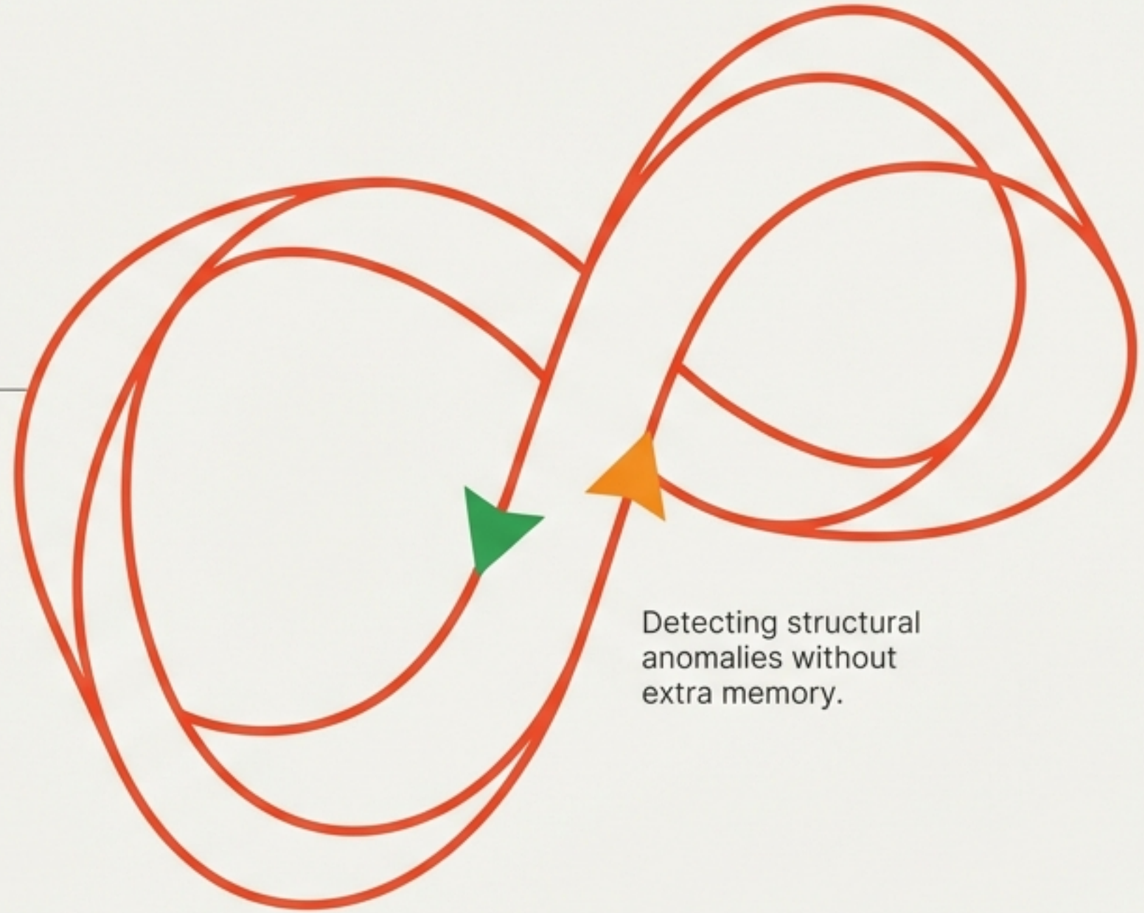
Fast & Slow Pointers Explained

Cracking Linked List Problems in O(N) Time | DSA Pattern #2



Detecting structural anomalies without extra memory.

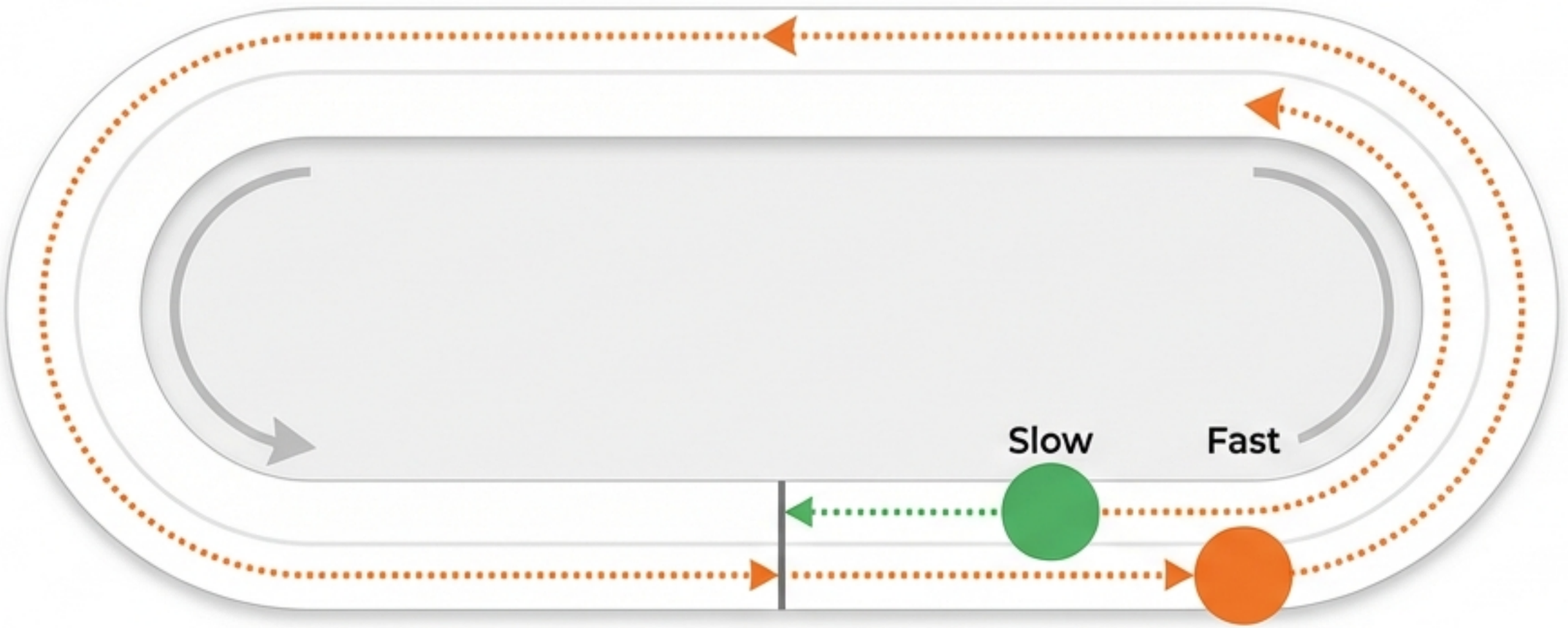
Based on 'Day 7/90' by CTO Bhaiya

The Core Intuition: A Tale of Two Runners

Scenario A: The Straight Path

Gap increases infinitely.
They never meet.

Scenario B: The Circular Track



The Setup:

- Two runners start at the same line.
- **The Speed:**
  Runner A (Slow) takes 1 step.
  Runner B (Fast) takes 2 steps.
- **The Insight:**
  On a linear path, the fast runner vanishes.
  On a cyclic path, they must eventually collide.
  "Life comes to a full circle."

Translating Intuition to Data Structures

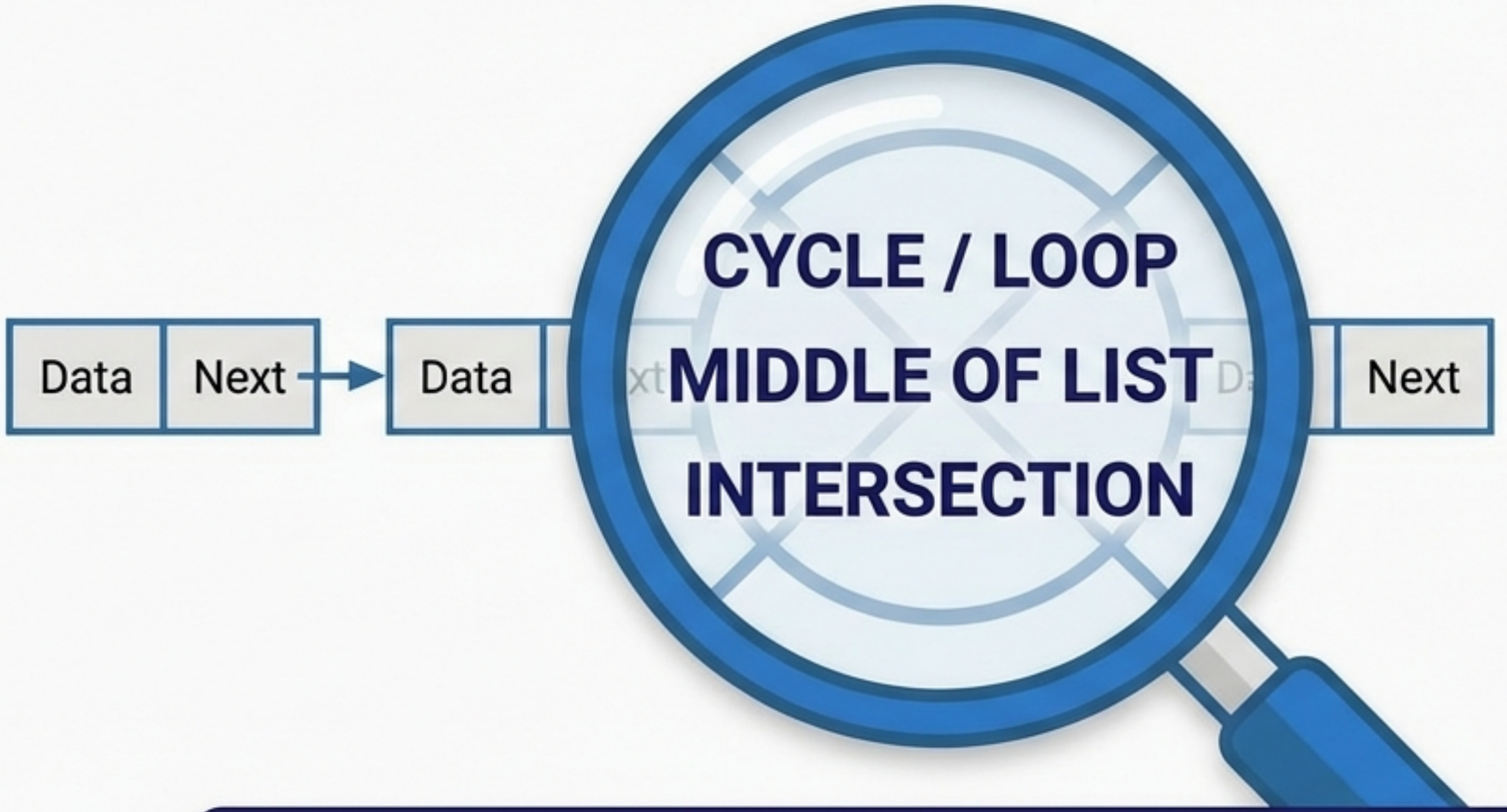
The Real World	The Data Structure	The Implementation
Track	Linked List / Array / String	Linear Traversal
Runners	Pointers (Variables)	slow = head, fast = head
Steps	node.next or index + 1	Logic below:

Slow Pointer:
Moves 1 step
(slow = slow.next)

Fast Pointer:
Moves 2 steps
(fast = fast.next.next)

When to Deploy This Pattern?

Pattern Recognition Signals



Look for linear structures with "weird" properties:

- "Determine if the list ends or loops..."
- "Find the midpoint in one pass..."
- "Find the start of a duplicate sequence..."

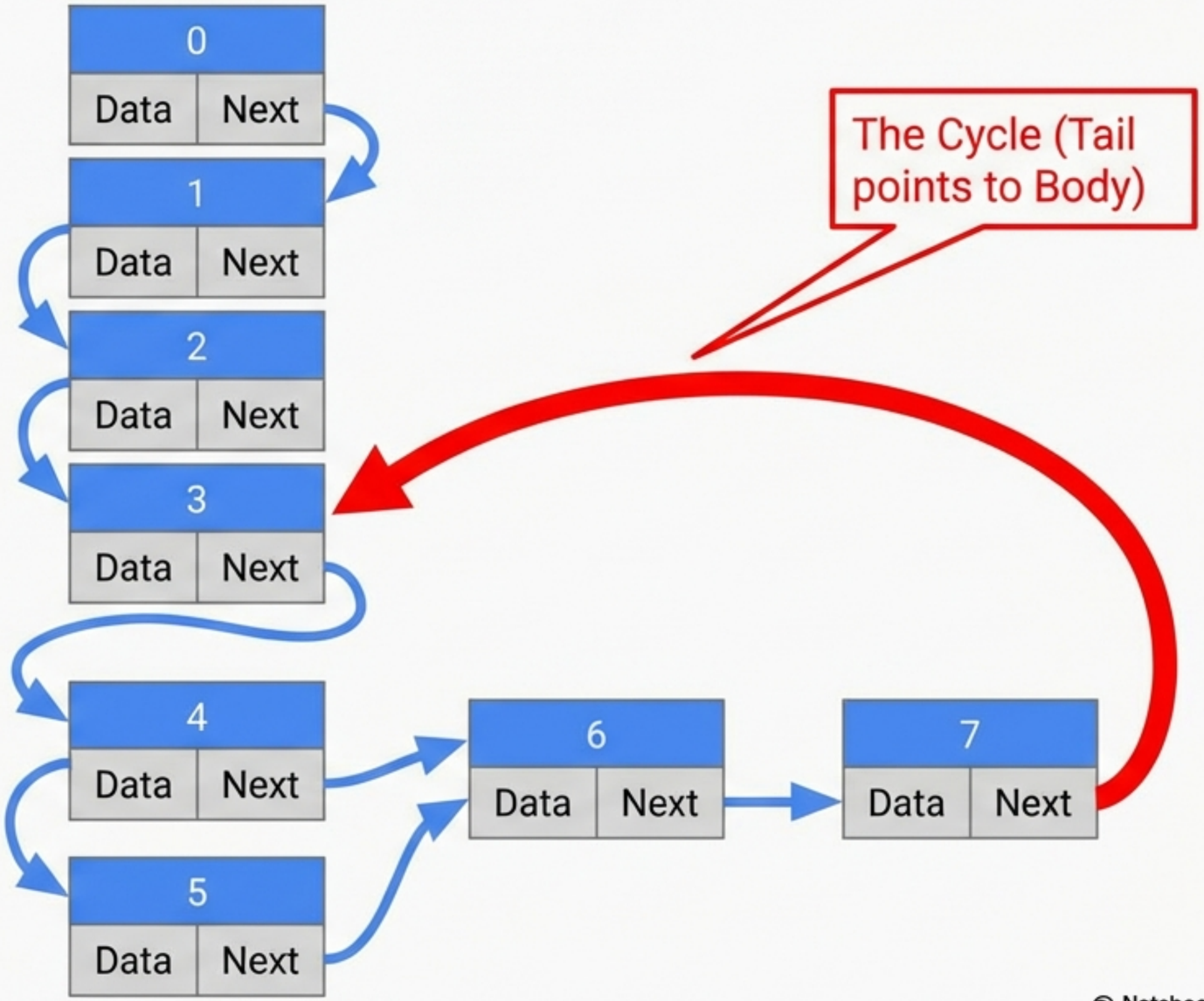
The Value Proposition: Why use this? It detects these anomalies with Space Complexity O(1). No HashSets required.

The Challenge: Cycle Detection

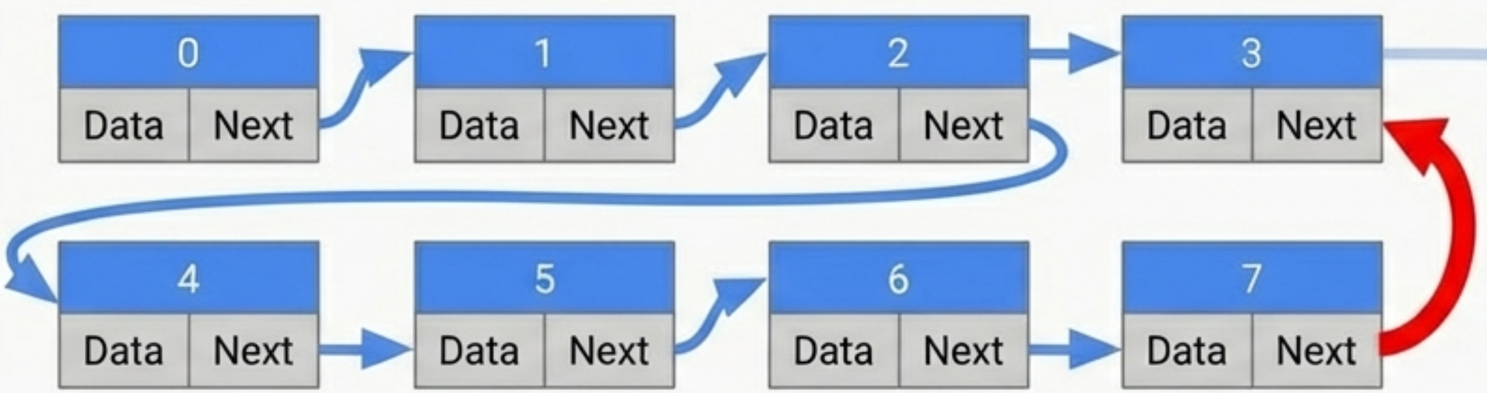
The Scenario: You are given the “head” of a Linked List.

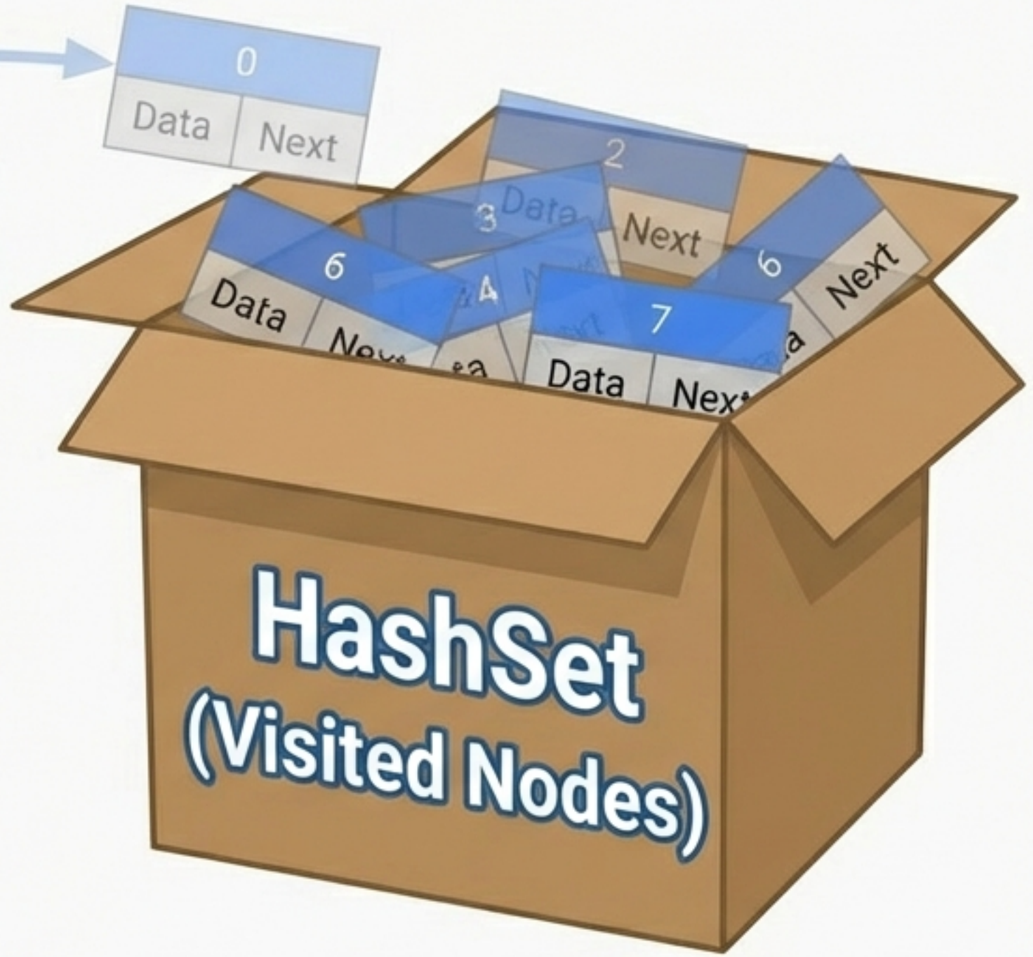
The Question: Does this list contain a cycle?

The Consequence: Standard traversal looks for NULL. In this structure, NULL does not exist. A standard loop will run **forever until the program crashes.**



The Brute Force Approach (and its Flaw)





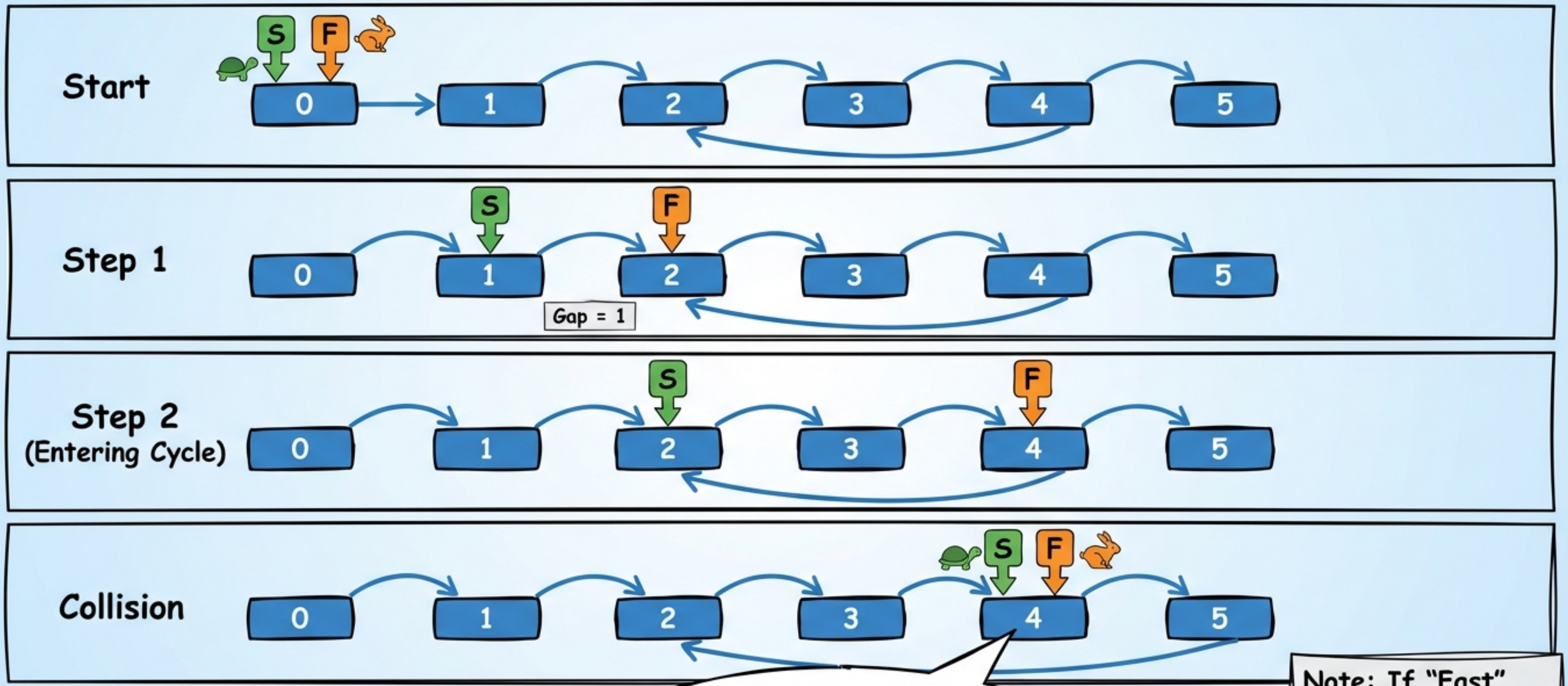
Logic: Store every visited node in a Set.
If we see a duplicate, a cycle exists.

Analysis: - Time Complexity: O(N) (Good)
- Space Complexity: O(N) (Bad)

The problem: We are storing a reference to **every single node**. This scales linearly with input size.

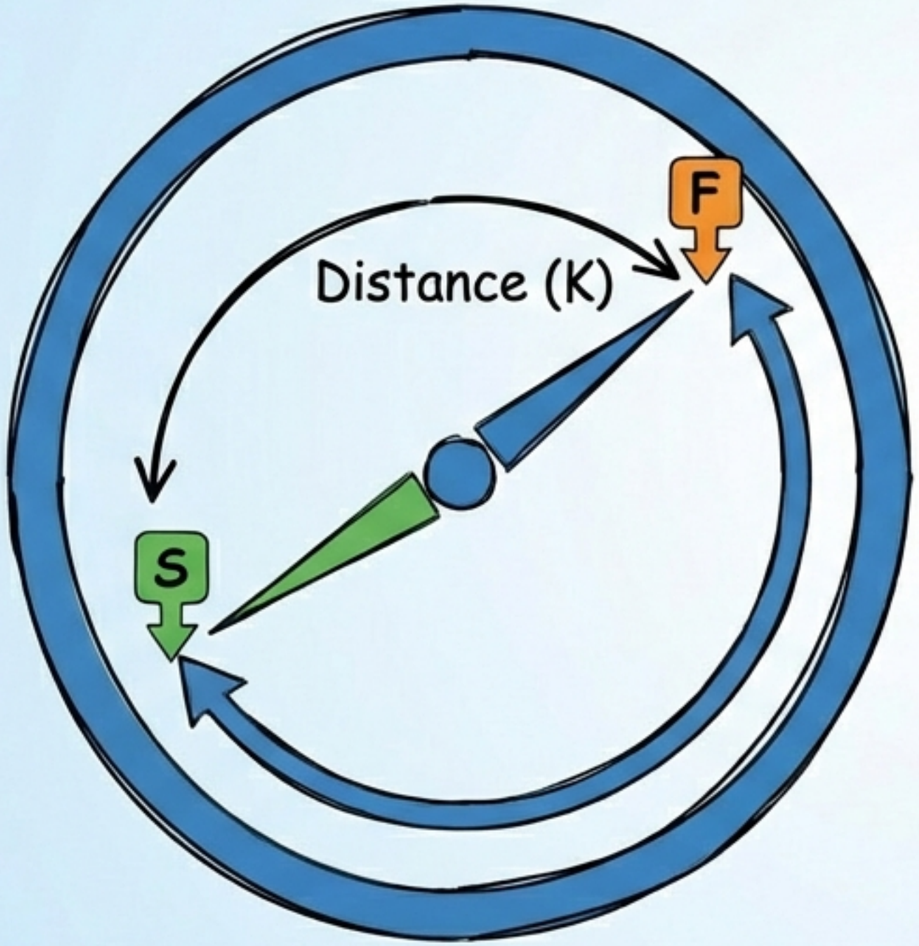
The Goal: Can we do this with O(1) space?

The Algorithm in Motion



The Math: Why 2 Steps Guarantee a Meeting

Why doesn't the Fast pointer hop over the Slow pointer forever?



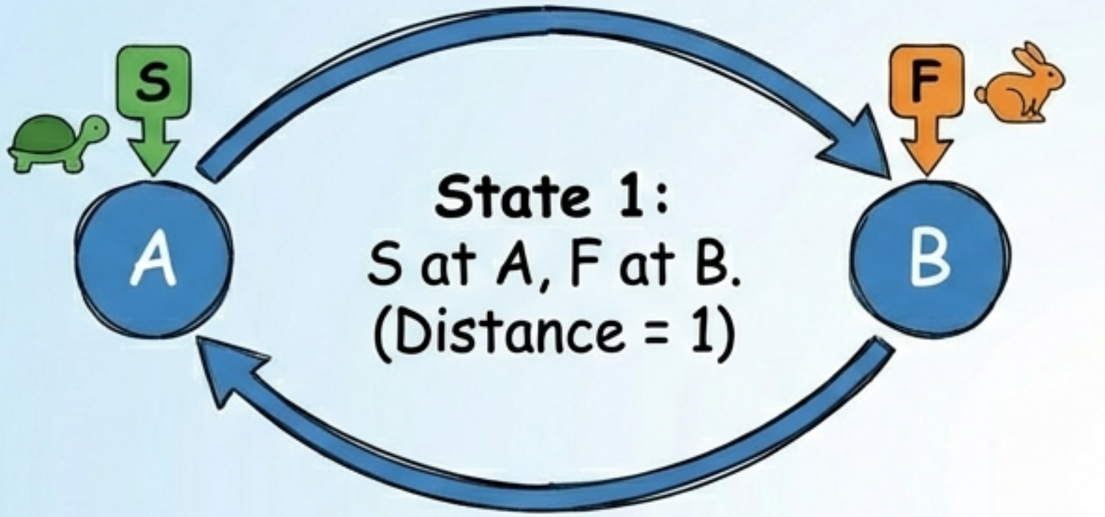
Relative Speed

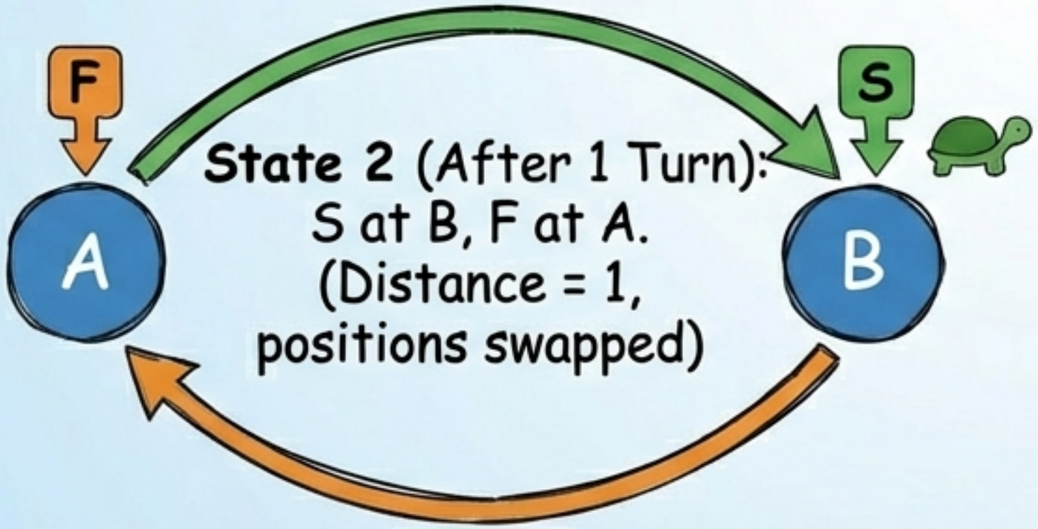
Fast Speed = 2 units/step
Slow Speed = 1 unit/step
Relative Speed = (2 - 1) = 1 unit/step

Gap Reduction Logic:
Iteration 0: Distance = K
Iteration 1: Distance = K - 1
Iteration 2: Distance = K - 2
...
Iteration K: Distance = 0 (Collision)

Note: Since the gap reduces by exactly 1 every step, they cannot miss each other.

Why Not 3 Steps?





Relative Speed & Gap Logic:

- If Fast moves 3 steps, Relative Speed = 2.
- The gap reduction is 2 steps per turn.
- If the gap is an odd number (e.g., 1), the gap logic becomes: 1 -> -1.
- The Fast pointer physically "hops over" the Slow pointer without landing on the same node, potentially missing the collision or delaying it indefinitely.

The Code Template: Cycle Detection

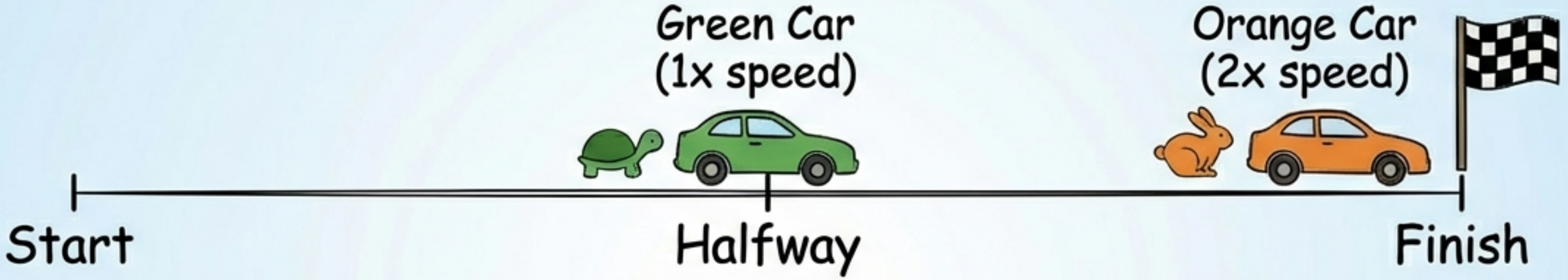
```python
def hasCycle(head):
    slow = head
    fast = head

    while fast is not None and fast.next is not None:
        slow = slow.next          # 1 Step
        fast = fast.next.next     # 2 Steps

        if slow == fast:
            return True           # Cycle Found

    return False                  # Reached End (No Cycle)
```

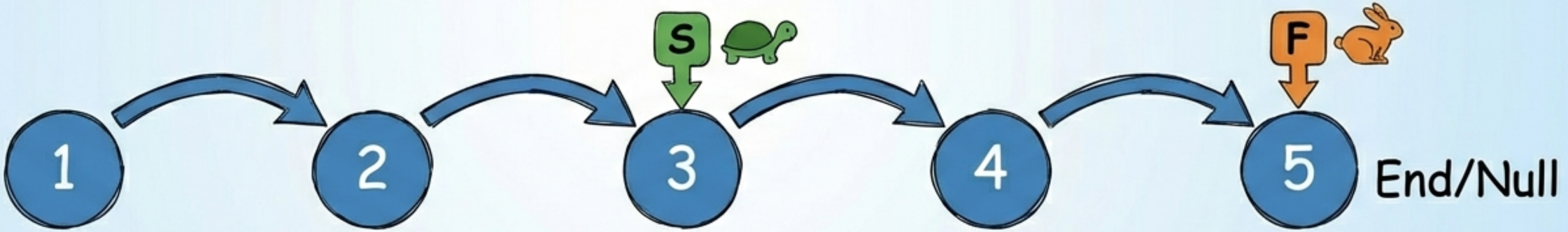
Safety Check: Must verify fast.next is not null before jumping 2 steps.

If loop breaks, the list is linear.

Application: Finding the Middle

The 10mph vs. 20mph Analogy



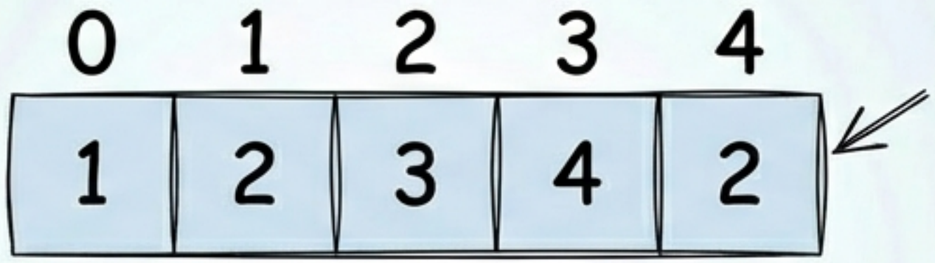


The Trick: Run the pointers. When Fast reaches the end, Slow is guaranteed to be at the middle.

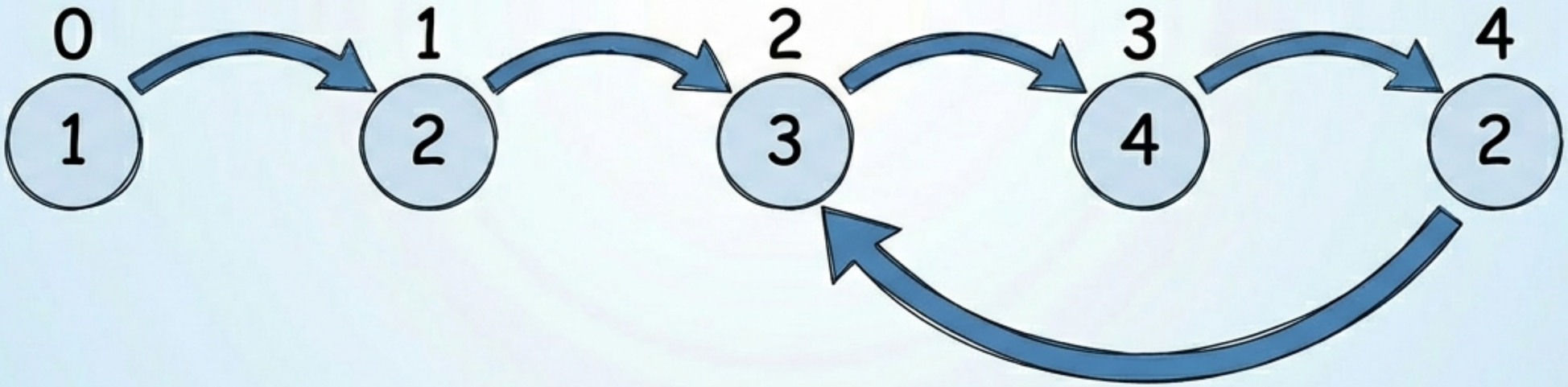
Use Case: Splitting lists for Merge Sort.

Application: Cycles in Arrays

We can treat Arrays as Linked Lists if the value at an index tells us the next index to visit.



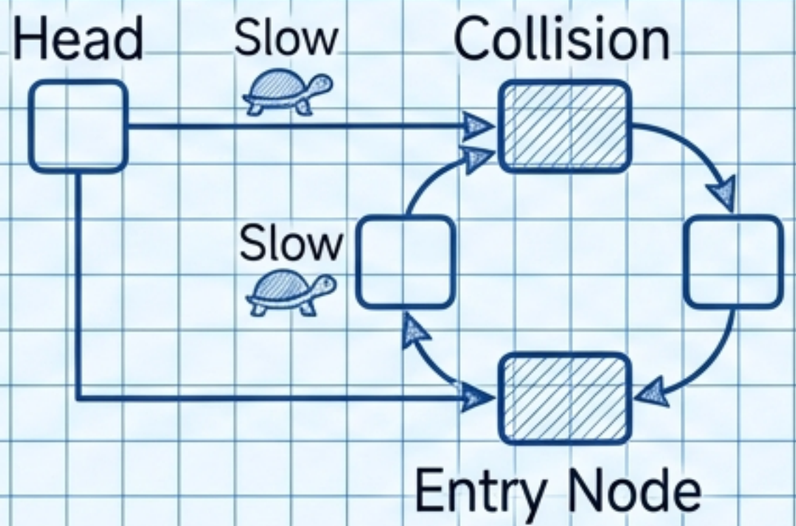
Index = Node Address.
Value = Next Pointer.



Logic: fast = arr[arr[index]], slow = arr[index].

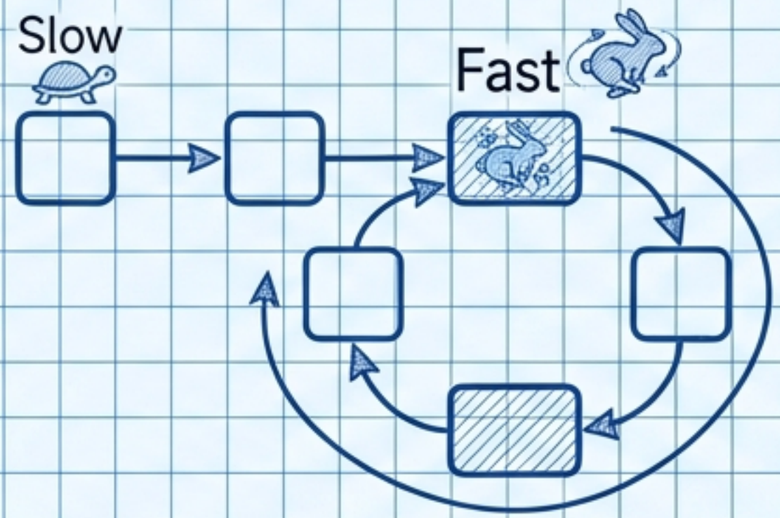
Advanced Variations

Finding the Cycle Start



Reset one pointer to Head.
Move both 1 step at a time.
Collision = Entry Node.

Calculating Cycle Length

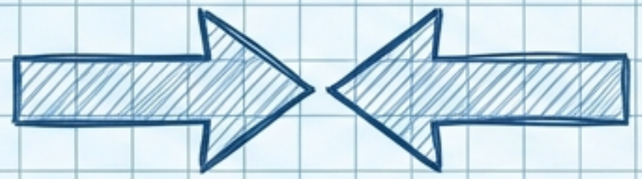
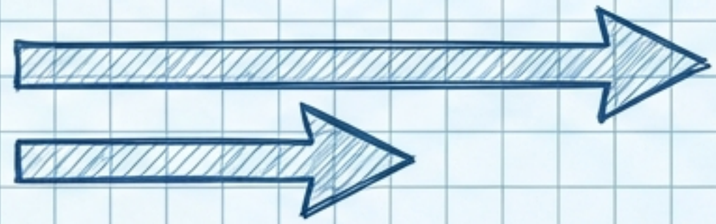


Freeze **Slow**.
Move **Fast** until it returns to **Slow**.
Count the steps.

Mathematical proofs for these variations are available in the full course material.

Pattern Distinction

Fast & Slow Pointers vs. Standard Two Pointers

Standard Two Pointers	Fast & Slow Pointers
	
• **Direction:** Opposite (Start & End) • **Movement:** Conditional (based on sums/values) • **Use Cases:** Sorting, Palindromes, Target Sums	• **Direction:** Same Direction • **Movement:** Fixed Relative Speed (Blind traversal) • **Use Cases:** Structural Discovery (Cycles, Middle, Intersections)

Summary & Next Steps

- **Time Complexity:** O(N) (Linear Traversal)
- **Space Complexity:** O(1) (Two variables only)
- **The Key:** Manipulate relative speeds to force collisions or ratio-based positioning.

"If you keep running in a circle, you are bound to meet again. Life comes to a full circle."

Next Up: Applying the template to Problem Set 1-6 (Medium/Hard).