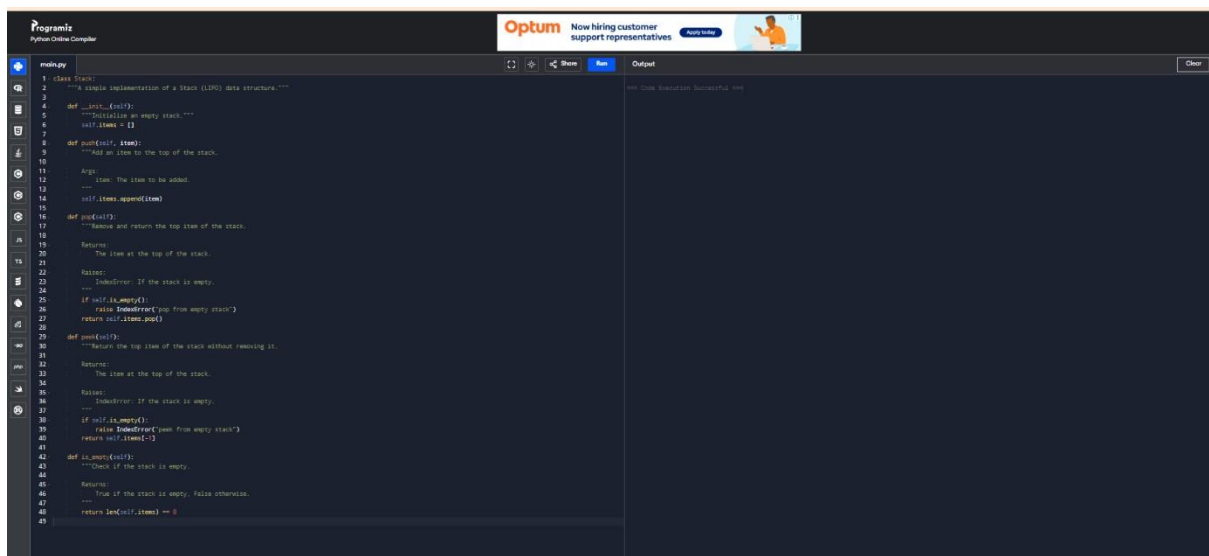


Assignment 11.4

Task-1:

1. Code skeleton with **Google-style docstrings**
2. Sample **test code** for stack operations
3. Suggested **optimizations and alternatives**

1. Stack Class in Python (Using List)



```
1 class Stack:
2     """A simple implementation of a stack (LIFO) data structure."""
3
4     def __init__(self):
5         """Initialize an empty stack."""
6         self.items = []
7
8     def push(self, item):
9         """Add an item to the top of the stack.
10
11         Args:
12             item: The item to be added.
13
14         """
15         self.items.append(item)
16
17     def pop(self):
18         """Remove and return the top item of the stack.
19
20         Returns:
21             The item at the top of the stack.
22
23         Raises:
24             IndexError: If the stack is empty.
25
26         """
27         if self.is_empty():
28             raise IndexError("pop from empty stack")
29         return self.items.pop()
30
31     def peek(self):
32         """Return the top item of the stack without removing it.
33
34         Returns:
35             The item at the top of the stack.
36
37         Raises:
38             IndexError: If the stack is empty.
39
40         """
41         if self.is_empty():
42             raise IndexError("peek from empty stack")
43         return self.items[-1]
44
45     def is_empty(self):
46         """Check if the stack is empty.
47
48         Returns:
49             True if the stack is empty, False otherwise.
50
51         """
52         return len(self.items) == 0
```

2. Sample **test code** for stack operations

```
Programiz
Python Online Compiler

main.py
1 class Stack:
2     def __init__(self):
3         self.items = []
4
5     def push(self, item):
6         self.items.append(item)
7
8     def pop(self):
9         if self.is_empty():
10            raise IndexError("Pop from empty stack")
11        return self.items.pop()
12
13    def peek(self):
14        if self.is_empty():
15            raise IndexError("Peek from empty stack")
16        return self.items[-1]
17
18    def is_empty(self):
19        return len(self.items) == 0
20
21
22 if __name__ == "__main__":
23     stack = Stack()
24
25     print("Pushing 1, 2, 3 onto stack...")
26     stack.push(1)
27     stack.push(2)
28     stack.push(3)
29
30     print("Top item (should be 3):", stack.peek())
31
32     print("Popping item (should be 3):", stack.pop())
33     print("Top item now (should be 2):", stack.peek())
34
35     print("Is the stack empty? (should be False):", stack.is_empty())
36
37     stack.pop()
38     stack.pop()
39
40     print("Is the stack empty? (should be True):", stack.is_empty())
41
42     # Uncomment to see error handling:
43     # stack.pop() # Should raise IndexError
44
```

```
Output
Pushing 1, 2, 3 onto stack...
Top item (should be 3): 3
Popping item (should be 3): 3
Top item now (should be 2): 2
Is the stack empty? (should be False): False
Is the stack empty? (should be True): True

=== Code Execution Successful ===
```

3. Suggested optimizations and alternatives

```
Programiz
Python Online Compiler

main.py
1 from collections import deque
2
3 class StackDeque:
4     """A stack implementation using collections.deque for better performance."""
5
6     def __init__(self):
7         """Initialize an empty stack using deque."""
8         self.items = deque()
9
10    def push(self, item):
11        """Add item to top of the stack."""
12        self.items.append(item)
13
14    def pop(self):
15        """Remove and return top item. Raise error if empty."""
16        if self.is_empty():
17            raise IndexError("pop from empty stack")
18        return self.items.pop()
19
20    def peek(self):
21        """Return top item without removing it."""
22        if self.is_empty():
23            raise IndexError("peek from empty stack")
24        return self.items[-1]
25
26    def is_empty(self):
27        """Return True if stack is empty."""
28        return not self.items
29
```

```
Output
=== Code Execution Successful ===
```

Cider House Dreams: A U.K. Country House With 77 Acres of Orchards Lists for £3.25 Million
Sponsored by: Wanson Global

Programiz PRO

Don't fall prey to fake information
Beware of imposters and protect yourself from fake schemes and frauds
Learn more

Task -2:

1.Queue python coding

Feature	QueueList (List)	QueueDequeue (collections.deque)
enqueue()	O(1)	O(1)
dequeue()	O(n) (due to shifting)	O(1)
Memory Overhead	Low	Slightly more (due to blocks)
Use Case & Stability	Small queues, low frequency use	High-performance, large queues frequency use

Task 33:

QueueList (List) QueueDeque (collections.deque)

```
[2] ✓ On
# Create a new linked list
my_list = LinkedList()

# Insert elements
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("Linked List after insertions:")
my_list.traverse()

# Delete a value
my_list.delete_value(20)
print("\nLinked List after deleting 20:")
my_list.traverse()

# Delete a value that doesn't exist
my_list.delete_value(50)

# Delete the head node
my_list.delete_value(10)
print("\nLinked List after deleting 10:")
my_list.traverse()

# Delete the last node
my_list.delete_value(40)
print("\nLinked List after deleting 40:")
my_list.traverse()

# Try deleting from an empty list
my_list.delete_value(30)
print("\nLinked List after deleting 30:")
my_list.traverse()

my_list.delete_value(100)

Linked List after insertions:
10 -> 20 -> 30 -> 40 -> None

Linked List after deleting 20:
10 -> 30 -> 40 -> None
Value 50 not found in the list.

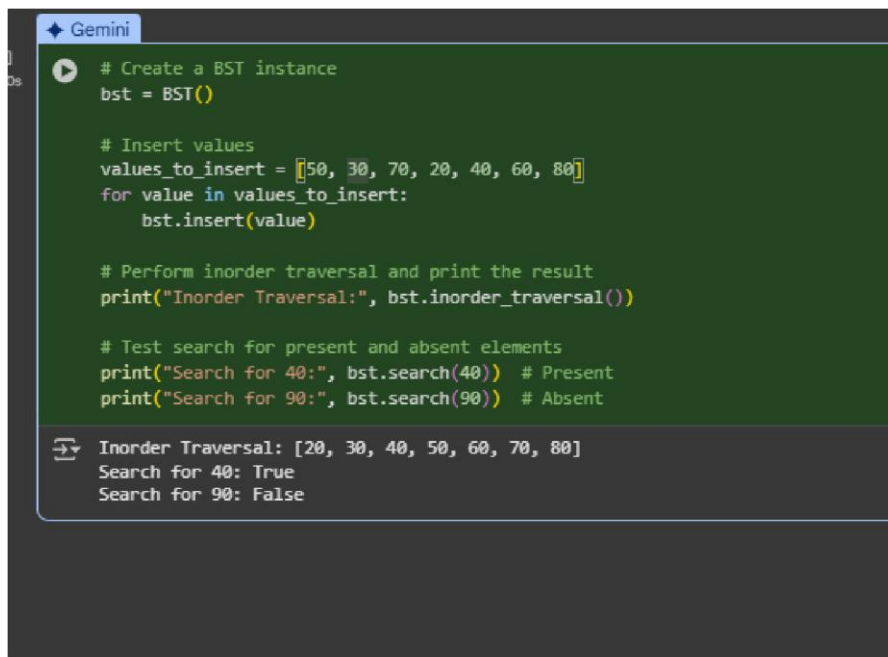
Linked List after deleting 10:
30 -> 40 -> None

Linked List after deleting 40:
30 -> None

Linked List after deleting 30:
List is empty.
List is empty. Nothing to delete.
```

Task-4::

Code & output::



The image shows a code editor window with a tab labeled "Gemini". The editor contains Python code for creating a Binary Search Tree (BST), inserting values, performing an inorder traversal, and testing search functionality. The code is as follows:

```
# Create a BST instance
bst = BST()

# Insert values
values_to_insert = [50, 30, 70, 20, 40, 60, 80]
for value in values_to_insert:
    bst.insert(value)

# Perform inorder traversal and print the result
print("Inorder Traversal:", bst.inorder_traversal())

# Test search for present and absent elements
print("Search for 40:", bst.search(40)) # Present
print("Search for 90:", bst.search(90)) # Absent
```

Below the code, the output is displayed:

```
Inorder Traversal: [20, 30, 40, 50, 60, 70, 80]
Search for 40: True
Search for 90: False
```

Task-5:

Code & output:

```

Untitled29.ipynb
File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[1]
✓ m
print("DFS starting from node (start_node):")

while queue:
    current_node = queue.popleft() # Get the next node from the queue

    if current_node not in visited:
        print(current_node, end=" ") # Process the current node
        visited.add(current_node) # Mark the current node as visited

        # Add neighbors to the queue that haven't been visited
        for neighbor in self.adjacency_list.get(current_node, []):
            if neighbor not in visited:
                queue.append(neighbor)

    print() # Newline for cleaner output

def dfs(self, start_node):
    """Returns a Depth-First Search starting from a given node (iterative approach)."""
    visited = set() # Keep track of visited nodes
    stack = [start_node] # Initialize a stack with the starting node

    print("DFS starting from node (start_node) (iterative):")

    while stack:
        current_node = stack.pop() # Get the next node from the stack

        if current_node not in visited:
            print(current_node, end=" ") # Process the current node
            visited.add(current_node) # Mark the current node as visited

            # Add unvisited neighbors to the stack in reverse order to explore them in the correct DFS order
            # We reverse because stack is LIFO, and we want to process neighbors in the order they appear
            for neighbor in reversed(self.adjacency_list.get(current_node, [])):
                if neighbor not in visited:
                    stack.append(neighbor)

        print() # Newline for cleaner output

def dfs_recursive(self, start_node):
    """Returns a Depth-First Search starting from a given node (recursive approach)."""
    visited = set() # Keep track of visited nodes
    print("DFS starting from node (start_node) (recursive):")
    self._dfs_recursive_helper(start_node, visited)
    print() # Newline for cleaner output

def _dfs_recursive_helper(self, current_node, visited):
    """Helper method for recursive DFS."""
    visited.add(current_node) # Mark the current node as visited
    print(current_node, end=" ") # Process the current node

    # Recursively visit unvisited neighbors
    for neighbor in self.adjacency_list.get(current_node, []):
        if neighbor not in visited:
            self._dfs_recursive_helper(neighbor, visited)

[2]
✓ m
# Example Adjacency List
adjacency_list = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}

# Create a Graph Instance
graph = Graph(adjacency_list)

# Perform DFS starting from 'A'
graph.dfs('A')

# Perform iterative DFS starting from 'A'
graph.dfs('A')

# Perform recursive DFS starting from 'A'
graph.dfs_recursive('A')

DFS starting from node A:
A B C D E F
DFS starting from node A (iterative):
A B D E F C
DFS starting from node A (recursive):
A B D E F C

```
