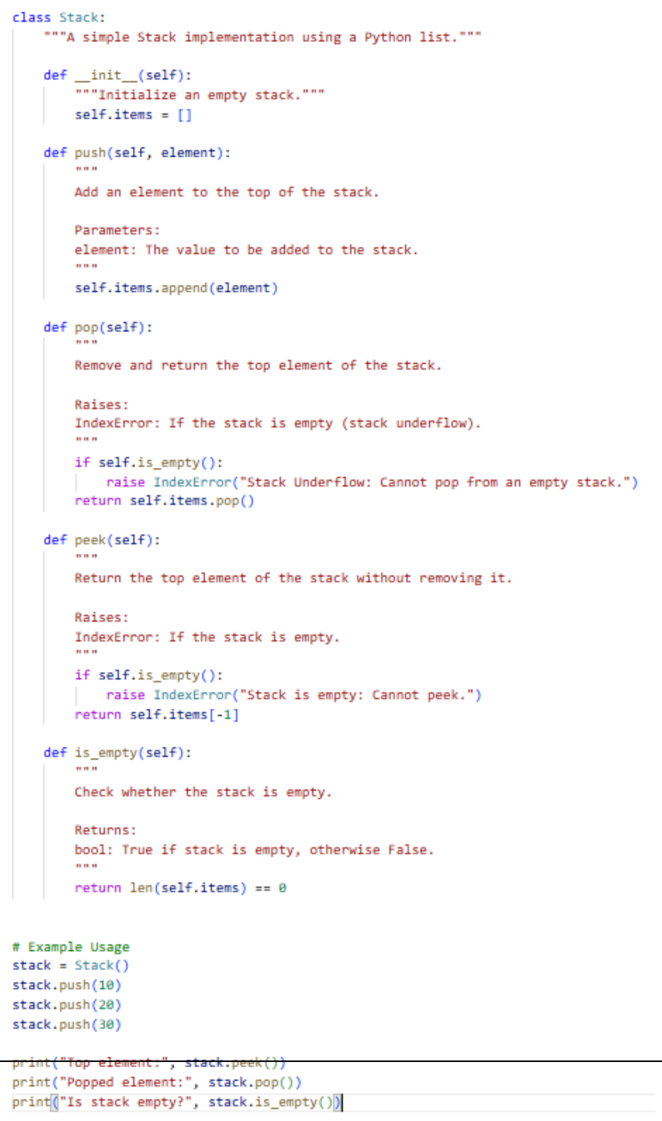
NAME: Mohammed Suhaib Iqbal
Batch-34
Roll no: 2303A52168
Ai assisment coding
Assignmnet :12

Task 1: (Data Structures – Stack Implementation with AI Assistance)

Prompt:

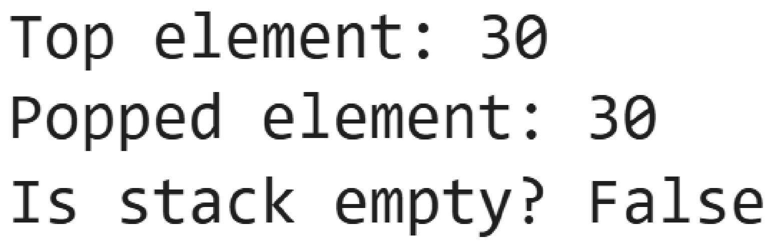
Generate a Python program that implements a Stack class with push, pop, peek, and is_empty methods including proper error handling for stack underflow. Include clear docstrings for each method and demonstrate usage with example operations.

Code:



```python
class Stack:
    """A simple Stack implementation using a Python list."""

    def __init__(self):
        """Initialize an empty stack."""
        self.items = []

    def push(self, element):
        """
        Add an element to the top of the stack.

        Parameters:
        element: The value to be added to the stack.
        """
        self.items.append(element)

    def pop(self):
        """
        Remove and return the top element of the stack.

        Raises:
        IndexError: If the stack is empty (stack underflow).
        """
        if self.is_empty():
            raise IndexError("Stack Underflow: Cannot pop from an empty stack.")
        return self.items.pop()

    def peek(self):
        """
        Return the top element of the stack without removing it.

        Raises:
        IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("Stack is empty: Cannot peek.")
        return self.items[-1]

    def is_empty(self):
        """
        Check whether the stack is empty.

        Returns:
        bool: True if stack is empty, otherwise False.
        """
        return len(self.items) == 0


# Example Usage
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)

print("Top element:", stack.peek())
print("Popped element:", stack.pop())
print("Is stack empty?", stack.is_empty())
```

Smple Input/Output:



```
Top element: 30
Popped element: 30
Is stack empty? False
```

Explanation:

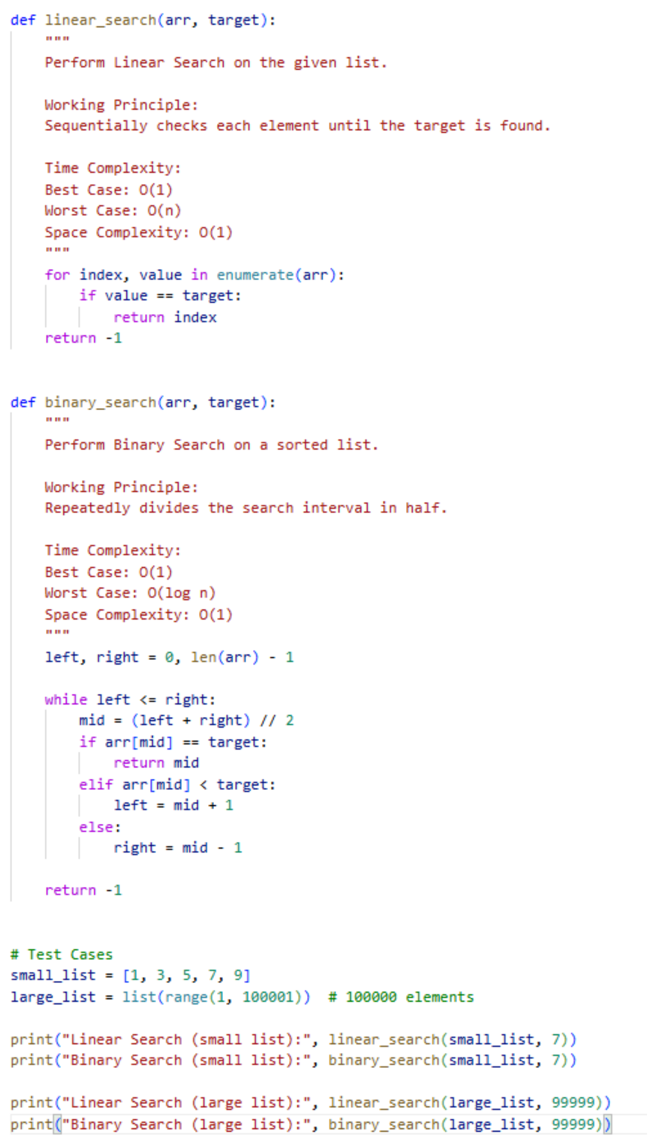
- A Stack class is created using a Python list as the underlying data structure.
- The __init__ method initializes an empty list to store stack elements.
- The push() method adds elements to the top of the stack using the append() function.
- The pop() method removes the top element and includes proper underflow error handling.
- The peek() method returns the top element without removing it and checks if the stack is empty.
- The is_empty() method returns a boolean value indicating whether the stack contains elements.
- Meaningful docstrings are provided for each method explaining purpose, parameters, return values, and exceptions.
- Example usage demonstrates stack operations and verifies correctness.

Task 2: **(Algorithms – Linear vs Binary Search Analysis)**

Prompt:

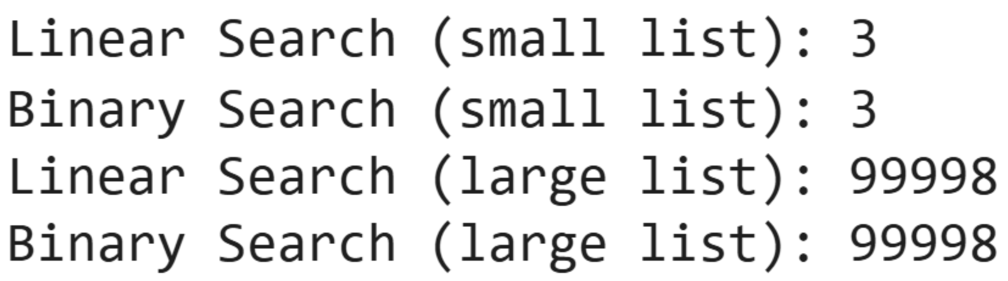
Generate Python implementations of linear_search(arr, target) and binary_search(arr, target) with proper docstrings explaining their working principles and time complexity. Include test cases with different input sizes and print results to compare correctness and behavior.

Code:



```python
def linear_search(arr, target):
    """
    Perform Linear Search on the given list.

    Working Principle:
    Sequentially checks each element until the target is found.

    Time Complexity:
    Best Case: O(1)
    Worst Case: O(n)
    Space Complexity: O(1)
    """
    for index, value in enumerate(arr):
        if value == target:
            return index
    return -1


def binary_search(arr, target):
    """
    Perform Binary Search on a sorted list.

    Working Principle:
    Repeatedly divides the search interval in half.

    Time Complexity:
    Best Case: O(1)
    Worst Case: O(log n)
    Space Complexity: O(1)
    """
    left, right = 0, len(arr) - 1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1


# Test Cases
small_list = [1, 3, 5, 7, 9]
large_list = list(range(1, 100001))  # 100000 elements

print("Linear Search (small list):", linear_search(small_list, 7))
print("Binary Search (small list):", binary_search(small_list, 7))

print("Linear Search (large list):", linear_search(large_list, 99999))
print("Binary Search (large list):", binary_search(large_list, 99999))
```

Smple Input/Output:



```
Linear Search (small list): 3
Binary Search (small list): 3
Linear Search (large list): 99998
Binary Search (large list): 99998
```

Explanation:

- Two search algorithms are implemented: Linear Search and Binary Search.

- Linear Search checks elements sequentially until the target is found or the list ends.
- Binary Search works only on sorted arrays and repeatedly divides the search space into halves.
- Linear Search has a worst-case time complexity of O(n), making it slower for large datasets.
- Binary Search has a worst-case complexity of O(log n), making it much faster for large sorted data.
- Both functions return the index of the target element or -1 if not found.
- Test cases are provided for both small and large input sizes to verify correctness.
- The comparison demonstrates how Binary Search performs more efficiently as input size increases.

Task 3: (Test Driven Development – Simple Calculator Function)

Prompt:

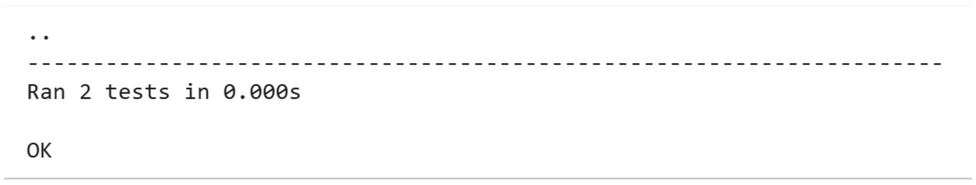
Generate Python unit test cases for addition and subtraction functions using unittest before implementing them. Then implement the calculator functions so that all tests pass successfully and demonstrate test execution.

Code:

```
def add(a, b):
    """
    Return the sum of two numbers.
    Time Complexity: O(1)
    Space Complexity: O(1)
    """
    return a + b

def subtract(a, b):
    """
    Return the difference between two numbers.
    Time Complexity: O(1)
    Space Complexity: O(1)
    """
    return a – b
```

```
import unittest
from calculator import add, subtract

class TestCalculator(unittest.TestCase):

    def test_add(self):
        self.assertEqual(add(5, 3), 8)
        self.assertEqual(add(-1, 1), 0)

    def test_subtract(self):
        self.assertEqual(subtract(10, 4), 6)
        self.assertEqual(subtract(0, 5), -5)

if __name__ == "__main__":
    unittest.main()
```

<u>Smple Input/Output</u>:



```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

<u>Explanation</u>:

- Test Driven Development (TDD) begins by writing unit tests before implementation.
- The unittest framework is used to define structured test cases.
- Test cases check both positive and negative scenarios for addition and

subtraction.

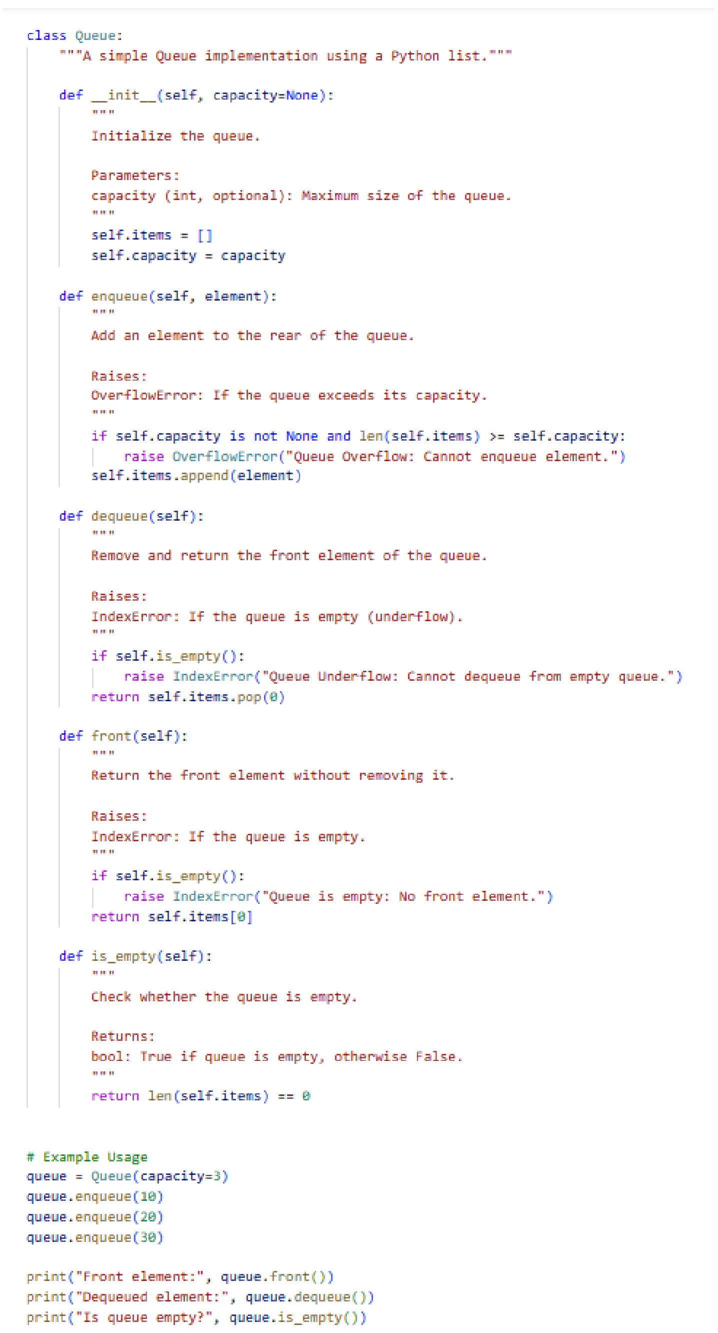
- Initially, tests fail because the implementation file does not exist.
- The calculator.py file is then created with add and subtract functions.
- After implementation, tests are executed again to verify correctness.
- Both functions pass all test cases, confirming successful development.
- This approach ensures reliability, reduces bugs, and promotes clean software design.

Task 4: (Data Structures – Queue Implementation with AI Assistance)

Prompt:

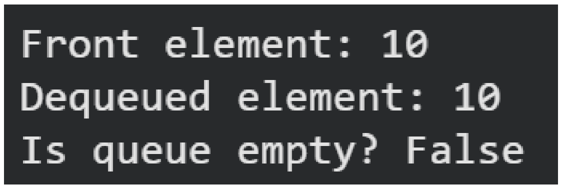
Generate a Python program that implements a Queue class with enqueue, dequeue, front, and is_empty methods including proper handling of overflow and underflow conditions. Include clear docstrings for each method and demonstrate the queue operations with example usage.

Code:



```python
class Queue:
    """A simple Queue implementation using a Python list."""

    def __init__(self, capacity=None):
        """
        Initialize the queue.

        Parameters:
        capacity (int, optional): Maximum size of the queue.
        """
        self.items = []
        self.capacity = capacity

    def enqueue(self, element):
        """
        Add an element to the rear of the queue.

        Raises:
        OverflowError: If the queue exceeds its capacity.
        """
        if self.capacity is not None and len(self.items) >= self.capacity:
            raise OverflowError("Queue Overflow: Cannot enqueue element.")
        self.items.append(element)

    def dequeue(self):
        """
        Remove and return the front element of the queue.

        Raises:
        IndexError: If the queue is empty (underflow).
        """
        if self.is_empty():
            raise IndexError("Queue Underflow: Cannot dequeue from empty queue.")
        return self.items.pop(0)

    def front(self):
        """
        Return the front element without removing it.

        Raises:
        IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Queue is empty: No front element.")
        return self.items[0]

    def is_empty(self):
        """
        Check whether the queue is empty.

        Returns:
        bool: True if queue is empty, otherwise False.
        """
        return len(self.items) == 0


# Example Usage
queue = Queue(capacity=3)
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

print("Front element:", queue.front())
print("Dequeued element:", queue.dequeue())
print("Is queue empty?", queue.is_empty())
```

<u>Smple Input/Output</u>:



```
Front element: 10
Dequeued element: 10
Is queue empty? False
```

<u>Explanation</u>:

- A Queue class is implemented using a Python list to store elements.
- The enqueue() method adds elements to the rear of the queue and checks for overflow using the defined capacity.

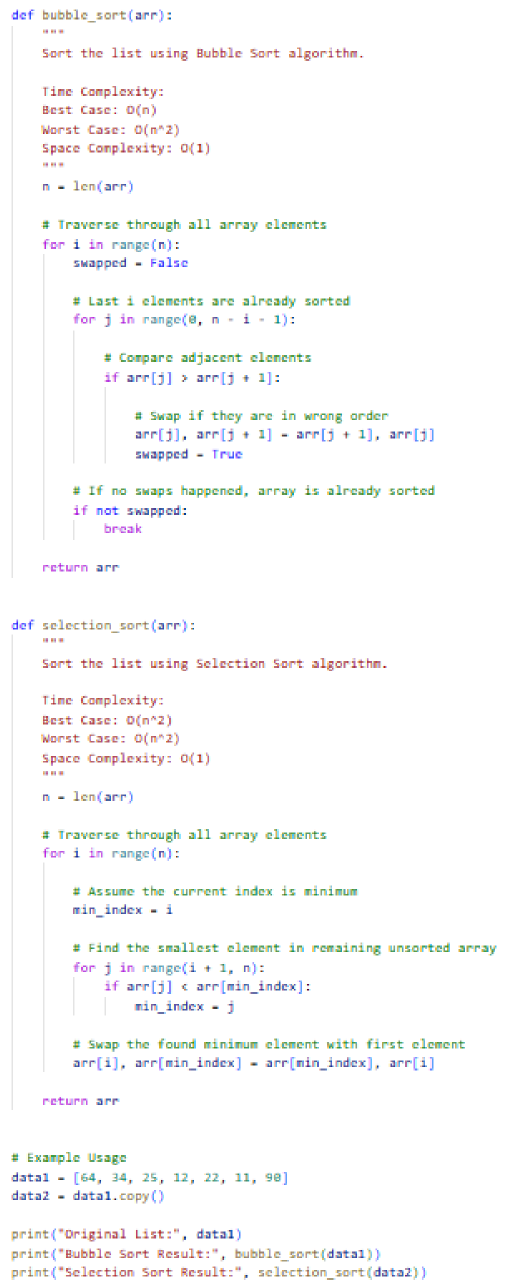
- The dequeue() method removes elements from the front and handles underflow if the queue is empty.
- The front() method returns the first element without removing it.
- The is_empty() method checks whether the queue contains elements.
- Overflow is handled using OverflowError when capacity is exceeded.
- Underflow is handled using IndexError when attempting to remove from an empty queue.
- Example usage demonstrates correct functionality and validates queue behavior.

Task 5: **(Algorithms – Bubble Sort vs Selection Sort)**

Prompt:

Generate Python implementations of bubble_sort(arr) and selection_sort(arr) including step-by-step comments and docstrings mentioning time and space complexity. Demonstrate both algorithms with example input and compare their behavior.

Code:



```python
def bubble_sort(arr):
    """
    Sort the list using Bubble Sort algorithm.

    Time Complexity:
    Best Case: O(n)
    Worst Case: O(n^2)
    Space Complexity: O(1)
    """
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):
        swapped = False

        # Last i elements are already sorted
        for j in range(0, n - i - 1):

            # Compare adjacent elements
            if arr[j] > arr[j + 1]:

                # Swap if they are in wrong order
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # If no swaps happened, array is already sorted
        if not swapped:
            break

    return arr


def selection_sort(arr):
    """
    Sort the list using Selection Sort algorithm.

    Time Complexity:
    Best Case: O(n^2)
    Worst Case: O(n^2)
    Space Complexity: O(1)
    """
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Assume the current index is minimum
        min_index = i

        # Find the smallest element in remaining unsorted array
        for j in range(i + 1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        # Swap the found minimum element with first element
        arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr


# Example Usage
data1 = [64, 34, 25, 12, 22, 11, 90]
data2 = data1.copy()

print("Original List:", data1)
print("Bubble Sort Result:", bubble_sort(data1))
print("Selection Sort Result:", selection_sort(data2))
```

Smple Input/Output:

```
Original List: [64, 34, 25, 12, 22, 11, 90]
Bubble Sort Result: [11, 12, 22, 25, 34, 64, 90]
Selection Sort Result: [11, 12, 22, 25, 34, 64, 90]
```

Explanation:

- Two sorting algorithms are implemented: Bubble Sort and Selection Sort.
- Bubble Sort repeatedly compares adjacent elements and swaps them if

they are in the wrong order.

- It includes an optimization using a swapped flag to stop early if the list becomes sorted.
- Selection Sort repeatedly selects the minimum element from the unsorted portion and places it correctly.
- Bubble Sort has a best-case complexity of O(n) when the list is already sorted.
- Selection Sort has $O(n^2)$ time complexity in both best and worst cases.
- Both algorithms use O(1) extra space since sorting is done in-place.
- Example usage demonstrates correctness and allows comparison of behavior.