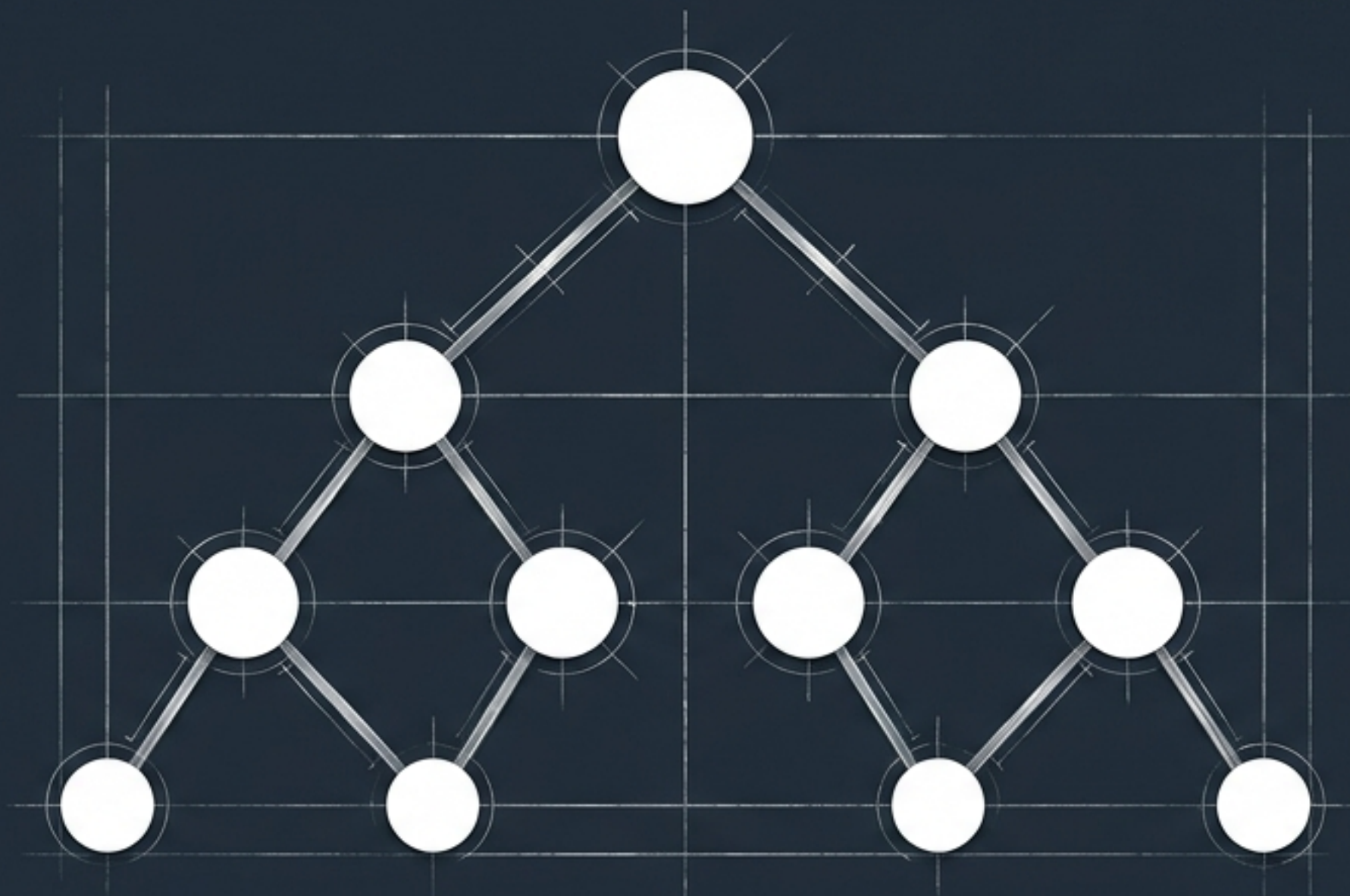
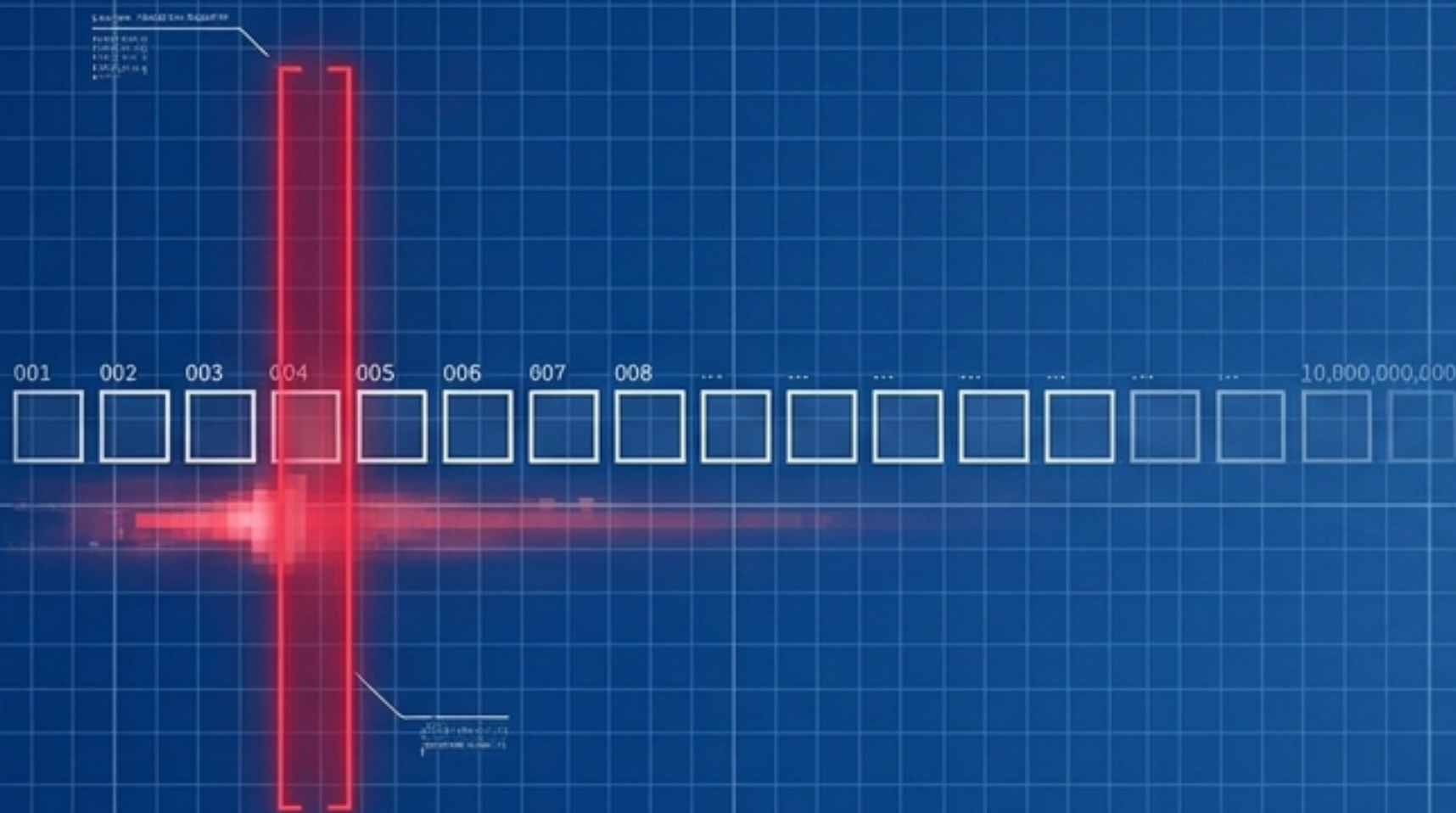




# 情報の設計図：二分探索木

膨大なデータから「一瞬」で探し出すアルゴリズムの仕組み

## The Linear Nightmare



順番に探す（線形探索 /  $O(n)$ ）

10億個のデータがある場合、最悪10億回の確認が必要（約16分）。

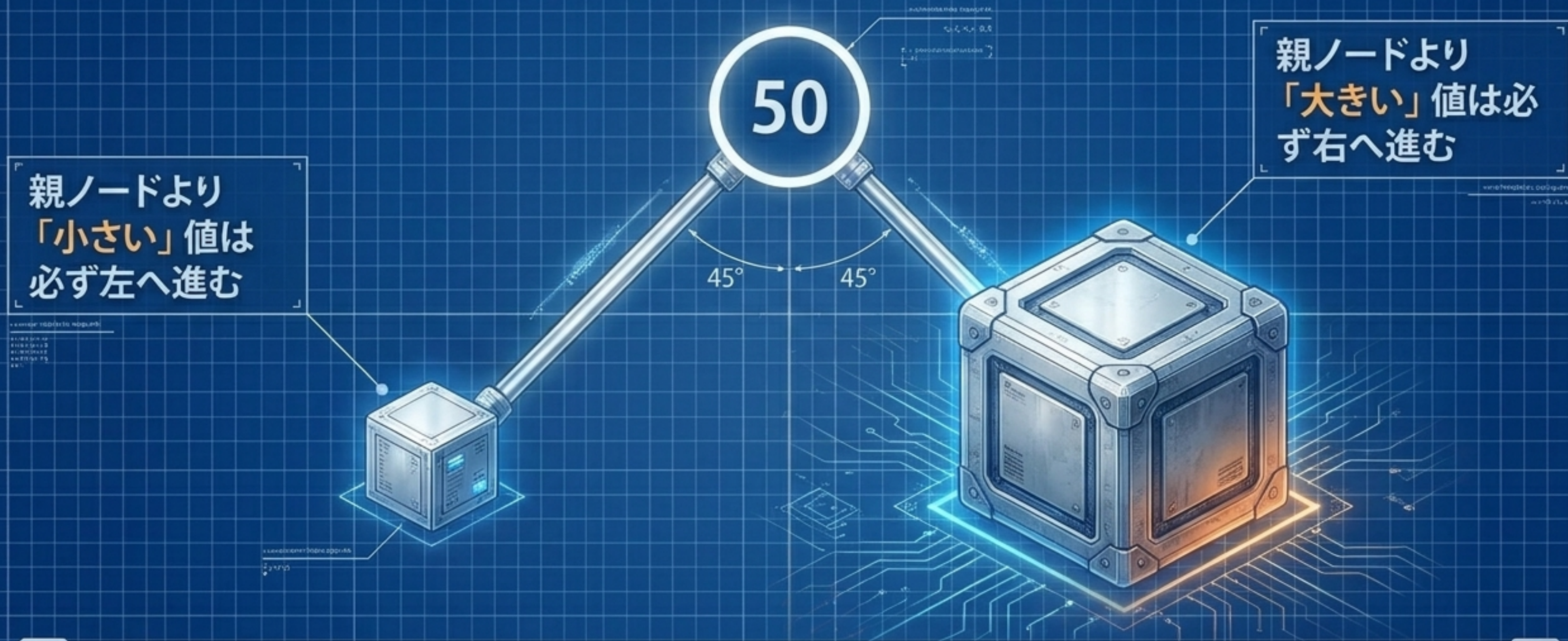
## The Logarithmic Magic



半分に絞り込む（二分探索 /  $O(\log n)$ ）

10億個のデータがあっても、わずか約30回の確認で完了（約0.000000002秒）。

# 二分探索木の「たった1つのルール」



このシンプルな仕掛けが、1ステップごとに残りの探索範囲を「半分」に切り捨てる。

# 探索 (Search) : 「7」 を見つけ出すまでの軌跡



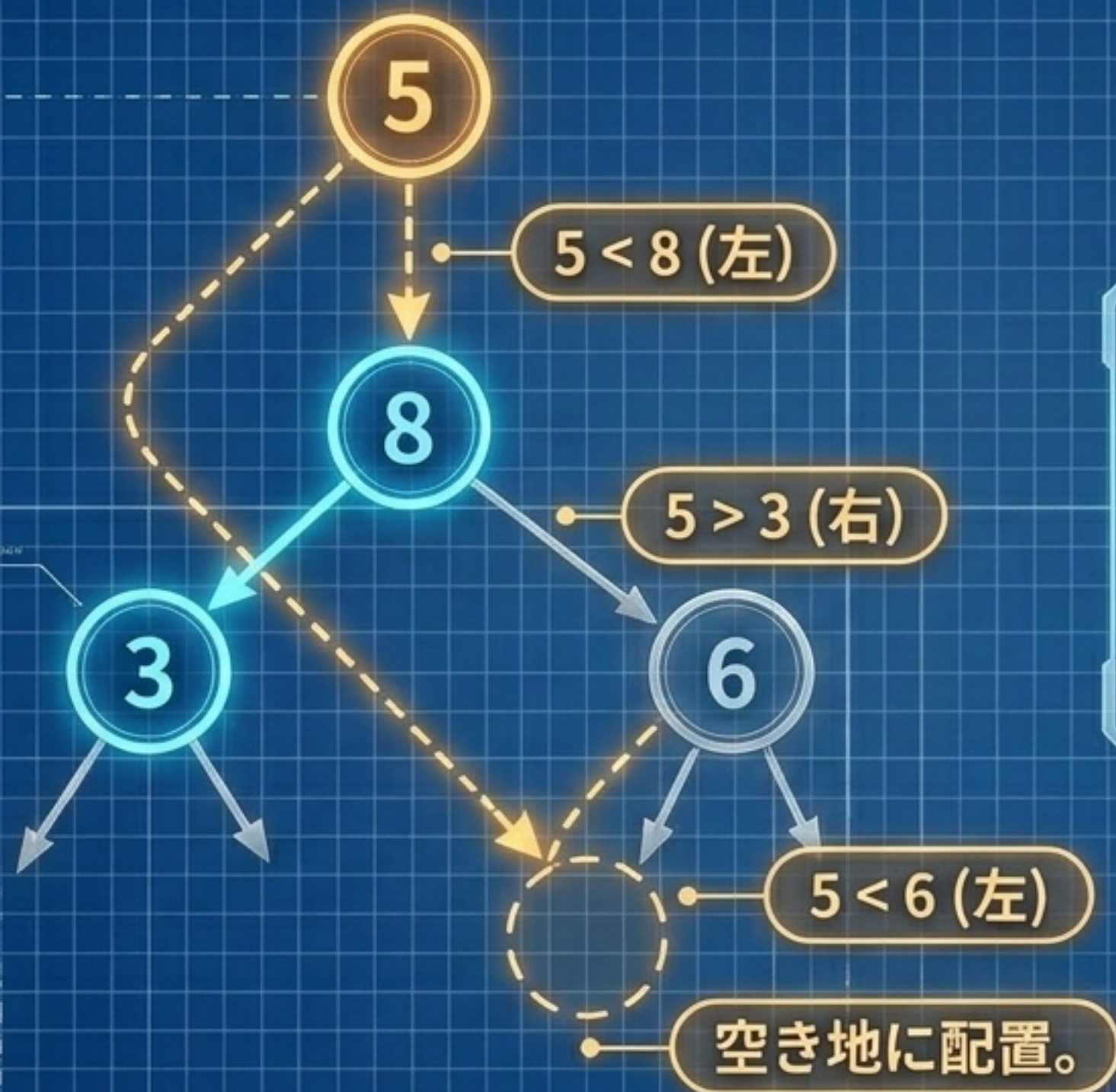
Step 1: 「7を探したい」  
→ スタートは「8」。  
7は8より小さいので左へ。

Step 2: 次は「3」。  
7は3より大きいので右へ。

Step 3: 次は「6」。  
7は6より大きいので右へ。

Step 4: 「7」を発見。  
わずか3回の移動で完了。

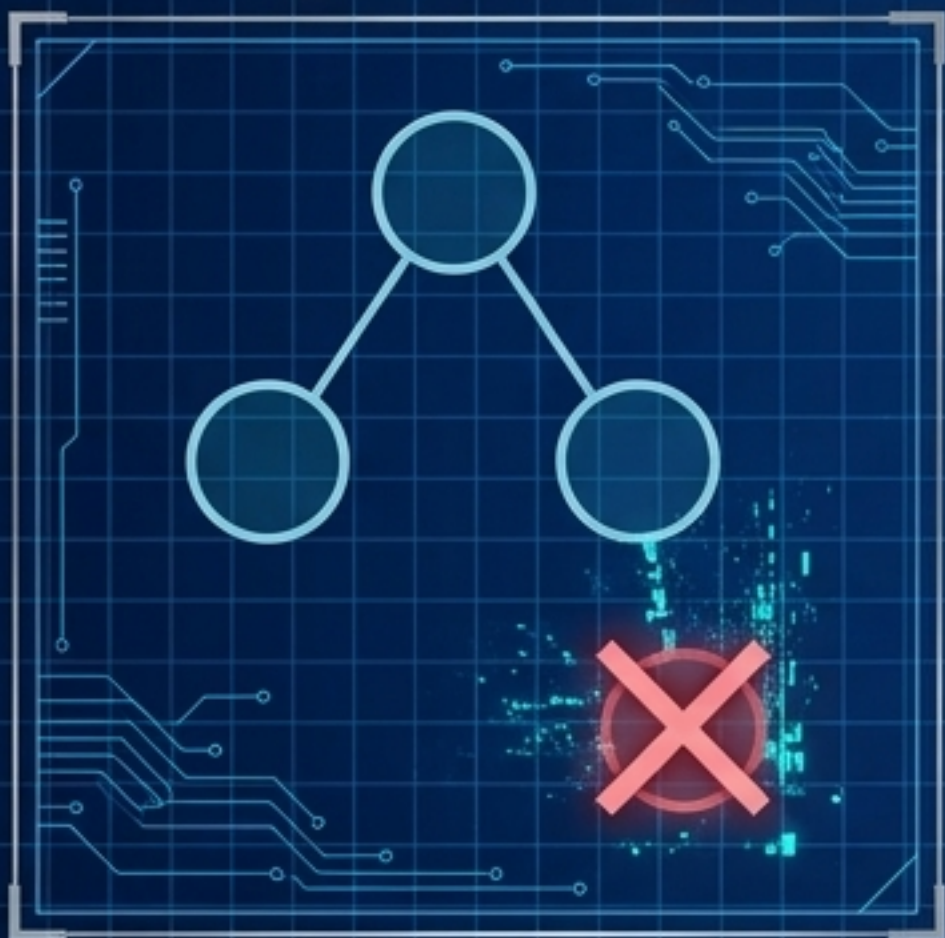
# 挿入 (Insert) : 新しいデータの居場所を決める



探索と全く同じルールで進み、  
行き止まり(リーフ)になった場所が、  
新しいデータの定位置になる。

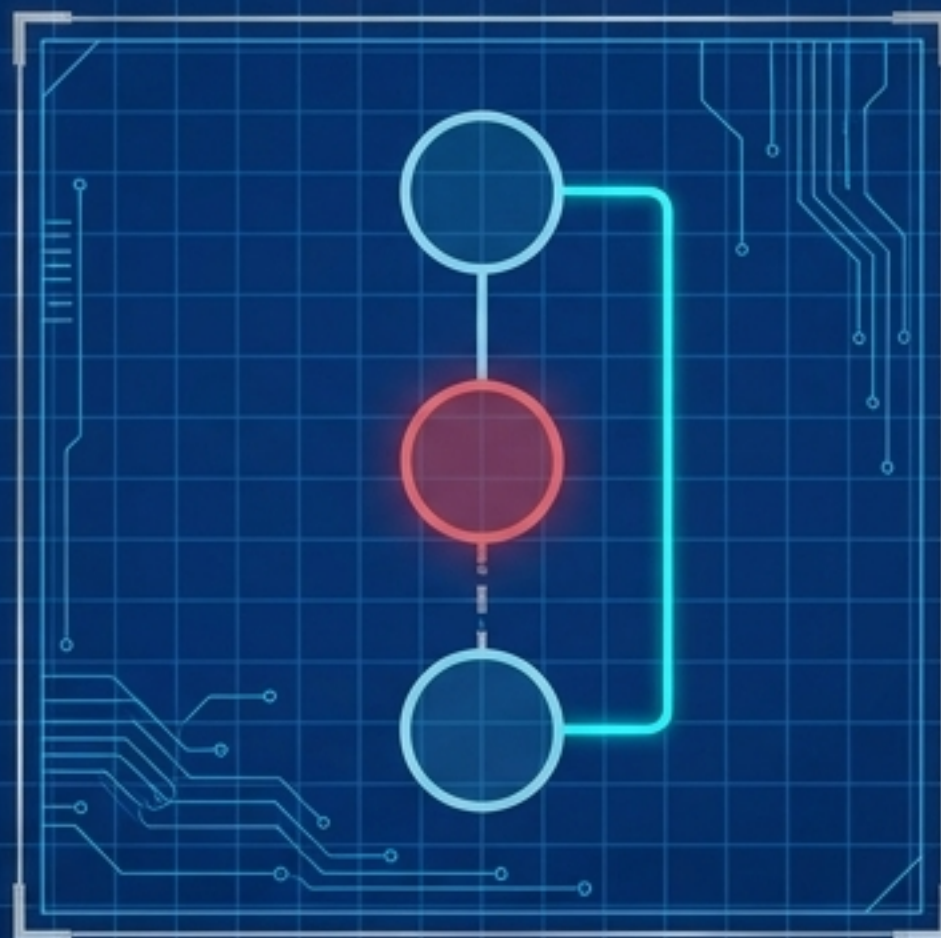
# 削除 (Delete) : 3つの診断ケース

子がない場合  
(0 Children)



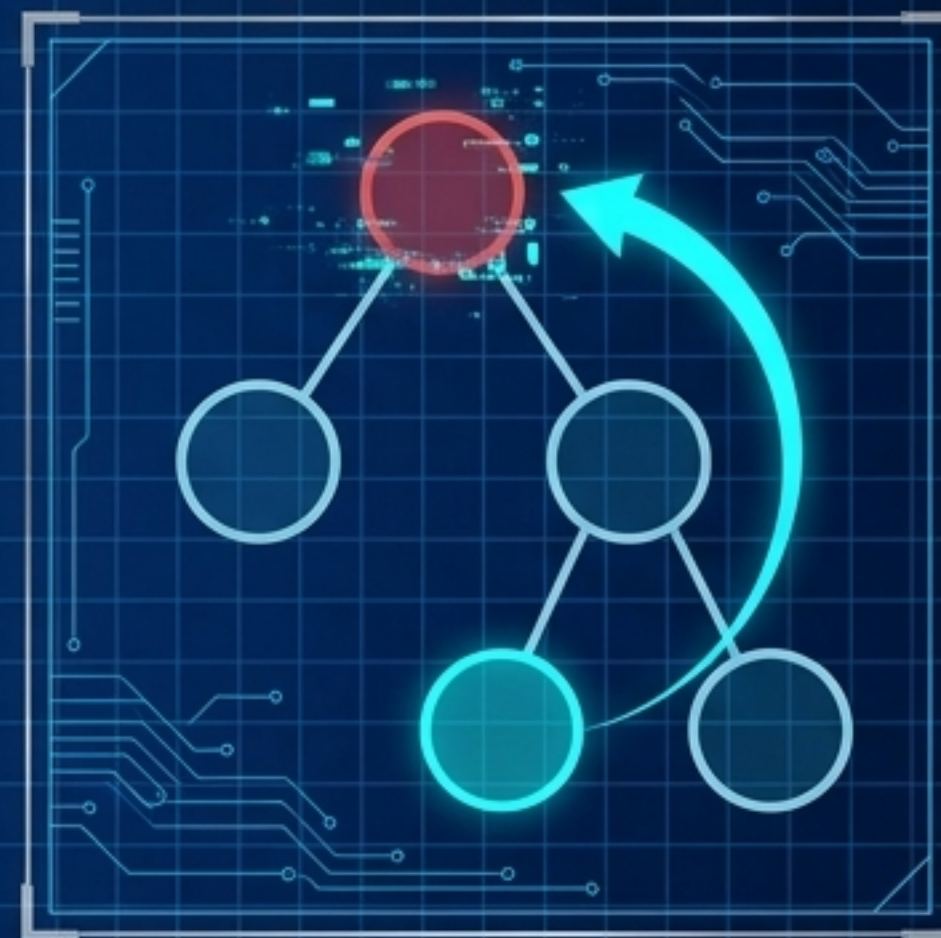
そのまま消去するだけ。

子が1つの場合  
(1 Child)



削除されたノードの位置に、  
その子を昇格させる。

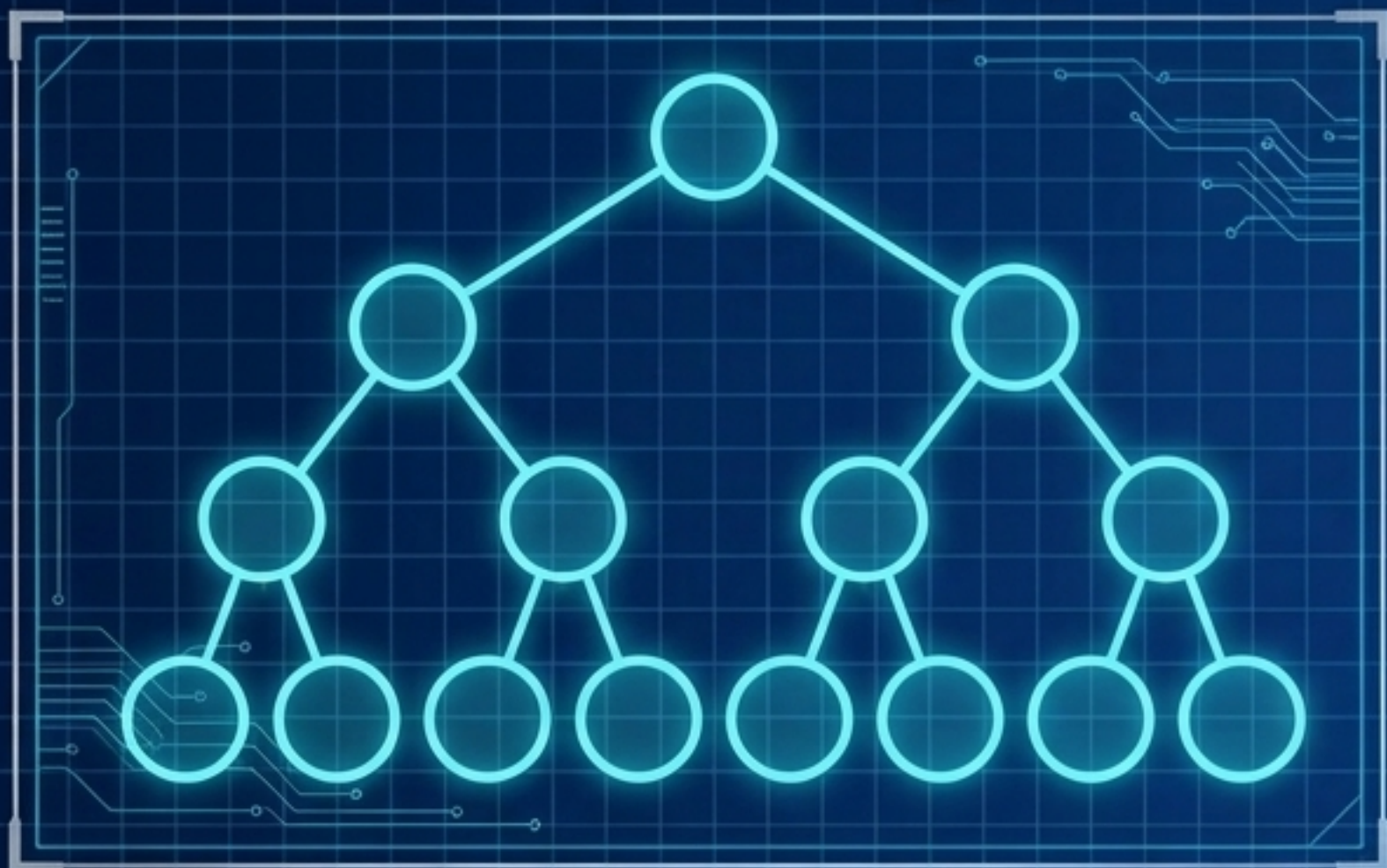
子が2つの場合  
(2 Children)



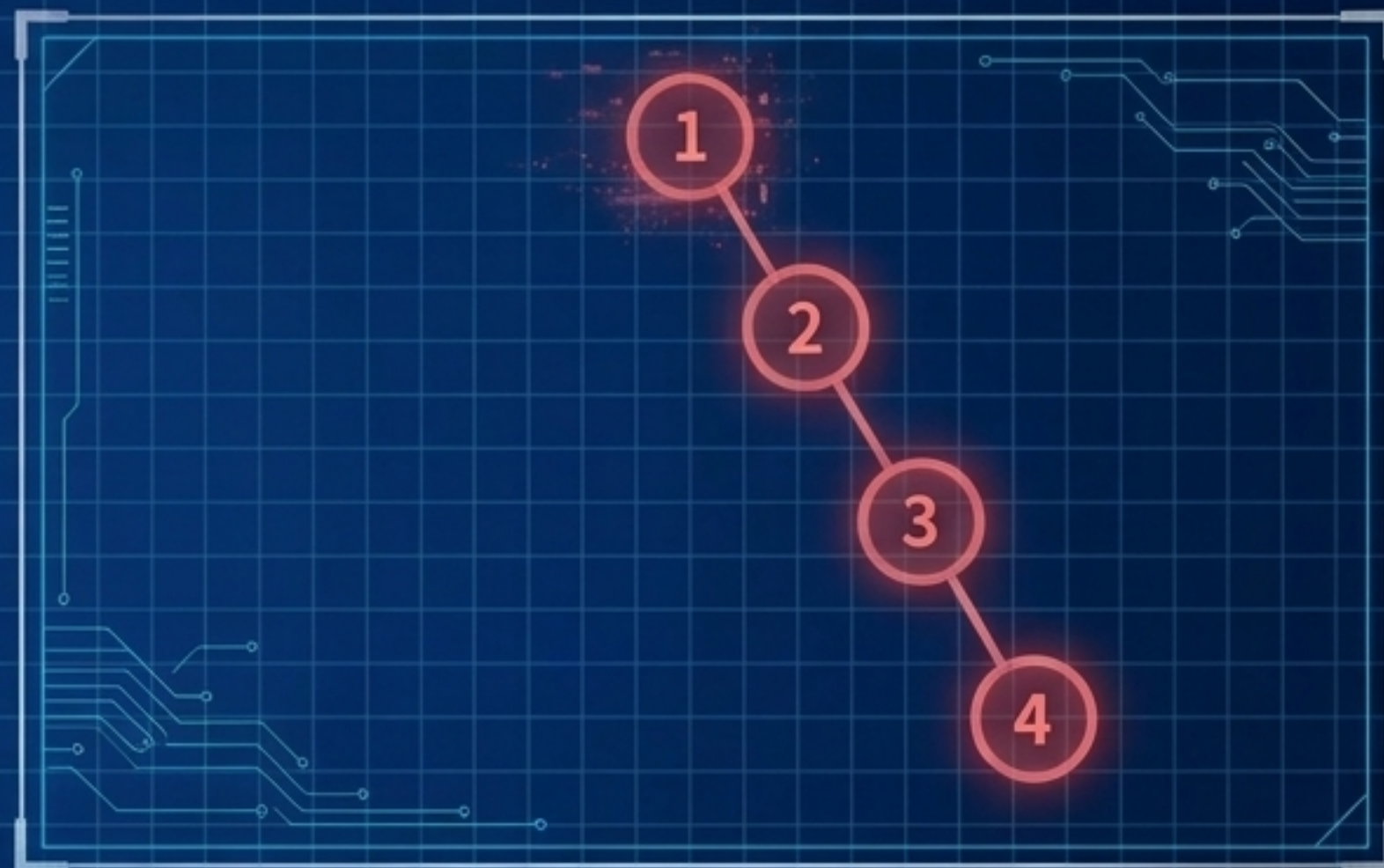
右側の木の中で「一番小さい値」を  
持ってきて入れ替える（二分木の  
ルールを壊さないための工夫）。

# 単純な二分探索木の「致命的な弱点」

理想的な状態 ( $O(\log n)$ )



危険な状態



データがすでに整列された状態 (1, 2, 3, 4...) で順番に挿入されると、木は右にしか伸びず「ただの直線」になってしまう。

これでは線形探索 ( $O(n)$ ) に逆戻り。木が「偏る」ことを防ぐ自己調整の仕組みが必要になる。

# 進化の系譜：自己バランス木の登場

| データ構造                                                                                                                      | 戦略 (Strategy) & メリット (Pros)                                                 | 実用例 (Real-World Use Cases)   |
|----------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------|------------------------------|
| <b>Treap</b><br>(ランダム二分探索木)<br>          | <b>戦略:</b> 乱数 (優先度) による回転<br><b>メリット:</b> 期待値として高速。実装が比較的シンプル。              | 動的でシンプルなデータセット               |
| <b>Scapegoat Tree</b><br>(スケープゴート木)<br> | <b>戦略:</b> 偏りを見つけ、部分的に再構築<br><b>メリット:</b> 余分なポインタなどのメモリ消費がない。               | メモリに制限があるシステム                |
| <b>Red-Black Tree</b><br>(赤黒木)<br>     | <b>戦略:</b> ノードを色分けし、厳密なルールで高さを制限<br><b>メリット:</b> 最悪の場合でも必ず $O(\log n)$ を保証。 | Linuxカーネル、C++/Javaの標準ライブラリ   |
| <b>B-Tree</b><br>(B木)<br>              | <b>戦略:</b> 1つのノードに複数のデータを持たせる<br><b>メリット:</b> ハードディスク等の外部メモリアクセスを劇的に削減。     | データベース、ファイルシステム (Ext4, NTFS) |