SR UNIVERSITY

School of Computer Science and Artificial Intelligence

Lab Assignment # 12

Name of Student : **A.JEEVAN SAI**
Enrollment No. : **2303A51420**
Batch No. : **21**

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

- **Task: Use AI assistance to generate a Python program that implements a Stack data structure.**

Instructions: Prompt AI to create a Stack class with the following methods:

- **push(element), pop(), peek(), is_empty()**
- **Ensure proper error handling for stack underflow.**
- **Ask AI to include clear docstrings for each method. Expected Output:**
- **A functional Python program implementing a Stack using a class.**
- **Properly documented methods with docstrings.**

Prompt:-

Create a well-structured Python program that defines a Stack class with push, pop, peek, is_empty, and size methods. Ensure proper handling of stack underflow and overflow conditions, and include clear and concise docstrings for each method.

Code:-

```python
'''Create a well-structured Python program that defines a Stack class with push, pop, peek, is_empty, and size methods.
Ensure proper handling of stack underflow and overflow conditions, and include clear and concise docstrings for each method.'''
class Stack:
    def __init__(self, capacity):
        """Initialize the stack with a given capacity."""
        self.capacity = capacity
        self.stack = []
    def push(self, item):
        """Add an item to the top of the stack. Raises an exception if the stack is full."""
        if len(self.stack) >= self.capacity:
            raise OverflowError("Stack overflow: Cannot push item, stack is full.")
        self.stack.append(item)
    def pop(self):
        """Remove and return the item at the top of the stack. Raises an exception if the stack is empty."""
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot pop item, stack is empty.")
        return self.stack.pop()
    def peek(self):
        """Return the item at the top of the stack without removing it. Raises an exception if the stack is empty."""
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot peek item, stack is empty.")
        return self.stack[-1]
    def is_empty(self):
        """Return True if the stack is empty, False otherwise."""
        return len(self.stack) == 0
    def size(self):
        """Return the number of items currently in the stack."""
        return len(self.stack)
# Example usage:
if __name__ == "__main__":
    stack = Stack(5)
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print("Top item:", stack.peek())  # Output: Top item: 3
    print("Stack size:", stack.size())  # Output: Stack size: 3
    print("Popped item:", stack.pop())  # Output: Popped item: 3
    print("Is stack empty?", stack.is_empty())  # Output: Is stack empty? False
    stack.pop()
    stack.pop()
    print("Is stack empty?", stack.is_empty())  # Output: Is stack empty? True
    # Uncommenting the following line will raise an exception due to stack underflow
    # stack.pop()
    # Uncommenting the following line will raise an exception due to stack overflow
    # stack.push(4)
```

Output:-

```
Is stack empty? True
PS C:\Users\lenovo\AppData\Local\Programs\Microsoft VS Code> &
Data/Local/Microsoft/Windows/INetCache/IE/RAYTNS63/12.2.py
Top item: 3
Stack size: 3
Popped item: 3
Is stack empty? False
Is stack empty? True
```

Justification:-

The Stack class is implemented with methods for pushing, popping, peeking, checking if the stack is empty, and getting the size of the stack. Each method includes error handling for stack overflow and underflow conditions. The example usage demonstrates how to use the Stack class and handles edge cases effectively. the time complexity for push and pop operations is O(1), while peek, isEmpty, and size operations also have a time complexity of O(1). The space complexity is O(n) where n is the number of items in the stack.

SR UNIVERSITY

Task Description -2 (Algorithms – Linear vs Binary Search Analysis)

- **Task: Use AI to implement and compare Linear Search and Binary Search algorithms in Python. Instructions:**
- **Prompt AI to generate:**
- **linear_search(arr, target) ➢ binary_search(arr, target)**
- **Include docstrings explaining:**
- **Working principle**
- **Test both algorithms using different input sizes.**

Expected Output:

- **Python implementations of both search algorithms.**
- **AI-generated comments and complexity analysis.**
- **Test results showing correctness and comparison.**

Prompt:-

Develop a Python program that implements and compares Linear Search and Binary Search algorithms. Include separate functions for both methods, allow testing with different input sizes, measure execution time, handle edge cases properly, and provide descriptive docstrings explaining their working principles and complexity.

Justification:-

The program implements both linear search and binary search algorithms, allowing us to compare their performance with different input sizes. The time complexity of linear search is O(n), while the time complexity of binary search is O(log n). The program also handles edge cases effectively, such as searching for an element that is not present in the list. Each function

includes a docstring explaining its functionality, and the example usage demonstrates how to use both search methods and measure their execution time.

Code:-

```python
'''Develop a Python program that implements and compares Linear Search and Binary Search algorithms.
Include separate functions for both methods, allow testing with different input sizes, measure execution time, handle edge cases
properly, and provide descriptive docstrings explaining their working principles and complexity.'''
import time
def linear_search(arr, target):
    """
    Perform a linear search for the target element in the given array.

    Parameters:
    arr (list): The list of elements to search through.
    target: The element to search for.

    Returns:
    int: The index of the target element if found, otherwise -1.

    Complexity:
    Time Complexity: O(n) - In the worst case, it checks each element once.
    Space Complexity: O(1) - No additional space is used.
    """
    for index in range(len(arr)):
        if arr[index] == target:
            return index
    return -1
def binary_search(arr, target):
    """
    Perform a binary search for the target element in the given sorted array.

    Parameters:
    arr (list): The sorted list of elements to search through.
    target: The element to search for.

    Returns:
    int: The index of the target element if found, otherwise -1.

    Complexity:
    Time Complexity: O(log n) - In each step,it eliminates half of the remaining elements.
    Space Complexity: O(1) - No additional space is used.
    """
    low = 0
    high = len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
#Example usage and testing
if __name__ == "__main__":
    # Test data
    arr = [1,2,3,4,5,6,7,8,9,10]
    target = 5

    # Testing Linear Search
    start_time = time.time()
    result_linear = linear_search(arr, target)
    end_time = time.time()
    print(f"Linear Search: Target found at index {result_linear}, Time taken: {end_time - start_time:.6f} seconds")

    # Testing Binary Search
    start_time = time.time()
    result_binary = binary_search(arr, target)
    end_time = time.time()
    print(f"Binary Search: Target found at index {result_binary}, Time taken: {end_time - start_time:.6f} seconds")
    # Edge case: Target not in the array
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
inary Search: Target found at index 4, Time taken: 0.000004 seconds
PS C:\Users\lenovo\AppData\Local\Programs\Microsoft VS Code>
```

Output:-

```
Binary Search: Target found at index 4, Time taken: 0.000014 seconds
PS C:\Users\lenovo\AppData\Local\Programs\Microsoft VS Code> & C:\Users\le
pData/Local/Microsoft/Windows/INetCache/IE/RAYTNS63/'''Develop a Python pr
Linear Search: Target found at index 4, Time taken: 0.000015 seconds
Binary Search: Target found at index 4, Time taken: 0.000010 seconds
PS C:\Users\lenovo\AppData\Local\Programs\Microsoft VS Code>
```

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

Apply Test Driven Development (TDD) using AI assistance to develop a calculator function.

SR UNIVERSITY

Instructions:

➢ **Prompt AI to first generate unit test cases for addition and subtraction.**

➢ **Run the tests and observe failures.**

➢ **Ask AI to implement the calculator functions to pass all tests.**

➢ **Re-run the tests to confirm success.**

Expected Output:

➢ **Separate test file and implementation file.**

➢ **Test cases executed before implementation.**

➢ **Final implementation passing all test cases.**

Prompt:-

Generate Python unit test cases for a calculator function that performs addition and subtraction operations following Test-Driven Development principles. Do not provide the implementation initially; only generate the test cases with clear docstrings.

Code:-

```
'''Generate Python unit test cases for a calculator function that performs addition and subtraction operations following
Test-Driven Development principles. Do not provide the implementation initially;
only generate the test cases with clear docstrings'''
import unittest

def calculator(a, b, operation):
    if operation == 'add':
        return a + b
    if operation == 'subtract':
        return a - b
    raise ValueError("operation must be 'add' or 'subtract'")


class TestCalculator(unittest.TestCase):
    def test_addition(self):
        """Test that the calculator correctly adds two numbers."""
        self.assertEqual(calculator(2, 3, 'add'), 5)
        self.assertEqual(calculator(-1, 1, 'add'), 0)
        self.assertEqual(calculator(0, 0, 'add'), 0)

    def test_subtraction(self):
        """Test that the calculator correctly subtracts two numbers."""
        self.assertEqual(calculator(5, 2, 'subtract'), 3)
        self.assertEqual(calculator(1, -1, 'subtract'), 2)
        self.assertEqual(calculator(0, 0, 'subtract'), 0)
if __name__ == '__main__':   unittest.main()
```

Output:-

```
pData/Local/Microsoft/Windows/INetCache/IE/RAYTNS63/'''Generate Python unit
..
----------------------------------------------------------------------
Ran 2 tests in 0.002s

OK
PS C:\Users\lenovo\AppData\Local\Programs\Microsoft VS Code>
```

Justification:-
The program defines a calculator function that performs basic arithmetic operations based on the given operation. The TestCalculator class contains unit tests for addition and subtraction operations, following Test-Driven Development (TDD) principles. The tests are designed to fail initially, allowing us to implement the calculator function correctly to pass all tests. Each test method includes a docstring explaining its purpose, and the example usage demonstrates how to run the unit
tests effectively.

Task Description -4 (Data Structures – Queue Implementation with AI Assistance) ➢ Task:
Use AI assistance to generate a Python program that implements a Queue datastructure.
Instructions: Prompt AI to create a Queue class with the following methods:
• enqueue(element), dequeue(), front(), is_empty()
➢ Handle queue overflow and underflow conditions.
➢ Include appropriate docstrings for all methods.
Expected Output:
➢ A fully functional Queue implementation in Python.
➢ Proper error handling and documentation.

Prompt:-
Write a Python program that defines a Queue class with enqueue, dequeue, front, is_empty, and size methods. Ensure proper error handling for queue underflow and overflow conditions and include concise docstrings for all methods.

Code:-

SR UNIVERSITY

```python
'''Write a Python program that defines a Queue class with enqueue, dequeue, front, is_empty, and size methods. Ensure proper error
handling for queue underflow and overflow conditions and include concise docstrings for all methods.'''
class Queue:
    """A simple implementation of a queue data structure."""

    def __init__(self, capacity):
        """Initialize the queue with a specified capacity."""
        self.capacity = capacity
        self.queue = []

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

        Raises:
            OverflowError: If the queue is full.
        """
        if self.size() >= self.capacity:
            raise OverflowError("Queue overflow: Cannot enqueue item, the queue is full.")
        self.queue.append(item)

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

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

    def front(self):
        """Return the item at the front of the queue without removing it.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot access front item, the queue is empty.")
        return self.queue[0]
    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.queue) == 0
    def size(self):
        """Return the number of items in the queue."""
        return len(self.queue)
# Example usage:
if __name__ == "__main__":
    q = Queue(5)
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print("Front item:", q.front())  # Output: Front item: 1
    print("Queue size:", q.size())   # Output: Queue size: 3
    print("Dequeue item:", q.dequeue())  # Output: Dequeue item: 1
    print("Is queue empty?", q.is_empty())  # Output: Is queue empty? False
    q.dequeue()
    q.dequeue()
    print("Is queue empty?", q.is_empty())  # Output: Is queue empty? True
    try:
        q.dequeue()  # This will raise an error since the queue is empty
    except IndexError as e:
        print(e)  # Output: Queue underflow: Cannot dequeue item, the queue is empty.
    try:        q.front()  # This will also raise an error since the queue is empty
    except IndexError as e:
        print(e)  # Output: Queue underflow: Cannot access front item, the queue is empty.
    q.enqueue(4)
    q.enqueue(5)
```

Output:-

```
n313\python.exe "c:/Users/lenovo/AppData/Local/Microsoft/Windo
hat defines a.py"
Front item: 1
Queue size: 3
Dequeue item: 1
Is queue empty? False
Is queue empty? True
Queue underflow: Cannot dequeue item, the queue is empty.
Queue underflow: Cannot access front item, the queue is empty.
```

Justification:-

The Queue class is implemented with methods for enqueuing, dequeuing, peeking, checking if the queue is empty, and getting the size of the queue. Each method includes error handling

for queue overflow and underflow conditions. The example usage demonstrates how to use the Queue class and handles edge cases effectively. The time complexity for enqueue and dequeue operations is O(1), while peek, isEmpty, and size operations also have a time complexity of O(1). The space complexity is O(n) where n is the number of items in the queue.

Task Description -5 (Algorithms – Bubble Sort vs Selection Sort) ➢
Task:
Use AI to implement Bubble Sort & Selection Sort algorithm & compare their behavior.
Instructions:
• bubble_sort(arr), • selection_sort(arr)
➢ Include comments explaining each step.
➢ Add docstrings mentioning time and space complexity.
Expected Output:
• Correct Python implementations of both sorting algorithms.
• Complexity analysis in docstrings.

Prompt:-
Create a Python program that implements Bubble Sort and Selection Sort algorithms. Include functions for both methods, test them with different input sizes, measure execution time, handle edge cases such as empty lists and duplicates, and provide docstrings mentioning time and space complexity.

Justification:-
The program implements both bubble sort and selection sort algorithms, allowing us to compare their performance with different input sizes. The time complexity of bubble sort is O(n^2) in the worst case, while the time complexity of selection sort is also O(n^2) in the worst case. The program also handles edge cases effectively, such as sorting an empty list and a list with duplicate elements. Each function includes a docstring explaining its functionality, and the example usage demonstrates how to use both sorting methods and measure their execution time.

SR UNIVERSITY

Code:-

```python
'''Create a Python program that implements Bubble Sort and Selection Sort algorithms.
Include functions for both methods, test them with different input sizes, measure execution time,
handle edge cases such as empty lists and duplicates, and provide docstrings mentioning time and
space complexity.'''
import time
def bubble_sort(arr):
    """
    Sorts a list using the Bubble Sort algorithm.

    Time Complexity: O(n^2) in the worst and average cases, O(n) in the best case (when the list is already sorted).
    Space Complexity: O(1) - Bubble Sort is an in-place sorting algorithm.

    Args:
        arr (list): The list to be sorted.
    Returns:
        list: The sorted list.
    """
    n = len(arr)
    for i in range(n):
        # Flag to check if any swapping occurs
        swapped = False
        for j in range(0, n-i-1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        # If no swapping occurred, the list is already sorted
        if not swapped:
            break
    return arr
def selection_sort(arr):
    """Sorts a list using the Selection Sort algorithm.
    Time Complexity: O(n^2) in all cases (best, average, and worst).
    Space Complexity: O(1) - Selection Sort is an in-place sorting algorithm.
    Args:
        arr (list): The list to be sorted.
    Returns:
        list: The sorted list.
    """
    n = len(arr)
    for i in range(n):
        # Find the minimum element in the unsorted portion of the list
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j
        # Swap the found minimum element with the first element of the unsorted portion
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
    return arr
# Test the sorting algorithms with different input sizes and edge cases
if __name__ == "__main__":
    test_cases = [
        [],  # Empty list
        [5],  # Single element
        [3, 1, 2, 1, 3],  # List with duplicates
        [64, 34, 25, 12, 22, 11, 90],  # Random list
        [1, 2, 3, 4, 5],  # Already sorted list
        [5, 4, 3, 2, 1]   # Reverse sorted list
```

Output:-

```
--------------------------------------------------
Original list: [1, 2, 3, 4, 5]
Bubble Sort: [1, 2, 3, 4, 5], Time taken: 0.000007 seconds
Selection Sort: [1, 2, 3, 4, 5], Time taken: 0.000011 seconds
--------------------------------------------------
Original list: [5, 4, 3, 2, 1]
Bubble Sort: [1, 2, 3, 4, 5], Time taken: 0.000014 seconds
Selection Sort: [1, 2, 3, 4, 5], Time taken: 0.000011 seconds
--------------------------------------------------
PS C:\Users\lenovo\AppData\Local\Programs\Microsoft VS Code>
```