ASSIGNMENT-11

TASK-1

PROMPT

Write a Python implementation of a Stack class that includes the push, pop, peek, and is\_empty methods. This code skeleton uses a standard Python list to represent the stack and includes Google-style docstrings and inline comments for clarity.

CODE

#Create a new stack

my\_stack = Stack()

# Test is\_empty() on an empty stack

print(f"Is the stack empty? {my\_stack.is\_empty()}") # Expected: True

# Push some items onto the stack

my\_stack.push(10)

my\_stack.push(20)

my\_stack.push(30)

# Test is\_empty() on a non-empty stack

print(f"Is the stack empty? {my\_stack.is\_empty()}") # Expected: False

# Test size()

print(f"Size of the stack: {my\_stack.size()}") # Expected: 3

# Test peek()

print(f"Top element: {my\_stack.peek()}") # Expected: 30

# Test pop()

print(f"Popped element: {my\_stack.pop()}") # Expected: 30

print(f"Size after pop: {my\_stack.size()}") # Expected: 2

print(f"Top element after pop: {my\_stack.peek()}") # Expected: 20

# Pop remaining elements

print(f"Popped element: {my\_stack.pop()}") # Expected: 20

print(f"Popped element: {my\_stack.pop()}") # Expected: 10

# Test is\_empty() after popping all elements

print(f"Is the stack empty? {my\_stack.is\_empty()}") # Expected: True

# Test popping from an empty stack (should raise an error)

try:

    my\_stack.pop()

except IndexError as e:

    print(f"Caught expected error: {e}")

# Test peeking from an empty stack (should raise an error)

try:

    my\_stack.peek()

except IndexError as e:

    print(f"Caught expected error: {e}")

OUTPUT

Is the stack empty? True

Is the stack empty? False

Size of the stack: 3

Top element: 30

Popped element: 30

Size after pop: 2

Top element after pop: 20

Popped element: 20

Popped element: 10

Is the stack empty? True

Caught expected error: pop from empty stack

Caught expected error: peek from empty stack

EXPLANATION

It operates on a **Last-In, First-Out (LIFO)** principle, much like a stack of plates—the last plate you put on top is the first one you take off. This particular implementation uses a standard Python list as its underlying storage mechanism.

The code defines a Stack class with four core methods:

* \_\_init\_\_: This is the constructor. It initializes an empty list named \_items, which will hold the elements of the stack. The leading underscore indicates that it's intended for internal use within the class.
* is\_empty(): This method checks if the stack is empty by verifying if the \_items list has any elements. It returns True if it's empty and False otherwise.
* push(): This method adds a new item to the top of the stack. It uses the append() method of the Python list, which efficiently adds an element to the end of the list.
* pop(): This method removes and returns the item from the top of the stack. It first checks if the stack is empty to avoid an error. If not empty, it uses the pop() method of the Python list to remove and return the last element.
* peek(): This method allows you to look at the top item of the stack without removing it. Similar to pop(), it first checks for an empty stack. It then accesses the last element of the list using the index [-1], which is a common Python idiom for accessing the last item.

The rest of the code demonstrates how to use the Stack class by creating an instance, performing various operations like pushing and popping elements, and showing how the IndexError is handled when trying to pop or peek from an empty stack

TASK-2

PROMPT

Write a python program implementation of a **Queue** class in Python using a standard list. The code includes enqueue(), dequeue(), and is\_empty() methods, along with docstrings and a demonstration of its usage.

CODE

# Create an instance of the Queue class

my\_queue = Queue()

# Test is\_empty() on an empty queue

print(f"Is the queue empty? {my\_queue.is\_empty()}") # Expected: True

# Enqueue some items onto the queue

my\_queue.enqueue(10)

my\_queue.enqueue(20)

my\_queue.enqueue(30)

# Test is\_empty() on a non-empty queue

print(f"Is the queue empty? {my\_queue.is\_empty()}") # Expected: False

# Test size()

print(f"Size of the queue: {my\_queue.size()}") # Expected: 3

# Dequeue items from the queue

print(f"Dequeued element: {my\_queue.dequeue()}") # Expected: 10

print(f"Size after dequeue: {my\_queue.size()}") # Expected: 2

print(f"Dequeued element: {my\_queue.dequeue()}") # Expected: 20

print(f"Size after dequeue: {my\_queue.size()}") # Expected: 1

print(f"Dequeued element: {my\_queue.dequeue()}") # Expected: 30

print(f"Size after dequeue: {my\_queue.size()}") # Expected: 0

# Test is\_empty() after dequeuing all elements

print(f"Is the queue empty? {my\_queue.is\_empty()}") # Expected: True

# Test dequeuing from an empty queue (should raise an error)

try:

    my\_queue.dequeue()

except IndexError as e:

    print(f"Caught expected error: {e}")

OUTPUT

Is the queue empty? True

Is the queue empty? False

Size of the queue: 3

Dequeued element: 10

Size after dequeue: 2

Dequeued element: 20

Size after dequeue: 1

Dequeued element: 30

Size after dequeue: 0

Is the queue empty? True

Caught expected error: dequeue from empty queue

EXPLANATION

The code defines a Queue class with the following methods:

* **\_\_init\_\_**: This is the constructor that initializes the queue. It creates an empty list, \_items, to hold the queue's elements. The underscore in \_items is a convention indicating that this is an internal variable of the class.
* **is\_empty()**: This method checks if the queue is empty by seeing if the \_items list is empty. It returns True if there are no items and False if there are.
* **enqueue()**: This method adds an item to the **end** of the queue. It uses the list's append() method, which is very efficient, with a time complexity of O(1).

**dequeue()**: This method removes and returns the item from the **front** of the queue. It first checks if the queue is empty and raises an IndexError if it is. The core of this method is self.\_items.pop(0). This operation is **inefficient** for a list because when an element is removed from the beginning, all other elements must be shifted to the left to fill the gap. This makes it an O(n) operation, where 'n' is the number of items in the queue.

 **peek()**: This method allows you to look at the first item in the queue without removing it. Like dequeue(), it first checks for an empty queue and then accesses the element at index 0.

TASK-3

PROMPT

Write a python program to complete implementation that includes the insert\_at\_end(), delete\_value(), and traverse() methods, with inline comments to explain the key logic for pointer manipulation.

CODE

 #Create a new LinkedList

my\_list = LinkedList()

# Insert elements at the end

my\_list.insert\_at\_end(10)

my\_list.insert\_at\_end(20)

my\_list.insert\_at\_end(30)

my\_list.insert\_at\_end(40)

# Traverse the list

print("List after insertions:")

my\_list.traverse()

# Delete a value

print("\nDeleting value 20:")

my\_list.delete\_value(20)

# Traverse the list again

print("List after deleting 20:")

my\_list.traverse()

# Delete a non-existent value

print("\nDeleting value 50:")

my\_list.delete\_value(50)

# Delete the head node

print("\nDeleting value 10:")

my\_list.delete\_value(10)

# Traverse the list after deleting the head

print("List after deleting 10:")

my\_list.traverse()

# Delete the last node

print("\nDeleting value 40:")

my\_list.delete\_value(40)

# Traverse the list after deleting the last node

print("List after deleting 40:")

my\_list.traverse()

# Delete the remaining node

print("\nDeleting value 30:")

my\_list.delete\_value(30)

# Traverse the list after deleting the last remaining node

print("List after deleting 30:")

my\_list.traverse()

# Try to delete from an empty list

print("\nDeleting from an empty list:")

my\_list.delete\_value(100)

OUTPUT

List after insertions:

10 -> 20 -> 30 -> 40 -> None

Deleting value 20:

List after deleting 20:

10 -> 30 -> 40 -> None

Deleting value 50:

Value 50 not found in the list.

Deleting value 10:

List after deleting 10:

30 -> 40 -> None

Deleting value 40:

List after deleting 40:

30 -> None

Deleting value 30:

List after deleting 30:

List is empty.

Deleting from an empty list:

List is empty, cannot delete.

EXPLANATION

The implementation consists of two classes:

* **Node Class**: This class is the building block of the linked list. Each Node object holds a piece of data and a next attribute, which initially points to None. This next pointer is what links one node to the next.
* **LinkedList Class**: This class manages the entire list. It contains a single attribute, head, which is initially set to None to signify an empty list. The class provides three main methods to manipulate the list:
  + **insert\_at\_end(data)**: This method adds a new node to the end of the list. It creates a new Node with the provided data. If the list is empty (self.head is None), it makes the new node the head. Otherwise, it **traverses** the list from the head until it finds the very last node (the one whose next pointer is None). It then updates this last node's next pointer to point to the newly created node, effectively linking it to the end of the list.
  + **delete\_value(key)**: This method removes the first node that contains the specified key. It handles two main cases:
    1. **Deleting the head**: If the head node itself contains the key, it simply updates the list's head pointer to point to the next node, causing the original head to be "orphaned" and eventually cleaned up by Python's garbage collector.
    2. **Deleting a non-head node**: It traverses the list using two pointers, a current and a previous. When current finds the node to be deleted, it modifies the next pointer of the previous node to skip over current, linking previous directly to current's successor. This effectively removes the node from the list's chain.
  + **traverse()**: This method iterates through the list, starting from the head, and prints the data of each node. It follows the next pointers from one node to the next until it reaches the end of the list (current becomes None).

TASK-4

PROMPT

Write a python program to a complete implementation of a **Binary Search Tree (BST)** in Python, including the Node and BST classes with the specified methods. This code is ready for you to test and use.

CODE

class Node:

"""A node in a BST, containing a value and references to left and right children."""

def \_\_init\_\_(self, value):

self.value = value

self.left = None

self.right = None

class BinarySearchTree:

"""A BST with methods for insertion, searching, and traversal."""

def \_\_init\_\_(self):

self.root = None

def insert(self, value):

if self.root is None:

self.root = Node(value)

else:

self.\_insert\_recursive(self.root, value)

def \_insert\_recursive(self, current\_node, value):

if value < current\_node.value:

if current\_node.left is None:

current\_node.left = Node(value)

else:

self.\_insert\_recursive(current\_node.left, value)

else:

if current\_node.right is None:

current\_node.right = Node(value)

else:

self.\_insert\_recursive(current\_node.right, value)

def search(self, value):

return self.\_search\_recursive(self.root, value)

def \_search\_recursive(self, current\_node, value):

if current\_node is None or current\_node.value == value:

return current\_node

if value < current\_node.value:

return self.\_search\_recursive(current\_node.left, value)

return self.\_search\_recursive(current\_node.right, value)

def inorder\_traversal(self):

result = []

self.\_inorder\_recursive(self.root, result)

return result

def \_inorder\_recursive(self, current\_node, result):

if current\_node:

self.\_inorder\_recursive(current\_node.left, result)

result.append(current\_node.value)

self.\_inorder\_recursive(current\_node.right, result)

# --- Example Usage ---

bst = BinarySearchTree()

elements = [50, 30, 70, 20, 40, 60, 80]

for element in elements:

bst.insert(element)

print("In-order traversal of the tree:", bst.inorder\_traversal())

# Expected output: [20, 30, 40, 50, 60, 70, 80]

print("\nSearching for 40:", bst.search(40) is not None)

# Expected output: True

print("Searching for 99:", bst.search(99) is not None)

OUTPUT

In-order traversal of the tree: [20, 30, 40, 50, 60, 70, 80]

Searching for 40: True

Searching for 99: False

EXPLANATION

The code is built with two main parts:

**1. The Node Class**: This is the basic building block of the tree. Think of each node as a single person in the family tree. Each one holds three pieces of information:

* value: The data itself (like a person's name).
* left: A link to the "left child" node. This link is for values that are **smaller** than the current node's value.
* right: A link to the "right child" node. This link is for values that are **larger** than the current node's value.

**2. The BinarySearchTree Class**: This class manages the entire tree. It holds a reference to the very top node, called the **root**. It provides the main operations to work with the tree:

* insert(value): This method adds a new value to the tree. It starts at the root and moves down the tree, deciding to go left or right at each step until it finds the correct empty spot for the new value, following the "smaller-goes-left, larger-goes-right" rule.
* search(value): This method looks for a specific value in the tree. It follows the same path as the insert method, moving left or right at each node until it finds the value or determines it's not in the tree.
* inorder\_traversal(): This method visits every node in a specific order: it goes all the way left, then visits the current node, and then goes all the way right. The result of this process is a list of all the values in the tree, sorted from smallest to largest. This is a key feature that makes BSTs so useful.

The rest of the code is a simple demonstration that shows how to use these methods by creating a tree, adding some numbers, and then checking if the search function works as expected for both numbers that are in the tree and numbers that are not.

TASK-5

PROMPT

Write a Python implementation of a **Graph** data structure using an **adjacency list**, which is a dictionary where each key represents a vertex and its value is a list of its neighbors. This implementation includes both **Breadth-First Search (BFS)** and **Depth-First Search (DFS)** traversal algorithms.

CODE

"""

Graph representation and traversal algorithms.

"""

from collections import deque

class Graph:

"""

A simple graph class represented by an adjacency list.

"""

def \_\_init\_\_(self):

"""Initializes an empty graph with an adjacency list."""

self.adjacency\_list = {}

def add\_edge(self, u, v):

"""

Adds an edge to the graph. For an undirected graph,

this adds an edge from u to v and from v to u.

"""

if u not in self.adjacency\_list:

self.adjacency\_list[u] = []

if v not in self.adjacency\_list:

self.adjacency\_list[v] = []

self.adjacency\_list[u].append(v)

self.adjacency\_list[v].append(u)

def bfs(self, start\_node):

"""

Performs a Breadth-First Search (BFS) traversal of the graph.

"""

if start\_node not in self.adjacency\_list:

return []

visited = set()

queue = deque([start\_node])

visited.add(start\_node)

traversal\_order = []

while queue:

current\_node = queue.popleft()

traversal\_order.append(current\_node)

for neighbor in self.adjacency\_list.get(current\_node, []):

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

return traversal\_order

def dfs(self, start\_node):

"""

Performs a Depth-First Search (DFS) traversal of the graph.

This uses a stack for iterative traversal.

"""

if start\_node not in self.adjacency\_list:

return []

visited = set()

stack = [start\_node]

traversal\_order = []

while stack:

current\_node = stack.pop()

if current\_node not in visited:

traversal\_order.append(current\_node)

visited.add(current\_node)

for neighbor in reversed(self.adjacency\_list.get(current\_node, [])):

if neighbor not in visited:

stack.append(neighbor)

return traversal\_order

# --- Example Usage ---

if \_\_name\_\_ == "\_\_main\_\_":

g = Graph()

g.add\_edge('A', 'B')

g.add\_edge('A', 'C')

g.add\_edge('B', 'D')

g.add\_edge('B', 'E')

g.add\_edge('C', 'F')

g.add\_edge('C', 'G')

print("--- Graph Adjacency List ---")

for node, neighbors in g.adjacency\_list.items():

print(f"{node}: {neighbors}")

print("\n--- Performing BFS Traversal ---")

bfs\_result = g.bfs('A')

print(f"BFS traversal order: {bfs\_result}")

print("\n--- Performing DFS Traversal ---")

dfs\_result = g.dfs('A')

print(f"DFS traversal order: {dfs\_result}")

OUTPUT

--- Graph Adjacency List --- A: ['B', 'C'] B: ['A', 'D', 'E'] C: ['A', 'F', 'G'] D: ['B'] E: ['B'] F: ['C'] G: ['C'] --- Performing BFS Traversal --- BFS traversal order: ['A', 'B', 'C', 'D', 'E', 'F', 'G'] --- Performing DFS Traversal --- DFS traversal order: ['A', 'C', 'G', 'F', 'B', 'E', 'D']

EXPLANATION

The two main algorithms are:

* **Breadth-First Search (BFS)**: This algorithm explores the graph layer by layer. It starts at a designated node, visits all of its immediate neighbors, then visits all of their neighbors, and so on. To keep track of which nodes to visit next, it uses a **queue** (a "first-in, first-out" structure). This ensures that nodes closer to the starting point are visited first.
* **Depth-First Search (DFS)**: This algorithm explores as far as possible down each branch of the graph before it "backtracks" to try a different path. This implementation uses a **stack** (a "last-in, first-out" structure) to manage the order of nodes to visit.