

ASSIGNMENT 11.1

Data Structures with AI: Implementing Fundamental Structures

Atla Sreeja

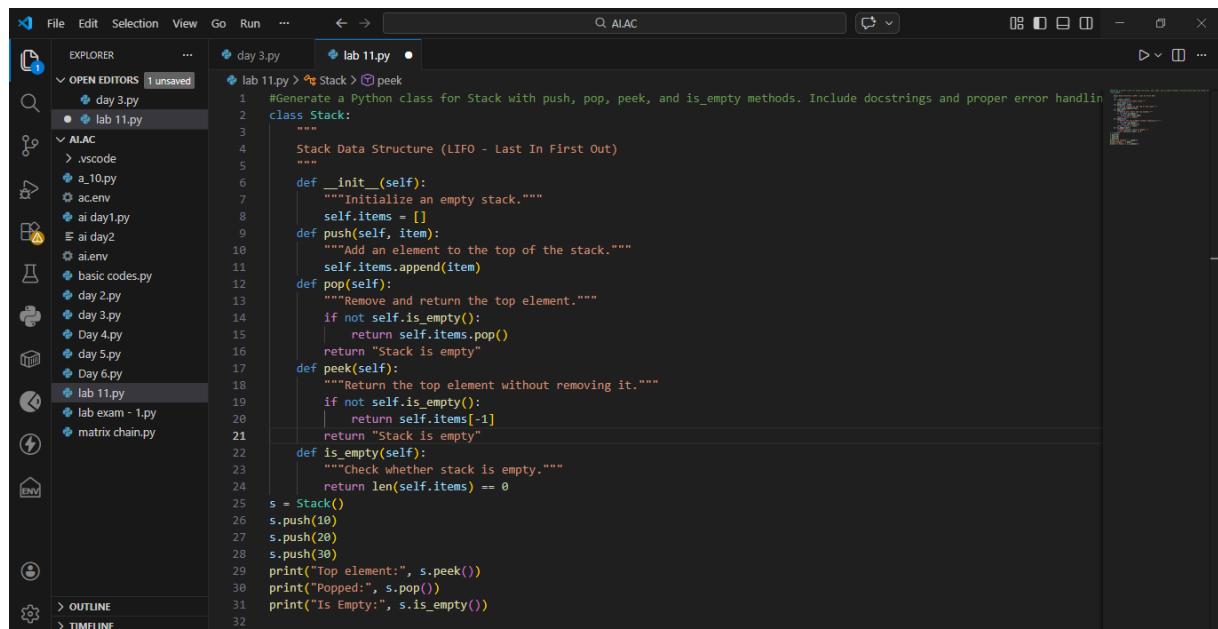
2403A51L02

B-51

Task 1: Stack Implementation

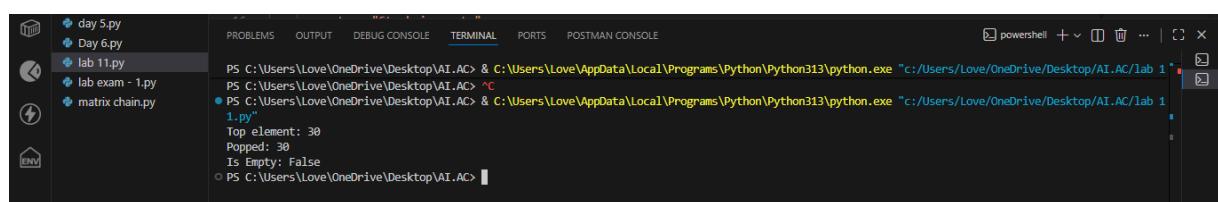
Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

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



```
File Edit Selection View Go Run ... 🔍 AIAC
EXPLORER OPEN EDITORS 1 unsaved
  day 3.py
  lab 11.py
AIAC
  .vscode
  a.10.py
  ac.env
  ai day1.py
  ai day2
  ai.env
  basic codes.py
  day 2.py
  day 3.py
  Day 4.py
  day 5.py
  Day 6.py
  lab 11.py
  lab exam - 1.py
  matrix chain.py
  lab 11.py
  lab 11.py > ⚙️ Stack > ⚡ peek
  #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:



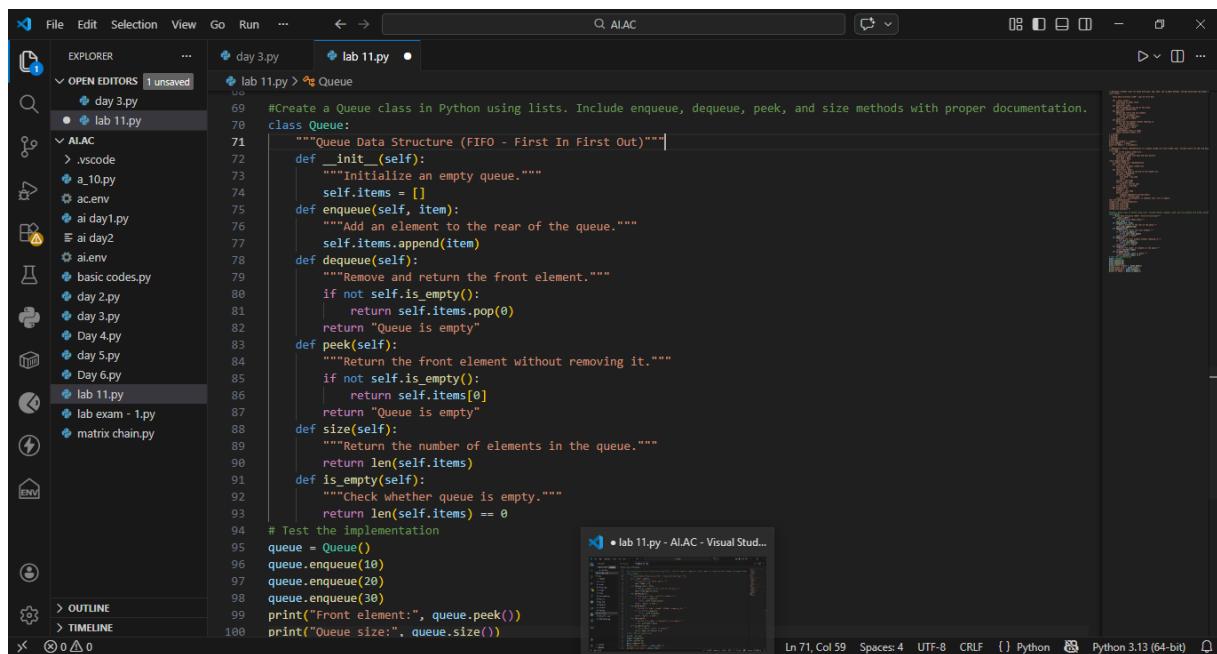
```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE powershell + × 🗑️ ...
day 5.py
Day 6.py
lab 11.py
lab exam - 1.py
matrix chain.py
PS C:\Users\Love\OneDrive\Desktop\AI.AC> & C:\Users\Love\AppData\Local\Programs\Python\Python313\python.exe "c:/Users/Love/OneDrive/Desktop/AI.AC/lab 1.py"
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"
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.



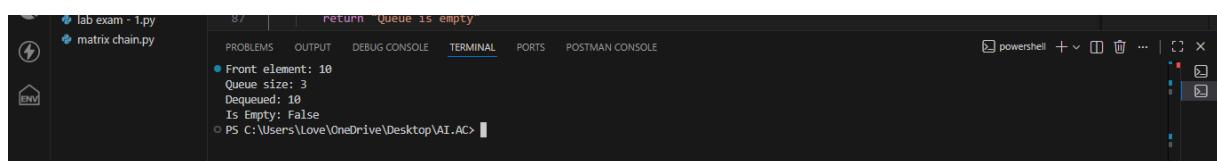
The screenshot shows the Visual Studio Code interface with the following details:

- EXPLORER:** Shows files like day 3.py, lab 11.py, .vscode, a.10.py, ac.env, ai day1.py, ai day2, ai.env, basic codes.py, day 2.py, day 3.py, Day 4.py, Day 5.py, Day 6.py, lab 11.py, lab exam - 1.py, and matrix chain.py.
- EDITOR:** The file lab 11.py is open, containing the following Python code:

```
#Create a Queue class in Python using lists. Include enqueue, dequeue, peek, and size methods with proper documentation.
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())
```

The status bar at the bottom indicates: Ln 71, Col 59 | Spaces: 4 | UTF-8 | CRLF | Python | Python 3.13 (64-bit)

OUTPUT:



The screenshot shows the Visual Studio Code terminal window with the following output:

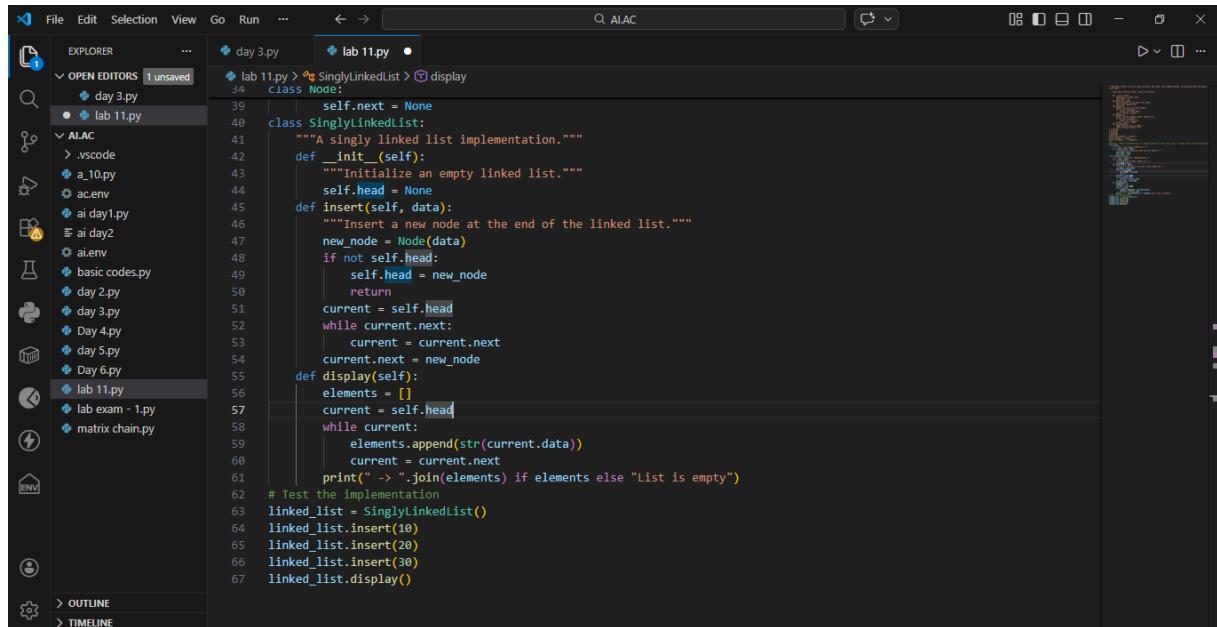
```
Front element: 10
Queue size: 3
Dequeued: 10
Is Empty: False
PS C:\Users\Love\OneDrive\Desktop\AI.AC>
```

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.

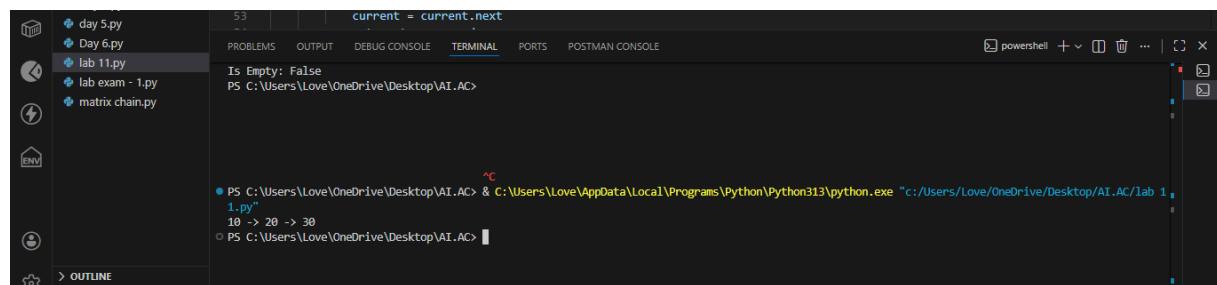
Prompt : Generate a Python implementation of a Singly Linked List with a Node class. Include insert (at end) and display methods with docstrings.



The screenshot shows the Visual Studio Code interface. The left sidebar displays a file tree with various Python files like 'day 3.py', 'day 4.py', and 'lab 11.py'. The main editor area contains the following Python code for a singly linked list:

```
class Node:
    def __init__(self):
        self.next = None
class SinglyLinkedList:
    """A singly linked list implementation."""
    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."""
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(str(current.data))
            current = current.next
        print(" -> ".join(elements) if elements else "List is empty")
# Test the implementation
linked_list = SinglyLinkedList()
linked_list.insert(10)
linked_list.insert(20)
linked_list.insert(30)
linked_list.display()
```

OUTPUT:



The screenshot shows the VS Code terminal window. It displays the execution of the Python script and its 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 11.py"
Is Empty: False
PS C:\Users\Love\OneDrive\Desktop\AI.AC>
```

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.

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

- File Explorer:** Shows files like `ASSIGN-11-1.py`, `ASSIGN-1-3.py`, `ASSIGN-2-2.py`, etc.
- Editor Area:** Displays the following Python code for a Binary Search Tree (BST) with recursive insert and in-order traversal methods. The code includes AI-assisted coding comments and docstrings.

```
150 ## 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.
151
152     class Node:
153         def __init__(self, data):
154             self.data = data
155             self.left = None
156             self.right = None
157
158     class BinarySearchTree:
159         def __init__(self):
160             self.root = None
161
162         def insert(self, data):
163             if self.root is None:
164                 self.root = Node(data)
165                 print(f"{data} inserted as root of the BST.")
166             else:
167                 self._insert_recursive(self.root, data)
168
169         def _insert_recursive(self, node, data):
170             if data < node.data:
171                 if node.left is None:
172                     node.left = Node(data)
173                     print(f"{data} inserted to the left of {node.data}.")
174                 else:
175                     self._insert_recursive(node.left, data)
176
177                 if node.right is None:
178                     node.right = Node(data)
179                     print(f"{data} inserted to the right of {node.data}.")
180                 else:
181                     self._insert_recursive(node.right, data)
182
183         def in_order_traversal(self):
184             elements = []
185             self._in_order_recursive(self.root, elements)
186             print("In-order Traversal: " + " ".join(map(str, elements)))
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
```

The code editor has a dark theme and includes a sidebar for AI assistance and sessions.

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

- File Explorer:** Shows files like `ASSIGN-11-1.py`, `ASSIGN-1-3.py`, `ASSIGN-2-2.py`, etc.
- Editor Area:** Displays the completed Python code for a Binary Search Tree (BST) with both recursive insert and in-order traversal methods.

```
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("\n1. 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:

The screenshot shows a terminal window with the following output:

```
PS C:\Users\Arik\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.

The screenshot shows a code editor with the following code in the 'lab 11.py' file:

```
157
158 #Implement a Hash Table in Python using chaining for collision handling. Include insert, search, and delete methods with comments.
159
160     """Hash Table implementation using chaining for collision handling."""
161     """Initialize hash table with given size."""
162     self.size = size
163     self.table = [None] * size
164
165     def __init__(self, size=10):
166         self.size = size
167         self.table = [None] * size
168
169     def _hash(self, key):
170         """Compute hash index for given key."""
171         return hash(key) % self.size
172
173     def insert(self, key, value):
174         """Insert key-value pair into hash table."""
175         index = self._hash(key)
176
177         if self.table[index] == None:
178             self.table[index] = [(key, value)]
179         else:
180             for i in range(len(self.table[index])):
181                 if self.table[index][i][0] == key:
182                     self.table[index][i] = (key, value)
183                     return
184
185             self.table[index].append((key, value))
186
187     def search(self, key):
188         """Search for value by key. Return value or None."""
189         index = self._hash(key)
190
191         for i in range(len(self.table[index])):
192             if self.table[index][i][0] == key:
193                 return self.table[index][i][1]
194
195         return None
```

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

- File Explorer (Left):** Shows the workspace structure with files like day 3.py, lab 11.py, .vscode, a.10.py, ac.env, ai day1.py, ai day2, ai.env, basic codes.py, day 2.py, day 3.py, Day 4.py, Day 5.py, Day 6.py, lab 11.py, lab exam - 1.py, and matrix chain.py.
- Editor (Center):** Displays the content of `lab 11.py`, which contains Python code for a HashTable class. The code includes methods for insertion, search, and deletion of key-value pairs.
- Terminal (Bottom):** Shows the command line output of running the script, demonstrating its functionality with examples of inserting 'name' and 'age' and then deleting 'age'.

OUTPUT:

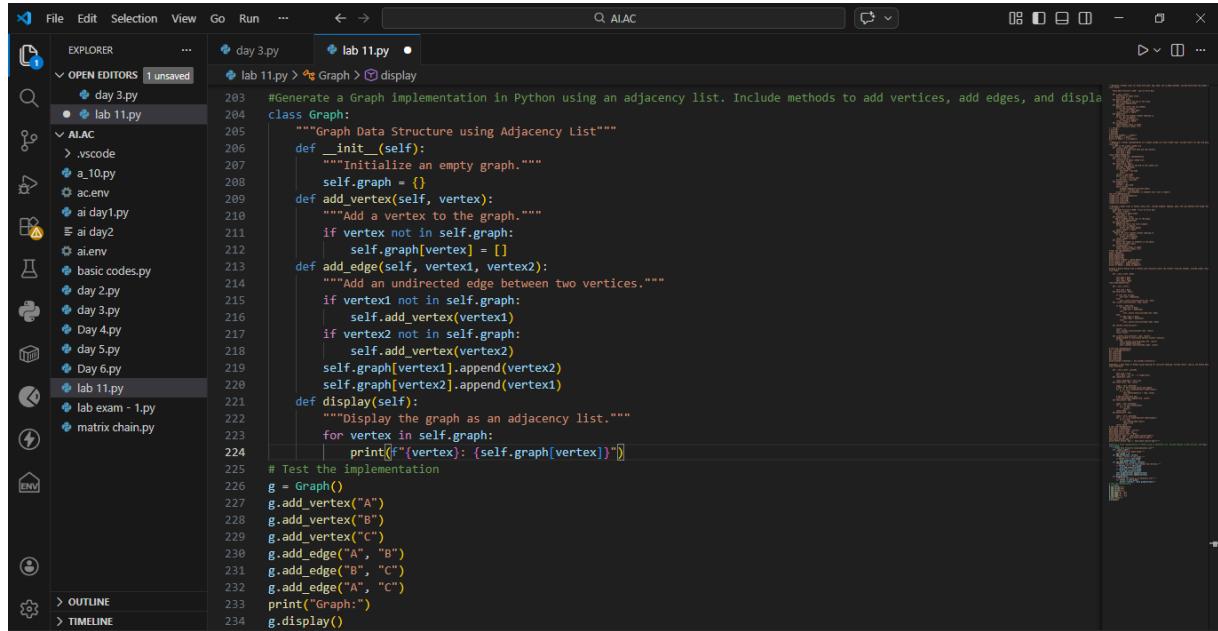
```
PS C:\Users\Love\OneDrive\Desktop\AI.AC> python lab 11.py
Search 'name': Alice
Search 'age': 25
After delete 'age': None
```

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.

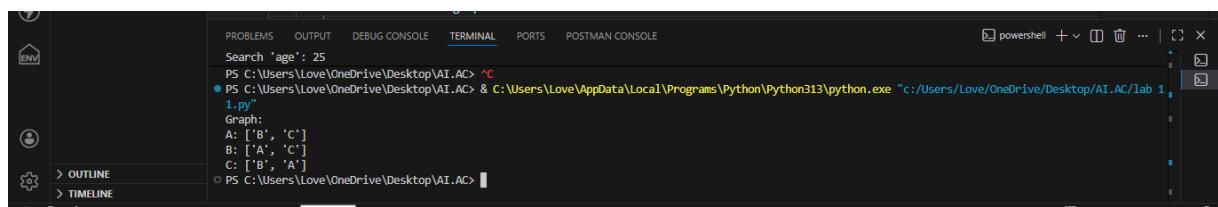


The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files like day 3.py, lab 11.py, and various .py files under the ALAC folder.
- Editor:** Displays the content of lab 11.py, which contains Python code for a Graph class using an adjacency list.
- Terminal:** Shows the command "python lab 11.py" being run.
- Output:** Shows the execution results, including the creation of vertices A, B, and C, and the addition of edges between them.

```
203 #Generate a Graph implementation in Python using an adjacency list. Include methods to add vertices, add edges, and display
204
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:")
235 g.display()
```

Output:



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

- File Explorer:** Shows files like day 3.py, lab 11.py, and various .py files under the ALAC folder.
- Terminal:** Displays the command "python lab 11.py" and its output.

```
Search 'age': 25
PS C:\Users\Love\OneDrive\Desktop\AI.AC> ^
● 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 11.py"
Graph:
A: ['B', 'C']
B: ['A', 'C']
C: ['B', 'A']

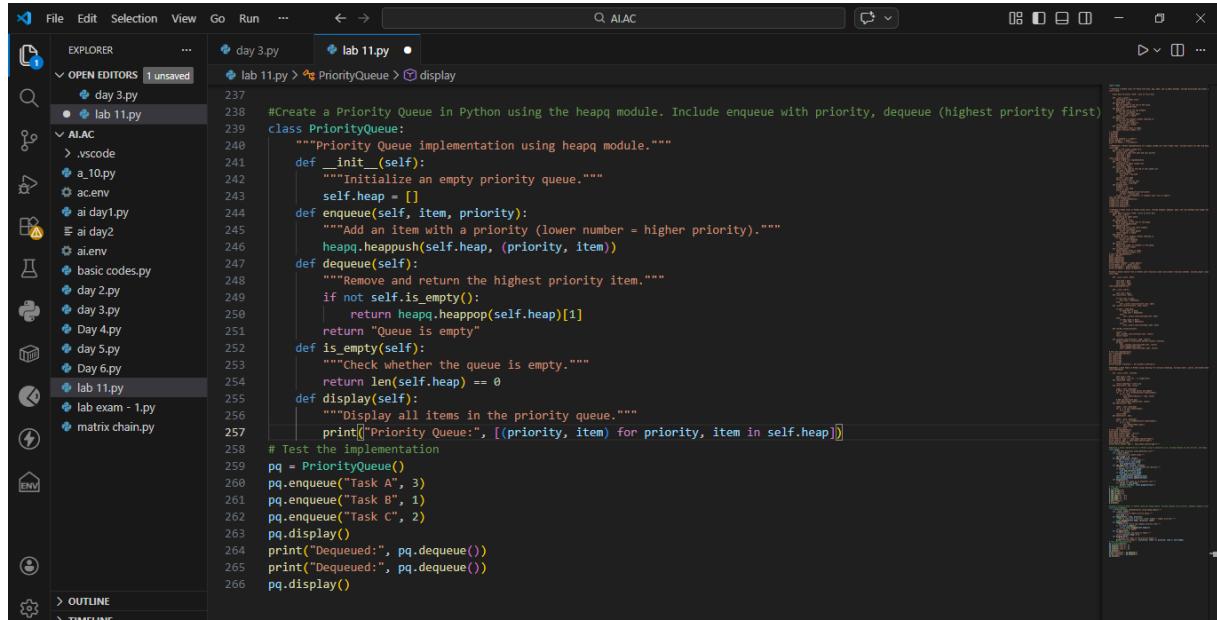
PS C:\Users\Love\OneDrive\Desktop\AI.AC>
```

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 heap module.

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

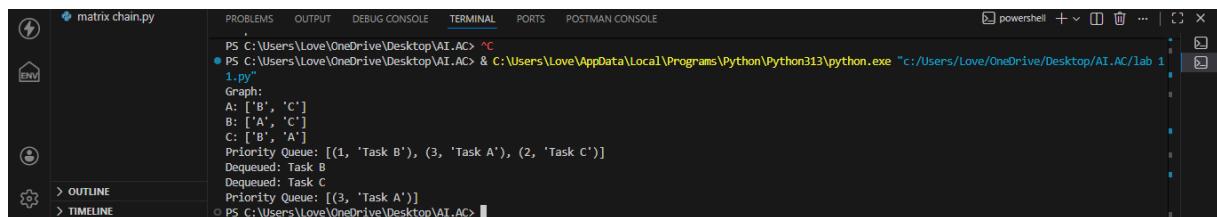


The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows a folder structure for "AI.AC" containing files like "day 3.py", "lab 11.py", "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 11.py", "lab exam - 1.py", and "matrix chain.py". "lab 11.py" is the active file.
- Code Editor:** Displays the code for "lab 11.py". The code implements a Priority Queue using the heapq module. It includes methods for enqueueing items with priority, dequeuing the highest priority item, checking if the queue is empty, and displaying all items.

```
237
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     print("Dequeued:", pq.dequeue())
265     print("Dequeued:", pq.dequeue())
266     pq.display()
```

Output:



The screenshot shows the terminal window in VS Code with the following details:

- Matrix Chain:** The current file being worked on is "matrix chain.py".
- Terminal Content:** The terminal shows the command "python lab 1.py" being run, followed by the output of the Priority Queue implementation. The output shows the enqueueing of tasks A, B, and C, and their subsequent dequeuing in reverse order (C, B, A).

```
PS C:\Users\Love\OneDrive\Desktop\AI.AC> python lab 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 C
Dequeued: Task B
Dequeued: Task A
Priority Queue: [(3, 'Task A')]
```

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.



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

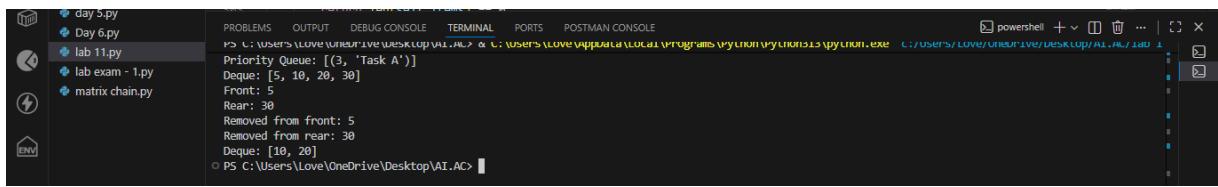
- File Explorer:** Shows the project structure with files like `day 3.py`, `lab 11.py`, `a_10.py`, `ai.env`, `ai day1.py`, `ai day2`, `ai.env`, `basic codes.py`, `day 2.py`, `day 3.py`, `Day 4.py`, `day 5.py`, `Day 6.py`, `lab 11.py` (selected), `lab exam - 1.py`, and `matrix chain.py`.
- Editor:** Displays the content of `lab 11.py` which implements a double-ended queue (Deque) using `collections.deque`. The code includes methods for adding items to the front and rear, removing items from both ends, peeking at the front item, and checking if the deque is empty.
- Search Bar:** Contains the text "ALAC".
- Activity Bar:** Shows various icons for file operations, terminal, and extensions.

The screenshot shows the Visual Studio Code interface with the AIAC extension installed. The left sidebar displays the file tree with several Python files open. The main editor area shows the code for `lab 11.py`, which implements a Deque class. The right side features the AIAC interface, which includes a search bar at the top, a large central panel for displaying AI-generated code or documentation, and a bottom panel for viewing detailed information about the selected code snippet.

```
class Deque:
    def __init__(self):
        self.items = []
    def add_front(self, item):
        self.items.insert(0, item)
    def add_rear(self, item):
        self.items.append(item)
    def remove_front(self):
        if len(self.items) == 0:
            return "Deque is empty"
        return self.items.pop(0)
    def remove_rear(self):
        if len(self.items) == 0:
            return "Deque is empty"
        return self.items.pop()
    def peek_front(self):
        if len(self.items) == 0:
            return "Deque is empty"
        return self.items[0]
    def peek_rear(self):
        if len(self.items) == 0:
            return "Deque is empty"
        return self.items[-1]
    def is_empty(self):
        return len(self.items) == 0
    def display(self):
        print("Deque:", self.items)

# Test the implementation
dq = Deque()
dq.add_front(10)
dq.add_rear(20)
dq.add_front(5)
dq.add_rear(30)
dq.display()
print("Front:", dq.peek_front())
print("Rear:", dq.peek_rear())
print("Removed from front:", dq.remove_front())
print("Removed from rear:", dq.remove_rear())
dq.display()
```

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

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