# NAME:SAI KUSHAL

# ROLL NO:2403A51338

# BATCH:14

# DATE:08-10-2025

# ASSIGNMENT-11.2

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

PROMPT: Create a Python Stack class with push, pop, peek, and is\_empty methods.

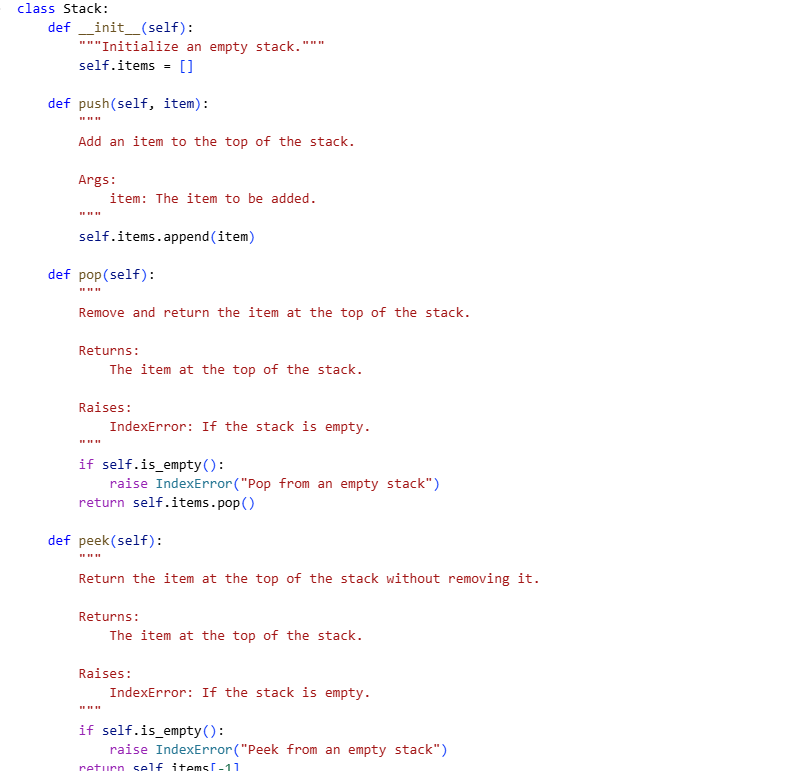
Include proper docstrings for each method.

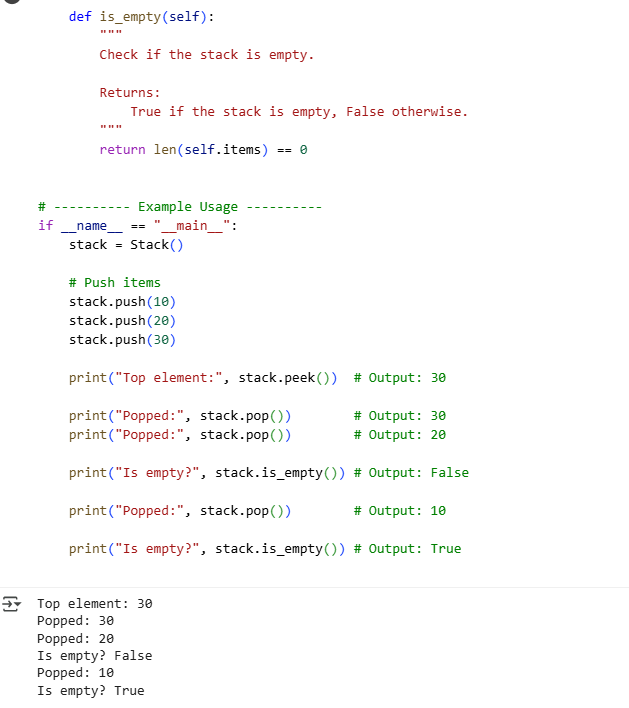
Raise IndexError when popping or peeking from an empty stack.

After the class, add example usage that demonstrates all methods.

Output everything as a single complete code block.

## CODE&OUTPUT:





### EXPLAINATION:

 This prompt asks the AI to create a Stack class in Python.

 The stack should have four key methods: push, pop, peek, and is\_empty.

 Each method must include a docstring to explain its purpose and usage.

 When the user tries to pop or peek from an empty stack, the method should raise an IndexError.

 The stack should use a Python list internally to store its items.

 After defining the class, the prompt requests a demonstration of how to use it.

 The example should push three items (like 10, 20, and 30) onto the stack.

 It should then call peek to show the current top value without removing it.

 The example must also show how to pop items off the stack and display them.

 Finally, it should use is\_empty to check whether the stack has any items left, before and after popping everything.

Task Description #2 **– Queue Implementation**

Task: Use AI to implement a Queue using Python lists.

Sample Input Code:

class Queue:

pass

PROMPT: Implement a Python Queue class using lists with the following methods: enqueue(item), dequeue(), peek(), and size().

- enqueue(item) adds an item to the queue.

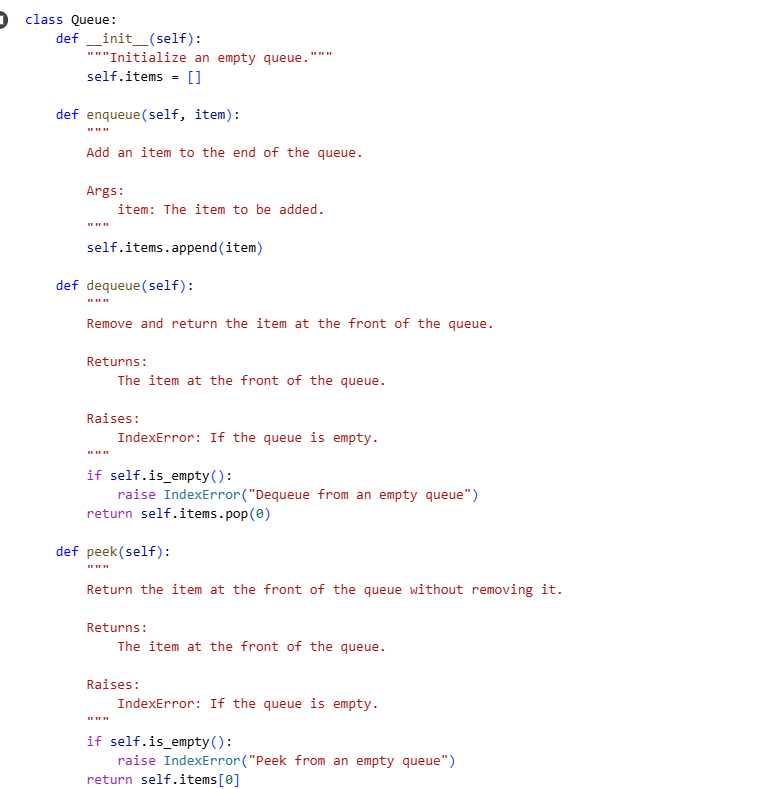
- dequeue() removes and returns the front item, raising IndexError if empty.

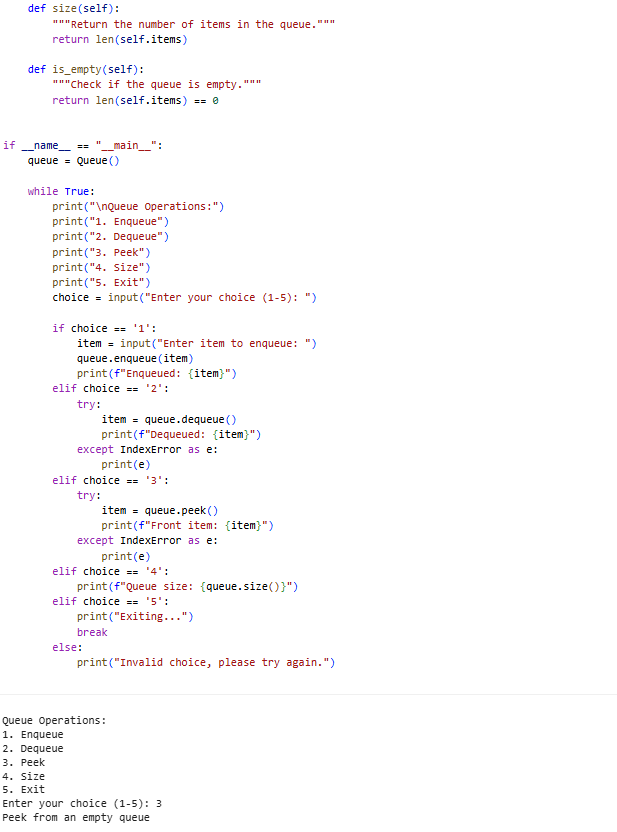
- peek() returns the front item without removing it, raising IndexError if empty.

- size() returns the number of items in the queue.

After the class, write an interactive console program that lets the user perform enqueue, dequeue, peek, size, or exit operations repeatedly.

# CODE&OUTPUT:





## EXPLANATION:

 Create a Python Queue class that uses a list internally to store elements.

 Implement an enqueue(item) method to add an item at the end of the queue.

 Implement a dequeue() method to remove and return the front item; raise IndexError if empty.

 Implement a peek() method to return the front item without removing it; raise IndexError if empty.

 Implement a size() method to return the number of items in the queue.

 Each method should have clear docstrings explaining their behavior.

 After the class definition, write an interactive console program.

 The program should repeatedly let the user choose to enqueue, dequeue, peek, check size, or exit.

 Input from the user should be used to perform the corresponding queue operations.

 Handle errors gracefully, like trying to dequeue or peek when the queue is empty.

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

PROMPT: Write a Python program that implements a singly linked list with two classes: Node and LinkedList.

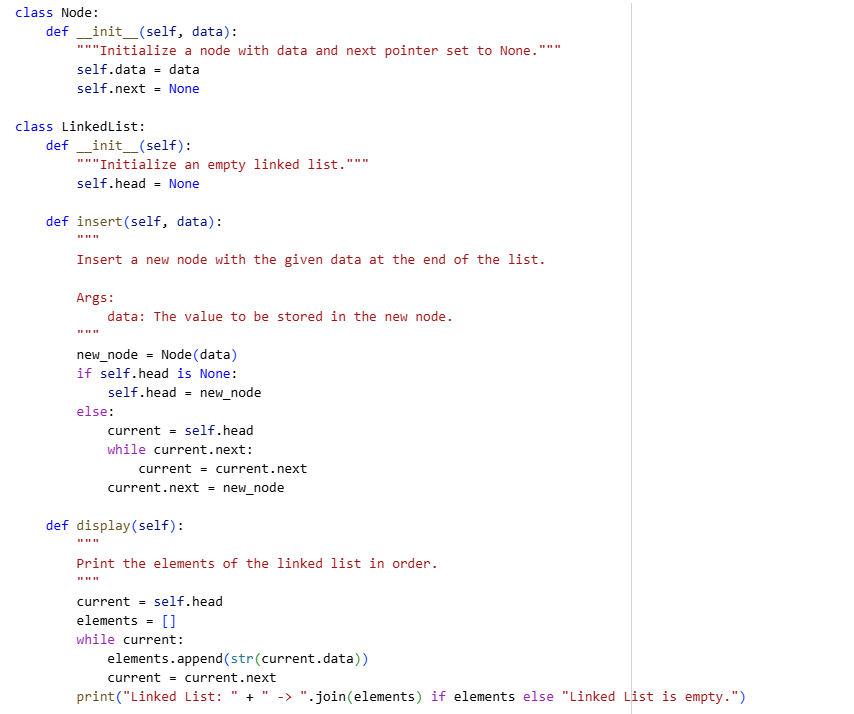
The LinkedList class should have an insert(data) method that adds nodes to the end of the list, and a display() method that prints all node values in order.

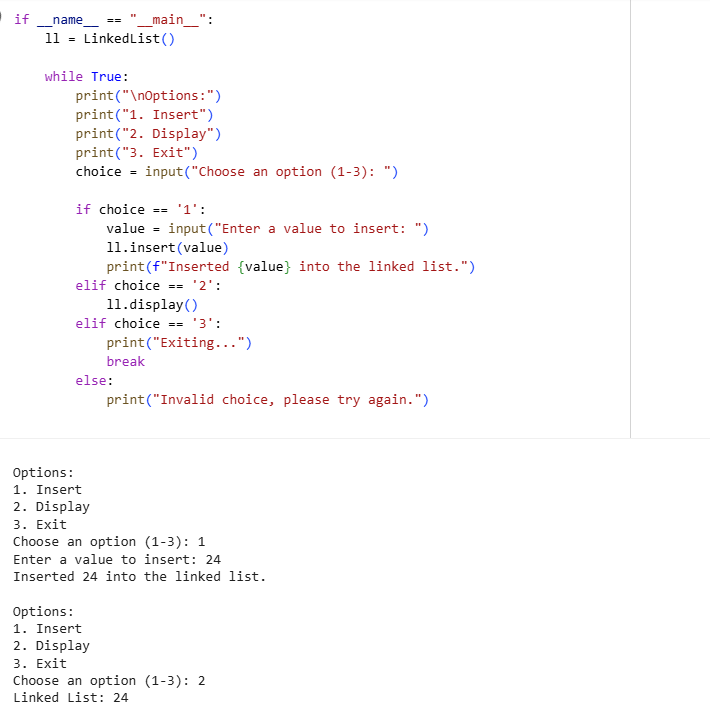
After defining the classes, include an interactive menu in the main program that lets users:

1) Insert a value into the linked list,

2) Display the linked list

# CODE&OUTPUT:





### EXPLAINATION:

 Implement two Python classes: Node and LinkedList.

 The Node class stores a data value and a reference to the next node.

 The LinkedList class manages the list, starting with a head pointer.

 Include an insert(data) method to add nodes at the end of the list.

 Include a display() method that prints all node values in order.

 After the classes, create an interactive menu-driven program for the user.

 The menu should offer options: insert a value, display the list, or exit.

 Use input() to capture user choices and values for insertion.

 Handle invalid inputs gracefully, prompting the user again if needed.

 Add docstrings to all methods explaining their purpose and usage.

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

## PROMPT:

Create a Python class for a Binary Search Tree (BST) with insert and in-order traversal methods.

- The insert(data) method should add a node with the given integer data into the BST.

- The in\_order\_traversal() method should return a list of all node values in sorted order.

After the class definitions, write an interactive console program that repeatedly lets the user:

1) Insert integers into the BST,

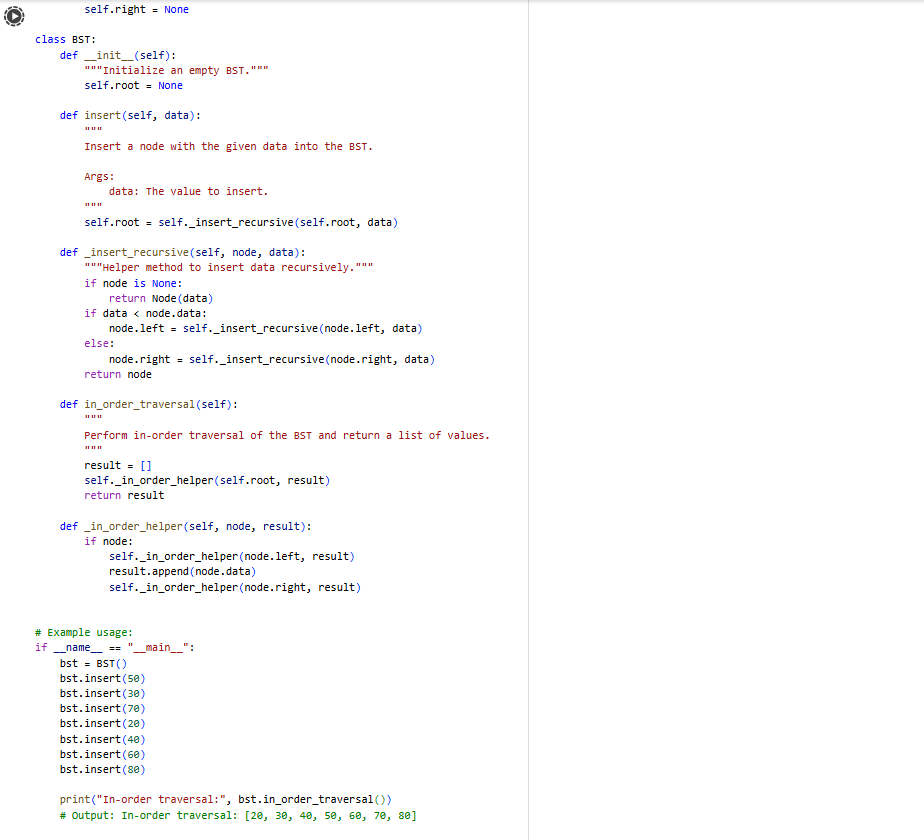
2) Display the in-order traversal of the BST, or

3) Exit the program.

Use input() to read user choices and data, and handle invalid inputs gracefully.

Include docstrings for all methods.

## CODE&OUTPUT:



### EXPLAINATION:

 Implement two Python classes: Node and LinkedList.

 The Node class stores a data value and a reference to the next node.

 The LinkedList class manages the list, starting with a head pointer.

 Include an insert(data) method to add nodes at the end of the list.

 Include a display() method that prints all node values in order.

 After the classes, create an interactive menu-driven program for the user.

 The menu should offer options: insert a value, display the list, or exit.

 Use input() to capture user choices and values for insertion.

 Handle invalid inputs gracefully, prompting the user again if needed.

 Add docstrings to all methods explaining their purpose and usage.

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

### PROMPT:

Write a Python class `HashTable` that uses separate chaining to handle collisions.

Implement the following methods:

- `insert(key, value)` to add or update key-value pairs,

- `search(key)` to return the value for a given key or None if not found,

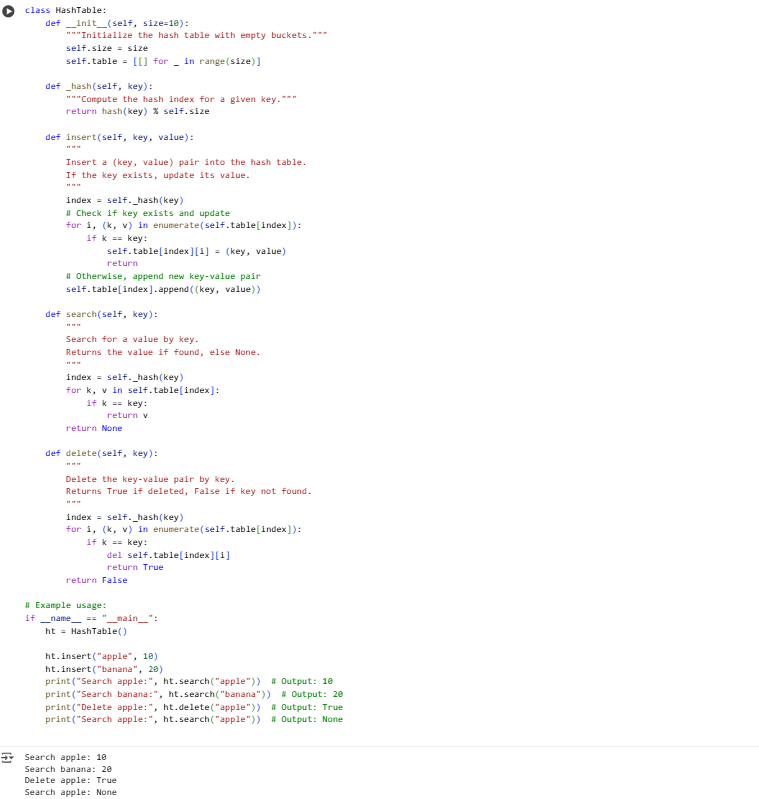
- `delete(key)` to remove the key-value pair and return True if deleted or False if key not found.

Use a fixed-size internal list of buckets and a hash function to compute bucket indices.

Include docstrings for all methods.

After the class, provide example usage demonstrating insert, search, and delete operations.

## CODE&OUTPUT:



## EXPLAINATION:

 Create a HashTable class using a list of buckets to store data.

 Use a hash function to convert keys into bucket indices.

 Handle collisions with separate chaining—store multiple items in each bucket.

 The insert(key, value) method adds or updates a key-value pair in the table.

 The search(key) method looks for a key and returns its value if found.

 The delete(key) method removes a key-value pair and returns success status.

 Initialize the hash table with a fixed number of empty buckets.

 Each bucket contains a list of key-value tuples to manage collisions.

 Include docstrings to explain each method’s purpose and behavior.

 Provide example usage to demonstrate inserting, searching, and deleting keys.

Task Description #6 **– Graph Representation**

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

Sample Input Code:

class Graph:

pass

## PROMPT:

Write a Python class `Graph` that uses an adjacency list to represent a graph.

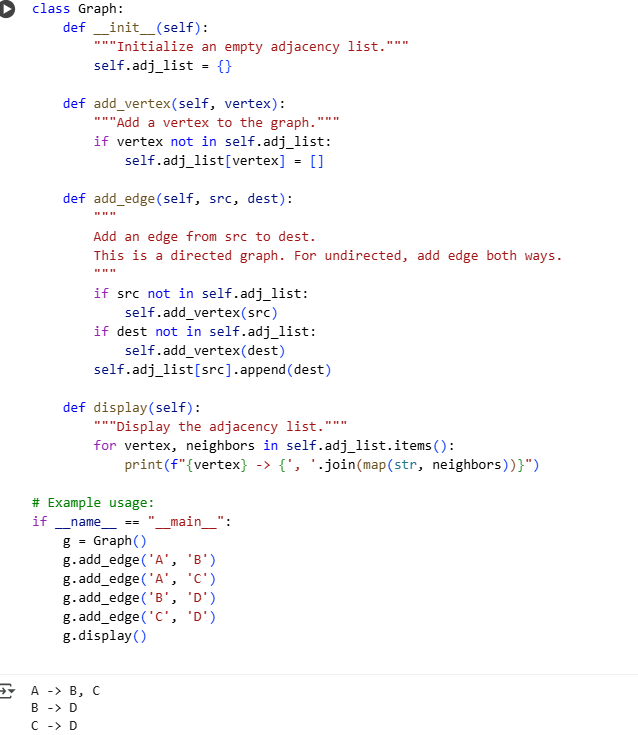
Include methods to add vertices and add edges (directed).

Implement a method to display the adjacency list of the graph.

After defining the class, show example usage by creating a graph, adding edges, and displaying the adjacency list.

Include docstrings for all methods explaining their purpose.

## CODE&OUTPUT:



## EXPLAINATION:

 Create a Graph class that uses a dictionary for the adjacency list.

 Implement an add\_vertex method to add new vertices if they don’t exist.

 Implement an add\_edge method to add a directed edge from one vertex to another.

 If vertices don’t exist when adding an edge, create them automatically.

 The adjacency list stores each vertex’s neighbors as a list.

 Implement a display method to print the adjacency list clearly.

 The graph is directed; edges go from source to destination only.

 The example code demonstrates adding edges and showing the graph.

 Each method includes docstrings to describe its functionality.

 This setup allows easy extension for traversal or other graph algorithms.

Task Description #7 **– Priority Queue**

Task: Use AI to implement a priority queue using Python’s heapq module.

Sample Input Code:

class PriorityQueue:

pass

## PROMPT:

Write a Python class `PriorityQueue` that uses the `heapq` module to implement a priority queue.

Include methods:

- `push(item, priority)` to add items with priority (lower number = higher priority),

- `pop()` to remove and return the highest priority item,

- `peek()` to view the highest priority item without removing it,

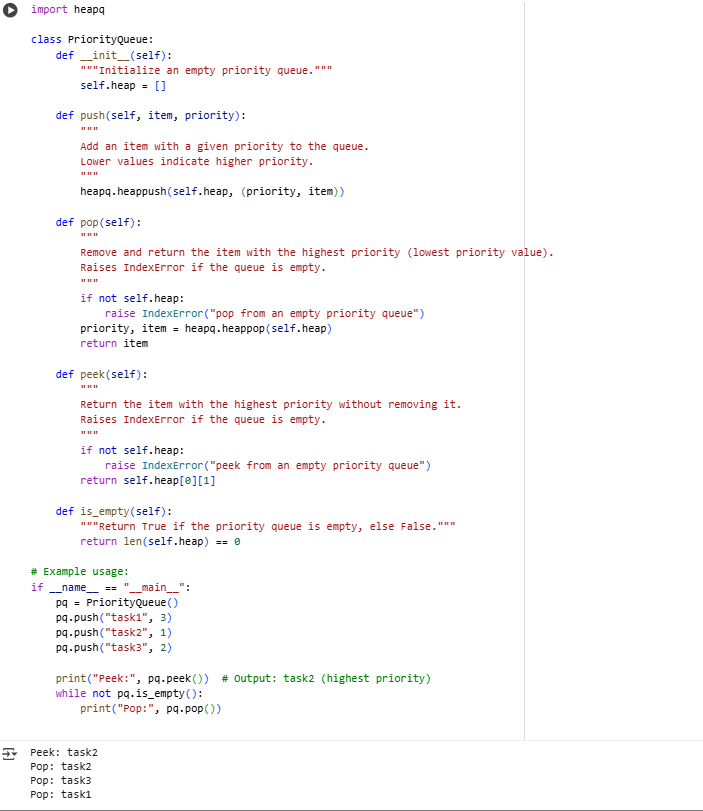
- `is\_empty()` to check if the queue is empty.

Add appropriate error handling for popping or peeking from an empty queue.

Include docstrings for all methods.

Provide example usage demonstrating these methods.

## CODE&OUTPUT:



## EXPLAINATION:

 Create a PriorityQueue class using Python’s built-in heapq module.

 The push(item, priority) method adds an item with its priority to the queue.

 Lower priority numbers mean higher priority (e.g., 1 is higher than 3).

 The pop() method removes and returns the item with the highest priority.

 The peek() method returns the highest priority item without removing it.

 The is\_empty() method checks if the priority queue has no items.

 Handle errors if pop() or peek() is called when the queue is empty.

 Use a list (self.heap) internally to maintain the heap structure.

 Each element in the heap is a tuple (priority, item) for sorting.

 Include docstrings for clarity and provide example code for demonstration.

Task Description #8 **– Deque**

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

Sample Input Code:

class DequeDS:

pass

## PROMPT:

Write a Python class `DequeDS` using `collections.deque` to implement a double-ended queue.

Include methods to:

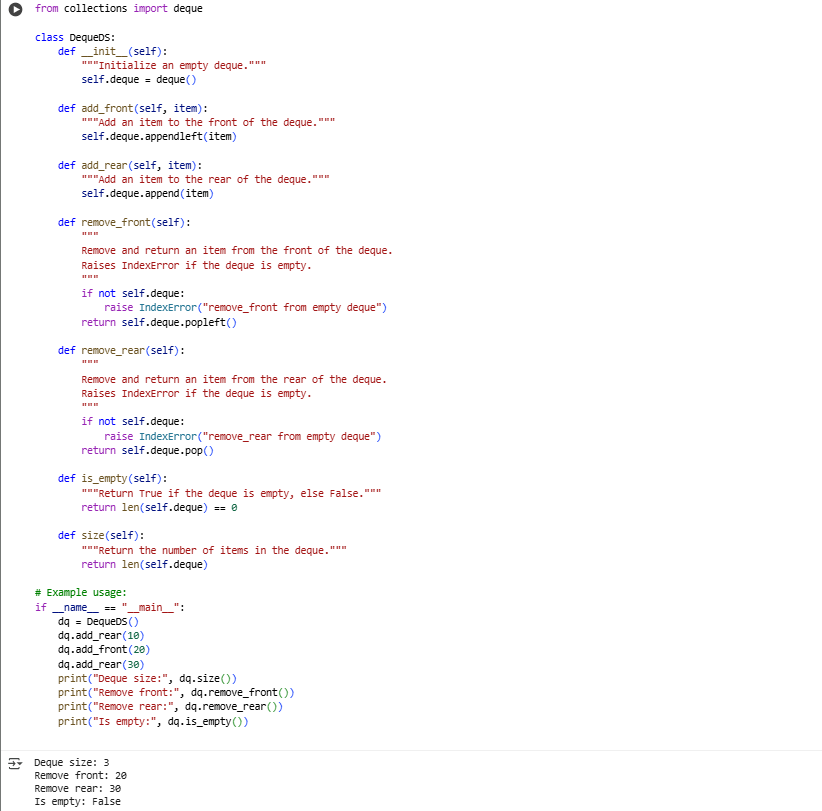
- add items to the front (`add\_front`) and rear (`add\_rear`),

- remove items from the front (`remove\_front`) and rear (`remove\_rear`),

- check if the deque is empty (`is\_empty`),

- return the current size (`size`).

## CODE&OUTPUT:



EXPLAINATION:

 Use Python’s built-in collections.deque to create a double-ended queue.

 The add\_front(item) method adds an element to the front of the deque.

 The add\_rear(item) method adds an element to the rear of the deque.

 The remove\_front() method removes and returns the front element, raising an error if empty.

 The remove\_rear() method removes and returns the rear element, raising an error if empty.

 The is\_empty() method checks if the deque has no elements.

 The size() method returns the number of elements currently in the deque.

 The deque supports efficient additions and removals from both ends.

 Docstrings explain the purpose and behavior of each method.

 Example usage demonstrates all main operations clearly.