1. Definition of Stack

A **Stack** is a linear data structure that follows the **LIFO (Last In First Out)** principle:

- The **last** element added is the **first** to be removed.
- Think of it like a **stack of plates**: you add to the top and remove from the top.

Basic Operations:

- push(x) – add an element to the top
- pop() – remove the top element
- peek() or top() – see the top element without removing it
- isEmpty() – check if the stack is empty

2. Where Stack is Applied (Real-World & Coding)

Real-World Use Cases:

- Browser history (back navigation)
- Undo functionality in editors
- Reversing a string
- Expression evaluation (postfix, infix)
- Call stack in programming languages

Coding Problem Use Cases:

- Balanced Parentheses (()[]{})
- Next Greater Element
- Evaluate Reverse Polish Notation
- Decode Strings ("3[a2[c]]")
- Backspace String Compare
- Histogram (Largest Rectangle)
- Remove k Digits
- Monotonic Stack Problems

3. When to Think of Using a Stack in a Problem

Here are **clues** in a problem statement that should trigger "maybe I need a stack":

Clue/Keyword	What it suggests
Nested structures like "(([]))"	Balanced parentheses → Stack
“Undo”, “Backtrack”, “Previous State”	LIFO structure → Stack
“Next Greater/Smaller Element”	Monotonic Stack
“Evaluate expression”	Operator precedence → Stack
“Backspace”, “Reversal”	Track last characters → Stack
“Track previous elements”	Stack of indices/values

4. How to Apply Stack in Coding

In Java:

```
Stack<Integer> stack = new Stack<>();
stack.push(10);
int top = stack.peek();
stack.pop();
boolean empty = stack.isEmpty();
```

In problems:

- Use Stack<Integer> for indices (monotonic stacks)
- Use Stack<Character> for parsing characters
- Use Stack<String> for decoding problems

5. Benefits of Using Stack

Benefit	Description
Simple & Fast	Push/pop is O(1)
Natural fit for recursive/ backtracking problems	Manual control of call stack
Memory-efficient	Keeps only what's needed (LIFO)
Easy to implement	Built-in in most languages

Tips

- Use **Monotonic Stack** when you're asked for "next greater/smaller element".
- For nested structures or reversal → **Classic Stack**
- Use stack of **pairs (value, index)** for advanced use (like tracking previous or frequency)
- You can simulate recursion using a **stack** manually if recursion depth is a concern.

#	Problem Title	Pattern	Link
1	Valid Parentheses	Bracket Matching	🔗 Link

2	Min Stack	Custom Stack Design	🔗 Link
3	Daily Temperatures	Monotonic Stack	🔗 Link
4	Next Greater Element I	Monotonic Stack	🔗 Link
5	Next Greater Element II	Monotonic Stack + Circular Array	🔗 Link
6	Decode String	Stack + String Parsing	🔗 Link
7	Evaluate Reverse Polish Notation	Postfix Evaluation	🔗 Link
8	Remove All Adjacent Duplicates In String	Stack for Deduplication	🔗 Link
9	Remove K Digits	Greedy + Monotonic Stack	🔗 Link
10	Largest Rectangle in Histogram	Hard Monotonic Stack	🔗 Link
11	Asteroid Collision	Stack Simulation	🔗 Link
12	Backspace String Compare	Stack Simulation	🔗 Link
13	Implement Stack using	Stack Simulation	🔗 Link

	Queues		
14	Simplify Path	Path Resolution using Stack	🔗 Link
15	Basic Calculator II	Expression Evaluation	🔗 Link