COMP0005 Algorithms – Stock Trading Coursework | Group 5 Report

Core implementation

Our platform implements a 'directory' of binary trees accessed through a dictionary, to maintain a record of stock transactions in a sorted manner, categorised by the company traded. Hence, keys in this master dictionary derive company names, and their values comprise individual Red Black search trees. This 'binning' technique lends itself to the efficiency of required API operations at scale since they always involve querying a particular company's trade history at one given time.

Data Structures

Hash-table (Python Dictionary): The companies are stored as strings in a dictionary, access operations have possible $O(N)$ (worst) or $O(1)$ (average) time complexity. After hashing the 12 supplied stock names, no collisions were found, so in fact worst and average time complexity is constant, at $O(1)$.

Transaction Tree: Transactions are stored in a Left-Leaning Red Black (LLRB) tree, this approach guarantees logarithmic time i.e., $O(\log N)$ for insertion and search operations, since the maximum traversal is the tree's height, worst case: $2\log N + 1$ with N being the number of nodes in the tree. The tree is self-balancing and holds transactions sorted by trade value.

Transaction Nodes: We specialise the nodes of our tree to store multiple transactions under the same key, within a list at each node. This circumvents the issue of having duplicate keys in a binary tree, because trade value is a product of two values and therefore is not necessarily unique. We also adapted the insertion operation to identify duplicate transactions and append them to an existing node's transaction list.

Transaction Objects: We unpack tuple-formatted transaction records into a transaction object. These store price, quantity, and timestamp of logged transactions, as well as their computed trade value. This avoids recomputing trade value, which is used in many comparison operations since it is our sorting key. Transaction objects can be compared directly by their trade value using our overrides for __**eq**__ and __**lt**__ relied on by $<$ and $==$ operators. This solves the problem of float comparisons that would otherwise fail equality checks. We check whether two transactions' trade values are within a specified absolute tolerance (0.001).

Algorithms

logTransaction: A transaction record tuple is unpacked into transaction object. We tried parsing the date-string as a float, since this reduces space usage by a factor of 2, but this was a costly operation to perform for each logged transaction timewise. The insert algorithm checks if a node in the relevant company tree exists, generating a new node if necessary, and this transaction object becomes both the node key and the first item in the node's transaction list.

sortedTransactions: Our sorting function uses a nested function to avoid creating more than one list to accumulate list elements. We recursively traverse the tree in-order, using the **extend** operation to concatenate the sorted list with each node's individual items. In python, this is implemented as successive **append** operations, with a respective amortised time complexity of $O(1)$, resulting in time complexity of $O(M)$ appends, where M is all the transactions in the respective tree.

minTransactions/maxTransactions: The function(s) will traverse down the left (min) or right (max) subtrees recursively until the most extreme node is found, and then the current node is returned as a list. If the chosen company had no transactions so the tree is empty, an empty list is returned.

floorTransactions/ceilingTransactions: We used an iterative approach to traverse the tree until the node just below or above the threshold is found, at which point the node's list is returned, otherwise an empty list.

rangeTransactions: Again, a nested function is used to avoid creating multiple lists; this function uses a recursive approach to traverse down the tree and accumulate transactions from nodes whose keys are within the given range. This makes use of the **extend** list operation, as **sortedTransactions** does above.

Theoretical Analysis

To undertake theoretical analysis, we define the following:

N = Number of nodes, M = Total number of transactions stored in the tree.

We also assume, as detailed in the transaction tree section above, our tree balances correctly, and therefore its height is bounded by $2\log N + 1$.

logTransaction: Logging a new transaction has an average (and worst) time complexity of $O(\log N)$, this is because of the properties of a LLRB which gives the time complexity of $N\log(N)$ when inserting N transactions in an empty tree. Comparing keys pairwise like this follows the Stirling approximation where log(N!) becomes roughly $N\log(N)$. Inserting a node will have recursive stack of $\log(N)$, and the final space complexity is $O(N)$ as it's an LLRB tree.

minTransactions/maxTransactions: The operations have time complexity equal to the height of the tree for the average case $O(\log N)$, worst case is $2(\log N) + 1 \approx \log(N)$, as its equal to the height of the tree when the tree is in the worst case. As these operations require the furthest node away from the root node hence traversing the whole tree to the left or right, the recursion stack has a maximum equal the height of the tree hence $O(\log N)$.

floorTransactions/ceilingTransactions: The time complexity is like that of the minimum and maximum operations however, it has a space complexity of $O(1)$ since an iterative approach was used.

sortedTransactions: The operation has an average/worst time complexity of $O(N + M)$ assuming the tree is balanced, as the function would traverse through all the nodes N, and append all the transactions M . The space complexity is $O(M + \log N)$ as the size of the array is bounded by M and the recursion stack has a maximum of $\log N$ when collecting the transactions in the furthest node away from the root node.

rangeTransactions: The operation has an average/worst time complexity of $O(Y + Z)$ where Y is all the nodes with keys in the threshold and$Z$ is all the transactions in the threshold, the function will traverse **nodes with keys in the threshold**, Y and append Z transactions. The space complexity is $O(Z + \log Y)$ as the size of the array is bounded by Z and the recursion stack has a maximum of the height of the node with key closest and in range of the threshold $O(\log Y)$.

Experimental Analysis

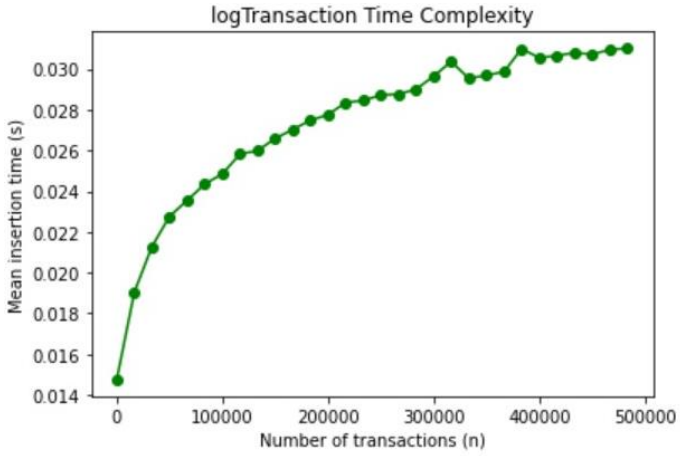

Figure 1: Average time for a transaction to be inserted into a tree, for all twelve stocks, for a given tree size, n.

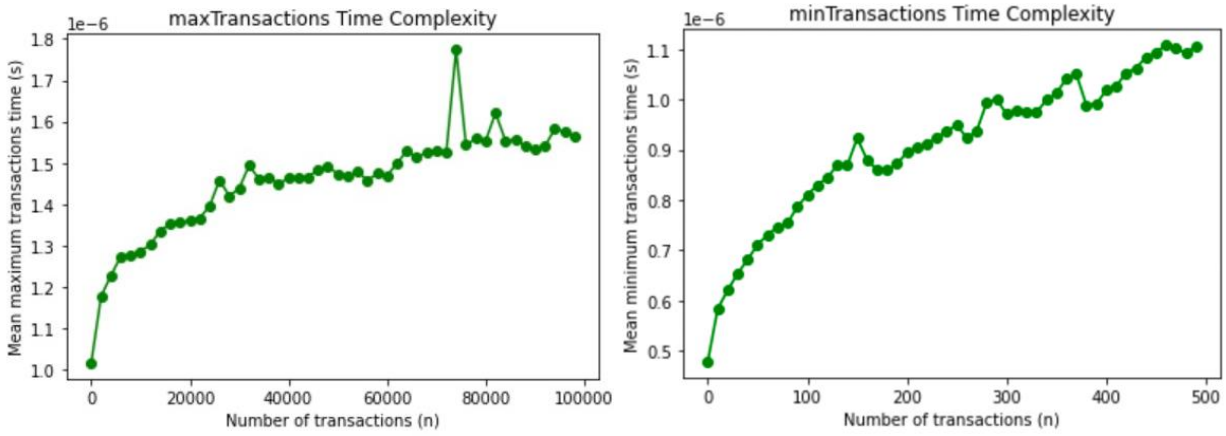

*Figure 2: Average time to get a maximum transaction by **tradeValue** for a given number of transactions, n (left). Average time to get a minimum transaction by **tradeValue** for a given number of transactions, n (right).*

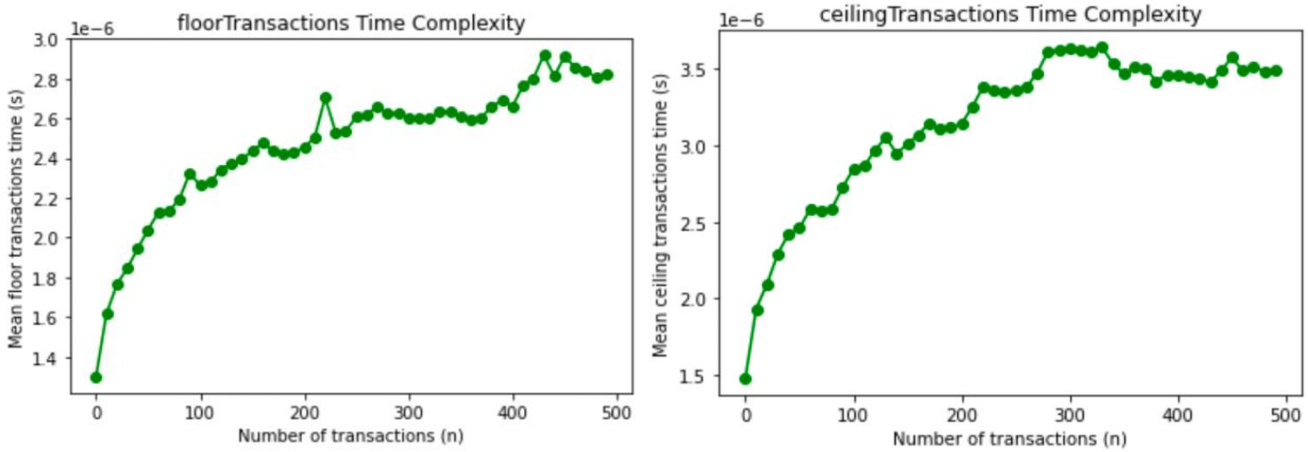

*Figure 3: Average time for finding the trade with the biggest **tradeValue** below a threshold value, for a given number of transactions, n (left). Average time for finding the trade with the smallest **tradeValue** above a threshold value, for a given number of transactions, n (right).*

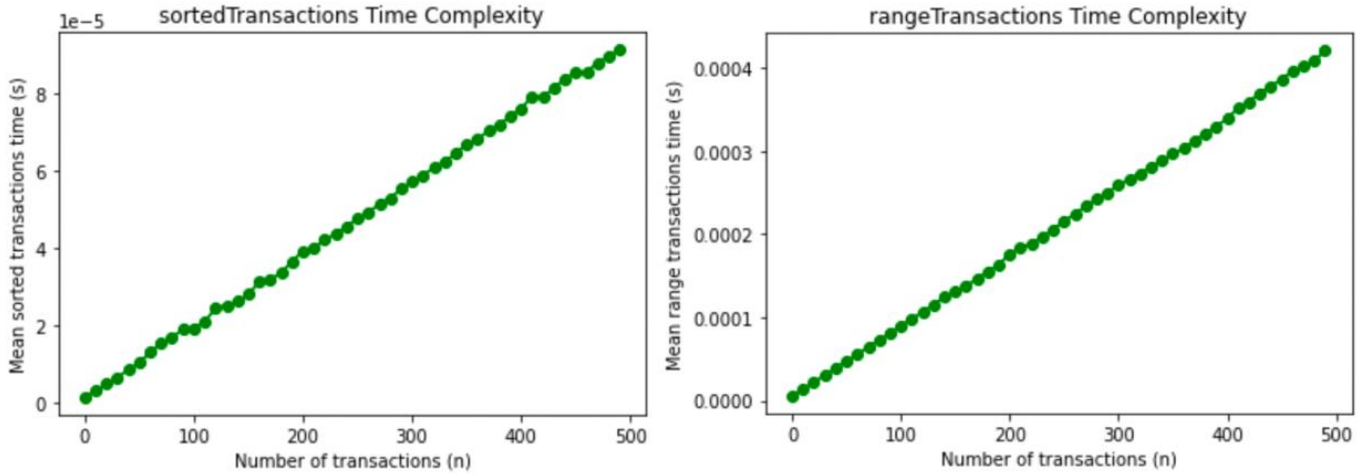

Figure 3: Average time for the platform to return a sorted list for a given number of transactions, n (left). Average time taken for to return a list of transactions within two parameters, for a given number of transactions, n (right).

Duplicate keys

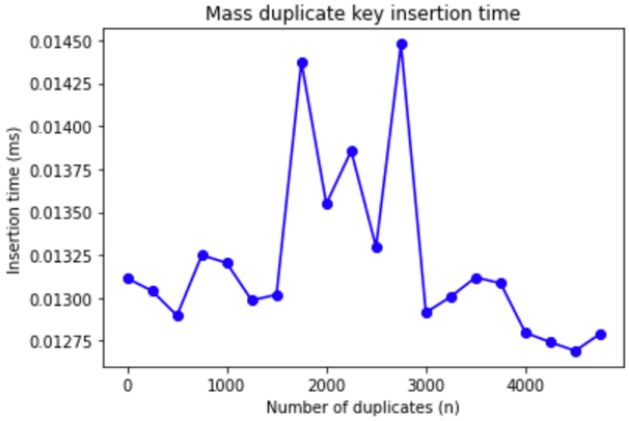

Figure 4: Mean insertion time when randomly picked duplicates into all trees.

An important test for our platform is inserting many duplicates. Given our implementation, where tree nodes store duplicates as a list, we want to confirm that adding duplicates anywhere in the tree, where an arbitrary number of duplicates already exists, always takes the same time. This is because duplicates are appended to a node's list which is a constant time operation. This graph illustrates that this is indeed the case – of course there is some random distribution based on the traversal depth - how deeply the random node exists in the tree.

Our experimental analysis confirmed all assumptions made in the phase of theoretical analysis, by proving time complexities previously stated through timed tests and graphing. Moreover, apart from testing times for different operations, the framework also includes the correctness tests for their return values. API is tested through comparing the return values with correct values found by iteration of reference list (transaction data), itself. Once the cell for framework is run, API functions are tested for transactions of all companies.

While being executed, testing functions will be printing results (both time and correctness) below the cell confirming the accuracy of the platform, as well as the time taken for its execution. Some of the tests taken, not directly connected to API functions themselves, but very important for the platform overall are checking whether there are collisions in the dictionary of companies, as well as checking the height of tree after logging arbitrary number of transactions. Some of the notable tests include checking floor and ceiling operations, which are studied in two main cases: threshold value existing inside the tree and threshold value not existing in the tree. In both cases, it is confirmed that correct values are being returned, with expected timing, even with multiple transactions of the same **tradeValue**.

Conclusion

Our trading platform is the most efficient when there are a lot of transactions of the same **tradeValue** as this keeps the number of nodes within the tree minimal – giving faster access times. One possibility that could cause inefficiencies in our platform would be the addition of numerous more companies, this may cause collisions in the dictionary, would hence result in slower access times. Our trading platform is least efficient when all transactions have unique **tradeValues**, as a new node would need to be created for each transaction. Although being least efficient with unique **tradeValues**, it still guarantees timing complexities stated in previous analyses, due to implementing balanced search tree, with **tradeValues** as keys.