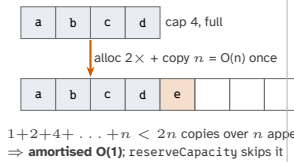
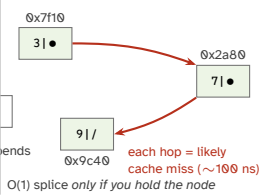


In one line: Every structure is a **memory layout** plus an **invariant** that makes some operations cheap and others expensive. Pick by the operation that must be fast, then check the constants: **contiguous memory + cache lines** beat pointer chasing at equal big-O on every modern CPU.

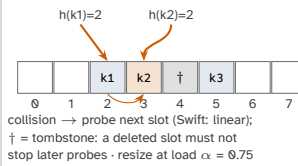
Array: append past capacity



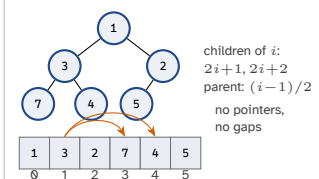
Linked list: pointer chasing



Hash table: open addressing



Min-heap = complete tree in array



How each one works

- Array** — one contiguous block: $\text{base} + i \cdot \text{stride} \Rightarrow O(1)$ index. Sequential scans ride the hardware **prefetcher**; a cache line (64 B x86, 128 B Apple silicon) brings 8–16 ints per miss. Insert/delete in the middle shifts $O(n)$. Swift grows geometrically ($2\times$).
- Linked list** — one heap allocation per node + next (and prev) pointers. Wins only for $O(1)$ splice at a *held* node (LRU, free lists) and stable node addresses. Finding the node is $O(n)$.
- Stack** LIFO (calls, DFS, undo) · **queue** FIFO (BFS, scheduling) · **deque** both ends $O(1)$ (sliding window, work stealing). Array queue with `removeFirst()` is $O(n)$ per dequeue → ring buffer (head index) or two stacks.
- Hash table** — $\text{hash}(\text{key}) \bmod m \rightarrow \text{bucket}$. Load $\alpha = n/m$. **Chaining**: bucket = list/tree, tolerates $\alpha > 1$, pointer-heavy. **Open addressing**: entries inline, probe on collision (linear/quadratic/double), cache-friendly, needs $\alpha < 1$ and tombstones. Resize = rehash every entry, $O(n)$ once.
- BST** — left < node < right; all ops $O(h)$. Sorted inserts make a list: $h = n$. **AVL**: subtree heights differ ≤ 1 , $h \leq 1.44 \log n \rightarrow$ faster reads, more rotations. **Red-black**: no red-red, equal black height, $h \leq 2 \log(n+1)$, ≤ 2 rotations per insert, ≤ 3 per delete → cheaper writes (std::map, Java TreeMap). Disk/DB: **B-tree**, fan-out ~ 100 s, one node = one page.
- Binary heap** — complete tree in an array, parent $\leq$ children. peek $O(1)$, push = append + sift-up $O(\log n)$, pop = move last to root + sift-down $O(\log n)$, **heapify** $O(n)$. Priority queues, Dijkstra, top-k.
- Trie** — node per prefix, children by character, isEnd flag. $O(L)$ per op, independent of key count; prefix queries a hash set can't do.
- Graph** — adjacency **list** $O(V+E)$ space, neighbours in $O(\deg)$; sparse graphs (almost all). **Matrix** $V \times V$ bits/weights, $O(1)$ "edge $u-v$?": dense graphs, Floyd-Warshall.
- Union-find** — parent array + path compression + union by rank $\rightarrow O(\alpha(n)) \approx O(1)$: components, Kruskal.

Big-O — average (worst)

structure	access	search	insert / delete	note
array	1	n	n; end: 1^{st}	sorted: search log n
linked list	n	n	1 at node	find the node: n
stack/queue/deque	ends 1	n	ends 1	array queue front: n
hash table	—	1 (n)	1^{st} (n)	hashing key = $O(\text{len})$
BST, plain	—	$\log n$ (n)	$\log n$ (n)	sorted input $\rightarrow$ n
AVL / red-black	—	$\log n$	$\log n$	in-order = sorted
binary heap	min 1	n	$\log n$	build $O(n)$
trie	—	L	L	L = key length
adj. list / matrix	—	$\deg / 1$	1 / 1	space $V+E / V^2$

Swift mapping

Array	dynamic array, value type + copy-on-write; ContiguousArray skips NSArray bridging
Set / Dictionary	hash tables, open addressing with linear probing, max load 3/4; order unspecified
Deque	swift-collections: ring buffer, $O(1)$ both ends
Heap	swift-collections: min-max heap — min/max $O(1)$, popMin/popMax $O(\log n)$
OrderedSet / OrderedDictionary	array + hash index: $O(1)$ lookup, insertion order kept, remove from middle $O(n)$
TreeSet / TreeDictionary	persistent hash tries (CHAMP) — <i>not</i> sorted trees
sorted tree, list	none built in : sorted array + binary search, or write one

Interview traps

- ! "Linked lists insert faster" — only once you hold the node; the $O(n)$ walk and cache misses usually lose to Array's memmove.
- ! Hash $O(1)$ is *average*, *amortised*: $O(n)$ on collisions/flooding, one insert can rehash $O(n)$, and hashing a long string is $O(\text{length})$.
- ! A heap is not sorted: no binary search, in-order isn't ordered, "contains" is $O(n)$.
- ! Dictionary order and hashValue change per launch (seeded hasher) — never persist them or assert on them in a test.
- ! Swift class list: a 1M-node chain frees *recursively* on deinit $\rightarrow$ stack overflow; unlink iteratively. Make prev weak/unowned or a doubly linked list is a retain cycle.
- ! Amortised $\neq$ worst case: a resize inside a 16 ms frame or audio callback still hitches — pre-size or use a fixed ring buffer.
- ! String is not Int-indexable (grapheme clusters) — Array(s) once if you need random access.

Remember

Layout decides speed, invariant decides big-O. Contiguous by default; pointers only when you must splice.

Likely questions

- Why is append amortised $O(1)$? — doubling: total copies $< 2n$.
- Chaining vs open addressing? — pointers & $\alpha > 1$ vs inline & probing.
- AVL or red-black? — reads vs writes; RB rotates less.
- Heapify cost? — $O(n)$: most nodes are near the leaves.
- Sparse graph storage? — adjacency list, $O(V+E)$.
- Ordered map in Swift? — none built in: sorted keys + binary search, or a tree.
- Why is a DB index a B-tree, not a BST? — fan-out: few page reads per lookup.

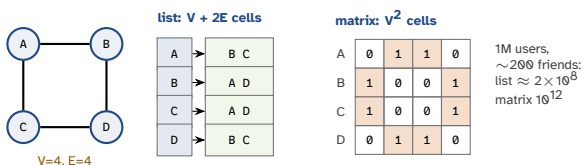
Example — a queue that is really $O(1)$

```

struct Queue<T> {
    // amortised O(1), no Deque needed
    private var items: [T?] = []; private var head = 0
    mutating func enqueue(_ x: T) { items.append(x) }
    mutating func dequeue() -> T? {
        guard head < items.count, let x = items[head] else { return nil }
        items[head] = nil; head += 1 // O(1): no shifting
        if head > 64 && head + 2 > items.count { // compact now and then
            items.removeFirst(head); head = 0 }
        return x }
}

```

Picture — one graph, two layouts



Related: designing-data-structures (LRU, ring buffer) · hashing-deep · algorithm-patterns · strings-and-collections · value-vs-reference (COW)