Basics of Algorithmization

Tutorials 3–4 Mini-Projects

Lecture title: Core data structures — mini-project options

Choose **one** mini-project per group and present your solution in the tutorial session.

Overview

Quick facts

Item	Description
Who	Student groups (recommended 2–3 students)
When	Tutorial session 4
What	Short live demo (approx. 10–15 minutes per group)
We practice	Data structure choice, correctness tests, informal time/memory complexity, benchmarking
Grading	Credit to the continuous assessment (tutorial part of the course)

What every group must include

- **Problem statement:** input, output, and edge cases (5–10 lines).
- **Operation profile:** what operations dominate? (membership / push-pop / enqueue-dequeue / key lookup / inserts).
- **At least two approaches:** same task, different data structure choices.
- **Complexity:** informal Big-O time and memory discussion for each approach.
- **Benchmark:** measure runtime for **5+ input sizes** and interpret the trend (a table is enough; a plot is optional but welcomed).
- **Correctness checks:** should answer the question "is my program doing what it is supposed to do?", a small test suite (simple assert tests are sufficient).
- **Conclusion:** when would you use which approach, and why?

Submission format

- No files needed to submit beforehand.
- Only deliverable is the **live demonstration**.
- Use slides, a notebook, terminal demo — anything is fine, as long as you cover the required content.

Live demonstration checklist (structure)

Aim for a concise demo. You can split speaking roles inside the group.

Suggested structure that fits *data-structure* mini-projects well:

- problem statement + example I/O + edge cases.

- operation profile (what do we need to do many times?).
- approaches (Approach A baseline, Approach B improvement, optional Approach C).
- complexity (time + memory) per approach.
- correctness (run tests or show representative tests).
- benchmarking (methodology + table + interpretation).
- final takeaway (trade-offs, when each wins).

Assessment (20 points)

Criterion	Points	What we look for
Correctness & tests	0–5	Works on edge cases; tests are clear and actually run
DS choice & explanation	0–5	Operation profile makes sense; choices are justified
Benchmark & interpretation	0–5	5+ sizes; fair timing; conclusions match results
Demo clarity	0–5	Clear structure, readable visuals, good pacing

Mini-project options

Pick **exactly one**. Send me an email with what your group has picked - it is first come, first served.

1) Playlist similarity (List scanning vs Set operations)

Goal. Given two users' playlists (lists of song IDs), compute:

- number of common songs,
- the union size,
- and a similarity score such as Jaccard: `|A ∩ B| / |A ∪ B|`.

Required content

- Define inputs (IDs as integers/strings) and output format (counts + score).
- Include edge cases: empty playlists, identical playlists, no overlap, many duplicates inside one playlist.
- Implement at least two methods and ensure identical outputs on the same tests.
- Benchmark across 5+ playlist sizes and vary overlap ratio (e.g., 0%, 10%, 50%).

Suggested approaches to compare

- Method 1 (baseline): nested loops / list scanning to count overlaps.
- Method 2: convert to sets and use `intersection` / `union`.
- Optional Method 3: keep playlists as lists but also maintain a set for faster repeated comparisons.

Demo focus

- Explain why set operations are "linear-ish" while the baseline tends to become quadratic.
- Show a benchmark table where the set method scales smoothly.

Stretch goals (optional)

- Discuss how duplicates in playlists should be treated (set ignores them) and why that matters.

2) Tiny URL router (List of pairs vs Dict)

Goal. Given route definitions `(path, handler_id)` and queries `path`, return the handler or `404`.

Required content

- Specify `n = number of routes` and `q = number of queries`.
- Include edge cases: missing path, duplicated path definitions (define your rule), empty routing table.

- Compare at least two strategies, one of which includes preprocessing.
- Benchmark for different `q/n` ratios (few queries vs many queries).

Suggested approaches to compare

- Method 1 (baseline): store routes in a list of pairs; linear scan per query.
- Method 2: preprocess into a dict `path -> handler_id`; dict lookup per query.
- Optional Method 3: sort routes once + binary search per query.

Demo focus

- Explain the trade-off: preprocessing time + memory vs fast queries.
- Show a break-even story: when does preprocessing pay off?

Stretch goals (optional)

- Support “prefix routes” like `/api/*` and discuss why the dict approach breaks.

3) Round-robin scheduler (List vs Deque)

Goal. Simulate a simple CPU scheduler. Each task has a remaining work counter. Repeatedly:

- take the oldest task,
- run it for a fixed quantum `q` (decrease remaining work),
- if it is not finished, put it back to the end.

Required content

- Define inputs (task list + quantum) and outputs (e.g., completion order + total steps).
- Include edge cases: empty task list, quantum larger than remaining work, tasks that finish immediately.
- Implement at least two queue backends and compare them.
- Benchmark for increasing number of tasks and total work (5+ sizes).

Suggested approaches to compare

- Method 1 (baseline): represent the ready queue as a list; take with `pop(0)` and re-add with `append`.
- Method 2: represent the ready queue as `collections.deque`; take with `popleft()` and re-add with `append()`.

Demo focus

- Explain the hidden cost of “shifting” elements in a list.

- Show that the deque approach scales smoothly as you increase the workload.

Stretch goals (optional)

- Allow tasks to arrive while the simulation is running.

4) Bracket checker (Repeated replacement vs Stack)

Goal. Given a string containing `()[]{}` brackets, decide if it is **properly nested**.

Required content

- Define input/output and edge cases: empty string, odd length, early closing bracket.
- Implement at least two methods and ensure identical results on the same tests.
- Benchmark across 5+ input sizes. Include a “hard” case (deep nesting) and a “noisy” case (random chars).

Suggested approaches to compare

- Method 1 (baseline): repeatedly remove `()`, `[]`, `{}` from the string until it stops changing.
- Method 2: one-pass stack scan (push opening brackets, pop and match on closing).

Demo focus

- Explain why the replacement method can be surprisingly slow.
- Show that the stack solution is linear in the string length.

Stretch goals (optional)

- Also compute the maximum nesting depth (extra information you can get “for free” with a stack).

5) Course roster builder (Nested lists vs Dict-of-sets)

Goal. You receive registrations `(student_id, course_code)`. Build rosters: for each course, a list of **unique** students.

Required content

- Define whether rosters must be sorted or can be in insertion order.
- Include edge cases: repeated registrations, many courses with few students, one course with many students.
- Implement at least two methods and ensure identical results on the same tests.
- Benchmark for increasing number of registrations.

Suggested approaches to compare

- Method 1 (baseline): for each registration, find the course entry by scanning; then scan that course list to avoid duplicates.
- Method 2: dict `course -> set of students` during processing; convert to list at the end.

Demo focus

- Explain why the baseline becomes quadratic-ish as data grows.
- Highlight the “composition” idea: dict for grouping + set for uniqueness.

Stretch goals (optional)

- Keep insertion order of students while still preventing duplicates (hint: set + list).

Closing note

Keep it simple. The goal is not “maximum performance at any cost” — it is learning how **data structure choice** and **complexity reasoning** translate into real runtime behavior.