

Assignment-11.4

G.Sai teja

2303A52135

Batch 41

Task-1: Stack Implementation for Undo Operations (LIFO)

PROMPT:

Create a Python UndoStack class that implements a stack for undo operations. Use a private list to store actions. Include push() to add actions, pop() to remove and return the most recent (return None if empty), peek() to view without removing (return None if empty), is_empty() to check emptiness, size() to return the count, and __str__() for string representation. Use type hints (typing.Any, typing.Optional). Add a test function that exercises all methods, including edge cases like empty stacks. Handle empty stack cases without exceptions.

CODE:

```
Lab-11.py Lab-12.py Lab-13.py X
Close (Ctrl+F4)
Lab
1 #Create a Python UndoStack class that implements a stack for undo operations. Use a private list to store actions.
2 from typing import Any, Optional
3 class UndoStack:
4     """Stack implementation for undo functionality."""
5     def __init__(self):
6         self._items = []
7     def push(self, action: Any) -> None:
8         """Add an action to the top of the stack."""
9         self._items.append(action)
10    def pop(self) -> Optional[Any]:
11        """Remove and return the most recent action."""
12        return self._items.pop() if not self.is_empty() else None
13    def peek(self) -> Optional[Any]:
14        """Return the most recent action without removing it."""
15        return self._items[-1] if not self.is_empty() else None
16    def is_empty(self) -> bool:
17        """Check if the stack is empty."""
18        return len(self._items) == 0
19    def size(self) -> int:
20        """Return the number of actions in the stack."""
21        return len(self._items)
22    def __str__(self) -> str:
23        return str(self._items)
24    def test_undo_stack():
25        """Test the UndoStack class."""
26        print("TESTING UndoStack CLASS")
27        print("=" * 100)
28        stack = UndoStack()
29        # Test 1: Pushing actions onto the stack
30        print("\n[Test 1] Pushing actions onto the stack")
31        stack.push("Action 1")
32        stack.push("Action 2")
33        stack.push("Action 3")
```

```

Lab-13.py X
Close (Ctrl+F4)
Lab
24 def test_undo_stack():
34     print(f"Current stack: {stack}")
35     # Test 2: Peeking at the top action
36     print("\n[Test 2] Peeking at the top action")
37     top_action = stack.peak()
38     expected = "Action 3"
39     print(f"Top action: {top_action}, Expected: {expected}")
40     if top_action == expected:
41         print("✓ PASSED")
42     # Test 3: Popping actions from the stack
43     print("\n[Test 3] Popping actions from the stack")
44     popped1 = stack.pop()
45     expected1 = "Action 3"
46     print(f"Popped action: {popped1}, Expected: {expected1}")
47     popped2 = stack.pop()
48     expected2 = "Action 2"
49     print(f"Popped action: {popped2}, Expected: {expected2}")
50     # Test 4: Checking if the stack is empty
51     print("\n[Test 4] Checking if the stack is empty")
52     is_empty_before = stack.is_empty()
53     expected_empty_before = False
54     print(f"Is stack empty? {is_empty_before}, Expected: {expected_empty_before}")
55     if is_empty_before == expected_empty_before:
56         print("✓ PASSED")
57     # Pop the remaining action
58     stack.pop()
59     is_empty_after = stack.is_empty()
60     expected_empty_after = True
61     print(f"Is stack empty after popping all actions? {is_empty_after}, Expected: {expected_empty_after}")
62     if is_empty_after == expected_empty_after:
63         print("✓ PASSED")
64 if __name__ == "__main__":
65     test_undo_stack()

```

OUTPUT:

```

TESTING UndoStack CLASS
=====

[Test 1] Pushing actions onto the stack
Current stack: ['Action 1', 'Action 2', 'Action 3']

[Test 2] Peeking at the top action
Top action: Action 3, Expected: Action 3
✓ PASSED

[Test 3] Popping actions from the stack
Popped action: Action 3, Expected: Action 3
Popped action: Action 2, Expected: Action 2

[Test 4] Checking if the stack is empty
Is stack empty? False, Expected: False
✓ PASSED
Is stack empty after popping all actions? True, Expected: True
✓ PASSED
PS C:\Users\saita\Downloads\AI ASSISTENT CODING>

```

EXPLANATION:

The UndoStack class implements a stack using a private `_items` list. It provides `push()` to add actions, `pop()` to remove and return the top item (returns `None` if empty), `peek()` to view the top without removing it (returns `None` if empty), `is_empty()` to check if the stack is empty, `size()` to return the count, and `__str__()` for string representation. The class uses type hints (`Any`, `Optional`) and handles empty stacks without exceptions. The `test_undo_stack()` function exercises all methods: it pushes three actions, peeks at the top, pops items in LIFO order, and verifies `is_empty()` before and after emptying the stack. The implementation follows standard stack behavior where the last item added is the first one removed.

Task-2: Queue for Customer Service Requests (FIFO)

List Based Queue:

PROMPT:

Create a Python `CustomerServiceQueue` class implementing a FIFO queue using a private list. Include `enqueue(request: Any)`, `dequeue() -> Optional[Any]` (returns None if empty), `is_empty() -> bool`, `size() -> int`, and `__str__()` methods. Use type hints (`typing.Any`, `typing.Optional`) and handle empty queue cases gracefully. Provide a comprehensive test function covering all methods and edge cases, including dequeuing from an empty queue.

CODE:

```
Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py
61 # Create a Python CustomerServiceQueue class implementing a FIFO queue using a private List. Include enqueue
62 from typing import Any, Optional
63 class CustomerServiceQueue:
64     """Queue implementation for customer service requests (FIFO)."""
65     def __init__(self):
66         """Initialize an empty queue."""
67         self._items = []
68     def enqueue(self, request: Any) -> None:
69         """Add a service request to the end of the queue."""
70         self._items.append(request)
71     def dequeue(self) -> Optional[Any]:
72         """Remove and return the first service request (FIFO)."""
73         return self._items.pop(0) if not self.is_empty() else None
74     def is_empty(self) -> bool:
75         """Check if the queue is empty."""
76         return len(self._items) == 0
77     def size(self) -> int:
78         """Return the number of requests in the queue."""
79         return len(self._items)
80     def __str__(self) -> str:
81         """Return string representation of the queue."""
82         return str(self._items)
83     def test_customer_service_queue():
84         """Test the CustomerServiceQueue class."""
85         print("TESTING CustomerServiceQueue CLASS")
86         print("-" * 100)
87         queue = CustomerServiceQueue()
88         # Test 1: Enqueue service requests
89         print("\n[Test 1] Adding service requests to the queue")
90         queue.enqueue("Request 1: Password Reset")
91         queue.enqueue("Request 2: Account Inquiry")
92         queue.enqueue("Request 3: Billing Question")
93         print(f"Current queue: {queue}")
```

```
Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py
83 def test_customer_service_queue():
84     print(f"Queue size: {queue.size()}")
85     # Test 2: Check if queue is empty
86     print("\n[Test 2] Checking if queue is empty")
87     is_empty = queue.is_empty()
88     print(f"Is queue empty? {is_empty}, Expected: False")
89     if is_empty == False:
90         print("✓ PASSED")
91     # Test 3: Dequeue requests (FIFO order)
92     print("\n[Test 3] Processing service requests (FIFO)")
93     request1 = queue.dequeue()
94     expected1 = "Request 1: Password Reset"
95     print(f"Dequeued: {request1}, Expected: {expected1}")
96     if request1 == expected1:
97         print("✓ PASSED")
98     request2 = queue.dequeue()
99     expected2 = "Request 2: Account Inquiry"
100    print(f"Dequeued: {request2}, Expected: {expected2}")
101    if request2 == expected2:
102        print("✓ PASSED")
103    print(f"Remaining queue: {queue}")
104    # Test 4: Dequeue remaining request
105    request3 = queue.dequeue()
106    expected3 = "Request 3: Billing Question"
107    print(f"Dequeued: {request3}, Expected: {expected3}")
108    if request3 == expected3:
109        print("✓ PASSED")
110    # Test 5: Check empty queue
111    print("\n[Test 4] Checking empty queue")
112    is_empty_after = queue.is_empty()
113    print(f"Is queue empty? {is_empty_after}, Expected: True")
114    if is_empty_after == True:
115        print("✓ PASSED")
116    print("\nAll tests passed successfully!")
```

```
Lab-13.py X
Lab- C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py
83 def test_customer_service_queue():
126     # Test 6: Dequeue from empty queue
127     print("\n[Test 5] Dequeue from empty queue")
128     empty_dequeue = queue.dequeue()
129     print(f"Dequeued from empty queue: {empty_dequeue}, Expected: None")
130     if empty_dequeue is None:
131         print("✓ PASSED")
132 if __name__ == "__main__":
133     test_customer_service_queue()
```

OUTPUT:

```
Programs/Python/Python314/python.exe "c:/Users/saite/Downloads/AI ASSISTED CODING/As11.py/task2.py"
TESTING CustomerServiceQueue CLASS
=====

[Test 1] Adding service requests to the queue
Current queue: ['Request 1: Password Reset', 'Request 2: Account Inquiry', 'Request 3: Billing Question']
Queue size: 3

[Test 2] Checking if queue is empty
Is queue empty? False, Expected: False
✓ PASSED

[Test 3] Processing service requests (FIFO)
Dequeued: Request 1: Password Reset, Expected: Request 1: Password Reset
✓ PASSED
Dequeued: Request 2: Account Inquiry, Expected: Request 2: Account Inquiry
✓ PASSED
Remaining queue: ['Request 3: Billing Question']
Dequeued: Request 3: Billing Question, Expected: Request 3: Billing Question
✓ PASSED

[Test 4] Checking empty queue
Is queue empty? True, Expected: True
✓ PASSED
```

Deque-based optimized queue:

PROMPT:

Create a Python CustomerServiceQueue class using collections.deque for an optimized FIFO queue. Include enqueue(request: Any), dequeue() -> Optional[Any] (returns None if empty), is_empty() -> bool, size() -> int, and __str__() methods. Ensure all operations are efficient (O(1)) and handle empty queue cases gracefully. Also, provide a test function covering all methods and edge cases, with a brief explanation of why deque is more efficient than a list-based queue.

CODE:

```
Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py
136 #Create a Python CustomerServiceQueue class using collections.deque for an optimized FIFO queue. Include a
137 from collections import deque
138 from typing import Any, Optional
139 class CustomerServiceQueue:
140     """
141     A queue for managing customer service requests using FIFO order,
142     optimized using collections.deque for efficient enqueue and dequeue operations.
143     """
144     def __init__(self) -> None:
145         self._queue: deque[Any] = deque[Any]() # private deque to store requests
146     def enqueue(self, request: Any) -> None:
147         """Add a service request to the end of the queue."""
148         self._queue.append(request)
149     def dequeue(self) -> Optional[Any]:
150         """
151         Remove and return the first service request from the front of the queue.
152         Returns None if the queue is empty.
153         """
154         if self.is_empty():
155             return None
156         return self._queue.popleft() # O(1) operation
157     def is_empty(self) -> bool:
158         """Check if the queue is empty."""
159         return len(self._queue) == 0
160     def size(self) -> int:
161         """Return the number of requests in the queue."""
162         return len(self._queue)
163     def __str__(self) -> str:
164         """Return a string representation of the queue."""
165         return " <- ".join(str(req) for req in self._queue)
166 # ----- Test Function -----
167 def test_customer_service_queue():
168     queue = CustomerServiceQueue()
```

```
Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py
167 def test_customer_service_queue():
169     print("Queue empty?", queue.is_empty())
170     print("Dequeue from empty queue:", queue.dequeue())
171     # Enqueue requests
172     queue.enqueue("Request 1")
173     queue.enqueue("Request 2")
174     queue.enqueue("Request 3")
175     print("Queue after enqueues:", queue)
176     # Dequeue requests
177     print("Dequeued:", queue.dequeue())
178     print("Queue now:", queue)
179     # Test size and empty checks
180     print("Queue size:", queue.size())
181     print("Queue empty?", queue.is_empty())
182     # Dequeue remaining requests
183     queue.dequeue()
184     queue.dequeue()
185     print("Queue after removing all requests:", queue)
186     print("Dequeue from empty queue:", queue.dequeue())
187 # Run the test
188 test_customer_service_queue()
```

OUTPUT:

```
Queue empty? True
Dequeue from empty queue: None
Queue after enqueues: Request 1 <- Request 2 <- Request 3
Dequeued: Request 1
Queue now: Request 2 <- Request 3
Queue size: 2
Queue empty? False
Queue after removing all requests:
Dequeue from empty queue: None
PS C:\Users\saite\Downloads\AI ASSISTENT CODING> 
```

EXPLANATION:

In a customer service system, a list-based queue can be implemented using a Python list where enqueue is performed with `append()` and dequeue with `pop(0)`. While adding requests is efficient ($O(1)$), removing requests from the front is slow ($O(n)$) because all remaining elements must be shifted, which can cause delays as the number of requests grows. An optimized alternative uses `collections.deque`, where both `append()` and `popleft()` operations run in $O(1)$ time. This makes deque-based queues highly efficient for FIFO operations, allowing the system to handle large numbers of service requests quickly and consistently, ensuring that requests are processed in the order they arrive without performance degradation.

Task-3: Singly Linked List for Dynamic Playlist Management

PROMPT:

Create a Python implementation of a Singly Linked List for dynamic playlist management. The class should include `insert_at_end(song)` to add a song at the end, `delete_value(song)` to remove a song by value, and `traverse()` to display all songs in order. Add inline comments explaining pointer manipulation and highlight tricky parts such as inserting at the end, deleting the head, or handling a single-node list. Also, suggest and implement test cases for edge scenarios, including an empty list, a single-node list, and deletion at the head, middle, and tail.

CODE:

```
Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py

185 #Create a Python implementation of a Singly Linked List for dynamic playlist management. The class should ir
186 from typing import Optional
187 class Node:
188     """A node in the singly linked list representing a song."""
189     def __init__(self, data: str):
190         self.data: str = data
191         self.next: Optional['Node'] = None # Pointer to the next node
192 class Playlist:
193     """Singly linked list implementation for dynamic playlist management."""
194     def __init__(self):
195         self.head: Optional[Node] = None # Start of the playlist
196     def insert_at_end(self, song: str) -> None:
197         """Insert a new song at the end of the playlist."""
198         new_node = Node(song)
199         if self.head is None:
200             # List is empty, new node becomes the head
201             self.head = new_node
202             return
203         # Traverse to the end of the List
204         current = self.head
205         while current.next:
206             current = current.next
207         # Link the Last node to the new node
208         current.next = new_node
209     def delete_value(self, song: str) -> bool:
210         """
211         Delete the first occurrence of a song from the playlist.
212         Returns True if deleted, False if not found.
213         """
214         current = self.head
215         prev = None
216         # Traverse the List to find the song
217         while current:
```

```

Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py

192 class Playlist:
209     def delete_value(self, song: str) -> bool:
218         if current.data == song:
219             if prev is None:
220                 # Deleting the head node
221                 self.head = current.next
222             else:
223                 # Skip over the current node
224                 prev.next = current.next
225             return True
226         prev = current
227         current = current.next
228     return False # Song not found
229 def traverse(self) -> None:
230     """Print all songs in the playlist in order."""
231     current = self.head
232     if not current:
233         print("Playlist is empty.")
234     return
235     songs = []
236     while current:
237         songs.append(current.data)
238         current = current.next
239     print("-> ".join(songs))
240 # ----- Test Function -----
241 def test_playlist():
242     playlist = Playlist()
243     # Edge case: traverse empty playlist
244     playlist.traverse()
245     # Insert songs
246     playlist.insert_at_end("Song A")
247     playlist.insert_at_end("Song B")
248     playlist.insert_at_end("Song C")

```

```

Lab-13.py X
C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-13.py

241 def test_playlist():
249     playlist.traverse()
250     # Delete head song
251     playlist.delete_value("Song A")
252     playlist.traverse()
253     # Delete middle song
254     playlist.delete_value("Song B")
255     playlist.traverse()
256     # Delete tail song
257     playlist.delete_value("Song C")
258     playlist.traverse()
259     # Attempt to delete from empty list
260     result = playlist.delete_value("Song X")
261     print("Delete non-existent song:", result)
262 # Run test
263 test_playlist()
264

```

OUTPUT:

```

Playlist is empty.
Song A -> Song B -> Song C
Song B -> Song C
Song C
Playlist is empty.
Delete non-existent song: False

```

PS C:\Users\saita\Downloads\AI ASSISTENT CODING>

EXPLANATION:

The Playlist class uses a singly linked list to manage songs dynamically. `insert_at_end` adds a song at the end by traversing the list and linking the new node, while `delete_value` removes a song by updating pointers, handling edge cases like deleting the head, tail, or a single-node list. `traverse` prints all songs in order, or indicates if the playlist is empty. This structure allows efficient insertion and deletion without shifting elements, and the test cases cover empty lists, single-node lists, and deletions at different positions to ensure robust functionality.

Task-4: Binary Search Tree for Fast Record Lookup PROMPT:

Complete a Python Binary Search Tree (BST) implementation for a student record system, where quick searching by roll number is needed. The class should include `insert(value)` to add a new roll number, `search(value)` to check if a roll number exists, and `inorder_traversal()` to return all roll numbers in sorted order. Add meaningful docstrings, ensure correct BST behavior, and provide a brief explanation of how BST improves search efficiency compared to linear search, including best-case and worst-case performance.

CODE:

```
Lab-13.py X
Lab-13.py > ...
258 #Complete a Python Binary Search Tree (BST) implementation for a student record system, where quick searchi
259 from typing import Optional, List
260 class Node:
261     """Node of a Binary Search Tree representing a student record."""
262     def __init__(self, value: int):
263         self.value: int = value
264         self.left: Optional['Node'] = None
265         self.right: Optional['Node'] = None
266 class BST:
267     """Binary Search Tree for fast student record lookup by roll number."""
268     def __init__(self):
269         self.root: Optional[Node] = None
270     def insert(self, value: int) -> None:
271         """Insert a value into the BST."""
272         if self.root is None:
273             self.root = Node(value)
274             return
275         self._insert_recursive(self.root, value)
276     def _insert_recursive(self, node: Node, value: int) -> None:
277         if value < node.value:
278             if node.left is None:
279                 node.left = Node(value)
280             else:
281                 self._insert_recursive(node.left, value)
282         elif value > node.value:
283             if node.right is None:
284                 node.right = Node(value)
285             else:
286                 self._insert_recursive(node.right, value)
287         # Ignore duplicates
288     def search(self, value: int) -> bool:
289         """Search for a value in the BST. Returns True if found, else False."""
290         return self._search_recursive(self.root, value)
```



```

Lab-13.py X
Lab-13.py > ...
266 class BST:
267     def _search_recursive(self, node: Optional[Node], value: int) -> bool:
268         if node is None:
269             return False
270         if value == node.value:
271             return True
272         elif value < node.value:
273             return self._search_recursive(node.left, value)
274         else:
275             return self._search_recursive(node.right, value)
276     def inorder_traversal(self) -> List[int]:
277         """Return a list of values from the BST in sorted order."""
278         result: List[int] = []
279         self._inorder_recursive(self.root, result)
280         return result
281     def _inorder_recursive(self, node: Optional[Node], result: List[int]) -> None:
282         if node:
283             self._inorder_recursive(node.left, result)
284             result.append(node.value)
285             self._inorder_recursive(node.right, result)
286 # ----- Test Function -----
287 def test_bst():
288     bst = BST()
289     # Insert roll numbers
290     for roll in [50, 30, 70, 20, 40, 60, 80]:
291         bst.insert(roll)
292     print("Inorder Traversal (sorted):", bst.inorder_traversal())
293     print("Search 40:", bst.search(40))
294     print("Search 90:", bst.search(90))
295 test_bst()
296

```

OUTPUT:

```

Inorder Traversal (sorted): [20, 30, 40, 50, 60, 70, 80]
Search 40: True
Search 90: False
PS C:\Users\saita\Downloads\AI ASSISTENT CODING> 

```

EXPLANATION:

The BST stores student roll numbers so that each node's left child has smaller values and the right child has larger values. This structure allows search to skip half of the remaining nodes at each step, improving efficiency over linear search. In the best case, the BST is balanced, giving $O(\log n)$ search time, while in the worst case, it becomes skewed like a linked list, leading to $O(n)$ search time. Inorder traversal returns all records in sorted order, making BST both efficient and ordered for record lookup.

Task-5: Graph Traversal for Social Network Connections

PROMPT:

Implement a Python graph using an adjacency list to model a social network where users are connected to friends. Include Breadth-First Search (BFS) to find nearby connections and Depth-First Search (DFS) to explore deeper connection paths. Add

inline comments explaining traversal steps, compare recursive and iterative DFS approaches, and suggest practical use cases for BFS versus DFS.

CODE:

```
Lab-13.py X
Lab-13.py > ...

317 #Implement a Python graph using an adjacency List to model a social network where users are connected to fri
318 from collections import deque
319 from typing import Dict, List, Set
320 class SocialNetwork:
321     """Graph representation of a social network using adjacency list."""
322     def __init__(self):
323         self.graph: Dict[str, List[str]] = {}
324     def add_user(self, user: str) -> None:
325         """Add a user to the network."""
326         if user not in self.graph:
327             self.graph[user] = []
328     def add_connection(self, user1: str, user2: str) -> None:
329         """Add a bidirectional friendship connection."""
330         self.graph.setdefault(user1, []).append(user2)
331         self.graph.setdefault(user2, []).append(user1)
332     def bfs(self, start: str) -> List[str]:
333         """Breadth-First Search to find nearby connections."""
334         visited: Set[str] = set[str]()
335         queue: deque[str] = deque[str]([start])
336         result: List[str] = []
337         while queue:
338             user = queue.popleft() # Explore the front of the queue
339             if user not in visited:
340                 visited.add(user)
341                 result.append(user)
342                 # Add all unvisited friends to the queue
343                 for friend in self.graph.get(user, []):
344                     if friend not in visited:
345                         queue.append(friend)
346         return result
347     def dfs_recursive(self, start: str) -> List[str]:
348         """Depth-First Search (recursive) to explore deep connections."""
349         visited: Set[str] = set[str]()
```

```
Lab-13.py X
Lab-13.py > ...

320 class SocialNetwork:
347     def dfs_recursive(self, start: str) -> List[str]:
350         result: List[str] = []
351     def dfs(self, user: str):
352         if user not in visited:
353             visited.add(user)
354             result.append(user)
355             for friend in self.graph.get(user, []):
356                 dfs(friend)
357         dfs(start)
358         return result
359     def dfs_iterative(self, start: str) -> List[str]:
360         """Depth-First Search (iterative using stack)."""
361         visited: Set[str] = set[str]()
362         stack: List[str] = [start]
363         result: List[str] = []
364         while stack:
365             user = stack.pop() # Explore Last added node
366             if user not in visited:
367                 visited.add(user)
368                 result.append(user)
369                 # Add friends to stack (reversed for consistent order)
370                 for friend in reversed(self.graph.get(user, [])):
371                     if friend not in visited:
372                         stack.append(friend)
373         return result
374 # ----- Test Function -----
375 def test_social_network():
376     sn = SocialNetwork()
377     users = ["Alice", "Bob", "Charlie", "David", "Eve"]
378     for user in users:
379         sn.add_user(user)
380     connections = [("Alice", "Bob"), ("Alice", "Charlie"), ("Bob", "David"), ("Charlie", "Eve")]
```

```
Lab-13.py X
Lab-13.py > ...

375 def test_social_network():
381     for u1, u2 in connections:
382         sn.add_connection(u1, u2)
383     print("BFS from Alice:", sn.bfs("Alice"))
384     print("DFS Recursive from Alice:", sn.dfs_recursive("Alice"))
385     print("DFS Iterative from Alice:", sn.dfs_iterative("Alice"))
386 test_social_network()
```

OUTPUT:

```
BFS from Alice: ['Alice', 'Bob', 'Charlie', 'David', 'Eve']  
DFS Recursive from Alice: ['Alice', 'Bob', 'David', 'Charlie', 'Eve']  
DFS Iterative from Alice: ['Alice', 'Bob', 'David', 'Charlie', 'Eve']  
PS C:\Users\saita\Downloads\AI ASSISTENT CODING> 
```

EXPLANATION:

The graph uses an adjacency list to represent user connections. BFS explores nearby friends level by level, making it ideal for finding shortest paths or friends-of-friends. DFS explores as deep as possible along each path, either recursively or iteratively with a stack, useful for tracing long connection chains or detecting cycles. Recursive DFS is simpler to write, while iterative DFS avoids recursion limits. BFS is practical for recommendations or network reach, whereas DFS is better for in-depth network exploration or backtracking tasks.