

抽象的なアルゴリズムを実用コードに翻訳する: Pythonで解き明かす「ヒープ構築」の完全解剖

平成20年度 秋期 ソフトウェア開発技術者 午後 I 問5 徹底解説

```
def heap_correct(A, i, length):  
    largest = i  
    l_child = 2 * i + 1  
    r_child = 2 * i + 2  
    ...
```

- ベースソース: 擬似コードからPython (0-index) への実践的マッピング
- フォーカス: 複雑な再帰論理の視覚化と計算量証明

二分木を「ヒープ」へと昇華させる2つの絶対条件

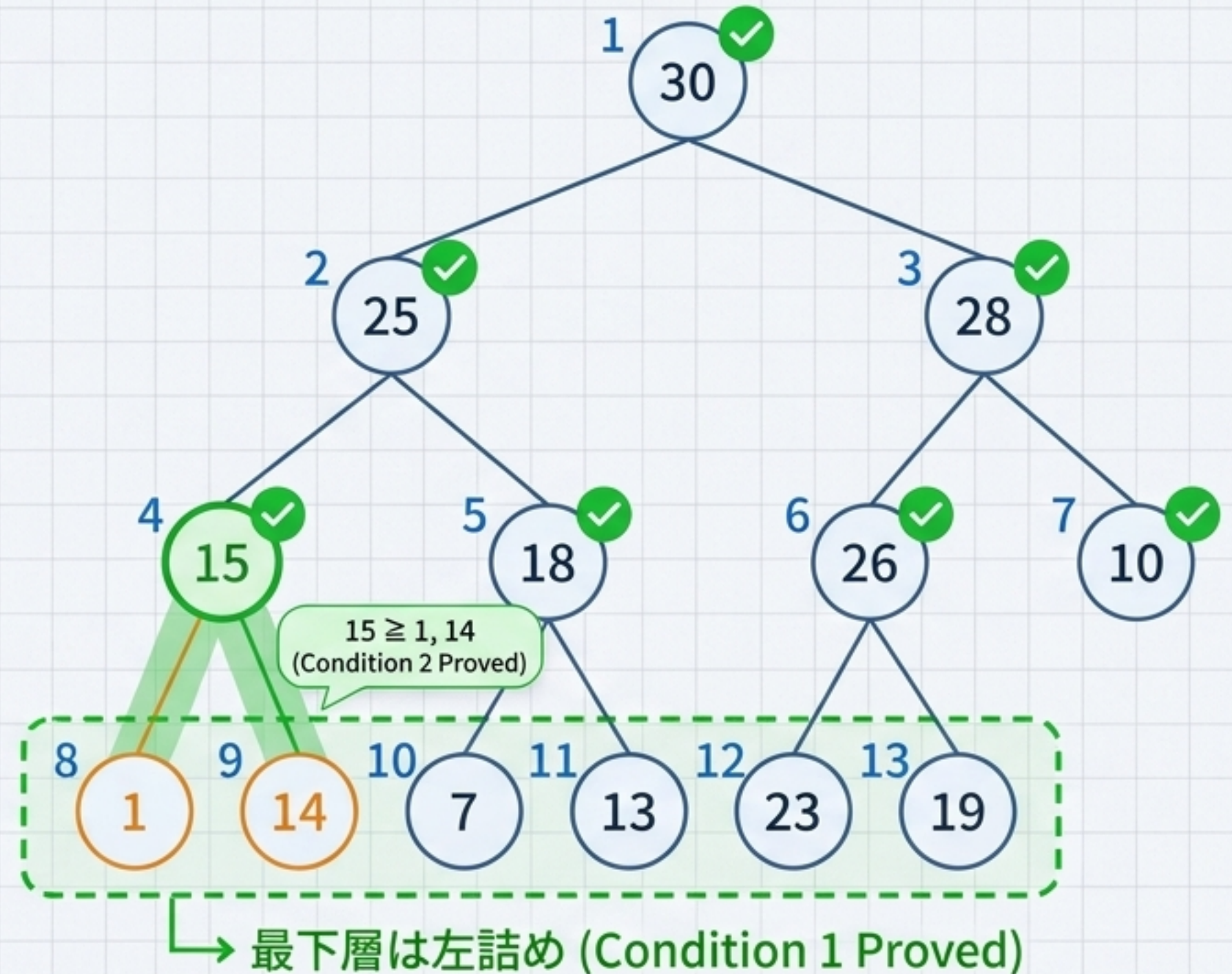
条件1 (形状)

葉の深さは最大 d または $d-1$ のみ。かつ、最下層の葉は必ず「左詰め」で配置されること。

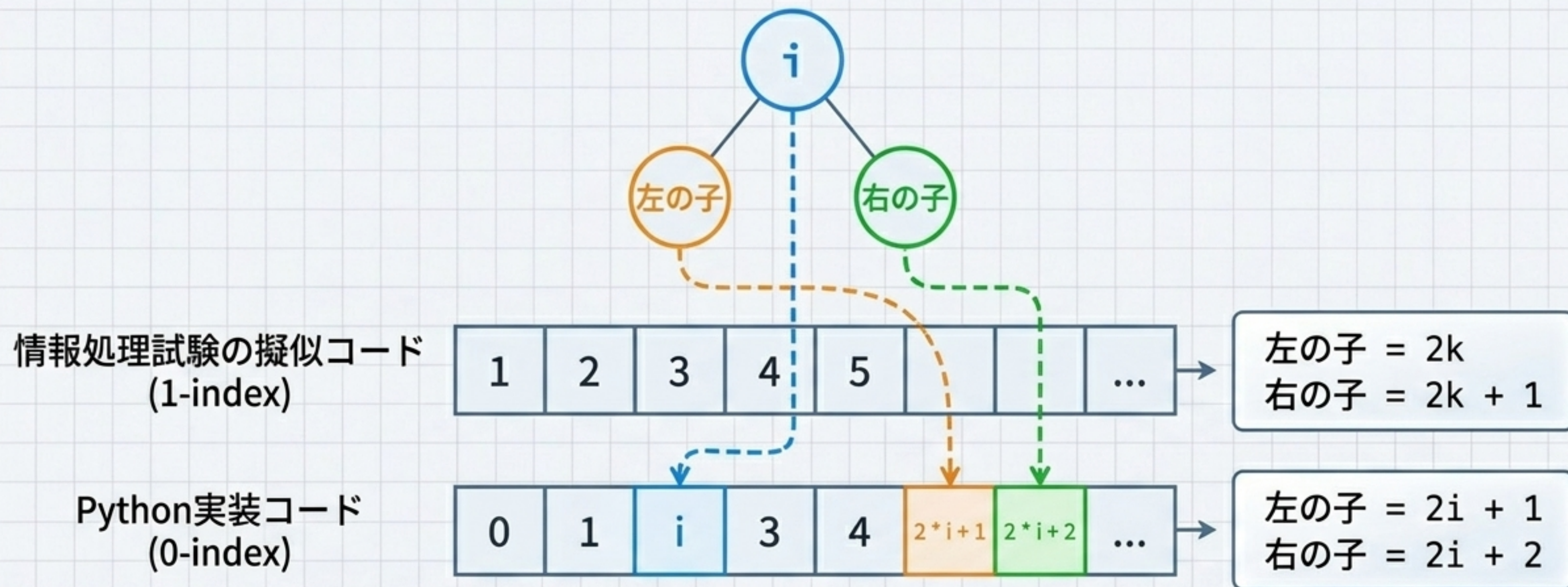
条件2 (順序)

すべての親ノードの値は、その子ノードの値よりも「大きい、または等しい」こと。
(親 $\geq$ 子)

Insight: この2つのルールを守るためのエンジンが `heap_correct`



データ構造の変換ギャップ：木構造と配列を結ぶインデックスの数式

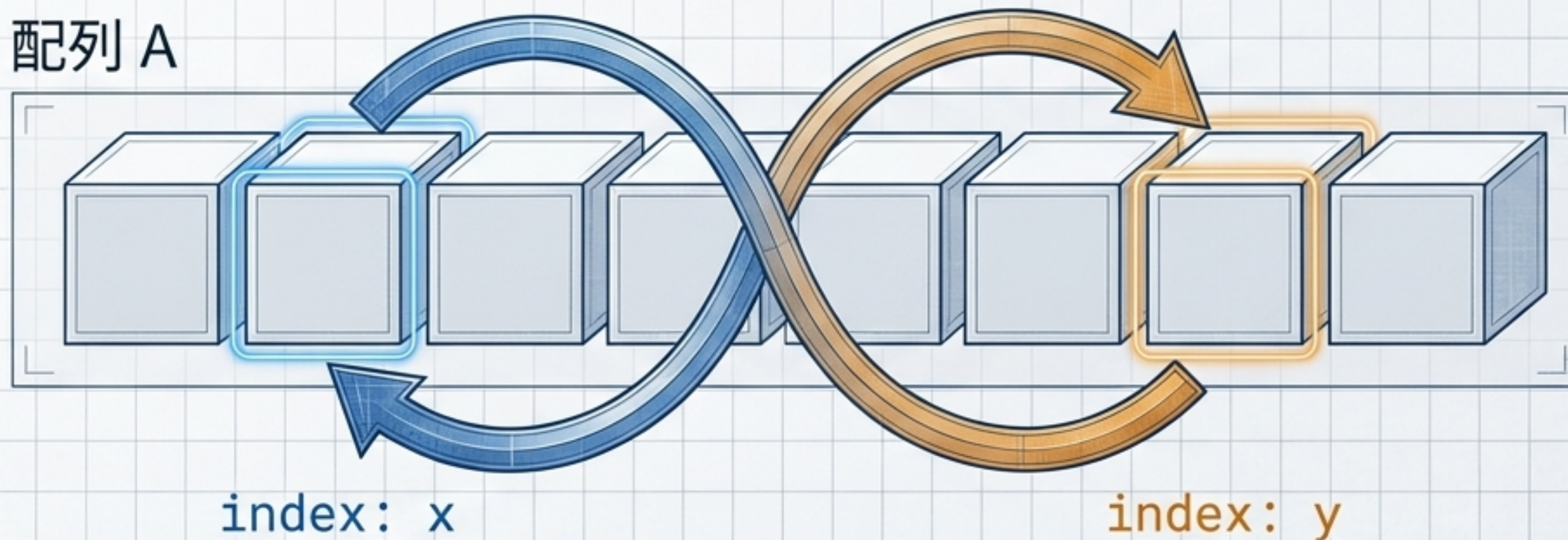


Takeaway: 木構造の「ノードの親子関係」は、配列の「インデックス計算式」として完全にマッピング可能。Pythonコードはすべてこの数式で駆動する。

Code Block 1: 配列操作の最小単位 exchange

```
def exchange(A, x, y):  
    A[x], A[y] = A[y], A[x]
```

配列 A

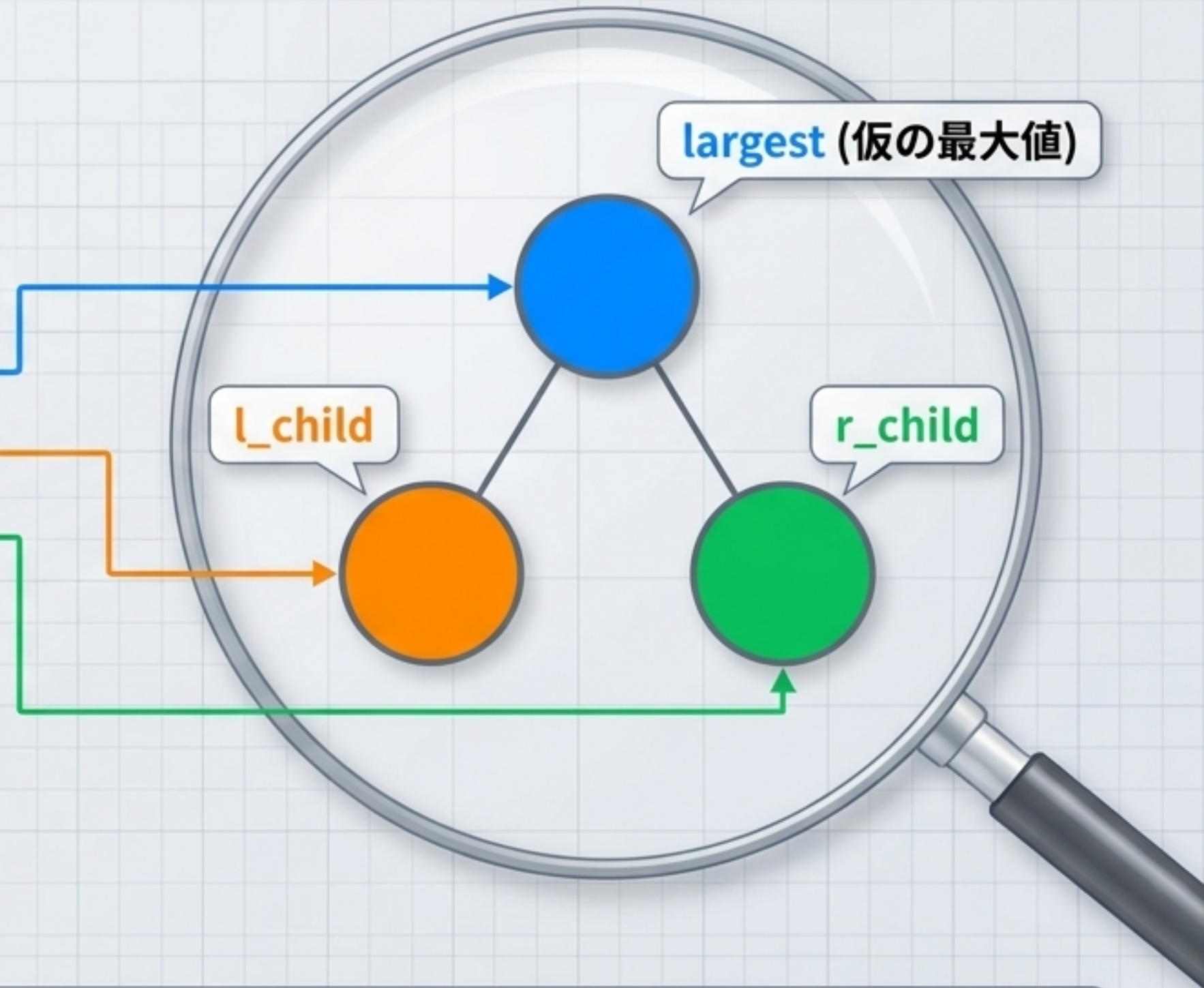


配列 A 内の要素を入れ替えるユーティリティ関数。

Python特有の「タプルアンパック」機能により、一時変数 (temp) を用意することなく、1行で安全かつ高速にスワップ処理を完了できる。

Code Block 2 (heap_correct 前半): 調査対象ノードのロックオン

```
def heap_correct(A, i, length):  
    largest = i  
    l_child = 2 * i + 1  
    r_child = 2 * i + 2
```



目的: 起点となる親ノード i と左右の子ノードの配列内インデックスを特定する。初期状態では、親自身を「仮の最大値」として扱う。

Code Block 3 (heap_correct 中盤): 3ノード間の「王座決定戦」

```
if l_child < length and A[l_child] > A[largest]:  
    largest = l_child  
  
if r_child < length and A[r_child] > A[largest]:  
    largest = r_child
```

設問1 [ア]: A[largest]より大きい

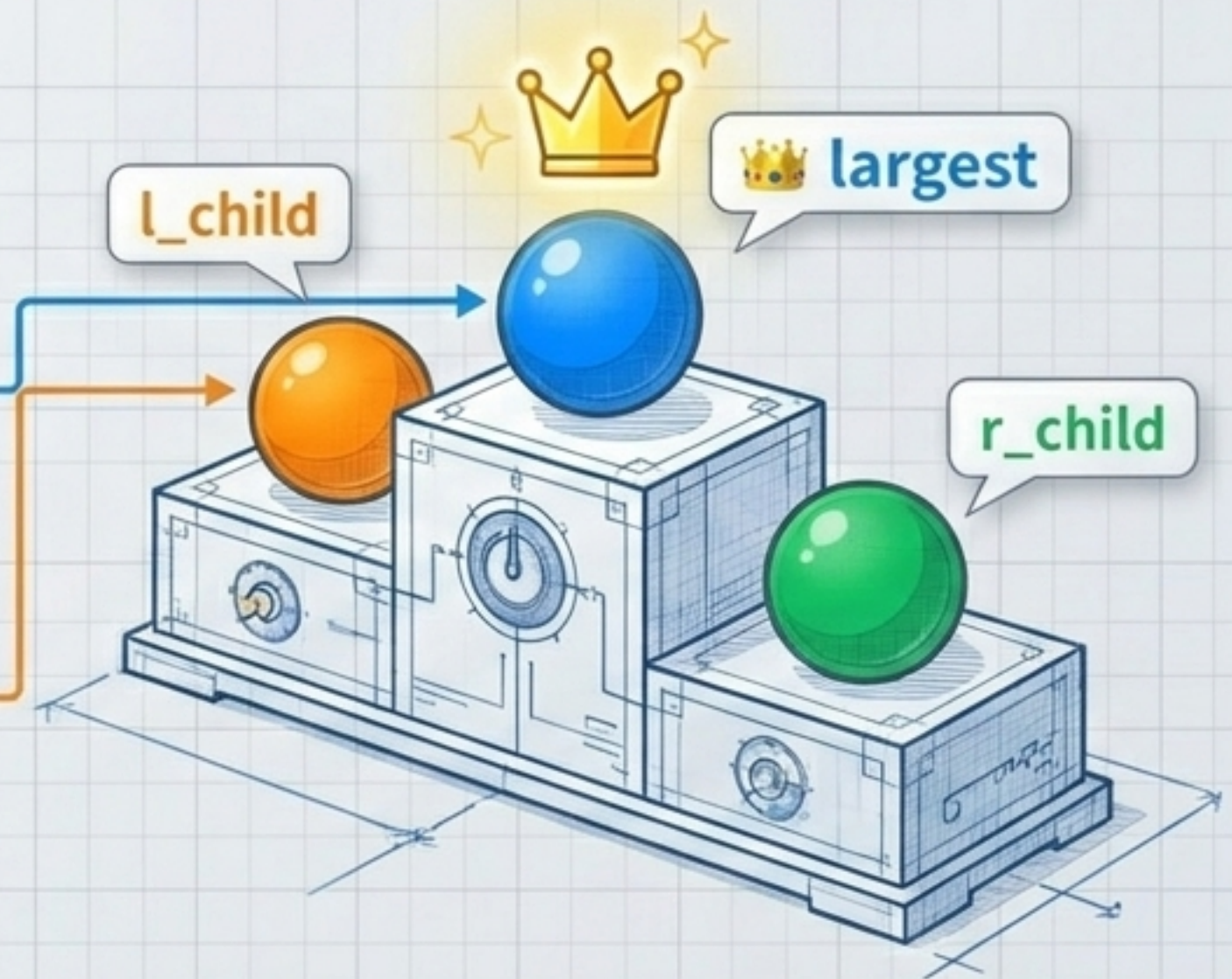
A[l_child] > A[largest]:

largest = l_child

A[r_child] > A[largest]:

largest = r_child

設問1 [イ]: A[largest]より大きい



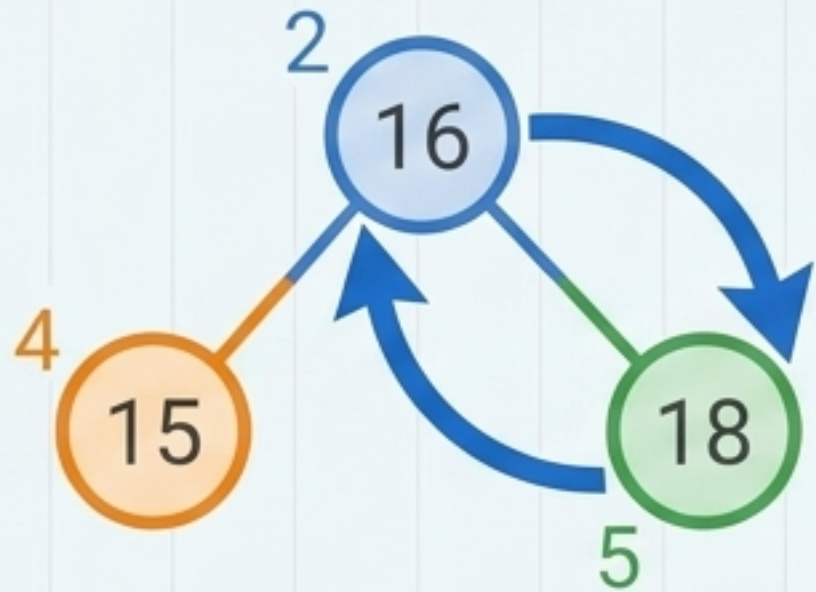
配列の範囲内 (length未満) であることを確認した上で、親・左の子・右の子の3つの「値」を比較。最も値が大きいノードを探し、勝者の位置を **largest** に上書きする。

Code Block 4 (heap_correct 後半) : 秩序の回復と再帰的連鎖

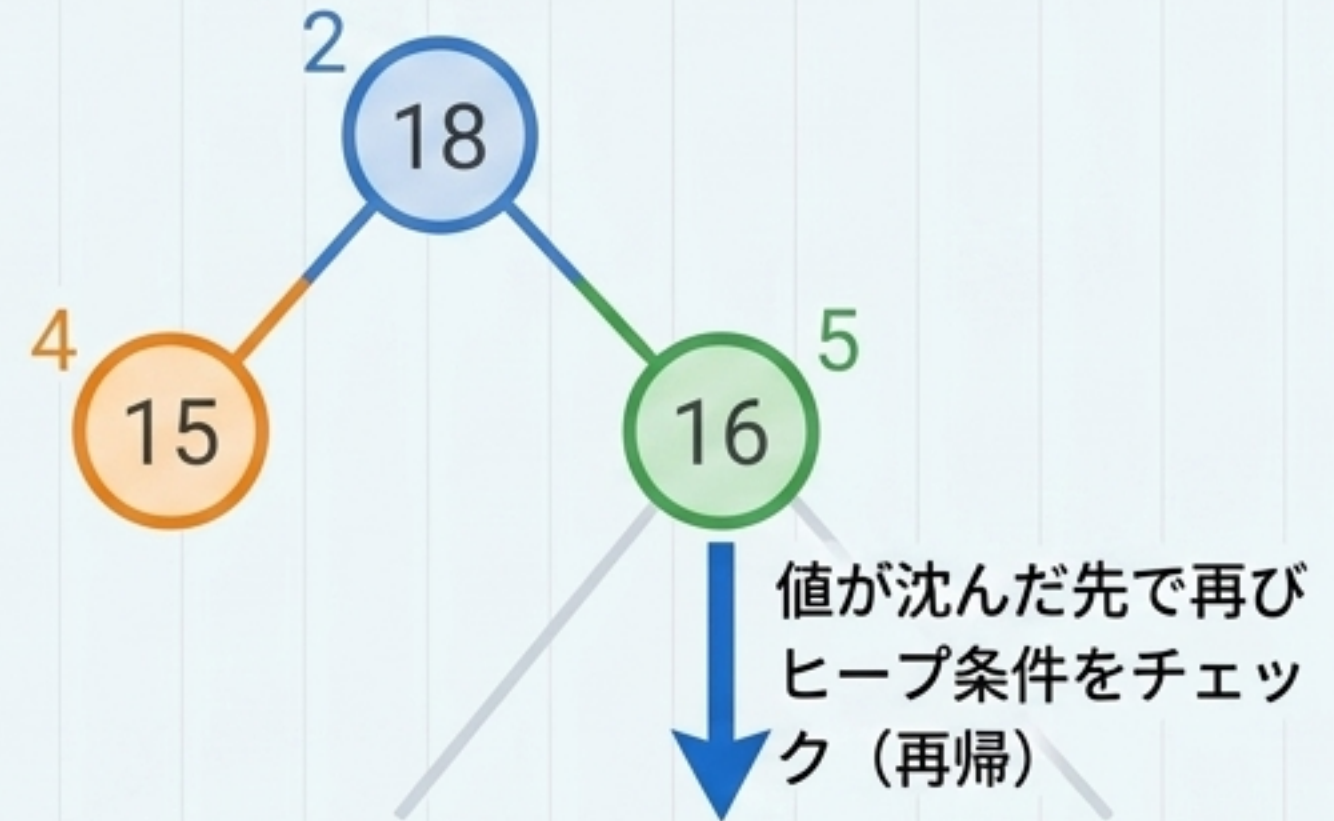
```
if largest != i:  
    exchange(A, i, largest)  
    heap_correct(A, largest, length)
```

設問4[カ]: イより大きい

Before



After



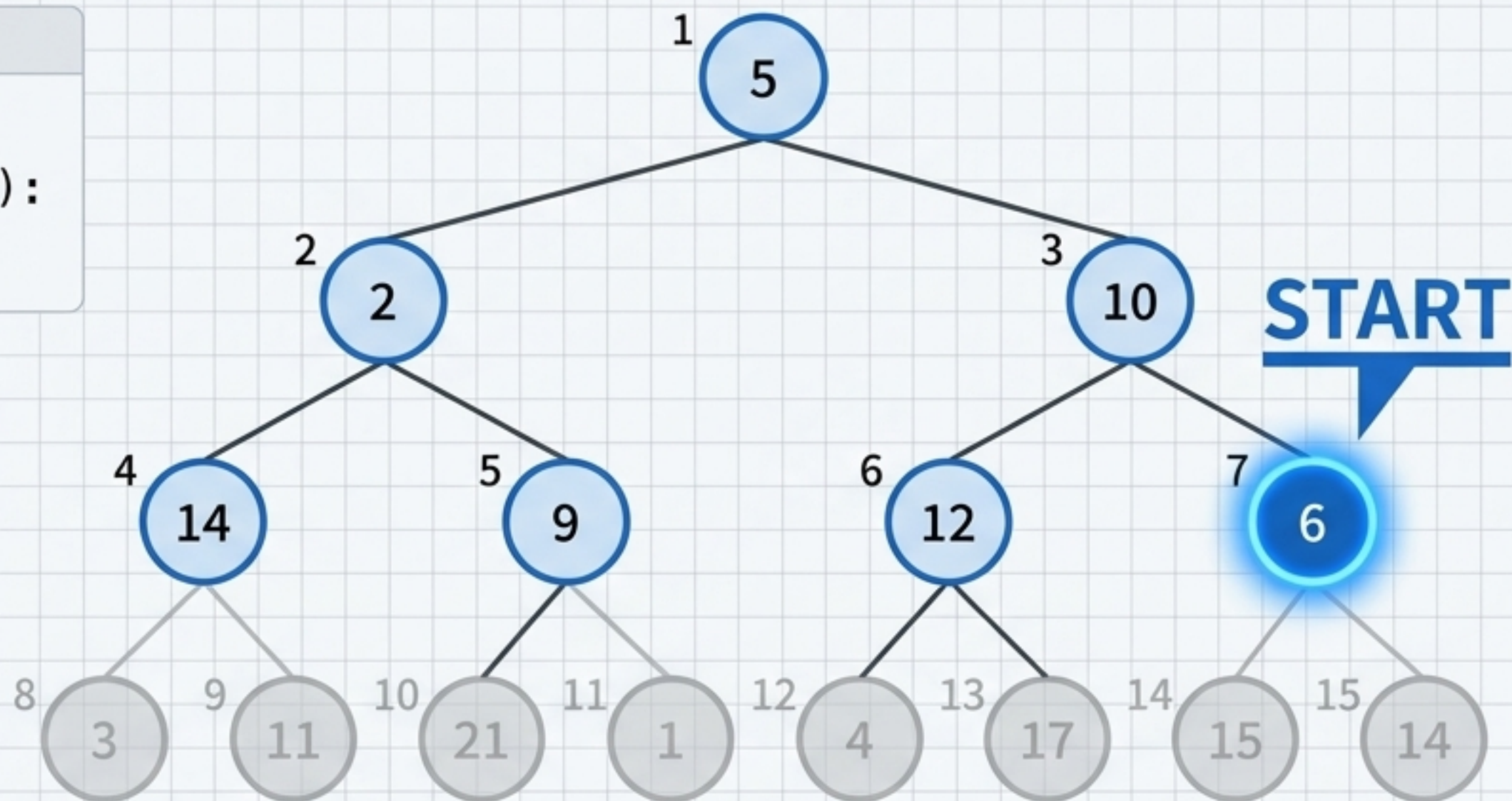
親が最大でなかった場合、値を入れ替え (exchange) で秩序を回復する。値が下へ「沈んだ」場合、移動先の部分木でさらに条件が壊れていないか、自身を再帰的に呼び出して底まで確認を続ける。

Code Block 5 (heap_make): ボトムアップ構築の全体コントローラー

```
start_node = (length // 2) - 1
for i in range(start_node, -1, -1):
    heap_correct(A, i, length)
```

設問2[ウ]

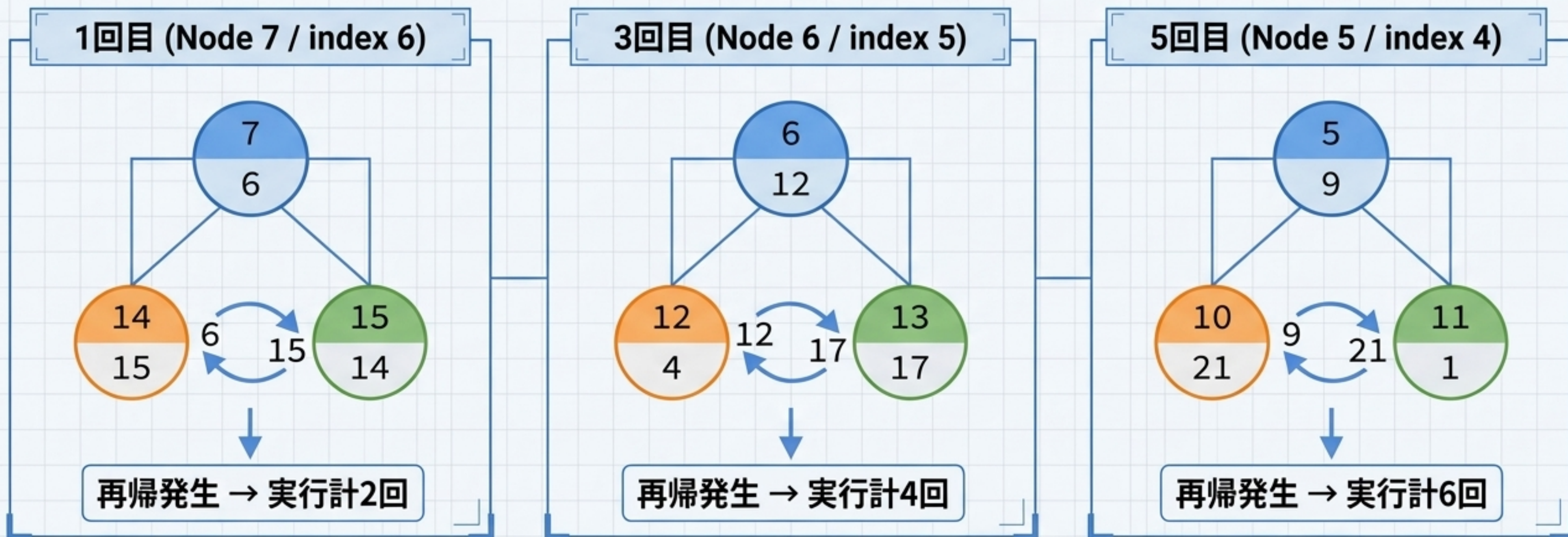
設問2[エ]



バラバラの木をヒープ化するには、下層から上層へ順に `heap_correct` を適用する。葉ノードは子を持たないため処理不要。開始位置は常に「子を持つ最後の親ノード」。要素数15の場合、Python(0-index)の計算式 $(15 // 2) - 1 = 6$ (ノード番号7) が始点となる。

Execution Trace (1): heap_make の軌跡と再帰回数を追跡

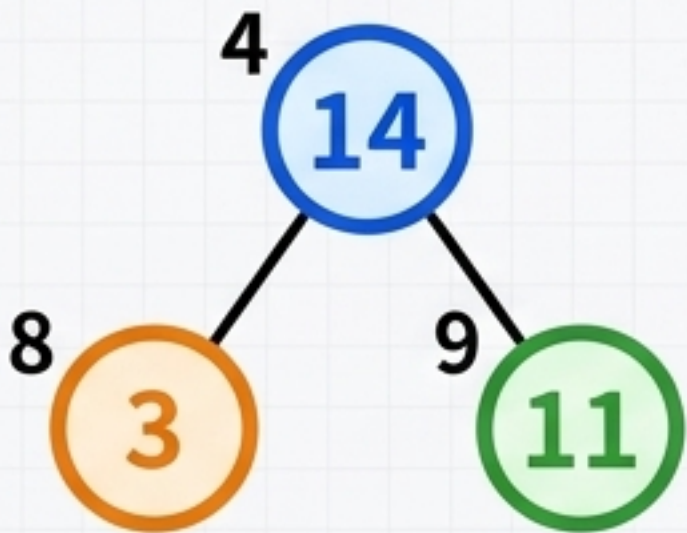
設問3に基づく、ノード7からノード5までの処理の解剖



Insight: 各ノードでの比較と入れ替え後、値が「沈む」ため必ず1回の再帰呼び出しが発生している。
この時点で `heap_correct` は合計6回実行されている。

Execution Trace (2): 頂点への到達と最大値の確定

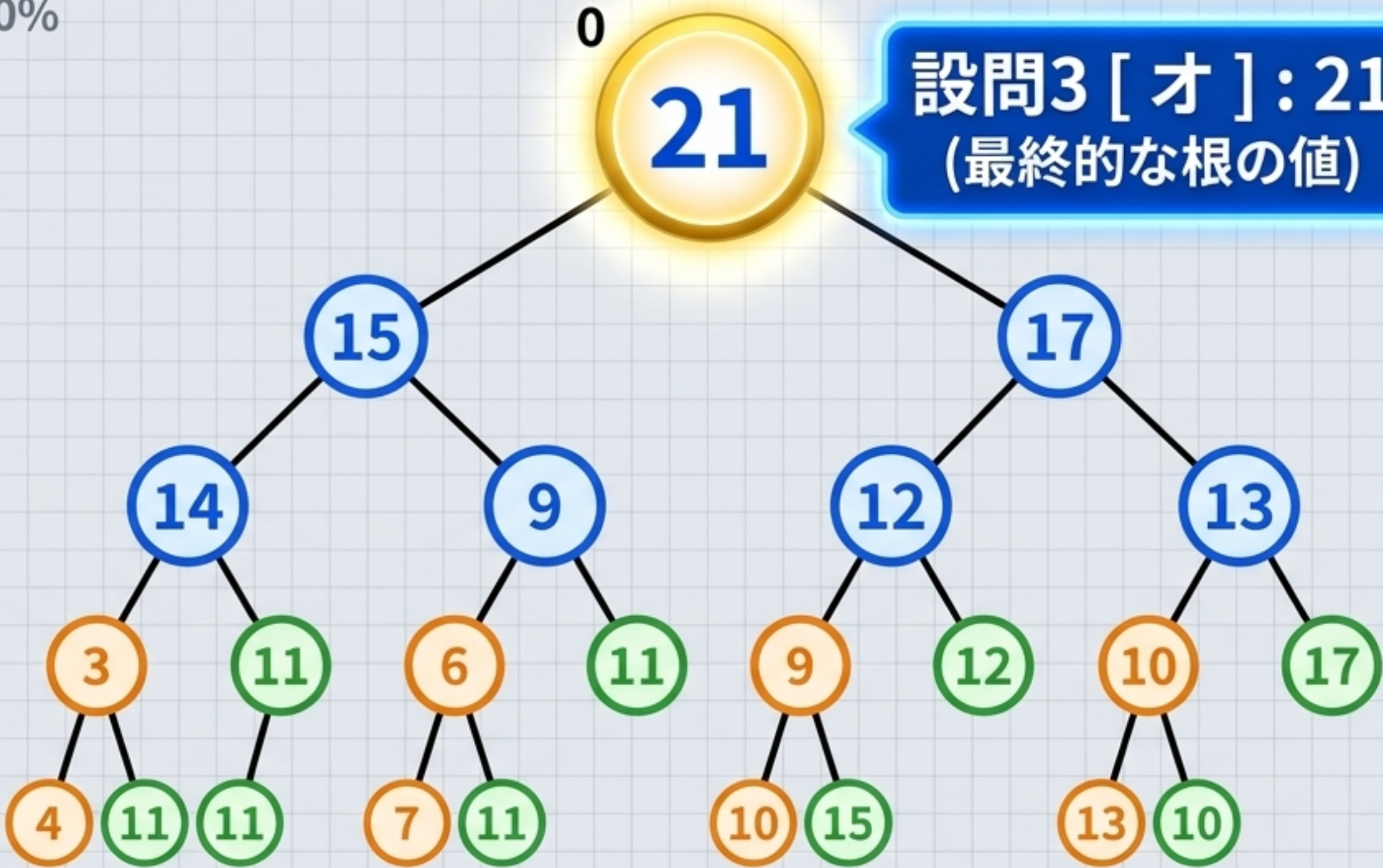
30% ノード4の処理



親14は子3, 11より大きい。
スワップなし、再帰なし。

設問3 [キ]: 7回
(これまでの6回 + 今回の1回)

70%



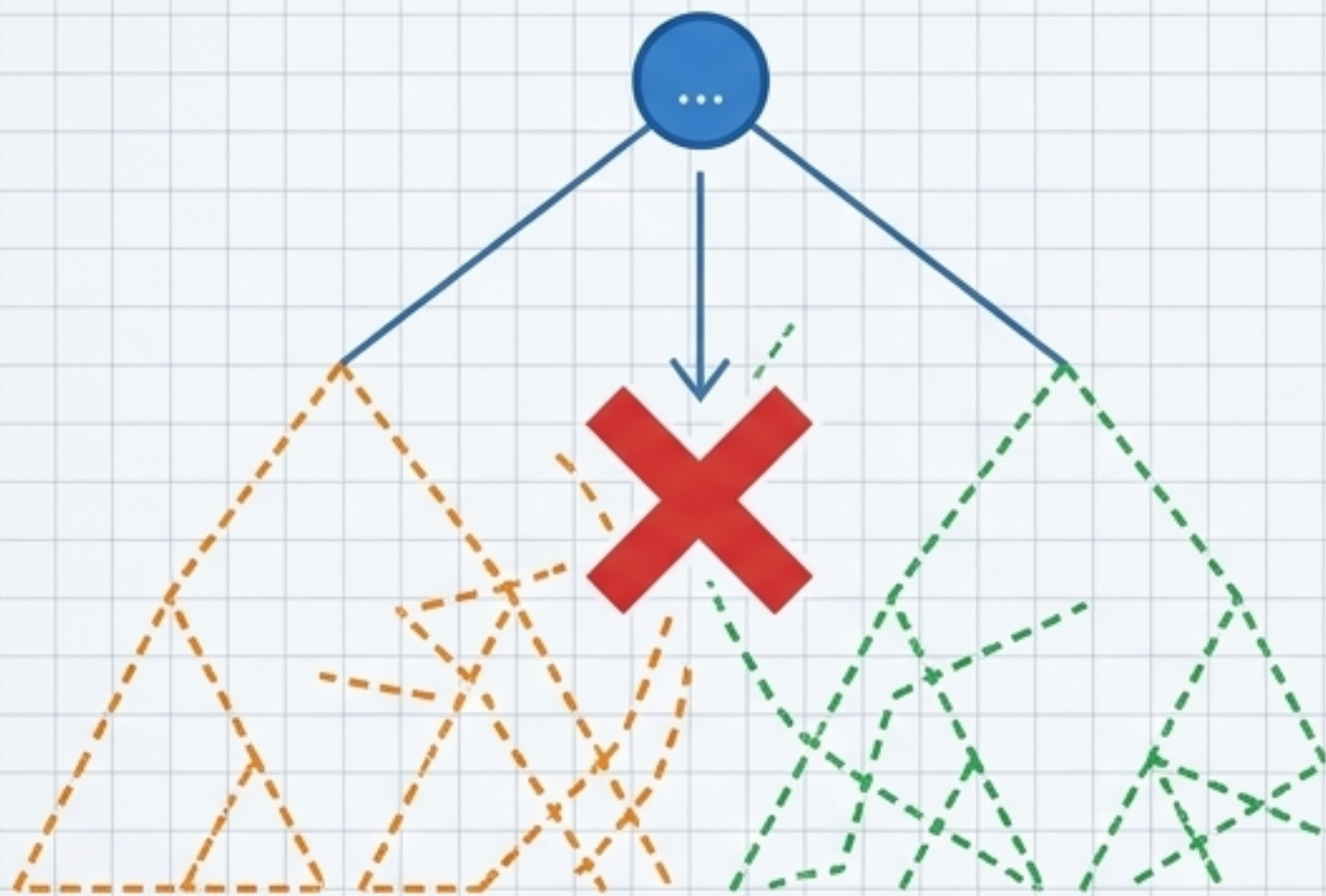
設問3 [オ]: 21
(最終的な根の値)

変換後配列: [21, 15, 17, 14, 9, 12, ..]

Insight: なぜ heap_make は「下から上へ」構築するのか？

heap_correct の大前提：「対象の子ノードを根とする部分木は、あらかじめヒープ条件を満たしている必要がある」

上から構築した場合 (Anti-Pattern)



値が沈んだ先が**無法地帯**であるため、どこまでも崩壊が続き秩序が作れない。

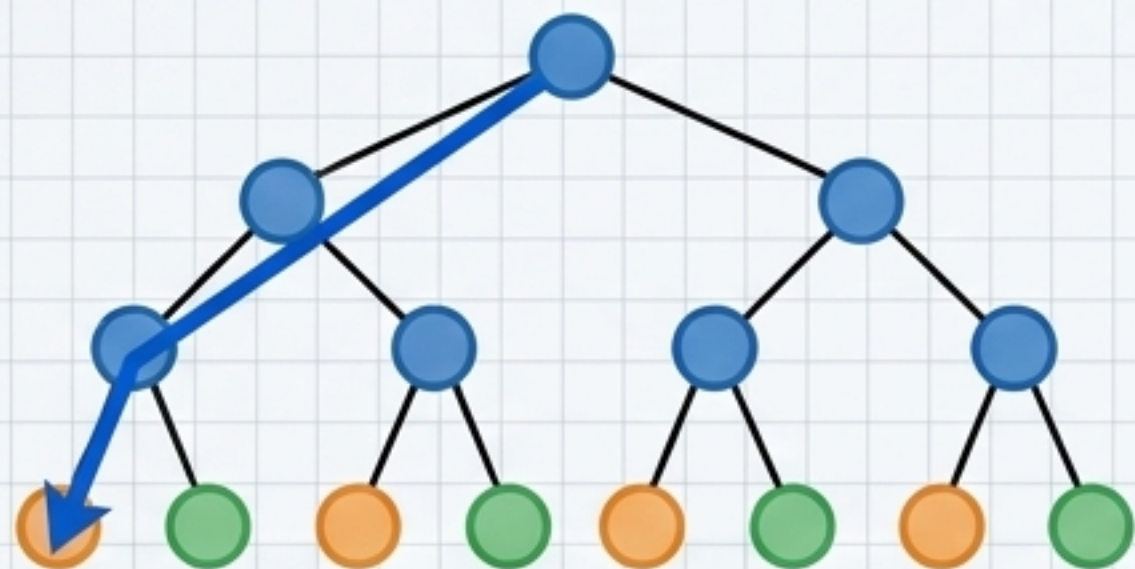
下から構築した場合 (The Solution)



下層の小さな木から順に**完全な秩序**を完了させ、それを**ブロック**のように積み上げることで、全体の**ヒープ化**を保証している。

アルゴリズムのパフォーマンス評価: 計算量の視覚的証明

① heap_correct 単体の計算量



比較ごとにツリーの階層を1段下がる。
ノード数 N に対し、高さは $\log N$ に比例するため、
最悪でも木の高さ分しか処理が走らない。

設問5 ① [イ]: $O(\log N)$

② 降順ソート済み配列での heap_make

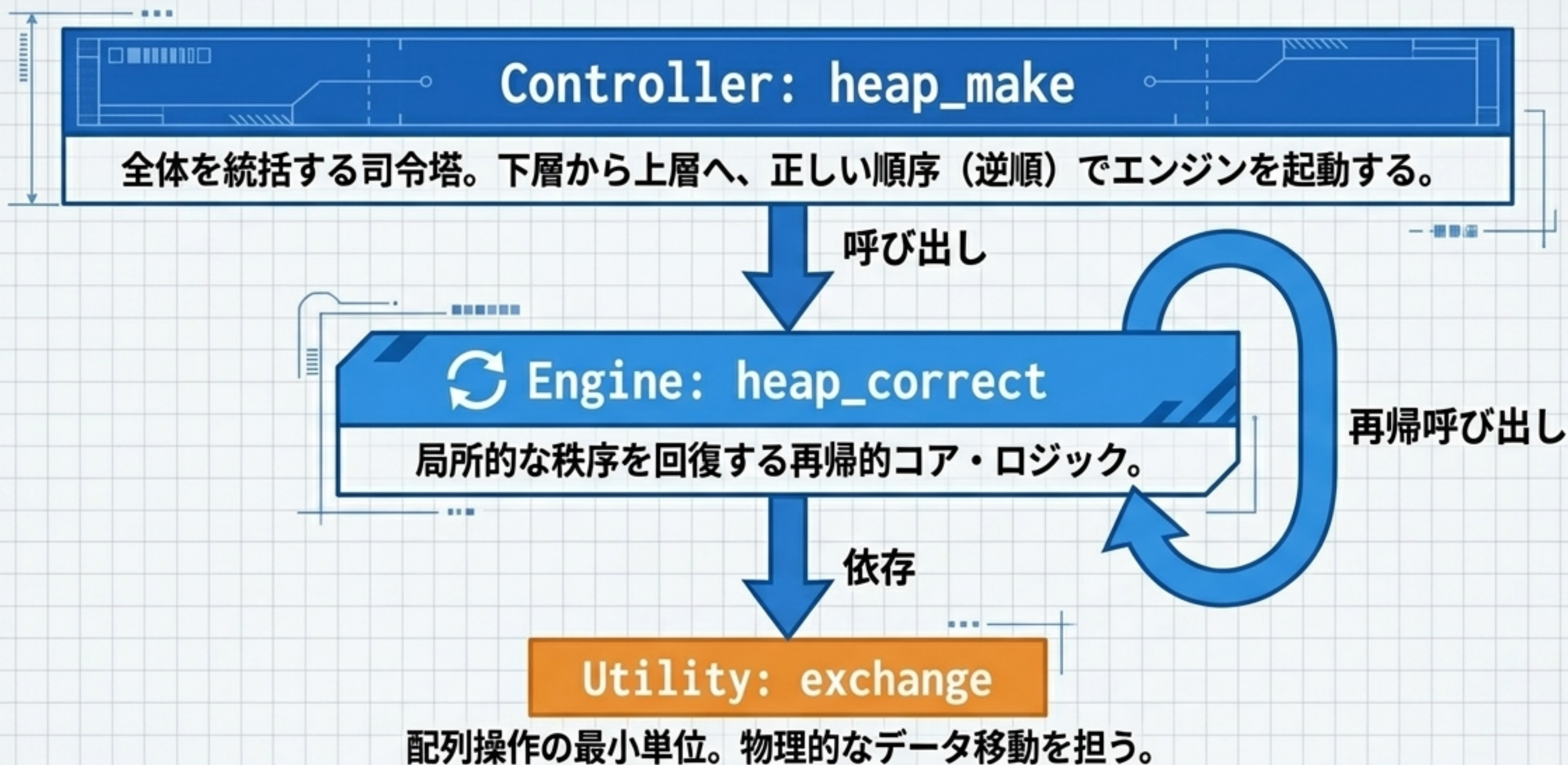


設問5 ② [ウ]: $O(N)$

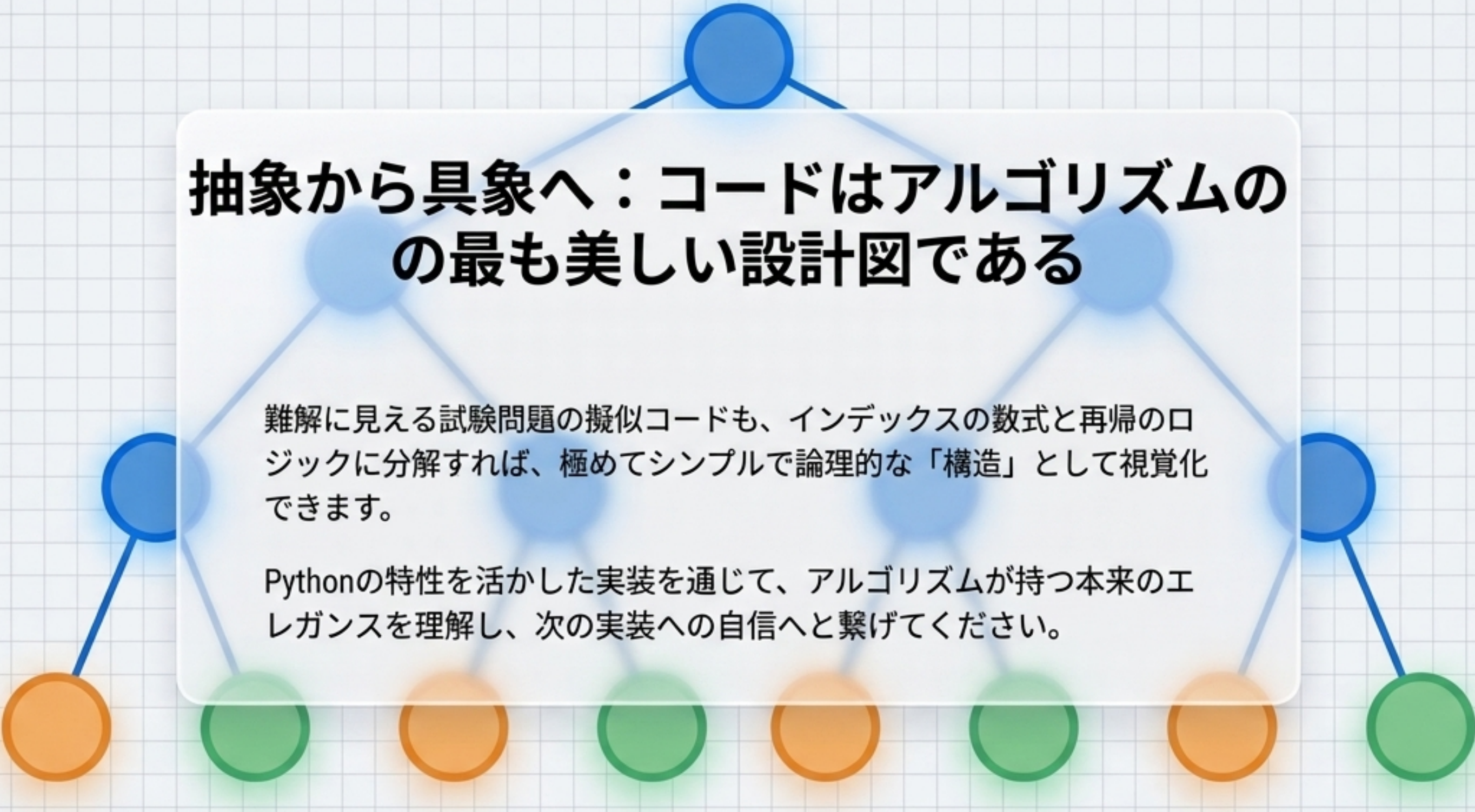
すべての部分木が既にヒープ条件を満たすため、
スワップも再帰も発生しない。
全ノードで初回判定のみ（各1回）で即座に
停止する。

設問5 ② [ウ]: $O(N)$

System Architecture: 3つの関数が織りなすソートの全体像



これらが密に連携することで、任意の配列が確実に高速にヒープ構造へと変換される。



抽象から具象へ：コードはアルゴリズムの最も美しい設計図である

難解に見える試験問題の擬似コードも、インデックスの数式と再帰のロジックに分解すれば、極めてシンプルで論理的な「構造」として視覚化できます。

Pythonの特性を活かした実装を通じて、アルゴリズムが持つ本来のエleganceを理解し、次の実装への自信へと繋げてください。