

AI ASSISTED CODING

LAB ASSIGNMENT – 11.4

HALL TICKET NUM : 2403A52403

BATCH NO : 14

Task – 01:

Test the Stack operations

```
stack = Stack()
```

```
print("Is the stack empty?", stack.is_empty())
```

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

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

```
stack.push(30)
```

```
print("Stack after pushing elements:", stack)
```

```
print("Is the stack empty?", stack.is_empty())
```

```
print("Top element (peek):", stack.peek())
```

```
print("Popped element:", stack.pop())
```

```
print("Stack after popping:", stack)
```

```
print("Popped element:", stack.pop())
```

```
print("Stack after popping again:", stack)
```

```
print("Is the stack empty?", stack.is_empty())
```

Attempt to pop from an empty stack

try:

```
    stack.pop()
```

except IndexError as e:

```
print("Error:", e)

# Attempt to peek from an empty stack

try:
    stack.peak()
except IndexError as e:
    print("Error:", e)
```

Output:

```
➞ Is the stack empty? True
   Stack after pushing elements: [10, 20, 30]
   Is the stack empty? False
   Top element (peek): 30
   Popped element: 30
   Stack after popping: [10, 20]
   Popped element: 20
   Stack after popping again: [10]
   Is the stack empty? False
   Error: peek from empty stack
```

Explanation :

1. **Code Cell 1 (Stack Class**

Definition): This cell defines a Python class named Stack which implements a Last-In, First-Out (LIFO) data structure.

- `__init__(self)`: This is the constructor. It initializes an empty list `_items` which will be used to store the elements of the stack.

- `push(self, item)`: This method adds an item to the top of the stack. It uses the `append()` method of the internal list, which adds the item to the end of the list.
- `pop(self)`: This method removes and returns the item from the top of the stack. It first checks if the stack is empty using `is_empty()`. If not empty, it uses the `pop()` method of the internal list, which removes and returns the last element (the top of the stack). If the stack is empty, it raises an `IndexError`.
- `peek(self)`: This method returns the item at the top of the stack without removing it. Similar to `pop()`, it checks if the stack is empty. If not empty, it returns the last element of the list (`self._items[-1]`). If the stack is empty, it raises an `IndexError`.

- `is_empty(self)`: This method checks if the stack is empty by checking if the length of the internal list `_items` is 0.
- `__str__(self)`: This method provides a user-friendly string representation of the stack, which is simply the string representation of the internal list.
- `__repr__(self)`: This method provides a developer-friendly string representation of the stack, showing the class name and the internal list.

2. **Code Cell 2 (Stack Operations**

Test): This cell demonstrates how to use the Stack class and tests its operations.

- It creates an instance of the Stack class.
- It checks if the newly created stack is empty using `is_empty()`.
- It pushes three elements (10, 20, 30) onto the stack using the `push()` method.

- It prints the stack's state after pushing using the `print()` function, which calls the `__str__()` method.
- It checks if the stack is empty again.
- It peeks at the top element using `peek()`.
- It pops elements from the stack twice using `pop()`, printing the popped element and the stack's state after each pop.
- It checks if the stack is empty after popping.
- Finally, it includes `try...except` blocks to demonstrate how the `IndexError` is raised when attempting to `pop()` or `peek()` from an empty stack.

In summary, the code implements a basic Stack data structure using a Python list and provides methods for common stack operations, along with a test script to show how to use the class and handle potential errors.

Task – 02:

```
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 – 03:

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:**

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

- `__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:**

- This class represents the entire linked list.
- `__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.
- `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.
- 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.
- `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.

The code then provides three test cases (Test Case 1, Test Case 2, and Test Case 3) that demonstrate how to use these methods and verify their correctness.

0 / 2000

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Task – 04:

```
# Test with a list of integers
```

```
bst = BST()
```

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

```
for element in elements_to_insert:
```

```
    bst.insert(element)
```

```
print("In-order traversal:")
```

```
print(bst.inorder_traversal())
```

```
# Test search for present and absent elements
```

```
print("\nTesting search:")
```

```
present_element = 40
```

```
absent_element = 90
```

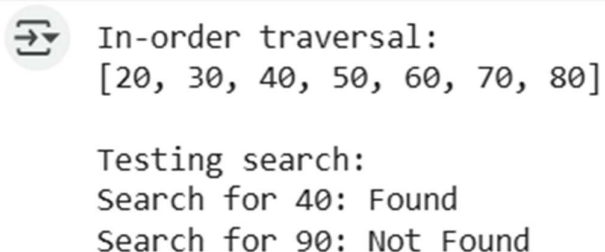
```
search_present = bst.search(present_element)
```

```
search_absent = bst.search(absent_element)
```

```
if search_present:
    print(f"Search for {present_element}: Found")
else:
    print(f"Search for {present_element}: Not Found")
```

```
if search_absent:
    print(f"Search for {absent_element}: Found")
else:
    print(f"Search for {absent_element}: Not Found")
```

Output :

A terminal window with a light gray background and a blue border. It contains the following text: In-order traversal: [20, 30, 40, 50, 60, 70, 80], Testing search: Search for 40: Found, and Search for 90: Not Found.

```
In-order traversal:
[20, 30, 40, 50, 60, 70, 80]

Testing search:
Search for 40: Found
Search for 90: Not Found
```

Explanation :

1. Node Class:

- This class represents a single node within the BST.
- `__init__(self, key)`: The constructor initializes a node with a key (the value stored in the node) and sets left and right attributes to None, which will eventually point to the left and right child nodes, respectively.

2. BST Class:

- This class represents the entire Binary Search Tree.
- `__init__(self)`: The constructor initializes an empty BST by setting the root attribute to None.
- `insert(self, key)`: This method is used to insert a new node with the given key into the BST.
 - If the tree is empty (`self.root` is None), the new node becomes the root.

- Otherwise, it calls the private helper method `_insert_recursive` to find the correct position for the new node.
- `_insert_recursive(self, current_node, key)`: This is a recursive helper method for insertion.
 - It compares the key to the `current_node.key`.
 - If key is less than `current_node.key`, it moves to the left child. If the left child is `None`, it inserts the new node there; otherwise, it recursively calls itself on the left child.
 - If key is greater than `current_node.key`, it moves to the right child, following a similar logic.
 - If key is equal, it does nothing (assuming the BST does not allow duplicate keys).
- `search(self, key)`: This method searches for a node with the given key in the BST.

- It calls the private helper method `_search_recursive`.
- `_search_recursive(self, current_node, key)`:
This is a recursive helper method for searching.
 - It checks if the `current_node` is `None` (key not found) or if the `current_node.key` matches the key. If either is true, it returns the `current_node`.
 - If key is less than `current_node.key`, it recursively calls itself on the left child.
 - Otherwise (if key is greater), it recursively calls itself on the right child.
- `inorder_traversal(self)`: This method performs an in-order traversal of the BST and returns a list of the keys in ascending order.
 - It calls the private helper method `_inorder_recursive`.

- `_inorder_recursive(self, current_node, result)`: This is a recursive helper method for in-order traversal.
 - It checks if the `current_node` exists.
 - If it does, it first recursively traverses the left subtree
`(_inorder_recursive(current_node.left, result))`.
 - Then, it appends the `current_node.key` to the result list.
 - Finally, it recursively traverses the right subtree
`(_inorder_recursive(current_node.right, result))`.

Testing Code (cell with df73c9d2):

- An instance of the BST class is created.
- A list of integers `elements_to_insert` is defined.

- The code iterates through this list and inserts each element into the BST using the `insert()` method.
- It then calls `inorder_traversal()` and prints the resulting sorted list of keys. This demonstrates that the insertion and in-order traversal are working correctly.
- Finally, it tests the `search()` method with an element that is present (`present_element = 40`) and an element that is absent (`absent_element = 90`). It prints whether the element was found or not based on the return value of the `search()` method.

In summary, the code provides a complete implementation of a Binary Search Tree with core functionalities and includes tests to verify its correctness.

Task – 05:

Create a graph instance

```
g = Graph()
```



```
# Add edges to the graph
```

```
g.add_edge(0, 1)
```

```
g.add_edge(0, 2)
```

```
g.add_edge(1, 2)
```

```
g.add_edge(2, 0)
```

```
g.add_edge(2, 3)
```

```
g.add_edge(3, 3)
```

```
# Perform BFS traversal
```

```
print("BFS traversal:", end=" ")
```

```
g.bfs(2)
```

```
print() # Print a newline after BFS output
```

```
# Perform DFS traversal
```

```
print("DFS traversal:", end=" ")
```

```
g.dfs(2)
```

```
print() # Print a newline after DFS output
```

Output:

```
➡ BFS traversal: 2 0 1 3  
DFS traversal: 2 0 1 3
```

Explanation :

This code demonstrates how to use the Graph class, which you've implemented earlier, to create a graph, add edges, and perform both Breadth-First Search (BFS) and Depth-First Search (DFS) traversals.

Here's a breakdown:

1. **g = Graph():** This line creates an instance of your Graph class, effectively initializing an empty graph.
2. **g.add_edge(0, 1), g.add_edge(0, 2),** etc.: These lines call the add_edge method of the g object to add undirected edges between the specified nodes. For example, g.add_edge(0, 1) adds an edge connecting node 0 and node 1.

3. **print("BFS traversal:", end=" ")**: This line simply prints the label "BFS traversal: " to the console. The end=" " prevents a newline character from being printed, so the BFS output will appear on the same line.
4. **g.bfs(2)**: This line calls the bfs method on the graph g, starting the Breadth-First Search traversal from node 2. The output of the BFS traversal (the nodes visited in order) will be printed by the bfs method itself.
5. **print()**: This prints a newline character to separate the BFS output from the DFS output.
6. **print("DFS traversal:", end=" ")**: Similar to the BFS print statement, this prints the label "DFS traversal: " without a newline at the end.
7. **g.dfs(2)**: This line calls the dfs method on the graph g, starting the Depth-First Search traversal from node 2. The output of

the DFS traversal will be printed by the dfs method.

8. **print()**: This prints a final newline character after the DFS output.

In essence, this code block sets up a small example graph and then shows how to traverse it using the two implemented algorithms (BFS and DFS) starting from a specific node (node 2).