Stacks & Queues Cheat Sheet (Theory Only)

---

1. Stack (Abstract Data Type - ADT)

A **stack** is a **linear data structure** that follows **LIFO (Last In, First Out)** principle.

Operations on a Stack:

- **Push(x):** Adds an element x to the top of the stack.
- **Pop():** Removes the top element from the stack.
- **Peek()/Top():** Returns the top element without removing it.
- **isEmpty():** Checks if the stack is empty.

Example Implementation in Python:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        return "Stack is empty"

    def peek(self):
        return self.stack[-1] if not self.is_empty() else "Stack is empty"

    def is_empty(self):
        return len(self.stack) == 0

# Example Usage
s = Stack()
s.push(10)
s.push(20)
print(s.pop())  # Output: 20
print(s.peek()) # Output: 10
```

---

2. Applications of Stack

(A) Expression Conversion & Evaluation

Infix Expression → Operators between operands (e.g., A + B).
Postfix Expression → Operators after operands (e.g., A B +).
Prefix Expression → Operators before operands (e.g., + A B).
Algorithm for Expression Conversion (Infix to Postfix using Stack)

- **Operands are added directly to the output.**
- **Operators follow precedence and associativity rules.**
- **Parentheses control precedence.**

Example: Convert (A + B) * C to Postfix

- Infix: (A + B) * C
- Postfix: A B + C *

Evaluating a Postfix Expression using Stack

```
def evaluate_postfix(expression):
    stack = []
```

```
    for char in expression:
        if char.isdigit():
            stack.append(int(char))
        else:
            b = stack.pop()
            a = stack.pop()
            if char == '+':
                stack.append(a + b)
            elif char == '-':
                stack.append(a - b)
            elif char == '*':
                stack.append(a * b)
            elif char == '/':
                stack.append(a / b)
    return stack.pop()

print(evaluate_postfix("23*5+"))  # Output: 11
```

---

3. Queue (Abstract Data Type - ADT)

A **queue** is a **linear data structure** that follows **FIFO (First In, First Out)** principle.

Operations on a Queue:

- **Enqueue(x):** Adds an element x to the rear of the queue.
- **Dequeue():** Removes the front element.
- **Front():** Returns the front element without removing it.
- **isEmpty():** Checks if the queue is empty.

Example Implementation in Python:

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        return "Queue is empty"

    def front(self):
        return self.queue[0] if not self.is_empty() else "Queue is empty"

    def is_empty(self):
        return len(self.queue) == 0

# Example Usage
q = Queue()
q.enqueue(10)
q.enqueue(20)
print(q.dequeue())  # Output: 10
print(q.front())    # Output: 20
```

---

4. Types of Queues

(A) Simple Queue

Follows **FIFO order** (Elements inserted at the rear and removed from the front).
Example: **Job Scheduling, Printer Queue.**

(B) Circular Queue

The last position is connected to the first to **utilize empty spaces** efficiently.
Overcomes the issue of **wasted space in a simple queue**.
Example: **CPU Process Scheduling.**
Example Implementation:

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = self.rear = -1

    def enqueue(self, item):
        if (self.rear + 1) % self.size == self.front:
            return "Queue is full"
        elif self.front == -1:
            self.front = self.rear = 0
        else:
            self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = item

    def dequeue(self):
        if self.front == -1:
            return "Queue is empty"
        elif self.front == self.rear:
            val = self.queue[self.front]
            self.front = self.rear = -1
            return val
        else:
            val = self.queue[self.front]
            self.front = (self.front + 1) % self.size
            return val
```

(C) Priority Queue

Elements are **dequeued based on priority** instead of FIFO.
Example: **Hospital Emergency Queue, Dijkstra's Algorithm.**
Example Implementation:

```
import heapq

class PriorityQueue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item, priority):
        heapq.heappush(self.queue, (priority, item))

    def dequeue(self):
        return heapq.heappop(self.queue)[1] if self.queue else "Queue is 
empty"
```

```
pq = PriorityQueue()
pq.enqueue("Task1", 2)
pq.enqueue("Task2", 1)
pq.enqueue("Task3", 3)
print(pq.dequeue())  # Output: Task2 (Highest Priority)
```

(D) Deque (Double-Ended Queue)

Insertion & Deletion possible at both ends.

Types of Deque:

- **Input-Restricted Deque:** Deletion from both ends, but insertion at one end.
- **Output-Restricted Deque:** Insertion from both ends, but deletion at one end.

Example: **Undo/Redo operations in editors.**

Example Implementation:

```
from collections import deque

dq = deque()
dq.appendleft(10)  # Insert at front
dq.append(20)      # Insert at rear
dq.pop()           # Remove from rear
dq.popleft()       # Remove from front
```

---

5. Comparison Between Stack & Queue

Feature	Stack (LIFO)	Queue (FIFO)
Insertion	push(x) at top	enqueue(x) at rear
Deletion	pop() from top	dequeue() from front
Access	peek() returns top	front() returns first element
Usage	Function calls, Expression evaluation	Scheduling, Buffer management

---

Key Takeaways

Stack (LIFO): Used in function calls, backtracking, and expression evaluation.
Queue (FIFO): Used in scheduling, job queues, and data buffering.
Types of Queues: Simple, Circular, Priority, Deque.
Stack vs. Queue: Stacks operate on **Last In, First Out (LIFO)**, while Queues use **First In, First Out (FIFO)**.

This **Stacks & Queues Cheat Sheet** covers **ADT operations, applications (expression evaluation), types of queues, and implementations**. Let me know if you need further explanations!