

Assignment-11.1

Name: Arjun Manoj

H.No:2303A52134

Batch:44

Task Description #1– Stack Implementation

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

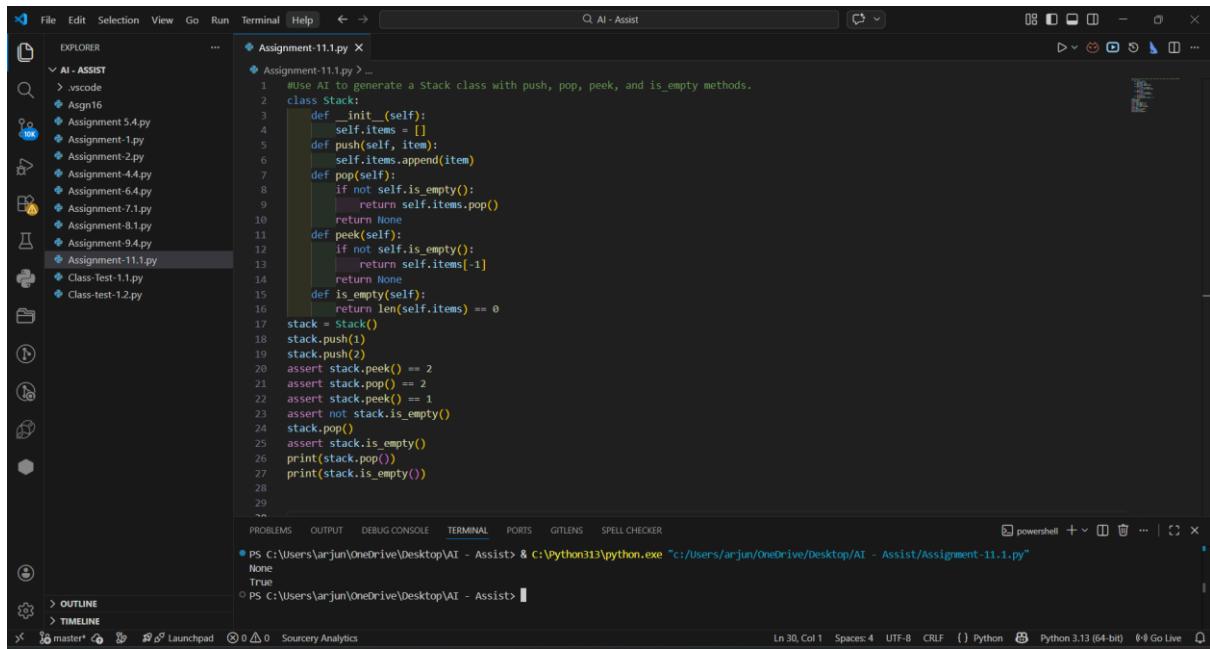
Prompt:

Use AI to generate a Python class that implements a Stack data structure with the following methods:

- `push(item)` – Add an element to the stack
- `pop()` – Remove and return the top element
- `peek()` – Return the top element without removing it
- `is_empty()` – Check whether the stack is empty

The stack should follow the LIFO (Last In First Out) principle and handle empty stack conditions safely.

Code:



```
#Use AI to generate a Stack class with push, pop, peek, and is_empty methods.
class Stack:
    def __init__(self):
        self.items = []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return None
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return None
    def is_empty(self):
        return len(self.items) == 0
stack = Stack()
stack.push(1)
stack.push(2)
assert stack.peek() == 2
assert stack.pop() == 2
assert stack.peek() == 1
assert not stack.is_empty()
stack.pop()
assert stack.is_empty()
print(stack.pop())
print(stack.is_empty())

```

Observation:

The implemented Stack class correctly follows the LIFO principle, where the last element pushed is the first to be removed. The push() method adds elements, while pop() and peek() safely handle empty stack conditions by returning None when the stack is empty. The is_empty() method efficiently checks stack status using length comparison. Overall, the implementation demonstrates proper stack behavior using Python lists

Task Description #2 – Queue Implementation

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

Prompt

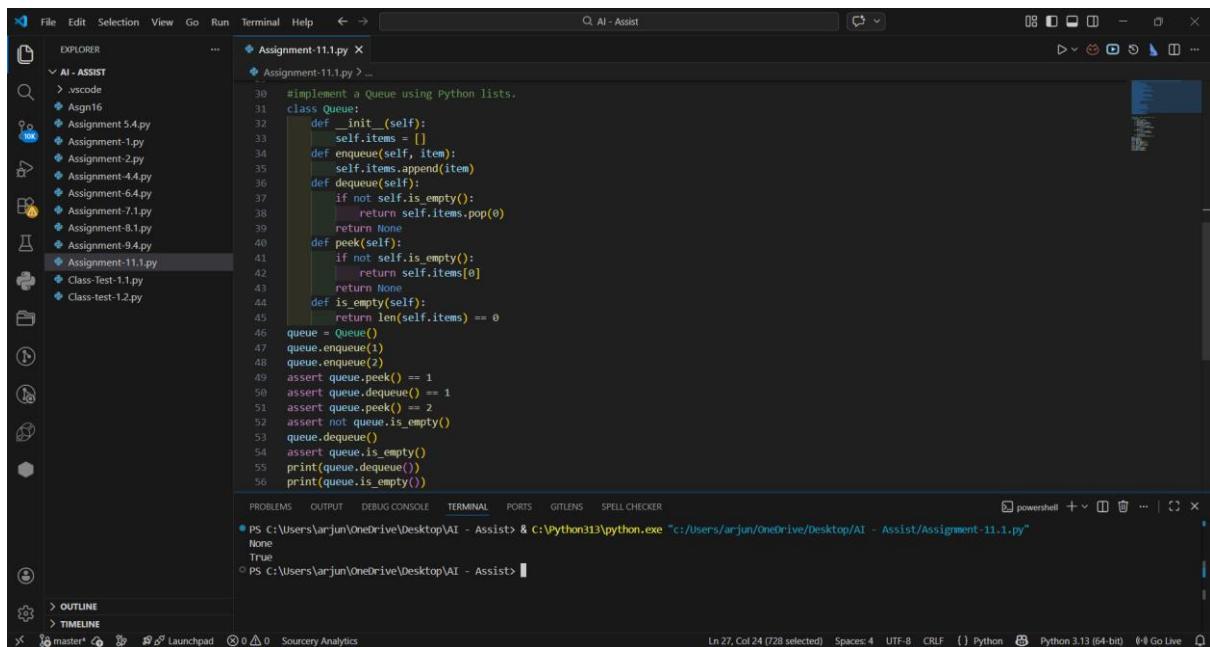
Use AI to generate a Python class that implements a Queue data structure with the following methods:

- enqueue(item) – Add an element to the rear of the queue
- dequeue() – Remove and return the front element

- peek() – Return the front element without removing it
- is_empty() – Check whether the queue is empty

The queue should follow the FIFO (First In First Out) principle and handle empty queue conditions safely.

Code:



```

File Edit Selection View Go Run Terminal Help < > AI - Assist
EXPLORER AI - ASSIST Assignment-11.1.py ...
Assignment-11.1.py > ...
30 # implement a Queue using Python lists.
31
32 class Queue:
33     def __init__(self):
34         self.items = []
35     def enqueue(self, item):
36         self.items.append(item)
37     def dequeue(self):
38         if not self.is_empty():
39             return self.items.pop(0)
40         return None
41     def peek(self):
42         if not self.is_empty():
43             return self.items[0]
44         return None
45     def is_empty(self):
46         return len(self.items) == 0
47
48 queue = Queue()
49 queue.enqueue(1)
50 queue.enqueue(2)
51 assert queue.peek() == 1
52 assert queue.dequeue() == 1
53 assert queue.peek() == 2
54 assert not queue.is_empty()
55 queue.dequeue()
56 assert queue.is_empty()
57 print(queue.dequeue())
58 print(queue.is_empty())

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & C:\Python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py"

None
True

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist>

Observation:

The implemented Queue class correctly follows the FIFO principle, where the first element inserted is the first one removed. The enqueue() method adds elements at the end, and dequeue() removes elements from the front using pop(0). The peek() method returns the front element without removal, and is_empty() checks whether the queue contains elements. The implementation works correctly, though using pop(0) has O(n) time complexity because elements are shifted after removal.

Task Description #3 – Linked List

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

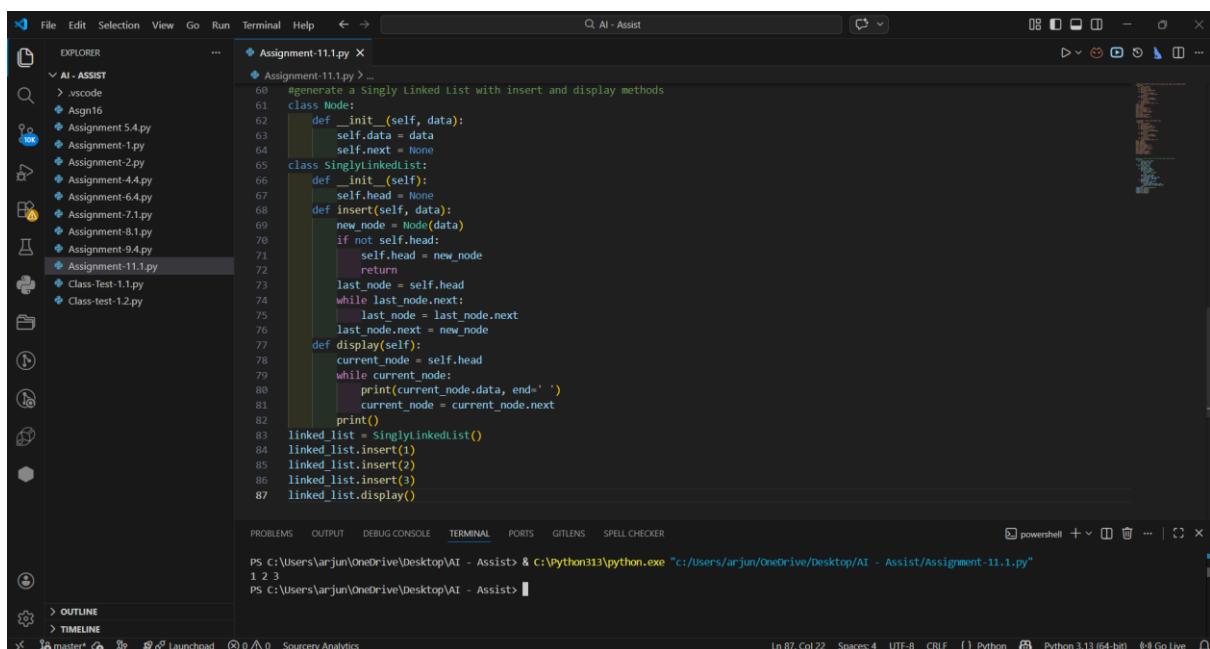
Prompt

Use AI to generate a Python program that implements a Singly Linked List with the following:

- A Node class containing data and next pointer
- An insert(data) method to add elements at the end of the list
- A display() method to print all elements in the list

The linked list should dynamically store elements and maintain proper node connections.

Code:



A screenshot of the Visual Studio Code (VS Code) interface. The left sidebar shows a file tree with several Python files listed under 'AI - ASSIST'. The main editor area displays the following Python code for a singly linked list:

```
#generate a Singly Linked List with insert and display methods
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
    def display(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
            print()
linked_list = singlyLinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()
```

The terminal tab at the bottom shows the command PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & c:/python313/python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py" followed by the output 1 2 3.

Observation:

The implemented Singly Linked List correctly maintains dynamic connections between nodes using pointers. The `insert()` method adds new elements at the end of the list by traversing to the last node and updating its next reference. The `display()` method traverses the list from head to tail and prints each element. This implementation demonstrates efficient dynamic memory usage without requiring contiguous storage like arrays.

Task Description #4 – Binary Search Tree (BST)

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

Prompt:

Use AI to generate a Python program that implements a Binary Search Tree (BST) with the following features:

- A Node class containing data, left, and right pointers
- An `insert(data)` method to add elements while maintaining BST properties
- An `in_order_traversal()` method to display elements in sorted order

The BST should follow the rule:

Left subtree < Root < Right subtree.

Code:

The screenshot shows a dark-themed code editor interface with various toolbars and panels. The main area displays Python code for a binary search tree assignment. The code includes class definitions for Node and BinarySearchTree, recursive insert methods, and an in-order traversal method. At the bottom, a main block creates a BST and prints its in-order traversal.

```
File Edit Selection View Go Run Terminal Help < - > AI - Assist
EXPLORER Assignment-11.py > Node
Assignment-11.py > Node
320 #create a BST with insert and in-order traversal methods.
321 class Node:
322     def __init__(self, data):
323         self.data = data
324         self.left = None
325         self.right = None
326 class BinarySearchTree:
327     def __init__(self):
328         self.root = None
329     def insert(self, data):
330         if self.root is None:
331             self.root = Node(data)
332         else:
333             self._insert_recursive(self.root, data)
334     def _insert_recursive(self, node, data):
335         if data < node.data:
336             if node.left is None:
337                 node.left = Node(data)
338             else:
339                 self._insert_recursive(node.left, data)
340         else:
341             if node.right is None:
342                 node.right = Node(data)
343             else:
344                 self._insert_recursive(node.right, data)
345     def in_order_traversal(self):
346         return self._in_order_recursive(self.root)
347     def _in_order_recursive(self, node):
348         res = []
349         if node:
350             res = self._in_order_recursive(node.left)
351             res.append(node.data)
352             res = res + self._in_order_recursive(node.right)
353         return res
354 bst = BinarySearchTree()
355 bst.insert(5)
356 bst.insert(3)
357 bst.insert(7)
358 print(bst.in_order_traversal())
```

Observation:

The implemented Binary Search Tree correctly maintains the BST property during insertion by placing smaller values in the left subtree and larger values in the right subtree. The recursive insertion method ensures proper node placement. The `in_order_traversal()` method visits nodes in Left → Root → Right order, producing a sorted list of elements. This demonstrates how BST enables efficient searching and sorted data retrieval.

Task Description #5 – Hash Table

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

Prompt:

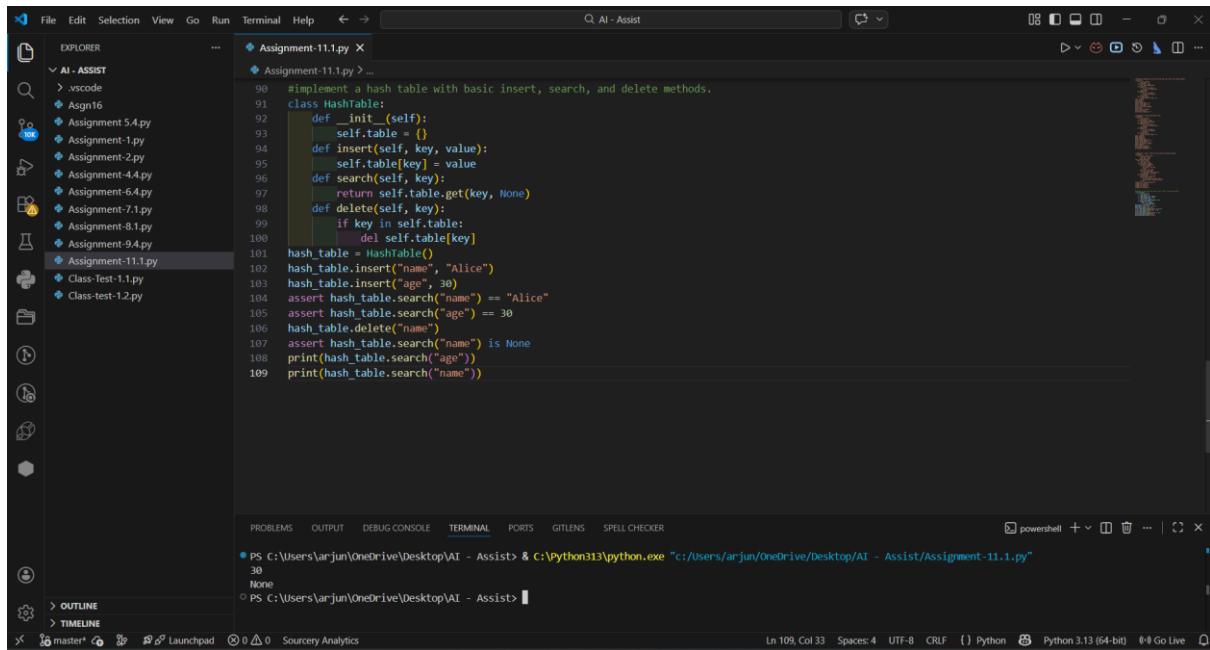
Use AI to generate a Python program that implements a Hash Table with the following methods:

- `insert(key, value)` – Store a key-value pair

- `search(key)` – Retrieve value using key
- `delete(key)` – Remove a key-value pair

The hash table should allow fast lookup and handle cases where the key does not exist.

Code:



```

File Edit Selection View Go Run Terminal Help < > AI - Assist
EXPLORER ... Assignment-11.1.py ...
AI - ASSIST > vscode
Assignment-11.1.py
Assignment-5.4.py
Assignment-1.py
Assignment-2.py
Assignment-4.py
Assignment-6.4.py
Assignment-7.1.py
Assignment-8.1.py
Assignment-9.4.py
Assignment-11.1.py
Class-Test-1.1.py
Class-test-1.2.py
90 # implement a hash table with basic insert, search, and delete methods.
91 class HashTable:
92     def __init__(self):
93         self.table = {}
94     def insert(self, key, value):
95         self.table[key] = value
96     def search(self, key):
97         return self.table.get(key, None)
98     def delete(self, key):
99         if key in self.table:
100             del self.table[key]
101
102 hash_table = HashTable()
103 hash_table.insert("name", "Alice")
104 hash_table.insert("age", 30)
105 assert hash_table.search("name") == "Alice"
106 assert hash_table.search("age") == 30
107 hash_table.delete("name")
108 assert hash_table.search("name") is None
109 print(hash_table.search("age"))
110 print(hash_table.search("name"))

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & C:\Python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py"

None

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist>

Ln 109, Col 33 Spaces: 4 UFT-8 CRLF Python 3.13 (64-bit) 8:0 Go Live

Observation:

The implemented Hash Table uses Python's built-in dictionary to store key-value pairs efficiently. The `insert()` method adds or updates entries, `search()` retrieves values in average $O(1)$ time, and `delete()` removes entries if they exist. The implementation demonstrates how hashing enables fast data access and efficient key-based storage.

Task Description #6 – Graph Representation

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

Prompt:

Use AI to generate a Python program that implements a **Graph** using an **Adjacency List** representation.

The graph should include:

- `add_vertex(vertex)` – Add a new vertex
- `add_edge(vertex1, vertex2)` – Add an undirected edge between two vertices
- `display()` – Print all vertices and their connected edges

The graph should dynamically store connections between vertices.

Code:

A screenshot of the Visual Studio Code (VS Code) interface. The left sidebar shows the Explorer view with several Python files listed under 'AI - ASSIST'. The main editor area displays the code for 'Assignment-11.1.py'. The code defines a 'Graph' class with methods for initializing the adjacency list, adding vertices, adding edges, and displaying the graph. The terminal at the bottom shows the output of running the script, which creates vertices A, B, and C, and adds edges between them, resulting in the output: 'A: [B, C]', 'B: [A]', and 'C: [A]'. The status bar at the bottom indicates the file is 'master*' and the code is in Python 3.13 (64-bit).

```
112 # implement a graph using an adjacency list.
113
114 class Graph:
115     def __init__(self):
116         self.adjacency_list = {}
117     def add_vertex(self, vertex):
118         if vertex not in self.adjacency_list:
119             self.adjacency_list[vertex] = []
120     def add_edge(self, vertex1, vertex2):
121         if vertex1 in self.adjacency_list and vertex2 in self.adjacency_list:
122             self.adjacency_list[vertex1].append(vertex2)
123             self.adjacency_list[vertex2].append(vertex1)
124     def display(self):
125         for vertex, edges in self.adjacency_list.items():
126             print(f'{vertex}: {edges}')
127
128 graph = Graph()
129 graph.add_vertex("A")
130 graph.add_vertex("B")
131 graph.add_vertex("C")
132 graph.add_edge("A", "B")
133 graph.add_edge("A", "C")
134 graph.display()
```

Observation:

The implemented Graph uses a dictionary to represent the adjacency list, where each vertex maps to a list of connected vertices. The `add_vertex()` method ensures vertices are created before edges are

added, and add_edge() connects two vertices in both directions, forming an undirected graph. This structure efficiently represents relationships between nodes and is space-efficient for sparse graphs.

Task Description #7 – Priority Queue

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

Prompt:

Use AI to generate a Python program that implements a Priority Queue using Python's built-in heapq module.

The class should include:

- push(item, priority) – Insert an element with a given priority
- pop() – Remove and return the element with the highest priority (lowest priority number in min-heap)
- peek() – View the highest priority element without removing it
- is_empty() – Check whether the priority queue is empty

The implementation should follow the heap property for efficient priority-based retrieval.

Code:

```

Assignment-11.1.py ...
Assignment-11.1.py ...
135     #implement a priority queue using Python's heapq module.
136     import heapq
137     class PriorityQueue:
138         def __init__(self):
139             self.elements = []
140         def push(self, item, priority):
141             heapq.heappush(self.elements, (priority, item))
142         def pop(self):
143             return heapq.heappop(self.elements)[1] if self.elements else None
144         def peek(self):
145             return self.elements[0][1] if self.elements else None
146         def is_empty(self):
147             return len(self.elements) == 0
148
149 pq = PriorityQueue()
150 pq.push("task1", priority=2)
151 pq.push("task2", priority=1)
152 assert pq.peek() == "task2"
153 assert pq.pop() == "task2"
154 assert pq.peek() == "task1"
155 assert not pq.is_empty()
156 assert pq.pop()
157 print(pq.pop())
158 print(pq.is_empty())

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS SPELL CHECKER

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & C:\Python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py"

None
True

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist>

Ln 158, Col 21 Spaces: 4 UTF-8 CRLF {} Python Python 3.13 (64-bit) Go Live

Observation:

The implemented Priority Queue uses Python's heapq, which internally maintains a min-heap structure. Elements are stored as (priority, item) tuples, ensuring that the item with the smallest priority value is removed first. The push() and pop() operations run in O(log n) time, while peek() operates in O(1) time. This implementation efficiently manages tasks based on priority rather than insertion order.

Task Description #8 – Deque

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

Prompt:

Use AI to generate a Python program that implements a Double-Ended Queue (Deque) using Python's built-in collections.deque module.

The class should include:

- `append(item)` – Add element to the right end
- `appendleft(item)` – Add element to the left end
- `pop()` – Remove and return element from the right end
- `popleft()` – Remove and return element from the left end
- `peek()` – View the front element
- `is_empty()` – Check whether the deque is empty

The implementation should allow insertion and deletion from both ends efficiently.

Code:

```
# implement a double-ended queue using collections.deque
from collections import deque
class Deque:
    def __init__(self):
        self.deque = deque()
    def append(self, item):
        self.deque.append(item)
    def appendleft(self, item):
        self.deque.appendleft(item)
    def pop(self):
        return self.deque.pop() if self.deque else None
    def popleft(self):
        return self.deque.popleft() if self.deque else None
    def peek(self):
        return self.deque[0] if self.deque else None
    def is_empty(self):
        return len(self.deque) == 0
dq = Deque()
dq.append("item1")
dq.appendleft("item2")
assert dq.peek() == "item2"
assert dq.pop() == "item1"
assert dq.peek() == "item2"
assert not dq.is_empty()
dq.popleft()
assert dq.is_empty()
print(dq.pop())
print(dq.is_empty())

```

TERMINAL

```
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & C:\Python313\python.exe "c:/Users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py"
None
True
PS C:\Users\arjun\OneDrive\Desktop\AI - Assist>
```

Observation:

The implemented Deque uses Python's `collections.deque`, which allows efficient insertion and deletion from both ends in $O(1)$ time. The `append()` and `appendleft()` methods add elements to the rear and

front respectively, while pop() and popleft() remove elements from both ends. The peek() method retrieves the front element without removing it. This structure is flexible and combines features of both Stack and Queue.

Task Description #9 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.**

Prompt:

Design and implement a Campus Resource Management System that efficiently manages multiple campus operations using appropriate data structures.

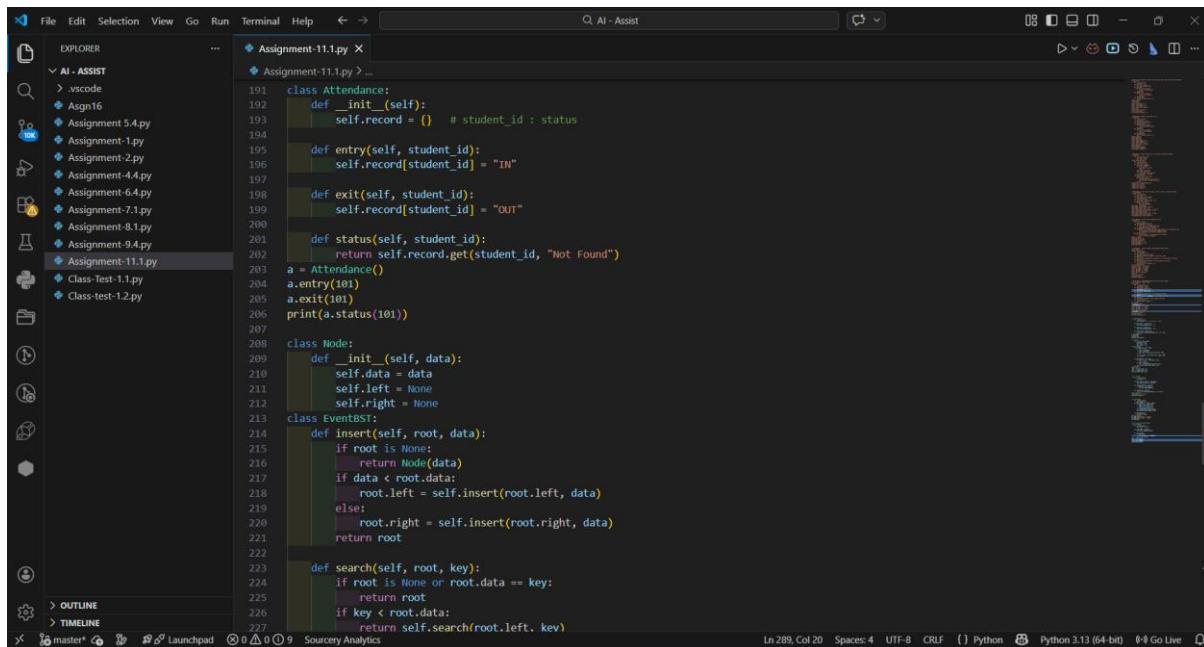
Each feature must use a suitable data structure based on its operational requirements such as fast search, insertion, deletion, ordering, or connectivity handling.

The system includes:

1. Student Attendance Tracking
2. Event Registration System
3. Library Book Borrowing
4. Bus Scheduling System
5. Cafeteria Order Queue

Each module is implemented independently using the most efficient data structure for that task.

Code:



A screenshot of the Visual Studio Code (VS Code) interface. The left sidebar shows a file tree with several Python files under the 'AI - ASSIST' folder. The main editor area displays a Python script named 'Assignment-11.1.py'. The code defines a class 'Attendance' with methods for entry, exit, and status. It also defines a class 'Node' and an 'EventBST' class that uses a binary search tree structure. The code is color-coded for syntax highlighting. The bottom status bar shows the file path as 'master' and other standard VS Code status indicators.

```
191 class Attendance:
192     def __init__(self):
193         self.record = {} # student_id : status
194
195     def entry(self, student_id):
196         self.record[student_id] = "IN"
197
198     def exit(self, student_id):
199         self.record[student_id] = "OUT"
200
201     def status(self, student_id):
202         return self.record.get(student_id, "Not Found")
203
204 a = Attendance()
205 a.entry(101)
206 a.exit(101)
207 print(a.status(101))
208
209 class Node:
210     def __init__(self, data):
211         self.data = data
212         self.left = None
213         self.right = None
214
215     def insert(self, root, data):
216         if root is None:
217             return Node(data)
218         if data < root.data:
219             root.left = self.insert(root.left, data)
220         else:
221             root.right = self.insert(root.right, data)
222
223         return root
224
225     def search(self, root, key):
226         if root is None or root.data == key:
227             return root
228         if key < root.data:
229             return self.search(root.left, key)
230         else:
231             return self.search(root.right, key)
```

File Edit Selection View Go Run Terminal Help ↻ 🔍

Q AI - Assist

EXPLORER

AI - ASSIST

Assignment-11.py

Assignment-11.py

```
213     class EventBST:
214         def search(self, root, key):
215             if root is None or root.data == key:
216                 return root
217             if key < root.data:
218                 return self.search(root.left, key)
219             return self.search(root.right, key)
220
221     e = EventBST()
222     root = None
223     root = e.insert(root, 50)
224     root = e.insert(root, 30)
225     print(e.search(root, 30))
226
227     class Library:
228         def __init__(self):
229             self.books = {}
230
231         def borrow(self, book_id, due_date):
232             self.books[book_id] = due_date
233
234         def return_book(self, book_id):
235             if book_id in self.books:
236                 del self.books[book_id]
237
238         def check(self, book_id):
239             return self.books.get(book_id, "Available")
240
241     lib = Library()
242     lib.borrow(1, "10 Feb")
243     print(lib.check(1))
244
245     class BusGraph:
246         def __init__(self):
247             self.graph = {}
248
249         def add_route(self, stop1, stop2):
250             if stop1 not in self.graph:
251                 self.graph[stop1] = []
252             if stop2 not in self.graph:
253                 self.graph[stop2] = []
254             self.graph[stop1].append(stop2)
255             self.graph[stop2].append(stop1)
256
257         def display(self):
258             print(self.graph)
259
260 bus = BusGraph()
261 bus.add_route("StopA", "StopB")
262 bus.add_route("StopB", "StopC")
263 bus.display()
```

Ln 224, Col 45 Spaces: 4 UTF-8 CRLF ⚡ Python 🌐 Python 3.13 (64-bit) ⚡ Go Live

File Edit Selection View Go Run Terminal Help ↻ 🔍

Q AI - Assist

EXPLORER

AI - ASSIST

Assignment-11.py

Assignment-11.py

```
255     class BusGraph:
256         def __init__(self):
257             self.graph = {}
258         def add_route(self, stop1, stop2):
259             if stop1 not in self.graph:
260                 self.graph[stop1] = []
261             if stop2 not in self.graph:
262                 self.graph[stop2] = []
263             self.graph[stop1].append(stop2)
264             self.graph[stop2].append(stop1)
265
266         def display(self):
267             print(self.graph)
268
269 bus = BusGraph()
270 bus.add_route("StopA", "StopB")
271 bus.add_route("StopB", "StopC")
272 bus.display()
273
274 from collections import deque
275 class Cafeteria:
276     def __init__(self):
277         self.queue = deque()
278
279     def order(self, student):
280         self.queue.append(student)
281
282     def serve(self):
283         if self.queue:
284             return self.queue.popleft()
285         return "No Orders"
286
287 cafe = Cafeteria()
288 cafe.order("Sushma")
289 cafe.order("Arjun")
290 print(cafe.serve())
```

```

class Cafeteria:
    def order(self, student):
        self.queue.append(student)

    def serve(self):
        if self.queue:
            return self.queue.pop(0)
        return "No orders"

cafe = Cafeteria()
cafe.order("Sushma")
cafe.order("Arjun")
print(cafe.serve())

```

PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & C:\Python313\python.exe "c:/users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py"
 PS C:\Users\arjun\OneDrive\Desktop\AI - Assist> & C:\Python313\python.exe "c:/users/arjun/OneDrive/Desktop/AI - Assist/Assignment-11.1.py"
 OUT
 <__main__.Node object at 0x00000263E7808A50>
 10 Feb
 {'StopA': ['StopB'], 'StopB': ['StopA', 'StopC'], 'StopC': ['StopB']}
 Sushma

Observation:

The system demonstrates how different real-world problems require different data structures for optimal performance.

- Hash Tables provide constant-time access for tracking attendance and books.
- Binary Search Tree enables structured storage with efficient search and removal.
- Graph effectively models interconnected bus routes.
- Queue ensures fair servicing in cafeteria using FIFO principle.

This implementation shows that choosing the correct data structure improves efficiency, maintainability, and scalability of a system.

Task Description #10: Smart E-Commerce Platform – Data Structure

Challenge

An e-commerce company wants to build a Smart Online Shopping System

with:

- 1. Shopping Cart Management – Add and remove products dynamically.**
- 2. Order Processing System – Orders processed in the order they are placed.**
- 3. Top-Selling Products Tracker – Products ranked by sales count.**
- 4. Product Search Engine – Fast lookup of products using product ID.**
- 5. Delivery Route Planning – Connect warehouses and delivery locations.**

Prompt:

Design a Smart E-Commerce Platform that efficiently manages shopping operations using suitable data structures. Each feature must use the most appropriate data structure based on operational requirements such as dynamic insertion, ranking, fast searching, order processing, and network connectivity.

Code:

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

- File Explorer:** Shows a folder named "AI - ASSIST" containing several Python files: Assignment 5.4.py, Assignment 1.py, Assignment 4.4.py, Assignment 6.4.py, Assignment 7.1.py, Assignment 8.1.py, Assignment 9.4.py, Assignment 11.1.py (which is the active file), Class-Test-1.1.py, and Class-test-1.2.py.
- Code Editor:** Displays the content of `Assignment-11.1.py`. The code defines a class `OrderProcessingSystem` that uses a deque to manage orders. It includes methods for placing orders, processing them, and displaying pending orders. The code is annotated with comments explaining its functionality.
- Terminal:** Shows the output of running the script, indicating successful order placement and processing.
- Status Bar:** Provides information about the current file path (C:\Users\arjun\OneDrive\Desktop\AI - Assist>), line number (Ln 316), column number (Col 24), and encoding (UTF-8).

Observation:

Different modules in an e-commerce system require different data structures to achieve efficiency. Queue ensures fair order processing, Linked List allows flexible cart updates, Priority Queue helps in ranking top-selling products, Hash Table enables instant product lookup, and Graph models delivery routes effectively. Selecting the correct data structure improves system performance and scalability.