# lab Assesment-11.4

* task 1: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is\_empty().
* Prompt: help me to implement a Stack class in Python with the following operations: push(), pop(), peek(), and is\_empty() ,generate code skeleton with docstrings
* Code:
* class Stack:
* def \_\_init\_\_(self):
* """Initializes an empty stack."""
* pass # TODO: Initialize the stack
* def is\_empty(self):
* """Returns True if the stack is empty, False otherwise."""
* pass # TODO: Implement the check for an empty stack
* def push(self, item):
* """Adds an item to the top of the stack."""
* pass # TODO: Implement adding an item
* def pop(self):
* """Removes and returns the item from the top of the stack.
* Raises IndexError if the stack is empty."""
* pass # TODO: Implement removing and returning the top item
* def peek(self):
* """Returns the item at the top of the stack without removing it.
* Raises IndexError if the stack is empty."""
* pass # TODO: Implement returning the top item without removing it
* print(f"Is the stack empty initially? {my\_stack.is\_empty()}")
* # Assuming the Stack class from the previous cell is defined
* # Create a Stack instance
* my\_stack = Stack()
* # Test is\_empty()
* print(f"Is the stack empty initially? {my\_stack.is\_empty()}")
* # Test push()
* my\_stack.push(10)
* my\_stack.push(20)
* my\_stack.push(30)
* print(f"Is the stack empty after pushing elements? {my\_stack.is\_empty()}")
* # Test peek()
* print(f"The top element is: {my\_stack.peek()}")
* # Test pop()
* popped\_item = my\_stack.pop()
* print(f"Popped element: {popped\_item}")
* print(f"The top element after popping is: {my\_stack.peek()}")
* popped\_item = my\_stack.pop()
* print(f"Popped element: {popped\_item}")
* print(f"The top element after popping is: {my\_stack.peek()}")
* popped\_item = my\_stack.pop()
* print(f"Popped element: {popped\_item}")
* # Test is\_empty() after popping all elements
* print(f"Is the stack empty after popping all elements? {my\_stack.is\_empty()}")
* # Test popping from an empty stack (should raise IndexError)
* try:
* my\_stack.pop()
* except IndexError as e:
* print(f"Caught expected error when popping from empty stack: {e}")
* # Test peeking from an empty stack (should raise IndexError)
* try:
* my\_stack.peek()
* except IndexError as e:
* print(f"Caught expected error when peeking from empty stack: {e}")
* Output:
* Is the stack empty initially? None
* Is the stack empty after pushing elements? None
* The top element is: None
* Popped element: None
* The top element after popping is: None
* Popped element: None
* The top element after popping is: None
* Popped element: None
* Is the stack empty after popping all elements? None
* Task 2: Implement a **Queue** with enqueue(), dequeue(), and is\_empty() methods
* Prompt: : Implement a **Queue** with enqueue(), dequeue(), and is\_empty() methods
* Output:
* class Queue:
* def \_\_init\_\_(self):
* """Initializes an empty queue."""
* self.\_items = []
* def is\_empty(self):
* """Returns True if the queue is empty, False otherwise."""
* return not self.\_items
* def enqueue(self, item):
* """Adds an item to the rear of the queue."""
* self.\_items.append(item)
* def dequeue(self):
* """Removes and returns the item from the front of the queue.
* Raises IndexError if the queue is empty."""
* if not self.\_items:
* raise IndexError("dequeue from empty queue")
* return self.\_items.pop(0) # Remove from the front
* # Example usage:
* queue = Queue()
* print(f"Is queue empty? {queue.is\_empty()}")
* queue.enqueue(10)
* queue.enqueue(20)
* queue.enqueue(30)
* print(f"Is queue empty? {queue.is\_empty()}")
* print(f"Dequeued element: {queue.dequeue()}")
* print(f"Dequeued element: {queue.dequeue()}")
* print(f"Dequeued element: {queue.dequeue()}")
* print(f"Is queue empty? {queue.is\_empty()}")
* # Test dequeue from empty queue (should raise IndexError)
* try:
* queue.dequeue()
* except IndexError as e:
* print(f"Caught expected error when dequeuing from empty queue: {e}")
* Is queue empty? True
* Is queue empty? False
* Dequeued element: 10
* Dequeued element: 20
* Dequeued element: 30
* Is queue empty? True
* Caught expected error when dequeuing from empty queue: dequeue from empty queue
* Output(enhanced):

from collections import deque

class DequeQueue:

    def \_\_init\_\_(self):

        """Initializes an empty queue using a deque."""

        self.\_items = deque()

    def is\_empty(self):

        """Returns True if the queue is empty, False otherwise."""

        return not self.\_items

    def enqueue(self, item):

        """Adds an item to the rear of the queue."""

        self.\_items.append(item) # Add to the right end

    def dequeue(self):

        """Removes and returns the item from the front of the queue.

        Raises IndexError if the queue is empty."""

        if not self.\_items:

            raise IndexError("dequeue from empty queue")

        return self.\_items.popleft() # Remove from the left end

# Example usage:

deque\_queue = DequeQueue()

print(f"Is deque queue empty? {deque\_queue.is\_empty()}")

deque\_queue.enqueue(100)

deque\_queue.enqueue(200)

deque\_queue.enqueue(300)

print(f"Is deque queue empty? {deque\_queue.is\_empty()}")

print(f"Dequeued element: {deque\_queue.dequeue()}")

print(f"Dequeued element: {deque\_queue.dequeue()}")

print(f"Dequeued element: {deque\_queue.dequeue()}")

print(f"Is deque queue empty? {deque\_queue.is\_empty()}")

# Test dequeue from empty queue (should raise IndexError)

try:

    deque\_queue.dequeue()

except IndexError as e:

    print(f"Caught expected error when dequeuing from empty deque queue: {e}")

* Output:

Is deque queue empty? True

Is deque queue empty? False

Dequeued element: 100

Dequeued element: 200

Dequeued element: 300

Is deque queue empty? True

Caught expected error when dequeuing from empty deque queue: dequeue from empty queue

* Task3: Implement a **Singly Linked List** with operations: insert\_at\_end(), delete\_value(), and traverse().
* Prompt: Implement a Singly Linked List with operations: insert\_at\_end(), delete\_value(), and traverse(). Start with a simple class-based implementation (Node, LinkedList).
* Output:
* class Node:
* """Represents a node in a singly linked list."""
* def \_\_init\_\_(self, data=None):
* self.data = data
* self.next = None
* class LinkedList:
* """Represents a singly linked list."""
* def \_\_init\_\_(self):
* self.head = None
* def insert\_at\_end(self, data):
* """Inserts a new node with the given data at the end of the linked list."""
* new\_node = Node(data)
* if not self.head:
* self.head = new\_node
* return
* last\_node = self.head
* while last\_node.next:
* last\_node = last\_node.next
* last\_node.next = new\_node
* def delete\_value(self, value):
* """Deletes the first node with the given value from the linked list."""
* if not self.head:
* print("List is empty. Cannot delete.")
* return
* # If the head node contains the value to be deleted
* if self.head.data == value:
* self.head = self.head.next
* return
* current\_node = self.head
* while current\_node.next and current\_node.next.data != value:
* current\_node = current\_node.next
* # If the value was found and it's not the head node
* if current\_node.next:
* current\_node.next = current\_node.next.next
* else:
* print(f"Value {value} not found in the list.")
* def traverse(self):
* """Traverses the linked list and prints the data of each node."""
* current\_node = self.head
* if not current\_node:
* print("Linked list is empty.")
* return
* print("Linked List:", end=" ")
* while current\_node:
* print(current\_node.data, end=" -> ")
* current\_node = current\_node.next
* print("None") # Indicate the end of the list
* # Assuming the LinkedList class from the previous cell is defined
* # Create a LinkedList instance
* my\_list = LinkedList()
* # Test insert\_at\_end()
* my\_list.insert\_at\_end(1)
* my\_list.insert\_at\_end(2)
* my\_list.insert\_at\_end(3)
* my\_list.insert\_at\_end(4)
* # Test traverse()
* my\_list.traverse()
* # Test delete\_value()
* print("\nDeleting value 3:")
* my\_list.delete\_value(3)
* my\_list.traverse()
* print("\nDeleting value 1:")
* my\_list.delete\_value(1)
* my\_list.traverse()
* print("\nDeleting value 5 (not in the list):")
* my\_list.delete\_value(5)
* my\_list.traverse()
* print("\nDeleting value 4:")
* my\_list.delete\_value(4)
* my\_list.traverse()
* print("\nDeleting from an empty list:")
* my\_list.delete\_value(10)
* my\_list.traverse()

output:

Linked List: 1 -> 2 -> 3 -> 4 -> None

Deleting value 3:

Linked List: 1 -> 2 -> 4 -> None

Deleting value 1:

Linked List: 2 -> 4 -> None

Deleting value 5 (not in the list):

Value 5 not found in the list.

Linked List: 2 -> 4 -> None

Deleting value 4:

Linked List: 2 -> None

Deleting from an empty list:

Value 10 not found in the list.

Linked List: 2 -> None

* Task4: : Implement a **Binary Search Tree** with methods for insert(), search(), and inorder\_traversal().
* Prompt: : Implement a **Binary Search Tree** with methods for insert(), search(), and inorder\_traversal().complete missing methods and add docstrings.generate a code to run this task
* Code:
* class TreeNode:
* """Represents a node in a Binary Search Tree."""
* def \_\_init\_\_(self, key):
* self.key = key
* self.left = None
* self.right = None
* class BinarySearchTree:
* """Represents a Binary Search Tree."""
* def \_\_init\_\_(self):
* self.root = None
* def insert(self, key):
* """Inserts a new node with the given key into the BST."""
* if self.root is None:
* self.root = TreeNode(key)
* else:
* self.\_insert\_recursive(self.root, key)
* def \_insert\_recursive(self, node, key):
* if key < node.key:
* if node.left is None:
* node.left = TreeNode(key)
* else:
* self.\_insert\_recursive(node.left, key)
* elif key > node.key:
* if node.right is None:
* node.right = TreeNode(key)
* else:
* self.\_insert\_recursive(node.right, key)
* else:
* # Value already exists, do nothing
* pass
* def search(self, key):
* """Searches for a node with the given key in the BST."""
* return self.\_search\_recursive(self.root, key)
* def \_search\_recursive(self, node, key):
* if node is None or node.key == key:
* return node
* if key < node.key:
* return self.\_search\_recursive(node.left, key)
* else:
* return self.\_search\_recursive(node.right, key)
* def inorder\_traversal(self):
* """Performs an in-order traversal of the BST and prints the keys."""
* result = []
* self.\_inorder\_recursive(self.root, result)
* return result
* def \_inorder\_recursive(self, node, result):
* if node:
* self.\_inorder\_recursive(node.left, result)
* result.append(node.key)
* self.\_inorder\_recursive(node.right, result)
* # Test the Binary Search Tree implementation
* bst = BinarySearchTree()
* # Insert elements
* elements\_to\_insert = [50, 30, 70, 20, 40, 60, 80]
* print("Inserting elements:", elements\_to\_insert)
* for element in elements\_to\_insert:
* bst.insert(element)
* # Perform inorder traversal
* print("\nIn-order traversal:")
* inorder\_result = bst.inorder\_traversal()
* print(inorder\_result)
* # Test search() for present and absent elements
* print("\nTesting search():")
* elements\_to\_search = [40, 90, 20, 55, 80]
* for element in elements\_to\_search:
* found\_node = bst.search(element)
* if found\_node:
* print(f"Element {element} found in the tree.")
* else:
* print(f"Element {element} not found in the tree.")
* Output:
* Inserting elements: [50, 30, 70, 20, 40, 60, 80]
* In-order traversal:
* [20, 30, 40, 50, 60, 70, 80]
* Testing search():
* Element 40 found in the tree.
* Element 90 not found in the tree.
* Element 20 found in the tree.
* Element 55 not found in the tree.
* Element 80 found in the tree.
* Task5: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().
* Prompt: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().generate BFS and DFS implementations with inline comments.
* Code:
* from collections import deque
* class Graph:
* def \_\_init\_\_(self, adjacency\_list):
* """Initializes a graph with the given adjacency list."""
* self.adjacency\_list = adjacency\_list
* def get\_neighbors(self, node):
* """Returns the neighbors of a given node."""
* return self.adjacency\_list.get(node, [])
* def bfs(self, start\_node):
* """
* Performs a Breadth-First Search starting from the start\_node.
* Prints the nodes visited during traversal.
* """
* visited = set()  # Set to keep track of visited nodes
* queue = deque([start\_node])  # Initialize a queue with the start node
* visited.add(start\_node)  # Mark the start node as visited
* print("BFS Traversal:")
* while queue:
* current\_node = queue.popleft()  # Dequeue a node from the front
* print(current\_node, end=" ")  # Print the current node
* # Get all adjacent nodes of the dequeued node
* for neighbor in self.get\_neighbors(current\_node):
* # If a neighbor has not been visited, mark it visited and enqueue it
* if neighbor not in visited:
* visited.add(neighbor)
* queue.append(neighbor)
* print("\n")
* def dfs(self, start\_node):
* """
* Performs a Depth-First Search starting from the start\_node (iterative approach).
* Prints the nodes visited during traversal.
* """
* visited = set()  # Set to keep track of visited nodes
* stack = [start\_node]  # Initialize a stack with the start node
* print("DFS Traversal (Iterative):")
* while stack:
* current\_node = stack.pop()  # Pop a node from the top of the stack
* if current\_node not in visited:
* print(current\_node, end=" ")  # Print the current node if not visited
* visited.add(current\_node)  # Mark the current node as visited
* # Get all adjacent nodes (in reverse order for consistent output with recursive DFS)
* # Add unvisited neighbors to the stack
* for neighbor in reversed(self.get\_neighbors(current\_node)):
* if neighbor not in visited:
* stack.append(neighbor)
* print("\n")
* def dfs\_recursive(self, start\_node, visited=None):
* """
* Performs a Depth-First Search starting from the start\_node (recursive approach).
* Prints the nodes visited during traversal.
* """
* if visited is None:
* visited = set()
* if start\_node not in visited:
* print(start\_node, end=" ")  # Print the current node if not visited
* visited.add(start\_node)  # Mark the current node as visited
* # Recursively visit neighbors
* for neighbor in self.get\_neighbors(start\_node):
* self.dfs\_recursive(neighbor, visited)
* # Example Usage:
* # Representing a graph using an adjacency list
* adjacency\_list = {
* 'A': ['B', 'C'],
* 'B': ['A', 'D', 'E'],
* 'C': ['A', 'F'],
* 'D': ['B'],
* 'E': ['B', 'F'],
* 'F': ['C', 'E']
* }
* graph = Graph(adjacency\_list)
* # Perform BFS traversal
* graph.bfs('A')
* # Perform iterative DFS traversal
* graph.dfs('A')
* # Perform recursive DFS traversal
* print("DFS Traversal (Recursive):")
* graph.dfs\_recursive('A')
* print("\n")
* output
* BFS Traversal:
* A B C D E F
* DFS Traversal (Iterative):
* A B D E F C
* DFS Traversal (Recursive):
* A B D E F C