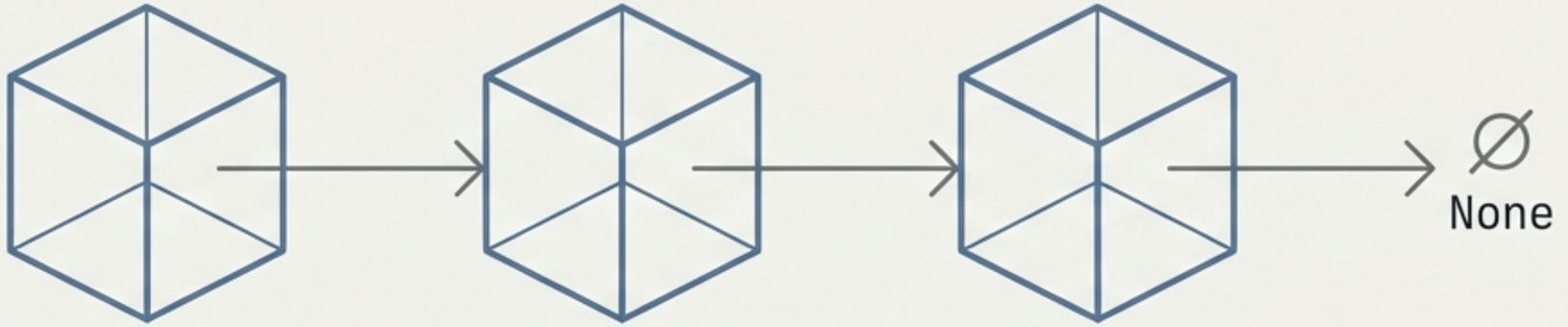


Data Structures & Algorithms: The Interview Essentials

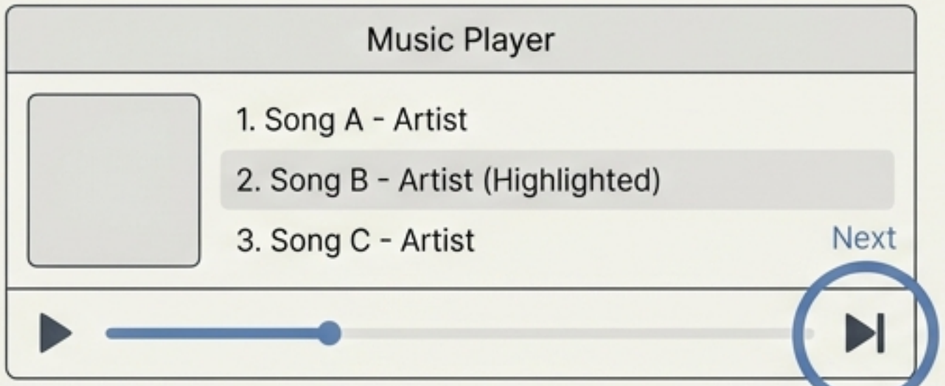
Mastering Singly Linked Lists & The Two Pointer Pattern


```
curriculum_source: 'Chai aur Code' + 'CTO Bhaiya' // focus: visual_intuition
```

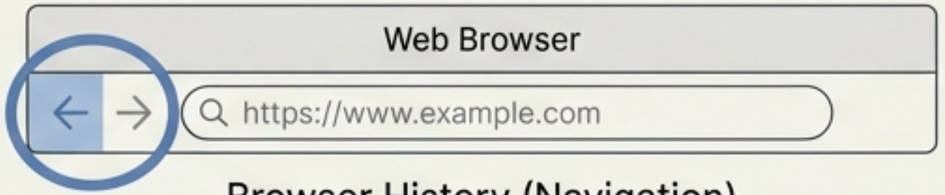
Logic Over Syntax: Why These Structures Matter

The **Linked List** is the first topic where an interviewer tests logical reasoning rather than just syntax.

Real World Application



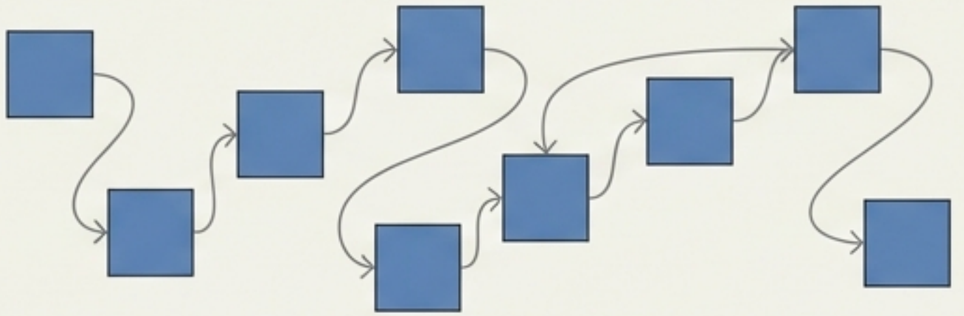
Spotify Playlist (Queueing)



Browser History (Navigation)

The Memory Constraint

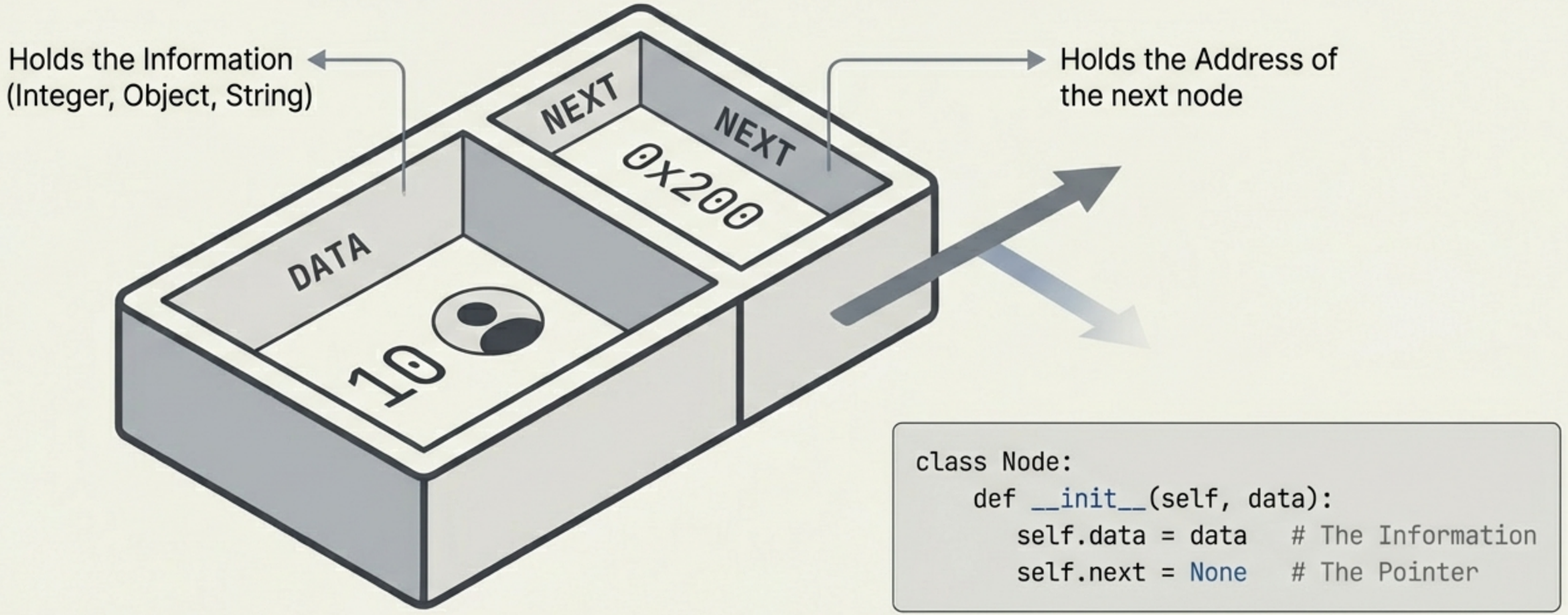
Array (Continuous Memory - Fixed)



Linked List (Non-Continuous Memory - Dynamic)

Unlike Arrays, Linked Lists utilize scattered memory blocks. They allow for dynamic allocation at runtime—you can grow the structure indefinitely without pre-defining the size.

The Anatomy of a Node

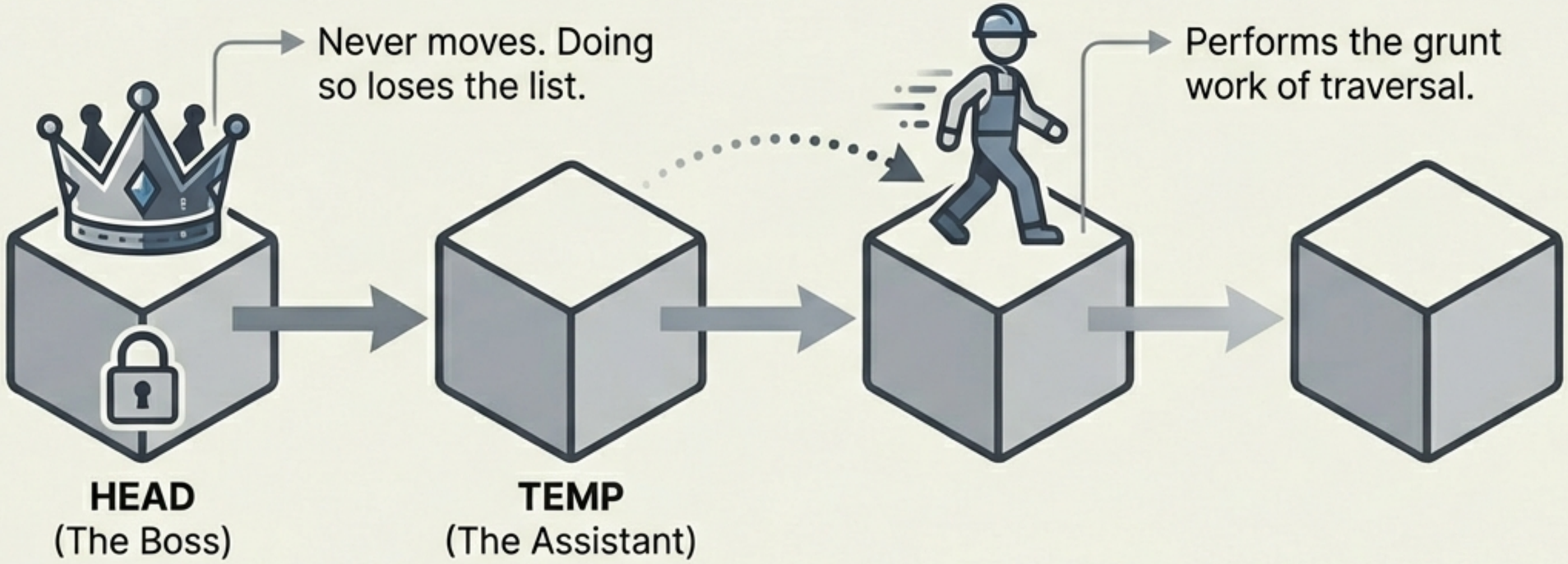



```
class Node:
    def __init__(self, data):
        self.data = data   # The Information
        self.next = None   # The Pointer
```

A Linked List is a collection of these Nodes scattered in memory, connected only by reference addresses. If you lose the reference, you lose the connection.

The Golden Rule of Traversal

The Boss and The Assistant




```
temp = head  # Assign Assistant
while temp is not None:
    print(temp.data)
    temp = temp.next  # Move Assistant forward
```

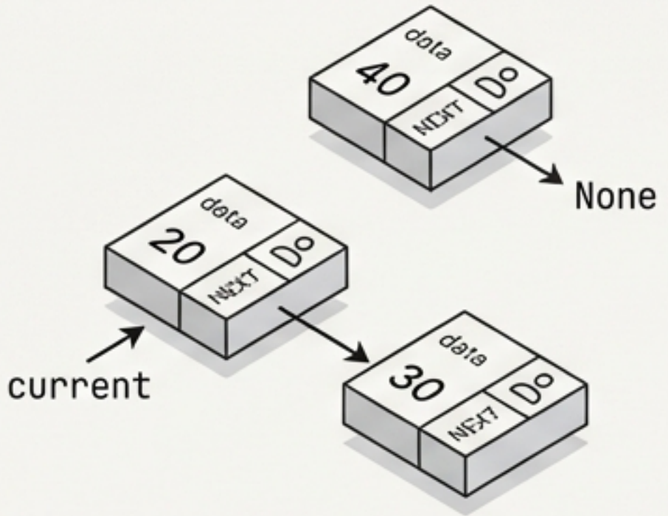
Key Insight

There is no backward movement in a Singly Linked List. Once "temp" moves forward, it cannot look back. Always preserve the Head.

Insertion Mechanics: Stitching the Chain

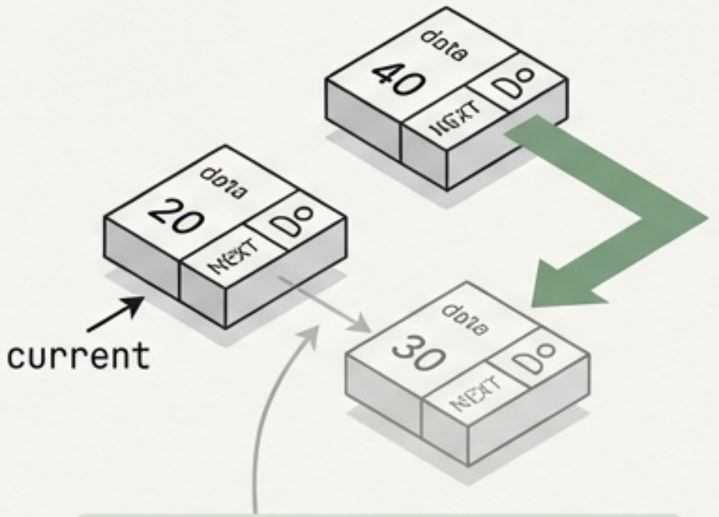
Inserting a Node (40) between Node (20) and Node (30)

State 1: Isolation



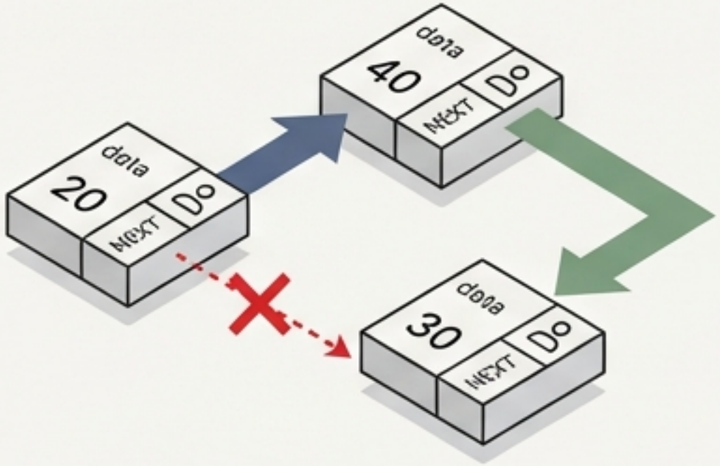
Existing chain: Node[20] → Node[30].
New Node[40] is floating above, unconnected.
Pointer `current` is at Node[20].

State 2: The Safe Link (Step 1)



1. `new_node.next = current.next`

State 3: The Switch (Step 2)



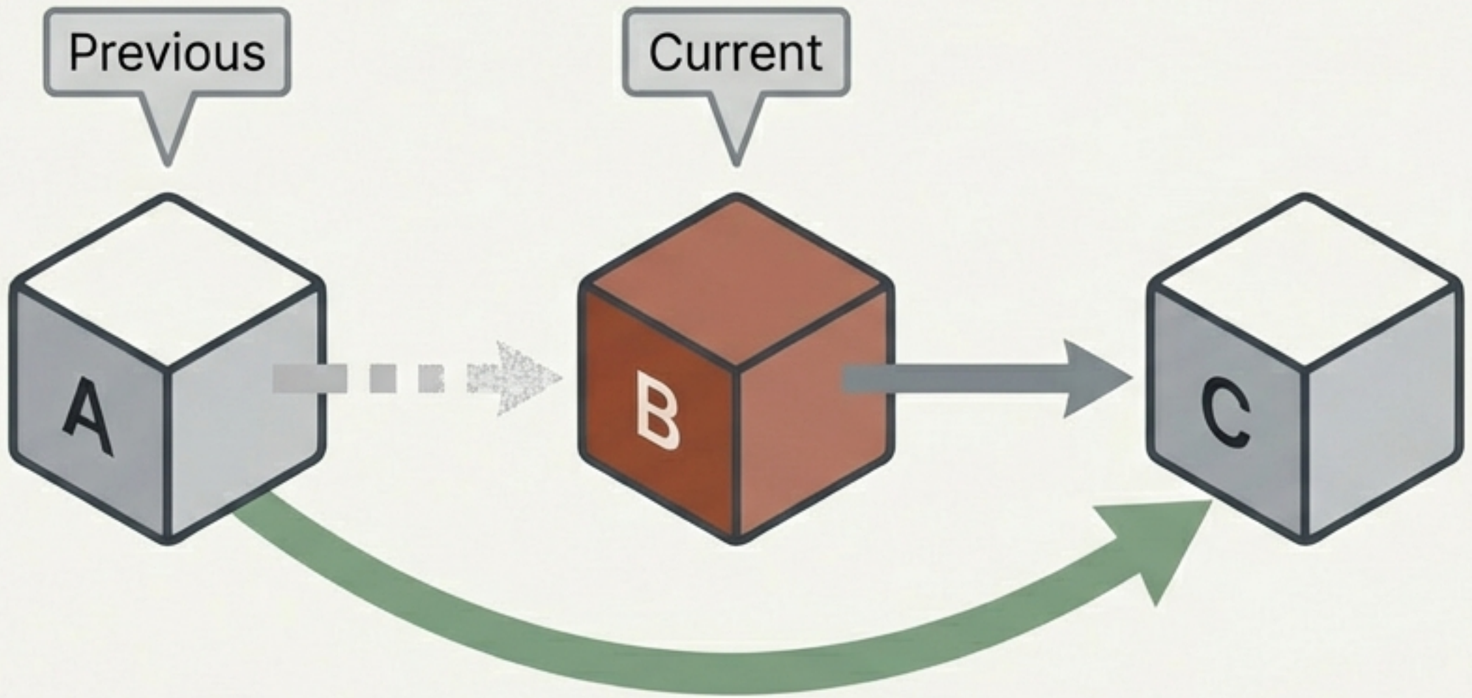
2. `current.next = new_node`

CRITICAL: Order Matters.

If you connect Node 20 to 40 *before* connecting 40 to 30, the address of 30 is overwritten and the rest of the list is lost forever.

Deletion Mechanics: Bridging the Gap

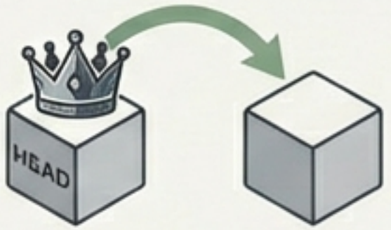
Since we cannot move backward, we must maintain a pointer to the Previous node to re-stitch the chain around the deleted node.




```
previous.next = current.next
```

Handling the Extremes

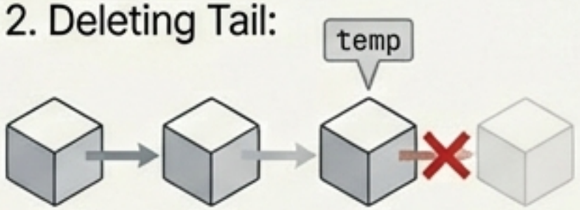
1. Deleting Head:



`head = head.next`

(The boss passes the crown to the second in command).

2. Deleting Tail:



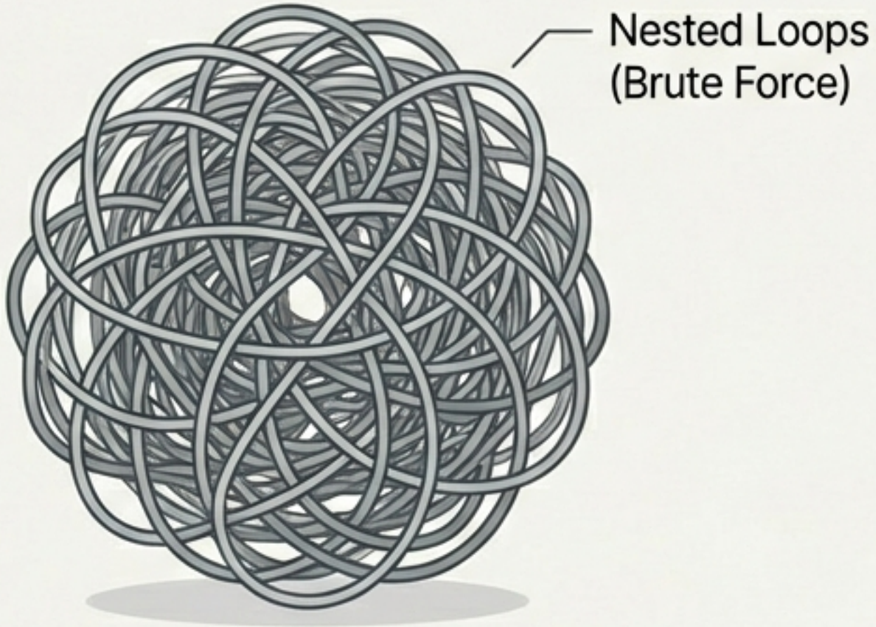
Iterate until `temp.next` is the last node.

`temp.next = None`

Act II: The Two Pointer Strategy

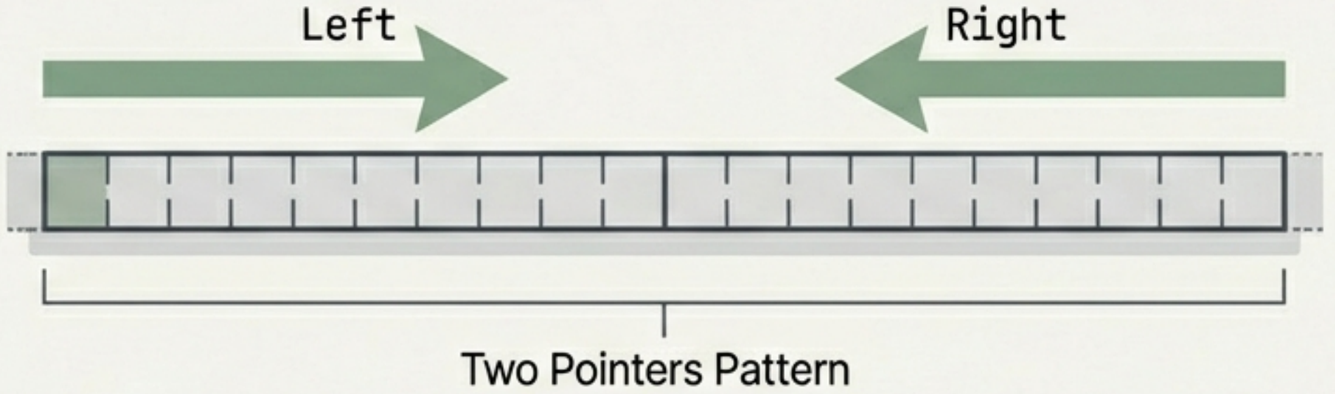
The Ultimate Optimization for Linear Structures

The Quadratic Struggle $O(n^2)$



Checks every pair against every other pair. Slow and inefficient.

The Linear Flow O(n)

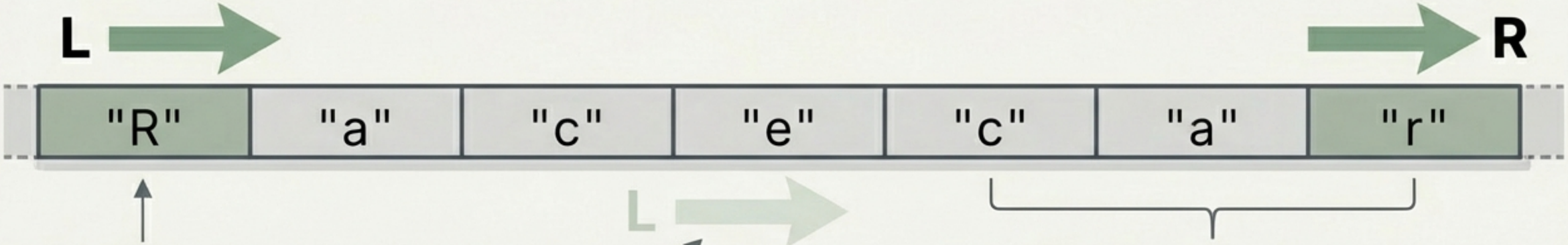


Two variables working in tandem to solve constraints in a single pass.

Applies to Arrays, Strings, and Linked Lists. This pattern typically reduces Time Complexity from $O(n^2)$ to $O(n)$ while maintaining Constant Space $O(1)$.

Pattern 1: The Convergence

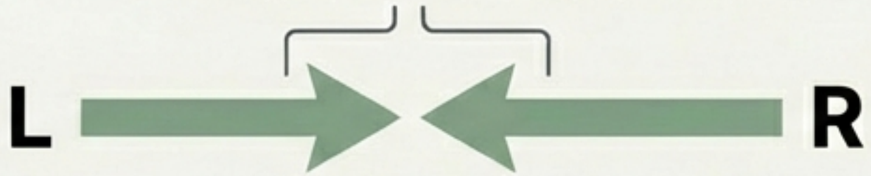
Problem: Valid Palindrome



1. Initialization: L pointer at index 0 ('R'), R pointer at last index ('r').

2. The Filter: while !isalnum(s[L]): L++ (Skip non-alphanumeric - Not applicable to 'Racecar' example but essential for general cases like 'A man, a plan...')

3. The Comparison: Lower('R') == Lower('r')? Match!

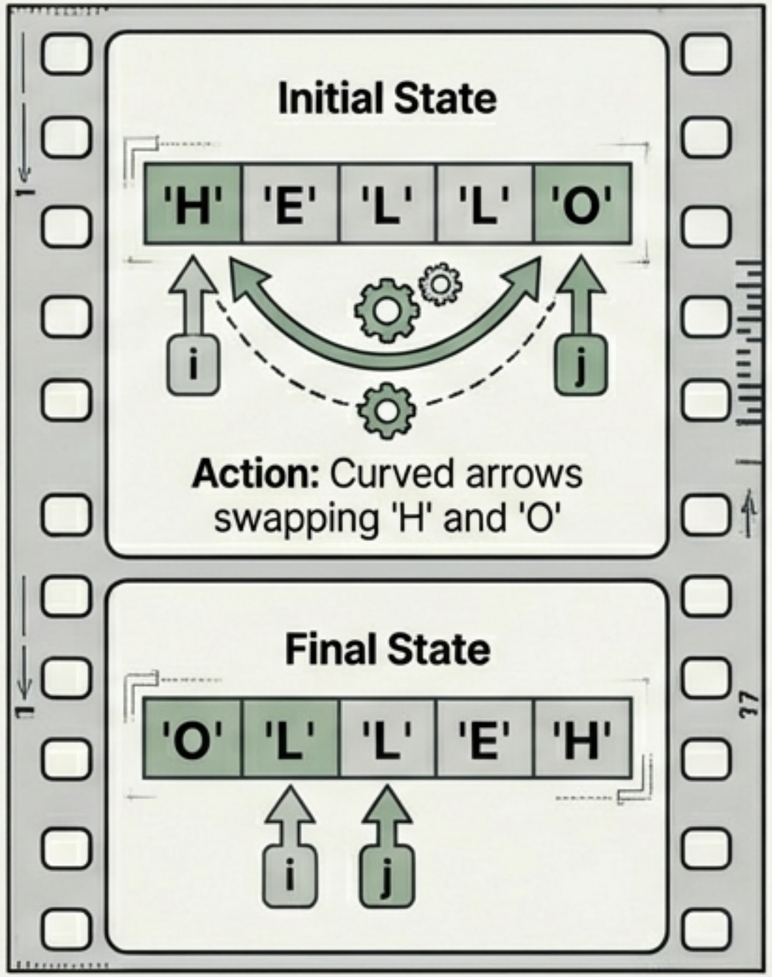


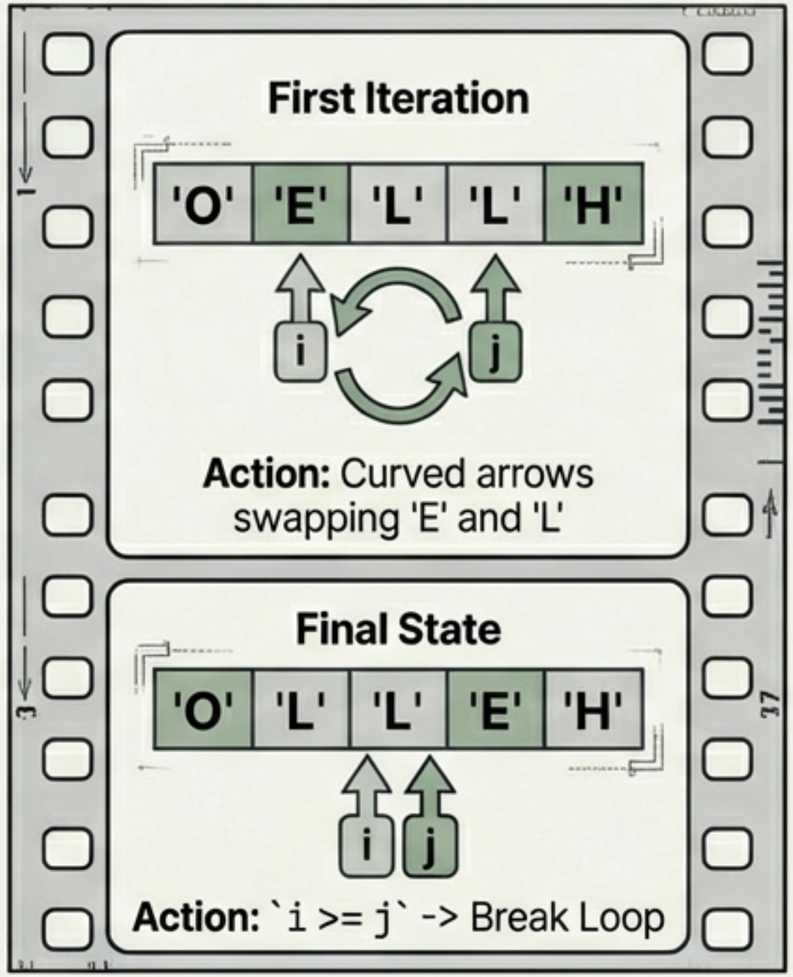
Arrows move inward. Match! Continue...

```
while left < right:
    if s[left] != s[right]: return False
    left += 1; right -= 1
return True
```

Pattern 2: The Swap

Problem: Reverse String In-Place O(1)





Code Logic (Right Side):

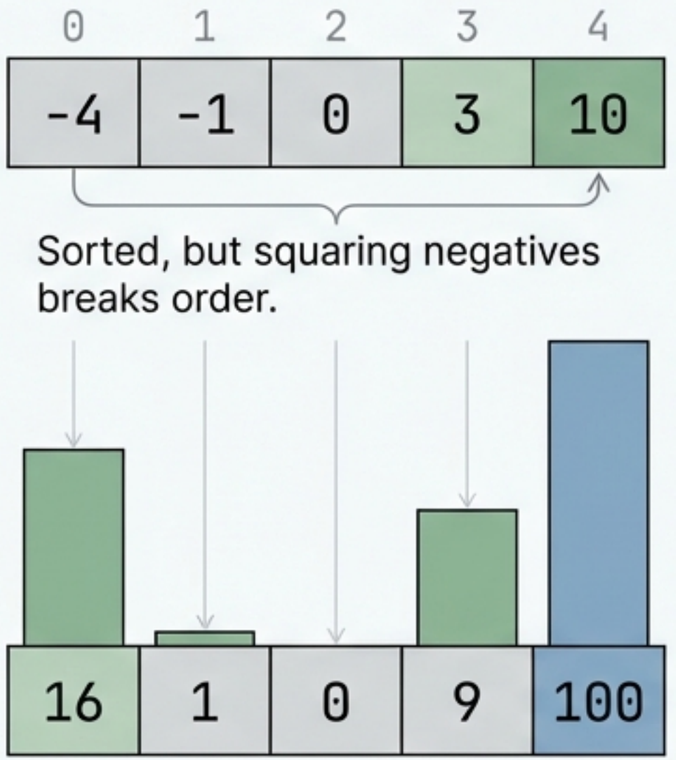
```
temp = s[i]
s[i] = s[j]
s[j] = temp
i += 1
j -= 1
```

In-place swap using a temporary variable, followed by pointer convergence.

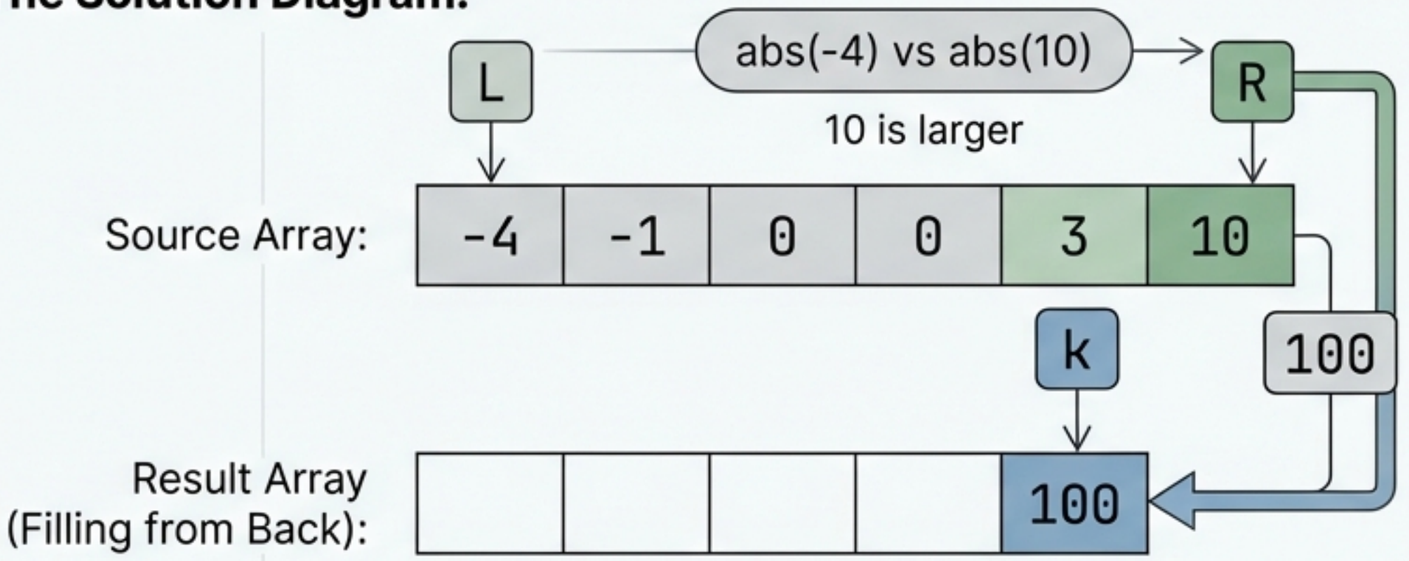
Pattern 3: The End-Fill Strategy

Problem: Squares of a Sorted Array

The Challenge Visual:



The Solution Diagram:

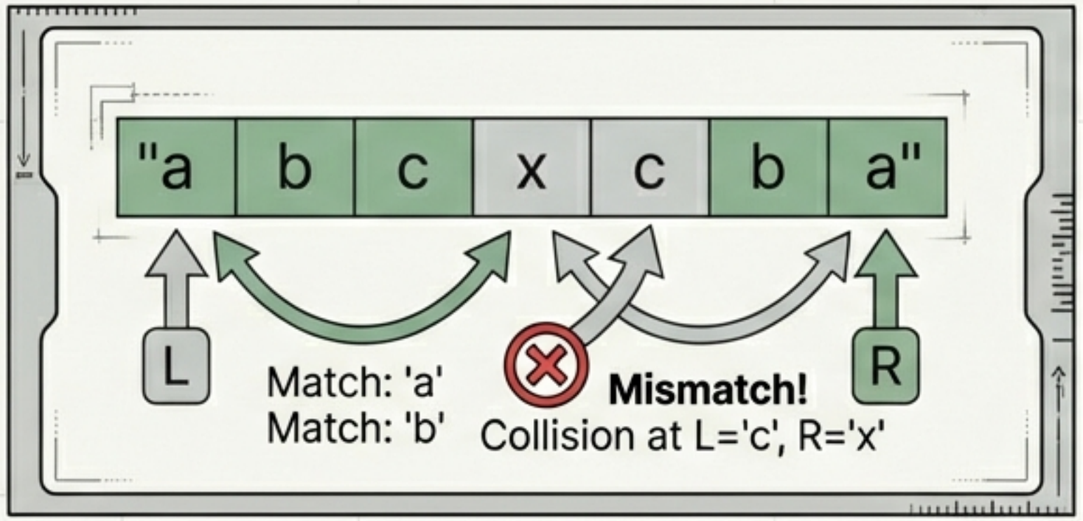


Key Insight:

The largest squares are always at the extremes (Left or Right).
Compare the ends, pick the winner, fill the result from the back.

Pattern 4: The Superpower

Problem: Valid Palindrome II (Delete at most one char)



The Split
(The Superpower)

Timeline A (Delete Left)

"a b ~~x~~ c b a"

x c b"

Check rest: 'x' vs 'b' - Not Palindrome

False

Timeline B (Delete Right)

"a b c ~~c~~ b a"

"c"

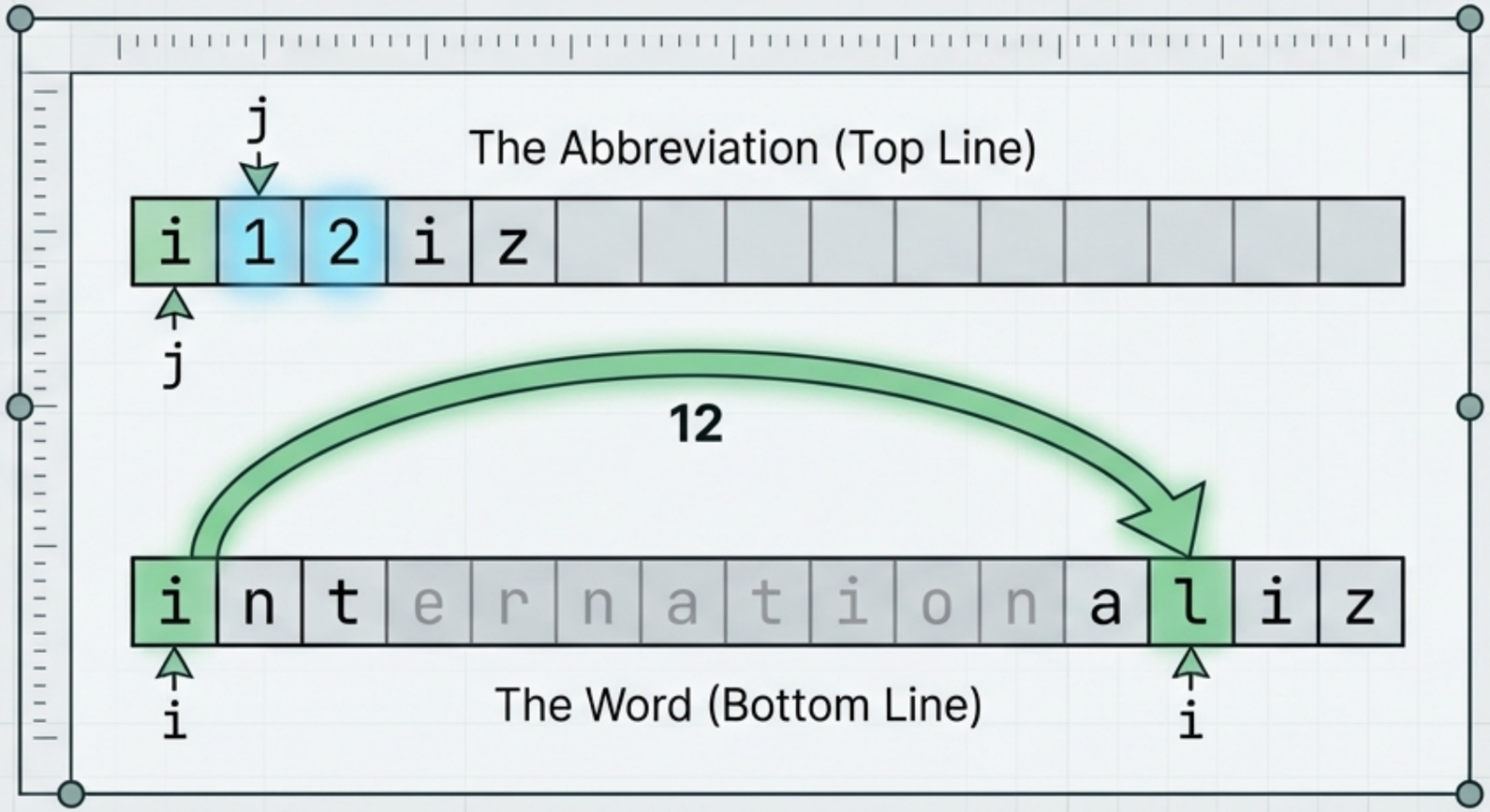
Check rest: 'c' vs 'c' - Valid Palindrome!

True

"You get **one superpower**: When a mismatch occurs, **split reality**. If either timeline is a valid palindrome, the answer is **True**."

Pattern 5: The Complex Parse

Problem: Valid Word Abbreviation (e.g., 'i12iz' vs 'internationaliz') in Inter



Logic Steps:

1. **Match:** chars are equal -> advance both.
2. **Digit:** Parse full number (e.g., 12).
3. **Jump:** `i += number`.

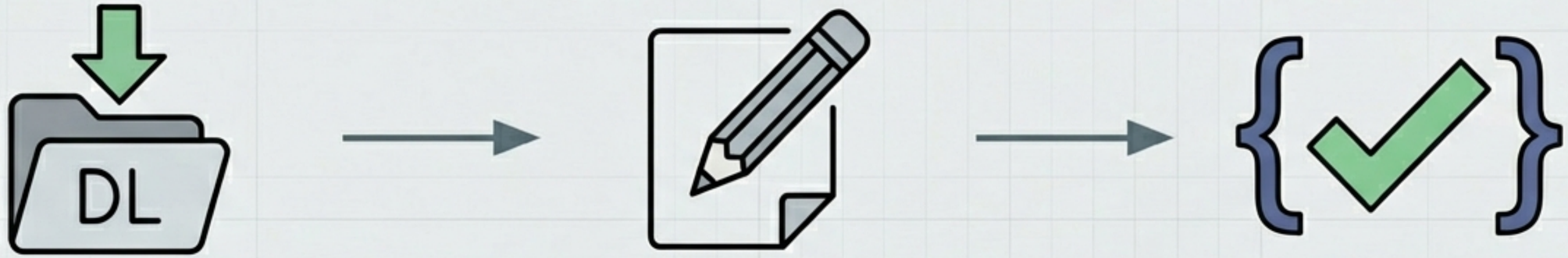
Complexity Cheat Sheet

Algorithm / Problem	Strategy	Time Complexity	Space Complexity
Linked List Traversal	Boss (Head) & Assistant (Temp)	O(n)	O(1)
Valid Palindrome	Two Pointers (Converge)	O(n)	O(1)
Reverse String	Two Pointers (Swap)	O(n)	O(1)
Squares of Sorted Array	Two Pointers (End-Fill)	O(n)	O(n) (Output Array)
Valid Word Abbr	Two Pointers (Skip/Jump)	O(Length)	O(1)

Two Pointer pattern excels at optimizing space to O(1) for linear structures.

From Theory to Mastery

The Recommended Workflow



The Start Sheet

Download the raw problem set. No code, just definitions.

Visual Trace

Draw the pointers. Dry run the logic on paper before touching the keyboard.

The End Sheet

Code the solution. Compare with the reference implementation.

"Code it yourself. Watching a video is not learning. Consistency is key."