Choice of Data Structures and Algorithms

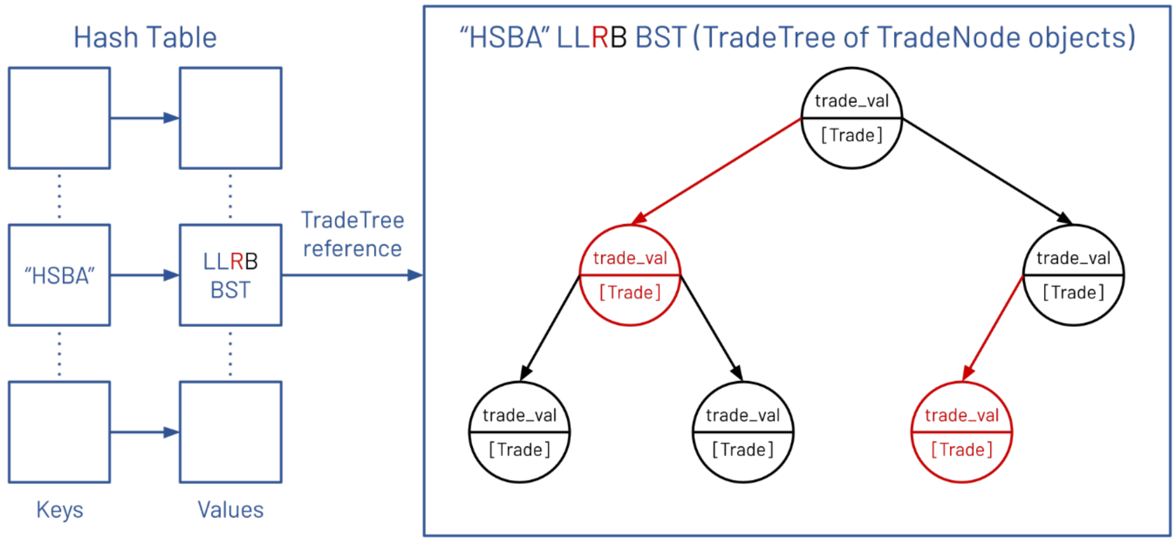
To support the various operations in the stock trading platform API, a combination of several data structures and algorithms was utilized.

Given that all the operations – except logging a transaction (`Trade`) – are called on a single stock name, the approach taken was to store the transactions associated with each stock separately. This would allow the runtime of the single-stock operations to be independent of the number of transactions stored under other stock names, enabling greater efficiency. To realize this separation of storing transactions, the hash table ADT was implemented using Python's built-in dictionary data structure. The reason for choosing a hash table is that it facilitates an ideal constant lookup time on average. By setting the keys of the dictionary to be the stock names (e.g., Barclays, HSBA, etc.), the API can quickly fetch the transactions under a certain stock for processing.

Regarding the chosen data structure to store the transactions under a particular stock name, the most crucial deciding factor was performance guarantees. Considering that the API mostly consisted of ordered operations based on the trade value of transactions, a balanced search tree ADT was the clear and optimal choice. This is because it maintains a logarithmic height regardless of the insertion order of transactions, significantly minimizing the worst-case time complexity of ordered operations. To realize this ADT, a left-leaning red-black binary search tree (LLRB BST) data structure was implemented (`TradeTree`), in which the key of each node (`TradeNode`) is the trade value of transactions. Using this ADT, the API can rapidly traverse through trade values and locate the desired transaction(s) to return.

Since it is possible that transactions can have the same trade value, the API needs to be able to handle that appropriately. To tackle this, it was decided that the value segment of an LLRB BST node should be a list of transactions with the same trade value as the key of the node. This seemed like a reasonable choice in comparison to storing transactions with the same trade value in separate nodes. This is because the separate node approach would make it unnecessarily complex to support the operations that return transactions associated with a single trade value. More specifically, once a node with the desired trade value is found, the separate node approach would require traversing through neighboring nodes, while the list approach can just return the list of transactions in the node.

For a depiction of the overall structure of the stock trading platform API, refer to the figure below.



With an overview of the design and implementation, the performance of the various API operations can now be analyzed in more detail.

Theoretical Analysis

Let T be the total number of transactions, S be the total number of stocks, and T_{stock} be the number of transactions with distinct trade values for some stock.

All API operations will require lookup in the hash table of stock-tree pairs, which is $O(1)$. Additional stocks may be added to the hash table trivially, which is $O(1)$, except for the rare case when the hash table must be resized. This would require S move operations but is sufficiently seldom to be negligible for performance analysis.

Hash Table Operation	Performance (Time Complexity)
Lookup and retrieval by stock name	$O(1)$
Insertion of additional stocks	$O(1)$ (With negligible $O(S)$ on update)

Due to the constant nature of these operations, they do not affect performance of other operations.

Once the tree of a certain stock has been fetched, there are several operations that can be performed:

- Logging a new transaction consists of an insertion on the tree and as such only requires a single tree traversal followed by tree balancing, which is bounded by $\Theta(\log(T_{stock}))$. This is implemented recursively, requiring $\Theta(\log(T_{stock}))$ function calls and no additional space.
- Returning sorted transactions requires a full traversal of all transactions, bounded by $\Theta(T_{stock})$, and pulling out all node transactions into a list, which requires space complexity $\Theta(T_{stock})$. This is implemented recursively and requires $\Theta(T_{stock})$ function calls.
- Retrieving minimum transactions requires a single traversal of the tree, along with returning a reference to the list stored in the tree of all transactions with trade value equal to the minimum for that stock. Thus, the time complexity is again bounded by $\Theta(\log(T_{stock}))$ and space complexity is constant due to returning a reference.
- Retrieving maximum transactions is very similar, requiring a single tree traversal, bounded by $\Theta(\log(T_{stock}))$, and returning a reference to a list of maximum transactions.
- Retrieving floor and ceiling transactions are very similar operationally to minimum and maximum transactions, needing a single tree traversal, with performance bounded by $\Theta(\log(T_{stock}))$, and constant space to return a list reference.
- Range transactions behavior is subtly more complex. Supposing K values are included in the specified range, then a single tree traversal, bounded by $\log(T_{stock})$, is required to find the first value in the range (or to determine no values lie in the range), and K further operations are required to retrieve values. This function is recursive, so it requires $\Theta(K + \log(T_{stock}))$ stack calls, as well as space to return all values in the range, bounded by $\Theta(K)$.

API Operation	Performance (Time Complexity)	Additional Space Complexity	Call Stack Requirements
Log transaction	$\Theta(\log(T_{stock}))$	$O(1)$	$\Theta(\log(T_{stock}))$
Sorted transactions	$\Theta(T_{stock})$	$\Theta(T_{stock})$	$\Theta(T_{stock})$
Min transactions	$\Theta(\log(T_{stock}))$	$O(1)$	$O(1)$
Max transactions	$\Theta(\log(T_{stock}))$	$O(1)$	$O(1)$
Floor transactions	$\Theta(\log(T_{stock}))$	$O(1)$	$O(1)$
Ceiling transactions	$\Theta(\log(T_{stock}))$	$O(1)$	$O(1)$
Range transactions (with K transactions in range)	$\Theta(K + \log(T_{stock}))$	$\Theta(K)$	$\Theta(K + \log(T_{stock}))$

Overall space complexity is bounded by $\Theta(T)$, the total number of transactions. Generally, best case performance is bounded by $\Omega(1)$, when only the root is retrieved, but this is uninteresting. Worst case is equal to the average case, except where explicitly specified otherwise.

Experimental Analysis

Operation of Experimental Framework

The framework is parameterized with the following integer arguments: M, the maximum number of transactions to be logged; N, the step size; and R, the number of trials to take an average execution time from. For each trial, the framework begins by generating M transactions under a particular randomly chosen stock name and starts logging them. For every N transactions logged, the time taken to log an additional transaction is recorded, and then all other API operations are examined under several different conditions. This process is repeated twice; once with M transactions in random order, and once more with M transactions in order of decreasing trade value. The reason each trial is isolated to a particular random stock is that hash table lookup performance is constant. This means that the average execution times for each API operation will converge to their theoretical asymptotic bound regardless of the chosen stock names. The figures below demonstrate the performance of the API with values $M = 10{,}000$, $N = 100$, and $R = 20$, which were chosen to generate meaningful graphs.

Graphical Analysis

Figure 1: Log Transaction Performance

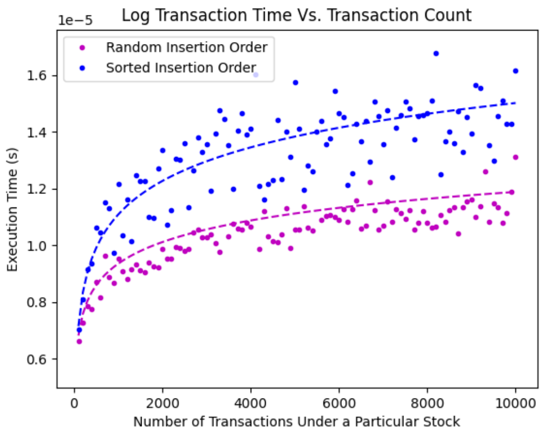


Figure 1 depicts the performance of the log transaction operation under random insertion order and insertion in order of decreasing trade value. It is evident that execution times grows logarithmically in both cases, which aligns with theoretical expectations. However, performance is marginally worse when insertions are in decreasing order. This is likely because there are significantly more calls to the LLRB BST balancing methods such as rotations and color flips in this case. While the two cases have a minor performance disparity, the guaranteed logarithmic bound highlights an advantage of the chosen data structure compared to an unbalanced BST. Then again, an unbalanced BST may perform slightly better with random insertions due to there being no calls to balancing operations at all.

Figure 2: Minimum Transactions Performance

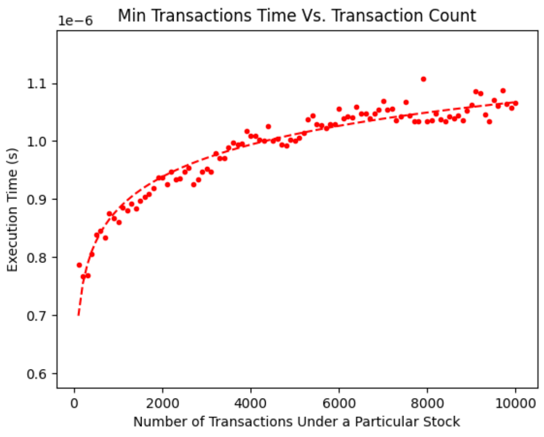
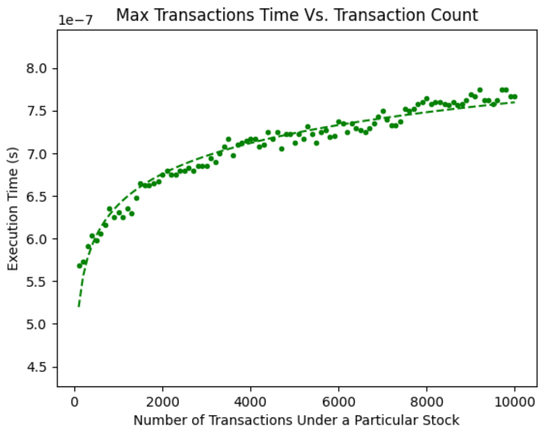


Figure 3: Maximum Transactions Performance



Figures 2 and 3 illustrate the performance of the minimum and maximum transactions operations respectively. Like for the log operation, there is a clear logarithmic growth trend in the execution time of both operations as expected. An interesting result to point out is that the minimum operation performs notably worse than the maximum operation. A potential rationale for this is that the implemented RB BST is left leaning, which means that there will be a greater number of nodes to traverse through to get to the minimum value in comparison to the maximum value.

Figure 4: Floor Transactions Performance

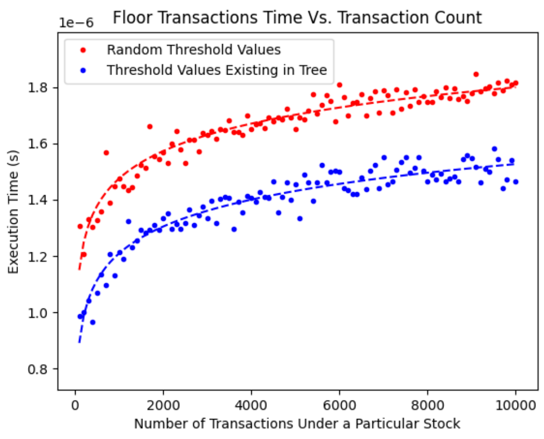
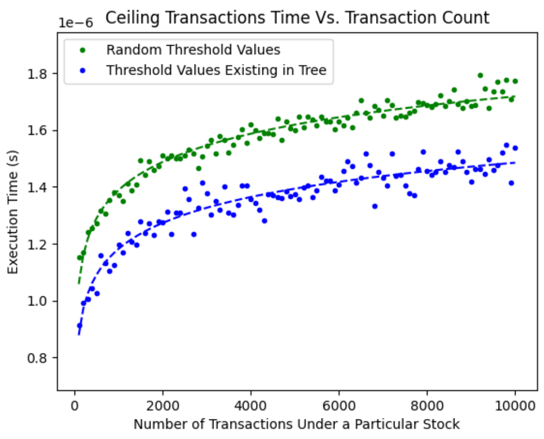


Figure 5: Ceiling Transactions Performance



Figures 4 and 5 portray the performance of the floor and ceiling transactions operations respectively, examining the cases of random threshold values as well as threshold values that already exist in the LLRB BST of a particular stock. As theoretically analyzed, the asymptotic growth of execution time is distinctly logarithmic for both operations. Looking at the variation in performance between the two different cases of threshold values, performance is moderately better for existing threshold values. This is likely because random threshold values are probabilistically non-existent in the LLRB BST, requiring the floor and ceiling operations to traverse through its entire height. Whereas, in the existing threshold values case, the floor and ceiling operations would return somewhere in the middle of the height of the LLRB BST, meaning fewer nodes are visited which takes less time overall.

Figure 6: Sorted Transactions Performance

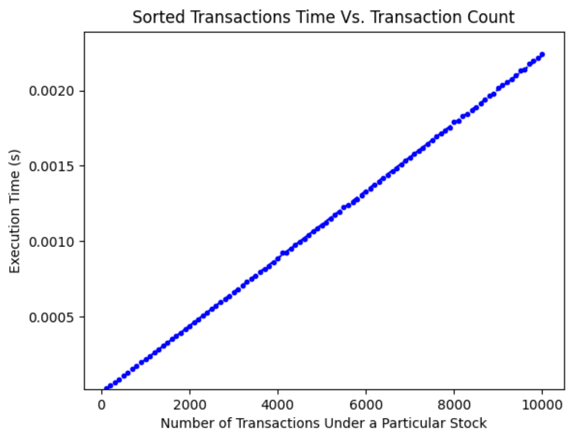
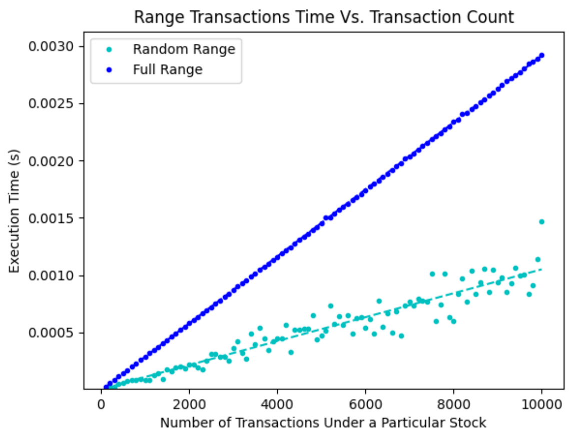


Figure 7: Range Transactions Performance



Figures 6 and 7 visualize the performance of the sorted and range transactions operations respectively, also looking at the cases of a random and full range for the range operation. As expected, the execution time of the sorted operation grows linearly, however, the experimental performance of the range operation provides insight on its theoretical analysis. Fetching the full range takes roughly double the time of a random range since it requires visiting all the nodes in the LLRB BST, mirroring the performance of the sorted operation. This means that while the expected bound on the range operation is $\Theta(K + \log(T_{stock}))$, it appears that $K = \frac{T_{stock}}{2}$ on average given that $K = T_{stock}$ for a full range. This suggests that the linear term K dominates the logarithmic term $\log(T_{stock})$ in general.

Conclusion of Results

Drawing from our experimental analysis, we can conclude that the operations have logarithmic performance due to the nature of binary trees, except the sorted and range operations because they involve scanning through all nodes. Moreover, we can establish that the logging of random transactions is highly efficient, as there are fewer balancing calls than in the case of insertions in sorted order. This makes the platform ideal for stock trading, as transactions are constantly fluctuating and unpredictable.