

# ASSIGNMENT 11.1

## Data Structures with AI: Implementing Fundamental Structures

Amruth Sagar Vemuganti

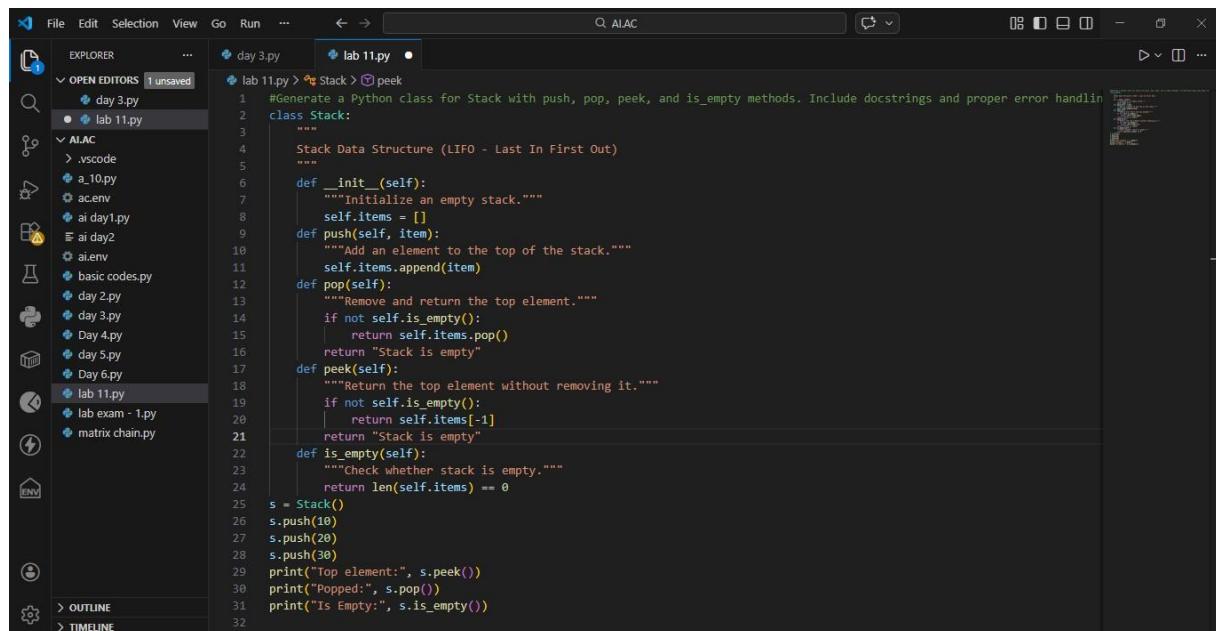
2403A51L44

B-52

### Task 1: Stack Implementation

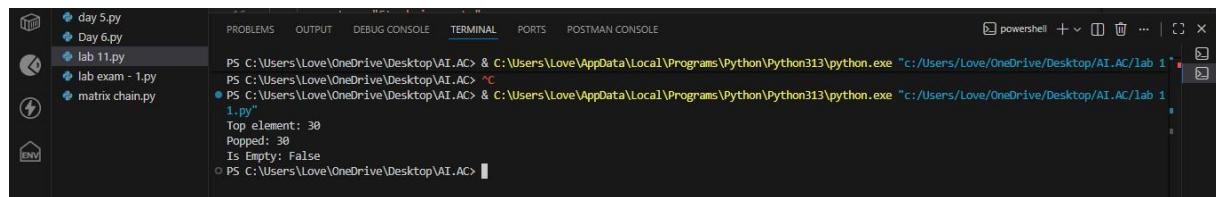
**Task:** Use AI to generate a Stack class with push, pop, peek, and is\_empty methods.

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



```
#Generate a Python class for Stack with push, pop, peek, and is_empty methods. Include docstrings and proper error handling
class Stack:
    """
    Stack Data Structure (LIFO - Last In First Out)
    """
    def __init__(self):
        """
        Initialize an empty stack.
        """
        self.items = []
    def push(self, item):
        """
        Add an element to the top of the stack.
        """
        self.items.append(item)
    def pop(self):
        """
        Remove and return the top element.
        """
        if not self.is_empty():
            return self.items.pop()
        return "Stack is empty"
    def peek(self):
        """
        Return the top element without removing it.
        """
        if not self.is_empty():
            return self.items[-1]
        return "Stack is empty"
    def is_empty(self):
        """
        Check whether stack is empty.
        """
        return len(self.items) == 0
s = Stack()
s.push(10)
s.push(20)
s.push(30)
print("Top element:", s.peek())
print("Popped:", s.pop())
print("Is Empty:", s.is_empty())
```

### OUTPUT:



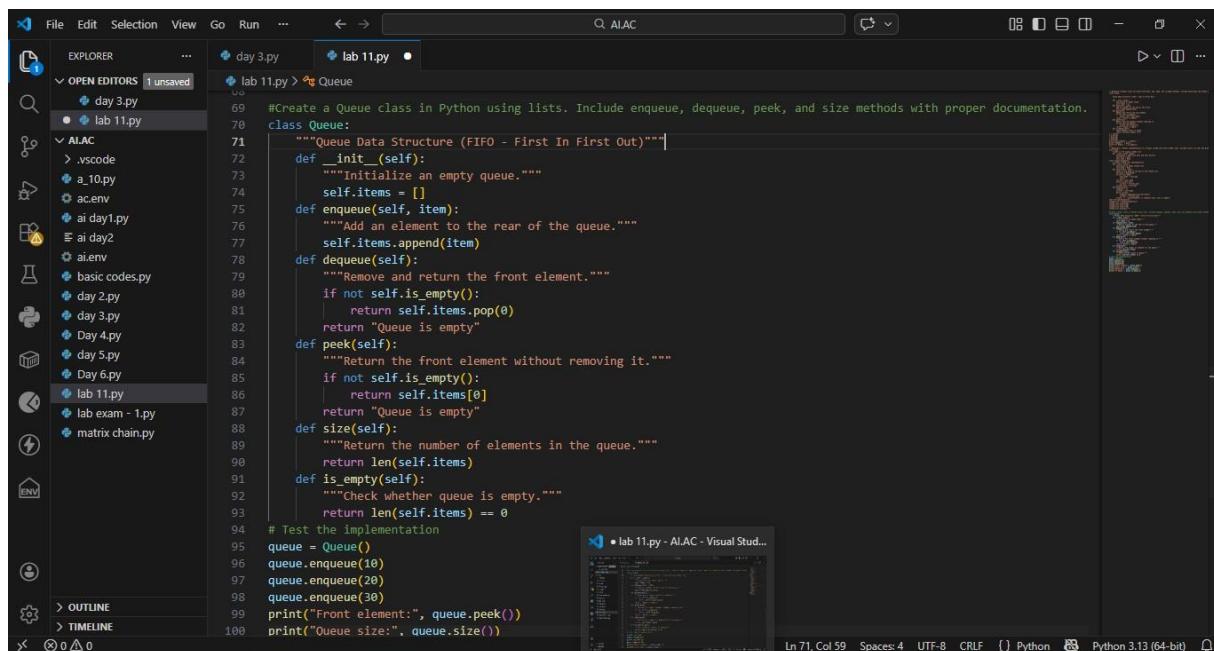
```
PS C:\Users\Love\OneDrive\Desktop\AI.AC> & C:\Users\Love\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/Love/OneDrive/Desktop/AI.AC/lab 1.py"
PS C:\Users\Love\OneDrive\Desktop\AI.AC> & C:\Users\Love\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/Love/OneDrive/Desktop/AI.AC/lab 1.py"
Top element: 30
Popped: 30
Is Empty: False
PS C:\Users\Love\OneDrive\Desktop\AI.AC>
```

**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.

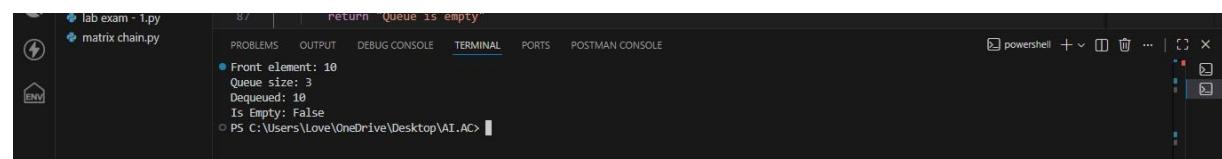


```

File Edit Selection View Go Run ...
OPEN EDITORS 1 unsaved
EXPLORER ...
OPEN EDITORS 1 unsaved
day 3.py lab 11.py
day 3.py > Queue
71 #Create a Queue class in Python using lists. Include enqueue, dequeue, peek, and size methods with proper documentation.
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
class Queue:
    """Queue Data Structure (FIFO - First In First Out)"""
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """Add an element to the rear of the queue."""
        self.items.append(item)
    def dequeue(self):
        """Remove and return the front element."""
        if not self.is_empty():
            return self.items.pop(0)
        return "Queue is empty"
    def peek(self):
        """Return the front element without removing it."""
        if not self.is_empty():
            return self.items[0]
        return "Queue is empty"
    def size(self):
        """Return the number of elements in the queue."""
        return len(self.items)
    def is_empty(self):
        """Check whether queue is empty."""
        return len(self.items) == 0
# Test the implementation
queue = Queue()
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
print("Front element:", queue.peek())
print("Queue size:", queue.size())

```

## OUTPUT:



```

Front element: 10
Queue size: 3
Dequeued: 10
Is Empty: False
PS C:\Users\Love\OneDrive\Desktop\AI.AC>

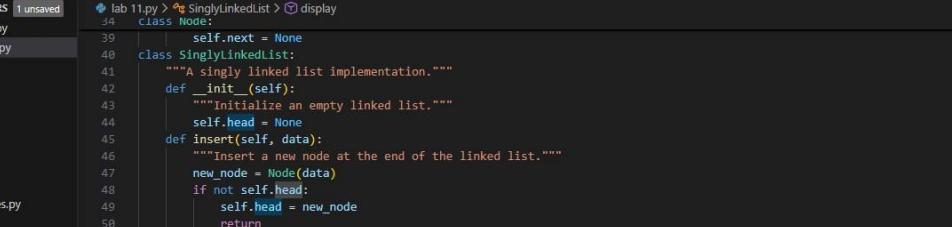
```

**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.



The screenshot shows a code editor interface with the following details:

- File Explorer:** On the left, it shows the project structure with files like `day 3.py`, `lab 11.py`, `a 10.py`, etc.
- Code Editor:** The main area displays Python code for a singly linked list implementation. The code includes classes for `Node` and `SinglyLinkedList`, methods for insertion and display, and a test block at the bottom.
- Search Bar:** At the top center, it says "Search ALAC".
- Toolbar:** At the top right, there are standard window control buttons (minimize, maximize, close).

```
File Edit Selection View Go Run ... < > Search ALAC OPEN EDITORS 1 unsaved EXPLORER day 3.py lab 11.py ... lab 11.py SinglyLinkedList display
34 class Node:
35     self.next = None
36
37 class SinglyLinkedList:
38     """A singly linked list implementation."""
39     def __init__(self):
40         """Initialize an empty linked list."""
41         self.head = None
42
43     def insert(self, data):
44         """Insert a new node at the end of the linked list."""
45         new_node = Node(data)
46         if not self.head:
47             self.head = new_node
48             return
49         current = self.head
50         while current.next:
51             current = current.next
52         current.next = new_node
53
54     def display(self):
55         elements = []
56         current = self.head
57         while current:
58             elements.append(str(current.data))
59             current = current.next
60         print(" -> ".join(elements) if elements else "List is empty")
61
62 # Test the implementation
63 linked_list = SinglyLinkedList()
64 linked_list.insert(10)
65 linked_list.insert(20)
66 linked_list.insert(30)
67 linked_list.display()
```

## **OUTPUT:**

The screenshot shows the Visual Studio Code interface with the following details:

- SIDE BAR:** On the left, there are icons for File, Day 5.py, Day 6.py, lab 11.py (which is selected), lab exam - 1.py, and matrix chain.py.
- TOP BAR:** The title bar shows "53 current = current.next". Below it are tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (which is selected), PORTS, and POSTMAN CONSOLE. To the right are icons for powershell, terminal, file operations, and a close button.
- TERMINAL:** The main area displays a PowerShell prompt: "PS C:\Users\Love\OneDrive\Desktop\AI.AC>". The user has run the command "python lab 1.py", which outputs the result: "Is Empty: False".

**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.

```

150  ## TASK-4: Create a Binary Search Tree in Python with a nested Node class. Implement recursive insert and in-order traversal
151  methods following BST properties. Add proper docstrings.
152
153  class Node:
154      def __init__(self, data):
155          self.data = data
156          self.left = None
157          self.right = None
158
159  class BinarySearchTree:
160      def __init__(self):
161          self.root = None
162
163      def insert(self, data):
164          if self.root is None:
165              self.root = Node(data)
166              print(f"({data}) inserted as root of the BST.")
167          else:
168              self._insert_recursive(self.root, data)
169
170      def _insert_recursive(self, node, data):
171          if data < node.data:
172              if node.left is None:
173                  node.left = Node(data)
174                  print(f"({data}) inserted to the left of {node.data}.")
175              else:
176                  self._insert_recursive(node.left, data)
177
178          if data > node.data:
179              if node.right is None:
180                  node.right = Node(data)
181                  print(f"({data}) inserted to the right of {node.data}.")
182              else:
183                  self._insert_recursive(node.right, data)
184
185      def in_order_traversal(self):
186          elements = []
187          self._in_order_recursive(self.root, elements)
188          print("In-order Traversal: " + ", ".join(map(str, elements)))
189
190      def _in_order_recursive(self, node, elements):
191          if node:
192              self._in_order_recursive(node.left, elements)
193              elements.append(node.data)
194              self._in_order_recursive(node.right, elements)
195
196      bst = BinarySearchTree()
197      while True:
198          print("1. Insert")
199          print("2. In-order Traversal")
200          print("3. Exit")
201          choice = input("Enter your choice: ")
202          if choice == '1':
203              value = input("Enter value to insert: ")
204              bst.insert(value)
205          elif choice == '2':
206              bst.in_order_traversal()
207          elif choice == '3':
208              print("Exiting program...")
209              break
210          else:
211              print("Invalid choice! Try again.")

```

```

157  class BinarySearchTree:
158
159      def in_order_traversal(self):
160          elements = []
161          self._in_order_recursive(self.root, elements)
162          print("In-order Traversal: " + ", ".join(map(str, elements)))
163
164      def _in_order_recursive(self, node, elements):
165          if node:
166              self._in_order_recursive(node.left, elements)
167              elements.append(node.data)
168              self._in_order_recursive(node.right, elements)
169
170      bst = BinarySearchTree()
171      while True:
172          print("1. Insert")
173          print("2. In-order Traversal")
174          print("3. Exit")
175          choice = input("Enter your choice: ")
176          if choice == '1':
177              value = input("Enter value to insert: ")
178              bst.insert(value)
179          elif choice == '2':
180              bst.in_order_traversal()
181          elif choice == '3':
182              print("Exiting program...")
183              break
184          else:
185              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/OneD...
rive/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

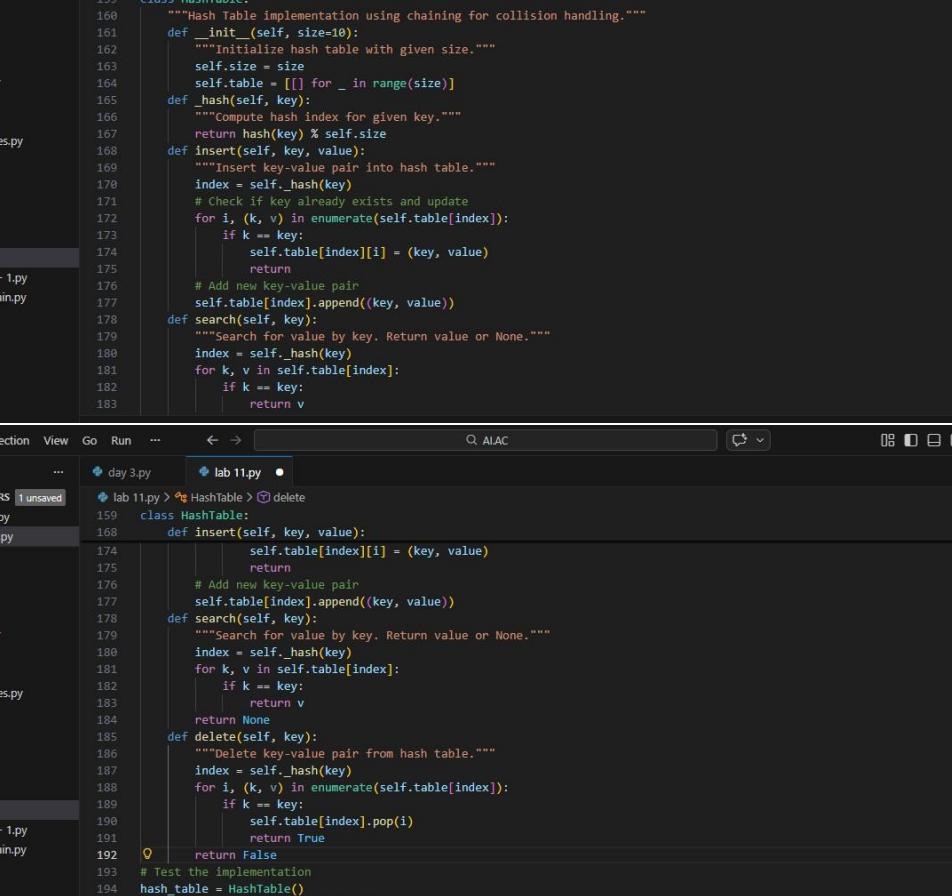
```

**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.



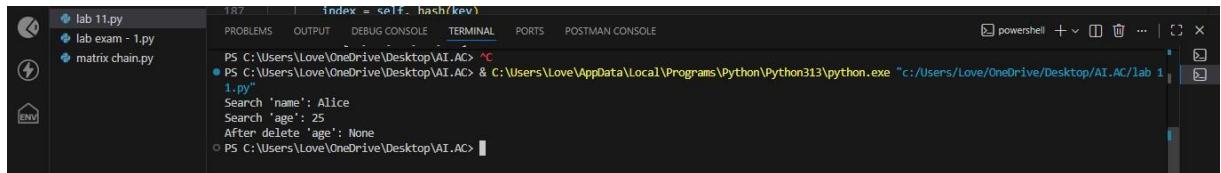
```
File Edit Selection View Go Run ... 🔍 AIAC

EXPLORER ... day 3.py lab 11.py ●
OPEN EDITORS 1 unsaved
lab 11.py > HashTable > delete
158 # Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with collision handling.
159 class HashTable:
160     """Hash Table implementation using chaining for collision handling."""
161     def __init__(self, size=10):
162         """Initialize hash table with given size."""
163         self.size = size
164         self.table = [[] for _ in range(size)]
165     def __hash__(self, key):
166         """Compute hash index for given key."""
167         return hash(key) % self.size
168     def insert(self, key, value):
169         """Insert key-value pair into hash table."""
170         index = self.__hash__(key)
171         # Check if key already exists and update
172         for i, (k, v) in enumerate(self.table[index]):
173             if k == key:
174                 self.table[index][i] = (key, value)
175                 return
176         # Add new key-value pair
177         self.table[index].append((key, value))
178     def search(self, key):
179         """Search for value by key. Return value or None."""
180         index = self.__hash__(key)
181         for k, v in self.table[index]:
182             if k == key:
183                 return v
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
```

```
File Edit Selection View Go Run ... 🔍 AIAC

EXPLORER ... day 3.py lab 11.py ●
OPEN EDITORS 1 unsaved
lab 11.py > HashTable > delete
159 class HashTable:
160     def insert(self, key, value):
161         self.table[index][i] = (key, value)
162         return
163     # Add new key-value pair
164     self.table[index].append((key, value))
165     def search(self, key):
166         """Search for value by key. Return value or None."""
167         index = self.__hash__(key)
168         for k, v in self.table[index]:
169             if k == key:
170                 return v
171         return None
172     def delete(self, key):
173         """Delete key-value pair from hash table."""
174         index = self.__hash__(key)
175         for i, (k, v) in enumerate(self.table[index]):
176             if k == key:
177                 self.table[index].pop(i)
178                 return True
179         return False
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
```

## OUTPUT:



The screenshot shows the VS Code interface with the terminal tab active. The terminal window displays the following Python script execution:

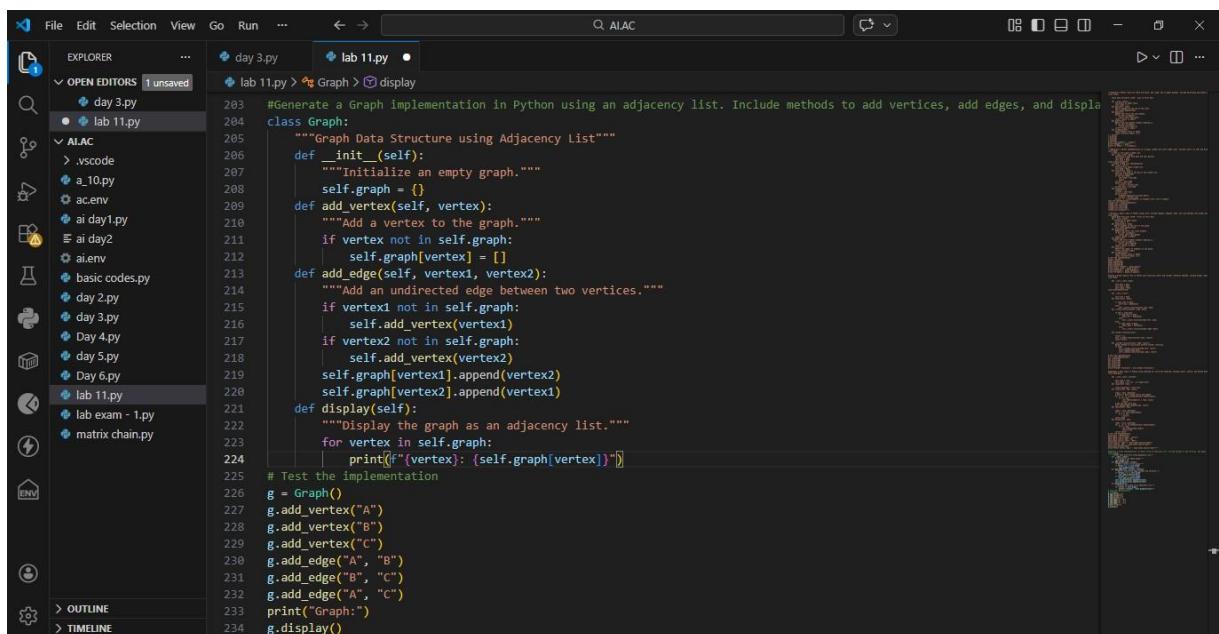
```
PS C:\Users\Love\OneDrive\Desktop\AI.AC> ^C
● PS C:\Users\Love\OneDrive\Desktop\AI.AC> & C:/Users/Love/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/Love/OneDrive/Desktop/AI.AC/lab 1.py"
Search 'name': Alice
Search 'age': 25
After delete 'age': None
○ PS C:\Users\Love\OneDrive\Desktop\AI.AC>
```

**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.



The screenshot shows the VS Code interface with the code editor tab active. The code editor displays a Python script named `lab 11.py` which implements a graph using an adjacency list. The code includes methods for adding vertices and edges, and displaying the graph as an adjacency list. The code editor has a dark theme and shows line numbers and syntax highlighting.

```
203 #Generate a Graph implementation in Python using an adjacency list. Include methods to add vertices, add edges, and display
204 class Graph:
205     """Graph Data Structure using Adjacency List"""
206     def __init__(self):
207         """Initialize an empty graph."""
208         self.graph = {}
209     def add_vertex(self, vertex):
210         """Add a vertex to the graph."""
211         if vertex not in self.graph:
212             self.graph[vertex] = []
213     def add_edge(self, vertex1, vertex2):
214         """Add an undirected edge between two vertices."""
215         if vertex1 not in self.graph:
216             self.add_vertex(vertex1)
217         if vertex2 not in self.graph:
218             self.add_vertex(vertex2)
219         self.graph[vertex1].append(vertex2)
220         self.graph[vertex2].append(vertex1)
221     def display(self):
222         """Display the graph as an adjacency list."""
223         for vertex in self.graph:
224             print(f'{vertex}: {self.graph[vertex]}')
225
226 # Test the implementation
227 g = Graph()
228 g.add_vertex("A")
229 g.add_vertex("B")
230 g.add_vertex("C")
231 g.add_edge("A", "B")
232 g.add_edge("B", "C")
233 g.add_edge("A", "C")
234 print("Graph:")
g.display()
```

## Output:

The screenshot shows the VS Code interface with the terminal tab selected. The terminal window displays the following output:

```
Search 'age': 25
PS C:\Users\Love\OneDrive\Desktop\AI.AC> ^C
● PS C:\Users\Love\OneDrive\Desktop\AI.AC & C:\Users\Love\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/Love/OneDrive/Desktop/AI.AC/lab 1.py"
Graph:
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A']

PS C:\Users\Love\OneDrive\Desktop\AI.AC>
```

**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 heapq module.

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

The screenshot shows the VS Code interface with the code editor tab selected. The editor window displays the following Python code:

```
238 #Create a Priority Queue in Python using the heapq module. Include enqueue with priority, dequeue (highest priority first)
239 class PriorityQueue:
240     """Priority Queue implementation using heapq module."""
241     def __init__(self):
242         """Initialize an empty priority queue."""
243         self.heap = []
244     def enqueue(self, item, priority):
245         """Add an item with a priority (lower number = higher priority)."""
246         heapq.heappush(self.heap, (priority, item))
247     def dequeue(self):
248         """Remove and return the highest priority item."""
249         if not self.is_empty():
250             return heapq.heappop(self.heap)[1]
251         return "Queue is empty"
252     def is_empty(self):
253         """Check whether the queue is empty."""
254         return len(self.heap) == 0
255     def display(self):
256         """Display all items in the priority queue."""
257         print("Priority Queue:", [(priority, item) for priority, item in self.heap])
258
259 # Test the implementation
260 pq = PriorityQueue()
261 pq.enqueue("Task A", 3)
262 pq.enqueue("Task B", 1)
263 pq.enqueue("Task C", 2)
264 pq.display()
265 print("Dequeued:", pq.dequeue())
266 print("Dequeued:", pq.dequeue())
267 pq.display()
```

## Output:

```

matrix_chain.py
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE
PS C:\Users\Love\OneDrive\Desktop\AI.AC> ^C
● PS C:\Users\Love\OneDrive\Desktop\AI.AC & C:\Users\Love\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/Love/OneDrive/Desktop/AI.AC/lab 1
1.py"
Graph:
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A']
Priority Queue: [(1, 'Task B'), (3, 'Task A'), (2, 'Task C')]
Dequeued: Task B
Dequeued: Task C
Priority Queue: [(3, 'Task A')]
PS C:\Users\Love\OneDrive\Desktop\AI.AC>

```

**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.

```

File Edit Selection View Go Run ... ← → Q AIAC 08 □ □ - ○ ...
EXPLORER OPEN EDITORS 1 unsaved lab 11.py ●
lab 11.py > Deque > display
268 #Implement a double-ended queue (Deque) in Python using collections.deque. Include methods to insert and remove from both
269 class Deque:
270     """Double-ended Queue (Deque) implementation using collections.deque."""
271     def __init__(self):
272         """Initialize an empty deque."""
273         self.items = deque()
274     def add_front(self, item):
275         """Add an item to the front of the deque."""
276         self.items.appendleft(item)
277     def add_rear(self, item):
278         """Add an item to the rear of the deque."""
279         self.items.append(item)
280     def remove_front(self):
281         """Remove and return the item from the front."""
282         if not self.is_empty():
283             return self.items.popleft()
284         return "Deque is empty"
285     def remove_rear(self):
286         """Remove and return the item from the rear."""
287         if not self.is_empty():
288             return self.items.pop()
289         return "Deque is empty"
290     def peek_front(self):
291         """Return the front item without removing it."""
292         if not self.is_empty():
293             return self.items[0]
294         return "Deque is empty"
295

```

The screenshot shows the VS Code interface with the following details:

- EXPLORER**: Shows files like day 3.py, lab 11.py, .vscode, a\_10.py, ac.env, ai day1.py, ai day2.py, ai.env, basic codes.py, day 2.py, day 3.py, Day 4.py, day 5.py, Day 6.py, lab exam - 1.py, and matrix chain.py.
- CODE EDITOR**: The active file is lab 11.py, which contains Python code for a Deque class. The code includes methods for adding items at front and rear, removing items from front and rear, peeking at front and rear, checking if the deque is empty, and displaying all items.
- OUTPUT**: Shows the terminal output of the code execution.

```
270     class Deque:
271         def __init__(self):
272             self.items = []
273
274         def add_front(self, item):
275             self.items.insert(0, item)
276
277         def add_rear(self, item):
278             self.items.append(item)
279
280         def remove_front(self):
281             if len(self.items) == 0:
282                 return None
283             return self.items.pop(0)
284
285         def remove_rear(self):
286             if len(self.items) == 0:
287                 return None
288             return self.items.pop()
289
290         def peek_front(self):
291             if len(self.items) == 0:
292                 return None
293             return self.items[0]
294
295         def peek_rear(self):
296             if len(self.items) == 0:
297                 return None
298             return self.items[-1]
299
300         def is_empty(self):
301             return len(self.items) == 0
302
303         def display(self):
304             print("Deque:", self.items)
305
306     # Test the implementation
307     dq = Deque()
308     dq.add_front(10)
309     dq.add_rear(20)
310     dq.add_front(5)
311     dq.add_rear(30)
312     dq.display()
313
314     print("Front:", dq.peek_front())
315     print("Rear:", dq.peek_rear())
316
317     print("Removed from front:", dq.remove_front())
318     print("Removed from rear:", dq.remove_rear())
319
320     dq.display()
```

## Output:

The screenshot shows the VS Code interface with the following details:

- TERMINAL**: Displays the command-line output of the code execution.

```
PS C:\Users\Love\OneDrive\Desktop\AI.AC> python lab 11.py
Priority Queue: [(3, 'Task A')]
Deque: [5, 10, 20, 30]
Front: 5
Rear: 30
Removed from front: 5
Removed from rear: 30
Deque: [10, 20]
```

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