2303a510f2 – Anitha

Batch – 14

Assignment – 11.1

TASK 1: Stack Implementat

Prompt:

Generate a Python class named Stack that implements a stack data structure using a list.
The class must include the following methods:

- push(item) to add an element to the stack
- pop() to remove and return the top element
- peek() to return the top element without removing it
- is_empty() to check whether the stack is empty

The implementation should include proper docstrings for the class and each method.

CODE:

```
class Stack:
    """A simple implementation of a stack data structure using a list."""

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

    def push(self, item):
        """Add an element to the top of the stack.

        Args:
            item: The element to be added to the stack.
        """
        self.stack.append(item)
```

```
    def pop(self):
        """Remove and return the top element of the stack.

        Returns:
            The top element of the stack if it is not empty; otherwise, raises an
exception.
        """
        if self.is_empty():
            raise IndexError("Pop from an empty stack")
        return self.stack.pop()

    def peek(self):
        """Return the top element of the stack without removing it.

        Returns:
            The top element of the stack if it is not empty; otherwise, raises an
exception.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty stack")
        return self.stack[-1]

    def is_empty(self):
        """Check whether the stack is empty.

        Returns:
            True if the stack is empty; otherwise, False.
        """
        return len(self.stack) == 0

n = int(input())
arr = list(map(int, input().split()))
st = Stack()
for num in arr:
    st.push(num)
q = int(input())
for _ in range(q):
    command = input().split()
    if command[0] == "push":
        st.push(int(command[1]))
    elif command[0] == "pop":
        print(st.pop())
    elif command[0] == "peek":
```

```
        print(st.peek())
    elif command[0] == "is_empty":
        print(st.is_empty())
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\
3.13.exe c:/Users/akshi/OneDrive/De
5
1 5 3 4 6
5
peek
6
push 8
pop
8
is_empty
False
pop
6
```

TASK 2: Queue Implementation

PROMPT:

Generate a Python class named Queue that implements a FIFO (First In, First Out) queue using a Python list, including the methods enqueue(item) to add an element to the rear, dequeue() to remove and return the front element, peek() to return the front element without removing it, and size() to return the number of elements in the queue, with proper docstrings for the class and all methods.

CODE:

```
class Queue:
    """A simple implementation of a FIFO (First In, First Out) queue using a
list."""

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

    def enqueue(self, item):
        """Add an element to the rear of the queue.

        Args:
            item: The element to be added to the queue.
        """
        self.queue.append(item)

    def dequeue(self):
        """Remove and return the front element of the queue.

        Returns:
            The front element of the queue if it is not empty; otherwise, raises
an exception.
        """
        if self.is_empty():
            raise IndexError("Dequeue from an empty queue")
        return self.queue.pop(0)

    def peek(self):
        """Return the front element of the queue without removing it.

        Returns:
            The front element of the queue if it is not empty; otherwise, raises
an exception.
        """
        if self.is_empty():
            raise IndexError("Peek from an empty queue")
        return self.queue[0]

    def size(self):
        """Return the number of elements in the queue.

        Returns:
            The number of elements in the queue.
```

```
        """
        return len(self.queue)

    def is_empty(self):
        """Check whether the queue is empty.

        Returns:
            True if the queue is empty; otherwise, False.
        """
        return len(self.queue) == 0

n = int(input())
arr = list(map(int, input().split()))
q = int(input())
queue = Queue()
for num in arr:
    queue.enqueue(num)
for _ in range(q):
    command = input().split()
    if command[0] == "enqueue":
        queue.enqueue(int(command[1]))
    elif command[0] == "dequeue":
        print(queue.dequeue())
    elif command[0] == "peek":
        print(queue.peek())
    elif command[0] == "size":
        print(queue.size())
    elif command[0] == "is_empty":
        print(queue.is_empty())
    else:
        print("Invalid command")
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\f
s/akshi/OneDrive/Desktop/11.1.py
5
1 3 6 5 4
6
peek
1
enqueue 8
dequeue
1
peek
3
size
5
is_empty
False
```

TASK 3: Linked List

PROMPT:

Generate a Python implementation of a Singly Linked List using two classes, Node and LinkedList, where the LinkedList class includes insert(data) to add a node at the end of the list and display() to print all elements in the list, with clear docstrings for all classes and methods.

CODE:

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

    def __init__(self, data):
        """Initialize a node with the given data and a reference to the next
node.

        Args:
            data: The data to be stored in the node.
        """
        self.data = data
        self.next = None
class LinkedList:
    """A class representing a singly linked list."""
```

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

    def insert(self, data):
        """Add a node with the given data at the end of the list.

        Args:
            data: The data to be stored in the new node.
        """
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def display(self):
        """Print all elements in the linked list."""
        current_node = self.head
        while current_node:
            print(current_node.data, end='->')
            current_node = current_node.next
        print("NONE")  # for a new line after printing all elements
n = int(input())
arr = list(map(int, input().split()))
linked_list = LinkedList()
for num in arr:
    linked_list.insert(num)
linked_list.display()
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Deskto
s/akshi/OneDrive/Desktop/11.1.py
5
10 20 30 40 50
10->20->30->40->50->NONE
```

TASK 4:BINARY SEARCH TREE

PROMPT:

Generate a Python class named BST that implements a Binary Search Tree with recursive insert(data) and inorder() traversal methods. The insert method should recursively add elements while maintaining BST properties, and the inorder method should perform an in-order traversal (Left → Root → Right). Include proper docstrings for all classes and methods.

CODE:

```
class Node:
    """A class representing a node in a binary search tree."""

    def __init__(self, data):
        """Initialize a node with the given data and references to left and right
children.

        Args:
            data: The data to be stored in the node.
        """
        self.data = data
        self.left = None
        self.right = None
class BST:
    """A class representing a binary search tree."""

    def __init__(self):
        """Initialize an empty binary search tree."""
        self.root = None

    def insert(self, data):
        """Recursively add an element to the binary search tree while maintaining
BST properties.
```

```
        Args:
            data: The data to be inserted into the tree.
        """
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        """Helper method to recursively insert a new node into the BST.

        Args:
            node: The current node in the recursion.
            data: The data to be inserted.
        """
        if data < node.data:
            if node.left is None:
                node.left = Node(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = Node(data)
            else:
                self._insert_recursive(node.right, data)

    def inorder(self):
        """Perform an in-order traversal of the BST and print the elements."""
        self._inorder_recursive(self.root)

    def _inorder_recursive(self, node):
        """Helper method to recursively perform in-order traversal.

        Args:
            node: The current node in the recursion.
        """
        if node is not None:
            self._inorder_recursive(node.left)
            print(node.data, end=' ')
            self._inorder_recursive(node.right)
n = int(input())
arr = list(map(int, input().split()))
bst = BST()
```

```
for num in arr:
    bst.insert(num)
bst.inorder()
```

OUTPUT:

```
PS C:\Users\akshi\OneD
s/akshi/OneDrive/Deskt
5
50 30 70 20 40
20 30 40 50 70
```

TASK 5: HASH TABLE

PROMPT:

Implement a Hash Table in Python with insert(key, value), search(key), and delete(key) methods using separate chaining (linked lists or Python lists) for collision handling, a simple modulo-based hash function, well-commented code, and include example usage demonstrating insert, search, and delete operations.

CODE:

```
class HashTable:
    """A simple implementation of a hash table using separate chaining for
collision handling."""

    def __init__(self, size=10):
        """Initialize the hash table with a specified size.

        Args:
            size: The number of buckets in the hash table (default is 10).
        """
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists
for separate chaining
```

```
    def _hash(self, key):
        """Compute the hash value for a given key using a simple modulo-based
hash function.

        Args:
            key: The key to be hashed.

        Returns:
            The hash value corresponding to the key.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table.

        Args:
            key: The key to be inserted.
            value: The value associated with the key.
        """
        index = self._hash(key)
        # Check if the key already exists and update its value
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        # If the key does not exist, add a new key-value pair
        self.table[index].append((key, value))

    def search(self, key):
        """Search for a value associated with a given key in the hash table.

        Args:
            key: The key to be searched.

        Returns:
            The value associated with the key if found; otherwise, None.
        """
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if the key is found
        return None  # Return None if the key is not found
```

```
    def delete(self, key):
        """Delete a key-value pair from the hash table based on the given key.

        Args:
            key: The key to be deleted.
        """
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair if found
                return
# Example usage
if __name__ == "__main__":

    hash_table = HashTable()
    hash_table.insert("name", "Alice")
    hash_table.insert("age", 30)
    hash_table.insert("city", "New York")

    print(hash_table.search("name"))  # Output: Alice
    print(hash_table.search("age"))   # Output: 30
    print(hash_table.search("city"))  # Output: New York
    print(hash_table.search("country"))  # Output: None

    hash_table.delete("age")
    print(hash_table.search("age"))   # Output: NonE
```

OUTPUT:

```
PS C:\Users\akshi\C
s/akshi/OneDrive/De
Alice
30
New York
None
None
```

TASK 6:GRAPH REPRESENTATION

PROMPT:

Implement a Graph class in Python using an adjacency list representation with methods to add vertices, add edges, and display connections, including clear and well-commented code with example usage.

CODE:

```
class Graph:
    """A simple implementation of a graph using an adjacency list
representation."""

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

    def add_vertex(self, vertex):
        """Add a vertex to the graph.

        Args:
            vertex: The vertex to be added.
        """
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph.

        Args:
            vertex1: The first vertex.
            vertex2: The second vertex.
        """
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)  # For undirected graph

    def display(self):
        """Display the connections in the graph."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {', '.join(edges)}")
# Example usage
if __name__ == "__main__":
    graph = Graph()
    graph.add_vertex("A")
```

```
    graph.add_vertex("B")
    graph.add_vertex("C")
    graph.add_edge("A", "B")
    graph.add_edge("A", "C")
    graph.add_edge("B", "C")

    graph.display()
    # Output:
    # A: B, Cc
```

OUTPUT:

```
PS C:\Users\a
s/akshi/OneDr
A: B, C
B: A, C
C: A, B
```

TASK 7: PRIORITY QUEUE

PROMPT:

Implement a PriorityQueue class in Python using the heapq module with enqueue (priority), dequeue (highest priority), and display methods, including clear and well-commented code with example usage.

CODE:

```
import heapq
class PriorityQueue:
    """A simple implementation of a priority queue using the heapq module."""

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

    def enqueue(self, item, priority):
```

```
        """Add an item to the priority queue with a given priority.

        Args:
            item: The item to be added to the queue.
            priority: The priority of the item (lower values indicate higher
priority).
        """
        heapq.heappush(self.elements, (priority, item))

    def dequeue(self):
        """Remove and return the item with the highest priority from the queue.

        Returns:
            The item with the highest priority if the queue is not empty;
otherwise, raises an exception.
        """
        if not self.elements:
            raise IndexError("Dequeue from an empty priority queue")
        return heapq.heappop(self.elements)[1]  # Return only the item, not the
priority

    def display(self):
        """Display all items in the priority queue along with their
priorities."""
        for priority, item in self.elements:
            print(f"Item: {item}, Priority: {priority}")
# Example usage
if __name__ == "__main__":
    pq = PriorityQueue()
    pq.enqueue("Task 1", priority=2)
    pq.enqueue("Task 2", priority=1)
    pq.enqueue("Task 3", priority=3)

    print("Priority Queue:")
    pq.display()

    print("\nDequeueing items:")
    while True:
        try:
            print(pq.dequeue())
        except IndexError:
            print("Priority queue is empty.")
            break
```

OUTPUT:

```
Item: Task 3, Priority: 3

Dequeueing items:
Task 2
Task 1
Task 3
Priority queue is empty.
```

TASK 8: DEQUEUE

PROMPT:

Implement a DequeDS class in Python using collections.deque with methods to insert and remove elements from both ends, including proper docstrings and example usage.

CODE:

```
from collections import deque
class DequeDS:

    """A simple implementation of a double-ended queue (deque) using
collections.deque."""

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

    def insert_front(self, item):
        """Insert an element at the front of the deque.

        Args:
            item: The element to be inserted at the front.
```

```
        """
        self.deque.appendleft(item)

    def insert_rear(self, item):
        """Insert an element at the rear of the deque.

        Args:
            item: The element to be inserted at the rear.
        """
        self.deque.append(item)

    def remove_front(self):
        """Remove and return the element at the front of the deque.

        Returns:
            The element at the front if the deque is not empty; otherwise, raises
an exception.
        """
        if not self.deque:
            raise IndexError("Remove from an empty deque")
        return self.deque.popleft()

    def remove_rear(self):
        """Remove and return the element at the rear of the deque.

        Returns:
            The element at the rear if the deque is not empty; otherwise, raises
an exception.
        """
        if not self.deque:
            raise IndexError("Remove from an empty deque")
        return self.deque.pop()
# Example usage
if __name__ == "__main__":
    deque_ds = DequeDS()
    deque_ds.insert_rear("Task 1")
    deque_ds.insert_rear("Task 2")
    deque_ds.insert_front("Task 0")

    print("Deque after insertions:")
    print(deque_ds.deque)

    print("\nRemoving from front:", deque_ds.remove_front())
```

```
    print("Removing from rear:", deque_ds.remove_rear())

    print("\nDeque after removals:")
    print(deque_ds.deque)
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\full s
s/akshi/OneDrive/Desktop/11.1.py
Deque after insertions:
deque(['Task 0', 'Task 1', 'Task 2'])

Removing from front: Task 0
Removing from rear: Task 2

Deque after removals:
deque(['Task 1'])
```

TASK 9: Real-Time Application Challenge – Choose the Right Data Structure

PROMPT:

Implement a Cafeteria Order Queue system in Python using an appropriate Queue data structure that supports adding orders, serving orders in FIFO order, and displaying pending orders, with proper comments, docstrings, and example usage.

CODE:

```
class CafeteriaOrderQueue:
    """A simple implementation of a cafeteria order queue system using a list."""

    def __init__(self):
        """Initialize an empty order queue."""
        self.orders = []
```

```
    def add_order(self, order):
        """Add an order to the queue.

        Args:
            order: The order to be added to the queue.
        """
        self.orders.append(order)

    def serve_order(self):
        """Serve the next order in the queue (FIFO).

        Returns:
            The next order if the queue is not empty; otherwise, raises an
exception.
        """
        if not self.orders:
            raise IndexError("No orders to serve")
        return self.orders.pop(0)

    def display_pending_orders(self):
        """Display all pending orders in the queue."""
        if not self.orders:
            print("No pending orders.")
        else:
            print("Pending Orders:")
            for order in self.orders:
                print(order)
# Example usage
if __name__ == "__main__":
    cafeteria_queue = CafeteriaOrderQueue()
    cafeteria_queue.add_order("Order 1: Coffee")
    cafeteria_queue.add_order("Order 2: Sandwich")
    cafeteria_queue.add_order("Order 3: Salad")

    cafeteria_queue.display_pending_orders()

    print("\nServing orders:")
    while True:
        try:
            print(cafeteria_queue.serve_order())
        except IndexError:
            print("All orders have been served.")
            break
```

```
cafeteria_queue.display_pending_orders()
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop
s/akshi/OneDrive/Desktop/11.1.py
Pending Orders:
Order 1: Coffee
Order 2: Sandwich
Order 3: Salad

Serving orders:
Order 1: Coffee
Order 2: Sandwich
Order 3: Salad
All orders have been served.
No pending orders.
```

JUSTIFICATION:

Feature	Chosen Data Structure	Justification
Student Attendance Tracking	**Hash Table**	A hash table allows fast O(1) average-time insertion and lookup of student records using student ID as the key. Since attendance needs quick entry and exit logging per student, direct access via hashing is efficient and scalable.
Event Registration System	**Binary Search Tree (BST)**	A BST enables efficient searching, insertion, and deletion in O(log n) time (average case). Since events require quick participant search and removal, maintaining sorted participant records helps in organized management.
Library Book Borrowing	**Hash Table**	Books can be stored using book ID as the key for fast lookup, availability checking, and due-date tracking. Hash tables allow efficient updates when books are borrowed or returned.
Bus Scheduling System	**Graph**	A graph naturally represents bus routes and stops as vertices and connections as edges. It is ideal

		for modeling routes and finding paths between stops.
Cafeteria Order Queue	**Queue**	A queue follows FIFO (First In First Out), which perfectly matches serving students in the order they arrive. It ensures fair and orderly processing of cafeteria orders.

TASK 10: Smart E-Commerce Platform – Data Structure Challenge

PROMPT:

Implement a Product Search Engine in Python using a Hash Table that supports adding products, searching by product ID, deleting products, and displaying all products, with proper comments, docstrings, and example usage.

CODE:

```
class ProductSearchEngine:
    """A simple implementation of a product search engine using a hash table."""

    def __init__(self):
        """Initialize an empty product search engine."""
        self.products = {}

    def add_product(self, product_id, product_name):
        """Add a product to the search engine.

        Args:
            product_id: The unique identifier for the product.
            product_name: The name of the product.
        """
        self.products[product_id] = product_name

    def search_product(self, product_id):
        """Search for a product by its ID.

        Args:
            product_id: The unique identifier for the product to be searched.

        Returns:
            The name of the product if found; otherwise, None.
        """
        return self.products.get(product_id)
```

```
    def delete_product(self, product_id):
        """Delete a product from the search engine by its ID.

        Args:
            product_id: The unique identifier for the product to be deleted.
        """
        if product_id in self.products:
            del self.products[product_id]

    def display_products(self):
        """Display all products in the search engine."""
        if not self.products:
            print("No products available.")
        else:
            print("Products:")
            for product_id, product_name in self.products.items():
                print(f"ID: {product_id}, Name: {product_name}")
# Example usage
if __name__ == "__main__":
    search_engine = ProductSearchEngine()
    search_engine.add_product(1, "Laptop")
    search_engine.add_product(2, "Smartphone")
    search_engine.add_product(3, "Headphones")

    search_engine.display_products()

    print("\nSearching for product with ID 2:")
    print(search_engine.search_product(2))  # Output: Smartphone

    print("\nDeleting product with ID 1.")
    search_engine.delete_product(1)

    print("\nProducts after deletion:")
    search_engine.display_products()
```

OUTPUT:

```
PS C:\Users\akshi\OneDrive\Desktop\fu
s/akshi/OneDrive/Desktop/11.1.py
Products:
ID: 1, Name: Laptop
ID: 2, Name: Smartphone
ID: 3, Name: Headphones

Searching for product with ID 2:
Smartphone

Deleting product with ID 1.

Products after deletion:
Products:
ID: 2, Name: Smartphone
ID: 3, Name: Headphones
```

JUSTIFICATION:

Feature	Chosen Data Structure	Justification
Shopping Cart Management	Linked List	Allows dynamic addition and removal of products efficiently without shifting elements.
Order Processing System	Queue	Processes orders in FIFO order ensuring fairness and correct sequence.
Top-Selling Products Tracker	Priority Queue	Efficiently maintains products ranked by highest sales count.
Product Search Engine	Hash Table	Provides fast O(1) average-time lookup using product ID.
Delivery Route Planning	Graph	Represents warehouses and delivery locations as nodes and routes as edges for efficient path management.