LAB ASSIGNMENT-11

NAME : P.HANSINI REDDY

ROLL NO : 2403A510D5

BATCH : 01

COURSE : AI ASSISTED CODING

QUESTION

Learning Outcomes
After completing this lab, students will be able to:

1. Implement stack, queue, and linked list using Python with AI support.
2. Use AI tools to optimize and refactor basic data structure operations.
3. Compare multiple AI-suggested implementations for the same structure.
4. Apply AI assistance to generate test cases for verifying data structure behavior.
5. Demonstrate understanding of trade-offs in AI-generated solutions.

Task Description #1 – Stack class implementation
Task: Ask AI to implement a stack class with push(), pop(), peek() and is_empty() methods

Task Description #2 – Queue Implementation
Task: Use AI to generate a Queue class with enqueue(), dequeue(), and is_empty().

Task Description #3 – Linked List Implementation
Task: Ask AI to create a singly linked list with insert_at_end(), insert_at_beginning(), and display().

Task Description #4 – Binary Search Tree (BST)
Task: Ask AI to generate a simple BST with insert() and inorder_traversal().

TASK 1

PROMPT

Implement a Stack class in Python with push(), pop(), peek(), and is_empty() methods. Show an example.

```python
class Stack:
    def __init__(self):
        self._items = []

    def is_empty(self):
        return not self._items

    def push(self, item):
        self._items.append(item)

    def pop(self):
        if not self.is_empty():
            return self._items.pop()
        else:
            return "Stack is empty"

    def peek(self):
        if not self.is_empty():
            return self._items[-1]
        else:
            return "Stack is empty"
```

```python
# Example Usage
stack = Stack()

print(f"Is the stack empty? {stack.is_empty()}")

stack.push(10)
stack.push(20)
stack.push(30)

print(f"Is the stack empty after pushing elements? {stack.is_empty()}")
print(f"Top element (peek): {stack.peek()}")
print(f"Popped element: {stack.pop()}")
print(f"Top element after pop: {stack.peek()}")
print(f"Popped element: {stack.pop()}")
print(f"Popped element: {stack.pop()}")
print(f"Popped element from empty stack: {stack.pop()}")
print(f"Is the stack empty after popping all elements? {stack.is_empty()}")
```

```
Is the stack empty? True
Is the stack empty after pushing elements? False
Top element (peek): 30
Popped element: 30
Top element after pop: 20
Popped element: 20
Popped element: 10
Popped element from empty stack: Stack is empty
Is the stack empty after popping all elements? True
```

EXPLAINATION

1. Stack Implementation

store and manage data in a *Last-In-First-Out (LIFO)* order.
The program will let you add, remove, and view the top element using methods like push(), pop(), and peek().

TASK 2

PROMPT

Create a Queue class in Python with enqueue(), dequeue(), and is_empty() methods. Include example usage.

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

    def is_empty(self):
        return not self._items

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

    def dequeue(self):
        if not self.is_empty():
            return self._items.pop(0)  # Remove and return the first element
        else:
            return "Queue is empty"
```

```python
# Example Usage
queue = Queue()

print(f"Is the queue empty? {queue.is_empty()}")

queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

print(f"Is the queue empty after enqueuing elements? {queue.is_empty()}")
print(f"Dequeued element: {queue.dequeue()}")
print(f"Dequeued element: {queue.dequeue()}")
print(f"Dequeued element: {queue.dequeue()}")
print(f"Dequeued element from empty queue: {queue.dequeue()}")
print(f"Is the queue empty after dequeuing all elements? {queue.is_empty()}")
```

```
Is the queue empty? True
Is the queue empty after enqueuing elements? False
Dequeued element: 10
Dequeued element: 20
Dequeued element: 30
Dequeued element from empty queue: Queue is empty
Is the queue empty after dequeuing all elements? True
```

EXPLAINATION

Queue Implementation

First-In-First-Out (FIFO) data handling method.
The class will help you insert (enqueue) and remove (dequeue) elements in order, just like people waiting in a line.

TASK 3

PROMPT

Write a Python program for a singly linked list with insert_at_end(), insert_at_beginning(), and display() methods.

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        last = self.head
        while (last.next):
            last = last.next
        last.next = new_node

    def display(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        print(elements)
```

```
# Example Usage
sll = SinglyLinkedList()

sll.insert_at_beginning(10)
sll.insert_at_beginning(5)
sll.insert_at_end(20)
sll.insert_at_end(25)

print("Singly Linked List:")
sll.display()
```

```
Singly Linked List:
[5, 10, 20, 25]
```

EXPLAINATION

Linked List Implementation

data elements (nodes) are connected using links instead of indexes.
You can insert elements at the beginning or end and display the full list dynamically.

TASK 4

PROMPT

Implement a simple Binary Search Tree in Python with insert() and inorder_traversal() methods. Show example output.

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)

    def _insert(self, root, key):
        if key < root.val:
            if root.left is None:
                root.left = Node(key)
            else:
                self._insert(root.left, key)
        else:
            if root.right is None:
                root.right = Node(key)
            else:
                self._insert(root.right, key)

    def inorder_traversal(self):
        elements = []
        self._inorder_traversal(self.root, elements)
        return elements

    def _inorder_traversal(self, root, elements):
        if root:
            self._inorder_traversal(root.left, elements)
            elements.append(root.val)
            self._inorder_traversal(root.right, elements)
```

```python
                root.right = Node(key)
            else:
                self._insert(root.right, key)

    def inorder_traversal(self):
        elements = []
        self._inorder_traversal(self.root, elements)
        return elements

    def _inorder_traversal(self, root, elements):
        if root:
            self._inorder_traversal(root.left, elements)
            elements.append(root.val)
            self._inorder_traversal(root.right, elements)
```

```python
# Example Usage
bst = BinarySearchTree()
bst.insert(50)
bst.insert(30)
bst.insert(20)
bst.insert(40)
bst.insert(70)
bst.insert(60)
bst.insert(80)

print("Inorder traversal of the BST:")
print(bst.inorder_traversal())
```

```
Inorder traversal of the BST:
[20, 30, 40, 50, 60, 70, 80]
```

EXPLAINATION

. Binary Search Tree (BST) Implementation

organize data hierarchically to allow fast searching, inserting, and traversal. The code builds a tree and prints elements in sorted order using inorder_traversal().