Reversing a linked list

While reversing a linked list is a common interview problem in itself, it is also a technique that can be a step in solving different problems. It requires a good understanding of how to move the pointers around.

The main idea is to use extra pointers to keep track of nodes whose links may be broken during the process:

```
def reverse_list(head):
    prev = None
    curr = head
    while curr:
        next_node = curr.next # first, make sure we don't lose the next node
        curr.next = prev      # reverse the direction of the pointer
        prev = curr           # set the current node to prev for the next
node
        curr = next_node      # move on

    return prev
```

The time complexity is $O(n)$ where n is the number of nodes in the linked list. The while loop runs n times, and the work done at each iteration is $O(1)$. The space complexity is $O(1)$ as we are only using a few pointers.

In this example, we had the following thought process:

1. When we are at node `curr`, we need to set its `next` pointer to the node we were at previously.
    - Use a `prev` pointer to track t he previous node.
2. The prev pointer needs to also update every iteration.
    - After updating `current.next`, set `prev = curr` in preparation for the next node.
3. If we set `curr.next = prev`, then we close the reference to the original `curr.next`.
    - Use `nextNode` to keep a reference to the original `curr.next`.

Example: 24 - Swap Nodes in Pairs

Given the `head` of a linked list, swap every pair of nodes. For example, given a linked list `1 -> 2 -> 3 -> 4 -> 5 -> 6`, return a linked list `2 -> 1 -> 4 -> 3 -> 6 -> 5`.

Consider the first pair of nodes as `A -> B`:

1. Starting with `head` at node `A`, we need node `B` to point here. We can accomplish this by doing `head.next.next = head`.
2. However, if we change `B.next`, we will lose access to the rest of the list. Before applying the change in step 1, save a pointer `nextNode = head.next.next`.
3. We now have `B` pointing at `A`. We need to move on to the next pair `C, D`. However, `A` is still pointing at `B`. If we move on to the next pair immediately, we will lose a reference to `A`, and won't be able to change `A.next`. Save `A` in another pointer with `prev = head` (we haven't changed `head` yet so it's still pointing at `A`). To move to the next pair, do `head = nextNode`.
4. Once we move on to the next pair `C -> D`, we need `A` to point to `D`. Now that `head` is at `C`, and `prev` is at `A`, we can do `prev.next = head.next`.
5. The first pair `A, B` is fully completed. `B` points to `A` and `A` points to `D`. When we started, we had `head` pointing to `A`. After going through the steps, we completed `A, B`. Right now, we have `head` pointing to `C`. If we go through the steps again, we will have complete `C, D`, and be ready for the next pair. We can repeat. But what do we return at the end? Once all the pairs are finished, we need to return `B`. Unfortunately, we lost the reference to `B` a long time ago. We can fix this by saving `B` in a `dummy` node before starting the algorithm.
6. What if there is an odd number of nodes? In step 4, we set `A.next` to `C.next`. What if there were only 3 nodes, so `C.next` was null? Before moving on to the next pair, set `head.next = nextNode`. This is setting `A.next` to `C`. Note that this effect will be overridden by step 4 in the next swap if there is still a pair of nodes remaining. Since in step 2 we do `head.next.next`, we need our while loop condition to check for both `head` and `head.next`. That means if there is only one node left in the list, the while loop will end after the current iteration. As such, this effect wouldn't be overridden. For example, consider the list `A -> B -> C -> D`. At some point, we have `B <-> A C -> D`. Here, we perform step 6, and we get `B -> A -> C -> D`. When we start swapping the pair `C, D`, step 4 will set `A.next` to `D`, which overrides what we just did with step 6. But if `D` didn't exist, then the iteration would have just ended. In that scenario, we would have `B -> A -> C`, which is what we want.

To summarize:

1. Perform an edge swap from `A -> B -> C -> ...` to `A <-> B C -> ...`.
2. Make sure we can still access the rest of the list beyond the current pair (saves `C`)

3. Now that `A <-> B` is isolated from the rest of the list, save a pointer to `A` to connect it with the rest of the list later. Move to the next pair.
4. Connect the previous pair to the rest of the list. In this case connecting `A -> D`.
5. Use a dummy pointer to keep a reference to what we want to return.
6. Handle the case when there's an odd number of nodes.

The order of the steps is not chronological. It's just an order that we might think of when we are trying to consider the requirements of the problem.

```
class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        # Check edge case: linked list has 0 or 1 nodes, just return
        if not head or not head.next:
            return head

        dummy = head.next               # Step 5
        prev = None                     # Initialize for step 3
        while head and head.next:
            if prev:
                prev.next = head.next   # Step 4
            prev = head                 # Step 3

            next_node = head.next.next  # Step 2
            head.next.next = head       # Step 1

            head.next = next_node       # Step 6
            head = next_node            # Move to next pair (Step 3)

        return dummy
```

The time complexity is $O(n)$ where n is the number of nodes in the linked list because the while loop runs n times, and the work done at each iteration is $O(1)$. The space complexity is $O(1)$ as we are only using a few pointers.

Reversing a linked list is "also a technique that can be a step in solving different problems".

2130 - Maximum Twin Sum of a Linked List

This problem asks for the maximum pair sum where the pairs are the first and last node, second and second to last node, etc.

The elegant $O(1)$ space solution is as follows:

1. Find the middle of the linked list using the fast and slow pointer technique.
2. Once at the middle of the linked list, perform a reversal. Basically, reverse only the second half of the list.
3. After reversing the second half, every node is spaced `n / 2` apart from its pair node, where n is the number of nodes in the list which we can find from step 1.
4. Create another fast pointer `n / 2` ahead of `slow`. Now, just iterate `n / 2` times from `head` to find every pair sum `slow.val + fast.val`.

The reversal is not the entire point of the problem, but is a tool used to arrive at a better solution. We also used fast and slow pointers.

Linked lists are used in hash table collision resolution with chaining and to implement deques for instance.

SEE PROBLEM 92