



Welcome to this session: Coding Interview Workshop - Linked Lists

The session will start shortly...

Questions? Drop them in the chat.
We'll have dedicated moderators
answering questions.



Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:



Ian Wyles
Designated Safeguarding
Lead



Simone Botes



Rafiq Manan



Charlotte Witcher



Nurhaan Snyman



Ronald Munodawafa



Tevin Pitts

Scan to report a
safeguarding concern



or email the Designated
Safeguarding Lead:
Ian Wyles

safeguarding@hyperiondev.com

Skills Bootcamp Coding Interview Workshop

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Skills Bootcamp Coding Interview Workshop

- For all **non-academic questions**, please submit a query:
www.hyperiondev.com/support
- **Report a safeguarding incident:** **www.hyperiondev.com/safeguardreporting**
- We would love your feedback on lectures: **[Feedback on Lectures](#)**
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Learning Outcomes

- ❖ **Identify and describe the components of a node** in linked lists, including data and next/previous pointers.
- ❖ **Compare and contrast single and double linked lists**, highlighting their structures, use cases, and the pros and cons of each.
- ❖ **Implement basic operations** on single and double linked lists in Python and JavaScript, such as insertion, deletion, searching, and traversal.
- ❖ **Analyse and explain the time complexity of operations** on linked lists to evaluate performance implications in applications.
- ❖ **Demonstrate how to use linked lists to solve various problems** and compare the performance on the algorithms to ones that use simple arrays.

What is the time complexity of inserting a node at the beginning of a singly linked list?

- A. $O(1)$
- B. $O(n)$
- C. $O(\log n)$
- D. $O(n^2)$

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

Answer: A

- ❖ Inserting a node at the beginning of a singly linked list has a time complexity of $O(1)$ because it only requires updating the head pointer and the next pointer of the new node, regardless of the size of the linked list.



What is the main advantage of using a doubly linked list over a singly linked list?

- A. Faster search operations
- B. More efficient memory usage
- C. Ability to traverse both forward and backward
- D. Simpler implementation

Answer: C

- ❖ The main advantage of using a doubly linked list over a singly linked list is the ability to traverse both forward and backward. The previous pointer in each node allows for efficient backward traversal, which is not possible in a singly linked list without additional space complexity.

Lecture Overview

- Linked Lists
- Singly Linked Lists
- Doubly Linked Lists

Linked Lists

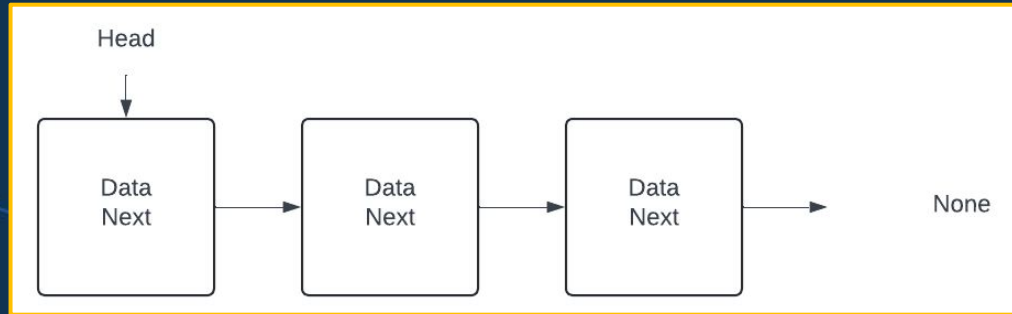
A linear data structure consisting of a sequence of nodes, each node contains data and a reference (link) to the next node in the sequence

- ❖ Components of a node:
 - **Data:** The actual information stored in the node
 - **Next pointer:** A reference to the next node in the linked list
 - **Previous pointer (for doubly linked lists):** A reference to the previous node in the linked list
- ❖ Nodes are **connected to form a linked list**, allowing for efficient insertion and deletion of elements at any position in the sequence

Singly Linked Lists

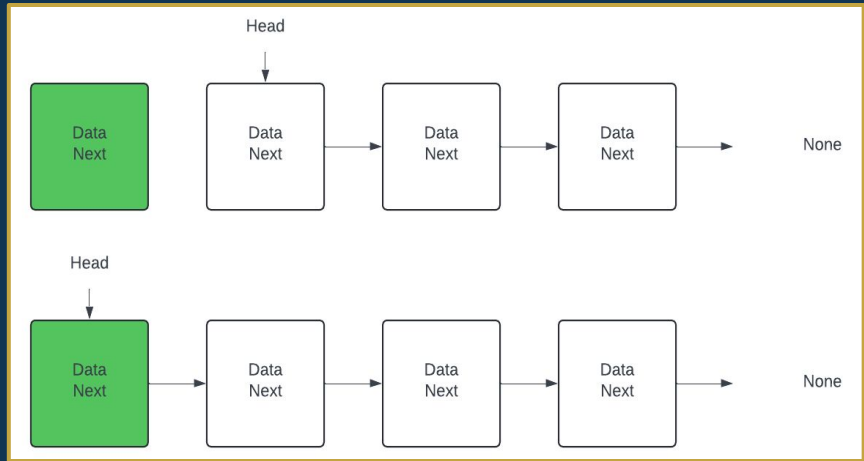
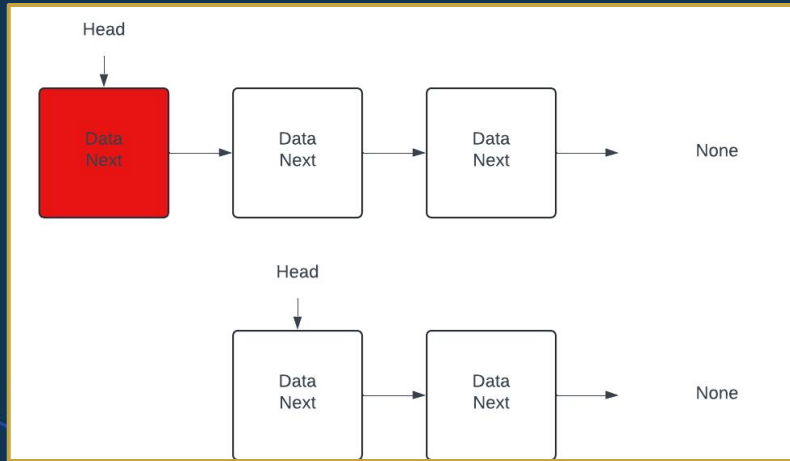
❖ Pros:

- **Dynamic size:** Linked lists can grow or shrink as needed during runtime
- **Efficient insertion and deletion:** $O(1)$ time complexity for inserting or deleting elements at the beginning of the list



Singly Linked Lists

❖ Why $O(1)$? Glad you asked, so 🤓...



❖ For removal we only take **1 step no matter the size of the list**, and the same for insertion, thus $O(1)$ for worst case

Singly Linked Lists

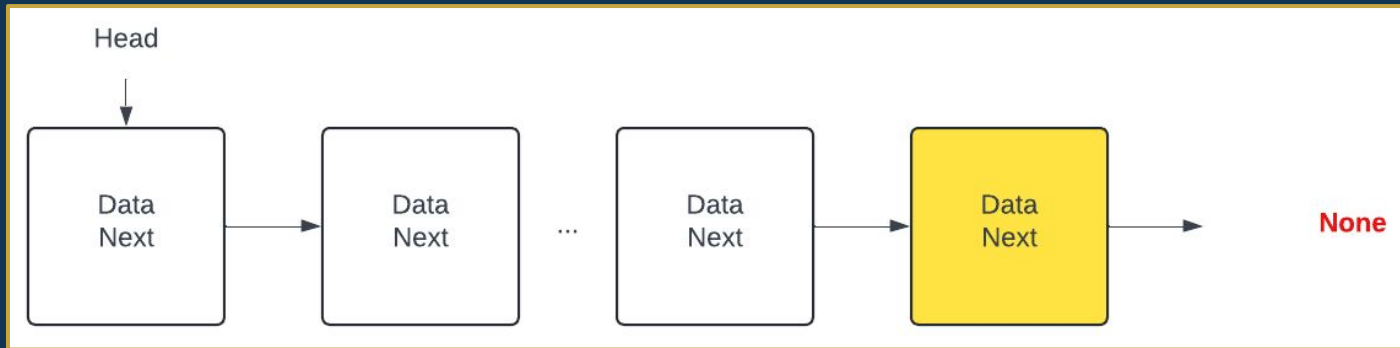


Cons:

- **No random access:** Accessing an element at a specific index requires traversing the list from the beginning, resulting in $O(n)$ time complexity
- **Extra memory:** Linked lists require additional memory for storing the next pointers

Singly Linked Lists

- ❖ Why $O(n)$? You flatter me 😊...



- ❖ This list has n nodes, to get to the yellow (last) one we need to pass n nodes, so in worst case $O(n)$

Implementation

Node Class

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

Constructor

```
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None
```

Implementation

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

As mentioned at the start, the complexity is $O(1)$ to insert at the beginning

```
def insert_at_end(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        return  
    last = self.head  
    while last.next:  
        last = last.next  
    last.next = new_node
```

$O(n)$, to loop through all n elements to insert the new node

Implementation

```
def insert_after(self, prev_node, data):  
    if prev_node is None:  
        print("The given previous node must be in linked list")  
        return  
    new_node = Node(data)  
    new_node.next = prev_node.next  
    prev_node.next = new_node
```

The complexity is $O(n)$, since in the worst case we insert at the end (if we insert after the last element)

Implementation

```
def delete_at_beginning(self):  
    if self.head is None:  
        return  
    self.head = self.head.next
```

$O(1)$ to delete at the beginning

```
def delete_at_end(self):  
    if self.head is None:  
        return  
    if self.head.next is None:  
        self.head = None  
        return  
    second_last = self.head  
    while second_last.next.next:  
        second_last = second_last.next  
    second_last.next = None
```

$O(n)$ since you have to loop through all n nodes

Implementation

```
def delete_node(self, key):
    temp = self.head
    if temp is not None:
        if temp.value == key:
            self.head = temp.next
            temp = None
            return
    while temp is not None:
        if temp.value == key:
            break
        prev = temp
        temp = temp.next
    if temp == None:
        return
    prev.next = temp.next
    temp = None
```

The time complexity of deleting a specified is $O(n)$ since in the worst case we delete at the end



Implementation

```
def search(self, key):  
    current = self.head  
    while current:  
        if current.value == key:  
            return True  
        current = current.next  
    return False
```

The time complexity to search is $O(n)$ since in the worst case we need to search for the last element

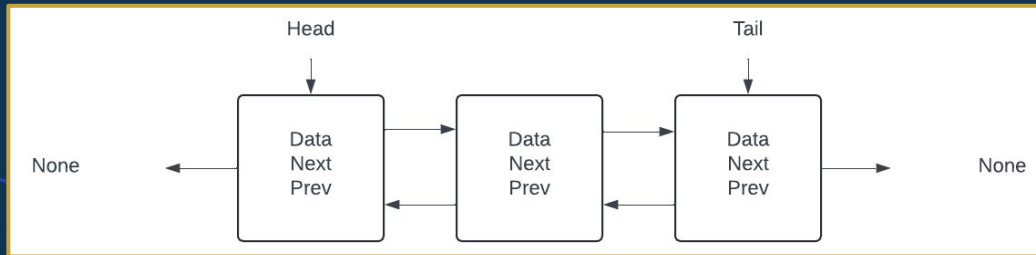
Let's Breathe!

Let's take a small break
before moving on to
the next topic.



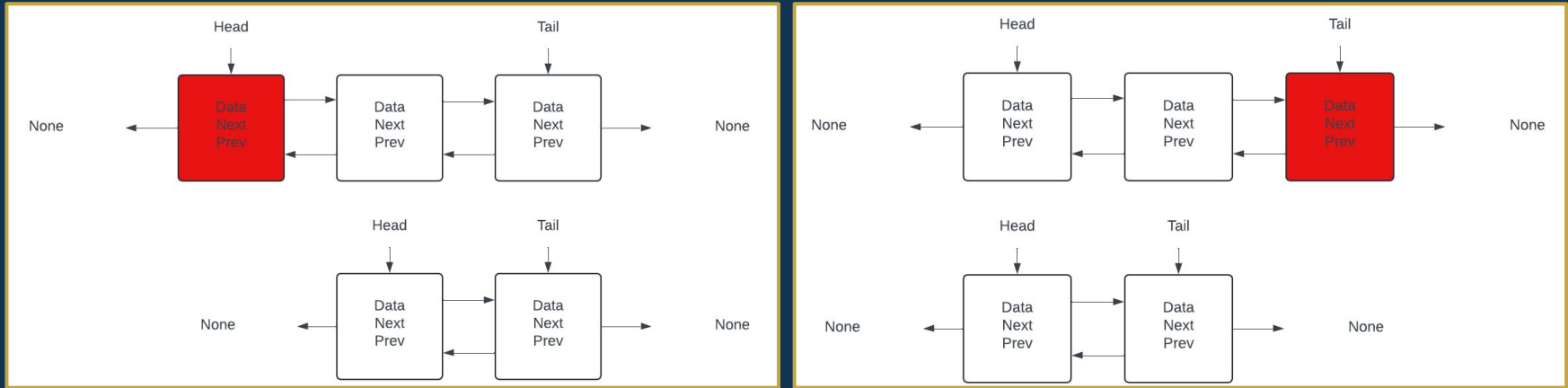
Doubly Linked Lists

- ❖ Pros:
 - **Efficient insertion and deletion:** $O(1)$ time complexity for inserting or deleting elements at both ends of the list
 - **Backward traversal:** Doubly linked lists allow traversal in both forward and backward directions



Doubly Linked Lists

❖ Why $O(1)$?



- ❖ Thanks to the tail pointer deletion or addition on either sides **take 1 step at worst**, thus $O(1)$

Doubly Linked Lists

❖ Cons:

- **Extra memory:** Doubly linked lists require more memory than singly linked lists due to the additional previous pointers
- **More complex implementation:** Maintaining the previous pointers requires more careful management of node connections

Implementation

Node Class

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

Constructor

```
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
```

Implementation

```
def insert_at_beginning(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        self.tail = new_node  
    else:  
        new_node.next = self.head  
        self.head.prev = new_node  
        self.head = new_node
```

$O(1)$ to insert at the beginning

```
def insert_at_end(self, data):  
    new_node = Node(data)  
    if self.head is None:  
        self.head = new_node  
        self.tail = new_node  
    else:  
        new_node.prev = self.tail  
        self.tail.next = new_node  
        self.tail = new_node
```

$O(1)$ compared to $O(n)$ for
Singly Linked Lists

Implementation

```
def insert_after_node(self, key, data):
    new_node = Node(data)
    current = self.head
    while current:
        if current.data == key:
            new_node.next = current.next
            new_node.prev = current
            if current.next:
                current.next.prev = new_node
            else:
                self.tail = new_node
            current.next = new_node
            break
        current = current.next
```

The complexity is $O(n)$, since in the worst case we insert at the end (if we insert after the last element)



Implementation

```
def delete_at_beginning(self):  
    if self.head is None:  
        return  
    if self.head.next is None:  
        self.head = None  
        self.tail = None  
    else:  
        self.head = self.head.next  
        self.head.prev = None
```

$O(1)$ to delete at the beginning

```
def delete_at_end(self):  
    if self.head is None:  
        return  
    if self.head.next is None:  
        self.head = None  
        self.tail = None  
    else:  
        self.tail = self.tail.prev  
        self.tail.next = None
```

$O(1)$ complexity compared
to $O(n)$ for SLLs

Implementation

```
def delete_node(self, key):  
    if self.head is None:  
        return  
    if self.head.data == key:  
        self.delete_at_beginning()  
        return  
    if self.tail.data == key:  
        self.delete_at_end()  
        return  
    current = self.head  
    while current:  
        if current.data == key:  
            current.prev.next = current.next  
            current.next.prev = current.prev  
            break  
        current = current.next
```

The time complexity of deleting a specified is $O(n)$ due to the potential need to traverse the list

Implementation

```
def search(self, key):  
    current = self.head  
    while current:  
        if current.data == key:  
            return True  
        current = current.next  
    return False
```

The time complexity to search is $O(n)$ since in the worst case we need to **traverse the list**

Interview-Style Question - Linked List

- ❖ Reverse a Singly Linked list

```
def reverse_linked_list(head):  
    current = head  
    prev = None  
    while current:  
        next_node = current.next  
        current.next = prev  
        current.prev = next_node  
        prev = current  
        current = next_node  
    return prev
```

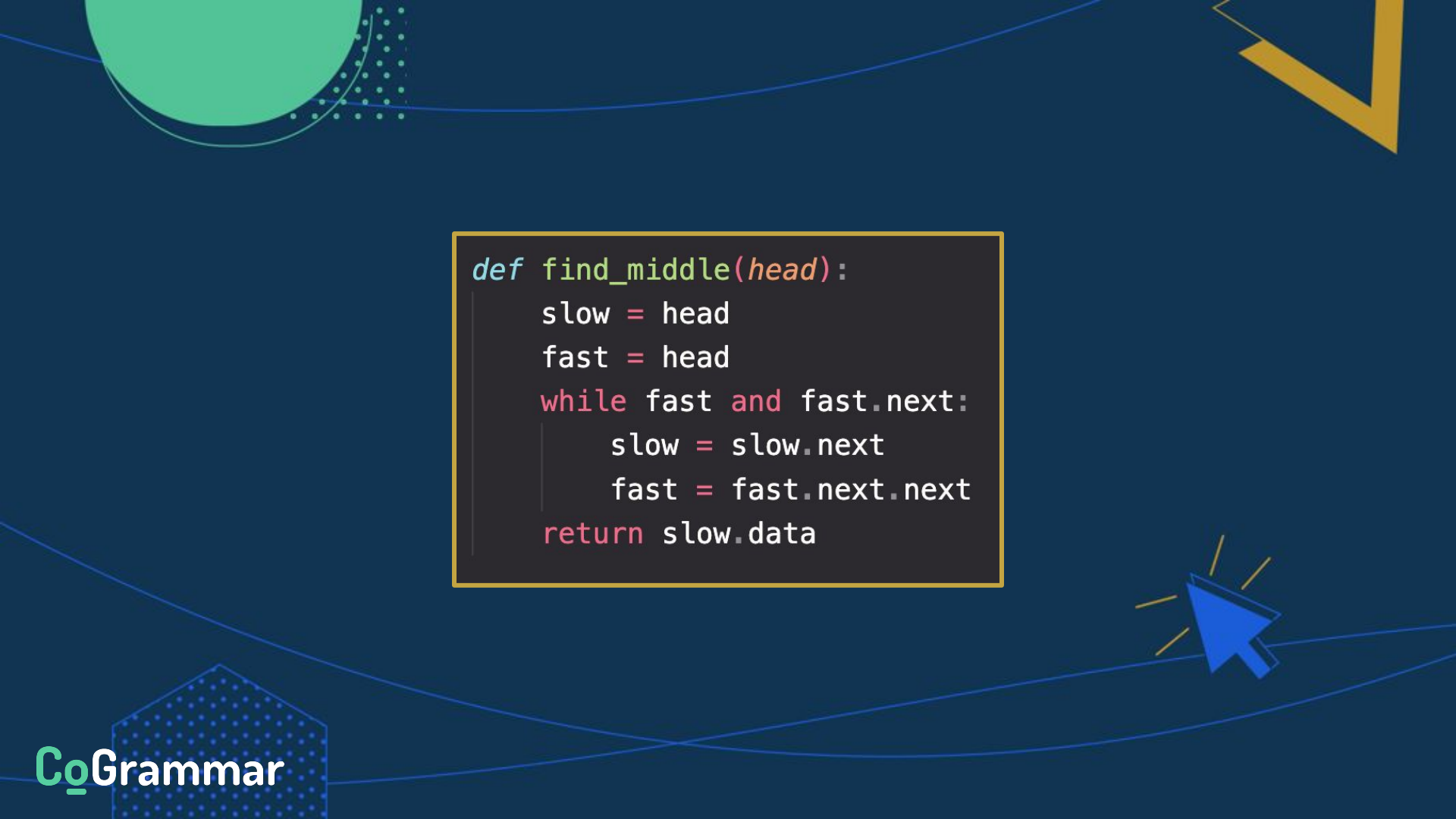
Interview-Style Question - Linked List

- ❖ Detect a loop in a linked list

```
def detect_loop(head):  
    slow = head  
    fast = head  
    while slow and fast and fast.next:  
        slow = slow.next  
        fast = fast.next.next  
        if slow == fast:  
            return True  
    return False
```

Interview-Style Question - Linked List

- ❖ Find the middle element of a linked list



```
def find_middle(head):  
    slow = head  
    fast = head  
    while fast and fast.next:  
        slow = slow.next  
        fast = fast.next.next  
    return slow.data
```



Which of the following is NOT a property of a linked list?

- A. Dynamic size
- B. Efficient insertion and deletion at the beginning
- C. Random access to elements
- D. Sequential access to elements



Answer: C

- ❖ Linked lists do not support random access to elements, which means accessing an element at a specific index requires traversing the list from the beginning. This results in a time complexity of $O(n)$ for accessing elements, unlike arrays that provide random access with $O(1)$ time complexity.

Homework

Practice the skills we've developed by completing the following problems:

- ❖ The next slide contains two questions to test your theoretical understanding of linked lists.
- ❖ We'll be going through two LeetCode examples in the lecture over the weekend, attempt them yourself and come ready with questions:
 - [Example 1](#)
 - [Example 2](#)
- ❖ Practice speaking through your solutions and explaining how you approached each problem.

Homework

Imagine you're tasked with creating a feature for a music streaming service that allows users to manage their playlists. Users should be able to add songs to the beginning, the end, and delete specific songs efficiently.

1. Which linear data structure would you use to implement the playlist feature and why?
2. How would the chosen data structure affect the implementation of adding or removing songs?

Homework

Imagine you're tasked with creating a feature for a music streaming service that allows users to manage their playlists. Users should be able to add songs to the beginning, the end, and delete specific songs efficiently.

1. Which linear data structure would you use to implement the playlist feature and why?
Singly Linked List: For the playlist, a singly linked list is ideal due to its efficient insertion and deletion operations at both the beginning and the end.
2. How would the chosen data structure affect the implementation of adding or removing songs?
Adding Songs: With a singly linked list, songs can be added to the start (prepend) in $O(1)$ time or to the end (append) in $O(1)$ time if a tail pointer is maintained. Removing Songs: Deletion from any point is efficient, especially when a direct reference to the node (song) is available, otherwise, it takes $O(n)$ time to search and remove a node.

CoGrammar

Q & A SECTION

**Please use this time to ask
any questions relating to the
topic, should you have any.**

Thank you for attending



CoGrammar



Department
for Education