ASSIGNMENT – 11.2

Name: CH.Harshini

Roll Number: 2303A51185

Batch - 03

Task 1:-

```python
assignment 11.py > Stack > __init__

class Stack:
    """
    A Stack data structure that follows Last-In-First-Out (LIFO) principle.

    This implementation uses a Python list as the underlying data structure.
    Supports basic stack operations: push, pop, peek, and empty check.

    Attributes:
        items (list): Internal list to store stack elements
    """

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

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

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

    def pop(self):
        """
        Remove and return the top item from the stack.

        Returns:
            The top item from the stack

        Raises:
            IndexError: If the stack is empty
        """
        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.

        Returns:
            The top item from the stack

        Raises:
            IndexError: If the stack is empty
        """
        if self.is_empty():
            raise IndexError("Peek from empty stack")
```

Output:-

```
Data Structures Implementation - Lab 11
==================================================

1. Stack Operations:
Stack after pushes: size = 3
Peek: 30
Pop: 30
Pop: 20
Is empty: False
```

Task 1: Stack Implementation

The Stack was implemented to demonstrate the **Last-In-First-Out (LIFO)** principle, which is fundamental in computer science. Stacks are widely used in:

- Expression evaluation

- Undo/Redo operations
- Function call management (call stack)
- Backtracking algorithms

This implementation uses a Python list for simplicity and efficiency. Proper error handling (raising `IndexError`) ensures robustness. The object-oriented design improves modularity, readability, and reusability of the code.

Task 2:

```python
class Queue:
    """
    A Queue data structure that follows First-In-First-Out (FIFO) principle.

    This implementation uses a Python list as the underlying data structure.
    Supports basic queue operations: enqueue, dequeue, front access, and size.

    Attributes:
        items (list): Internal list to store queue elements
    """

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

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

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

    def dequeue(self):
        """
        Remove and return the front item from the queue.

        Returns:
            The front item from the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)
```

```python
    def front(self):
        """
        Return the front item from the queue without removing it.

        Returns:
            The front item from the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
            raise IndexError("Front from empty queue")
        return self.items[0]

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

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

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

        Returns:
            int: Number of items in the queue
        """
        return len(self.items)
```

Output:-

```
2. Queue Operations:
Queue after enqueues: size = 3
Front: 10
Dequeue: 10
Dequeue: 20
Is empty: False
```

Task 2: Queue Implementation

The Queue was implemented to demonstrate the **First-In-First-Out (FIFO)** principle. Queues are essential in:

- CPU scheduling
- Breadth-First Search (BFS)
- Printer task scheduling
- Real-time systems

Using a Python list makes the implementation easy to understand. The structure includes enqueue, dequeue, front, and size operations with proper boundary checks to prevent runtime errors.

Task 3:-

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

    Attributes:
        data: The data stored in the node
        next: Reference to the next node in the list
    """

    def __init__(self, data):
        """
        Initialize a node with given data.

        Args:
            data: The data to store in the node
        """
        self.data = data
        self.next = None

class SinglyLinkedList:
    """
    A singly linked list data structure.

    Supports insertion at the beginning and traversal/display operations.

    Attributes:
        head: Reference to the first node in the list
    """

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

    def insert_at_beginning(self, data):
        """
        Insert a new node with given data at the beginning of the list.

        Args:
            data: The data to insert
        """
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node
```

```python
def display(self):
    """
    Display all elements in the linked list.

    Prints the elements separated by ' -> ' and ends with 'None'
    """
    current = self.head
    while current:
        print(current.data, end=" -> ")
        current = current.next
    print("None")
```

Output:-

```
Is empty: False

3. Singly Linked List Operations:
Linked List after insertions:
10 -> 20 -> 30 -> None
```

Task 3: Singly Linked List Implementation

The Singly Linked List demonstrates dynamic memory allocation and non-contiguous data storage. Unlike arrays, linked lists allow efficient insertion and deletion operations.

This implementation helps in understanding:

- Pointer/reference-based data structures
- Dynamic memory usage
- Node-based traversal

It provides insertion at the beginning and traversal functionality, forming the foundation for more complex structures like doubly linked lists and circular linked lists.

Task 4:

```python
class BSTNode:
    Attributes:
        data: The data stored in the node
        left: Reference to the left child node
        right: Reference to the right child node
    """

    def __init__(self, data):
        """
        Initialize a BST node with given data.

        Args:
            data: The data to store in the node (must be comparable)
        """
        self.data = data
        self.left = None
        self.right = None

class BinarySearchTree:
    """
    A Binary Search Tree data structure.

    Supports insertion and in-order traversal operations.

    Attributes:
        root: Reference to the root node of the tree
    """

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

    def insert(self, data):
        """
        Insert a new node with given data into the BST.

        Args:
            data: The data to insert (must be comparable)
        """
        if self.root is None:
```

```python
        """
        if data < node.data:
            if node.left is None:
                node.left = BSTNode(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = BSTNode(data)
            else:
                self._insert_recursive(node.right, data)

    def in_order_traversal(self):
        """
        Perform in-order traversal of the BST and print the elements.

        In-order traversal visits left subtree, current node, right subtree.
        """
        self._in_order_recursive(self.root)
        print()  # New line after traversal

    def _in_order_recursive(self, node):
        """
        Helper method for recursive in-order traversal.

        Args:
            node: Current node in the recursion
        """
        if node:
            self._in_order_recursive(node.left)
            print(node.data, end=" ")
            self._in_order_recursive(node.right)
```

Output:

```
4. Binary Search Tree Operations:
BST after insertions - In-order traversal:
20 30 40 50 60 70 80
```

Task 4: Binary Search Tree (BST) Implementation

The Binary Search Tree was implemented to demonstrate hierarchical data storage and efficient searching.

BST provides:

- Average-case search complexity: **O(log n)**
- Ordered data storage
- Efficient insertion and traversal

In-order traversal prints elements in sorted order, demonstrating the correctness of BST properties. Recursive implementation improves clarity and reflects real-world tree algorithms.

Task 5:-

```python
class HashTable:
    """
    A Hash Table data structure with collision handling using chaining.

    Uses separate chaining (linked lists) to handle collisions.
    Supports insert, search, and delete operations.

    Attributes:
        size (int): The size of the hash table
        table (list): Internal list of buckets (each bucket is a list)
    """

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

        Args:
            size (int): The number of buckets in the hash table
        """
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """
        Compute the hash value for a given key.

        Args:
            key: The key to hash

        Returns:
            int: The hash value (bucket index)
        """
        return hash(key) % self.size

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

        If the key already exists, updates the value.

        Args:
            key: The key to insert
```

```python
        # Check if key already exists
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)  # Update value
                return

        # Key doesn't exist, add new entry
        bucket.append((key, value))

    def search(self, key):
        """
        Search for a value by its key in the hash table.

        Args:
            key: The key to search for

        Returns:
            The value associated with the key, or None if not found
        """
        bucket_index = self._hash(key)
        bucket = self.table[bucket_index]

        for k, v in bucket:
            if k == key:
                return v

        return None

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

        Args:
            key: The key to delete

        Returns:
            bool: True if key was found and deleted, False otherwise
        """
        bucket_index = self._hash(key)
        bucket = self.table[bucket_index]

        for i, (k, v) in enumerate(bucket):
```

Output:-

```
5. Hash Table Operations:
Hash Table after insertions:
Bucket 0: [('cherry', 30)]
Bucket 1: []
Bucket 2: [('apple', 10), ('banana', 20)]
Bucket 3: []
Bucket 4: []
Bucket 5: []
Bucket 6: [('date', 40)]
Bucket 7: []
Bucket 8: []
Bucket 9: []
Search 'apple': 10
Search 'grape': None
Delete 'banana': True
Hash Table after deletion:
Bucket 0: [('cherry', 30)]
Bucket 1: []
Bucket 2: [('apple', 10)]
Bucket 3: []
Bucket 4: []
Bucket 5: []
Bucket 6: [('date', 40)]
Bucket 7: []
Bucket 8: []
Bucket 9: []
PS C:\Users\hp\OneDrive\Desktop\ai>
```

Task 5: Hash Table Implementation

The Hash Table demonstrates fast data retrieval using hashing.

Key advantages:

- Average-case time complexity: **O(1)** for insert, search, delete
- Efficient key-value mapping
- Collision handling using separate chaining

This implementation introduces hashing concepts and collision resolution techniques, which are fundamental in database indexing, caching systems, and symbol tables.

Extra question:

```python

SIZE = 5
stack = []
top = -1

def push(value):
    global top
    if top == SIZE - 1:
        print("Stack Overflow")
    else:
        stack.append(value)
        top += 1
        print(value, "inserted")

def pop():
    global top
    if top == -1:
        print("Stack Underflow")
    else:
        removed = stack.pop()
        print(removed, "removed")
        top -= 1

# Testing the additional stack implementation
print("\n6. Additional Stack Implementation (Array-based):")

# Inserting elements
push('a')
push('b')
push('c')
push('d')
push('e')
push('f')   # This should show Overflow

print()

# Removing elements
pop()
pop()
pop()
pop()
pop()
pop()       # This should show Underflow
```

Output:

```
6. Additional Stack Implementation (Array-based):
a inserted
b inserted
c inserted
d inserted
e inserted
Stack Overflow

e removed
d removed
c removed
b removed
a removed
Stack Underflow
```

This version demonstrates stack implementation using a fixed-size array and global variables. It highlights:

- Stack overflow condition
- Stack underflow condition
- Manual top pointer management

This helps compare structured (OOP-based) and procedural approaches, improving conceptual clarity.