Lab 11 – Data Structures with AI: Implementing Fundamental Structures

Assignment number: 11.4
Enrollment number:2503A51L42
Name: YASHWANTH

Lab Objectives

- Use AI to assist in designing and implementing fundamental data structures in Python.
- Learn how to prompt AI for structure creation, optimization, and documentation.
- Improve understanding of Lists, Stacks, Queues, Linked Lists, Trees, Graphs, and Hash Tables.
- Enhance code quality with AI-generated comments and performance suggestions.

Task 1: Implementing a Stack (LIFO)

- **Task**: Use AI to help implement a **Stack** class in Python with the following operations: push(), pop(), peek(), and is_empty().
- **Instructions**:
    - Ask AI to generate code skeleton with docstrings.
    - Test stack operations using sample data.
    - Request AI to suggest optimizations or alternative implementations (e.g., using collections.deque).

Prompt:

- Implement a Stack class in Python with methods push(), pop(), peek(), and is_empty(). Add Google-style docstrings and comments explaining each part in simple language.
- Test stack operations using sample data.
- Suggest optimizations or alternative implementations (e.g., using collections.deque).

Code:

```
# class Stack: Implements a basic stack data structure.
class Stack:
    """A simple stack implementation using a Python list.

    Methods:
        push(item): Adds an item to the top of the stack.
        pop(): Removes and returns the top item of the stack.
        peek(): Returns the top item without removing it.
        is_empty(): Returns True if the stack is empty, False otherwise.
    """
    def __init__(self):
        """Initializes an empty stack."""
        self.items = []  # List to store stack items

    def push(self, item):
        """Adds an item to the top of the stack.

        Args:
            item: The item to be added.
        """
        self.items.append(item)  # Add item to the end of the list

    def pop(self):
        """Removes and returns the top item of the stack.

        Returns:
            The item at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
```

```
AI assistant coding/assignment 11.4/task1.py"
:\Users\Suhana Rehan\OneDrive\Desktop\AI assistant coding>
```

Output:

```
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI assistant coding> & "C:/Users/
Rehan/OneDrive/Desktop/AI assistant coding/assignment 11.4/task1.py"
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI assistant coding>
```

Code:

```
class Stack:
    def peek(self):
            The item at the top of the stack.
        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self.items[-1]  # Return the last item

    def is_empty(self):
        """Checks if the stack is empty.

        Returns:
            bool: True if the stack is empty, False otherwise.
        """
        return len(self.items) == 0  # True if no items in the list
# Sample usage and testing of Stack operations
if __name__ == "__main__":
    stack = Stack()
    print("Is stack empty?", stack.is_empty())  # Should be True
    stack.push(10)
    stack.push(20)
    stack.push(30)
    print("Stack after pushes:", stack.items)  # Should show [10, 20, 30]
    print("Peek top item:", stack.peek())      # Should show 30
    print("Pop top item:", stack.pop())        # Should remove and show 30
    print("Stack after pop:", stack.items)     # Should show [10, 20]
    print("Is stack empty?", stack.is_empty())  # Should be False
```

Output:

```
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI a
Rehan/OneDrive/Desktop/AI assistant coding/as
Pushing items: 1, 2, 3
Current stack: [1, 2, 3]
Top item (peek): 3
Popping top item: 3
Stack after pop: [1, 2]
Is stack empty? False
Is stack empty after popping all items? True
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI a
```

Code:

```python
from collections import deque
class Stack:
    """A simple stack implementation using a Python list.

    Methods:
        push(item): Add an item to the top of the stack.
        pop(): Remove and return the top item of the stack.
        peek(): Return the top item without removing it.
        is_empty(): Check if the stack is empty.
    """

    def __init__(self):
        """Initializes an empty stack."""
        self.items = deque()  # List to store stack items

    def push(self, item):
        """Adds an item to the top of the stack.

        Args:
            item: The item to add to the stack.
        """
        self.items.append(item)  # Add item to the end of the list

    def pop(self):
        """Removes and returns the top item of the stack.

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
```

Output:

```
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI
/Local/Programs/Python/Python312/python.exe"
nt coding/assignment 11.4/task1.py"
Pushing items: 1, 2, 3
Current stack: deque([1, 2, 3])
Top item (peek): 3
Popping top item: 3
Stack after pop: deque([1, 2])
Is stack empty? False
Is stack empty after popping all items? True
```

Observation:

I learned how a stack works with the LIFO method using Python's list.
I made sure to handle errors when popping or peeking from an empty stack.
AI helped me with the structure, but I test the sample data by my own

Task 2: Queue Implementation with Performance Review

- **Task:** Implement a **Queue** with enqueue(), dequeue(), and is_empty() methods.
- **Instructions**:
    - First, implement using Python lists.
    - Then, ask AI to review performance and suggest a more efficient implementation (using collections.deque).

Prompt: Review performance and suggest a more efficient implementation (using collections.deque).

Code without AI:

signment 11.4 > task2.py > ...

```python
class Queue:
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.pop(0)

    def is_empty(self):
        return len(self.items) == 0



if __name__ == "__main__":
    queue = Queue()
    print(queue.is_empty())
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.is_empty())
    print(queue.dequeue())
    print(queue.dequeue())
    print(queue.dequeue())
    try:
        queue.dequeue()
    except IndexError as e:
        print("Exception caught:", e)
```

Output without AI:

```
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI assistant coding>
rams/Python/Python312/python.exe" "c:/Users/Suhana Rehan/OneDr
1.4/task2.py"
True
False
1
2
3
Exception caught: dequeue from empty queue
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI assistant coding>
```

Code with AI:

```
from collections import deque

class Queue:
    """Efficient queue implementation using collections.deque."""
    def __init__(self):
        self.items = deque()

    def enqueue(self, item):
        """Add item to the end of the queue."""
        self.items.append(item)

    def dequeue(self):
        """Remove and return the item from the front of the queue."
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.popleft()

    def is_empty(self):
        """Check if the queue is empty."""
        return not self.items
if __name__ == "__main__":
    queue = Queue()
    print(queue.is_empty())
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    print(queue.is_empty())
    print(queue.dequeue())
    print(queue.dequeue())
    print(queue.dequeue())
    try:
        queue.dequeue()
    except IndexError as e:
```

Output with AI:

```
PS C:\Users\Suhana Rehan\OneDrive\Desktop\AI
n.exe" "c:/Users/Suhana Rehan/OneDrive/Deskto
True
False
1
2
3
Exception caught: dequeue from empty queue
```

Observation:

Here the manual took more time to execute and ai generate code took less time to execute

Ai generated code looks more put-together than manually written one

Task 3: Singly Linked List with Traversal

- **Task**: Implement a **Singly Linked List** with operations: insert_at_end(), delete_value(), and traverse().
- **Instructions**:

  - Start with a simple class-based implementation (Node, LinkedList).
  - Use AI to generate inline comments explaining pointer updates (which are non-trivial).
  - Ask AI to suggest test cases to validate all operations.

Prompt:

Generate inline comments explaining pointer updates and Suggest test cases to validate all operations

Code:

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node
    def delete_value(self, data):
        if self.head is None:
            return

        if self.head.data == data:
            self.head = self.head.next
            return

        current_node = self.head
        while current_node.next and current_node.next.data != data:
            current_node = current_node.next

        if current_node.next:
            current_node.next = current_node.next.next
    def traverse(self):
        current_node = self.head
        while current_node:
```

Output:

```
PS C:\Users\Suhana Rehan\OneDrive\De
/Programs/Python/Python312/python.ex
signment 11.4/task3.py"
Original list:
10 -> 20 -> 30 -> 40 -> None
List after deleting 20:
10 -> 30 -> 40 -> None
List after deleting 10:
30 -> 40 -> None
PS C:\Users\Suhana Rehan\OneDrive\De
```

Code after AI:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node  # If list is empty, new node becomes head
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next  # Move to next node until end
        last_node.next = new_node  # Update last node's next pointer to new node
    def delete_value(self, data):
        if self.head is None:
            return  # List is empty, nothing to delete

        if self.head.data == data:
            self.head = self.head.next  # Move head pointer to next node
            return

        current_node = self.head
        # Traverse until we find the node before the one to delete
        while current_node.next and current_node.next.data != data:
            current_node = current_node.next

        if current_node.next:
            # Update pointer to skip the node to be deleted
            current_node.next = current_node.next.next
    def traverse(self):
```

Output after AI:

```
Original list:
10 -> 20 -> 30 -> 40 -> None
List after deleting 20:
10 -> 30 -> 40 -> None
List after deleting 10:
30 -> 40 -> None
List after deleting 40:
30 -> None
List after trying to delete 99 (not in list):
30 -> None
List after deleting 30 (should be empty):
None
```

Observation:

AI generated clear comments explaining the logic of insertions and deletions and Suggested test cases to validate all operations

Task 4: Binary Search Tree (BST)

- **Task**: Implement a **Binary Search Tree** with methods for insert(), search(), and inorder_traversal().
- **Instructions**:
  - AI with a partially written Node and BST class
  - Ask AI to complete missing methods and add docstrings.
  - Test with a list of integers and compare outputs of search() for present vs absent elements.

Prompt:

Code:

Output:

Observation:

- A BST class with clean implementation, meaningful docstrings, and correct traversal output.

Task 5: Graph Representation and BFS/DFS Traversal

- **Task**: Implement a **Graph** using an adjacency list, with traversal methods BFS() and DFS().
- **Instructions**:
  - Start with an adjacency list dictionary.
  - Ask AI to generate BFS and DFS implementations with inline comments.
  - Compare recursive vs iterative DFS if suggested by AI.

Prompt:

Code:

Output:

Observation:

` A graph implementation with BFS and DFS traversal methods, with AI-generated comments explaining traversal steps.