TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Session 2: Linked Lists II

Session Overview

In this session, students will deepen their ability to work with linked lists, solving problems involving traversal, node manipulation, and cycle detection. These exercises will cover techniques common to many linked lists problems and enhance their understanding of this crucial data structure.

> You can find all resources from today including session slide decks, session recordings, and more on the resources tab

🎢 Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.

👨‍💻 Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
    - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!
- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

▶ **Note on Expectations**

🔎 Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▶ **Comment on UPI**

▶ **UPI Example**

Breakout Problems Session 2

▼ Standard Problem Set Version 1

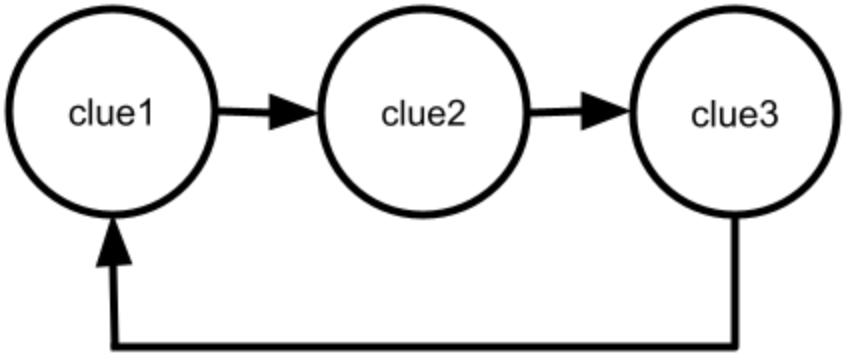
Problem 1: Wild Goose Chase

You're a detective and have been given an anonymous tip on your latest case, but something about it seems fishy - you suspect the clue might be a red herring meant to send you around in circles. Write a function `is_circular()` that accepts the head of a singly linked list `clues` and returns `True` if the tail of the linked list points at the head of the linked list. Otherwise, return `False`.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

def is_circular(clues):
        pass
```

Example Usage:




```
clue1 = Node("The stolen goods are at an abandoned warehouse")
clue2 = Node("The mayor is accepting bribes")
clue3 = Node("They dumped their disguise in the lake")
clue1.next = clue2
clue2.next = clue3
clue3.next = clue1

print(is_circular(clue1))
```

Example Output:

```
True
```

▶ 💡 **Hint: Which technique?**

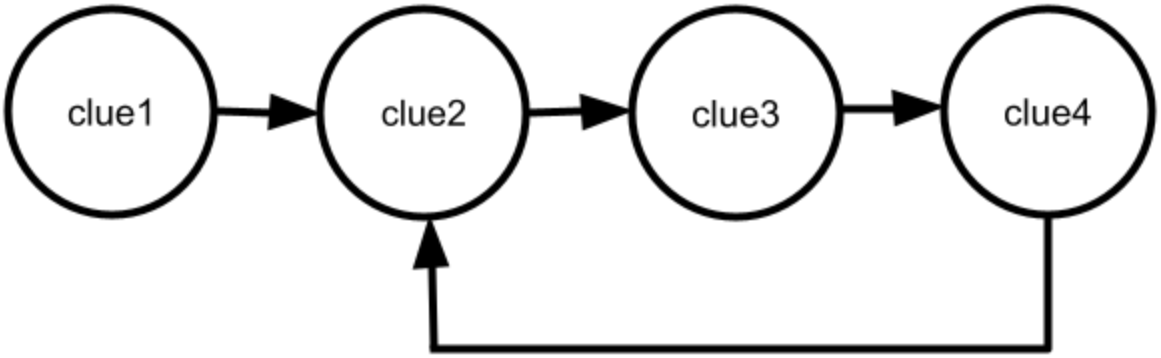
Problem 2: Breaking the Cycle

All the clues that lead us in circles are false evidence we need to purge! Given the head of a linked list `evidence`, clean up the evidence list by identifying any false clues. Write a function `collect_false_evidence()` that returns an array containing all `value`s that are part of any cycle in `evidence`. Return the `value`s in any order.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

def collect_false_evidence(evidence):
        pass
```

Example Usage:




```
clue1 = Node("Unmarked sedan seen near the crime scene")
clue2 = Node("The stolen goods are at an abandoned warehouse")
clue3 = Node("The mayor is accepting bribes")
clue4 = Node("They dumped their disguise in the lake")
clue1.next = clue2
clue2.next = clue3
clue3.next = clue4
clue4.next = clue2

clue5 = Node("A masked figure was seen fleeing the scene")
clue6 = Node("Footprints lead to the nearby woods")
clue7 = Node("A broken window was found at the back")
clue5.next = clue6
clue6.next = clue7

print(collect_false_evidence(clue1))
print(collect_false_evidence(clue5))
```

Example Output:

```
['The stolen goods are at an abandoned warehouse', 'The mayor is accepting bribes',
'They dumped their disguise in the lake']
[]
```

▶ 💡 **Hint: Multiple Pass Technique**

Problem 3: Prioritizing Suspects

You've identified a list of suspect, but time is limited and you won't be able to question all of them today. Write a function `partition()` to help prioritize the order in which you question suspects. Given the head of a linked list of integers `suspect_ratings`, where each integer represents the suspiciousness of the a given suspect and a value `threshold`, partition the linked list such that all nodes with values greater than `threshold` come before nodes with values less than or equal to `threshold`.

Return the head of the partitioned list.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def partition(suspect_ratings, threshold):
        pass
```

Example Usage:

```
suspect_ratings = Node(1, Node(4, Node(3, Node(2, Node(5, Node(2))))))

print_linked_list(partition(suspect_ratings, 3))
```

Example Output:

```
4 -> 5 -> 1 -> 3 -> 2 -> 2
Explanation:
Note that nodes 4 and 5 can be in any order in the result list so long as they come
3, 2, and 1.
Similarly, 3, 2, and 1 can come in any order so long as they are after 4 and 5.
5 -> 4 -> 3 -> 1 -> 2 -> 2 would also be a possible acceptable answer
```

▶ 💡 **Hint: Temporary Head Technique**

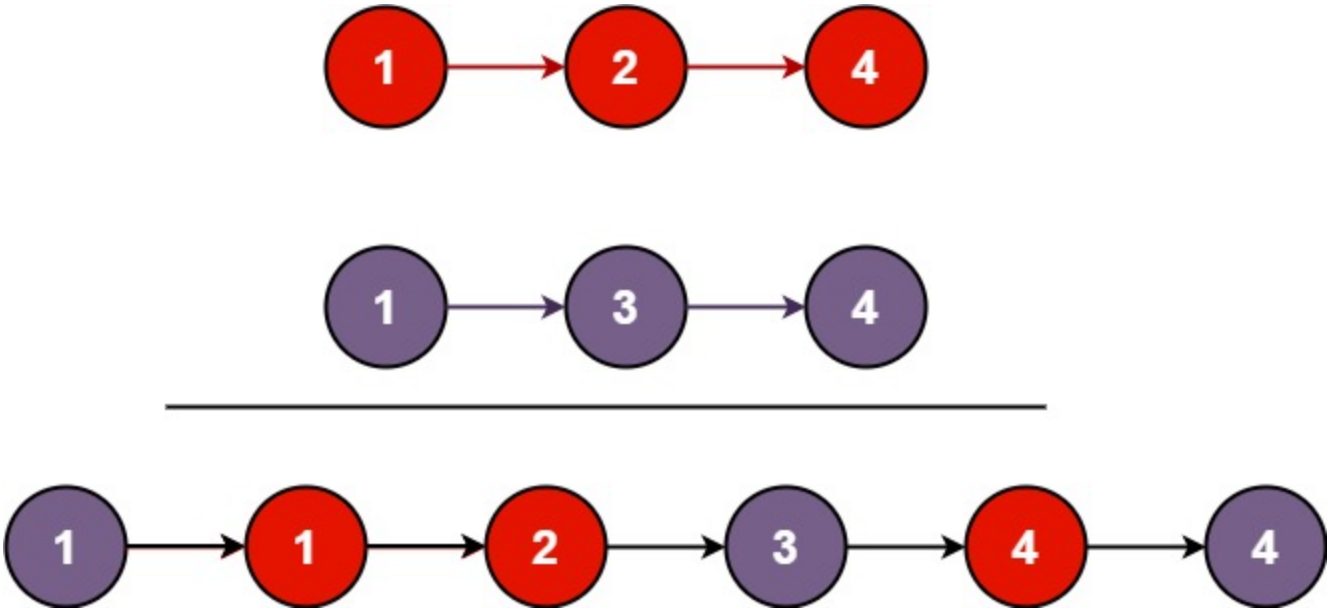
Problem 4: Puzzling it Out

A new witness has emerged and provided a new account of events the night of the crime. Given the heads of two sorted linked lists, `known_timeline` and `witness_timeline`, each representing a numbered sequence of events, merge the two timelines into one **sorted** sequence of events. The resulting linked list should be made by splicing together the nodes of the first two timelines. Return the head of the merged timeline.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_timelines(known_timeline, witness_timeline):
        pass
```

Example Usage:




```
known_timeline = Node(1, Node(2, Node(4)))
witness_timeline = Node(1, Node(3, Node(4)))

print_linked_list(merge_timelines(known_timeline, witness_timeline))
```

Example Output:

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

Problem 5: A New Perspective

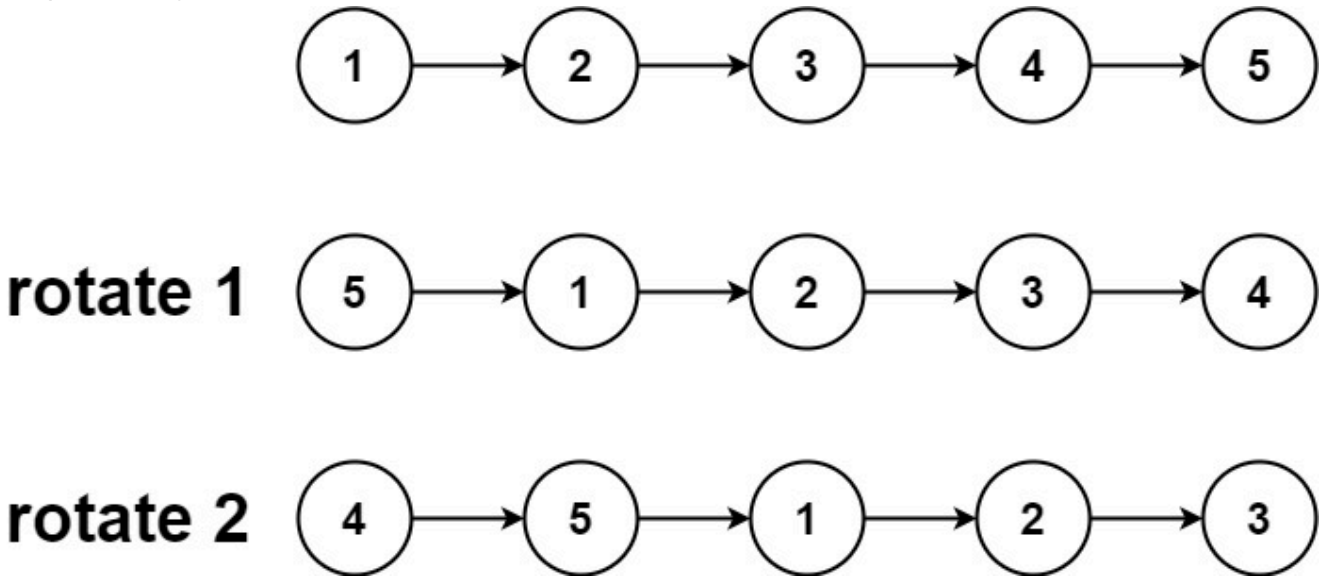
You're having a tough time making a break in the case, and it's time to shake things up to gain a new perspective. Given the head of a linked list of numbered pieces of evidence `evidence`, and a non-negative integer `k`, rotate the list to the right by `k` places. Return the head of the rotated list.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

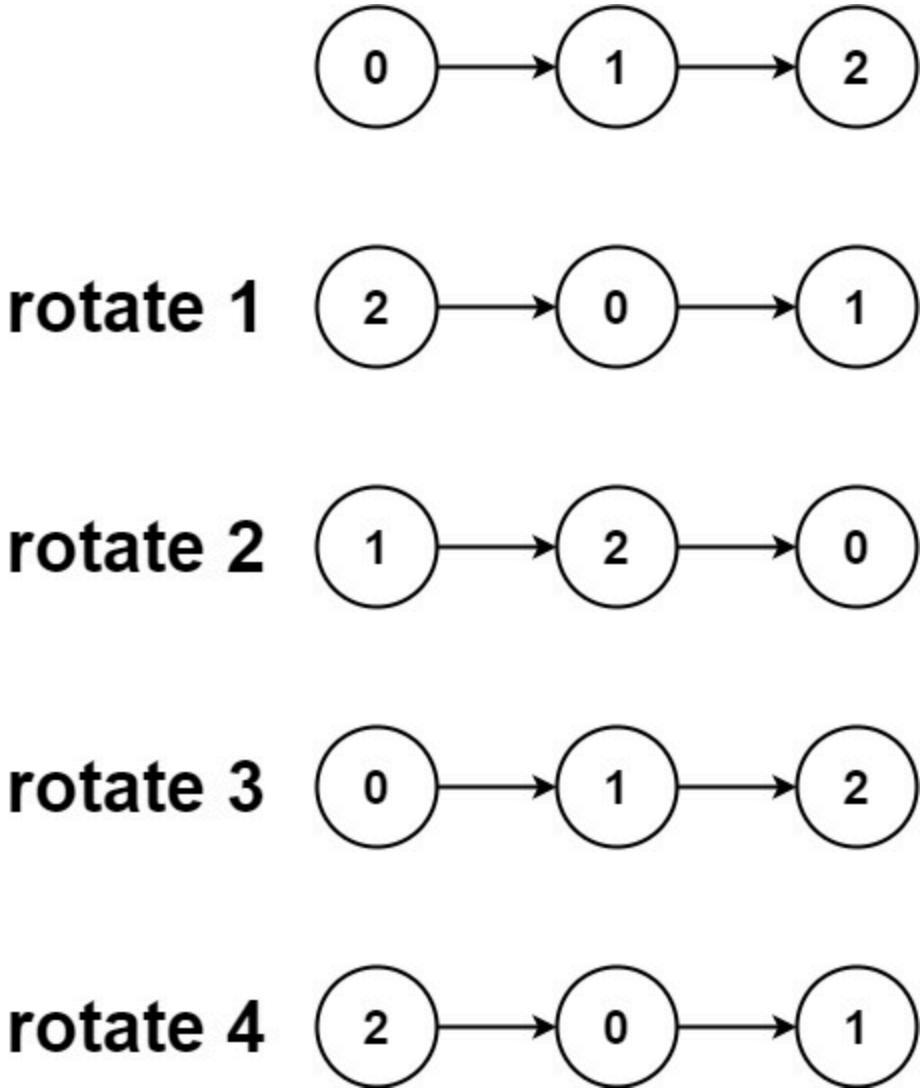
```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def rotate_right(head, k):
        pass
```

Example Usage:

Example 1 Image



Example 2 Image




```
evidence_list1 = Node(1, Node(2, Node(3, Node(4, Node(5)))))
evidence_list2 = Node(0, Noe(1, Node(2)))

print_linked_list(rotate_right(evidence_list1, 2))
print_linked_list(rotate_right(evidence_list2, 4))
```

Example Output:

```
4 -> 5 -> 1 -> 2 -> 3
2 -> 0 -> 1
```

Problem 6: Adding Up the Evidence

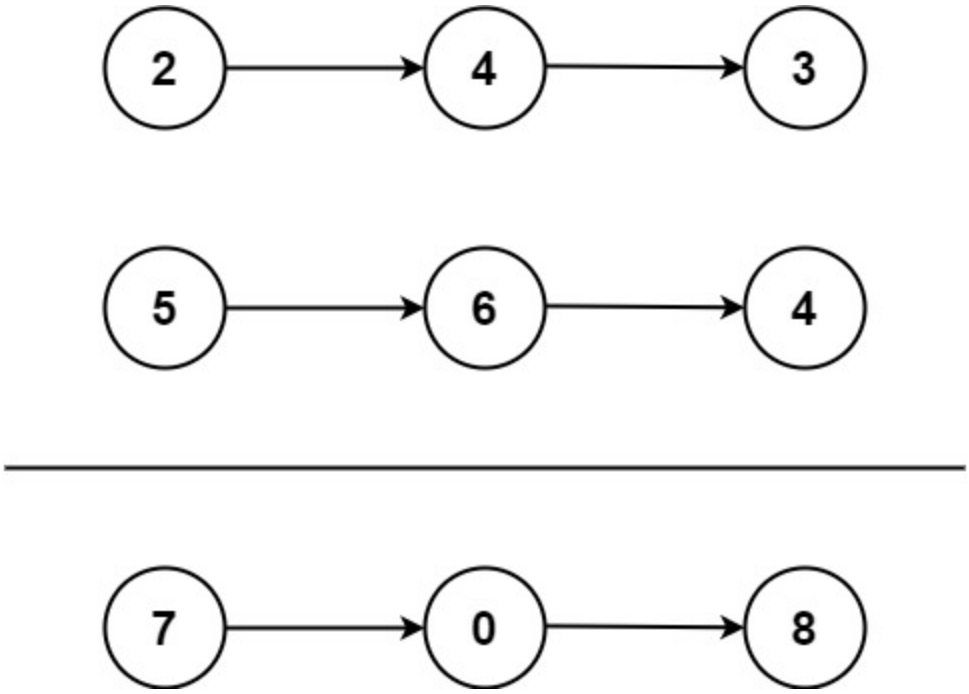
You have all your evidence, and it's time to sum it to the final answer! You are given the heads of two non-empty non-empty linked lists `head_a` and `head_b` representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

The digits of the sum should also be stored in reverse order with each node containing a single digit.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def add_two_numbers(head_a, head_b):
    pass
```

Example Usage:




```
head_a = Node(2, Node(4, Node(3))) # 342
head_b = Node(5, Node(6, Node(4))) # 465

print_linked_list(add_two_numbers(head_a, head_b))
```

Example Output:

```
7 -> 0 -> 8
Explanation: 342 + 465 = 807
```

Close Section

▼ Standard Problem Set Version 2

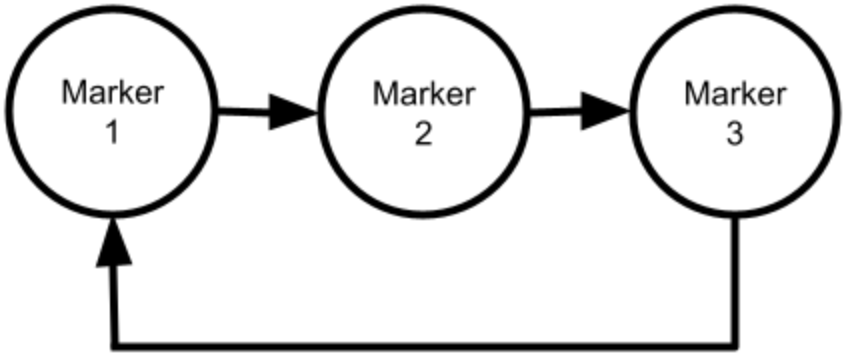
Problem 1: Measuring Loop Length

As a trail worker, you've been tasked with measuring the length of a loop trail that circles back to its starting point. Given the head of a linked list `trailhead` where each node represents a trail marker and the last marker points back to the first marker, return the length of the trail. Assume the length of the trail is equal to the number of markers.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

def trail_length(trailhead):
        pass
```

Example Usage:




```
marker1 = Node("Marker 1")
marker2 = Node("Marker 2")
marker3 = Node("Marker 3")
marker1.next = marker2
marker2.next = marker3
marker3.next = marker1

print(trail_length(marker1))
```

Example Output:

```
3
```

▶ 💡 **Hint: Which technique?**

Problem 2: Clearing the Path

While maintaining a trail, you discover that some parts of the path loop back on themselves, creating confusing detours. Given the head of a linked list that may contain cycles `trailhead`, wite a function that removes any loops/cycles in the trail ensuring a clear, straightforward path. Return the head of the cleared trail.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

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

# For testing - careful this will cause an infinite loop when used on a list w/cycle.
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def clear_trail(trailhead):
        pass
```

Example Usage:

```
marker1 = Node("Trailhead")
marker2 = Node("Trail Fork")
marker3 = Node("The Falls")
marker4 = Node("Peak")
marker1.next = marker2
marker2.next = marker3
marker3.next = marker4
marker4.next = marker2

print_linked_list(clear_trail(marker1))
```

Example Output:

```
Trailhead -> Trail Fork -> The Falls -> Peak
```

▶ 💡 **Hint: Multiple Pass Technique**

Problem 3: Removing Duplicate Markers

When clearing an old trail, you notice some markers have been placed more than once, confusing hikers. Given the head of a sorted linked list of numbered trail markers, `trailhead`, write a function that removes all duplicate markers, keeping only the unique ones. Return the head of the updated trail.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def remove_duplicate_markers(trailhead):
    pass
```

Example Usage:

```
trailhead = Node(1, Node(2, Node(3, Node(3, Node(4)))))

print_linked_list(remove_duplicate_markers(trailhead))
```

Example Output:

```
1 -> 2 -> 4
Explanation: 3 appears more than once so it is deleted from the list
```

▶ 💡 **Hint: Temporary Head Technique**

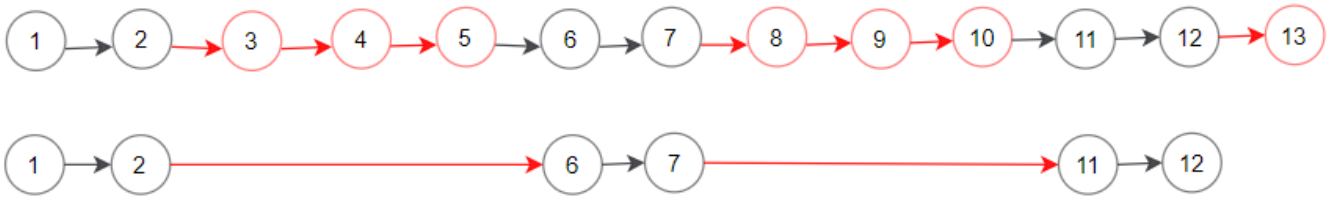
Problem 4: Controlled Burns

You are working with local foresters on a section of trail through local wilderness with particularly dense forests. The foresters recommend doing controlled burns on certain sections of the forest to help decrease severe wildfire risk and promote biodiversity which means certain parts of the trail will be off limits for the upcoming season. Given the head of a linked list of trail markers, `trailhead` and two integers `m` and `n`, write a function to traverse the trail, keeping only the first `m` markers, and then removing the next `n` markers. Continue this pattern until the end of the trail is reached. Return the head of the updated trail.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def selective_trail_clearing(trailhead, m, n):
    pass
```

Example usage:

Example 1




```
trailhead = Node(1, Node(2, Node(3, Node(4, Node(5, Node(6, Node(7, Node(8, Node(9, I

print_linked_list(selective_trail_clearing(trailhead, 2, 3))
```

Example Output:

```
1 -> 2 -> 6 -> 7 -> 11 -> 12
Explanation: Keep the first (m = 2) nodes starting from the head of the linked List
(1 -> 2) show in black nodes.
Delete the next (n = 3) nodes (3 -> 4 -> 5) show in red nodes.
Continue with the same procedure until reaching the tail of the Linked List.
```

Problem 5: Geocaching

You are hiking on a trail that has a geocache hidden at each marker. Each cache is also labeled with a `0` or `1`. The geocaches are arranged in a sequence, forming a binary code that represents the coordinates of a special, hidden cache. The most significant bit is at the first marker on the trail. Given the head of a linked list `cache_labels` representing the sequence of `0`s and `1`s you found at each marker, write a function `locate_cache()` that decodes the sequence and returns the decimal value of the hidden cache's coordinates.

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

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def locate_cache(cache_labels):
    pass
```

Example Usage:

```
cache_labels = Node(1, Node(0, Node(1))) # 101 base 2

print(locate_cache(cache_labels))
```

Example Output:

```
5
Explanation: (101) in base 2 = (5) in base 10
```

▶ 💡 **Hint: Binary to Decimal**

Problem 6: Merging Trail Segments

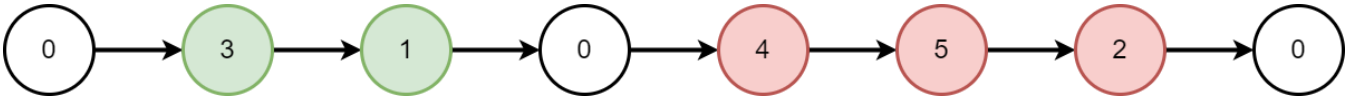
While constructing a new trail, you've set up several segments separated by temporary markers. Once the segments are ready, you want to merge them into continuous trails. Given the head of a linked list of trail markers `trailhead`, merge the nodes between the temporary markers (`0`s) by summing their values into a single marker. The final trail should not contain any temporary markers. Return the head of the merged trail.

Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

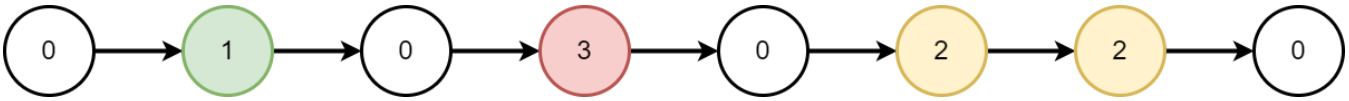
```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

# For testing
def print_linked_list(head):
    current = head
    while current:
        print(current.value, end=" -> " if current.next else "\n")
        current = current.next

def merge_trail(trailhead):
    pass
```

Example Usage:

Example 1



Example 2




```
trail1 = Node(0, Node(3, Node(1, Node(0, Node(4, Node(5, Node(4, Node(2, Node(0))))))
trail2 = Node(0, Node(1, Node(0, Node(3, Node(0, Node(2, Node(2, Node(0)))))))))

print_linked_list(merge_trail(trail1))
print_linked_list(merge_trail(trail2))
```

Example Output:

```
4 -> 11
Example 1 Explanation:
The modified list contains
- The sum of the nodes marked in green: 3 + 1 = 4.
- The sum of the nodes marked in red: 4 + 5 + 2 = 11.

1 -> 3 -> 4
Example 2 Explanation: The modified list contains
- The sum of the nodes marked in green: 1 = 1.
- The sum of the nodes marked in red: 3 = 3.
- The sum of the nodes marked in yellow: 2 + 2 = 4.
```

Close Section

▸ Advanced Problem Set Version 1

Advanced Problem Set Version 2