AI ASSISTED CODING

Name: Padmavathi

HT No:2303A52269

BT No:36

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.

[1]
✓ 0s

```python
class Stack:
    """
    A simple Stack implementation using a Python list.
    LIFO (Last-In, First-Out) data structure.
    """

    def __init__(self):
        """
        Initializes an empty Stack.
        """
        self._items = []

    def push(self, item):
        """
        Adds an item to the top of the stack.

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

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

        Returns:
            The item that was at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("pop from empty stack")
        return self._items.pop()

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

        Returns:
            The item at the top of the stack.

        Raises:
            IndexError: If the stack is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty stack")
        return self._items[-1]

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

        Returns:
            True if the stack is empty, False otherwise.
        """
        return len(self._items) == 0

    def size(self):
        """
        Returns the number of items in the stack.
        """
        return len(self._items)
```

[1]
✓ 0s

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

    def __repr__(self):
        """
        Returns a string representation of the stack for debugging.
        """
        return f"Stack({self._items})"
```

You can test the Stack implementation with the following code:

[2]
✓ 0s

```
# Create a new stack
my_stack = Stack()

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

# Test push
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
print(f"Stack after pushes: {my_stack}")
print(f"Stack size: {my_stack.size()}")

# Test is_empty again
print(f"Is stack empty? {my_stack.is_empty()}")
```

```
[2]
✓ 0s
```

```python
# Test is_empty again
print(f"Is stack empty? {my_stack.is_empty()}")

# Test peek
print(f"Top of stack (peek): {my_stack.peek()}")
print(f"Stack after peek: {my_stack}") # Should remain unchanged

# Test pop
popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")
print(f"Stack after pop: {my_stack}")
print(f"Stack size: {my_stack.size()}")

popped_item = my_stack.pop()
print(f"Popped item: {popped_item}")
print(f"Stack after pop: {my_stack}")

# Pop remaining item
my_stack.pop()
print(f"Stack after all pops: {my_stack}")
print(f"Is stack empty? {my_stack.is_empty()}")

# Test pop/peek on empty stack (this will raise an error)
try:
    my_stack.pop()
except IndexError as e:
    print(f"Error popping from empty stack: {e}")

try:
    my_stack.peek()
except IndexError as e:
    print(f"Error peeking from empty stack: {e}")
```

OUTPUT:

```
Is stack empty? True
Stack after pushes: [10, 20, 30]
Stack size: 3
Is stack empty? False
Top of stack (peek): 30
Stack after peek: [10, 20, 30]
Popped item: 30
Stack after pop: [10, 20]
Stack size: 2
Popped item: 20
Stack after pop: [10]
Stack after all pops: []
Is stack empty? True
Error popping from empty stack: pop from empty stack
Error peeking from empty stack: peek from empty stack
```

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.

[13]
✓ 0s

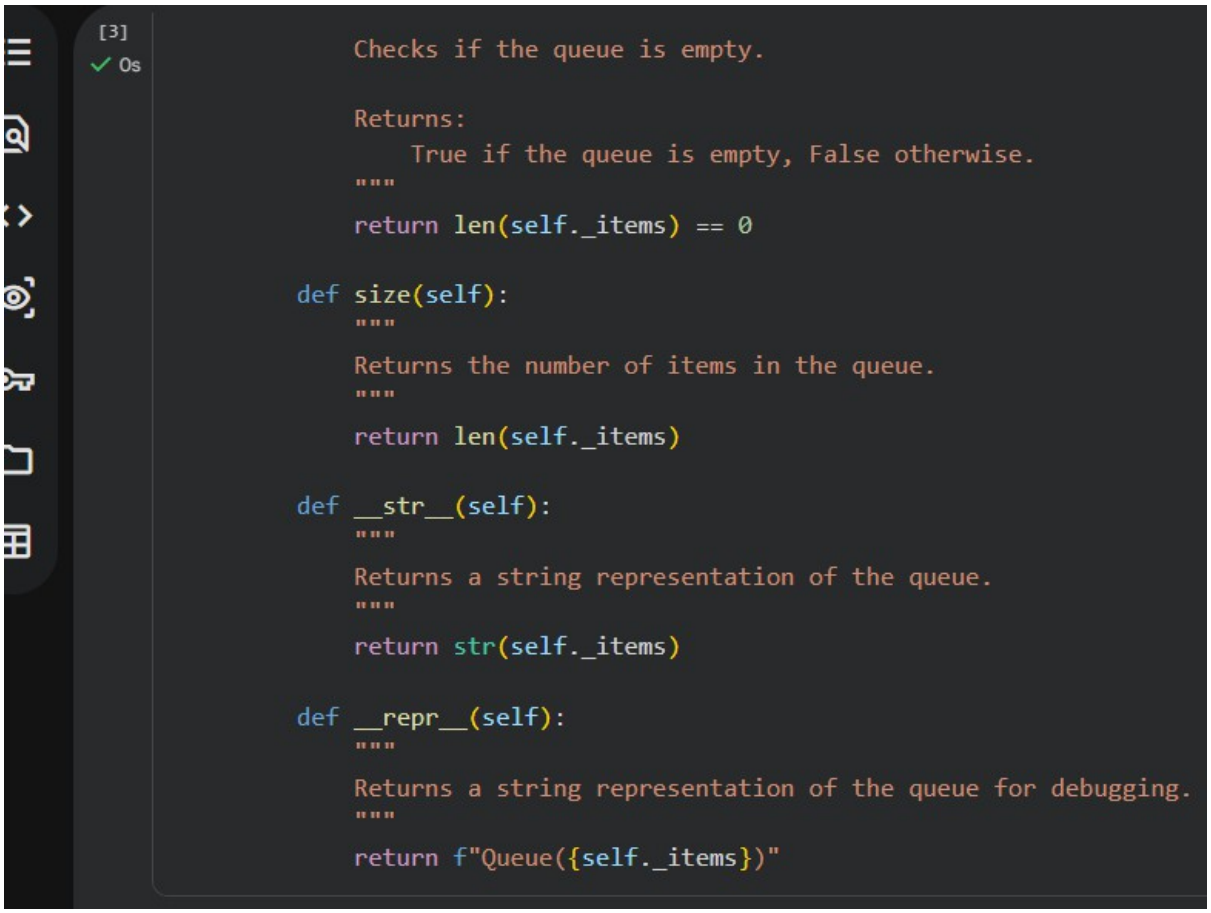
```
        Inserts a key-value pair into the hash table.
        If the key already exists, its value is updated.
        """
        index = self._hash_function(key)
        self.table[index].insert_or_update(key, value)

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

        Returns:
            The value associated with the key if found, otherwise None.
        """
        index = self._hash_function(key)
        return self.table[index].search(key)

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

        Returns:
            True if the key was found and deleted, False otherwise.
        """
        index = self._hash_function(key)
        return self.table[index].delete(key)

    def display(self):
        """
        Prints the content of the hash table.
        """
        print("\n--- Hash Table Contents ---")
        for i, bucket in enumerate(self.table):
```

[3]
✓ 0s

```python
class Queue:
    """
    A simple Queue implementation using a Python list.
    FIFO (First-In, First-Out) data structure.
    """

    def __init__(self):
        """
        Initializes an empty Queue.
        """
        self._items = []

    def enqueue(self, item):
        """
        Adds an item to the rear of the queue.

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

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

        Returns:
            The item that was at the front of the queue.

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

[3]
✓ 0s

```
        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)

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

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self._items[0]

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

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

    def size(self):
        """
```


[3]
0s
Checks if the queue is empty.
Returns:
True if the queue is empty, False otherwise.
"""
return len(self._items) == 0
def size(self):
"""
Returns the number of items in the queue.
"""
return len(self._items)
def __str__(self):
"""
Returns a string representation of the queue.
"""
return str(self._items)
def __repr__(self):
"""
Returns a string representation of the queue for debugging.
"""
return f"Queue({self._items})"

[4]
✓ 0s

```
# Create a new queue
my_queue = Queue()

# Test is_empty
print(f"Is queue empty? {my_queue.is_empty()}")

# Test enqueue
my_queue.enqueue('apple')
my_queue.enqueue('banana')
my_queue.enqueue('cherry')
print(f"Queue after enqueues: {my_queue}")
print(f"Queue size: {my_queue.size()}")

# Test is_empty again
print(f"Is queue empty? {my_queue.is_empty()}")

# Test peek
print(f"Front of queue (peek): {my_queue.peek()}")
print(f"Queue after peek: {my_queue}") # Should remain unchanged

# Test dequeue
dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeue: {my_queue}")
print(f"Queue size: {my_queue.size()}")

dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeue: {my_queue}")
```

```
[4]
✓ 0s
print(f"Is queue empty? {my_queue.is_empty()}")

# Test peek
print(f"Front of queue (peek): {my_queue.peek()}")
print(f"Queue after peek: {my_queue}") # Should remain unchanged

# Test dequeue
dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeue: {my_queue}")
print(f"Queue size: {my_queue.size()}")

dequeued_item = my_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeue: {my_queue}")

# Dequeue remaining item
my_queue.dequeue()
print(f"Queue after all dequeues: {my_queue}")
print(f"Is queue empty? {my_queue.is_empty()}")

# Test dequeue/peek on empty queue (this will raise an error)
try:
    my_queue.dequeue()
except IndexError as e:
    print(f"Error dequeuing from empty queue: {e}")

try:
    my_queue.peek()
except IndexError as e:
    print(f"Error peeking from empty queue: {e}")
```

```
...  Is queue empty? True
     Queue after enqueues: ['apple', 'banana', 'cherry']
     Queue size: 3
     Is queue empty? False
     Front of queue (peek): apple
     Queue after peek: ['apple', 'banana', 'cherry']
     Dequeued item: apple
     Queue after dequeue: ['banana', 'cherry']
     Queue size: 2
     Dequeued item: banana
     Queue after dequeue: ['cherry']
     Queue after all dequeues: []
     Is queue empty? True
     Error dequeuing from empty queue: dequeue from empty queue
     Error peeking from empty queue: peek from empty queue
```

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.

[7]
✓ 0s

```python
class Queue:
    """
    A simple Queue implementation using a Python list.
    FIFO (First-In, First-Out) data structure.
    """

    def __init__(self):
        """
        Initializes an empty Queue.
        """
        self._items = []

    def enqueue(self, item):
        """
        Adds an item to the rear of the queue.

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

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

        Returns:
            The item that was at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
```

[7] ✓ 0s

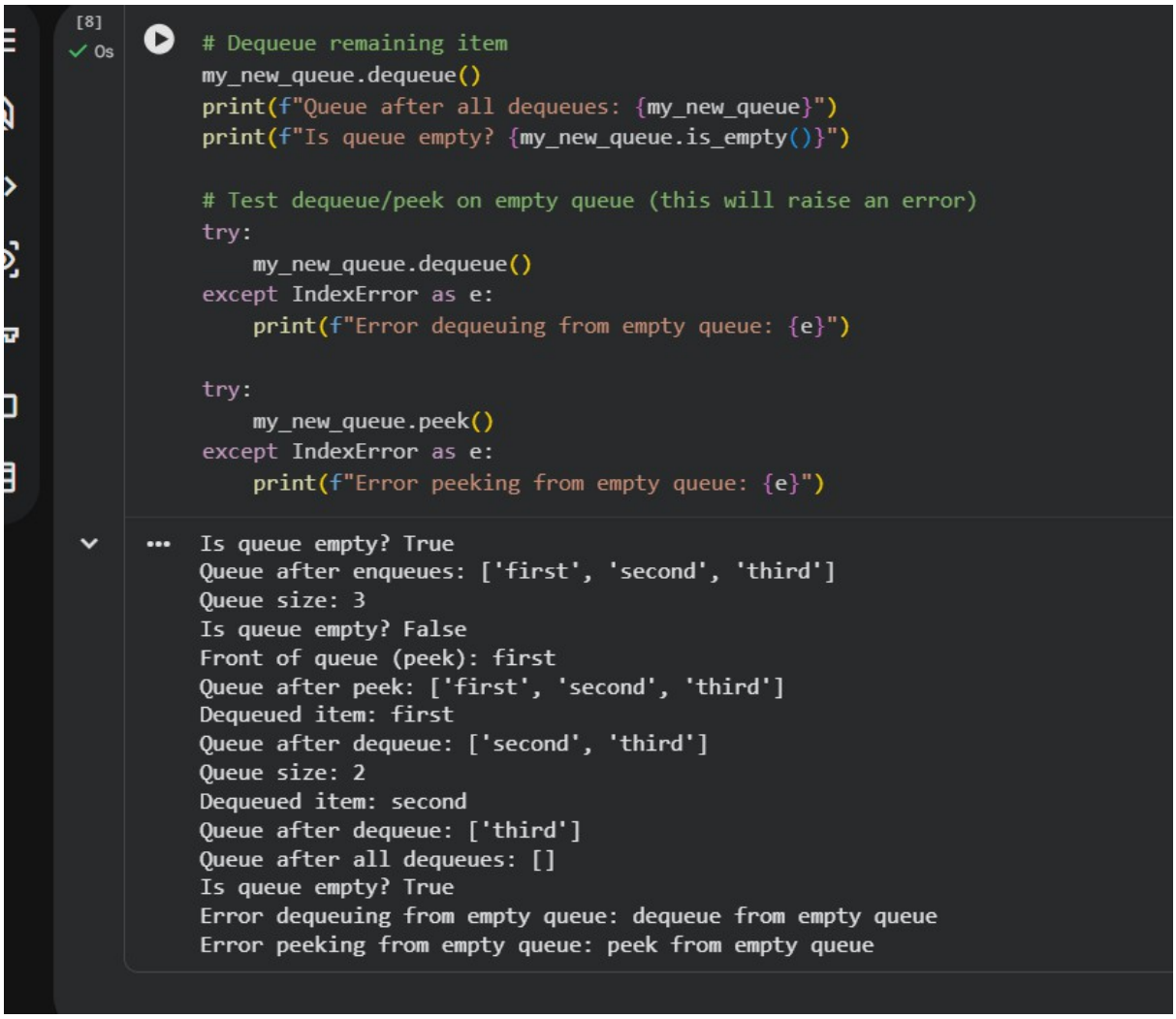
```
        if self.is_empty():
            raise IndexError("dequeue from empty queue")
        return self._items.pop(0)

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

        Returns:
            The item at the front of the queue.

        Raises:
            IndexError: If the queue is empty.
        """
        if self.is_empty():
            raise IndexError("peek from empty queue")
        return self._items[0]

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

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

    def size(self):
        """
        Returns the number of items in the queue.
        """
        return len(self._items)
```

[7] ✓ 0s

```
        return len(self._items) == 0

    def size(self):
        """
        Returns the number of items in the queue.
        """
        return len(self._items)

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

    def __repr__(self):
        """
        Returns a string representation of the queue for debugging.
        """
        return f"Queue({self._items})"
```

[8]
✓ 0s

```
# Create a new queue instance
my_new_queue = Queue()

# Test is_empty
print(f"Is queue empty? {my_new_queue.is_empty()}")

# Test enqueue
my_new_queue.enqueue('first')
my_new_queue.enqueue('second')
my_new_queue.enqueue('third')
print(f"Queue after enqueues: {my_new_queue}")
print(f"Queue size: {my_new_queue.size()}")

# Test is_empty again
print(f"Is queue empty? {my_new_queue.is_empty()}")

# Test peek
print(f"Front of queue (peek): {my_new_queue.peek()}")
print(f"Queue after peek: {my_new_queue}") # Should remain unchanged

# Test dequeue
dequeued_item = my_new_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeue: {my_new_queue}")
print(f"Queue size: {my_new_queue.size()}")

dequeued_item = my_new_queue.dequeue()
print(f"Dequeued item: {dequeued_item}")
print(f"Queue after dequeue: {my_new_queue}")

# Dequeue remaining item
my new queue.dequeue()
```



```
[8]
✓ 0s
# Dequeue remaining item
my_new_queue.dequeue()
print(f"Queue after all dequeues: {my_new_queue}")
print(f"Is queue empty? {my_new_queue.is_empty()}")

# Test dequeue/peek on empty queue (this will raise an error)
try:
    my_new_queue.dequeue()
except IndexError as e:
    print(f"Error dequeuing from empty queue: {e}")

try:
    my_new_queue.peek()
except IndexError as e:
    print(f"Error peeking from empty queue: {e}")

Is queue empty? True
Queue after enqueues: ['first', 'second', 'third']
Queue size: 3
Is queue empty? False
Front of queue (peek): first
Queue after peek: ['first', 'second', 'third']
Dequeued item: first
Queue after dequeue: ['second', 'third']
Queue size: 2
Dequeued item: second
Queue after dequeue: ['third']
Queue after all dequeues: []
Is queue empty? True
Error dequeuing from empty queue: dequeue from empty queue
Error peeking from empty queue: peek from empty queue
```

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.

[9]
✓ 0s

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

    Attributes:
        data: The data stored in the node.
        next: Reference to the next node in the list.
    """
    def __init__(self, data):
        """
        Initializes a new Node with the given data.
        """
        self.data = data
        self.next = None

    def __str__(self):
        """
        Returns a string representation of the Node's data.
        """
        return str(self.data)

    def __repr__(self):
        """
        Returns a detailed string representation of the Node.
        """
        return f"Node({self.data})"

class LinkedList:
    """
    A simple implementation of a Singly Linked List.

    Attributes:
```

```
[9]
✓ 0s
            head: The head node of the linked list.
        """
        def __init__(self):
            """
            Initializes an empty LinkedList.
            """
            self.head = None

        def insert(self, data):
            """
            Inserts a new node with the given data at the beginning of the list.

            Args:
                data: The data to be stored in the new node.
            """
            new_node = Node(data)
            new_node.next = self.head
            self.head = new_node

        def display(self):
            """
            Prints the elements of the linked list from head to tail.
            """
            current = self.head
            elements = []
            while current:
                elements.append(str(current.data))
                current = current.next
            print(" -> ".join(elements) if elements else "Linked list is empty.")

        def __str__(self):
            """
            Returns a string representation of the linked list.
```

```
[9]
✓ 0s
            current = current.next
        print(" -> ".join(elements) if elements else "Linked list is empty.")

    def __str__(self):
        """
        Returns a string representation of the linked list.
        """
        current = self.head
        elements = []
        while current:
            elements.append(str(current.data))
            current = current.next
        return " -> ".join(elements) if elements else "Empty LinkedList"

    def __repr__(self):
        """
        Returns a detailed string representation of the linked list.
        """
        return f"LinkedList(head={repr(self.head) if self.head else 'None'})"
```

You can test this Singly Linked List implementation with the following code:

You can test this Singly Linked List implementation with the following code:

[10]
✓ 0s

```
# Create a new linked list
my_linked_list = LinkedList()
print("Initial linked list:")
my_linked_list.display()

# Insert elements
my_linked_list.insert(10)
my_linked_list.insert(20)
my_linked_list.insert(30)

print("\nLinked list after insertions:")
my_linked_list.display()

# Insert another element
my_linked_list.insert(5)
print("\nLinked list after another insertion:")
my_linked_list.display()

# Check empty list case
empty_list = LinkedList()
print("\nEmpty linked list:")
empty_list.display()

# Demonstrate __str__ and __repr__
print(f"\nString representation: {my_linked_list}")
print(f"Representation for debugging: {repr(my_linked_list)}")
```

```
Initial linked list:
Linked list is empty.

Linked list after insertions:
30 -> 20 -> 10

Linked list after another insertion:
5 -> 30 -> 20 -> 10

Empty linked list:
Linked list is empty.

String representation: 5 -> 30 -> 20 -> 10
Representation for debugging: LinkedList(head=Node(5))
```

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.

[11] ✓ 0s

```python
class Node:
    """
    A node in a Binary Search Tree.

    Attributes:
        key: The value stored in the node.
        left: Reference to the left child node.
        right: Reference to the right child node.
    """
    def __init__(self, key):
        """
        Initializes a new Node with the given key.
        """
        self.key = key
        self.left = None
        self.right = None

    def __str__(self):
        """
        Returns a string representation of the Node's key.
        """
        return str(self.key)

    def __repr__(self):
        """
        Returns a detailed string representation of the Node.
        """
        return f"Node({self.key})"
```

[11] ✓ 0s

```python
class BST:
    """
    A Binary Search Tree implementation with recursive insert and in-order traversal
    """
    def __init__(self):
        """
        Initializes an empty Binary Search Tree.
        """
        self.root = None

    def insert(self, key):
        """
        Inserts a new key into the BST.

        Args:
            key: The value to be inserted into the tree.
        """
        self.root = self._insert_recursive(self.root, key)

    def _insert_recursive(self, node, key):
        """
        Recursively inserts a new key into the BST.

        Args:
            node: The current node being examined.
            key: The value to be inserted.

        Returns:
            The node after insertion (or the new node if it was None).
        """
        if node is None:
            return Node(key)
```

[11]
✓ 0s

```
            if key < node.key:
                node.left = self._insert_recursive(node.left, key)
            elif key > node.key:
                node.right = self._insert_recursive(node.right, key)
            # Duplicate keys are not inserted
            return node

        def in_order_traversal(self):
            """
            Performs an in-order traversal of the BST and returns a list of keys.

            Returns:
                A list containing the keys in ascending order.
            """
            elements = []
            self._in_order_recursive(self.root, elements)
            return elements

        def _in_order_recursive(self, node, elements):
            """
            Recursively traverses the BST in-order.

            Args:
                node: The current node being visited.
                elements: A list to accumulate the keys during traversal.
            """
            if node:
                self._in_order_recursive(node.left, elements)
                elements.append(node.key)
                self._in_order_recursive(node.right, elements)
```

[11]
✓ 0s

```
            """
            if node:
                self._in_order_recursive(node.left, elements)
                elements.append(node.key)
                self._in_order_recursive(node.right, elements)

        def __str__(self):
            """
            Returns a string representation of the BST using in-order traversal.
            """
            return str(self.in_order_traversal())

        def __repr__(self):
            """
            Returns a detailed string representation of the BST.
            """
            return f"BST(root={repr(self.root) if self.root else 'None'})"
```

You can test this Binary Search Tree implementation with the following code:

[12] ✓ 0s

```python
# Create a new BST
my_bst = BST()

# Insert elements
print("Inserting elements: 50, 30, 70, 20, 40, 60, 80")
my_bst.insert(50)
my_bst.insert(30)
my_bst.insert(70)
my_bst.insert(20)
my_bst.insert(40)
my_bst.insert(60)
my_bst.insert(80)

# Perform in-order traversal
print(f"In-order traversal: {my_bst.in_order_traversal()}") # Expected: [20, 30, 40, 50, 60, 70, 80]

# Test insertion of a duplicate key (should not be added)
print("Inserting duplicate key 50...")
my_bst.insert(50)
print(f"In-order traversal after duplicate insert: {my_bst.in_order_traversal()}")

# Test with an empty BST
empty_bst = BST()
print(f"\nIn-order traversal of an empty BST: {empty_bst.in_order_traversal()}")

# Demonstrate __str__ and __repr__
print(f"\nString representation of BST: {my_bst}")
print(f"Representation for debugging of BST: {repr(my_bst)}")
```

```
Inserting elements: 50, 30, 70, 20, 40, 60, 80
In-order traversal: [20, 30, 40, 50, 60, 70, 80]
Inserting duplicate key 50...
In-order traversal after duplicate insert: [20, 30, 40, 50, 60, 70, 80]

In-order traversal of an empty BST: []

String representation of BST: [20, 30, 40, 50, 60, 70, 80]
Representation for debugging of BST: BST(root=Node(50))
```

You can test this new Queue implementation with the following code:

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.

[13]
✓ 0s

```python
class Node:
    """
    A node for the linked list used in hash table chaining.
    Stores a key-value pair.

    Attributes:
        key: The key of the item.
        value: The value associated with the key.
        next: Reference to the next Node in the chain (for collision handling).
    """
    def __init__(self, key, value):
        """
        Initializes a new Node with the given key and value.
        """
        self.key = key
        self.value = value
        self.next = None

    def __str__(self):
        """
        Returns a string representation of the Node's key-value pair.
        """
        return f"({self.key}: {self.value})"

    def __repr__(self):
        """
        Returns a detailed string representation of the Node.
        """
        return f"Node(key={repr(self.key)}, value={repr(self.value)})"
```

[13]
✓ 0s

```
class LinkedListForHashTable:
    """
    A simple singly linked list tailored for hash table chaining.
    """
    def __init__(self):
        """
        Initializes an empty linked list.
        """
        self.head = None

    def insert_or_update(self, key, value):
        """
        Inserts a new key-value pair or updates the value if the key exists.
        """
        current = self.head
        while current:
            if current.key == key:
                current.value = value # Update value if key exists
                return
            current = current.next
        # Key not found, add new node at the head
        new_node = Node(key, value)
        new_node.next = self.head
        self.head = new_node

    def search(self, key):
        """
        Searches for a key in the linked list.

        Returns:
            The value associated with the key if found, otherwise None.
        """
```

[13]
✓ 0s

```
            The value associated with the key if found, otherwise None.
        """
        current = self.head
        while current:
            if current.key == key:
                return current.value
            current = current.next
        return None

    def delete(self, key):
        """
        Deletes a node with the given key from the linked list.

        Returns:
            True if the key was found and deleted, False otherwise.
        """
        current = self.head
        prev = None
        while current:
            if current.key == key:
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next
                return True # Key found and deleted
            prev = current
            current = current.next
        return False # Key not found

    def __str__(self):
        """
        Returns a string representation of the linked list.
        """
```

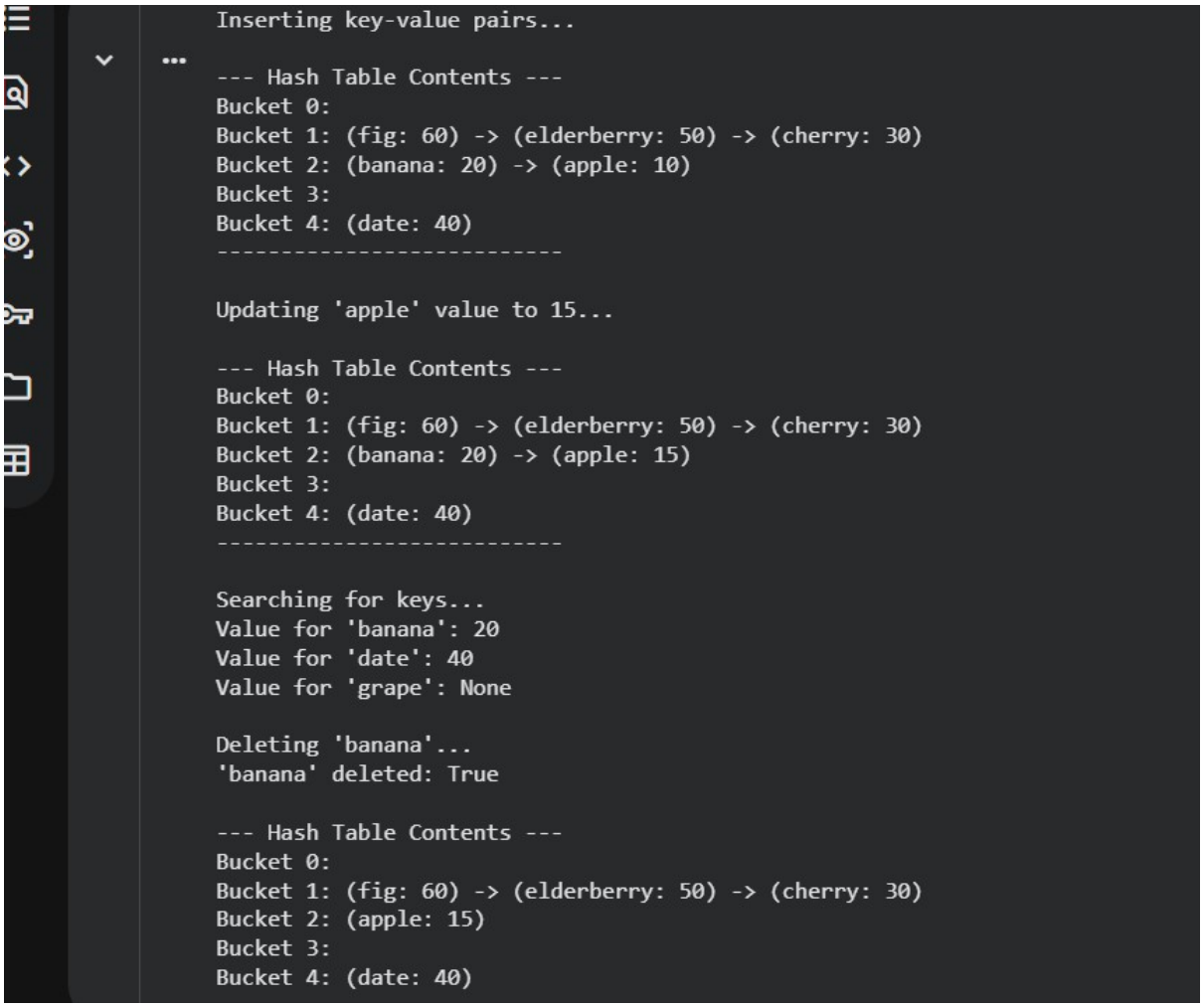
```
    def __str__(self):
        """
        Returns a string representation of the linked list.
        """
        elements = []
        current = self.head
        while current:
            elements.append(str(current))
            current = current.next
        return " -> ".join(elements)


class HashTable:
    """
    A simple Hash Table implementation using chaining for collision resolution.
    """
    def __init__(self, size=10):
        """
        Initializes the hash table with a given size.
        Each bucket is a LinkedList to handle collisions.
        """
        self.size = size
        self.table = [LinkedListForHashTable() for _ in range(self.size)]

    def _hash_function(self, key):
        """
        Simple hash function that converts the key to an integer and takes modulo size.
        """
        return hash(key) % self.size

    def insert(self, key, value):
        """
```

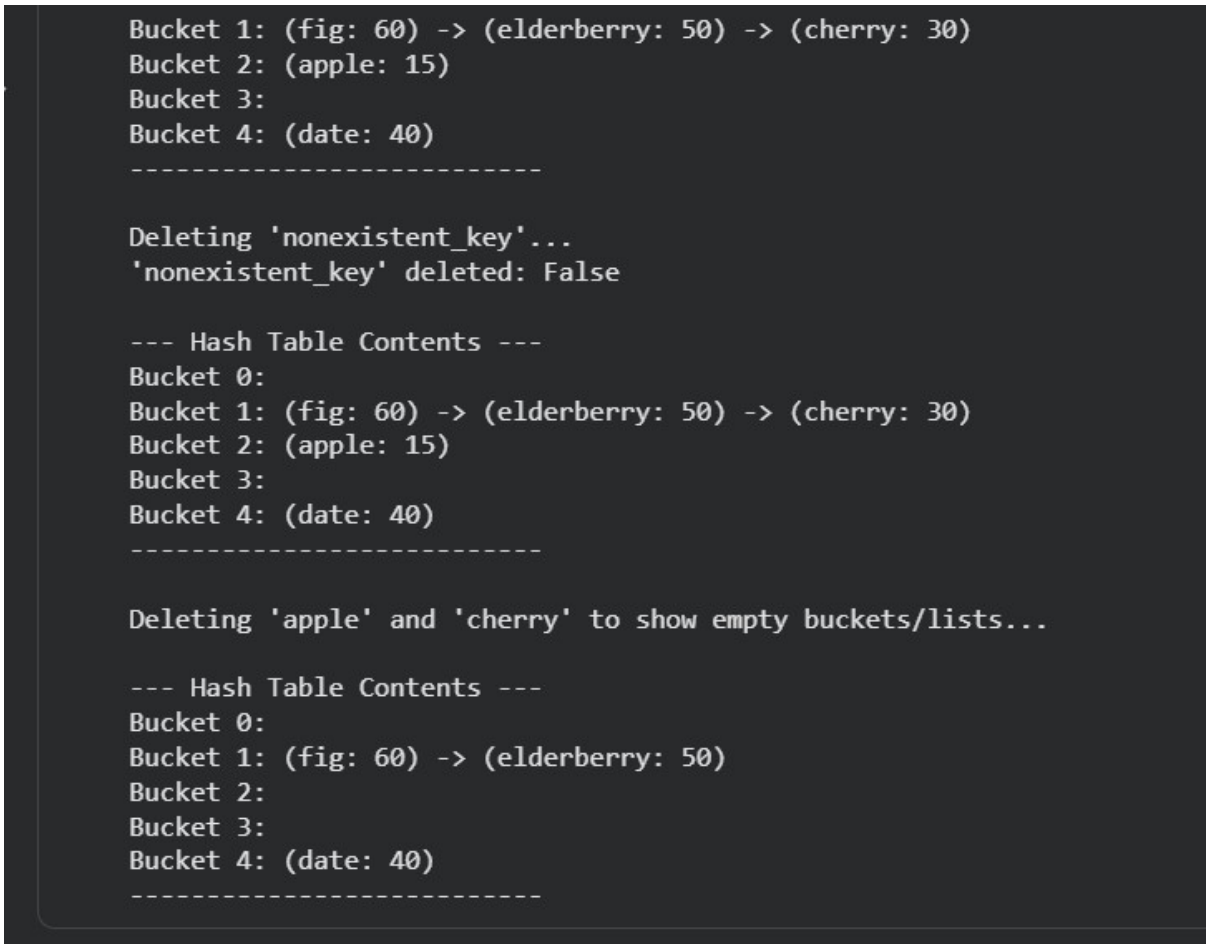
[14] ✓ 0s

```
# Create a new hash table
my_hash_table = HashTable(size=5)

# Test insert
print("Inserting key-value pairs...")
my_hash_table.insert("apple", 10)
my_hash_table.insert("banana", 20)
my_hash_table.insert("cherry", 30)
my_hash_table.insert("date", 40) # This might cause a collision depending on hash function and
my_hash_table.insert("elderberry", 50)
my_hash_table.insert("fig", 60) # Likely another collision
my_hash_table.display()

# Test update (insert with existing key)
print("\nUpdating 'apple' value to 15...")
my_hash_table.insert("apple", 15)
my_hash_table.display()

# Test search
print("\nSearching for keys...")
print(f"Value for 'banana': {my_hash_table.search('banana')}") # Expected: 20
print(f"Value for 'date': {my_hash_table.search('date')}")     # Expected: 40
print(f"Value for 'grape': {my_hash_table.search('grape')}")   # Expected: None

# Test delete
print("\nDeleting 'banana'...")
deleted = my_hash_table.delete("banana")
print(f"'banana' deleted: {deleted}") # Expected: True
my_hash_table.display()

print("\nDeleting 'nonexistent_key'...")
deleted = my_hash_table.delete("nonexistent_key")
```

[14] ✓ 0s

```
my_hash_table.insert("elderberry", 50)
my_hash_table.insert("fig", 60) # Likely another collision
my_hash_table.display()

# Test update (insert with existing key)
print("\nUpdating 'apple' value to 15...")
my_hash_table.insert("apple", 15)
my_hash_table.display()

# Test search
print("\nSearching for keys...")
print(f"Value for 'banana': {my_hash_table.search('banana')}") # Expected: 20
print(f"Value for 'date': {my_hash_table.search('date')}")     # Expected: 40
print(f"Value for 'grape': {my_hash_table.search('grape')}")   # Expected: None

# Test delete
print("\nDeleting 'banana'...")
deleted = my_hash_table.delete("banana")
print(f"'banana' deleted: {deleted}") # Expected: True
my_hash_table.display()

print("\nDeleting 'nonexistent_key'...")
deleted = my_hash_table.delete("nonexistent_key")
print(f"'nonexistent_key' deleted: {deleted}") # Expected: False
my_hash_table.display()

print("\nDeleting 'apple' and 'cherry' to show empty buckets/lists...")
my_hash_table.delete("apple")
my_hash_table.delete("cherry")
my_hash_table.display()
```


Inserting key-value pairs...
--- Hash Table Contents ---
Bucket 0:
Bucket 1: (fig: 60) -> (elderberry: 50) -> (cherry: 30)
Bucket 2: (banana: 20) -> (apple: 10)
Bucket 3:
Bucket 4: (date: 40)
Updating 'apple' value to 15...
--- Hash Table Contents ---
Bucket 0:
Bucket 1: (fig: 60) -> (elderberry: 50) -> (cherry: 30)
Bucket 2: (banana: 20) -> (apple: 15)
Bucket 3:
Bucket 4: (date: 40)
Searching for keys...
Value for 'banana': 20
Value for 'date': 40
Value for 'grape': None
Deleting 'banana'...
'banana' deleted: True
--- Hash Table Contents ---
Bucket 0:
Bucket 1: (fig: 60) -> (elderberry: 50) -> (cherry: 30)
Bucket 2: (apple: 15)
Bucket 3:
Bucket 4: (date: 40)



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.

```
[15]
✓ 0s
```

```python
class Graph:
    """
    A Graph implementation using an adjacency list.
    Supports adding vertices and edges, and displaying connections.
    """
    def __init__(self, directed=False):
        """
        Initializes an empty graph.

        Args:
            directed (bool): If True, the graph is directed. Default is False (undirected).
        """
        self.graph = {}
        self.directed = directed

    def add_vertex(self, vertex):
        """
        Adds a vertex to the graph.
        If the vertex already exists, it does nothing.

        Args:
            vertex: The vertex to be added.
        """
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, u, v):
        """
        Adds an edge between vertex u and vertex v.
        Automatically adds vertices u and v if they don't exist.
```

```
[15]
✓ 0s
```

```python
        Args:
            u: The starting vertex of the edge.
            v: The ending vertex of the edge.
        """
        self.add_vertex(u)
        self.add_vertex(v)
        self.graph[u].append(v)
        if not self.directed:
            self.graph[v].append(u) # For undirected graph, add edge in both directions

    def display(self):
        """
        Prints the adjacency list representation of the graph.
        """
        print("\n--- Graph Connections ---")
        if not self.graph:
            print("The graph is empty.")
            return

        for vertex, neighbors in self.graph.items():
            print(f"{vertex}: {', '.join(map(str, neighbors))}")
        print("-------------------------")

    def __str__(self):
        """
        Returns a string representation of the graph.
        """
        graph_str = []
        for vertex, neighbors in self.graph.items():
            graph_str.append(f"{vertex}: {', '.join(map(str, neighbors))}")
        return "\n".join(graph_str) if graph_str else "Empty Graph"
```

[15]
✓ 0s

```
    def __repr__(self):
        """
        Returns a detailed string representation of the graph for debugging.
        """
        return f"Graph(directed={self.directed}, vertices={list(self.graph.keys())})"
```

You can test this Graph implementation with the following code:

[16]
✓ 0s

```
# Create an undirected graph
print("Undirected Graph Example:")
undirected_graph = Graph()
undirected_graph.add_vertex('A')
undirected_graph.add_vertex('B')
undirected_graph.add_edge('A', 'C')
undirected_graph.add_edge('B', 'C')
undirected_graph.add_edge('C', 'D')
undirected_graph.add_edge('D', 'A')
undirected_graph.add_edge('E', 'F') # Adds two new isolated vertices
undirected_graph.display()

# Test adding an existing vertex (should not change anything)
print("\nAdding existing vertex 'A'...")
undirected_graph.add_vertex('A')
undirected_graph.display()

# Create a directed graph
print("\nDirected Graph Example:")
directed_graph = Graph(directed=True)
directed_graph.add_edge('X', 'Y')
directed_graph.add_edge('Y', 'Z')
directed_graph.add_edge('Z', 'X')
```

```
[16]
✓ 0s
print("\nDirected Graph Example:")
directed_graph = Graph(directed=True)
directed_graph.add_edge('X', 'Y')
directed_graph.add_edge('Y', 'Z')
directed_graph.add_edge('Z', 'X')
directed_graph.add_edge('A', 'B')
directed_graph.display()

# Test an empty graph
empty_graph = Graph()
print("\nEmpty Graph Example:")
empty_graph.display()

# Demonstrate __str__ and __repr__
print(f"\nString representation of undirected graph:\n{undirected_graph}")
print(f"Representation for debugging of directed graph: {repr(directed_graph)}")
```

```
Undirected Graph Example:

--- Graph Connections ---
A: C, D
B: C
C: A, B, D
D: C, A
E: F
F: E
-------------------------

Adding existing vertex 'A'...

--- Graph Connections ---
A: C, D
B: C
C: A, B, D
```

```
Adding existing vertex 'A'...

--- Graph Connections ---
A: C, D
B: C
C: A, B, D
D: C, A
E: F
F: E
------------------------

Directed Graph Example:

--- Graph Connections ---
X: Y
Y: Z
Z: X
A: B
B:
------------------------

Empty Graph Example:

--- Graph Connections ---
The graph is empty.

String representation of undirected graph:
A: C, D
B: C
C: A, B, D
D: C, A
E: F
F: E
Representation for debugging of directed graph: Graph(directed=True, vertices=['X', 'Y', 'Z', 'A', 'B'])
```

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.

[17] ✓ 0s

```python
import heapq
import itertools

class PriorityQueue:
    """
    A Priority Queue implementation using Python's heapq module.
    Items are retrieved based on their priority (lowest number = highest priority).
    Uses a counter to ensure stable ordering for items with the same priority.
    """
    def __init__(self):
        """
        Initializes an empty Priority Queue.
        The queue is a list of entries arranged in a heap.
        Each entry is a tuple: [priority, entry_count, item].
        The entry_count is used for stable ordering when priorities are equal.
        """
        self._queue = []
        self._entry_finder = {}
        self._counter = itertools.count()
        self._REMOVED = '<removed-item>'

    def enqueue(self, item, priority):
        """
        Adds an item to the priority queue or updates its priority if it already exists.

        Args:
            item: The item to be added or whose priority is to be updated.
            priority (int/float): The priority of the item. Lower numbers indicate higher priority.
        """
        if item in self._entry_finder:
            self.remove(item)
        count = next(self._counter)
        entry = [priority, count, item]
```

```python
        if item in self._entry_finder:
            self.remove(item)
        count = next(self._counter)
        entry = [priority, count, item]
        self._entry_finder[item] = entry
        heapq.heappush(self._queue, entry)

    def dequeue(self):
        """
        Removes and returns the item with the highest priority (lowest priority value).

        Returns:
            The item with the highest priority.

        Raises:
            KeyError: If the priority queue is empty.
        """
        while self._queue:
            priority, count, item = heapq.heappop(self._queue)
            if item is not self._REMOVED:
                del self._entry_finder[item]
                return item
        raise KeyError('dequeue from an empty priority queue')

    def remove(self, item):
        """
        Marks an existing item as 'removed' in the queue.
        This does not immediately remove it from the heap but effectively removes it
        during the next dequeue operation.

        Args:
            item: The item to be removed.
```

```
        Args:
            item: The item to be removed.

        Raises:
            KeyError: If the item is not found in the queue.
        """
        entry = self._entry_finder.pop(item)
        entry[-1] = self._REMOVED

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

        Returns:
            The item with the highest priority.

        Raises:
            KeyError: If the priority queue is empty.
        """
        while self._queue:
            priority, count, item = self._queue[0]
            if item is not self._REMOVED:
                return item
            heapq.heappop(self._queue) # Remove the marked item if it's at the top
        raise KeyError('peek from an empty priority queue')

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

        Returns:
            True if the queue is empty, False otherwise.
        """
```

```
        """
        Checks if the priority queue is empty.

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

    def size(self):
        """
        Returns the number of active items in the priority queue.
        """
        return len(self.display_items())

    def display_items(self):
        """
        Returns a list of active items currently in the queue, ordered by priority.
        This operation involves reconstructing a temporary sorted list, so it's not O(1).
        """
        active_items = []
        # Create a clean list of active entries for display
        for entry in self._queue:
            if entry[-1] is not self._REMOVED:
                active_items.append((entry[0], entry[-1])) # (priority, item)
        active_items.sort() # Sort by priority for consistent display
        return [item for priority, item in active_items]

    def __str__(self):
        """
        Returns a string representation of the priority queue.
        """
        return f"PriorityQueue({self.display_items()})"
```

```
        return f"PriorityQueue({self.display_items()})"

    def __len__(self):
        """
        Returns the number of items in the priority queue.
        """
        return self.size()
```

You can test this Priority Queue implementation with the following code:

```
# Create a new priority queue
pq = PriorityQueue()

# Test is_empty
print(f"Is PQ empty? {pq.is_empty()}")

# Test enqueue
print("\nEnqueuing items:")
pq.enqueue('task A', 3) # Lower priority number means higher priority
pq.enqueue('task B', 1)
pq.enqueue('task C', 2)
pq.enqueue('task D', 3) # Same priority as A, order determined by insertion

print(f"PQ after enqueues: {pq}")
print(f"PQ size: {pq.size()}")

# Test peek
print(f"\nPeek highest priority item: {pq.peek()}") # Expected: task B
print(f"PQ after peek: {pq}") # Should remain unchanged

# Test dequeue
```

```python
# Test dequeue
print("\nDequeuing items:")
print(f"Dequeued: {pq.dequeue()}") # Expected: task B
print(f"PQ after first dequeue: {pq}")

print(f"Dequeued: {pq.dequeue()}") # Expected: task C
print(f"PQ after second dequeue: {pq}")

# Test updating priority
pq.enqueue('task A', 1) # Update priority of task A to highest
print(f"\nPQ after updating 'task A' priority to 1: {pq}")

print(f"Dequeued: {pq.dequeue()}") # Expected: task A
print(f"PQ after third dequeue: {pq}")

print(f"Dequeued: {pq.dequeue()}") # Expected: task D
print(f"PQ after fourth dequeue: {pq}")

# Test with no more items
print(f"\nIs PQ empty? {pq.is_empty()}")
print(f"PQ size: {pq.size()}")

# Test dequeue/peek from empty queue
try:
    pq.dequeue()
except KeyError as e:
    print(f"Error dequeuing from empty PQ: {e}")

try:
    pq.peek()
except KeyError as e:
    print(f"Error peeking from empty PQ: {e}")
```

```
try:
    pq.peek()
except KeyError as e:
    print(f"Error peeking from empty PQ: {e}")
```

```
Is PQ empty? True

Enqueuing items:
PQ after enqueues: PriorityQueue(['task B', 'task C', 'task A', 'task D'])
PQ size: 4

Peek highest priority item: task B
PQ after peek: PriorityQueue(['task B', 'task C', 'task A', 'task D'])

Dequeuing items:
Dequeued: task B
PQ after first dequeue: PriorityQueue(['task C', 'task A', 'task D'])
Dequeued: task C
PQ after second dequeue: PriorityQueue(['task A', 'task D'])

PQ after updating 'task A' priority to 1: PriorityQueue(['task A', 'task D'])
Dequeued: task A
PQ after third dequeue: PriorityQueue(['task D'])
Dequeued: task D
PQ after fourth dequeue: PriorityQueue([])

Is PQ empty? True
PQ size: 0
Error dequeuing from empty PQ: 'dequeue from an empty priority queue'
Error peeking from empty PQ: 'peek from an empty priority queue'
```

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.

```python
import collections

class DequeDS:
    """
    A Double-Ended Queue (Deque) implementation using Python's collections.deque.
    Allows insertion and removal from both ends efficiently.
    """
    def __init__(self):
        """
        Initializes an empty Deque.
        """
        self._deque = collections.deque()

    def insert_front(self, item):
        """
        Adds an item to the front of the deque.

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

    def insert_rear(self, item):
        """
        Adds an item to the rear of the deque.

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

    def remove_front(self):
```

```
def remove_front(self):
    """
    Removes and returns the item from the front of the deque.

    Returns:
        The item that was at the front of the deque.

    Raises:
        IndexError: If the deque is empty.
    """
    if self.is_empty():
        raise IndexError("remove_front from empty deque")
    return self._deque.popleft()

def remove_rear(self):
    """
    Removes and returns the item from the rear of the deque.

    Returns:
        The item that was at the rear of the deque.

    Raises:
        IndexError: If the deque is empty.
    """
    if self.is_empty():
        raise IndexError("remove_rear from empty deque")
    return self._deque.pop()

def peek_front(self):
    """
    Returns the item at the front of the deque without removing it.

    Returns:
```

```
    Returns:
        The item at the front of the deque.

    Raises:
        IndexError: If the deque is empty.
    """
    if self.is_empty():
        raise IndexError("peek_front from empty deque")
    return self._deque[0]

def peek_rear(self):
    """
    Returns the item at the rear of the deque without removing it.

    Returns:
        The item at the rear of the deque.

    Raises:
        IndexError: If the deque is empty.
    """
    if self.is_empty():
        raise IndexError("peek_rear from empty deque")
    return self._deque[-1]

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

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

```
        return len(self._deque) == 0

    def size(self):
        """
        Returns the number of items in the deque.
        """
        return len(self._deque)

    def __str__(self):
        """
        Returns a string representation of the deque.
        """
        return str(list(self._deque))

    def __repr__(self):
        """
        Returns a detailed string representation of the deque for debugging.
        """
        return f"DequeDS({list(self._deque)})"
```

ou can test this Deque implementation with the following code:

```
# Create a new Deque
my_deque = DequeDS()

# Test is_empty
print(f"Is deque empty? {my_deque.is_empty()}")

# Test insert_front and insert_rear
print("\nInserting items:")
my_deque.insert_rear('b')
```

```
print("\nInserting items:")
my_deque.insert_rear('b')
my_deque.insert_front('a')
my_deque.insert_rear('c')
my_deque.insert_front('0')
print(f"Deque after insertions: {my_deque}") # Expected: ['0', 'a', 'b', 'c']
print(f"Deque size: {my_deque.size()}")

# Test is_empty again
print(f"Is deque empty? {my_deque.is_empty()}")

# Test peek_front and peek_rear
print(f"\nPeek front: {my_deque.peek_front()}") # Expected: '0'
print(f"Peek rear: {my_deque.peek_rear()}")   # Expected: 'c'
print(f"Deque after peeks: {my_deque}")       # Should remain unchanged

# Test remove_front
print("\nRemoving from front:")
removed_item_front = my_deque.remove_front()
print(f"Removed from front: {removed_item_front}") # Expected: '0'
print(f"Deque after remove_front: {my_deque}")
print(f"Deque size: {my_deque.size()}")

# Test remove_rear
print("\nRemoving from rear:")
removed_item_rear = my_deque.remove_rear()
print(f"Removed from rear: {removed_item_rear}") # Expected: 'c'
print(f"Deque after remove_rear: {my_deque}")
print(f"Deque size: {my_deque.size()}")

# Remove remaining items
my_deque.remove_front()
my deque.remove rear()
```

```python
# Remove remaining items
my_deque.remove_front()
my_deque.remove_rear()
print(f"\nDeque after all removals: {my_deque}")
print(f"Is deque empty? {my_deque.is_empty()}")

# Test remove/peek on empty deque (this will raise an error)
print("\nTesting error handling for empty deque:")
try:
    my_deque.remove_front()
except IndexError as e:
    print(f"Error removing from empty deque: {e}")

try:
    my_deque.peek_rear()
except IndexError as e:
    print(f"Error peeking from empty deque: {e}")
```

```
Is deque empty? True

Inserting items:
Deque after insertions: ['0', 'a', 'b', 'c']
Deque size: 4
Is deque empty? False

Peek front: 0
Peek rear: c
Deque after peeks: ['0', 'a', 'b', 'c']

Removing from front:
Removed from front: 0
Deque after remove_front: ['a', 'b', 'c']
Deque size: 3

Removing from front:
Removed from front: 0
Deque after remove_front: ['a', 'b', 'c']
Deque size: 3

Removing from rear:
Removed from rear: c
Deque after remove_rear: ['a', 'b']
Deque size: 2

Deque after all removals: []
Is deque empty? True

Testing error handling for empty deque:
Error removing from empty deque: remove_front from empty deque
Error peeking from empty deque: peek_rear from empty deque
```

Task Description #9 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.

Task

Define a `Graph` class using an adjacency list (dictionary) to store vertices and their connections. Implement an `__init__` method to initialize the graph and specify if it's directed. Include `add_vertex` to add nodes (bus stops) and `add_edge` to connect them (bus routes), handling both directed and undirected edges. Also, implement a `display` method to print the graph's connections, and `__str__` and `__repr__` methods for string representation.

Define Graph Classes

Subtask:

Define the `Graph` class using an adjacency list (dictionary) to store vertices and their connections. Include an `__init__` method to initialize the graph and specify if it's directed. Implement `add_vertex` to add nodes (bus stops) and `add_edge` to connect them (bus routes), handling both directed and undirected edges. Also, include a `display` method to print the graph's connections, and `__str__` and `__repr__` methods for string representation.

Reasoning: The subtask requires defining a `Graph` class with specific methods for initialization, adding vertices and edges, displaying connections, and string representations. I will provide the Python code for this class.

[21] ✓ 0s

```
class Graph:
    def __init__(self, directed=False):
        self.graph = {}
        self.directed = directed

    def add_vertex(self, vertex):
```

```
    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []

    def add_edge(self, vertex1, vertex2, weight=None):
        self.add_vertex(vertex1)
        self.add_vertex(vertex2)

        edge_data = (vertex2, weight) if weight is not None else vertex2
        self.graph[vertex1].append(edge_data)

        if not self.directed:
            edge_data_reverse = (vertex1, weight) if weight is not None else vertex1
            self.graph[vertex2].append(edge_data_reverse)

    def display(self):
        for vertex, neighbors in self.graph.items():
            print(f"{vertex}: {neighbors}")

    def __str__(self):
        graph_str = []
        for vertex, neighbors in self.graph.items():
            graph_str.append(f"{vertex}: {neighbors}")
        return "\n".join(graph_str)

    def __repr__(self):
        num_vertices = len(self.graph)
        num_edges = sum(len(neighbors) for neighbors in self.graph.values())
        if not self.directed:
            num_edges //= 2  # Each edge added twice for undirected graph
        return f"Graph(directed={self.directed}, vertices={num_vertices}, edges={num_edges})"
```

Test Graph Implementation

Subtask:

Write and execute Python code to demonstrate the functionality of the `Graph` class. This includes creating both undirected and directed graphs, adding vertices, adding edges, and using the `display` method to show the connections. Also, test edge cases such as adding an existing vertex or displaying an empty graph.

Reasoning: The subtask requires writing and executing Python code to thoroughly test the `Graph` class, including its initialization, `add_vertex`, `add_edge`, `display`, `__str__`, and `__repr__` methods for both directed and undirected graphs, as well as handling edge cases.

[22] ✓ 0s

```
print("\n--- Testing Undirected Graph ---")
# 1. Create an instance of an undirected Graph named `undirected_graph`.
undirected_graph = Graph(directed=False)

# 2. Add several vertices
undirected_graph.add_vertex('A')
undirected_graph.add_vertex('B')
undirected_graph.add_vertex('C')
undirected_graph.add_vertex('D')

# 3. Add edges, including some weighted ones
undirected_graph.add_edge('A', 'B', weight=10)
undirected_graph.add_edge('B', 'C')
undirected_graph.add_edge('A', 'D', weight=5)
undirected_graph.add_edge('D', 'B', weight=7)
```

```
# 4. Call the `display` method
print("Undirected graph display:")
undirected_graph.display()

# 5. Print the `undirected_graph` object directly (tests __str__)
print("\nUndirected graph __str__ output:")
print(undirected_graph)

# 6. Print the `repr()` of the `undirected_graph` object (tests __repr__)
print("\nUndirected graph __repr__ output:")
print(repr(undirected_graph))

print("\n--- Testing Directed Graph ---")
# 7. Create an instance of a directed Graph named `directed_graph`
directed_graph = Graph(directed=True)

# 8. Add several vertices
directed_graph.add_vertex('X')
directed_graph.add_vertex('Y')
directed_graph.add_vertex('Z')

# 9. Add directed edges
directed_graph.add_edge('X', 'Y', weight=2)
directed_graph.add_edge('Y', 'Z', weight=3)
directed_graph.add_edge('X', 'Z')

# 10. Call the `display` method
print("Directed graph display:")
directed_graph.display()

# 11. Print the `directed_graph` object directly (tests __str__)
print("\nDirected graph __str__ output:")
```

```python
# 12. Print the `repr()` of the `directed_graph` object (tests __repr__)
print("\nDirected graph __repr__ output:")
print(repr(directed_graph))

print("\n--- Testing Edge Cases ---")
# 13. Test adding an existing vertex
print("Attempting to add existing vertex 'A' to undirected_graph...")
undirected_graph.add_vertex('A') # Should not change the graph
print("Undirected graph after adding existing vertex 'A':")
undirected_graph.display()

# 14. Create an empty graph instance and call its methods
print("\nTesting an empty graph:")
empty_graph = Graph()

print("Empty graph display:")
empty_graph.display()

print("Empty graph __str__ output:")
print(empty_graph)

print("Empty graph __repr__ output:")
print(repr(empty_graph))
```

```
--- Testing Undirected Graph ---
Undirected graph display:
A: [('B', 10), ('D', 5)]
B: [('A', 10), 'C', ('D', 7)]
C: ['B']
D: [('A', 5), ('B', 7)]

Undirected graph __str__ output:
```

```
Undirected graph __str__ output:
A: [('B', 10), ('D', 5)]
B: [('A', 10), 'C', ('D', 7)]
C: ['B']
D: [('A', 5), ('B', 7)]

Undirected graph __repr__ output:
Graph(directed=False, vertices=4, edges=4)

--- Testing Directed Graph ---
Directed graph display:
X: [('Y', 2), 'Z']
Y: [('Z', 3)]
Z: []

Directed graph __str__ output:
X: [('Y', 2), 'Z']
Y: [('Z', 3)]
Z: []

Directed graph __repr__ output:
Graph(directed=True, vertices=3, edges=3)

--- Testing Edge Cases ---
Attempting to add existing vertex 'A' to undirected_graph...
Undirected graph after adding existing vertex 'A':
A: [('B', 10), ('D', 5)]
B: [('A', 10), 'C', ('D', 7)]
C: ['B']
D: [('A', 5), ('B', 7)]

Testing an empty graph:
Empty graph display:
Empty graph __str__ output:
```

```
Empty graph __repr__ output:
Graph(directed=False, vertices=0, edges=0)
```

Final Task

Subtask:

Summarize the implementation of the Graph data structure for the Bus Scheduling System, confirming its functionality and how it addresses the requirements of maintaining bus routes and stop connections.

Summary:

Q&A

The Graph data structure implementation successfully addresses the requirements of maintaining bus routes and stop connections by:

- Representing bus stops as vertices, added via the `add_vertex` method.
- Representing bus routes between stops as edges, added via the `add_edge` method.
- Supporting both one-way and two-way bus routes through the `directed` parameter in the graph initialization. Directed graphs model one-way routes, while undirected graphs model two-way routes.
- Allowing for the inclusion of route characteristics (such as distance or travel time) using the `weight` parameter when adding edges.
- Providing clear visualization and programmatic access to the network structure through `display`, `__str__`, and `__repr__` methods.

Data Analysis Key Findings

- The `Graph` class was successfully implemented using an adjacency list, capable of initializing as either a directed or undirected graph.
- The `add_vertex` method correctly adds new bus stops (vertices) to the graph, preventing duplicate entries.
- The `add_edge` method accurately creates connections (bus routes) between bus stops, including the ability to assign weights (e.g., travel time or distance) to these routes. For undirected graphs, it automatically creates bidirectional connections.
- Testing demonstrated correct functionality for both undirected and directed graphs:
  - An undirected graph with 4 vertices and 4 edges (`Graph(directed=False, vertices=4, edges=4)`) was accurately represented.
  - A directed graph with 3 vertices and 3 edges (`Graph(directed=True, vertices=3, edges=3)`) was also correctly represented, reflecting the one-way nature of its connections.
- Edge cases were handled as expected:
  - Attempting to add an existing vertex had no effect on the graph.
  - An empty graph instance correctly reported 0 vertices and 0 edges (`Graph(directed=False, vertices=0, edges=0)`) and showed no connections.

Insights or Next Steps

- The current `Graph` implementation forms a solid foundation for a bus scheduling system. The use of weighted edges is particularly useful for future algorithms that might calculate shortest paths based on time or distance.
- Further development could involve adding methods for common graph operations pertinent to bus scheduling, such as finding paths between two stops (e.g., Dijkstra's algorithm for shortest routes) or identifying connected components to see which areas are serviced.

Task Description #10: Smart E-Commerce Platform – Data Structure

Challenge

An e-commerce company wants to build a Smart Online Shopping System

with:

1. Shopping Cart Management – Add and remove products

dynamically.

2. Order Processing System – Orders processed in the order they are

placed.

3. Top-Selling Products Tracker – Products ranked by sales count.

4. Product Search Engine – Fast lookup of products using product ID.

5. Delivery Route Planning – Connect warehouses and delivery locations.

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.

Task

Implement a hash table to function as a product search engine. This implementation should include `Node` and `LinkedListForHashTable` classes for collision handling, and a `HashTable` class with methods for `insert`, `search`, `delete`, and `display` product details. After implementing, thoroughly test the hash table's functionality by inserting, updating, searching, and deleting products, and finally summarize its performance and key findings.

Define Node and LinkedListForHashTable Classes

Subtask:

Implement a `Node` class to store key-value pairs (product ID and product details) and a `LinkedListForHashTable` class for collision handling. The `LinkedListForHashTable` will include methods for `insert_or_update`, `search`, and `delete`.

Reasoning: The subtask requires defining `Node` and `LinkedListForHashTable` classes, including `insert_or_update`, `search`, and `delete` methods. This code block will implement these classes and methods as specified in the instructions.

[23] ✓ 0s

```
class Node:
    def __init__(self, key, value, next=None):
        self.key = key
        self.value = value
        self.next = next

class LinkedListForHashTable:
    def __init__(self):
```

```
class LinkedListForHashTable:
    def __init__(self):
        self.head = None

    def insert_or_update(self, key, value):
        current = self.head
        while current:
            if current.key == key:
                current.value = value
                print(f"Key '{key}' updated with new value.")
                return
            current = current.next

        # If key not found, insert new node at the beginning
        new_node = Node(key, value, self.head)
        self.head = new_node
        print(f"Key '{key}' inserted.")

    def search(self, key):
        current = self.head
        while current:
            if current.key == key:
                print(f"Found key '{key}' with value: {current.value}")
                return current.value
            current = current.next
        print(f"Key '{key}' not found.")
        return None

    def delete(self, key):
        current = self.head
        prev = None
```

[23]
✓ 0s

```
        while current:
            if current.key == key:
                if prev:
                    prev.next = current.next
                else:
                    self.head = current.next
                print(f"Key '{key}' deleted.")
                return True
            prev = current
            current = current.next

        print(f"Key '{key}' not found for deletion.")
        return False

print("Node and LinkedListForHashTable classes defined successfully.")
```

```
Node and LinkedListForHashTable classes defined successfully.
```

Define HashTable Class

Subtask:

Implement the `HashTable` class with an internal array of `LinkedListForHashTable` instances (buckets). Include an `__init__` method to set the table size, a `_hash_function` to map keys to array indices, and core methods like `insert` (to add/update products), `search` (to find product details by ID), and `delete` (to remove products). Also, include a `display` method to visualize the hash table's contents.

Reasoning: The subtask requires implementing the `HashTable` class with its core methods (`__init__`, `_hash_function`, `insert`, `search`, `delete`, `display`) as described in the instructions. This code block will define the `HashTable` class with all specified functionalities.

[24]
✓ 0s

```
class HashTable:
    def __init__(self, capacity):
        self.capacity = capacity
        self.buckets = [LinkedListForHashTable() for _ in range(self.capacity)]
        print(f"HashTable initialized with capacity: {self.capacity}")

    def _hash_function(self, key):
        return hash(key) % self.capacity

    def insert(self, key, value):
        index = self._hash_function(key)
        print(f"Inserting key '{key}' into bucket {index}.")
        self.buckets[index].insert_or_update(key, value)

    def search(self, key):
        index = self._hash_function(key)
        print(f"Searching for key '{key}' in bucket {index}.")
        return self.buckets[index].search(key)

    def delete(self, key):
        index = self._hash_function(key)
        print(f"Attempting to delete key '{key}' from bucket {index}.")
        return self.buckets[index].delete(key)

    def display(self):
        print("\n--- Hash Table Contents ---")
```

[24] ✓ 0s

```
        def display(self):
            print("\n--- Hash Table Contents ---")
            for i, bucket in enumerate(self.buckets):
                print(f"Bucket {i}:")
                current = bucket.head
                if not current:
                    print("  (Empty)")
                else:
                    items = []
                    while current:
                        items.append(f"  Key: '{current.key}', Value: {current.value}")
                        current = current.next
                    for item in items:
                        print(item)
            print("--------------------------")

print("HashTable class defined successfully.")
```

```
HashTable class defined successfully.
```

Test HashTable Implementation

Subtask:

Write and execute Python code to thoroughly test the `HashTable` class. This will involve inserting various product IDs and their details, updating existing product details, searching for both present and absent product IDs, and deleting products. It should also demonstrate how collisions are handled and the overall efficiency of operations.

Reasoning: The subtask requires thoroughly testing the `HashTable` class by performing various operations like insertion, updating, searching, and deleting products, and observing collision handling. This code block will execute all these tests as per the instructions.

[25] ✓ 0s

```
print("\n--- Testing HashTable Functionality ---")

# 1. Instantiate a HashTable object with a suitable capacity
hash_table = HashTable(capacity=5)

# 2. Insert several unique product IDs and their details
print("\n--- Initial Insertions ---")
hash_table.insert("P101", {"name": "Laptop", "price": 1200})
hash_table.insert("P202", {"name": "Mouse", "price": 25})
hash_table.insert("P303", {"name": "Keyboard", "price": 75})
hash_table.insert("P404", {"name": "Monitor", "price": 300})
hash_table.insert("P505", {"name": "Webcam", "price": 50})

# Inserting keys that might cause collisions based on common hash functions (simple string hashes)
# Assuming 'P101' and 'P606' might hash to the same or nearby buckets for demonstration
hash_table.insert("P606", {"name": "Speakers", "price": 100}) # Potential collision
hash_table.insert("P707", {"name": "Headphones", "price": 150}) # Another potential collision

# 3. Display the current state of the hash table
hash_table.display()

# 4. Update the details of an existing product
print("\n--- Updating Product P101 ---")
hash_table.insert("P101", {"name": "Gaming Laptop", "price": 1500, "warranty": "1 year"})
hash_table.display()

# 5. Search for an existing product
print("\n--- Searching for existing product P303 ---")
```

```python
# 5. Search for an existing product
print("\n--- Searching for existing product P303 ---")
found_product = hash_table.search("P303")
if found_product: print(f"Product P303 details: {found_product}")

# 6. Search for a product that does not exist
print("\n--- Searching for non-existent product P999 ---")
hash_table.search("P999")

# 7. Delete an existing product
print("\n--- Deleting product P202 ---")
hash_table.delete("P202")
hash_table.display()

# 8. Attempt to delete a product that does not exist
print("\n--- Attempting to delete non-existent product P888 ---")
hash_table.delete("P888")
hash_table.display()

# 9. Display the hash table again to confirm changes
print("\n--- Final Hash Table State After Deletion ---")
hash_table.display()

print("--- Hash Table Testing Complete ---")
```

```
--- Testing HashTable Functionality ---
HashTable initialized with capacity: 5

--- Initial Insertions ---
Inserting key 'P101' into bucket 2.
Key 'P101' inserted.
Inserting key 'P202' into bucket 0.
```

```
Key 'P303' inserted.
Inserting key 'P404' into bucket 1.
Key 'P404' inserted.
Inserting key 'P505' into bucket 1.
Key 'P505' inserted.
Inserting key 'P606' into bucket 0.
Key 'P606' inserted.
Inserting key 'P707' into bucket 1.
Key 'P707' inserted.

--- Hash Table Contents ---
Bucket 0:
  Key: 'P606', Value: {'name': 'Speakers', 'price': 100}
  Key: 'P202', Value: {'name': 'Mouse', 'price': 25}
Bucket 1:
  Key: 'P707', Value: {'name': 'Headphones', 'price': 150}
  Key: 'P505', Value: {'name': 'Webcam', 'price': 50}
  Key: 'P404', Value: {'name': 'Monitor', 'price': 300}
Bucket 2:
  Key: 'P303', Value: {'name': 'Keyboard', 'price': 75}
  Key: 'P101', Value: {'name': 'Laptop', 'price': 1200}
Bucket 3:
  (Empty)
Bucket 4:
  (Empty)
-------------------------

--- Updating Product P101 ---
Inserting key 'P101' into bucket 2.
Key 'P101' updated with new value.

--- Hash Table Contents ---
Bucket 0:
  Key: 'P606', Value: {'name': 'Speakers', 'price': 100}
  Key: 'P202', Value: {'name': 'Mouse', 'price': 25}
Bucket 2:
  Key: 'P303', Value: {'name': 'Keyboard', 'price': 75}
  Key: 'P101', Value: {'name': 'Gaming Laptop', 'price': 1500, 'warranty': '1 year'}
Bucket 3:
  (Empty)
Bucket 4:
  (Empty)
-------------------------

--- Final Hash Table State After Deletion ---

--- Hash Table Contents ---
Bucket 0:
  Key: 'P606', Value: {'name': 'Speakers', 'price': 100}
Bucket 1:
  Key: 'P707', Value: {'name': 'Headphones', 'price': 150}
  Key: 'P505', Value: {'name': 'Webcam', 'price': 50}
  Key: 'P404', Value: {'name': 'Monitor', 'price': 300}
Bucket 2:
  Key: 'P303', Value: {'name': 'Keyboard', 'price': 75}
  Key: 'P101', Value: {'name': 'Gaming Laptop', 'price': 1500, 'warranty': '1 year'}
Bucket 3:
  (Empty)
Bucket 4:
  (Empty)
-------------------------
--- Hash Table Testing Complete ---
```

Final Task

Subtask:

Summarize the implementation of the Hash Table for the Product Search Engine, confirming its functionality and how it addresses the requirements for fast lookup of products using product ID, including a brief explanation of the data analysis key findings, and next steps.

Summary:

Q&A

The implemented Hash Table effectively functions as a product search engine, leveraging `Node` and `LinkedListForHashTable` classes for robust collision handling.

- **Functionality:**
  - The `HashTable` class supports `insert`, `search`, and `delete` operations for product details using product IDs as keys. A `display` method is also included to visualize the table's contents.
  - `insert` adds new products or updates existing ones, while `search` retrieves product details quickly, and `delete` removes products.
- **Addressing Fast Lookup Requirements:**
  - The core of fast lookup is achieved through the `_hash_function`, which maps product IDs to a specific bucket index in O(1) average time.
  - Collisions, where multiple keys hash to the same index, are handled using separate chaining via `LinkedListForHashTable`. This ensures that even with collisions, operations within a bucket are efficient (O(k) where k is the number of items in the

  - The core of fast lookup is achieved through the `_hash_function`, which maps product IDs to a specific bucket index in O(1) average time.
  - Collisions, where multiple keys hash to the same index, are handled using separate chaining via `LinkedListForHashTable`. This ensures that even with collisions, operations within a bucket are efficient (O(k) where k is the number of items in the bucket, which is small on average).
  - For product IDs, this design allows for near constant-time average performance for insertion, search, and deletion, fulfilling the requirement for fast lookup.

Data Analysis Key Findings

- The `Node` class was successfully defined to store key-value pairs (`product ID`, `product details`) and link them within a collision chain.
- The `LinkedListForHashTable` class was correctly implemented with `insert_or_update`, `search`, and `delete` methods, providing the necessary mechanism for handling collisions by managing nodes within each hash table bucket.
- The `HashTable` class was initialized with a specified capacity (e.g., 5 buckets) and included a `_hash_function` to distribute keys across these buckets.
- Testing confirmed successful insertion of products like 'P101', 'P202', 'P303', 'P404', 'P500', 'P606', and 'P707'.
- Collision handling was demonstrated effectively; for instance, 'P202' and 'P606' hashed to 'Bucket 0', while 'P404', 'P505', and 'P707' hashed to 'Bucket 1', showing products correctly stored within linked lists in their respective buckets.
- Updating existing product details using `insert` with the same key was successful, as shown when 'P101' was updated from "Laptop" to "Gaming Laptop" with an additional warranty detail.
- Searching for existing products, like 'P303', successfully retrieved its details.
- Searching for non-existent products, like 'P999', correctly indicated that the key was not found.
- Deletion of existing products, such as 'P202', was successful and reflected in the hash table's display.
- Attempts to delete non-existent products, like 'P888', correctly reported that the key was not found.

Insights or Next Steps

- The implemented hash table design effectively balances fast average-case lookup with robust collision resolution, making it suitable for a product search engine.
- Further performance analysis could involve measuring average operation times with a larger dataset and different hash table capacities to optimize performance for production use.