AI ASSISTED CODING LAB

ASSIGNMENT 11.4

ENROLLMENT NO :2503A51L47

BATCH NO: 20

NAME: N.AKASH

TASK1

TASK1 DESCRIPTION:-

Use AI to help implement a **Stack** class in Python with the following operations: push (), pop (), peek (), and is empty ().

Instructions:

- o Ask AI to generate code skeleton with docstrings.
- o Test stack operations using sample data.
- o Request AI to suggest optimizations or alternative implementations (e.g., using collections. Deque).

PROMPT:-

Implement a Stack class with push(item), pop(), peek(), and is_empty(); include Google-style docstrings, inline comments for tricky parts, and a short main that tests pushing 1,2,3 then peeking and popping to show correct LIFO behavior.

CODE:-

t1.py > Stack > push

```python
class Stack:
    """A simple Stack (LIFO) implementation.

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

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

    def push(self, item):
        """Pushes an item onto the stack.

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

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

        Returns:
            The item at the top of 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):
        """Returns the top item of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

```

t1.py > Stack > push

```python
class Stack:
    def pop(self):
        Returns:
            The item at the top of 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):
        """Returns the top item of the stack without removing it.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]

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

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

# Sample usage and test
if __name__ == "__main__":
    stack = Stack()
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.peek())    # Should print 3
    print(stack.pop())     # Should print 3
    print(stack.pop())     # Should print 2
    print(stack.is_empty())# Should print False
    print(stack.pop())     # Should print 1
    print(stack.is_empty())# Should print True
```

OUTPUT:-

```
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11> & C:/Users/khaja/anaconda3/python.exe "c:/Users/khaja/OneDrive/Pictures/Screenshots/cyc/New folder/AI_11/t1.py"
3
3
2
False
1
True
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11>
```

OBSERVATION:-

The Stack implementation provides a simple LIFO container using a Python list with push, pop, peek, and is_empty methods. Pushing and popping at the list end are O(1) amortized, and peek/is_empty are O(1); pop/peek raise IndexError on empty stacks for explicit error handling. The design is minimal and easy to test, suitable for most single-threaded uses; for thread-safety or alternate performance characteristics consider synchronization or other container types.

TASK2

TASK2 DESCRIPTION:-

Implement a **Queue** with enqueue (), dequeue (), and is empty () methods.

- **Instructions**:
  - First, implement using Python lists.
  - Then, ask AI to review performance and suggest a more efficient implementation (using collections. Deque).

PROMPT :-

Create ListQueue (using list) and DequeQueue (using collections.deque) with enqueue(item), dequeue(), and is_empty(); add docstrings, note performance differences (O(n) vs O(1)), and include sample code comparing both on a small sequence.

CODE:-

t2.py > DequeQueue

```python
class ListQueue: ···

# Optimized version using collections.deque
from collections import deque

class DequeQueue: ···

# Performance Comparison:
# - ListQueue: enqueue is O(1), dequeue is O(n) (slow for large queues).
# - DequeQueue: both enqueue and dequeue are O(1) (fast for large queues).

if __name__ == "__main__":
    # Example for ListQueue
    lq = ListQueue()
    for v in [1, 2, 3]: ···

    out_lq = []
    while not lq.is_empty():
        out_lq.append(lq.dequeue())
    assert out_lq == [1, 2, 3]
    print("ListQueue dequeued:", out_lq)

    # Example for DequeQueue
    dq = DequeQueue()
    for v in [1, 2, 3]:
        dq.enqueue(v)

    out_dq = []
    while not dq.is_empty():
        out_dq.append(dq.dequeue())
    assert out_dq == [1, 2, 3]
    print("DequeQueue dequeued:", out_dq)

    # Demonstrate empty-dequeue behavior
    try:
        lq.dequeue()
    except IndexError as e:
        print("ListQueue empty dequeue raised:", e)

    try:
        dq.dequeue()
    except IndexError as e:
        print("DequeQueue empty dequeue raised:", e)

    print("All queue examples passed.")
```

OUTPUT:-

```
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11> & C:/Users/khaja/anaconda3/python.exe
/Screenshots/cyc/New folder/AI_11/t2.py"
ListQueue dequeued: [1, 2, 3]
DequeQueue dequeued: [1, 2, 3]
ListQueue empty dequeue raised: dequeue from empty queue
DequeQueue empty dequeue raised: dequeue from empty queue
All queue examples passed.
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11>
```

OBSERVATION:-

Two queue variants are provided: ListQueue uses a list where enqueue is O(1) but dequeue (pop(0)) is O(n), while DequeQueue uses collections.deque giving O(1) enqueue and dequeue (append/popleft). The code and comments clearly demonstrate the performance trade-off and make deque the preferred choice for real queues or large workloads. Both implementations raise on empty dequeues, so tests should include empty-queue behavior.

TASK3

TASK3 DESCRIPTION:-

Implement a **Singly Linked List** with operations: instated (), delete value (), and traverse ().

- **Instructions**:
    - Start with a simple class-based implementation (Node, LinkedList).
    - Use AI to generate inline comments explaining pointer updates (which are non-trivial).
    - Ask AI to suggest test cases to validate all operations.

PROMPT:-

Implement a singly LinkedList with Node and LinkedList classes supporting insert(value), delete_value(value), and traverse(); include inline comments explaining pointer updates for head/middle/tail deletions, maintain tail and size, and add example tests for head, middle, tail, duplicate and absent deletions.

CODE:-

t3.py > ...

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

    Attributes:
        value: Stored data.
        next (Node|None): Reference to the next node.
    """

    def __init__(self, value):
        self.value = value
        self.next = None


class LinkedList:
    """Singly linked list with basic operations.

    Methods:
        insert(value): Append value to the end of the list.
        delete_value(value): Delete first occurrence of value, return True if deleted.
        traverse(): Return list of values (from head to tail).
    """

    def __init__(self):
        self.head = None
        self.tail = None  # keep tail for O(1) inserts at end
        self._size = 0    # maintain size for O(1) length queries

    def insert(self, value):
        """Append a value to the end of the list.

        Args:
            value: Value to append.
        """
        node = Node(value)
        if self.head is None:
            # empty list: head and tail both point to new node
            self.head = node
            self.tail = node
        else:
            # non-empty: attach new node after tail and update tail pointer
            self.tail.next = node  # old tail now points to new node
            self.tail = node       # move tail to the new last node
        self._size += 1

    def delete_value(self, value): ...
```

t3.py > ...

```python
def run_examples():
    assert ll.delete_value(2) is True
    assert ll.traverse() == [1, 3]
    assert len(ll) == 2

    # delete head
    assert ll.delete_value(1) is True
    assert ll.traverse() == [3]
    assert ll.head.value == 3
    assert ll.tail.value == 3  # single element => head == tail

    # delete tail (which is also head now)
    assert ll.delete_value(3) is True
    assert ll.traverse() == []
    assert ll.head is None and ll.tail is None
    assert len(ll) == 0

    # delete non-existent
    assert ll.delete_value(999) is False

    # insert duplicates and delete only first occurrence
    ll.insert("a")
    ll.insert("b")
    ll.insert("a")
    assert ll.traverse() == ["a", "b", "a"]
    assert ll.delete_value("a") is True
    assert ll.traverse() == ["b", "a"]
    assert len(ll) == 2

    # insert after deletions
    ll.insert("z")
    assert ll.traverse() == ["b", "a", "z"]
    assert ll.tail.value == "z"

    # iterate and repr checks
    collected = [x for x in ll]
    assert collected == ["b", "a", "z"]
    assert repr(ll) == "LinkedList(['b', 'a', 'z'])"

    print("All examples and assertions passed.")
    # print a small demonstration
    print("Final list:", ll)

if __name__ == "__main__":
    run_examples()
```

OUTPUT:-

```
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11> & C:/Users/khaja/anaconda3/python.exe
/Screenshots/cyc/New folder/AI_11/t3.py"
All examples and assertions passed.
Final list: LinkedList(['b', 'a', 'z'])
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11>
```

OBSERVATION:-

The linked list implements Node and LinkedList with head, tail, and a maintained size for O(1) append and O(1) length queries; delete_value scans O(n) to remove the first matching node. Inline comments explain pointer updates for deleting head, middle, and tail nodes and ensure tail and size are updated correctly, covering common edge cases (empty list, single element, duplicates). The API (traverse, **iter**, **len**, **repr**) improves testability and readability.

TASK4

TASK4 DESCRIPTION:-

Implement a **Binary Search Tree** with methods for insert (), search (), and inorder_traversal ().

- **Instructions**:
    - Provide AI with a partially written Node and BST class.
    - Ask AI to complete missing methods and add docstrings.

Test with a list of integers and compare outputs of search () for present vs absent elements.

PROMPT:-

Implement a BinarySearchTree with Node and BinarySearchTree classes providing insert(value), search(value), and inorder_traversal(); include docstrings, ignore duplicates, and add an example that inserts [7,3,9,1,5,8,10], asserts inorder == sorted(values), and checks search for present and absent keys.

CODE :-

```
t4.py > ...
class Node:
    """Node for a binary search tree.

    Attributes:
        value: Stored key.
        left (Node|None): Left child (keys < value).
        right (Node|None): Right child (keys > value).
    """

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

    def __repr__(self):
        return f"Node({self.value})"


class BinarySearchTree:...


def run_examples():
    """Example usage and simple tests for insert, search, and traversal."""
    values = [7, 3, 9, 1, 5, 8, 10]
    bst = BinarySearchTree()
    for v in values:
        bst.insert(v)

    # In-order should produce a sorted list
    inorder = bst.inorder_traversal()
    assert inorder == sorted(values), f"inorder {inorder} != sorted {sorted(values)}"

    # search present and absent elements
    assert bst.search(5) is True   # present
    assert bst.search(6) is False  # absent

    print("BST in-order traversal:", inorder)
    print("Search 5 ->", bst.search(5))
    print("Search 6 ->", bst.search(6))
    print("All example assertions passed.")

if __name__ == "__main__":
    run_examples()
```

OUTPUT:-

```
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11> & C:/Users/khaja/anaconda3/python.exe
/Screenshots/cyc/New folder/AI_11/t4.py"
BST in-order traversal: [1, 3, 5, 7, 8, 9, 10]
Search 5 -> True
Search 6 -> False
All example assertions passed.
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11>
```

OBSERVATION:-Binary Search Tree (BST)

The BST offers insert (ignoring duplicates), iterative search, and recursive inorder_traversal that returns sorted values. Typical complexities are O(log n) average for insert/search and O(n) worst-case for an unbalanced tree; inorder traversal is useful for verification. This simple BST is great for teaching and small datasets; for predictable logarithmic performance consider balanced variants (AVL or red-black trees) when needed.

TASK5

TASK5 DESCRIPTION:-

Implement a **Graph** using an adjacency list, with traversal methods BFS () and DFS ().

- **Instructions**:
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.
  - Compare recursive vs iterative DFS if suggested by AI.

PROMPT:-

Build a Graph using an adjacency-list dict with add_node/add_edge and traversal methods bfs(start), dfs_recursive(start), dfs_iterative(start); include inline comments about visited marking and queue/stack behavior, compare recursive vs iterative DFS ordering, and add an example graph plus assertions for BFS/DFS outputs.

CODE :-

t5.py > ...

```python
from collections import deque
from typing import Dict, List, Set, Any


class Graph:
    """Simple directed/undirected graph using an adjacency list.

    Attributes:
        adj (dict): Mapping node -> list of neighbor nodes.
        directed (bool): If False, add_edge will add both directions.
    """

    def __init__(self, directed: bool = False):
        self.adj: Dict[Any, List[Any]] = {}
        self.directed = directed

    def add_node(self, node: Any) -> None:
        """Ensure node exists in adjacency list."""
        if node not in self.adj:
            self.adj[node] = []

    def add_edge(self, u: Any, v: Any) -> None:
        """Add an edge u -> v. If undirected, also add v -> u.

        Inline notes:
        - For adjacency list we keep neighbors in a list; adding an edge
          appends the neighbor. Duplicate edges are not checked here.
        """
        self.add_node(u)
        self.add_node(v)
        self.adj[u].append(v)
        if not self.directed:
            # for undirected graphs add reverse link
            self.adj[v].append(u)

    def bfs(self, start: Any) -> List[Any]:
        """Breadth-first search from `start`. Returns list of visited nodes
        in BFS order.

        Implementation notes:
        - Uses deque as a queue (O(1) pops from left).
        - Mark nodes as visited when enqueued to avoid duplicate enqueues.
        """
        if start not in self.adj:
            return []
```

```python
t5.py > ...
class Graph:

    def __repr__(self):
        return f"Graph(nodes={list(self.adj.keys())})"


def run_examples():
    """Build a sample graph and show BFS/DFS outputs and simple assertions."""
    g = Graph(directed=False)
    # build a small graph:
    #     1
    #    / \
    #   2   3
    #   |    \
    #   4     5
    edges = [(1, 2), (1, 3), (2, 4), (3, 5)]
    for u, v in edges:
        g.add_edge(u, v)

    bfs_order = g.bfs(1)
    dfs_rec = g.dfs_recursive(1)
    dfs_it = g.dfs_iterative(1)

    print("Adjacency:", g.adj)
    print("BFS  from 1:", bfs_order)
    print("DFS (rec) from 1:", dfs_rec)
    print("DFS (it)  from 1:", dfs_it)

    # Basic checks:
    assert bfs_order == [1, 2, 3, 4, 5]
    # DFS orders may differ between recursive and iterative depending on neighbor order,
    # but both should be valid DFS traversals covering all reachable nodes starting at 1.
    assert set(dfs_rec) == {1, 2, 3, 4, 5}
    assert set(dfs_it) == {1, 2, 3, 4, 5}

    # Search absent start
    assert g.bfs(999) == []
    assert g.dfs_recursive(999) == []
    assert g.dfs_iterative(999) == []

    print("All example assertions passed.")


if __name__ == "__main__":
    run_examples()
```

OUTPUT:-

```
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11> & C:/Users/khaja/anaconda3/python.exe
/Screenshots/cyc/New folder/AI_11/t5.py"
Adjacency: {1: [2, 3], 2: [1, 4], 3: [1, 5], 4: [2], 5: [3]}
BFS  from 1: [1, 2, 3, 4, 5]
DFS (rec) from 1: [1, 2, 4, 3, 5]
DFS (it)  from 1: [1, 2, 4, 3, 5]
All example assertions passed.
PS C:\Users\khaja\OneDrive\Pictures\Screenshots\cyc\New folder\AI_11>
```

OBSERVATION:-Graph (adjacency list) with BFS/DFS

The Graph uses an adjacency-list dict and supports directed or undirected edges, with BFS (deque-based) and two DFS variants (recursive and iterative). Traversals run in O(V+E), BFS marks visited on enqueue to avoid duplicates, recursive DFS uses the call stack (risking recursion depth issues on deep graphs), and iterative DFS uses an explicit stack and can reverse neighbor order to match recursive visitation. The examples demonstrate expected traversal orders and handle absent-start cases; use iterative DFS or increase recursion limits for very large/deep graphs.