CSCI2100B Project Report (2025)

Chung King Lam 1155205651

1. **Table of content**

1. Overview

This report describes the implementation and evaluation of a real-time stock market data tracker developed as a course project for CSCI 2100. The system allows inserting, updating, and querying stock records based on ID, price, and trading volume.

Key operations include:

• insert-new-stock(x, p): insert a new stock with an ID x whose current price is p.

• update-price(x, p): update the stock price with ID x to p.

• increase-volume(x, vinc): increase the volume of the stock with ID x by vinc.

• lookup-by-id(x): find the price and volume of the stock with ID x if such a stock exists.

• price-range(p1, p2): return the IDs of all the stocks whose prices are in the interval [p1, p2].

• max-vol (): return the highest volume among all the stocks and one stock having this volume.

I use Python for this project.

2. Data Structure

2.1 ID Hash table

```
class id_hash_table:
    def __init__(self):
        self.m = 5000 #numebr of bucket hash table
        self.p = 1000019 #smallest prime larger than U
        self.a = 792481 #alpha between {1,2,...,p-1}
        self.b = 12839 #beta between {0,1,2,...,p-1}
        self.buckets = [[] for i in range (self.m)]
```

A hash table with universal hashing is created to store all the Node, where each node stores ID, price, and volume, while ID is treated as key value. The hashing is done by a function from universal hash family (((a * key +b)%p)%m). The value of the variable is shown in the picture above. This reduces the collision in each bucket, thus guaranteeing O(1) expected query time.

The hash table is implemented as a 2D dynamic array, where each row represents a bucket. The hash function computes an index in the range [0, 4999], and the corresponding entry is stored in the row at that index. Within each row (bucket), nodes are stored according to the hash value of its key value (ID).

This hash table is used to support the implementation of insert-new-stock(x, p), update-price(x, p), increase-volume(x, vinc) and lookup-by-id(x).

2.2 Price_AVL_Tree

An AVL_tree is implemented to mange the price data. This AVL tree use (Price, ID) as key . The AVL tree will first compare the price, if the price are the same, the tree will decide it's direction by comparing the ID. This AVL tree supports insert and delete operations and it is done in a recursive manner. Each node stores a left pointer and a right pointer that points to the left subtree or right subtree

When (price, id) is inserted, the algorithm will compare the inserted price with the key at different level from the root to bottom. If the inserted value is larger than the key, go right, vice versa, until the algorithm reaches to an empty node (e.g. self.left = None). Then, the inserted value will be inserted to

the empty node as a key. The balance-checking process will start from bottom to top to ensure the tree is balanced. The algorithm will obtain the height of the left subtree and right subtree at each level. If any subtree is higher by 2, then the algorithm will start the rebalancing process. The algorithm will then decide whether it is a LL, LR, RR, or RL case and do the rotation accordingly. (LL→right rotate, LR→right rotate then left rotate, RR→ left rotate, RL→left rotate then right rotate). As a result, the AVL tree is always balanced

For deletion, the algorithm will locate the target key from the root to bottom. If the target key is larger than the key that is traversed, then go right, until the algorithm locates the target key in the tree. Then, the algorithm will delete that node and replace it with the following principle:
Let u be the node to be deleted
Case 1: if u is the leaf node, simply remove it from the tree
Case 2: if u only has a left child, then replace u with the left child
Case 3: if u only has the right child, then replace u with right child
If there are two children, it is possible that u have a left or right subtree with more than two nodes. Therefore, we need to find the in-order successor in the right sub-tree.
Case 4: if u has two children, use minValueNode(node.right) to find the successor of u in the right sub tree. Then replace the deleted node with the succesor
After deleting and replacing the target node. Next, we need to do the balancing process from the bottom to the top as the insert operation.

This AVL tree can support update_price and price_range under specific time requirements.

2.3 Volume AVL_tree

It is implemented in the same way with price AVL_tree. The only difference is that the key is (volume, id) in this AVL tree. This AV: tree can support increase)volume and max_vol under specific time requirement

3. Key operation

3.1 Insert-new-stock(x,p)

Firstly, the algorithms will check whether the stock with the same ID already exists by querying in the ID hash table. The algorithm will go to the bucket of the to-be-inserted ID and check whether it exists; if so, ignore the operation. This validation wil be done by function check_duplicate(self, id). If the ID already exists, it will return 1, vice versa. O(1) expected time is guaranteed. After validation, three insert operations will be executed, namely insert_id(self,id,price), insert_price and insert_volume(self, node, id, volume). Firstly, the insert_id function will create a node (struct) for a new stock which contains its ID, price and volume. Afterward, the function will hash the new stock into the hash table according to its ID. This takes O(1) guaranteed expected time given that the hash function is universal. Next, the insert_price function will insert a key value pair (price, id) to the price-AVL-tree. This takes O(logn) time. Lastly, the insert volume function will insert a key value pair (vol, id) to the volume-AVL tree. All the volume of the new stocks is set to 0 initially. This will take O(logn) time. Since O(1)+O(1)+O(logn)+O(logn) = O(logn). The time complexity of this key function is O(logn).

3.2 update-price(x,p)

The algorithm will check if the ID exists first as the function above. After validating, three operations will be executed, which are get_and_update_price, delete_price and insert_price. The algorithm will get the current price of the stock with the given id, which is returned by the function get_and_update_price. Next, the new price will be updated in the hash table. This query and update cost O(1) expected time. After getting the current price, the node with the current price and given ID will be deleted from the price-AVL-tree. Next a key-value pair with the same given id and new price will be inserted in the price-AVL-tree. This finishes the update-price process and the price-AVL tree maintains perfectly. The delete and insert process cost O(logn) time. Since O(1) +O(1)+ O(logn)+ O(logn) = O(logn). This operation cost O(logn) time.

3.3 increase-volume(x, v_{inc})

After checking the ID does not exist in the hash table, four function will be excited, which is get_volume, update_volume, delete_volume and insert_volume. The algorithm will first get the current volume of the stock with the given id in the id-hash table, which is stored in the variable old_volume. Next, update_volume will be called, which add v_{inc} to the current volume, storing the sum as new_volume and updating the volume in the id-hash table. These can be done in O(1) time. Afterward, the node with given id and volume will be deleted from the volume AVL tree by calling delete_volume(self, id ,old_volume). Lastly, the key-value pair with the same given id and new volume will be inserted into the volume AVL tree by calling insert_volume(self, id, new_volume). The insert and delete operation can be done in O(logn) time. After these operations, the volume AVL tree maintains nicely. Since O(1) + O(1) + O(1) + O(logn) + O(logn) = O(logn), this key operation can be done in O(log) time

3.4 lookup-by-id(x)

This is a simple operation that can be done by querying in the hash table. The algorithm will first calculate the hash_index of the key(id) by the universal hash function. Then, it will go to the bucket with the corresponding hash index. Lastly, the algorithm will scan through the bucket to see if there is a matching ID. If yes, return a tuple that stores (ID, price, and volume) from the node that matches the given id. This takes O(1)expected time.

3.5 price-range(p1,p2):

This operation can be built by simply calling the range counting operation that is supported by the price AVL tree. I will further explain how is implemented here. The recursion in self.price.range_recursion is a modified in-order traversal of the price AVL tree to efficiently find all stock IDs with prices within a specified range [p1, p2]. The traversal order is to first explore the left subtree, then process the current node if its price lies within the range, and finally explore the right subtree. If the current node's price is less than p1, the left subtree is entirely skipped because all prices in the left subtree are smaller

than the current node's price and thus cannot satisfy the condition. Similarly, if the current node's price is greater than p2, the right subtree is skipped. This selective pruning ensures that only relevant parts of the tree are visited, resulting in a time complexity of $O((1 + k) \cdot \log n)$, where k is the number of matching nodes reported.

3.6 max-vol

This can be done by traveling the rightmost node of the volume AVL tree. It is implemented by a for loop to visit every right node until None is reached. This can be done in O(logn) time given that the tree is a balanced AVL tree.

4. Algorithm Design Layout

1. id_Node

- Purpose: Represents a stock record storing ID, price, and trading volume.
- Attributes:

id: Integer representing the stock ID.

price: Float representing the current stock price.

volume: Integer representing the trading volume (initialized to 0).

2. price_Node

- Purpose: Represents a node in the price AVL tree.
- Attributes:

id: Stock ID.

price: Stock price.

left, right: Child nodes.

height: Height of the node for AVL balancing.

3. volume_Node

- Purpose: Represents a node in the volume AVL tree.
- Attributes:

id: Stock ID.

volume: Trading volume.

left, right: Child nodes.

height: Height of the node for AVL balancing.

4. id_hash_table

- Purpose: Provides fast expected O(1) access to stock information by stock ID using universal hashing.
- Attributes:

m: Number of buckets (5000).

p: A large prime number (1000019).

a, b: Random constants for hashing.

buckets: A list of lists (dynamic buckets).

- Methods:

hash_index(id): Compute the bucket index.

check_duplicate(id): Check if a stock ID already exists.

insert_id(id, price): Insert a new stock into the hash table.

get_and_update_price(id, new_price): Update stock price and return old price.

get_volume(id): Retrieve the trading volume of a stock.

update_volume(id, volume_inc): Increase the volume of a stock.

get_stock_info(id): Retrieve the stock information.

5. price_tree

- Purpose: Maintains stocks ordered by price using an AVL tree to support efficient price range queries.
- Attributes:

root: Root node of the price AVL tree.

- Methods:

get_height(node): Return the height of a node.

get_balance(node): Calculate the balance factor.

right_rotate(node): Perform a right rotation.

left_rotate(node): Perform a left rotation.

insert_price(node, id, price): Insert a new stock into the price tree.

minValueNode(node): Find the node with the minimum price.

delete_price(node, id, price): Delete a stock from the price tree.

range_reporting(root, p1, p2, result): Collect stock IDs within the price range [p1, p2].

6. volume_tree

- Purpose: Maintains stocks ordered by trading volume using an AVL tree to efficiently find the maximum volume stock.
- Attributes:

root: Root node of the volume AVL tree.

- Methods:

get_height(node): Return the height of a node.

get_balance(node): Calculate the balance factor.

right_rotate(node): Perform a right rotation.

left_rotate(node): Perform a left rotation.

insert_volume(node, id, volume): Insert a new stock into the volume tree.

minValueNode(node): Find the node with the minimum volume.

delete_volume(node, id, volume): Delete a stock from the volume tree.

get_max_volume(root): Retrieve the stock with the highest trading volume.

7. stock_tracker

- Purpose: The main controller that integrates the hash table, price AVL tree, and volume AVL tree to perform all system operations.
- Attributes:

id: Instance of id_hash_table.

price: Instance of price_tree.

volume: Instance of volume_tree.

- Methods:

insert_new_stock(id, price): Insert a new stock.

update_price(id, new_price): Update the price of an existing stock.

increase_volume(id, volume_inc): Increase the volume of an existing stock.

lookup_by_id(id): Retrieve the stock's ID, price, and volume.

price_range(p1, p2): Return stock IDs within a price range.

max_vol(): Retrieve the stock ID and volume of the highest-volume stock.