NAME : SHAIK SALIHA

ROLL NO : 2503A52L13

BATCH : 16

Task -1: 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.

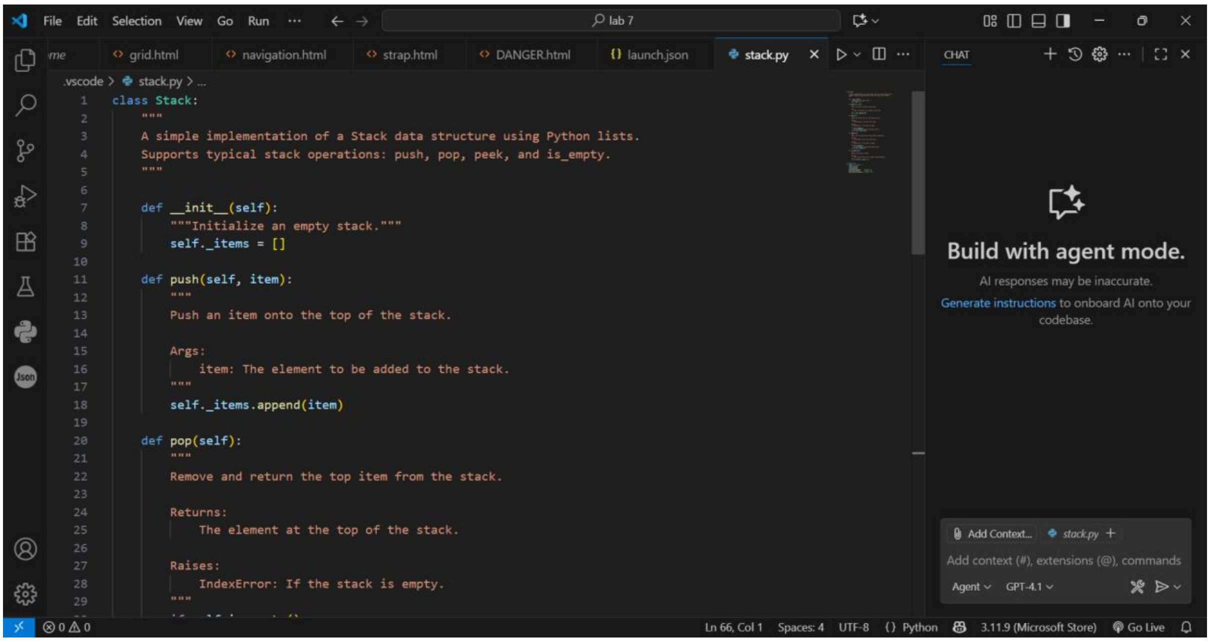
PROMPT : Generate a Stack class with push, pop, peek, and is_empty methods.

CODE : class Stack:

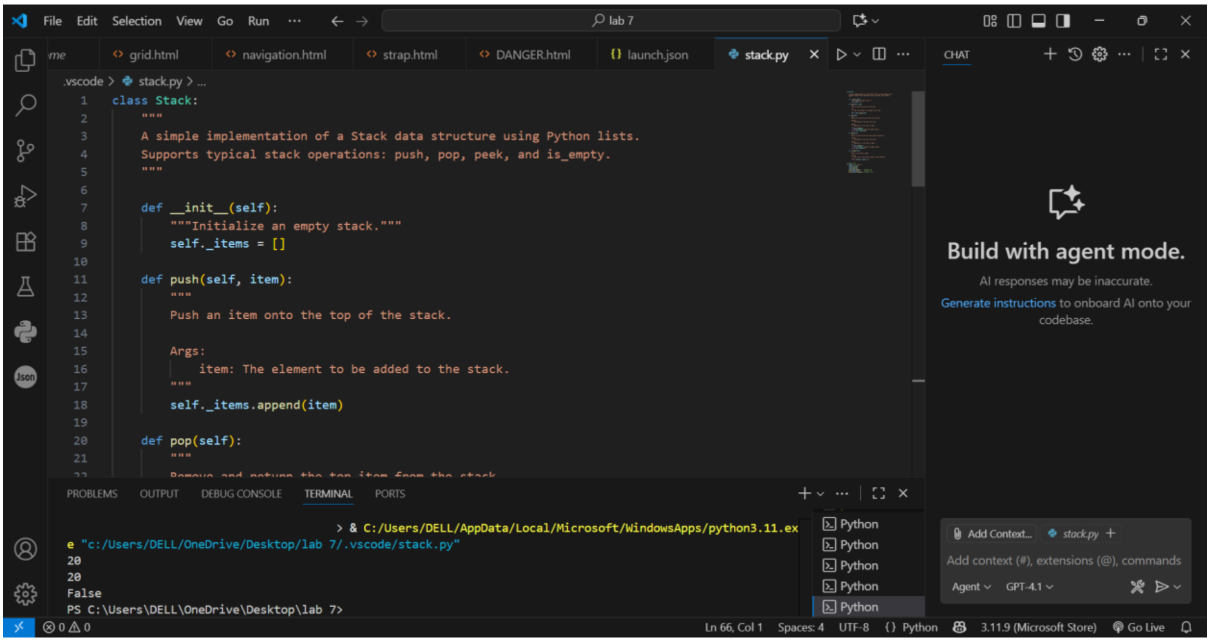
```
    """
    A simple implementation of a Stack data structure using Python lists.
    Supports typical stack operations: push, pop, peek, and is_empty.
    """

    def __init__(self):
        """Initialize an empty stack."""
        self._items = []

    def push(self, item):
        """
        Push an item onto the top of the stack.

        Args:
            item: The element to be added to the stack.
        """
        self._items.append(item)
```

```python
def pop(self):
    """
    Remove and return the top item from the stack.

    Returns:
        The element at the top of the stack.

    Raises:
        IndexError: If the stack is 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.

    Returns:
        The element at the top of the stack.

    Raises:
        IndexError: If the stack is 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.
```

```
        Returns:
            bool: True if the stack is empty, False otherwise.
        """
        return len(self._items) == 0


# Example usage:
if __name__ == "__main__":
    stack = Stack()
    stack.push(10)
    stack.push(20)
    print(stack.peek())     # Output: 20
    print(stack.pop())      # Output: 20
    print(stack.is_empty()) # Output: False
```



OUTPUT :



OBSERVATION : This is a Python Stack class using a list internally.

- **Methods included**:
  - push(item): Adds an element to the stack.
  - pop(): Removes and returns the top element (raises error if empty).
  - peek(): Returns the top element without removing it.
  - is_empty(): Checks if the stack is empty.

Task -2: 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.

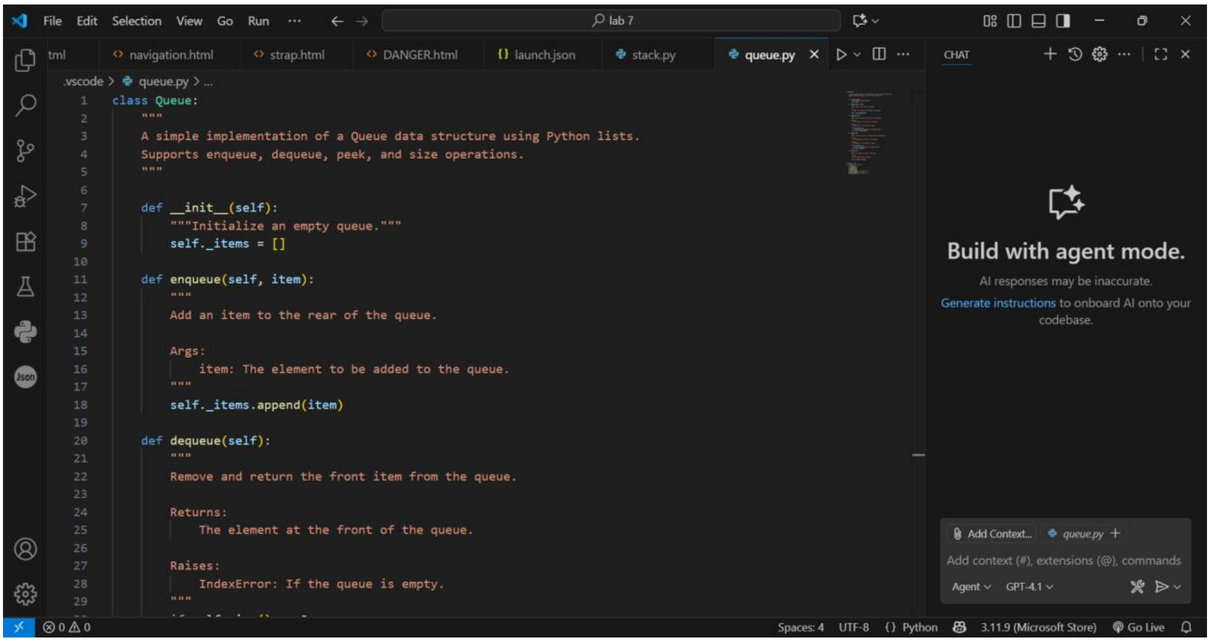
PROMPT : Implement a Queue using Python lists.

CODE : class Queue:

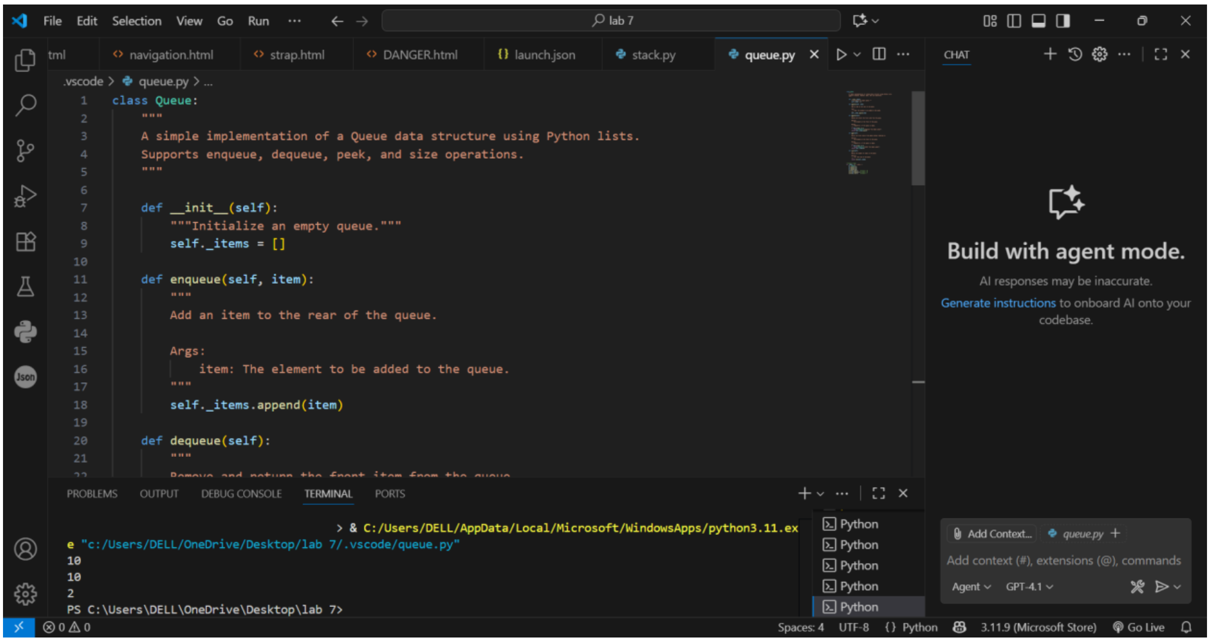
```
    """
    A simple implementation of a Queue data structure using Python lists.
    Supports enqueue, dequeue, peek, and size operations.
    """

    def __init__(self):
        """Initialize an empty queue."""
        self._items = []

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

        Args:
            item: The element to be added to the queue.
        """
        self._items.append(item)

    def dequeue(self):
        """
        Remove and return the front item from the queue.

        Returns:
            The element at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.size() == 0:
            raise IndexError("dequeue from empty queue")
```

```
        return self._items.pop(0)

    def peek(self):
        """
        Return the front item of the queue without removing it.

        Returns:
            The element at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.size() == 0:
            raise IndexError("peek from empty queue")
        return self._items[0]

    def size(self):
        """
        Return the number of items in the queue.

        Returns:
            int: The size of the queue.
        """
        return len(self._items)


# Example usage:
if __name__ == "__main__":
    q = Queue()
    q.enqueue(10)
    q.enqueue(20)
```

```
q.enqueue(30)
print(q.peek())   # Output: 10
print(q.dequeue())# Output: 10
print(q.size())   # Output: 2
```



OUTPUT :



OBSERVATION : This is a Python Queue class using a list with FIFO behavior.

- **Methods included**:
    - enqueue(item): Adds an element at the rear.
    - dequeue(): Removes and returns the front element (error if empty).
    - peek(): Shows the front element without removing it.
    - size(): Returns the number of items in the queue.

Task -3: 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

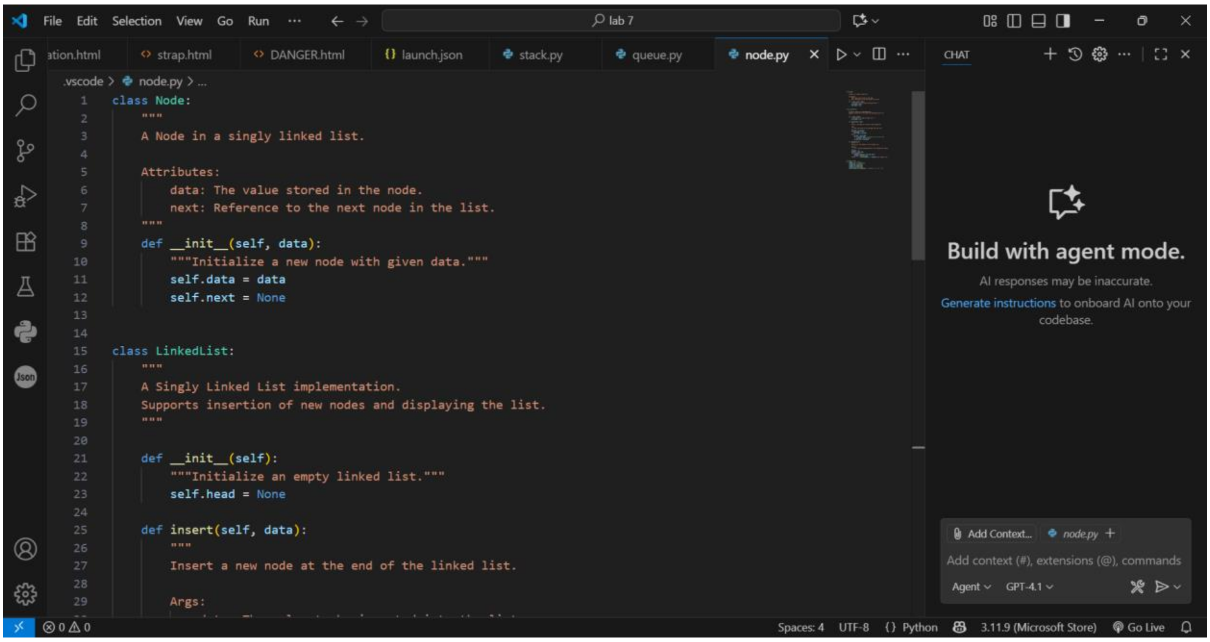
PROMPT : Generate a Singly Linked List with insert and display methods.
Sample Input Code:
class Node:
pass
class LinkedList:
pass
Expected Output:
• A working linked list implementation with clear method
documentation

CODE : class Node:

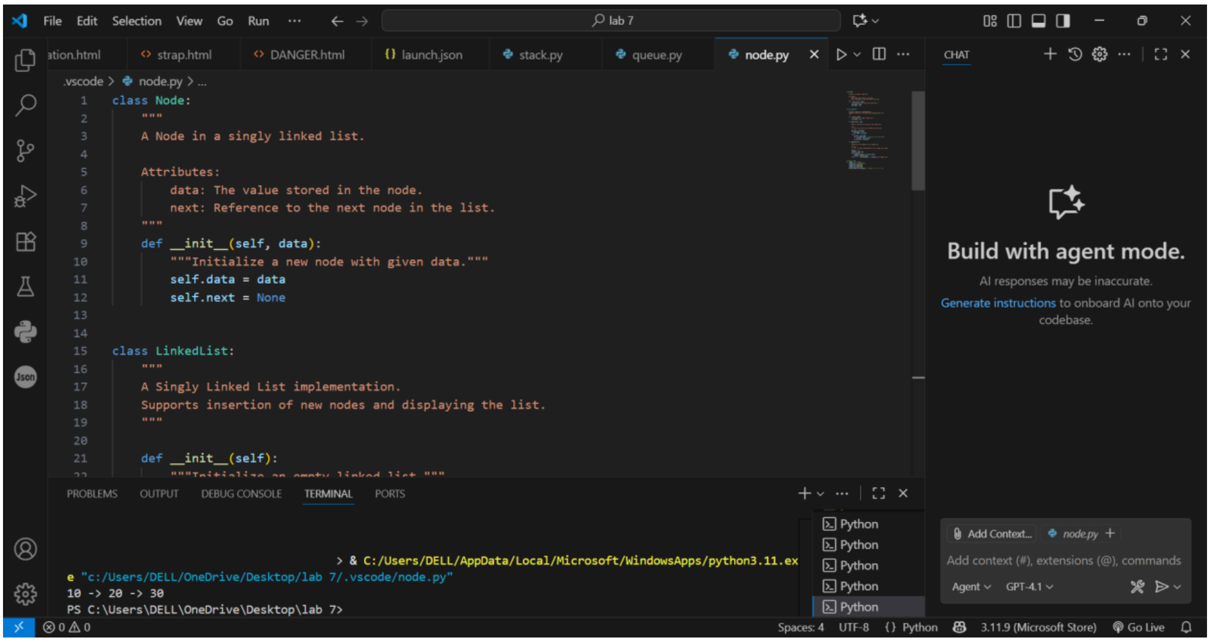
```
    """
    A Node in a singly linked list.

    Attributes:
        data: The value stored in the node.
        next: Reference to the next node in the list.
    """
    def __init__(self, data):
```

```python
        """Initialize a new node with given data."""
        self.data = data
        self.next = None


class LinkedList:
    """
    A Singly Linked List implementation.
    Supports insertion of new nodes and displaying the list.
    """

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """
        Insert a new node at the end of the linked list.

        Args:
            data: The value to be inserted into the list.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:  # Traverse to the last node
                current = current.next
            current.next = new_node
```

```python
    def display(self):
        """
        Display all the elements of the linked list.

        Returns:
            str: A string representation of the linked list values.
        """
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        return " -> ".join(elements) if elements else "Empty List"


# Example usage:
if __name__ == "__main__":
    linked_list = LinkedList()
    linked_list.insert(10)
    linked_list.insert(20)
    linked_list.insert(30)
    print(linked_list.display())  # Output: 10 -> 20 -> 30
```



OUTPUT :



OBSERVATION : This is a **singly linked list** implementation in Python using a Node class and a LinkedList class.

- **Node class**: Stores data and a reference to the next node.
- **LinkedList class methods**:
    - insert(data): Adds a new node at the end.

- display(): Returns a string showing the list elements in order.

Task -4: 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

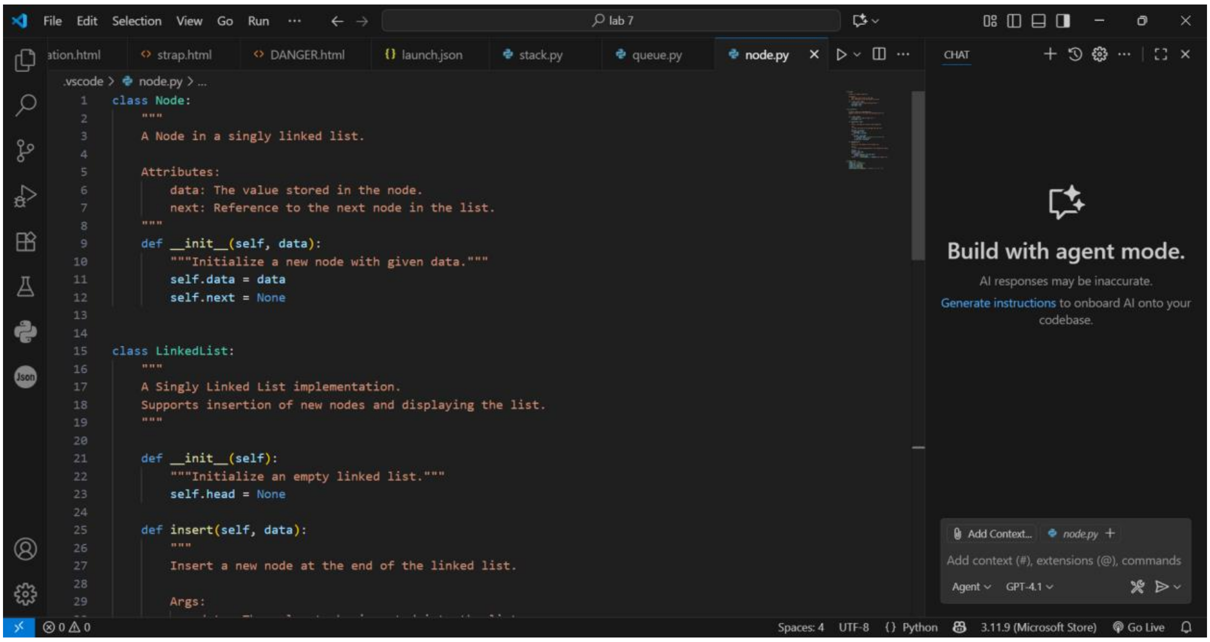
PROMPT : create a BST with insert and in-order traversal methods.

CODE : class Node:

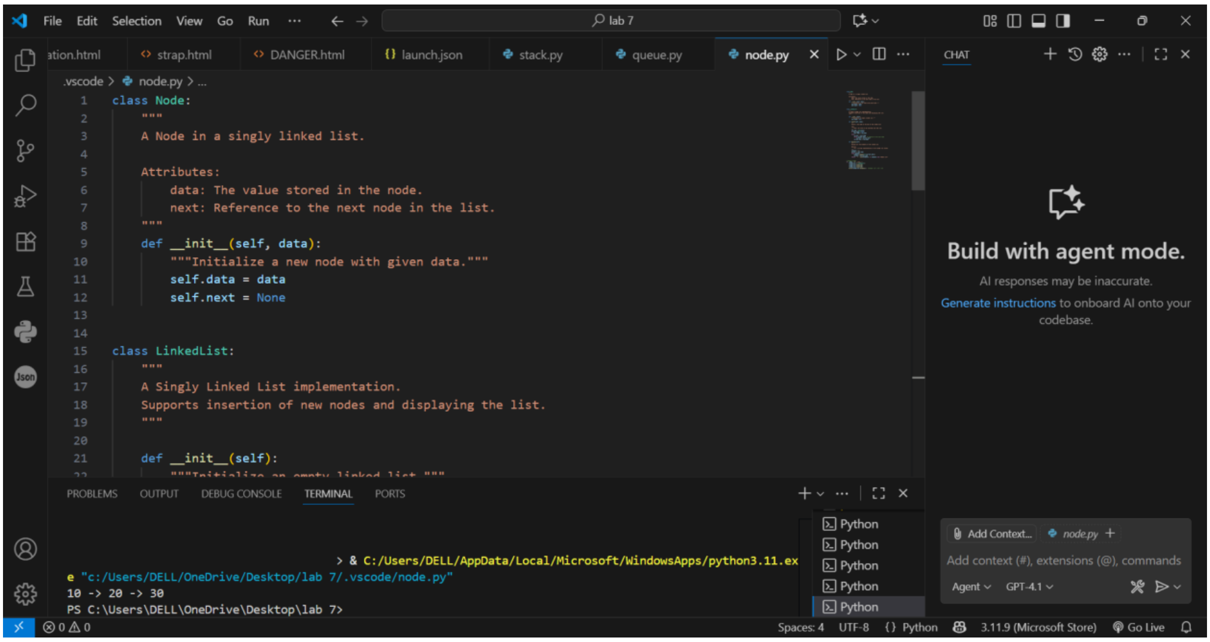
```
    """
    A Node in a Binary Search Tree.

    Attributes:
        data: The value stored in the node.
        left: Reference to the left child node.
        right: Reference to the right child node.
    """
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BST:
    """
    A Binary Search Tree (BST) implementation.
    Supports recursive insertion and in-order traversal.
    """
```

```
def __init__(self):
    """Initialize an empty BST."""
    self.root = None

def insert(self, value):
    """
    Insert a new value into the BST.

    Args:
        value: The value to be inserted.
    """
    self.root = self._insert_recursive(self.root, value)

def _insert_recursive(self, node, value):
    """Helper method to insert a value recursively."""
    if node is None:
        return Node(value)
    if value < node.data:
        node.left = self._insert_recursive(node.left, value)
    elif value > node.data:
        node.right = self._insert_recursive(node.right, value)
    # If value == node.data, do nothing (no duplicates in this BST)
    return node

def inorder_traversal(self):
    """
    Perform in-order traversal of the BST.

    Returns:
        list: Values of the BST in sorted order.
    """
```

```
        return self._inorder_recursive(self.root)

    def _inorder_recursive(self, node):
        """Helper method to recursively perform in-order traversal."""
        result = []
        if node:
            result.extend(self._inorder_recursive(node.left))
            result.append(node.data)
            result.extend(self._inorder_recursive(node.right))
        return result


# Example usage
if __name__ == "__main__":
    bst = BST()
    bst.insert(50)
    bst.insert(30)
    bst.insert(70)
    bst.insert(20)
    bst.insert(40)
    bst.insert(60)
    bst.insert(80)

    print("In-order Traversal:", bst.inorder_traversal())
    # Output: In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
```



OUTPUT :



OBSERVATION : This is a **Binary Search Tree (BST)** implementation in Python with recursive methods.

- **Node class**: Stores data, left child, and right child.
- **BST class methods**:
    - insert(value): Recursively inserts a value into the BST (ignores duplicates).

- inorder_traversal(): Returns a sorted list of values using in-order traversal.

Task -5: Use AI to implement a hash table with basic insert, search, and deletemethods.
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 deletemethods.

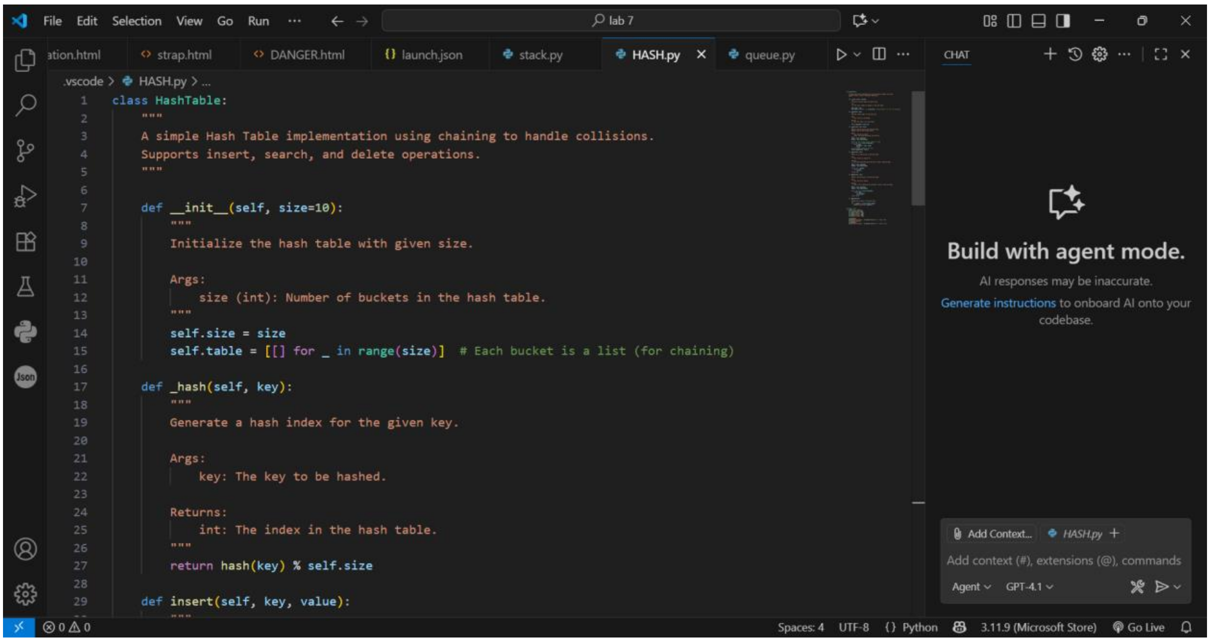
CODE : class HashTable:

```
    """
    A simple Hash Table implementation using chaining to handle collisions.
    Supports insert, search, and delete operations.
    """

    def __init__(self, size=10):
        """
        Initialize the hash table with given size.

        Args:
            size (int): Number of buckets in the hash table.
        """
        self.size = size
        self.table = [[] for _ in range(size)]  # Each bucket is a list (for chaining)

    def _hash(self, key):
        """
        Generate a hash index for the given key.
```

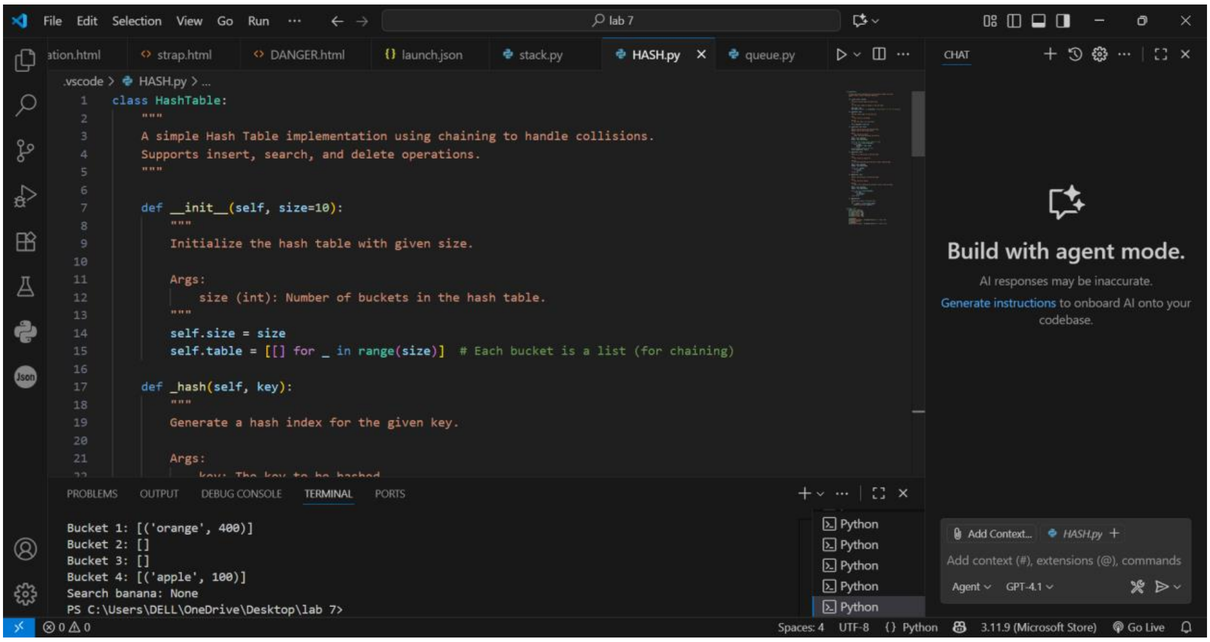
```
        Args:
            key: The key to be hashed.

        Returns:
            int: The index in the hash table.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Insert a key-value pair into the hash table.
        Updates value if key already exists.

        Args:
            key: The key to insert.
            value: The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]

        # Check if key already exists, update if found
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        # If not found, append as new entry
        bucket.append((key, value))

    def search(self, key):
        """
        Search for a value by key in the hash table.
```

```
        Args:
            key: The key to search for.

        Returns:
            The value associated with the key if found, otherwise None.
        """
        index = self._hash(key)
        bucket = self.table[index]

        for (k, v) in bucket:
            if k == key:
                return v
        return None

    def delete(self, key):
        """
        Delete a key-value pair from the hash table.

        Args:
            key: The key to delete.

        Returns:
            bool: True if deletion was successful, False if key not found.
        """
        index = self._hash(key)
        bucket = self.table[index]

        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
```

```python
            return True
        return False

    def display(self):
        """
        Display the contents of the hash table.
        """
        for i, bucket in enumerate(self.table):
            print(f"Bucket {i}: {bucket}")


# Example usage:
if __name__ == "__main__":
    ht = HashTable(size=5)
    ht.insert("apple", 100)
    ht.insert("banana", 200)
    ht.insert("grape", 300)
    ht.insert("orange", 400)

    ht.display()
    print("Search banana:", ht.search("banana"))  # Output: 200
    ht.delete("banana")
    ht.display()
    print("Search banana:", ht.search("banana"))  # Output: None
```



OUTPUT :



OBSERVATION : This is a **hash table implementation** in Python using **chaining** to handle collisions.

- **Structure**: An array of buckets, each bucket is a list of (key, value) pairs.
- **Methods**:
    - insert(key, value): Adds or updates a key-value pair.

- search(key): Finds the value for a given key, returns None if not found.
- delete(key): Removes a key-value pair, returns True/False based on success.
- display(): Prints the contents of the hash table.

Task -6: 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

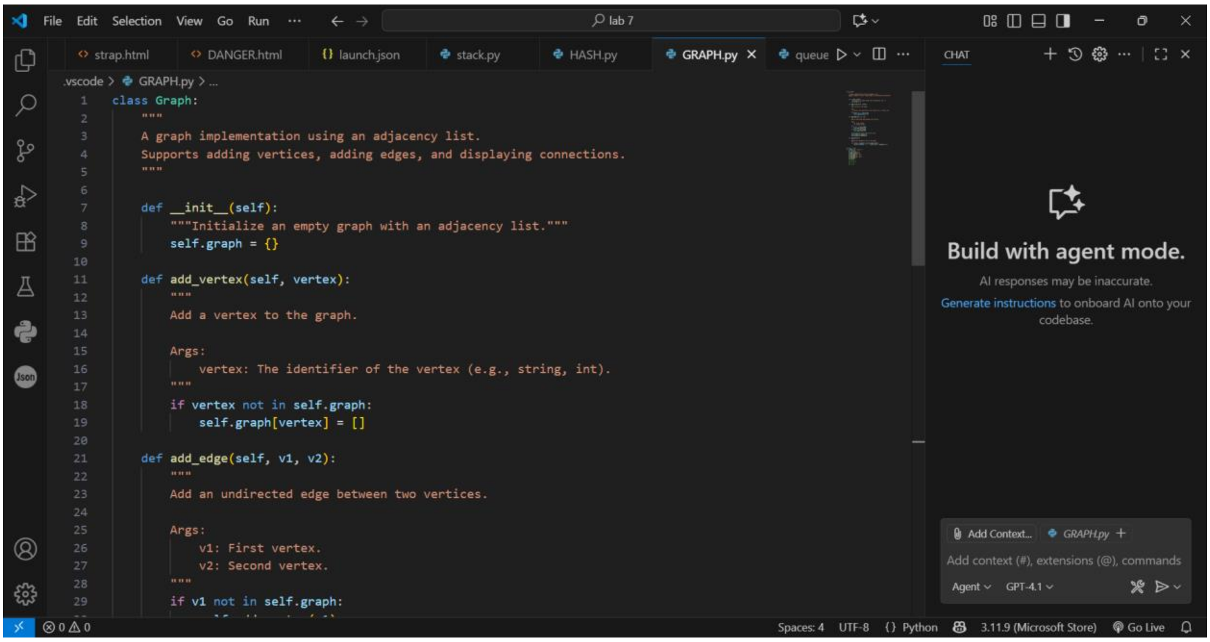
PROMPT : implement a graph using an adjacency list.

CODE : class Graph:

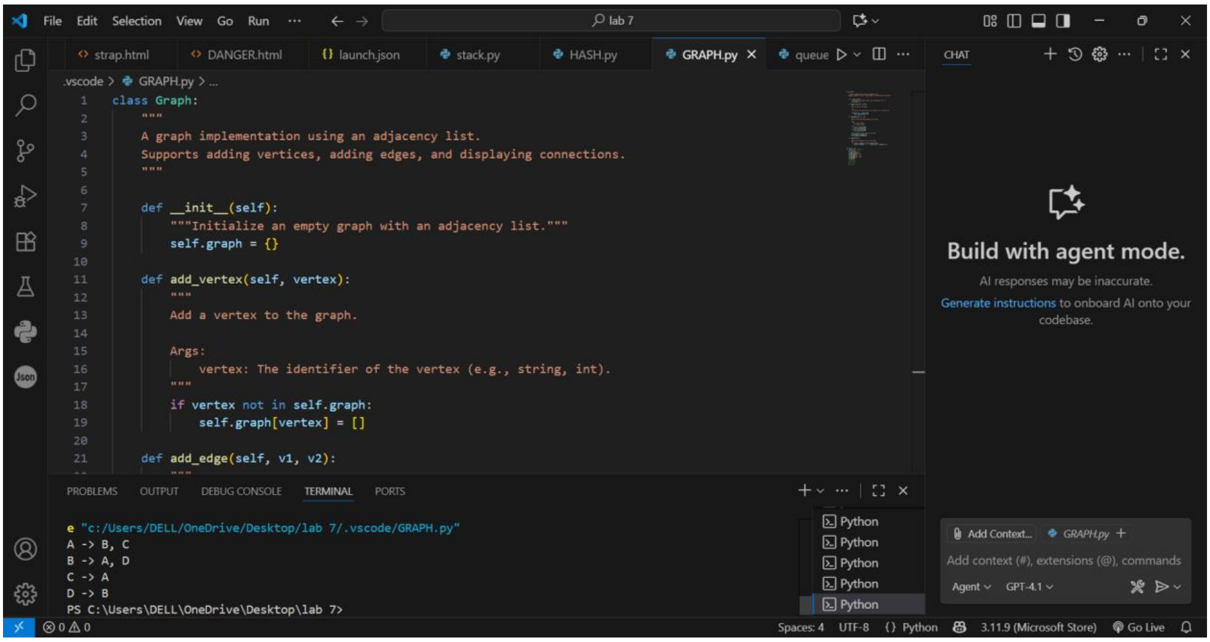
```
    """
    A graph implementation using an adjacency list.
    Supports adding vertices, adding edges, and displaying connections.
    """

    def __init__(self):
        """Initialize an empty graph with an adjacency list."""
        self.graph = {}

    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.

        Args:
            vertex: The identifier of the vertex (e.g., string, int).
        """
        if vertex not in self.graph:
            self.graph[vertex] = []
```

```python
    def add_edge(self, v1, v2):
        """
        Add an undirected edge between two vertices.

        Args:
            v1: First vertex.
            v2: Second vertex.
        """
        if v1 not in self.graph:
            self.add_vertex(v1)
        if v2 not in self.graph:
            self.add_vertex(v2)

        # Add edge both ways (undirected graph)
        self.graph[v1].append(v2)
        self.graph[v2].append(v1)

    def display(self):
        """
        Display the adjacency list of the graph.
        """
        for vertex, neighbors in self.graph.items():
            print(f"{vertex} -> {', '.join(map(str, neighbors))}")


# Example usage
if __name__ == "__main__":
    g = Graph()
    g.add_vertex("A")
    g.add_vertex("B")
```

```python
g.add_edge("A", "B")
g.add_edge("A", "C")
g.add_edge("B", "D")
g.display()
# Output:
# A -> B, C
# B -> A, D
# C -> A
# D -> B
```



OUTPUT :



OBSERVATION : This is a **graph implementation** in Python using an **adjacency list**.

- **Graph structure**: A dictionary where each vertex maps to a list of its neighbors.
- **Methods**:
    - add_vertex(vertex): Adds a new vertex if it does not exist.
    - add_edge(v1, v2): Adds an undirected edge between two vertices (creates vertices if missing).
    - display(): Prints all vertices and their connected neighbors.

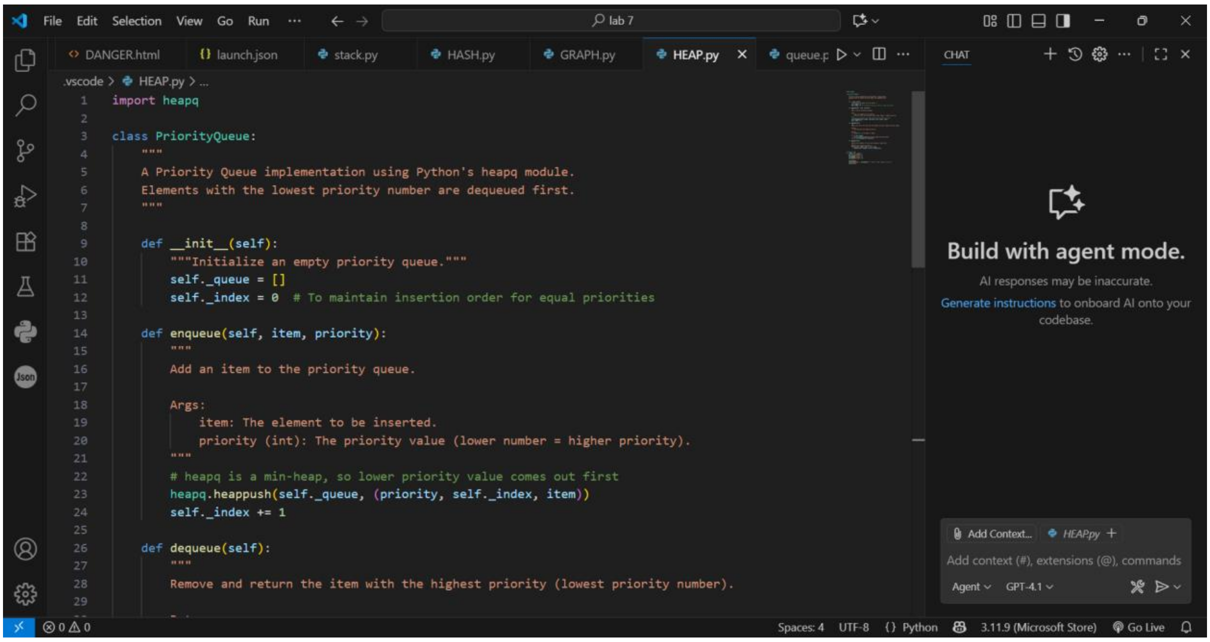
Task -7: Use AI to implement a priority queue using Python's heapq module.
Sample Input Code:
class PriorityQueue:
pass
Expected Output:
• Implementation with enqueue (priority), dequeue (highest priority), and display methods.

PROMPT : Implement a priority queue using Python's heapq module.

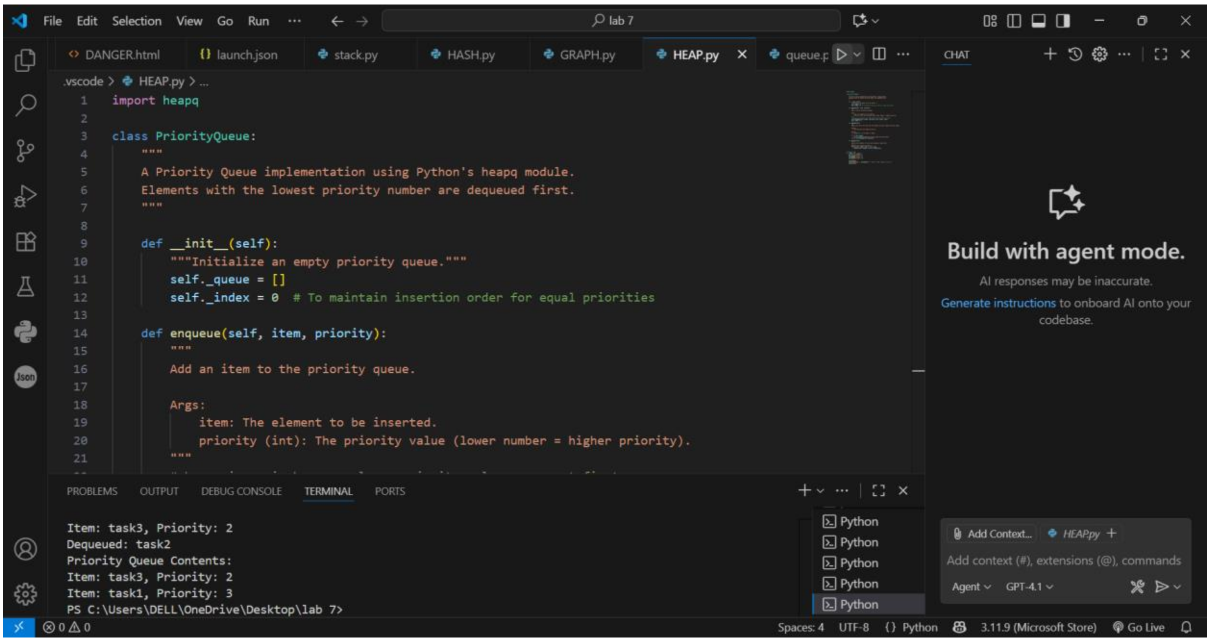
CODE : import heapq

```
class PriorityQueue:
    """
    A Priority Queue implementation using Python's heapq module.
    Elements with the lowest priority number are dequeued first.
    """

    def __init__(self):
        """Initialize an empty priority queue."""
        self._queue = []
        self._index = 0  # To maintain insertion order for equal priorities

    def enqueue(self, item, priority):
        """
        Add an item to the priority queue.

        Args:
            item: The element to be inserted.
            priority (int): The priority value (lower number = higher priority).
        """
        # heapq is a min-heap, so lower priority value comes out first
        heapq.heappush(self._queue, (priority, self._index, item))
        self._index += 1

    def dequeue(self):
        """
        Remove and return the item with the highest priority (lowest priority number).

        Returns:
            The item with the highest priority.

        Raises:
```

```
            IndexError: If the queue is empty.
        """
        if not self._queue:
            raise IndexError("dequeue from an empty priority queue")
        return heapq.heappop(self._queue)[2]

    def display(self):
        """
        Display the elements of the priority queue in heap order.
        """
        print("Priority Queue Contents:")
        for priority, index, item in self._queue:
            print(f"Item: {item}, Priority: {priority}")


# Example usage
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.enqueue("task1", 3)
    pq.enqueue("task2", 1)
    pq.enqueue("task3", 2)

    pq.display()
    print("Dequeued:", pq.dequeue())  # Output: task2 (highest priority)
    pq.display()
```



OUTPUT :



OBSERVATION : This is a **Priority Queue implementation** in Python using **heapq** (min-heap).

- **Data structure**: A heap storing (priority, index, item) to keep order when priorities match.
- **Methods**:
    - enqueue(item, priority): Adds an item with a given priority (lower = higher priority).

- dequeue(): Removes and returns the item with the highest priority.
- display(): Prints all elements with their priorities.

Task -8: 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

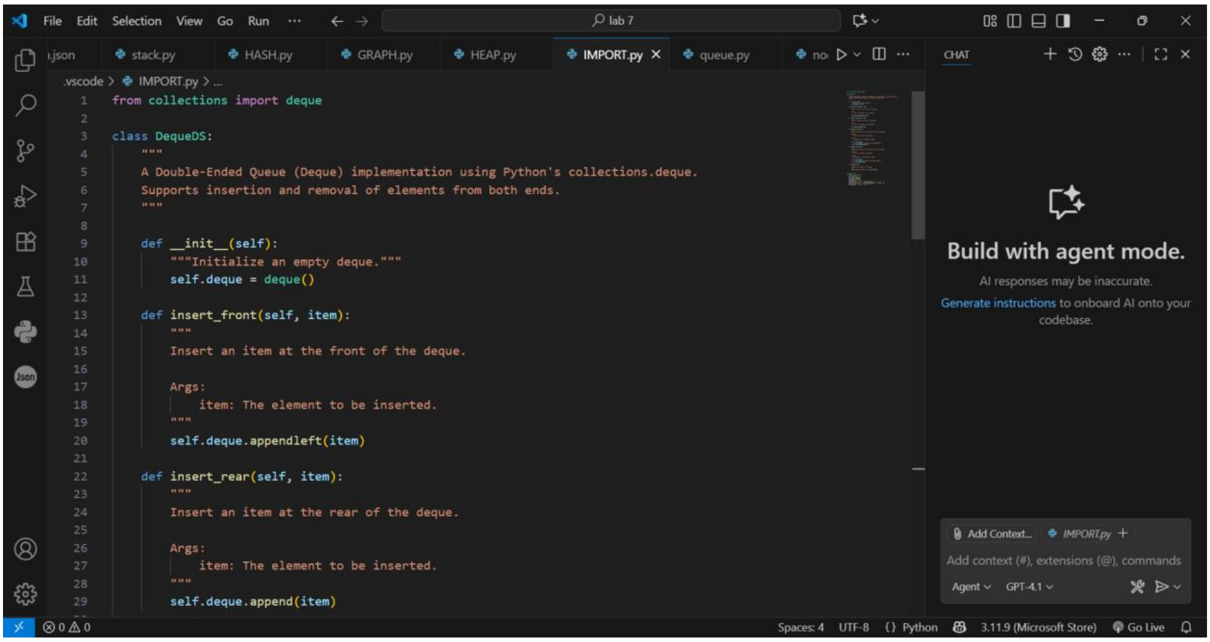
PROMPT : Implement a double-ended queue using collections.deque.

CODE : from collections import deque

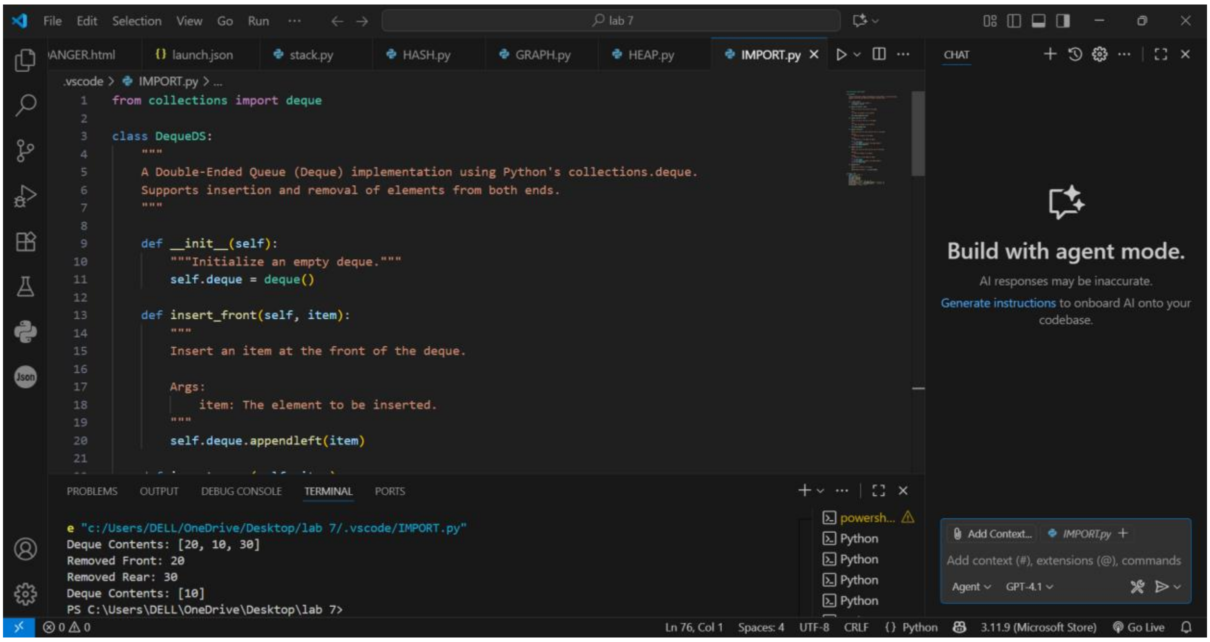
```
class DequeDS:
    """
    A Double-Ended Queue (Deque) implementation using Python's collections.deque.
    Supports insertion and removal of elements from both ends.
    """

    def __init__(self):
        """Initialize an empty deque."""
        self.deque = deque()

    def insert_front(self, item):
        """
        Insert an item at the front of the deque.

        Args:
            item: The element to be inserted.
        """
```

```
        self.deque.appendleft(item)

    def insert_rear(self, item):
        """
        Insert an item at the rear of the deque.

        Args:
            item: The element to be inserted.
        """
        self.deque.append(item)

    def remove_front(self):
        """
        Remove and return an item from the front of the deque.

        Returns:
            The front element of the deque.

        Raises:
            IndexError: If the deque is empty.
        """
        if not self.deque:
            raise IndexError("remove from empty deque")
        return self.deque.popleft()

    def remove_rear(self):
        """
        Remove and return an item from the rear of the deque.

        Returns:
            The rear element of the deque.
```

```
        Raises:
            IndexError: If the deque is empty.
        """
        if not self.deque:
            raise IndexError("remove from empty deque")
        return self.deque.pop()

    def display(self):
        """
        Display the contents of the deque.
        """
        print("Deque Contents:", list(self.deque))


# Example usage
if __name__ == "__main__":
    dq = DequeDS()
    dq.insert_rear(10)
    dq.insert_front(20)
    dq.insert_rear(30)
    dq.display()  # Output: [20, 10, 30]
    print("Removed Front:", dq.remove_front())  # Output: 20
    print("Removed Rear:", dq.remove_rear())    # Output: 30
    dq.display()  # Output: [10]
```



OUTPUT :



OBSERVATION : This code defines a **DequeDS class** that uses Python's collections.deque to implement a double-ended queue.

- You can **insert and remove** elements from both the **front** and **rear**.
- It provides **peek methods** to check elements at both ends without removing them.
- It includes **is_empty()** to check if the deque is empty and **size()** to get the number of elements.

Task -9: 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
Expected Output:
• A markdown table with structure names, operations, and complexities

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

CODE :

Data Structure	**Insertion Complexity**	**Deletion Complexity**	**Access Complexity**	**Description/Use Cases**
Stack	O(1)	O(1)	O(1) (top)	LIFO structure; used in function call expression evaluation
Queue	O(1)	O(1)	O(1) (front/rear)	FIFO structure; used in scheduling, buffering, BFS
Linked List	O(1) (head)	O(1) (head)	O(n)	Dynamic size, efficient insertion/del at ends
Array	O(n) (worst)	O(n) (worst)	O(1)	Fixed size; fast access but costly insertion/deletion
Doubly Linked List	O(1) (ends)	O(1) (ends)	O(n)	Like linked list but bidirectional; use deques
Hash Table	O(1) average	O(1) average	O(1) average	Key-value mapping; very fast access insertion, and deletion
Binary Search Tree (BST)	O(log n) avg	O(log n) avg	O(log n) avg	Sorted data, efficient search, inserti and deletion
Priority Queue (H eap)	O(log n)	O(log n)	O(1) (min/max)	Efficient retrieval of min/max eleme

- OBSERVATION : Stack: Push/pop operations run in constant time as elements are added/removed from the top.
- Queue: Enqueue/dequeue operations run in constant time adding at rear and removing from front.
- Linked List: Inserting and deleting at head or tail is efficient, but access is linear time.
- Array: Provides constant time access but insertion and deletion require shifting elements.
- Doubly Linked List: Supports efficient insertions/removals at both ends, crucial for deque implementation.
- Hash Table: Provides very fast average time for dictionary-like lookups and updates.
- BST: Balanced BSTs offer logarithmic time complexity for search, insert, and delete.
- Priority Queue: Maintains elements with priority using heaps, allowing quick access to min/max.