Introduction of Basic Data Structure

📚 Big Picture: Categories and Relationships

Category	Data Structures	Relationships & Notes
Linear Structures	Array, Linked List, Doubly Linked List, Stack, Queue	Stack/Queue are *specialized uses* of arrays or linked lists
Hierarchical Structures	Binary Tree, AVL Tree, Red-Black Tree	AVL and Red-Black Trees are *self-balancing binary search trees*
Heap Structures	Heap (Min-Heap / Max-Heap), Priority Queue	Heap is the *underlying structure* for Priority Queues
Object-Oriented Programming (OOP)	Classes and objects model all of these ADTs	Needed to define reusable, modular, abstract data types

---

📊 Time Complexities (Summary Table)

Data Structure	Insert	Delete	Search/Access	Special Properties
Array	O(1) at end, O(n) at middle	O(n)	O(1) by index, O(n) by value	Fixed-size, fast random access
Linked List	O(1) at head, O(n) at position	O(1) at head, O(n) at position	O(n)	Dynamic size, sequential access

Doubly Linked List	O(1) at head/tail, O(n) at position	O(1) at head/tail, O(n) at position	O(n)	Bi-directional traversal
Stack (Array/LinkedList)	O(1) push/pop	O(1)	O(n) search	LIFO (Last In First Out)
Queue (Array/LinkedList)	O(1) enqueue/dequeue	O(1)	O(n) search	FIFO (First In First Out)
Binary Search Tree (BST)	O(h)	O(h)	O(h)	h = height; O(log n) if balanced, O(n) if skewed
AVL Tree	O(log n)	O(log n)	O(log n)	Strictly balanced BST (rotations after insert/delete)
Red-Black Tree	O(log n)	O(log n)	O(log n)	Loosely balanced BST (fewer rotations)
Heap (Priority Queue)	O(log n) insert	O(log n) extract	O(1) find max/min	Complete binary tree

---

🧩 How They Are Related

From	Related To	Why
Array	Stack, Queue	Implemented as array sometimes
Linked List	Stack, Queue	Also can be implemented via linked list
Binary Tree	AVL, Red-Black Tree	Add balancing rules on binary trees

Binary Tree	Heap	Heap is a specialized *complete* binary tree
Heap	Priority Queue	Heap is the underlying mechanism
OOP	Everything	We define nodes, trees, lists, etc., as *classes*

🎯 Applications of Each

Data Structure	Typical Applications
Array	Storing fixed-size collections, random access (e.g., image data, table data)
Linked List	Dynamic memory usage, implementation of stacks/queues
Doubly Linked List	Browser history (back/forward), undo-redo systems
Stack	Expression parsing (e.g., compilers), backtracking algorithms (e.g., DFS)
Queue	Scheduling tasks (e.g., CPU task scheduling, BFS traversal)
Binary Search Tree (BST)	Sorted data storage, range queries
AVL Tree	Systems needing *fast read-heavy operations*, databases (strict balance)
Red-Black Tree	Real-world databases (e.g., TreeMap in Java, C++ STL map/set)
Heap (Priority Queue)	Dijkstra's shortest path algorithm, CPU process scheduling, real-time simulations

---

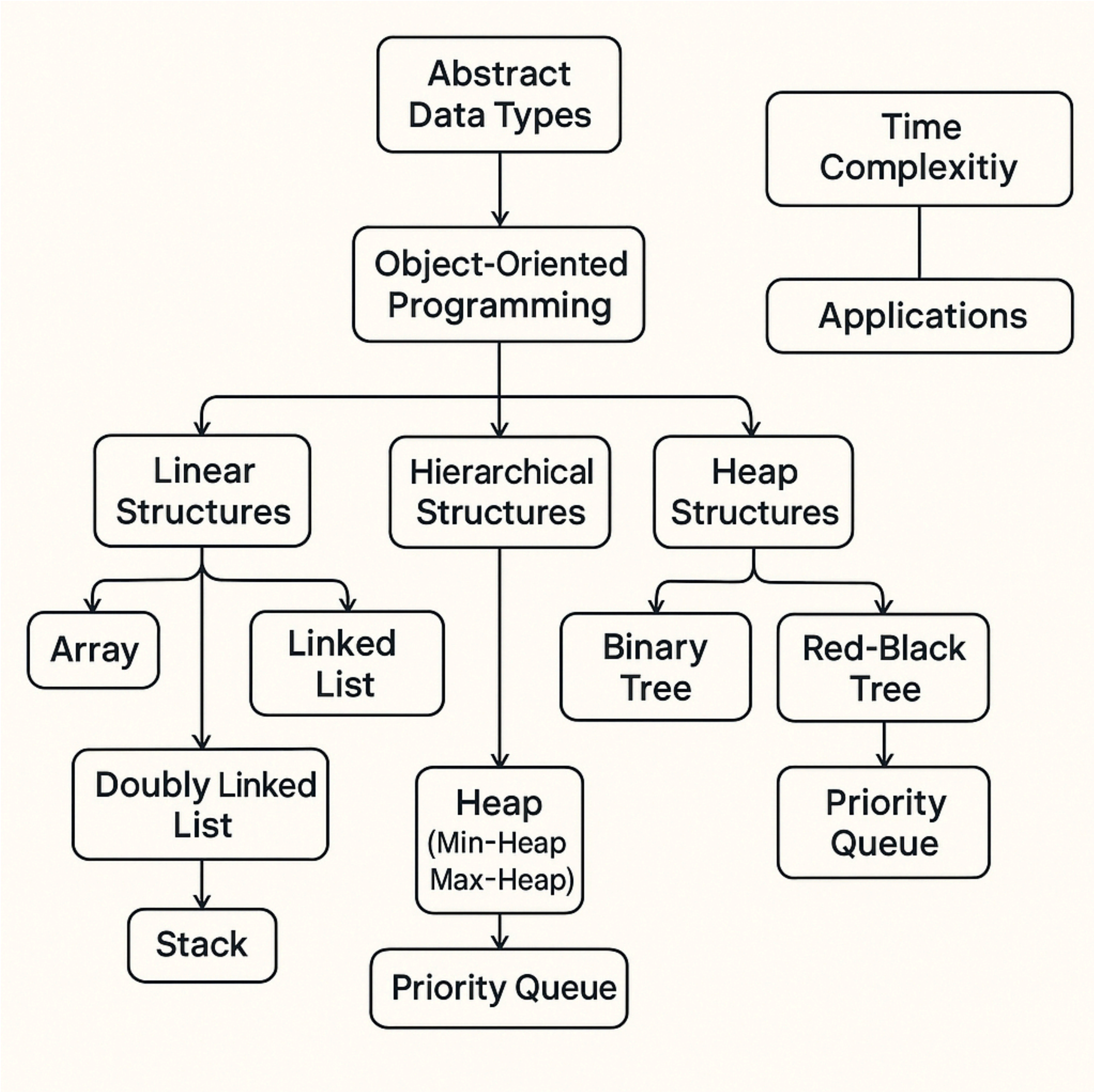
🔥 Summary in One Sentence

Arrays and **Linked Lists** build **Stacks** and **Queues**, while **Binary Trees** evolve into **AVL Trees** and **Red-Black Trees** for fast search, and **Heaps** build **Priority Queues** for urgent tasks — all managed and organized through **OOP**.

📈 Tip for Deep Understanding

If you imagine:

- **Array/LinkedList** = "Simple collection"
- **Stack/Queue** = "Special usage rules on collections"
- **Binary Tree** = "Hierarchy for searching"
- **Heap** = "Hierarchy for urgency (min/max priority)"
- **AVL/RB Tree** = "Hierarchy for speed (balance search tree)"



Addition

🧩 What is an Associative Array?

- Also called: **Map**, **Symbol Table**, **Dictionary**.
- Instead of **indexing by integer** (like arrays), you **index by *key***.
- You store **(key, value)** pairs.

⚡ Why is it fast?

- A **normal array** takes **O(n)** time to search (`linear scan`).
- **Associative arrays (dictionaries)** achieve **O(1) average time** for **search, insert, delete** because:
  - They use a data structure called a **hash table** underneath.
  - **Hashing** converts the key into a *fixed-size index* where the value is stored.
  - So it directly jumps to the memory spot without scanning everything.

🧠 Basic Time Complexity

Operation	Array	Linked List	Associative Array (Hash Table)
Search	O(n)	O(n)	O(1) (average)
Insert	O(1) at end, O(n) at middle	O(1) at head	O(1) (average)
Delete	O(n)	O(n)	O(1) (average)

⚠️ *But* in the **worst-case** (many collisions), hash table operations degrade to **O(n)**.

🏛️ How Hash Tables Work (Behind the Scenes)

Step	Description
1. Hash Function	Converts the key (like "apple") into an integer (hash code).
2. Modulo Operation	Use `hash(key) % table_size` to find the index.
3. Handle Collisions	If two keys hash to same index, use: chaining (linked list) or open addressing (probing).