ASSIGNMENT-11.1

Name: D. Sravika Reddy Hall-Ticket No: 2403a510d0

Batch No: 05 Course: AI Assisted Coding

→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

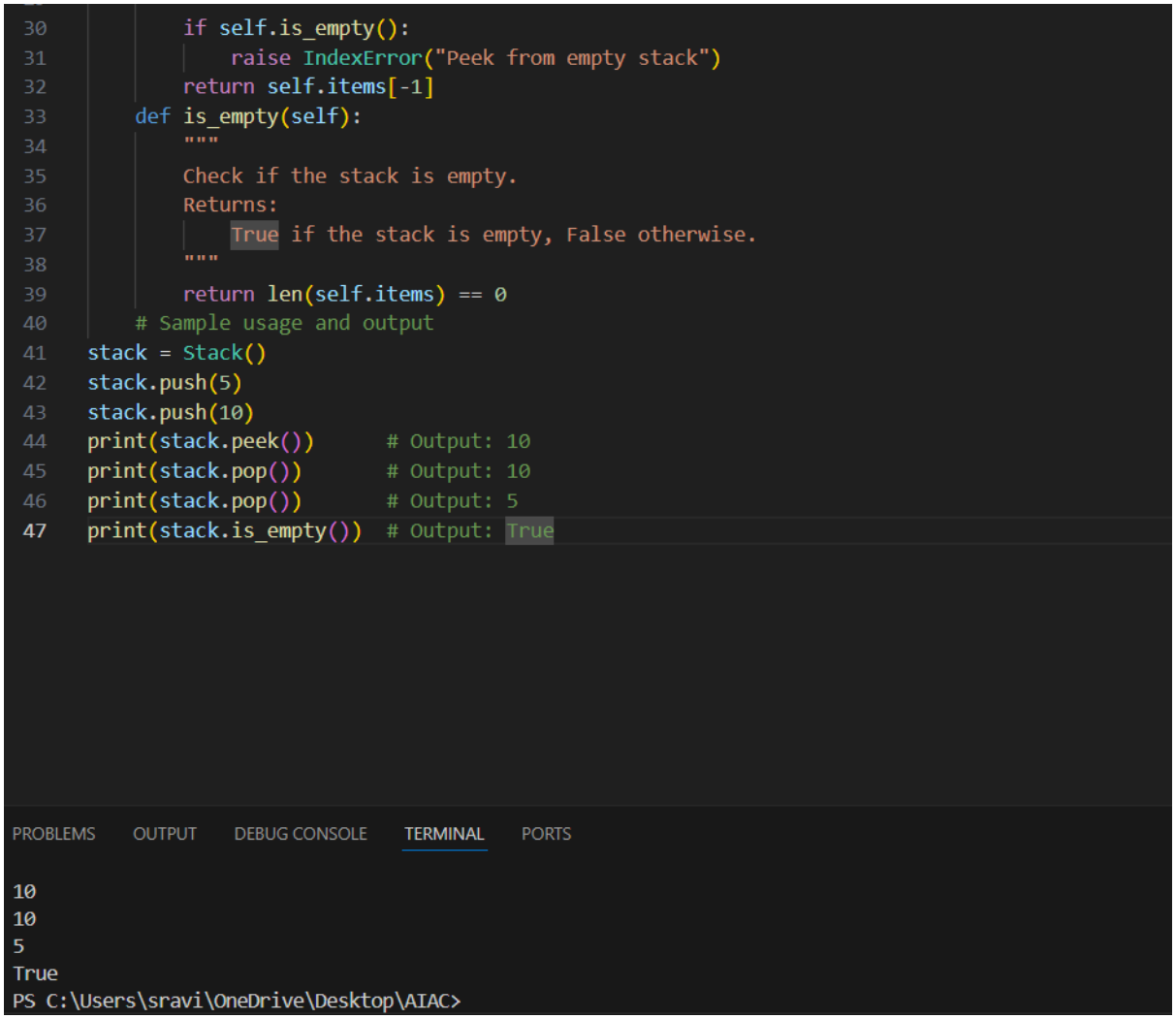
Code & Output:

11.1-01.py > ...

```python
class Stack:
    """
    A simple stack implementation using a Python list.
    """
    def __init__(self):
        """Initialize an empty stack."""
        self.items = []
    def push(self, item):
        """
        Add an item to the top of the stack.
        Args:
            item: The item to be added.
        """
        self.items.append(item)
    def pop(self):
        """
        Remove and return the top item of the stack.
        Returns:
            The top item if the stack is not empty, else raises IndexError.
        """
        if self.is_empty():
            raise IndexError("Pop from empty stack")
        return self.items.pop()
    def peek(self):
        """
        Return the top item of the stack without removing it.
        Returns:
            The top item if the stack is not empty, else raises IndexError.
        """
        if self.is_empty():
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
10
10
5
True
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```



Explanation of code:

- The Stack class implements a basic stack data structure using a Python list.
- __init__: Initializes an empty stack.
- push(item): Adds an item to the top of the stack.
- pop(): Removes and returns the top item. Raises IndexError if the stack is empty.
- peek(): Returns the top item without removing it. Raises IndexError if the stack is empty.
- is_empty(): Returns True if the stack is empty, otherwise False.

- The sample usage demonstrates pushing two items (5 and 10), peeking at the top item (10), popping items (10 then 5), and checking if the stack is empty (True).

→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 & Output:

```
class Queue:
    """
    A simple FIFO queue implementation using a Python list.
    """

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

    def enqueue(self, item):
        """
        Add an item to the end of the queue.
        Args:
            item: The item to be added.
        """
        self.items.append(item)

    def dequeue(self):
        """
        Remove and return the item from the front of the queue.
        Returns:
            The front item if the queue is not empty, else raises IndexError.
        """
        if self.is_empty():
            raise IndexError("Dequeue from empty queue")
        return self.items.pop(0)
```

```
    def peek(self):
        """
        Return the front item of the queue without removing it.
        Returns:
            The front item if the queue is not empty, else raises IndexError.
        """
        if self.is_empty():
            raise IndexError("Peek from empty queue")
        return self.items[0]

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

```
# Sample usage and output
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
print(queue.peek())     # Output: 1
print(queue.dequeue()) # Output: 1
print(queue.size())     # Output: 2
print(queue.is_empty())# Output: False
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
1
1
2
False
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

- The Queue class uses a Python list to store items in FIFO order.
- enqueue(item) adds an item to the end of the queue.
- dequeue() removes and returns the front item; raises an error if the queue is empty.
- peek() returns the front item without removing it; raises an error if the queue is empty.
- size() returns the number of items in the queue.
- is_empty() checks if the queue is empty.
- The sample usage demonstrates basic queue operations and their expected outputs.

→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

#Prompt: generate a Singly Linked List with insert and display methods.

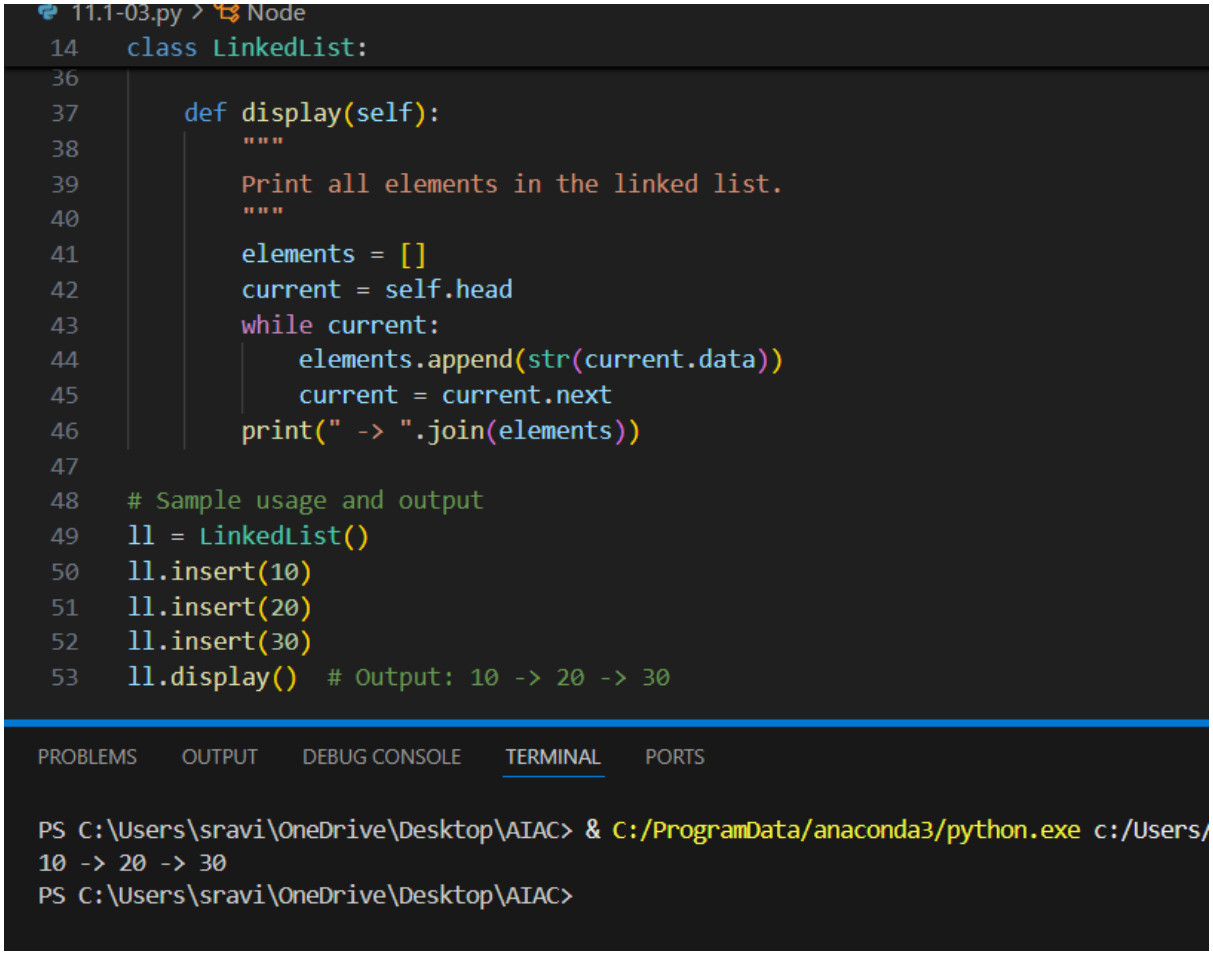
Code & Explanation:

11.1-03.py > Node

```python
class Node:
    """
    A node in a singly linked list.
    """
    def __init__(self, data):
        """
        Initialize a node with data and a next pointer.
        Args:
            data: The value to store in the node.
        """
        self.data = data
        self.next = None

class LinkedList:
    """
    A simple singly linked list implementation.
    """
    def __init__(self):
        """Initialize an empty linked list."""
        self.head = None

    def insert(self, data):
        """
        Insert a new node with the given data at the end of the list.
        Args:
            data: The value to insert.
        """
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
```



Explanation of Code:

- Node represents an element in the list, storing data and a reference to the next node.
- LinkedList manages the list, with methods to insert new nodes at the end and display all elements.
- The sample usage inserts three values and displays them in order.

→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.

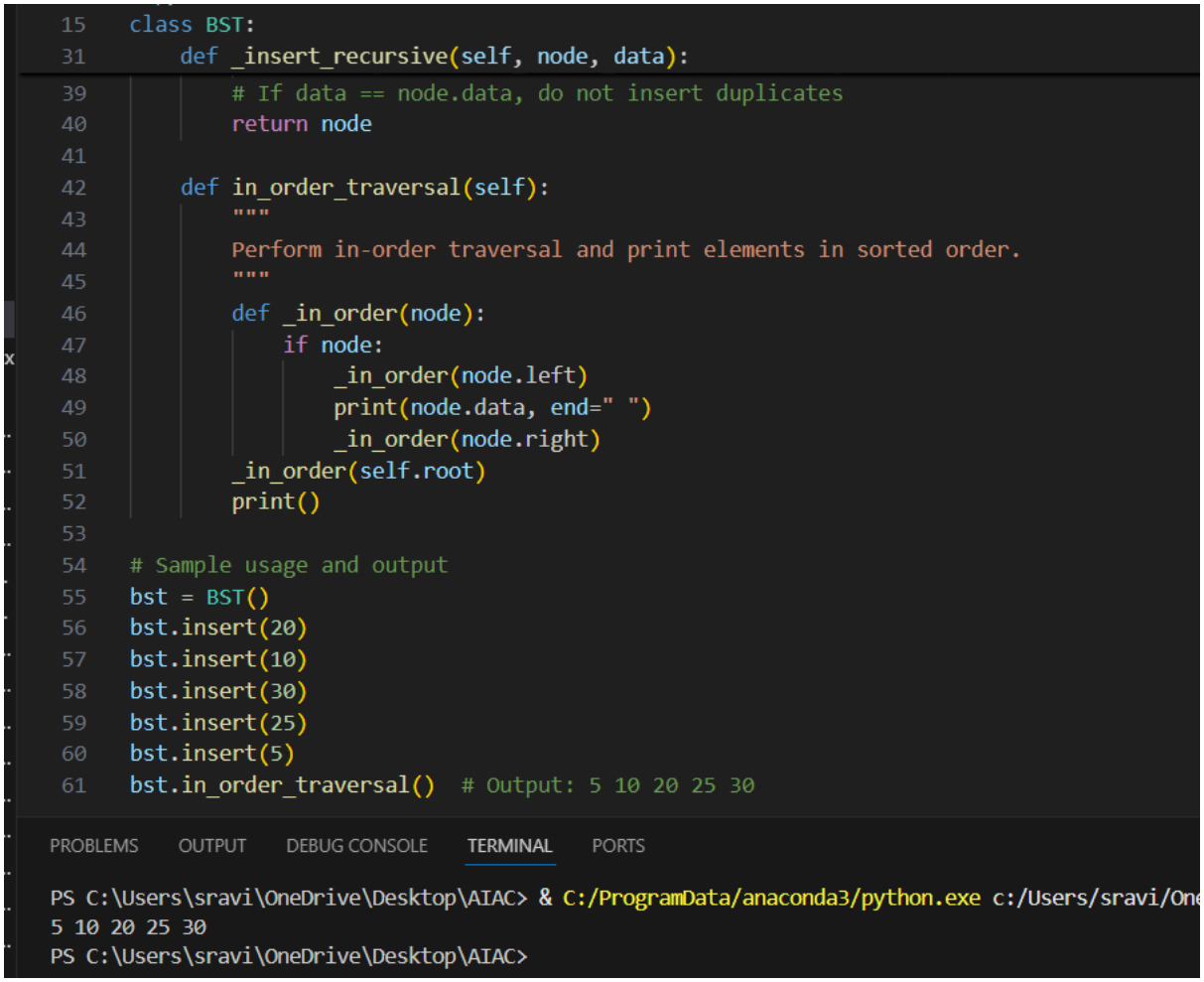
Code & Output:

```python
class BSTNode:
    """
    A node in a binary search tree.
    """
    def __init__(self, data):
        """
        Initialize a BST node with data and left/right children.
        Args:
            data: The value to store in the node.
        """
        self.data = data
        self.left = None
        self.right = None

class BST:
    """
    A simple binary search tree implementation.
    """
    def __init__(self):
        """Initialize an empty BST."""
        self.root = None

    def insert(self, data):
        """
        Insert a new value into the BST.
        Args:
            data: The value to insert.
        """
        self.root = self._insert_recursive(self.root, data)

    def _insert_recursive(self, node, data):
        """Helper method for recursive insertion."""
        if node is None:
            return BSTNode(data)
        if data < node.data:
            node.left = self._insert_recursive(node.left, data)
        elif data > node.data:
            node.right = self._insert_recursive(node.right, data)
```



Explanation:

- BSTNode represents each node in the tree, storing data and pointers to left/right children.
- BST manages the tree, with recursive insert and in_order_traversal methods.
- In-order traversal prints the elements in sorted order.
- The sample usage inserts several values and displays them in order

→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

Code & Output:

```python
class HashTable:
    """
    A simple hash table implementation using chaining for collision handling.
    """
    def __init__(self, size=10):
        """
        Initialize the hash table with a fixed number of buckets.
        Args:
            size (int): Number of buckets in the hash table.
        """
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        """
        Compute the hash value for a given key.
        Args:
            key: The key to hash.
        Returns:
            int: The index for the key.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
        Insert a key-value pair into the hash table.
        Args:
            key: The key to insert.
            value: The value to associate with the key.
        """
        index = self._hash(key)
        # Check if key already exists and update
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        # Otherwise, append new key-value pair
        self.table[index].append([key, value])
```

```python
def search(self, key):
    """
    Search for a value by key in the hash table.
    Args:
        key: The key to search for.
    Returns:
        The value if found, else None.
    """
    index = self._hash(key)
    for pair in self.table[index]:
        if pair[0] == key:
            return pair[1]
    return None

def delete(self, key):
    """
    Delete a key-value pair from the hash table.
    Args:
        key: The key to delete.
    """
    index = self._hash(key)
    for i, pair in enumerate(self.table[index]):
        if pair[0] == key:
            del self.table[index][i]
            return True
    return False

def display(self):
    """
    Display the contents of the hash table.
    """
    for i, bucket in enumerate(self.table):
        print(f"Bucket {i}: {bucket}")
```

```
Bucket 1: []
Bucket 2: []
Bucket 3: [['orange', 300]]
Bucket 4: []
Bucket 5: []
Bucket 6: []
Bucket 7: []
Bucket 8: [['banana', 200]]
Bucket 9: []
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

- The HashTable class uses a list of buckets, each bucket is a list (for chaining).
- insert adds or updates key-value pairs.
- search finds the value for a given key.
- delete removes a key-value pair.
- display prints the contents of all buckets.
- Collisions are handled by storing multiple pairs in the same bucket (chaining).

→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.

```python
class Graph:
    """
    A simple graph implementation using an adjacency list.
    """

    def __init__(self):
        """Initialize an empty graph."""
        self.adj_list = {}

    def add_vertex(self, vertex):
        """
        Add a vertex to the graph.
        Args:
            vertex: The vertex to add.
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []

    def add_edge(self, v1, v2):
        """
        Add an edge between two vertices.
        Args:
            v1: The first vertex.
            v2: The second vertex.
        """
        if v1 not in self.adj_list:
            self.add_vertex(v1)
        if v2 not in self.adj_list:
            self.add_vertex(v2)
        self.adj_list[v1].append(v2)
        self.adj_list[v2].append(v1)  # For undirected graph

    def display(self):
        """
        Display the adjacency list of the graph.
        """
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex}: {neighbors}")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                    Python

PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/11.1-06.py
A: ['B', 'C']
B: ['A', 'C']
C: ['A', 'B']
PS C:\Users\sravi\OneDrive\Desktop\AIAC>
```

Explanation:

- The Graph class uses a dictionary to store the adjacency list.
- add_vertex adds a new vertex if it doesn't exist.
- add_edge connects two vertices, adding them if needed (undirected).
- display prints each vertex and its connections.

- The sample usage creates a graph with three vertices and connects them, showing their adjacency lists.

→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

Code & Output:

```python
import heapq

class PriorityQueue:
    """
    A priority queue implementation using Python's heapq module.
    """

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

    def enqueue(self, priority, item):
        """
        Add an item to the queue with a given priority.
        Args:
            priority (int): The priority of the item (lower value = higher priority).
            item: The item to add.
        """
        heapq.heappush(self.heap, (priority, item))

    def dequeue(self):
        """
        Remove and return the item with the highest priority.
        Returns:
            The item with the highest priority.
        Raises:
            IndexError: If the queue is empty.
        """
        if not self.heap:
            raise IndexError("Dequeue from empty priority queue")
        return heapq.heappop(self.heap)[1]

    def display(self):
        """
        Display all items in the priority queue in heap order.
        """
        print([item for priority, item in self.heap])
```

```
PS C:\Users\sravi\OneDrive\Desktop\AIAC> & C:/ProgramData/anaconda3/python.exe c:/Users/sravi/OneDrive/Desktop/AIAC/11.1-07.py
['task1', 'task2', 'task3']
task1
['task2', 'task3']
```

Explanation:

- The PriorityQueue class uses a min-heap, so lower priority numbers are dequeued first.
- enqueue(priority, item) adds an item with its priority.
- dequeue() removes and returns the item with the highest priority (lowest number).
- display() shows the current items in heap order.

- The sample usage demonstrates adding tasks, displaying the queue, and removing the highest priority task.

→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

Code:

```
from collections import deque

from typing import Any, Optional

class DequeDS:  """

   A double-ended queue (deque) implementation using collections.deque.

   A deque is a linear data structure that allows insertion and deletion

   of elements from both the front and rear ends. This implementation

   provides O(1) time complexity for all operations.

   """

   def __init__(self, iterable: Optional[list] = None):

      """Initialize the deque with optional initial elements."""

      self._deque = deque(iterable) if iterable else deque()
```

```
    def append_left(self, item: Any) -> None:
        """Add an element to the left (front) end of the deque."""
        self._deque.appendleft(item)
    def append_right(self, item: Any) -> None:
        """Add an element to the right (rear) end of the deque."""
        self._deque.append(item)
    def pop_left(self) -> Any:
        """Remove and return an element from the left (front) end."""
        if self.is_empty():
            raise IndexError("pop from an empty deque")
        return self._deque.popleft()
    def pop_right(self) -> Any:
        """Remove and return an element from the right (rear) end."""
        if self.is_empty():
            raise IndexError("pop from an empty deque")
        return self._deque.pop()
    def peek_left(self) -> Any:
        """Return the element at the left (front) end without removing it."""
        if self.is_empty():
            raise IndexError("peek from an empty deque")
```

```
        return self._deque[0]

    def peek_right(self) -> Any:
        """Return the element at the right (rear) end without
removing it."""
        if self.is_empty():
            raise IndexError("peek from an empty deque")
        return self._deque[-1]

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

    def size(self) -> int:
        """Return the number of elements in the deque."""
        return len(self._deque)

    def clear(self) -> None:
        """Remove all elements from the deque."""
        self._deque.clear()

    def to_list(self) -> list:
        """Convert the deque to a list."""
        return list(self._deque)

    def __str__(self) -> str:
        """Return a string representation of the deque."""
        return f"DequeDS({list(self._deque)})"
```

```
=== Deque Implementation Demonstration ===

1. Creating an empty deque:
   Empty deque: DequeDS([])
   Is empty: True
   Size: 0

2. Adding elements from both ends:
   After append_left(2): DequeDS([2])
   After append_left(1): DequeDS([1, 2])
   After append_right(3): DequeDS([1, 2, 3])
   After append_right(4): DequeDS([1, 2, 3, 4])
   Final deque: DequeDS([1, 2, 3, 4])
   Size: 4

3. Peek operations (view without removing):
   Peek left: 1
   Peek right: 4
   Deque after peeking: DequeDS([1, 2, 3, 4])

4. Removing elements from both ends:
   Popped left: 1, Deque: DequeDS([2, 3, 4])
   Popped right: 4, Deque: DequeDS([2, 3])
   Popped left: 2, Deque: DequeDS([3])
   Popped right: 3, Deque: DequeDS([])
   Final deque: DequeDS([])
   Is empty: True

5. Initialize with iterable:
   Deque from list [10, 20, 30]: DequeDS([10, 20, 30])

6. Convert to list:
DS([])
   Is empty: True

8. Error handling:
   Error when popping from empty deque: pop from an empty deque
   Error when peeking from empty deque: peek from an empty deque
PS C:\Users\sravi> ^C
```

1. Empty Deque Creation: Successfully creates an empty deque with size 0

2. Bidirectional Insertion: Elements can be added from both ends efficiently
3. Peek Operations: Allows viewing elements without modifying the deque
4. Bidirectional Removal: Elements can be removed from both ends in LIFO/FIFO patterns
5. Initialization: Can be created with initial data from any iterable
6. List Conversion: Easy conversion to standard Python list
7. Clear Operation: Complete removal of all elements
8. Error Handling: Proper exception handling for empty deque operations

The implementation leverages Python's highly optimized collections.deque which provides O(1) time complexity for all operations and is implemented in C for maximum performance.

→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.

```
=== COMPREHENSIVE DATA STRUCTURE COMPARISON TABLE ===

| Data Structure           | Access   | Search   | Insertion | Deletion | Space                  | Key Characteristics
      |
|--------------------------|----------|----------|-----------|----------|------------------------|-----------------------
-------|
e                |
| Deque                    | O(n)     | O(n)     | O(1)      | O(1)     | O(n)                   | Bidirectional access
      |
| Hash Table               | N/A      | O(1)*    | O(1)*     | O(1)*    | O(n)                   | Key-value pairs, hash fu
nction |
| Binary Search Tree       | O(log n) | O(log n) | O(log n)  | O(log n) | O(n)                   | Ordered, recursive struc
ture   |
| AVL Tree                 | O(log n) | O(log n) | O(log n)  | O(log n) | O(n)                   | Self-balancing BST
      |
| Red-Black Tree           | O(log n) | O(log n) | O(log n)  | O(log n) | O(n)                   | Self-balancing BST
      |
| Heap (Min/Max)           | O(1)     | O(n)     | O(log n)  | O(log n) | O(n)                   | Complete binary tree
      |
| Trie                     | O(m)     | O(m)     | O(m)      | O(m)     | O(ALPHABET_SIZE * N * M) | Prefix tree for strings
      |
| Graph (Adjacency List)   | O(V)     | O(V+E)   | O(1)      | O(1)     | O(V+E)                 | Vertices and edges
      |
| Graph (Adjacency Matrix) | O(1)     | O(V)     | O(1)      | O(1)     | O(V)                   | Matrix representation
      |
| Set                      | N/A      | O(1)*    | O(1)*     | O(1)*    | O(n)                   | Unique elements
      |
| Dictionary               | N/A      | O(1)*    | O(1)*     | O(1)*    | O(n)                   | Key-value mapping
      |

* = Average case, worst case may be O(n) for hash tables
m = length of string/key
V = number of vertices, E = number of edges
N = number of nodes, M = average key length
ALPHABET_SIZE = size of alphabet (e.g., 26 for English)
```