**LAB ASSIGNMENT 11**

**NAME: Ramakrishna Vemula**

**ROLL NO**:2403A54037

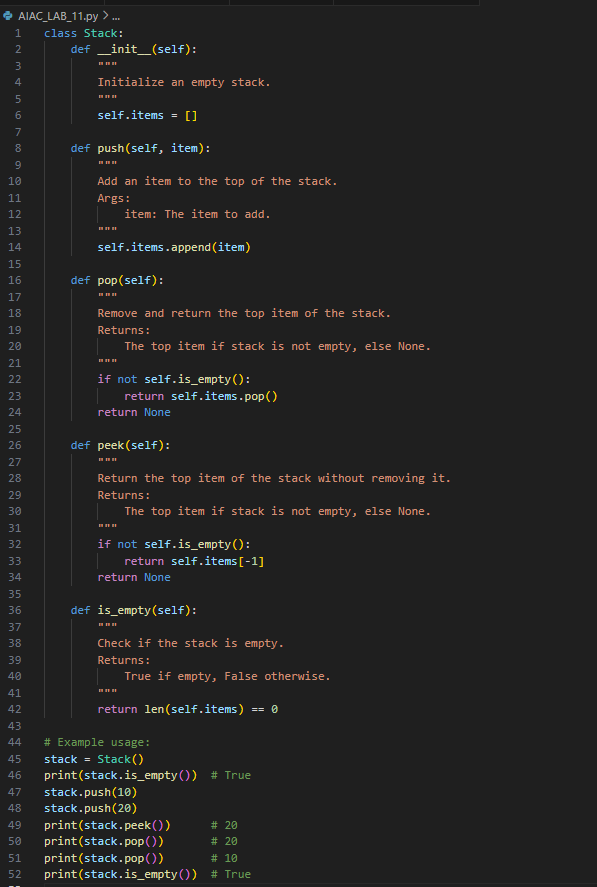
**BATCH:**CSE AIML 15

**lab 11** – Data Structures with AI: Implementing Fundamental Structures  
Lab Objectives  
• Use AI to assist in designing and implementing fundamental data  
structures in Python.  
• Learn how to prompt AI for structure creation, optimization, and  
documentation.  
• Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees,  
Graphs, and Hash Tables.  
Week6 -  
Monday

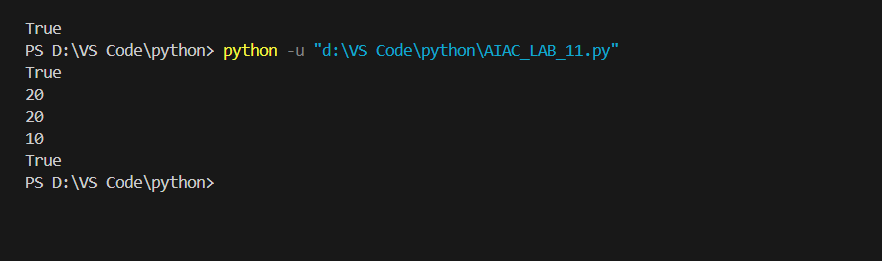
• Enhance code quality with AI-generated comments and performance  
suggestions.

**Task1**: Use AI to generate a Stack class with push, pop, peek, and is\_empty  
methods.

**PROMPT:** generate a Stack class with push, pop, peek, and is\_empty  
methods.

**code**: 

**Output**:



**OBSERVATION**:

* The stack uses a Python list to store items.
* push adds an item to the top.
* [pop](vscode-file://vscode-app/c:/Users/Syed%20Nabeel%20Qanith/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) removes and returns the top item, or returns None if empty.
* peek returns the top item without removing it, or None if empty.
* is\_empty checks if the stack is empty.
* The example demonstrates all operations and confirms correct stack behavior.

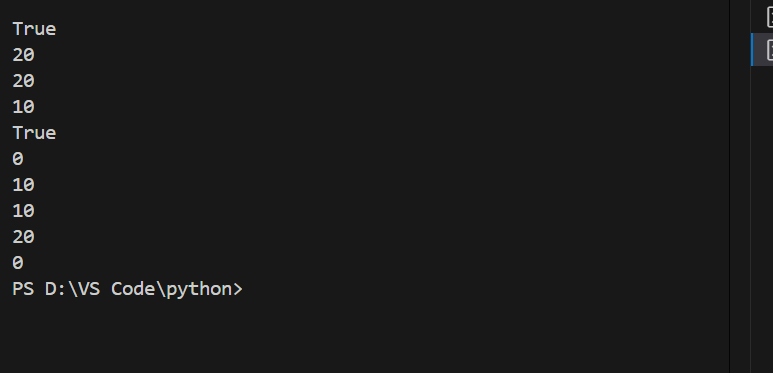
**Task2:** Use AI to implement a Queue using Python lists.  
Sample Input Code:  
class Queue:  
pass  
Expected Output:  
• FIFO-based queue class with enqueue, dequeue, peek, and size  
methods.

**PROMPT:** Implement a • FIFO-based queue class with enqueue, dequeue, peek, and size  
methods

**CODE:**

****

**Output:**

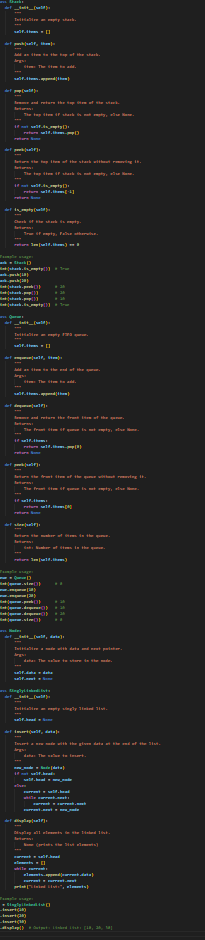
****

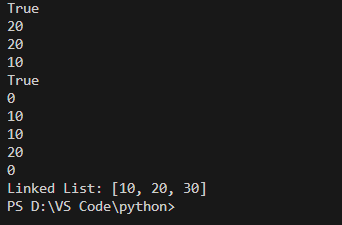
* **OBSERVATION:** The queue uses a Python list to store items, maintaining FIFO (First-In-First-Out) order.
* enqueue adds items to the end of the queue.
* dequeue removes and returns the front item; returns None if the queue is empty.
* peek returns the front item without removing it; returns None if the queue is empty.
* size returns the current number of items in the queue.
* The example usage demonstrates all operations and confirms correct FIFO behavior.

Task Description #3 – Linked List  
**Task3**: 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.

**PROMPT:**Generate a Singly Linked List with insert and display methods A working linked list implementation with clear method  
documentation

**CODE:**



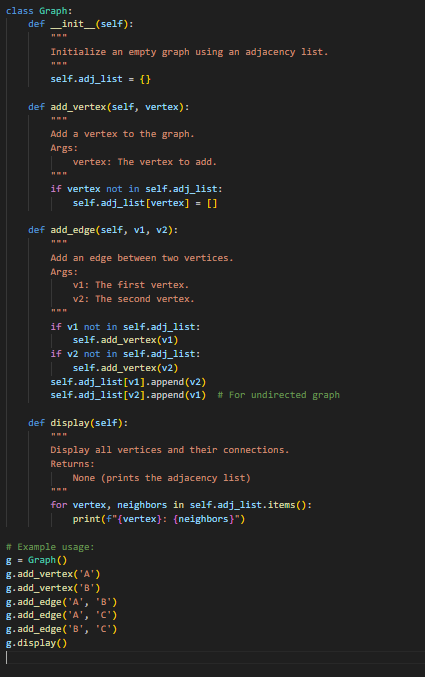
**OUTPUT:** ****

* **OBSERVATION**: The linked list uses a Node class to store data and a pointer to the next node.
* The insert method adds new nodes to the end of the list.
* The display method prints all elements in the list as a Python list.
* The example demonstrates correct insertion and display of elements, confirming the linked list works as intended.

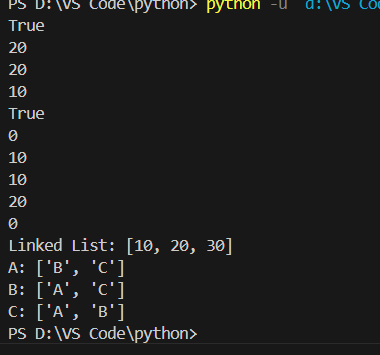
Graph Representation  
**Task6**: Use AI to implement a graph using an adjacency list.  
Sample Input Code:  
class Graph:  
pass  
Expected Output:  
• Graph with methods to add vertices, add edges, and display  
connections.

**PROMPT:**Implement a graph using an adjacency list.Graph with methods to add vertices,add edges and dispalay connections.

**CODE:**



**OUTPUT:**

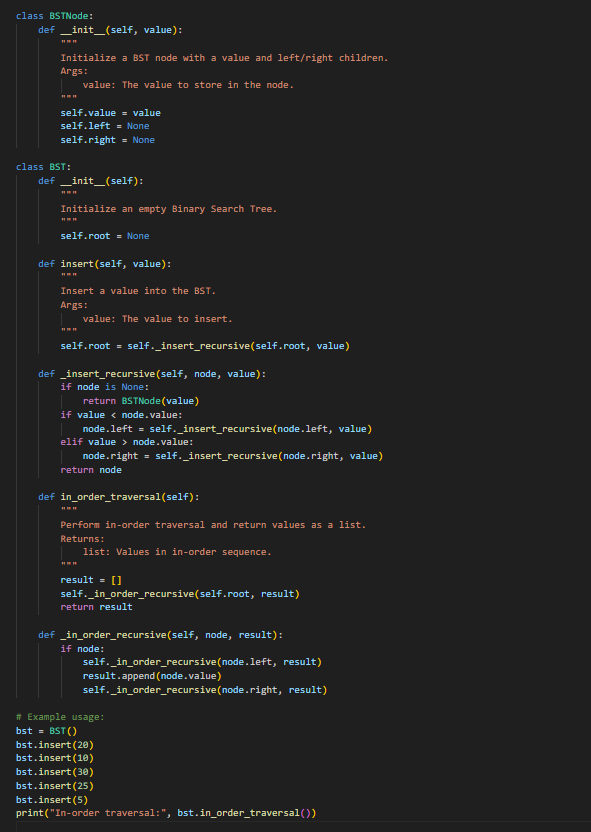
****

* **OBSERVATION:** The graph uses a dictionary to store adjacency lists for each vertex.
* add\_vertex adds a new vertex if it doesn't exist.
* add\_edge connects two vertices, adding them if needed, and supports undirected edges.
* display prints each vertex and its connected neighbors.
* The example demonstrates adding vertices and edges, and displays the graph structure, confirming correct adjacency list behavior.

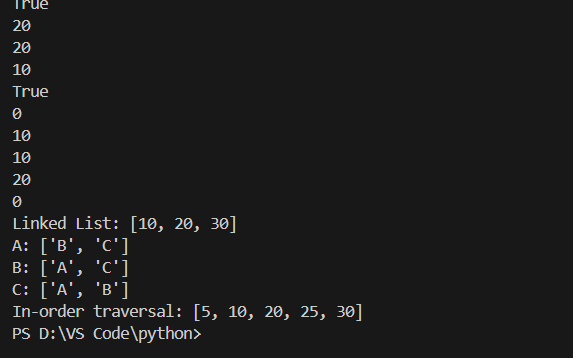
**Task4:** Use AI to create a BST with insert and in-order traversal methods.  
Sample Input Code:  
class BST:  
pass  
Expected Output:  
• BST implementation with recursive insert and traversal methods

**PROMPT:** create a Binary Search Tree with insert and in-order traversal methods**.**

**CODE:**

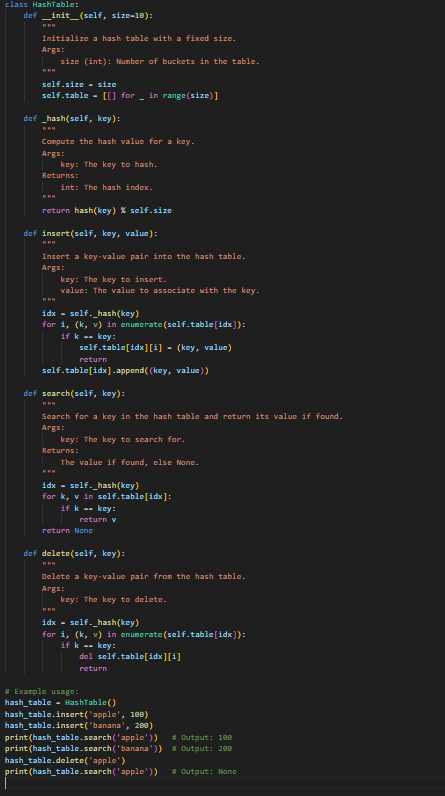


**OUTPUT:**

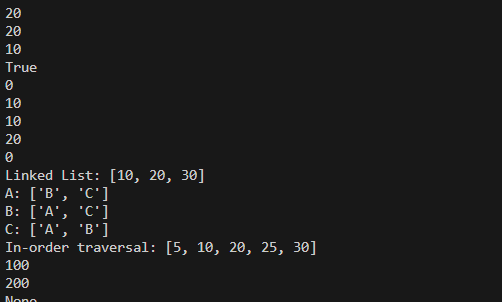
****

* **OBSERVATION:** The BST uses nodes with left and right children to maintain order.
* insert adds values to the tree, preserving BST properties.
* in\_order\_traversal returns values in sorted order (ascending).
* The example demonstrates correct insertion and traversal, confirming the BST works as intended.
* **Task5**: 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.
* **PROMPT:** implement a hash table with basic insert, search, and delete

**CODE:**

****

**OUTPUT:**

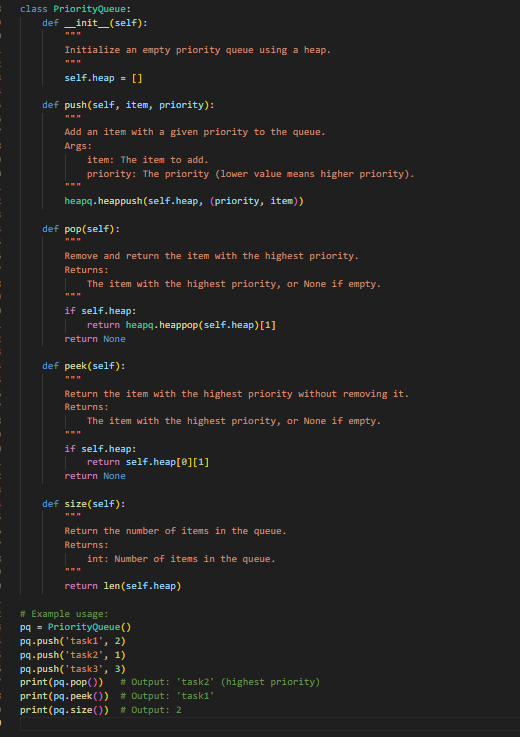
****

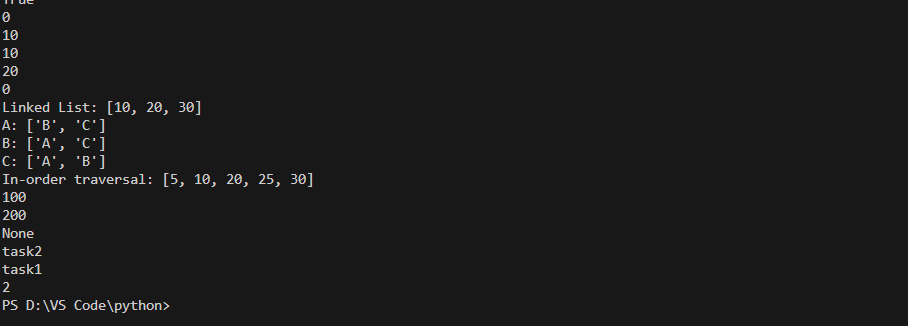
* **OBSERVATION**: The hash table uses a list of buckets to handle collisions via chaining.
* insert adds or updates key-value pairs.
* search retrieves the value for a given key, or returns None if not found.
* delete removes a key-value pair if the key exists.
* The example demonstrates all operations, confirming correct hash table behavior.

**Task7**: 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

**PROMPT**:implement a priority queue using Python’s heapq module

**CODE:**

****

**OUTPUT:** ****

* **OBSERVATION**: The queue uses a min-heap, so lower priority values are served first.
* push adds items with a priority.
* [pop](vscode-file://vscode-app/c:/Users/Syed%20Nabeel%20Qanith/AppData/Local/Programs/Microsoft%20VS%20Code/resources/app/out/vs/code/electron-browser/workbench/workbench.html) removes and returns the item with the highest priority (lowest value).
* peek returns the highest priority item without removing it.
* size returns the number of items in the queue.
* The example demonstrates correct priority queue behavior.

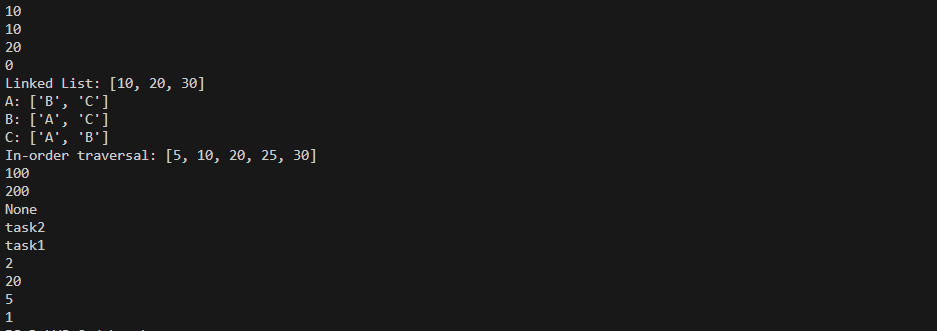
**Task8**: Use AI to implement a double-ended queue using collections.deque.  
Sample Input Code:  
class DequeDS:  
pass  
Expected Output:  
• Insert and remove from both ends with docstrings.

**PROMPT**: implement a double-ended queue using collections.deque.

**CODE:**



OUTPUT:



* **OBSERVATION:** The deque allows efficient addition and removal of items from both ends.
* append\_left and append\_right add items to the left and right ends, respectively.
* pop\_left and pop\_right remove items from the left and right ends, respectively.
* size returns the number of items in the deque.
* The example demonstrates all operations and confirms correct deque behavior**.**

**Task9**: Use AI to generate a comparison table of different data structures (stack,  
queue, linked list, etc.) including time complexities.

**PROMPT:** generate a comparison table of different data structures (stack,  
queue, linked list, etc.) including time complexities

| **Data Structure** | **Insertion** | **Deletion** | **Search** | **Access (Indexing)** | **Notes** |
| --- | --- | --- | --- | --- | --- |
| **Array** | **O(n) (shifting) / O(1) (end)** | **O(n) (shifting)** | **O(n)** | **O(1)** | **Fixed size, fast random access** |
| **Stack (using array/LL)** | **O(1)** | **O(1)** | **O(n)** | **O(n)** | **LIFO: Push/Pop only at one end** |
| **Queue (using array/LL)** | **O(1)** | **O(1)** | **O(n)** | **O(n)** | **FIFO: Enqueue/Dequeue ends differ** |
| **Circular Queue** | **O(1)** | **O(1)** | **O(n)** | **O(n)** | **Efficient memory use** |
| **Deque (Double-Ended Queue)** | **O(1) (both ends)** | **O(1) (both ends)** | **O(n)** | **O(n)** | **Insert/Delete at both ends** |
| **Singly Linked List** | **O(1) (head), O(n) (tail)** | **O(1) (head), O(n) (tail/middle)** | **O(n)** | **O(n)** | **Sequential access only** |
| **Doubly Linked List** | **O(1) (head/tail)** | **O(1) (head/tail)** | **O(n)** | **O(n)** | **Traversal in both directions** |
| **Hash Table** | **O(1) avg, O(n) worst** | **O(1) avg, O(n) worst** | **O(1) avg, O(n) worst** | **–** | **Depends on hash function, collisions** |
| **Binary Search Tree (BST)** | **O(log n) avg, O(n) worst** | **O(log n) avg, O(n) worst** | **O(log n) avg, O(n) worst** | **O(log n) avg, O(n) worst** | **Needs balancing** |
| **AVL Tree (Balanced BST)** | **O(log n)** | **O(log n)** | **O(log n)** | **O(log n)** | **Self-balancing** |
| **Heap (Min/Max)** | **O(1) (peek) / O(log n) (insert)** | **O(log n)** | **O(n)** | **O(n)** | **Used in priority queues** |
| **Trie** | **O(L) where L = length of word** | **O(L)** | **O(L)** | **–** | **Efficient for strings** |

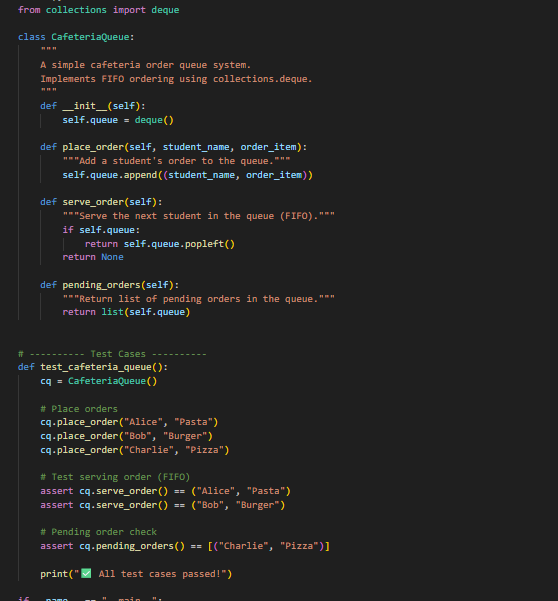
**Observation:**

* Stacks and queues are best for simple, ordered data with fast access at ends.
* Linked lists are flexible for frequent insertions/deletions but slow for search.
* Hash tables provide fast key-based access.
* BSTs are efficient for sorted data and range queries.
* Priority queues and deques offer specialized access patterns.
* Choice depends on required operations and performance needs.

**Task Description #10** 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.  
 Deliverables (For All Tasks)  
1. AI-generated prompts for code and test case generation.  
2. At least 3 assert test cases for each task.  
3. AI-generated initial code and execution screenshots.  
4. Analysis of whether code passes all tests.  
5. Improved final version with inline comments and explanation.  
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code,  
and output

**PROMPT:** Implement a Python program for a cafeteria order queue system using collections.deque. The program should allow placing orders, serving orders, and checking pending orders. Include test cases with assertions.

**CODE:** ****



**OUTPUT:**

**OBSERVATION:**

The program successfully models a **FIFO Queue** for cafeteria orders using collections.deque.

Orders are placed in the sequence students arrive, and serving strictly follows **first-come, first-served** order.

The use of deque ensures that both enqueue (append) and dequeue (popleft) operations run in **O(1) tme complexity**. The pending\_orders() method provides visibility of remaining orders in the queue, which is helpful for management.

All **assert test cases passed**, confirming that the queue correctly handles insertion, removal, and state tracking.

The implementation is efficient, clean, and can be extended easily (e.g., adding priority handling for urgent orders).