Editorial : Linked List 1st Assignment

Question : Insert a Node at the Head

```python
def insertNodeAtHead(head, data):
    newNode = Node(data)
    newNode.next = head
    return newNode
```

Explanation

1. **Function Signature**
   - `insertNodeAtHead(head, data)` takes:
     - `head`: the current head of the linked list (may be `None` if the list is empty).
     - `data`: the value to store in the new node.
2. **Create a New Node**

   ```python
   newNode = Node(data)
   ```

   - We instantiate a new `Node` object, passing in `data`.
3. **Link the New Node**

   ```python
   newNode.next = head
   ```

   - The `newNode` becomes the new head, so we set its `next` pointer to the **old** head.
4. **Return the New Head**

   ```python
   return newNode
   ```

   - The newly created node is now at the front of the list, so we return it as the new `head`.

Question : Insert a Node at the Tail

```python
class Solution:
    def insertNodeAtTail(self, head, data):
        t = Node(data)
        if head is None:
            head = t
        else:
            current = head
            while current.next:
                current = current.next
            current.next = t
        return head
```

Explanation

1. **Function Definition**

    - `insertNodeAtTail(self, head, data)` takes:
        - `head`: the head of the linked list.
        - `data`: the value to be inserted at the end of the list.

2. **Create a New Node**

    ```python
    t = Node(data)
    ```

    - We create a new node `t` with the given `data`.

3. **Check if the List is Empty**

    ```python
    if head is None:
        head = t
    ```

    - If there is no existing list (`head` is `None`), the new node `t` becomes the head.

4. **Traverse to the End**

    ```python
    else:
        current = head
        while current.next:
            current = current.next
        current.next = t
    ```

    - Otherwise, we start from `head` and move `current` forward until `current.next` is `None`.

- Once we reach the last node, we link it to `t` by setting `current.next = t`.

5. **Return the Updated Head**

```
return head
```

- The `head` itself doesn't change (except in the case of an initially empty list), but we return it to confirm the updated list.

Question : Insert at a Specific Position

```
class Solution:
    def insertNodeAtaPosition(self, head, data, position):
        node = Node(data)
        if position == 0:
            node.next = head
            return node
        else:
            current = head
            prev = head
            after = head.next
            position -= 1
            while position > 0:
                after = after.next
                prev = prev.next
                position -= 1
            node.next = after
            prev.next = node
            return current
```

Explanation

1. **Parameters**

- `head`: The head of the linked list.
- `data`: The value for the new node.
- `position`: The 0-based index where we want to insert the new node.

2. **Create the New Node**

```
node = Node(data)
```

3. **If Inserting at the Head (position == 0)**

```
if position == 0:
    node.next = head
    return node
```

   - We simply link the new node to the old head and return the new node as the new head.

4. **Otherwise, Traverse to the Insertion Point**

```
current = head
prev = head
after = head.next
position -= 1
while position > 0:
    after = after.next
    prev = prev.next
    position -= 1
```

   - We reduce `position` by 1 (since we already accounted for the head).
   - We move `prev` and `after` forward until `position` reaches 0.
   - By the end, `prev` is the node **before** the insertion spot, and `after` is the node **after** the insertion spot.

5. **Link the New Node**

```
node.next = after
prev.next = node
```

   - We insert the new node between `prev` and `after`.

6. **Return the Original Head**

```
return current
```

   - The head of the list does not change (unless `position` was 0), so we return the original `current`.

Question : Deleting a Node

```python
class Solution:
    def deleteNode(self, head, position):
        if position == 0:
            return head.next

        current = head
        prev = head
        after = head.next
        position -= 1
        while position > 0:
            after = after.next
            prev = prev.next
            position -= 1
        prev.next = after.next
        return current
```

Explanation

1. **Parameters**

   - `head` : The head of the linked list.
   - `position` : The 0-based index of the node to delete.

2. **If Deleting the Head (position == 0)**

   ```python
   if position == 0:
       return head.next
   ```

   - We simply skip the current head by returning `head.next`.

3. **Traverse to the Node Before the Target**

   ```python
   current = head
   prev = head
   after = head.next
   position -= 1
   while position > 0:
       after = after.next
       prev = prev.next
       position -= 1
   ```

- We move `prev` and `after` until `position` becomes 0.
- At this point, `prev` is the node before the one to delete, and `after` is the node to be deleted.

4. **Remove the Node**

```
prev.next = after.next
```

   - We skip over the `after` node, effectively removing it from the list.

5. **Return the (Potentially Unchanged) Head**

```
return current
```

   - If we didn't delete the head, the head is unchanged. Otherwise, we handled that case separately.

Question : Middle Node

```
class Solution:
    def middleNode(self, head):

        slow_ptr = head
        fast_ptr = head
        while fast_ptr is not None and fast_ptr.next is not None:
            slow_ptr = slow_ptr.next
            fast_ptr = fast_ptr.next.next
        return slow_ptr.data
```

Explanation

1. **Parameters**

   - `head`: The head of the linked list.

2. **Initialize Two Pointers**

```
slow_ptr = head
fast_ptr = head
```

- `slow_ptr` will move **one** node at a time.
- `fast_ptr` will move **two** nodes at a time.

3. **Traverse the List**

```
while fast_ptr is not None and fast_ptr.next is not None:
    slow_ptr = slow_ptr.next
    fast_ptr = fast_ptr.next.next
```

- We move both pointers until `fast_ptr` reaches the end (or just before the end).
- By the time `fast_ptr` can't move further, `slow_ptr` is exactly at the middle.

4. **Return the Middle Node's Data**

```
return slow_ptr.data
```

- The problem statement says to return the data of the middle node.
- If there are two middle nodes (even-length list), this code will return the **second** middle node (since `slow_ptr` moves after `fast_ptr` each time).

Question : Rotate Linked List

```
# Definition of Linked List Node

# class Node:
#   def __init__(self, data):
#       self.data = data
#       self.next = None

# Complete the function below

class Solution:

    def rotateRight(self,head,k):

        n = 0;
        temp = head;
        while(temp):
            temp = temp.next;
            n += 1;
```

```
    # Reduce k so it doesn't exceed the length of the list
    k = k % n;

    # Perform the rotation k times
    for i in range(k):
        prev = None;
        curr = head;

        # Traverse to the end of the list
        while(curr.next != None):
            prev = curr;
            curr = curr.next;

        # 'curr' is now the last node, 'prev' is the node before the last
        prev.next = None;  # Break the link before the last node
        curr.next = head;  # Link the last node to the front
        head = curr;       # Update the head to be the last node

    return head;
```

Explanation

1. **Finding the Length of the List**

   ```
   n = 0;
   temp = head;
   while(temp):
       temp = temp.next;
       n += 1;
   ```

   - We initialize a counter `n` to `0`.
   - We traverse the linked list using a temporary pointer `temp`, incrementing `n` for each node until we reach the end.
   - After this loop, `n` holds the total number of nodes in the linked list.

2. **Adjusting `k`**

   ```
   k = k % n;
   ```

   - If `k` is larger than the length of the list, rotating the list more than `n` times would be redundant (every `n` rotations brings the list back to its original state).

- We therefore take `k` modulo `n` to handle cases where `k >= n`.

3. **Rotating the List `k` Times**

```
for i in range(k):
    prev = None;
    curr = head;
    while(curr.next != None):
        prev = curr;
        curr = curr.next;

    prev.next = None;
    curr.next = head;
    head = curr;
```

   - **Outer Loop** (`for i in range(k)`): We will perform the 1-step rotation process exactly `k` times.
   - **Inner Loop**: We start from the head (`curr = head`) and move forward until `curr.next` is `None`.
     - At the end of this traversal, `curr` points to the **last node** in the list, and `prev` points to the node just before the last node.
   - **Re-linking**:
     - `prev.next = None;` removes the last node from its current position by making the second-to-last node the new tail.
     - `curr.next = head;` places the last node at the **front** of the list.
     - `head = curr;` updates the list's head pointer to this newly moved node.

4. **Return the New Head**

```
return head;
```

   - After `k` such single-step rotations, we return the updated `head` of the list.

Time Complexity Note

- This approach performs a **1-step rotation** `k` times. Each 1-step rotation requires traversing the list (in the worst case), leading to a time complexity of approximately **O(n*k)**.
- There is a more optimal solution that runs in **O(n)** by connecting the tail to the head to form a cycle, then breaking at the correct spot. However, the above code correctly implements a simpler, step-by-step rotation approach.

Question : Delete Middle Node

```
import math
def solve(head, N):
    # write code here
    curr = head;
    length = 0;
    while(curr!=None):
        length+=1;
        curr = curr.next;
    if(length%2 == 0):
        length = math.ceil(length/2);
        temp = head;
        for _ in range(1,length):
            temp = temp.next;

        temp.next = temp.next.next
        return head;
    length = length//2;
    temp = head;
    for _ in range(1,length):
            temp = temp.next;

    temp.next = temp.next.next;
    return head;
```

Explanation

1. **Calculate the Length of the List**

   ```
   curr = head
   length = 0
   while curr != None:
       length += 1
       curr = curr.next
   ```

   - We iterate through the list to find the total number of nodes (`length`).

2. **Determine the Middle Index**

   ```
   if (length % 2) == 0:
       length = math.ceil(length / 2)
       ...
   else:
   ```

```
    length = length // 2
    ...
```

- If the list has an **even** number of nodes, we take the ceiling of `length / 2`.
- If the list has an **odd** number of nodes, we use integer division `length // 2`.
- This effectively decides which node is considered the “middle” to delete.

3. **Traverse to the Node Before the Middle**

```
temp = head
for _ in range(1, length):
    temp = temp.next
```

- We move `temp` to the node **just before** the middle node we want to remove.

4. **Delete the Middle Node**

```
temp.next = temp.next.next
```

- We skip over the middle node by linking `temp.next` to the node after the middle.

5. **Return the Modified Head**

```
return head
```

- The head might not change if we’re not deleting the first node, so we return the original `head`.

Question : Sort Binary List

```
def solve(head):
    # Write code here

    curr = head;
    countO = 0;
    countOne = 1;

    while(curr != None):
        if(curr.data == 0):
            countO += 1;
        else:
```

```
            countOne += 1;

        curr = curr.next;

    temp = head;
    # print(count0,countOne);
    while(count0>0):
        temp.data = 0;
        if(temp.next != None):
            temp = temp.next;
        else:
            break;
        count0 -= 1;

    while(countOne>0):
        temp.data = 1;
        if(temp.next != None):
            temp = temp.next;
        else:
            break;
        countOne -= 1;


    return head;
```

Explanation

1. **Counting Zeros and Ones**

   ```
   curr = head
   count0 = 0
   countOne = 1
   while curr != None:
       if curr.data == 0:
           count0 += 1
       else:
           countOne += 1
       curr = curr.next
   ```

   - We traverse the linked list once, counting the number of nodes that contain `0` and `1`.
   - The variable `count0` tracks how many `0` s there are, and `countOne` tracks how many `1` s.
   - **Note**: In the code, `countOne` starts from `1`, which slightly offsets the actual count by +1.

2. **Rewrite the List with Zeros and Ones**

```python
temp = head
while count0 > 0:
    temp.data = 0
    ...
    count0 -= 1

while countOne > 0:
    temp.data = 1
    ...
    countOne -= 1
```

- We iterate again from the head, first assigning `0` to the data of each node until we've placed all zeros.
- Then we assign `1` to the remaining nodes until we've placed all ones.
- This ensures the list is sorted in non-decreasing order (all `0` s followed by `1` s).

3. **Return the Head**

```python
return head
```

- We return the head of the now "sorted" list.

Question : Remove Maximum Element in Linked List

```python
class Solution:
    def deleteMaximum(self, head):
        # Write your code here
        if(head == None):
            return -1;

        if(head.next==None):
            return None;

        i = 0;
        pos = -1;
        curr = head;
        mx = float("-inf");
        while(curr!=None):
            if(curr.data>=mx):
                mx = curr.data;
                pos = i;
            curr = curr.next;
```

```
            i+=1;

        prev = None;
        temp = head;
        for i in range(pos):
            prev = temp;
            temp = temp.next;

        if(prev == None):
            return temp.next;

        prev.next = temp.next;
        return head;
```

Explanation

1. **Check for Empty or Single-Node List**

```
if head == None:
    return -1  # Indicate an empty list
if head.next == None:
    return None # Removing the only node leaves the list empty
```

   - If the list is empty, return `-1`.
   - If there's only one node, removing it returns `None`.

2. **Find the Maximum Value and Its Position**

```
i = 0
pos = -1
curr = head
mx = float("-inf")
while curr != None:
    if curr.data >= mx:
        mx = curr.data
        pos = i
    curr = curr.next
    i += 1
```

   - We track the **largest value** `mx` and the **position** `pos` where it occurs.
   - If multiple nodes share the maximum value, we update `pos` to the **last** occurrence (because `>=` is used).

3. **Traverse to the Node Before the Maximum**

```python
prev = None
temp = head
for i in range(pos):
    prev = temp
    temp = temp.next
```

- We move `temp` to the node **at** position `pos`.
- `prev` will end up as the node **before** `temp`.

4. **Delete the Maximum Node**

```python
if prev == None:
    return temp.next  # If the max node is at the head
prev.next = temp.next
return head
```

- If `prev` is `None`, it means the max node was at the head, so we skip it by returning `temp.next`.
- Otherwise, we link `prev.next` to `temp.next`, removing the maximum node from the chain.

Question : Insert Node in Doubly Linked List at the End

```python
def addNode(A, B):
    X = Node(B)
    if A is None:
        return X
    head = A
    while A.next != None:
        A = A.next
    A.next = X
    X.prev = A
    return head
```

Explanation

1. **Function Parameters**

- `A`: The head of the doubly linked list.
- `B`: The value to be inserted into a new node.

2. **Create the New Node**

```
X = Node(B)
```

- We instantiate a new `Node` with the data `B`.

3. **Check if the List is Empty**

```
if A is None:
    return X
```

- If there is no existing list, the new node `X` becomes the head.

4. **Traverse to the End**

```
head = A
while A.next != None:
    A = A.next
```

- We move through the list until we find the last node (`A.next == None`).

5. **Link the New Node**

```
A.next = X
X.prev = A
return head
```

- We attach `X` as the last node by setting `A.next = X`.
- We set the `prev` pointer of `X` to `A`.
- We return the original `head` of the list.

Summary

- **Delete Middle Node**: Finds the list's length, determines the middle, and removes it.
- **Sort Binary List**: Counts the number of `0`s and `1`s, then rewrites the list data accordingly.
- **Remove Maximum Element**: Finds the position of the **last** occurrence of the maximum value, then removes that node.
- **Insert Node in Doubly Linked List at the End**: Traverses to the end and attaches a new node in a doubly linked list.

Each snippet addresses a specific linked-list manipulation scenario, from single to doubly linked lists.