TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Summer 2025 (@ Section 1b | Tuesdays and Thursdays 3PM - 5PM PDT)
Personal Member ID#: 126663

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**

Please Note: **It is not required or expected** that you complete all of the practice problems! In some sessions you may only complete 1 problem and that's okay.

Strengthening your **approach** to problems, and your **ability to speak and engage through the process** are key skills most often underdeveloped for engineers at this stage - focus on those in our small groups for your long term success!

You can always return to problems independently, after class time, to embrace the technical concepts and gain additional practice.

Close Section

🔎 Problem Solving Approach

We will approach problems using the six steps in the UMPIRE approach.

UMPIRE: Understand, Match, Plan, Implement, Review, Evaluate.

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

We will learn to:

- **Understand** the problem
- **Match** identifies common approaches you've seen/used before
- **Plan** a solution step-by-step, and
- **Implement** the solution
- **Review** your solution
- **Evaluate** your solution's time and space complexity and think critically about the advantages and disadvantages of your chosen approach.

Breakout Problems Session 2

▼ Standard Problem Set Version 1

Problem 1: Mutual Friends

In the `Villager` class below, each villager has a `friends` attribute, which is a list of other villagers they are friends with.

Write a method `get_mutuals()` that takes one parameter, a `Villager` instance `new_contact`, and returns a list with the `name` of all friends the current villager and `new_contact` have in common.

```
class Villager:
    def __init__(self, name, species, catchphrase):
        self.name = name
        self.species = species
        self.catchphrase = catchphrase
        self.friends = []

    def get_mutuals(self, new_contact):
        pass
```

Example Usage:

```
bob = Villager("Bob", "Cat", "pthhhpth")
marshal = Villager("Marshal", "Squirrel", "sulky")
ankha = Villager("Ankha", "Cat", "me meow")
fauna = Villager("Fauna", "Deer", "dearie")
raymond = Villager("Raymond", "Cat", "crisp")
stitches = Villager("Stitches", "Cub", "stuffin")

bob.friends = [stitches, raymond, fauna]
marshal.friends = [raymond, ankha, fauna]
print(bob.get_mutuals(marshal))

ankha.friends = [marshal]
print(bob.get_mutuals(ankha))
```

Example Output:

```
['Raymond', 'Fauna']
[]
```

▼ ✨ AI Hint: Class Methods

Key Skill: Use AI to explain code concepts

This question requires you to be familiar with class methods, which are functions attached to an object. To help, we've included more info Unit 5 Cheatsheet

If you'd still like to see more examples or ask follow-up questions, try using an AI tool like ChatGPT or GitHub Copilot. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please provide 2-3 examples of how Class Methods are used in Python, and explain how each one works."

You might also want to ask questions like:

"Can you explain the difference between class methods, instance methods, and functions?"

Problem 2: Linked Up

A **linked list** is a data structure that, similar to a normal list or array, allows us to store pieces of data sequentially. The key difference is how the elements are stored in memory.

In a normal list, elements are stored in adjacent memory locations. If we know the location of the first element, we can easily access any other element in the list.

In a linked list, individual elements, called **nodes**, are not stored in sequential memory locations. Instead, each node stores a reference or pointer to the next node in the list, allowing us to traverse the list.

Connect the provided node instances below to create the linked list `kk_slider -> harriet -> saharah -> isabelle`.

A function `print_linked_list()` which accepts the **head**, or first element, of a linked list and prints the values of the list has also been provided for testing purposes.

```
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

kk_slider = Node("K.K. Slider")
harriet = Node("Harriet")
saharah = Node("Saharah")
isabelle = Node("Isabelle")

# Add code here to link the above nodes
```

Example Usage:

```
print_linked_list(kk_slider)
```

Example Output:

```
K.K. Slider -> Harriet -> Saharah -> Isabelle
```

▼ ✨ AI Hint: Linked Lists

Key Skill: Use AI to explain code concepts

This question requires you to be familiar with Linked Lists, a incredibly useful but sometimes tricky data structure. To help, we've included a review of linked lists Unit 5 Cheatsheet

You can also use an AI tool like ChatGPT or GitHub Copilot to get more examples or ask follow-up questions. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Can you help me understand linked lists conceptually, using analogies to real-world objects?"

Once you understand the concept of Linked Lists, you can also ask follow-up questions like:

"Can you provide examples of how to implement a linked list in Python, and explain how each part works?"

"Here is a provided Linked List class: (CODE). Can you give me an example of how to access the data in this linked list?"

Problem 3: Daily Tasks

Imagine a linked list used as a daily task list where each node represents a task. Write a function `add_task()` that takes in the `head` of a linked list and adds a new node to the front of the task list.

The function should insert a new `Node` object with the value `task` as the new `head` of the linked list and return the new node.

Note: The "head" of a linked list is the first element in the linked list. It is equivalent to `lst[0]` of a normal list.

```
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_first(head, task):
    pass
```

Example Usage:

```
task_1 = Node("shake tree")
task_2 = Node("dig fossils")
task_3 = Node("catch bugs")
task_1.next = task_2
task_2.next = task_3

# Linked List: shake tree -> dig fossils -> catch bugs
print_linked_list(add_first(task_1, "check turnip prices"))
```

Example Output:

```
check turnip prices -> shake tree -> dig fossils -> catch bugs
```

Problem 4: Halve List

Write a function `halve_list()` that accepts the `head` of a linked list whose values are integers and divides each value by two. Return the `head` of the modified list.

```
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 halve_list(head):
    pass
```

Example Usage:

```
node_one = Node(5)
node_two = Node(6)
node_three = Node(7)
node_one.next = node_two
node_two.next = node_three

# Input List: 5 -> 6 -> 7
print_linked_list(halve_list(node_one))
```

Example Output:

```
2.5  -> 3 -> 3.5
```

▼ 💡 **Hint: Linked List Traversal**

This problem requires you to traverse a linked list. In other words, it requires you to iterate over the nodes of a linked list. For a break down of how to traverse a linked list, check out the unit cheatsheet.

Problem 5: Remove Last

Write a function `delete_tail()` that accepts the `head` of a linked list and removes the last node in the list. Return the `head` of the modified list.

Note: The "tail" of a list is the last element in the linked list. It is equivalent to `lst[-1]` in a normal list.

```
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_tail(head):
    pass
```

Example Usage:

```
butterfly = Node("Common Butterfly")
ladybug = Node("Ladybug")
beetle = Node("Scarab Beetle")
butterfly.next = ladybug
ladybug.next = beetle

# Input List: butterfly -> ladybug -> beetle
print_linked_list(delete_tail(butterfly))
```

Example Output:

```
Common Butterfly -> Ladybug
```

Problem 6: Find Minimum in Linked List

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

```
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_min(head):
    pass
```

Example Usage:

```
head1 = Node(5, Node(6, Node(7, Node(8))))
head2 = Node(8, Node(5, Node(6, Node(7))))

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

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

Expected Output:

```
5
5
```

▼ 💡 **Hint: Nested Constructors**

This problem requires you to be familiar with nesting constructors. The `Node` class below accepts two parameters:

- the value of the Node object.
- the next Node object in the linked list or `None` if the Node is not linked to another node.

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

In the past, we constructed each node in the list individually, then linked them together.

```
node_one = Node(1)
node_two = Node(2)
node_one.next = node_two
```

We can instead chain together our constructor calls, and pass in a second Node object `Node(2)` as the `next` argument for the first node.

```
head = Node(1, Node(2))
```

This technique is commonly used when generating test cases for linked lists.

Problem 7: Remove From Inventory

Imagine a linked list used to store a player's inventory. Write a function `delete_item()` that takes in the `head` of a linked list and a value `item` as parameters.

The function should remove the first node it finds in the linked list with the value `item` and return the `head` of the modified list. If no node can be found with the value `item`, return the list unchanged.

```
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_item(head, item):
    pass
```

Example Usage:

```
slingshot = Node("Slingshot")
peaches = Node("Peaches")
beetle = Node("Scarab Beetle")
slingshot.next = peaches
peaches.next = beetle

# Linked List: slingshot -> peaches -> beetle
print_linked_list(delete_item(slingshot, "Peaches"))

# Linked List: slingshot -> beetle
print_linked_list(delete_item(slingshot, "Triceratops Torso"))
```

Example Output:

```
Slingshot -> Scarab Beetle
Slingshot -> Scarab Beetle
```

Problem 8: Move Tail to Front of Linked List

Write a function `tail_to_head()` that takes in the `head` of a linked list as a parameter and moves the tail of the linked list to the front.

```
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 tail_to_head(head):
    pass
```

Example Usage:

```
daisy = Node("Daisy")
mario = Node("Mario")
toad = Node("Toad")
peach = Node("Peach")
daisy.next = mario
mario.next = toad
toad.next = peach

# Linked List: Daisy -> Mario -> Toad -> Peach
print_linked_list(tail_to_head(daisy))
```

Example Output:

```
Peach -> Daisy -> Mario -> Toad
```

Problem 9: Create Double Links

One of the drawbacks of a linked list is that it's difficult to go backwards because each `Node` only knows about the `Node` in front of it. (E.g., `A -> B -> C`)

A **doubly linked list** solves this problem! Instead of just having a `next` attribute, a doubly linked list also has a `prev` attribute that points to the `Node` before it. (E.g., `A <-> B <-> C`)

Update the `Node` constructor below so that the code creates a doubly linked list with `head <-> tail`.

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

head = Node("Isabelle")
tail = Node("K.K. Slider")

head.next = tail
tail.prev = head
```

Example Usage:

```
print(head.value, "<->", head.next.value)
print(tail.prev.value, "<->", tail.value)
```

Example Output:

```
Isabelle <-> K.K. Slider
Isabelle <-> K.K. Slider
```

Problem 10: Print Backwards

Write a function `print_reverse()` that takes in the `tail` of a doubly linked list as a parameter.

It should print out the values of the linked list in reverse order, each separated by a space.

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

def print_reverse(tail):
    pass
```

Example Usage:

```
isabelle = Node("Isabelle")
kk_slider = Node("K.K. Slider")
saharah = Node("Saharah")
isabelle.next = kk_slider
kk_slider.next = saharah
saharah.prev = kk_slider
kk_slider.prev = isabelle

# Linked List: Isabelle <-> K.K. Slider <-> Saharah
print_reverse(saharah)
```

Example Output:

```
Saharah K.K. Slider Isabelle
```

Close Section

▾ Standard Problem Set Version 2

Problem 1: Calculate Tournament Placement

In the `Player` class below, each player has a `race_outcomes` attribute which holds a list of integers describing what place they came in for each race in a tournament.

Write a method `get_tournament_place()` that takes in one parameter, `opponents`, a list of other player objects also participating in the tournament, and returns the place in the overall tournament.

- Rank in the tournament is determined by the **lowest** average race outcome. (1st place is better than 2nd!)
- Each opponent in `opponents` is guaranteed to have participated in the same number of races as the current player.

```
class Player:
    def __init__(self, character, kart, outcomes):
        self.character = character
        self.kart = kart
        self.items = []
        self.race_outcomes = outcomes

    def get_tournament_place(self, opponents):
        pass
```

Example Usage:

```
player1 = Player("Mario", "Standard", [1, 2, 1, 1, 3])
player2 = Player("Luigi", "Standard", [2, 1, 3, 2, 2])
player3 = Player("Peach", "Standard", [3, 3, 2, 3, 1])

opponents = [player2, player3]
print(player1.get_tournament_place(opponents))
```

Example Output:

```
1
Explanation: Mario/player1's average place is 1.6, Luigi's is 2.0, and Peach's is 2.4
```

▼ ✨ AI Hint: Class Methods

Key Skill: Use AI to explain code concepts

This question requires you to be familiar with class methods, which are functions attached to an object. To help, we've included more info Unit 5 Cheatsheet

If you'd still like to see more examples or ask follow-up questions, try using an AI tool like ChatGPT or GitHub Copilot. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Please provide 2-3 examples of how Class Methods are used in Python, and explain how each one works."

You might also want to ask questions like:

"Can you explain the difference between class methods, instance methods, and functions?"

Problem 2: Update Linked List Sequence

A linked list is a data structure that allows us to store pieces of data sequentially, similar to a normal list or array. The key difference between a linked list and a normal list is how each element is stored in a computer's memory.

In a normal list, individual elements are stored in adjacent memory locations according to their order in the list. If we know where the first element is stored, it's easy to access any other element in the list.

In a linked list, individual elements, called **nodes**, are not stored in sequential memory locations. Each node may be stored in an unrelated memory location. To connect nodes into a sequential list, each node stores a reference or **pointer** to the next node in the list.

Using the provided `Node` class and the linked list below, update the current linked list `shy_guy -> diddy_kong -> dry_bones` to `shy_guy -> link -> diddy_kong -> toad -> dry_bones`.

A function `print_linked_list()` that accepts the **head**, or first element, of a linked list and prints the values of the list has also been provided for testing purposes.

```
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

shy_guy = Node("Shy Guy")
diddy_kong = Node("Diddy Kong")
dry_bones = Node("Dry Bones")
shy_guy.next = diddy_kong
diddy_kong.next = dry_bones

# Add code to update the list here
```

Example Usage:

```
print("Current List:")
print_linked_list(shy_guy)
```

Example Output:

```
Current List:
shy_guy -> diddy_kong -> dry_bones
```

▼ ✨ AI Hint: Linked Lists

Key Skill: Use AI to explain code concepts

This question requires you to be familiar with Linked Lists, a incredibly useful but sometimes tricky data structure. To help, we've included a review of linked lists Unit 5 Cheatsheet

You can also use an AI tool like ChatGPT or GitHub Copilot to get more examples or ask follow-up questions. You can use the following prompt as a starting point:

"You're an expert computer science tutor. Can you help me understand linked lists conceptually, using analogies to real-world objects?"

Once you understand the concept of Linked Lists, you can also ask follow-up questions like:

"Can you provide examples of how to implement a linked list in Python, and explain how each part works?"

"Here is a provided Linked List class: (CODE). Can you give me an example of how to access the data in this linked list?"

Problem 3: Insert Node as Second Element

Write a function `add_second()` that takes in the `head` of a linked list and a value `val` as parameters. It should insert `val` as the second node in the linked list and return the **head** of the linked list. (You can assume `head` is not `None`.)

Note: The "head" of a linked list is the first element in the linked list. It is equivalent to `lst[0]` of a normal list.

```
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_second(head, val):
    pass
```

Example Usage:

```
original_list_head = Node("banana")
second = Node("blue shell")
third = Node("bullet bill")
original_list_head.next = second
second.next = third


# Linked list: "banana" -> "blue shell" -> "bullet bill"
new_list = add_second(head, "red shell")
print_linked_list(new_list)
```

Example Output:

```
banana -> red shell -> blue shell -> bullet bill
```

Problem 4: Increment Linked List Node Values

Write a function `increment_ll()` that takes in the `head` of a linked list of integer values and returns the same list, but with each node's value incremented by 1. Return the `head` of the list.

```
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 increment_ll(head):
    pass
```

Example Usage:

```
node_one = Node(5)
node_two = Node(6)
node_three = Node(7)
node_one.next = node_two
node_two.next = node_three

# Input List: 5 -> 6 -> 7
print_linked_list(increment_ll(node_one))
```

Example Output:

```
6 -> 7 -> 8
```

▼ 💡 **Hint: Linked List Traversal**

This problem requires you to traverse a linked list. In other words, it requires you to iterate over the nodes of a linked list. For a break down of how to traverse a linked list, check out the unit cheatsheet.

Problem 5: Copy Linked List

Write a function `copy_ll()` that takes in the `head` of a linked list and creates a complete **copy** of that linked list.

The function should return the `head` of a new linked list which is identical to the given list in terms of its structure and contents, but does not use any of the node objects from the original list.

```
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 copy_ll(head):
    pass
```

Example Usage:

```
mario = Node("Mario")
daisy = Node("Daisy")
luigi = Node("Luigi")
mario.next = daisy
daisy.next = luigi

# Linked List: Mario -> Daisy -> Luigi
copy = copy_ll(mario)

# Change original list -- should not affect the copy
mario.value = "Original Mario"

print_linked_list(head)
print_linked_list(copy)
```

Example Output:

```
Original Mario -> Daisy -> Luigi
Mario -> Daisy -> Luigi
```

Problem 6: Making the Cut

Imagine that a linked list is used to track the order players finished in a race. Write a function `top_n_finishers()` that takes in the `head` of a linked list and a non-negative integer `n` as parameters.

The function should return a list of the values of the first `n` nodes.

- If `n` is greater than the length of the linked list, return a list of the values of all nodes in the linked list.

```
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 top_n_finishers(head, n):
    pass
```

Example Usage:

```
head = Node("Daisy", Node("Mario", Node("Toad", Node("Yoshi"))))

# Linked List: Daisy -> Mario -> Toad -> Yoshi
print(top_n_finishers(head, 3))

# Linked List: Daisy -> Mario -> Toad -> Yoshi
print(top_n_finishers(head, 5))
```

Example Output:

```
["Daisy", "Mario", "Toad"]
["Daisy", "Mario", "Toad", "Yoshi"]
```

▼ 💡 Hint: Nested Constructors

This problem requires you to be familiar with nesting constructors. The `Node` class below accepts two parameters:

- the value of the Node object.
- the next Node object in the linked list or `None` if the Node is not linked to another node.

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

In the past, we constructed each node in the list individually, then linked them together.

```
node_one = Node(1)
node_two = Node(2)
node_one.next = node_two
```

We can instead chain together our constructor calls, and pass in a second Node object `Node(2)` as the `next` argument for the first node.

```
head = Node(1, Node(2))
```

This technique is commonly used when generating test cases for linked lists.

Problem 7: Remove Racer

Write a function `remove_racer()` that takes in the `head` of a linked list and a value `racer` as parameters.

The function should remove the first node with the value `racer` from the linked list and return the `head` of the modified list. If `racer` is not in the list, return the `head` of the original list.

```
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_racer(head, racer):
    pass
```

Example Usage:

```
head = Node("Daisy", Node("Mario", Node("Toad", Node("Mario"))))

# Linked List: Daisy -> Mario -> Toad -> Mario
print_linked_list(remove_racer(head, "Mario"))

# Linked List: Daisy -> Mario -> Toad
print_linked_list(remove_racer(head, "Yoshi"))
```

Example Output:

```
Daisy -> Mario -> Toad
Daisy -> Mario -> Toad
```

Problem 8: 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()` which accepts the **head**, or first element, of a linked list and prints the `character` attribute of each `Player` in the linked list has also been provided for testing purposes.

```
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 9: Convert Singly Linked List to Doubly Linked List

One of the drawbacks of a linked list is that it's difficult to go backwards, because each `Node` only knows about the `Node` in front of it. (E.g., `A -> B -> C`)

A **doubly linked list** solves this problem! Instead of just having a `next` attribute, a doubly linked list also has a `prev` attribute that points to the `Node` before it. (E.g., `A <-> B <-> C`)

Update the code below to convert the singly linked list to a doubly linked list.

Two functions, `print_linked_list()` and `print_linked_list_backwards()`, have been provided for testing purposes. `print_linked_list()` accepts the `head` of a linked list and prints the values of each node in the list, starting at the `head` and iterating in a forward direction. `print_linked_list_backwards()` accepts the `tail` of a linked list and prints the values of each node in the list, starting at the `tail` and iterating in a backward direction.

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

koopa_troopa = Node("Koopa Troopa")
toadette = Node("Toadette")
waluigi = Node("Waluigi")
koopa_troopa.next = toadette
toadette.next = waluigi

# Add code to convert to doubly linked list here
```

Example Usage:

```
print_linked_list(koopa_troopa)
print_linked_list_backwards(waluigi)
```

Example Output:

```
Koopa Troopa -> Toadette -> Waluigi
Waluigi -> Toadette -> Koopa Troopa
```

Problem 10: Find Length of Doubly Linked List from Any Node

Write a function `get_length()` that takes in a `node` at an unknown position within a doubly linked list. The function should return the length of the entire list.

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

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

def get_length(node):
    pass
```

Example Usage:

```
yoshi_falls = Node("Yoshi Falls")
moo_moo_farm = Node("Moo Moo Farm")
rainbow_road = Node("Rainbow Road")
dk_mountain = Node("DK Mountain")
yoshi_falls.next = moo_moo_farm
moo_moo_farm.next = rainbow_road
rainbow_road.next = dk_mountain
dk_mountain.prev = rainbow_road
rainbow_road.prev = moo_moo_farm
moo_moo_farm.prev = yoshi_falls

# List: Yoshi Falls <-> Moo Moo Farm <-> Rainbow Road <-> DK Mountain
print(get_length(rainbow_road))
```

Example Output:

```
4
```

Close Section

▾ 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.

```
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:

```
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**

This problem requires you to traverse a linked list. In other words, it requires you to iterate over the nodes of a linked list. For a break down of how to traverse a linked list, check out the unit cheatsheet.

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.

```
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:

```
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**

This problem may benefit from the temporary head technique. For reference, check out the unit cheatsheet.

Problem 4: Does it Cycle?

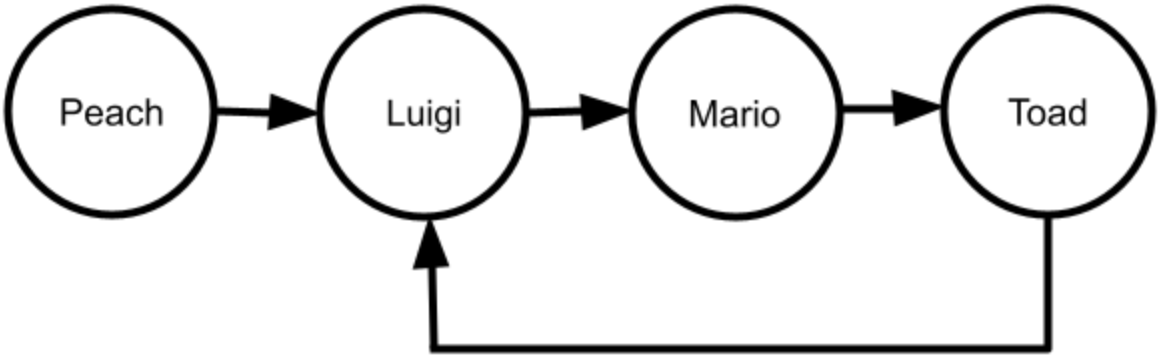
A variation of the two-pointer technique introduced earlier in the course is to have a slow and a fast pointer that increment at different rates. Given the `head` of a linked list, use the slow-fast pointer technique to write a function `has_cycle()` that returns `True` if the list has a cycle in it 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"))))

# Toad.next = Luigi
peach.next.next.next = peach.next

print(has_cycle(peach))
```

Example Output:

```
True
```

▼ ✨ AI Hint: Slow and Fast Pointers

Key Skill: Use AI to explain code concepts

This problem requires a variation of the two-pointer technique called the slow and fast pointer technique! For reference, check out the Unit 5 Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the slow and fast pointer technique.

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**

This problem requires you to traverse a linked list. In other words, it requires you to iterate over the nodes of a linked list. For a break down of how to traverse a linked list, check out the unit cheatsheet.

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**

This problem may benefit from the temporary head technique. For reference, check out the unit cheatsheet.

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.

```
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:

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

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

Example Output:

```
True
False
```

▼ ✨ AI Hint: Slow and Fast Pointers

Key Skill: Use AI to explain code concepts

This problem requires a variation of the two-pointer technique called the slow and fast pointer technique! For reference, check out the Unit 5 Cheatsheet.

For more help, you can use an AI tool like ChatGPT or GitHub Copilot to show you examples of the slow and fast pointer technique.

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.

```
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:

```
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.

```
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