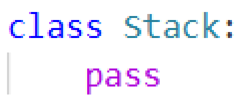
Lab Assignment 11.1

Program	**: B. Tech (CSE)**
Course Title	**: AI Assisted Coding**
Course Code	**: 23CS002PC304**
Semester	**: III**
Academic Session	**: 2025-2026**
Name of Student	**: Sruthi**
Enrollment No.	**: 2403A51L10**
Batch No.	**: 51**
Date	**: 17-02-2026**

Task 1:

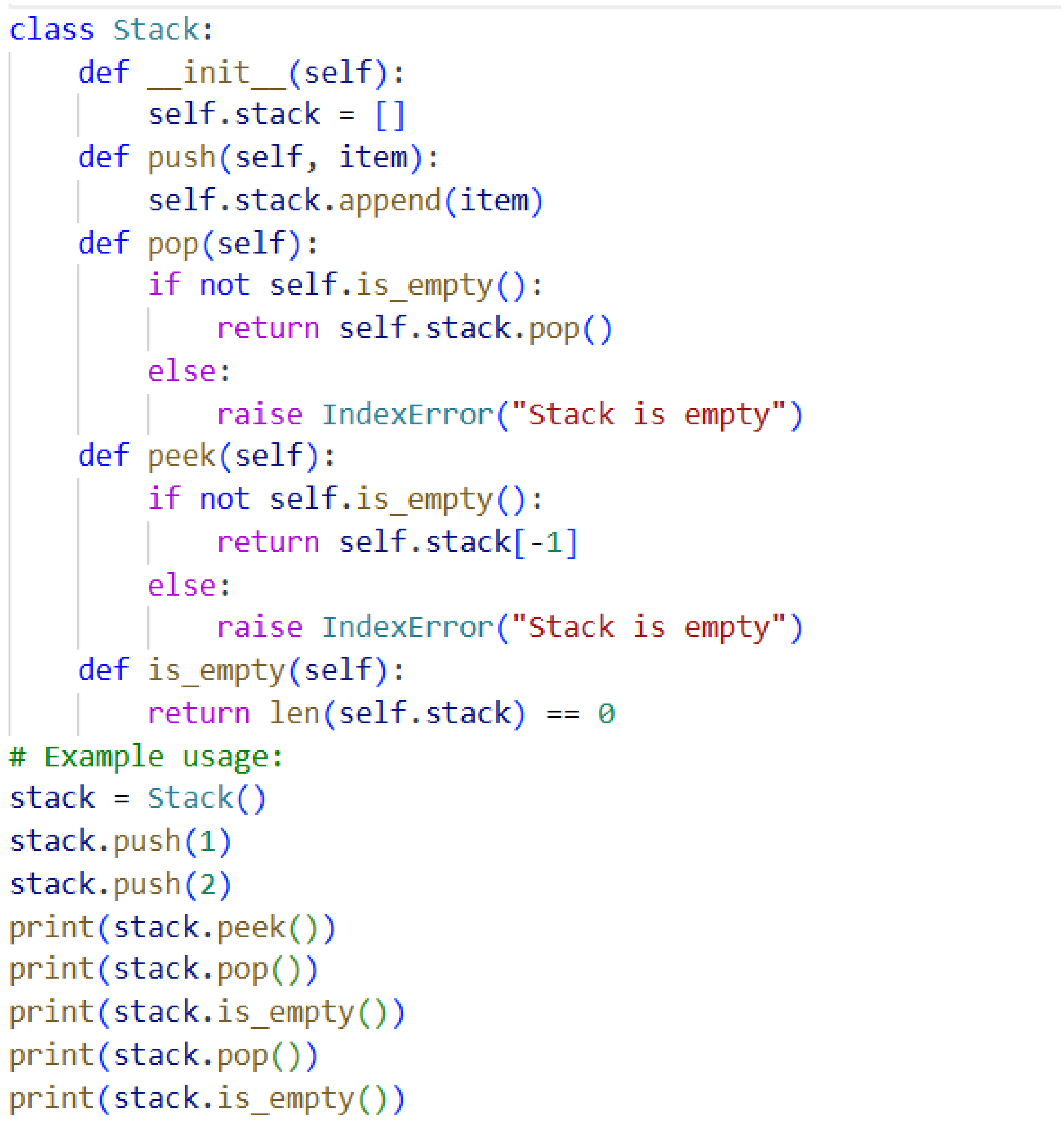
Sample input code:



```python
class Stack:
    pass
```

Prompt: Write a python program a stack class that follows the push, pop, peek and is_empty methods.

Code:



```python
class Stack:
    def __init__(self):
        self.stack = []
    def push(self, item):
        self.stack.append(item)
    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("Stack is empty")
    def peek(self):
        if not self.is_empty():
            return self.stack[-1]
        else:
            raise IndexError("Stack is empty")
    def is_empty(self):
        return len(self.stack) == 0
# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
print(stack.peek())
print(stack.pop())
print(stack.is_empty())
print(stack.pop())
print(stack.is_empty())
```

Output:

```
2
2
False
1
True
```

Explanation:

The Stack class is implemented using a list to store the elements.
The push method adds an item to the top of the stack.
The pop method removes and returns the top item.
The peek method allows you to see the top item without removing it.
The is_empty method checks if the stack is empty.

Task 2:

Sample code:

```python
class Queue:
    pass
```

Prompt: Write a python program to implement a queue.

Code:

```python
class Queue:
    def __init__(self):
        self.queue = []
    def enqueue(self, item):
        self.queue.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise IndexError("Queue is empty")
    def peek(self):
        if not self.is_empty():
            return self.queue[0]
        else:
            raise IndexError("Queue is empty")
    def is_empty(self):
        return len(self.queue) == 0
# Example usage:
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
print(queue.peek())
print(queue.dequeue())
print(queue.is_empty())
print(queue.dequeue())
print(queue.is_empty())
```

Output:

```
1
1
False
2
True
```

Explanation:

A Queue class that uses a list to store the elements of the queue.

- The enqueue method adds an item to the end of the queue.
- The dequeue method removes and returns the item at the front of the queue.
- The peek method allows us to see the front item without removing it, and
- The is_empty method checks if the queue is empty.

Task 3:

Sample Code:

```
class Node:
    pass
class LinkedList:
    pass
```

Prompt: Generate a Singly linked list with insert and display methods.

Code:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=' ')
            current = current.next
# Example usage:
linked_list = LinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()
```

Output:

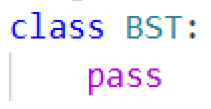
```
1 2 3
```

Explanation:

Defined a Node class to represent each node in the linked list, which contains data and a pointer to the next node. The LinkedList class contains methods to insert new nodes at the end of the list and to display the contents of the list. The insert method creates a new node and adds it to the end of the list, while the display method traverses the list and prints each node's data.

Task 4:

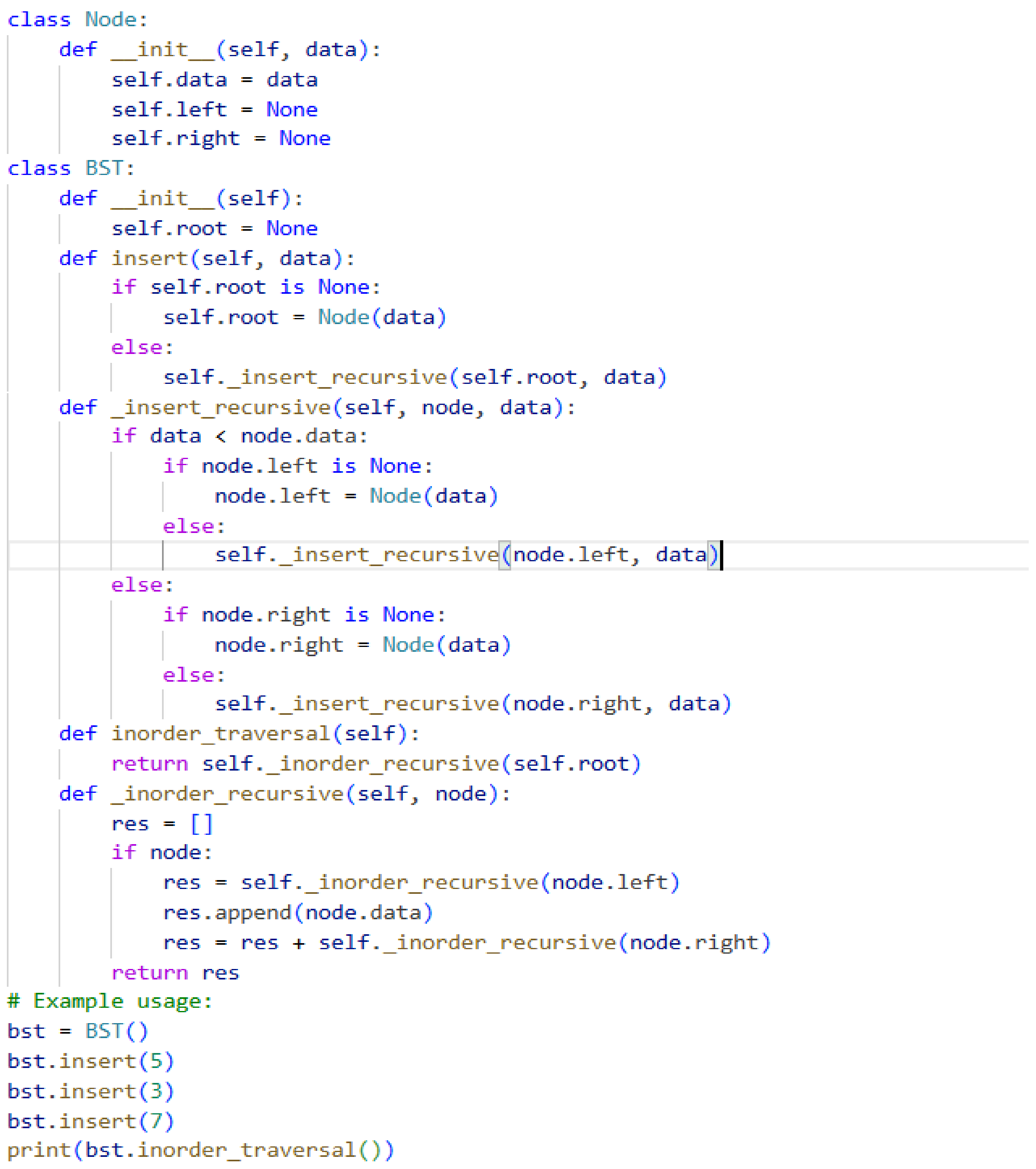
Sample Code:



```python
class BST:
    pass
```

Prompt: Generate a BST with insert and in-order traversal methods.

Code:



```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
class BST:
    def __init__(self):
        self.root = None
    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)
    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = Node(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = Node(data)
            else:
                self._insert_recursive(node.right, data)
    def inorder_traversal(self):
        return self._inorder_recursive(self.root)
    def _inorder_recursive(self, node):
        res = []
        if node:
            res = self._inorder_recursive(node.left)
            res.append(node.data)
            res = res + self._inorder_recursive(node.right)
        return res
# Example usage:
bst = BST()
bst.insert(5)
bst.insert(3)
bst.insert(7)
print(bst.inorder_traversal())
```

Output:

```
[3, 5, 7]
```

Explanation:

Defined a Node class to represent each node in the binary search tree (BST) and a BST class to manage the tree structure. The BST class has methods to insert new values and perform an in-order traversal.
The insert method adds new nodes to the tree while maintaining the BST property, and the inorder_traversal method returns a list of values in sorted order.

Task 5:

Sample Code:

```python
class HashTable:
    pass
```

Prompt: Implement a hash table with insert, search and delete methods.

Code:

```python
class HashTable:
    def __init__(self):
        self.table = {}
    def insert(self, key, value):
        self.table[key] = value
    def search(self, key):
        return self.table.get(key, None)
    def delete(self, key):
        if key in self.table:
            del self.table[key]
# Example usage:
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print(hash_table.search("name"))
print(hash_table.search("age"))
hash_table.delete("name")
print(hash_table.search("name"))
```

Output:

```
Alice
30
None
```

Explanation:

Define a HashTable class that uses a Python dictionary to store key-value pairs.
The insert method adds a new key-value pair to the hash table.
The search method retrieves the value associated with a given key.
The delete method removes a key-value pair from the hash table.

Task 6:

Sample code:

```
class Graph:
    pass
```

Prompt: Implement a graph using an adjacency list with add edge, add vertices and display methods.

Code:

```
class Graph:
    def __init__(self):
        self.graph = {}
    def add_vertex(self, vertex):
        if vertex not in self.graph:
            self.graph[vertex] = []
    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)
    def display(self):
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {edges}")
# Example usage:
graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_edge("A", "B")
graph.add_edge("A", "C")
graph.display()
```

Output:

```
A: ['B', 'C']
B: ['A']
C: ['A']
```

Explanation:

Defined a Graph class that uses an adjacency list to represent the graph.
The add_vertex method allows us to add vertices to the graph.
The add_edge method adds edges between two vertices.
The display method prints out the graph structure.

Task 7:

Sample Code:

```
class PriorityQueue:
    pass
```

Prompt: Implement a priority queue using Python's heapq module.

Code:

```
import heapq
class PriorityQueue:
    def __init__(self):
        self.elements = []
    def push(self, item, priority):
        heapq.heappush(self.elements, (priority, item))
    def pop(self):
        return heapq.heappop(self.elements)[1]
    def peek(self):
        return self.elements[0][1] if self.elements else None
    def is_empty(self):
        return not self.elements
# Example usage:
pq = PriorityQueue()
pq.push("task1", 2)
pq.push("task2", 1)
print(pq.peek())
print(pq.pop())
print(pq.is_empty())
```

Output:

```
task2
task2
False
```

Explanation:

The heapq module maintain a priority queue.
The push method adds an item with a specified priority.
The pop method retrieves and removes the item with the highest priority (lowest numerical value).
The peek method allows us to see the item with the highest priority without removing it.
The is_empty method checks if the priority queue is empty.

Task 8:

Sample Code:

```
class DequeDS:
    pass
```

Prompt: Implement a code for a double ended queue using collections.deque module in Python.

Code:

```python
from collections import deque
class DequeDS:
    def __init__(self):
        self.deque = deque()
    def add_front(self, item):
        self.deque.appendleft(item)
    def add_rear(self, item):
        self.deque.append(item)
    def remove_front(self):
        if not self.is_empty():
            return self.deque.popleft()
        else:
            raise IndexError("Deque is empty")
    def remove_rear(self):
        if not self.is_empty():
            return self.deque.pop()
        else:
            raise IndexError("Deque is empty")
    def peek_front(self):
        return self.deque[0] if not self.is_empty() else None
    def peek_rear(self):
        return self.deque[-1] if not self.is_empty() else None
    def is_empty(self):
        return len(self.deque) == 0
# Example usage:
deque_ds = DequeDS()
deque_ds.add_rear(1)
deque_ds.add_rear(2)
deque_ds.add_front(0)
print(deque_ds.peek_front())
print(deque_ds.peek_rear())
print(deque_ds.remove_front())
print(deque_ds.remove_rear())
print(deque_ds.is_empty())
```

Output:

```
0
2
0
2
False
```

Explanation:

Define a class `DequeDS` that implements a double-ended queue (deque) using Python's `collections.deque` module. The class provides methods to add and remove items from both the front and rear of the deque, as well as methods to peek at the front and rear items and check if the deque is empty. The example usage demonstrates how to use the `DequeDS` class to perform various operations on the deque.

Task 9:

Prompt:

College wants to develop a Campus Resource Management System that handles:

1. Student Attendance: For tracking student attendance use the data structure hash table.
2. Event Registration System: For managing event registrations use the data structure BST.
3. Library Book Borrowing: For managing book borrowing and returns use the data structure Priority queue.
4. Bus Scheduling System: For managing bus schedules and routes use the data structure Graph.
5. Cafeteria Order Queue: For managing cafeteria orders use the data structure Queue.

give the code for the above problem statement with comments and docstrings.

1.Student Attendance Code:

```python
class HashTable:
    """A simple hash table implementation for tracking student attendance."""
    def __init__(self):
        self.table = {}
    def insert(self, key, value):
        """Insert a key-value pair into the hash table."""
        self.table[key] = value
    def search(self, key):
        """Search for a value by key in the hash table."""
        return self.table.get(key, None)
    def delete(self, key):
        """Delete a key-value pair from the hash table."""
        if key in self.table:
            del self.table[key]
# Example usage:
# Student Attendance
attendance = HashTable()
attendance.insert("Alice", "Present")
attendance.insert("Bob", "Absent")
print(attendance.search("Alice"))
```

Output:

```
Present
```

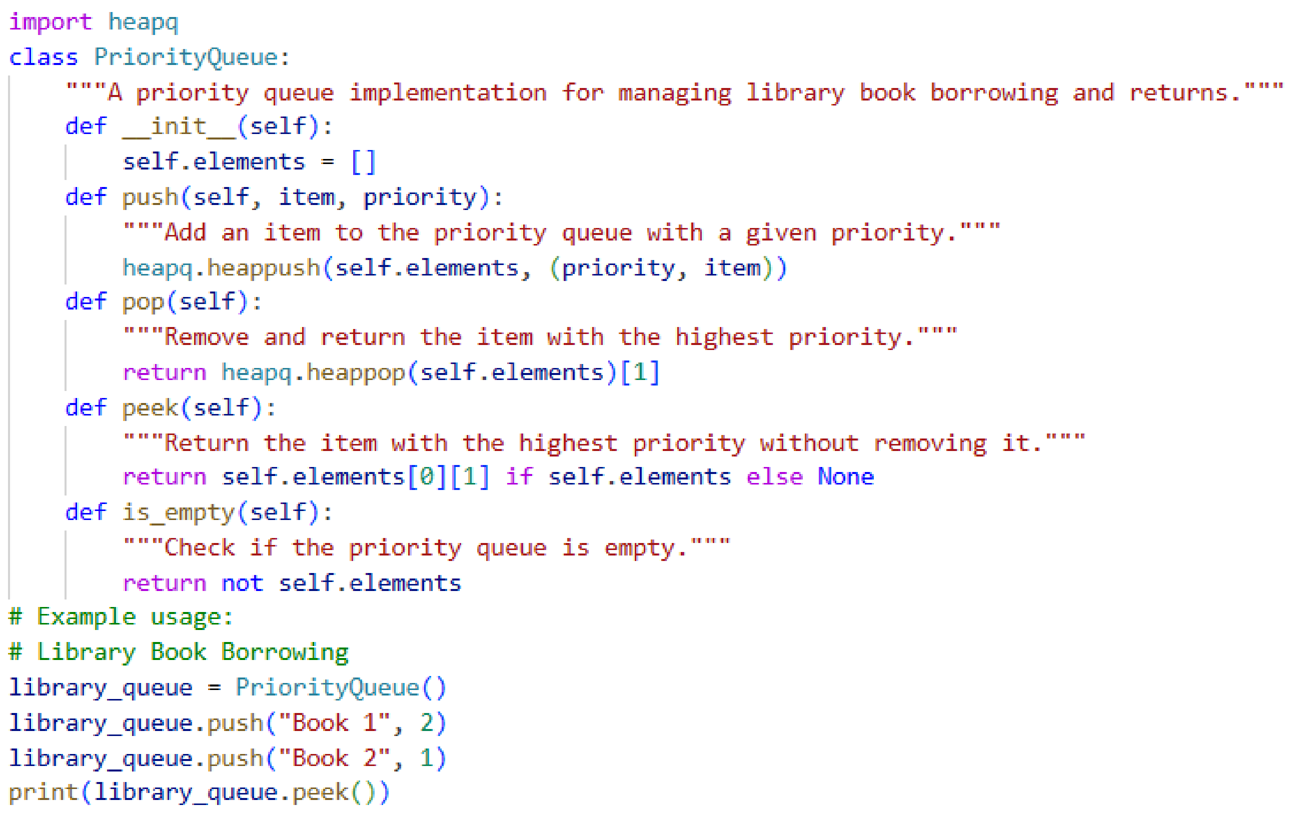
2. Event Registration System Code:

```
class BST:
    """A binary search tree implementation for managing event registrations."""
    class Node:
        def __init__(self, data):
            self.data = data
            self.left = None
            self.right = None
    def __init__(self):
        self.root = None
    def insert(self, data):
        """Insert a new event registration into the BST."""
        if self.root is None:
            self.root = self.Node(data)
        else:
            self._insert_recursive(self.root, data)
    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = self.Node(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = self.Node(data)
            else:
                self._insert_recursive(node.right, data)
    def inorder_traversal(self):
        """Return an inorder traversal of the BST."""
        return self._inorder_recursive(self.root)
    def _inorder_recursive(self, node):
        res = []
        if node:
            res = self._inorder_recursive(node.left)
            res.append(node.data)
            res = res + self._inorder_recursive(node.right)
        return res
# Example usage:
# Event Registration
event_registrations = BST()
event_registrations.insert("Event A")
event_registrations.insert("Event B")
print(event_registrations.inorder_traversal())
```

Output:

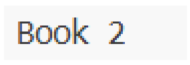
```
['Event A', 'Event B']
```

3. Library Book Borrowing Code:



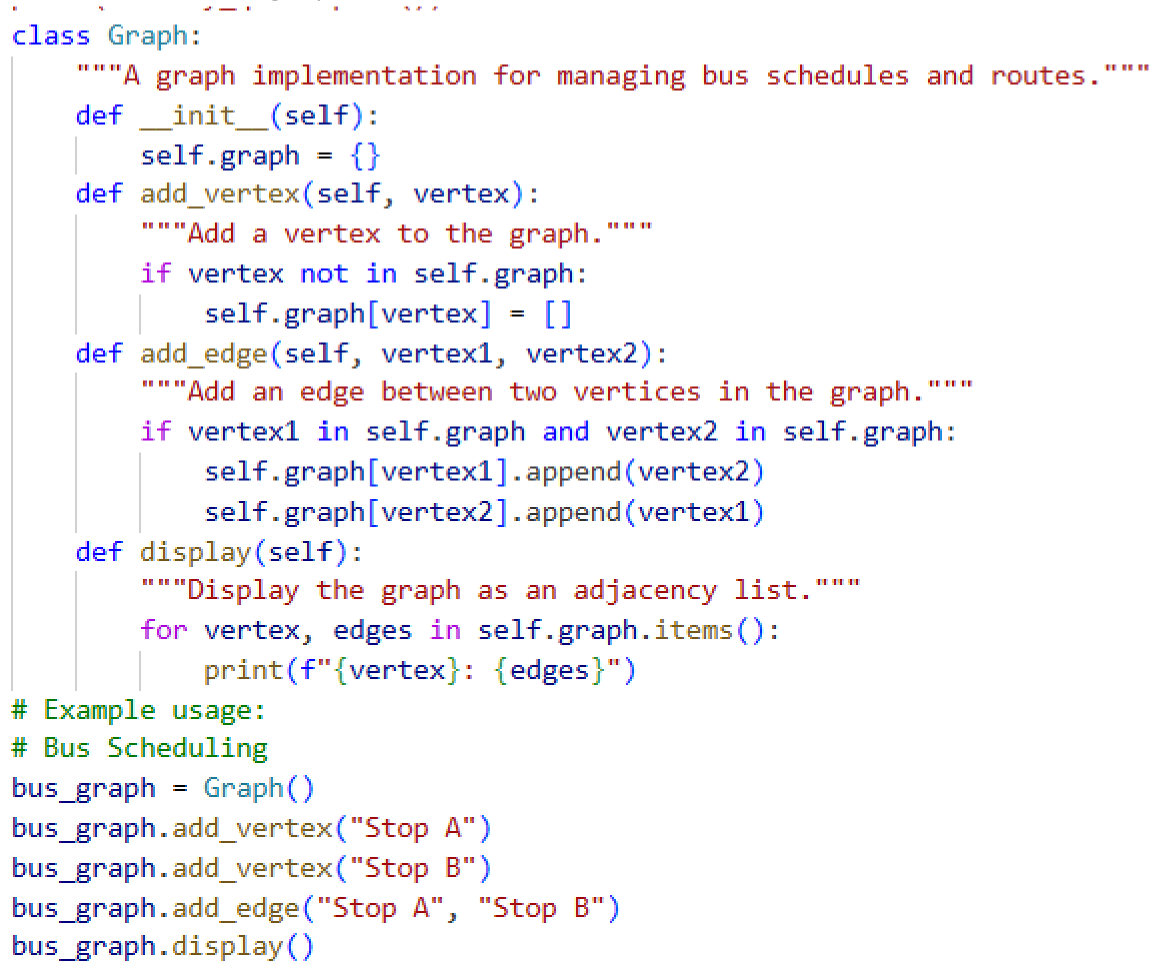
```python
import heapq
class PriorityQueue:
    """A priority queue implementation for managing library book borrowing and returns."""
    def __init__(self):
        self.elements = []
    def push(self, item, priority):
        """Add an item to the priority queue with a given priority."""
        heapq.heappush(self.elements, (priority, item))
    def pop(self):
        """Remove and return the item with the highest priority."""
        return heapq.heappop(self.elements)[1]
    def peek(self):
        """Return the item with the highest priority without removing it."""
        return self.elements[0][1] if self.elements else None
    def is_empty(self):
        """Check if the priority queue is empty."""
        return not self.elements
# Example usage:
# Library Book Borrowing
library_queue = PriorityQueue()
library_queue.push("Book 1", 2)
library_queue.push("Book 2", 1)
print(library_queue.peek())
```

Output:



```
Book 2
```

4. Bus Scheduling System Code:



```python
class Graph:
    """A graph implementation for managing bus schedules and routes."""
    def __init__(self):
        self.graph = {}
    def add_vertex(self, vertex):
        """Add a vertex to the graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
    def add_edge(self, vertex1, vertex2):
        """Add an edge between two vertices in the graph."""
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)
    def display(self):
        """Display the graph as an adjacency list."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {edges}")
# Example usage:
# Bus Scheduling
bus_graph = Graph()
bus_graph.add_vertex("Stop A")
bus_graph.add_vertex("Stop B")
bus_graph.add_edge("Stop A", "Stop B")
bus_graph.display()
```

Output:

```
Stop A: ['Stop B']
Stop B: ['Stop A']
```

5. Cafeteria Order Queue Code:

```python
class Queue:
    """A queue implementation for managing cafeteria orders."""
    def __init__(self):
        self.queue = []
    def enqueue(self, item):
        """Add an item to the rear of the queue."""
        self.queue.append(item)
    def dequeue(self):
        """Remove and return the item at the front of the queue."""
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise IndexError("Queue is empty")
    def peek(self):
        """Return the item at the front of the queue without removing it."""
        if not self.is_empty():
            return self.queue[0]
        else:
            raise IndexError("Queue is empty")
    def is_empty(self):
        """Check if the queue is empty."""
        return len(self.queue) == 0
# Example usage:
# Cafeteria Order Queue
cafeteria_queue = Queue()
cafeteria_queue.enqueue("Order 1")
cafeteria_queue.enqueue("Order 2")
print(cafeteria_queue.peek())
print(cafeteria_queue.dequeue())
print(cafeteria_queue.is_empty())
```

Output:

```
Order 1
Order 1
False
```

Feature	Chosen Data Structure	Justification
Student Attendance Tracking	Hash Table	A hash table allows quick insertion and lookup of student records using student ID as a key. Daily entry/exit logs can be efficiently updated in constant time **O (1)**. It avoids searching through the entire dataset.
Event Registration System	Binary Search Tree (BST)	BST enables fast searching, insertion, and deletion of participants in **O (log n)** time (average case). It keeps participants sorted (e.g., by registration ID or name), making quick search and removal efficient.
Library Book Borrowing	Priority Queue	Books with earlier due dates can be prioritized for reminders or returns. A priority queue efficiently retrieves the book with the nearest due date in **O (log n)** time.
Bus Scheduling System	Graph	Bus routes and stops naturally form nodes and edges. A graph helps represent connections between stops and supports route-finding algorithms like shortest path.
Cafeteria Order Queue	Queue	Students are served in First-In-First-Out (FIFO) order. A queue perfectly models this real-world waiting system, ensuring fairness.

Task 10:

Prompt:

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

1. Shopping Cart Management: Using a data structure linked list
2. Order Processing System: Using a data structure Queue
3. Top-Selling Products Tracker: Using a data structure Priority Queue
4. Product Search Engine: Using a data structure Hash Table
5. Delivery Route Planning: Using a data structure Graph

Give the code for the above problem statement with comments and docstrings.

1. **Shopping Cart Management Code:**

```python
class Node:
    """A node in a linked list."""
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    """A simple linked list implementation for managing a shopping cart."""
    def __init__(self):
        self.head = None
    def insert(self, data):
        """Insert an item into the shopping cart."""
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
    def display(self):
        """Display the items in the shopping cart."""
        current = self.head
        while current:
            print(current.data, end=' ')
            current = current.next
# Example usage:
# Shopping Cart Management
shopping_cart = LinkedList()
shopping_cart.insert("Item 1")
shopping_cart.insert("Item 2")
shopping_cart.display()
```

Output:

```
Item 1 Item 2
```

2. Order Processing System Code:

```
class Queue:
    """A simple queue implementation for managing order processing."""
    def __init__(self):
        self.queue = []
    def enqueue(self, item):
        """Add an order to the processing queue."""
        self.queue.append(item)
    def dequeue(self):
        """Remove and return the next order to be processed."""
        if not self.is_empty():
            return self.queue.pop(0)
        else:
            raise IndexError("Queue is empty")
    def peek(self):
        """Return the next order to be processed without removing it."""
        if not self.is_empty():
            return self.queue[0]
        else:
            raise IndexError("Queue is empty")
    def is_empty(self):
        """Check if the processing queue is empty."""
        return len(self.queue) == 0
# Example usage:
order_queue = Queue()
order_queue.enqueue("Order 1")
order_queue.enqueue("Order 2")
print(order_queue.peek())
print(order_queue.dequeue())
print(order_queue.is_empty())
print(order_queue.dequeue())
print(order_queue.is_empty())
```

Output:

```
Order 1
Order 1
False
Order 2
True
```

3. Top-Selling Products Tracker Code:

```python
import heapq
class PriorityQueue:
    """A priority queue implementation for tracking top-selling products."""
    def __init__(self):
        self.elements = []
    def push(self, item, priority):
        """Add a product to the tracker with a given sales count as priority."""
        heapq.heappush(self.elements, (priority, item))
    def pop(self):
        """Remove and return the product with the highest sales count."""
        return heapq.heappop(self.elements)[1]
    def peek(self):
        """Return the product with the highest sales count without removing it."""
        return self.elements[0][1] if self.elements else None
    def is_empty(self):
        """Check if the tracker is empty."""
        return not self.elements
# Example usage:
product_tracker = PriorityQueue()
product_tracker.push("Product A", 100)
product_tracker.push("Product B", 150)
print(product_tracker.peek())
print(product_tracker.pop())
print(product_tracker.is_empty())
print(product_tracker.pop())
print(product_tracker.is_empty())
```

Output:

```
Product A
Product A
False
Product B
True
```

4. Product Search Engine Code:

```python
class HashTable:
    """A simple hash table implementation for managing product search."""
    def __init__(self):
        self.table = {}
    def insert(self, key, value):
        """Insert a product into the search engine."""
        self.table[key] = value
    def search(self, key):
        """Search for a product by name."""
        return self.table.get(key, None)
    def delete(self, key):
        """Delete a product from the search engine."""
        if key in self.table:
            del self.table[key]
# Example usage:
search_engine = HashTable()
search_engine.insert("Product A", "Description of Product A")
search_engine.insert("Product B", "Description of Product B")
print(search_engine.search("Product A"))
print(search_engine.search("Product B"))
search_engine.delete("Product A")
print(search_engine.search("Product A"))
```

Output:

```
Description of Product A
Description of Product B
None
```

5. **Delivery Route Planning Code:**

```python
class Graph:
    """A graph implementation for managing delivery route planning."""
    def __init__(self):
        self.graph = {}
    def add_vertex(self, vertex):
        """Add a location to the delivery route graph."""
        if vertex not in self.graph:
            self.graph[vertex] = []
    def add_edge(self, vertex1, vertex2):
        """Add a route between two locations in the delivery graph."""
        if vertex1 in self.graph and vertex2 in self.graph:
            self.graph[vertex1].append(vertex2)
            self.graph[vertex2].append(vertex1)
    def display(self):
        """Display the delivery route graph as an adjacency list."""
        for vertex, edges in self.graph.items():
            print(f"{vertex}: {edges}")
# Example usage:
delivery_graph = Graph()
delivery_graph.add_vertex("Warehouse")
delivery_graph.add_vertex("Customer A")
delivery_graph.add_vertex("Customer B")
delivery_graph.add_edge("Warehouse", "Customer A")
delivery_graph.add_edge("Warehouse", "Customer B")
delivery_graph.display()
```

Output:

```
Warehouse: ['Customer A', 'Customer B']
Customer A: ['Warehouse']
Customer B: ['Warehouse']
```

Feature	Chosen Data Structure	Justification
Shopping Cart Management	Linked List	A linked list allows dynamic addition and removal of products without shifting elements like arrays. It is memory efficient and flexible for frequent insertions and deletions.
Order Processing System	Queue	Orders must be processed in First-In-First-Out (FIFO) order. A queue ensures fairness and processes orders exactly in the sequence they are placed.
Top-Selling Products Tracker	Priority Queue	A priority queue helps maintain products ranked by sales count. The product with the highest sales can be retrieved efficiently in O (log n) time.
Product Search Engine	Hash Table	A hash table enables fast lookup of products using product ID in O (1) average time. It is ideal for quick searching in large inventories.
Delivery Route Planning	Graph	Warehouses and delivery locations form nodes, and routes between them form edges. A graph structure supports shortest path and route optimization algorithms.