🧠 Introduction to Data Structures

🌍 What's the Big Idea?

Every program handles **data**. To manage data efficiently, we use **data structures** — organized ways of storing and accessing information in memory.

We divide them into two broad categories:

1. **Physical Data Structures**
2. **Logical Data Structures**

⚙️ 1. Physical Data Structures

These define **how data is actually stored in memory** — how bytes are arranged and managed. They determine *memory organization*.

There are two core physical data structures:

- **Array**
- **Linked List**

🧩 Array

Definition:

A **collection of contiguous memory locations** that stores elements of the same data type.

Key Points:

- **Contiguous allocation:** All elements are side-by-side in memory.
- **Fixed size:** Once declared, its size **cannot change**.
- **Static in size:** Memory is allocated at creation and remains fixed.
- **Direct support:** Arrays are **natively supported** in C, C++, and Java.
- **Location:** Can exist in **stack** or **heap** memory.

When to Use:

- When you **know the exact size** (number of elements) beforehand.
- When **fast indexing** matters (O(1) access time).

In Python:

- Python doesn't have "C-style arrays."
  Instead, you use:
  - **Lists** (most common): Dynamic and can grow or shrink (arr.append() / arr.pop()).
  - **array module**: import array for numeric-only arrays.

- **NumPy arrays**: for high-performance numerical operations.

```python
# Python Example

arr = [10, 20, 30, 40]

arr[2]  # O(1) access

arr.append(50)  # O(1) amortized, dynamic resizing
```

Python lists are actually implemented as **dynamic arrays in heap memory**, which automatically resize when full.

🔗 **Linked List**

Definition:

A **collection of nodes** where each node contains:

1. **Data**
2. **Reference (link)** to the next node.

Key Points:

- **Non-contiguous memory:** Nodes are scattered in memory, linked by pointers/references.
- **Dynamic size:** Can grow or shrink at runtime.
- **Always allocated in heap memory.**
- **Head pointer** is used to track the first node (stored in stack, points into heap).
- **Efficient insertions and deletions**, especially in the middle.

When to Use:

- When the **size is unknown** or keeps changing.
- When **frequent insertions/deletions** are required.

In Python:

Python doesn't have a built-in linked list, but you can make one easily with classes.

```python
# Python Linked List Example

class Node:

  def __init__(self, data):

    self.data = data

    self.next = None


class LinkedList:
```

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

# Creating nodes dynamically (stored in heap)
n1 = Node(10)
n2 = Node(20)
n1.next = n2
```

Python's list is **not** a linked list — it's a dynamic array.

🧠 Summary: Physical Data Structures

Feature	Array	Linked List
Memory Allocation	Contiguous	Non-contiguous
Size	Fixed	Dynamic
Access Time	O(1)	O(n)
Insertion/Deletion	Costly	Easy
Location	Stack or Heap	Always Heap
Example in Python	list, array, numpy.array	Custom Node Class

🧮 2. Logical Data Structures

These define **how data is used or accessed** — the *discipline or rule* of usage, not how it's stored.

They decide *how elements are inserted, deleted, or processed*.

Main Types:

1. **Stack**
2. **Queue**
3. **Tree**
4. **Graph**
5. **Hash Table**

These are often implemented **using arrays or linked lists** underneath.

🥞 Stack

Rule:

- **LIFO (Last In, First Out)** — the last element inserted is the first to be removed.
- Like a stack of plates.

Operations:

- push() — Add element
- pop() — Remove top element
- peek() — View top element

In Python:

You can use a **list** or the collections.deque:

```
stack = []
stack.append(10)
stack.append(20)
stack.pop()  # removes 20
```

🚌 Queue

Rule:

- **FIFO (First In, First Out)** — first inserted is first removed.
- Like people standing in a line.

Operations:

- enqueue() — Add at the end
- dequeue() — Remove from the front

In Python:

Use collections.deque for efficient queues.

```
from collections import deque

queue = deque()
queue.append(10)
queue.append(20)
queue.popleft()  # removes 10
```

🌳 Tree

Rule:

- **Non-linear data structure**
- Hierarchical relationship — root → branches → leaves.
- Each node connects to child nodes.

In Python:

You can represent trees using nested classes, dictionaries, or Node objects.

```
class Node:
  def __init__(self, val):
    self.val = val
    self.left = None
    self.right = None
```

Used in algorithms, file systems, databases, etc.

🕸️ Graph

Rule:

- Collection of **nodes (vertices)** and **edges** (connections).
- Can be **directed/undirected**, **weighted/unweighted**.

In Python:

Represent with dictionary or adjacency list.

```
graph = {
  "A": ["B", "C"],
  "B": ["A", "D"],
  "C": ["A", "D"]
}
```

Used in social networks, route finding, dependency resolution.

🧾 Hash Table

Rule:

- Stores data in **key-value** pairs.
- Provides **O(1)** average-time access using a **hash function**.

In Python:

Built-in as **dictionaries (dict)**.

```
person = {"name": "Alice", "age": 25}

print(person["name"])  # O(1) access
```

🧩 Relationship Between Physical and Logical Structures

Concept	Description
Physical Data Structures	Define how data is **stored** in memory.
Logical Data Structures	Define how data is **used and accessed.**
Implementation Link	Logical data structures are implemented using physical ones — usually **arrays** or **linked lists.**

💡 Key Takeaways

- **Physical structures (array, linked list)** decide *how memory is organized*.
- **Logical structures (stack, queue, tree, graph, hash table)** decide *how data is used*.
- In **Python**, lists, dicts, and classes make it easy to implement all these concepts.
- Arrays are **static**, linked lists are **dynamic**.
- Logical data structures are often **implemented** using arrays or linked lists as the base.