🧩 Course 2 — Data Structures with Python

Week 3: Stacks & Queues

🎯 Learning Objectives

By the end of this week, students will be able to:

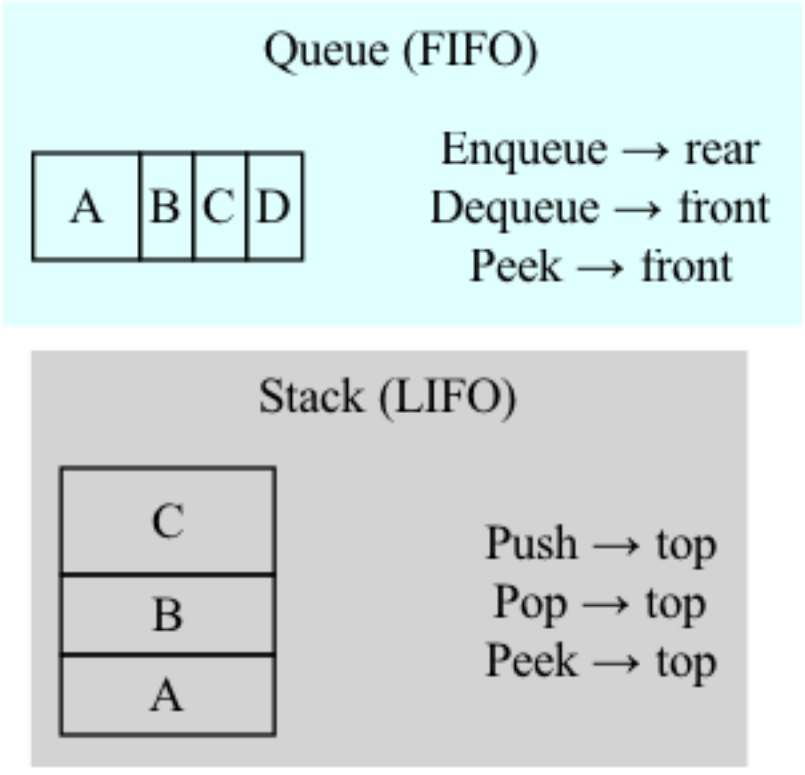
- ✅ Understand the principles of **LIFO (Stack)** and **FIFO (Queue)** data structures.
- ✅ Implement stacks and queues using **Python lists** and the **collections.deque** module.
- ✅ Apply these data structures to solve real-world computational problems.
- ✅ Analyze the **time complexity** of stack and queue operations.
- ✅ Recognize common pitfalls and best practices in usage.

📘 1. Introduction: LIFO and FIFO Concepts

A **Stack** follows the principle of **Last In, First Out (LIFO)** — the last element added is the first one removed.
A **Queue** follows **First In, First Out (FIFO)** — the first element added is the first one processed.

Real-world analogy:

- **Stack:** A pile of plates — you remove the top one first.
- **Queue:** A line of people — the first person in line is served first.



Stack vs Queue Analogy

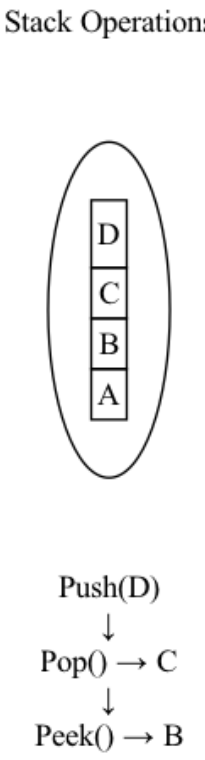
---

2. Stack Implementation in Python

```
# Using a Python list as a Stack
stack = []

# Push operation
stack.append('A')
stack.append('B')
stack.append('C')

# Pop operation
print(stack.pop())  # Removes 'C'
print(stack)
```

Time Complexity

Operation	Description	Time Complexity
append()	`Push Element`	O(1)
pop()	`Remove Top element`	O(1)
peek()	`Access Top Element`	O(1)



Stack Operation Diagram

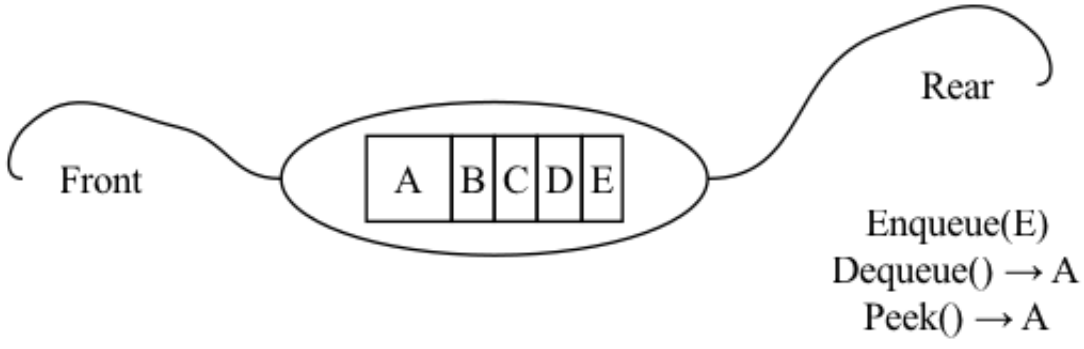
🌀 3. Queue Implementation in Python

Python lists are inefficient for queues since `pop(0)` is O(n).
We use **`collections.deque`**, a double-ended queue.

```
from collections import deque

queue = deque()

queue.append("Alice")
queue.append("Bob")
queue.append("Charlie")

print(queue.popleft())  # Alice
print(queue)
```

Time Complexity

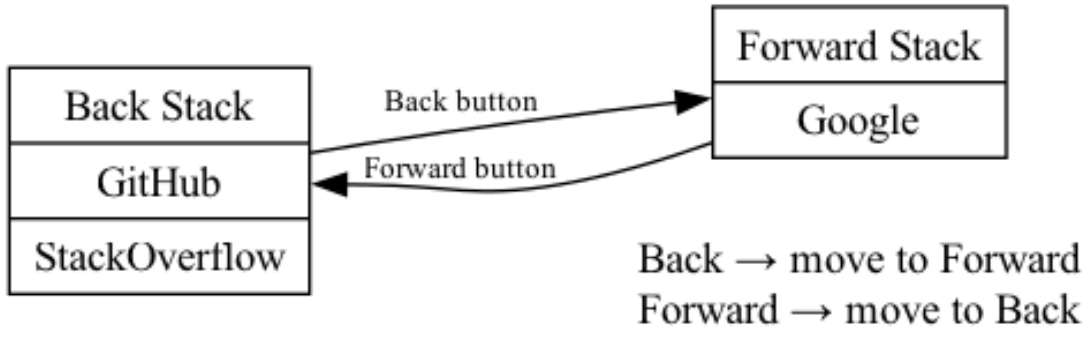
Operation	Description	Time Complexity
append()	`Enqueue`	O(1)
popleft()	`Dequeue`	O(1)
peek()	`Access Front`	O(1)



Queue Operation Diagram

🧩 4. Practical Example — Browser History (Stack)

```
back_stack = []
forward_stack = []

def visit_page(url):
    back_stack.append(url)
    print(f"Visited: {url}")

def go_back():
    if back_stack:
        page = back_stack.pop()
        forward_stack.append(page)
        print(f"Back to: {back_stack[-1] if back_stack else 'Home'}")

visit_page("google.com")
visit_page("github.com")
go_back()
```



Browser History Stack Diagram

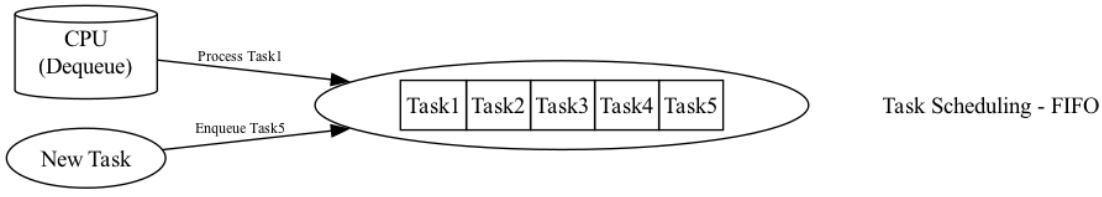
⚙️ 5. Real-World Example — Task Scheduling (Queue)

```
from collections import deque

tasks = deque(["task1", "task2", "task3"])

while tasks:
```

```
    current = tasks.popleft()
    print("Processing:", current)
```

Output:

```
Processing: task1
Processing: task2
Processing: task3
```



Task Queue Diagram

In-Class Exercises

Exercise 1 — Reverse String Using Stack

Write a function `reverse_string(s)` that reverses a string using a stack.

Exercise 2 — Queue Simulation

Simulate a customer service line where each customer is dequeued and served.

Exercise 3 — Palindrome Checker

Use a stack to check if a word is a palindrome.

🏠 Take-Home Assignments

Assignment 1 — Stack Class

Implement a `Stack` class with methods: `push()`, `pop()`, `peek()`, and `is_empty()`.

Assignment 2 — Queue Class

Implement a `Queue` class using `deque`. Include methods: `enqueue()`, `dequeue()`, and `display()`.

Assignment 3 — Undo Feature Simulation

Simulate a text editor "Undo" operation using stacks.

❌ Common Mistakes & 💡 Best Practices

Common Mistakes

- ❌ Using `pop(0)` on a list as a queue (inefficient).
- ❌ Forgetting to check for underflow (popping empty stack).
- ❌ Misusing stack and queue interchangeably.

Best Practices

- ✅ Use `deque` for both stacks and queues (efficient O(1) ops).
- ✅ Visualize stack/queue operations before implementing.
- ✅ Handle edge cases (empty structure).

🔭 Next Week Preview

In **Week 4**, we'll explore:

1. **Linked Lists** — Node-based memory management
2. **Singly vs Doubly Linked Lists**
3. **Insertion and Deletion operations**
4. **Diagram-based visualization of links**
