# Tutorial 4

## Core data structures — practice and benchmarking

Jan Bures

18YZALG – Basics of Algorithmization, Summer Semester 2026

# Today

- **Recap:** operations $\rightarrow$ data structure.
- **Practice:** three real-ish problems.
- **Benchmarking:** avoid common timing mistakes.
- **Mini-projects:** what your demo must contain.

# Recap in one slide

| Structure | What it stores | Fast operations | Typical use |
|---|---|---|---|
| List | sequence | index, append/pop end | logs, arrays, simple stacks |
| Stack | LIFO | push/pop top | undo, parsing, DFS-like tasks |
| Queue | FIFO | push back / pop front | scheduling, BFS-like tasks |
| Set | unique keys | membership, add/remove | dedup, "seen" items |
| Dict | key $\to$ value | lookup/update | lookup tables, grouping, caching |

**Rule of thumb**

Choose the structure that makes the **most frequent operation** cheap.

# Recap in one slide

| Structure | What it stores | Fast operations | Typical use |
|---|---|---|---|
| List | sequence | index, append/pop end | logs, arrays, simple stacks |
| Stack | LIFO | push/pop top | undo, parsing, DFS-like tasks |
| Queue | FIFO | push back / pop front | scheduling, BFS-like tasks |
| Set | unique keys | membership, add/remove | dedup, “seen” items |
| Dict | key $\to$ value | lookup/update | lookup tables, grouping, caching |

**Rule of thumb**

Choose the structure that makes the **most frequent operation** cheap.

# Case study 1: “Have I seen this ID already?”

- You process a stream of IDs.
- Task: report the first repeated ID (or “none”).
- Constraints: up to $10^6$ IDs.

**Think**

What operation dominates? **Membership check** for “seen” elements.

**Expected answer**

Use a **set** of seen IDs (average $O(1)$ membership).

# Case study 1: "Have I seen this ID already?"

- You process a stream of IDs.
- Task: report the first repeated ID (or "none").
- Constraints: up to $10^6$ IDs.

**Think**

What operation dominates? **Membership check** for "seen" elements.

**Expected answer**

Use a **set** of seen IDs (average $O(1)$ membership).

# Case study 1: “Have I seen this ID already?”

- You process a stream of IDs.
- Task: report the first repeated ID (or “none”).
- Constraints: up to $10^6$ IDs.

**Think**

What operation dominates? **Membership check** for “seen” elements.

**Expected answer**

Use a **set** of seen IDs (average $O(1)$ membership).

## Case study 1: “Have I seen this ID already?”

- You process a stream of IDs.
- Task: report the first repeated ID (or “none”).
- Constraints: up to $10^6$ IDs.

**Think**

What operation dominates? **Membership check** for “seen” elements.

**Expected answer**

Use a **set** of seen IDs (average $O(1)$ membership).

# Case study 1: “Have I seen this ID already?”

- You process a stream of IDs.
- Task: report the first repeated ID (or “none”).
- Constraints: up to $10^6$ IDs.

**Think**

What operation dominates? **Membership check** for “seen” elements.

**Expected answer**

Use a **set** of seen IDs (average $O(1)$ membership).

# Case study 1: “Have I seen this ID already?”

- You process a stream of IDs.
- Task: report the first repeated ID (or “none”).
- Constraints: up to $10^6$ IDs.

**Think**

What operation dominates? **Membership check** for “seen” elements.

**Expected answer**

Use a **set** of seen IDs (average $O(1)$ membership).

# Case study 2: "Undo / Redo"

- Commands arrive: `type`, `delete`, `undo`, `redo`.
- You need to support undo/redo efficiently.
- "Undo" must revert the last command.

**Think**

This is a **last-in-first-out** story.

**Expected answer**

Two **stacks**: one for undo history, one for redo history.

# Case study 2: “Undo / Redo”

- Commands arrive: `type`, `delete`, `undo`, `redo`.
- You need to support undo/redo efficiently.
- “Undo” must revert the last command.

**Think**

This is a **last-in-first-out** story.

**Expected answer**

Two **stacks**: one for undo history, one for redo history.

# Case study 2: “Undo / Redo”

- Commands arrive: `type`, `delete`, `undo`, `redo`.
- You need to support undo/redo efficiently.
- “Undo” must revert the last command.

**Think**

This is a **last-in-first-out** story.

**Expected answer**

Two **stacks**: one for undo history, one for redo history.

# Case study 2: "Undo / Redo"

- Commands arrive: `type`, `delete`, `undo`, `redo`.
- You need to support undo/redo efficiently.
- "Undo" must revert the last command.

**Think**

This is a **last-in-first-out** story.

**Expected answer**

Two **stacks**: one for undo history, one for redo history.

# Case study 2: "Undo / Redo"

- Commands arrive: `type`, `delete`, `undo`, `redo`.
- You need to support undo/redo efficiently.
- "Undo" must revert the last command.

**Think**

This is a **last-in-first-out** story.

**Expected answer**

Two **stacks**: one for undo history, one for redo history.

# Case study 2: "Undo / Redo"

- Commands arrive: `type`, `delete`, `undo`, `redo`.
- You need to support undo/redo efficiently.
- "Undo" must revert the last command.

**Think**

This is a **last-in-first-out** story.

**Expected answer**

Two **stacks**: one for undo history, one for redo history.

# Case study 3: "Requests in arrival order"

- Requests arrive over time.
- You always handle the oldest waiting request.
- Sometimes there are bursts: $10^5$ requests in a second.

**Expected answer**

Use a **queue** (in Python: `collections.deque`).

# Case study 3: “Requests in arrival order”

- Requests arrive over time.
- You always handle the oldest waiting request.
- Sometimes there are bursts: $10^5$ requests in a second.

**Expected answer**

Use a **queue** (in Python: `collections.deque`).

# Case study 3: “Requests in arrival order”

- Requests arrive over time.
- You always handle the oldest waiting request.
- Sometimes there are bursts: $10^5$ requests in a second.

**Expected answer**

Use a **queue** (in Python: `collections.deque`).

# Case study 3: "Requests in arrival order"

- Requests arrive over time.
- You always handle the oldest waiting request.
- Sometimes there are bursts: $10^5$ requests in a second.

**Expected answer**

Use a **queue** (in Python: `collections.deque`).

# Case study 3: “Requests in arrival order”

- Requests arrive over time.
- You always handle the oldest waiting request.
- Sometimes there are bursts: $10^5$ requests in a second.

**Expected answer**

Use a **queue** (in Python: `collections.deque`).

# Bonus: Bloom filter (probabilistic set)

## Idea (what it is)

- Keep a bit-array $B[0..m-1]$ (initially all zeros).
- **Insert** item $x$: compute $k$ hashes $h_1(x), \ldots, h_k(x)$ and set those bits in $B$ to 1.
- **Query** membership:
  - if any checked bit is 0 $\Rightarrow$ definitely not present,
  - if all checked bits are 1 $\Rightarrow$ maybe present (can be a false positive).

## When it shines (why it's cool)

- Huge membership filters when memory is tight.
- As a cheap **pre-check** before an expensive operation (disk / DB / network).
- Best when most answers are "no" and you want those fast.

## Takeaway

Bloom filter answers: **"no" is certain**, **"yes" is uncertain** (fast and memory-light).

# Bonus: Trie (prefix tree)

## Idea (what it is)

- A tree where edges are characters (or digits).
- A path from the root spells a key; nodes mark **end-of-word**.
- Search/insert time is $\mathcal{O}(\text{len}(\text{key}))$ (independent of how many keys you store).

## What it makes cheap

- **Prefix queries**: "is there any word starting with `pre`?"
- **Autocomplete**: after reaching a prefix node, list a few completions.
- **Longest prefix match**: "how much of this string is a known prefix?"

## Trade-off

Very fast for prefix tasks, but can use **more memory** than a flat set/dict for random strings.

# Bonus: Disjoint Set Union (Union–Find)

## Problem it solves

- Maintain a partition of elements into **groups**.
- Support queries while groups keep merging:
  - "Are $a$ and $b$ in the same group?"
  - "Merge the groups containing $a$ and $b$."

## Tiny API (how you use it)

- `find(x)` $\rightarrow$ representative of $x$'s group
- `union(a,b)` merges two groups
- With **path compression** + **union by rank**, operations are *almost* constant time in practice.

## Where you'll meet it later

Dynamic connectivity, clustering/grouping, and some graph algorithms.

# Closing

**The mindset**

Know your operations, pick the structure, verify with a benchmark for your context.

- Next topics later: sorting, recursion, graphs.
- **These data structures show up everywhere in those topics**.