LINKED LIST – PYTHON

Table of Contents

1. Construtor

- Node: Represents a node in a singly linked list.

PDS&A > Node.py > LinkedList

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None


class LinkedList:
    def __init__(self, value):
        new_node = Node(value)
        self.head = new_node
        self.tail = new_node
        self.length = 1
```

2. Print

- print_list: Prints out all the elements in the list.

```python
    def print_list(self):
        temp = self.head
        while temp is not None:
            print(temp.value)
            temp = temp.next
```

3. Append

- append: Adds an element to the end of the list.

```python
    def append(self, value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
        self.length += 1
        return True
```

4. Pop

- pop: Removes the last element from the list and return it.

```python
    def pop(self):
        if self.length == 0:
            return None
        temp = self.head
        pre = self.head
        while(temp.next):
            pre = temp
            temp = temp.next
        self.tail = pre
        self.tail.next = None
        self.length -= 1
        if self.length == 0:
            self.head = None
            self.tail = None
        return temp
```

5. Prepend

- prepend: Adds an element to the start of the list.

```python
    def prepend(self, value):
        new_node = Node(value)
        if self.length == 0:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head = new_node
        self.length += 1
        return True
```

6. Pop_first

- pop_first: Removes the first element from the list and return it.

```python
    def pop_first(self):
        if self.length == 0:
            return None
        temp = self.head
        self.head = self.head.next
        temp.next = None
        self.length -= 1
        if self.length == 0:
            self.tail = None
        return temp
```

7. Get

- get: Gets the node at a given index.

```python
    def get(self, index):
        if index < 0 or index >= self.length:
            return None
        temp = self.head
        for _ in range(index):
            temp = temp.next
        return temp
```

8. Set_value

- set_value: Sets the value of the node at a given index.

```python
    def set_value(self, index, value):
        temp = self.get(index)
        if temp:
            temp.value = value
            return True
        return False
```

9. Insert

- insert: Inserts a node at a given index.

```python
    def insert(self, index, value):
        if index < 0 or index > self.length:
            return False
        if index == 0:
            return self.prepend(value)
        if index == self.length:
            return self.append(value)
        new_node = Node(value)
        temp = self.get(index - 1)
        new_node.next = temp.next
        temp.next = new_node
        self.length += 1
        return True
```

10. Remove

- remove: Removes a node at a given index.

```python
    def remove(self, index):
        if index < 0 or index >= self.length:
            return None
        if index == 0:
            return self.pop_first()
        if index == self.length - 1:
            return self.pop()
        pre = self.get(index - 1)
        temp = pre.next
        pre.next = temp.next
        temp.next = None
        self.length -= 1
        return temp
```

11. Reverse

- reverse: Reverses the linked list.

```python
    def reverse(self):
        if self.length == 0:
            return None
        temp = self.head
        self.head = self.tail
        self.tail = temp
        before = None
        after = temp.next
        for _ in range(self.length):
            after = temp.next
            temp.next = before
            before = temp
            temp = after
```

12. Find_middle_node

- Find_middle_node: finding the middle node of a linked list, and the constraints typically associated with it are:

    - Single Pass: The algorithm should traverse the linked list only once. Using a double pointer technique (one moving faster and another slower) is a popular approach to satisfy this constraint.

    - No Length Utilization: The solution should not first calculate the length of the linked list and then locate the middle. It should find the middle in a single pass without the need for the length information.

```python
    def find_middle_node(self):
        if self.head is None:
            print("List is Empty")
            return None
        fast = slow = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
        print(slow.value)
        return slow
```

13. Has_loop

- Has_loop: The provided code is an implementation of Floyd's cycle-finding algorithm (often called the "tortoise and the hare" approach) to detect loops or cycles in a linked list.

  - Two pointers, slow and fast, move through the list at different speeds;
  - If there is a loop in the linked list, the fast pointer will eventually catch up to the slow pointer, and they will be equal (slow == fast);

```python
    def has_loop(self):
        if self.head is None:
            print("List is Empty")
            return None
        fast = slow = self.head
        while fast is not None and fast.next is not None:
            slow = slow.next
            fast = fast.next.next
            if slow == fast:
                return True
        return False
```

14. find_kth_from_end

- find_kth_from_end: Implement the find_kth_from_end function, which takes the LinkedList (ll) and an integer k as input, and returns the k-th node from the end of the linked list WITHOUT USING LENGTH.

Given this LinkedList:

  - 1 -> 2 -> 3 -> 4
  - If k=1 then return the first node from the end (the last node) which contains the value of 4.
  - If k=2 then return the second node from the end which contains the value of 3, etc.
  - If the linked list has fewer than k items, the program should return None.

The find_kth_from_end function should follow these requirements:

- The function should utilize two pointers, slow and fast, initialized to the head of the linked list.

- The fast pointer should move k nodes ahead in the list.
- If the fast pointer becomes None before moving k nodes, the function should return None, as the list is shorter than k nodes.
- The slow and fast pointers should then move forward in the list at the same time until the fast pointer reaches the end of the list.
- The function should return the slow pointer, which will be at the k-th position from the end of the list.

```python
    def find_kth_from_end(self, k):
        fast = slow = self.head
        for _ in range(k): # Move fast pointer k steps ahead
            if fast is None:  # If list has less than k nodes
                return None
            fast = fast.next
        while fast is not None: # Now, move both pointers until fast reaches the end
            fast = fast.next
            slow = slow.next
        print(slow.value)
        return slow # When fast pointer is at the end, slow pointer will be at the kth node from the end
```