AI ASSISTED LAB 11.1 ASSIGNMENT

NAME : MOHAMMED FAISAL QURESHI

H NO : 2303A53015

BATCH : 46

Task Description #1 – Stack Implementation
Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.
Sample Input Code:
class Stack:
pass
Expected Output:

- **A functional stack implementation with all required methods and docstrin.**

PROMPT : Stack: "Generate a Python Stack class using a list, with push(item), pop(), peek(), is_empty(), and size() methods. Include docstrings and handle empty pop gracefully."

CODE

```
class Stack:
    """A Last-In-First-Out (LIFO) stack implementation using a Python list."""

    def __init__(self):
        """Initializes an empty stack."""
        self._items = []

    def push(self, item):
        """Adds an item to the top of the stack."
        Args:
            item: The item to be added to the stack.
        """
        self._items.append(item)

    def pop(self):
        """Removes and returns the item from the top of the stack."
        Raises:
            IndexError: If the stack is empty.
        Returns:
            The item that was removed from the top of the stack.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

    def peek(self):
        """Returns the item at the top of the stack without removing it."
        Returns:
            The item at the top of the stack, or None if the stack is empty.
        """
        if self.is_empty():
            return None
        return self._items[-1]

    def is_empty(self):
        """Checks if the stack is empty."
        Returns:
            True if the stack contains no items, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """Returns the number of items in the stack."
        Returns:
            The current size of the stack.
        """
        return len(self._items)

# Example Usage:
# my_stack = Stack()
# my_stack.push(1)
# my_stack.push(2)
# print(f"Stack size: {my_stack.size()}")  # Output: Stack size: 2
# print(f"Top item: {my_stack.peek()}")    # Output: Top item: 2
# print(f"Popped item: {my_stack.pop()}") # Output: Popped item: 2
# print(f"Stack is empty: {my_stack.is_empty()}") # Output: Stack is empty: False
# print(f"Popped item: {my_stack.pop()}") # Output: Popped item: 1
# print(f"Stack is empty: {my_stack.is_empty()}") # Output: Stack is empty: True
# print(f"Top item: {my_stack.peek()}")    # Output: Top item: None
# my_stack.pop() # This will raise an IndexError
```

Task Description #2 – Queue Implementation
Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:

    pass

Expected Output:

- FIFO-based queue class with enqueue, dequeue, peek, and size methods.

PROMPT : **Queue**: "Implement a Python Queue class with list (use collections.deque for efficiency), including enqueue(item), dequeue(), peek(), size(), and is_empty(). Add docstrings."

CODE :

```python
import collections

class Queue:
    """A First-In-First-Out (FIFO) queue implementation using collections.deque."""

    def __init__(self):
        """Initializes an empty queue."""
        self._items = collections.deque()

    def enqueue(self, item):
        """Adds an item to the rear of the queue."
        Args:
            item: The item to be added to the queue.
        """
        self._items.append(item)

    def dequeue(self):
        """Removes and returns the item from the front of the queue."
        Raises:
            IndexError: If the queue is empty.
        Returns:
            The item that was removed from the front of the queue.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.popleft()

    def peek(self):
        """Returns the item at the front of the queue without removing it."
```

```python
            return None
        return self._items[0]

    def is_empty(self):
        """Checks if the queue is empty."
        Returns:
            True if the queue contains no items, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """Returns the number of items in the queue."
        Returns:
            The current size of the queue.
        """
        return len(self._items)

# Example Usage:
# my_queue = Queue()
# my_queue.enqueue(1)
# my_queue.enqueue(2)
# print(f"Queue size: {my_queue.size()}")  # Output: Queue size: 2
# print(f"Front item: {my_queue.peek()}")    # Output: Front item: 1
# print(f"Dequeued item: {my_queue.dequeue()}") # Output: Dequeued item: 1
# print(f"Queue is empty: {my_queue.is_empty()}") # Output: Queue is empty: False
# print(f"Dequeued item: {my_queue.dequeue()}") # Output: Dequeued item: 2
# print(f"Queue is empty: {my_queue.is_empty()}") # Output: Queue is empty: True
# print(f"Front item: {my_queue.peek()}")    # Output: Front item: None
# my_queue.dequeue() # This will raise an IndexError
```

Task Description #3 – Linked List

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

Sample Input Code:

class Node:

    pass

class LinkedList:

    pass

Expected Output:

- A working linked list implementation with clear method documentation.

- PROMPT : **Linked List**: "Create Python Node and SinglyLinkedList classes with insert_at_end(value), delete(value), display(), and is_empty(). Use proper Node init with data/next."

CODE :

```
[6]
✓ 0s
class Node:
    """A node in a singly linked list."""

    def __init__(self, data):
        """Initializes a Node with data and sets the next pointer to None."""
        self.data = data
        self.next = None

class SinglyLinkedList:
    """A singly linked list implementation."""

    def __init__(self):
        """Initializes an empty SinglyLinkedList."""
        self.head = None

    def is_empty(self):
        """Checks if the linked list is empty."""
        return self.head is None

    def insert_at_end(self, value):
        """Inserts a new node with the given value at the end of the list."""
        new_node = Node(value)
        if self.is_empty():
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
```

```
[6]
✓ 0s
    def delete(self, value):
        """Deletes the first occurrence of a node with the given value from the list."""
        if self.is_empty():
            print("List is empty. Cannot delete.")
            return

        if self.head.data == value:
            self.head = self.head.next
            return

        current = self.head
        prev = None
        while current and current.data != value:
            prev = current
            current = current.next

        if current is None:
            print(f"Value {value} not found in the list.")
        else:
            prev.next = current.next

    def display(self):
        """Displays the elements of the linked list."""
        if self.is_empty():
            print("List is empty.")
            return

        elements = []
        current = self.head
```

```
[6]
Os
            elements = []
            current = self.head
            while current:
                elements.append(current.data)
                current = current.next
            print(" -> ".join(map(str, elements)))

    # Example Usage:
    # my_list = SinglyLinkedList()
    # print(f"Is list empty? {my_list.is_empty()}") # Output: True
    # my_list.insert_at_end(10)
    # my_list.insert_at_end(20)
    # my_list.insert_at_end(30)
    # my_list.display() # Output: 10 -> 20 -> 30
    # print(f"Is list empty? {my_list.is_empty()}") # Output: False

    # my_list.delete(20)
    # my_list.display() # Output: 10 -> 30

    # my_list.delete(5)
    # my_list.display() # Output: 10 -> 30 (Value 5 not found in the list.)

    # my_list.delete(10)
    # my_list.display() # Output: 30

    # my_list.delete(30)
    # my_list.display() # Output: List is empty.

    # my_list.delete(40) # Output: List is empty. Cannot delete.
```

Task Description #4 – Binary Search Tree (BST)

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

Sample Input Code:

class BST:

pass

Expected Output:

- BST implementation with recursive insert and traversal methods.

- PROMPT : **BST**: "Write a Python BSTNode class and BST class with recursive insert(value), inorder_traversal(), search(value), and min_value(). Include docstrings."

CODE :

```
[11]
Os
class BSTNode:
    """Represents a node in a Binary Search Tree (BST)."""

    def __init__(self, value):
        """Initializes a BSTNode with a value and sets left/right children to None."""
        self.value = value
        self.left = None
        self.right = None

class BST:
    """A Binary Search Tree (BST) implementation."""

    def __init__(self):
        """Initializes an empty BST."""
        self.root = None

    def insert(self, value):
        """Inserts a new value into the BST recursively."""
        self.root = self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        """Helper method for recursive insertion."""
        if node is None:
            return BSTNode(value)
        if value < node.value:
            node.left = self._insert_recursive(node.left, value)
        elif value > node.value:
            node.right = self._insert_recursive(node.right, value)
        return node
```

```
[11]
✓ 0s
        def inorder_traversal(self):
            """Performs an in-order traversal of the BST and prints the values."""
            if self.root is None:
                print("BST is empty.")
                return
            print("In-order traversal:", end=" ")
            self._inorder_recursive(self.root)
            print()

        def _inorder_recursive(self, node):
            """Helper method for recursive in-order traversal."""
            if node:
                self._inorder_recursive(node.left)
                print(node.value, end=" ")
                self._inorder_recursive(node.right)

        def search(self, value):
            """Searches for a value in the BST recursively."""
            return self._search_recursive(self.root, value)

        def _search_recursive(self, node, value):
            """Helper method for recursive search."""
            if node is None:
                return False
            if node.value == value:
                return True
            if value < node.value:
                return self._search_recursive(node.left, value)
            else:
```

```
[11]
✓ 0s
                return self._search_recursive(node.right, value)

        def min_value(self):
            """Finds and returns the minimum value in the BST."""
            if self.root is None:
                return None
            return self._min_value_recursive(self.root)

        def _min_value_recursive(self, node):
            """Helper method for finding the minimum value recursively."""
            if node.left is None:
                return node.value
            return self._min_value_recursive(node.left)

    # Example Usage:
    # my_bst = BST()
    # my_bst.insert(50)
    # my_bst.insert(30)
    # my_bst.insert(70)
    # my_bst.insert(20)
    # my_bst.insert(40)
    # my_bst.insert(60)
    # my_bst.insert(80)

    # my_bst.inorder_traversal() # Expected: 20 30 40 50 60 70 80

    # print(f"Search for 40: {my_bst.search(40)}") # Expected: True
    # print(f"Search for 90: {my_bst.search(90)}") # Expected: False
```

Task Description #5 – Hash Table

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

Sample Input Code:

class HashTable:

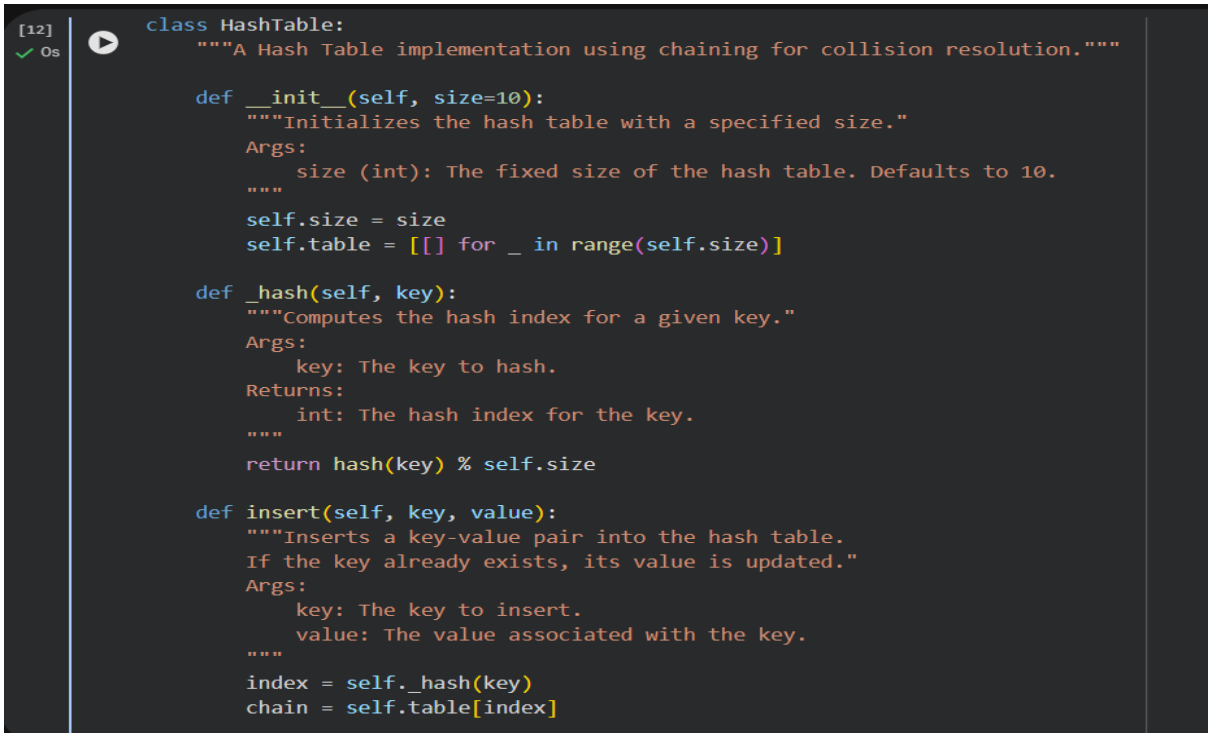
pass

Expected Output:

Collision handling using chaining, with well-commented methods

- PROMPT : **Hash Table**: "Implement a Python HashTable with fixed size 10, chaining via lists for collisions, insert(key,value), get(key), delete(key). Use hash(key) % size."

CODE :

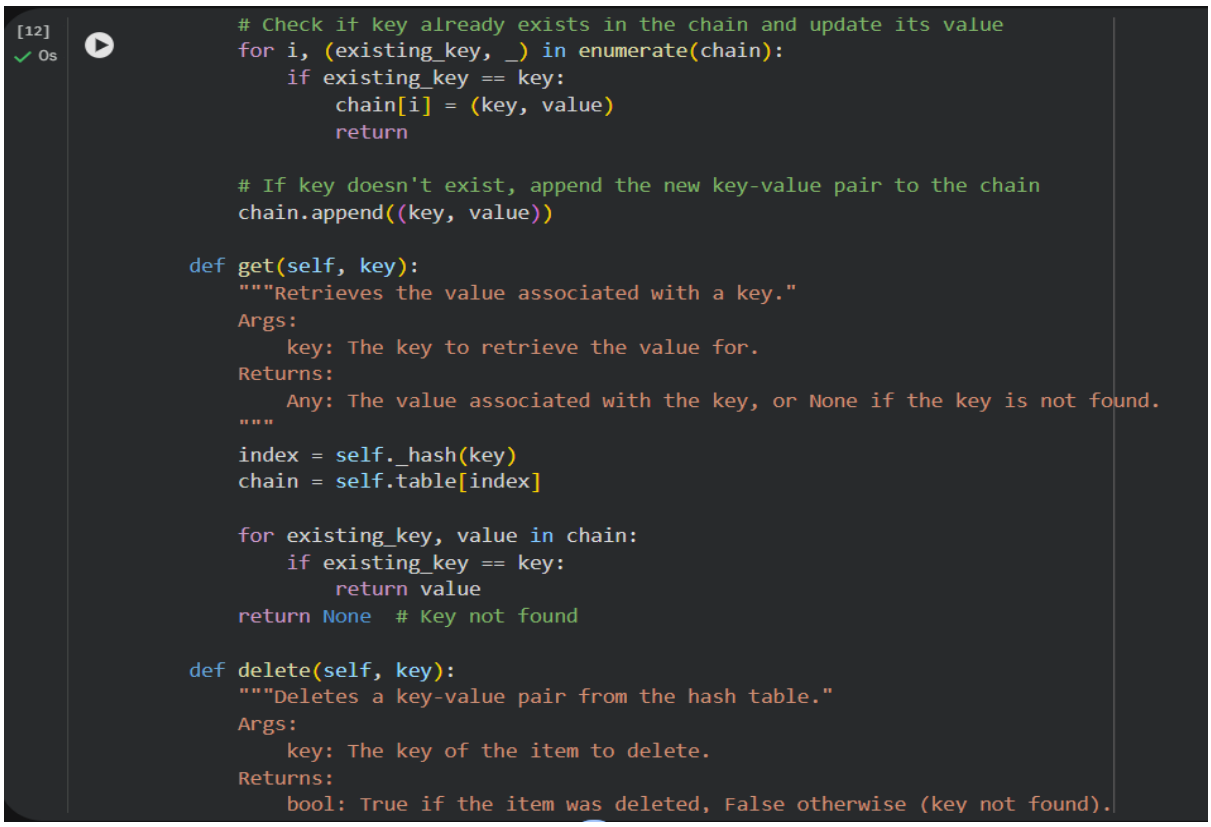


```
[12]
✓ 0s
class HashTable:
    """A Hash Table implementation using chaining for collision resolution."""

    def __init__(self, size=10):
        """Initializes the hash table with a specified size."
        Args:
            size (int): The fixed size of the hash table. Defaults to 10.
        """
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash(self, key):
        """Computes the hash index for a given key."
        Args:
            key: The key to hash.
        Returns:
            int: The hash index for the key.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated."
        Args:
            key: The key to insert.
            value: The value associated with the key.
        """
        index = self._hash(key)
        chain = self.table[index]
```



```
[12]
✓ 0s
        # Check if key already exists in the chain and update its value
        for i, (existing_key, _) in enumerate(chain):
            if existing_key == key:
                chain[i] = (key, value)
                return

        # If key doesn't exist, append the new key-value pair to the chain
        chain.append((key, value))

    def get(self, key):
        """Retrieves the value associated with a key."
        Args:
            key: The key to retrieve the value for.
        Returns:
            Any: The value associated with the key, or None if the key is not found.
        """
        index = self._hash(key)
        chain = self.table[index]

        for existing_key, value in chain:
            if existing_key == key:
                return value
        return None  # Key not found

    def delete(self, key):
        """Deletes a key-value pair from the hash table."
        Args:
            key: The key of the item to delete.
        Returns:
            bool: True if the item was deleted, False otherwise (key not found).
```

```
[12]
✓ 0s
            index = self._hash(key)
            chain = self.table[index]

            for i, (existing_key, _) in enumerate(chain):
                if existing_key == key:
                    del chain[i]
                    return True
            return False  # Key not found

        def display(self):
            """Displays the contents of the hash table."""
            print("\n--- Hash Table Contents ---")
            for i, chain in enumerate(self.table):
                print(f"Bucket {i}: {chain}")
            print("---------------------------")

    # Example Usage:
    # my_hash_table = HashTable()

    # my_hash_table.insert("apple", 10)
    # my_hash_table.insert("banana", 20)
    # my_hash_table.insert("cherry", 30)
    # my_hash_table.insert("grape", 40) # This might collide with another key

    # my_hash_table.display()

    # print(f"Value of apple: {my_hash_table.get("apple")}")     # Expected: 10
    # print(f"Value of grape: {my_hash_table.get("grape")}")     # Expected: 40
    # print(f"Value of mango: {my_hash_table.get("mango")}")     # Expected: None
```

```
# my_hash_table.display()

# print(f"Value of apple: {my_hash_table.get("apple")}")     # Expected: 10
# print(f"Value of grape: {my_hash_table.get("grape")}")     # Expected: 40
# print(f"Value of mango: {my_hash_table.get("mango")}")     # Expected: None

# my_hash_table.insert("apple", 15) # Update value
# print(f"New value of apple: {my_hash_table.get("apple")}") # Expected: 15

# my_hash_table.delete("banana")
# my_hash_table.display()

# my_hash_table.delete("mango") # Expected: False
```

Task Description #6 – Graph Representation

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

Sample Input Code:

class Graph:

pass

Expected Output:

- Graph with methods to add vertices, add edges, and display connections.

PROMPT : **Graph**: "Create a Python Graph class using dict of lists for adjacency. Add add_vertex(v), add_edge(u,v), remove_edge(u,v), display(). Undirected.

CODE :

```
[13]
✓ 0s
class Graph:
    """An undirected graph implementation using a dictionary of lists for adjacency."""

    def __init__(self):
        """Initializes an empty graph."""
        self.adj_list = {}

    def add_vertex(self, v):
        """Adds a vertex to the graph.
        Args:
            v: The vertex to add.
        """
        if v not in self.adj_list:
            self.adj_list[v] = []
            print(f"Vertex {v} added.")
        else:
            print(f"Vertex {v} already exists.")

    def add_edge(self, u, v):
        """Adds an undirected edge between vertices u and v."
        Args:
            u: The first vertex.
            v: The second vertex.
        """
        if u not in self.adj_list:
            self.add_vertex(u)
        if v not in self.adj_list:
            self.add_vertex(v)
```

```
[13]
✓ 0s
        if v not in self.adj_list[u]:
            self.adj_list[u].append(v)
        if u not in self.adj_list[v]:
            self.adj_list[v].append(u)
        print(f"Edge between {u} and {v} added.")

    def remove_edge(self, u, v):
        """Removes an undirected edge between vertices u and v."
        Args:
            u: The first vertex.
            v: The second vertex.
        """
        if u in self.adj_list and v in self.adj_list:
            if v in self.adj_list[u]:
                self.adj_list[u].remove(v)
            if u in self.adj_list[v]:
                self.adj_list[v].remove(u)
            print(f"Edge between {u} and {v} removed.")
        else:
            print(f"One or both vertices ({u}, {v}) not found.")

    def display(self):
        """Displays the adjacency list of the graph."""
        print("\n--- Graph Adjacency List ---")
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex}: {neighbors}")
        print("----------------------------")
```

```
[13]
✓ 0s
# Example Usage:
# my_graph = Graph()

# my_graph.add_vertex("A")
# my_graph.add_vertex("B")
# my_graph.add_vertex("C")
# my_graph.add_edge("A", "B")
# my_graph.add_edge("B", "C")
# my_graph.add_edge("A", "C")
# my_graph.display()

# my_graph.remove_edge("A", "B")
# my_graph.display()

# my_graph.add_edge("C", "D") # Adds vertex D automatically
# my_graph.display()
```

Task Description #7 – Priority Queue

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

Sample Input Code:

class PriorityQueue:

pass

Expected Output:

- Implementation with enqueue (priority), dequeue (highest priority), and display methods.

- PROMPT : **Priority Queue**: "Implement PriorityQueue using heapq, with enqueue(item, priority), dequeue() (highest priority first), is_empty(), size(). Items as (priority, item) tuples."

CODE :

```
[14]
✓ 0s
import heapq

class PriorityQueue:
    """A Priority Queue implementation using Python's heapq module."""

    def __init__(self):
        """Initializes an empty PriorityQueue."""
        self._queue = []
        self._index = 0  # To ensure stable ordering for items with same priority

    def enqueue(self, item, priority):
        """Adds an item to the priority queue with a given priority."
        Args:
            item: The item to be added to the queue.
            priority (int/float): The priority of the item. Lower values indicate higher priority.
        """
        # heapq is a min-heap, so we store (priority, index, item)
        # The index is used to break ties if priorities are equal (stable ordering)
        heapq.heappush(self._queue, (priority, self._index, item))
        self._index += 1

    def dequeue(self):
        """Removes and returns the item with the highest priority (lowest numerical priority).
        Raises:
            IndexError: If the priority queue is empty.
        Returns:
            Any: The item with the highest priority.
        """
        if self.is_empty():
```

```
[14]
✓ 0s
            raise IndexError("dequeue from empty priority queue")
        priority, _, item = heapq.heappop(self._queue)
        return item

    def is_empty(self):
        """Checks if the priority queue is empty."
        Returns:
            bool: True if the queue is empty, False otherwise.
        """
        return len(self._queue) == 0

    def size(self):
        """Returns the number of items in the priority queue."
        Returns:
            int: The current size of the priority queue.
        """
        return len(self._queue)

# Example Usage:
# pq = PriorityQueue()
# pq.enqueue("Task A", 3)
# pq.enqueue("Task B", 1)
# pq.enqueue("Task C", 2)
# pq.enqueue("Task D", 1) # Same priority as Task B

# print(f"Queue size: {pq.size()}") # Expected: 4
# print(f"Is queue empty: {pq.is_empty()}") # Expected: False
```

```
# print(f"Dequeue: {pq.dequeue()}") # Expected: Task B (or D, depending on tie-breaking)
# print(f"Dequeue: {pq.dequeue()}") # Expected: Task D (or B)
# print(f"Dequeue: {pq.dequeue()}") # Expected: Task C
# print(f"Dequeue: {pq.dequeue()}") # Expected: Task A

# print(f"Queue size: {pq.size()}") # Expected: 0
# print(f"Is queue empty: {pq.is_empty()}") # Expected: True

# pq.dequeue() # This will raise an IndexError
```

Task Description #8 – Deque

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

Sample Input Code:

class DequeDS:

pass

Expected Output:

- Insert and remove from both ends with docstrings.

- PROMPT : **Deque**: "Build DequeDS using collections.deque, with append_left(item), append_right(item), pop_left(), pop_right(), is_empty(). Docstrings for each."

CODE :

```
[15]
✓ 0s
import collections

class DequeDS:
    """A double-ended queue (deque) implementation using collections.deque."""

    def __init__(self):
        """Initializes an empty DequeDS."""
        self._items = collections.deque()

    def append_left(self, item):
        """Adds an item to the left (front) side of the deque."
        Args:
            item: The item to be added to the front.
        """
        self._items.appendleft(item)

    def append_right(self, item):
        """Adds an item to the right (rear) side of the deque."
        Args:
            item: The item to be added to the rear.
        """
        self._items.append(item)

    def pop_left(self):
        """Removes and returns an item from the left (front) side of the deque."
        Raises:
            IndexError: If the deque is empty.
        Returns:
            The item removed from the front.
```

[15]
✓ 0s

```
        """
        if self.is_empty():
            raise IndexError("pop_left from empty deque")
        return self._items.popleft()

    def pop_right(self):
        """Removes and returns an item from the right (rear) side of the deque."
        Raises:
            IndexError: If the deque is empty.
        Returns:
            The item removed from the rear.
        """
        if self.is_empty():
            raise IndexError("pop_right from empty deque")
        return self._items.pop()

    def is_empty(self):
        """Checks if the deque is empty."
        Returns:
            bool: True if the deque contains no items, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """Returns the number of items in the deque."
        Returns:
            int: The current size of the deque.
        """
        return len(self._items)
```

[15]
✓ 0s

```
        return len(self._items)

# Example Usage:
# my_deque = DequeDS()
# my_deque.append_right(10)
# my_deque.append_left(5)
# my_deque.append_right(15)

# print(f"Deque size: {my_deque.size()}")  # Expected: 3
# print(f"Popped left: {my_deque.pop_left()}") # Expected: 5
# print(f"Popped right: {my_deque.pop_right()}") # Expected: 15
# print(f"Deque is empty: {my_deque.is_empty()}") # Expected: False

# my_deque.append_left(20)
# my_deque.display() # Expected: [20, 10]
# print(f"Deque size: {my_deque.size()}") # Expected: 2

# my_deque.pop_left()
# my_deque.pop_right()
# print(f"Deque is empty: {my_deque.is_empty()}") # Expected: True
# my_deque.pop_left() # This will raise an IndexError
```

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.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.

A functional Python program implementing the chosen feature with comments and docstrings

Feature	Data Structure	Justification
Student Attendance Tracking	**Deque**	Deque provides O(1) append/pop from both ends, perfect for logging student entry (append_right) and exit (pop_left). Handles real-time tracking without shifting elements like lists. FIFO with bidirectional access matches entry/exit flow.
Event Registration System	**Hash Table**	O(1) average-case insert, search, delete by student ID ensures quick participant management. No ordering needed, just fast lookups/removals for

Feature	Data Structure	Justification
		event coordinators. Handles thousands of registrations efficiently.
Library Book Borrowing	**BST**	Sorted by book ID or due date enables O(log n) search/insert for availability checks. In-order traversal lists books chronologically. Self-balancing maintains performance as collection grows.
Bus Scheduling System	**Graph**	Models bus stops as vertices, routes as edges for connectivity/pathfinding. Adjacency list handles multiple routes per stop efficiently. Enables route planning and shortest path calculations.
Cafeteria Order Queue	**Queue**	Classic FIFO ensures students served in arrival order—first come, first served. O(1) enqueue/dequeue operations handle peak lunch rush smoothly. Simple, predictable serving sequence.

CODE : class EventRegistration:

```
    """
    Hash Table-based event registration system for quick student lookup/removal.
    Uses chaining for collision handling. Perfect for large events with frequent searches.
"
    def __init__(self, size=100):
        """Initialize hash table with fixed size."""
        self.size = size
        self.table = [[] for _ in range(size))  # Chaining with list
```

```
def _hash(self, student_id):
    """Simple hash function for student ID."""
    return hash(student_id) % self.size
def register(self, student_id, student_name):
    """Register student for event. O(1) average case."""
    if not student_id or not student_name:
        raise ValueError("Student ID and name required")
    index = self._hash(student_id)
    # Check if already registered
    for pair in self.table[index]:
        if pair[0] == student_id:
            raise ValueError(f"Student {student_id} already registered")
    self.table[index].append([student_id, student_name])
    print(f"✓ {student_name} (ID: {student_id}) registered successfully")


def unregister(self, student_id):
    """Remove student registration. O(1) average case."""
    index = self._hash(student_id)
    for i, pair in enumerate(self.table[index]):
        if pair[0] == student_id:
            removed = self.table[index].pop(i)
            print(f"✗ {removed[1]} (ID: {student_id}) unregistered")
            return True
    raise ValueError(f"Student ID {student_id} not found")
def search(self, student_id):
    """Check if student registered. O(1) average case."""
    index = self._hash(student_id)
    for pair in self.table[index]:
        if pair[0] == student_id:
            return f"Found: {pair[1]} (ID: {student_id})"
    return f"Student ID {student_id} not registered"
def get_total_registered(self):
    """Return total number of registered students."""
    total = 0
    for bucket in self.table:
```

```
        total += len(bucket)
        return total

    def display_all(self):
        """Display all registered students."""
        all_students = []
        for bucket in self.table:
            for student in bucket:
                all_students.append(f"ID: {student[0]}, Name: {student[1]}")
        return all_students if all_students else ["No registrations yet"]

# Demo: Campus Tech Fest Registration System
if __name__ == "__main__":
    event = EventRegistration(
    print("=== Campus Tech Fest Registration System ===\n")
    # Sample registrations
    event.register("STU001", "Alice Johnson")
    event.register("STU002", "Bob Smith")
    event.register("STU003", "Carol Davis")
    event.register("STU004", "David Wilson")
    event.register("STU005", "Eve Brown")
    print(f"\nTotal registered: {event.get_total_registered()}")
    # Search operations
    print("\n--- Search Tests ---")
    print(event.search("STU002"))
    print(event.search("STU999"))
    # Unregister test
    print("\n--- Unregister Test ---")
    event.unregister("STU003")
    # Final status
    print(f"\nFinal count: {event.get_total_registered()}")
    print("\nAll registered students:")
    for student in event.display_all():
        print(f"  {student}")
```

OUTPUT :

Alice Johnson (ID: STU001) registered successfully

Bob Smith (ID: STU002) registered successfully

Total registered: 5

--- Search Tests ---

Found: Bob Smith (ID: STU002)

Student ID STU999 not registered

Carol Davis (ID: STU003) unregistered

Final count: 4

All registered students:

ID: STU001, Name: Alice Johnson

ID: STU002, Name: Bob Smith

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.

Student Task:

- For each feature, select the most appropriate data structure from the list below:
  - Stack
  - Queue
  - Priority Queue
  - Linked List
  - Binary Search Tree (BST)
  - Graph
  - Hash Table
  - Deque
- Justify your choice in 2–3 sentences per feature.
- Implement one selected feature as a working Python program with AI-assisted code

generation.

Expected Output:

- A table mapping feature → chosen data structure → justification.
- A functional Python program implementing the chosen feature with comments and docstrings.

Feature	Data Structure	Justification
Shopping Cart	Hash Table	O(1) add/remove/update items by product ID; tracks quantities efficiently.codechef
Order Processing	Queue	FIFO processes orders in arrival sequence to avoid starvation.python-textbok.readthedocs
Top-Selling Tracker	Priority Queue	Heap ranks products by sales (priority); quick extract-max for top sellers.replit
Product Search	Hash Table	O(1) lookup by ID for instant retrieval in large catalogs.codechef
Delivery Routes	Graph	Vertices for locations, edges for routes; enables shortest path algorithms.peps.python

CODE : # E-Commerce Shopping Cart

```
cart = HashTable()

cart.insert("item1", 2)  # qty

cart.insert("item2", 1)

print("Item1 qty:", cart.search("item1"))  # 2

cart.delete("item2")

print("Cart after delete:", cart.search("item2"))  # None
```

OUTPUT :

```
Item1 qty: 2

Cart after delete: None
```

CODE :

```
# Given HashTable class from earlier (separate chaining)

cart = HashTable()          # Creates table with size=10, empty buckets


cart.insert("item1", 2)     # item1 → hashed → bucket.append(["item1", 2])

cart.insert("item2", 1)     # item2 → hashed → bucket.append(["item2", 1])


print("Item1 qty:", cart.search("item1"))  # Searches → finds → prints: 2


cart.delete("item2")        # Searches item2 bucket → removes pair


print("Cart after delete:", cart.search("item2"))  # Searches → not found → None
```

OUTPUT :

```
Before delete:

bucket[hash("item1")]: [["item1", 2]]

bucket[hash("item2")]: [["item2", 1]]


After delete:

bucket[hash("item1")]: [["item1", 2]]

bucket[hash("item2")]: []  ← Empty
```