# AI ASSISTED CODING

# LAB-11.2

NAME:K.Bharath

ENROLL.NO:2403A52065

BATCH:03

## TASK-01:

Use AI to generate a Stack class with push, pop, peek, and
is_empty methods.
Sample Input Code:
class Stack:
pass

## PROMPT:

Generate a python stack class using push
,pop, is empty.

## CODE:

```python
from typing import Any, List

class Stack:
    """
    A simple implementation of a Stack data structure.
    """
    def __init__(self) -> None:
        """
        Initializes an empty stack.
        """
        self._items: List[Any] = []

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

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

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

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

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

        Raises:
            IndexError: If the stack is empty.

        Returns:
            Any: The item at the top of the stack.
        """
        if self.is_empty():
            raise IndexError("pop from an empty stack")
        return self._items.pop()

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

```python
        Raises:
            IndexError: If the stack is empty.

        Returns:
            Any: The item at the top of the stack.
        """
        if self.is_empty():
            raise IndexError("peek from an empty stack")
        return self._items[-1]

    def __str__(self) -> str:
        """
        Returns a string representation of the stack.
        """
        return str(self._items)

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

# --- Example Usage ---
if __name__ == "__main__":
    print("Creating a new stack...")
    s = Stack()

    print(f"Is the stack empty? {s.is_empty()}")  # Expected: True

    print("\nPushing items: 10, 20, 30")
    s.push(10)
    s.push(20)
    s.push(30)

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

    print(f"\nPeeking at the top item: {s.peek()}")  # Expected: 30
    print(f"Current stack after peek: {s}")

    print(f"\nPopping an item: {s.pop()}")  # Expected: 30
    print(f"Current stack after pop: {s}")

    print(f"Popping another item: {s.pop()}")  # Expected: 20
    print(f"Current stack: {s}")

    print("\nPopping the last item...")
    s.pop()
    print(f"Is the stack empty now? {s.is_empty()}")  # Expected: True
```

# OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.1.py
Creating a new stack...
Is the stack empty? True

Pushing items: 10, 20, 30
Current stack: [10, 20, 30]
Stack size: 3
Is the stack empty? False

Peeking at the top item: 30
Current stack after peek: [10, 20, 30]

Popping an item: 30
Current stack after pop: [10, 20]
Popping another item: 20
Current stack: [10]

Popping the last item...
Is the stack empty now? True
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

# OBSERVATION:

The AI generated the code in an efficient way according to the prompt as it developed the stack class.

# TASK-02:

Use AI to implement a Queue using Python lists.
Sample Input Code:
class Queue:
pass

# PROMPT:

Generate a python code to implement the queue in data structures

# CODE:

```python
from typing import Any, List

class Queue:
    """
    A simple implementation of a Queue data structure using a Python list.
    Note: For performance-critical applications, collections.deque is preferred.
    """
    def __init__(self) -> None:
        """
        Initializes an empty queue.
        """
        self._items: List[Any] = []

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

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

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

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

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

        Raises:
            IndexError: If the queue is empty.

        Returns:
            Any: The item at the front of the queue.
        """
        if self.is_empty():
            raise IndexError("dequeue from an empty queue")
        return self._items.pop(0)

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

        Raises:
            IndexError: If the queue is empty.

        Returns:
            Any: The item at the front of the queue.
        """
        if self.is_empty():
            raise IndexError("peek from an empty queue")
        return self._items[0]
```

```python
class Queue:
        return self._items[0]

    def __str__(self) -> str:
        """
        Returns a string representation of the queue.
        """
        return str(self._items)

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


# --- Example Usage ---
if __name__ == "__main__":
    print("Creating a new queue...")
    q = Queue()

    print(f"Is the queue empty? {q.is_empty()}")  # Expected: True

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

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

    print(f"\nPeeking at the front item: {q.peek()}")  # Expected: 'A'
    print(f"Current queue after peek: {q}")

    print(f"\nDequeuing an item: {q.dequeue()}")  # Expected: 'A'
    print(f"Current queue after dequeue: {q}")

    print(f"Dequeuing another item: {q.dequeue()}")  # Expected: 'B'
    print(f"Current queue: {q}")

    print("\nDequeuing the last item...")
    q.dequeue()
    print(f"Is the queue empty now? {q.is_empty()}")  # Expected: True
```

# OUTPUT:



```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.2.py
Creating a new queue...
Is the queue empty? True

Enqueuing items: 'A', 'B', 'C'
Current queue: ['A', 'B', 'C']
Queue size: 3
Is the queue empty? False

Peeking at the front item: A
Current queue after peek: ['A', 'B', 'C']

Dequeuing an item: A
Current queue after dequeue: ['B', 'C']
Dequeuing another item: B
Current queue: ['C']

Dequeuing the last item...
Is the queue empty now? True
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

# OBSERVATION:

The code generated by AI is more accurate about the queue in data structures and it also passed all the test cases.

# TASK-03:

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

# PROMPT:

Generate a python code of singly linked list with insert and display methods.

# CODE:

```python
from typing import Any, Optional

class Node:
    """
    An object for storing a single node of a linked list.
    Models two attributes: data and the link to the next node in the list.
    """
    def __init__(self, data: Any, next_node: Optional['Node'] = None):
        """
        Initializes a Node.

        Args:
            data (Any): The data to be stored in the node.
            next_node (Optional['Node']): The next node in the list. Defaults to None.
        """
        self.data = data
        self.next = next_node

class LinkedList:
    """
    A simple implementation of a Singly Linked List.
    """
    def __init__(self):
        """
        Initializes an empty linked list.
        """
        self.head: Optional[Node] = None

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

        Returns:
            bool: True if the list is empty, False otherwise.
        """
        return self.head is None

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

        Args:
            data (Any): The data for the new node.
        """
        new_node = Node(data)
        # If the list is empty, the new node becomes the head.
        if self.is_empty():
            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 pointer of the last node to the new node.
        last_node.next = new_node
```

```
19    class LinkedList:
58
59        def display(self) -> None:
60            """
61            Traverses the linked list and prints the data of each node.
62            """
63            if self.is_empty():
64                print("Linked list is empty.")
65                return
66
67            current = self.head
68            nodes = []
69            while current:
70                nodes.append(str(current.data))
71                current = current.next
72            print(" -> ".join(nodes))
73
74    # --- Example Usage ---
75    if __name__ == "__main__":
76        print("Creating a new linked list...")
77        ll = LinkedList()
78
79        print("Is the list empty?", ll.is_empty()) # Expected: True
80        ll.display() # Expected: Linked list is empty.
81
82        print("\nInserting elements: 10, 20, 30")
83        ll.insert(10)
84        ll.insert(20)
85        ll.insert(30)
86
87        print("Is the list empty?", ll.is_empty()) # Expected: False
88
89        print("\nDisplaying the list:")
90        ll.display() # Expected: 10 -> 20 -> 30
91
92        print("\nInserting another element: 40")
93        ll.insert(40)
94
95        print("Displaying the final list:")
96        ll.display() # Expected: 10 -> 20 -> 30 -> 40
97
```

# OUTPUT:

## OBSERVATION:

The code generated by the AI is used to perform the operations in the data structures like singly linked list performing insert and delete operations accurately.

## TASK-04:

Use AI to create a BST with insert and in-order traversal methods.
Sample Input Code:
class BST:
pass

## PROMPT:

Generate a python code which creates a BST with insert and in-order methods.

# CODE:

```python
from typing import Any, Optional, Generator
class Node:
    """
    An object for storing a single node of a Binary Search Tree.
    Models three attributes: the node's value, and pointers to the
    left and right children.
    """
    def __init__(self, value: Any):
        """
        Initializes a Node.
        Args:
            value (Any): The value to be stored in the node.
        """
        self.value = value
        self.left: Optional['Node'] = None
        self.right: Optional['Node'] = None
class BST:
    """
    A simple implementation of a Binary Search Tree (BST).
    """
    def __init__(self):
        """
        Initializes an empty BST.
        """
        self.root: Optional[Node] = None
    def insert(self, value: Any) -> None:
        """
        Inserts a new value into the BST, maintaining the BST property.

        Args:
            value (Any): The value to insert into the tree.
        """
        if self.root is None:
            self.root = Node(value)
        else:
            self._insert_recursive(self.root, value)
    def _insert_recursive(self, current_node: Node, value: Any) -> Node:
        """
        A private helper method to recursively find the correct position
        and insert the new node.

        Args:
            current_node (Node): The node to start the search from.
            value (Any): The value to insert.

        Returns:
            Node: The (potentially new) root of the subtree.
        """
        if value < current_node.value:
            if current_node.left is None:
                current_node.left = Node(value)
            else:
                self._insert_recursive(current_node.left, value)
        elif value > current_node.value:
            if current_node.right is None:
                current_node.right = Node(value)
            else:
                self._insert_recursive(current_node.right, value)
```

```python
     class BST:
         def _insert_recursive(self, current_node: Node, value: Any) -> Node:

             # If value == current_node.value, we do nothing (no duplicates).
             return current_node

         def inorder_traversal(self) -> None:
             """
             Performs an in-order traversal of the tree and prints the values.
             In-order traversal visits nodes in ascending order.
             """
             print("In-order Traversal:", end=" ")
             nodes = list(self._inorder_generator(self.root))
             print(" -> ".join(map(str, nodes)))

         def _inorder_generator(self, current_node: Optional[Node]) -> Generator[Any, None, None]:
             """
             A private generator for recursively performing in-order traversal.
             It yields nodes in the order: left subtree, root, right subtree.

             Args:
                 current_node (Optional[Node]): The node to start traversal from.
             """
             if current_node:
                 yield from self._inorder_generator(current_node.left)
                 yield current_node.value
                 yield from self._inorder_generator(current_node.right)


     # --- Example Usage ---
     if __name__ == "__main__":
         print("Creating a new Binary Search Tree...")
         bst = BST()

         # The order of insertion matters for the tree's shape.
         print("Inserting values: 50, 30, 70, 20, 40, 60, 80")
         values_to_insert = [50, 30, 70, 20, 40, 60, 80]
         for val in values_to_insert:
             bst.insert(val)

         # The resulting tree structure will be:
         #        50
         #       /  \
         #      30    70
         #     / \   / \
         # 20  40 60  80

         # In-order traversal should print the nodes in sorted order.
         bst.inorder_traversal() # Expected: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80

         print("\nInserting a new value: 55")
         bst.insert(55)
         bst.inorder_traversal() # Expected: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80

         print("\nAttempting to insert a duplicate value: 40")
         bst.insert(40) # This should not change the tree
         bst.inorder_traversal() # Expected: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80
```

# OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.4.py
Creating a new Binary Search Tree...
Inserting values: 50, 30, 70, 20, 40, 60, 80
In-order Traversal: 20 -> 30 -> 40 -> 50 -> 60 -> 70 -> 80

Inserting a new value: 55
In-order Traversal: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80

Attempting to insert a duplicate value: 40
In-order Traversal: 20 -> 30 -> 40 -> 50 -> 55 -> 60 -> 70 -> 80
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

## OBSERVATION:

The BST in data structures with insertion and in-order traversal methods are generated in python which are more helpful and in an efficient way to understand.

## TASK-05:

Use AI to implement a hash table with basic insert, search, and delete

methods.
Sample Input Code:
class HashTable:
pass

## PROMPT:

Generate a python code which helps to implement the hashtable with basic insert and delete methods.

## CODE:

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

class HashTable:
    """
    A simple implementation of a Hash Table using chaining for collision resolution.
    """
    def __init__(self, size: int = 101):
        """
        Initializes the hash table.

        Args:
            size (int): The number of buckets in the hash table. A prime number
                        is often recommended to help distribute keys more uniformly.
        """
        if size < 1:
            raise ValueError("Hash table size must be a positive integer.")
        self.size = size
        # Initialize the table with empty lists (chains) for each bucket
        self.table: List[List[Tuple[Any, Any]]] = [[] for _ in range(self.size)]

    def _hash(self, key: Any) -> int:
        """
        A private method to compute the hash of a key and map it to a bucket index.
        This uses Python's built-in hash() and the modulo operator.

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

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

    def insert(self, key: Any, value: Any) -> None:
        """
        Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated.

        Args:
            key (Any): The key to insert.
            value (Any): The value associated with the key.
        """
        index = self._hash(key)
        bucket = self.table[index]

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

        # Key not found, append the new key-value pair to the chain
        bucket.append((key, value))

    def search(self, key: Any) -> Optional[Any]:
        """
        Searches for a key in the hash table and returns its value.
```

```python
    class HashTable:
    def search(self, key: Any) -> Optional[Any]:

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

            Returns:
                Optional[Any]: The value associated with the key if found, otherwise None.
            """
            index = self._hash(key)
            bucket = self.table[index]

            # Traverse the chain in the bucket
            for existing_key, value in bucket:
                if existing_key == key:
                    return value  # Key found, return its value

            return None  # Key not found

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

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

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

        # Find the key in the chain and remove it
        for i, (existing_key, _) in enumerate(bucket):
            if existing_key == key:
                bucket.pop(i)
                return True  # Deletion successful

        return False  # Key was not found

    def __str__(self) -> str:
        """
        Returns a string representation of the hash table's contents.
        """
        elements = []
        for i, bucket in enumerate(self.table):
            if bucket:
                elements.append(f"Bucket {i}: {bucket}")
        return "\n".join(elements) if elements else "HashTable is empty."

# --- Example Usage ---
if __name__ == "__main__":
    print("Creating a new hash table of size 10...")
    ht = HashTable(10)

    print("\n--- Inserting Elements ---")
    ht.insert("name", "Alice")
    ht.insert("age", 30)
```

```
      111
      112      print("\n--- Inserting Elements ---")
      113      ht.insert("name", "Alice")
      114      ht.insert("age", 30)
      115      ht.insert("city", "New York")
      116      ht.insert(99, "is a number") # Keys can be different types
      117      ht.insert("name", "Bob") # This will update the value for the key "name"
      118
      119      print("Current Hash Table state:")
      120      print(ht)
      121
      122      print("\n--- Searching for Elements ---")
      123      print(f"Search for 'name': {ht.search('name')}")          # Expected: Bob
      124      print(f"Search for 'age': {ht.search('age')}")            # Expected: 30
      125      print(f"Search for 'country': {ht.search('country')}")    # Expected: None
      126
      127      print("\n--- Deleting Elements ---")
      128      print(f"Deleting 'age'... Success: {ht.delete('age')}")
      129      print(f"Deleting 'country'... Success: {ht.delete('country')}")
      130
      131      print("\nFinal Hash Table state:")
      132      print(ht)
      133
      134      print(f"\nSearching for 'age' again: {ht.search('age')}") # Expected: None
      135
```

# OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.5.py
Creating a new hash table of size 10...

--- Inserting Elements ---
Current Hash Table state:
Bucket 0: [('name', 'Bob')]
Bucket 1: [('age', 30), ('city', 'New York')]
Bucket 9: [(99, 'is a number')]

--- Searching for Elements ---
Search for 'name': Bob
Search for 'age': 30
Search for 'country': None

--- Deleting Elements ---
Deleting 'age'... Success: True
Deleting 'country'... Success: False

Final Hash Table state:
Bucket 0: [('name', 'Bob')]
Bucket 1: [('city', 'New York')]
Bucket 9: [(99, 'is a number')]

Searching for 'age' again: None
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

# OBSERVATION:

AI generated the code of data structures topic hashtable with basic insert, search and delete which aids us to understand the hashtable using the programming language like python.

# TASK-06:

Use AI to implement a graph using an adjacency list.

Sample Input Code:

class Graph:

pass

# PROMPT:

Generate a python code which implements the graph using an adjacency list in data structures.

# CODE:

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

class Graph:
    """
    A simple implementation of an undirected graph using an adjacency list.
    """
    def __init__(self):
        """
        Initializes an empty graph.
        The adjacency list is represented as a dictionary where keys are
        vertices and values are lists of their adjacent vertices.
        """
        self.adj_list: Dict[Any, List[Any]] = {}

    def add_vertex(self, vertex: Any) -> bool:
        """
        Adds a vertex to the graph.

        Args:
            vertex (Any): The vertex to be added. Must be a hashable type.

        Returns:
            bool: True if the vertex was added, False if it already exists.
        """
        if vertex not in self.adj_list:
            self.adj_list[vertex] = []
            return True
        return False

    def add_edge(self, v1: Any, v2: Any) -> bool:
        """
        Adds an undirected edge between two vertices.
        If the vertices do not exist, they are created first.

        Args:
            v1 (Any): The first vertex.
            v2 (Any): The second vertex.

        Returns:
            bool: True if the edge was added, False if it already existed.
        """
        # Ensure both vertices exist in the graph
        self.add_vertex(v1)
        self.add_vertex(v2)

        # Add the edge for an undirected graph
        if v2 not in self.adj_list[v1]:
            self.adj_list[v1].append(v2)
            self.adj_list[v2].append(v1)
            return True
        return False

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

```python
   3    class Graph:
  53        def display(self) -> None:
  58                print("Graph is empty.")
  59                return
  60            for vertex in self.adj_list:
  61                print(f"{vertex}: {self.adj_list[vertex]}")
  62
  63        def get_vertices(self) -> List[Any]:
  64            """
  65            Returns a list of all vertices in the graph.
  66            """
  67            return list(self.adj_list.keys())
  68
  69        def get_edges(self) -> List[tuple[Any, Any]]:
  70            """
  71            Returns a list of all edges in the graph.
  72            """
  73            edges = set()
  74            for vertex, neighbors in self.adj_list.items():
  75                for neighbor in neighbors:
  76                    # To avoid duplicate edges like (A, B) and (B, A)
  77                    if vertex < neighbor:
  78                        edges.add((vertex, neighbor))
  79            return list(edges)
  80
  81    # --- Example Usage ---
  82    if __name__ == "__main__":
  83        print("Creating a new graph...")
  84        g = Graph()
  85
  86        print("\n--- Adding Vertices ---")
  87        g.add_vertex("A")
  88        g.add_vertex("B")
  89        g.add_vertex("C")
  90        print("Graph after adding vertices:")
  91        g.display()
  92
  93        print("\n--- Adding Edges ---")
  94        g.add_edge("A", "B")
  95        g.add_edge("B", "C")
  96        g.add_edge("C", "A")
  97        # Adding an edge with a new vertex
  98        g.add_edge("A", "D")
  99        print("Graph after adding edges:")
 100        g.display()
 101
 102        print("\n--- Retrieving Vertices and Edges ---")
 103        print("Vertices:", g.get_vertices())
 104        print("Edges:", g.get_edges())
 105
 106        print("\n--- Adding a duplicate edge (A, B) ---")
 107        success = g.add_edge("A", "B")
 108        print(f"Was the edge added? {success}") # Expected: False
 109        print("Graph state remains the same:")
 110        g.display()
 111
```

# OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.6.py
Creating a new graph...

--- Adding Vertices ---
Graph after adding vertices:
A: []
B: []
C: []

--- Adding Edges ---
Graph after adding edges:
A: ['B', 'C', 'D']
B: ['A', 'C']
C: ['B', 'A']
D: ['A']

--- Retrieving Vertices and Edges ---
Vertices: ['A', 'B', 'C', 'D']
Edges: [('A', 'C'), ('A', 'B'), ('B', 'C'), ('A', 'D')]

--- Adding a duplicate edge (A, B) ---
Was the edge added? False
Graph state remains the same:
A: ['B', 'C', 'D']
B: ['A', 'C']
C: ['B', 'A']
D: ['A']
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

## OBSERVATION:

Implementation of graph with the adjacency list is generated by the AI which makes us know about the graph in an efficient way and easy to understand.

## TASK-07:

Use AI to implement a priority queue using Python's heapq module.
Sample Input Code:
class PriorityQueue:
pass

## PROMPT:

Write a python code which implement the priority queue using the heapq module.

## CODE:

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

class PriorityQueue:
    """
    A simple implementation of a Priority Queue using Python's heapq module.
    Lower numbers indicate higher priority.
    """
    def __init__(self):
        """
        Initializes an empty priority queue.
        """
        self._heap: List[Tuple[int, Any]] = []

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

        Args:
            item (Any): The item to be stored in the queue.
            priority (int): The priority of the item. Lower numbers are higher priority.
        """
        # heapq is a min-heap, so it naturally keeps the item with the
        # smallest priority value at the front.
        heapq.heappush(self._heap, (priority, item))

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

        Raises:
            IndexError: If the priority queue is empty.

        Returns:
            Any: The item with the highest priority.
        """
        if self.is_empty():
            raise IndexError("pop from an empty priority queue")
        # heappop removes and returns the smallest item from the heap.
        # We return only the item (index 1), not its priority (index 0).
        return heapq.heappop(self._heap)[1]

    def peek(self) -> Any:
        """
        Returns the item with the highest priority without removing it.

        Raises:
            IndexError: If the priority queue is empty.

        Returns:
            Any: The item with the highest priority.
        """
        if self.is_empty():
            raise IndexError("peek from an empty priority queue")
        # The highest priority item is always at the root of the heap (index 0).
        return self._heap[0][1]

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

```python
class PriorityQueue:
    def is_empty(self) -> bool:

        Checks if the priority queue is empty.

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

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

    def __str__(self) -> str:
        """
        Returns a string representation of the priority queue's heap.
        Note: The list representation is not necessarily sorted.
        """
        return str(self._heap)

# --- Example Usage ---
if __name__ == "__main__":
    pq = PriorityQueue()
    print(f"Is the queue empty? {pq.is_empty()}") # Expected: True

    print("\nPushing tasks with priorities...")
    # (item, priority)
    pq.push("Task: Write report", 3)
    pq.push("Task: Answer urgent email", 1) # Highest priority
    pq.push("Task: Go to meeting", 2)
    pq.push("Task: Get coffee", 4) # Lowest priority

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

    print(f"\nHighest priority task (peek): {pq.peek()}") # Expected: Task: Answer urgent email

    print("\nProcessing tasks in order of priority:")
    while not pq.is_empty():
        task = pq.pop()
        print(f"  - Popped: {task}")

    print(f"\nIs the queue empty now? {pq.is_empty()}") # Expected: True
```

# OUTPUT:

PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS

PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.7.py
Is the queue empty? True

Pushing tasks with priorities...
Current queue size: 4
Internal heap representation: [(1, 'Task: Answer urgent email'), (3, 'Task: Write report'), (2, 'Task: Go to meeting'), (4, 'Task: Get coffee')]
Is the queue empty? False

Highest priority task (peek): Task: Answer urgent email

Processing tasks in order of priority:
   - Popped: Task: Answer urgent email
   - Popped: Task: Go to meeting
   - Popped: Task: Write report
   - Popped: Task: Get coffee

Is the queue empty now? True
PS C:\Users\ramch\OneDrive\Desktop\ai> []

# OBSERVATION:

The priority queue is generated using the heaqp module of python which implements them accurately.

# TASK-08:

Use AI to implement a double-ended queue using collections.deque.
Sample Input Code:
class DequeDS:
pass

# PROMPT:

Write a python code to implement the double-ended queue using collections .deque

# CODE:

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

class DequeDS:
    """
    A wrapper class for Python's collections.deque to provide a clear
    and explicit double-ended queue interface.
    """
    def __init__(self):
        """
        Initializes an empty deque.
        """
        self._deque = deque()

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

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

    def add_last(self, item: Any) -> None:
        """
        Adds an item to the back of the deque.

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

    def remove_first(self) -> Any:
        """
        Removes and returns the item from the front of the deque.

        Raises:
            IndexError: If the deque is empty.

        Returns:
            Any: The item from the front.
        """
        if self.is_empty():
            raise IndexError("remove_first from an empty deque")
        return self._deque.popleft()

    def remove_last(self) -> Any:
        """
        Removes and returns the item from the back of the deque.

        Raises:
            IndexError: If the deque is empty.

        Returns:
            Any: The item from the back.
        """
        if self.is_empty():
            raise IndexError("remove_last from an empty deque")
```

```python
class DequeDS:
        return self._deque.pop()

    def peek_first(self) -> Any:
        """
        Returns the item at the front of the deque without removing it.

        Raises:
            IndexError: If the deque is empty.

        Returns:
            Any: The item at the front.
        """
        if self.is_empty():
            raise IndexError("peek_first from an empty deque")
        return self._deque[0]

    def peek_last(self) -> Any:
        """
        Returns the item at the back of the deque without removing it.

        Raises:
            IndexError: If the deque is empty.

        Returns:
            Any: The item at the back.
        """
        if self.is_empty():
            raise IndexError("peek_last from an empty deque")
        return self._deque[-1]

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

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

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

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

# --- Example Usage ---
if __name__ == "__main__":
    d = DequeDS()
    print(f"Is deque empty? {d.is_empty()}") # Expected: True

    print("\nAdding 'B' and 'C' to the back...")
```

```
112    u = DequeDS()
113    print(f"Is deque empty? {d.is_empty()}") # Expected: True
114
115    print("\nAdding 'B' and 'C' to the back...")
116    d.add_last("B")
117    d.add_last("C")
118    print(d) # Expected: DequeDS(['B', 'C'])
119
120    print("\nAdding 'A' to the front...")
121    d.add_first("A")
122    print(d) # Expected: DequeDS(['A', 'B', 'C'])
123
124    print(f"\nSize of deque: {len(d)}") # Expected: 3
125    print(f"Peek first: {d.peek_first()}") # Expected: A
126    print(f"Peek last: {d.peek_last()}")   # Expected: C
127
128    print("\nRemoving from the back...")
129    item = d.remove_last()
130    print(f"Removed '{item}'. Current deque: {d}") # Expected: DequeDS(['A', 'B'])
131
132    print("\nRemoving from the front...")
133    item = d.remove_first()
134    print(f"Removed '{item}'. Current deque: {d}") # Expected: DequeDS(['B'])
135
136    print("\nClearing the deque...")
137    d.remove_first()
138    print(f"Is deque empty? {d.is_empty()}") # Expected: True
139
```

# OUTPUT:

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   PORTS

PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.8.py
Is deque empty? True

Adding 'B' and 'C' to the back...
DequeDS(['B', 'C'])

Adding 'A' to the front...
DequeDS(['A', 'B', 'C'])

Size of deque: 3
Peek first: A
Peek last: C

Removing from the back...
Removed 'C'. Current deque: DequeDS(['A', 'B'])

Removing from the front...
Removed 'A'. Current deque: DequeDS(['B'])

Clearing the deque...
Is deque empty? True
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

# OBSERVATION:

The code generated by the AI of data structures topic of double-ended queue with the help of python to easily understand the topic accurately.

# TASK-09:

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

# PROMPT:

Generate a comparative table of different data structures including time complexities .

# TABLE:

**Data Structure Time Complexity Comparison**

| Data Structure | Access (by index/key) | Search (by value) | Insertion (at end) | Insertion (at start) | Deletion (at end) | Deletion (at start) | Notes |
|---|---|---|---|---|---|---|---|
| Python List (Array) | O(1) | O(n) | O(1) | O(n) | O(1) | O(n) | Insertion/deletion at the start is slow because all other elements must be shifted. |
| Stack (using List) | O(n) | O(n) | O(1) | N/A | O(1) | N/A | Operations are only on the "top" (end of the list). Accessing arbitrary elements is not a standard stack operation. |
| Queue (using List) | O(n) | O(n) | O(1) | N/A | N/A | O(n) | dequeue (from the start) is inefficient. collections.deque is highly preferred. |
| Queue (using deque) | O(n) | O(n) | O(1) | N/A | N/A | O(1) | collections.deque is optimized for fast appends and pops from both ends. |
| Singly Linked List | O(n) | O(n) | O(n) | O(1) | O(n) | O(1) | Insertion/deletion at the start is fast. Finding the end requires traversing the whole list. |
| Doubly Linked List | O(n) | O(n) | O(1) | O(1) | O(1) | O(1) | Faster end-of-list operations than a singly linked list because it can be tracked. Uses more memory per node. |
| Hash Table (Dict) | O(1) | O(1) | O(1) | O(1) | O(1) | O(1) | Average case is O(1). Worst case (due to hash collisions) can be O(n). |
| Binary Search Tree | O(log n) | O(log n) | O(log n) | O(log n) | O(log n) | O(log n) | Assumes the tree is balanced. In the worst case (an unbalanced tree), it degrades to O(n). |

# TASK-10:

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.

# PROMPT:

Generate the python code according to the given above details.

# CODE:

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

class EventRegistration:
    """
    Manages participant registration for an event using a hash table.
    This provides fast lookups, additions, and removals of participants.
    """
    def __init__(self, event_name: str):
        """
        Initializes the registration system for a specific event.

        Args:
            event_name (str): The name of the event.
        """
        self.event_name = event_name
        # The hash table (dictionary) to store participants.
        # Key: student_id, Value: student_name
        self._participants: Dict[int, str] = {}
        print(f"Event Registration System for '{self.event_name}' is now open!")

    def register_participant(self, student_id: int, student_name: str) -> bool:
        """
        Registers a new participant for the event.

        Args:
            student_id (int): The unique ID of the student.
            student_name (str): The name of the student.

        Returns:
            bool: True if registration was successful, False if the student
                  is already registered.
        """
        if student_id in self._participants:
            print(f"-> Error: Student {student_name} (ID: {student_id}) is already registered.")
            return False

        self._participants[student_id] = student_name
        print(f"-> Success: {student_name} (ID: {student_id}) has been registered.")
        return True

    def remove_participant(self, student_id: int) -> bool:
        """
        Removes a participant from the event.

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

        Returns:
            bool: True if the student was found and removed, False otherwise.
        """
        if student_id in self._participants:
            student_name = self._participants.pop(student_id)
            print(f"-> Success: {student_name} (ID: {student_id}) has been removed.")
            return True

        print(f"-> Error: Student with ID {student_id} not found.")
        return False
```

```python
class EventRegistration:
        return False

    def find_participant(self, student_id: int) -> Optional[str]:
        """
        Searches for a participant by their student ID.

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

        Returns:
            Optional[str]: The name of the student if found, otherwise None.
        """
        return self._participants.get(student_id)

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


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

    print("\n--- Registering Participants ---")
    codefest_event.register_participant(101, "Alice")
    codefest_event.register_participant(102, "Bob")
    codefest_event.register_participant(103, "Charlie")

    # Attempt to register a duplicate
    codefest_event.register_participant(101, "Alice")

    codefest_event.display_participants()

    print("\n--- Searching for a Participant ---")
    student_id_to_find = 102
    student_name = codefest_event.find_participant(student_id_to_find)
    if student_name:
        print(f"Found participant with ID {student_id_to_find}: {student_name}")
    else:
        print(f"Participant with ID {student_id_to_find} not found.")

    print("\n--- Removing a Participant ---")
    codefest_event.remove_participant(103) # Successful removal
    codefest_event.remove_participant(999) # Failed removal (not found)

    codefest_event.display_participants()
```

# OUTPUT:

```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/11.2.10.py
Event Registration System for 'CodeFest 2024' is now open!

--- Registering Participants ---
-> Success: Alice (ID: 101) has been registered.
-> Success: Bob (ID: 102) has been registered.
-> Success: Charlie (ID: 103) has been registered.
-> Error: Student Alice (ID: 101) is already registered.

--- Registered Participants for 'CodeFest 2024' ---
  - ID: 101, Name: Alice
  - ID: 102, Name: Bob
  - ID: 103, Name: Charlie
-------------------------------------------------

--- Searching for a Participant ---
Found participant with ID 102: Bob

--- Removing a Participant ---
-> Success: Charlie (ID: 103) has been removed.
-> Error: Student with ID 999 not found.

--- Registered Participants for 'CodeFest 2024' ---
  - ID: 101, Name: Alice
  - ID: 102, Name: Bob
-------------------------------------------------
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

# TABLE:

## Data Structure Selection and Justification

Here is a table that maps each system feature to the most suitable data structure and provides a justification for the choice.

| Feature | Chosen Data Structure | Justification |
|---|---|---|
| 1. Student Attendance Tracking | Stack | A stack is ideal for tracking entries and exits. When a student enters, their ID is pushed onto the stack; when they exit, their ID is popped. This Last-In, First-Out (LIFO) model naturally represents the most recent students on campus and can quickly show who is currently inside. |
| 2. Event Registration System | Hash Table | A hash table (like a Python dictionary) is perfect for managing event participants. It allows for extremely fast $O(1)$ average-time lookups, insertions, and removals using a unique student ID as the key. This efficiency is crucial for quickly checking if a student is registered or for managing a large list of attendees. |
| 3. Library Book Borrowing | Binary Search Tree (BST) | A BST, keyed on book titles or ISBNs, is a great choice for managing available books. It keeps the books in a sorted order, allowing for efficient $O(\log n)$ searching. This is much faster than a linear scan when the library has thousands of books. |
| 4. Bus Scheduling System | Graph | A graph is the most natural way to model a bus network. Each bus stop can be represented as a vertex, and the routes between stops can be represented as edges. This structure allows for solving complex problems like finding the shortest path between two stops or identifying all possible routes. |
| 5. Cafeteria Order Queue | Queue | A queue is the perfect data structure for this task as it follows the First-In, First-Out (FIFO) principle. Students are served in the exact order they arrive, just like a real-world line. This ensures fairness and is the most intuitive way to manage an order system. |

## OBSERVATION:

The AI generated the code in an efficient way according to the details given in which it include all the data structures concepts to make all easily understand . As the task contains much more information it should be handled in an efficient way.