Binary Search Trees

Robert Horvick
SOFTWARE ENGINEER

@bubbafat www.roberthorvick.com

Overview

What are trees?

Binary search trees

Basic operations

- Adding data
- Removing data

Traversals

- Pre-order, post-order, in-order

Update contact manager

Tree

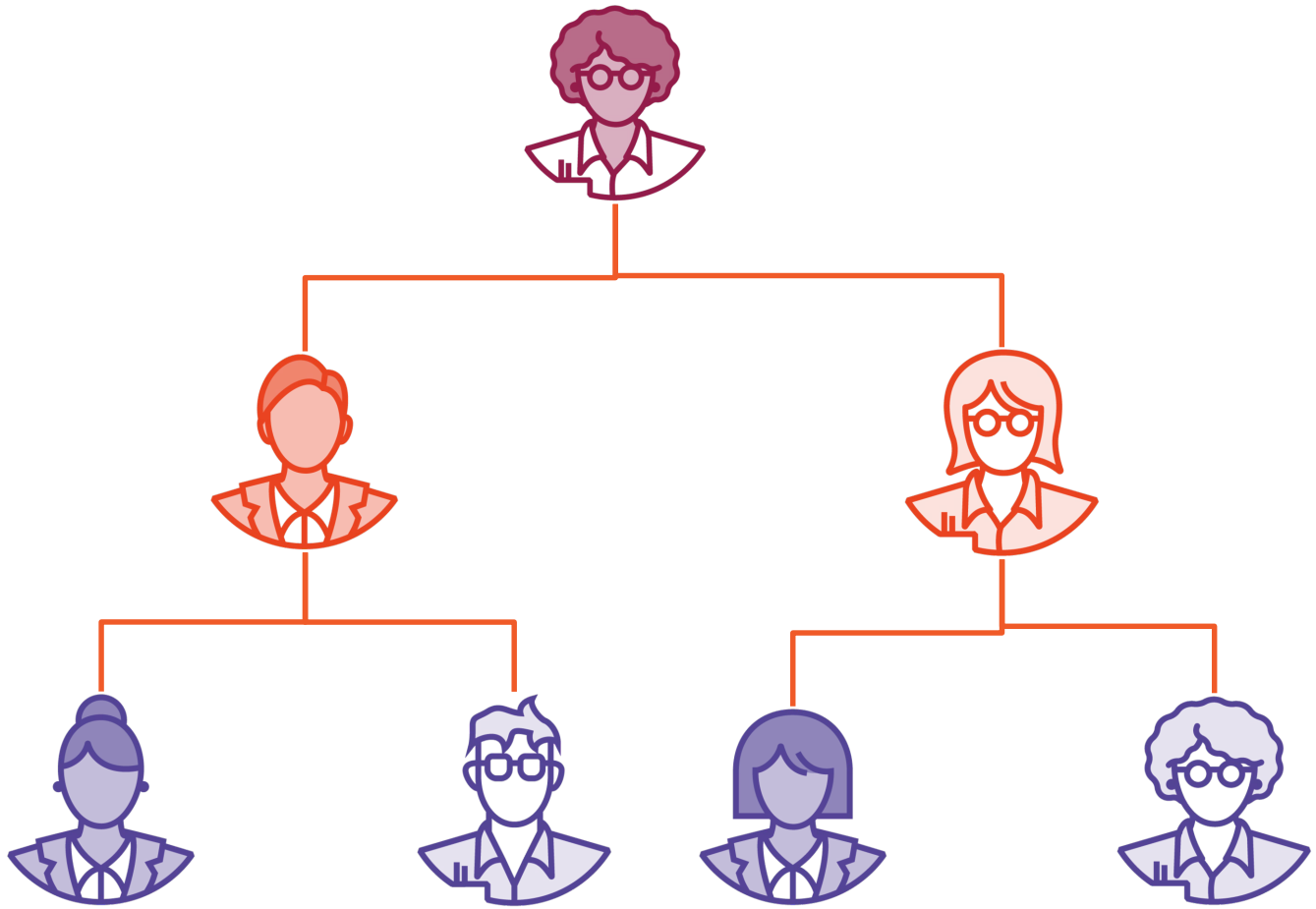
A data structure where nodes have a 1:N parent-child relationship.

Begins at the root

Branches expand out

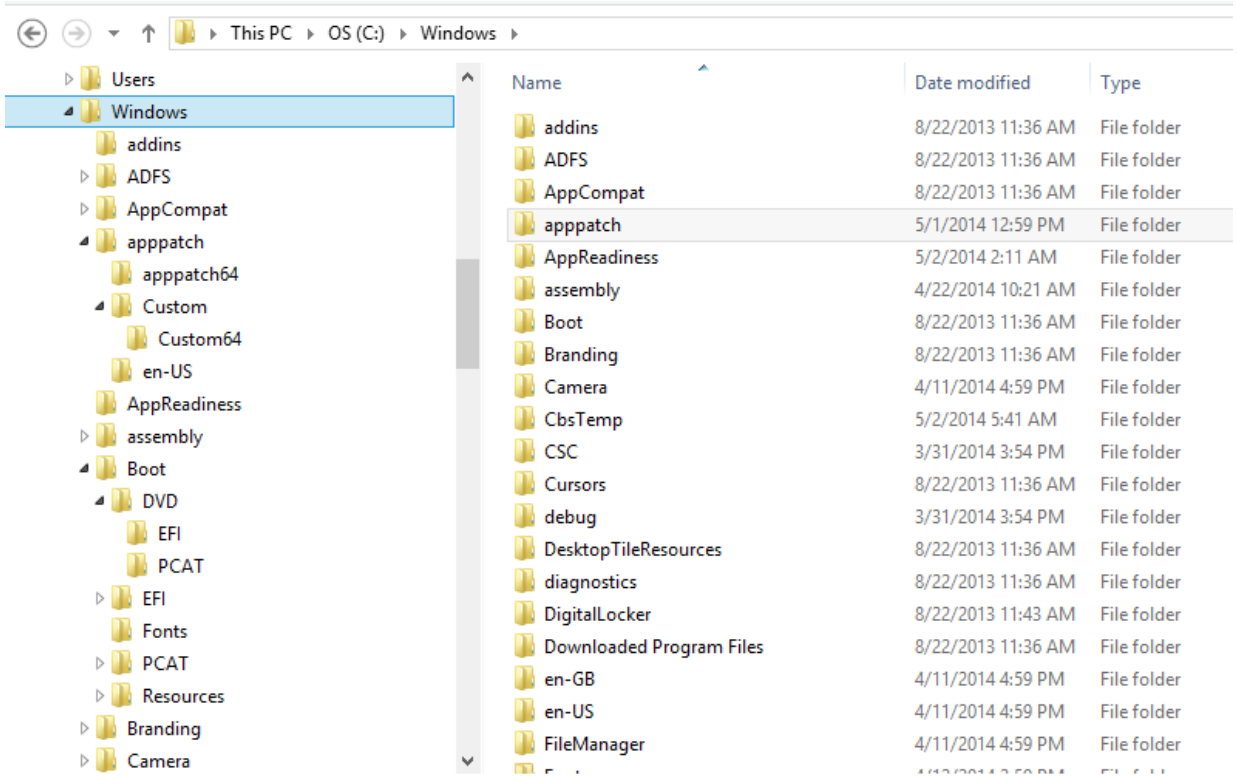
Leaves are outer boundary

Reporting structures



Reporting structures

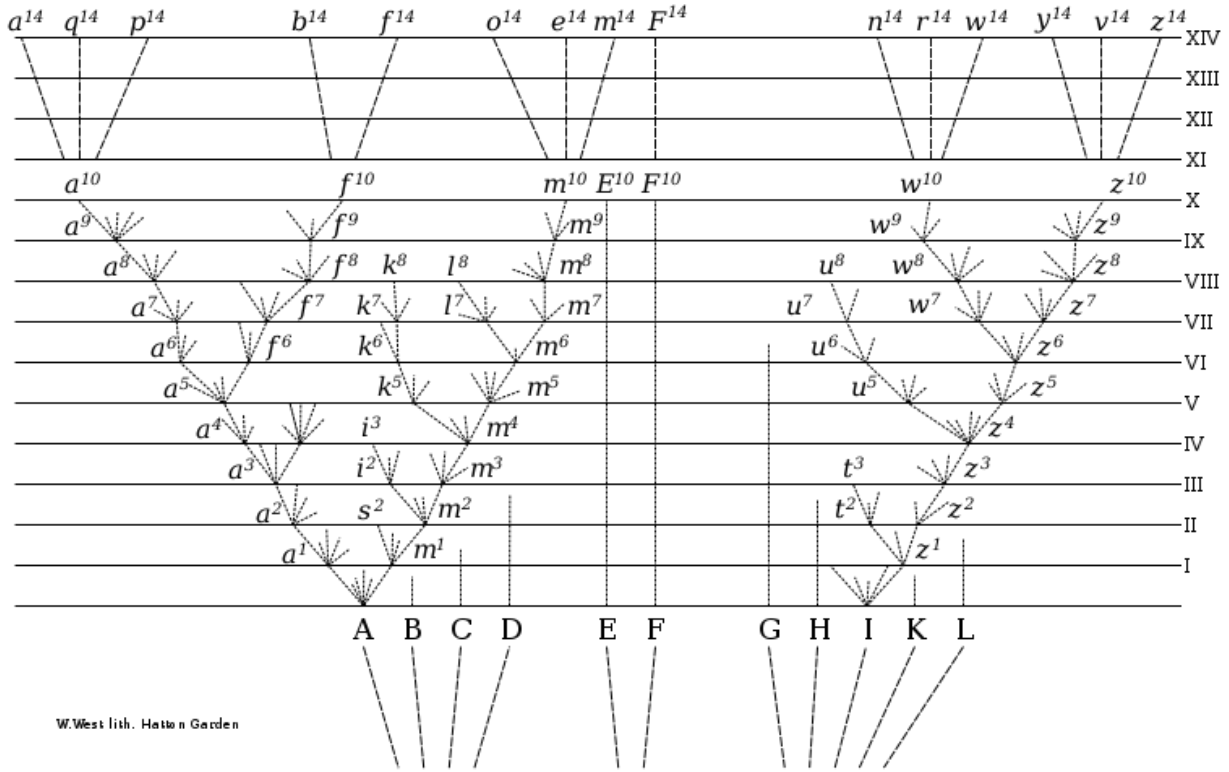
File systems



Reporting structures

File systems

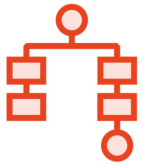
Categorization



http://commons.wikimedia.org/wiki/File:Origin_of_Species.svg

Properties of Trees

0 or 1 parent node



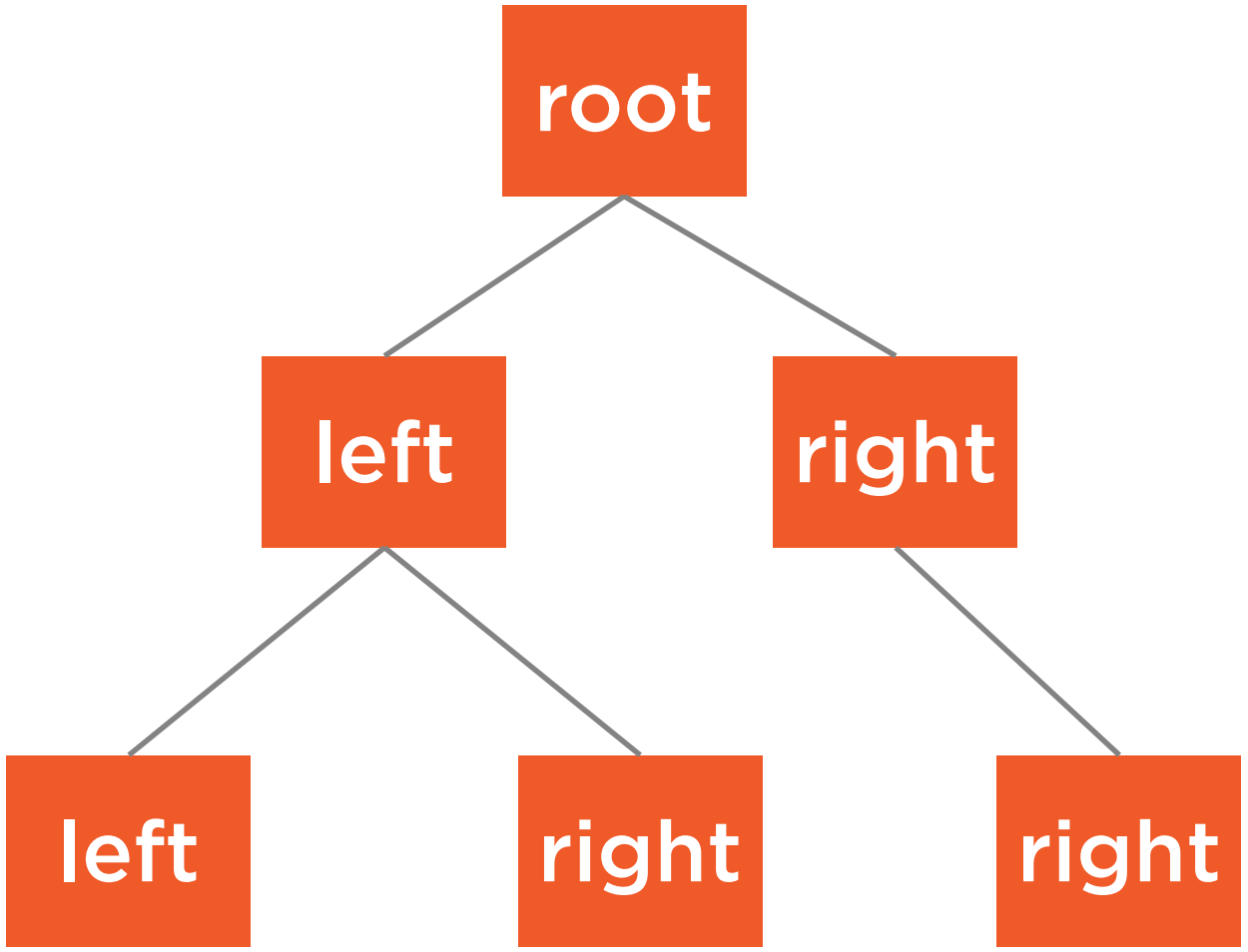
0-N child nodes (binary, trinary, k-ary)

Leaf nodes have no children

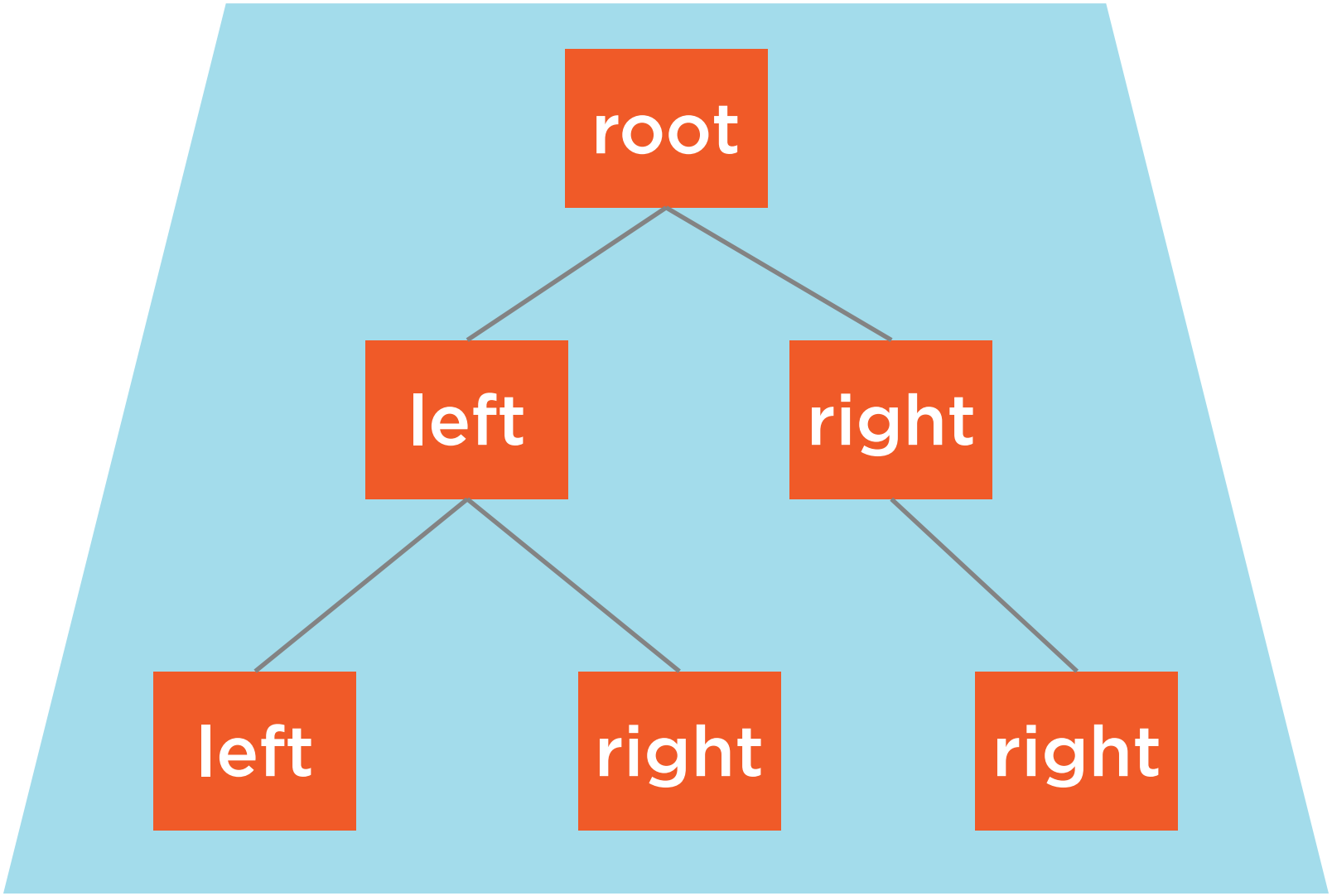
One data item per node

Binary Trees

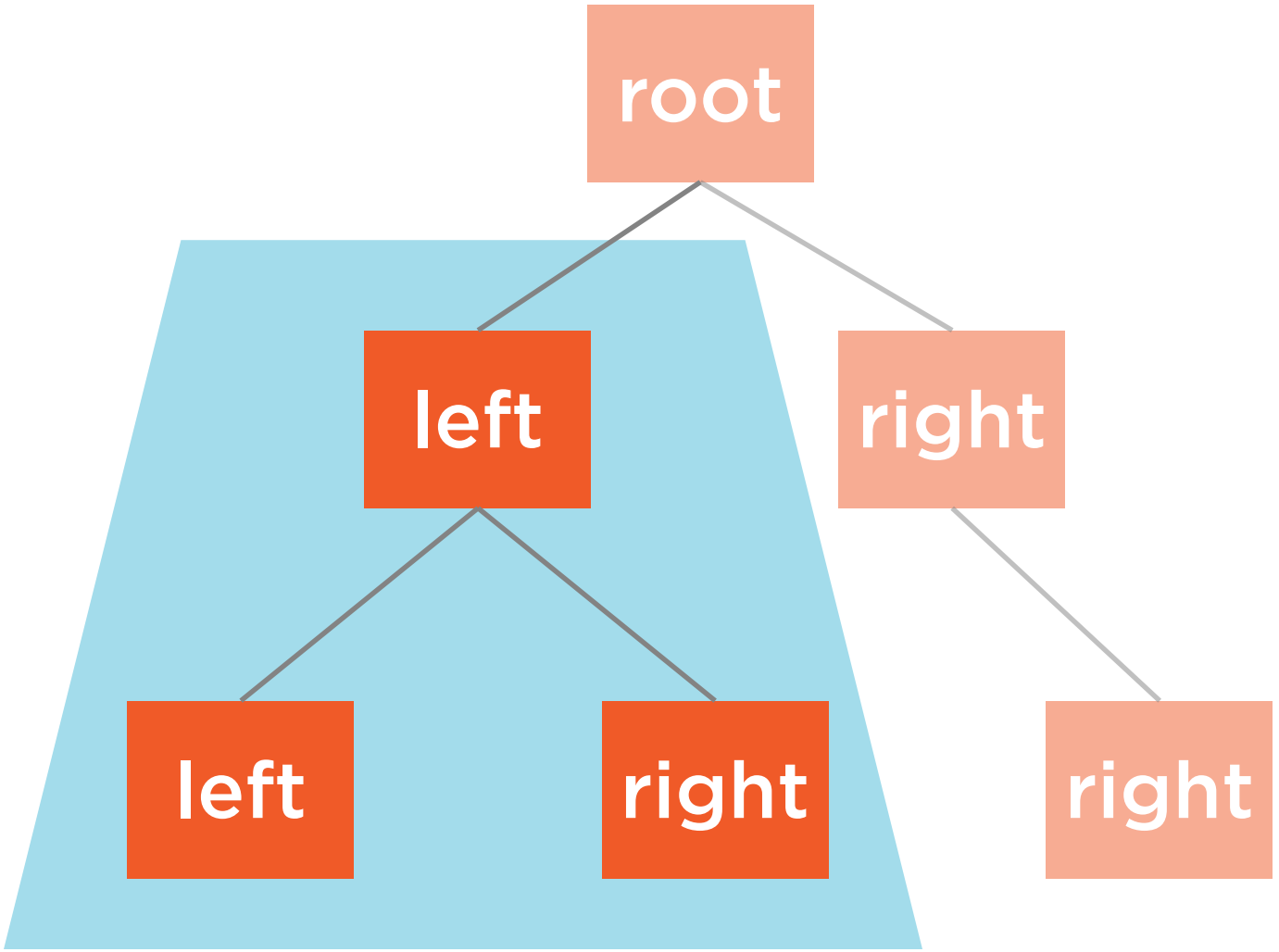
Binary Trees



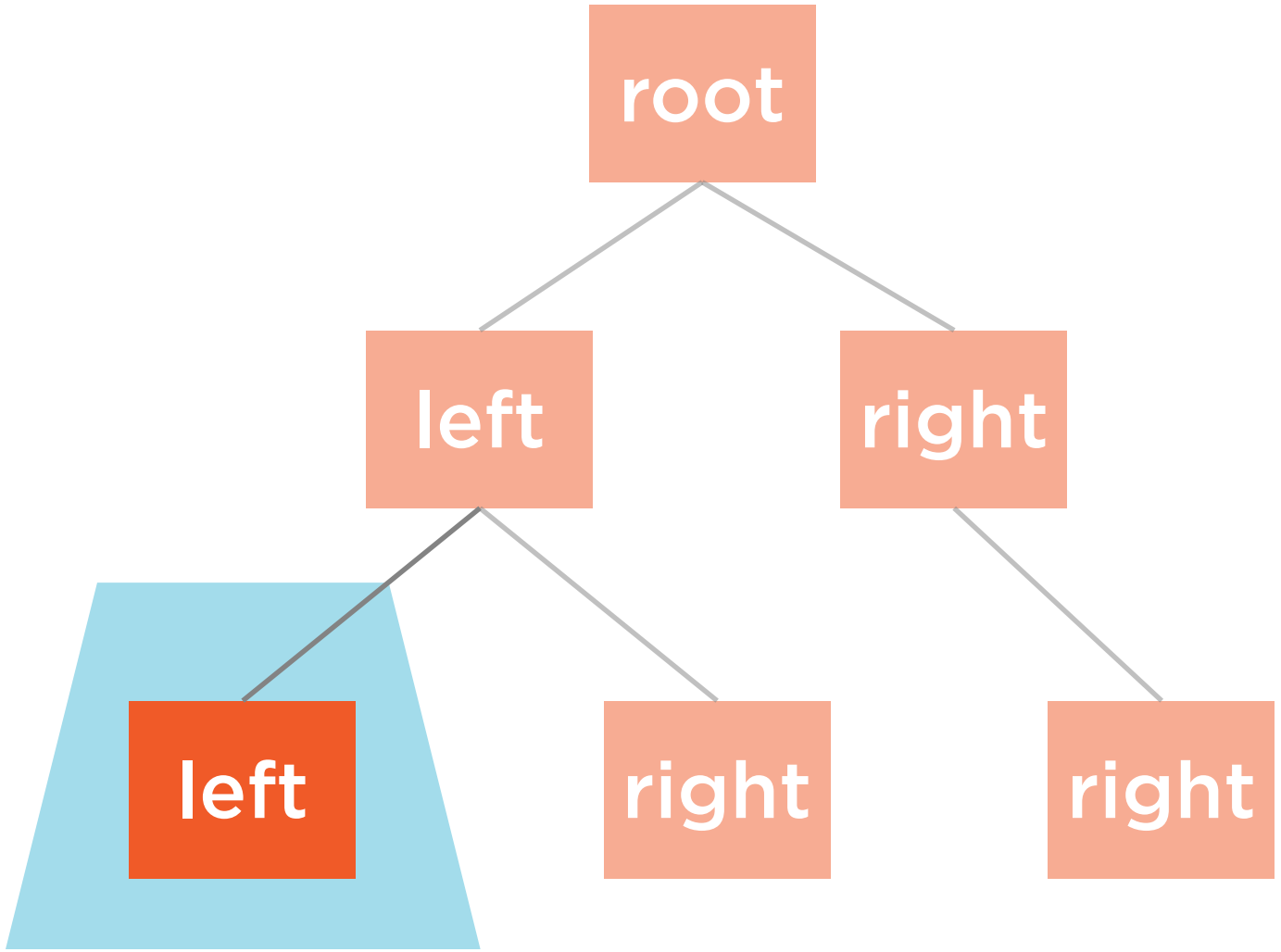
Binary Trees



Binary Trees



Binary Trees



Root node

Parents and children

Edges connect nodes

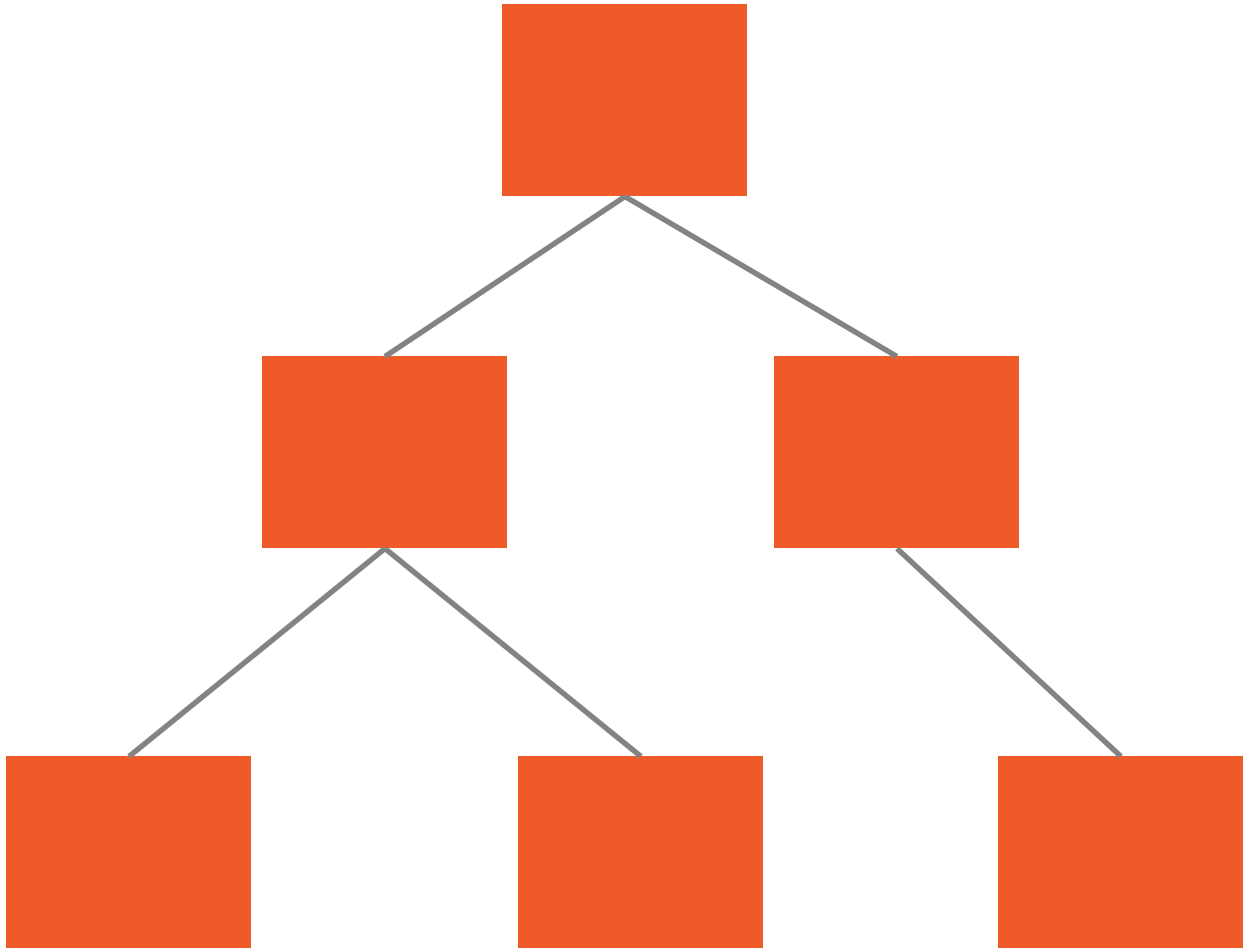
Leaf nodes have no children

Internal nodes

Degree is max children

Height counts edges

Level counts the edges to root



Binary Tree Nodes per Level

Level	Max Nodes on Level	Max Total Nodes
1	1	1
2	2	3
3	4	7

Binary Tree Nodes per Level

Level	Max Nodes on Level	Max Total Nodes
1	1	1
2	2	3
3	4	7
4	8	15
5	16	31
6	32	63
7	64	127
8	128	255

Binary Tree Nodes per Level

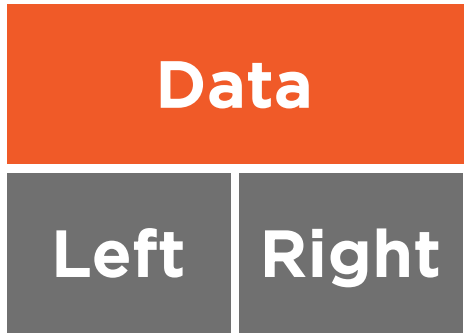
Level	Max Nodes on Level	Max Total Nodes
1	1	1
2	2	3
3	4	7
4	8	15
5	16	31
6	32	63
7	64	127
8	128	255
16	65536	131071
24	16777216	33554431
32	4294967296	8589934591

Binary Tree Node

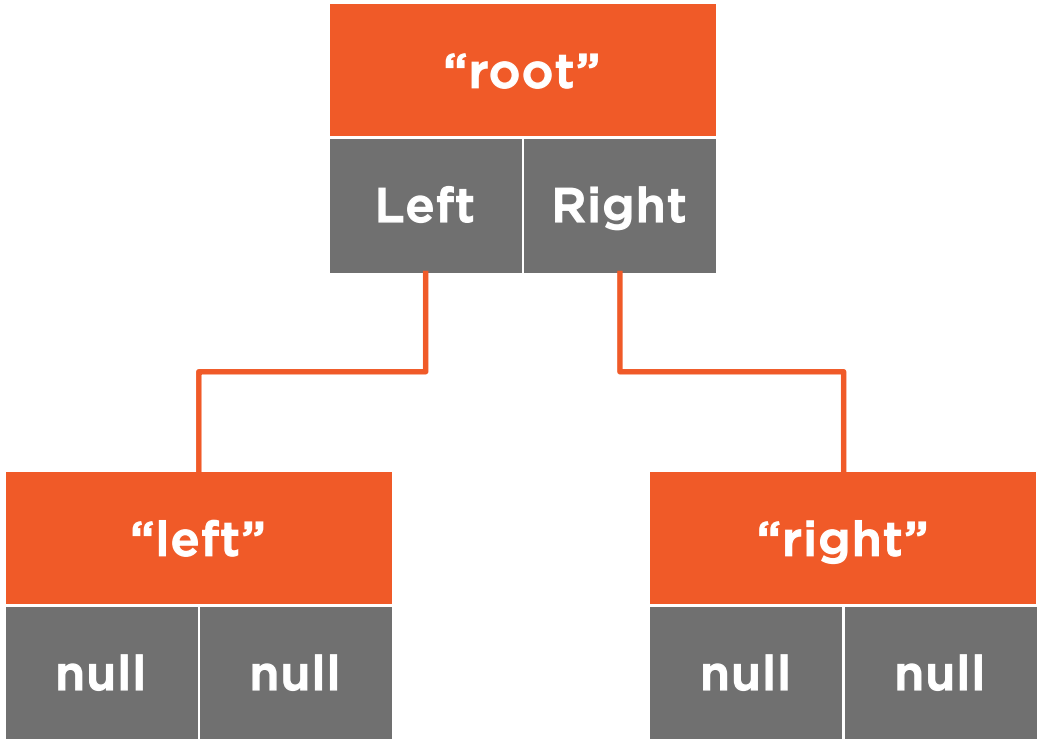
A node which contains a single data item and pointers for the left and right child.

```
class BSTNode<T> {

    public BSTNode(T value) {
        Data = value;
    }

    public T Data;
    public BSTNode<T> Left;
    public BSTNode<T> Right;
}
```

Binary Tree Node



```
BSTNode<string> root = new     BSTNode<string>("root");

BSTNode<string> left = new     BSTNode<string>("left");

BSTNode<string> right = new
        BSTNode<string>("right");


root.Left = left;

root.Right = right;
```



Binary Search Tree

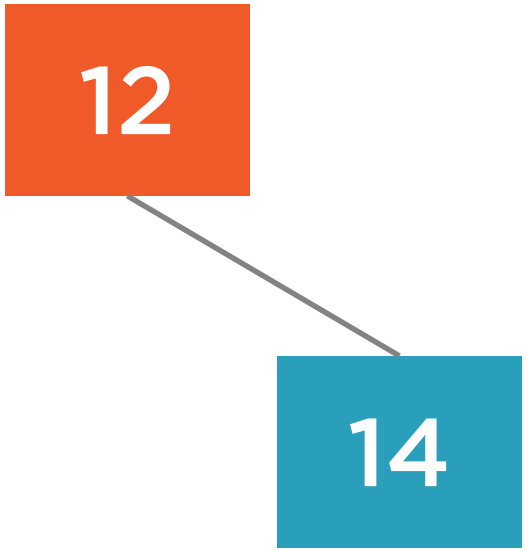
A binary tree where nodes with lessor values are placed to the left of the root, and nodes with equal or greater values are placed to the right.

12

12

12

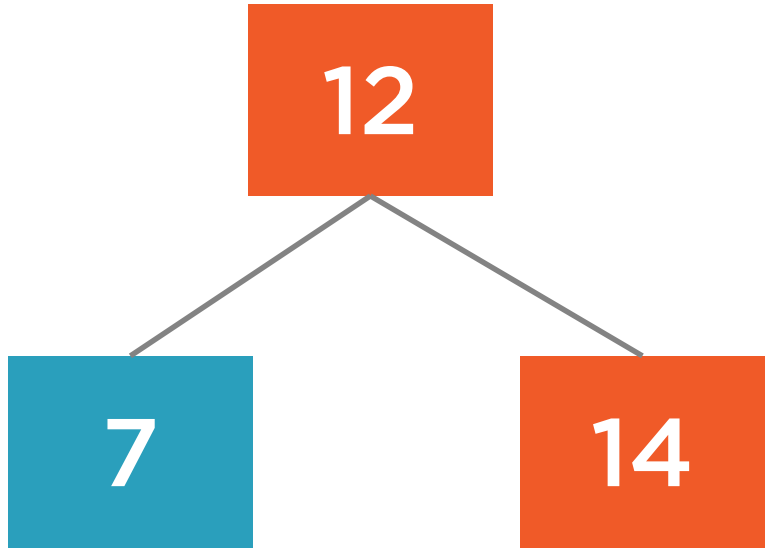
14



12

14

7

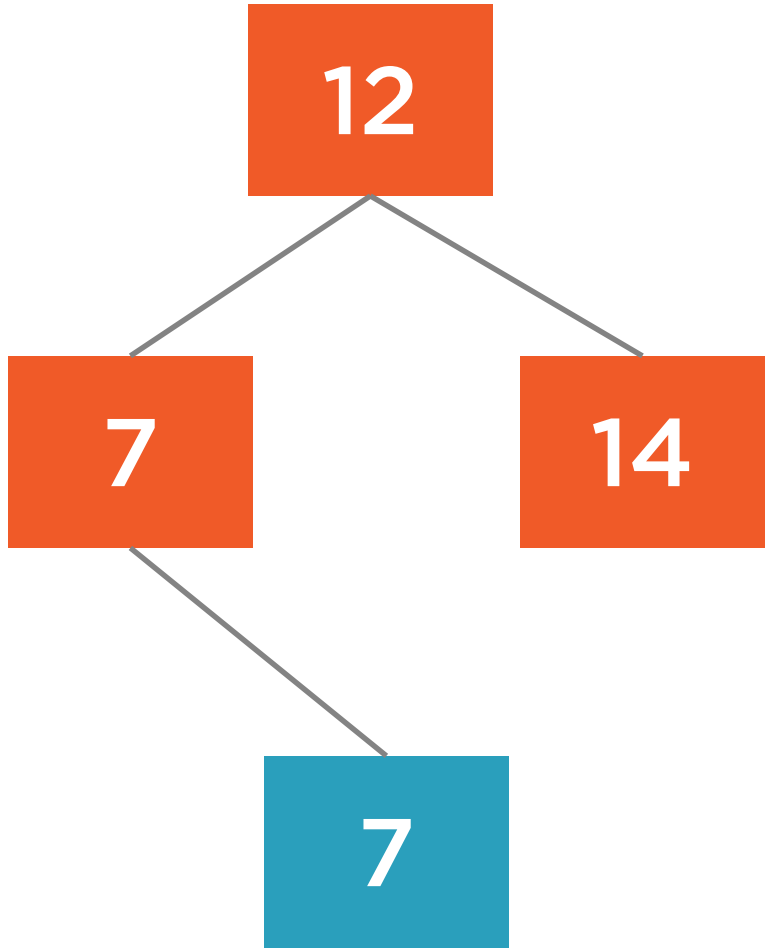


12

14

7

7



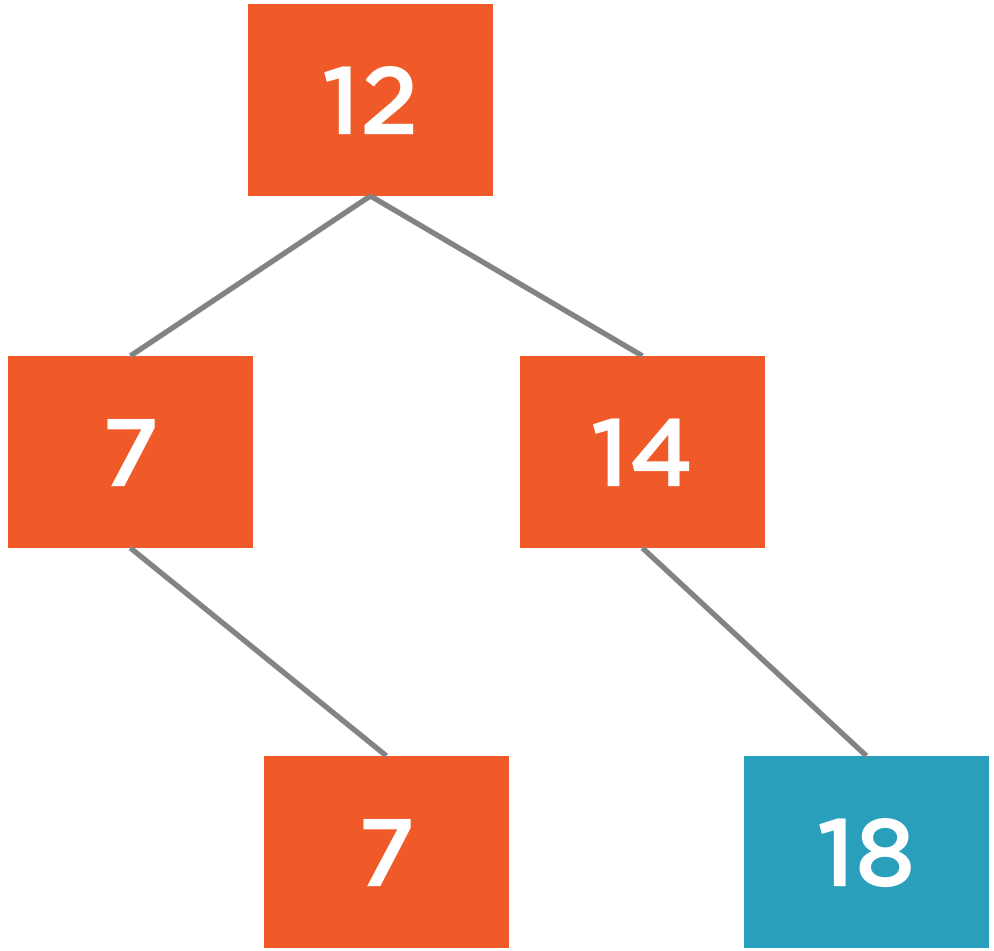
12

14

7

7

18



12

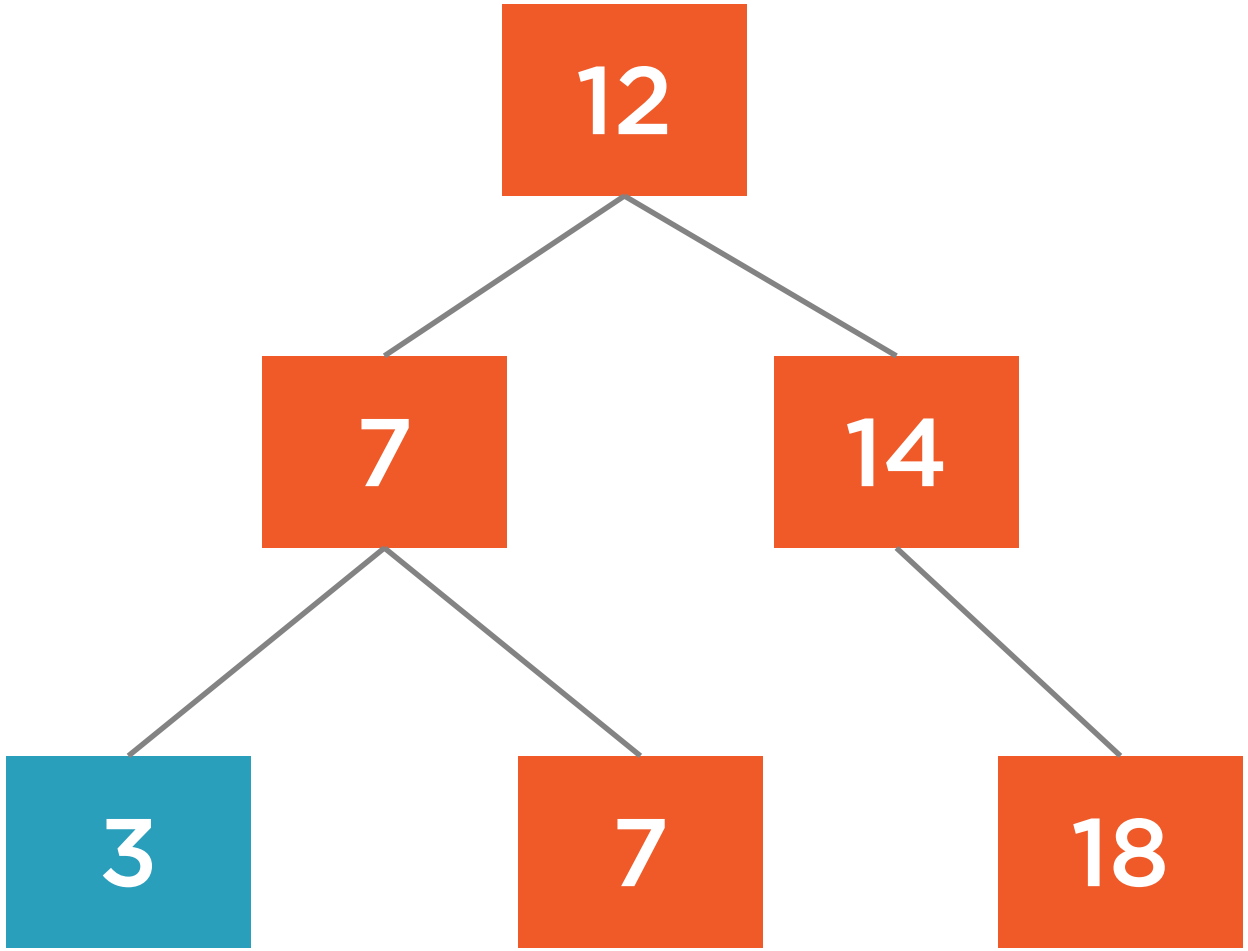
14

7

7

18

3



12

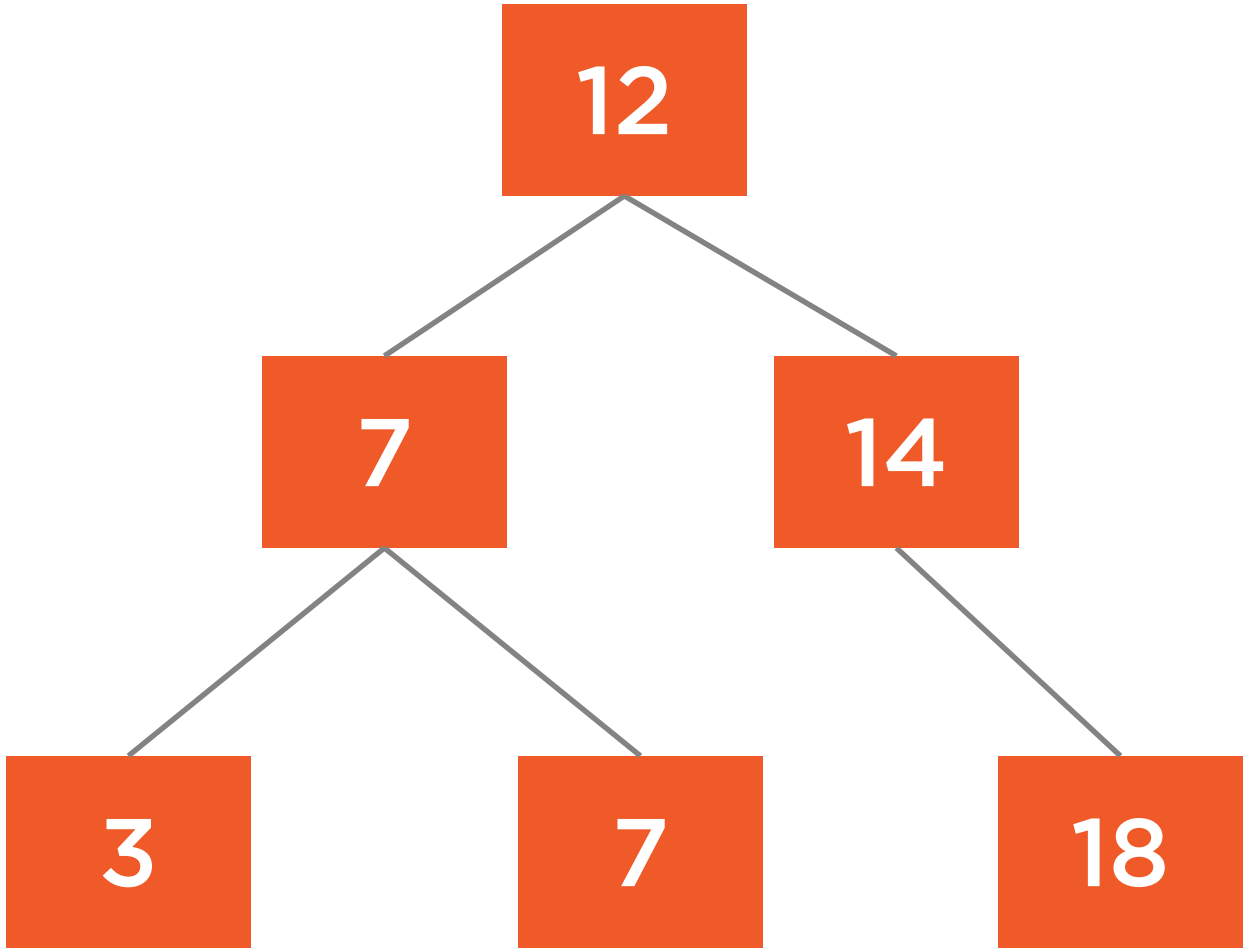
14

7

7

18

3



12

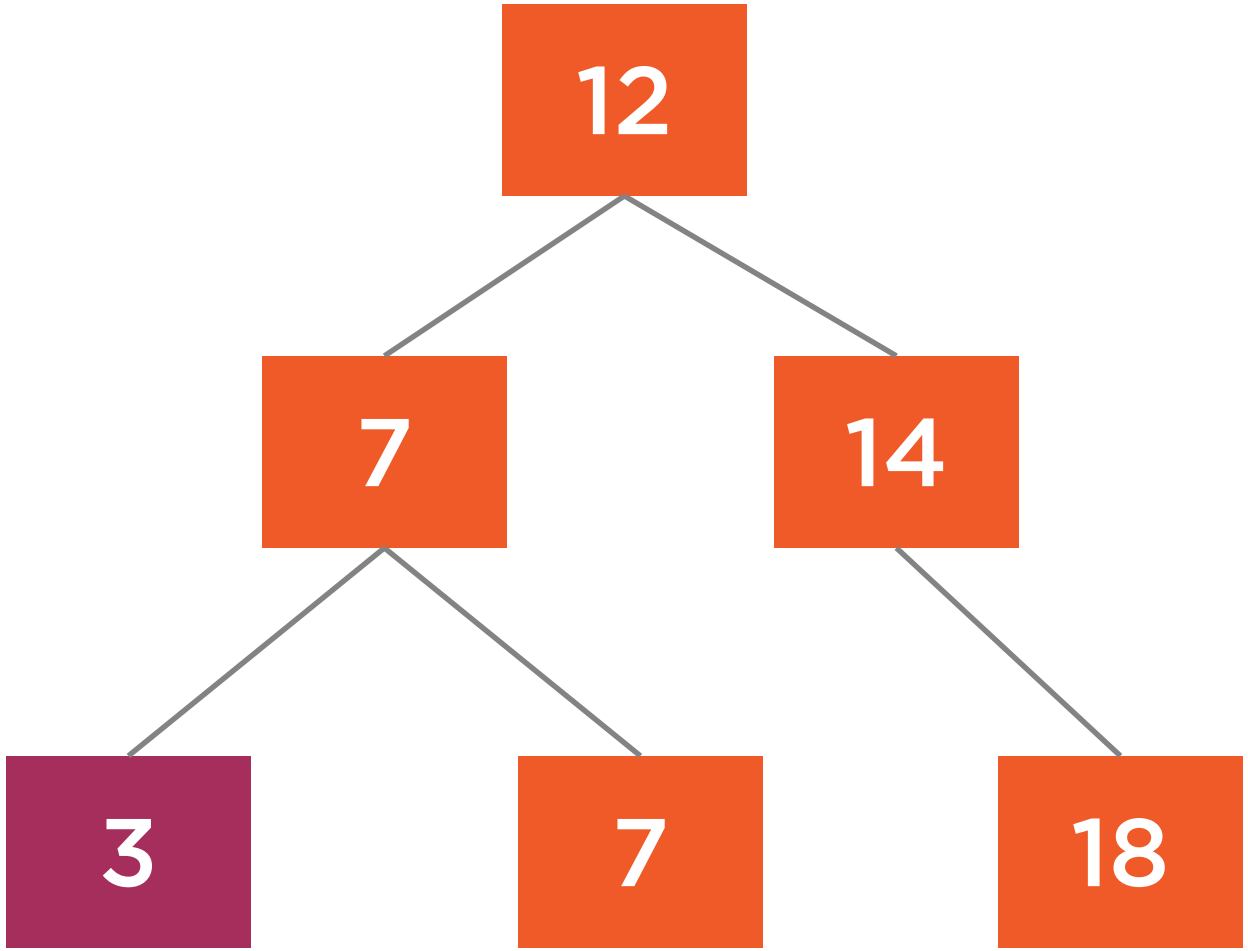
14

7

7

18

3



12

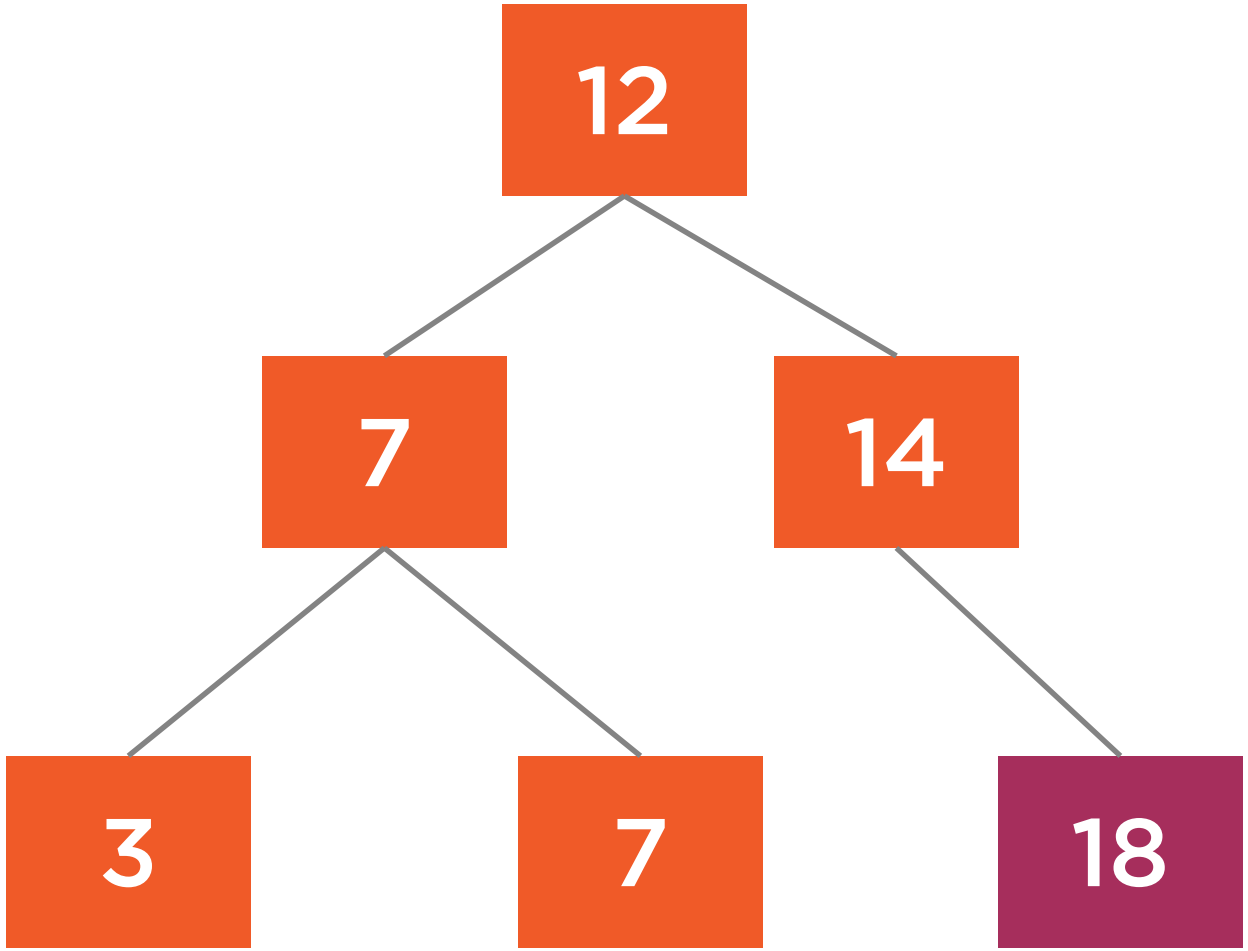
14

7

7

18

3



12

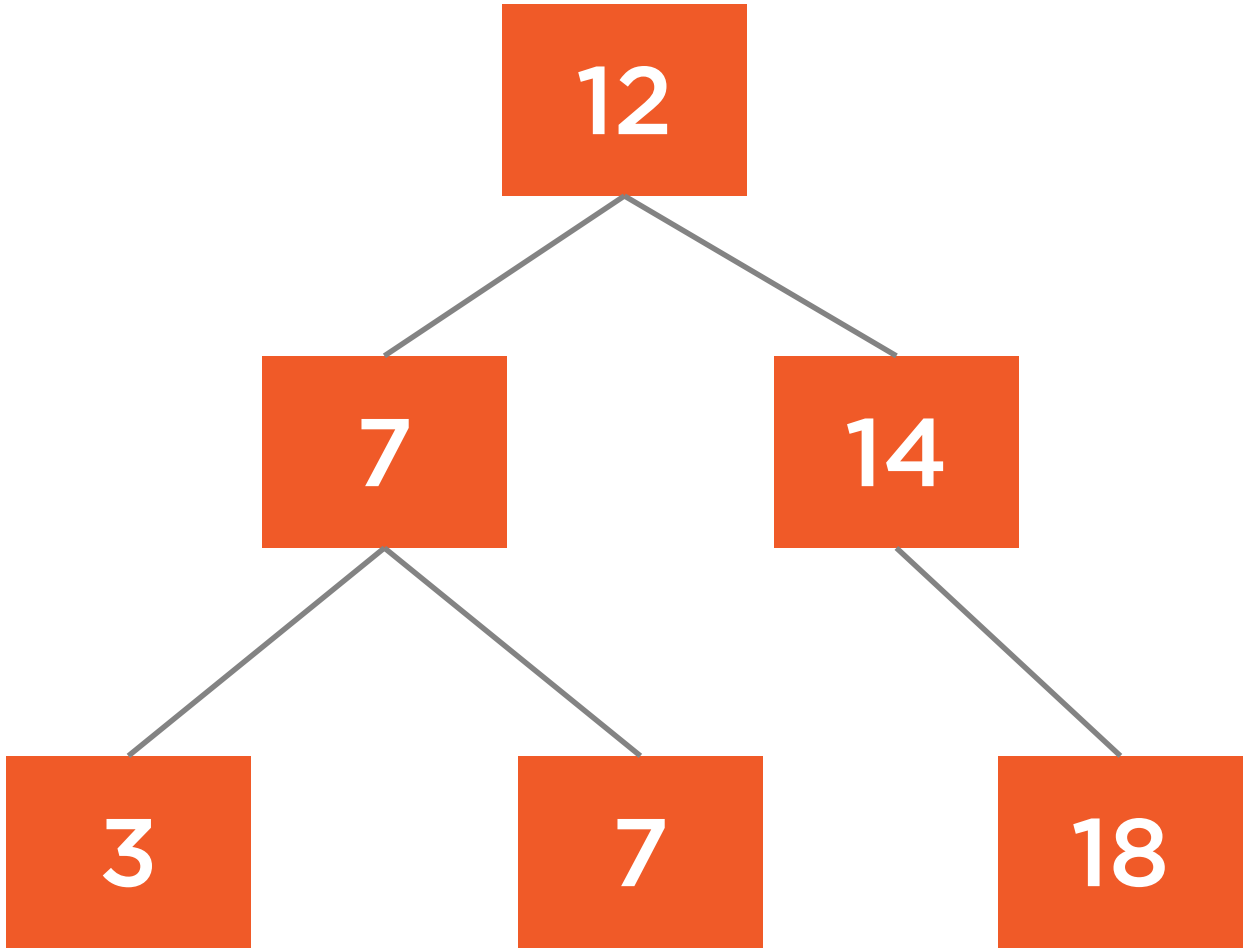
14

7

7

18

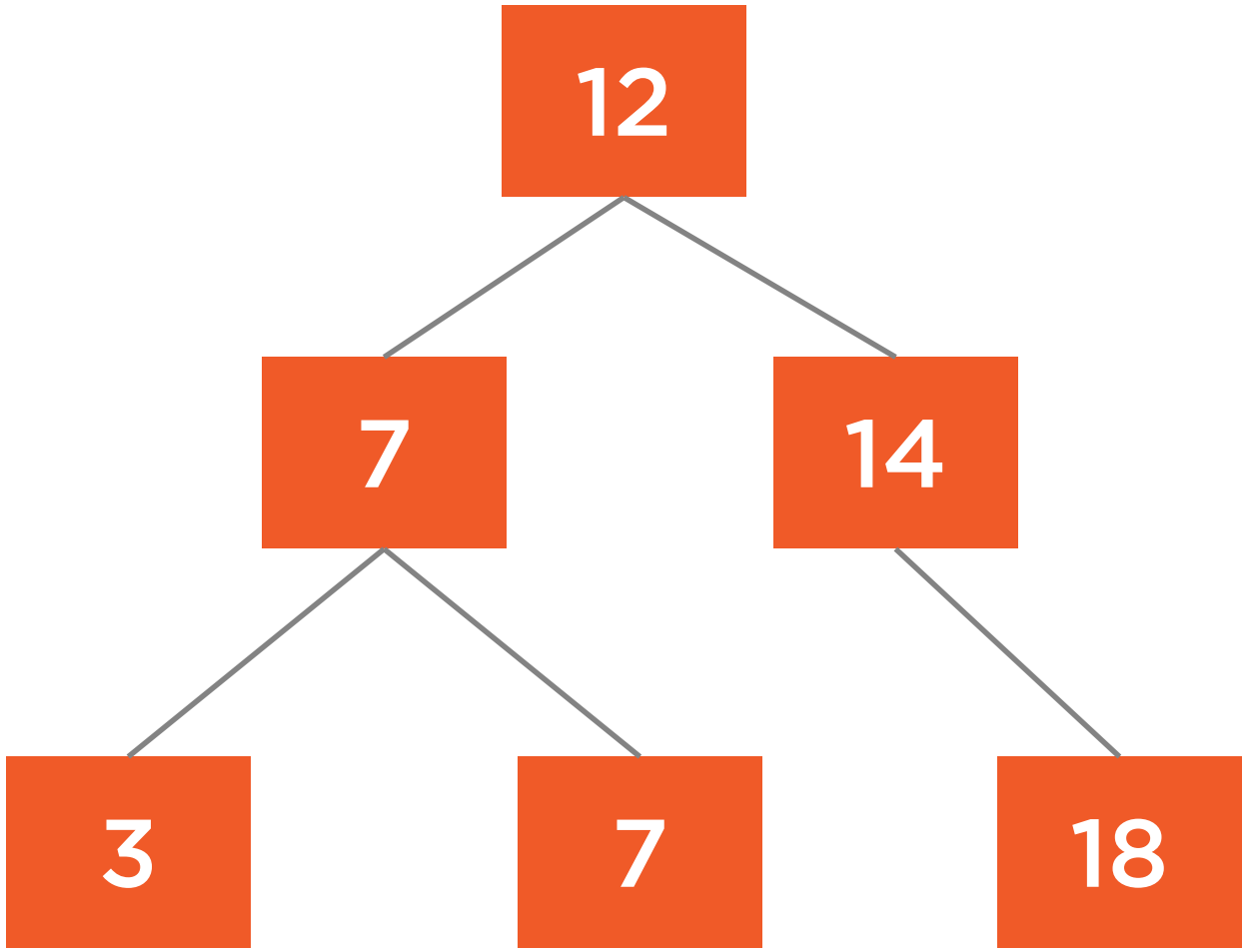
3



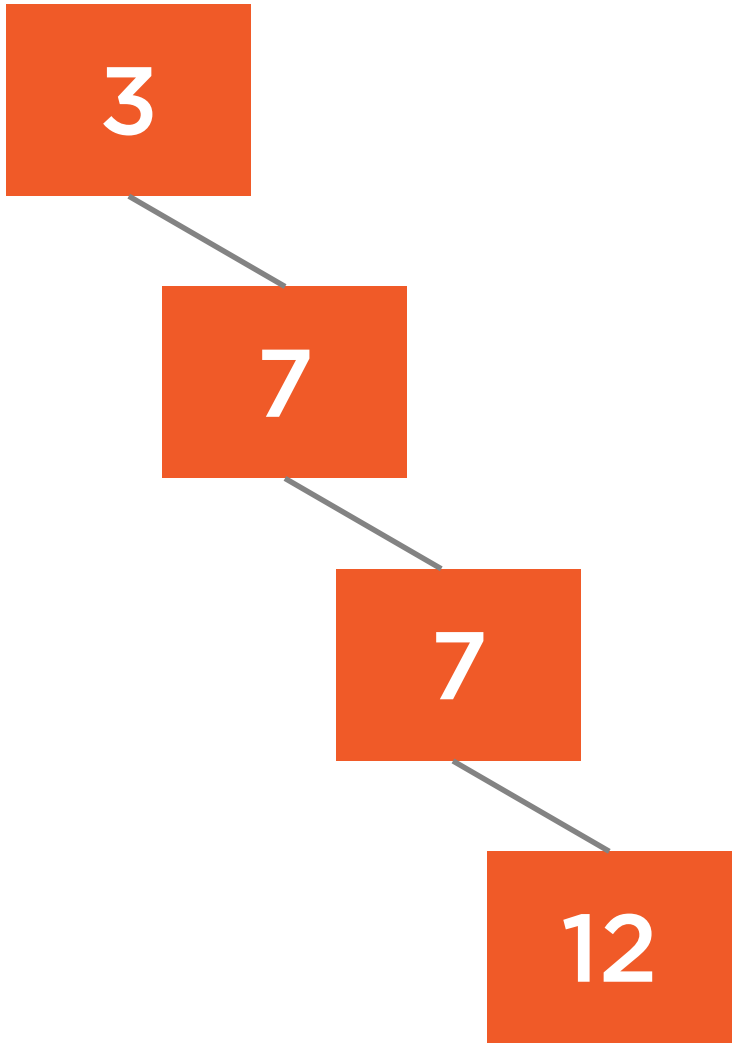
Insertion Complexity

Average case	Worst case
O(log n)	O(n)

Balanced Tree



Unbalanced Tree



Traversals

Traversal Operations

Visit the left child

Visit the right child

Process the current value

Tree Traversals

Pre-order

The node is visited before it's children

In-order

The left child is visited before the node, then the right child

Post-order

The left and right children are visited before the node

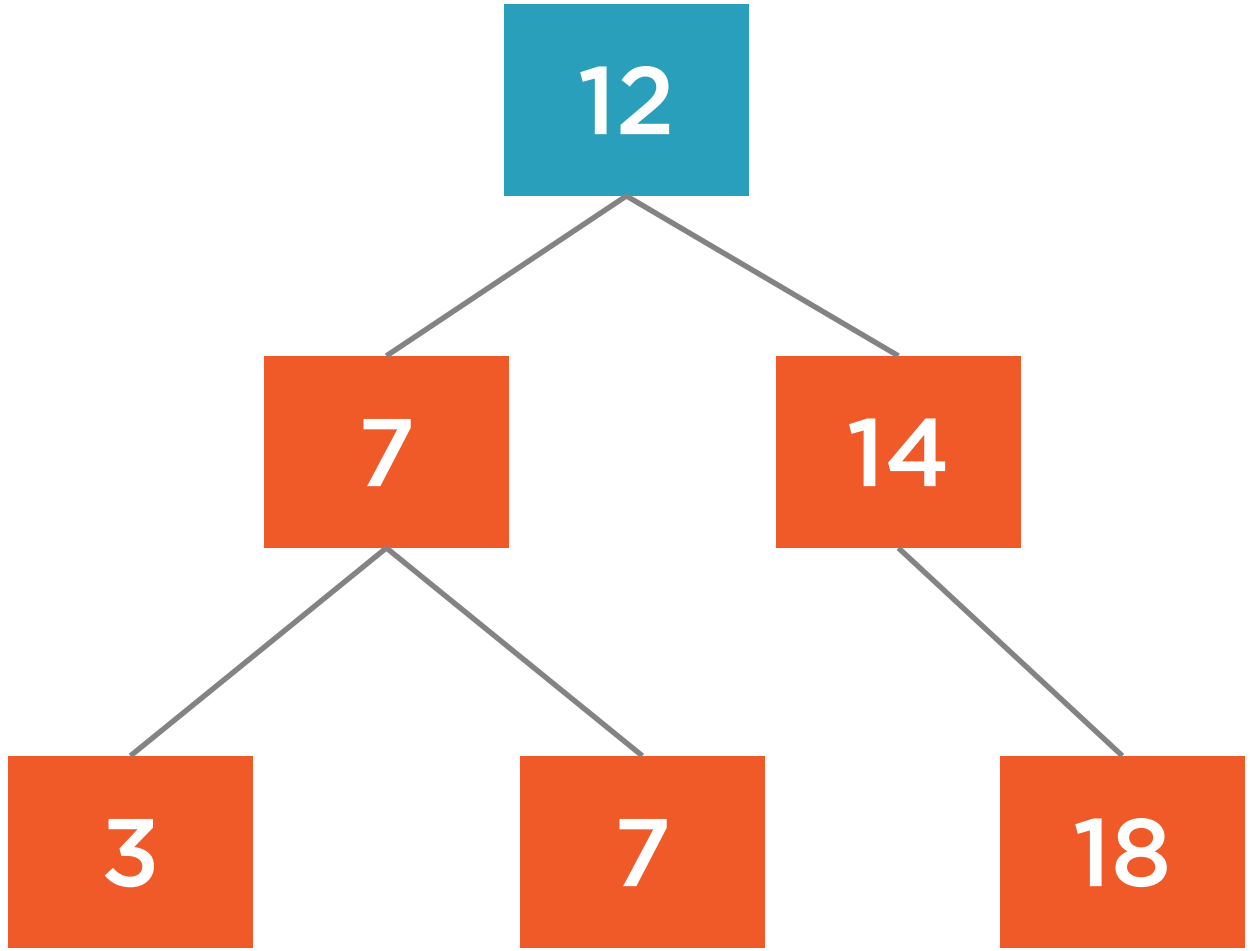
Pre-order Traversal

Process the current value

Visit the left child

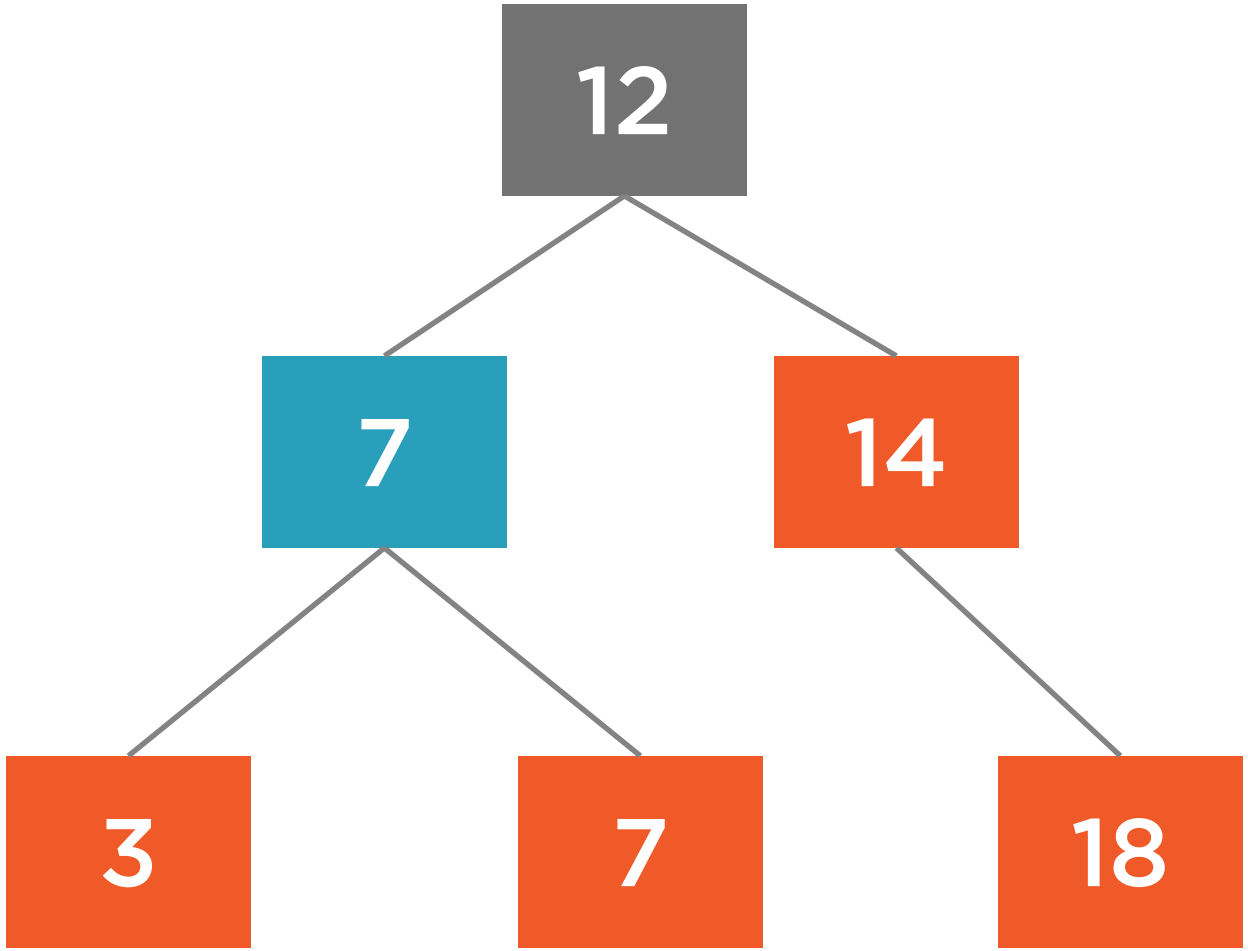
Visit the right child

12



12

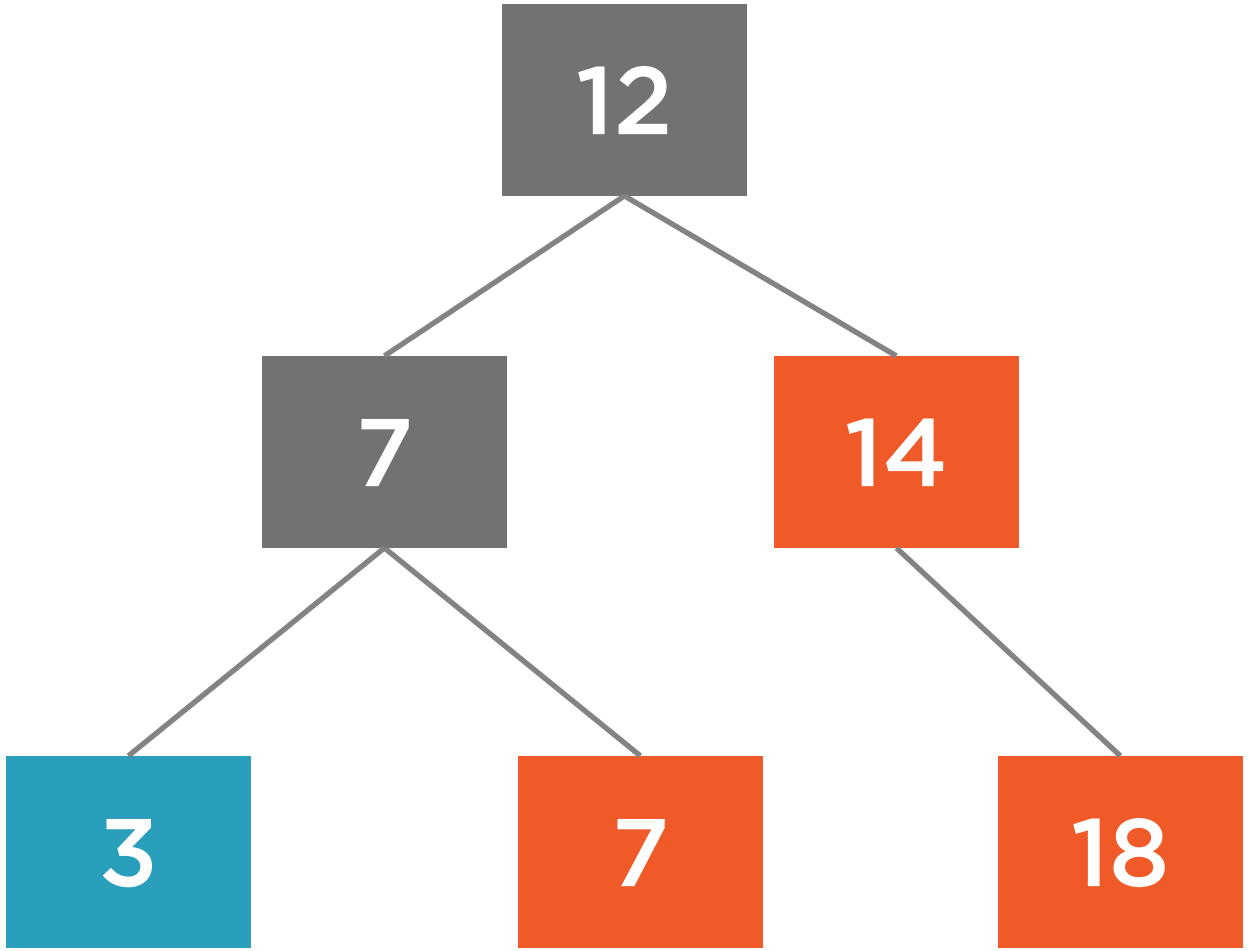
7



12

7

3

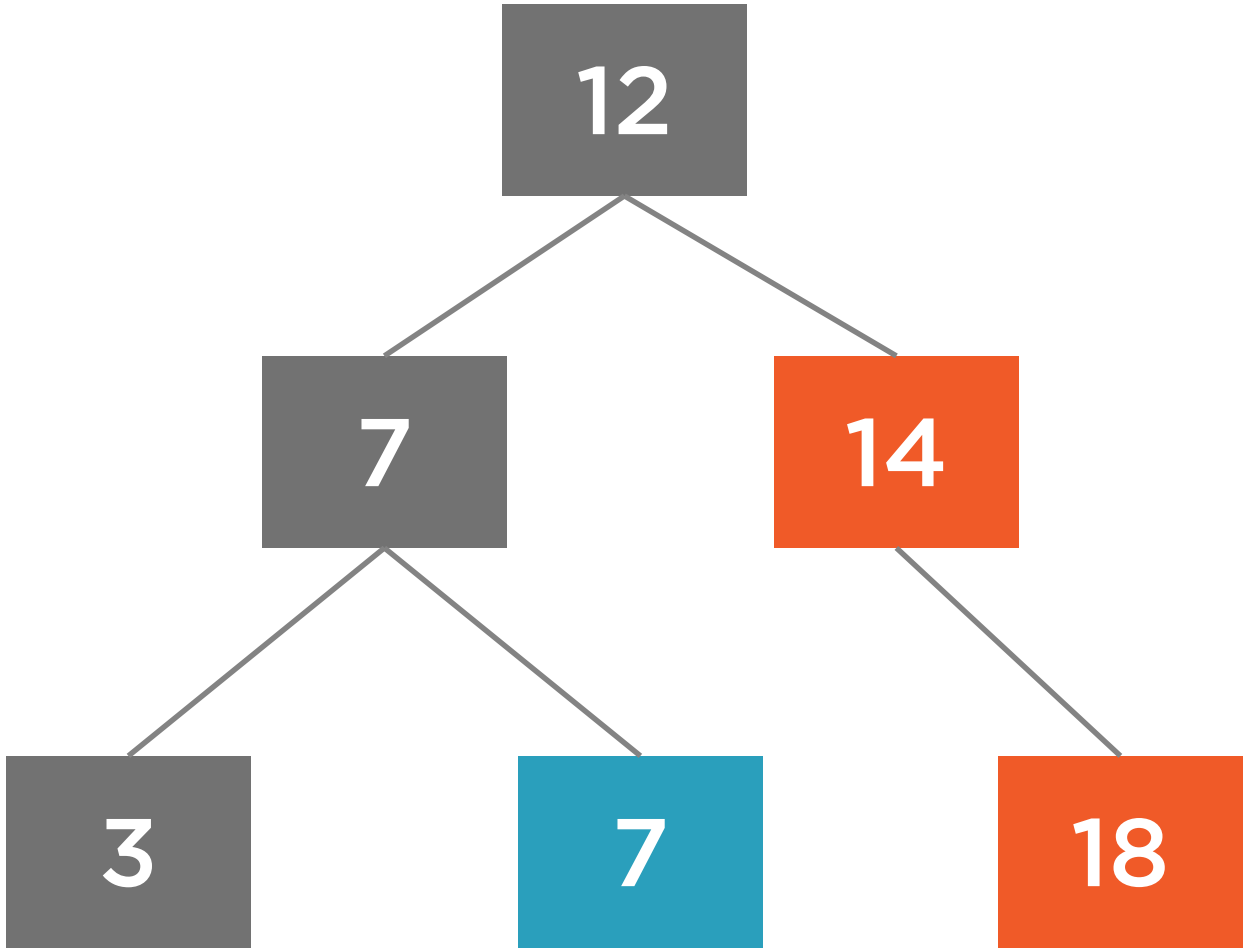


12

7

3

7



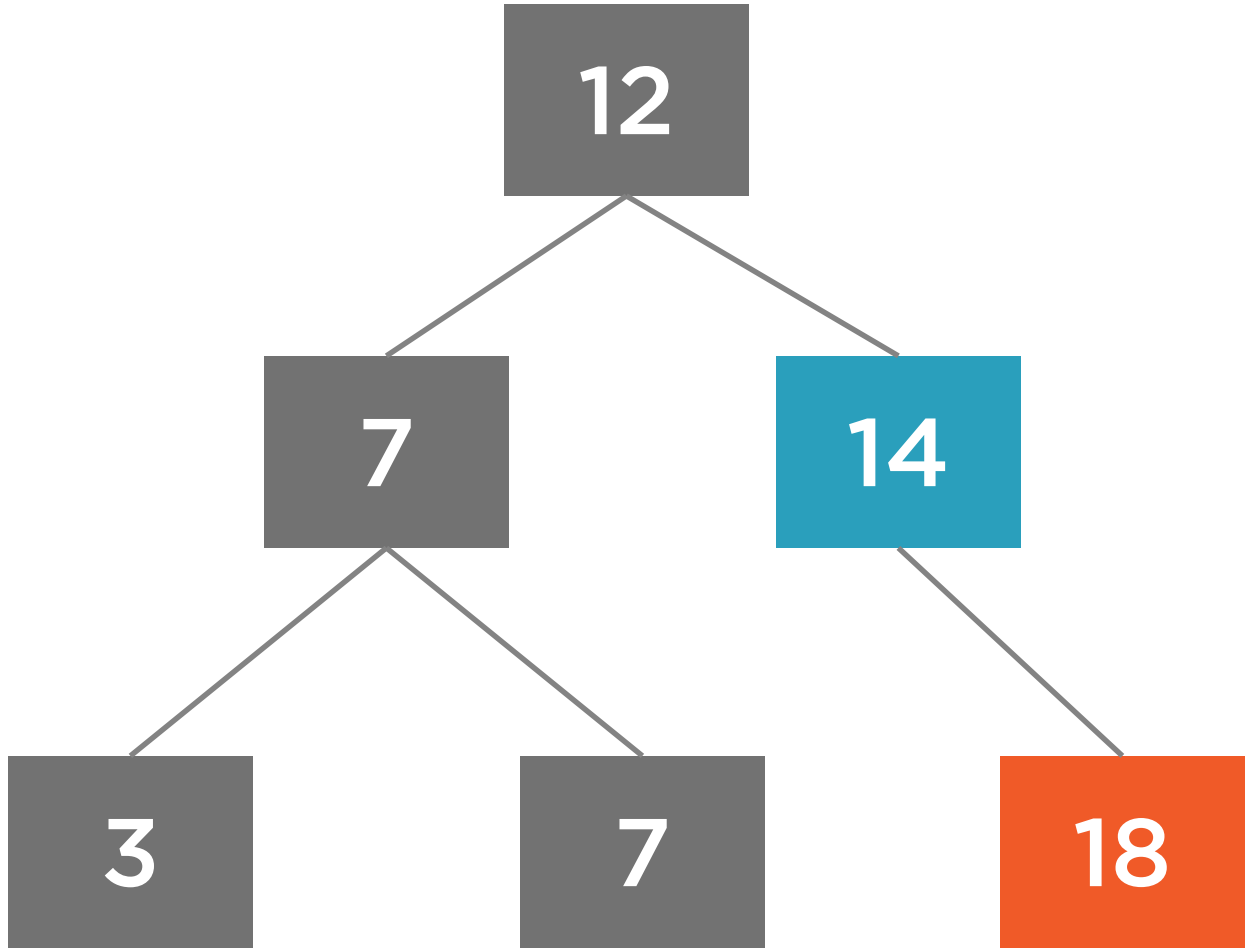
12

7

3

7

14



12

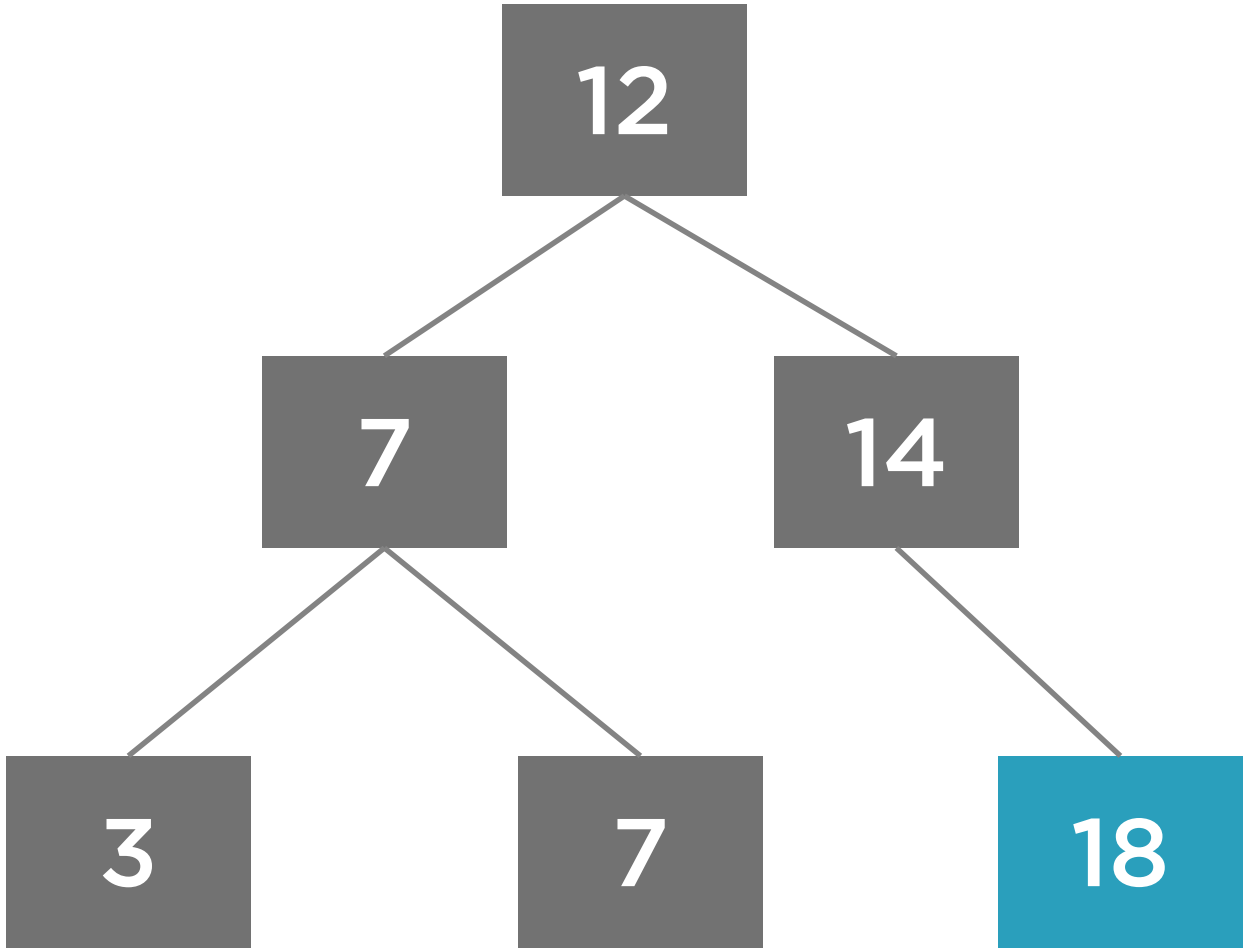
7

3

7

14

18



12

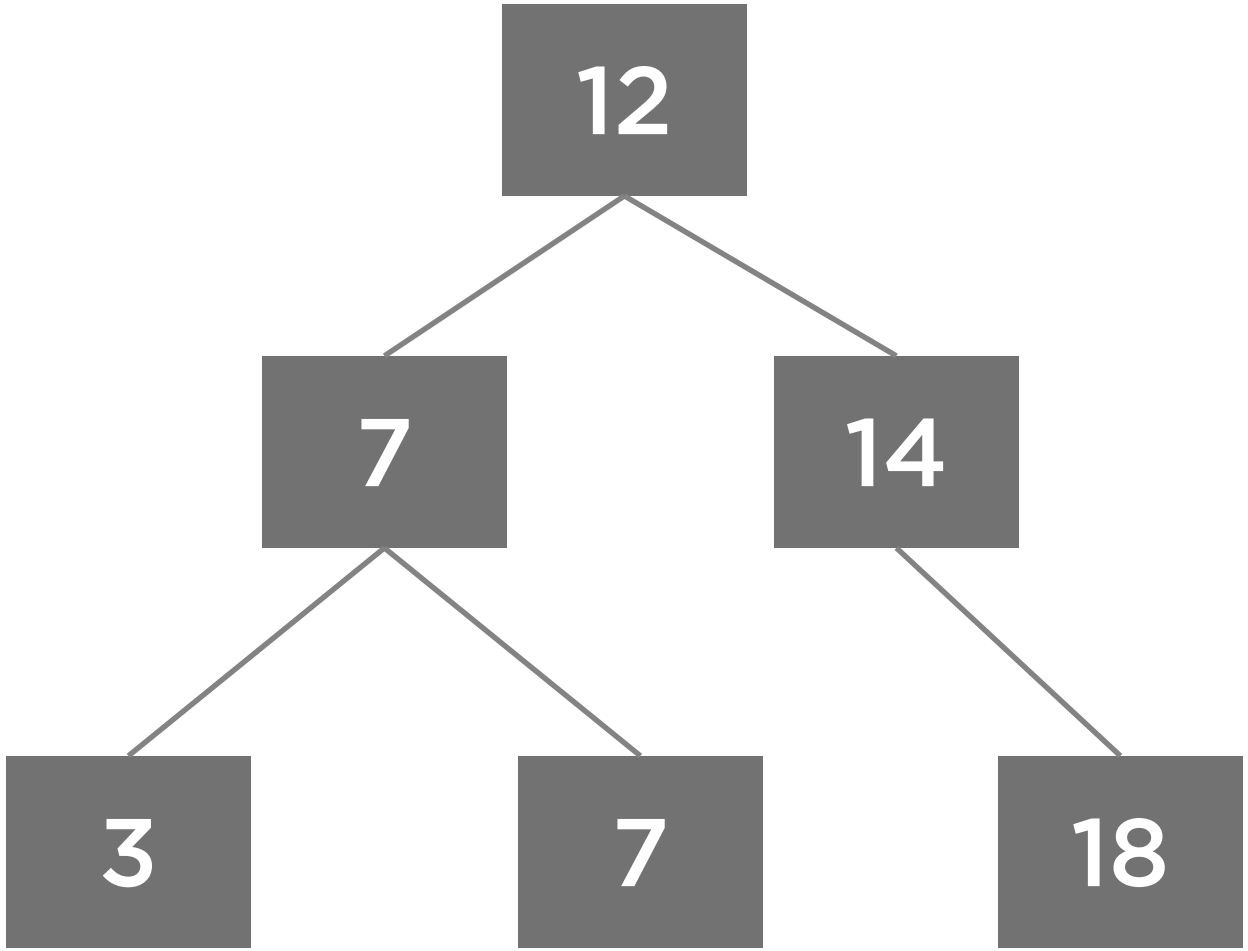
7

3

7

14

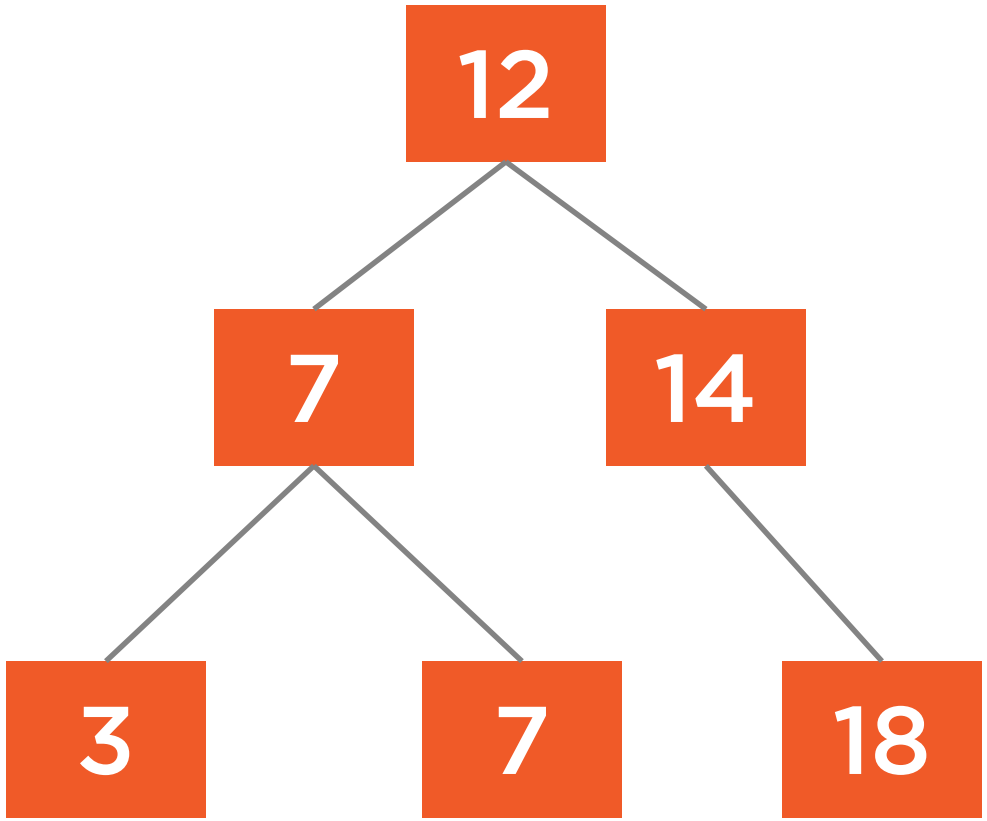
18



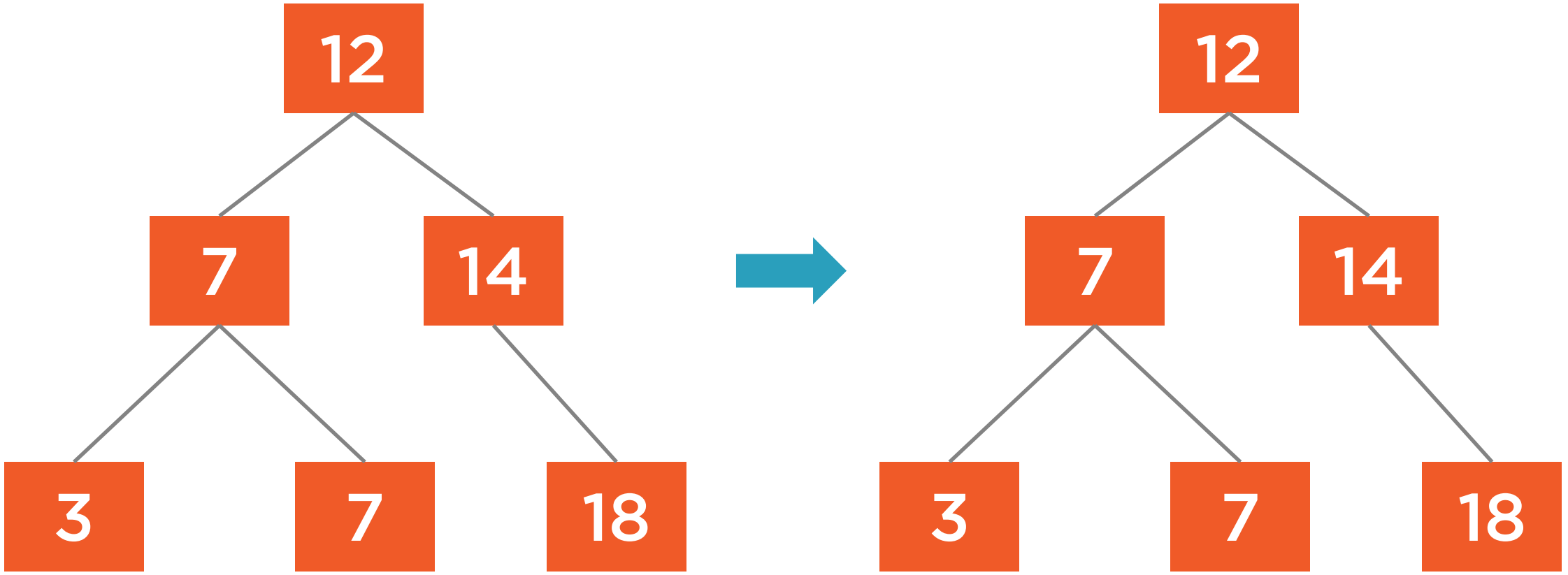
```
public void PreOrderTraversal(
                        Action<T> action)
{
   PreOrderTraversal(action, Root);
}

private void PreOrderTraversal(
                        Action<T> action,
                        BSTNode<T> node)
{
   if (node != null)
   {
      action(node.Value);
      PreOrderTraversal(action,
                                    node.Left);
      PreOrderTraversal(action,
                                    node.Right);
   }
}
```

- Accept an action to perform when processing a node (will be invoked once for each node)
- Call the private recursive function
- Recursive function that visits and processes each node
- Process the node before visiting children
- Visit the left child (recursively)
- Visit the right child (recursively)

Example Usage



Example Usage



```
BinaryTree<int> original = new BinaryTree<int>();

// values are added to the tree

BinaryTree<int> copy = new BinaryTree<int>();

original.PreOrderTraversal((value) => copy.Add(value));
```

Example: Copying a Tree

The pre-order traversal iterates over the nodes in an order that allows creating a copy that has the same values in the same relative position in the new tree.

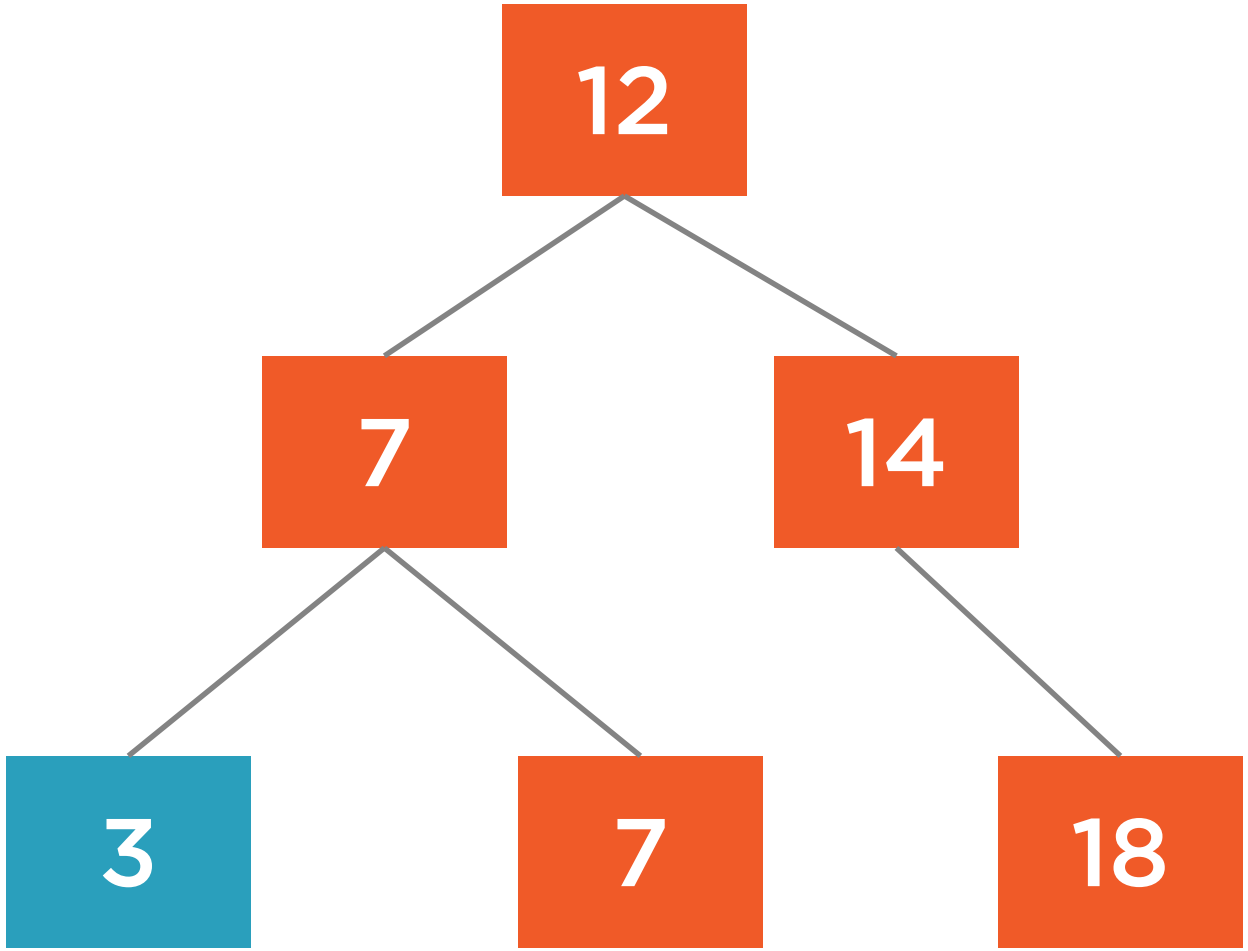
In-order Traversal

Visit the left child

Process the current value

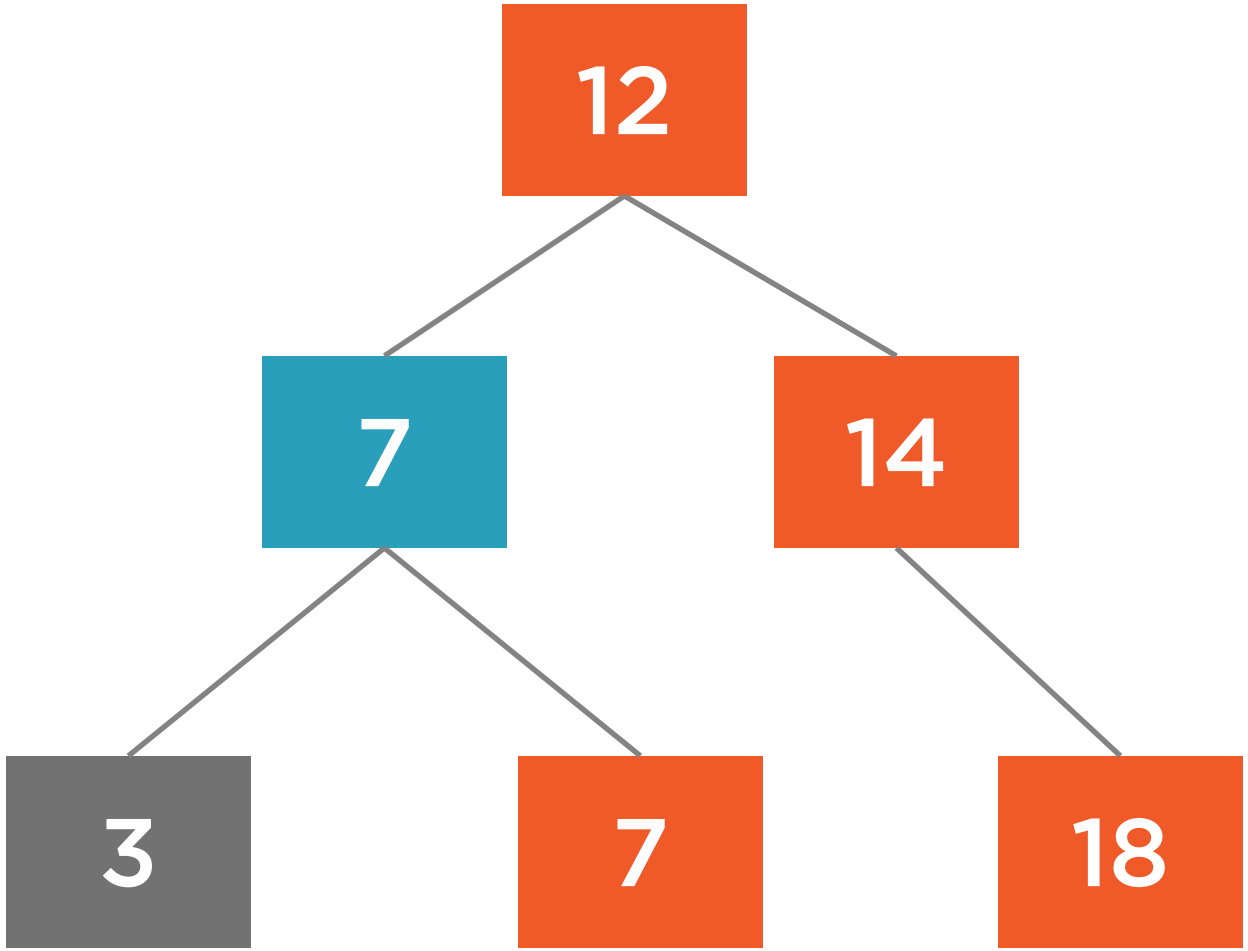
Visit the right child

3



3

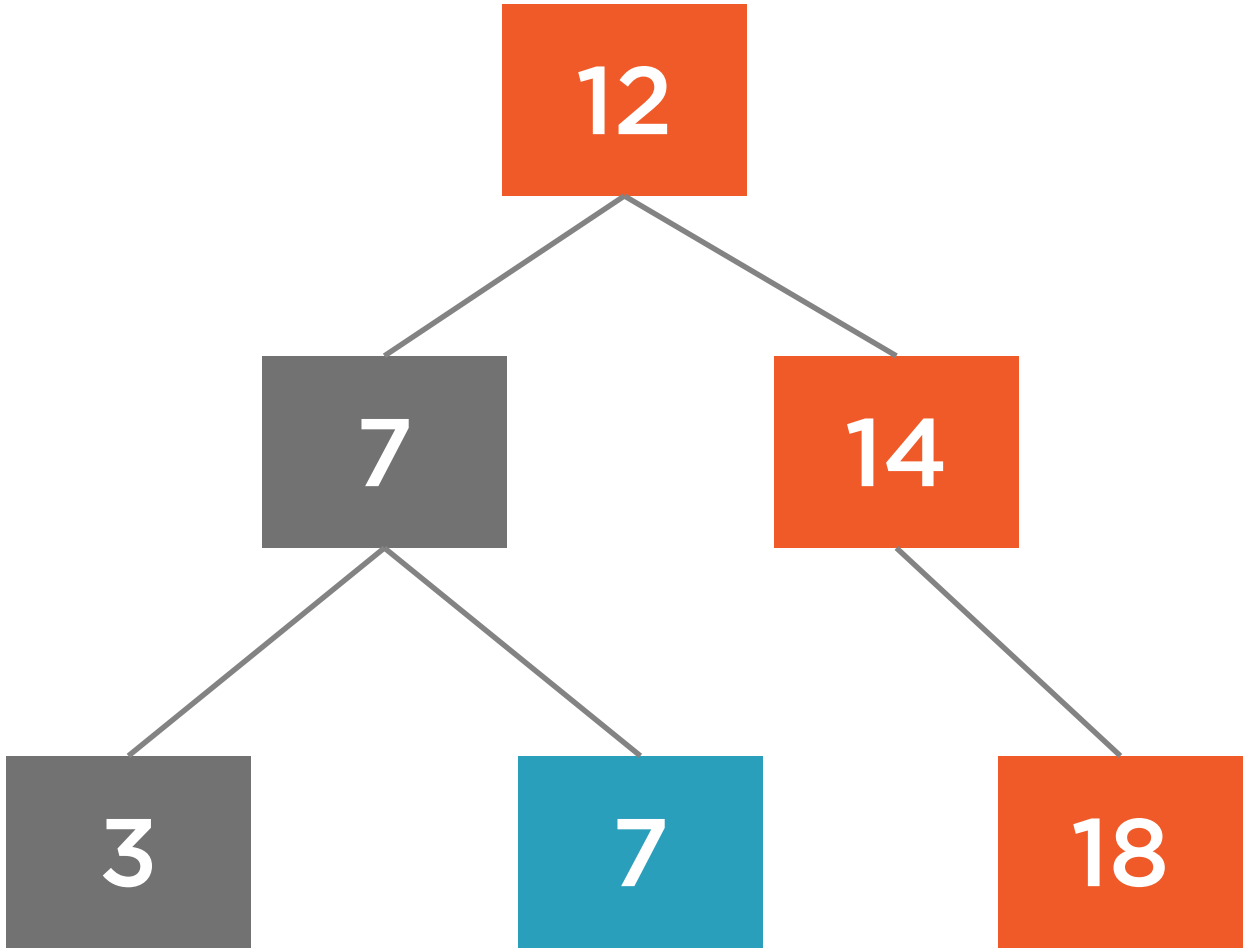
7



3

7

7

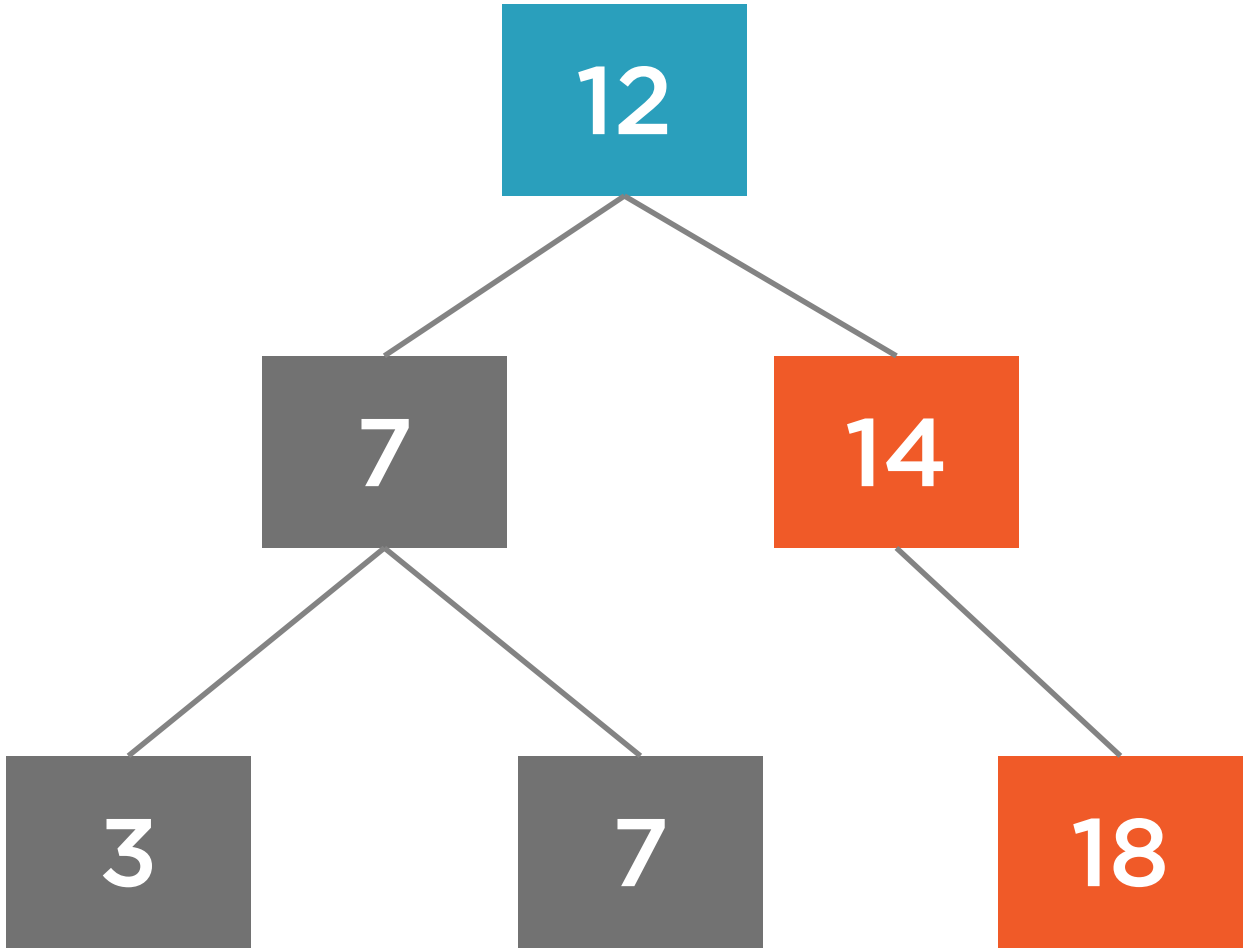


3

7

7

12



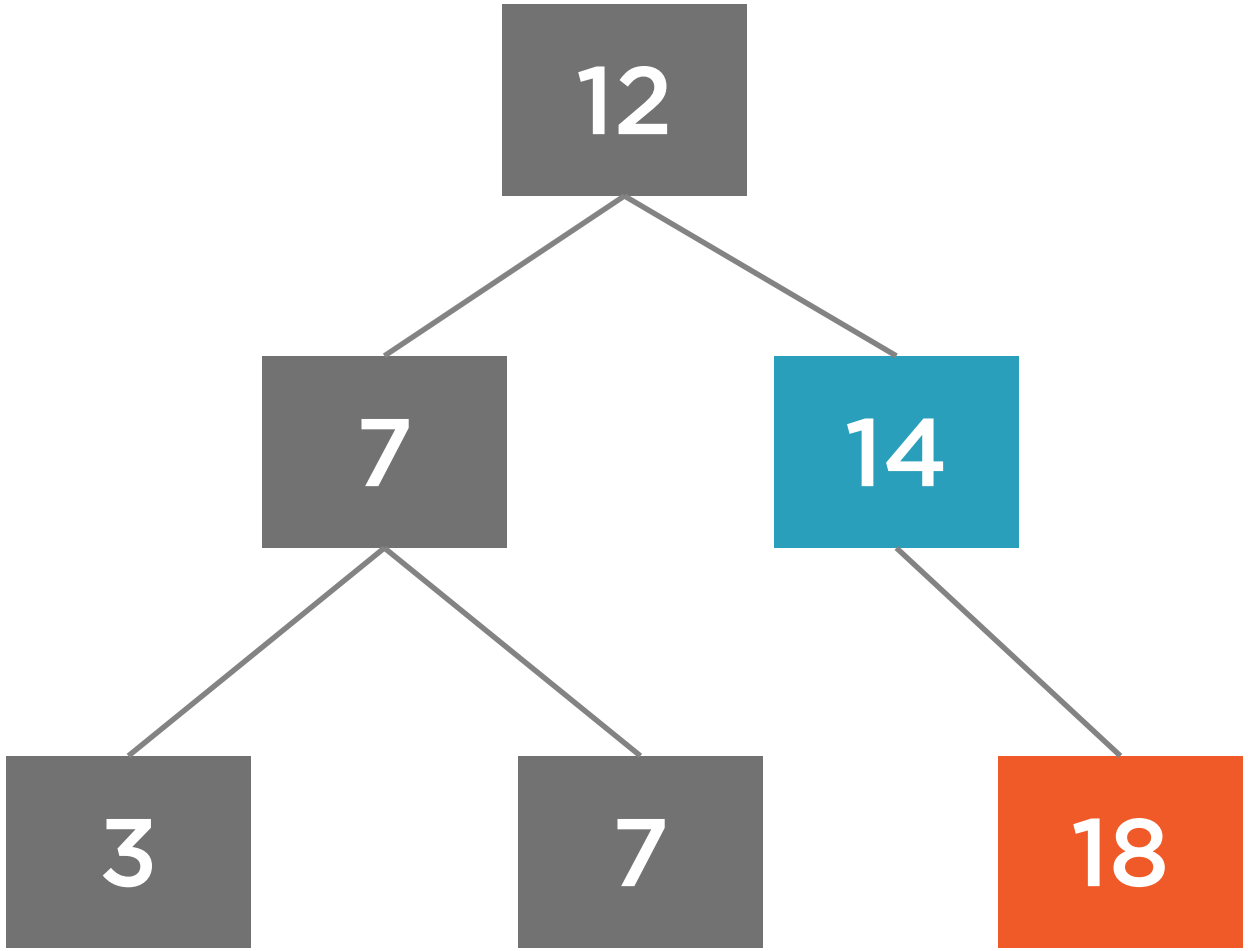
3

7

7

12

14



3

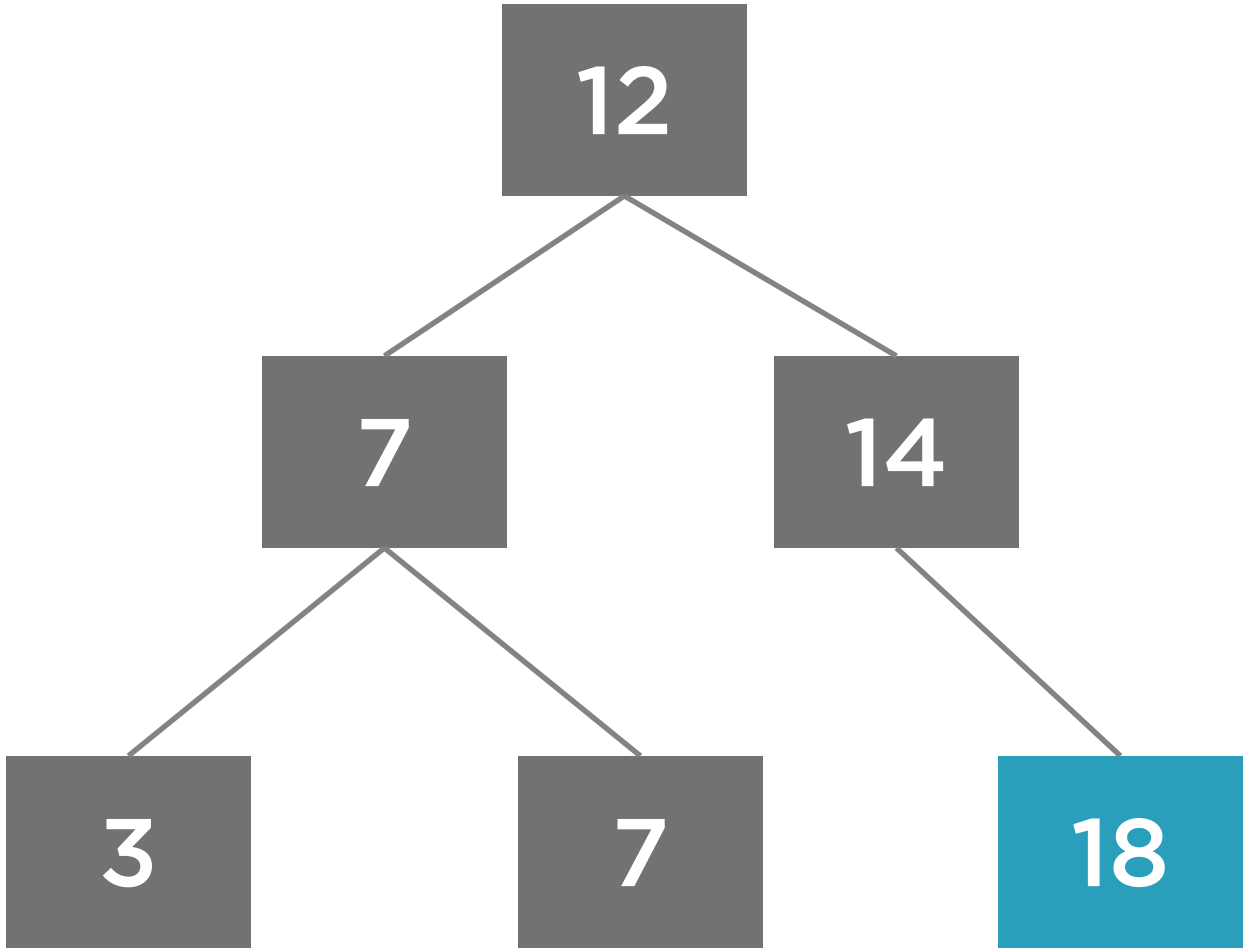
7

7

12

14

18



3

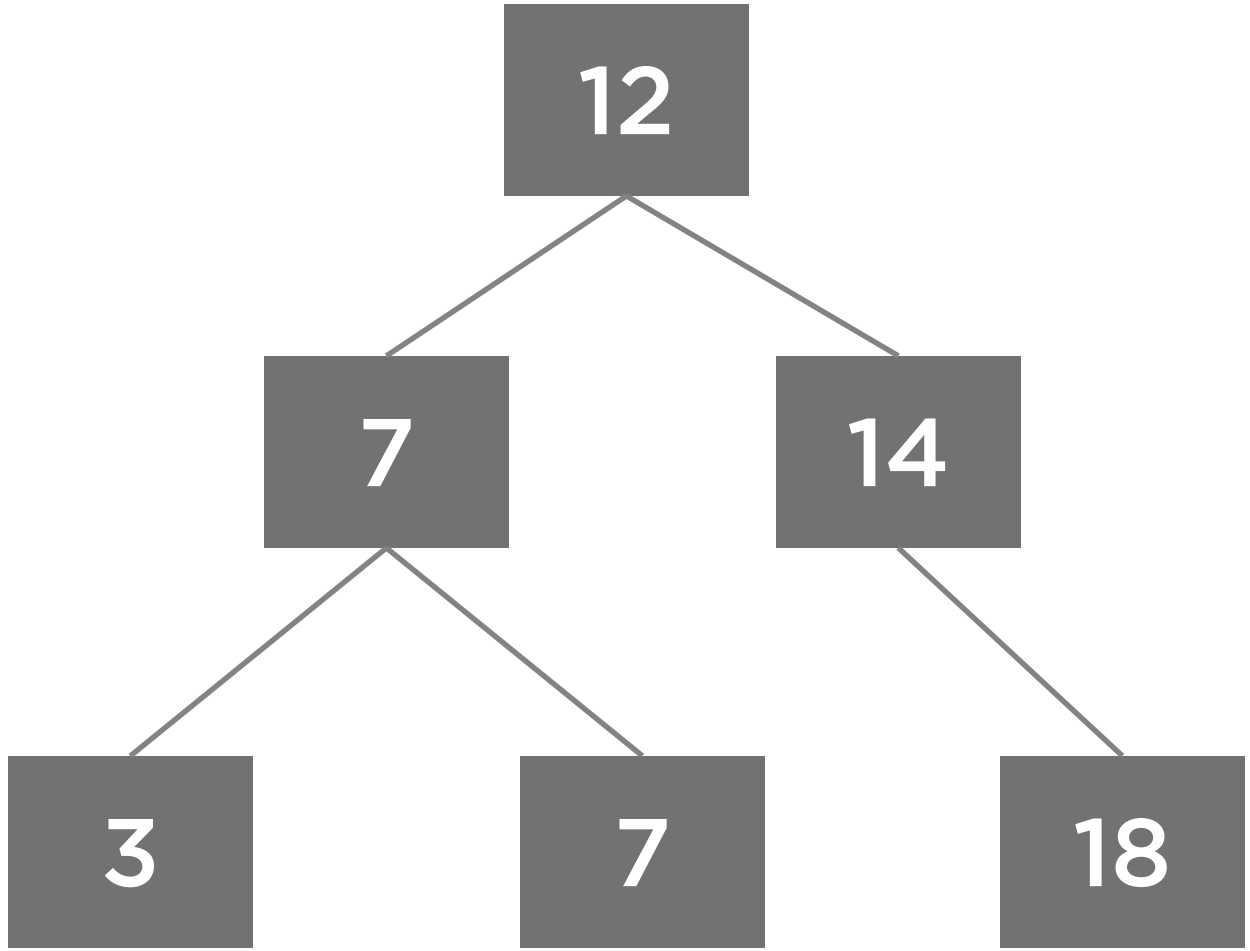
7

7

12

14

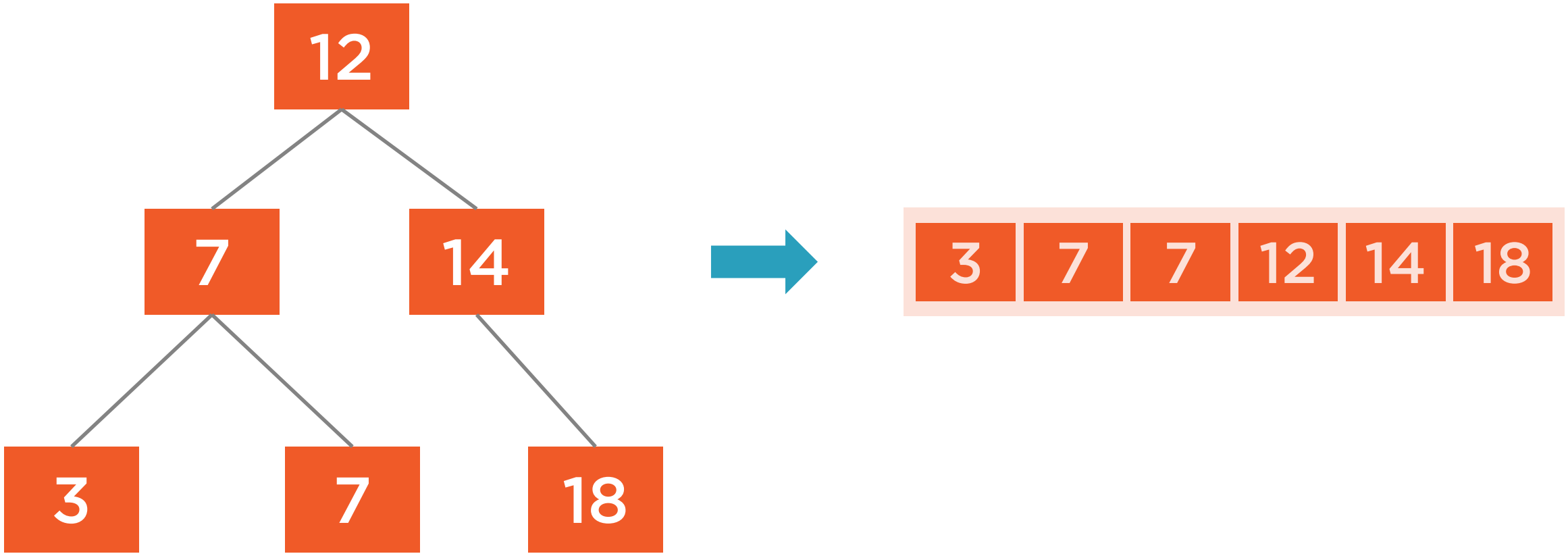
18



```
public void InOrderTraversal(
                        Action<T> action)
{
   InOrderTraversal(action, Root);
}

private void InOrderTraversal(
                        Action<T> action,
                        BSTNode<T> node)
{
   if (node != null)
   {
          InOrderTraversal(action,
                                  node.Left);

      action(node.Value);

          InOrderTraversal(action,
                                  node.Right);
   }
}
```

- Accept an action to perform when processing a node (will be invoked once for each node)
- Call the private recursive function
- Recursive function that visits and processes each node
- Visit the left child (recursively)
- Process the node before visiting children
- Visit the right child (recursively)

Example Usage



```
BinaryTree<int> tree = new BinaryTree<int>();

// values are added to the tree

tree.InOrderTraversal((value) => Console.WriteLine(value));
```

Example: Printing Values in Sort Order

The in-order traversal iterates over the nodes in sort order

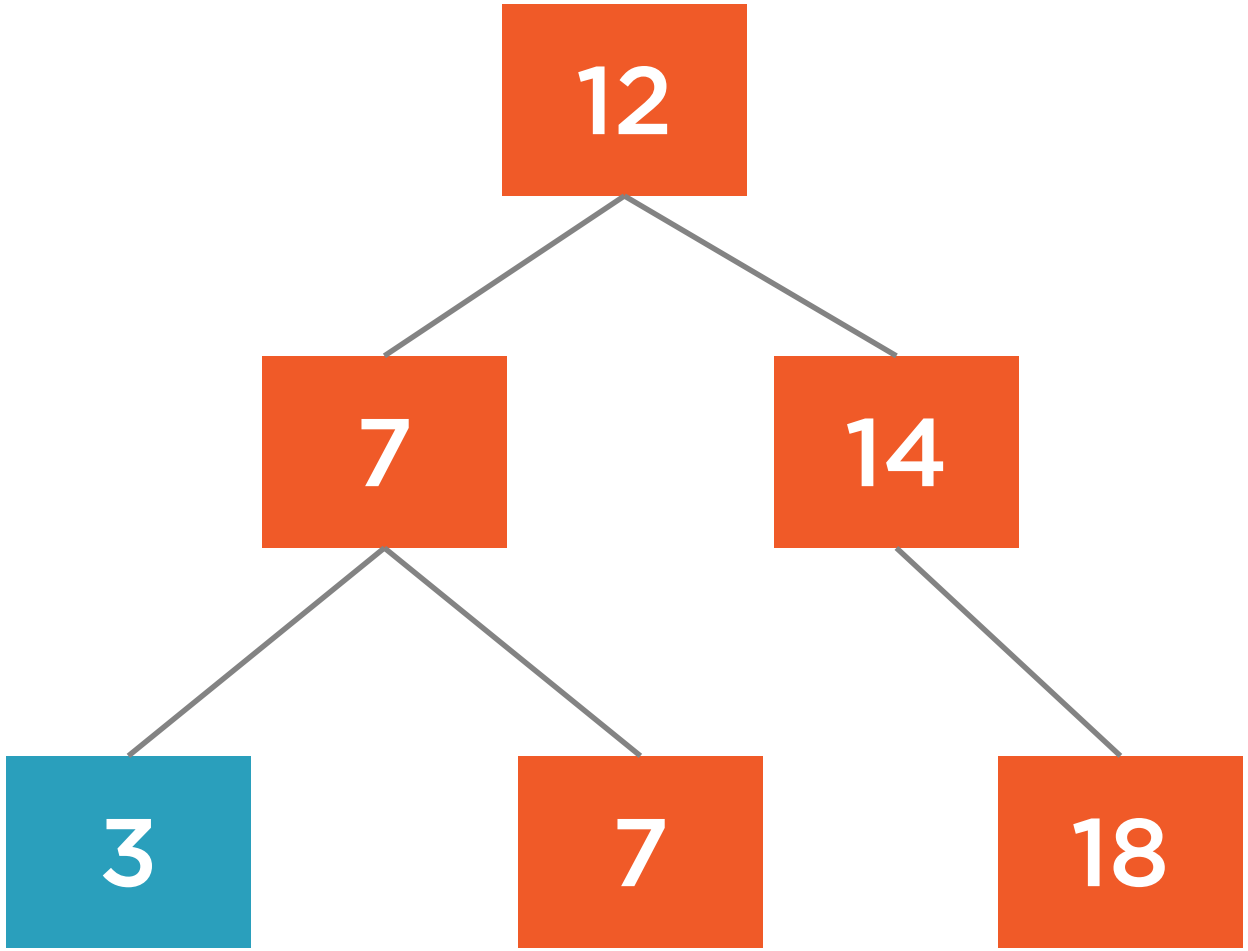
Post-order Traversal

Visit the left child

Visit the right child

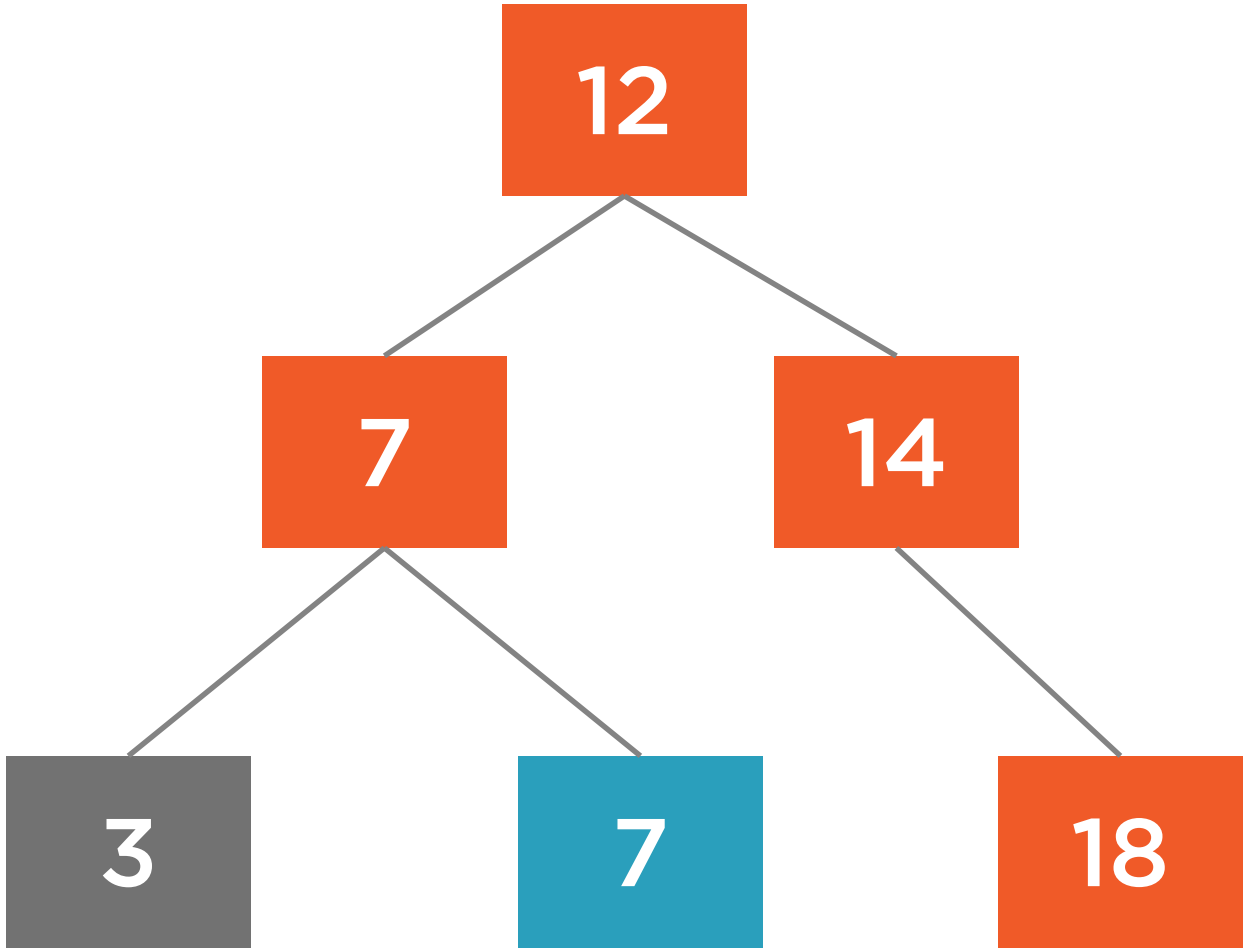
Process the current value

3



3

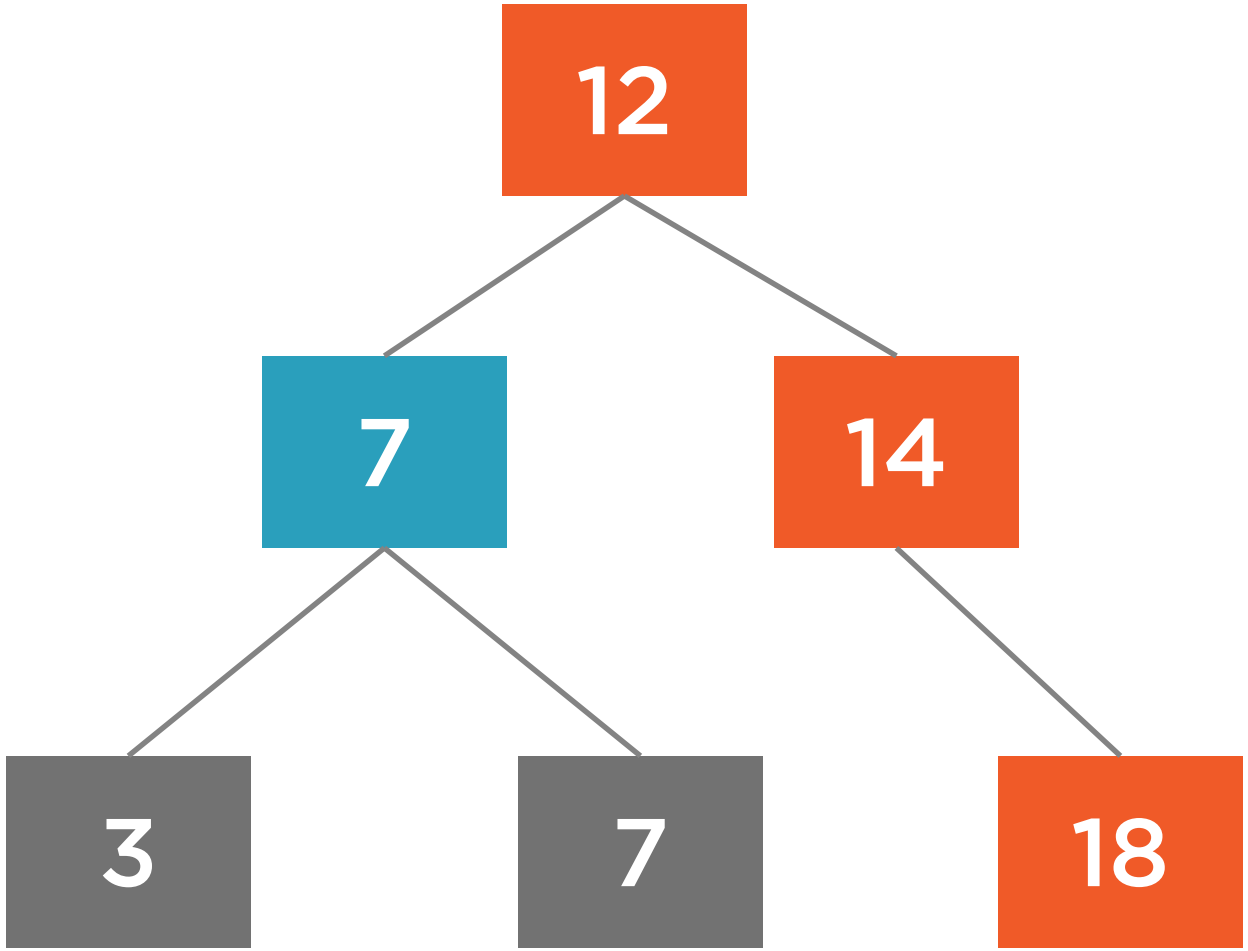
7



3

7

7

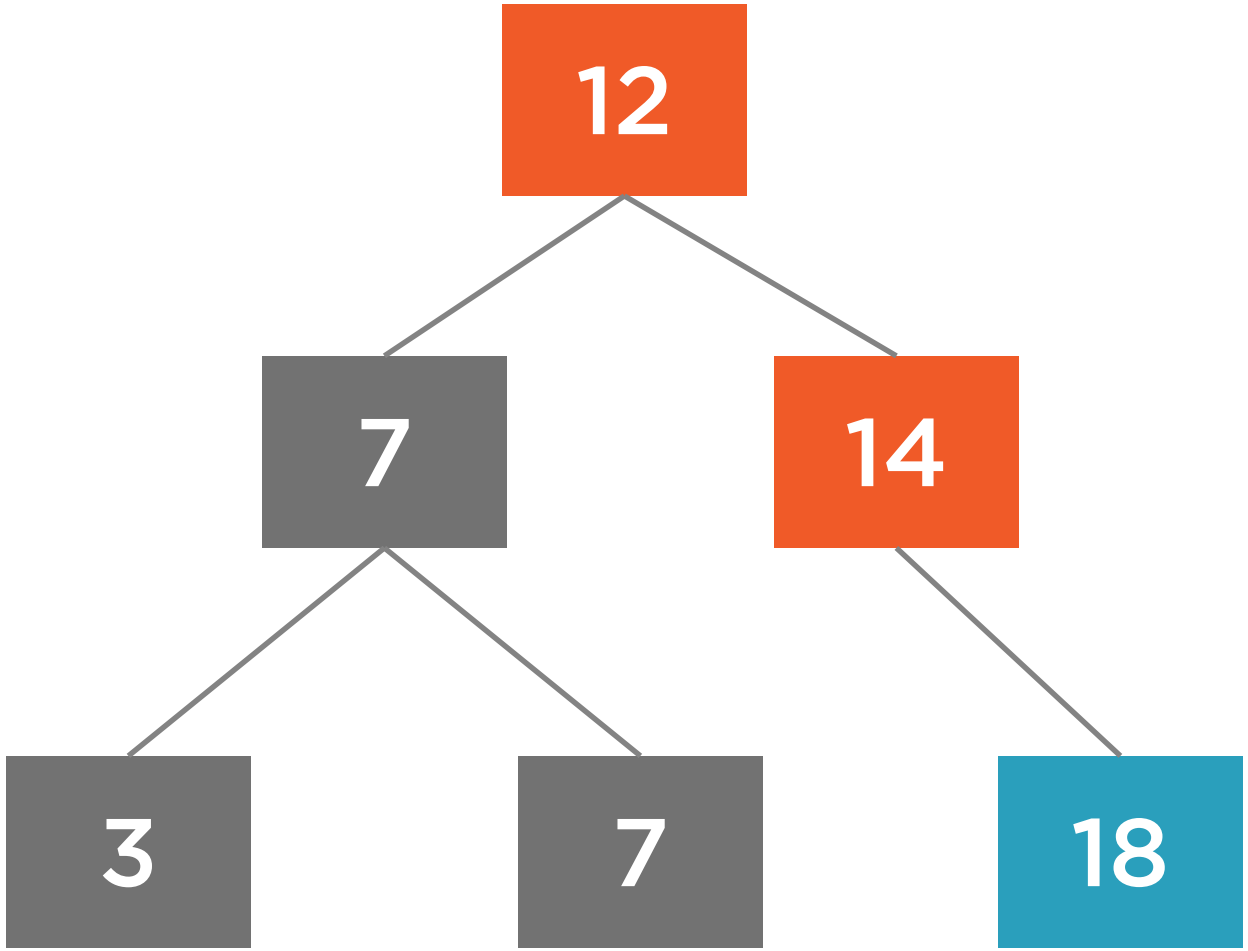


3

7

7

18



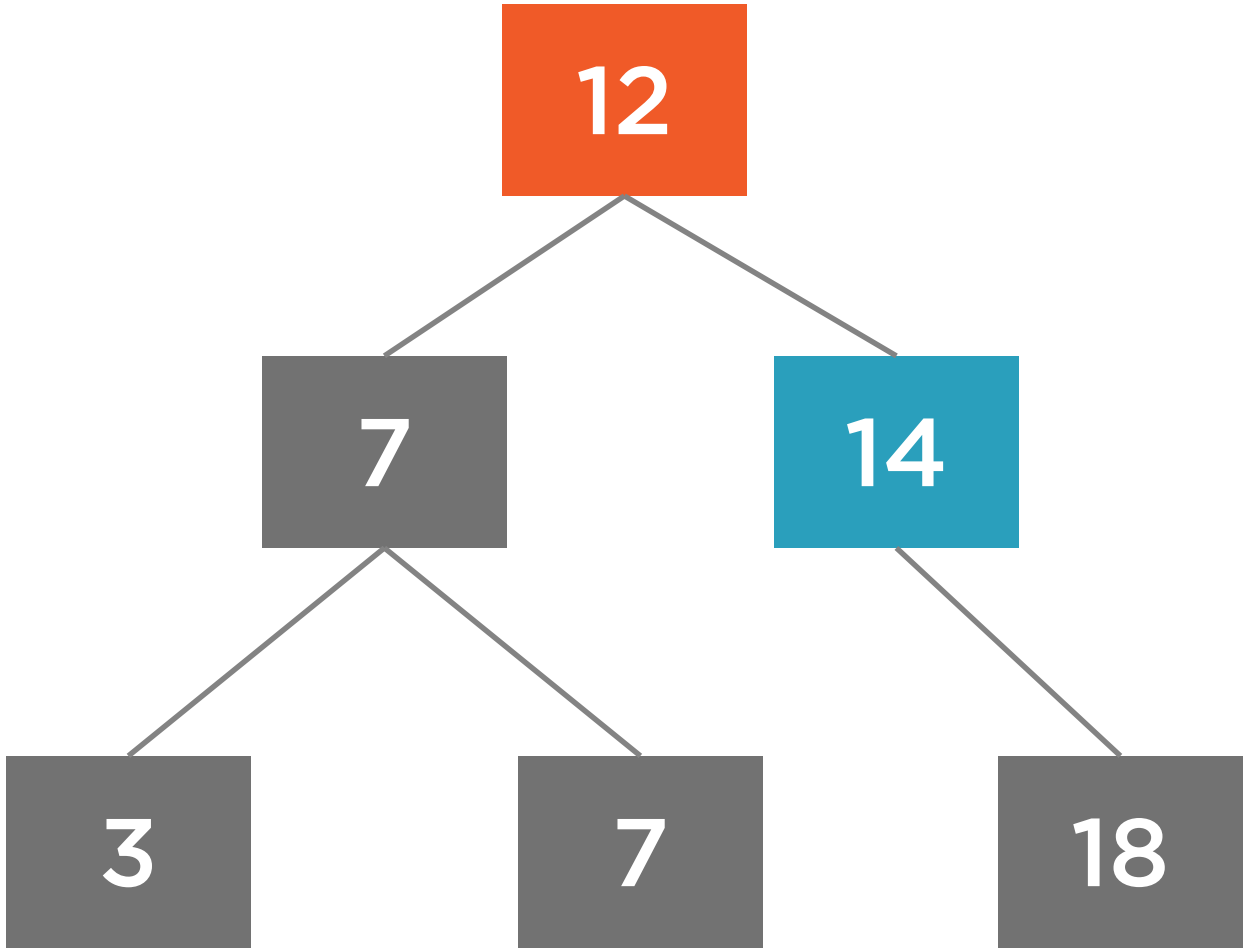
3

7

7

18

14



3

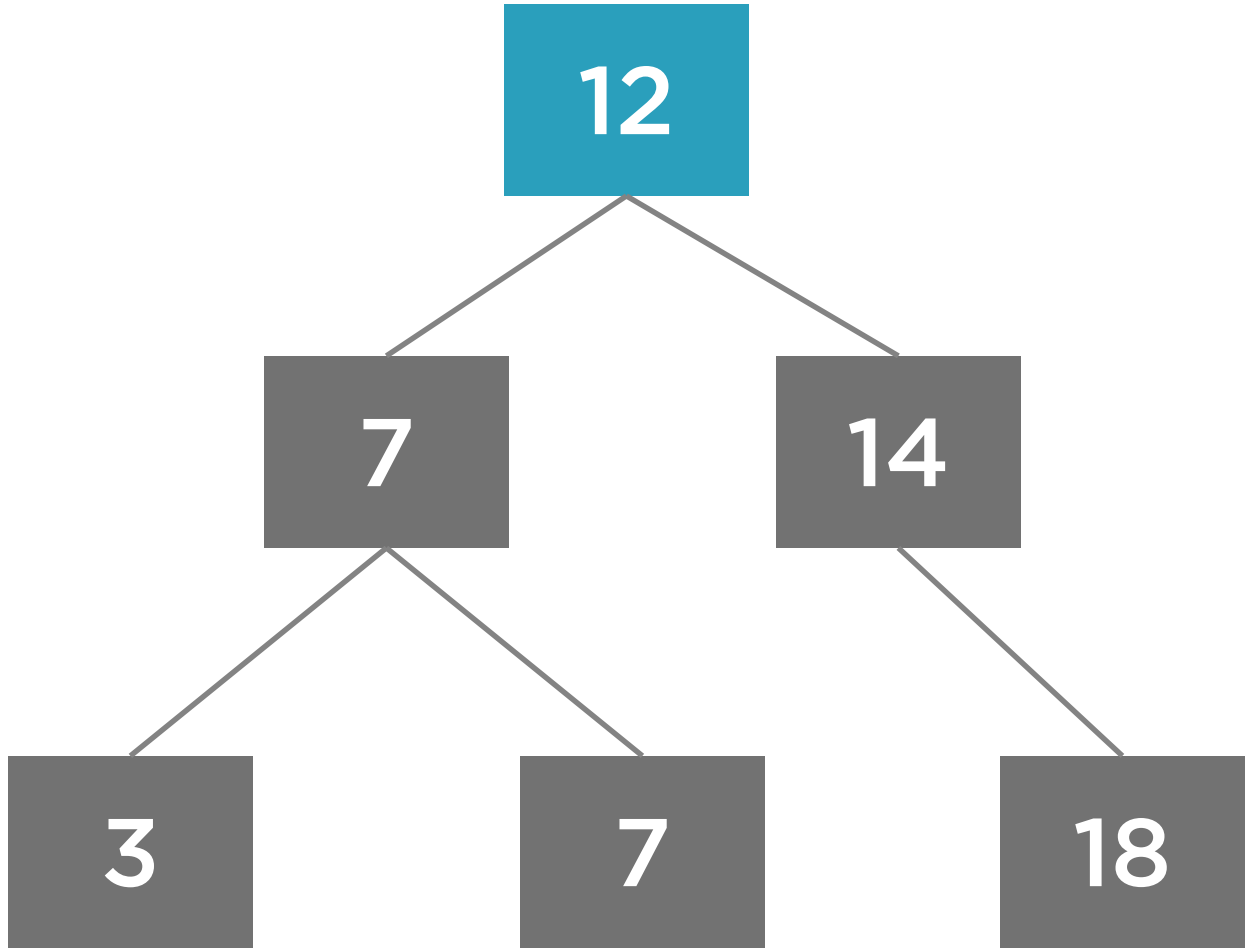
7

7

18

14

12



3

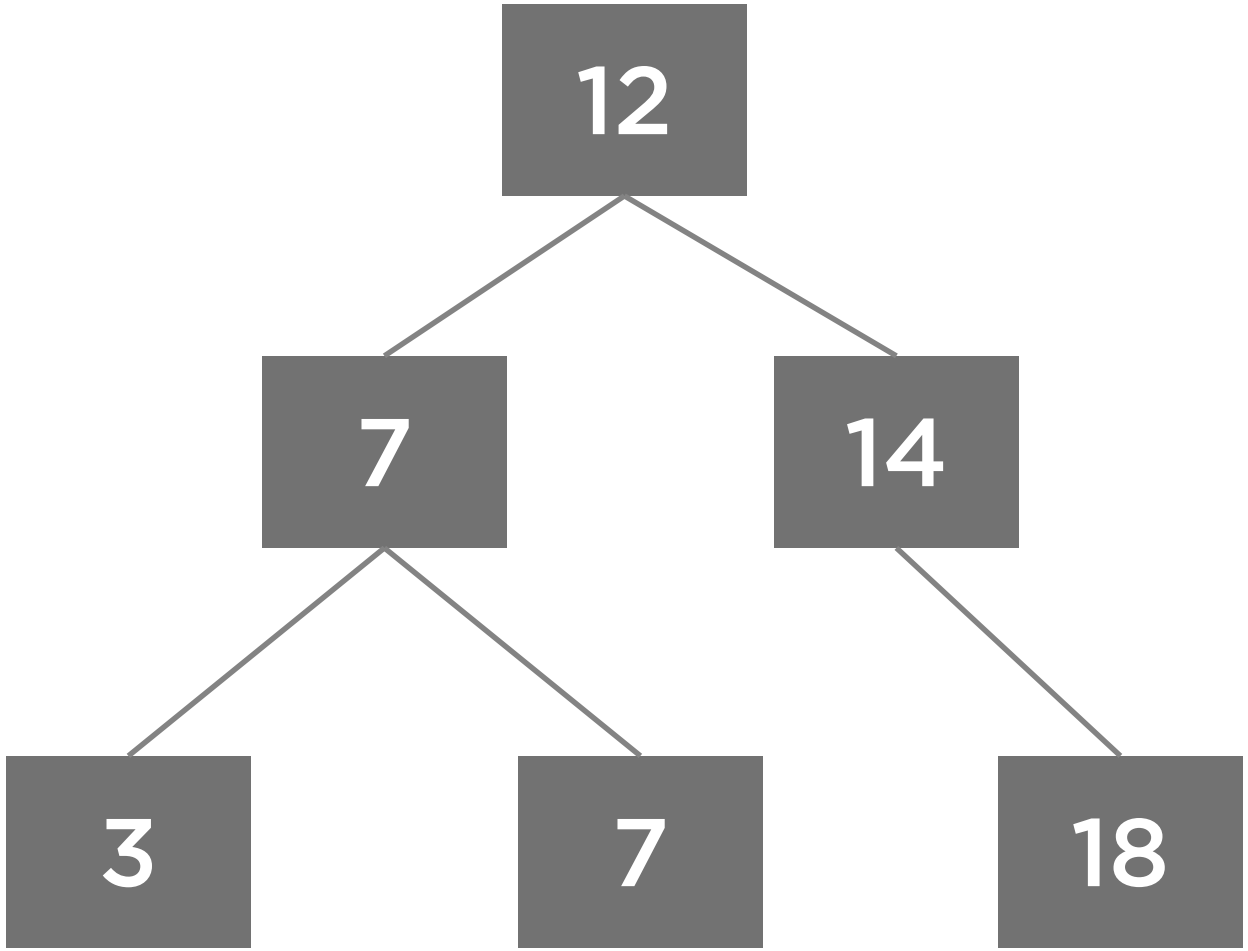
7

7

18

14

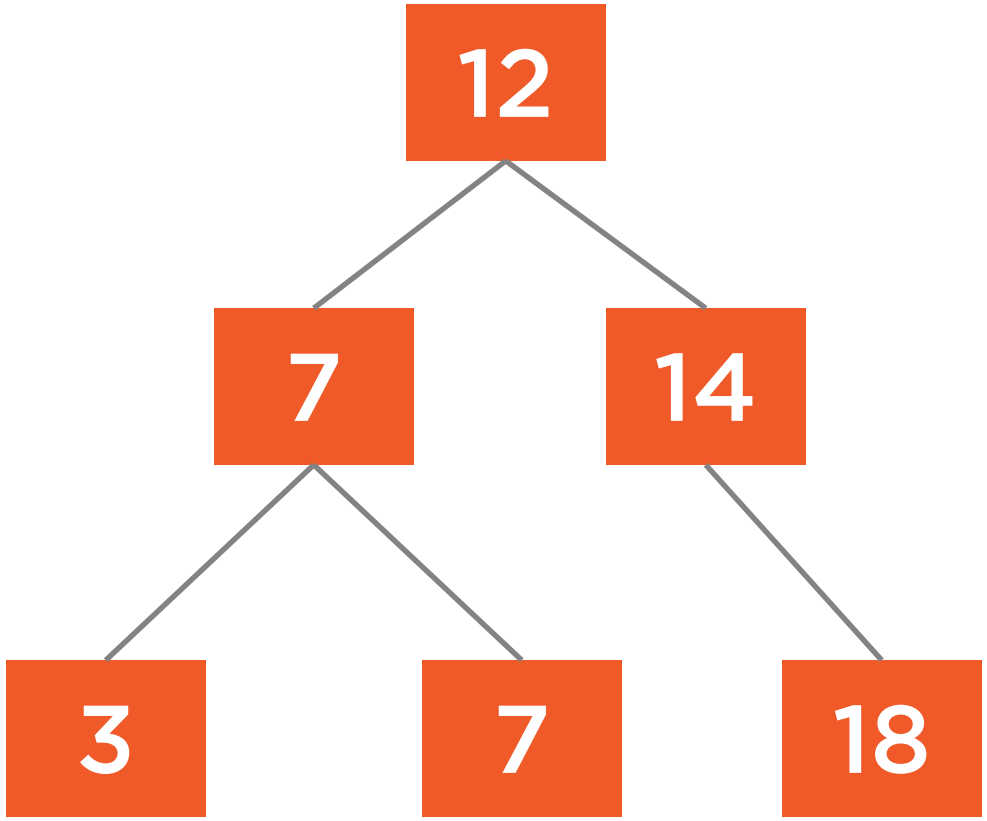
12



```
public void PostOrderTraversal(
                        Action<T> action)
{
   PostOrderTraversal(action, Root);
}

private void PostOrderTraversal(
                        Action<T> action,
                        BSTNode<T> node)
{
   if (node != null)
   {
         PostOrderTraversal(action,
                                    node.Left);

         PostOrderTraversal(action,
                                    node.Right);

         action(node.Value);
   }
}
```

- Accept an action to perform when processing a node (will be invoked once for each node)
- Call the private recursive function
- Recursive function that visits and processes each node
- Visit the left child (recursively)
- Visit the right child (recursively)
- Process the node before visiting children

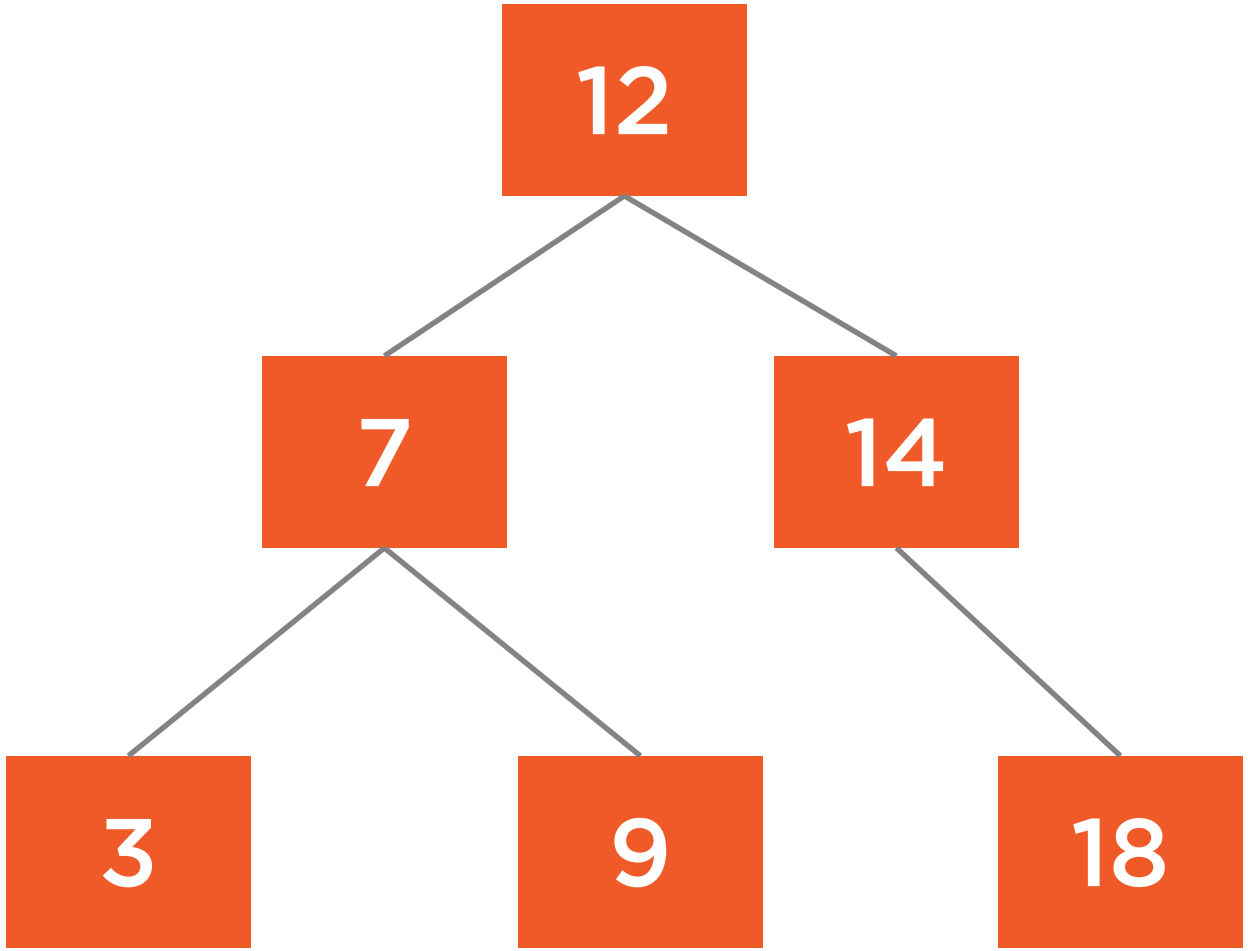
Example Usage



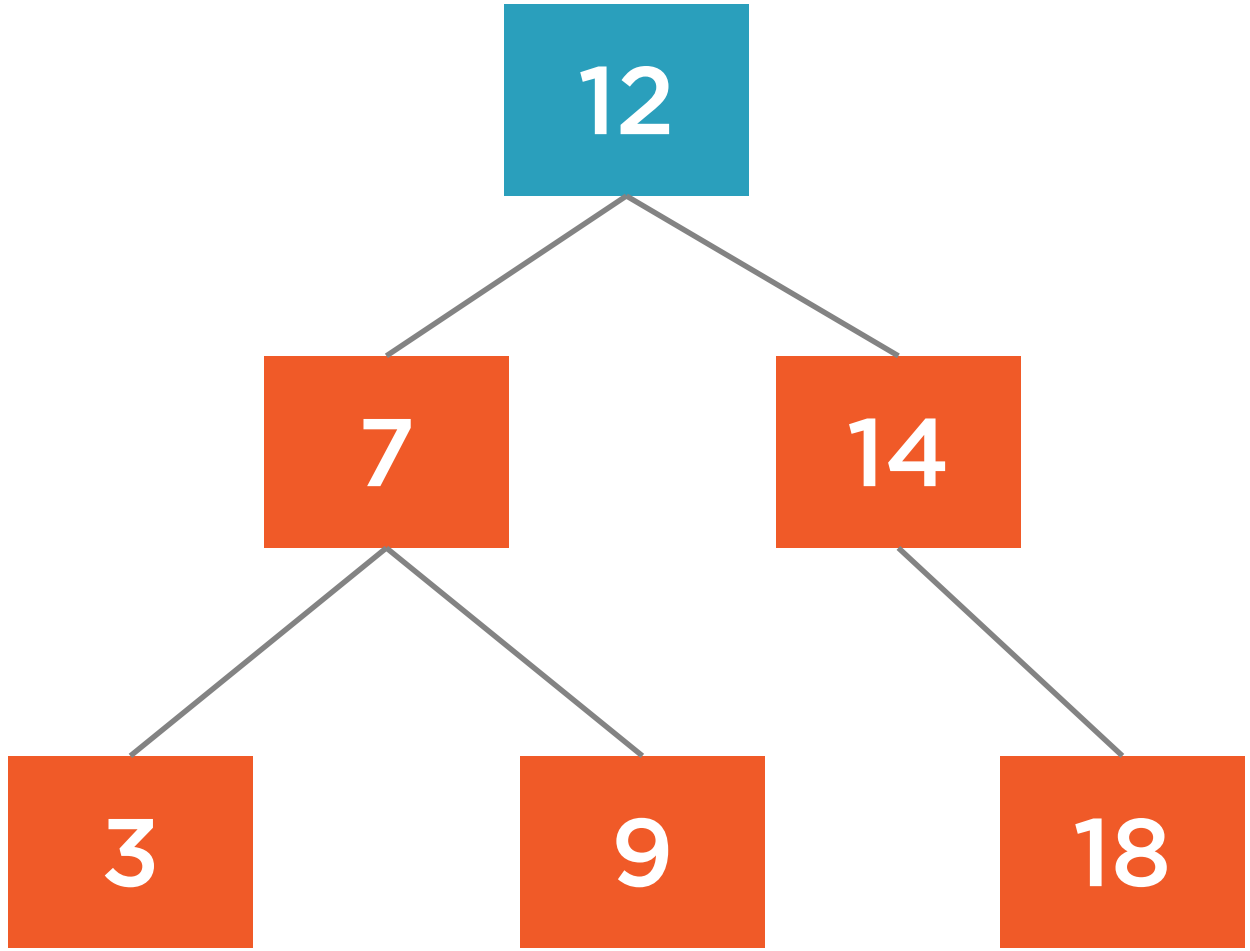
Traversal Complexity

Average case	Worst case
O(n)	O(n)

Searching

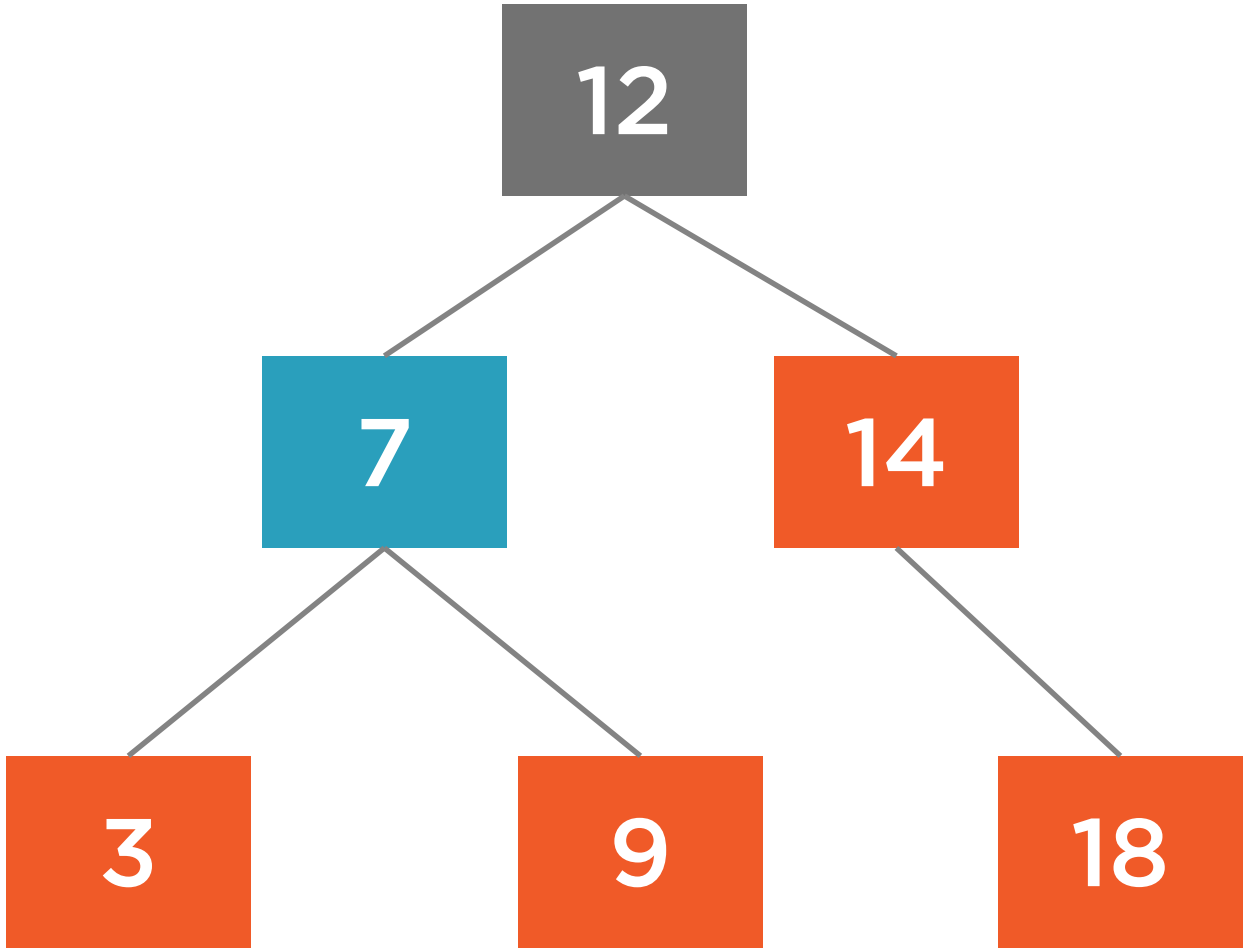

12
7
14
3
9
18

12



12

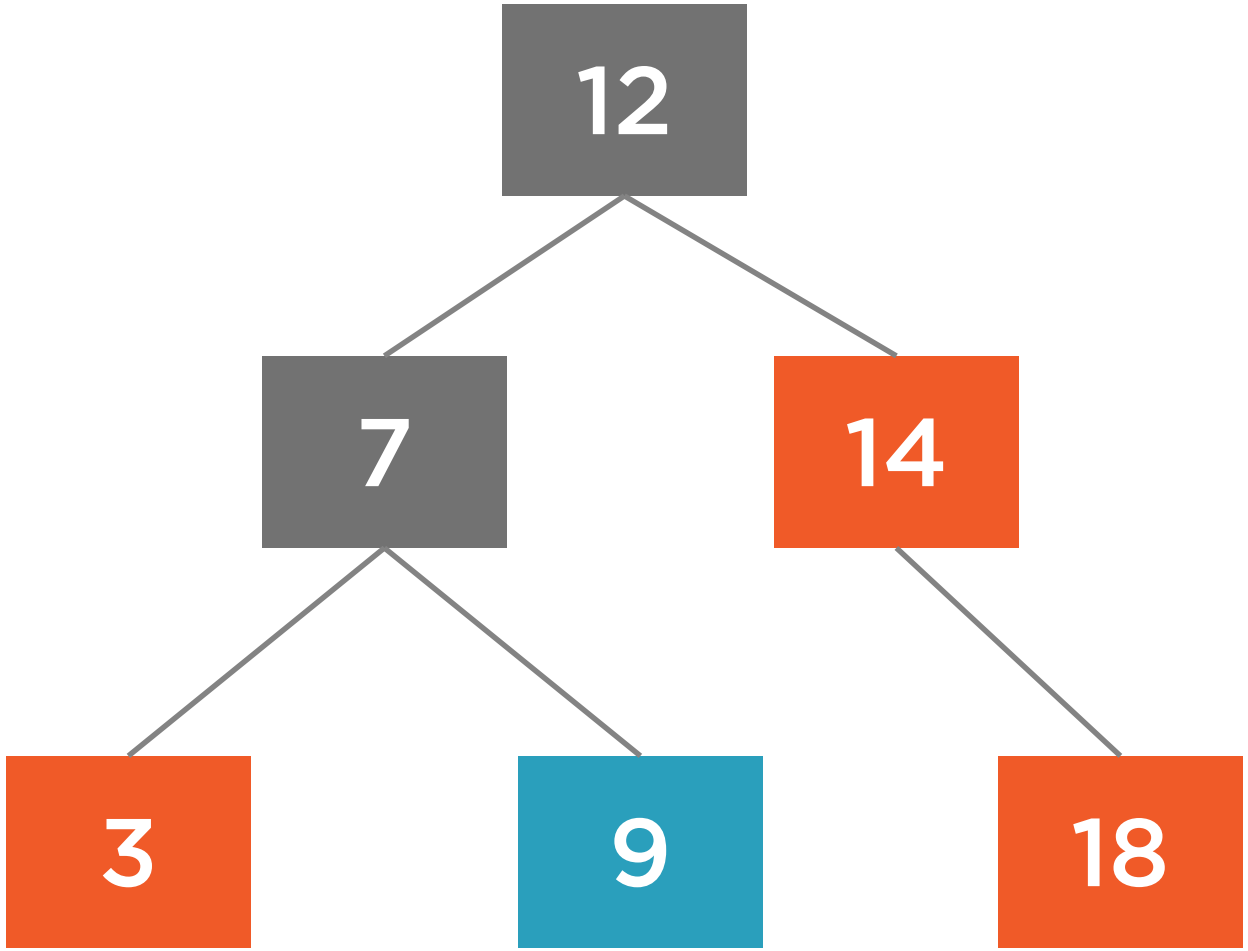
7



12

7

7



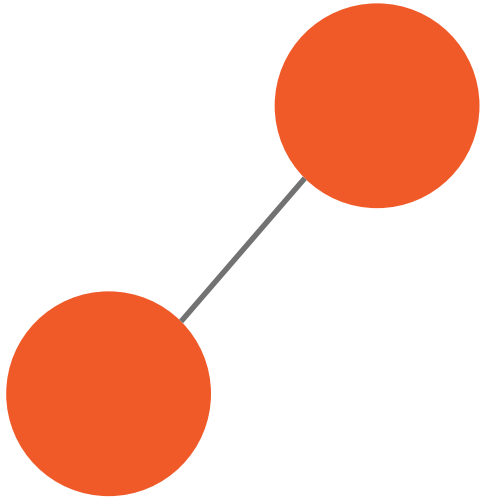
Traversal Complexity

Average case	Worst case
O(log n)	O(n)

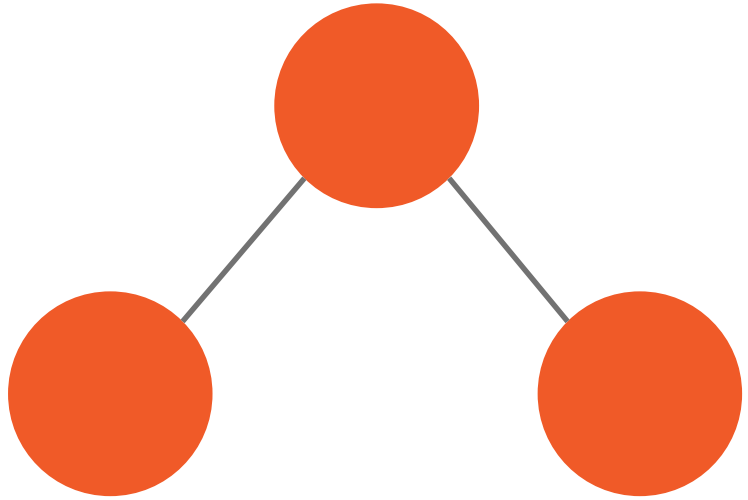
Removal

Removal Cases

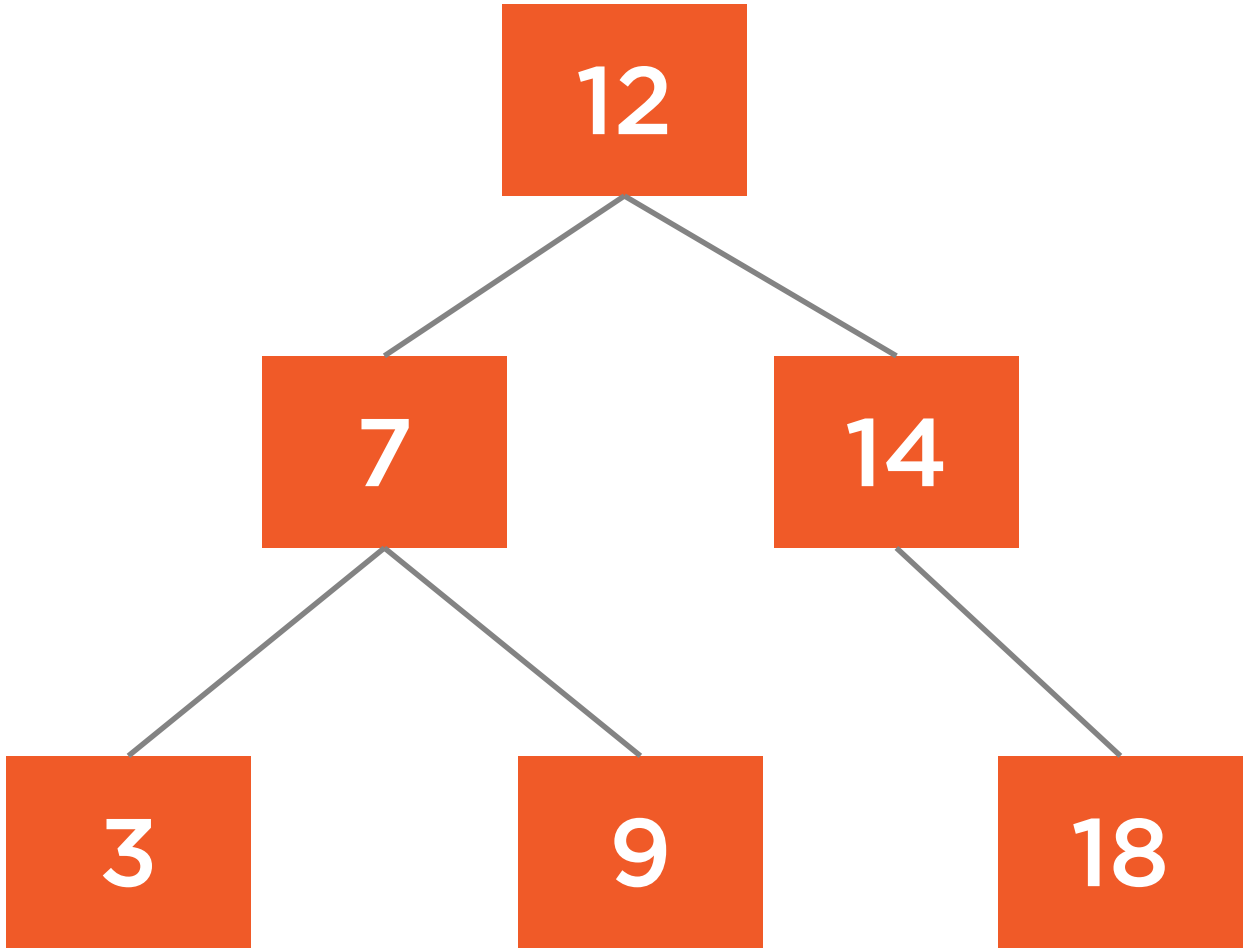
No Children
(Leaf node)



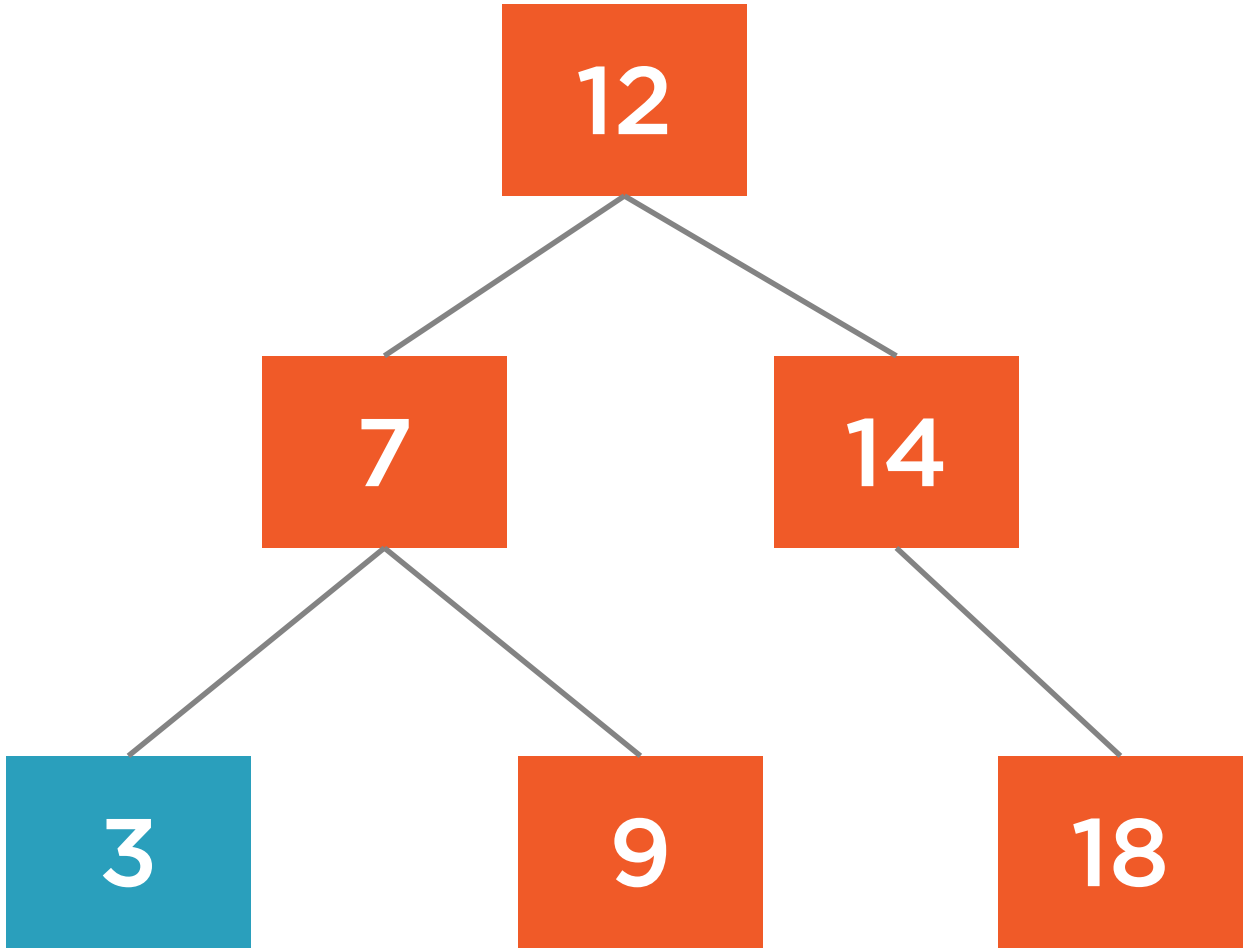
One Child



Two Children

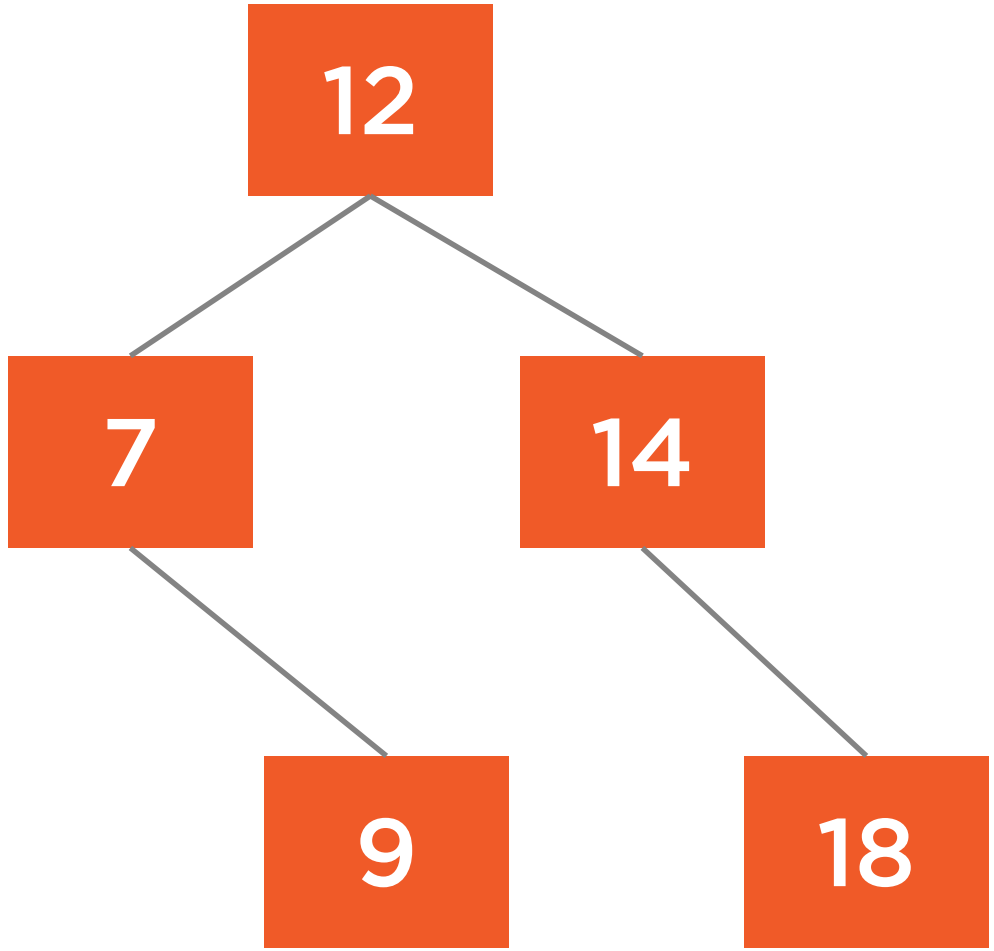

12
7
14
3
9
18

Find the node



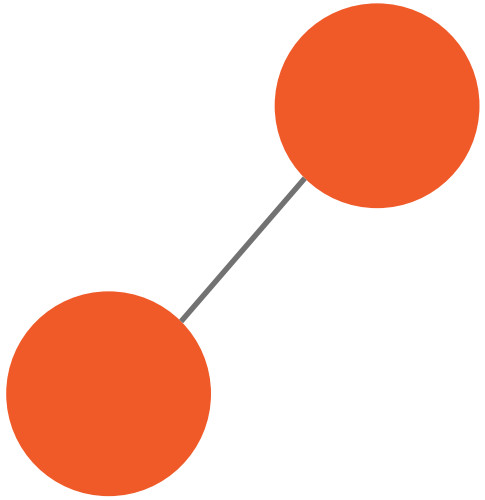
Find the node

Unlink from parent

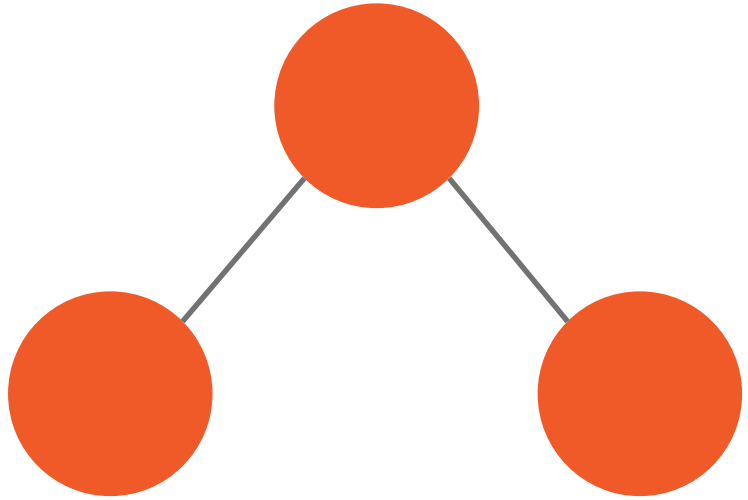


Removal Cases

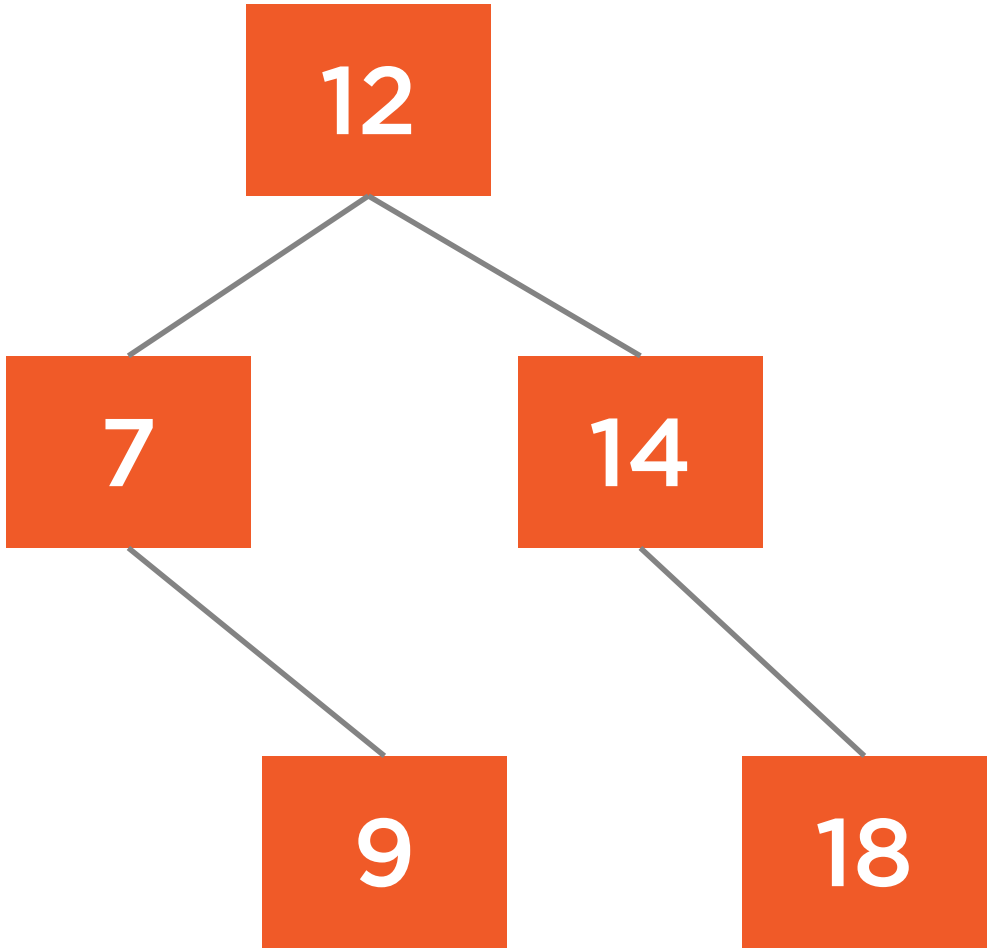
No Children
(Leaf node)



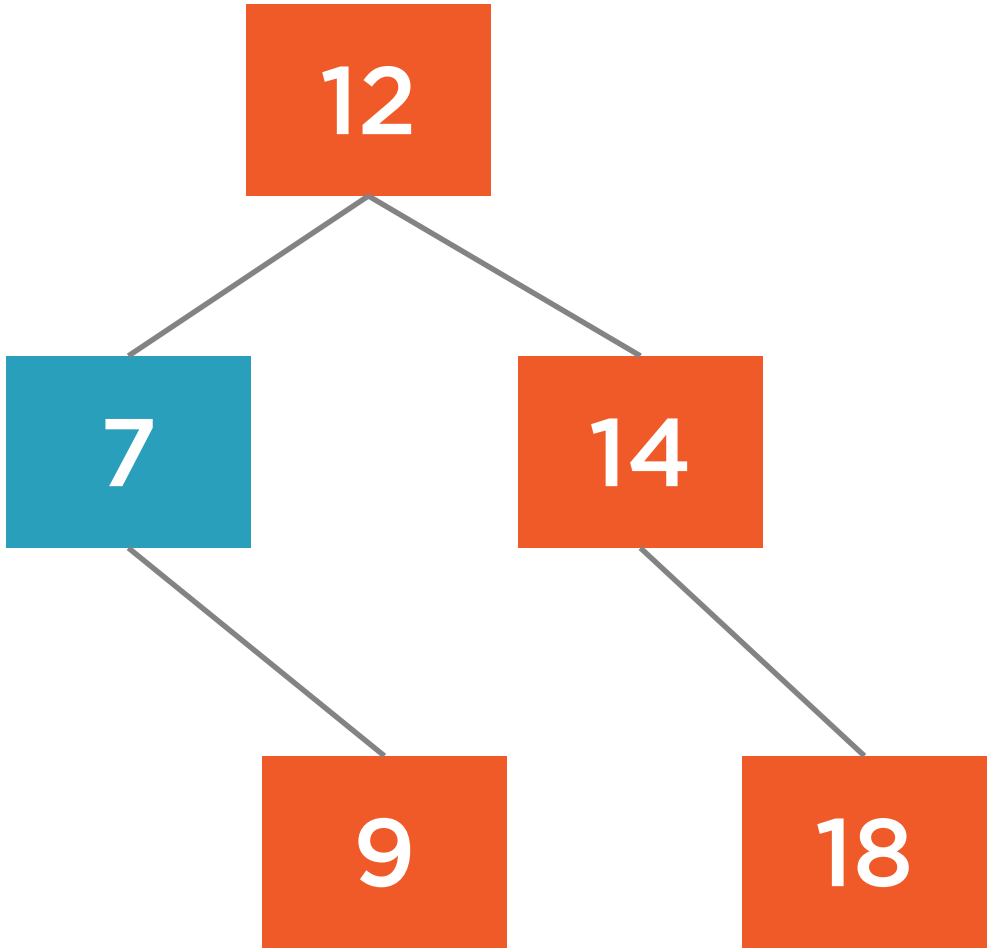
One Child



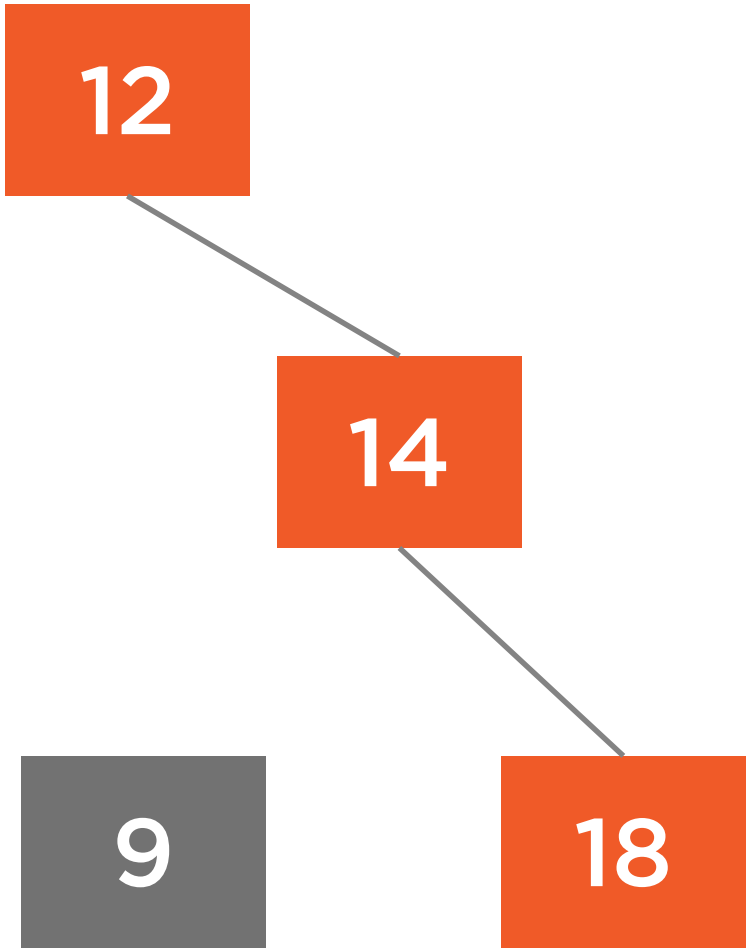
Two Children


12
7
14
9
18

Find the node

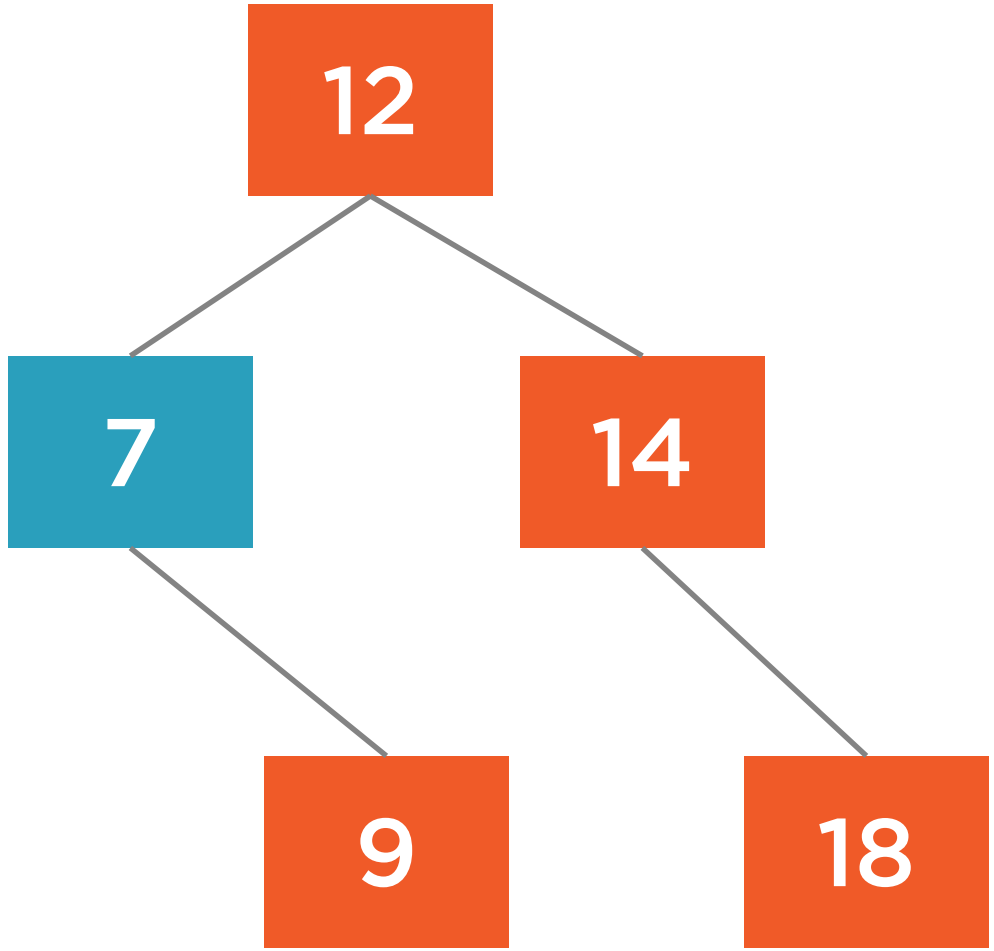


Find the node



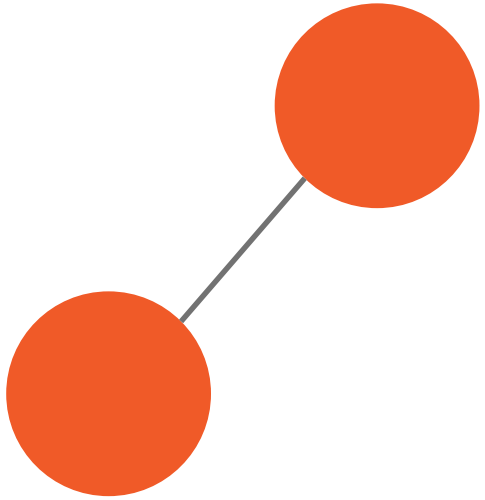
Find the node

Promote the child

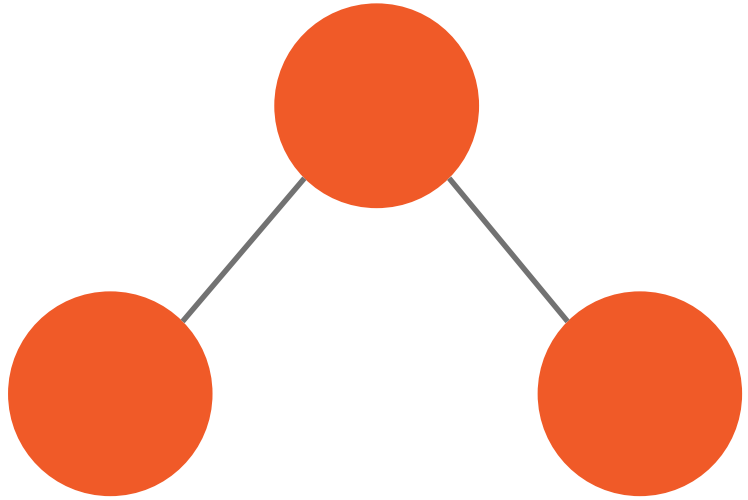


Removal Cases

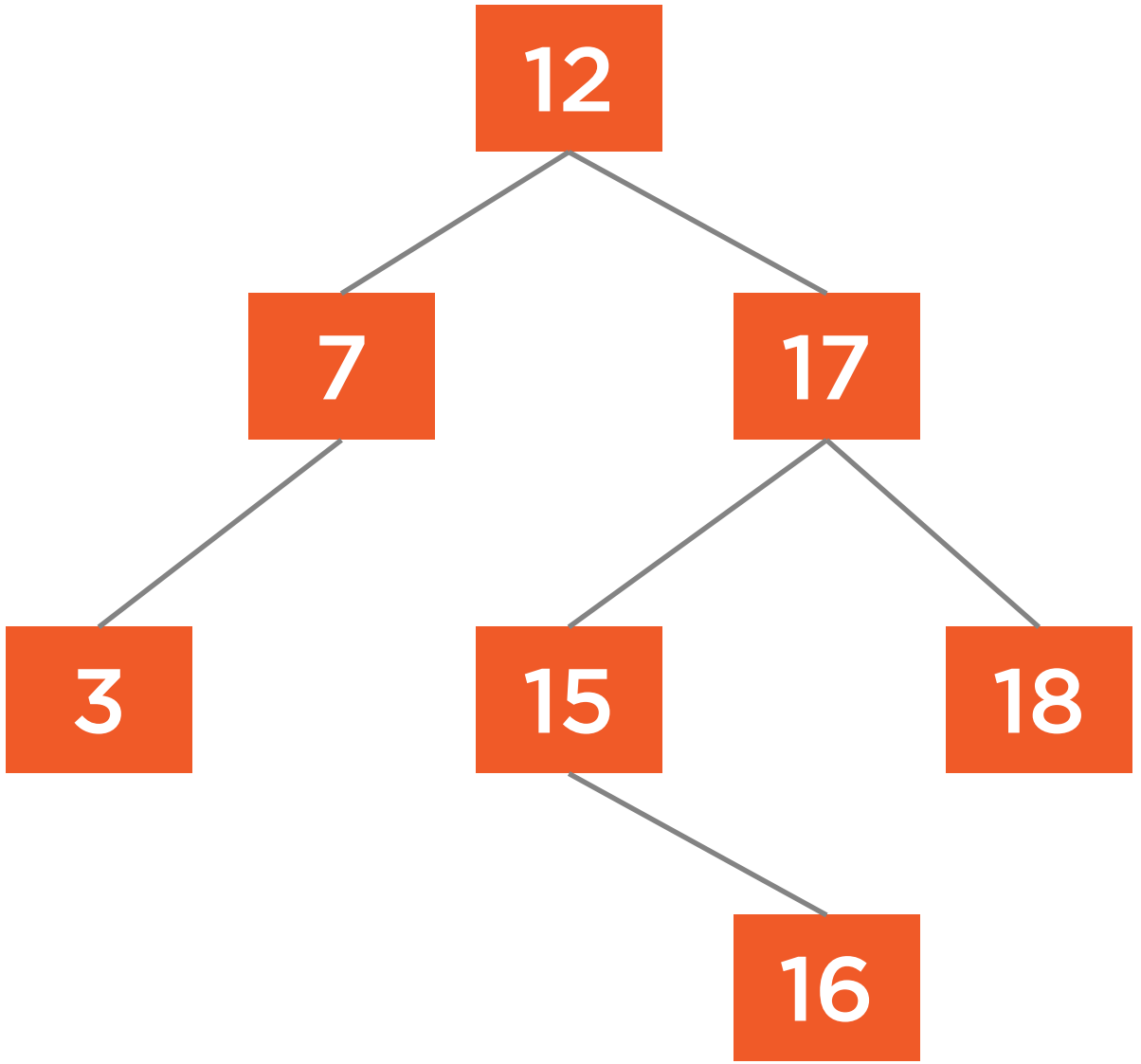
No Children
(Leaf node)



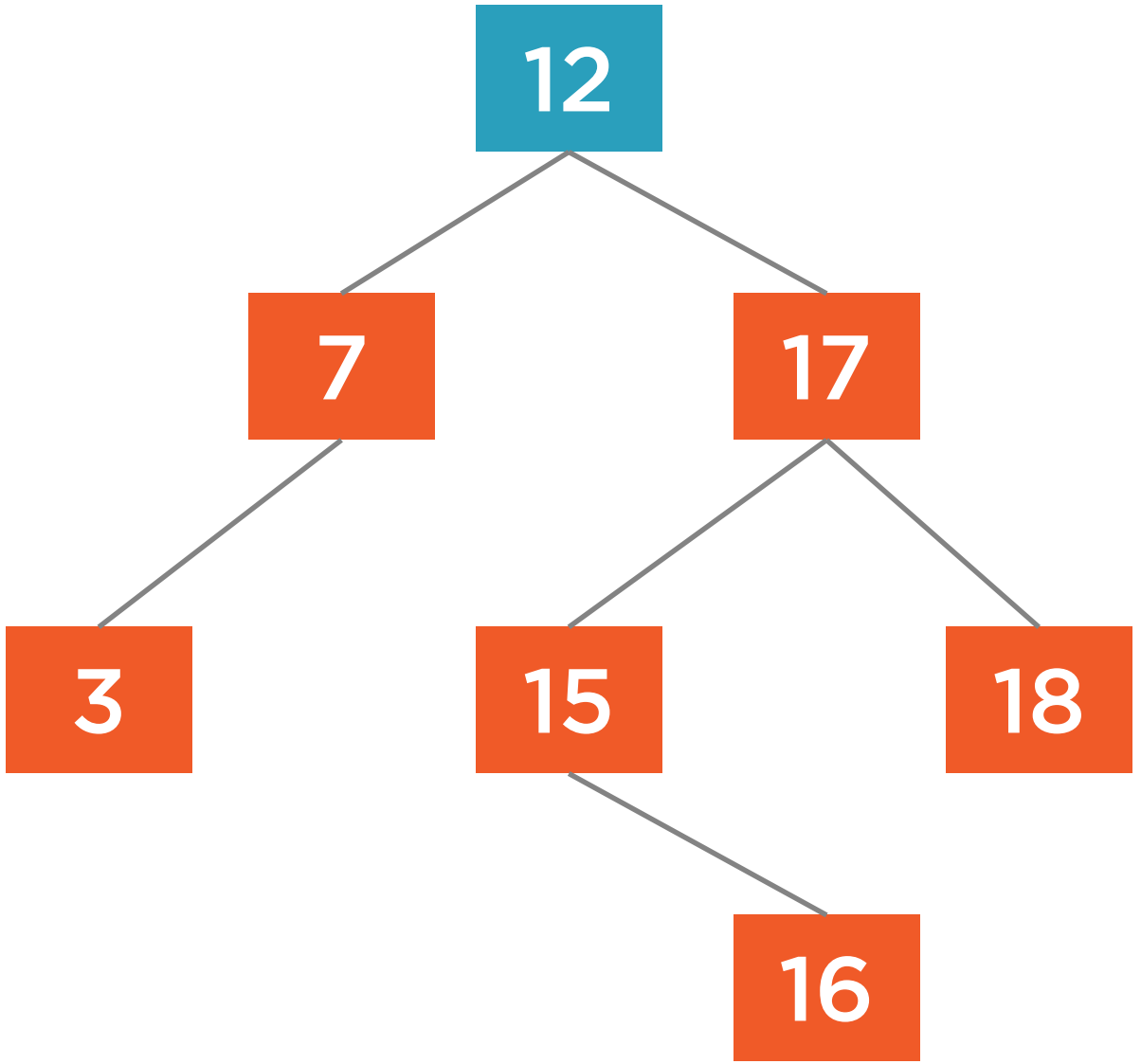
One Child



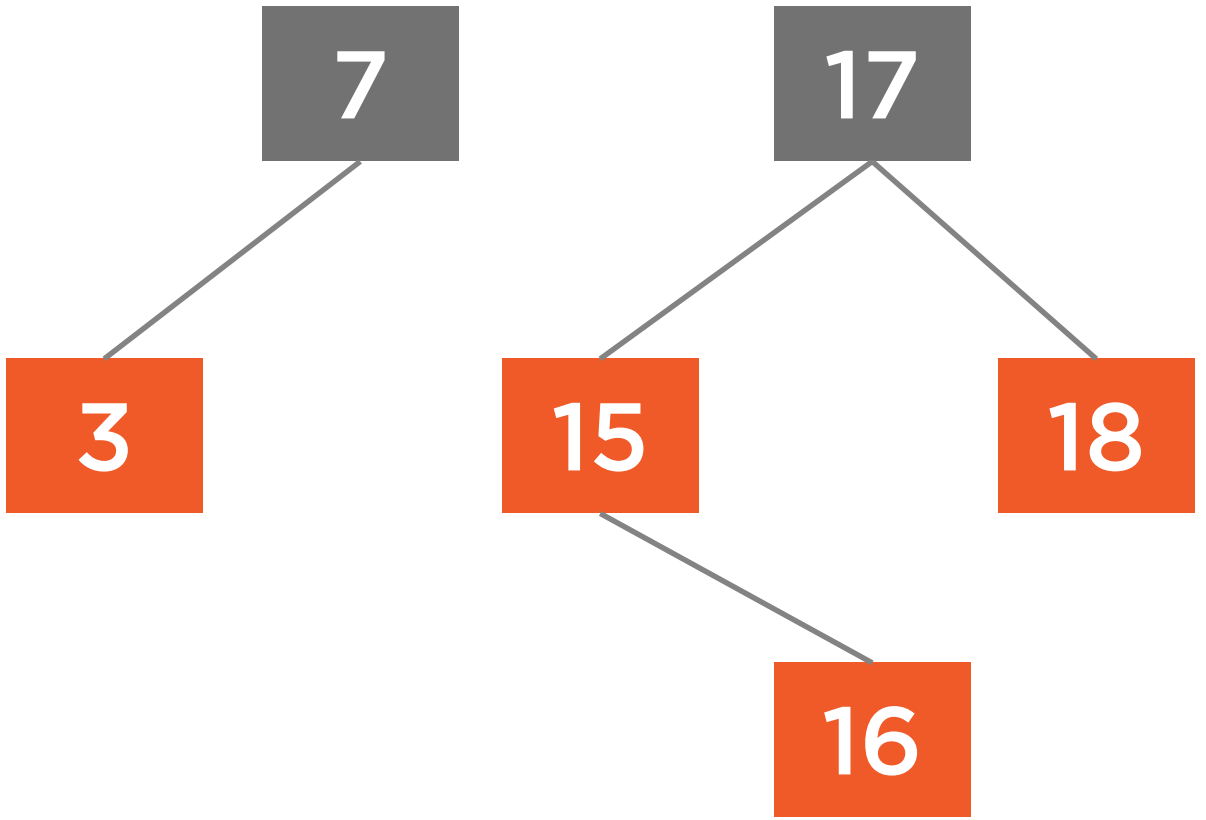
Two Children


12
7
17
3
15
18
16

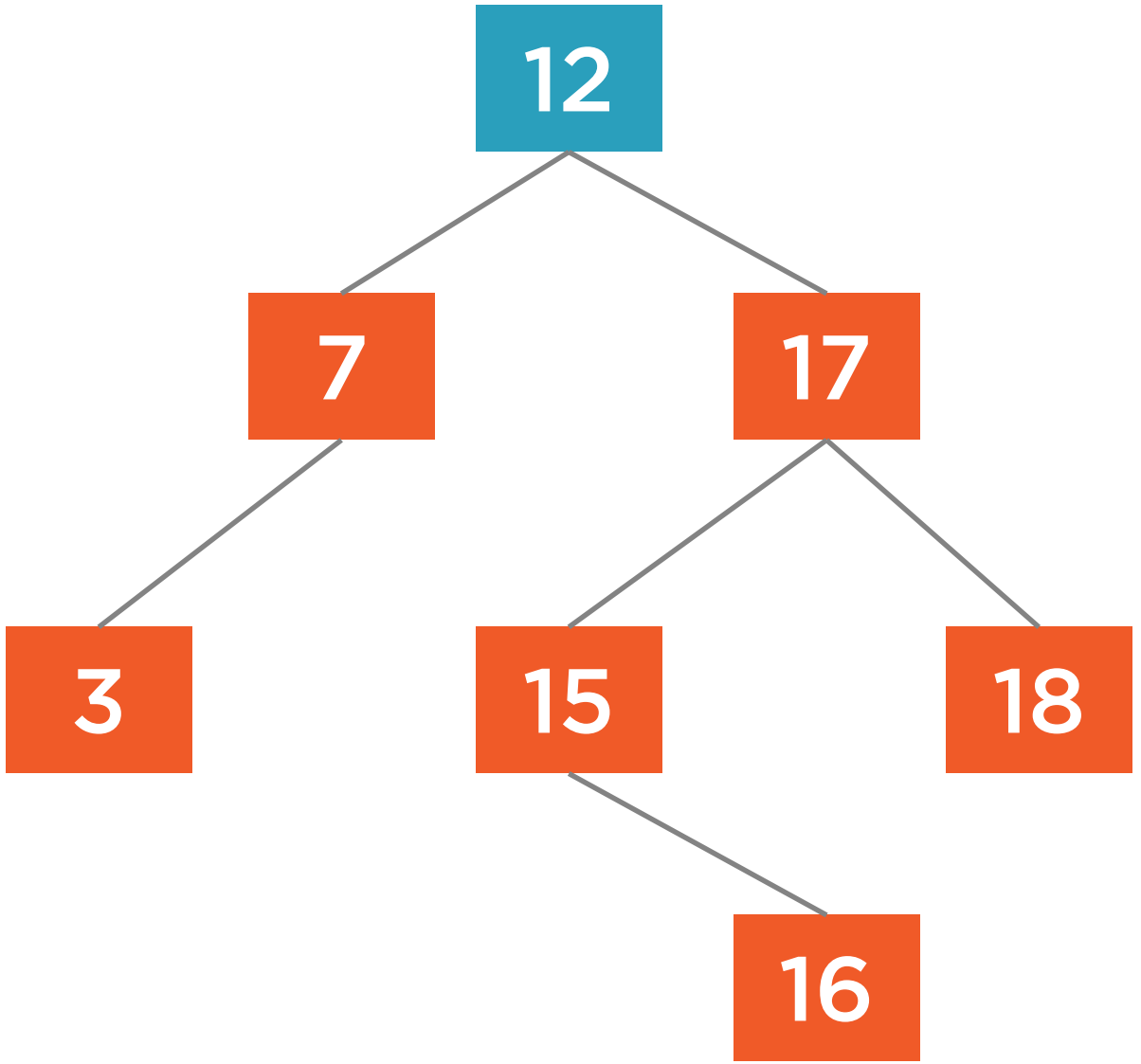
Find the node



Find the node

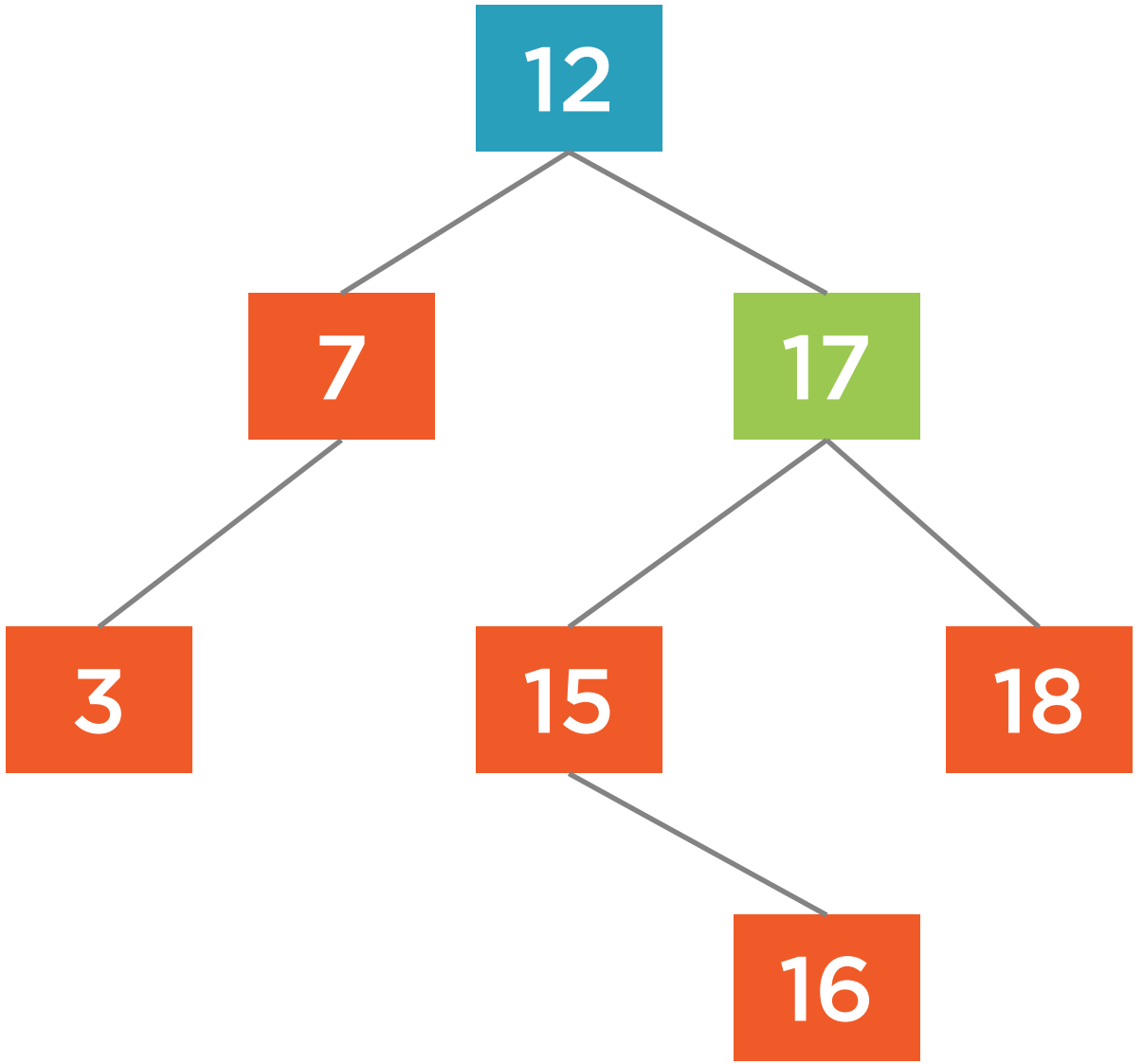


Find the node



Find the node

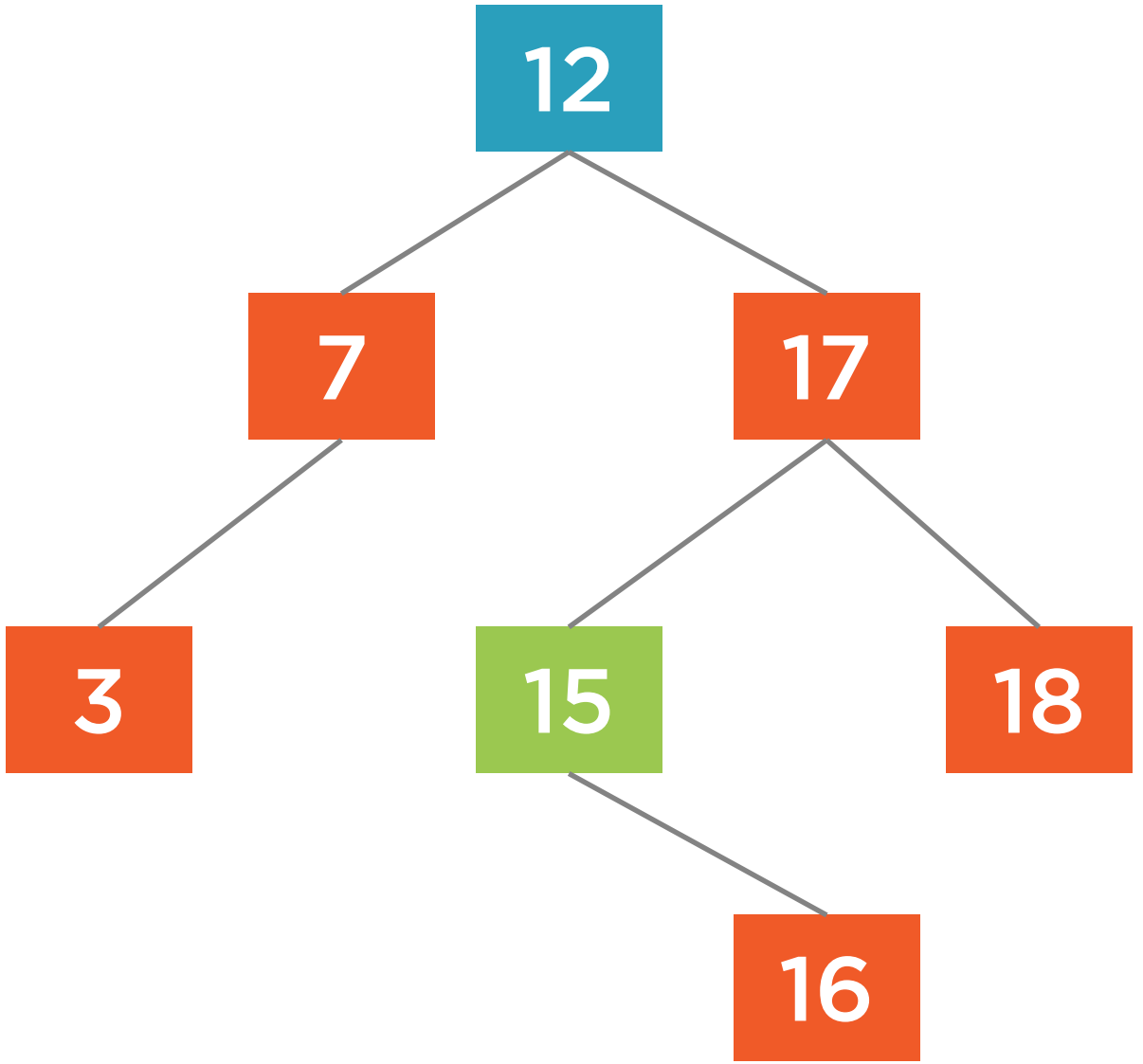
Go to the right child



Find the node

Go to the right child

Go to the left-most child

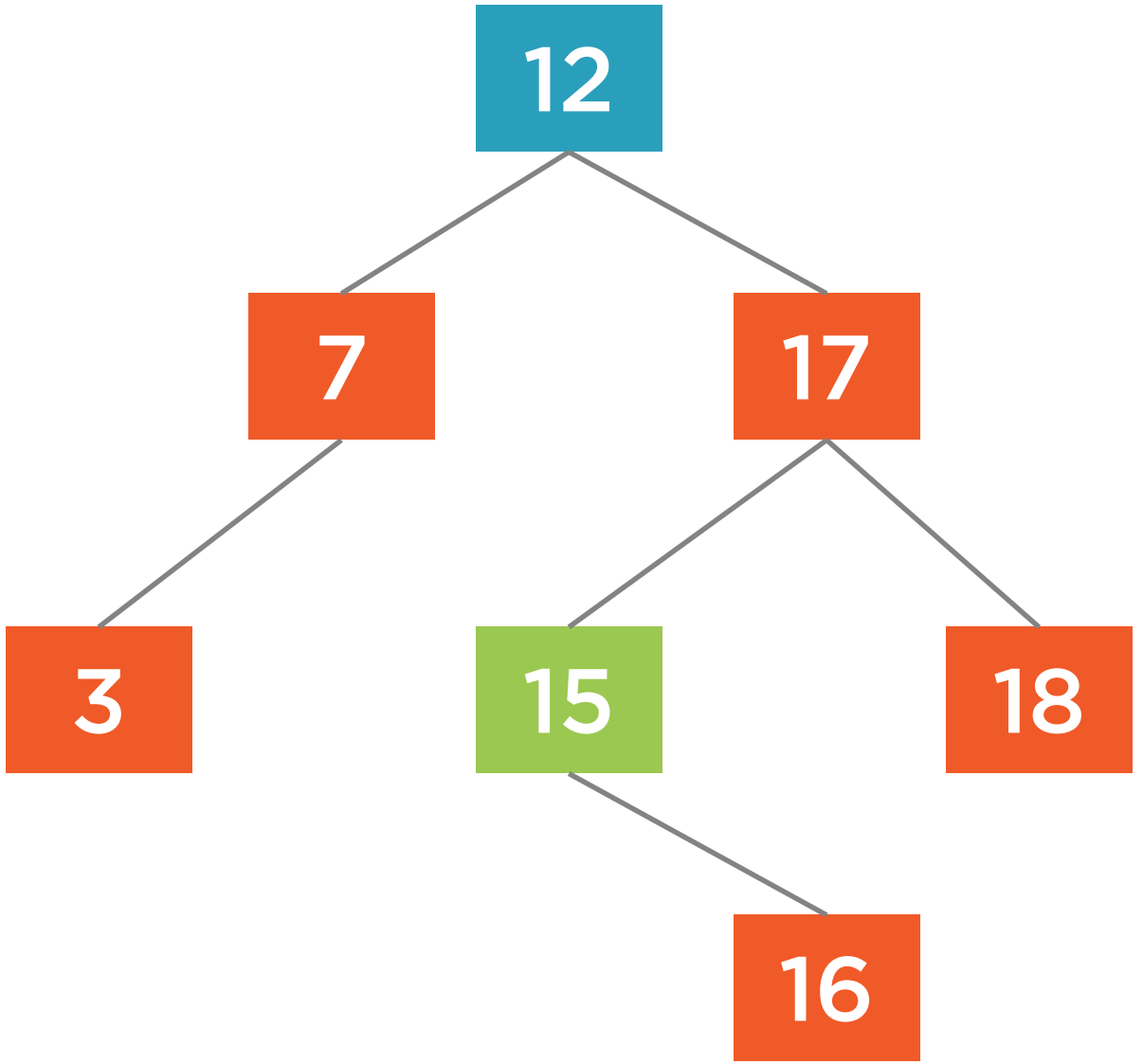


Find the node

Go to the right child

Go to the left-most child

Replace deleted node with successor



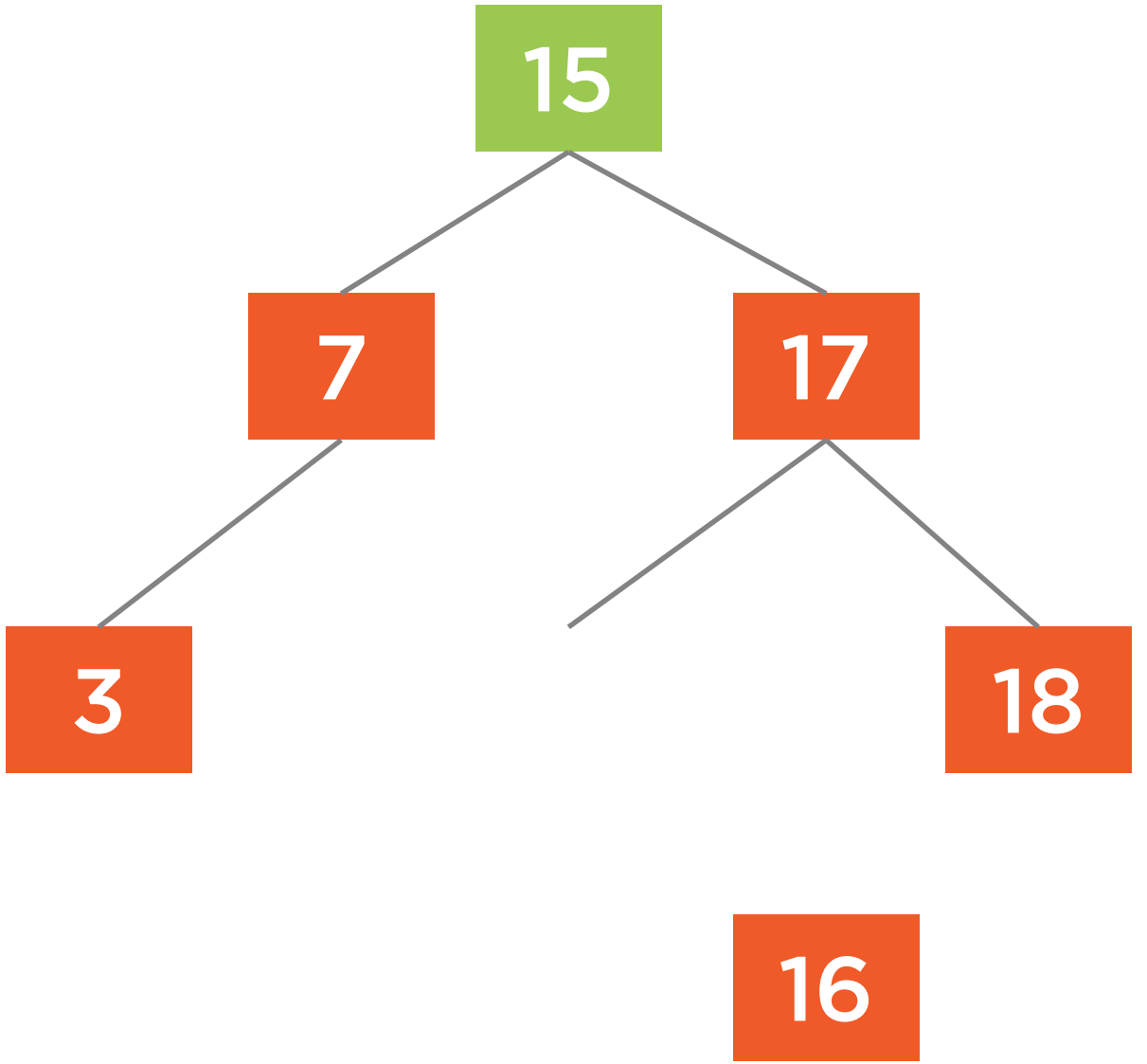
Find the node

Go to the right child

Go to the left-most child

Replace deleted node with successor

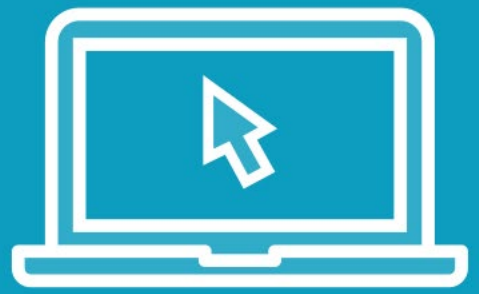
Move the successor's child up



Removal Complexity

Average case	Worst case
O(log n)	O(n)

Demo



Replace backing store with a binary search tree

- O(log n) operations
- Sorted output