

UNIT 2

The Stack

(3 Hours) — Data Structures and Algorithms (CACS201)

2.1 Introduction

A **Stack** is a linear data structure in which insertion and deletion of elements are performed only from one end, called the **TOP** of the stack. Because of this restriction, a stack follows the **LIFO (Last-In-First-Out)** principle — the element inserted last is the first one to be removed.

Everyday examples of stack behaviour include a stack of plates (the plate placed last is taken off first), a stack of books, or a pile of coins. In computing, stacks are used to manage function calls, undo operations in editors, and to evaluate arithmetic expressions — topics covered later in this unit.

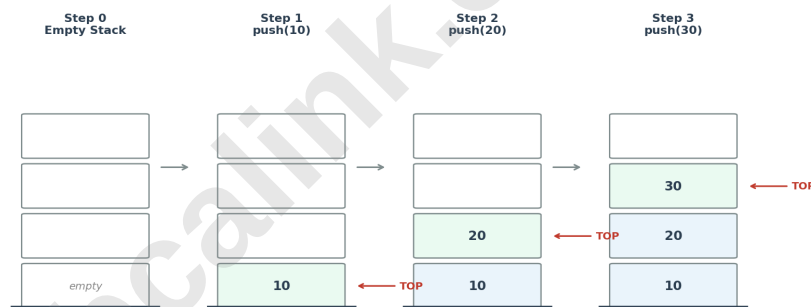


Figure 2.1 — A sequence of `push()` operations on a stack of size 4

*Note: In an array-based implementation, a variable called **TOP** keeps track of the index of the topmost element. **TOP** = -1 (or 0, depending on convention) indicates an empty stack.*

2.2 Stack as an Abstract Data Type

As an ADT, the Stack specifies a set of operations and the LIFO rule that governs them, without prescribing how the stack is implemented internally (array or linked list).

Operation	Description	Typical Return
<code>push(item)</code>	Inserts an element on top of the stack	None / success flag
<code>pop()</code>	Removes and returns the top element	Removed element
<code>peek() / top()</code>	Returns the top element without removing it	Top element

Operation	Description	Typical Return
isEmpty()	Checks whether the stack has no elements	true / false
isFull()	Checks whether the stack has reached its capacity	true / false (array only)
size()	Returns the number of elements currently in the stack	Integer count

2.3 PUSH and POP Operations

2.3.1 PUSH Operation

The push() operation inserts a new element at the top of the stack. In an array implementation of maximum size N, TOP is first checked against N-1 to detect a **stack overflow** condition before the element is inserted.

Algorithm — push(item)

1. If $TOP = N - 1$, print "Stack Overflow" and exit.
2. Set $TOP = TOP + 1$.
3. Set $STACK[TOP] = item$.
4. Exit.

2.3.2 POP Operation

The pop() operation removes and returns the element currently at the top of the stack. Before removing, TOP is checked to detect a **stack underflow** condition (attempting to pop from an empty stack).

Algorithm — pop()

1. If $TOP = -1$, print "Stack Underflow" and exit.
2. Set $item = STACK[TOP]$.
3. Set $TOP = TOP - 1$.
4. Return item.

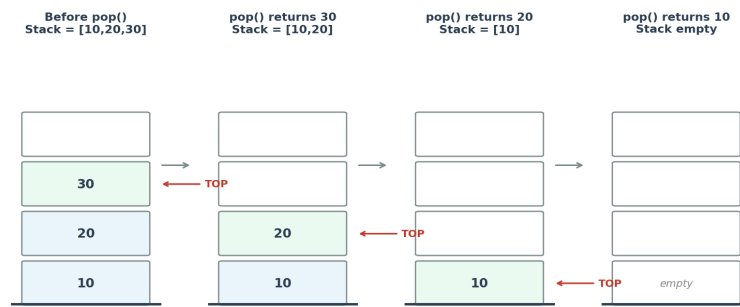


Figure 2.2 — A sequence of pop() operations on a stack

Worked Example — push/pop Trace

Start with an empty stack of size 4 (TOP = -1).

push(10) → TOP = 0, Stack = [10]

push(20) → TOP = 1, Stack = [10, 20]

push(30) → TOP = 2, Stack = [10, 20, 30]

pop() → returns 30, TOP = 1, Stack = [10, 20]

peek() → returns 20 (TOP unchanged)

2.4 Stack Application: Evaluation of Infix, Postfix, and Prefix Expressions

One of the most important applications of a stack is in evaluating and converting arithmetic expressions. Expressions can be written in three notations, depending on where the operator is placed relative to its operands:

Notation	Operator Position	Example (A + B)
Infix	Between the two operands	A + B
Postfix (Reverse Polish)	After both operands	A B +
Prefix (Polish)	Before both operands	+ A B

Infix expressions are natural for humans to read but require operator precedence and parentheses to resolve ambiguity. Postfix and prefix expressions remove this ambiguity entirely, which is why compilers convert infix expressions into postfix or prefix form before evaluating them — and a stack is the natural tool for both the conversion and the evaluation.

Operator	Precedence	Associativity
^ (exponent)	Highest (3)	Right to Left

Operator	Precedence	Associativity
$*, /$	Medium (2)	Left to Right
$+, -$	Lowest (1)	Left to Right

2.4.1 Evaluation of a Postfix Expression

A postfix expression is evaluated in a single left-to-right scan using one stack, since the notation already encodes the order of operations — no precedence rules or parentheses are needed.

Algorithm — Evaluate Postfix Expression

1. Scan the expression from left to right, one token at a time.
2. If the token is an **operand**, push it onto the stack.
3. If the token is an **operator**, pop the top two elements ($op2$ first, then $op1$), compute $op1$ *operator* $op2$, and push the result back.
4. After the scan ends, the single value remaining on the stack is the final result.

Worked Example — Evaluate Postfix "5 3 + 2 *"

Token '5' → operand, push → Stack = [5]

Token '3' → operand, push → Stack = [5, 3]

Token '+' → pop 3, pop 5, compute $5 + 3 = 8$, push → Stack = [8]

Token '2' → operand, push → Stack = [8, 2]

Token '*' → pop 2, pop 8, compute $8 * 2 = 16$, push → Stack = [16]

End of expression → **Result = 16**

2.4.2 Evaluation of a Prefix Expression

A prefix expression is evaluated similarly, but the scan proceeds from **right to left**.

Algorithm — Evaluate Prefix Expression

1. Scan the expression from right to left, one token at a time.
2. If the token is an **operand**, push it onto the stack.
3. If the token is an **operator**, pop the top two elements ($op1$ first, then $op2$), compute $op1$ *operator* $op2$, and push the result back.
4. After the scan ends, the single value remaining on the stack is the final result.

Worked Example — Evaluate Prefix `"* + 5 3 2"`

Scanning right to left: 2, 3, 5, +, *

Token '2' → push → Stack = [2]

Token '3' → push → Stack = [2, 3]

Token '5' → push → Stack = [2, 3, 5]

Token '+' → pop 5 (op1), pop 3 (op2), compute $5 + 3 = 8$, push → Stack = [2, 8]

Token '*' → pop 8 (op1), pop 2 (op2), compute $8 * 2 = 16$, push → Stack = [16]

End of expression → **Result = 16**

2.5 Conversion of Expressions

2.5.1 Infix to Postfix Conversion

A stack is used to temporarily hold operators and parentheses while building the postfix output string.

Algorithm — Infix to Postfix

1. Scan the infix expression from left to right.
2. If the token is an **operand**, append it directly to the output.
3. If the token is '(', push it onto the stack.
4. If the token is ')', pop and append operators to the output until '(' is popped (discard both parentheses).
5. If the token is an **operator**, pop and append all operators from the stack that have **greater or equal precedence**, then push the current operator.
6. After the scan ends, pop all remaining operators from the stack to the output.

Convert the infix expression `A + B * C - D / E` to postfix:

Symbol	Stack (bottom→top)	Output
A	(empty)	A
+	+	A
B	+	AB
*	+ *	AB
C	+ *	ABC
-	-	ABC*+ (pop *, pop +)
D	-	ABC*+D

Symbol	Stack (bottom→top)	Output
/	- /	ABC*+D
E	- /	ABC*+DE
end	(empty)	ABC*+DE/- (pop /, pop -)

Result

Infix: $A + B * C - D / E$

Postfix: $A B C * + D E / -$

2.5.2 Infix to Prefix Conversion

Infix-to-prefix conversion reuses the postfix algorithm with a simple three-step trick:

Algorithm — Infix to Prefix

1. Reverse the infix expression, and swap every '(' with ')' and every ')' with '('.
2. Convert the reversed expression to postfix using the Infix-to-Postfix algorithm above (treating equal-precedence operators as right-to-left for this step).
3. Reverse the resulting postfix expression to obtain the prefix expression.

Worked Example — Convert $(A + B) * (C - D)$ to Prefix

Step 1 – Reverse and swap brackets: $(D - C) * (B + A)$

Step 2 – Convert to postfix: $D C - B A + *$

Step 3 – Reverse the postfix string: $* + A B - C D$

Prefix result: $* + A B - C D$ — read as: multiply (A+B) by (C-D)

Note: Postfix-to-infix and prefix-to-infix conversions follow the reverse idea: scan the expression and, each time an operator is found, pop two operands, combine them as "(operand1 operator operand2)", and push the combined string back — the same stack mechanics used for evaluation, but strings instead of numbers.

Key Takeaways

- ✓ A Stack is a linear structure that follows the LIFO principle; all insertions and deletions happen at the TOP.
- ✓ As an ADT, a Stack exposes push, pop, peek, isEmpty, and isFull without exposing its internal implementation.
- ✓ push() must guard against **overflow** and pop() must guard against **underflow**.
- ✓ Postfix and prefix expressions remove the need for precedence rules and parentheses, so they can be evaluated in a single stack-based scan.
- ✓ Infix-to-postfix conversion uses a stack to hold operators until an operator of lower or equal precedence forces them out; infix-to-prefix reuses this algorithm on the reversed expression.

— End of Unit 2 —

bcalink.com