Solution: Reverse Linked List

Let's solve the Reverse Linked List problem using the In-place Reversal of a Linked List pattern.

We'll cover the following

Statement

Given the head of a singly linked list, reverse the linked list and return its updated head.

Constraints:

Let `n` be the number of nodes in a linked list.

- $1 \leq$ `n` ≤ 500
- $-5000 \leq$ `Node.value` ≤ 5000

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 one to follow based on considerations, such as time complexity and any implementation constraints.

Naive approach

The naive approach to solve the reverse linked list problem is to create a new linked list by traversing the original linked list in reverse order. To do this, we can copy the nodes of the original linked list into another data structure, for example, a stack. Then, we can pop the nodes from the stack one by one, creating a new linked list with each node we pop.

This approach has a time complexity of $O(n)$, since we need to iterate through the entire original list and then iterate through the stack. However, the space complexity is also $O(n)$, since we need to store all the nodes in the data structure. This means that if the original linked list is very large, we may run into memory issues. Overall, while this approach is simple to implement, it may not be the most efficient solution for large linked lists.

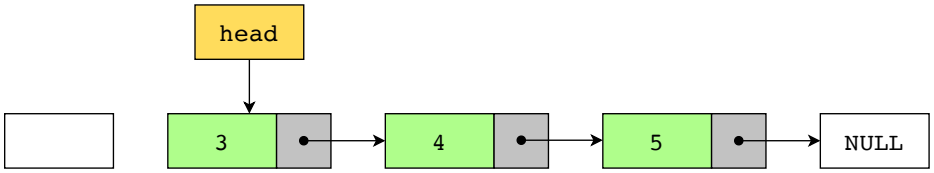
Optimized approach using in-place reversal of a linked list

To reverse the entire linked list without occupying extra memory, we can utilize 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, we will follow these steps:

- Initialize three pointers: `prev`, `next`, and `curr`. The `prev` and `next` pointers are initialized as NULL, while the `curr` pointer is initialized to the head of the linked list.
- Iterate over the linked list. 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` pointer 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'll update `prev` as `curr` and `curr` as `next` using `prev = curr` and `curr = next` respectively.
- After reversing the whole linked list, we'll change the head pointer to the `prev` pointer because `prev` will be pointing to the new head node.

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



Given the linked list above, we'll reverse it in place.

1 of 12

Let's implement the algorithm as discussed above:

Java

main.java

LinkedListNode.java

LinkedList.java

PrintList.java

```
1  import java.util.*;
2
3  class ReverseLinkedList {
4      public static LinkedListNode reverse(LinkedListNode head) {
5          // initialize prev and next pointer to NULL
6          LinkedListNode prev = null;
7          LinkedListNode next = null;
8          // set current pointer to the head node
9          LinkedListNode curr = head;
10         // while the current pointer is not NULL
11         while (curr != null) {
12             // set the next pointer to the next node in the list
13             next = curr.next;
```

```
14              // reverse the current node's pointer to point to the previous nc
15              curr.next = prev;
16              // set the previous pointer to the current node
17              prev = curr;
18              // move the current pointer to the next node
19              curr = next;
20          }
21          // set the head pointer to the last node, which is the new first node
24          return head;
25      }
26
27      public static void main(String[] args) {
28
```

Reverse Linked List

Solution summary

The solution summary is divided in the following parts:

- Initialize three pointers: `prev`, `next`, and `curr`.
- Reverse the links between adjacent nodes in a loop using the `next`, `curr`, and `prev` pointers.
- After reversing the linked list, update the head pointer to the last node of the original linked list, which is now the first node of the reversed linked list.
- Return the updated head pointer.

Time complexity

The time complexity of this solution is $O(n)$, because we reversed the linked list in a single pass, where n is the number of nodes in a linked list.

Space complexity

The space complexity of this solution is $O(1)$, because no extra memory is used.

Back
Reverse Linked List

Next
Reverse Nodes in k-Gr...

Mark as Completed