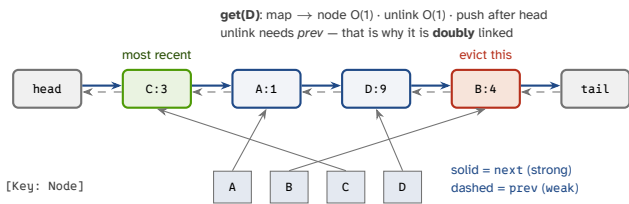


**In one line:** Name the operation that must be O(1), pick the structure whose **invariant** makes it cheap, and when no single structure does all of them, **compose two** that index the same nodes (hash map for *find*, list for *order*; heap for *extreme*, array for *random*). Say what you pay: memory, amortised vs worst case, thread safety.

### LRU = hash map (find) + doubly linked list (order)



LRU cache — [K: Node] + DLL

get/put O(1)

Every access — **read or write** — moves the node to the front; a full put evicts the tail and removes its key from the map (so a node stores its key).

```
final class LRUCache<Key: Hashable, Value> {
    private final class Node { let key: Key; var value: Value
        var next: Node?; weak var prev: Node? // weak: no cycle
        init(_ k: Key, _ v: Value) { key = k; value = v } }
    private var map: [Key: Node] = [:]; private let capacity: Int
    private var head: Node?, tail: Node? // head = newest
    init(capacity: Int) { precondition(capacity > 0); self.capacity = capacity }
    func get(_ k: Key) -> Value? {
        guard let n = map[k] else { return nil }
        unlink(n); pushFront(n); return n.value // a read counts
    }
    func put(_ k: Key, _ v: Value) {
        if let n = map[k] { n.value = v; unlink(n); pushFront(n); return }
        if map.count >= capacity, let old = tail { unlink(old); map[old.key] = nil }
        let n = Node(k, v); map[k] = n; pushFront(n)
    }
    private func unlink(_ n: Node) {
        if let p = n.prev { p.next = n.next } else { head = n.next }
        if let x = n.next { x.prev = n.prev } else { tail = n.prev }
        n.prev = nil; n.next = nil
    }
    private func pushFront(_ n: Node) {
        n.next = head; if let h = head { h.prev = n } else { tail = n }; head = n }
}
```

Thread-safe: guard with OSAllocatedUnfairLock (iOS 16) / Mutex (iOS 18), or make it an actor (then get is async).

LFU cache — key → (value, freq) + freq → DLL + minFreq

O(1)

Access: move the node from list freq to freq+1; if the old list is now empty and was minFreq, minFreq += 1. Evict = tail of lists[minFreq] (ties broken by recency). A **new** key resets minFreq = 1 — the step candidates forget.

Min-stack — a parallel stack of running minimums

all O(1)

push(x): mins.append(min(x, mins.last ?? x)); pop pops both. O(1)-extra-space variant stores 2x - min — overflows, only if asked.

Queue from two stacks — inbox + outbox

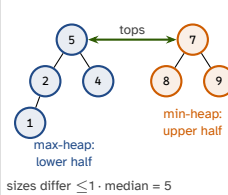
amortised O(1)

Push to inbox. Pop from outbox; if empty, move *all* of inbox over (reverses order). Each element moves once ⇒ O(1) amortised, O(n) worst on the refill pop.

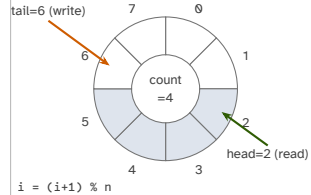
## Which structure makes X cheap?

| must be fast                        | invariant that gives it                  |
|-------------------------------------|------------------------------------------|
| lookup by key, O(1)                 | hash map                                 |
| splice / move a known element, O(1) | doubly linked list (+ map to find it)    |
| both ends, O(1)                     | deque · ring buffer                      |
| current min/max while pushing       | aux stack · monotonic stack/queue        |
| k-th / extreme of a changing set    | heap(s), O(log n) per update             |
| ordered iteration + range queries   | balanced BST / sorted array (read-heavy) |
| prefix queries                      | trie                                     |
| random element, O(1)                | dense array + index map                  |
| "seen?" over huge sets, tiny memory | Bloom filter (false positives only)      |

### Median: two heaps



### Ring buffer



Rate limiter — per key: log or bucket

O(1) / O(limit)

|                 |                                                                            |                                    |
|-----------------|----------------------------------------------------------------------------|------------------------------------|
| sliding log     | Deque of timestamps; drop $\leq$ now - W, allow if count < limit           | exact; O(limit) memory/key         |
| sliding counter | cur + prev · (1 - elapsed/W)                                               | 2 ints; approx.                    |
| token bucket    | tokens = min(b, tokens + $\Delta t \cdot r$ ), lazily on each call; take 1 | 2 numbers; bursts $\leq b$         |
| fixed window    | counter per minute                                                         | $2 \times$ limit across a boundary |

Distributed: the state lives in a shared store (Redis) — atomically.

Autocomplete — trie + top-k cached per node

O(prefix)

Walk the prefix O(p); either DFS the subtree (slow for short prefixes) or store the **top-k completions at every node**, updated on insert — reads O(p), memory  $\times k$ . Node children: [Character: Node] or a 26-array.

Median of a stream — max-heap (low) + min-heap (high) add  $\log n$  · median 1

Push into low if  $\leq$  its max, else high; rebalance so low.count is high.count or +1. Median = Low.max, or the mean of both tops. Swift: two swift-collections Heaps.

Ring buffer — fixed array + head + count

O(1), zero allocs

head == tail means *full or empty*: keep a count, or leave one slot free. Capacity  $2^k \rightarrow i \ \& \ (n-1)$  instead of %. Overwrite-oldest for "last N" logs; reject-when-full for producer-consumer.

Distributed cache — consistent-hash ring + vnodes

lookup  $\log(vN)$

Node positions sorted; key → first node clockwise (binary search). Adding a node moves only  $\sim K/N$  keys (hash % N moves almost all). 100–200 virtual nodes per server even out the arcs. Drawn on *hashing-deep*.

Insert / delete / getRandom — array + [value: index]

all O(1)

Delete = swap with last, pop, fix the moved element's index.

## Interview traps

- ! LRU on OrderedDictionary is **not** O(1): moving a key to the end removes from the middle, O(n). Same for key-array + dict.
- ! Reordering only on put; a singly linked list (unlink is O(n)); strong prev (retain cycle).
- ! NSCache is thread-safe and evicts on memory pressure but is **not** strict LRU — never promise its order.
- ! Array removeFirst() in a queue or sliding log is O(n) per call.
- ! Amortised O(1) is not bounded latency: the two-stack refill or a map resize can land on the audio/render thread — pre-size or use a ring.

## Remember

**One structure finds, the other orders** — and both point at the same node.

## Likely questions

1. Why doubly linked? — unlink a node you found via the map without a walk.
2. Two-stack queue cost? — O(1) amortised; each item moves once.
3. Token bucket vs sliding log? — 2 numbers + bursts vs exact + memory.
4. Streaming p99, not median? — t-digest / histogram buckets (approximate).
5. LRU with TTL? — expiry per node, checked lazily on get; sweep via a min-heap by expiry.