COMP0005 Algorithms: Technical Report

Imaad Zaffar, Aaryaman Sharma, Arvind Sethu, Aiste Mikstaite

1 Implementation

1.1 Data Structures

For our implementation, we used a combination of a **hash table** and **left-leaning red-black binary search trees** to store, sort and look up transactions.

We have a **Transaction** class, which is used to store the transaction data. The timestamp is stored as a date string, stock quantity is stored as an integer, and the stock price is stored as a float.

We chose a **LLRB BST** to store individual transactions, because the average time complexity of standard operations such as insertion and search are $\log N$, which is more efficient compared to other tree structures and easier to implement than multiple trees. The tree is sorted by the trade value of a transaction, with this being assigned to the 'key' field of each node. An array of all transactions with the same trade value is stored in the 'value' field.

Within the tree, the nodes are stored in the **Node** class. The tree is sorted by the trade value of a transaction, with this value being assigned to the 'key' parameter of the node. The rest of the details regarding the transaction are stored as an array, with this array being assigned to the 'value' parameter of the node.

We chose a **hash table** to store the trees for a specific stock name. The keys of the hash table are the stock names, and the values are instances of the Transactions class, which stores the BST.

Using these properties, we create an instance of the Transactions class, which, as mentioned stores the BST corresponding to a specific stock name and contains all the methods required for the API.

1.2 Algorithms

For **logTransaction**, we first make a new Transaction object and use a standard BST insertion algorithm to insert a node with trade value as the key of the node and the Transactions array as the value. If the trade value already exists in the tree, we append the transaction to the 'value' field of the node.

For **minTransactions** and **maxTransactions**, our tree automatically stores the nodes with min and max values whenever a new node is inserted. Thus to find the min/max, we just lookup the values of the stored nodes.

For **floorTransactions**, we recursively traverse down the left child nodes of the current node. Once we encounter a node that has a lower trade value than the threshold value, we store the node and then recursively traverse through the node's right subtree. If we find a node with a better floor value, we return its values, otherwise, we return the values of the node that we stored. During the traversal, if we find a node with the same trade value as the threshold, we simply return its values.

For **ceilingTransactions**, we recursively traverse down the right child nodes of the current node. Once we encounter a node that has a higher trade value than the threshold value, we store the node and then recursively traverse through the node's left subtree. If we find a node with a better ceiling value, we return its values, otherwise, we return the values of the node that we stored. During the traversal, if we find a node with the same trade value as the threshold, we simply return its values.

For the **rangeTransactions** function, we use a modified version of inorder traversal, where we only traverse the left child nodes if the trade value of the current node is greater than the lower bound of the provided range. If this condition is violated, we move back up the recursion stack. We then check if the current node trade value is inbetween the range, and if so, we append its values to a list. At the end, we return the list containing all the node values within the given range.

2 Theoretical Analysis

2.1 Time Complexity

The time complexities for both average and worst case of the different functions are:

logTransactions() - $\Theta(\log N)$ - traversing through height of BST

sortedTransactions() - $\Theta(N)$ - inorder traversal of BST

minTransactions() - $O(1)$ - constant time lookup from stored value in BST

maxTransactions() - $O(1)$ - constant time lookup from stored value in BST

floorTransactions() - $\Theta(\log N)$ - traversing through height of BST

ceilingTransactions() - $\Theta(\log N)$ - traversing through height of BST

rangeTransactions() - $\Theta(N)$ - slightly modified inorder traversal of BST

2.2 Space Complexity

For the **min/max** functions, the space complexity is $O(1)$ because our tree already stores the nodes with min/max values.

For the **log, floor**, and **ceiling** transaction functions, the space complexity is $O(\log N)$ in both the average and worst cases. This is because the size of the call stack for these recursive functions is proportional to the height of the red-black tree. Since red-black trees have a height proportional to $\log N$, the space complexity of these functions is $O(\log N)$.

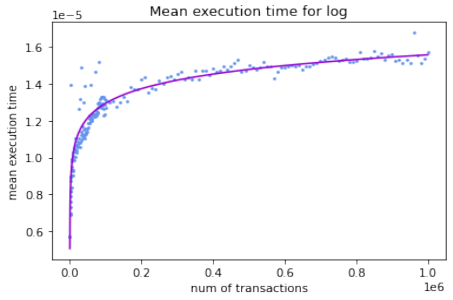
For the **range** and **sort** functions, the recursive call stack for inorder traversal also has a size proportional to $\log N$. However, in these functions, a list is passed by reference to store values during the inorder traversal. Since this list's size is proportional to the number of nodes, the average and worst case space complexities are $O(\log N) + O(N)$, which reduces to $O(N)$ complexity.

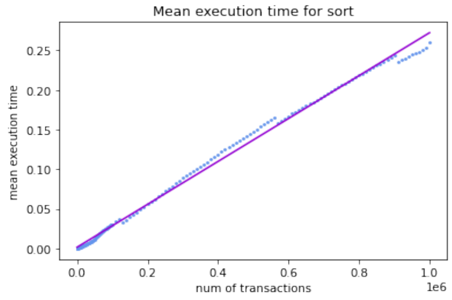
3 Experimental Analysis

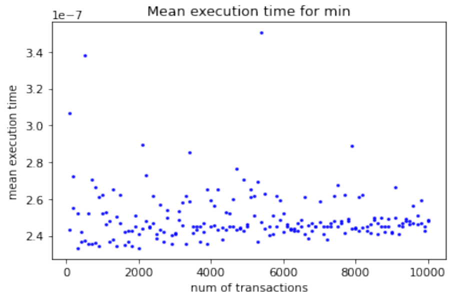
3.1 Description

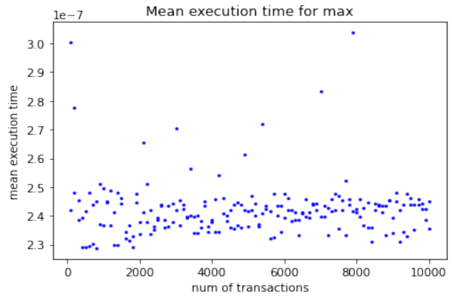
In order to check the efficiency of our functions, we executed each function multiple times and took the mean time. We repeated this at regular intervals, from 0 to 1000000 transactions. Then, we plotted graphs from these results and finally we plotted the lines of best fit.

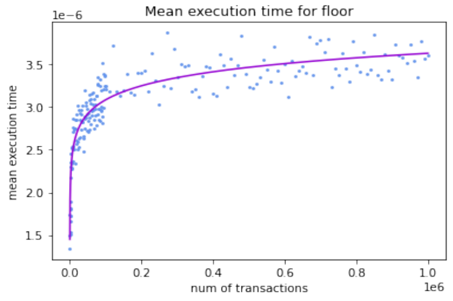
3.2 Results

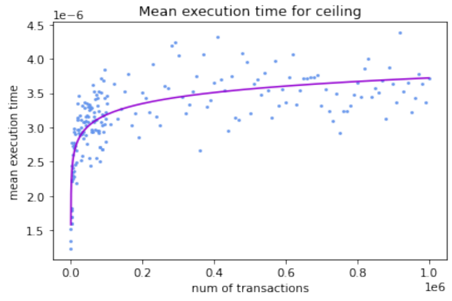


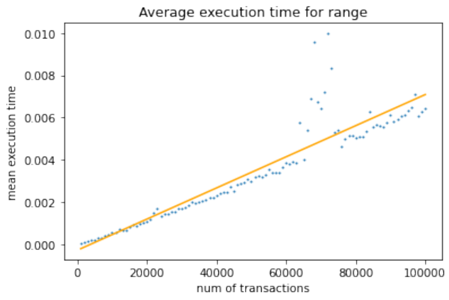












3.3 Discussion of Results

The function which calculates the time it takes to log a certain number of transactions is logarithmic. Since the time complexity of the function in the average case is $O(\log N)$, the graph does not increase significantly as the maximum and minimum time it takes to execute the operations differ by 1×10^6 which proves the theoretical estimated time complexity.

The average execution time for the sort function can best be described using a linear relationship, as theorized in our theoretical analysis where the mean execution time linearly grows with the number of transactions.

As predicted in our theoretical analysis, the functions for min and max transactions have a constant time of processing which does not depend on the number of transactions.

The graphs displaying time taking to process the varying number of floor and ceiling transactions are logarithmic which the expected time complexity calculated theoretically. The difference in the time it takes to return the values is 2.5×10^6s proving that increasing the number of transactions does not affect the efficiency of the algorithm.

The time it takes to return a list with transactions within a given range is of a linear complexity which accords with the calculated theoretical estimation. However, a significant number of time values scatter below the linear graph due to different sizes of ranges the program is required to return the lists of transactions.

4 Conclusion

The main advantage of our solution is the efficient time and space complexity of the min/max functions, which involve constant time lookups, and the floor and ceiling functions, which only involve traversals limited to the height of the red-black tree.
However, logging new transactions in the tree is less efficient compared to other possible implmentations such as adding new transactions to the end of an 1-D array. Along with this, our solution may not be very effective in scenarios where transactions need to be deleted.
Nonetheless, since insertion in a red-black tree maintains sorted order, our trees can be easily converted to a sorted list via inorder traversal. This means the sort function in our solution also has an efficient time and space complexity.

Our solution will work well for scenarios where there are a large amount of transactions that need to be logged, stored and accessed in a quick amount of time.