Linked List

Introduction

A linked list is a linear data structure where elements, called nodes, are not stored in contiguous memory locations. Each node contains two parts:

1. Data
2. A reference (or link) to the next node in the sequence.

This structure allows for efficient insertion and deletion of elements.

Key Concepts

1. **Node**: The basic unit of a linked list, containing data and a pointer to the next node.
2. **Head**: The first node in the linked list.
3. **Tail**: The last node, which points to `None`.
4. **Null (`None`)**: Indicates the end of the linked list.

Types of Linked Lists

1. **Singly Linked List**: Each node points to the next node.
2. **Doubly Linked List**: Each node points to both the next and the previous node.
3. **Circular Linked List**: The last node points back to the first node, forming a circle.

Implementation of a Singly Linked List in Python

Let's start with a basic singly linked list.

1. **Node Class**: Defines the structure of a node.
2. **LinkedList Class**: Manages the linked list operations.

```
class Node:
    def __init__(self, data):
```

```
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def prepend(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def delete_with_value(self, data):
        if not self.head:
            return
        if self.head.data == data:
            self.head = self.head.next
            return
        current = self.head
        while current.next:
            if current.next.data == data:
                current.next = current.next.next
                return
            current = current.next

    def print_list(self):
```

```
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Usage
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.prepend(0)
ll.print_list()
ll.delete_with_value(2)
ll.print_list()
```

Common Questions

1. **Why use a linked list over an array?**
    - Linked lists offer efficient insertions and deletions compared to arrays, especially when dealing with dynamic data where the size isn't known beforehand.
2. **What is the time complexity of common operations?**
    - Insertion: O(1) at the head, O(n) at the tail.
    - Deletion: O(1) if the node is known, O(n) otherwise.
    - Access: O(n).
3. **What are the limitations of linked lists?**
    - Random access is not possible.
    - Memory overhead due to storing pointers.

Code Examples

Appending to a Linked List:

```
ll.append(4)
ll.print_list()  # Output: 0 -> 1 -> 3 -> 4 -> None
```

Prepending to a Linked List:

```
ll.prepend(-1)
ll.print_list()  # Output: -1 -> 0 -> 1 -> 3 -> 4 -> None
```

Deleting a Node:

```
ll.delete_with_value(1)
ll.print_list()  # Output: -1 -> 0 -> 3 -> 4 -> None
```

Practice Questions

1. Reverse a Linked List

Tips:

- Use three pointers: `prev`, `current`, and `next`.
- Iterate through the list and adjust the `next` pointer of each node to point to the previous node.

Steps:

1. Initialize `prev` to `None` and `current` to `head`.
2. Iterate through the list.
3. Save `current.next` in `next`.
4. Set `current.next` to `prev`.
5. Move `prev` and `current` one step forward.
6. Update `head` to `prev`.

Sample Starting Code:

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

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def reverse(self):
        # TODO: Implement this method
        pass

    def print_list(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Usage
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
```

```
ll.print_list()  # Output: 1 -> 2 -> 3 -> None
ll.reverse()
ll.print_list()  # Expected Output after reverse: 3 -> 2 -> 1
-> None
```

Expected Input and Output:

- Input: `1 -> 2 -> 3 -> None`
- Output: `3 -> 2 -> 1 -> None`

2. Detect a Cycle in a Linked List

Tips:

- Use Floyd's Cycle-Finding Algorithm (Tortoise and Hare).
- Use two pointers: slow (moves one step) and fast (moves two steps).

Steps:

1. Initialize `slow` and `fast` to `head`.
2. Iterate through the list.
3. Move `slow` one step and `fast` two steps.
4. If `slow` equals `fast`, a cycle exists.
5. If `fast` reaches `None`, no cycle exists.

Sample Starting Code:

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

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
```

```
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def detect_cycle(self):
        # TODO: Implement this method
        pass

# Usage
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.head.next.next.next = ll.head  # Create a cycle
print(ll.detect_cycle())  # Expected Output: True
```

Expected Input and Output:

- Input: `1 -> 2 -> 3 -> (points back to 1)`
- Output: `True`

3. Find the Middle Node

Tips:

- Use two pointers: slow and fast.
- Slow pointer moves one step, fast pointer moves two steps.

Steps:

1. Initialize `slow` and `fast` to `head`.
2. Move `slow` one step and `fast` two steps.

3. When `fast` reaches the end, `slow` is at the middle.

Sample Starting Code:

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

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def find_middle(self):
        # TODO: Implement this method
        pass

# Usage
ll = LinkedList()
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(4)
ll.append(5)
print(ll.find_middle())  # Expected Output: 3
```

Expected Input and Output:

- Input: `1 -> 2 -> 3 -> 4 -> 5 -> None`
- Output: `3`

4. Merge Two Sorted Linked Lists

Tips:

- Use a dummy node to simplify the merge process.
- Compare nodes of both lists and attach the smaller one to the merged list.

Steps:

1. Initialize a dummy node.
2. Use two pointers for the two lists.
3. Compare nodes and attach the smaller node to the merged list.
4. Continue until one list is exhausted.
5. Attach the remaining nodes.

Sample Starting Code:

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

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
```

```
        while last.next:
            last = last.next
        last.next = new_node

def merge_sorted_lists(l1, l2):
    # TODO: Implement this method
    pass

# Usage
ll1 = LinkedList()
ll1.append(1)
ll1.append(3)
ll1.append(5)
ll2 = LinkedList()
ll2.append(2)
ll2.append(4)
ll2.append(6)
merged_list = merge_sorted_lists(ll1.head, ll2.head)

# Print merged list
current = merged_list
while current:
    print(current.data, end=" -> ")
    current = current.next
print("None")
# Expected Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6 -> None
```

Expected Input and Output:

- Input: `1 -> 3 -> 5 -> None` and `2 -> 4 -> 6 -> None`
- Output: `1 -> 2 -> 3 -> 4 -> 5 -> 6 -> None`

5. Remove Duplicates from a Sorted Linked List

Tips:

- Traverse the list and compare each node with the next node.

- If they are the same, skip the next node.

Steps:

1. Initialize `current` to `head`.
2. Iterate through the list.
3. If `current.data` equals `current.next.data`, skip the next node.
4. Move `current` to the next node.

Sample Starting Code:

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

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last = self.head
        while last.next:
            last = last.next
        last.next = new_node

    def remove_duplicates(self):
        # TODO: Implement this method
        pass

# Usage
ll = LinkedList()
```

```
ll.append(1)
ll.append(1)
ll.append(2)
ll.append(3)
ll.append(3)
ll.remove_duplicates()
ll.print_list()  # Expected Output: 1 -> 2 -> 3 -> None
```

Expected Input and Output:

- Input: `1 -> 1 -> 2 -> 3 -> 3 -> None`
- Output: `1 -> 2 -> 3 -> None`

These expanded explanations, tips, steps, and sample starting codes should help your students understand and tackle the linked list problems effectively.

Conclusion

Linked lists are fundamental data structures that provide flexibility in memory allocation and efficient insertions and deletions. Understanding and implementing linked lists in Python equips you with a deeper grasp of data structures, preparing you for more complex programming challenges. Practice the above questions to solidify your understanding.