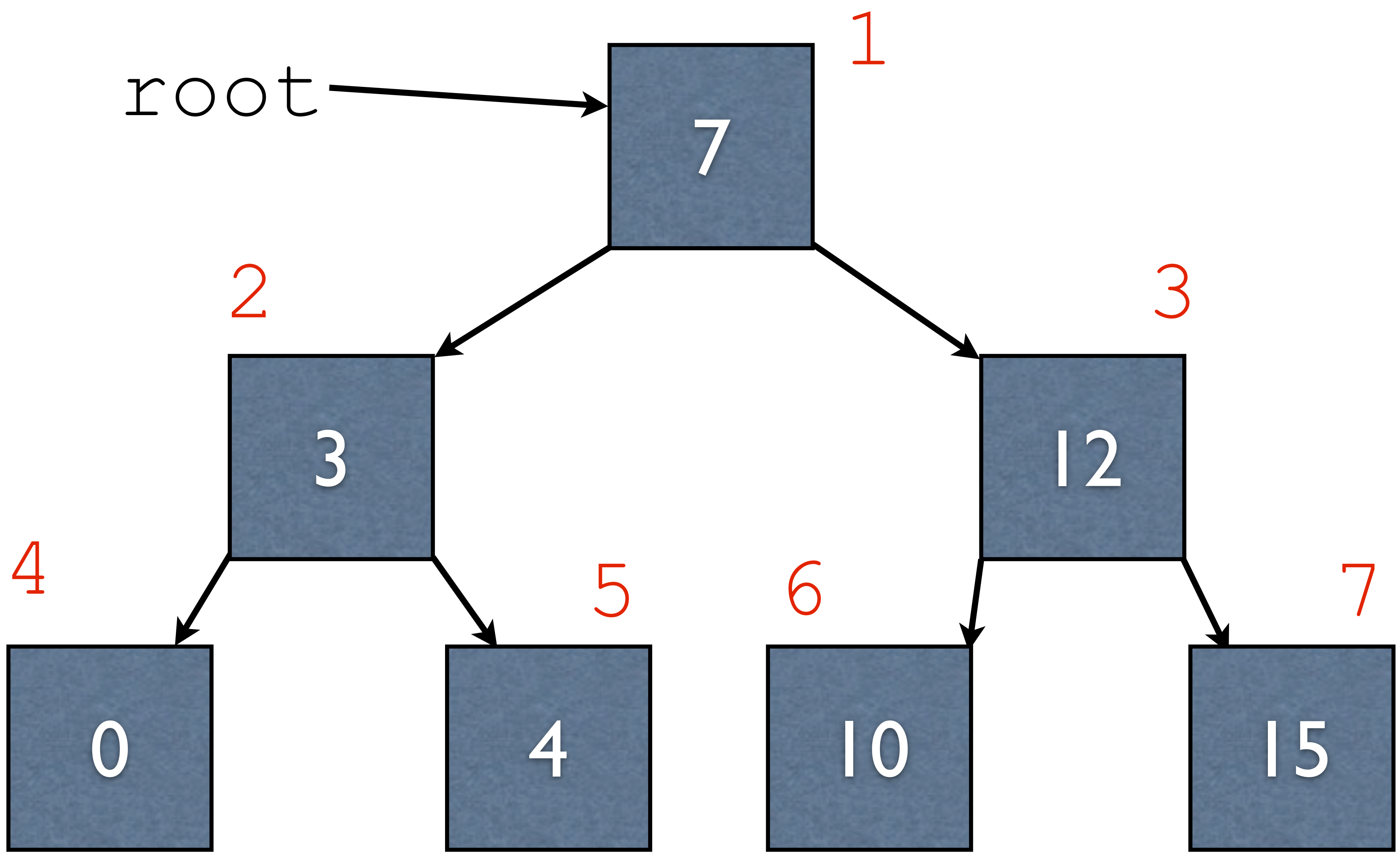

Queue: 10, 15

# Implementing BFS



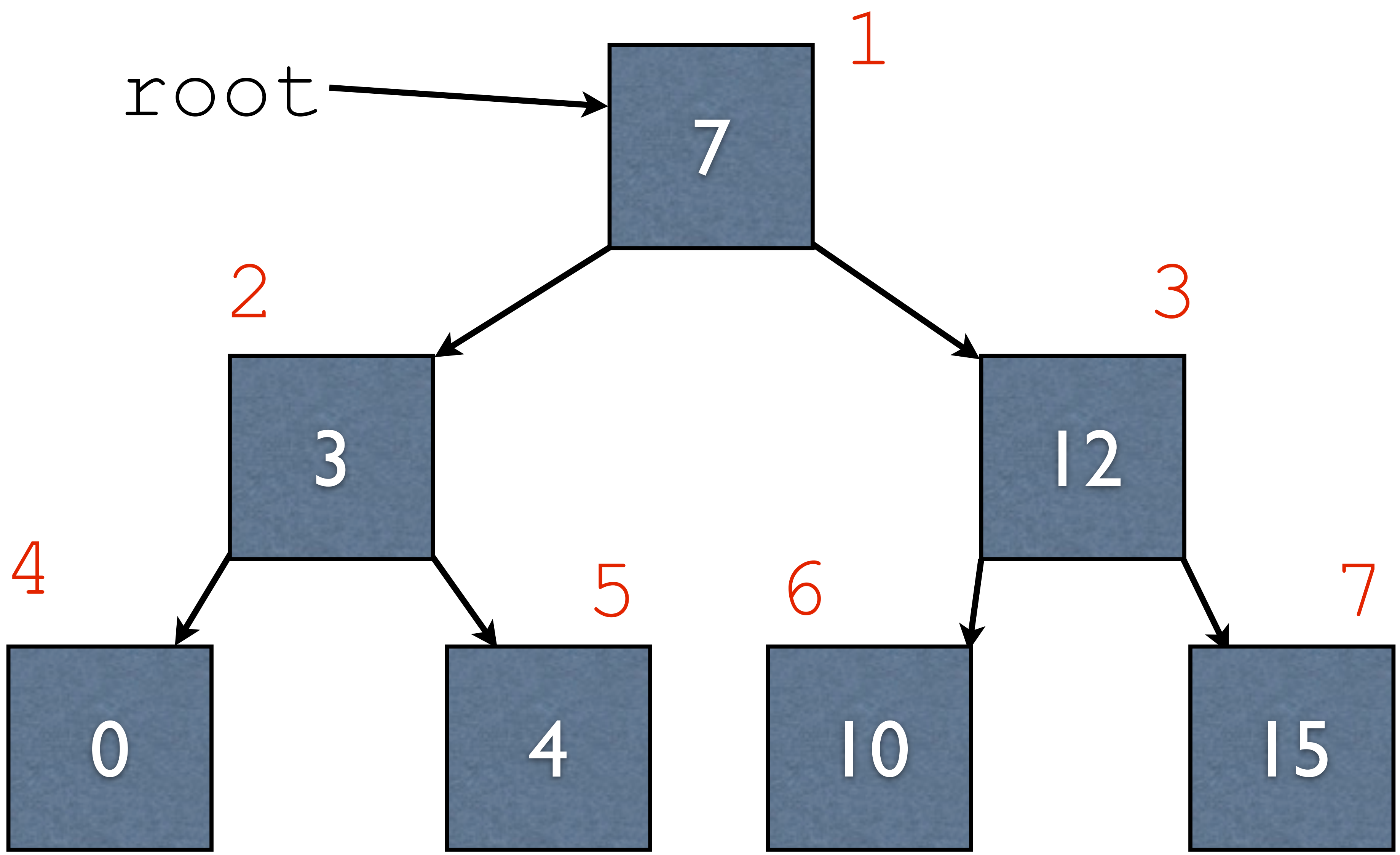

**Queue:** `15`

# Implementing BFS




Queue: 15

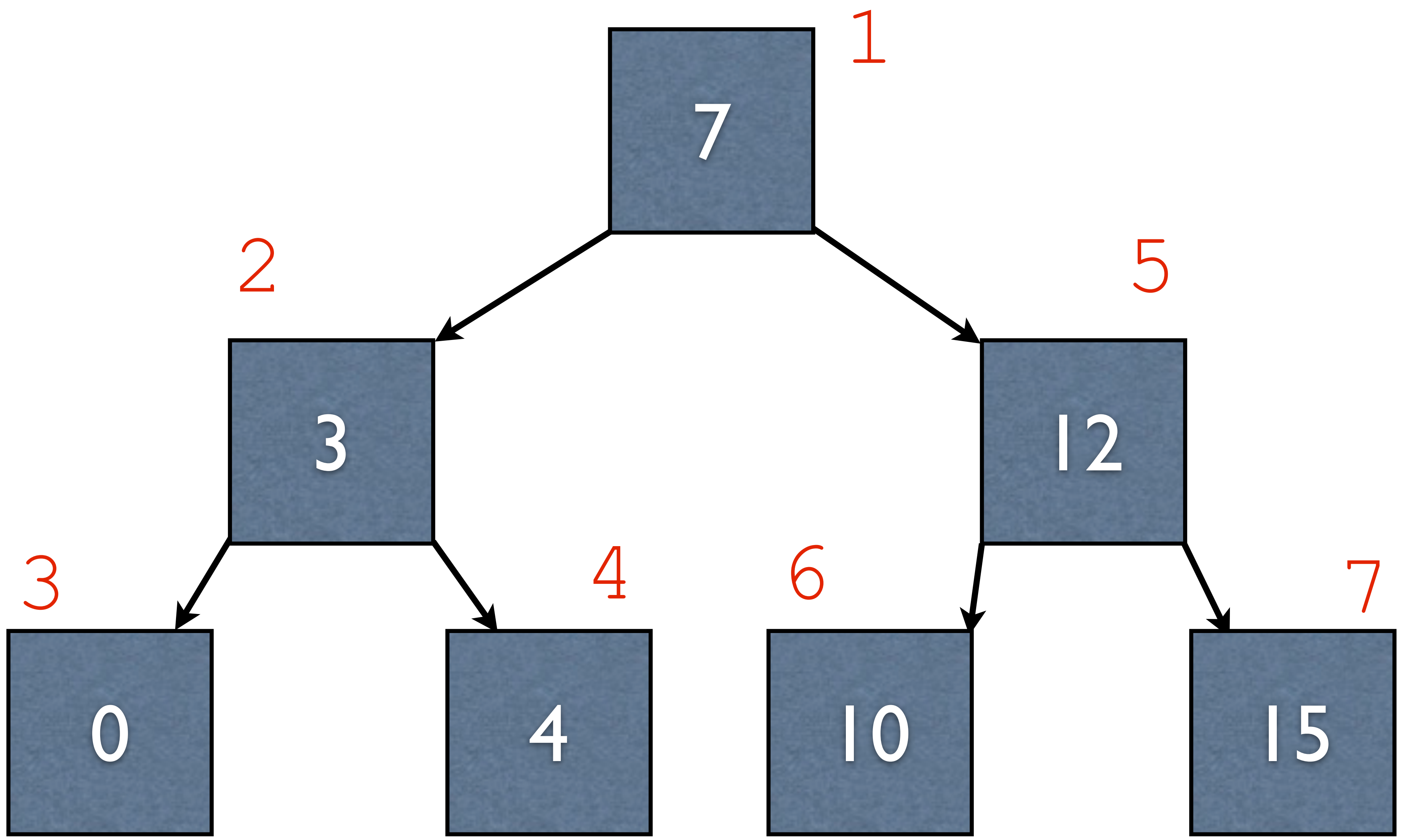
# Implementing BFS




Queue: <<empty>>

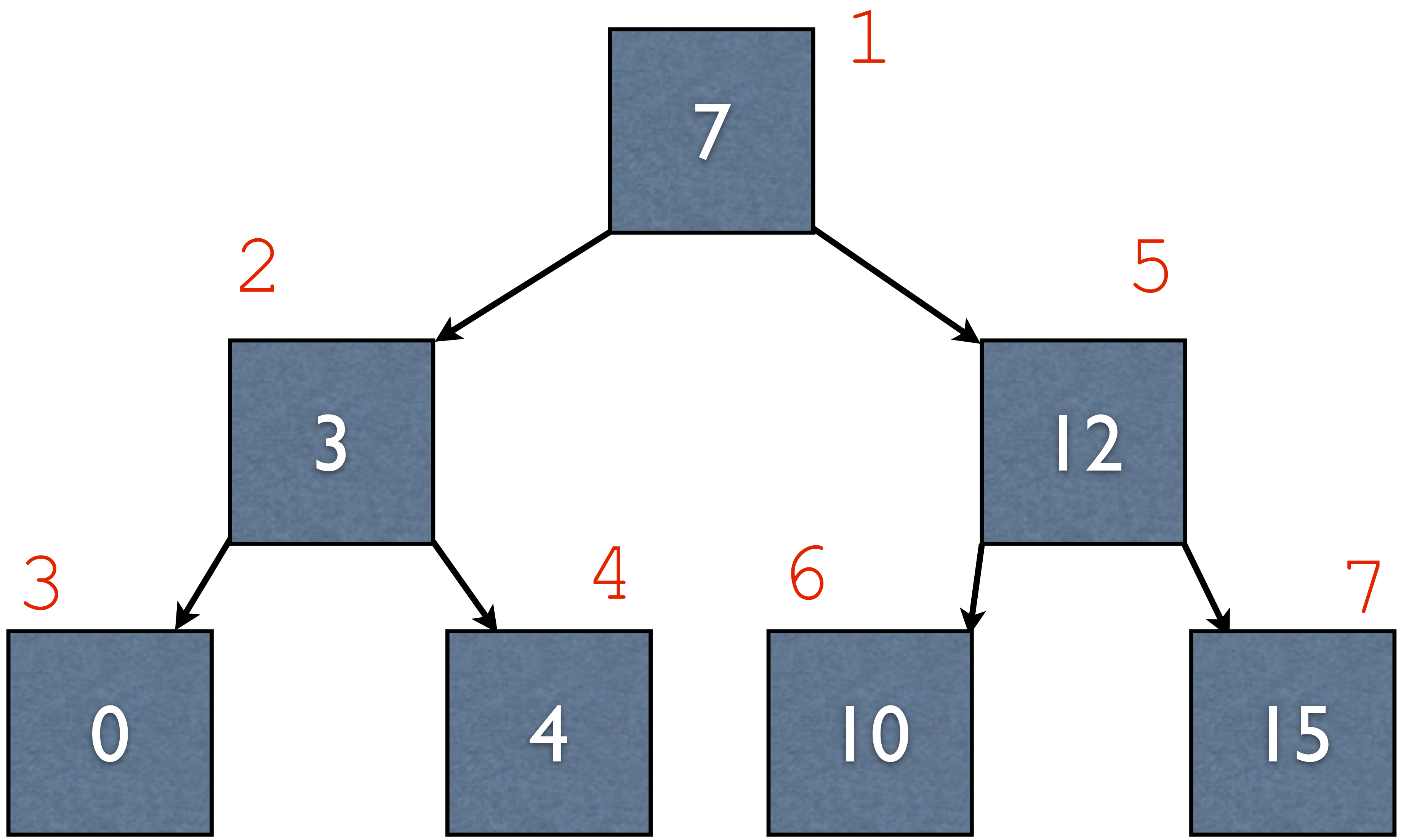
# Depth-First Search (DFS)

- Favor going down towards the left first



# Implementing DFS

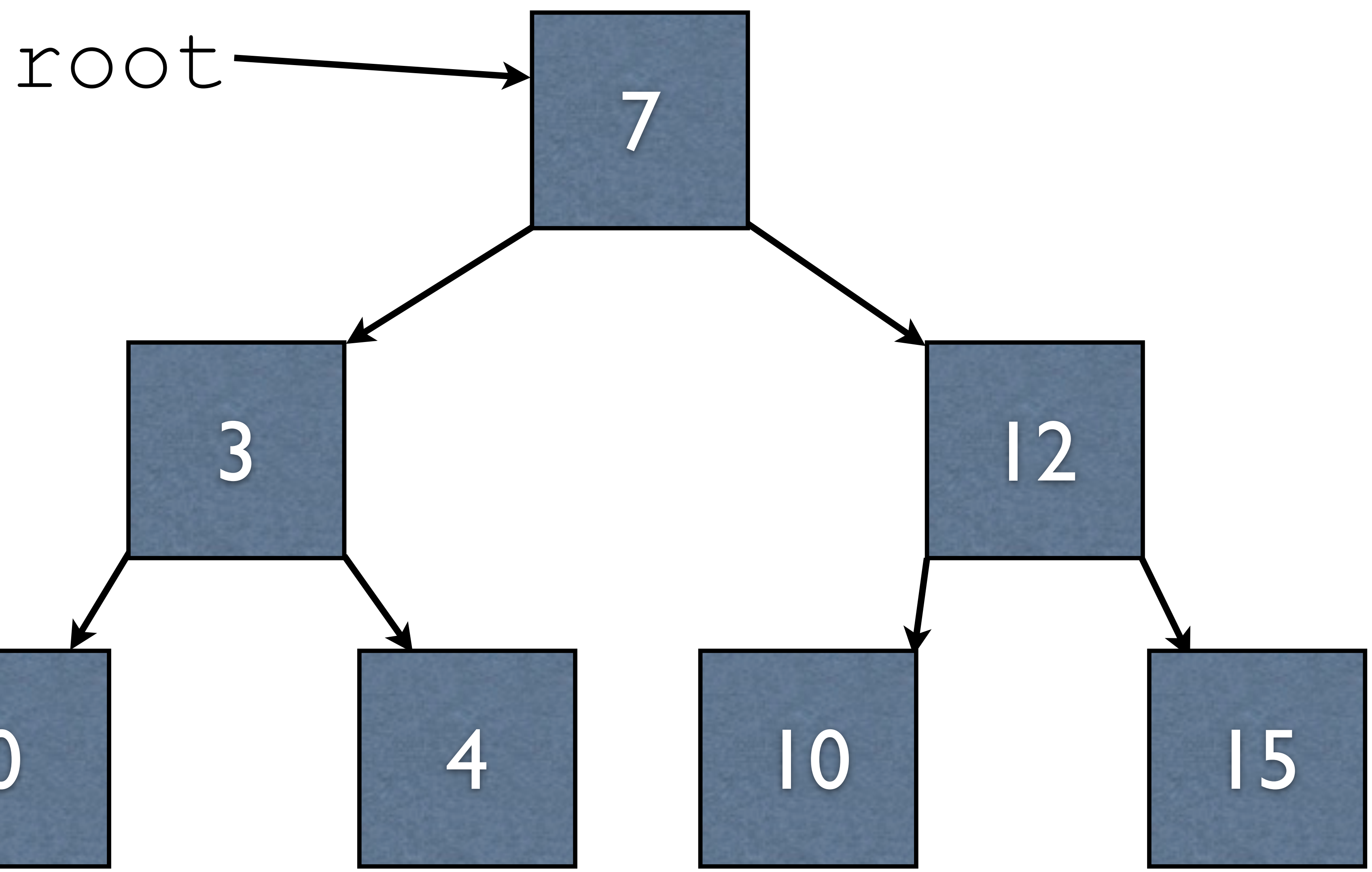
- Question: how might we implement DFS?
  - Hint: you'll need a data structure you've implemented before



# Implementing DFS

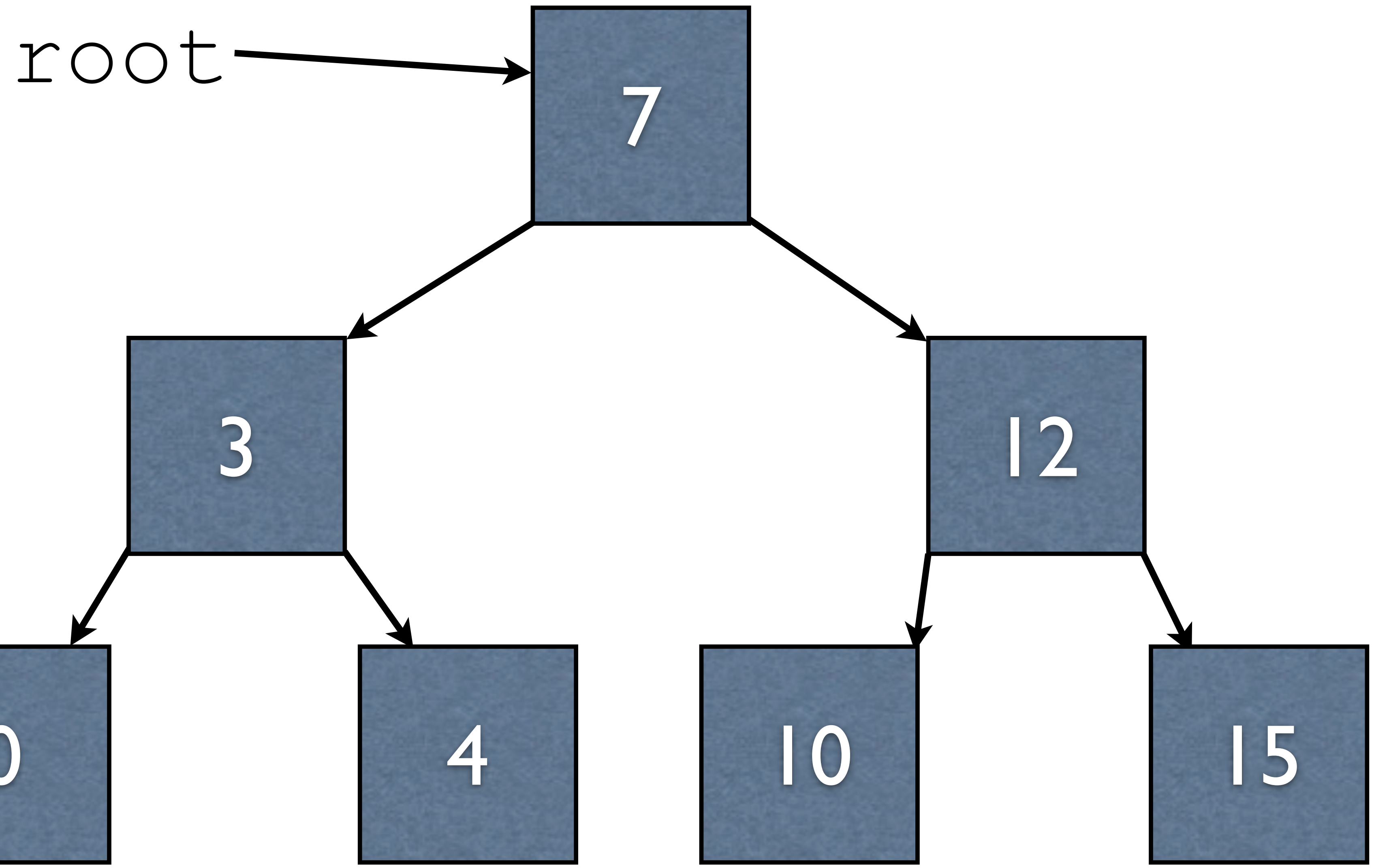
- Idea: put nodes on a **stack**
- Visit nodes according to the stack order
- When we are done with a node, push its children onto the top of the stack

# Implementing DFS



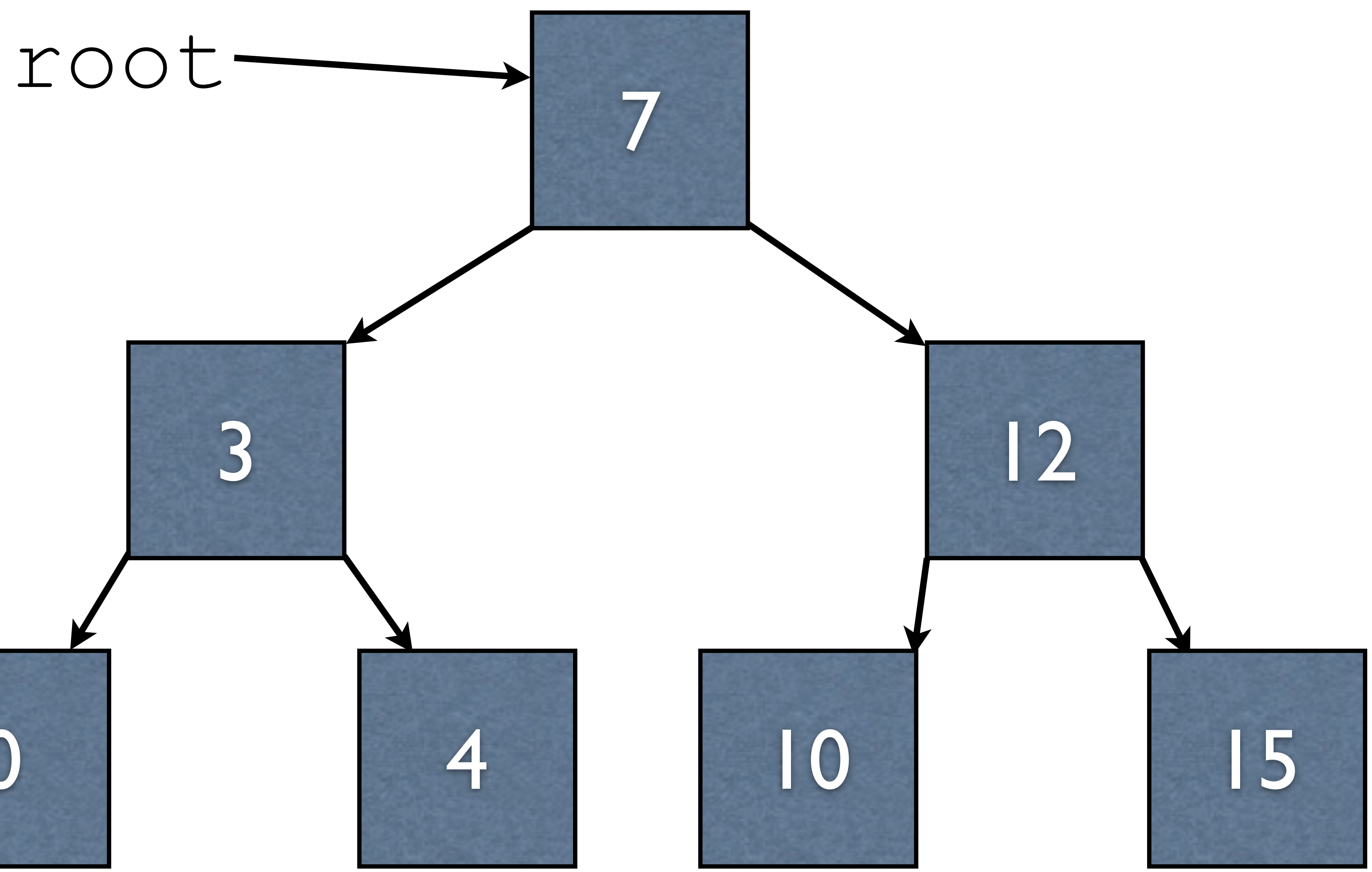

Stack: `<<empty>>`

# Implementing DFS



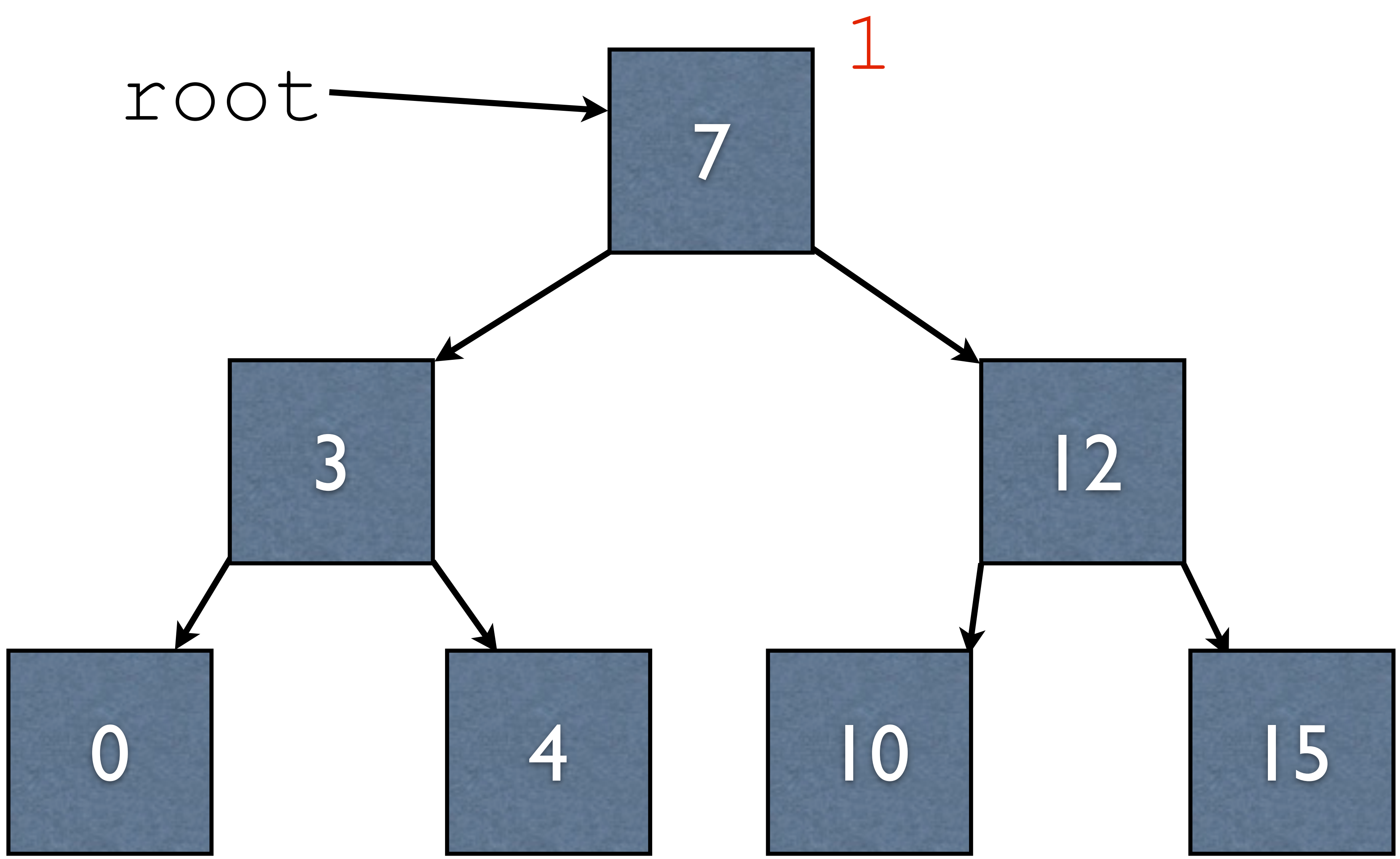

Stack: 7

# Implementing DFS



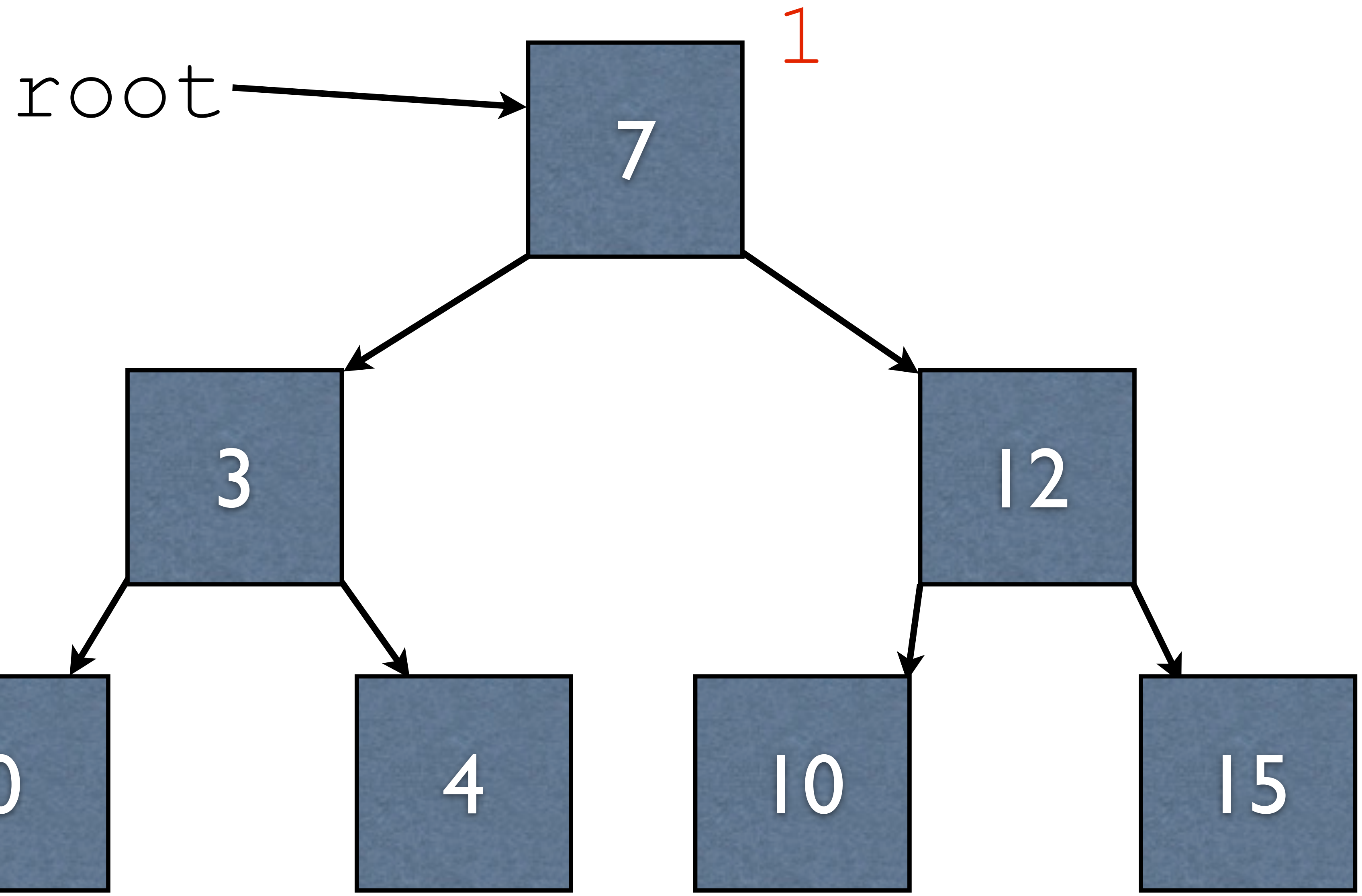

Stack: 7

# Implementing DFS



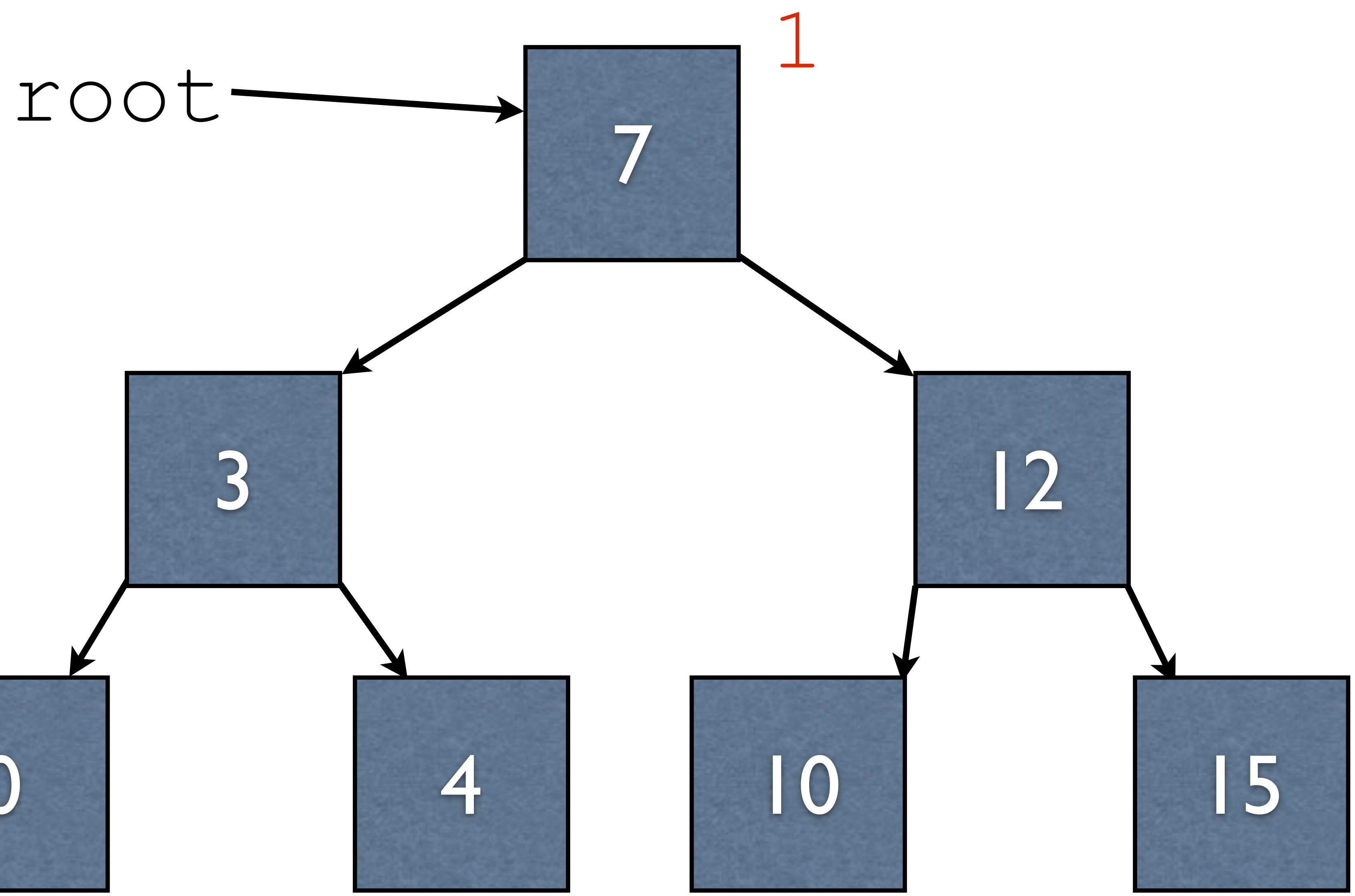

Stack: 7

# Implementing DFS



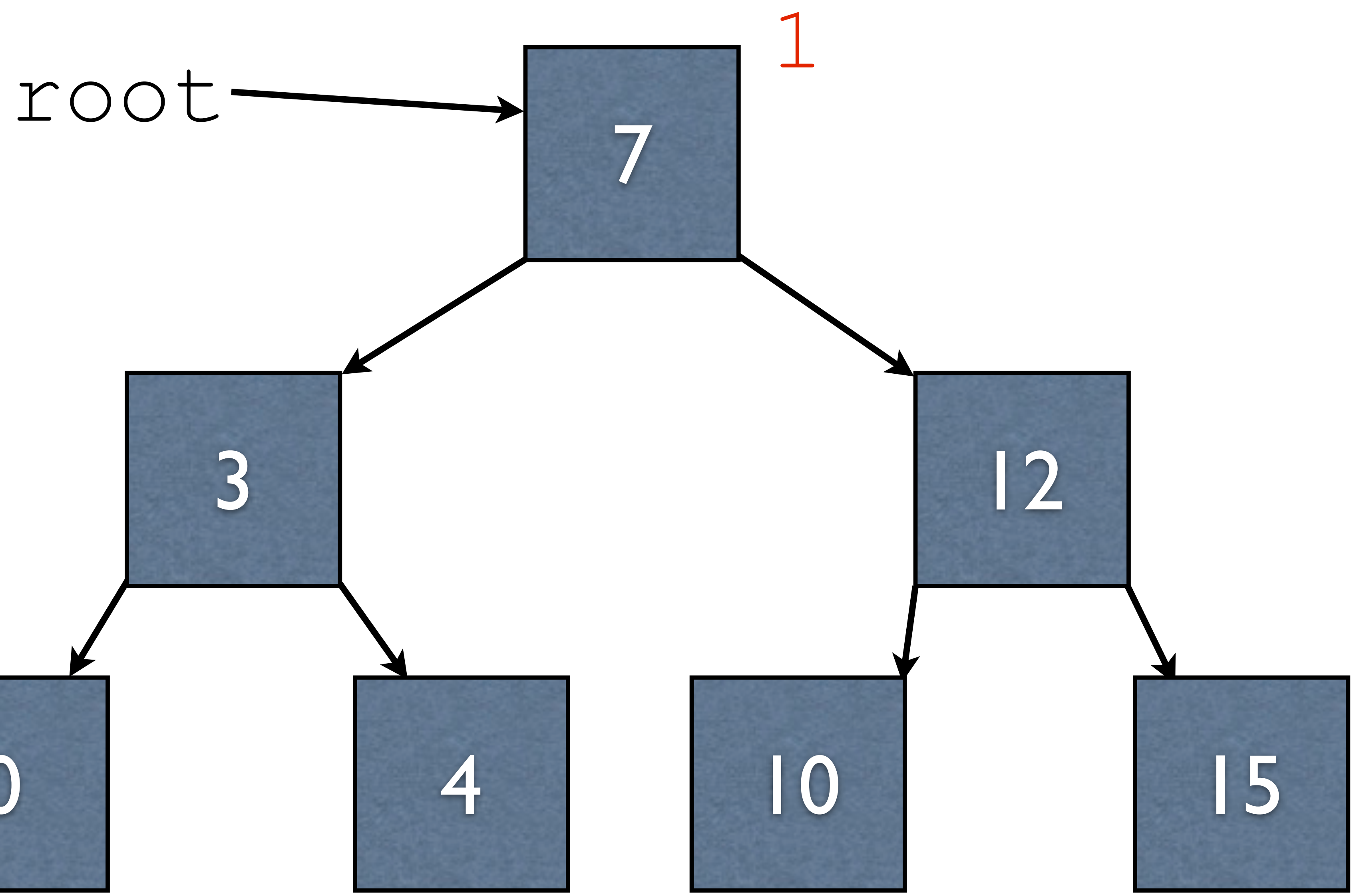

**Stack:** <<empty>>

# Implementing DFS



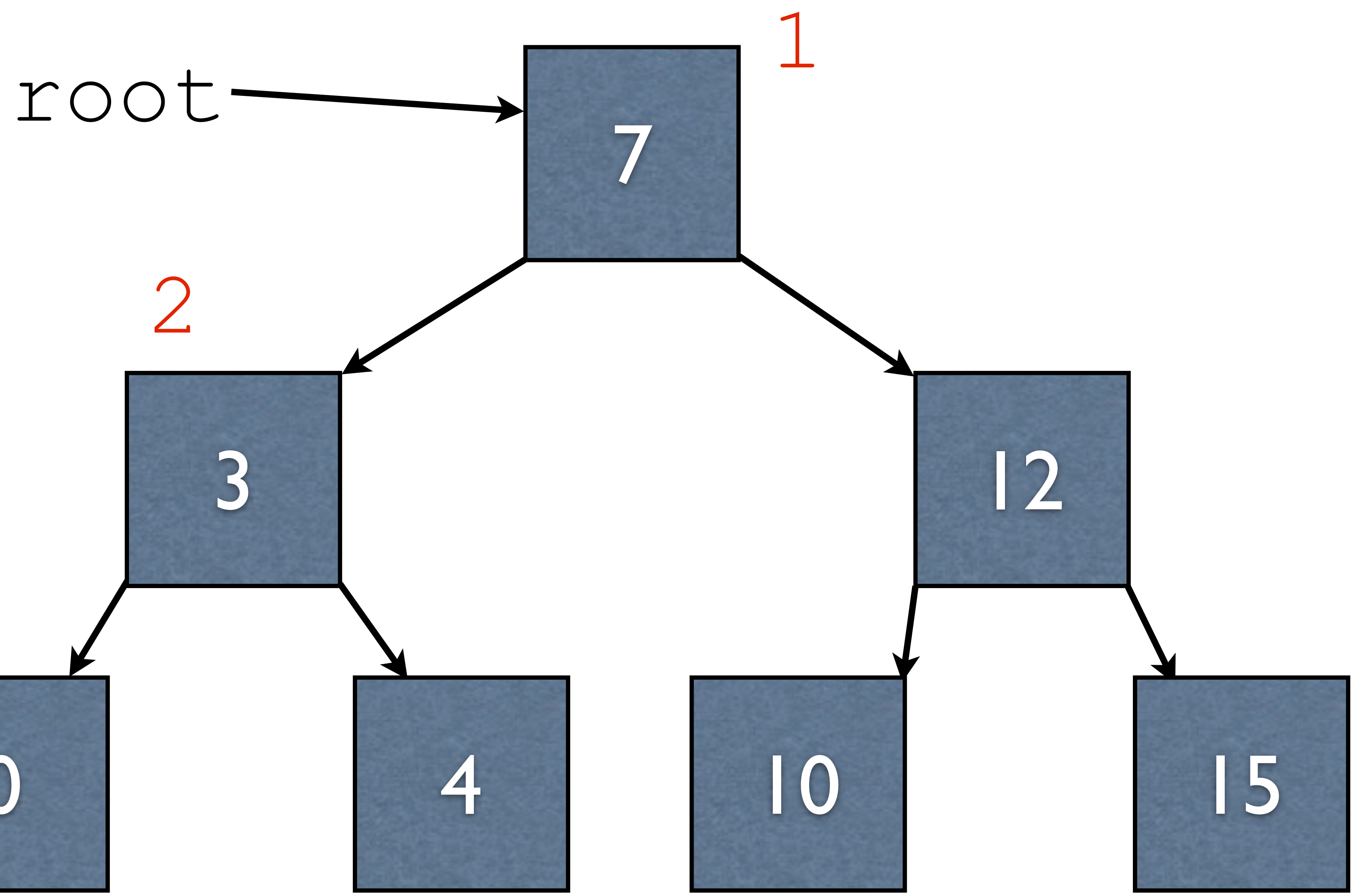

Stack: 3, 12

# Implementing DFS



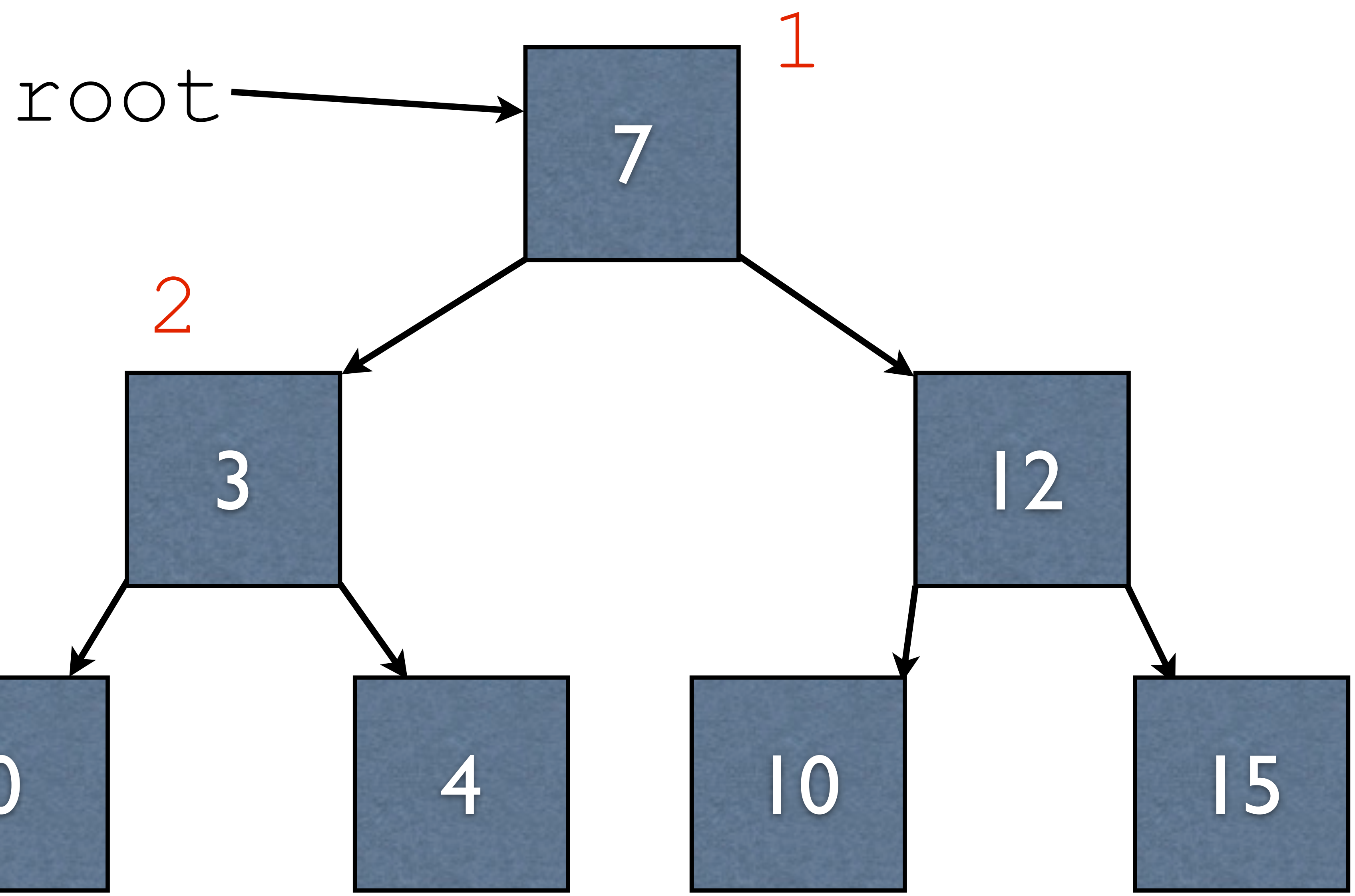

**Stack:** `3, 12`

# Implementing DFS



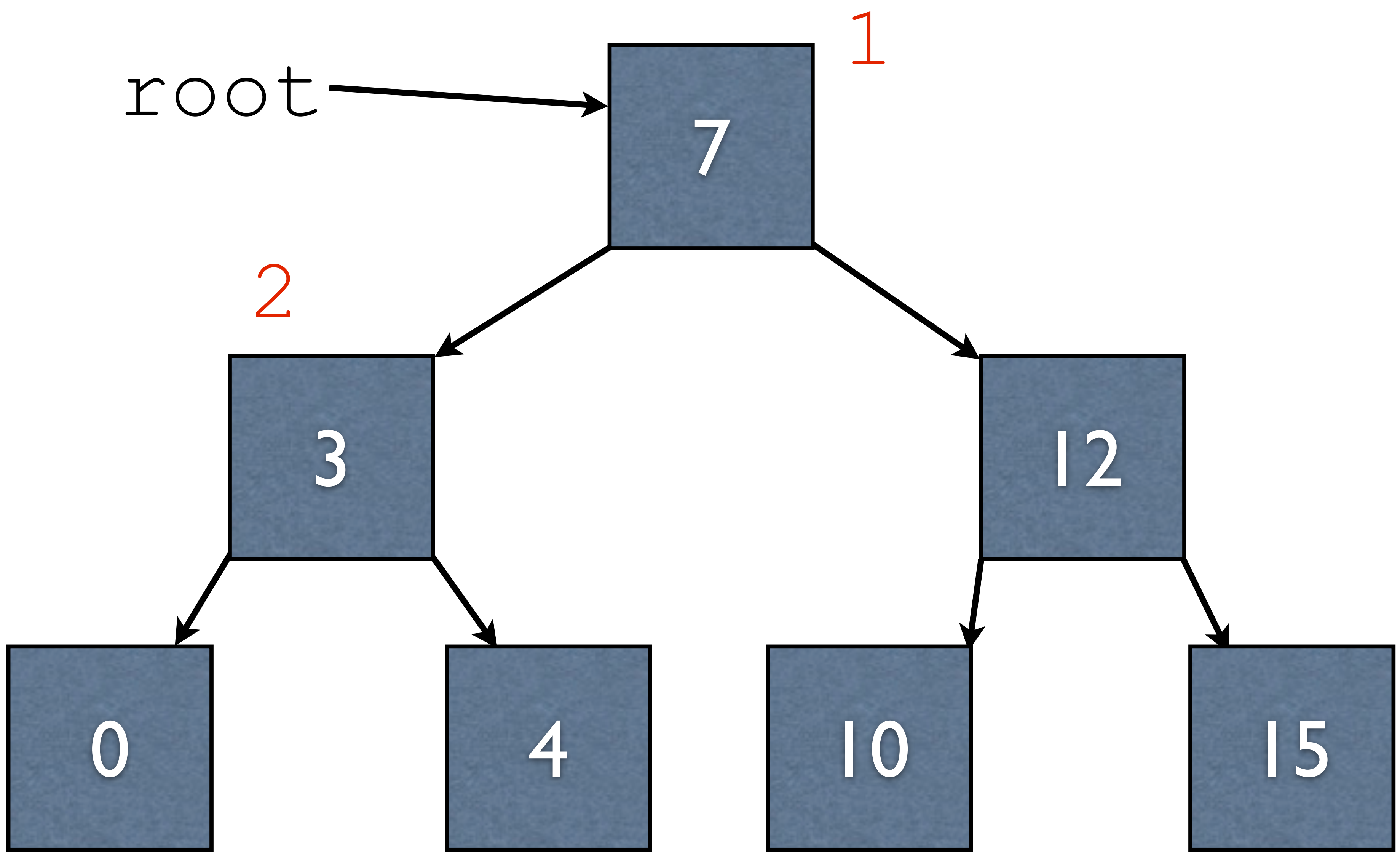

Stack: 3, 12

# Implementing DFS



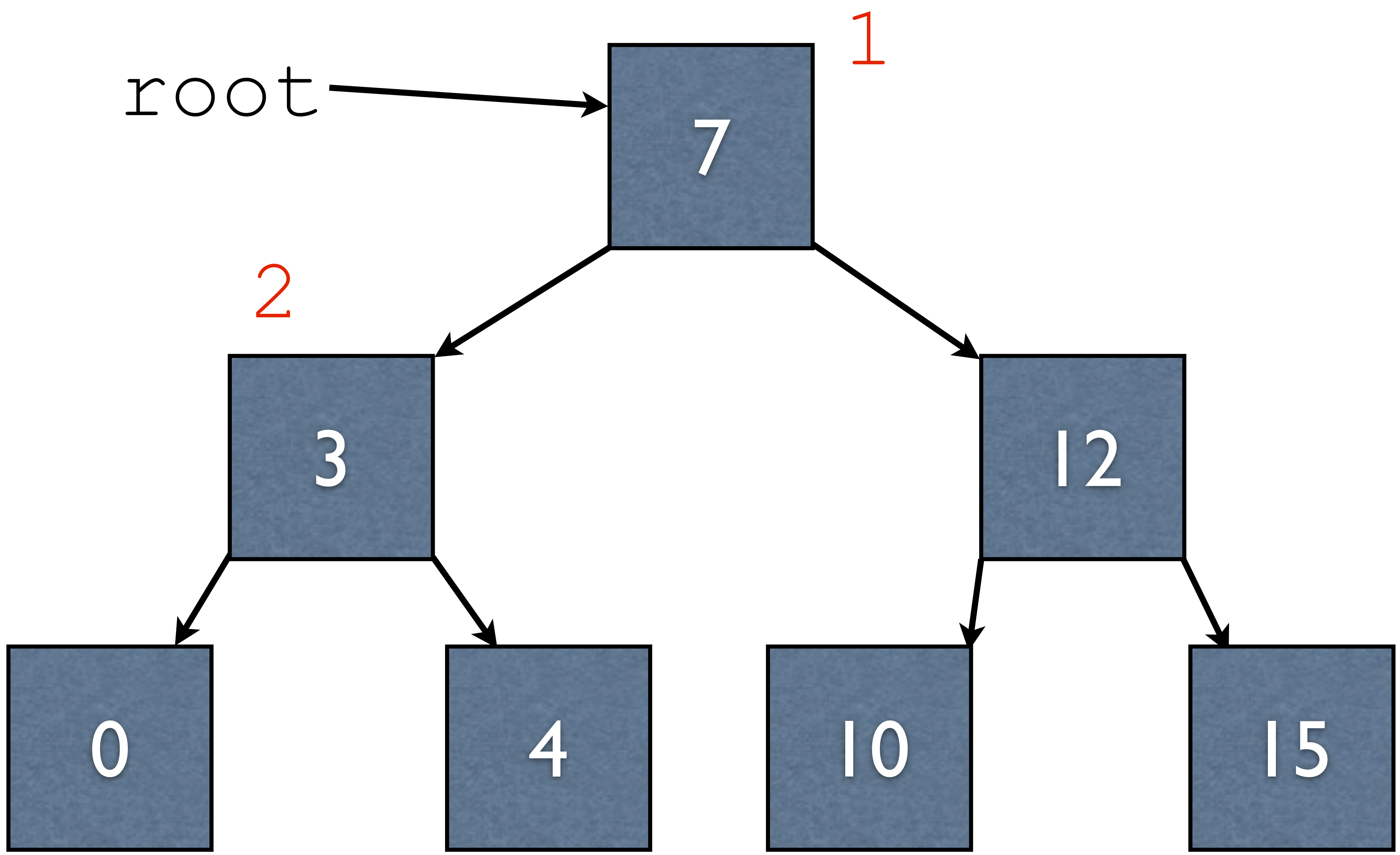

**Stack:** `12`

# Implementing DFS



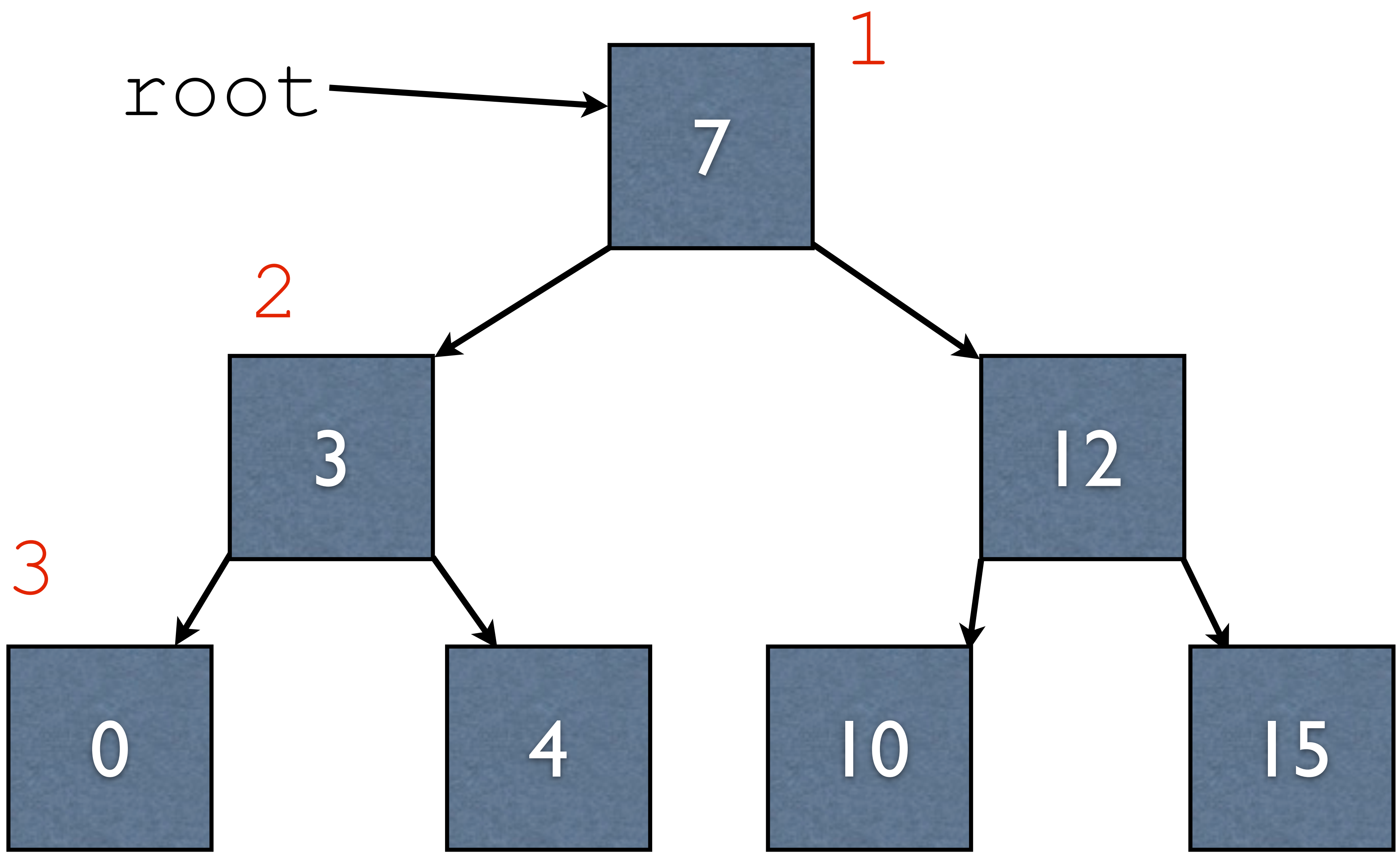

Stack: 0, 4, 12

# Implementing DFS



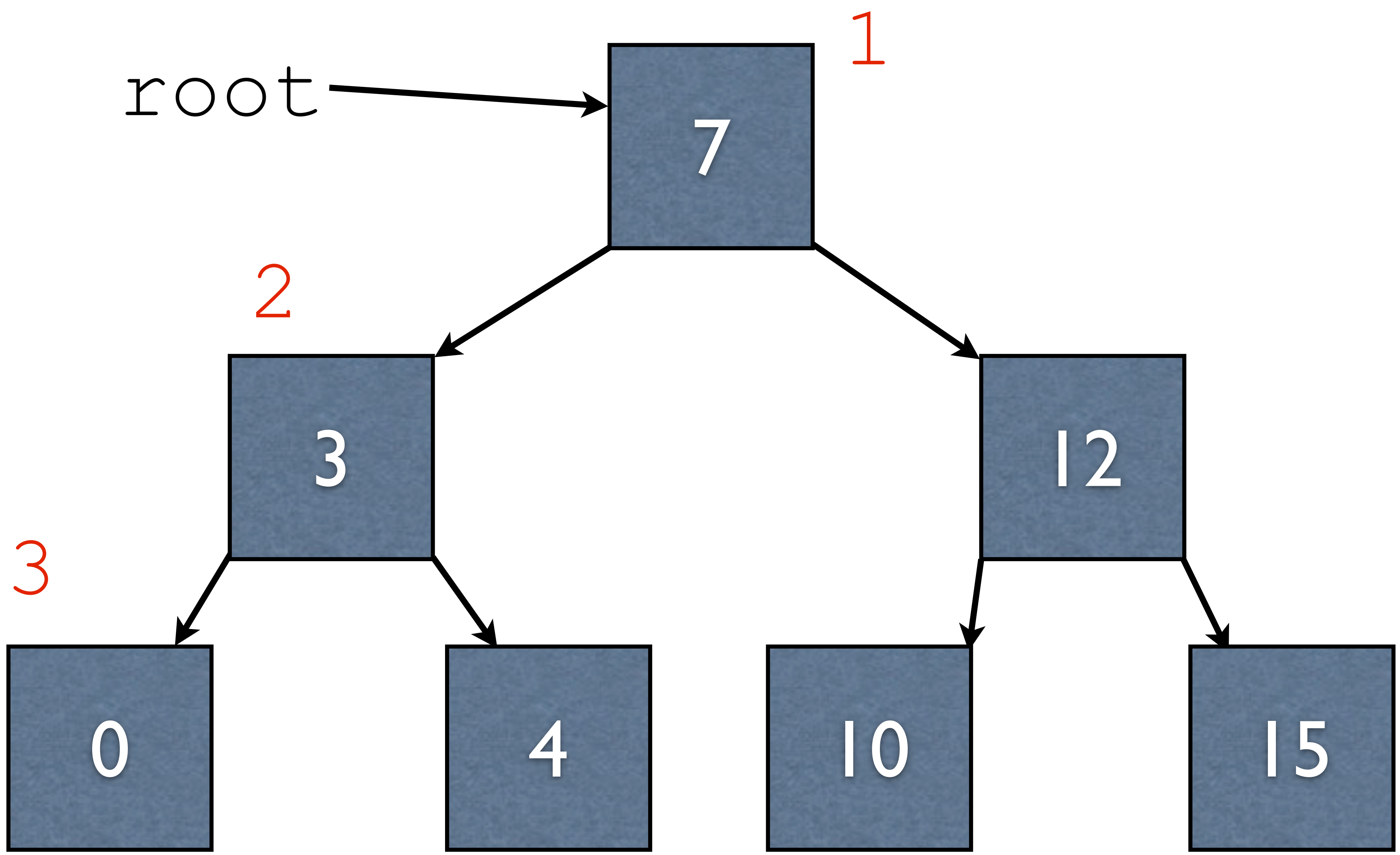

**Stack:** `0, 4, 12`

# Implementing DFS



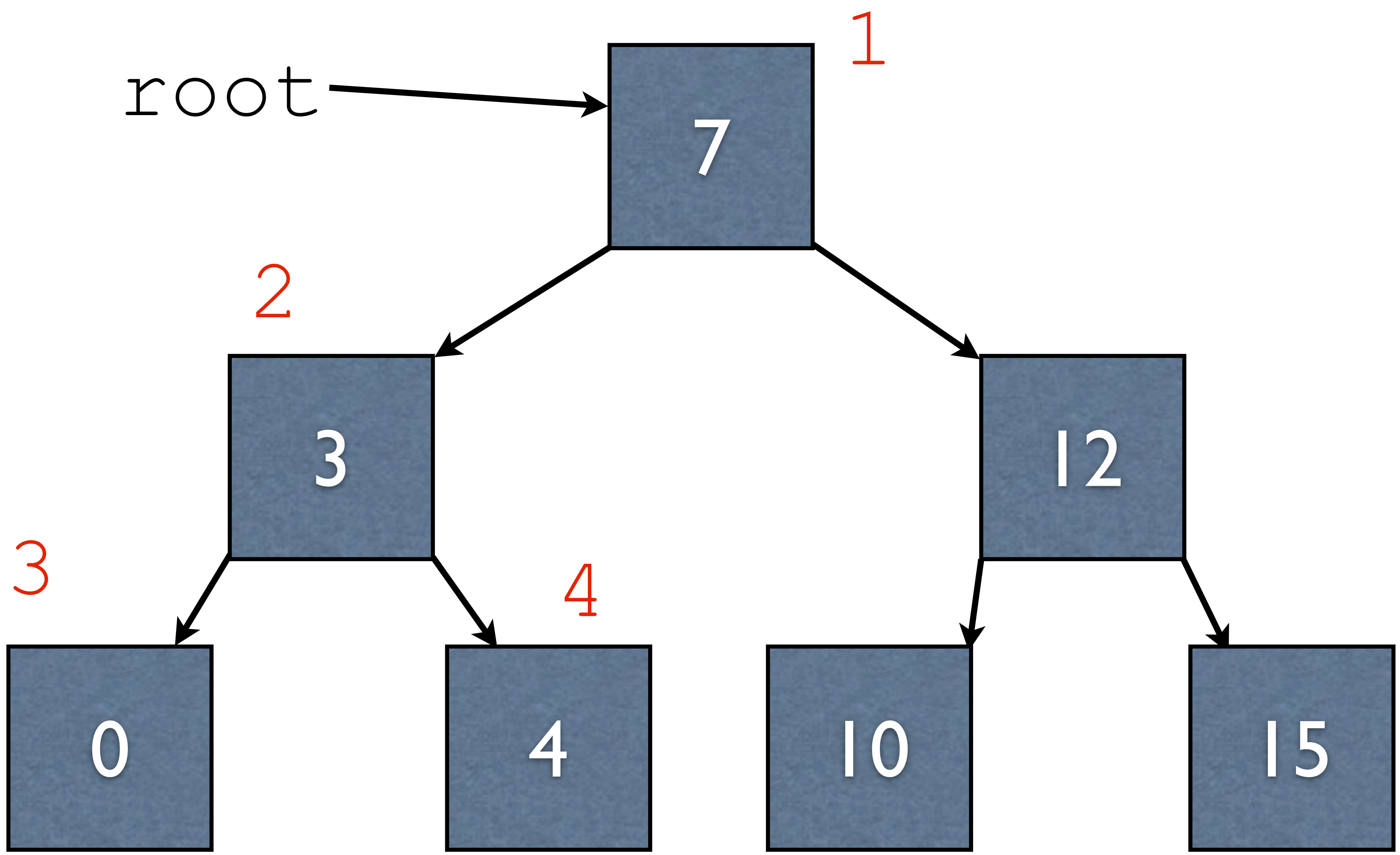

Stack: 0, 4, 12

# Implementing DFS



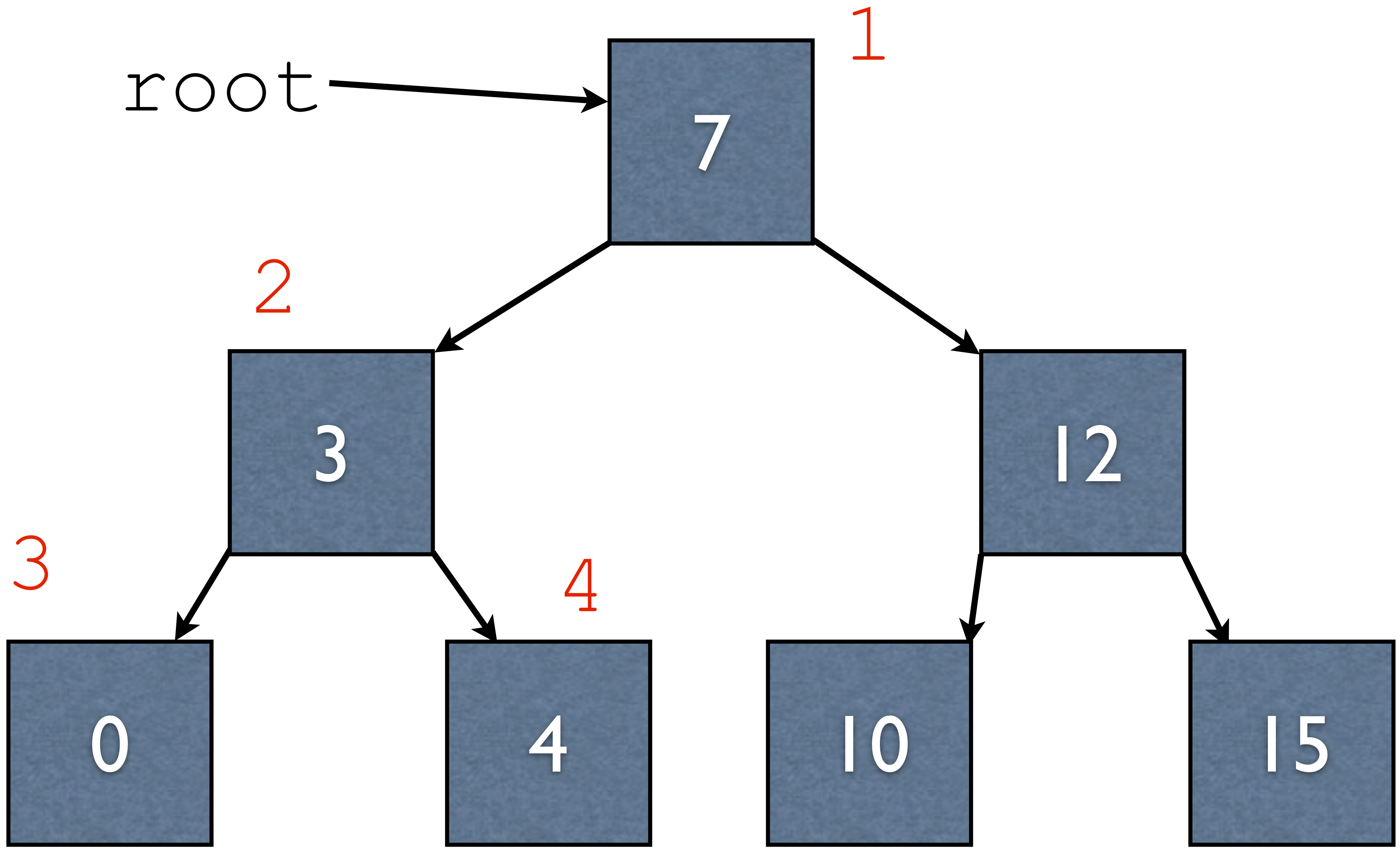

**Stack:** `4, 12`

# Implementing DFS



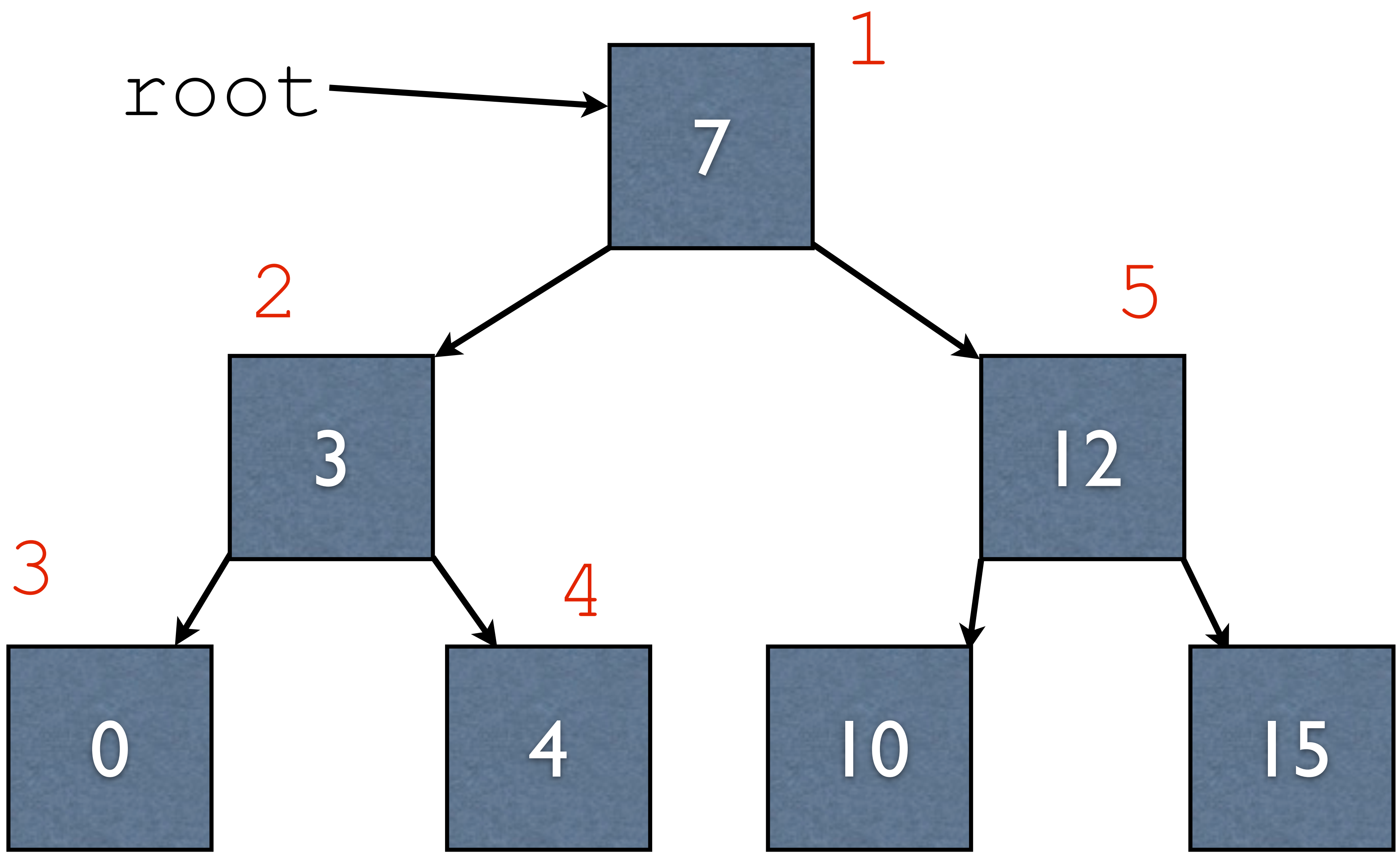

Stack: 4, 12

# Implementing DFS



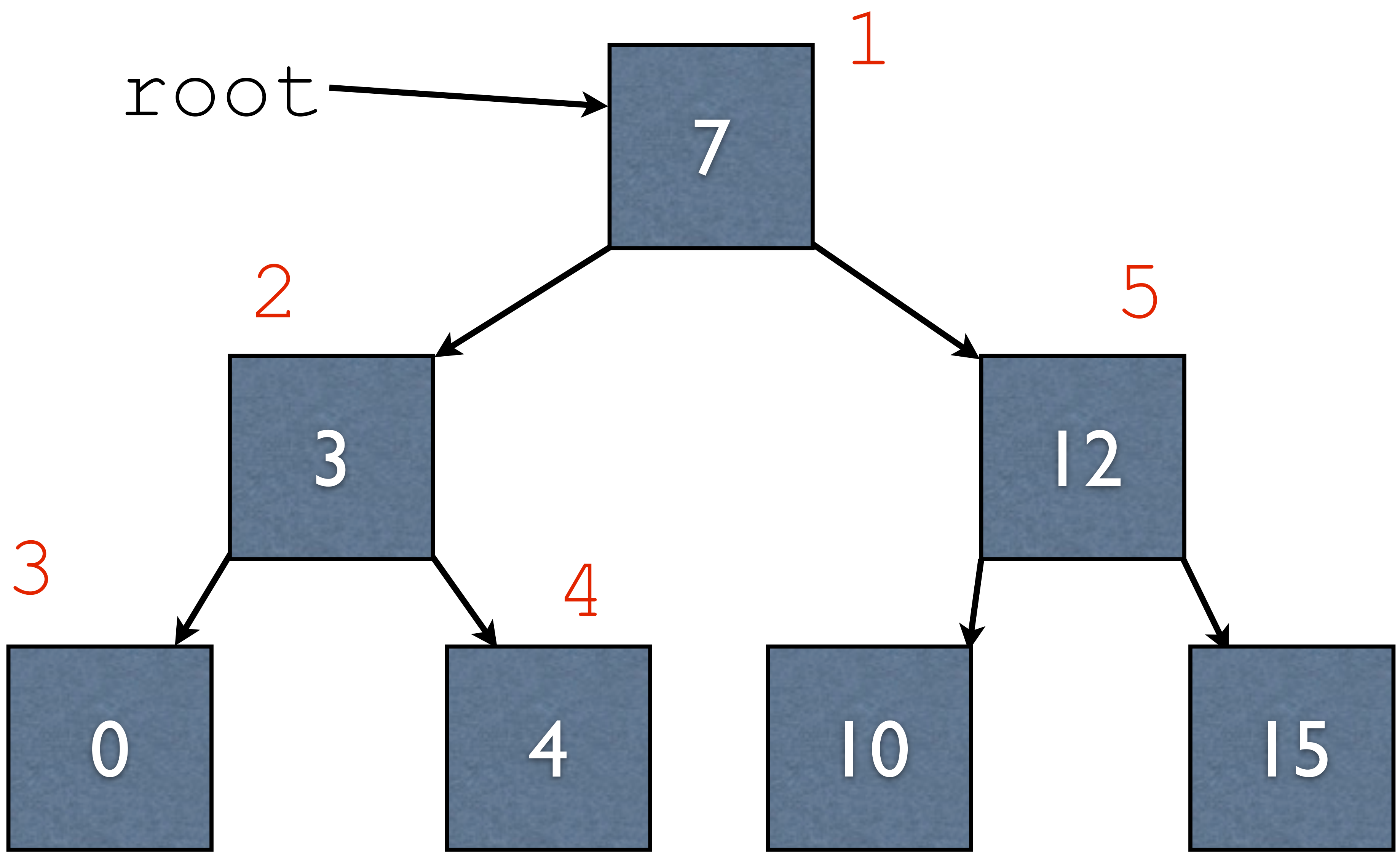

**Stack:** 12

# Implementing DFS



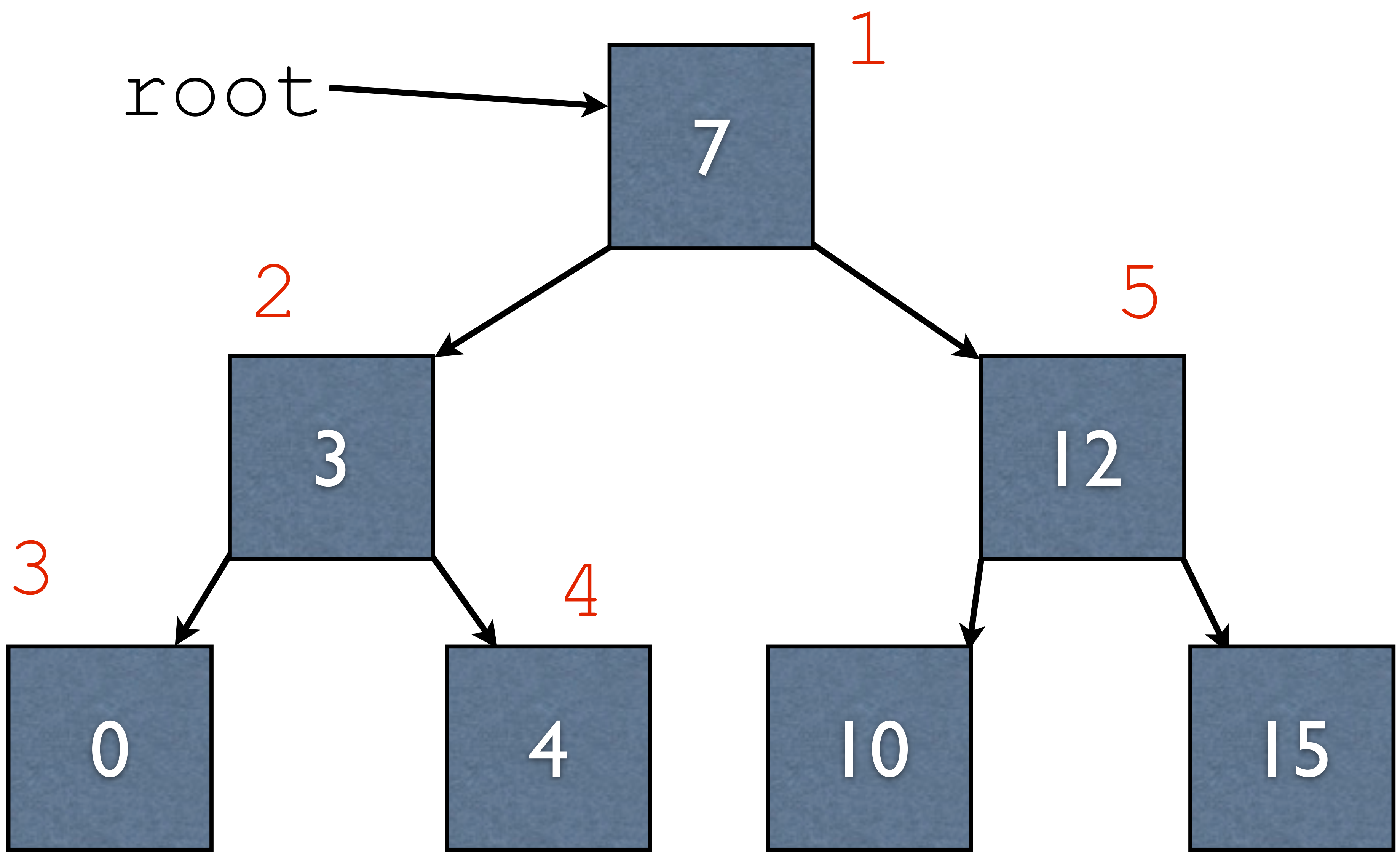

Stack: 12

# Implementing DFS



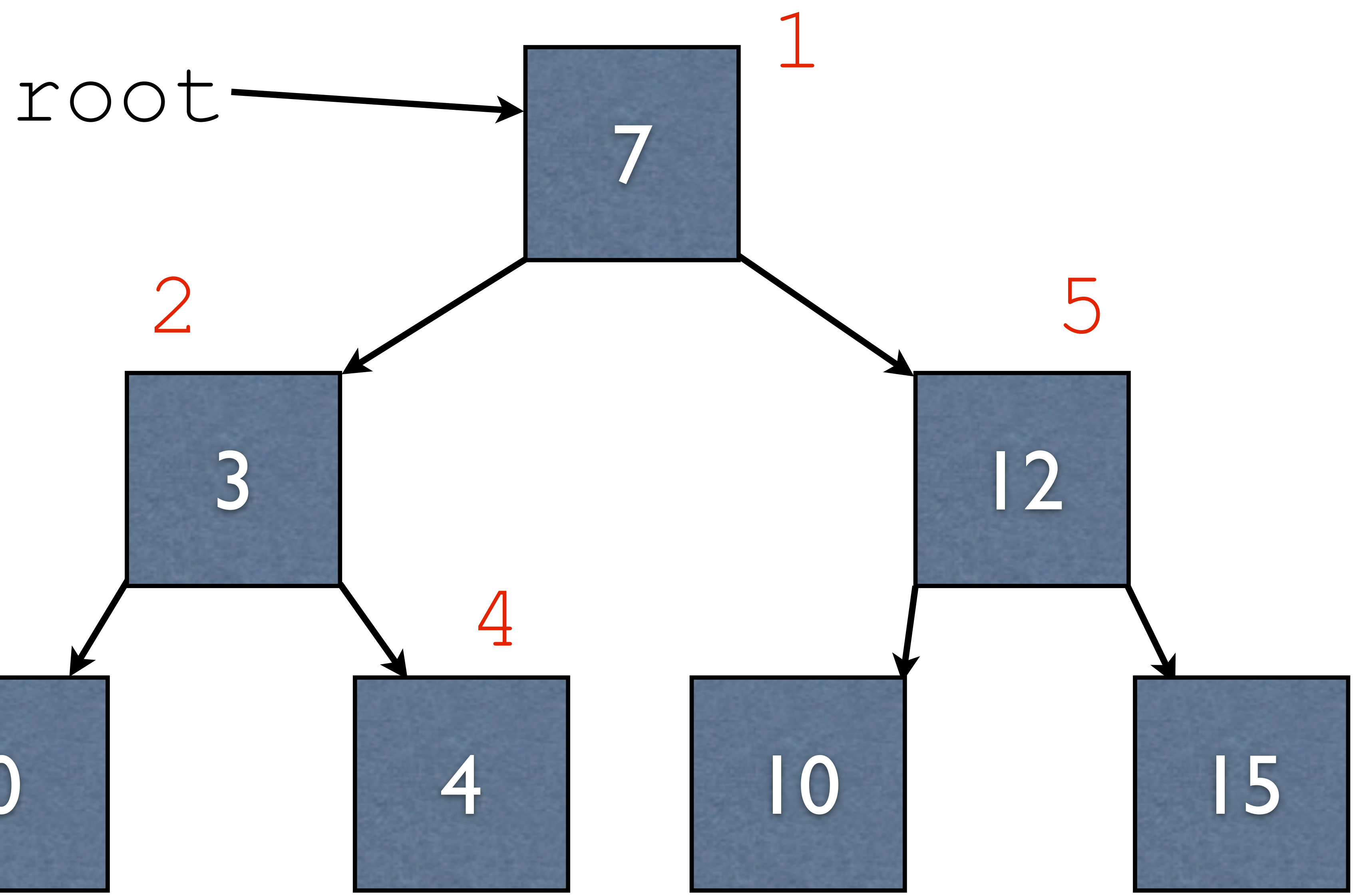

**Stack:** `<<empty>>`

# Implementing DFS



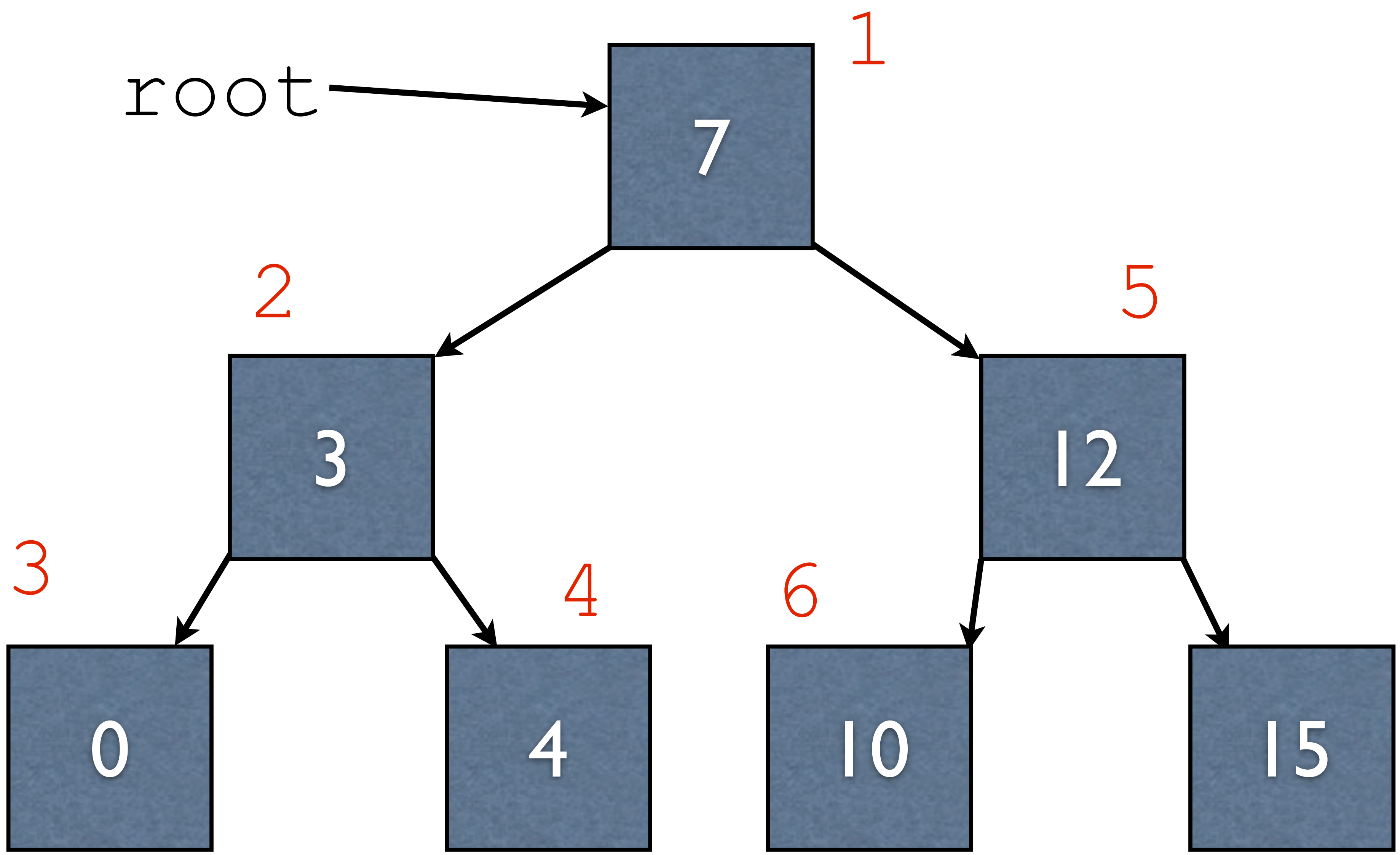

Stack: 10, 15

# Implementing DFS



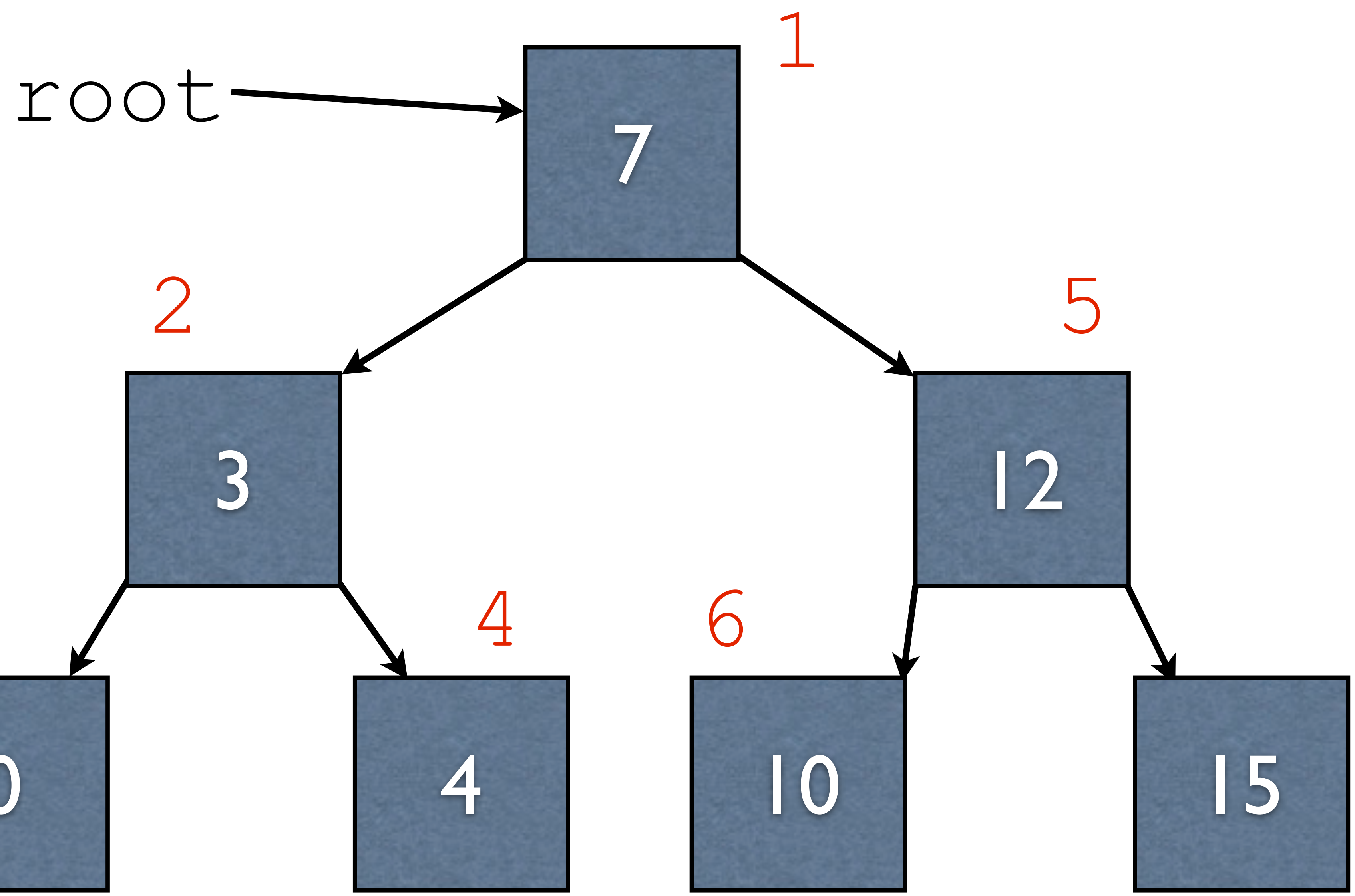

**Stack:** `10, 15`

# Implementing DFS



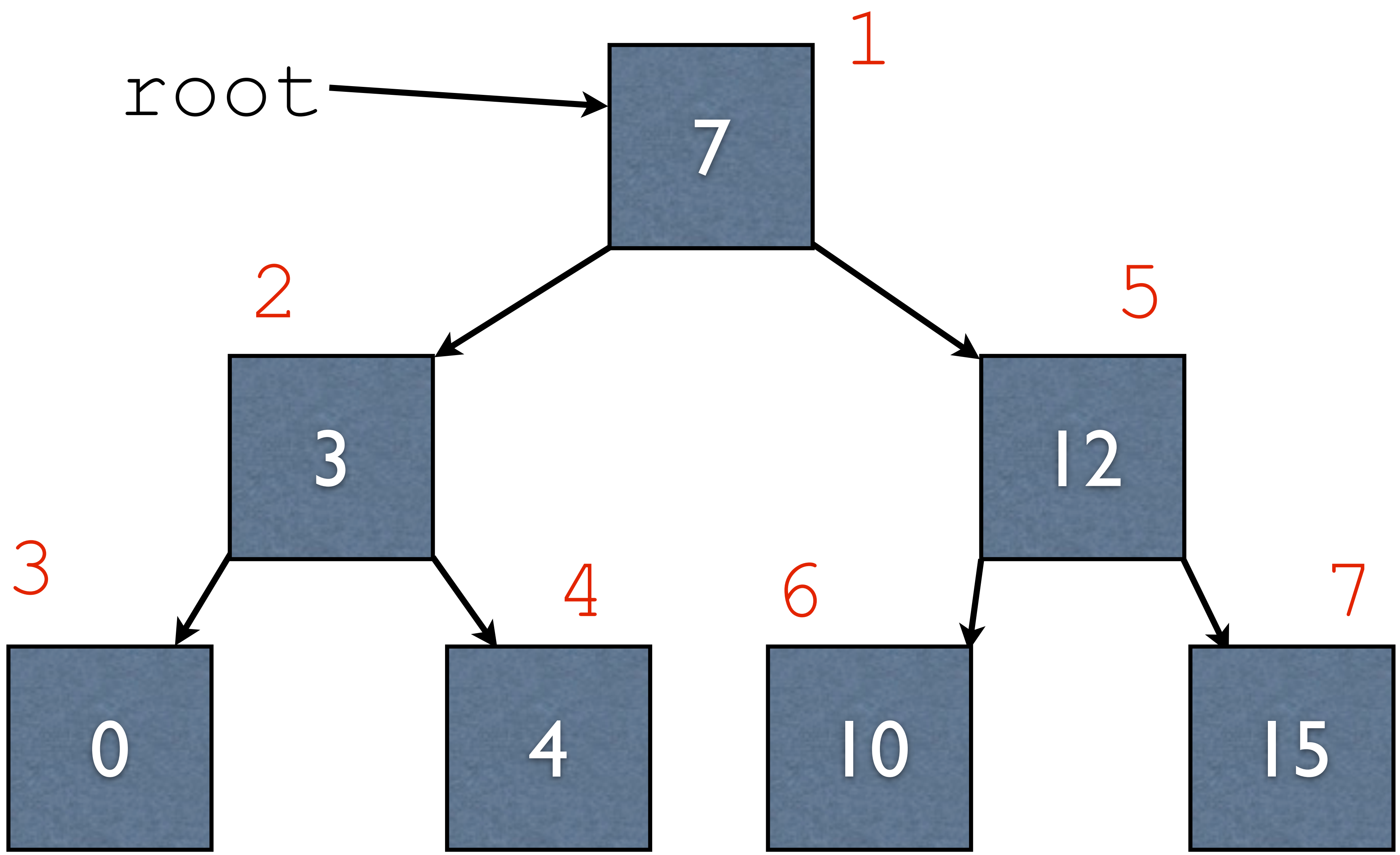

Stack: 10, 15

# Implementing DFS



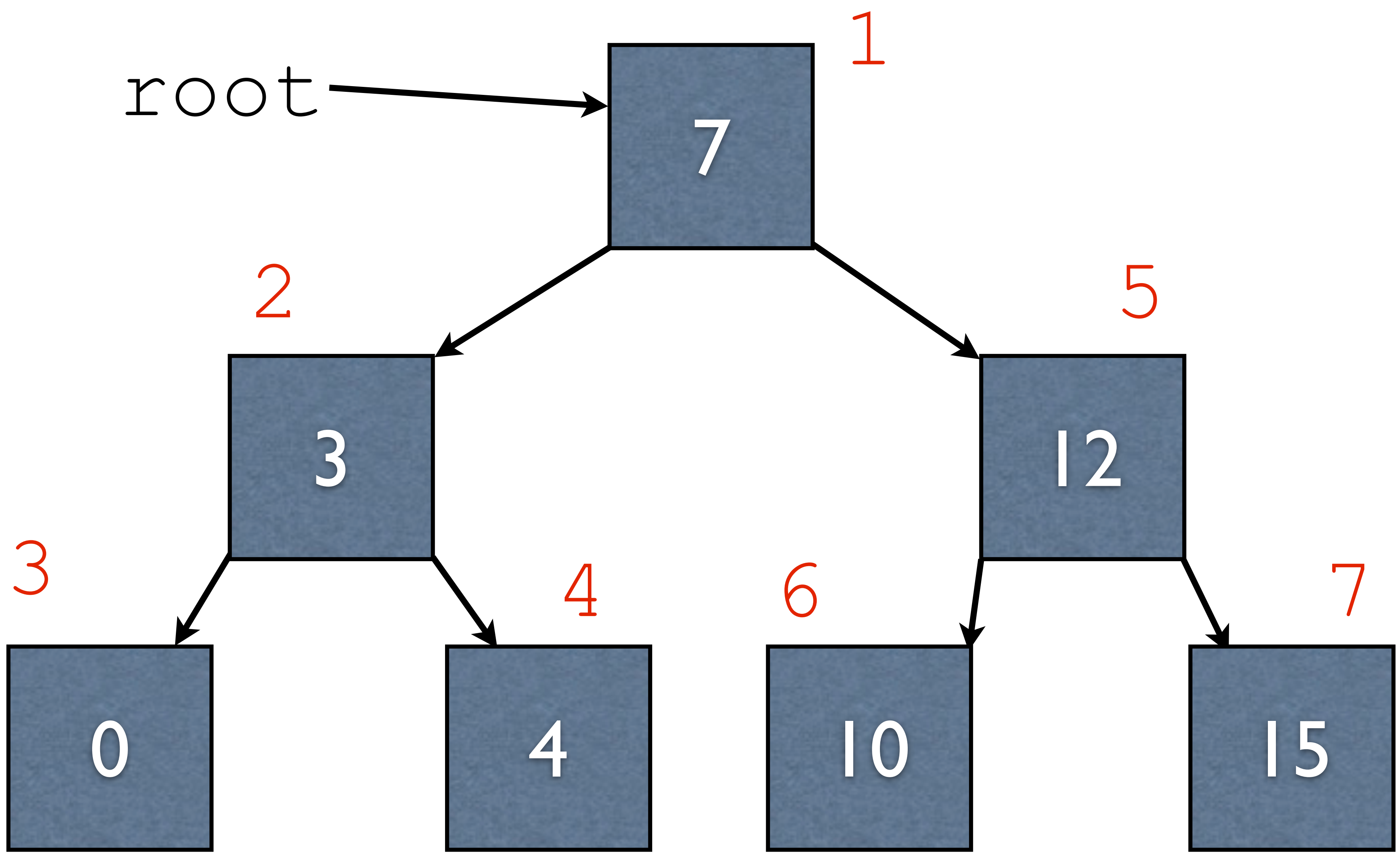

**Stack:** 15

# Implementing DFS




Stack: 15

# Implementing DFS




**Stack:** <<empty>>

# On Using Stacks

- We can cut out the explicit stack by using the call stack implicitly via recursion

```
void traverse(Node* current) {
  if (current != NULL) {
    traverse(current->getLeft());
    traverse(current->getRight());
  }
}
```

# Specific Kinds of DFS Traversals

- Depending on when we process the current node, there are three general kinds of DFS traversals:
  - Pre-order: process current first
  - In-order: process current between left and right
  - Post-order: process current after left and right

# Pre-Order Traversal

```
void traverse(Node* current) {
  if (current != NULL) {
    process(current);
    traverse(current->getLeft());
    traverse(current->getRight());
  }
}
```

# In-Order Traversal

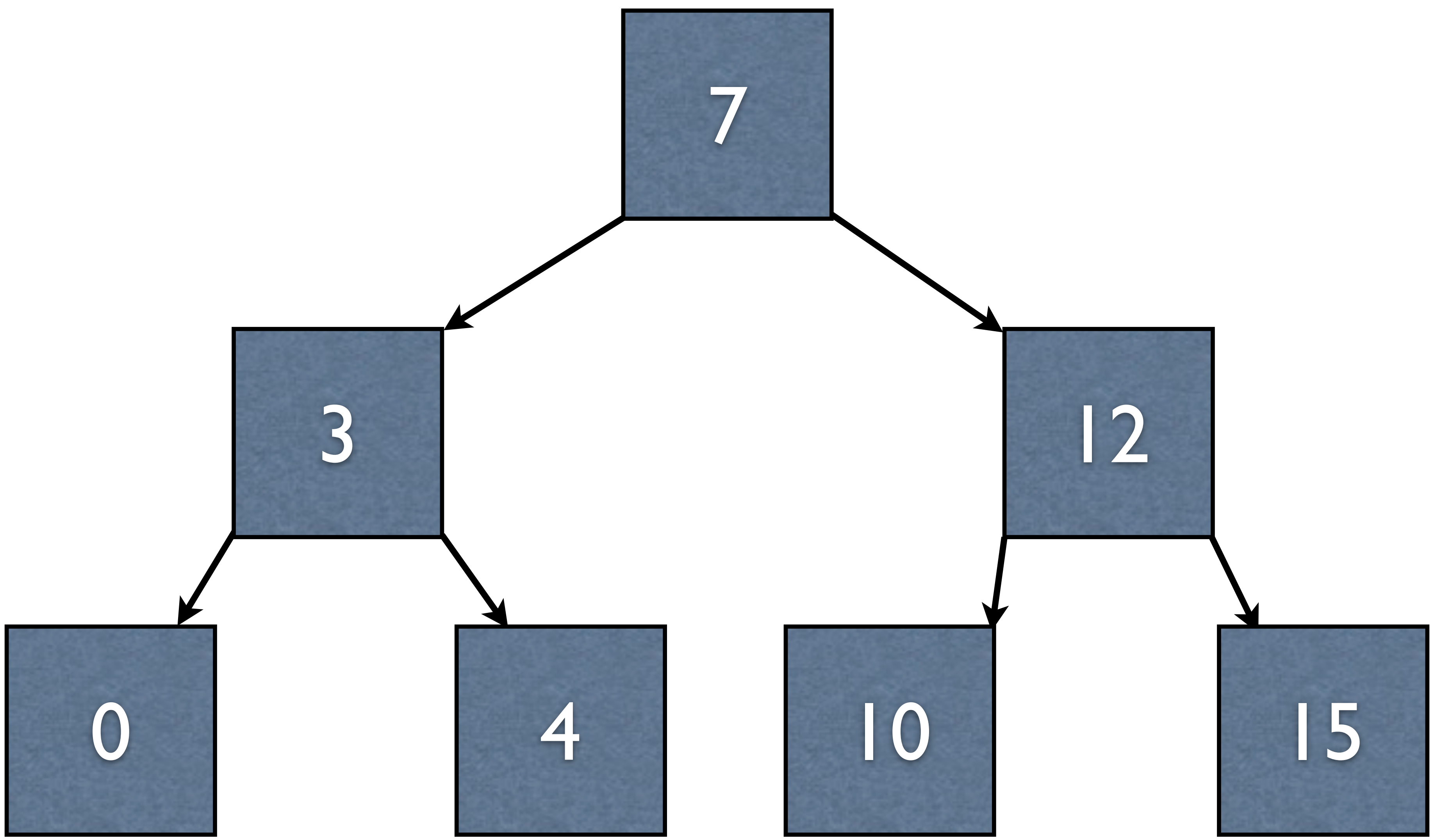
```
void traverse(Node* current) {
  if (current != NULL) {
    traverse(current->getLeft());
    process(current);
    traverse(current->getRight());
  }
}
```

# Post-Order Traversal

```
void traverse(Node* current) {
  if (current != NULL) {
    traverse(current->getLeft());
    traverse(current->getRight());
    process(current);
  }
}
```

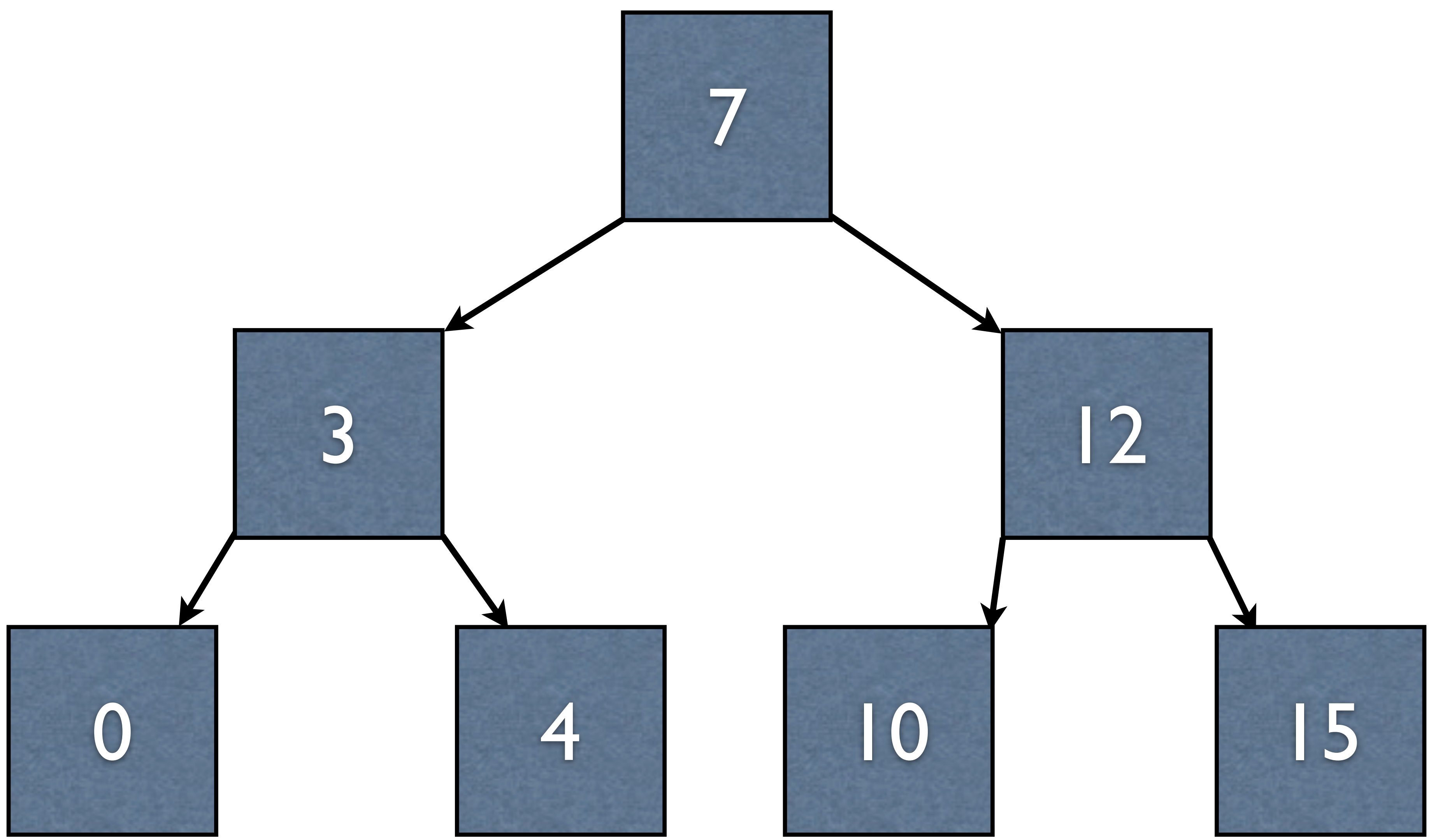
# Using Traversals

- Say we want to print out the contents of a binary search tree in sorted order
- What kind of traversal should we use?



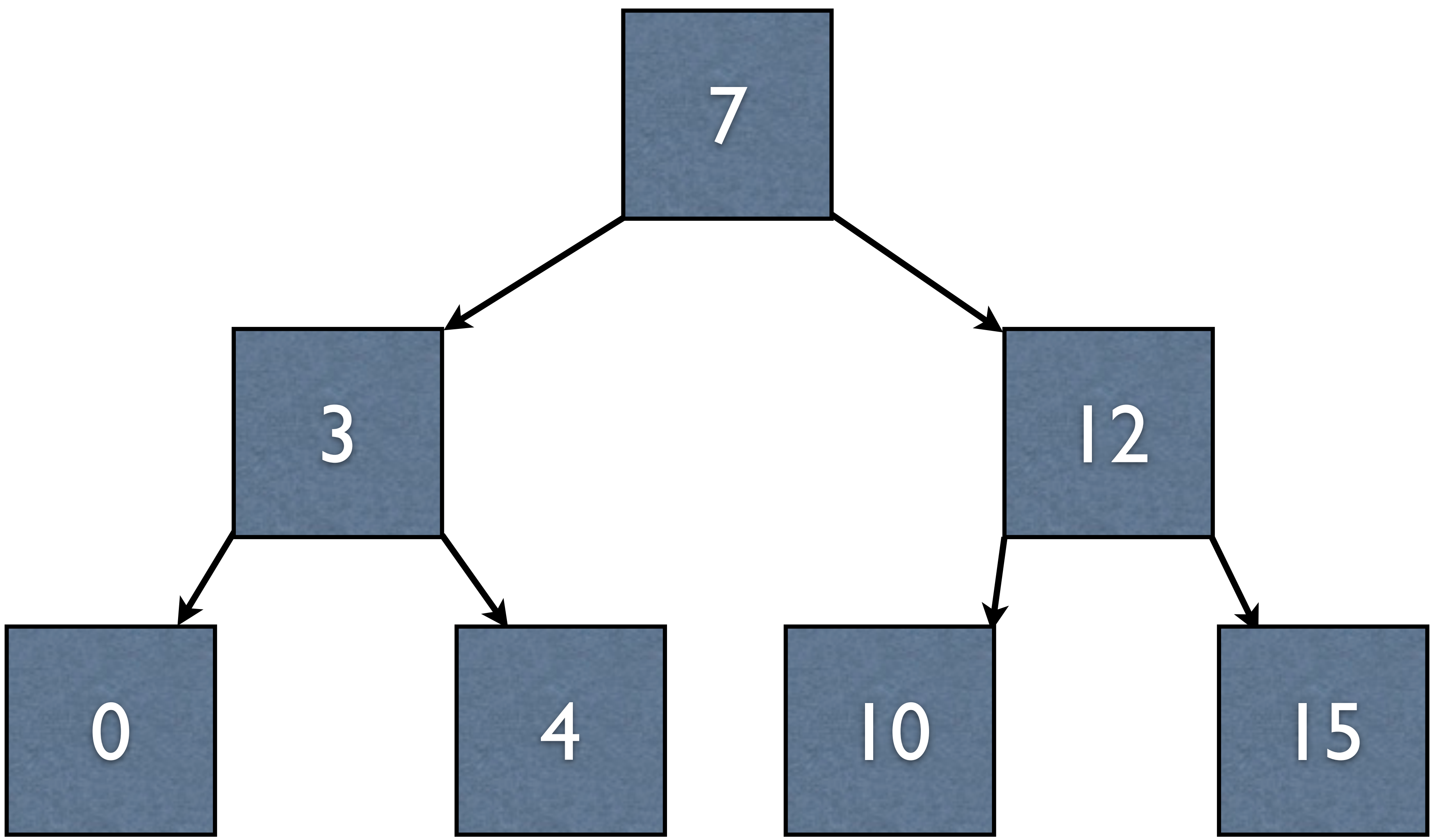
# Using Traversals

- Say we want to print out the contents of a binary search tree in sorted order
- What kind of traversal should we use? - in-order



# Using Traversals

- Say we want to delete a binary search tree
- Which traversal is best?



# Using Traversals

- Say we want to delete a binary search tree
- Which traversal is best? - post-order