Linked List Practice Question
1.
A doubly linked list is a type of linked list in which each node contains a reference or pointer to both the next node and the previous node in the sequence. This allows traversal of the list in both forward and backward directions.

2.

To reverse a linked list in-place means to modify the links between nodes so that the list is reversed without using any additional data structures. Here's a Python function to achieve that:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverse_linked_list(head):
    prev_node = None
    current_node = head

    while current_node:
        next_node = current_node.next
        current_node.next = prev_node
        prev_node = current_node
        current_node = next_node

    return prev_node
```

This function takes the head of the linked list as input and iterates through the list, reversing the links between nodes. Finally, it returns the new head of the reversed list.

```
# Example usage:
# Create a linked list: 1 -> 2 -> 3 -> 4 -> None
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)

# Reverse the linked list
head = reverse_linked_list(head)

# Print the reversed linked list
current_node = head
while current_node:
    print(current_node.val, end=" ")
    current_node = current_node.next
# Output: 4 3 2 1
```

This will reverse the linked list `1 -> 2 -> 3 -> 4 -> None` to `4 -> 3 -> 2 -> 1 -> None`.

3.

To detect a cycle in a linked list, you can use Floyd's Tortoise and Hare algorithm (also known as Floyd's cycle detection algorithm). Here's a Python function to implement it:

```
class ListNode:
```

```
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def has_cycle(head):
    if not head:
        return False

    slow_ptr = head
    fast_ptr = head

    while fast_ptr and fast_ptr.next:
        slow_ptr = slow_ptr.next
        fast_ptr = fast_ptr.next.next

        if slow_ptr == fast_ptr:
            return True

    return False
```

This function takes the head of the linked list as input and uses two pointers: a slow pointer and a fast pointer. The slow pointer moves one node at a time while the fast pointer moves two nodes at a time. If there is a cycle in the linked list, eventually the fast pointer will catch up to the slow pointer. If there is no cycle, the fast pointer will reach the end of the list (i.e., become None).

```
# Example usage:
# Create a linked list with a cycle: 1 -> 2 -> 3 -> 4 -> 5 -> 2 (cycle)
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(5)
head.next.next.next.next.next = head.next  # Create a cycle

# Check if the linked list has a cycle
print(has_cycle(head))  # Output: True
```

4.
To merge two sorted linked lists into one, you can iterate through both lists simultaneously, comparing the values of the nodes and creating a new merged list. Here's how you can do it in Python:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def merge_two_lists(l1, l2):
    # Create a dummy node to serve as the head of the merged list
    dummy = ListNode()
    curr = dummy

    # Traverse both lists and compare values
    while l1 and l2:
        if l1.val < l2.val:
            curr.next = l1
            l1 = l1.next
        else:
```

```
            curr.next = l2
            l2 = l2.next
        curr = curr.next

    # If any list is not fully traversed, append the remaining nodes to the
merged list
    if l1:
        curr.next = l1
    if l2:
        curr.next = l2

    # Return the merged list (skip the dummy node)
    return dummy.next


# Create the first sorted linked list: 1 -> 3 -> 5 -> 7 -> None
l1 = ListNode(1)
l1.next = ListNode(3)
l1.next.next = ListNode(5)
l1.next.next.next = ListNode(7)

# Create the second sorted linked list: 2 -> 4 -> 6 -> 8 -> None
l2 = ListNode(2)
l2.next = ListNode(4)
l2.next.next = ListNode(6)
l2.next.next.next = ListNode(8)

# Merge the two lists
merged_list = merge_two_lists(l1, l2)

# Print the merged list
while merged_list:
    print(merged_list.val, end=" -> ")
    merged_list = merged_list.next
print("None")


1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> None

5.
You can achieve this by using two pointers, one moving ahead by `n+1` steps and
the other starting from the head. When the first pointer reaches the end, the
second pointer will be at the `n`th node from the end. Here's the implementation
in Python:

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def remove_nth_from_end(head, n):
    # Create a dummy node to handle edge cases
    dummy = ListNode(0)
    dummy.next = head

    # Initialize two pointers
    first = dummy
    second = dummy

    # Move the first pointer ahead by n+1 steps
    for _ in range(n+1):
        first = first.next

    # Move both pointers until the first pointer reaches the end
```

```
    while first:
        first = first.next
        second = second.next

    # Remove the nth node from the end
    second.next = second.next.next

    # Return the updated head of the list
    return dummy.next

# Create the linked list: 1 -> 2 -> 3 -> 4 -> 5 -> 6
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(5)
head.next.next.next.next.next = ListNode(6)

# Remove the 2nd node from the end
head = remove_nth_from_end(head, 2)

# Print the modified linked list
while head:
    print(head.val, end=" -> ")
    head = head.next
print("None")
```

This will output:

```
1 -> 2 -> 3 -> 4 -> 6 -> None
```

6.

You can remove duplicates from a sorted linked list by iterating through the list and comparing each node's value with the next node's value. If they are the same, you skip the next node, effectively removing duplicates. Here's the implementation in Python:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def delete_duplicates(head):
    current = head
    while current and current.next:
        if current.val == current.next.val:
            current.next = current.next.next
        else:
            current = current.next
    return head

# Create the linked list: 1 -> 2 -> 3 -> 3 -> 4 -> 4 -> 4 -> 5
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(3)
head.next.next.next.next = ListNode(4)
head.next.next.next.next.next = ListNode(4)
head.next.next.next.next.next.next = ListNode(4)
head.next.next.next.next.next.next.next = ListNode(5)

# Remove duplicates
head = delete_duplicates(head)
```

```
# Print the modified linked list
while head:
    print(head.val, end=" -> ")
    head = head.next
print("None")
```

This will output:

```
1 -> 2 -> 3 -> 4 -> 5 -> None
```

7.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def get_intersection(headA, headB):
    # Find the lengths of both linked lists
    lenA, lenB = 0, 0
    nodeA, nodeB = headA, headB
    while nodeA:
        lenA += 1
        nodeA = nodeA.next
    while nodeB:
        lenB += 1
        nodeB = nodeB.next

    # Reset the pointers to the heads of the linked lists
    nodeA, nodeB = headA, headB

    # Traverse the longer list by the difference in lengths
    if lenA > lenB:
        for _ in range(lenA - lenB):
            nodeA = nodeA.next
    else:
        for _ in range(lenB - lenA):
            nodeB = nodeB.next

    # Traverse both lists in parallel until intersection or end
    while nodeA and nodeB and nodeA != nodeB:
        nodeA = nodeA.next
        nodeB = nodeB.next

    return nodeA

# Example usage
# Create linked lists
listA = ListNode(1)
listA.next = ListNode(2)
listA.next.next = ListNode(3)
listA.next.next.next = ListNode(4)
listA.next.next.next.next = ListNode(8)
listA.next.next.next.next.next = ListNode(6)
listA.next.next.next.next.next.next = ListNode(9)

listB = ListNode(5)
listB.next = ListNode(1)
listB.next.next = ListNode(6)
listB.next.next.next = ListNode(7)

# Set intersection node
intersection = listA.next.next.next
```

```
# Connect listB to the intersection node
listB.next.next.next = intersection

# Find intersection
intersection_node = get_intersection(listA, listB)

if intersection_node:
    print("Intersection found at node with value:", intersection_node.val)
else:
    print("No intersection found")
```

This will output:

```
Intersection found at node with value: 4
```

8.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def rotate_right(head, k):
    if not head or not head.next or k == 0:
        return head

    # Find the length of the linked list
    length = 1
    tail = head
    while tail.next:
        length += 1
        tail = tail.next

    # Calculate the actual rotation count
    actual_rotation_count = k % length

    # If actual rotation count is 0, no need to rotate
    if actual_rotation_count == 0:
        return head

    # Traverse to the (length - actual_rotation_count - 1)th node
    new_tail = head
    for _ in range(length - actual_rotation_count - 1):
        new_tail = new_tail.next

    # Set the next pointer of the current tail to None
    tail.next = head

    # Set the head of the original list to the node after the new tail
    new_head = new_tail.next

    # Set the next pointer of the new tail to None to make it the new tail
    new_tail.next = None

    return new_head

# Function to print the linked list
def print_linked_list(head):
    current = head
    while current:
        print(current.val, end=" -> " if current.next else " -> None\n")
```

```
        current = current.next

# Example usage
# Create linked list: 1 -> 2 -> 3 -> 4 -> 8 -> 6 -> 9
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)
head.next.next.next = ListNode(4)
head.next.next.next.next = ListNode(8)
head.next.next.next.next.next = ListNode(6)
head.next.next.next.next.next.next = ListNode(9)

# Rotate the list by 2 positions to the right
rotated_head = rotate_right(head, 2)

# Print the rotated linked list
print("Rotated linked list:")
print_linked_list(rotated_head)
```

This will output:

```
Rotated linked list:
6 -> 9 -> 1 -> 2 -> 3 -> 4 -> 8 -> None
```

9.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def add_two_numbers(l1, l2):
    dummy_head = ListNode()
    current = dummy_head
    carry = 0

    while l1 or l2:
        # Get values from the current nodes or set to 0 if the node is None
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # Calculate the sum and carry
        total = val1 + val2 + carry
        carry = total // 10

        # Create a new node with the remainder of the sum
        current.next = ListNode(total % 10)
        current = current.next

        # Move to the next nodes
        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    # Add a new node for the carry if it exists
    if carry:
        current.next = ListNode(carry)

    return dummy_head.next

# Function to print the linked list
def print_linked_list(head):
    current = head
```

```
    while current:
        print(current.val, end=" -> " if current.next else " -> None\n")
        current = current.next

# Example usage
# Create linked list: 2 -> 4 -> 3
l1 = ListNode(2)
l1.next = ListNode(4)
l1.next.next = ListNode(3)

# Create linked list: 5 -> 6 -> 4
l2 = ListNode(5)
l2.next = ListNode(6)
l2.next.next = ListNode(4)

# Add the two numbers
result = add_two_numbers(l1, l2)

# Print the result linked list
print("Result linked list:")
print_linked_list(result)

Output:
Result linked list:
7 -> 0 -> 8 -> None
```

10.

The image you sent depicts a linked list with two special pointers for each node:

Next pointer: This behaves like a standard linked list pointer, referencing the next node in the sequence.
Random pointer: This pointer can point to any node arbitrarily within the linked list, or even to itself or null.
The task is to clone this linked list, creating a deep copy in O(N) time, where N is the number of nodes in the list. This means that the copied list should have its own set of nodes, and that the original and copied lists should not share any references. Additionally, both the next and random pointers in the copied list should have the same relative structure as the original list.

There are two approaches to solve this problem:

Using Extra Space:

Create a hash table to map nodes in the original list to their corresponding nodes in the copied list.
Iterate through the original list, creating a new node for each one and adding it to the hash table.
As you create new nodes, set their next pointers to match the next pointers in the original list.
Iterate through the original list again. This time, use the hash table to find the corresponding nodes in the copied list. Set the random pointers in the copied list to point to the appropriate copied nodes.
Without Extra Space:

Iterate through the original list and create a new node for each existing node. Place the new node right after the original node (technically, youâ ™re weaving the new nodes between the original nodes).
Set the next pointer of the new node to point to the next node of the original node.
Now that you have both the original and copied nodes side-by-side, iterate through the list again and assign the random pointers in the copied list to

point to the corresponding copied nodes.
Lastly, untangle the two lists by carefully adjusting the next pointers of the original and copied nodes.
Both approaches achieve a time complexity of O(N), but the first approach uses extra space for the hash table.