LAB ASSIGNMENT – 11

NAME : M.SHIVA

HALL.NO : 2403A52377

BATCH.NO : AI 14

Prompt 01:

Write a simple Python Stack class with push, pop, peek, and is_empty methods. Add short comments and docstrings.

Code:

```
class Stack:
    """A simple Stack class implementation."""

    def __init__(self):
        """Initializes an empty stack."""
        self._items = []  # Internal list to store stack elements

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

        Args:
            item: The item to be added.
        """
        self._items.append(item)

    def pop(self):
        """Removes and returns the item from the top of the stack.

        Returns:
            The item removed from the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self.is_empty():
            return self._items.pop()
        else:
            raise IndexError("pop from empty stack")
```

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

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if not self.is_empty():
            return self._items[-1]
        else:
            raise IndexError("peek from empty stack")

    def is_empty(self):
        """Checks if the stack is empty.

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

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

Code explanation:

- class Stack: This line defines a new class named Stack.
- "A simple Stack class implementation."""This is a docstring that explains the purpose of the class.
- def __init__(self):: This is the constructor method. It's called when you create a new Stack object.
    - self._items = []: Inside the constructor, an empty list named _items is initialized. This list will be used internally to store the elements of the stack. The underscore prefix (_) is a convention to indicate that this is an internal detail of the class.
- def push(self, item): This method adds an item to the top of the stack.
    - self._items.append(item): The append() method of the list adds the item to the end of the _items list, which is considered the "top" of the stack in this implementation.
- def pop(self):: This method removes and returns the item from the top of the stack.
    - if not self.is_empty():: It first checks if the stack is not empty using the is_empty() method.

    - return self._items.pop(): If the stack is not empty, the pop() method of the list is used to remove and return the last element (the top of the stack).
    - else: raise IndexError("pop from empty stack"): If the stack is empty, it raises an IndexError with a descriptive message.
- def peek(self):: This method returns the item at the top of the stack without removing it.
    - if not self.is_empty():: It also checks if the stack is not empty.
    - return self._items[-1]: If not empty, it returns the last element of the list (_items[-1]) using negative indexing, which accesses the last element.
    - else: raise IndexError("peek from empty stack"): If empty, it raises an IndexError.
- def is_empty(self): This method checks if the stack is empty.
    - return len(self._items) == 0: It returns True if the length of the _items list is 0, and False otherwise.
- def size(self):: This method returns the number of items in the stack.
    - return len(self._items): It simply returns the length of the _items list.

Performing with example :

```
# Try to pop from an empty stack (will raise an error)
print("\nTrying to pop from an empty stack:")
try:
    my_stack.pop()
except IndexError as e:
    print(f"Caught expected error: {e}")
```

```
Pushing elements onto the stack:
Stack after pushing 10: [10]
Stack after pushing 20: [10, 20]
Stack after pushing 30: [10, 20, 30]

Peeking at the top element:
Top element: 30
Stack after peeking: [10, 20, 30]

Popping elements from the stack:
Popped element: 30
Stack after popping: [10, 20]
Popped element: 20
Stack after popping: [10]

Checking if the stack is empty:
Is stack empty? False
Popped element: 10
Stack after popping: []
Is stack empty? True

Trying to pop from an empty stack:
Caught expected error: pop from empty stack
```

```
# Create a new stack
my_stack = Stack()

# Push elements onto the stack
print("Pushing elements onto the stack:")
my_stack.push(10)
print(f"Stack after pushing 10: {my_stack._items}")
my_stack.push(20)
print(f"Stack after pushing 20: {my_stack._items}")
my_stack.push(30)
print(f"Stack after pushing 30: {my_stack._items}")

# Peek at the top element
print("\nPeeking at the top element:")
print(f"Top element: {my_stack.peek()}")
print(f"Stack after peeking: {my_stack._items}") # Stack remains unchanged

# Pop elements from the stack
print("\nPopping elements from the stack:")
print(f"Popped element: {my_stack.pop()}")
print(f"Stack after popping: {my_stack._items}")
print(f"Popped element: {my_stack.pop()}")
print(f"Stack after popping: {my_stack._items}")

# Check if the stack is empty
print("\nChecking if the stack is empty:")
print(f"Is stack empty? {my_stack.is_empty()}")

# Pop the last element
print(f"Popped element: {my_stack.pop()}")
print(f"Stack after popping: {my_stack._items}")

# Check if the stack is empty again
print(f"Is stack empty? {my_stack.is_empty()}")
```

Prompt 02:

Create a Python Queue class using lists with enqueue, dequeue, and is_empty methods. Then suggest a faster version using collections. Deque.

Code:

```python
class Queue:
    """A simple Queue class implementation using a list."""

    def __init__(self):
        """Initializes an empty queue."""
        self._items = []  # Internal list to store queue elements

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

        Args:
            item: The item to be added.
        """
        self._items.append(item)

    def dequeue(self):
        """Removes and returns the item from the front of the queue.

        Returns:
            The item removed from the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if not self.is_empty():
            # Removing from the front of a list is O(n)
            return self._items.pop(0)
        else:
            raise IndexError("dequeue from empty queue")
```

```python
    def is_empty(self):
        """Checks if the queue is empty.

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

# Suggestion for a faster Queue implementation
```

Code explanation:

class Queue:: This line defines a new class named Queue.

- **"""A simple Queue class implementation using a list."""**: This is a docstring explaining the class's purpose and implementation detail (using a list).

- **def __init__(self):**: This is the constructor.
    - **self._items = []**: An empty list _items is initialized to hold the queue elements.
- **def enqueue(self, item):**: This method adds an item to the back (rear) of the queue.
    - **self._items.append(item)**: The append() method is used to add the item to the end of the list. This is an efficient operation (O(1) on average).
- **def dequeue(self):**: This method removes and returns the item from the front of the queue.
    - **if not self.is_empty():**: It checks if the queue is not empty.
    - **return self._items.pop(0)**: If not empty, pop(0) is used to remove and return the element at index 0 (the front of the list). **However, this operation can be inefficient (O(n))** because all subsequent elements need to be shifted to the left to fill the gap.
    - **else: raise IndexError("dequeue from empty queue")**: If the queue is empty, it raises an IndexError.
- **def is_empty(self):**: This method checks if the queue is empty by checking if the length of the _items list is 0.

Performing with example :

```
# Create a new queue
my_queue = Queue()

# Enqueue elements onto the queue
print("Enqueuing elements onto the queue:")
my_queue.enqueue(10)
print(f"Queue after enqueuing 10: {my_queue._items}")
my_queue.enqueue(20)
print(f"Queue after enqueuing 20: {my_queue._items}")
my_queue.enqueue(30)
print(f"Queue after enqueuing 30: {my_queue._items}")

# Check if the queue is empty
print("\nChecking if the queue is empty:")
print(f"Is queue empty? {my_queue.is_empty()}")

# Dequeue elements from the queue
print("\nDequeuing elements from the queue:")
print(f"Dequeued element: {my_queue.dequeue()}")
print(f"Queue after dequeuing: {my_queue._items}")
print(f"Dequeued element: {my_queue.dequeue()}")
print(f"Queue after dequeuing: {my_queue._items}")

# Check if the queue is empty again
print(f"Is queue empty? {my_queue.is_empty()}")

# Dequeue the last element
print(f"Dequeued element: {my_queue.dequeue()}")
print(f"Queue after dequeuing: {my_queue._items}")

# Check if the queue is empty one more time
print(f"Is queue empty? {my_queue.is_empty()}")
```

```
# Check if the queue is empty one more time
print(f"Is queue empty? {my_queue.is_empty()}")


# Try to dequeue from an empty queue (will raise an error)
print("\nTrying to dequeue from an empty queue:")
try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Caught expected error: {e}")
```

```
Enqueuing elements onto the queue:
Queue after enqueuing 10: [10]
Queue after enqueuing 20: [10, 20]
Queue after enqueuing 30: [10, 20, 30]

Checking if the queue is empty:
Is queue empty? False

Dequeuing elements from the queue:
Dequeued element: 10
Queue after dequeuing: [20, 30]
Dequeued element: 20
Queue after dequeuing: [30]
Is queue empty? False
Dequeued element: 30
Queue after dequeuing: []
Is queue empty? True

Trying to dequeue from an empty queue:
Caught expected error: dequeue from empty queue
```

Prompt 03:

Make a Python singly linked list with insert_at_end, delete_value, and traverse methods. Add comments to explain pointer changes.

Code:

```
        def __init__(self, data):
            self.data = data  # Data stored in the node
            self.next = None  # Pointer to the next node

    class SinglyLinkedList:
        """A simple singly linked list implementation."""
        def __init__(self):
            self.head = None  # Head of the list (initially None)

        def insert_at_end(self, data):
            """Inserts a new node with the given data at the end of the list."""
            new_node = Node(data)
            if not self.head:
                self.head = new_node  # If the list is empty, the new node becomes the head
                return

            last_node = self.head
            while last_node.next:
                last_node = last_node.next  # Traverse to the last node
            last_node.next = new_node  # Point the current last node's next to the new node

        def delete_value(self, value):
            """Deletes the first node with the given value from the list."""
            if not self.head:
                return  # Empty list, nothing to delete

            if self.head.data == value:
                self.head = self.head.next  # If the head contains the value, update the head
                return

            current_node = self.head
            while current_node.next and current_node.next.data != value:
                current_node = current_node.next  # Traverse until the next node has the value or end of list

            if current_node.next:
                current_node.next = current_node.next.next  # Skip the node to be deleted (update pointer)
```

```
def traverse(self):
    """Traverses the list and prints the data of each node."""
    current_node = self.head
    while current_node:
        print(current_node.data, end=" -> ")
        current_node = current_node.next  # Move to the next node
    print("None")
```

Code explanation:

The Node Class:

- class Node:: This defines the building block of the linked list. Each Node object will hold a piece of data and a reference to the next node in the sequence.
- def __init__(self, data):: This is the constructor for the Node.
    - self.data = data: This line stores the actual data that the node will hold.
    - self.next = None: This is the crucial part for linking. self.next is a pointer (or reference) to the next node in the list. When a node is initially created, it doesn't point to anything yet, so it's set to None.

2. The SinglyLinkedList Class:

- class SinglyLinkedList:: This class represents the entire linked list. It manages the sequence of nodes.
- def __init__(self):: This is the constructor for the SinglyLinkedList.
    - self.head = None: This initializes the head of the list to None. The head is a pointer to the very first node in the list. An empty list has no head.
- def insert_at_end(self, data):: This method adds a new node with the given data to the end of the list.
    - new_node = Node(data): A new Node object is created with the provided data.
    - if not self.head:: This checks if the list is currently empty.
        - self.head = new_node: If the list is empty, the new node becomes the head.
        - return: The method finishes here if the list was empty.
    - last_node = self.head: If the list is not empty, a temporary pointer last_node is set to the current head.
    - while last_node.next:: This loop traverses the list until last_node points to the last node (the one whose next pointer is None).
        - last_node = last_node.next: The last_node pointer moves to the next node in each iteration.

    - last_node.next = new_node: Once the loop finishes, last_node is at the end. Its next pointer is updated to point to the new_node, effectively adding the new node to the end of the list.

- def delete_value(self, value):: This method deletes the first node that contains the given value.
    - if not self.head:: Checks if the list is empty. If so, there's nothing to delete.
    - if self.head.data == value:: Checks if the head node contains the value to be deleted.
        - self.head = self.head.next: If the head contains the value, the head pointer is updated to point to the next node, effectively removing the original head.
        - return: The method finishes after deleting the head.
    - current_node = self.head: If the value is not in the head, a current_node pointer is initialized to the head.
    - while current_node.next and current_node.next.data != value:: This loop traverses the list until either the next node exists and its data matches the value, or the end of the list is reached.
        - current_node = current_node.next: The current_node pointer moves to the next node.
    - if current_node.next:: After the loop, this checks if a node with the value was found (i.e., current_node.next is not None).
        - current_node.next = current_node.next.next: If the node was found, the next pointer of the node before the one to be deleted (current_node) is updated to point to the node after the one to be deleted (current_node.next.next). This bypasses the node to be deleted, removing it from the list's sequence.
- def traverse(self):: This method goes through the list from beginning to end and prints the data of each node.
    - current_node = self.head: A current_node pointer starts at the head.
    - while current_node:: This loop continues as long as current_node is not None (meaning there are still nodes to visit).

- print(current_node.data, end=" -> "): The data of the current node is printed, followed by " -> ". end=" -> " prevents a newline after each print, keeping the output on one line.
- current_node = current_node.next: The current_node pointer moves to the next node.

- print("None"): After the loop finishes (when current_node becomes None), "None" is printed to indicate the end of the list.

Performing with example:

```
# Create a new singly linked list
my_list = SinglyLinkedList()

# Insert elements at the end
print("Inserting elements at the end:")
my_list.insert_at_end(10)
my_list.traverse()
my_list.insert_at_end(20)
my_list.traverse()
my_list.insert_at_end(30)
my_list.traverse()
my_list.insert_at_end(40)
my_list.traverse()

# Delete a value from the list
print("\nDeleting value 20:")
my_list.delete_value(20)
my_list.traverse()

print("\nDeleting value 10 (the head):")
my_list.delete_value(10)
my_list.traverse()

print("\nDeleting value 40 (the last node):")
my_list.delete_value(40)
my_list.traverse()

print("\nTrying to delete a value that doesn't exist (99):")
my_list.delete_value(99)
my_list.traverse()

print("\nDeleting the last remaining value (30):")
my_list.delete_value(30)
my_list.traverse()
```

```
print("\nTrying to delete from an empty list:")
my_list.delete_value(50)
my_list.traverse()
```

```
Inserting elements at the end:
10 -> None
10 -> 20 -> None
10 -> 20 -> 30 -> None
10 -> 20 -> 30 -> 40 -> None

Deleting value 20:
10 -> 30 -> 40 -> None

Deleting value 10 (the head):
30 -> 40 -> None

Deleting value 40 (the last node):
30 -> None

Trying to delete a value that doesn't exist (99):
30 -> None

Deleting the last remaining value (30):
None

Trying to delete from an empty list:
None
```

Prompt 04:

Write a Python BST with insert, search, and inorder_traversal methods. Add comments for recursion and traversal.

Code&Output:

```python
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, current_node, key):
        # Recursive step for insertion
        if key < current_node.key:
            if current_node.left is None:
                current_node.left = Node(key)
            else:
                self._insert_recursive(current_node.left, key)
        else:
            if current_node.right is None:
                current_node.right = Node(key)
            else:
                self._insert_recursive(current_node.right, key)

    def search(self, key):
        return self._search_recursive(self.root, key)
```

```python
    def search(self, key):
        return self._search_recursive(self.root, key)

    def _search_recursive(self, current_node, key):
        # Recursive step for search
        if current_node is None or current_node.key == key:
            return current_node
        if key < current_node.key:
            return self._search_recursive(current_node.left, key)
        else:
            return self._search_recursive(current_node.right, key)

    def inorder_traversal(self):
        result = []
        self._inorder_recursive(self.root, result)
        return result

    def _inorder_recursive(self, current_node, result):
        # Recursive step for inorder traversal
        if current_node:
            self._inorder_recursive(current_node.left, result)
            result.append(current_node.key)
            self._inorder_recursive(current_node.right, result)

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

print("Inorder traversal:", bst.inorder_traversal())
```

```python
print("Search for 40:", bst.search(40).key if bst.search(40) else "Not found")
print("Search for 90:", bst.search(90).key if bst.search(90) else "Not found")
```

```
Inorder traversal: [20, 30, 40, 50, 60, 70, 80]
Search for 40: 40
Search for 90: Not found
```

Code explanation:

1. **Node Class:** This class represents a single node within the BST. Each node has:
    - key: The value stored in the node.
    - left: A reference to the left child node.
    - right: A reference to the right child node.
2. **BST Class:** This class represents the entire Binary Search Tree.

- __init__(self): The constructor initializes an empty BST by setting the root to None.
- insert(self, key): This method inserts a new node with the given key into the BST. It handles the case of an empty tree and then calls a recursive helper method _insert_recursive.
- _insert_recursive(self, current_node, key): This is a recursive helper function for insertion. It compares the key to the current_node's key and moves down the left or right subtree until it finds an appropriate place to insert the new node.
- search(self, key): This method searches for a node with the given key in the BST. It calls a recursive helper method _search_recursive.
- _search_recursive(self, current_node, key): This is a recursive helper function for searching. It checks if the current_node is the target node or if the target is in the left or right subtree and recursively searches accordingly.
- inorder_traversal(self): This method performs an in-order traversal of the BST, which visits nodes in ascending order of their keys. It calls a recursive helper method _inorder_recursive.
- _inorder_recursive(self, current_node, result): This is a recursive helper function for in-order traversal. It visits the left subtree, then the current node, and then the right subtree, appending the node's key to the result list.

3. **Example Usage:** The code then demonstrates how to use the BST class by:
    - Creating a BST instance.
    - Inserting several values into the tree.
    - Performing an in-order traversal and printing the result.
    - Searching for existing and non-existing keys and printing the results.

Prompt 05:

Implement a Python graph using an adjacency list with BFS and DFS methods. Add short comments to explain how each traversal works.

Code& Output:

```python
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        # For an undirected graph, uncomment the line below
        # self.graph[v].append(u)

    def bfs(self, start_node):
        # BFS uses a queue to explore nodes level by level
        visited = set()
        queue = deque([start_node])
        visited.add(start_node)
        result = []

        while queue:
            node = queue.popleft()
            result.append(node)

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        return result

    def dfs(self, start_node):
        # DFS uses recursion (or a stack) to explore as deep as possible along each branch
        visited = set()
        result = []
```

```python
        def _dfs_recursive(node):
            visited.add(node)
            result.append(node)

            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    _dfs_recursive(neighbor)

        _dfs_recursive(start_node)
        return result

# Example Usage:
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

print("BFS starting from node 2:", g.bfs(2))
print("DFS starting from node 2:", g.dfs(2))
```

```
BFS starting from node 2: [2, 0, 3, 1]
DFS starting from node 2: [2, 0, 1, 3]
```

Code explanation:

1. **Graph Class:**

- __init__(self): The constructor initializes an empty graph using defaultdict(list). This creates a dictionary where keys are nodes and values are lists of their neighbors. If a node is accessed that isn't already in the dictionary, a new entry is created with an empty list as its value.
- add_edge(self, u, v): This method adds a directed edge from node u to node v. It appends v to the list of neighbors for u. The commented-out line self.graph[v].append(u) would be included for an undirected graph to add the edge in both directions.

2. **bfs(self, start_node):**
    - This method performs a Breadth-First Search starting from the start_node.
    - It uses a deque (double-ended queue) to manage the nodes to visit. BFS explores the graph level by level.
    - visited is a set to keep track of nodes that have already been visited to avoid infinite loops.
    - The while queue: loop continues as long as there are nodes to visit.
    - queue.popleft() removes the node at the beginning of the queue (FIFO - First-In, First-Out).
    - It then iterates through the neighbors of the current node and adds unvisited neighbors to the visited set and the queue.
    - The result list stores the nodes in the order they are visited during the traversal.
3. **dfs(self, start_node):**
    - This method performs a Depth-First Search starting from the start_node.
    - It uses recursion (implicitly through function calls) to explore as deep as possible along each branch before backtracking.
    - visited is a set to keep track of visited nodes.
    - The inner function _dfs_recursive(node) is the recursive helper.
    - It marks the current node as visited and adds it to the result list.
    - Then, it iterates through the neighbors of the current node. If a neighbor hasn't been visited, it recursively calls _dfs_recursive on that neighbor.

4. **Example Usage:** The code demonstrates how to create a Graph instance, add edges to define the graph structure, and then perform BFS and DFS starting from a specific node (node 2 in this case), printing the order of visited nodes for each traversal.