Arrays

📌 **Definition**: A fixed-size collection of elements stored in contiguous memory.
⏳ **Time Complexity**:

- **Access**: O(1) ✅
- **Search**: O(n) ❌
- **Insert/Delete**: O(n) ❌

🔹 **When to Use?**
✔️ Fast random access (e.g., `arr[i]`)
✔️ Storing ordered data
✔️ Small datasets

🔹 **Optimized Trick**:

- **Use HashMaps** for O(1) lookups instead of searching in an array.
- **Sort first** before binary search (O(log n) instead of O(n)).

2️⃣ Linked List

📌 **Definition**: A dynamic data structure where each element (node) contains a value and a pointer to the next node.
⏳ **Time Complexity**:

- **Access**: O(n) ❌
- **Search**: O(n) ❌
- **Insert/Delete**: O(1) ✅

🔹 **When to Use?**
✔️ Fast insert/delete in dynamic lists (no shifting needed like arrays)
✔️ Implementing stacks & queues

🔹 **Optimized Trick**:

- **Use a dummy node** for easy insertions.
- **Use slow & fast pointers** to detect cycles in O(n).

python
CopierModifier

```python
# Detect cycle in Linked List
def hasCycle(head):
    slow, fast = head, head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
```

```
        if slow == fast:
            return True
    return False
```

3️⃣HashMap / Dictionary

📌 **Definition**: Stores key-value pairs with O(1) lookup using a hash function.
⏳ **Time Complexity**:

- **Access/Search/Insert/Delete**: O(1) ✅

🔹 **When to Use?**
✔️ Fast lookups (e.g., counting frequencies, caching, two sum)
✔️ Removing duplicates

🔹 **Optimized Trick**:

- **Use `defaultdict` from `collections` for easier key handling.**
- **Avoid collisions by choosing a good hash function.**

python
CopierModifier

```
from collections import defaultdict
freq = defaultdict(int)
for num in [1, 2, 2, 3]:
    freq[num] += 1
```

4️⃣Stack

📌 **Definition**: LIFO (Last In, First Out) structure.
⏳ **Time Complexity**:

- **Push/Pop**: O(1) ✅
- **Search**: O(n) ❌

🔹 **When to Use?**
✔️ Backtracking (e.g., DFS, parentheses validation)
✔️ Function call stack

🔹 **Optimized Trick**:

- **Use a stack for efficient parenthesis matching or reversing strings.**

python

CopierModifier

```
# Valid Parentheses using Stack
def isValid(s):
    stack, mapping = [], {')': '(', '}': '{', ']': '['}
    for char in s:
        if char in mapping:
            if not stack or stack.pop() != mapping[char]: return 
False
        else:
            stack.append(char)
    return not stack
```

5️⃣ Queue (FIFO - First In, First Out)

📌 **Definition**: Elements are added at the end and removed from the front.
⏳ **Time Complexity**:

- **Enqueue/Dequeue**: O(1) ✅

🔹 **When to Use?**
✔️ BFS (Breadth-First Search)
✔️ Scheduling (CPU, task processing)

🔹 **Optimized Trick**:

- **Use `collections.deque` instead of lists for O(1) pop from both ends.**

python
CopierModifier

```
from collections import deque
queue = deque([1, 2, 3])
queue.append(4)  # Enqueue
queue.popleft()  # Dequeue
```

6️⃣ Priority Queue (Min/Max Heap - Binary Heap)

📌 **Definition**: A queue where elements are processed based on priority.
⏳ **Time Complexity**:

- **Insert**: O(log n) ✅
- **Pop (Get Min/Max)**: O(log n) ✅

🔹 **When to Use?**
✔️ Dijkstra's shortest path algorithm
✔️ Scheduling

🔹 **Optimized Trick**:

- **Use heapq for an efficient priority queue in Python.**

python
CopierModifier
```
import heapq
heap = []
heapq.heappush(heap, 3)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)
print(heapq.heappop(heap))  # Returns 1 (smallest element)
```

7️⃣ Graph (Adjacency List / Matrix)

📌 **Definition**: Represents nodes and edges (connections).
⏳ **Time Complexity**:

- **Adjacency List (Sparse Graph)**: $O(V + E)$ ✅
- **Adjacency Matrix (Dense Graph)**: $O(V^2)$ ❌

🔹 **When to Use?**
✔️ **Adjacency List** for sparse graphs
✔️ **Adjacency Matrix** for dense graphs

🔹 **Optimized Trick**:

- **Use BFS/DFS for traversal.**
- **Use Dijkstra's algorithm for shortest path problems.**

python
CopierModifier
```
# BFS in Graph
from collections import deque
graph = {0: [1, 2], 1: [2, 3], 2: [3], 3: []}
def bfs(start):
    queue, visited = deque([start]), set([start])
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
```

```
                visited.add(neighbor)
                queue.append(neighbor)
bfs(0)
```

🔥 Data Structure Optimization Cheatsheet

Data Structure	Best Use Case	Optimization Tricks
Array	Random access, small data	Use **sorting** for faster searches
Linked List	Insert/Delete frequently	Use **dummy nodes** to simplify insertions
HashMap	Fast lookups	Use `defaultdict` to handle missing keys
Stack	Backtracking (DFS, parenthesis matching)	Use list as stack for **O(1) push/pop**
Queue	FIFO processing (BFS, scheduling)	Use `deque` instead of list for **O(1) dequeuing**
Heap	Priority-based retrieval	Use `heapq` for **O(log n) insertions**
Graph	Network connections (BFS, DFS)	Use **Adjacency List** for space efficiency

---

🔥 Final Tips

✅ **Know when to use each data structure**
✅ **Use HashMaps for O(1) lookups instead of searching arrays**
✅ **Use sorting + binary search instead of linear search**
✅ **Use `deque` instead of lists for O(1) queue operations**
✅ **Use Heaps (Priority Queue) for quick min/max retrieval**