linked-list-practice-questions

March 18, 2024

0.0.1 Helper

[1]:
```
class Node:
    def __init__(self, data, next=None):
        """
        Initializes a linked list.

        Parameters:
        - data: The data or value.
        - next: A reference to the next node in the sequence.
        """
        self.data = data
        self.next = next

def traverse(head):
    """
    Traverse through a linked list, printing its elements and connections.

    Parameters:
    - head: The head node of the linked list.

    Prints:
    - Each element of the linked list, along with an arrow indidicating the␣
 ↪next node.
    """
    temp = head
    while temp:
        print(temp.data, end=' -> ')
        temp = temp.next

def length(head):
    """
    Length of the linked list.

    Parameters:
    - head: The head node of the linked list.

    Returns:
```

```
    - count (int): The number of nodes in the linked list.
    """
    temp = head
    count = 0

    while temp:
        count += 1
        temp = temp.next

    return count
```

Problem 1: Define a doubly linked list

Solution: A doubly linked list is a data structure that consists of a sequence of elements called nodes. Each node in a doubly linked list contains two pointers(links) instead of one, pointing to the previous and next nodes in the sequence. This bidirectional linking allows for traversal in both forward and backward directions.

Here's a basic defination of a doubly linked list node in Python:

[85]:
```
class DoublyNode:
    def __init__(self, data=None, prev=None, next=None):
        # Doubly linked list node with data, previous, and next pointers.
        self.data = data
        self.prev = prev
        self.next = next

def print_doubly_linked_list(head):
    # Print the doubly linked list in the forward direction.
    temp = head
    while temp:
        print(temp.data, end=' <-> ')
        temp = temp.next
    print(None)

# Driver code
# Creating instances of DoublyNode
head = DoublyNode(1)
node2 = DoublyNode(2)
node3 = DoublyNode(3)
node4 = DoublyNode(4)
node5 = DoublyNode(5)

# Creating linkage between each DoublyNode instance
head.next = node2
node2.prev, node2.next = head, node3
node3.prev, node3.next = node2, node4
node4.prev, node4.next = node3, node5
```

```
# Print the doubly linked list
print_doubly_linked_list(head)
```

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

Problem 2: Define a function to reverse a linked list in-place.

Solution:

[80]:
```
# Time complexity: O(n), where 'n' is the size of the input linked list.
# Space complexity: O(1), only constant additional space is used.

def reverse(lst):
    # Initialize pointers: 'prev' is initially None, and 'current' starts at␣
 ↪the head of the list.
    prev, current = None, lst

    # Traverse the linked list.
    while current:
        # Store the next node in a temporary variable.
        next_node = current.next
        # Reverse the direction of the current node by updating its next␣
 ↪pointer.
        current.next = prev
        # Move 'prev' and 'current' pointers one step forward.
        prev = current
        current = next_node

    # 'prev' now points to the new head of the reversed linked list.
    return prev

# Driver code
lst = Node(1, Node(2, Node(3, Node(4, Node(5)))))
result = reverse(lst)
traverse(result)
```

```
5 -> 4 -> 3 -> 2 -> 1 ->
```

Problem 3: Detect cycle in a linked list

Solution:

[79]:
```
# Time complexity: O(n/2), where 'n' is the size of the input linked list.
# Space complexity: O(1), only constant additional space is used.

def has_cycle(head):
    # Check for empty list or a list with only one node (no cycle possible).
    if not head or not head.next:
        return False
```

```
    # Initialize two pointers, 'fast' and 'slow', both starting at the head of␣
 ↪the list.
    fast = slow = head

    # Traverse the list using two pointers.
    while fast.next and fast.next.next:
        # Move 'fast' two steps at a time.
        fast = fast.next.next
        # Move 'slow' one step at a time.
        slow = slow.next

        # Check if 'fast' and 'slow' pointers meet at the same node (cycle␣
 ↪detected).
        if fast == slow:
            return True

    # If 'fast' reaches the end of the list without meeting 'slow', no cycle is␣
 ↪detected.
    return False

# Driver code

# Creating Node instances
head = Node(1)
node2 = Node(2)
node3 = Node(3)

# Creating linkage between each and every nodes
head.next = node2
node2.next = node3
node3.next = head

# Checking has cycle or not and print the obtained result
result = has_cycle(head)
print(result)
```

```
True
```

Problem 4: Merged two sorted linked list into one.

Solution:

[76]:
```
# Time complexity: O(min(M, N)), where 'M' and 'N' are the size of the input␣
 ↪linked list1 and list2
# Space complexity: O(1), only constant additional space is used

def merge_sorted_lists(lst1, lst2):
```

```
    # Create a dummy node to represent the begining of the merged list.
    dummy = Node(0)

    # Initialize a pointer 'current' to keep track of the current position in␣
 ↪the merged list.
    current = dummy

    # Merge the two sorted linked lists.
    while lst1 and lst2:
        # Compare the data of current nodes in list1 and list2.
        if lst1.data < lst2.data:
            # If the data in list1 is smaller, append the node from list1 to␣
 ↪the merged list.
            current.next = lst1
            lst1 = lst1.next
        else:
            # If the data in list2 is smaller or equal, append the node from␣
 ↪lst2 to the merged list.
            current.next = lst2
            lst2 = lst2.next

        # Move the 'current' pointer to the newly added node.
        current = current.next

    # Append the remaining nodes from either lst1 or lst2 (whichever is not␣
 ↪empty).
    if lst1:
        current.next = lst1
    elif lst2:
        current.next = lst2

    # Return the head of the merged list (excluding the dummy node).
    return dummy.next

# Driver code
lst1 = Node(1, Node(3, Node(5, Node(6))))
lst2 = Node(2, Node(4, Node(6, Node(8))))
result = merge_sorted_lists(lst1, lst2)
traverse(result)
```

```
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 6 -> 8 ->
```

Problem 5: Write a function to remove nth node from the end in a linked list

Solution:

[75]:
```
# Time complexity: O(n), where 'n' is the size of the input linked list
# Space complexity: O(1), only constant additional space is used.
```

```
def removeNthFromEnd(lst, n):
    # Calculate the length of the linked list.
    lstLen = length(lst)

    # Check for invalid values of 'n'.
    if n == 0 or n > lstLen:
        return lst
    elif n == lstLen:
        # If 'n' is equal to the length of the list, remove the first node.
        lst = lst.next
    else:
        # Traverse the list to the node before the one to be removed.
        current = lst
        j = 0
        while j < (lstLen-n-1):
            current = current.next
            j += 1

        # Remove the nth node by updating the next pointer.
        current.next = current.next.next

    # Return the head of the updated linked list
    return lst

# Driver code
lst = Node(1, Node(2, Node(3, Node(4, Node(5)))))
n = 3
result = removeNthFromEnd(lst, n)
traverse(result)
```

```
1 -> 2 -> 4 -> 5 ->
```

Problem 6: Remove duplicates from a sorted linked list.

Solution:

[74]:
```
# Time complexity: O(n), where 'n' is the size of the input linked list.
# Space complexity: O(1), only constant additional space is used.

def remove_duplicates(lst):
    # Create a dummy node to represent the begining of the updated linked list.
    dummy = Node(0)

    # Initialize pointers: prev points to the last unique node, and current is␣
 ↪used to traverse the list.
    prev, current = dummy, lst
```

```
    # Traverse the input linked list.
    while current:
        # Check if the data of the current node is different from the previous unique node.
        if current.data != prev.data:
            # If different, update the next pointer of the previous node to the current node.
            prev.next = current
            # Update the previous pointer to the current node.
            prev = prev.next

        # Move to the next node in the input linked list.
        current = current.next

    # Set the next pointer of the last unique node to None.
    prev.next = None

    # Return the head of the updated linked list (excluding the dummy node).
    return dummy.next

# Driver code
lst = Node(1, Node(1, Node(2, Node(2, Node(3, Node(4, Node(5)))))))
result = remove_duplicates(lst)
traverse(result)
```

```
1 -> 2 -> 3 -> 4 -> 5 ->
```

Problem 7: Find the intersection of two linked lists

Solution:

[72]:
```
# Time complexity: O(max(M, N)), where 'M' and 'N' are the size of input linked list1 and list2 respectively.
# Space complexity: O(M), where 'M' is the size of the list1

def find_intersection(lst1, lst2):
    # Create an empty set to store visited nodes.
    visited_nodes = set()

    # Create a dummy node to represent the begining of the intersection list.
    intersection = Node(0)
    current = intersection

    # Traverse the first linked list (lst1).
    while lst1:
        # Add the data of each node from lst1 to the set of visited nodes
        visited_nodes.add(lst1.data)
        lst1 = lst1.next
```

```
    # Traverse the second linked list (lst2).
    while lst2:
        # If the data is found, create a new node in the intersection list and update the current pointer.
        if lst2.data in visited_nodes:
            current.next = lst2
            current = current.next

        # Move to the next node in lst2
        lst2 = lst2.next

    # Set the next pointer of the last node in the intersection list to None.
    current.next = None

    # Return the head of the intersection list (excluding the dummy node).
    return intersection.next

# Driver code
lst1 = Node(1, Node(2, Node(3, Node(4, Node(8, Node(6, Node(9)))))))
lst2 = Node(5, Node(1, Node(6, Node(7))))
result = find_intersection(lst1, lst2)
traverse(result)
```

```
1 -> 6 ->
```

Problem 8: Rotate a linked list by k positions to the right

Solution:

[84]:
```
# Time complexity: O(n), where 'n' is the size of the input linked list.
# Space complexity: O(1), only constant additional space is used.

def rotateLinkedLists(lst, k):
    # Check for edge cases where rotation is not necessary or not possible.
    if lst is None or k <= 0:
        return lst

    # Calculate the effective rotation by taking the modulo with the length of the linked list.
    k = k % length(lst)

    # If the effective rotation is 0, no rotation is needed.
    if k == 0:
        return lst

    # Initialize pointers: 'current' points to the head of the list, and 'j' is the counter.
```

```
    current, j = lst, 0

    # Traverse the list to the (k-1)th node.
    while j < k - 1 and current.next:
        current = current.next
        j += 1

    # Separate the remaining nodes after rotation.
    remaining_nodes = current.next
    current.next = None

    # Move to the last node of the remaining nodes.
    end_node = remaining_nodes
    while end_node.next:
        end_node = end_node.next

    # Connect the last node of the remaining nodes to the original head.
    end_node.next = lst

    # Return the head of the rotated linked list.
    return remaining_nodes

# Driver code
lst = Node(1, Node(2, Node(3, Node(4, Node(5)))))
k = 4
result = rotateLinkedLists(lst, k)
traverse(result)
```

```
5 -> 1 -> 2 -> 3 -> 4 ->
```

Problem 9: Add two numbers represented by linked lists.

Solution:

[82]:
```
# Time complexity: O(max(M, N)), where 'M' and 'N' are the sizes of the input␣
 ↪linked list1 and list2
# Space complexity: O(max(M, N))

def addLinkedListsNumbers(lst1, lst2):
    # Create a dummy node to represent the beginning of the result linked list.
    dummy = Node(0)
    # Initialize 'current' pointer to the dummy node.
    current = dummy
    # Initialize 'carry' to 0.
    carry = 0

    # Continue the loop until there are no more nodes in lst1, lst2, or there's␣
 ↪a carry.
```

```
    while lst1 or lst2 or carry:
        # Extract the current digits from lst1 and lst2 (or use 0 if the node is None).
        x = lst1.data if lst1 else 0
        y = lst2.data if lst2 else 0

        # Sum the current digits along with the carry from the previous calculation.
        _sum = x + y + carry

        # Update carry for the next calculation.
        carry = _sum // 10

        # Create a new node with the result of the current digit sum and update the 'current' pointer.
        current.next = Node(_sum % 10)
        current = current.next

        # Move to the next nodes in lst1 and lst2 if available.
        if lst1:
            lst1 = lst1.next
        if lst2:
            lst2 = lst2.next

    # Return the head of the result linked list (excluding the dummy node).
    return dummy.next

# Driver code
lst1 = Node(9, Node(8, Node(7)))
lst2 = Node(6, Node(5, Node(4)))
result = addLinkedListsNumbers(lst1, lst2)
traverse(result)
```

```
5 -> 4 -> 2 -> 1 ->
```

Problem 10: Clone a linked list with next and random pointer

Solution:

[1]:
```
# Time complexity: O(N), where 'N' is the size of the original linked list.
# Space complexity: O(N), where 'N' is the size of the original linked list.

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

```python
def clone_linked_list(head):
    if not head:
        return None

    # Step 1: Insert copied nodes next to the original nodes
    current = head
    while current:
        copied_node = Node(current.data)
        copied_node.next = current.next
        current.next = copied_node
        current = copied_node.next

    # Step2: Update arbitrary pointers of copied nodes
    current = head
    while current:
        if current.arbitrary:
            current.next.arbitrary = current.arbitrary.next
        current = current.next.next

    # Step3: Separate original and cloned lists
    original_head = head
    cloned_head = head.next
    cloned_current = cloned_head
    while original_head:
        original_head.next = original_head.next.next
        original_head = original_head.next
        if cloned_current.next:
            cloned_current.next = cloned_current.next.next
            cloned_current = cloned_current.next

    return cloned_head

# Example usage
# Create a sample linked list with arbitrary pointers
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
node4 = Node(4)
node5 = Node(5)

node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node5

node1.arbitrary = node3
node2.arbitrary = node1
```

```
node3.arbitrary = node5
node4.arbitrary = node3
node5.arbitrary = node2

# Clone the linked list
cloned_head = clone_linked_list(node1)

# Printing cloned list abribtrary pointers for verification
current = cloned_head
while current:
    print("Data:", current.data, "Arbitrary Data:", current.arbitrary.data if␣
 ↪current.arbitrary else None)
    current = current.next
```

```
Data: 1 Arbitrary Data: 3
Data: 2 Arbitrary Data: 1
Data: 3 Arbitrary Data: 5
Data: 4 Arbitrary Data: 3
Data: 5 Arbitrary Data: 2
```