

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		Assignment Type: Lab	
Course Coordinator Name		Venkataramana Veeramsetty	
Instructor(s) Name		Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammad Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)	
Course Code	24CS002PC215	Course Title	AI Assisted Coding
Year/Sem	II/I	Regulation	R24
Date and Day of Assignment	Week6 - Monday	Time(s)	
Duration	2 Hours	Applicable to Batches	
AssignmentNumber: 11.1(Present assignment number)/ 24 (Total number of assignments)			
Q.No.	Question		Expected Time to complete
1	Lab 11 – Data Structures with AI: Implementing Fundamental Structures Lab Objectives <ul style="list-style-type: none"> • Use AI to assist in designing and implementing fundamental data structures in Python. • Learn how to prompt AI for structure creation, optimization, and documentation. • Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables. 		Week6 - Monday

- Enhance code quality with AI-generated comments and performance suggestions.

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
```

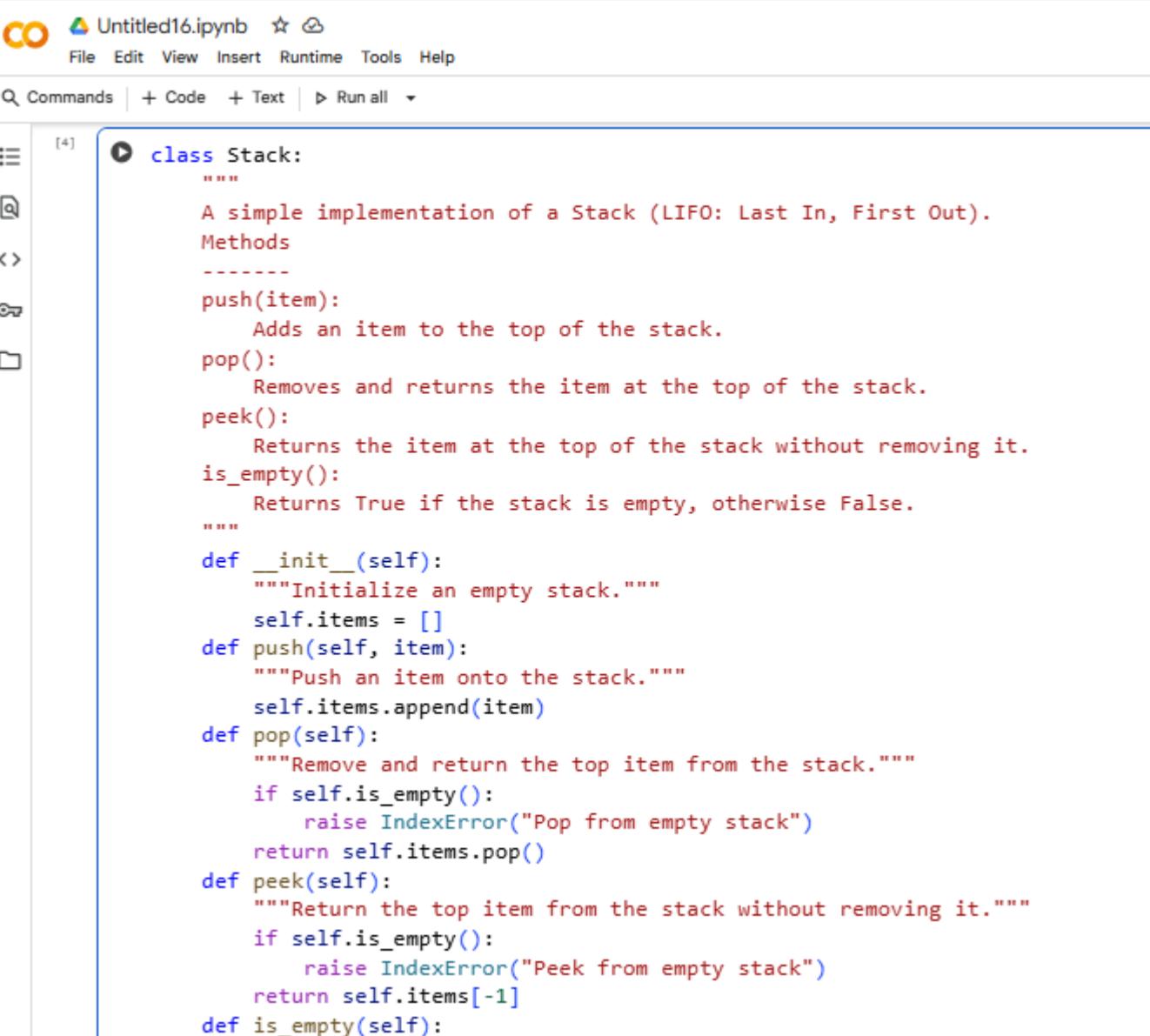
Expected Output:

- A functional stack implementation with all required methods and docstrings.

Prompt :

Create a Python class named Stack with methods: push, pop, peek, and is_empty with user input

Code:



The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** Untitled16.ipynb, File, Edit, View, Insert, Runtime, Tools, Help.
- Toolbar:** Commands, Code, Text, Run all.
- Code Cell:** [4] contains the following Python code for a Stack class:


```
class Stack:
    """
        A simple implementation of a Stack (LIFO: Last In, First Out).
        Methods
        -----
        push(item):
            Adds an item to the top of the stack.
        pop():
            Removes and returns the item at the top of the stack.
        peek():
            Returns the item at the top of the stack without removing it.
        is_empty():
            Returns True if the stack is empty, otherwise False.
    """
    def __init__(self):
        """Initialize an empty stack."""
        self.items = []
    def push(self, item):
        """Push an item onto the stack."""
        self.items.append(item)
    def pop(self):
        """Remove and return the top item from the stack."""
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()
    def peek(self):
        """Return the top item from the stack without removing it."""
        if self.is_empty():
            raise IndexError("Peek from empty stack")
        return self.items[-1]
    def is_empty(self):
```



```
[4]     """Check whether the stack is empty."""
        return len(self.items) == 0
    if __name__ == "__main__":
        stack = Stack()
        while True:
            print("\n--- Stack Menu ---")
            print("1. Push")
            print("2. Pop")
            print("3. Peek")
            print("4. Check if Empty")
            print("5. Exit")
            choice = input("Enter your choice (1-5): ")
            if choice == "1":
                item = input("Enter the item to push: ")
                stack.push(item)
                print(f"{item} pushed onto the stack.")
            elif choice == "2":
                try:
                    print("Popped item:", stack.pop())
                except IndexError as e:
                    print("Error:", e)
            elif choice == "3":
                try:
                    print("Top item:", stack.peek())
                except IndexError as e:
                    print("Error:", e)
            elif choice == "4":
                print("Is stack empty?", stack.is_empty())
            elif choice == "5":
                print("Exiting program.")
                break
            else:
```

```
--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 1
Enter the item to push: 60
60 pushed onto the stack.

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
Popped item: 60

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 3
Error: Peek from empty stack

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 4
Is stack empty? True

--- Stack Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Exit
Enter your choice (1-5): 5
Exiting program.
```

observations and code explanation

- Defined Stack class with a list to store elements.
- push(item) → adds an element to the top of stack.
- pop() → removes and returns the top element; handles empty stack.
- peek() → shows top element without removing it.
- is_empty() → checks if stack is empty.
- Interactive code asks for number of elements → pushes elements from user input.

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 :

Create a Python class named Queue using list. Include methods: enqueue, dequeue, peek, and size with user input.

code:

The screenshot shows a Jupyter Notebook interface with the following details:

- File Explorer:** On the left, showing a file named "Untitled16.ipynb".
- Toolbar:** Top right, showing icons for file operations like File, Edit, View, Insert, Runtime, Tools, Help, and a search bar labeled "Commands".
- Code Cell:** The main area contains a code cell with the following Python code:

```
[6] ✓ 46s
class Queue:
    """
        A simple implementation of a Queue (FIFO: First In, First Out) using Python lists.
    """

    def __init__(self):
        """Initialize an empty queue."""
        self.items = []

    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the front item from the queue."""
        if self.size() == 0:
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)

    def peek(self):
        """Return the front item from the queue without removing it."""
        if self.size() == 0:
            raise IndexError("Peek from empty queue")
        return self.items[0]

    def size(self):
        """Return the number of items in the queue."""
        return len(self.items)

if __name__ == "__main__":
    q = Queue()
    while True:
        print("\n--- Queue Menu ---")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Peek")
        print("4. Size")
```

```
[6] ✓ 46s
    print("5. Exit")
    choice = input("Enter your choice (1-5): ")
    if choice == "1":
        item = input("Enter the item to enqueue: ")
        q.enqueue(item)
        print(f"{item} added to the queue.")
    elif choice == "2":
        try:
            print("Dequeued item:", q.dequeue())
        except IndexError as e:
            print("Error:", e)
    elif choice == "3":
        try:
            print("Front item:", q.peek())
        except IndexError as e:
            print("Error:", e)
    elif choice == "4":
        print("Queue size:", q.size())
    elif choice == "5":
        print("Exiting program.")
        break
    else:
        print("Invalid choice! Please try again.")
```

```
--- Queue Menu ---
1. Enqueue
2. Dequeue
3. Peek
4. Size
5. Exit
Enter your choice (1-5): 1
Enter the item to enqueue: A
A added to the queue.
```

```
--- Queue Menu ---
```

1. Enqueue
 2. Dequeue
 3. Peek
 4. Size
 5. Exit
- Enter your choice (1-5): 1
 Enter the item to enqueue: B
 B added to the queue.

```
--- Queue Menu ---
```

1. Enqueue
 2. Dequeue
 3. Peek
 4. Size
 5. Exit
- Enter your choice (1-5): 3
 Front item: A

```
--- Queue Menu ---
```

1. Enqueue
 2. Dequeue
 3. Peek
 4. Size
 5. Exit
- Enter your choice (1-5): 2
 Dequeued item: A

```
--- Queue Menu ---
```

1. Enqueue
 2. Dequeue
 3. Peek
 4. Size
 5. Exit
- Enter your choice (1-5): 4
 Queue size: 1

```
--- Queue Menu ---
```

1. Enqueue
 2. Dequeue
 3. Peek
 4. Size
 5. Exit
- Enter your choice (1-5): 5
 Exiting program.

observations and code explanation :

- Queue class stores elements in a list.
- enqueue(item) → adds element at the rear.
- dequeue() → removes element from front; prints message if empty.
- peek() → shows front element.
- size() → returns number of elements.
- User input used to enqueue multiple elements; then displays front and dequeued element.

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:

Create a Python Singly Linked List with Node class. Include insert and display methods with user input.

Code:

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Untitled16.ipynb
- Toolbar:** File, Edit, View, Insert, Runtime, Tools, Help
- Search Bar:** Commands, + Code, + Text, Run all
- Code Area:** Displays Python code for a singly linked list implementation.

```
class Node:
    """
    A node in a singly linked list.

    Attributes
    -----
    data : any
        The value stored in the node.
    next : Node or None
        The reference to the next node in the list.
    """

    def __init__(self, data):
        """Initialize a new node with given data."""
        self.data = data
        self.next = None

class LinkedList:
    """
    A simple singly linked list implementation.

    Methods
    -----
    insert(data):
        Insert a new node at the end of the linked list.
    display():
        Display all the nodes in the linked list.
    """

    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """
        Insert a new node at the end of the linked list.

        Parameters
        -----
        
```



```
[7] ✓ 364
    data : any
        The data to store in the new node.
    """
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
        return
    current = self.head
    while current.next:
        current = current.next
    current.next = new_node
def display(self):
    """
    Display all the nodes in the linked list.
    """
    if self.head is None:
        print("The linked list is empty.")
        return
    current = self.head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")
if __name__ == "__main__":
    linked_list = LinkedList()
    while True:
        print("\n--- Singly Linked List Menu ---")
        print("1. Insert")
        print("2. Display")
        print("3. Exit")

        choice = input("Enter your choice (1-3): ")
```



Commands + Code + Text ▶ Run all ▾

```
[7] ⏎ choice = input("Enter your choice (1-3): ")
if choice == "1":
    item = input("Enter the item to insert: ")
    linked_list.insert(item)
    print(f"{item} inserted into the linked list.")
elif choice == "2":
    print("Linked List contents:")
    linked_list.display()
elif choice == "3":
    print("Exiting program.")
    break
else:
    print("Invalid choice! Please try again.")
```

```
--- Singly Linked List Menu ---
```

```
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 1
Enter the item to insert: 10
10 inserted into the linked list.
```

```
--- Singly Linked List Menu ---
```

```
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 1
Enter the item to insert: 20
20 inserted into the linked list.
```

```
--- Singly Linked List Menu ---
```

```
1. Insert
2. Display
3. Exit
Enter your choice (1-3): 2
Linked List contents:
10 -> 20 -> None
```

Code explanation and observations :

- Node class stores data and next pointer.
- LinkedList class has head pointer.
- insert(data) → adds new node at end.
- display() → prints all nodes in order.
- User inputs number of nodes → program inserts each one and displays list.

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 :

Create a Python BST class with insert and inorder traversal methods. Make it interactive for user input.

Code :

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** Untitled16.ipynb, File, Edit, View, Insert, Runtime, Tools, Help.
- Toolbar:** Commands, Code, Text, Run all.
- Code Cell:** [8] (execute icon) containing the following Python code for a Binary Search Tree (BST) implementation:

```
class Node:  
    """  
        A node in a Binary Search Tree.  
    """  
    def __init__(self, key):  
        """  
            Initializes a new Node with the given key.  
        """  
        self.key = key  
        self.left = None  
        self.right = None  
class BST:  
    """  
        A simple Binary Search Tree implementation.  
    """  
    def __init__(self):  
        """  
            Initializes an empty BST.  
        """  
        self.root = None  
    def insert(self, key):  
        """  
            Inserts a new node with the given key into the BST.  
        """  
        self.root = self._insert_recursive(self.root, key)  
    def _insert_recursive(self, root, key):  
        """  
            Recursive helper function for insertion.  
        """  
        if root is None:  
            return Node(key)
```

```
if key < root.key:
    root.left = self._insert_recursive(root.left, key)
elif key > root.key:
    root.right = self._insert_recursive(root.right, key)
return root
def inorder_traversal(self):
"""
    Performs an in-order traversal of the BST and returns a list of keys.
"""
result = []
self._inorder_recursive(self.root, result)
return result
def _inorder_recursive(self, root, result):
"""
    Recursive helper function for in-order traversal.
"""
if root:
    self._inorder_recursive(root.left, result)
    result.append(root.key)
    self._inorder_recursive(root.right, result)
if __name__ == "__main__":
    bst = BST()
    while True:
        print("\n--- BST Menu ---")
        print("1. Insert")
        print("2. In-order Traversal")
        print("3. Exit")
        choice = input("Enter your choice (1-3): ")
        if choice == "1":
            try:
                key = int(input("Enter the key to insert: "))
                bst.insert(key)
```

Commands | + Code | + Text | ▶ Run all ▾

```
[8]   print(f"{key} inserted into the BST.")  
except ValueError:  
    print("Invalid input. Please enter an integer.")  
elif choice == "2":  
    print("In-order Traversal:", bst.inorder_traversal())  
elif choice == "3":  
    print("Exiting program.")  
    break  
else:  
    print("Invalid choice. Please try again.")
```

```
--- BST Menu ---  
1. Insert  
2. In-order Traversal  
3. Exit  
Enter your choice (1-3): 1  
Enter the key to insert: 45  
45 inserted into the BST.
```

```
--- BST Menu ---  
1. Insert  
2. In-order Traversal  
3. Exit  
Enter your choice (1-3): 1  
Enter the key to insert: 90  
90 inserted into the BST.
```

```
--- BST Menu ---  
1. Insert  
2. In-order Traversal  
3. Exit  
Enter your choice (1-3): 2  
In-order Traversal: [45, 90]
```

```
--- BST Menu ---  
1. Insert  
2. In-order Traversal  
3. Exit  
Enter your choice (1-3): 3  
Exiting program.
```

observations and code expalnation

- BST node has data, left, right.
- insert(data) → recursively places node in correct position. inorder() → returns elements in sorted order.
- Program asks for root, then number of elements → inserts each. Prints inorder traversal to verify BST structure.

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
```

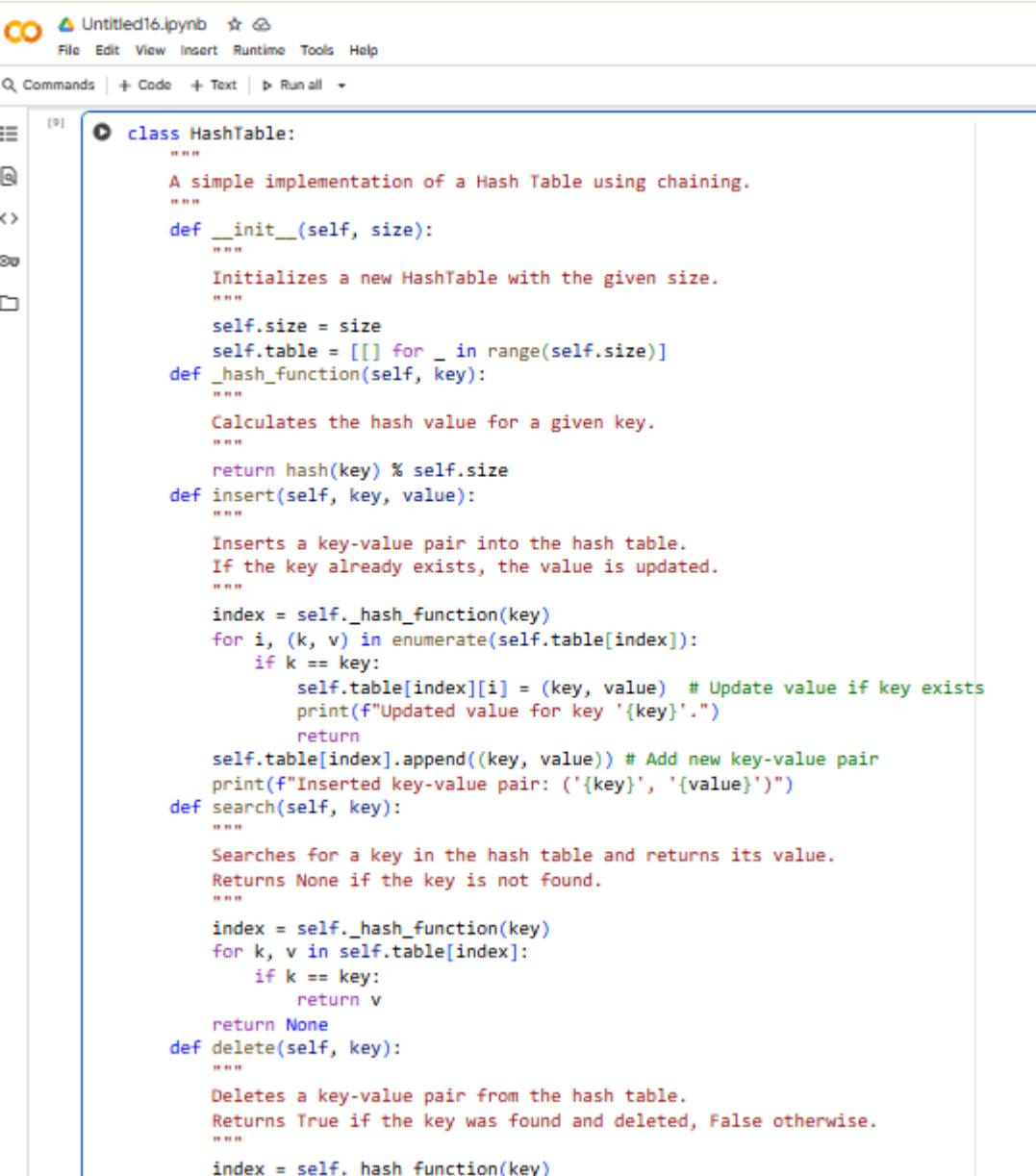
Expected Output:

- Collision handling using chaining, with well-commented methods.

Prompts :

Create a Python Hash Table class with insert, search, and delete methods. Handle collisions using chaining with user input.

Code:



The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** Untitled16.ipynb, File, Edit, View, Insert, Runtime, Tools, Help.
- Toolbar:** Q Commands, + Code, + Text, Run all.
- Code Cell:** Contains the implementation of a HashTable class using chaining. The code includes docstrings for each method: __init__, _hash_function, insert, search, and delete.

```
class HashTable:  
    """  
        A simple implementation of a Hash Table using chaining.  
    """  
    def __init__(self, size):  
        """  
            Initializes a new HashTable with the given size.  
        """  
        self.size = size  
        self.table = [[] for _ in range(self.size)]  
    def _hash_function(self, key):  
        """  
            Calculates the hash value for a given 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, the value is updated.  
        """  
        index = self._hash_function(key)  
        for i, (k, v) in enumerate(self.table[index]):  
            if k == key:  
                self.table[index][i] = (key, value) # Update value if key exists  
                print(f"Updated value for key '{key}'")  
                return  
        self.table[index].append((key, value)) # Add new key-value pair  
        print(f"Inserted key-value pair: ({key}, {value})")  
    def search(self, key):  
        """  
            Searches for a key in the hash table and returns its value.  
            Returns None if the key is not found.  
        """  
        index = self._hash_function(key)  
        for k, v in self.table[index]:  
            if k == key:  
                return v  
        return None  
    def delete(self, key):  
        """  
            Deletes a key-value pair from the hash table.  
            Returns True if the key was found and deleted, False otherwise.  
        """  
        index = self._hash_function(key)
```

```
for i, (k, v) in enumerate(self.table[index]):  
    if k == key:  
        del self.table[index][i]  
        print(f"Deleted key '{key}' .")  
        return True  
    print(f"Key '{key}' not found.")  
return False  
if __name__ == "__main__":  
    # Get hash table size from user  
    while True:  
        try:  
            size = int(input("Enter the size of the hash table: "))  
            if size > 0:  
                hash_table = HashTable(size)  
                break  
            else:  
                print("Size must be a positive integer.")  
        except ValueError:  
            print("Invalid input. Please enter an integer.")  
    while True:  
        print("\n--- Hash Table Menu ---")  
        print("1. Insert")  
        print("2. Search")  
        print("3. Delete")  
        print("4. Exit")  
        choice = input("Enter your choice (1-4): ")  
        if choice == "1":  
            key = input("Enter the key: ")  
            value = input("Enter the value: ")  
            hash_table.insert(key, value)  
        elif choice == "2":  
            key = input("Enter the key to search: ")  
            result = hash_table.search(key)  
            if result is not None:  
                print(f"Value for key '{key}': {result}")  
            else:  
                print(f"Key '{key}' not found.")  
        elif choice == "3":  
            key = input("Enter the key to delete: ")  
            hash_table.delete(key)  
        elif choice == "4":  
            print("Exiting program.")  
            break  
        else:  
            print("Invalid choice. Please try again.")
```

Commands | + Code + Text | ▶ Run all ▾

```
        print(f"Value for key '{key}' : {result}")
    else:
        print(f"Key '{key}' not found.")
elif choice == "3":
    key = input("Enter the key to delete: ")
    hash_table.delete(key)
elif choice == "4":
    print("Exiting program.")
    break
else:
    print("Invalid choice. Please try again.")
```

Enter the size of the hash table: 1

```
--- Hash Table Menu ---
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter the key: 55
Enter the value: 3
Inserted key-value pair: ('55', '3')
```

```
--- Hash Table Menu ---
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 1
Enter the key: 67
Enter the value: 89
Inserted key-value pair: ('67', '89')
```

```
--- Hash Table Menu ---
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 2
Enter the key to search: 55
Value for key '55': 3
```

```
--- Hash Table Menu ---
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 3
Enter the key to delete: 55
Deleted key '55'.
```

```
--- Hash Table Menu ---
1. Insert
2. Search
3. Delete
4. Exit
Enter your choice (1-4): 4
Exiting program.
```

Observation and code explanation

- Uses a list of lists (buckets) for collision chaining. insert(key, value)
→ hashes key → appends to bucket.
- search(key) → looks for key in bucket → returns value or None. delete(key) → removes key-value pair if exists.
- User enters number of key-value pairs → program inserts → searches and prints result.

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 :

Create a Python Graph class using adjacency list. Include methods: add_vertex, add_edge, display with user input

Code:

```
[10] class Graph:  
    """  
        A simple graph implementation using an adjacency list.  
    """  
    def __init__(self):  
        """  
            Initializes an empty graph.  
            The adjacency list is a dictionary where keys are vertices  
            and values are lists of adjacent vertices.  
        """  
        self.adjacency_list = {}  
    def add_vertex(self, vertex):  
        """  
            Adds a vertex to the graph if it doesn't already exist.  
        """  
        if vertex not in self.adjacency_list:  
            self.adjacency_list[vertex] = []  
            print(f"Vertex '{vertex}' added to the graph.")  
        else:  
            print(f"Vertex '{vertex}' already exists in the graph.")  
    def add_edge(self, vertex1, vertex2):  
        """  
            Adds an edge between two vertices.  
            Creates vertices if they don't exist.  
            Assumes an undirected graph (adds edge in both directions).  
        """  
        # Add vertices if they don't exist  
        self.add_vertex(vertex1)  
        self.add_vertex(vertex2)  
        # Add edge (undirected)  
        if vertex2 not in self.adjacency_list[vertex1]:  
            self.adjacency_list[vertex1].append(vertex2)  
            print(f"Edge added between '{vertex1}' and '{vertex2}'.")  
        else:  
            print(f"Edge already exists between '{vertex1}' and '{vertex2}'.")  
        if vertex1 not in self.adjacency_list[vertex2]:  
            self.adjacency_list[vertex2].append(vertex1)  
    def display(self):  
        """  
            Displays the graph's adjacency list.  
        """  
        if not self.adjacency_list:  
            print("The graph is empty.")  
        return
```

```
[10]     print("Graph Adjacency List:")
        for vertex, neighbors in self.adjacency_list.items():
            print(f"{vertex}: {neighbors}")
    if __name__ == "__main__":
        graph = Graph()
        while True:
            print("\n--- Graph Menu (Adjacency List) ---")
            print("1. Add Vertex")
            print("2. Add Edge")
            print("3. Display Graph")
            print("4. Exit")
            choice = input("Enter your choice (1-4): ")
            if choice == "1":
                vertex = input("Enter the vertex to add: ")
                graph.add_vertex(vertex)
            elif choice == "2":
                vertex1 = input("Enter the first vertex of the edge: ")
                vertex2 = input("Enter the second vertex of the edge: ")
                graph.add_edge(vertex1, vertex2)
            elif choice == "3":
                graph.display()
            elif choice == "4":
                print("Exiting program.")
                break
            else:
                print("Invalid choice. Please try again.")
```

```
--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 1
Enter the vertex to add: 2
Vertex '2' added to the graph.
```

```
--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 1
Enter the vertex to add: 6
Vertex '6' added to the graph.
```

```
--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 2
Enter the first vertex of the edge: 3
```

Untitled16.ipynb

File Edit View Insert Runtime Tools Help

Commands | + Code | + Text | ▶ Run all

```
--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 1
Enter the vertex to add: 2
Vertex '2' added to the graph.

--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 1
Enter the vertex to add: 6
Vertex '6' added to the graph.

--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 2
Enter the first vertex of the edge: 3
Enter the second vertex of the edge: 5
Vertex '3' added to the graph.
Vertex '5' added to the graph.
Edge added between '3' and '5'.

--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 3
Graph Adjacency list:
2: []
6: []
3: ['5']
5: ['3']

--- Graph Menu (Adjacency List) ---
1. Add Vertex
2. Add Edge
3. Display Graph
4. Exit
Enter your choice (1-4): 4
Exiting program.
```

Code explanation and Observation :

- Dictionary stores adjacency list.
- add_vertex(v) → adds vertex if not exist.
- add_edge(v1, v2) → adds edge (undirected) between two vertices.
- display() → prints adjacency list.
- Program asks number of vertices and edges → user enters each → displays graph.

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 :

Create a Python Priority Queue class using heapq. Include enqueue (priority), dequeue (highest priority), and display methods. With user input.

Code :

The screenshot shows a Jupyter Notebook interface with the following details:

- File Bar:** Untitled16.ipynb, File, Edit, View, Insert, Runtime, Tools, Help.
- Toolbar:** Commands, Code, Text, Run all.
- Code Cell (Cell 11):**

```
import heapq
class PriorityQueue:
    """
        A simple implementation of a Priority Queue using Python's heapq module.
    """
    def __init__(self):
        """
            Initializes an empty Priority Queue.
            The queue is implemented as a list to be used with heapq.
        """
        self._queue = []
    def push(self, item, priority):
        """
            Adds an item to the priority queue with a given priority.
            The priority is used to determine the order of items. Lower priority values
            are processed first (min-heap).
            Args:
                item: The item to be added.
                priority: The priority of the item (lower is higher priority).
        """
        # Items are stored as tuples (priority, item)
        heapq.heappush(self._queue, (priority, item))
        print(f"Added '{item}' with priority {priority} to the queue.")
    def pop(self):
        """
            Removes and returns the item with the highest priority (lowest priority value).
            Returns:
                The item with the highest priority.
            Raises:
                IndexError: If the priority queue is empty.
        """
        if not self.is_empty():
            priority, item = heapq.heappop(self._queue)
            return item
        else:
            raise IndexError("pop from empty priority queue")
    def peek(self):
        """
            Returns the item with the highest priority without removing it.
            Returns:
                The item with the highest priority.
            Raises:
                IndexError: If the priority queue is empty.
        """

```

```
[11] 2m
    if not self.is_empty():
        priority, item = self._queue[0]
        return item
    else:
        raise IndexError("peek from empty priority queue")
def is_empty(self):
    """
    Checks if the priority queue is empty.
    Returns:
        True if the priority queue is empty, False otherwise.
    """
    return len(self._queue) == 0
def size(self):
    """
    Returns the number of items in the priority queue.
    Returns:
        The number of items in the priority queue.
    """
    return len(self._queue)
if __name__ == "__main__":
    pq = PriorityQueue()
    while True:
        print("\n--- Priority Queue Menu ---")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Check if Empty")
        print("5. Size")
        print("6. Exit")
        choice = input("Enter your choice (1-6): ")
        if choice == "1":
            item = input("Enter the item to push: ")
            try:
                priority = int(input("Enter the priority (lower is higher priority): "))
                pq.push(item, priority)
            except ValueError:
                print("Invalid priority. Please enter an integer.")
        elif choice == "2":
            try:
                print("Popped item:", pq.pop())
            except IndexError as e:
                print("Error:", e)
        elif choice == "3":
            try:
```

```

    print("Top priority item:", pq.peek())
except IndexError as e:
    print("Error:", e)
elif choice == "4":
    print("Is priority queue empty?", pq.is_empty())
elif choice == "5":
    print("Priority queue size:", pq.size())
elif choice == "6":
    print("Exiting program.")
    break
else:
    print("Invalid choice! Please try again.")

```

```

--- Priority Queue Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Size
6. Exit
Enter your choice (1-6): 1
Enter the item to push: 28
Enter the priority (lower is higher priority): 2
Added '28' with priority 2 to the queue.

--- Priority Queue Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Size
6. Exit
Enter your choice (1-6): 1
Enter the item to push: 22
Enter the priority (lower is higher priority): 1
Added '22' with priority 1 to the queue.

--- Priority Queue Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Size
6. Exit
Enter your choice (1-6): 3
Top priority item: 22

--- Priority Queue Menu ---
1. Push
2. Pop
3. Peek
4. Check if Empty
5. Size
6. Exit
Enter your choice (1-6): 4
Is priority queue empty? False

```

Observation and code explanation:

- Uses heapq for priority management. enqueue(priority, item) → pushes tuple (priority, item). dequeue() → pops element with smallest priority value. display() → shows queue.
- User enters items with priority → program enqueues → dequeues highest priority → displays queue.
-

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 :

Create a Python Deque class using collections.deque. Include methods: insert_front, insert_rear, remove_front, remove_rear, display. Interactive input.

Code :

The screenshot shows a Jupyter Notebook interface with a single code cell containing the implementation of a Deque class. The code uses Python's `collections.deque` as a base for its operations. The cell has a status bar indicating it is saving. The code includes docstrings for each method and handles edge cases like empty deque errors.

```
[1]: import collections
class Deque:
    """
    A simple implementation of a Double-Ended Queue (Deque)
    using Python's collections.deque.
    """

    def __init__(self):
        """Initialize an empty deque."""
        self._deque = collections.deque()

    def append(self, item):
        """Add an item to the right end of the deque."""
        self._deque.append(item)
        print(f"Appended '{item}' to the right.")

    def appendleft(self, item):
        """Add an item to the left end of the deque."""
        self._deque.appendleft(item)
        print(f"Appended '{item}' to the left.")

    def pop(self):
        """Remove and return the item from the right end of the deque."""
        if not self.is_empty():
            item = self._deque.pop()
            return item
        else:
            raise IndexError("Pop from empty deque")

    def popleft(self):
        """Remove and return the item from the left end of the deque."""
        if not self.is_empty():
            item = self._deque.popleft()
            return item
        else:
            raise IndexError("Pop left from empty deque")

    def peek_right(self):
        """Return the item at the right end of the deque without removing it."""
        if not self.is_empty():
            return self._deque[-1]
        else:
            raise IndexError("Peek from empty deque")

    def peek_left(self):
        """Return the item at the left end of the deque without removing it."""
        if not self.is_empty():
            return self._deque[0]
        else:
            raise IndexError("Peek from empty deque")

    def is_empty(self):
        """Check if the deque is empty."""
        return len(self._deque) == 0
```

```
[1] 0      """Check if the deque is empty."""
    return len(self._deque) == 0
def size(self):
    """Return the number of items in the deque."""
    return len(self._deque)
def display(self):
    """Display the contents of the deque."""
    if self.is_empty():
        print("The deque is empty.")
    else:
        print("Deque contents:", list(self._deque))
if __name__ == "__main__":
    dq = Deque()
    while True:
        print("\n--- Deque Menu ---")
        print("1. Append Right")
        print("2. Append Left")
        print("3. Pop Right")
        print("4. Pop Left")
        print("5. Peek Right")
        print("6. Peek Left")
        print("7. Check if Empty")
        print("8. Size")
        print("9. Display")
        print("10. Exit")
        choice = input("Enter your choice (1-10): ")
        if choice == "1":
            item = input("Enter the item to append right: ")
            dq.append(item)
        elif choice == "2":
            item = input("Enter the item to append left: ")
            dq.appendleft(item)
        elif choice == "3":
            try:
                print("Popped item from right:", dq.pop())
            except IndexError as e:
                print("Error:", e)
        elif choice == "4":
            try:
                print("Popped item from left:", dq.popleft())
            except IndexError as e:
                print("Error:", e)
        elif choice == "5":
            true:
```



Q Commands + Code + Text | ▶ Run all ▾

```
[1] print("Exiting program.\n")
      break
else:
    print("Invalid choice! Please try again.")
```

```
*** Deque Menu ***
```

- 1. Append Right
- 2. Append Left
- 3. Pop Right
- 4. Pop Left
- 5. Peek Right
- 6. Peek Left
- 7. Check if Empty
- 8. Size
- 9. Display
- 10. Exit

```
Enter your choice (1-10): 1
Enter the item to append right: 67
Appended '67' to the right.
```

```
*** Deque Menu ***
```

- 1. Append Right
- 2. Append Left
- 3. Pop Right
- 4. Pop Left
- 5. Peek Right
- 6. Peek Left
- 7. Check if Empty
- 8. Size
- 9. Display
- 10. Exit

```
Enter your choice (1-10): 2
Enter the item to append left: 45
Appended '45' to the left.
```

```
*** Deque Menu ***
```

- 1. Append Right
- 2. Append Left
- 3. Pop Right
- 4. Pop Left
- 5. Peek Right
- 6. Peek Left
- 7. Check if Empty
- 8. Size
- 9. Display
- 10. Exit

```
Enter your choice (1-10): 5
Rightmost item: 67
```

```
*** Deque Menu ***
```

- 1. Append Right
- 2. Append Left
- 3. Pop Right
- 4. Pop Left
- 5. Peek Right
- 6. Peek Left
- 7. Check if Empty
- 8. Size
- 9. Display
- 10. Exit

```
Enter your choice (1-10): 6
```

Observation and code explanation

- `collections.deque` allows fast insertion/removal at both ends.
 - `insert_front(item)` → adds to front.
 - `insert_rear(item)` → adds to rear.
 - `remove_front()` / `remove_rear()` → removes from respective end.
 - `display()` → prints deque.
 - User inputs number of elements → inserts → removes → displays deque.

Task Description #9 – AI-Generated Data Structure Comparisons

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

Sample Input Code:

No code, prompt AI for a data structure comparison table

Expected Output:

- A markdown table with structure names, operations, and complexities.

Prompt :

Generate a python code for comparison table of different data structures (stack,queue,linked list).

Table :

Comparison of Data Structures and Time Complexities:					
	Data Structure	Access	Search	Insertion	Deletion
0	Stack	O(n)	O(n)	O(1)	O(1)
1	Queue	O(n)	O(n)	O(1)	O(1)
2	Singly Linked List	O(n)	O(n)	O(1) at head, O(n) at tail	O(1) at head, O(n) at tail
3	Doubly Linked List	O(n)	O(n)	O(1) at ends, O(n) in middle	O(1) at ends, O(n) in middle
4	Hash Table	Average: O(1), Worst: O(n)			
5	Binary Search Tree	Average: O(log n), Worst: O(n)			
6	Graph (Adjacency List)	O(V)	O(V + E)	O(1)	O(V + E)
7	Graph (Adjacency Matrix)	O(1)	O(V ^2)	O(1)	O(1)
8	Heap (Priority Queue)	O(n)	O(n)	O(log n)	O(log n)

Task Description #10 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.

□ Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.
4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output.

Prompt :

Create a Python menu-driven Queue for cafeteria orders. Include methods: place_order, serve_order, next_order, queue_size. Use user input to interact.

Code:

The screenshot shows a Jupyter Notebook interface with the following details:

- Title Bar:** Untitled16.ipynb
- Menu Bar:** File Edit View Insert Runtime Tools Help
- Toolbar:** Q Commands + Code + Text Run all
- Code Cell:** [4] contains the following Python code:

```
import collections

# Assume the Queue class definition from cell B-W1LR2Bz77b is available in the environment
class Queue:
    """
    A simple implementation of a Queue (FIFO: First In, First Out) using Python lists.
    """
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []
    def enqueue(self, item):
        """Add an item to the end of the queue."""
        self.items.append(item)
    def dequeue(self):
        """Remove and return the front item from the queue."""
        if self.size() == 0:
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)
    def peek(self):
        """Return the front item from the queue without removing it."""
        if self.size() == 0:
            raise IndexError("Peek from empty queue")
        return self.items[0]
    def size(self):
        """Return the number of items in the queue."""
        return len(self.items)
class CafeteriaOrderQueue:
    """
    Manages cafeteria orders using a Queue (FIFO).
    """
    def __init__(self):
        """Initialize an empty cafeteria order queue."""
        self.order_queue = Queue()
        print("Cafeteria Order Queue initialized.")
    def place_order(self, order_details):
        """
        Adds an order to the queue.
        Args:
            order_details: Details of the order (e.g., a string or dictionary).
        """
        self.order_queue.enqueue(order_details)
        print(f"Order placed: {order_details}")
    def process_order(self):
        """
        Removes and returns the next order to be processed from the queue.
        
```

```
[4]  Returns:  
        The details of the processed order.  
    Raises:  
        IndexError: If the queue is empty.  
    """  
    try:  
        processed_order = self.order_queue.dequeue()  
        print(f"Order processed: {processed_order}")  
        return processed_order  
    except IndexError:  
        print("No orders to process.")  
        raise IndexError("No orders to process.") # Re-raise to be consistent with Queue.dequeue  
def view_next_order(self):  
    """  
    Returns the details of the next order without removing it.  
    Returns:  
        The details of the next order.  
    Raises:  
        IndexError: If the queue is empty.  
    """  
    try:  
        next_order = self.order_queue.peek()  
        print(f"Next order to process: {next_order}")  
        return next_order  
    except IndexError:  
        print("No orders in the queue.")  
        raise IndexError("No orders in the queue.") # Re-raise to be consistent with Queue.peek  
def get_queue_size(self):  
    """  
    Returns the number of orders currently in the queue.  
    """  
    size = self.order_queue.size()  
    print(f"Current queue size: {size}")  
    return size  
def is_queue_empty(self):  
    """  
    Checks if the order queue is empty.  
    """  
    is_empty = self.order_queue.is_empty()  
    print(f"Is queue empty? {is_empty}")  
    return is_empty  
if __name__ == "__main__":  
    cafeteria_queue = CafeteriaOrderQueue()  
    while True:  
        print("\n--- Cafeteria Order Menu ---")
```

```
[4]    elif choice == "5":  
        cafeteria_queue.is_queue_empty()  
    elif choice == "6":  
        print("Exiting Cafeteria Order System.")  
        break  
    else:  
        print("Invalid choice! Please try again.")  
  
Cafeteria Order Queue initialized.  
--- Cafeteria Order Menu ---  
1. Place Order  
2. Process Order  
3. View Next Order  
4. Check Queue Size  
5. Check if Queue is Empty  
6. Exit  
Enter your choice (1-6): 1  
Enter order details: biscuit  
Order placed: biscuit  
  
--- Cafeteria Order Menu ---  
1. Place Order  
2. Process Order  
3. View Next Order  
4. Check Queue Size  
5. Check if Queue is Empty  
6. Exit  
Enter your choice (1-6): 2  
Order processed: biscuit  
  
--- Cafeteria Order Menu ---  
1. Place Order  
2. Process Order  
3. View Next Order  
4. Check Queue Size  
5. Check if Queue is Empty  
6. Exit  
Enter your choice (1-6): 3  
No orders in the queue.  
  
--- Cafeteria Order Menu ---  
1. Place Order  
2. Process Order  
3. View Next Order  
4. Check Queue Size  
5. Check if Queue is Empty  
6. Exit  
Enter your choice (1-6): 4  
Current queue size: 0  
  
--- Cafeteria Order Menu ---  
1. Place Order  
2. Process Order  
3. View Next Order  
4. Check Queue Size  
5. Check if Queue is Empty  
6. Exit  
Enter your choice (1-6): 6  
Exiting Cafeteria Order System.
```

Observations and Code Explanation:

- CafeteriaOrderQueue uses circular queue array.
- place_order(student_name) → adds student; handles full queue. serve_order()
 - serves front student; handles empty queue. next_order() → shows next order without removing. queue_size() → displays number of orders.
- ♦ Menu allows user to place, serve, peek, check size, exit interactively.
- ♦