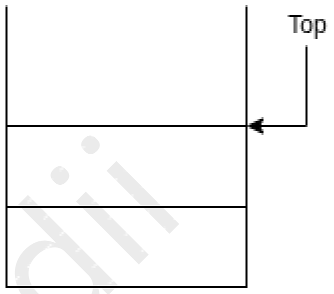
Stack

By Mani Abedii

A **stack** is a linear data structure that follows the **LIFO** (Last In, First Out) principle. This means the last element added to the stack is the first one to be removed, much like a stack of plates in a cafeteria — you can only take the plate on top, and when you add a plate, it goes on top of the stack.



Stacks support a few fundamental operations, all in **O(1)** time complexity:

1. **Push:** Adds an element to the top of the stack.
2. **Pop:** Removes and returns the element from the top of the stack.
3. **Peek** or **Top:** Returns the top element without removing it.
4. **isEmpty:** Checks if the stack is empty.

There are two major errors that can occur in stacks.
Overflow happens when we try to push a new element into a stack that is already full, which usually occurs in fixed-size implementations.
Underflow happens when we try to pop an element from a stack that is empty.

Use Cases of Stack:

1. **Function call stack** (Managing function calls/recursion).
2. **Undo/Redo** operations in editors.
3. **Expression evaluation** (see below).
4. **Balanced parentheses checking** (see below).
5. **Backtracking** problems (like maze solving, DFS).

Expression Evaluation with Stacks

There are 3 main notations for writing mathematical expressions.

- Infix: Operators between operands (e.g., 3 + 4)
- PostFix (Reverse Polish): Operators after operands (e.g., 3 4 +)
- Prefix (Polish): Operators before operands (e.g., + 3 4)

Computers find postfix easier to evaluate because it doesn't need parentheses or precedence rules.

```
E.g.:
a * (b + c) - (c - d) / e      <- Infix
==PostFix==>      a * (bc +) - (cd -) / e
            =>    (a bc + *) - (cd - e /)
            =>    a bc + * cd - e / -
```

Infix → Postfix (Shunting-yard algorithm)

1. Scan from left to right.
2. Operands go directly to the output.
3. Operators are pushed onto a stack. Pop from the stack to the output if the operator on the stack has higher or equal precedence.
4. Left parenthesis (is pushed onto the stack.
5. On encountering), pop until (is found.
6. Finally, pop all remaining operators to the output.

Infix → Prefix

1. Reverse the infix string.
2. Swap (with).
3. Convert to postfix using the Shunting-yard algorithm.
4. Reverse the result.

Postfix or **Prefix → Infix**

1. Scan from left to right (Postfix) / from right to left (Prefix).
2. Operands are pushed onto the stack.
3. If operator → pop two operands, combine as (left operator right), push result back.
4. Final stack item = infix expression.

Evaluating Postfix

1. Scan from left to right.
2. Operands are pushed onto the stack.
3. If operator → pop two operands, apply the operator, push the result back.
4. Final stack element = result

```
Postfix: 3 4 2 * + 5 -
Step 1: Push 3
Step 2: Push 4
Step 3: Push 2
Step 4: * → (4 * 2 = 8) → push 8          Stack = [3, 8]
Step 5: + → (3 + 8 = 11) → push 11        Stack = [11]
Step 6: Push 5
Step 7: - → (11 - 5 = 6) → push 6         Final Answer = 6
```

Balanced parentheses checking

One of the most common applications of stacks is checking if parentheses in an expression are balanced.

When we say an expression has balanced parentheses, it means that every opening symbol like `(`, `{`, or `[` has a matching closing symbol `)`, `}`, or `]`, and they appear in the correct order.

The idea is simple:

1. Scan the expression from left to right.
2. Whenever we see an opening bracket, we push it onto the stack.
3. Whenever we see a closing bracket, we check the top of the stack. If the top has the matching opening bracket, we pop it. Otherwise, the expression is not balanced.
4. At the end, if the stack is empty, the parentheses are balanced. If the stack still has items, it means there are unmatched brackets.

```
Expression: (a + b * (c - d)
Stack process: ( → push, ( → push, ) → pop
End: one ( remains in stack
Result: Not Balanced
```