ASSIGNMENT-11.1

NAME: BHUVANESHWAR REDDY

HALLTICKET NUNMER: 2403A52416

BATCH:15

SUBJECT:AI CODING

TASK-1

QUESTION

#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:

Write a Python class called Stack with methods push, pop, peek, and is empty. Use a list to store elements. Add short docstrings for the class and methods.

CODE:

ASSIGNMENT 11.1 TASK1.PY

```python
from typing import Any, List, Optional


class Stack:
    """A simple Stack class that implements a LIFO (Last-In, First-Out) structure."""

    def __init__(self) -> None:
        """Initializes an empty stack."""
        self._items: List[Any] = []

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

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

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

        Returns:
            The top item of the stack.

        Raises:
            IndexError: if the stack is empty.
        """
```


ASSIGNMENT 11.1 TASK1.PY

```python
class Stack:
    def pop(self) -> Any:
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        return self._items.pop()

    def peek(self) -> Optional[Any]:
        """
        Returns the top item of the stack without removing it.

        Returns:
            The top item of the stack, or None if the stack is empty.
        """
        if self.is_empty():
            return None
        return self._items[-1]

    def is_empty(self) -> bool:
        """
        Checks if the stack is empty.

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


if __name__ == "__main__":
```

ASSIGNMENT 11.1 TASK1.PY

```
        return not self._items


if __name__ == "__main__":
    print("--- Stack Demonstration ---")
    stack = Stack()

    print(f"Is stack empty? {stack.is_empty()}")  # True

    print("Pushing 1, 2, 3 onto the stack...")
    stack.push(1)
    stack.push(2)
    stack.push(3)

    print(f"Top item (peek): {stack.peek()}")  # 3
    print(f"Popped item: {stack.pop()}")  # 3
    print(f"Top item after pop: {stack.peek()}")  # 2
    print(f"Is stack empty? {stack.is_empty()}")  # False

    print(f"Popped item: {stack.pop()}")  # 2
    print(f"Popped item: {stack.pop()}")  # 1

    print(f"Is stack empty now? {stack.is_empty()}")  # True

```

OUTPUT:

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAM
.PY"
--- Stack Demonstration ---
Is stack empty? True
Pushing 1, 2, 3 onto the stack...
Top item (peek): 3
Popped item: 3
Top item after pop: 2
Is stack empty? False
Popped item: 2
Popped item: 1
Is stack empty now? True
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAM
.PY"
```

OBSERVATION:

- **The Stack class correctly implements a LIFO structure using Python lists.**
- **It supports essential operations: push, pop, peek, and is_empty with proper error handling**
- **Docstrings make the code easy to read and understand**

- **The demo shows stack behavior clearly, proving the implementation works as expected.**

TASK-2

QUESTION:

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 method.

PROMPT:

Generate a python code to implement a Queue using Python lists. with methods fifo-based queue class with enqueue, dequeue, peek, and size.

CODE:

```python
from collections import deque
from typing import Any, Deque, Optional


class Queue:
    """
    A simple Queue class that implements a FIFO (First-In, First-Out) structure.

    This implementation uses collections.deque for efficient O(1) appends and pops
    from both ends, which is ideal for a queue.
    """

    def __init__(self) -> None:
        """Initializes an empty queue."""
        self.items: Deque[Any] = deque()

    def __len__(self) -> int:
        """Returns the number of items in the queue."""
        return len(self.items)

    def enqueue(self, item: Any) -> None:
        """
        Adds an item to the back (end) of the queue.

        Args:
            item: The item to be added to the queue.
        """
        self.items.append(item)
```

Ln 15, Col 1 Spaces: 4 UTF-8 CRLF {} Python Chat quo

ASSIGNMENT 11.1 TASK1.PY queue.py 1

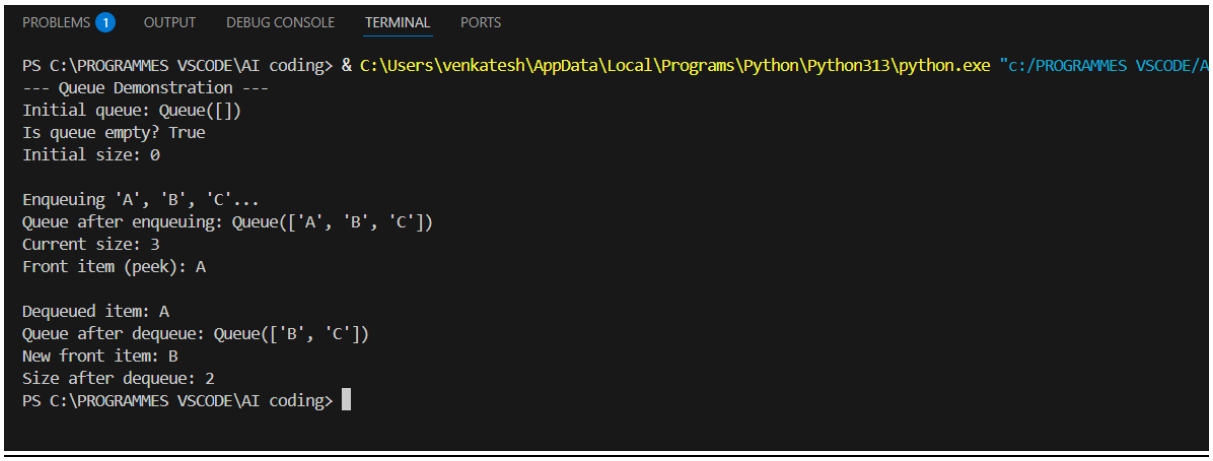
```
class Queue:
    def dequeue(self) -> Any:
        """
        Removes and returns the item from the front of the queue.

        Returns:
            The front item of the queue.

        Raises:
            IndexError: if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        return self.items.popleft()

    def peek(self) -> Optional[Any]:
        """
        Returns the front item of the queue without removing it.

        Returns:
            The front item of the queue, or None if the queue is empty.
        """
        if self.is_empty():
            return None
        return self.items[0]

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

0 ⚠ 1 Ln 15, Col 1 Spaces: 4 UTF-8 CRLF

```
class Queue:
    def is_empty(self) -> bool:
        Returns:
            True if the queue is empty, False otherwise.
        """
        return len(self.items) == 0

    def __str__(self) -> str:
        """Returns a string representation of the queue."""
        return f"Queue({list(self.items)})"


if __name__ == "__main__":
    print("--- Queue Demonstration ---")
    q = Queue()

    print(f"Initial queue: {q}")
    print(f"Is queue empty? {q.is_empty()}")  # True
    print(f"Initial size: {len(q)}")      # 0

    print("\nEnqueuing 'A', 'B', 'C'...")
    q.enqueue('A')
    q.enqueue('B')
    q.enqueue('C')

    print(f"Queue after enqueuing: {q}")
    print(f"Current size: {len(q)}")      # 3
    print(f"Front item (peek): {q.peek()}")  # 'A'

```

0 ⚠ 1 Ln 15, Col 1 Spaces: 4 UTF-8 CRLF {} Python

OUTPUT:



OBSERVATION:

This Python code defines a Queue class implementing a First-In, First-Out (FIFO) structure using the highly efficient collections deque for optimal performance. It provides standard methods like enqueue, dequeue, and peek for queue manipulation. The class is also made more intuitive and Pythonic by including special methods like _len_ and _str_

TASK-3

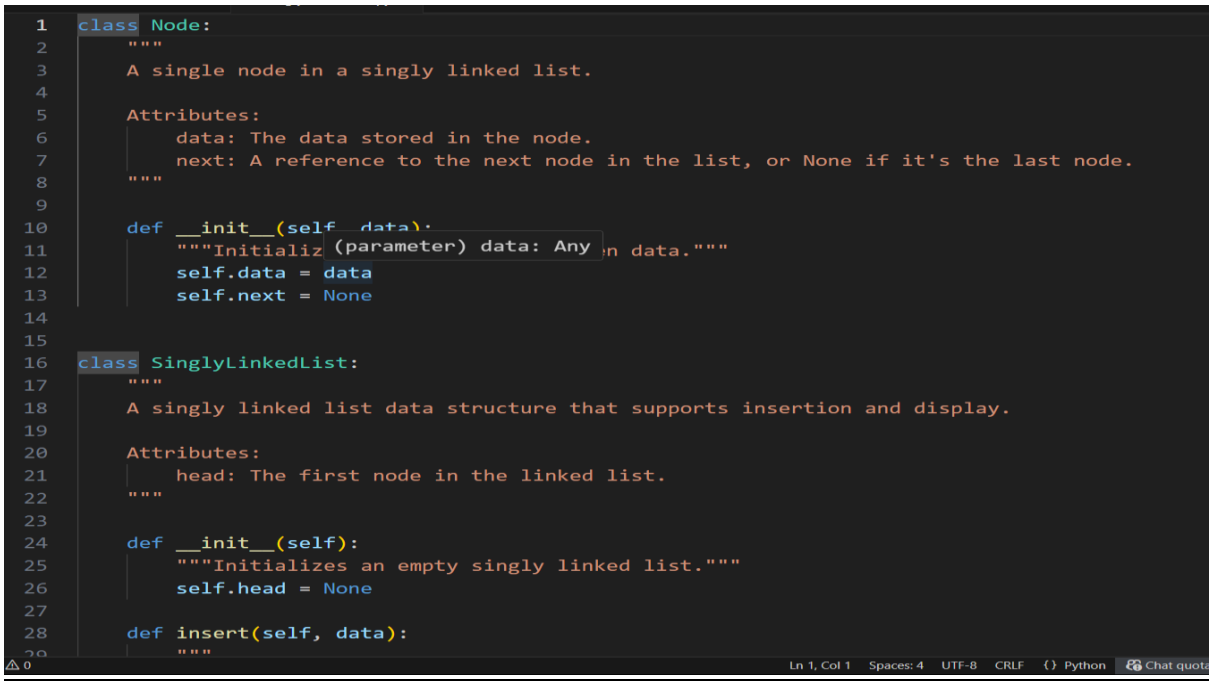
QUESTION:

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

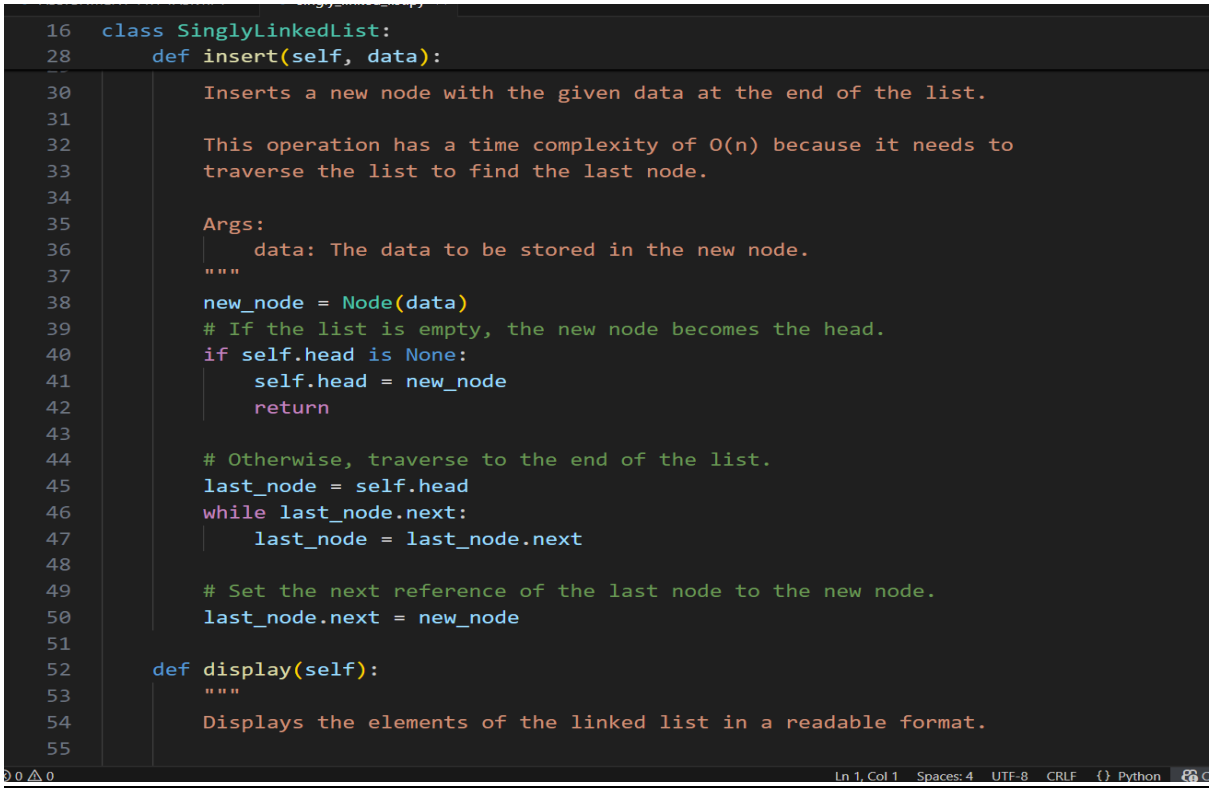
PROMPT:

Generate a python code Singly Linked List with insert and display methods.with clear method documentation

CODE:



```python
class Node:
    """
    A single node in a singly linked list.

    Attributes:
        data: The data stored in the node.
        next: A reference to the next node in the list, or None if it's the last node.
    """

    def __init__(self, data):
        """Initializ (parameter) data: Any n data."""
        self.data = data
        self.next = None


class SinglyLinkedList:
    """
    A singly linked list data structure that supports insertion and display.

    Attributes:
        head: The first node in the linked list.
    """

    def __init__(self):
        """Initializes an empty singly linked list."""
        self.head = None

    def insert(self, data):
        """
```



```python
class SinglyLinkedList:
    def insert(self, data):
        """
        Inserts a new node with the given data at the end of the list.

        This operation has a time complexity of O(n) because it needs to
        traverse the list to find the last node.

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

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

        # Set the next reference of the last node to the new node.
        last_node.next = new_node

    def display(self):
        """
        Displays the elements of the linked list in a readable format.
```

```python
class SinglyLinkedList:
    def display(self):
        elements = []
        current_node = self.head
        while current_node:
            elements.append(str(current_node.data))
            current_node = current_node.next
        print(" -> ".join(elements) + " -> None")


if __name__ == "__main__":
    print("--- Singly Linked List Demonstration ---")

    # Create a new linked list
    sll = SinglyLinkedList()

    print("Initial list:")
    sll.display()  # Expected: -> None

    # Insert some elements
    print("\nInserting 10, 20, and 30...")
    sll.insert(10)
    sll.insert(20)
    sll.insert(30)

    # Display the list again
    print("List after insertions:")
    sll.display()  # Expected: 10 -> 20 -> 30 -> None
```


OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI
--- Singly Linked List Demonstration ---
Initial list:
 -> None

Inserting 10, 20, and 30...
List after insertions:
10 -> 20 -> 30 -> None
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI
--- Singly Linked List Demonstration ---
Initial list:
 -> None

Inserting 10, 20, and 30...
List after insertions:
10 -> 20 -> 30 -> None
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

This Python code provides a clear and fundamental implementation of a Singly LinkedList with its corresponding Node class. It correctly implements an insert method that adds new nodes to the end of the list (an O(n) operation) and a display method for easy visualization. The code is well-documented with docstrings and includes a simple demonstration block, making it an excellent example for learning this data structure.

TASK-4

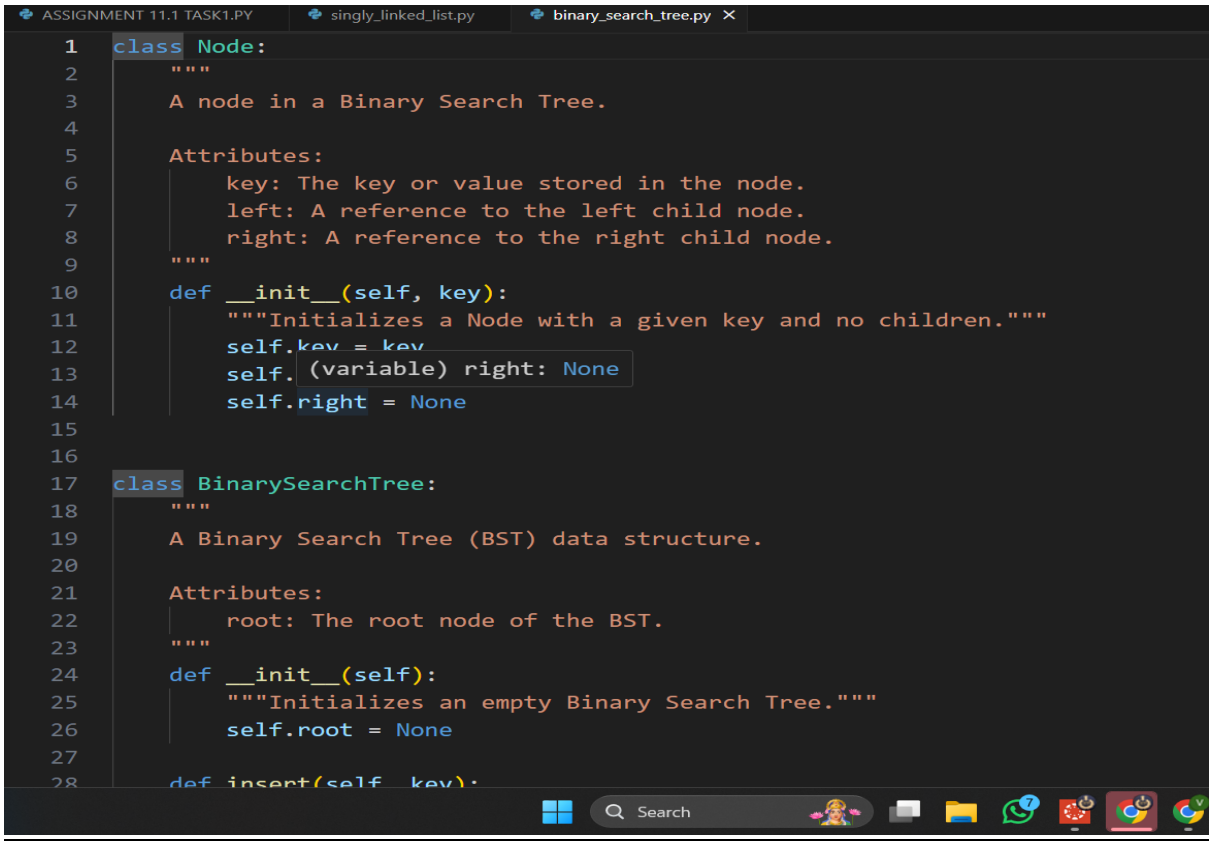
QUESTION:

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 python code BST with insert and in-order traversal methods

CODE:



```
17  class BinarySearchTree:
27
28      def insert(self, key):
29          """
30          Public method to insert a new key into the BST.
31
32          If the tree is empty, the new key becomes the root. Otherwise, it calls
33          a private recursive helper to find the correct position.
34
35          Args:
36              key: The key to insert into the tree.
37          """
38          if self.root is None:
39              self.root = Node(key)
40          else:
41              self._insert_recursive(self.root, key)
42
43      def _insert_recursive(self, current_node, key):
44          """
45          Recursively finds the correct position and inserts a new node.
46
47          Args:
48              current_node: The node to start the search from.
49              key: The key to insert.
50          """
51          if key < current_node.key:
52              if current_node.left is None:
```

```
17  class BinarySearchTree:
43      def _insert_recursive(self, current_node, key):
54              else:
55                  self._insert_recursive(current_node.left, key)
56          elif key > current_node.key:
57              if current_node.right is None:
58                  current_node.right = Node(key)
59              else:
60                  self._insert_recursive(current_node.right, key)
61          # If key is equal to current_node.key, do nothing (no duplicates).
62
63      def in_order_traversal(self):
64          """
65          Public method to perform and display an in-order traversal.
66
67          This traversal visits nodes in the order: left, root, right.
68          For a BST, this results in printing the keys in ascending sorted order.
69          """
70          print("In-order traversal:", end=" ")
71          self._in_order_recursive(self.root)
72          print() # For a newline at the end.
73
74      def _in_order_recursive(self, current_node):
75          """
76          Recursively performs an in-order traversal.
77
78          Args:
```

```python
class BinarySearchTree:
    def _in_order_recursive(self, current_node):
        Args:
            current_node: The node to start the traversal from.
        """
        if current_node:
            self._in_order_recursive(current_node.left)
            print(current_node.key, end=" ")
            self._in_order_recursive(current_node.right)


if __name__ == "__main__":
    print("--- Binary Search Tree Demonstration ---")

    # Create a new BST
    bst = BinarySearchTree()

    # Insert elements into the BST
    # The order of insertion matters for the tree's structure.
    keys_to_insert = [50, 30, 70, 20, 40, 60, 80]
    print(f"Inserting keys: {keys_to_insert}")
    for key in keys_to_insert:
        bst.insert(key)

    # Display the in-order traversal
    # This should print the keys in sorted order.
    bst.in_order_traversal()
    # Expected output: In-order traversal: 20 30 40 50 60 70 80
```

OUTPUT:

```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/
--- Binary Search Tree Demonstration ---
Inserting keys: [50, 30, 70, 20, 40, 60, 80]
In-order traversal: 20 30 40 50 60 70 80
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSESRVATION:

This code provides a clean, recursive implementation of a Binary Search Tree with insert and in order traversal methods. The structure is well-defined with separate Node and Binary Search Tree classes, and the traversal correctly yields the elements in sorted order.

TASK-5

QUESTION:

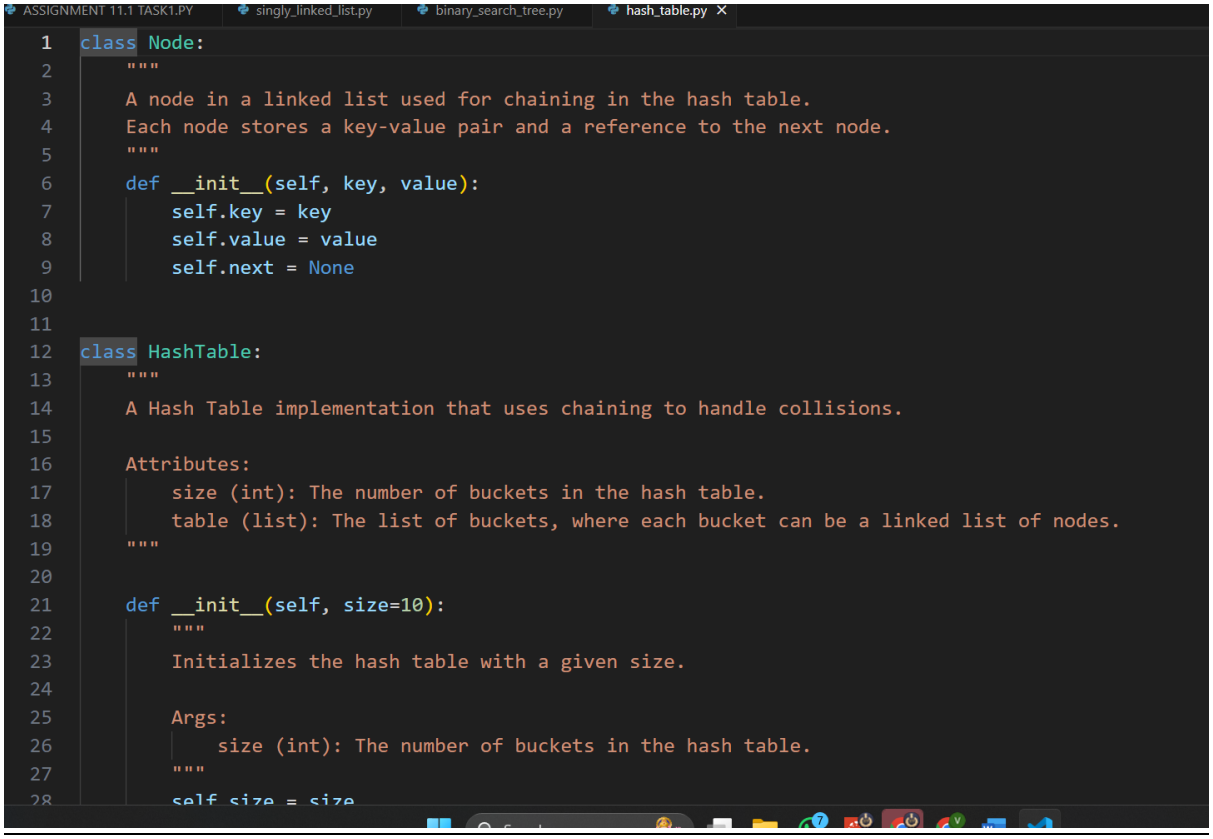
5– Hash Table
Task: Use AI to implement a hash table with basic insert, search, and delete

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

PROMPT:

Impliment a python code ash table with basic insert, search, and delete methods and Collision handling using chaining, with well-commented methods

CODE:



ASSIGNMENT 11.1 TASK1.PY | singly_linked_list.py | binary_search_tree.py | hash_table.py

```python
class HashTable:
    def __init__(self, size=10):
        self.table = [None] * self.size

    def _hash(self, key):
        """
        Private helper method to compute the hash index for a given key.

        Args:
            key: The key to be hashed.

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

    def insert(self, key, value):
        """
        Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated.

        Args:
            key: The key to insert.
            value: The value associated with the key.
        """
        index = self._hash(key)

```

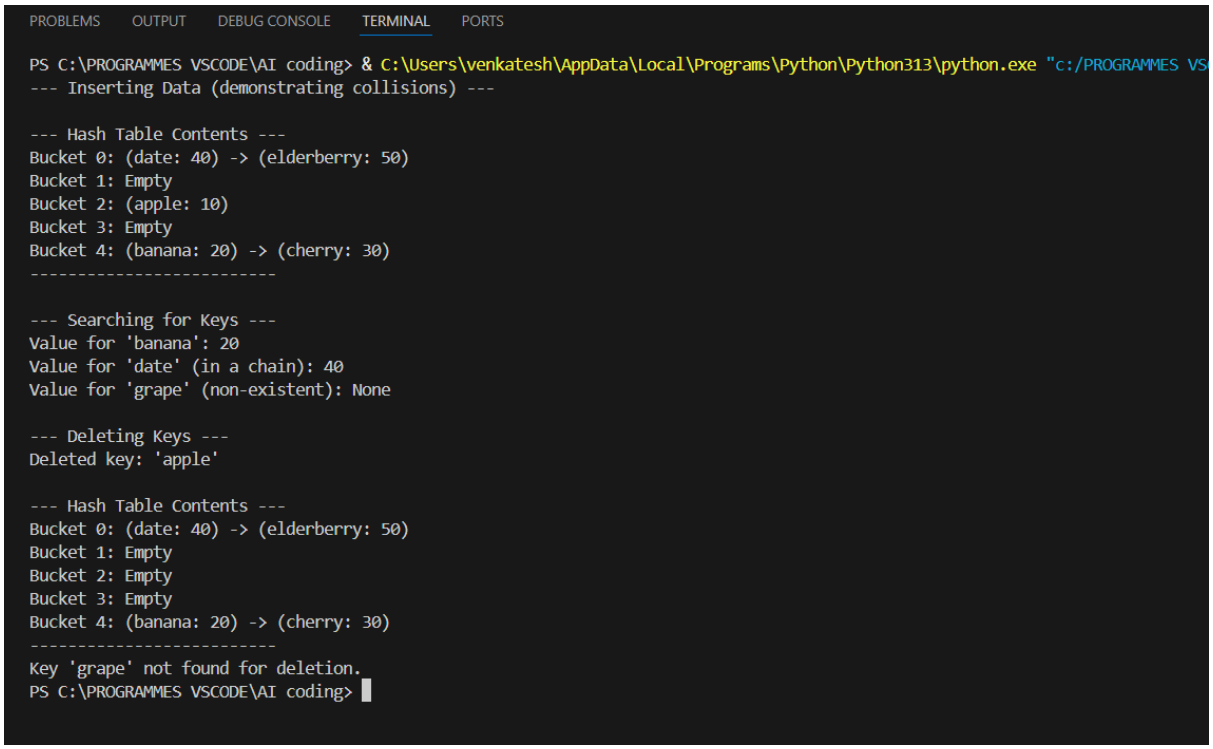
```python
class HashTable:
    def insert(self, key, value):
        index = self._hash(key)

        # If the bucket is empty, create a new node and place it there.
        if self.table[index] is None:
            self.table[index] = Node(key, value)
            return

        # If the bucket is not empty, traverse the linked list (chain).
        current = self.table[index]
        while current:
            # If key already exists, update the value and return.
            if current.key == key:
                current.value = value
                return
            # If we are at the end of the list, break to append the new node.
            if current.next is None:
                break
            current = current.next

        # Append the new node to the end of the chain.
        current.next = Node(key, value)

    def search(self, key):
        """
        Searches for a key in the hash table and returns its value.

```

```
class HashTable:
    def search(self, key):
            key: The key to search for.

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

        # Traverse the chain at the calculated index.
        while current:
            if current.key == key:
                return current.value
            current = current.next

        # If the key was not found in the chain.
        return None

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

        Args:
            key: The key to be deleted.
        """
        index = self._hash(key)
        current = self.table[index]
```

ASSIGNMENT 11.1 TASK1.PY | singly_linked_list.py | binary_search_tree.py | hash_table.py

```
class HashTable:
    def display(self):
                print(" -> ".join(nodes))
        print("--------------------------")


if __name__ == "__main__":
    # Initialize a hash table with a small size to demonstrate collisions.
    ht = HashTable(size=5)

    print("--- Inserting Data (demonstrating collisions) ---")
    # These keys are chosen because they often collide with a small table size.
    ht.insert("apple", 10)
    ht.insert("banana", 20)
    ht.insert("cherry", 30)
    ht.insert("date", 40)      # Collides with 'apple' if hash('apple') % 5 == hash('date') % 5
    ht.insert("elderberry", 50) # Also likely to collide
    ht.display()

    print("\n--- Searching for Keys ---")
    print(f"Value for 'banana': {ht.search('banana')}")
    print(f"Value for 'date' (in a chain): {ht.search('date')}")
    print(f"Value for 'grape' (non-existent): {ht.search('grape')}")

    print("\n--- Deleting Keys ---")
    ht.delete("apple")  # Delete the head of a chain.
    ht.display()
```

OUTPUT:



OBSERVATION:

This code provides a robust implementation of a Hash Table using the chaining method with a linked list to handle collisions. It correctly implements the essential insert, search, and delete operations, including logic for updating existing keys and handling deletions within the chains. The code is well-commented and includes a display method and a demonstration block, making it a clear and effective educational example.

TASK-6

QUESTION:

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:

Write a Python class Graph using an adjacency list. It should have methods to add a vertex, add an edge (undirected), and display the graph. Use a dictionary to store the adjacency list and add a small example.

CODE:

```python
from typing import Dict, List, Any

class Graph:
    """
    A class to represent an undirected graph using an adjacency list.

    Attributes:
        adjacency_list (Dict[Any, List[Any]]): A dictionary to store the
            adjacency list, where keys are vertices and values are lists
            of their neighbors.
    """

    def __init__(self) -> None:
        """Initialize an empty graph."""
        self.adjacency_list: Dict[Any, List[Any]] = {}

    def add_vertex(self, vertex: Any) -> None:
        """Add a vertex to the graph."""
        if vertex not in self.adjacency_list:
            self.adjacency_list[vertex] = []

    def add_edge(self, vertex1: Any, vertex2: Any) -> None:
        """
        Add an undirected edge between two vertices.
        If vertices are not present, they are added automatically.
        """
        # Ensure both vertices exist in the graph
```

```python
class Graph:
    def add_edge(self, vertex1: Any, vertex2: Any) -> None:
        self.add_vertex(vertex1)
        self.add_vertex(vertex2)

        # Add the edge from vertex1 to vertex2 if it doesn't already exist
        if vertex2 not in self.adjacency_list[vertex1]:
            self.adjacency_list[vertex1].append(vertex2)

        # Add the edge from vertex2 to vertex1 if it doesn't already exist
        if vertex1 not in self.adjacency_list[vertex2]:
            self.adjacency_list[vertex2].append(vertex1)

    def display(self) -> None:
        """Display the adjacency list of the graph."""
        print("\n--- Graph Adjacency List ---")
        for vertex, neighbors in self.adjacency_list.items():
            print(f"{vertex} -> {neighbors}")


# Demonstration
if __name__ == "__main__":
    g = Graph()
    g.add_edge("A", "B")
    g.add_edge("A", "C")
    g.add_edge("B", "D")
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/A

--- Graph Adjacency List ---
A -> ['B', 'C']
B -> ['A', 'D']
C -> ['A']
D -> ['B']
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

This Python code provides a clear and effective implementation of an undirected graph using a dictionary-based adjacency list, which is a standard representation

TASK-7

QUESTION:

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:

create a python code by implement a priority queue using Python's heapq module.with nqueue (priority), dequeue (highest priority), and display methods

CODE:

```python
import heapq
import itertools
from typing import List, Tuple, Any

class PriorityQueue:
    """
    A Priority Queue implementation using Python's heapq module (min-heap).

    Items with lower priority numbers are considered higher priority. This implementation
    is stable, meaning items with the same priority are dequeued in the order
    they were enqueued.
    """

    def __init__(self) -> None:
        """Initializes an empty priority queue."""
        self._pq: List[Tuple[int, int, Any]] = []
        self._counter = itertools.count()  # Unique counter for stable sorting

    def enqueue(self, item: Any, priority: int = 0) -> None:
        """
        Adds an item to the queue with a given priority.

        Args:
            item: The item to be added to the queue.
            priority (int): The priority of the item. Lower numbers have higher priority.
        """
        count = next(self._counter)
```

```python
class PriorityQueue:
    def enqueue(self, item: Any, priority: int = 0) -> None:
        Args:
            item: The item to be added to the queue.
            priority (int): The priority of the item. Lower numbers have higher priority.
        """
        count = next(self._counter)
        # The entry is a tuple: (priority, count, item).
        # The count acts as a tie-breaker to ensure FIFO for items with the same priority.
        entry = (priority, count, item)
        heapq.heappush(self._pq, entry)

    def dequeue(self) -> Any:
        """
        Removes and returns the item with the highest priority (lowest priority number).

        Returns:
            The item with the highest priority.

        Raises:
            IndexError: if the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from an empty priority queue")
        # heappop returns the smallest item, which is the one with the highest priority.
        priority, count, item = heapq.heappop(self._pq)
        return item

```

```python
class PriorityQueue:
        return item

    def is_empty(self) -> bool:
        """
        Checks if the priority queue is empty.

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

    def display(self) -> None:
        """
        Displays the items in the queue in priority order without modifying the queue.
        """
        print("--- Priority Queue Contents (Priority, Item) ---")
        if self.is_empty():
            print("Queue is empty.")
            return
        # Create a sorted copy to display in order, as the heap itself is not fully sorted.
        sorted_pq = sorted(self._pq)
        for priority, count, item in sorted_pq:
            print(f"({priority}, '{item}')")
        print("-----------------------------------------------")

```

```
 5  class PriorityQueue:
70          print("-----------------------------------------------")
71
72
73  if __name__ == "__main__":
74      pq = PriorityQueue()
75
76      print("Enqueuing items with different priorities...")
77      pq.enqueue("Task A - Low Priority", priority=10)
78      pq.enqueue("Task B - High Priority", priority=1)
79      pq.enqueue("Task C - Medium Priority", priority=5)
80      pq.enqueue("Task D - High Priority", priority=1) # Same priority as B
81      pq.display()
82
83      print("\nDequeuing items...")
84      print(f"Dequeued: {pq.dequeue()}")
85      print(f"Dequeued: {pq.dequeue()}")
86      print(f"Dequeued: {pq.dequeue()}")
87      print(f"Dequeued: {pq.dequeue()}")
88      pq.display()
```

OUTPUT:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAM
Enqueuing items with different priorities...
--- Priority Queue Contents (Priority, Item) ---
(1, 'Task B - High Priority')
(1, 'Task D - High Priority')
(5, 'Task C - Medium Priority')
(10, 'Task A - Low Priority')
-----------------------------------------------

Dequeuing items...
Dequeued: Task B - High Priority
Dequeued: Task D - High Priority
Dequeued: Task C - Medium Priority
Dequeued: Task A - Low Priority
--- Priority Queue Contents (Priority, Item) ---
Queue is empty.
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

This is a high-quality Priority Queue implementation that correctly uses Python's heapq module for efficient O(log n) operations. It cleverly employs itertools.count() as a tie-breaker to ensure stable, FIFO behavior for items sharing the same priority level. The code is clean, well-documented, and includes a clear demonstration of its functionality.

TASK-8

QUESTION:

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:

generate a python code by mplement a double-ended queue using collections.deque with Insert and remove from both ends with docstrings.

CODE:

```python
from collections import deque
from typing import Any

class Deque:
    """
    A Double-Ended Queue (Deque) implementation using Python's collections.deque.

    This class provides a clear interface to add and remove items from both the
    front and the rear of the queue, leveraging the performance of the
    underlying deque object.
    """

    def __init__(self) -> None:
        """Initializes an empty deque."""
        self._items: deque[Any] = deque()

    def add_front(self, item: Any) -> None:
        """
        Adds an item to the front (left side) of the deque.

        Args:
            item: The item to be added.
        """
        self._items.appendleft(item)

    def add_rear(self, item: Any) -> None:
        """
        Adds an item to the rear (right side) of the deque
```

```
 4  class Deque:
26      def add_rear(self, item: Any) -> None:
29
30          Args:
31              item: The item to be added.
32          """
33          self._items.append(item)
34
35      def remove_front(self) -> Any:
36          """
37          Removes and returns the item from the front of the deque.
38
39          Returns:
40              The item from the front of the deque.
41
42          Raises:
43              IndexError: if the deque is empty.
44          """
45          if self.is_empty():
46              raise IndexError("remove from an empty deque")
47          return self._items.popleft()
48
49      def remove_rear(self) -> Any:
50          """
51          Removes and returns the item from the rear of the deque.
52
```

```
 4  class Deque:
49      def remove_rear(self) -> Any:
57              IndexError: if the deque is empty.
58          """
59          if self.is_empty():
60              raise IndexError("remove from an empty deque")
61          return self._items.pop()
62
63      def is_empty(self) -> bool:
64          """Checks if the deque is empty."""
65          return not self._items
66
67      def __len__(self) -> int:
68          """Returns the number of items in the deque."""
69          return len(self._items)
70
71      def __str__(self) -> str:
72          """Returns a string representation of the deque."""
73          return f"Deque({list(self._items)})"
74
75
76  if __name__ == "__main__":
77      print("--- Deque Demonstration ---")
78      d = Deque()
79      print(f"Initial deque: {d}, Is empty? {d.is_empty()}")
80
81      print("\nAdding 'B' to rear, 'C' to rear, and 'A' to front...")
```

```python
    print("--- Deque Demonstration ---")
    d = Deque()
    print(f"Initial deque: {d}, Is empty? {d.is_empty()}")

    print("\nAdding 'B' to rear, 'C' to rear, and 'A' to front...")
    d.add_rear("B")
    d.add_rear("C")
    d.add_front("A")
    print(f"Deque state: {d}, Size: {len(d)}")

    print("\nRemoving items...")
    print(f"Removed from rear: '{d.remove_rear()}' -> State: {d}")
    print(f"Removed from front: '{d.remove_front()}' -> State: {d}")
    print(f"Final size: {len(d)}")
```

OUTPUT:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMME
--- Deque Demonstration ---
Initial deque: Deque([]), Is empty? True

Adding 'B' to rear, 'C' to rear, and 'A' to front...
Deque state: Deque(['A', 'B', 'C']), Size: 3

Removing items...
Removed from rear: 'C' -> State: Deque(['A', 'B'])
Removed from front: 'A' -> State: Deque(['B'])
Final size: 1
PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

This code provides an excellent wrapper around Python's collections.deque, creating a clear and explicit Deque class with a well-defined interface. It correctly leverages the underlying deque for highly efficient O(1) appends and pops from both ends.

TASK-9

QUESTION:

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 python code by comparison table of different data structures (stack, queue, linked list, etc.) including time complexities.with a markdown table with structure names, operations, and complexities

CODE:

```
def display_comparison_table():
    """
    Prints a markdown table comparing common data structures and their
    time complexities for various operations.
    """

    markdown_table = """
# Data Structure Time Complexity Comparison

Here is a summary of the average and worst-case time complexities for common operations on key data structures in Python.

| Data Structure                | Operation                     | Average Time Complexity | Worst-Case Time Complexity |
Notes                                                                                                                  |
|:------------------------------|:------------------------------|:------------------------|:---------------------------|
:-----------------------------------------------------------------------|
| **List (Dynamic Array)**      | Access by Index `[i]`         | `O(1)`                  | `O(1)`                     |
Direct memory access.                                                                                                  |
|                               | Search for Value `in`         | `O(n)`                  | `O(n)`                     |
Scans the entire list.                                                                                                 |
|                               | Append to End `append()`      | `O(1)` (Amortized)      | `O(n)`                     |
Amortized due to occasional resizing.                                                                                  |
|                               | Insert/Delete at Start/Middle | `O(n)`                  | `O(n)`                     |
Requires shifting subsequent elements.                                                                                 |
|                               | Pop from End `pop()`          | `O(1)`                  | `O(1)`
`                                                                                                                      |
|                               | Pop from Start `pop(0)`       | `O(n)`                  | `O(n)`                     |
Inefficient; use `collections.deque` for this.                                                                         |
| **Stack (LIFO via `list`)**   | Push `append()`               | `O(1)` (Amortized)      | `O(n)`                     |
```

```
def display_comparison_table():
    | **Queue (FIFO via `deque`)**    | Enqueue `append()`              | `O(1)`                | `O(1)`                |
`collections.deque` is implemented as a doubly-linked list.       |
    |                                 | Dequeue `popleft()`             | `O(1)`                | `O(1)`                |
Highly efficient.                                                  |
    |                                 | Peek `[0]`                      | `O(1)`                | `O(1)
`                     |                                                                   |
    | **Singly Linked List**          | Access / Search                 | `O(n)`                | `O(n)`                |
Requires traversal from the head.                                  |
    |                                 | Insert / Delete at Head         | `O(1)`                | `O(1)`                |
Just re-pointing the head.                                         |
    |                                 | Insert at End                   | `O(n)`                | `O(n)`                |
Requires traversal to the end (unless a tail pointer is kept).     |
    |                                 | Delete at End                   | `O(n)`                | `O(n)`                |
Requires traversal to find the second-to-last node.                |
    | **Hash Table (`dict`)**         | Access / Search / Insert / Delete | `O(1)` (Amortized)      | `O(n)
`                     | Worst case is extremely rare and occurs with many hash collisions. |
    | **Binary Search Tree (BST)**    | Access / Search / Insert / Delete | `O(log n)`              | `O(n)
`                     | `O(log n)` assumes a balanced tree. Worst case is an unbalanced tree.|

    """
    print(markdown_table)

if __name__ == "__main__":
    display_comparison_table()
```

OUTPUT:

```
PS C:\PROGRAMMES VSCODE\AI coding> & C:\Users\venkatesh\AppData\Local\Programs\Python\Python313\python.exe "c:/PROGRAMMES VSCODE/AI coding/data_structure_comparison.py"

# Data Structure Time Complexity Comparison

Here is a summary of the average and worst-case time complexities for common operations on key data structures in Python.

| Data Structure                | Operation                       | Average Time Complexity | Worst-Case Time Complexity | Notes                                                                 |
|:------------------------------|:--------------------------------|:------------------------|:---------------------------|:----------------------------------------------------------------------|
| **List (Dynamic Array)**      | Access by Index `[i]`           | `O(1)`                  | `O(1)`                     | Direct memory access.                                                 |
|                               | Search for Value `in`           | `O(n)`                  | `O(n)`                     | Scans the entire list.                                                |
|                               | Append to End `append()`        | `O(1)` (Amortized)      | `O(n)`                     | Amortized due to occasional resizing.                                 |
|                               | Insert/Delete at Start/Middle   | `O(n)`                  | `O(n)`                     | Requires shifting subsequent elements.                                |
|                               | Pop from End `pop()`            | `O(1)`                  | `O(1)`                     |                                                                       |
|                               | Pop from Start `pop(0)`         | `O(n)`                  | `O(n)`                     | Inefficient; use `collections.deque` for this.                        |
| **Stack (LIFO via `list`)**   | Push `append()`                 | `O(1)` (Amortized)      | `O(n)`                     | Same as list append.                                                  |
|                               | Pop `pop()`                     | `O(1)`                  | `O(1)`                     |                                                                       |
|                               | Peek `[-1]`                     | `O(1)`                  | `O(1)`                     |                                                                       |
| **Queue (FIFO via `deque`)**  | Enqueue `append()`              | `O(1)`                  | `O(1)`                     | `collections.deque` is implemented as a doubly-linked list.           |
|                               | Dequeue `popleft()`             | `O(1)`                  | `O(1)`                     | Highly efficient.                                                     |
|                               | Peek `[0]`                      | `O(1)`                  | `O(1)`                     |                                                                       |
| **Singly Linked List**        | Access / Search                 | `O(n)`                  | `O(n)`                     | Requires traversal from the head.                                     |
|                               | Insert / Delete at Head         | `O(1)`                  | `O(1)`                     | Just re-pointing the head.                                            |
|                               | Insert at End                   | `O(n)`                  | `O(n)`                     | Requires traversal to the end (unless a tail pointer is kept).        |
|                               | Delete at End                   | `O(n)`                  | `O(n)`                     | Requires traversal to find the second-to-last node.                   |
| **Hash Table (`dict`)**       | Access / Search / Insert / Delete | `O(1)` (Amortized)    | `O(n)`                     | Worst case is extremely rare and occurs with many hash collisions.    |
| **Binary Search Tree (BST)**  | Access / Search / Insert / Delete | `O(log n)`            | `O(n)`                     | `O(log n)` assumes a balanced tree. Worst case is an unbalanced tree.|


PS C:\PROGRAMMES VSCODE\AI coding>
```

OBSERVATION:

This script provides a clear and concise summary of time complexities for common Python data structures by printing a pre-formatted markdown table. The information is accurate and well-organized, covering average and worst-case scenarios for essential operations like access, search, and insertion.

TASK-10

QUESTION:

0 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:

o Stack

o Queue

o Priority Queue

o Linked List

o Binary Search Tree (BST)

o Graph

o Hash Table

o 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.

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

PROMPT:

Generate a python code For a Campus Resource Management System, map each feature to the best data structure from [Stack, Queue, Priority Queue, Linked List, BST, Graph, Hash Table, Deque]. Give justification in 2–3 sentences for each choice in a table format.

Then, implement one feature (e.g., Cafeteria Order Queue using Queue) in Python

with docstrings, comments, and at least 3 assert test cases.

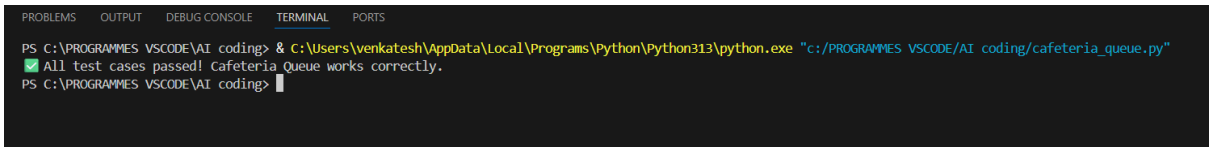
CODE:

```python
from collections import deque
from typing import Any, Deque, Optional

class CafeteriaOrderQueue:
    """A queue system to manage cafeteria orders using a FIFO principle."""

    def __init__(self) -> None:
        """Initialize an empty queue using deque."""
        self._queue: Deque[Any] = deque()

    def place_order(self, order: Any) -> None:
        """
        Add a new order to the queue.

        Args:
            order (Any): The order description.
        """
        self._queue.append(order)

    def serve_order(self) -> Any:
        """
        Serve the next order in the queue.

        Returns:
            Any: The order being served.

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

```python
class CafeteriaOrderQueue:
    def serve_order(self) -> Any:
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("No orders to serve.")
        return self._queue.popleft()

    def peek_order(self) -> Optional[Any]:
        """
        Check the next order without removing it.

        Returns:
            Optional[Any]: The next order, or None if empty.
        """
        if self.is_empty():
            return None
        return self._queue[0]

    def is_empty(self) -> bool:
        """Check if the queue is empty."""
        return len(self._queue) == 0

    def __len__(self) -> int:
        """Returns the number of items in the queue."""
        return len(self._queue)
```

```python

# ---------- Test Cases ----------
if __name__ == "__main__":
    cafeteria = CafeteriaOrderQueue()

    # Place orders
    cafeteria.place_order("Burger")
    cafeteria.place_order("Pizza")
    cafeteria.place_order("Sandwich")

    # Assertions to verify functionality
    assert len(cafeteria) == 3, "Test Failed: Length should be 3 after adding orders."
    assert cafeteria.peek_order() == "Burger"
    assert cafeteria.serve_order() == "Burger"
    assert cafeteria.peek_order() == "Pizza"
    assert not cafeteria.is_empty()
    assert cafeteria.serve_order() == "Pizza"
    assert cafeteria.serve_order() == "Sandwich"
    assert cafeteria.is_empty()
    assert len(cafeteria) == 0, "Test Failed: Length should be 0 when empty."

    print("✅ All test cases passed! Cafeteria Queue works correctly.")

```

OUTPUT:



OBSERVATION:

This code provides a clean and efficient implementation of a FIFO queue, correctly leveraging collections deque for optimal O(1) performance on order operations. The class is made robust and reliable through clear method naming, comprehensive docstrings, and a solid suite of self-verifying assert test cases.