

AI ASSISTED CODING

ASSIGNMENT 11.4

TASK 1

```
# Create an instance of the Stack class
stack = Stack()

# Test is_empty() on an empty stack
print(f"Is the stack empty? {stack.is_empty()}")

# Push some elements onto the stack
stack.push(10)
stack.push(20)
stack.push(30)

# Test size() after pushing elements
print(f"Stack size after pushing elements: {stack.size()}")

# Test peek()
print(f"Top element of the stack: {stack.peek()}")
```

```
# Test pop()
print(f"Popped element: {stack.pop()}")

# Test size() after popping
print(f"Stack size after popping: {stack.size()}")

# Test is_empty() after popping
print(f"Is the stack empty? {stack.is_empty()}")

# Pop the remaining elements
print(f"Popped element: {stack.pop()}")
print(f"Popped element: {stack.pop()}")

# Test is_empty() again
print(f"Is the stack empty? {stack.is_empty()}")

# Attempt to pop from an empty stack to demonstrate error handling
try:
    stack.pop()
except IndexError as e:
    print(f"Caught expected error when popping from empty stack: {e}")

# Attempt to peek from an empty stack to demonstrate error handling
try:
    stack.peek()
except IndexError as e:
    print(f"Caught expected error when peeking from empty stack: {e}")
```

OUTPUT:

```
Is the stack empty? True
Stack size after pushing elements: 3
Top element of the stack: 30
Popped element: 30
Stack size after popping: 2
Is the stack empty? False
Popped element: 20
Popped element: 10
Is the stack empty? True
Caught expected error when popping from empty stack: pop from empty stack
Caught expected error when peeking from empty stack: peek from empty stack
```

EXPLANATION :

`stack = Stack():` This line creates an instance of the `Stack` class you defined earlier.
This stack object will be used to perform stack operations.

`print(f"Is the stack empty? {stack.is_empty()}"")`: This calls the `is_empty()` method on the stack object. Since the stack is newly created and has no elements, `is_empty()` will return `True`, and the output will reflect that.

```
stack.push(10)
stack.push(20)
```

`stack.push(30):` These lines use the `push()` method to add elements (10, 20, and 30) to the top of the stack. The stack now contains these elements in the order they were pushed, with 30 being the top element.

`print(f"Stack size after pushing elements: {stack.size()}"")`: This calls the `size()` method, which returns the number of elements currently in the stack. After pushing three elements, the size will be 3.

`print(f"Top element of the stack: {stack.peek()}"")`: This calls the `peek()` method, which returns the top element of the stack without removing it. In this case, it will return 30.

`print(f"Popped element: {stack.pop()}"")`: This calls the `pop()` method, which removes and returns the top element of the stack. This will remove 30 and print it.

`print(f"Stack size after popping: {stack.size()}"")`: After popping one element, the size of the stack is now 2.

`print(f"Is the stack empty? {stack.is_empty()}"")`: The stack is not empty (it still contains 10 and 20), so this will print `False`.

`print(f"Popped element: {stack.pop()}"")`

`print(f"Popped element: {stack.pop()}"")`: These lines pop the remaining elements (20 and then 10) from the stack.

```
print(f"Is the stack empty? {stack.is_empty()}"): The stack is now empty, so this will  
print True.
```

try...except IndexError blocks: These blocks demonstrate how the pop() and peek() methods handle the case where the stack is empty. When you try to pop or peek from an empty stack, the IndexError that is raised by the methods is caught, and a message indicating the expected error is printed. This shows the error handling implemented in your Stack class.

In summary, this code provides a comprehensive test of all the core operations of your Stack class, including pushing and popping elements, checking the size and emptiness, peeking at the top element, and handling errors when attempting to operate on an empty stack.

Task 2

```
class Queue:
```

```
    """A simple Queue implementation using a Python list."""
```

```
    def __init__(self):
```

```
        """Initializes an empty Queue."""
```

```
        self._items = []
```

```
    def enqueue(self, item):
```

```
        """
```

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

Args:

item: The item to be added to the queue.

```
        """
```

```
        self._items.append(item)
```

```
def dequeue(self):
```

```
    """
```

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

Returns:

The item from the front of the queue.

Raises:

IndexError: If the queue is empty.

```
    """
```

```
if not self.is_empty():
```

```
    # Removing from the front of a list can be inefficient
```

```
    return self._items.pop(0)
```

```
else:
```

```
    raise IndexError("dequeue from empty queue")
```

```
def is_empty(self):
```

```
    """
```

Checks if the queue is empty.

Returns:

True if the queue is empty, False otherwise.

```
    """
```

```
return len(self._items) == 0
```

```
# Test the Queue implementation
```

```
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"Dequeue element: {queue.dequeue()}")  
print(f"Dequeue element: {queue.dequeue()}")  
print(f"Dequeue element: {queue.dequeue()}")  
  
print(f"Is queue empty? {queue.is_empty()}")
```

try:

```
    queue.dequeue()  
except IndexError as e:  
    print(f"Caught expected error: {e}")
```

output:

```
Is queue empty? True  
Is queue empty? False  
Dequeue element: 10  
Dequeue element: 20  
Dequeue element: 30  
Is queue empty? True  
Caught expected error: dequeue from empty queue
```

Explanation :

implements a basic Queue data structure in Python using a list. Here's a breakdown:

- *class Queue:: This defines a new class named Queue.*
- *_init_(self):: This is the constructor of the class. It initializes an empty list _items which will store the elements of the queue.*
- *enqueue(self, item):: This method adds an item to the rear of the queue. It uses the append() method of the list, adding the item to the end. In this list-based implementation, the end of the list represents the rear of the queue.*
- *dequeue(self):: This method removes and returns the item from the front of the queue. It first checks if the queue is empty using is_empty(). If not empty, it uses self._items.pop(0) to remove and return the element at index 0, which is the front of the list and thus the front of the queue. Removing from the beginning of a list can be slow for large lists because all subsequent elements need to be shifted. If the queue is empty, it raises an IndexError.*
- *is_empty(self):: This method checks if the queue is empty by checking the length of the _items list. It returns True if the length is 0, and False otherwise.*

The code also includes test cases to show how to use the Queue class and its methods, including handling the IndexError when trying to dequeue from an empty queue.

Task 3

```
# Test Case 1: Insert at the end and traverse
print("Test Case 1:")
my_list = LinkedList()
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
my_list.traverse()
```

```
# Test Case 2: Delete a value
print("\nTest Case 2:")
my_list = LinkedList()
```

```
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)
my_list.insert_at_end(20)
my_list.insert_at_end(40)
print("Original list:")
my_list.traverse()
my_list.delete_value(20)
print("After deleting first 20:")
my_list.traverse()
my_list.delete_value(20)
print("After deleting second 20:")
my_list.traverse()
my_list.delete_value(50)
print("After deleting 50 (not in list):")
my_list.traverse()
my_list.delete_value(10)
print("After deleting 10 (head):")
my_list.traverse()
my_list.delete_value(40)
print("After deleting 40 (last):")
my_list.traverse()
my_list.delete_value(30)
print("After deleting 30 (only node):")
my_list.traverse()

# Test Case 3: Delete from an empty list
print("\nTest Case 3:")
```

```
my_list = LinkedList()
print("Empty list:")
my_list.traverse()
my_list.delete_value(10)
print("After attempting to delete from empty list:")
my_list.traverse()
```

Output :

```
 ➔ Test Case 1:
10 -> 20 -> 30 -> None

Test Case 2:
Original list:
10 -> 20 -> 30 -> 20 -> 40 -> None
After deleting first 20:
10 -> 30 -> 20 -> 40 -> None
After deleting second 20:
10 -> 30 -> 40 -> None
After deleting 50 (not in list):
10 -> 30 -> 40 -> None
After deleting 10 (head):
30 -> 40 -> None
After deleting 40 (last):
30 -> None
After deleting 30 (only node):
None

Test Case 3:
Empty list:
None
After attempting to delete from empty list:
None
```

Explanation :

1. *Node Class:*

- o *This class represents a single element (or node) within the linked list.*

- o `_init_(self, data)`: The constructor takes data as input, which is the value the node will hold. It initializes the node's data attribute and sets the next attribute to None. The next attribute is a pointer that will link this node to the next node in the sequence.

2. *LinkedList Class:*

- o `This class represents the entire linked list.`
- o `_init_(self)`: The constructor initializes the head attribute to None. The head is a pointer to the first node in the list. An empty list has no head, hence None.
- o `insert_at_end(self, data)`: This method adds a new node with the given data to the end of the list.

It first creates a new_node.

If the list is empty (self.head is None), the new_node becomes the head.

If the list is not empty, it traverses the list starting from the head until it finds the last node (the one whose next is None).

It then updates the next pointer of the last node to point to the new_node.

- o `delete_value(self, value)`: This method deletes the first node it finds with the given value.

It handles the case where the list is empty.

It checks if the head node contains the value and updates the head if it does.

If the value is not in the head, it traverses the list with current_node until it finds the node before the one to be deleted.

If the node to be deleted is found, it updates the next pointer of the current_node to skip the node with the value, effectively removing it from the list.

- o `traverse(self)`: This method iterates through the linked list from the head to the end, printing the data of each node. It prints "None" at the end to indicate the end of the list.

Task 4

```
# Test the BST implementation
```

```
bst = BST()
```

```
elements_to_insert = [50, 30, 70, 20, 40, 60, 80]

# Insert elements
print("Inserting elements:", elements_to_insert)
for element in elements_to_insert:
    bst.insert(element)

# Perform in-order traversal
print("\nIn-order traversal:")
inorder_result = bst.inorder_traversal()
print(inorder_result)

# Test search for present elements
elements_to_search_present = [20, 50, 80]
print("\nSearching for present elements:")
for element in elements_to_search_present:
    found_node = bst.search(element)
    if found_node:
        print(f"Element {element} found in BST.")
    else:
        print(f"Element {element} not found in BST.") # Should not happen for these
elements

# Test search for absent elements
elements_to_search_absent = [25, 55, 90]
print("\nSearching for absent elements:")
for element in elements_to_search_absent:
    found_node = bst.search(element)
```

```
if found_node:  
    print(f"Element {element} found in BST.") # Should not happen for these elements  
else:  
    print(f"Element {element} not found in BST.")
```

out put :

Inserting elements: [50, 30, 70, 20, 40, 60, 80]

In-order traversal:

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

Searching for present elements:

Element 20 found in BST.

Element 50 found in BST.

Element 80 found in BST.

Explanation :

Data Analysis Key Findings

The Node class was successfully defined with value, left, and right attributes.

The BST class was successfully defined with a root attribute, initialized to None.

The insert method was implemented to add nodes while maintaining BST properties using a recursive helper function.

The search method was implemented to find values in the BST using a recursive helper function.

The inorder_traversal method was implemented using a recursive helper function, producing a sorted list of elements.

Docstrings were added to all classes and methods.

Testing the search method confirmed it correctly identifies both present (40) and absent (99) values, returning True and False respectively.

The inorder traversal of the constructed BST [50, 30, 70, 20, 40, 60, 80] resulted in the sorted list [20, 30, 40, 50, 60, 70, 80].

Insights or Next Steps

The successful implementation and testing of the core BST operations (insert, search, inorder_traversal) demonstrate a solid foundation for more complex tree algorithms.

Consider adding methods for other common BST operations such as deletion, finding minimum/maximum, or different traversal types (preorder, postorder).

Task 5

```
# Create a sample Graph object
graph = Graph()

# Add edges to represent a sample graph structure
graph.add_edge('A', 'B')
graph.add_edge('A', 'C')
graph.add_edge('B', 'D')
graph.add_edge('B', 'E') # Corrected the syntax error here
graph.add_edge('D', 'E')
graph.add_edge('E', 'F')
graph.add_edge('C', 'G')

# Choose a starting node for traversal
start_node = 'A'

# Call the bfs method
graph.bfs(start_node)

# Call the dfs_recursive method
print(f"DFS (Recursive) starting from node {start_node}:")
graph.dfs_recursive(start_node)
print() # Print a newline after recursive DFS
```

```
# Call the dfs_iterative method  
graph.dfs_iterative(start_node)
```

output:

```
BFS starting from node A:  
A B C D E G F  
DFS (Recursive) starting from node A:  
A B D E F C G  
DFS (Iterative) starting from node A:  
A B D E F C G
```

Explanation :

The key findings mentioned are:

The successful creation of the Graph class with an adjacency list.

The implementation of add_edge, bfs, dfs_recursive, and dfs_iterative methods.

The addition of inline comments to explain the traversal steps.

The successful testing of BFS and both recursive and iterative DFS on a sample graph.

The observation that for the given sample graph and starting node, the recursive and iterative DFS produced the same output order, with an explanation of why this might happen and how the iterative implementation was designed to potentially mimic the recursive one.

A discussion of the conceptual differences, advantages, and disadvantages of recursive vs. iterative DFS, noting the implicit use of the call stack in recursive DFS and the explicit stack in iterative DFS, as well as the potential for stack overflow in recursive DFS for deep graphs.

The suggested next steps include:

Recognizing that the current implementation provides a good base for more advanced graph algorithms.

Considering the addition of methods for different types of graphs (directed, weighted) or other traversal/search algorithms (Dijkstra's, A).*

Essentially, this summary confirms that the task of implementing a graph with BFS and DFS was completed successfully and offers ideas for how to extend this work.