Practical No 6

Aim: Write A Program To Implement Linear Queue and Circular Queue

```
class LinearQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = 0
    def enqueue(self, task):
        if self.rear == self.capacity:  # No space left, even if front moved
            print("Linear Queue Overflow! Cannot enqueue:", task)
            return
        self.queue[self.rear] = task
        self.rear += 1
        print(f"Enqueued (Linear): {task}")
    def dequeue(self):
        if self.front == self.rear:
            print("Linear Queue Underflow! Cannot dequeue.")
            return
        task = self.queue[self.front]
        self.front += 1
        print(f"Dequeued (Linear): {task}")
        return task
    def display(self):
        if self.front == self.rear:
            print("Linear Queue is empty.")
        else:
            print("Linear Queue:", self.queue[self.front:self.rear])

class CircularQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = -1
        self.rear = -1
    def enqueue(self, task):
        # Queue is full when next position of rear == front
        if (self.rear + 1) % self.capacity == self.front:
            print("Circular Queue is Full! Cannot enqueue:", task)
            return
        if self.front == -1:  # First element
            self.front = 0
        self.rear = (self.rear + 1) % self.capacity
        self.queue[self.rear] = task
        print(f"Enqueued (Circular): {task}")
    def dequeue(self):
        if self.front == -1:
            print("Circular Queue is Empty! Cannot dequeue.")
            return
        task = self.queue[self.front]
```

```
        if self.front == self.rear:  # Queue has only one element
            self.front = -1
            self.rear = -1
        else:
            self.front = (self.front + 1) % self.capacity
        print(f"Dequeued (Circular): {task}")
        return task
    def display(self):
        if self.front == -1:
            print("Circular Queue is empty.")
            return
        print("Circular Queue:", end=" ")
        i = self.front
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.capacity
        print()

# Example Simulation of Task Scheduling
if __name__ == "__main__":
    tasks = ["Task1", "Task2", "Task3", "Task4", "Task5"]
    print("=== Linear Queue Simulation ===")
    lq = LinearQueue(3)
    for task in tasks:
        lq.enqueue(task)
    lq.display()
    lq.dequeue()
    lq.dequeue()
    lq.enqueue("Task6")
    lq.display()
    print("\n=== Circular Queue Simulation ===")
    cq = CircularQueue(3)
    for task in tasks:
        cq.enqueue(task)
    cq.display()
    cq.dequeue()
    cq.enqueue("Task6")
    cq.display()
```

```
=== Linear Queue Simulation ===
Enqueued (Linear): Task1
Enqueued (Linear): Task2
Enqueued (Linear): Task3
Linear Queue Overflow! Cannot enqueue: Task4
Linear Queue Overflow! Cannot enqueue: Task5
Linear Queue: ['Task1', 'Task2', 'Task3']
Dequeued (Linear): Task1
Dequeued (Linear): Task2
Linear Queue Overflow! Cannot enqueue: Task6
Linear Queue: ['Task3']
=== Circular Queue Simulation ===
Enqueued (Circular): Task1
Enqueued (Circular): Task2
Enqueued (Circular): Task3
Circular Queue is Full! Cannot enqueue: Task4
Circular Queue is Full! Cannot enqueue: Task5
Circular Queue: Task1 Task2 Task3
Dequeued (Circular): Task1
Enqueued (Circular): Task6
Circular Queue: Task2 Task3 Task6
```

Practical No 7

Aim : Write A Program To Implement Binary Search Tree (BST)

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

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

    def insert(self, key):
        """Insert a new node into the BST."""
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)

    def _insert(self, root, key):
        if key < root.key:
            if root.left is None:
                root.left = Node(key)
            else:
                self._insert(root.left, key)
        else:
            if root.right is None:
                root.right = Node(key)
            else:
                self._insert(root.right, key)

    # Traversals
    def inorder(self, root):
        """Left → Root → Right"""
        return self.inorder(root.left) + [root.key] + self.inorder(root.right) if root else []

    def preorder(self, root):
        """Root → Left → Right"""
        return [root.key] + self.preorder(root.left) + self.preorder(root.right) if root else []

    def postorder(self, root):
        """Left → Right → Root"""
        return self.postorder(root.left) + self.postorder(root.right) + [root.key] if root else []

# Example Usage
if __name__ == "__main__":
    # Dataset
    dataset = [50, 30, 20, 40, 70, 60, 80]
```

```python
bst = BST()
for value in dataset:
    bst.insert(value)

print("Dataset:", dataset)
print("\n--- Tree Traversals ---")
print("In-order Traversal   :", bst.inorder(bst.root))
print("Pre-order Traversal  :", bst.preorder(bst.root))
print("Post-order Traversal :", bst.postorder(bst.root))

print("\nSorted sequence (from in-order):", bst.inorder(bst.root))
```

```
Dataset: [50, 30, 20, 40, 70, 60, 80]

--- Tree Traversals ---
In-order Traversal   : [20, 30, 40, 50, 60, 70, 80]
Pre-order Traversal  : [50, 30, 20, 40, 70, 60, 80]
Post-order Traversal : [20, 40, 30, 60, 80, 70, 50]

Sorted sequence (from in-order): [20, 30, 40, 50, 60, 70, 80]
```

Practical No 8

Aim : Write A Program To Implement Balanced Trees & Priority Queues

```
class AVLNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1

class AVLTree:
    def get_height(self, node):
        return node.height if node else 0

    def get_balance(self, node):
        return self.get_height(node.left) - self.get_height(node.right) if node else 0

    def right_rotate(self, y):
        x = y.left
        T2 = x.right

        # Perform rotation
        x.right = y
        y.left = T2

        # Update heights
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        return x

    def left_rotate(self, x):
        y = x.right
        T2 = y.left

        # Perform rotation
        y.left = x
        x.right = T2

        # Update heights
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def insert(self, node, key):
        # 1. Normal BST insert
        if not node:
            return AVLNode(key)
        elif key < node.key:
            node.left = self.insert(node.left, key)
        else:
```

```
            node.right = self.insert(node.right, key)

        # 2. Update height
        node.height = 1 + max(self.get_height(node.left), self.get_height(node.right))

        # 3. Balance factor
        balance = self.get_balance(node)

        # 4. Rebalance cases
        # Left Left
        if balance > 1 and key < node.left.key:
            return self.right_rotate(node)
        # Right Right
        if balance < -1 and key > node.right.key:
            return self.left_rotate(node)
        # Left Right
        if balance > 1 and key > node.left.key:
            node.left = self.left_rotate(node.left)
            return self.right_rotate(node)
        # Right Left
        if balance < -1 and key < node.right.key:
            node.right = self.right_rotate(node.right)
            return self.left_rotate(node)

        return node

    def inorder(self, root):
        return self.inorder(root.left) + [root.key] + self.inorder(root.right) if root else []

# -------------------- Priority Queue (Min-Heap) --------------------
import heapq

class PriorityQueue:
    def __init__(self):
        self.queue = []

    def push(self, priority, task):
        """Insert into priority queue (min-heap)."""
        heapq.heappush(self.queue, (priority, task))

    def pop(self):
        """Remove and return highest priority task (lowest priority number)."""
        if self.queue:
            return heapq.heappop(self.queue)
        return None

    def show(self):
        return sorted(self.queue)

# -------------------- Example Usage --------------------
if __name__ == "__main__":
    print("\n--- AVL Tree Example ---")
    avl = AVLTree()
```

```
root = None
values = [10, 20, 30, 40, 50, 25]

for v in values:
    root = avl.insert(root, v)
    print(f"Inserted {v}, In-order Traversal: {avl.inorder(root)}")

print("\nFinal Balanced AVL Tree (In-order):", avl.inorder(root))

print("\n--- Priority Queue Example ---")
pq = PriorityQueue()
pq.push(2, "Job A")
pq.push(1, "Emergency Patient")
pq.push(3, "Routine Checkup")
pq.push(5, "Background Task")
pq.push(4, "Job B")

print("Current Queue:", pq.show())
while pq.queue:
    priority, task = pq.pop()
    print(f"Processing Task: {task} (Priority {priority})")
```

```
--- AVL Tree Example ---
Inserted 10, In-order Traversal: [10]
Inserted 20, In-order Traversal: [10, 20]
Inserted 30, In-order Traversal: [10, 20, 30]
Inserted 40, In-order Traversal: [10, 20, 30, 40]
Inserted 50, In-order Traversal: [10, 20, 30, 40, 50]
Inserted 25, In-order Traversal: [10, 20, 25, 30, 40, 50]

Final Balanced AVL Tree (In-order): [10, 20, 25, 30, 40, 50]

--- Priority Queue Example ---
Current Queue: [(1, 'Emergency Patient'), (2, 'Job A'), (3, 'Routine Checkup'), (4, 'Job
B'), (5, 'Background Task')]
Processing Task: Emergency Patient (Priority 1)
Processing Task: Job A (Priority 2)
Processing Task: Routine Checkup (Priority 3)
Processing Task: Job B (Priority 4)
Processing Task: Background Task (Priority 5)
```

Practical No 9

Aim: Write A Program To Implement Graph

```
from collections import deque, defaultdict

class Graph:
    def __init__(self, vertices):
        self.V = vertices
        # Adjacency Matrix
        self.adj_matrix = [[0] * vertices for _ in range(vertices)]
        # Adjacency List
        self.adj_list = defaultdict(list)

    def add_edge(self, u, v):
        """Add an undirected edge between u and v"""
        # For adjacency matrix
        self.adj_matrix[u][v] = 1
        self.adj_matrix[v][u] = 1
        # For adjacency list
        self.adj_list[u].append(v)
        self.adj_list[v].append(u)

    def display_matrix(self):
        print("\nAdjacency Matrix:")
        for row in self.adj_matrix:
            print(row)

    def display_list(self):
        print("\nAdjacency List:")
        for node in self.adj_list:
            print(node, "->", self.adj_list[node])

    def bfs(self, start):
        visited = [False] * self.V
        queue = deque([start])
        visited[start] = True
        result = []

        while queue:
            node = queue.popleft()
            result.append(node)
            for neighbor in self.adj_list[node]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)
        return result

    def dfs(self, start):
        visited = [False] * self.V
        result = []
```

```python
        def dfs_recursive(v):
            visited[v] = True
            result.append(v)
            for neighbor in self.adj_list[v]:
                if not visited[neighbor]:
                    dfs_recursive(neighbor)

        dfs_recursive(start)
        return result

# -------------------- Example Usage --------------------
if __name__ == "__main__":
    # Example: Social Network (0=Alice, 1=Bob, 2=Charlie, 3=David, 4=Eve)
    g = Graph(5)

    # Connections
    g.add_edge(0, 1)  # Alice - Bob
    g.add_edge(0, 2)  # Alice - Charlie
    g.add_edge(1, 3)  # Bob - David
    g.add_edge(2, 4)  # Charlie - Eve
    g.add_edge(3, 4)  # David - Eve

    # Display representations
    g.display_matrix()
    g.display_list()

    # Traversals
    print("\nBFS Traversal (from Alice/0):", g.bfs(0))
    print("DFS Traversal (from Alice/0):", g.dfs(0))
```

```
Adjacency Matrix:
[0, 1, 1, 0, 0]
[1, 0, 0, 1, 0]
[1, 0, 0, 0, 1]
[0, 1, 0, 0, 1]
[0, 0, 1, 1, 0]

Adjacency List:
0 -> [1, 2]
1 -> [0, 3]
2 -> [0, 4]
3 -> [1, 4]
4 -> [2, 3]

BFS Traversal (from Alice/0): [0, 1, 2, 3, 4]
DFS Traversal (from Alice/0): [0, 1, 3, 4, 2]
```

Practical No 10

Aim: Write A Program To Implement Hashing Concepts & Collision Handling

```
# -------------------- Hash Table with Chaining --------------------
class HashTableChaining:
    def __init__(self, size=7):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        return key % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        # Update if key already exists
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
                return
        self.table[index].append([key, value])

    def search(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
        return None

    def delete(self, key):
        index = self.hash_function(key)
        for i, pair in enumerate(self.table[index]):
            if pair[0] == key:
                del self.table[index][i]
                return True
        return False

    def display(self):
        print("\nHash Table (Chaining):")
        for i, bucket in enumerate(self.table):
            print(i, "->", bucket)

# -------------------- Hash Table with Linear Probing --------------------
class HashTableLinearProbing:
    def __init__(self, size=7):
        self.size = size
        self.table = [None] * size

    def hash_function(self, key):
        return key % self.size
```

```python
    def insert(self, key, value):
        index = self.hash_function(key)
        start_index = index
        while self.table[index] is not None and self.table[index][0] != key:
            index = (index + 1) % self.size
            if index == start_index:
                print("Hash Table Full! Cannot insert:", key)
                return
        self.table[index] = (key, value)

    def search(self, key):
        index = self.hash_function(key)
        start_index = index
        while self.table[index] is not None:
            if self.table[index][0] == key:
                return self.table[index][1]
            index = (index + 1) % self.size
            if index == start_index:
                break
        return None

    def delete(self, key):
        index = self.hash_function(key)
        start_index = index
        while self.table[index] is not None:
            if self.table[index][0] == key:
                self.table[index] = None
                return True
            index = (index + 1) % self.size
            if index == start_index:
                break
        return False

    def display(self):
        print("\nHash Table (Linear Probing):")
        for i, val in enumerate(self.table):
            print(i, "->", val)

# -------------------- Example Usage --------------------
if __name__ == "__main__":
    # Using Chaining
    ht_chain = HashTableChaining()
    ht_chain.insert(10, "Alice")
    ht_chain.insert(20, "Bob")
    ht_chain.insert(30, "Charlie")
    ht_chain.insert(17, "David")  # Collides with 10 (10 % 7 == 3, 17 % 7 == 3)
    ht_chain.display()
    print("Search key 20:", ht_chain.search(20))
    ht_chain.delete(10)
    ht_chain.display()

    # Using Linear Probing
```

```
ht_lp = HashTableLinearProbing()
ht_lp.insert(10, "Red")
ht_lp.insert(20, "Blue")
ht_lp.insert(30, "Green")
ht_lp.insert(17, "Yellow")  # Collision, will probe next slot
ht_lp.display()
print("Search key 17:", ht_lp.search(17))
ht_lp.delete(20)
ht_lp.display()
```

```
Hash Table (Chaining):
0 -> []
1 -> []
2 -> [[30, 'Charlie']]
3 -> [[10, 'Alice'], [17, 'David']]
4 -> []
5 -> []
6 -> [[20, 'Bob']]
Search key 20: Bob

Hash Table (Chaining):
0 -> []
1 -> []
2 -> [[30, 'Charlie']]
3 -> [[17, 'David']]
4 -> []
5 -> []
6 -> [[20, 'Bob']]

Hash Table (Linear Probing):
0 -> None
1 -> None
2 -> (30, 'Green')
3 -> (10, 'Red')
4 -> (17, 'Yellow')
5 -> None
6 -> (20, 'Blue')
Search key 17: Yellow

Hash Table (Linear Probing):
0 -> None
1 -> None
2 -> (30, 'Green')
3 -> (10, 'Red')
4 -> (17, 'Yellow')
5 -> None
6 -> None
```