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

Session Overview

In this session, students will delve into the intricate world of linked lists, focusing on both singly and doubly linked list structures. They will engage in a series of hands-on problems that challenge them to perform operations such as insertion, deletion, copying, and value manipulation within linked lists.

> 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**
▶ **Standard Problem Set Version 2**
▼ **Advanced Problem Set Version 1**

Problem 1: Greatest Node

Write a function `find_max()` that takes in the `head` of a linked list and returns the maximum value in the linked list. You can assume the linked list will contain only numeric values.

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 find_max(head):
    pass
```

Example Usage:

```python
head1 = Node(5, Node(6, Node(7, Node(8))))

# Linked List: 5 -> 6 -> 7 -> 8
print(find_max(head1))

head2 = Node(5, Node(8, Node(6, Node(7))))

# Linked List: 5 -> 8 -> 6 -> 7
print(find_max(head2))
```

Expected Output:

```
8
8
```

▶ 💡 **Hint: Linked List Traversal**

Problem 2: Remove Tail

The following code incorrectly implements the function `remove_tail()`. When correctly implemented, `remove_tail()` accepts the `head` of a singly linked list and removes the last node (the tail) in the list. The function should return the `head` of the modified list.

Step 1: Copy this code into Replit.

Step 2: Create your own test cases to run the code against. Use print statements, `print_linked_list()`, and the stack trace to identify and fix any bugs so that the function correctly removes the last node from the list.

```python
class Node:
    def __init__(self, value=None, 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_tail(head):
    if head is None:
        return None
    if head.next is None:
        return None

    current = head
    while current.next:
        current = current.next

    current.next = None
    return head
```

Example Usage:

```python
head = Node("Isabelle", Node("Alfonso", Node("Cyd")))

# Linked List: Isabelle -> Alfonso -> Cyd
print_linked_list(remove_tail(head))
```

Expected Output:

```
Isabelle -> Alfonso
```

Problem 3: Delete Duplicates in a Linked List

Given the `head` of a sorted linked list, delete all elements that occur more than once in the list (*not just the duplicates*). The resulting list should maintain sorted order. Return the head of the linked 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 delete_dupes(head):
    pass
```

Example Usage:

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

# Linked List: 1 -> 2 -> 3 -> 3 -> 4 -> 5
print_linked_list(delete_dupes(head))
```

Example Output:

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

▶ 💡 **Hint: Temporary Head Technique**

Problem 4: Does it Cycle?

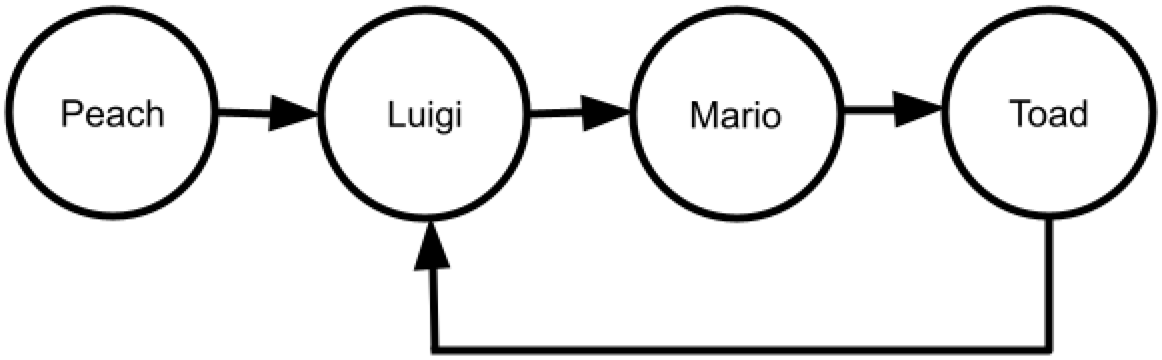
A variation of the two-pointer technique introduced earlier in the course is to have a slow and a fast pointer that increment at different rates. Given the `head` of a linked list, use the slow-fast pointer technique to write a function `has_cycle()` that returns `True` if the list has a cycle in it 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 Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

def has_cycle(head):
    pass
```

Example Usage:



```
peach = Node("Peach", Node("Luigi", Node("Mario", Node("Toad"))))

print(has_cycle(peach))
```

Example Output:

```
True
```

▶ 💡 **Hint: Slow and Fast Pointers**

Problem 5: Remove Nth Node From End of List

Given the `head` of a linked list and an integer `n`, write a function `remove_nth_from_end()` that removes the `nth` node from the end of the list. The function should return the head of the modified 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 remove_nth_from_end(head, n):
    pass
```

Example Usage:

```
head1 = Node("apple", Node("cherry", Node("orange", Node("peach", Node("pear")))))
head2 = Node("Rainbow Trout", Node("Ray"))
head3 = Node("Rainbow Stag")


print_linked_list(remove_nth_from_end(head1, 2))
print_linked_list(remove_nth_from_end(head2, 1))
print_linked_list(remove_nth_from_end(head3, 1))
```

Example Output:

```
apple -> cherry -> orange -> pear
Rainbow Trout

Example 3 Explanation: The last example returns an empty list.
```

Problem 6: Careful Reverse

Given the `head` of a singly linked list and an integer `k`, reverse the first k elements of the linked list. Return the new head of the linked list. If `k` is larger than the length of the list, reverse the entire 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 reverse_first_k(head, k):
        pass
```

Example Usage:

```
head = Node("apple", Node("cherry", Node("orange", Node("peach", Node("pear")))))

print_linked_list(reverse_first_k(head, 3))
```

Example Output:

```
orange -> cherry -> apple -> peach -> pear
```

Close Section

▾ Advanced Problem Set Version 2

Problem 1: Array to Linked List

Write a function `arr_to_ll()` that accepts an *array* of `Player` instances `arr` and converts `arr` into a linked list. The function should return the head of the linked list. If `arr` is empty, return `None`.

A function `print_linked_list()` is provided, which accepts the **head**, or first element, of a linked list and prints the `character` attribute of each `Player` in the linked list for testing purposes.

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 Player:
    def __init__(self, character, kart):
        self.character = character
        self.kart = kart
        self.items = []

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.character, end=" -> " if current.next else "\n")
        current = current.next

def arr_to_ll(arr):
    pass
```

Example Usage:

```
mario = Player("Mario", "Mushmellow")
luigi = Player("Luigi", "Standard LG")
peach = Player("Peach", "Bumble V")

print_linked_list(arr_to_ll([mario, luigi, peach]))
print_linked_list(arr_to_ll([peach]))
```

Example Output:

```
Mario -> Luigi -> Peach
Peach
```

Problem 2: Get it Out of Here!

The following code incorrectly implements the function `remove_by_value()`. When implemented correctly, `remove_by_value()` accepts the `head` of a singly linked list and a value `val`, and removes the first node in the linked list with the value `val`. It should return the `head` of the modified list.

Step 1: Copy this code into Replit.

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

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

# Function with a bug!
def remove_by_value(head, val):
    if not head:
        return None
    if head.value == val:
        return head.next

    current = head
    while current.next:
        if current.next.value == val:
            current = current.next.next
            return head
        current = current.next

    return head
```

Example Usage:

```
head = Node("Daisy", Node("Mario", Node("Waluigi", Node("Baby Peach"))))

print_linked_list(remove_by_value(head, "Waluigi"))
```

Expected Output:

```
Daisy -> Mario -> Baby Peach
```

▶ 💡 **Hint: Linked List Traversal**

Problem 3: Partition List Around Value

Given the `head` of a linked list with integer values and a value `val`, write a function `partition()` that partitions the linked list around `val` such that all nodes with values less than `val` come before nodes with values greater than or equal to `val`.

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(head, val):
    pass
```

Example Usage:

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

print_linked_list(partition(head, 3))
```

Example Output:

```
1 -> 2 -> 2 -> 4 -> 3 -> 5
Explanation: There are multiple possible solutions.
E.g. 2 -> 2 -> 1 -> 5 -> 4 -> 3
```

▶ 💡 **Hint: Temporary Head Technique**

Problem 4: Middle Match

A variation of the two-pointer technique introduced earlier in the course is to have a slow and a fast pointer that increment at different rates. Given the `head` of a linked list, and a value `val`, use the slow-fast pointer technique to determine if `val` matches the middle node of the list. If there are two middle nodes, return `True` if the second middle node matches the value `val`.

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 middle_match(head, val):
    pass
```

Example Usage:

```python
kart_choices = Node("Bullet Bike", Node("Wild Wing", Node("Pirahna Prowler")))
tournament_tracks = Node("Rainbow Road", Node("Bowser Castle", Node("Sherbet Land", 

print(middle_match(kart_choices, "Wild Wing"))
print(middle_match(tournament_tracks, "Bowser Castle"))
```

Example Output:

```
True
False
```

▶ 💡 **Hint: Slow and Fast Pointers**

Problem 5: Put it in Reverse

Given the `head` of a singly linked list, reverse the list, and return the head of the reversed list. You must reverse the list in place. 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(head):
        pass
```

Example Usage:

```python
kart_choices = Node("Bullet Bike", Node("Wild Wing", Node("Pirahna Prowler")))

print_linked_list(reverse(kart_choices))
```

Example Output:

```
Pirahna Prowler -> Wild Wing -> Bullet Bike
```

Problem 6: Symmetrical

Given the head of a singly linked list, return `True` if the values of the linked list nodes read the same forwards and backwards. Otherwise, return `False`. Use the two-pointer technique in your solution.

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 is_symmetric(head):
        pass
```

Example Usage:

```
head1 = Node("Bitterling", Node("Crawfish", Node("Bitterling")))
head2 = Node("Bitterling", Node("Carp", Node("Koi")))

print(is_symmetric(head1))
print(is_symmetric(head2))
```

Example Output:

```
True
False
```

Close Section