MSCS-532

Partial Implementation of Data Structures for Dynamic Inventory Management

----

Sandesh Pokharel

University of Cumberlands

MSCS-532-A01: Algorithms and Data Structures

1. Introduction

In this phase of the project, we're implementing the core data structures and logic behind a **Dynamic Inventory Management System**. The goal here is to show off some key features, like how we manage and retrieve products using a **hash table**, **min-heap**, and **AVL tree**. These data structures help keep the system efficient, ensuring quick lookups and easy access to the lowest-priced items.

We've designed the system to be interactive via a **command-line interface (CLI)**. Users can add, update, delete products, retrieve the lowest-priced product, and query products by price range. There's also error handling built in to catch any mistakes and prevent crashes. Plus, to make testing easier, we added a few dummy products to the system automatically when it starts.

2. Partial Implementation Overview

Let's go over the core data structures and how they work in this system. Here's a quick rundown of the three main structures:

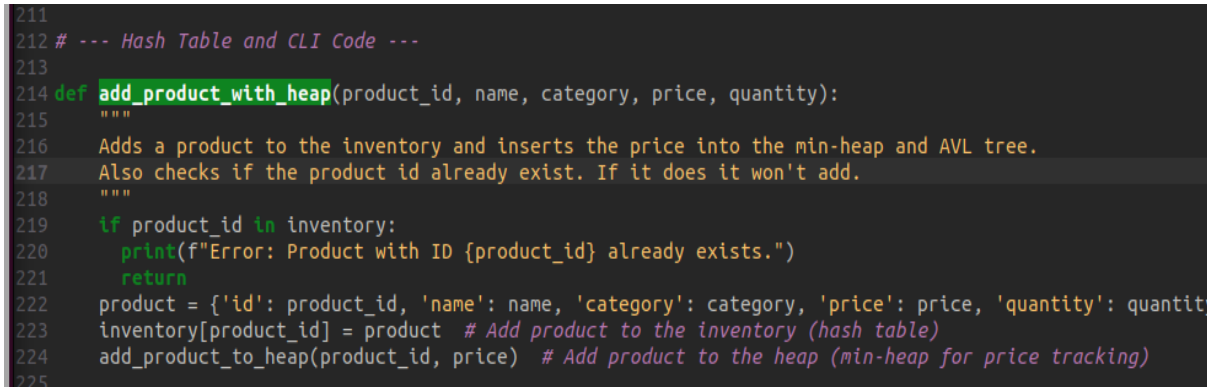
Hash Table (Dictionary): Used for fast lookups, adding, and deleting products. Each product is stored with a unique product_id, and since we're using a Python dictionary, operations on this are pretty much constant-time *O(1)* (Goodrich et al., 2014).

Min-Heap: This structure lets us quickly retrieve the product with the lowest price. We use Python's heapq library for this. When a new product is added, we push its price into the heap, and the heap ensures that the lowest price is always at the top (Cormen et al., 2009).

AVL Tree: This balanced binary search tree is used for querying products in a specific price range. The tree automatically balances itself after each insertion, so searching, inserting, and deleting all take logarithmic time $O(n \log n)$ (Mehta & Sahni, 2004).

Hash Table: Storing and Updating Products

We're using a hash table (a Python dictionary) to store each product. Below is the code snippet that handles adding products to the inventory. When adding a new product, we first check if the product_id already exists. If it does, an error is shown to prevent duplicate entries. Then, the product is added to the hash table for quick lookups, and its price is added to the heap for price-based retrieval.

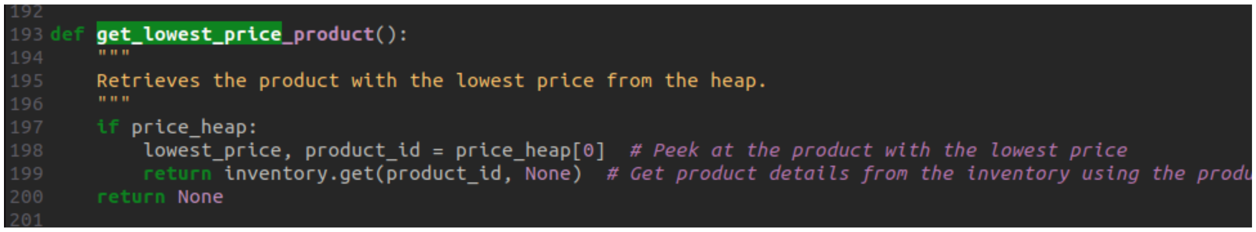


```
# --- Hash Table and CLI Code ---

def add_product_with_heap(product_id, name, category, price, quantity):
    """
    Adds a product to the inventory and inserts the price into the min-heap and AVL tree.
    Also checks if the product id already exist. If it does it won't add.
    """
    if product_id in inventory:
      print(f"Error: Product with ID {product_id} already exists.")
      return
    product = {'id': product_id, 'name': name, 'category': category, 'price': price, 'quantity': quantit
    inventory[product_id] = product  # Add product to the inventory (hash table)
    add_product_to_heap(product_id, price)  # Add product to the heap (min-heap for price tracking)
```

Min-Heap: Finding the Lowest Priced Product

The min-heap ensures that we can always quickly retrieve the product with the lowest price. The heap keeps prices in a structure where the smallest price is always at the root:

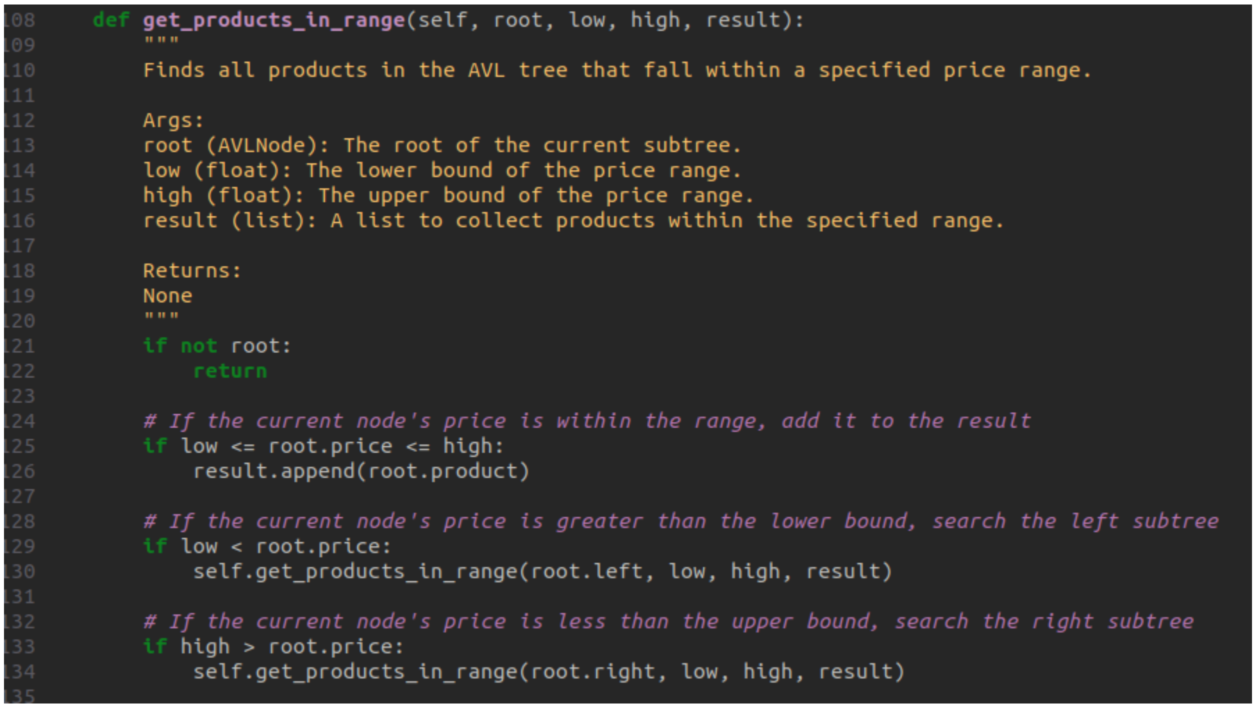
This function below simply grabs the lowest price from the heap, which is guaranteed to be at the top due to the min-heap property. Then, it looks up the product by its ID in the hash table and returns the product details.



```
def get_lowest_price_product():
    """
    Retrieves the product with the lowest price from the heap.
    """
    if price_heap:
        lowest_price, product_id = price_heap[0]  # Peek at the product with the lowest price
        return inventory.get(product_id, None)  # Get product details from the inventory using the produ
    return None
```

AVL Tree: Querying Products by Price Range

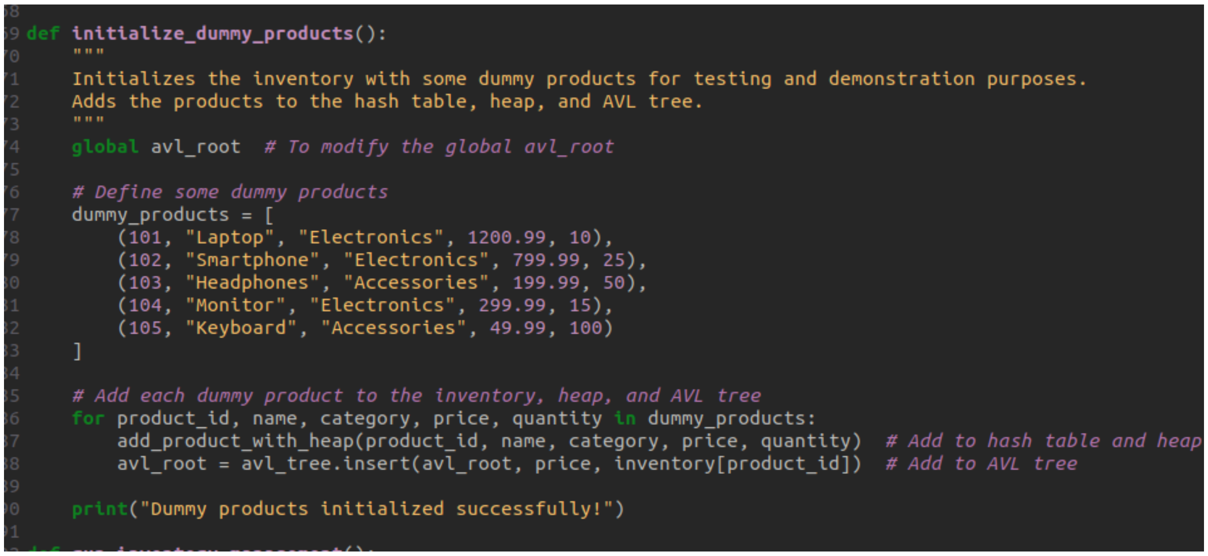
The AVL tree is used to find all products that fall within a certain price range. This is useful for e-commerce platforms where users might want to filter products by price. This function below searches through the AVL tree to find all products that have prices between low and high. It traverses the tree efficiently, only exploring subtrees that could potentially have prices in the desired range. The result is a list of products that meet the criteria.



```python
    def get_products_in_range(self, root, low, high, result):
        """
        Finds all products in the AVL tree that fall within a specified price range.

        Args:
        root (AVLNode): The root of the current subtree.
        low (float): The lower bound of the price range.
        high (float): The upper bound of the price range.
        result (list): A list to collect products within the specified range.

        Returns:
        None
        """
        if not root:
            return

        # If the current node's price is within the range, add it to the result
        if low <= root.price <= high:
            result.append(root.product)

        # If the current node's price is greater than the lower bound, search the left subtree
        if low < root.price:
            self.get_products_in_range(root.left, low, high, result)

        # If the current node's price is less than the upper bound, search the right subtree
        if high > root.price:
            self.get_products_in_range(root.right, low, high, result)
```

3. Demonstrating and testing

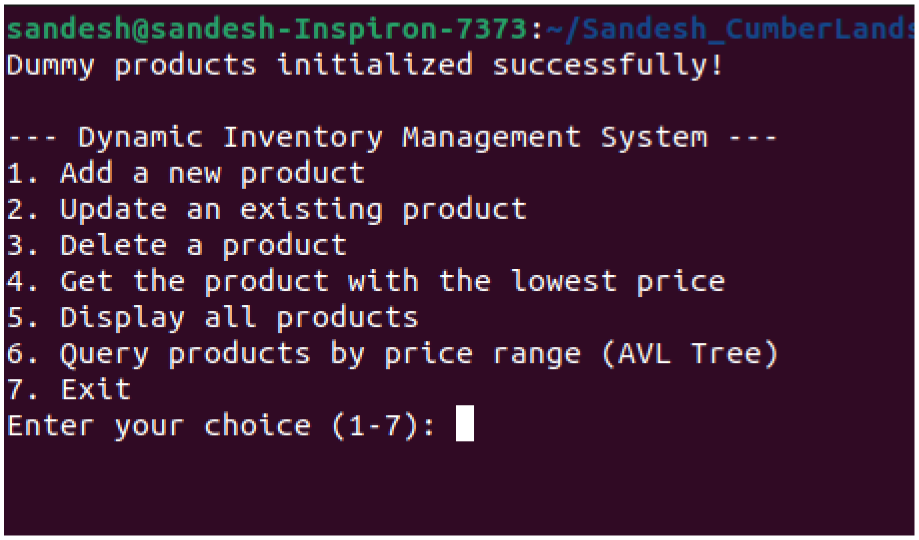
When the program starts, it preloads the system with some dummy products for testing purposes. This makes it easy to demonstrate the system without needing to manually add products every time. Here's how that part of the code looks:



```python
def initialize_dummy_products():
    """
    Initializes the inventory with some dummy products for testing and demonstration purposes.
    Adds the products to the hash table, heap, and AVL tree.
    """
    global avl_root  # To modify the global avl_root

    # Define some dummy products
    dummy_products = [
        (101, "Laptop", "Electronics", 1200.99, 10),
        (102, "Smartphone", "Electronics", 799.99, 25),
        (103, "Headphones", "Accessories", 199.99, 50),
        (104, "Monitor", "Electronics", 299.99, 15),
        (105, "Keyboard", "Accessories", 49.99, 100)
    ]

    # Add each dummy product to the inventory, heap, and AVL tree
    for product_id, name, category, price, quantity in dummy_products:
        add_product_with_heap(product_id, name, category, price, quantity)  # Add to hash table and heap
        avl_root = avl_tree.insert(avl_root, price, inventory[product_id])  # Add to AVL tree

    print("Dummy products initialized successfully!")
```

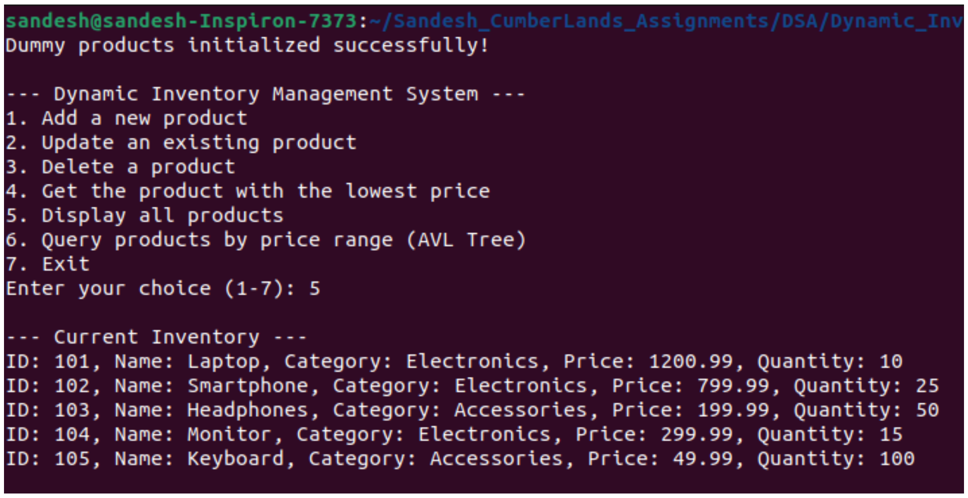
When users choose the option to display all products, they can see this list of pre-added products. This is particularly helpful for quickly testing the system's other features, like querying products by price or updating product details.
Here are some screenshots of features of program and how it looks when it loads:

At first:



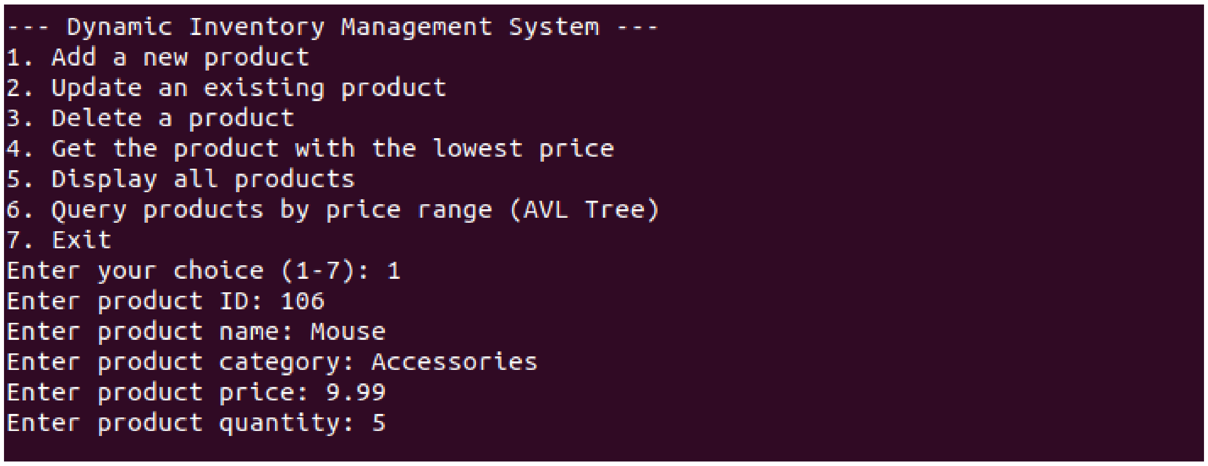
```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands
Dummy products initialized successfully!

--- Dynamic Inventory Management System ---
1. Add a new product
2. Update an existing product
3. Delete a product
4. Get the product with the lowest price
5. Display all products
6. Query products by price range (AVL Tree)
7. Exit
Enter your choice (1-7): 
```

Displaying all items:



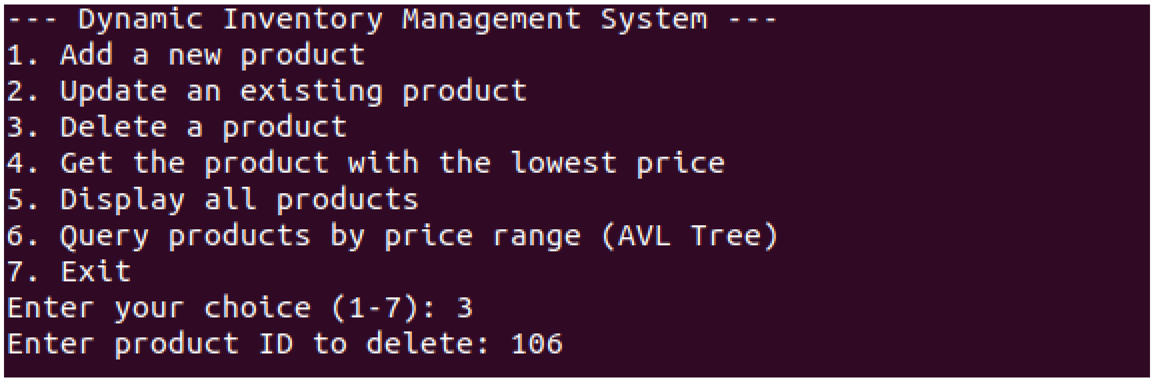
```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Dynamic_Inv
Dummy products initialized successfully!

--- Dynamic Inventory Management System ---
1. Add a new product
2. Update an existing product
3. Delete a product
4. Get the product with the lowest price
5. Display all products
6. Query products by price range (AVL Tree)
7. Exit
Enter your choice (1-7): 5

--- Current Inventory ---
ID: 101, Name: Laptop, Category: Electronics, Price: 1200.99, Quantity: 10
ID: 102, Name: Smartphone, Category: Electronics, Price: 799.99, Quantity: 25
ID: 103, Name: Headphones, Category: Accessories, Price: 199.99, Quantity: 50
ID: 104, Name: Monitor, Category: Electronics, Price: 299.99, Quantity: 15
ID: 105, Name: Keyboard, Category: Accessories, Price: 49.99, Quantity: 100
```

Adding product



```
--- Dynamic Inventory Management System ---
1. Add a new product
2. Update an existing product
3. Delete a product
4. Get the product with the lowest price
5. Display all products
6. Query products by price range (AVL Tree)
7. Exit
Enter your choice (1-7): 1
Enter product ID: 106
Enter product name: Mouse
Enter product category: Accessories
Enter product price: 9.99
Enter product quantity: 5
```

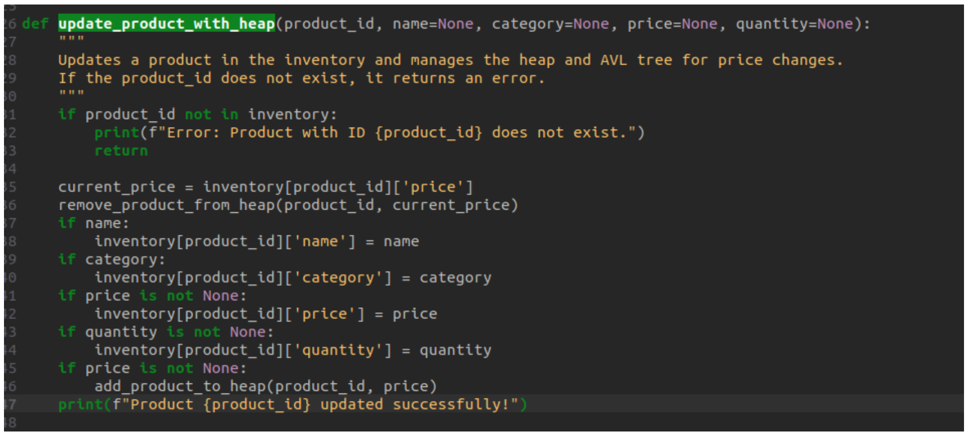
Deleting Product



```
--- Dynamic Inventory Management System ---
1. Add a new product
2. Update an existing product
3. Delete a product
4. Get the product with the lowest price
5. Display all products
6. Query products by price range (AVL Tree)
7. Exit
Enter your choice (1-7): 3
Enter product ID to delete: 106
```

4. Error Handling and Robustness

Error handling is a big part of this system. We've made sure that users can't crash the program by entering invalid inputs or performing illegal operations. For instance, when updating or deleting products, the system checks whether the product exists before proceeding. Similarly, it prevents adding products with duplicate IDs.
Here's an example of error handling when updating a product: If the product_id doesn't exist, it returns an error, keeping everything safe and consistent.



```python
def update_product_with_heap(product_id, name=None, category=None, price=None, quantity=None):
    """
    Updates a product in the inventory and manages the heap and AVL tree for price changes.
    If the product_id does not exist, it returns an error.
    """
    if product_id not in inventory:
        print(f"Error: Product with ID {product_id} does not exist.")
        return

    current_price = inventory[product_id]['price']
    remove_product_from_heap(product_id, current_price)
    if name:
        inventory[product_id]['name'] = name
    if category:
        inventory[product_id]['category'] = category
    if price is not None:
        inventory[product_id]['price'] = price
    if quantity is not None:
        inventory[product_id]['quantity'] = quantity
    if price is not None:
        add_product_to_heap(product_id, price)
    print(f"Product {product_id} updated successfully!")
```

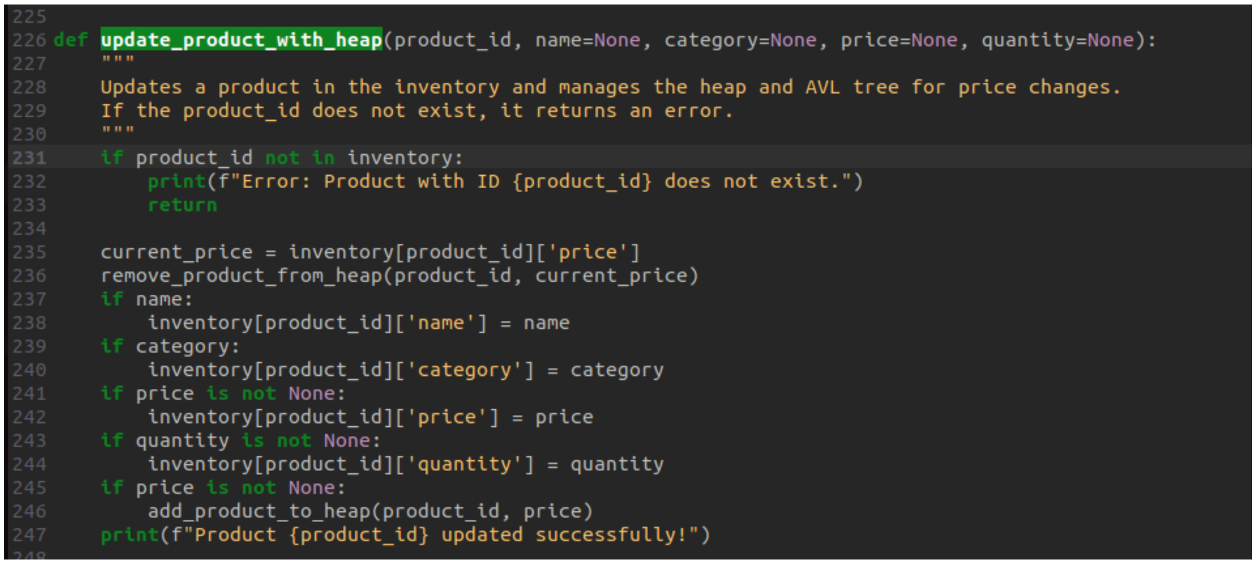
5. Implementation Challenges and solutions

As with any complex system, we faced a few challenges while implementing this dynamic inventory management system. Below are some of the key challenges and how we tackled them.

Maintaining Consistency Across Multiple Data Structures

Challenge: One of the biggest challenges was ensuring that the same product information was consistently updated across all three data structures: the hash table, the min-heap, and the AVL tree. For example, if the price of a product was updated, the change had to

be reflected in the hash table (for fast lookups), the heap (for retrieving the lowest-priced product), and the AVL tree (for price range queries).
Solution: To address this, we made sure to first remove the old price from the heap and AVL tree before adding the new price. Here's a snippet that shows how we handled this when updating a product's price:



```python
def update_product_with_heap(product_id, name=None, category=None, price=None, quantity=None):
    """
    Updates a product in the inventory and manages the heap and AVL tree for price changes.
    If the product_id does not exist, it returns an error.
    """
    if product_id not in inventory:
        print(f"Error: Product with ID {product_id} does not exist.")
        return

    current_price = inventory[product_id]['price']
    remove_product_from_heap(product_id, current_price)
    if name:
        inventory[product_id]['name'] = name
    if category:
        inventory[product_id]['category'] = category
    if price is not None:
        inventory[product_id]['price'] = price
    if quantity is not None:
        inventory[product_id]['quantity'] = quantity
    if price is not None:
        add_product_to_heap(product_id, price)
    print(f"Product {product_id} updated successfully!")
```

Explanation: By first removing the old price from the heap, we avoid inconsistencies. After updating the product details, we reinsert the updated price back into the heap and AVL tree. This ensures that all data structures remain synchronized.

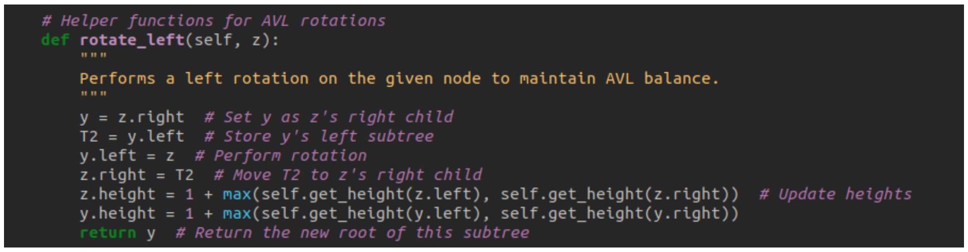
Handling Invalid Inputs Gracefully

Challenge: Users may sometimes enter invalid inputs, such as non-numeric values for product prices or trying to update a non-existing product. Handling these cases was essential to prevent crashes or data corruption.
Solution: We incorporated robust error handling using try-except blocks to catch invalid inputs and provide clear error messages to users. This allows the program to handle mistakes gracefully and ensures a smooth user experience. For instance, when updating a product, we check whether the product_id exists in the inventory before proceeding with the update.
Explanation: If the user enters an invalid data type (e.g., a string instead of a number), the program catches the error and prompts the user to enter valid data, preventing crashes.

Balancing the AVL Tree Efficiently

Challenge: The AVL tree needs to be balanced after every insertion to maintain its efficiency for search and range queries. Implementing the necessary rotations and ensuring the tree remains balanced without errors was tricky, especially when handling edge cases like inserting nodes that create unbalanced subtrees.

Solution: We implemented the rotation logic carefully, testing various cases to ensure the tree rebalances correctly after every insertion. Here's an example of how we implemented the left rotation in the AVL tree:



```
    # Helper functions for AVL rotations
    def rotate_left(self, z):
        """
        Performs a left rotation on the given node to maintain AVL balance.
        """
        y = z.right  # Set y as z's right child
        T2 = y.left  # Store y's left subtree
        y.left = z  # Perform rotation
        z.right = T2  # Move T2 to z's right child
        z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))  # Update heights
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y  # Return the new root of this subtree
```

Explanation: This rotation function ensures that when the right subtree is heavier than the left (a "Right-Right" imbalance), we can perform a left rotation to restore balance. The AVL tree uses similar logic for right rotations and other complex cases like "Left-Right" and "Right-Left" imbalances.

Optimizing for Performance

Challenge: Balancing between read and write operations across different data structures presented a performance trade-off. While the hash table provides constant-time lookups, the AVL tree and heap require logarithmic time to insert and delete, which can slow down the system if not optimized properly.

Solution: To manage this, we limited complex operations (like balancing the AVL tree and reordering the heap) to essential updates only. By keeping operations modular, we ensured that performance remained acceptable, even as the dataset grows.

6. Next steps

So far, we've implemented the core features of the system. Here's what comes next:

- Scaling: Right now, the system handles a moderate number of products well, but as the inventory grows, we might need to consider optimization techniques for larger datasets.
- Advanced Queries: We could implement more complex query options, like searching by product category or combining multiple filters.
- Improved User Interface: A graphical user interface (GUI) could make the system more user-friendly, especially for non-technical users.
- More better error handling. Like, telling user right away if they entered something wrong. Example, for now in case of updating system it asks for product_id and all other information. And at last if will not update if the product_id does not exist. We can make that catch at the prompt next to it and make the user to enter it right instantly.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Python. Wiley.
3. Mehta, D. P., & Sahni, S. (2004). Handbook of Data Structures and Applications. Chapman and Hall/CRC.