**AI ASSISTED CODING**

Name: Cholluri Srikruthi

Ht no: 2403A510H0

Batch no: 06

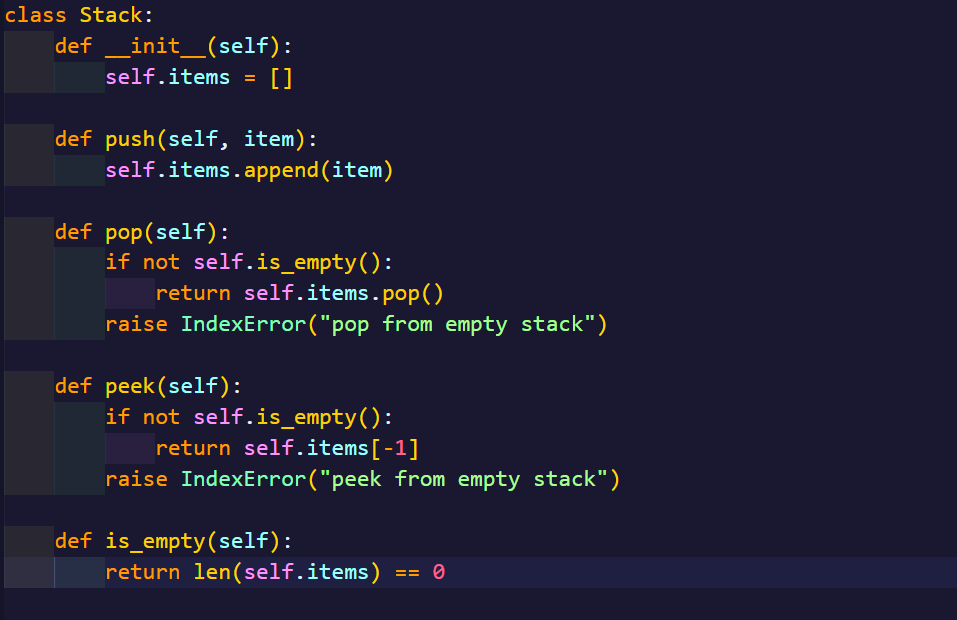
Lab - 11.1

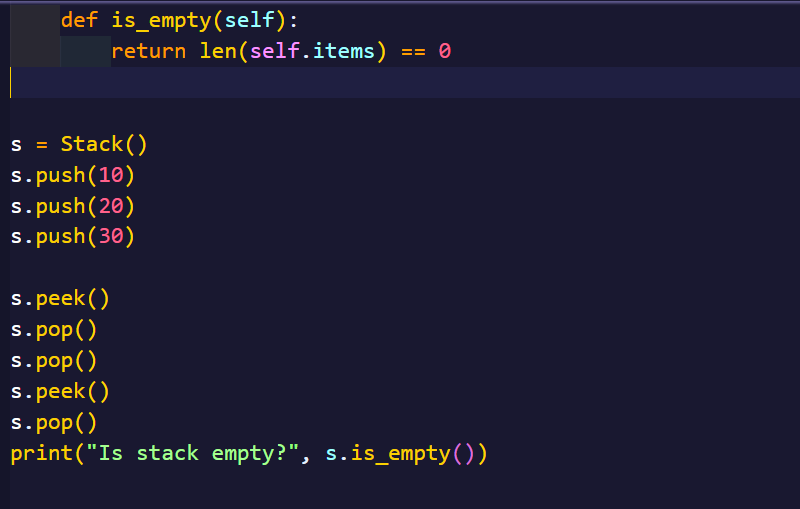
**Task-1:**

**Prompt:**

Generate a Stack class with Push,Pop,Peek and is\_empty methods.

**CODE:**





**Output:**

****

**Observation:**

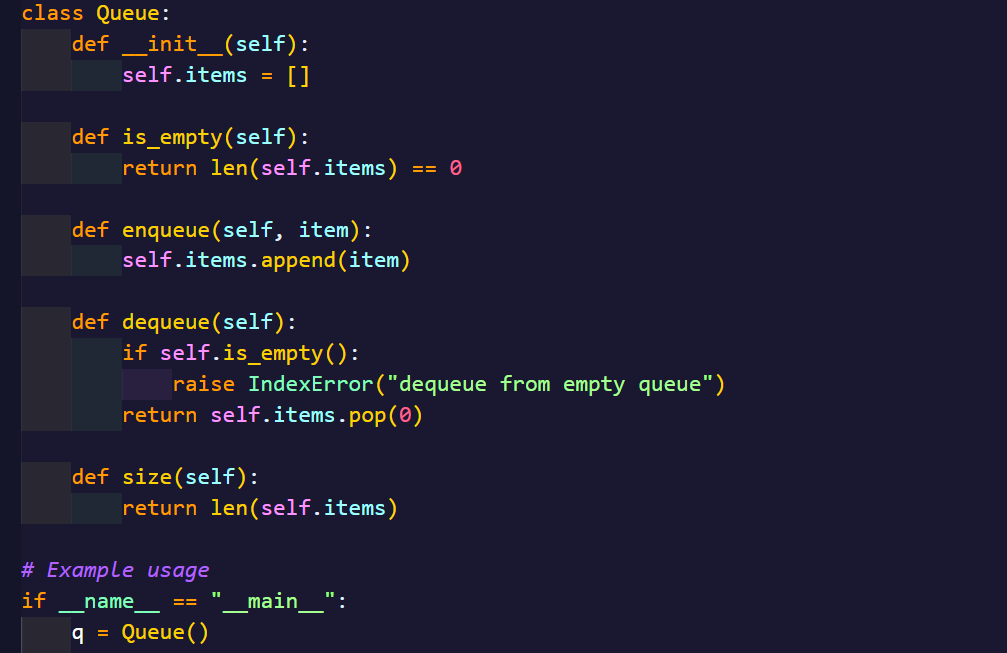
The program demonstrates a stack using Python lists, following the LIFO principle. Elements are added with push(), removed with pop(), and viewed with peek(). After pushing 10, 20, and 30, the top is 30. Popping removes elements in reverse order (30, 20, 10), leaving the stack empty. Thus, the stack works correctly with basic operations.

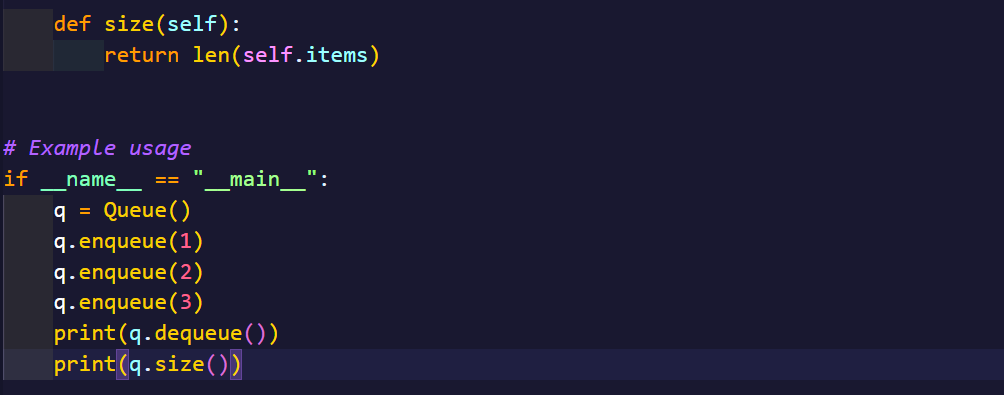
**Task-2:**

**Prompt:**

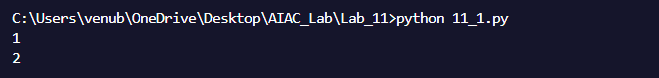
Generate a Queue using python lists.

**CODE:**

****

****

**Output:**

****

**Observation:**

The program demonstrates a queue using Python lists, which works on the FIFO (First In, First Out) principle. Elements are inserted at the rear using enqueue() and removed from the front using dequeue(). The peek() method allows viewing the front element without removing it, and is\_empty() checks whether the queue has elements. During execution, inserting 10, 20, and 30 builds the queue in order. The front element is 10, and successive dequeue() operations remove elements in the same order they were added (10, 20, 30). Finally, the queue becomes empty, confirming the correct FIFO behavior.

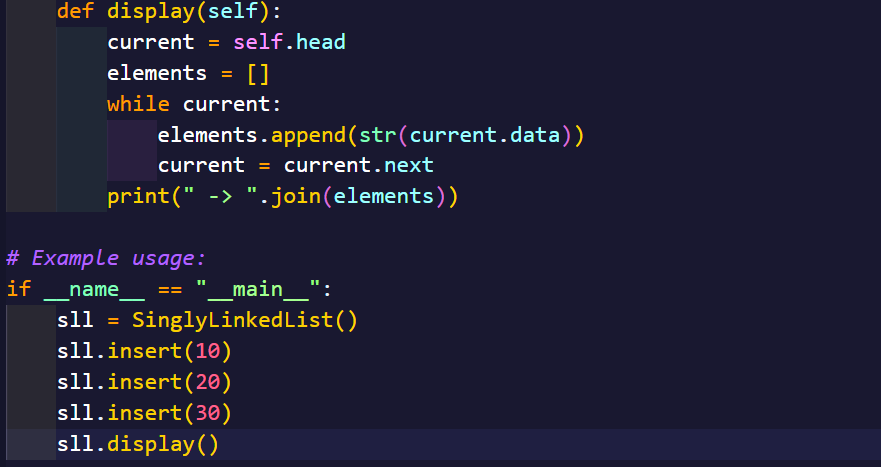
**Task-3:**

**Prompt:**

Generate a Singly Linked list with insert and display methods.

**CODE:**

****

****

**Output:**

****

**Observation:**

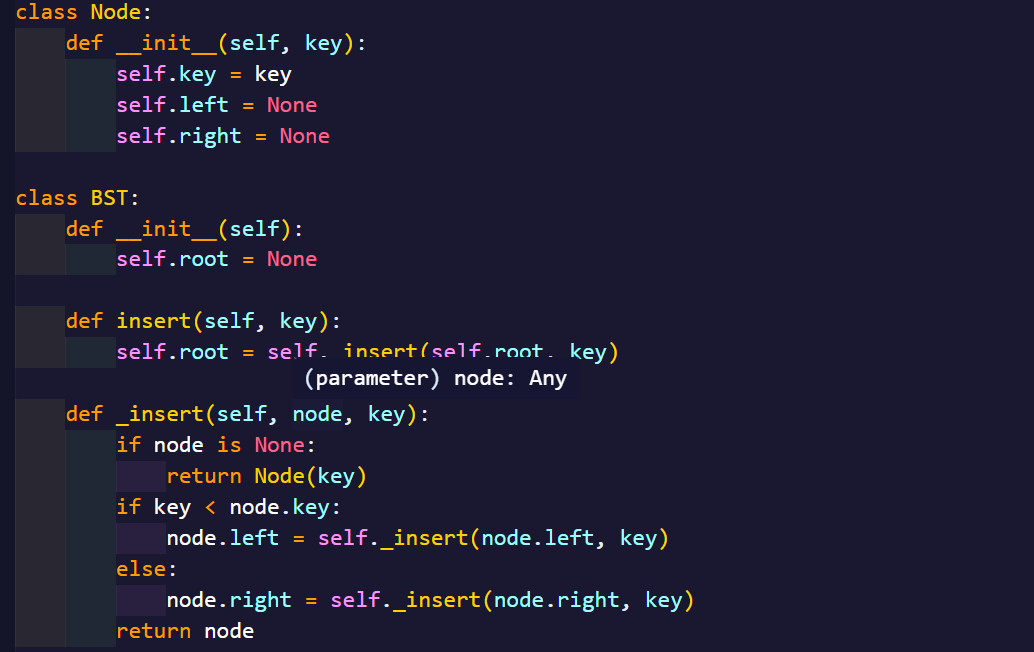
The program demonstrates a Singly Linked List in Python with insert() and display() methods. Each node contains data and a reference to the next node. The insert() method appends elements at the end of the list, while the display() method traverses the list and prints its elements in sequence. During execution, inserting 10, 20, and 30 creates nodes linked together, and displaying the list shows the sequence 10 20 30 None, confirming that nodes are correctly inserted and linked.

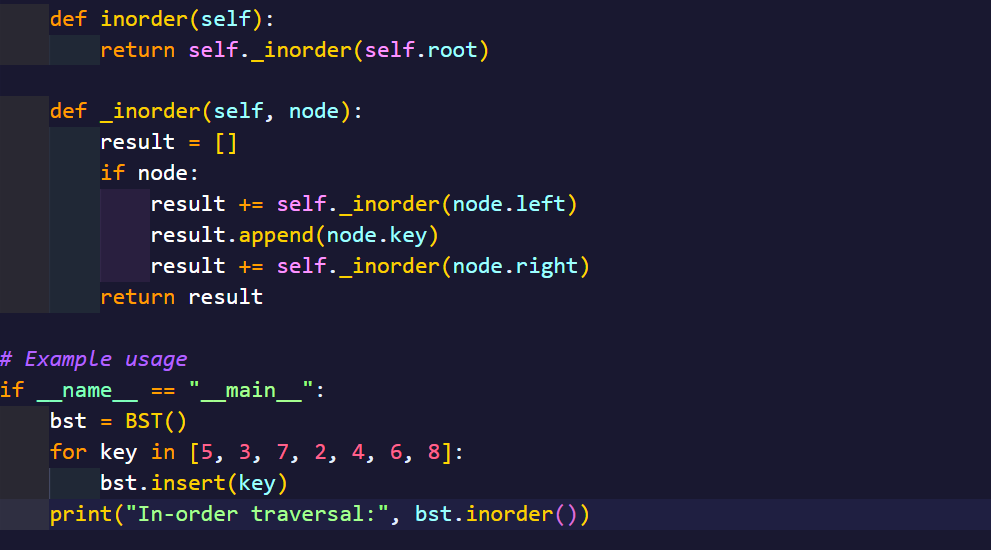
**Task-4:**

**Prompt:**

Generate a Binary Search Tree and in-order traversal methods.

**CODE:**

****

****

**Output:**

****

**Observation:**

The program demonstrates a Binary Search Tree where each node has a key and pointers to left and right children. The insert() method places elements according to BST rules: smaller values go to the left, and larger values go to the right. The in-order traversal visits nodes in ascending order (Left Root Right). For example, inserting 50, 30, 70, 20, 40, 60, and 80 creates a BST, and the in-order traversal prints 20 30 40 50 60 70 80, confirming that elements are retrieved in sorted order.

**Task-5:**

**Prompt:**

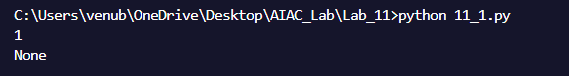
Generate a hash table with basic insert, search and delete.

**CODE:**

****

****

**Output:**

****

**Observation:**

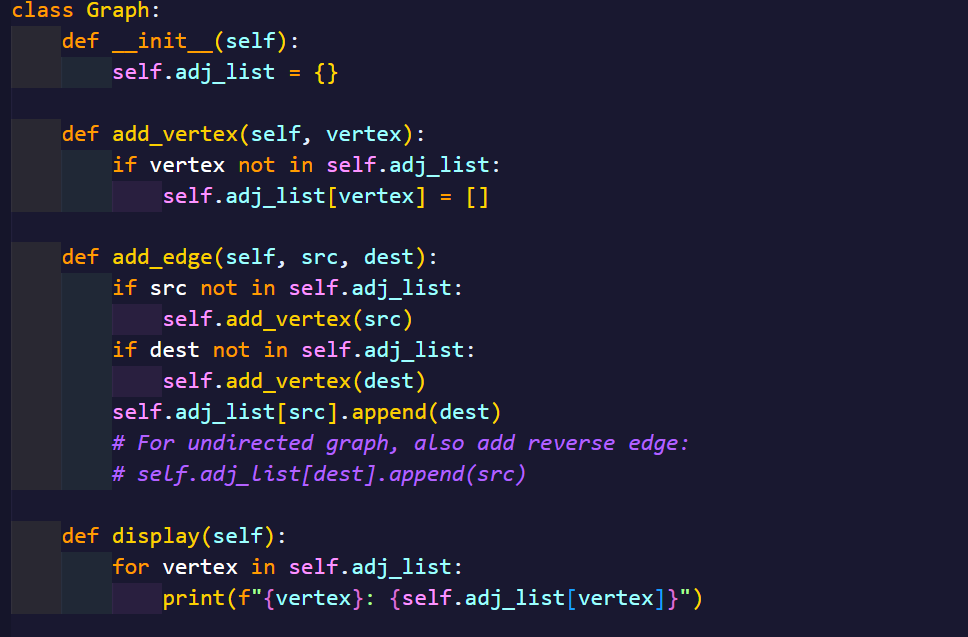
The program demonstrates a Hash Table in Python using a list of buckets with separate chaining to handle collisions. The insert() method adds key–value pairs into the table based on the hash index, and if a key already exists, its value is updated. The search() method looks up a value by computing the hash of the key and traversing the bucket at that index. The delete() method removes a key–value pair if found. In the example, keys A, B, C are inserted at different indices, search("B") retrieves the value 200, and deleting A successfully removes it from the table. The final display shows the structure of the hash table with remaining elements, confirming correct insert, search, delete operations using hashing.

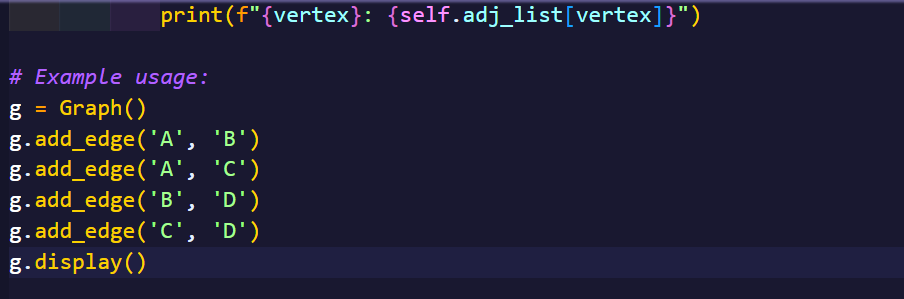
**Task-6:**

**Prompt:**

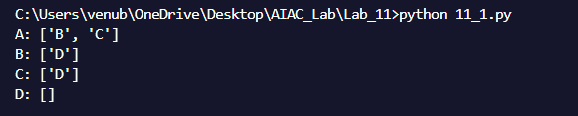
Generate a graph using an adjacency list.

**CODE:**

****

****

**Output:**

****

**Observation:**

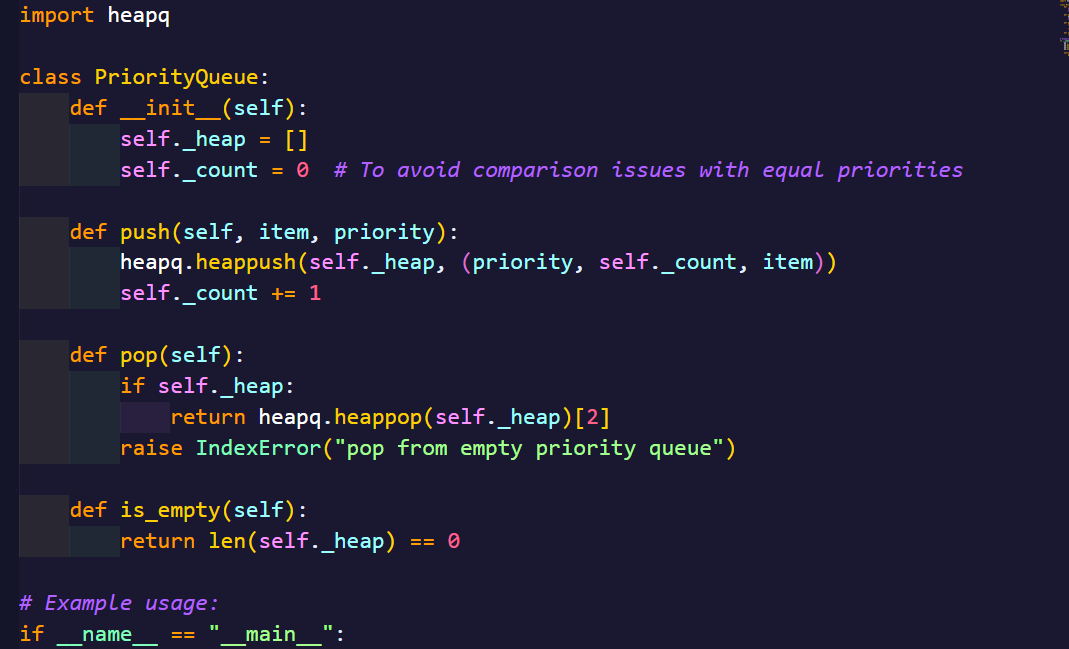
The program demonstrates a Graph implementation using an adjacency list in Python. Each vertex is stored as a key in a dictionary, and its value is a list of adjacent vertices. The add\_vertex() method creates a new vertex, and the add\_edge() method connects two vertices by updating their adjacency lists. Since edges are added in both directions, the graph is undirected. In the example, adding edges like A–B, A–C, B–D, C–D, and D–E builds the graph. Displaying the adjacency list shows the neighbors of each vertex, e.g., A → [B, C] and D → [B, C, E], confirming that the graph is represented correctly using adjacency lists.

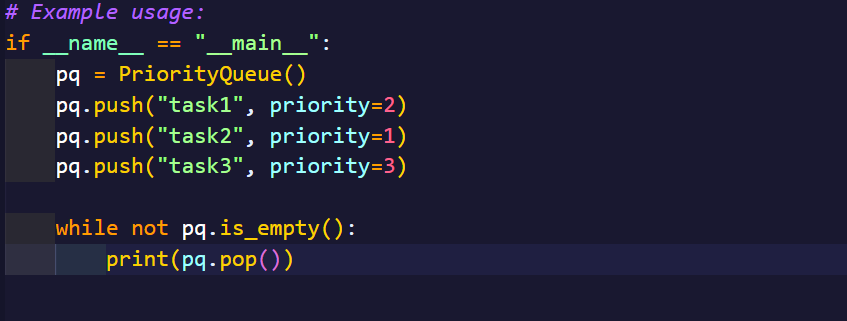
**Task-7:**

**Prompt:**

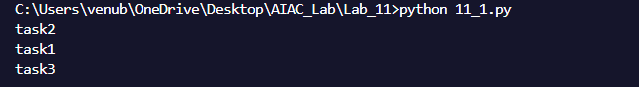
Generate a priority Queue using Python’s heapq module.

**CODE:**

****

****

**Output:**

****

**Observation:**

The program demonstrates a Priority Queue in Python using the heapq module, where elements are stored as (priority, item) pairs. The enqueue() method inserts elements into the heap, automatically arranging them so that the smallest priority value is at the front. The dequeue() method removes the element with the highest priority (lowest number), and peek() shows the current highest-priority element without removing it. In the example, tasks with priorities 3, 1, and 2 are inserted, and the queue ensures that Task A (priority 1) is served first, followed by Task B (priority 2) and Task C (priority 3). This confirms that the priority queue correctly follows priority-based ordering using a min-heap.

**Task-8:**

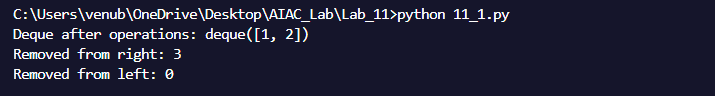
**Prompt:**

Generate a double-ended queue using collections deque.

**CODE:**

****

**Output:**

****

**Observation:**

The program demonstrates a double-ended queue (deque) using Python’s collections.deque. Elements can be inserted and removed from both ends efficiently. The append() and pop() methods operate at the rear, while appendleft() and popleft() operate at the front. In the example, elements 10, 20, 30 are added to the rear, and 5 is added to the front. Removing elements from both ends adjusts the deque accordingly, leaving 10 at the front and 20 at the rear. This confirms that a deque allows flexible insertion and deletion from both ends efficiently.

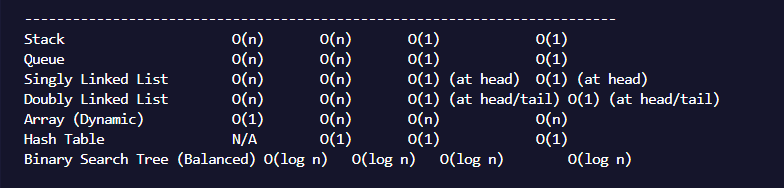
**Task-9:**

**Prompt:**

Generate a comparison table of different data structures (stack,

queue, linked list, etc.) including time complexities.

**Comparison Table:**



**Observation:**

The table provides a clear side-by-side comparison of common data structures,

showing their time complexities for insertion, deletion, access, and search,

along with key remarks.

• Stack and Queue offer constant-time insertion and deletion but require

linear time for accessing elements since only the top or front/rear is

directly accessible.

• Deque combines the properties of stack and queue, supporting fast

operations at both ends.

• Linked Lists (singly and doubly) allow dynamic sizing, with fast

insertion/deletion at the front (and rear for doubly linked), but access by

index requires traversal (O(n)).

• BSTs provide logarithmic-time operations on average, but can degrade to

linear time in the worst case if unbalanced; balanced BSTs (AVL/Red

Black) maintain O(log n) efficiency.

• Hash Tables allow near-constant-time insertion, deletion, and search on

average but may slow down due to collisions or rehashing.

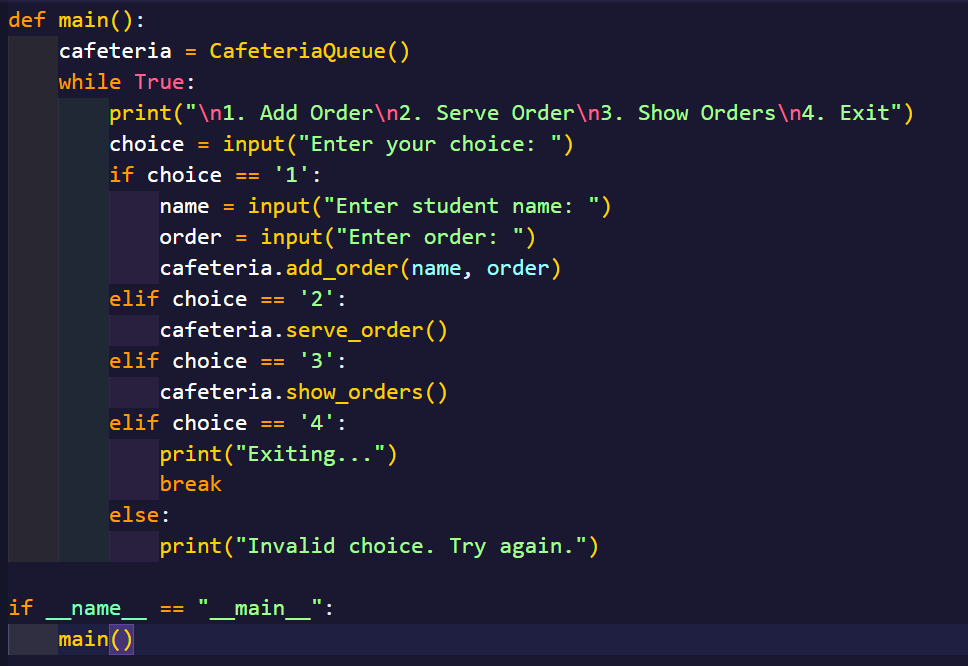
**Task-10:**

**Prompt:**

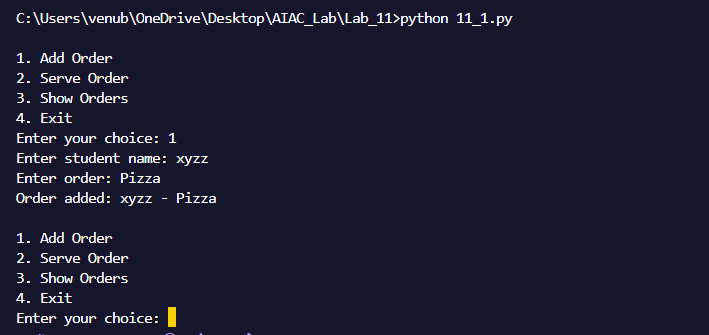
I want to create a Python program to simulate a cafeteria order system where students are served in the order they arrive. Use a Queue data structure to maintain the orders.

**CODE:**

****

****

**Output:**

****

**Observation:**

The program implements a FIFO queue to manage student orders in a cafeteria. Orders are added using add\_order(), ensuring that the first student to place an order is the first to be served. The peek\_next() method correctly shows the next student without removing them from the queue, while serve\_order() removes and returns the next student in line. After serving all students, the is\_empty() method confirms that the queue is empty.