

Name: K.Likitha

Hall Ticket: 2303A52144

Batch: 41

Task 1 Stack Implementation for Undo Operations (LIFO)

Prompt:

Create a Python Stack class for undo operations with push, pop, peek, is_empty. Add docstrings, sample test cases, explain why stack suits undo (LIFO), and show alternative using collections.deque with short comparison.

Code:

```
class Stack:
    """Implements a Last-In-First-Out (LIFO) stack data structure."""

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

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

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

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

        Raises:
            IndexError: If the stack is empty.
        """
        Returns:
            The item removed from the top of the stack.
        """
        if self.is_empty():
            raise IndexError("Cannot pop from an empty stack")
        return self._items.pop()

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

        Raises:
            IndexError: If the stack is empty.
        """
        Returns:
            The item at the top of the stack.
        """
        if self.is_empty():
            raise IndexError("Cannot peek into an empty stack")
        return self._items[-1]
```

```
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.

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

def __str__(self):
    """Returns a string representation of the stack."""
    return str(self._items)

def main_stack_interaction():
    undo_stack = Stack()
    print("--- Interactive Stack Operations (List-based) ---")

    while True:
        print("\nCurrent Stack: ", undo_stack)
        print("1. Push (add action)")
        print("2. Pop (undo last action)")
        print("3. Peek (view last action)")
        print("4. Check if Empty")
        print("5. Exit")

        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            action = input("Enter action to push: ")
            undo_stack.push(action)
            print(f'{action} pushed to stack.')
        elif choice == '2':
            try:
                action = undo_stack.pop()
                print(f'{action} popped (undone).')
            except IndexError as e:
                print(f"Error: {e}")
        elif choice == '3':
            try:
                action = undo_stack.peek()
                print(f'Top action: {action}.')
            except IndexError as e:
                print(f"Error: {e}")
        elif choice == '4':
            if undo_stack.is_empty():
                print("Stack is empty.")
```

```

        else:
            print("Stack is not empty.")
        elif choice == '5':
            print("Exiting interactive stack operations.")
            break
        else:
            print("Invalid choice. Please enter a number between 1 and 5.")

# Run the interactive session
main_stack_interaction()

```

Output:

```

--- Interactive Stack Operations (List-based) ---
...
Current Stack: []
1. Push (add action)
2. Pop (undo last action)
3. Peek (view last action)
4. Check if Empty
5. Exit
Enter your choice (1-5): 4
Stack is empty.

Current Stack: []
1. Push (add action)
2. Pop (undo last action)
3. Peek (view last action)
4. Check if Empty
5. Exit
Enter your choice (1-5): 1
Enter action to push: ai
'ai' pushed to stack.

Current Stack: ['ai']
1. Push (add action)
2. Pop (undo last action)
3. Peek (view last action)
4. Check if Empty
5. Exit
Enter your choice (1-5): 2
'ai' popped (undone).

Current Stack: []
1. Push (add action)
2. Pop (undo last action)
3. Peek (view last action)
4. Check if Empty
5. Exit
Enter your choice (1-5): 5
Exiting interactive stack operations.

```

Explanation:

A stack is ideal for undo because it follows Last-In-First-Out, so the most recent action is undone first. A list is simple to use for a stack, while collections.deque is slightly more efficient for frequent push and pop operations.

Task 2 Queue for Customer Service Requests (FIFO)

Prompt:

Create a Python Queue class (FIFO) with enqueue, dequeue, is_empty using a list. Then review its performance, implement an optimized version using collections.deque, and give a very short paragraph comparing time complexity and real-world impact.

Code:

```
class ListQueue:
    """Implements a First-In-First-Out (FIFO) queue data structure using a Python
list."""

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

    def enqueue(self, request):
        """Adds a request to the rear of the queue.

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

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

        Raises:
            IndexError: If the queue is empty.

        Returns:
            The item removed from the front of the queue.
        """
        if self.is_empty():
            raise IndexError("Cannot dequeue from an empty queue")
        return self._items.pop(0) # Removing from the beginning of the list

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

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

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

        Returns:
            The number of items in the queue.
        """
        return len(self._items)

    def __str__(self):
        """Returns a string representation of the queue."""
        return str(self._items)

def main_list_queue_interaction():
    service_queue = ListQueue()
```

```
print("---- Interactive List-based Queue Operations ---")

while True:
    print("\nCurrent Queue: ", service_queue)
    print("1. Enqueue (add request)")
    print("2. Dequeue (handle next request)")
    print("3. Check if Empty")
    print("4. Exit")

    choice = input("Enter your choice (1-4): ")

    if choice == '1':
        request = input("Enter request to enqueue: ")
        service_queue.enqueue(request)
        print(f'{request} enqueued.')
    elif choice == '2':
        try:
            request = service_queue.dequeue()
            print(f"Handling: {request}.")
        except IndexError as e:
            print(f"Error: {e}")
    elif choice == '3':
        if service_queue.is_empty():
            print("Queue is empty.")
        else:
            print("Queue is not empty.")
    elif choice == '4':
        print("Exiting interactive queue operations.")
        break
    else:
        print("Invalid choice. Please enter a number between 1 and 4.")

# Run the interactive session for ListQueue
main_list_queue_interaction()
```

Output:

```
... --- Interactive List-based Queue Operations ---  
  
Current Queue: []  
1. Enqueue (add request)  
2. Dequeue (handle next request)  
3. Check if Empty  
4. Exit  
Enter your choice (1-4): 1  
Enter request to enqueue: password  
'password' enqueued.  
  
Current Queue: ['password']  
1. Enqueue (add request)  
2. Dequeue (handle next request)  
3. Check if Empty  
4. Exit  
Enter your choice (1-4): 2  
Handling: 'password'.  
  
Current Queue: []  
1. Enqueue (add request)  
2. Dequeue (handle next request)  
3. Check if Empty  
4. Exit  
Enter your choice (1-4): 3  
Queue is empty.  
  
Current Queue: []  
1. Enqueue (add request)  
2. Dequeue (handle next request)  
3. Check if Empty  
4. Exit  
Enter your choice (1-4): 4  
Exiting interactive queue operations.
```

Explanation:

A queue works on First-In-First-Out, so the earliest request is handled first. A list-based queue is simple but dequeue is slow because it shifts elements ($O(n)$). A deque is faster since enqueue and dequeue are $O(1)$, making it better for real customer service systems.

Task 3: Singly Linked List for Dynamic Playlist Management**Prompt:**

Create a Python Singly Linked List for playlist management with `insert_at_end`, `delete_value`, and `traverse`. Add inline comments explaining pointer manipulation, highlight tricky parts in insertion/deletion, include edge case tests (empty list, single node, delete head), and give a very short paragraph explaining how the code works.

Code:

```
class Node:  
    """Represents a node in a singly linked list."""  
  
    def __init__(self, data):  
        """Initializes a new node.  
  
        Args:  
            data: The data to be stored in the node (e.g., song name).  
        """  
        self.data = data  
        self.next = None # Pointer to the next node, initially None
```

```
class SinglyLinkedList:
    """Manages a singly linked list for dynamic playlist management."""

    def __init__(self):
        """Initializes an empty singly linked list."""
        self.head = None # The head points to the first node, initially None

    def insert_at_end(self, song):
        """Inserts a new song (node) at the end of the linked list.

        Args:
            song: The song data to be inserted.
        """
        new_node = Node(song)
        if self.head is None:
            # If the list is empty, the new node becomes the head.
            self.head = new_node
            return

        # Traverse to the last node
        current = self.head
        while current.next:
            current = current.next

        # Link the last node's 'next' pointer to the new node
        current.next = new_node

    def delete_value(self, song):
        """Deletes the first occurrence of a node with the given song value.

        Args:
            song: The song data to be deleted.
        """
        current = self.head

        # Case 1: If the head itself holds the song to be deleted
        if current and current.data == song:
            self.head = current.next # Move head to the next node
            current = None # Dereference the old head node
            print(f"Deleted '{song}' (head).")
            return

        prev = None
        # Search for the song to be deleted, keep track of the previous node
        while current and current.data != song:
            prev = current
            current = current.next

        # Case 2: If song was not present in linked list
        if current is None:
            print(f"'{song}' not found in playlist.")
            return
```

```

# Case 3: If song is found (not head)
# Unlink the node from the linked list
if prev:
    prev.next = current.next
else:
    # This case should ideally be handled by Case 1, but as a safeguard.
    self.head = current.next

current = None # Dereference the node
print(f"Deleted '{song}'.")

def traverse(self):
    """Traverses the linked list and prints all song names."""
    songs = []
    current = self.head
    if not current:
        print("Playlist is empty.")
        return
    while current:
        songs.append(current.data)
        current = current.next # Move to the next node using its 'next' pointer
    print("Current Playlist: ", " -> ".join(songs))

def __str__(self):
    """Returns a string representation of the playlist."""
    songs = []
    current = self.head
    while current:
        songs.append(str(current.data))
        current = current.next
    return " -> ".join(songs) if songs else "Empty Playlist"

# --- Test Cases for Singly Linked List Playlist ---
print("\n--- Playlist Management Test Cases ---")
playlist = SinglyLinkedList()

# Edge Case: Empty list traversal and deletion
print("\nTesting with empty playlist:")
playlist.traverse()
playlist.delete_value("Nonexistent Song")

# Insert songs at the end
print("\nInserting songs:")
playlist.insert_at_end("Song A")
playlist.insert_at_end("Song B")
playlist.insert_at_end("Song C")
playlist.traverse()

# Delete a song from the middle
print("\nDeleting 'Song B' (middle):")
playlist.delete_value("Song B")
playlist.traverse()

```

```
# Insert another song
print("\nInserting 'Song D':")
playlist.insert_at_end("Song D")
playlist.traverse()

# Edge Case: Delete a song at the head
print("\nDeleting 'Song A' (head):")
playlist.delete_value("Song A")
playlist.traverse()

# Edge Case: Delete the only remaining song (single node list)
print("\nDeleting 'Song C' (single node list):")
playlist.delete_value("Song C")
playlist.traverse()

# List should now be 'Song D'
print("\nDeleting 'Song D' (last remaining):")
playlist.delete_value("Song D")
playlist.traverse()

# Attempt to delete from an empty list again
print("\nAttempting to delete from an empty list:")
playlist.delete_value("Any Song")
playlist.traverse()

# Test inserting into an empty list after deletions
print("\nInserting into an empty list after deletions:")
playlist.insert_at_end("New Song 1")
playlist.traverse()
playlist.insert_at_end("New Song 2")
playlist.traverse()

# Delete a non-existent song
print("\nDeleting non-existent song:")
playlist.delete_value("Nonexistent Song 2")
playlist.traverse()
```

Output:

```
... --- Playlist Management Test Cases ---  
  
Testing with empty playlist:  
Playlist is empty.  
'Nonexistent Song' not found in playlist.  
  
Inserting songs:  
Current Playlist: Song A -> Song B -> Song C  
  
Deleting 'Song B' (middle):  
Deleted 'Song B'.  
Current Playlist: Song A -> Song C  
  
Inserting 'Song D':  
Current Playlist: Song A -> Song C -> Song D  
  
Deleting 'Song A' (head):  
Deleted 'Song A' (head).  
Current Playlist: Song C -> Song D  
  
Deleting 'Song C' (single node list):  
Deleted 'Song C' (head).  
Current Playlist: Song D  
  
Deleting 'Song D' (last remaining):  
Deleted 'Song D' (head).  
Playlist is empty.  
  
Attempting to delete from an empty list:  
'Any Song' not found in playlist.  
Playlist is empty.  
  
Inserting into an empty list after deletions:  
Current Playlist: New Song 1  
Current Playlist: New Song 1 -> New Song 2  
  
Deleting non-existent song:  
'Nonexistent Song 2' not found in playlist.  
Current Playlist: New Song 1 -> New Song 2
```

Explanation:

A singly linked list stores each song in a node that points to the next song. Insertion at the end requires traversing until the last node and updating its next pointer. Deletion is tricky because you must correctly update the previous node's pointer, especially when removing the head or when the list has only one node.

Task 4: Binary Search Tree for Fast Record Lookup

Prompt:

Complete a partially written Python Binary Search Tree with insert, search, and inorder_traversal. Add proper docstrings and give a very short paragraph explaining how BST improves search efficiency compared to linear search, including best vs worst case time complexity.

Code:

```
class Node:  
    """Represents a node in a Binary Search Tree (BST)."""  
  
    def __init__(self, key):  
        """Initializes a new node with a given key.  
  
        Args:  
            key: The value (e.g., roll number) to be stored in the node.  
        """
```

```

"""
    self.key = key
    self.left = None    # Pointer to the left child node
    self.right = None   # Pointer to the right child node

class BinarySearchTree:
    """Implements a Binary Search Tree for efficient record lookup."""

    def __init__(self):
        """Initializes an empty Binary Search Tree."""
        self.root = None  # The root of the BST, initially None

    def insert(self, key):
        """Inserts a new key into the BST.

        Args:
            key: The value to be inserted.
        """
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        """Helper function to recursively insert a key into the BST.

        Args:
            node: The current node being examined.
            key: The value to be inserted.

        Returns:
            The node after insertion.
        """
        if node is None:
            return Node(key)  # If current position is empty, create new node

        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        elif key > node.key:
            node.right = self._insert_recursive(node.right, key)
        # If key is equal to node.key, we don't insert duplicates in this
implementation
        return node

    def search(self, key):
        """Searches for a key in the BST.

        Args:
            key: The value to search for.

        Returns:
            True if the key is found, False otherwise.
        """
        return self._search_recursive(self.root, key)

    def _search_recursive(self, node, key):
        """Helper function to recursively search for a key in the BST.

```

```

Args:
    node: The current node being examined.
    key: The value to search for.

Returns:
    True if the key is found, False otherwise.
"""

if node is None:
    return False # Key not found

if node.key == key:
    return True # Key found
elif key < node.key:
    return self._search_recursive(node.left, key)
else: # key > node.key
    return self._search_recursive(node.right, key)

def inorder_traversal(self):
    """Performs an in-order traversal of the BST and prints the keys.
    In-order traversal visits nodes in ascending order of their keys.
    """
    result = []
    self._inorder_recursive(self.root, result)
    print("In-order Traversal (Sorted Roll Numbers):", result)

def _inorder_recursive(self, node, result):
    """Helper function for recursive in-order traversal.

Args:
    node: The current node being visited.
    result: A list to store the keys in order.
"""

    if node:
        self._inorder_recursive(node.left, result)
        result.append(node.key)
        self._inorder_recursive(node.right, result)

# --- Test Cases for Binary Search Tree ---
print("--- Binary Search Tree for Student Records ---")
bst = BinarySearchTree()

# Insert student roll numbers
print("\nInserting roll numbers: 50, 30, 70, 20, 40, 60, 80, 55")
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)
bst.insert(55)

```

```

# Perform in-order traversal (should print sorted roll numbers)
bst.inorder_traversal()

# Search for existing roll numbers
print("\nSearching for roll number 40:", bst.search(40)) # Expected: True
print("Searching for roll number 80:", bst.search(80)) # Expected: True
print("Searching for roll number 50:", bst.search(50)) # Expected: True

# Search for non-existing roll numbers
print("Searching for roll number 25:", bst.search(25)) # Expected: False
print("Searching for roll number 90:", bst.search(90)) # Expected: False

# Test with an empty BST
print("\nTesting with an empty BST:")
empty_bst = BinarySearchTree()
print("Searching for 100 in empty BST:", empty_bst.search(100)) # Expected: False
empty_bst.inorder_traversal() # Expected: Empty list output

# Test inserting duplicate (should not be added in this implementation)
print("\nInserting 50 again:")
bst.insert(50)
bst.inorder_traversal() # Should remain the same

```

Output:

```

*** --- Binary Search Tree for Student Records ---

Inserting roll numbers: 50, 30, 70, 20, 40, 60, 80, 55
In-order Traversal (Sorted Roll Numbers): [20, 30, 40, 50, 55, 60, 70, 80]

Searching for roll number 40: True
Searching for roll number 80: True
Searching for roll number 50: True
Searching for roll number 25: False
Searching for roll number 90: False

Testing with an empty BST:
Searching for 100 in empty BST: False
In-order Traversal (Sorted Roll Numbers): []

Inserting 50 again:
In-order Traversal (Sorted Roll Numbers): [20, 30, 40, 50, 55, 60, 70, 80]

```

Explanation:

A Binary Search Tree stores smaller values on the left and larger values on the right, allowing faster searching compared to linear search. In the best case (balanced tree), search takes $O(\log n)$, but in the worst case (skewed tree), it becomes $O(n)$, similar to linear search.

Task 5: Graph Traversal for Social Network Connections

Prompt:

Create a Python graph using an adjacency list and implement BFS and DFS. Add inline comments explaining traversal steps, compare recursive vs iterative DFS briefly, and give a very short paragraph explaining BFS vs DFS use cases.

Code:

```

from collections import deque

class Graph:
    """Represents a social network using an adjacency list."""

```

```

def __init__(self):
    """Initializes an empty graph."""
    self.graph = {} # Adjacency list: {node: [neighbor1, neighbor2, ...]}

def add_edge(self, u, v):
    """Adds an edge between two users (nodes) u and v.

    Since it's a social network (undirected), add edges in both directions.

    Args:
        u: The first user (node).
        v: The second user (node).
    """
    if u not in self.graph:
        self.graph[u] = []
    if v not in self.graph:
        self.graph[v] = []
    self.graph[u].append(v)
    self.graph[v].append(u)

def bfs(self, start_node):
    """Performs a Breadth-First Search (BFS) starting from a given node.
    Finds all reachable nodes level by level.

    Args:
        start_node: The node to start the traversal from.

    Returns:
        A list of nodes visited in BFS order.
    """
    visited = set() # Keep track of visited nodes to avoid cycles and re-
processing
    queue = deque() # Use a deque for efficient O(1) appends and pops from
left
    bfs_order = [] # Store the order of nodes visited

    if start_node not in self.graph:
        print(f"Node {start_node} not in graph.")
        return []

    # Start BFS
    queue.append(start_node)
    visited.add(start_node)

    while queue:
        current_node = queue.popleft() # Dequeue the first node
        bfs_order.append(current_node) # Add to the traversal order

        # Explore all unvisited neighbors of the current node
        for neighbor in self.graph[current_node]:
            if neighbor not in visited:
                visited.add(neighbor) # Mark as visited
                queue.append(neighbor) # Enqueue for future exploration

```

```

        return bfs_order

    def dfs_recursive(self, start_node):
        """Performs a Depth-First Search (DFS) recursively starting from a given
node.

        Explores as far as possible along each branch before backtracking.

    Args:
        start_node: The node to start the traversal from.

    Returns:
        A list of nodes visited in DFS order.
    """
    visited = set()      # Keep track of visited nodes
    dfs_order = []       # Store the order of nodes visited

    if start_node not in self.graph:
        print(f"Node {start_node} not in graph.")
        return []

    def _dfs_util(node):
        visited.add(node)           # Mark the current node as visited
        dfs_order.append(node)      # Add to traversal order

        # Recursively visit all unvisited neighbors
        if node in self.graph:
            for neighbor in self.graph[node]:
                if neighbor not in visited:
                    _dfs_util(neighbor)

    _dfs_util(start_node)
    return dfs_order

    def dfs_iterative(self, start_node):
        """Performs a Depth-First Search (DFS) iteratively using a stack.

    Args:
        start_node: The node to start the traversal from.

    Returns:
        A list of nodes visited in DFS order.
    """
    visited = set()      # Keep track of visited nodes
    stack = []           # Use a list as a stack (LIFO)
    dfs_order = []       # Store the order of nodes visited

    if start_node not in self.graph:
        print(f"Node {start_node} not in graph.")
        return []

        # Push the start node onto the stack and mark as visited initially (or when
popped)
        stack.append(start_node)

```

```

    visited.add(start_node) # Mark as visited when pushing to avoid re-adding
    to stack

    while stack:
        current_node = stack.pop()          # Pop the top node from the stack
        dfs_order.append(current_node)     # Add to traversal order

        # Explore neighbors. Push unvisited neighbors onto stack.
        # Pushing in reverse order ensures correct traversal order if
processing top of stack.
        # For example, if neighbors are [B, C], pushing C then B means B is
processed first.
        # To ensure consistent order with recursive DFS, process neighbors in
sorted/reverse sorted order.
        if current_node in self.graph:
            # Sort neighbors to ensure consistent output for testing, otherwise
order might vary
            # depending on adjacency list order. We iterate in reverse to
simulate pushing B then C
            # such that C is processed first (as it's popped last from the
stack).

            for neighbor in sorted(self.graph[current_node], reverse=True):
                if neighbor not in visited:
                    visited.add(neighbor)    # Mark as visited
                    stack.append(neighbor)   # Push onto stack
    return dfs_order

# --- Test Cases for Graph Traversal ---
print("--- Social Network Connections (Graph Traversal) ---")
network = Graph()

# Add users and their connections
network.add_edge("Alice", "Bob")
network.add_edge("Alice", "Carol")
network.add_edge("Bob", "David")
network.add_edge("Carol", "Eve")
network.add_edge("David", "Frank")
network.add_edge("Eve", "Grace")
network.add_edge("Frank", "Heidi")
network.add_edge("Grace", "Ivan")
network.add_edge("Heidi", "Ivan")

print("\nBFS from Alice (nearby connections):")
print(network.bfs("Alice"))
# Expected: ['Alice', 'Bob', 'Carol', 'David', 'Eve', 'Frank', 'Grace', 'Heidi',
'Ivan']

print("\nRecursive DFS from Alice (deep connection paths):")
print(network.dfs_recursive("Alice"))
# Expected (order can vary slightly based on adjacency list order, but path is
deep):
# E.g., ['Alice', 'Bob', 'David', 'Frank', 'Heidi', 'Ivan', 'Grace', 'Eve',
'Carol']

```

```

print("\nIterative DFS from Alice (deep connection paths):")
print(network.dfs_iterative("Alice"))
# Expected: ['Alice', 'Carol', 'Eve', 'Grace', 'Ivan', 'Heidi', 'Frank', 'David',
'Bob']
# (Order might be different from recursive due to explicit stack manipulation vs.
call stack)

print("\nBFS from a node not in graph:")
print(network.bfs("Zoe")) # Expected: Node Zoe not in graph. []

print("\nDFS from a node not in graph:")
print(network.dfs_recursive("Zoe")) # Expected: Node Zoe not in graph. []

```

Output:

```

--- Social Network Connections (Graph Traversal) ---

BFS from Alice (nearby connections):
['Alice', 'Bob', 'Carol', 'David', 'Eve', 'Frank', 'Grace', 'Heidi', 'Ivan']

Recursive DFS from Alice (deep connection paths):
['Alice', 'Bob', 'David', 'Frank', 'Heidi', 'Ivan', 'Grace', 'Eve', 'Carol']

Iterative DFS from Alice (deep connection paths):
['Alice', 'Bob', 'David', 'Frank', 'Heidi', 'Ivan', 'Grace', 'Eve', 'Carol']

BFS from a node not in graph:
Node Zoe not in graph.
[]

DFS from a node not in graph:
Node Zoe not in graph.
[]

```

Explanation:

A graph uses an adjacency list to store connections between users. BFS explores level by level, making it useful for finding nearby connections, while DFS goes deep into one path before backtracking, which is useful for exploring complete connection chains. Recursive DFS uses the call stack, while iterative DFS uses an explicit stack structure.