Order Book Benchmark Analysis

Performance Summary

Note on Definitions: "Depth" as used in "DeepBook" and "ShallowBook" refers to volume per price level.
Test Configuration: 1M total orders, Seed: 42
Hardware: CPU: M1 Pro,

Averaged Results (Multiple Runs)

Operation	Book Type	Runs	Avg Time/Run (s)	Performance vs ShallowBook
Add 1M Orders	ShallowBook (~10 orders/level)	10	1.1809	Baseline
Add 1M Orders	DeepBook (~10k orders/level)	10	0.5982	**2× faster**
Match 10k Orders	ShallowBook (~10 orders/level)	10	0.0043	Baseline
Match 10k Orders	DeepBook (~10k orders/level)	10	0.0045	~Same
Cancel 1k Orders	ShallowBook (~10 orders/level)	100	0.0014	Baseline
Cancel 1k Orders	DeepBook (~10k orders/level)	100	0.1134	**81× slower**

Single-Run Profiler Results

Operation	Book Type	Time	Scaling Component
Additions	ShallowBook	2.513s	PriceBook.add(): 0.638s

Operation	Book Type	Time	Scaling Component
Additions	DeepBook	1.916s	PriceBook.add(): 0.331s
Cancellations	ShallowBook	0.181s	deque.remove(): 0.033s
Cancellations	DeepBook	11.519s	deque.remove(): 11.343s

Key Findings

1. O(N) Queue Cancellations Favour Shallower Books

Observation: DeepBook cancellations are 81× slower than ShallowBook (0.1134s vs 0.0014s per 1k cancellations)

Root Cause: The `collections.deque.remove()` operation at `price_level.py:143` exhibits O(N) complexity:

- **ShallowBook:** deque.remove() = 0.033s for 100k cancellations (searching through ~10 orders per level)
- **DeepBook:** deque.remove() = 11.343s for 100k cancellations (searching through ~10k orders per level)

The performance slows because deque.remove() must linearly scan the deque to locate the target order. With 1000× more orders per level, the operation consumes **98.5% of total cancellation time** in DeepBook scenarios.

Fix: A hash-linked list cancellation time-complexity may be reduced to O(1). However, I decided to use standard python data-structures for simplicity.

2. O(log N) Heap Operations Favor Deeper Books

Observation: DeepBook additions are 2× faster than ShallowBook (0.5982s vs 1.1809s average per 1M orders). In particular PriceBook.add(), which pushes to the heap, takes twice as long for DeepBook as for ShallowBook.

Root Causes:
Heap operations at the PriceBook level (`price_book.py:60`) exhibit O(log N) complexity where N = number of unique price levels:

- **ShallowBook:** More unique price levels (~100k) → more heap push/pop operations

- **DeepBook:** Fewer unique price levels (~100) → fewer heap operations, despite more orders per level

ShallowBook creates many more unique price levels (~100k vs ~100 in DeepBook), resulting in both more frequent heap operations AND each operation working on a larger heap.

3. Cache Effects on Order Book Performance

I thought the slowdown would be entirely explained by Heap operation complexity; however further inspection of the cProfile indicated an unexpected slowdown elsewhere.

The profiler showed `price_level.add()` took **twice as long** in ShallowBook compared to DeepBook, even though the operation itself is O(1).

```
def add(self, order):
    self.orders.append(order)
    self.volume += order.volume
```

Doing a bit of research I found this might be due to cache effects.

Testing the Hypothesis

I isolated the deque append operation to see if cache effects were real:

Number of Deques	Time (s)	Relative Performance
10	0.398	1.00× (baseline)
100	0.383	0.96×
1,000	0.394	0.99×
10,000	0.389	0.98×
100,000	**0.527**	**1.32×**
500,000	0.512	1.29×

Potential Explanation

There's a **clear performance cliff between 10k and 100k deques** - a 32% slowdown once we cross that threshold. Performance plateaus after 100k, suggesting we've exceeded some cache capacity limit and are now consistently hitting different memory (maybe L2 cache or main memory).

The exact threshold depends on hardware architecture and cache size, but the effect is measurable and reproducible.

Why the 2× Slowdown in Real Code?

The plain deque test shows a 32% cache effect. But the actual `PriceLevel.add()` method shows a 2× slowdown. The difference might come from **amplification through multiple attribute lookups**:

Each call to `add()` performs:

- `self.orders` - attribute lookup
- `self.volume` - attribute lookup
- `order.volume` - attribute lookup

With 100 price levels: All objects fit in cache, lookups are fast.
With 100k price levels: Objects scattered across memory mean each lookup is more likely to miss cache.

Result: The ~30% cache penalty gets multiplied across multiple operations per method call, producing the observed 2× overall slowdown.

Further testing would be required to verify whether the slow down can truly be attributed to cache effects.