Balanced Binary Search Trees

Robert Horvick
SOFTWARE ENGINEER

@bubbafat www.roberthorvick.com

Overview

Binary Search Tree

- Unbalanced and Balanced
- Height and Balance Factor

AVL Tree

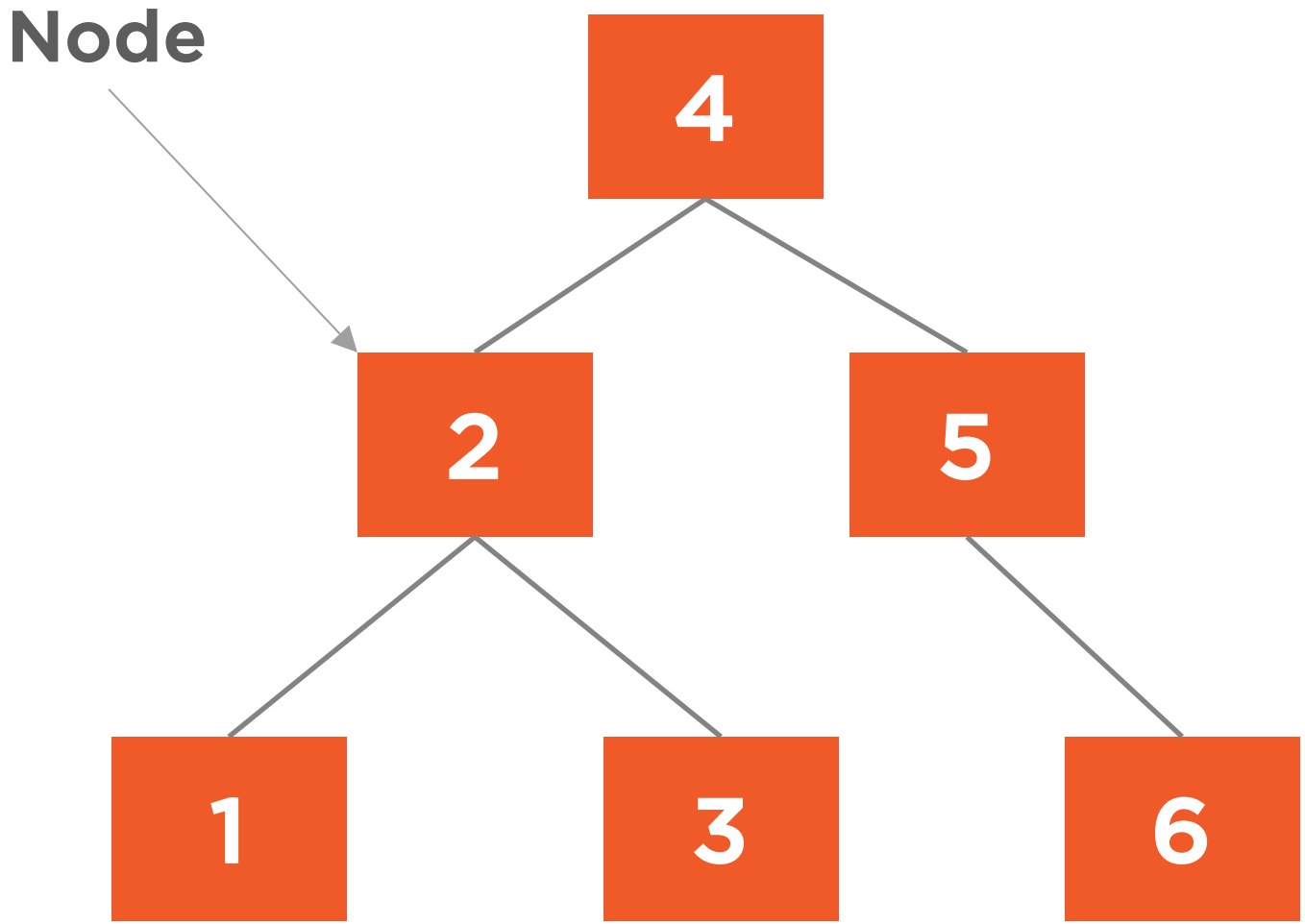
- Self-balancing Binary Search Tree

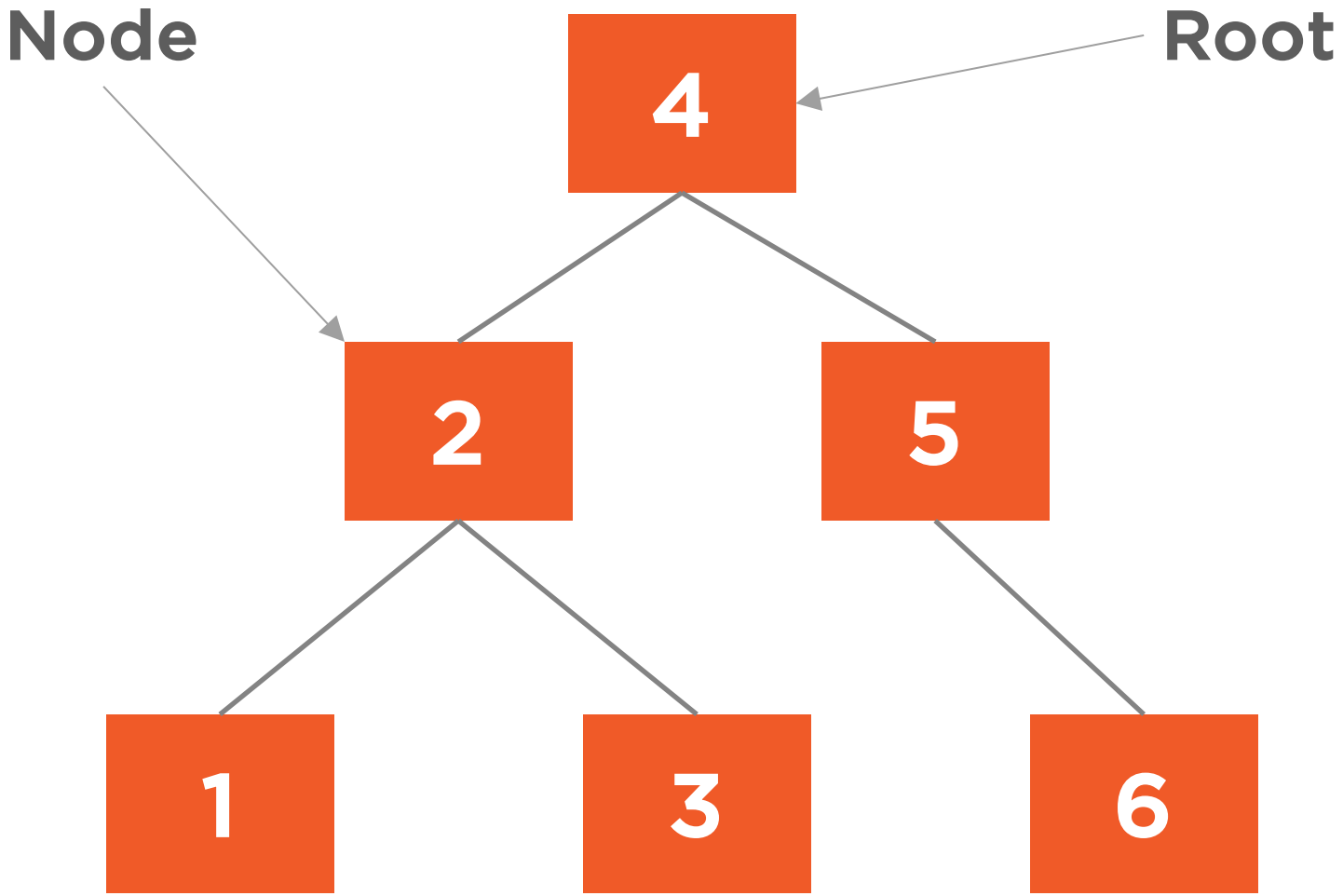
Balancing Algorithms

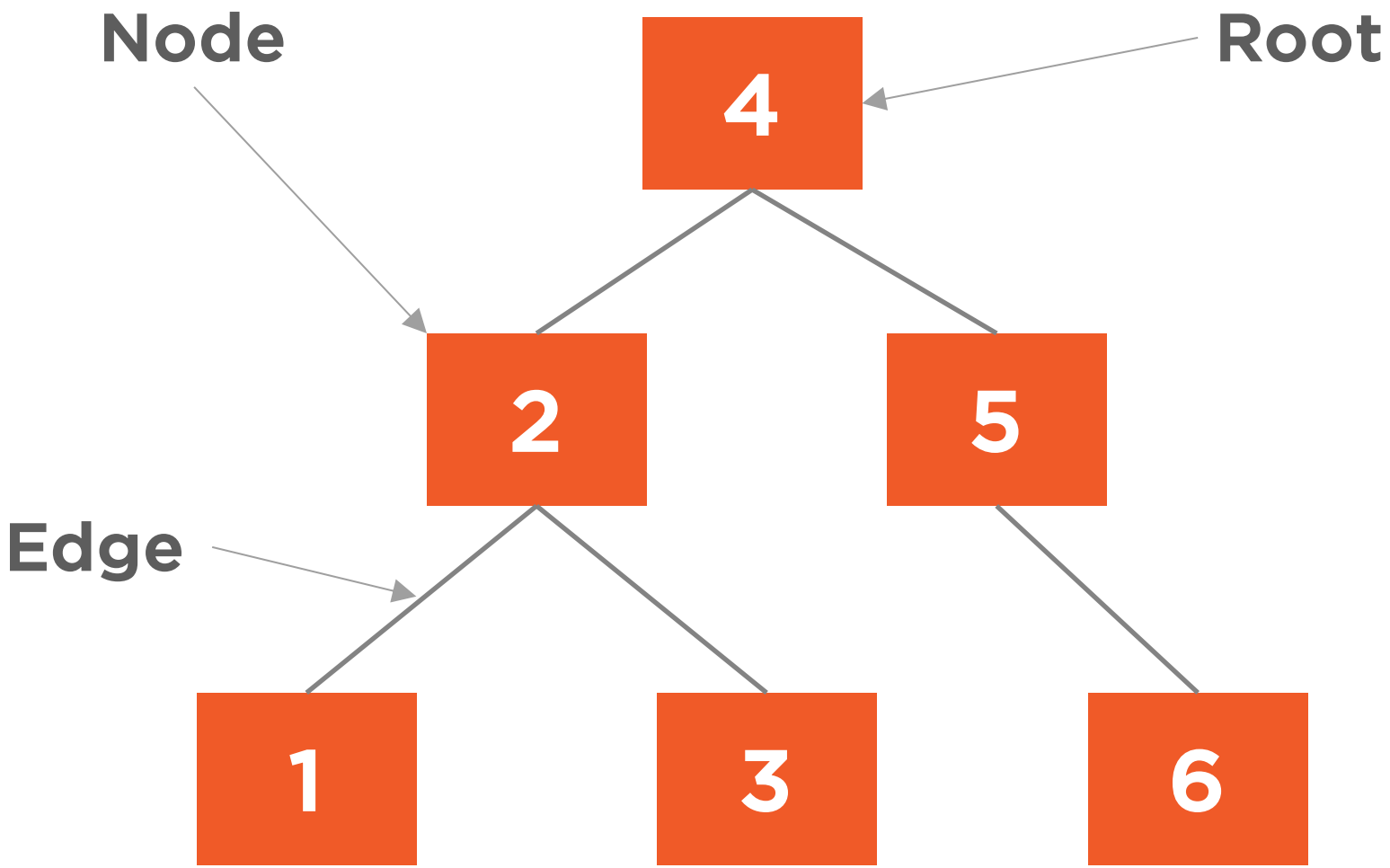
- Right Rotation
- Left Rotation
- Right-Left Rotation
- Right-Left Rotation

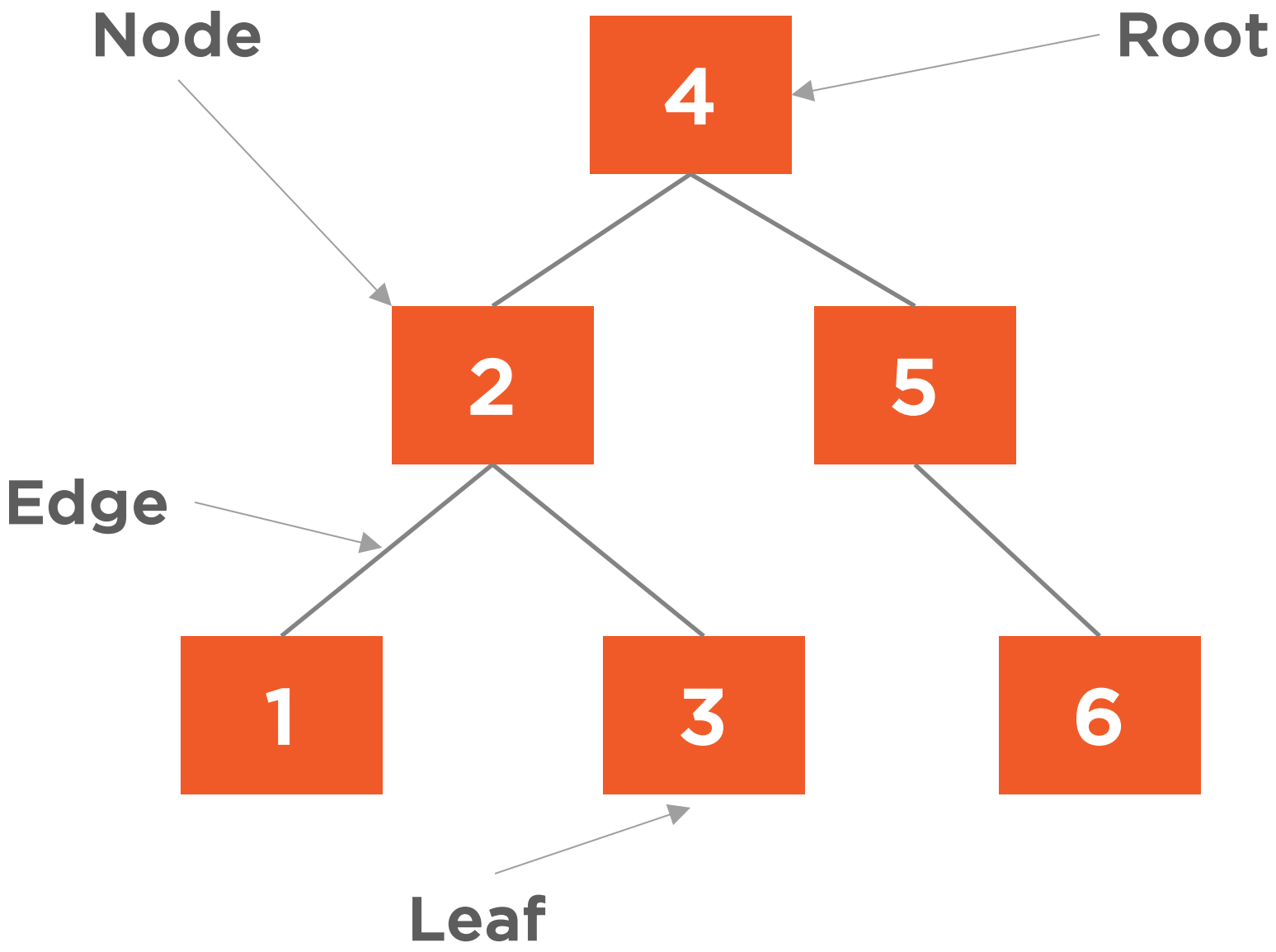
Binary Search Tree

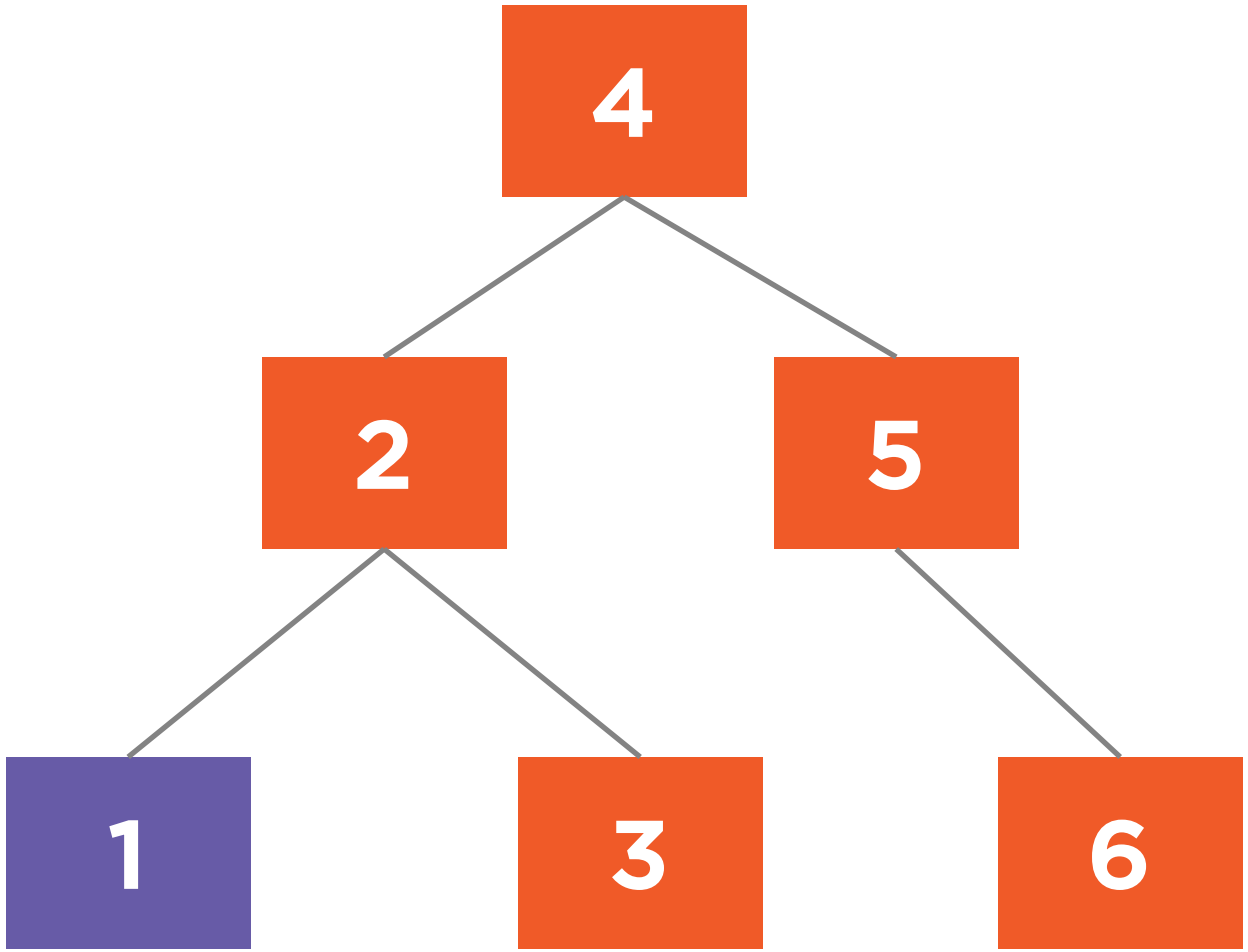
A sorted data structure where each node can have 0-2 children and each node, except the root, has exactly one parent.

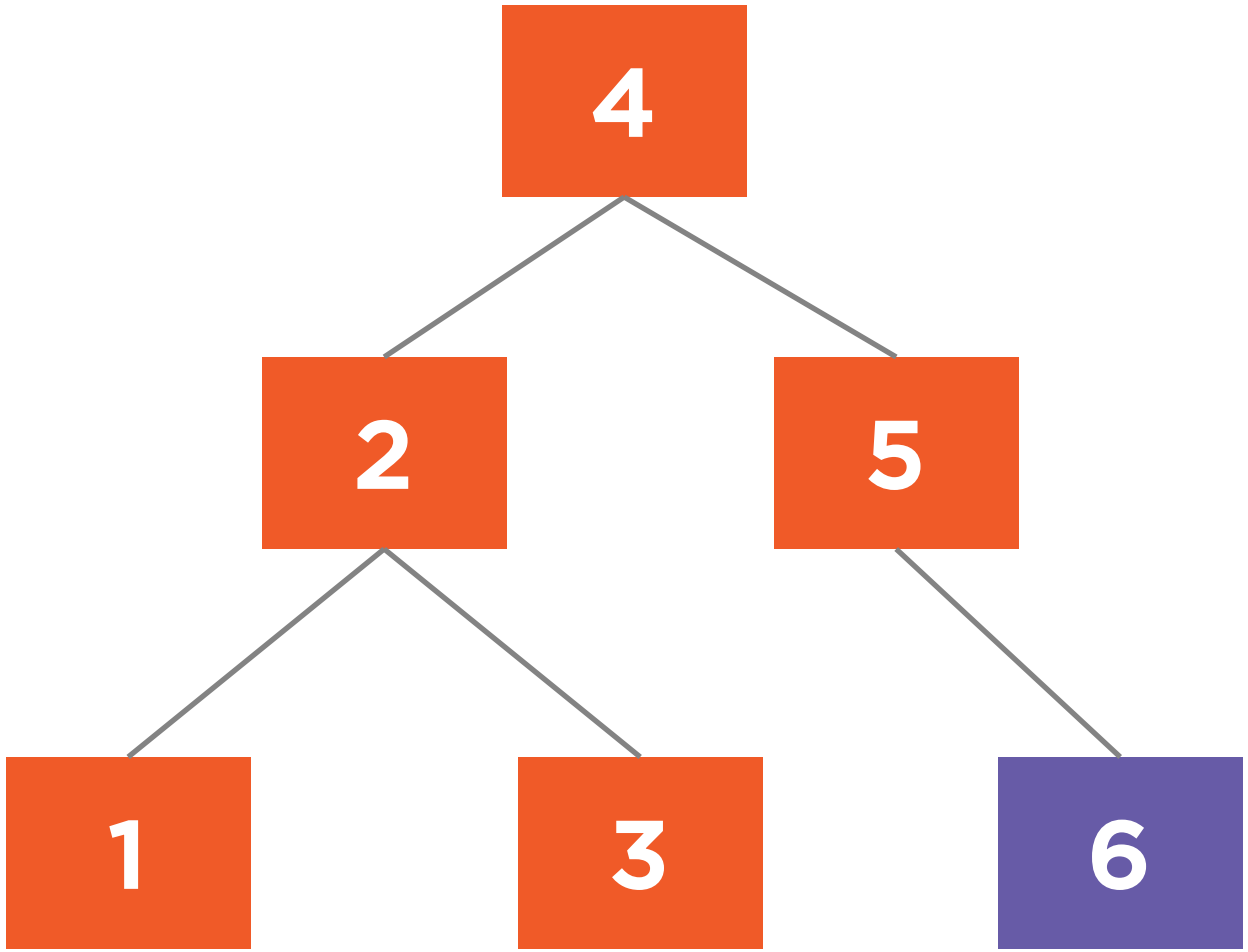

Node
4
2
5
1
3
6


Node
Root
4
2
5
1
3
6


Node
Root
4
2
5
Edge
1
3
6


Node
Root
4
2
5
Edge
1
3
6
Leaf


4
2
5
1
3
6


4
2
5
1
3
6

Binary tree insert, remove, and search operations are O(log n) average case

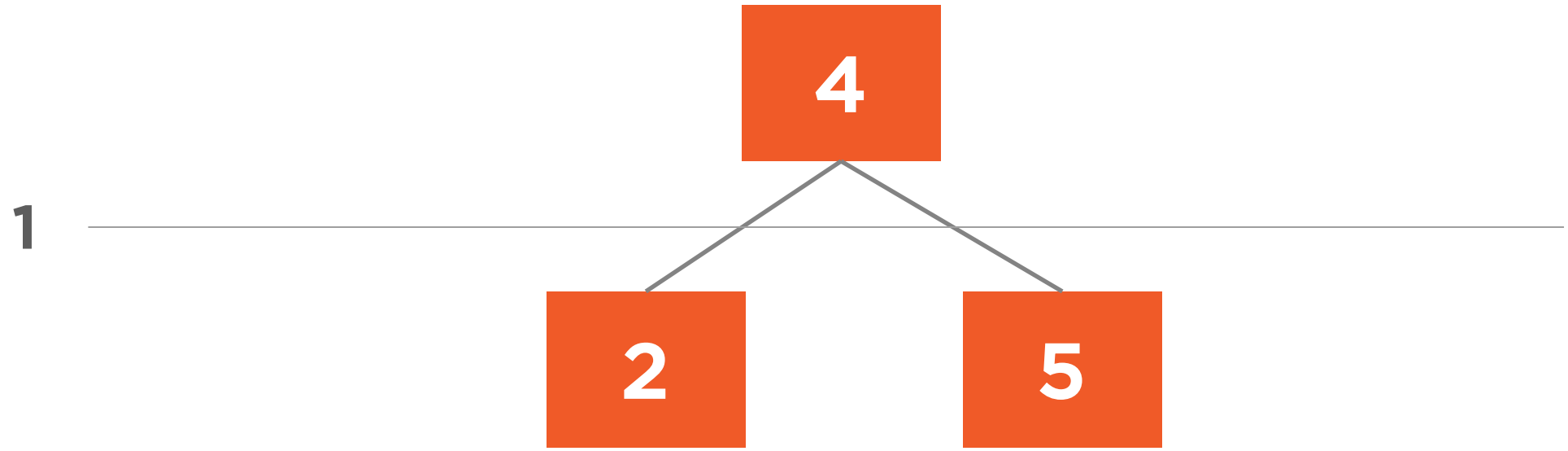
Unbalanced Tree

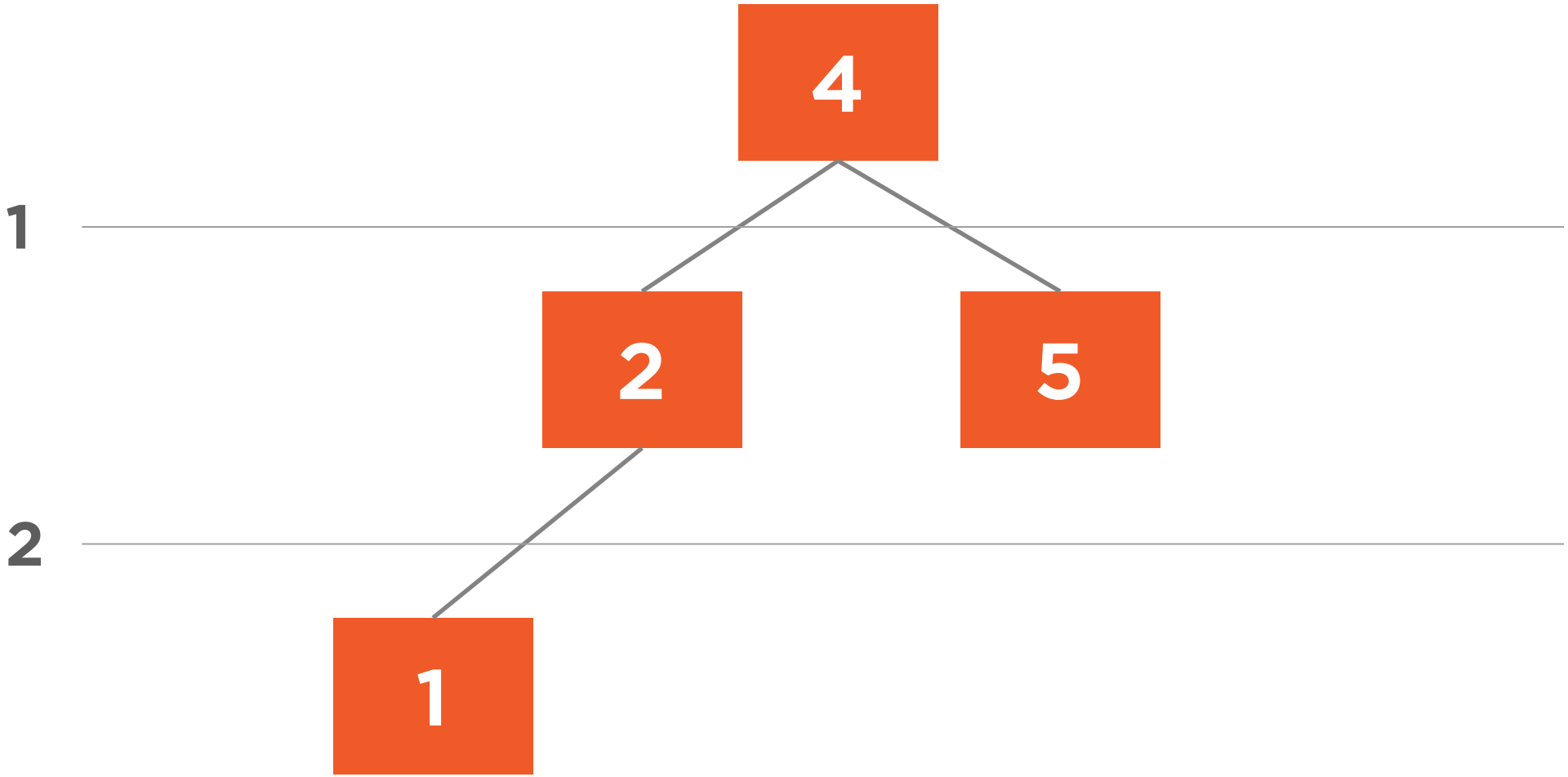
A tree whose left and right children have uneven heights.

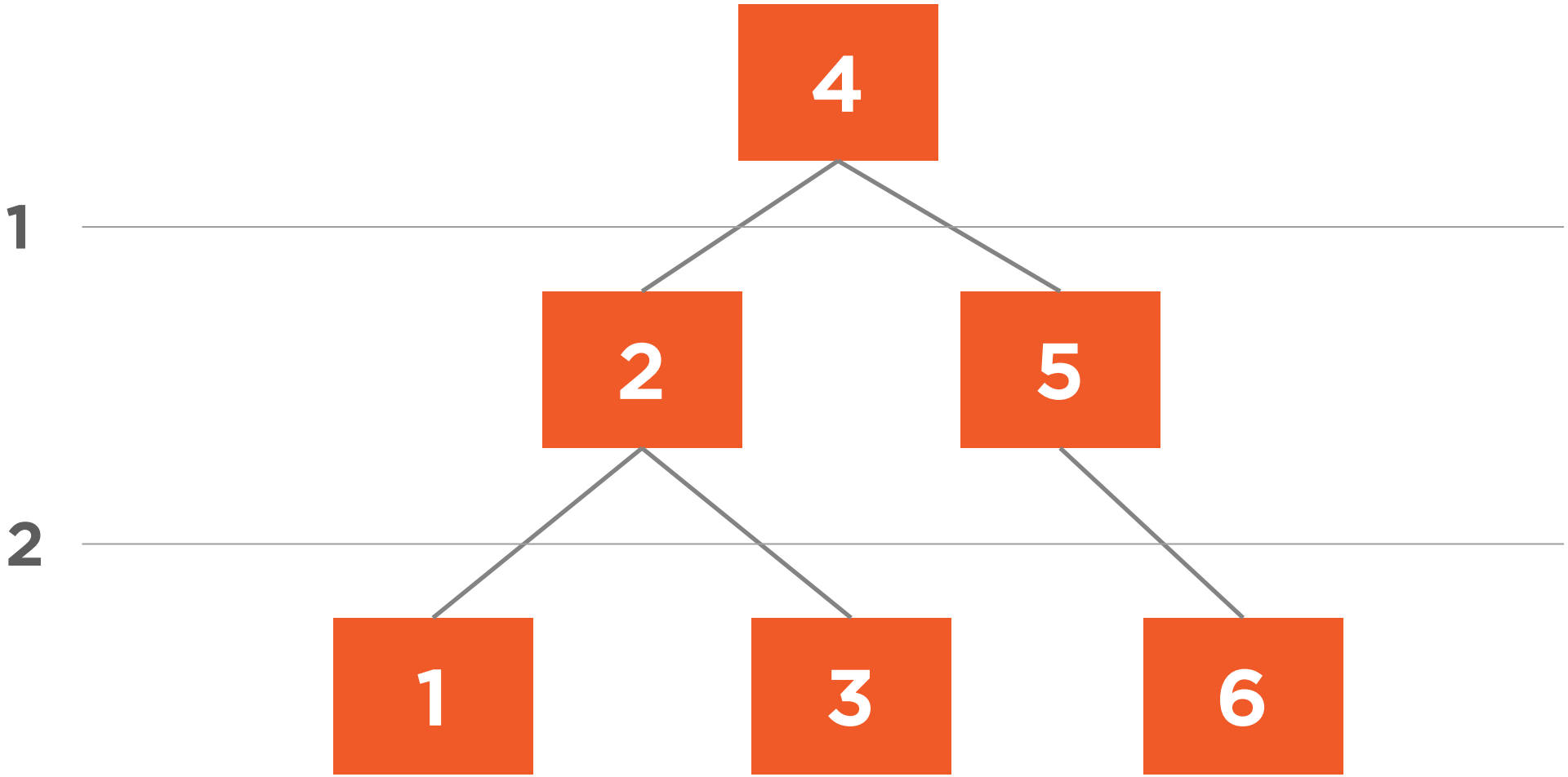
Height

The maximum number of edges between the root and leaf nodes.

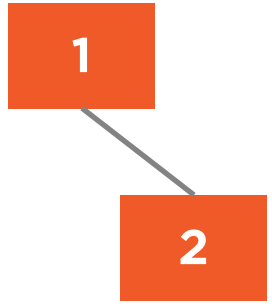
4

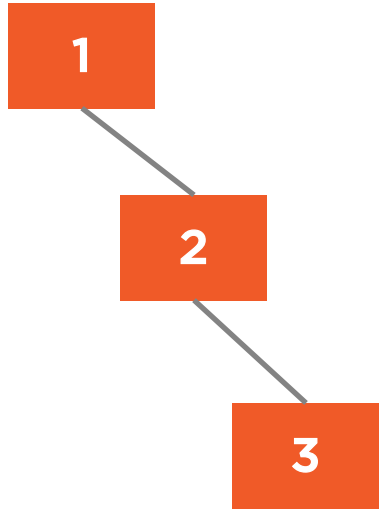

4
1
2
5

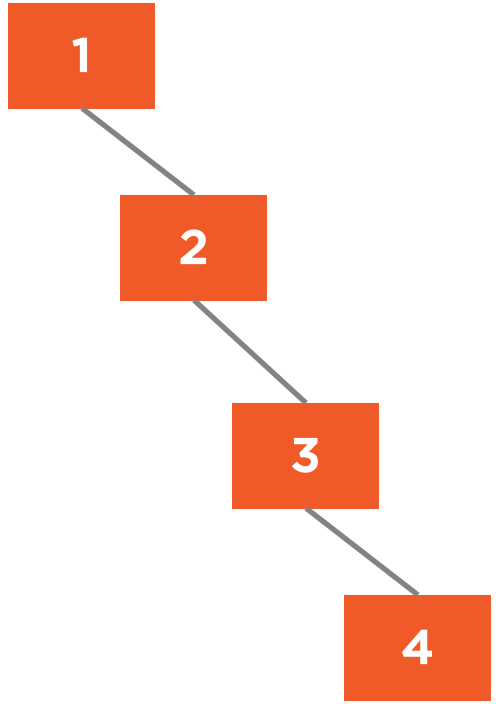

4
1
2
5
2
1

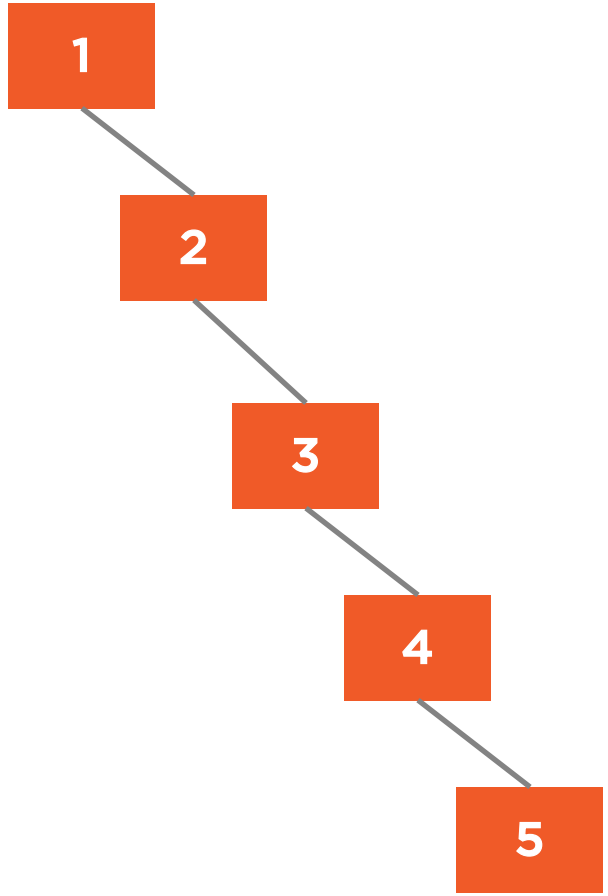

4
1
2
5
2
1
3
6

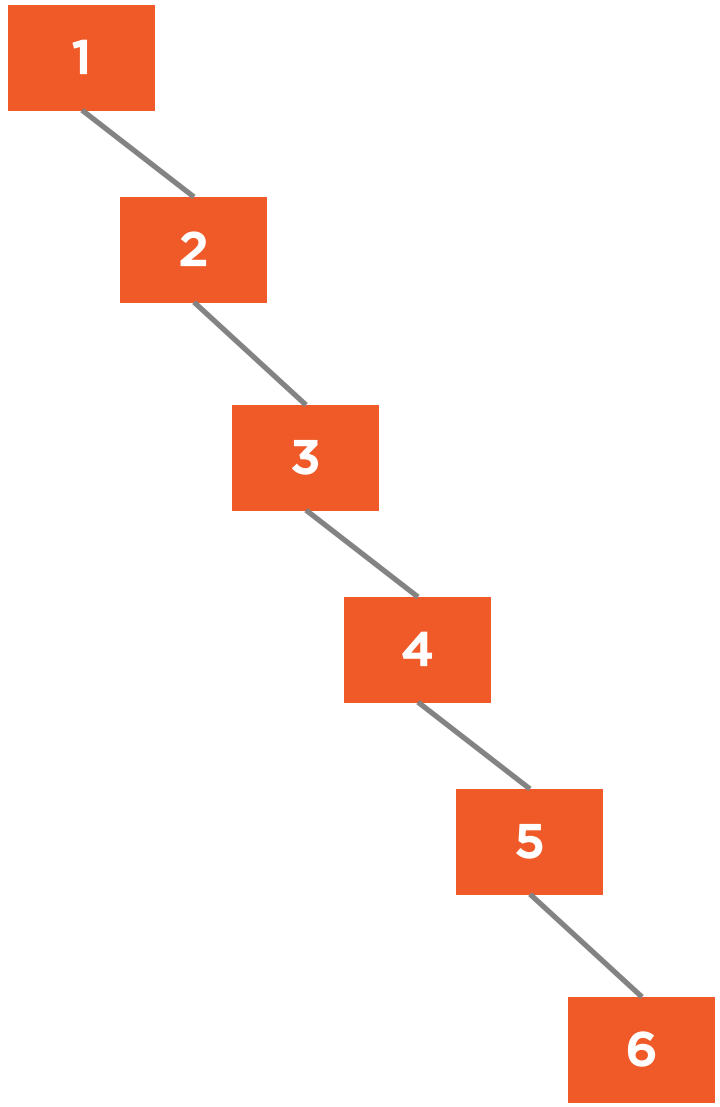
1


1
2

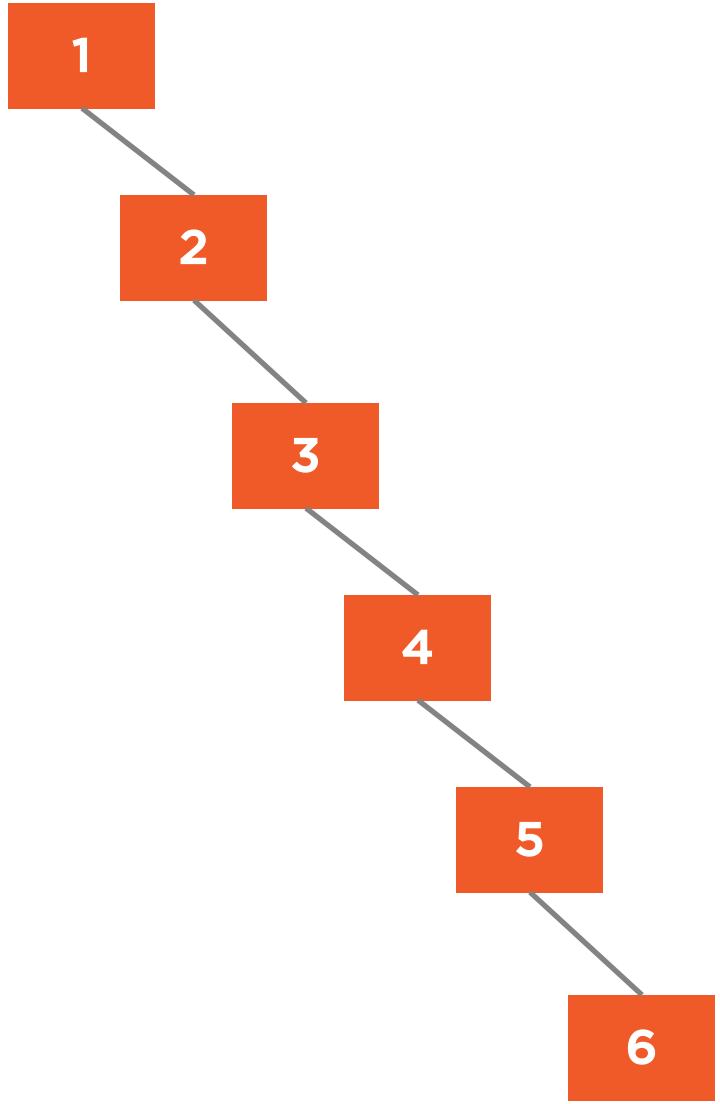

1
2
3

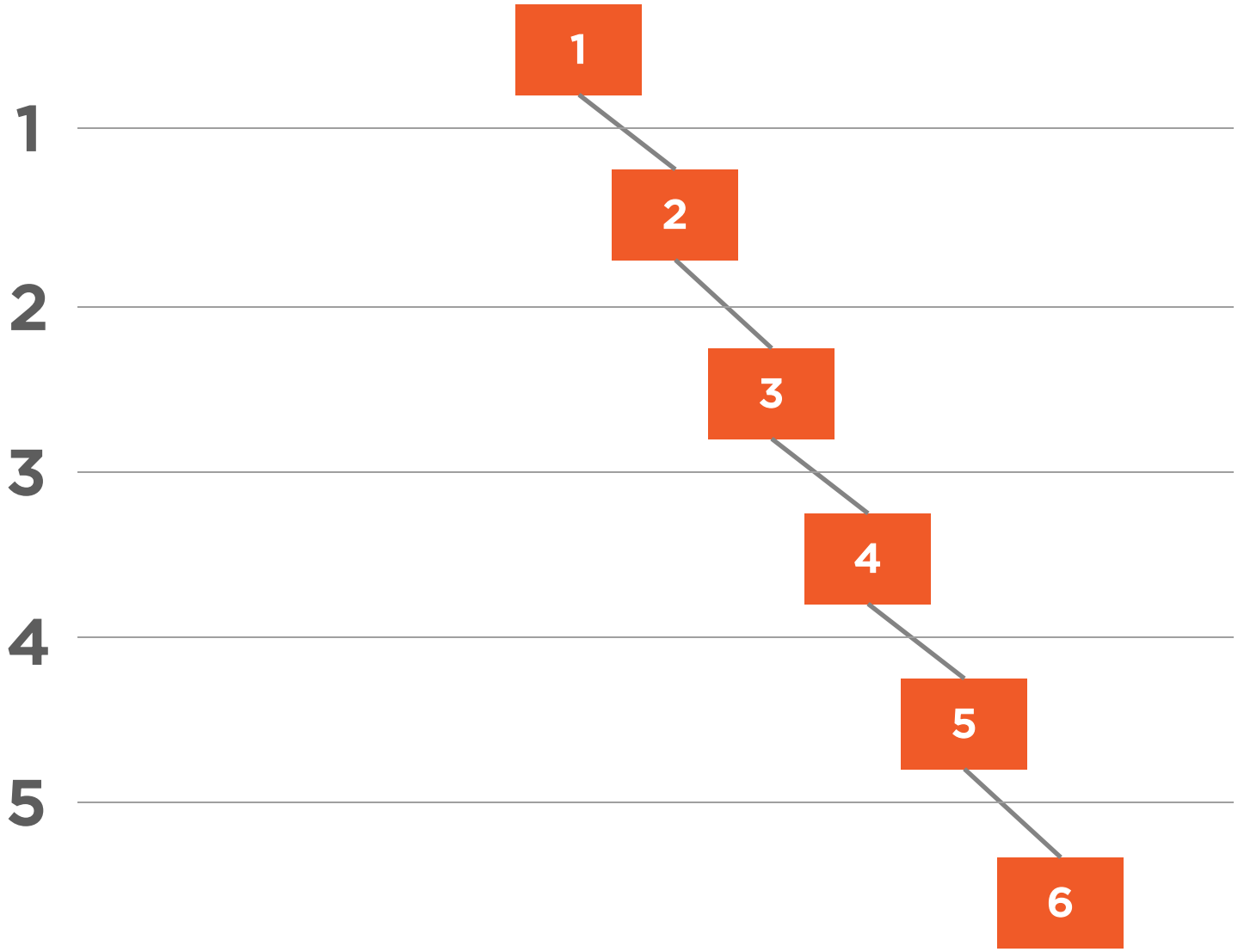

1
2
3
4

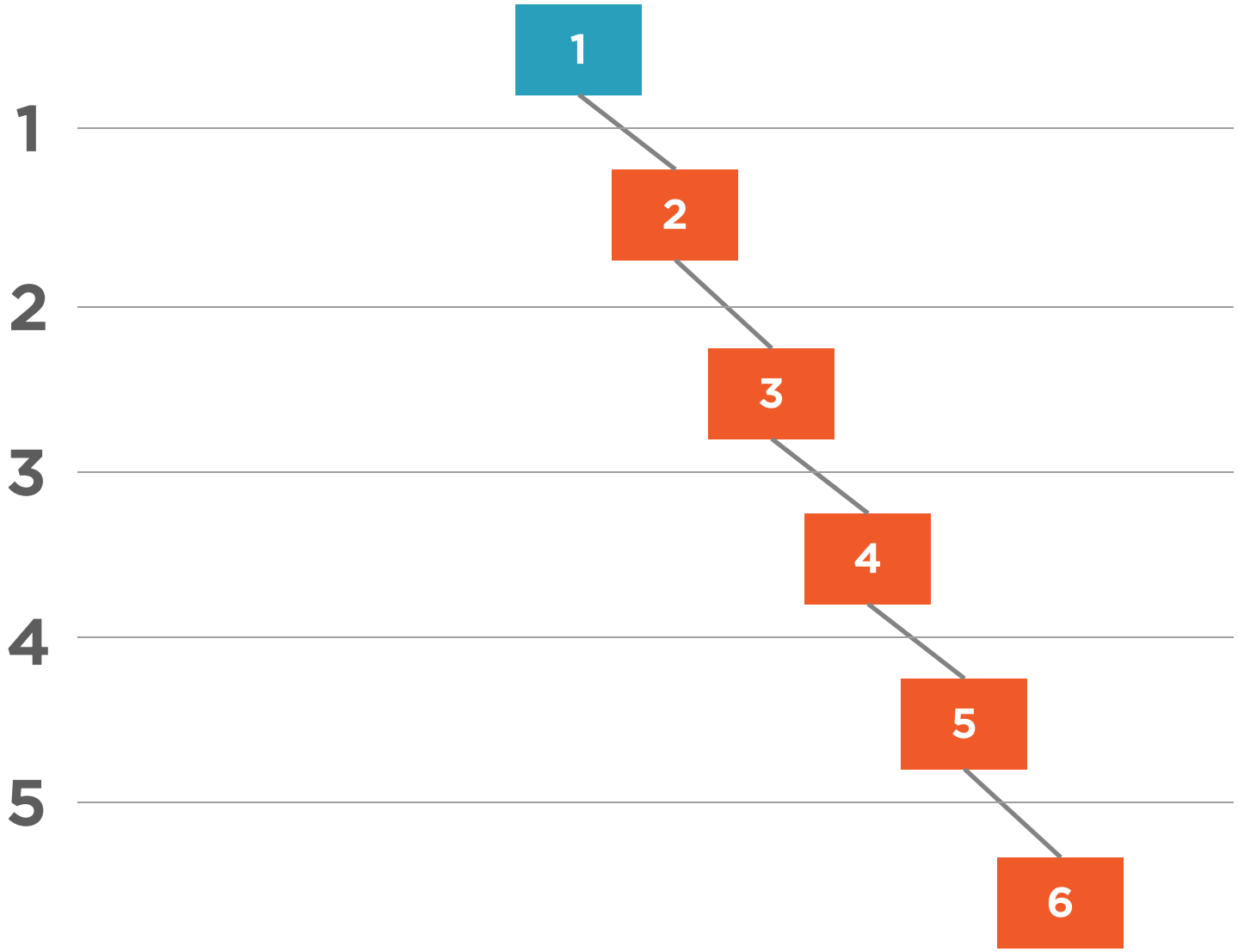

1
2
3
4
5

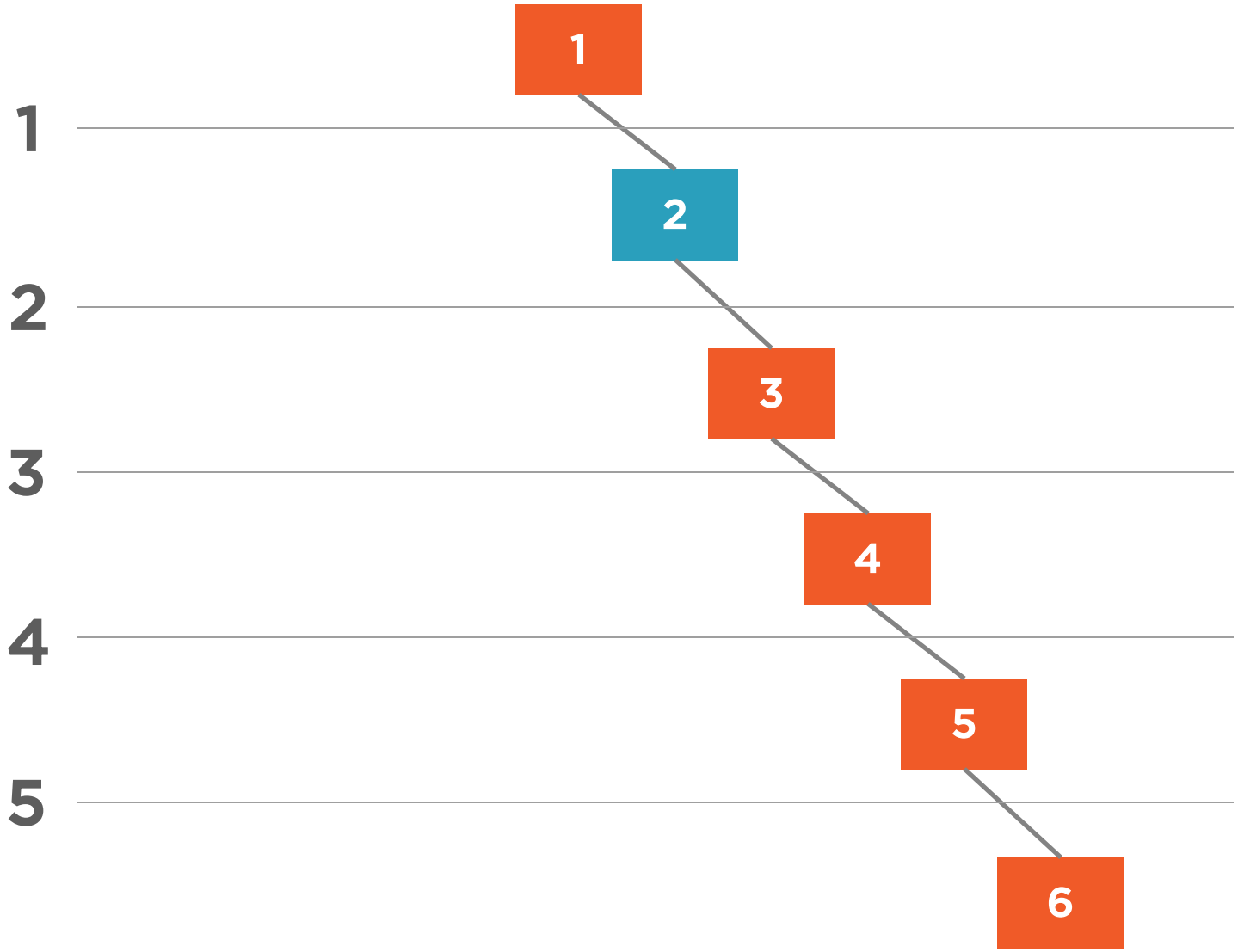

1
2
3
4
5
6

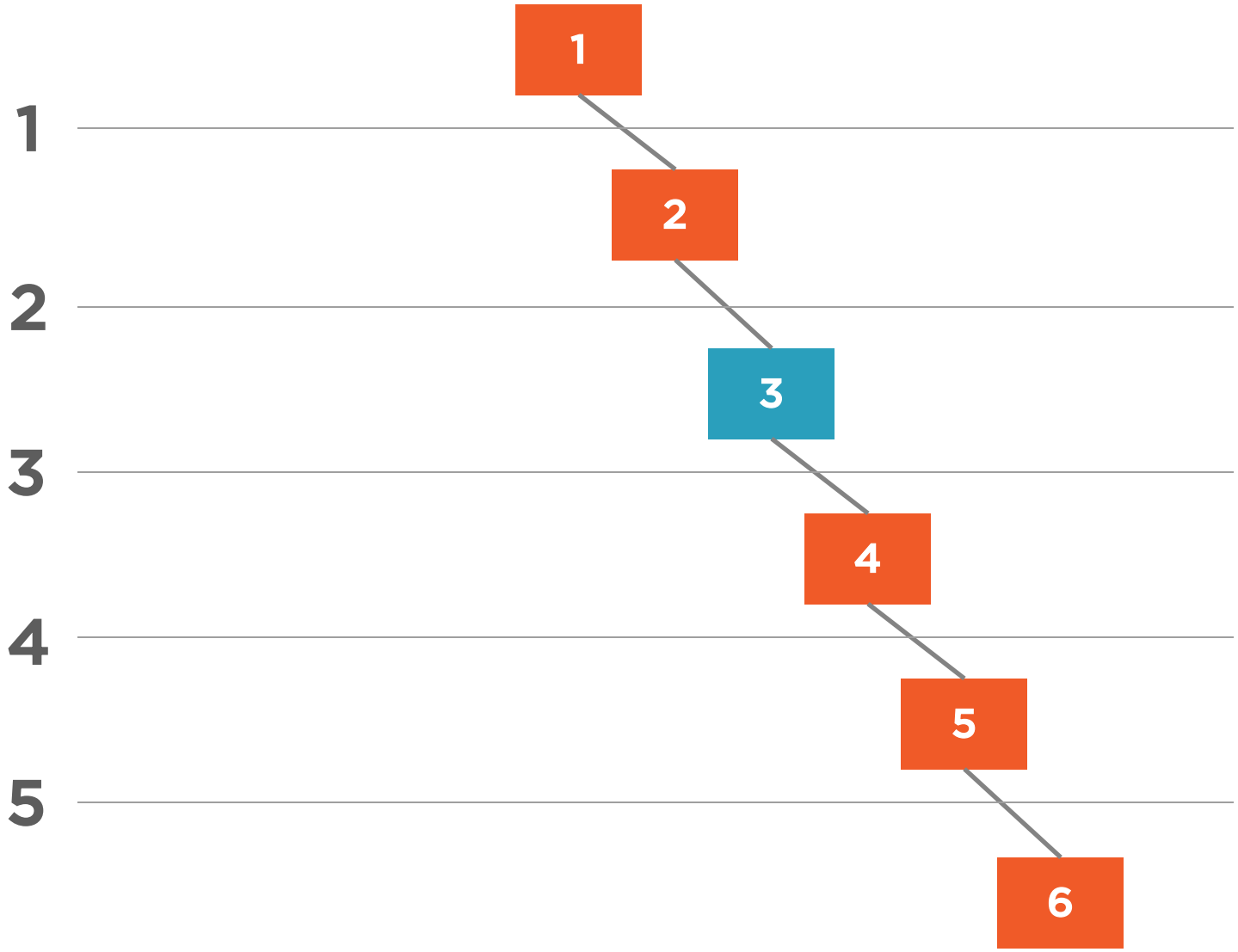

1
5

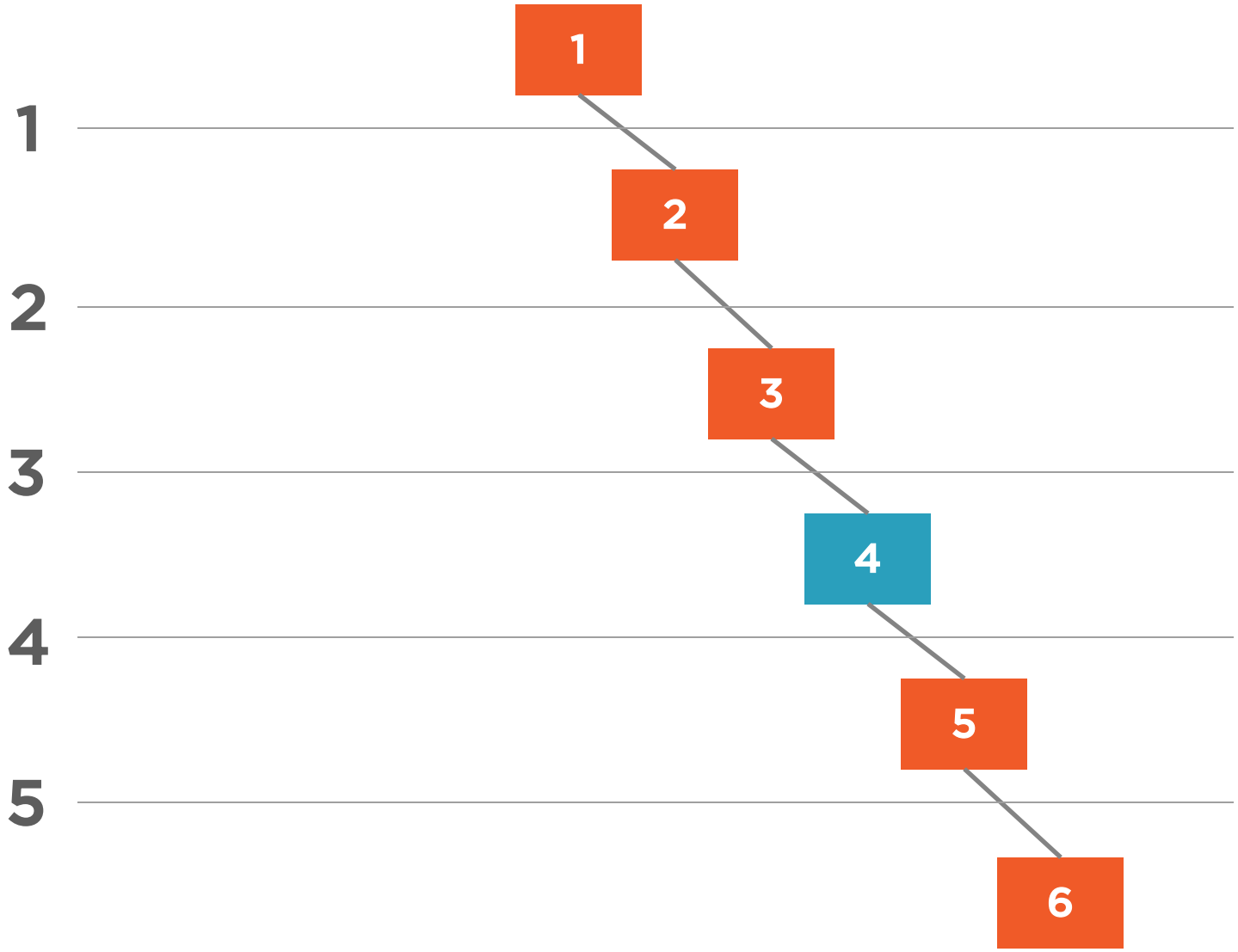

1
2
3
4
5
6

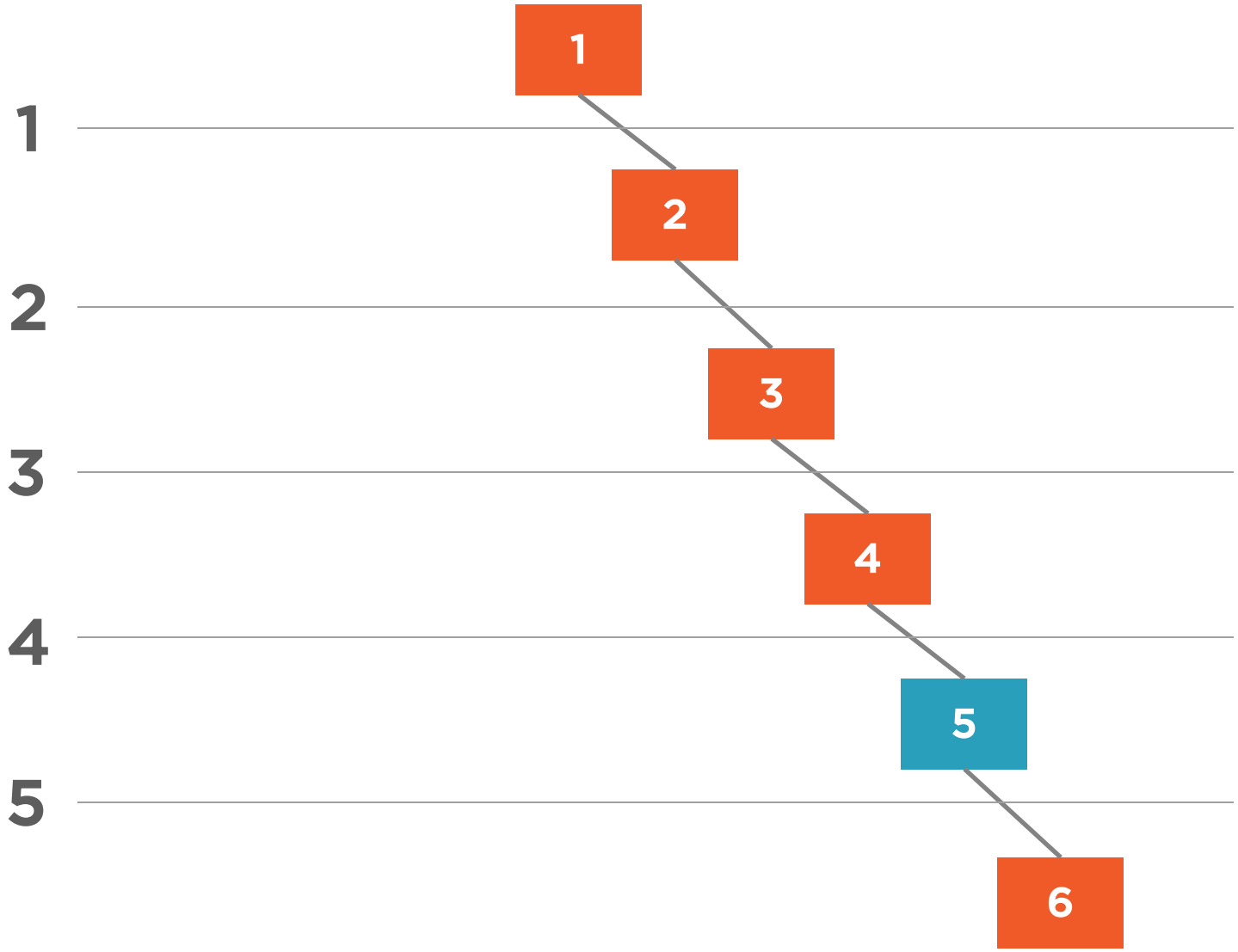

1
2
3
4
5
1
2
3
4
5
6

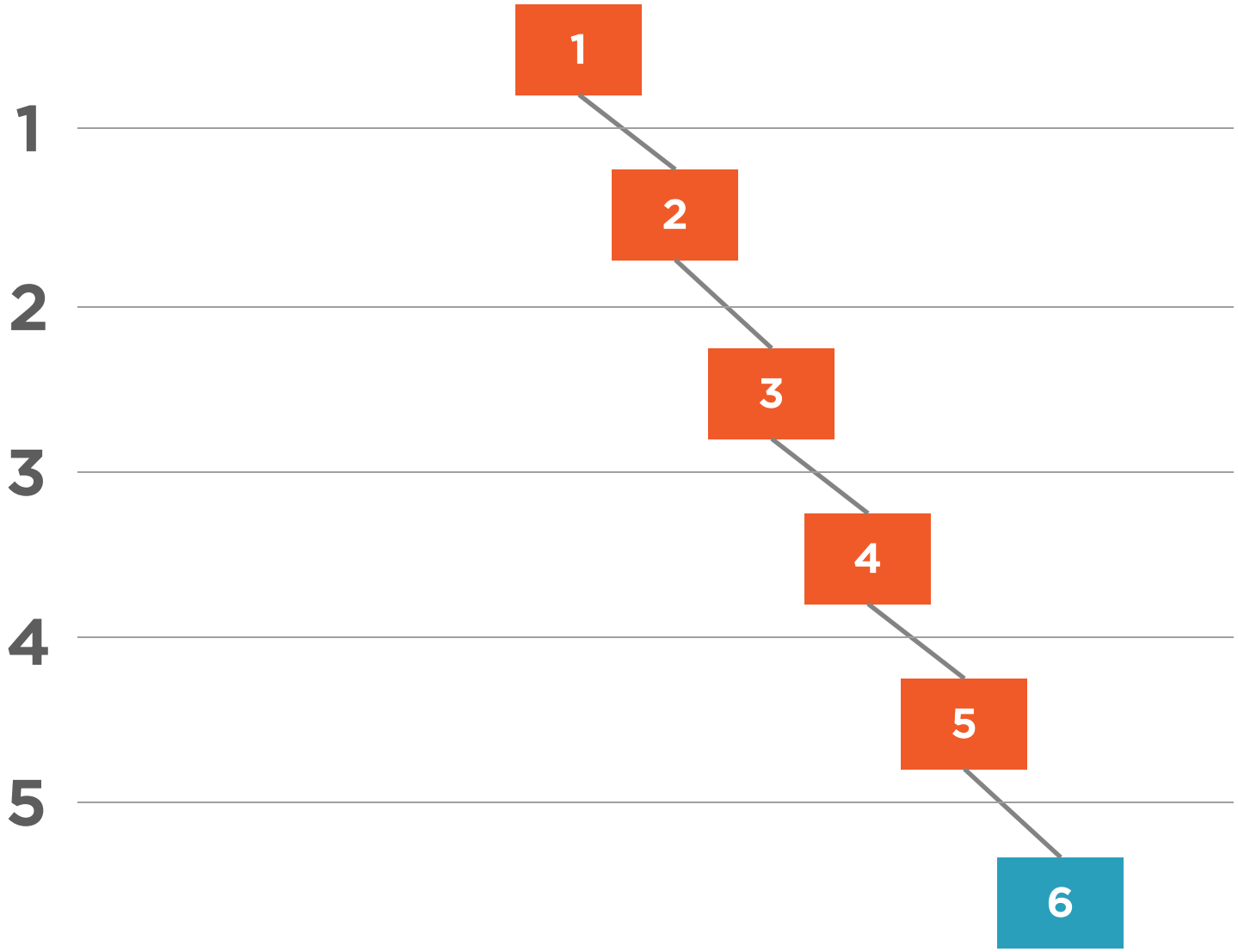

1
2
3
4
5
1
2
3
4
5
6


1
2
3
4
5
1
2
3
4
5
6


1
2
3
4
5
1
2
3
4
5
6


1
2
3
4
5
1
2
3
4
5
6

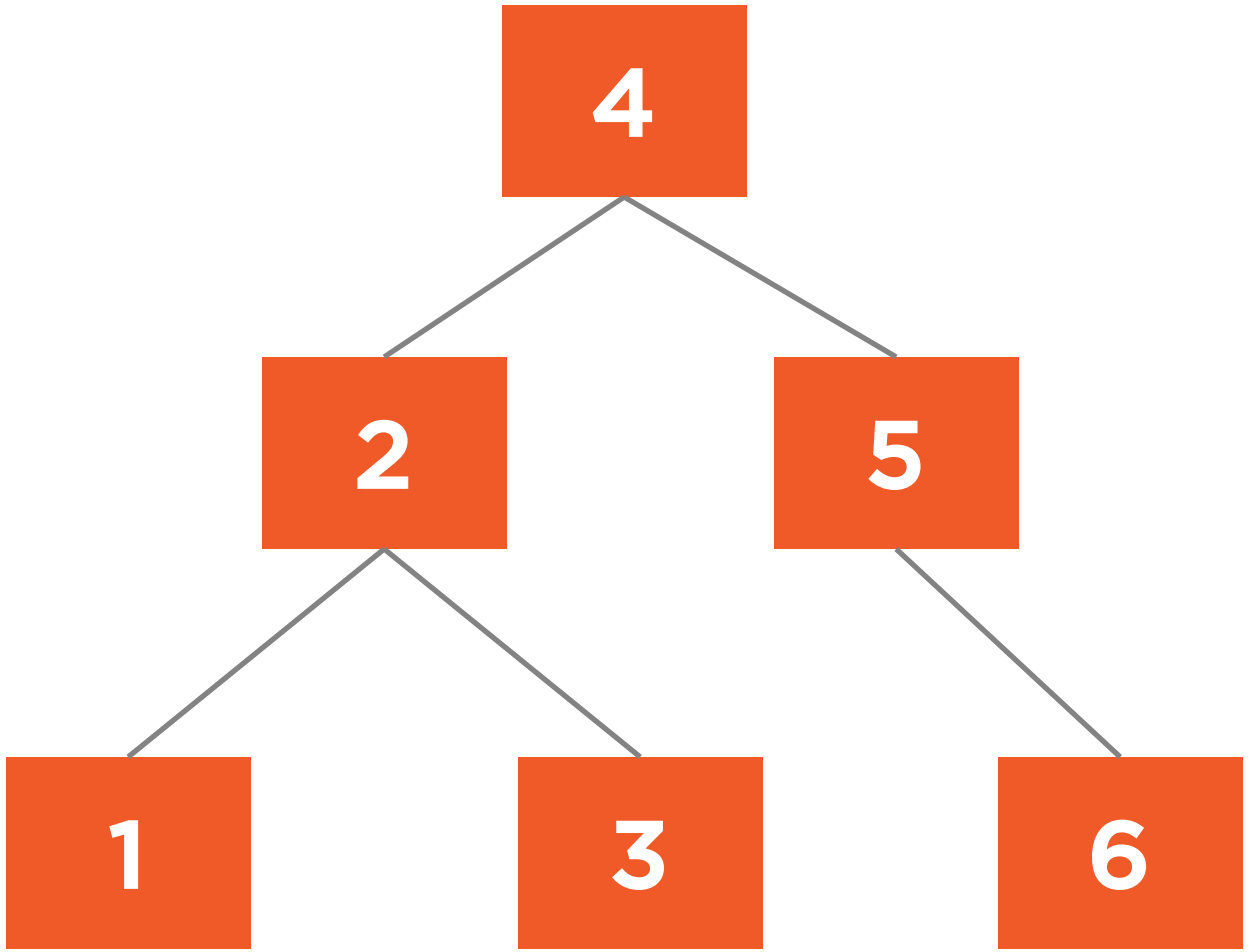

1
2
3
4
5
1
2
3
4
5
6


1
2
3
4
5
1
2
3
4
5
6

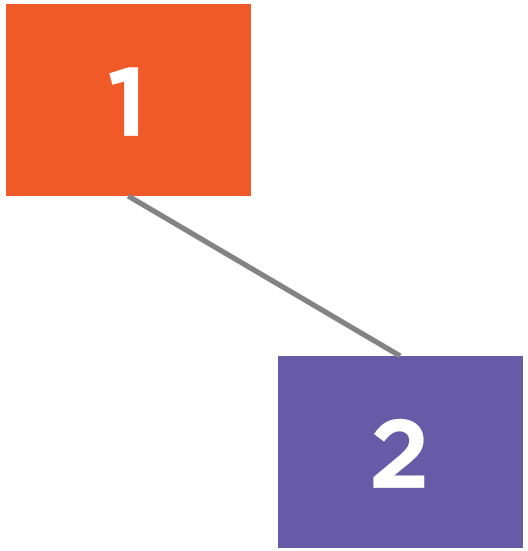
A fully unbalanced binary tree is just a linked list with O(n) algorithmic complexity

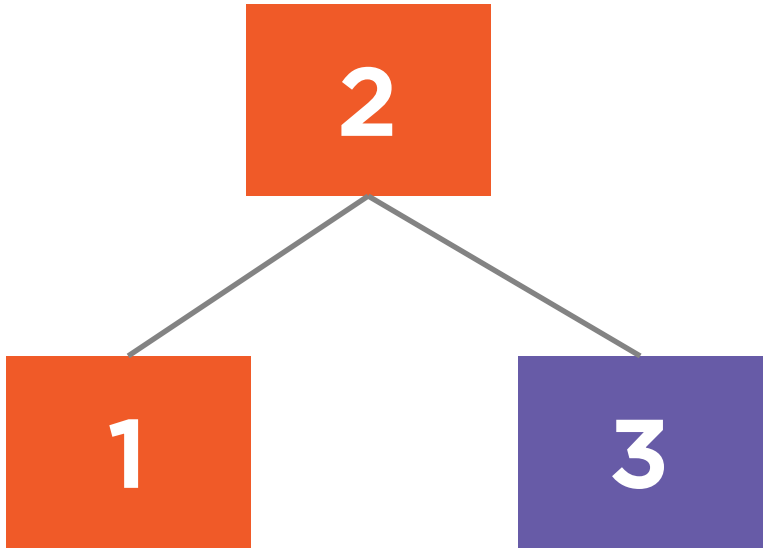
Balanced Tree

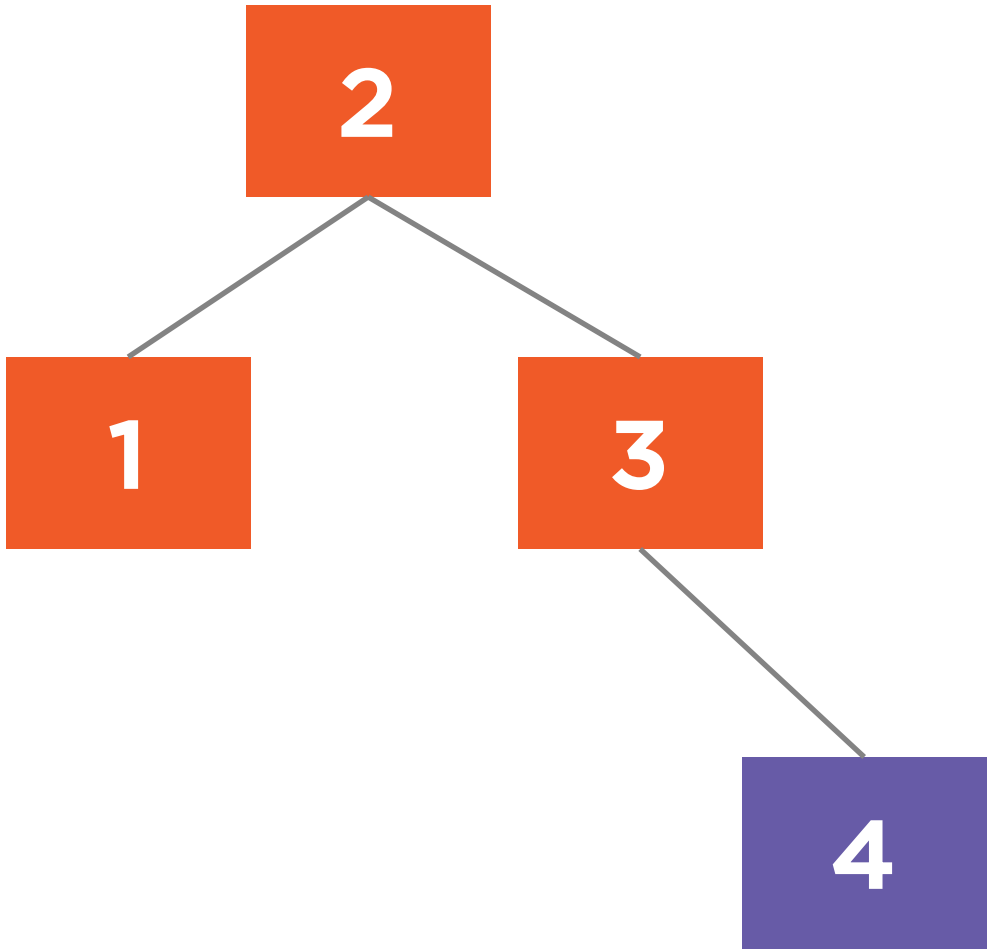
A binary search tree whose maximum height is minimized

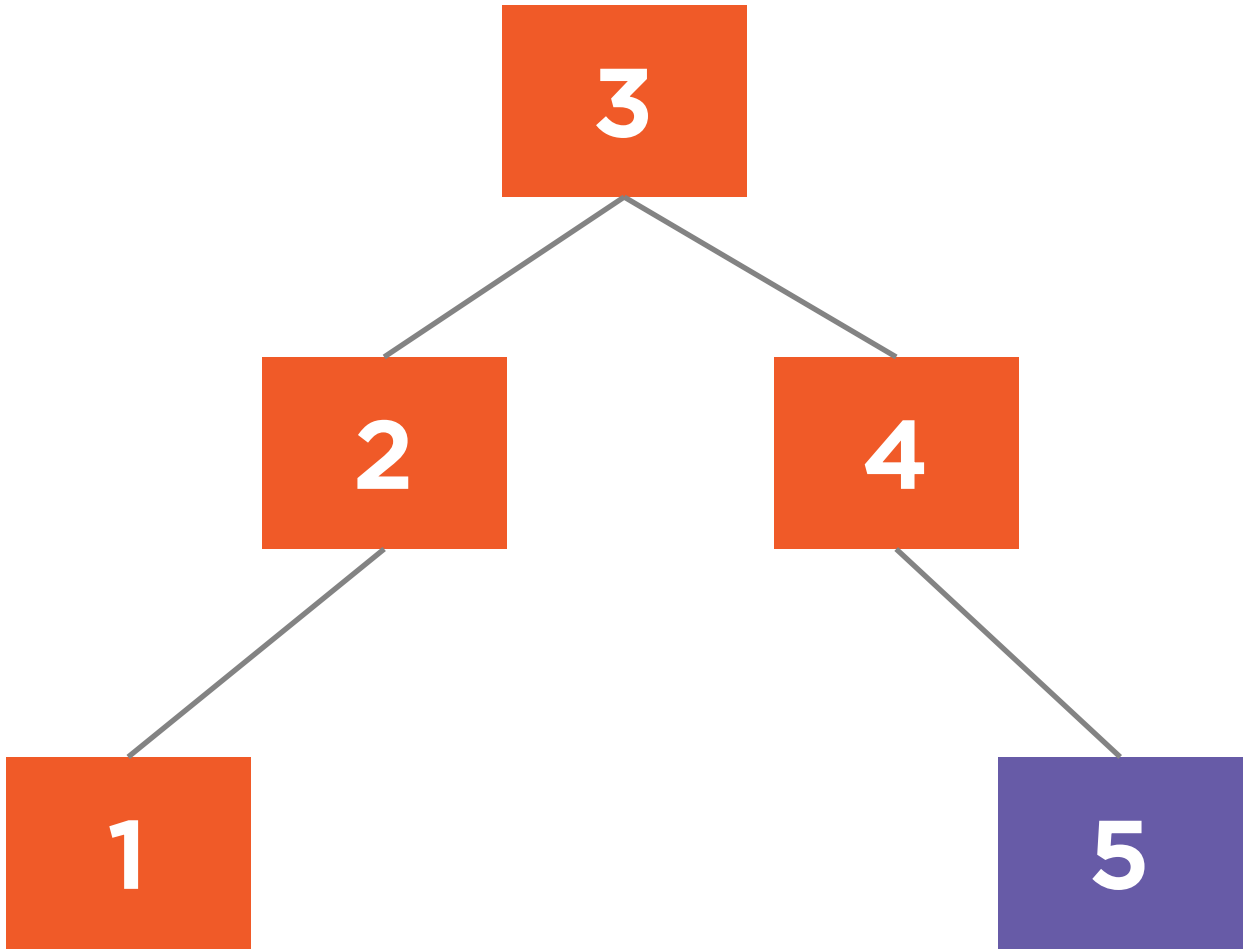

4
2
5
1
3
6

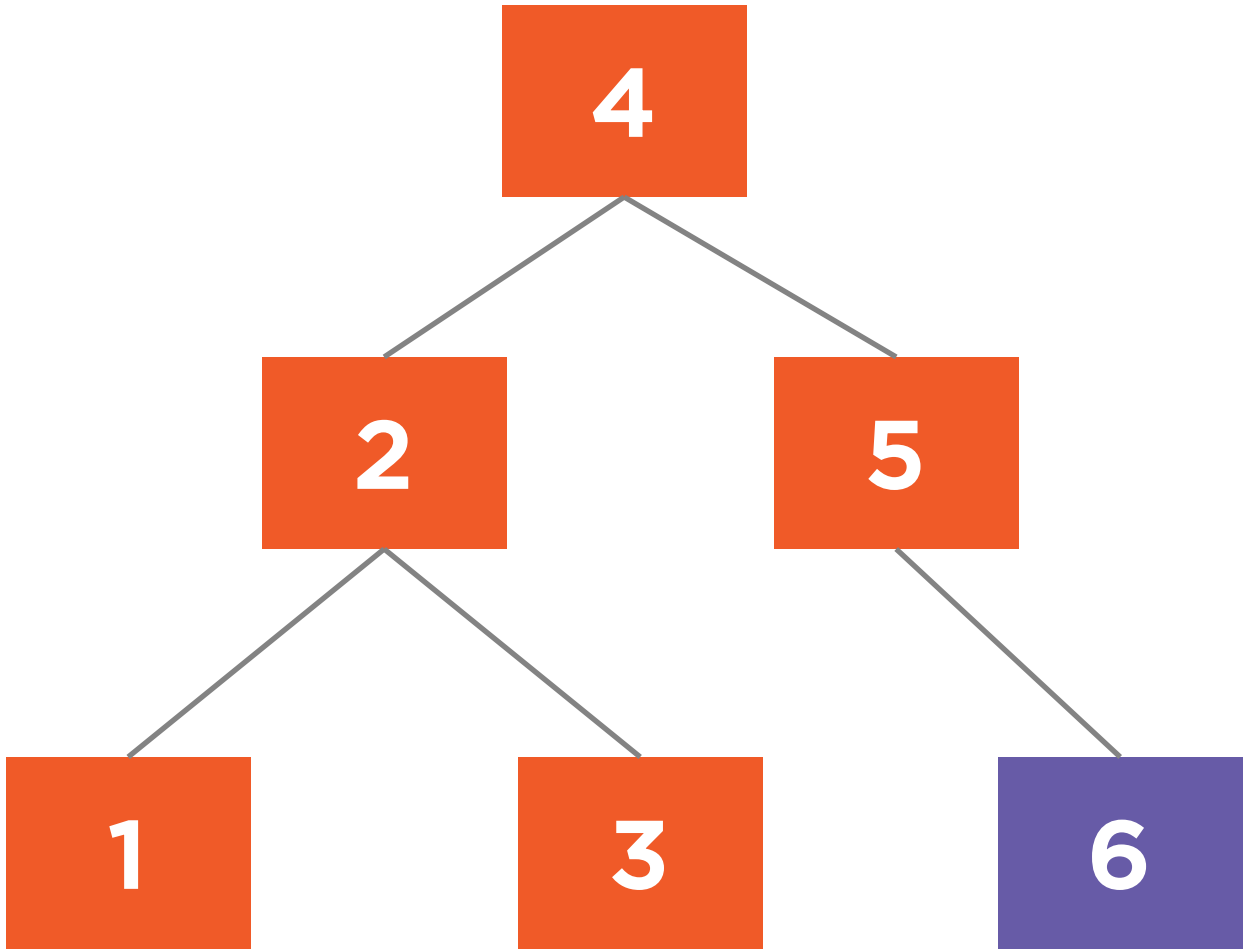
1

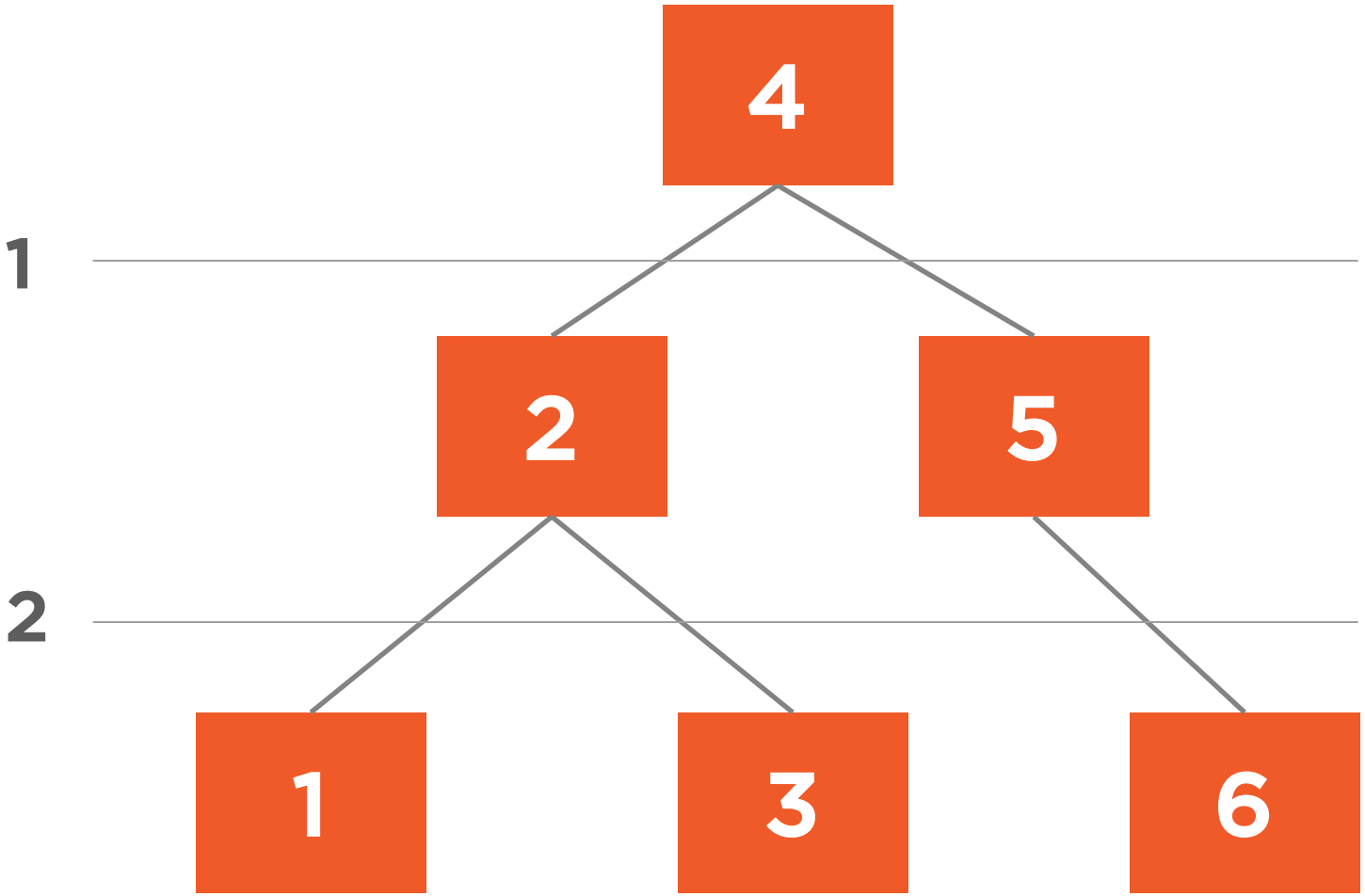

1
2

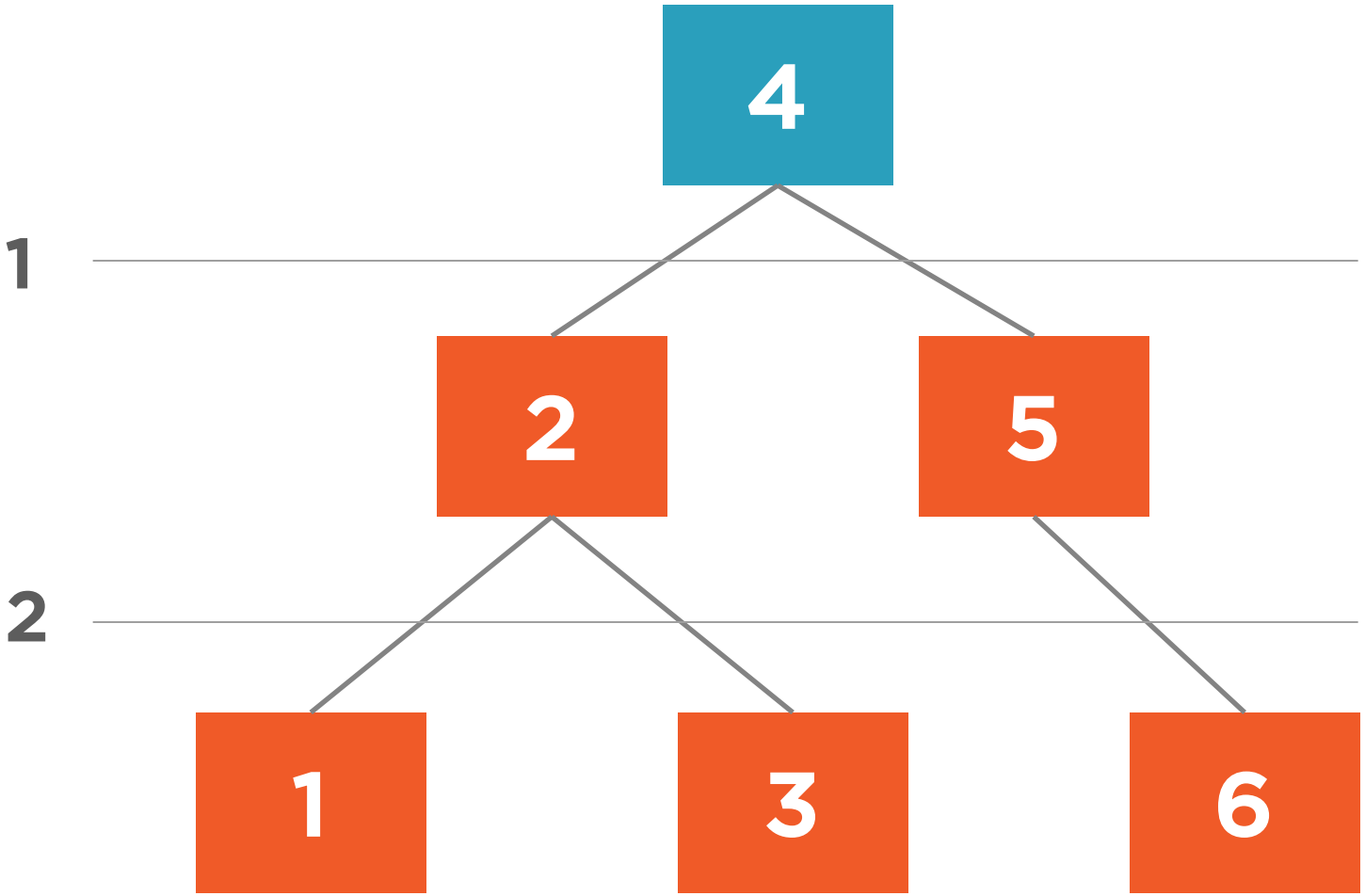

2
1
3

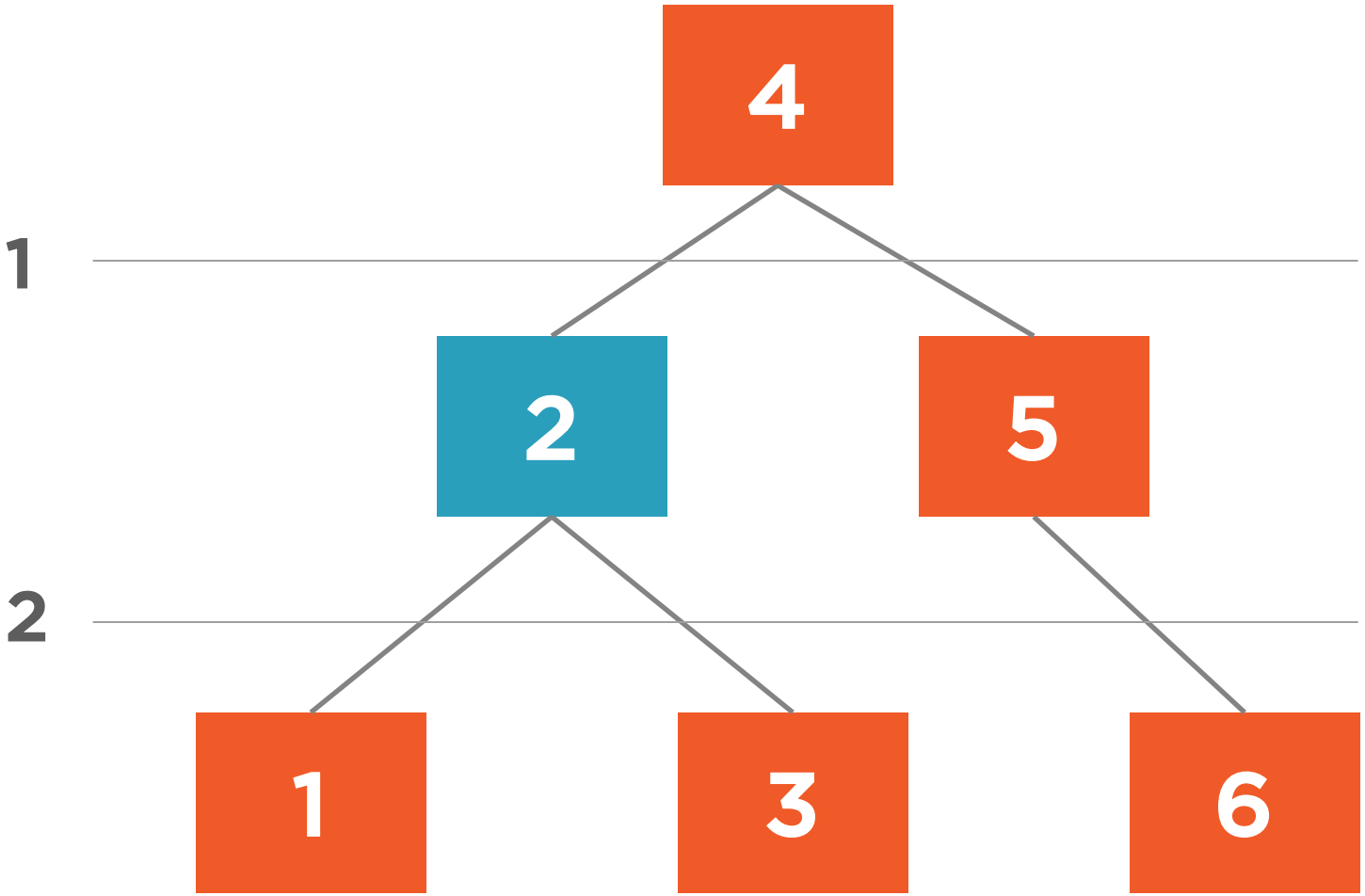

2
1
3
4

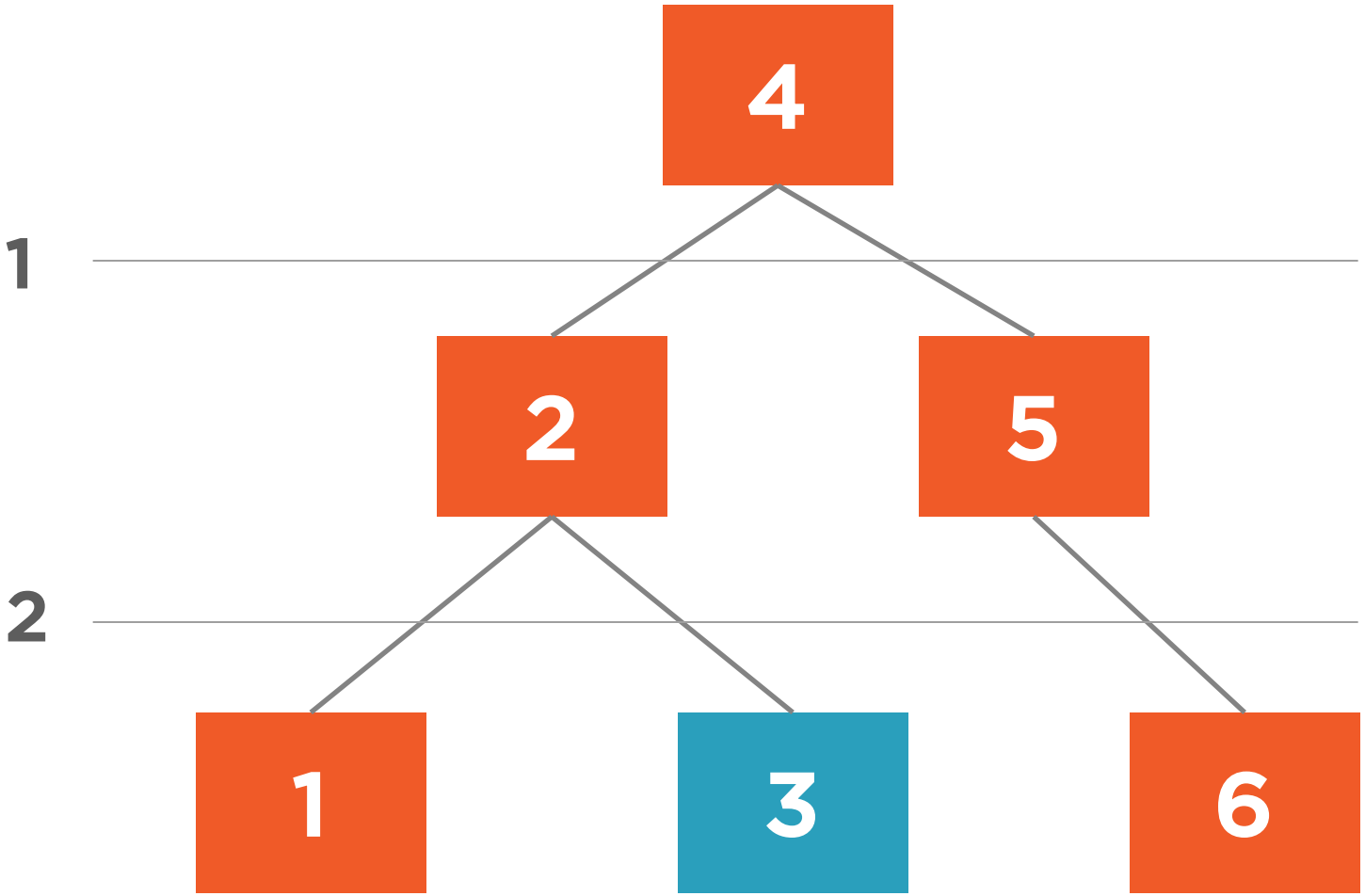

3
2
4
1
5


4
2
5
1
3
6


4
1
2
5
2
1
3
6


4
1
2
5
2
1
3
6


4
1
2
5
2
1
3
6


4
1
2
5
2
1
3
6

O(log n)

Balance Factor

The difference between the height of the left and right sub-trees.

Left Height: 0

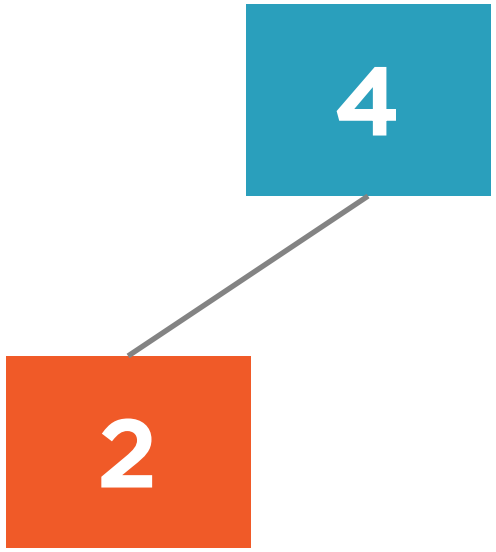
Right Height: 0

Balance Factor: 0

Left Height: 1

Right Height: 0

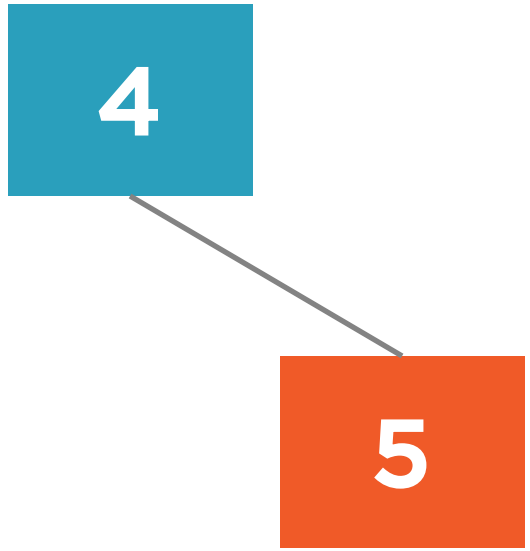
Balance Factor: -1



Left Height: 0

Right Height: 1

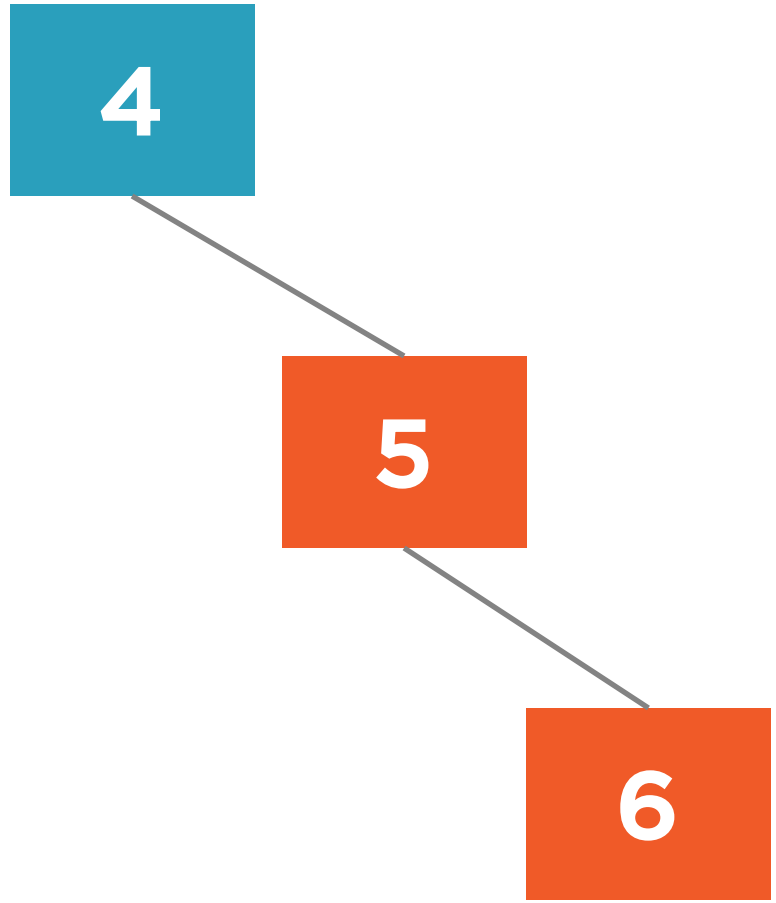
Balance Factor: 1



Left Height: 0

Right Height: 2

Balance Factor: 2



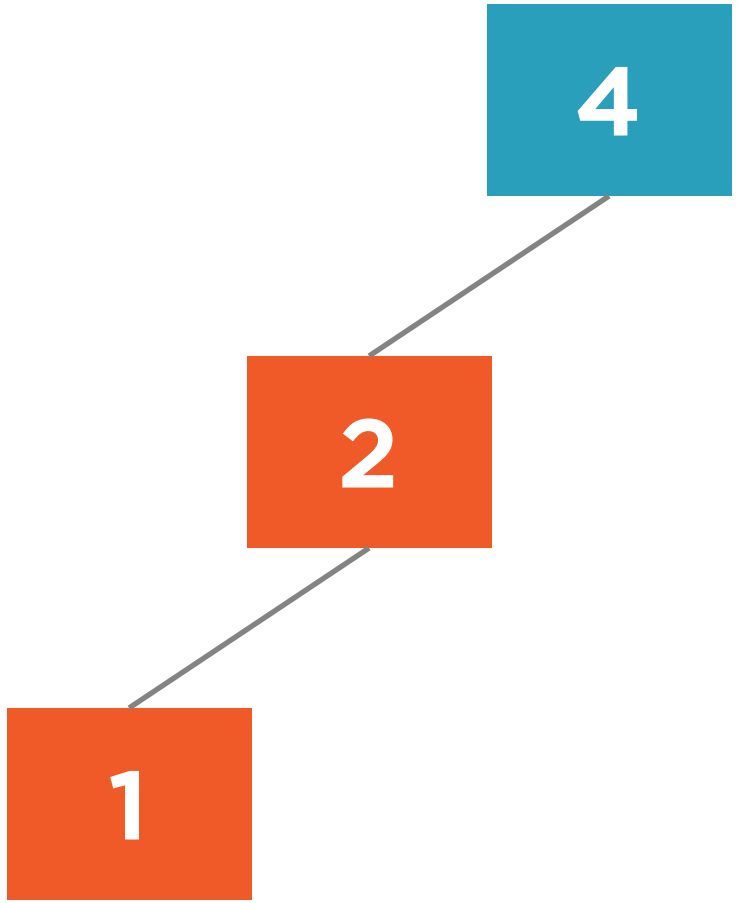
Heavy

The state when the balance factor of a node differs by more than one.

Left Height: 2

Right Height: 0

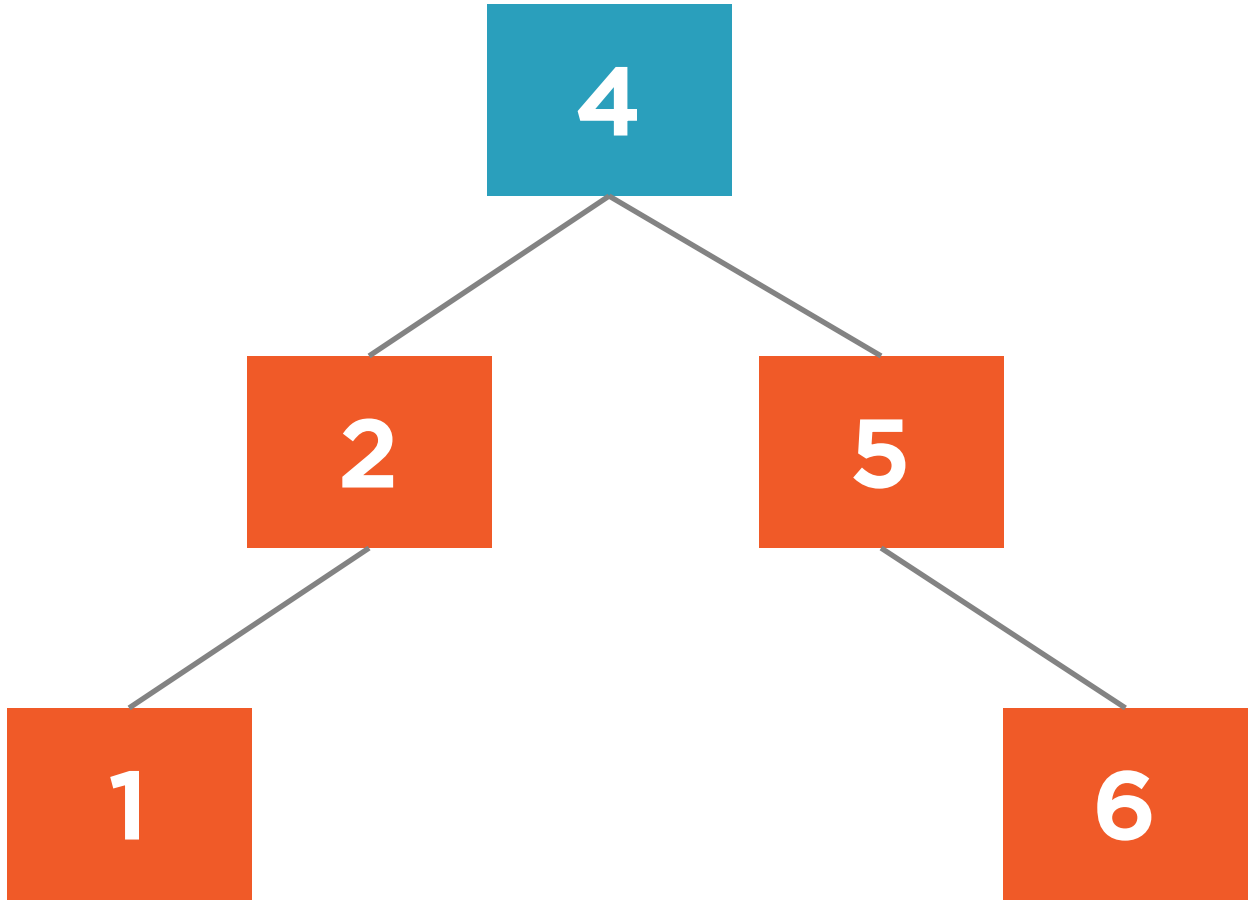
Balance Factor: -2



Left Height: 2

Right Height: 2

Balance Factor: 0



AVL Tree

AVL Tree

A self-balancing binary search tree. Named after it's inventors Georgy Adelson-Velsky and Evgenii Landis.

Self Balancing

The tree is balanced as nodes are added or removed from the tree.

Self Balancing Algorithms

Left Rotation

Right Rotation

Left-Right Rotation

Right-Left Rotation

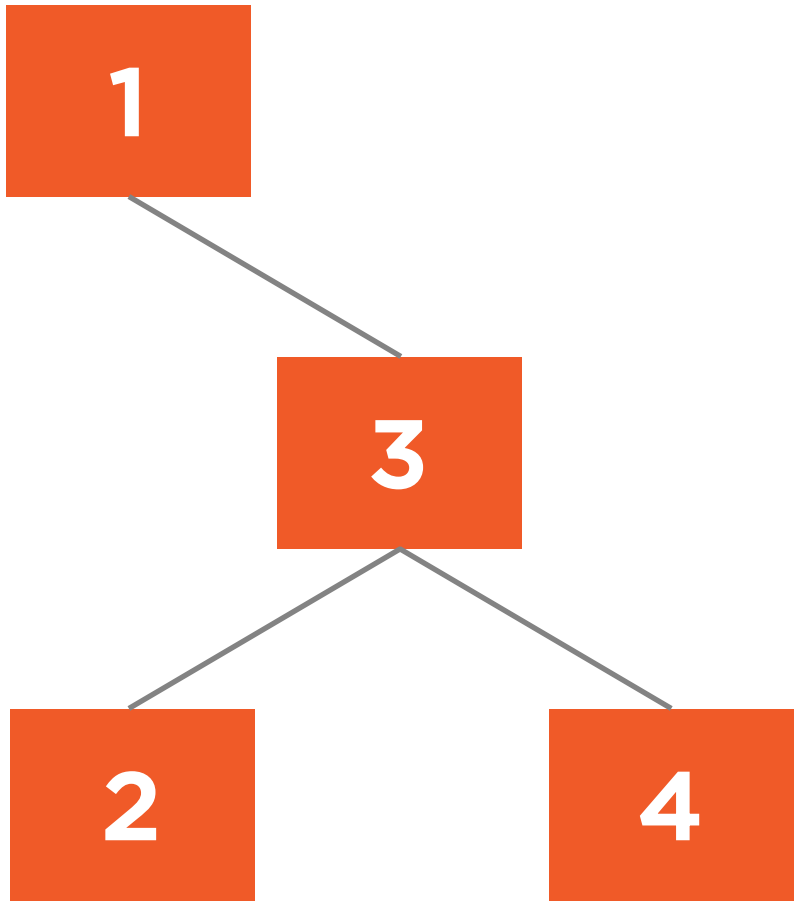
Left Rotation

Algorithm to balance a right-heavy tree by rotating nodes to the left.

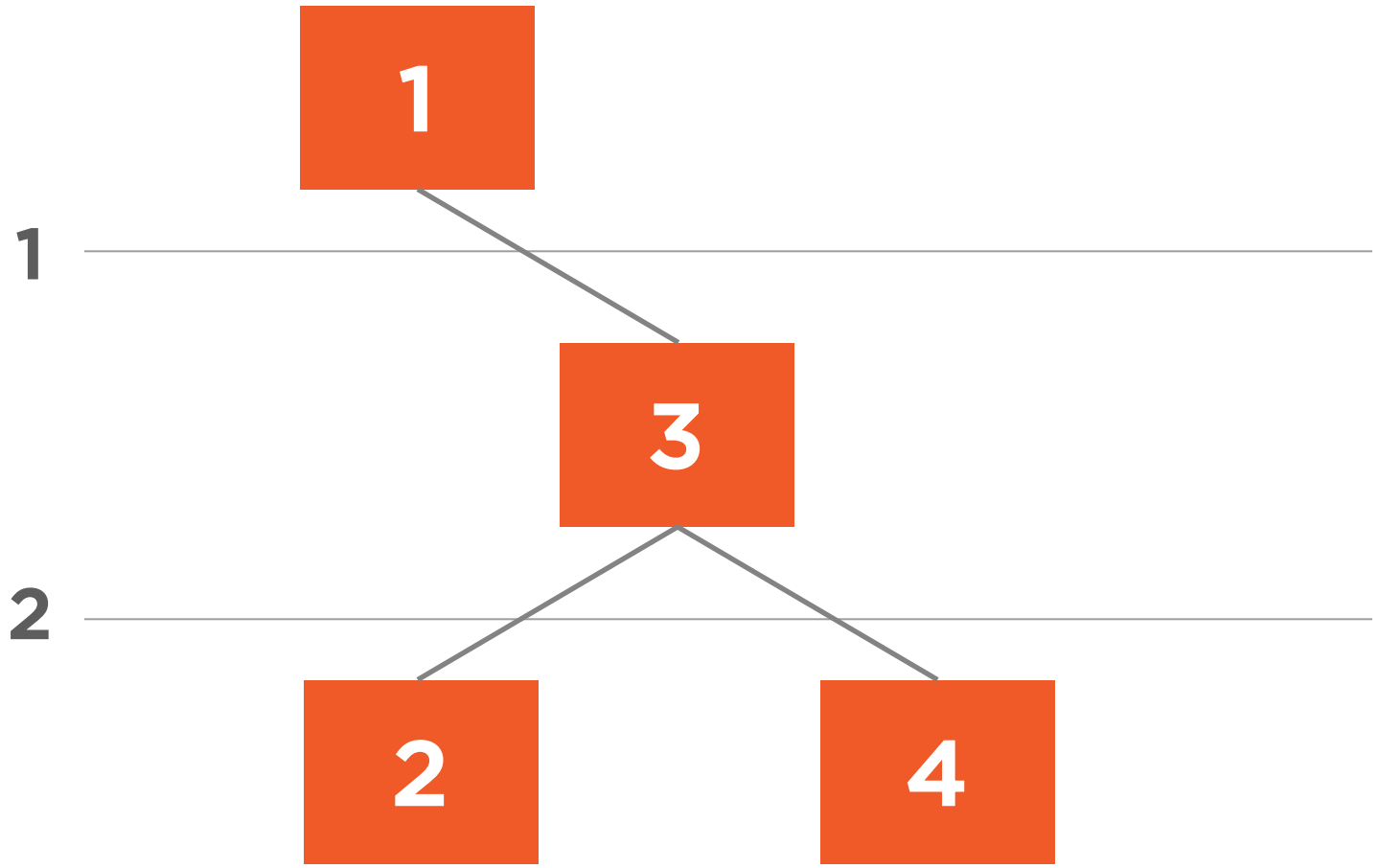
Left Rotation

1. **Right child becomes the new root**
2. **Left child of the new root is assigned to right child of the old root**
3. **Previous root becomes the new root's left child**

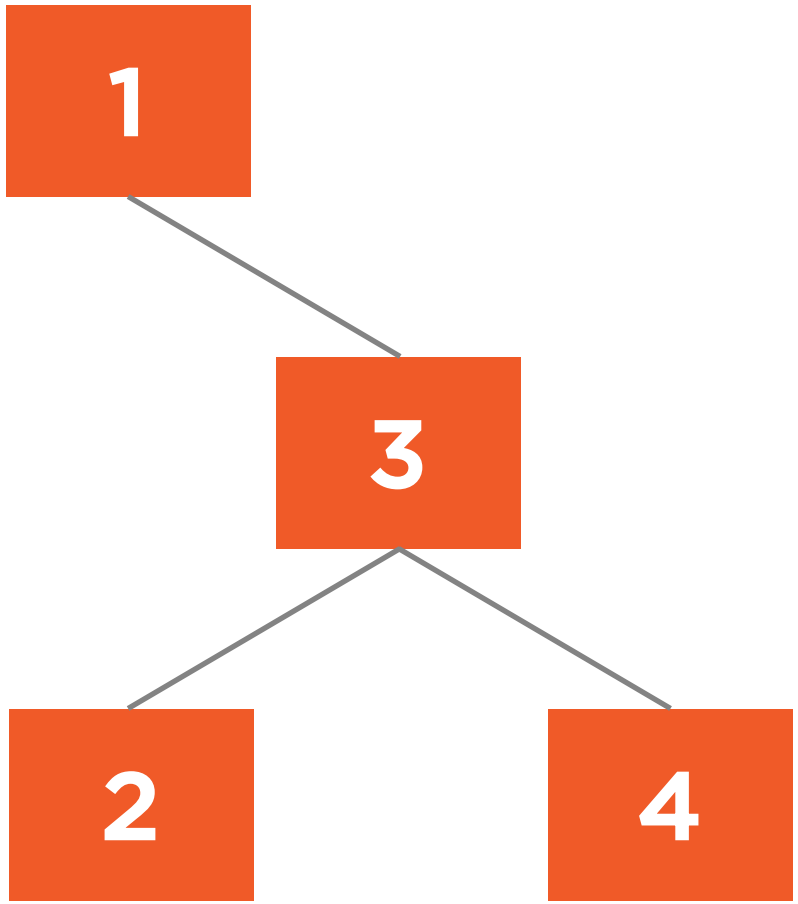
Left Rotation



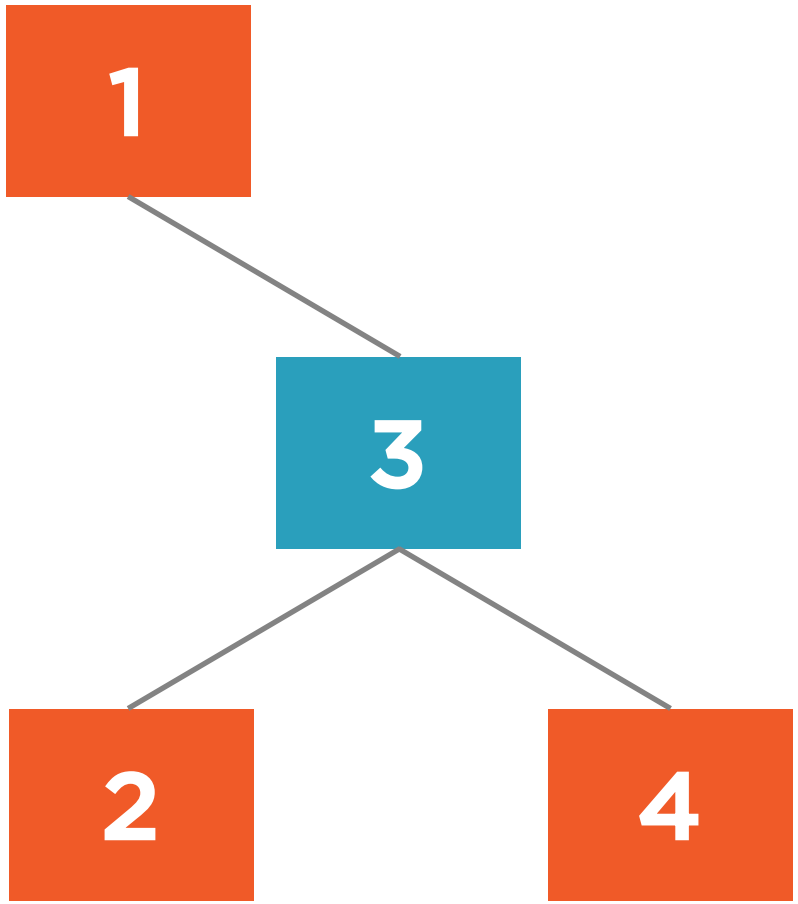
Left Rotation



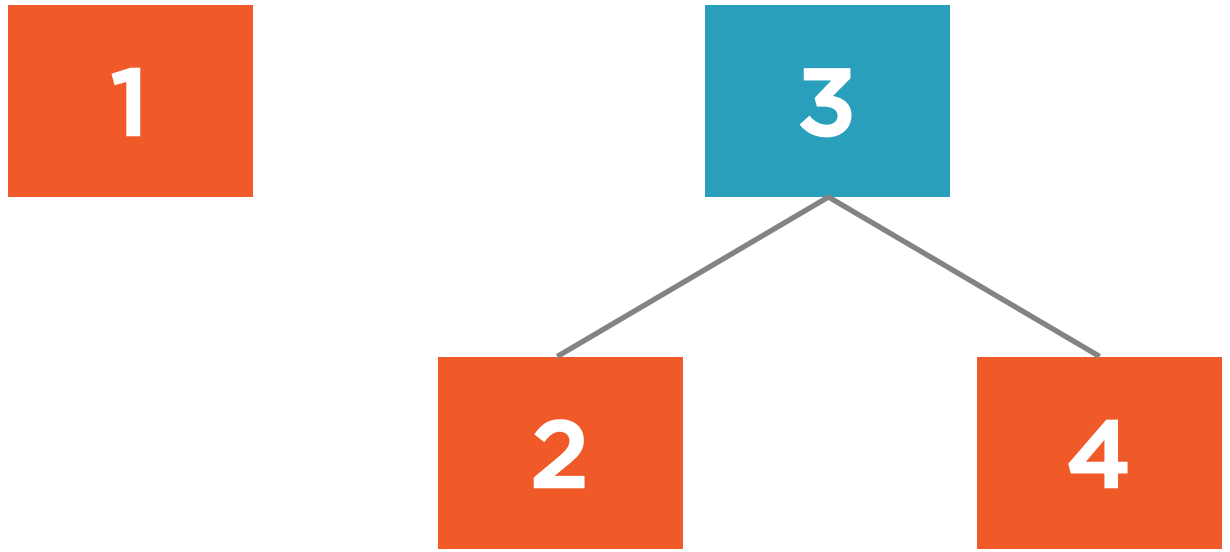
Right child becomes the new root



Right child becomes the new root

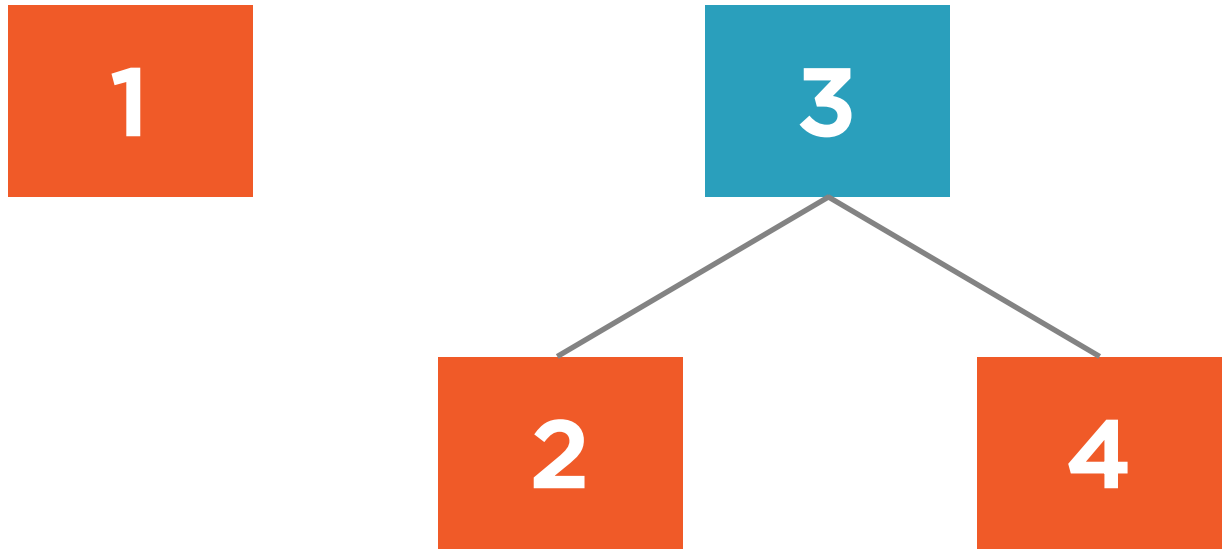


Right child becomes the new root



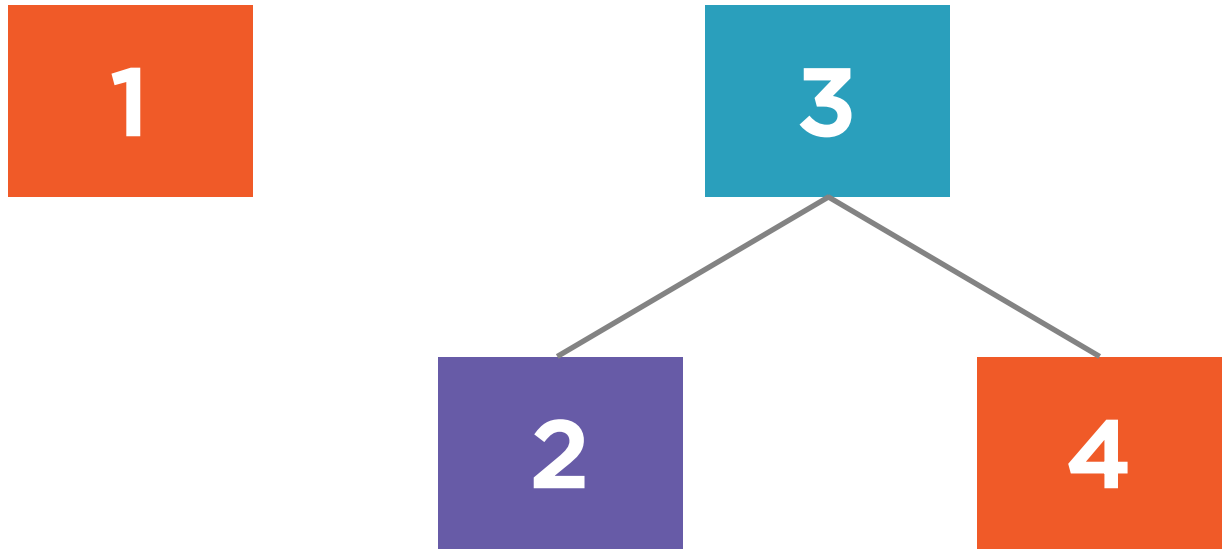
Right child becomes the new root

Left child of the new root is assigned to right child of the old root



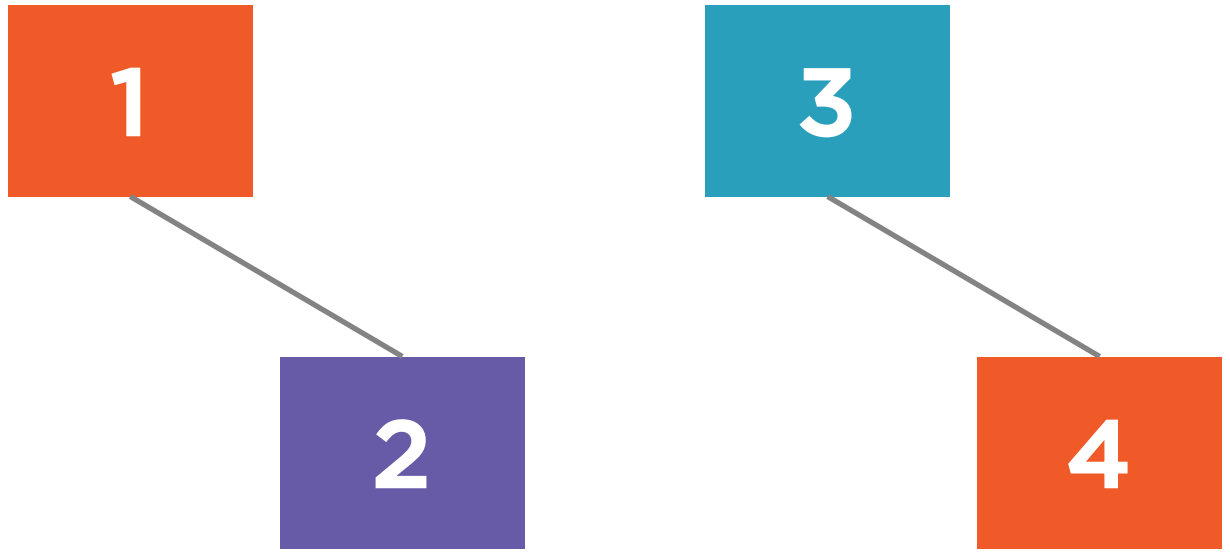
Right child becomes the new root

Left child of the new root is assigned to right child of the old root



Right child becomes the new root

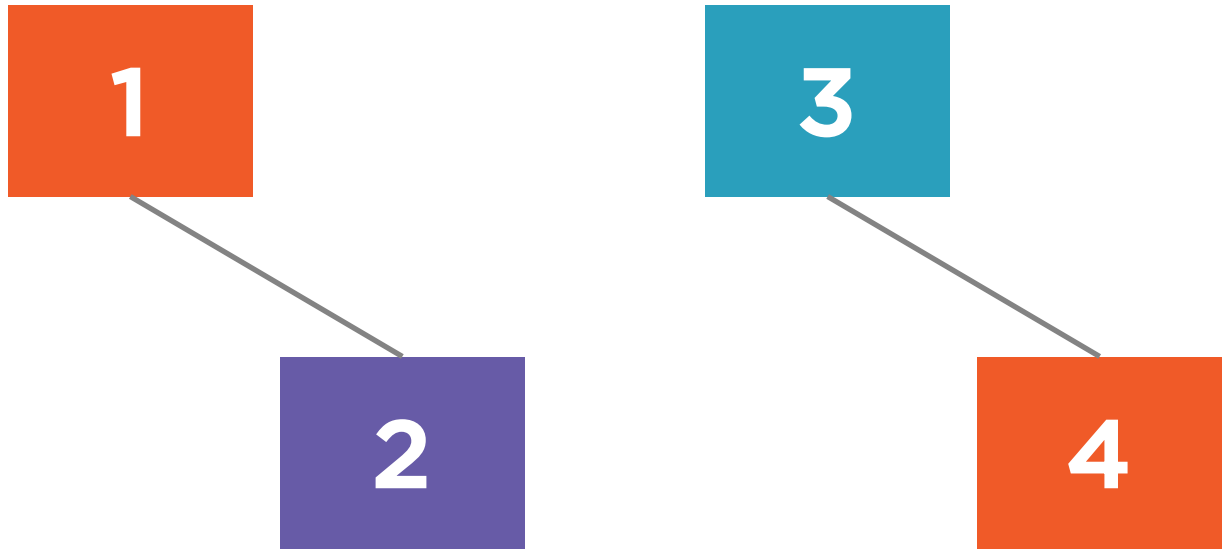
Left child of the new root is assigned to right child of the old root



Right child becomes the new root

Left child of the new root is assigned to right child of the old root

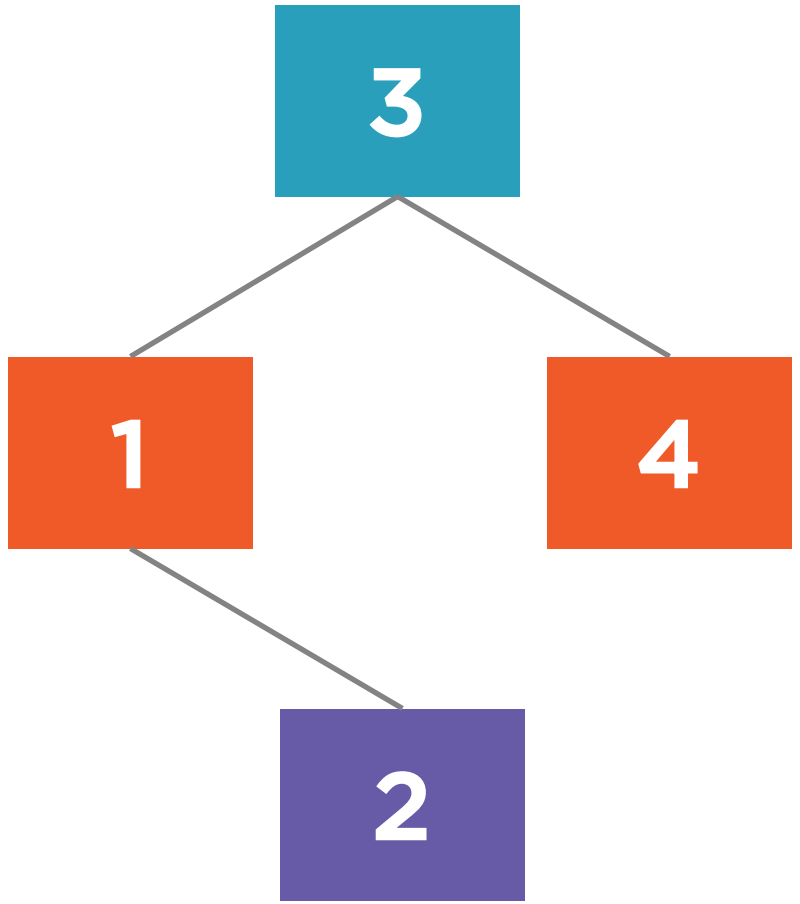
Previous root becomes the new root's left child



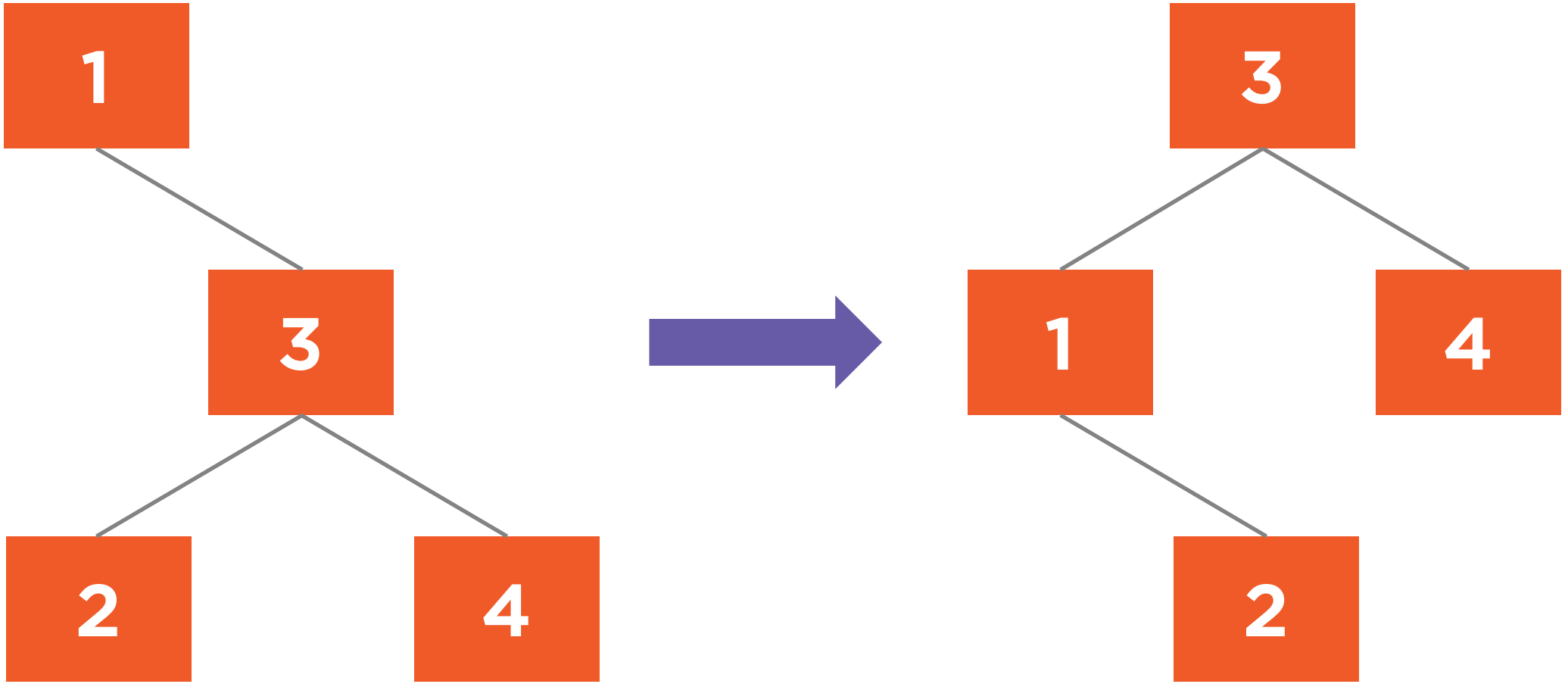
Right child becomes the new root

Left child of the new root is assigned to right child of the old root

Previous root becomes the new root's left child



Before and After: Left Rotation



Unbalanced

Balanced

```
private void LeftRotation() {
    AVLTreeNode<T> newRoot = Right;

    ReplaceRootWith(newRoot);

    Right = newRoot.Left;

    newRoot.Left = this;
}
```

- Replace root with right child
- Set right child to be left of new root
- Set left node of new root to current

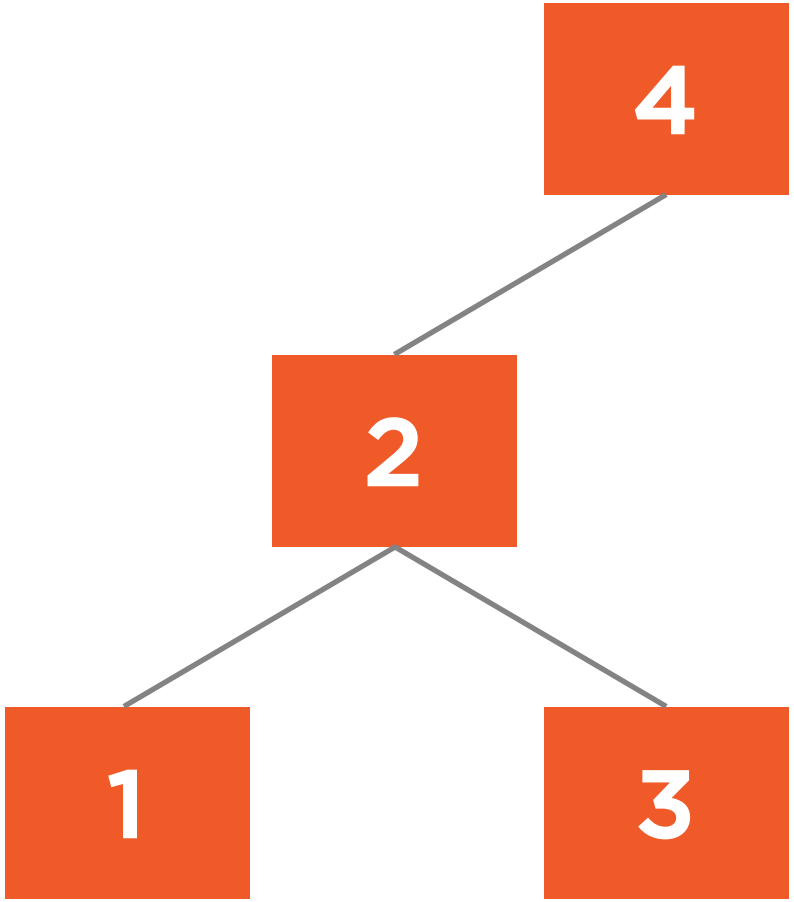
Right Rotation

Algorithm to balance a left-heavy tree by rotating nodes to the right.

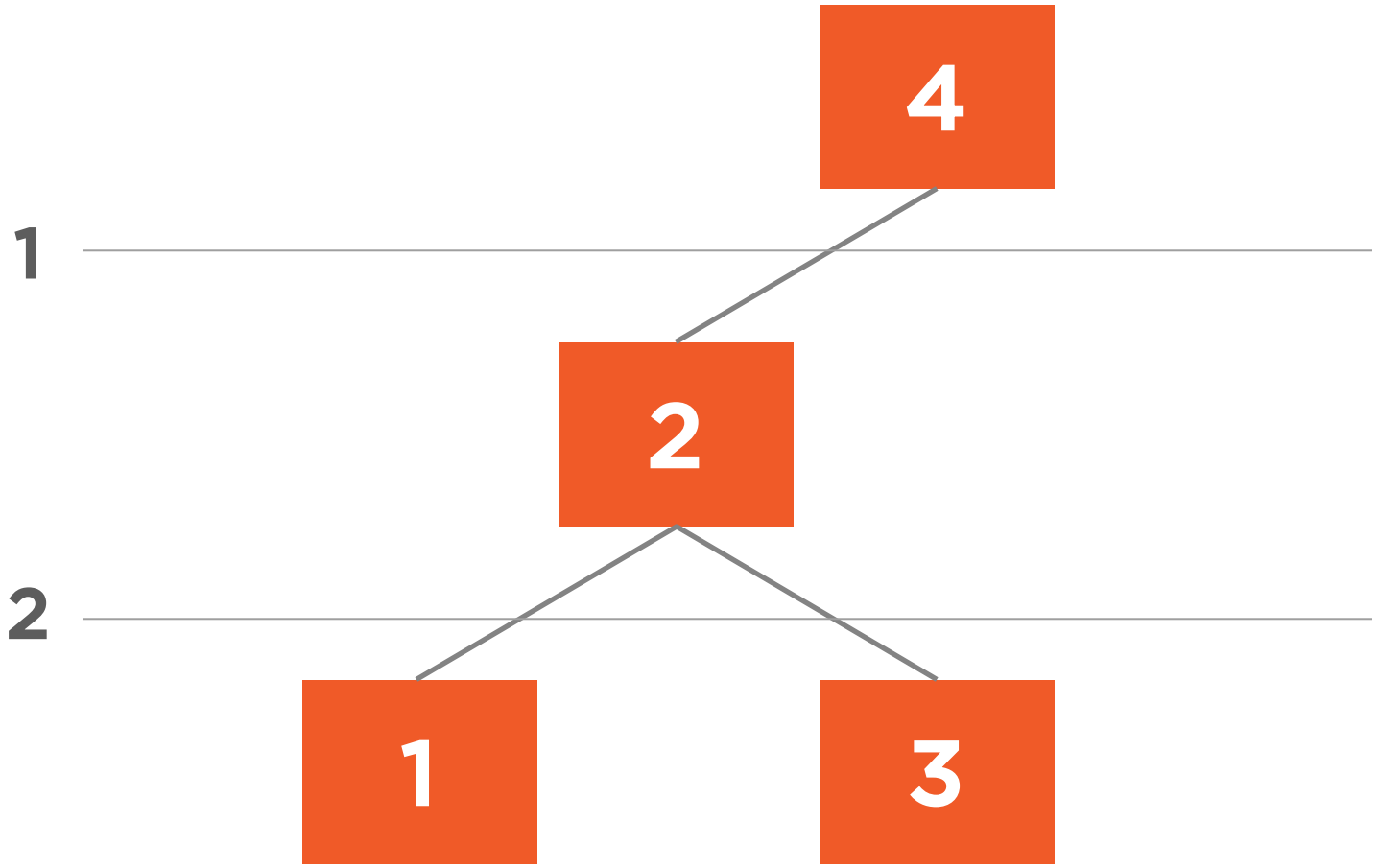
Right Rotation

1. **Left child becomes the new root**
2. **Right child of the new root is assigned to left child of the old root**
3. **Previous root becomes the new root's right child**

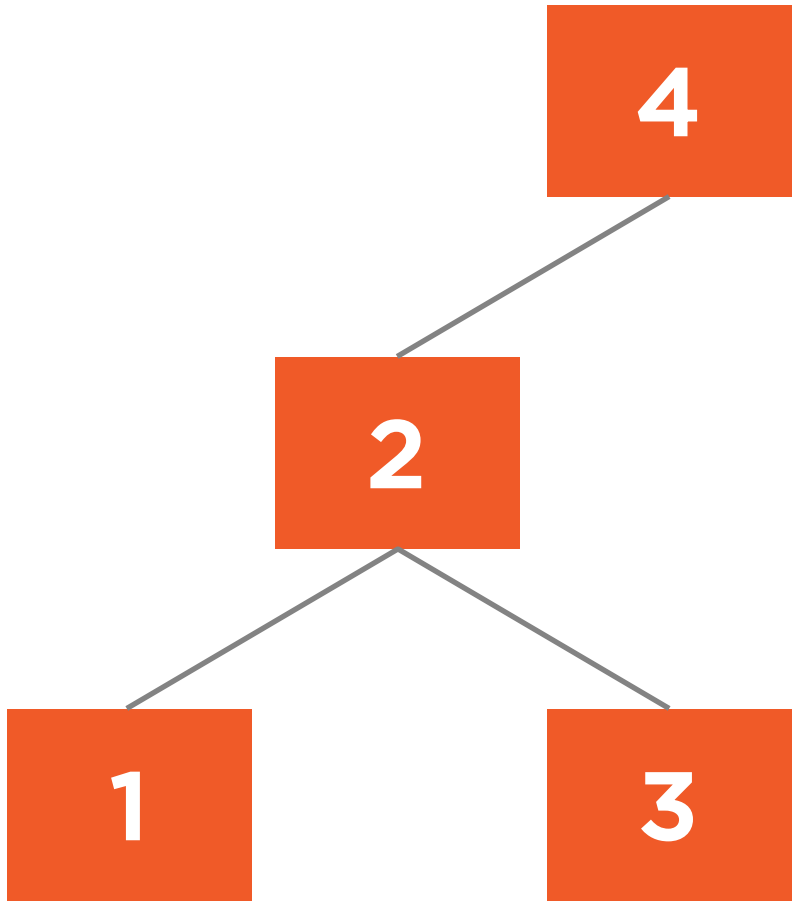
Right Rotation



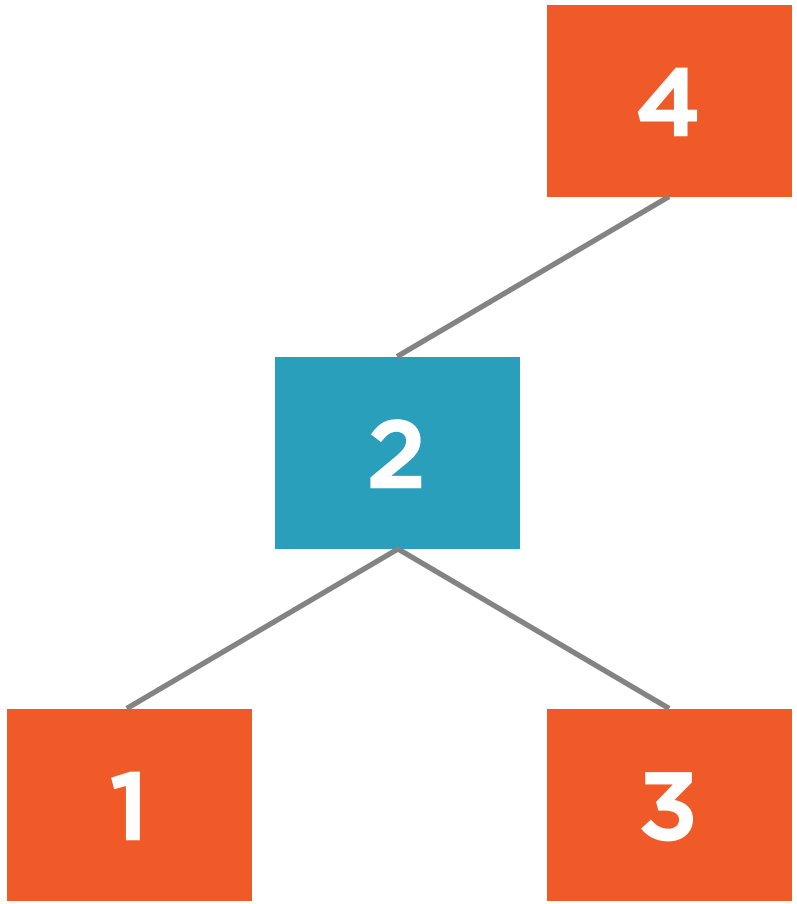
Right Rotation



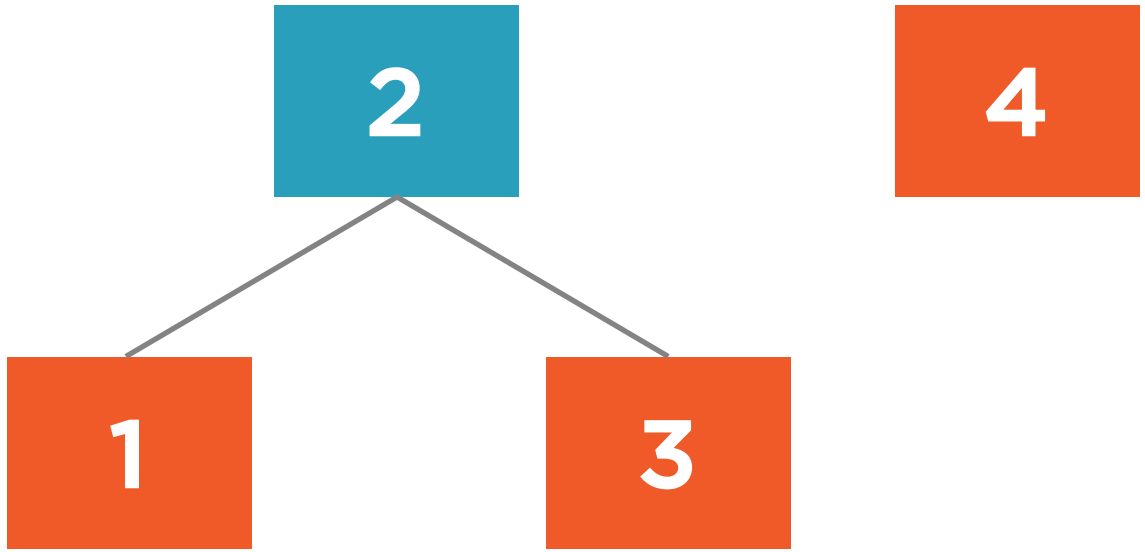
Left child becomes the new root



Left child becomes the new root

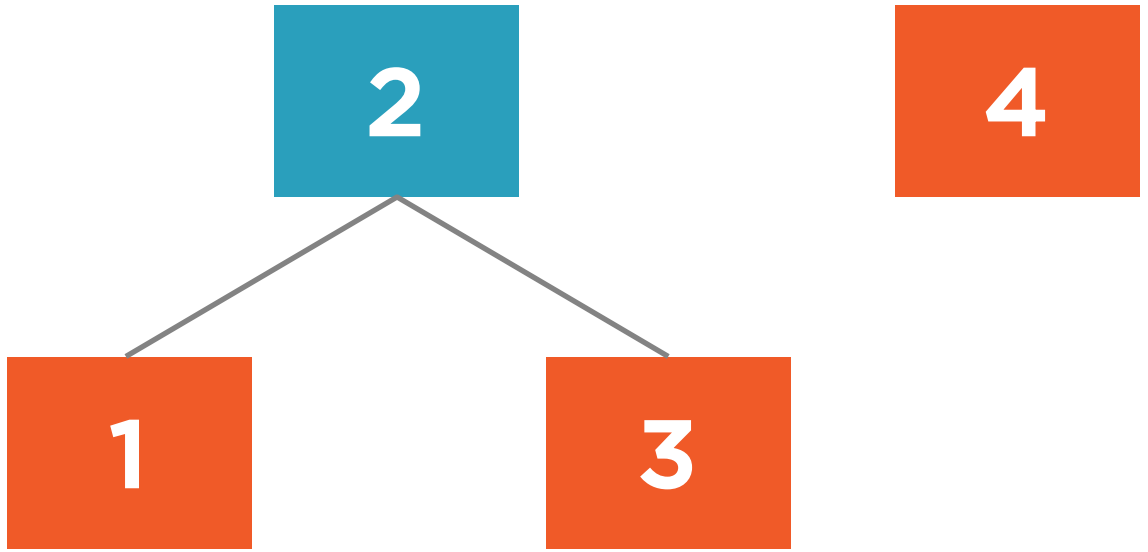


Left child becomes the new root



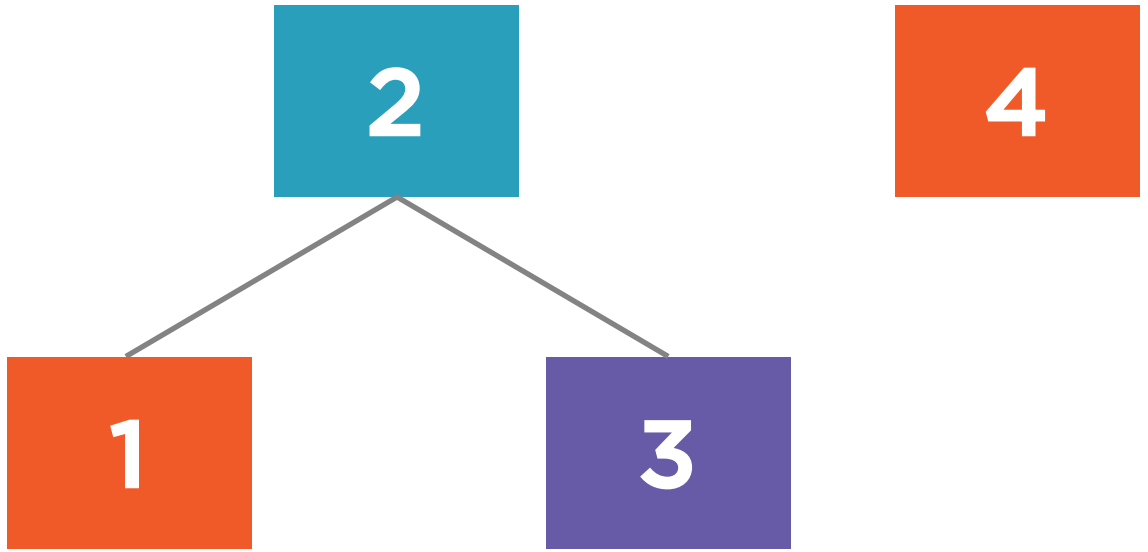
Left child becomes the new root

Right child of the new root is assigned to the left child of the old root



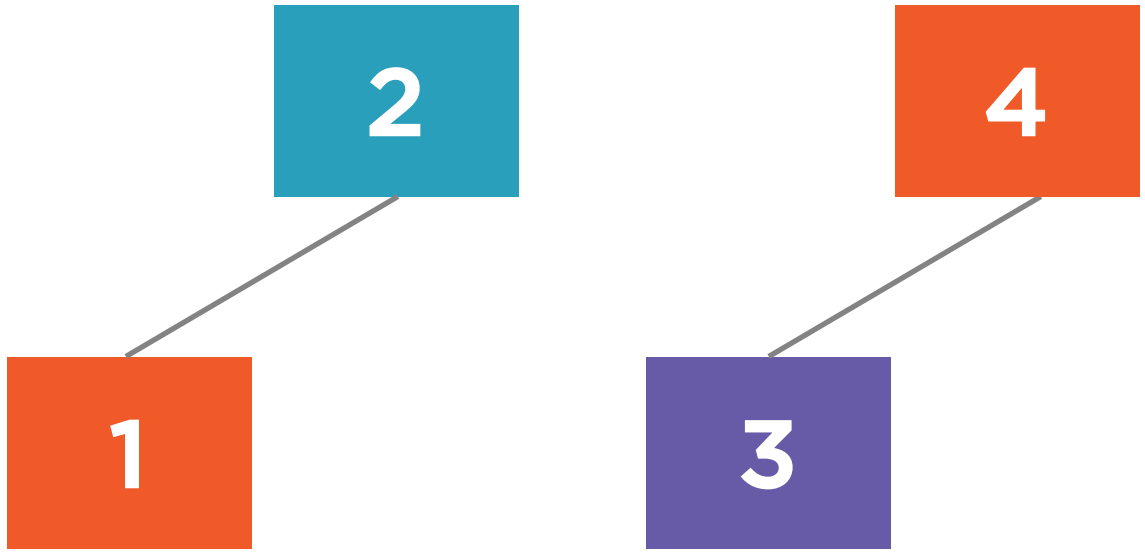
Left child becomes the new root

Right child of the new root is assigned to the left child of the old root



Left child becomes the new root

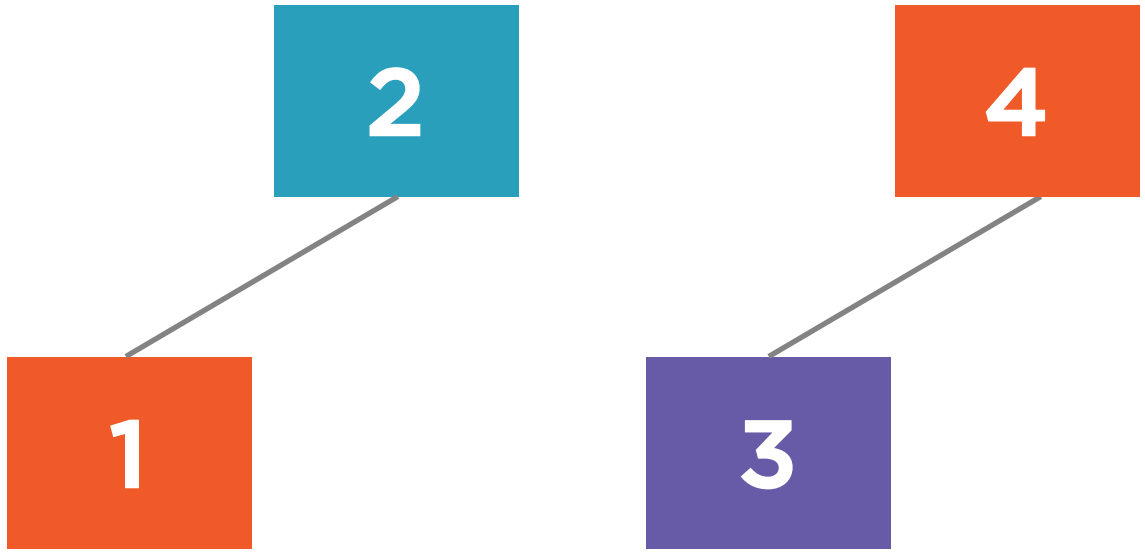
Right child of the new root is assigned to the left child of the old root



Left child becomes the new root

Right child of the new root is assigned to the left child of the old root

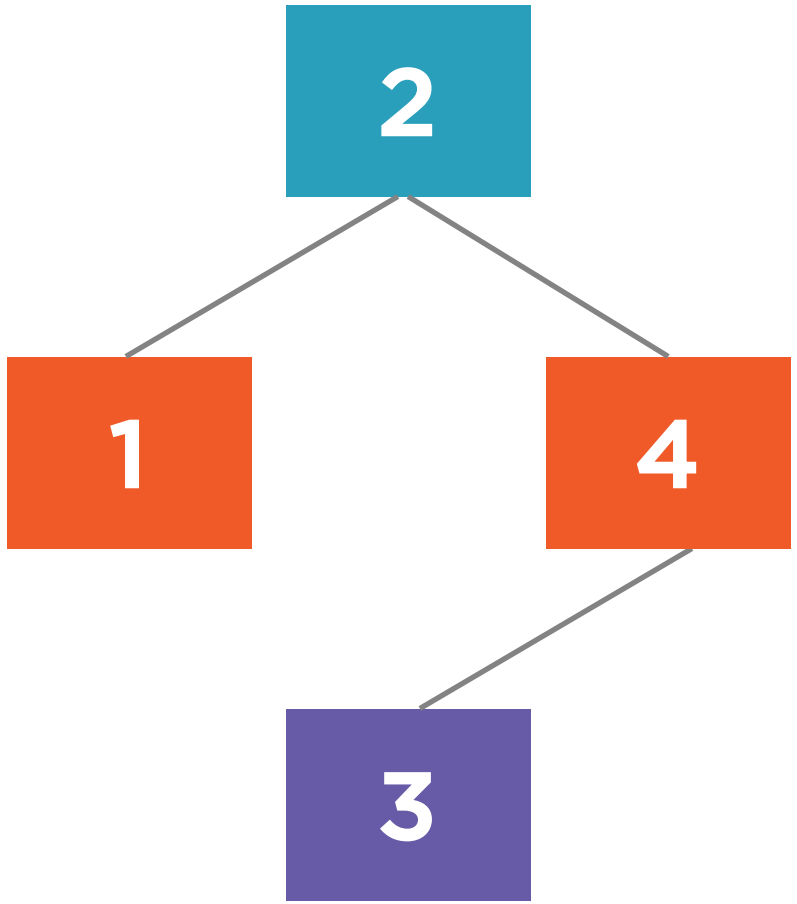
Previous root becomes **the new root's right child**



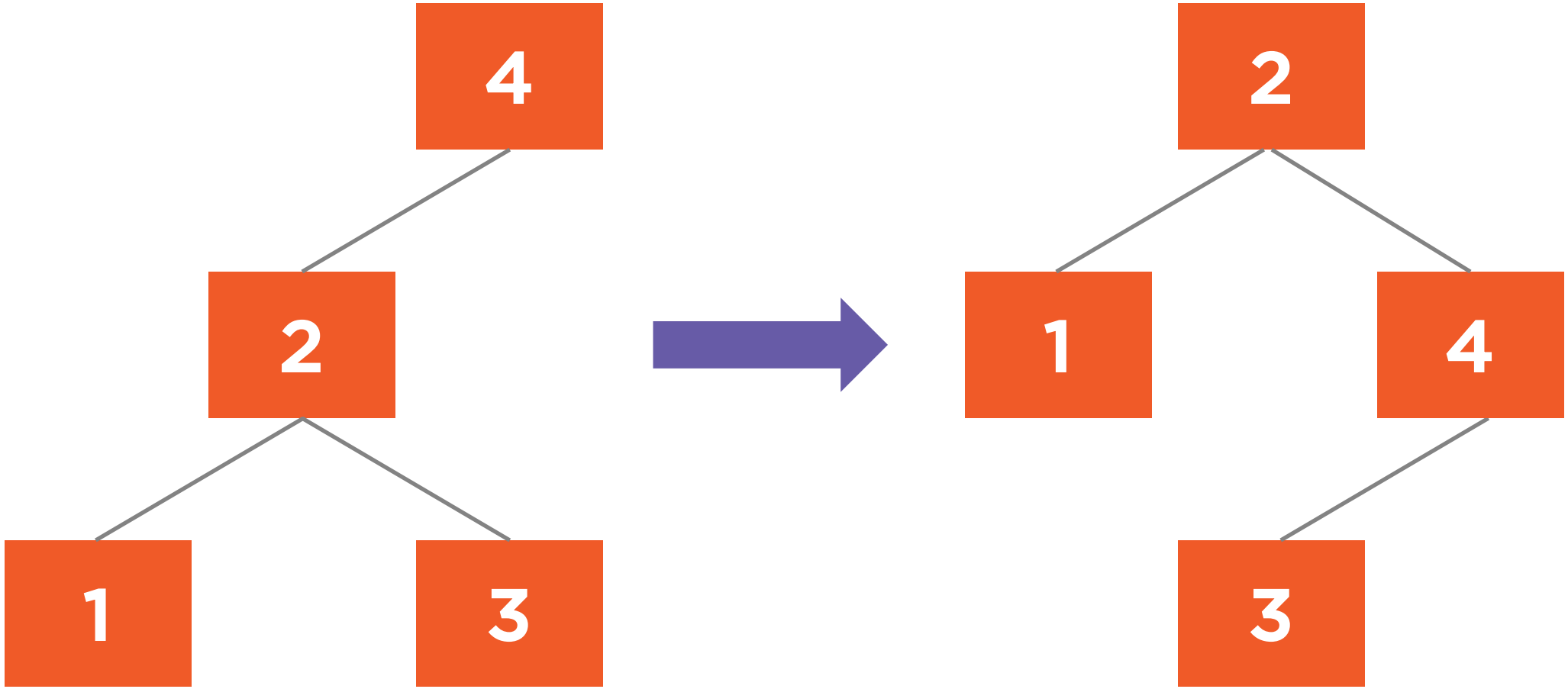
Left child becomes the new root

Right child of the new root is assigned to the left child of the old root

Previous root becomes **the new root's right child**



Before and After: Right Rotation



Unbalanced

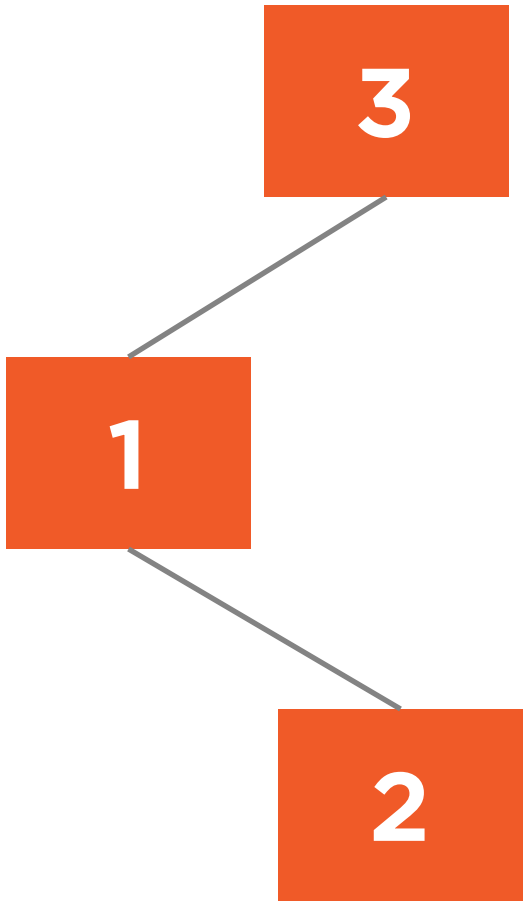
Balanced

```
private void RightRotation() {
    AVLTreeNode<T> newRoot = Left;

    ReplaceRootWith(newRoot);

    Left = newRoot.Right;

    newRoot.Right = this;
}
```

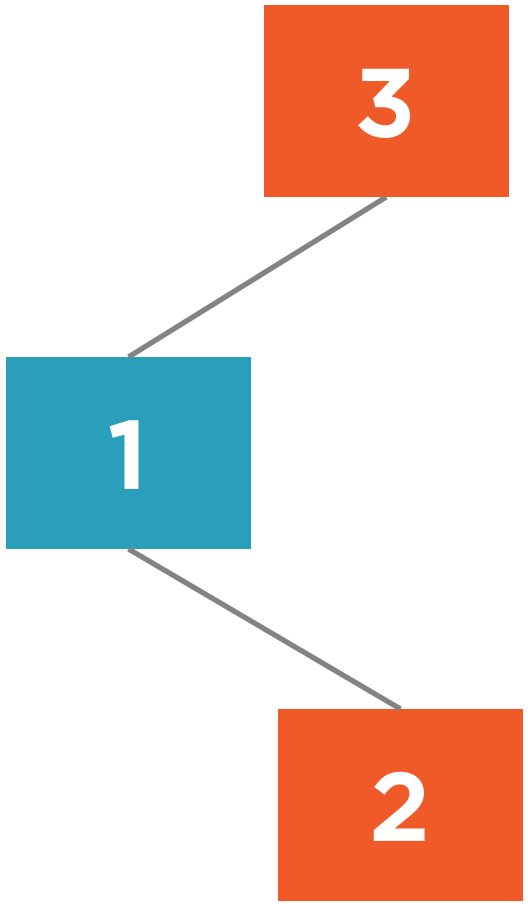
- Replace root with left child
- Set left child to be right of new root
- Set right node of new root to current

Right and left rotations don’t always solve balancing problems

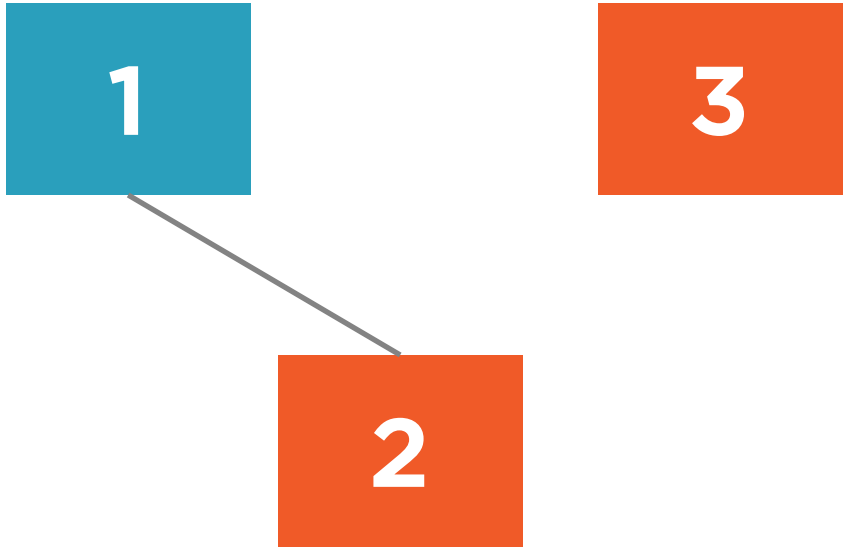
Unbalanced Right Rotation



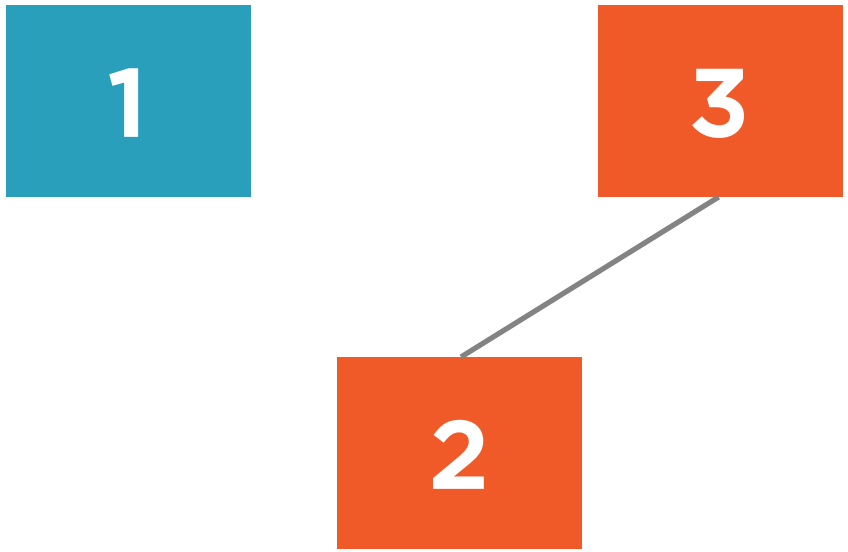
Unbalanced Right Rotation



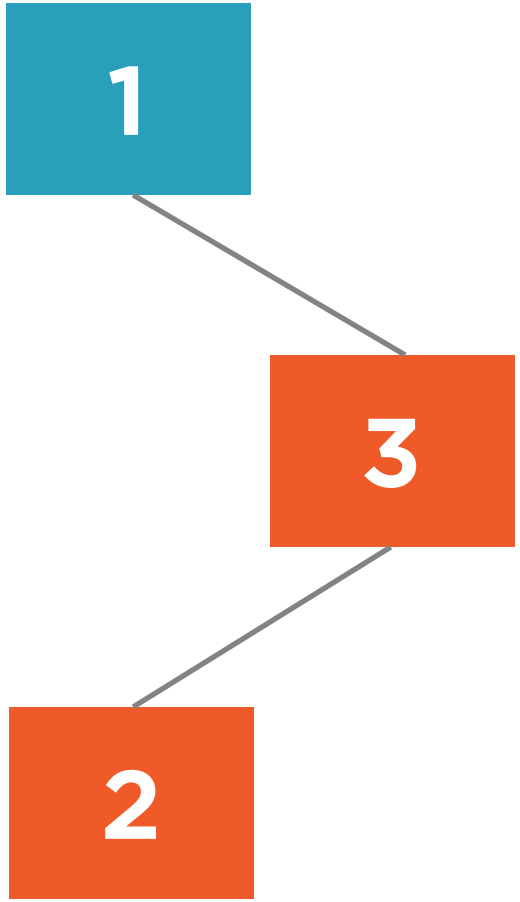
Unbalanced Right Rotation



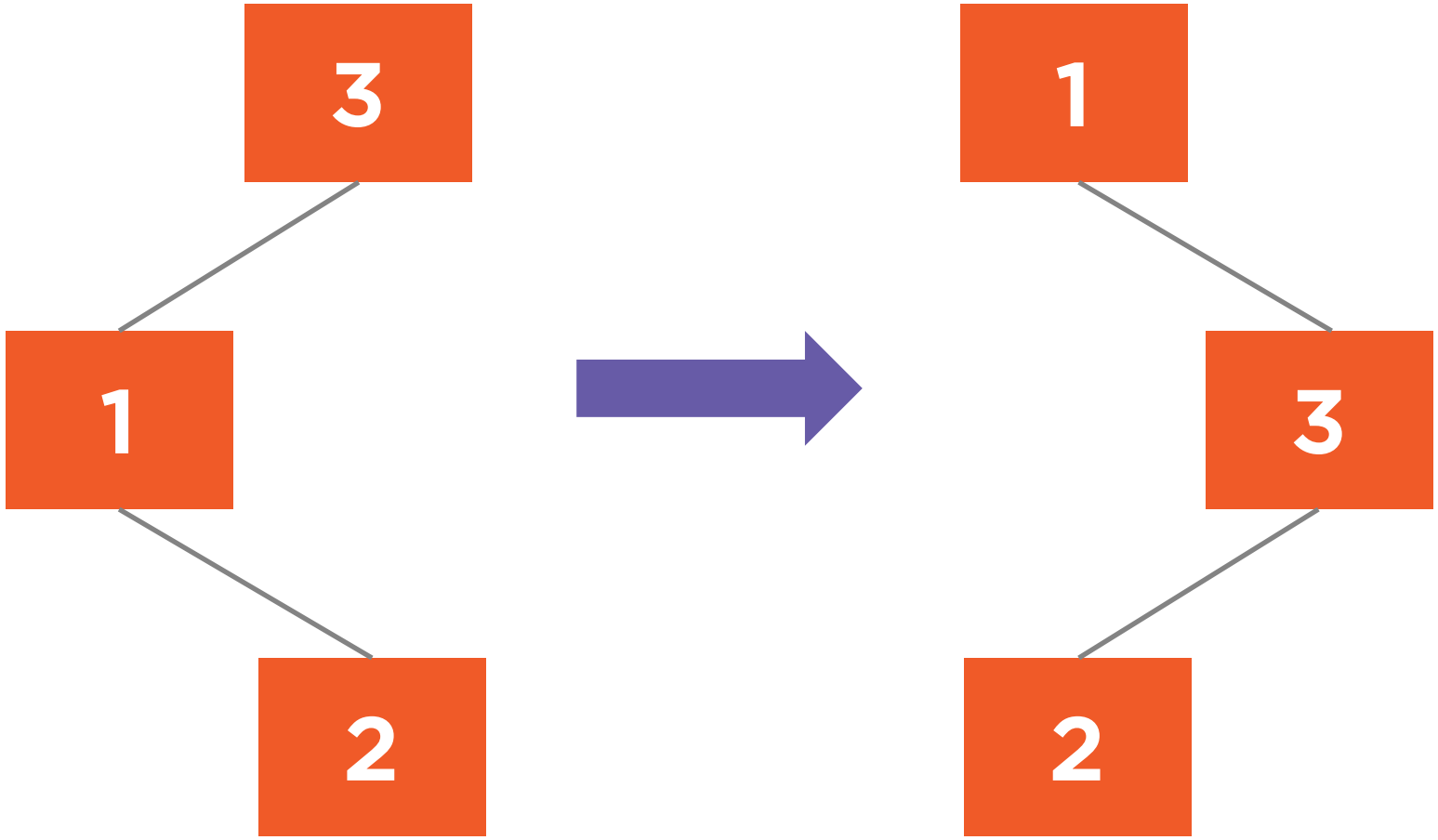
Unbalanced Right Rotation



Unbalanced Right Rotation



Unbalanced Right Rotation



Right-Left Rotation

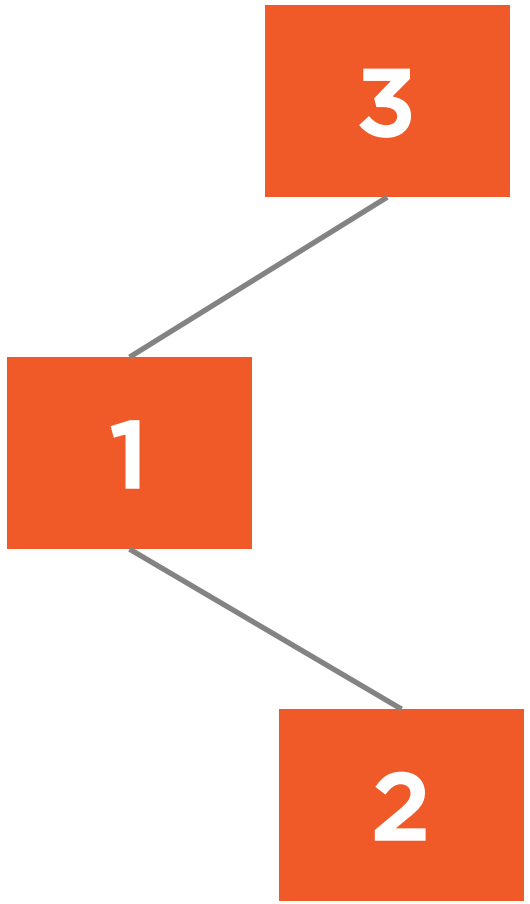
A right-rotation of a left-rotated tree.

Right-Left Rotation

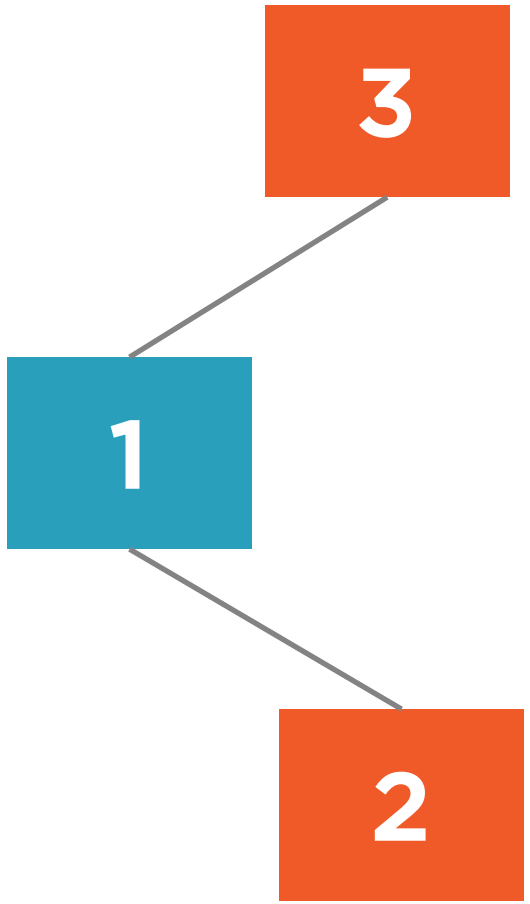
1. **Left rotate the left child**

2. **Right rotate the root**

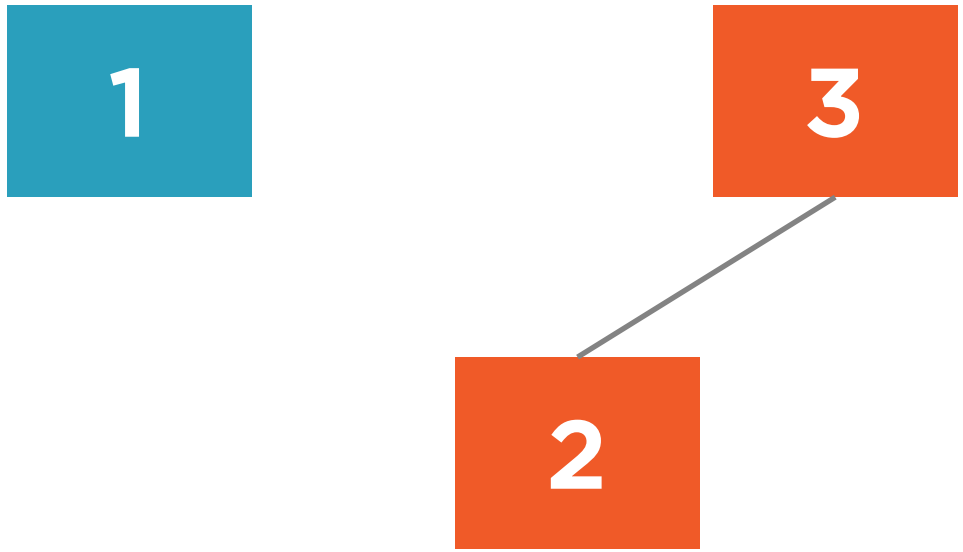
Left rotate the left child



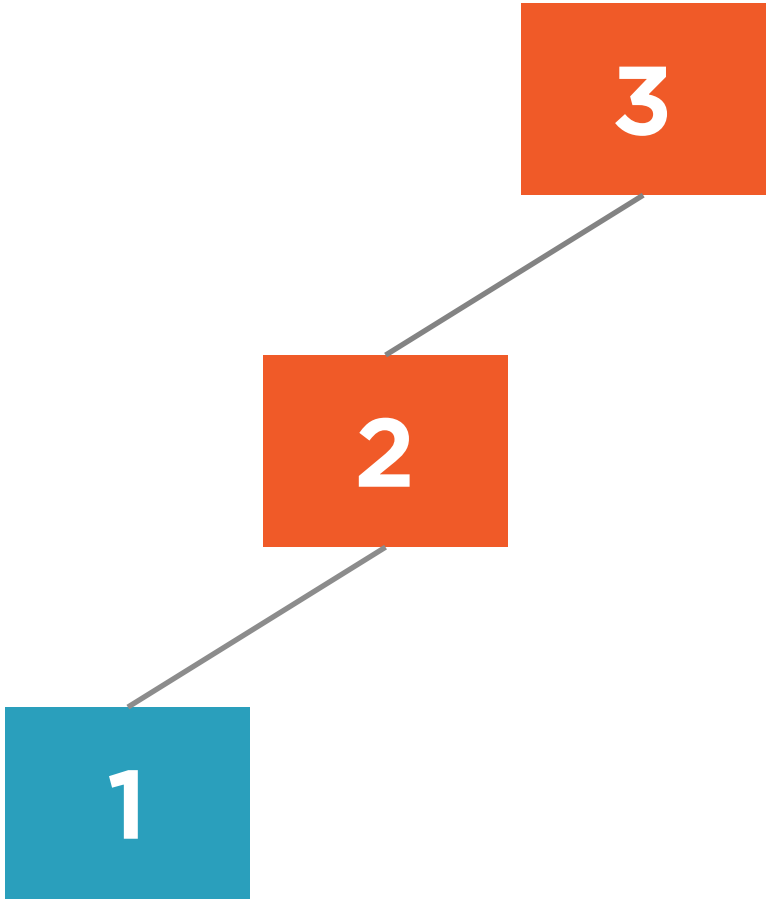
Left rotate the left child



Left rotate the left child

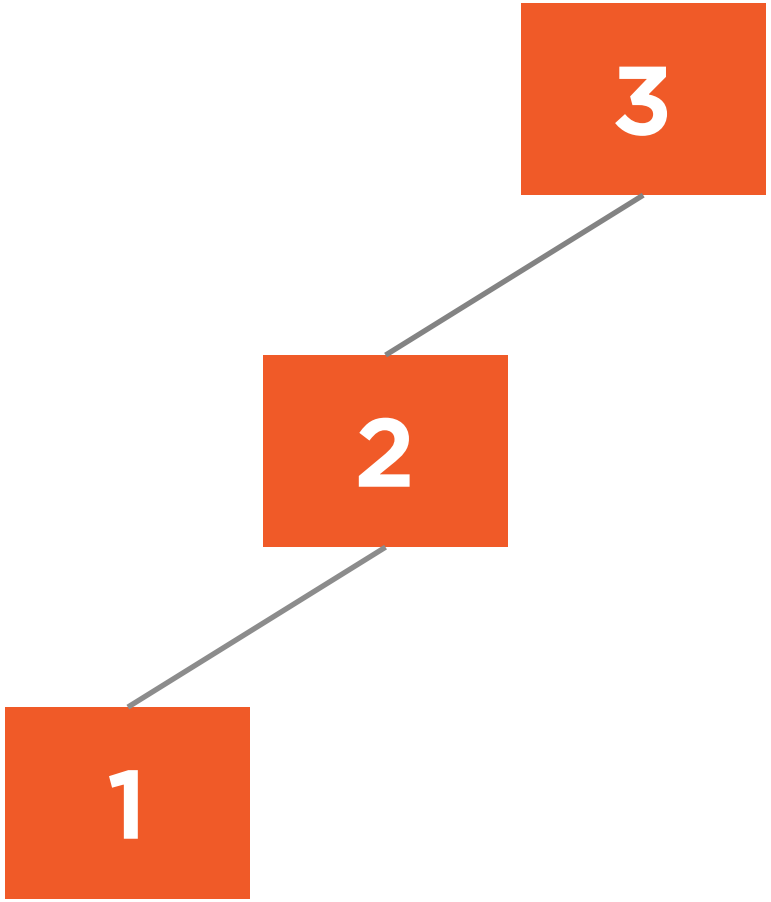


Left rotate the left child



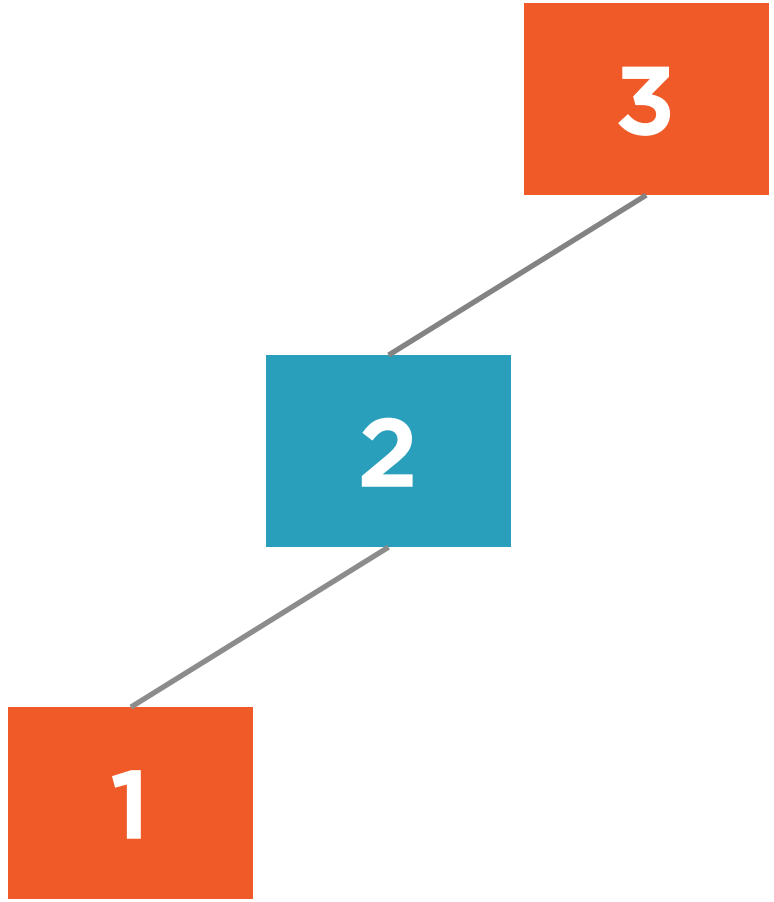
Left rotate the left child

Right rotate the root



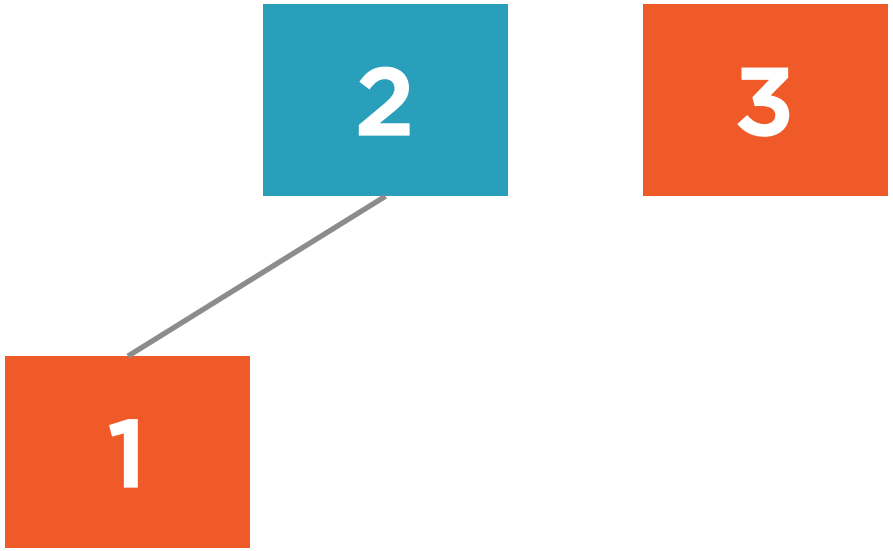
Left rotate the left child

Right rotate the root



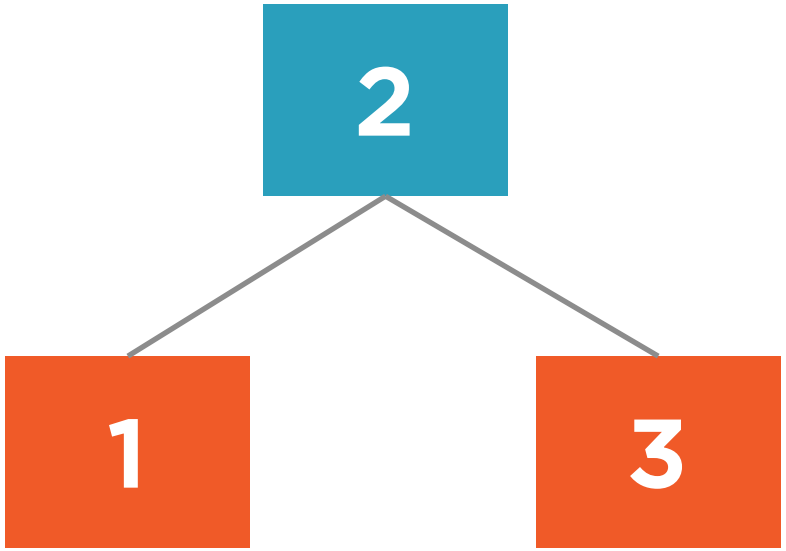
Left rotate the left child

Right rotate the root

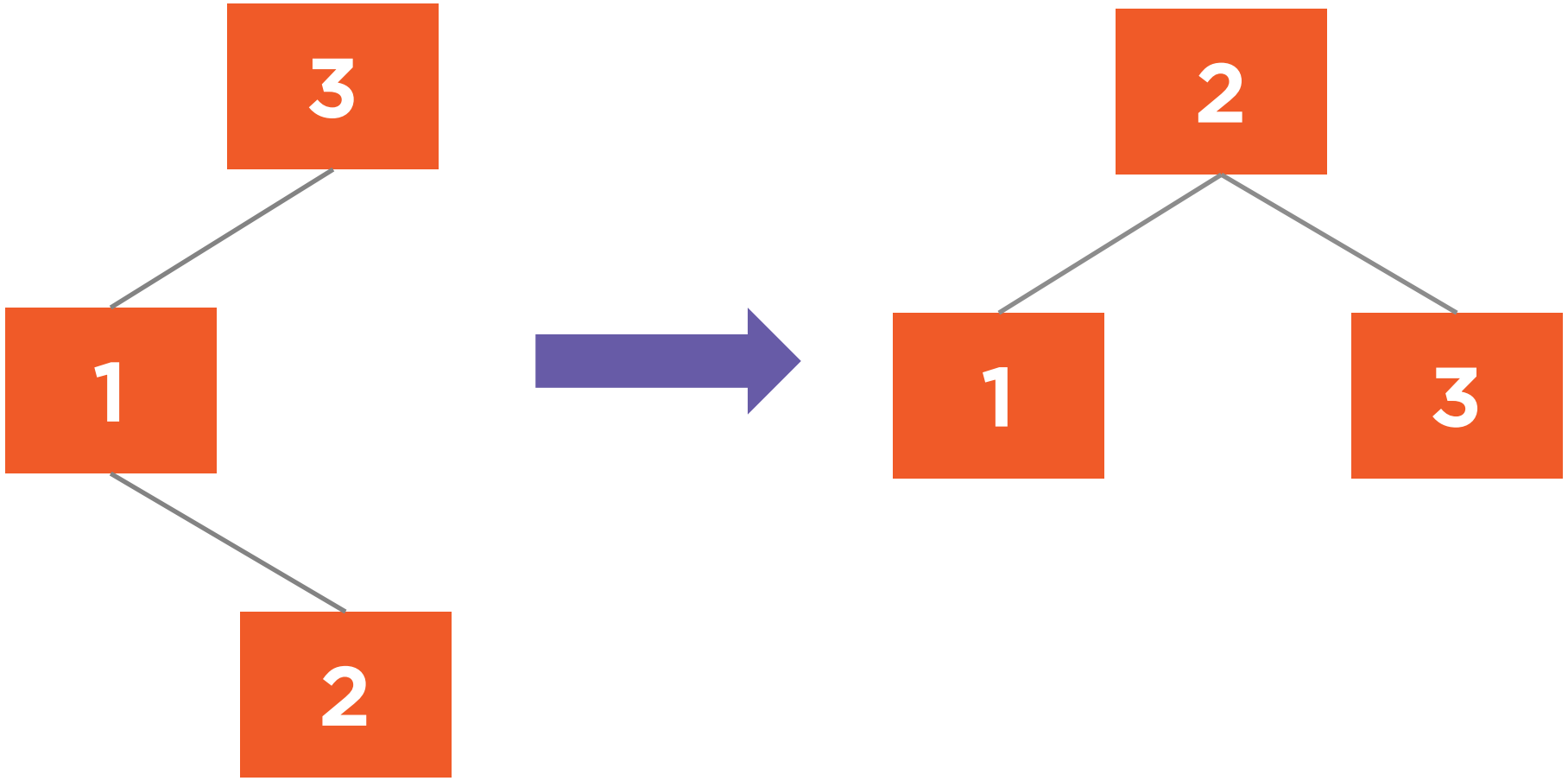


Left rotate the left child

Right rotate the root



Before and After: Right-Left Rotation



Left-Right Rotation

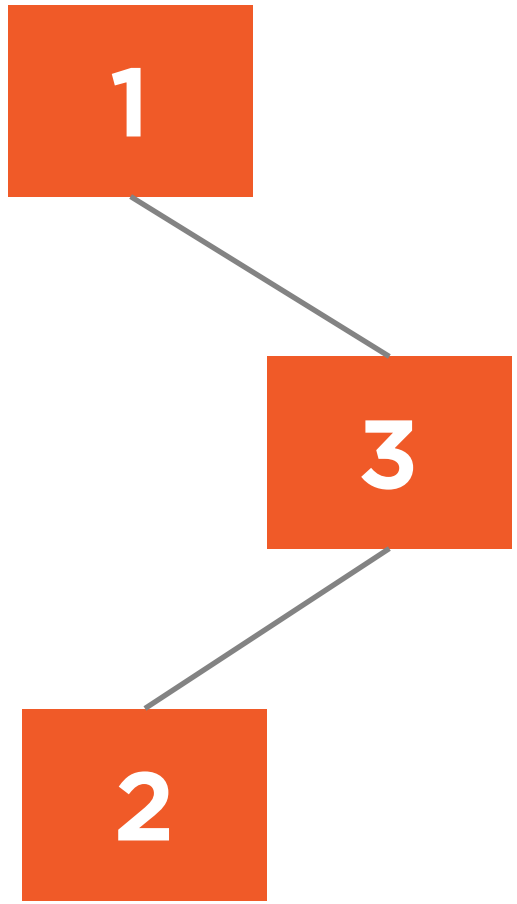
A left-rotation of a right-rotated tree.

Left-Right Rotation

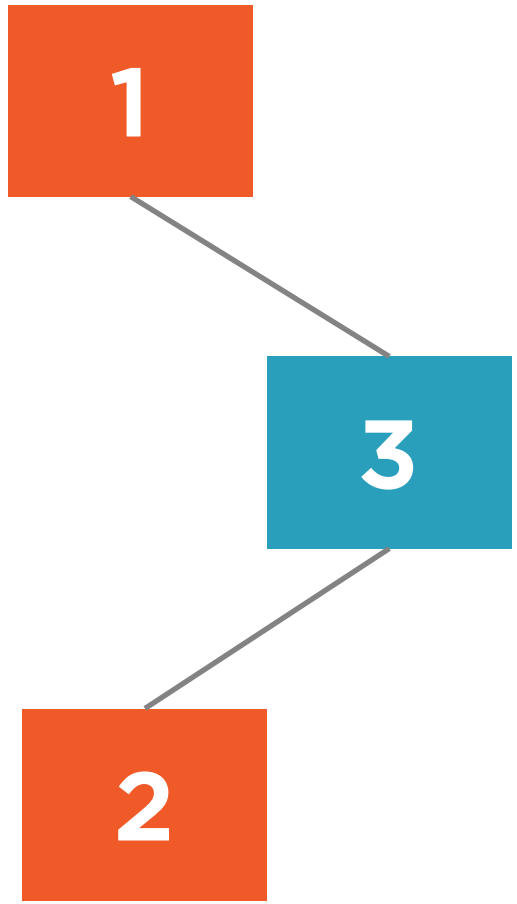
1. **Right rotate the right child**

2. **Left rotate the root**

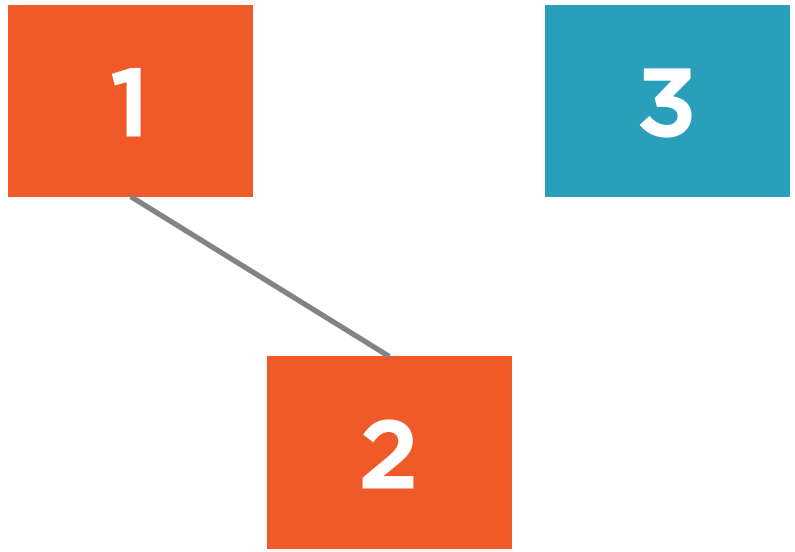
Right rotate the right child



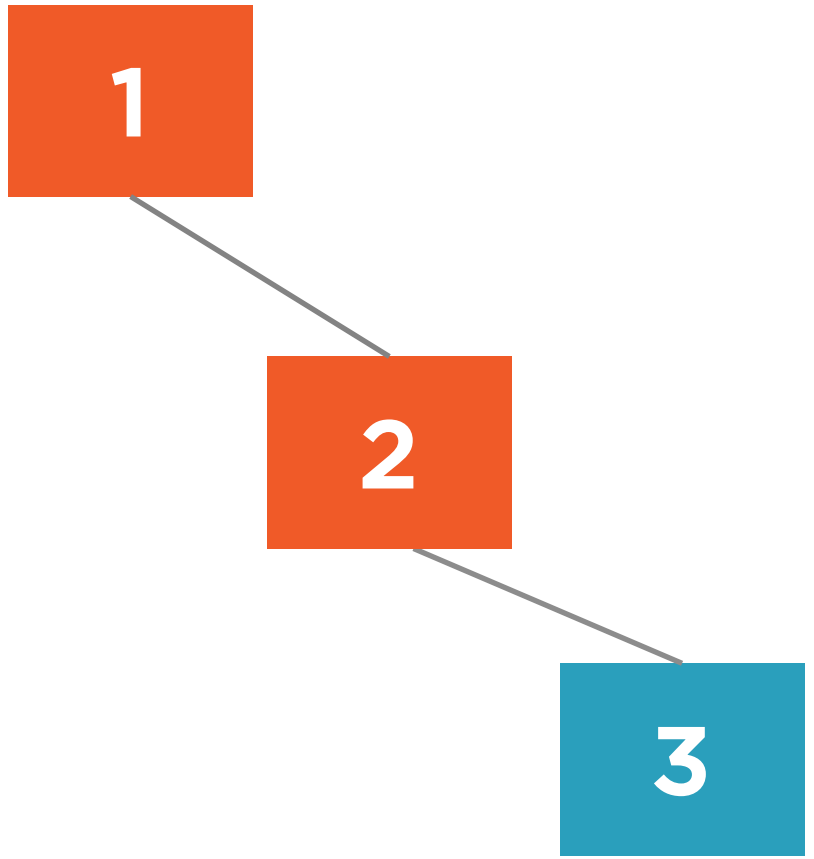
Right rotate the right child



Right rotate the right child

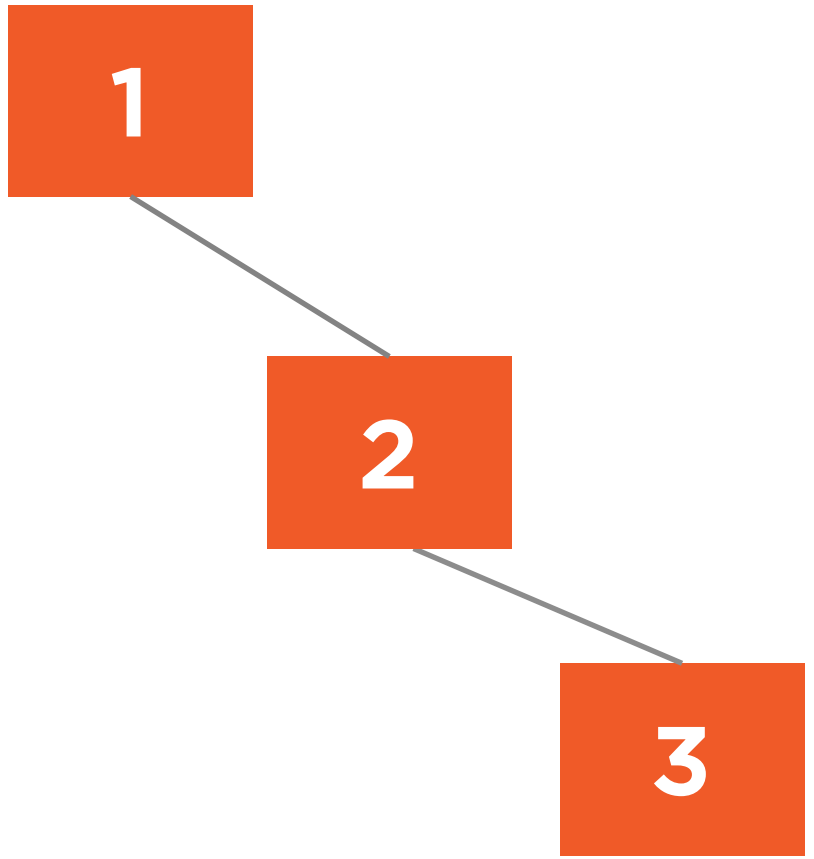


Right rotate the right child



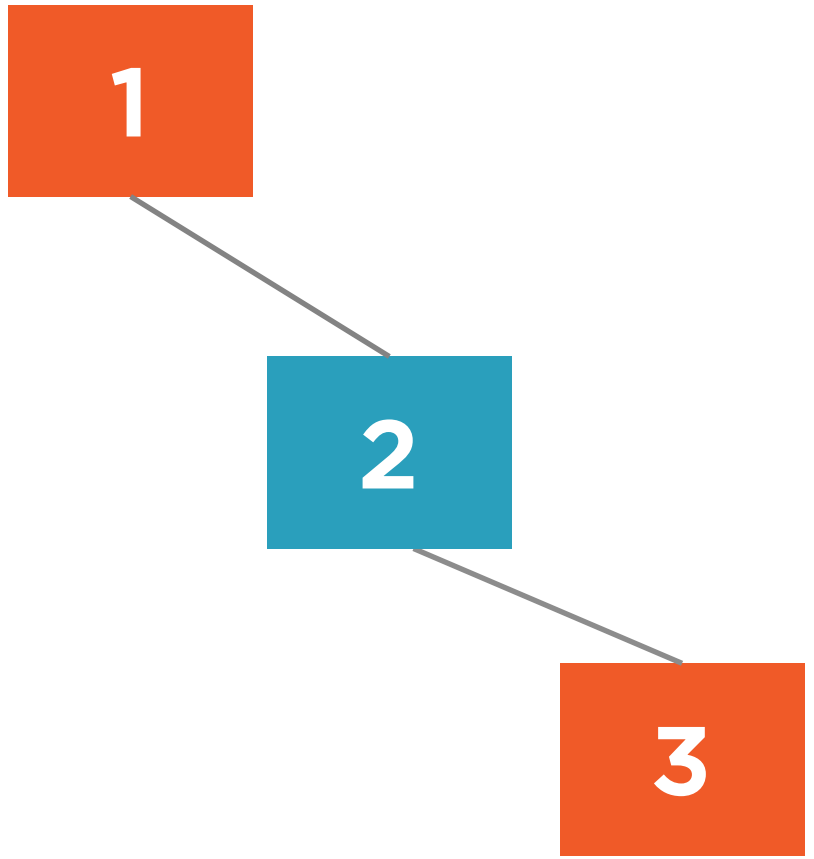
Right rotate the right child

Left rotate the root



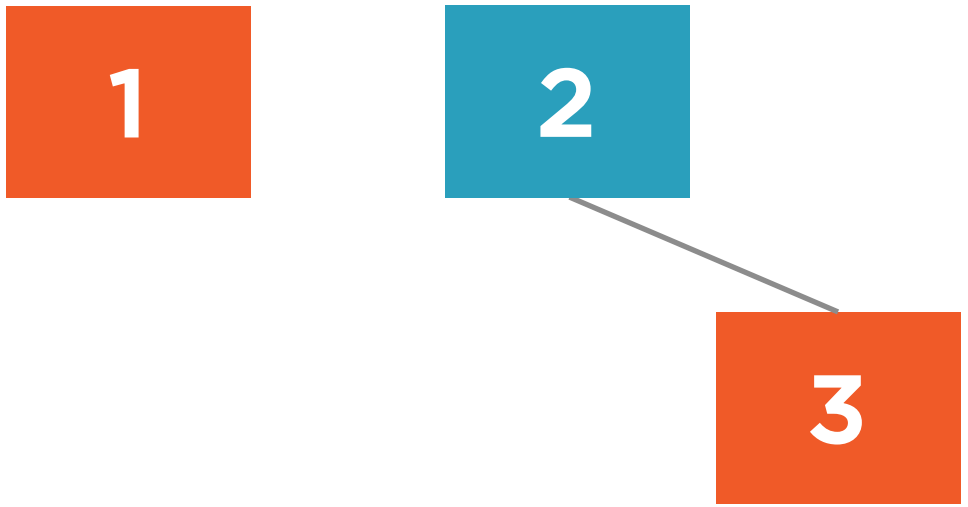
Right rotate the right child

Left rotate the root



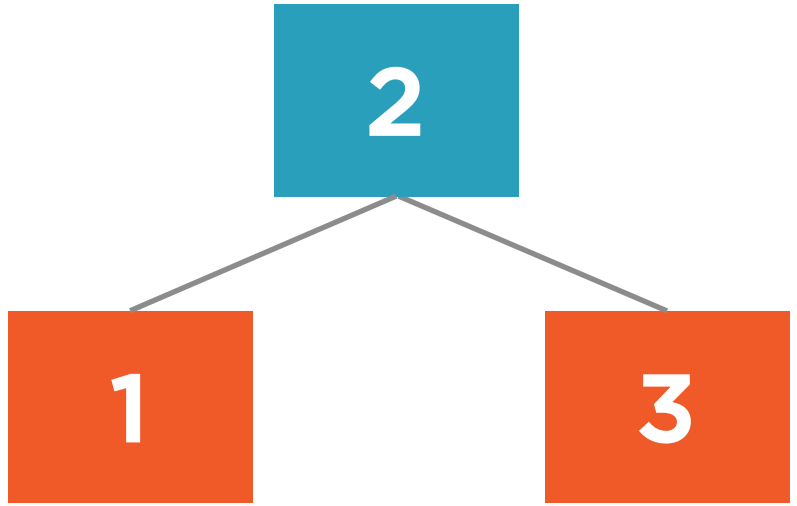
Right rotate the right child

Left rotate the root

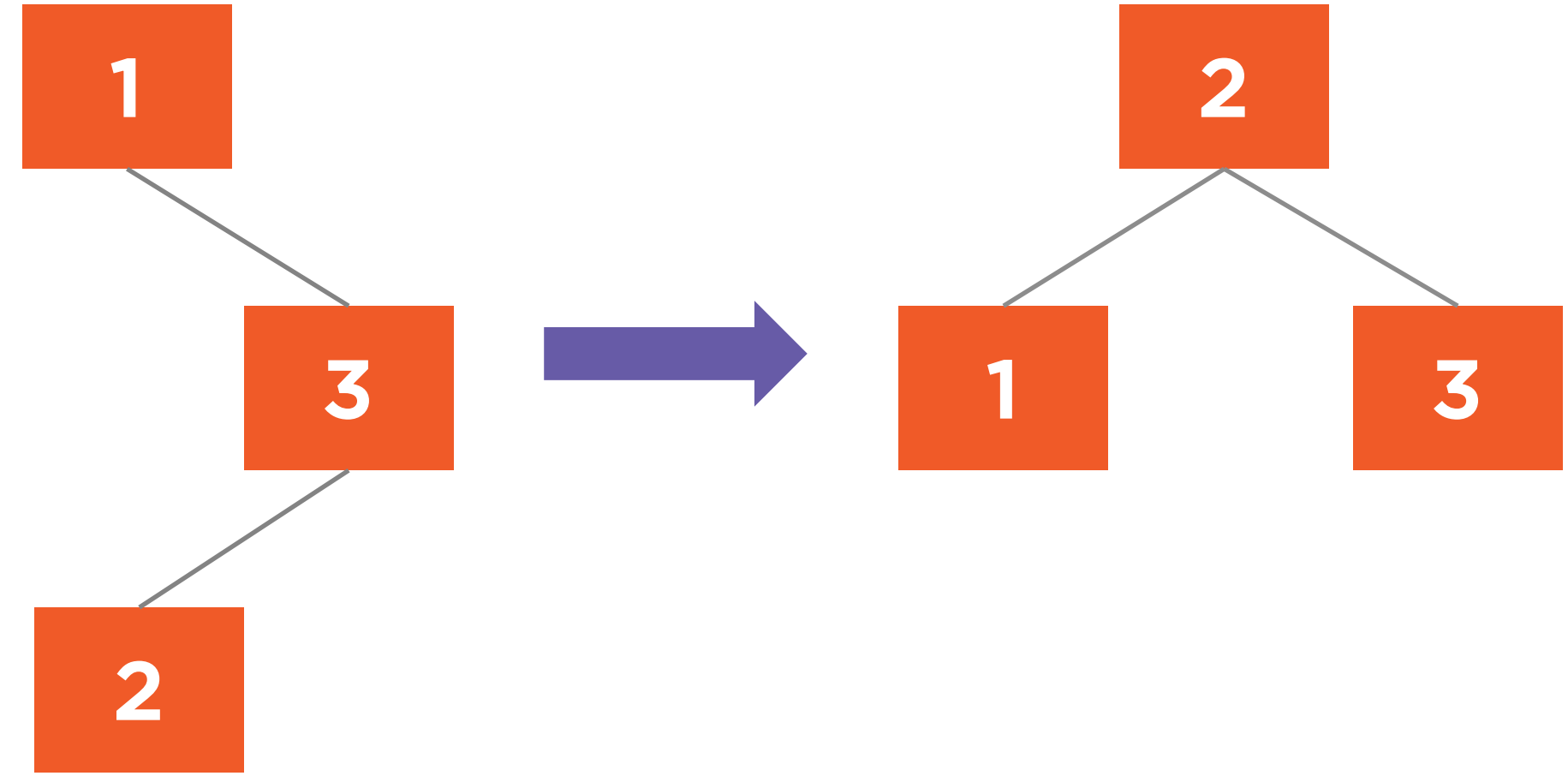


Right rotate the right child

Left rotate the root



Before and After: Left-Right Rotation



Which rotation should be used to balance the tree?

Which Rotation?

Left-heavy Tree

```
if (Left?.BalanceFactor > 0) {
    RightLeftRotation();
}
else {
    RightRotation();
}
```

Right-heavy Tree

```
if (Right?.BalanceFactor < 0) {
    LeftRightRotation();
}
else {
    LeftRotation();
}
```

Demo

Review AVL Tree code

- Class overview
- Height
- Heaviness

Balancing Operations

- Left, Right
- Left-right, Right-left