HW5 Team HW

Jonathan Ran, Amy Wang, Victor Qian, Robert Xing

May 2025

1 Benchmark Result for Robinhood Hashing

Robinhood hashing should on average have O(1) of time complexity in terms of insertion and lookup. in the worst case scenario when many collisions occur, then the lookup time can be O(ln n). unordered map is also O(1) on average as the underlying structure is a standard hash table.

in the following table, we can see that that robinhood hash actually takes more time upfront in terms of inserting elements into the table because it will do the swapping at the insertion. By minimizing the distance between each key and its original position, this results a more balanced and efficient hash table compared to unordered map. in return, we get a faster lookup time compared to unordered map, which is the trade off we take here.

by having this open addressing, we have a more consistent probe lengths on average, and that will give us good spatial locality. the cons again is insertion cost can spike more if we need to steal multiple times in a probe chain.

Table 1: Benchmark Results (Average over 5 runs)

N	**Hash Table**	**Insert (ms)**	**Lookup (ms)**
100,000	Robin Hood	9.0	2.2
	`std::unordered_map`	5.4	1.4
1,000,000	Robin Hood	91.4	36.2
	`std::unordered_map`	148.8	44.2
10,000,000	Robin Hood	1365.6	385.4
	`std::unordered_map`	2474.8	672.4

2 Benchmark Result for Priority Queue

for our binary heap-based priority queue, the push/pop time complexity should be O(logn) because the underlying is basically a tree, and the height of the tree is in the order of log n. std:: priority queue also has the same big-o complexity.

From the table, the cost for push and pop is very similar. In this case, the priority queue is already a highly optimized version, and behaved nicely for pop operation. so if we need to remove the top orders from the order book, we probably want to use standard priority queue. on the other hand, if there's a huge amount of orders inflow, then we care about more for order-book insertion, which in consequence we should use the max-heap and min-heap.

Table 2: Priority Queue Benchmark Results (Average over 5 runs)

N	**PQ Type**	**Push (ms)**	**Pop (ms)**
100,000	Custom PQ	1.6	13.2
	`std::priority_queue`	1.4	9.0
1,000,000	Custom PQ	16.4	185.8
	`std::priority_queue`	19.0	137.8
10,000,000	Custom PQ	173.6	3643.2
	`std::priority_queue`	246.4	2846.6

3 Time series Processing using SIMD

we can see that by using SIMD vectorization, we can speed up on average 2x. Given the architecture of mac (we all have apple silicon chips), we can't used SSE/AVX2, so we had to use ARM NEON in our SIMD version.

Table 3: Time Series Processing Benchmark (Standard vs SIMD in sec)

Size	**Standard**	**SIMD**	**Speedup**
10,000	0.014271	0.006645	2.147594
100,000	0.075854	0.027300	2.778594
1,000,000	0.582419	0.273118	2.132481
10,000,000	5.883939	2.772008	2.122627

4 Optimized Order Book

in this last section, we used a hash table for the "optimized order book", and benchmarked the result against an ordered map ("the unoptimized order book"). not surprisingly, the optimized order book is really good at query by price level and delete orders because it's contiguous memory pool. the ordered map version is pretty good at inserts/modifications and top-of-book lookups, since it has lower per-operation overhead.

for the throughput - hash table gives us very stable O(1) bulk deletes and the whole scan, so we are able to fetch a large chunks of book in one batch.

for memory usage, the optimized version uses a single pre-allocated block, so it has lower fragmentation and has contiguous memory. on the other hand, the ordered map data structure can lead to allocater overhead and memory fragmentation. in the HFT setup, we'd lean towards using the hash table due to cache locality.

Table 4: Optimized vs. Unoptimized OrderBook Performance

Operation	**N**	**Optimized (s)**	**Unoptimized (s)**	**Ratio**
Add orders	100,000	0.04883	0.02799	1.75×
Modify orders	50,000	0.06281	0.05415	1.16×
Delete orders	25,000	0.02919	0.03322	0.88×
Query 10,000 price levels	10,000	0.00450	0.00688	0.65×
10,000 BBO lookups	10,000	0.0002316	0.0002057	1.12×