ASSIGNMENT 11

RAMIDI . SAI CHARAN

2403A52124

TASK 1 :

```
stack = Stack()
print(f"Is the stack empty? {stack.is_empty()}")

# Test push()
stack.push(10)
stack.push(20)
stack.push(30)
print(f"After pushing items, stack size: {stack.size()}")

# Test peek()
print(f"Top item (peek): {stack.peek()}")

# Test pop()
print(f"Popped item: {stack.pop()}")
print(f"After popping, stack size: {stack.size()}")

# Test pop() again
print(f"Popped item: {stack.pop()}")
print(f"After popping again, stack size: {stack.size()}")

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

# Pop the last item
print(f"Popped item: {stack.pop()}")
print(f"After popping the last item, stack size: {stack.size()}")
# Test is_empty() on an empty stack again
print(f"Is the stack empty? {stack.is_empty()}")
# Test pop() on an empty stack (should raise an error)
try:
    stack.pop()
except IndexError as e:
    print(f"Caught expected error when popping from empty stack: {e}")

# Test peek() on an empty stack (should raise an error)
try:
    stack.peek()
except IndexError as e:
    print(f"Caught expected error when peeking from empty stack: {e}")
```

ASSIGNMENT 11

```
Is the stack empty? True
After pushing items, stack size: 3
Top item (peek): 30
Popped item: 30
After popping, stack size: 2
Popped item: 20
After popping again, stack size: 1
Is the stack empty? False
Popped item: 10
After popping the last item, stack size: 0
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 :

Certainly! Let's go into more detail about the test code and what each part is doing:

1. `stack = Stack()`: This line creates an instance of our `Stack` class, essentially creating an empty stack data structure in memory.
2. `print(f"Is the stack empty? {stack.is_empty()}")`: Here, we call the `is_empty()` method on our newly created stack. Since it's empty at this point, the `is_empty()` method should return `True`, and the output will confirm this.
3. `stack.push(10)`
   `stack.push(20)`
   `stack.push(30)`: These lines demonstrate the `push()` operation. Each call adds an element to the top of the stack. So, 10 is added first, then 20 is placed on top of 10, and finally 30 is placed on top of 20. The stack now contains [10, 20, 30] from bottom to top.
4. `print(f"After pushing items, stack size: {stack.size()}")`: We call the `size()` method to get the current number of items in the stack. After pushing three items, the size should be 3, and this line prints that value.
5. `print(f"Top item (peek): {stack.peek()}")`: The `peek()` method allows us to look at the item at the very top of the stack without removing it. In this case, 30 was the last item pushed, so `peek()` will return 30.
6. `print(f"Popped item: {stack.pop()}")`: This line performs the `pop()` operation, which removes the item from the top of the stack and returns it. Since 30 was on top, it is removed and printed. The stack now contains [10, 20].
7. `print(f"After popping, stack size: {stack.size()}")`: We check the size again after the first `pop()`. The size should now be 2.
8. `print(f"Popped item: {stack.pop()}")`: Another `pop()` operation is performed. This time, 20 is at the top, so it's removed and printed. The stack now contains [10].
9. `print(f"After popping again, stack size: {stack.size()}")`: The size is checked again and should now be 1.
10. `print(f"Is the stack empty? {stack.is_empty()}")`: We call `is_empty()` again. Since the stack still contains 10, it is not empty, and this should print `False`.
11. `print(f"Popped item: {stack.pop()}")`: The final item, 10, is popped from the stack and printed. The stack is now empty.
12. `print(f"After popping the last item, stack size: {stack.size()}")`: The size is checked one last time and should be 0.
13. `print(f"Is the stack empty? {stack.is_empty()}")`: With the stack now empty, `is_empty()` should return `True`, and this line confirms that.
14. `try...except IndexError as e:`: These blocks are included to demonstrate how the `pop()` and `peek()` methods handle the error condition when the stack is empty. Attempting to `pop()` or `peek()` from an empty stack is not allowed and should raise an `IndexError`. The `try...except` block catches this expected error and prints a message confirming that the error was caught.

This detailed test code covers all the basic operations of the Stack class and also shows how it behaves in edge cases like being empty.

ASSIGNMENT 11

TASK 2 :

```python
class Queue:
    def __init__(self):
        """Initializes an empty queue."""
        self.items = []

    def is_empty(self):
        """Checks if the queue is empty."""
        return len(self.items) == 0

    def enqueue(self, item):
        """Adds an item to the back of the queue."""
        self.items.append(item)

    def dequeue(self):
        """Removes and returns the item from the front of the queue."""
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        return self.items.pop(0)

# Example usage:
my_queue = Queue()

print(f"Is the queue empty? {my_queue.is_empty()}")

my_queue.enqueue('A')
my_queue.enqueue('B')
my_queue.enqueue('C')

print(f"Enqueued items. Queue is now empty? {my_queue.is_empty()}")

print(f"Dequeued item: {my_queue.dequeue()}")
print(f"Dequeued item: {my_queue.dequeue()}")

print(f"Is the queue empty after dequeuing? {my_queue.is_empty()}")

my_queue.enqueue('D')
print(f"Enqueued 'D'. Current queue: {my_queue.items}")

print(f"Dequeued item: {my_queue.dequeue()}")
print(f"Dequeued item: {my_queue.dequeue()}")
print(f"Is the queue empty now? {my_queue.is_empty()}")

try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Caught expected error when dequeuing from empty queue: {e}")
```

ASSIGNMENT 11

```
Is the queue empty? True
Enqueued items. Queue is now empty? False
Dequeued item: A
Dequeued item: B
Is the queue empty after dequeuing? False
Enqueued 'D'. Current queue: ['C', 'D']
Dequeued item: C
Dequeued item: D
Is the queue empty now? True
Caught expected error when dequeuing from empty queue: dequeue from an empty queue
```

EXPLANATION :

This code defines a `Queue` class and then provides an example of how to use it.

Here's a breakdown:

Queue Class Definition:

- `class Queue:` : This line starts the definition of a new class named `Queue`.
- `def __init__(self):` : This is the constructor of the class. It's called when you create a new `Queue` object. It initializes an empty list called `self.items` which will be used to store the elements of the queue.
- `def is_empty(self):` : This method checks if the queue is empty by checking if the `self.items` list has a length of 0. It returns `True` if empty, `False` otherwise.
- `def enqueue(self, item):` : This method adds an `item` to the back of the queue. It uses the `append()` method of the list to add the item to the end of `self.items`.
- `def dequeue(self):` : This method removes and returns the item from the front of the queue.
    - It first checks if the queue is empty using `self.is_empty()`. If it is, it raises an `IndexError` because you cannot dequeue from an empty queue.
    - If the queue is not empty, it uses `self.items.pop(0)` to remove and return the element at index 0 (the front of the list).

Example Usage:

- `my_queue = Queue()` : This creates an instance of the `Queue` class.
- `print(f"Is the queue empty? {my_queue.is_empty()}")` : This checks and prints if the newly created queue is empty (it should be).
- `my_queue.enqueue('A')`
  `my_queue.enqueue('B')`
  `my_queue.enqueue('C')` : These lines add the items 'A', 'B', and 'C' to the queue using the `enqueue()` method. 'A' is at the front, 'B' is next, and 'C' is at the back.
- `print(f"Enqueued items. Queue is now empty? {my_queue.is_empty()}")` : Checks and prints that the queue is no longer empty.
- `print(f"Dequeued item: {my_queue.dequeue()}")` : This removes and prints the item at the front of the queue, which is 'A'.
- `print(f"Dequeued item: {my_queue.dequeue()}")` : This removes and prints the next item at the front, which is 'B'.
- `print(f"Is the queue empty after dequeuing? {my_queue.is_empty()}")` : Checks and prints if the queue is empty after removing 'A' and 'B' (it should not be, as 'C' is still there).
- `my_queue.enqueue('D')` : Adds 'D' to the back of the queue. The queue is now ['C', 'D'].
- `print(f"Enqueued 'D'. Current queue: {my_queue.items}")` : Prints the internal list representation of the queue to show its current state.
- `print(f"Dequeued item: {my_queue.dequeue()}")` : Removes and prints 'C'.
- `print(f"Dequeued item: {my_queue.dequeue()}")` : Removes and prints 'D'.
- `print(f"Is the queue empty now? {my_queue.is_empty()}")` : Checks and prints that the queue is now empty.
- `try...except IndexError as e:` : This block demonstrates what happens when you try to `dequeue()` from an empty queue. It catches the expected `IndexError` and prints a message.

ASSIGNMENT 11

TASK 3 :

```python
class SinglyLinkedList:
    class Node:
        def __init__(self, data):
            self.data = data
            self.next = None

    def __init__(self):
        self.head = None

    def insert_at_end(self, data):
        if not self.head:
            self.head = self.Node(data)
            return

        node = self.head
        while node.next:
            node = node.next
        node.next = self.Node(data)
```

```python
    def delete_value(self, value):
        if self.head and self.head.data == value:
            self.head = self.head.next
            return

        node = self.head
        while node and node.next:
            if node.next.data == value:
                node.next = node.next.next
                return
            node = node.next

    def traverse(self):
        node = self.head
        output = []
        while node:
            output.append(str(node.data))
            node = node.next
        print(" -> ".join(output) + " -> None")
```

ASSIGNMENT 11

```python
# Example Usage:
my_list = SinglyLinkedList()
my_list.insert_at_end(10)
my_list.insert_at_end(20)
my_list.insert_at_end(30)

print("List after insertions:")
my_list.traverse()

my_list.delete_value(20)
print("\nList after deleting 20:")
my_list.traverse()

my_list.delete_value(10)
print("\nList after deleting the head (10):")
my_list.traverse()

my_list.delete_value(40)
print("\nList after trying to delete a non-existent value:")
my_list.traverse()
```

```
List after insertions:
10 -> 20 -> 30 -> None

List after deleting 20:
10 -> 30 -> None

List after deleting the head (10):
30 -> None

List after trying to delete a non-existent value:
30 -> None
```

EXPLANATION :

This Python code implements a `SinglyLinkedList`. It includes a nested `Node` class for list elements. Each `Node` stores `data` and links to the `next` node. The `SinglyLinkedList` class manages the `head` of the list. `insert_at_end(data)` adds a new node to the list's tail. `delete_value(value)` removes the first node matching the value. It handles deleting the head or values not found. `traverse()` prints the list's elements sequentially. The example code demonstrates these operations. It shows adding, deleting, and printing the list state.

ASSIGNMENT 11

TASK 4 :

```python
class Node:
    """A single node in a Binary Search Tree."""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BinarySearchTree:
    """A Binary Search Tree data structure."""
    def __init__(self):
        self.root = None

    def insert(self, key):

        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        """A private helper method for recursive insertion."""
        if node is None:
            return Node(key)

        if key < node.key:
            node.left = self._insert_recursive(node.left, key)
        elif key > node.key:
            node.right = self._insert_recursive(node.right, key)

        return node
```

```python
def search(self, key):

    return self._search_recursive(self.root, key)

def _search_recursive(self, node, key):
    """A private helper method for recursive searching."""
    if node is None or node.key == key:
        return node is not None
```

ASSIGNMENT 11

```
        if key < node.key:
            return self._search_recursive(node.left, key)
        else:
            return self._search_recursive(node.right, key)

    def inorder_traversal(self):
        result = []
        self._inorder_traversal_recursive(self.root, result)
        return result

    def _inorder_traversal_recursive(self, node, result):
        """A private helper method for recursive inorder traversal."""
        if node:
            self._inorder_traversal_recursive(node.left, result)
            result.append(node.key)
            self._inorder_traversal_recursive(node.right, result)
# Initialize the BST
bst = BinarySearchTree()
elements_to_insert = [50, 30, 70, 20, 40, 60, 80]

# Insert elements
for element in elements_to_insert:
    bst.insert(element)

# Perform inorder traversal
print("Inorder traversal of the BST:")
print(bst.inorder_traversal())  # Output: [20, 30, 40, 50, 60, 70, 80]
print("-" * 20)

# Test search for present and absent elements
search_key_present = 40
print(f"Is {search_key_present} in the BST? {bst.search(search_key_present)}")

search_key_absent = 99
print(f"Is {search_key_absent} in the BST? {bst.search(search_key_absent)}")
```

```
Inorder traversal of the BST:
[20, 30, 40, 50, 60, 70, 80]
--------------------
Is 40 in the BST? True
Is 99 in the BST? False
```

ASSIGNMENT 11

EXPALNATION :

This code provides the basic structure for a Binary Search Tree (BST).

- The `Node` class represents each element in the tree. Each node stores a `key` (the value of the node) and has pointers (`left` and `right`) to its child nodes. In a BST, the left child's key is always less than the parent's key, and the right child's key is always greater.
- The `BinarySearchTree` class represents the entire tree. It holds a reference to the `root` node, which is the starting point of the tree.
- The `insert()`, `search()`, and `inorder_traversal()` methods are defined as placeholders. These are the core operations for a BST, and they will be implemented in the next step to allow adding elements, finding elements, and visiting elements in sorted order, respectively.

TASK 5 :

```
from collections import deque

class Graph:

    def __init__(self):
        self.adjacency_list = {}

    def add_edge(self, u, v, directed=False):

        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)
        if not directed:
            self.adjacency_list[v].append(u)

    def bfs(self, start_node):

        visited = set()
        queue = deque([start_node])
        visited.add(start_node)

        print("BFS Traversal starting from", start_node, ":")
```

ASSIGNMENT 11

```python
    visited = set()
    queue = deque([start_node])
    visited.add(start_node)

    print("BFS Traversal starting from", start_node, ":")

    while queue:
        current_node = queue.popleft()
        print(current_node, end=" ")

        # Explore neighbors of the current node
        for neighbor in sorted(self.adjacency_list.get(current_node, [])):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    print("\n")

ef dfs(self, start_node):

    visited = set()
    stack = [start_node]

    print("DFS Traversal (Iterative) starting from", start_node, ":")

    while stack:
        current_node = stack.pop()

        if current_node not in visited:
            print(current_node, end=" ")
            visited.add(current_node)

            for neighbor in sorted(self.adjacency_list.get(current_node, []), reverse=True):
                if neighbor not in visited:
                    stack.append(neighbor)
    print("\n")
```

```python
# Create a new graph and add edges
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('E', 'F')

# Print the adjacency list for clarity
print("Adjacency List:")
for node, neighbors in g.adjacency_list.items():
    print(f"{node}: {sorted(neighbors)}")
print("-" * 20)

# Perform BFS traversal
g.bfs('A')

# Perform DFS traversal
g.dfs('A')
```

```
Adjacency List:
A: ['B', 'C']
B: ['A', 'D', 'E']
C: ['A', 'F']
D: ['B']
E: ['B', 'F']
F: ['C', 'E']
--------------------
BFS Traversal starting from A :
A B C D E F

DFS Traversal (Iterative) starting from A :
A B D E F C
```

ASSIGNMENT 11

This code defines a `Graph` class using an adjacency list, a dictionary where keys are nodes and values are lists of neighbors.

The `__init__` method initializes this empty adjacency list.

The `add_edge(u, v, directed=False)` method adds connections between nodes `u` and `v`. If `directed` is False (default), it adds edges `u` to `v` and `v` to `u`. It also ensures nodes `u` and `v` exist in the adjacency list.

The `bfs(start_node)` method performs a Breadth-First Search. It uses a `deque` as a queue and a `set` to track visited nodes. It explores neighbors layer by layer, adding unvisited neighbors to the queue.

The `dfs(start_node)` method performs an iterative Depth-First Search using a list as a stack and a `set` for visited nodes. It explores as deeply as possible along a path before backtracking, adding unvisited neighbors to the stack.

The example usage creates a `Graph` instance. It adds several undirected edges between nodes 'A' through 'F'. It then prints the resulting adjacency list. Finally, it demonstrates both BFS and DFS traversals starting from node 'A'. The output shows the order in which nodes are visited by each algorithm. BFS visits nodes level by level (A, B, C, then D, E, F). DFS explores one branch fully before moving to another (A, B, D, then E, then F, then C). This code provides a clear example of how to represent and traverse a graph.