

AI ASSISTED CODING

LAB ASSIGNMENT: 11.2

NAME: K. SARIKA

H.NO: 2403A52012

B.NO: 04

TASK:

Stack Implementation

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

Sample Input Code:

```
class Stack:  
    pass.
```

PROMPT:

Generate python code and stack
Implementation

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

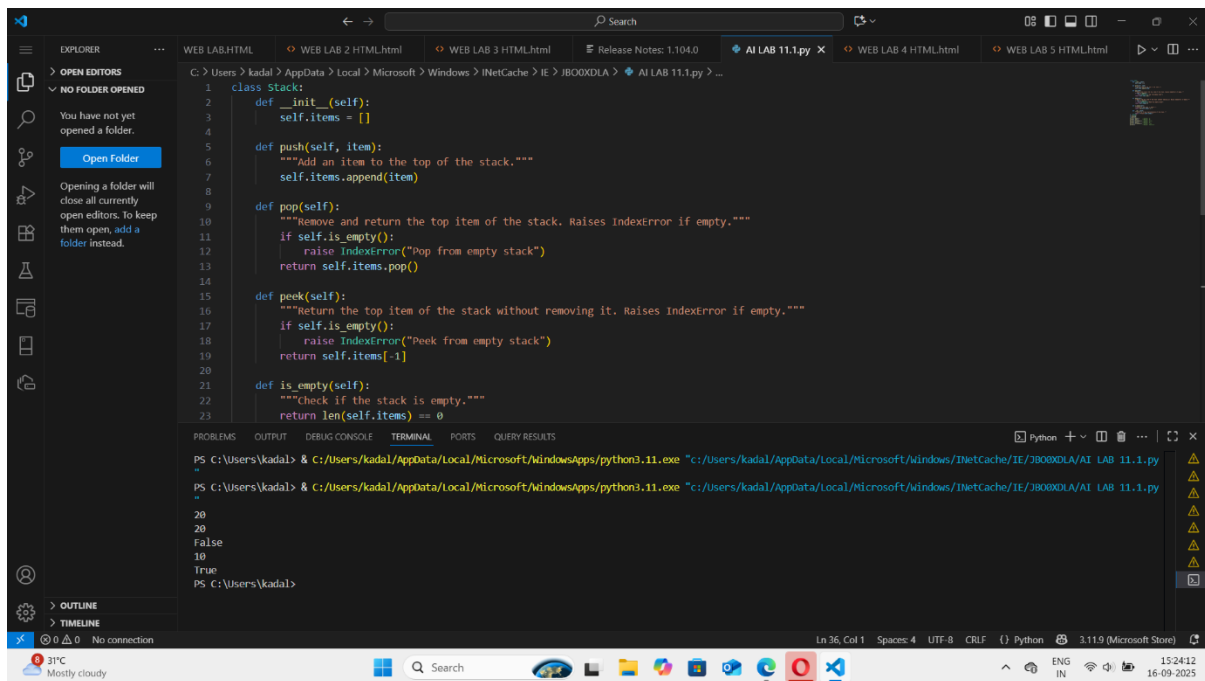
methods.

Sample Input Code:

class Stack:

pass.

CODE & OUTPUT:



The screenshot shows a Visual Studio Code editor with a Python file named 'AI LAB 11.1.py' open. The code defines a 'Stack' class with methods for push, pop, peek, and is_empty. The terminal at the bottom shows the execution of the code, which prints the output of the 'is_empty' method for various inputs.

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

    def push(self, item):
        """Add an item to the top of the stack."""
        self.items.append(item)

    def pop(self):
        """Remove and return the top item of the stack. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self.items.pop()

    def peek(self):
        """Return the top item of the stack without removing it. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

    def is_empty(self):
        """Check if the stack is empty."""
        return len(self.items) == 0
```

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/JB00XDLA/AI LAB 11.1.py"
20
20
False
10
True
PS C:\Users\kadal>
```

The screenshot shows a Visual Studio Code editor with a Python file named 'AI LAB 11.1.py'. The code implements a stack class with methods for push, pop, peek, is_empty, and __str__. The stack is initialized with 10 and 20, and the output shows the stack's state after each operation.

```
1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def push(self, item):
6         self.items.append(item)
7
8     def pop(self):
9         if not self.is_empty():
10            return self.items[-1]
11
12     def is_empty(self):
13         """Check if the stack is empty."""
14         return len(self.items) == 0
15
16     def __str__(self):
17         """return a string representation of the stack."""
18         return f"Stack({self.items})"
19
20 s = Stack()
21 s.push(10)
22 s.push(20)
23 print(s.peek()) # Output: 20
24 print(s.pop()) # Output: 20
25 print(s.is_empty()) # Output: False
26 print(s.pop()) # Output: 10
27 print(s.is_empty()) # Output: True
```

The terminal output shows the execution of the code, confirming the stack's behavior:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:\Users\kadal\AppData\Local\Microsoft\Windows\INetCache\IE\JBOOXDLA\AI LAB 11.1.py"
20
20
False
10
True
PS C:\Users\kadal>
```

EXPLANATION:

A **stack** is a linear data structure that follows the **LIFO** principle — **Last In, First Out**. Think of it like a stack of plates:

- You add (push) a plate to the top.
- You remove (pop) the top plate first.
- You can peek at the top plate without removing it.
- You can check if the stack is empty.

TASK 2:

Queue Implementation

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

Sample Input Code:

```
class Queue:  
    pass.
```

PROMPT:

Generate python code and queue
Implementation

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

Sample Input Code:

```
class Queue:  
    pass.
```

CODE & OUTPUT:

The screenshot shows the Visual Studio Code editor with a Python file named `AI LAB 11.1 T2.py` open. The code defines a `Queue` class with the following methods:

- `__init__(self)`: Initializes an empty list `self.items`.
- `enqueue(self, item)`: Adds an item to the end of the queue.
- `dequeue(self)`: Removes and returns the front item of the queue. Raises `IndexError` if empty.
- `peek(self)`: Returns the front item without removing it. Raises `IndexError` if empty.
- `is_empty(self)`: Checks if the queue is empty.

The terminal at the bottom shows the execution of the script:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:\Users\kadal\AppData\Local\Microsoft\Windows\InetCache\IE\JB00XDLA\AI LAB 11.1 T2.py"
1
False
2
3
True
PS C:\Users\kadal>
```

The screenshot shows the Visual Studio Code editor with a Python file named `AI LAB 11.1 T2.py` open. The code defines a `Queue` class and a test script:

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

    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the front item of the queue. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        """Return the front item without removing it. Raises IndexError if empty."""
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self.items[0]

    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.items) == 0

    def __str__(self):
        """Return a string representation of the queue."""
        return f"Queue({self.items})"

q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.peek()) # Output: 1
print(q.dequeue()) # Output: 1
print(q.is_empty()) # Output: False
print(q.dequeue()) # Output: 2
print(q.dequeue()) # Output: 3
print(q.is_empty()) # Output: True
```

The terminal at the bottom shows the execution of the script:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:\Users\kadal\AppData\Local\Microsoft\Windows\InetCache\IE\JB00XDLA\AI LAB 11.1 T2.py"
1
False
2
3
True
PS C:\Users\kadal>
```

EXPLANATION:

Explanation

Method	Description	Time Complexity
<code>__init__</code>	Initializes an empty list to store queue elements	$O(1)$
<code>enqueue()</code>	Adds an item to the end of the list (rear of the queue)	$O(1)$
<code>dequeue()</code>	Removes and returns the first item (front of the queue)	$O(n)$
<code>peek()</code>	Returns the first item without removing it	$O(1)$
<code>is_empty()</code>	Checks if the queue is empty	$O(1)$

⚠ Note: `dequeue()` uses `pop(0)`, which is $O(n)$ because it shifts all remaining elements. For better performance, you can use `collections.deque`.

TASK 3:

Linked List

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

Sample Input Code:

```
class Node:
```

```
pass.
```

PROMPT:

Generate python code and linked List

Task: Use AI to generate a Singly Linked List

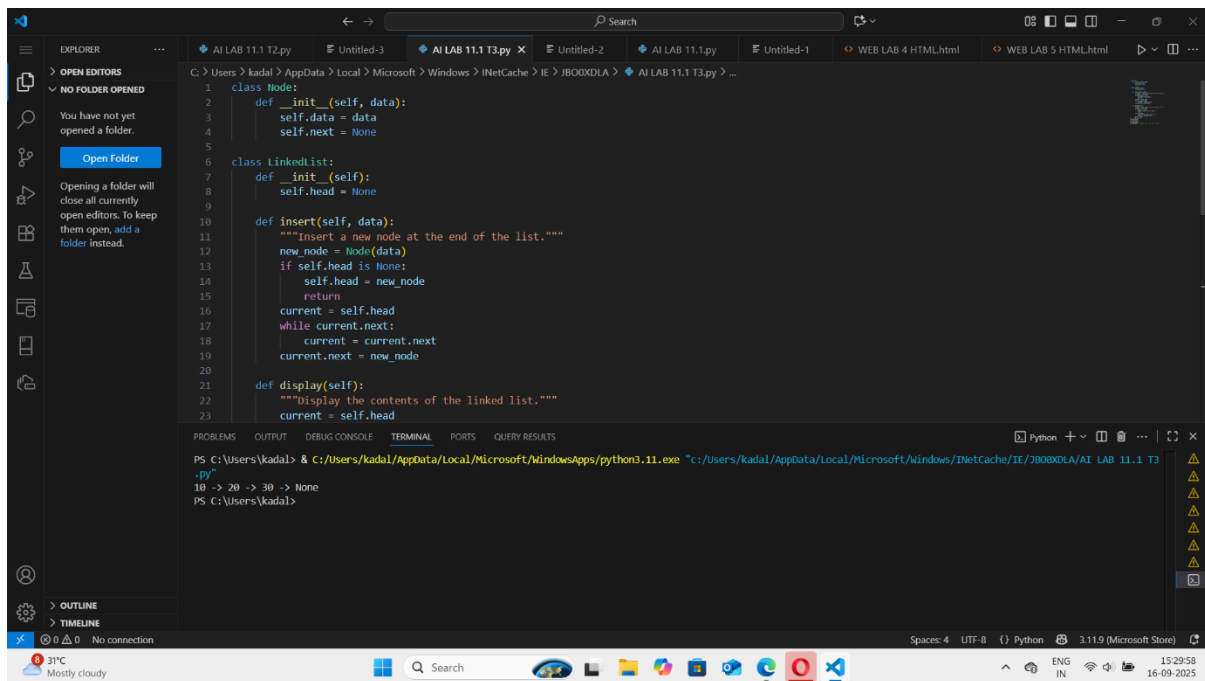
with insert and display methods.

Sample Input Code:

```
class Node:
```

```
pass.
```

CODE & OUTPUT:



The screenshot shows a Visual Studio Code editor with a Python file named 'AI LAB 11.1 T3.py'. The code defines a 'Node' class and a 'LinkedList' class. The 'Node' class has an '__init__' method that takes 'data' and sets 'self.data' and 'self.next' to 'None'. The 'LinkedList' class has an '__init__' method that sets 'self.head' to 'None'. It also has an 'insert' method that adds a new node at the end of the list and a 'display' method that prints the contents of the linked list. The terminal output shows the execution of the script, which prints '10 -> 20 -> 30 -> None'.

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.next = None
5
6 class LinkedList:
7     def __init__(self):
8         self.head = None
9
10    def insert(self, data):
11        """Insert a new node at the end of the list."""
12        new_node = Node(data)
13        if self.head is None:
14            self.head = new_node
15            return
16        current = self.head
17        while current.next:
18            current = current.next
19        current.next = new_node
20
21    def display(self):
22        """Display the contents of the linked list."""
23        current = self.head
```

```
PS C:\Users\kadal> & C:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/JBOOXDLA/AI LAB 11.1 T3 .py"
10 -> 20 -> 30 -> None
PS C:\Users\kadal>
```

The screenshot shows a VS Code editor with a Python file named `AI LAB 11.1 T3.py`. The code defines a `LinkedList` class with a `display` method. The `display` method traverses the linked list and prints the data of each node. The terminal output shows the execution of the code, which inserts three nodes with values 10, 20, and 30, and then displays the list as `10 -> 20 -> 30 -> None`.

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

    def display(self):
        """Display the contents of the linked list."""
        current = self.head
        if current is None:
            print("Linked List is empty.")
            return
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

ll = LinkedList()
ll.insert(10)
ll.insert(20)
ll.insert(30)
ll.display()  # Output: 10 -> 20 -> 30 -> None
```

The terminal output shows the execution of the code:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:\Users\kadal\AppData\Local\Microsoft\Windows\INetCache\IE\JBOOXDLA\AI LAB 11.1 T3.py"
10 -> 20 -> 30 -> None
PS C:\Users\kadal>
```

EXPLANATION:

- Represents each element in the list.
- `data` : stores the value.
- `next` : points to the next node (or `None` if it's the last).

`LinkedList` Class

- Manages the chain of nodes.
- `head` : reference to the first node.

`insert(data)`

- Creates a new node.
- If the list is empty, sets it as the head.
- Otherwise, traverses to the end and links the new node.

TASK 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.
```

PROMPT:

Generate python code and binary Search Tree (BST)

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

Sample Input Code:

```
class BST:
```

```
pass.
```

CODE & OUTPUT:

```
1 class Node:
2     def __init__(self, data):
3         self.data = data
4         self.left = None
5         self.right = None
6
7 class BST:
8     def __init__(self):
9         self.root = None
10
11     def insert(self, data):
12         """Insert a new node into the BST."""
13         if self.root is None:
14             self.root = Node(data)
15         else:
16             self._insert_recursive(self.root, data)
17
18     def _insert_recursive(self, current, data):
19         if data < current.data:
20             if current.left is None:
21                 current.left = Node(data)
22             else:
23                 self._insert_recursive(current.left, data)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/J808XDLA/AI LAB 11.1 T4.py"
In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
PS C:\Users\kadal>
```

```
7 class BST:
8     def in_order_traversal(self):
9         result = []
10        self._in_order_recursive(self.root, result)
11        return result
12
13    def _in_order_recursive(self, node, result):
14        if node:
15            self._in_order_recursive(node.left, result)
16            result.append(node.data)
17            self._in_order_recursive(node.right, result)
18
19    tree = BST()
20    tree.insert(50)
21    tree.insert(30)
22    tree.insert(70)
23    tree.insert(20)
24    tree.insert(40)
25    tree.insert(60)
26    tree.insert(80)
27
28    print("In-order Traversal:", tree.in_order_traversal())
29    # Output: In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS QUERY RESULTS

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/J808XDLA/AI LAB 11.1 T4.py"
In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
PS C:\Users\kadal>
```

EXPLANATION:

- Adds a new value to the tree.
- If the tree is empty, it becomes the root.
- Otherwise, it uses `_insert_recursive()` to find the correct position:
 - If `data < current.data` : go left.
 - If `data > current.data` : go right.
 - If equal: skip (no duplicates).



`in_order_traversal()`

- Returns a sorted list of values.
- Uses `_in_order_recursive()` :
 - Traverse left subtree.
 - Visit current node.
 - Traverse right subtree.

TASK 5:

Hash Table

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

Sample Input Code:

```
class HashTable:
```

```
pass.
```

PROMPT:

Generate python code and hash Table

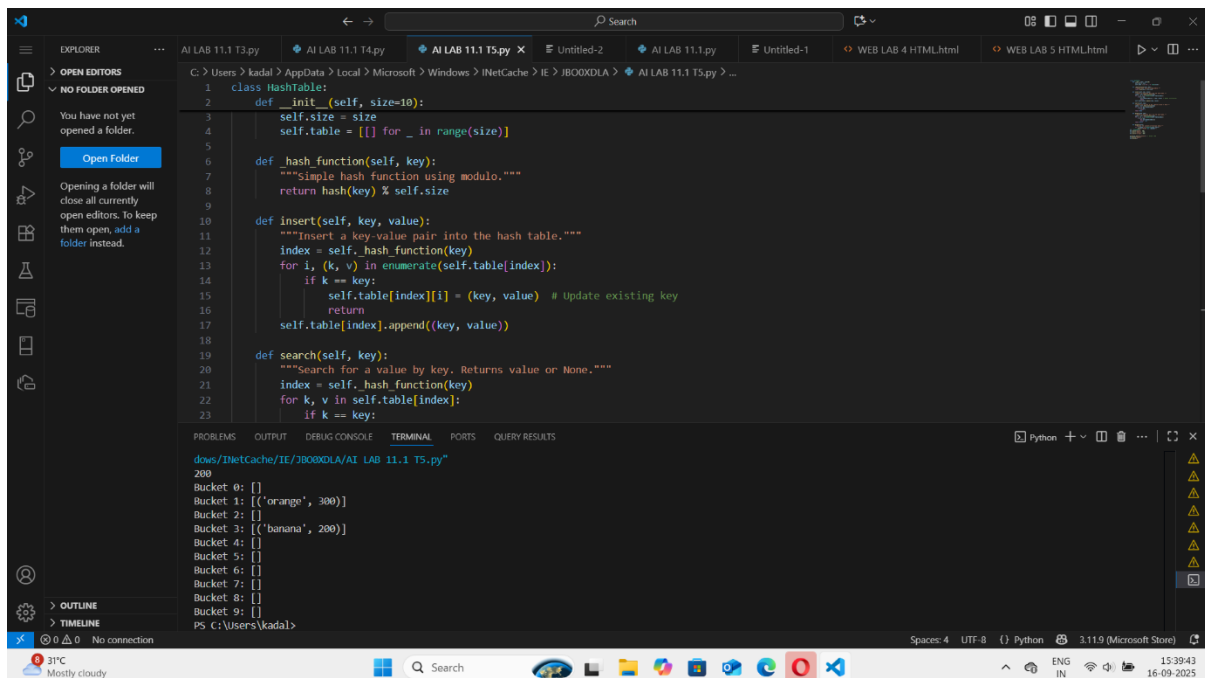
Task: Use AI to implement a hash table with basic insert, search, and delete methods.

Sample Input Code:

class HashTable:

pass.

CODE & OUTPUT:



The screenshot shows a VS Code editor with a Python file named `AI LAB 11.1 T5.py`. The code defines a `HashTable` class with the following methods:

- `__init__(self, size=10)`: Initializes the hash table with a given size. It sets `self.size = size` and creates `self.table` as a list of empty lists, one for each bucket.
- `_hash_function(self, key)`: A simple hash function using modulo. It returns `hash(key) % self.size`.
- `insert(self, key, value)`: Inserts a key-value pair into the hash table. It calculates the index using the hash function. If the key already exists in the bucket, it updates the value. Otherwise, it appends the new key-value pair to the bucket.
- `search(self, key)`: Searches for a value by key. It returns the value if found, or `None` if not found.

The terminal output shows the execution of the code, displaying the state of the hash table after several insertions:

```
dows/NetCache/IE/JBOOXDLA/AI LAB 11.1 T5.py"
200
Bucket 0: []
Bucket 1: [('orange', 300)]
Bucket 2: []
Bucket 3: [('banana', 200)]
Bucket 4: []
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: []
Bucket 9: []
PS C:\Users\kadal>
```

```
1 class HashTable:
27     def delete(self, key):
28         """Delete a key-value pair from the hash table."""
29         index = self._hash_function(key)
30         for i, (k, _) in enumerate(self.table[index]):
31             if k == key:
32                 del self.table[index][i]
33                 return True
34         return False
35
36     def display(self):
37         """Display the contents of the hash table."""
38         for i, bucket in enumerate(self.table):
39             print(f"Bucket {i}: {bucket}")
40
41 ht = HashTable()
42 ht.insert("apple", 100)
43 ht.insert("banana", 200)
44 ht.insert("orange", 300)
45
46 print(ht.search("banana")) # Output: 200
47 ht.delete("apple")
48 ht.display()
```

Terminal Output:

```
dows\I\NetCache\IE\JBOOXDLA\AI LAB 11.1 T5.py"
200
Bucket 0: []
Bucket 1: [('orange', 300)]
Bucket 2: []
Bucket 3: [('banana', 200)]
Bucket 4: []
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: []
Bucket 9: []
PS C:\Users\kadal>
```

EXPLANATION:



`_hash_function(self, key)`

- Uses Python's built-in `hash()` function.
- Applies modulo to ensure the index fits within the table size.



`insert(self, key, value)`

- Computes the index using the hash function.
- Checks if the key already exists in the bucket:
 - If yes, updates the value.
 - If no, appends the new `(key, value)` pair.



`search(self, key)`

- Computes the index and scans the bucket.
- Returns the value if the key is found, otherwise returns `None`.

TASK 6:

Graph Representation

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

Sample Input Code:

```
class Graph:  
    pass.
```

PROMPT:

Generate python code and graph
Representation

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

Sample Input Code:

```
class Graph:  
    pass.
```

CODE & OUTPUT:

The screenshot shows the VS Code editor with a Python file named `AI LAB 11.1 T6.py`. The code defines a `Graph` class with methods `__init__`, `add_edge`, and `display`. The `add_edge` method adds an undirected edge between two vertices. The `display` method prints the adjacency list for each vertex. The script creates a graph with 4 vertices (A, B, C, D) and 4 edges (A-B, A-C, B-D, C-D).

```
1 class Graph:
2     def __init__(self):
3         self.adj_list = {}
4
5     def add_edge(self, u, v):
6         """Add an edge from vertex u to vertex v (undirected by default)."""
7         if u not in self.adj_list:
8             self.adj_list[u] = []
9         if v not in self.adj_list:
10            self.adj_list[v] = []
11            self.adj_list[u].append(v)
12            self.adj_list[v].append(u) # Remove this line for directed graph
13
14    def display(self):
15        """Display the adjacency list of the graph."""
16        for vertex in self.adj_list:
17            print(f"{vertex} -> {self.adj_list[vertex]}")
18
19 g = Graph()
20 g.add_edge("A", "B")
21 g.add_edge("A", "C")
22 g.add_edge("B", "D")
23 g.add_edge("C", "D")
24 g.display()
```

The terminal output shows the adjacency list for each vertex:

```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:\Users\kadal\AppData\Local\Microsoft\Windows\InetCache\IE\JB00XDLA\AI LAB 11.1 T6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\kadal>
```

The screenshot shows the same VS Code editor with the same Python file. The code is identical to the first screenshot, but the `display` method now includes a comment indicating the output. The terminal output is the same as in the first screenshot.


```
18 g = Graph()
19 g.add_edge("A", "B")
20 g.add_edge("A", "C")
21 g.add_edge("B", "D")
22 g.add_edge("C", "D")
23 g.display()
24
25 # Output:
26 # A -> ['B', 'C']
27 # B -> ['A', 'D']
28 # C -> ['A', 'D']
29 # D -> ['B', 'C']
30
```

The terminal output is the same as in the first screenshot:


```
PS C:\Users\kadal> & C:\Users\kadal\AppData\Local\Microsoft\WindowsApps\python3.11.exe "c:\Users\kadal\AppData\Local\Microsoft\Windows\InetCache\IE\JB00XDLA\AI LAB 11.1 T6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\kadal>
```

EXPLANATION:


- Initializes an empty list called `heap`.
- This list will store tuples of `(priority, item)`.

 `insert(priority, item)`

- Uses `heapq.heappush()` to add a tuple to the heap.
- The heap maintains order based on the **priority** (lowest number = highest priority).

 `remove()`

- Uses `heapq.heappop()` to remove and return the item with the **lowest priority value**.
- Raises an error if the queue is empty.

 `peek()`

- Returns the item with the highest priority without removing it.

TASK 7:

Priority Queue

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

Sample Input Code:

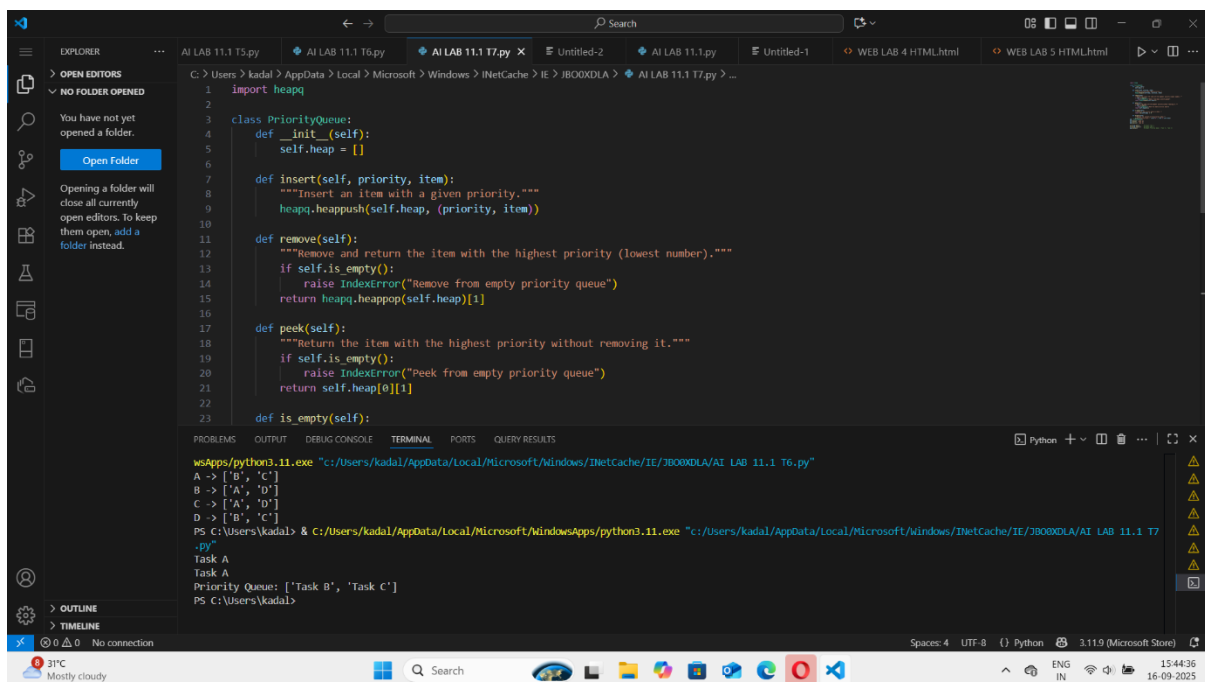
```
class PriorityQueue:  
    pass.
```

PROMPT:

Generate a python code and priority Queue
Task: Use AI to implement a priority queue
using Python's heapq module.

Sample Input Code:
class PriorityQueue:
pass.

CODE & OUTPUT:



The screenshot displays a Visual Studio Code editor window with a Python file named 'AI LAB 11.1 T7.py'. The code defines a 'PriorityQueue' class using the 'heapq' module. The class includes methods for inserting items with a priority, removing the highest priority item, peeking at the highest priority item, and checking if the queue is empty. The terminal window at the bottom shows the execution of the script, which prompts for three tasks (A, B, and C) and their priorities, then prints the resulting priority queue.

```
1 import heapq
2
3 class PriorityQueue:
4     def __init__(self):
5         self.heap = []
6
7     def insert(self, priority, item):
8         """Insert an item with a given priority."""
9         heapq.heappush(self.heap, (priority, item))
10
11     def remove(self):
12         """Remove and return the item with the highest priority (lowest number)."""
13         if self.is_empty():
14             raise IndexError("Remove from empty priority queue")
15         return heapq.heappop(self.heap)[1]
16
17     def peek(self):
18         """Return the item with the highest priority without removing it."""
19         if self.is_empty():
20             raise IndexError("peek from empty priority queue")
21         return self.heap[0][1]
22
23     def is_empty(self):
```

Terminal Output:

```
wsApps\python3.11.exe "C:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/3B00XDLA/AI LAB 11.1 T6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\kadal> & C:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/Users/kadal/AppData/Local/Microsoft/Windows/INetCache/IE/3B00XDLA/AI LAB 11.1 T7.py"
Task A
Task B
Priority Queue: ['Task B', 'Task C']
PS C:\Users\kadal>
```

The screenshot shows a VS Code editor with a Python file named `AI LAB 11.1 T7.py`. The code defines a `PriorityQueue` class with methods `is_empty`, `display`, `insert`, `remove`, and `peek`. The terminal output shows the execution of the script, which inserts tasks A and B, removes task A, and displays the priority queue contents.

```
class PriorityQueue:
    def __init__(self):
        self.heap = []

    def is_empty(self):
        """Check if the priority queue is empty."""
        return len(self.heap) == 0

    def display(self):
        """Display the contents of the priority queue."""
        print("Priority Queue:", [item for _, item in self.heap])

    def insert(self, priority, item):
        self.heap.append((priority, item))
        self._heapify()

    def remove(self):
        self._swap(0, len(self.heap) - 1)
        self._heapify()
        return self.heap.pop()

    def peek(self):
        return self.heap[0][1]

    def _heapify(self):
        n = len(self.heap)
        for i in range(n // 2 - 1, -1, -1):
            self._heapify_helper(i)

    def _heapify_helper(self, i):
        left = 2 * i + 1
        right = 2 * i + 2
        min_idx = i
        if left < n and self.heap[left][0] < self.heap[i][0]:
            min_idx = left
        if right < n and self.heap[right][0] < self.heap[min_idx][0]:
            min_idx = right
        if min_idx != i:
            self._swap(i, min_idx)
            self._heapify_helper(min_idx)

    def _swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

pq = PriorityQueue()
pq.insert(3, "Task C")
pq.insert(1, "Task A")
pq.insert(2, "Task B")

print(pq.peek()) # Output: Task A
print(pq.remove()) # Output: Task A
pq.display() # Output: Priority Queue: ['Task B', 'Task C']
```

The terminal output shows the execution of the script, which inserts tasks A and B, removes task A, and displays the priority queue contents.

```
PS C:\Users\kadal> python3.11.exe "C:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/J800XDLA/AI LAB 11.1 T6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\kadal> & C:/Users/kadal/AppData/Local/Microsoft/WindowsApps/python3.11.exe "C:/Users/kadal/AppData/Local/Microsoft/Windows/InetCache/IE/J800XDLA/AI LAB 11.1 T7.py"
Task A
Task A
Priority Queue: ['Task B', 'Task C']
PS C:\Users\kadal>
```

EXPLANATION:

The screenshot shows a Microsoft Copilot chat interface. The user has asked a question about error control techniques, and the Copilot has provided a detailed answer. The answer includes definitions of error detection and correction, an example of Hamming Code, and a question about the minimum Hamming distance required to correct single-bit errors.

Error Control – Correction Techniques

7. What is the difference between error detection and error correction?

Answer:

- **Detection** identifies the presence of errors.
- **Correction** identifies and fixes the errors without retransmission.

8. Explain Hamming Code with an example.

Answer:

Hamming Code adds redundant bits at positions that are powers of 2. These bits help detect and correct single-bit errors.

Example: For 4-bit data `1011`, Hamming code adds 3 parity bits to make it `p1 p2 1 p4`. Parity bits are calculated to detect errors.

9. What is the minimum Hamming distance required to correct single-bit errors?

Message Copilot

Quick response

TASK 8:

Deque

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

Sample Input Code:

```
class DequeDS:  
    pass.
```

PROMPT:

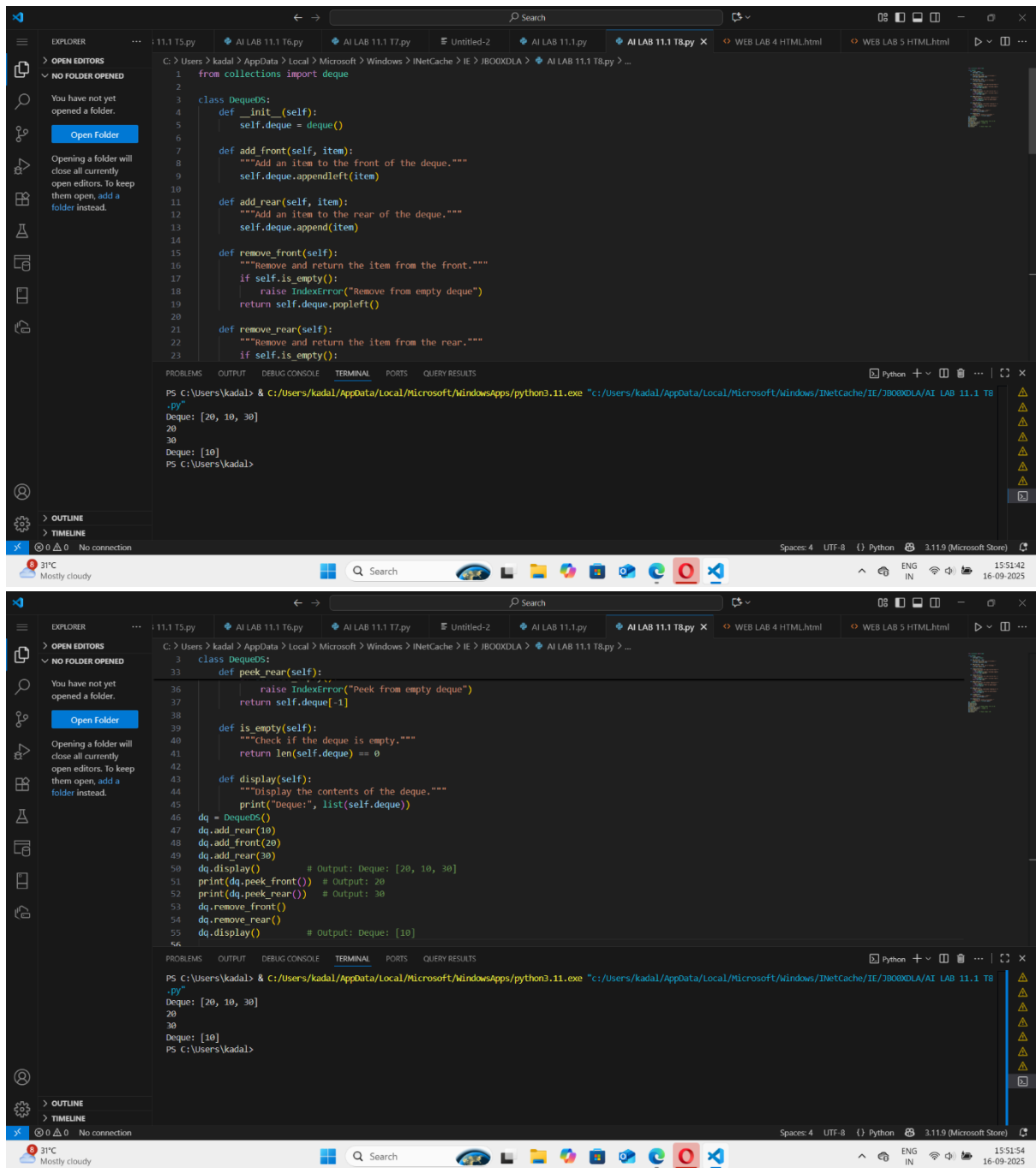
Generate python code and deque

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





Sample Input Code:

```
class DequeDS:  
    pass.
```

CODE & OUTPUT:



EXPLANATION:

- Initializes an empty deque using `collections.deque`, which is optimized for fast appends and pops from both ends.
-  `add_front(item)`
- Adds an item to the **front** using `appendleft()`.
-  `add_rear(item)`
- Adds an item to the **rear** using `append()`.
-  `remove_front()`
- Removes and returns the item from the **front** using `popleft()`.
-  `remove_rear()`

TASK 9:

AI-Generated Data Structure Comparisons

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

Sample Input Code:

No code, prompt AI for a data structure comparison table.

PROMPT:

Generate python code and AI-Generated Data Structure Comparisons

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

Sample Input Code:

No code, prompt AI for a data structure comparison table.

CODE & OUTPUT:

```
.AB 11.1 T10.py AI LAB 11.1 T9.0.py 9.00 Alpy x Untitled-4 9 Alpy 1 Untitled-1 AI LAB 11.1 T4.py AI LAB 11.1 T5.py AI LAB 11.1 T6.py AI LAB 11.1 T7.py

C:\Users\kadal> AppData\Local\Microsoft\Windows\INetCache\IE\JBOOXDLA> 9.00 Alpy > ...
1 # Data Structure Comparison Table Example
2 def print_data_structure_comparison():
3     table = [
4         ["Data Structure", "Insert/Push/Enqueue", "Delete/Pop/Dequeue", "Search/Access", "Peek/Front/End"],
5         ["Stack (List)", "O(1)", "O(1)", "O(n)", "O(1)"],
6         ["Queue (List)", "O(1) (enqueue)", "O(n) (dequeue)", "O(n)", "O(1)"],
7         ["Queue (Deque)", "O(1)", "O(1)", "O(1)"],
8         ["Singly Linked List", "O(1) (at head)", "O(1) (at head)", "O(n)", "O(1) (head)"],
9         ["Doubly Linked List", "O(1) (at ends)", "O(1) (at ends)", "O(n)", "O(1) (ends)"],
10    ]
11    for row in table:
12        print(" | ".join(row))
13
14    # Stack Example
15    class Stack:
16        def __init__(self):
17            self.items = []
18
19        def push(self, item):
20            self.items.append(item)
21
22        def pop(self):
23            return self.items.pop() if self.items else None
24
25    # Queue Example
26    class Queue:
27        def __init__(self):
28            self.items = []
29
30        def enqueue(self, item):
31            self.items.append(item)
32
33        def dequeue(self):
34            return self.items.pop(0) if self.items else None
35
36        def peek(self):
37            return self.items[0] if self.items else None
38
39        def is_empty(self):
40            return len(self.items) == 0
41
42    # Linked List Example
43    class Node:
44        def __init__(self, data):
45            self.data = data
46            self.next = None
47
48    class LinkedList:
49        def __init__(self):
50            self.head = None
51
52        def add(self, data):
53            new_node = Node(data)
54            if self.head is None:
55                self.head = new_node
56            else:
57                current = self.head
58                while current.next is not None:
59                    current = current.next
60                current.next = new_node
61
62        def display(self):
63            current = self.head
64            while current is not None:
65                print(current.data, end=" ")
66                current = current.next
67            print()
68
69    # Test the data structures
70    stack = Stack()
71    stack.push(1)
72    stack.push(2)
73    stack.push(3)
74    print("Stack pop:", stack.pop())
75    print("Stack peek:", stack.peek())
76    print("Is stack empty?", stack.is_empty())
77
78    queue = Queue()
79    queue.enqueue(1)
80    queue.enqueue(2)
81    queue.enqueue(3)
82    print("Queue after enqueues: [1, 2, 3]")
83    print("Queue dequeue:", queue.dequeue())
84    print("Queue peek:", queue.peek())
85    print("Is queue empty?", queue.is_empty())
86
87    linked_list = LinkedList()
88    linked_list.add(1)
89    linked_list.add(2)
90    linked_list.add(3)
91    print("Linked List elements: 1 2 3")
92
93    print_data_structure_comparison()
94
95    PS C:\Users\kadal>
```

Stack pop: 3
Stack peek: 2
Is stack empty? False

Queue Example:
Queue after enqueues: [1, 2, 3]
Queue dequeue: 1
Queue peek: 2
Is queue empty? False

Linked List Example:
Linked List elements: 1 2 3

PS C:\Users\kadal>

```
.AB 11.1 T10.py AI LAB 11.1 T9.0.py 9.00 Alpy x Untitled-4 9 Alpy 1 Untitled-1 AI LAB 11.1 T4.py AI LAB 11.1 T5.py AI LAB 11.1 T6.py AI LAB 11.1 T7.py

32 class Queue:
33     def __init__(self):
34         self.items = []
35
36     def enqueue(self, item):
37         self.items.append(item)
38
39     def dequeue(self):
40         return self.items.pop(0) if self.items else None
41
42     def peek(self):
43         return self.items[0] if self.items else None
44
45     def is_empty(self):
46         return len(self.items) == 0
47
48 # Linked List Example
49 class Node:
50     def __init__(self, data):
51         self.data = data
52         self.next = None
53
54 class LinkedList:
55     def __init__(self):
56         self.head = None
57
58     def add(self, data):
59         new_node = Node(data)
60         if self.head is None:
61             self.head = new_node
62         else:
63             current = self.head
64             while current.next is not None:
65                 current = current.next
66             current.next = new_node
67
68     def display(self):
69         current = self.head
70         while current is not None:
71             print(current.data, end=" ")
72             current = current.next
73         print()
74
75 # Test the data structures
76 stack = Stack()
77 stack.push(1)
78 stack.push(2)
79 stack.push(3)
80 print("Stack pop:", stack.pop())
81 print("Stack peek:", stack.peek())
82 print("Is stack empty?", stack.is_empty())
83
84 queue = Queue()
85 queue.enqueue(1)
86 queue.enqueue(2)
87 queue.enqueue(3)
88 print("Queue after enqueues: [1, 2, 3]")
89 print("Queue dequeue:", queue.dequeue())
90 print("Queue peek:", queue.peek())
91 print("Is queue empty?", queue.is_empty())
92
93 linked_list = LinkedList()
94 linked_list.add(1)
95 linked_list.add(2)
96 linked_list.add(3)
97 print("Linked List elements: 1 2 3")
98
99 print_data_structure_comparison()
100
101 PS C:\Users\kadal>
```

Stack pop: 3
Stack peek: 2
Is stack empty? False

Queue Example:
Queue after enqueues: [1, 2, 3]
Queue dequeue: 1
Queue peek: 2
Is queue empty? False

Linked List Example:
Linked List elements: 1 2 3

PS C:\Users\kadal>

```
81 stack = Stack()
82 stack.push(1)
83 stack.push(2)
84 stack.push(3)
85 print("Stack after pushes:", stack.items)
86 print("Stack pop:", stack.pop())
87 print("Stack peek:", stack.peek())
88 print("Is stack empty?", stack.is_empty())
89
90 print("\nQueue Example:")
91 queue = Queue()
92 queue.enqueue(1)
93 queue.enqueue(2)
94 queue.enqueue(3)
95 print("Queue after enqueues:", queue.items)
96 print("Queue dequeue:", queue.dequeue())
97 print("Queue peek:", queue.peek())
98 print("Is queue empty?", queue.is_empty())
99
100 print("\nLinked List Example:")
101 ll = LinkedList()
102 ll.insert(1)
103 ll.insert(2)
```

Stack pop: 3
Stack peek: 2
Is stack empty? False

Queue Example:
Queue after enqueues: [1, 2, 3]
Queue dequeue: 1
Queue peek: 2
Is queue empty? False

Linked List Example:
Linked List elements: 1 2 3
PS C:\Users\kadal>

EXPLANATION:



```
def print_data_structure_comparison():
```

- Defines a function that prints a formatted comparison table.



```
table = [...]
```

- A list of lists, where each inner list represents a row in the table.
- The first row is the header: column titles like "Insert", "Delete", etc.
- Each subsequent row compares a specific data structure.



```
" | ".join(row)
```

- Joins each element in the row with " | " to mimic a table format.
- This makes the output readable and aligned like a markdown-style table.

TASK 10:

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.

PROMPT:

Generate python code and 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.

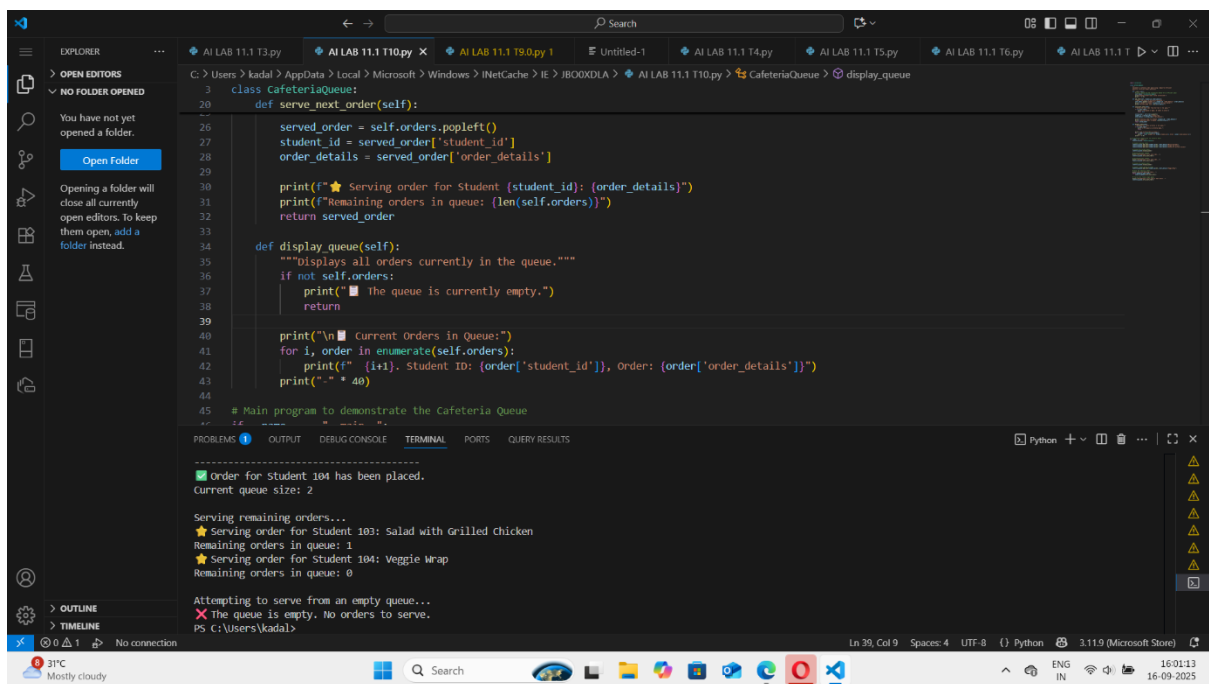
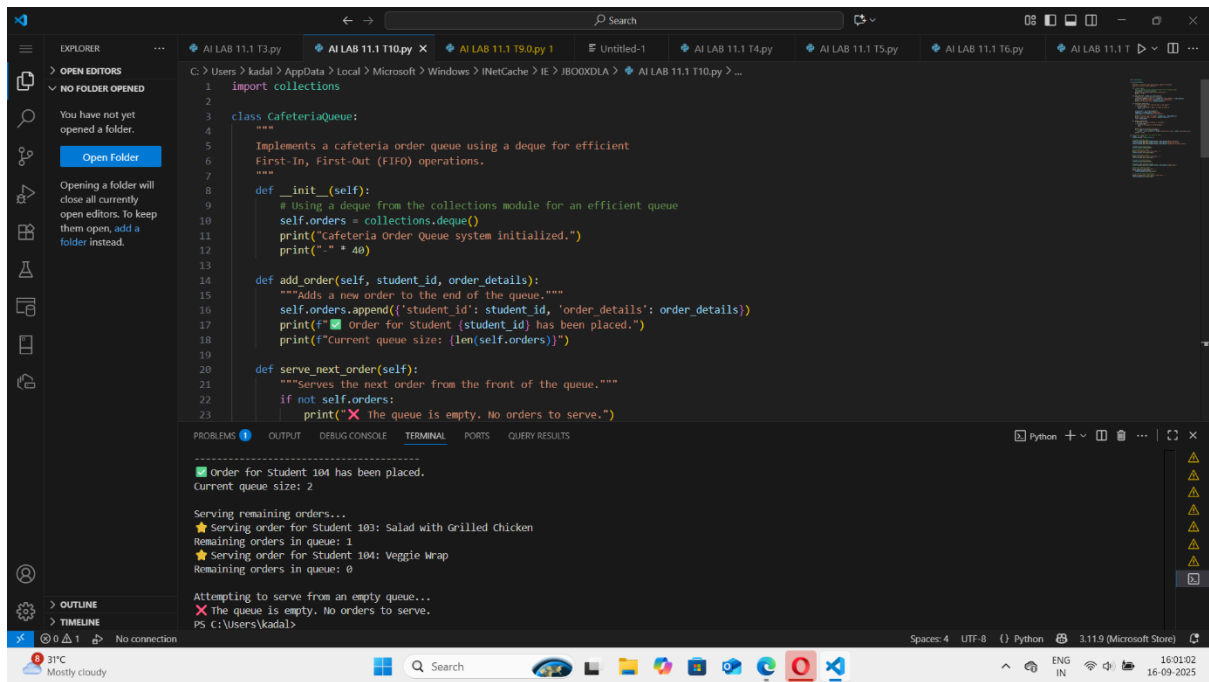
Student Task:

- 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.

CODE & OUTPUT:



```
60
61
62 # Serving the next student
63 print("\nAttempting to serve next order...")
64 cafeteria_queue.serve_next_order()
65
66 # Display the updated queue
67 cafeteria_queue.display_queue()
68
69 # Another student places an order
70 cafeteria_queue.add_order(student_id=104, order_details="Veggie Wrap")
71
72 # Serve the remaining orders
73 print("\nServing remaining orders...")
74 while cafeteria_queue.orders:
75     cafeteria_queue.serve_next_order()
76
77 # Try to serve from an empty queue
78 print("\nAttempting to serve from an empty queue...")
79 cafeteria_queue.serve_next_order()
```

PROBLEMS | OUTPUT | DEBUG CONSOLE | **TERMINAL** | PORTS | QUERY RESULTS

```
-----
[✓] Order for Student 104 has been placed.
Current queue size: 2

Serving remaining orders...
★ Serving order for Student 103: Salad with Grilled Chicken
Remaining orders in queue: 1
★ Serving order for Student 104: Veggie Wrap
Remaining orders in queue: 0

Attempting to serve from an empty queue...
✗ The queue is empty. No orders to serve.
PS C:\Users\kadal>
```

EXPLANATION:

- Initializes an empty list `queue` to store student names.



`place_order(student_name)`

- Adds a student to the end of the queue using `append()` — $O(1)$ time.



`serve_order()`

- Removes and returns the first student using `pop(0)` — $O(n)$ time due to shifting.
- Raises an error if the queue is empty.



`peek_next()`

- Returns the first student without removing them — $O(1)$ time.



`is_empty()`

- Checks if the queue is empty — $O(1)$ time.

