

## **AI ASSISTED CODING**

**NAME: NAREDDY JASMITHA**

**ENROLLMENT NO: 2403A51281**

**BRANCH :CSE/AIML**

**COURSE : AI CODING**

**DATE: 08-10-2025**

**Task Description #1 – Stack class  
implementation**

**Task: Ask AI to implement a stack class with  
push(), pop(), peek() and  
is\_empty() methods**

**PROMPT:**

**Generate a code to develop an AI to  
implement a stack class with push(), pop(),  
peek() and is\_empty() methods**

**EXPECTED OUTPUT:**

```

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

    def is_empty(self):
        """Checks if the stack is empty."""
        return not self.items

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

    def pop(self):
        """Removes and returns the item from the top of the stack."""
        if not self.is_empty():
            return self.items.pop()
        else:
            return "Stack is empty"

    def peek(self):
        """Returns the item at the top of the stack without removing it."""
        if not self.is_empty():
            return self.items[-1]
        else:
            return "Stack is empty"

```

```

        return self.items[-1]
    else:
        return "Stack is empty"

# Example usage:
stack = Stack()
print(f"Is stack empty? {stack.is_empty()}")
stack.push(10)
stack.push(20)
stack.push(30)
print(f"Stack after pushes: {stack.items}")
print(f"Top element: {stack.peek()}")
print(f"Popped element: {stack.pop()}")
print(f"Stack after pop: {stack.items}")
print(f"Is stack empty? {stack.is_empty()}")
print(f"Popped element: {stack.pop()}")
print(f"Popped element: {stack.pop()}")
print(f"Popped element: {stack.pop()}") # Trying to pop from an empty stack

```

```

Is stack empty? True
Stack after pushes: [10, 20, 30]
Top element: 30
Popped element: 30
Stack after pop: [10, 20]
Is stack empty? False
Popped element: 20
Popped element: 10
Popped element: Stack is empty

```

## EXPLANATION:

- `__init__(self)`: This is the constructor of the class. It initializes an empty list called `self.items` which will be used to store the elements of the stack.
- `is_empty(self)`: This method checks if the stack is empty by checking if the `self.items` list is empty. 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. It does this by appending the `item` to the end of 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. 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. Similar to `pop()`, it 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:**

**Develop a code to generate Task: Use AI to generate a Queue class with enqueue(), dequeue(), and is\_empty()**

## EXPECTED OUTPUT:

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

    def is_empty(self):
        """Checks if the queue is empty."""
        return not self.items

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

    def dequeue(self):
        """Removes and returns the item from the front of the queue."""
        if not self.is_empty():
            return self.items.pop(0) # Remove from the front (index 0)
        else:
            return "Queue is empty"

    def peek(self):
        """Returns the item at the front of the queue without removing it."""
        if not self.is_empty():
            return self.items[0] # Return the item at the front (index 0)
        else:
            return "Queue is empty"
```

```

        return self.items[0] # return the item at the front (index 0)
    else:
        return "Queue is empty"

# Example usage:
queue = Queue()
print(f"Is queue empty? {queue.is_empty()}")
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)
print(f"Queue after enqueues: {queue.items}")
print(f"Front element: {queue.peek()}")
print(f"Dequeued element: {queue.dequeue()}")
print(f"Queue after dequeue: {queue.items}")
print(f"Is queue empty? {queue.is_empty()}")
print(f"Dequeued element: {queue.dequeue()}")
print(f"Dequeued element: {queue.dequeue()}")
print(f"Dequeued element: {queue.dequeue()}") # Trying to dequeue from an empty queue

```

```

> Is queue empty? True
Queue after enqueues: [10, 20, 30]
Front element: 10
Dequeued element: 10
Queue after dequeue: [20, 30]
Is queue empty? False
Dequeued element: 20
Dequeued element: 30
Dequeued element: Queue is empty

```

## EXPLANATION:

- **`__init__(self)`**: This is the constructor of the class. It initializes an empty list called `self.items`, which will be used to store the elements of the queue.
- **`is_empty(self)`**: This method checks if the queue is empty by checking if the `self.items` list is empty. 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. It uses the `append()` method to add the `item` to the end of 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. If not empty, it uses the `pop(0)` method of the list to remove and return the element at index 0 (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. Similar to `dequeue()`, it checks if the queue is empty. If not empty, it returns the element at index 0 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:

**Generate to develop a code Task: Ask AI to create a singly linked list with `insert_at_end()`, `insert_at_beginning()`, and `display()`**

### EXPECTED OUTPUT:

```
class Node:
    """Represents a node in a singly linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    """Represents a singly linked list."""
    def __init__(self):
        self.head = None

    def insert_at_beginning(self, data):
        """Inserts a new node at the beginning of the linked list."""
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    def insert_at_end(self, data):
        """Inserts a new node at the end of the linked list."""
        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 display(self):
        """Display the elements of the linked list """
```


```
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

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

# Example usage:
sll = SinglyLinkedList()
sll.insert_at_end(10)
sll.insert_at_beginning(5)
sll.insert_at_end(20)
sll.insert_at_beginning(2)

print("Linked list after insertions:")
sll.display()
```

---

 Linked list after insertions:  
2 -> 5 -> 10 -> 20 -> None

---

## EXPLANATION:

- **Node Class:**
  - `__init__(self, data)`: This is the constructor for a `Node`. Each node stores a piece of `data` and a reference to the `next` node in the list. Initially, `next` is set to `None`.
- **SinglyLinkedList Class:**
  - `__init__(self)`: The constructor for the linked list. It initializes the `head` of the list to `None`, indicating an empty list.
  - `insert_at_beginning(self, data)`: This method adds a new node with the given `data` at the beginning of the list. It creates a new `Node`, sets its `next` pointer to the current `head`, and then updates the `head` to point to the new node.
  - `insert_at_end(self, data)`: This method adds a new node with the given `data` at the end of the list. If the list is empty, the new node becomes the `head`. Otherwise, it traverses the list to find the last node and sets its `next` pointer to the new node.

## **Task Description #4 – Binary Search Tree (BST)**

**Task: Ask AI to generate a simple BST with `insert()` and `inorder_traversal()`.**

**PROMPT:**



## Generate a code to develop a Task: Ask AI to generate a simple BST with insert() and inorder\_traversal().

### EXPECTED OUTPUT:

```
class Node:
    """Represents a node in a Binary Search Tree."""
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    """Represents a Binary Search Tree."""
    def __init__(self):
        self.root = None

    def insert(self, key):
        """Inserts a new node with the given key into the BST."""
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, root, key):
        """Helper function for recursive insertion."""
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self._insert_recursive(root.left, key)
        elif key > root.key:
            root.right = self._insert_recursive(root.right, key)
        return root

    def inorder_traversal(self):
        """Performs an inorder traversal of the BST and prints the keys."""
```

```
def inorder_traversal(self):
    """Performs an inorder traversal of the BST and prints the keys."""
    self._inorder_recursive(self.root)
    print() # Print a newline after traversal

def _inorder_recursive(self, root):
    """Helper function for recursive inorder traversal."""
    if root:
        self._inorder_recursive(root.left)
        print(root.key, end=" ")
        self._inorder_recursive(root.right)

# Example usage:
bst = BST()
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:")
bst.inorder_traversal()
```

➞ Inorder traversal of the BST:  
20 30 40 50 60 70 80

## EXPLANATION:

- **Node Class:**

- `__init__(self, key)`: This is the constructor for a `Node`. Each node stores a `key` and references to its `left` and `right` children, initially set to `None`.

- **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 method inserts a new node with the given `key` into the BST. It calls the recursive helper function `_insert_recursive` to handle the actual insertion logic.
- `_insert_recursive(self, root, key)`: This is a private helper function that recursively inserts a new node.
  - If the current `root` is `None`, a new `Node` is created and returned.
  - If the `key` is less than the current `root`'s key, the insertion is done in the left subtree.
  - If the `key` is greater than the current `root`'s key, the

- If the current `root` is `None`, a new `Node` is created and returned.
  - If the `key` is less than the current `root`'s key, the insertion is done in the left subtree.
  - If the `key` is greater than the current `root`'s key, the insertion is done in the right subtree.
  - If the `key` is equal to the current `root`'s key, no insertion is performed (assuming no duplicate keys).
- `inorder_traversal(self)`: This method performs an inorder traversal of the BST. In an inorder traversal, the left subtree is visited first, then the current node, and finally the right subtree. This results in printing the keys in ascending order in a BST. It calls the recursive helper function `_inorder_recursive`.
  - `_inorder_recursive(self, root)`: This is a private helper function that recursively performs the inorder traversal. It checks if the current `root` is not `None`, then recursively traverses the left subtree, prints the current node's key, and finally recursively traverses the right subtree.