ASSIGNMENT 11.1

Data Structures with AI: Implementing Fundamental Structures

A.NagaKoushik

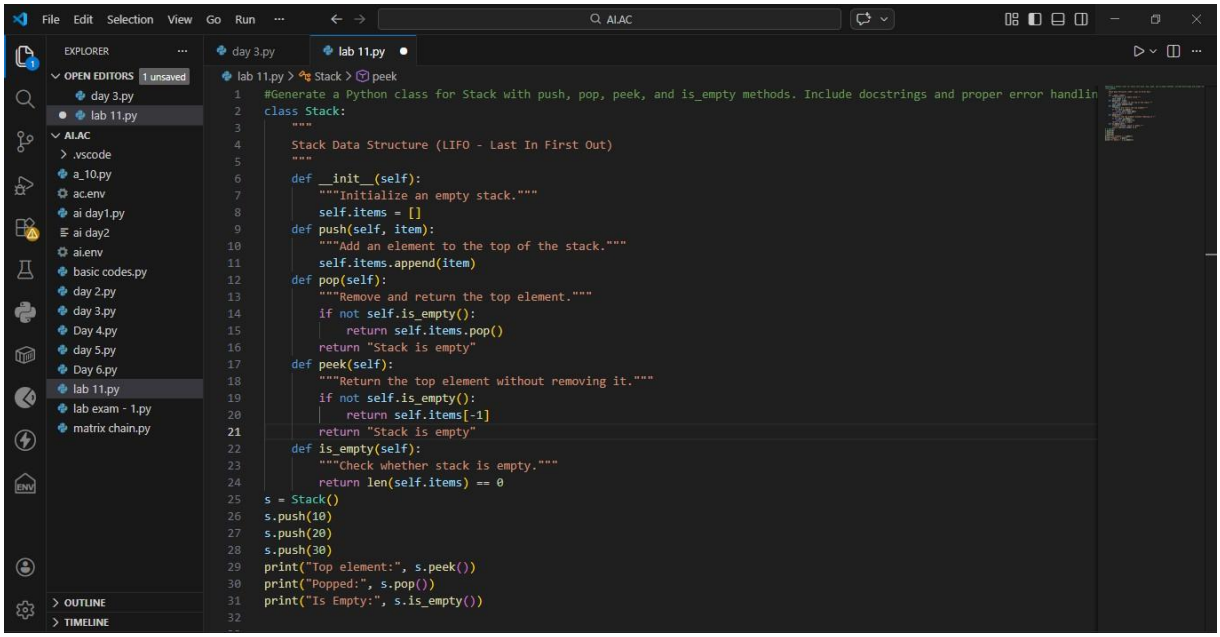
2403A51L22

Batch-51

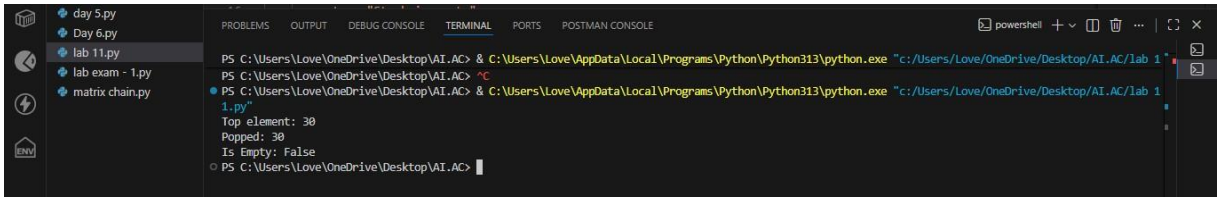
Task 1: Stack Implementation

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

Prompt: Generate a Python class for Stack with push, pop, peek, and is_empty methods. Include docstrings and proper error handling.



OUTPUT:

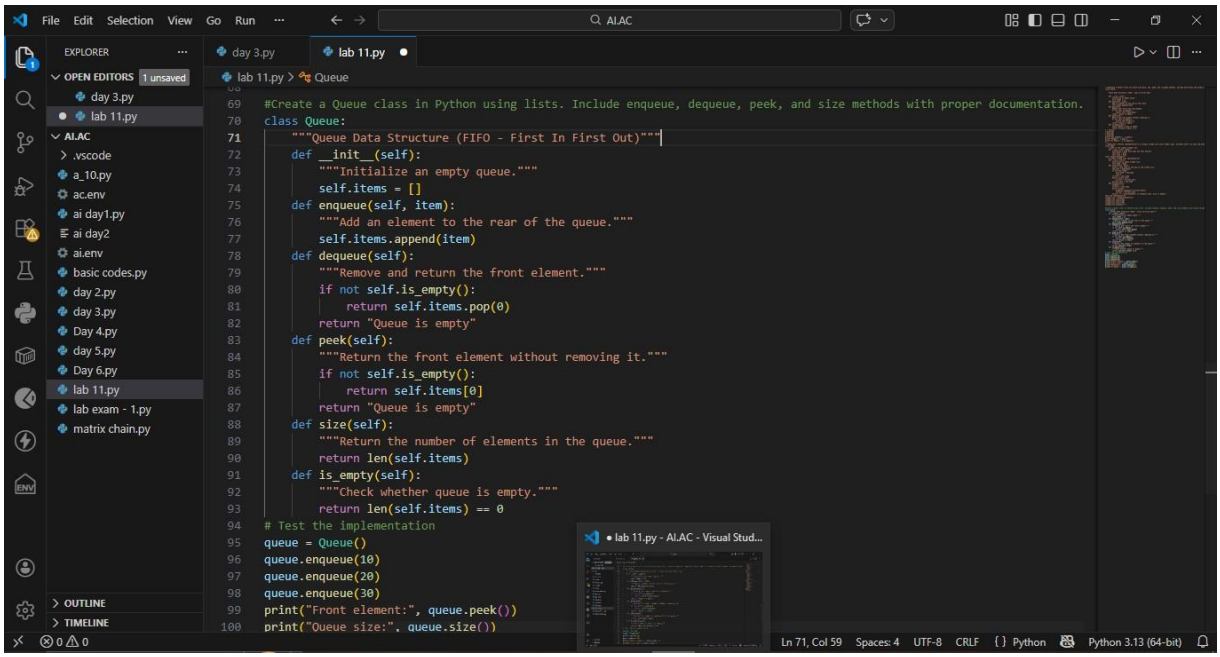


Explanation: A Stack is a linear data structure that follows the LIFO (Last In First Out) principle, where the last element inserted is the first one removed. Operations such as push, pop, and peek are performed at one end called the top. It is commonly used in function calls, undo operations, and expression evaluation.

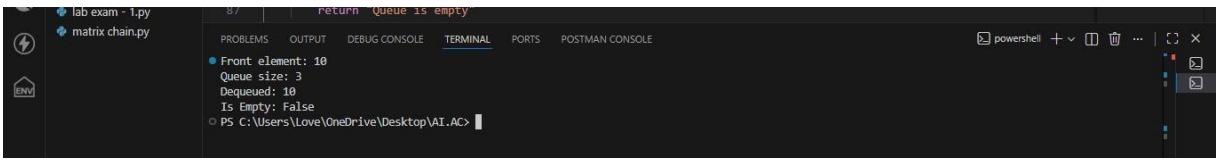
Task Description #2: Queue Implementation

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

Prompt: Create a Queue class in Python using lists. Include enqueue,dequeue, peek, and size methods with proper documentation.



OUTPUT:

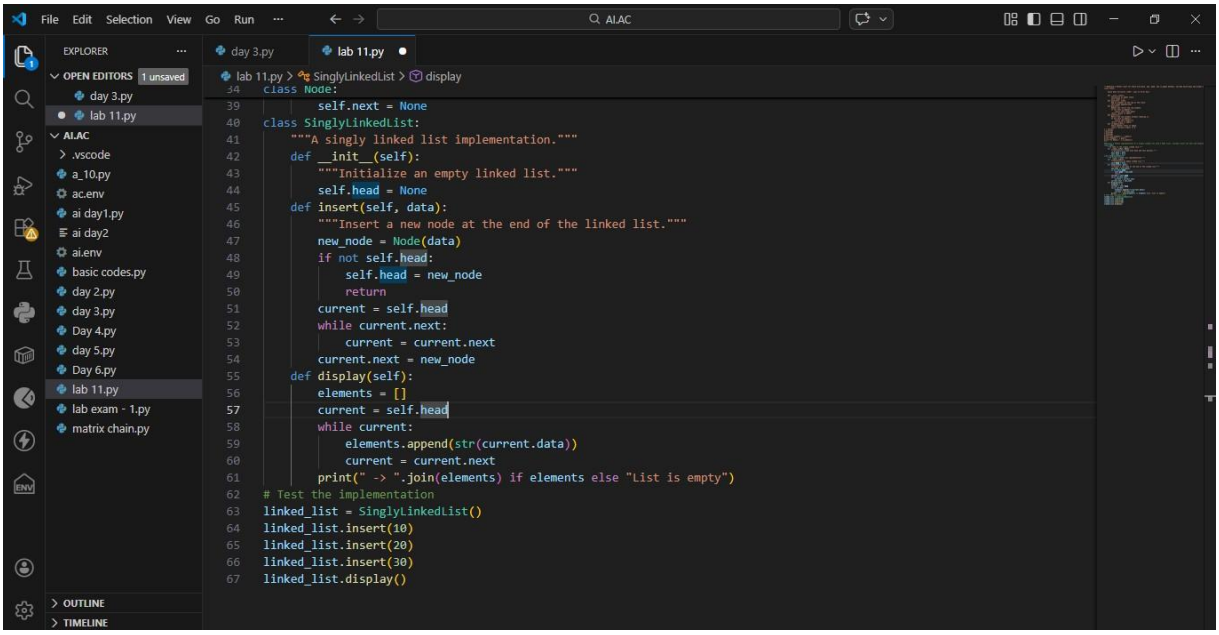


Explanation: A Queue is a linear data structure that follows the FIFO (First In First Out) principle. This means the first element inserted is the first one removed.

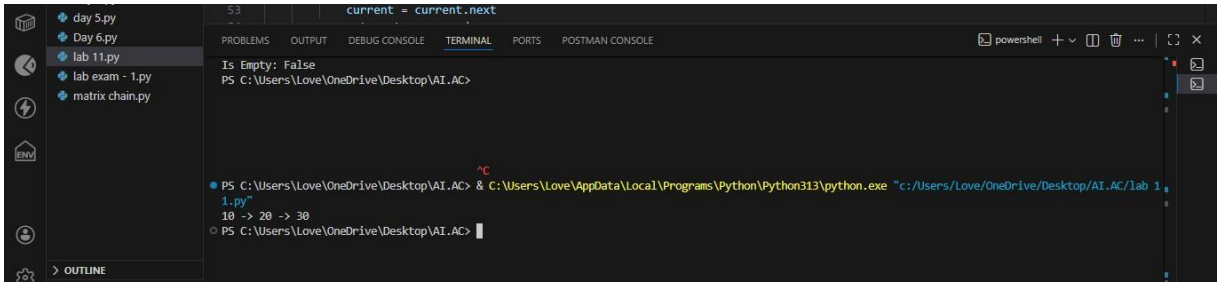
Task Description #3: Linked List

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

Prompt : Generate a Python implementation of a Singly Linked List with a Node class. Include insert (at end) and display methods with docstrings.



OUTPUT:

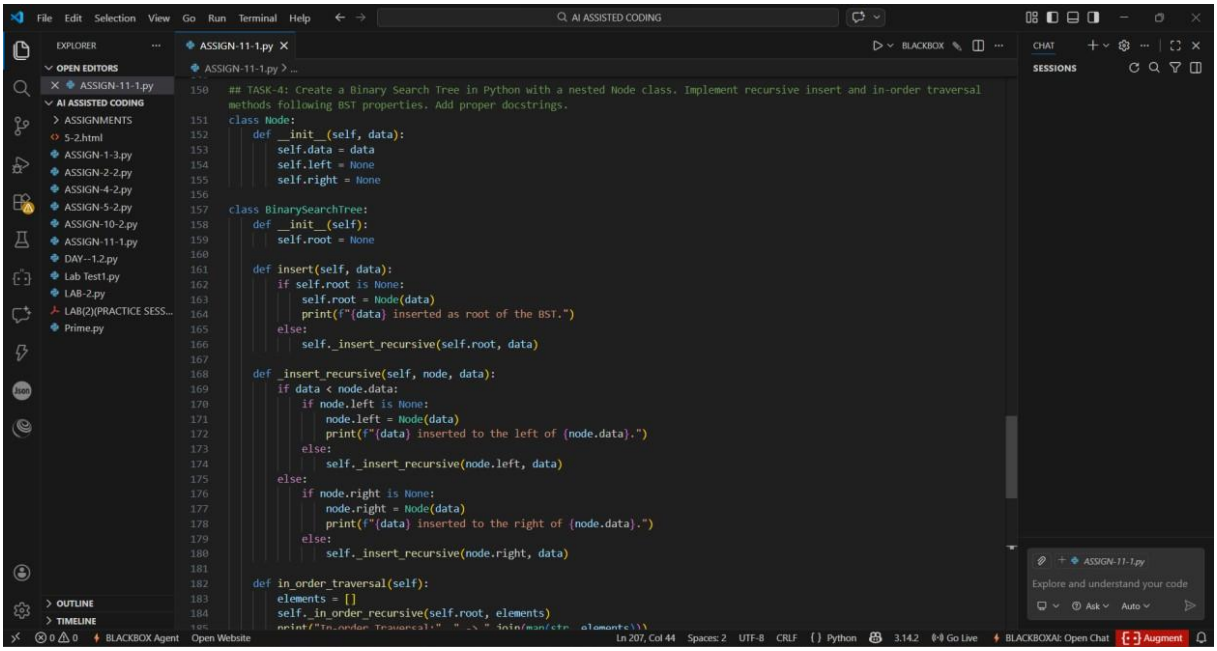


Explanation: A Singly Linked List is a dynamic data structure where elements (nodes) are connected using pointers. Linked Lists are useful when frequent insertions and deletions are required, as they do not require shifting elements like arrays.

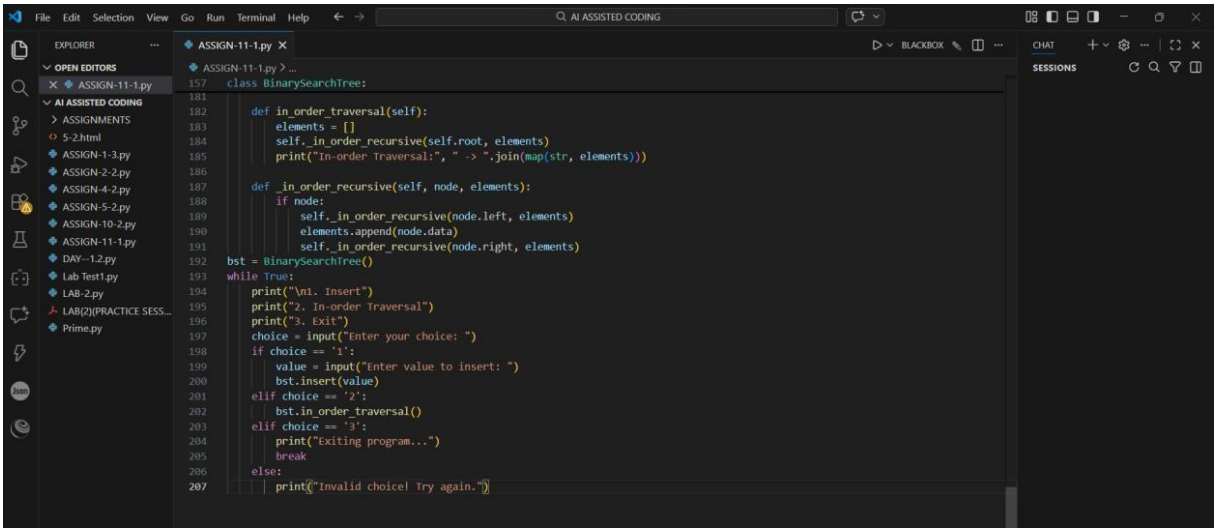
Task Description #4: Binary Search Tree (BST)

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

Prompt: Create a Binary Search Tree in Python with recursive insert and inorder traversal methods. Include proper class structure and documentation.

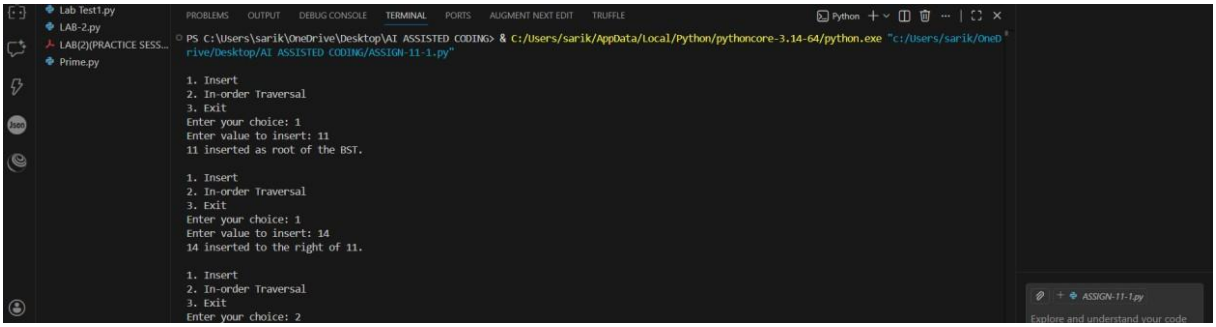


```python
## TASK-4: Create a Binary Search Tree in Python with a nested Node class. Implement recursive insert and in-order traversal
methods following BST properties. Add proper docstrings.
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
            print(f"{data} inserted as root of the BST.")
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = Node(data)
                print(f"{data} inserted to the left of {node.data}.")
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = Node(data)
                print(f"{data} inserted to the right of {node.data}.")
            else:
                self._insert_recursive(node.right, data)

    def in_order_traversal(self):
        elements = []
        self._in_order_recursive(self.root, elements)
```



```python
class BinarySearchTree:

    def in_order_traversal(self):
        elements = []
        self._in_order_recursive(self.root, elements)
        print("In-order Traversal:", " -> ".join(map(str, elements)))

    def _in_order_recursive(self, node, elements):
        if node:
            self._in_order_recursive(node.left, elements)
            elements.append(node.data)
            self._in_order_recursive(node.right, elements)
bst = BinarySearchTree()
while True:
    print("\n1. Insert")
    print("2. In-order Traversal")
    print("3. Exit")
    choice = input("Enter your choice: ")
    if choice == '1':
        value = input("Enter value to insert: ")
        bst.insert(value)
    elif choice == '2':
        bst.in_order_traversal()
    elif choice == '3':
        print("Exiting program...")
        break
    else:
        print("Invalid choice! Try again.")
```

OUTPUT:



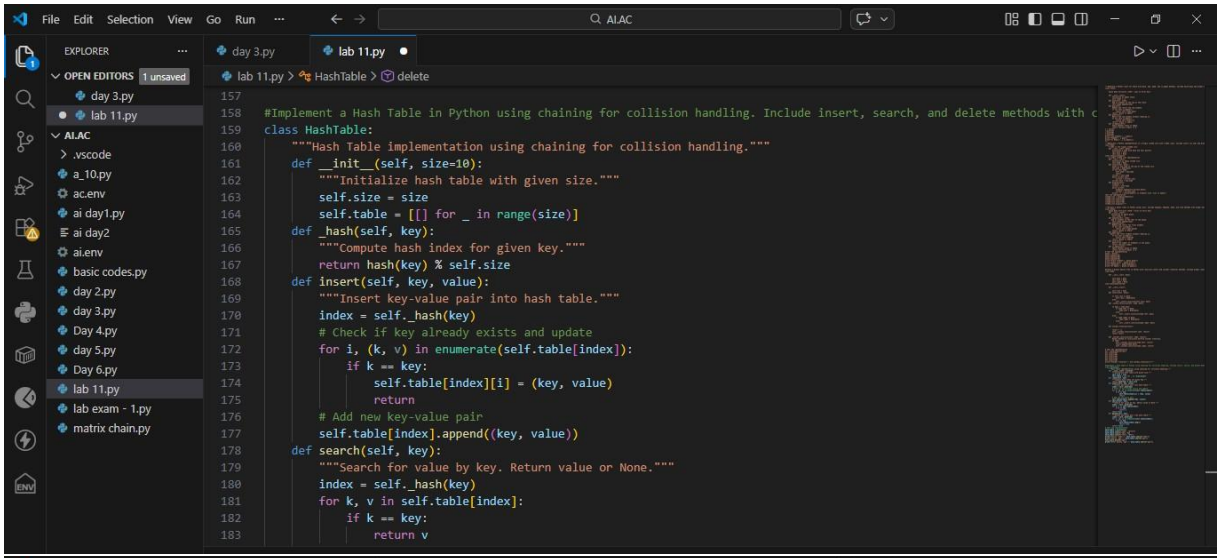
```
PS C:\Users\sarik\OneDrive\Desktop\AI ASSISTED CODING> & C:/Users/sarik/AppData/Local/Python/pythoncore-3.14-64/python.exe "c:/Users/sarik/OneDrive/Desktop/AI ASSISTED CODING/ASSIGN-11-1.py"

1. Insert
2. In-order Traversal
3. Exit
Enter your choice: 1
Enter value to insert: 11
11 inserted as root of the BST.

1. Insert
2. In-order Traversal
3. Exit
Enter your choice: 1
Enter value to insert: 14
14 inserted to the right of 11.

1. Insert
2. In-order Traversal
3. Exit
Enter your choice: 2
```

Explanation: A Binary Search Tree is a hierarchical data structure where the left child contains smaller values and the right child contains larger values than the root. This property makes searching, insertion, and deletion efficient.

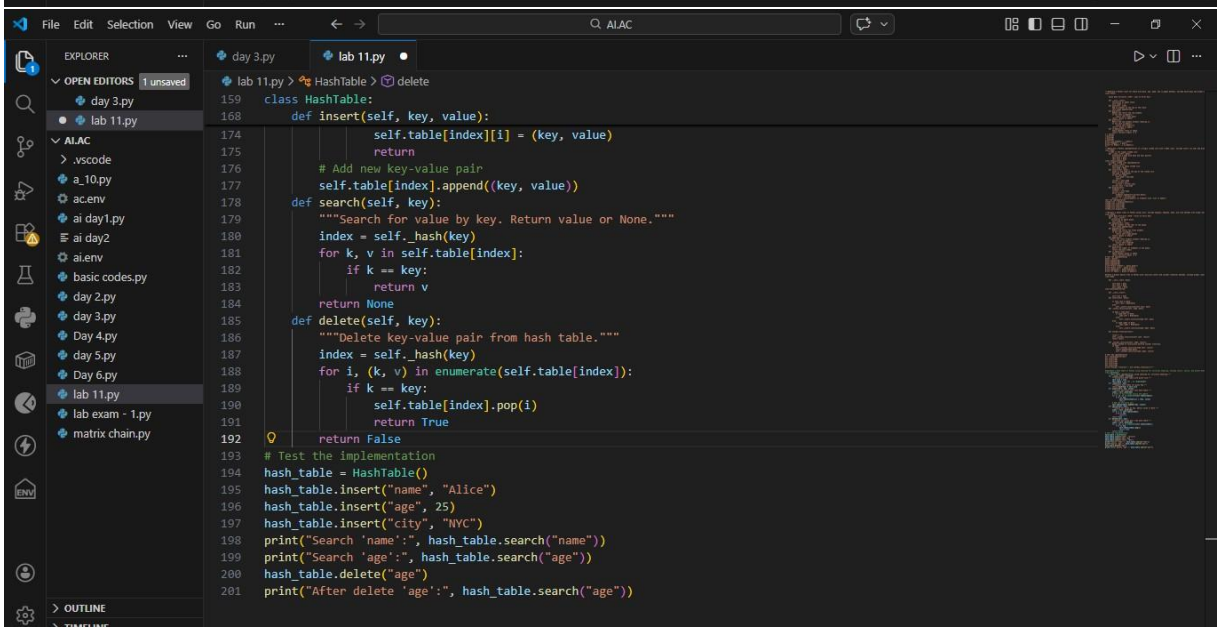
Task Description #5: Hash Table

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

Prompt: Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with comments.

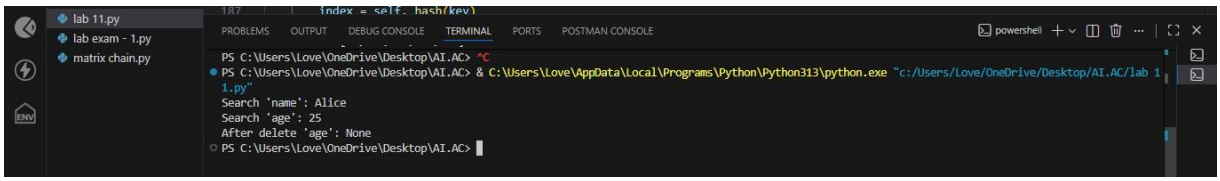


```python
#Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with c
class HashTable:
    """Hash Table implementation using chaining for collision handling."""
    def __init__(self, size=10):
        """Initialize hash table with given size."""
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        """Compute hash index for given key."""
        return hash(key) % self.size
    def insert(self, key, value):
        """Insert key-value pair into hash table."""
        index = self._hash(key)
        # Check if key already exists and update
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)
                return
        # Add new key-value pair
        self.table[index].append((key, value))
    def search(self, key):
        """Search for value by key. Return value or None."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
```



```python
class HashTable:
    def insert(self, key, value):
                self.table[index][i] = (key, value)
                return
        # Add new key-value pair
        self.table[index].append((key, value))
    def search(self, key):
        """Search for value by key. Return value or None."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None
    def delete(self, key):
        """Delete key-value pair from hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index].pop(i)
                return True
        return False
# Test the implementation
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 25)
hash_table.insert("city", "NYC")
print("Search 'name':", hash_table.search("name"))
print("Search 'age':", hash_table.search("age"))
hash_table.delete("age")
print("After delete 'age':", hash_table.search("age"))
```

OUTPUT:

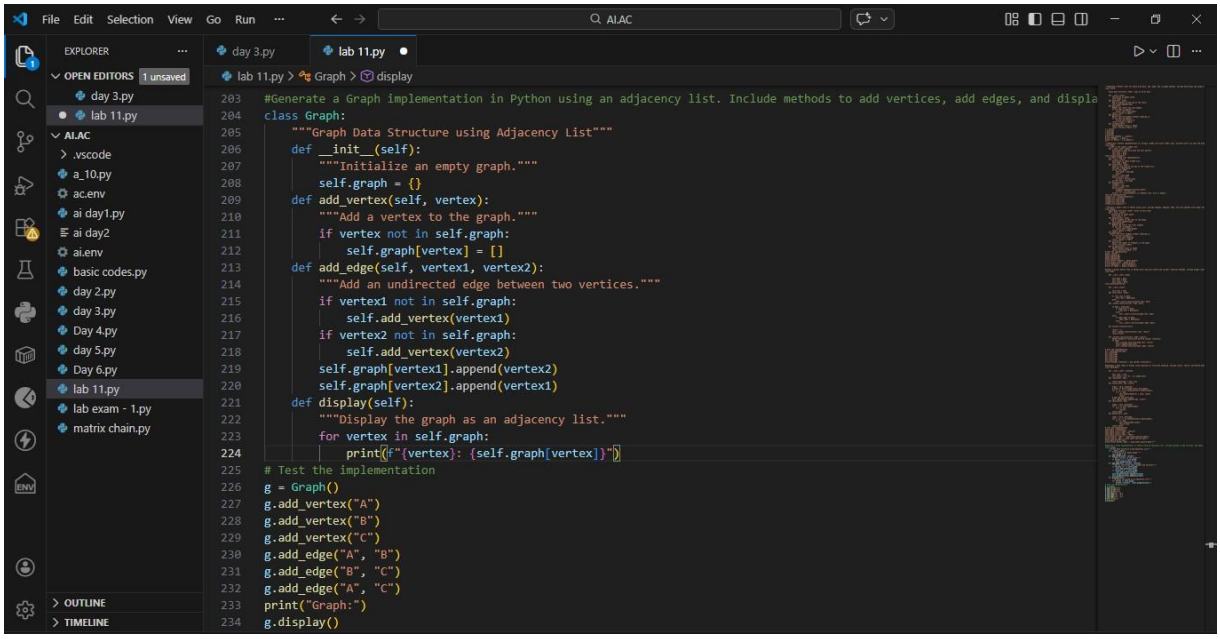


Explanation: A Hash Table stores data in key-value pairs using a hash function to compute an index. It provides fast average-case time complexity for search, insertion, and deletion operations.

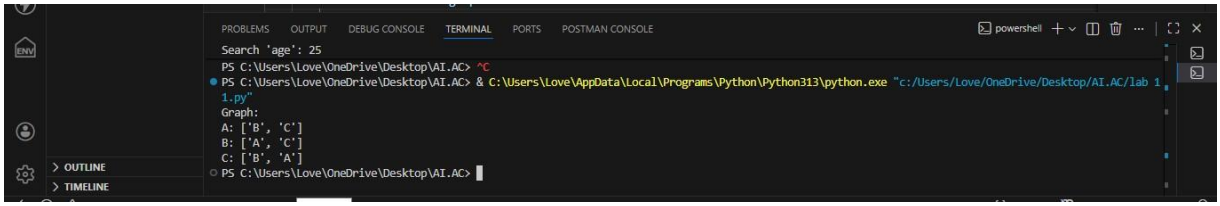
Task Description #6: Graph Representation

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

Prompt: Generate a Graph implementation in Python using an adjacency list. Include methods to add vertices, add edges, and display the graph.



Output:

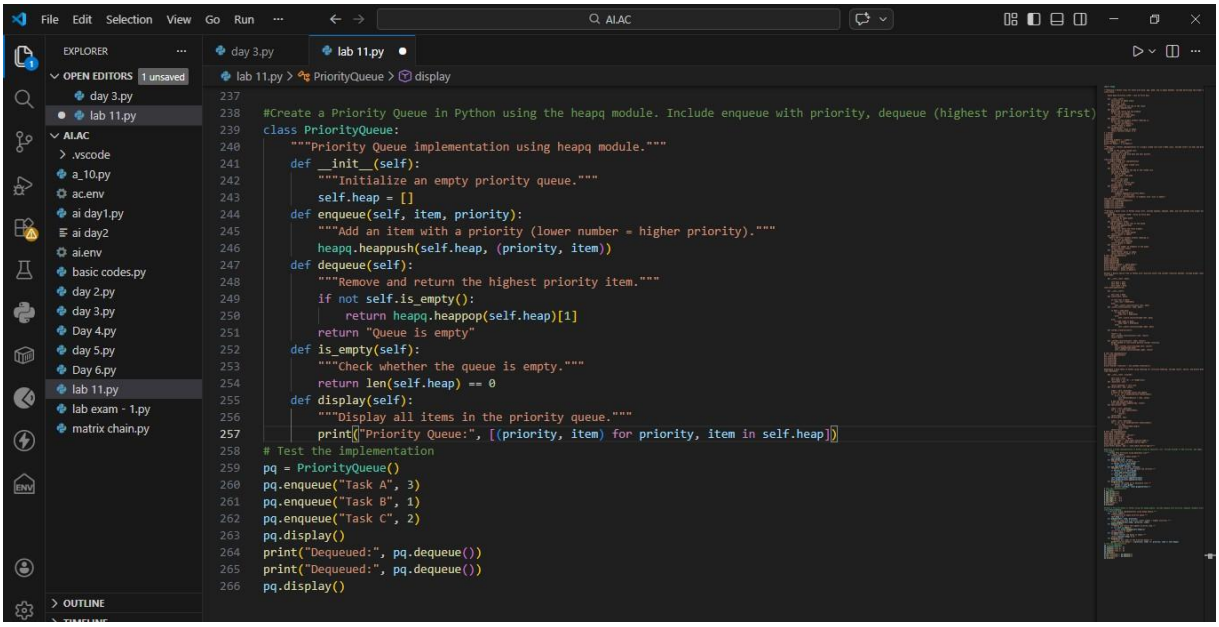


Explanation: A Graph is a non-linear data structure used to represent relationships between entities. It consists of vertices (nodes) and edges (connections).

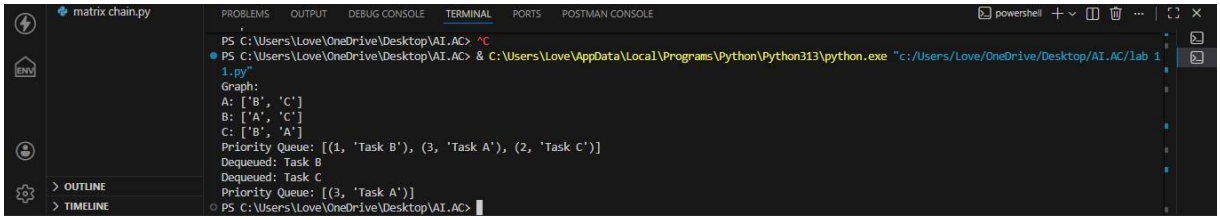
Task Description #7: Priority Queue

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

Prompt: Create a Priority Queue in Python using the heapq module. Include enqueue with priority, dequeue (highest priority first), and display methods.



Output:

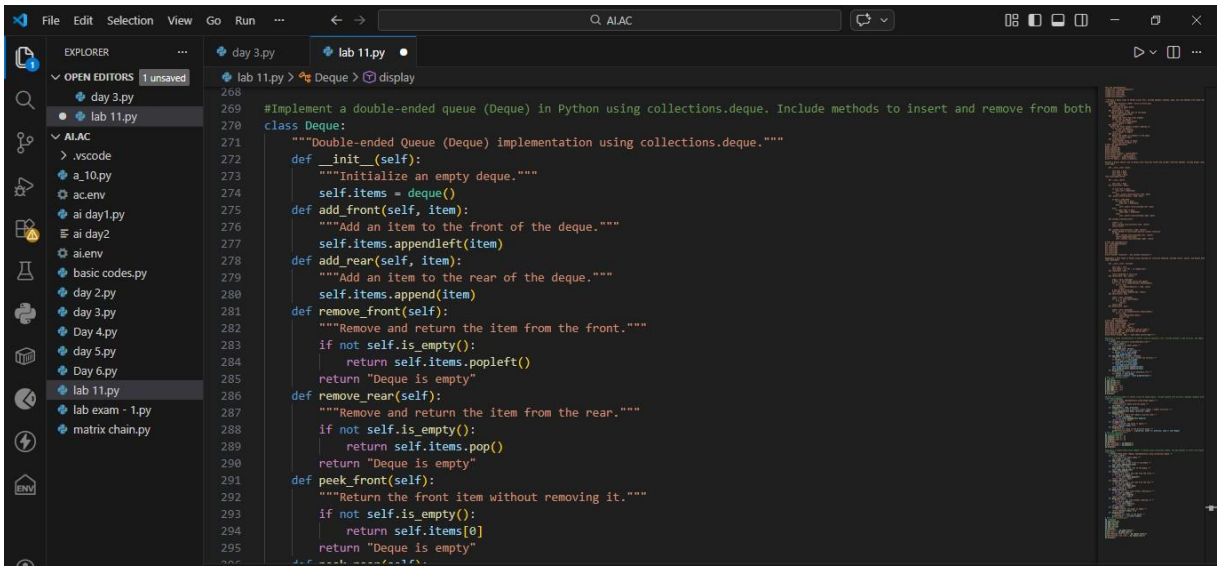


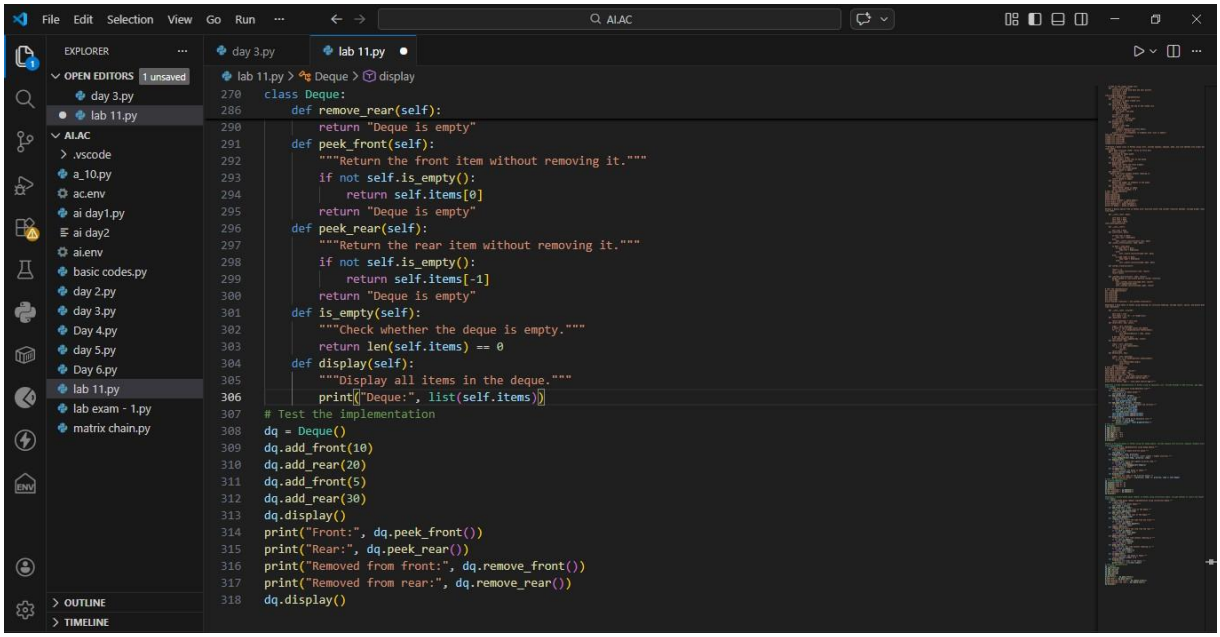
Explanation: A Priority Queue is a special type of queue where elements are removed based on priority rather than order of insertion. Higher priority elements are processed first. It is typically implemented using a heap for efficiency.

Task Description #8 – Deque

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

Prompt: Implement a double-ended queue (Deque) in Python using collections, deque. Include methods to insert and remove from both ends with documentation.





Output:

Explanation: A Deque (Double Ended Queue) allows insertion and deletion of elements from both the front and rear ends.