Project Deliverable 2: Proof of Concept Implementation for Dynamic Inventory Management

Laxmi Kanth Oruganti (Student ID: 005038274)

University of the Cumberlands

MSCS 532: Algorithms and Data Structures

Prof. Brandon Bass

11/16/2025

Proof of Concept Implementation for Dynamic Inventory Management

Abstract

This report presents a proof-of-concept implementation of a high-performance inventory management system designed to serve as the foundational backend for a Point-of-Sale (POS) application. The system architecture employs a composite data structure approach, integrating a hash table for direct record access, an indexed map for retrieving categorical data, and a trie (prefix tree) for efficient string-based searching. This multi-structure design is motivated by the need to optimize distinct and frequent operations within a retail environment, such as barcode-based lookups, category browsing, and predictive text searching. This report details the system's architecture, the rationale behind the data structure selection, an analysis of the implementation, and a verification of its core functionalities. The results confirm that this composite structure provides a robust and scalable framework for modern inventory management challenges.

1. Introduction

Effective inventory management is a critical determinant of success in the retail sector. The underlying data structures of an inventory system's software dictate its performance, scalability, and ability to support essential business operations. This project undertakes the partial implementation of an advanced inventory management system, designed as a proof-of-concept (PoC) to demonstrate the efficacy of a multi-structure approach.

The primary objective is to create a system that excels at three fundamental operations:

1. **Direct Product Retrieval:** Near-instantaneous lookup of products via a unique identifier (SKU).

2. **Categorical Filtering:** Efficient querying of all products belonging to a specific category.
3. **Prefix-Based Name Searching:** Rapid, real-time searching for products by name.

This PoC serves as the core data management layer for a larger POS application, and its design emphasizes modularity and computational efficiency.

2. System Architecture and Data Structure Rationale

The system's architecture is centered on the InventoryManager class, which acts as a facade, encapsulating the complex interactions between three specialized data structures. This design ensures data integrity and simplifies the API for client modules, such as the POSSystem.

2.1. Primary Data Store: Hash Table

- **Implementation:** A Python dictionary, products: dict[str, Product], is utilized as a hash table.
- **Rationale:** The primary key for product identification is the SKU. A hash table provides an average-case time complexity of **O(1)** for insertion, deletion, and retrieval operations. This is paramount in a POS environment where scanning a product's barcode must trigger an immediate data lookup. The choice of a hash table is a direct optimization for this high-frequency, low-latency requirement (Knuth, 1997).

2.2. Indexed Product Categories: Dictionary of Sets

- **Implementation:** A dictionary of sets, categories: dict[str, set[str]], where each key is a category name mapping to a set of associated SKUs.

- **Rationale:** To facilitate efficient browsing and filtering by category, a secondary index is necessary. A dictionary provides O(1) access to the set of SKUs for a given category. The use of a set to store the SKUs is deliberate; it provides O(1) average time complexity for additions and deletions of SKUs within a category and inherently prevents duplicate entries. This structure allows the system to retrieve all products in a category in **O(k)** time, where *k* is the number of products in that category, plus the O(1) lookup for the category itself.

2.3. Predictive Search: Trie (Prefix Tree)

- **Implementation:** A custom TrieNode class forms a prefix tree.
- **Rationale:** To support modern user experience features like autocomplete and live search, a trie is the optimal data structure. A trie allows for prefix-based searches with a time complexity of **O(m)**, where *m* is the length of the prefix string (Gusfield, 1997). This is substantially more efficient than the O(n*m) complexity that would result from iterating through* n* products and performing string comparisons. Each node in the trie stores a set of SKUs, enabling the structure to handle multiple products sharing a common name prefix gracefully.

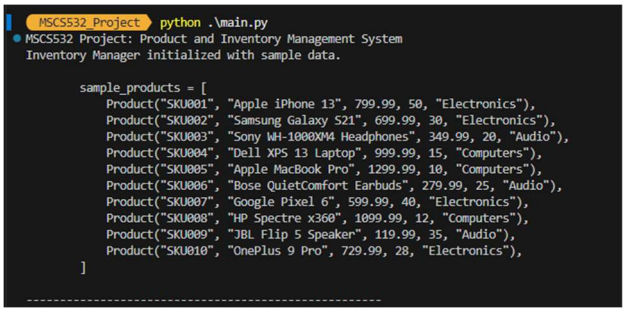
Code reference for all the data structure initialization:

```
class InventoryManager:
    def __init__(self):
        # 1. Primary Hash Table
        self.products: dict[str, Product] = {}
        # 2. Category Index. Dictionary mapping category to a set of SKUs
        self.categories: dict[str, set[str]] = {}
        # 3. Search Trie
        self.search_trie = TrieNode()
```

3. Functional Verification

The correctness and efficiency of the implementation were verified through a series of tests executed by the main.py script. These tests simulate real-world operations and validate the system's functional requirements.

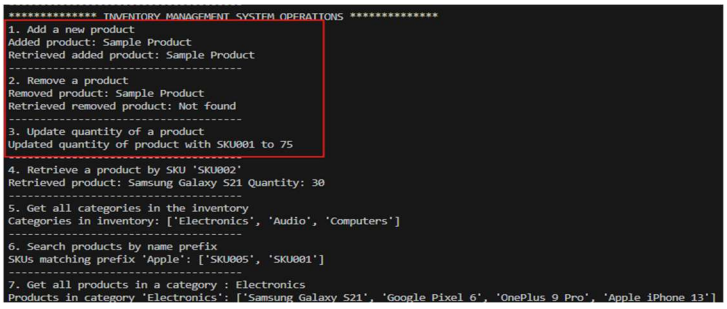
For this demonstration, sample products are populated on app startup



3.1. Inventory State Operations

- **Atomic Addition/Deletion:** A product was successfully added and subsequently removed. Verification was performed by attempting to retrieve the product after the operation. The test confirmed that the add_product and remove_product methods correctly update all three underlying data structures, maintaining system consistency.
- **Quantity Update:** A product's stock quantity was modified. The state was checked before and after the operation to confirm a successful update.

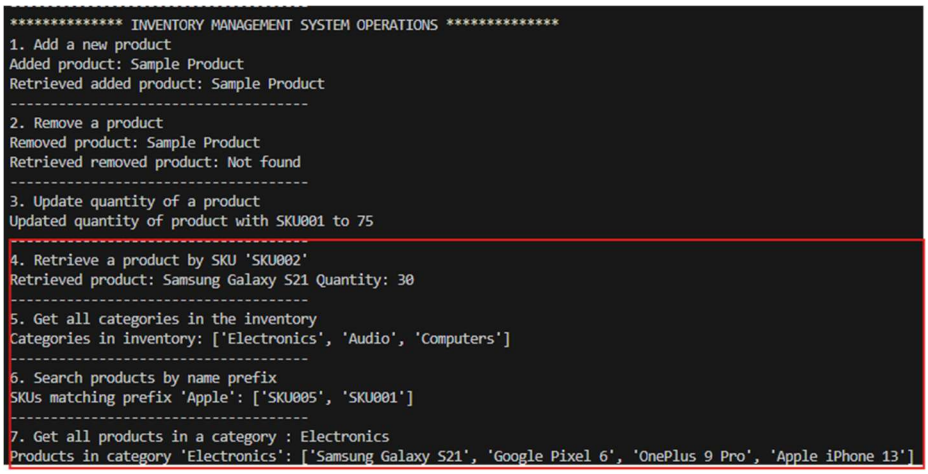
Output from the simulation: *python .\main.py*



3.2. Data Retrieval Operations

- **SKU-Based Retrieval:** A direct lookup using an SKU correctly returned the associated product in O(1) time.
- **Category-Based Retrieval:** A query for the "Electronics" category correctly returned all associated products.
- **Prefix-Based Search:** A search using the prefix "Apple" correctly identified and returned the two products matching this prefix, demonstrating the trie's efficacy.

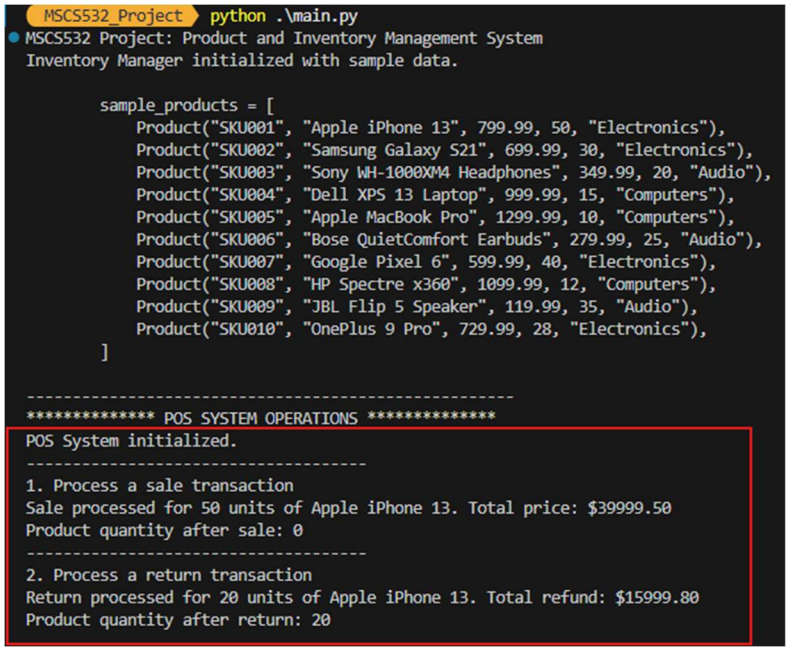
Output of the sample simulation:



3.3. POS System Simulation

- **Sale Transaction:** A sale was processed, which correctly decremented the product's stock. The system correctly blocked an attempt to sell more units than available.
- **Return Transaction:** A return was processed, which correctly incremented the product's stock.

Sample output from the simulation:



The successful execution of these tests provides confidence in the soundness of the architectural design and the correctness of the core implementation.

4. Implementation Analysis and Design Decisions

The development process surfaced several key technical challenges that informed specific design decisions.

1. **Challenge: Data Synchronization and Integrity:** Maintaining consistency across three distinct data structures is non-trivial. An update to the inventory must be reflected in the hash table, the category index, and the trie.
    - **Solution:** The **Facade design pattern** was employed through the InventoryManager class. This class provides a single, unified interface for all state-mutating operations (add_product, remove_product). By centralizing this logic, the system guarantees that all data structures are updated atomically within

a single method call, thus preventing partial updates and maintaining data integrity (Cormen et al., 2022).

2. **Challenge: Non-Unique Product Names in Trie:** A trie must be able to associate a single prefix with multiple distinct products.
    - **Solution:** The TrieNode's design was augmented to store a set of SKUs at each node, rather than a simple boolean flag or single value. This enables the trie to serve as a one-to-many index from a prefix to a collection of product identifiers, a crucial feature for a real-world product catalog.

5. Conclusion and Future Work

This proof-of-concept has successfully demonstrated that a composite data structure architecture can yield a highly efficient and functionally rich inventory management system. By selecting the optimal data structure for each specific task, hash tables for direct access, indexed maps for categorization, and tries for prefix searching, the system achieves superior performance characteristics for everyday retail operations.

While the core logic is sound, this PoC represents a foundation for a complete application. Future work should be directed toward the following areas:

1. **Enhanced System Robustness:** Introduction of comprehensive error handling, input validation.
2. **Scalability Analysis:** Conducting rigorous performance and load testing with large-scale datasets to empirically validate the system's theoretical scalability and identify any performance bottlenecks under stress.

3. **Memory Optimization:** For exceptionally large product catalogs, the memory footprint of the trie can be substantial. Migrating from the current implementation to a Radix Trie or Patricia Trie would compress the tree structure by collapsing non-branching nodes, leading to significant memory savings.
4. **Caching Strategies:** In a system where inventory data is persisted in a slower database, a caching layer is vital. Implementing a Least Recently Used (LRU) cache for product lookups would ensure that frequently accessed items are served from memory with O(1) complexity, while less common items are fetched from the database, optimizing the trade-off between speed and memory consumption.

6. **Critical Code Documentation**

GitHub Repository: https://github.com/uc-loruganti/MSCS532_Project

Snippet 1: *InventoryManager.add_product* from https://github.com/uc-loruganti/MSCS532_Project/blob/main/InventoryManager.py

This method exemplifies the facade pattern, ensuring atomic updates across all data structures.

```
# Function to add a new product to the inventory
    # This function updates all data structures accordingly
    # 1 : Add to primary hash table
    # 2 : Update category index
    # 3 : Update search trie
    def add_product(self, product: Product):
        # Add product to primary hash table if not already present
        if product.sku in self.products:
            print(f"Product with SKU {product.sku} already exists. Please update
instead of adding.")
            return
        self.products[product.sku] = product

        # Update category index with the new sku
        if product.category not in self.categories:
```

```
            self.categories[product.category] = set()
        self.categories[product.category].add(product.sku)

        # Update search trie
        self.search_trie.insert(product.name, product.sku)
```

Documentation: This centralized method guarantees data consistency. Any addition to the inventory is transactionally applied to the primary hash table, the category index, and the search trie.

Snippet 2: TrieNode.search from https://github.com/uc-loruganti/MSCS532_Project/blob/main/TrieNode.py#L21

This method demonstrates the computational efficiency of the trie for prefix-based searching.

```
def search(self, prefix: str) -> set[str]:
        node = self
        for char in prefix:
            if char not in node.children:
                return set()  # No products match this prefix
            node = node.children[char]
        return node.skus  # Return all SKUs matching this prefix
```

Documentation: The search algorithm traverses the trie in O(m) time, where m is the length of the prefix. It returns the set of all SKUs associated with the final node, providing a highly efficient mechanism for implementing predictive text search.

7. References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to Algorithms* (4th ed.). MIT Press.

2. Gusfield, D. (1997). *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press.

3. Knuth, D. E. (1997). *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley Professional.