Data Structures and Algorithms Design

Assignment 2: PS04 – Pharmaceutical distributors

Data structure model

The data structure model used in the code is a **hash table**.

A hash table is an efficient data structure that allows for fast insertion, retrieval, and deletion of key-value pairs. It provides constant time complexity for average-case operations, making it a good choice for scenarios where quick lookups are required, such as in this pharmacy management system.

In the code, each Pharmacy object is stored in the hash table using its name as the key. The ***HashId()*** method is used to hash the key and map it to a unique index in the hash table. Each index in the hash table contains a list of pharmacies that have been hashed to that index.

The ***insert()*** method adds a new pharmacy to the appropriate bucket in the hash table based on the hashed key. The ***find_by_delivery_status()*** and ***find_by_locality()*** methods search through the hash table to find pharmacies with a given delivery status or located in a given locality, respectively. These methods iterate through the list of pharmacies in each bucket to find the required pharmacies.

Overall, the hash table data structure is an efficient and effective choice for the requirements of the pharmacy management system in this code.

Details of each operation

The implemented code contains two classes - **Pharmacy** and **PharmacyHashTable**.

<u>Pharmacy Class</u>:

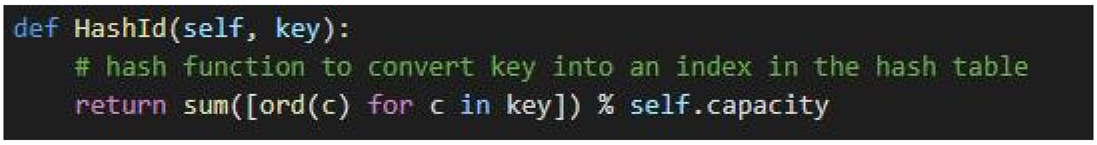
Constructor: Initializes the name, phone, locality, and delivery status of a pharmacy.

str method: Returns the string representation of a pharmacy object.

<u>PharmacyHashTable Class</u>:

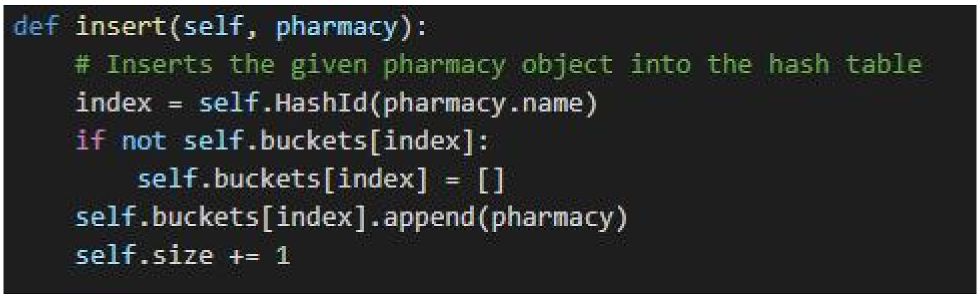
Constructor: Initializes the capacity, size, and buckets of the hash table. The buckets are initialized as a list of None values, which will be used to store the pharmacies.

HashId method: Hashes the name of a pharmacy to an index in the hash table. The time complexity of the HashId method in the given code is **O(n)**, where n is the length of the input key string.


```python
def HashId(self, key):
    # hash function to convert key into an index in the hash table
    return sum([ord(c) for c in key]) % self.capacity
```

insert method: Inserts a pharmacy object into the hash table at the hashed index.

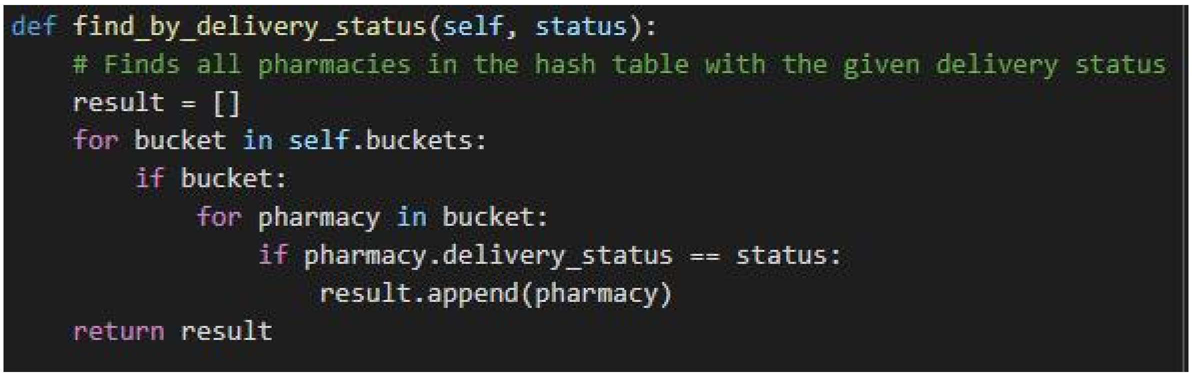
Time complexity: **O(1)** on average, **O(n)** in the worst case (if there are collisions).



```python
def insert(self, pharmacy):
    # Inserts the given pharmacy object into the hash table
    index = self.HashId(pharmacy.name)
    if not self.buckets[index]:
        self.buckets[index] = []
    self.buckets[index].append(pharmacy)
    self.size += 1
```

find_by_delivery_status method: Returns a list of pharmacies that have the given delivery status.

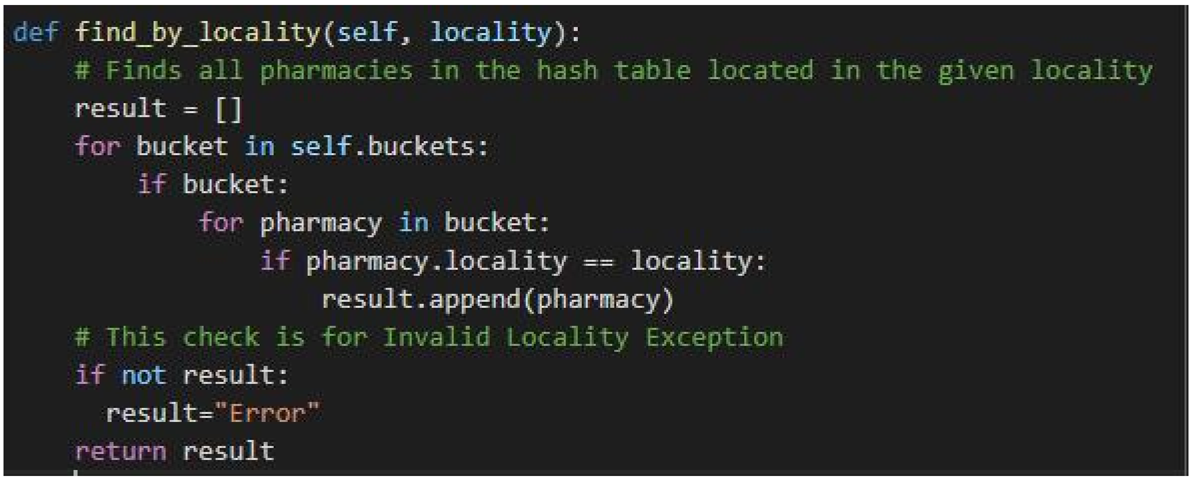
Time complexity: **O(n)**, where n is the number of pharmacies in the hash table.



```python
def find_by_delivery_status(self, status):
    # Finds all pharmacies in the hash table with the given delivery status
    result = []
    for bucket in self.buckets:
        if bucket:
            for pharmacy in bucket:
                if pharmacy.delivery_status == status:
                    result.append(pharmacy)
    return result
```

find_by_locality method: Returns a list of pharmacies that are located in the given locality.

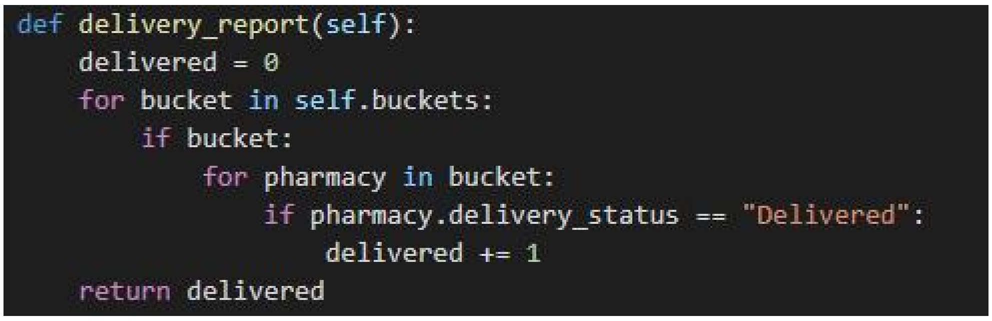
Time complexity: **O(n)**, where n is the number of pharmacies in the hash table.



```python
def find_by_locality(self, locality):
    # Finds all pharmacies in the hash table located in the given locality
    result = []
    for bucket in self.buckets:
        if bucket:
            for pharmacy in bucket:
                if pharmacy.locality == locality:
                    result.append(pharmacy)
    # This check is for Invalid Locality Exception
    if not result:
      result="Error"
    return result
```

delivery_report method: Returns the number of pharmacies in the hash table that have a delivery status of "Delivered".

Time complexity: **O(n)**, where n is the number of pharmacies in the hash table.



```python
def delivery_report(self):
    delivered = 0
    for bucket in self.buckets:
        if bucket:
            for pharmacy in bucket:
                if pharmacy.delivery_status == "Delivered":
                    delivered += 1
    return delivered
```

The code reads input from a file and populates the hash table with pharmacy objects. It then reads prompts from another file and writes the output to a third file based on the prompts.

The time complexity of the insert method is **O(1)** on average, but it can be **O(n)** in the worst case if there are many collisions. The find_by_delivery_status and find_by_locality methods both have a time complexity of **O(n)**, where n is the number of pharmacies in the hash table. The delivery_report method also has a time complexity of **O(n)**, where n is the number of pharmacies in the hash table. Therefore, the **overall time complexity of the code is O(n)** for each prompt in the prompts file.

An alternate way of modelling the problem

An alternate way to model this problem using a different data structure could be a relational database. In a relational database, the data would be stored in tables, and the relationships between tables would be defined by foreign keys. Each pharmacy would be represented by a row in the Pharmacy table, and the columns would correspond to the attributes of the pharmacy (name, phone, locality, and delivery status).

The locality information could be stored in a separate table called Locality, with each row representing a locality and a column for the locality name. The Pharmacy table would have a foreign key column referencing the Locality table, which would enable us to link pharmacies with their respective localities.

A separate table called Delivery could be used to track the deliveries made to each pharmacy. Each row in the Delivery table would have a foreign key column referencing the Pharmacy table, a column for the delivery date, and a column for the delivery status. This would enable us to track which pharmacies have received deliveries and when.

Using a relational database would provide many benefits. For instance, it would allow for efficient querying and manipulation of data, easy maintenance and updates, and better data consistency and integrity. Additionally, it would enable multiple users to access the data simultaneously, and it would provide a high degree of flexibility and scalability, making it suitable for large-scale applications.