# ASSIGNMENT: 11.3

## NAME : BHAWWANA SHRE

## ENROLLMENT NO : 2403A52311

## BATCH NO: 01

## SUBJECT: AI ASSISTANT CODING

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

*Prompt:*

*Implement a stack class with push(), pop(), peek() and is_empty() methods*

*SCREENSHOT:*

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

    def is_empty(self):
        return 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
            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()
# stack.push(1)
# stack.push(2)
# print(stack.peek())
# print(stack.pop())
# print(stack.is_empty())
```

```python
# Example usage:
stack = Stack()
print("Is the stack empty initially?", stack.is_empty())

stack.push(10)
print("After pushing 10, is the stack empty?", stack.is_empty())
print("Top element after pushing 10:", stack.peek())

stack.push(20)
print("After pushing 20, top element is:", stack.peek())

print("Popping element:", stack.pop())
print("Top element after popping:", stack.peek())

print("Popping element:", stack.pop())
print("Top element after popping:", stack.peek()) # Should be empty

print("Is the stack empty now?", stack.is_empty())

print("Trying to pop from an empty stack:", stack.pop())
print("Trying to peek at an empty stack:", stack.peek())
```

```
Is the stack empty initially? True
After pushing 10, is the stack empty? False
Top element after pushing 10: 10
```


```python
print("Trying to pop from an empty stack:", stack.pop())
print("Trying to peek at an empty stack:", stack.peek())
```


```
Is the stack empty initially? True
After pushing 10, is the stack empty? False
Top element after pushing 10: 10
After pushing 20, top element is: 20
Popping element: 20
Top element after popping: 10
Popping element: 10
Top element after popping: Stack is empty
Is the stack empty now? True
Trying to pop from an empty stack: Stack is empty
Trying to peek at an empty stack: Stack is empty
```

*EXPLAINATION:*

- *__init__(self): This is the constructor of the class. It initializes an empty list called self.items to store the elements of the stack.*
- *is_empty(self): This method checks if the stack is empty by comparing self.items to an empty list. It returns True if the stack is empty, and False otherwise.*
- *push(self, item): This method adds an item to the top of the stack by appending it to the self.items list.*
- *pop(self): This method removes and returns the item from the top of the stack. It first checks if the stack is empty using is_empty(). If not empty, it uses the pop() method of the list to remove and return the last element (which is the top of the stack). If the stack is empty, it returns the string "Stack is empty".*
- *peek(self): This method returns the item at the top of the stack without removing it. It also checks if the stack is empty. If not empty, it returns the last element of the self.items list using indexing [-1]. If the stack is empty, it returns "Stack is empty".*

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

*PROMPT:*

*generate a Queue class with enqueue(), dequeue(), and is_empty() with output*

*SCREENSHOTS:*

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

    def is_empty(self):
        return self.items == []

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

    def dequeue(self):
        if not self.is_empty():
            # Queues are FIFO, so we remove the first element
            return self.items.pop(0)
        else:
            return "Queue is empty"

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            return "Queue is empty"
```

```python
# Example usage with output:
queue = Queue()
print("Is the queue empty initially?", queue.is_empty())

queue.enqueue(10)
print("After enqueueing 10, is the queue empty?", queue.is_empty())
print("Front element after enqueueing 10:", queue.peek())

queue.enqueue(20)
print("After enqueueing 20, front element is:", queue.peek())

print("Dequeueing element:", queue.dequeue())
print("Front element after dequeueing:", queue.peek())

print("Dequeueing element:", queue.dequeue())
print("Front element after dequeueing:", queue.peek()) # Should be empty

print("Is the queue empty now?", queue.is_empty())

print("Trying to dequeue from an empty queue:", queue.dequeue())
print("Trying to peek at an empty queue:", queue.peek())
```

```
Is the queue empty initially? True
After enqueueing 10, is the queue empty? False
Front element after enqueueing 10: 10
After enqueueing 20, front element is: 10
Dequeueing element: 10
Front element after dequeueing: 20
Dequeueing element: 20
Front element after dequeueing: Queue is empty
Is the queue empty now? True
Trying to dequeue from an empty queue: Queue is empty
Trying to peek at an empty queue: Queue is empty
```

*EXPLAINATION:*

- *__init__(self): This is the constructor of the class. It initializes an empty list called self.items to store the elements of the queue.*
- *is_empty(self): This method checks if the queue is empty by comparing self.items to an empty list. It returns True if the queue is empty, and False otherwise.*
- *enqueue(self, item): This method adds an item to the end of the queue by appending it to the self.items list.*
- *dequeue(self): This method removes and returns the item from the front of the queue. It first checks if the queue is empty using is_empty(). If not empty, it uses the pop(0) method of the list to remove and return the first element (which is the front of the queue). If the queue is empty, it returns the string "Queue is empty".*
- *peek(self): This method returns the item at the front of the queue without removing it. It also checks if the queue is empty. If not empty, it returns the first element of the self.items list using indexing [0]. If the queue is empty, it returns "Queue 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()*

*PROMPT:*

*create a singly linked list with insert_at_end(), insert_at_beginning(), and display() with output*

*SCREENSHOTS:*

```python
class Node:
    def __ini (variable) data: Any
        self.data = data
        self.next = None

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

    def display(self):
        current = self.head
```

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

    def display(self):
        current = self.head
        if current is None:
            print("Linked list is empty.")
            return
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Demonstrate usage:
linked_list = LinkedList()

print("Inserting at the beginning:")
linked_list.insert_at_beginning(10)
linked_list.display()

print("\nInserting at the end:")
linked_list.insert_at_end(20)
linked_list.display()
```

```
print("\nInserting another at the beginning:")
linked_list.insert_at_beginning(5)
linked_list.display()

print("\nInserting another at the end:")
linked_list.insert_at_end(30)
linked_list.display()

print("\nFinal linked list:")
linked_list.display()
```

```
Inserting at the beginning:
10 -> None

Inserting at the end:
10 -> 20 -> None

Inserting another at the beginning:
5 -> 10 -> 20 -> None

Inserting another at the end:
5 -> 10 -> 20 -> 30 -> None

Final linked list:
5 -> 10 -> 20 -> 30 -> None
```

*EXPLAINATION:*

- *Node Class:*
    - *This class represents an individual element (a node) within the linked list.*
    - *__init__(self, data): The constructor for a Node. It takes data as an argument and stores it in self.data. It also initializes self.next to None, indicating that this node doesn't point to another node yet.*
- *LinkedList Class:*
    - *This class represents the entire linked list.*
    - *__init__(self): The constructor for the LinkedList. It initializes self.head to None. The head is a reference to the first node in the list. An empty list has no head, hence None.*
    - *insert_at_end(self, data): This method adds a new node with the given data to the end of the linked list.*
        - *It creates a new_node instance.*
        - *If the list is empty (self.head is None), the new_node becomes the head.*

        - *If the list is not empty, it traverses the list starting from the head until it finds the last node (the one whose next is None).*
        - *It then sets the next of the last node to the new_node.*
    - *insert_at_beginning(self, data): This method adds a new node with the given data to the beginning of the linked list.*
        - *It creates a new_node instance.*
        - *It sets the next of the new_node to the current self.head.*
        - *It then updates self.head to point to the new_node, making it the new first node.*
    - *display(self): This method prints the data in each node of the linked list from the head to the end.*
        - *It starts from the self.head.*
        - *If the list is empty, it prints "Linked list is empty."*
        - *If not empty, it iterates through the nodes, printing the data of each node followed by " -> ".*
        - *Finally, it prints "None" to indicate the end of the list.*

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

*PROMPT:*

*generate a simple BST with insert() and inorder_traversal() with output*

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

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

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

    def _insert_recursive(self, current_node, key):
        if key < current_node.key:
            if current_node.left is None:
                current_node.left = Node(key)
            else:
                self._insert_recursive(current_node.left, key)
        elif key > current_node.key:
            if current_node.right is None:
                current_node.right = Node(key)
```

```python
        # Duplicate keys are ignored in this simple implementation

    def inorder_traversal(self):
        if self.root is not None:
            self._inorder_recursive(self.root)

    def _inorder_recursive(self, current_node):
        if current_node is not None:
            self._inorder_recursive(current_node.left)
            print(current_node.key, end=" ")
            self._inorder_recursive(current_node.right)
```

```python
# Demonstrate usage:
bst = BST()
elements_to_insert = [50, 30, 70, 20, 40, 60, 80]

print("Inserting elements:", elements_to_insert)
for element in elements_to_insert:
    bst.insert(element)

print("\nIn-order traversal (should be sorted):")
bst.inorder_traversal()
print() # Add a newline for cleaner output
```

```
Inserting elements: [50, 30, 70, 20, 40, 60, 80]
```


```
Inserting elements: [50, 30, 70, 20, 40, 60, 80]
```


```
output actions
In-order traversal (should be sorted):
20 30 40 50 60 70 80
```

*Explaination:*

- ***Node Class:***
  - *__init__(self, key): Represents a single node in the BST. It stores a key (the value of the node) and has pointers left and right initialized to None for its child nodes.*
- ***BST Class:***
  - *__init__(self): The constructor for the BST. It initializes the root of the tree to None, indicating an empty tree.*
  - *insert(self, key): This is the public method to insert a new key into the BST.*
    - *If the tree is empty (self.root is None), the new key becomes the root.*

    - *Otherwise, it calls the private helper method _insert_recursive to find the correct position for the new key.*
  - *_insert_recursive(self, current_node, key): This is a recursive helper method for inserting a key.*
    - *It compares the key to the current_node.key.*
    - *If the key is less than the current_node.key, it goes to the left child. If the left child is None, it creates a new Node there. Otherwise, it recursively calls _insert_recursive on the left child.*
    - *If the key is greater than the current_node.key, it goes to the right child following similar logic.*
    - *Duplicate keys are ignored in this implementation.*
  - *inorder_traversal(self): This is the public method to perform an in-order traversal of the BST.*
    - *If the tree is not empty, it calls the private helper method _inorder_recursive.*
  - *_inorder_recursive(self, current_node): This is a recursive helper method for the in-order traversal.*
    - *It follows the pattern: visit left subtree, visit the current node, visit right subtree.*
    - *It recursively calls itself on the left child, then prints the current_node.key, and finally recursively calls itself on the right child. This traversal method visits nodes in ascending order of their keys in a BST.*

- *It creates a BST object.*
- *It defines a list of elements_to_insert.*
- *It then iterates through this list and calls the insert() method for each element to build the BST.*

- *Finally, it calls the inorder_traversal() method, which prints the elements of the tree in sorted order, demonstrating that the insert() method maintained the BST property.*