MSCS532_PROJECT PHASE 2

DATE: 17TH NOVEMBER 2024

Shafan Nazeer Ahmed
005030047

Hash Table

To store product information in the inventory system, a hash table was implemented using Python's built-in dictionary. The hash table has an average time complexity of O(1) for insertions, updates and lookups. It is an optimal choice for managing product information, including name, quantity, price, and category, as this design guarantees that products can be promptly accessed and updated.

Key operations that have been implemented for the hash table include:-

i. Insertion - Adding a new product to the inventory.
ii. Update - Revises the quantity or price of an existing product.
iii. Inventory Deletion - Removing a product from the inventory.
iv. Product Search - Utilizes the product's ID to locate the product.

All of these operations enable the inventory system to effectively manage the dynamic nature of inventory, which necessitates frequent additions, updates and lookups.

Binary Search Tree

In order to manage and query products according to their price the Binary Search Tree (BST) was implemented. Ensuring that products can be queried by price range, a common operation in inventory systems that deal with promotions and price-sensitive decision-making, the BST allows for efficient O(log n) time complexity for insertion and search operations.

Key operations that have been implemented for the Binary Search Table:-

- Insertion - Product is inserted into tree.
- Search by price range - Retrieves products that fall within a particular price range.

The foundation for further optimization is established by this partial implementation, which includes the ability to manage larger datasets and intricate querying options.

Test Cases

The test cases verify the implementation like missing products and invalid updates.

1. Add a Product

   Testing of {Adding a product to the inventory}.

   Input - add_product(1, "Mobile", 10, 1200, "Electronics")

   Output - Product 1 added.

2. Update a Product

   Testing of {Updating a product's quantity and price}.

   Input - update_product(1, quantity=15)

   - Output - Product 1 updated.

3. Remove a Product

   Testing of {Removing a product from the inventory}.

      Input - remove_product(2)

      Output - Product 2 removed.

4. Search for a Product by ID

   Testing of {Searching for a product by its ID}.

      Input - search_product(1)

      Output - {'name': 'Mobile', 'quantity': 15, 'price': 1200, 'category': 'Electronics'}

5. Query Products by Price Range

   Testing of {Retrieving products within a price range using the BST}.

      Input - search_price_range(500, 1500)

      Output - ['Mobile']

Challenges

The system verifies whether the ID already exists in the hash table when attempting to add a product with a duplicate ID.

To prevent data overwriting, the user is informed that the product already exists if the ID exists.

The performance can be degraded from O(log n) to O(n) by inserting products in either ascending or descending order of price.

Tree balancing is currently being postponed for future optimization phases.
A self-balancing tree, such as an AVL or Red Black tree, may be implemented at a later time to ensure optimal performance.

The system incorporates checks to prevent the updating or deletion of nonexistent products.

In order to prevent system crashes, appropriate error messages are displayed when a product is not located.

Next Steps

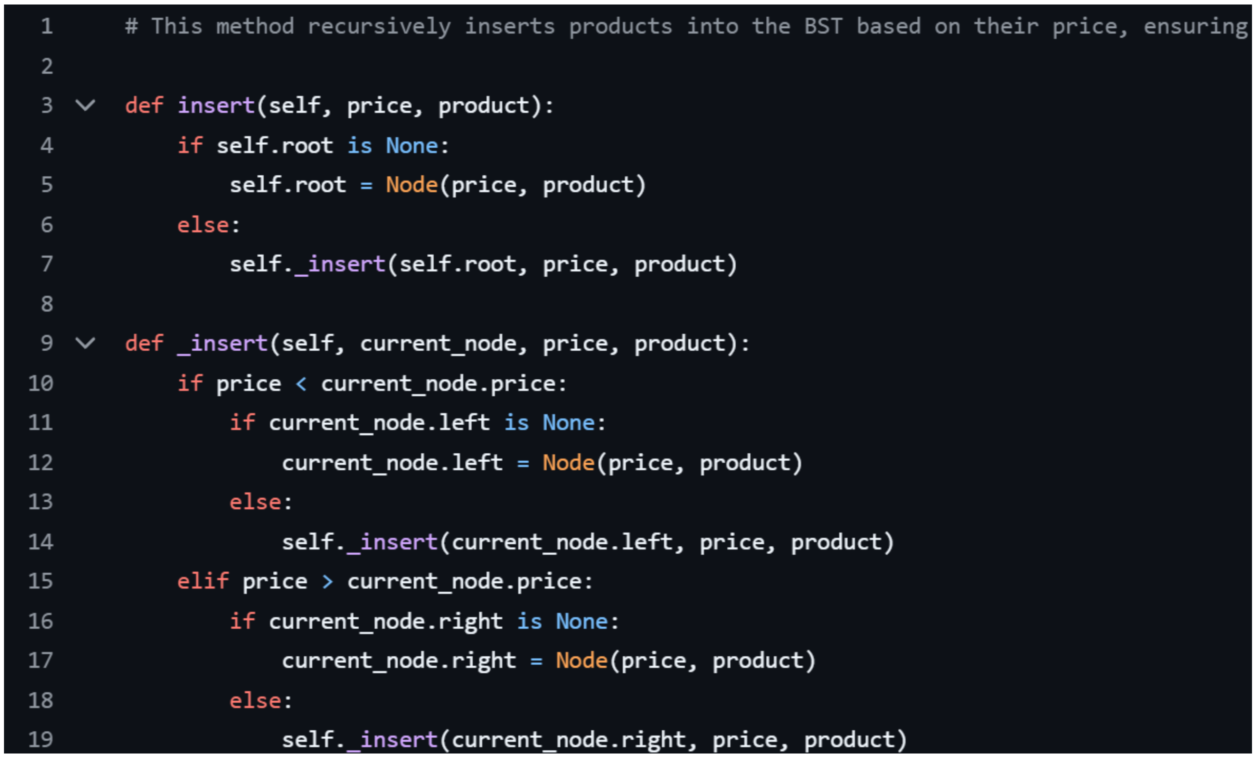
1. Implement tree balancing.
The current Binary Search Tree (BST) implementation may become unbalanced due to sequential insertions of increasing/ decreasing prices. The next step is to implement a self-balancing tree (such as an AVL tree or a Red-Black tree), which will ensure that all operations—insertions, deletions, and searches—are completed in O(log n) time complexity, even in the worst-case scenarios.

2. Extend Data Structure Functionality.
The current implementation focuses on basic operations; however, the inventory system could benefit from more advanced features.

Handling Product categories use a more complex structure to manage hierarchical categories (for example, subcategories within categories).
Developing more efficient search algorithms, particularly for large datasets.

3. Memory Management.
As the inventory grows, memory management becomes critical. To optimize memory usage, techniques such as lazy deletion (marking nodes for deletion but not removing them right away) and caching frequently accessed items could be used.

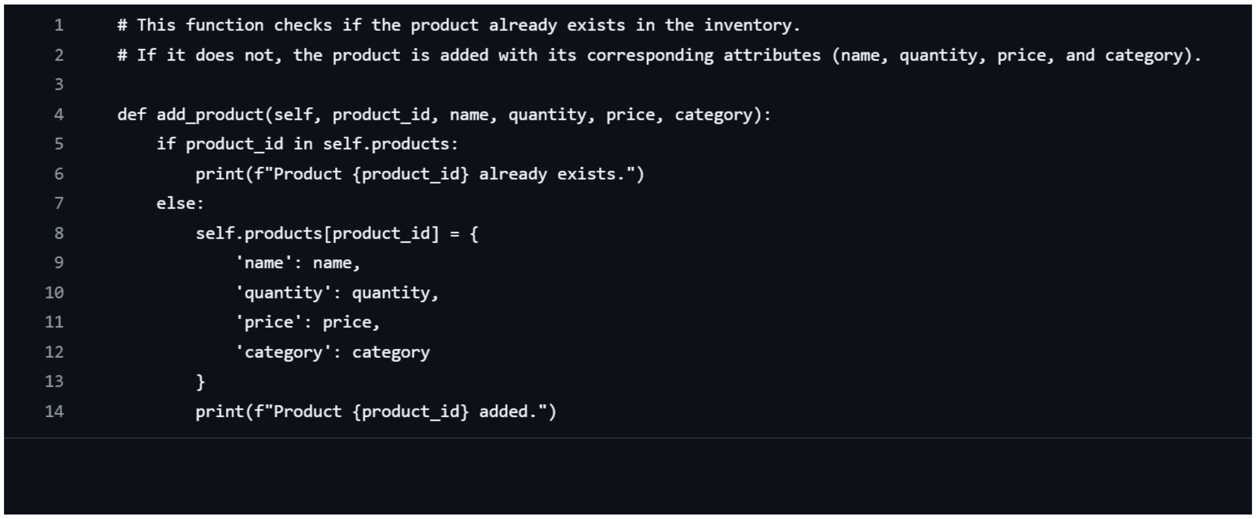
Code Snippets & Github Repository



```python
# This method recursively inserts products into the BST based on their price, ensuring

def insert(self, price, product):
    if self.root is None:
        self.root = Node(price, product)
    else:
        self._insert(self.root, price, product)

def _insert(self, current_node, price, product):
    if price < current_node.price:
        if current_node.left is None:
            current_node.left = Node(price, product)
        else:
            self._insert(current_node.left, price, product)
    elif price > current_node.price:
        if current_node.right is None:
            current_node.right = Node(price, product)
        else:
            self._insert(current_node.right, price, product)
```



```python
# This function checks if the product already exists in the inventory.
# If it does not, the product is added with its corresponding attributes (name, quantity, price, and category).

def add_product(self, product_id, name, quantity, price, category):
    if product_id in self.products:
        print(f"Product {product_id} already exists.")
    else:
        self.products[product_id] = {
            'name': name,
            'quantity': quantity,
            'price': price,
            'category': category
        }
        print(f"Product {product_id} added.")
```

https://github.com/sahmed30047/MSCS532_Phase-2.git

References

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2011). *Data Structures and Algorithms in Python.* Wiley.

Knuth, D. E. (1998). *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley.