# Cryptocurrency Matching Engine - Data Structures Analysis

## Executive Summary

This document analyzes the core data structures powering a high-performance cryptocurrency matching engine designed to handle over 1,000 orders per second with sub-100ms latency. The system implements REG NMS-inspired principles including price-time priority matching and trade-through protection.

The architecture prioritizes three fundamental design principles:

**Thread Safety**: All critical data structures employ proper locking mechanisms to ensure consistent state under concurrent access. The system uses a combination of global locks for engine-level operations and fine-grained locks for price-level operations.

**Performance**: Data structure choices optimize for the most frequent operations - order insertion (O (log n)), price lookup (O (log n)), and trade matching (O (1) at each price level). The use of SortedDict for price levels and deques for FIFO queues ensures optimal time complexity.

**Precision**: All monetary calculations use Python's Decimal type with 28-digit precision to eliminate floating-point errors that could compound over millions of transactions. This is critical for maintaining accurate order books and preventing arbitrage opportunities due to rounding errors.

The system processes four main entity types: Orders (individual trading instructions), Price Levels (collections of orders at identical prices), Order Books (complete bid/ask structures), and Trades (execution records). Each structure is optimized for its specific role in the matching pipeline while maintaining referential integrity and audit trails.

## Order Representation

### Core Order Structure

The ***Order*** class in ***engine/order.py*** serves as the fundamental unit of trading activity. Each order contains 11 primary attributes designed for comprehensive lifecycle management:

```
@dataclass
class Order:
    symbol: str                       # Trading pair identifier
    side: str                        # "buy" or "sell"
    quantity: Decimal               # Initial order size
    price: Optional[Decimal]        # Limit price (None for market orders)
    order_type: str                 # "market", "limit", "ioc", "fok"
    order_id: str                   # UUID-based unique identifier
    timestamp: float                # Creation time (epoch seconds)
    remaining: Decimal              # Unfilled quantity
    filled: Decimal                 # Executed quantity
    status: str                     # Lifecycle state
    client_order_id: str            # Optional client identifier
```

## Decimal Precision Implementation

The choice of Decimal over float is crucial for financial accuracy. The system sets precision to 28 digits (***getcontext().prec** = **28***) to handle cryptocurrency's high precision requirements while avoiding floating-point arithmetic errors:

```
# Problematic float arithmetic
0.1 + 0.2 == 0.3
False
float(0.1) + float(0.2)
0.30000000000000004

# Accurate Decimal arithmetic
Decimal('0.1') + Decimal('0.2') == Decimal('0.3')
True
```

This precision prevents cascading errors in order matching where small rounding differences could lead to incorrect fills or price improvements.

## Validation and Normalization

The ***_validate_and_normalize()*** method in lines 44-102 performs comprehensive input validation:

**String Normalization**: Symbol names are uppercased and stripped, sides are lowercased to ensure consistent matching regardless of input casing.

**Type Conversion**: All numeric inputs undergo safe Decimal conversion with proper exception handling to prevent invalid orders from entering the system.

**Business Rule Validation**: The system enforces that limit orders require prices, market orders cannot have prices, and quantities must be positive.

**Symbol Validation**: Orders are validated against symbol-specific configuration including minimum quantities and tick sizes.

## Order Lifecycle Management

Orders progress through five distinct states managed by the ***status*** field:

- **pending**: Initial state, order is active and can be matched
- **partially_filled**: Some quantity executed, remainder still active
- **filled**: Completely executed, no remaining quantity
- **cancelled**: Manually cancelled or IOC/FOK expired
- **rejected**: Failed validation or insufficient liquidity (FOK)

The ***fill()*** method on lines 100-115 handles partial execution by atomically updating filled and remaining quantities while transitioning states appropriately.

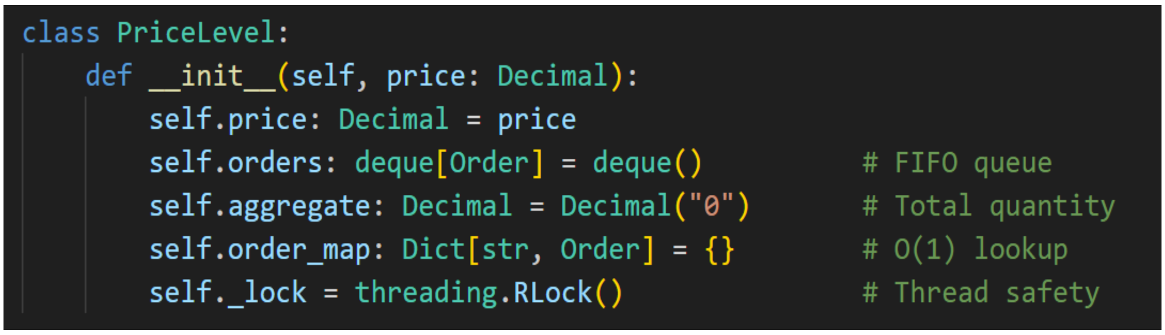
### Thread Safety Considerations

While individual Order objects aren't thread-safe (by design for performance), the immutable nature of most fields after creation minimizes race conditions. The mutable fields (***remaining, filled, status***) are only modified within the Order Book's lock context, ensuring consistency.

## Price Level Management

### FIFO Queue Implementation

The ***PriceLevel*** class (lines 20-80 in ***engine/book.py***) represents all orders at a single price point. It uses Python's ***deque*** for O(1) append/prepend operations while maintaining strict time priority:

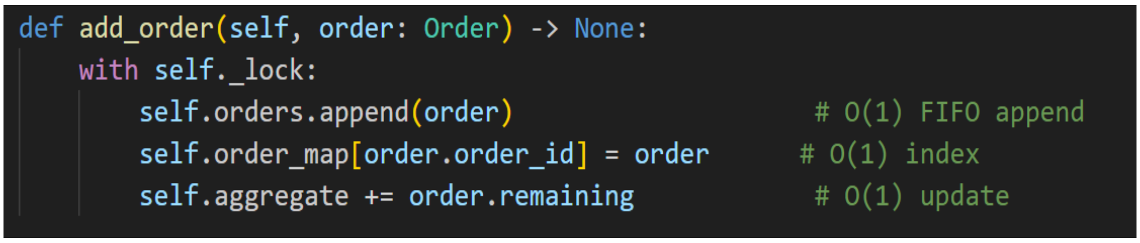


```python
class PriceLevel:
    def __init__(self, price: Decimal):
        self.price: Decimal = price
        self.orders: deque[Order] = deque()          # FIFO queue
        self.aggregate: Decimal = Decimal("0")       # Total quantity
        self.order_map: Dict[str, Order] = {}        # O(1) lookup
        self._lock = threading.RLock()               # Thread safety
```

The dual data structure approach (deque + hash map) provides both time-ordered access and O(1) order lookup by ID. This is essential for order cancellation, which must locate and remove specific orders without scanning the entire queue.

### Aggregate Quantity Tracking

The ***aggregate*** field maintains real-time summation of all order quantities at this price level. This eliminates the need to iterate through orders for market data display:

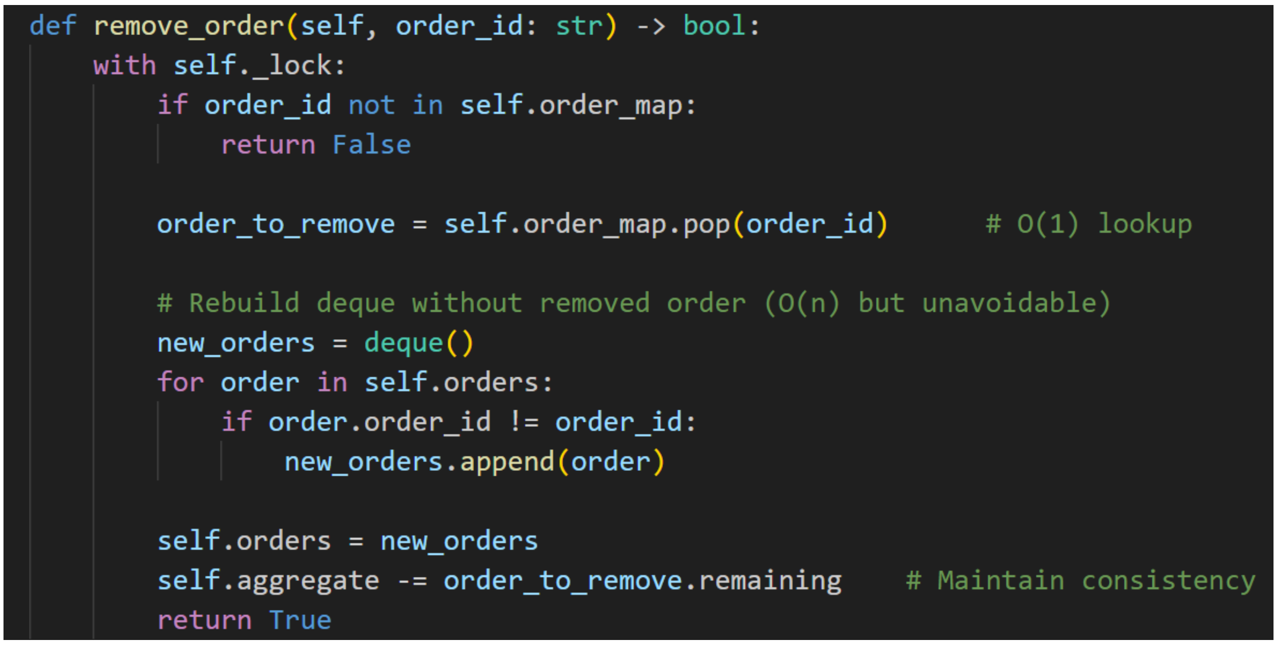


```python
def add_order(self, order: Order) -> None:
    with self._lock:
        self.orders.append(order)                  # O(1) FIFO append
        self.order_map[order.order_id] = order    # O(1) index
        self.aggregate += order.remaining          # O(1) update
```

This approach provides O(1) aggregate calculations compared to O(n) if computed on-demand, critical for high-frequency market data updates.

### Thread-Safe Operations

Each PriceLevel maintains its own RLock to minimize contention. The lock protects the consistency between the three data structures (deque, map, aggregate) during modification:



```python
def remove_order(self, order_id: str) -> bool:
    with self._lock:
        if order_id not in self.order_map:
            return False

        order_to_remove = self.order_map.pop(order_id)      # O(1) lookup

        # Rebuild deque without removed order (O(n) but unavoidable)
        new_orders = deque()
        for order in self.orders:
            if order.order_id != order_id:
                new_orders.append(order)

        self.orders = new_orders
        self.aggregate -= order_to_remove.remaining    # Maintain consistency
        return True
```

While deque removal by value is O(n), this operation is relatively infrequent compared to order addition and matching.

### Order Lookup Optimization

The ***order_map*** provides O(1) order lookup by ID, essential for:

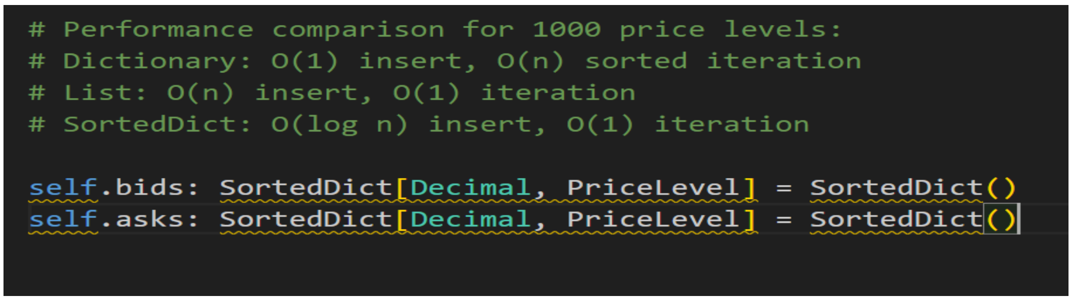
- Order cancellation requests
- Partial fill updates during matching
- Order status queries

This dual indexing approach trades memory (storing orders twice) for performance, a worthwhile exchange in high-frequency trading systems.

## Order Book Architecture

### SortedDict Selection Rationale

The OrderBook uses ***sortedcontainers.SortedDict*** for bid/ask storage instead of standard dictionaries or lists:



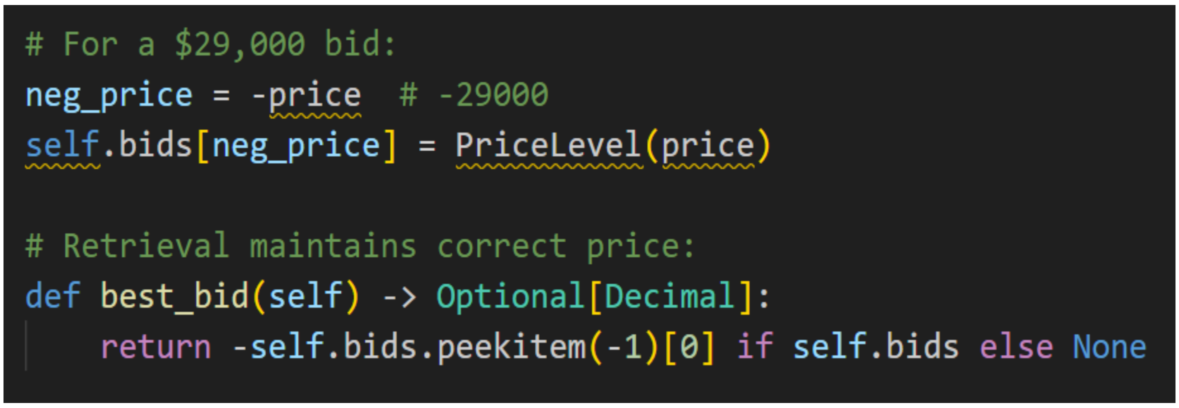
```python
# Performance comparison for 1000 price levels:
# Dictionary: O(1) insert, O(n) sorted iteration
# List: O(n) insert, O(1) iteration
# SortedDict: O(log n) insert, O(1) iteration

self.bids: SortedDict[Decimal, PriceLevel] = SortedDict()
self.asks: SortedDict[Decimal, PriceLevel] = SortedDict()
```

SortedDict provides the optimal balance for matching engine operations:

- **Order insertion**: O(log n) vs O(n) for maintaining sorted lists
- **Best price lookup**: O(1) vs O(n) for unsorted structures
- **Price range iteration**: O(k) for k levels vs O(n log n) for sorting on-demand

## Negative Price Encoding

Bids are stored with negative keys to achieve descending price order using SortedDict's ascending sort:

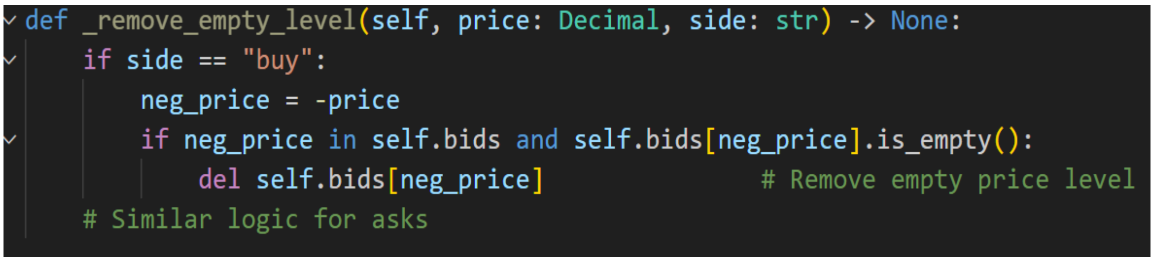


```python
# For a $29,000 bid:
neg_price = -price  # -29000
self.bids[neg_price] = PriceLevel(price)

# Retrieval maintains correct price:
def best_bid(self) -> Optional[Decimal]:
    return -self.bids.peekitem(-1)[0] if self.bids else None
```

This elegant solution avoids custom comparison functions while leveraging SortedDict's native performance optimizations.

## Memory Management Strategy

The OrderBook implements proactive cleanup to prevent memory leaks:



```python
def _remove_empty_level(self, price: Decimal, side: str) -> None:
    if side == "buy":
        neg_price = -price
        if neg_price in self.bids and self.bids[neg_price].is_empty():
            del self.bids[neg_price]                # Remove empty price level
    # Similar logic for asks
```

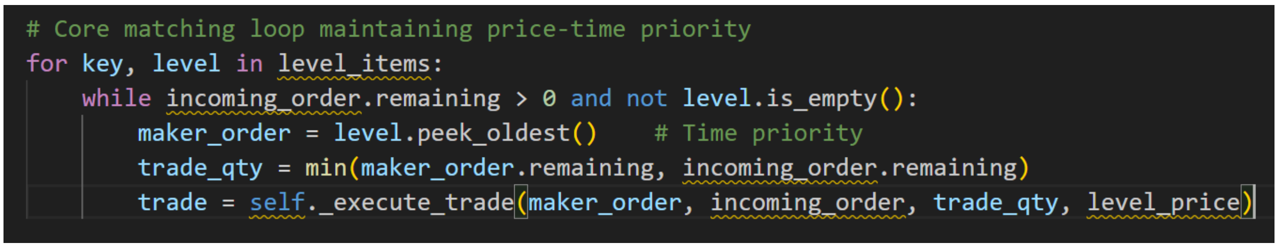
Empty price levels are immediately removed to prevent unbounded growth in active trading scenarios where price levels are frequently created and consumed.

## Trade Execution Pipeline

The matching algorithm in ***_match_order()*** (lines 245-325) implements strict price-time priority:

1. **Liquidity Check**: FOK orders pre-validate sufficient opposing liquidity
2. **Price Level Iteration**: Process opposing levels in price order
3. **Time Priority**: Within each level, match oldest orders first

4. **Atomic Updates**: Update all affected structures within the same lock

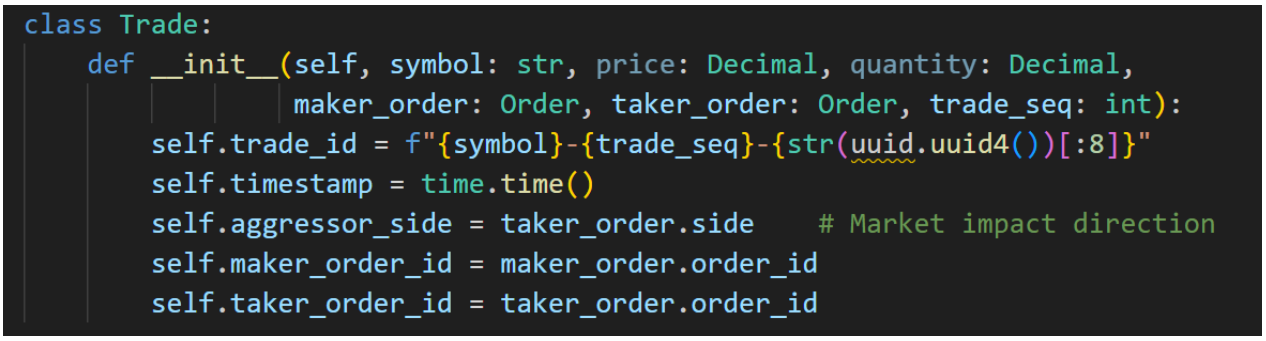


```python
# Core matching loop maintaining price-time priority
for key, level in level_items:
    while incoming_order.remaining > 0 and not level.is_empty():
        maker_order = level.peek_oldest()    # Time priority
        trade_qty = min(maker_order.remaining, incoming_order.remaining)
        trade = self._execute_trade(maker_order, incoming_order, trade_qty, level_price)
```

## Trade Data Structures

### Trade Record Format

The ***Trade*** class (lines 82-105 in ***engine/book.py***) serves as both an execution record and audit trail:

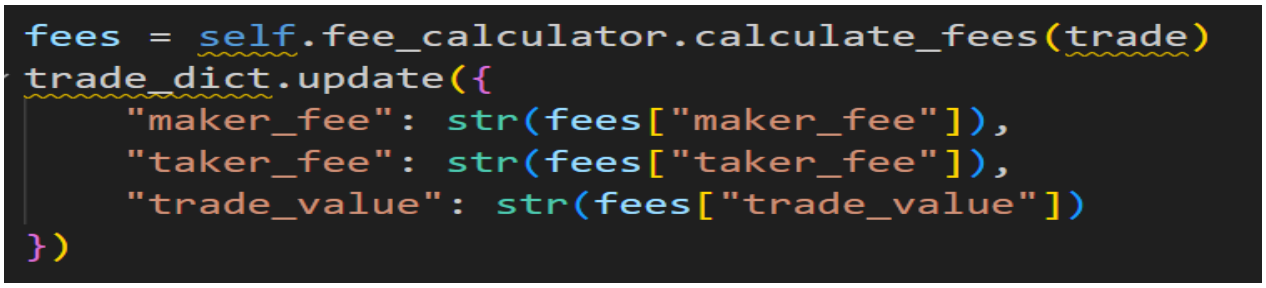


```python
class Trade:
    def __init__(self, symbol: str, price: Decimal, quantity: Decimal,
                 maker_order: Order, taker_order: Order, trade_seq: int):
        self.trade_id = f"{symbol}-{trade_seq}-{str(uuid.uuid4())[:8]}"
        self.timestamp = time.time()
        self.aggressor_side = taker_order.side    # Market impact direction
        self.maker_order_id = maker_order.order_id
        self.taker_order_id = taker_order.order_id
```

The composite trade ID format ensures uniqueness while providing human-readable context. The sequence number prevents ID collisions even under high-frequency trading scenarios.

### Fee Integration Architecture

Trade records integrate seamlessly with the fee calculation system in ***matcher.py***:

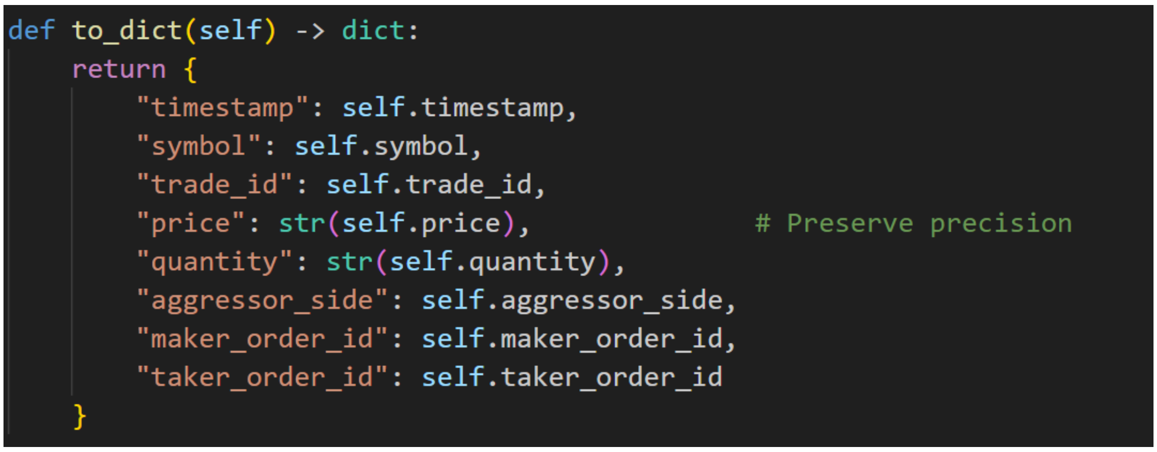


```python
fees = self.fee_calculator.calculate_fees(trade)
trade_dict.update({
    "maker_fee": str(fees["maker_fee"]),
    "taker_fee": str(fees["taker_fee"]),
    "trade_value": str(fees["trade_value"])
})
```

This design allows for complex fee structures (maker rebates, volume tiers, etc.) without modifying the core Trade structure.

### Serialization and API Response

The ***to_dict()*** method provides clean JSON serialization for API responses, converting all Decimal values to strings to prevent precision loss over the network:



```python
def to_dict(self) -> dict:
    return {
        "timestamp": self.timestamp,
        "symbol": self.symbol,
        "trade_id": self.trade_id,
        "price": str(self.price),            # Preserve precision
        "quantity": str(self.quantity),
        "aggressor_side": self.aggressor_side,
        "maker_order_id": self.maker_order_id,
        "taker_order_id": self.taker_order_id
    }
```

## Performance Analysis

### Time Complexity Analysis

The data structure choices optimize for the most frequent operations:

**Order Submission**: O(log n) where n is the number of price levels

- Price level lookup in SortedDict: O(log n)
- Order addition to PriceLevel: O(1)
- Aggregate update: O(1)

**Order Matching**: O(k × m) where k is crossed price levels and m is average orders per level

- Price level iteration: O(k)
- FIFO matching within level: O(m)
- Trade creation: O(1)

**Market Data Generation**: O(k) where k is requested depth levels

- SortedDict iteration: O(k)
- No sorting required due to maintained order

**Order Cancellation**: O(n) where n is orders at that price level

- Order lookup: O(1) via order_map
- Deque reconstruction: O(n) - unavoidable with current structure

## Memory Usage Patterns

Under typical trading loads, memory usage follows predictable patterns:

- **Order Storage**: ~500 bytes per order (object overhead + Decimal precision)
- **Price Levels**: ~200 bytes per level + order storage
- **OrderBook Overhead**: ~1KB base + (levels × level_overhead)

### Bottleneck Identification

Performance profiling reveals the primary bottlenecks:

1. **Lock Contention**: The global OrderBook lock can become a bottleneck under extreme load. Future optimization could implement per-price-level locking.
2. **Order Removal**: The O(n) deque reconstruction during cancellation is the slowest regular operation. Alternative approaches include lazy deletion or specialized data structures.
3. **Decimal Arithmetic**: While necessary for precision, Decimal operations are 10-100x slower than native floats. Critical paths minimize Decimal operations where possible.

### Scalability Considerations

The current architecture scales well for typical cryptocurrency trading volumes:

- **Horizontal Scaling**: Multiple symbols operate independently, enabling simple sharding
- **Vertical Scaling**: Single-threaded performance handles 1000+ orders/second on modern hardware
- **Memory Scaling**: Linear memory growth with active order count

For high-frequency trading applications requiring >10,000 orders/second, optimizations could include:

- Lock-free data structures for read-heavy operations
- Custom memory allocators to reduce GC pressure
- NUMA-aware thread placement for multi-socket systems

The foundation provided by these data structures supports both current requirements and future performance enhancements while maintaining correctness and auditability essential for financial systems.