

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      X
ai.py > ...
1   from typing import TypeVar, Generic, List
2
3   # Define a generic type 'T' to allow the stack to hold any type of element.
4   T = TypeVar('T')
5
6   class Stack(Generic[T]):
7       """
8           A simple and efficient implementation of a Stack data structure (LIFO).
9
10          This class provides the fundamental stack operations: push, pop, peek,
11          and is_empty. It is built upon Python's list type.
12
13          Attributes:
14              _items (List[T]): A private list to store the stack elements.
15          """
16
17          def __init__(self) -> None:
18              """Initializes a new, empty stack."""
19              self._items: List[T] = []
20
21          def push(self, item: T) -> None:
22              """
23                  Adds an item to the top of the stack.
24
25                  This operation has an average time complexity of O(1).
26
27                  Args:
28                      item (T): The item to be added.
29                  """
30              self._items.append(item)
```

```
6   class Stack(Generic[T]):
7
8       def pop(self) -> T:
9           """
10              Removes and returns the item from the top of the stack.
11
12              This operation has a time complexity of O(1).
13
14              Returns:
15                  T: The item that was at the top of the stack.
16
17              Raises:
18                  IndexError: If pop is called on an empty stack.
19
20              if self.is_empty():
21                  raise IndexError("pop from an empty stack")
22              return self._items.pop()
23
24
25       def peek(self) -> T:
26           """
27               Returns the item at the top of the stack without removing it.
28
29               This operation has a time complexity of O(1).
30
31               Returns:
32                   T: The item currently at the top of the stack.
33
34               Raises:
35                   IndexError: If peek is called on an empty stack.
36
```

```
ai.py > ...
6   class Stack(Generic[T]):
48     def peek(self) -> T:
50       if self.is_empty():
51         raise IndexError("peek from an empty stack")
52       return self._items[-1]
53
54     def is_empty(self) -> bool:
55       """
56       Checks if the stack contains any items.
57
58       This operation has a time complexity of O(1).
59
60       Returns:
61         bool: True if the stack is empty, False otherwise.
62       """
63       return not self._items
64
65     def size(self) -> int:
66       """
67       Returns the number of items in the stack.
68
69       Returns:
70         int: The current size of the stack.
71       """
72       return len(self._items)
73
74     def __str__(self) -> str:
75       """
76       Provides a string representation of the stack.
77
```

```
ai.py > ...
6   class Stack(Generic[T]):
44     def __str__(self) -> str:
45
46       Returns:
47         str: A string showing the stack's contents from bottom to top.
48       """
49       return f"Stack({self._items})"
50
51   # --- Example Usage ---
52   if __name__ == "__main__":
53     # Create a new stack for integers
54     s = Stack[int]()
55
56     # 1. Check if the new stack is empty
57     print(f"\nIs the stack empty? {s.is_empty()}") # Expected: True
58     print(f"Initial stack: {s}")
59
60     # 2. Push items onto the stack
61     print("\nPushing 10, 20, then 30...")
62     s.push(10)
63     s.push(20)
64     s.push(30)
65     print(f"Stack after pushes: {s}")
66     print(f"Current size: {s.size()}") # Expected: 3
67     print(f"Is the stack empty? {s.is_empty()}") # Expected: False
68
69     # 3. Peek at the top item
70     print(f"\nPeeking at the top item: {s.peek()}") # Expected: 30
71     print(f"Stack remains unchanged: {s}")
72
73
```

```
ai.py > ...
114
115     # 4. Pop an item from the stack
116     popped_item = s.pop()
117     print(f"\nPopped item: {popped_item}")           # Expected: 30
118     print(f"Stack after one pop: {s}")
119     print(f"New top item is: {s.peek()}")           # Expected: 20
120
121     # 5. Pop all remaining items
122     s.pop()
123     s.pop()
124     print(f"\nStack after popping all items: {s}")
125     print(f"Is the stack empty now? {s.is_empty()}") # Expected: True
126
127     # 6. Demonstrate error handling on an empty stack
128     print("\nAttempting to pop from an empty stack...")
129     try:
130         s.pop()
131     except IndexError as e:
132         print(f"Successfully caught expected error: {e}")
133
```

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:

```
1  Strengths
2
3  Generic Type Support
4
5  You used TypeVar and Generic[T] properly, which makes the stack flexible for any type (int, str, custom objects,
etc.).
6
7  Encapsulation
8
9  _items is a private list, so the stack maintains control over its data.
10 Error Handling
11
12 pop() and peek() correctly raise IndexError when the stack is empty, which is consistent with Python's built-in
list behavior.
13
14 Efficiency
15
16 All main operations (push, pop, peek, is_empty, __len__) are O(1) time complexity.
17
18 This makes the stack efficient for real use cases.
19
20 Readability and Documentation
21
22 Excellent use of docstrings: each method has a clear explanation, time complexity, return type, and raised
exceptions.
23
24 Easy to understand and maintain.
25
26 String Representation
27
28 __str__ gives a clean representation of stack contents, making debugging easy (Stack([10, 20, 30])).
```

Cursor Tab | Ln 34 Col 41 | Spaces: 4 | UTF-8 | CR LF | () | Markdown | @ Go Live | □

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:

```
FJ: / Queue / is_empty
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():
            raise IndexError("Queue is empty")
        return self.items.pop(0)

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

            Returns:
                The item at the front of the queue
        """
        if self.is_empty():
            raise IndexError("Queue is empty")
        return self.items[0]

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

            Returns:
                The size of the queue
        """
        return len(self.items)

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

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

```

1  class Queue:
20     def dequeue(self):
30         if self.is_empty():
31             raise IndexError("dequeue from empty queue")
32             return self.items.pop(0)
33
34     def front(self):
35         """
36             Return the item at the front of the queue without removing it.
37
38             Returns:
39                 The item at the front of the queue
40
41             Raises:
42                 IndexError: If the queue is empty
43             """
44             if self.is_empty():
45                 raise IndexError("front called on empty queue")
46             return self.items[0]
47
48     def rear(self):
49         """
50             Return the item at the rear of the queue without removing it.
51
52             Returns:
53                 The item at the rear of the queue
54
55             Raises:
56                 IndexError: If the queue is empty
57             """

```

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

```

1  class Queue:
48     def rear(self):
58         if self.is_empty():
59             raise IndexError("rear called on empty queue")
60             return self.items[-1]
61
62     def is_empty(self):
63         """
64             Check if the queue is empty.
65
66             Returns:
67                 bool: True if the queue is empty, False otherwise
68             """
69             return len(self.items) == 0
70
71     def size(self):
72         """
73             Return the number of items in the queue.
74
75             Returns:
76                 int: The number of items in the queue
77             """
78             return len(self.items)
79
80     def clear(self):
81         """
82             Remove all items from the queue.
83             self.items = []
84
85     def __str__(self):
86         """

```

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

```

1   class Queue:
2       def __str__(self):
3           """
4               Return a string representation of the queue.
5           """
6           Returns:
7               str: String representation showing items from front to rear
8           """
9           return f"Queue({self.items})"
10
11      def __repr__(self):
12          """Return a detailed string representation of the queue."""
13          return f"Queue(items={self.items})"
14
15      def __len__(self):
16          """Return the length of the queue."""
17          return len(self.items)
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102     # Example usage and demonstration
103     if __name__ == "__main__":
104         # Create a new queue
105         queue = Queue()
106
107         print("== Queue Implementation Demo ==")
108         print(f"Initial queue: {queue}")
109         print(f"Is empty: {queue.is_empty()}")
110         print(f"Size: {queue.size()}")
111
112         # Add items to the queue

```

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

```

111
112         # Add items to the queue
113         print("\n--- Adding items ---")
114         queue.enqueue("first")
115         queue.enqueue("second")
116         queue.enqueue("third")
117         print(f"After enqueueing 'first', 'second', 'third': {queue}")
118         print(f"Size: {queue.size()}")
119         print(f"Front item: {queue.front()}")
120         print(f"Rear item: {queue.rear()}")
121
122         # Remove items from the queue
123         print("\n--- Removing items ---")
124         item1 = queue.dequeue()
125         print(f"Dequeued: {item1}")
126         print(f"Queue after dequeue: {queue}")
127
128         item2 = queue.dequeue()
129         print(f"Dequeued: {item2}")
130         print(f"Queue after dequeue: {queue}")
131
132         # Add more items
133         print("\n--- Adding more items ---")
134         queue.enqueue("fourth")
135         queue.enqueue("fifth")
136         print(f"After adding 'fourth', 'fifth': {queue}")
137
138         # Show remaining operations
139         print(f"Front: {queue.front()}")
140         print(f"Rear: {queue.rear()}")

```

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

```

141     print(f"Size: {queue.size()}")
142
143     # Clear the queue
144     print("\n--- Clearing queue ---")
145     queue.clear()
146     print(f"After clearing: {queue}")
147     print(f"Is empty: {queue.is_empty()}")
148
149     # Demonstrate error handling
150     print("\n--- Error handling demo ---")
151     try:
152         queue.dequeue()
153     except IndexError as e:
154         print(f"Error when dequeuing from empty queue: {e}")
155
156     try:
157         queue.front()
158     except IndexError as e:
159         print(f"Error when accessing front of empty queue: {e}")
160

```

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 enqueueing '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

Key Observation: Performance of `dequeue`

The `dequeue` method uses `self.items.pop(0)`. While this works correctly, it is inefficient.

- **Problem:** In a standard Python list, removing an element from the beginning is an **$O(n)$** operation (where 'n' is the number of items in the list). This is because after the first element is removed, every subsequent element must be shifted one position to the left. For a very large queue, this can become a major performance bottleneck.
- **Solution:** The Pythonic and highly efficient way to implement a queue is by using `collections.deque` (double-



- **Solution:** The Pythonic and highly efficient way to implement a queue is by using `collections.deque` (double-ended queue). It is specifically designed for fast appends and pops from both ends, making both `enqueue` and `dequeue` operations **$O(1)$** .

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 > ...
1  from typing import TypeVar, Generic, Optional
2
3  # Define a generic type 'T' for the data within the nodes.
4  T = TypeVar('T')
5
6  class Node(Generic[T]):
7      """
8          A single node in a Singly Linked List.
9
10     Attributes:
11         data (T): The data stored within the node.
12         next (Optional[Node[T]]): A reference to the next node in the list.
13     """
14
15     def __init__(self, data: T):
16         self.data: T = data
17         self.next: Optional['Node[T]'] = None
18
19  class SinglyLinkedList(Generic[T]):
20      """
21          A Singly Linked List data structure.
22
23          This class manages a sequence of nodes, providing methods to insert
24          new nodes and display the contents of the list.
25
26     Attributes:
27         head (Optional[Node[T]]): The first node in the linked list.
28     """
29
30     def __init__(self) -> None:
31         """Initializes an empty linked list."""

```

```
ai_4.py > ...
18 class SinglyLinkedList(Generic[T]):
28
29     def __init__(self) -> None:
30         """Initializes an empty linked list."""
31         self.head: Optional[Node[T]] = None
32
33     def insert(self, data: T) -> None:
34         """
35             Inserts a new node with the given data at the end of the list.
36
37             This method traverses the list to find the last node and appends
38             the new node there. If the list is empty, the new node becomes the head.
39             Time complexity: O(n) where n is the number of nodes.
40
41             Args:
42                 data (T): The data for the new node.
43             """
44
45             new_node = Node(data)
46             # If the list is empty, the new node becomes the head.
47             if self.head is None:
48                 self.head = new_node
49                 return
50
51             # Otherwise, traverse to the end of the list.
52             last_node = self.head
53             while last_node.next:
54                 last_node = last_node.next
55
56             # Append the new node at the end.
57             last_node.next = new_node
```

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

```
# 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
    list = SinglyLinkedList()
    list.append("A")
    list.append("B")
    list.append("C")
    list.append("D")
    list.append("E")
```

```

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

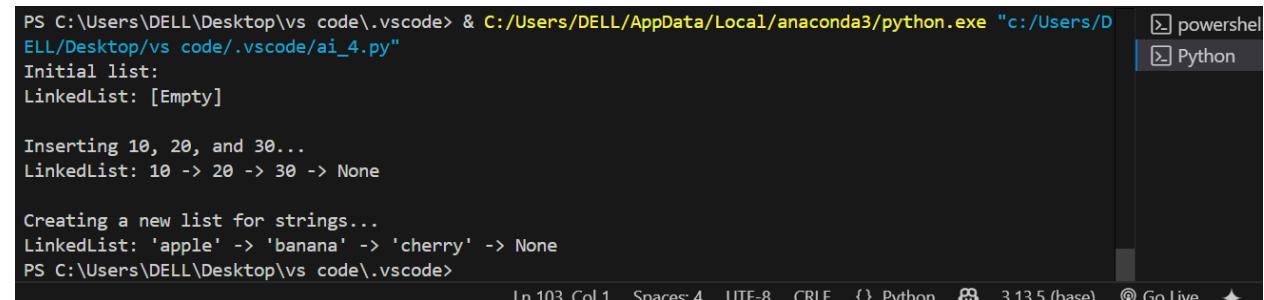
    # 1. Display the empty list
    print("Initial list:")
    sll.display() # Expected: LinkedList: [Empty]

    # 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:



```

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"
Initial list:
LinkedList: [Empty]

Inserting 10, 20, and 30...
LinkedList: 10 -> 20 -> 30 -> None

Creating a new list for strings...
LinkedList: 'apple' -> 'banana' -> 'cherry' -> None
PS C:\Users\DELL\Desktop\vs code\.vscode>

```

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

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, and 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 and strings) and serves as a simple effective test case.
- The `display` method's

to try

Key Observation & Potential Improvement

The most significant observation relates to the performance of the `insert` method.

- **Observation:** The current `insert` method adds new nodes to the *end* of the list. To do this, it must traverse the entire list from the `head` every single time, which results in an $O(n)$ time complexity. While correct, this can be inefficient for very long lists where many insertions are performed.
- **Suggestion:** A common optimization for linked lists is to keep track of the `tail` (the last node) in addition to the `head`. This allows for $O(1)$ insertions at the end of the list, as no traversal is required.

Prompts 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:

```
# arr.py > ...
1   from typing import TypeVar, Generic, Optional, List
2
3   # Define a generic type 'T' that can be ordered (e.g., int, float, str).
4   T = TypeVar('T')
5
6   class Node(Generic[T]):
7       """
8           A single node in a Binary Search Tree.
9
10          Attributes:
11              key (T): The value stored within the node.
12              left (Optional[Node[T]]): A reference to the left child node.
13              right (Optional[Node[T]]): A reference to the right child node.
14      """
15
16      def __init__(self, key: T):
17          self.key: T = key
18          self.left: Optional['Node[T]'] = None
19          self.right: Optional['Node[T]'] = None
20
21  class BinarySearchTree(Generic[T]):
22      """
23          A Binary Search Tree (BST) data structure.
24
25          This class manages a collection of nodes, ensuring the BST property
26          is maintained on insertion. It provides an in-order traversal method
27          to retrieve the keys in sorted order.
28
29          Attributes:
30              root (Optional[Node[T]]): The root node of the tree.
31      """
```

```
32     def __init__(self) -> None:
33         """Initializes an empty Binary Search Tree."""
34         self.root: Optional[Node[T]] = None
35
36     def insert(self, key: T) -> None:
37         """
38             Inserts a new key into the BST, maintaining the BST property.
39
40             If the tree is empty, the new key becomes the root. Otherwise, it
41             finds the correct position for the key and inserts it as a leaf node.
42             Duplicate keys are ignored.
43
44             Args:
45                 key (T): The key to be inserted into the tree.
46             """
47             if self.root is None:
48                 self.root = Node(key)
49             else:
50                 self._insert_recursive(self.root, key)
51
52     def _insert_recursive(self, current_node: Node[T], key: T) -> None:
53         """A private helper method for recursive insertion."""
54         if key < current_node.key:
55             if current_node.left is None:
56                 current_node.left = Node(key)
57             else:
58                 self._insert_recursive(current_node.left, key)
59             elif key > current_node.key:
60                 if current_node.right is None:
61                     current_node.right = Node(key)
62                 else:
63                     self._insert_recursive(current_node.right, key)
64             # If key == current_node.key, do nothing (ignore duplicate).
65
66     def in_order_traversal(self) -> List[T]:
67         """
68             Performs an in-order traversal of the tree.
69
70             In-order traversal visits nodes in the order: left subtree, root,
71             right subtree. For a BST, this results in a sorted list of the keys.
72
73             Returns:
74                 List[T]: A list of keys from the tree in ascending order.
75             """
76             result: List[T] = []
77             self._in_order_recursive(self.root, result)
78             return result
79
80     def _in_order_recursive(self, current_node: Optional[Node[T]], result: List[T]) -> None:
81         """A private helper method for recursive in-order traversal."""
82         if current_node is not None:
83             self._in_order_recursive(current_node.left, result)
84             result.append(current_node.key)
85             self._in_order_recursive(current_node.right, result)
```

```
ai1.py > ...
87  # --- Example Usage ---
88 if __name__ == "__main__":
89     # Create a new Binary Search Tree for integers
90     bst = BinarySearchTree[int]()
91
92     # 1. Insert items into the tree
93     # The order of insertion matters for the tree's structure.
94     nodes_to_insert = [50, 30, 70, 20, 40, 60, 80]
95     print(f"Inserting nodes: {nodes_to_insert}")
96     for node_key in nodes_to_insert:
97         bst.insert(node_key)
98
99     # 2. Perform an in-order traversal
100    # This should print the keys in sorted order.
101    print("\nPerforming in-order traversal...")
102    sorted_keys = bst.in_order_traversal()
103    print(f"Sorted keys: {sorted_keys}") # Expected: [20, 30, 40, 50, 60, 70, 80]
104
105    # 3. Demonstrate with strings
106    bst_str = BinarySearchTree[str]()
107    print("\n--- String BST Demo ---")
108    str_nodes = ["David", "Alice", "Charlie", "Bob", "Eve"]
109    print(f"Inserting nodes: {str_nodes}")
110    for name in str_nodes:
111        bst_str.insert(name)
112
113    sorted_names = bst_str.in_order_traversal()
114    print(f"Sorted names: {sorted_names}") # Expected: ['Alice', 'Bob', 'Charlie', 'David', 'Eve']
115
```

In 115 Col 1 Spaces: 4 UTE-8 CRLF {} Python 3.13.5 (base) @ Go Live

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>
```

In 115 Col 1 Spaces: 4 UTE-8 CRLF {} Python 3.13.5 (base) @

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 the recursive logic is a standard and highly readable pattern.
- **Good Documentation:** The



3. Usability:

- The `if __name__ == "__main__":` block is excellent. It serves as both a clear demonstration of how to use the class and as an effective set of informal tests for both integer and string keys.

Potential Improvements (Minor Suggestions)

The code is already of high quality. The following are suggestions for alternative approaches and for extending the class's functionality, which you might find useful.



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 > ...
1  from typing import TypeVar, Generic, List, Tuple, Optional, Any
2
3  # Define generic types for keys and values.
4  KT = TypeVar('KT')
5  VT = TypeVar('VT')
6
7  class HashTable(Generic[KT, VT]):
8      """
9          A basic implementation of a Hash Table (also known as a hash map).
10
11         This implementation uses chaining with Python lists to handle hash collisions.
12         It supports basic operations: insert, search, and delete.
13
14     Attributes:
15         size (int): The number of buckets in the hash table.
16         table (List[List[Tuple[KT, VT]]]): The internal table, where each
17                                         element is a "bucket" (a list
18                                         of key-value pairs).
19     """
20
21     def __init__(self, size: int = 101) -> None:
22         """
23             Initializes the hash table.
24
25         Args:
26             size (int): The size of the hash table. A prime number is
27                         often recommended to help distribute keys more uniformly.
28         """
29         if size <= 0:
30             raise ValueError("Hash table size must be positive.")
```

```
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.
    """
```

```
def insert(self, key: KT, value: VT) -> None:
    """
    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)
```

```
def search(self, key: KT) -> Optional[VT]:
    """
    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\> & 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 time.
- **Update on Insert:** The `insert`

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, explaining the purpose of the class, its methods, arguments, and return values. The comment about

3. Robustness and Usability:

- **Error Handling:** The `__init__` method includes a check to prevent a non-positive size, which is good practice.
- **User-Friendly Output:** The `__str__` method provides a very clear and readable representation of the hash table's internal state, which is incredibly helpful for debugging and understanding.
- **Comprehensive Example:** The `if __name__ == "__main__":` block is excellent. It thoroughly tests all functionalities, including insertion, searching, updating, and deletion, as well as handling

wants to try.

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:

```
ai_3.py > ...
1   from collections import defaultdict
2   from typing import TypeVar, Generic, Dict, List, Set
3
4   # Define a generic type 'T' for the vertices.
5   T = TypeVar('T')
6
7   class Graph(Generic[T]):
8       """
9           A basic implementation of an undirected Graph using an adjacency list.
10
11           The graph is represented by a dictionary where keys are vertices and
12           values are lists of their adjacent vertices.
13
14           Attributes:
15               adjacency_list (Dict[T, List[T]]): The underlying data structure
16                               representing the graph.
17       """
18
19       def __init__(self) -> None:
20           """Initializes an empty graph."""
21           self.adjacency_list: Dict[T, List[T]] = defaultdict(list)
22
23       def add_vertex(self, vertex: T) -> None:
24           """
25               Adds a vertex to the graph.
26
27               If the vertex already exists, this method does nothing.
28
29           Args:
30               vertex (T): The vertex to be added.
```

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

```
30     |     vertex (T): The vertex to be added.
31     | """
32     | # defaultdict handles this implicitly, but an explicit method is good practice.
33     | if vertex not in self.adjacency_list:
34     |     self.adjacency_list[vertex] = []
35
36     def add_edge(self, vertex1: T, vertex2: T) -> None:
37         """
38             Adds an undirected edge between two vertices.
39
40             If the vertices do not exist, they are automatically added to the graph.
41
42             Args:
43                 vertex1 (T): The first vertex of the edge.
44                 vertex2 (T): The second vertex of the edge.
45             """
46
47             # Add the edge from vertex1 to vertex2
48             self.adjacency_list[vertex1].append(vertex2)
49             # Add the edge from vertex2 to vertex1 (since it's an undirected graph)
50             self.adjacency_list[vertex2].append(vertex1)
51
52     def get_neighbors(self, vertex: T) -> List[T]:
53         """
54             Retrieves the list of neighbors for a given vertex.
55
56             Args:
57                 vertex (T): The vertex whose neighbors to retrieve.
58
59             Returns:
60                 List[T]: A list of adjacent vertices. Returns an empty list if
61                 | the vertex does not exist.
62             """
63
64             return self.adjacency_list.get(vertex, [])
65
66     def get_vertices(self) -> List[T]:
67         """
68             Returns a list of all vertices in the graph.
69
70             Returns:
71                 List[T]: All vertices in the graph.
72             """
73
74             return list(self.adjacency_list.keys())
75
76     def display(self) -> None:
77         """Prints the adjacency list representation of the graph."""
78         if not self.adjacency_list:
79             print("Graph is empty.")
80             return
81
82             print("Graph Adjacency List:")
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

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

```
51     def get_neighbors(self, vertex: T) -> List[T]:
52         """
53             Retrieves the list of neighbors for a given vertex.
54
55             Args:
56                 vertex (T): The vertex whose neighbors to retrieve.
57
58             Returns:
59                 List[T]: A list of adjacent vertices. Returns an empty list if
60                 | the vertex does not exist.
61             """
62
63             return self.adjacency_list.get(vertex, [])
64
65     def get_vertices(self) -> List[T]:
66         """
67             Returns a list of all vertices in the graph.
68
69             Returns:
70                 List[T]: All vertices in the graph.
71             """
72
73
74             return list(self.adjacency_list.keys())
75
76     def display(self) -> None:
77         """Prints the adjacency list representation of the graph."""
78         if not self.adjacency_list:
79             print("Graph is empty.")
80             return
81
82             print("Graph Adjacency List:")
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

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

```

75         print("Graph Adjacency List: ")
76         for vertex, neighbors in self.adjacency_list.items():
77             print(f" {vertex}: {neighbors}")
78
79     # --- Example Usage ---
80     if __name__ == "__main__":
81         # Create a new graph for strings (e.g., city names)
82         g = Graph[str]()
83
84         print("--- Adding vertices and edges ---")
85         g.add_vertex("New York")
86         g.add_vertex("Los Angeles")
87         g.add_vertex("Chicago")
88         g.add_vertex("Houston")
89         g.add_vertex("Phoenix")
90
91         g.add_edge("New York", "Chicago")
92         g.add_edge("Los Angeles", "Phoenix")
93         g.add_edge("Chicago", "Houston")
94         g.add_edge("New York", "Houston") # A direct flight!
95
96     # Add an edge where one vertex ('Denver') is new
97     g.add_edge("Chicago", "Denver")
98
99     g.display()
100
101     print("\n--- Getting vertices and neighbors ---")
102     print(f"All vertices in the graph: {g.get_vertices()}")
103
104
105
106
107
108
109
110

```

```

102     g.display()
103
104
105     print("\n--- Getting vertices and neighbors ---")
106     print(f"All vertices in the graph: {g.get_vertices()}")
107     print(f"Neighbors of 'Chicago': {g.get_neighbors('Chicago')}")"
108     print(f"Neighbors of 'Phoenix': {g.get_neighbors('Phoenix')}")"
109     print(f"Neighbors of 'New York': {g.get_neighbors('New York')}")"
110

```

Output:

The screenshot shows the VS Code interface with the terminal tab active. The terminal window displays the command run and the resulting output of the Python script.

```

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>
```

In 110 Col 1 Spaces: 4 LITE 8 CR LF {} Python 8 3.12.5 /base @ Col

Observations:

Strengths:

1. **Clarity and Readability:** The code is clean, well-commented, and the method names (`add_vertex`, `add_edge`, `get_neighbors`) are intuitive and self-explanatory.
2. **Correctness:** The logic for an undirected graph is correctly implemented. Adding an edge between `v1` and `v2` correctly updates the adjacency lists for both vertices.
3. **Use of `defaultdict`:** Using `collections.defaultdict(list)` is an excellent choice. It simplifies the `add_edge` method by automatically creating an entry for a new vertex, making the code more concise and robust.

robust.

4. Generics and Type Hinting: The use of

`TypeVar` and `Generic[T]` makes the `Graph` class flexible and type-safe. It can be used with any hashable vertex type (like `int`, `str`, or custom objects) while allowing static analysis tools to catch type-related errors.

5. Good Structure: The class is well-

organized, and the example usage block under `if __name__ == "__main__":` is a fantastic way to demonstrate functionality and make the file runnable for testing.

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 > ...
1  import heapq
2  from typing import TypeVar, Generic, List, Tuple
3
4  # Define a generic type 'T' for the items in the queue.
5  T = TypeVar('T')
6
7  class PriorityQueue(Generic[T]):
8      """
9          A Priority Queue implementation using Python's heapq module (min-heap).
10
11         Items are stored along with their priority. The item with the lowest
12         priority value is considered the highest priority and will be popped first.
13
14         Attributes:
15             _heap (List[Tuple[int, T]]): A private list that stores tuples of
16                                         (priority, item). This list is maintained
17                                         as a min-heap by the heapq module.
18
19
20     def __init__(self) -> None:
21         """Initializes an empty Priority Queue."""
22         self._heap: List[Tuple[int, T]] = []
23
24     def push(self, item: T, priority: int) -> None:
25         """
26             Adds an item to the queue with a specified priority.
27
28             The lower the priority number, the higher the priority.
29
30             Args:
```

```
30             item (T): The item to be added to the queue.
31             priority (int): The priority of the item.
32
33
34         # heapq pushes a tuple (priority, item) onto the heap.
35         # It will use the first element of the tuple (priority) for sorting.
36         heapq.heappush(self._heap, (priority, item))
37
38     def pop(self) -> T:
39         """
40             Removes and returns the item with the highest priority (lowest priority value).
41
42             Returns:
43                 T: The item with the highest priority.
44
45             Raises:
46                 IndexError: If pop is called on an empty queue.
47
48         if self.is_empty():
49             raise IndexError("pop from an empty priority queue")
50
51         # heappop removes and returns the smallest item from the heap.
52         # We return the second element of the tuple, which is the actual item.
53         priority, item = heapq.heappop(self._heap)
54         return item
55
56     def is_empty(self) -> bool:
57         """
```

```

55     def is_empty(self) -> bool:
56         """Checks if the priority queue is empty.
57
58         Returns:
59             bool: True if the queue is empty, False otherwise.
60             """
61         return not self._heap
62
63     def __len__(self) -> int:
64         """Returns the number of items in the priority queue.
65         Allows the use of the built-in len() function.
66
67         Returns:
68             int: The number of items in the queue.
69             """
70         return len(self._heap)
71
72
73     # --- Example Usage ---
74
75     if __name__ == "__main__":
76         # Create a new priority queue to manage tasks
77         # The items will be strings (task descriptions)
78         task_queue = PriorityQueue[str]()
79
80
81         # 1. Check if the new queue is empty
82         print(f"Is the queue empty? {task_queue.is_empty()}") # Expected: True
83
84         # 2. Push tasks with different priorities
85         # Priority 1 is the highest. 5 is the lowest.

```

```

86         # ...Priority 1 is the highest, 5 is the lowest.
87         print("\nAdding tasks with priorities...")
88         task_queue.push("Write report", 3)
89         task_queue.push("Answer urgent emails", 1)
90         task_queue.push("Fix critical bug", 1) # Same priority as another task
91         task_queue.push("Schedule team meeting", 4)
92         task_queue.push("Refill coffee", 5)
93
94         print(f"Current queue size: {len(task_queue)}")
95         print(f"Is the queue empty? {task_queue.is_empty()}") # Expected: False
96
97         # 3. Pop tasks from the queue
98         # They should come out in order of their priority number (lowest first).
99         print("\nProcessing tasks in order of priority:")
100
101         # The order of items with the same priority (like the two priority 1 tasks)
102         # is not guaranteed by heapq, but they will both come out before priority 3.
103         task1 = task_queue.pop()
104         print(f"Processing (Priority 1): '{task1}'")
105
106         task2 = task_queue.pop()
107         print(f"Processing (Priority 1): '{task2}'")
108
109         task3 = task_queue.pop()
110         print(f"Processing (Priority 3): '{task3}'")
111
112         print(f"\nRemaining tasks in queue: {len(task_queue)}")
113
114         # 4. Pop all remaining tasks
115         while not task_queue.is_empty():
116             ...

```

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

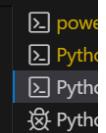
Adding tasks with priorities...
Current queue size: 5
Is the queue empty? False

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 needs
to think in terms of items and priorities.

3. **Excellent Typing and Generics:** The use
of `TypeVar` and `Generic[T]` makes the
class robust, reusable, and type-safe. It

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 is a perfect example of how to use the class. It tests all the main features, including handling items with the same priority and makes the file runnable for

The code is already very strong, but for certain advanced use cases, a couple of enhancements could be considered.

1. Stable Ordering for Equal Priorities:

As noted in the example usage comments, `heapq` does not guarantee the order of items that have the same priority. If two items are pushed with priority 1, their pop order is undefined. In scenarios where a "First-In, First-Out" (FIFO) order is desired for items of equal priority, you can introduce a counter.

By storing items as a 3-element tuple `(priority, sequence_number, item)`, you can ensure that if priorities are equal, the item that was pushed first will be popped first.

Task Description #8

Deque

Task: Use AI to implement a double-ended queue using `collections.deque`.

Sample Input Code:

```
class DequeDS:
```

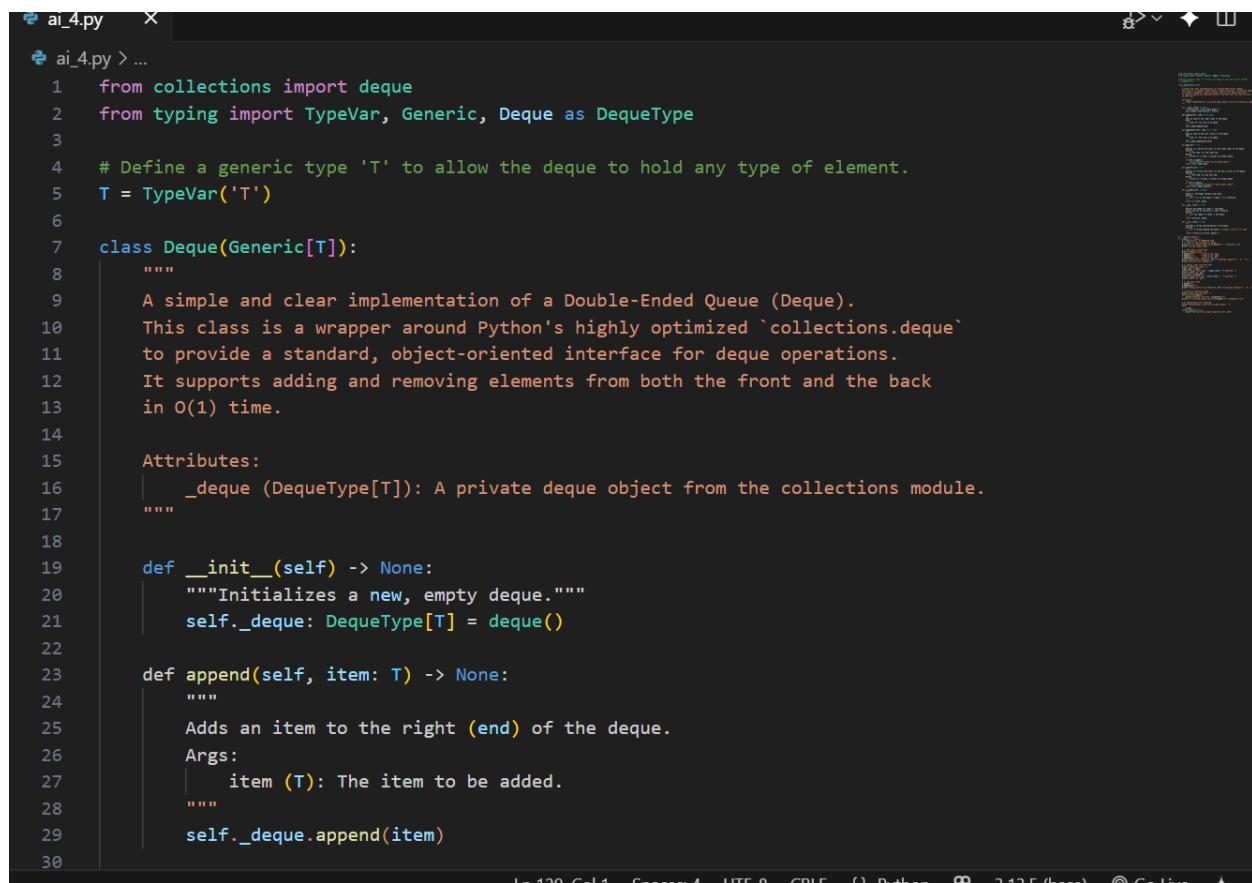
```
    pass
```

Expected Output:

- Insert and remove from both ends with docstrings.

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

Code:



The screenshot shows a code editor window with a dark theme. The file is named 'ai_4.py'. The code implements a generic double-ended queue (Dequeue) using Python's built-in 'collections.deque' module. It includes type hints from the 'typing' module and a detailed docstring explaining the class's purpose, attributes, and methods. The code editor has a status bar at the bottom with various icons and text.

```
ai_4.py > ...
1  from collections import deque
2  from typing import TypeVar, Generic, Deque as DequeType
3
4  # Define a generic type 'T' to allow the deque to hold any type of element.
5  T = TypeVar('T')
6
7  class Deque(Generic[T]):
8      """
9          A simple and clear implementation of a Double-Ended Queue (Deque).
10         This class is a wrapper around Python's highly optimized `collections.deque`
11         to provide a standard, object-oriented interface for deque operations.
12         It supports adding and removing elements from both the front and the back
13         in O(1) time.
14
15         Attributes:
16             _deque (DequeType[T]): A private deque object from the collections module.
17         """
18
19     def __init__(self) -> None:
20         """Initializes a new, empty deque."""
21         self._deque: DequeType[T] = deque()
22
23     def append(self, item: T) -> None:
24         """
25             Adds an item to the right (end) of the deque.
26             Args:
27                 item (T): The item to be added.
28             """
29         self._deque.append(item)
30
```

```
29     self._deque.append(item)
30
31     def appendleft(self, item: T) -> None:
32         """
33             Adds an item to the left (front) of the deque.
34             Args:
35                 item (T): The item to be added.
36             """
37         self._deque.appendleft(item)
38
39     def pop(self) -> T:
40         """
41             Removes and returns the item from the right (end) of the deque.
42             Returns:
43                 T: The item from the right end.
44             Raises:
45                 IndexError: If pop is called on an empty deque.
46             """
47         if self.is_empty():
48             raise IndexError("pop from an empty deque")
49         return self._deque.pop()
50
51     def popleft(self) -> T:
52         """
53             Removes and returns the item from the left (front) of the deque.
54             Returns:
55                 T: The item from the left end.
56             Raises:
57                 IndexError: If pop is called on an empty deque.
```

```
51     def popleft(self) -> T:
52         """
53             Raises:
54                 IndexError: If pop is called on an empty deque.
55             """
56
57         if self.is_empty():
58             raise IndexError("popleft from an empty deque")
59         return self._deque.popleft()
60
61     def is_empty(self) -> bool:
62         """
63             Checks if the deque contains any items.
64             Returns:
65                 bool: True if the deque is empty, False otherwise.
66             """
67
68         return not self._deque
69
70     def __len__(self) -> int:
71         """
72             Returns the number of items in the deque.
73             Allows the use of the built-in len() function.
74             Returns:
75                 int: The number of items in the deque.
76             """
77
78         return len(self._deque)
79
80     def __str__(self) -> str:
81         """
82             Provides a string representation of the deque.
83             Returns:
84                 str: A string showing the deque's contents from left to right.
```

```
85     """
86     return f"Deque({list(self._deque)})"
87
88 # --- Example Usage ---
89 if __name__ == "__main__":
90     d = Deque[str]() # CORRECTED LINE
91     # 1. Check if the new deque is empty
92     print(f"Is the deque empty? {d.is_empty()}") # Expected: True
93     print(f"Initial deque: {d}")
94
95     # 2. Add items to both ends
96     print("\nAdding items...")
97     d.append("B") # Add to the right
98     d.append("C") # Add to the right
99     d.appendleft("A") # Add to the left
100    print(f"Deque after additions: {d}") # Expected: Deque(['A', 'B', 'C'])
101    print(f"Current size: {len(d)}")
102
103    # 3. Remove items from both ends
104    print("\nRemoving items...")
105    right_item = d.pop()
106    print(f"Popped from right: '{right_item}'") # Expected: 'C'
107    print(f"Deque now: {d}")
108    left_item = d.popleft()
109    print(f"Popped from left: '{left_item}'") # Expected: 'A'
110    print(f"Deque now: {d}")
111
112    # 4. Add more items
113    d.append("X")
114    d.appendleft("Y")
115    print(f"\nDeque after more additions: {d}") # Expected: Deque(['Y', 'B', 'X'])
116
117    # 5. Pop all remaining items
118    print("\nClearing the deque...")
119    while not d.is_empty():
120        print(f"Popping from left: {d.popleft()}")
121    print(f"Is the deque empty now? {d.is_empty()}") # Expected: True
122
123    # 6. Demonstrate error handling
124    print("\nAttempting to pop from an empty deque...")
125    try:
126        d.pop()
127    except IndexError as e:
128        print(f"Successfully caught expected error: {e}")
129
```

```
112    # 4. Add more items
113    d.append("X")
114    d.appendleft("Y")
115    print(f"\nDeque after more additions: {d}") # Expected: Deque(['Y', 'B', 'X'])
116
117    # 5. Pop all remaining items
118    print("\nClearing the deque...")
119    while not d.is_empty():
120        print(f"Popping from left: {d.popleft()}")
121    print(f"Is the deque empty now? {d.is_empty()}") # Expected: True
122
123    # 6. Demonstrate error handling
124    print("\nAttempting to pop from an empty deque...")
125    try:
126        d.pop()
127    except IndexError as e:
128        print(f"Successfully caught expected error: {e}")
129
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    FORTS

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 deque empty? True
Initial deque: Deque([])

Adding items...
Deque after additions: Deque(['A', 'B', 'C'])
Current size: 3

Removing items...
Popped from right: 'C'

Ln 129, Col 1  Spaces: 4  UTF-8  CRLF  {} Python  ⚙  3.13.5 (base)  ⌂ G
```

```
Deque now: Deque(['A', 'B'])
Popped from left: 'A'
Deque now: Deque(['B'])

Deque after more additions: Deque(['Y', 'B', 'X'])

Clearing the deque...
Popping from left: Y
Popping from left: B
Popping from left: X
Is the deque empty now? True
```

```
Popping from left: X
Is the deque empty now? True

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

Ln 129, Col 1 Spaces: 4 UTF-8 CRLF {} Python ⚙ 3.13.5 (base)

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

The implementation is already very solid for its intended purpose. However, we could add a couple of common "quality of life" methods that are often found in collection-style classes to make it even more versatile.

1. **Peek Functionality:** It's often useful to be able to see the element at either end of the deque *without* removing it. We can add `peek()` and `peekleft()` methods for this.
2. **Clear Method:** A `clear()` method to remove all items from the deque at once can be more convenient and explicit than looping with `pop`.



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 > ...
1  import pandas as pd
2  from tabulate import tabulate
3
4  def create_data_structures_comparison():
5      """
6          Creates a comprehensive comparison table of different data structures
7          with their time complexities for various operations.
8      """
9
10     # Define the data structures and their time complexities
11     data = {
12         'Data Structure': [
13             'Array (Static)',
14             'Array (Dynamic)',
15             'Linked List (Singly)',
16             'Linked List (Doubly)',
17             'Stack (Array-based)',
18             'Stack (Linked List-based)',
19             'Queue (Array-based)',
20             'Queue (Linked List-based)',
21             'Deque (Double-ended Queue)',
22             'Hash Table (Average)',
23             'Hash Table (Worst)',
24             'Binary Search Tree (Average)',
25             'Binary Search Tree (Worst)',
26             'AVL Tree'.
27
28         ],
29         'Red-Black Tree',
30         'B-Tree',
31         'Heap (Binary)',
32         'Trie',
33         'Graph (Adjacency List)',
34         'Graph (Adjacency Matrix)'
35     ],
36     'Access': [
37         'O(1)',      # Array (Static)
38         'O(1)',      # Array (Dynamic)
39         'O(n)',      # Linked List (Singly)
40         'O(n)',      # Linked List (Doubly)
41         'O(n)',      # Stack (Array-based)
42         'O(n)',      # Stack (Linked List-based)
43         'O(n)',      # Queue (Array-based)
44         'O(n)',      # Queue (Linked List-based)
45         'O(n)',      # Deque
46         'O(1)',      # Hash Table (Average)
47         'O(n)',      # Hash Table (Worst)
48         'O(log n)',  # BST (Average)
49         'O(n)',      # BST (Worst)
50         'O(log n)',  # AVL Tree
51         'O(log n)',  # Red-Black Tree
52         'O(log n)',  # B-Tree
53         'None'       # None
```

```

4   def create_data_structures_comparison():
50      'O(log n)',    # B-Tree
51      'N/A',        # Heap
52      'O(m)',       # Trie ( $m = \text{key Length}$ )
53      'O(V + E)',   # Graph (Adjacency List)
54      'O(1)'        # Graph (Adjacency Matrix)
55
56  ],
57  'Search': [
58      'O(n)',       # Array (Static)
59      'O(n)',       # Array (Dynamic)
60      'O(n)',       # Linked List (Singly)
61      'O(n)',       # Linked List (Doubly)
62      'O(n)',       # Stack (Array-based)
63      'O(n)',       # Stack (Linked List-based)
64      'O(n)',       # Queue (Array-based)
65      'O(n)',       # Queue (Linked List-based)
66      'O(n)',       # Deque
67      'O(1)',       # Hash Table (Average)
68      'O(n)',       # Hash Table (Worst)
69      'O(log n)',   # BST (Average)
70      'O(n)',       # BST (Worst)
71      'O(log n)',   # AVL Tree
72      'O(log n)',   # Red-Black Tree
73      'O(log n)',   # B-Tree
74      'O(n)',       # Heap
75      '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

+ ⌂ ⋮

```

    ],
    'Deletion': [
101     'N/A',           # Array (Static)
102     'O(n)',          # Array (Dynamic)
103     'O(1)',          # Linked List (Singly) - if node known
104     'O(1)',          # Linked List (Doubly) - if node known
105     'O(1)',          # Stack (Array-based)
106     'O(1)',          # Stack (Linked List-based)
107     'O(1)',          # Queue (Array-based)
108     'O(1)',          # Queue (Linked List-based)
109     'O(1)',          # Deque
110     'O(1)',          # Hash Table (Average)
111     'O(n)',          # Hash Table (Worst)
112     'O(log n)',      # BST (Average)
113     'O(n)',          # BST (Worst)
114     'O(log n)',      # AVL Tree
115     'O(log n)',      # Red-Black Tree
116     'O(log n)',      # B-Tree
117     'O(log n)',      # Heap
118     'O(m)',          # Trie
119     'O(V + E)',      # Graph (Adjacency List)
120     'O(V2)'        # Graph (Adjacency Matrix)
121   ],
    'Space Complexity': [
122     'O(n)',          # Array (Static)

```

```

123   ],
    'Space Complexity': [
124     'O(n)',          # Array (Static)
125     'O(n)',          # Array (Dynamic)
126     'O(n)',          # Linked List (Singly)
127     'O(n)',          # Linked List (Doubly)
128     'O(n)',          # Stack (Array-based)
129     'O(n)',          # Stack (Linked List-based)
130     'O(n)',          # Queue (Array-based)
131     'O(n)',          # Queue (Linked List-based)
132     'O(n)',          # Deque
133     'O(n)',          # Hash Table (Average)
134     'O(n)',          # Hash Table (Worst)
135     'O(n)',          # BST (Average)
136     'O(n)',          # BST (Worst)
137     'O(n)',          # AVL Tree
138     'O(n)',          # Red-Black Tree
139     'O(n)',          # B-Tree
140     'O(n)',          # Heap
141     'O(ALPHABET_SIZE * N * M)', # Trie
142     'O(V + E)',      # Graph (Adjacency List)
143     'O(V2)'        # Graph (Adjacency Matrix)
144   ]
145
146   # Create DataFrame

```

```

145     def create_data_structures_comparison():
146
147         # Create DataFrame
148         df = pd.DataFrame(data)
149
150     return df
151
151 def display_comparison_table():
152     """
153     Display the data structures comparison table in a formatted way.
154     """
155     df = create_data_structures_comparison()
156
157     print("=" * 120)
158     print("DATA STRUCTURES TIME COMPLEXITY COMPARISON TABLE".center(120))
159     print("=" * 120)
160
161     # Display using tabulate for better formatting
162     print(tabulate(df, headers='keys', tablefmt='grid', showindex=False))
163
164     print("\n" + "=" * 120)
165     print("LEGEND AND NOTES".center(120))
166     print("=" * 120)
167
168     notes = [
169         "- O(1) - Constant time"

```

```

151     def display_comparison_table():
152
153         notes = [
154             "- O(1) - Constant time",
155             "- O(log n) - Logarithmic time",
156             "- O(n) - Linear time",
157             "- O(n log n) - Linearithmic time",
158             "- O(n²) - Quadratic time",
159             "- V = Number of vertices, E = Number of edges",
160             "- m = Length of the key/string",
161             "- N/A = Operation not applicable or not typically supported",
162             "",
163             "IMPORTANT NOTES:",
164             "- Linked List insertion/deletion assumes you have a reference to the node",
165             "- Hash Table performance depends on hash function quality and load factor",
166             "- BST performance degrades to O(n) in worst case (unbalanced tree)",
167             "- AVL and Red-Black trees maintain balance, guaranteeing O(log n) operations",
168             "- Trie space complexity: ALPHABET_SIZE * N * M where N = number of words, M = average length",
169             "- Array access is by index; search is for finding a specific value",
170             "- Stack/Queue operations refer to push/pop and enqueue/dequeue respectively"
171         ]
172
173         for note in notes:
174             print(note)
175
176     return df

```

```
ai_1.py > ...
4  def create_data_structures_comparison():
5      # Create DataFrame
6      df = pd.DataFrame(data)
7
8      return df
9
10
11 def display_comparison_table():
12     """
13     Display the data structures comparison table in a formatted way.
14     """
15
16     df = create_data_structures_comparison()
17
18     print("=" * 120)
19     print("DATA STRUCTURES TIME COMPLEXITY COMPARISON TABLE".center(120))
20     print("=" * 120)
21
22     # Display using tabulate for better formatting
23     print(tabulate(df, headers='keys', tablefmt='grid', showindex=False))
24
25
26     print("\n" + "=" * 120)
27     print("LEGEND AND NOTES".center(120))
28     print("=" * 120)
29
30
31     notes = [
32         "* O(1) - Constant time",
33         "* O( $n^2$ ) - Quadratic time"
34     ]
```

```
151 def display_comparison_table():
152     notes = [
153         "• O(1) - Constant time",
154         "• O(log n) - Logarithmic time",
155         "• O(n) - Linear time",
156         "• O(n log n) - Linearithmic time",
157         "• O(n2) - Quadratic time",
158         "• V = Number of vertices, E = Number of edges",
159         "• m = Length of the key/string",
160         "• N/A = Operation not applicable or not typically supported",
161         "",
162         "IMPORTANT NOTES:",
163         "• Linked List insertion/deletion assumes you have a reference to the node",
164         "• Hash Table performance depends on hash function quality and load factor",
165         "• BST performance degrades to O(n) in worst case (unbalanced tree)",
166         "• AVL and Red-Black trees maintain balance, guaranteeing O(log n) operations",
167         "• Trie space complexity: ALPHABET_SIZE * N * M where N = number of words, M = average length",
168         "• Array access is by index; search is for finding a specific value",
169         "• Stack/Queue operations refer to push/pop and enqueue/dequeue respectively"
170     ]
171
172     for note in notes:
173         print(note)
174
175     return df
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
```

```
ai_1.py > ...
151     def display_comparison_table():
152         return df
153
154     def create_detailed_comparison():
155         """
156             Create a more detailed comparison with additional properties.
157         """
158         detailed_data = {
159             'Data Structure': [
160                 'Array',
161                 'Linked List',
162                 'Stack',
163                 'Queue',
164                 'Hash Table',
165                 'Binary Search Tree',
166                 'AVL Tree',
167                 'Heap',
168                 'Trie',
169                 'Graph'
170             ],
171             'Best Use Cases': [
172                 'Random access, mathematical operations, cache-friendly',
173                 'Frequent insertions/deletions, unknown size',
174                 'LIFO operations, function calls, undo operations',
175                 'FIFO operations, scheduling, buffering',
176             ]
177         }
178
179         return detailed_data
```

Problems Output Debug Console Terminal Ports

+ × ↻ ⌂ ⌄ X

```
193     def create_detailed_comparison():
194         'Best Use Cases': [
195             'Random access, mathematical operations, cache-friendly',
196             'Frequent insertions/deletions, unknown size',
197             'LIFO operations, function calls, undo operations',
198             'FIFO operations, scheduling, buffering',
199             'Fast lookups, caching, database indexing',
200             'Sorted data, range queries, in-order traversal',
201             'Guaranteed balanced operations, real-time systems',
202             'Priority queues, heap sort, finding min/max',
203             'Autocomplete, spell checkers, IP routing',
204             'Networks, relationships, pathfinding'
205         ],
206         'Advantages': [
207             'Fast access, memory efficient, cache-friendly',
208             'Dynamic size, efficient insertion/deletion',
209             'Simple LIFO semantics, memory efficient',
210             'Fair scheduling, simple FIFO semantics',
211             'Average O(1) operations, flexible keys',
212             'Sorted order, efficient range operations',
213             'Guaranteed balance, predictable performance',
214             'Efficient priority operations, in-place sorting',
215             'Prefix matching, memory efficient for strings',
216             'Flexible relationships, various algorithms'
217         ],
218         'Disadvantages': [
219             'Memory overhead for pointers',
220             'Complex insertion/deletion logic',
221             'Inefficient search for unsorted data',
222             'Space complexity for large datasets',
223             'Inconsistent performance across different operations',
224             'Sensitivity to initial state for some operations',
225             'Higher memory usage compared to arrays',
226             'Less cache-friendly than arrays for certain operations',
227             'More complex memory management required',
228             'Inability to efficiently support multiple inheritance or polymorphism'
229         ]
230     ],
231     'Disadvantages': [
232         'Memory overhead for pointers',
233         'Complex insertion/deletion logic',
234         'Inefficient search for unsorted data',
235         'Space complexity for large datasets',
236         'Inconsistent performance across different operations',
237         'Sensitivity to initial state for some operations',
238         'Higher memory usage compared to arrays',
239         'Less cache-friendly than arrays for certain operations',
240         'More complex memory management required',
241         'Inability to efficiently support multiple inheritance or polymorphism'
242     ]
243 ]
```

```

193 def create_detailed_comparison():
194     'Disadvantages': [
195         'Fixed size (static), expensive insertion/deletion',
196         'No random access, extra memory overhead',
197         'Limited access pattern, potential stack overflow',
198         'Limited access pattern, no random access',
199         'Worst-case O(n), memory overhead, no ordering',
200         'Can become unbalanced, worst-case O(n)',
201         'Complex implementation, higher constants',
202         'No efficient search, limited structure',
203         'Memory overhead, complex deletion',
204         'Memory overhead, complex algorithms'
205     ]
206 }
207
208 detailed_df = pd.DataFrame(detailed_data)
209
210 print("\n" + "=" * 150)
211 print("DETAILED DATA STRUCTURES COMPARISON".center(150))
212 print("=" * 150)
213
214 # Print each row separately for better readability
215 for index, row in detailed_df.iterrows():
216     print(f"\n{row['Data Structure'].upper()}")
217     print("-" * len(row['Data Structure']))
218     print(f"Best Use Cases: {row['Best Use Cases']}")

```

```

5     "create_detailed_comparison": [
6         for index, row in detailed_df.iterrows():
7             print(f"\n{row['Data Structure'].upper()}")
8             print("-" * len(row['Data Structure']))
9             print(f"Best Use Cases: {row['Best Use Cases']}")
10            print(f"Advantages: {row['Advantages']}")
11            print(f"Disadvantages: {row['Disadvantages']}")
12
13        return detailed_df
14
15 if __name__ == "__main__":
16     # Display the main comparison table
17     df = display_comparison_table()
18
19     # Display detailed comparison
20     detailed_df = create_detailed_comparison()
21
22     # Save to CSV for future reference
23     df.to_csv('data_structures_comparison.csv', index=False)
24     detailed_df.to_csv('data_structures_detailed.csv', index=False)
25
26     print(f"\n{'*50}'")
27     print("Tables saved to CSV files for future reference!")
28     print("• data_structures_comparison.csv")
29     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)	$O(1)$	$O(V)$	$O(1)$	$O(V^2)$	$O(V^2)$

Python
 Python

LEGEND AND NOTES

- =====
- $O(1)$ - Constant time
- $O(\log n)$ - Logarithmic time
- $O(n)$ - Linear time
- $O(n \log n)$ - Linearithmic time
- $O(n^2)$ - 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

Ctrl+K to generate a command

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 table results in a highly readable and professional-looking console output.
The centered headers and separators are a nice touch.

3. Clear and Professional Output: The

use of the `tabulate` library for the complexity table and the custom formatted print for the detailed table results in a highly readable and professional-looking console output. The centered headers and separators are a nice touch.

4. Good Use of Libraries: Leveraging

`pandas` to structure the data is a great choice. It simplifies data management and makes it trivial to export the tables to a persistent format like CSV.

5. Persistence: Saving the generated

tables to CSV files is a thoughtful feature, allowing the user to reference the data later without re-running the script.

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 > ...
1  from typing import Dict, Optional
2
3  class EventRegistration:
4      """
5          Manages participant registration for an event using a hash table.
6
7          This system allows for efficient addition, removal, and lookup of
8          participants by their unique student ID.
9
10     Attributes:
11         event_name (str): The name of the event.
12         participants (Dict[int, str]): A hash table (dictionary) mapping
13             |           |           |
14             |           |           student IDs to student names.
15
16     def __init__(self, event_name: str):
17         """
18             Initializes a new registration system for a given event.
19
20         Args:
21             event_name (str): The name of the event.
22
23         if not event_name:
24             raise ValueError("Event name cannot be empty.")
25         self.event_name: str = event_name
26         self.participants: Dict[int, str] = {}
27         print(f"Registration system created for event: '{self.event_name}'")
28
29     def register_student(self, student_id: int, student_name: str) -> None:
30         """

```

In 122 Col 1 Spaces: 4 UTF-8 CRLF ⓘ Python 3.13.5 (base) ⌂ Go Live ▲ ⌂

```

3  class EventRegistration:
4
5      def register_student(self, student_id: int, student_name: str) -> None:
6          """
7              Registers a student for the event.
8
9              If the student is already registered, it prints a message.
10             This operation has an average time complexity of O(1).
11
12         Args:
13             student_id (int): The unique ID of the student.
14             student_name (str): The name of the student.
15
16         if student_id in self.participants:
17             print(f"-> Warning: Student {student_name} (ID: {student_id}) is already registered.")
18         else:
19             self.participants[student_id] = student_name
20             print(f"-> Success: Registered {student_name} (ID: {student_id}) for '{self.event_name}'")
21
22     def remove_student(self, student_id: int) -> None:
23         """
24
25             Removes a student from the event registration.
26
27             If the student is not found, it prints a message.
28             This operation has an average time complexity of O(1).
29
30         Args:
31             student_id (int): The ID of the student to remove.
32
33         if student_id in self.participants:
34             del self.participants[student_id]
35             print(f"-> Success: Removed {student_name} (ID: {student_id}) from '{self.event_name}'")
36
37     def lookup_student(self, student_id: int) -> str:
38         """
39
40             Looks up a student by their ID and returns their name.
41             If the student is not found, it prints a message.
42             This operation has an average time complexity of O(1).
43
44         if student_id in self.participants:
45             return self.participants[student_id]
46         else:
47             print(f"-> Warning: Student {student_id} was not found in the registration system.")
48
49     def print_all_students(self):
50         """
51
52             Prints all registered students and their IDs.
53
54         for student_id, student_name in self.participants.items():
55             print(f"Student {student_name} (ID: {student_id}) is registered for '{self.event_name}'")
56
57     def print_event_info(self):
58         """
59
60             Prints information about the event.
61
62         print(f"Event '{self.event_name}' is currently active and has {len(self.participants)} registered participants.")
63
64     def __str__(self) -> str:
65         """
66
67             Returns a string representation of the event registration system.
68
69         return f"Event Registration System for {self.event_name}: {self.participants}"
70
71     def __repr__(self) -> str:
72         """
73
74             Returns a formal string representation of the event registration system.
75
76         return f"EventRegistration(event_name={self.event_name}, participants={self.participants})"
77
78     def __eq__(self, other):
79         """
80
81             Checks if two event registration systems are equal based on their event names and participant dictionaries.
82
83         if isinstance(other, EventRegistration):
84             return self.event_name == other.event_name and self.participants == other.participants
85         return False
86
87     def __hash__(self):
88         """
89
90             Returns a hash value for the event registration system.
91
92         return hash((self.event_name, tuple(self.participants.items())))
93
94     def __len__(self):
95         """
96
97             Returns the number of registered participants.
98
99         return len(self.participants)
100
101    def __iter__(self):
102        """
103
104            Returns an iterator over the registered participants.
105
106        for student_id, student_name in self.participants.items():
107            yield (student_id, student_name)
108
109    def __getitem__(self, student_id):
110        """
111
112            Returns the name of the student with the specified ID.
113
114        if student_id in self.participants:
115            return self.participants[student_id]
116        raise KeyError(f"Student with ID {student_id} not found in registration system.")
117
118    def __setitem__(self, student_id, student_name):
119        """
120
121            Adds a new student to the registration system.
122
123        self.register_student(student_id, student_name)
124
125    def __delitem__(self, student_id):
126        """
127
128            Removes a student from the registration system.
129
130        self.remove_student(student_id)
131
132    def __contains__(self, student_id):
133        """
134
135            Checks if a student with the specified ID is registered.
136
137        return student_id in self.participants
138
139    def __add__(self, other):
140        """
141
142            Adds another event registration system to this one.
143
144        merged_system = EventRegistration(self.event_name)
145        merged_system.participants.update(other.participants)
146        return merged_system
147
148    def __iadd__(self, other):
149        """
150
151            Adds another event registration system to this one in place.
152
153        self += other
154        return self
155
156    def __sub__(self, other):
157        """
158
159            Subtracts another event registration system from this one.
160
161        merged_system = EventRegistration(self.event_name)
162        merged_system.participants = {student_id: name for (student_id, name) in self.participants if student_id not in other.participants}
163        return merged_system
164
165    def __isub__(self, other):
166        """
167
168            Subtracts another event registration system from this one in place.
169
170        self -= other
171        return self
172
173    def __mul__(self, count):
174        """
175
176            Multiplies the current event registration system by a given count.
177
178        merged_system = EventRegistration(self.event_name)
179        merged_system.participants = {student_id: name for (student_id, name) in self.participants * count}
180        return merged_system
181
182    def __imul__(self, count):
183        """
184
185            Multiplies the current event registration system by a given count in place.
186
187        self *= count
188        return self
189
190    def __truediv__(self, count):
191        """
192
193            Divides the current event registration system by a given count.
194
195        merged_system = EventRegistration(self.event_name)
196        merged_system.participants = {student_id: name for (student_id, name) in self.participants // count}
197        return merged_system
198
199    def __itruediv__(self, count):
200        """
201
202            Divides the current event registration system by a given count in place.
203
204        self /= count
205        return self
206
207    def __floordiv__(self, count):
208        """
209
210            Floor divides the current event registration system by a given count.
211
212        merged_system = EventRegistration(self.event_name)
213        merged_system.participants = {student_id: name for (student_id, name) in self.participants // count}
214        return merged_system
215
216    def __ifloordiv__(self, count):
217        """
218
219            Floor divides the current event registration system by a given count in place.
220
221        self //= count
222        return self
223
224    def __mod__(self, count):
225        """
226
227            Returns the remainder of the current event registration system divided by a given count.
228
229        merged_system = EventRegistration(self.event_name)
230        merged_system.participants = {student_id: name for (student_id, name) in self.participants % count}
231        return merged_system
232
233    def __imod__(self, count):
234        """
235
236            Returns the remainder of the current event registration system divided by a given count in place.
237
238        self %= count
239        return self
240
241    def __lshift__(self, count):
242        """
243
244            Shifts the current event registration system left by a given count.
245
246        merged_system = EventRegistration(self.event_name)
247        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
248        return merged_system
249
250    def __ilshift__(self, count):
251        """
252
253            Shifts the current event registration system left by a given count in place.
254
255        self <<= count
256        return self
257
258    def __rshift__(self, count):
259        """
260
261            Shifts the current event registration system right by a given count.
262
263        merged_system = EventRegistration(self.event_name)
264        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
265        return merged_system
266
267    def __irshift__(self, count):
268        """
269
270            Shifts the current event registration system right by a given count in place.
271
272        self >>= count
273        return self
274
275    def __and__(self, other):
276        """
277
278            Returns the intersection of the current event registration system and another one.
279
280        merged_system = EventRegistration(self.event_name)
281        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
282        return merged_system
283
284    def __iand__(self, other):
285        """
286
287            Returns the intersection of the current event registration system and another one in place.
288
289        self &=amp; other
290        return self
291
292    def __or__(self, other):
293        """
294
295            Returns the union of the current event registration system and another one.
296
297        merged_system = EventRegistration(self.event_name)
298        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
299        return merged_system
300
301    def __ior__(self, other):
302        """
303
304            Returns the union of the current event registration system and another one in place.
305
306        self |=amp; other
307        return self
308
309    def __xor__(self, other):
310        """
311
312            Returns the symmetric difference of the current event registration system and another one.
313
314        merged_system = EventRegistration(self.event_name)
315        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
316        return merged_system
317
318    def __ixor__(self, other):
319        """
320
321            Returns the symmetric difference of the current event registration system and another one in place.
322
323        self ^=amp; other
324        return self
325
326    def __invert__(self):
327        """
328
329            Returns the complement of the current event registration system.
330
331        merged_system = EventRegistration(self.event_name)
332        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
333        return merged_system
334
335    def __iinvert__(self):
336        """
337
338            Returns the complement of the current event registration system in place.
339
340        self ^=amp; other
341        return self
342
343    def __lshift__(self, count):
344        """
345
346            Shifts the current event registration system left by a given count.
347
348        merged_system = EventRegistration(self.event_name)
349        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
350        return merged_system
351
352    def __ilshift__(self, count):
353        """
354
355            Shifts the current event registration system left by a given count in place.
356
357        self <<= count
358        return self
359
360    def __rshift__(self, count):
361        """
362
363            Shifts the current event registration system right by a given count.
364
365        merged_system = EventRegistration(self.event_name)
366        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
367        return merged_system
368
369    def __irshift__(self, count):
370        """
371
372            Shifts the current event registration system right by a given count in place.
373
374        self >>= count
375        return self
376
377    def __and__(self, other):
378        """
379
380            Returns the intersection of the current event registration system and another one.
381
382        merged_system = EventRegistration(self.event_name)
383        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
384        return merged_system
385
386    def __iand__(self, other):
387        """
388
389            Returns the intersection of the current event registration system and another one in place.
390
391        self &=amp; other
392        return self
393
394    def __or__(self, other):
395        """
396
397            Returns the union of the current event registration system and another one.
398
399        merged_system = EventRegistration(self.event_name)
400        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
401        return merged_system
402
403    def __ior__(self, other):
404        """
405
406            Returns the union of the current event registration system and another one in place.
407
408        self |=amp; other
409        return self
410
411    def __xor__(self, other):
412        """
413
414            Returns the symmetric difference of the current event registration system and another one.
415
416        merged_system = EventRegistration(self.event_name)
417        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
418        return merged_system
419
420    def __ixor__(self, other):
421        """
422
423            Returns the symmetric difference of the current event registration system and another one in place.
424
425        self ^=amp; other
426        return self
427
428    def __lshift__(self, count):
429        """
430
431            Shifts the current event registration system left by a given count.
432
433        merged_system = EventRegistration(self.event_name)
434        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
435        return merged_system
436
437    def __ilshift__(self, count):
438        """
439
440            Shifts the current event registration system left by a given count in place.
441
442        self <<= count
443        return self
444
445    def __rshift__(self, count):
446        """
447
448            Shifts the current event registration system right by a given count.
449
450        merged_system = EventRegistration(self.event_name)
451        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
452        return merged_system
453
454    def __irshift__(self, count):
455        """
456
457            Shifts the current event registration system right by a given count in place.
458
459        self >>= count
460        return self
461
462    def __and__(self, other):
463        """
464
465            Returns the intersection of the current event registration system and another one.
466
467        merged_system = EventRegistration(self.event_name)
468        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
469        return merged_system
470
471    def __iand__(self, other):
472        """
473
474            Returns the intersection of the current event registration system and another one in place.
475
476        self &=amp; other
477        return self
478
479    def __or__(self, other):
480        """
481
482            Returns the union of the current event registration system and another one.
483
484        merged_system = EventRegistration(self.event_name)
485        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
486        return merged_system
487
488    def __ior__(self, other):
489        """
490
491            Returns the union of the current event registration system and another one in place.
492
493        self |=amp; other
494        return self
495
496    def __xor__(self, other):
497        """
498
499            Returns the symmetric difference of the current event registration system and another one.
500
501        merged_system = EventRegistration(self.event_name)
502        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
503        return merged_system
504
505    def __ixor__(self, other):
506        """
507
508            Returns the symmetric difference of the current event registration system and another one in place.
509
510        self ^=amp; other
511        return self
512
513    def __lshift__(self, count):
514        """
515
516            Shifts the current event registration system left by a given count.
517
518        merged_system = EventRegistration(self.event_name)
519        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
520        return merged_system
521
522    def __ilshift__(self, count):
523        """
524
525            Shifts the current event registration system left by a given count in place.
526
527        self <<= count
528        return self
529
530    def __rshift__(self, count):
531        """
532
533            Shifts the current event registration system right by a given count.
534
535        merged_system = EventRegistration(self.event_name)
536        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
537        return merged_system
538
539    def __irshift__(self, count):
540        """
541
542            Shifts the current event registration system right by a given count in place.
543
544        self >>= count
545        return self
546
547    def __and__(self, other):
548        """
549
550            Returns the intersection of the current event registration system and another one.
551
552        merged_system = EventRegistration(self.event_name)
553        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
554        return merged_system
555
556    def __iand__(self, other):
557        """
558
559            Returns the intersection of the current event registration system and another one in place.
560
561        self &=amp; other
562        return self
563
564    def __or__(self, other):
565        """
566
567            Returns the union of the current event registration system and another one.
568
569        merged_system = EventRegistration(self.event_name)
570        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
571        return merged_system
572
573    def __ior__(self, other):
574        """
575
576            Returns the union of the current event registration system and another one in place.
577
578        self |=amp; other
579        return self
580
581    def __xor__(self, other):
582        """
583
584            Returns the symmetric difference of the current event registration system and another one.
585
586        merged_system = EventRegistration(self.event_name)
587        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
588        return merged_system
589
590    def __ixor__(self, other):
591        """
592
593            Returns the symmetric difference of the current event registration system and another one in place.
594
595        self ^=amp; other
596        return self
597
598    def __lshift__(self, count):
599        """
600
601            Shifts the current event registration system left by a given count.
602
603        merged_system = EventRegistration(self.event_name)
604        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
605        return merged_system
606
607    def __ilshift__(self, count):
608        """
609
610            Shifts the current event registration system left by a given count in place.
611
612        self <<= count
613        return self
614
615    def __rshift__(self, count):
616        """
617
618            Shifts the current event registration system right by a given count.
619
620        merged_system = EventRegistration(self.event_name)
621        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
622        return merged_system
623
624    def __irshift__(self, count):
625        """
626
627            Shifts the current event registration system right by a given count in place.
628
629        self >>= count
630        return self
631
632    def __and__(self, other):
633        """
634
635            Returns the intersection of the current event registration system and another one.
636
637        merged_system = EventRegistration(self.event_name)
638        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
639        return merged_system
640
641    def __iand__(self, other):
642        """
643
644            Returns the intersection of the current event registration system and another one in place.
645
646        self &=amp; other
647        return self
648
649    def __or__(self, other):
650        """
651
652            Returns the union of the current event registration system and another one.
653
654        merged_system = EventRegistration(self.event_name)
655        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
656        return merged_system
657
658    def __ior__(self, other):
659        """
660
661            Returns the union of the current event registration system and another one in place.
662
663        self |=amp; other
664        return self
665
666    def __xor__(self, other):
667        """
668
669            Returns the symmetric difference of the current event registration system and another one.
670
671        merged_system = EventRegistration(self.event_name)
672        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
673        return merged_system
674
675    def __ixor__(self, other):
676        """
677
678            Returns the symmetric difference of the current event registration system and another one in place.
679
680        self ^=amp; other
681        return self
682
683    def __lshift__(self, count):
684        """
685
686            Shifts the current event registration system left by a given count.
687
688        merged_system = EventRegistration(self.event_name)
689        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
690        return merged_system
691
692    def __ilshift__(self, count):
693        """
694
695            Shifts the current event registration system left by a given count in place.
696
697        self <<= count
698        return self
699
700    def __rshift__(self, count):
701        """
702
703            Shifts the current event registration system right by a given count.
704
705        merged_system = EventRegistration(self.event_name)
706        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
707        return merged_system
708
709    def __irshift__(self, count):
710        """
711
712            Shifts the current event registration system right by a given count in place.
713
714        self >>= count
715        return self
716
717    def __and__(self, other):
718        """
719
720            Returns the intersection of the current event registration system and another one.
721
722        merged_system = EventRegistration(self.event_name)
723        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
724        return merged_system
725
726    def __iand__(self, other):
727        """
728
729            Returns the intersection of the current event registration system and another one in place.
730
731        self &=amp; other
732        return self
733
734    def __or__(self, other):
735        """
736
737            Returns the union of the current event registration system and another one.
738
739        merged_system = EventRegistration(self.event_name)
740        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
741        return merged_system
742
743    def __ior__(self, other):
744        """
745
746            Returns the union of the current event registration system and another one in place.
747
748        self |=amp; other
749        return self
750
751    def __xor__(self, other):
752        """
753
754            Returns the symmetric difference of the current event registration system and another one.
755
756        merged_system = EventRegistration(self.event_name)
757        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
758        return merged_system
759
760    def __ixor__(self, other):
761        """
762
763            Returns the symmetric difference of the current event registration system and another one in place.
764
765        self ^=amp; other
766        return self
767
768    def __lshift__(self, count):
769        """
770
771            Shifts the current event registration system left by a given count.
772
773        merged_system = EventRegistration(self.event_name)
774        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
775        return merged_system
776
777    def __ilshift__(self, count):
778        """
779
780            Shifts the current event registration system left by a given count in place.
781
782        self <<= count
783        return self
784
785    def __rshift__(self, count):
786        """
787
788            Shifts the current event registration system right by a given count.
789
790        merged_system = EventRegistration(self.event_name)
791        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
792        return merged_system
793
794    def __irshift__(self, count):
795        """
796
797            Shifts the current event registration system right by a given count in place.
798
799        self >>= count
800        return self
801
802    def __and__(self, other):
803        """
804
805            Returns the intersection of the current event registration system and another one.
806
807        merged_system = EventRegistration(self.event_name)
808        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
809        return merged_system
810
811    def __iand__(self, other):
812        """
813
814            Returns the intersection of the current event registration system and another one in place.
815
816        self &=amp; other
817        return self
818
819    def __or__(self, other):
820        """
821
822            Returns the union of the current event registration system and another one.
823
824        merged_system = EventRegistration(self.event_name)
825        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
826        return merged_system
827
828    def __ior__(self, other):
829        """
830
831            Returns the union of the current event registration system and another one in place.
832
833        self |=amp; other
834        return self
835
836    def __xor__(self, other):
837        """
838
839            Returns the symmetric difference of the current event registration system and another one.
840
841        merged_system = EventRegistration(self.event_name)
842        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
843        return merged_system
844
845    def __ixor__(self, other):
846        """
847
848            Returns the symmetric difference of the current event registration system and another one in place.
849
850        self ^=amp; other
851        return self
852
853    def __lshift__(self, count):
854        """
855
856            Shifts the current event registration system left by a given count.
857
858        merged_system = EventRegistration(self.event_name)
859        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
860        return merged_system
861
862    def __ilshift__(self, count):
863        """
864
865            Shifts the current event registration system left by a given count in place.
866
867        self <<= count
868        return self
869
870    def __rshift__(self, count):
871        """
872
873            Shifts the current event registration system right by a given count.
874
875        merged_system = EventRegistration(self.event_name)
876        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
877        return merged_system
878
879    def __irshift__(self, count):
880        """
881
882            Shifts the current event registration system right by a given count in place.
883
884        self >>= count
885        return self
886
887    def __and__(self, other):
888        """
889
890            Returns the intersection of the current event registration system and another one.
891
892        merged_system = EventRegistration(self.event_name)
893        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
894        return merged_system
895
896    def __iand__(self, other):
897        """
898
899            Returns the intersection of the current event registration system and another one in place.
900
901        self &=amp; other
902        return self
903
904    def __or__(self, other):
905        """
906
907            Returns the union of the current event registration system and another one.
908
909        merged_system = EventRegistration(self.event_name)
910        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
911        return merged_system
912
913    def __ior__(self, other):
914        """
915
916            Returns the union of the current event registration system and another one in place.
917
918        self |=amp; other
919        return self
920
921    def __xor__(self, other):
922        """
923
924            Returns the symmetric difference of the current event registration system and another one.
925
926        merged_system = EventRegistration(self.event_name)
927        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
928        return merged_system
929
930    def __ixor__(self, other):
931        """
932
933            Returns the symmetric difference of the current event registration system and another one in place.
934
935        self ^=amp; other
936        return self
937
938    def __lshift__(self, count):
939        """
940
941            Shifts the current event registration system left by a given count.
942
943        merged_system = EventRegistration(self.event_name)
944        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
945        return merged_system
946
947    def __ilshift__(self, count):
948        """
949
950            Shifts the current event registration system left by a given count in place.
951
952        self <<= count
953        return self
954
955    def __rshift__(self, count):
956        """
957
958            Shifts the current event registration system right by a given count.
959
960        merged_system = EventRegistration(self.event_name)
961        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
962        return merged_system
963
964    def __irshift__(self, count):
965        """
966
967            Shifts the current event registration system right by a given count in place.
968
969        self >>= count
970        return self
971
972    def __and__(self, other):
973        """
974
975            Returns the intersection of the current event registration system and another one.
976
977        merged_system = EventRegistration(self.event_name)
978        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
979        return merged_system
980
981    def __iand__(self, other):
982        """
983
984            Returns the intersection of the current event registration system and another one in place.
985
986        self &=amp; other
987        return self
988
989    def __or__(self, other):
990        """
991
992            Returns the union of the current event registration system and another one.
993
994        merged_system = EventRegistration(self.event_name)
995        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
996        return merged_system
997
998    def __ior__(self, other):
999        """
1000
1001            Returns the union of the current event registration system and another one in place.
1002
1003        self |=amp; other
1004        return self
1005
1006    def __xor__(self, other):
1007        """
1008
1009            Returns the symmetric difference of the current event registration system and another one.
1010
1011        merged_system = EventRegistration(self.event_name)
1012        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
1013        return merged_system
1014
1015    def __ixor__(self, other):
1016        """
1017
1018            Returns the symmetric difference of the current event registration system and another one in place.
1019
1020        self ^=amp; other
1021        return self
1022
1023    def __lshift__(self, count):
1024        """
1025
1026            Shifts the current event registration system left by a given count.
1027
1028        merged_system = EventRegistration(self.event_name)
1029        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
1030        return merged_system
1031
1032    def __ilshift__(self, count):
1033        """
1034
1035            Shifts the current event registration system left by a given count in place.
1036
1037        self <<= count
1038        return self
1039
1040    def __rshift__(self, count):
1041        """
1042
1043            Shifts the current event registration system right by a given count.
1044
1045        merged_system = EventRegistration(self.event_name)
1046        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
1047        return merged_system
1048
1049    def __irshift__(self, count):
1050        """
1051
1052            Shifts the current event registration system right by a given count in place.
1053
1054        self >>= count
1055        return self
1056
1057    def __and__(self, other):
1058        """
1059
1060            Returns the intersection of the current event registration system and another one.
1061
1062        merged_system = EventRegistration(self.event_name)
1063        merged_system.participants = {student_id: name for (student_id, name) in self.participants & other.participants}
1064        return merged_system
1065
1066    def __iand__(self, other):
1067        """
1068
1069            Returns the intersection of the current event registration system and another one in place.
1070
1071        self &=amp; other
1072        return self
1073
1074    def __or__(self, other):
1075        """
1076
1077            Returns the union of the current event registration system and another one.
1078
1079        merged_system = EventRegistration(self.event_name)
1080        merged_system.participants = {student_id: name for (student_id, name) in self.participants | other.participants}
1081        return merged_system
1082
1083    def __ior__(self, other):
1084        """
1085
1086            Returns the union of the current event registration system and another one in place.
1087
1088        self |=amp; other
1089        return self
1090
1091    def __xor__(self, other):
1092        """
1093
1094            Returns the symmetric difference of the current event registration system and another one.
1095
1096        merged_system = EventRegistration(self.event_name)
1097        merged_system.participants = {student_id: name for (student_id, name) in self.participants ^ other.participants}
1098        return merged_system
1099
1100    def __ixor__(self, other):
1101        """
1102
1103            Returns the symmetric difference of the current event registration system and another one in place.
1104
1105        self ^=amp; other
1106        return self
1107
1108    def __lshift__(self, count):
1109        """
1110
1111            Shifts the current event registration system left by a given count.
1112
1113        merged_system = EventRegistration(self.event_name)
1114        merged_system.participants = {student_id: name for (student_id, name) in self.participants << count}
1115        return merged_system
1116
1117    def __ilshift__(self, count):
1118        """
1119
1120            Shifts the current event registration system left by a given count in place.
1121
1122        self <<= count
1123        return self
1124
1125    def __rshift__(self, count):
1126        """
1127
1128            Shifts the current event registration system right by a given count.
1129
1130        merged_system = EventRegistration(self.event_name)
1131        merged_system.participants = {student_id: name for (student_id, name) in self.participants >> count}
1132        return merged_system
1133
1134    def __irshift__(self, count):
1135        """
1
```

```
45     def remove_student(self, student_id: int) -> None:
46         if student_id in self.participants:
47             student_name = self.participants.pop(student_id)
48             print(f"--> Success: Removed {student_name} (ID: {student_id}) from the event.")
49         else:
50             print(f"--> Warning: Student with ID {student_id} not found in registration list.")
51
52     def find_student(self, student_id: int) -> Optional[str]:
53         """
54             Finds a student's name by their ID.
55
56             This operation has an average time complexity of O(1).
57
58             Args:
59                 student_id (int): The ID of the student to find.
60
61             Returns:
62                 Optional[str]: The student's name if found, otherwise None.
63             """
64
65         return self.participants.get(student_id)
66
67     def display_participants(self) -> None:
68         """Prints a list of all registered participants."""
69         print(f"\n--- Participants for '{self.event_name}' ({self.get_participant_count()}) ---")
70         if not self.participants:
71             print("No students are currently registered.")
72         else:
73             for student_id, student_name in self.participants.items():
74                 print(f"ID: {student_id} Name: {student_name}")
75
76
77
78
79
80
81
82
83
84
85
86
87     def get_participant_count(self) -> int:
88         """Returns the total number of registered participants."""
89         return len(self.participants)
90
91 # --- Example Usage ---
92 if __name__ == "__main__":
93     # 1. Create a registration system for a "CodeFest 2024" event
94     codefest_regs = EventRegistration("CodeFest 2024")
95
96     # 2. Register several students
97     print("\n--- Registering Students ---")
98     codefest_regs.register_student(101, "Alice")
99     codefest_regs.register_student(205, "Bob")
100    codefest_regs.register_student(315, "Charlie")
101    codefest_regs.register_student(101, "Alice") # Attempt to register a duplicate
102
103    # 3. Display the current list of participants
104    codefest_regs.display_participants()
105
106    # 4. Find a specific student
107    print("\n--- Searching for a Student ---")
108    student_id_to_find = 205
109    found_name = codefest_regs.find_student(student_id_to_find)
110    if found_name:
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.")
```

```
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
```

```
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:

Strengths:

1. **Excellent Use of Data Structure:** The choice of a dictionary (`self.participants`) is ideal. It correctly leverages the O(1) average time complexity for insertions, deletions, and lookups, which is exactly what's needed for an efficient registration system.

2. **Clarity and Readability:** The code is exceptionally clean and well-documented. Method names like `register_student`, `remove_student`, and `find_student` are intuitive, and the docstrings clearly explain the purpose, arguments, and time complexity of each method.

3. **Robustness:** The code includes good

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 is a perfect demonstration of the class's functionality. It walks through all the 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 currently print success or failure messages but



1. Boolean Return Values for State-Changing Methods:

The `register_student` and `remove_student` methods currently print success or failure messages but don't return any value (`None`). It's often more useful for these methods to return a boolean (`True` for success, `False` for failure). This makes the class more versatile, as calling code can programmatically check the outcome of an operation instead of relying on parsing printed output.

2. Batch Operations:

For larger events, it might be useful to register or remove multiple students at once. Adding `register_batch` and `remove_batch`

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