Q1: Logistics Company and Priority Queue

1. Why is a priority queue appropriate for managing deliveries? A priority queue is the most suitable data structure for managing deliveries because:

- **Efficient Prioritization**: It allows tasks to be processed based on their priority level, ensuring urgent deliveries are handled first.
- **Stable Order for Equal Priority**: For tasks with the same priority, it processes them in the order of their arrival (FIFO behavior for equal priorities).
- **Dynamic Reordering**: It dynamically adjusts the order of tasks as new tasks are added.

Relevant Features:

- **Heap-based Implementation**: Efficient insertion and retrieval of the highest-priority element in time.
- **Applicability**: Matches the company's requirements for urgency and fairness.

2. Processing Order of Tasks: Given Tasks:

- Task A: Priority 1 (Urgent delivery)
- Task B: Priority 3 (Bulk delivery)
- Task C: Priority 2 (Standard delivery)
- Task D: Priority 1 (Urgent delivery)

Processing Order: Task A, Task D, Task C, Task B.

Explanation:

- Tasks with Priority 1 (Task A and Task D) are processed first, in the order they arrived.
- Task C (Priority 2) is processed next.
- Task B (Priority 3) is processed last.

Q2: AVL Tree for Course Registration

1. Efficiency of AVL Tree vs. Unbalanced Binary Search Tree (BST):

- **Self-Balancing Property**: AVL Trees maintain a height difference of at most 1 between subtrees, ensuring complexity for insertions and searches.
- **Unbalanced BST**: Can degenerate into a linked list, leading to complexity in the worst case.

2. Structure of the AVL Tree: Insertions: 50, 30, 70, 20, 40, 60, 80.

Step-by-step Construction:

- Insert 50: Root node.
- Insert 30: Becomes the left child of 50.
- Insert 70: Becomes the right child of 50.
- Insert 20: Becomes the left child of 30.
- Insert 40: Becomes the right child of 30. (Tree remains balanced.)
- Insert 60: Becomes the left child of 70.
- Insert 80: Becomes the right child of 70. (Tree remains balanced.)

Final AVL Tree Structure:

```
       50
      /  \
    30    70
   /  \   /  \
 20   40 60  80
```

Rotations: No rotations are required as the tree remains balanced after each insertion.

Q3: Transportation Network (Graph)

a) Shortest Paths using Dijkstra's Algorithm:

- **Steps Involved:**

1. Start with Node A (central hub) and set its distance to 0; other nodes are initialized to infinity.
2. Select the node with the smallest distance and update the distances to its neighbors.
3. Repeat until all nodes have been processed.

Example: Graph: Neighborhoods (Nodes: A, B, C, D, E), Edge Weights (A-B: 4, A-C: 2, B-C: 3, C-D: 2, D-E: 6).

Shortest Paths:

- A to B: 4 (via A-B)
- A to C: 2 (direct)
- A to D: 4 (via C-D)
- A to E: 10 (via D-E)

b) Minimum Spanning Tree (MST) using Kruskal's Algorithm:

- **Steps Involved:**

1. Sort edges by weight.
2. Add edges to the MST, ensuring no cycles are formed.

Selected Edges:

- D-C
- D-E
- C-A
- C-B

MST Edges: D-C, D-E, C-A, C-B.

Q4: Sorting and Algorithm Analysis

Input Array: [8, 3, 1, 7, 0, 10, 2]

a) Bubble Sort Steps:

1. Compare adjacent elements and swap if necessary.
2. Repeat until no swaps are needed.

Steps:

1. [3, 1, 7, 0, 8, 2, 10]
2. [1, 3, 0, 7, 2, 8, 10]
3. [1, 0, 3, 2, 7, 8, 10]
4. [0, 1, 2, 3, 7, 8, 10]

b) Merge Sort Steps:

1. Divide array into halves until each subarray has one element.
2. Merge subarrays in sorted order.

Steps:

1. [8, 3, 1, 7] [0, 10, 2]
2. [3, 8], [1, 7], [0, 2, 10]
3. [1, 3, 7, 8], [0, 2, 10]
4. [0, 1, 2, 3, 7, 8, 10]

b) Comparisons and Big-O Analysis:

- **Bubble Sort:** comparisons in the worst case.
- **Merge Sort:** comparisons in all cases.

Comparison: Merge Sort is more efficient for large inputs due to its logarithmic depth.