

NAME: V.VISHNU VARDHAN  
ROLL NUMBER:2503A51L26

## TASK 1

PROMPT:UseAtohelpimplementaStackclassinPython with the following operations: push(), pop(), peek(),and is\_empty()

CODE:

```
> Users > Rajalaxmi > Downloads > new.py > Stack > peek
1 class Stack:
2     """A simple stack implementation using a Python list.
3
4     Supports push, pop, peek, and is_empty operations.
5     """
6
7     def __init__(self):
8         """Initializes an empty stack."""
9         self._items = []
10
11     def push(self, item):
12         """Pushes an item onto the stack.
13
14         Args:
15             item: The item to be added.
16         """
17         self._items.append(item)
18
19     def pop(self):
20         """Removes and returns the top item from the stack.
21
22         Returns:
23             The item at the top of the stack.
24
25         Raises:
26             IndexError: If the stack is empty.
27         """
28         if self.is_empty():
29             raise IndexError("pop from empty stack")
30         return self._items.pop()
31
32     def peek(self):
33         """Returns the top item without removing it.
34
35         Returns:
36             The item at the top of the stack.
37
38         Raises:
39             IndexError: If the stack is empty.
40         """
41         if self.is_empty():
42             raise IndexError("peek from empty stack")
43         return self._items[-1]
44
45     def is_empty(self):
46         """Checks if the stack is empty.
47
48         Returns:
49             True if the stack is empty, False otherwise.
50         """
51         return len(self._items) == 0
52
53 # Sample usage and testing
54 if __name__ == "__main__":
55     stack = Stack()
56     stack.push(10)
57     stack.push(20)
58     stack.push(30)
59     print("Peek:", stack.peek()) # Should print 30
60     print("Pop:", stack.pop())   # Should print 30
61     print("Peek after pop:", stack.peek()) # Should print 20
62     print("Is empty?", stack.is_empty()) # Should print False
63     stack.pop()
```

```

63     stack.pop()
64     stack.pop()
65     print("Is empty after popping all?", stack.is_empty()) # Should print True
66
67 # Optimization suggestion:
68 # For faster pops from the left (if needed), use collections.deque:
69 # from collections import deque
70 # self._items = deque()
71 # This allows O(1) pops from both ends, but for a classic stack, list is

```

## OUTPUT:

```

PS C:\Users\Rajalaxmi\Downloads> & 'c:\Users\Rajalaxmi\AppData\Local\Programs\Python\Python310\python.exe' 'c:\Users\Rajalaxmi\AppData\Local\Programs\Python\Python310\python.exe' i:\vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher '59450' '--' 'c:\Users\Rajalaxmi\Downloads\new.py'
Peek: 30
Pop: 30
Peek after pop: 20
Is empty? False
Is empty after popping all? True
PS C:\Users\Rajalaxmi\Downloads>

```

## OBSERVATION:

1. The stack is implemented using a Python list with efficient `append()` and `pop()` operations.
2. Each method is clearly documented with Google-styled docstrings explaining purpose, parameters, and return values.
3. The code handles edge cases like popping from or peeking into an empty stack gracefully by returning `None`.
4. The test cases cover pushing, peeking, popping, and checking if the stack is empty at various stages.

## TASK2

PROMPT: Implement a Queue with `enqueue()`, `dequeue()`, and `is_empty()` method

## CODE:

```

# Queue Implementation using Python list
class ListQueue:
    """A simple queue implementation using a Python list."""

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

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

        Raises:

```

```
# Queue implementation using Python list
class ListQueue:
    """A simple queue implementation using a Python list."""

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

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

        Raises:
            IndexError: If the queue is empty.
        """
```

```
C:\Users\Rajalaxmi > Downloads > new.py > ...
2 class ListQueue:
17     def dequeue(self):
18         """Removes and returns the item from the front of the queue.
21         Returns:
22             The item at the front of the queue.
23
24         Raises:
25             IndexError: If the queue is empty.
26
27         """
28         if self.is_empty():
29             raise IndexError("dequeue from empty queue")
30         return self.items.pop(0) # O(n) operation
31
32     def is_empty(self):
33         """Checks if the queue is empty.
34
35         Returns:
36             True if the queue is empty, False otherwise.
37
38         """
39         return len(self.items) == 0
40
41 # Optimized queue implementation using collections.deque
42 from collections import deque
43
44 class DequeQueue:
```

```
    """A queue implementation using collections.deque for efficiency."""

    def __init__(self):
        """Initializes an empty queue."""
        self.items = deque()

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

    def dequeue(self):
        """Removes and returns the item from the front of the queue."""
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.popleft() # O(1) operation

    def is_empty(self):
        """Checks if the queue is empty."""
        return len(self.items) == 0

# Performance Review:
# ListQueue: dequeue() uses pop(0), which is O(n) because all elements must shift.
# DequeQueue: dequeue() uses popleft(), which is O(1) and much faster for large queues.
# For production code, prefer collections.deque for queue implementations.
```

```

3 # ListQueue: dequeue() uses pop(0), which is O(n) because all elements must shift.
4 # DequeQueue: dequeue() uses popleft(), which is O(1) and much faster for large queues.
5 # For production code, prefer collections.deque for queue implementations.
6
7 # Sample usage and testing
8 if __name__ == "__main__":
9     print("Testing ListQueue:")
10    lq = ListQueue()
11    lq.enqueue(1)
12    lq.enqueue(2)
13    lq.enqueue(3)
14    print(lq.dequeue()) # 1
15    print(lq.dequeue()) # 2
16    print("Is empty?", lq.is_empty()) # False
17
18    print("\nTesting DequeQueue:")
19    dq = DequeQueue()
20    dq.enqueue(10)
21    dq.enqueue(20)
22    dq.enqueue(30)
23    print(dq.dequeue()) # 10
24    print(dq.dequeue()) # 20
25    print("Is empty?", dq.is_empty())

```

OUTPUT:

```

python310\python.exe 'c:\Users\Rajalaxmi\.vscode\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher
r' '51812' '-' 'C:\Users\Rajalaxmi\Downloads\new.py'
Testing ListQueue:
1
2
Is empty? False

Testing DequeQueue:
10
20
Is empty? False
PS C:\Users\Rajalaxmi\Downloads>

```

## OBSERVATION:

1. The list-based queue is simple but inefficient for frequent dequeue() operations because removing from the front requires shifting all elements.
2. The deque-based queue is optimal as it provides  $O(1)$  complexity for both enqueue and dequeue.
3. For real-world applications where queues are used extensively or with many elements, collections.deque is the superior choice.
4. Both implementations handle empty queues gracefully and include clear docstrings and edge-case handling

## TASK3

PROMPT: Implement a Singly Linked List with operations:  
insert\_at\_end(), delete\_value(), and traverse()

## CODE:

```

class Node:
    """Represents a node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None # pointer to the next node

class LinkedList:
    """Singly linked list implementation."""
    def __init__(self):
        self.head = None # Start with an empty list

    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 list is empty, new node becomes head
            return
        current = self.head
        while current.next:
            current = current.next # Traverse to the last node
        current.next = new_node # Update last node's next pointer

    def delete_value(self, value):
        """Deletes the first node with the specified value."""
        prev = None
        current = self.head
        while current:
            if current.data == value:
                if prev:
                    # Bypass the node to be deleted
                    prev.next = current.next
                else:
                    # Deleting the head node
                    self.head = current.next
                return True # Value found and deleted
            prev = current
            current = current.next
        return False # Value not found

    def traverse(self):
        """Traverse the linked list and print all nodes"""
        current = self.head
        while current:
            print(current.data)
            current = current.next

```

```

    prev = current
    current = current.next # Move both pointers forward
    return False # Value not found

def traverse(self):
    """Traverse the list and returns a list of node values."""
    result = []
    current = self.head
    while current:
        result.append(current.data)
        current = current.next # Move to next node
    return result

# Suggested test cases to validate all operations
if __name__ == '__main__':
    ll = LinkedList()
    # Test insertion
    ll.insert_at_end(10)
    ll.insert_at_end(20)
    ll.insert_at_end(30)
    print("After insertions:", ll.traverse()) # [10, 20, 30]

    # Test deletion (middle value)
    ll.delete_value(20)
    print("After deleting 20:", ll.traverse()) # [10, 30]

    # Test deletion (head value)
    ll.delete_value(10)
    print("After deleting 10:", ll.traverse()) # [30]

    # Test deletion (non-existent value)
    result = ll.delete_value(99)
    print("Delete 99 (should be False):", result) # False

    # Test deletion (last value)

```

```

61     print("After deleting 20:", ll.traverse()) # [10, 30]
62
63     # Test deletion (head value)
64     ll.delete_value(10)
65     print("After deleting 10:", ll.traverse()) # [30]
66
67     # Test deletion (non-existent value)
68     result = ll.delete_value(99)
69     print("Delete 99 (should be False):", result) # False
70
71     # Test deletion (last value)
72     ll.delete_value(30)
73     print("After deleting 30:", ll.traverse()) # []
74
75     # Test insertion after all deletions
76     ll.insert_at_end(40)
77     print("After inserting 40:", ll.traverse()) # [40]

```

## OUTPUT:

```

After insertions: [10, 20, 30]
After deleting 20: [10, 30]
After deleting 10: [30]
Delete 99 (should be False): False
After deleting 30: []
After inserting 40: [40]
PS C:\Users\Rajalaxmi\Downloads>

```

## OBSERVATION:

- The linkedlist correctly supports insertion at the end, deletion by value, and traversal.
- Pointer updates are handled carefully:
  - When inserting at the end, the last node's pointer is updated to link the new node.
  - When deleting, both head and non-head deletions are treated properly.
- Edge cases like deleting from an empty list or deleting non-existent elements are handled safely.
- The test cases cover normal usage and edge cases ensuring robustness.

## TASK4

PROMPT:

implement a Binary Search Tree with methods for insert(), search(), and inorder\_traversal()

CODE:

```
1 class Node:
2     """Represents a node in a Binary Search Tree."""
3     def __init__(self, data):
4         self.data = data
5         self.left = None # Left child
6         self.right = None # Right child
7
8 class BST:
9     """Binary Search Tree implementation."""
10    def __init__(self):
11        self.root = None
12
13    def insert(self, data):
14        """Inserts a value into the BST.
15
16        Args:
17            data: The value to insert.
18        """
19        def _insert(node, data):
20            if node is None:
21                return Node(data)
22            if data < node.data:
23                node.left = _insert(node.left, data)
24            elif data > node.data:
25                node.right = _insert(node.right, data)
26            # If data == node.data, do not insert duplicates
27            return node
28        self.root = _insert(self.root, data)
29
30    def search(self, value):
31        """Searches for a value in the BST.
32
33        Args:
34            value: The value to search for.
35
36        Returns:
37            True if found, False otherwise.
38        """
39        def _search(node, value):
40            if node is None:
41                return False
42            if value == node.data:
43                return True
44            elif value < node.data:
45                return _search(node.left, value)
46            else:
47                return _search(node.right, value)
48        return _search(self.root, value)
49
50    def inorder_traversal(self):
51        """Performs inorder traversal and returns a list of values."""
52        result = []
53        def _inorder(node):
54            if node:
55                _inorder(node.left)
56                result.append(node.data)
57                _inorder(node.right)
58        _inorder(self.root)
59        return result
60
61 # Test cases
62 if __name__ == "__main__":
63     bst = BST()
64     nums = [50, 30, 70, 20, 40, 60, 80]
65     for num in nums:
66         bst.insert(num)
67
68     print("Inorder Traversal:", bst.inorder_traversal()) # [20, 30, 40, 50, 60, 70, 80]
69     print("Search 40:", bst.search(40)) # True
```

```
70
71 Returns:
72     True if found, False otherwise.
73 """
74 def _search(node, value):
75     if node is None:
76         return False
77     if value == node.data:
78         return True
79     elif value < node.data:
80         return _search(node.left, value)
81     else:
82         return _search(node.right, value)
83 return _search(self.root, value)
84
85 def inorder_traversal(self):
86     """Performs inorder traversal and returns a list of values."""
87     result = []
88     def _inorder(node):
89         if node:
90             _inorder(node.left)
91             result.append(node.data)
92             _inorder(node.right)
93     _inorder(self.root)
94     return result
95
96 # Test cases
97 if __name__ == "__main__":
98     bst = BST()
99     nums = [50, 30, 70, 20, 40, 60, 80]
100    for num in nums:
101        bst.insert(num)
102
103    print("Inorder Traversal:", bst.inorder_traversal()) # [20, 30, 40, 50, 60, 70, 80]
104    print("Search 40:", bst.search(40)) # True
```

OUTPUT:

```
Inorder Traversal: [20, 30, 40, 50, 60, 70, 80]
Inorder Traversal: [20, 30, 40, 50, 60, 70, 80]
Search 40: True
Search 40: True
Search 100: False
PS C:\Users\Rajalaxmi\Downloads> 
```

OBSERVATION:

### 1. Correctness

- The BST correctly inserts elements maintaining the binary search property.
- Inorder traversal produces a sorted list of inserted elements.
- The search method correctly identifies whether elements are present or absent.

### 2. Edge cases handled

- Searching for elements not in the tree returns False.
- Searching for elements in the tree returns True.

### 3. Design clarity

- The recursive approach in both insertion and search ensures clean and concise logic.
- The helper functions clearly separate recursion from the public API methods.

## TASK 5

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

CODE:

```
class Graph:
    """Graph represented using an adjacency list."""

    def __init__(self):
        self.adj_list = {} # Dictionary to store adjacency list

    def add_edge(self, u, v):
        """Adds an edge from u to v (undirected)."""
        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def bfs(self, start):
        """Performs Breadth-First Search (BFS) traversal from the start node.

        Returns:
            List of nodes in BFS order.
        """
        visited = set()
        queue = [start]
        order = []

        while queue:
            node = queue.pop(0) # Dequeue the front node
            if node not in visited:
                visited.add(node)
                order.append(node)
                # Enqueue all unvisited neighbors
                for neighbor in self.adj_list.get(node, []):
                    if neighbor not in visited:
                        queue.append(neighbor)
        return order

    def dfs_recursive(self, start):
```

```

def dfs_recursive(self, start):
    """Performs Depth-First Search (DFS) recursively from the start node.

    Returns:
    | List of nodes in DFS order.
    """
    visited = set()
    order = []
    (parameter) node: Any

    def _dfs(node):
        if node not in visited:
            visited.add(node)
            order.append(node)
            # Recursively visit all neighbors
            for neighbor in self.adj_list.get(node, []):
                _dfs(neighbor)
    _dfs(start)
    return order

def dfs_iterative(self, start):
    """Performs Depth-First Search (DFS) iteratively from the start node.

    Returns:
    | List of nodes in DFS order.
    """
    visited = set()
    stack = [start]
    order = []

    while stack:
        node = stack.pop() # Pop the top node
        if node not in visited:
            visited.add(node)
            order.append(node)
            # Push all unvisited neighbors to stack
            for neighbor in reversed(self.adj_list.get(node, [])):
                if neighbor not in visited:
                    stack.append(neighbor)
    return order

# Example usage and comparison
if __name__ == "__main__":
    g = Graph()
    g.add_edge('A', 'B')
    g.add_edge('A', 'C')
    g.add_edge('B', 'D')
    g.add_edge('C', 'E')
    g.add_edge('D', 'E')
    g.add_edge('E', 'F')

    print("Adjacency List:", g.adj_list)
    print("BFS from A:", g.bfs('A')) # ['A', 'B', 'C', 'D', 'E', 'F']
    print("Recursive DFS from A:", g.dfs_recursive('A')) # DFS order
    print("Iterative DFS from A:", g.dfs_iterative('A')) # DFS order

    # Note: Recursive DFS uses function calls, iterative DFS uses a stack.
    # Both visit all reachable nodes, but order may differ depending on adjacency list.

```

OUTPUT:

```

Adjacency List: {'A': ['B', 'C'], 'B': ['A', 'D'], 'C': ['A', 'E'], 'D': ['B', 'E'], 'E': ['C', 'D', 'F'], 'F': ['E']}
BFS from A: ['A', 'B', 'C', 'D', 'E', 'F']
Recursive DFS from A: ['A', 'B', 'D', 'E', 'C', 'F']
Iterative DFS from A: ['A', 'B', 'D', 'E', 'C', 'F']
PS C:\Users\Rajalaxmi\Downloads>

```

OBSERVATION:

## 1. Graph Representation:

- The adjacency list is efficient in representing sparse graphs.
- Neighbors are stored in lists for each node.

## 2. BFS Traversal:

- Uses a queue (FIFO structure).
- Visits all neighbors level by level.
- Suitable for shortest path problems in unweighted graphs.

#### DFS TRAVERSAL:

- Can be implemented both recursively and iteratively.
- Recursive DFS is simpler to write and more elegant.
- Iterative DFS avoids recursion depth issues and is more robust for large graphs.