StockTracker Implementation Report

Time Complexity Requirements

The project specifies the following operations with their required time complexities:

- insert_new_stock(x, p): Insert a stock with ID x and price p, O(log n).
- update_price(x, p): Update the price of stock x to p, O(log n).
- increase_volume(x, vinc): Increase the trading volume of stock x by vinc, O(log n).
- lookup_by_id(x): Return the price and volume of stock x, O(1).
- price_range(p1, p2): Return IDs of stocks with prices in [p1, p2], O(log n + k), where k is the number of stocks in the range.
- max_vol(): Return the maximum trading volume and its stock ID, O(1) for retrieval, O(log n) for heap cleanup.

Here, n is the number of stocks stored.

Data Structures and Algorithms

To meet these time guarantees, the StockTracker class employs a combination of data structures: a hash table, a Binary Search Tree (BST), a defaultdict for price-to-ID mapping, and a max heap with UUIDs for volume tracking. Below, each operation is analyzed with its supporting data structure and algorithm.

1. Hash Table (self.stocks)

- **Purpose**: Stores stock information (price and volume) for each stock ID.
- **Implementation**: A Python dictionary (self.stocks), where keys are stock IDs (x) and values are lists [price, volume].
- **Operations**:
    - **Lookup**: Checking if a stock exists (x in self.stocks) or retrieving its data (self.stocks[x]) is O(1) on average due to the hash table's efficient key-value mapping.

- **Insert/Update**: Adding or updating a stock (self.stocks[x] = [p, 0]) is O(1).
- **Usage**: Critical for lookup_by_id, insert_new_stock, update_price, and increase_volume to achieve fast access and updates.

2. Binary Search Tree (self.price_sorted)

- **Purpose**: Maintains stock IDs sorted by price for efficient range queries.
- **Implementation**: A custom BinarySearchTree class with BSTNode objects, each storing a price and id. The BST supports:
  - **Insert**: Adds a node with price and id, O(log n) in a balanced BST (average case).
  - **Delete**: Removes a node with price and id, O(log n).
  - **Range Query**: Retrieves IDs with prices in [p1, p2] using in-order traversal, O(log n + k).
- **Algorithm Details**:
  - **Insert**: Traverses the tree based on price (and id for ties), inserting a new node at the appropriate leaf.
  - **Delete**: Finds the node, handles three cases (no children, one child, two children), and uses the minimum node in the right subtree as a successor for two-child cases.
  - **Range Query**: Performs an in-order traversal, pruning branches outside [p1, p2], collecting IDs within the range.
- **Usage**: Supports insert_new_stock, update_price (delete and insert), and price_range.

3. Defaultdict (self.price_to_ids)

- **Purpose**: Maps prices to sets of stock IDs to handle multiple stocks with the same price efficiently.
- **Implementation**: A defaultdict(set) where keys are prices and values are sets of stock IDs.
- **Operations**:
  - **Add/Remove**: Adding an ID (self.price_to_ids[p].add(x)) or removing an ID (self.price_to_ids[p].discard(x)) is O(1).
  - **Cleanup**: Deleting an empty set (del self.price_to_ids[old_price]) is O(1).
- **Usage**: Used in insert_new_stock and update_price to track IDs per price, ensuring

efficient BST updates.

4. Max Heap (self.volume_heap) with UUIDs (self.id_to_uuid)

- **Purpose**: Tracks the stock with the maximum trading volume.
- **Implementation**: A Python list (self.volume_heap) used as a max heap via heapq, storing tuples (-volume, x, uuid). self.id_to_uuid maps stock IDs to their latest UUID to invalidate outdated heap entries.
- **Operations**:
  - **Push**: Adding a new volume entry (heapq.heappush) is O(log n).
  - **Peek/Cleanup**: Checking the top entry and removing invalid entries (due to volume updates) is O(1) for peeking and O(log n) for popping invalid entries.
- **Algorithm Details**:
  - Each volume update generates a new UUID, pushing a new (-volume, x, uuid) tuple.
  - max_vol checks the heap's top entry, popping entries with outdated UUIDs or deleted stocks until a valid entry is found.
- **Usage**: Supports increase_volume (push) and max_vol (peek/cleanup).

Implementation in Source Code

The stock_tracker.py file implements the above data structures and algorithms as follows:

1. insert_new_stock(self, x, p) - O(log n)

- **Code**:

```
def insert_new_stock(self, x, p):
    if x in self.stocks:
        return
    self.stocks[x] = [p, 0]
    self.price_sorted.insert(p, x)
    self.price_to_ids[p].add(x)
    unique_id = str(uuid.uuid4())
    self.id_to_uuid[x] = unique_id
    heapq.heappush(self.volume_heap, (0, x, unique_id))
```

- **Steps**:
    - Checks if x exists in self.stocks (O(1)).
    - Adds [p, 0] to self.stocks (O(1)).
    - Inserts (p, x) into the BST (self.price_sorted.insert, O(log n)).
    - Adds x to self.price_to_ids[p] (O(1)).
    - Generates a UUID and pushes (0, x, uuid) to the max heap (O(log n)).
- **Time Guarantee**: Dominated by BST insert and heap push, both O(log n).

2. update_price(self, x, p) - O(log n)

- **Code**:
```
def update_price(self, x, p):
    if x not in self.stocks:
        return
    old_price, volume = self.stocks[x]
    self.stocks[x][0] = p
    self.price_to_ids[old_price].discard(x)
    if not self.price_to_ids[old_price]:
        del self.price_to_ids[old_price]
    self.price_sorted.delete(old_price, x)
    self.price_sorted.insert(p, x)
    self.price_to_ids[p].add(x)
```

- **Steps**:
    - Checks if x exists (O(1)).
    - Updates price in self.stocks (O(1)).
    - Removes x from self.price_to_ids[old_price] and cleans up if empty (O(1)).
    - Deletes (old_price, x) from BST (O(log n)).
    - Inserts (p, x) into BST (O(log n)).
    - Adds x to self.price_to_ids[p] (O(1)).
- **Time Guarantee**: Dominated by BST delete and insert, O(log n).

3. increase_volume(self, x, vinc) - O(log n)

- **Code**:
```
def increase_volume(self, x, vinc):
    if x not in self.stocks:
```

```
        return
    self.stocks[x][1] += vinc
    new_volume = self.stocks[x][1]
    unique_id = str(uuid.uuid4())
    self.id_to_uuid[x] = unique_id
    heapq.heappush(self.volume_heap, (-new_volume, x, unique_id))
```

- **Steps**:
  - Checks if x exists (O(1)).
  - Increments volume in self.stocks (O(1)).
  - Generates a new UUID and pushes (-new_volume, x, uuid) to the max heap (O(log n)).
- **Time Guarantee**: Dominated by heap push, O(log n).

4. lookup_by_id(self, x) - O(1)

- **Code**:def lookup_by_id(self, x):

```
    if x not in self.stocks:
        return None
    return self.stocks[x]
```

- **Steps**:
  - Checks if x exists in self.stocks (O(1)).
  - Returns self.stocks[x] or None (O(1)).
- **Time Guarantee**: Hash table operations are O(1) on average.

5. price_range(self, p1, p2) - O(log n + k)

- **Code**:def price_range(self, p1, p2):

```
    return self.price_sorted.range_query(p1, p2)
```

- **Steps**:
  - Calls BinarySearchTree.range_query, which:
    - Traverses the BST to the first node where price >= p1 (O(log n)).
    - Performs in-order traversal, collecting IDs where p1 <= price <= p2

(O(k)).

    - Prunes branches where price > p2 (reduces unnecessary traversal).

- **Time Guarantee**: O(log n) to reach the range, O(k) to collect k IDs.

6. max_vol(self) - O(1) retrieval, O(log n) cleanup

- **Code**:def max_vol(self):

```
while self.volume_heap:
    neg_vol, x, uuid = self.volume_heap[0]
    if x not in self.stocks or self.id_to_uuid.get(x) != uuid:
        heapq.heappop(self.volume_heap)
        continue
    return -neg_vol, x
return 0, None
```

- **Steps**:
    - Peeks at the heap's top entry (O(1)).
    - If the entry is invalid (stock deleted or UUID outdated), pops it (O(log n)) and repeats.
    - Returns the valid (-neg_vol, x) or (0, None) if empty.
- **Time Guarantee**: O(1) for valid top entry, O(log n) per pop for cleanup (amortized low due to infrequent updates).

Implementation Notes

- **BST Design**: The BST uses price as the primary key and id to break ties, ensuring unique nodes. The implementation avoids balancing (e.g., AVL or Red-Black Tree) for simplicity, assuming random insertions approximate balance (O(log n) average case).
- **Heap Cleanup**: The max heap uses lazy deletion (checking UUIDs in max_vol), reducing the need for immediate updates in increase_volume.
- **Hash Table Efficiency**: Python's dictionary ensures O(1) for lookup_by_id and other hash-based operations, critical for frequent lookups in testing.