# ALGORITHMS FOR GRAPHS AND TREES

## Chapter 6 – Part2

**Prepared by: Enas Abu Samra**

# KEY POINTS OF CHAPTER 6

- Graph and Tree Traversal (Search) Algorithms.
  - ✓ Depth First Search.
  - ✓ Breadth First Search.
- Spanning Tree and Minimum Spanning Tree (MST).
- Two Greedy Algorithms For MSP:
  - ✓ Kruskal’s Algorithm.
  - ✓ Prim’s Algorithm.
- Shortest Path Algorithms in Graphs
  - ✓ Dijkstra’s Algorithm.
  - ✓ Bellman-ford Algorithm.
- The Difference Between MST and The Shortest Path.

# Graph and Tree Traversal (Search) Algorithms

# Graphs Traversal Algorithms

# GRAPHS TRAVERSAL ALGORITHMS

- There are many traversal algorithms in Graphs:
  1. Depth-First-Search (DFS)
  2. Breadth-First-Search (BFS)

- In **DFS**, the node branch is explored as far as possible before being forced to backtrack and expand other nodes. (using stack)
- In **BFS**, all the node neighbors are all explored at the present depth prior to moving on to the nodes at the next depth level. (using queue)
- **Time Complexity of DFS and BFS** = O(V+E) where V is vertices and E is edges.

# DEPTH-FIRST-SEARCH (DFS)

**Procedure DFS (L: adjacency List, v: vertex)**

initialize stack.

visit, mark and push(v)

while stack is nonempty do

    while there is an unmarked vertex w adjacent to top (stack) do

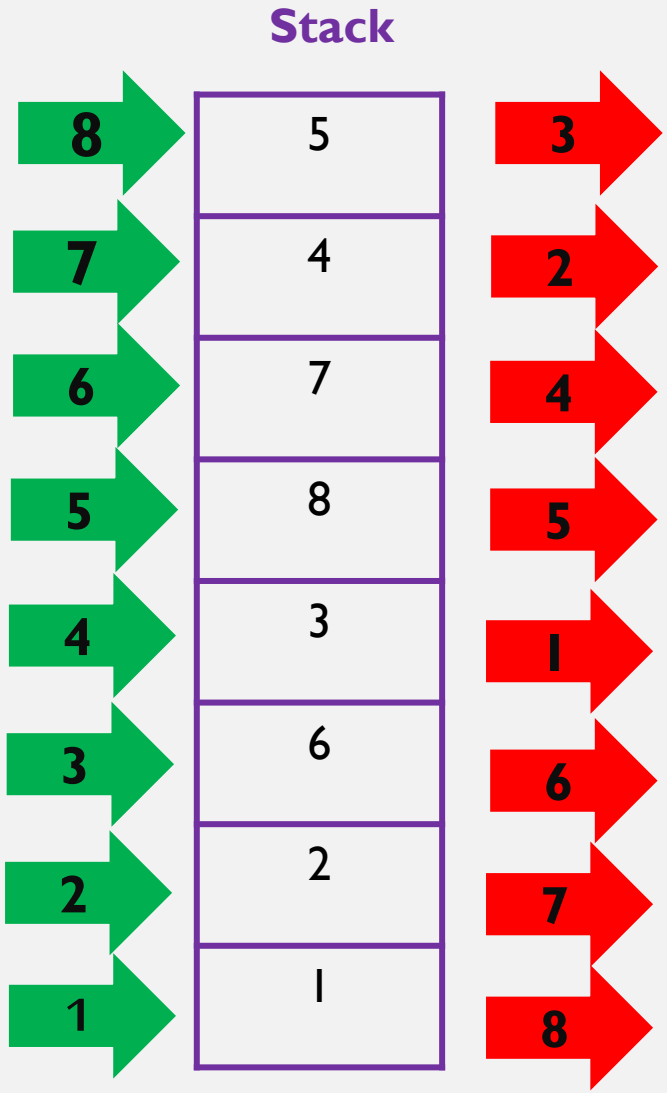
        visit, mark and push(w)

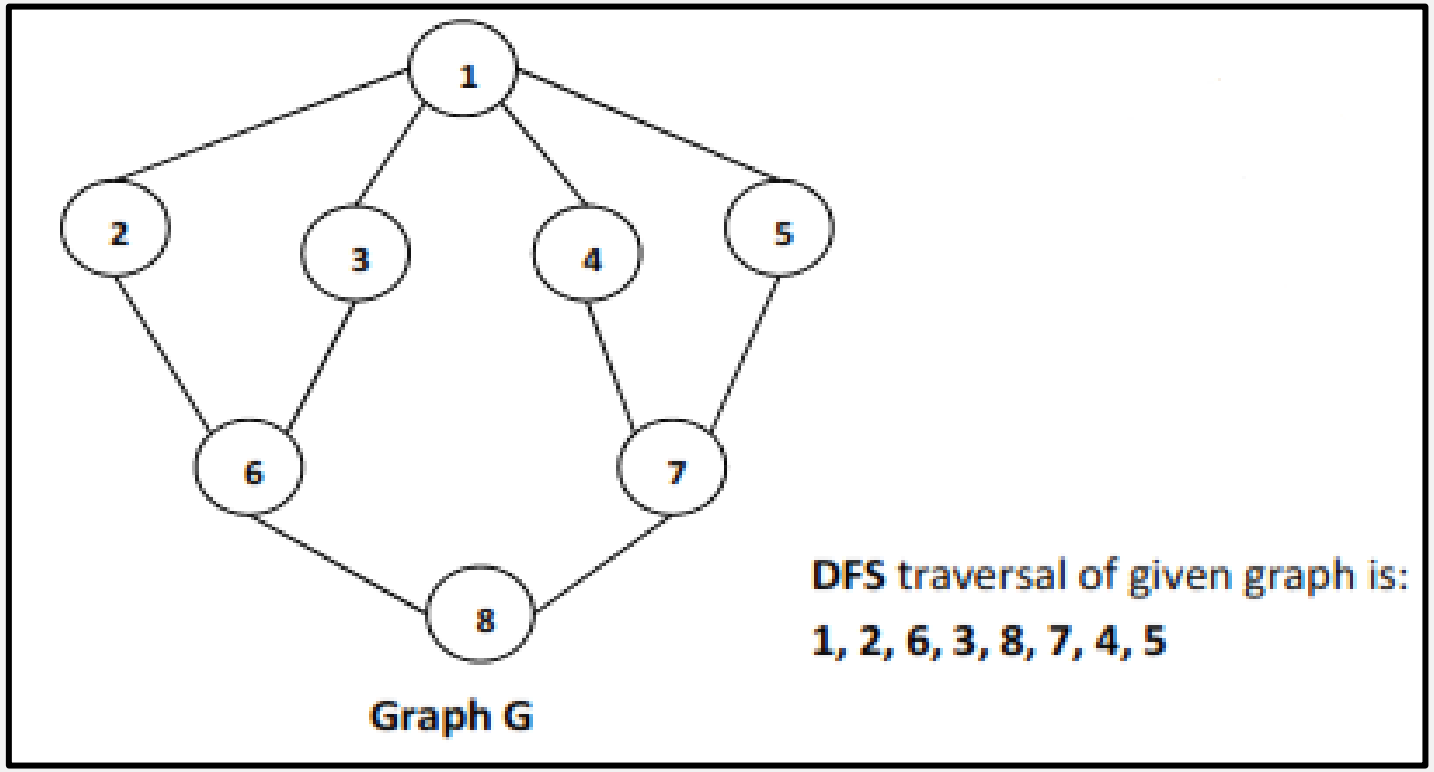
    end

    pop(stack) // the top(stack) was completely explored

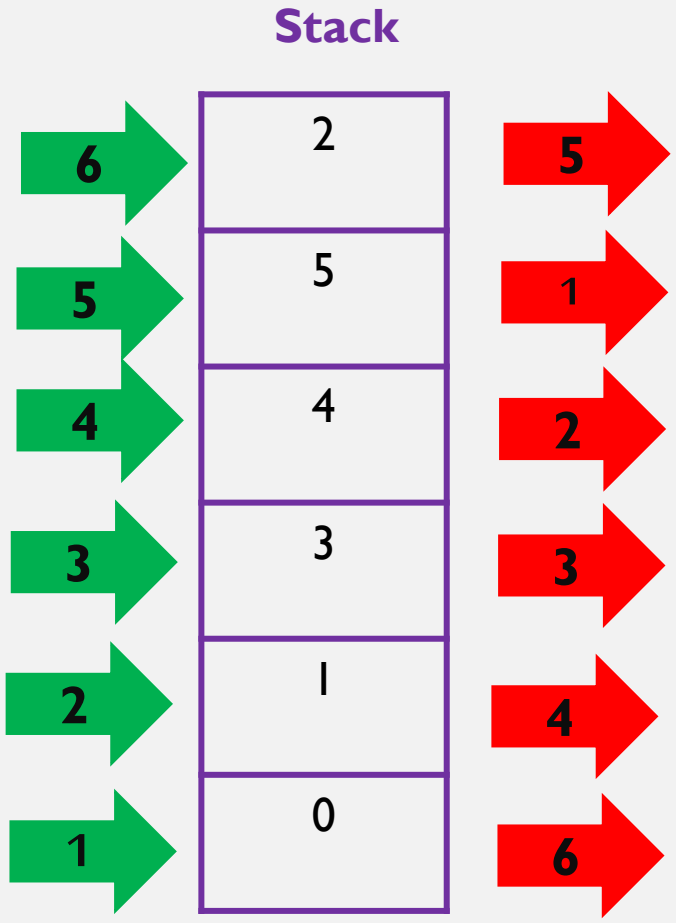
end

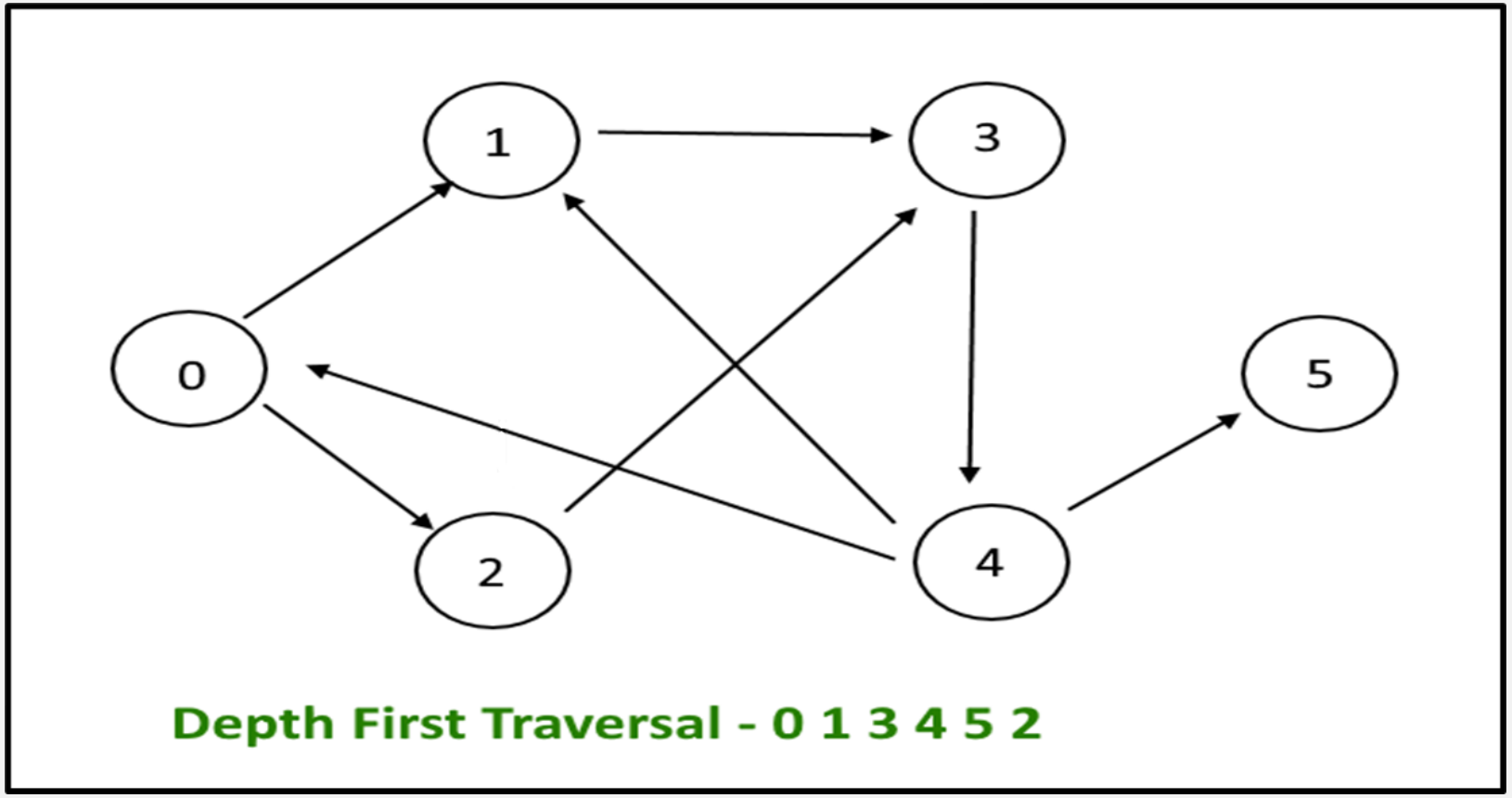
# APPLYING DFS ALGORITHMS ON GRAPHS



# APPLYING DFS ALGORITHMS ON GRAPHS



# BREADTH-FIRST-SEARCH (BFS)

**Procedure BFS (L: adjacency List, v: vertex)**

initialize the queue Q

visit and mark v

enqueue (v, Q)

while Qis nonempty do

    x= the next vertex in the queue Q

    for each unmarked vertex w adjacent to x do

        visit and mark w

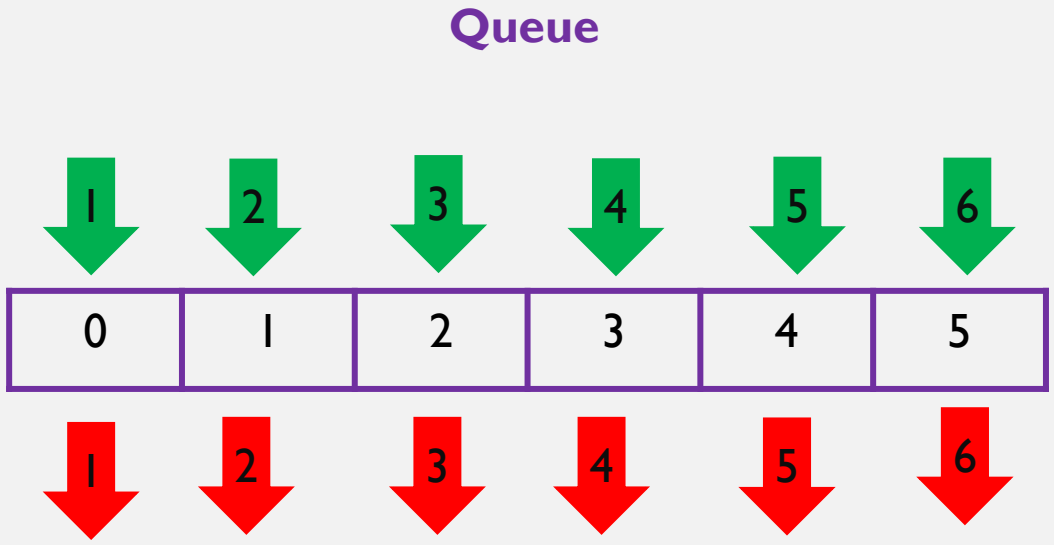
        enqueue(w, Q)

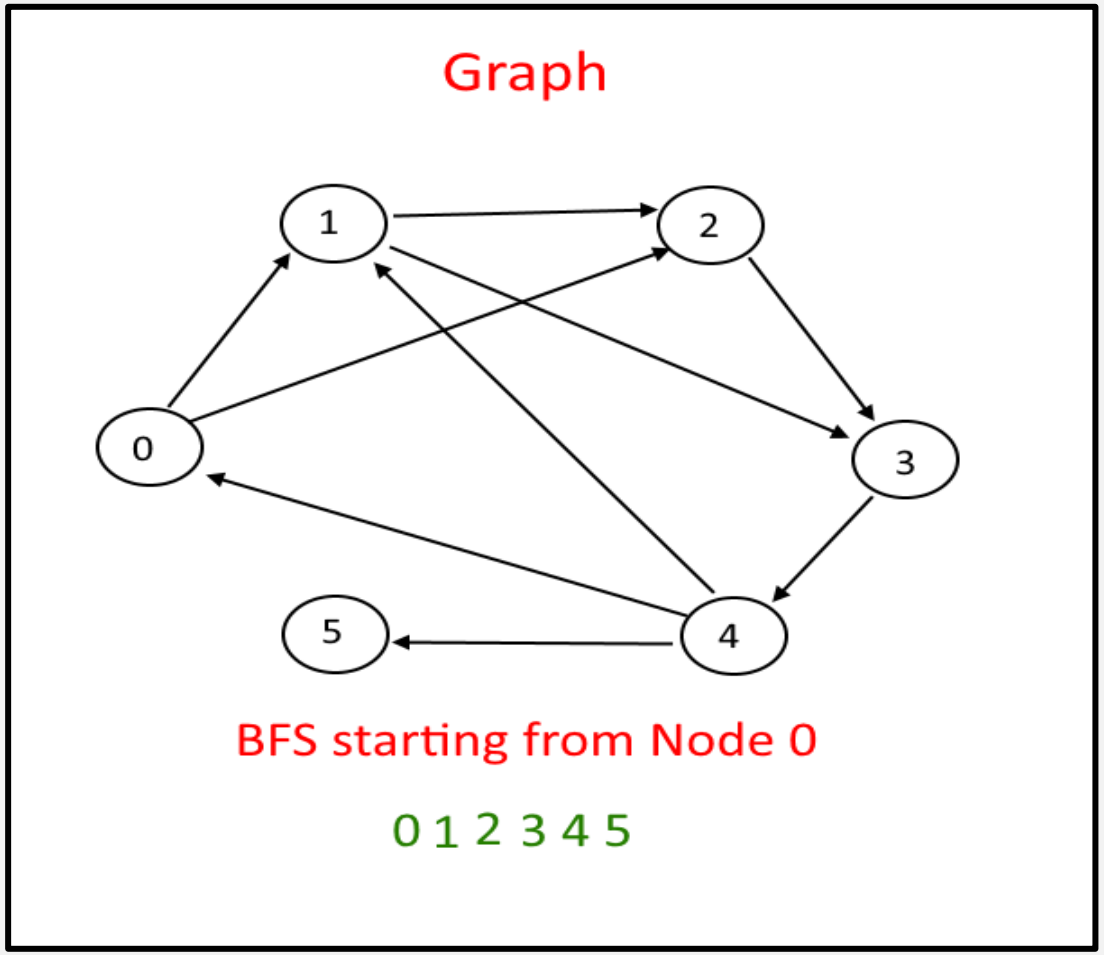
    end

    remove x from the queue Q

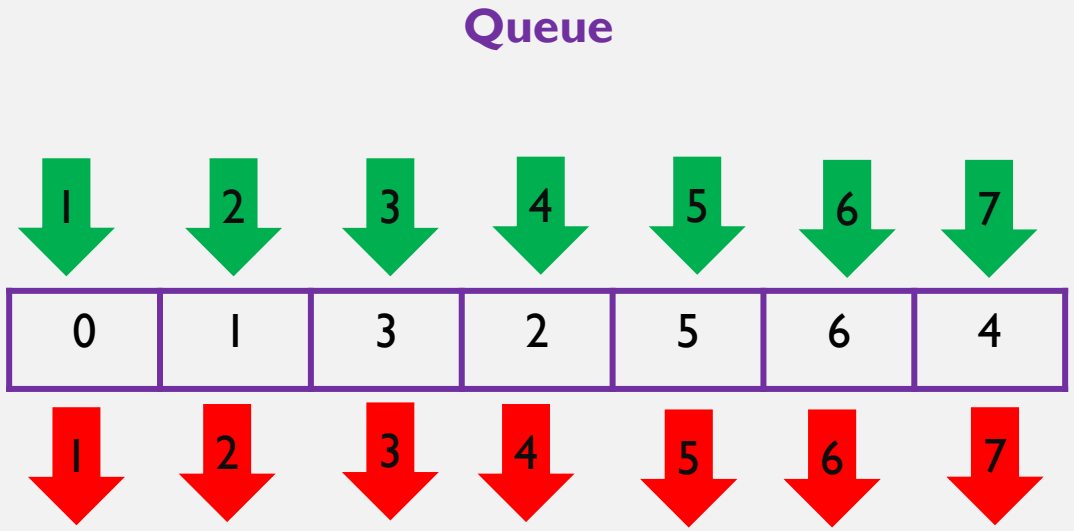
end

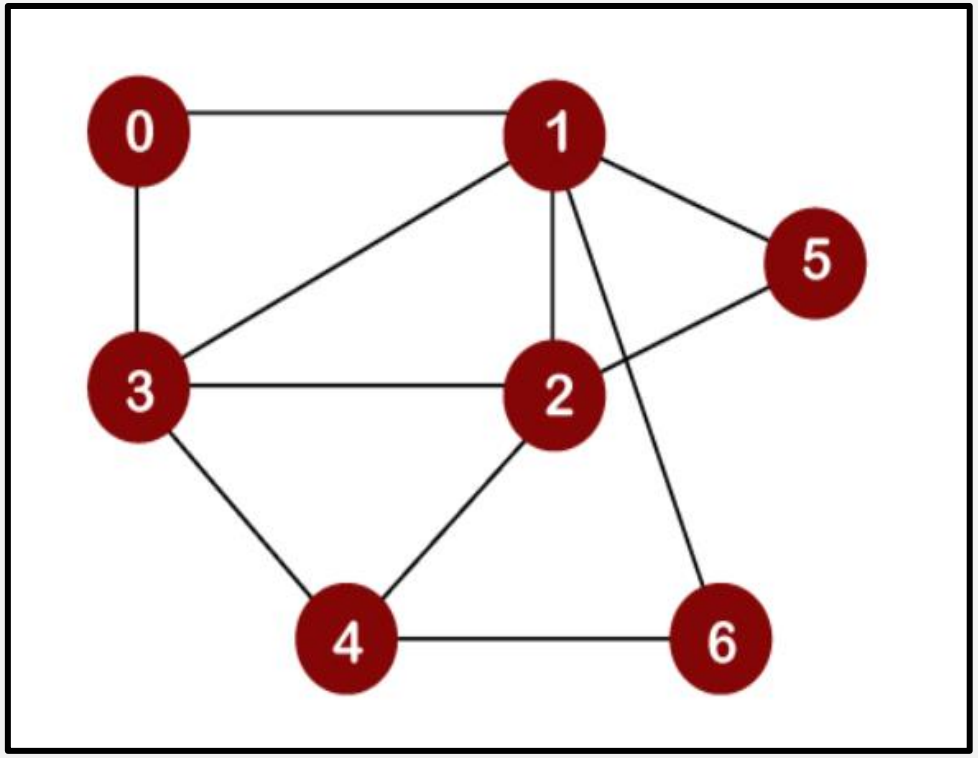
# APPLYING BFS ALGORITHMS ON GRAPHS



# APPLYING BFS ALGORITHMS ON GRAPHS




Start at node (0) ➔ BFS: 0, 1, 3, 2, 5, 6, 4

# Trees Traversal Algorithms

# TREES TRAVERSAL ALGORITHMS

- Traversal is a process to visit all the nodes of a tree.
- Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.
- Because all nodes are connected via edges (links) we always start from the root (head) node.
- You cannot randomly access a node in a tree.
- There are two ways to perform the traversal:

1. **Depth First Traversals:**
   - ✓ In-order (Left, Root, Right).
   - ✓ Pre-order (Root, Left, Right).
   - ✓ Post-order (Left, Right, Root).
2. **Breadth First or Level Order Traversal**

# IN-ORDER ALGORITHM

- **In-Order Algorithm (Left, Root, Right)**
- Until all nodes are traversed:

**Step 1:** Recursively traverse **left** subtree.

**Step 2:** Visit **root** node.

**Step 3:** Recursively traverse **right** subtree.

# PRE-ORDER ALGORITHM

- **Pre-Order Algorithm (Root, Left, Right)**
- Until all nodes are traversed:

**Step 1:** Visit **root** node.

**Step 2:** Recursively traverse **left** subtree.

**Step 3:** Recursively traverse **right** subtree.

# IN-ORDER ALGORITHM

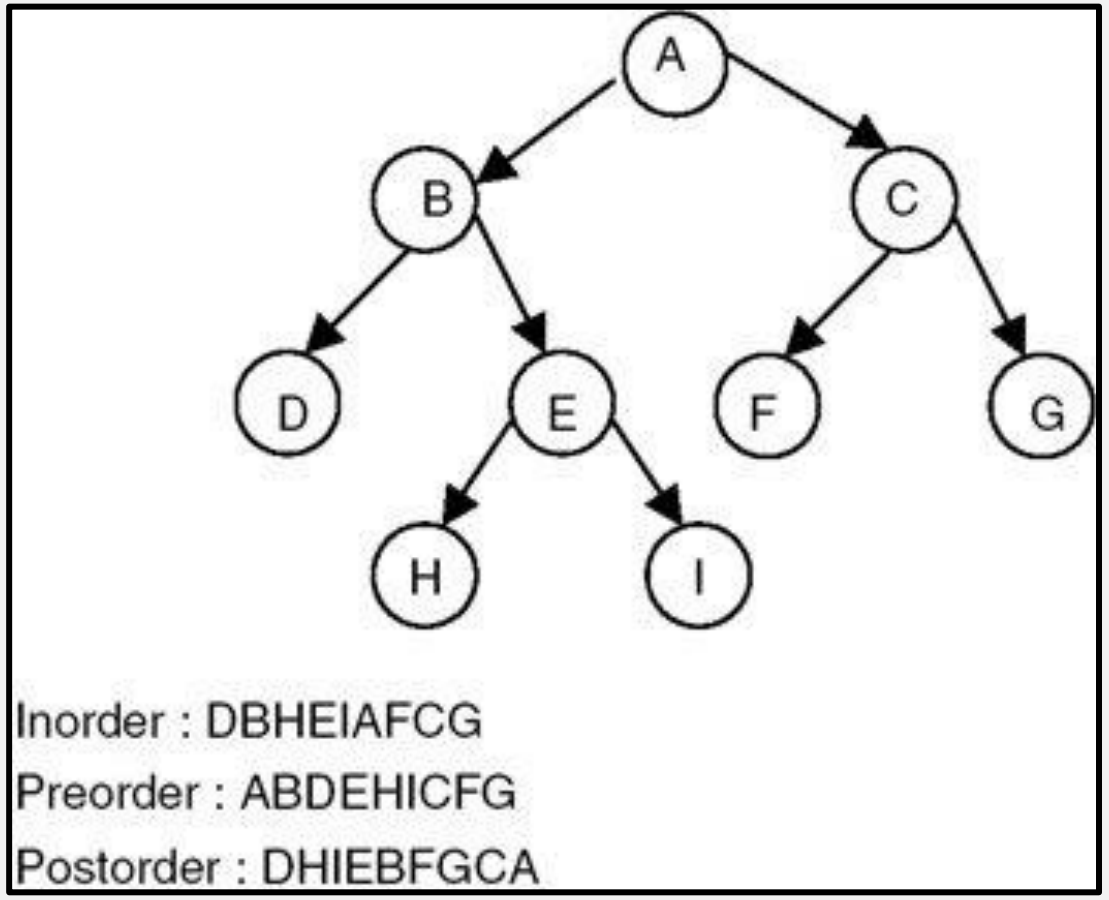
- **Post-Order Algorithm (Left, Right, Root)**
- Until all nodes are traversed:

**Step 1:** Recursively traverse **left** subtree.

**Step 2:** Recursively traverse **right** subtree.

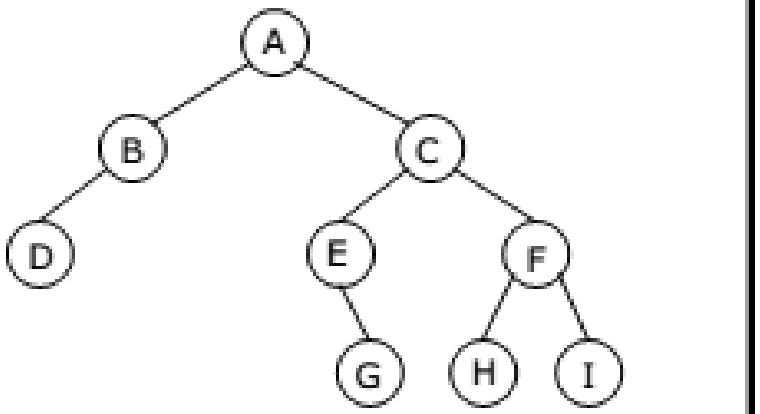
**Step 3:** Visit **root** node.

# APPLYING DFS AND BFS ALGORITHMS ON TREES




BFS order ➔ A B C D E F G H I

# APPLYING DFS AND BFS ALGORITHMS ON TREES




Binary Tree

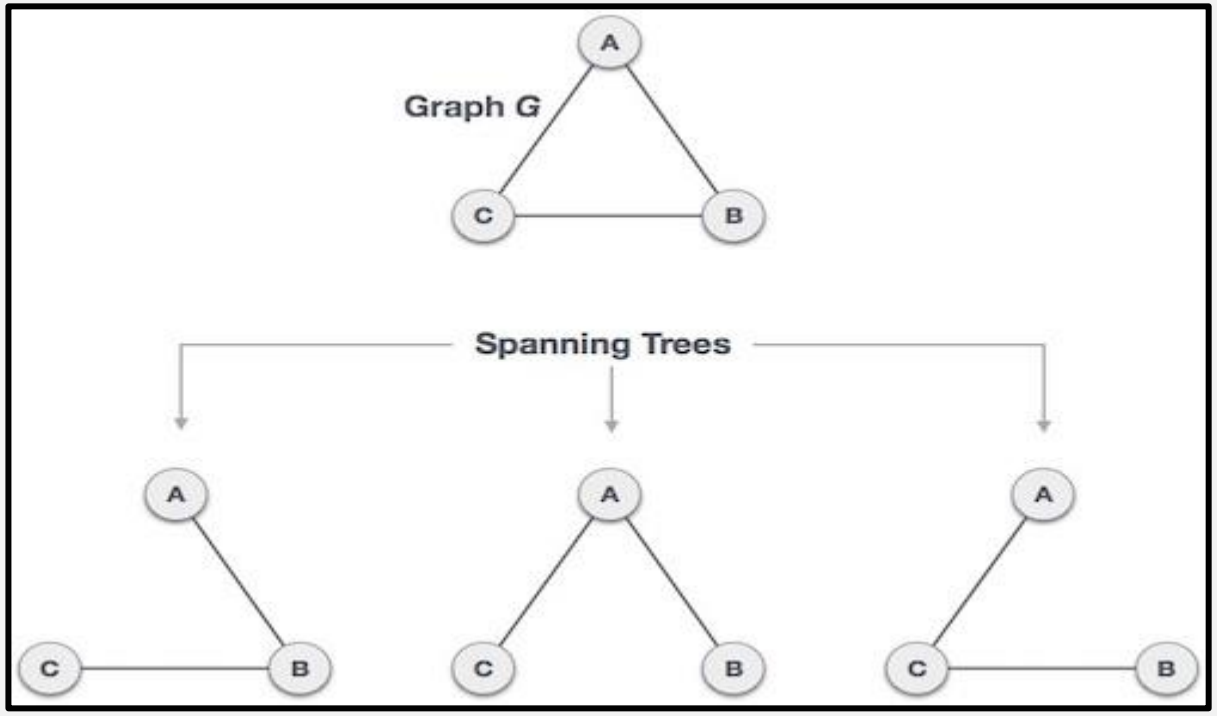
- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I
- Level order traversal yields:
  A, B, C, D, E, F, G, H, I

Pre, Post, Inorder and level order Traversing

# Spanning Tree
# and
# Minimum Spanning Tree (MST)

# SPANNING TREE

- A spanning tree of a graph G is a subgraph of G that is a tree that contains all the vertices of G with no cycles.
- **A single graph can have many different spanning trees.**



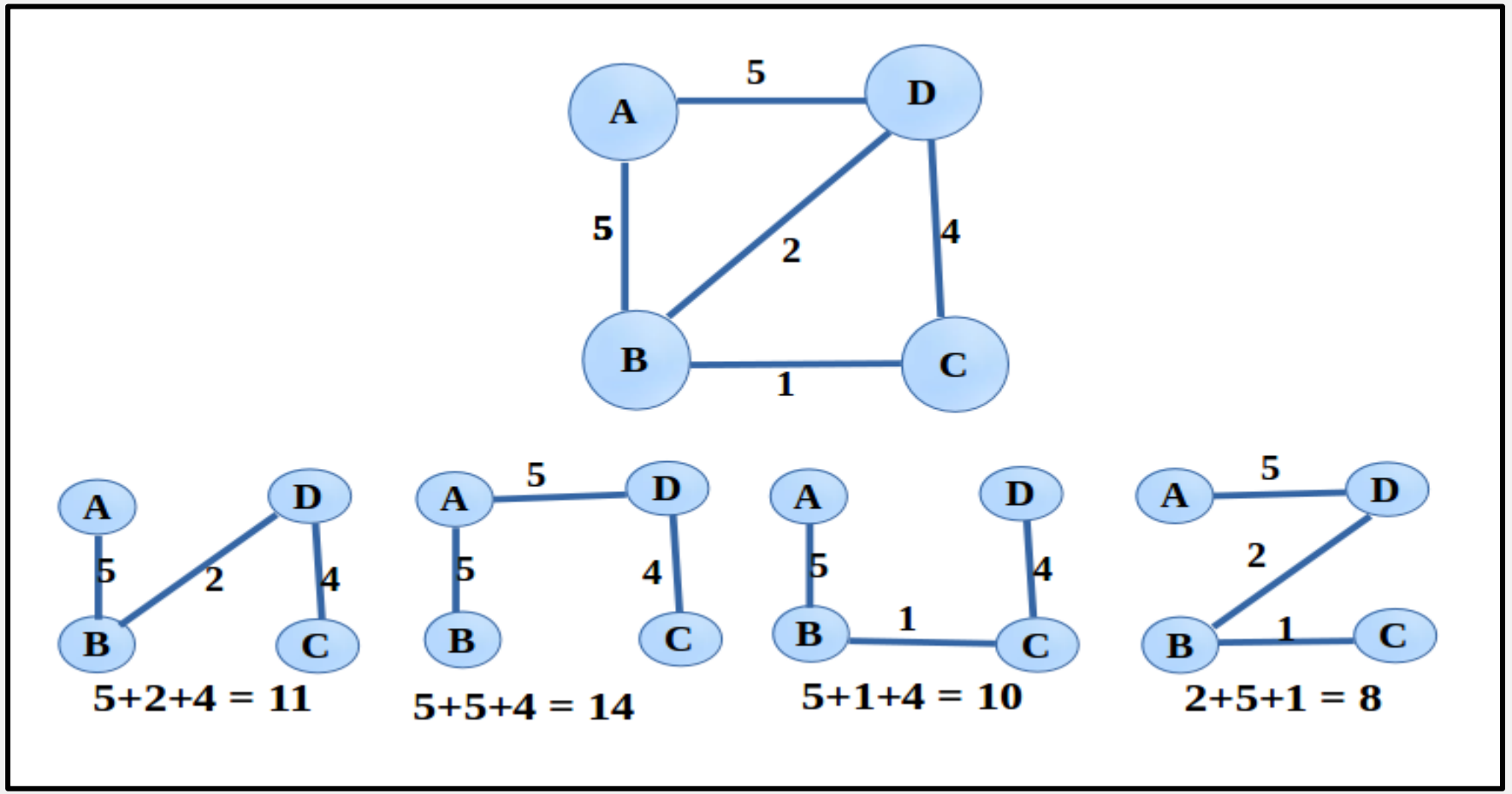
# GENERAL PROPERTIES OF SPANNING TREE

- All possible spanning trees of graph G, have the same number of edges and vertices.
- The spanning tree **does not have any cycle** (loops).
- Removing one edge from the spanning tree will make the graph disconnected.
- Adding one edge to the spanning tree will create a circuit or loop.

# MINIMUM SPANNING TREE (MST)

- A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a **connected, edge-weighted undirected graph** that connects all the vertices together, **without any cycles** and with the minimum possible total edge weight.
- That is, it is a spanning tree **whose sum of edge weights is as small as possible.**
- If each edge has a distinct weight, then there will be only one, unique minimum spanning tree.

# MINIMUM SPANNING TREE (MST)



# TWO GREEDY ALGORITHMS FOR MST

- **Kruskal's Algorithm**

  Adds edges one by one into a growing spanning tree.

- **Prim's Algorithm**

  Attaches vertices to a partially built tree by adding small- cost edges **repeatedly**.

# Kruskal's Algorithm for Finding MST

# KRUSKAL'S ALGORITHM FOR FINDING MST

- This algorithm was described by Joseph Bernard Kruskal, Jr. in 1956.
- Kruskal's Algorithm builds the spanning tree by **adding edges** one by one into a growing spanning tree. Kruskal's algorithm follows the **greedy approach** as in each iteration it finds an edge that has **the least weight** and adds it to the growing spanning tree.

# KRUSKAL'S ALGORITHM STEPS

**Step1:** Sort all the edges in **ascending** order of their weight.

**Step2:** Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far.

If a cycle is not formed, include this edge. Else, discard it.

**Step3:** Repeat step#2 until number of edges become (v-1) in the spanning tree or all vertices are included.

# COMPLEXITY OF KRUSKAL'S ALGORITHM

- In Kruskal's algorithm, most time-consuming operation is sorting because the total complexity of the Disjoint-Set operations will be **O(ElogV),** which is the overall **Time Complexity** of the algorithm.
- **Space Complexity ➔ O(E+V)**

# KRUSKAL'S ALGORITHM EXAMPLE(1)

| | |
|---|---|
| ad | 2 ★ |
| bc | 2 ★ |
| dc | 2 ★ |
| fe | 3 ★ |
| ab | 4 |
| bd | 4 |
| ce | 5 ★ |
| cf | 6 |
| df | 7 |

Create 6 disjoints sets = {a}, {b}, {c}, {d}, {e}, {f}.

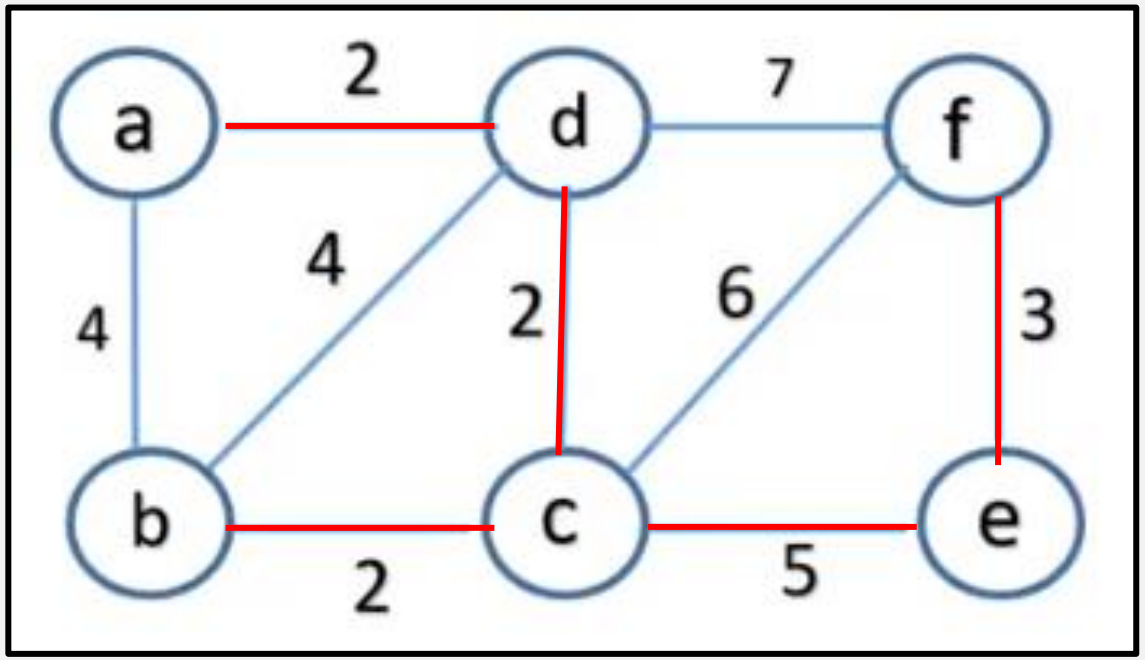
{a, d}, {b}, {c}, {e}, {f}

{a, d}, {b, c}, {e}, {f}

{a, b, c, d}, {e}, {f}

{a, b, c, d}, {e, f}

{a, b, c, d, e, f}



# KRUSKAL'S ALGORITHM EXAMPLE (2)

| | |
|---|---|
| gh | 1 ★ |
| ci | 2 ★ |
| fg | 2 ★ |
| ab | 4 ★ |
| cf | 4 ★ |
| gi | 6 |
| ah | 7 ★ |
| cd | 7 ★ |
| hi | 7 |
| bc | 8 |
| de | 9 ★ |
| ef | 10 |
| bh | 11 |
| df | 14 |

Create 9 disjoints sets =

{a}, {b}, {c}, {d}, {e}, {f}, {g}, {h}, {i}

{a}, {b}, {c}, {d}, {e}, {f}, {g, h}, {i}

{a}, {b}, {c, i}, {d}, {e}, {f}, {g, h}

{a}, {b}, {c, i}, {d}, {e}, {f, g, h}

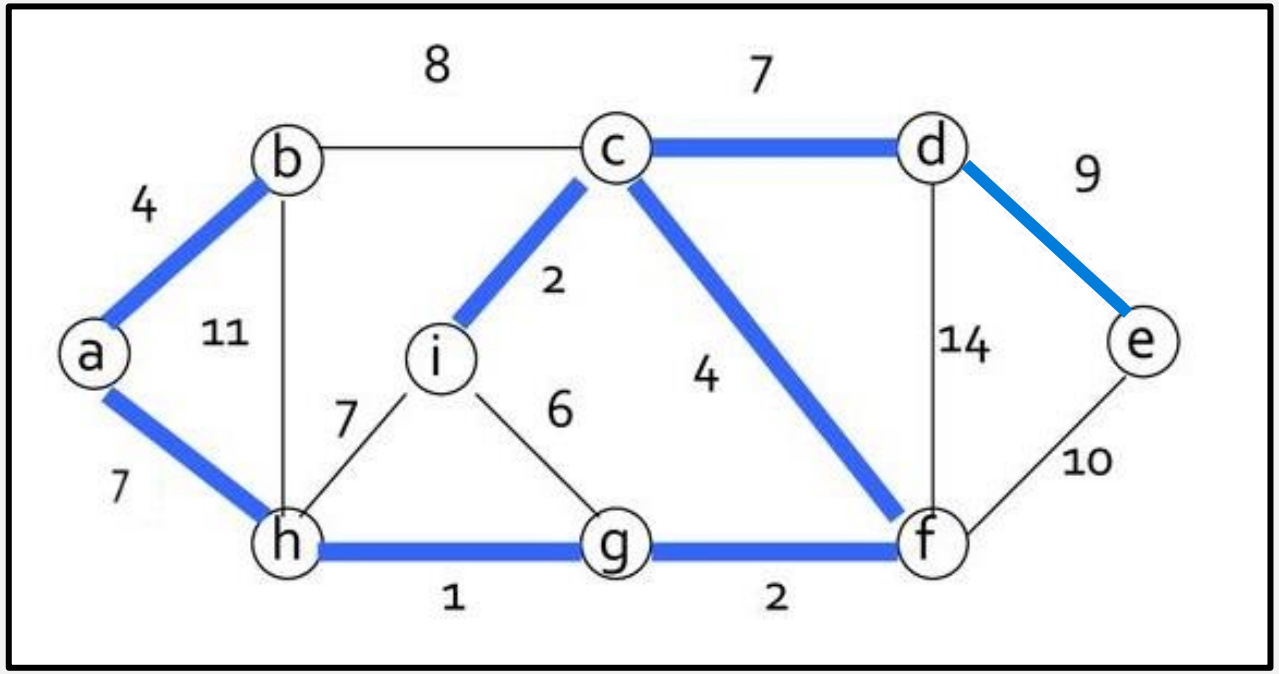
{a, b}, {c, i}, {d}, {e}, {f, g, h}

{a, b}, {c, i, f, g, h}, {d}, {e}

{a, b, c, i, f, g, h}, {d}, {e}

{a, b, c, i, f, g, h, d}, {e}

{a, b, c, i, f, g, h, d, e}



# Prim's Algorithm for Finding MST

# PRIM'S ALGORITHM FOR FINDING MST

- Prim's Algorithm also use **Greedy approach** to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from **a starting position.**
- Unlike an edge in Kruskal's, we **add vertex** to the growing spanning tree in Prim's.
- **Idea ➔** Add nearest vertex just beyond the frontier
- Prim's algorithm starts with the single node and explores all the adjacent nodes with all the connecting edges at every step. The edges with the minimal weights causing no cycles in the graph got selected.

# PRIM'S ALGORITHM STEPS

**Step1: Remove all loops and parallel edges from the given graph.** In case of parallel edges, keep the one which has the least cost associated and remove all others.

**Step2:** Initialize an MST with the **randomly** chosen vertex.

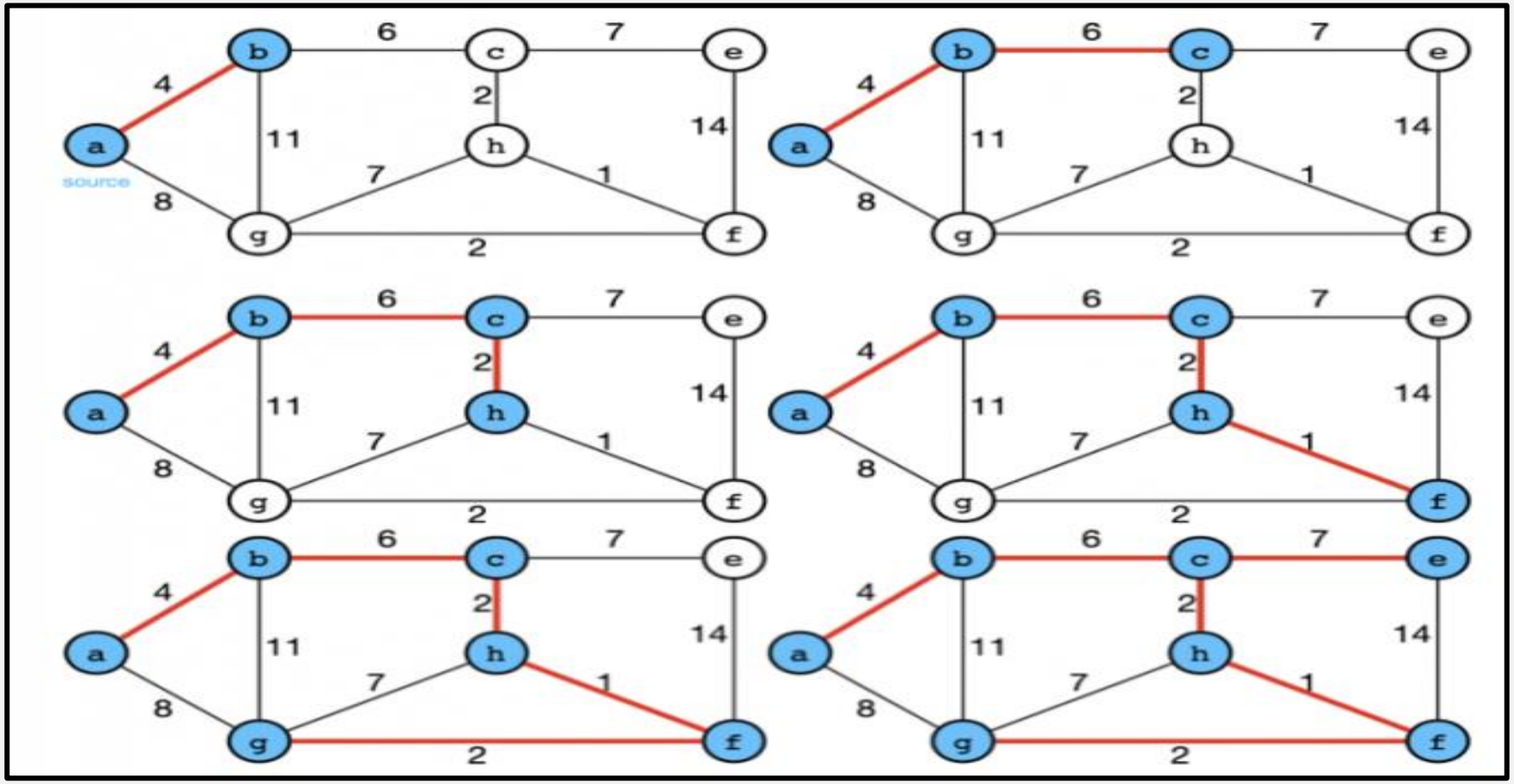
**Step3:** Find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.

**Step4:** Repeat step 3 until the minimum spanning tree is formed.

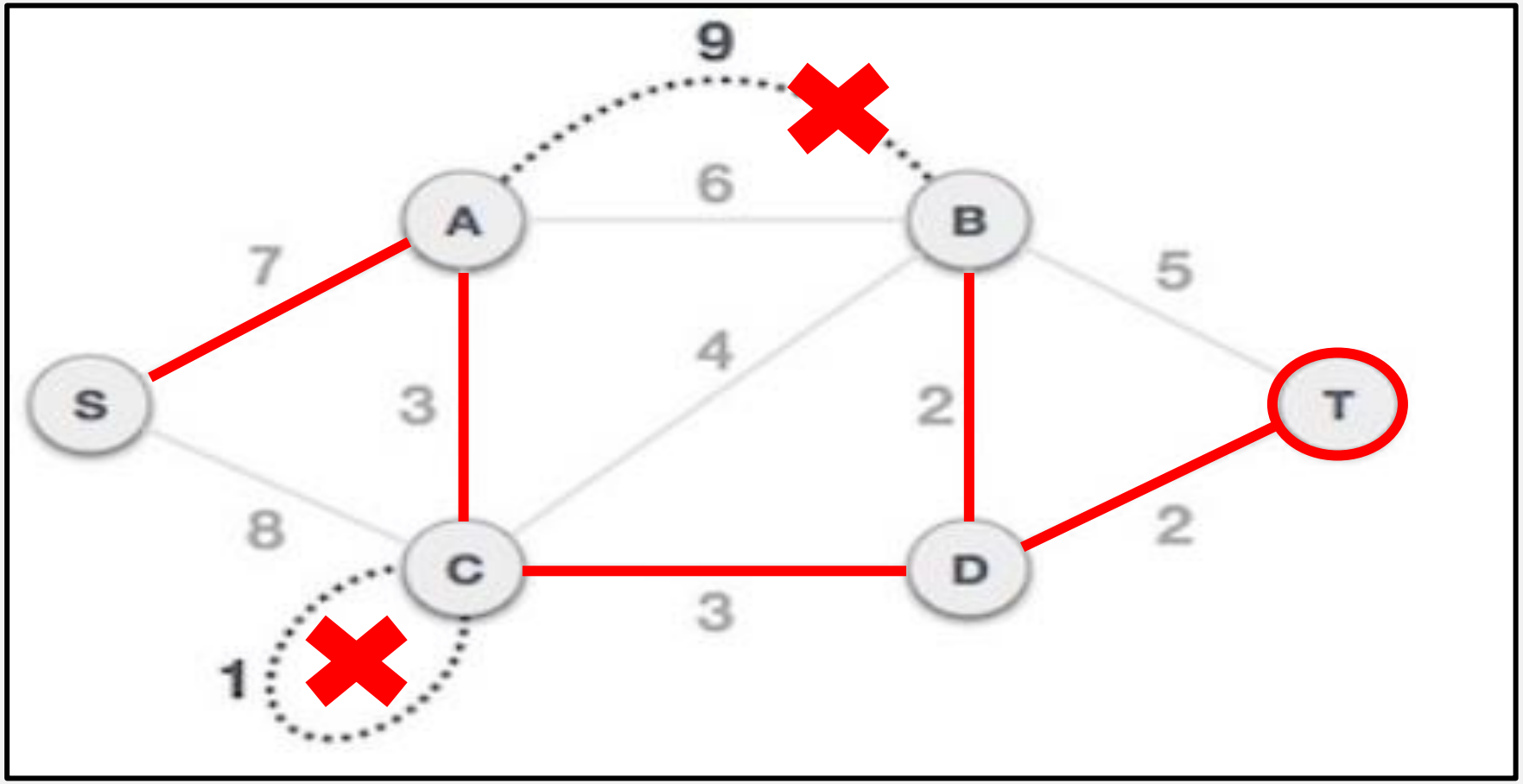
# COMPLEXITY OF PRIM'S ALGORITHM

- If the graph G is represented as an **adjacency matrix**, the time complexity is ➔ **O(V^2)**
- If the graph G is represented as an **adjacency list**, the time complexity is ➔ **O(ElogV)**

# PRIM'S ALGORITHM EXAMPLE (1)



# PRIM'S ALGORITHM EXAMPLE (2)



# Shortest Path Algorithms in Graphs

# SHORTEST PATH ALGORITHMS IN GRAPHS

- The shortest path problem is about finding a path between two vertices in a graph such that the total sum of the weights of the edges is **minimum (Optimization Problem).**

- There are different methods for Finding the Shortest Path in a Graph:

1. Depth-First Search (DFS)
2. Breadth-First Search (BFS)
3. Dijkstra's algorithm
4. Bellman-Ford Algorithm

- The choice of the proper algorithm depends on the **use-case.**

# SHORTEST PATH ALGORITHMS IN GRAPHS

- The problem for finding the shortest path can be categorized as:

1. **Single-source the shortest path**: In this, the shortest path is calculated **from a source vertex to all other vertices present inside the graph**.
2. **Single-destination the shortest path:** In this, the shortest path is calculated **from all vertices in the directed graph to a single destination vertex.** This can be converted into a single pair with the shortest path problem by reversing the edges of the directed graph.
3. **All pairs the shortest path:** In this, the shortest path is calculated **between every pair of vertices.**

# Dijkstra’s Algorithm

# DIJKSTRA'S ALGORITHM

- Dijkstra's algorithm has many variants but the most common one is to find the shortest paths from the source vertex to all other vertices in the graph.

**Single Source Shortest Path Algorithm**

- Any vertex can be selected as a source.
- Dijkstra uses **greedy approach**.
- It can be applied to directed and undirected connected Graphs.
- **Dijkstra doesn't work for Graphs with negative weight edges.**

# DIJKSTRA'S ALGORITHM STEPS

**Step1:** Mark your **selected initial node** with a current distance of **0 and the rest with infinity.**

**Step2:** Set the non-visited node with the smallest current distance as the current node.

**For each neighbor N of your current node C:**

Add the current distance of C with the weight of the edge connecting C-N.

If it's smaller than the current distance of N, set it as the new current distance of N.

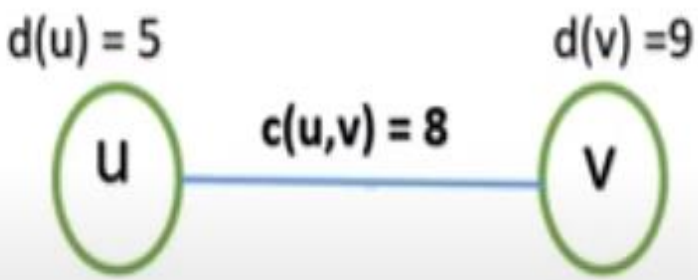
**Step3:** Mark the current node C as visited.

**Step4:** If there are non-visited nodes, go to step 2.

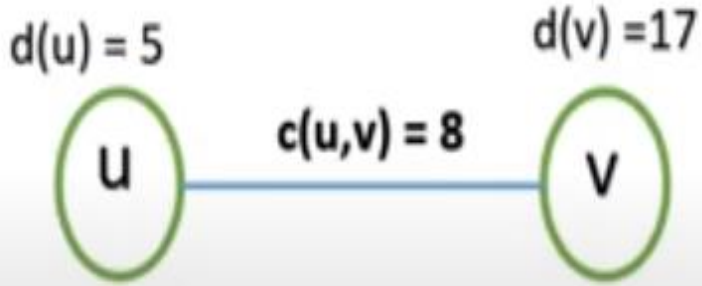
# RELAXATION PROCESS

If (d(u) + c(u,v) < d(v)) Then

d(v) = d(u) + c(u, v)




(5 + 8) < 9 ➔ (13) < 9 NO!!, so d(v) = 9

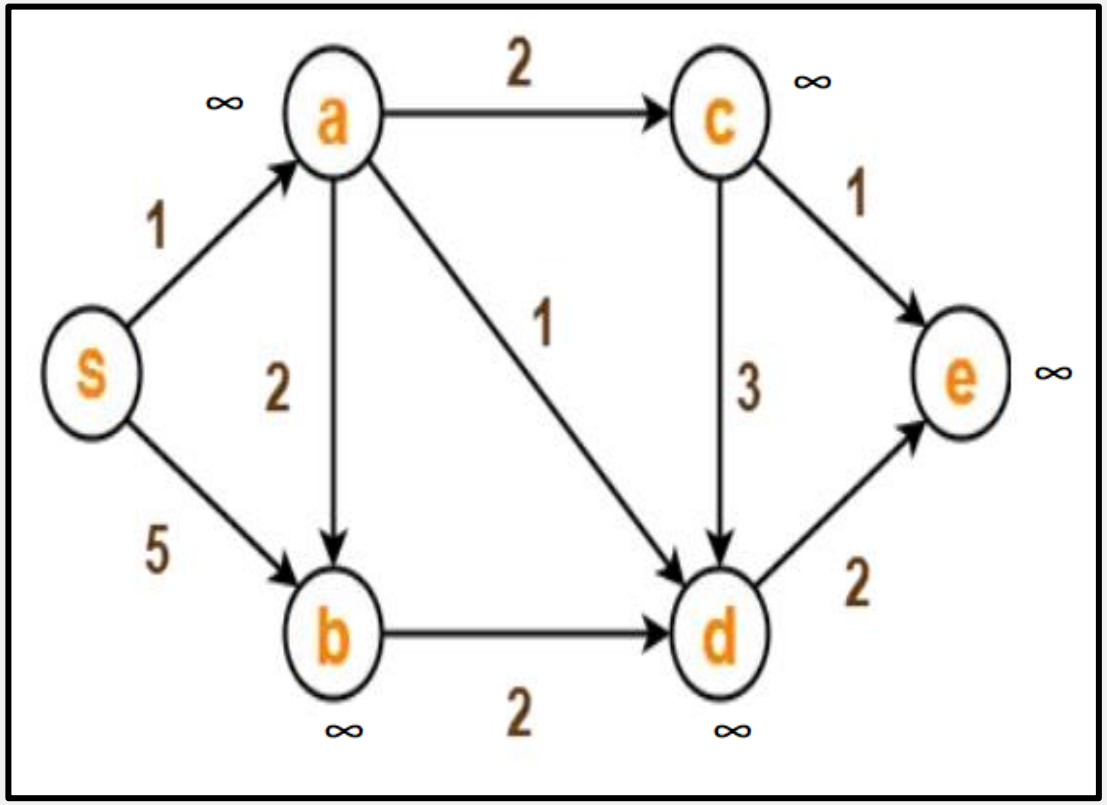


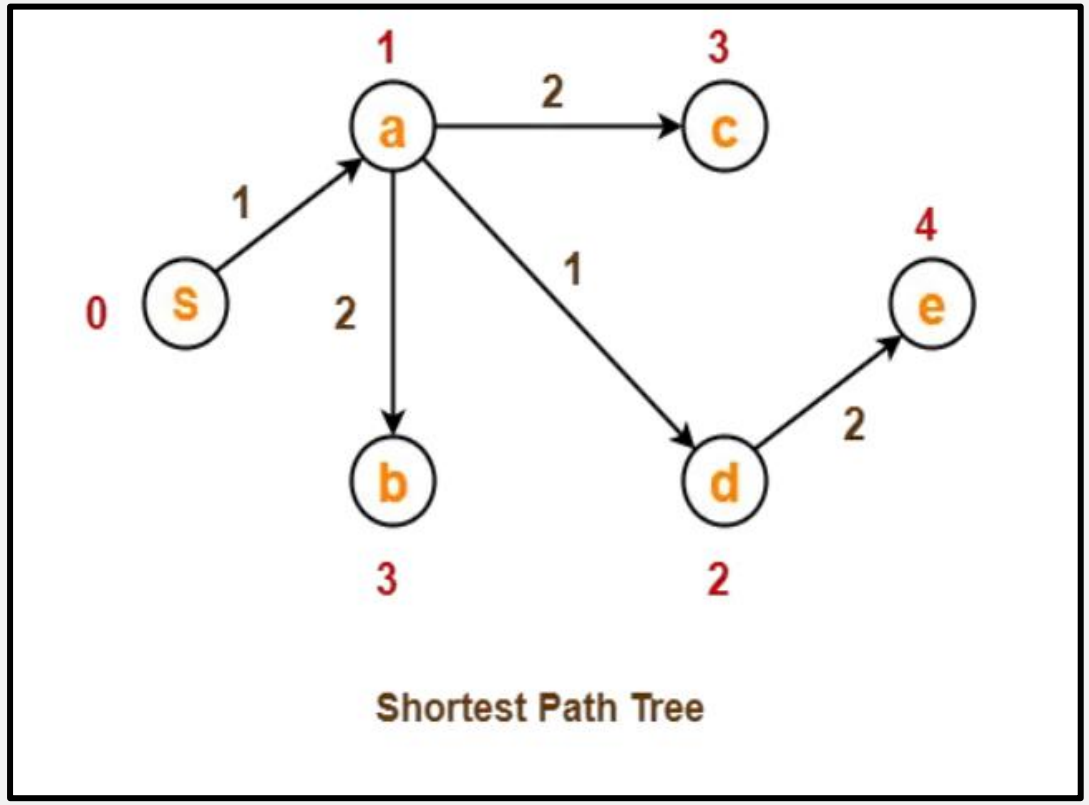

(5 + 8) < 17 ➔ (13) < 17 YES!!, so d(v) = 13

# COMPLEXITY OF DIJKSTRA'S ALGORITHM

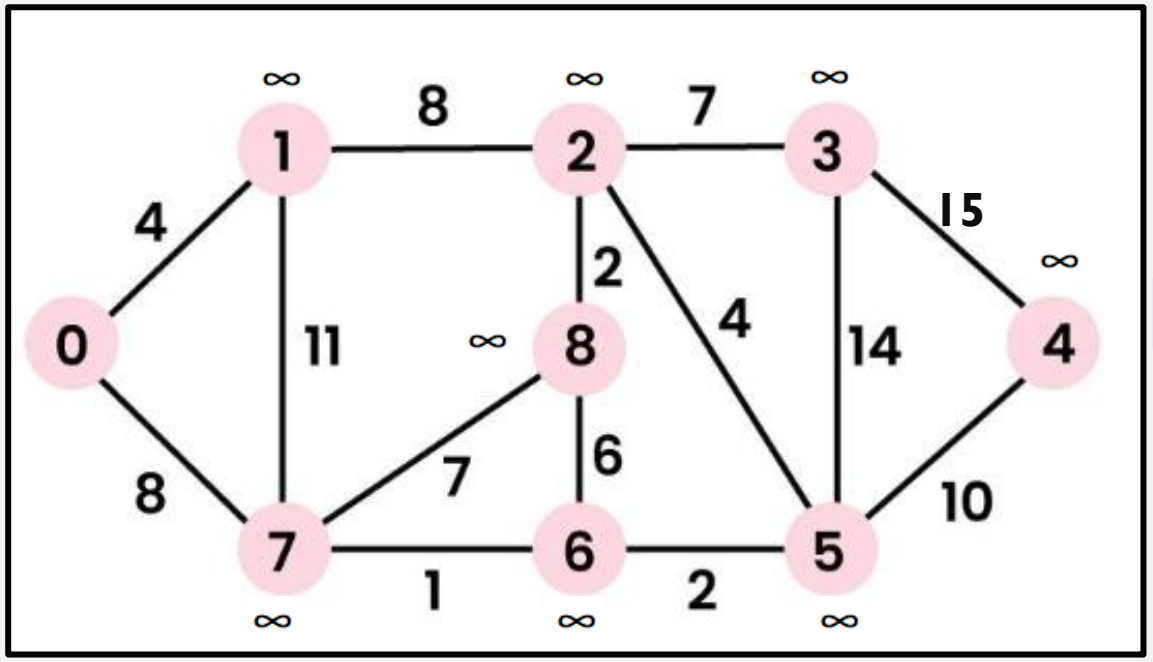
- If the graph G is represented as an **adjacency matrix**, the time complexity is ➔ **O(V^2)**
- If the graph G is represented as an **adjacency list**, the time complexity is ➔ **O(ElogV)**
- **Space complexity ➔ O(E + V)**

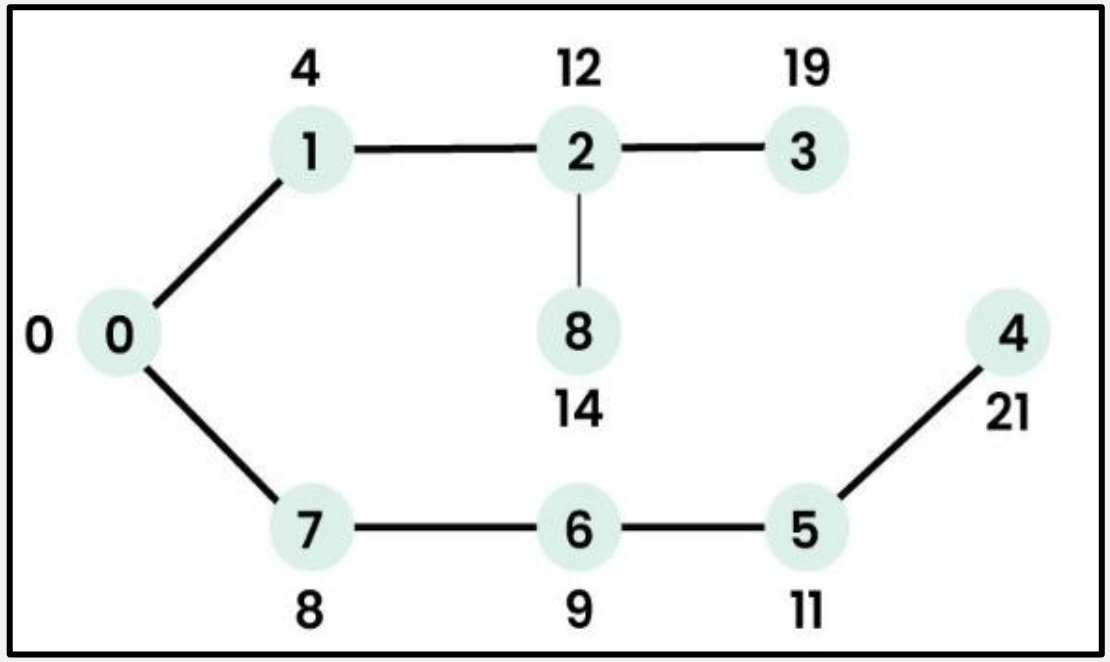
# DIJKSTRA'S ALGORITHM EXAMPLE (1)



# DIJKSTRA'S ALGORITHM EXAMPLE (2)



# Bellman-Ford Algorithm

# BELLMAN-FORD ALGORITHM (BF)

- BF is a single source shortest path algorithm.
- **BF is slower than Dijkstra's Algorithm, it works in the cases when the weight of the edge is negative, and it also finds negative weight cycle in the graph.**
- The idea of this algorithm is to **relax all the edges** of the graph one-by-one in some **random** order at most (n-1) times.
- **BF uses dynamic programming approach.**
- **The time complexity is: O(VE) or O(V^2).**

# BELLMAN-FORD ALGORITHM STEPS

**Step1:** determine: **S** ➔ Starting node, **E** ➔ # of edges, **V** ➔ # of nodes, and **D** ➔ Array that tracks the best distance from S to all nodes

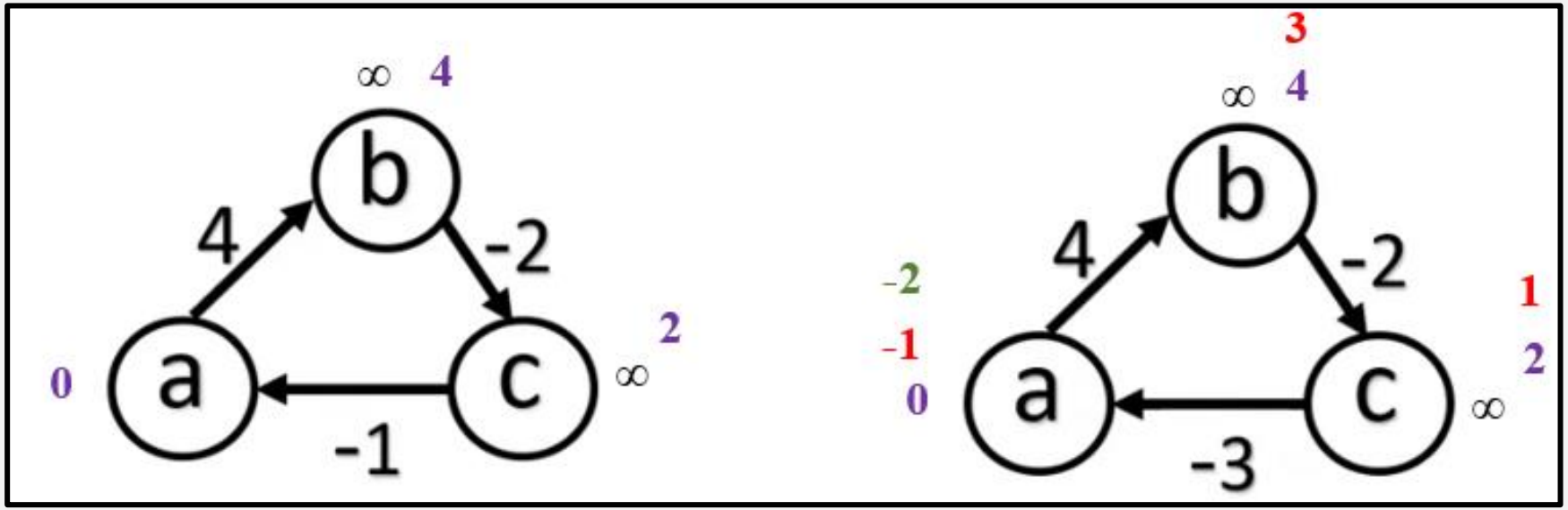
**Step2:** set D[S] to 0

**Step3:** Set every entry in D to ∞

**Step4:** Relax each edge V-1 times

# DETERMINE NEGATIVE LOOP USING BELLMAN-FORD ALGORITHM

- Bellman-ford algorithm can determine a negative loop (cyclic), if the sum of edges is negative then the loop is **negative (infinite loop)** else the loop is not negative.
- **Example:**




$$\sum(4-2-1)=1$$

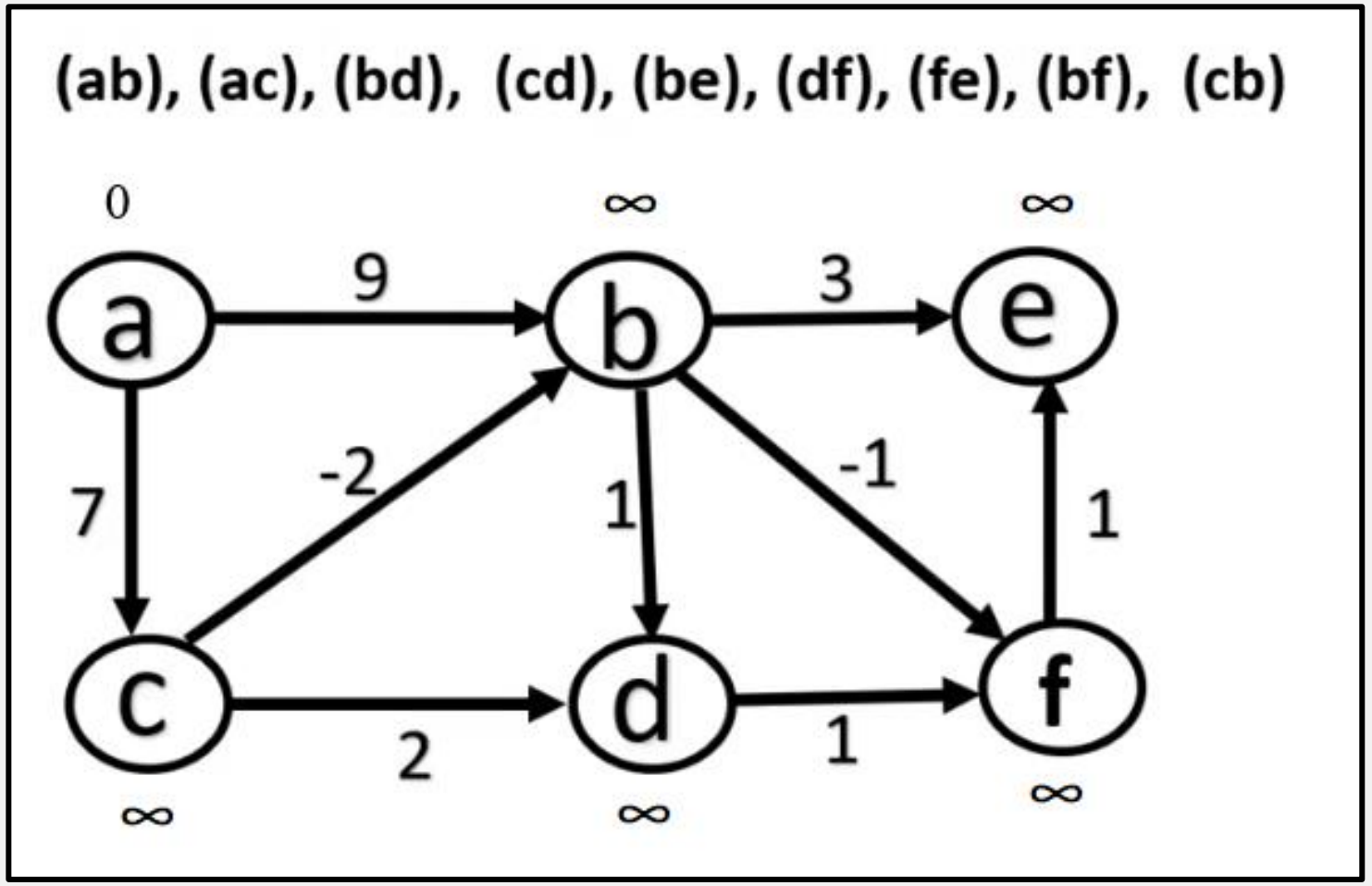
**Non-Negative Loop**

$$\sum(4-2-3)=-1$$

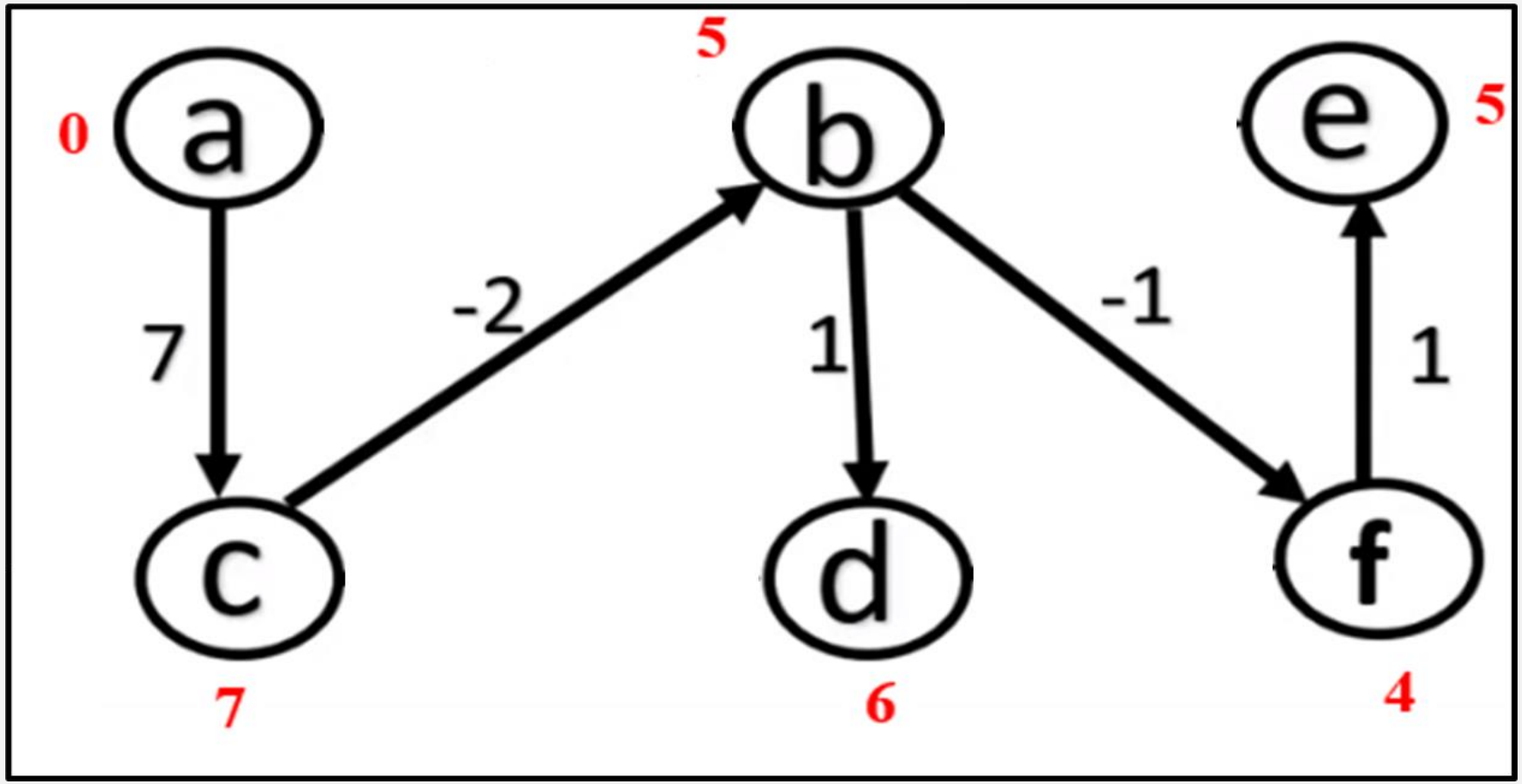
**Negative Loop**

# BELLMAN-FORD ALGORITHM EXAMPLE(1)

| | A | B | C | D | E | F | |
|---|---|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | |
| 1 | 0 | 5 | 7 | 9 | 11 | 8 | |
| 2 | 0 | 5 | 7 | 6 | 8 | 4 | |
| 3 | 0 | 5 | 7 | 6 | 5 | 4 | Same, Stop |
| 4 | 0 | 5 | 7 | 6 | 5 | 4 | |
| 5 | 0 | | | | | | |



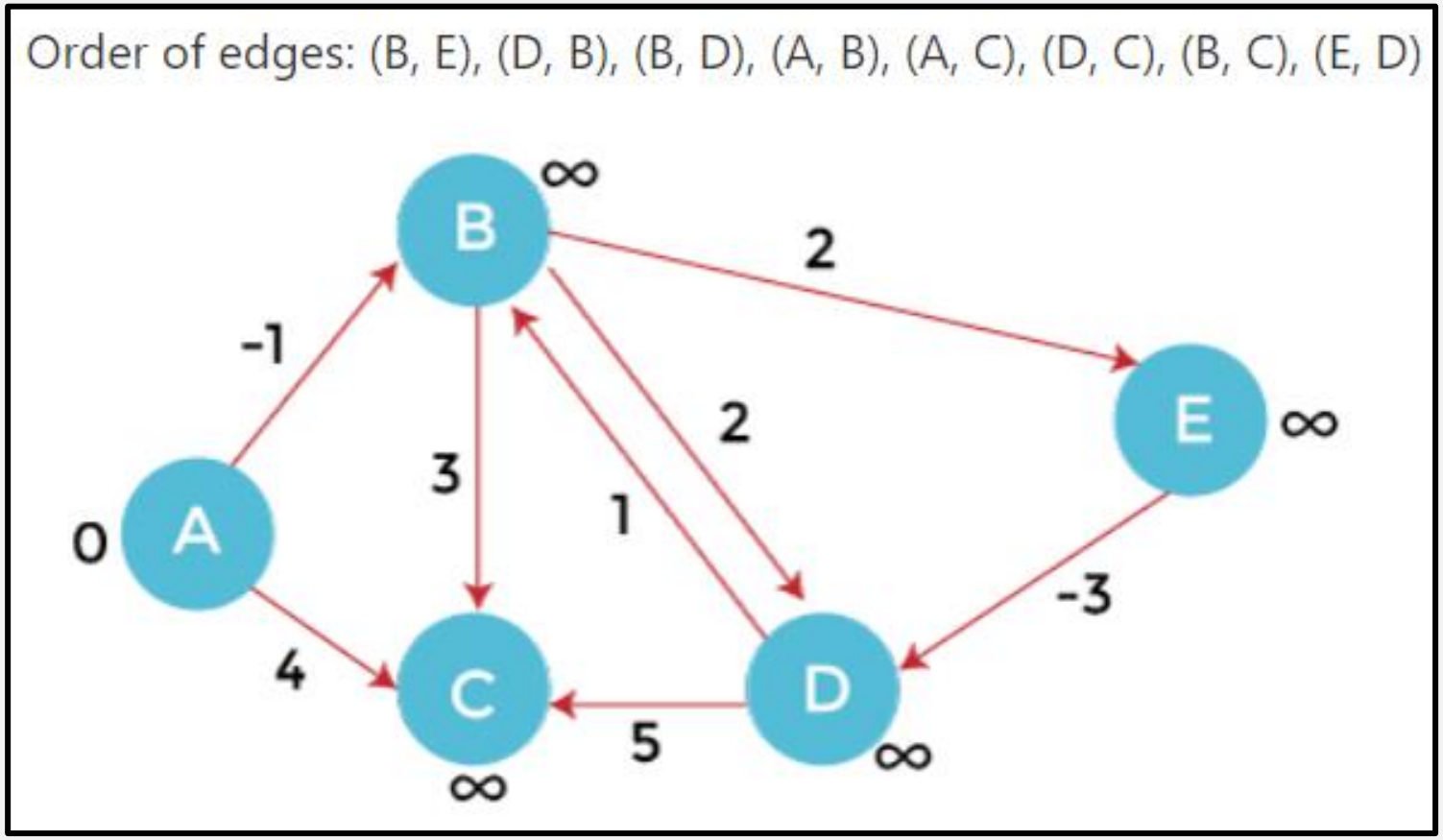
# BELLMAN-FORD ALGORITHM EXAMPLE (1)



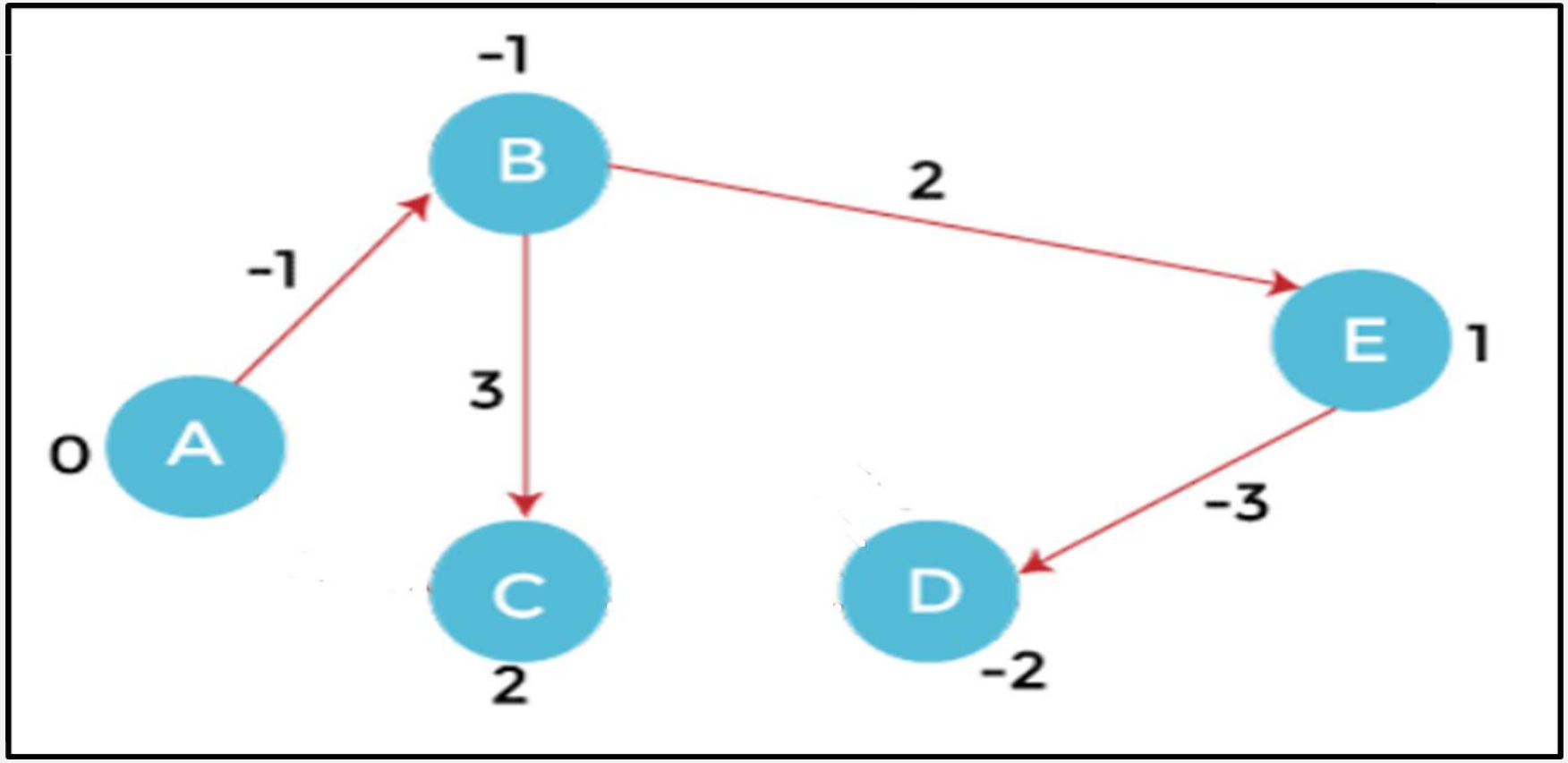
# BELLMAN-FORD ALGORITHM EXAMPLE (2)

| | A | B | C | D | E |
|---|---|---|---|---|---|
| | 0 | ∞ | ∞ | ∞ | ∞ |
| **1** | 0 | -1 | 2 | ∞ | ∞ |
| **2** | 0 | -1 | 2 | -2 | 1 |
| **3** | 0 | -1 | 2 | -2 | 1 |
| **4** | 0 | | | | |

Same, Stop



# BELLMAN-FORD ALGORITHM EXAMPLE (2)



# The Difference Between MST and The Shortest Path

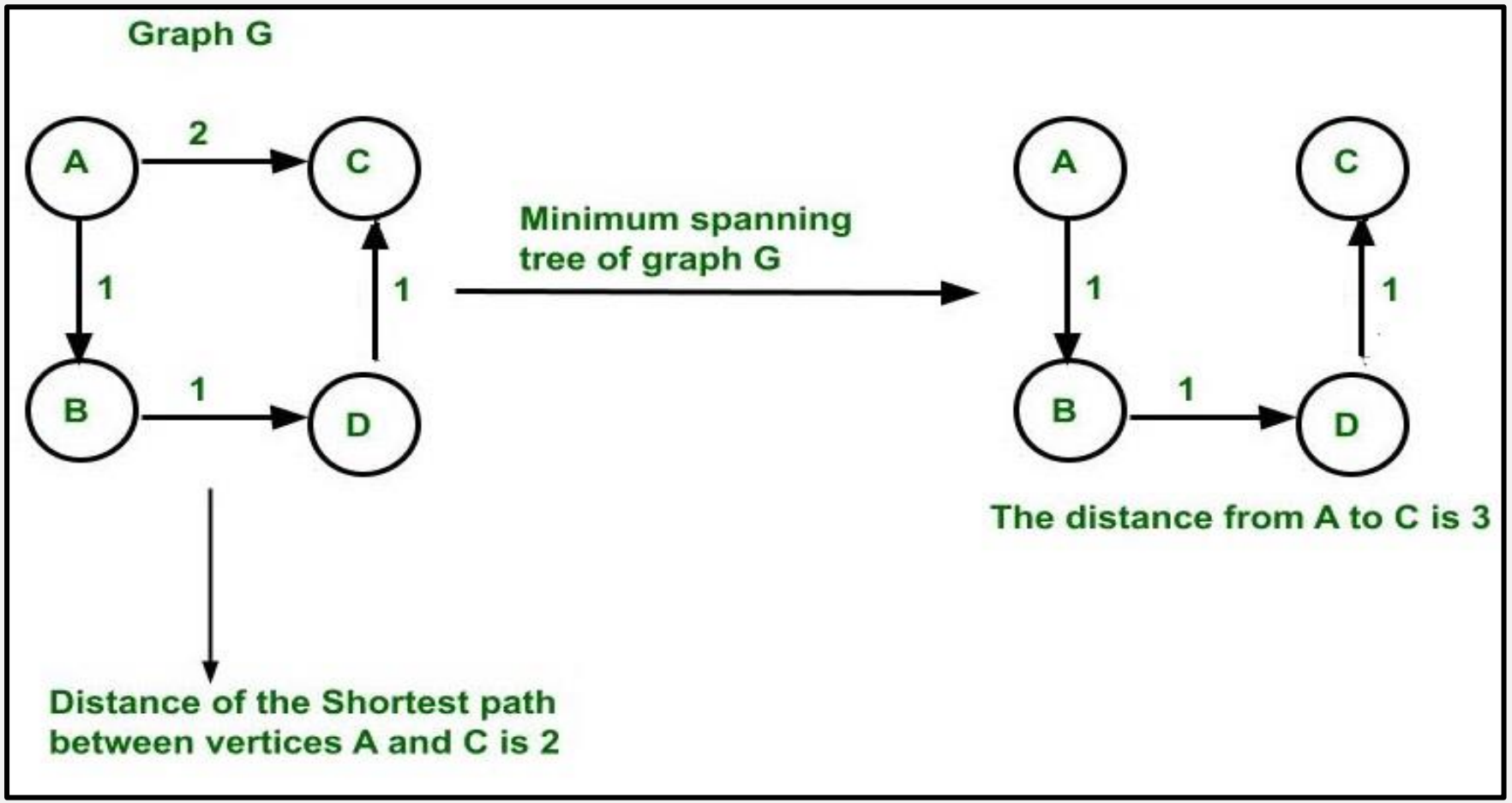
# THE DIFFERENCE BETWEEN MST AND THE SHORTEST PATH

| MST | Shortest Path |
| --- | --- |
| In MST there is **no source and no destination**, but it is the subset (tree) of the graph(**G**) which connects all the vertices of the graph **G** without any cycles and the minimum possible total edge weight. | There is a **source and destination**, and one need to find out the shortest path between them |
| **Graph (G) should be connected, undirected, edge-weighted, labeled.** | It is **not necessary** for the Graph **(G)** to be connected, undirected, edge-weighted, labeled. |
| Here **relaxation of edges** is **not performed** but here the minimum edge weight is chosen one by one from the set of all edge weights (sorted according to min weight) and the tree is formed by them (i.e. there should not be any cycle). | Here the **relaxation of edges** is **performed**. |

# THE DIFFERENCE BETWEEN MST AND THE SHORTEST PATH

| MST | Shortest Path |
| --- | --- |
| A minimum spanning tree can be formed but **negative weights edge** cycles are not generally used. Using the cycle property of MST, the minimum edge weight among all the edge weights in the negative edge cycle can be selected. | The **negative edge cycle** can be detected using the Bellman-Ford algorithm. |
| It is used in **network design** and in **circuit design applications**, and many more. | It is used to find out direction between physical locations like in **Google Maps.** |

# THE DIFFERENCE BETWEEN MST AND THE SHORTEST PATH



# END OF CHAPTER 6 – PART2