

Name:Dugyala Ashmitha

Enrollno:2403a510g5

Assignment:11.1

Task1:

Prompt:

Generate a Stack class with push, pop, peek, and is empty methods.

Code:

```

1 class Stack:
2     def __init__(self):
3         self.items = [] # initialize an empty list for the stack
4
5     def push(self, item):
6         """Add an item to the top of the stack"""
7         self.items.append(item)
8
9     def pop(self):
10        """Remove and return the top item of the stack"""
11        if not self.is_empty():
12            return self.items.pop()
13        return "Stack is empty"
14
15    def peek(self):
16        """Return the top item without removing it"""
17        if not self.is_empty():
18            return self.items[-1]
19        return "Stack is empty"
20
21    def is_empty(self):
22        """Check if the stack is empty"""
23        return len(self.items) == 0
24
25
26    # Example usage:
27    s = Stack()
28    s.push(10)
29    s.push(20)
30    print(s.peek())    # 20
31    print(s.pop())     # 20
32    print(s.is_empty()) # False
33    print(s.pop())     # 10
34    print(s.is_empty()) # True
35

```

Output:

```

PS C:\Users\ASHMITHA\Desktop\ai_ ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai_ ass/task1.py"
20
20
False
10
True

```

Observation: A stack object is created using `stack = Stack()`.

At this point, the stack is empty → `stack.is_empty()` returns True.

Push Operation

Elements (5, 10, 15) are pushed one by one.

Internally, they are stored in a Python list → [5, 10, 15].

Each push() call adds the element to the top of the stack (end of the list).

Peek Operation

stack.peek() checks the last element without removing it.

Returns 15 (the top element) while keeping the stack as [5, 10, 15].

Pop Operation

stack.pop() removes and returns the top element.

Returns 15 and updates the stack to [5, 10].

Peek after Pop

stack.peek() now returns 10 (new top of the stack).

Final Check

stack.is_empty() returns False since stack still has [5, 10].

Task2:

Prompt:

Implement a Queue using Python lists.

Code:

```

task2.py > q
1 class Queue:
2     def __init__(self):
3         self.items = []
4
5     def enqueue(self, item):
6         """Add an item to the end of the queue"""
7         self.items.append(item)
8
9     def dequeue(self):
10        """Remove and return the item from the front of the queue"""
11        if not self.is_empty():
12            return self.items.pop(0)
13        return "Queue is empty"
14
15    def peek(self):
16        """Return the front item without removing it"""
17        if not self.is_empty():
18            return self.items[0]
19        return "Queue is empty"
20
21    def is_empty(self):
22        """Check if the queue is empty"""
23        return len(self.items) == 0
24    def size(self):
25        """Return the number of items in the queue"""
26        return len(self.items)
27
28 q = Queue()
29 q.enqueue(10)
30 q.enqueue(20)
31 q.enqueue(30)
32
33 print(q.peek())      # 10 (front of queue)
34 print(q.dequeue())   # 10 (removed from front)
35 print(q.size())      # 2
36 print(q.is_empty()) # False

```

Output:

```

PS C:\Users\ASHMITHA\Desktop\ai ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai ass/task2.py"
10
10
2
False

```

Observation:

The queue works on the **FIFO (First-In, First-Out)** principle. When elements 10, 20, and 30 are enqueued, they are stored in the order [10, 20, 30]. The peek() method shows 10 as the front element without removing it. The dequeue() operation removes the first inserted element (10), leaving [20, 30] in the queue. The size() method then shows 2, and the is_empty() method correctly returns False since the queue still contains elements.

Task3:

Prompt:

generate a Singly Linked List with insert and display methods.

Code:

```
task3.py > ...
1  class Node:
2      def __init__(self, data):
3          self.data = data      # store data
4          self.next = None     # pointer to the next node
5
6
7  class SinglyLinkedList:
8      def __init__(self):
9          self.head = None     # start with an empty list
10
11     def insert(self, data):
12         """Insert a new node at the end of the linked list"""
13         new_node = Node(data)
14         if self.head is None:  # if list is empty
15             self.head = new_node
16         else:
17             temp = self.head
18             while temp.next:    # traverse until last node
19                 temp = temp.next
20             temp.next = new_node # link new node at the end
21
22     def display(self):
23         """Display all nodes in the linked list"""
24         if self.head is None:
25             print("List is empty")
26         else:
27             temp = self.head
28             while temp:
29                 print(temp.data, end=" -> ")
30                 temp = temp.next
31             print("None")      # indicate end of list
32
33 ll = SinglyLinkedList()
34 ll.insert(10)
35 ll.insert(20)
36 ll.insert(30)
37
38 ll.display()    # Output: 10 -> 20 -> 30 -> None
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai_ ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai_ ass/task3.py"
10 -> 20 -> 30 -> None
PS C:\Users\ASHMITHA\Desktop\ai_ ass>
```

Observation:

The program creates a singly linked list where each node contains some data and a pointer to the next node. Using the insert() method, new nodes (10, 20, and 30) are successfully added to the end of the list. The display() method then traverses the list starting from the head and prints the nodes in sequence as 10 -> 20 -> 30 -> None, which shows the correct linkage between nodes and confirms that insertion and traversal are working properly.

Task4:

Prompt:

Create a BST with insert and in-order traversal methods.

Code:

task4.py > ...

```
1  class Node:
2      def __init__(self, data):
3          self.data = data
4          self.left = None
5          self.right = None
6  class BST:
7      def __init__(self):
8          self.root = None
9      def insert(self, data):
10         """Insert a new node into the BST"""
11         if self.root is None:
12             self.root = Node(data)
13         else:
14             self._insert(self.root, data)
15
16     def _insert(self, current, data):
17         if data < current.data: # go left
18             if current.left is None:
19                 current.left = Node(data)
20             else:
21                 self._insert(current.left, data)
22         elif data > current.data: # go right
23             if current.right is None:
24                 current.right = Node(data)
25             else:
26                 self._insert(current.right, data)
27         # if data == current.data, do nothing (no duplicates)
28
29     def inorder(self):
30         """Perform in-order traversal of the BST"""
31         return self._inorder(self.root)
32
33     def _inorder(self, node):
34         result = []
35         if node:
36             result += self._inorder(node.left) # left
37             result.append(node.data)           # root
```

```

32
33     def _inorder(self, node):
34         result = []
35         if node:
36             result += self._inorder(node.left)    # left
37             result.append(node.data)              # root
38             result += self._inorder(node.right)  # right
39         return result
40 bst = BST()
41 bst.insert(50)
42 bst.insert(30)
43 bst.insert(70)
44 bst.insert(20)
45 bst.insert(40)
46 bst.insert(60)
47 bst.insert(80)
48 print("In-order Traversal:", bst.inorder())
49

```

Output:

```

> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai_ ass/task4.py"
In-order Traversal: [20, 30, 40, 50, 60, 70, 80]
PS C:\Users\ASHMITHA\Desktop\ai_ ass>

```

Observation:

The program constructs a Binary Search Tree (BST) by inserting nodes such that smaller values are placed in the left subtree and larger values in the right subtree. After inserting the nodes 50, 30, 70, 20, 40, 60, 80, the BST structure is correctly formed. The `inorder()` traversal visits nodes in the order **Left → Root → Right**, which produces the output [20, 30, 40, 50, 60, 70, 80]. This result is sorted, confirming that the BST insertion and in-order traversal methods work as expected.

Task5:

Prompt:

Implement a hash table with basic insert, search, and delete methods.

Code:

```
1  class HashTable:
2      def __init__(self, size=10):
3          self.size = size
4          self.table = [[] for _ in range(size)] # list of buckets
5
6      def _hash(self, key):
7          """Simple hash function"""
8          return hash(key) % self.size
9
10     def put(self, key, value):
11         """Insert or update a key-value pair"""
12         index = self._hash(key)
13         bucket = self.table[index]
14
15         # check if key already exists -> update
16         for i, (k, v) in enumerate(bucket):
17             if k == key:
18                 bucket[i] = (key, value)
19                 return
20         # otherwise insert new pair
21         bucket.append((key, value))
22
23     def get(self, key):
24         """Search for a key and return its value"""
25         index = self._hash(key)
26         bucket = self.table[index]
27
28         for k, v in bucket:
29             if k == key:
30                 return v
31         return None # key not found
32
33     def remove(self, key):
34         """Delete a key-value pair"""
35         index = self._hash(key)
36         bucket = self.table[index]
37
```

```

1  class HashTable:
2
33     def remove(self, key):
34         """Delete a key-value pair"""
35         index = self._hash(key)
36         bucket = self.table[index]
37
38         for i, (k, v) in enumerate(bucket):
39             if k == key:
40                 del bucket[i]
41                 return True
42         return False # key not found
43
44     def display(self):
45         """Show the hash table contents"""
46         for i, bucket in enumerate(self.table):
47             print(f"Bucket {i}: {bucket}")
48
49
50 # Example usage
51 ht = HashTable()
52
53 # Insert
54 ht.put("apple", 10)
55 ht.put("banana", 20)
56 ht.put("grape", 30)
57
58 # Display
59 ht.display()
60
61 # Search
62 print("Search 'banana':", ht.get("banana"))
63
64 # Delete
65 ht.remove("apple")
66 ht.display()
67

```

Output:

```

PS C:\Users\ASHMITHA\Desktop\ai ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai ass/task5.py"
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: []
Bucket 4: [('banana', 20)]
Bucket 5: []
Bucket 6: []
Bucket 7: [('apple', 10)]
Bucket 8: [('grape', 30)]
Bucket 9: []
Search 'banana': 20
Bucket 0: []
Bucket 1: []
Bucket 2: []
Bucket 3: []
Bucket 4: [('banana', 20)]
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: [('grape', 30)]
Bucket 9: []
PS C:\Users\ASHMITHA\Desktop\ai ass>

```

Observation:

The program successfully implements a hash table using lists with chaining to handle collisions. Keys are mapped to specific buckets using a hash function ($\text{hash}(\text{key}) \% \text{size}$). When inserting, key–value pairs such as ("apple", 10), ("banana", 20), and ("grape", 30) are placed in their respective buckets. The `get()` method retrieves values correctly (e.g., searching for "banana" returns 20). The `remove()` method deletes a key–value pair (e.g., "apple" is removed, leaving only "banana" and "grape"). The `display()` method confirms the distribution of elements across buckets. Thus, insertion, search, and deletion operations work as expected in the hash table.

Task6:

Prompt:

Implement a graph using an adjacency list.

Code:

```

task6.py > ...
1  class Graph:
2      def __init__(self):
3          self.adj_list = {}  # dictionary to hold adjacency list
4
5      def add_vertex(self, vertex):
6          """Add a new vertex to the graph"""
7          if vertex not in self.adj_list:
8              self.adj_list[vertex] = []
9
10     def add_edge(self, v1, v2):
11         """Add an undirected edge between v1 and v2"""
12         if v1 in self.adj_list and v2 in self.adj_list:
13             self.adj_list[v1].append(v2)
14             self.adj_list[v2].append(v1)  # remove this line for directed graph
15
16     def display(self):
17         """Display the adjacency list of the graph"""
18         for vertex in self.adj_list:
19             print(vertex, "->", self.adj_list[vertex])
20
21 g = Graph()
22 # Add vertices
23 g.add_vertex("A")
24 g.add_vertex("B")
25 g.add_vertex("C")
26 g.add_vertex("D")
27 # Add edges
28 g.add_edge("A", "B")
29 g.add_edge("A", "C")
30 g.add_edge("B", "D")
31 g.add_edge("C", "D")
32 # Display graph
33 g.display()

```

Output:

```

PS C:\Users\ASHMITHA\Desktop\ai assss> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai assss/task6.py"
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A', 'D']
D -> ['B', 'C']
PS C:\Users\ASHMITHA\Desktop\ai assss>

```

Observation:

The program successfully implements a hash table using lists with chaining to handle collisions. Keys are mapped to specific buckets using a hash function ($\text{hash}(\text{key}) \% \text{size}$). When inserting, key–value pairs such as ("apple", 10), ("banana", 20), and ("grape", 30) are placed in their respective buckets. The `get()` method retrieves values correctly (e.g., searching for "banana" returns 20). The `remove()` method deletes a key–value pair (e.g., "apple" is removed, leaving only "banana" and "grape"). The `display()` method confirms the

distribution of elements across buckets. Thus, insertion, search, and deletion operations work as expected in the hash table.

Task7:

Prompt:

Implement a priority queue using Python's heapq module.

Code:

```
task7.py 7 ...
3 class PriorityQueue:
6
7     def push(self, priority, item):
8         """Insert an item with a given priority"""
9         heapq.heappush(self.heap, (priority, item))
10
11    def pop(self):
12        """Remove and return the item with the smallest priority"""
13        if self.is_empty():
14            return "Priority Queue is empty"
15        return heapq.heappop(self.heap)[1] # return only the item
16
17    def peek(self):
18        """Return the item with the smallest priority without removing it"""
19        if self.is_empty():
20            return "Priority Queue is empty"
21        return self.heap[0][1]
22
23    def is_empty(self):
24        """Check if the priority queue is empty"""
25        return len(self.heap) == 0
26
27    def display(self):
28        """Display the internal heap (priority, item) pairs"""
29        print(self.heap)
30
31    # Example usage
32    pq = PriorityQueue()
33    pq.push(2, "Task B")
34    pq.push(1, "Task A")
35    pq.push(3, "Task C")
36    pq.display()           # [(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]
37    print(pq.peek())      # Task A (highest priority = lowest number)
38    print(pq.pop())       # Task A
39    print(pq.pop())       # Task B
40    print(pq.is_empty())  # False
41    print(pq.pop())       # Task C
42    print(pq.is_empty())  # True
```

```

30 # Example usage
31 pq = PriorityQueue()
32 pq.push(2, "Task B")
33 pq.push(1, "Task A")
34 pq.push(3, "Task C")
35 pq.display()           # [(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]
36 print(pq.peek())      # Task A (highest priority = lowest number)
37 print(pq.pop())       # Task A
38 print(pq.pop())       # Task B
39 print(pq.is_empty())  # False
40 print(pq.pop())       # Task C
41 print(pq.is_empty())  # True
42

```

Output:

```

PS C:\Users\ASHMITHA\Desktop\ai ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai ass/task7.py"
[(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]
Task A
Task A
Task B
False
Task C
True

```

Observation:

The program implements a priority queue using Python's `heapq` module, which maintains elements in a min-heap structure. Each item is stored as a (priority, value) pair, where the element with the **lowest priority number** is always served first. When inserting tasks with priorities 2 (Task B), 1 (Task A), and 3 (Task C), the heap is internally arranged as [(1, 'Task A'), (2, 'Task B'), (3, 'Task C')]. The `peek()` method shows "Task A" as the highest-priority element without removing it. Successive `pop()` operations return "Task A", "Task B", and "Task C" in the correct priority order. The `is_empty()` method accurately detects when the queue becomes empty. This confirms that insertion, retrieval, and deletion all work as expected in the priority queue.

Task8:

Prompt:

Implement a double-ended queue using `collections.deque`.

Code:

task8.py > ...

```
1  from collections import deque
2
3  class DequeExample:
4      def __init__(self):
5          self.deque = deque()
6
7      def add_front(self, item):
8          """Insert an item at the front of the deque"""
9          self.deque.appendleft(item)
10
11     def add_rear(self, item):
12         """Insert an item at the rear of the deque"""
13         self.deque.append(item)
14
15     def remove_front(self):
16         """Remove and return the front item"""
17         if self.is_empty():
18             return "Deque is empty"
19         return self.deque.popleft()
20
21     def remove_rear(self):
22         """Remove and return the rear item"""
23         if self.is_empty():
24             return "Deque is empty"
25         return self.deque.pop()
26
27     def peek_front(self):
28         """View the front item without removing it"""
29         if self.is_empty():
30             return "Deque is empty"
31         return self.deque[0]
32
33     def peek_rear(self):
34         """View the rear item without removing it"""
35         if self.is_empty():
36             return "Deque is empty"
37         return self.deque[-1]
```

```

42
43     def display(self):
44         """Display the current deque"""
45         print(list(self.deque))
46
47
48     # Example usage
49     dq = DequeExample()
50
51     dq.add_rear(10)
52     dq.add_rear(20)
53     dq.add_front(5)
54     dq.display()           # [5, 10, 20]
55
56     print(dq.remove_front()) # 5
57     print(dq.remove_rear())  # 20
58     dq.display()           # [10]
59
60     print(dq.peek_front())   # 10
61     print(dq.peek_rear())    # 10
62     print(dq.is_empty())     # False
63

```

Output:

```

PS C:\Users\ASHMITHA\Desktop\ai ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai ass/task8.py"
[5, 10, 20]
5
20
[10]
10
10
False

```

Observation:

The program uses Python's `collections.deque` to implement a double-ended queue that allows insertion and deletion of elements from both the front and the rear in constant time. When elements 10 and 20 are added at the rear and 5 at the front, the deque becomes [5, 10, 20]. The `remove_front()` operation deletes 5, and the `remove_rear()` operation deletes 20, leaving [10] in the deque. Both `peek_front()` and `peek_rear()` correctly show 10 as the only

remaining element. The `is_empty()` method returns `False`, confirming that the deque still contains elements. This demonstrates that the deque supports efficient double-ended operations as expected.

Task9:

Prompt:

generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

data structure comparison table:

Data Structure	Insertion	Deletion	Access (by index)	Search	Remarks
Stack (LIFO)	$O(1)$ (push)	$O(1)$ (pop)	$O(n)$	$O(n)$	Access is only from top; used for undo/recursion
Queue (FIFO)	$O(1)$ (enqueue)	$O(1)$ (dequeue)	$O(n)$	$O(n)$	Access only front/rear; used in scheduling
Deque (Double-ended queue)	$O(1)$ (front/rear)	$O(1)$ (front/rear)	$O(n)$	$O(n)$	Supports both stack & queue operations
Singly Linked List	$O(1)$ (front), $O(n)$ (end)	$O(1)$ (front), $O(n)$ (end)	$O(n)$	$O(n)$	Dynamic size, sequential access
Doubly Linked List	$O(1)$ (front/end)	$O(1)$ (front/end)	$O(n)$	$O(n)$	Can traverse in both directions
Binary Search Tree (BST)	$O(\log n)$ (avg), $O(n)$ (worst)	$O(\log n)$ (avg), $O(n)$ (worst)	$O(\log n)$ (avg), $O(n)$ (worst)	$O(\log n)$ (avg), $O(n)$ (worst)	Efficient if balanced, but can degrade
Balanced BST (AVL, Red-Black)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	Always balanced for efficiency

Data Structure	Insertion	Deletion	Access (by index)	Search	Remarks
Hash Table	$O(1)$ (avg), $O(n)$ (worst with collisions)	$O(1)$ (avg)	N/A	$O(1)$ (avg)	Fast lookups, may require rehashing
Array	$O(1)$ (end), $O(n)$ (middle)	$O(1)$ (end), $O(n)$ (middle)	$O(1)$	$O(n)$ (linear), $O(\log n)$ if sorted + binary search	Fixed size unless dynamic
Graph (Adjacency List)	$O(1)$ (edge insert)	$O(1)$ (edge remove, avg)	$O(V)$	$O(V+E)$ (traversal)	Efficient for sparse graphs

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.
- **Arrays** provide constant-time access by index but have slower insertions/deletions in the middle ($O(n)$), although appending at the end is $O(1)$.

- **Graphs (Adjacency List)** efficiently store sparse graphs, allowing constant-time edge insertions and traversal in $O(V+E)$.

Task10:

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. The program should have methods to:

1. Add a student order.
2. Serve the next student in line.
3. Peek at the next student without removing them.
4. Check if the queue is empty.
5. Display all current orders.

Include docstrings for each method and the class. Also, provide at least 3 assert test cases to verify that the queue operations (enqueue, dequeue, peek) work correctly.

Code:

task10.py > ...

```
1 class CafeteriaQueue:
2     """
3     A class to simulate a cafeteria order queue using FIFO principle.
4     """
5
6     def __init__(self):
7         self.queue = []
8
9     def add_order(self, student_name):
10        """
11        Add a student's order to the end of the queue.
12        :param student_name: str
13        """
14        self.queue.append(student_name)
15
16    def serve_order(self):
17        """
18        Serve the next order in the queue (FIFO).
19        :return: str
20        """
21        if self.is_empty():
22            return "No orders to serve."
23        return self.queue.pop(0)
24
25    def peek_next(self):
26        """
27        Peek at the next order without removing it.
28        :return: str
29        """
30        if self.is_empty():
31            return "No orders in the queue."
32        return self.queue[0]
33
```

```
task10.py > ...
1  class CafeteriaQueue:
33
34      def is_empty(self):
35          """
36          Check if the queue is empty.
37          :return: bool
38          """
39          return len(self.queue) == 0
40
41      def display_orders(self):
42          """
43          Display all current orders in the queue.
44          """
45          return list(self.queue)
46
47
48      # Example Usage
49      if __name__ == "__main__":
50          cafe_queue = CafeteriaQueue()
51
52          # Adding orders
53          cafe_queue.add_order("Alice")
54          cafe_queue.add_order("Bob")
55          cafe_queue.add_order("Charlie")
56
57          print("Current Orders:", cafe_queue.display_orders())
58          print("Next to Serve:", cafe_queue.peek_next())
59          print("Serving Order:", cafe_queue.serve_order())
60          print("Orders After Serving:", cafe_queue.display_orders())
61
62          # Assert Test Cases
63          assert cafe_queue.peek_next() == "Bob"
64          assert cafe_queue.serve_order() == "Bob"
65          assert cafe_queue.serve_order() == "Charlie"
66          assert cafe_queue.is_empty() == True
67
```

Output:

```
PS C:\Users\ASHMITHA\Desktop\ai ass> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/ASHMITHA/Desktop/ai ass/task10.py"
Current Orders: ['Alice', 'Bob', 'Charlie']
Next to Serve: Alice
Serving Order: Alice
Orders After Serving: ['Bob', 'Charlie']
PS C:\Users\ASHMITHA\Desktop\ai ass> |
```

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. The assert test cases validate that enqueue, dequeue, and peek operations function as expected, demonstrating that the queue reliably maintains the correct order of service.