

ASSIGNMENT-11.1

NAME: T. Bhavana Sri

H.TNO: 2303A52428

BATCH:36

Task Description #1 – Stack Implementation

Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

Sample Input Code:

```
class Stack:
```

```
    pass
```

Expected Output:

- A functional stack implementation with all required methods and docstrings.

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Untitled28.ipynb
- Toolbar:** File Edit View Insert Runtime Tools Help
- Code Cell:** Task 1
- Code Content:**

```
my_stack = Stack()
print("Is stack empty initially? ", my_stack.is_empty())
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)

print("Stack after pushes: ", my_stack)
print("Top element (peek): ", my_stack.peek())
print("Popped element: ", my_stack.pop())
print("Stack after pop: ", my_stack)
print("Length of stack: ", len(my_stack))
print("Is stack empty now? ", my_stack.is_empty())
```
- Output Cell:**

```
Is stack empty initially? True
Stack after pushes: Stack([10, 20, 30])
Top element (peek): 30
Popped element: 30
Stack after pop: Stack([10, 20])
Length of stack: 2
Is stack empty now? False
```

EXPLANATION:

A stack follows the principle LIFO (Last In, First Out).

The last element inserted is the first one removed.

We use a Python list (self.items) to store elements.

Task Description #2 – Queue Implementation

Task: Use AI to implement a Queue using Python lists.

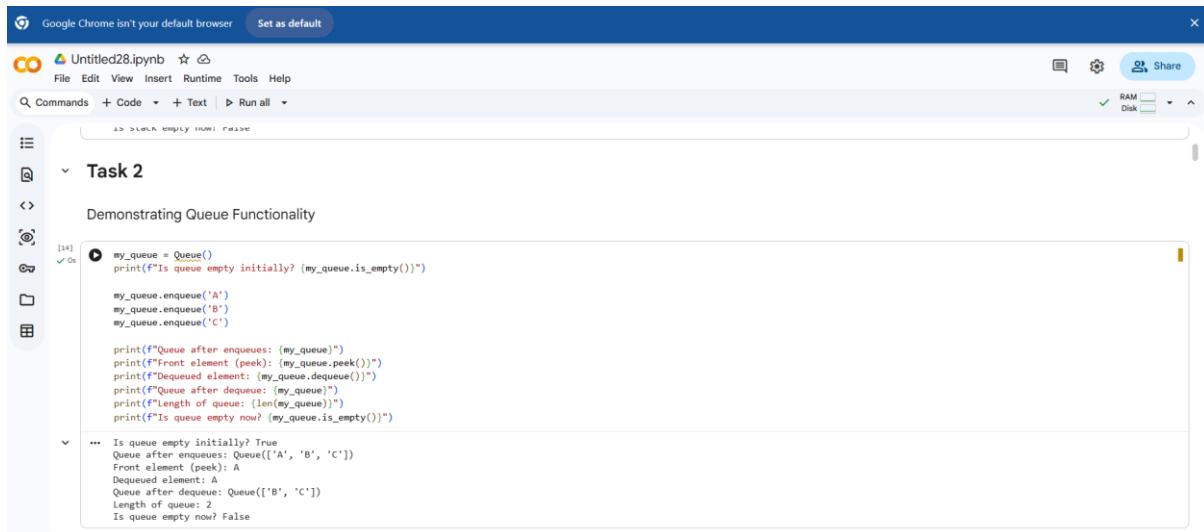
Sample Input Code:

```
class Queue:
```

```
    pass
```

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.



The screenshot shows a Jupyter Notebook interface in Google Colab. The code cell contains Python code for a Queue class. The output cell shows the execution results, demonstrating the enqueue, dequeue, peek, and is_empty methods. The output is as follows:

```
[14]: my_queue = Queue()
print(f"Is queue empty initially? {my_queue.is_empty()}")
my_queue.enqueue('A')
my_queue.enqueue('B')
my_queue.enqueue('C')

print(f"Queue after enqueues: {my_queue}")
print(f"Front element (peek): {my_queue.peek()}")
print(f"Dequeued element: {my_queue.dequeue()}")
print(f"Queue after dequeue: {my_queue}")
print(f"Length of queue: {len(my_queue)}")
print(f"Is queue empty now? {my_queue.is_empty()}")

...
Is queue empty initially? True
Queue after enqueues: Queue(['A', 'B', 'C'])
Front element (peek): A
Dequeued element: A
Queue after dequeue: Queue(['B', 'C'])
Length of queue: 2
Is queue empty now? False
```

EXPLANATION:

A queue works on the principle FIFO (First In, First Out).

The first element inserted is the first one removed.

We use a Python list called self.items to store elements.

Task Description #3 – Linked List

Task: Use AI to generate a Singly Linked List with insert and display methods.

Sample Input Code:

```
class Node:
```

```
    pass
```

```
class LinkedList:
```

```
    pass
```

Expected Output:

- A working linked list implementation with clear method

Documentation.

```

[18] In [18]
my_linked_list = LinkedList()
print("Inserting at beginning:")
my_linked_list.insert_at_beginning(30)
my_linked_list.insert_at_beginning(20)
my_linked_list.insert_at_beginning(10)
my_linked_list.display()

print("Inserting at end:")
my_linked_list.insert_at_end(40)
my_linked_list.insert_at_end(50)
my_linked_list.display()

print(f"Linked List representation: {my_linked_list}")

```

```

Inserting at beginning:
10 -> 20 -> 30
Inserting at end:
10 -> 20 -> 30 -> 40 -> 50
Linked List representation: LinkedList([10, 20, 30, 40, 50])

```

EXPLANATION (Linked List):

A singly linked list is a collection of nodes where:

Each node stores data

And a reference to the next node

■ Insert(data)

Creates a new node

Traverses to the last node

Task Description #4 – Binary Search Tree (BST)

Task: Use AI to create a BST with insert and in-order traversal methods.

Sample Input Code:

class BST:

 pass

Expected Output:

- BST implementation with recursive insert and traversal methods.

```

10 -> 20 -> 30 -> 40 -> 50
Linked List representation: LinkedList([10, 20, 30, 40, 50])

TASK 4

Demonstrating Binary Search Tree (BST) Functionality

my_bst = BST()
# Insert elements
my_bst.insert(50)
my_bst.insert(30)
my_bst.insert(70)
my_bst.insert(20)
my_bst.insert(40)
my_bst.insert(60)
my_bst.insert(80)

print("BST in-order traversal: (my_bst.in_order_traversal())")
print("BST representation: (my_bst)")

BST in-order traversal: [20, 30, 40, 50, 60, 70, 80]
BST representation: BST([20, 30, 40, 50, 60, 70, 80])

```

EXPLANATION:

A Binary Search Tree (BST) stores values such that:

Left child contains smaller values

Right child contains larger values

Insert (recursive):

If the tree is empty, the first value becomes the root.

Otherwise, compare the value with the current node:

Go left if smaller

Go right if larger

Repeat this process recursively until an empty spot is found and insert there.

In-order traversal:

Visits nodes in the order: Left → Root → Right. This prints the elements in sorted order. So, the BST keeps data ordered and allows efficient insertion and sorted traversal.

Task Description #5 – Hash Table

Task: Use AI to implement a hash table with basic insert, search, and delete methods.

Sample Input Code:

```
class HashTable:
```

```
    pass
```

Expected Output:

- Collision handling using chaining, with well-commented methods.

```

class HashTable:
    """A simple Hash Table implementation using chaining for collision resolution."""

    def __init__(self, size=10):
        """Initializes the hash table with a given size. Each slot is an empty list (chain)."""
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash(self, key):
        """Generates a hash index for a given key using the modulo operator."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated.
        """
        index = self._hash(key)
        # Iterate through the chain to check if the key already exists
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                # Key found, update the value
                self.table[index][i] = (key, value)
                return
        # Key not found, add the new key-value pair to the chain
        self.table[index].append((key, value))

```

```

def search(self, key):
    """Searches for a key in the hash table and returns its value.
    Returns None if the key is not found.
    """
    index = self._hash(key)
    # Iterate through the chain to find the key
    for k, v in self.table[index]:
        if k == key:
            return v # Key found, return its value
    return None # Key not found

def delete(self, key):
    """Deletes a key-value pair from the hash table.
    Does nothing if the key is not found.
    """
    index = self._hash(key)
    # Use a list comprehension to rebuild the chain without the key to be deleted
    # This effectively removes the item from the chain
    self.table[index] = [(k, v) for k, v in self.table[index] if k != key]

def __repr__(self):
    """Returns a string representation of the hash table."""
    items = []
    for i, chain in enumerate(self.table):
        if chain:
            items.append(f"Slot {i}: {chain}")
    return "\n".join(items)

def __str__(self):
    """Returns a user-friendly string representation of the hash table."""
    return self.__repr__()

```

EXPLANATION:

A hash table stores data using a key-value pair.

A hash function converts the key into an index.

Task Description #6 – Graph Representation

Task: Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

Expected Output:

- Graph with methods to add vertices, add edges, and display Connections.

The screenshot shows a Google Colab notebook titled "Untitled28.ipynb". The code cell contains Python code for creating a graph and adding vertices and edges. The output shows the graph representation as an adjacency list.

```

my_graph = Graph()
print("Adding vertices and undirected edges:")
my_graph.add_edge('A', 'B', weight=10)
my_graph.add_edge('B', 'C', weight=5)
my_graph.add_edge('C', 'A', weight=15)
my_graph.add_vertex('D') # Add a disconnected vertex
my_graph.add_edge('B', 'D', weight=7)

my_graph.display_connections()

print("\nAdding a directed edge:")
my_graph.add_edge('D', 'E', weight=3, directed=True)
my_graph.display_connections()

print(f"\nGraph representation: {my_graph}")

```

Graph representation:

```

Graph representation: Graph([('A': [('B', 10), ('C', 15)], 'B': [('A', 10), ('C', 5)], 'C': [('A', 15)], 'D': []}, directed=False)

```

EXPLANATION:

A graph consists of:

Vertices (nodes)

Edges (connections)

Using an adjacency list, each vertex stores a list of its connected vertices.

Task Description #7 – Priority Queue

Task: Use AI to implement a priority queue using Python's heapq module.

Sample Input Code:

```
class PriorityQueue:
```

```
    pass
```

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

```

my_pq = PriorityQueue()
print(f"Is Priority Queue empty initially? {my_pq.is_empty()}")
print("Enqueuing items with priorities:")
my_pq.enqueue("Task A", 3)
my_pq.enqueue("Task B", 1)
my_pq.enqueue("Task C", 2)
my_pq.enqueue("Task D", 1) # Same priority as Task B
my_pq.display()
print(f"Length of Priority Queue: {len(my_pq)}")

print("\nDequeueing highest priority items:")
print(f"Dequeued: {my_pq.dequeue()}") # Should be Task B or D (due to tie-breaking)
my_pq.display()
print(f"Length of Priority Queue: {len(my_pq)}")

print(f"Dequeued: {my_pq.dequeue()}") # Should be the other one with priority 1
my_pq.display()
print(f"Length of Priority Queue: {len(my_pq)}")

print(f"Dequeued: {my_pq.dequeue()}") # Should be Task C
my_pq.display()
print(f"Length of Priority Queue: {len(my_pq)}")

```

```

Length of Priority Queue: 4
Dequeued: (1, Task B) # Should be Task C
Length of Priority Queue: 3
Is Priority Queue empty now? {my_pq.is_empty()}
Priority Queue representation: (my_pq)

... Is Priority Queue empty initially? True
Enqueuing items with priorities:
Priority Queue (priority, item):
(1, Task B)
(1, Task D)
(2, Task C)
(3, Task A)
Length of Priority Queue: 4

Dequeueing highest priority items:
Dequeued: Task B
Priority Queue (priority, item):
(1, Task D)
(2, Task C)
(3, Task A)
Length of Priority Queue: 3
Dequeued: Task D
Priority Queue (priority, item):
(2, Task C)
(3, Task A)
Length of Priority Queue: 2
Dequeued: Task C
Priority Queue (priority, item):
(3, Task A)
Length of Priority Queue: 1
Is Priority Queue empty now? False
Priority Queue representation: PriorityQueue([(3, 'Task A')])
```

EXPLANATION:

A priority queue stores elements with a priority value.

The element with the highest priority is removed first.

Python's heapq provides a min-heap, so to get highest priority first, we:

Store priority as negative value.

This makes larger priorities come out first.

Task Description #8 – Deque

Task: Use AI to implement a double-ended queue using collections.deque.

Sample Input Code:

```
class DequeDS:
```

```
    pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

The screenshot shows a Jupyter Notebook cell with the code for a DequeDS class. The code adds items to the front and rear, prints the deque after each addition, removes items from both front and rear, and prints the deque after each removal. It also checks if the deque is empty at various points. The output shows the state of the deque after each operation.

```
[28] In [28]
my_deque = DequeDS()
print("Is deque empty initially? ", my_deque.is_empty())
print("\nAdding items to front and rear:")
my_deque.add_rear('B')
my_deque.add_front('A')
my_deque.add_rear('C')
my_deque.add_front('E')
print("Deque after additions: ", my_deque)
print("Front element (peek_front): ", my_deque.peek_front())
print("Rear element (peek_rear): ", my_deque.peek_rear())
print("Length of deque: ", len(my_deque))

print("\nRemoving items from front and rear:")
print("Removed from front: ", my_deque.remove_front())
print("Deque after remove_front: ", my_deque)

print("Removed from rear: ", my_deque.remove_rear())
print("Deque after remove_rear: ", my_deque)

print("Length of deque: ", len(my_deque))
print("Is deque empty now? ", my_deque.is_empty())

print("\nFinal Deque representation: ", my_deque)
```

The screenshot shows the execution of the deque code. The code adds items to the deque, removes them, and prints the final state. The output shows the deque's state after each operation, including its length and the elements at the front and rear.

```
[29] In [29]
print("Length of deque: ", len(my_deque))

print("\nRemoving items from front and rear:")
print("Removed from front: ", my_deque.remove_front())
print("Deque after remove_front: ", my_deque)

print("Removed from rear: ", my_deque.remove_rear())
print("Deque after remove_rear: ", my_deque)

print("Length of deque: ", len(my_deque))
print("Is deque empty now? ", my_deque.is_empty())

print("\nFinal Deque representation: ", my_deque)

... Is deque empty initially? True
Adding items to front and rear:
Deque after additions: DequeDS(['E', 'A', 'B', 'C'])
Front element (peek_front): E
Rear element (peek_rear): C
Length of deque: 4

Removing items from front and rear:
Removed from front: 
Deque after remove_front: DequeDS(['A', 'B', 'C'])
Removed from rear: C
Deque after remove_rear: DequeDS(['A', 'B'])
Length of deque: 2
Is deque empty now? False

Final Deque representation: DequeDS(['A', 'B'])
```

EXPLANATION:

A Deque (Double-Ended Queue) allows:

Insertion at front and rear

Deletion from front and rear

It is more flexible than a normal queue.

Using collections.deque:

`appendleft()` → insert front

`append()` → insert rear

`popleft()` → remove front

`pop()` → remove rear

All operations are $O(1)$ (efficient).

Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

Scenario:

Your college wants to develop a Campus Resource Management System that handles:

1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
 - o Stack
 - o Queue
 - o Priority Queue
 - o Linked List
 - o Binary Search Tree (BST)
 - o Graph
 - o Hash Table

o Deque

- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

I'll start by generating the table for **Feature-Data Structure Mapping and Justification**.

```
%write_and_execute_cell
# Feature-Data Structure Mapping and Justification

print("### Campus Feature to Data Structure Mapping")
print("Below is a table outlining common campus features, the most suitable data structure for each, and a brief justification for why it's appropriate:")
print("\n| Campus Feature | Data Structure | Justification")
print("-----|-----|-----")
print("| Cafeteria Order Queue | Queue | Orders are processed in a First-In, First-Out (FIFO) manner. A Queue is suitable for this because it maintains the chronological order of items and allows for efficient insertion and removal from both ends." )
print("| Student Management System | Hash Table | Enables quick insertion, deletion, and retrieval of student records. A Hash Table provides constant-time complexity for these operations, making it ideal for managing large datasets of student information." )
print("| Course Enrollment System | Graph | Course prerequisites and dependencies can be effectively modeled as a directed graph. A Graph allows for complex relationships between courses and tracks how changes in one course affect others." )
print("| Campus Navigation (Shortest Path) | Graph | The campus layout (buildings, paths) can be represented as a graph. A Graph is useful for finding the shortest path between two points based on weighted edges (e.g., distance or time).")
print("| Book Lending System (Library) | Binary Search Tree | Books can be stored and retrieved efficiently by unique identifier. A Binary Search Tree provides logarithmic time complexity for search operations, making it suitable for a library system where books are identified by ISBN numbers." )
print("| Event Scheduling (Priority) | Priority Queue | Events often have different levels of urgency or importance. A Priority Queue ensures that the most critical events are handled first, maintaining their relative priority throughout the process." )
print("| Undo/Redo Functionality | Stack | Actions are added and removed in a Last-In, First-Out (LIFO) manner. A Stack is appropriate for this because it follows the principle of last-in, first-out, which is essential for undo and redo operations." )
print("| Recent Activity Feed | Deque | New activities are added to one end, and older activities can be easily removed from the other end. A Deque is suitable for this because it supports efficient insertion and deletion from both ends." )
```

```
[26] ✓ 0s
class CafeteriaOrderQueue:
    """Simulates a cafeteria order queue using the Queue data structure."""

    def __init__(self):
        self.order_queue = Queue()
        print("Cafeteria Order Queue initialized.")

    def place_order(self, order_details):
        """Adds a new order to the queue."""
        print(f"Placing order: {order_details}")
        self.order_queue.enqueue(order_details)
        self.display_queue_status()

    def fulfill_order(self):
        """Removes and returns the next order to be fulfilled."""
        if not self.order_queue.is_empty():
            fulfilled_order = self.order_queue.dequeue()
            print(f"Fulfilling order: {fulfilled_order}")
            self.display_queue_status()
            return fulfilled_order
        else:
            print("No orders in the queue to fulfill.")
            return None

    def display_queue_status(self):
        """Prints the current status of the order queue."""
        print(f"Current orders in queue (length {len(self.order_queue)}): {self.order_queue}")
        if not self.order_queue.is_empty():
            print(f"Next order to be fulfilled: {self.order_queue.peek()}")
        print("-----")

# --- Simulation of Cafeteria Order Process ---
print("\n--- Simulating Cafeteria Order Process ---")
my_cafeteria = CafeteriaOrderQueue()
```

```

my_cafeteria = CafeteriaOrderQueue()

# Students placing orders
my_cafeteria.place_order("Student A - Pizza with Coke")
my_cafeteria.place_order("Student B - Burger with Fries")
my_cafeteria.place_order("Student C - Salad with Water")
my_cafeteria.place_order("Student D - Pasta")

# Cafeteria fulfilling orders
print("\n--- Fulfilling Orders ---")
my_cafeteria.fulfill_order()
my_cafeteria.fulfill_order()

# Another student places an order while others are being served
my_cafeteria.place_order("Student E - Sandwich")

my_cafeteria.fulfill_order()
my_cafeteria.fulfill_order()
my_cafeteria.fulfill_order()

# Try to fulfill an order when the queue is empty
my_cafeteria.fulfill_order()

*** --- Simulating Cafeteria Order Process ---
Cafeteria Order Queue initialized.
Placing order: Student A - Pizza with Coke
Current orders in queue (length 1): Queue(['Student A - Pizza with Coke'])
Next order to be fulfilled: Student A - Pizza with Coke
-----
Placing order: Student B - Burger with Fries
Current orders in queue (length 2): Queue(['Student A - Pizza with Coke', 'Student B - Burger with Fries'])
Next order to be fulfilled: Student A - Pizza with Coke
-----
Placing order: Student C - Salad with Water

```

```

Placing order: Student B - Burger with Fries
Current orders in queue (length 2): Queue(['Student A - Pizza with Coke', 'Student B - Burger with Fries'])
Next order to be fulfilled: Student A - Pizza with Coke
-----
Placing order: Student C - Salad with Water
Current orders in queue (length 3): Queue(['Student A - Pizza with Coke', 'Student B - Burger with Fries', 'Student C - Salad with Water'])
Next order to be fulfilled: Student A - Pizza with Coke
-----
Placing order: Student D - Pasta
Current orders in queue (length 4): Queue(['Student A - Pizza with Coke', 'Student B - Burger with Fries', 'Student C - Salad with Water', 'Student D - Pasta'])
Next order to be fulfilled: Student A - Pizza with Coke
-----
--- Fulfilling Orders ---
Fulfilling order: Student A - Pizza with Coke
Current orders in queue (length 3): Queue(['Student B - Burger with Fries', 'Student C - Salad with Water', 'Student D - Pasta'])
Next order to be fulfilled: Student B - Burger with Fries
-----
Fulfilling order: Student B - Burger with Fries
Current orders in queue (length 2): Queue(['Student C - Salad with Water', 'Student D - Pasta'])
Next order to be fulfilled: Student C - Salad with Water
-----
Placing order: Student E - Sandwich
Current orders in queue (length 3): Queue(['Student C - Salad with Water', 'Student D - Pasta', 'Student E - Sandwich'])
Next order to be fulfilled: Student C - Salad with Water
-----
Fulfilling order: Student C - Salad with Water
Current orders in queue (length 2): Queue(['Student D - Pasta', 'Student E - Sandwich'])
Next order to be fulfilled: Student D - Pasta
-----
Fulfilling order: Student D - Pasta
Current orders in queue (length 1): Queue(['Student E - Sandwich'])
Next order to be fulfilled: Student E - Sandwich
-----
Fulfilling order: Student E - Sandwich
Current orders in queue (length 0): Queue([])
-----
No orders in the queue to fulfill.

```

EXPLANATION:

The cafeteria system uses a Queue because it follows the First In, First Out (FIFO) rule.

Students who arrive first are served first.

The program supports adding orders, serving orders, and displaying pending orders.

Task Description #10: Smart E-Commerce Platform – Data Structure

Challenge

An e-commerce company wants to build a Smart Online Shopping System with:

1. Shopping Cart Management – Add and remove products

dynamically.

2. Order Processing System – Orders processed in the order they are placed.
3. Top-Selling Products Tracker – Products ranked by sales count.
4. Product Search Engine – Fast lookup of products using product ID.
5. Delivery Route Planning – Connect warehouses and delivery locations.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
 - o Stack
 - o Queue
 - o Priority Queue
 - o Linked List
 - o Binary Search Tree (BST)
 - o Graph
 - o Hash Table
 - o Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Google Chrome isn't your default browser Set as default

Untitled28.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text | Run all

```
[27] 0s
class OrderProcessingSystem:
    """Simulates an order processing system using the Queue data structure."""

    def __init__(self):
        self.order_queue = Queue()
        print("Order Processing System initialized.")

    def place_order(self, order_details):
        """Adds a new order to the queue."""
        print(f"Placing order: {order_details}")
        self.order_queue.enqueue(order_details)
        self.display_queue_status()

    def process_order(self):
        """Removes and returns the next order to be processed."""
        if not self.order_queue.is_empty():
            processed_order = self.order_queue.dequeue()
            print(f"Processing order: {processed_order}")
            self.display_queue_status()
            return processed_order
        else:
            print("No orders in the queue to process.")
            return None

    def display_queue_status(self):
        """Prints the current status of the order queue."""
        print(f"Current orders in queue (length {len(self.order_queue)}): {self.order_queue}")
        if not self.order_queue.is_empty():
            print(f"Next order to be processed: {self.order_queue.peek()}")
            print("-----")
        print("OrderProcessingSystem class defined.")

... OrderProcessingSystem class defined.
```

Untitled28.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text | Run all

```
[28] 0s
print("\n--- Simulating Order Processing System ---")
my_order_system = OrderProcessingSystem()

# Simulate customers placing orders
my_order_system.place_order("Order #001: Laptop, Mouse")
my_order_system.place_order("Order #002: Keyboard, Monitor")
my_order_system.place_order("Order #003: Webcam, Microphone")

# Simulate processing some orders
print("\n--- Processing Orders ---")
my_order_system.process_order()
my_order_system.process_order()

# Another order comes in while others are being processed
my_order_system.place_order("Order #004: USB Hub")

my_order_system.process_order()
my_order_system.process_order()

# Attempt to process an order when the queue is empty
my_order_system.process_order()

... --- Simulating Order Processing System ---
Order Processing System initialized.
Placing order: Order #001: Laptop, Mouse
Current orders in queue (length 1): Queue(['Order #001: Laptop, Mouse'])
Next order to be processed: Order #001: Laptop, Mouse
-----
Placing order: Order #002: Keyboard, Monitor
```

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Untitled28.ipynb
- Menu Bar:** File, Edit, View, Insert, Runtime, Tools, Help
- Toolbar:** Commands, + Code, + Text, Run all
- Code Output:**

```
Current orders in queue (length 1): Queue(['Order #001: Laptop, Mouse'])
Next order to be processed: Order #001: Laptop, Mouse
...
Placing order: Order #002: Keyboard, Monitor
Current orders in queue (length 2): Queue(['Order #001: Laptop, Mouse', 'Order #002: Keyboard, Monitor'])
Next order to be processed: Order #001: Laptop, Mouse
-----
Placing order: Order #003: Webcam, Microphone
Current orders in queue (length 3): Queue(['Order #001: Laptop, Mouse', 'Order #002: Keyboard, Monitor', 'Order #003: Webcam, Microphone'])
Next order to be processed: Order #001: Laptop, Mouse
-----
--- Processing Orders ---
Processing order: Order #001: Laptop, Mouse
Current orders in queue (length 2): Queue(['Order #002: Keyboard, Monitor', 'Order #003: Webcam, Microphone'])
Next order to be processed: Order #002: Keyboard, Monitor
-----
Processing order: Order #002: Keyboard, Monitor
Current orders in queue (length 1): Queue(['Order #003: Webcam, Microphone'])
Next order to be processed: Order #003: Webcam, Microphone
-----
Placing order: Order #004: USB Hub
Current orders in queue (length 2): Queue(['Order #003: Webcam, Microphone', 'Order #004: USB Hub'])
Next order to be processed: Order #003: Webcam, Microphone
-----
Processing order: Order #003: Webcam, Microphone
Current orders in queue (length 1): Queue(['Order #004: USB Hub'])
Next order to be processed: Order #004: USB Hub
-----
Processing order: Order #004: USB Hub
Current orders in queue (length 0): Queue([])
-----
No orders in the queue to process.
```

EXPLANATION:

The product search system uses a Hash Table because product IDs can be used as keys for instant lookup.

Insertion, search, and deletion operations are very fast (average O(1)).

This makes it suitable for large-scale e-commerce platforms with thousands of products.