

AI ASSISTED CODING

LAB-11: *Data Structures with AI: Implementing Fundamental Structures* *Lab Objectives*

Roll no: 2503A51L44

Name: Meer Burhan Ali Hashmi

Batch: 20

Task 1 Description: Implementing a Stack (LIFO)

- **Task:** Use AI to help implement a Stack class in Python with the following operations: push(), pop(), peek(), and is_empty().
- **Instructions:**
 - Ask AI to generate code skeleton with docstrings.
 - Test stack operations using sample data.
 - Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque)

Prompt:

Implement a Stack class in Python with the following operations: push(), pop(), peek(), and is_empty().

- **Instructions:**
 - Generate code skeleton with docstrings.
 - Test stack operations using sample data.
 - Suggest optimizations or alternative implementations (e.g., using collections.deque).
 - Modify the output so that pushed items pop items peek item and is empty functions are also displayed

Code Generated:

```

task3-11.py • task4-11.py • task5-11.py • task2-11.py • task1-11.py •
assignment-11 > task1-11.py > ...
1
2 class Stack:
3     """
4     A simple Stack implementation using a Python list.
5     Supports push, pop, peek, and is_empty operations.
6     """
7     def __init__(self):
8         """Initialize an empty stack."""
9         self._items = []
10
11     def push(self, item):
12         """Add an item to the top of the stack."""
13         self._items.append(item)
14
15     def pop(self):
16         """Remove and return the top item of the stack. Raises IndexError if empty."""
17         if self.is_empty():
18             raise IndexError("pop from empty stack")
19         return self._items.pop()
20
21     def peek(self):
22         """Return the top item of the stack without removing it. Raises IndexError if empty."""
23         if self.is_empty():
24             raise IndexError("peek from empty stack")
25         return self._items[-1]
26
27     def is_empty(self):
28         """Return True if the stack is empty, False otherwise."""
29         return len(self._items) == 0
30
31
32 if __name__ == "__main__":
33     # Test stack operations with sample data
34     stack = Stack()
35     print("Is stack empty?", stack.is_empty())
36
37     # Push items
38     for item in [10, 20, 30]:
39         stack.push(item)
40         print(f"Pushed {item}: Current stack = {stack._items}")
41
42     # Peek at the top item
43     print(f"Peeked item: {stack.peek()}")
44
45     # Pop items
46     while not stack.is_empty():
47         popped = stack.pop()
48         print(f"Popped {popped}: Current stack = {stack._items}")
49
50     # Check if stack is empty
51     print("Is stack empty after pops?", stack.is_empty())
52
53     # Push another item
54     stack.push(40)
55     print(f"Pushed 40: Current stack = {stack._items}")
56
57     # Pop again
58     print(f"Popped {stack.pop()}: Current stack = {stack._items}")
59
60     # Uncomment below to see error handling
61     print(stack.pop()) # Should raise IndexError
62
63     # Optimization/Alternative Suggestion:
64     # For better performance with large stacks (especially if you need O(1) pops from both ends),
65     # consider using collections.deque instead of list:
66     # from collections import deque
67     # class Stack:
68     #     def __init__(self):
69     #         self._items = deque()
70     #     ...|
71

```

Output:

```

PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\AIAssistedCoding>
Is stack empty? True
Pushed 10: Current stack = [10]
Pushed 20: Current stack = [10, 20]
Pushed 30: Current stack = [10, 20, 30]
Peeked item: 30
Popped 30: Current stack = [10, 20]
Popped 20: Current stack = [10]
Popped 10: Current stack = []
Is stack empty after pops? True
Pushed 40: Current stack = [40]
Popped 40: Current stack = []

```

Observation:

The stack implementation correctly demonstrated push, pop, peek, and is_empty operations. The output validated LIFO behavior, where the last inserted element was removed first. The AI-assisted skeleton made the logic clear and structured. This shows how AI can speed up coding while ensuring correctness.

Task-2 Description: Queue Implementation with Performance Review

- Task: 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-1: Implement a Queue with enqueue(), dequeue(), and is_empty() methods. First, implement using Python lists.

Code Generated-1:

```
task3-11.py • task4-11.py • task5-11.py • task2-11.py • task1-11.py •
assignment-11 > task2-11.py > _
1  import time
2  class ListQueue:
3      """
4      Queue implementation using a Python list.
5      Enqueue is O(1), but dequeue (pop(0)) is O(n).
6      """
7      def __init__(self):
8          self._items = []
9
10     def enqueue(self, item):
11         self._items.append(item)
12
13     def dequeue(self):
14         if self.is_empty():
15             raise IndexError("dequeue from empty queue")
16         return self._items.pop(0)
17
18     def is_empty(self):
19         return len(self._items) == 0
20
21 if __name__ == "__main__":
22
23     print("\n--- List-based Queue Demo (pop(0) is O(n)) ---")
24     lq = ListQueue()
25     print("Is queue empty?", lq.is_empty())
26     for item in [1, 2, 3]:
27         lq.enqueue(item)
28         print(f"Enqueued {item}: Current queue = {lq._items}")
29     while not lq.is_empty():
30         removed = lq.dequeue()
31         print(f"Dequeued {removed}: Current queue = {lq._items}")
32     print("Is queue empty after dequeues?", lq.is_empty())
33
34     # Performance demonstration with large data set
35     N = 10**5
36     lq = ListQueue()
37     start = time.time()
38     for i in range(N):
39         lq.enqueue(i)
40     for i in range(N):
41         lq.dequeue()
42     elapsed_list = time.time() - start
43     print(f"ListQueue: Time for 10^5 enqueue+dequeue: {elapsed_list:.4f} seconds")
44
45     # Note:
46     # - ListQueue: dequeue is O(n) (slow for large queues)
47     # - DequeueQueue: dequeue is O(1) (fast for all sizes)
```

Output:

```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\AIAssignment-11\task2-11.py"

--- List-based Queue Demo (pop(0) is O(n)) ---
Is queue empty? True
Enqueued 1: Current queue = [1]
Enqueued 2: Current queue = [1, 2]
Enqueued 3: Current queue = [1, 2, 3]
Dequeued 1: Current queue = [2, 3]
Dequeued 2: Current queue = [3]
Dequeued 3: Current queue = []
Is queue empty after dequeues? True
ListQueue: Time for 10^5 enqueue+dequeue: 1.0647 seconds
```

Observation: The queue implemented using lists worked but showed performance limits during repeated dequeues. The optimized version using collections.deque improved efficiency, especially for front removals. Output confirmed proper FIFO behavior, highlighting AI's role in suggesting better data structures.

Prompt-2: Review performance and suggest a more efficient implementation (using collections.deque). Perform any operation in this code so that it shows that it is different as compared to the list.

Code Generated-2:

```
task3-11.py • task4-11.py • task5-11.py • task2-11.py • task1-11.py •
assignment-11 > task2-11.py > ...
1  # --- Deque-based Queue (Efficient) ---
2  import time
3  from collections import deque
4  class DequeQueue:
5      """
6      Queue implementation using collections.deque.
7      Both enqueue and dequeue are O(1).
8      """
9      def __init__(self):
10         self._items = deque()
11
12     def enqueue(self, item):
13         self._items.append(item)
14
15     def dequeue(self):
16         if self.is_empty():
17             raise IndexError("dequeue from empty queue")
18         return self._items.popleft()
19
20     def is_empty(self):
21         return len(self._items) == 0
22
23
24 if __name__ == "__main__":
25     print("\n--- Deque-based Queue Demo (popleft() is O(1)) ---")
26     dq = DequeQueue()
27     print("Is queue empty?", dq.is_empty())
28     for item in [1, 2, 3]:
29         dq.enqueue(item)
30         print(f"Enqueued {item}: Current queue = {list(dq._items)}")
31     while not dq.is_empty():
32         removed = dq.dequeue()
33         print(f"Dequeued {removed}: Current queue = {list(dq._items)}")
34     print("Is queue empty after dequeues?", dq.is_empty())
35
36     dq = DequeQueue()
37     start = time.time()
38     for i in range(10):
39         dq.enqueue(i)
40     for i in range(10):
41         dq.dequeue()
42     elapsed_deque = time.time() - start
43     print(f"DequeQueue: Time for 10^5 enqueue+dequeue: {elapsed_deque:.4f} seconds")
44
45 # Note:
46 # - ListQueue: dequeue is O(n) (slow for large queues)
47 # - DequeQueue: dequeue is O(1) (fast for all sizes)
```

Output:

```
--- Deque-based Queue Demo (popleft() is O(1)) ---
Is queue empty? True
Enqueued 1: Current queue = [1]
Enqueued 2: Current queue = [1, 2]
Enqueued 3: Current queue = [1, 2, 3]
Dequeued 1: Current queue = [2, 3]
Dequeued 2: Current queue = [3]
Dequeued 3: Current queue = []
Is queue empty after dequeues? True
DequeQueue: Time for 10^5 enqueue+dequeue: 0.0449 seconds
```

Observation: The queue

implemented using lists worked but showed performance limits during repeated dequeues. The optimized version using collections.deque improved efficiency, especially for front removals. Output confirmed proper FIFO behavior, highlighting AI's role in suggesting better data structures.

Task-3 Description: Singly Linked List with Traversal

- **Task:** Implement a Singly Linked List with operations: insert_at_end(), 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 Linked List with operations: insert_at_end(), delete_value(), and traverse().

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

Code Generated:

```
task3-11.py x task4-11.py task5-11.py task2-11.py task1-11.py
assignment-11 > task3-11.py > _
1
2 class Node:
3     """A node in a singly linked list."""
4     def __init__(self, data):
5         self.data = data
6         self.next = None
7
8
9 class LinkedList:
10     """Singly linked list with basic operations."""
11     def __init__(self):
12         self.head = None
13
14     def insert_at_end(self, data):
15         """Insert a new node with the given data at the end of the list."""
16         new_node = Node(data)
17         if not self.head:
18             self.head = new_node
19             return
20         current = self.head
21         while current.next:
22             current = current.next
23         # current.next is None, so we attach the new node here
24         current.next = new_node
25
26     def delete_value(self, value):
27         """Delete the first node with the specified value."""
28         current = self.head
29         prev = None
30         while current:
31             if current.data == value:
32                 if prev:
33                     # Bypass the current node
34                     prev.next = current.next
35                 else:
36                     # Deleting the head node
37                     self.head = current.next
38                 return True # Value found and deleted
39             prev = current
40             current = current.next
41         return False # Value not found
42
```

```
42
43     def traverse(self):
44         """Return a list of all elements in the linked list."""
45         elements = []
46         current = self.head
47         while current:
48             elements.append(current.data)
49             current = current.next
50         return elements
51
52 if __name__ == "__main__":
53     # Test cases for LinkedList
54     ll = LinkedList()
55     print("Initial list (should be empty):", ll.traverse())
56
57     # Insert at end
58     ll.insert_at_end(10)
59     ll.insert_at_end(20)
60     ll.insert_at_end(30)
61     print("After inserting 10, 20, 30:", ll.traverse())
62
63     # Delete head
64     ll.delete_value(10)
65     print("After deleting 10 (head):", ll.traverse())
66
67     # Delete middle
68     ll.delete_value(20)
69     print("After deleting 20 (middle):", ll.traverse())
70
71     # Delete tail
72     ll.delete_value(30)
73     print("After deleting 30 (tail):", ll.traverse())
74
75     # Try deleting from empty list
76     print("Delete from empty list (should be False):", ll.delete_value(100))
77
78     # Insert again and delete non-existent value
79     ll.insert_at_end(40)
80     print("After inserting 40:", ll.traverse())
81     print("Delete non-existent value (should be False):", ll.delete_value(99))
82     print("Final list:", ll.traverse())
83
84 # Suggested test cases:
85 # - Insert multiple elements and check order with traverse()
86 # - Delete head, middle, and tail values
87 # - Delete a value not in the list (should return False)
88 # - Delete from an empty list (should return False)
89 # - Traverse an empty list (should return [])
```

Output:

```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\U
sers/Admin/OneDrive/Documents/Custom Office Templates/Desktop/
Initial list (should be empty): []
After inserting 10, 20, 30: [10, 20, 30]
After deleting 10 (head): [20, 30]
After deleting 20 (middle): [30]
After deleting 30 (tail): []
Delete from empty list (should be False): False
After inserting 40: [40]
Delete non-existent value (should be False): False
Final list: [40]
```

Observation: The linked list supported insertion, deletion, and traversal operations successfully. Pointer updates were explained clearly with inline AI-generated comments, aiding conceptual understanding. Test cases confirmed robustness against different scenarios like deleting head or absent values.

Task-4 Description: Binary Search Tree (BST)

- **Task:** 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.

Code given to AI:

```
task4-11.py X
assignment-11 > task4-11.py > BST > inorder
1  # Node class for the BST
2  class Node:
3      def __init__(self, value):
4          self.value = value
5          self.left = None # left child
6          self.right = None # right child
7
8
9  # Binary Search Tree class
10 class BST:
11     def __init__(self):
12         self.root = None # start with empty tree
13
14     def insert(self, value):
15         # Write code to insert value into the tree
16         pass
17
18     def search(self, value):
19         # Write code to search for value in the tree
20         pass
21
22     def inorder(self):
23         # Write code to print values in inorder (L → Root → R)
24         pass
25
```

Prompt: Complete missing methods and add docstrings. Test with a list of integers and compare outputs of search() for present vs absent elements.

Code Generated:

```
task4-11.py X
assignment-11 > task4-11.py > ...
1  # Node class for the BST
2  class Node:
3      def __init__(self, value):
4          self.value = value
5          self.left = None # left child
6          self.right = None # right child
7
8
9  # Binary Search Tree class
10 class BST:
11     def __init__(self):
12         self.root = None # start with empty tree
13
14
15     def insert(self, value):
16         """
17         Insert a value into the BST.
18         """
19         def _insert(node, value):
20             if node is None:
21                 return Node(value)
22             if value < node.value:
23                 node.left = _insert(node.left, value)
24             elif value > node.value:
25                 node.right = _insert(node.right, value)
26             return node
27         self.root = _insert(self.root, value)
28
29
30     def search(self, value):
31         """
32         Search for a value in the BST. Returns True if found, else False.
33         """
34         def _search(node, value):
35             if node is None:
36                 return False
37             if value == node.value:
38                 return True
39             elif value < node.value:
40                 return _search(node.left, value)
41             else:
42                 return _search(node.right, value)
43         return _search(self.root, value)
44
45
46     def inorder(self):
47         """
48         Return a list of values from the BST in inorder (L → Root → R).
49         """
50         result = []
51         def _inorder(node):
52             if node:
53                 _inorder(node.left)
54                 result.append(node.value)
55                 _inorder(node.right)
56         _inorder(self.root)
57         return result
58
59
60 # Test code
61 if __name__ == "__main__":
62     bst = BST()
63     values = [15, 10, 20, 8, 12, 17, 25]
64     for v in values:
65         bst.insert(v)
66     print("Inorder traversal:", bst.inorder())
67
68     # Test search for present and absent elements
69     test_values = [10, 25, 13, 30]
70     for val in test_values:
71         found = bst.search(val)
72         print(f"Search for {val}: {'Found' if found else 'Not found'}")
73
```

Output:

```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\
assignment-11/task4-11.py"
Inorder traversal: [8, 10, 12, 15, 17, 20, 25]
Search for 10: Found
Search for 25: Found
Search for 13: Not found
Search for 30: Not found
```

Observation:

The BST allowed insertion, searching, and inorder traversal, producing sorted output as expected. Searches returned correct results for both present and absent values. The AI-generated partial class guided step-by-step completion, reinforcing understanding of recursive structures.

Task-5 Description:Graph Representation and BFS/DFS Traversal

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

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

Code Generated:

```
task3-11.py • task4-11.py • task5-11.py • task2-11.py • task1-11.py •
assignment-11 > task5-11.py > ...
1 from collections import deque
2 class Graph:
3     """
4     Graph implemented using an adjacency list.
5     Supports BFS and both recursive and iterative DFS traversals.
6     """
7     def __init__(self):
8         self.adj = {} # adjacency list: node -> list of neighbors
9
10    def add_edge(self, u, v):
11        """Add an edge from u to v (undirected by default)."""
12        if u not in self.adj:
13            self.adj[u] = []
14        if v not in self.adj:
15            self.adj[v] = []
16        self.adj[u].append(v)
17        self.adj[v].append(u)
18
19    def bfs(self, start):
20        """Breadth-First Search traversal from start node."""
21        visited = set()
22        queue = deque([start])
23        order = []
24        while queue:
25            node = queue.popleft()
26            if node not in visited:
27                order.append(node)
28                visited.add(node)
29                # Add all unvisited neighbors to the queue
30                for neighbor in self.adj.get(node, []):
31                    if neighbor not in visited:
32                        queue.append(neighbor)
33        return order
34
35    def dfs_recursive(self, start):
36        """Recursive Depth-First Search traversal from start node."""
37        order = []
38        visited = set()
39        def dfs(node):
```

```
39         def dfs(node):
40             visited.add(node)
41             order.append(node)
42             for neighbor in self.adj.get(node, []):
43                 if neighbor not in visited:
44                     dfs(neighbor)
45         dfs(start)
46         return order
47
48    def dfs_iterative(self, start):
49        """Iterative Depth-First Search traversal from start node."""
50        visited = set()
51        stack = [start]
52        order = []
53        while stack:
54            node = stack.pop()
55            if node not in visited:
56                order.append(node)
57                visited.add(node)
58                # Add neighbors in reverse order for correct traversal order
59                for neighbor in reversed(self.adj.get(node, [])):
60                    if neighbor not in visited:
61                        stack.append(neighbor)
62        return order
63
64    if __name__ == "__main__":
65        # Example usage and test
66        g = Graph()
67        g.add_edge('A', 'B')
68        g.add_edge('A', 'C')
69        g.add_edge('B', 'D')
70        g.add_edge('C', 'E')
71        g.add_edge('D', 'E')
72        g.add_edge('E', 'F')
73
74        print("Adjacency List:", g.adj)
75        print("BFS from A:", g.bfs('A'))
76        print("DFS (recursive) from A:", g.dfs_recursive('A'))
77        print("DFS (iterative) from A:", g.dfs_iterative('A'))
78
79        # Comparison:
80        # - Recursive DFS is elegant and easy to read, but may hit recursion limits for very deep graphs.
81        # - Iterative DFS avoids recursion depth issues and is more suitable for large/deep graphs.
82
```

Output:

```
PS C:\Users\Admin\OneDrive\Documents\Custom Office Templates\Desktop\AIAssistedCoding> & C:/Users/Admin/anaconda3/python.exe  
assignment-11/task5-11.py"  
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']  
DFS (recursive) from A: ['A', 'B', 'D', 'E', 'C', 'F']  
DFS (iterative) from A: ['A', 'B', 'D', 'E', 'C', 'F']
```

Observation: Graph traversal using BFS and DFS was implemented with adjacency lists. The outputs validated correct order of node visits for both search strategies. Recursive vs iterative DFS approaches were highlighted, showing AI's role in presenting alternative solutions.