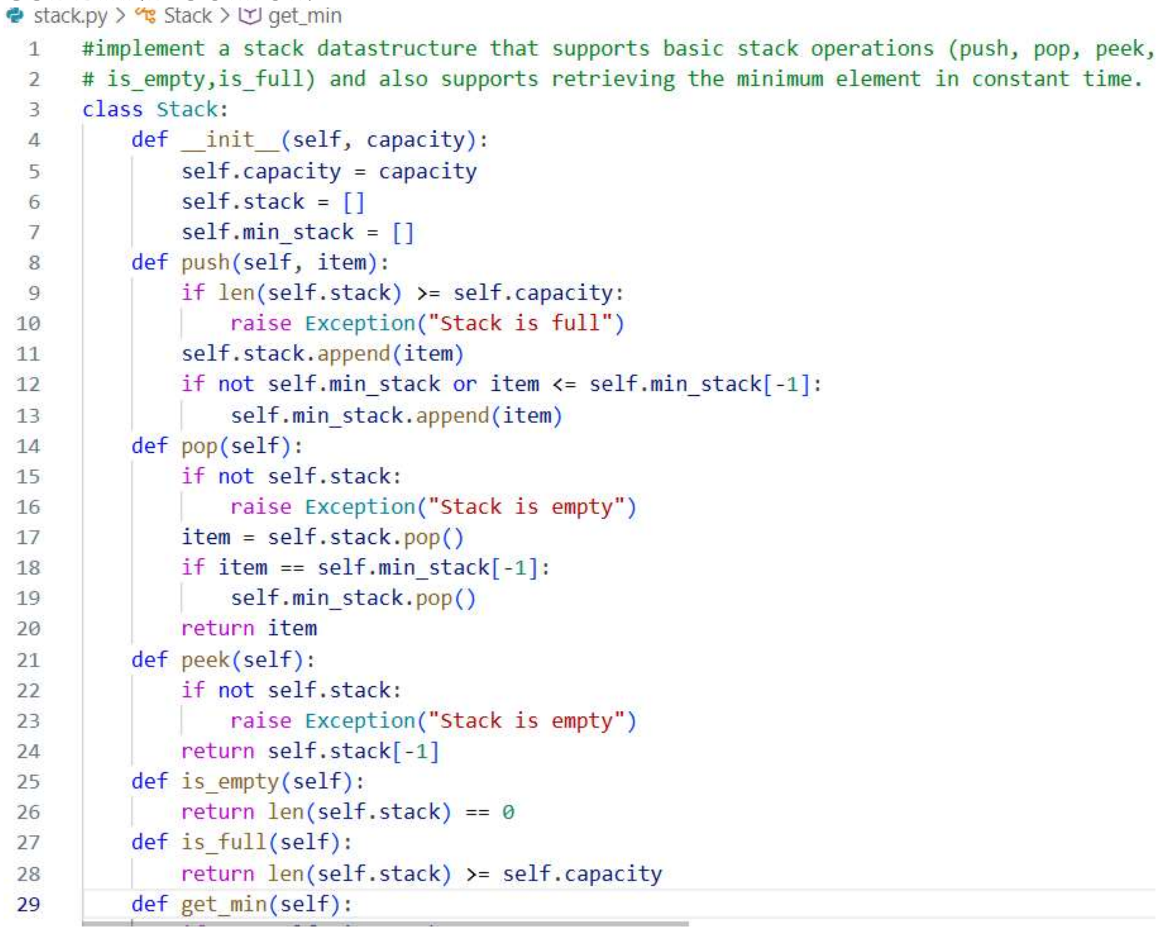
AI - Assisted Coding-Assignment - 11.2

HTNO:2303A52228
Batch:37

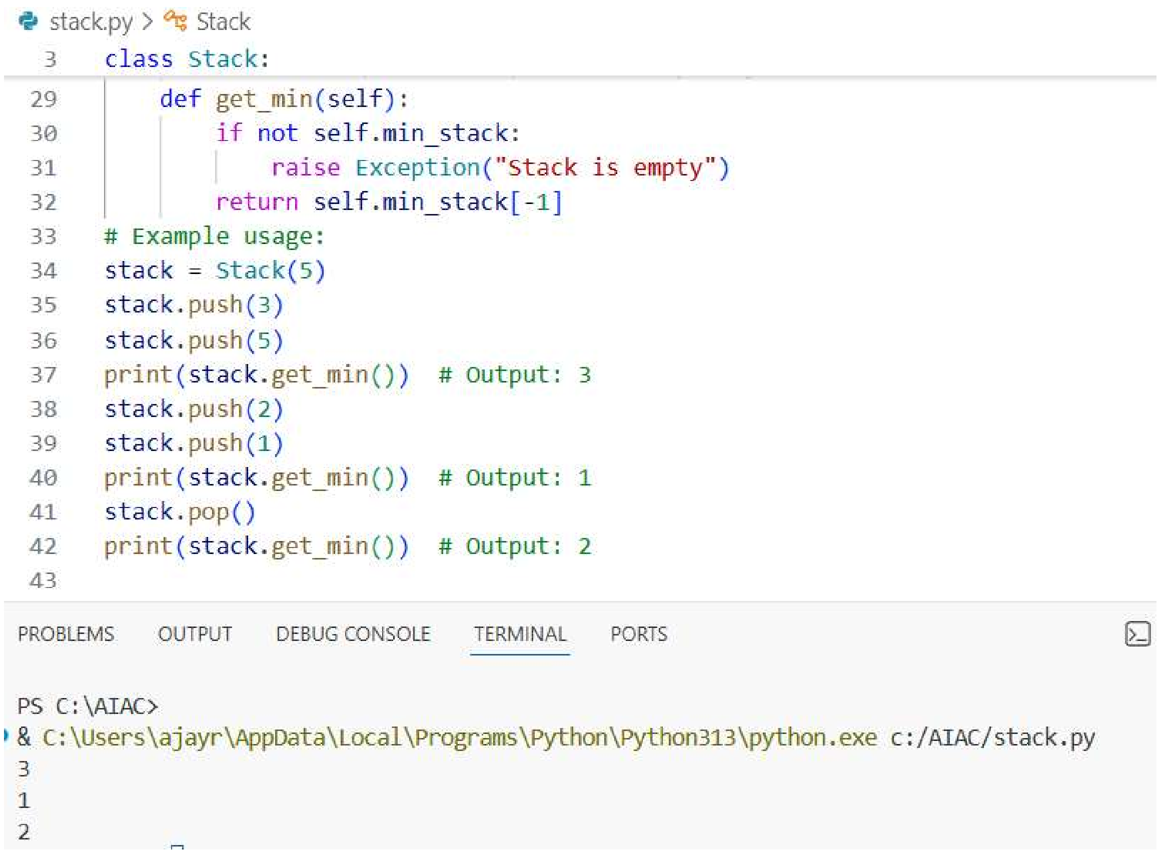
Lab 11 – Data Structures with AI: Implementing Fundamental Structures.

Task Description -1 – Stack Using AI Guidance.
Expected Output:
A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.
Prompt:
#implement a stack datastructure that supports basic stack operations (push, pop, peek,
is_empty,is_full) and also supports retrieving the minimum element in constant time.
CODE AND OUTPUT:



stack.py > Stack > get_min

```python
#implement a stack datastructure that supports basic stack operations (push, pop, peek,
# is_empty,is_full) and also supports retrieving the minimum element in constant time.
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = []
        self.min_stack = []
    def push(self, item):
        if len(self.stack) >= self.capacity:
            raise Exception("Stack is full")
        self.stack.append(item)
        if not self.min_stack or item <= self.min_stack[-1]:
            self.min_stack.append(item)
    def pop(self):
        if not self.stack:
            raise Exception("Stack is empty")
        item = self.stack.pop()
        if item == self.min_stack[-1]:
            self.min_stack.pop()
        return item
    def peek(self):
        if not self.stack:
            raise Exception("Stack is empty")
        return self.stack[-1]
    def is_empty(self):
        return len(self.stack) == 0
    def is_full(self):
        return len(self.stack) >= self.capacity
    def get_min(self):
```



stack.py > Stack

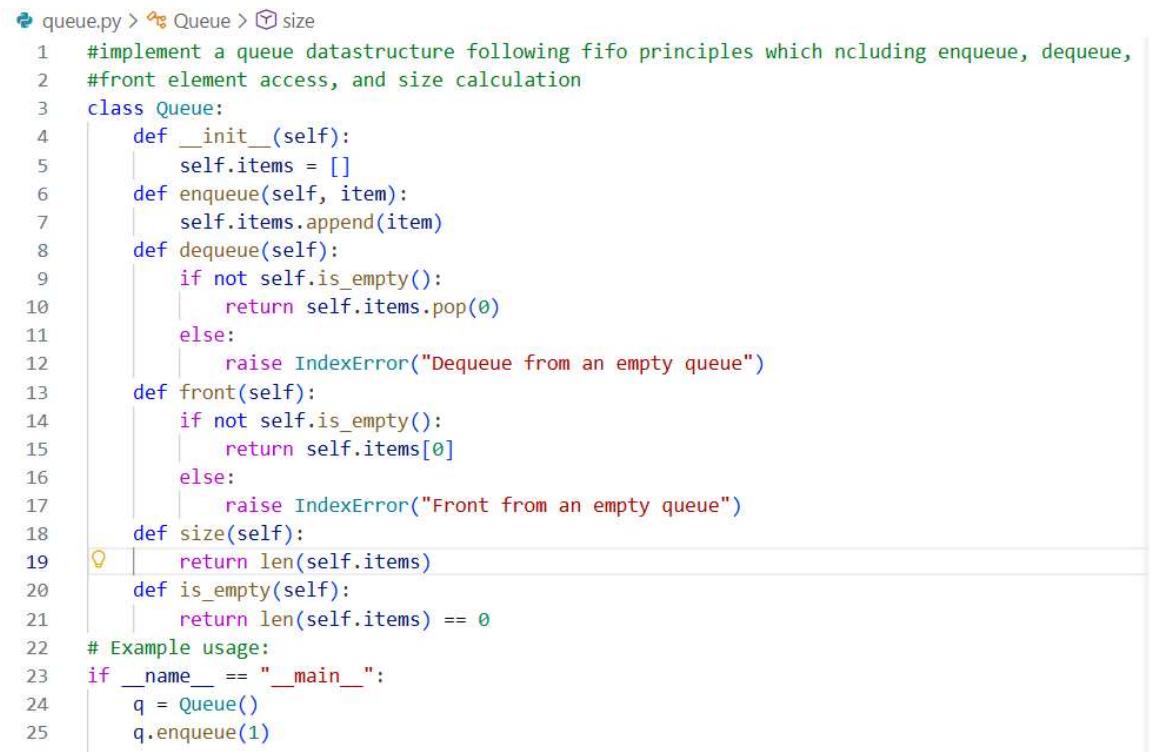
```
class Stack:
    def get_min(self):
        if not self.min_stack:
            raise Exception("Stack is empty")
        return self.min_stack[-1]
# Example usage:
stack = Stack(5)
stack.push(3)
stack.push(5)
print(stack.get_min())  # Output: 3
stack.push(2)
stack.push(1)
print(stack.get_min())  # Output: 1
stack.pop()
print(stack.get_min())  # Output: 2

```

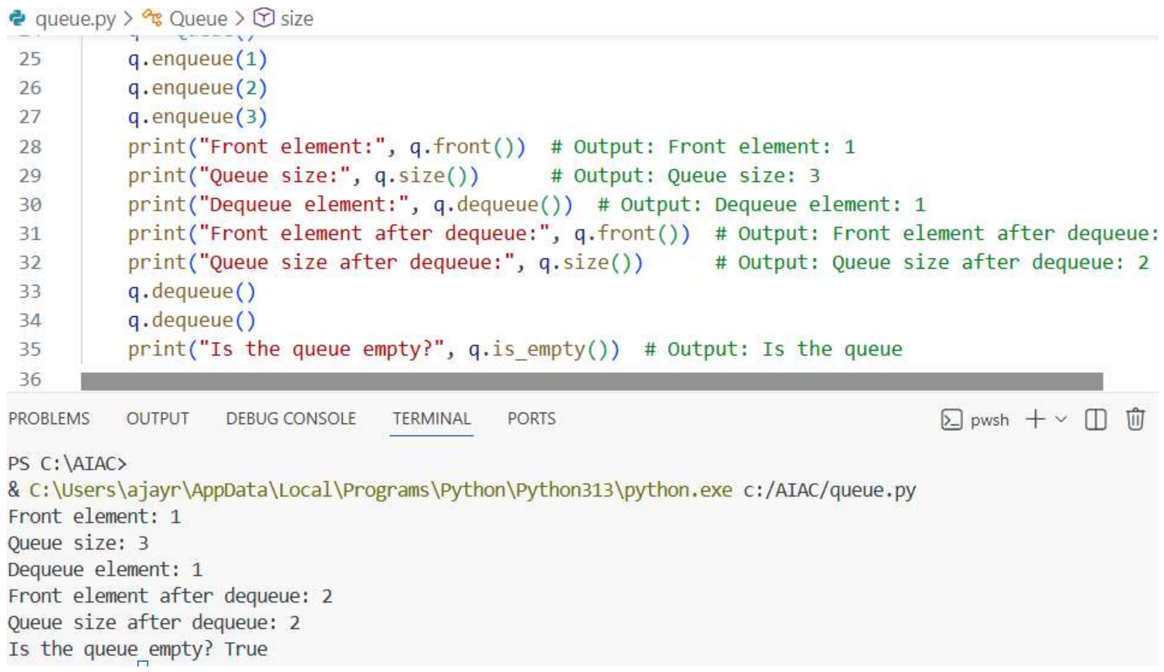
```
PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AIAC/stack.py
3
1
2
```

Explanation:
The code creates a Stack class using a Python list to store elements. The constructor initializes an empty list. The push() method adds an element to the top of the stack, while pop() removes and returns the top element after checking the stack is not empty. The peek() method shows the top element without removing it. The is_empty() method checks whether the stack has no elements, and size() returns how many elements are present. The example at the end demonstrates how to use these operations on the stack.

Task Description -2 – Queue Design.
Expected Output:
A complete Queue implementation including enqueue, dequeue,
front element access, and size calculation.
Prompt:
#implement a queue datastructure following fifo principles which ncluding enqueue,
#dequeue,front element access, and size calculation.
CODE AND OUTPUT:



```
queue.py > Queue > size
#implement a queue datastructure following fifo principles which ncluding enqueue, dequeue,
#front element access, and size calculation
class Queue:
    def __init__(self):
        self.items = []
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("Dequeue from an empty queue")
    def front(self):
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("Front from an empty queue")
    def size(self):
        return len(self.items)
    def is_empty(self):
        return len(self.items) == 0
# Example usage:
if __name__ == "__main__":
    q = Queue()
    q.enqueue(1)
```



```
queue.py > Queue > size
    q.enqueue(1)
    q.enqueue(2)
    q.enqueue(3)
    print("Front element:", q.front())  # Output: Front element: 1
    print("Queue size:", q.size())      # Output: Queue size: 3
    print("Dequeue element:", q.dequeue())  # Output: Dequeue element: 1
    print("Front element after dequeue:", q.front())  # Output: Front element after dequeue:
    print("Queue size after dequeue:", q.size())      # Output: Queue size after dequeue: 2
    q.dequeue()
    q.dequeue()
    print("Is the queue empty?", q.is_empty())  # Output: Is the queue
```

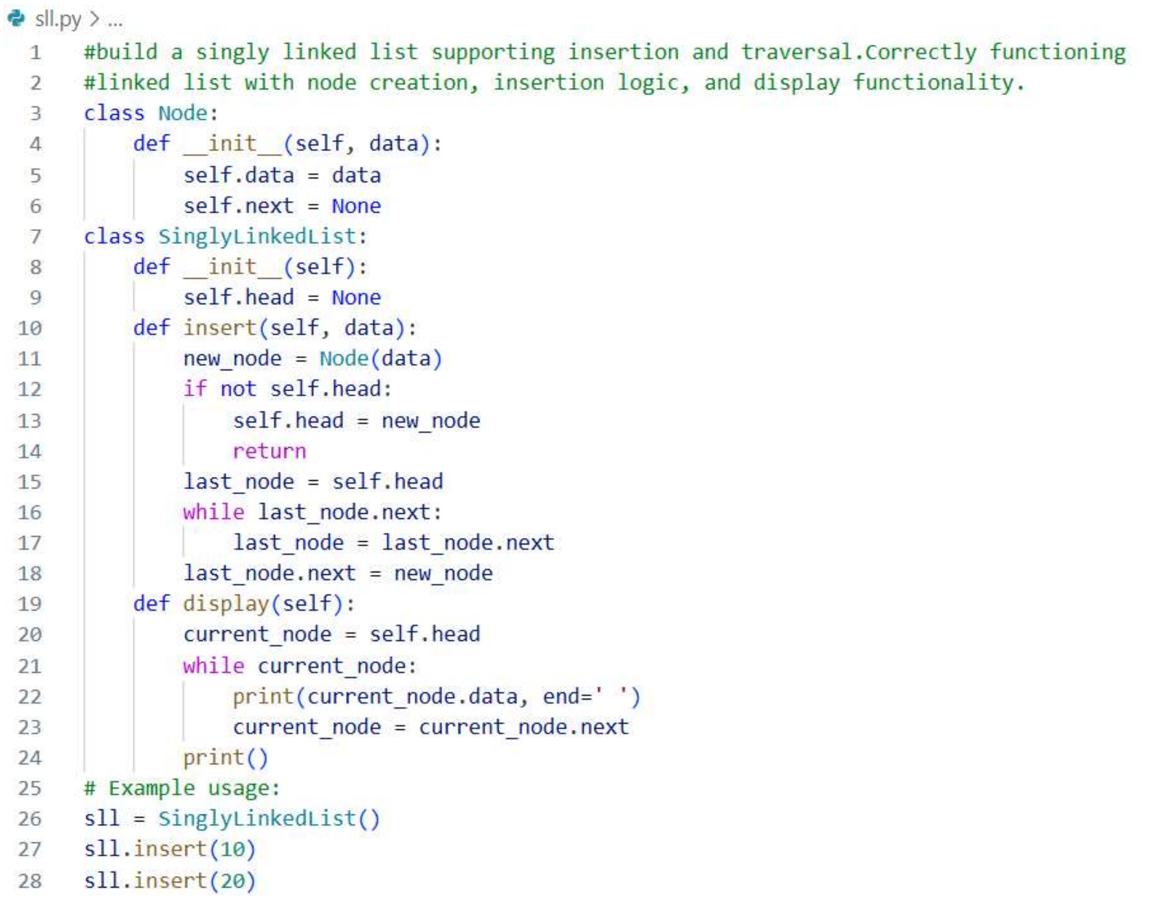
```
PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AIAC/queue.py
Front element: 1
Queue size: 3
Dequeue element: 1
Front element after dequeue: 2
Queue size after dequeue: 2
Is the queue empty? True
```

Explanation:

The code creates a Queue class using a Python list to store elements in FIFO order. The enqueue() method adds elements to the end, dequeue() removes the first element, and front() shows the first element without removing it. The size() method returns the number of elements, and is_empty() checks if the queue has no items.

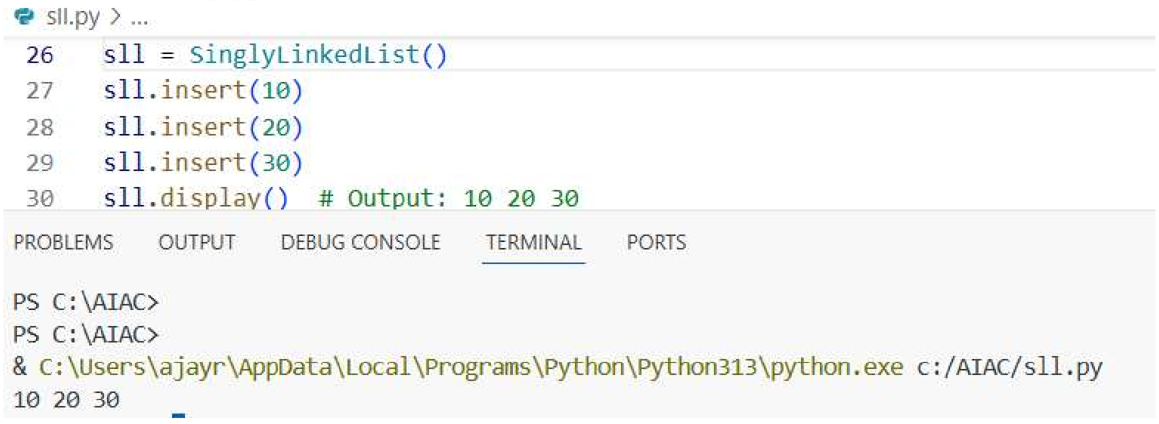
Task Description -3 –Singly Linked List Construction.

Expected Output:
Correctly functioning linked list with node creation, insertion logic, and display functionality.
Prompt:
#build a singly linked list supporting insertion and traversal.Correctly functioning #linked list with node creation, insertion logic, and display functionality.
CODE AND OUTPUT:



sll.py > ...

```python
#build a singly linked list supporting insertion and traversal.Correctly functioning
#linked list with node creation, insertion logic, and display functionality.
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if not self.head:
            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):
        current_node = self.head
        while current_node:
            print(current_node.data, end=' ')
            current_node = current_node.next
        print()
# Example usage:
sll = SinglyLinkedList()
sll.insert(10)
sll.insert(20)
```



sll.py > ...

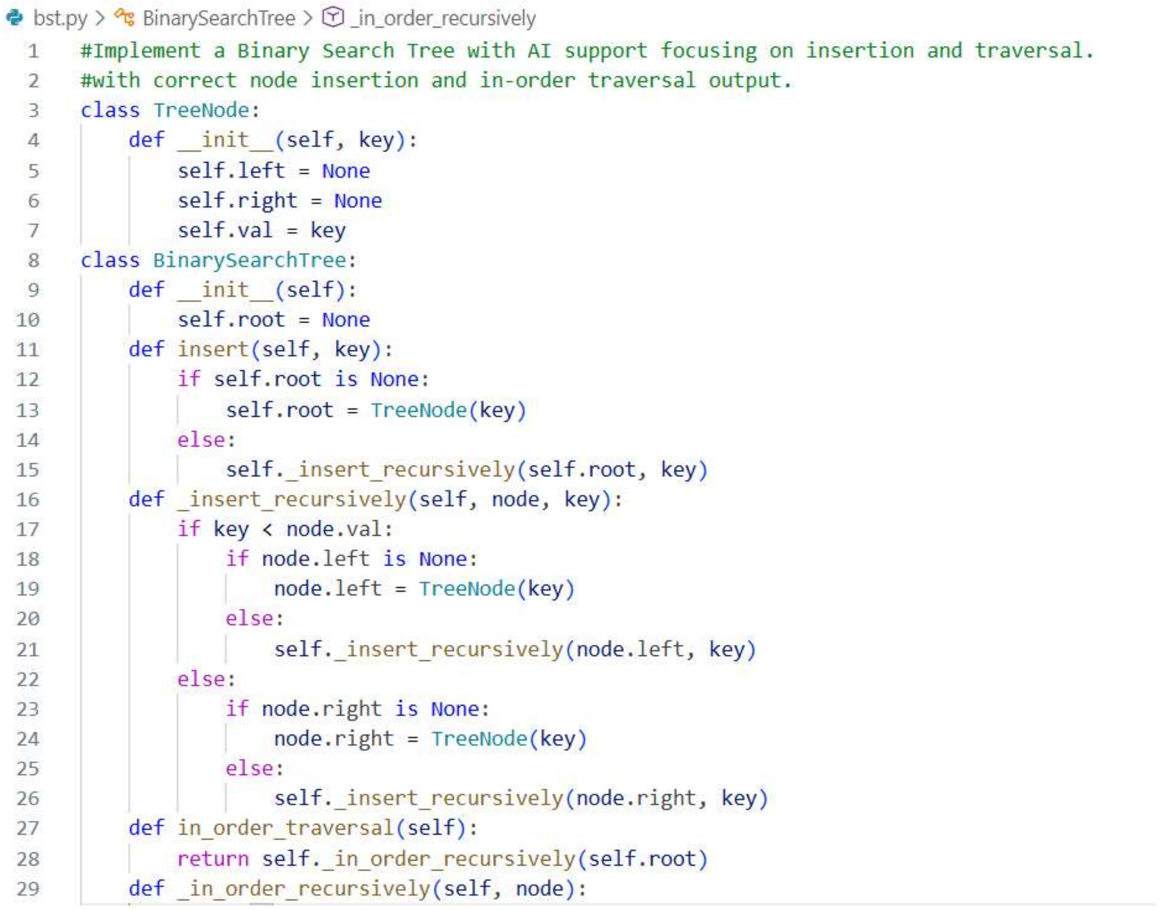
```python
sll = SinglyLinkedList()
sll.insert(10)
sll.insert(20)
sll.insert(30)
sll.display()  # Output: 10 20 30
```

```
PS C:\AIAC>
PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AIAC/sll.py
10 20 30
```

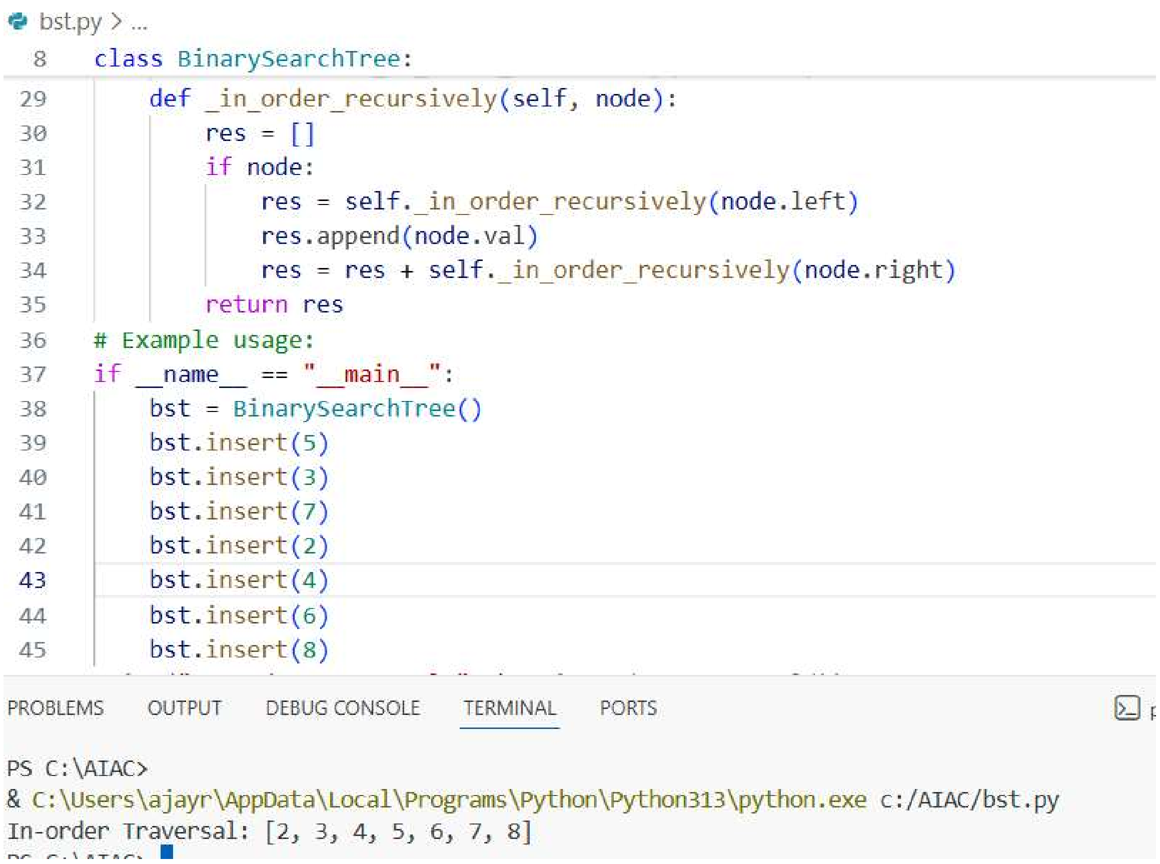
Explanation:
The code creates a singly linked list using a Node class to store data and a reference to the next node. The insert() method adds a new node at the end of the list, and the

display() method traverses from the head node to the end, printing each element. This shows how nodes are connected one after another in the list.

Task Description -4 – Binary Search Tree Operations.
Expected Output:
BST program with correct node insertion and in-order traversal output.
Prompt:
#Implement a Binary Search Tree with AI support focusing on insertion and traversal.
#with correct node insertion and in-order traversal output.
CODE AND OUTPUT:


bst.py > BinarySearchTree > _in_order_recursively

```python
#Implement a Binary Search Tree with AI support focusing on insertion and traversal.
#with correct node insertion and in-order traversal output.
class TreeNode:
    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 = TreeNode(key)
        else:
            self._insert_recursively(self.root, key)
    def _insert_recursively(self, node, key):
        if key < node.val:
            if node.left is None:
                node.left = TreeNode(key)
            else:
                self._insert_recursively(node.left, key)
        else:
            if node.right is None:
                node.right = TreeNode(key)
            else:
                self._insert_recursively(node.right, key)
    def in_order_traversal(self):
        return self._in_order_recursively(self.root)
    def _in_order_recursively(self, node):
```

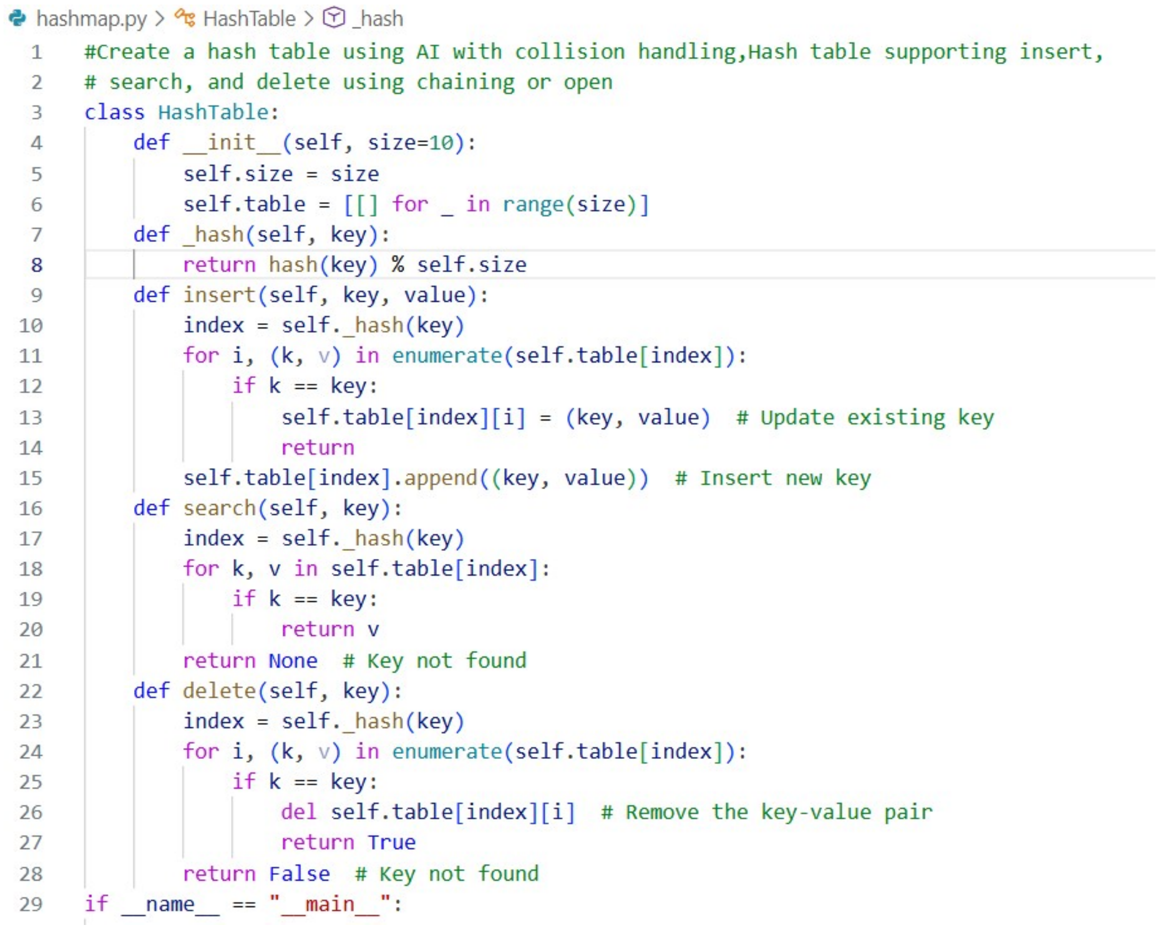


```python
class BinarySearchTree:
    def _in_order_recursively(self, node):
        res = []
        if node:
            res = self._in_order_recursively(node.left)
            res.append(node.val)
            res = res + self._in_order_recursively(node.right)
        return res
# Example usage:
if __name__ == "__main__":
    bst = BinarySearchTree()
    bst.insert(5)
    bst.insert(3)
    bst.insert(7)
    bst.insert(2)
    bst.insert(4)
    bst.insert(6)
    bst.insert(8)
```

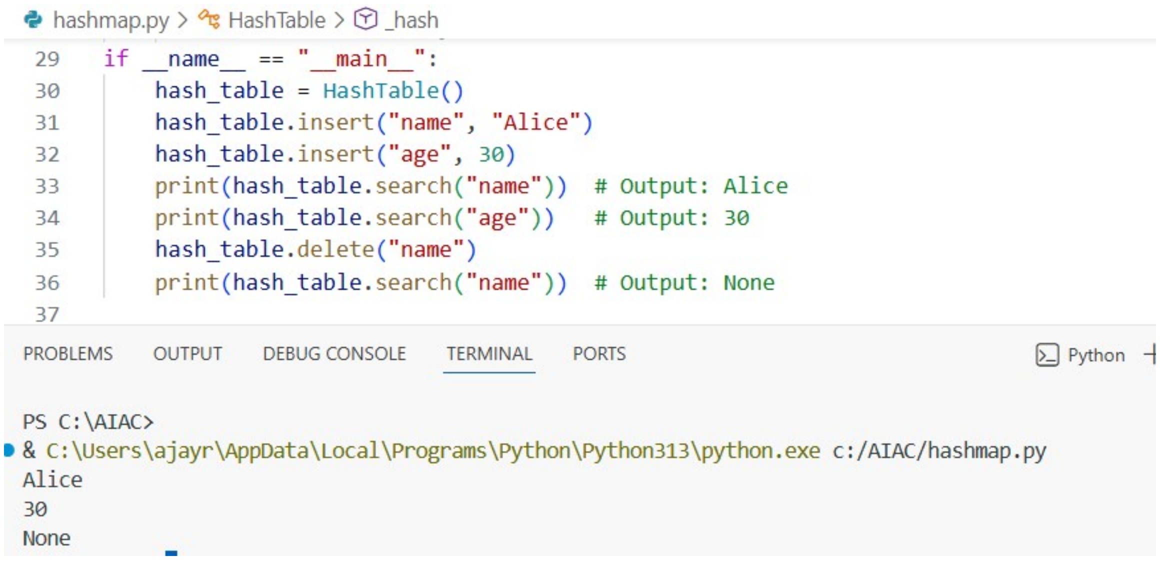
```
PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AIAC/bst.py
In-order Traversal: [2, 3, 4, 5, 6, 7, 8]
```

Explanation:
The program creates a Binary Search Tree where each node stores a value and links to left and right children. The insert() method places elements in the correct position based on BST rules (smaller to left, larger to right). The inorder() traversal prints nodes in sorted order by visiting left subtree, root, then right subtree.

Task Description -5 – Hash Table Implementation.
Expected Output:
Hash table supporting insert, search, and delete using chaining
or open.
Prompt:
#Create a hash table using AI with collision handling,Hash table supporting insert,
# search, and delete using chaining or open
CODE AND OUTPUT:



```python
#Create a hash table using AI with collision handling,Hash table supporting insert,
# search, and delete using chaining or open
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]
    def _hash(self, key):
        return hash(key) % self.size
    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))  # Insert new key
    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        return None  # Key not found
    def delete(self, key):
        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 False  # Key not found
if __name__ == "__main__":
```



```python
if __name__ == "__main__":
    hash_table = HashTable()
    hash_table.insert("name", "Alice")
    hash_table.insert("age", 30)
    print(hash_table.search("name"))  # Output: Alice
    print(hash_table.search("age"))   # Output: 30
    hash_table.delete("name")
    print(hash_table.search("name"))  # Output: None
```

```
PS C:\AIAC>
& C:\Users\ajayr\AppData\Local\Programs\Python\Python313\python.exe c:/AIAC/hashmap.py
Alice
30
None
```

Explanation:
The program implements a hash table using chaining to handle collisions. A list of buckets is created, where each bucket stores key–value pairs. The _hash() function converts a key into an index of the table. The insert() method places the key–value pair in the correct bucket and updates it if the key already exists. The search() method looks

through the bucket to find and return the value for a given key.