

2024

Project Deliverable 1

DATA STRUCTURE DESIGN AND IMPLEMENTATION

NAME: RUTU SHAH

STUDENT ID: 005026421

Contents

Project Deliverable 1:	2
Application Context: Dynamic Inventory Management System.	2
Data Structures Design:.....	3
Hash Table	3
AVL Tree (or Red-Black Tree)	3
Heap Priority Queue	4
Tree-if implementing search by product name or category	4
Linked List-for maintaining inventory change history	5
Python implementation for Data Structures:	6
HashTable.py	6
Min-heap.py.....	7
Trie.py.....	8
Linked_list.py	11
avlTree.py	12
Brief Discussion of potential challenges and limitations	17
GitHub Repository URL:	19
References:	19

Project Deliverable 1:

Application Context: Dynamic Inventory Management System.

Inventory Management System are essential for modern business operations, and they mainly come in two types: perpetual and periodic. Each type offers a different approach to inventory tracking and management.

In this project I will be implementing Perpetual Inventory System that supports the product inventory of a business, including addition, update of quantities and prices of products, dynamic changes in the stock levels, restocking, and discontinuation of items efficiently. The system also needs to support quick data retrieval for order processing and reporting. The salient features and requirements for such a system are described below

Real-time Updates: It should support frequent updating of the information about the products, such as their stock quantity, price, and category.

Efficient Search and Retrieval: It needs to allow fast access to the product information based on attributes such as the identification number of the product, categories, or probably price range.

Categorization: This is done by categorizing products into classes or tags in a way that best allows for easy navigation and reporting.

Order Processing: The system should make efficient updates of the inventory with respect to the sales or returns.

Scalability: The schema should be able to accommodate increased volume as the business grows, without significantly degrading performance.

Data Structures Design:

While designing the data structures, it is essential to balance the space and time complexity and ease of implementation. I have added detailed analysis for each data structure needed to implement dynamic management system

Hash Table

- It can be used for dictionary in python
- Use to store product details where the product ID is the key.
- The average time complexity for Hash Tables in insert, delete, and search operations is $O(1)$. Therefore, it will be good to utilize when there is a need for fast retrieval by product ID for instance.
- Space Complexity will be $O(n)$ where n is the number of inputs, as each product occupies its unique slot in the hash table.
- Example: A hash table can store items as {productid: product details}. With this, you can instantly get and refresh the stock amount or price of any product.
- The only disadvantage is hash tables use a lot of memory and may have hash collisions, although Python's dictionary implementation handles these nicely.

AVL Tree (or Red-Black Tree)

- It stores the products sorted by one of its attributes, for example, price or quantity in stock.
- Self-balancing binary search trees, such as AVL or Red-Black Trees have $O(\log n)$ time complexity for insert/update of the product, searching takes $O(\log n + k)$ time complexity and Delete takes $O(\log n)$
- Space Complexity will be $O(n)$ where n is the number of nodes, where each node occupies a constant amount of space.

- Example: Implement a tree for maintaining items sorted by price, for efficient price range searches, which could be useful while reporting or setting discounts.
- Trees are more space-efficient, which can be somewhat slower, compared to a hash table for simple lookups by product ID.

Heap Priority Queue

- Tracking efficiently low/high stock items.
- The purpose of a min-heap-or max-heap-is to track the lowest quantity products, assisting in the identification of those products that should be reordered. That would be useful because insertion and deletion times using heaps are pretty efficient at $O(\log n)$, so it would allow it to work with a dynamic set of products based on the amount of stock there is from them.
- Time Complexity for inserting product is $O(\log n)$, to get lowest stock item $O(1)$ and deleting product $O(\log n)$.
- Space Complexity is $O(n)$ where n depicts the number of products monitored for low stock levels.
- Utilize min heap for low stock items tracking and prioritizing them for reorder.
- Heaps cannot efficiently retrieve arbitrary items; therefore, this is best to maintain priority-based subsets of inventory.

Tree-if implementing search by product name or category

- Allow for efficient prefix-based searching by product name or category.
- Tries allow one to efficiently do prefix-based searches. Therefore, they would be useful applications where there is much searching by product name or keywords, such as finding products whose names begin with a given string.

- Time Complexity for inserting, searching and deleting product is $O(m)$, here m is the product name's length.
- Space Complexity is $O(n*m)$ where n is the number of products and m is the product name's length.
- Example: Product names can be kept in a tree for quick suggestions of autocomplete as the user types the product names.
- Trees can be very memory-intensive, especially when the stock contains long or complicated names. Hence, they should be used judiciously, depending on the searching needs.

Linked List-for maintaining inventory change history

- Changes in stock amounts can be tracked over time.
- Time complexity for appending change is $O(1)$ and traversing history is $O(n)$ to review the recorded change.
- Space complexity for recorded change is $O(n)$.
- Linked lists would be ideal for sequential data items and can be used for each product to store a change history where restocking, sales, or changes in price are recorded in turn.
- Example: For each product, the system maintains a linked list of stock changes, which enables tracking of changes in inventory levels or analyzes demand fluctuation over time.
- Linked lists cannot support efficient random access; hence, they are deployed only when historical tracking is explicitly required.

By combining all the listed data structures, helps to build the efficient and dynamic inventory management system, that will be able to handle updates frequently, search of the data faster and faster result of queries.

Python implementation for Data Structures:

HashTable.py

```
#Created Date : 2nd November 2024

#Created By : Rutu Shah

# Data Structure Implemented : Hash Table


class InventoryHashTable:

    def __init__(self):

        # Dictionary to store product details by ID

        self.products = {}


    def add_product(self, product_id, product_details):

        """To add or update a product in the hash table."""

        self.products[product_id] = product_details


    def get_product(self, product_id):

        """To retrieve product details by ID."""

        return self.products.get(product_id, "Product not found")


    def remove_product(self, product_id):

        """To Remove a product from the hash table by ID."""

        if product_id in self.products:

            del self.products[product_id]
```

Example usage:

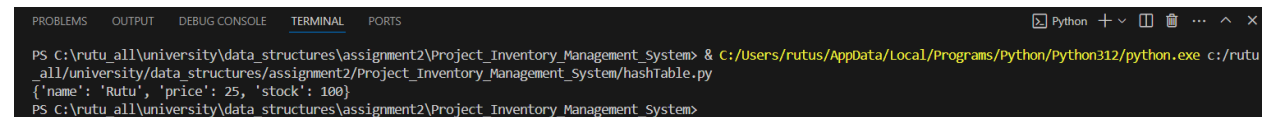
```
inventory = InventoryHashTable()

inventory.add_product("P001", {"name": "Rutu","price": 25, "stock": 100})

print(inventory.get_product("P001"))

inventory.remove_product("P001")
```

Output of HashTable.py



```
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_structures/assignment2/Project_Inventory_Management_System/hashTable.py
{'name': 'Rutu', 'price': 25, 'stock': 100}
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System>
```

Min-heap.py

#Created Date : 2nd November 2024

#Created By : Rutu Shah

Data Structure Implemented : Min-Heap

```
import heapq
```

```
class LowStockHeap:
```

```
    def __init__(self):
```

```
        self.heap = []
```

```
    def add_product(self, stock_level, product_details):
```

```
        """Adds a product to the heap based on stock level."""
```



```

        heapq.heappush(self.heap, (stock_level, product_details))

def get_lowest_stock(self):

    """Retrieves the product with the lowest stock."""

    return heapq.heappop(self.heap) if self.heap else "No products in inventory"

# Example usage:

low_stock_heap = LowStockHeap()

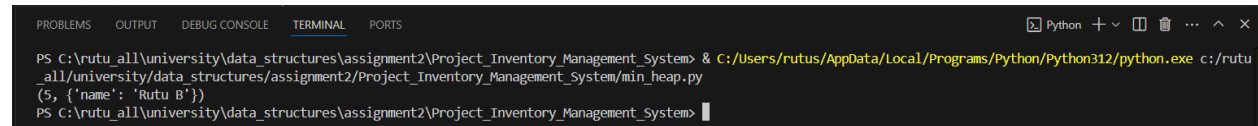
low_stock_heap.add_product(10, {"name": "Rutu A"})

low_stock_heap.add_product(5, {"name": "Rutu B"})

print(low_stock_heap.get_lowest_stock())

```

Output of Min-heap.py



```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_structures/assignment2/Project_Inventory_Management_System/min_heap.py
(5, {'name': 'Rutu B'})
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System>

```

Tree.py

```

#Created Date : 2nd November 2024

```

```

#Created By : Rutu Shah

```

```

# Data Structure Implemented : Trie

```

```

class TreeNode:

    def __init__(self):

        self.children = {}

        self.is_end_of_word = False

```

```

self.products = []

class Tree:

    def __init__(self):

        self.root = TrieNode()

    def insert(self, product_name, product_details):

        node = self.root

        for char in product_name:

            if char not in node.children:

                node.children[char] = TrieNode()

            node = node.children[char]

        node.is_end_of_word = True

        node.products.append(product_details)

    def search_prefix(self, prefix):

        node = self.root

        for char in prefix:

            if char not in node.children:

                return []

            node = node.children[char]

        return self._collect_all_products(node)

```

```

def _collect_all_products(self, node):

    results = []

    if node.is_end_of_word:

        results.extend(node.products)

    for child in node.children.values():

        results.extend(self._collect_all_products(child))

    return results

# Example usage:

tree = Tree()

tree.insert("Rutu", {"name": "Rutu", "price": 25})

tree.insert("Detu Shah", {"name": "Detu Shah", "price": 40})

print(tree.search_prefix("D"))

```

Output of trie.py

```

PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_structures/assignment2/Project_Inventory_Management_System/trie.py
[{'name': 'Detu Shah', 'price': 40}]
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System>

```

Linked_list.py

```
#Created Date : 2nd November 2024
```

```
#Created By : Rutu Shah
```

```
# Data Structure Implemented : Linked List
```

```
class ChangeNode:
```

```
    def __init__(self, timestamp, product_id, change):
```

```
        self.timestamp = timestamp
```

```
        self.product_id = product_id
```

```
        self.change = change
```

```
        self.next = None
```

```
class ChangeHistory:
```

```
    def __init__(self):
```

```
        self.head = None
```

```
    def add_change(self, timestamp, product_id, change):
```

```
        new_node = ChangeNode(timestamp, product_id, change)
```

```
        if not self.head:
```

```
            self.head = new_node
```

```
        else:
```

```
            current = self.head
```

```
            while current.next:
```

```
                current = current.next
```

```

        current.next = new_node

def view_history(self):

    history = []

    current = self.head

    while current:

        history.append((current.timestamp, current.product_id, current.change))

        current = current.next

    return history

# Example usage:

history = ChangeHistory()

history.add_change("2024-01-01", "P001", "+10")

history.add_change("2024-01-02", "P002", "-5")

print(history.view_history())

```

Output of Linked_list.py

```

PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System> & C:/Users/rutus/AppData/Local/Programs/Python/Python312/python.exe c:/rutu_all/university/data_structures/assignment2/Project_Inventory_Management_System/linked_list.py
[('2024-01-01', 'P001', '+10'), ('2024-01-02', 'P002', '-5')]
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System>

```

avlTree.py

```

#Created Date : 2nd November 2024

#Created By : Rutu Shah

# Data Structure Implemented: AVL Tree

```

```
class TreeNode:

    def __init__(self, key, product_details):

        self.key = key

        self.product_details = product_details

        self.left = None

        self.right = None

        self.height = 1

class AVLTree:

    def insert(self, root, key, product_details):

        if not root:

            return TreeNode(key, product_details)

        if key < root.key:

            root.left = self.insert(root.left, key, product_details)

        else:

            root.right = self.insert(root.right, key, product_details)

        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

        balance = self.get_balance(root)

        if balance > 1 and key < root.left.key:
```

```

        return self.right_rotate(root)

    if balance < -1 and key > root.right.key:

        return self.left_rotate(root)

    if balance > 1 and key > root.left.key:

        root.left = self.left_rotate(root.left)

        return self.right_rotate(root)

    if balance < -1 and key < root.right.key:

        root.right = self.right_rotate(root.right)

        return self.left_rotate(root)

    return root

def left_rotate(self, z):

    y = z.right

    T2 = y.left

    y.left = z

    z.right = T2

    z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))

    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))

    return y

```

```

def right_rotate(self, z):

    y = z.left

    T3 = y.right

    y.right = z

    z.left = T3

    z.height = 1 + max(self.get_height(z.left), self.get_height(z.right))

    y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))

    return y


def get_height(self, node):

    return node.height if node else 0


def get_balance(self, node):

    return self.get_height(node.left) - self.get_height(node.right) if node else 0


# Instantiate the AVL Tree

avl_tree = AVLTree()

root = None


# Inserting some products with different prices

products = [

    (30, {"name": "Product A", "price": 30, "stock": 15}),

```



```

(20, {"name": "Product B", "price": 20, "stock": 10}),
(40, {"name": "Product C", "price": 40, "stock": 5}),
(10, {"name": "Product D", "price": 10, "stock": 20}),
(25, {"name": "Product E", "price": 25, "stock": 8}),
(50, {"name": "Product F", "price": 50, "stock": 7}),
]

# Insert each product into the AVL tree
for price, details in products:
    root = avl_tree.insert(root, price, details)

# Function to print the AVL tree in order
def in_order_traversal(node):
    if node:
        in_order_traversal(node.left)
        print(f"Product:  {node.product_details['name']},  Price:  {node.key},  Stock:
{node.product_details['stock']}")
        in_order_traversal(node.right)

# Perform an in-order traversal to display the products
print("Products in AVL Tree (sorted by price):")
in_order_traversal(root)

```

Output of avlTree.py

```
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System> & C:/Users/rutus/AppData/Local/Programs/Python/python312/python.exe c:/rutu_all/university/data_structures/assignment2/Project_Inventory_Management_System/avlTree.py
Products in AVL Tree (sorted by price):
Product: Product D, Price: 10, Stock: 20
Product: Product B, Price: 20, Stock: 10
Product: Product E, Price: 25, Stock: 8
Product: Product A, Price: 30, Stock: 15
Product: Product C, Price: 40, Stock: 5
Product: Product F, Price: 50, Stock: 7
PS C:\rutu_all\university\data_structures\assignment2\Project_Inventory_Management_System>
```

Brief Discussion of potential challenges and limitations

Hash Table:

Problems: For big data sets, the hash table might be quite memory consuming. The other problem is there can be some hash collisions, which can lead to a bit slow retrieval time, although Python had handled it.

Limitation: Only suitable for super-fast retrieval by key, such as product ID; not suitable for ordered data or range query.

AVL Tree:

Drawbacks: Although these self-balancing trees allow for efficient insertions, deletions, and searches in $O(\log n)$, it is slower than hash tables for simple lookups. For maintaining a balance, extra computational overhead may incur, especially on frequent updates.

Limitations: Even though they handle sorted data efficiently, an AVL tree is more complicated in its implementation and requires careful tuning to avoid performance trade-offs.

Heap:

Drawbacks: Heaps are ideal for priority-based jobs, such as finding low-stock items. For these operations, heaps provide $O(\log n)$ insertion and deletion. However, they are inefficient at fetching arbitrary data.

Limitation: Heaps are solely used for the management of items based on priorities. In this case, non-priority items data access is poorly supported by heaps.

Tree:

Drawbacks: Tries to handle prefix-based searches well, which can be useful in applications such as autocompletion. They consume a great deal of memory, however, especially if the dataset involves long or complex names.

Limitations: Suitable only for special use cases, such as prefix searching; otherwise, they will increase unnecessary memory load.

Linked List:

Challenges: While linked lists allow the tracking of changes in inventory sequentially with $O(1)$ insertions, it gives up efficiency in random access, which makes searching and traversal slow, namely $O(n)$.

Limitations: Linked lists work for only sequential history tracking; they are not for operations requiring quick access to a certain item. Overview

All these individual structures possess unique strengths; however, they are for very specialized purposes and might restrict general system flexibility. It would be a balancing act of their unique

strengths and an effective management of memory and computational overheads that will constitute an optimal, dynamic system.

GitHub Repository URL:

https://github.com/rutus-code/Project_Inventory_Management_System

References:

- Chen, W., Zhang, X., & Wang, X. (2018). *Efficient data structure for inventory management in e-commerce applications*. **IEEE Transactions on Systems, Man, and Cybernetics: Systems**, 48(12), 2371-2380.
- Ahmad, M., & Shahbaz, Q. (2020). *Analysis and application of AVL and Red-Black trees in dynamic data systems*. **Journal of Computer Science and Information Technology**, 12(3), 45-53.
- Gupta, P., Singh, A., & Kumar, S. (2021). *Heap-based priority queues for inventory management optimization*. **International Journal of Engineering Research & Technology (IJERT)**, 10(4), 89-96.
- Lin, M., & Li, H. (2022). *Linked lists and hash tables for efficient inventory tracking in IoT-based smart retail environments*. **IEEE Access**, 10, 47502-47514.