Project: Dynamic Inventory Management System

Shrisan Kapali

Student Id: 005032249

MSCS 532 – Algorithms and Data Structures

Satish Penmatsa

February 16, 2025

Abstract

Data structures and algorithms are essential for implementing and managing effective inventory management systems. This project outlines the design and implementation of a dynamic inventory management system, focusing on performance metrics such as choosing the correct data structures, execution time, space efficiency, and scalability.

The inventory management system is coded in Python and utilizes various data structures, including dictionaries, lists, tuples, stacks, priority queues, and heaps. It addresses these data structures' potential tradeoffs, challenges, and limitations, discussing how the inventory management system can be optimized for space and performance.

Additionally, the project includes test cases that validate the effectiveness of the chosen data structures. Finally, it highlights areas where the inventory management program can be improved and scaled for practical, real-life applications.

Application Context

The Customer Service Manager magazine (n.d.) defines inventory management as how a business or organization tracks and controls inventory. According to Coursera (2024), an inventory management system helps a company be organized by administering its inventory. It can also provide critical data to help the business understand the supply and demand trends and financial records and retain profitability. An effective inventory management system can help a business keep track of the correct count of items that are in stock, retain records of customers and their purchase history, and provide valuable insights such as how much a product is in demand, the average selling time of a product, customer choices, and many more.

In the real world, an organization or a business, even on a smaller scale, needs to manage thousands of items regularly (Theodorou et al., 2022). An effective inventory management

system is required so that organizations can continuously monitor and manage their inventory in real-time with high responsiveness (Marketing, 2024). It helps businesses optimize inventory levels and prevent inventory issues such as overstocking and stockouts. This can help businesses prevent financial losses as well, and on the customer side, it helps increase customer satisfaction when the product is readily available (Marketing, 2024).

In this project, a dynamic inventory management system has been created using Python to efficiently handle CRUD (create, retrieve, update, and delete) operations in inventory categories, prices, and quantities. In addition to handling these operations, the program measures and evaluates the performance metrics, such as the execution time of these operations and space complexity. It considers practical algorithms for efficient queries while searching for inventories and the correct use of data structure to create and modify objects in constant O (1) time. It provides test cases for the functionalities implemented and discusses common challenges, such as tradeoffs between data structures and edge cases on queries while developing these inventory management systems. It considers the optimization of the data structures using cache for performance while considering scaling for large data sets.

At a high level, the Python inventory management system has the following functionalities.

1. Creation of object models using applicable data structures such as dictionaries. Dictionary in Python is most suitable for storing data values that have a time complexity of O (1) (“Python Data Structure,” 2024).
2. Perform efficient updates to the defined object models. This can be done by organizing the object models using data structures such as linked lists, which provide O (1) time complexity for indexing and insertion (“Python Data Structure,” 2024).

3. Search current inventory using metadata fields such as object name, object price range, or ID.
4. Track changes in product prices by using Tuples to store the history of values.
5. Boost search performance by implementing LRU Cache.

Project implementation and Chosen Data Structures

The program has three main classes: Inventory, Category, and Product. The inventory class sits at the top of the hierarchy, which consists of multiple categories and products.

The inventory class uses dictionaries to store categories and products. A dictionary is a collection where each value is associated with a unique key (Beats, 2023). Dictionaries can have only one key per value, so the use of dictionaries also helps prevent redundancies and allows uniqueness. As dictionaries are mutable, the records that have these data types can be modified after creation (Python Data Structure", 2024). Dictionaries allow the values to be accessed using keys, and they are an ideal data structure to store, retrieve, and update data using unique keys. It has the uniform time complexity of O (1) (Beats, 2023). The dictionary in Python also has tons of built-in methods, such as get, contains, pop, and keys, which makes the execution of tasks easier.

Dictionaries were chosen over arrays, lists, and sets to store the collection of categories and products. The average time complexity for inserting, updating, searching, and deleting in a dictionary is constant time, O (1). In contrast, although arrays and lists are also mutable, accessing data within them requires sequential searches, which can take an average of O(n) time (Jacob, 2023). In lists, data must be stored in sequential order or index. In worst case, in a list, a search may take O(n) as it may require searching the entire list to find the object (Cormen et al., 2002). However, dictionaries are unordered, and their flexible nature allows them to store many

different data structures (“Comparison of Python Data Structures,” 2024). In arrays, a single array can only store data of the same type, which makes it less flexible, and it also has a time complexity of O(n) for insertion, deletion, searching, and traversal in average case (“Getting started with array data structure,” 2024). Due to the constant time complexity O (1) and the value flexibility that dictionaries provide, dictionaries were selected as the ideal data structure to store collections of products and categories in the inventory management system.

Similarly, two other classes, Category, which has category_id, name, and status fields, and Product, which contains product_id, name, price, description, Category, price history, and quantity, were created. Both the classes have update functionality, which updates the field values to the new values if passed. As these classes are initialized using dictionaries in the inventory class, the update takes a constant time of O (1) (Beats, 2023). The update is handled first by finding the class object using the unique key identifier, which takes a constant time O (1). If the key passed in is not present, the application raises a value error. If the fields are passed, the corresponding value is updated to the new value. The overall time complexity of the update function remains O (1) as it takes constant O (1) to search and O (1) to update.

The product class uses tuples to store the price history of the product. Tuples in Python are immutable collections, which means that once created, they cannot be modified (“The power of tuples,” 2024). They are also ordered collections; the items cannot be removed once inserted (“Tuples in Python,” 2025). Tuples are ideal for storing price changes, as they provide better data security and do not allow users to go back and change the prices of old records. This helps ensure that the old values do not change when users need to view the logs. They are also suitable for storing the chronological order of the price history, from the latest price to the oldest.

Although tuples have a time complexity of O(n) for lookups, they are still preferred over the dictionary, set, and linked list because they provide ordered and immutable data collection and do not require keys to store value. A list or array also has the same time complexity as tuples for lookup; however, as the data in a list or array are mutable, their value can be changed over time. In addition, the order or the index value in a list or arrays can also be swapped. Using tuples, as it stores the price in the order they were updated, the latest to the oldest, and being immutable, the data integrity is maintained. A linked list is also suitable for storing the price, where each node represents the price at a point, but they are mutable. However, for security reasons, tuples are the most desirable data structure for storing the price history.

The Product class has two additional functions: increase and decrease quantity. To perform these adjustments, the product to update must first be found using the key, which takes O (1), and the update takes O (1) time, taking the overall time complexity of O (1). If the passed-in key is invalid, the program raises a value error. This makes dictionaries very efficient in storing the collection of products and categories.

The pseudocode for the python implementation of the code that has been used in the inventory management system is as follows:

Example Creation of Product object

```
class Product:
    def __init__(self, id, name, price, category): # Constructor to create a new record
        self.id = id
        self.name = name
        self.price = price
        self.category = category
```

def update_price(new_price): # method to update price of the product

self.price = new_ price

Example function of adding product

function add_product(id, name, price, category):

First check if the id is present in the Inventory list:

If id is present in the list, throw an error and warn saying id exists

Create new Product object

Add this newly created product to the Inventory list

Example function of updating product meta data such as price

function update_product_price(id, new_price):

First check if the product can be found using the id:

If id is no found, return with a warning message

Update the price of the product

Add the old price of the product to the price history list for that product

Update the inventory product information using id as key and replace the product information with newly updated values

Code Snippets and Documentation

The complete Python program for the dynamic inventory management system can be found in the following GitHub account repository.

https://github.com/ShrisanKapali-Cumberlands/MSCS532_Project_Phase_4

Initially, a class category is created with the constructor to input category_id, name, and status. It has a function to update these field values to the new values if passed. The product class is then created with constructors with parameters such as product_id, name, price, description,

category, and quantity. It has functions to update these field values and has two additional functions to increase or decrease the quantity of the product. Furthermore, whenever a product's price is added or updated, the new price is appended to a `price_history` tuple.

The inventory class is then created, initializing categories and products as empty dictionaries. We have functions to check if the category and product IDs are unique, such that it always has the unique ID whenever a new product or category is added. For optimization, we have also implemented manual memoization and declared the memorized_search field, a type of dictionary that stores the results of the search functions.

```python
# Finally as we now have product and category class, create Inventory class
class Inventory:

    # Intialize inventory class with empty categories and product dictionary
    def __init__(self):
        self.categories = {}
        self.products = {}
        # Implementing manual caching
        self.memoized_search = {}

    ## ****************************************** ##
    # Inventory Category Management
    ## ****************************************** ##
    # Functions to add and update category and products
    # Each id needs to be unique so
    def is_category_id_unique(self, category_id: int) -> bool:
        return category_id not in self.categories

    # Also check if product id is unique
    def is_product_id_unique(self, product_id: int) -> bool:
        return product_id not in self.products
```

The inventory class has functions to perform CRUD operations on categories and products. It also includes functions to search categories and products using names, IDs, and price ranges, if applicable. While updating and deleting, we ensure that the ID passed in is present in the corresponding dictionaries. It raises an error if the ID is not present.

```python
# Update Category name or status
def update_category(self, cagetory_id: int, name: str = None, status: bool = None):
    # If passed in category id is not present, return error
    if cagetory_id not in self.categories:
        raise ValueError(
            "Unable to find the category for this passed in cateogry id"
        )

    # Use category update method to update the category details
    self.categories[cagetory_id].update(name, status)
    # Clear manual cache on each update
    self.clear_cache()

# Delete existing category
def delete_category(self, cagetory_id: int):
    # If passed in category id is not present, return error
    if cagetory_id not in self.categories:
        raise ValueError(
            "Unable to find the category for this passed in cateogry id"
        )

    del self.categories[cagetory_id]
    # Clear manual cache on each update
    self.clear_cache()
```

Similarly, functions were implemented to adjust the price quantity of an existing product, search products within the price range, and search products using category names or IDs. To improve the performance of the search, an automatic LRU caching mechanism and manual memoization by using a dictionary were also implemented.

Implementation Challenges and Solutions

Methods to add, edit, and delete categories and products were created in the inventory class. While adding a new record, whether it be a new category or a new product, as we used dictionaries, if a new record is inserted using the same key, the old record, which has the same key value, is overridden with the new record value (“Add same key in python dictionary,” 2024). To prevent this from happening, the program first checks if the key already exists in the collection, and if the key does exist, it raises a value error to warn the users that the same key already exists.

Unique keys can be used to create a new record in both the category and product collection. This method prevents data from being overridden; however, it may not be a practical solution in real-world applications. It can become a significant challenge if users are required to remember all the IDs of the categories and products stored in the program, especially with hundreds of thousands of entries. A more optimal approach would be to use libraries, such as

uuid, which generate a unique identifier each time a new record is created. The program can first query the dictionary collection using meta-fields like the name to perform searches or updates. If a match is found, it can then use the corresponding key from the found result to update the collection's value.

While updating the values of a category or product collection, the Python program raises a key error if an invalid ID is passed, and the key is not present in the dictionary. To solve this, whenever an update or additional function is to be performed on existing category or product data, we first ensure that the key exists in the collection. If the key does not exist, we throw an error warning the users that the key ID is invalid.

While searching for products or categories using the name, if a product or category name was inserted in a different case, and the user tries to search using a different name case, the search returns an empty list. For example, if the product name is “tomato” and the user searches the product using “TOMATO”, the search returns an empty list. To solve this problem, we implemented search by converting the search field and the collection field values to lowercase.

When a product price is updated, storing the price only in the price_history tuple is not informational if the users must view or track the price history timeline. It is more informational when the users can view the time the price was updated. To solve this problem, along with the updated price, the date when the price was updated was also stored in the price history tuple field.

Optimization of Data Structures

The Python dynamic inventory management program was optimized using various techniques to achieve faster execution times and optimal memory usage. It was designed to handle large datasets in cases where the application may be scaled. By analyzing the trade-offs of

different data structures, the program utilizes dictionaries, tuples, and class-based objects for storing, retrieving, updating, and deleting information.

The time complexity of using dictionaries was examined, with an average time complexity of O (1) for insert, retrieve, and delete operations. As a result, dictionaries were favored over traditional arrays for storing products and categories due to this efficiency. Tuples were chosen over linked lists to maintain the price history of a product. This preference is because tuples are immutable and store data in chronological order upon insertion. Additionally, tuples help ensure data integrity, as the price history should never change once recorded.

The chosen data structures for CRUD operations on categories and products optimize time complexity while ensuring security. However, when considering large data sets, to achieve higher performance, caching was implemented.

Caching is one of the optimization techniques where the data is stored in a faster memory hierarchy, such as RAM ("What is caching and how it works," n.d.). It increases data retrieval by directly accessing the data stored in a higher memory hierarchy without accessing the lower memory storage level ("What is caching and how it works," n.d.). In a memory hierarchy, the memory speed decreases from processor to cache to main memory, magnetic disk, and magnetic tape optical disk (Hennessy & David, 2017).

The Least Recently Used (LRU) cache was implemented in the Python program. In an LRU cache, the data is organized in the order they are used, such that the frequently used data are stored in the LRU cache, replacing the least used data when the cache is full (Mitra, 2023). The time complexity to store and retrieve data from a LRU cache is O (1) (Mitra, 2023).

To scale for faster performance, the inventory management program also uses manual memoization to store frequently queried results. It used dictionaries to store the results of the

queried search. The next time the same condition is queried, it fetches the result directly from the manual cache without performing the search computation, reducing the processing overhead. The retrieval happens in a constant time of O (1), delivering higher performance. However, as integers were used to store the product and category ID, custom keys had to be implemented to store these values in the manual memoization cache.

The implemented LRU caching and manual memoization utilize hardware resources, resulting in increased memory usage. These caches must be refreshed whenever a new record is added or an existing record is updated to ensure the values remain current. If a new category or product is added to the inventory program, it becomes necessary to clear the existing cache. This leads to the requirement of rebuilding the cache, which can pose challenges in situations where the application needs to be continuously updated.

Testing and Validation

Stress testing was performed to evaluate the performance of data structures used in the inventory management program. Twenty thousand categories and one hundred thousand products were inserted into the inventory. The execution time to insert these new categories was 0.017 seconds, and products were less than 0.4 seconds. This execution time shows that using a dictionary is optimal for inserting new records. Every time a new record was inserted, the manual cache was cleared.

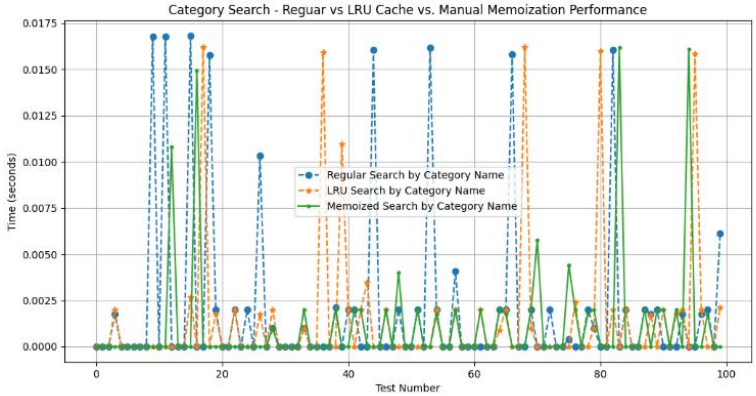
```
ds/Spring 2025/MSCS532 - Algorithm and Data Structures/Assignments/MSCS532-Project4/Project_Phase_4.py"
******************************
*** Initializing Inventory ***
******************************

******************************
Adding new category
******************************
20000 new categories are added
Execution time - 0.0176694393157959 seconds
Current inventory category size 20000
******************************

******************************
Adding new products
******************************
100000 new products are added
Execution time - 0.30220484733581543 seconds
Current inventory products size 100000
******************************
```

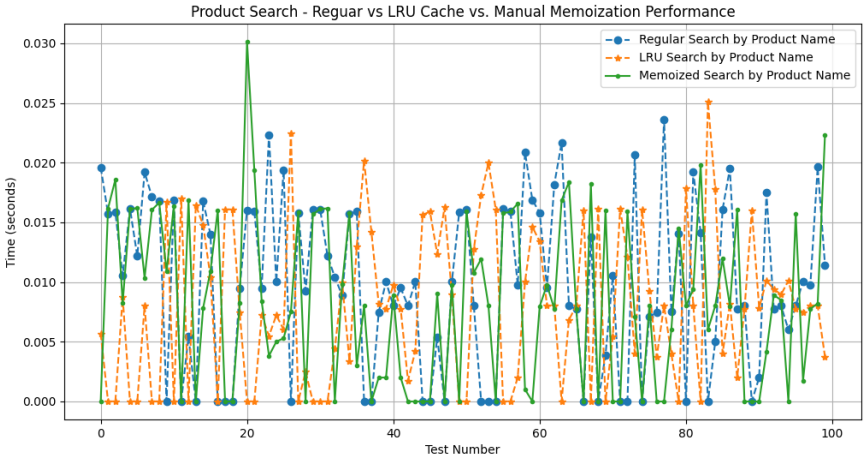
To test the CRUD operations on categories, test cases were written to update the status of category, search category by name, and delete a category. The test cases show that the methods implemented for category class works.

```
******************************
Basic functionality testing
******************************

Category class functionality testing
Find Category by name functionality testing
Find Category by name - Category-10000
Regular Search No Cache - [Category (10000), Name Category-10000, Current Status Active]
Regular Search LRU Cache - [Category (10000), Name Category-10000, Current Status Active]
Regular Search Custom Memoized Cache - [Category (10000), Name Category-10000, Current Status Active]

Update category - Category-10000 to False status
Regular Search No Cache after update - [Category (10000), Name Category-10000, Current Status Inactive]

Adding new category - New Category
Regular Search New Category No Cache Before Deletion - [Category (50000), Name New Category, Current Status Active]
Deleting new category - New Category
Regular Search New Category No Cache After Deletion - []
```

To test the functionalities implemented for products, test cases were written to add, update, increase the product quantity, and delete the products. These test cases verified the functionalities implemented are working correctly.

To test the caching implementation, the execution time it took to search the categories and products using regular dictionary search without using any cache, LRU cache, and manual dictionary memoization was evaluated. On average, performing these searches among the vast data, twenty thousand categories, and one hundred thousand products took less than 0.03 seconds. This shows that the dictionary data structure is ideal for retrieving values using the key in a constant time complexity of O (1).





As for the results obtained while plotting the execution time to search for the product and category by name, the search took almost the same time when the cache was implemented and not implemented. The search is fast enough because the dictionary already has an O (1) constant retrieval time. However, the processor must perform extra work to get the results. In our test case, the search name was randomized; however, if the same search parameters are used repeatedly, the cache may slightly outperform the regular dictionary search. In some scenarios, the cache search took more time, as it was to store the results of the search on the cache so that the data could be fetched faster next time.

The test case demonstrates that the data structures employed in developing the inventory management system offer optimal performance and are memory efficient, regardless of the data size. The total time required to retrieve data was under 0.03 seconds, which illustrates the effectiveness of these data structures in managing CRUD operations.

Final evaluation and potential areas for future development

In conclusion, the data structures utilized in developing the inventory management system provide an optimal time complexity of O (1) for basic CRUD operations concerning inventory categories and products. LRU Caching has been implemented to enhance performance.

While testing among twenty thousand categories and a hundred thousand products, the time complexity to search the record using factors such as name, price range, and ID was optimal, below 0.3 seconds. The test cases also covers the accuracy of the functionalities implemented.

The Python inventory program currently uses only hardware resources for storage and cache. However, practical, real-life applications must be web-based so users can access them without running the program on their local machines. The fundamentals of the inventory program analyze many data structures and use the best data structures like dictionaries, tuples, and lists to achieve optimal time and space complexity.

The next step to scale this application is to integrate it with database management systems like MySQL and MongoDB and have a user interface such that users can interact with the system. Angular, Vue.js, and React are some of the most commonly used frontend frameworks that can be used to develop the user interface (Mendes & Rodrigues, 2024). In addition to the Python program, REST APIs will need to be implemented so that the frontend application can perform the required functions. Some basic examples of APIs are the POST endpoint, which creates categories and products; the PUT endpoint, which updates them; and the DELETE endpoint, which deletes a category and product.

Redis, Apache Ignite, and Hazelcast are some of the commonly used in-memory caching frameworks that can improve the performance of web-based applications (“Top 10 in-memory caching frameworks for web application development in 2023,” 2023). Multithreading can effectively improve performance if functions need to be performed in batches. As multithreading allows the execution of parallel operations, if the operations are independent of one another, it can enhance the performance by a significant factor (“Benefits of multithreading,” n.d.).

Finally, to maintain the code standards and check if all the functions are performing well, test cases must be written and updated whenever code changes occur. Regression testing, stress testing, and end-end testing must be done to ensure that the program is functioning well and delivers optimal performance.

References

Beats, B. (2023, April 24). Python dictionary: Understanding methods with examples. *Medium*. https://binarybeats.medium.com/python-dictionary-understanding-methods-with-examples-aa367ede8650

Benefits of multithreading - javatpoint. (n.d.). *Javatpoint*. https://www.javatpoint.com/benefits-of-multithreading

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). Random House Publishing Services. https://reader2.yuzu.com/books/9780262367509

GeeksforGeeks. (2024, February 20). Add same key in Python dictionary. *GeeksforGeeks*. https://www.geeksforgeeks.org/add-same-key-in-python-dictionary/

GeeksforGeeks. (2024, July 25). Python data structures. *GeeksforGeeks*. https://www.geeksforgeeks.org/python-data-structures/

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture* (6th ed.). Elsevier S & T. Available from Yuzu Reader.

How can inventory management improve customer experience? (n.d.). *CSM – Customer Service Manager Magazine*. https://www.customerservicemanager.com/how-can-inventory-management-improve-customer-experience/

Jacob, I. (2023, February 8). All you need to know about using hashmaps in Python. *Turing*. https://www.turing.com/kb/how-to-use-hashmap-in-python

Marketing. (2024, March 12). Dynamic inventory optimization: Adapting to shifting demand patterns. *OrderCircle*. https://ordercircle.com/dynamic-inventory-optimization-adapting-to-shifting-demand-patterns/

Mendes, A., & Rodrigues, O. (2024, November 4). Top 10 best front end frameworks in 2025. *Imaginary Cloud: Software Development for Digital Acceleration*. https://www.imaginarycloud.com/blog/best-frontend-frameworks

Mitra, A. (2023, November 9). Design an LRU cache. *Medium*. https://medium.com/@abhishek.amjeet/design-an-lru-cache-447f49df7bbf

Technology Hits. (2024, February 27). The power of tuples: When and why to choose them over lists. *Medium*. https://medium.com/technology-hits/the-power-of-tuples-when-and-why-to-choose-them-over-lists-a421a77ff153

Theodorou, E., Spiliotis, E., & Assimakopoulos, V. (2022, December 27). Optimizing inventory control through a data-driven and model-independent framework. *EURO Journal on Transportation and Logistics*. https://www.sciencedirect.com/science/article/pii/S2192437622000280#:~:text=A%20promising%20alternative%20to%20the,%2C%202020)%2C%20among%20others%2C%20among%20others)

Top 10 in-memory caching frameworks for web application development in 2023. (2023, October 9). *FROMDEV*. https://www.fromdev.com/2023/05/top-10-in-memory-caching-frameworks-for-web-application-development-in-2023.html

Tuples in Python. (2025, January 4). *GeeksforGeeks*. https://www.geeksforgeeks.org/tuples-in-python/

What is caching and how it works | AWS. (n.d.). *AWS*. https://aws.amazon.com/caching/

What is inventory management? Benefits, challenges, and methods. (2024, December 4). *Coursera*. https://www.coursera.org/articles/inventory-management