Data Structures

Week 7 – The stack: LIFO principle, implementation, applications

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

Review

- **Array:** direct access, insertion and deletion at the end are **O(1)**
- **Abstract data type (ADT):** **what** it can do, not **how** (week 1)
- **Set:** different representations, different efficiency

Questions for today

- What is a stack, and why is it so important?
- How can it be implemented with an array?
- How does a computer evaluate an expression like `(3 + 4) * 2`?

The concept of a stack

A **stack** is a data structure in which only the **most recently inserted** element is accessible.

LIFO – **Last In, First Out**: what went in last comes out first.

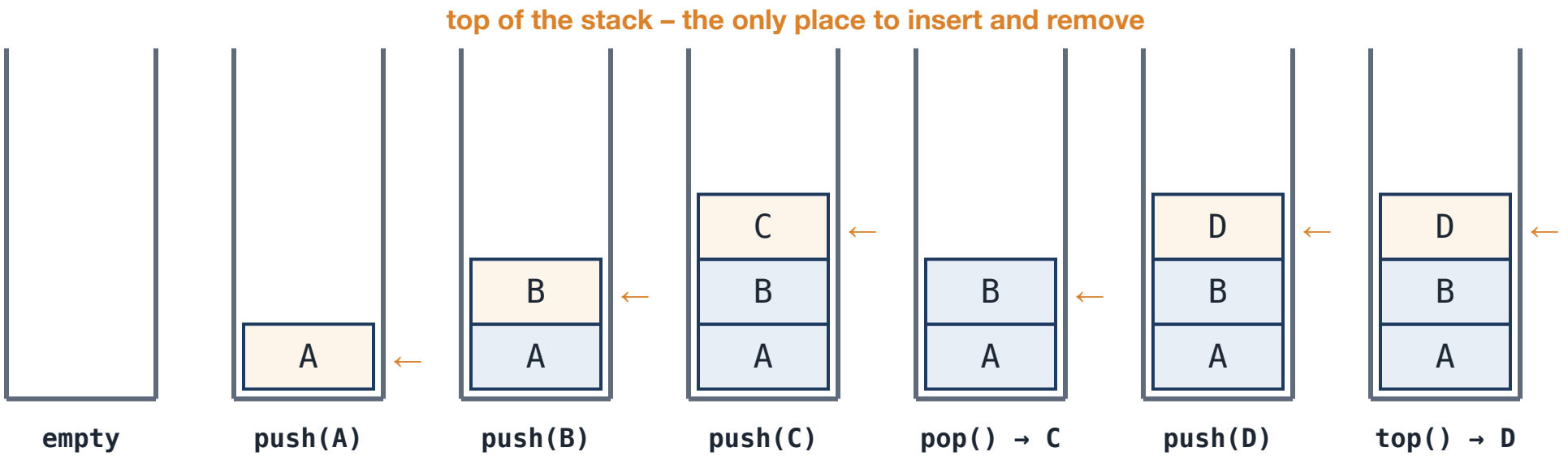
Everyday examples

- A pile of plates
- A Pringles chips can
- The “Back” button of a browser
- “Undo” in a text editor (Ctrl+Z)

Properties

Elements	homogeneous
Relationship	linear
Access	**restricted** – only the top
Size	static or dynamic

Stack operations



Stack operations – summary

Operation	Meaning	Steps
Create	empty stack	O(1)
Push	element onto the top	O(1)
Pop	removing and returning the top	O(1)
Top / Peek	reading the top without removing it	O(1)
IsEmpty? · IsFull?	querying the state	O(1)

Invalid operations

- **Pop or Top on an empty stack** → **underflow** (stack underflow)
- **Push onto a full stack** → **overflow** (stack overflow) – only for a fixed-size stack

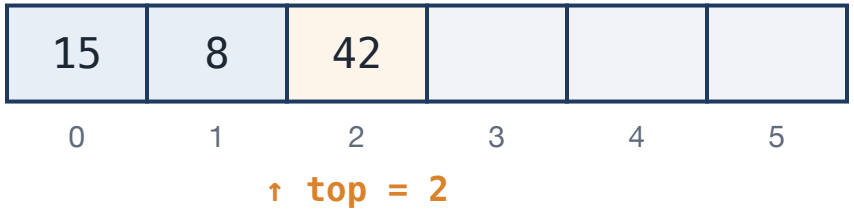
Before the operations, always check **IsEmpty?** / **IsFull?** – or let the implementation signal the error (Java: exception, Go: `error` return value).

This is where the name **Stack Overflow** comes from: with too deep recursion the program's **call stack** fills up (see later and week 12).

Implementation with an array

Stack with an array

Array stack: bottom at index 0, top at index top




```
push(x): top ← top + 1; T[top] ← x
pop(): x ← T[top]; top ← top − 1
empty: top = −1 full: top = n − 1
```

both O(1)

- The **bottom** of the stack is the beginning of the array, the **top** is at index `top`
- Push and Pop work at the **end** of the array → no elements need to be moved → **O(1)**

Stack with an array – pseudocode

```
// array T[1..n], top: index of the last element

Procedure Create
  top ← 0
End procedure

Function IsEmpty
  Return top = 0
End function

Function IsFull
  Return top = n
End function
```

```
Procedure Push(x)
  If IsFull then ERROR("overflow")
  else
    top ← top + 1
    T[top] ← x
  End if
End procedure

Function Pop
  If IsEmpty then ERROR("underflow")
  else
    x ← T[top]
    top ← top − 1
    Return x
  End if
End function
```

The pseudocode indexes from 1, so for an empty stack top = 0; in Java/Go we index from 0, so an empty stack has top = −1.

Stack with an array – Java and Go

Java

```java
class ArrayStack {
    private final int[] data;
    private int top = -1;

    ArrayStack(int capacity) {
        data = new int[capacity];
    }
    boolean isEmpty() { return top == -1; }
    boolean isFull()  { return top == data.length - 1; }

    void push(int x) {
        if (isFull())
            throw new IllegalStateException("full");
        data[++top] = x;
    }
    int pop() {
        if (isEmpty())
            throw new IllegalStateException("empty");
        return data[top--];
    }
}
```

Go – generic, with a slice

```go
type Stack[T any] struct {
    items []T
}

func (s *Stack[T]) Push(x T) {
    s.items = append(s.items, x)
}

func (s *Stack[T]) Pop() T {
    n := len(s.items) - 1
    x := s.items[n]
    s.items = s.items[:n]
    return x
}

func (s *Stack[T]) IsEmpty() bool {
    return len(s.items) == 0
}
```

A slice is a dynamic array → there is no “full” state.

Built-in stacks

Java

```java
Deque<Integer> st = new ArrayDeque<>();
st.push(1);
st.push(2);
st.peek();     // 2
st.pop();      // 2
st.isEmpty();  // false
```

Instead of the old `java.util.Stack` class, **`ArrayDeque`** is recommended (faster).

Go

```go
st := []int{}
st = append(st, 1)        // push
st = append(st, 2)
top := st[len(st)-1]      // peek
st = st[:len(st)-1]       // pop
empty := len(st) == 0
```

Go has no separate stack type – a **slice** is perfectly suitable.

Application: evaluating expressions

Infix, prefix, postfix

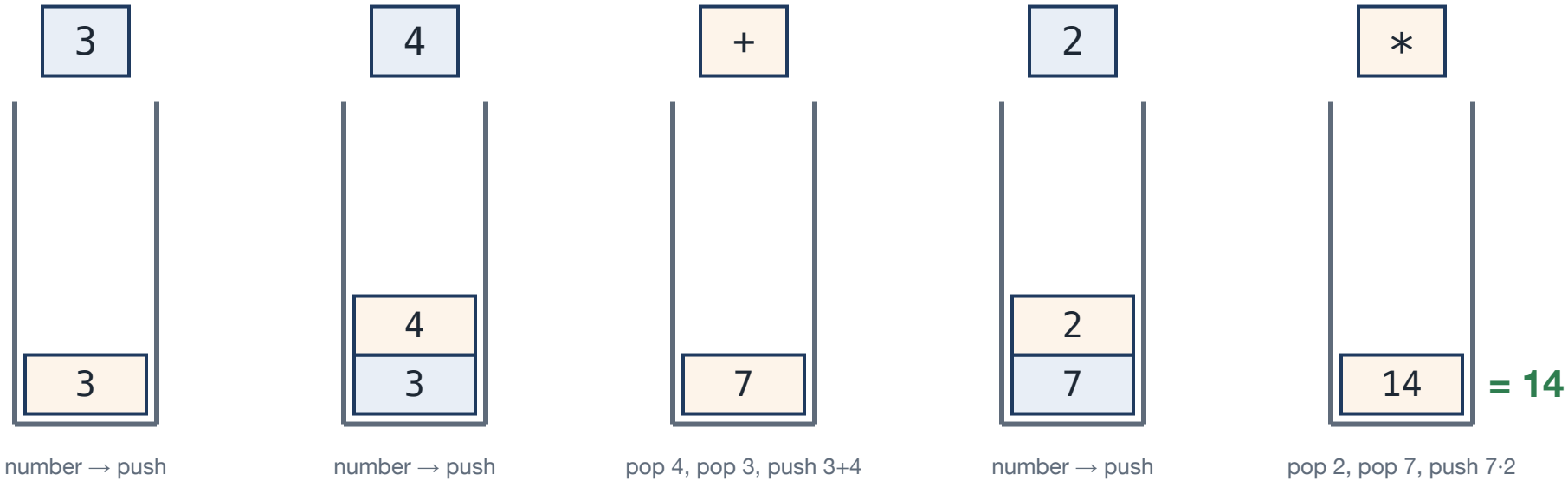
Notation	Example	Position of the operator
Infix	`(3 + 4) * 2`	between the operands – parentheses needed
Prefix (Polish)	`* + 3 4 2`	before the operands
Postfix (reverse Polish)	`3 4 + 2 *`	after the operands

Advantages of postfix:

- **No parentheses** and no precedence rules needed
- Can be evaluated from left to right, in **a single pass**, with one stack
- Compilers, the Java virtual machine and HP calculators work this way

Evaluating postfix with a stack

Postfix expression: 3 4 + 2 ∗ (= (3 + 4) · 2)



Evaluating postfix – algorithm

Pseudocode

```
Function PostfixValue(expression)
  Create
  For every token t from left to right
    If t is a number then
      Push(t)
    else                    // t is an operator
      b ← Pop               // right operand!
      a ← Pop               // left operand
      Push(a t b)
    End if
  End for
  Return Pop
End function
```

The **right** operand comes out first! `5 2 -` → b = 2, a = 5 → 5 – 2 = 3

Java

```java
ArrayDeque<Double> st = new ArrayDeque<>();
for (String t : expr.split("\\s+")) {
    if (isOperator(t)) {
        double b = st.pop();
        double a = st.pop();
        st.push(apply(a, b, t));
    } else {
        st.push(Double.parseDouble(t));
    }
}
return st.pop();
```

Steps: O(n) – every token is processed once

Invalid expression: Pop on an empty stack, or not exactly one element left at the end

Example: `1 2 + 3 4 * - 9 /`

Token	Operation	Stack (from the bottom)
1	push	1
2	push	1 2
+	1 + 2	3
3	push	3 3
4	push	3 3 4
*	3 · 4	3 12
–	3 – 12	–9
9	push	–9 9
/	–9 / 9	**–1**

In infix form: ((1 + 2) – 3 · 4) / 9 = –1

From infix to postfix: the shunting-yard algorithm

E. W. Dijkstra (1961), simplified version – the **operators** go onto the stack:

1. **Operand** → straight to the output
2. `(` → onto the stack
3. `)` → everything from the stack to the output up to the first `(`; the `(` is discarded
4. **Operator** → while the top of the stack holds an operator **at least as strong**, move it to the output; then push the new operator
5. At the end, move the rest of the stack to the output

Infix	Postfix
`3 + 4 * 2`	`3 4 2 * +`
`(3 + 4) * 2`	`3 4 + 2 *`
`(a + b) * (c - d)`	`a b + c d - *`

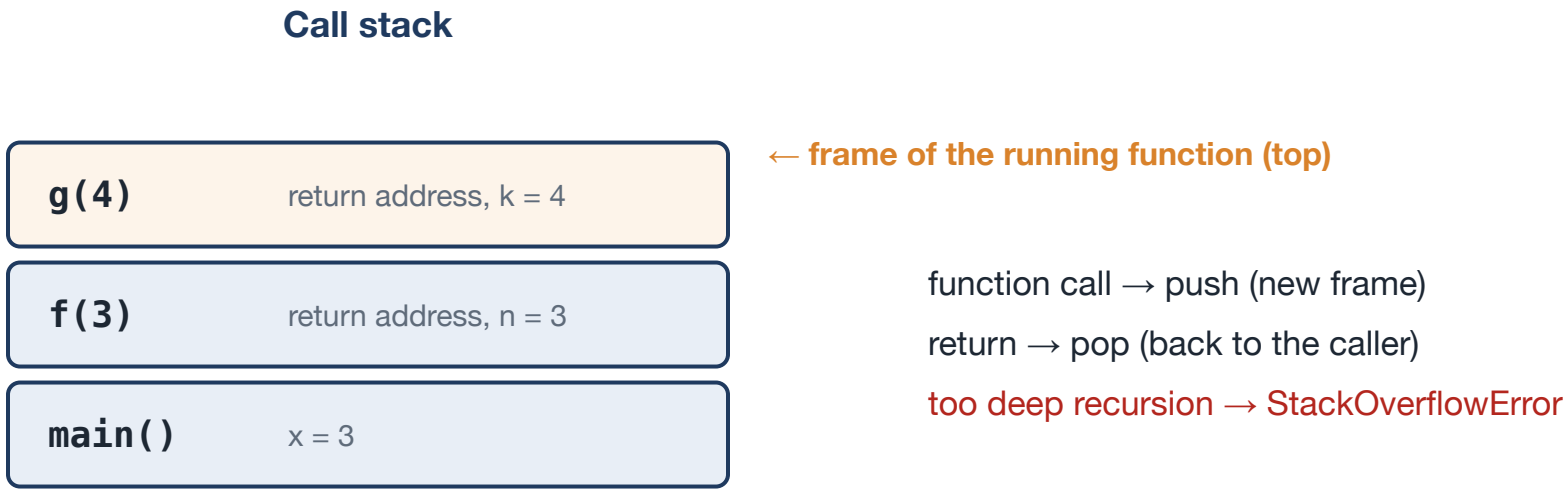
Application: checking brackets

Are the brackets correct? `{[()()]}` ✓ · `([)]` ✗ · `((a + b)` ✗

```
Function Balanced(S)
  Create
  For every character c
    If c is an opening bracket then Push(c)
    else if c is a closing bracket then
      If IsEmpty then Return false          // no opening pair
      If Pop is not the pair of c then Return false
    End if
  End for
  Return IsEmpty                            // anything left unclosed?
End function
```

Applications: compilers, text editors, checking HTML/XML · **O(n)**

Application: the call stack



Every function call pushes a **frame** (local variables, parameters, return address) onto the stack – this is what makes **recursion** work (week 12).

Further applications

- **Undo:** the state before every modification is pushed onto a stack
- **Browser history:** “Back” = pop, new page = push
- **Backtracking:** maze, sudoku, 8 queens – the decisions so far are on the stack
- **Depth-first traversal** of trees and graphs
- **Base conversion:** the remainders in reverse order → stack
- **Reversing text:** push every character, then pop them all

Summary

- **Stack:** LIFO – only the top is accessible
- Operations: **Push, Pop, Top, IsEmpty?, IsFull?** – all **O(1)**
- **Array implementation:** a `top` index at the end of the array; overflow is possible with a fixed size
- Java: `ArrayDeque` · Go: slice (`append` , `s[:len(s)-1]`)
- **Postfix evaluation:** one pass, one stack, O(n) – the right operand comes out first!
- Applications: brackets, undo, call stack, backtracking

Next week: the queue – FIFO principle, circular queue, double-ended queue, priority queue.

Review questions

1. What does it print? `push(1); push(2); pop(); push(3); push(4); pop(); pop(); pop();` (pop prints the element)
2. Evaluate with a stack, writing down the state of the stack after every step: `6 2 / 3 1 - *`
3. Convert to postfix form: `a * (b + c) - d / e`
4. What does the following code print for `n = 50`? What does it do in general?

```
while (n > 0) { s.push(n % 2); n = n / 2; } while (!s.isEmpty())
print(s.pop());
```

5. True or false: in the array implementation, Pop takes O(n) steps because the elements must be shifted.

Practice

✎ Demo programs

Program	Java	Go	What does it show?
Array stack	`ArrayStack.java`	`w07/arraystack`	push, pop, overflow/underflow, generic stack
Postfix	`Postfix.java`	`w07/postfix`	evaluation, printing the state of the stack
Infix → postfix	`InfixToPostfix.java`	`w07/infixtopostfix`	shunting-yard algorithm
Brackets	`Brackets.java`	`w07/brackets`	checking `()`, `[]`, `{}`

✎ Exercises

1. **Binary form:** convert a decimal number to binary with a stack!
2. **Reversing:** reverse a text with a stack! (In Go, Chinese text too!)
3. **Min-stack:** implement a stack whose `min()` operation is O(1) too!
4. **Undo:** simulate the undo function of a text editor!
5. **Code tracing:** letter = push, `*` = pop and print. What does it print? `D A T * A * S * T R * U C * T`
6. **Neighbouring pairs:** repeatedly delete neighbouring equal letter pairs! (`"abbaca" → "ca"`)

Solutions: `Exercises07.java` , `w07/exercises`

✎ Homework

1. Extend the postfix evaluator with the `^` (power) and `%` (remainder) operators!
2. Extend the infix → postfix converter for **multi-digit numbers** (e.g. `12 + 345 * 6`)!
3. Write a program that implements a **queue** (FIFO) using two stacks! How many steps do the operations take?
4. Write a program that checks whether the tags (`<b>…</b>`, `<i>…</i>`) in an HTML text are correctly nested!