LAB 11.1

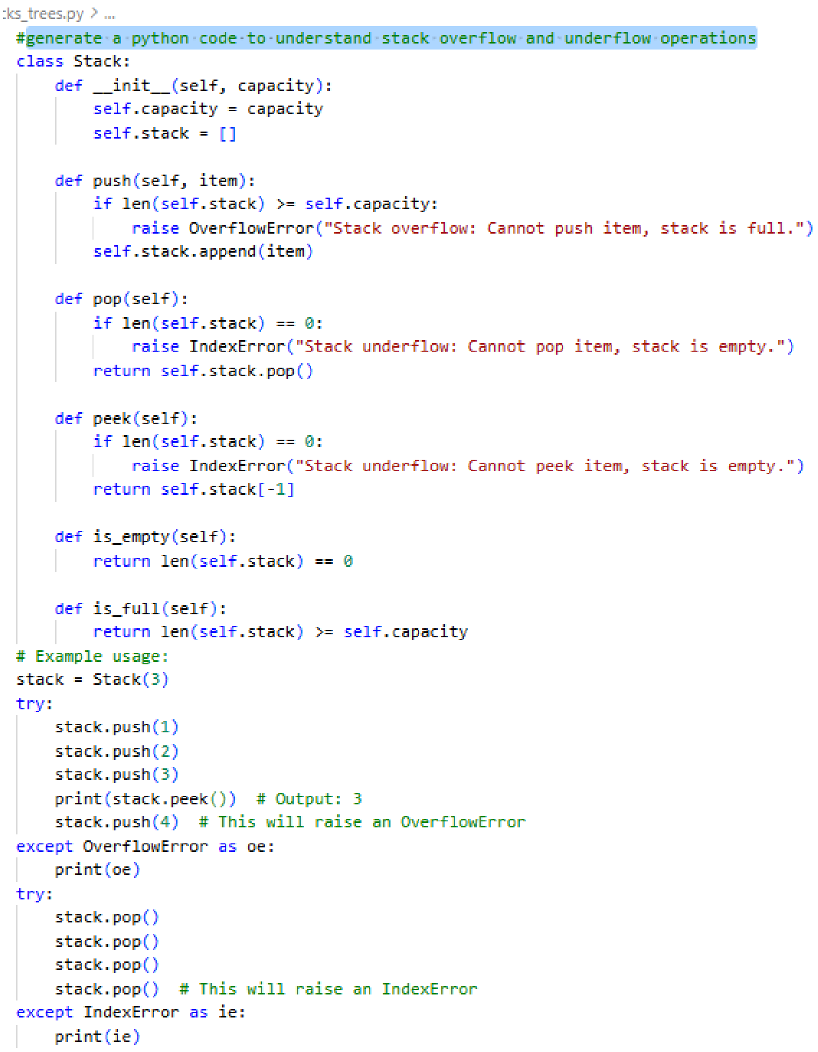
2303A51494

B27

Task: understand stack overflow and underflow operations

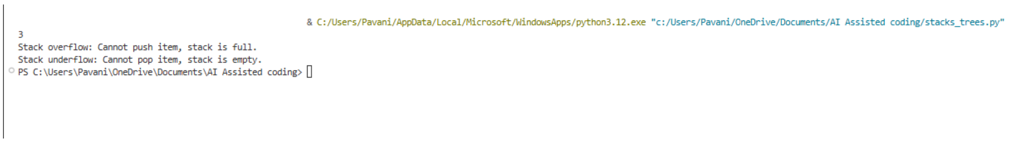
Prompt: generate a python code to understand stack overflow and underflow operations

CODE:



```python
#generate a python code to understand stack overflow and underflow operations
class Stack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = []

    def push(self, item):
        if len(self.stack) >= self.capacity:
            raise OverflowError("Stack overflow: Cannot push item, stack is full.")
        self.stack.append(item)

    def pop(self):
        if len(self.stack) == 0:
            raise IndexError("Stack underflow: Cannot pop item, stack is empty.")
        return self.stack.pop()

    def peek(self):
        if len(self.stack) == 0:
            raise IndexError("Stack underflow: Cannot peek item, stack is empty.")
        return self.stack[-1]

    def is_empty(self):
        return len(self.stack) == 0

    def is_full(self):
        return len(self.stack) >= self.capacity
# Example usage:
stack = Stack(3)
try:
    stack.push(1)
    stack.push(2)
    stack.push(3)
    print(stack.peek())  # Output: 3
    stack.push(4)  # This will raise an OverflowError
except OverflowError as oe:
    print(oe)
try:
    stack.pop()
    stack.pop()
    stack.pop()
    stack.pop()  # This will raise an IndexError
except IndexError as ie:
    print(ie)
```

OUTPUT:



```
                                                & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/stacks_trees.py"
3
Stack overflow: Cannot push item, stack is full.
Stack underflow: Cannot pop item, stack is empty.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> 
```

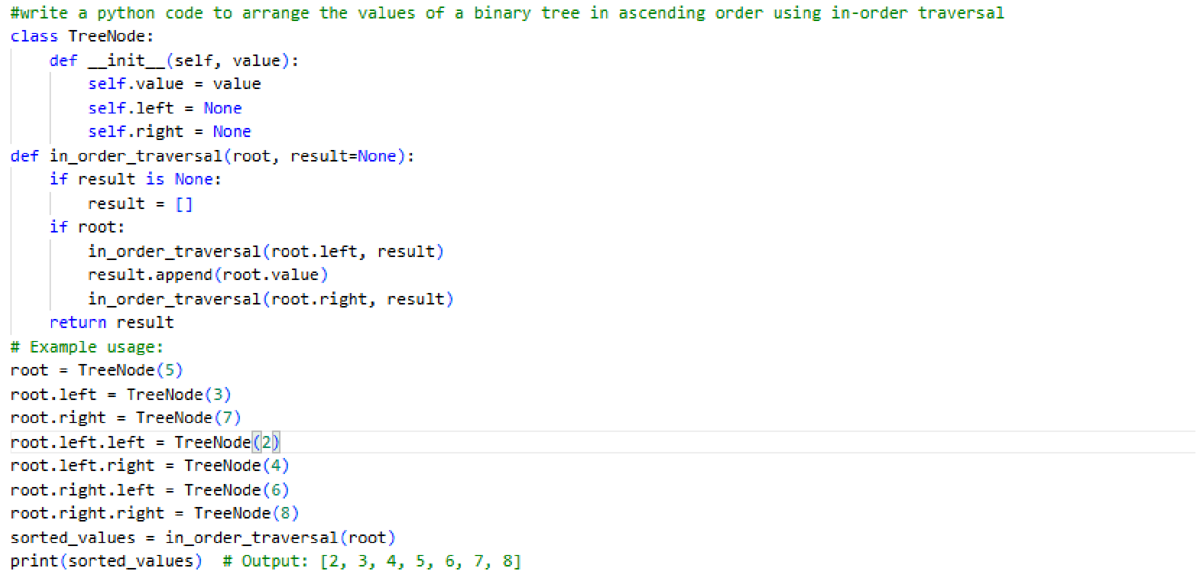
EXPLANATION:

- The Stack class implements a stack data structure with a specified capacity, allowing for push and pop operations while checking for overflow and underflow conditions.
- The push method adds an item to the stack if it is not full, while
- The pop method removes and returns the top item if the stack is not empty. The peek method allows you to see the top item without removing it, and the is_empty and is_full methods check the state of the stack.
- The example usage demonstrates how to use the Stack class and handles exceptions for stack overflow and underflow scenarios, providing clear error messages when these conditions occur.

TASK:Binary tree inorder traversal

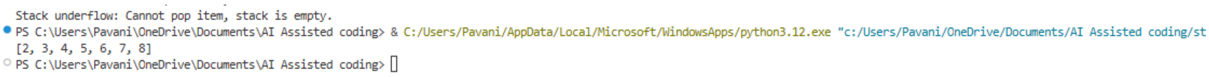
PROMPT: write a python code to arrange the values of a binary tree in ascending order using in-order traversal

CODE:



```python
#write a python code to arrange the values of a binary tree in ascending order using in-order traversal
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
def in_order_traversal(root, result=None):
    if result is None:
        result = []
    if root:
        in_order_traversal(root.left, result)
        result.append(root.value)
        in_order_traversal(root.right, result)
    return result
# Example usage:
root = TreeNode(5)
root.left = TreeNode(3)
root.right = TreeNode(7)
root.left.left = TreeNode(2)
root.left.right = TreeNode(4)
root.right.left = TreeNode(6)
root.right.right = TreeNode(8)
sorted_values = in_order_traversal(root)
print(sorted_values)  # Output: [2, 3, 4, 5, 6, 7, 8]
```

OUTPUT:



```
Stack underflow: Cannot pop item, stack is empty.
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/st
[2, 3, 4, 5, 6, 7, 8]
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> 
```

EXPLANATION:

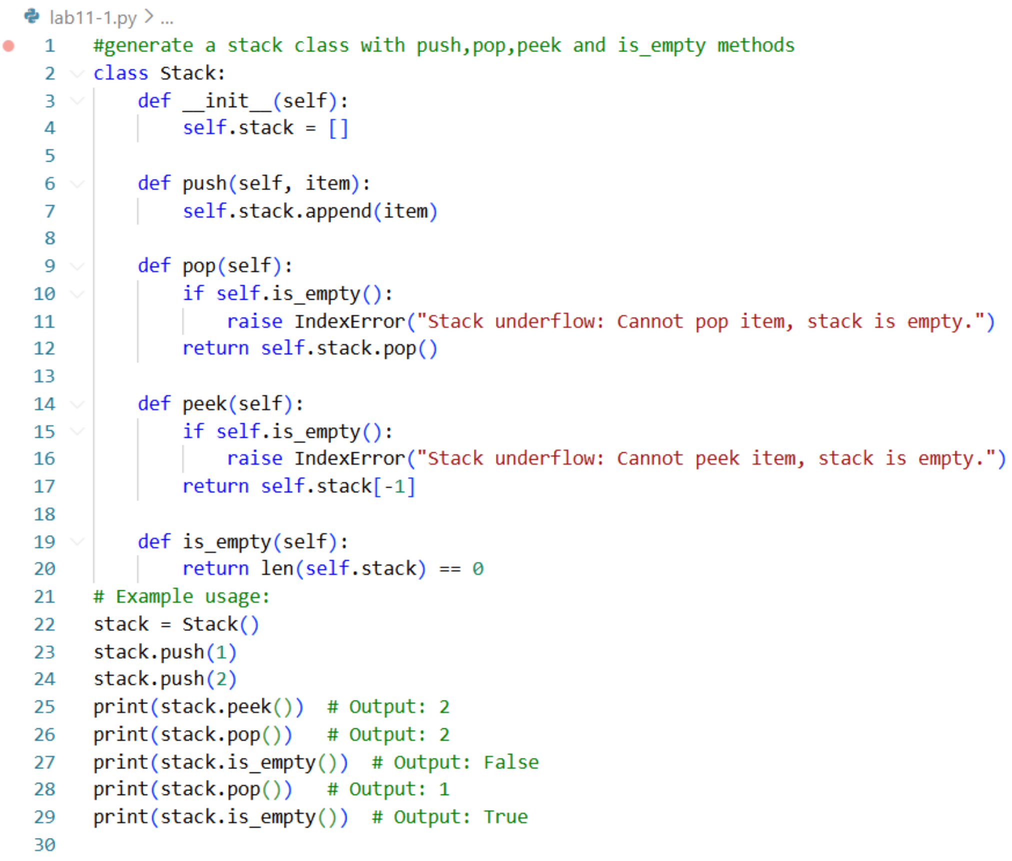
- The TreeNode class defines a node in a binary tree, with attributes for the node's value and pointers to its left and right children.
- The in_order_traversal function performs an in-order traversal of the binary tree

- It recursively visits the left subtree, appends the current node's value to the result list, and then visits the right subtree.
- This traversal method ensures that the values are collected in ascending order for a binary search tree.

Task-1: Use AI to generate a Stack class with push, pop, peek, and is_empty methods.

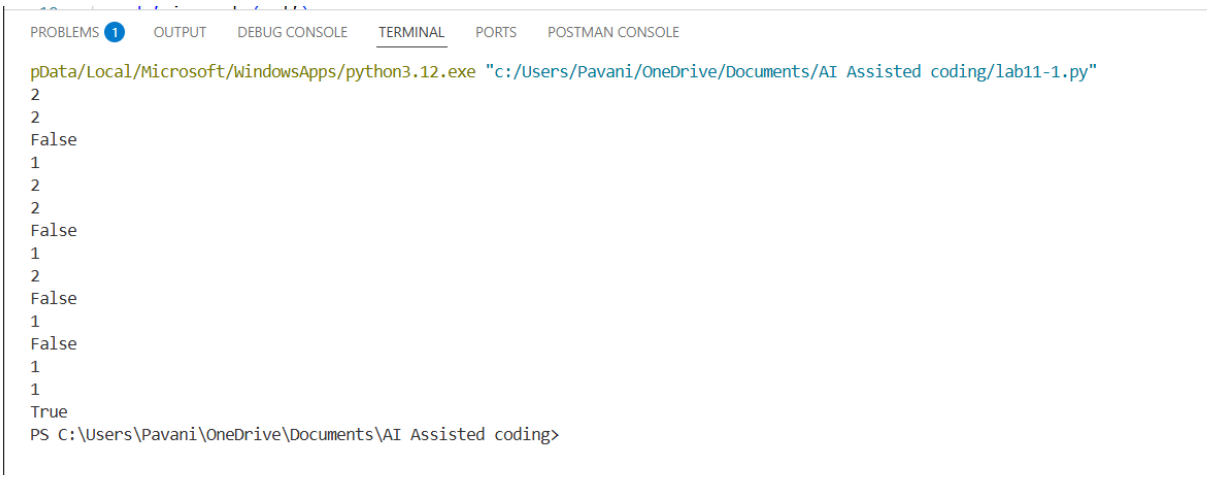
PROMPT:generate a stack class with push,pop,peek and is_empty methods

CODE:



```python
#generate a stack class with push,pop,peek and is_empty methods
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, item):
        self.stack.append(item)

    def pop(self):
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot pop item, stack is empty.")
        return self.stack.pop()

    def peek(self):
        if self.is_empty():
            raise IndexError("Stack underflow: Cannot peek item, stack is empty.")
        return self.stack[-1]

    def is_empty(self):
        return len(self.stack) == 0
# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
print(stack.peek())  # Output: 2
print(stack.pop())   # Output: 2
print(stack.is_empty())  # Output: False
print(stack.pop())   # Output: 1
print(stack.is_empty())  # Output: True

```

OUTPUT:



```
pData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab11-1.py"
2
2
False
1
2
2
False
1
2
False
1
False
1
1
True
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

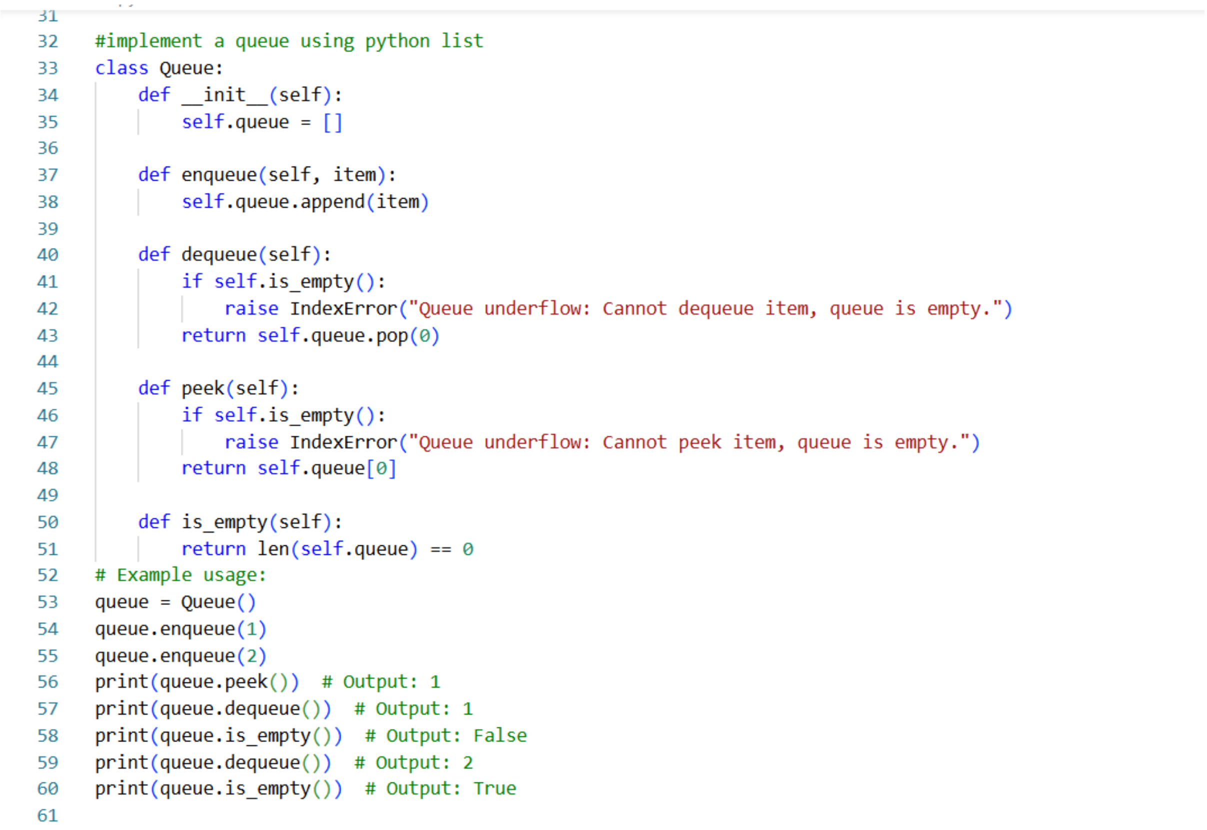
EXPLANATION:

- The Stack class uses a Python list to implement the stack data structure, allowing for dynamic resizing
- The push method adds an item to the top of the stack
- while the pop method removes and returns the top item, checking for stack underflow to prevent errors.
- The peek method allows you to see the top item without removing it, and the is_empty method checks if the stack is empty.

Task Description #2 – Queue Implementation

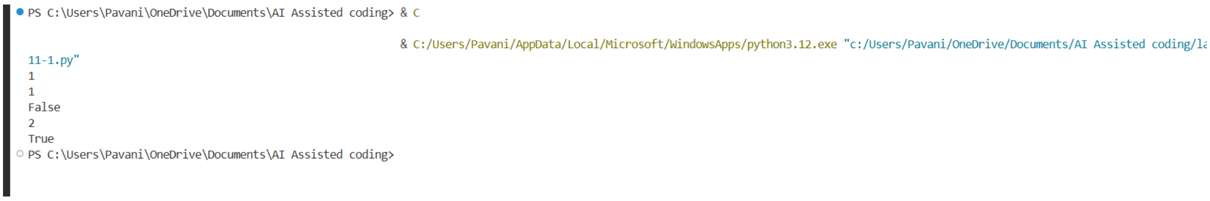
PROMPT:implement a Queue using Python lists.

CODE:



```
#implement a queue using python list
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot dequeue item, queue is empty.")
        return self.queue.pop(0)

    def peek(self):
        if self.is_empty():
            raise IndexError("Queue underflow: Cannot peek item, queue is empty.")
        return self.queue[0]

    def is_empty(self):
        return len(self.queue) == 0
# Example usage:
queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
print(queue.peek())  # Output: 1
print(queue.dequeue())  # Output: 1
print(queue.is_empty())  # Output: False
print(queue.dequeue())  # Output: 2
print(queue.is_empty())  # Output: True
```

OUTPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C

                                                   & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/la
11-1.py"
1
1
False
2
True
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

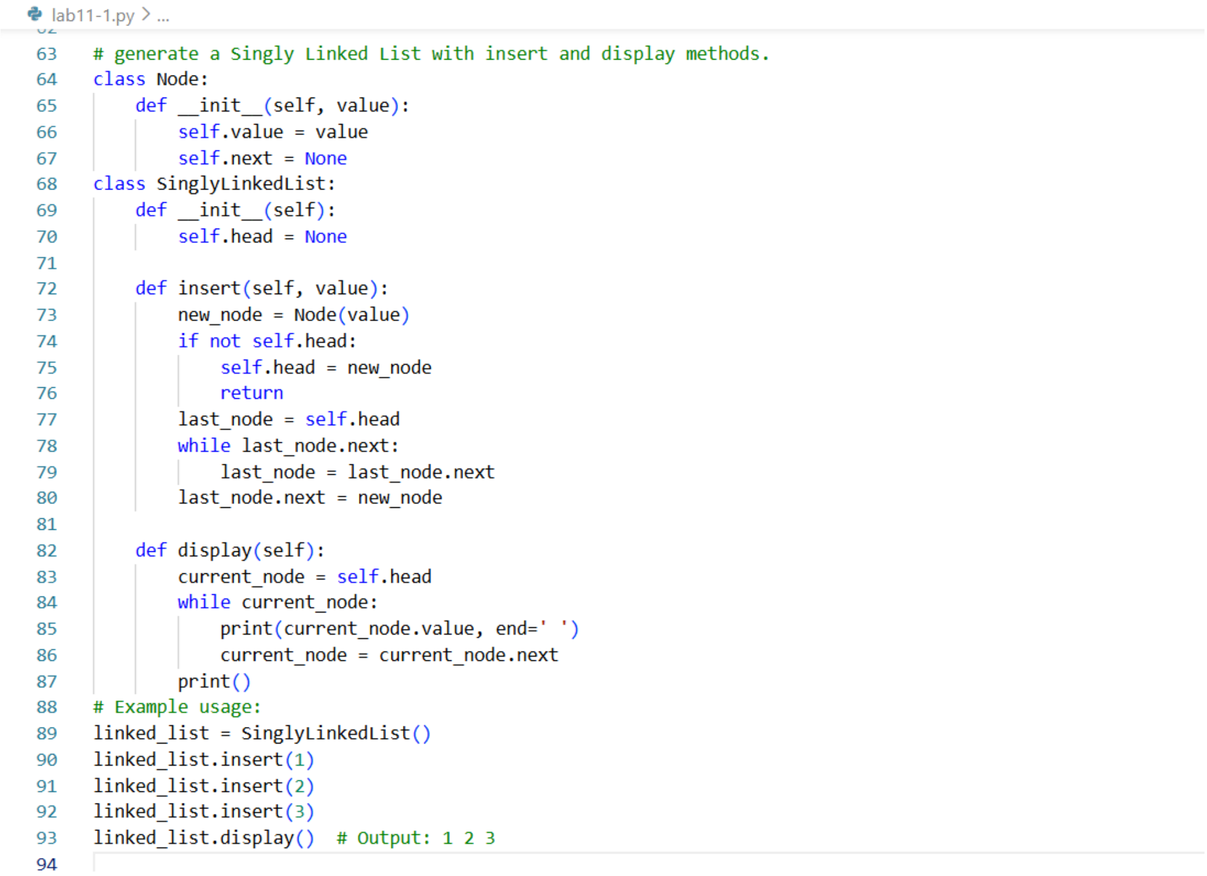
EXPLANATION:

- The Queue class uses a Python list to implement the queue data structure, allowing for dynamic resizing.
- The enqueue method adds an item to the end of the queue
- The deque method removes and returns the front item, checking for queue underflow to prevent errors.
- The peek method allows you to see the front item without removing it
- the is_empty method checks if the queue is empty.

Task Description #3 – Linked List

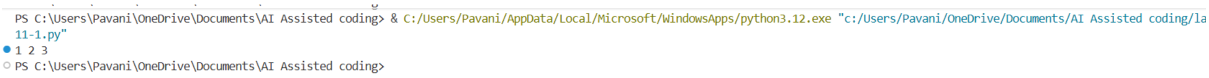
PROMPT: generate a Singly Linked List with insert and display methods.

CODE:



```python
# generate a Singly Linked List with insert and display methods.
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def insert(self, value):
        new_node = Node(value)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def display(self):
        current_node = self.head
        while current_node:
            print(current_node.value, end=' ')
            current_node = current_node.next
        print()
# Example usage:
linked_list = SinglyLinkedList()
linked_list.insert(1)
linked_list.insert(2)
linked_list.insert(3)
linked_list.display()  # Output: 1 2 3

```

OUTPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/la
11-1.py"
● 1 2 3
○ PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

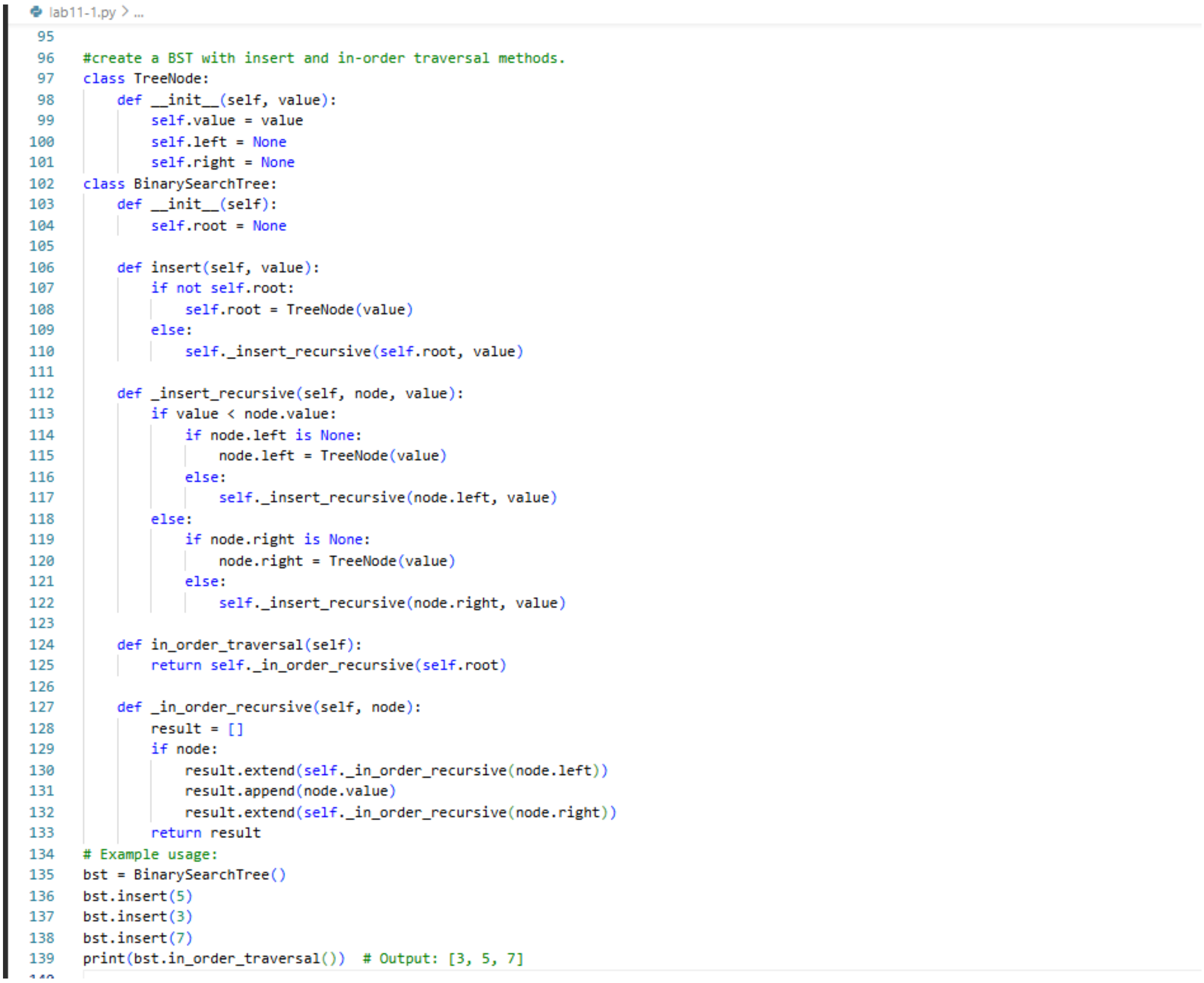
EXPLANATION:

- The Node class represents an individual node in the linked list, containing a value and a reference to the next node.
- The SinglyLinkedList class manages the linked list, allowing for insertion of new nodes at the end and displaying the list.
- The insert method creates a new node and appends it to the end of the list, while the display method traverses the list and prints the values of each node.

Task Description #4 – Binary Search Tree (BST)

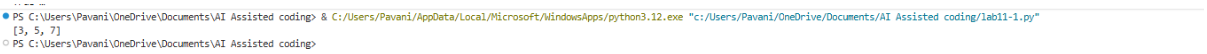
PROMPT: create a BST with insert and in-order traversal methods.

CODE:



```python

#create a BST with insert and in-order traversal methods.
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, value):
        if not self.root:
            self.root = TreeNode(value)
        else:
            self._insert_recursive(self.root, value)

    def _insert_recursive(self, node, value):
        if value < node.value:
            if node.left is None:
                node.left = TreeNode(value)
            else:
                self._insert_recursive(node.left, value)
        else:
            if node.right is None:
                node.right = TreeNode(value)
            else:
                self._insert_recursive(node.right, value)

    def in_order_traversal(self):
        return self._in_order_recursive(self.root)

    def _in_order_recursive(self, node):
        result = []
        if node:
            result.extend(self._in_order_recursive(node.left))
            result.append(node.value)
            result.extend(self._in_order_recursive(node.right))
        return result
# Example usage:
bst = BinarySearchTree()
bst.insert(5)
bst.insert(3)
bst.insert(7)
print(bst.in_order_traversal())  # Output: [3, 5, 7]
```

OUTPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab11-1.py"
[3, 5, 7]
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

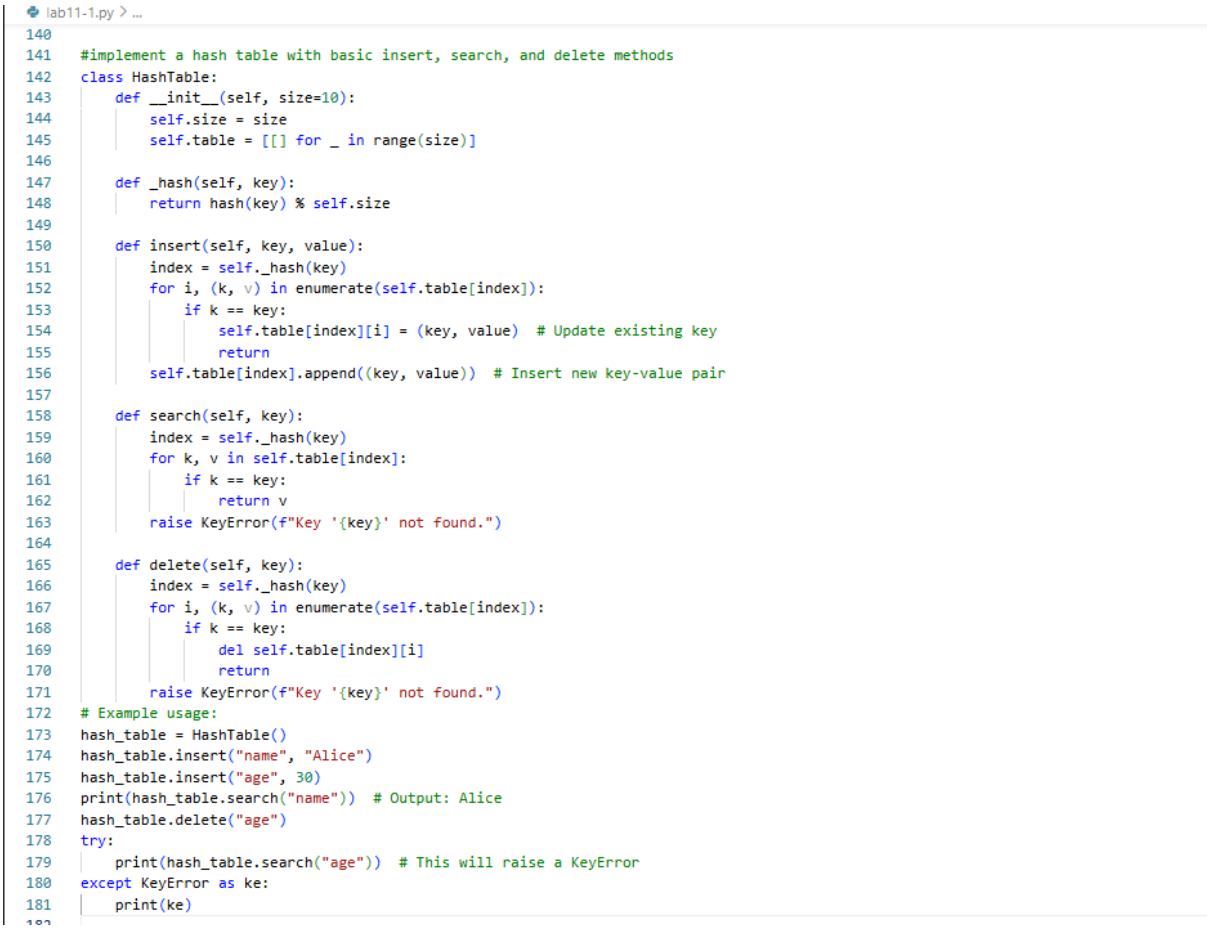
EXPLANATION:

- The TreeNode class represents a node in the binary search tree, containing a value and references to the left and right child nodes.
- The BinarySearchTree class manages the tree structure, allowing for insertion of values while maintaining the binary search property.
- The insert method adds values to the tree in the correct position
- while the in_order_traversal method returns a sorted list of values by traversing the tree in order (left, root, right)

Task Description #5 – Hash Table

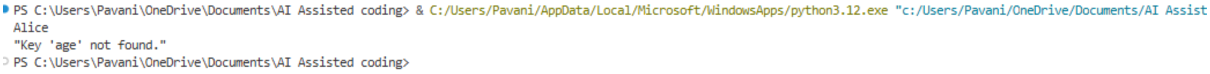
PROMPT: implement a hash table with basic insert, search, and delete methods

CODE:



```python

#implement a hash table with basic insert, search, and delete methods
class HashTable:
    def __init__(self, size=10):
        self.size = size
        self.table = [[] for _ in range(size)]

    def _hash(self, key):
        return hash(key) % self.size

    def insert(self, key, value):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                self.table[index][i] = (key, value)  # Update existing key
                return
        self.table[index].append((key, value))  # Insert new key-value pair

    def search(self, key):
        index = self._hash(key)
        for k, v in self.table[index]:
            if k == key:
                return v
        raise KeyError(f"Key '{key}' not found.")

    def delete(self, key):
        index = self._hash(key)
        for i, (k, v) in enumerate(self.table[index]):
            if k == key:
                del self.table[index][i]
                return
        raise KeyError(f"Key '{key}' not found.")
# Example usage:
hash_table = HashTable()
hash_table.insert("name", "Alice")
hash_table.insert("age", 30)
print(hash_table.search("name"))  # Output: Alice
hash_table.delete("age")
try:
    print(hash_table.search("age"))  # This will raise a KeyError
except KeyError as ke:
    print(ke)
```

OUTPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assist
Alice
"Key 'age' not found."
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

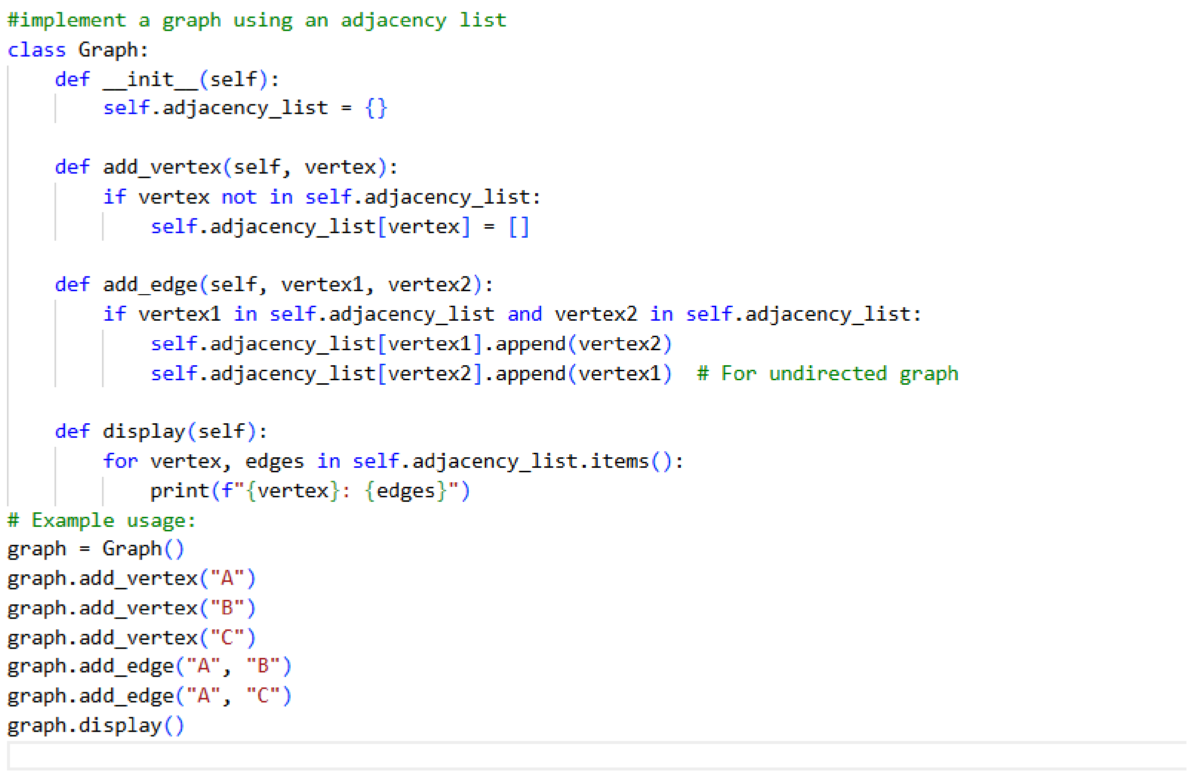
EXPLANATION:

- The HashTable class implements a hash table using separate chaining for collision resolution, where each bucket is a list that can store multiple key-value pairs.
- The _hash method computes the hash of a key and maps it to an index in the table, while the insert method adds or updates key-value pairs in the appropriate bucket.
- The search method looks for a key in the corresponding bucket and returns its value
- while the delete method removes a key-value pair from the bucket if it exists, otherwise it raises a KeyError.

Task Description #6 – Graph Representation

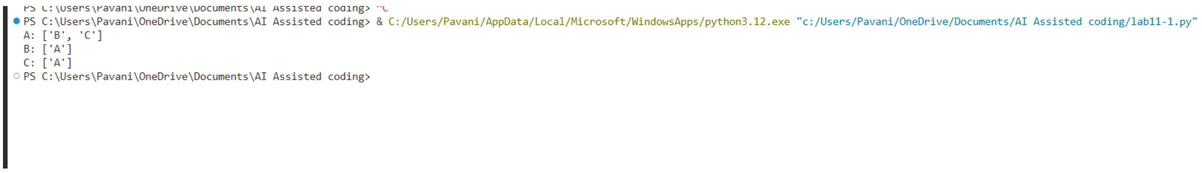
PROMPT: implement a graph using an adjacency list.

CODE:



```python
#implement a graph using an adjacency list
class Graph:
    def __init__(self):
        self.adjacency_list = {}

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

    def add_edge(self, vertex1, vertex2):
        if vertex1 in self.adjacency_list and vertex2 in self.adjacency_list:
            self.adjacency_list[vertex1].append(vertex2)
            self.adjacency_list[vertex2].append(vertex1)  # For undirected graph

    def display(self):
        for vertex, edges in self.adjacency_list.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:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> ^C
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab11-1.py"
A: ['B', 'C']
B: ['A']
C: ['A']
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

EXPLANATION:

The Graph class uses a dictionary to represent the adjacency list, where each key is a vertex and its value is a list of adjacent vertices.

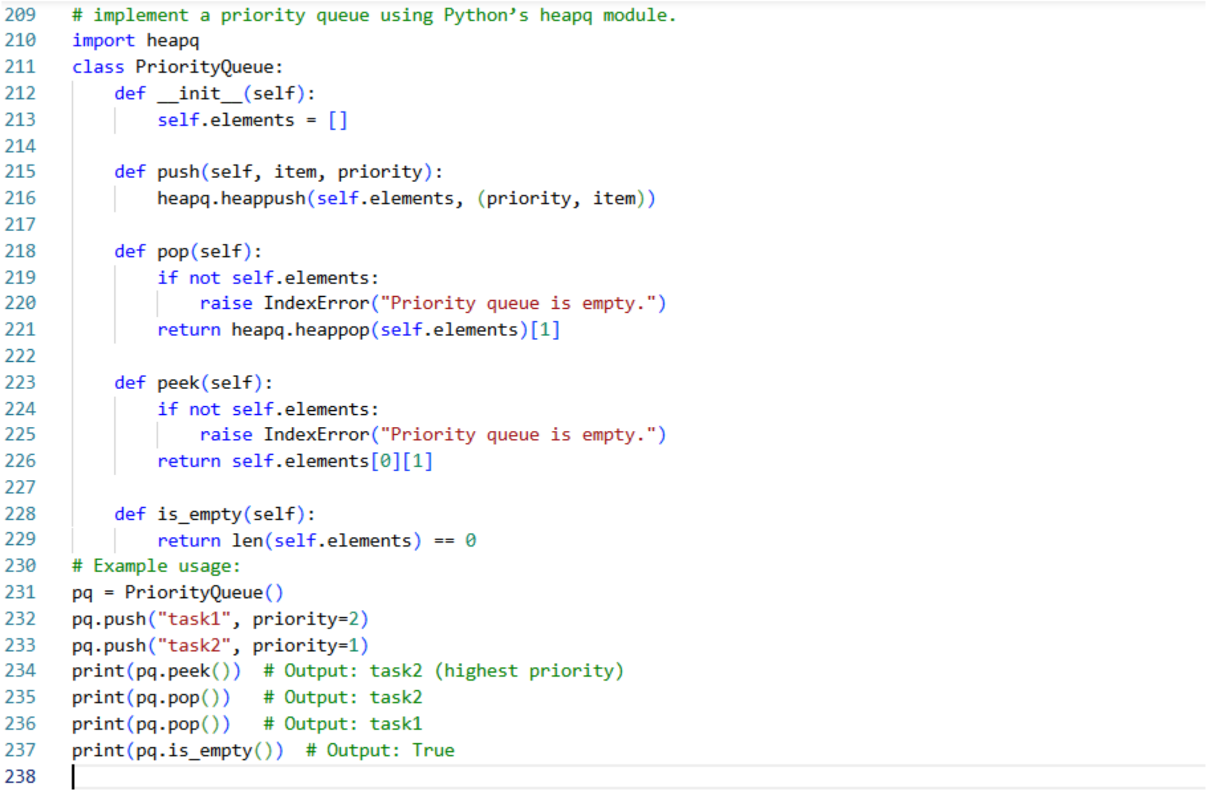
The add_vertex method adds a new vertex to the graph, while the add_edge method creates an undirected edge between two vertices by adding each vertex to the other's adjacency list.

The display method prints the adjacency list, showing each vertex and its connected vertices.

Task Description #7 – Priority Queue

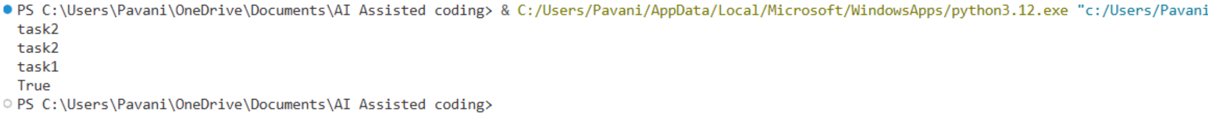
PROMPT: implement a priority queue using Python’s heapq module.

CODE:



```python
# implement a priority queue using Python’s heapq module.
import heapq
class PriorityQueue:
    def __init__(self):
        self.elements = []

    def push(self, item, priority):
        heapq.heappush(self.elements, (priority, item))

    def pop(self):
        if not self.elements:
            raise IndexError("Priority queue is empty.")
        return heapq.heappop(self.elements)[1]

    def peek(self):
        if not self.elements:
            raise IndexError("Priority queue is empty.")
        return self.elements[0][1]

    def is_empty(self):
        return len(self.elements) == 0
# Example usage:
pq = PriorityQueue()
pq.push("task1", priority=2)
pq.push("task2", priority=1)
print(pq.peek())  # Output: task2 (highest priority)
print(pq.pop())   # Output: task2
print(pq.pop())   # Output: task1
print(pq.is_empty())  # Output: True

```

OUTPUT:



```
● PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani
  task2
  task2
  task1
  True
○ PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

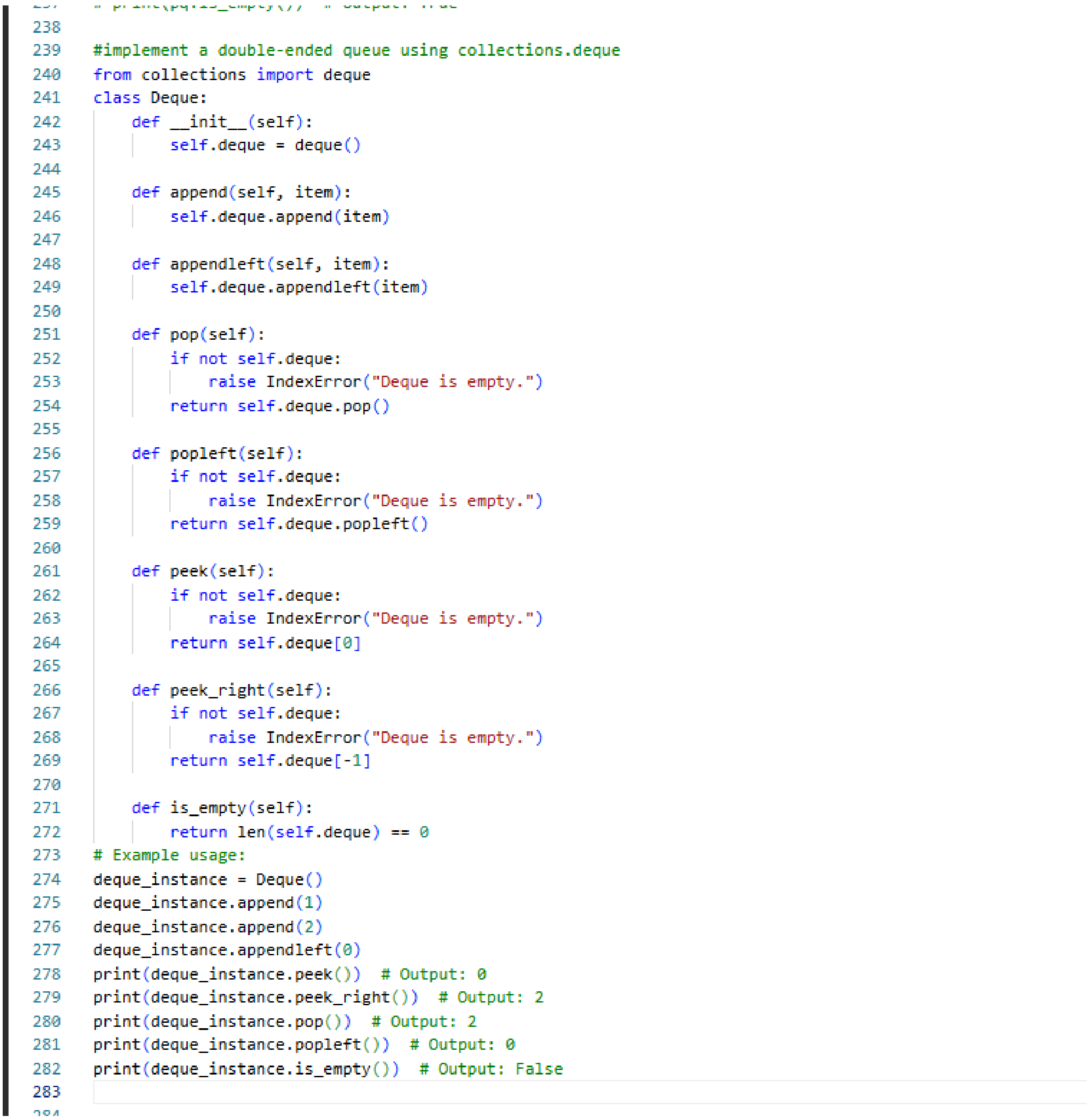
EXPLANATION:

- The PriorityQueue class uses a list to store elements as tuples of (priority, item), and the heapq module is used to maintain the heap property based on priority.
- The push method adds an item with its priority to the queue, while the pop method removes and returns the item with the highest priority (lowest priority number).
- The peek method allows you to see the item with the highest priority without removing it, and the is_empty method checks if the priority queue is empty.

Task Description #8 – Deque

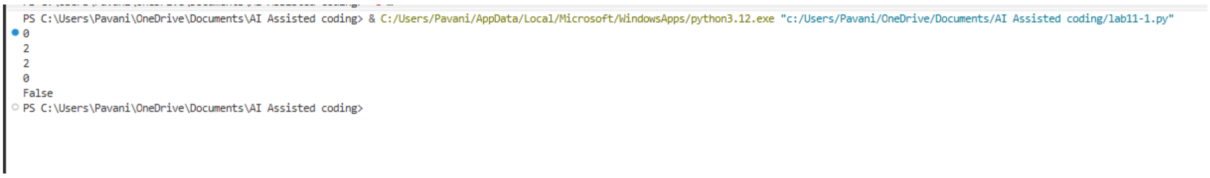
PROMPT: implement a double-ended queue using collections.deque.

CODE:



```python

#implement a double-ended queue using collections.deque
from collections import deque
class Deque:
    def __init__(self):
        self.deque = deque()

    def append(self, item):
        self.deque.append(item)

    def appendleft(self, item):
        self.deque.appendleft(item)

    def pop(self):
        if not self.deque:
            raise IndexError("Deque is empty.")
        return self.deque.pop()

    def popleft(self):
        if not self.deque:
            raise IndexError("Deque is empty.")
        return self.deque.popleft()

    def peek(self):
        if not self.deque:
            raise IndexError("Deque is empty.")
        return self.deque[0]

    def peek_right(self):
        if not self.deque:
            raise IndexError("Deque is empty.")
        return self.deque[-1]

    def is_empty(self):
        return len(self.deque) == 0
# Example usage:
deque_instance = Deque()
deque_instance.append(1)
deque_instance.append(2)
deque_instance.appendleft(0)
print(deque_instance.peek())  # Output: 0
print(deque_instance.peek_right())  # Output: 2
print(deque_instance.pop())  # Output: 2
print(deque_instance.popleft())  # Output: 0
print(deque_instance.is_empty())  # Output: False

```

OUTPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab11-1.py"
0
2
2
0
False
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

EXPLANATION:

- The Deque class uses the collections.deque data structure, which allows for efficient appending and popping from both ends of the queue.
- The append and appendleft methods add items to the right and left ends of the deque, respectively, while the pop and popleft methods remove items from the right and left ends.

- The peek and peek_right methods allow you to see the front and back items of the deque without removing them, and the is_empty method checks if the deque is empty.

Task Description #9 Real-Time Application Challenge – Choose the Right Data Structure

PROMPT:

generate a code to develop a campus resource management system that handles:

Student Attendance Tracking – Daily log of students entering/exiting the campus.

Event Registration System – Manage participants in events with quick search and removal.

Library Book Borrowing – Keep track of available books and their due dates.

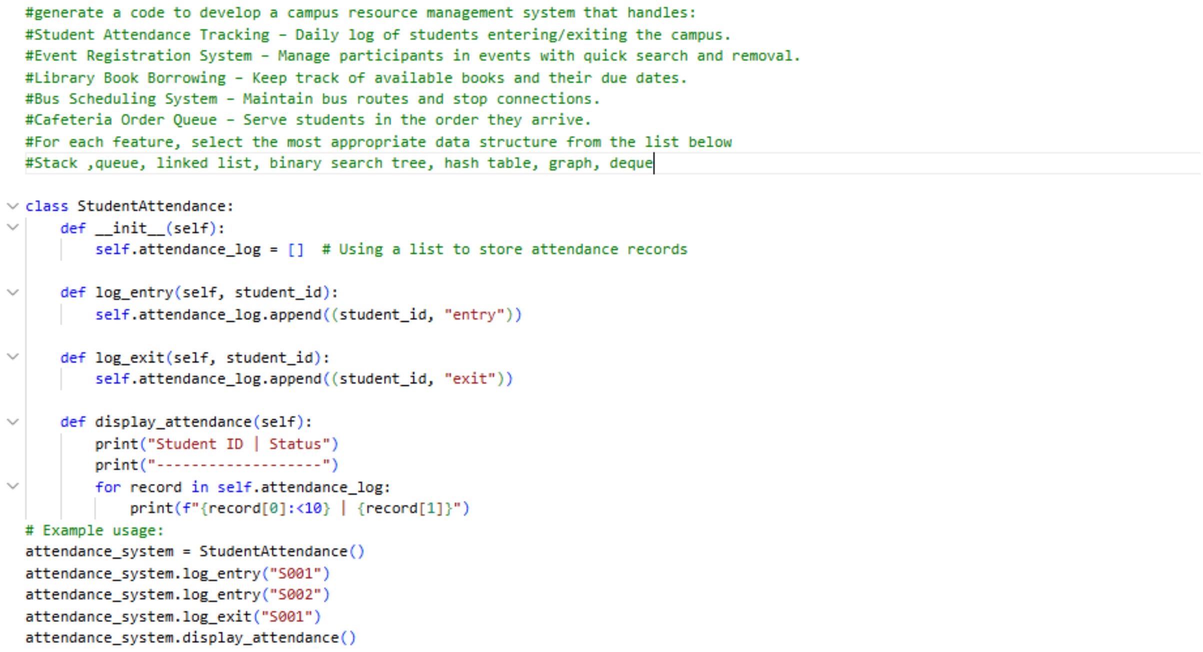
Bus Scheduling System – Maintain bus routes and stop connections.

Cafeteria Order Queue – Serve students in the order they arrive.

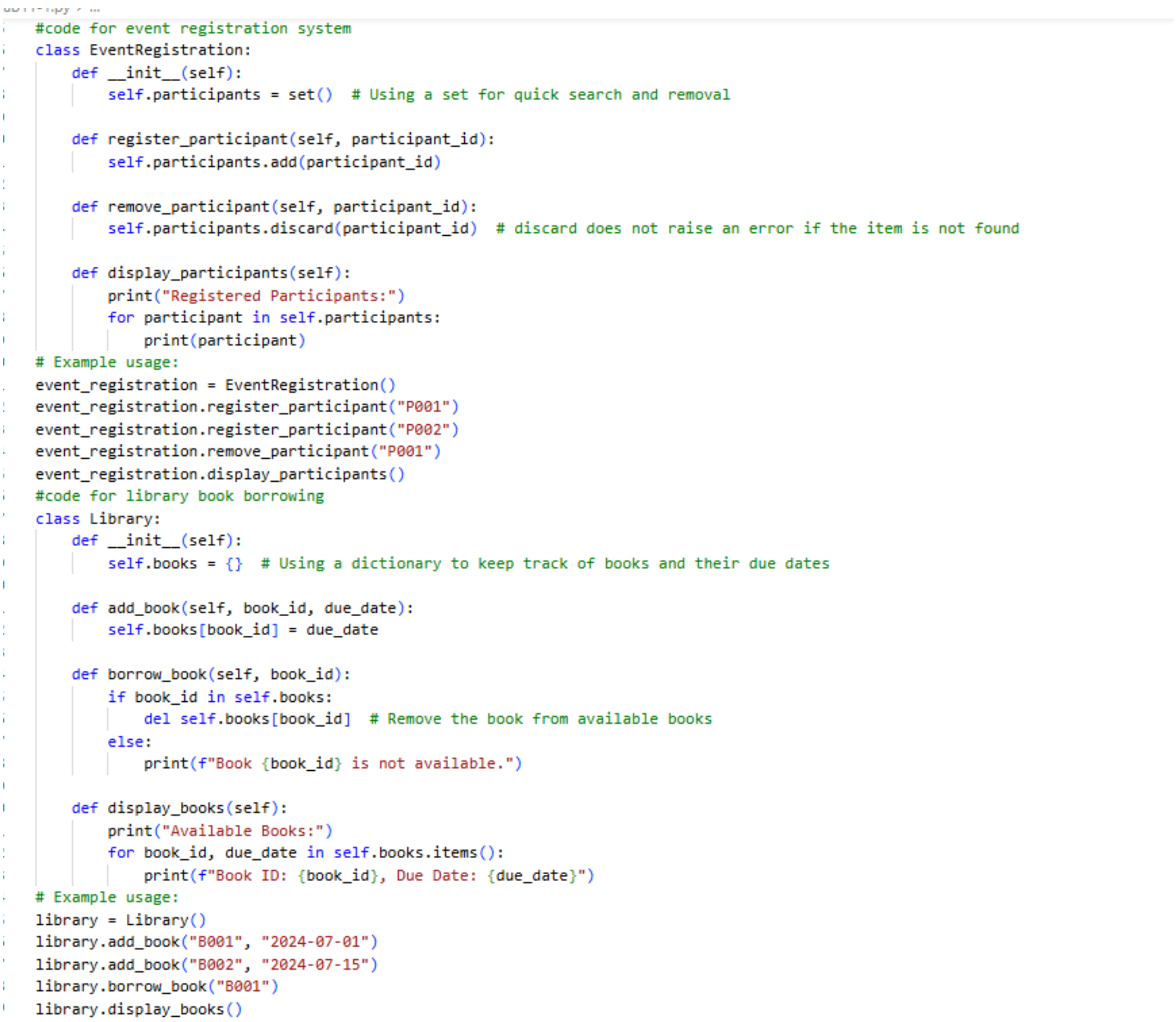
For each feature, select the most appropriate data structure from the list below

Stack ,queue, linked list, binary search tree, hash table, graph, deque

CODE:



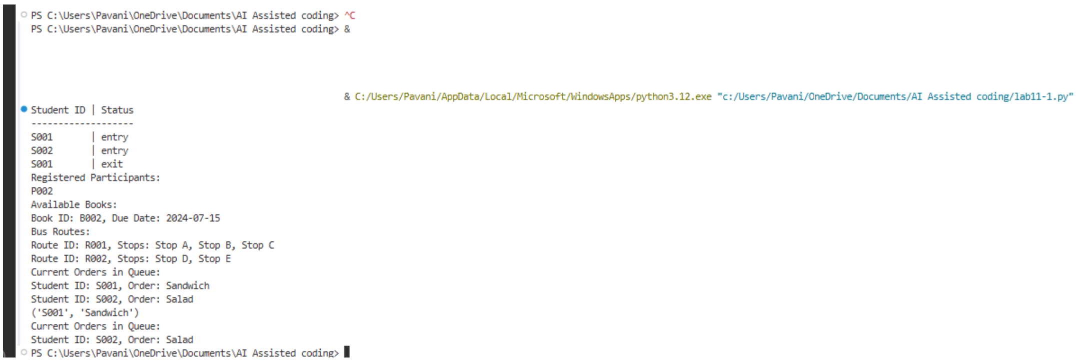
```
#generate a code to develop a campus resource management system that handles:
#Student Attendance Tracking - Daily log of students entering/exiting the campus.
#Event Registration System - Manage participants in events with quick search and removal.
#Library Book Borrowing - Keep track of available books and their due dates.
#Bus Scheduling System - Maintain bus routes and stop connections.
#Cafeteria Order Queue - Serve students in the order they arrive.
#For each feature, select the most appropriate data structure from the list below
#Stack ,queue, linked list, binary search tree, hash table, graph, deque

class StudentAttendance:
    def __init__(self):
        self.attendance_log = []  # Using a list to store attendance records

    def log_entry(self, student_id):
        self.attendance_log.append((student_id, "entry"))

    def log_exit(self, student_id):
        self.attendance_log.append((student_id, "exit"))

    def display_attendance(self):
        print("Student ID | Status")
        print("-------------------")
        for record in self.attendance_log:
            print(f"{record[0]:<10} | {record[1]}")
# Example usage:
attendance_system = StudentAttendance()
attendance_system.log_entry("S001")
attendance_system.log_entry("S002")
attendance_system.log_exit("S001")
attendance_system.display_attendance()
```



```python
#code for event registration system
class EventRegistration:
    def __init__(self):
        self.participants = set()  # Using a set for quick search and removal

    def register_participant(self, participant_id):
        self.participants.add(participant_id)

    def remove_participant(self, participant_id):
        self.participants.discard(participant_id)  # discard does not raise an error if the item is not found

    def display_participants(self):
        print("Registered Participants:")
        for participant in self.participants:
            print(participant)
# Example usage:
event_registration = EventRegistration()
event_registration.register_participant("P001")
event_registration.register_participant("P002")
event_registration.remove_participant("P001")
event_registration.display_participants()
#code for library book borrowing
class Library:
    def __init__(self):
        self.books = {}  # Using a dictionary to keep track of books and their due dates

    def add_book(self, book_id, due_date):
        self.books[book_id] = due_date

    def borrow_book(self, book_id):
        if book_id in self.books:
            del self.books[book_id]  # Remove the book from available books
        else:
            print(f"Book {book_id} is not available.")

    def display_books(self):
        print("Available Books:")
        for book_id, due_date in self.books.items():
            print(f"Book ID: {book_id}, Due Date: {due_date}")
# Example usage:
library = Library()
library.add_book("B001", "2024-07-01")
library.add_book("B002", "2024-07-15")
library.borrow_book("B001")
library.display_books()
```

```python
#code for bus scheduling system
class BusSchedule:
    def __init__(self):
        self.routes = {}  # Using a dictionary to maintain bus routes and stop connections

    def add_route(self, route_id, stops):
        self.routes[route_id] = stops

    def display_routes(self):
        print("Bus Routes:")
        for route_id, stops in self.routes.items():
            print(f"Route ID: {route_id}, Stops: {', '.join(stops)}")
# Example usage:
bus_schedule = BusSchedule()
bus_schedule.add_route("R001", ["Stop A", "Stop B", "Stop C"])
bus_schedule.add_route("R002", ["Stop D", "Stop E"])
bus_schedule.display_routes()
#code for cafeteria order queue
from collections import deque
class CafeteriaOrderQueue:
    def __init__(self):
        self.order_queue = deque()  # Using a deque to serve students in the order they arrive

    def place_order(self, student_id, order):
        self.order_queue.append((student_id, order))

    def serve_order(self):
        if self.order_queue:
            return self.order_queue.popleft()  # Serve the next order in line
        else:
            print("No orders to serve.")

    def display_orders(self):
        print("Current Orders in Queue:")
        for student_id, order in self.order_queue:
            print(f"Student ID: {student_id}, Order: {order}")
# Example usage:
cafeteria_queue = CafeteriaOrderQueue()
cafeteria_queue.place_order("S001", "Sandwich")
cafeteria_queue.place_order("S002", "Salad")
cafeteria_queue.display_orders()
print(cafeteria_queue.serve_order())  # Output: ('S001', 'Sandwich')
cafeteria_queue.display_orders()
```

OUTPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> ^C
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> &



                                                  & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab11-1.py"
Student ID | Status
-------------------
S001       | entry
S002       | entry
S001       | exit
Registered Participants:
P002
Available Books:
Book ID: B002, Due Date: 2024-07-15
Bus Routes:
Route ID: R001, Stops: Stop A, Stop B, Stop C
Route ID: R002, Stops: Stop D, Stop E
Current Orders in Queue:
Student ID: S001, Order: Sandwich
Student ID: S002, Order: Salad
('S001', 'Sandwich')
Current Orders in Queue:
Student ID: S002, Order: Salad
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding>
```

EXPLANATION:

- Each class executes a specific function of the campus resourece management system
- The StudentAttendance class manages the attendance log of students entering and exiting the campus using a list to store records.

- The EventRegistration class uses a set to manage event participants, allowing for quick registration, removal, and display of participants.
- The Library class maintains a dictionary to track available books and their due dates, while
- The BusSchedule class uses a dictionary to manage bus routes and their stops, and the CafeteriaOrderQueue class utilizes a deque to serve students in the order they arrive.

Task 10: Smart E-Commerce Platform – Data Structure Challenge

PROMPT:

write a python code to build a smart online shopping system that includes:

Shopping Cart Management – Add and remove products dynamically

Order Processing System – Orders processed in the order they are placed

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

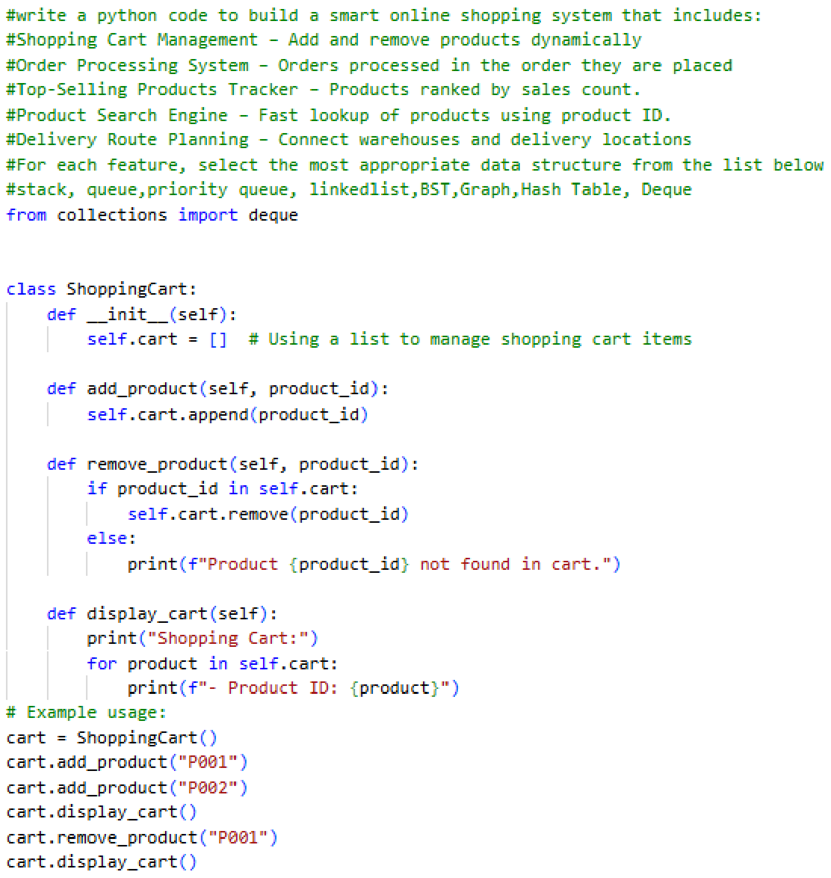
Product Search Engine – Fast lookup of products using product ID.

Delivery Route Planning – Connect warehouses and delivery locations

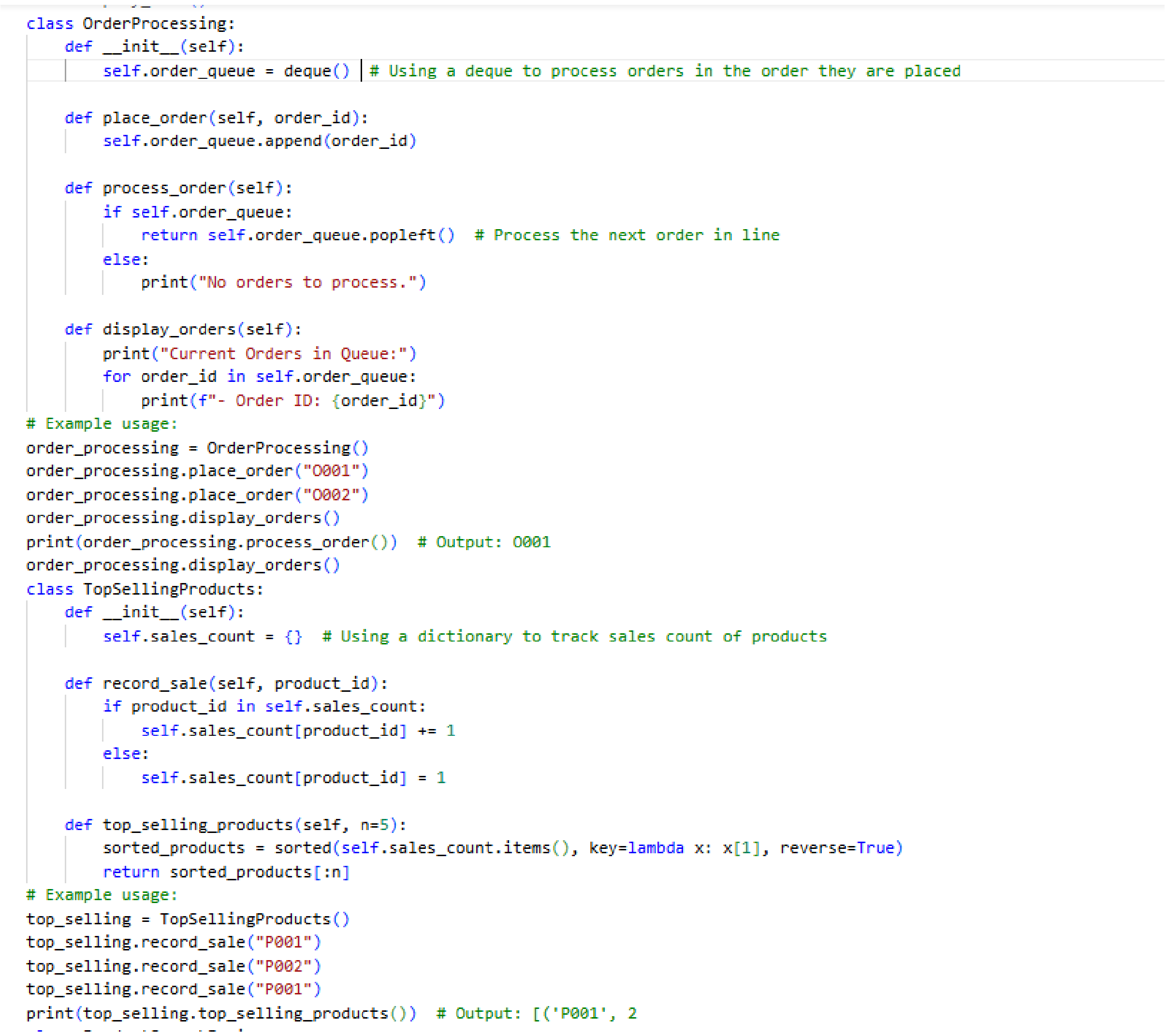
For each feature, select the most appropriate data structure from the list below

stack, queue,priority queue, linkedlist,BST,Graph,Hash Table, Deque

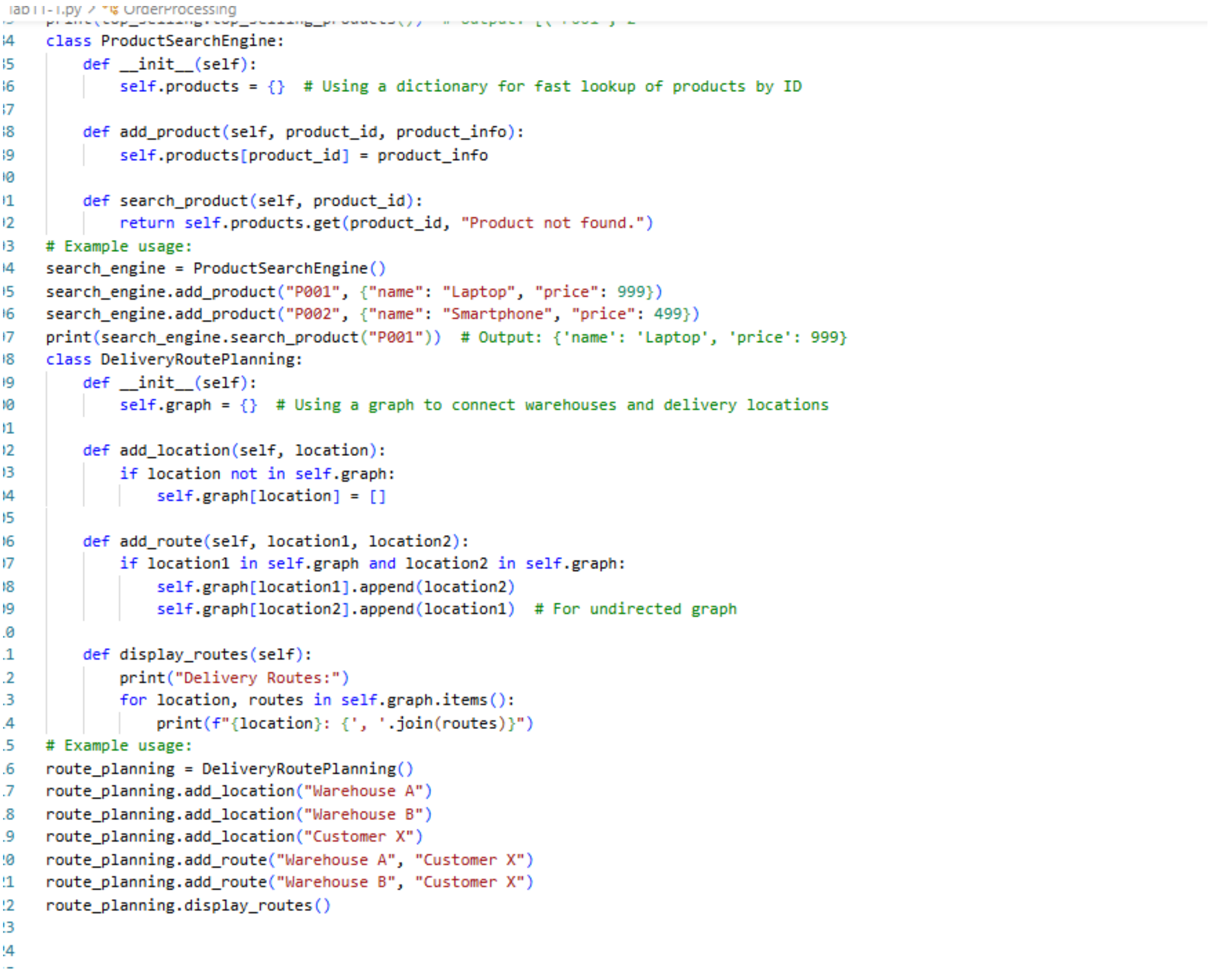
CODE:



```python
#write a python code to build a smart online shopping system that includes:
#Shopping Cart Management - Add and remove products dynamically
#Order Processing System - Orders processed in the order they are placed
#Top-Selling Products Tracker - Products ranked by sales count.
#Product Search Engine - Fast lookup of products using product ID.
#Delivery Route Planning - Connect warehouses and delivery locations
#For each feature, select the most appropriate data structure from the list below
#stack, queue,priority queue, linkedlist,BST,Graph,Hash Table, Deque
from collections import deque


class ShoppingCart:
    def __init__(self):
        self.cart = []  # Using a list to manage shopping cart items

    def add_product(self, product_id):
        self.cart.append(product_id)

    def remove_product(self, product_id):
        if product_id in self.cart:
            self.cart.remove(product_id)
        else:
            print(f"Product {product_id} not found in cart.")

    def display_cart(self):
        print("Shopping Cart:")
        for product in self.cart:
            print(f"- Product ID: {product}")
# Example usage:
cart = ShoppingCart()
cart.add_product("P001")
cart.add_product("P002")
cart.display_cart()
cart.remove_product("P001")
cart.display_cart()
```

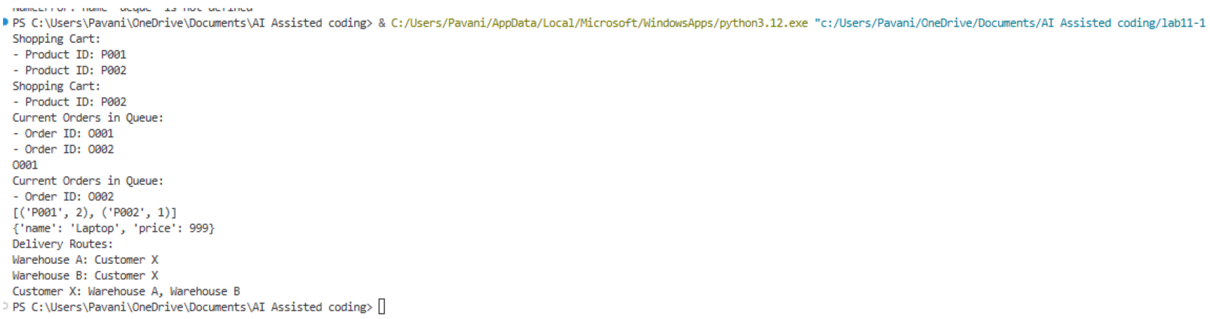


```
class OrderProcessing:
    def __init__(self):
        self.order_queue = deque() # Using a deque to process orders in the order they are placed

    def place_order(self, order_id):
        self.order_queue.append(order_id)

    def process_order(self):
        if self.order_queue:
            return self.order_queue.popleft()  # Process the next order in line
        else:
            print("No orders to process.")

    def display_orders(self):
        print("Current Orders in Queue:")
        for order_id in self.order_queue:
            print(f"- Order ID: {order_id}")
# Example usage:
order_processing = OrderProcessing()
order_processing.place_order("O001")
order_processing.place_order("O002")
order_processing.display_orders()
print(order_processing.process_order())  # Output: O001
order_processing.display_orders()
class TopSellingProducts:
    def __init__(self):
        self.sales_count = {}  # Using a dictionary to track sales count of products

    def record_sale(self, product_id):
        if product_id in self.sales_count:
            self.sales_count[product_id] += 1
        else:
            self.sales_count[product_id] = 1

    def top_selling_products(self, n=5):
        sorted_products = sorted(self.sales_count.items(), key=lambda x: x[1], reverse=True)
        return sorted_products[:n]
# Example usage:
top_selling = TopSellingProducts()
top_selling.record_sale("P001")
top_selling.record_sale("P002")
top_selling.record_sale("P001")
print(top_selling.top_selling_products())  # Output: [('P001', 2
```



```python
class ProductSearchEngine:
    def __init__(self):
        self.products = {}  # Using a dictionary for fast lookup of products by ID

    def add_product(self, product_id, product_info):
        self.products[product_id] = product_info

    def search_product(self, product_id):
        return self.products.get(product_id, "Product not found.")
# Example usage:
search_engine = ProductSearchEngine()
search_engine.add_product("P001", {"name": "Laptop", "price": 999})
search_engine.add_product("P002", {"name": "Smartphone", "price": 499})
print(search_engine.search_product("P001"))  # Output: {'name': 'Laptop', 'price': 999}
class DeliveryRoutePlanning:
    def __init__(self):
        self.graph = {}  # Using a graph to connect warehouses and delivery locations

    def add_location(self, location):
        if location not in self.graph:
            self.graph[location] = []

    def add_route(self, location1, location2):
        if location1 in self.graph and location2 in self.graph:
            self.graph[location1].append(location2)
            self.graph[location2].append(location1)  # For undirected graph

    def display_routes(self):
        print("Delivery Routes:")
        for location, routes in self.graph.items():
            print(f"{location}: {', '.join(routes)}")
# Example usage:
route_planning = DeliveryRoutePlanning()
route_planning.add_location("Warehouse A")
route_planning.add_location("Warehouse B")
route_planning.add_location("Customer X")
route_planning.add_route("Warehouse A", "Customer X")
route_planning.add_route("Warehouse B", "Customer X")
route_planning.display_routes()
```

OUPUT:



```
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> & C:/Users/Pavani/AppData/Local/Microsoft/WindowsApps/python3.12.exe "c:/Users/Pavani/OneDrive/Documents/AI Assisted coding/lab11-1
Shopping Cart:
- Product ID: P001
- Product ID: P002
Shopping Cart:
- Product ID: P002
Current Orders in Queue:
- Order ID: O001
- Order ID: O002
O001
Current Orders in Queue:
- Order ID: O002
[('P001', 2), ('P002', 1)]
{'name': 'Laptop', 'price': 999}
Delivery Routes:
Warehouse A: Customer X
Warehouse B: Customer X
Customer X: Warehouse A, Warehouse B
PS C:\Users\Pavani\OneDrive\Documents\AI Assisted coding> []
```

Explanation:

Each class executes a specific function of the smart online shopping system

The ShoppingCart class manages the shopping cart using a list to add and remove products dynamically,

while the OrderProcessing class uses a deque to process orders in the order they are placed.

The TopSellingProducts class maintains a dictionary to track sales count of products and provides a method to retrieve the top-selling products, while the ProductSearchEngine class uses a dictionary for fast lookup of products by their ID.

The DeliveryRoutePlanning class implements a graph to connect warehouses and delivery locations, allowing for the management of delivery routes.