ASSIGNMENT-11.5

NAME: K. NAVYA SRI
ROLL NO: 2503A52L12
Batch-16

Task 1: Smart Contact Manager

Part A: Using Arrays (Python List)

Task 2: Emergency Help Desk (Stack Implementation)

Prompt:

Write Python code to implement a stack class with push, pop, and peek methods.

Show how to simulate 5 IT help desk tickets being pushed and popped from the stack.

Add extra stack methods like is_empty() and size().

Show example output after adding and resolving tickets.

CODE:

```python
# Stack implementation for Emergency Help Desk

class Stack:
    def __init__(self):
        self.items = []

    # Push a new ticket
    def push(self, ticket):
        self.items.append(ticket)

    # Pop the latest ticket
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        return "No tickets to resolve"

    # Peek the latest ticket without removing
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        return "No tickets"

    # Check if stack is empty
    def is_empty(self):
        return len(self.items) == 0

    # Get current stack size
    def size(self):
        return len(self.items)
```

```python
# Example usage
helpdesk = Stack()

# Simulate 5 tickets arriving
helpdesk.push("Ticket 1: Internet issue")
helpdesk.push("Ticket 2: Printer not working")
helpdesk.push("Ticket 3: Software installation")
helpdesk.push("Ticket 4: Email problem")
helpdesk.push("Ticket 5: Computer crash")

print("Current top ticket:", helpdesk.peek())
print("Total tickets:", helpdesk.size())

# Resolving tickets (LIFO order)
print("Resolved:", helpdesk.pop())
print("Resolved:", helpdesk.pop())
print("Remaining tickets:", helpdesk.size())
print("Current top ticket:", helpdesk.peek())
```

OUTPUT:

```
Current top ticket: Ticket 3: Software installation
PS C:\Users\91901\Desktop\TRAINING CLASSES\23-JUNE-2025 PYTHON TRAINING CLASSES\DAY-1\AI assist coding> & C:/Users/91901/AppData/Local/Programs/Python/Python312/python.exe "c:/Users/91901/Desktop/TRAINING CLASSES/23-JUNE-2025 PYTHON TRAINING CLASSES/DAY-1/AI assist coding/11.5 assignment"
Current top ticket: Ticket 5: Computer crash
Total tickets: 5
Resolved: Ticket 5: Computer crash
Resolved: Ticket 4: Email problem
Remaining tickets: 3
Current top ticket: Ticket 3: Software installation
PS C:\Users\91901\Desktop\TRAINING CLASSES\23-JUNE-2025 PYTHON TRAINING CLASSES\DAY-1\AI assist coding>
```

OBSERVATION:

Stack follows LIFO (Last In, First Out):

The last ticket received is the first to be resolved.

Operations:

push() → O(1) (just add to end).

pop() → O(1) (remove last).

peek() → O(1) (check last).

Advantages:

Easy to manage urgent issues in reverse order.

Very efficient for insert/remove operations.

Limitation:

You cannot directly access middle tickets without popping others.

Task 3: Library Book Search (Queues & Priority Queues)

Prompt:

Write Python code for a queue class with enqueue and dequeue methods (FIFO).

Simulate students requesting books using a queue.

Extend the queue into a priority queue where faculty requests get higher priority than student requests.

Show example usage with a mix of student and faculty requests.

CODE:

```python
class Queue:
    def __init__(self):
        self.items = []

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

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        return None

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

    def __str__(self):
        return str(self.items)

# Simulate students requesting books
print("=== Student Book Requests (FIFO Queue) ===")
student_queue = Queue()
student_queue.enqueue("Student1 requests BookA")
student_queue.enqueue("Student2 requests BookB")
student_queue.enqueue("Student3 requests BookC")

while not student_queue.is_empty():
    print(student_queue.dequeue())

# Priority Queue: Faculty > Student
class PriorityQueue:
    def __init__(self):
        self.faculty_queue = []
        self.student_queue = []

```

```python
    def enqueue(self, requester_type, request):
        if requester_type == "faculty":
            self.faculty_queue.append(request)
        else:
            self.student_queue.append(request)

    def dequeue(self):
        if self.faculty_queue:
            return self.faculty_queue.pop(0)
        elif self.student_queue:
            return self.student_queue.pop(0)
        return None

    def is_empty(self):
        return not (self.faculty_queue or self.student_queue)

    def __str__(self):
        return f"Faculty: {self.faculty_queue}, Students: {self.student_queue}"

# Example usage with mixed requests
print("\n=== Mixed Requests (Priority Queue) ===")
pq = PriorityQueue()
pq.enqueue("student", "Student1 requests BookA")
pq.enqueue("faculty", "Faculty1 requests BookX")
pq.enqueue("student", "Student2 requests BookB")
pq.enqueue("faculty", "Faculty2 requests BookY")
pq.enqueue("student", "Student3 requests BookC")

while not pq.is_empty():
    print(pq.dequeue())
```

OUTPUT:

```
----- Normal Queue (FIFO) -----
All Requests: ['Book Request: Student A', 'Book Request: Student B', 'Book Request: Student C']
Serving: Book Request: Student A
Remaining: ['Book Request: Student B', 'Book Request: Student C']

----- Priority Queue (Faculty First) -----
All Requests: [('Book Request: Faculty X', 'faculty'), ('Book Request: Faculty Y', 'faculty'), ('Book Request: Student A', 'student'), ('Book Request: Student B
', 'student')]
Serving: ('Book Request: Faculty X', 'faculty')
Serving: ('Book Request: Faculty Y', 'faculty')
Remaining: [('Book Request: Student A', 'student'), ('Book Request: Student B', 'student')]
PS C:\Users\91901\Desktop\TRAINING CLASSES\23-JUNE-2025 PYTHON TRAINING CLASSES\DAY-1\AI assist coding> & C:/Users/91901/AppData/Local/Programs/Python/Python31
/python.exe "c:/Users/91901/Desktop/TRAINING CLASSES/23-JUNE-2025 PYTHON TRAINING CLASSES/DAY-1/AI assist coding/11.5 assignment"
----- Normal Queue (FIFO) -----
All Requests: ['Book Request: Student A', 'Book Request: Student B', 'Book Request: Student C']
Serving: Book Request: Student A
Remaining: ['Book Request: Student B', 'Book Request: Student C']

----- Priority Queue (Faculty First) -----
All Requests: [('Book Request: Faculty X', 'faculty'), ('Book Request: Faculty Y', 'faculty'), ('Book Request: Student A', 'student'), ('Book Request: Student B
', 'student')]
Serving: ('Book Request: Faculty X', 'faculty')
Serving: ('Book Request: Faculty Y', 'faculty')
Remaining: [('Book Request: Student A', 'student'), ('Book Request: Student B', 'student')]
PS C:\Users\91901\Desktop\TRAINING CLASSES\23-JUNE-2025 PYTHON TRAINING CLASSES\DAY-1\AI assist coding>
```

OBSERVATION:

Queue (FIFO):

First student who requested → gets served first.

Works like a line/queue in real life.

Operations:

enqueue() = O(1)

dequeue() = O(n) in Python list (because it shifts elements).

Priority Queue:

Faculty requests served before students, no matter order.

Useful for real-world scenarios where priority matters.

Slightly more complex to manage but ensures fairness for priority groups.

Conclusion:

Normal Queue → simple, first-come-first-serve.

Priority Queue → faculty first, then students. Better for library system.

Task 4: Navigation Assistant (Trees & Graphs)

Prompt:

Write Python code to implement a Binary Search Tree (BST) with insert, search, and traversal (inorder, preorder, postorder).

Show how to store building names in alphabetical order using BST and display traversals.

Write Python code to implement a graph with adjacency list representation.

Add a method to find the shortest path between two nodes using BFS.

Demonstrate graph navigation between buildings/rooms.

CODE:

```python
# ---------------- Binary Search Tree ----------------

class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

class BST:
    def __init__(self):
        self.root = None

    # Insert node into BST
    def insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        return root

    # Search node
    def search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self.search(root.left, key)
        return self.search(root.right, key)

    # Inorder traversal (Alphabetical order)
    def inorder(self, root):
        return self.inorder(root.left) + [root.key] + self.inorder(root.right) if root else []

    # Preorder traversal
    def preorder(self, root):
        return [root.key] + self.preorder(root.left) + self.preorder(root.right) if root else []

    # Postorder traversal
    def postorder(self, root):
        return self.postorder(root.left) + self.postorder(root.right) + [root.key] if root else []
```

```python
# ---------------- Graph with BFS ----------------

from collections import deque

class Graph:
    def __init__(self):
        self.adj_list = {}

    def add_edge(self, u, v):
        if u not in self.adj_list:
            self.adj_list[u] = []
        if v not in self.adj_list:
            self.adj_list[v] = []
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)  # Undirected graph

    def bfs_shortest_path(self, start, goal):
        visited = set()
        queue = deque([[start]])

        while queue:
            path = queue.popleft()
            node = path[-1]

            if node == goal:
                return path

            if node not in visited:
                for neighbor in self.adj_list.get(node, []):
                    new_path = list(path)
                    new_path.append(neighbor)
                    queue.append(new_path)
                visited.add(node)
        return None
```

```python
# --------------- Example Usage ---------------

# BST Example
print("----- Binary Search Tree -----")
bst = BST()
root = None
buildings = ["Library", "Auditorium", "Cafeteria", "Gym", "Hostel"]

for b in buildings:
    root = bst.insert(root, b)

print("Inorder Traversal (Alphabetical):", bst.inorder(root))
print("Preorder Traversal:", bst.preorder(root))
print("Postorder Traversal:", bst.postorder(root))
print("Search 'Gym':", "Found" if bst.search(root, "Gym") else "Not Found")

# Graph Example
print("\n----- Graph with BFS -----")
g = Graph()
g.add_edge("Library", "Auditorium")
g.add_edge("Library", "Cafeteria")
g.add_edge("Cafeteria", "Gym")
g.add_edge("Gym", "Hostel")

print("Graph Adjacency List:", g.adj_list)
print("Shortest path Library -> Hostel:", g.bfs_shortest_path("Library", "Hostel"))
```

OUTPUT:

```
----- Binary Search Tree -----
Inorder Traversal (Alphabetical): ['Auditorium', 'Cafeteria', 'Gym', 'Hostel', 'Library']
Preorder Traversal: ['Library', 'Auditorium', 'Cafeteria', 'Gym', 'Hostel']
Postorder Traversal: ['Hostel', 'Gym', 'Cafeteria', 'Auditorium', 'Library']
Search 'Gym': Found

----- Graph with BFS -----
Graph Adjacency List: {'Library': ['Auditorium', 'Cafeteria'], 'Auditorium': ['Library'], 'Cafeteria': ['Library', 'Gym'], 'Gym': ['Cafeteria', 'Hostel'], 'Host
el': ['Gym']}
Shortest path Library -> Hostel: ['Library', 'Cafeteria', 'Gym', 'Hostel']
PS C:\Users\91901\Desktop\TRAINING CLASSES\23-JUNE-2025 PYTHON TRAINING CLASSES\DAY-3\AI assist coding>
```

Observations:

Binary Search Tree (BST):

Stores buildings in sorted order.

Traversals:

Inorder → Alphabetical order.

Preorder → Root first, then left-right.

Postorder → Children first, then root.

Search is efficient (O(log n) average).

Graph with BFS:

Models buildings/rooms as nodes and paths as edges.

BFS finds the shortest path (minimum steps).

Example: Shortest path from Library to Hostel goes via Cafeteria and Gym.

Conclusion:

BST is good for storing data in sorted order (easy lookup).

Graph with BFS is best for navigation problems (finding routes).