Assignment-11.3

HT.NO:2303A51100
Batch:05

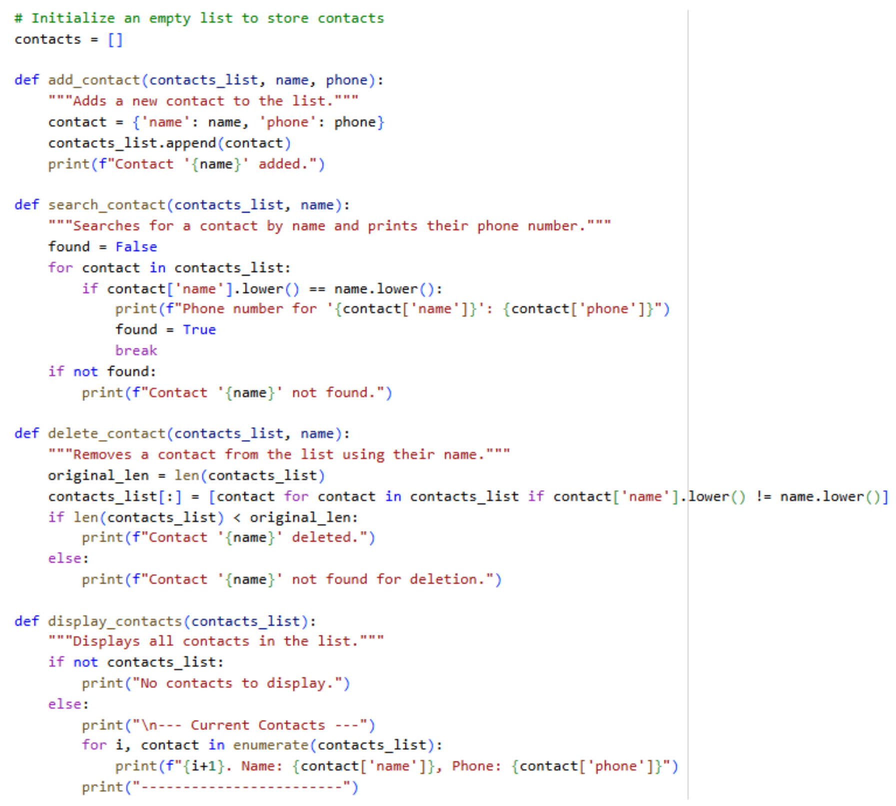
Task 1: Smart Contact Manager (Array & Linked List)

1A:Array Implementation

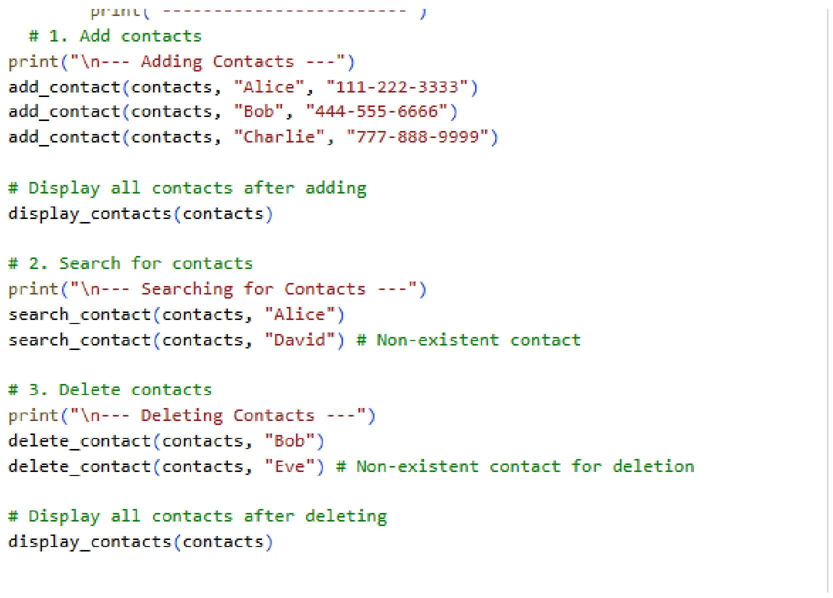
Prompt

1. Create a simple contact manager using Python list to store name and phone number together.
2. Write a function to add a new contact into the list structure.
3. Write a search function that finds contact by name and prints phone number.
4. Write a delete function that removes contact from the list using name.
5. Display all contacts after performing add, search and delete operations clearly.

Code:

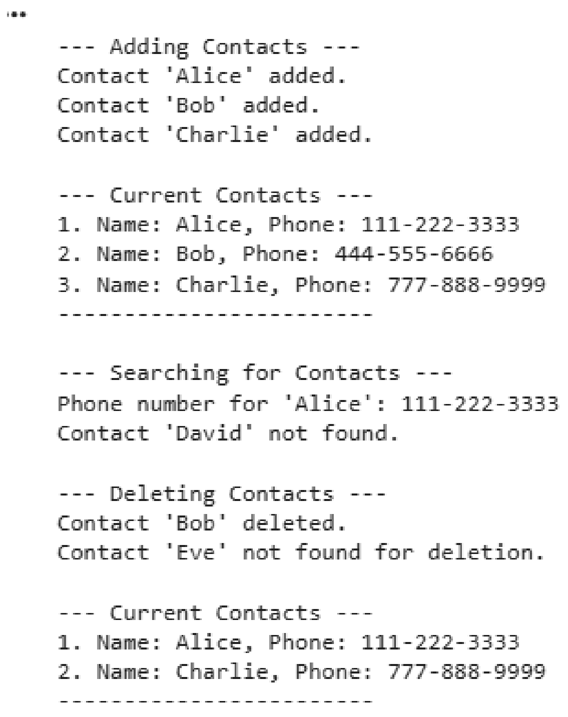


```python
# Initialize an empty list to store contacts
contacts = []

def add_contact(contacts_list, name, phone):
    """Adds a new contact to the list."""
    contact = {'name': name, 'phone': phone}
    contacts_list.append(contact)
    print(f"Contact '{name}' added.")

def search_contact(contacts_list, name):
    """Searches for a contact by name and prints their phone number."""
    found = False
    for contact in contacts_list:
        if contact['name'].lower() == name.lower():
            print(f"Phone number for '{contact['name']}': {contact['phone']}")
            found = True
            break
    if not found:
        print(f"Contact '{name}' not found.")

def delete_contact(contacts_list, name):
    """Removes a contact from the list using their name."""
    original_len = len(contacts_list)
    contacts_list[:] = [contact for contact in contacts_list if contact['name'].lower() != name.lower()]
    if len(contacts_list) < original_len:
        print(f"Contact '{name}' deleted.")
    else:
        print(f"Contact '{name}' not found for deletion.")

def display_contacts(contacts_list):
    """Displays all contacts in the list."""
    if not contacts_list:
        print("No contacts to display.")
    else:
        print("\n--- Current Contacts ---")
        for i, contact in enumerate(contacts_list):
            print(f"{i+1}. Name: {contact['name']}, Phone: {contact['phone']}")
        print("------------------------")
```



```
        print("------------------------")
  # 1. Add contacts
print("\n--- Adding Contacts ---")
add_contact(contacts, "Alice", "111-222-3333")
add_contact(contacts, "Bob", "444-555-6666")
add_contact(contacts, "Charlie", "777-888-9999")

# Display all contacts after adding
display_contacts(contacts)

# 2. Search for contacts
print("\n--- Searching for Contacts ---")
search_contact(contacts, "Alice")
search_contact(contacts, "David") # Non-existent contact

# 3. Delete contacts
print("\n--- Deleting Contacts ---")
delete_contact(contacts, "Bob")
delete_contact(contacts, "Eve") # Non-existent contact for deletion

# Display all contacts after deleting
display_contacts(contacts)
```

Output:



```
--- Adding Contacts ---
Contact 'Alice' added.
Contact 'Bob' added.
Contact 'Charlie' added.

--- Current Contacts ---
1. Name: Alice, Phone: 111-222-3333
2. Name: Bob, Phone: 444-555-6666
3. Name: Charlie, Phone: 777-888-9999
------------------------

--- Searching for Contacts ---
Phone number for 'Alice': 111-222-3333
Contact 'David' not found.

--- Deleting Contacts ---
Contact 'Bob' deleted.
Contact 'Eve' not found for deletion.

--- Current Contacts ---
1. Name: Alice, Phone: 111-222-3333
2. Name: Charlie, Phone: 777-888-9999
------------------------
```

Explanation:

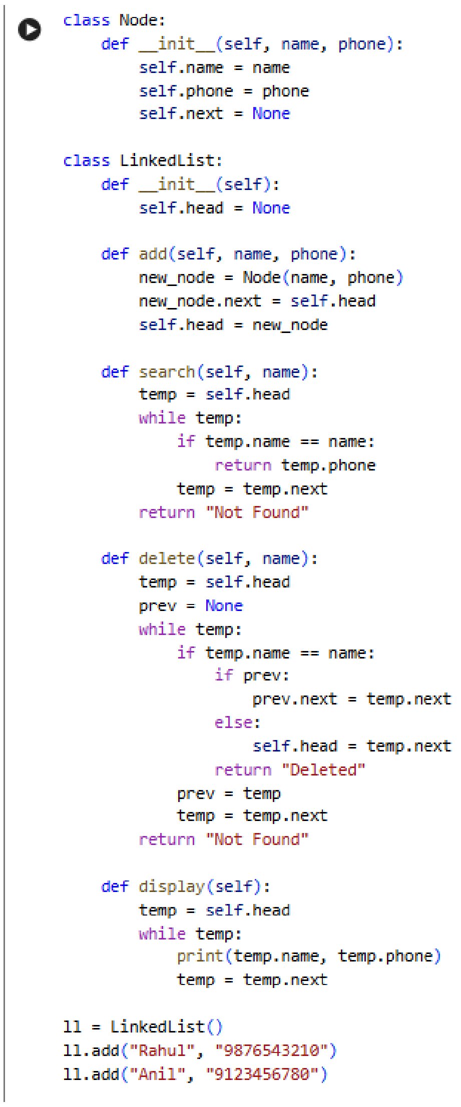
- We used Python list as array to store contacts.
- Adding contact is easy using append method.
- Searching and deleting requires checking each element one by one.
- Deletion in array is slower because elements need shifting.

1B. LINKED LIST IMPLEMENTATION

Prompt

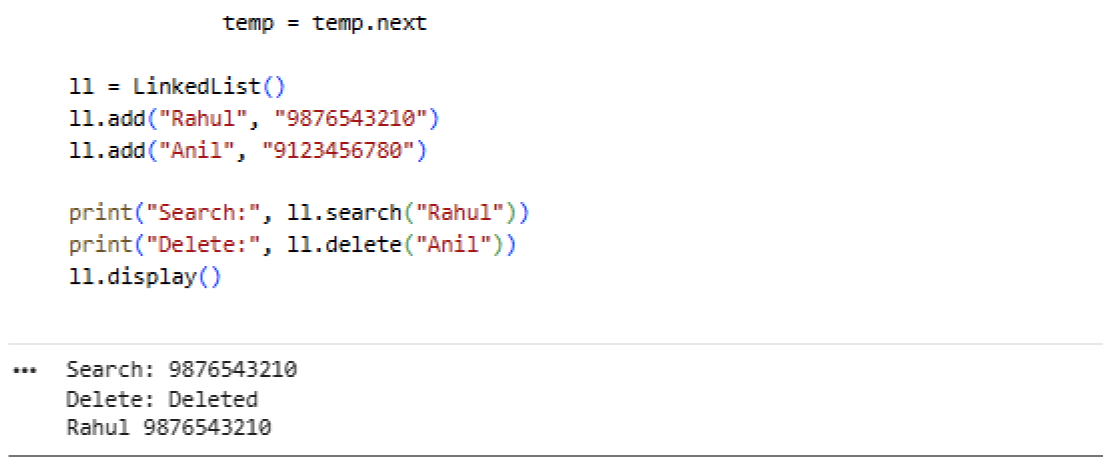
1. Create a linked list structure for storing contact details dynamically.
2. Define a Node class that stores name, phone and next pointer.
3. Implement add, search and delete operations using linked list logic.
4. Ensure memory is dynamically allocated when new contact is added.
5. Print all contacts to verify correct insertion and deletion.

Code:



```python
class Node:
    def __init__(self, name, phone):
        self.name = name
        self.phone = phone
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def add(self, name, phone):
        new_node = Node(name, phone)
        new_node.next = self.head
        self.head = new_node

    def search(self, name):
        temp = self.head
        while temp:
            if temp.name == name:
                return temp.phone
            temp = temp.next
        return "Not Found"

    def delete(self, name):
        temp = self.head
        prev = None
        while temp:
            if temp.name == name:
                if prev:
                    prev.next = temp.next
                else:
                    self.head = temp.next
                return "Deleted"
            prev = temp
            temp = temp.next
        return "Not Found"

    def display(self):
        temp = self.head
        while temp:
            print(temp.name, temp.phone)
            temp = temp.next

ll = LinkedList()
ll.add("Rahul", "9876543210")
ll.add("Anil", "9123456780")
```

Output:



```
            temp = temp.next

ll = LinkedList()
ll.add("Rahul", "9876543210")
ll.add("Anil", "9123456780")

print("Search:", ll.search("Rahul"))
print("Delete:", ll.delete("Anil"))
ll.display()
```

```
Search: 9876543210
Delete: Deleted
Rahul 9876543210
```

Explanation:

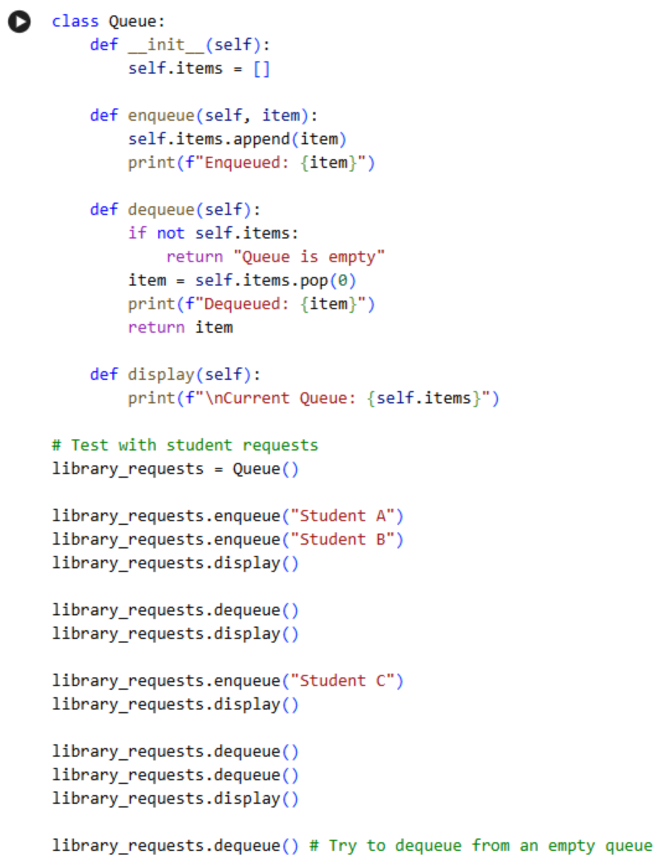
1. Linked list uses nodes connected by pointers.
2. Insertion is fast because no shifting is required.
3. Deletion is easier compared to array.
4. Memory is allocated dynamically.

Task 2: Library Book Search System (Queue & Priority Queue)

Prompt:

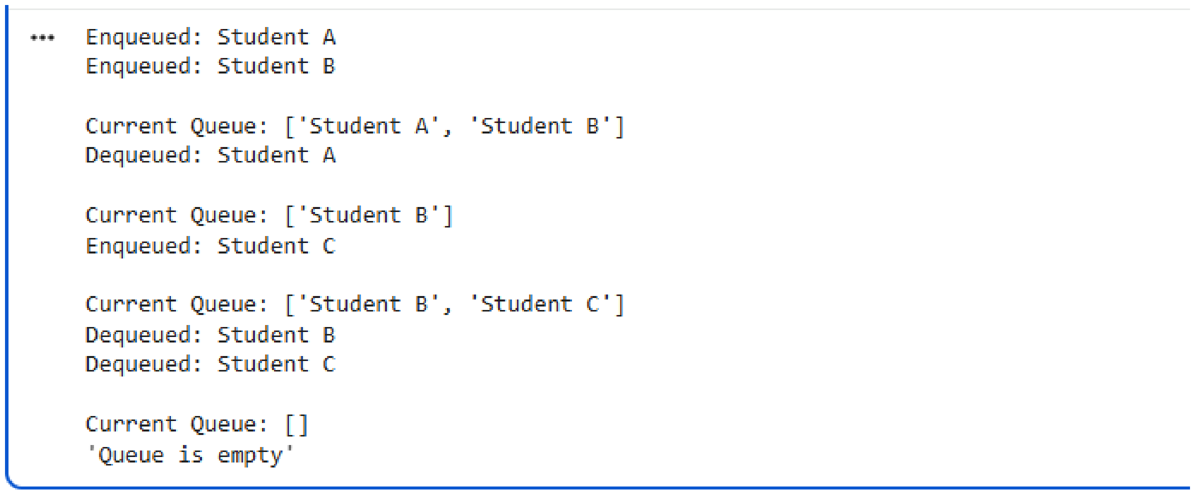
- Create a queue system to manage library book requests in FIFO order.
- Implement enqueue method to add request at rear side.
- Implement dequeue method to remove request from front side.
- Test with student requests and print processing order.Display queue behavior clearly after operations

Code:



```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        if not self.items:
            return "Queue is empty"
        item = self.items.pop(0)
        print(f"Dequeued: {item}")
        return item

    def display(self):
        print(f"\nCurrent Queue: {self.items}")

# Test with student requests
library_requests = Queue()

library_requests.enqueue("Student A")
library_requests.enqueue("Student B")
library_requests.display()

library_requests.dequeue()
library_requests.display()

library_requests.enqueue("Student C")
library_requests.display()

library_requests.dequeue()
library_requests.dequeue()
library_requests.display()

library_requests.dequeue() # Try to dequeue from an empty queue
```

Output:



```
Enqueued: Student A
Enqueued: Student B

Current Queue: ['Student A', 'Student B']
Dequeued: Student A

Current Queue: ['Student B']
Enqueued: Student C

Current Queue: ['Student B', 'Student C']
Dequeued: Student B
Dequeued: Student C

Current Queue: []
'Queue is empty'
```

Explanation:

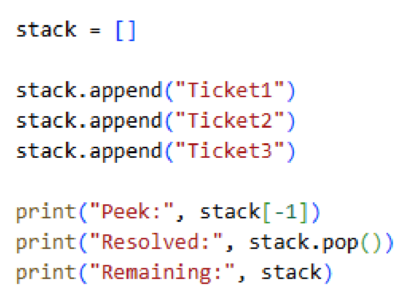
1. Queue follows First In First Out rule.
2. First student request is processed first.
3. append adds element at rear.
4. popleft removes element from front.

Task 3: Emergency Help Desk (Stack)

Prompt:

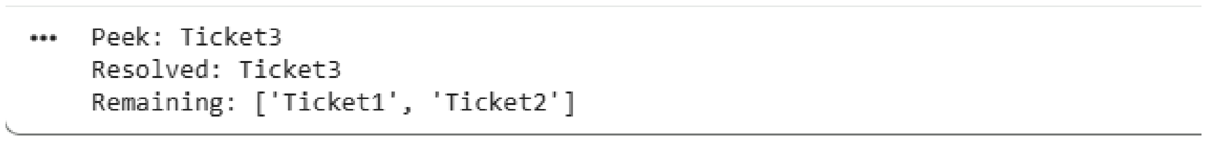
1. Create a stack to manage help desk tickets.
2. Implement push operation to add new ticket.
3. Implement pop operation to resolve latest ticket.
4. Implement peek operation to check current ticket.
5. Simulate five tickets and show LIFO behavior clearly.

Code:



```python
stack = []

stack.append("Ticket1")
stack.append("Ticket2")
stack.append("Ticket3")

print("Peek:", stack[-1])
print("Resolved:", stack.pop())
print("Remaining:", stack)
```

Output:



```
Peek: Ticket3
Resolved: Ticket3
Remaining: ['Ticket1', 'Ticket2']
```

Explanation:

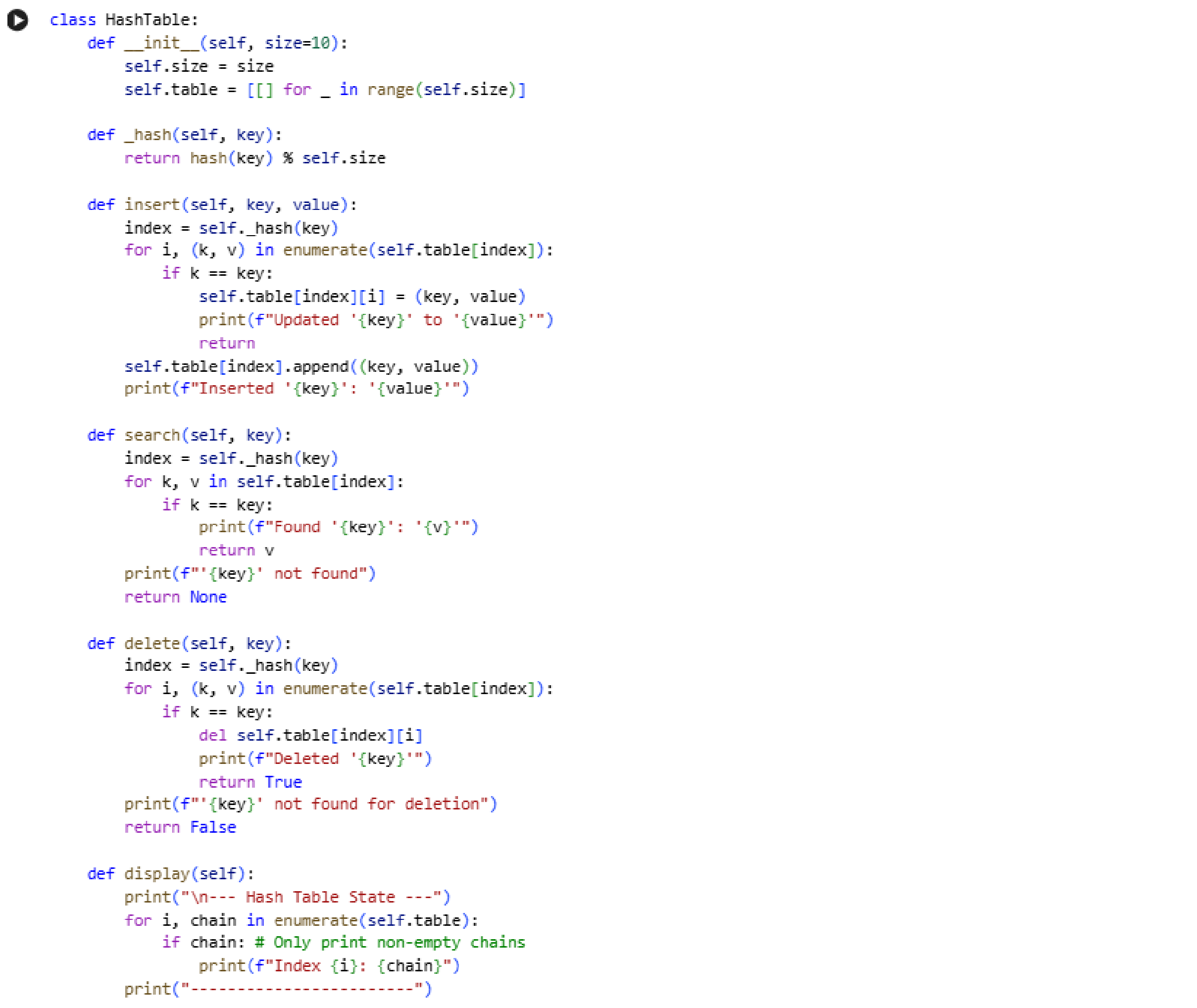
1. Stack follows Last In First Out rule.
2. Latest ticket is resolved first.
3. append is used as push operation.
4. pop removes last inserted element.

Task 4: Hash Table

Prompt

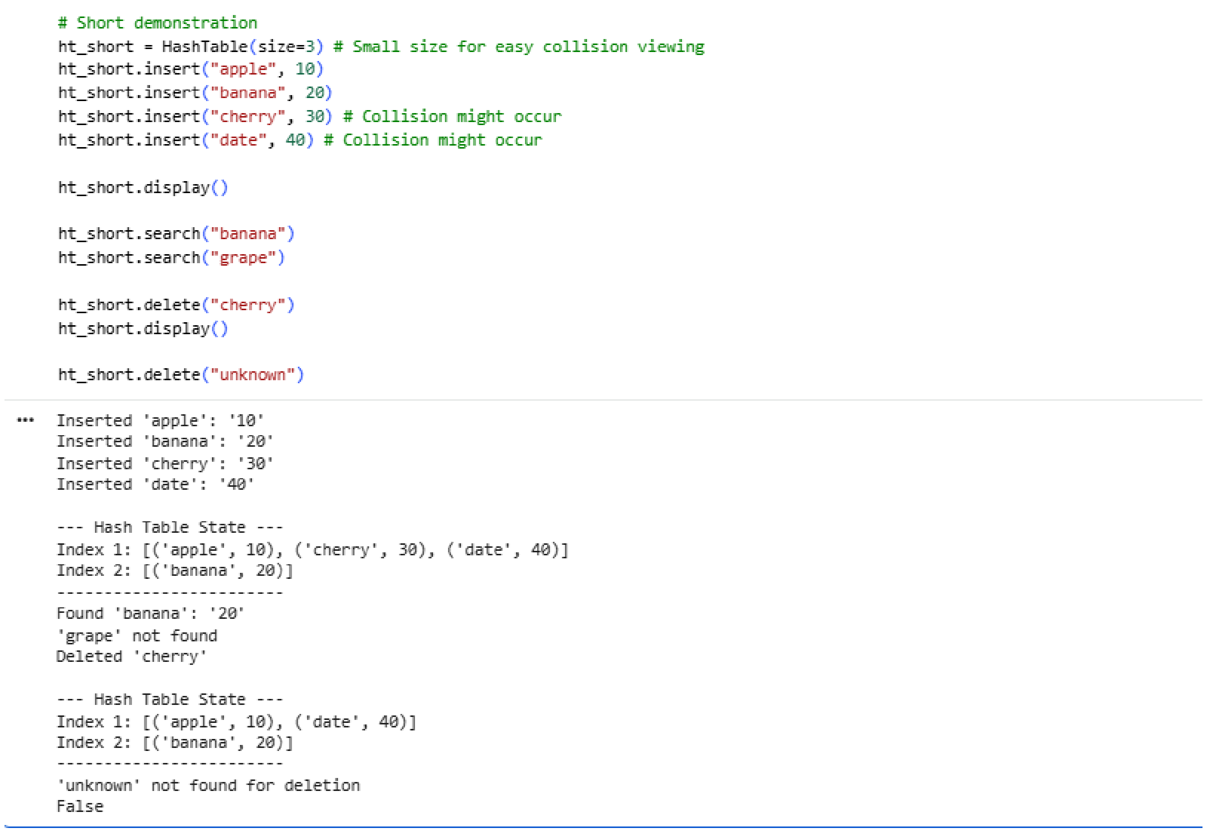
1. Create a hash table class using Python dictionary concept.
2. Implement insert method to store key value pairs.
3. Implement search method to find value using key.
4. Implement delete method to remove element safely.
5. Handle collision using chaining method.

Code:



```python
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                print(f"Updated '{key}' to '{value}'")
                return
        self.table[index].append((key, value))
        print(f"Inserted '{key}': '{value}'")

    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                print(f"Found '{key}': '{v}'")
                return v
        print(f"'{key}' not found")
        return None

    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                print(f"Deleted '{key}'")
                return True
        print(f"'{key}' not found for deletion")
        return False

    def display(self):
        print("\n--- Hash Table State ---")
        for i, chain in enumerate(self.table):
            if chain: # Only print non-empty chains
                print(f"Index {i}: {chain}")
        print("------------------------")
```

Output:



```
# Short demonstration
ht_short = HashTable(size=3) # Small size for easy collision viewing
ht_short.insert("apple", 10)
ht_short.insert("banana", 20)
ht_short.insert("cherry", 30) # Collision might occur
ht_short.insert("date", 40) # Collision might occur

ht_short.display()

ht_short.search("banana")
ht_short.search("grape")

ht_short.delete("cherry")
ht_short.display()

ht_short.delete("unknown")
```

```
Inserted 'apple': '10'
Inserted 'banana': '20'
Inserted 'cherry': '30'
Inserted 'date': '40'

--- Hash Table State ---
Index 1: [('apple', 10), ('cherry', 30), ('date', 40)]
Index 2: [('banana', 20)]
------------------------
Found 'banana': '20'
'grape' not found
Deleted 'cherry'

--- Hash Table State ---
Index 1: [('apple', 10), ('date', 40)]
Index 2: [('banana', 20)]
------------------------
'unknown' not found for deletion
False
```

Explanation:

1. Hash function converts key into index.
2. Collision handled using chaining method.
3. Insert stores data inside list at index.
4. Search and delete check inside that list.

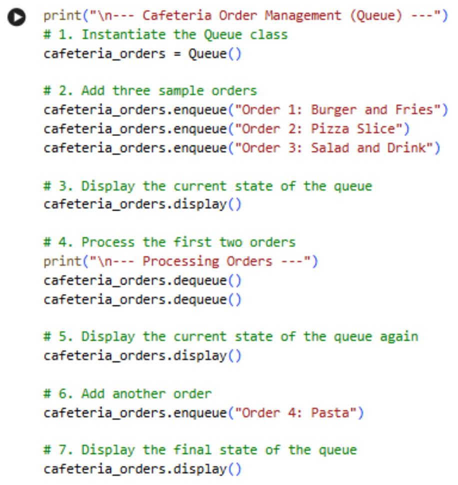
Task 5: Real-Time Application Challenge

Prompt :

1. Design a Campus Resource Management System using appropriate data structures for different real-time features.
2. Choose suitable data structures for attendance tracking, event registration, library borrowing, bus scheduling, and cafeteria order management.
3. Justify why each data structure is best suited for that specific feature in simple words.
4. Implement one selected feature using Python code with proper insert and retrieval operations.

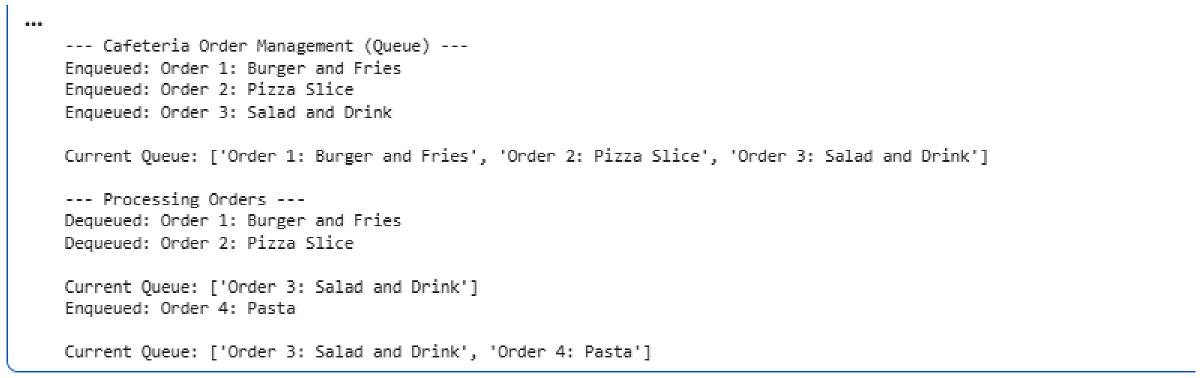
5. Display the output clearly and explain how the selected data structure improves efficiency.

Code:



```
print("\n--- Cafeteria Order Management (Queue) ---")
# 1. Instantiate the Queue class
cafeteria_orders = Queue()

# 2. Add three sample orders
cafeteria_orders.enqueue("Order 1: Burger and Fries")
cafeteria_orders.enqueue("Order 2: Pizza Slice")
cafeteria_orders.enqueue("Order 3: Salad and Drink")

# 3. Display the current state of the queue
cafeteria_orders.display()

# 4. Process the first two orders
print("\n--- Processing Orders ---")
cafeteria_orders.dequeue()
cafeteria_orders.dequeue()

# 5. Display the current state of the queue again
cafeteria_orders.display()

# 6. Add another order
cafeteria_orders.enqueue("Order 4: Pasta")

# 7. Display the final state of the queue
cafeteria_orders.display()
```

Output:



```
...
    --- Cafeteria Order Management (Queue) ---
    Enqueued: Order 1: Burger and Fries
    Enqueued: Order 2: Pizza Slice
    Enqueued: Order 3: Salad and Drink

    Current Queue: ['Order 1: Burger and Fries', 'Order 2: Pizza Slice', 'Order 3: Salad and Drink']

    --- Processing Orders ---
    Dequeued: Order 1: Burger and Fries
    Dequeued: Order 2: Pizza Slice

    Current Queue: ['Order 3: Salad and Drink']
    Enqueued: Order 4: Pasta

    Current Queue: ['Order 3: Salad and Drink', 'Order 4: Pasta']
```

Explanation:

1. Hash table gives fast lookup time.
2. Roll number works as unique key.
3. Attendance can be updated easily.
4. Searching student record is very fast.