TIP102 | Intermediate Technical Interview Prep

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

Session 1: 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 1

▼ Standard Problem Set Version 1

Problem 1: Building a Playlist

The assignment statement to the `top_hits_2010s` variable below creates the linked list `Uptown Funk -> Party Rock Anthem -> Bad Romance`. Break apart the assignment statement into multiple lines with one call to the `Node` constructor per line to recreate the list.

```
class SongNode:
        def __init__(self, song, next=None):
                self.song = song
                self.next = next

# For testing
def print_linked_list(node):
    current = node
    while current:
        print(current.song, end=" -> " if current.next else "")
        current = current.next
    print()

top_hits_2010s = SongNode("Uptown Funk", SongNode("Party Rock Anthem", SongNode("Bad
```

Example Usage:

```
print_linked_list(top_hits_2010s)
```

Example Output:

```
Uptown Funk -> Party Rock Anthem -> Bad Romance
```

▶ 💡 **Hint: Nested Constructors**

Problem 2: Top Artists

Given the head of a linked list `playlist`, return a dictionary that maps each *artist* in the list to its frequency.

Evaluate the time 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 SongNode:
        def __init__(self, song, artist, next=None):
                self.song = song
        self.artist = artist
                self.next = next

# For testing
def print_linked_list(node):
    current = node
    while current:
        print((current.song, current.artist), end=" -> " if current.next else "")
        current = current.next
    print()


def get_artist_frequency(playlist):
        pass
```

Example Usage:

```
playlist = SongNode("Saturn", "SZA",
                SongNode("Who", "Jimin",
                        SongNode("Espresso", "Sabrina Carpenter",
                                SongNode("Snooze", "SZA"))))

get_artist_frequency(playlist)
```

Example Output:

```
{ "SZA": 2, "Jimin" : 1, "Sabrina Carpenter": 1}
```

Problem 3: Glitching Out

The following code attempts to remove the first node with a given `song` from a singly linked list with head `playlist_head` but it contains a bug!

Step 1: Copy this code into Replit.

Step 2: Create your own test cases to run the code against, and use print statements and the stack trace to identify and fix the bug so that the function correctly removes a node by value from the list.

Step 3: Evaluate the time and space complexity of the fixed solution. Define your variables and provide a rationale for why you believe the solution has the stated time and space complexity.

```
class SongNode:
        def __init__(self, song, artist, next=None):
                self.song = song
        self.artist = artist
                self.next = next

# For testing
def print_linked_list(node):
    current = node
    while current:
        print((current.song, current.artist), end=" -> " if current.next else "")
        current = current.next
    print()


# Function with a bug!
def remove_song(playlist_head, song):
    if not playlist_head:
        return None
    if playlist_head.song == song:
        return playlist_head.next

    current = playlist_head
    while current.next:
        if current.next.song == song:
            current = current.next.next
            return playlist_head
        current = current.next

    return playlist_head
```

Example Usage:

```
playlist = SongNode("SOS", "ABBA",
                SongNode("Simple Twist of Fate", "Bob Dylan",
                    SongNode("Dreams", "Fleetwood Mac",
                        SongNode("Lovely Day", "Bill Withers"))))

print_linked_list(remove_song(playlist, "Dreams"))
```

Expected Output:

```
('SOS', 'ABBA') -> ('Simple Twist of Fate', 'Bob Dylan') -> ('Lovely Day', 'Bill Wit
```

Problem 4: On Repeat

A variation of the two-pointer technique introduced in previous units is to have a slow and a fast pointer that increment at different rates.

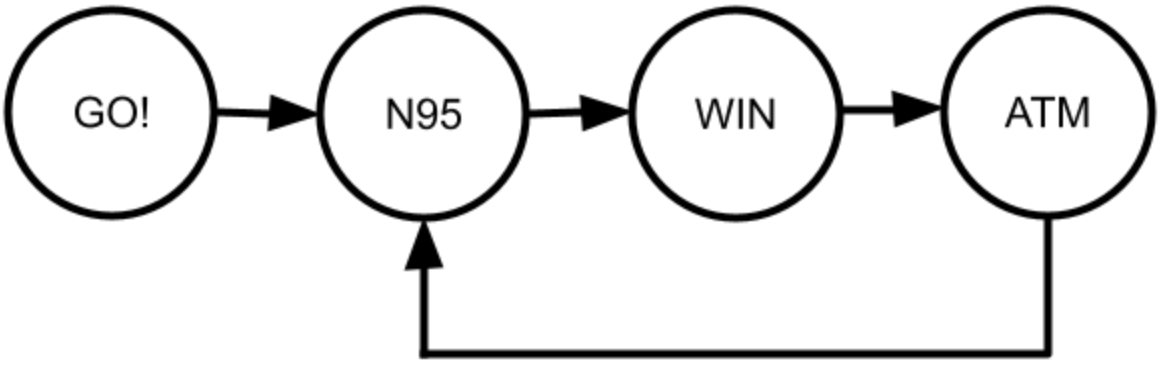
We would like to check whether our playlist loops or not. Given the head of a linked list `playlist_head`, return `True` if the playlist has a cycle in it and `False` otherwise. A linked list has a cycle if at some point in the list, the node's next pointer points back to a previous node in the list.

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 SongNode:
        def __init__(self, song, artist, next=None):
                self.song = song
        self.artist = artist
                self.next = next

def on_repeat(playlist_head):
        pass
```

Example Usage:



```
song1 = SongNode("GO!", "Common")
song2 = SongNode("N95", "Kendrick Lamar")
song3 = SongNode("WIN", "Jay Rock")
song4 = SongNode("ATM", "J. Cole")
song1.next = song2
song2.next = song3
song3.next = song4
song4.next = song2

print(on_repeat(song1))
```

Example Output:

```
True
```

▶ 💡 **Hint: Slow and Fast Pointers**

Problem 5: Looped

Given the head of a linked list `playlist_head` that may contain a cycle, use the fast and slow pointer method to return the length of the cycle. If the list does not contain a cycle, return `0`.

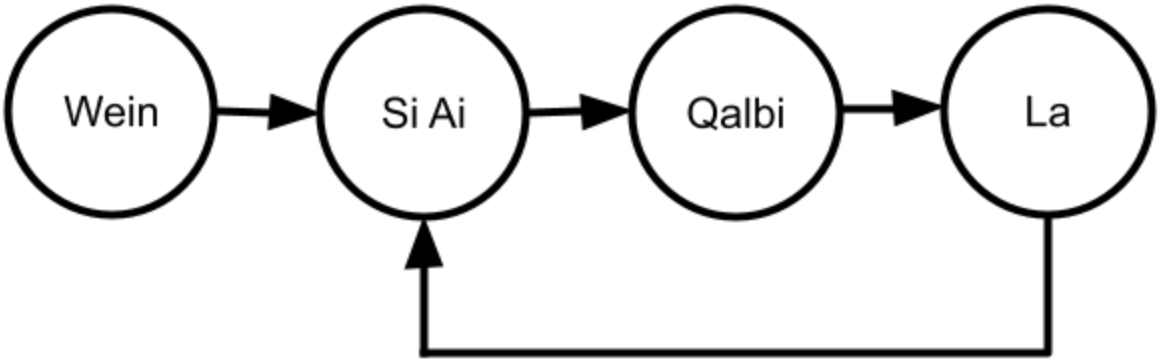
Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
class SongNode:
        def __init__(self, song, artist, next=None):
                self.song = song
        self.artist = artist
                self.next = next

# For testing
def print_linked_list(node):
    current = node
    while current:
        print((current.song, current.artist), end=" -> " if current.next else "")
        current = current.next
    print()

def loop_length(playlist_head):
        pass
```

Example Usage:




```
song1 = SongNode("Wein", "AL SHAMI")
song2 = SongNode("Si Ai", "Tayna")
song3 = SongNode("Qalbi", "Yasser Abd Alwahab")
song4 = SongNode("La", "DYSTINCT")
song1.next = song2
song2.next = song3
song3.next = song4
song4.next = song2

print(loop_length(song1))
```

Example Output:

```
3
```

▶ 💡 **Hint: Multiple Pass Technique**

Problem 6: Volume Control

You are working as an engineer normalizing volume levels on songs. Given the head of a singly linked list with integer values `song_audio` representing volume levels at different points in a song, return the number of critical points. A critical point is a local minima or maxima.

- The head and tail nodes are not considered critical points.
- A node is a local minima if both the next and previous elements are greater than the current element
- A node is a local maxima if both the next and previous elements are less than the current element

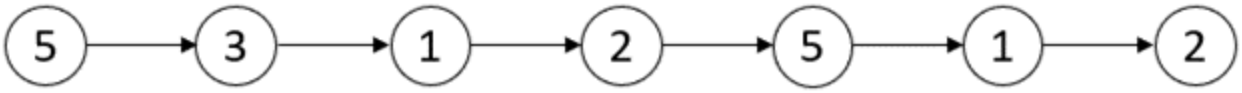
Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
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 count_critical_points(song_audio):
        pass
```

Example Usage:




```
song_audio = Node(5, Node(3, Node(1, Node(2, Node(5, Node(1, Node(2)))))))

print(count_critical_points(song_audio))
```

Example Output:

```
3
Explanation: There are three critical points:
- The third node is a local minima because 1 is less than 3 and 2.
- The fifth node is a local maxima because 5 is greater than 2 and 1.
- The sixth node is a local minima because 1 is less than 5 and 2.
```

▶ 💡 **Hint: Which technique?**

Close Section

▼ Standard Problem Set Version 2

Problem 1: Why is it Always You Three

In a single assignment statement, create the linked list `Harry -> Ron -> Hermione`.

```
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

# Add your assignment statement here
```

Example Usage:

```
print_linked_list(head)
```

Expected Output:

```
Harry -> Ron -> Hermione
```

▶ 💡 **Hint: Nested Constructors**

Problem 2: 200 Points for Gryffindor

It's almost the end of the year, and Gryffindor students want to see if they have any competition for first place. Given the head of a linked list `house_points` and the Gryffindor's `score`, return the frequency of `score` in the 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, house, score, next=None):
        self.house = house
                self.value = score
                self.next = next

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

def count_element(house_points, score):
        pass
```

Example Usage:

```
house_points = Node("Gryffindor", 600,
                Node("Ravenclaw", 300,
                    Node("Slytherin", 500,
                        Node("Hufflepuff", 600))))

print(count_element(house_points, score))
```

Example Output:

```
2
```

Problem 3: Target Practice

You are practicing the accuracy of your spellwork by trying to extract the middle-most ingredient in a line of potions. Given the head of a linked list, `potions`, use a variation of the two-pointer technique to return the middle `potion`. If there are two middle nodes, return the `potion` of the second middle node.

The two-pointer variation you should use is called the 'slow and fast pointer' or 'tortoise and the hare' technique. In this variation, a slow and a fast pointer are incremented at different rates.

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, potion, next=None):
        self.potion = potion
        self.next = next

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

def find_middle_potion(potions):
        pass
```

Example Usage:

```
potions1 = Node("Poison Antidote", Node("Shrinking Solution", Node("Trollblood Tinct
potions2 = Node("Elixir of Life", Node("Sleeping Draught", Node("Babbling Beverage",

print(find_middle_potion(potions1))
print(find_middle_potion(potions2))
```

Example Output:

```
Shrinking Solution
Sleeping Draught
```

▶ 💡 **Hint: Slow and Fast Pointers**

Problem 4: Turn Back Time

A spell gone wrong has reversed time! Write a function `reverse()` that accepts the head of a singly linked list `events` and restores order by reversing the order of elements. Return the head of the reversed 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.

```python
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 reverse(events):
    pass
```

Example Usage:

```python
events = Node("Potion Brewing",
            Node("Spell Casting",
                Node("Wand Making",
                    Node("Dragon Taming",
                        Node("Broomstick Flying")))))

print_linked_list(reverse(events))
```

Example Output:

```
Broomstick Flying -> Dragon Taming -> Wand Making -> Spell Casting -> Potion Brewing
```

Problem 5: Mirror, Mirror

You think another bit of wonky spell casting may have left your enchanted mirror broken. Write a function `is_mirrored()` to test if your mirror successfully reflects objects back. The function accepts the `head` of a linked list and should return `True` if the values of the linked list read the same backwards and forwards, and `False` otherwise.

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 is_mirrored(head):
        pass
```

Example Usage:

```
list1 = Node("Phoenix", Node("Dragon", Node("Phoenix")))
list2 = Node("Werewolf", Node("Vampire", Node("Griffin")))

print(is_mirrored(list1))
print(is_mirrored(list2))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Multiple Pass Technique**

▶ 💡 **Hint: Pseudocode**

Problem 6: Magic Loop

In a nearby enchanted forest, magical paths sometimes loop back on themselves, creating never-ending cycles. Write a function `loop_start()` to help you keep your way. The function accepts the head of a linked list `path_start` and returns the `value` of the node where the cycle starts.

If the path has no cycle, return `None`.

A linked list has a cycle if, at some point in the list, the node's next pointer points back to a previous node in the list.

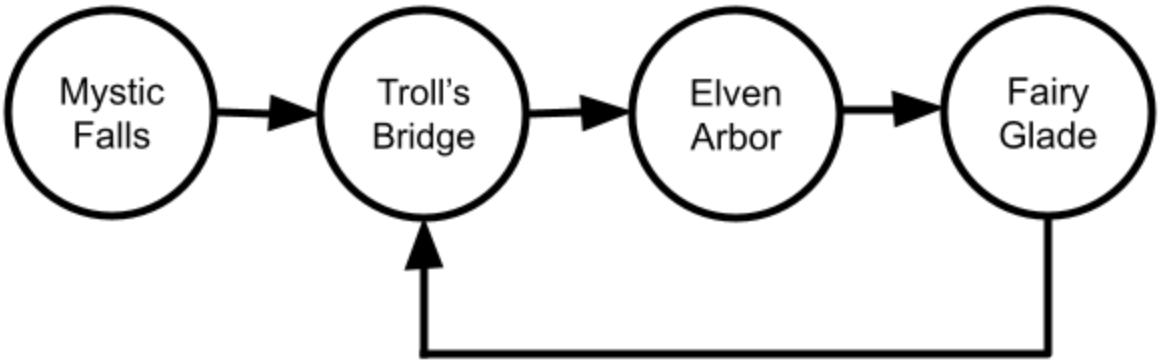
Evaluate the time and space complexity of your solution. Define your variables and provide a rationale for why you believe your solution has the stated time and space complexity.

```
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 loop_start(path_start):
    pass
```

Example Usage:




```
path_start = Node("Mystic Falls")
waypoint1 = Node("Troll's Bridge")
waypoint2 = Node("Elven Arbor")
waypoint3 = Node("Fairy Glade")

path_start.next = waypoint1
waypoint1.next = waypoint2
waypoint2.next = waypoint3
waypoint3.next = waypoint1

print(loop_start(path_start))
```

Example Output:

```
Troll's Bridge
```

- **Advanced Problem Set Version 1**
- **Advanced Problem Set Version 2**