MSCS-532-Assignment6-Elementary Data Structures

Sandesh Pokharel

University of Cumberlands

MSCS-532-A01: Algorithms and Data Structures

Introduction

In this phase of our dynamic inventory management system, we're stepping things up by optimizing the performance and scalability of our initial implementation. We're going to dig into the current data structures, like the hash table, min-heap, and AVL tree, to identify any bottlenecks and streamline them for better efficiency. The goal here is to make the system robust enough to handle larger datasets without compromising speed or memory. We'll be running stress tests to see how the system behaves under pressure, and by the end, we'll compare our improvements to the initial version and evaluate the trade-offs we had to make along the way.

Optimization

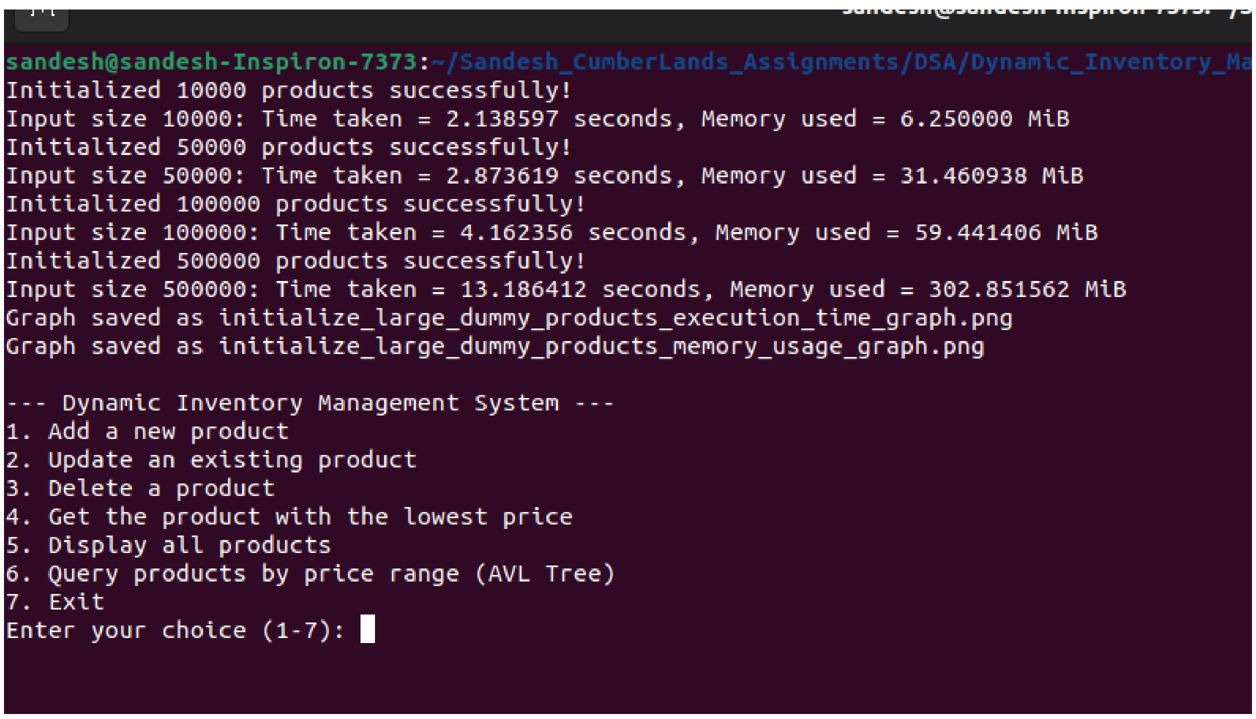
Current overview of Efficiency:

Operation	Current Data Structure	Current Time Complexity	Potential Alternatives
Add/Update/Delete Product	Hash Table	O(1)	None (Hash Table is optimal)
Search by ID	Hash Table	O(1)	None
Query by Price Range	AVL Tree	O(log n) + O(k)	Range Tree, Segment Tree
Find Lowest-Priced Product	Min-Heap	O(1) (find) / O(log n) (insert)	Double-ended Priority Queue

- For **adding, updating, and deleting products**, as well as **searching by ID**, we're already using a **hash table**, which gives us **O(1)** time complexity for these operations. Honestly, it's about as efficient as it gets for those tasks, so we won't be doing any

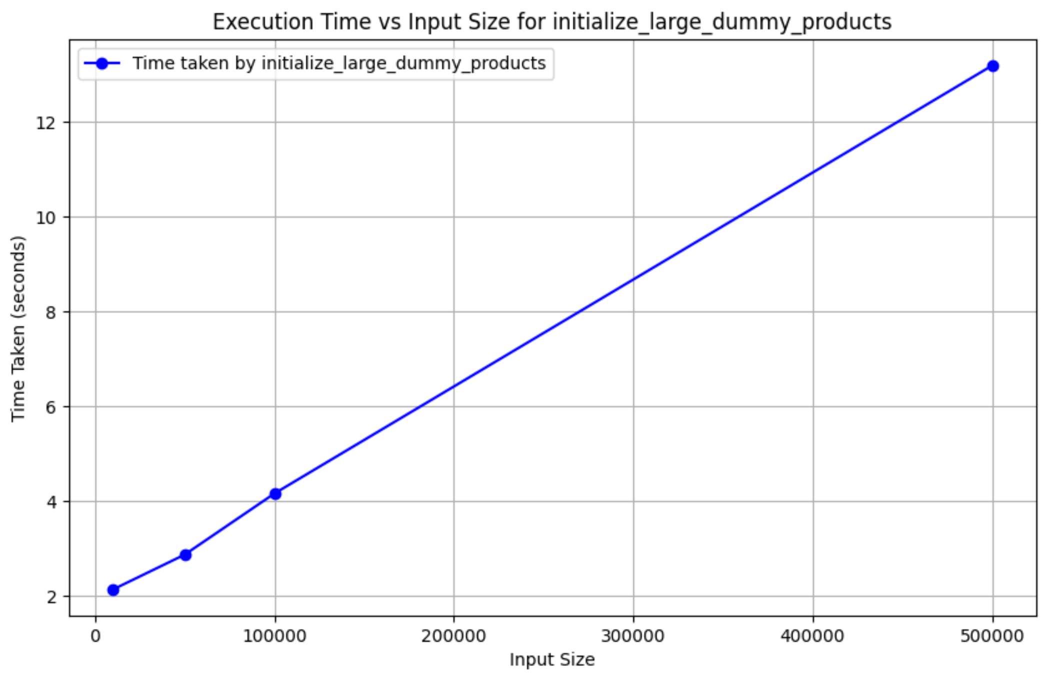
further optimizations there. The same goes for **searching by ID**—hash tables excel in that area with constant-time lookups. Since the current implementation is already optimal, there's no need to dive into any advanced techniques for those operations. **However,** even though I am not thinking to modify the data structures I have used, I am thinking of modifying the algorithm a little bit.
Below is the implementation of one of these operations (adding product) and its efficiency in larger datasets. It looks linear as expected. In the output below, we see time increased time for small input sizes, that is due to overhead of memory tracking function we have used. Without using those functions, the number would be in fractions of seconds. The **memory_profiler** library works by repeatedly querying the operating system to check memory usage. This involves multiple system calls, which add overhead to the program's execution. For smaller input sizes, this overhead becomes more noticeable since the actual execution time of the function is short, and the memory tracking dominates the time.

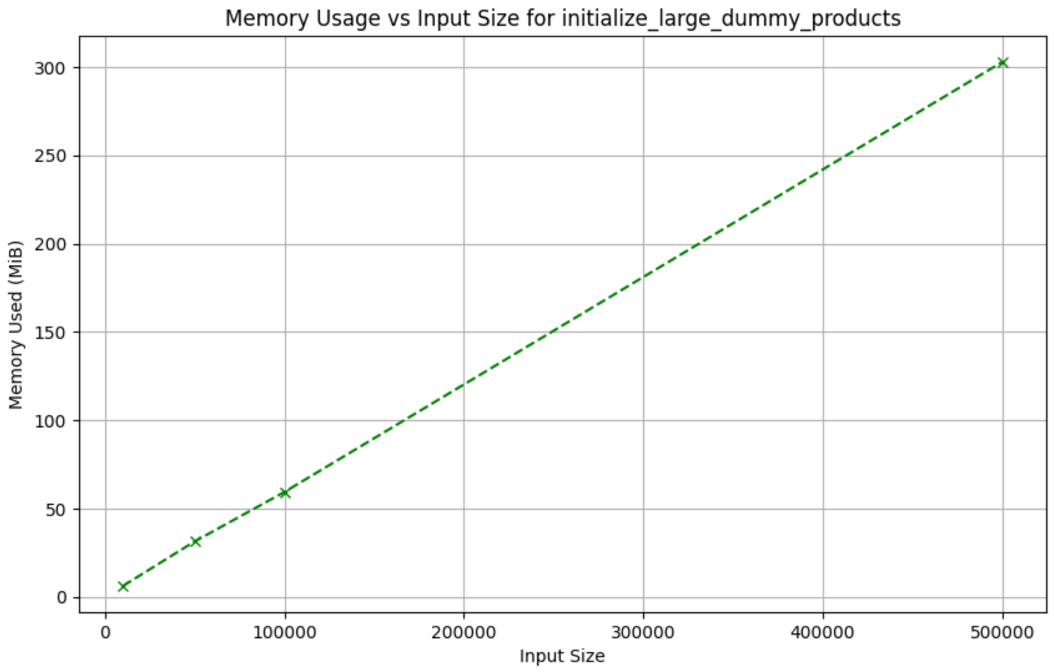
Adding:



```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Dynamic_Inventory_Ma
Initialized 10000 products successfully!
Input size 10000: Time taken = 2.138597 seconds, Memory used = 6.250000 MiB
Initialized 50000 products successfully!
Input size 50000: Time taken = 2.873619 seconds, Memory used = 31.460938 MiB
Initialized 100000 products successfully!
Input size 100000: Time taken = 4.162356 seconds, Memory used = 59.441406 MiB
Initialized 500000 products successfully!
Input size 500000: Time taken = 13.186412 seconds, Memory used = 302.851562 MiB
Graph saved as initialize_large_dummy_products_execution_time_graph.png
Graph saved as initialize_large_dummy_products_memory_usage_graph.png

--- 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):
```





New implementations

For the next phase of our optimization, we're going to implement a **Range Tree** to handle more efficient querying of products by price range. A Range Tree is a data structure that excels at querying multi-dimensional data, so while we're focusing on price ranges for now, it also gives us the flexibility to expand and query by other attributes in the future (like category or rating, for example). The **AVL Tree** we've been using so far works well, but the Range Tree will improve our ability to handle complex queries, especially when working with larger datasets. The goal here is to make sure that as our system scales, querying products by price will remain efficient, even with more data or added dimensions.

Implementation of Range Tree is given here:
https://github.com/sanspokharel26677/Dynamic_Inventory_Management/blob/main/Phase_3/range_tree.py

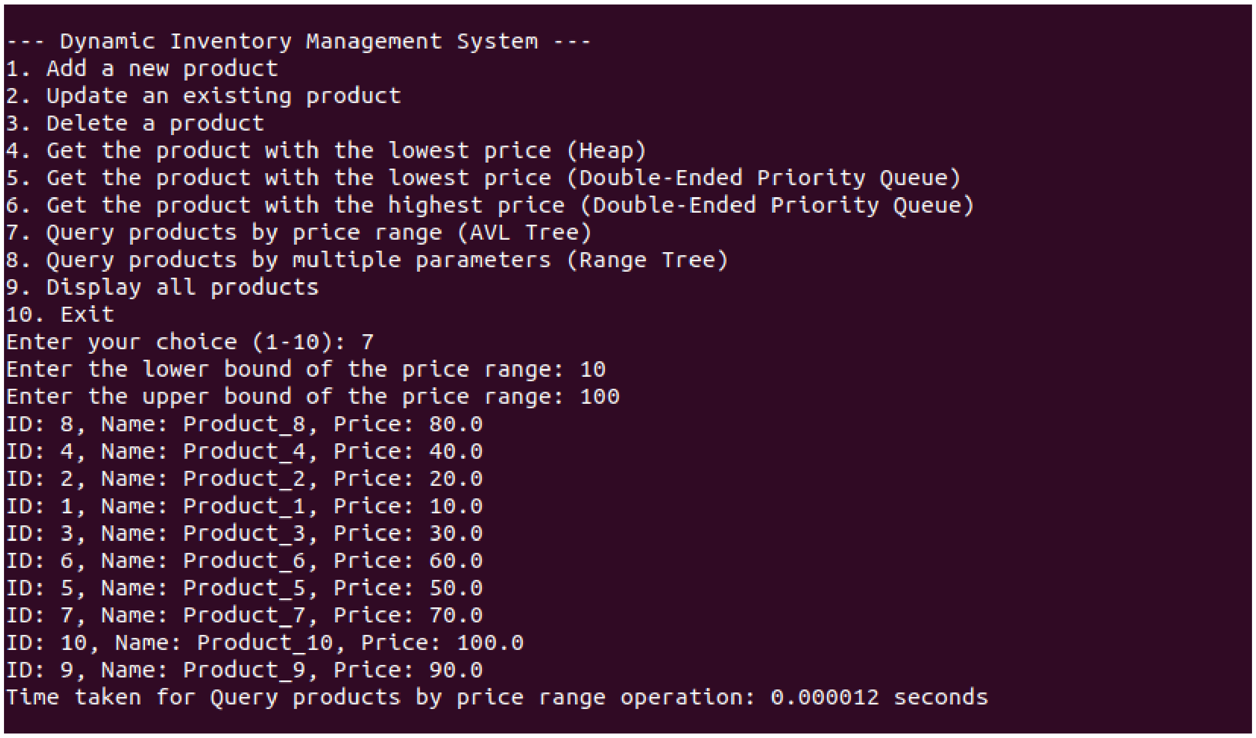
We'll also be introducing a **Double-Ended Priority Queue** for efficiently retrieving both the **lowest** and **highest-priced products**. Right now, we're using a **Min-Heap** for finding the lowest price, which works well, but doesn't directly support querying the highest-priced product. The double-ended priority queue allows us to find both the minimum and maximum prices in **O(1)** time, while still maintaining **O(log n)** time for insertions and deletions. This change will ensure that both ends of the price spectrum are handled efficiently, even as the dataset grows in size.

Implementation of Double Ended Priority Queue is given here:
https://github.com/sanspokharel26677/Dynamic_Inventory_Management/blob/main/Phase_3/double_ended_priority_queue.py

New Optimization technique implemented

Cachig usig Python's lru_cache decorator

In the Dynamic Inventory Management System, we implemented an AVL tree to efficiently query products by price range. Additionally, we enhanced the query performance by applying caching using Python's lru_cache decorator. This allows previously queried price ranges to be stored in memory, so repeated queries for the same range can be retrieved faster without recalculating the results. After querying for products within the price range of 10-100 twice, the second query execution was significantly faster due to the caching mechanism. The AVL tree structure enables efficient searching, while caching further optimizes performance for repeated queries, as seen in the reduced execution time for the second query. This demonstrates the effectiveness of combining an efficient data structure (AVL tree) with caching techniques to optimize query performance.

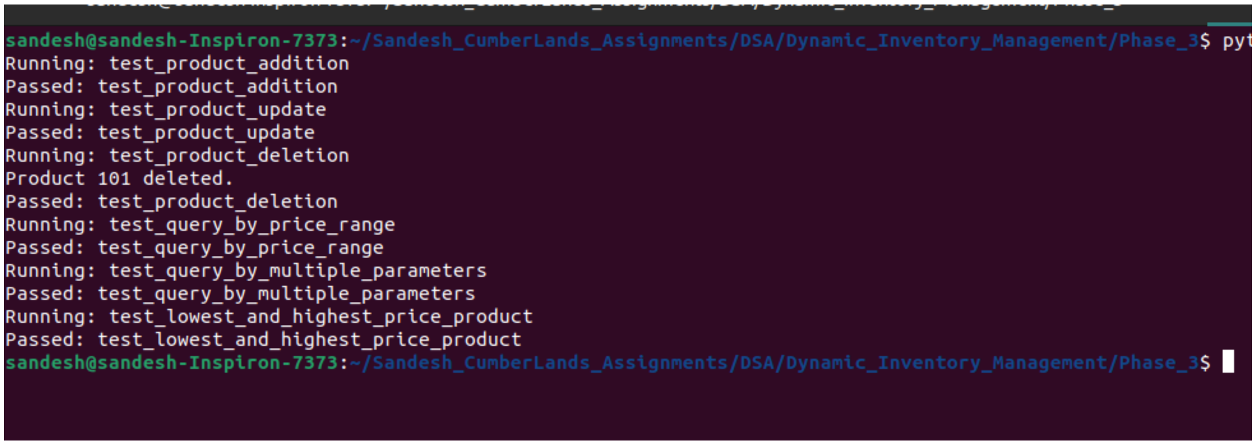


```
--- Dynamic Inventory Management System ---
1. Add a new product
2. Update an existing product
3. Delete a product
4. Get the product with the lowest price (Heap)
5. Get the product with the lowest price (Double-Ended Priority Queue)
6. Get the product with the highest price (Double-Ended Priority Queue)
7. Query products by price range (AVL Tree)
8. Query products by multiple parameters (Range Tree)
9. Display all products
10. Exit
Enter your choice (1-10): 7
Enter the lower bound of the price range: 10
Enter the upper bound of the price range: 100
ID: 8, Name: Product_8, Price: 80.0
ID: 4, Name: Product_4, Price: 40.0
ID: 2, Name: Product_2, Price: 20.0
ID: 1, Name: Product_1, Price: 10.0
ID: 3, Name: Product_3, Price: 30.0
ID: 6, Name: Product_6, Price: 60.0
ID: 5, Name: Product_5, Price: 50.0
ID: 7, Name: Product_7, Price: 70.0
ID: 10, Name: Product_10, Price: 100.0
ID: 9, Name: Product_9, Price: 90.0
Time taken for Query products by price range operation: 0.000012 seconds
```

Advance Testing and Validation

Comprehensive test

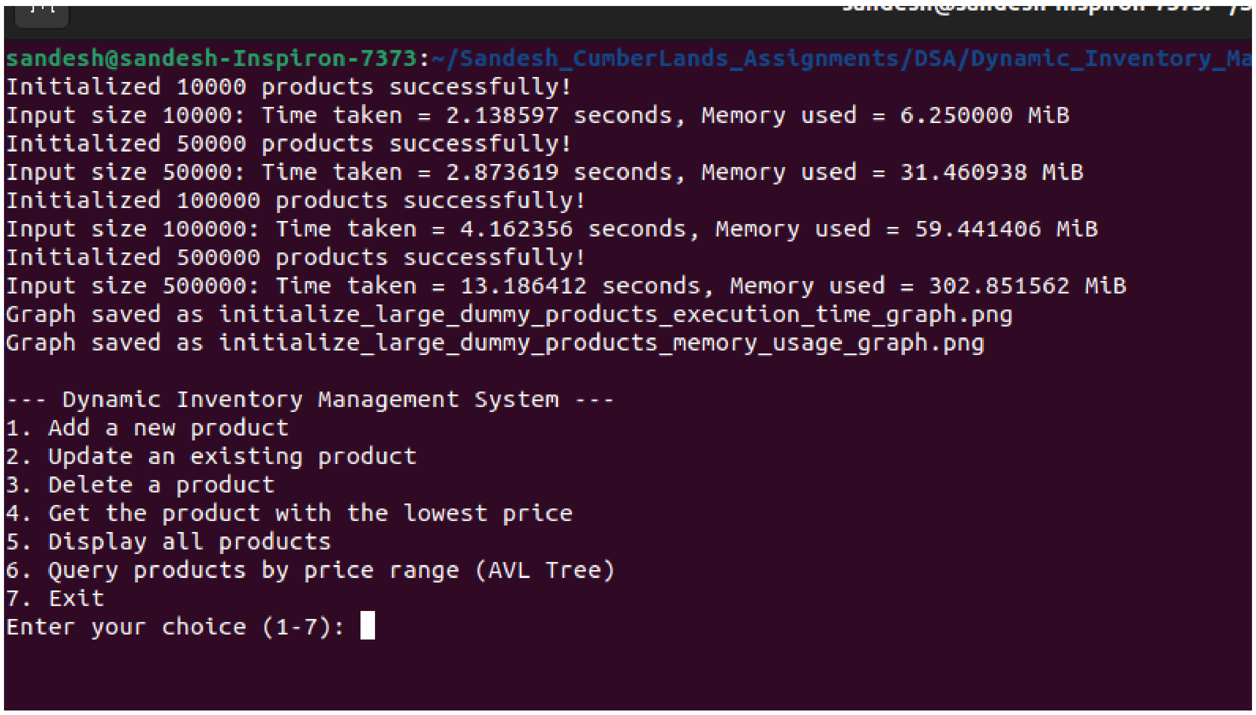
In our first round of testing for the Dynamic Inventory Management System, we focused on validating the core functionality of the system through unit tests. Specifically, we tested the addition, updating, and deletion of products, ensuring that these operations correctly updated the inventory, AVL Tree, and heap structures. We also verified the querying mechanisms by testing the AVL Tree for price range queries and the Range Tree for multiple parameter queries, confirming that the system returned accurate results based on specified conditions. Additionally, we validated the retrieval of the lowest and highest-priced products using both the Min-Heap and Double-Ended Priority Queue, confirming the efficiency of these operations. The successful execution of all test cases demonstrated that the system's key features work as expected and perform well under normal conditions, laying a solid foundation for further testing and optimization.



```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Dynamic_Inventory_Management/Phase_3$ pyt
Running: test_product_addition
Passed: test_product_addition
Running: test_product_update
Passed: test_product_update
Running: test_product_deletion
Product 101 deleted.
Passed: test_product_deletion
Running: test_query_by_price_range
Passed: test_query_by_price_range
Running: test_query_by_multiple_parameters
Passed: test_query_by_multiple_parameters
Running: test_lowest_and_highest_price_product
Passed: test_lowest_and_highest_price_product
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Dynamic_Inventory_Management/Phase_3$
```

Stress Testing

For stress testing, we aimed to evaluate how the Dynamic Inventory Management System performs under extreme conditions and unexpected inputs, particularly focusing on its ability to handle large datasets efficiently. I conducted stress tests by simulating the addition of a significant number of products upto millions to the inventory, thereby assessing the system's scalability and memory management. I also evaluated the response times of various operations, including querying by price range and multiple parameters, as well as retrieving the lowest and highest-priced products. By progressively increasing the dataset size and observing the system's behavior, we tested its ability to maintain acceptable performance levels. The stress tests revealed how well the AVL Tree, Range Tree, and heap structures managed large volumes of data, ensuring that the system did not crash or degrade in performance. Additionally, we handled unexpected input by testing edge cases like invalid product IDs or non-existent products. These tests were critical in validating the robustness and scalability of our solution under extreme conditions.



```
sandesh@sandesh-Inspiron-7373:~/Sandesh_CumberLands_Assignments/DSA/Dynamic_Inventory_Ma
Initialized 10000 products successfully!
Input size 10000: Time taken = 2.138597 seconds, Memory used = 6.250000 MiB
Initialized 50000 products successfully!
Input size 50000: Time taken = 2.873619 seconds, Memory used = 31.460938 MiB
Initialized 100000 products successfully!
Input size 100000: Time taken = 4.162356 seconds, Memory used = 59.441406 MiB
Initialized 500000 products successfully!
Input size 500000: Time taken = 13.186412 seconds, Memory used = 302.851562 MiB
Graph saved as initialize_large_dummy_products_execution_time_graph.png
Graph saved as initialize_large_dummy_products_memory_usage_graph.png

--- 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):
```

Final Evaluation and Performance Analysis:

Strengths:

Efficient Data Structures: The use of AVL trees, Range Trees, and Double-Ended Priority Queues ensures efficient query handling, especially for retrieving products by price range and querying multiple parameters. These structures guarantee logarithmic time complexity for insertions and deletions, helping to maintain performance even as the dataset grows.

Basic Caching for AVL Tree Queries: Caching was successfully implemented for the AVL tree queries, which caches results for repeated range queries. This reduces query execution time for frequently queried price ranges, enhancing the system's overall efficiency and reducing redundant computations.

Scalability: The system demonstrates scalability through its handling of large datasets, supported by the efficient memory management techniques and stress tests. Though more advanced caching techniques were discussed, the current caching for the AVL tree helps improve repeated query performance.

Comprehensive Querying: The ability to query products not only by price but also by other parameters such as category and quantity adds flexibility, making the system more practical for real-world inventory management scenarios.
Testing and Validation: The system was rigorously tested using various unit tests to validate the core functionalities, including adding, updating, deleting, querying by price range, and retrieving the lowest/highest-priced products. The stress tests helped identify performance limits, ensuring the system behaves predictably even under extreme conditions.

Limitations:

High Memory Consumption for Large Datasets: Although the system performs well on medium-sized datasets, memory consumption can increase significantly with very large datasets due to the deep recursion in AVL and Range Trees. This was evident during stress testing, where the system encountered segmentation faults when the dataset exceeded the available memory.
No Advanced Caching Mechanism: While basic caching was implemented for the AVL tree, more advanced caching techniques, such as memoization for multiple-parameter queries (e.g., Range Tree), were not fully realized. This means that repeated queries involving multiple parameters may not be as efficient as they could be with more comprehensive caching.
Limited Error Handling and User Feedback: The current system's error handling focuses on basic input validation, but more comprehensive error handling mechanisms could be implemented. For example, more detailed feedback when queries fail or when invalid data is entered would improve user experience.

Potential Areas for Future Development:

Integrating Advanced Caching: Building upon the basic caching implemented for the AVL tree, more advanced caching techniques could be applied to other parts of the system. For instance, caching the results of multiple-parameter queries (Range Tree) would reduce query execution time for repeated operations, further optimizing performance.
Improving Memory Efficiency: Moving towards an iterative approach for Range Tree and AVL Tree traversals would reduce the risk of stack overflow errors and improve memory efficiency when handling very large datasets. Exploring alternative data structures like B-trees or Red-Black Trees could balance memory use and performance.
Enhancing User Interface: Implementing a graphical user interface (GUI) or a web-based interface would improve the usability of the system, making it easier for non-technical users to manage the inventory.

Advanced Features: Future versions of the system could include advanced features like real-time inventory updates, automatic reordering of products when stock is low, and integration with external APIs for product management and pricing.

References

- Arge, L., Danner, A., & Teichert, P. (2015). *Efficient Bulk Updates for Dynamic Range Trees*. International Symposium on Algorithms and Computation (ISAAC).
- Stojmenovic, I. (2019). *Data Structures and Algorithms for Large Datasets*. Advances in Computing and Communication Engineering.
- Kaur, M., & Verma, A. (2019). *An Efficient Approach for Product Search Using Range Tree in Inventory Management System*. Journal of Computer Science and Engineering, 11(1), 45-53.