Welcome to the CoGrammar Linear Data Structures & Strings

The session will start shortly...

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

Coding Interview Workshop Housekeeping

- 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**

Coding Interview Workshop Housekeeping cont.

- 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**

Skills Bootcamp
8-Week Progression Overview

Fulfil 4 Criteria to Graduation

✅ Criterion 1: Initial Requirements

Timeframe: First 2 Weeks
Guided Learning Hours (GLH): Minimum of 15 hours
Task Completion: First four tasks

Due Date: 24 March 2024

✅ Criterion 2: Mid-Course Progress

60 Guided Learning Hours

Data Science - **13 tasks**
Software Engineering - **13 tasks**
Web Development - **13 tasks**

Due Date: 28 April 2024

CoGrammar

Skills Bootcamp Progression Overview

✅ Criterion 3: Course Progress

Completion: All mandatory tasks, including Build Your Brand and resubmissions by study period end
Interview Invitation: Within 4 weeks post-course
Guided Learning Hours: Minimum of 112 hours by support end date (10.5 hours average, each week)

✅ Criterion 4: Demonstrating Employability

Final Job or Apprenticeship Outcome: Document within 12 weeks post-graduation
Relevance: Progression to employment or related opportunity

Lecture Objectives

- 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.

CoGrammar

Lecture Objectives

- 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.

CoGrammar

Lecture Objectives

- Explain the necessity of character encoding in representing text data within computer systems.
- Explore and differentiate between common character encoding standards like ASCII, Unicode, UTF-8, focusing on their roles in global text representation and processing.

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. $O(n^2)$

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

```javascript
insertAtBeginning(data) {
  let newNode = new Node(data);
  newNode.next = this.head;
  this.head = newNode;
}
```

CoGrammar

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.

Which character encoding standard is backward compatible with ASCII?

A. Unicode

B. UTF-8

C. UTF-16

D. UTF-32

Answer: B

- UTF-8 is a variable-length encoding scheme for Unicode characters that is backward compatible with ASCII. This means that any valid ASCII character is encoded the same way in UTF-8, allowing seamless integration with older systems.

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

CoGrammar

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.

Portfolio Assignment Reviews

Submit you solutions here!

CoGrammar

Introduction

- Linear data structures and character encoding are fundamental concepts in software engineering, web development, and data science
- They are essential for efficiently storing, organising, and manipulating data
- Understanding these concepts is crucial for solving complex problems and optimizing code performance

Data Structures and Strings

1. Linked Lists
2. Singly Linked Lists
3. Doubly Linked Lists
4. Character Encoding
5. ASCII
6. Unicode
7. UTF-8

CoGrammar

Linked Lists

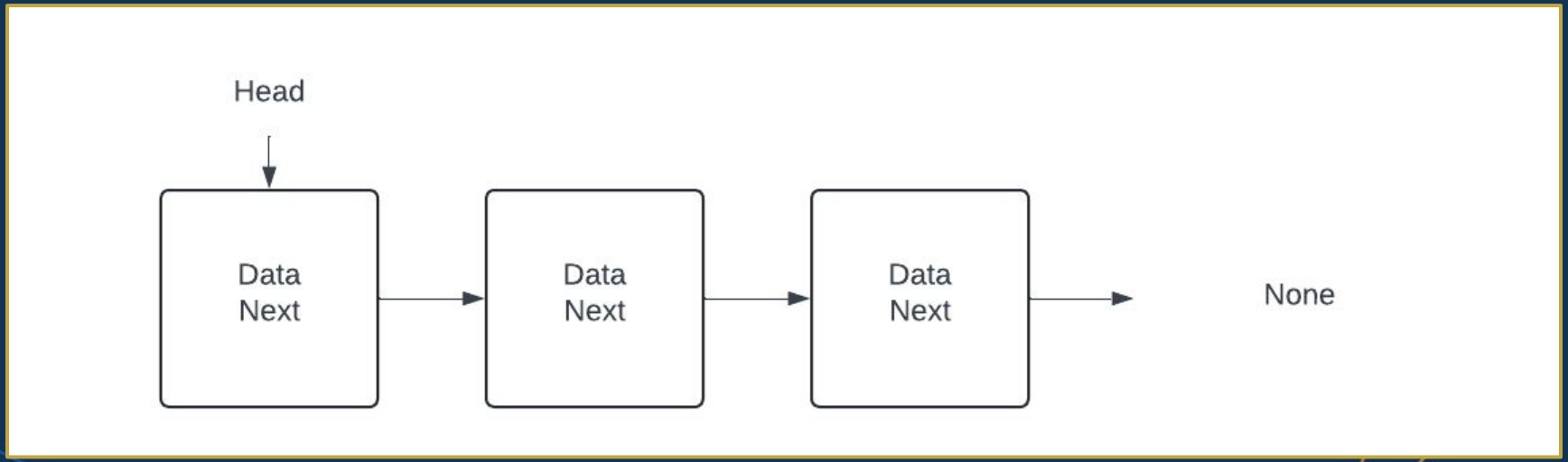
A linked list is a linear data structure consisting of a sequence of nodes, where each node contains data and a reference (link) to the next node in the sequence

- 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

Linked Lists

- 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

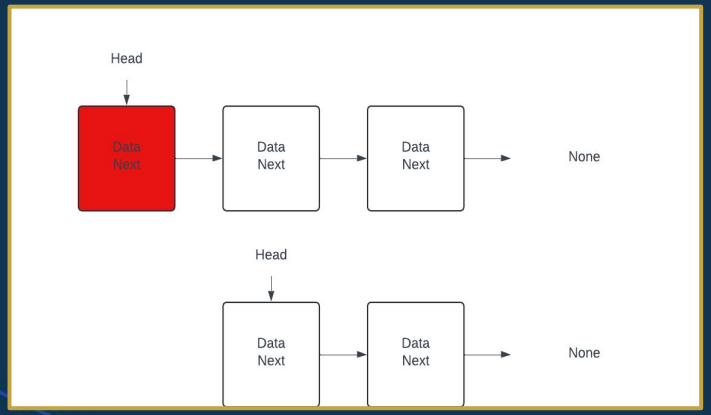


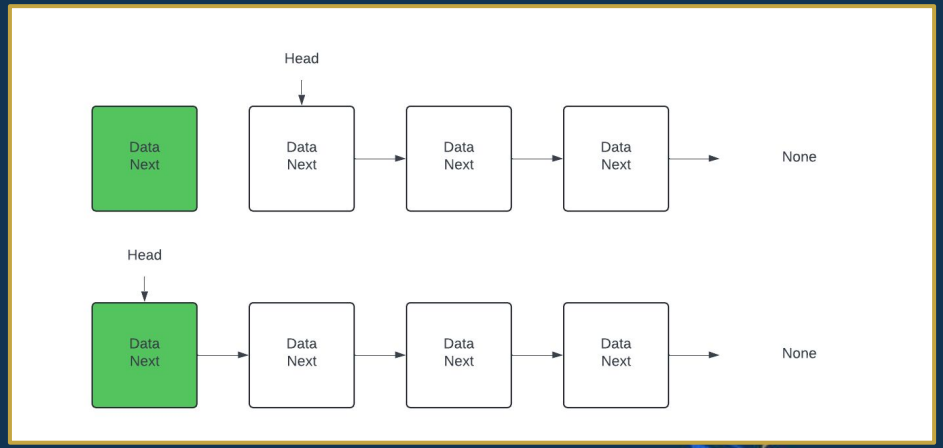
Singly Linked Lists

- 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), you ask? Glad you asked, so 🤓...

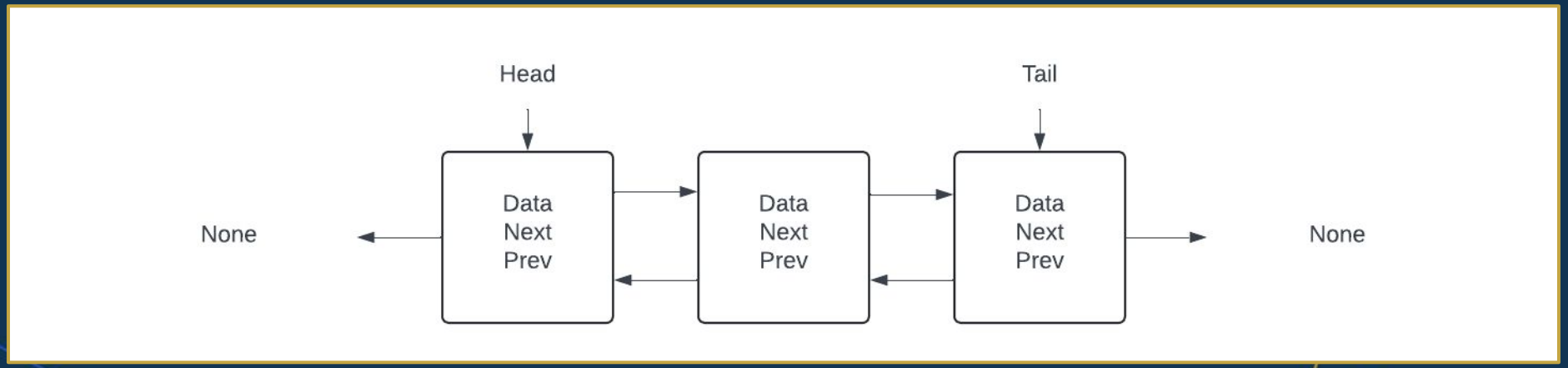




- For removal we only take 1 step no matter the size of the list, and the same for insertion, thus O(1) for worst case

CoGrammar

Doubly Linked Lists

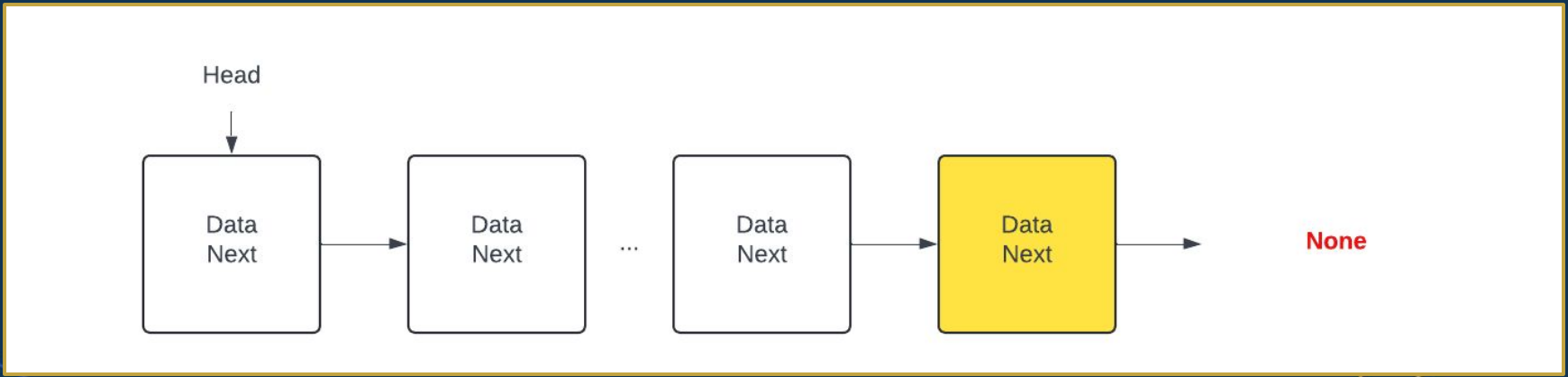


Singly Linked Lists

- 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)

Singly Linked List Node

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

```javascript
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}
```

Singly Linked List Constructor

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

```javascript
class SinglyLinkedList {
  constructor() {
    this.head = null;
  }
```

Singly Linked List Constructor

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

```javascript
class SinglyLinkedList {
  constructor() {
    this.head = null;
  }
```

Singly Linked List Insert at the beginning

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

```javascript
insertAtBeginning(data) {
  let newNode = new Node(data);
  newNode.next = this.head;
  this.head = newNode;
}
```

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

Singly Linked List Insert at the end

```python
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
```

```javascript
insertAtEnd(data) {
  let newNode = new Node(data);
  if (this.head === null) {
    this.head = newNode;
    return;
  }
  let last = this.head;
  while (last.next) {
    last = last.next;
  }
  last.next = newNode;
}
```

- The complexity is O(n), since you need to loop through all n elements to insert the new node

Singly Linked List Insert after some node

```python
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
```

```javascript
insertAfter(prevNode, data) {
  if (prevNode === null) {
    console.log("The given previous node must be in LinkedList.");
    return;
  }
  let newNode = new Node(data);
  newNode.next = prevNode.next;
  prevNode.next = newNode;
}
```

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

Singly Linked List delete at the beginning

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

```javascript
deleteAtBeginning() {
  if (this.head === null) {
    return;
  }
  this.head = this.head.next;
}
```

- As mentioned at the start, the complexity is O(1) to delete at the beginning

Singly Linked List delete at the end

```python
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
```

```javascript
deleteAtEnd() {
  if (this.head === null) {
    return;
  }
  if (this.head.next === null) {
    this.head = null;
    return;
  }
  let secondLast = this.head;
  while (secondLast.next.next) {
    secondLast = secondLast.next;
  }
  secondLast.next = null;
}
```

- The time complexity of deleting a node at the end is O(n) since you have to loop through all n nodes

Singly Linked List delete a specified node

```python
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
```

```javascript
deleteNode(key) {
  let temp = this.head,
    prev = null;

  if (temp !== null && temp.value === key) {
    this.head = temp.next;
    return;
  }

  while (temp !== null && temp.value !== key) {
    prev = temp;
    temp = temp.next;
  }

  if (temp === null) return;

  prev.next = temp.next;
}
```

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

Singly Linked List search

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

```javascript
search(value) {
  let current = this.head;
  while (current !== null) {
    if (current.value === value) {
      return true;
    }
    current = current.next;
  }
  return false;
}
```

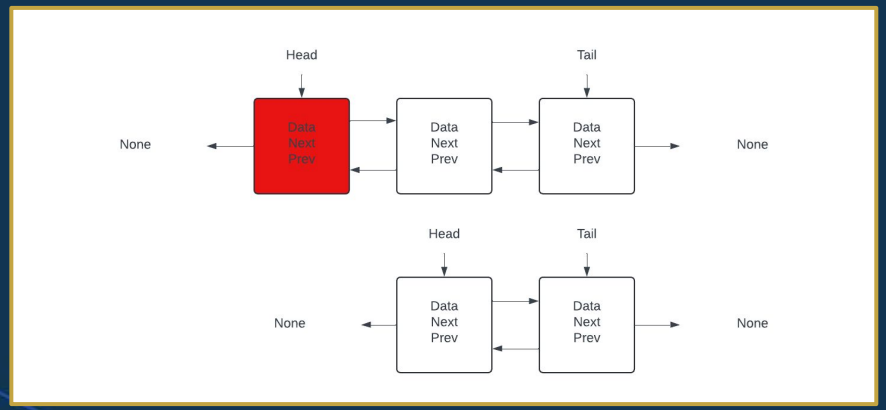
- The time complexity to search is O(n) since in the worst case we need to search for the last element

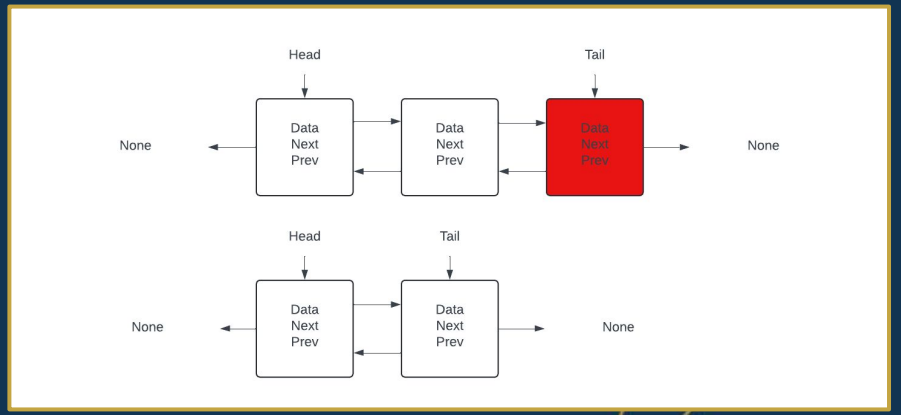
Doubly Linked Lists

- 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)

CoGrammar

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

Doubly Linked List Node

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

```javascript
class Node {
  constructor(data) {
    this.data = data;
    this.next = null;
    this.prev = null;
  }
}
```

Singly Linked List Constructor

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

```javascript
class DoublyLinkedList {
  constructor() {
    this.head = null;
    this.tail = null;
  }
```

CoGrammar

Doubly Linked List Insert at the beginning

```python
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
```

```javascript
insertAtBeginning(data) {
  let newNode = new Node(data);
  if (!this.head) {
    this.head = newNode;
    this.tail = newNode;
  } else {
    newNode.next = this.head;
    this.head.prev = newNode;
    this.head = newNode;
  }
}
```

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

Doubly Linked List Insert at the end

```python
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
```

```javascript
insertAtEnd(data) {
  let newNode = new Node(data);
  if (!this.head) {
    this.head = newNode;
    this.tail = newNode;
  } else {
    newNode.prev = this.tail;
    this.tail.next = newNode;
    this.tail = newNode;
  }
}
```

- The complexity is O(1), unlike for Singly Linked Lists which has a complexity of (n)

Doubly Linked List Insert after some node

```python
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
```

```javascript
insertAfterNode(data, key) {
  let newNode = new Node(data);
  let temp = this.head;
  while (temp) {
    if (temp.data === key) {
      newNode.next = temp.next;
      newNode.prev = temp;
      if (temp.next) {
        temp.next.prev = newNode;
      } else {
        this.tail = newNode;
      }
      temp.next = newNode;
      break;
    }
    temp = temp.next;
  }
}
```

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

Doubly Linked List delete at the beginning

```python
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
```

```javascript
deleteFromBeginning() {
  if (!this.head) {
    return;
  }
  if (this.head === this.tail) {
    this.head = null;
    this.tail = null;
  } else {
    this.head = this.head.next;
    this.head.prev = null;
  }
}
```

- As mentioned at the start, the complexity is O(1) to delete at the beginning

Doubly Linked List delete at the end

```python
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
```

```javascript
deleteAtEnd() {
  if (!this.head) {
    return;
  }
  if (this.head === this.tail) {
    this.head = null;
    this.tail = null;
  } else {
    this.tail = this.tail.prev;
    this.tail.next = null;
  }
}
```

- Unlike Singly Linked List which has a complexity of O(n) to delete at end, the Doubly Linked List has a O(1) complexity

Doubly Linked List delete a specified node

```python
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
```

```javascript
deleteNode(key) {
  let temp = this.head;
  while (temp) {
    if (temp.data === key) {
      if (temp === this.head) {
        this.deleteFromBeginning();
      } else if (temp === this.tail) {
        this.deleteAtEnd();
      } else {
        temp.prev.next = temp.next;
        temp.next.prev = temp.prev;
      }
      break;
    }
    temp = temp.next;
  }
}
```

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

Doubly Linked List search

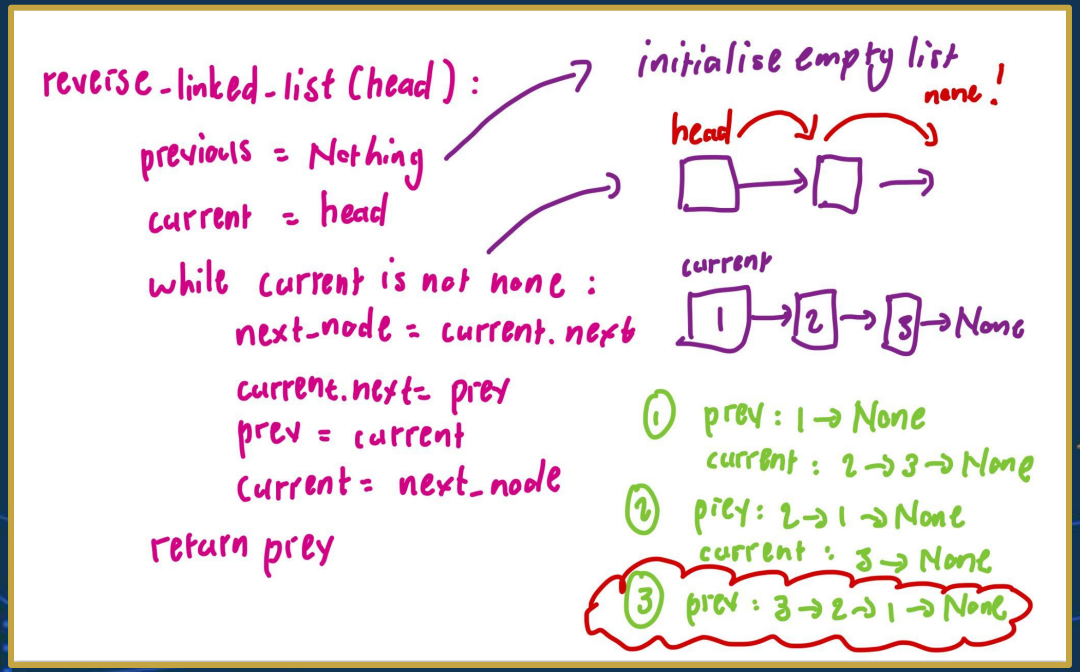
```python
def search(self, key):
    current = self.head
    while current:
        if current.data == key:
            return True
        current = current.next
    return False
```

```javascript
searchNode(key) {
  let temp = this.head;
  while (temp) {
    if (temp.data === key) {
      return true;
    }
    temp = temp.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



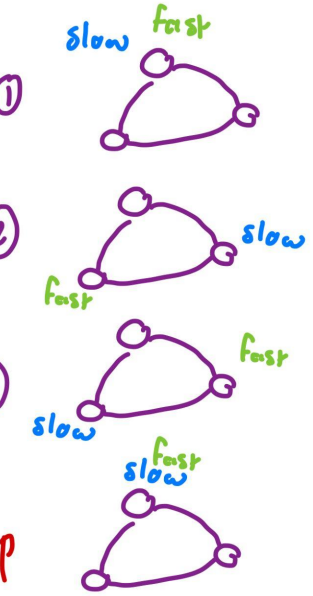
```python
def reverse_linked_list(head):
    current = head
    prev = None
    while current:
        next_node = current.next
        current.next = prev
        current.prev = next_node
        prev = current
        current = next_node
    return prev
```

```javascript
function reverseLinkedList(list) {
  let temp = null;
  let current = list.head;
  while (current) {
    temp = current.prev;
    current.prev = current.next;
    current.next = temp;
    current = current.prev;
  }
  if (temp) {
    list.head = temp.prev;
  }
}
```

Interview-Style Question - Linked List

- Detect a loop in a linked list

```
detect_loop(head)
    slow = head
    fast = head
    while slow and fast and fast.next :
        increment slow
        increment fast twice
        if slow == false
            return true
    return false
```



```python
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
```

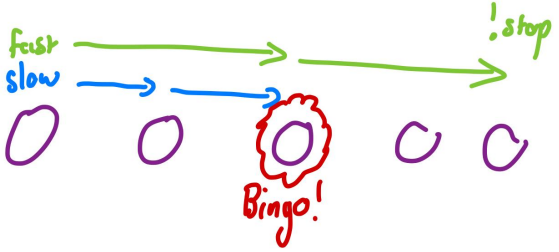
```javascript
function detectLoop(list) {
  let slow = list.head;
  let fast = list.head;
  while (slow && fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
    if (slow === fast) {
      return true;
    }
  }
  return false;
}
```

CoGrammar

Interview-Style Question - Linked List

- Find the middle element of a linked list

```
find_middle(head):
    slow = head
    fast = head
    while fast and fast.next:
        increment slow
        increment fast twice   } so slow would be half of fast at any point
    return slow value
```



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

```javascript
function findMiddle(list) {
  let slow = list.head;
  let fast = list.head;
  while (fast && fast.next) {
    slow = slow.next;
    fast = fast.next.next;
  }
  return slow.data;
}
```

CoGrammar

Let’s Breathe!

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

CoGrammar

Character Encoding

Character encoding is the process of representing characters from a given character set using a specific encoding scheme

- Importance:
  - Ensures consistent representation and interpretation of text across different systems and platforms
  - Enables the exchange of text data between computers with different architectures and operating systems

Character Encoding

- Common character encoding standards:
  - ASCII (American Standard Code for Information Interchange)
  - Unicode (Universal Coded Character Set)
  - UTF-8 (Unicode Transformation Format - 8-bit)

ASCII

- Represents 128 characters, including:
  - Uppercase and lowercase English letters (A-Z, a-z)
  - Digits (0-9)
  - Punctuation marks and special characters
- Limitations:
  - Not sufficient for representing characters from non-English languages
  - No support for emoji or special symbols 🥺

ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Source: https://theasciicode.com.ar/

Unicode

- International standard for character encoding
- Assigns a unique code point to every character from various writing systems (1,112,064 code points as of 2024)
- Code points are typically represented in the format "U+XXXX", where XXXX is a hexadecimal number

Source: https://home.unicode.org/

UTF-8

- A Unicode transformation format (UTF) is an algorithmic mapping from every Unicode code point (except surrogate code points) to a unique byte sequence.
- Backward compatible with ASCII
- Uses 1 to 4 bytes to represent each character
- Python 3 uses Unicode (UTF-8) as the default encoding for strings
- JavaScript uses Unicode (UTF-16) as the default encoding for strings

UTF-8

- Examples:
  - The character "A" (U+0041) is encoded as a single byte: 0x41
  - The character "Ω" (U+03A9) is encoded as two bytes: 0xCE 0xA9
  - The character "😀" (U+1F600) is encoded as four bytes: 0xF0 0x9F 0x98 0x80

```python
text = "Hello, world! 😀"
encoded_text = text.encode("utf-8")
print(encoded_text)  # Output: b'Hello, world! \xf0\x9f\x98\x80'

decoded_text = encoded_text.decode("utf-8")
print(decoded_text)  # Output: Hello, world! 😀
```

```javascript
const text = "Hello, world! 😀";
const encodedText = encodeURIComponent(text);
console.log(encodedText); // Output: Hello%2C%20world!%20%F0%9F%98%80

const decodedText = decodeURIComponent(encodedText);
console.log(decodedText); // Output: Hello, world! 😀
```

Interview-Style Question - Encoding

- Implement a function to detect the character encoding of a given text file
- Pseudocode
  - Open a text file and read it in
  - Use already existing functions to detect encoding
  - This is an example, unlikely that companies would ask this or much encoding-related questions 😉

```python
import chardet

def detect_encoding(file_path):
    with open(file_path, 'rb') as file:
        result = chardet.detect(file.read())
        return result['encoding']
```

```javascript
const chardet = require("chardet");
const fs = require("fs");

function detectEncoding(filePath) {
  const buffer = fs.readFileSync(filePath);
  return chardet.detect(buffer);
}
```

CoGrammar

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.

What is the maximum number of bytes used to represent a single character in UTF-8 encoding?

A. 1

B. 2

C. 3

D. 4

Answer: D

- UTF-8 is a variable-length encoding scheme that uses 1 to 4 bytes to represent each Unicode character. Characters with higher code points, such as some emoji or less common symbols, require 4 bytes for their UTF-8 representation.

Which of the following is an advantage of using Unicode over ASCII?

A. Faster processing speed

B. Smaller memory footprint

C. Support for a wider range of characters

D. Compatibility with older systems

CoGrammar

Answer: D

- Unicode is an international standard for character encoding that assigns a unique code point to every character from various writing systems. Compared to ASCII, which only supports 128 characters, Unicode provides support for a much wider range of characters, including those from non-English languages, emoji, and special symbols.

Portfolio Assignment: SE

Linked List Library

Objective: Develop a library for efficient manipulation of linked lists in Python or JavaScript, providing an easy-to-use interface for creating, manipulating, and querying linked lists, optimised for performance and thoroughly tested.

Portfolio Assignment: SE

Requirements:

- Implement a linked list library in Python or JavaScript that provides an easy-to-use interface for creating, manipulating, and querying linked lists.
- Include support for both singly and doubly linked lists, with methods for insertion, deletion, searching, and traversal.
- Optimise the library for performance, considering the time complexity of each operation.
- Write unit tests to ensure the correctness of the library's functionality.
- Create a README file with documentation on how to install, use, and contribute to the library.

Portfolio Assignment: DS

Character Encoding Detector

Objective: Develop a Python script that detects the character encoding of text samples in a given dataset, analyses the distribution of different encodings, and discusses the implications of character encoding on data cleaning, preprocessing, and analysis in data science projects.

Portfolio Assignment: DS

Requirements:

- Create a Python script that takes a dataset containing text data as input and detects the character encoding of each text sample.
- Use libraries like chardet or cChardet to perform encoding detection.
- Analyse the distribution of different character encodings in the dataset and visualise the results using matplotlib or seaborn.
- Discuss the implications of character encoding on data cleaning, preprocessing, and analysis in the context of data science projects.
- Provide a Jupyter Notebook with well-documented code, explanations, and insights.

Portfolio Assignment: WD

Web-Based Text Converter

Objective: Create a web application that allows users to input text and convert it between different character encodings, providing an intuitive user interface and error handling for unsupported characters or encoding-related issues.

Portfolio Assignment: WD

Requirements:

- Create a web application using HTML, CSS, and JavaScript that allows users to input text and convert it between different character encodings (e.g., ASCII, UTF-8, UTF-16).
- Implement the encoding conversion logic using JavaScript or a server-side language like Python or Node.js.
- Design an intuitive user interface that displays the input text, allows selecting the source and target encodings, and shows the converted output.
- Include error handling for unsupported characters or encoding-related issues.
- Deploy the web application on a platform like GitHub Pages or Heroku and provide a link to the live demo in the README file.

CoGrammar

Q & A SECTION

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

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar