# DYNAMIC INVENTORY MANAGEMENT SYSTEM

Phase 3

AUGUST 3, 2025

BISHNU SHARMA
ID: 005036007

## Optimization, Scaling, and Final Evaluation of a Dynamic Inventory Management System

### Introduction

This final phase of the project focuses on optimizing the dynamic inventory management system initially developed in Phases 1 and 2. The goal is to scale the implementation to handle large datasets while improving performance, memory usage, and code quality. Enhancements include replacing the basic binary search tree (BST) with a self-balancing AVL tree for price sorting, simulating larger datasets to stress test the system, and refining operations for robustness. These improvements aim to validate the system's scalability and readiness for real-world inventory tasks.

### Optimization Techniques

The primary optimization involved replacing the unbalanced BST with an AVL tree to maintain sorted product listings efficiently. The AVL tree maintains O(log n) time complexity for insertions and lookups, which significantly improved performance over the original implementation (Cormen et al., 2009). Additionally, heap operations were optimized by refining the Product class’s comparison logic to prioritize stock-based ordering. The overall design was modularized to reduce redundancy and increase maintainability, aligning with software engineering principles outlined in Goodrich et al. (2014).

Other optimizations included code cleanup, better exception handling, and reducing unnecessary data duplication across modules. This helped lower memory overhead, particularly during batch processing of large datasets.

## Scaling Strategy

To test scalability, a script was written to generate over 1,000 random product entries with varying quantities and prices. This simulated a realistic inventory environment and revealed how each data structure handled increasing loads. The heap correctly filtered low-stock items, and the AVL tree maintained accurate price-based order even after numerous insertions. The test was repeated with larger batch sizes to assess memory behavior and time complexity trends. These experiments confirmed that the improved structures could efficiently handle scaling without significant performance drops.

## Testing and Validation

Two test scenarios were used. First, a small dataset with familiar items such as "Laptop" and "Mouse" was used to validate basic functionality, including product insertion, retrieval, and sorting. This test confirmed correctness of core features.

Second, a large dataset of 1,000+ products validated the system's scalability. Heap operations returned the lowest-stock items instantly, and AVL in-order traversal preserved price order. As shown in *Figure 1*, the output from basic operations verified the correctness of heap and tree behavior. *Figure 2* demonstrates the large dataset output with no observable lag, proving that the structures remain effective under stress.

**Figure 1:** ***Output of Basic Product Operations Including Low-Stock Retrieval and Price Sorting***

```
Low-stock items:
Mouse - 2 left
Laptop - 5 left

Price-sorted products:
Mouse - $20
Keyboard - $50
Laptop - $1000
```

**Figure 2:** ***Output Demonstrating Heap-Based Low Stock Retrieval and AVL Tree Price Sorting with a Large Dataset***

```
10 Low Stock Items:
Item333 - Qty: 0
Item459 - Qty: 0
Item318 - Qty: 0
Item93 - Qty: 0
Item861 - Qty: 0
Item242 - Qty: 0
Item510 - Qty: 0
Item848 - Qty: 0
Item843 - Qty: 0
Item280 - Qty: 1

First 10 Products by Price:
Item254 - Price: $5.4
Item26 - Price: $5.72
Item690 - Price: $6.29
Item438 - Price: $7.05
Item229 - Price: $8.36
Item549 - Price: $8.43
Item36 - Price: $9.71
Item589 - Price: $9.81
Item561 - Price: $10.44
Item725 - Price: $10.58
```

**Performance Analysis**

Replacing the unbalanced BST with an AVL tree showed significant gains in insert and traversal efficiency. With the old BST, inserting 1,000+ items showed signs of imbalance in output order and time delay. The AVL tree maintained balance, ensuring quick searches and sorted listings (Knuth, 1998).

Memory usage remained stable even as the dataset grew due to Python’s efficient memory model and the decision to avoid duplicating product instances across structures. The heap's O(n log k) performance for retrieving k lowest items remained consistent throughout tests. Overall, optimized structures resulted in lower CPU usage, reduced execution time, and more reliable outputs.

## Final Evaluation

This phase demonstrated a successful transition from proof-of-concept to a more scalable and efficient inventory system. By introducing an AVL tree and validating through stress testing, the system proved its readiness for real-world applications. While the current system supports insertion, low-stock filtering, and price-based sorting, future improvements could include persistent storage, search enhancements, deletion features, and user interfaces.

The progress made across all three phases reflects the importance of good data structure choices in building performant applications. With continued development, this inventory management tool could be deployed in real retail or warehouse settings.

## GitHub

The GitHub link is provided below:

https://github.com/iambissnuu/MSCS532_InventotySystem

**References**

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). *Data structures and algorithms in Python*. Wiley.

Knuth, D. E. (1998). *The art of computer programming, volume 3: Sorting and searching* (2nd ed.). Addison-Wesley.