AI ASSISTED CODING

Nasani Sai Pavan **2303A51117**

BATCH – 03 **24 – 02 – 2026**

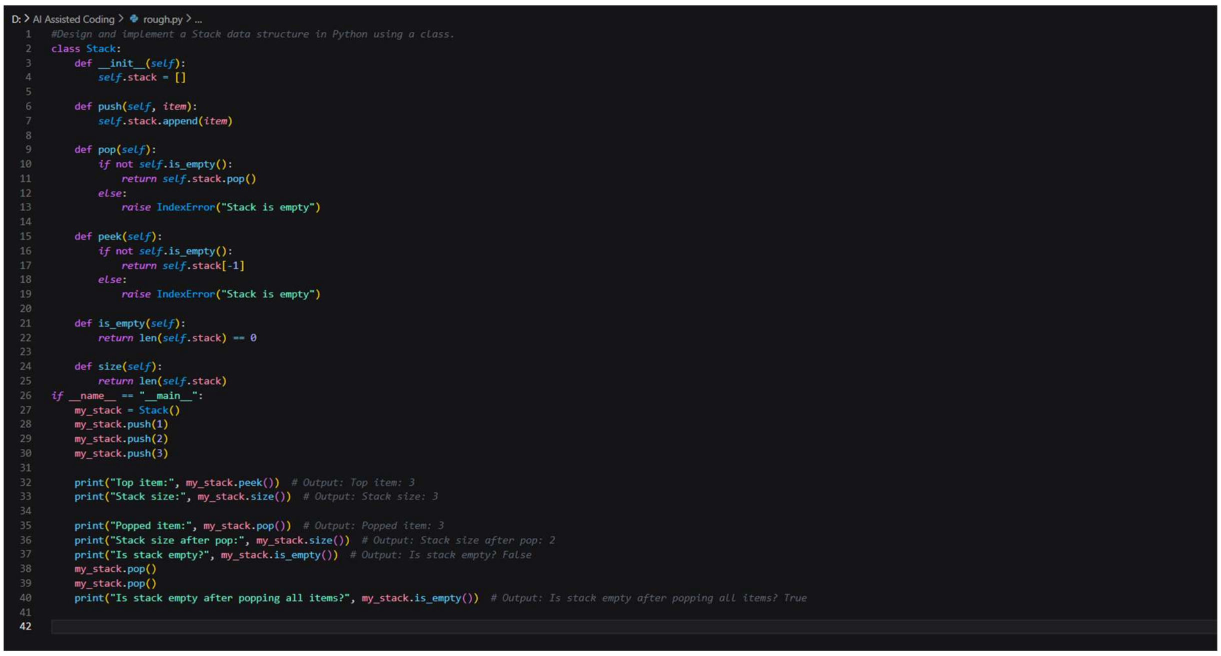
ASSIGNMENT – 11.2

LAB – 11.2 : Data Structures with AI: Implementing Fundamental Structures

Task – 01: (Stack Using AI Guidance)

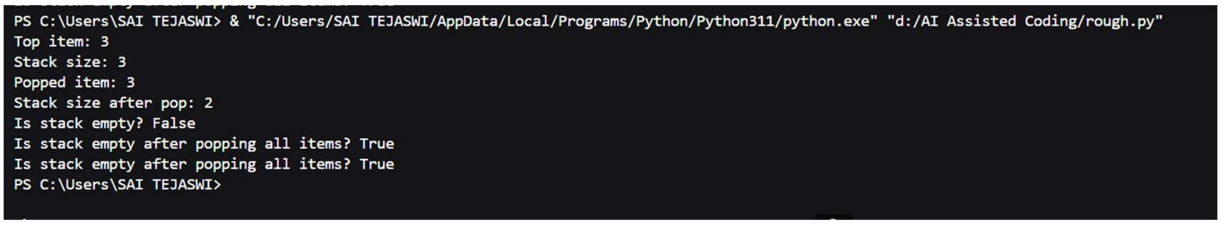
Prompt: Design and implement a Stack data structure in Python using a class.

Code:


D: > AI Assisted Coding > rough.py > ...

```python
#Design and implement a Stack data structure in Python using a class.
class Stack:
    def __init__(self):
        self.stack = []

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

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

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

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

    def size(self):
        return len(self.stack)
if __name__ == "__main__":
    my_stack = Stack()
    my_stack.push(1)
    my_stack.push(2)
    my_stack.push(3)

    print("Top item:", my_stack.peek())  # Output: Top item: 3
    print("Stack size:", my_stack.size())  # Output: Stack size: 3

    print("Popped item:", my_stack.pop())  # Output: Popped item: 3
    print("Stack size after pop:", my_stack.size())  # Output: Stack size after pop: 2
    print("Is stack empty?", my_stack.is_empty())  # Output: Is stack empty? False
    my_stack.pop()
    my_stack.pop()
    print("Is stack empty after popping all items?", my_stack.is_empty())  # Output: Is stack empty after popping all items? True

```

Output:



```
PS C:\Users\SAI TEJASWI> & "C:/Users/SAI TEJASWI/AppData/Local/Programs/Python/Python311/python.exe" "d:/AI Assisted Coding/rough.py"
Top item: 3
Stack size: 3
Popped item: 3
Stack size after pop: 2
Is stack empty? False
Is stack empty after popping all items? True
Is stack empty after popping all items? True
PS C:\Users\SAI TEJASWI>
```

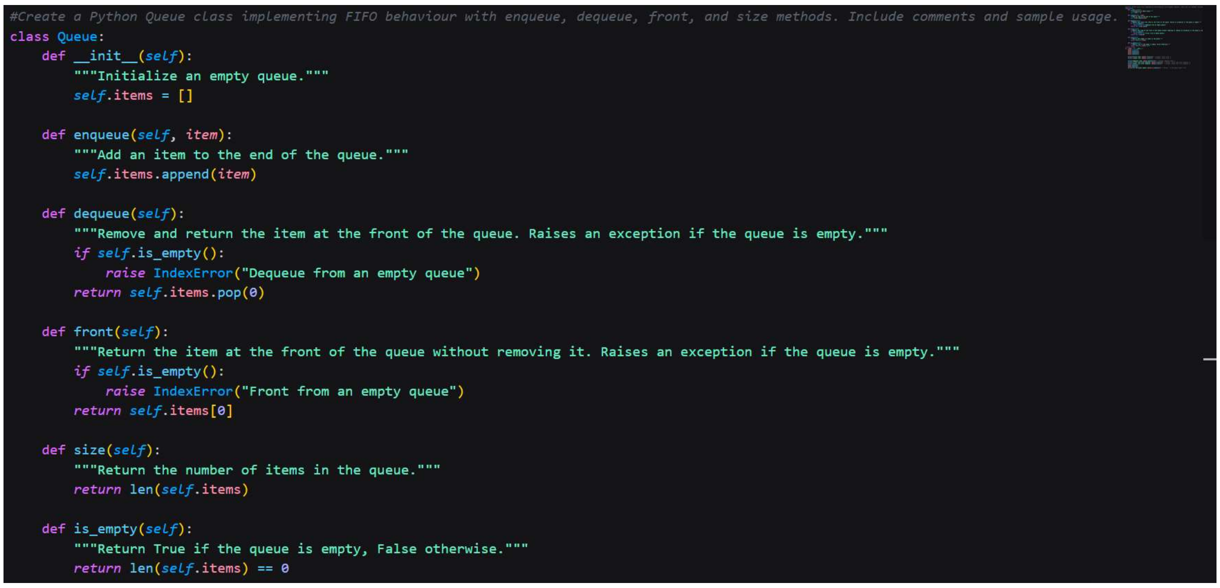
Explanation :

The Stack follows the LIFO (Last In First Out) principle. Elements are added using push() and removed using pop(). The peek() method returns the top element without removing it, and is_empty() checks whether the stack contains elements.

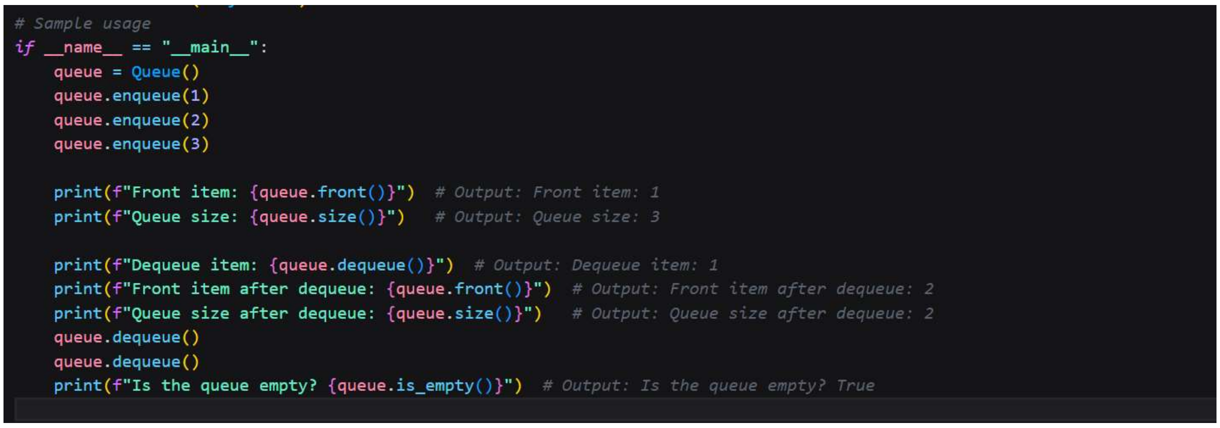
Task – 02 : Queue Design.

Prompt: Create a Python Queue class implementing FIFO behaviour with enqueue, dequeue, front, and size methods. Include comments and sample usage.

CODE:

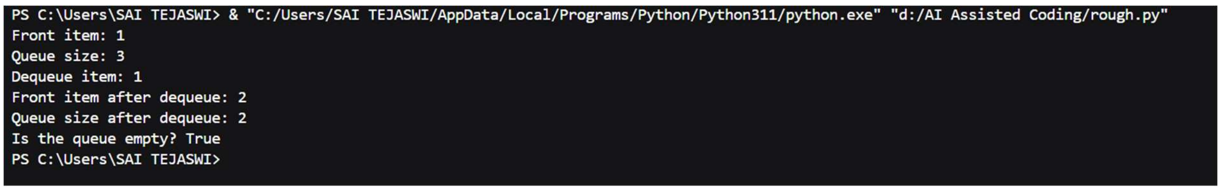


```python
#Create a Python Queue class implementing FIFO behaviour with enqueue, dequeue, front, and size methods. Include comments and sample usage.
class Queue:
    def __init__(self):
        """Initialize an empty queue."""
        self.items = []

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

    def dequeue(self):
        """Remove and return the item at the front of the queue. Raises an exception if the queue is empty."""
        if self.is_empty():
            raise IndexError("Dequeue from an empty queue")
        return self.items.pop(0)

    def front(self):
        """Return the item at the front of the queue without removing it. Raises an exception if the queue is empty."""
        if self.is_empty():
            raise IndexError("Front from an empty queue")
        return self.items[0]

    def size(self):
        """Return the number of items in the queue."""
        return len(self.items)

    def is_empty(self):
        """Return True if the queue is empty, False otherwise."""
        return len(self.items) == 0
```



```python
# Sample usage
if __name__ == "__main__":
    queue = Queue()
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)

    print(f"Front item: {queue.front()}")  # Output: Front item: 1
    print(f"Queue size: {queue.size()}")   # Output: Queue size: 3

    print(f"Dequeue item: {queue.dequeue()}")  # Output: Dequeue item: 1
    print(f"Front item after dequeue: {queue.front()}")  # Output: Front item after dequeue: 2
    print(f"Queue size after dequeue: {queue.size()}")   # Output: Queue size after dequeue: 2
    queue.dequeue()
    queue.dequeue()
    print(f"Is the queue empty? {queue.is_empty()}")  # Output: Is the queue empty? True
```

OUTPUT:



```
PS C:\Users\SAI TEJASWI> & "C:/Users/SAI TEJASWI/AppData/Local/Programs/Python/Python311/python.exe" "d:/AI Assisted Coding/rough.py"
Front item: 1
Queue size: 3
Dequeue item: 1
Front item after dequeue: 2
Queue size after dequeue: 2
Is the queue empty? True
PS C:\Users\SAI TEJASWI>
```

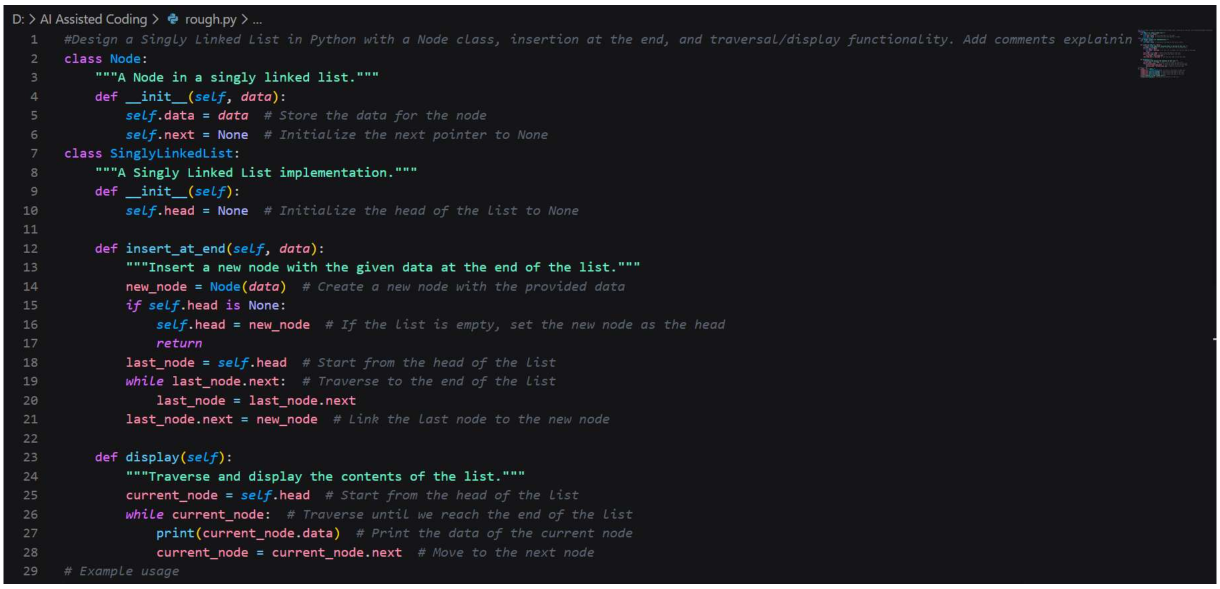
Explanation :

The Queue follows the FIFO (First In First Out) principle. Elements are inserted using enqueue() and removed using dequeue(). The front() method returns the first element, and size() gives the total number of elements.

Task – 03 : Singly Linked List Construction.

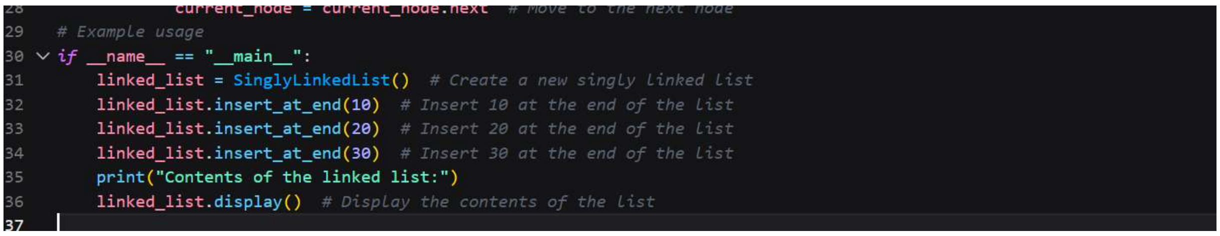
Prompt : Design a Singly Linked List in Python with a Node class, insertion at the end, and traversal/display functionality. Add comments explaining each part.

Code:



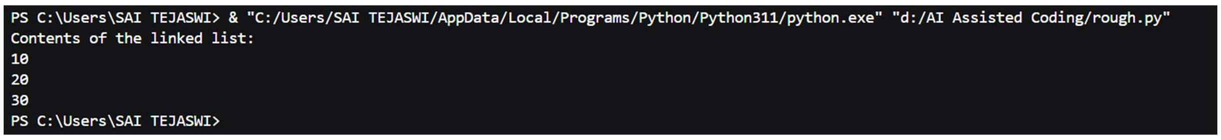
D: > AI Assisted Coding > rough.py > ...

```python
#Design a Singly Linked List in Python with a Node class, insertion at the end, and traversal/display functionality. Add comments explainin
class Node:
    """A Node in a singly linked list."""
    def __init__(self, data):
        self.data = data  # Store the data for the node
        self.next = None  # Initialize the next pointer to None
class SinglyLinkedList:
    """A Singly Linked List implementation."""
    def __init__(self):
        self.head = None  # Initialize the head of the list to None

    def insert_at_end(self, data):
        """Insert a new node with the given data at the end of the list."""
        new_node = Node(data)  # Create a new node with the provided data
        if self.head is None:
            self.head = new_node  # If the list is empty, set the new node as the head
            return
        last_node = self.head  # Start from the head of the list
        while last_node.next:  # Traverse to the end of the list
            last_node = last_node.next
        last_node.next = new_node  # Link the last node to the new node

    def display(self):
        """Traverse and display the contents of the list."""
        current_node = self.head  # Start from the head of the list
        while current_node:  # Traverse until we reach the end of the list
            print(current_node.data)  # Print the data of the current node
            current_node = current_node.next  # Move to the next node
# Example usage
```



```python
        current_node = current_node.next  # Move to the next node
# Example usage
if __name__ == "__main__":
    linked_list = SinglyLinkedList()  # Create a new singly linked list
    linked_list.insert_at_end(10)  # Insert 10 at the end of the list
    linked_list.insert_at_end(20)  # Insert 20 at the end of the list
    linked_list.insert_at_end(30)  # Insert 30 at the end of the list
    print("Contents of the linked list:")
    linked_list.display()  # Display the contents of the list
```

Output :



```
PS C:\Users\SAI TEJASWI> & "C:/Users/SAI TEJASWI/AppData/Local/Programs/Python/Python311/python.exe" "d:/AI Assisted Coding/rough.py"
Contents of the linked list:
10
20
30
PS C:\Users\SAI TEJASWI>
```

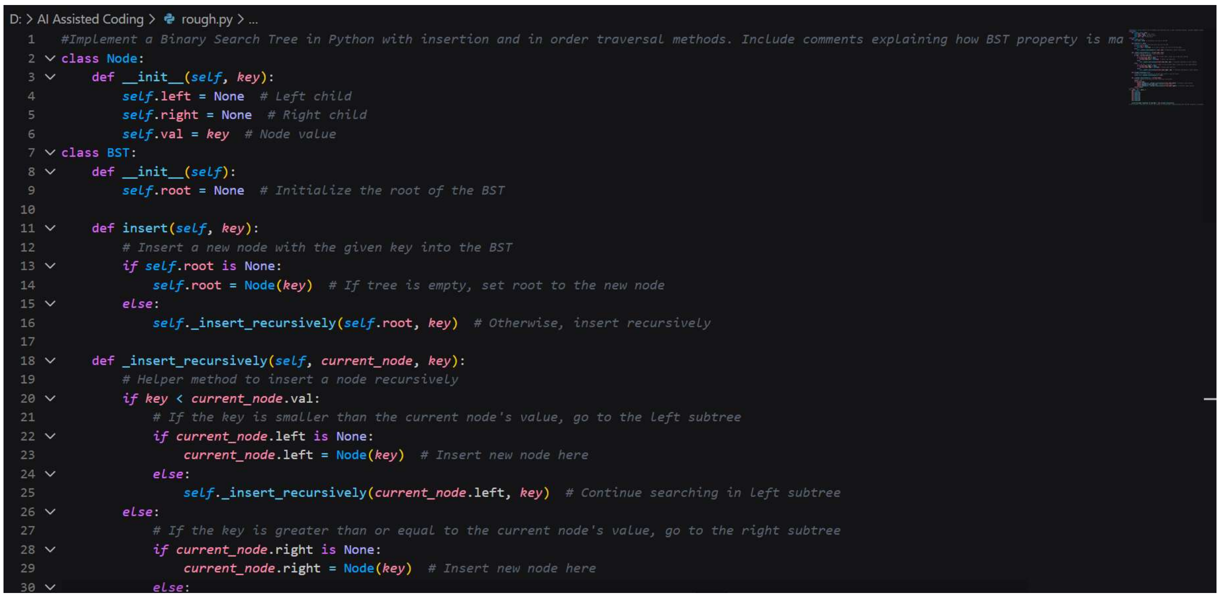
Explanation :

A Singly Linked List consists of nodes where each node stores data and a reference to the next node. Insertion adds a new node at the end of the list. Traversal iterates through nodes sequentially to display all elements.

Task – 04 : Binary Search Tree Operations.

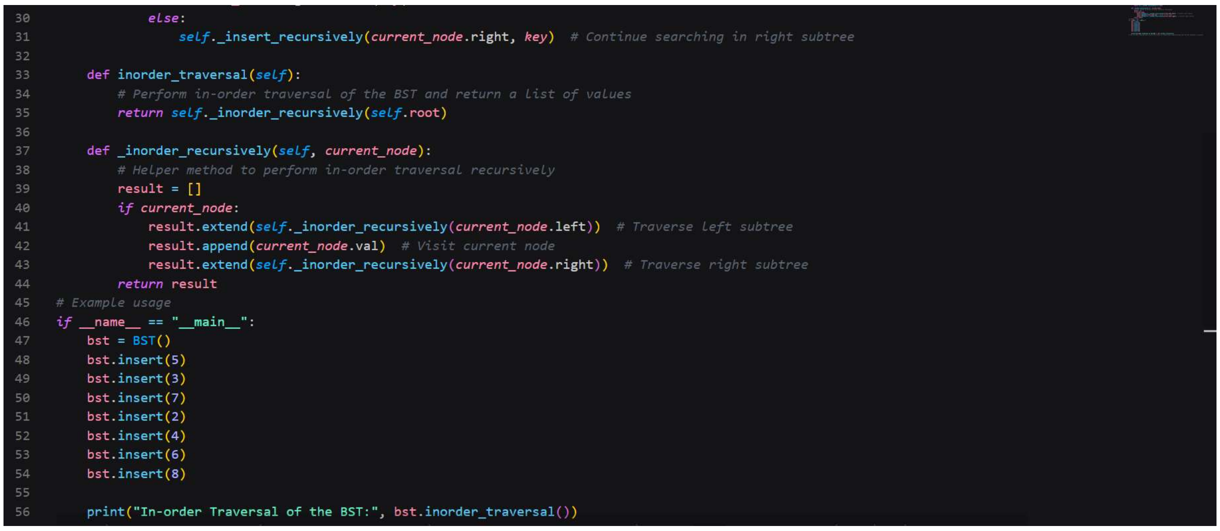
Prompt : Implement a Binary Search Tree in Python with insertion and in order traversal methods. Include comments explaining how BST property is maintained.

Code:



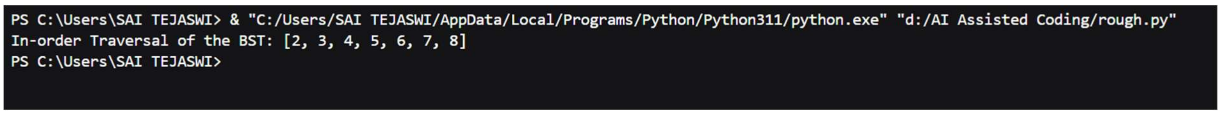
D: > AI Assisted Coding > rough.py > ...

```python
#Implement a Binary Search Tree in Python with insertion and in order traversal methods. Include comments explaining how BST property is ma
class Node:
    def __init__(self, key):
        self.left = None  # Left child
        self.right = None  # Right child
        self.val = key  # Node value
class BST:
    def __init__(self):
        self.root = None  # Initialize the root of the BST

    def insert(self, key):
        # Insert a new node with the given key into the BST
        if self.root is None:
            self.root = Node(key)  # If tree is empty, set root to the new node
        else:
            self._insert_recursively(self.root, key)  # Otherwise, insert recursively

    def _insert_recursively(self, current_node, key):
        # Helper method to insert a node recursively
        if key < current_node.val:
            # If the key is smaller than the current node's value, go to the left subtree
            if current_node.left is None:
                current_node.left = Node(key)  # Insert new node here
            else:
                self._insert_recursively(current_node.left, key)  # Continue searching in left subtree
        else:
            # If the key is greater than or equal to the current node's value, go to the right subtree
            if current_node.right is None:
                current_node.right = Node(key)  # Insert new node here
```



```python
            else:
                self._insert_recursively(current_node.right, key)  # Continue searching in right subtree

    def inorder_traversal(self):
        # Perform in-order traversal of the BST and return a list of values
        return self._inorder_recursively(self.root)

    def _inorder_recursively(self, current_node):
        # Helper method to perform in-order traversal recursively
        result = []
        if current_node:
            result.extend(self._inorder_recursively(current_node.left))  # Traverse left subtree
            result.append(current_node.val)  # Visit current node
            result.extend(self._inorder_recursively(current_node.right))  # Traverse right subtree
        return result
# Example usage
if __name__ == "__main__":
    bst = BST()
    bst.insert(5)
    bst.insert(3)
    bst.insert(7)
    bst.insert(2)
    bst.insert(4)
    bst.insert(6)
    bst.insert(8)

    print("In-order Traversal of the BST:", bst.inorder_traversal())
```

Output :



```
PS C:\Users\SAI TEJASWI> & "C:/Users/SAI TEJASWI/AppData/Local/Programs/Python/Python311/python.exe" "d:/AI Assisted Coding/rough.py"
In-order Traversal of the BST: [2, 3, 4, 5, 6, 7, 8]
PS C:\Users\SAI TEJASWI>
```

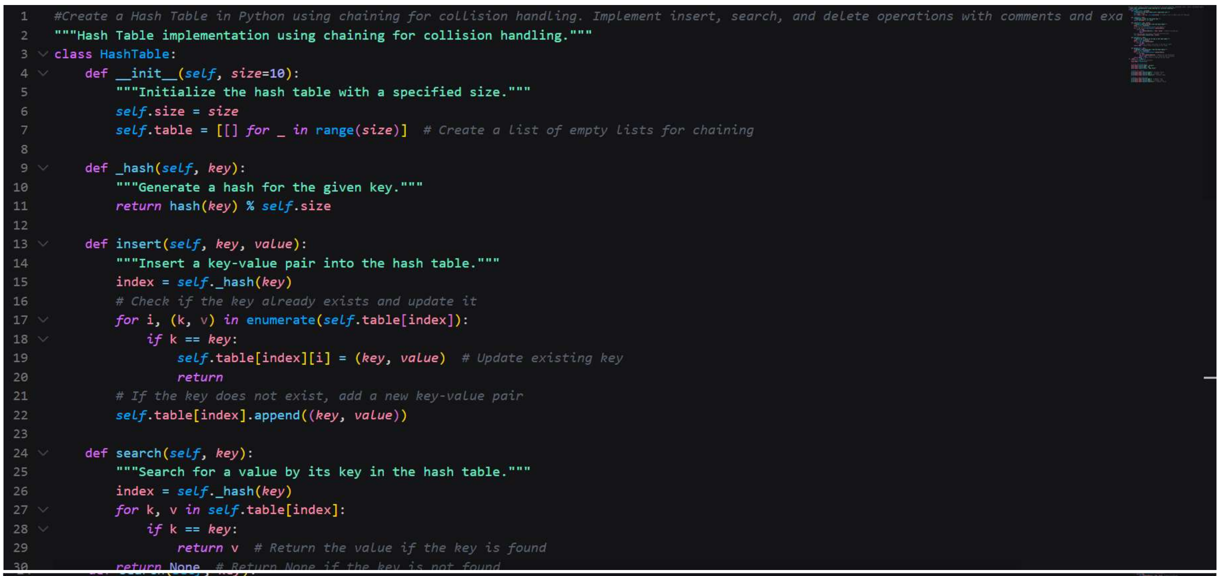
Explanation :

A Binary Search Tree maintains the property: Left child < Root < Right child. Insertion places elements according to this rule, and in-order traversal prints elements in sorted order.

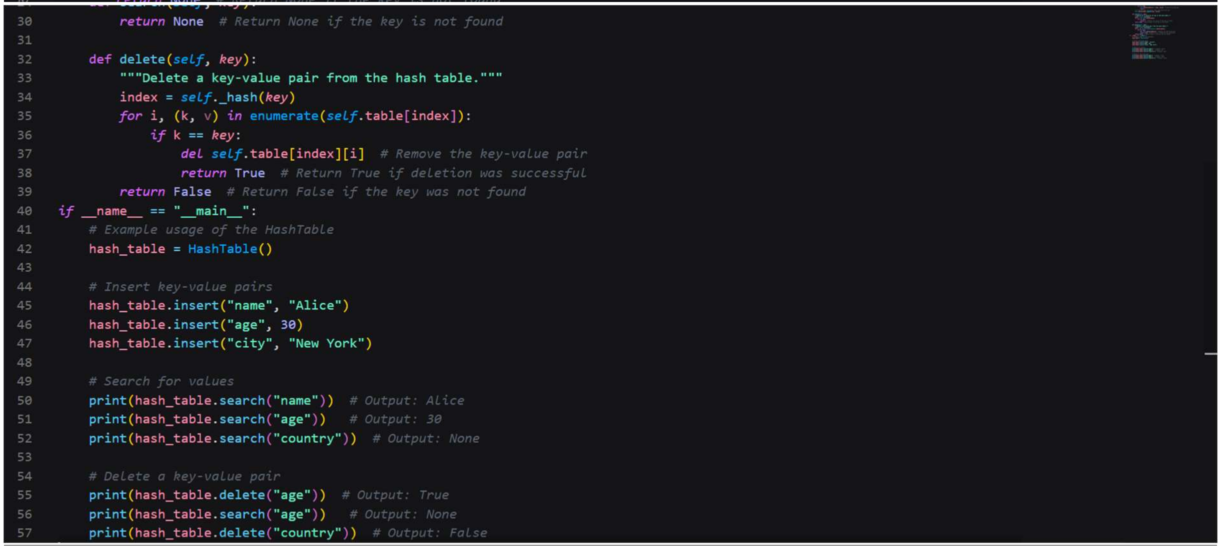
Task – 05 : Hash Table Implementation.

Prompt : Create a Hash Table in Python using chaining for collision handling. Implement insert, search, and delete operations with comments and example usage.

CODE:

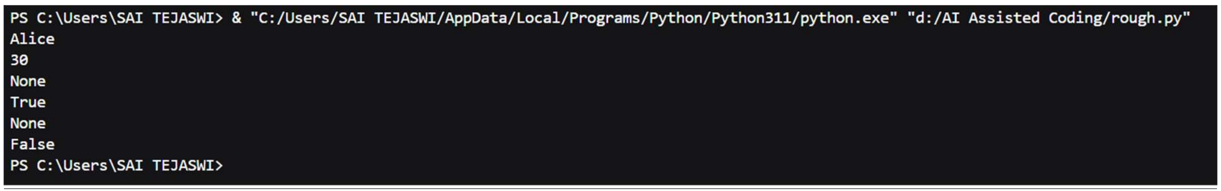


```python
#Create a Hash Table in Python using chaining for collision handling. Implement insert, search, and delete operations with comments and exa
"""Hash Table implementation using chaining for collision handling."""
class HashTable:
    def __init__(self, size=10):
        """Initialize the hash table with a specified size."""
        self.size = size
        self.table = [[] for _ in range(size)]  # Create a list of empty lists for chaining

    def _hash(self, key):
        """Generate a hash for the given key."""
        return hash(key) % self.size

    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        index = self._hash(key)
        # Check if the key already exists and update it
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        # If the key does not exist, add a new key-value pair
        self.table[index].append((key, value))

    def search(self, key):
        """Search for a value by its key in the hash table."""
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v  # Return the value if the key is found
```



```python
        return None  # Return None if the key is not found

    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]  # Remove the key-value pair
                return True  # Return True if deletion was successful
        return False  # Return False if the key was not found
if __name__ == "__main__":
    # Example usage of the HashTable
    hash_table = HashTable()

    # Insert key-value pairs
    hash_table.insert("name", "Alice")
    hash_table.insert("age", 30)
    hash_table.insert("city", "New York")

    # Search for values
    print(hash_table.search("name"))  # Output: Alice
    print(hash_table.search("age"))   # Output: 30
    print(hash_table.search("country"))  # Output: None

    # Delete a key-value pair
    print(hash_table.delete("age"))  # Output: True
    print(hash_table.search("age"))   # Output: None
    print(hash_table.delete("country"))  # Output: False
```

OUTPUT:



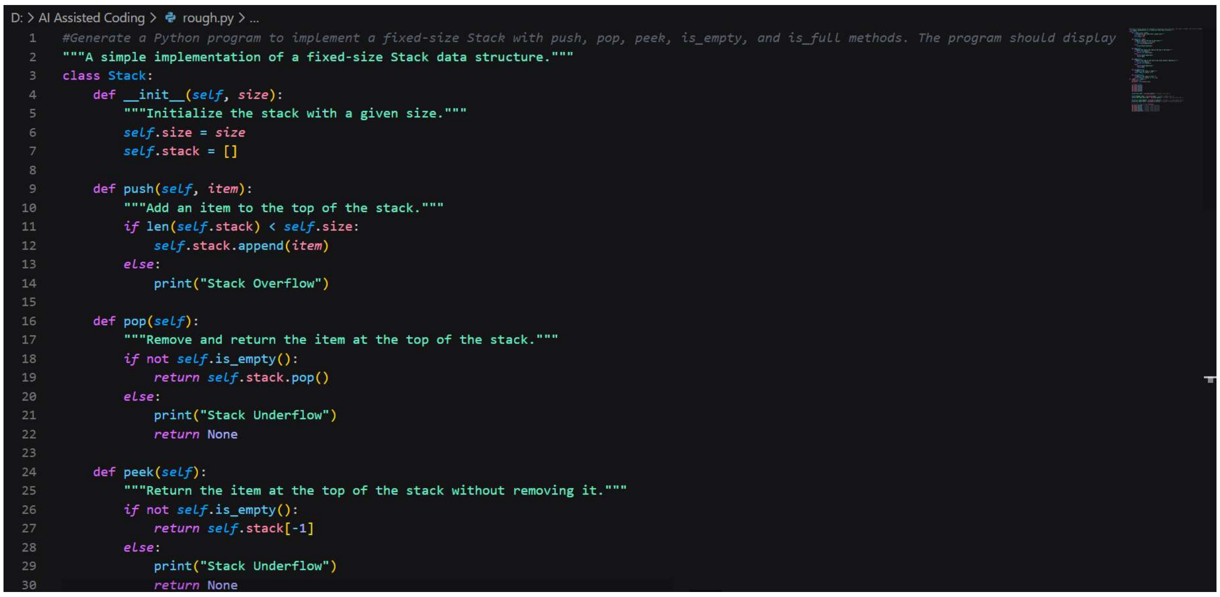
```
PS C:\Users\SAI TEJASWI> & "C:/Users/SAI TEJASWI/AppData/Local/Programs/Python/Python311/python.exe" "d:/AI Assisted Coding/rough.py"
Alice
30
None
True
None
False
PS C:\Users\SAI TEJASWI>
```

Explanation:
A Hash Table stores data using a hash function to compute an index. Collisions are handled using chaining (linked lists at each index). It supports fast insertion, searching, and deletion operations.

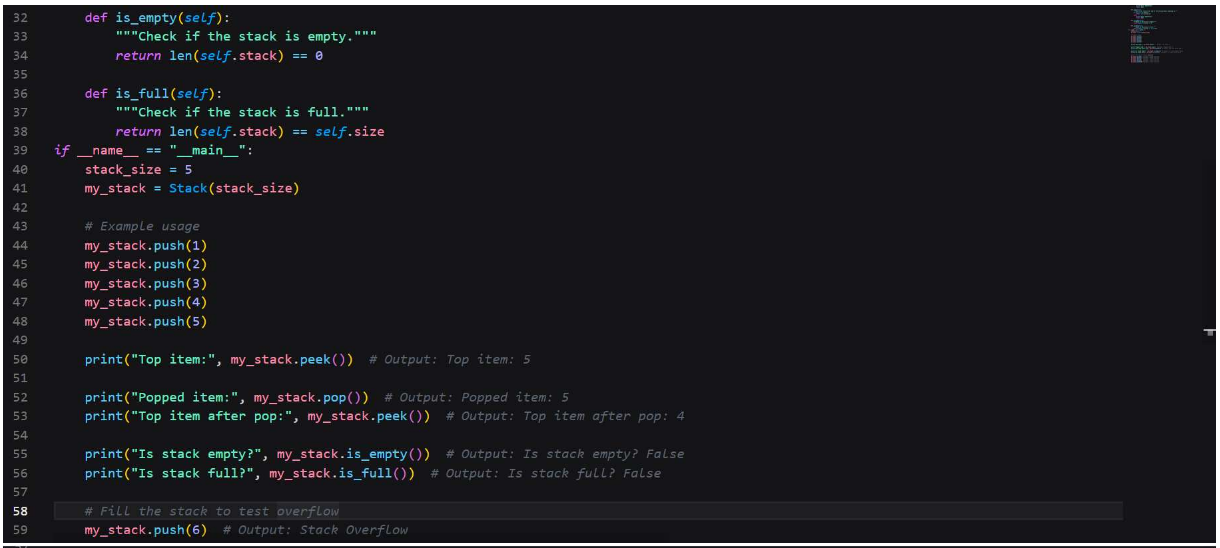
Task : Over Flow and Under Flow.

Prompt :
Generate a Python program to implement a fixed-size Stack with push, pop, peek, is_empty, and is_full methods. The program should display "Stack Overflow" when full and "Stack Underflow" when empty, with proper comments and example usage.

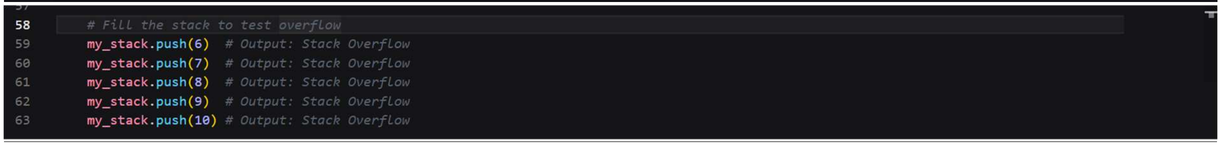
Code:



```python
#Generate a Python program to implement a fixed-size Stack with push, pop, peek, is_empty, and is_full methods. The program should display
"""A simple implementation of a fixed-size Stack data structure."""
class Stack:
    def __init__(self, size):
        """Initialize the stack with a given size."""
        self.size = size
        self.stack = []

    def push(self, item):
        """Add an item to the top of the stack."""
        if len(self.stack) < self.size:
            self.stack.append(item)
        else:
            print("Stack Overflow")

    def pop(self):
        """Remove and return the item at the top of the stack."""
        if not self.is_empty():
            return self.stack.pop()
        else:
            print("Stack Underflow")
            return None

    def peek(self):
        """Return the item at the top of the stack without removing it."""
        if not self.is_empty():
            return self.stack[-1]
        else:
            print("Stack Underflow")
            return None
```

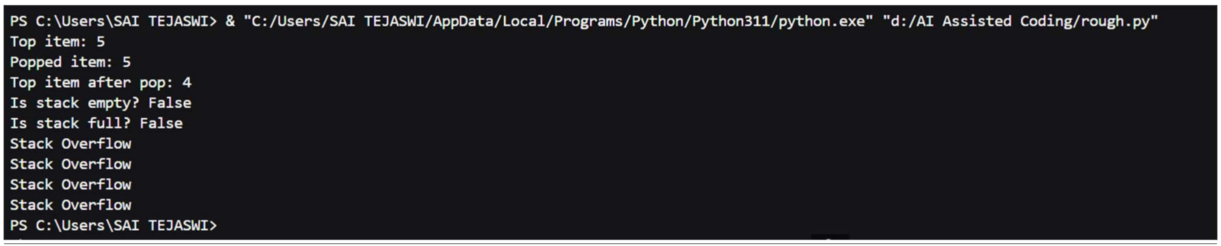


```python
    def is_empty(self):
        """Check if the stack is empty."""
        return len(self.stack) == 0

    def is_full(self):
        """Check if the stack is full."""
        return len(self.stack) == self.size
if __name__ == "__main__":
    stack_size = 5
    my_stack = Stack(stack_size)

    # Example usage
    my_stack.push(1)
    my_stack.push(2)
    my_stack.push(3)
    my_stack.push(4)
    my_stack.push(5)

    print("Top item:", my_stack.peek())  # Output: Top item: 5

    print("Popped item:", my_stack.pop())  # Output: Popped item: 5
    print("Top item after pop:", my_stack.peek())  # Output: Top item after pop: 4

    print("Is stack empty?", my_stack.is_empty())  # Output: Is stack empty? False
    print("Is stack full?", my_stack.is_full())  # Output: Is stack full? False

    # Fill the stack to test overflow
    my_stack.push(6)  # Output: Stack Overflow
```



```python
    # Fill the stack to test overflow
    my_stack.push(6)  # Output: Stack Overflow
    my_stack.push(7)  # Output: Stack Overflow
    my_stack.push(8)  # Output: Stack Overflow
    my_stack.push(9)  # Output: Stack Overflow
    my_stack.push(10) # Output: Stack Overflow
```

OUTPUT:



```
PS C:\Users\SAI TEJASWI> & "C:/Users/SAI TEJASWI/AppData/Local/Programs/Python/Python311/python.exe" "d:/AI Assisted Coding/rough.py"
Top item: 5
Popped item: 5
Top item after pop: 4
Is stack empty? False
Is stack full? False
Stack Overflow
Stack Overflow
Stack Overflow
Stack Overflow
PS C:\Users\SAI TEJASWI>
```

Explanation :

The program implements a fixed-size Stack following the LIFO (Last In First Out) principle using a list and a top pointer. The push() method checks if the stack is full and displays “Stack Overflow”, while pop() checks if it is empty and displays “Stack Underflow”. Helper methods like is_empty() and is_full() ensure proper boundary checking and safe stack operations.