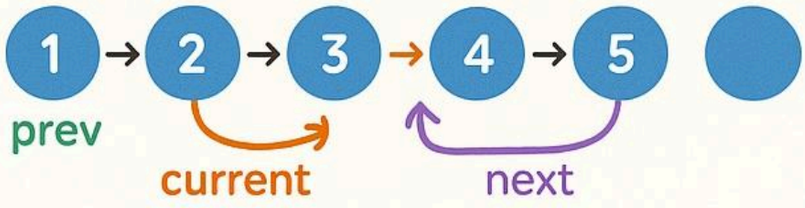
In-place Reversal of a Linked List Pattern

This pattern involves **reversing part or all of a singly linked list**, changing the direction of the `next` pointers **without using extra space** (no new list or array).
It's a fundamental technique often used as a subroutine in more complex linked list problems.

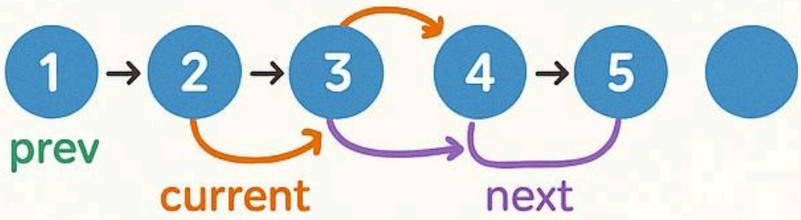
In-place Reversal of a Linked List Pattern

Task: Reverse the linked list in-place (no extra memory)

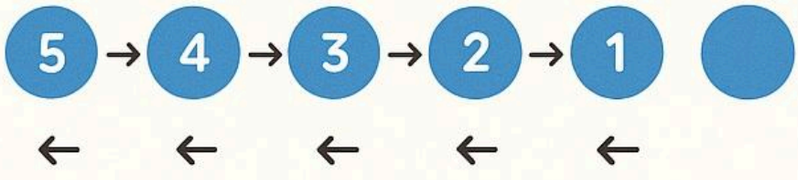
1 Use three pointers: prev, current, next



2 Reverse the link and move pointers forward



3 Repeat until end of list



Tip In-place reversal is key for many advanced linked list problems like k-group reversal, rotation, and palindrome checks

Classic Problem: Reverse an Entire Linked List

Problem Statement:
Given the head of a singly linked list, reverse the list **in place** (modify the `next` pointers so the last node becomes the first).

Explanation

1. **Use three pointers:**
   - `prev` (starts as NULL)
   - `current` (starts at head)
   - `next` (temporary, used to save the next node)
2. **Process:**
   - For each node:
     1. Store `current->next` in `next` (so you don't lose the rest of the list)
     2. Set `current->next` to `prev` (reverse the link)
     3. Move `prev` forward to `current`
     4. Move `current` forward to `next`
3. **Repeat until `current` is NULL.**
   At the end, `prev` is the new head.

C Code Example: Reverse a Linked List

```
#include <stdio.h>
#include <stdlib.h>

// Node definition
struct Node {
    int data;
    struct Node* next;
};

// Function to create a new node
struct Node* newNode(int data) {
    struct Node* node = (struct Node*)malloc(sizeof(struct Node));
    node->data = data;
    node->next = NULL;
    return node;
}

// Function to reverse the linked list in place
struct Node* reverseList(struct Node* head) {
    struct Node *prev = NULL, *current = head, *next = NULL;
    while (current != NULL) {
        next = current->next;   // Save next node
        current->next = prev;   // Reverse the link
        prev = current;         // Move prev forward
        current = next;         // Move current forward
    }
    return prev; // New head
}

// Utility to print list
void printList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

int main() {
    // Log Session linked list: 1->2->3->4->5
    struct Node* head = newNode(1);
    head->next = newNode(2);
    head->next->next = newNode(3);
    head->next->next->next = newNode(4);
    head->next->next->next->next = newNode(5);

    printf("Original List: ");
    printList(head);
```

```
    head = reverseList(head);

    printf("Reversed List: ");
    printList(head);

    return 0;
}
```

How it Works

- Start: `1 -> 2 -> 3 -> 4 -> 5 -> NULL`
- After 1st loop: `1 <- 2 -> 3 -> 4 -> 5` (1 points to NULL, 2 is current)
- After 2nd loop: `2 <- 1  3 -> 4 -> 5` (2 points to 1, 3 is current)
- ...and so on, until `5 -> 4 -> 3 -> 2 -> 1 -> NULL`

Where is this useful?

- Reverse the whole list
- Reverse a sub-list (between positions m and n)
- Problems like rotating lists, checking for palindrome lists, etc.

Practice Challenge

- Try reversing **only the first K elements** of a linked list.
- Try reversing a portion of the list (e.g., from position m to n).

Key Takeaways

- Use **three pointers** for in-place reversal.
- No extra space needed.
- The same logic is used in many advanced linked list problems!