Solution: Reverse Linked List II

Let's solve the Reverse Linked List II problem using the In-place reversal of a linked list pattern.

We'll cover the following

- Statement
- Solution
  - Naive approach
  - Optimized approach using in-place reversal of a linked list
    - Solution summary
    - Time complexity
    - Space complexity

Statement

Given a singly linked list with n nodes and two positions, `left` and `right`, the objective is to reverse the nodes of the list from `left` to `right`. Return the modified list.

Constraints:

- $1 \leq$ `n` ≤ 500
- $-5000 \leq$ `node.data` ≤ 5000
- $1 \leq$ `left` $\leq$ `right` $\leq$ `n`

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which to follow based on considerations such as time complexity and implementation constraints.

Naive approach

To reverse a portion of a linked list, starting from m^{th} position and ending at the n^{th} position, we will only swap the data of the nodes.

We will start from the head of the linked list and go to the m^{th} node. The `left` pointer will point to this node. We will then use another loop to go to the n^{th} node, starting from the m^{th} node. The `right` pointer will point to this node. With both pointers in place, we will swap the data of these nodes. After swapping, we will move the `left` pointer forward and the `right` pointer backward, and swap again. We will continue to do so until the portion of the linked list, from m to n, has been reversed. We will return the modified linked list.

The time complexity for this solution is $O(n^2)$, and in the worst case, n will be the length of the complete list. However, the space complexity for this naive approach is $O(1)$.

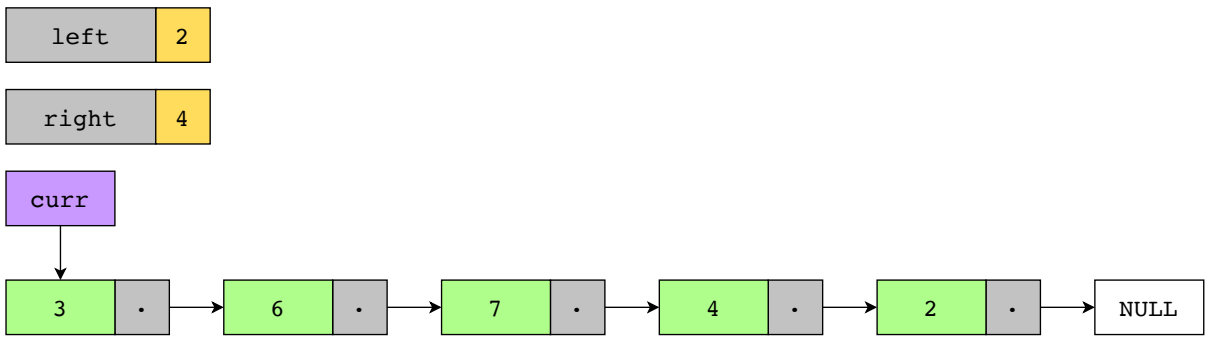
Optimized approach using in-place reversal of a linked list

The optimized approach to solve this problem is to use the in-place reversal pattern. This pattern allows us to modify the directions of the nodes in the linked list directly by keeping track of the current, next, and previous nodes without the need for any additional data structures.

To reverse the linked list from the left position node to the right position node, we will follow the steps mentioned below:

- We will use five pointers:
  - `curr`: This pointer will be initialized to the head of the linked list.
  - `prev`: This pointer will point to the previous node of the `curr` node.
  - `next`: This pointer will point to the next node of the `curr` node.
  - `lpn`: This pointer will point to the previous node of the left position node.
  - `rpn`: This pointer will point to the node after the right position node.
  - `right_n`: This pointer will point to the right position node.
- Iterate over the linked list using the `curr` pointer until:
  - The `curr` pointer points to the left position node.
  - The next node of `lpn` is the left position node.
- Iterate from the `curr` node to the right position node using the `right_n` pointer:
  - `rpn` points to the next node of the right position node.
- Start iterating from the left position node. While iterating, perform the following steps:
  - Before changing the next of `curr`, store the next node using the following line of code: `next = curr.next`.
  - Now, we will assign the `next` pointer of `curr` to the `prev` using the following line of code `curr.next = prev`. The effect of this line of code is that it will reverse the pointer from forward to backward to reverse the linked list.
  - After reversing the pointer, we will update `prev` as `curr` and `curr` as `next` using the following lines of code respectively: `prev = curr` and `curr = next`.
- After reversing the linked list from the left position node to the right position node, we will connect this reversed linked list with the original linked list using the `lpn` and `rpn`.

Let's look at the following illustration to get a better understanding of reversing the linked list:



Let's start by iterating the **curr** pointer forward until it reaches the left position node.

Let’s implement the algorithm as discussed above:

Java

main.java

PrintList.java

LinkedList.java

LinkedListNode.java

```java
class ReverseLinkedList {
    // Assume that the linked list has left to right nodes.
    // Reverse left to right nodes of the given linked list.
    public static LinkedListNode reverse(LinkedListNode head, int left, int r
        LinkedListNode prev = null;
        LinkedListNode curr = head;
        while (right >= left) {
            LinkedListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
            right--;
        }
        return prev; // Returns the head of the reversed list
    }
```

```java
        LinkedListNode curr = head;
        LinkedListNode lpn = null; // Previous node before the sublist
        LinkedListNode right_n = null; // Node after the sublist
        LinkedListNode reverse_head = null; // Head of the reversed sublist

        int count = 1;
        while (count < left && curr != null) {
            lpn = curr;
            curr = curr.next;
            count++;
```

Reverse Linked List II

Solution summary

Let’s recap the solution using the following points:

1. Find the left position node.
2. Reverse the list from the left position node to the right position node.
3. Merge the reversed list with the rest of the linked list.

Time complexity

Each node will be processed at most one time. Therefore, the time complexity will be $O(n)$, where n is the number of nodes in the linked list.

Space complexity

The space complexity will be $O(1)$, because we are using a constant number of additional variables to maintain the proper connections between the nodes during reversal.