

```
class Node:
    def __init__(self, key):
        self.val = val
```

```
def insert(self, key):
    if key < self.val:
        return self.val
```

```
else:
```

```
    root
```

```
    return Node(key, val)
```

```
root = Node(15)
```

```
root.insert(8)
```

```
root.insert(25)
```

```
root.insert(2)
```

```
root.insert(10)
```

```
root.insert(20)
```

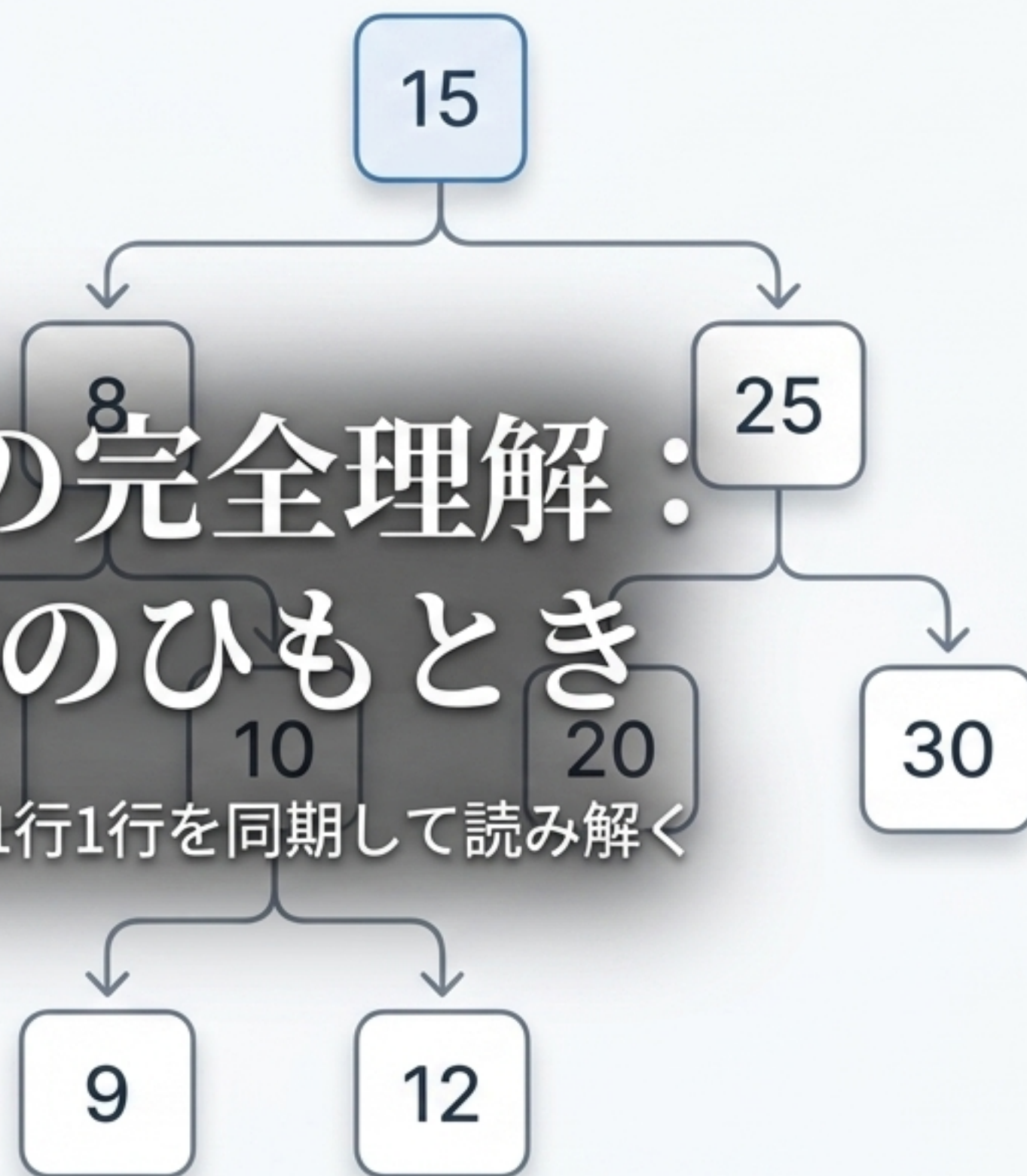
```
root.insert(30)
```

```
root.insert(9)
```

```
root.insert(12)
```

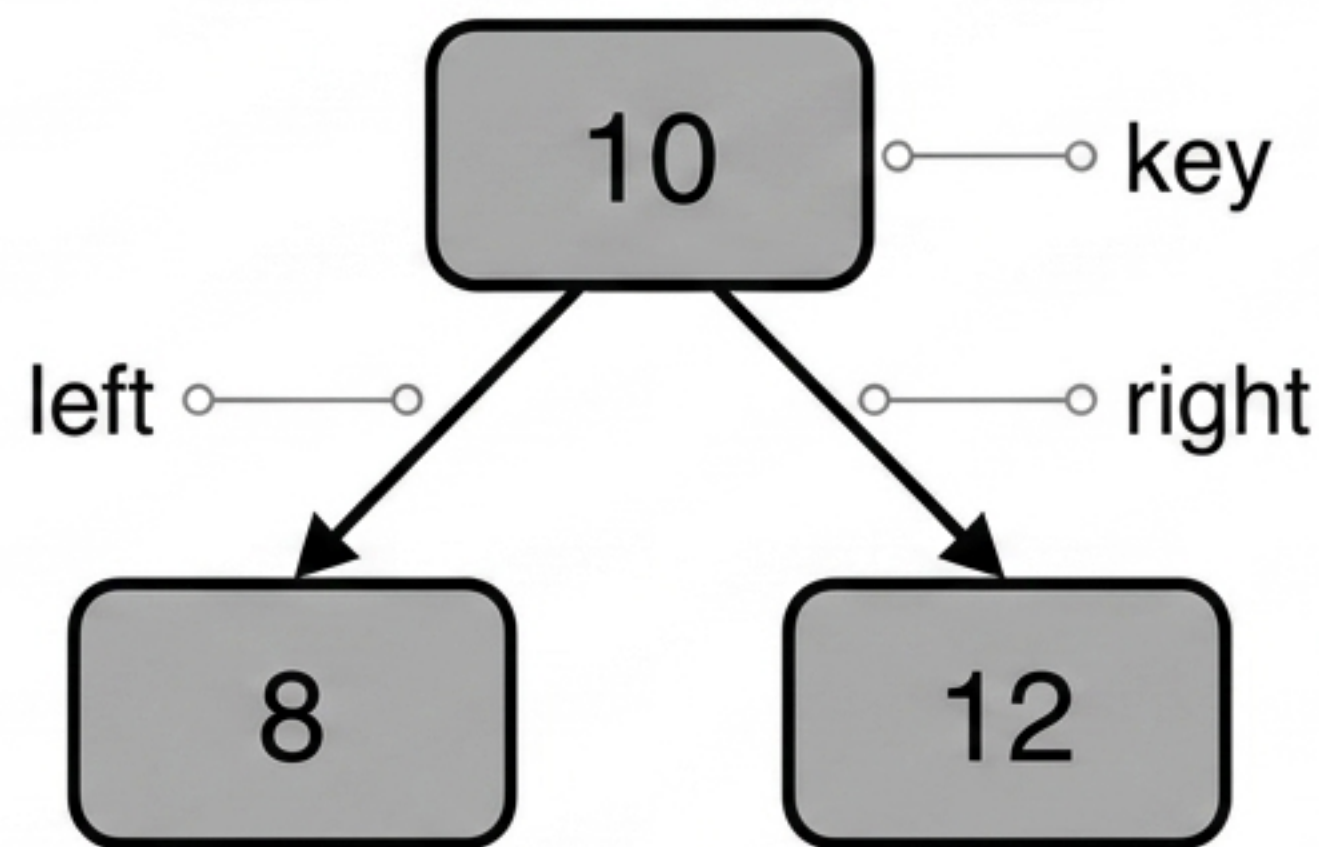
二分探索木 (BST) の完全理解 : 図解とPython実装のひもとき

アルゴリズムの視覚的メカニズムとコードの1行1行を同期して読み解く



黄金ルールと基本構造：「左は小さく、右は大きい」

```
class Node:
    def __init__(self, key):
        self.key = key      # ノードの値
        self.left = None    # 左の子への参照
        self.right = None   # 右の子への参照
```

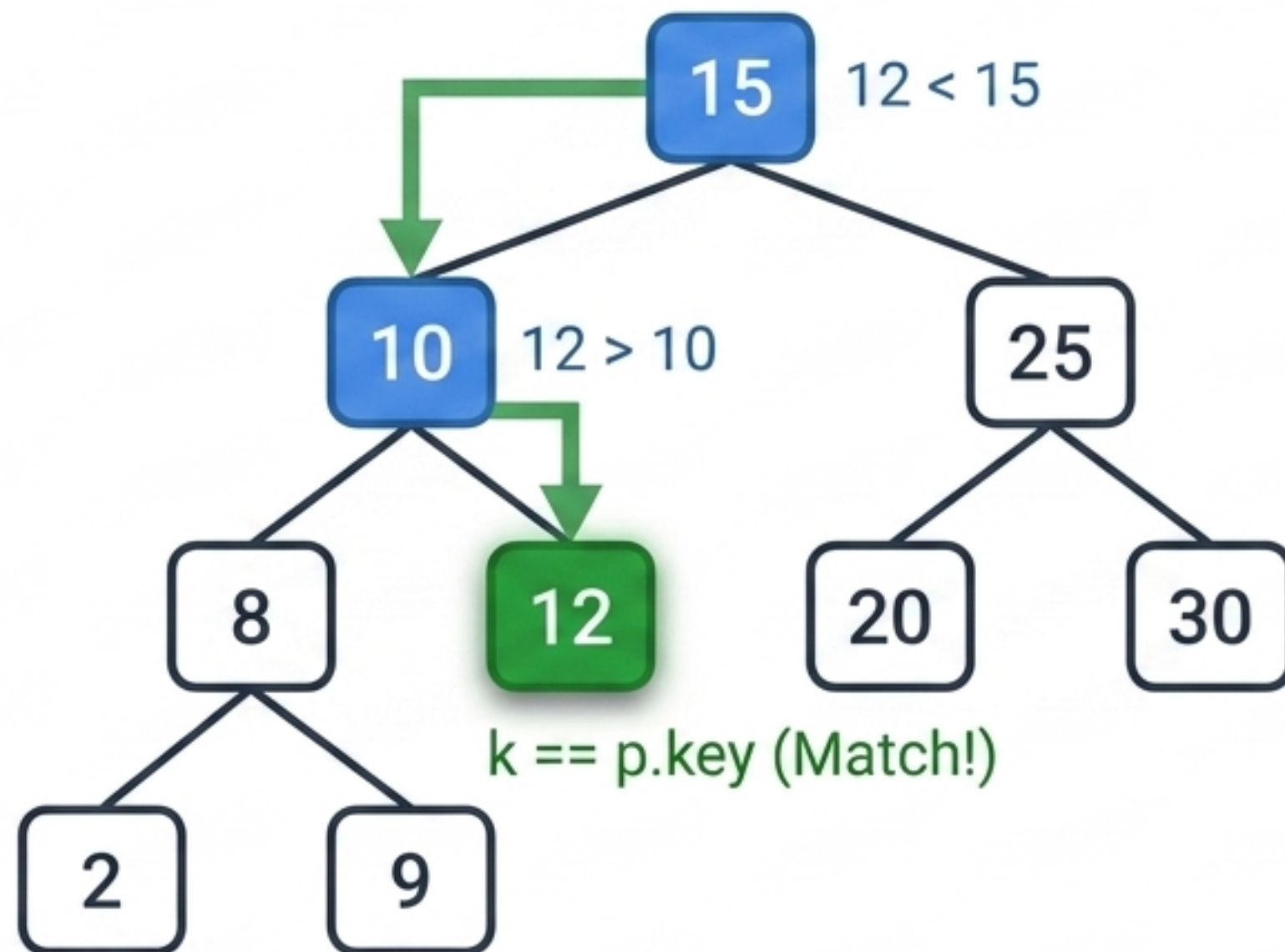


全てのノード N に対して、左部分木の全ノードは N より小さく、右部分木の全ノードは N より大きいという条件が成立する2分木。

探索 (Search) : 値の大小を比較し、木を下る

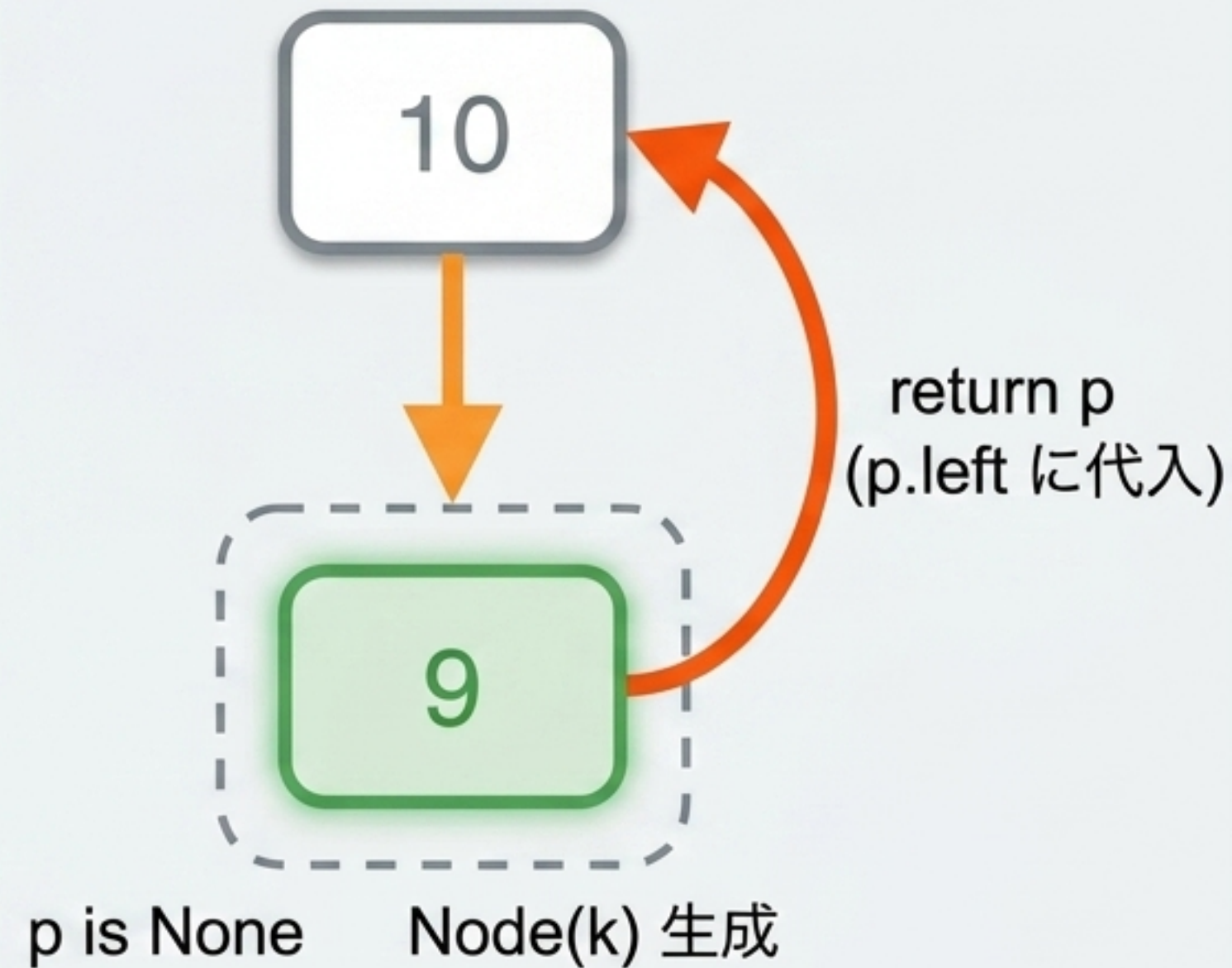
```
def search(k, p):  
    if p is None:  
        return None  
    elif k == p.key:  
        return p  
    elif k < p.key:  
        return search(k, p.left)  
    else:  
        return search(k, p.right)
```

Target: k = 12



挿入 (Insert) : 再帰を用いて新しい葉(Leaf)を正しい位置に繋ぐ

```
def addNode(k, p):  
    if p is None:  
        return Node(k)  
    elif k != p.key:  
        if k < p.key:  
            p.left = addNode(k, p.left)  
        else:  
            p.right = addNode(k, p.right)  
    return p
```



削除 (Delete) のメカニズム：子ノードの数に応じた3つのケース

子がない場合
(Case 1)



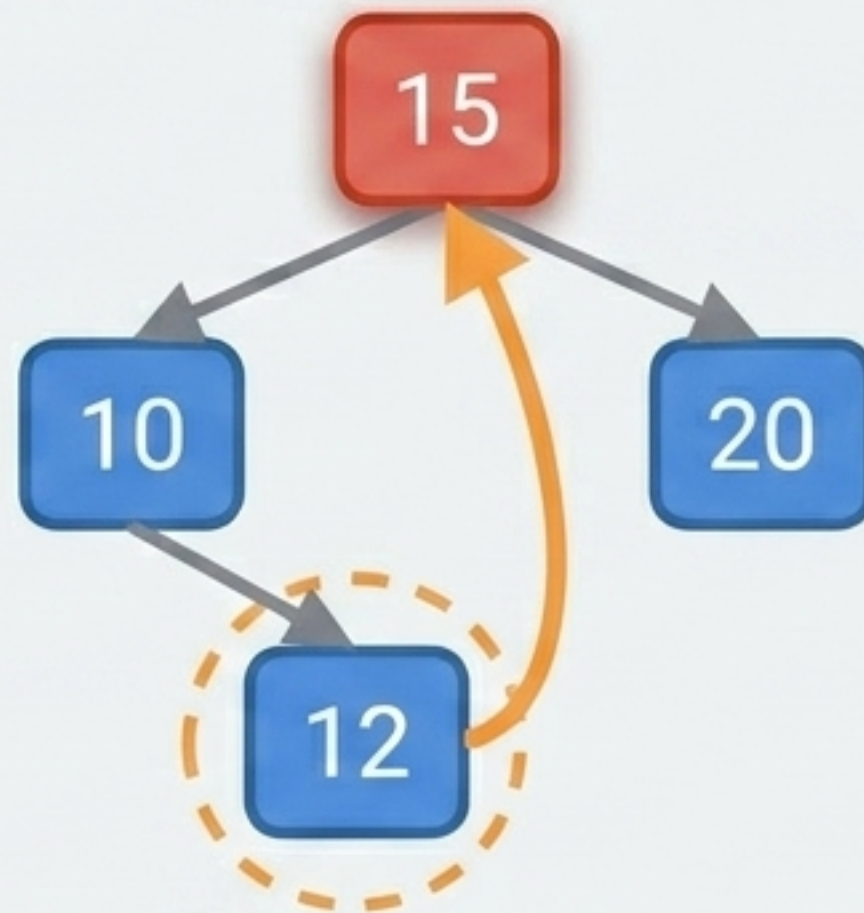
そのままノードを削除する。

子が1つの場合
(Case 2)



削除するノードの位置に、
その唯一の子ノードを昇格
させて置きき換える。

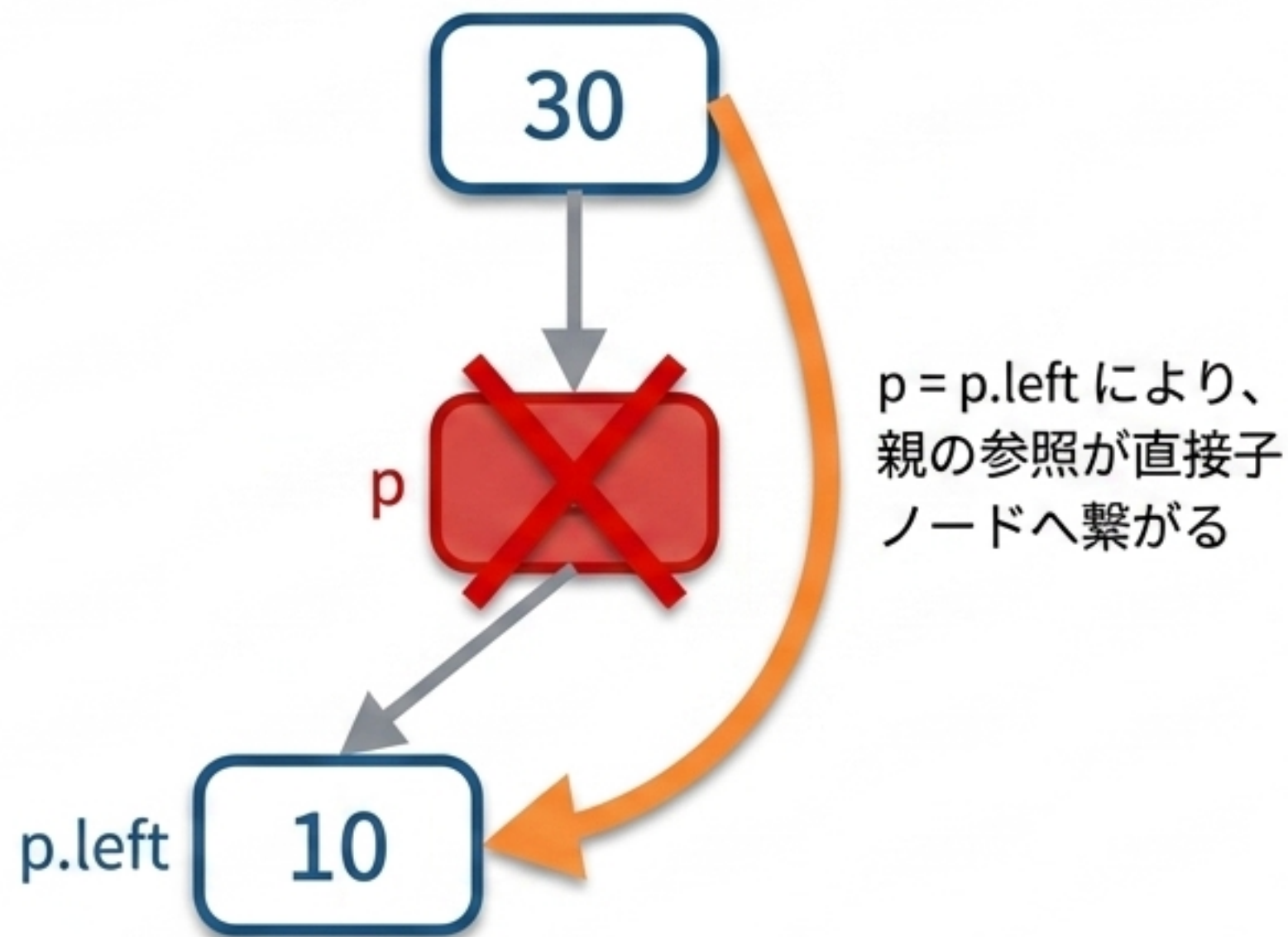
子が2つの場合
(Case 3)



左右の構造を保つため、
「左部分木の最大ノード」を
取り出して削除ノードの位
置に据える。

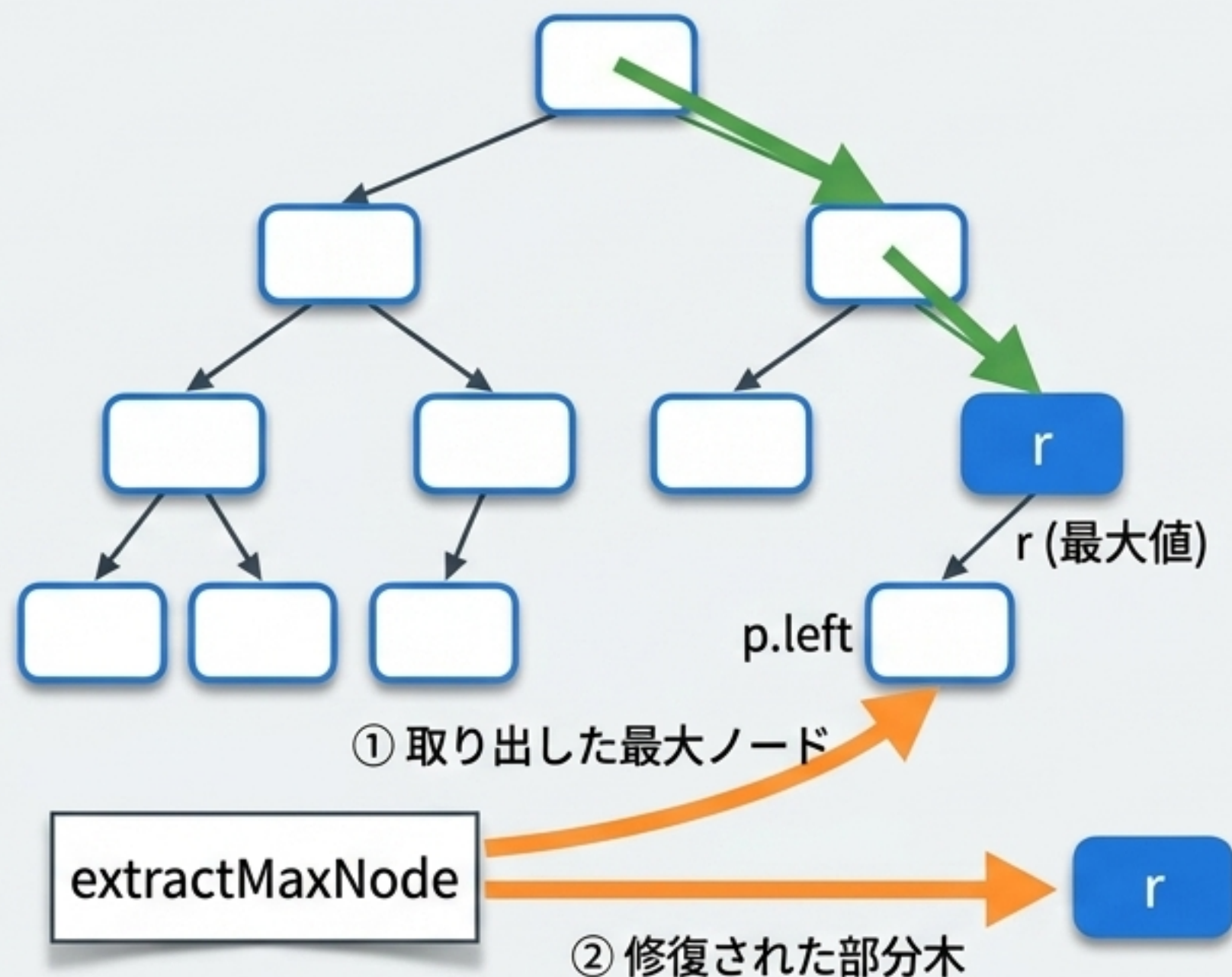
削除の実装 (Case 1 & 2) : ノードのバイパスと昇格

```
if p.left is None and p.right is None:  
    p = None    # 葉ノードならそのまま削除  
elif p.left is None:  
    p = p.right # 右の子を昇格させる  
elif p.right is None:  
    p = p.left  # 左の子を昇格させる
```



削除 Case 3の準備：「左部分木の最大値」を抽出する

```
1 def extractMaxNode(p):  
2     if p.right is None:  
3         return p.left, p  
4     else:  
5         p.right, r = extractMaxNode(p.right)  
6         return p, r  
7
```

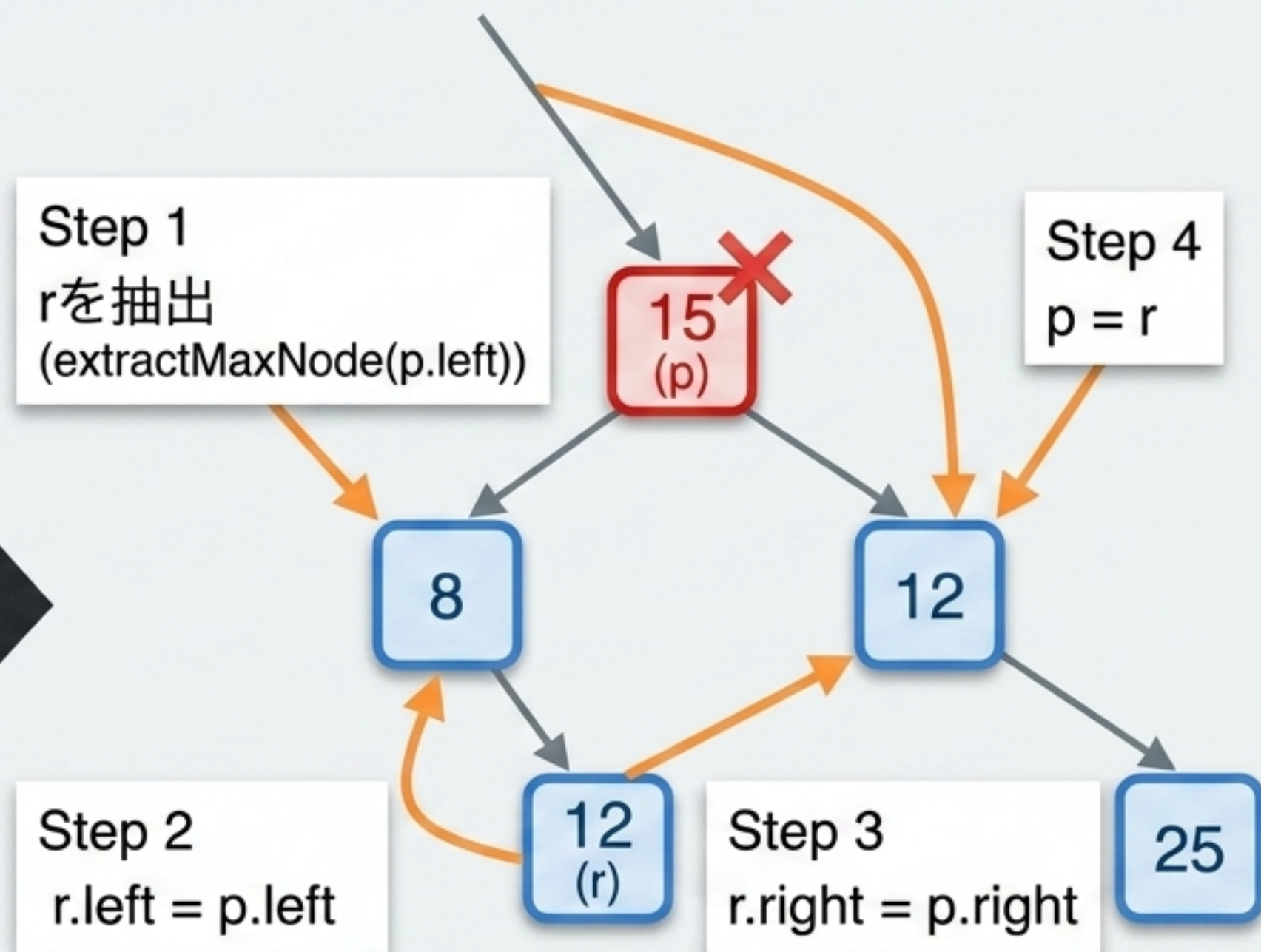


Python特有のタプル返却を利用し、①取り出した最大ノードと②修復された部分木の2つを同時に返す。

削除の実装 (Case 3) : ポインタの再配線によるノードのすげ替え

else:

```
# 左部分木の中の最大ノードを置く  
p.left, r = extractMaxNode(p.left)  
r.left = p.left  
r.right = p.right  
p = r
```



動作シミュレーション (1/3) : 挿入による木の構築

入力値列:

5

2

7

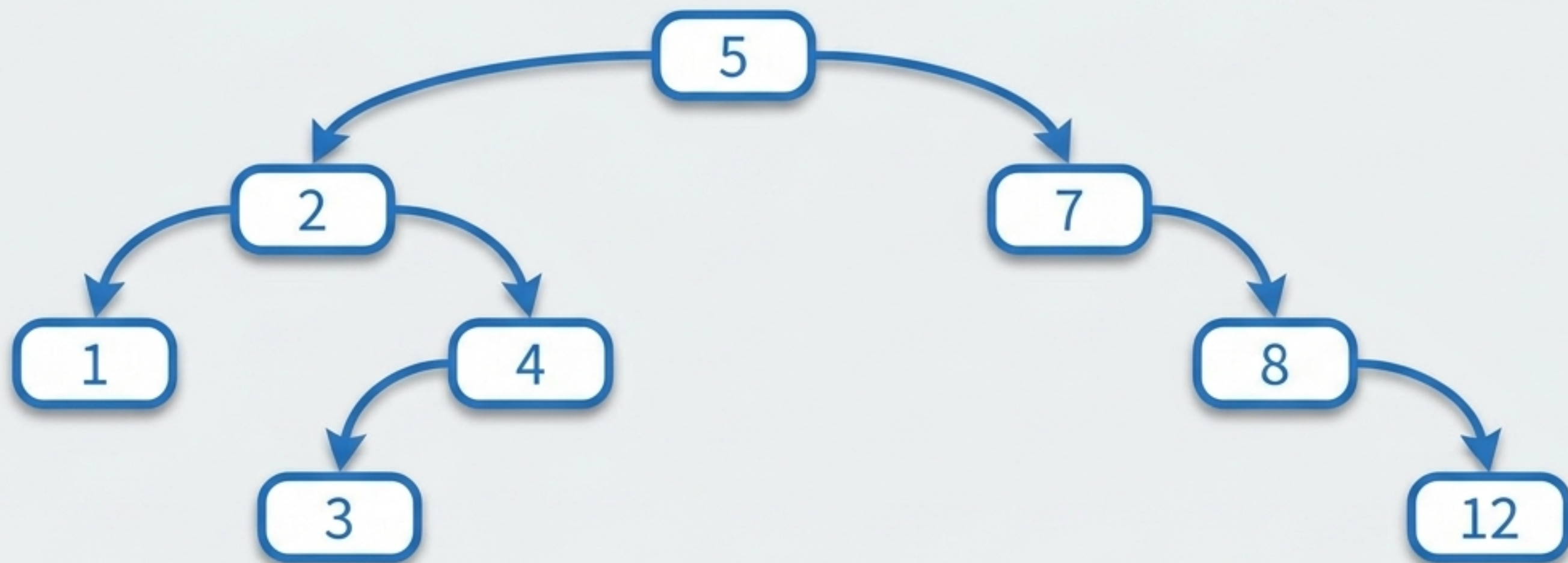
1

8

4

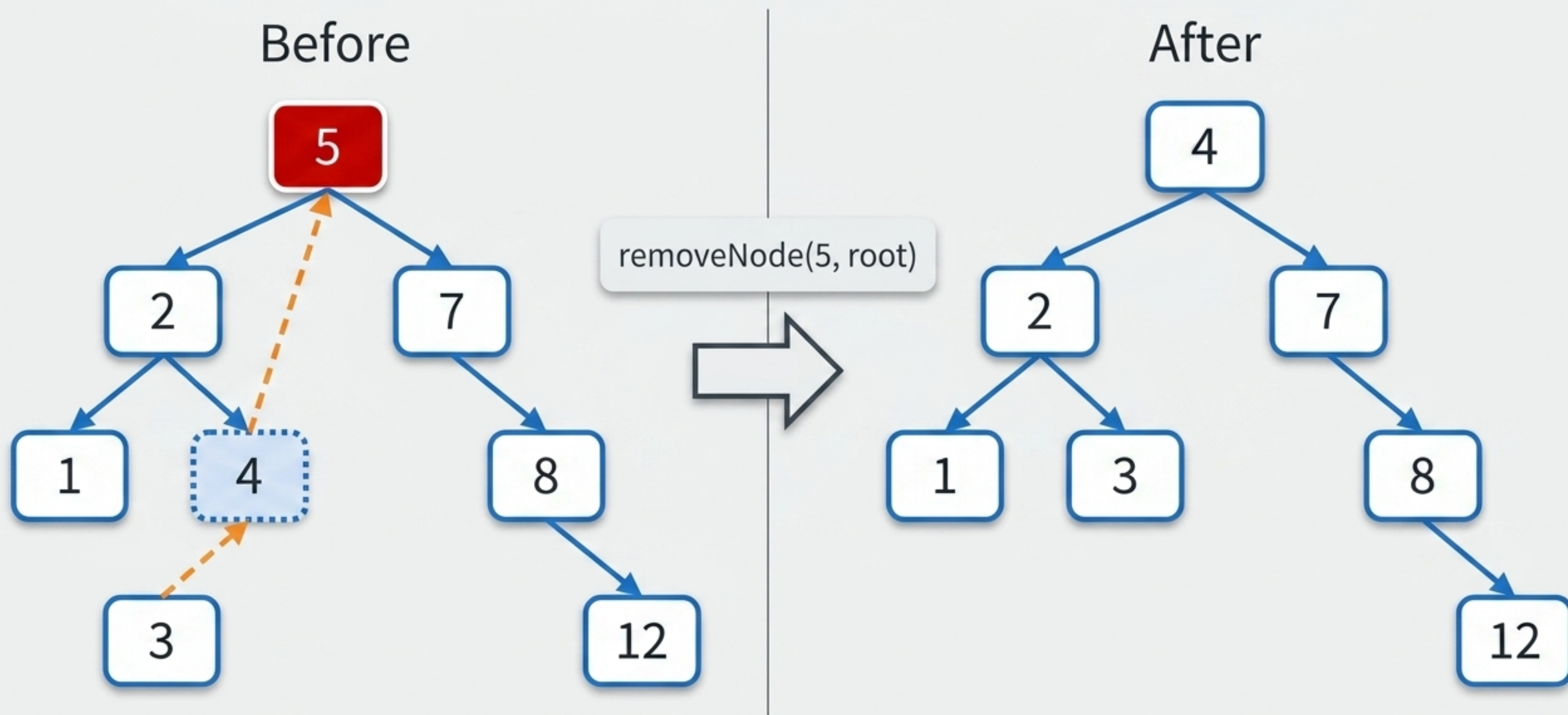
3

12



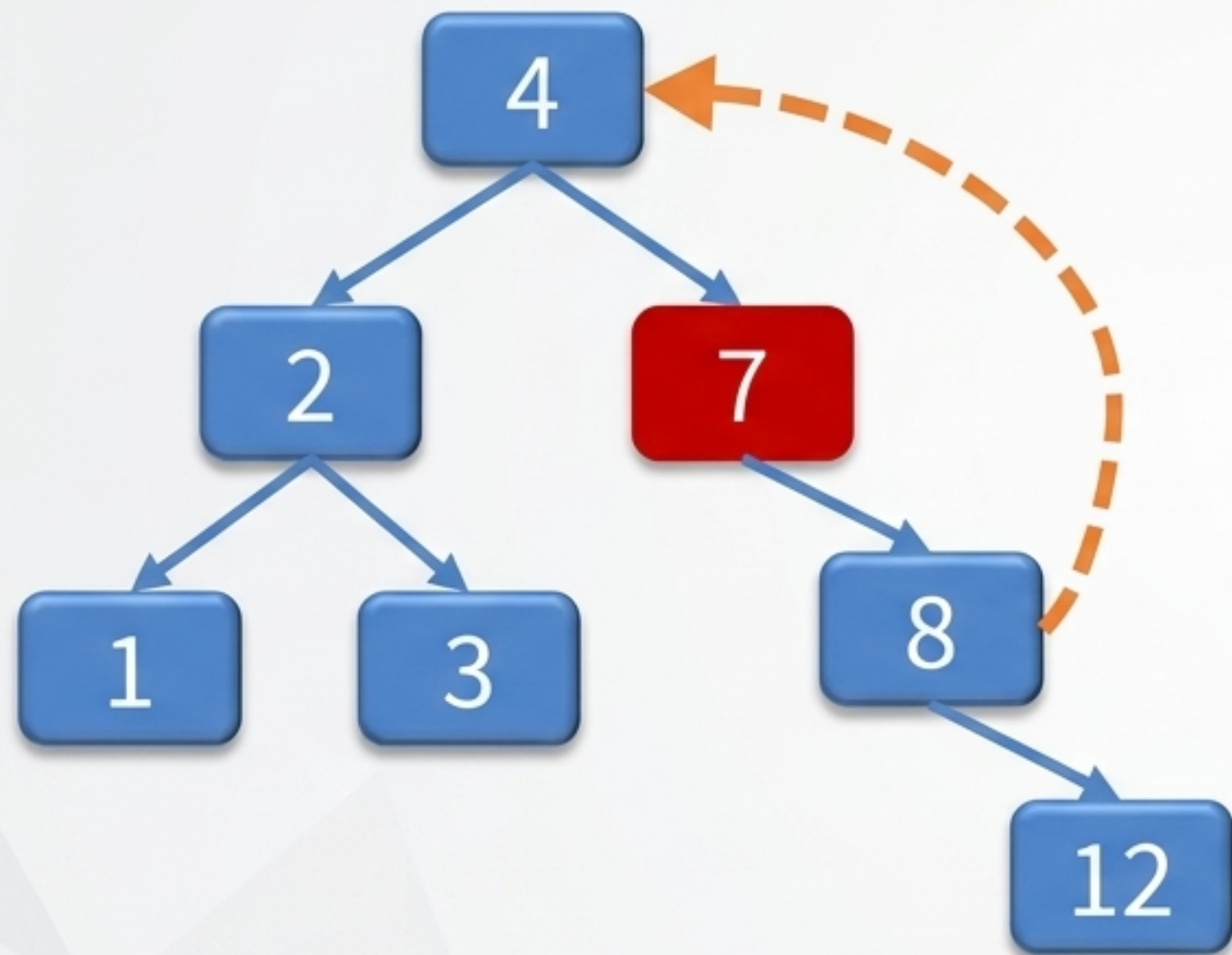
空の木から開始し、8つの値を順に addNode で挿入した完成形。

動作シミュレーション (2/3) : 根ノード(5)の削除

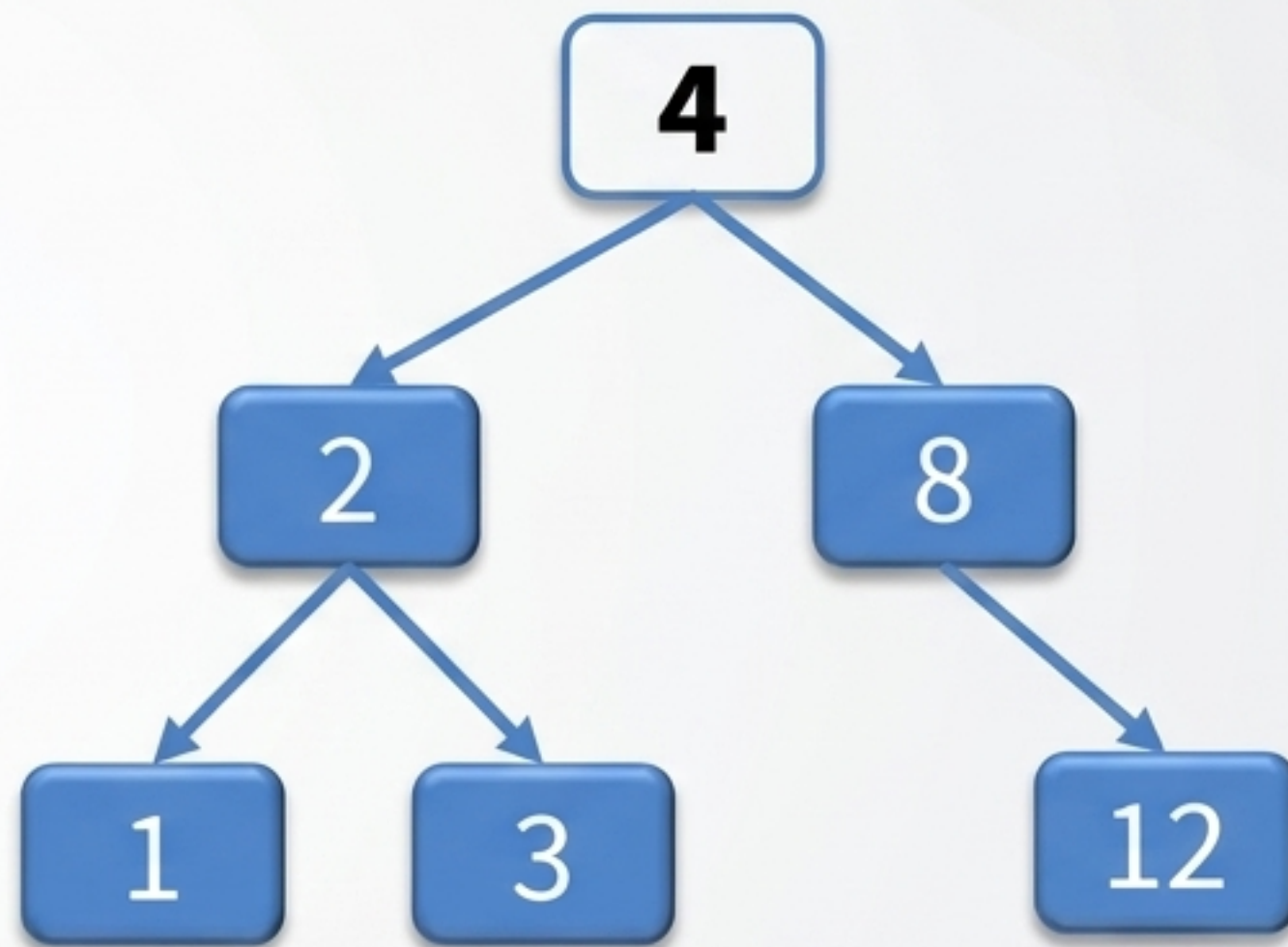


動作シミュレーション (3/3) : 内部ノード(7)の削除と最終形

Before



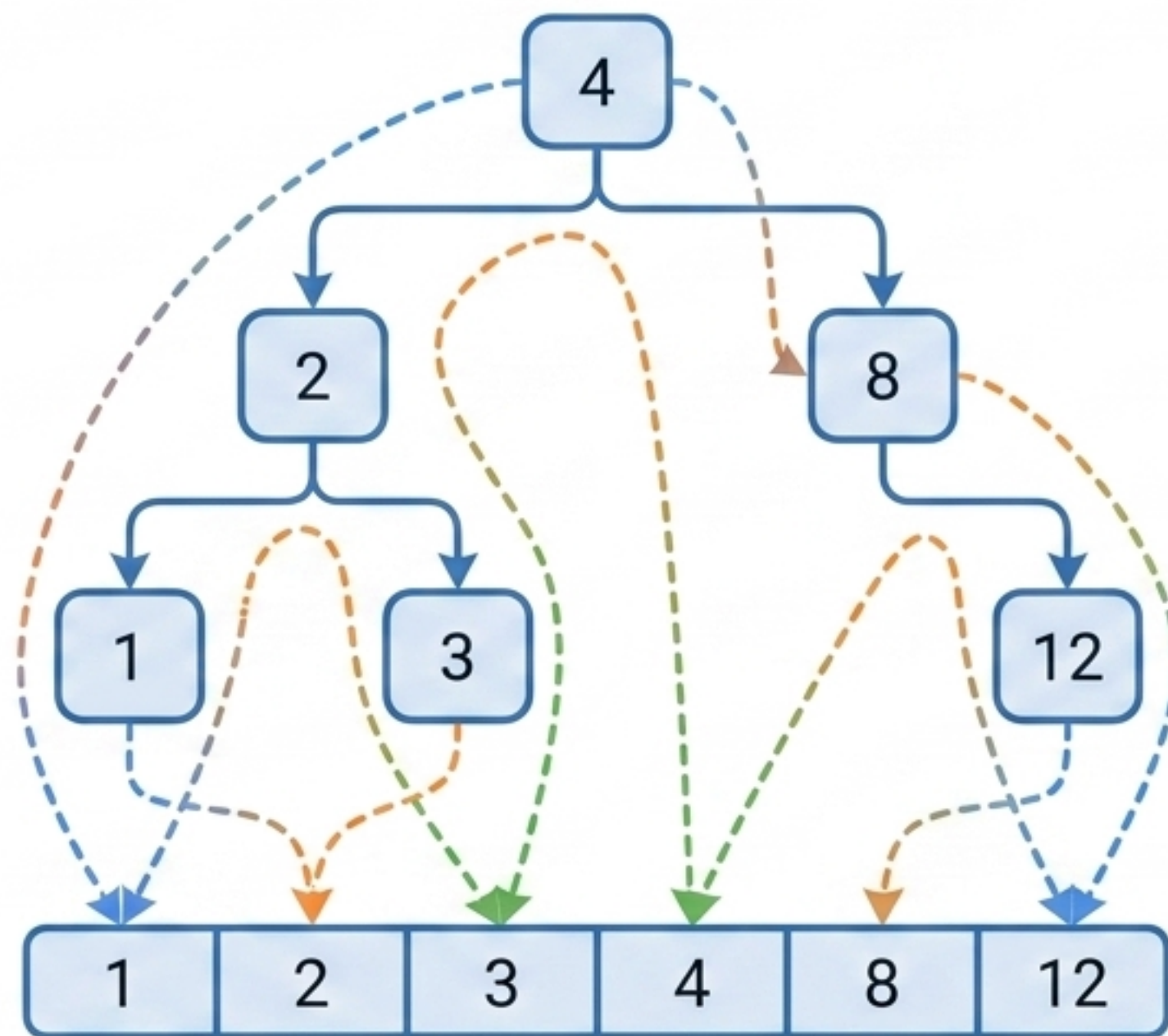
After



「設問4」の最終回答。最終的な根(Root)ノードは『4』となる。

中間順巡回 (In-order Traversal) : データを昇順で取り出す

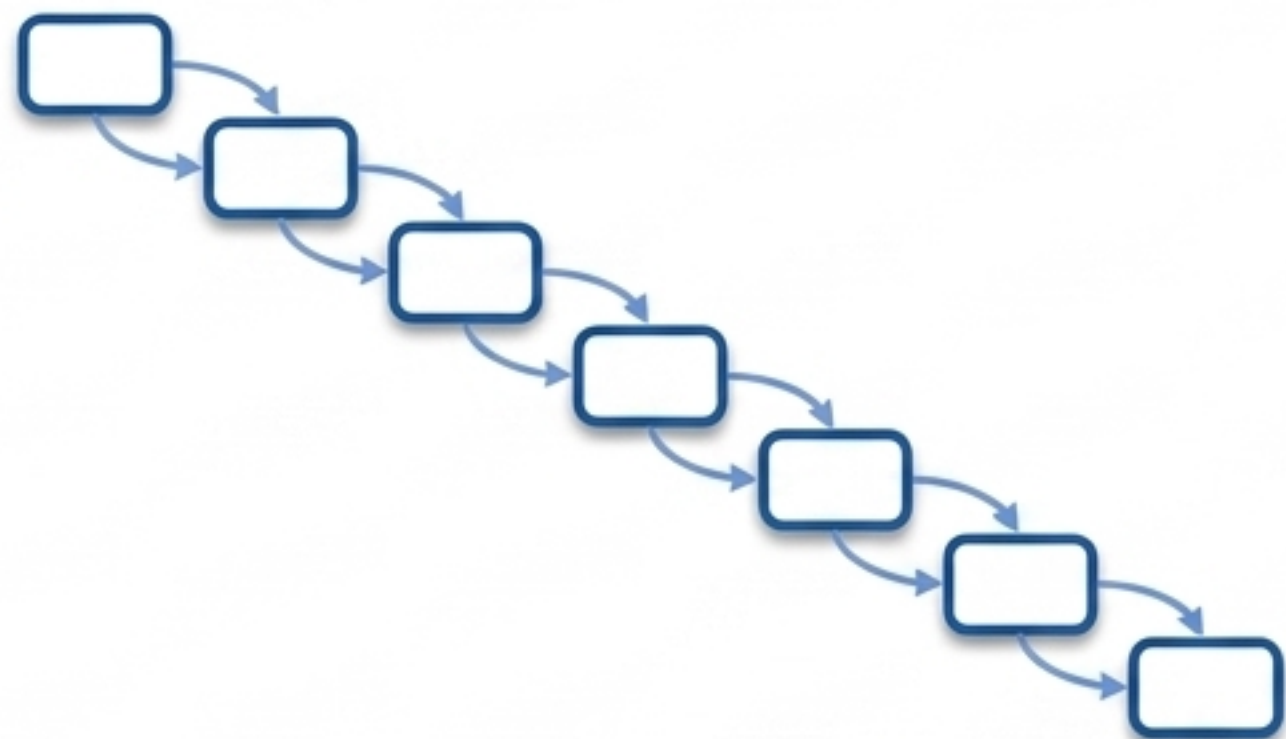
```
def inorder_traversal(node):  
    if node is not None:  
        inorder_traversal(node.left) # 左を先に処理  
        print(node.key)              # 自分を表示  
        inorder_traversal(node.right) # 右を後に処理
```



「左・自分・右」の順で再帰的に処理することで、二分探索木から自動的にソート（昇順）されたデータが得られる。

計算量 (Time Complexity) : 木の「高さ」がパフォーマンスを決める

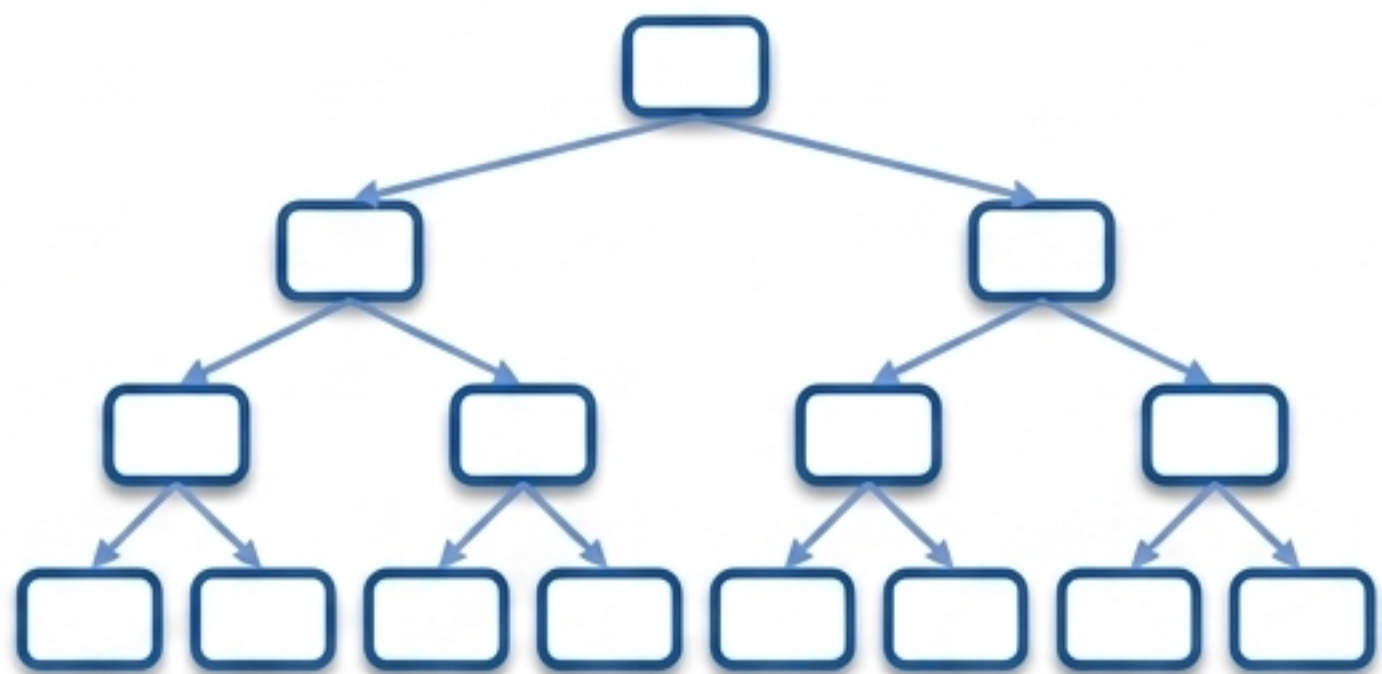
Worst Case (最悪ケース)



偏った木 (例: ソート済みデータを順に挿入した場合)

計算量: $O(n)$
(ノード数に比例)

Ideal Case (理想ケース)



完全2分木 (探索範囲を常に1/2に狭められる状態)

計算量: $O(\log n)$
(木の高さに依存)

二分探索木の真価を発揮するには、木が平衡 (バランス) を保つことが極めて重要となる。