Real-Time Stock Market Data Tracker

Overview

C++ is used to build the project. In the project, a volume-based AVL tree, a price-based AVL tree, and one hash map are used. The "Market" class contains the data structures used, while the "Stock" class contains the stock data.

Implementation of the Data Structure

1. Hash map
   Using the library <unordered_map>, the hash map "unordered_map <int, Stock*> stockMap" is initialized. The hash map contains two parameters: the key is the stock's ID, and the value is the stock's data.
2. AVL tree
   In the "Market" class, the following member functions are used to maintain the AVL tree:
   - **int height(Node* node)** returns the height of the tree or subtree.
   - **void updateHeight(Node* node)** updates the height of the tree or subtree.
   - **int balanceFactor(Node* node)** returns the balance factors of the subtree. The balance factor(bf) is calculated by subtracting the height of the left subtree from that of the right subtree.
   - **Node* rightRotate(Node* y)** is used for right rotation.
   - **Node* leftRotate(Node* x)** is used for left rotation.
   - **Node* balance(Node* node)** balances the tree by determining the balance factor(bf) of the node.

Inert-new-stock(x, p)

First, **"stockSearch(id)"** will be used to check if the stock exists by using a hash table. The required time complexity is **O(1)**. If "stockSearch(id)" returns "nullptr", which means the stock does not exist, the corresponding stock will be created.

Then, the following action will be taken:

1. **"stockMap[id] = stock"** inserts the new stock into the hash map. The required time complexity is **O(1)**.
2. **"insertPrice(Node* node, Stock* stock)"** inserts the new stock into the price-based AVL tree. The required time complexity is **O(logn)**.
3. **"insertVolume(Node* node, Stock* stock)"** inserts the new stock into the volume-based AVL tree. The required time complexity is **O(logn)**.

Therefore, the overall time complexity is **O(1) + O(logn) + O(logn) + O(1) = O(logn)**.

update-price(x, p)

First, **"stockSearch(id)"** will be used to check if the stock exists by using a hash table. The required time complexity is **O(1)**. If "stockSearch(id)" does not return "nullptr", then the following actions will be done:

1. **"deletePrice(Node* node, float price, int id)"** deletes the node with the original price. The required time complexity is **O(logn)**.
2. **"insertPrice(Node* node, Stock* stock)"** inserts the new stock into the price-based AVL tree. The required time complexity is **O(logn)**.

Therefore, the overall time complexity is **O(logn) + O(logn) = O(logn)**.

increase-volume(x, v_{vic})

First, **"stockSearch(id)"** will be used to check if the stock exists by using a hash table. The required time complexity is **O(1)**. If "stockSearch(id)" does not return "nullptr", then the following actions will be done:

1. **"deleteVolume(Node* node, int vol, int id)"** deletes the node with the original volume. The required time complexity is **O(logn)**.
2. **"insertVolume(Node* node, Stock* stock)"** inserts the new stock into the volume-based AVL tree. The required time complexity is **O(logn)**.

Therefore, the overall time complexity is **O(logn) + O(logn) = O(logn)**.

lookup-by-id(x)

A hash map will be used to look up the stock by ID. The key of the corresponding element is the stock's ID, and the value of the corresponding element is the stock's data. Therefore, the required time complexity is **O(1)**.

price-range(p_1, p_2)

To store all IDs of all stocks whose prices are in the inteval $[p_1, p_2]$, a vector **"priceRangeIDs"** is initialized in the class "Market". The **"collectPriceRange(Node *node, float lowerBound, float upperBound)"** function will collect the required IDs recursively in the price-based AVL tree up to k, where k is the number of elements in the interval $[p_1, p_2]$. Therefore, the required time complexity is **O((1+k)logn)**. Then, all the IDs in the vector will be printed out.

max-vol()

Return the rightmost node in the volume-based AVL tree. The required time complexity is **O(logn)**.