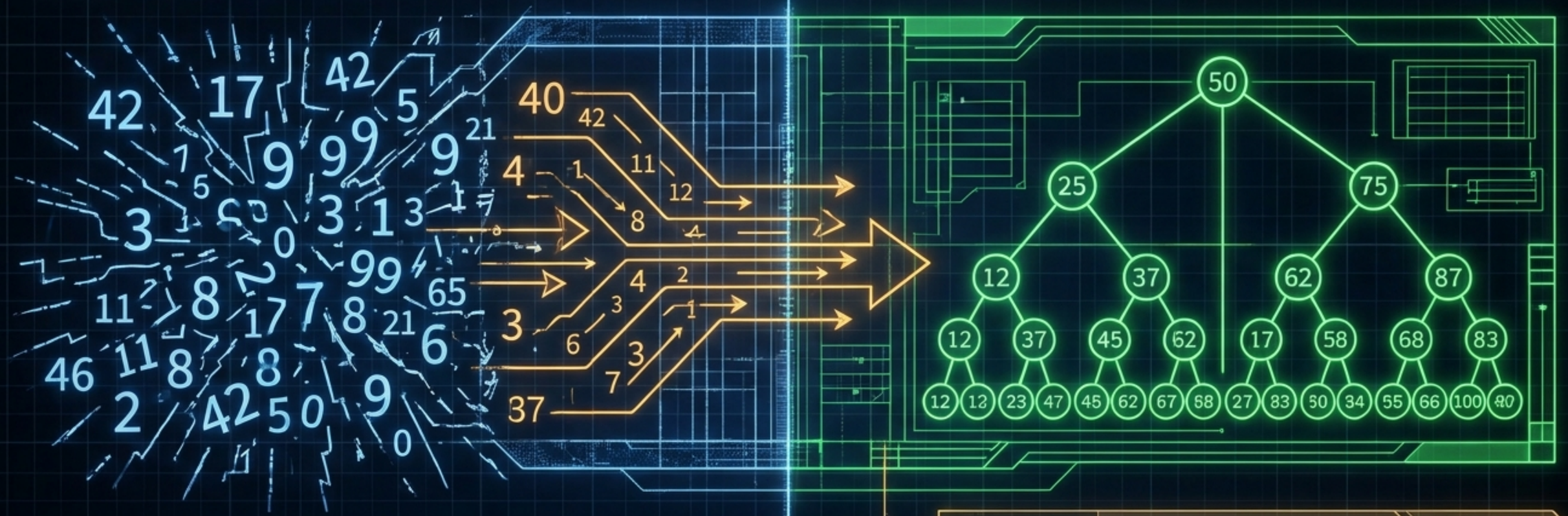
演算法大師課：排序與搜尋的底層邏輯

從零開始的費曼學習法 (From Zero to Mastery via the Feynman Technique)



Target: 從零開始的費曼學習法

Core Topics: Chapter 4.2 - 4.3 (Sorting & Searching Algorithms)

Feynman Callout

Professor's Note: 拋開死記硬背。理解資料的流動，你就能掌握演算法。

什麼是排序與搜尋？(The Core Philosophy)

Key English Terms:
> **Sorting**: Organizing data to minimize retrieval time.
> **Searching**: Locating a specific Key within a dataset.
> **Algorithm Analysis**: Evaluating efficiency via Big-O notation.

The Problem

The Solution

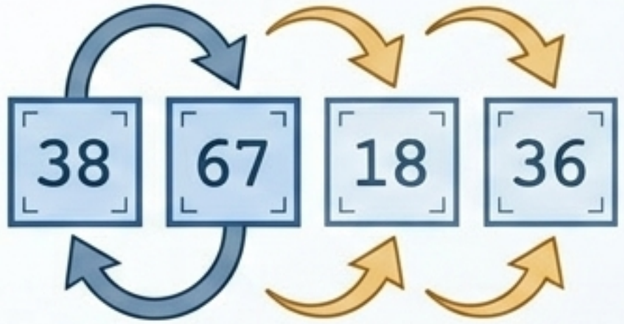
Feynman Callout

「想像圖書館裡散落一地的十萬本書。排序（Sorting）是把書按字母精準排好；搜尋（Searching）是在排好的書中瞬間找到你要的那本。沒有排序，搜尋就只能大海撈針。」

基礎迭代排序：直覺但緩慢的方法

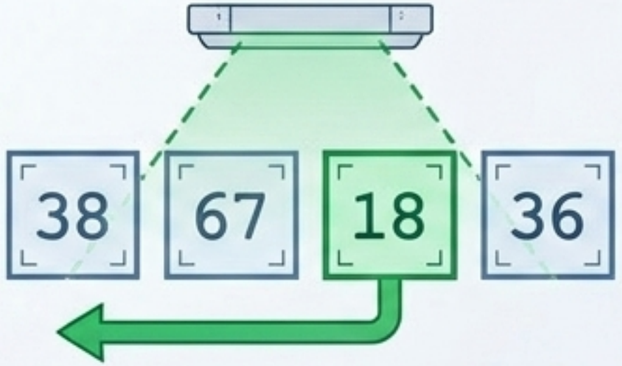
[38 67 18 36]

氣泡 (Bubble Sort)



Swap adjacent, push max to end.

選擇 (Selection Sort)



Find min, place at front.

插入 (Insertion Sort)

Insert into sorted section.

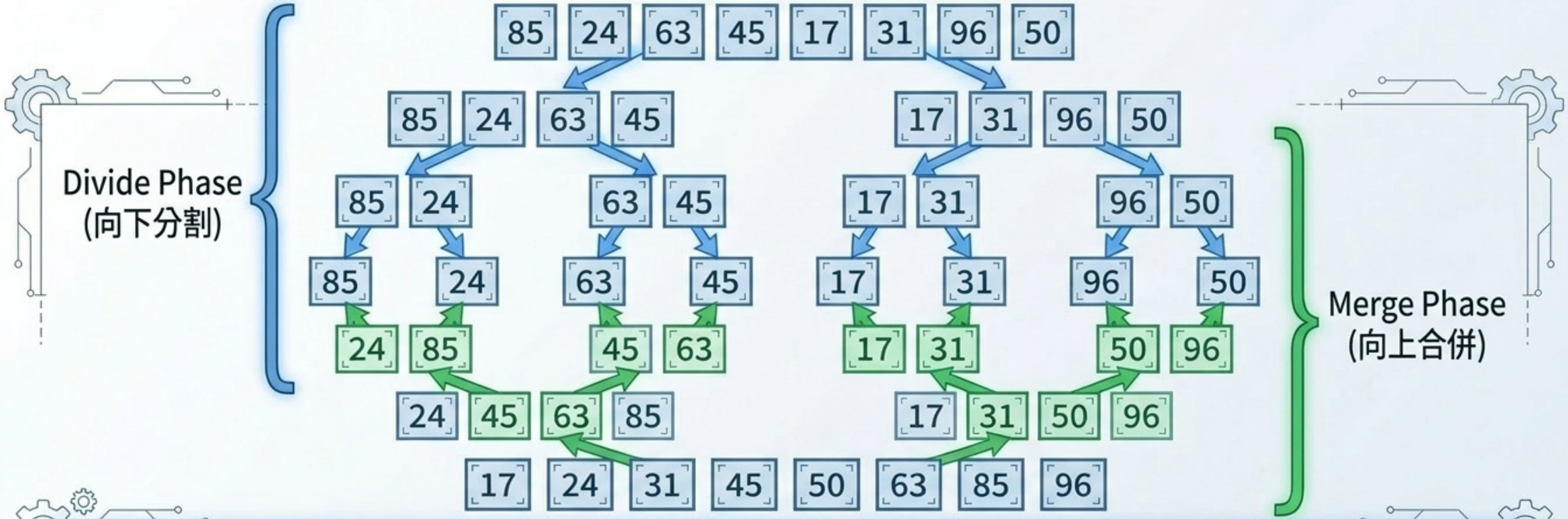
Complexity:
Average & Worst Case = **O(n^2)**

Feynman Callout

「這三種方法最符合人類直覺：相鄰互換、挑選最小、像整理撲克牌一樣插入。但當資料量過萬時，它們會慢得令人崩潰。」

突破瓶頸：合併排序法 (Merge Sort)

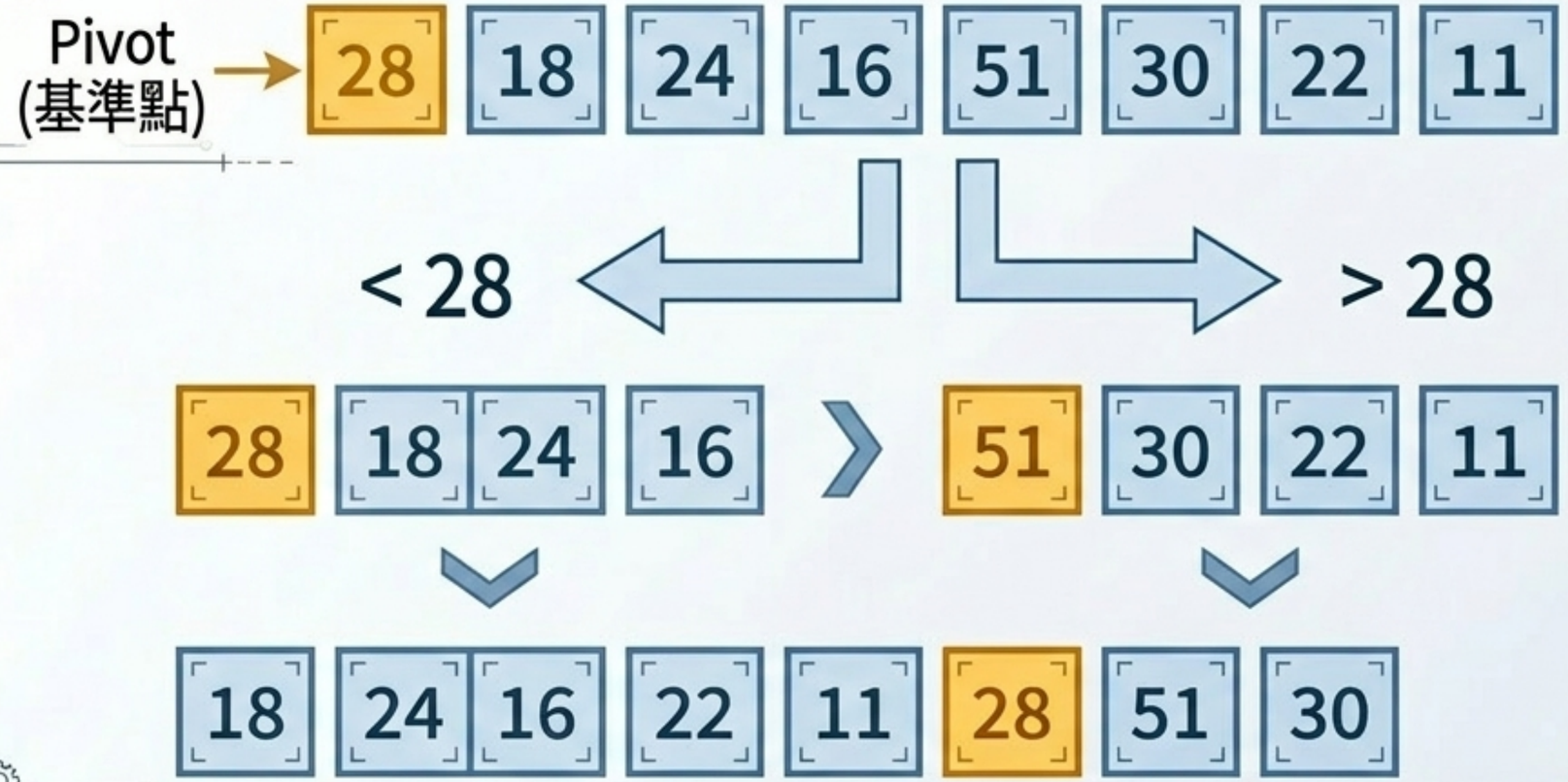
Time Complexity: O(n log n)
Drawback: Requires Additional array
Space Complexity: O(n)



Feynman Callout

「處理不了一大堆？那就切成兩半！(Divide & Conquer)。一直切到只剩一個數字，
然後像拉鍊一樣，把兩邊按順序合併起來。」

空間與速度的極致：快速排序法 (Quick Sort)



Insights

Average Time Complexity: O(n log n)

Space Advantage: O(log n)

(Better in terms of space compared to Merge Sort)

Feynman Callout

「找一個基準點 (Pivot)，比它小的通通站左邊，比它大的通通站右邊。
最厲害的是，它不需要像 Merge Sort 那樣借用一堆額外的空白陣列！」

排序演算法：效能總結與考試必備矩陣

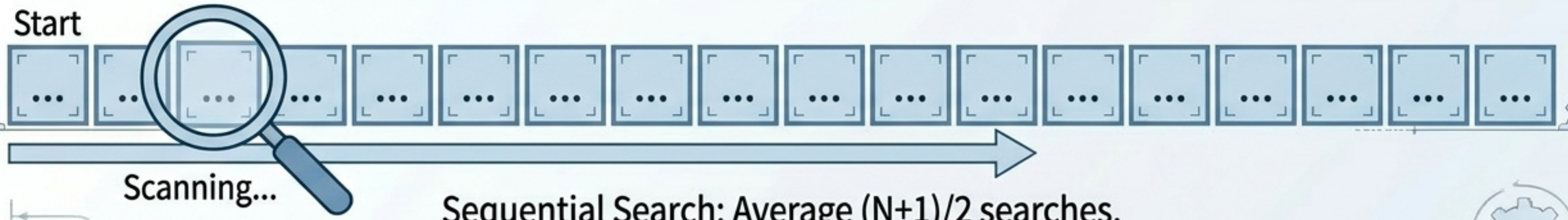
Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Bubble	O(n)	O(n^2)	O(n^2)	O(1)
Selection	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion	O(n)	O(n^2)	O(n^2)	O(1)
Merge	O(n log n)	O(n log n)	O(n log n)	O(n)
Quick	O(n log n)	O(n log n)	O(n^2)	O(log n)

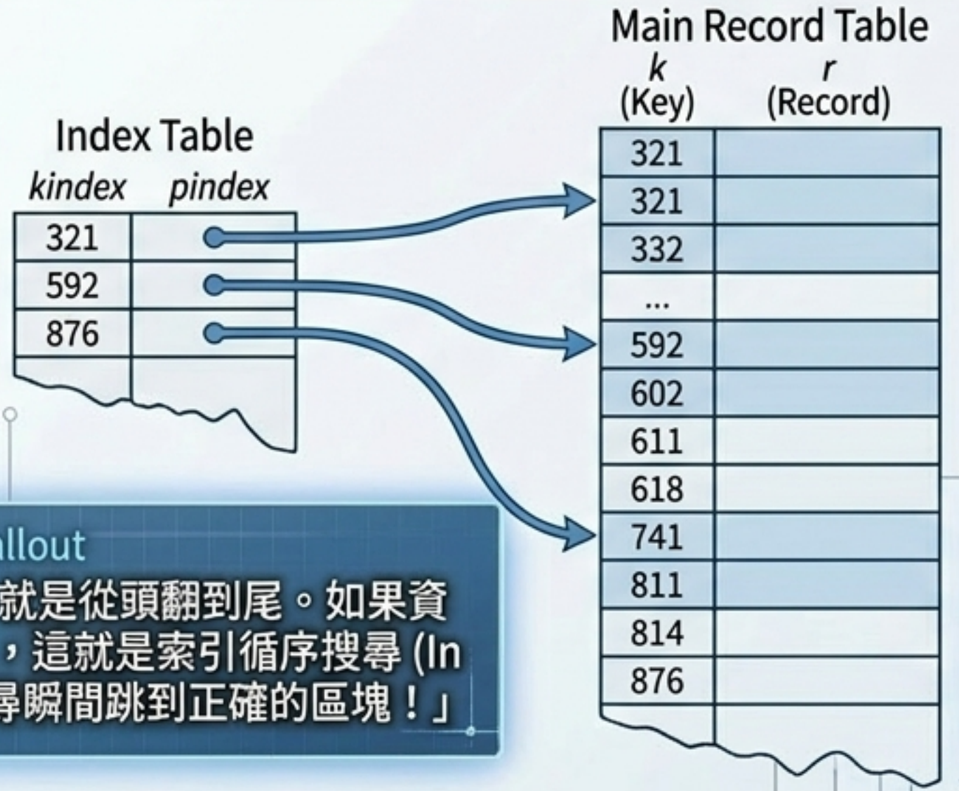
⚠️ **Exam Focus (考試必考點！):**

> **Merge Sort vs. Quick Sort:** 兩者 Average 都是 O(n log n)，但 Quick Sort 空間複雜度極低，實務上更受歡迎。

> **Quick Sort 的致命傷 (Worst Case):** 當資料已經排好序 (already sorted) 時，Quick Sort 退化成 O(n^2)！**(考試常考陷阱)**

基礎搜尋：循序搜尋與索引架構





Feynman Callout

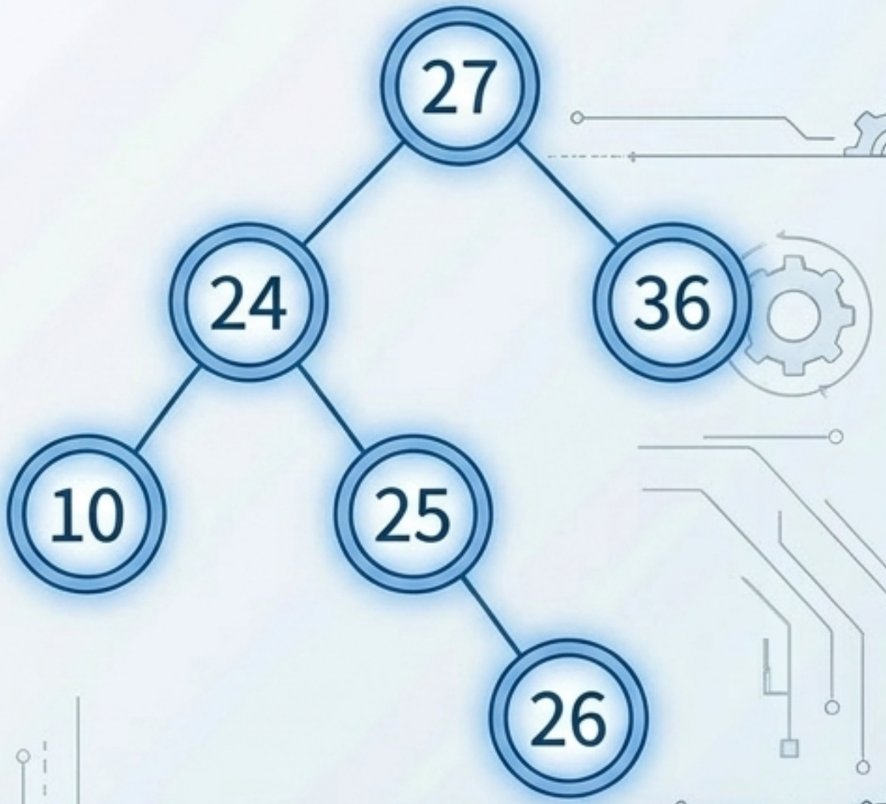
「循序搜尋 (Sequential Search) 就是從頭翻到尾。如果資料太多，我們就建一個『目錄』，這就是索引循序搜尋 (Indexed Sequential Search)，讓搜尋瞬間跳到正確的區塊！」

Key English Terms:

> Sequential Search
> Auxiliary table / Index (輔助表/索引)
> Pointers (指標)

樹狀搜尋：二元搜尋樹 (Binary Search Tree)

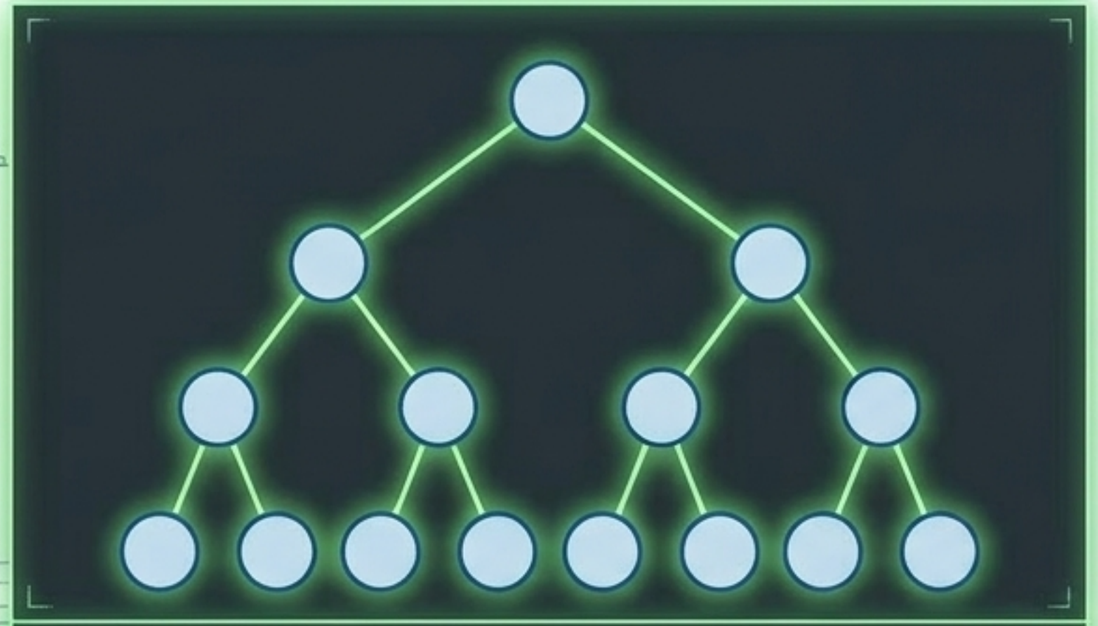
```
public class BinaryTreeNode {
    private int item; // Data
    private BinaryTreeNode left;  // Left Pointer
    private BinaryTreeNode right; // Right Pointer
}
```



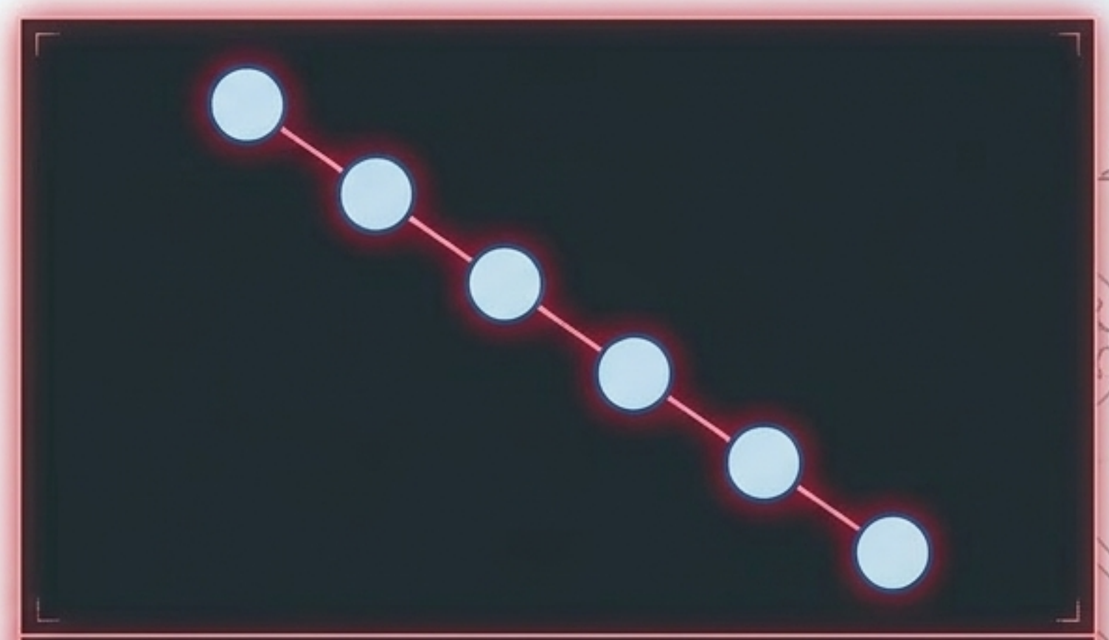
Feynman Callout

「把資料變成一棵樹！規則很簡單：左邊的樹枝永遠小於樹幹，右邊的樹枝永遠大於樹幹。透過 In-order traversal (中序遍歷)，印出來的數字就會是完美排序的！」

二元搜尋樹的致命傷 (The Fatal Flaw of BST)



Best Case: O(log n)
(Balanced)



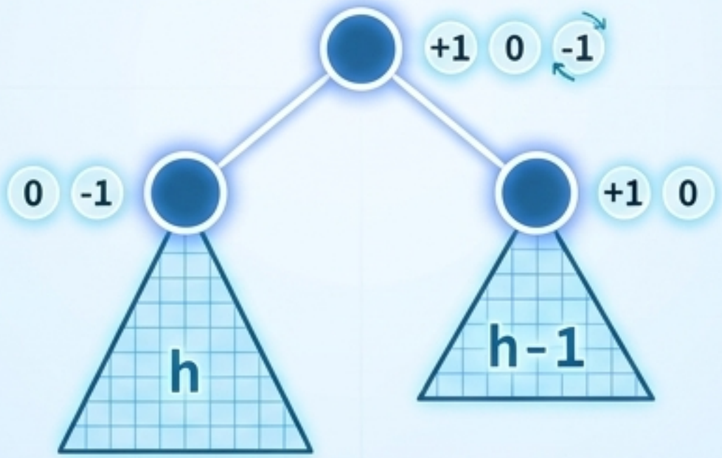
Worst Case: O(n)
(Skewed / Degraded to Linked List)

Feynman Callout

「如果輸入的數字『剛好已經照順序排好』(例如 1, 2, 3, 4, 5)，BST 就不會長出左右分支，而是長成一根長長的竹竿！這時它就退化成普通的 Linked List 了。」

完美平衡的解法：AVL Tree 的誕生

```
Balance_Factor = Height_of_Left - Height_of_Right
```



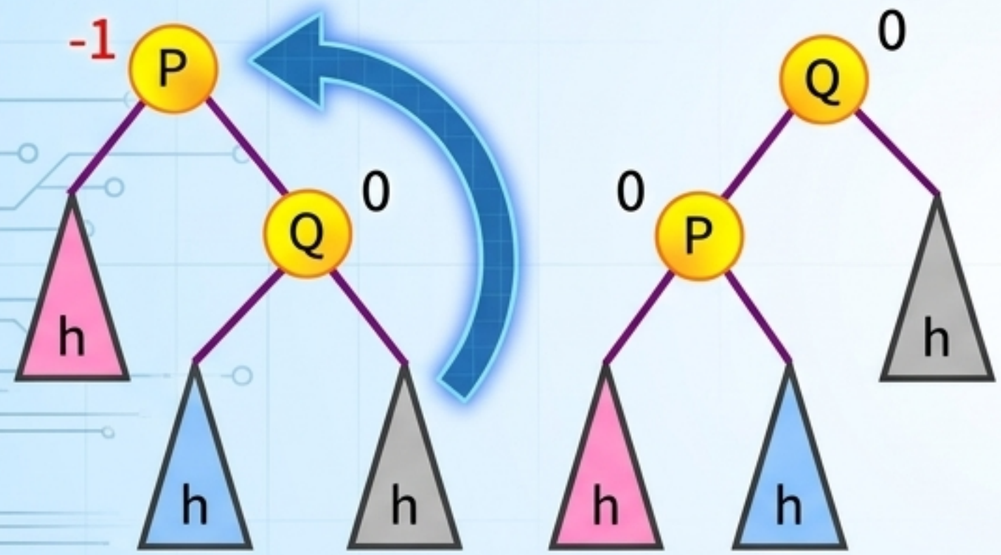
Feynman Callout

「AVL Tree 是一棵『會自我修正』的樹。它強迫每個關節檢查自己的左右樹枝高度。只要高度相差超過 1 (即 Balance Factor 變成 2 或 -2)，它就會立刻啟動『旋轉』來恢復平衡！」

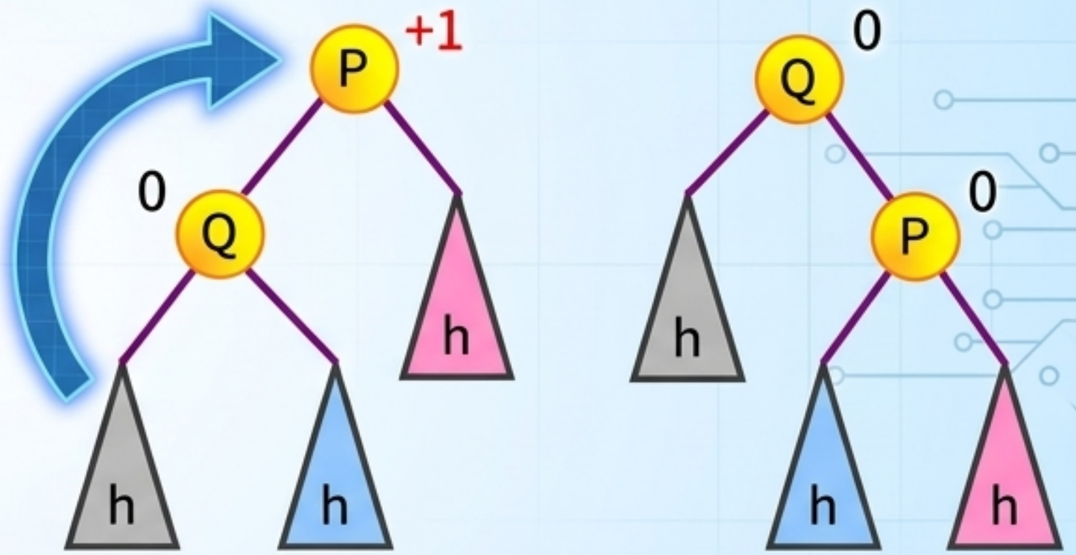
Must be within [-1, 0, 1]
Height-balanced tree

AVL 樹的自我修正 (一)：單旋轉 (Single Rotations)

LL Imbalance -> Right Rotation



RR Imbalance -> Left Rotation

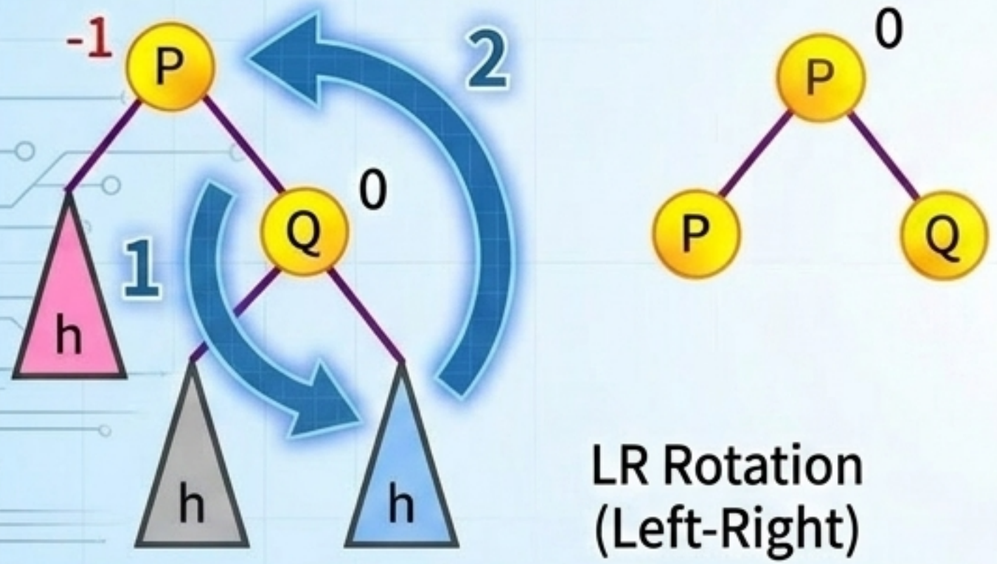


Feynman Callout

如果輸入的數字：「當樹枝往單一方向長得太長（單純的左傾或右傾），我們只需要『把關節往反方向扭一下』。這叫做單旋轉！」

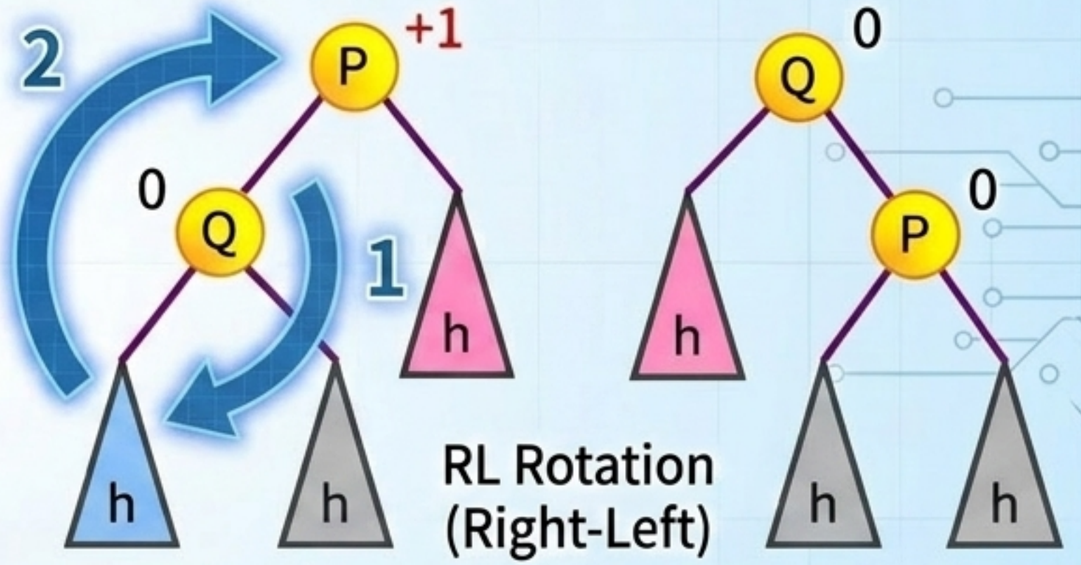
AVL 樹的自我修正 (二)：雙旋轉 (Double Rotations)

LR Imbalance -> Left-Right Rotation



LR Rotation
(Left-Right)

RL Imbalance -> Right-Left Rotation

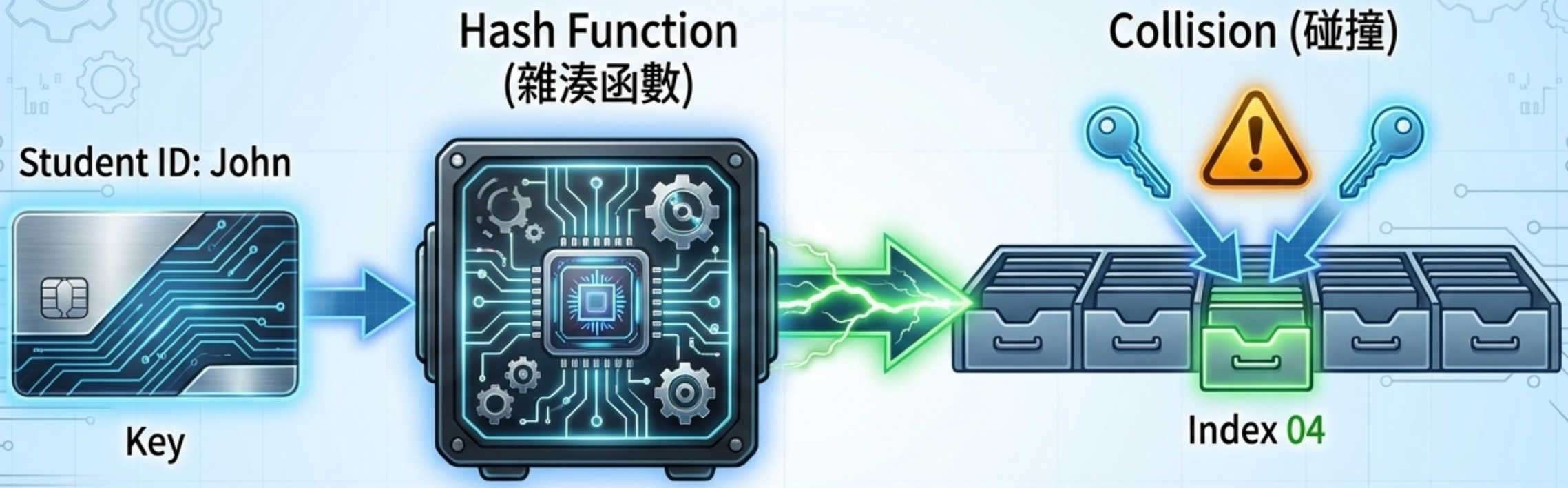


RL Rotation
(Right-Left)

Feynman Callout

「如果樹枝呈現『閃電狀』的彎曲（先左再右，或先右再左），單向旋轉是不夠的！我們必須先把它轉成一直線（第一次旋轉），再把它折平（第二次旋轉）。」

知識增潤：超越樹狀結構的哈希演算法 (Hash Algorithms)



Ultimate Time Complexity: O(1)

Feynman Callout

「如果連 O(log n) 的搜尋時間你都嫌慢，怎麼辦？我們請一位超級數學家 (Hash Function)，你給他名字，他瞬間算出資料在第幾個抽屜！一步到位！」

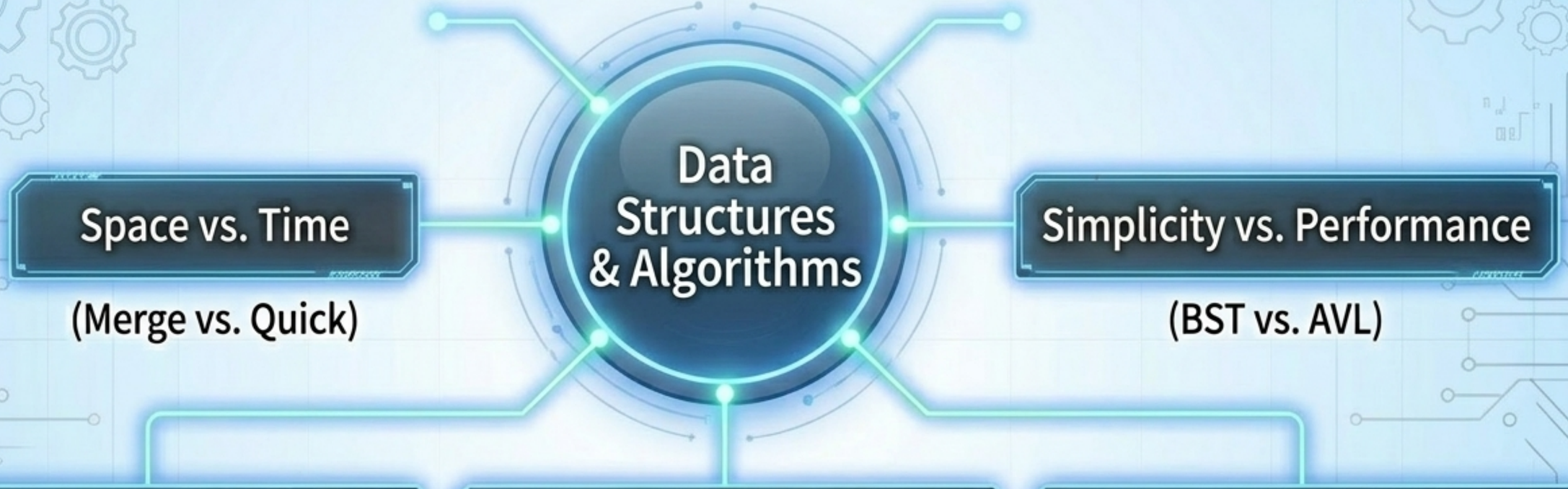
搜尋演算法：效能總結與資料結構矩陣

Algorithm	Data Structure	Best Case	Worst Case
Sequential Search	Array / List	O(1)	O(n)
Binary Search Tree (BST)	Tree	O(log n)	O(n)
AVL Tree	Balanced Tree	O(log n)	O(log n)
Hash Table	Hash Map	O(1)	O(n) (with collisions)

Exam Focus (考試必考題！)：

> BST vs. AVL: AVL 透過旋轉，保證了 Worst Case 永遠是 O(log n)，完美解決了普通 BST 退化成 Linked List 的致命傷。

教授的結語：演算法的終極權衡 (The Ultimate Trade-off)



1. 沒有完美的演算法 (No Silver Bullet)：
Quick Sort 省空間但怕排好的資料；Merge Sort 穩定但吃記憶體。

2. 資料結構決定命運：
為什麼 AVL 要花這麼大力人氣旋轉？因為『維持秩序的成本，遠低於混亂中尋找的代價』。

3. 考試必勝法則：
熟記兩個 Big-O 矩陣，理解 Quick Sort 的 Pivot 分割，並掌握 AVL Balance Factor 的計算。