Name: Nikhil N

HT No: 2303A52477

Lab 11: Data Structures with AI

Assignment: 11.1

Theme: Implementing Fundamental Structures

Task 1: Stack Using AI Guidance

Exact Task Description

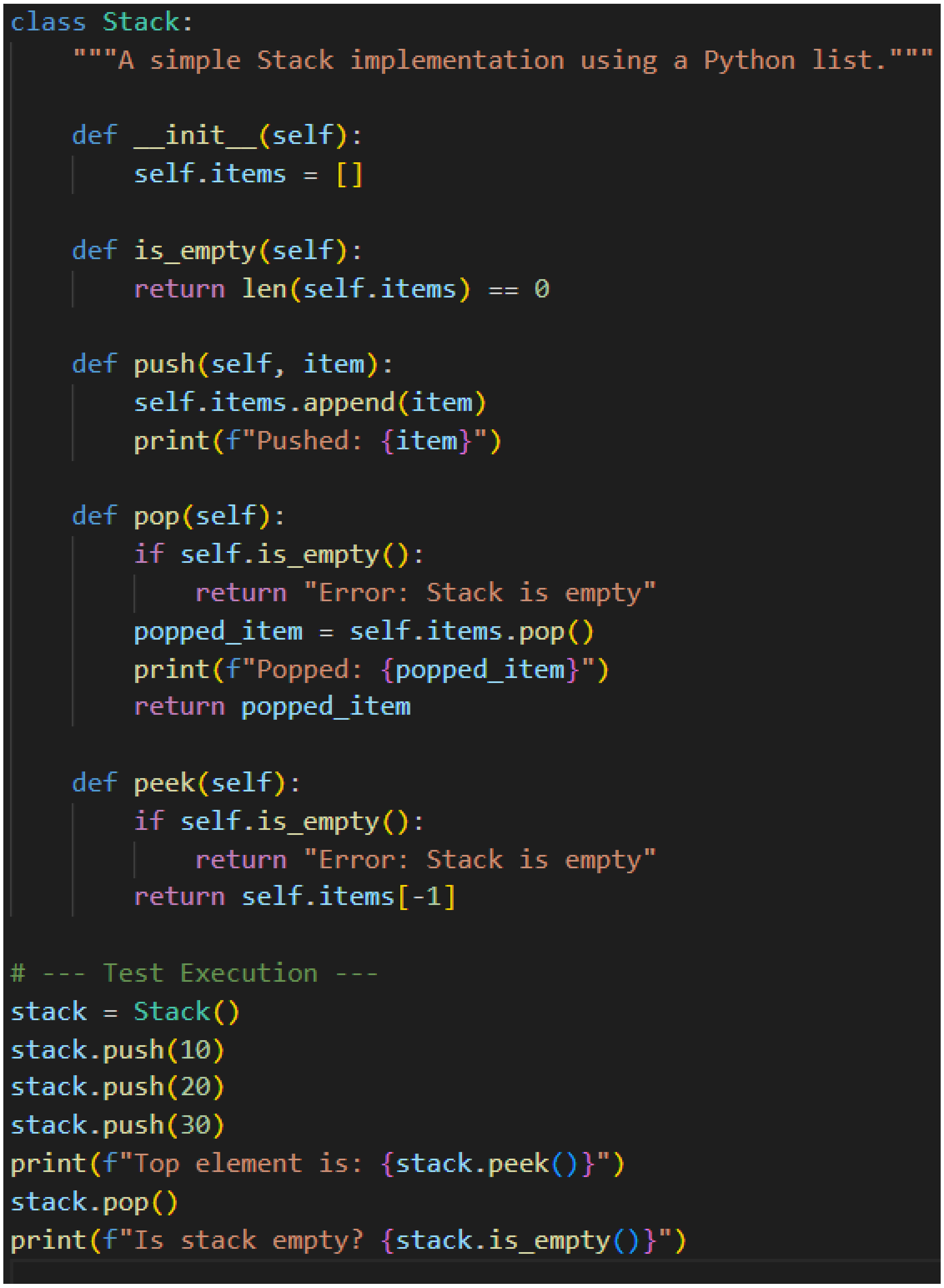
Task: With the help of AI, design and implement a Stack data structure supporting basic stack operations.

Expected Output: A Python Stack class supporting push, pop, peek, and empty-check operations with proper documentation.

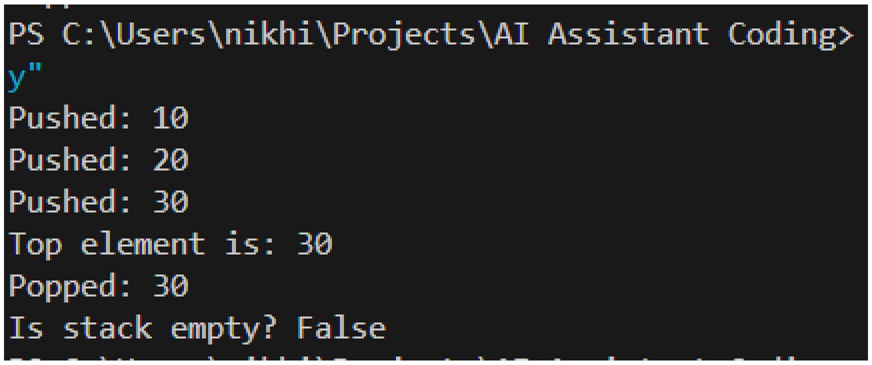
Prompt

> "Generate a Python class for a Stack data structure. Include methods for push, pop, peek, and is_empty. Provide well-documented code with an example usage to test all operations."

Code


```python
class Stack:
    """A simple Stack implementation using a Python list."""

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

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

    def push(self, item):
        self.items.append(item)
        print(f"Pushed: {item}")

    def pop(self):
        if self.is_empty():
            return "Error: Stack is empty"
        popped_item = self.items.pop()
        print(f"Popped: {popped_item}")
        return popped_item

    def peek(self):
        if self.is_empty():
            return "Error: Stack is empty"
        return self.items[-1]

# --- Test Execution ---
stack = Stack()
stack.push(10)
stack.push(20)
stack.push(30)
print(f"Top element is: {stack.peek()}")
stack.pop()
print(f"Is stack empty? {stack.is_empty()}")
```

Output



```
PS C:\Users\nikhi\Projects\AI Assistant Coding>
y"
Pushed: 10
Pushed: 20
Pushed: 30
Top element is: 30
Popped: 30
Is stack empty? False
```

Explanation

A Stack operates on a "Last-In, First-Out" (LIFO) principle. Think of it exactly like a stack of plates in a cafeteria. When you put a new plate on the stack (push), it goes right on top. When someone takes a plate (pop), they take the one from the very top. You can't easily grab the bottom plate without ruining the stack! In our Python code, we used a standard list to represent our plates, using .append() to add to the top and .pop() to remove from the top.

Task 2: Queue Design

Exact Task Description

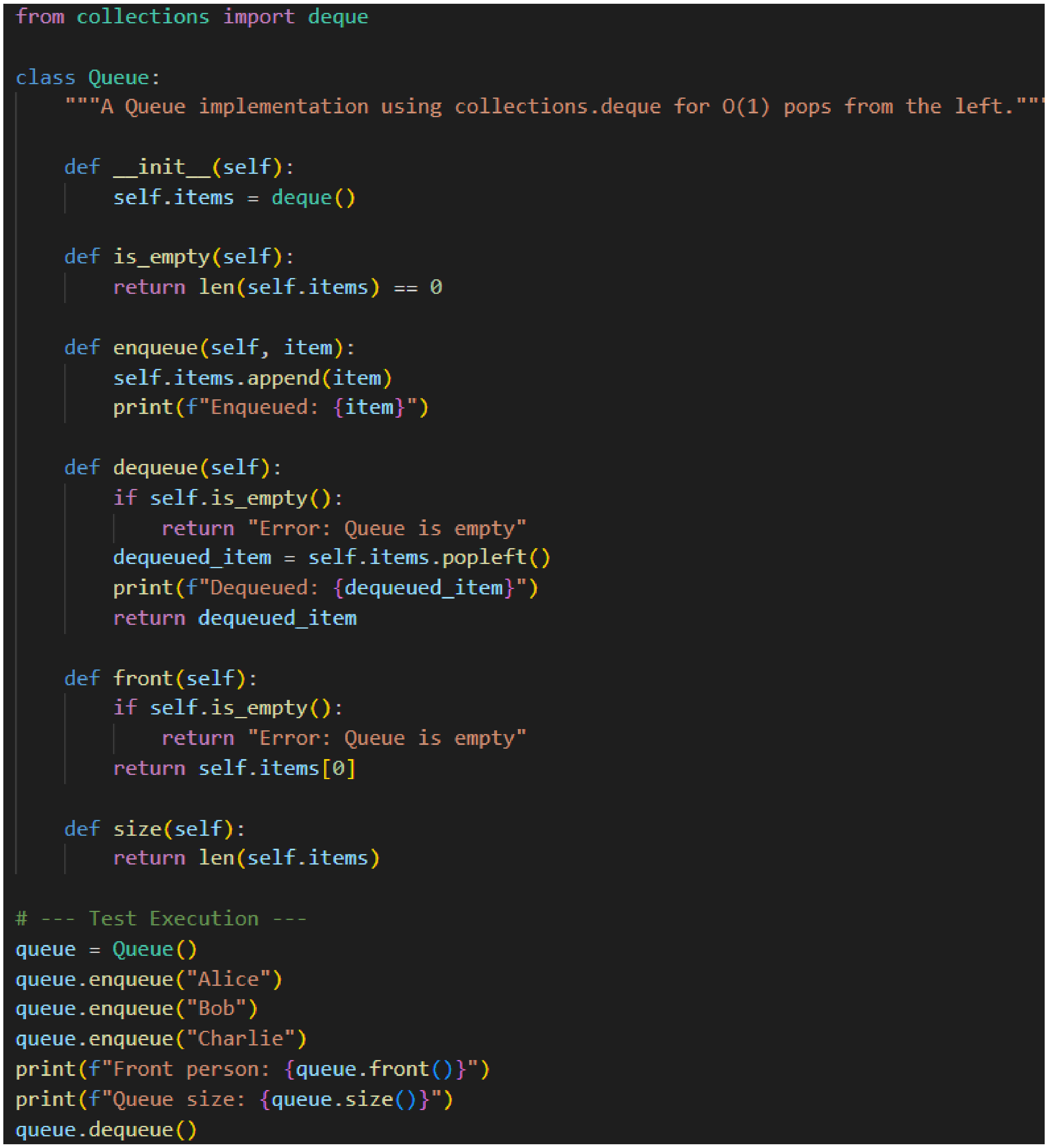
Task: Use AI assistance to create a Queue data structure following FIFO principles

Expected Output: A complete Queue implementation including enqueue, dequeue, front element access, and size calculation.

Prompt

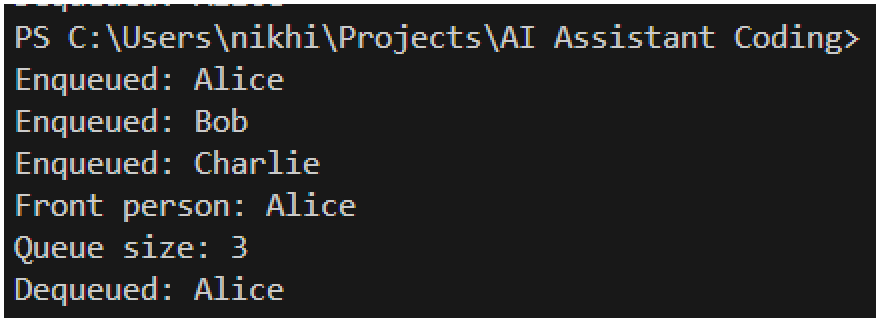
> "Write a Python class for a Queue data structure following the FIFO principle. Use collections.deque for better performance. Include methods for enqueue, dequeue, getting the front element, and calculating the size."

Code



```python
from collections import deque

class Queue:
    """A Queue implementation using collections.deque for O(1) pops from the left."""

    def __init__(self):
        self.items = deque()

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

    def enqueue(self, item):
        self.items.append(item)
        print(f"Enqueued: {item}")

    def dequeue(self):
        if self.is_empty():
            return "Error: Queue is empty"
        dequeued_item = self.items.popleft()
        print(f"Dequeued: {dequeued_item}")
        return dequeued_item

    def front(self):
        if self.is_empty():
            return "Error: Queue is empty"
        return self.items[0]

    def size(self):
        return len(self.items)

# --- Test Execution ---
queue = Queue()
queue.enqueue("Alice")
queue.enqueue("Bob")
queue.enqueue("Charlie")
print(f"Front person: {queue.front()}")
print(f"Queue size: {queue.size()}")
queue.dequeue()
```

Output



```
PS C:\Users\nikhi\Projects\AI Assistant Coding>
Enqueued: Alice
Enqueued: Bob
Enqueued: Charlie
Front person: Alice
Queue size: 3
Dequeued: Alice
```

Explanation

Unlike a stack, a Queue follows the "First-In, First-Out" (FIFO) rule. Imagine a line of people waiting to buy movie tickets. The first person to join the line is the first person to get their ticket and leave. Adding someone to the back of the line is enqueue, and serving the person at the front is dequeue. We specifically used Python's deque (double-ended queue) here because standard Python lists are very slow when you try to remove elements from the very beginning.

Task 3: Singly Linked List Construction

Exact Task Description

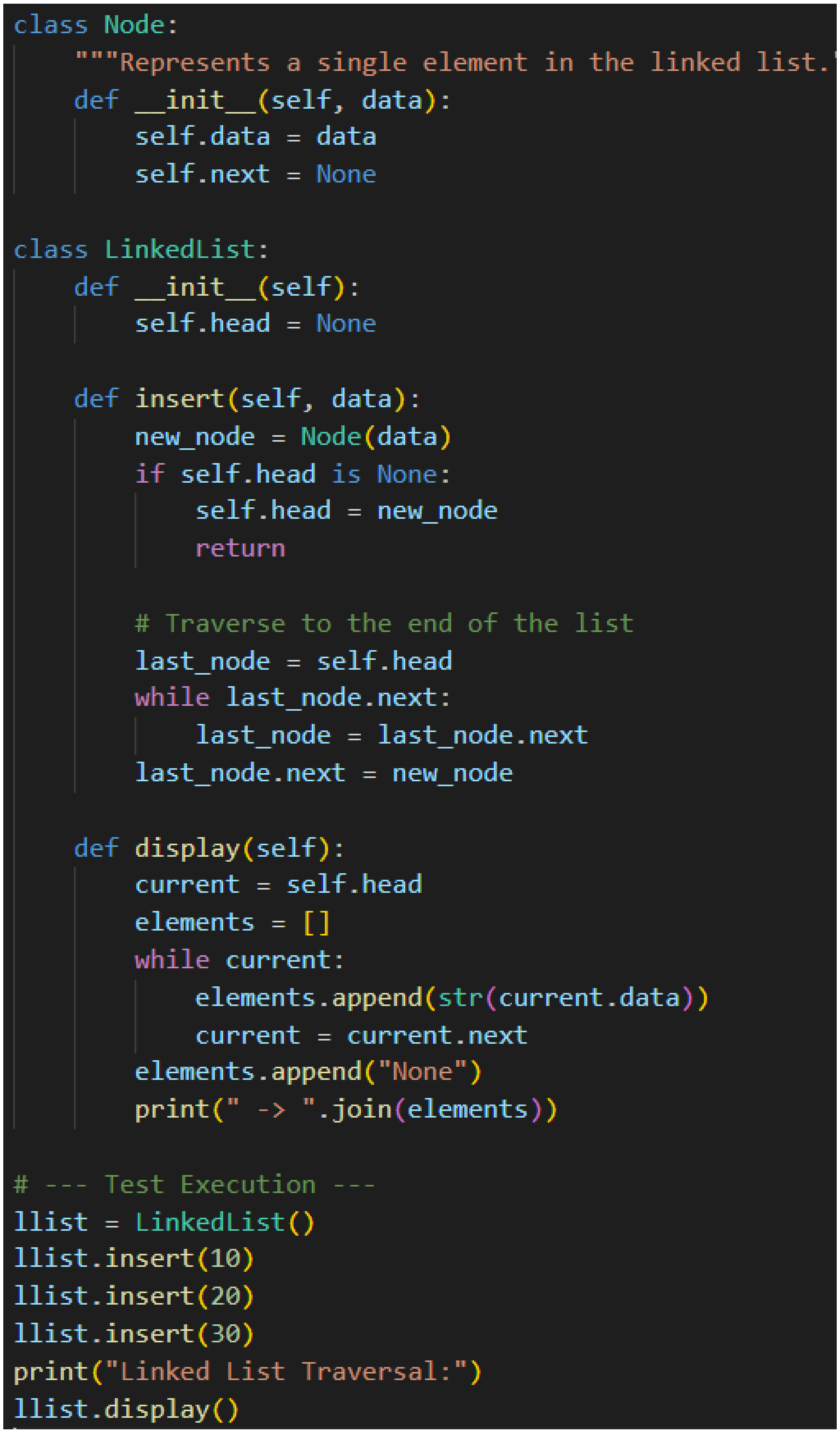
Task: Utilize AI to build a singly linked list supporting insertion and traversal.

Expected Output: Correctly functioning linked list with node creation, insertion logic, and display functionality.

Prompt

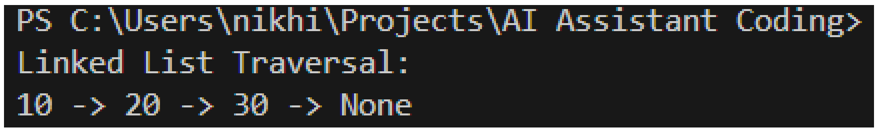
> "Create a Python implementation of a Singly Linked List. I need a Node class to hold data and the next pointer, and a LinkedList class with methods to insert a new node at the end and to traverse/display the list."

Code



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

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

    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return

        # Traverse to the end of the list
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def display(self):
        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        elements.append("None")
        print(" -> ".join(elements))

# --- Test Execution ---
llist = LinkedList()
llist.insert(10)
llist.insert(20)
llist.insert(30)
print("Linked List Traversal:")
llist.display()
```

Output



```
PS C:\Users\nikhi\Projects\AI Assistant Coding>
Linked List Traversal:
10 -> 20 -> 30 -> None
```

Explanation

Normally, lists (arrays) store items right next to each other in the computer's memory. A Linked List is different—it's like a scavenger hunt. The first item (head) holds some data, plus a clue (a "pointer") telling the computer exactly where to find the *next* item. To add a new item at the end, the program has to start at the beginning, follow the clues all the way to the last item, and then update that last clue to point to our newly added item.

Task 4: Binary Search Tree Operations

Exact Task Description

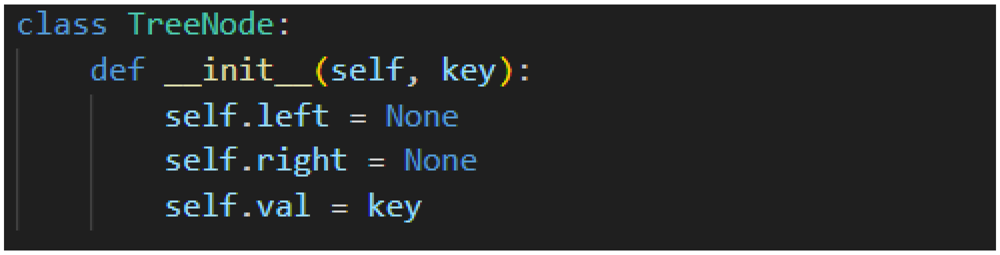
Task: Implement a Binary Search Tree with AI support focusing on insertion and traversal.

Expected Output: BST program with correct node insertion and in-order traversal output.

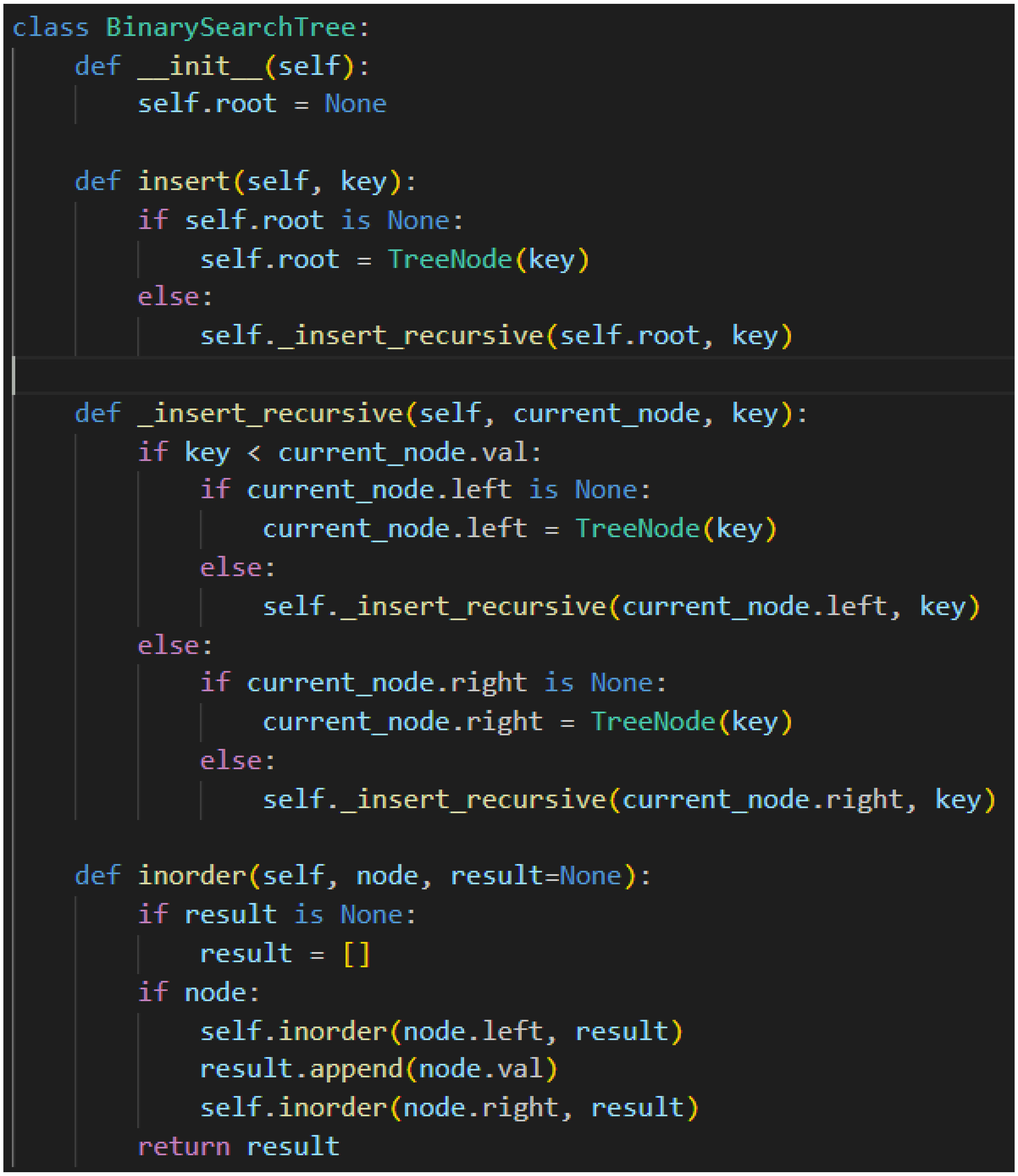
Prompt

> "Write a Python program to implement a Binary Search Tree (BST). Create a Node class, and functions to insert new values following BST rules. Finally, include an inorder_traversal function to print the nodes."

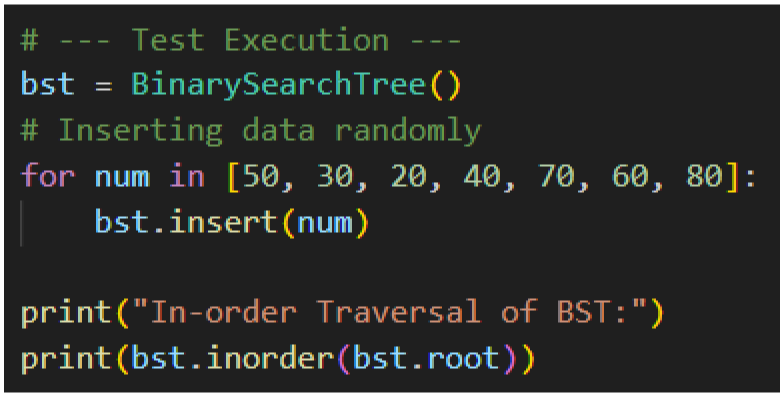
Code



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

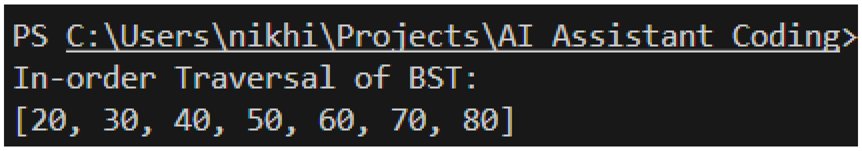


```python
class BinarySearchTree:
    def __init__(self):
        self.root = None

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

    def _insert_recursive(self, current_node, key):
        if key < current_node.val:
            if current_node.left is None:
                current_node.left = TreeNode(key)
            else:
                self._insert_recursive(current_node.left, key)
        else:
            if current_node.right is None:
                current_node.right = TreeNode(key)
            else:
                self._insert_recursive(current_node.right, key)

    def inorder(self, node, result=None):
        if result is None:
            result = []
        if node:
            self.inorder(node.left, result)
            result.append(node.val)
            self.inorder(node.right, result)
        return result
```



```python
# --- Test Execution ---
bst = BinarySearchTree()
# Inserting data randomly
for num in [50, 30, 20, 40, 70, 60, 80]:
    bst.insert(num)

print("In-order Traversal of BST:")
print(bst.inorder(bst.root))
```

Output



```
PS C:\Users\nikhi\Projects\AI Assistant Coding>
In-order Traversal of BST:
[20, 30, 40, 50, 60, 70, 80]
```

Explanation

A Binary Search Tree (BST) organizes data hierarchically, like an organizational chart, but with a strict mathematical rule: for any given "parent" node, everything branching to its left *must* be smaller, and everything branching to its right *must* be larger. When we insert data, the code asks, "Is this new number smaller or larger than where I am?" and walks down the tree until it finds an empty spot. The magic of a BST is that if you read it "In-Order" (Left child, then Parent, then Right child), it naturally spits the numbers out perfectly sorted!

Task 5: Hash Table Implementation

Exact Task Description

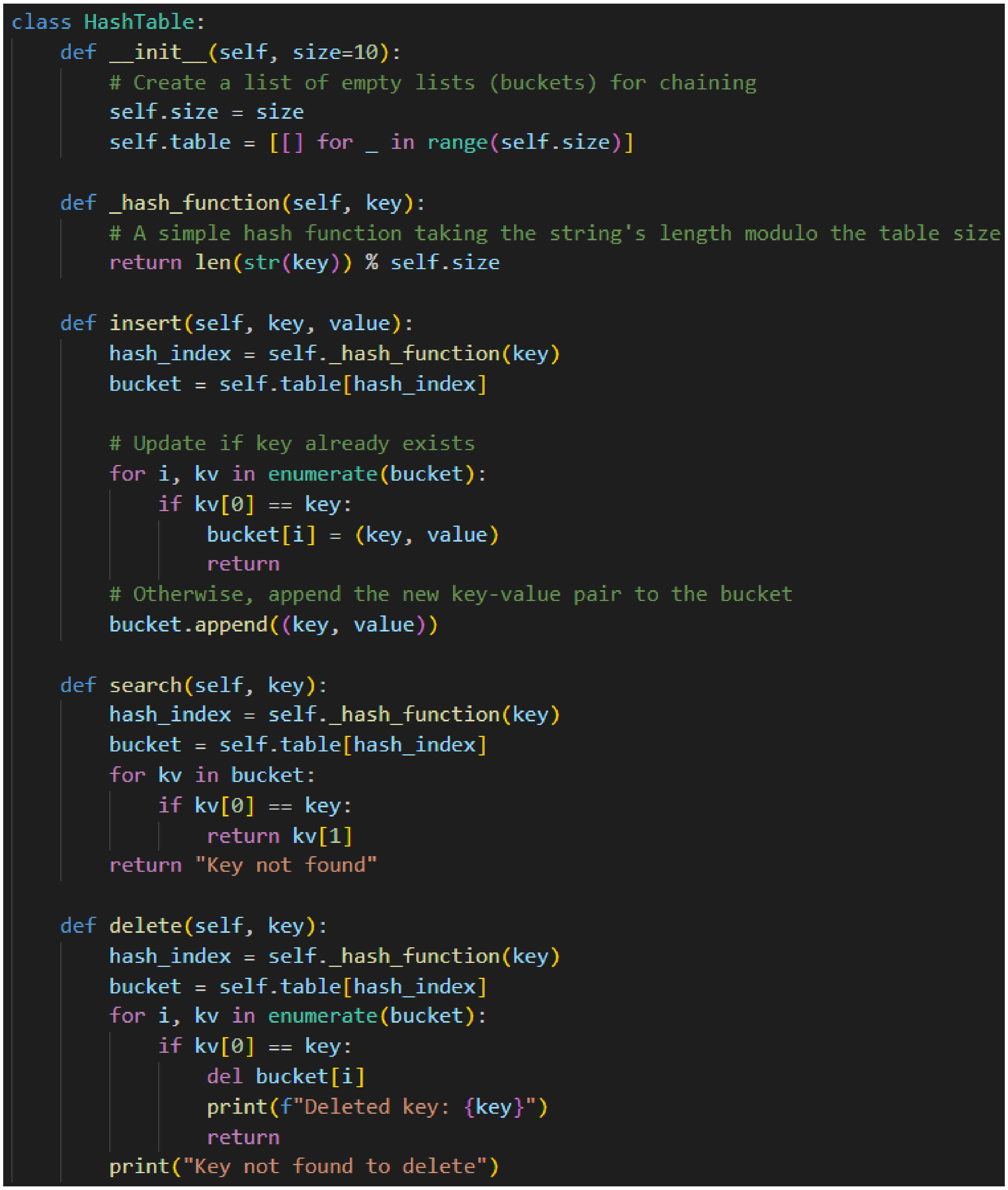
Task: Create a hash table using AI with collision handling

Expected Output: Hash table supporting insert, search, and delete using chaining or open addressing.

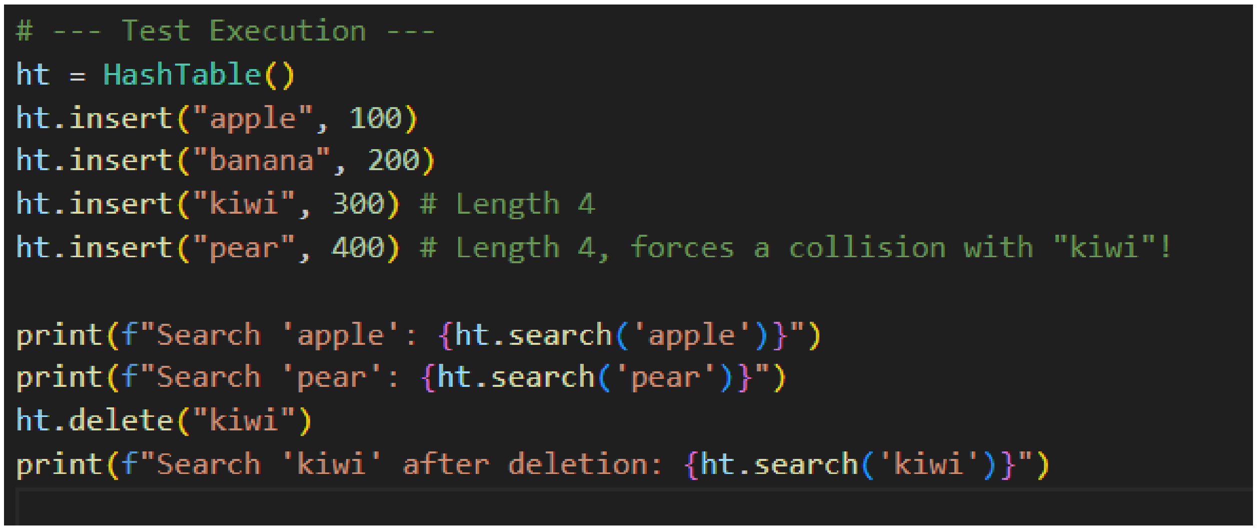
Prompt

> "Implement a Hash Table from scratch in Python without using the built-in dictionary. Support insert, search, and delete operations. Handle collisions using the chaining method (using nested lists)."

Code

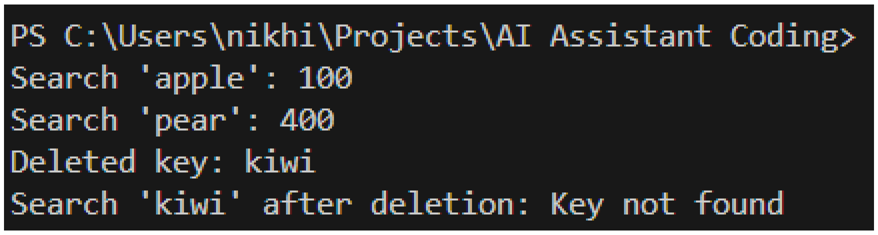


```python
class HashTable:
    def __init__(self, size=10):
        # Create a list of empty lists (buckets) for chaining
        self.size = size
        self.table = [[] for _ in range(self.size)]

    def _hash_function(self, key):
        # A simple hash function taking the string's length modulo the table size
        return len(str(key)) % self.size

    def insert(self, key, value):
        hash_index = self._hash_function(key)
        bucket = self.table[hash_index]

        # Update if key already exists
        for i, kv in enumerate(bucket):
            if kv[0] == key:
                bucket[i] = (key, value)
                return
        # Otherwise, append the new key-value pair to the bucket
        bucket.append((key, value))

    def search(self, key):
        hash_index = self._hash_function(key)
        bucket = self.table[hash_index]
        for kv in bucket:
            if kv[0] == key:
                return kv[1]
        return "Key not found"

    def delete(self, key):
        hash_index = self._hash_function(key)
        bucket = self.table[hash_index]
        for i, kv in enumerate(bucket):
            if kv[0] == key:
                del bucket[i]
                print(f"Deleted key: {key}")
                return
        print("Key not found to delete")
```



```python
# --- Test Execution ---
ht = HashTable()
ht.insert("apple", 100)
ht.insert("banana", 200)
ht.insert("kiwi", 300) # Length 4
ht.insert("pear", 400) # Length 4, forces a collision with "kiwi"!

print(f"Search 'apple': {ht.search('apple')}")
print(f"Search 'pear': {ht.search('pear')}")
ht.delete("kiwi")
print(f"Search 'kiwi' after deletion: {ht.search('kiwi')}")
```

Output



```
PS C:\Users\nikhi\Projects\AI Assistant Coding>
Search 'apple': 100
Search 'pear': 400
Deleted key: kiwi
Search 'kiwi' after deletion: Key not found
```

Explanation

A Hash Table is essentially a giant filing cabinet with numbered drawers. We pass a word (the "key") through a mathematical formula (the "hash function") to instantly calculate which drawer it belongs in. This makes finding data incredibly fast. However, sometimes two different words get assigned the exact same drawer—this is called a "collision" (like "kiwi" and "pear" in our code). To solve this, we used "chaining." This simply means that instead of throwing the second word away, we just toss both items into a list inside that specific drawer, so they can coexist peacefully.