Linear Structures

Algorithmics study and revision notes

Jaak Vilo

2026-08-22

Contents

1 Why linear structures matter 1

2 Learning goals 2

3 Prerequisites 2

4 Abstract sequences and representation invariants 2

5 Arrays: positions are arithmetic 2

6 Linked lists: positions are paths 3

7 Stacks, queues, and deques 3

8 Circular buffers 4

9 Worked trace: wraparound 4

10 Dynamic arrays and amortised resizing 4

11 Complexity assumptions and trade-offs 5

12 Connections 5

13 Common mistakes 5

14 Self-check 6

15 Revision summary 6

16 Implementations and hands-on exploration 6

16.1 Small experiments 6

17 Sources and further study 6

References 6

1 Why linear structures matter

A sequence has a first item, a last item, and a position for every item between them. Two representations dominate: contiguous arrays make positions cheap to compute, while linked nodes make local restructuring cheap. Neither is universally superior; the workload determines which costs matter.

Linear structures also separate an interface from its implementation. Arrays or links can realise a stack, queue, or deque if they preserve the same observable behaviour.

2 Learning goals

After studying this note, you should be able to:

- distinguish a sequence ADT from an array or linked representation;
- derive array addresses and operation costs from contiguity;
- insert and delete linked nodes without losing reachability;
- implement stacks, queues, and deques through clear contracts;
- state and use the invariant of a circular buffer;
- prove dynamic-array append has constant amortised cost;
- explain why shrinking needs hysteresis; and
- choose a representation from an operation mix and memory model.

3 Prerequisites

You should understand arrays, records, references or pointers, loops, modular arithmetic, and O/Θ notation. The preceding note introduced geometric sums and amortised analysis.

4 Abstract sequences and representation invariants

A **sequence ADT** stores an ordered finite collection. A possible interface includes:

- `size()`;
- `get(i)` and `set(i,x)` for $0 \le i < n$;
- `insert(i,x)` for $0 \le i \le n$; and
- `remove(i)` for $0 \le i < n$.

These operations specify observable behaviour, not storage. An array list and a linked list can implement the same interface with different costs.

Every representation needs an invariant. An array list with logical size n and capacity c maintains

$$0 \le n \le c,$$

with live elements exactly in positions `0` through `n-1`. A linked list maintains a chain reachable from `head`, ending at `tail`; its recorded size must equal the number of reachable nodes. Operations must preserve these facts even for empty and one-element structures.

5 Arrays: positions are arithmetic

An array stores equal-sized elements contiguously. If its base address is b, each element occupies w bytes, and indexing starts at zero, then

$$\text{address}(A[i]) = b + iw.$$

One multiplication and addition locate any valid position, giving $\Theta(1)$ indexed access in the word-RAM model. Bounds checking, if present, is also constant time.

Contiguity gives strong spatial locality: a sequential scan tends to reuse cache lines and needs little metadata. Structural changes are less convenient. Inserting at position i shifts the suffix `A[i..n)` one position right, so it moves $n - i$ elements. Deletion similarly closes a gap.

Appending to a **fixed-capacity** array is $\Theta(1)$ only while an unused slot exists. Deleting the last element is $\Theta(1)$, but an array that is already full cannot grow in place merely because its last position is known.

6 Linked lists: positions are paths

A singly linked node contains a value and a `next` reference. A doubly linked node also contains `prev`. Nodes may be scattered through memory, so the ith node is found by following i links from the head: $\Theta(i)$ time.

Given a reference `p` to a singly linked node, insertion after it is local:

```
insert_after(p, x):
    q = new_node(x)
    q.next = p.next
    p.next = q
    if tail == p:
        tail = q
    size = size + 1
```

The assignment order matters. Saving `p.next` in `q.next` before changing `p.next` preserves the old suffix. Reversing these steps can lose every node after `p`.

Deletion in a singly linked list normally needs the predecessor:

```
delete_after(p):
    require p.next != null
    q = p.next
    p.next = q.next
    if tail == q:
        tail = p
    release(q)
    size = size - 1
```

Consequently, deleting the tail of a singly linked list with only `head` and `tail` references is $\Theta(n)$: its predecessor must be found. It is $\Theta(1)$ only when that predecessor is already supplied.

A doubly linked list can unlink a known node `q` in $\Theta(1)$ by joining `q.prev` to `q.next` and updating both reverse links. It pays for this convenience with another reference per node and more updates. Sentinel nodes can replace many null boundary cases with uniform links, but the sentinel itself must never be returned as data.

Memory management is part of the representation. Manual-memory implementations must not dereference a released node or leak an unlinked one. Garbage collection prevents explicit deallocation errors but does not restore nodes made unreachable by incorrect pointer updates.

An XOR list combines previous and next addresses in one word. It illustrates that representation can trade metadata for complicated navigation. It interacts poorly with garbage collection, memory safety, debugging, and modern portability, so it is not a default replacement for a doubly linked list.

7 Stacks, queues, and deques

A **stack** is last-in, first-out. `push(x)` adds an item, `top()` observes the newest item, and `pop()` removes and returns it. An array uses its logical end as the top; a singly linked implementation uses its head. Both give constant-time operations apart from occasional dynamic-array resizing.

A **queue** is first-in, first-out. `enqueue(x)` adds at the back, `front()` observes the oldest item, and `dequeue()` removes it. A linked queue stores both head and tail. An array queue should be circular; shifting every remaining element after each dequeue would make removal linear.

A **deque** supports insertion and removal at both ends. It can implement either a stack or queue and is useful in sliding-window and bidirectional search algorithms. “Deque” names the structure; `dequeue` commonly names the removal operation of a queue.

All interfaces need explicit underflow behaviour. They may reject `pop` on an empty stack, return an optional value, or raise an exception, but the contract must choose one.

8 Circular buffers

Let a circular queue have an array of positive capacity c, a `head` index, and a logical `size`. Logical element j is stored at

$$(head + j) \bmod c.$$

Tracking size makes the empty state `size == 0` and the full state `size == c`; no slot needs to remain unused.

```
enqueue(x):
    require size < c
    A[(head + size) mod c] = x
    size = size + 1

dequeue():
    require size > 0
    x = A[head]
    head = (head + 1) mod c
    size = size - 1
    return x
```

The invariant is that the `size` logical elements occupy the cyclic positions beginning at `head`, in queue order, and $0 \le size \le c$. Enqueue writes exactly the next free cyclic position; dequeue advances over exactly the oldest position. Both preserve the invariant in $\Theta(1)$ time.

9 Worked trace: wraparound

Start with capacity five, `head = 0`, and `size = 0`. Enqueue `3,6,7`, then dequeue once, then enqueue `5,2,9`.

Operation	Physical array	`head`	`size`	Logical queue
enqueue 3	`[3,_,_,_,_]`	0	1	`[3]`
enqueue 6	`[3,6,_,_,_]`	0	2	`[3,6]`
enqueue 7	`[3,6,7,_,_]`	0	3	`[3,6,7]`
dequeue	`[3,6,7,_,_]`	1	2	`[6,7]`
enqueue 5	`[3,6,7,5,_]`	1	3	`[6,7,5]`
enqueue 2	`[3,6,7,5,2]`	1	4	`[6,7,5,2]`
enqueue 9	`[9,6,7,5,2]`	1	5	`[6,7,5,2,9]`

The physical order is not the logical order; `head`, `size`, and modular indexing provide the interpretation.

The trace deliberately leaves the removed value `3` in its old physical slot until wraparound overwrites it. In a managed-memory implementation, clearing a removed slot may be necessary so that the queue does not retain an otherwise unreachable object; this changes neither the logical queue nor the asymptotic bound.

10 Dynamic arrays and amortised resizing

A dynamic array keeps the array invariant while changing capacity. On append, double a full capacity, copy the live elements, then write the new item:

```
append(x):
    if size == capacity:
        resize(max(1, 2 * capacity))
    A[size] = x
    size = size + 1
```

One resize can copy $\Theta(n)$ elements. Across n appends from an empty array, however, copies occur at capacities $1, 2, 4, \ldots$. Their total is less than $2n$, and the ordinary writes add another n. Thus the total cost is $\Theta(n)$ and append costs $\Theta(1)$ **amortised**, while remaining $\Theta(n)$ in the worst case for one operation.

An accounting view reaches the same result: charge each append a constant number of credits; one pays for its own write and the rest accumulate to pay for the next copy.

Shrinking immediately when the size falls below one-half can cause thrashing: alternating one insertion and one deletion near the threshold repeatedly reallocates. A common policy halves capacity only when size falls to at most one-quarter of capacity. Immediately after shrinking, the array is at most half full, so many updates are required before the next resize. Keep a small minimum capacity to avoid a zero-sized representation.

11 Complexity assumptions and trade-offs

The table separates locating a position from updating an already known location.

Structure and available references	Index	Append	Delete tail	Insert/delete at known local position
Fixed array with spare slot	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(n)$ shifts
Dynamic array	$\Theta(1)$	amortised $\Theta(1)$; worst $\Theta(n)$	amortised $\Theta(1)$; worst $\Theta(n)$ if it shrinks	$\Theta(n)$ shifts
Singly linked, head and tail	$\Theta(n)$	$\Theta(1)$	$\Theta(n)$	$\Theta(1)$ after known predecessor
Doubly linked, head and tail	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$ for known node

Searching unsorted values is $\Theta(n)$ in all four structures. Linked insertion "at index i" is not constant time unless the relevant node is already known; locating it costs $\Theta(i)$. Arrays usually have better locality and smaller metadata. Links support stable node references and local splicing but incur allocation, pointer, and cache costs.

12 Connections

- *Sorting, Searching, and Recurrences* uses sorted arrays for binary search and introduces skip lists as randomized linked dictionaries.
- Tree and graph traversals use stacks and queues as frontier structures.
- Hash tables often use dynamic arrays or linked collision chains internally.
- The geometric resizing proof is a first example of the amortised reasoning used again for advanced heaps and union-find.

13 Common mistakes

- Claiming linked insertion is $O(1)$ while including an uncounted search for the position.
- Claiming singly linked tail deletion is constant time without a predecessor.
- Updating a pointer before preserving the rest of a linked chain.
- Confusing physical array order with logical order in a circular queue.
- Allowing `size` to exceed `capacity` or failing to define empty-operation behaviour.
- Confusing amortised constant time with constant worst-case time.
- Shrinking at the same threshold used for growth and causing repeated reallocations.
- Ignoring cache locality, unused capacity, and per-node metadata.

14 Self-check

1. State the representation invariant of an array list.
2. Why must `q.next = p.next` precede `p.next = q` during singly linked insertion?
3. Under what exact condition is singly linked deletion $\Theta(1)$?
4. Trace three queue operations that wrap around a circular buffer.
5. Why does tracking only `head` and `tail` require an additional convention to distinguish full and empty?
6. Prove that doubling capacity causes fewer than $2n$ copied elements over n appends.
7. How does the one-quarter shrinking threshold prevent resize thrashing?
8. Which workload properties favour a linked list over a dynamic array?

15 Revision summary

- ADTs specify behaviour; arrays and linked nodes are alternative representations.
- Arrays compute positions arithmetically and benefit from locality, but middle updates shift suffixes.
- Linked updates are constant time only after the relevant node or predecessor is known.
- Stacks, queues, and deques are access disciplines independent of representation.
- Circular-buffer correctness depends on a precise mapping from logical to physical positions.
- Doubling gives constant amortised append, not constant worst-case append.
- Separate growth and shrink thresholds avoid repeated reallocations.
- Choose a representation from operations, assumptions, locality, and memory overhead.

16 Implementations and hands-on exploration

- CPython's `listobject.c` shows the over-allocation and shifting machinery behind Python lists. These capacity details belong to CPython and are not a promise of the Python language.
- Python `collections.deque` supplies thread-safe appends and pops at either end and optional bounded capacity. Indexing slows toward the middle, so it is not a drop-in replacement for array indexing.
- Rust standard collections compares `Vec`, the growable ring buffer `VecDeque`, and `LinkedList`, including operation costs. The documentation notes that contiguous structures usually beat linked lists unless their specific operations are needed.
- Java `ArrayDeque` implements a resizable deque suitable for stacks and queues. It is not thread-safe and rejects `null`, so these semantics must be included in a client contract.
- Linux kernel linked lists provide intrusive circular doubly linked lists in C, including splicing and safe traversal patterns. Payloads embed the link node, and concurrent mutation requires explicit locking or an appropriate RCU pattern.

16.1 Small experiments

1. Implement the capacity-five circular queue from the trace and compare it with a Python `deque`. Generate legal random enqueue/dequeue sequences while asserting contents, size, and the physical-position invariant after every operation; test the chosen empty and full error contracts separately.
2. Instrument a dynamic array that doubles from capacity 1. After every append, assert the representation invariant and that cumulative copies are less than twice the number of appends; print their ratio after each power of two.

17 Sources and further study

- Linear structures lecture deck, Jaak Vilo.
- (Cormen et al. 2022), Chapters 10 and 16.

References

Cormen, Thomas H., Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2022. *Introduction to Algorithms*. 4th ed. MIT Press.