AI ASSISSTED CODING ASSIGNMENT 11.1

NAME: SANJANA

Batch no:05

ROLLNO:2403A510D4

COURSE: AI assissted coding

• Enhance code quality with AI-generated comments and performance suggestions.

Task Description #1 –

Stack Implementation
Task: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.
Sample Input Code:
class Stack:
pass
Expected Output:
• A functional stack implementation with all required methods and docstrings

PROMPT:

generate a Stack class with push, pop, peek, and is_empty methods.
.

Code:

ai.py

```python
from typing import TypeVar, Generic, List

# Define a generic type 'T' to allow the stack to hold any type of element.
T = TypeVar('T')

class Stack(Generic[T]):
    """
    A simple and efficient implementation of a Stack data structure (LIFO).

    This class provides the fundamental stack operations: push, pop, peek,
    and is_empty. It is built upon Python's list type.

    Attributes:
        _items (List[T]): A private list to store the stack elements.
    """

    def __init__(self) -> None:
        """Initializes a new, empty stack."""
        self._items: List[T] = []

    def push(self, item: T) -> None:
        """
        Adds an item to the top of the stack.

        This operation has an average time complexity of O(1).

        Args:
            item (T): The item to be added.
        """
        self._items.append(item)
```

```python
class Stack(Generic[T]):

    def pop(self) -> T:
        """
        Removes and returns the item from the top of the stack.

        This operation has a time complexity of O(1).

        Returns:
            T: The item that was at the top of the stack.

        Raises:
            IndexError: If pop is called on an empty stack.
        """
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        return self._items.pop()

    def peek(self) -> T:
        """
        Returns the item at the top of the stack without removing it.

        This operation has a time complexity of O(1).

        Returns:
            T: The item currently at the top of the stack.

        Raises:
            IndexError: If peek is called on an empty stack.
        """
```

ai.py > ...

```python
class Stack(Generic[T]):
    def peek(self) -> T:
        if self.is_empty():
            raise IndexError("peek from an empty stack")
        return self._items[-1]

    def is_empty(self) -> bool:
        """
        Checks if the stack contains any items.

        This operation has a time complexity of O(1).

        Returns:
            bool: True if the stack is empty, False otherwise.
        """
        return not self._items

    def size(self) -> int:
        """
        Returns the number of items in the stack.

        Returns:
            int: The current size of the stack.
        """
        return len(self._items)

    def __str__(self) -> str:
        """
        Provides a string representation of the stack.
```

```python
class Stack(Generic[T]):
    def __str__(self) -> str:

        Returns:
            str: A string showing the stack's contents from bottom to top.
        """
        return f"Stack({self._items})"

# --- Example Usage ---
if __name__ == "__main__":
    # Create a new stack for integers
    s = Stack[int]()

    # 1. Check if the new stack is empty
    print(f"Is the stack empty? {s.is_empty()}")  # Expected: True
    print(f"Initial stack: {s}")

    # 2. Push items onto the stack
    print("\nPushing 10, 20, then 30...")
    s.push(10)
    s.push(20)
    s.push(30)
    print(f"Stack after pushes: {s}")
    print(f"Current size: {s.size()}")            # Expected: 3
    print(f"Is the stack empty? {s.is_empty()}")  # Expected: False

    # 3. Peek at the top item
    print(f"\nPeeking at the top item: {s.peek()}") # Expected: 30
    print(f"Stack remains unchanged: {s}")
```

```python
    # 4. Pop an item from the stack
    popped_item = s.pop()
    print(f"\nPopped item: {popped_item}")        # Expected: 30
    print(f"Stack after one pop: {s}")
    print(f"New top item is: {s.peek()}")         # Expected: 20

    # 5. Pop all remaining items
    s.pop()
    s.pop()
    print(f"\nStack after popping all items: {s}")
    print(f"Is the stack empty now? {s.is_empty()}") # Expected: True

    # 6. Demonstrate error handling on an empty stack
    print("\nAttempting to pop from an empty stack...")
    try:
        s.pop()
    except IndexError as e:
        print(f"Successfully caught expected error: {e}")
```

Output:

```
PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai.py"
Is the stack empty? True
Initial stack: Stack([])

Pushing 10, 20, then 30...
Stack after pushes: Stack([10, 20, 30])
Current size: 3
Is the stack empty? False

Peeking at the top item: 30
```

```
Peeking at the top item: 30
Stack remains unchanged: Stack([10, 20, 30])

Popped item: 30
Stack after one pop: Stack([10, 20])
New top item is: 20

Stack after popping all items: Stack([])
Is the stack empty now? True
```

```
Popped item: 30
Stack after one pop: Stack([10, 20])
New top item is: 20

Stack after popping all items: Stack([])
Is the stack empty now? True

Attempting to pop from an empty stack...
Successfully caught expected error: pop from an empty stack
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

Observations:

Strengths

Generic Type Support

You used TypeVar and Generic[T] properly, which makes the stack flexible for any type (int, str, custom objects, etc.).

Encapsulation

_items is a private list, so the stack maintains control over its data.

Error Handling

pop() and peek() correctly raise IndexError when the stack is empty, which is consistent with Python's built-in list behavior.

Efficiency

All main operations (push, pop, peek, is_empty, __len__) are O(1) time complexity.

This makes the stack efficient for real use cases.

Readability and Documentation

Excellent use of docstrings: each method has a clear explanation, time complexity, return type, and raised exceptions.

Easy to understand and maintain.

String Representation

__str__ gives a clean representation of stack contents, making debugging easy (Stack([10, 20, 30])).

Example Usage Provided

The if __name__ == "__main__": block demonstrates all core features (push,pop, peek, error handling).

Makes it beginner-friendly and testable.

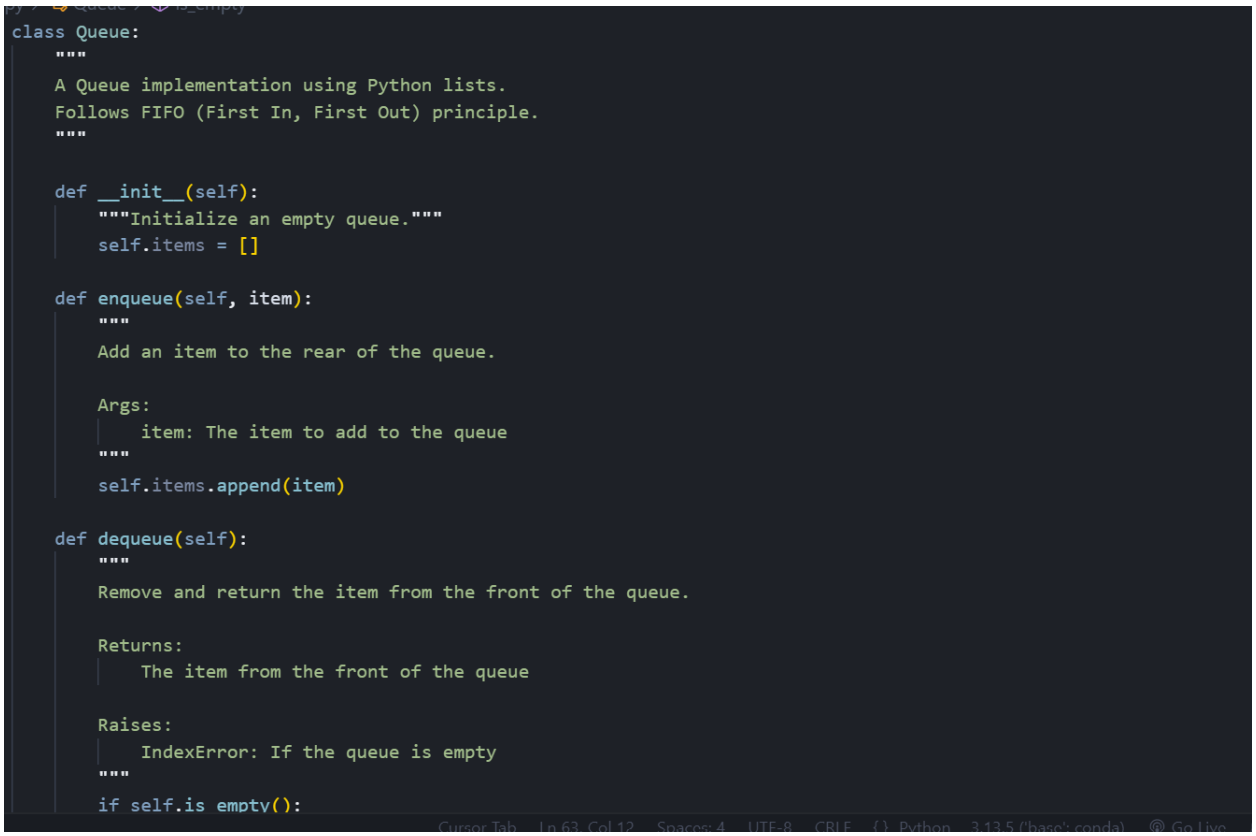
Task Description #2 –

Queue Implementation
Task: Use AI to implement a Queue using Python lists.
Sample Input Code:
class Queue:
pass
Expected Output:
• FIFO-based queue class with enqueue, dequeue, peek, and size methods.

PROMPT:

implement a Queue using Python lists

Code:



```
class Queue:
    """
    A Queue implementation using Python lists.
    Follows FIFO (First In, First Out) principle.
    """

    def __init__(self):
        """Initialize an empty queue."""
        self.items = []

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

        Args:
            item: The item to add to the queue
        """
        self.items.append(item)

    def dequeue(self):
        """
        Remove and return the item from the front of the queue.

        Returns:
            The item from the front of the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
```

```
class Queue:
    def dequeue(self):
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self.items.pop(0)

    def front(self):
        """
        Return the item at the front of the queue without removing it.

        Returns:
            The item at the front of the queue

        Raises:
            IndexError: If the queue is empty
        """
        if self.is_empty():
            raise IndexError("front called on empty queue")
        return self.items[0]

    def rear(self):
        """
        Return the item at the rear of the queue without removing it.

        Returns:
            The item at the rear of the queue

        Raises:
            IndexError: If the queue is empty
        """
```

Cursor Tab Ln 63, Col 12 Spaces: 4 UTF-8 CRLF {} Python 3.13.5 ('base': conda) Go Live

```
class Queue:
    def rear(self):
        if self.is_empty():
            raise IndexError("rear called on empty queue")
        return self.items[-1]

    def is_empty(self):
        """
        Check if the queue is empty.

        Returns:
            bool: True if the queue is empty, False otherwise
        """
        return len(self.items) == 0

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

        Returns:
            int: The number of items in the queue
        """
        return len(self.items)

    def clear(self):
        """Remove all items from the queue."""
        self.items = []

    def __str__(self):
        """
```

Cursor Tab Ln 63, Col 12 Spaces: 4 UTF-8 CRLF {} Python 3.13.5 ('base': conda) Go Live

```
class Queue:
    def __str__(self):
        Return a string representation of the queue.

        Returns:
            str: String representation showing items from front to rear
        """
        return f"Queue({self.items})"

    def __repr__(self):
        """Return a detailed string representation of the queue."""
        return f"Queue(items={self.items})"

    def __len__(self):
        """Return the length of the queue."""
        return len(self.items)


# Example usage and demonstration
if __name__ == "__main__":
    # Create a new queue
    queue = Queue()

    print("=== Queue Implementation Demo ===")
    print(f"Initial queue: {queue}")
    print(f"Is empty: {queue.is_empty()}")
    print(f"Size: {queue.size()}")

    # Add items to the queue
```

Cursor Tab Ln 63, Col 12 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 ('base': conda) Go Live

```
    # Add items to the queue
    print("\n--- Adding items ---")
    queue.enqueue("first")
    queue.enqueue("second")
    queue.enqueue("third")
    print(f"After enqueuing 'first', 'second', 'third': {queue}")
    print(f"Size: {queue.size()}")
    print(f"Front item: {queue.front()}")
    print(f"Rear item: {queue.rear()}")

    # Remove items from the queue
    print("\n--- Removing items ---")
    item1 = queue.dequeue()
    print(f"Dequeued: {item1}")
    print(f"Queue after dequeue: {queue}")

    item2 = queue.dequeue()
    print(f"Dequeued: {item2}")
    print(f"Queue after dequeue: {queue}")

    # Add more items
    print("\n--- Adding more items ---")
    queue.enqueue("fourth")
    queue.enqueue("fifth")
    print(f"After adding 'fourth', 'fifth': {queue}")

    # Show remaining operations
    print(f"Front: {queue.front()}")
    print(f"Rear: {queue.rear()}")
```

Cursor Tab Ln 63, Col 12 Spaces: 4 UTF-8 CRLF { } Python 3.13.5 ('base': conda) Go Live

```
    print(f"Size: {queue.size()}")

    # Clear the queue
    print("\n--- Clearing queue ---")
    queue.clear()
    print(f"After clearing: {queue}")
    print(f"Is empty: {queue.is_empty()}")

    # Demonstrate error handling
    print("\n--- Error handling demo ---")
    try:
        queue.dequeue()
    except IndexError as e:
        print(f"Error when dequeuing from empty queue: {e}")

    try:
        queue.front()
    except IndexError as e:
        print(f"Error when accessing front of empty queue: {e}")

```

Output:

```
--- Error handling demo ---
Error when dequeuing from empty queue: dequeue from empty queue
Error when accessing front of empty queue: front called on empty queue
PS C:\Users\DELL\Desktop> & C:/Users/DELL/AppData/Local/anaconda3/python.exe c:/Users/DELL/Desktop/ai_1.py
=== Queue Implementation Demo ===
Initial queue: Queue([])
Is empty: True
Size: 0

Ctrl+K to generate a command
```

```
--- Adding items ---
After enqueuing 'first', 'second', 'third': Queue(['first', 'second', 'third'])
Size: 3
Front item: first
Rear item: third

--- Removing items ---
Dequeued: first
```

```
--- Removing items ---
Dequeued: first
Queue after dequeue: Queue(['second', 'third'])
Dequeued: second
Queue after dequeue: Queue(['third'])

--- Adding more items ---
After adding 'fourth', 'fifth': Queue(['third', 'fourth', 'fifth'])
Front: third

Ctrl+K to generate a command
```

```
--- Adding more items ---
After adding 'fourth', 'fifth': Queue(['third', 'fourth', 'fifth'])
Front: third
Rear: fifth
Size: 3

--- Clearing queue ---
After clearing: Queue([])
Is empty: True
```

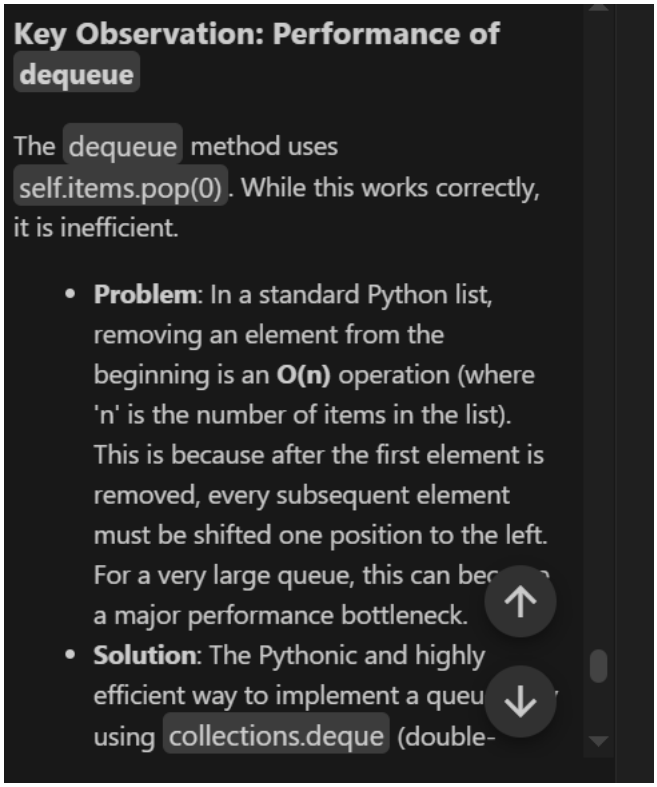
Observation:

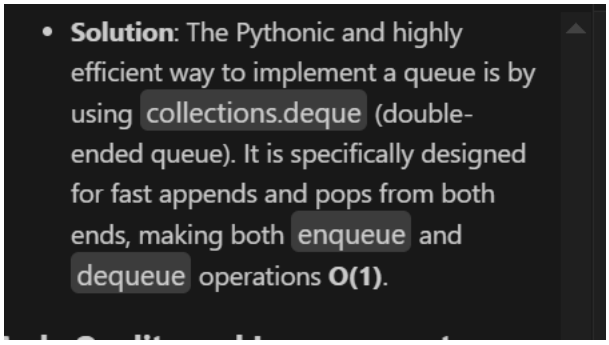
The class correctly implements the FIFO (First-In, First-Out) logic of a queue. The methods are well-named, and the inclusion of dunder methods like `__len__` and `__str__` makes the class intuitive to use. The example usage block is excellent for demonstrating functionality and verifying correctness.

The most critical observation relates to the choice of a Python `list` as the internal data structure, which has significant performance implications for the `dequeue` operation.

Key Observation: Performance of `dequeue`

The `dequeue` method uses





Task Description #3 –

Linked List
Task: Use AI to generate a Singly Linked List with insert and display methods.
Sample Input Code:
class Node:
pass
class LinkedList:
pass
Expected Output:

• A working linked list implementation with clear method documentation.

Code:

ai_4.py > ...

```python
from typing import TypeVar, Generic, Optional

# Define a generic type 'T' for the data within the nodes.
T = TypeVar('T')

class Node(Generic[T]):
    """
    A single node in a Singly Linked List.

    Attributes:
        data (T): The data stored within the node.
        next (Optional[Node[T]]): A reference to the next node in the list.
    """
    def __init__(self, data: T):
        self.data: T = data
        self.next: Optional['Node[T]'] = None

class SinglyLinkedList(Generic[T]):
    """
    A Singly Linked List data structure.

    This class manages a sequence of nodes, providing methods to insert
    new nodes and display the contents of the list.

    Attributes:
        head (Optional[Node[T]]): The first node in the linked list.
    """

    def __init__(self) -> None:
        """Initializes an empty linked list."""
```

ai_4.py > ...

```python
class SinglyLinkedList(Generic[T]):

    def __init__(self) -> None:
        """Initializes an empty linked list."""
        self.head: Optional[Node[T]] = None

    def insert(self, data: T) -> None:
        """
        Inserts a new node with the given data at the end of the list.

        This method traverses the list to find the last node and appends
        the new node there. If the list is empty, the new node becomes the head.
        Time complexity: O(n) where n is the number of nodes.

        Args:
            data (T): The data for the new node.
        """
        new_node = Node(data)
        # If the list is empty, the new node becomes the head.
        if self.head is None:
            self.head = new_node
            return

        # Otherwise, traverse to the end of the list.
        last_node = self.head
        while last_node.next:
            last_node = last_node.next

        # Append the new node at the end.
        last_node.next = new_node
```

Ln 103, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.13.5 (base) Go Live

```python
        # Append the new node at the end.
        last_node.next = new_node

    def display(self) -> None:
        """
        Prints the data of each node in the linked list in order.

        It traverses from the head to the end of the list, printing the
        data of each node, formatted to show the list structure.
        """
        nodes = []
        current_node = self.head
        while current_node:
            # Represent string data with quotes for clarity
            if isinstance(current_node.data, str):
                nodes.append(f"'{current_node.data}'")
            else:
                nodes.append(str(current_node.data))
            current_node = current_node.next

        if not nodes:
            print("LinkedList: [Empty]")
        else:
            print("LinkedList: " + " -> ".join(nodes) + " -> None")

# --- Example Usage ---
if __name__ == "__main__":
    # Create a new singly linked list
```

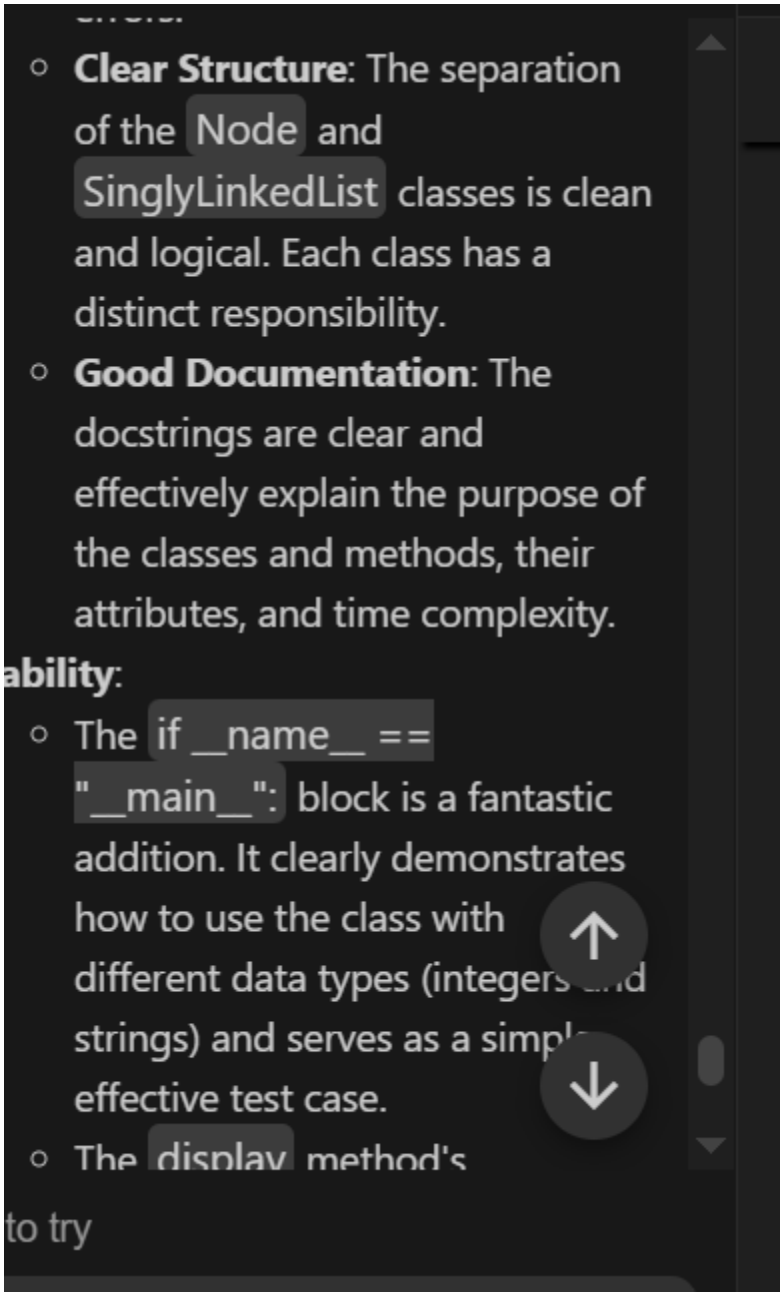
```python
# --- Example Usage ---
if __name__ == "__main__":
    # Create a new singly linked list
    sll = SinglyLinkedList[int]()

    # 1. Display the empty list
    print("Initial list:")
    sll.display()  # Expected: LinkedList: [Empty]

    # 2. Insert some integer items
    print("\nInserting 10, 20, and 30...")
    sll.insert(10)
    sll.insert(20)
    sll.insert(30)
    sll.display()  # Expected: LinkedList: 10 -> 20 -> 30 -> None

    # 3. Create and display a list of strings
    sll_str = SinglyLinkedList[str]()
    print("\nCreating a new list for strings...")
    sll_str.insert("apple")
    sll_str.insert("banana")
    sll_str.insert("cherry")
    sll_str.display() # Expected: LinkedList: 'apple' -> 'banana' -> 'cherry' -> None
```

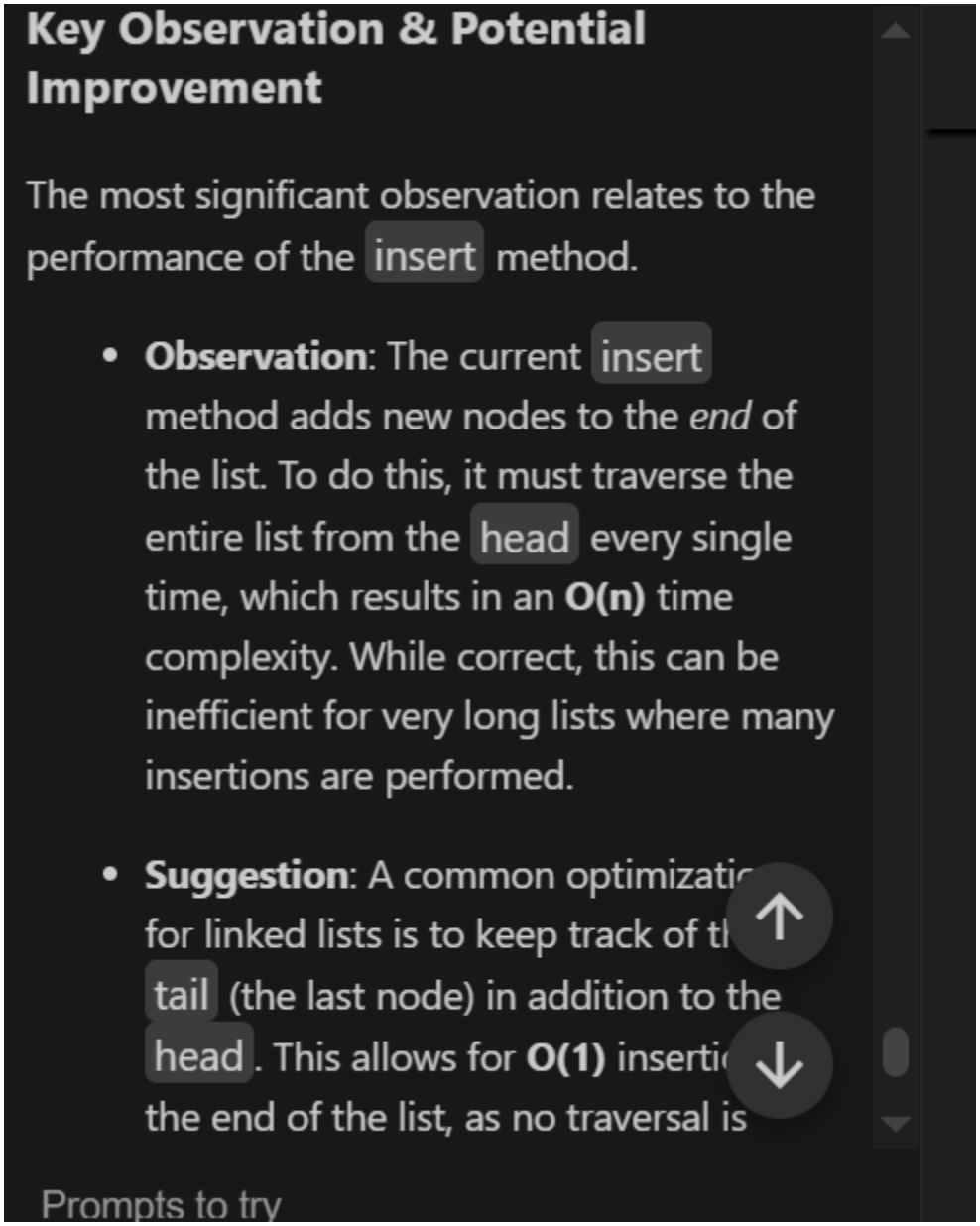
Output:

Observations:

Positive Observations (What's Done Well)

1. **Correctness and Logic**: The implementation correctly models a singly linked list. The `insert` method correctly appends nodes to the end, and the `display` method accurately traverses the list from head to tail.
2. **Code Quality and Readability**:
    - **Excellent Type Hinting**: The use of `typing` with `TypeVar`, `Generic`, and `Optional` is a best practice. It makes the code self-documenting, robust, a allows static analysis tools to catch potential type-related errors.
    - **Clear Structure**: The separation


Clear Structure: The separation of the Node and SinglyLinkedList classes is clean and logical. Each class has a distinct responsibility.
Good Documentation: The docstrings are clear and effectively explain the purpose of the classes and methods, their attributes, and time complexity.
ability:
The if __name__ == "__main__": block is a fantastic addition. It clearly demonstrates how to use the class with different data types (integers
strings) and serves as a simp
effective test case.
The display method's
to try



Task Description #4 –

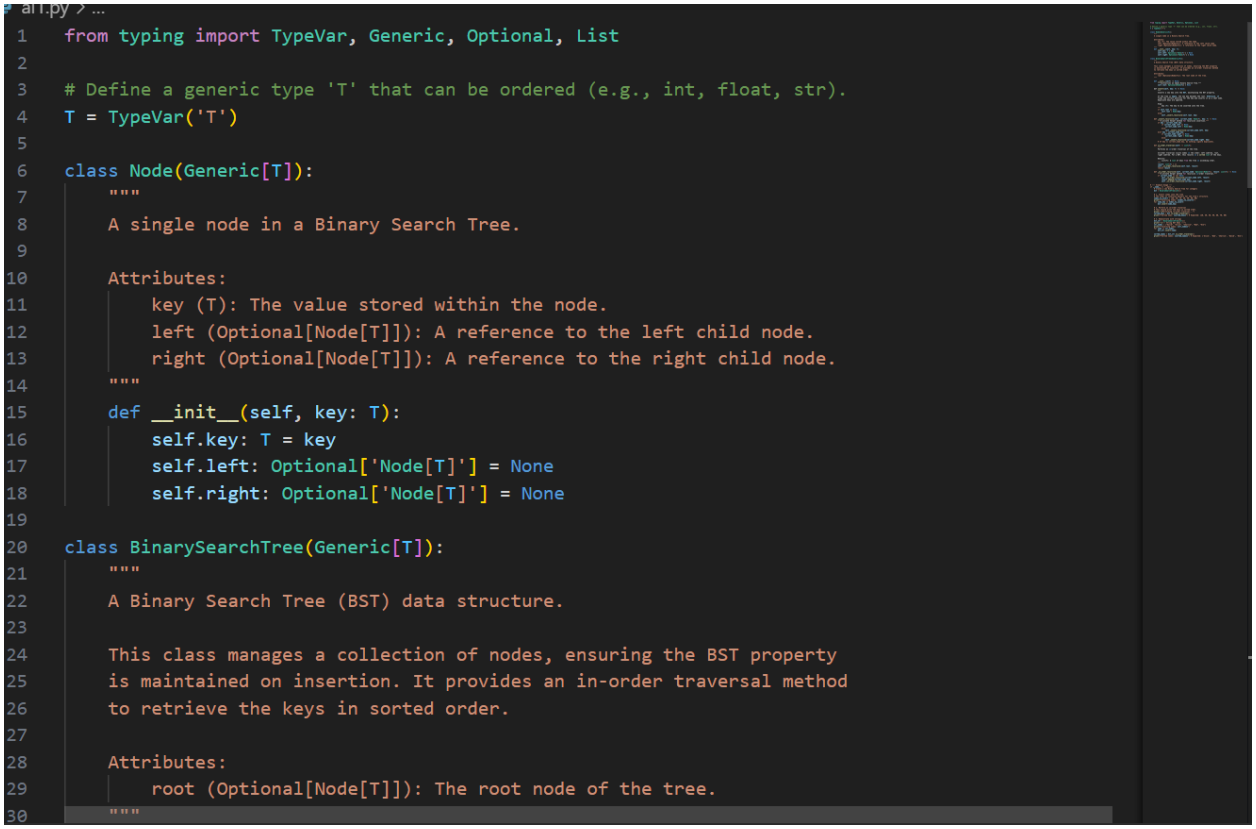
Binary Search Tree (BST)
Task: Use AI to create a BST with insert and in-order traversal methods.
Sample Input Code:
class BST:

pass
Expected Output:
• BST implementation with recursive insert and traversal methods.

PROMPT:

create a BST with insert and in-order traversal methods.

Code:



```python
    def __init__(self) -> None:
        """Initializes an empty Binary Search Tree."""
        self.root: Optional[Node[T]] = None

    def insert(self, key: T) -> None:
        """
        Inserts a new key into the BST, maintaining the BST property.

        If the tree is empty, the new key becomes the root. Otherwise, it
        finds the correct position for the key and inserts it as a leaf node.
        Duplicate keys are ignored.

        Args:
            key (T): The key to be inserted into the tree.
        """
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert_recursive(self.root, key)

    def _insert_recursive(self, current_node: Node[T], key: T) -> None:
        """A private helper method for recursive insertion."""
        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)
            else:
                self._insert_recursive(current_node.right, key)
        # If key == current_node.key, do nothing (ignore duplicate).

    def in_order_traversal(self) -> List[T]:
        """
        Performs an in-order traversal of the tree.

        In-order traversal visits nodes in the order: left subtree, root,
        right subtree. For a BST, this results in a sorted list of the keys.

        Returns:
            List[T]: A list of keys from the tree in ascending order.
        """
        result: List[T] = []
        self._in_order_recursive(self.root, result)
        return result

    def _in_order_recursive(self, current_node: Optional[Node[T]], result: List[T]) -> None:
        """A private helper method for recursive in-order traversal."""
        if current_node is not None:
            self._in_order_recursive(current_node.left, result)
            result.append(current_node.key)
            self._in_order_recursive(current_node.right, result)
```

```python
# --- Example Usage ---
if __name__ == "__main__":
    # Create a new Binary Search Tree for integers
    bst = BinarySearchTree[int]()

    # 1. Insert items into the tree
    # The order of insertion matters for the tree's structure.
    nodes_to_insert = [50, 30, 70, 20, 40, 60, 80]
    print(f"Inserting nodes: {nodes_to_insert}")
    for node_key in nodes_to_insert:
        bst.insert(node_key)

    # 2. Perform an in-order traversal
    # This should print the keys in sorted order.
    print("\nPerforming in-order traversal...")
    sorted_keys = bst.in_order_traversal()
    print(f"Sorted keys: {sorted_keys}") # Expected: [20, 30, 40, 50, 60, 70, 80]

    # 3. Demonstrate with strings
    bst_str = BinarySearchTree[str]()
    print("\n--- String BST Demo ---")
    str_nodes = ["David", "Alice", "Charlie", "Bob", "Eve"]
    print(f"Inserting nodes: {str_nodes}")
    for name in str_nodes:
        bst_str.insert(name)

    sorted_names = bst_str.in_order_traversal()
    print(f"Sorted names: {sorted_names}") # Expected: ['Alice', 'Bob', 'Charlie', 'David', 'Eve']

```

Output:

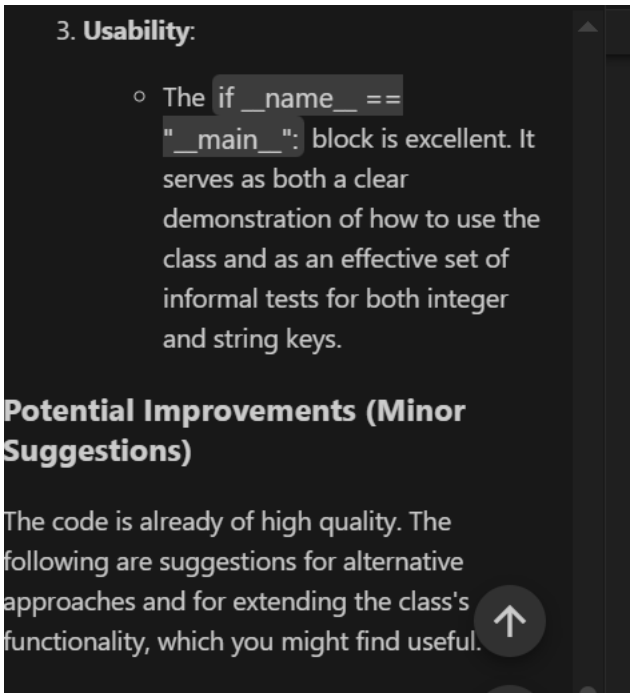
```
PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai1.py"
Inserting nodes: [50, 30, 70, 20, 40, 60, 80]

Performing in-order traversal...
Sorted keys: [20, 30, 40, 50, 60, 70, 80]

--- String BST Demo ---
Inserting nodes: ['David', 'Alice', 'Charlie', 'Bob', 'Eve']
Sorted names: ['Alice', 'Bob', 'Charlie', 'David', 'Eve']
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

Observations:

Positive Observations (What's Done Well)

1. **Correctness and Logic**: The implementation correctly adheres to the rules of a Binary Search Tree.
    - The `insert` method correctly places smaller keys to the left and larger keys to the right.
    - The `in_order_traversal` (Left-Root-Right) correctly produces a sorted list of the keys, which is a fundamental property and a great way to verify the tree's integrity.
    - The decision to ignore duplicate keys is a common and valid approach.

2. **Code Quality and Readability**:
    - **Excellent Type Hinting**: The use of `typing` with `TypeVar`, `Generic`, `Optional`, and `List` is superb. It makes the code robust, self-documenting, and allows static analysis tools to catch errors.
    - **Clear Structure**: The separation of concerns between the `Node` class and the `BinarySearchTree` class is clean. The use of private helper methods (`_insert_recursive`, `_in_order_recursive`) for th... recursive logic is a standard and highly readable pattern.
    - **Good Documentation**: The



Task Description #5 –

Hash Table
Task: Use AI to implement a hash table with basic insert, search, and delete

methods.
Sample Input Code:
class HashTable:
pass
Expected Output:
• Collision handling using chaining, with well-commented methods.

PROMPT:

implement a hash table with basic insert, search, and delete

methods.

Code:

ai_4.py > ...

```python
from typing import TypeVar, Generic, List, Tuple, Optional, Any

# Define generic types for keys and values.
KT = TypeVar('KT')
VT = TypeVar('VT')

class HashTable(Generic[KT, VT]):
    """
    A basic implementation of a Hash Table (also known as a hash map).

    This implementation uses chaining with Python lists to handle hash collisions.
    It supports basic operations: insert, search, and delete.

    Attributes:
        size (int): The number of buckets in the hash table.
        table (List[List[Tuple[KT, VT]]]): The internal table, where each
                                           element is a "bucket" (a list
                                           of key-value pairs).
    """

    def __init__(self, size: int = 101) -> None:
        """
        Initializes the hash table.

        Args:
            size (int): The size of the hash table. A prime number is
                        often recommended to help distribute keys more uniformly.
        """
        if size <= 0:
            raise ValueError("Hash table size must be positive.")
```

```python
    def __init__(self, size: int = 101) -> None:
        if size <= 0:
            raise ValueError("Hash table size must be positive.")
        self.size: int = size
        self.table: List[List[Tuple[KT, VT]]] = [[] for _ in range(self.size)]

    def _hash(self, key: KT) -> int:
        """
        Computes the hash of a key and maps it to a bucket index.

        Args:
            key (KT): The key to hash.

        Returns:
            int: The index of the bucket for the given key.
        """
        return hash(key) % self.size

    def insert(self, key: KT, value: VT) -> None:
        """
        Inserts a key-value pair into the hash table.

        If the key already exists, its value is updated. Otherwise, the new
        key-value pair is added.

        Args:
            key (KT): The key.
            value (VT): The value associated with the key.
        """
```

```
        Args:
            key (KT): The key.
            value (VT): The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]

        # Check if the key already exists and update it
        for i, (existing_key, _) in enumerate(bucket):
            if existing_key == key:
                bucket[i] = (key, value)
                return

        # If the key is new, append the pair to the bucket
        bucket.append((key, value))

    def search(self, key: KT) -> Optional[VT]:
        """
        Searches for a key and returns its corresponding value.

        Args:
            key (KT): The key to search for.

        Returns:
            Optional[VT]: The value associated with the key, or None if the
                          key is not found.
        """
        index = self._hash(key)
        bucket = self.table[index]

        for existing_key, value in bucket:
            if existing_key == key:
                return value

        return None

    def delete(self, key: KT) -> bool:
        """
        Deletes a key-value pair from the hash table.

        Args:
            key (KT): The key to delete.

        Returns:
            bool: True if the key was found and deleted, False otherwise.
        """
        index = self._hash(key)
        bucket = self.table[index]

        for i, (existing_key, _) in enumerate(bucket):
            if existing_key == key:
                bucket.pop(i)
                return True
```

```
        return False

    def __str__(self) -> str:
        """Provides a string representation of the hash table."""
        elements = []
        for i, bucket in enumerate(self.table):
            if bucket:
                elements.append(f"  Bucket {i}: {bucket}")
        if not elements:
            return "HashTable(empty)"
        return "HashTable:\n" + "\n".join(elements)

# --- Example Usage ---
if __name__ == "__main__":
    # Create a new hash table
    ht = HashTable[str, int](size=10)

    print("--- Inserting items ---")
    ht.insert("apple", 5)
    ht.insert("banana", 12)
    ht.insert("orange", 8)
    ht.insert("grape", 25) # This might collide with 'apple' depending on hash
    print(ht)

    print("\n--- Searching for items ---")
    print(f"Price of 'apple': {ht.search('apple')}")     # Expected: 5
    print(f"Price of 'banana': {ht.search('banana')}")   # Expected: 12
    print(f"Price of 'mango': {ht.search('mango')}")     # Expected: None
```

```
    print("\n--- Updating an item ---")
    ht.insert("apple", 10) # Update the value for 'apple'
    print(f"New price of 'apple': {ht.search('apple')}") # Expected: 10
    print(ht)

    print("\n--- Deleting items ---")
    deleted_banana = ht.delete("banana")
    print(f"Was 'banana' deleted? {deleted_banana}")     # Expected: True
    deleted_mango = ht.delete("mango")
    print(f"Was 'mango' deleted? {deleted_mango}")       # Expected: False
    print("\nHash table after deletions:")
    print(ht)
```

Output:

```
PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai_4.py"
--- Inserting items ---
HashTable:
  Bucket 2: [('apple', 5), ('banana', 12), ('grape', 25)]
  Bucket 7: [('orange', 8)]

--- Searching for items ---
Price of 'apple': 5
Price of 'banana': 12
Price of 'mango': None
```

```
--- Updating an item ---
New price of 'apple': 10
HashTable:
  Bucket 2: [('apple', 10), ('banana', 12), ('grape', 25)]
  Bucket 7: [('orange', 8)]

--- Deleting items ---
Was 'banana' deleted? True
Was 'mango' deleted? False
```

```
--- Deleting items ---
Was 'banana' deleted? True
Was 'mango' deleted? False

Hash table after deletions:
HashTable:
  Bucket 2: [('apple', 10), ('grape', 25)]
  Bucket 7: [('orange', 8)]
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

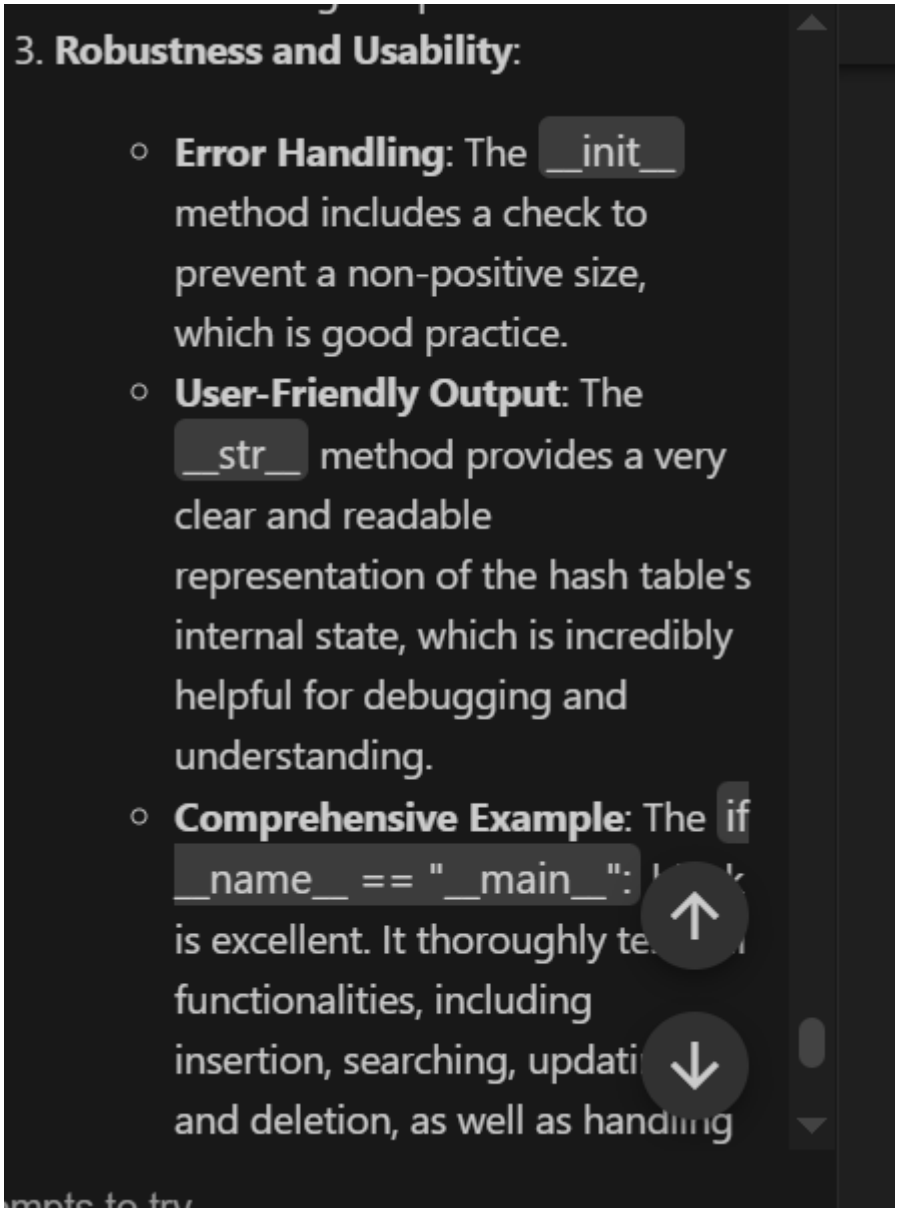
Observations:

Positive Observations (What's Done Well)

1. **Correctness and Logic**: The implementation is spot-on.
    - The `_hash` function correctly uses Python's built-in `hash()` and the modulo operator to map keys to bucket indices.
    - **Collision Handling**: The use of lists as buckets (chaining) is a standard and effective way to handle hash collisions. The `insert`, `search`, and `delete` methods all correctly iterate through the bucket to handle multiple items at the same i
    - **Update on Insert**: The `inser`

2. **Code Quality and Readability**:

- **Excellent Type Hinting**: The use of `typing` with `TypeVar`, `Generic`, `Optional`, `Tuple`, and `List` is superb. This makes the class generic and type-safe, allowing it to function as a map for any key-value types.
- **Clear Structure**: The methods are well-defined and have a single responsibility. The use of a private `_hash` method is a good design choice.
- **Good Documentation**: The docstrings are clear, explain the purpose of the class, its methods, arguments, and re values. The comment about



Task Description #6 –

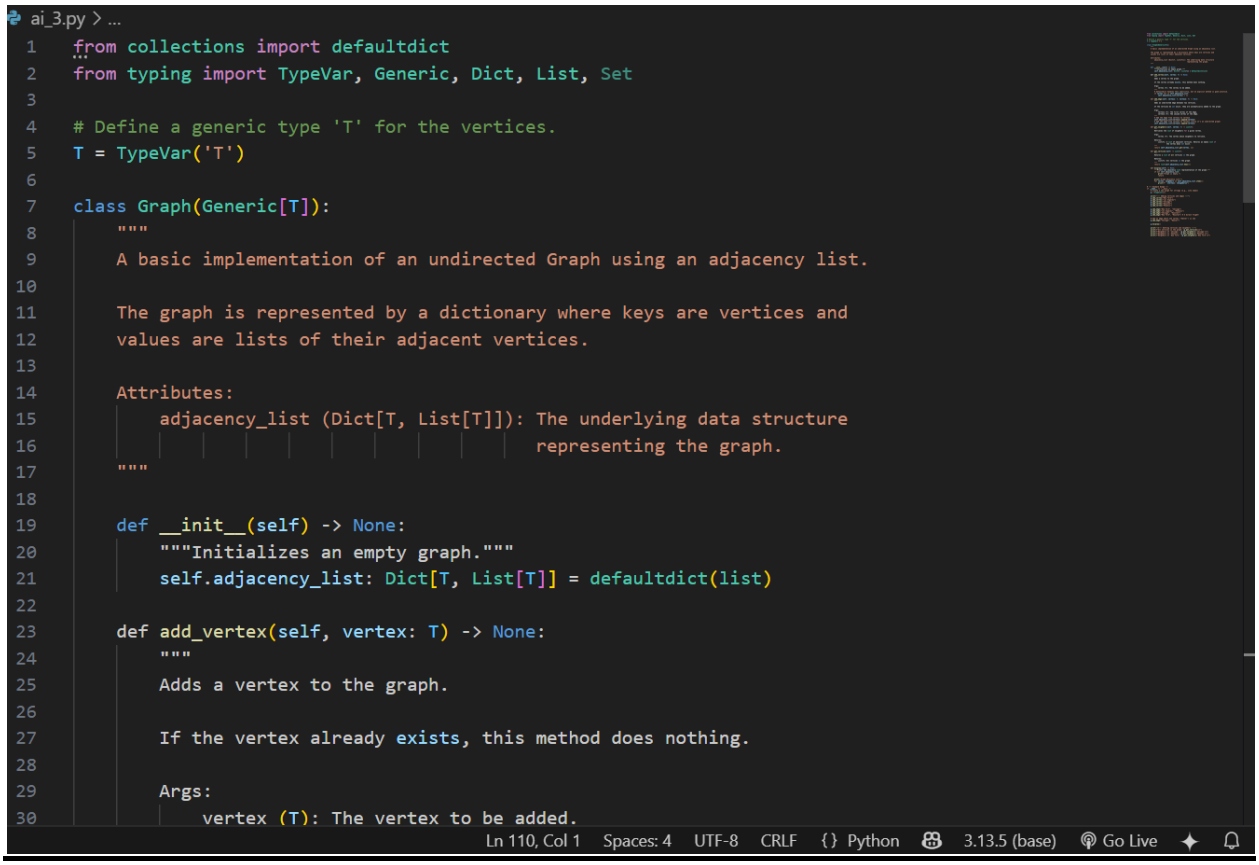
Graph Representation
Task: Use AI to implement a graph using an adjacency list.

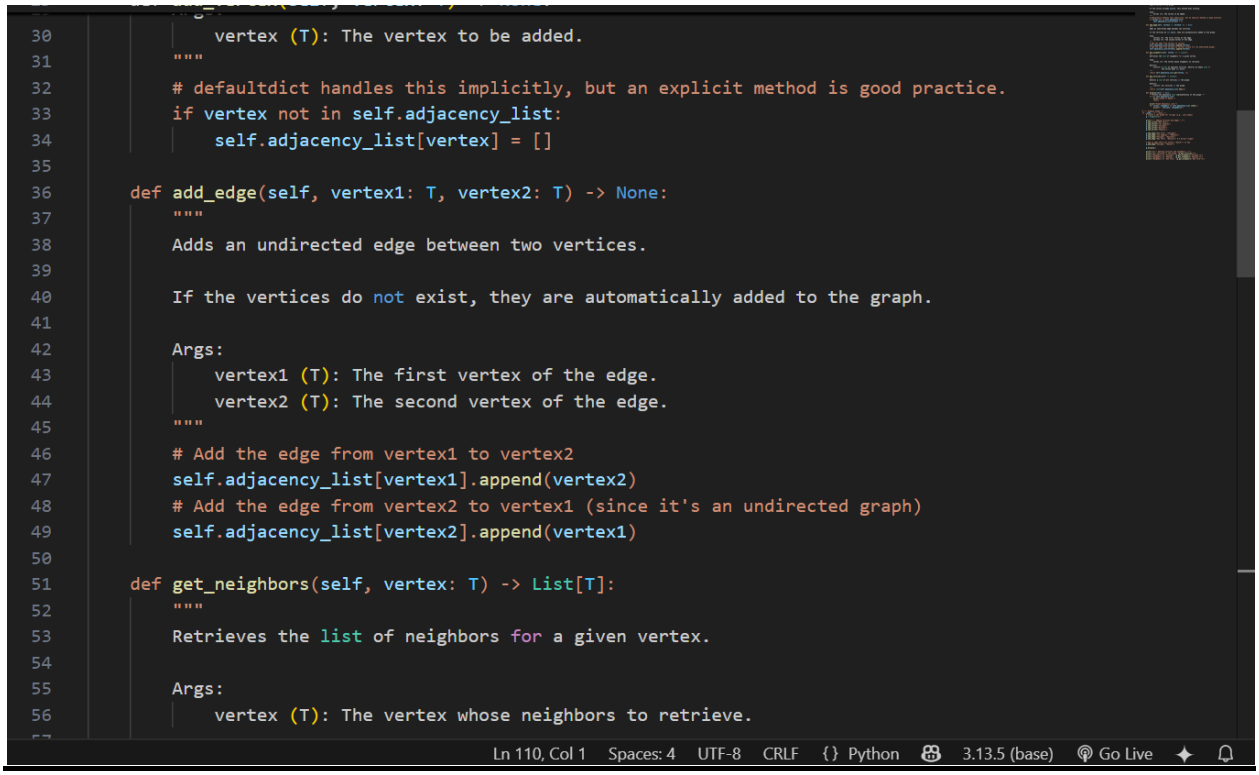
Sample Input Code:
class Graph:
pass
Expected Output:
• Graph with methods to add vertices, add edges, and display connections

PROMPT:

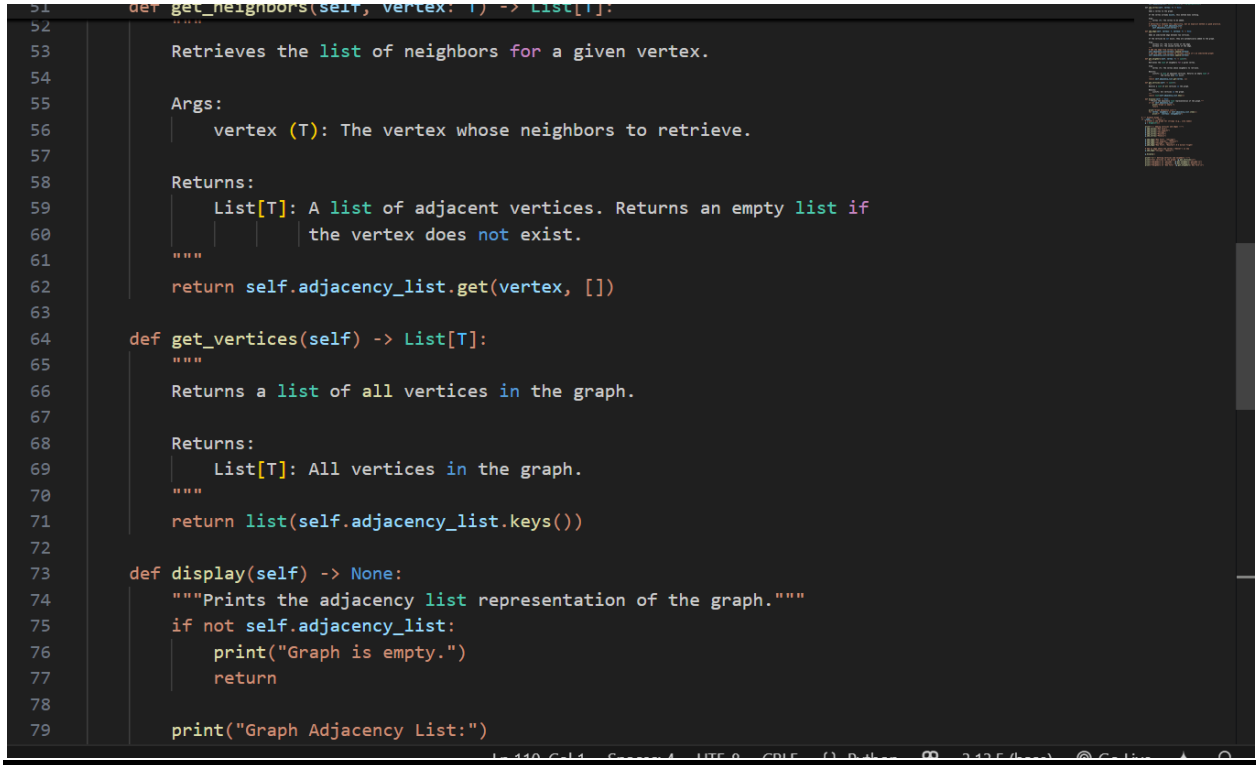
implement a graph using an adjacency list.

Code:





```
        vertex (T): The vertex to be added.
        """
        # defaultdict handles this implicitly, but an explicit method is good practice.
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = []

    def add_edge(self, vertex1: T, vertex2: T) -> None:
        """
        Adds an undirected edge between two vertices.

        If the vertices do not exist, they are automatically added to the graph.

        Args:
            vertex1 (T): The first vertex of the edge.
            vertex2 (T): The second vertex of the edge.
        """
        # Add the edge from vertex1 to vertex2
        self.adjacency_list[vertex1].append(vertex2)
        # Add the edge from vertex2 to vertex1 (since it's an undirected graph)
        self.adjacency_list[vertex2].append(vertex1)

    def get_neighbors(self, vertex: T) -> List[T]:
        """
        Retrieves the list of neighbors for a given vertex.

        Args:
            vertex (T): The vertex whose neighbors to retrieve.
```

```
    def get_neighbors(self, vertex: T) -> List[T]:
        """
        Retrieves the list of neighbors for a given vertex.

        Args:
            vertex (T): The vertex whose neighbors to retrieve.

        Returns:
            List[T]: A list of adjacent vertices. Returns an empty list if
                     the vertex does not exist.
        """
        return self.adjacency_list.get(vertex, [])

    def get_vertices(self) -> List[T]:
        """
        Returns a list of all vertices in the graph.

        Returns:
            List[T]: All vertices in the graph.
        """
        return list(self.adjacency_list.keys())

    def display(self) -> None:
        """Prints the adjacency list representation of the graph."""
        if not self.adjacency_list:
            print("Graph is empty.")
            return

        print("Graph Adjacency List:")
```

```
        for vertex, neighbors in self.adjacency_list.items():
            print(f"  {vertex}: {neighbors}")

# --- Example Usage ---
if __name__ == "__main__":
    # Create a new graph for strings (e.g., city names)
    g = Graph[str]()

    print("--- Adding vertices and edges ---")
    g.add_vertex("New York")
    g.add_vertex("Los Angeles")
    g.add_vertex("Chicago")
    g.add_vertex("Houston")
    g.add_vertex("Phoenix")

    g.add_edge("New York", "Chicago")
    g.add_edge("Los Angeles", "Phoenix")
    g.add_edge("Chicago", "Houston")
    g.add_edge("New York", "Houston") # A direct flight!

    # Add an edge where one vertex ('Denver') is new
    g.add_edge("Chicago", "Denver")

    g.display()

    print("\n--- Getting vertices and neighbors ---")
    print(f"All vertices in the graph: {g.get_vertices()}")
```

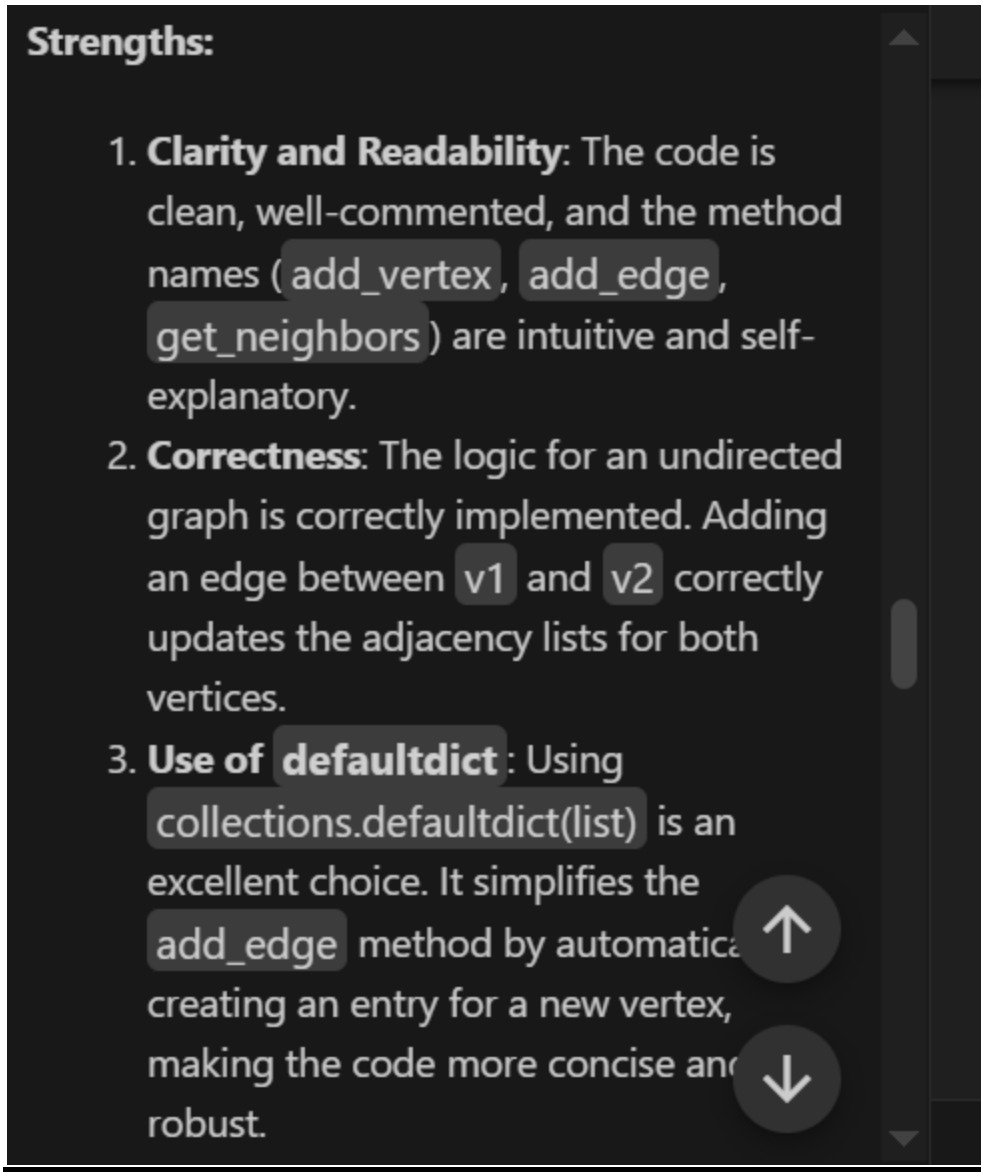
```
    g.display()

    print("\n--- Getting vertices and neighbors ---")
    print(f"All vertices in the graph: {g.get_vertices()}")
    print(f"Neighbors of 'Chicago': {g.get_neighbors('Chicago')}")
    print(f"Neighbors of 'Phoenix': {g.get_neighbors('Phoenix')}")
    print(f"Neighbors of 'New York': {g.get_neighbors('New York')}")
```

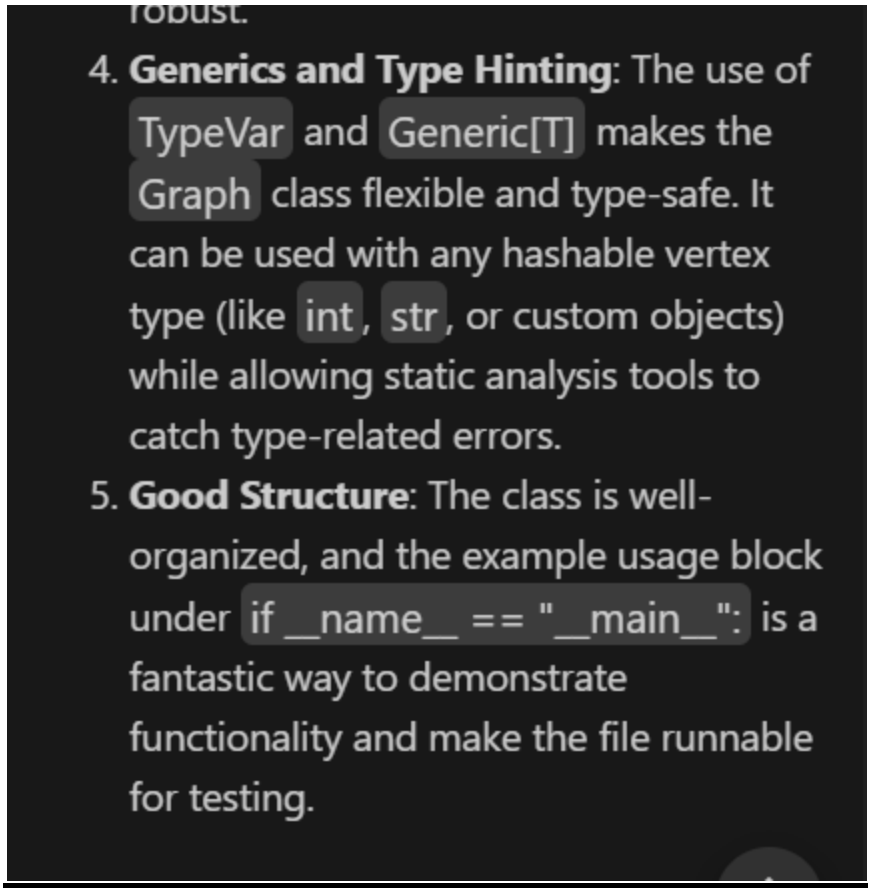
Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai_3.py"
--- Adding vertices and edges ---
Graph Adjacency List:
  New York: ['Chicago', 'Houston']
  Los Angeles: ['Phoenix']
  Chicago: ['New York', 'Houston', 'Denver']
  Houston: ['Chicago', 'New York']
  Phoenix: ['Los Angeles']
  Denver: ['Chicago']

Ln 110, Col 1  Spaces: 4  UTF-8  CRLF  {} Python  3.13.5 (base)  Go Live
```

```
--- Getting vertices and neighbors ---
All vertices in the graph: ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix', 'Denver']
Neighbors of 'Chicago': ['New York', 'Houston', 'Denver']
Neighbors of 'Phoenix': ['Los Angeles']
Neighbors of 'New York': ['Chicago', 'Houston']
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

Observations:





.

Task Description #7 –

Priority Queue
Task: Use AI to implement a priority queue using Python's heapq module.
Sample Input Code:
class PriorityQueue:
pass
Expected Output:
• Implementation with enqueue (priority), dequeue (highest priority), and display methods.

Prompt:

implement a priority queue using Python's heapq module.

Code:

ai_4.py > ...

```python
import heapq
from typing import TypeVar, Generic, List, Tuple

# Define a generic type 'T' for the items in the queue.
T = TypeVar('T')

class PriorityQueue(Generic[T]):
    """
    A Priority Queue implementation using Python's heapq module (min-heap).

    Items are stored along with their priority. The item with the lowest
    priority value is considered the highest priority and will be popped first.

    Attributes:
        _heap (List[Tuple[int, T]]): A private list that stores tuples of
                                     (priority, item). This list is maintained
                                     as a min-heap by the heapq module.
    """

    def __init__(self) -> None:
        """Initializes an empty Priority Queue."""
        self._heap: List[Tuple[int, T]] = []

    def push(self, item: T, priority: int) -> None:
        """
        Adds an item to the queue with a specified priority.

        The lower the priority number, the higher the priority.

        Args:
            item (T): The item to be added to the queue.
            priority (int): The priority of the item.
        """
        # heapq pushes a tuple (priority, item) onto the heap.
        # It will use the first element of the tuple (priority) for sorting.
        heapq.heappush(self._heap, (priority, item))

    def pop(self) -> T:
        """
        Removes and returns the item with the highest priority (lowest priority value).

        Returns:
            T: The item with the highest priority.

        Raises:
            IndexError: If pop is called on an empty queue.
        """
        if self.is_empty():
            raise IndexError("pop from an empty priority queue")

        # heappop removes and returns the smallest item from the heap.
        # We return the second element of the tuple, which is the actual item.
        priority, item = heapq.heappop(self._heap)
        return item

    def is_empty(self) -> bool:
        """
```

```
        Checks if the priority queue is empty.

        Returns:
            bool: True if the queue is empty, False otherwise.
        """
        return not self._heap

    def __len__(self) -> int:
        """
        Returns the number of items in the priority queue.
        Allows the use of the built-in len() function.

        Returns:
            int: The number of items in the queue.
        """
        return len(self._heap)

# --- Example Usage ---
if __name__ == "__main__":
    # Create a new priority queue to manage tasks
    # The items will be strings (task descriptions)
    task_queue = PriorityQueue[str]()

    # 1. Check if the new queue is empty
    print(f"Is the queue empty? {task_queue.is_empty()}")  # Expected: True

    # 2. Push tasks with different priorities
    # Priority 1 is the highest, 5 is the lowest.
```

```
    print("\nAdding tasks with priorities...")
    task_queue.push("Write report", 3)
    task_queue.push("Answer urgent emails", 1)
    task_queue.push("Fix critical bug", 1) # Same priority as another task
    task_queue.push("Schedule team meeting", 4)
    task_queue.push("Refill coffee", 5)

    print(f"Current queue size: {len(task_queue)}")
    print(f"Is the queue empty? {task_queue.is_empty()}")  # Expected: False

    # 3. Pop tasks from the queue
    # They should come out in order of their priority number (lowest first).
    print("\nProcessing tasks in order of priority:")

    # The order of items with the same priority (like the two priority 1 tasks)
    # is not guaranteed by heapq, but they will both come out before priority 3.
    task1 = task_queue.pop()
    print(f"Processing (Priority 1): '{task1}'")

    task2 = task_queue.pop()
    print(f"Processing (Priority 1): '{task2}'")

    task3 = task_queue.pop()
    print(f"Processing (Priority 3): '{task3}'")

    print(f"\nRemaining tasks in queue: {len(task_queue)}")

    # 4. Pop all remaining tasks
    while not task_queue.is_empty():
```

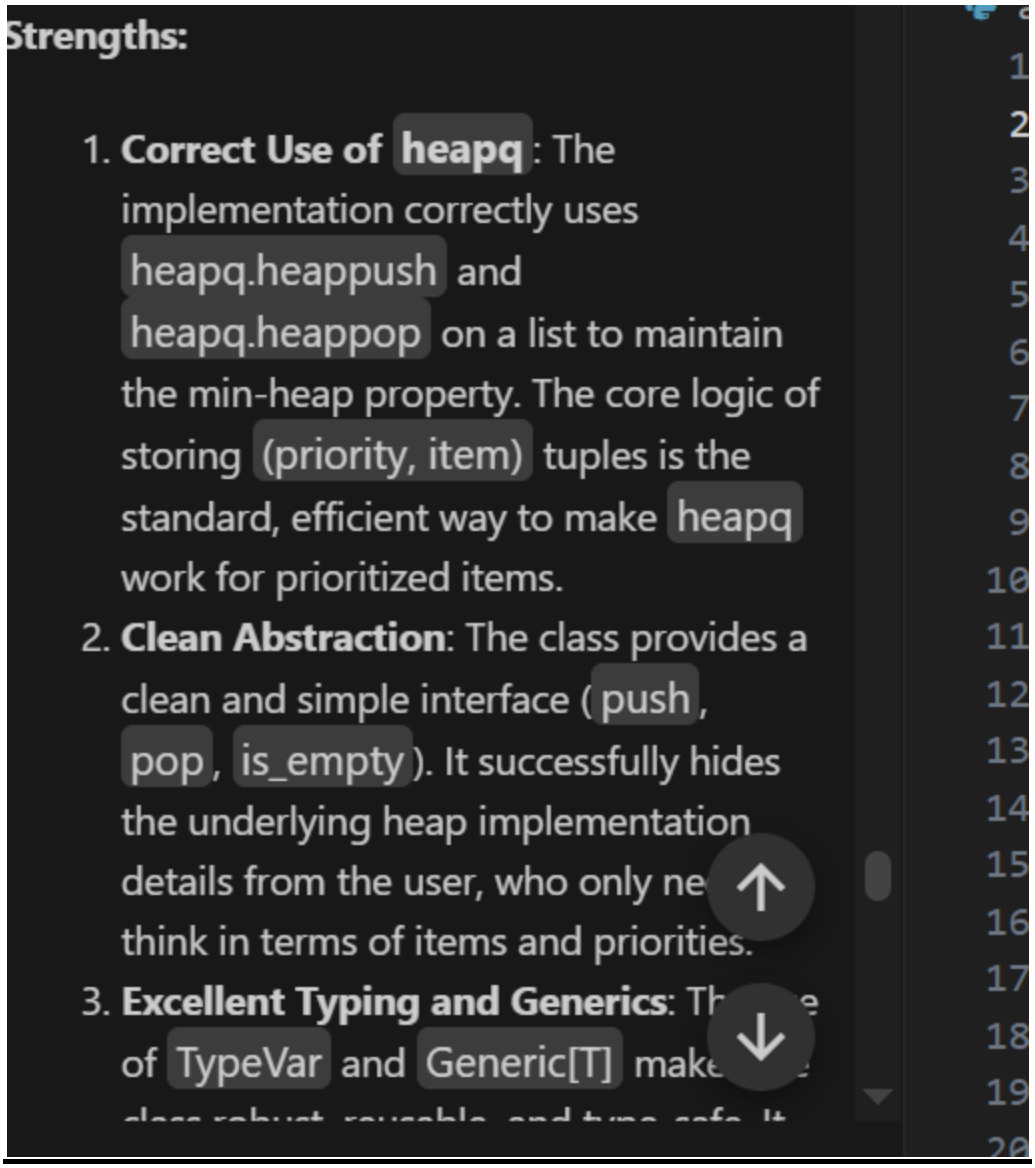
```
110
111     print(f"\nRemaining tasks in queue: {len(task_queue)}")
112
113     # 4. Pop all remaining tasks
114     while not task_queue.is_empty():
115         task = task_queue.pop()
116         print(f"Processing next task: '{task}'")
117
118     print(f"\nIs the queue empty now? {task_queue.is_empty()}") # Expected: True
119
```

Output:

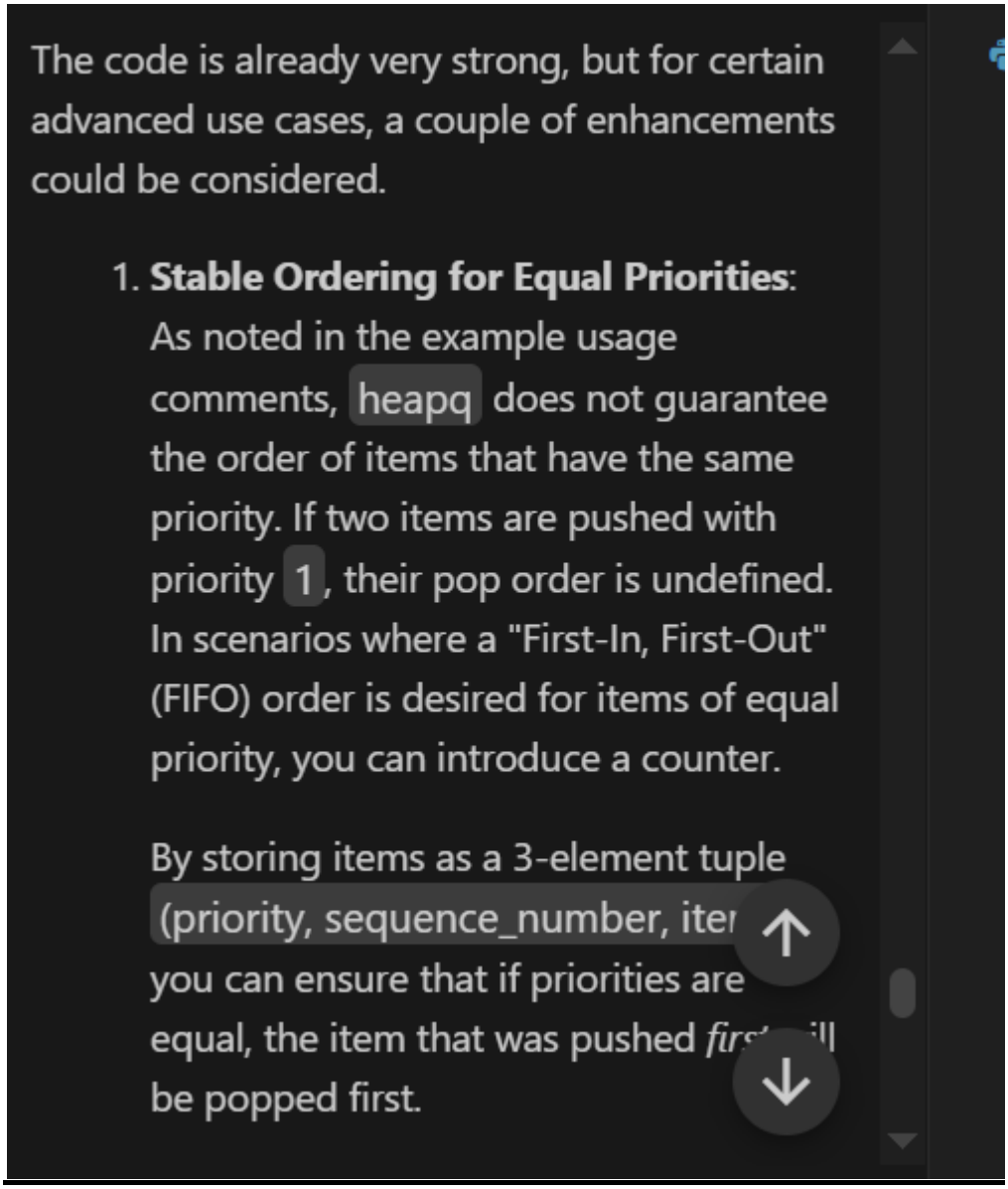
```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai_4.py"
Is the queue empty? True

Adding tasks with priorities...
Current queue size: 5
Is the queue empty? False

Processing tasks in order of priority:
Processing (Priority 1): 'Answer urgent emails'
Processing (Priority 1): 'Fix critical bug'
```

```
Processing tasks in order of priority:
Processing (Priority 1): 'Answer urgent emails'
Processing (Priority 1): 'Fix critical bug'
Processing (Priority 3): 'Write report'

Remaining tasks in queue: 2
Processing next task: 'Schedule team meeting'
Processing next task: 'Refill coffee'

Is the queue empty now? True
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

Observations:


Strengths:
1. Correct Use of heapq: The implementation correctly uses heapq.heappush and heapq.heappop on a list to maintain the min-heap property. The core logic of storing (priority, item) tuples is the standard, efficient way to make heapq work for prioritized items.
2. Clean Abstraction: The class provides a clean and simple interface (push, pop, is_empty). It successfully hides the underlying heap implementation details from the user, who only ne
think in terms of items and priorities.
3. Excellent Typing and Generics: Th
of TypeVar and Generic[T] make

3. **Excellent Typing and Generics**: The use of `TypeVar` and `Generic[T]` makes the class robust, reusable, and type-safe. It clearly communicates that the queue can hold any type of item, and it allows static analysis tools to catch potential errors.
4. **Good Documentation**: The docstrings for the class and its methods are clear, concise, and explain the purpose, arguments, and behavior (like "lowest priority value is highest priority") very well.
5. **Comprehensive Example**: The `if __name__ == "__main__":` block i… perfect example of how to use the … It tests all the main features, includin… handling items with the same prio… and makes the file runnable for

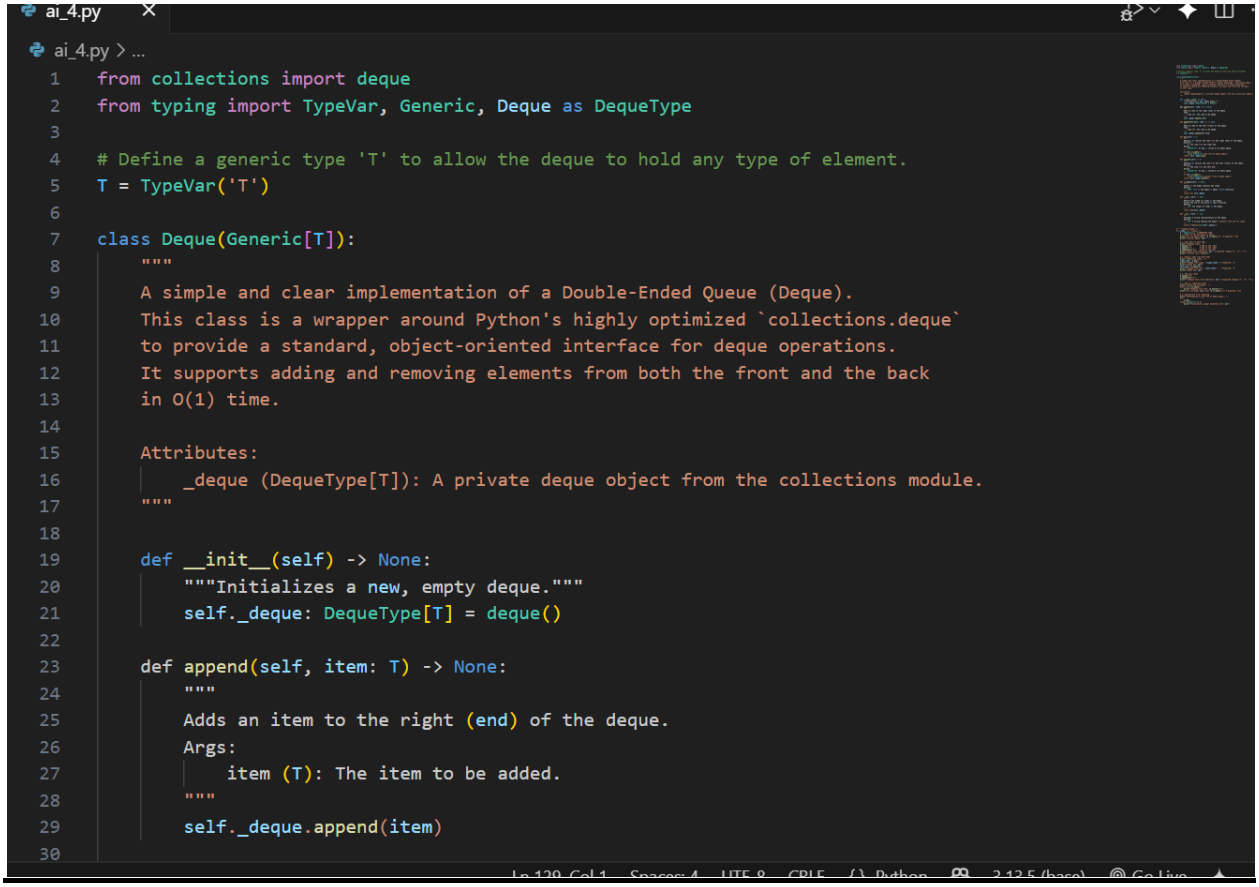


Task Description #8 –

Deque
Task: Use AI to implement a double-ended queue using collections.deque.
Sample Input Code:
class DequeDS:
pass
Expected Output:
• Insert and remove from both ends with docstrings.

Prompt: implement a double-ended queue using collections.deque

Code:



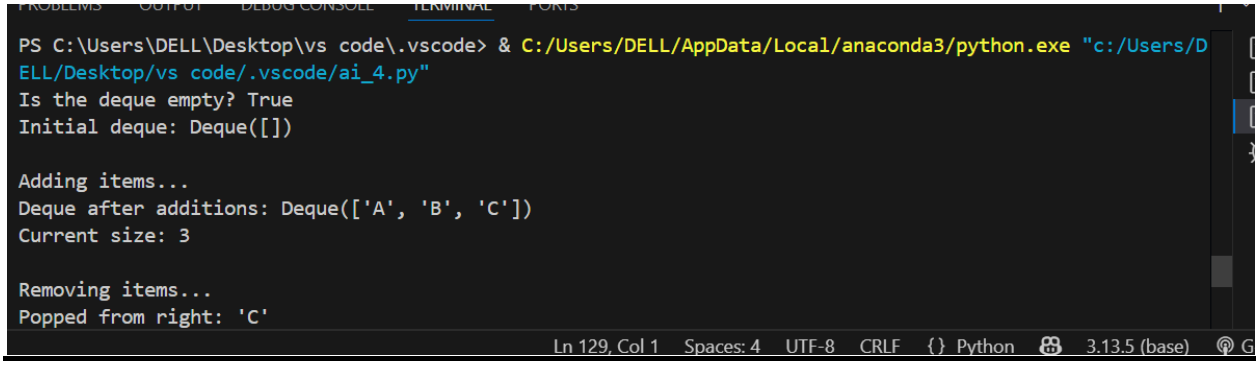
```python
from collections import deque
from typing import TypeVar, Generic, Deque as DequeType

# Define a generic type 'T' to allow the deque to hold any type of element.
T = TypeVar('T')

class Deque(Generic[T]):
    """
    A simple and clear implementation of a Double-Ended Queue (Deque).
    This class is a wrapper around Python's highly optimized `collections.deque`
    to provide a standard, object-oriented interface for deque operations.
    It supports adding and removing elements from both the front and the back
    in O(1) time.

    Attributes:
        _deque (DequeType[T]): A private deque object from the collections module.
    """

    def __init__(self) -> None:
        """Initializes a new, empty deque."""
        self._deque: DequeType[T] = deque()

    def append(self, item: T) -> None:
        """
        Adds an item to the right (end) of the deque.
        Args:
            item (T): The item to be added.
        """
        self._deque.append(item)

```

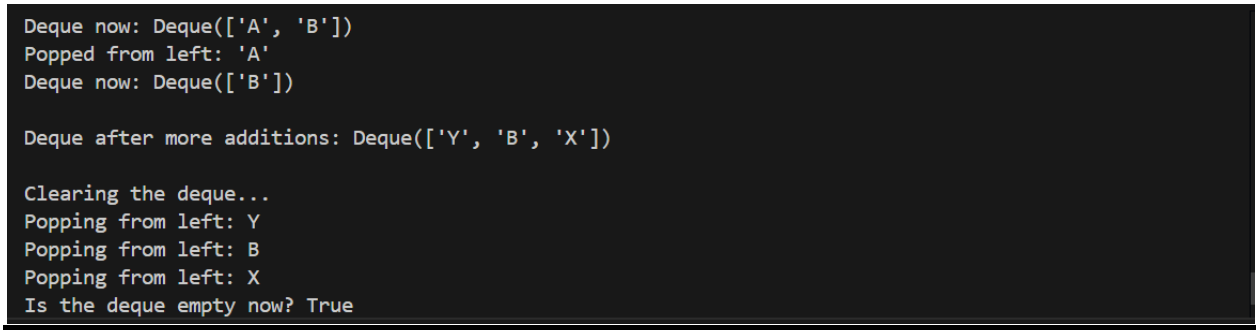
```python
        self._deque.append(item)

    def appendleft(self, item: T) -> None:
        """
        Adds an item to the left (front) of the deque.
        Args:
            item (T): The item to be added.
        """
        self._deque.appendleft(item)

    def pop(self) -> T:
        """
        Removes and returns the item from the right (end) of the deque.
        Returns:
            T: The item from the right end.
        Raises:
            IndexError: If pop is called on an empty deque.
        """
        if self.is_empty():
            raise IndexError("pop from an empty deque")
        return self._deque.pop()

    def popleft(self) -> T:
        """
        Removes and returns the item from the left (front) of the deque.
        Returns:
            T: The item from the left end.
        Raises:
            IndexError: If pop is called on an empty deque.
```

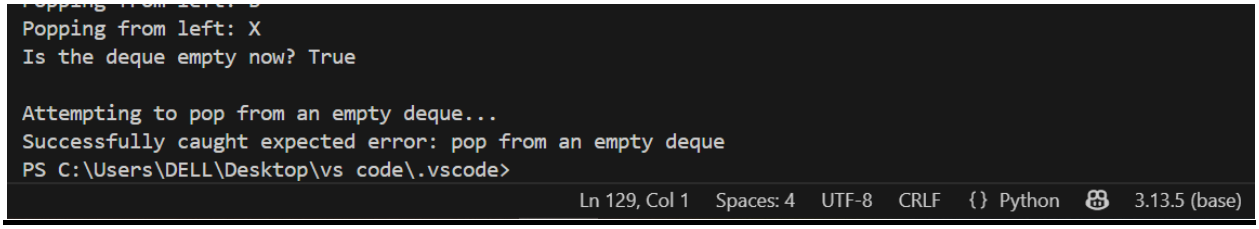
```python
    def popleft(self) -> T:
        """
        Removes and returns the item from the left (front) of the deque.
        Returns:
            T: The item from the left end.
        Raises:
            IndexError: If pop is called on an empty deque.
        """
        if self.is_empty():
            raise IndexError("popleft from an empty deque")
        return self._deque.popleft()

    def is_empty(self) -> bool:
        """
        Checks if the deque contains any items.
        Returns:
            bool: True if the deque is empty, False otherwise.
        """
        return not self._deque

    def __len__(self) -> int:
        """
        Returns the number of items in the deque.
        Allows the use of the built-in len() function.
        Returns:
            int: The number of items in the deque.
        """
        return len(self._deque)

    def __str__(self) -> str:
        """
        Provides a string representation of the deque.
        Returns:
            str: A string showing the deque's contents from left to right.
```

```
        """
        return f"Deque({list(self._deque)})"

# --- Example Usage ---
if __name__ == "__main__":
    d = Deque[str]()  # CORRECTED LINE
    # 1. Check if the new deque is empty
    print(f"Is the deque empty? {d.is_empty()}")  # Expected: True
    print(f"Initial deque: {d}")

    # 2. Add items to both ends
    print("\nAdding items...")
    d.append("B")        # Add to the right
    d.append("C")        # Add to the right
    d.appendleft("A")    # Add to the left
    print(f"Deque after additions: {d}") # Expected: Deque(['A', 'B', 'C'])
    print(f"Current size: {len(d)}")

    # 3. Remove items from both ends
    print("\nRemoving items...")
    right_item = d.pop()
    print(f"Popped from right: '{right_item}'") # Expected: 'C'
    print(f"Deque now: {d}")
    left_item = d.popleft()
    print(f"Popped from left: '{left_item}'")   # Expected: 'A'
    print(f"Deque now: {d}")

    # 4. Add more items
    d.append("X")
    d.appendleft("Y")
    print(f"\nDeque after more additions: {d}") # Expected: Deque(['Y', 'B', 'X'])

    # 5. Pop all remaining items
    print("\nClearing the deque...")
    while not d.is_empty():
        print(f"Popping from left: {d.popleft()}")
    print(f"Is the deque empty now? {d.is_empty()}") # Expected: True

    # 6. Demonstrate error handling
    print("\nAttempting to pop from an empty deque...")
    try:
        d.pop()
    except IndexError as e:
        print(f"Successfully caught expected error: {e}")
```

Output:





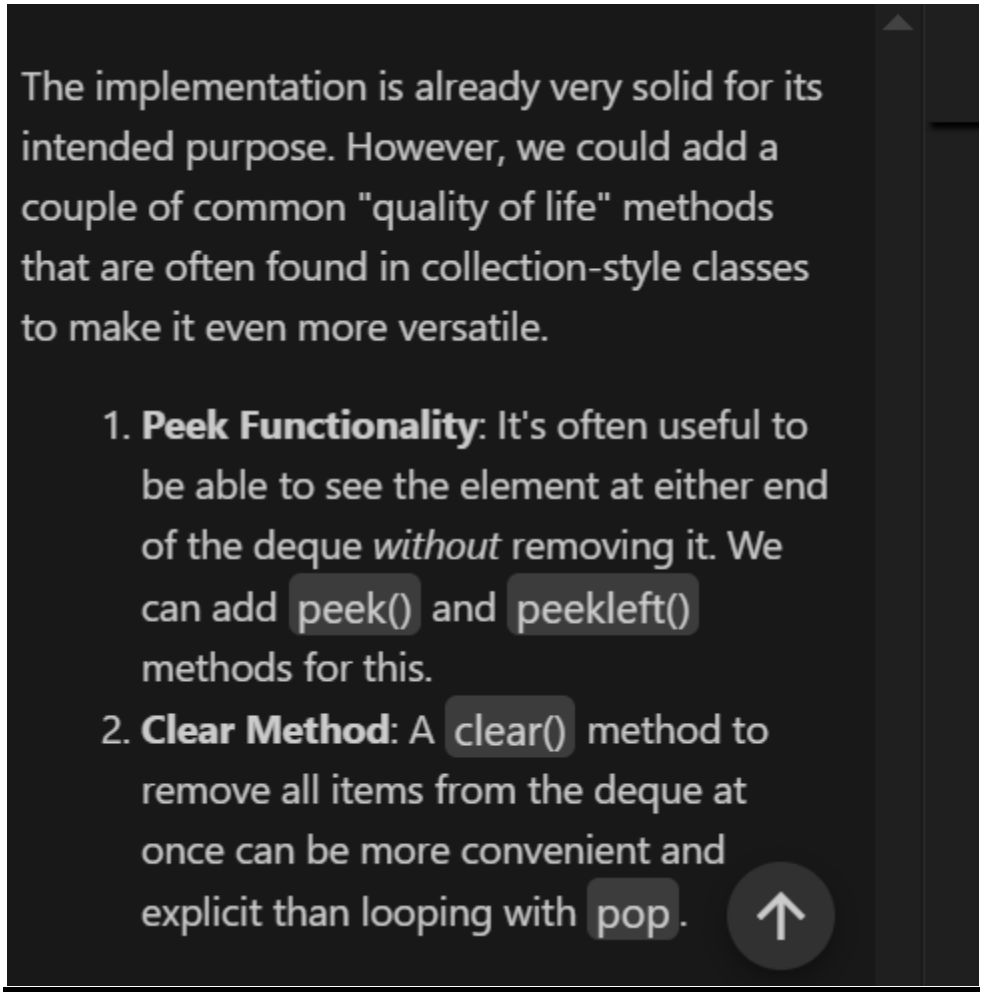


Observations:

Strengths:

1. **Excellent Abstraction**: The class provides a simple and intuitive interface (`append`, `pop`, `appendleft`, `popleft`) that cleanly abstracts away the underlying `collections.deque` object. This makes the code easy to use and understand.
2. **Correctness and Robustness**: The logic is sound. The methods correctly delegate to the wrapped `deque` object. The explicit checks for emptiness in `pop` and `popleft` before attempting the operation provide clear `IndexError` exceptions, which is great for robust error handling.

3. **Modern Python Features**: The use of `typing.Generic` and `TypeVar` is a standout feature. It makes your `Deque` class type-safe and flexible, allowing it to hold any type of element while enabling static analysis tools to catch errors.
4. **Clarity and Documentation**: The code is highly readable. The method names are standard and self-explanatory, and the docstrings are excellent. They clearly explain what each method does, its arguments, what it returns, and what errors it might raise.
5. **Comprehensive Example**: The `if __name__ == "__main__":` block is fantastic. It thoroughly tests every feature of the class, including edge cases like popping from an empty



Task Description #9 –

AI-Generated Data Structure Comparisons
Task: Use AI to generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.
Sample Input Code:
No code, prompt AI for a data structure comparison table
Expected Output:
• A markdown table with structure names, operations, and complexities.

Prompt:

generate a comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.

Code:

ai_1.py > ...

```python
import pandas as pd
from tabulate import tabulate

def create_data_structures_comparison():
    """
    Creates a comprehensive comparison table of different data structures
    with their time complexities for various operations.
    """

    # Define the data structures and their time complexities
    data = {
        'Data Structure': [
            'Array (Static)',
            'Array (Dynamic)',
            'Linked List (Singly)',
            'Linked List (Doubly)',
            'Stack (Array-based)',
            'Stack (Linked List-based)',
            'Queue (Array-based)',
            'Queue (Linked List-based)',
            'Deque (Double-ended Queue)',
            'Hash Table (Average)',
            'Hash Table (Worst)',
            'Binary Search Tree (Average)',
            'Binary Search Tree (Worst)',
            'AVL Tree',
```

```python
def create_data_structures_comparison():
            'Red-Black Tree',
            'B-Tree',
            'Heap (Binary)',
            'Trie',
            'Graph (Adjacency List)',
            'Graph (Adjacency Matrix)'
        ],
        'Access': [
            'O(1)',      # Array (Static)
            'O(1)',      # Array (Dynamic)
            'O(n)',      # Linked List (Singly)
            'O(n)',      # Linked List (Doubly)
            'O(n)',      # Stack (Array-based)
            'O(n)',      # Stack (Linked List-based)
            'O(n)',      # Queue (Array-based)
            'O(n)',      # Queue (Linked List-based)
            'O(n)',      # Deque
            'O(1)',      # Hash Table (Average)
            'O(n)',      # Hash Table (Worst)
            'O(log n)',  # BST (Average)
            'O(n)',      # BST (Worst)
            'O(log n)',  # AVL Tree
            'O(log n)',  # Red-Black Tree
            'O(log n)',  # B-Tree
            'N/A'        # Heap
```

```
4   def create_data_structures_comparison():
50              'O(log n)',  # B-Tree
51              'N/A',       # Heap
52              'O(m)',      # Trie (m = key length)
53              'O(V + E)',  # Graph (Adjacency List)
54              'O(1)'       # Graph (Adjacency Matrix)
55          ],
56          'Search': [
57              'O(n)',      # Array (Static)
58              'O(n)',      # Array (Dynamic)
59              'O(n)',      # Linked List (Singly)
60              'O(n)',      # Linked List (Doubly)
61              'O(n)',      # Stack (Array-based)
62              'O(n)',      # Stack (Linked List-based)
63              'O(n)',      # Queue (Array-based)
64              'O(n)',      # Queue (Linked List-based)
65              'O(n)',      # Deque
66              'O(1)',      # Hash Table (Average)
67              'O(n)',      # Hash Table (Worst)
68              'O(log n)',  # BST (Average)
69              'O(n)',      # BST (Worst)
70              'O(log n)',  # AVL Tree
71              'O(log n)',  # Red-Black Tree
72              'O(log n)',  # B-Tree
73              'O(n)',      # Heap
74              'O(m)',      # Trie
```

```
4   def create_data_structures_comparison():
78          'Insertion': [
79              'N/A',       # Array (Static)
80              'O(n)',      # Array (Dynamic)
81              'O(1)',      # Linked List (Singly) - at beginning
82              'O(1)',      # Linked List (Doubly) - at beginning
83              'O(1)',      # Stack (Array-based)
84              'O(1)',      # Stack (Linked List-based)
85              'O(1)',      # Queue (Array-based)
86              'O(1)',      # Queue (Linked List-based)
87              'O(1)',      # Deque
88              'O(1)',      # Hash Table (Average)
89              'O(n)',      # Hash Table (Worst)
90              'O(log n)',  # BST (Average)
91              'O(n)',      # BST (Worst)
92              'O(log n)',  # AVL Tree
93              'O(log n)',  # Red-Black Tree
94              'O(log n)',  # B-Tree
95              'O(log n)',  # Heap
96              'O(m)',      # Trie
97              'O(1)',      # Graph (Adjacency List)
98              'O(1)'       # Graph (Adjacency Matrix)
99          ],
100         'Deletion': [
101             'N/A',       # Array (Static)
102             'O(n)'       # Array (Dynamic)
```

Problems Output Debug Console **Terminal** Ports

```
def create_data_structures_comparison():
        ],
        'Deletion': [
            'N/A',       # Array (Static)
            'O(n)',      # Array (Dynamic)
            'O(1)',      # Linked List (Singly) - if node known
            'O(1)',      # Linked List (Doubly) - if node known
            'O(1)',      # Stack (Array-based)
            'O(1)',      # Stack (Linked List-based)
            'O(1)',      # Queue (Array-based)
            'O(1)',      # Queue (Linked List-based)
            'O(1)',      # Deque
            'O(1)',      # Hash Table (Average)
            'O(n)',      # Hash Table (Worst)
            'O(log n)',  # BST (Average)
            'O(n)',      # BST (Worst)
            'O(log n)',  # AVL Tree
            'O(log n)',  # Red-Black Tree
            'O(log n)',  # B-Tree
            'O(log n)',  # Heap
            'O(m)',      # Trie
            'O(V + E)',  # Graph (Adjacency List)
            'O(V²)'      # Graph (Adjacency Matrix)
        ],
        'Space Complexity': [
            'O(n)',      # Array (Static)
```

```
        'Space Complexity': [
            'O(n)',      # Array (Static)
            'O(n)',      # Array (Dynamic)
            'O(n)',      # Linked List (Singly)
            'O(n)',      # Linked List (Doubly)
            'O(n)',      # Stack (Array-based)
            'O(n)',      # Stack (Linked List-based)
            'O(n)',      # Queue (Array-based)
            'O(n)',      # Queue (Linked List-based)
            'O(n)',      # Deque
            'O(n)',      # Hash Table (Average)
            'O(n)',      # Hash Table (Worst)
            'O(n)',      # BST (Average)
            'O(n)',      # BST (Worst)
            'O(n)',      # AVL Tree
            'O(n)',      # Red-Black Tree
            'O(n)',      # B-Tree
            'O(n)',      # Heap
            'O(ALPHABET_SIZE * N * M)',  # Trie
            'O(V + E)',  # Graph (Adjacency List)
            'O(V²)'      # Graph (Adjacency Matrix)
        ]
    }

    # Create DataFrame
```

```
def create_data_structures_comparison():
    """

    # Create DataFrame
    df = pd.DataFrame(data)

    return df

def display_comparison_table():
    """
    Display the data structures comparison table in a formatted way.
    """
    df = create_data_structures_comparison()

    print("=" * 120)
    print("DATA STRUCTURES TIME COMPLEXITY COMPARISON TABLE".center(120))
    print("=" * 120)

    # Display using tabulate for better formatting
    print(tabulate(df, headers='keys', tablefmt='grid', showindex=False))

    print("\n" + "=" * 120)
    print("LEGEND AND NOTES".center(120))
    print("=" * 120)

    notes = [
        "- O(1) - Constant time"
```

```
def display_comparison_table():

    notes = [
        "• O(1) - Constant time",
        "• O(log n) - Logarithmic time",
        "• O(n) - Linear time",
        "• O(n log n) - Linearithmic time",
        "• O(n²) - Quadratic time",
        "• V = Number of vertices, E = Number of edges",
        "• m = Length of the key/string",
        "• N/A = Operation not applicable or not typically supported",
        "",
        "IMPORTANT NOTES:",
        "• Linked List insertion/deletion assumes you have a reference to the node",
        "• Hash Table performance depends on hash function quality and load factor",
        "• BST performance degrades to O(n) in worst case (unbalanced tree)",
        "• AVL and Red-Black trees maintain balance, guaranteeing O(log n) operations",
        "• Trie space complexity: ALPHABET_SIZE * N * M where N = number of words, M = average length",
        "• Array access is by index; search is for finding a specific value",
        "• Stack/Queue operations refer to push/pop and enqueue/dequeue respectively"
    ]

    for note in notes:
        print(note)

    return df
```

```python
ai_1.py > ...
def create_data_structures_comparison():

    # Create DataFrame
    df = pd.DataFrame(data)

    return df

def display_comparison_table():
    """
    Display the data structures comparison table in a formatted way.
    """
    df = create_data_structures_comparison()

    print("=" * 120)
    print("DATA STRUCTURES TIME COMPLEXITY COMPARISON TABLE".center(120))
    print("=" * 120)

    # Display using tabulate for better formatting
    print(tabulate(df, headers='keys', tablefmt='grid', showindex=False))

    print("\n" + "=" * 120)
    print("LEGEND AND NOTES".center(120))
    print("=" * 120)

    notes = [
        "• O(1) - Constant time",
```

```python
def display_comparison_table():
    notes = [
        "• O(1) - Constant time",
        "• O(log n) - Logarithmic time",
        "• O(n) - Linear time",
        "• O(n log n) - Linearithmic time",
        "• O(n²) - Quadratic time",
        "• V = Number of vertices, E = Number of edges",
        "• m = Length of the key/string",
        "• N/A = Operation not applicable or not typically supported",
        "",
        "IMPORTANT NOTES:",
        "• Linked List insertion/deletion assumes you have a reference to the node",
        "• Hash Table performance depends on hash function quality and load factor",
        "• BST performance degrades to O(n) in worst case (unbalanced tree)",
        "• AVL and Red-Black trees maintain balance, guaranteeing O(log n) operations",
        "• Trie space complexity: ALPHABET_SIZE * N * M where N = number of words, M = average length",
        "• Array access is by index; search is for finding a specific value",
        "• Stack/Queue operations refer to push/pop and enqueue/dequeue respectively"
    ]

    for note in notes:
        print(note)

    return df
```

Problems Output Debug Console **Terminal** Ports

```python
def display_comparison_table():
    return df

def create_detailed_comparison():
    """
    Create a more detailed comparison with additional properties.
    """
    detailed_data = {
        'Data Structure': [
            'Array',
            'Linked List',
            'Stack',
            'Queue',
            'Hash Table',
            'Binary Search Tree',
            'AVL Tree',
            'Heap',
            'Trie',
            'Graph'
        ],
        'Best Use Cases': [
            'Random access, mathematical operations, cache-friendly',
            'Frequent insertions/deletions, unknown size',
            'LIFO operations, function calls, undo operations',
            'FIFO operations, scheduling, buffering',
```

```python
def create_detailed_comparison():
        'Best Use Cases': [
            'Random access, mathematical operations, cache-friendly',
            'Frequent insertions/deletions, unknown size',
            'LIFO operations, function calls, undo operations',
            'FIFO operations, scheduling, buffering',
            'Fast lookups, caching, database indexing',
            'Sorted data, range queries, in-order traversal',
            'Guaranteed balanced operations, real-time systems',
            'Priority queues, heap sort, finding min/max',
            'Autocomplete, spell checkers, IP routing',
            'Networks, relationships, pathfinding'
        ],
        'Advantages': [
            'Fast access, memory efficient, cache-friendly',
            'Dynamic size, efficient insertion/deletion',
            'Simple LIFO semantics, memory efficient',
            'Fair scheduling, simple FIFO semantics',
            'Average O(1) operations, flexible keys',
            'Sorted order, efficient range operations',
            'Guaranteed balance, predictable performance',
            'Efficient priority operations, in-place sorting',
            'Prefix matching, memory efficient for strings',
            'Flexible relationships, various algorithms'
        ],
        'Disadvantages': [
```

```
def create_detailed_comparison():
        'Disadvantages': [
            'Fixed size (static), expensive insertion/deletion',
            'No random access, extra memory overhead',
            'Limited access pattern, potential stack overflow',
            'Limited access pattern, no random access',
            'Worst-case O(n), memory overhead, no ordering',
            'Can become unbalanced, worst-case O(n)',
            'Complex implementation, higher constants',
            'No efficient search, limited structure',
            'Memory overhead, complex deletion',
            'Memory overhead, complex algorithms'
        ]
    }

    detailed_df = pd.DataFrame(detailed_data)

    print("\n" + "=" * 150)
    print("DETAILED DATA STRUCTURES COMPARISON".center(150))
    print("=" * 150)

    # Print each row separately for better readability
    for index, row in detailed_df.iterrows():
        print(f"\n{row['Data Structure'].upper()}")
        print("-" * len(row['Data Structure']))
        print(f"Best Use Cases: {row['Best Use Cases']}")
```

```
def create_detailed_comparison():
    for index, row in detailed_df.iterrows():
        print(f"\n{row['Data Structure'].upper()}")
        print("-" * len(row['Data Structure']))
        print(f"Best Use Cases: {row['Best Use Cases']}")
        print(f"Advantages: {row['Advantages']}")
        print(f"Disadvantages: {row['Disadvantages']}")

    return detailed_df

if __name__ == "__main__":
    # Display the main comparison table
    df = display_comparison_table()

    # Display detailed comparison
    detailed_df = create_detailed_comparison()

    # Save to CSV for future reference
    df.to_csv('data_structures_comparison.csv', index=False)
    detailed_df.to_csv('data_structures_detailed.csv', index=False)

    print(f"\n{'='*50}")
    print("Tables saved to CSV files for future reference!")
    print("• data_structures_comparison.csv")
    print("• data_structures_detailed.csv")
```

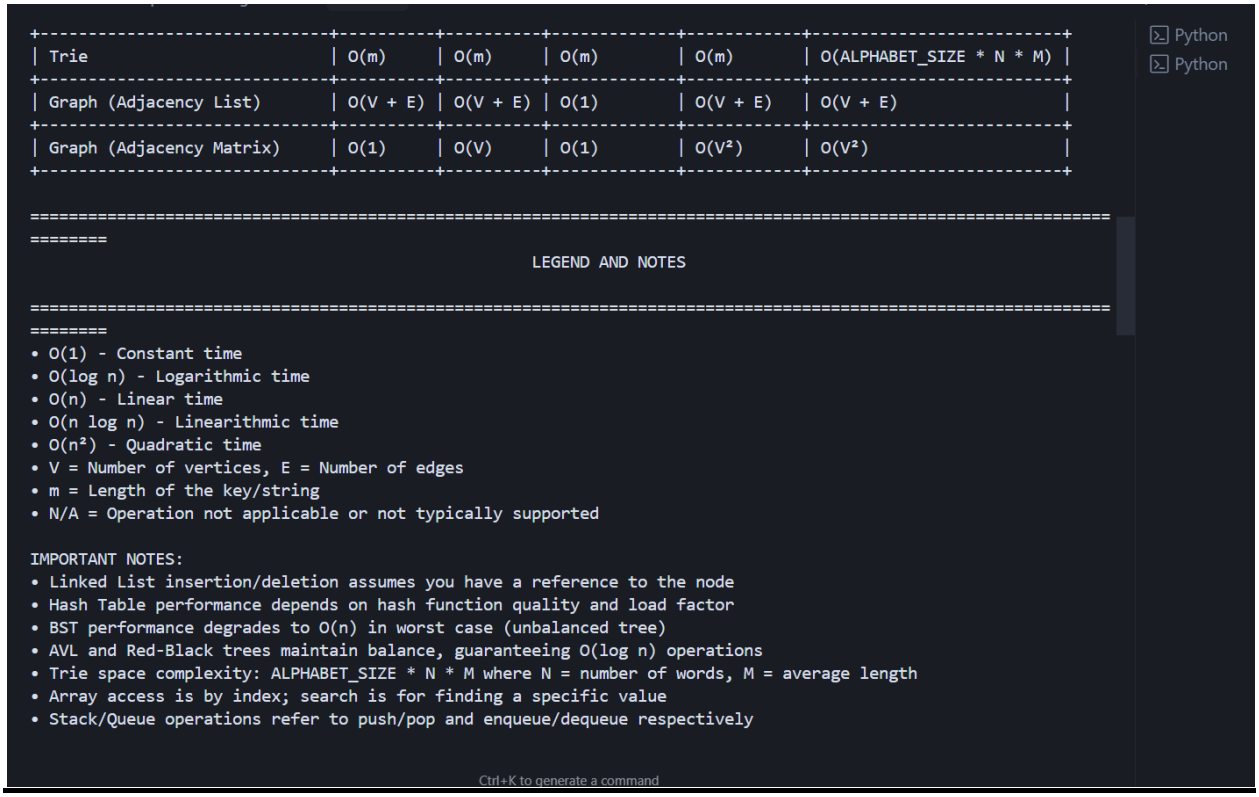
Output:

```
PS C:\Users\DELL\Desktop> & C:/Users/DELL/AppData/Local/anaconda3/python.exe c:/Users/DELL/Desktop/ai_1.py
================================================================================================================
========
                                DATA STRUCTURES TIME COMPLEXITY COMPARISON TABLE

================================================================================================================
========
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Data Structure               | Access   | Search   | Insertion   | Deletion    | Space Complexity         |
+==============================+==========+==========+=============+=============+==========================+
| Array (Static)               | O(1)     | O(n)     | N/A         | N/A         | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Array (Dynamic)              | O(1)     | O(n)     | O(n)        | O(n)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Linked List (Singly)         | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Linked List (Doubly)         | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Stack (Array-based)          | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Stack (Linked List-based)    | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Queue (Array-based)          | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Queue (Linked List-based)    | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Deque (Double-ended Queue)   | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Hash Table (Average)         | O(1)     | O(1)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Hash Table (Worst)           | O(n)     | O(n)     | O(n)        | O(n)        | O(n)                     |
```

Ctrl+K to generate a command

```
| Array (Dynamic)              | O(1)     | O(n)     | O(n)        | O(n)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Linked List (Singly)         | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Linked List (Doubly)         | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Stack (Array-based)          | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Stack (Linked List-based)    | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Queue (Array-based)          | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Queue (Linked List-based)    | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Deque (Double-ended Queue)   | O(n)     | O(n)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Hash Table (Average)         | O(1)     | O(1)     | O(1)        | O(1)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Hash Table (Worst)           | O(n)     | O(n)     | O(n)        | O(n)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Binary Search Tree (Average) | O(log n) | O(log n) | O(log n)    | O(log n)    | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Binary Search Tree (Worst)   | O(n)     | O(n)     | O(n)        | O(n)        | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| AVL Tree                     | O(log n) | O(log n) | O(log n)    | O(log n)    | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Red-Black Tree               | O(log n) | O(log n) | O(log n)    | O(log n)    | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| B-Tree                       | O(log n) | O(log n) | O(log n)    | O(log n)    | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
| Heap (Binary)                | N/A      | O(n)     | O(log n)    | O(log n)    | O(n)                     |
+------------------------------+----------+----------+-------------+-------------+--------------------------+
```

Ctrl+K to generate a command



Trie	O(m)	O(m)	O(m)	O(m)	O(ALPHABET_SIZE * N * M)
Graph (Adjacency List)	O(V + E)	O(V + E)	O(1)	O(V + E)	O(V + E)
Graph (Adjacency Matrix)	O(1)	O(V)	O(1)	O(V²)	O(V²)

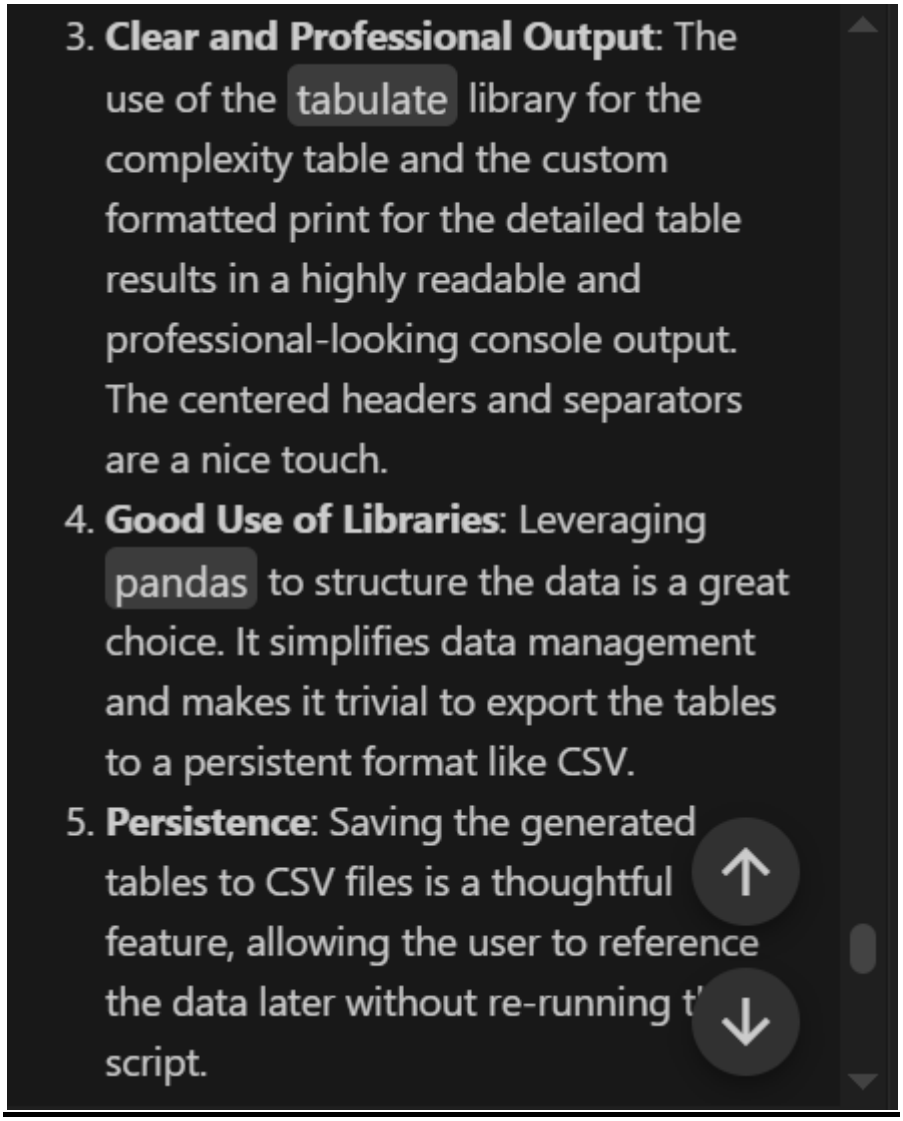
LEGEND AND NOTES

- O(1) - Constant time
- O(log n) - Logarithmic time
- O(n) - Linear time
- O(n log n) - Linearithmic time
- O(n²) - Quadratic time
- V = Number of vertices, E = Number of edges
- m = Length of the key/string
- N/A = Operation not applicable or not typically supported

IMPORTANT NOTES:
- Linked List insertion/deletion assumes you have a reference to the node
- Hash Table performance depends on hash function quality and load factor
- BST performance degrades to O(n) in worst case (unbalanced tree)
- AVL and Red-Black trees maintain balance, guaranteeing O(log n) operations
- Trie space complexity: ALPHABET_SIZE * N * M where N = number of words, M = average length
- Array access is by index; search is for finding a specific value
- Stack/Queue operations refer to push/pop and enqueue/dequeue respectively

Observations:

1. **Excellent Structure and Modularity**: The code is cleanly divided into functions with specific responsibilities (`create_data_structures_comparison`, `display_comparison_table`, `create_detailed_comparison`). This makes the script easy to read, understand, and maintain.
2. **Comprehensive and Accurate Data**: The information presented in both tables is thorough and accurate. It covers a wide range of essential data structures and correctly identifies their time complexities, use cases, and trade-offs. This is a high-quality "cheat sheet."
3. **Clear and Professional Output**: The use of the `tabulate` library for the complexity table and the custom formatted print for the detailed tal results in a highly readable and professional-looking console output. The centered headers and separat are a nice touch.



Task Description #10

Real-Time Application Challenge – Choose the
Right Data Structure
Scenario:

Your college wants to develop a Campus Resource Management System that handles:
1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due

dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.
Student Task:
• For each feature, select the most appropriate data structure from the list below:
- Stack
- Queue
- Priority Queue
- Linked List
- Binary Search Tree (BST)
- Graph
- Hash Table
- Deque

• Justify your choice in 2–3 sentences per feature.
• Implement one selected feature as a working Python program with AI-assisted code generation.
Expected Output:
• A table mapping feature → chosen data structure → justification.
• A functional Python program implementing the chosen feature with comments and docstrings.

Prompt:

Your college wants to develop a Campus Resource Management System that handles:
1. Student Attendance Tracking – Daily log of students entering/exiting the campus.
2. Event Registration System – Manage participants in events with quick search and removal.
3. Library Book Borrowing – Keep track of available books and their due dates.
4. Bus Scheduling System – Maintain bus routes and stop connections.
5. Cafeteria Order Queue – Serve students in the order they arrive.
Student Task:
• For each feature, select the most appropriate data structure from the list below:
- Stack
- Queue
- Priority Queue
- Linked List
- Binary Search Tree (BST)
- Graph
- Hash Table
- Deque

Code:

ai_4.py > ...

```python
from typing import Dict, Optional

class EventRegistration:
    """
    Manages participant registration for an event using a hash table.

    This system allows for efficient addition, removal, and lookup of
    participants by their unique student ID.

    Attributes:
        event_name (str): The name of the event.
        participants (Dict[int, str]): A hash table (dictionary) mapping
                                       student IDs to student names.
    """

    def __init__(self, event_name: str):
        """
        Initializes a new registration system for a given event.

        Args:
            event_name (str): The name of the event.
        """
        if not event_name:
            raise ValueError("Event name cannot be empty.")
        self.event_name: str = event_name
        self.participants: Dict[int, str] = {}
        print(f"Registration system created for event: '{self.event_name}'")

    def register_student(self, student_id: int, student_name: str) -> None:
        """
```

Ln 122, Col 1 Spaces: 4 UTF-8 CRLF {} Python 3.13.5 (base) Go Live

```python
class EventRegistration:

    def register_student(self, student_id: int, student_name: str) -> None:
        """
        Registers a student for the event.

        If the student is already registered, it prints a message.
        This operation has an average time complexity of O(1).

        Args:
            student_id (int): The unique ID of the student.
            student_name (str): The name of the student.
        """
        if student_id in self.participants:
            print(f"-> Warning: Student {student_name} (ID: {student_id}) is already registered.")
        else:
            self.participants[student_id] = student_name
            print(f"-> Success: Registered {student_name} (ID: {student_id}) for '{self.event_name}'")

    def remove_student(self, student_id: int) -> None:
        """
        Removes a student from the event registration.

        If the student is not found, it prints a message.
        This operation has an average time complexity of O(1).

        Args:
            student_id (int): The ID of the student to remove.
        """
        if student_id in self.participants:
```

```
        if student_id in self.participants:
            student_name = self.participants.pop(student_id)
            print(f"-> Success: Removed {student_name} (ID: {student_id}) from the event.")
        else:
            print(f"-> Warning: Student with ID {student_id} not found in registration list.")

    def find_student(self, student_id: int) -> Optional[str]:
        """
        Finds a student's name by their ID.

        This operation has an average time complexity of O(1).

        Args:
            student_id (int): The ID of the student to find.

        Returns:
            Optional[str]: The student's name if found, otherwise None.
        """
        return self.participants.get(student_id)

    def display_participants(self) -> None:
        """Prints a list of all registered participants."""
        print(f"\n--- Participants for '{self.event_name}' ({self.get_participant_count()}) ---")
        if not self.participants:
            print("No students are currently registered.")
        else:
            for student_id, student_name in self.participants.items():
```

```


    def get_participant_count(self) -> int:
        """Returns the total number of registered participants."""
        return len(self.participants)

# --- Example Usage ---
if __name__ == "__main__":
    # 1. Create a registration system for a "CodeFest 2024" event
    codefest_regs = EventRegistration("CodeFest 2024")

    # 2. Register several students
    print("\n--- Registering Students ---")
    codefest_regs.register_student(101, "Alice")
    codefest_regs.register_student(205, "Bob")
    codefest_regs.register_student(315, "Charlie")
    codefest_regs.register_student(101, "Alice") # Attempt to register a duplicate

    # 3. Display the current list of participants
    codefest_regs.display_participants()

    # 4. Find a specific student
    print("\n--- Searching for a Student ---")
    student_id_to_find = 205
    found_name = codefest_regs.find_student(student_id_to_find)
    if found_name:
        print(f"Search result: Student with ID {student_id_to_find} is '{found_name}'.")
    else:
        print(f"Search result: Student with ID {student_id_to_find} not found.")
```

```
ai_4.py > ...
111         print(f"Search result: Student with ID {student_id_to_find} is '{found_name}'.")
112     else:
113         print(f"Search result: Student with ID {student_id_to_find} not found.")
114
115     # 5. Remove a student
116     print("\n--- Removing a Student ---")
117     codefest_regs.remove_student(315) # Remove Charlie
118     codefest_regs.remove_student(999) # Attempt to remove a non-existent student
119
120     # 6. Display the final list of participants
121     codefest_regs.display_participants()
122
```

Output:

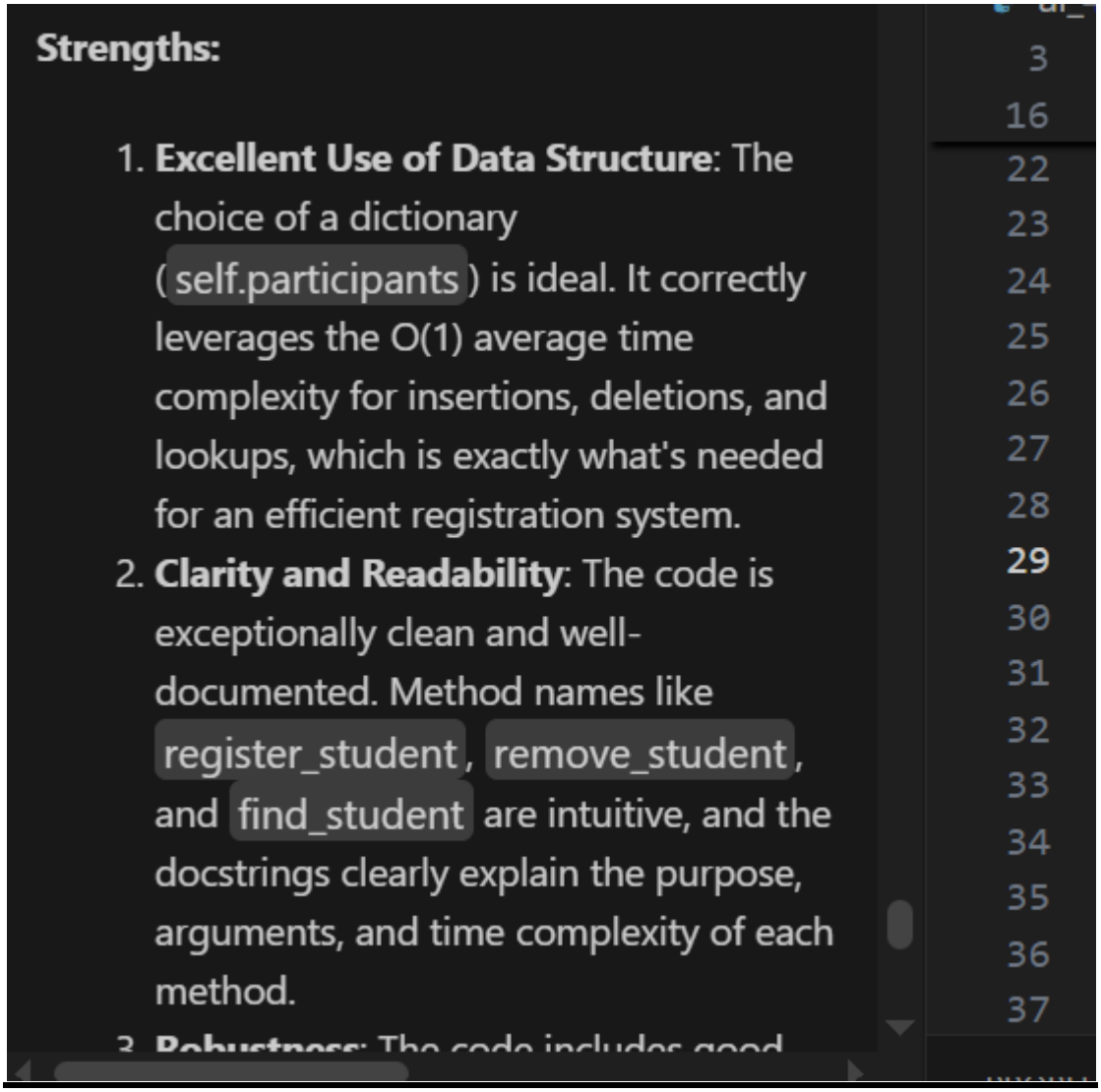
```
PS C:\Users\DELL\Desktop\vs code\.vscode> & C:/Users/DELL/AppData/Local/anaconda3/python.exe "c:/Users/DELL/Desktop/vs code/.vscode/ai_4.py"
Registration system created for event: 'CodeFest 2024'

--- Registering Students ---
-> Success: Registered Alice (ID: 101) for 'CodeFest 2024'.
-> Success: Registered Bob (ID: 205) for 'CodeFest 2024'.
-> Success: Registered Charlie (ID: 315) for 'CodeFest 2024'.
-> Warning: Student Alice (ID: 101) is already registered.

--- Participants for 'CodeFest 2024' (3) ---
```

```
--- Participants for 'CodeFest 2024' (3) ---
  - ID: 101, Name: Alice
  - ID: 205, Name: Bob
  - ID: 315, Name: Charlie
-------------------------------------------

--- Searching for a Student ---
Search result: Student with ID 205 is 'Bob'.

--- Removing a Student ---
```

```
  --- Searching for a Student ---
  Search result: Student with ID 205 is 'Bob'.

  --- Removing a Student ---
  -> Success: Removed Charlie (ID: 315) from the event.
  -> Warning: Student with ID 999 not found in registration list.

  --- Participants for 'CodeFest 2024' (2) ---
    - ID: 101, Name: Alice
    - ID: 205, Name: Bob
```

```
Search result: Student with ID 205 is 'Bob'.

--- Removing a Student ---
-> Success: Removed Charlie (ID: 315) from the event.
-> Warning: Student with ID 999 not found in registration list.

--- Participants for 'CodeFest 2024' (2) ---
  - ID: 101, Name: Alice
  - ID: 205, Name: Bob
------------------------------------------
PS C:\Users\DELL\Desktop\vs code\.vscode>
```

Observations:



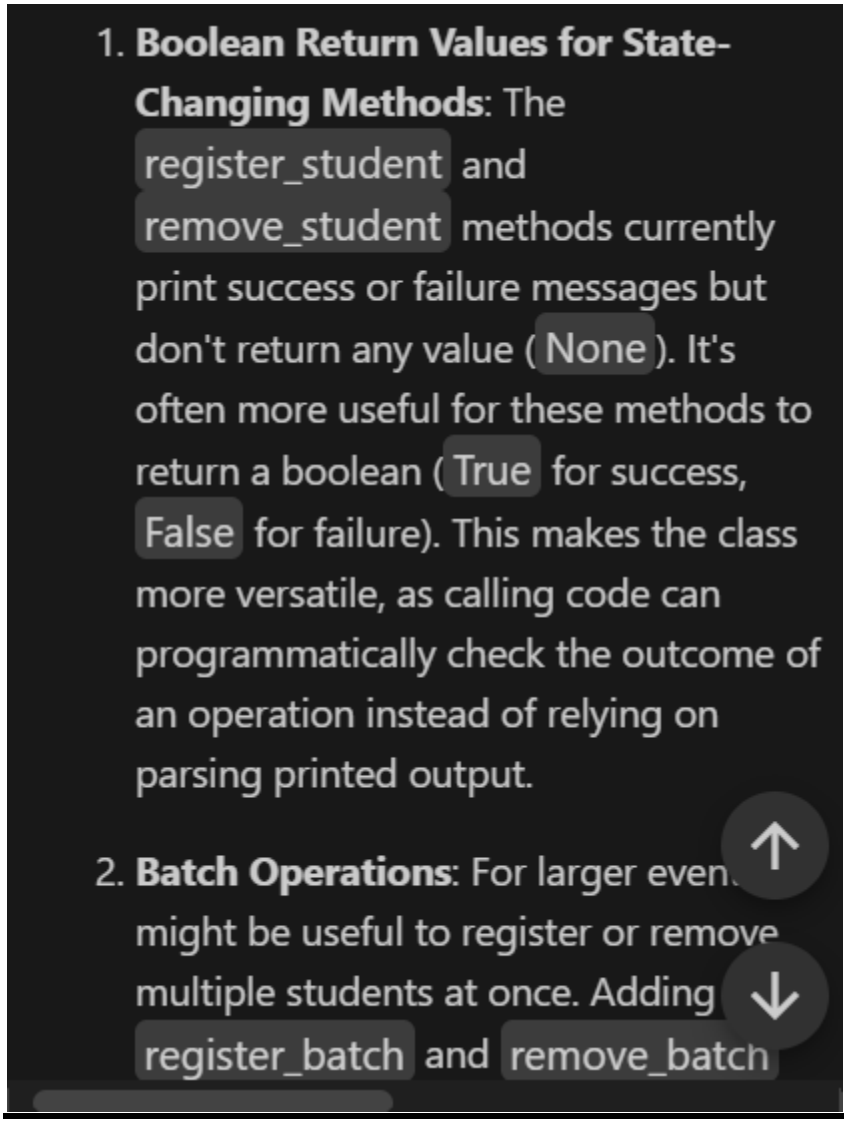
3. **Robustness**: The code includes good validation and error handling. The `__init__` method checks for an empty event name, and the `register_student` and `remove_student` methods gracefully handle cases where a student already exists or is not found, providing helpful feedback to the user.
4. **Modern Python Practices**: The use of type hints (`Dict`, `Optional`) makes the code more robust and easier for developers (and tools) to understand and maintain.
5. **Comprehensive Example**: The `if __name__ == "__main__":` block i… perfect demonstration of the class's functionality. It walks through all t… core features, including edge cases like

5. **Comprehensive Example**: The `if __name__ == "__main__":` block is a perfect demonstration of the class's functionality. It walks through all the core features, including edge cases like duplicate registrations and attempting to remove a non-existent student.

Potential Enhancements:

The code is already very solid, but here are a couple of suggestions that could enhance its functionality and return value consistency.

1. **Boolean Return Values for State-Changing Methods**: The `register_student` and `remove_student` methods currer print success or failure messages but



Deliverables (For All Tasks)

1. AI-generated prompts for code and test case generation.
2. At least 3 assert test cases for each task.
3. AI-generated initial code and execution screenshots.

4. Analysis of whether code passes all tests.
5. Improved final version with inline comments and explanation.
6. Compiled report (Word/PDF) with prompts, test cases, assertions, code, and output