Ask a Question

Introduction to In-Place Manipulation of a Linked List

Let’s go over the In-Place Manipulation of a Linked List pattern, its real-world applications, and some problems we can solve with it.

We'll cover the following

- About the pattern
- Examples
- Does your problem match this pattern?
- Real-world problems
- Strategy time!

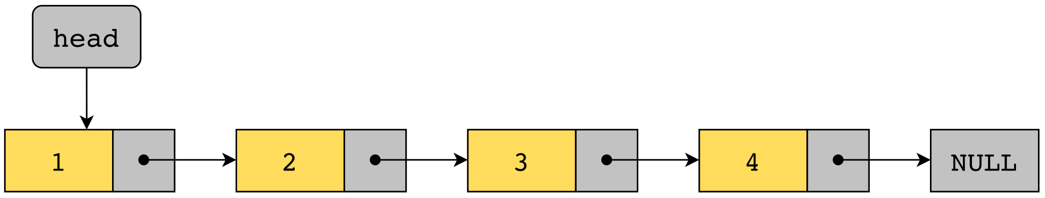
About the pattern

The **in-place manipulation of a linked list** pattern allows us to modify a linked list without using any additional memory. **In-place** refers to an algorithm that processes or modifies a data structure using only the existing memory space, without requiring additional memory proportional to the input size. This pattern is best suited for problems where we need to modify the structure of the linked list, i.e., the order in which nodes are linked together. For example, some problems require a reversal of a set nodes in a linked list which can extend to reversing the whole linked list. Instead of making a new linked list with reversed links, we can do it in place without using additional memory.

The naive approach to reverse a linked list is to traverse it and produce a new linked list with every link reversed. The time complexity of this algorithm is $O(n)$ while consuming $O(n)$ extra space. How can we implement the in-place reversal of nodes so that no extra space is used? We iterate over the linked list while keeping track of three nodes: the current node, the next node, and the previous node. Keeping track of these three nodes enables us to efficiently reverse the links between every pair of nodes. This in-place reversal of a linked list works in $O(n)$ time and consumes only $O(1)$ space.

The following illustration demonstrates the in-place modification of a linked list by depicting the reversal of the given linked list:

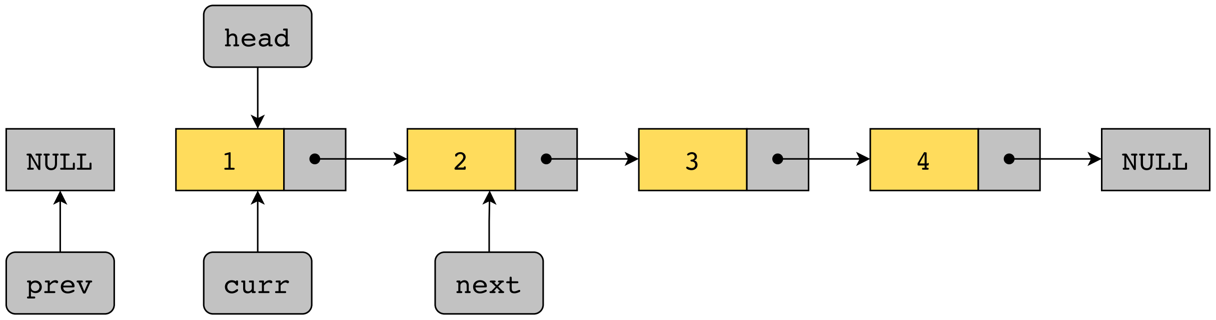
We'll demonstrate the in-place reversal of the following linked list.



1 of 10

Initialize three pointers:

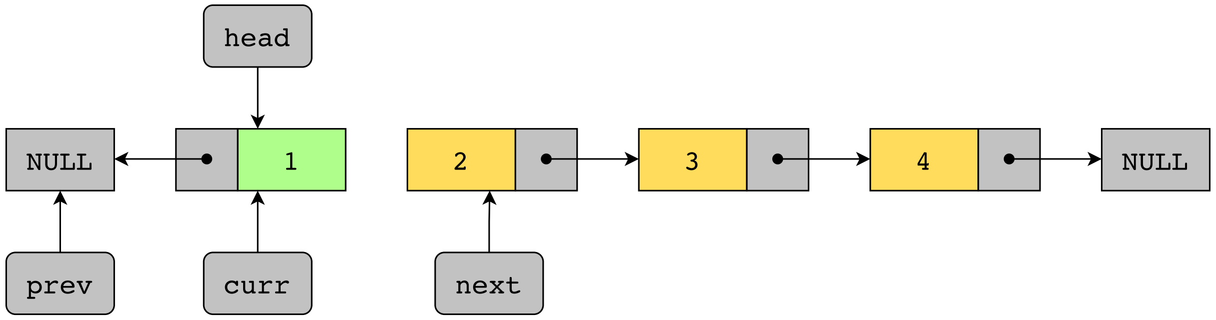
- **prev** at **NULL**
- **curr** at the **first** node
- **next** at the **second** node



2 of 10

Reverse the link of the **curr** node by making its **next** point to **prev**.

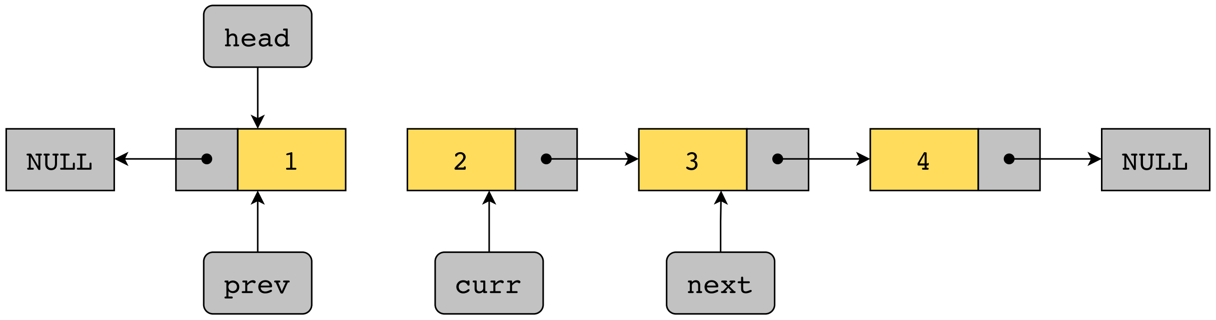
We don't modify the **head** pointer for now and will adjust it at the end.



3 of 10

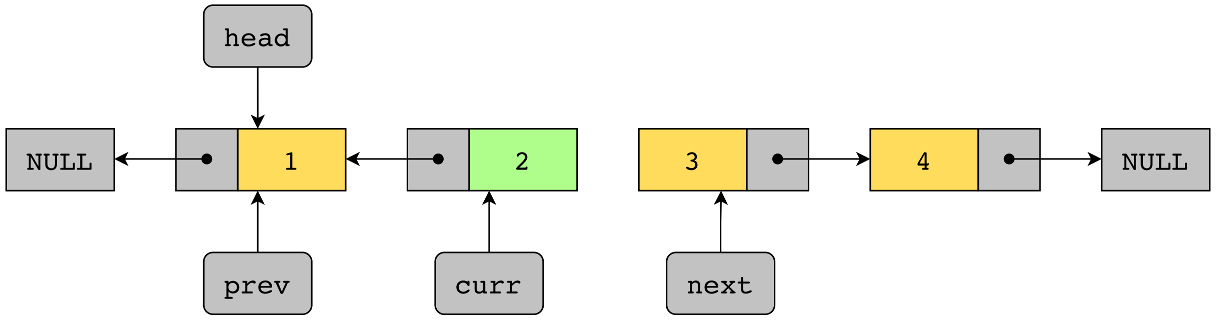
Move all three pointers one step forward:

- Move **prev** to **curr**
- Move **curr** to **next**
- Move **next** to the node next to it



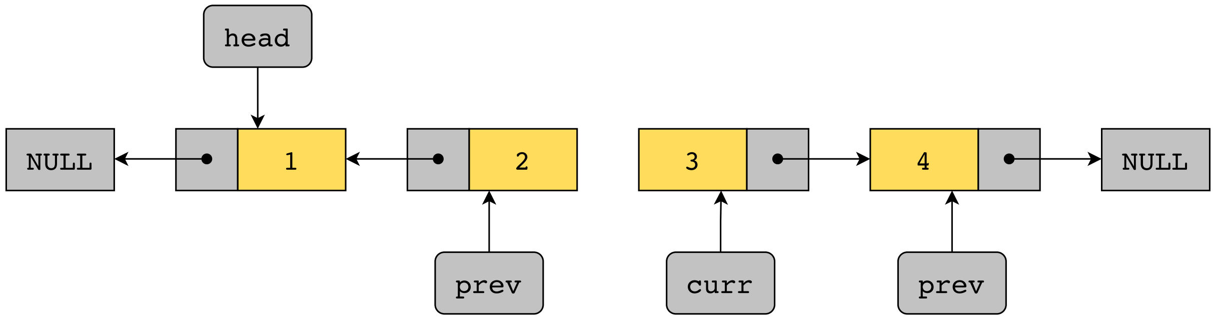
4 of 10

Reverse the link of the **curr** node by making its **next** point to **prev**.



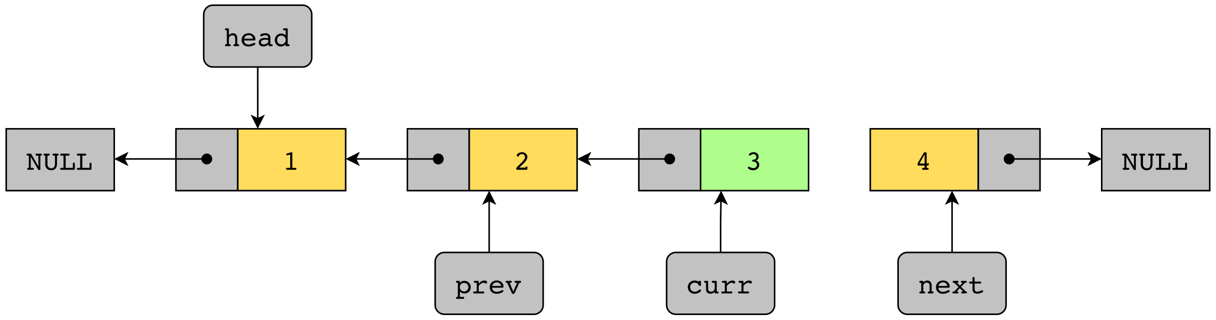
5 of 10

Move all three pointers one step forward.



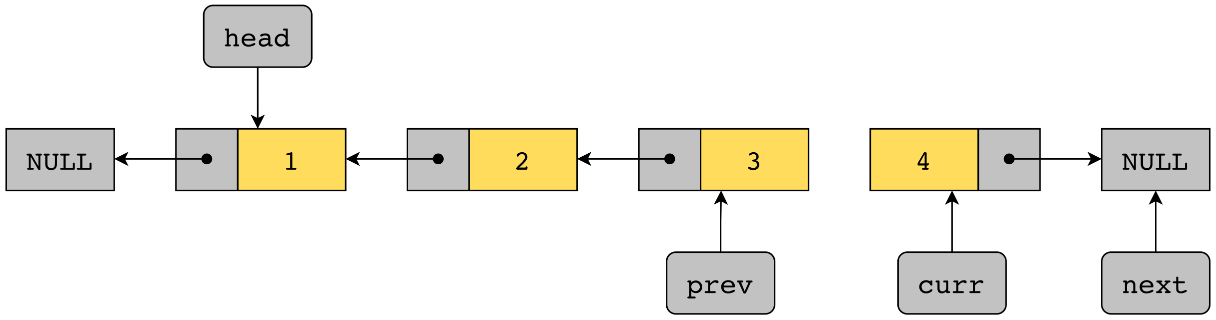
6 of 10

Reverse the link of the **curr** node.



7 of 10

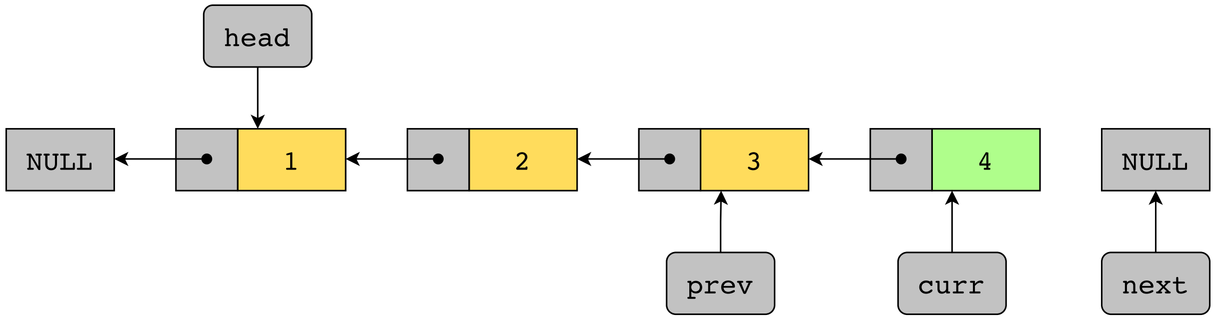
Move all three pointers one step forward.



8 of 10

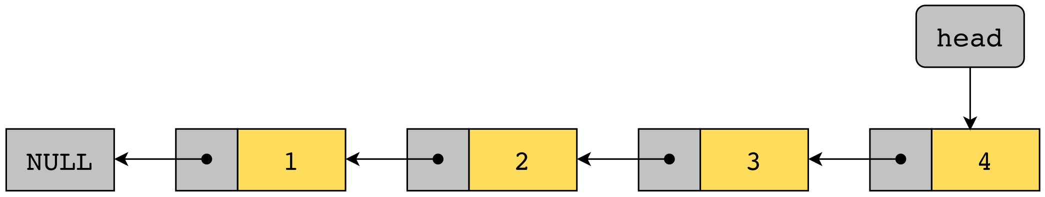
Reverse the link of the **curr** node.

At this point, the linked list has been completely reversed.



9 of 10

In the end, we adjust the **head** node.



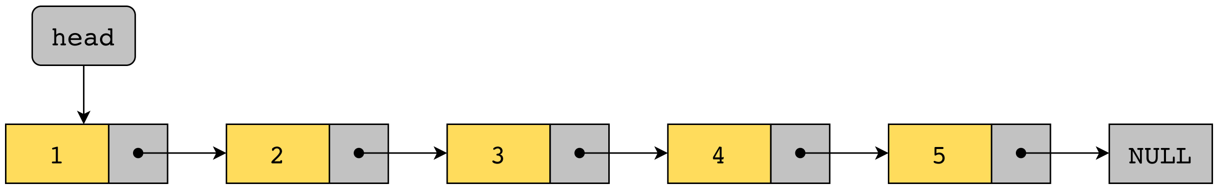
10 of 10

Examples

The following examples illustrate some problems that can be solved with this approach:

1. **Reverse the second half of a linked list:** Given a singly linked list, reverse the second half of the list.

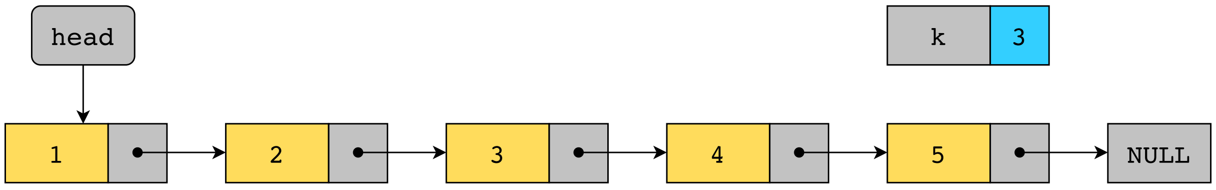
We have the following linked list and our task is to reverse its second half in-place.



1 of 10

2. **Rotate a linked list clockwise k times:** Given a singly linked list and an integer k, rotate the linked list clockwise k times.

This task is to rotate the following linked list **3** times without using any extra memory.



1 of 7

Does your problem match this pattern?

Yes, if both of these conditions are fulfilled:

- **Linked list restructuring:** The input data is given as a linked list, and the task is to modify its structure without modifying the data of the individual nodes.
- **In-place modification:** The modifications to the linked list must be made in place, that is, we're not allowed to use more than $O(1)$ additional space.

Real-world problems

Many problems in the real world use the in-place manipulation of a linked list pattern. Let’s look at some examples.

- **File system management:** File systems often use linked lists to manage directories and files. Operations such as rearranging files within a directory can be implemented by manipulating the underlying linked list in place.
- **Memory management**: In low-level programming or embedded systems, dynamic memory allocation and deallocation often involve manipulating linked lists of free memory blocks. Operations such as merging adjacent free blocks or splitting large blocks can be implemented in place to optimize memory usage.

Strategy time!

Match the problems that can be solved using the in-place manipulation of a linked list pattern.