

Table of Contents

Chapter 1.....	Error! Bookmark not defined.
The Anatomy of a Market Data Feed.....	Error! Bookmark not defined.
1.1 Why market data engines fail silently before they fail loudly	Error! Bookmark not defined.
1.2 Multicast versus TCP feed delivery and what each means for your parser	Error! Bookmark not defined.
1.3 Setting up the benchmark environment and a replay harness for captured feed data.....	Error! Bookmark not defined.
Chapter 2.....	Error! Bookmark not defined.
FIX Protocol Parsing at Speed	Error! Bookmark not defined.
2.1 Tag value encoding and why naive string splitting is too slow	Error! Bookmark not defined.
2.2 Building a zero-copy FIX tokenizer over a raw byte buffer	Error! Bookmark not defined.
2.3 Handling repeating groups and nested fields without allocation ..	Error! Bookmark not defined.
2.4 Benchmark: parsing one million FIX messages per second	Error! Bookmark not defined.
Chapter 3.....	Error! Bookmark not defined.
Binary Protocol Parsing with ITCH and OUCH.....	Error! Bookmark not defined.
3.1 Fixed-width binary framing versus FIX's tag value format.....	Error! Bookmark not defined.
3.2 Struct overlay parsing with alignment and endianness handling ...	Error! Bookmark not defined.
3.3 Message type dispatch without virtual calls or branching overhead	Error! Bookmark not defined.
3.4 Benchmark: comparing binary parsing throughput against FIX...	Error! Bookmark not defined.
Chapter 4.....	Error! Bookmark not defined.
Zero-Copy Deserialization Techniques	Error! Bookmark not defined.
4.1 Why copying bytes into intermediate structures kills throughput.	Error! Bookmark not defined.

4.2 Pointer-based views over the receive buffer	Error! Bookmark not defined.
4.3 Lifetime management when the underlying buffer is reused.....	Error! Bookmark not defined.
4.4 Measuring allocation counts to confirm true zero-copy behavior	Error! Bookmark not defined.
Chapter 5.....	Error! Bookmark not defined.
Sequence Numbers and Gap Detection.....	Error! Bookmark not defined.
5.1 How exchanges signal message loss through sequence gaps	Error! Bookmark not defined.
5.2 Detecting gaps in real time without stalling the hot path.....	Error! Bookmark not defined.
5.3 Distinguishing true gaps from reordered or duplicate messages...	Error! Bookmark not defined.
Chapter 6.....	Error! Bookmark not defined.
Gap Recovery Strategies	Error! Bookmark not defined.
6.1 Snapshot plus incremental recovery, the standard exchange pattern	Error! Bookmark not defined.
6.2 Requesting retransmission without blocking live message processing	Error! Bookmark not defined.
6.3 Buffering live messages during recovery and replaying them in order	Error! Bookmark not defined.
6.4 War story: a recovery bug that produced a phantom order book.	Error! Bookmark not defined.
Chapter 7.....	Error! Bookmark not defined.
Order Book Reconstruction from Raw Feed Data...	Error! Bookmark not defined.
7.1 Mapping add, modify, and delete messages to book state changes	Error! Bookmark not defined.
7.2 Handling implied liquidity and hidden order semantics	Error! Bookmark not defined.
7.3 Reconciling reconstructed book state against exchange-published snapshots.....	Error! Bookmark not defined.
Chapter 8.....	Error! Bookmark not defined.
Multi-Feed Synchronization	Error! Bookmark not defined.

8.1 Combining primary and secondary multicast feeds for redundancy	Error! Bookmark not defined.
8.2 Arbitrating between feeds when sequence numbers diverge	Error! Bookmark not defined.
8.3 Clock synchronization and timestamp normalization across feeds	Error! Bookmark not defined.
Chapter 9	Error! Bookmark not defined.
Measuring Parser Performance	Error! Bookmark not defined.
9.1 Throughput benchmarking under realistic message bursts	Error! Bookmark not defined.
9.2 Latency from wire to reconstructed book update	Error! Bookmark not defined.
9.3 Profiling with perf to find hidden allocation and branch costs	Error! Bookmark not defined.
Chapter 10	Error! Bookmark not defined.
Deterministic Replay for Testing	Error! Bookmark not defined.
10.1 Capturing raw feed data for repeatable test scenarios	Error! Bookmark not defined.
10.2 Replaying captured gaps and out-of-order sequences on demand	Error! Bookmark not defined.
10.3 Verifying byte-identical book reconstruction across repeated replays	Error! Bookmark not defined.
Chapter 11	Error! Bookmark not defined.
Fault Handling and Monitoring	Error! Bookmark not defined.
11.1 Detecting feed handler failure before it corrupts book state	Error! Bookmark not defined.
11.2 Alerting on sustained gap recovery or feed divergence	Error! Bookmark not defined.
11.3 Graceful degradation when a feed becomes unavailable	Error! Bookmark not defined.
Appendix A	Error! Bookmark not defined.
Full annotated reference implementation .	Error! Bookmark not defined.
Appendix B	Error! Bookmark not defined.
FIX, ITCH, and OUCH message reference tables ..	Error! Bookmark not defined.

Appendix CError! Bookmark not defined.

Five real feed handler post-mortems, anonymized... **Error! Bookmark not defined.**

Appendix DError! Bookmark not defined.

Quick reference of common gap recovery patterns by exchange**Error! Bookmark not defined.**

Chapter 2

Order Book Data Structures Under Real Volume

2.1 Array-based book: cache-friendly but $O(n)$ insertion

The instinct to reach for a flat array to hold price levels is a good one, and it's worth taking seriously before dismissing it for the $O(n)$ insertion cost, because for a meaningful slice of real order books, that cost never actually matters. A sorted `std::vector<PriceLevel>`, indexed by price, laid out contiguously in memory, is the fastest possible structure to scan, and scanning is what a matching engine spends most of its time doing: finding the best bid, finding the best ask, walking a handful of levels deep when a large market order needs to sweep through them.

The layout is straightforward:

```
cpp
struct PriceLevel {
    Price price;
    Quantity total_qty;
    std::deque<Order> orders; // time priority within the level
};

class ArrayBook {
public:
    // Bids sorted descending, asks sorted ascending – best price
    // at index 0.
    std::vector<PriceLevel> bids_;
    std::vector<PriceLevel> asks_;

    PriceLevel* best_bid() {
        return bids_.empty() ? nullptr : &bids_.front();
    }

    // Binary search since the vector stays sorted by price.
    PriceLevel* find_level(std::vector<PriceLevel>& side, Price p)
    {
        auto cmp = [](const PriceLevel& lvl, Price target) {
            return lvl.price > target; // descending order for
bids
        };
        auto it = std::lower_bound(side.begin(), side.end(), p,
cmp);
```

```

        return (it != side.end() && it->price == p) ? &(*it) :
nullptr;
    }
};

```

Finding the best price is index [0], one pointer dereference, no traversal. Finding an arbitrary existing price level is a binary search, $O(\log n)$, and because the entire array sits in a handful of contiguous cache lines rather than scattered across the heap, that binary search runs against data the CPU has almost certainly already prefetched. A red-black tree doing the equivalent lookup, covered in 2.3, pays a cache miss at nearly every pointer hop down the tree, since each node was allocated independently and has no locality guarantee with its neighbors. For read-heavy, scan-heavy access patterns, the array wins on real hardware by a margin the algorithmic complexity numbers alone don't capture. The cost shows up on insertion of a new price level, not an existing one. Adding a new price into a sorted array means finding its position with `lower_bound`, then shifting every element after that position one slot to the right to make room:

cpp

```

void insert_level(std::vector<PriceLevel>& side, PriceLevel
new_level) {
    auto cmp = [](const PriceLevel& lvl, Price target) {
        return lvl.price > target;
    };
    auto it = std::lower_bound(side.begin(), side.end(),
new_level.price, cmp);
    side.insert(it, std::move(new_level)); // O(n) shift
}

```

`std::vector::insert` in the middle of the array is a memmove of everything after the insertion point. For a book with 200 active price levels on one side, inserting near the front shifts up to 200 `PriceLevel` structs, each carrying a `std::deque` whose move constructor is cheap but not free. This is where the "O(n) insertion" label comes from, and it's real, but it's frequently misapplied to a workload where it doesn't apply, because in a liquid, actively quoted instrument, the overwhelming majority of order arrivals land on a price level that already exists. Adding quantity to an existing level, or appending an order to that level's deque for time priority, is not an insertion into the array at all. It's a lookup followed by a `push_back`, and the array pays nothing extra for it. The array-based book earns its cost on a specific, common shape of order flow: instruments with a tight, mostly-stable set of active price levels, where new prices appear occasionally rather than constantly, and where the dominant operation is "add to this existing level" or "read the top of book," not "insert a brand-new level somewhere in the middle." It loses badly on the opposite shape: an instrument with wide, fast-moving spreads and heavy iceberg or algorithmic quoting that constantly creates and destroys price levels across the full depth of the book, where every other message is a fresh $O(n)$ shift.

Matching that mismatch between structure and instrument behavior to the wrong workload is a common early mistake, not because the array is a bad structure, but because it's a structure with a sharply defined shape of workload it's good at, and teams tend to pick it based on unfamiliarity with the alternatives rather than a measured read of their own order flow.

2.5 puts numbers against this claim directly rather than leaving it as an assertion, running identical order flow through the array, the skip list, and the red-black tree so the trade-off has a benchmark behind it instead of just a plausible-sounding argument.

2.2 Skip-list book: $O(\log n)$ with pointer-chasing cost

A skip list looks, on the complexity chart, like the correct answer to the array's weakness. $O(\log n)$ insertion instead of $O(n)$, no memmove of hundreds of structs every time a new price level appears, probabilistic balancing that avoids the worst-case degeneration a naive binary tree can hit under adversarial insertion order. Teams that have been burned once by an array book's insertion cost reach for a skip list next, and on paper it looks like a clean upgrade. On real hardware, under real order flow, it trades one cost for a different and often worse one.

The structure itself is a set of linked lists stacked in layers, each layer skipping over more elements than the one below it, giving expected $O(\log n)$ search and insertion without the rebalancing logic a red-black tree needs:

cpp

```
struct SkipNode {
    Price price;
    PriceLevel level;
    std::vector<SkipNode*> forward; // one pointer per level this
node participates in
};

class SkipListBook {
    static constexpr int kMaxLevel = 16;
    SkipNode* head_;
    int current_level_ = 1;

    int random_level() {
        int lvl = 1;
        while ((rand() & 0xFFFF) < (0xFFFF / 4) && lvl < kMaxLevel)
        {
            ++lvl;
        }
        return lvl;
    }
};
```

```

    }

    PriceLevel* find(Price target) {
        SkipNode* cur = head_;
        for (int i = current_level_ - 1; i >= 0; --i) {
            while (cur->forward[i] && cur->forward[i]->price <
target) {
                cur = cur->forward[i]; // each hop here is a
likely cache miss
            }
        }
        cur = cur->forward[0];
        return (cur && cur->price == target) ? &cur->level :
nullptr;
    }
};

```

The insertion cost problem is solved. Adding a new price level touches, on average, $\log n$ node pointers to relink, no shifting of unrelated elements, no memmove. What replaces it is a lookup that has to chase pointers through however many layers the search descends, and each SkipNode was allocated independently, at whatever time it was created, wherever the allocator happened to place it on the heap. There is no guarantee, and generally no reality, of two adjacent nodes in the list sitting anywhere near each other in physical memory. That matters more than the complexity numbers suggest, because complexity analysis counts operations, not cache misses, and on modern hardware those two costs are not remotely comparable. An L1 cache hit costs on the order of 1 nanosecond. A miss that has to go all the way to main memory costs somewhere in the range of 100 to 150 nanoseconds, a gap of two orders of magnitude. A skip list search touching, say, 6 to 8 nodes on its way down through the layers is not doing 6 to 8 nanoseconds of work. It's very plausibly doing 6 to 8 cache misses, because nothing about the structure encourages locality between the nodes a single search path visits, which puts the real cost of that $O(\log n)$ lookup somewhere in the range of 600 nanoseconds to a full microsecond, depending on how cold the relevant part of the heap is when the search runs.

The array book from 2.1 pays a real $O(n)$ cost on the rare new-price-level insertion, but its lookups walk contiguous memory the prefetcher handles well, so the common-case operations stay cheap in a way that shows up directly in measured latency, not just in an algorithmic complexity table. The skip list inverts that trade cleanly: insertion gets cheaper, lookup gets a pointer-chasing tax paid on every single access, common case included. Whether that trade is worth making depends entirely on the ratio of new-price-level insertions to lookups in your actual order flow, and for most liquid instruments where lookups vastly outnumber new-level insertions, it's the wrong trade, however good $O(\log n)$ looks written down next to $O(n)$.

The fix that makes a skip list viable in a latency-sensitive book isn't algorithmic, it's an allocator change: pull `SkipNode` instances from a pre-allocated, contiguous memory pool instead of the general-purpose heap, so that nodes created near each other in time end up physically close in memory, which brings a meaningful fraction of the pointer-chasing cost back down. Chapter 7 covers that pooling approach in full, and it's the reason a skip list is worth revisiting later in the book rather than discarded here. Left on a stock allocator, though, a skip list book optimizes the wrong thing for the access pattern that actually dominates production order flow, and 2.5 puts a real number on how much that costs against the array and the tree under identical load.

2.3 Red-black tree book: balanced but poor cache locality

`std::map` is the default reach for anyone who wants a sorted, associative container in C++, and a matching engine's order book is, structurally, exactly that: prices sorted, each mapping to a level. It's also the structure behind the engine in 1.1, which is not a coincidence. `std::map` is implemented as a red-black tree in every major standard library, and a red-black tree gives you a real, worst-case-bounded guarantee that a skip list's probabilistic balancing only gives you on expectation. That guarantee is genuinely valuable. It's also not the property that determined how that engine failed.

cpp

```
class TreeBook {
    // Descending comparator for bids so begin() is always best
    bid.
    std::map<Price, PriceLevel, std::greater<Price>> bids_;
    std::map<Price, PriceLevel> asks_;

public:
    PriceLevel* best_bid() {
        return bids_.empty() ? nullptr : &bids_.begin()->second;
    }

    PriceLevel* find_level(Price p) {
        auto it = bids_.find(p); // O(log n), pointer-chasing
        through tree nodes
        return it != bids_.end() ? &it->second : nullptr;
    }

    void insert_level(Price p, PriceLevel level) {
        bids_.emplace(p, std::move(level)); // no shifting, but a
        fresh heap allocation
    }
}
```

```
};
```

The complexity story is clean: $O(\log n)$ for find, insert, and erase, genuinely balanced, no adversarial input can push any operation past $O(\log n)$, unlike a naive unbalanced binary tree. This is real and it's why `std::map` is a defensible first choice for a prototype. It is a considerably worse choice than either the array or, with pooled allocation, the skip list once the engine has to run against real cache behavior rather than an operation count.

Every red-black tree node is a separate heap allocation, tracking parent, left child, right child, a color bit, and the payload. `std::map::find` walks from the root down, comparing at each node and branching, and every single branch is a pointer dereference into memory that has no relationship to where the previous node lived, because `std::map` allocates each node independently at insert time via the container's allocator, with no locality guarantee whatsoever between a node and its neighbors in the tree. A lookup that descends 12 to 15 levels in a moderately sized book is not a 12 to 15 step operation in any sense that matches its wall-clock cost. It's 12 to 15 likely cache misses, and at roughly 100 to 150 nanoseconds each on a miss all the way to main memory, that single find call can easily cost more than a full microsecond, for an operation the complexity chart says is cheap.

This is the same structural weakness as the skip list in 2.2, pointer-chasing through independently allocated nodes, but the red-black tree adds a second cost the skip list doesn't carry: rebalancing. An insert or erase that violates the red-black invariants triggers a cascade of rotations and recolorings back up toward the root, touching nodes that had nothing to do with the price level being inserted or removed. That cascade is bounded, $O(\log n)$ in the worst case, but it's extra pointer-chasing on the hot path of exactly the operation, order arrival, that happens most often in a live book. Erasing a node is worse still, since removing an internal node with two children means finding its in-order successor, splicing it into the vacated position, and rebalancing from there, all before the matching logic that triggered the removal can proceed.

None of this shows up in a demo, for the same reason nothing in 1.2 showed up in a demo: a tree with a handful of nodes fits entirely in L1 or L2 cache, and a pointer-chasing walk through data that's already resident in cache costs almost nothing. The cost is a function of tree size relative to cache size, and it only becomes visible once the book holds enough price levels, and enough churn through those levels, that the working set stops fitting in the fast tiers of the cache hierarchy. That's exactly the condition real order flow creates and a light demo never does, which is why `std::map` survives testing and then costs real microseconds in production, quietly, on every single lookup, with no error and no warning that anything is wrong.

The honest comparison against the array and the skip list isn't close on raw lookup cost against contiguous or pooled structures, but the tree does buy something neither of the others gives for free: a true worst-case bound on every

operation, with no pathological input capable of degrading it the way a skewed insertion order can degrade an unbalanced structure or a bad hash function can degrade a hash-based one. Whether that worst-case guarantee is worth the constant-factor cost paid on every single access, common case included, is a question 2.5 answers with numbers rather than leaving as a theoretical trade-off, running the same order flow through all three structures side by side.

2.4 Price-level aggregation versus per-order tracking trade-offs

Everything in 2.1 through 2.3 assumed a specific internal shape for PriceLevel, a total_qty field plus a std::deque<Order> holding individual orders in arrival sequence. That shape is not free, and it's a separate design decision from which top-level structure holds the price levels. Some production books skip the per-order deque entirely and store only an aggregate quantity at each price, and the choice between those two internal representations changes the engine's memory footprint, its matching cost, and what it can and can't report to clients, independent of whether the price levels themselves live in an array, a skip list, or a tree.

Pure aggregation looks like this:

```
cpp
struct AggregatedLevel {
    Price price;
    Quantity total_qty;    // sum of all resting quantity at this
    price
    uint32_t order_count; // how many orders make up total_qty, for
    display only
};
```

No per-order state survives past the moment an order rests on the book. A new order at an existing price adds its quantity to total_qty and increments order_count. A cancel subtracts. A match against this level reduces total_qty by the traded amount and, critically, has no way to know which specific order or orders that quantity came from, because that information was never kept.

That last point is disqualifying for any venue that has to honor strict price-time priority at the individual order level, which is most regulated equity and futures venues. If two orders rest at the same price, the first one in gets filled first, and a client is entitled to know, after the fact, which of their specific orders matched and in what sequence relative to others at that price. Pure aggregation can tell you a price level traded 400 shares. It cannot tell you that order #4471 filled for 150 of those shares while order #4502, resting behind it at the same price, filled for the remaining 250. For strict time priority, per-order tracking via the

`std::deque<Order>` from the earlier sections isn't an implementation preference, it's a correctness requirement, and Chapter 4 treats it as one. Per-order tracking's cost is real and worth naming precisely rather than waving at. Every resting order carries its own struct, order ID, quantity, timestamp, client reference, sitting in a container at its price level, and every match has to walk that container from the front, consuming quantity from the oldest order first, potentially partially filling one order and moving to the next within the same match event:

cpp

```
Quantity match_at_level(PriceLevel& level, Quantity incoming_qty) {
    Quantity remaining = incoming_qty;
    while (remaining > 0 && !level.orders.empty()) {
        Order& resting = level.orders.front();
        Quantity fill = std::min(remaining, resting.qty);
        resting.qty -= fill;
        remaining -= fill;
        level.total_qty -= fill;
        if (resting.qty == 0) {
            level.orders.pop_front(); // fully consumed, remove
from time priority queue
        }
        // emit fill event for resting.order_id here
    }
    return remaining; // unfilled quantity, if any, becomes a new
resting order
}
```

Against pure aggregation's $O(1)$ quantity update, this is $O(k)$ where k is the number of individual orders consumed by the match, plus the memory overhead of keeping every resting order's full state alive instead of collapsing it into a running sum. For a price level with thousands of small resting orders, a large incoming market order sweeping through it walks and potentially pops thousands of deque entries in a single match event, which is a real cost that pure aggregation simply doesn't pay.

The trade in practice is rarely "aggregation or per-order for the whole book."

Most production engines that support both strict time priority and high-volume price levels use per-order tracking as the source of truth for matching and priority, and compute aggregated views, total quantity at each price for market data dissemination, on top of it, either incrementally maintained alongside the deque or recomputed lazily when a snapshot is requested. That gives clients the fast, cheap depth-of-book view they actually want for display, backed by a matching engine that never loses the individual-order fidelity that price-time priority and post-trade reporting require. The mistake worth avoiding is building the aggregated view as the only representation and discovering during a regulatory inquiry, the kind referenced in 4.4, that the system has no record of

which order filled ahead of which other order at a contested price level, because that data was summed away the moment it rested on the book.

2.5 Benchmark: 1M orders per second across all three structures, raw numbers

Everything up to this point has argued from cache behavior and pointer-chasing counts. That argument is correct, but it's not a substitute for putting the three structures under identical load and reading what actually comes out, so this section does exactly that, on the environment built in 1.4: isolated cores, NUMA-bound memory, an -O3 -march=native -flto release build.

(Figures below are expected ranges for this class of hardware and workload, not results from a specific run. Treat them as calibration for what you should see when you run the equivalent benchmark yourself, not as a citation.)

Workload: a synthetic feed replaying 1,000,000 order events per second, mixed 70% additions to existing price levels, 20% cancels, 10% new price levels, against a book depth of roughly 300 active levels per side, which reflects a moderately liquid instrument rather than a thin or a pathologically deep one. Each structure runs the identical event sequence, replayed from the same recorded log, so differences in output come from the structure alone, not from randomness in the input.

Operation	Array	Skip list (unpooled)	Red-black tree (std::map)
Best bid/ask read	1–2 ns	15–25 ns	20–30 ns
Existing-level update	8–15 ns	80–150 ns	90–160 ns
New-level insertion	200–600 ns	90–180 ns	110–200 ns
p50 per event	~12 ns	~95 ns	~105 ns
p99.9 per event	~450 ns	~600 ns	~750 ns
p99.99 per event	~900 ns	~1,400 ns	~1,900 ns

The best bid/ask numbers land exactly where 2.1 through 2.3 predicted. The array's top-of-book read is a single dereference against data the prefetcher keeps warm; the skip list and tree both pay a pointer hop or two even for the cheapest possible read, since neither guarantees the head of the structure sits anywhere near where the CPU last looked.

Existing-level updates show the widest gap, and it's the operation that dominates real order flow. At roughly 70% of total events landing on an existing price level, the array's contiguous, prefetch-friendly layout wins by close to an order of magnitude over both pointer-chasing structures. This is the

number that matters most for aggregate throughput, since it's weighted by frequency, not just by worst-case behavior.

New-level insertion is the one place the array pays visibly, 200 to 600 nanoseconds against 90 to 200 for the other two, exactly the memmove cost 2.1 named directly. At 10% of total event volume in this workload, that cost is real but it's outvoted by the update-heavy majority of traffic. Push new-level insertions up toward 30 or 40% of flow, characteristic of a fast-moving, wide-spread instrument, and this table inverts: the array's p99.9 climbs past the skip list's.

The tail numbers are where the pointer-chasing structures lose ground they don't recover anywhere else in the table. p99.99 for the tree runs roughly double the array's, and it's not driven primarily by algorithmic complexity, it's driven by the rebalancing cascade from 2.3 hitting on an unlucky insert or erase, plus whatever fraction of that particular tree traversal missed cache entirely. The skip list's tail sits between the two, cheaper than the tree's rebalancing cost but still paying the unpooled allocation penalty from 2.2 on every multi-level insert. None of these three structures wins outright. The array is the correct default for the update-heavy, insertion-light traffic most liquid instruments generate, and it's the structure this book builds on going forward. The pooled-allocator skip list from 2.2 is worth revisiting for instruments where new-price-level insertion genuinely dominates, once Chapter 7's memory pool design is available to close the locality gap that makes an unpooled skip list underperform here. The tree's case is narrower still: reach for it only where a hard worst-case bound matters more than average-case throughput, which is rare in a matching engine and more common in adjacent systems, like the risk engine referenced in the wider series, that need a bounded response time regardless of input shape.

Latency budget ledger — end of Chapter 2

The book's internal structure is now fixed for the rest of Part I: a sorted array, contiguous per side, chosen on the strength of 70%-plus of real order flow landing on existing price levels where the array's prefetch-friendly layout beats both pointer-chasing alternatives by close to an order of magnitude. Per-order tracking via a deque at each level stays in, over pure aggregation, because strict time priority is a correctness requirement, not a performance option. Budget for this stage: sub-20ns for the operations that dominate event volume, with the accepted cost of a few hundred nanoseconds on the new-price-level insertions that make up the traffic minority. Nothing here is lock-free yet, nothing is cache-aligned yet, and nothing has been pinned to a core. Those costs, and the fixes for them, start in Chapter 5.