

## LINEAR DATA STRUCTURES

**SKILLS  
FOR LIFE**

**SKILLS BOOTCAMPS**



Department  
for Education

## Foundational Sessions Housekeeping

---

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.  
**(FBV: Mutual Respect.)**
- 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 Open Classes.

You can submit these questions here:

[SE Open Class Questions](#) or [DS Open Class Questions](#)

## Foundational Sessions Housekeeping cont.

---

- For all **non-academic questions**, please submit a query: [www.hyperiondev.com/support](https://www.hyperiondev.com/support)
- Report a **safeguarding** incident: [www.hyperiondev.com/safeguardreporting](https://www.hyperiondev.com/safeguardreporting)
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

# Reminders!

## **Guided Learning Hours**

*By now, ideally you should have 7 GLHs per week accrued.  
Remember to attend any and all sessions for support, and to  
ensure you reach 112 GLHs by the close of your Skills Bootcamp.*

# Progression Criteria

## ✓ **Criterion 1: Initial Requirements**

- Complete 15 hours of Guided Learning Hours and the first four tasks within two weeks.

## ✓ **Criterion 2: Mid-Course Progress**

- Software Engineering: Finish 14 tasks by week 8.
- Data Science: Finish 13 tasks by week 8.

## ✓ **Criterion 3: Post-Course Progress**

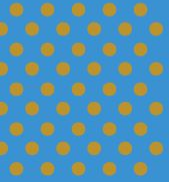

- Complete all mandatory tasks by 24th March 2024.
- Record an Invitation to Interview within 4 weeks of course completion, or by 30th March 2024.
- Achieve 112 GLH by 24th March 2024.

## ✓ **Criterion 4: Employability**

- Record a Final Job Outcome within 12 weeks of graduation, or by 23rd September 2024.





# What is a singly linked list?

- 
- A.** A type of array.
  - B.** A collection of nodes linked sequentially.
  - C.** A data structure with key-value pairs.
  - D.** A fixed-size data structure.
- 




# How do arrays differ from linked lists?

- 
- A.** Arrays are faster for appending elements.
  - B.** Arrays require more memory for storage.
  - C.** Arrays are non-linear data structures.
  - D.** Arrays allow direct access to elements.
- 



# Which of the following operations is typically more time-efficient in a dynamic array compared to a linked list?

- A.** Inserting an element at the end.
  - B.** Inserting an element at the beginning.
  - C.** Accessing an element by index.
  - D.** Deleting an element from the middle.
- 



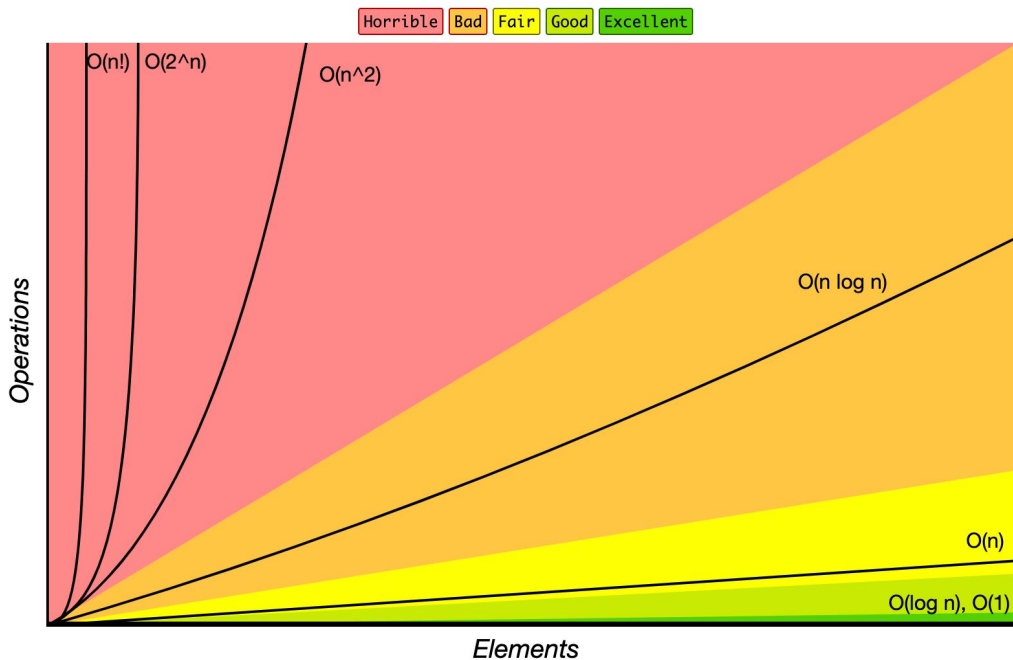
# Recap of Order Complexity



# Common Complexities

The most used complexities in industry.

Very important to understand which complexities are preferred in different situations.



# Asymptotes

- A line that a curve approaches as it heads towards infinity.
- Three types:
  1. **Vertical:** Indicates points where an algorithm becomes highly inefficient or fails, such as functions approaching infinity when the input nears a specific value.
  2. **Horizontal:** Represents algorithms whose performance stabilizes or becomes constant with large input sizes, akin to cache lookup operations.

# Asymptotes

**3. Oblique:** Demonstrates algorithms with complexity that increases non-linearly with input size, similar to complexities like  $O(n \log n)$ , growing more than linearly but less steeply than quadratic.

# Algorithmic Complexity

- The measure of the **amount of resources**, such as time and/or space, **required by an algorithm** to **solve a problem** as a **function of the size of the input**.

# Common Sorting and Searching Algorithms

## Common sorting algorithms:

1. **Bubble Sort** - Time Complexity:  $O(n^2)$ , Space Complexity:  $O(1)$
2. **Selection Sort** - Time Complexity:  $O(n^2)$ , Space Complexity:  $O(1)$
3. **Insertion Sort** - Time Complexity:  $O(n^2)$ , Space Complexity:  $O(1)$
4. **Merge Sort** - Time Complexity:  $O(n \log(n))$ , Space Complexity:  $O(n)$
5. **Quick Sort** - Time Complexity:  $O(n \log(n))$  average,  $O(n^2)$  worst-case, Space Complexity:  $O(\log(n))$
6. **Heap Sort** - Time Complexity:  $O(n \log(n))$ , Space Complexity:  $O(1)$

## Common searching algorithms:

1. **Linear Search** - Time Complexity:  $O(n)$ , Space Complexity:  $O(1)$
2. **Binary Search** - Time Complexity:  $O(\log(n))$ , Space Complexity:  $O(1)$
3. **Jump Search** - Time Complexity:  $O(\sqrt{n})$ , Space Complexity:  $O(1)$
4. **Interpolation Search** - Time Complexity:  $O(\log(\log(n)))$  average,  $O(n)$  worst-case, Space Complexity:  $O(1)$



# Linear Data Structure Topics

1. Singly Linked Lists
2. Arrays
3. Memory Allocation and Access Management of Lists and Arrays



# Music Streaming Service

Imagine a music streaming service like **Spotify** or **Apple Music**. Users can create playlists, **add and remove songs, and browse through a vast library.**

The service needs to handle millions of users and songs **efficiently.**

Choosing the right data structure for managing these playlists and song libraries is crucial for performance, especially for operations like searching for a song, adding a new song to a playlist, or rearranging songs in a playlist.

# Example: Organising Songs

## Random Songs:

- 'Bohemian Rhapsody'
- 'Shape of You'
- 'Yesterday'
- 'Uptown Funk'
- 'Rolling in the Deep'

## How would you organise these songs?

- Alphabetically, by genre, by artist?
- How would your method help in adding new songs, removing songs, and searching for songs efficiently?

Regardless of the specific data organization strategy used in your application, two fundamental linear data structures often utilized are **Linked Lists** and **Arrays**.

More on that in the lecture 😊

# Singly Linked Lists

A singly linked list is a collection of nodes, where each node contains data and a reference to the next node.

- Ideal for **dynamic data storage** where the size of the **data set isn't fixed**.
- **Each element in a linked list is a node**, represented by the Node class.

# Node Class in Python

A node has two attributes:

1. **Data** (the value it holds)
2. **Next** (reference to the next node).

```
class Node:  
    def __init__(self, data):  
        self.data = data # Store the data  
        self.next = None # Initialize the next reference to None
```

# SinglyLinkedList Class - Initialization

- The SinglyLinkedList class initializes the list (Hint: **OOP**).
- It starts with a head node, initially set to None.

```
class SinglyLinkedList:  
    def __init__(self):  
        self.head = None  # Initialize the head of the list to None
```

# Appending Elements

- Appending **adds a new node at the end of the list.**
- We traverse to the list's end and link the last node to the new one.

```
# Method to add a new element at the end of the list
def append(self, data):
    new_node = Node(data) # Create a new node with the given data
    if self.head is None:
        self.head = new_node # If the list is empty, make the new node the head
        return
    # If the list is not empty, traverse to the end of the list
    last_node = self.head
    while last_node.next:
        last_node = last_node.next
    last_node.next = new_node # Link the last node to the new node
```

# Prepending Elements

- Prepending **inserts a new node at the beginning**.
- It's an efficient operation, setting the new node as the head.

```
# Method to add a new element at the beginning of the list  
def prepend(self, data):  
    new_node = Node(data) # Create a new node with the given data  
    new_node.next = self.head # Link the new node to the current head  
    self.head = new_node # Make the new node the new head of the list
```



# Traversing the List

- Traversing means **visiting each node, starting from the head**.
- It's used for operations like printing the list's contents.

```
# Method to print all elements in the list
def print_list(self):
    cur_node = self.head
    while cur_node:
        print(cur_node.data, end=" -> ") # Print the data of each node
        cur_node = cur_node.next
    print("None") # Indicate the end of the list
```

# Singly Linked List in Action

- **Basic Example:** We append 3 and 5, then prepend 1 to the list.

```
sll = SinglyLinkedList()  
sll.append(3)  # Appending 3 to the list  
sll.append(5)  # Appending 5 to the list  
sll.prepend(1) # Prepending 1 to the list  
sll.print_list() # Output: 1 -> 3 -> 5 -> None
```

- **Advanced Example:** We delete 5 from the list.

```
sll.delete(5) # Deleting 5 from the list
sll.print_list() # Output: 1 -> 3 -> None
```

- Keep in mind the delete method seems more complex since it has more lines of code. After walking through it step by step it will make more sense, but this is an “extra credit” problem for the adventurous amongst you 😊

```
def delete(self, key):
    cur_node = self.head

    # Case 1: Deleting the head node
    if cur_node and cur_node.data == key:
        self.head = cur_node.next
        cur_node = None
        return

    # Case 2: Deleting a non-head node
    prev = None
    while cur_node and cur_node.data != key:
        prev = cur_node
        cur_node = cur_node.next

    # If the key is not found in the list
    if cur_node is None:
        return

    # Unlink the node from the list
    prev.next = cur_node.next
    cur_node = None
```

# Arrays

Arrays are a fixed size data structure for storing elements in a contiguous memory location.

- **Contiguous:** Next to or in sequence.
- They allow **fast access to elements** using indices.
- Arrays are of **fixed size**, and resizing them requires creating a new array.

# Array Operations

- Common operations include **accessing elements**, **updating values**, and **iterating over elements**.

```
# Example of array operations in Python
from array import array
arr = array('i', [1, 2, 3, 4])
# Access
print(arr[0]) # Output: 1
# Insertion
arr.append(5)
print(arr) # Output: array('i', [1, 2, 3, 4, 5])
```

```
# Deletion
arr.pop()
print(arr) # Output: array('i', [1, 2, 3, 4])
# Update
arr[0] = 0
arr[1] = 1
arr[2] = 2
print(arr) # Output: array('i', [0, 1, 2, 4])
# Iterate
for i in arr:
    print(i, end=" ") # Output: 0 1 2 4
```

# Let's Breathe

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

**Next up: Numpy Arrays and Python  
Lists 😊**



# Numpy Arrays and Python Lists

**Numpy arrays are designed for numerical computation, offering advantages in performance and memory usage. Python lists, while flexible, are less memory-efficient and slower in certain operations.**

# Numpy Array Operations

- Operations include **mathematical computations**, **reshaping**, **slicing**, and more. Numpy arrays also support **vectorized operations**, making calculations faster.

```
import numpy as np

# Creating a Numpy array
np_arr = np.array([1, 2, 3, 4])

# Mathematical Computations
# Vectorized addition
addition = np_arr + np_arr
print(addition) # Output: [2, 4, 6, 8]
```

```
# Reshaping
reshaped = np_arr.reshape((2, 2))
print(reshaped) # Output: [[1, 2], [3, 4]]

# Slicing
sliced = np_arr[1:3]
print(sliced) # Output: [2, 3]
```



- These operations are optimised for performance and are **more efficient than similar operations on Python lists.**

```
# Advanced: Broadcasting and Matrix operations  
multiplied = np_arr * np_arr  
print(multiplied)  # Output: [1, 4, 9, 16]
```

# Python List Operations

- Python lists are versatile, allowing for **storage of different data types** and supporting a variety of operations. They are ideal for collections that require **flexibility in data types and sizes**.

```
arr = [1, 2, 3, 4]
# Access
print(arr[0]) # Output: 1
# Insertion
arr.append(5)
print(arr) # Output: [1, 2, 3, 4, 5]
```

```
# Deletion
arr.pop()
print(arr) # Output: [1, 2, 3, 4]
# Update
arr[0] = 0
arr[1] = 1
arr[2] = 2
print(arr) # Output: [0, 1, 2, 4]
# Iterate
for i in arr:
    print(i, end=" ") # Output: 0 1 2 4
```

# Performance Comparison

## Numpy Array

- **Access:**  $O(1)$
- **Append:**  $O(n)$  - when creating new array.
- **Delete:**  $O(n)$
- **Iteration:**  $O(n)$ , but faster due to contiguous memory and cache efficiency.
- **Vectorized Operations:**  $O(n)$

## Python List

- **Access:**  $O(1)$
- **Append:**  $O(1)$
- **Delete:**  $O(n)$
- **Iteration:**  $O(n)$ , but slower, especially on large data sets.
- **Vectorized Operations:** Not supported.

Selecting the right data structure depends on the specific requirements of your application.

### Numpy Array

- Data Analysis and Scientific Computing - **better for handling large numerical datasets**
- High-Performance Computing - can **implement operations that can be vectorized for speed**

### Python List

- General-Purpose Programming - good for **collections of mixed data types or where the size changes frequently**
- Simple, Small Collections - more **suitable for smaller or less complex data**

## Worked Example

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?

## Worked Example

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.**

# Summary

---

## Singly Linked Lists

- ★ A collection of nodes where each node points to the next.
- ★ Efficient for operations like insertion and deletion, especially at the beginning.

## Arrays

- ★ A static data structure with elements stored in contiguous memory locations.
- ★ Offers quick access via indices but resizing is costly.

# Summary

---

## Complexity Analysis of Singly Linked Lists and Arrays

### ★ Singly Linked Lists:

- Search:  $O(n)$
- Insertion/Deletion at start:  $O(1)$
- Insertion/Deletion at end:  $O(n)$  (if tail is not tracked)

### ★ Arrays:


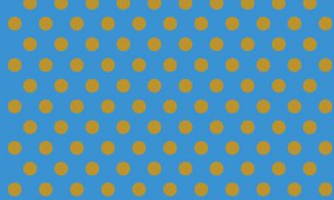
- Access:  $O(1)$
- Insertion/Deletion at end:  $O(1)$  (amortized for dynamic arrays)
- Insertion/Deletion at start or middle:  $O(n)$



# Further Learning



---

- [Understanding Numpy Arrays](#)
- [Python Lists in depth](#)
- [Performance of NumPy vs Lists](#)
- [Time Complexity of Python Data Structures](#)

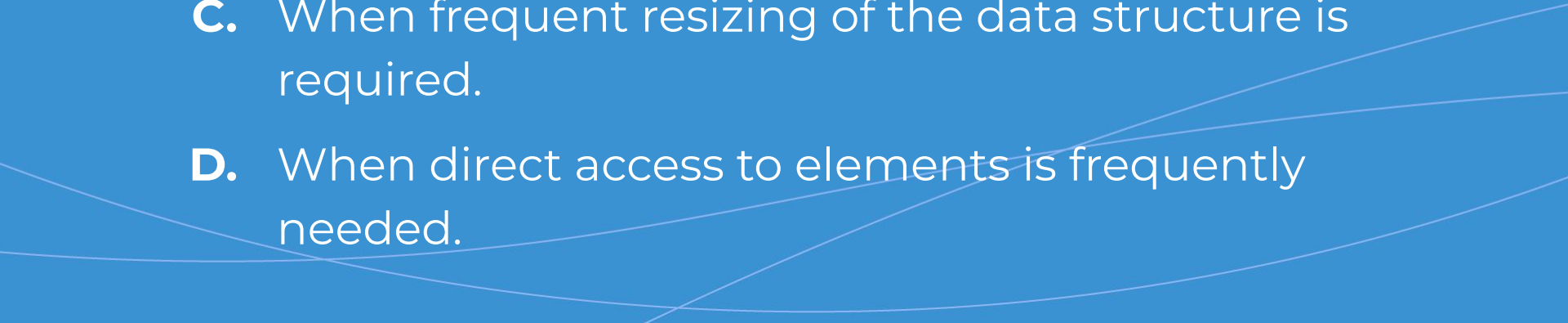


# Which data structure would you choose for quick access to elements in a large dataset?

- A. Singly linked list.
  - B. Python list.
  - C. Numpy array.
- 



# In what scenario is appending elements more efficient in a linked list than an array?

- A.** When memory is not a constraint.
  - B.** When dealing with a small number of elements.
  - C.** When frequent resizing of the data structure is required.
  - D.** When direct access to elements is frequently needed.
- 



# Questions and Answers

Questions around Linear Data Structures





# CoGrammar

# Thank you for joining us

1. Take regular breaks
2. Stay hydrated
3. Avoid prolonged screen time
4. Practice good posture
5. Get regular exercise

*“With great power comes great responsibility”*

---