Tutorial 3

Core data structures

Jan Bures

18YZALG – Basics of Algorithmization, Summer Semester 2026

Today

- Core idea: choose the structure that matches your operations
- Arrays / lists (dynamic arrays)
- Stack and queue (LIFO / FIFO)
- # Hash set and hash map / dictionary
- Benchmarking that is actually meaningful

Today

- Core idea: choose the structure that matches your operations
- Arrays / lists (dynamic arrays)
- Stack and queue (LIFO / FIFO)
- # Hash set and hash map / dictionary
- Benchmarking that is actually meaningful

Today

- Core idea: choose the structure that matches your operations
- Arrays / lists (dynamic arrays)
- Stack and queue (LIFO / FIFO)
- # Hash set and hash map / dictionary
- Benchmarking that is actually meaningful

Today

- Core idea: choose the structure that matches your operations
- Arrays / lists (dynamic arrays)
- Stack and queue (LIFO / FIFO)
- # Hash set and hash map / dictionary
- Benchmarking that is actually meaningful

Today

- Core idea: choose the structure that matches your operations
- Arrays / lists (dynamic arrays)
- Stack and queue (LIFO / FIFO)
- # Hash set and hash map / dictionary
- Benchmarking that is actually meaningful

Why data structures matter

- An algorithm is a **plan**. A data structure is the **storage + operations** you get cheaply.
- The same "idea" can become fast or slow depending on **how you store data**.
- Rule of thumb: **pick by operations**.

Mental model

Think: what do I do the most? (indexing? push/pop? membership? key $\rightarrow$ value lookup?)

Why data structures matter

- An algorithm is a **plan**. A data structure is the **storage + operations** you get cheaply.
- The same "idea" can become fast or slow depending on **how you store data**.
- Rule of thumb: **pick by operations**.

Mental model

Think: what do I do the most? (indexing? push/pop? membership? key $\to$ value lookup?)

Why data structures matter

- An algorithm is a **plan**. A data structure is the **storage + operations** you get cheaply.
- The same "idea" can become fast or slow depending on **how you store data**.
- Rule of thumb: **pick by operations**.

Mental model

Think: what do I do the most? (indexing? push/pop? membership? key $\rightarrow$ value lookup?)

Why data structures matter

- An algorithm is a **plan**. A data structure is the **storage + operations** you get cheaply.
- The same "idea" can become fast or slow depending on **how you store data**.
- Rule of thumb: **pick by operations**.

Mental model

Think: what do I do the most? (indexing? push/pop? membership? key $\rightarrow$ value lookup?)

Why data structures matter

- An algorithm is a **plan**. A data structure is the **storage + operations** you get cheaply.
- The same "idea" can become fast or slow depending on **how you store data**.
- Rule of thumb: **pick by operations**.

Mental model

Think: what do I do the most? (indexing? push/pop? membership? key $\rightarrow$ value lookup?)

Operation profile (the question to ask first)

For your problem, write 3 lines

1. What operations do I need? (add / remove / search / iterate / ...)
2. How often? (once, per input element, many queries, streaming)
3. What constraints? (order matters? duplicates allowed? memory?)

Outcome

You get a short "operation profile" that maps to a data structure choice.

Operation profile (the question to ask first)

For your problem, write 3 lines

1. What operations do I need? (add / remove / search / iterate / ...)
2. How often? (once, per input element, many queries, streaming)
3. What constraints? (order matters? duplicates allowed? memory?)

Outcome

You get a short “operation profile” that maps to a data structure choice.

Arrays / Lists (dynamic arrays)

- Store elements in a contiguous memory block (conceptually).
- Great for: **indexing** (random access), **iteration**, appending.
- Weak for: inserting/removing at the front or middle.

Typical operations

- $A[i]$
- `append(x)`
- `pop()`
- `insert(i,x)`

Common patterns

- "I need the i-th element quickly".
- "I process items in order".
- "I append new items as they arrive".

Pitfall

`insert(0, x)` and `pop(0)` shift the whole array.

Arrays / Lists (dynamic arrays)

- Store elements in a contiguous memory block (conceptually).
- Great for: **indexing** (random access), **iteration**, appending.
- Weak for: inserting/removing at the front or middle.

Typical operations

- $A[i]$
- `append(x)`
- `pop()`
- `insert(i,x)`

Common patterns

- "I need the i-th element quickly".
- "I process items in order".
- "I append new items as they arrive".

Pitfall

`insert(0, x)` and `pop(0)` shift the whole array.

List benchmark: append vs insert at front

Micro-benchmark (example numbers)

Task	N	Time
Append N items	5000	0.25 ms
Insert at front N items	5000	4.14 ms

- Appending is fast because the list grows with **occasional resizing** (amortized $O(1)$).
- Prepending is slower because every insert shifts many elements ($O(n)$ each).
- Takeaway: if you need fast front operations, use a **queue**.

List benchmark: append vs insert at front

Micro-benchmark (example numbers)

Task	N	Time
Append N items	5000	0.25 ms
Insert at front N items	5000	4.14 ms

- Appending is fast because the list grows with **occasional resizing** (amortized $O(1)$).
- Prepending is slower because every insert shifts many elements ($O(n)$ each).
- Takeaway: if you need fast front operations, use a **queue**.

List benchmark: append vs insert at front

Micro-benchmark (example numbers)

Task	N	Time
Append N items	5000	0.25 ms
Insert at front N items	5000	4.14 ms

- Appending is fast because the list grows with **occasional resizing** (amortized $O(1)$).
- Prepending is slower because every insert shifts many elements ($O(n)$ each).
- Takeaway: if you need fast front operations, use a **queue**.

List benchmark: append vs insert at front

Micro-benchmark (example numbers)

Task	N	Time
Append N items	5000	0.25 ms
Insert at front N items	5000	4.14 ms

- Appending is fast because the list grows with **occasional resizing** (amortized $O(1)$).
- Prepending is slower because every insert shifts many elements ($O(n)$ each).
- Takeaway: if you need fast front operations, use a **queue**.

Stack (LIFO)

- **Last-In, First-Out**
- Operations: `push`, `pop`, `peek`
- Great for: **undo**, parsing, depth-first exploration, managing nested structure

Tiny example

push(A), push(B), push(C)

pop() → C
pop() → B

Implementation (Python)

Use a list: `append` for push and `pop()` for pop.

Stack (LIFO)

- **Last-In, First-Out**
- Operations: `push`, `pop`, `peek`
- Great for: **undo**, parsing, depth-first exploration, managing nested structure

Implementation (Python)

Use a list: `append` for push and `pop()` for pop.

Tiny example

push(A), push(B), push(C)

pop() $\rightarrow$ C
pop() $\rightarrow$ B

Stack example: simplify a path

Idea

Scan path parts. Normal names push, `..` pops, `.` is ignored.

Code

```
def simplify(path):
    stack = []
    for part in path.split('/'):
        if part in ('', '.'):
            continue
        if part == '..':
            if stack: stack.pop()
        else:
            stack.append(part)
    return '/' + '/'.join(stack)

simplify('/a/b/../c/./d')  # '/a/c/d'
```

Queue (FIFO)

- **First-In, First-Out**
- Operations: `enqueue`, `dequeue`
- Great for: task scheduling, buffering, simulations, breadth-first processes

Implementation (Python)

Use standard queue or texttttdeque (more flexible).

Tiny example

enqueue(A), enqueue(B), enqueue(C)

dequeue() $\rightarrow$ A
dequeue() $\rightarrow$ B

Queue (FIFO)

- **First-In, First-Out**
- Operations: `enqueue`, `dequeue`
- Great for: task scheduling, buffering, simulations, breadth-first processes

Implementation (Python)

Use standard `queue` or `textttdeque` (more flexible).

Tiny example

enqueue(A), enqueue(B), enqueue(C)

dequeue() $\rightarrow$ A
dequeue() $\rightarrow$ B

Queue benchmark: list vs deque (more native in Python)

Pop all items (example numbers)

m pops	list pop(0)	deque popleft	speedup
2000	0.68 ms	0.18 ms	3.9 ×
5000	3.80 ms	0.69 ms	5.5 ×
10000	17.9 ms	0.98 ms	18 ×
20000	82.8 ms	1.78 ms	46 ×

- `pop(0)` shifts all remaining elements: $O(n)$ per pop.
- `deque.popleft()` is $O(1)$.
- Trend: the speed gap grows quickly with input size.

Queue benchmark: list vs deque (more native in Python)

Pop all items (example numbers)

m pops	list pop(0)	deque popleft	speedup
2000	0.68 ms	0.18 ms	3.9 ×
5000	3.80 ms	0.69 ms	5.5 ×
10000	17.9 ms	0.98 ms	18 ×
20000	82.8 ms	1.78 ms	46 ×

- `pop(0)` shifts all remaining elements: $O(n)$ per pop.
- `deque.popleft()` is $O(1)$.
- Trend: the speed gap grows quickly with input size.

Queue benchmark: list vs deque (more native in Python)

Pop all items (example numbers)

m pops	list pop(0)	deque popleft	speedup
2000	0.68 ms	0.18 ms	3.9 ×
5000	3.80 ms	0.69 ms	5.5 ×
10000	17.9 ms	0.98 ms	18 ×
20000	82.8 ms	1.78 ms	46 ×

- `pop(0)` shifts all remaining elements: $O(n)$ per pop.
- `deque.popleft()` is $O(1)$.
- Trend: the speed gap grows quickly with input size.

Queue benchmark: list vs deque (more native in Python)

Pop all items (example numbers)

m pops	list pop(0)	deque popleft	speedup
2000	0.68 ms	0.18 ms	3.9 ×
5000	3.80 ms	0.69 ms	5.5 ×
10000	17.9 ms	0.98 ms	18 ×
20000	82.8 ms	1.78 ms	46 ×

- `pop(0)` shifts all remaining elements: $O(n)$ per pop.
- `deque.popleft()` is $O(1)$.
- Trend: the speed gap grows quickly with input size.

Hashing at a glance

- A **hash function** maps a key to an integer "bucket".
- A **hash table** stores items by bucket, so lookup is fast on average.
- Two main forms:
  - **Set**: store keys only (membership, uniqueness)
  - **Map / dict**: store key → value (lookup by key)
- Worst-case exists, but in practice average $O(1)$ is why we love them.

Hashing at a glance

- A **hash function** maps a key to an integer "bucket".
- A **hash table** stores items by bucket, so lookup is fast on average.
- Two main forms:
  - **Set**: store keys only (membership, uniqueness)
  - **Map / dict**: store key → value (lookup by key)
- Worst-case exists, but in practice average $O(1)$ is why we love them.

Hashing at a glance

- A **hash function** maps a key to an integer "bucket".
- A **hash table** stores items by bucket, so lookup is fast on average.
- Two main forms:
  - **Set**: store keys only (membership, uniqueness)
  - **Map / dict**: store key $\rightarrow$ value (lookup by key)
- Worst-case exists, but in practice average $O(1)$ is why we love them.

Hashing at a glance

- A **hash function** maps a key to an integer "bucket".
- A **hash table** stores items by bucket, so lookup is fast on average.
- Two main forms:
  - **Set**: store keys only (membership, uniqueness)
  - **Map / dict**: store key $\rightarrow$ value (lookup by key)
- Worst-case exists, but in practice average $O(1)$ is why we love them.

Hashing at a glance

- A **hash function** maps a key to an integer "bucket".
- A **hash table** stores items by bucket, so lookup is fast on average.
- Two main forms:
  - **Set**: store keys only (membership, uniqueness)
  - **Map / dict**: store key $\rightarrow$ value (lookup by key)
- Worst-case exists, but in practice average $O(1)$ is why we love them.

Hashing at a glance

- A **hash function** maps a key to an integer "bucket".
- A **hash table** stores items by bucket, so lookup is fast on average.
- Two main forms:
  - **Set**: store keys only (membership, uniqueness)
  - **Map / dict**: store key $\rightarrow$ value (lookup by key)
- Worst-case exists, but in practice average $O(1)$ is why we love them.

Hash set (unique items + fast membership)

- Stores **unique** items (duplicates collapse).
- Operations: `add`, `remove`, `x in S`.
- Great for: membership queries, removing duplicates, visited tracking.

Set operations are also useful

union, intersection, difference

When not to use

- You need **order**
- You need **duplicates**
- You need "the smallest" / "the next" (needs ordering)

Hash set (unique items + fast membership)

- Stores **unique** items (duplicates collapse).
- Operations: `add`, `remove`, `x in S`.
- Great for: membership queries, removing duplicates, visited tracking.

When not to use

- You need **order**
- You need **duplicates**
- You need "the smallest" / "the next" (needs ordering)

Set operations are also useful

union, intersection, difference

Hash set (unique items + fast membership)

- Stores **unique** items (duplicates collapse).
- Operations: `add`, `remove`, `x in S`.
- Great for: membership queries, removing duplicates, visited tracking.

When not to use

- You need **order**
- You need **duplicates**
- You need "the smallest" / "the next" (needs ordering)

Set operations are also useful

union, intersection, difference

Hash set (unique items + fast membership)

- Stores **unique** items (duplicates collapse).
- Operations: `add`, `remove`, `x in S`.
- Great for: membership queries, removing duplicates, visited tracking.

When not to use

- You need **order**
- You need **duplicates**
- You need "the smallest" / "the next" (needs ordering)

Set operations are also useful

union, intersection, difference

Hash set (unique items + fast membership)

- Stores **unique** items (duplicates collapse).
- Operations: `add`, `remove`, `x in S`.
- Great for: membership queries, removing duplicates, visited tracking.

Set operations are also useful

union, intersection, difference

When not to use

- You need **order**
- You need **duplicates**
- You need "the smallest" / "the next" (needs ordering)

Membership benchmark: list vs set

q membership queries (example numbers)

n=q	list	set	speedup
2000	25.8 ms	0.41 ms	63×
5000	256 ms	0.56 ms	456×
10000	769 ms	1.26 ms	612×
20000	3021 ms	2.49 ms	1212×

Takeaway

If you do **many** membership checks, converting to a set is often a game-changer.

Membership benchmark: list vs set

q membership queries (example numbers)

n=q	list	set	speedup
2000	25.8 ms	0.41 ms	63×
5000	256 ms	0.56 ms	456×
10000	769 ms	1.26 ms	612×
20000	3021 ms	2.49 ms	1212×

Takeaway

If you do **many** membership checks, converting to a set is often a game-changer.

Hash map / Dictionary (key → value)

- Store pairs: `key` → `value`
- Great for: lookup tables, counting, grouping, memoization/caching
- Most common question: **do I already know the key?**

Typical operations

- `d[key]` / `d.get(key)`
- `d[key] = value`
- `key in d`

Pitfalls

- Keys must be hashable (immutable in Python)
- Order is insertion order (Python 3.7+), but do not rely on it unless allowed

Hash map / Dictionary (key → value)

- Store pairs: `key` → `value`
- Great for: lookup tables, counting, grouping, memoization/caching
- Most common question: **do I already know the key?**

Typical operations

- `d[key]` / `d.get(key)`
- `d[key] = value`
- `key in d`

Pitfalls

- Keys must be hashable (immutable in Python)
- Order is insertion order (Python 3.7+), but do not rely on it unless allowed

Hash map / Dictionary (key → value)

- Store pairs: `key` → `value`
- Great for: lookup tables, counting, grouping, memoization/caching
- Most common question: **do I already know the key?**

Typical operations

- `d[key]` / `d.get(key)`
- `d[key] = value`
- `key in d`

Pitfalls

- Keys must be hashable (immutable in Python)
- Order is insertion order (Python 3.7+), but do not rely on it unless allowed

Hash map / Dictionary (key → value)

- Store pairs: `key` → `value`
- Great for: lookup tables, counting, grouping, memoization/caching
- Most common question: **do I already know the key?**

Typical operations

- `d[key]` / `d.get(key)`
- `d[key] = value`
- `key in d`

Pitfalls

- Keys must be hashable (immutable in Python)
- Order is insertion order (Python 3.7+), but do not rely on it unless allowed

Hash map / Dictionary (key → value)

- Store pairs: `key` → `value`
- Great for: lookup tables, counting, grouping, memoization/caching
- Most common question: **do I already know the key?**

Typical operations

- `d[key]` / `d.get(key)`
- `d[key] = value`
- `key in d`

Pitfalls

- Keys must be hashable (immutable in Python)
- Order is insertion order (Python 3.7+), but do not rely on it unless allowed

Lookup benchmark: list of pairs vs dict

q lookups by key (example numbers)

Structure	n	q	Time
List of (key,value) pairs, linear scan	10000	20000	6297 ms
Dict, membership by key	10000	20000	4.13 ms

Interpretation

With many queries, preprocessing into a dict pays off extremely quickly.

Lookup benchmark: list of pairs vs dict

q lookups by key (example numbers)

Structure	n	q	Time
List of (key,value) pairs, linear scan	10000	20000	6297 ms
Dict, membership by key	10000	20000	4.13 ms

Interpretation

With many queries, preprocessing into a dict pays off extremely quickly.

Benchmarking

Checklist

- Measure the **operation**, not printing / input parsing.
- Use **multiple sizes** (show scaling).
- Use **repeats** and a robust summary (median).
- Pre-generate random data **outside** the timed region.
- Interpret trends, not just a single number.

Minimal Python template

```
import timeit, statistics, gc

def median_time(stmt, glb, repeat=7):
    gc.disable()
    try:
        t = timeit.Timer(stmt, globals=
     glb)
        return statistics.median(
            t.repeat(repeat=repeat,
     number=1)
        )
    finally:
        gc.enable()
```

Quick practice: choose the structure

1. You need fast **random access** by index and frequent appends.
2. You need to process events in **arrival order**.
3. You need to support **undo** of the last action.
4. You need to answer **“have we seen X?”** millions of times.
5. You need to map **ID** $\rightarrow$ **record** with fast lookups.

Quick practice: choose the structure

1. You need fast **random access** by index and frequent appends.
2. You need to process events in **arrival order**.
3. You need to support **undo** of the last action.
4. You need to answer **“have we seen X?”** millions of times.
5. You need to map **ID** $\rightarrow$ **record** with fast lookups.

Quick practice: choose the structure

1. You need fast **random access** by index and frequent appends.
2. You need to process events in **arrival order**.
3. You need to support **undo** of the last action.
4. You need to answer **“have we seen X?”** millions of times.
5. You need to map **ID** $\rightarrow$ **record** with fast lookups.

Quick practice: choose the structure

1. You need fast **random access** by index and frequent appends.
2. You need to process events in **arrival order**.
3. You need to support **undo** of the last action.
4. You need to answer **“have we seen X?”** millions of times.
5. You need to map **ID** $\rightarrow$ **record** with fast lookups.

Quick practice: choose the structure

1. You need fast **random access** by index and frequent appends.
2. You need to process events in **arrival order**.
3. You need to support **undo** of the last action.
4. You need to answer **“have we seen X?”** millions of times.
5. You need to map **ID** $\rightarrow$ **record** with fast lookups.

Quick practice: choose the structure

1. You need fast **random access** by index and frequent appends.
2. You need to process events in **arrival order**.
3. You need to support **undo** of the last action.
4. You need to answer **“have we seen X?”** millions of times.
5. You need to map **ID** $\rightarrow$ **record** with fast lookups.

Answers

List, Queue (deque), Stack, Set, Dict.

Summary (what to remember)

- Start with the **operation profile**.
- Lists: great for indexing + appends; bad at front/middle inserts.
- Stack: last-in-first-out (use list `append/pop`).
- Queue: first-in-first-out (use `deque` in Python).
- Set: uniqueness + fast membership.
- Dict: fast key $\rightarrow$ value lookup (and grouping / caching).
- Always confirm with a **small benchmark** when performance matters.