Advance Educational Activities Pvt. Ltd.

Unit 4: Stacks and Queues

4.1 — Introduction to Linear ADTs: Stack & Queue

Where data learns discipline.

1 What Are Linear ADTs?

ADT = Abstract Data Type
It defines what operations are allowed, not how they are implemented.

Stacks and Queues are **restricted-linear ADTs** —data is arranged in a **straight line**, but access is controlled by **rules**.

2 Why Do We Need Restricted Linear Structures?

Imagine a system where:

- Tasks must be done in order
- Some tasks must be undone
- Some must be prioritized
- Some must wait

Raw arrays or linked lists cannot control behavior. We need **structures that enforce order**.

Enter:

- Stack → LIFO discipline
- Queue → FIFO discipline

These restrictions solve real-world problems effortlessly.

3 Stack vs Queue (Core Idea)

ADT	Access Rule	Meaning
Stack	**LIFO** (Last In, First Out)	Last item added is first removed
Queue	**FIFO** (First In, First Out)	First item added is first removed

They're like two opposite philosophies of life.

4 Real-Life Analogies

Stack = Stack of Plates

- You place plates on top
- You remove plates from top

- Last plate placed is the first taken

→ LIFO behavior

Queue = Ticket Counter Queue

- First person standing in line gets ticket first
- Last person waits longer
- No skipping allowed

→ FIFO behavior

5 Why Restrictions? (The Real Purpose)

Restrictions make data easier to:

- Manage
- Predict
- Organize
- Control

Example:

- A browser's *Back* button must follow LIFO order
- A printer queue must follow FIFO order

Without these rules, systems become chaotic.

6 Operations Preview

Stack Operations

- push() → insert at top
- pop() → remove from top
- peek() → see top
- isEmpty()
- isFull()

Errors:

- **Overflow** → when pushing into full array stack
- **Underflow** → when popping empty stack

Queue Operations

- enqueue() → insert at rear
- dequeue() → remove from front
- front() / peek()
- isEmpty()
- isFull()

Errors:

- **Overflow** → queue full
- **Underflow** → queue empty

7 ADT Representation (Not Code)

Stack ADT

- push(x)
- pop()
- peek()
- isEmpty()

Queue ADT

- enqueue(x)
- dequeue()
- front()
- isEmpty()

Operations define the ADT —implementation comes later.

8 Usage Preview (Why They Matter)

Stack Applications

- Expression evaluation
- Undo/Redo
- Backtracking
- Function call management
- Reversal problems

Queue Applications

- CPU scheduling
- Network packet handling
- Printer queue
- BFS (Breadth First Search)
- Customer service systems

These two ADTs appear everywhere in Computer Science.

9 Stack vs Queue: Visual Intuition

Stack (LIFO)

```
    TOP
     ↓
   [40]
   [30]
   [20]
   [10]
  bottom
```

Queue (FIFO)

```
front → [10] [20] [30] [40] ← rear
```

The entire behavior is about **where you insert** and **from where you remove**.

🔟 Implementation Preview

In this unit, we will implement:

Stacks

- Using Array
- Using Linked List
- Applications:
  - Infix → Postfix
  - Infix → Prefix
  - Postfix Evaluation
  - Balanced Parentheses

Queues

- Using Array
- Using Linked List
- Circular Queue
- Priority Queue (Intro)
- Deque (Double-Ended Queue)

Quick Summary

- Stack → LIFO
- Queue → FIFO
- Both are **Linear ADTs**
- Stacks useful for undo, recursion, evaluation
- Queues useful for scheduling, buffering, BFS

Mantra:

"Stacks remember the past. Queues decide the future." Together, they govern the flow of data.

4.2 — Stack ADT (Concept & Operations)

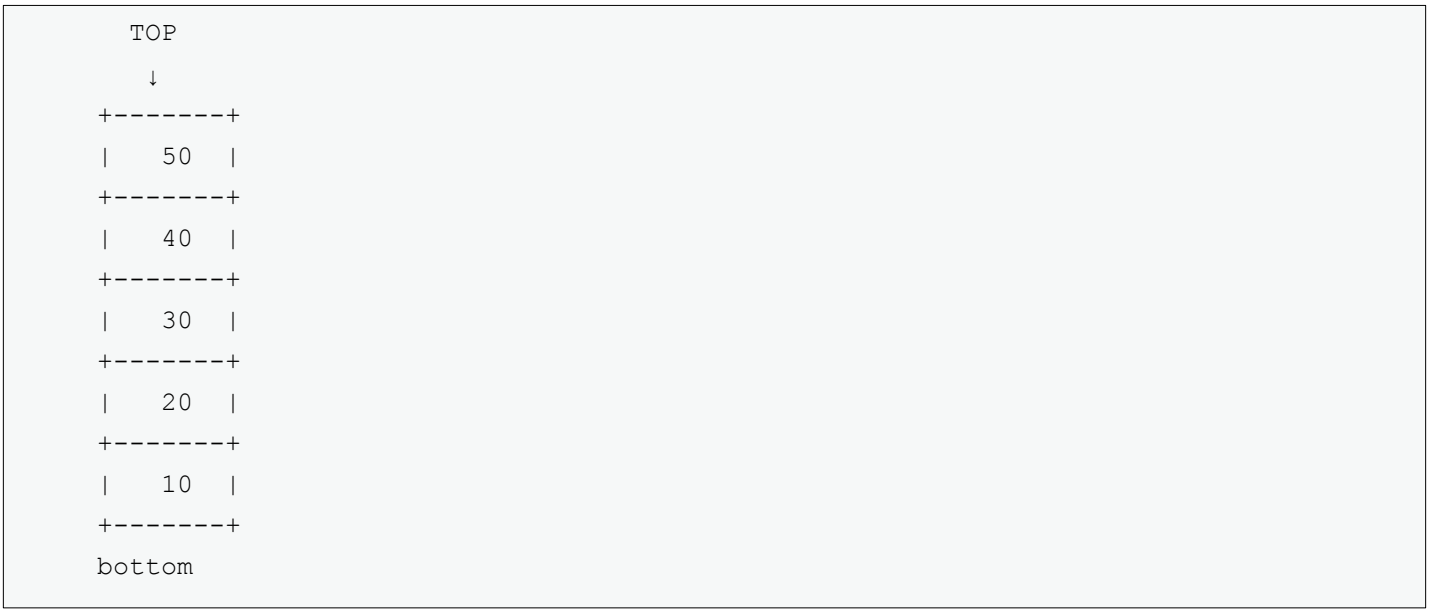
When data learns to behave.

1️ What Is a Stack?

Definition:
A Stack is a linear ADT that follows the LIFO (Last In, First Out) principle.

- The **last** element inserted is the **first** element removed.
- Only **one end** is used for insertion and deletion → **the TOP**.

2 Visual Intuition



We add and remove things only from the top, like a stack of plates.

3 Why Stacks? (The Purpose)

Stacks naturally model tasks like:

- Undo/Redo
- Function call stack
- Expression evaluation
- Parenthesis matching
- Backtracking (mazes, DFS, recursion)Wherever **reverse order processing** is needed — Stack is the hero.

4 Stack Terminology

Term	Meaning
push(x)	Insert element **x** at the top
pop()	Remove and return top element
peek()	Return top element without removing
isEmpty()	Checks if stack has no elements
isFull()	Relevant only for array-based stacks
top	Pointer/index of current top element

5 Stack ADT (Abstract Definition)

This is the **logical definition** independent of implementation.

- createStack()
- push(x)
- pop()

- peek()
- isEmpty()
- isFull() // only if using static array

These operations define what the Stack can do.

6 Characteristics of a Stack

- Single access point: **TOP**
- Insertion and deletion from the **same end**
- **Restricted** linear structure
- Follows a strict order: **LIFO**
- Very predictable and easy to reason about

7 Stack Overflow & Underflow

⚠️ Stack Overflow

Occurs when:

- We try to **push** into a **full** array-based stack.

```
if (top == size-1)
    Stack Overflow
```

⚠️ Stack Underflow

Occurs when:

- We try to **pop** from an **empty** stack.

```
if (top == -1)
    Stack Underflow
```

Overflow = no more space; Underflow = no more elements.

8 Stack Time Complexity

Operation	Time
push()	O(1)
pop()	O(1)
peek()	O(1)
isEmpty()	O(1)
isFull()	O(1)

Stacks are extremely efficient — constant time for all operations.

9 Stack Behavior in Memory

Using Array

- Stack grows upward (increasing index)
- `top` is an integer index

Using Linked List

- Stack grows by adding new nodes at the head
- `top` is a pointer to the first node

We'll implement both in **4.3 and 4.4**.

10 Internal Working Example (Conceptual)

Let's say we perform:

- push(10)
- push(20)
- push(30)
- pop()
- push(40)

State of Stack:

```
TOP
 ↓
 40
 20
 10
```

Explanation:

- 10, 20, 30 pushed
- 30 popped
- 40 added

11 Real-World Applications of Stack (Preview)

Application	Why Stack?
Undo/Redo	LIFO history
Backtracking	Last move undone first
Function Calls	Each call returns in reverse order
Expression Evaluation	Postfix, Prefix
Syntax Checking	Parenthesis balancing
Browser Back Button	Last visited page comes first

We will deeply explore these in **Unit 4.5**.

Quick Summary

- Stack = **LIFO ADT**
- Operates at **one end: TOP**
- Operations: `push`, `pop`, `peek`, `isEmpty`
- Overflow → pushing on full array stack
- Underflow → popping on empty stack
- All operations run in **O(1)**
- Perfect for reverse order, undo, recursion, expression problems

Mantra:

"Stacks are the memory of actions — they remember what happened last."

4.3 — Stack Implementation Using Array

Static memory. Fixed size. Maximum speed.

1 Why Array Implementation?

- Simple
- Fast
- Efficient
- Predictable
- Uses contiguous memory

This is the most basic and common implementation for Stacks.

Perfect for:

- Small, fixed-size stacks
- Expression evaluation
- Function call simulation
- Competitive programming

2 Conceptual Structure

```
Index:   0   1   2   3   4
Stack:  [ ] [ ] [ ] [ ] [ ]
                 ↑
               TOP
```

- `top` stores the index of the **current top element**
- Initially, `top = -1` (stack is empty)

3 Stack Operations Using Array

We implement:

- **push(x)** → insert at top
- **pop()** → remove from top

- **peek()** → view top element
- isEmpty()
- isFull()

Let's write this cleanly in Java.

4️⃣ Full Java Implementation

```
class Stack {
    int size;
    int top;
    int[] arr;

    // Constructor
    Stack(int size) {
        this.size = size;
        arr = new int[size];
        top = -1; // initially empty
    }

    // Push operation
    void push(int x) {
        if (isFull()) {
            System.out.println("Stack Overflow");
            return;
        }
        arr[++top] = x;
    }

    // Pop operation
    int pop() {
        if (isEmpty()) {
            System.out.println("Stack Underflow");
            return -1;
        }
        return arr[top--];
    }

    // Peek operation
    int peek() {
        if (isEmpty()) {
            System.out.println("Stack is Empty");
            return -1;
        }
        return arr[top];
    }
```

```
// Check empty
    boolean isEmpty() {
        return top == -1;
    }

    // Check full
    boolean isFull() {
        return top == size - 1;
    }

    // Display stack
    void display() {
        if (isEmpty()) {
            System.out.println("Stack is Empty");
            return;
        }
        System.out.print("Stack elements: ");
        for (int i = top; i >= 0; i--) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

5 How It Works Internally (Dry Run)

Let's perform this sequence:

- push(10)
- push(20)
- push(30)
- pop()
- push(40)

Step-by-Step Table:

Operation	top	Array State
Init	-1	[_ _ _ _ _]
push(10)	0	[10 _ _ _ _]
push(20)	1	[10 20 _ _ _]
push(30)	2	[10 20 30 _ _]
pop() → 30	1	[10 20 _ _ _]
push(40)	2	[10 20 40 _ _]

Stack (Top to Bottom):

```
TOP → 40
      20
      10
```

6 Overflow & Underflow Explained

⚠️ Stack Overflow

Occurs when:

```
top == size - 1
```

Example:

push(50);

push(60); // if size = 2 → overflow

⚠️ Stack Underflow

Occurs when:

```
top == -1
```

Example:

```
pop(); // when stack is empty
```

Array stack must be used carefully — fixed size can break the logic if not handled.

7 Time & Space Complexity

Operation	Time	Reason
push()	O(1)	Insert at top
pop()	O(1)	Remove from top
peek()	O(1)	Access top element
isEmpty() / isFull()	O(1)	Constant-time checks

Space: **O(n)** where n = size of array

Stacks are among the fastest linear ADTs.

8 Advantages of Array-Based Stack

- Fast access (direct indexing)
- Easy to implement
- Good for small predictable sizes
- Memory-friendly (no pointer overhead)

9️⃣ Limitations

- Fixed size → cannot grow
- Overflow is possible
- Not suitable for very dynamic data
- Reallocation is expensive

This is why we introduce Linked List stack in the next unit.

Quick Summary

- Stack uses array + top index
- Initial top = **-1**
- Push increments top
- Pop decrements top
- Overflow & underflow must be handled
- All operations are **O(1)**
- Simple & efficient but not flexible (fixed size)

Mantra:

"Array Stacks are fast but rigid — perfect for tight, controlled operations."

4.4 — Stack Implementation Using Linked List

When Stack learns to grow freely.

1️⃣ Why Linked List Implementation?

Limitations of Array Stack:

- Fixed size
- Overflow occurs easily
- Wastes memory if under-utilized

Linked List Stack solves it:

- Dynamic size
- Grows/shrinks as needed
- No overflow (unless system memory is full)
- Perfect for recursion, parsing, dynamic expression evaluation

It gives Stack the flexibility Linked Lists are famous for.

2️⃣ Conceptual Structure

In Linked List stack:

- New elements are added at the **head**
- Removed from the **head**
- The head node = TOP of stack

Visual

```
TOP → [40] → [30] → [20] → [10] → null
```

This mimics push/pop naturally.

3 Node Structure

```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        next = null;
    }
}
```

4 Stack Class Using Linked List

```
class LinkedListStack {
    Node top; // top of the stack

    LinkedListStack() {
        top = null;
    }

    // Push operation
    void push(int x) {
        Node newNode = new Node(x);
        newNode.next = top;
        top = newNode;
    }

    // Pop operation
    int pop() {
        if (isEmpty()) {
            System.out.println("Stack Underflow");
            return -1;
        }
        int value = top.data;
        top = top.next;
        return value;
    }
```

```
// Peek operation
    int peek() {
        if (isEmpty()) {
            System.out.println("Stack is Empty");
            return -1;
        }
        return top.data;
    }

    // Check if empty
    boolean isEmpty() {
        return top == null;
    }

    // Display stack
    void display() {
        if (isEmpty()) {
            System.out.println("Stack is Empty");
            return;
        }
        Node temp = top;
        System.out.print("Stack: ");
        while (temp != null) {
            System.out.print(temp.data + " → ");
            temp = temp.next;
        }
        System.out.println("null");
    }
}
```

5 How Push Works (Visual)

Operation:

```
push(50)
```

Before:

```
TOP → [40] → [30] → [20] → null
```

After:

```
TOP → [50] → [40] → [30] → [20] → null
```

Just one pointer update → O(1)

6 How Pop Works (Visual)

Operation:

```
pop()
```

Before:

```
TOP → [50] → [40] → [30] → ...
```

After:

```
TOP → [40] → [30] → [20] → null
```

Removed value = 50

Again → O(1)

7 Dry Run Example

Sequence:

```
push(10)
push(20)
push(30)
pop()
push(40)
peek()
```

State after operations:

```
TOP → 40 → 20 → 10 → null
```

Output:

- popped: **30**
- peek: **40**

8 Time & Space Complexity

Operation	Time	Space
push()	O(1)	O(1)
pop()	O(1)	O(1)
peek()	O(1)	O(1)
overall space	—	O(n) for n nodes

Perfect constant-time operations, just like array stack — but dynamic.

9 Advantages of Linked List Stack

- Dynamic size
- No overflow

- Efficient O(1) operations
- Memory used only when needed
- Ideal for recursion-heavy systems

🔟 Limitations

- Slight overhead of pointer memory
- More complex than array implementation
- Slightly slower due to pointer dereferencing
- Not cache friendly

But advantages outweigh limitations in most dynamic scenarios.

Quick Summary

- Stack implemented using **Linked List**
- Push = insert at head
- Pop = remove from head
- All operations remain **O(1)**
- No overflow → dynamic growth
- Perfect for expression evaluation & recursion logic

Mantra:

"Linked List Stack grows as you grow — unlimited, free, and dynamic."

4.5 — Applications of Stack

Where theory becomes real power.

Overview of Applications

Stacks are used in:

1. Expression Processing
    - Infix → Postfix
    - Infix → Prefix
    - Postfix Evaluation
2. Parentheses Balancing
3. Function Call Handling (Call Stack)
4. Undo/Redo
5. Backtracking
6. Reversing
7. Parsing
8. Memory Management

We'll focus on the *core DSA applications* first: Infix, Postfix, Prefix, Evaluation, Parentheses

1 Parentheses Balancing (Fundamental Problem)

Problem:

Check if expressions like:

```
(a+b)
(a+(b*c))
[()]{}{[()()]()}
```

are balanced.

Logic:

- Push opening bracket
- Pop when matching closing bracket
- If mismatch or empty stack → unbalanced
- If stack empty at end → balanced

Code:

```
boolean isBalanced(String s) {
    Stack<Character> st = new Stack<>();

    for (char ch : s.toCharArray()) {
        if (ch == '(' || ch == '{' || ch == '[') {
            st.push(ch);
        } else if (ch == ')' || ch == '}' || ch == ']') {
            if (st.isEmpty()) return false;
            char top = st.pop();
            if (!isMatch(top, ch)) return false;
        }
    }
    return st.isEmpty();
}

boolean isMatch(char open, char close) {
    return (open=='(' && close==')') ||
           (open=='{' && close==' }') ||
           (open=='[' && close==']');
}
```

2 Infix → Postfix Conversion (Shunting Yard Logic)

What is Postfix?

No parentheses, no precedence ambiguity.

Example:

Infix: A + B * C

Postfix: ABC*+

Rules:

- Operand → output
- '(' → push
- ')' → pop until '('
- Operator → pop higher/equal precedence operators
- End → pop remaining operators

Precedence Table

Operator	Precedence
^	Highest
*, /	Middle
+, -	Lowest

Code:

```
String infixToPostfix(String exp) {
    Stack<Character> st = new Stack<>();
    StringBuilder out = new StringBuilder();

    for (char ch : exp.toCharArray()) {

        if (Character.isLetterOrDigit(ch)) {
            out.append(ch);
        }
        else if (ch == '(') {
            st.push(ch);
        }
        else if (ch == ')') {
            while (!st.isEmpty() && st.peek() != '(')
                out.append(st.pop());
            st.pop(); // remove '('
        }
        else { // operator
            while (!st.isEmpty() && precedence(st.peek()) >= precedence(ch))
                out.append(st.pop());
            st.push(ch);
        }
    }

    while (!st.isEmpty())
        out.append(st.pop());

    return out.toString();
}
```

```
int precedence(char ch) {
    switch(ch) {
        case '+':
        case '-': return 1;
        case '*':
        case '/': return 2;
        case '^': return 3;
    }
    return -1;
}
```

3 Infix → Prefix Conversion

Logic:

Convert infix → prefix using reverse + postfix method.

Steps:

1. Reverse infix string
2. Swap '(' and ')'
3. Convert to postfix
4. Reverse the postfix → prefix

Example:

Infix: A + B * C

Reversed: C * B + A

Postfix: C B * A +

Prefix: + A * B C

Prefix: operator BEFORE operands.

4 Postfix Evaluation

Postfix Example:

Postfix: 23*54*+

Meaning: (2*3) + (5*4)

Output: 22

Logic:

- Operand → push
- Operator → pop two values, apply operation, push result

Code:

```
int evaluatePostfix(String exp) {
    Stack<Integer> st = new Stack<>();
```

```
for (char ch : exp.toCharArray()) {

        if (Character.isDigit(ch)) {
            st.push(ch - '0');
        }
        else { // operator
            int b = st.pop();
            int a = st.pop();
            int res = apply(a, b, ch);
            st.push(res);
        }
    }
    return st.pop();
}

int apply(int a, int b, char op) {
    switch(op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
    return 0;
}
```

5 Prefix Evaluation

Logic:

Same as postfix but traverse **right to left**.

Steps:

- Traverse from right
- Operand → push
- Operator → pop two → apply → push

Prefix: operator FIRST, so evaluation from right.

6 Function Call Stack (Real CS Application)

Every method call in Java uses a Stack Frame:

- Local variables
- Parameters
- Return address

Recursive calls go deeper and deeper into stack memory.

Example:

```
f(3)
 → f(2)
   → f(1)
     → f(0)
```

When returning:

f(0) returns

f(1) returns

f(2) returns

f(3) returns

This is literal **LIFO behavior**.

7 Undo / Redo (Double Stack Logic)

- Each action → push to Undo stack
- Undo → pop from Undo, push to Redo
- Redo → pop from Redo, push to Undo

Applications:

- Text editors
- Photoshop
- VS Code
- Browser tab history

8 Backtracking Applications

Used in:

- Maze solving
- Rat in the Maze
- N-Queens
- DFS
- Sudoku solving

Logic:

- Move forward → push path
- If dead end → pop → backtrack

Stack drives all "try → fail → undo" problems.

9 Reversing (Strings, Linked Lists)

Use stack to reverse:

- A string
- A linked list
- An array

Because stack gives reverse order extraction.

Quick Summary

- Stack excels in reverse order processing
- Expression conversion → heavy stack use
- Expression evaluation → postfix & prefix
- Balanced parentheses → classic stack pattern
- Undo/Redo → dual stack structure
- Recursion → actual system stack
- Backtracking → stack simulation
- Reversal → natural LIFO use

Mantra:

"Stacks are the brain of calculations and the heart of recursion."

4.6 — Queue ADT (Concept & Operations)

First come. First served. Always.

1 What Is a Queue?

Definition:
A Queue is a linear ADT that follows the FIFO — First In, First Out rule.

- The first element inserted is the first element removed
- Insert at **REAR**
- Remove from **FRONT**

2 Real-Life Analogy (Perfect Fit)

Queue at a ticket counter

- First person in line gets ticket first
- Everyone else waits
- No one jumps the line

Bus boarding queue

- People board in the order they arrive
- Newcomers join at the rear
- Front person goes first

Queues represent fairness and order.

3 Visual Structure

```
front → 10  20  30  40 ← rear
```

- `front` points to the **first** element
- `rear` points to the **last** element

- FIFO flow ← very important

4 Queue Terminology

Operation	Meaning
enqueue(x)	Insert element **x** at rear
dequeue()	Remove element from front
front() / peek()	View first element
isEmpty()	Check if queue has no elements
isFull()	Only for array queue
front & rear	Pointers/indices tracking ends

5 Queue ADT (Definition Only)

- createQueue()
- enqueue(x)
- dequeue()
- front()
- isEmpty()
- isFull() // only for array implementation

These functions define the queue conceptually. Implementation comes in upcoming units.

6 Queue Characteristics

- Ordered, linear structure
- Strict discipline: first inserted → first removed
- Uses two distinct ends
  - Front → deletion
  - Rear → insertion
- Can be implemented using **array**, **linked list**, **circular array**, **deque**, **priority queue**

7 Queue Operations (Detailed)

1. enqueue(x)

Insert element at **rear**

```
rear++
arr[rear] = x
```

2. dequeue()

Remove element from **front**

```
x = arr[front]
front++
```

3. front() / peek()

Return element at front.

4. isEmpty()

```
front > rear
```

5. isFull()

```
rear == size - 1
```

(This rule changes in circular queue.)

8 Types of Queues (Preview)

In upcoming subunits we cover:

1. Simple Queue - Basic FIFO model.
2. Circular Queue - Fixes the wasted-space problem of simple queue.
3. Priority Queue - Elements removed based on priority, not position.
4. Deque (Double-Ended Queue)
   - Insert/delete at both ends.
   - Queue family is HUGE — each variant solves a real-world constraint.

9 Queue in Memory (Intuition)

Using Array:

```
Index:  0   1   2   3   4   5
Queue: [ ] [ ] [ ] [ ] [ ] [ ]
front → 0
rear → -1
```

Using Linked List:

```
front → [10] → [20] → [30] → null ← rear
```

Linked list queues are dynamic and avoid overflow.

10 Queue Overflow & Underflow

⚠️ Overflow (in array queue)

Occurs when:

```
rear == size - 1
```

⚠️ Underflow

Occurs when:

```
front > rear
```

We'll fix overflow using **Circular Queue** in 4.9.

11 Time Complexity

Operation	Time
enqueue()	O(1)
dequeue()	O(1)
front()	O(1)
isEmpty()	O(1)

Queue is extremely efficient — no shifting needed in linked list implementation.
(But array queue *does* require shifting → covered in 4.7.)

12 Queue Applications (Preview)

Application	Why Queue?
CPU Scheduling (FCFS)	Serve requests in order
Printer Queue	First print request → first served
OS Process Management	Jobs handled fairly
Network Routers	Packet buffering
Customer Service Systems	People wait in line
BFS (Graph)	Level order traversal
Cache Management	FIFO removal

Queue is essential for *fairness and ordering*.

Quick Summary

- Queue = **FIFO ADT**
- Insert at **rear**, remove at **front**
- Operations: enqueue, dequeue, front, isEmpty
- Queue appears everywhere in OS and networks
- Forms base for **Circular Queue**, **Priority Queue**, **Deque**

Mantra:

"Queues teach data to wait — and move only when their turn comes."

4.7 — Queue Implementation Using Array

Simple. Straightforward. But with a flaw.

1 Why Array-Based Queue?

- Easy to implement
- Fast O(1) operations

- Suitable for fixed-size queue
- Common in low-level systems with known limits (printer queue, buffers)

But… it has a drawback we'll uncover soon.

2 Structure of Array Queue

We maintain two indices:

- **front** → points to first element
- **rear** → points to last element

Initial values:

front = 0

rear = -1

3 Visual Representation

Initially:

```
Index:  0  1  2  3  4
Queue: [ ] [ ] [ ] [ ] [ ]
front → 0
rear  → -1
```

After enqueue(10), enqueue(20), enqueue(30):

```
Index:  0   1   2   3   4
Queue: [10][20][30][ ] [ ]
front → 0
rear  → 2
```

After one dequeue():

```
Index:   0   1   2   3   4
Queue:  [10][20][30][ ][ ]
front → 1
rear  → 2
```

Notice: position 0 is now unusable → WASTED space!

We'll later fix this using Circular Queue (4.9).

4 Queue Operations Using Array

We implement:

- enqueue(x)
- dequeue()
- front()
- isEmpty()
- isFull()
- display()

5 Full Java Implementation

```
class ArrayQueue {
    int size;
    int front, rear;
    int[] arr;

    ArrayQueue(int size) {
        this.size = size;
        arr = new int[size];
        front = 0;
        rear = -1;
    }

    // enqueue operation
    void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue Overflow");
            return;
        }
        arr[++rear] = x;
    }

    // dequeue operation
    int dequeue() {
        if (isEmpty()) {
            System.out.println("Queue Underflow");
            return -1;
        }
        return arr[front++];
    }

    // peek front element
    int front() {
        if (isEmpty()) {
            System.out.println("Queue is Empty");
            return -1;
        }
        return arr[front];
    }

    // check empty
    boolean isEmpty() {
        return front > rear;
    }
```

```
// check full
    boolean isFull() {
        return rear == size - 1;
    }

    // display queue
    void display() {
        if (isEmpty()) {
            System.out.println("Queue is Empty");
            return;
        }
        System.out.print("Queue: ");
        for (int i = front; i <= rear; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

6 Dry Run Example (Step by Step)

Operations:

```
enqueue(10)
enqueue(20)
enqueue(30)
dequeue()
enqueue(40)
```

Stepwise Table:

Operation	front	rear	Queue Content
Init	0	-1	[_ _ _ _ _]
enq(10)	0	0	[10 _ _ _ _]
enq(20)	0	1	[10 20 _ _ _]
enq(30)	0	2	[10 20 30 _ _]
deq() → 10	1	2	[10 20 30 _ _]
enq(40)	1	3	[10 20 30 40 _]

Queue elements from front (1) to rear (3):

20 30 40

7️⃣ Overflow & Underflow

⚠️ UNDERFLOW

```
front > rear
```

Example:

```
front=0, rear=-1 → empty
```

⚠️ OVERFLOW

```
rear == size-1
```

Even if earlier positions are empty 😭

This is the MAJOR PROBLEM.

Example:

After:

```
dequeue() // front=3, rear=4
enqueue() // overflow but arr[0..2] are empty
```

8️⃣ The BIG PROBLEM: Wasted Space

Consider size = 5:

Operations:

enqueue 5 times → full

dequeue 3 times → free space at front

enqueue again → Overflow ❌

Even though free space exists, we cannot use it.

This leads us to the **Circular Queue**, which FIXES this beautifully.

9️⃣ Time Complexity

Operation	Time
enqueue	O(1)
dequeue	O(1)
front	O(1)
isEmpty	O(1)
isFull	O(1)

🔟 Advantages of Array Queue

- Simple
- Fast
- Predictable
- Perfect for small fixed-capacity queues

1️⃣1️⃣ Limitations

- Wasted space at front
- Cannot reuse freed cells
- Fixed size
- Overflow even when space exists inside

Quick Summary

- Array queue uses `front` and `rear` indices
- FIFO: insert at rear, delete at front
- All operations O(1)
- **Major flaw:** wasted space after dequeue
- Leads to false overflow
- Fixed-size structure

Mantra:

"Array queue is simple — but simplicity comes with a flaw."

4.8 — Queue Implementation Using Linked List

Dynamic, flexible, and ideal for real-world applications.

1️⃣ Why Linked List Queue?

The array queue had problems:

- Wasted space
- Fixed size
- False overflow

Linked List Queue fixes everything:

- Dynamic growth
- No shifting
- No wasted cells
- O(1) enqueue and dequeue with **front & rear pointers**

This is the BEST way to implement a queue.

2 Structure of Linked List Queue

We maintain:

- front → points to FIRST node
- rear → points to LAST node

Visual:

```
front → [10] → [20] → [30] → null ← rear
```

FIFO is naturally achieved:

- Insert at **rear**
- Delete at **front**

3 Node Structure

```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        next = null;
    }
}
```

4 Queue Class Using Linked List

```
class LinkedListQueue {
    Node front, rear;

    // constructor
    LinkedListQueue() {
        front = rear = null;
    }

    // enqueue operation
    void enqueue(int x) {
        Node newNode = new Node(x);

        if (rear == null) {          // empty queue
            front = rear = newNode;
            return;
        }

        rear.next = newNode;
        rear = newNode;
    }
```

```
    // dequeue operation
    int dequeue() {
        if (front == null) {
            System.out.println("Queue Underflow");
            return -1;
        }

        int value = front.data;
        front = front.next;

        if (front == null)          // queue became empty
            rear = null;

        return value;
    }

    // peek operation
    int front() {
        if (front == null) {
            System.out.println("Queue is Empty");
            return -1;
        }
        return front.data;
    }

    // check empty
    boolean isEmpty() {
        return front == null;
    }

    // display queue
    void display() {
        if (isEmpty()) {
            System.out.println("Queue is Empty");
            return;
        }

        Node temp = front;
        System.out.print("Queue: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
```

5 How enqueue works (Visual)

Operation:

```
enqueue(40)
```

Before:

```
front → 10 → 20 → 30 → null ← rear
```

After:

```
front → 10 → 20 → 30 → 40 → null ← rear
```

Add new node at the **end**.

6 How dequeue works (Visual)

Operation:

```
dequeue()
```

Before:

```
front → 10 → 20 → 30 → null ← rear
```

After:

```
front → 20 → 30 → null ← rear
```

Removed: 10

Delete from **front**.

7 Dry Run Example

Operations:

```
enqueue(10)
enqueue(20)
enqueue(30)
dequeue()
enqueue(40)
enqueue(50)
```

Final queue:

```
front → 20 → 30 → 40 → 50 → null ← rear
```

8 Time & Space Complexity

Operation	Time	Reason
enqueue	O(1)	Insert at rear

Operation	Time	Reason
dequeue	O(1)	Remove at front
peek	O(1)	Return front node
isEmpty	O(1)	Pointer check
Space	O(n)	n nodes

Stable constant-time performance.

9 Advantages

- No overflow
- Dynamic size
- No wasted space
- Perfect O(1) operations
- Ideal for large or unpredictable data size
- Used in BFS, OS scheduling, network packet handling

10 Limitations

- Requires dynamic memory
- Pointer overhead
- Slightly slower than array in CPU caching
- Implementation more complex

Quick Summary

- Queue implemented using **Linked List**
- front points to first node
- rear points to last node
- enqueue = insert at rear
- dequeue = delete at front
- All operations O(1)
- Perfect dynamic queue → no overflow, no wasted space

Mantra:

"Linked List Queue is the purest form of FIFO — free, dynamic, and fair."

4.9 — Circular Queue (CQ)

The perfect queue. Zero waste. 100% utilization.

1 Why Circular Queue? (The Problem in Linear Queue)

In simple array queue:

enqueue → rear moves forward

dequeue → front moves forward

After some operations, array looks like this:

```
[ X   X   X   40   50 ]
  ↑   ↑   ↑
```

abandoned space.

Even though there is empty space at the front, we cannot insert because:

```
rear = size - 1 → Overflow
```

- Wasted space
- False overflow

2 Solution: Circular Queue Concept

We treat the array as **circular**, not linear.

Meaning:

- After index size-1, next position becomes 0
- Front and rear "wrap around"

Formula:

```
rear = (rear + 1) % size
front = (front + 1) % size
```

3 Circular View (Visual)

Imagine the queue as a ring:

```
  [0] → [1] → [2] → [3] → [4]
    ↑                       ↓
     ← ← ← ← ← ← ← ← ← ← ← ←
```

- rear can rotate back to index 0
- front can rotate behind rear
- NO wasted space!

4 Important Condition: FULL or EMPTY?

In CQ:

Queue is Empty when:

```
front == -1
```

Queue is Full when:

```
(front == (rear + 1) % size)
```

This condition is very important in interviews.

5 Initial Conditions

```
front = -1
rear  = -1
```

6 Enqueue Operation (Insert)

Steps:

1. Check if full
2. Move rear → (rear + 1) % size
3. Insert
4. If queue was empty, set front = 0

Java Code — enqueue():

```
void enqueue(int x) {
    if (isFull()) {
        System.out.println("Queue Overflow");
        return;
    }

    if (front == -1) { // first element
        front = 0;
    }

    rear = (rear + 1) % size;
    arr[rear] = x;
}
```

7 Dequeue Operation (Delete)

Steps:

1. Check if empty
2. Remove element at front
3. If front == rear → queue becomes empty
4. Else move front → (front + 1) % size

Java Code — dequeue():

```
int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue Underflow");
        return -1;
    }

    int value = arr[front];
```

```
 if (front == rear) {   // single element deleted
        front = rear = -1;
    } else {
        front = (front + 1) % size;
    }

    return value;
}
```

8 isEmpty() and isFull()

```
boolean isEmpty() {
    return front == -1;
}

boolean isFull() {
    return (front == (rear + 1) % size);
}
```

9 Display Function

```
void display() {
    if (isEmpty()) {
        System.out.println("Queue is Empty");
        return;
    }

    System.out.print("Circular Queue: ");

    int i = front;
    while (true) {
        System.out.print(arr[i] + " ");
        if (i == rear) break;
        i = (i + 1) % size;
    }
    System.out.println();
}
```

10 Dry Run Example (Step by Step)

Queue size = 5

Operations:

```
enqueue(10)
enqueue(20)
enqueue(30)
enqueue(40)
```

```
dequeue()      // deletes 10
enqueue(50)
enqueue(60)    // wraps around
```

State:

```
Index:     0    1    2    3    4
           60   20   30   40   50
            ↑                   ↑
          rear                front
```

Notice how the queue wraps from end → start.

11 Visual Evolution

After 4 enqueues:

```
front → 0
rear  → 3

[10][20][30][40][_]
```

After dequeue:

```
front → 1
rear  → 3

[_][20][30][40][_]
```

enqueue(50):

```
front → 1
rear  → 4

[_][20][30][40][50]
```

enqueue(60) — WRAP AROUND:

```
front → 1
rear  → 0

[60][20][30][40][50]
```

- NO WASTED SPACE
- NO false overflow
- Perfect rotation

12 Time & Space Complexity

Operation	Time
enqueue	O(1)

Operation	Time
dequeue	O(1)
display	O(n)
Space	O(n)

1️⃣3️⃣ Applications of Circular Queue

Circular Queue is used in:

- CPU Scheduling (Round Robin)
- Task Scheduling
- Network Routers (buffering packets)
- Keyboard buffer
- Printers & hardware drivers
- Streaming media data packets

Perfect for cyclical, continuous systems.

Quick Summary

- Solves wasted-space issue from linear queue
- Uses modulo % for rotation
- front and rear wrap around
- Full when: front == (rear + 1) % size
- All operations O(1)
- Used heavily in OS and networks

Mantra:

"Circular Queue turns the line into a loop — no beginning, no waste, perfect flow."

4.10 — Priority Queue (Introduction)

When order depends on importance, not arrival.

1️⃣ What Is a Priority Queue?

A **Priority Queue** is a type of queue where:

- Every element has a **priority**
- Removal happens based on **highest priority**, not the arrival time
- If priorities are equal → use FIFO

Example (Descending Priority Queue):

Insert:

(5), (1), (10), (3)

Remove order:

10, 5, 3, 1

Priority decides who leaves first, not position in line.

2 Why Priority Queue? (Real Purpose)

Some tasks cannot wait simply because they arrived late.

Real-world examples:

- Emergency rooms treat most urgent patients first
- CPU scheduling runs highest priority process first
- Print queue prints important documents first
- Network routers send **important packets first**
- Dijkstra's algorithm picks **shortest distance node first**

Priority Queue = fairness + intelligence.

3 Types of Priority Queues

1. Ascending Priority Queue

- Smallest value → highest priority
- Example: Dijkstra's shortest path

2. Descending Priority Queue

- Largest value → highest priority
- Example: Job scheduling

4 Priority Queue vs Normal Queue

Feature	Normal Queue	Priority Queue
Rule	FIFO	Based on priority
Removal	Front item	Highest priority item
Insertion	Always at rear	Position depends on priority
Use-case	Order-based tasks	Urgency-based tasks

5 Implementations (Concept Level)

Priority Queue can be implemented using:

Unsorted Array

- Insert: O(1)
- Remove highest priority: O(n)

Sorted Array

- Insert: O(n)
- Remove highest priority: O(1)

Unsorted Linked List

- Insert: O(1)

- Remove highest priority: O(n)

Sorted Linked List

- Insert: O(n)
- Remove: O(1)

Binary Heap (Optimal)

- Insert: O(log n)
- Remove highest priority: O(log n)
- Peek: O(1)

Most real systems use **Min-Heap or Max-Heap** — covered in Tree/Heap unit.

6 Implementation — Priority Queue Using Sorted Linked List

Simple to understand and perfect for intro.

Rule:

- Insert elements in sorted order
- Front always holds highest priority

Code:

```
class Node {
    int data;
    Node next;

    Node(int data) {
        this.data = data;
        next = null;
    }
}
class PriorityQueueLL {
    Node front;

    void enqueue(int x) {
        Node newNode = new Node(x);

        // Empty or highest priority
        if (front == null || x > front.data) {
            newNode.next = front;
            front = newNode;
            return;
        }

        // Insert at correct position
        Node temp = front;
        while (temp.next != null && temp.next.data >= x) {
            temp = temp.next;
        }
```

```
        newNode.next = temp.next;
        temp.next = newNode;
    }

    int dequeue() {
        if (front == null) {
            System.out.println("Underflow");
            return -1;
        }

        int value = front.data;
        front = front.next;
        return value;
    }

    void display() {
        Node temp = front;
        System.out.print("PQ: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
```

7 Dry Run Example

Insert:

```
enqueue(40)
enqueue(10)
enqueue(50)
enqueue(30)
```

Stored as descending priority:

```
front → 50 → 40 → 30 → 10 → null
```

dequeue() → returns **50**
Next → **40**

8 Use Cases of Priority Queue

Operating Systems:

- CPU Scheduling (Preemptive Priority Scheduling)
- Task management
- Interrupt handling

Networks:

- Router packet scheduling
- Bandwidth allocation

Artificial Intelligence:

- A* Search
- Greedy algorithms

Graph Algorithms:

- Dijkstra's algorithm
- Prim's MST algorithm

Other:

- Event-driven simulation
- Hospital emergency room
- Ticketing systems (VIP first)

Priority Queue appears everywhere urgency matters.

9 Performance Summary Table

Implementation	Insert	Remove	Best Use
Unsorted Array	O(1)	O(n)	Many inserts, few removes
Sorted Array	O(n)	O(1)	Many removes, few inserts
Unsorted LL	O(1)	O(n)	Linked structure + fast insert
Sorted LL	O(n)	O(1)	Fast remove
Binary Heap	O(log n)	O(log n)	Real-world priority queue

Quick Summary

- Priority Queue removes elements based on **priority**, not position
- Higher priority → processed earlier
- Implemented using arrays/LL/heaps
- Heaps give the best performance
- Used in OS, AI, networks, shortest path algorithms
- Great stepping stone for understanding **Binary Heap**

Mantra:

"Queues decide order. Priority Queues decide importance."

4.11 — Deque (Double-Ended Queue)

Freedom at both ends.

1 What Is a Deque?

A **Deque** (pronounced “deck”) is a **Double-Ended Queue** where:
Insertions and deletions can happen at **both ends**.

It generalizes both:

- **Queue** (insert rear, delete front)
- **Stack** (insert/delete same end)

2 Types of Deques

1. Input-Restricted Deque

- Input (insert) only allowed at **one** end
- Output (delete) from **both** ends

2. Output-Restricted Deque

- Deletion only at **one** end
- Insertion allowed at **both** ends

Makes it easy to build special queues with constraints.

3 Operations of a Deque

Operation	Meaning
insertFront(x)	Insert at front
insertRear(x)	Insert at rear
deleteFront()	Delete from front
deleteRear()	Delete from rear
front()	Get front element
rear()	Get rear element
isEmpty()	Check empty
isFull()	Only for array deque

4 Array Implementation (Circular Deque)

Use modulo just like circular queue.

Key Formulas:

```
front = (front - 1 + size) % size
rear = (rear + 1) % size
```

Allows both ends to shift circularly.

5 Java Implementation (Array Circular Deque)

```
class Deque {
    int size;
    int front, rear;
    int[] arr;

    Deque(int size) {
        this.size = size;
        arr = new int[size];
        front = -1;
        rear = -1;
    }

    boolean isFull() {
        return (front == (rear + 1) % size);
    }

    boolean isEmpty() {
        return (front == -1);
    }

    void insertFront(int x) {
        if (isFull()) return;

        if (front == -1) {   // first element
            front = rear = 0;
        } else {
            front = (front - 1 + size) % size;
        }
        arr[front] = x;
    }

    void insertRear(int x) {
        if (isFull()) return;

        if (front == -1) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % size;
        }
        arr[rear] = x;
    }

    int deleteFront() {
        if (isEmpty()) return -1;
```

```
 int val = arr[front];

        if (front == rear)
            front = rear = -1;
        else
            front = (front + 1) % size;

        return val;
    }

    int deleteRear() {
        if (isEmpty()) return -1;

        int val = arr[rear];

        if (front == rear)
            front = rear = -1;
        else
            rear = (rear - 1 + size) % size;

        return val;
    }
}
```

6 Applications of Deque

1. Palindrome Checking

- Store characters in deque and compare front ↔ rear.

2. Sliding Window Maximum (VERY common in interviews)

- Deque helps maintain candidate max values.

3. LRU Cache Implementation

- Front = most recently used
- Rear = least recently used

4. Undo/Redo

- Two-ended control of operations.
- Deque is extremely powerful in real-time data processing.

7 Time Complexity

Operation	Time
Insert Front	O(1)
Insert Rear	O(1)
Delete Front	O(1)

Operation	Time
Delete Rear	O(1)
Access Front / Rear	O(1)

Quick Summary

- Deque allows insert & delete at **both ends**
- Variants: Input-restricted, Output-restricted
- Used in palindrome checking, LRU cache, sliding window
- Fast O(1) operations for all ends

Mantra:

"Deque gives complete freedom — front or back, insert or delete… anything is possible."

4.12 — Comparison & Applications

Choosing the right structure for the right flow.

1 Conceptual Comparison (At a Glance)

ADT	Access Rule	Insertion	Deletion	Typical Use
Stack	LIFO	At top	From top	Undo/Redo, Recursion
Queue	FIFO	Rear	Front	Scheduling, Buffering
Circular Queue	FIFO (looped)	Rear (modulo)	Front (modulo)	CPU RR Scheduling
Priority Queue	Priority-based	Anywhere (based on priority)	Highest/Lowest priority	Dijkstra, OS Scheduling
Deque	Both ends	Front + Rear	Front + Rear	LRU Cache, Sliding Window

Each structure solves a different behavioral need.

2 Behavior & Access Pattern

Feature	Stack	Queue	Circular Queue	Priority Queue	Deque
Order	LIFO	FIFO	FIFO (cyclic)	By priority	Flexible
Ends Used	One end	Two ends	Two ends (wrapped)	One end (based on priority)	Both ends
Random Access	❌	❌	❌	❌	❌
Dynamic variants	Yes	Yes	Yes	Yes	Yes

3 Operation Complexity

Assume optimal implementations:

Operation	Stack	Queue	C-Queue	Priority Queue*	Deque
Insert	O(1)	O(1)	O(1)	O(log n) (heap)	O(1)

Operation	Stack	Queue	C-Queue	Priority Queue*	Deque
Delete	O(1)	O(1)	O(1)	O(log n) (heap)	O(1)
Peek	O(1)	O(1)	O(1)	O(1)	O(1)

*In this unit we introduce PQ conceptually; full heap implementation comes in Tree/Heap chapter.

4 Strengths and Limitations

Stack

- **Strengths:** Fast, simple, perfect for reverse order
- **Limitations:** Only one-end operations

Queue

- **Strengths:** Fair order, perfect for scheduling
- **Limitations:** Wasted space in simple array implementation

Circular Queue

- **Strengths:** No wasted space, great for cyclic processes
- **Limitations:** Fixed size if array-based

Priority Queue

- **Strengths:** Urgency-based removal
- **Limitations:** Slower insertion/deletion due to sorting/heaps

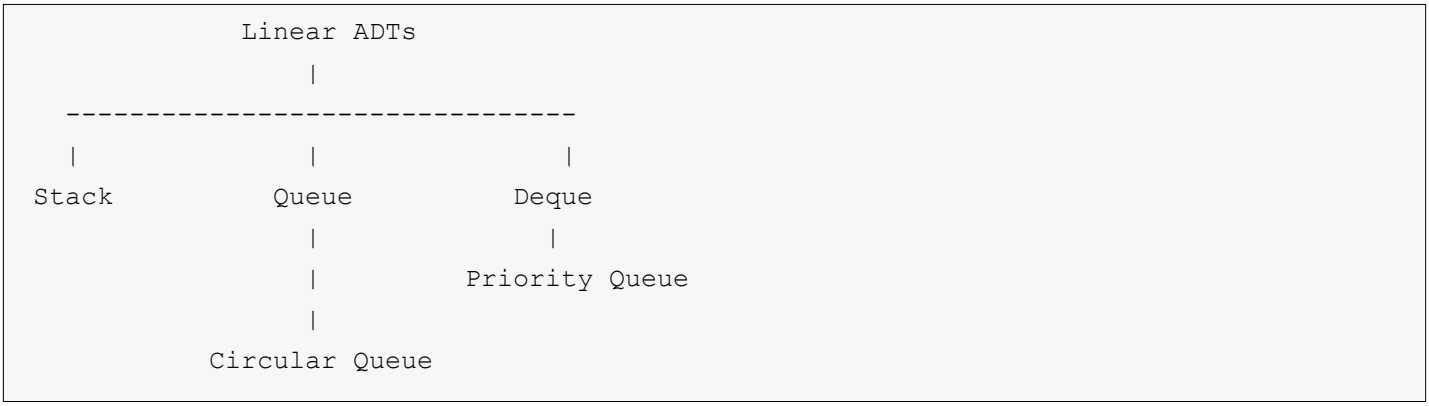
Deque

- **Strengths:** Maximum flexibility, two-way operations
- **Limitations:** Needs more careful pointer/index management

5 Real-World Use Cases Mapped

System / Domain	Ideal Structure	Reason
Browser Back Button	Stack	LIFO navigation
Undo/Redo	Stack	Reverse action order
Printer Queue	Queue	First job printed first
CPU Scheduling (RR)	Circular Queue	Cyclic rotation
OS Process Scheduling (priority)	Priority Queue	Urgency-based execution
BFS (Graph Traversal)	Queue	Level-order visiting
LRU Cache	Deque	Remove from rear, add to front
Sliding Window Problems	Deque	O(1) two-end operations
Network Routers	Circular Queue	Continuous packet buffering

6️⃣ Relationship Between All Structures



The ADTs form a family — each solving a different access and order challenge.

7️⃣ How to Choose the Right ADT (Rules of Thumb)

- Use **Stack** → when the last thing must be handled first
- Use **Queue** → when fairness and order matter
- Use **Circular Queue** → when continuous looping is needed
- Use **Priority Queue** → when importance beats arrival time
- Use **Deque** → when both ends matter equally

Quick Summary

- **Stack:** LIFO → undo, recursion, backtracking
- **Queue:** FIFO → scheduling, buffering
- **Circular Queue:** cyclic FIFO → RR scheduling
- **Priority Queue:** priority-based → graphs, OS
- **Deque:** two-end operations → LRU cache, sliding windows
- Each ADT has specific strengths depending on the required access pattern.

Mantra:

"Order defines behavior. Behavior defines the ADT."

4.13 — Summary & Reflection

When order, discipline, and flow come together.

1️⃣ Essence of Stacks & Queues

In one line:

Stacks control order of reversal. Queues control order of arrival.

These two ADTs form the backbone of:

- Expression evaluation
- Scheduling
- Recursion
- BFS

- Backtracking
- Buffering
- System-level operations

They aren't just data structures — they are **behaviors** of data.

2 The Family Tree

ADT	Core Rule	Operations	Implementations
Stack	LIFO	push, pop, peek	Array / Linked List
Queue	FIFO	enqueue, dequeue, front	Array / Linked List
Circular Queue	FIFO (circular)	enqueue, dequeue	Array (modulo)
Priority Queue	Remove based on priority	insert, delete-min/max	Array, LL, Heap
Deque	Insert/delete at both ends	addFront/addRear, deleteFront/deleteRear	Array

3 Implementation Summary

Stack

- Array → fast but fixed size
- Linked List → dynamic, no overflow

Operations:

push — O(1)

pop — O(1)

peek — O(1)

Queue

- Array queue → simple, but wastes space
- Linked List queue → dynamic
- Circular queue → perfect space utilization
- Deque → flexible from both ends
- Priority queue → ordered by priority

Operations (standard queue):

enqueue — O(1)

dequeue — O(1)

front — O(1)

4 Key Concepts to Remember

Stack Overflow & Underflow

- Overflow → push into full array stack
- Underflow → pop from empty stack

Queue Overflow & Underflow

- In linear queue
  - Overflow even when space exists -> the ghost cell problem
- Circular queue fixes this

CQ Full Condition

front == (rear + 1) % size

CQ Empty Condition

front == -1

5️⃣ Applications Snapshot

Stack

- Function calls (call stack)
- Expression conversion
- Expression evaluation
- Undo/Redo
- DFS, backtracking
- Parenthesis matching
- Balanced brackets

Queue

- CPU Scheduling
- Printer Spoolers
- Router buffering
- BFS traversal
- Multi-threaded task management
- Customer service systems

Circular Queue

- Round Robin Scheduling
- Hardware buffers
- Network packet queues

Priority Queue

- Dijkstra's algorithm
- OS process scheduling
- Data compression (Huffman Coding)

Deque

- Palindrome checking
- Sliding window maximum
- LRU Cache (foundation structure)

6 Complexity Table (Full Unit Summary)

Structure	Insert	Delete	Peek	Space
Stack (Array/LL)	O(1)	O(1)	O(1)	O(n)
Queue (Array/LL)	O(1)	O(1)	O(1)	O(n)
Circular Queue	O(1)	O(1)	O(1)	O(n)
Deque	O(1) both ends	O(1) both ends	O(1)	O(n)
Priority Queue (Array)	O(n)	O(n)	O(1)	O(n)

7 Mistakes to Avoid

- Forgetting modulo in Circular Queue
- Confusing full vs empty conditions
- Not resetting front & rear to -1 when last element is removed
- Treating both ends of deque the same (they have roles)
- Missing operator precedence in infix → postfix
- One missed step = wrong evaluation.

8 Mindflow

Stacks → LIFO → Recursion → Expressions → Backtracking

Queues → FIFO → Scheduling → BFS → OS/Networking

Circular Queue → Fix linear queue → Perfect rotation

Priority Queue → Task priority → Algorithms (Dijkstra)

Deque → Two-end control → Sliding windows & caching