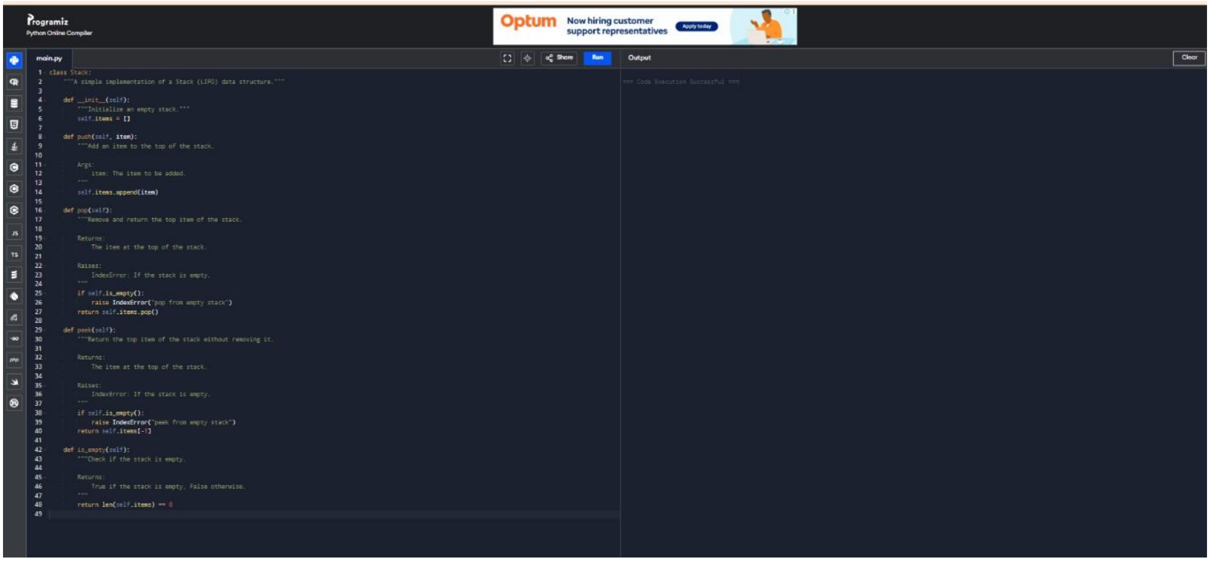
Assignment 11.4

Task-1:

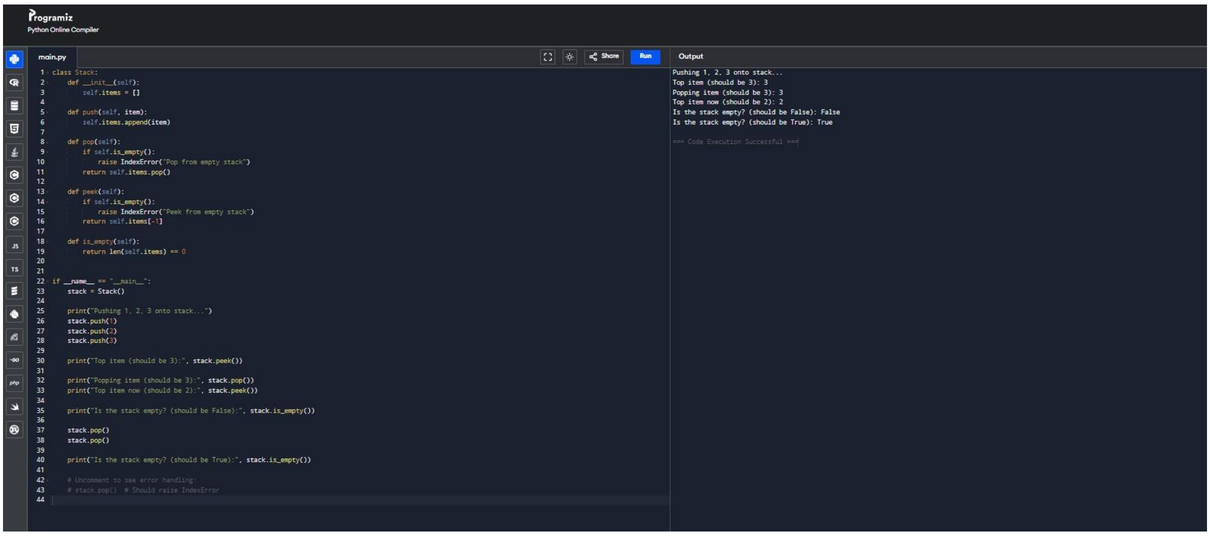
1. Code skeleton with **Google-style docstrings**
2. Sample **test code** for stack operations
3. Suggested **optimizations and alternatives**

---

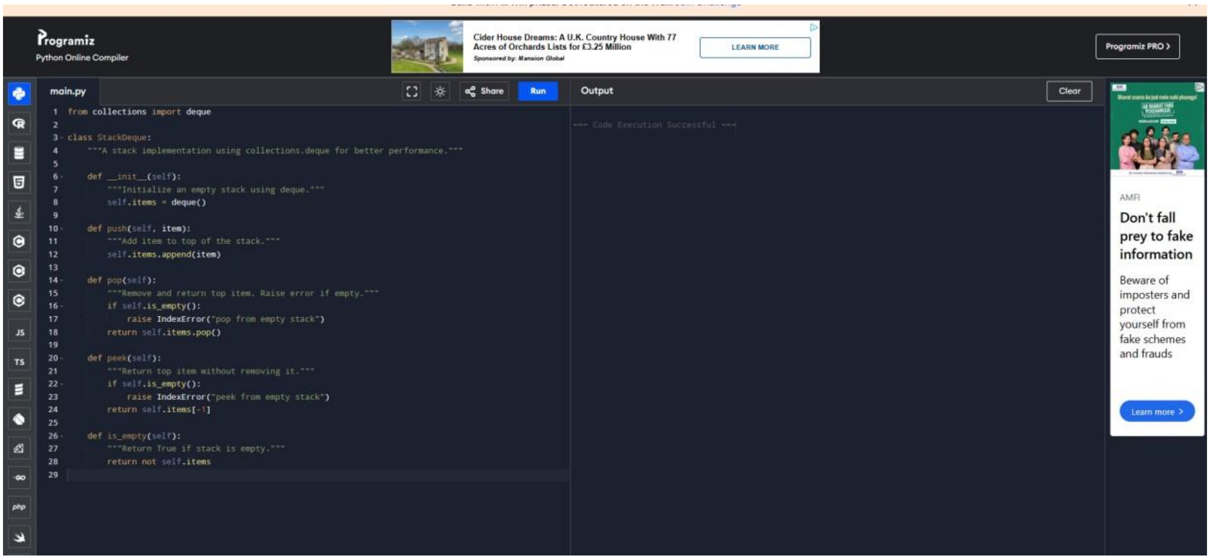
1. Stack Class in Python (Using List)



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

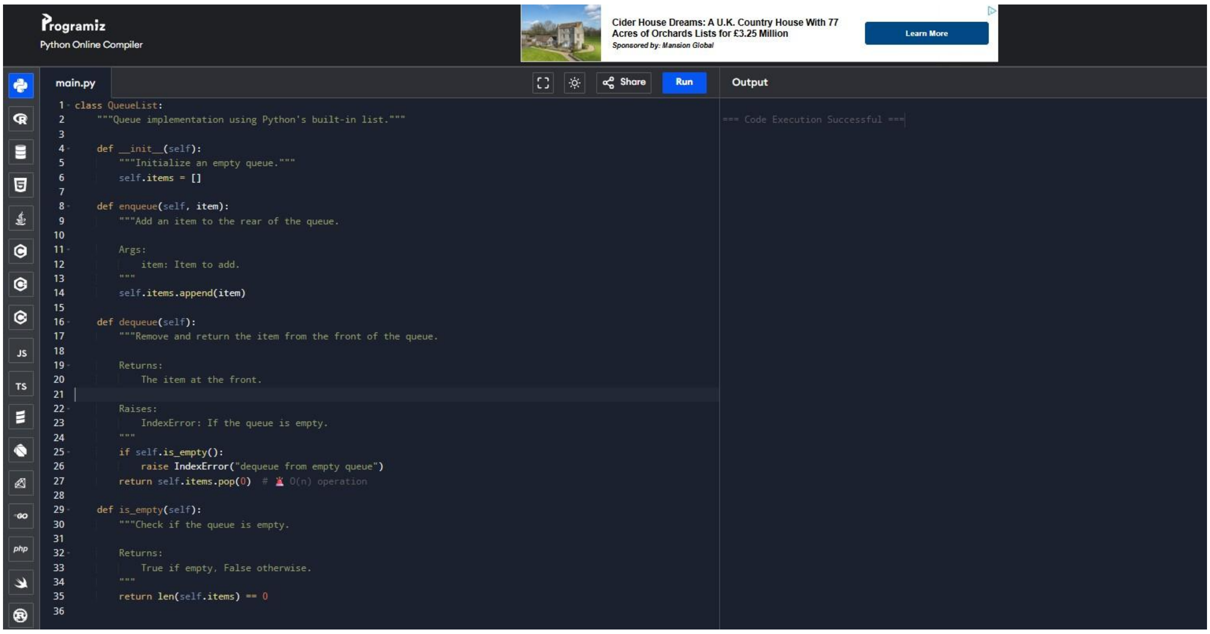


3. Suggested **optimizations and alternatives**

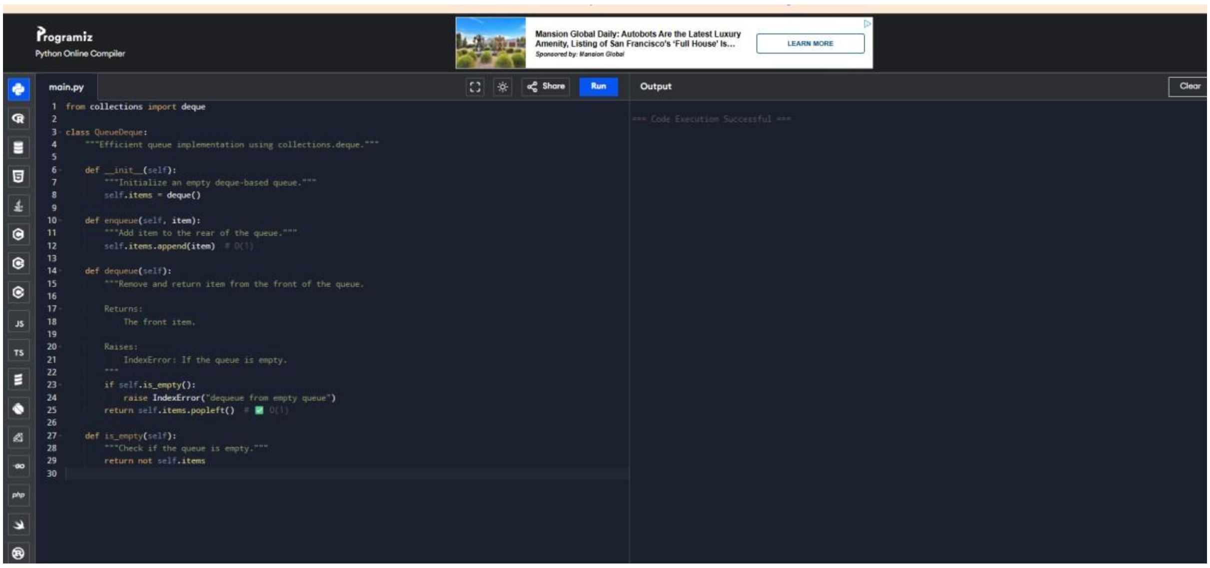


Task -2:

1.Queue python coding



2.optimized python coding



3.

QueueDeque

Feature	QueueList (List)	QueueDeque (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 Suitability	Small queues, low-frequency use	High-performance, large queues

Task 3:

QueueList (List)	QueueDeque (collections.deque)	Feature

```python
# 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:

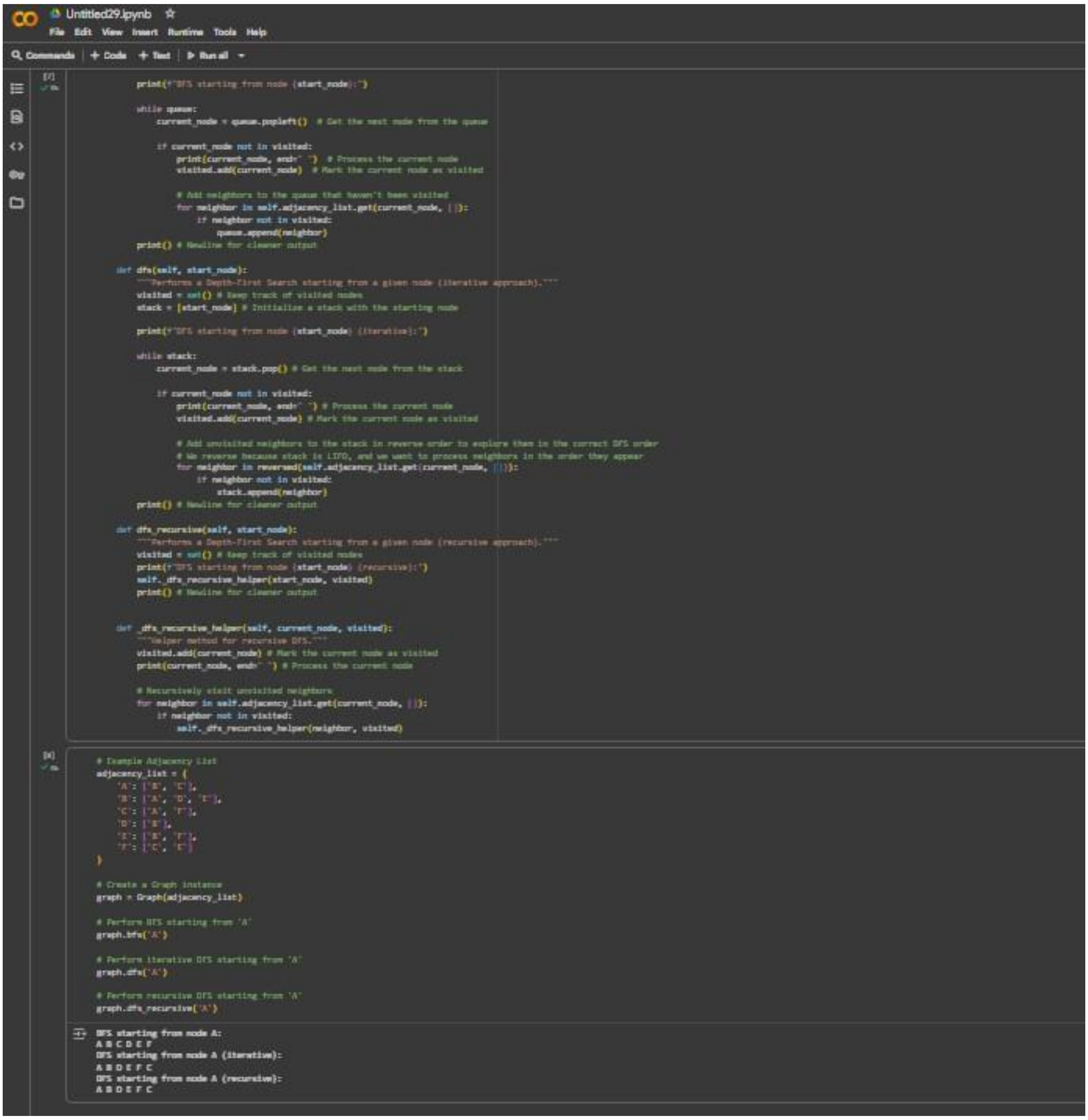
Code & output:

```python
# 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
```

```
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

```python
        print(f"BFS 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):
        """Performs 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(f"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):
        """Performs a Depth-First Search starting from a given node (recursive approach)."""
        visited = set() # Keep track of visited nodes
        print(f"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)
```

```python
# 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 BFS starting from 'A'
graph.bfs('A')

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

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

```
BFS 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
```

*****