Editorial : Linked List 2

1) Reverse the Linked List

```
def reverse(A):
    curr = A
    before = None
    after = None
    while curr is not None:
        after = curr.next
        curr.next = before
        before = curr
        curr = after
    A = before
    return A
```

Explanation

1. **Initialize Pointers**

```
curr = A
before = None
after = None
```

   - `curr` points to the current node we are processing (starting at `A`, the head).
   - `before` will track the node before `curr` (initially `None`).
   - `after` will temporarily store the next node while reversing links.

2. **Iterate and Reverse**

```
while curr is not None:
    after = curr.next
    curr.next = before
    before = curr
    curr = after
```

   - **Store `after`**: Save the next node (`curr.next`) in `after` so we don't lose it.
   - **Reverse Link**: Make `curr.next` point back to `before`.

- **Advance `before`**: Move `before` to `curr`.
- **Advance `curr`**: Move `curr` to `after`.

3. **Update the Head**

```
A = before
return A
```

- After the loop finishes, `before` points to the **new head** of the reversed list.
- We assign `A = before` and return it.

---

2) Linked List Cycle

```python
class Solution:
    def hasCycle(self, head):
        if not head.next:
            return False

        first = head
        second = head

        while (second != None and second.next != None):

            first = first.next
            second = second.next.next

            if first == second:
                return True

        return False
```

Explanation

1. **Edge Case**

```
if not head.next:
    return False
```

- If the list has no second node, it can't have a cycle.

2. **Floyd's Cycle Detection** (Tortoise and Hare)

```
first = head
second = head
while (second != None and second.next != None):
    first = first.next
    second = second.next.next
    if first == second:
        return True
```

- `first` moves **one** step at a time.
- `second` moves **two** steps at a time.
- If they **ever meet**, there is a cycle.

3. **Return False if No Cycle**

```
return False
```

- If `second` (or `second.next`) becomes `None`, we exit the loop, meaning the list ended without looping back.

---

3) Add Two Linked Lists

```
def addTwoNumbers(first, second):
    prev = None
    temp = None
    carry = 0
    head = None

    while (first is not None or second is not None):
        fdata = 0 if first is None else first.data
        sdata = 0 if second is None else second.data
        Sum = carry + fdata + sdata

        carry = 1 if Sum >= 10 else 0
        Sum = Sum if Sum < 10 else Sum % 10

        temp = Node(Sum)

        if head is None:
            head = temp
        else:
            prev.next = temp
```

```
        prev = temp

        if first is not None:
            first = first.next
        if second is not None:
            second = second.next

    if carry > 0:
        temp.next = Node(carry)

    return head
```

Explanation

1. **Initialize Variables**

```
carry = 0
head = None
prev = None
```

   - `carry` tracks any overflow from adding two digits (0 or 1).
   - `head` will be the head of the resulting sum list.
   - `prev` tracks the last node we created.

2. **Traverse Both Lists**

```
while (first is not None or second is not None):
    fdata = 0 if first is None else first.data
    sdata = 0 if second is None else second.data
    Sum = carry + fdata + sdata
    ...
```

   - We loop until both `first` and `second` are fully traversed.
   - If one list is shorter, we use 0 for its value once it's `None`.

3. **Calculate Digit and Carry**

```
carry = 1 if Sum >= 10 else 0
Sum = Sum if Sum < 10 else Sum % 10
```

   - If `Sum` is 10 or more, we set `carry` to 1 and reduce `Sum` to a single digit (`Sum % 10`).

4. **Create and Link the New Node**

```
temp = Node(Sum)
if head is None:
    head = temp
else:
    prev.next = temp
prev = temp
```

- If this is the first node, it becomes `head`.
- Otherwise, we attach it to the end of the result list by linking `prev.next = temp`.
- Then we move `prev` to `temp`.

5. **Advance Pointers**

```
if first is not None:
    first = first.next
if second is not None:
    second = second.next
```

6. **Check Remaining Carry**

```
if carry > 0:
    temp.next = Node(carry)
```

- If there's a leftover carry after the final addition, we add a new node with `carry`.

7. **Return the Head**

```
return head
```

- The resulting linked list's head is returned.

---

4) Nth Node from the End

```
def nthNode(A, n):
    first = A
    second = A
    count = 0
```

```
    while (count < n):
        second = second.next
        count += 1
    while (second is not None):
        second = second.next
        first = first.next
    return first.data
```

Explanation

1. **Two Pointers**
   - `first` and `second` both start at the head (`A`).
2. **Advance `second` by `n` Steps**

```
while (count < n):
    second = second.next
    count += 1
```

   - This ensures that `second` is `n` nodes ahead of `first`.
3. **Move Both Until `second` Reaches End**

```
while (second is not None):
    second = second.next
    first = first.next
```

   - When `second` reaches `None`, `first` will be at the **(length - n)th** node, i.e., the **nth node from the end**.
4. **Return the Data**

```
return first.data
```

   - We return the data of the node at `first`.

---

5) Delete Kth Node from End

```python
def deleteYAfterX(head, K):

    first = head
    second = head

    for _ in range(K):
        first = first.next

    if first is None:
        return head.next

    while first.next is not None:
        first = first.next
        second = second.next

    second.next = second.next.next

    return head
```

Explanation

1. **Two Pointers**

```python
first = head
second = head
for _ in range(K):
    first = first.next
```

   - Move `first` exactly `K` nodes ahead of `second`.

2. **Edge Case**

```python
if first is None:
    return head.next
```

   - If `first` becomes `None` after moving `K` steps, it means we need to remove the **head** (the Kth node from the end is the head).

3. **Traverse Until `first` Reaches the End**

```python
while first.next is not None:
    first = first.next
    second = second.next
```

- Once `first` is at the last node, `second` is exactly at the node before the Kth-from-end.

4. **Remove the Kth-from-End Node**

```
second.next = second.next.next
return head
```

- We skip the target node by re-linking `second.next`.

---

6) Add 1 to Linked List

```
def addOne(head):
    def reverseList(node):
        prev = None
        current = node
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        return prev

    head = reverseList(head)

    carry = 1
    current = head
    while current:
        current.data += carry
        if current.data < 10:
            carry = 0
            break
        current.data = 0
        if not current.next:
            current.next = Node(0)
        current = current.next

    head = reverseList(head)
    return head
```

Explanation

1. **Reverse the List**

```
def reverseList(node):
    ...
head = reverseList(head)
```

- We reverse the linked list so the **least significant digit** is at the front.

2. **Add One**

```
carry = 1
current = head
while current:
    current.data += carry
    if current.data < 10:
        carry = 0
        break
    current.data = 0
    if not current.next:
        current.next = Node(0)
    current = current.next
```

- Start with a carry of 1 (since we're adding 1).
- Add `carry` to `current.data`.
- If `current.data` is now 10 or more, set it to 0 and keep carry = 1.
- If `current.data` is less than 10, set carry = 0 and break out of the loop (no further carry to propagate).
- If we reach the end of the list and still have carry = 1, we create a new node (e.g., going from 999 to 1000).

3. **Reverse Again**

```
head = reverseList(head)
return head
```

- After adding 1, we reverse the list back to its original order.
- Return the updated head.

Summary

1. **Reverse the Linked List**: Uses a three-pointer method (`before` , `curr` , `after`) to reverse links.
2. **Linked List Cycle**: Implements Floyd's Cycle Detection (fast & slow pointers).

3. **Add Two Linked Lists**: Iterates over both lists digit by digit, handling carry.
4. **Nth Node from End**: Uses two pointers, offset by `n`.
5. **Delete Kth Node from End**: Similar approach, but instead of returning the node's value, we remove it.
6. **Add 1 to Linked List**: Reverses the list, adds 1, and reverses it back.
7. **Good Friend** : You can skip this question, issue from backend.