**Graph Algorithm**

A graph is an abstract data structure that represents the relationship between multiple nodes (entities) by connecting the nodes with edges.

G = (V, E). Here, G is a graph, which is a pair of two sets, where:

1. V is a finite set of vertices (nodes), and
2. E is a finite set of an ordered pair of the form (u, v), called an edge. The pair (u, v) indicates that there is an edge from vertex u to vertex v.

Graphs play a critical role in the application of social networks, transportation networks, etc. Certain applications of the graph data structure include the following:

1. Graphs help with identifying contacts on social networking websites, such as Facebook; the nodes can be different users and the edges represent the relationships between them.
2. Graphs help with finding the shortest distance between any two given locations; the nodes can be the different locations in a city and the edges represent the routes available between the locations.

Graph data structures do not have any restrictions like those of the tree data structure on the way different nodes are connected to each other. Tree data structures have usage restrictions in representing real-world scenarios, as each child node can have one parent node only. Let’s take a look at the two data structures in the diagram given below.

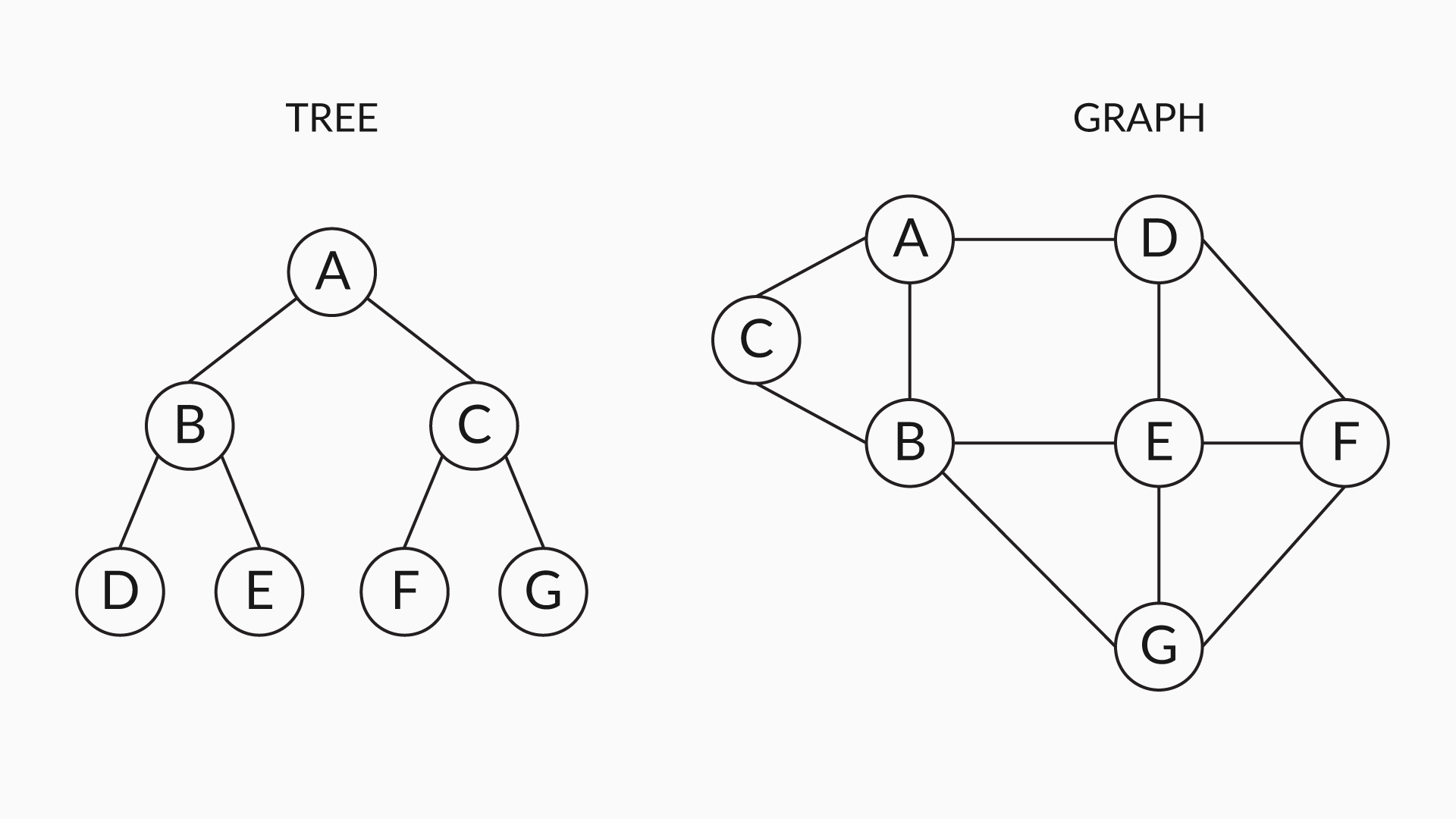


Figure 1

So, you have now been introduced to the concepts of graphs and have also looked at various real-world scenarios where graph data structures are used extensively.

Q1: Can you think of any other possible real-life examples where you can use graphs to define the relationship between objects or things in a scenario?

Ans:

1. Graphs are used in Google Maps where different locations are represented as nodes, and the roads connecting different locations act as edges between these nodes.
2. Recommendations on e-commerce websites are generated by the graph theory. When a person checks out a particular category of product, products of similar types are displayed to them as suggestions. Here, the products act as nodes, and the similarity between them forms a relationship and connects them to an edge that leads to related products (nodes).

These are two of many real-life examples where you can use graphs to depict scenarios.

The chart given below shows the classification of graphs.

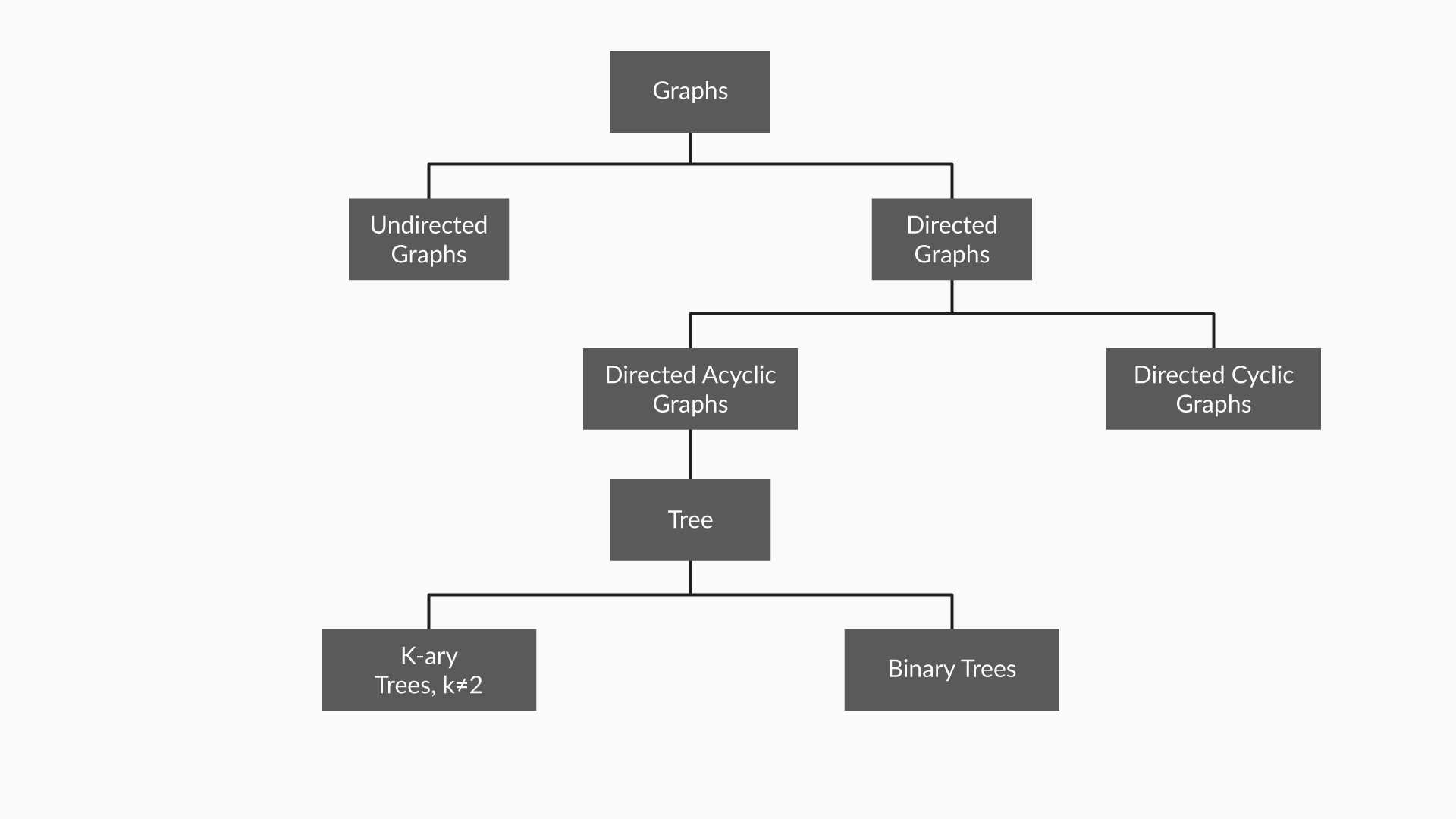
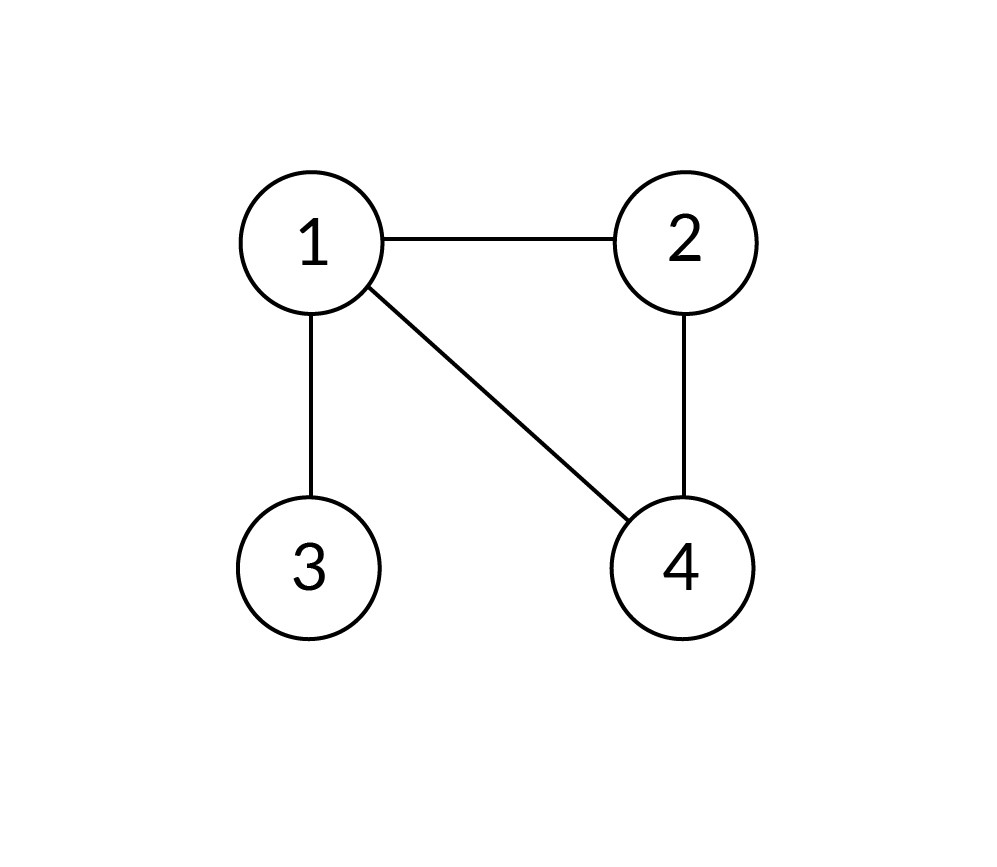


Figure 2

**Undirected graphs:** Graphs that show a symmetrical relationship between two connected nodes that are paired by an edge that represents a simple line are called undirected graphs.



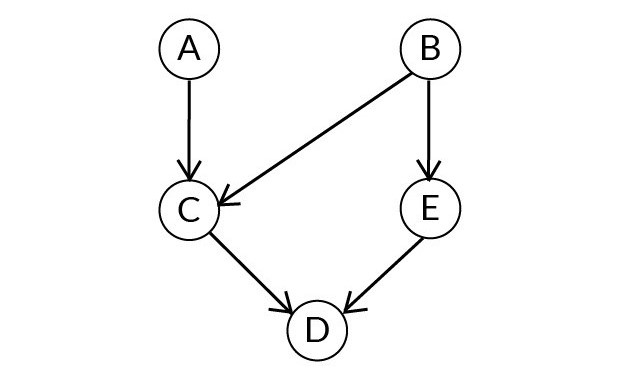
graph

G for the above graph is (V, E), where:

V = {1, 2, 3, 4} and

E = {(1, 2), (2, 4), (4, 1), (1, 3)}.

**Directed graphs:** Graphs that show an asymmetrical relationship between two connected nodes that are paired by an edge that indicates the direction of the relationship from one node to the other are called directed graphs.



Directed Graph

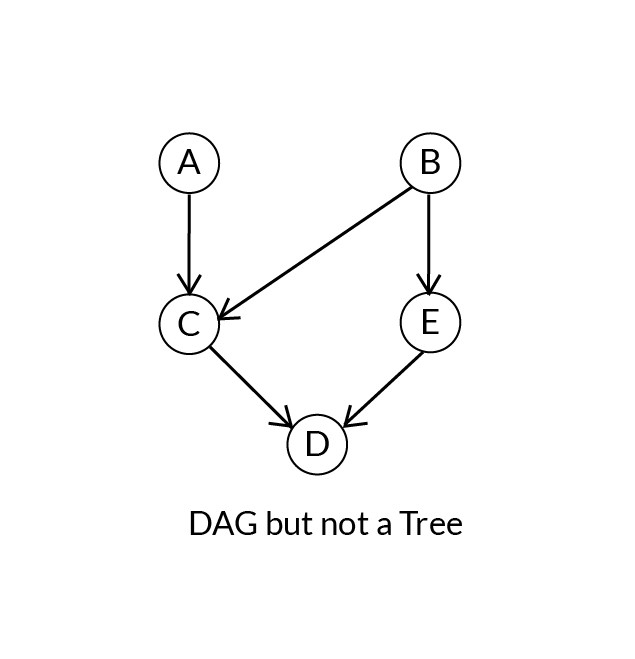
**Note:** The pair representing the edge is ordered, and (u, v) ≠ (v, u) in the case of a directed graph (digraph). In the case of an undirected graph, the order does not matter.

G for the graph above is (V, E), where:

V = {A, B, C, D, E} and

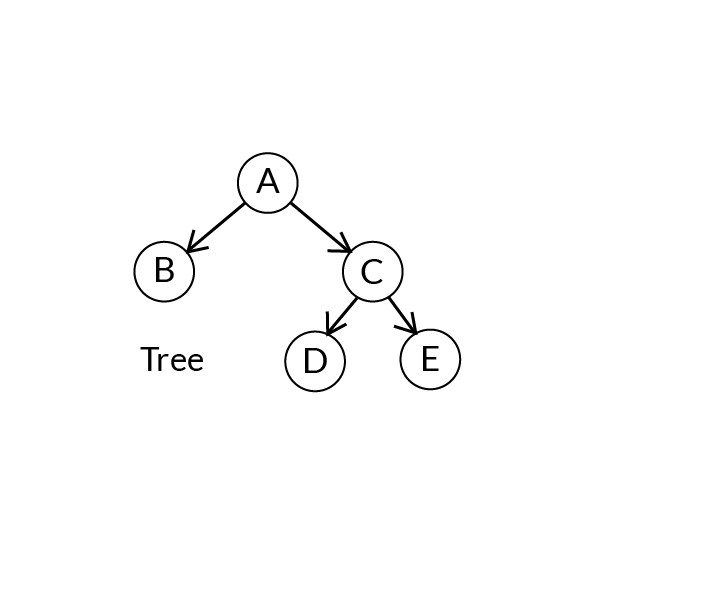
E = {(A, C), (B, C), (C, D), (E, D), (B, E)}.

**Directed acyclic graphs (DAGs):** These graphs fall under the sub-category of directed graphs. The only difference between a directed graph and a DAG (a directed acyclic graph) is that DAGs do not have cycles; this means if you start from any node in the graph and traverse through its connecting edges, then you can never return to the same node where you started.



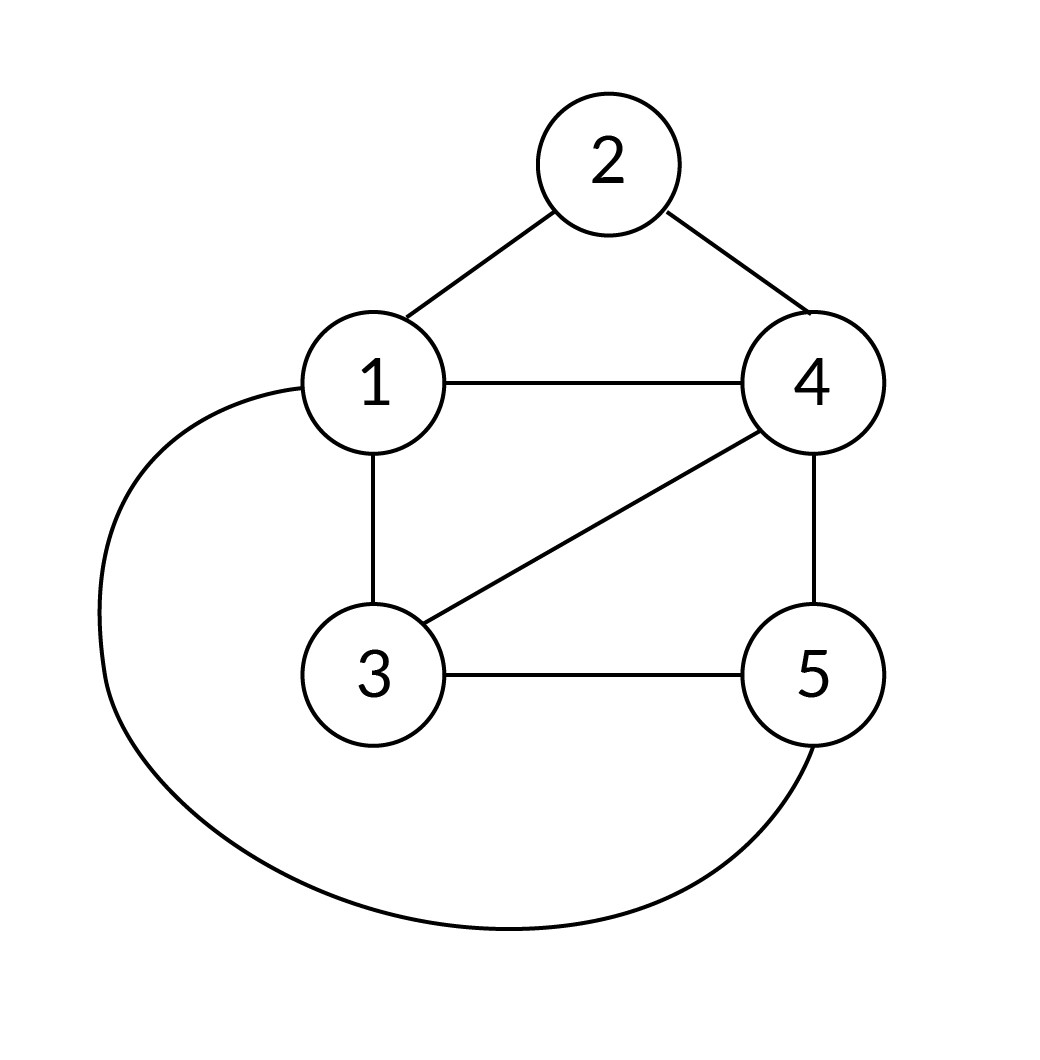
DAG

**Trees:** These are restricted forms of graphs, and they fit the category of directed acyclic graphs with the restriction that each child node can have only one parent node in the structure.



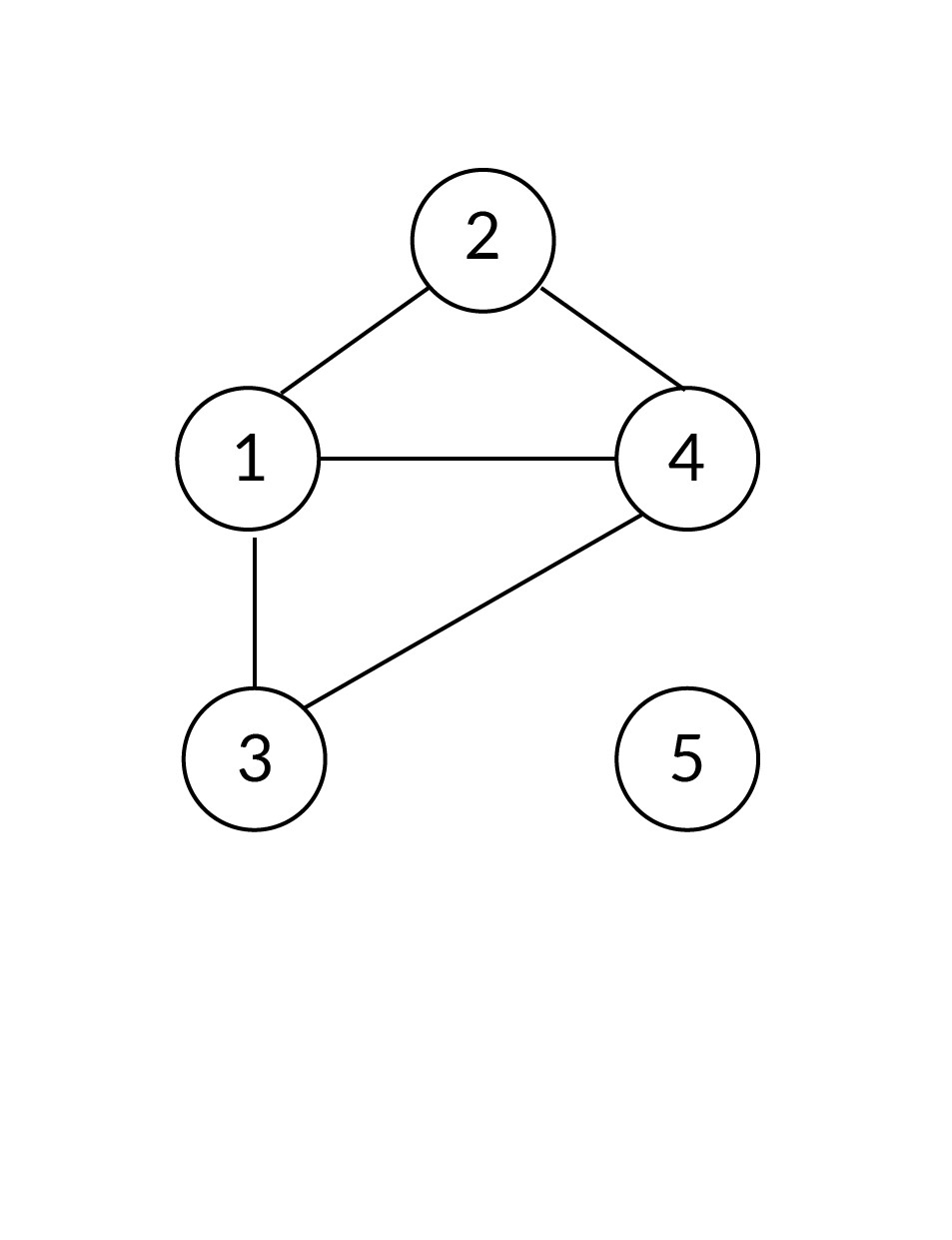
Trees

**Connected graph:**A graph is connected if a path exists from every vertex to every other vertex.



Connected

**Disconnected graph:**A graph is disconnected if there exist at least two nodes such that there is no path connecting them.



Disconnected

There does not exist any path from the set of vertices {1, 2, 4, 3} to vertex {5}.

Q2: What is the most significant difference between a directed graph and a directed acyclic graph?

Ans: Graphs, in general, contain cycles, i.e., if you start from a certain node and traverse through the graph along the connecting edges, then you will come back to the same node where you started.

Q3: How will you differentiate directed acyclic graphs from trees?

Ans: Directed acyclic graphs can be differentiated from trees by calculating the number of parent nodes connected with each child node.

The characteristic of the tree data structure is that each child node would have only one parent node. However, the child nodes of a directed acyclic graph may have more than one parent node.

The tree data structure is a restricted form of graphs, and it fits the category of directed acyclic graphs with the restriction that a child node can have one parent node only.

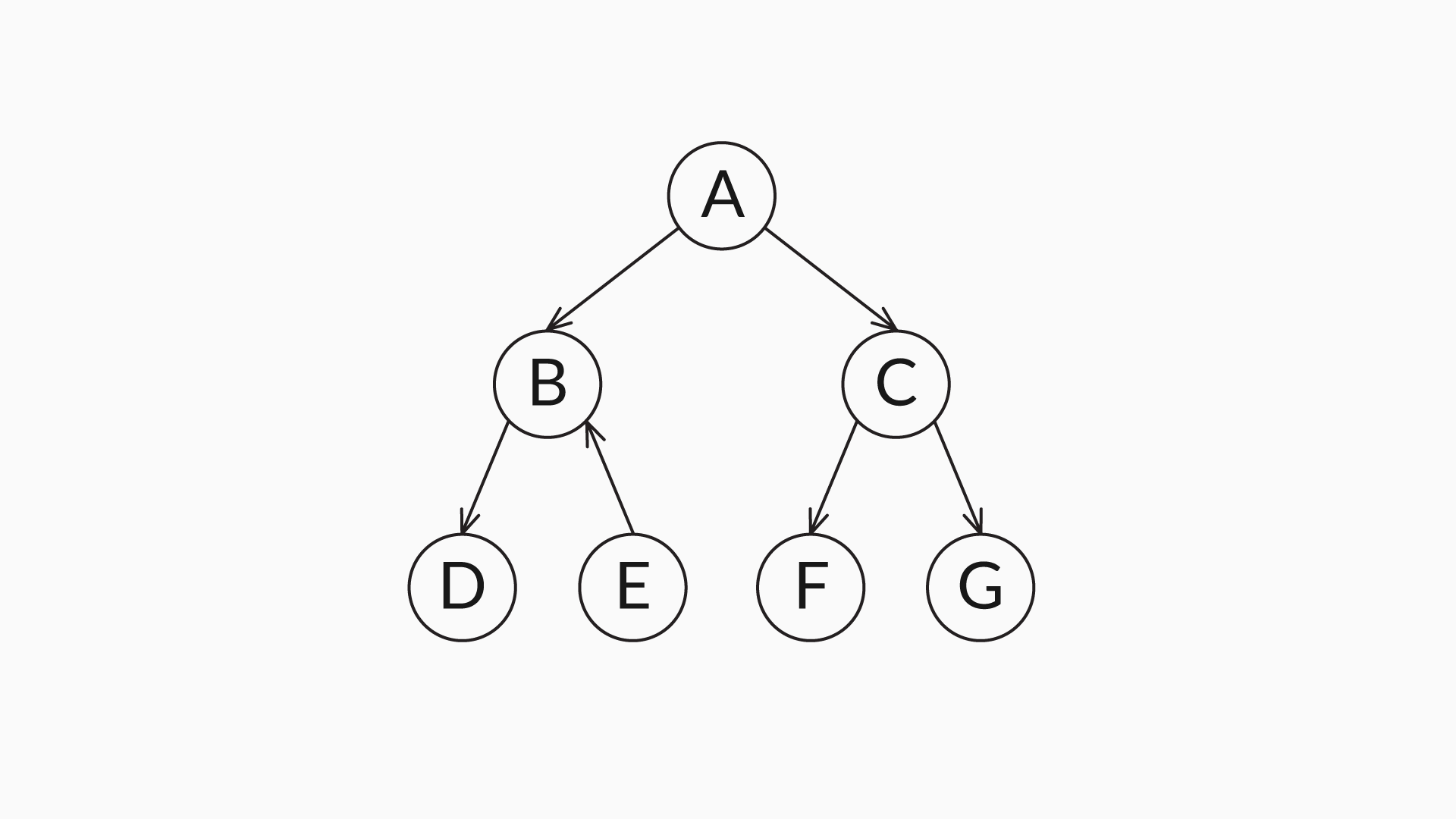
Q4: List out the differences between the following two abstract data structures: trees and graphs.

Ans: The differences between trees and graphs are listed in the table given below.

|  |  |
| --- | --- |
| **Trees** | **Graphs** |
| Trees consist of only one root node, and only one parent node can be assigned to each child node | Graphs do not have any specific root node and there is no such parent-child relation. Nodes are connected via edges wherever there is a relation between two nodes |
| The tree data structure is a restricted form of graphs and does not contain any cycles or self-loops | Graphs can be either cyclic or acyclic |
| The number of edges in a tree is always n-1, where n indicates the number of nodes | There is no restriction on the number of edges connecting the nodes, and the number of edges depends upon the graph’s structure |

#### Q5: Graphs

Identify the category in which the following diagram can be considered.



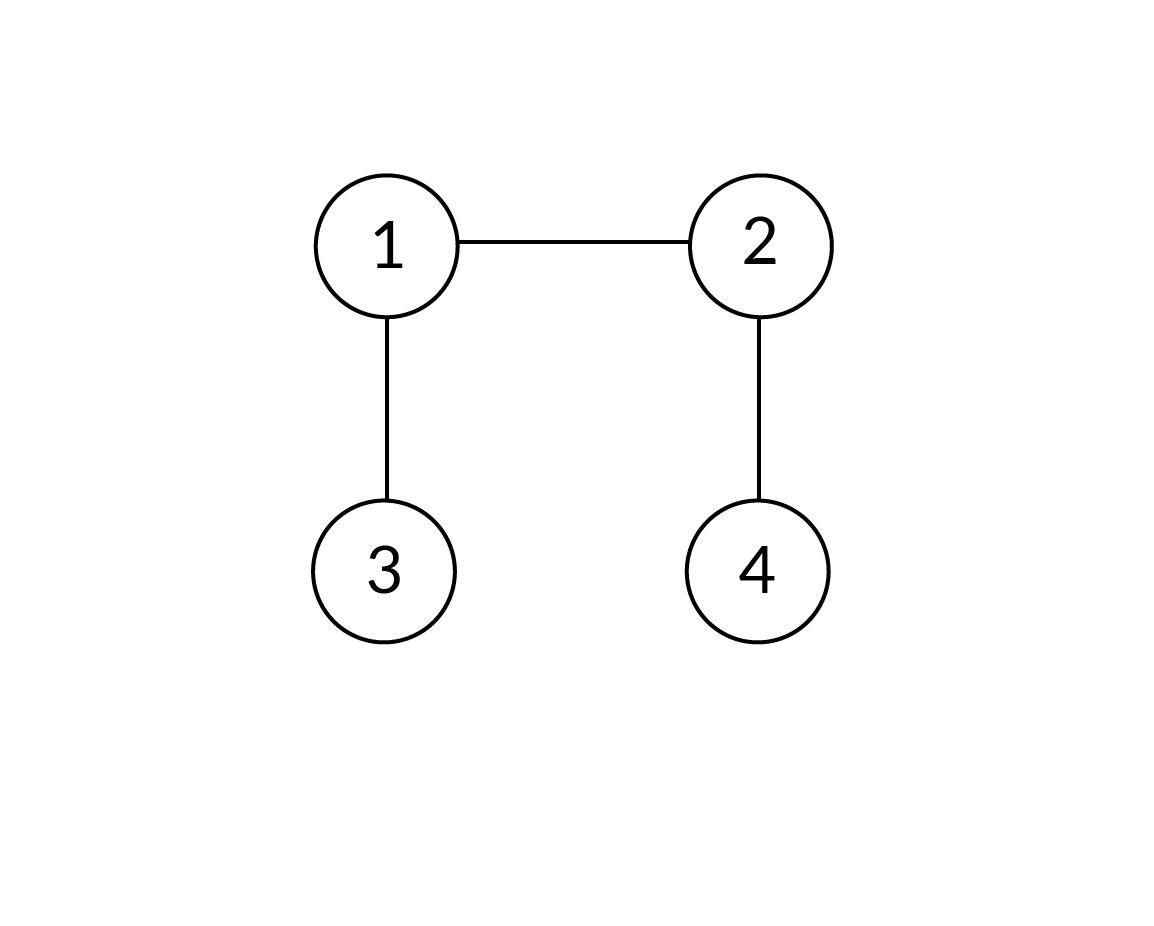
Ans: Directed acyclic graph

**✓ Correct**

**Feedback:**

In this structure, node B has two parent nodes: node E and node A. One of the characteristics of a tree data structure is that a child node will have one parent node only. This structure does not consist of any cycles. Hence, this is a directed acyclic graph, and not a tree.

Q6: Which of the statements below is/are a correct representation of the edge set of the given graph?



Ans: {(1, 2), (3, 1), (2, 4)}

**✓ Correct**

**Feedback:**

There are three edges in the given graph. There is an edge between 1 and 2, 3 and 1, and 2 and 4 each. Since the graph is undirected, the order of the vertices in the pair representing the edge does not matter. The graph does not follow any specific order in representing the set of edges.



{(2, 4), (3, 1), (1, 2)}

**✓ Correct**You missed this!

**Feedback:**

There are three edges in the given graph. There is an edge between 2 and 4, 3 and 1, and 1 and 2 each. Since the graph is undirected, the order of the vertices in the pair representing the edge does not matter. The graph does not follow any specific order in representing the set of edges.



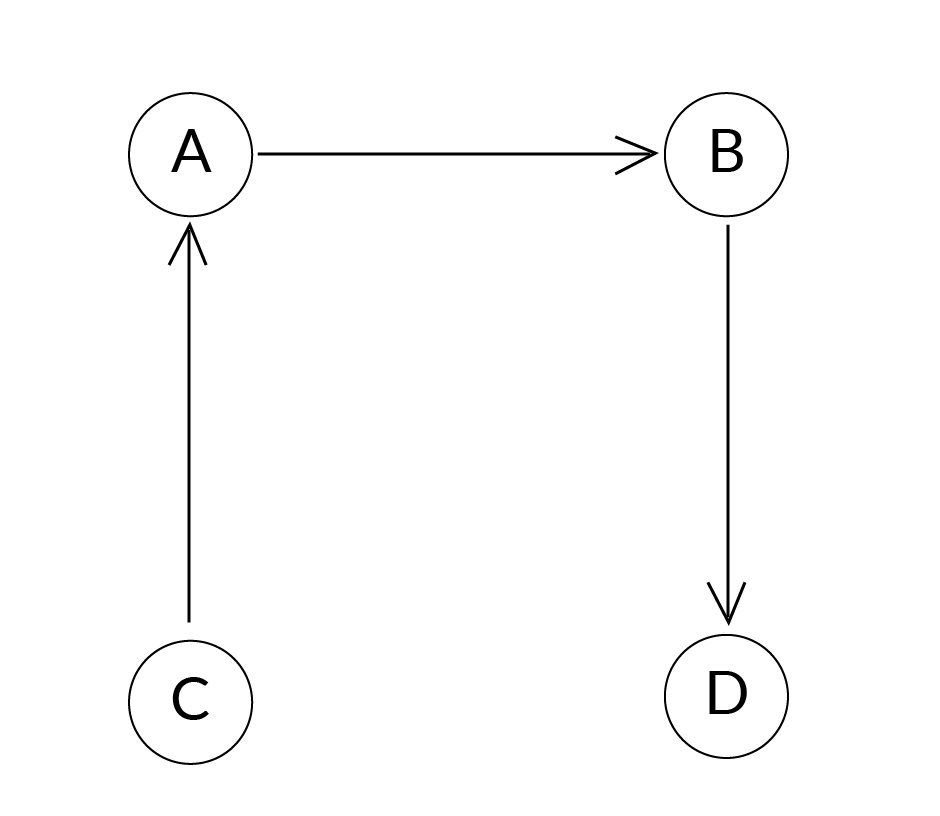
{(1, 2), (1, 3), (2, 4)}

**✓ Correct**

**Feedback:**

There are three edges in the given graph. There is an edge between 2 and 4, 3 and 1, and 1 and 2 each. Since the graph is undirected, the order of the vertices in the pair representing the edge does not matter. The graph does not follow any specific order in representing the set of edges.

Q7: Which of the statements below is the correct representation of the edge set of the given graph?



Ans: {(A, B), (B, D), (C, A)}

**✓ Correct**

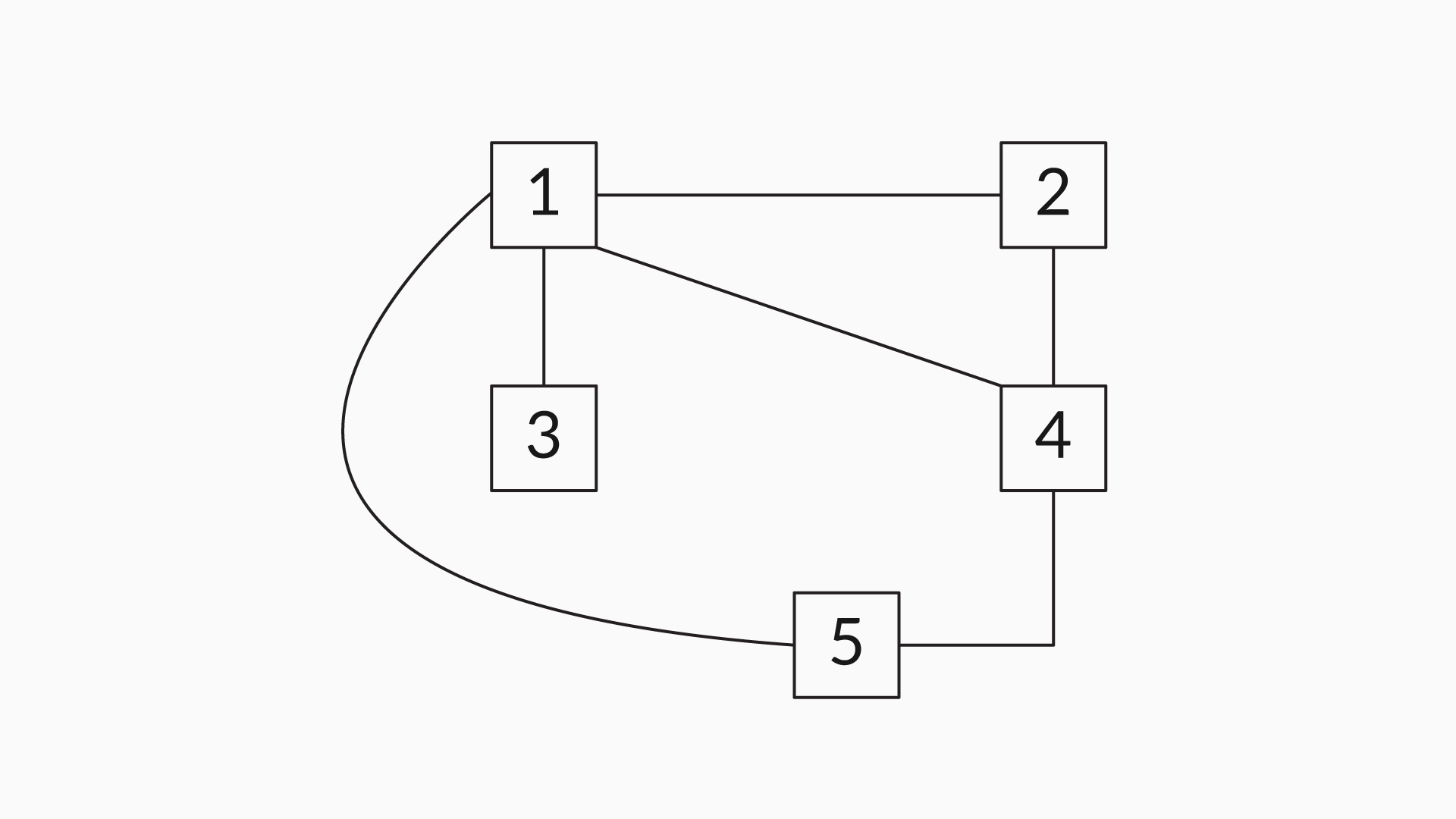
**Feedback:**

The given graph is directed. So, the pair representing the edge follows an order. There are three edges in the graph. There is an edge from A to B, B to D, and C to A each.

Q8: What is the maximum number of edges in a directed and undirected graph?

Ans: In a directed graph, each node can be connected with (n-1) other nodes. So, for n nodes, the maximum number of edges becomes n\*(n-1). However, in an undirected graph, the maximum number of edges is n\*(n-1)/2, because unlike in a directed graph, only one edge is required to represent a path from vertex ‘u’ to vertex ‘v’ and a path from vertex ‘v’ to vertex ‘u’. So, the number of edges becomes half in an undirected graph.

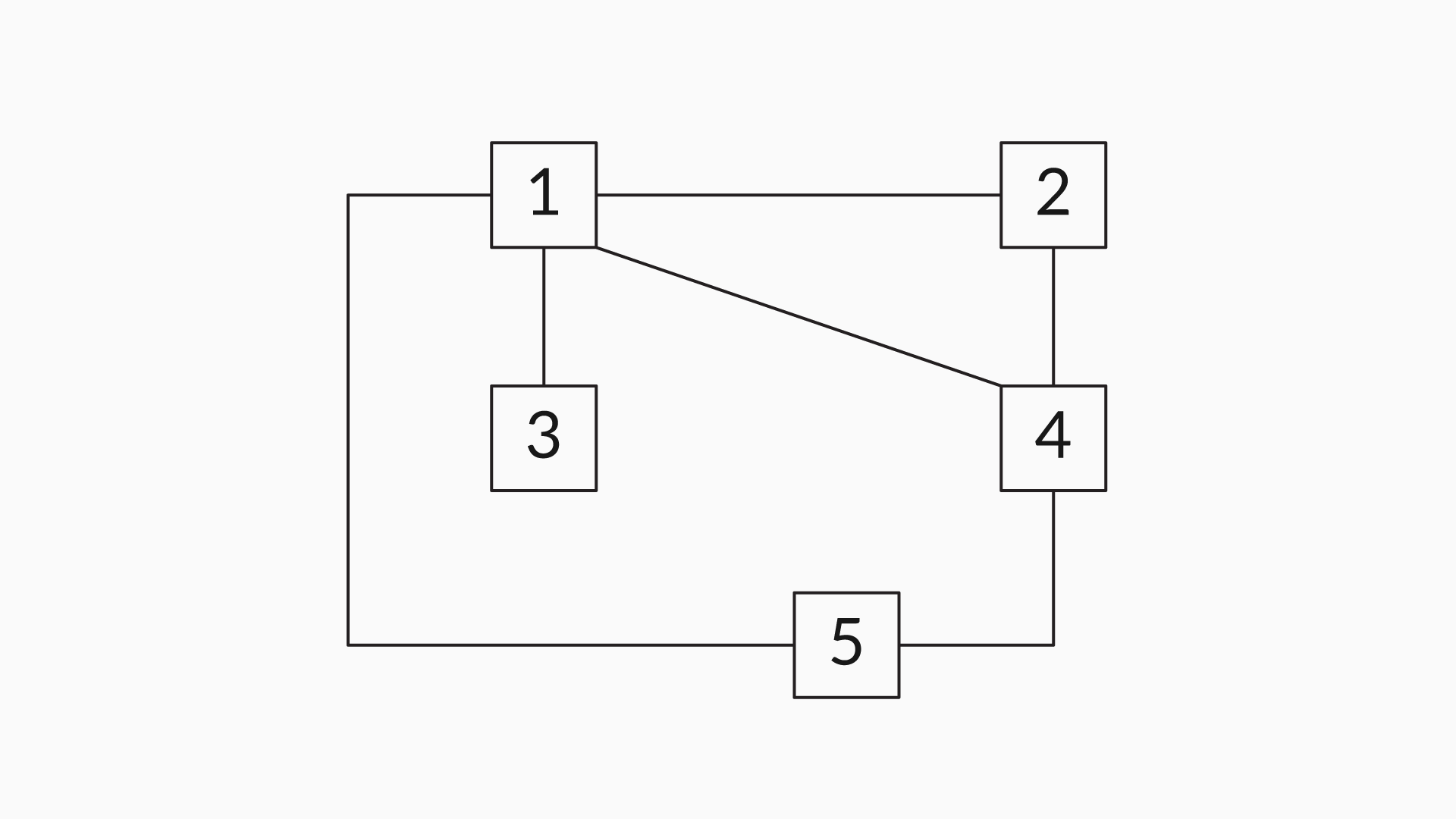
Q9: List out the neighbours of each node in the undirected graph given below.



Ans: The neighbours of all the nodes in the given graph are shown in the table given below.

|  |  |
| --- | --- |
| **Nodes** | **Neighbours** |
| 1 | {2, 3, 4, 5} |
| 2 | {1, 4} |
| 3 | {1} |
| 4 | {1, 2, 5} |
| 5 | {1, 4} |

Q10: List out all the different paths possible from node 2 to node 5 in the graph given below.

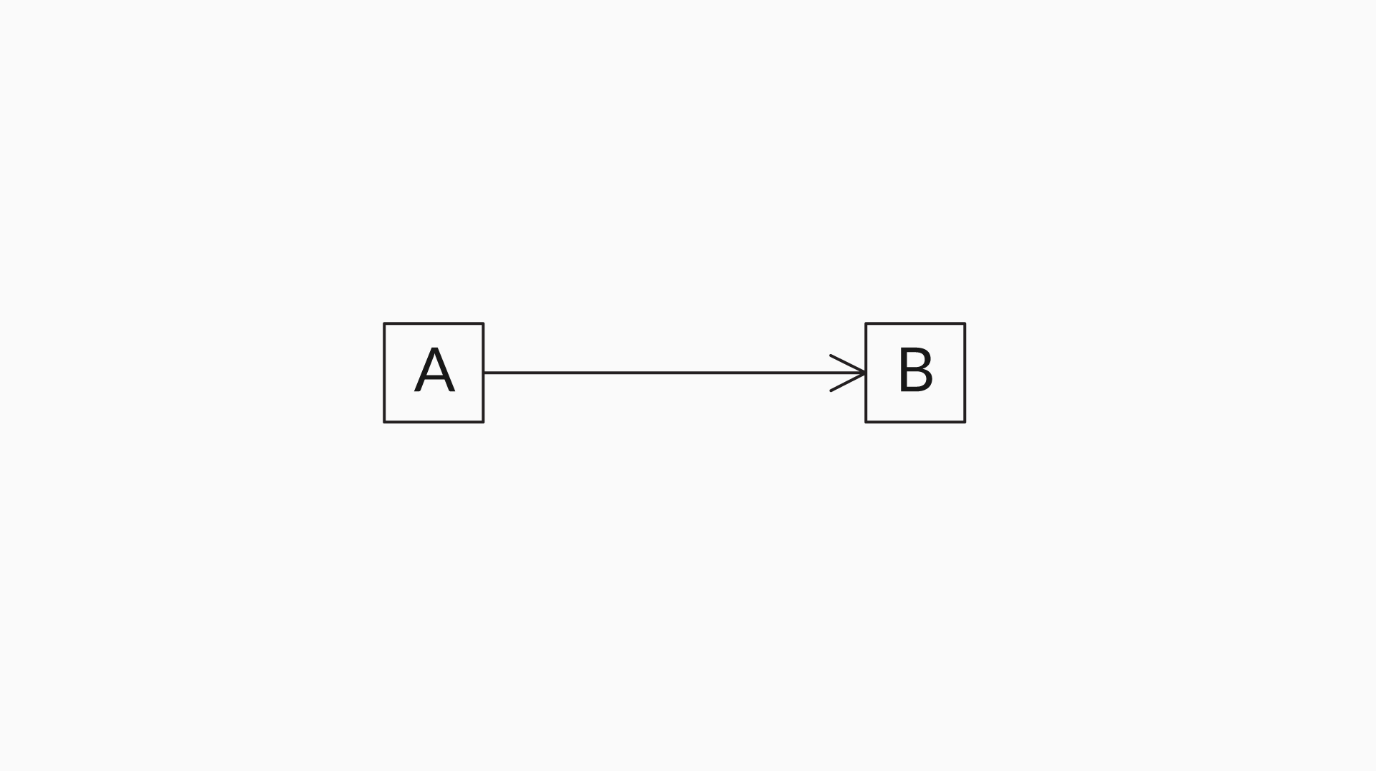


Ans: There are four different paths possible from node 2 to node 5 in the given graph. They are as follows:

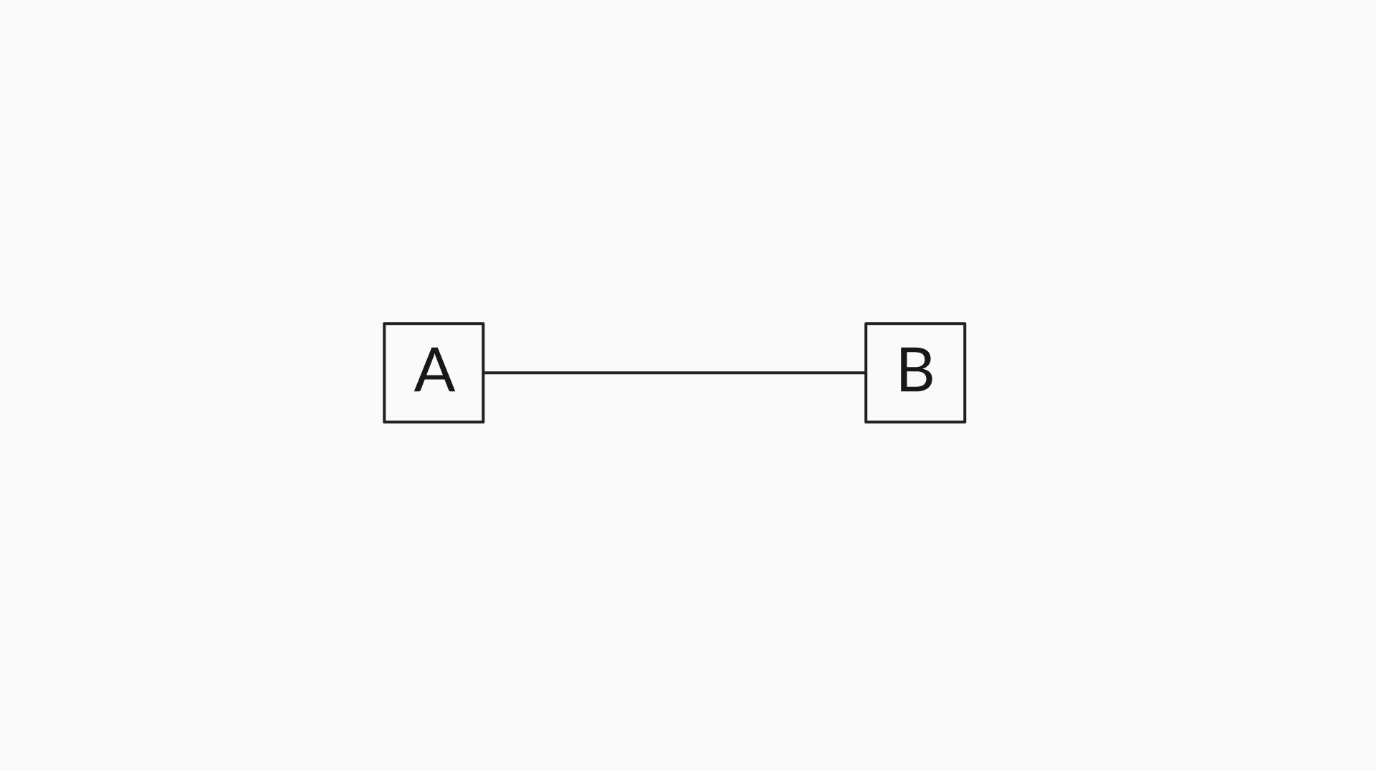
* 2 - > 4 - > 5
* 2 - > 1 - > 5
* 2 - > 1 - > 4 - > 5
* 2 - > 4 - > 1 - > 5

Q11: What is the most significant difference between an undirected and a directed graph when traversing along their edges?

Ans: As you have just learnt, a directed graph has asymmetric relationships between its nodes. If two nodes A and B are connected by a directed edge, then you will be able to traverse only from A to B but not from B to A.



However, for a similar undirected graph, it is possible to traverse in both directions, i.e., you can traverse from A to B and also from B to A, because of the symmetric relationships between the nodes.



**Neighbours:** If two nodes are adjacent to each other and connected by an edge, then those nodes are called neighbours.

**Degree:** The number of edges that are connected to a node is called the degree of the node.

Now, let us consider the undirected graph given below. We will now discuss how to determine the degree of each node in this graph.

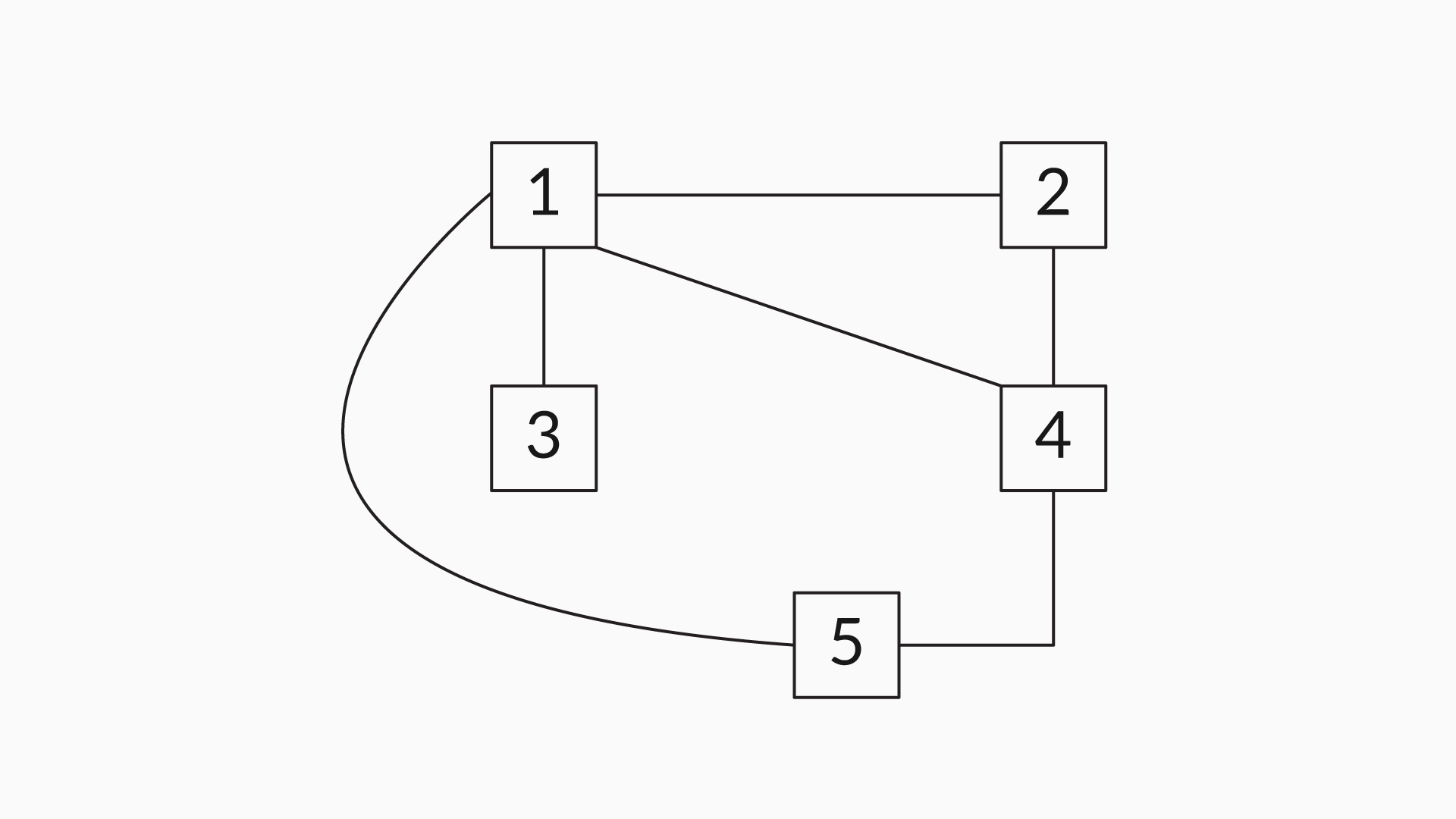


Figure 3

|  |  |  |
| --- | --- | --- |
| **Nodes** | **Neighbours** | **Degree** |
| Node 1 | {2, 3, 4, 5} | 4 |
| Node 2 | {1, 4} | 2 |
| Node 3 | {1} | 1 |
| Node 4 | {1, 2, 5} | 3 |
| Node 5 | {1, 4} | 2 |

In the case of directed graphs, degree can be classified as:

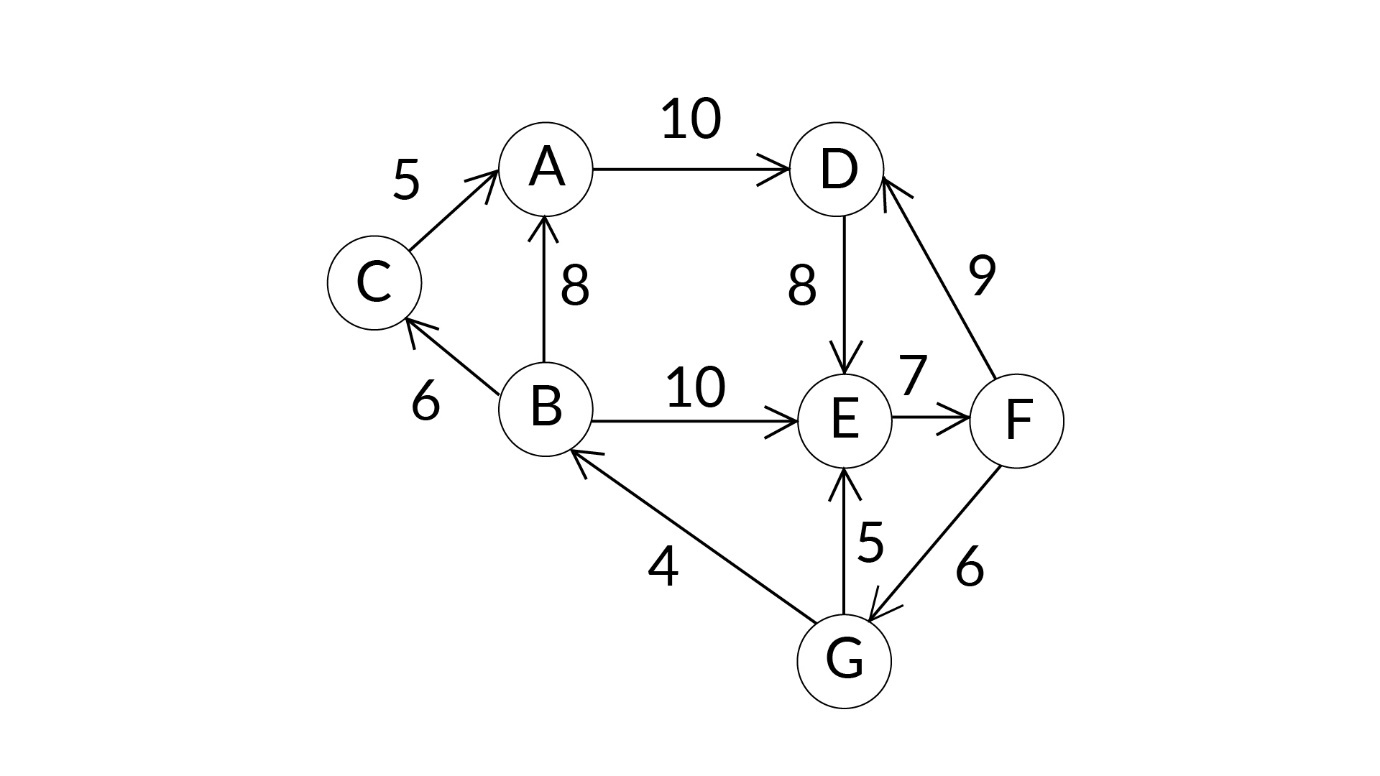
* **In-degree:** The number of incoming edges to a node
* **Out-degree:** The number of outgoing edges from a node

For a directed graph:  
Degree = In-degree (Edges pointing to the vertex) + Out-degree (Edges pointing away from the vertex).

**Path:** When a series of vertices are connected by a sequence of edges between two specific nodes in a graph, the sequence is called a path. For example, in the graph above, {2, 1, 4, 5} indicates the path between nodes 2 and 5, and the intermediate nodes are 1 and 4.

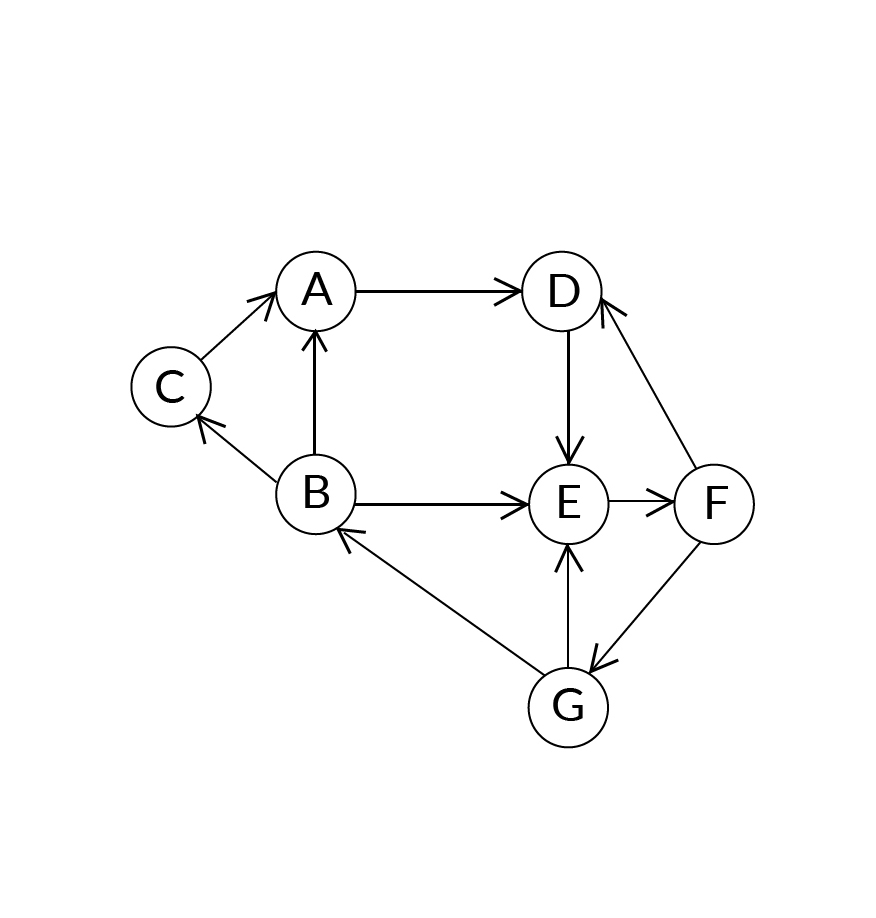
**Weighted graph:**A graph in which the edges contain some weights or values is called a weighted graph.

Example: If the nodes in a graph are considered to be cities and the edges are considered to be the paths between the cities, then the weights of these edges can be considered to be the distance between these cities.



Weighted graph

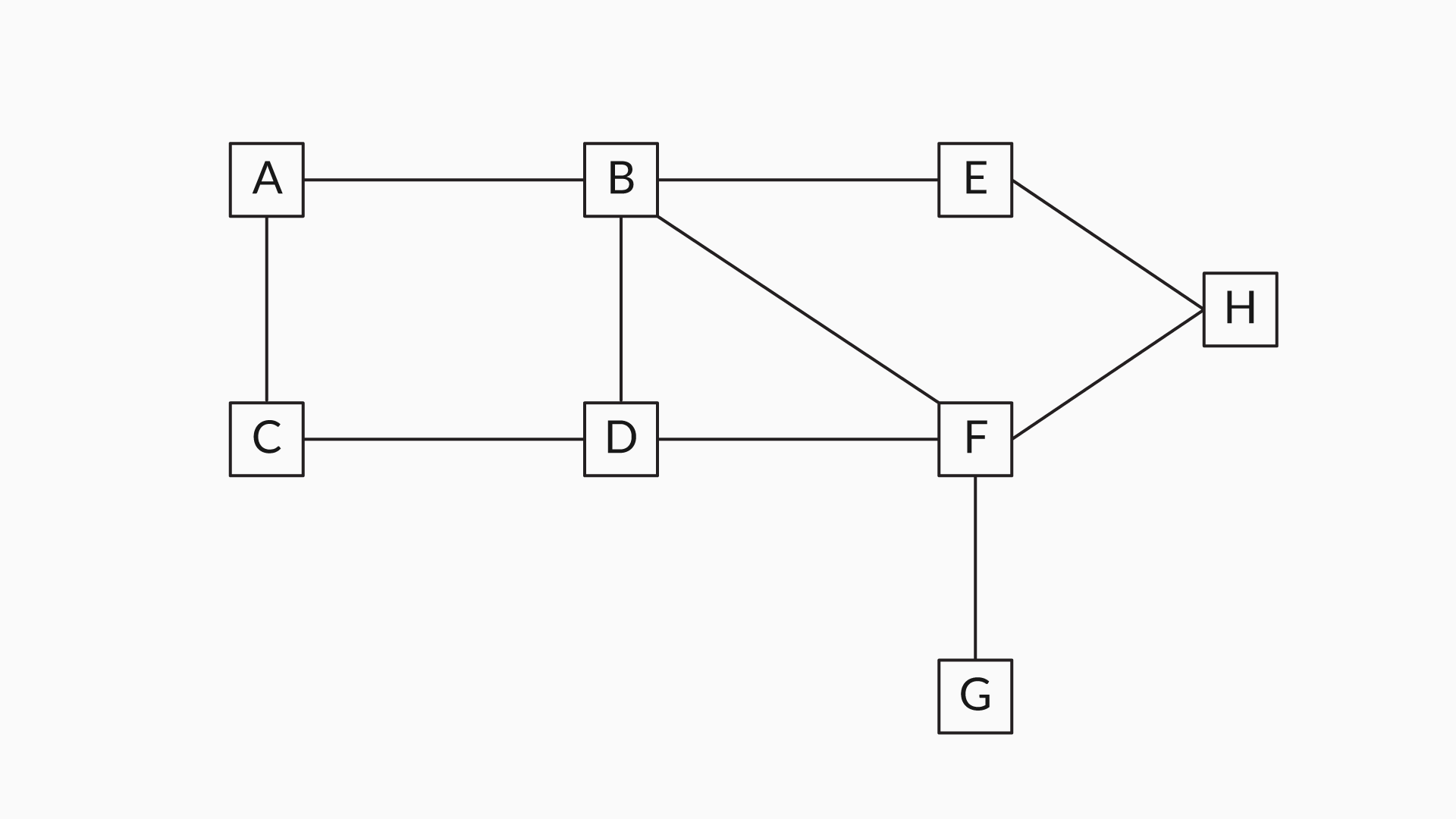
**Unweighted graph:** A graph in which edges contain no weight is called an unweighted graph.



Unweighted graph

So, now that you have a clear understanding of a graph’s properties, test your understanding by attempting the questions given below.

Q12: What is the degree of node F in the graph given below?



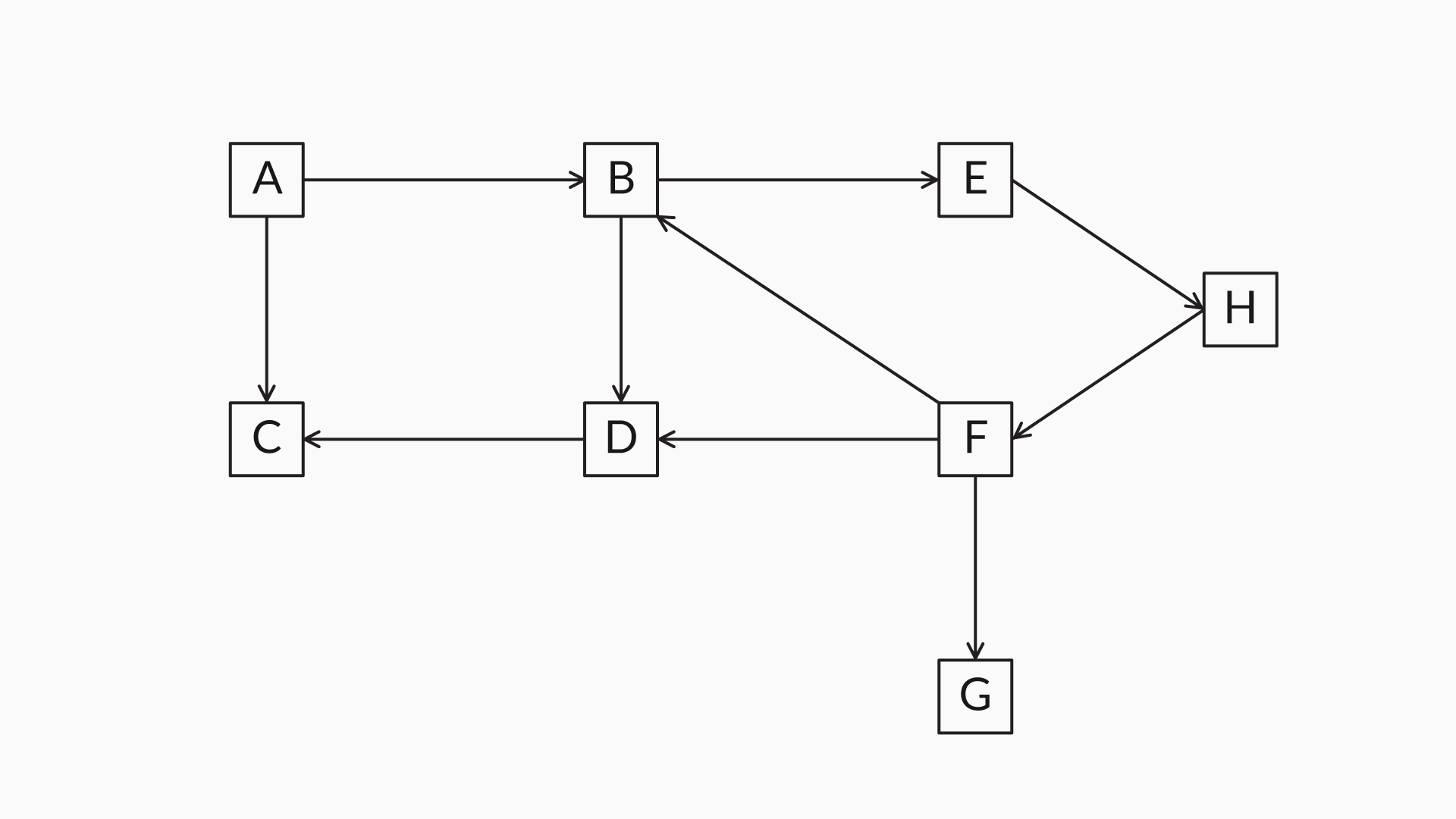
Ans:   
4

**✓ Correct**

**Feedback:**

In the given graph, node F is connected to nodes B, D, H and G. So, the degree of node F is 4.

Q13: Choose the correct path between node A and node G.



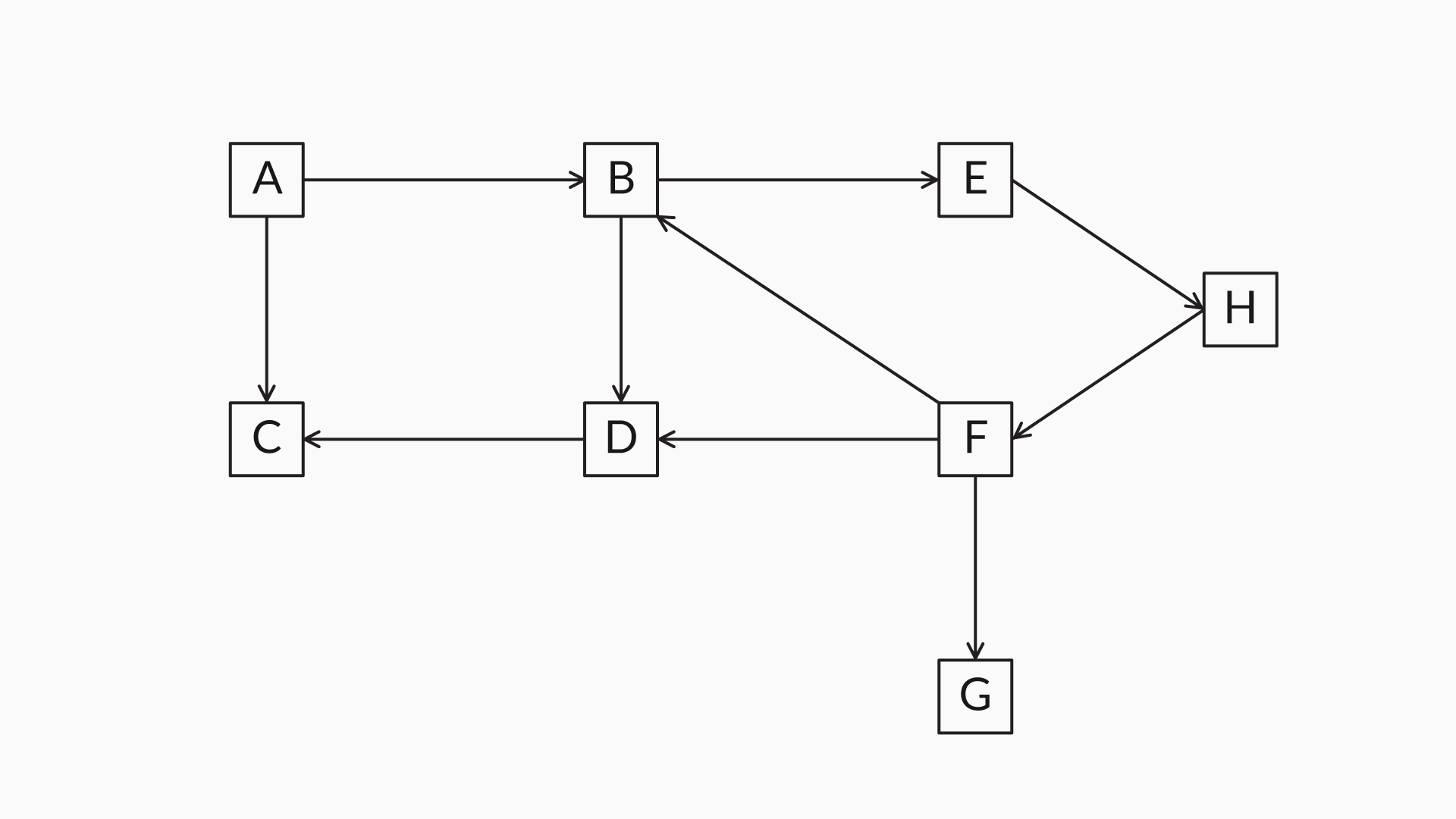
Ans: ABEHFG

**✓ Correct**

**Feedback:**

In the given directed graph, there is a path directed from node A to node B, and then from node B to node E, node E to node H, node H to node F, and, finally, a directed path from node F to node G. So, this is the correct path to traverse from node A to node G.

Q14: What is the degree of the node ‘B’?



Ans:   
4

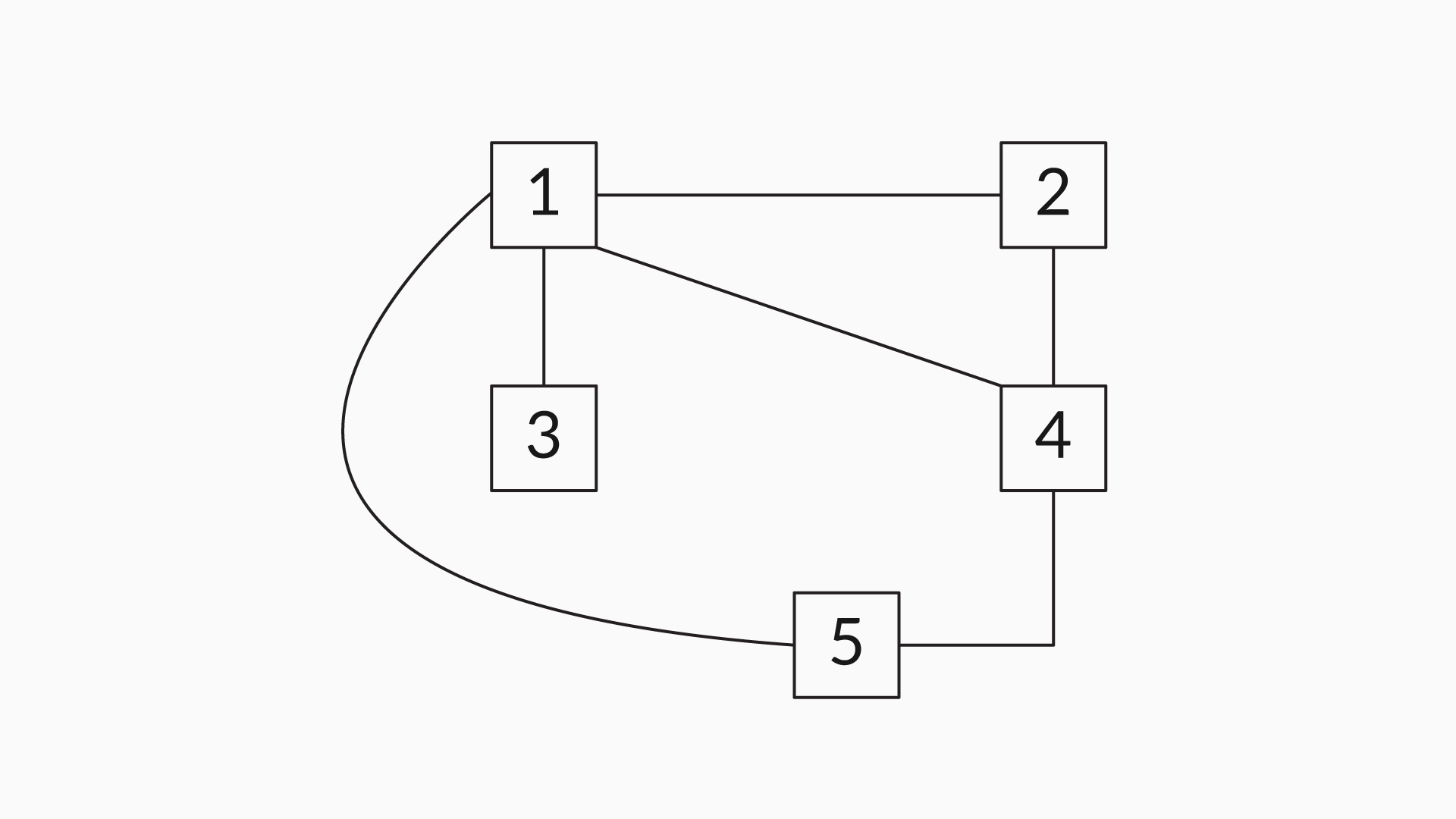
**✓ Correct**

**Feedback:**

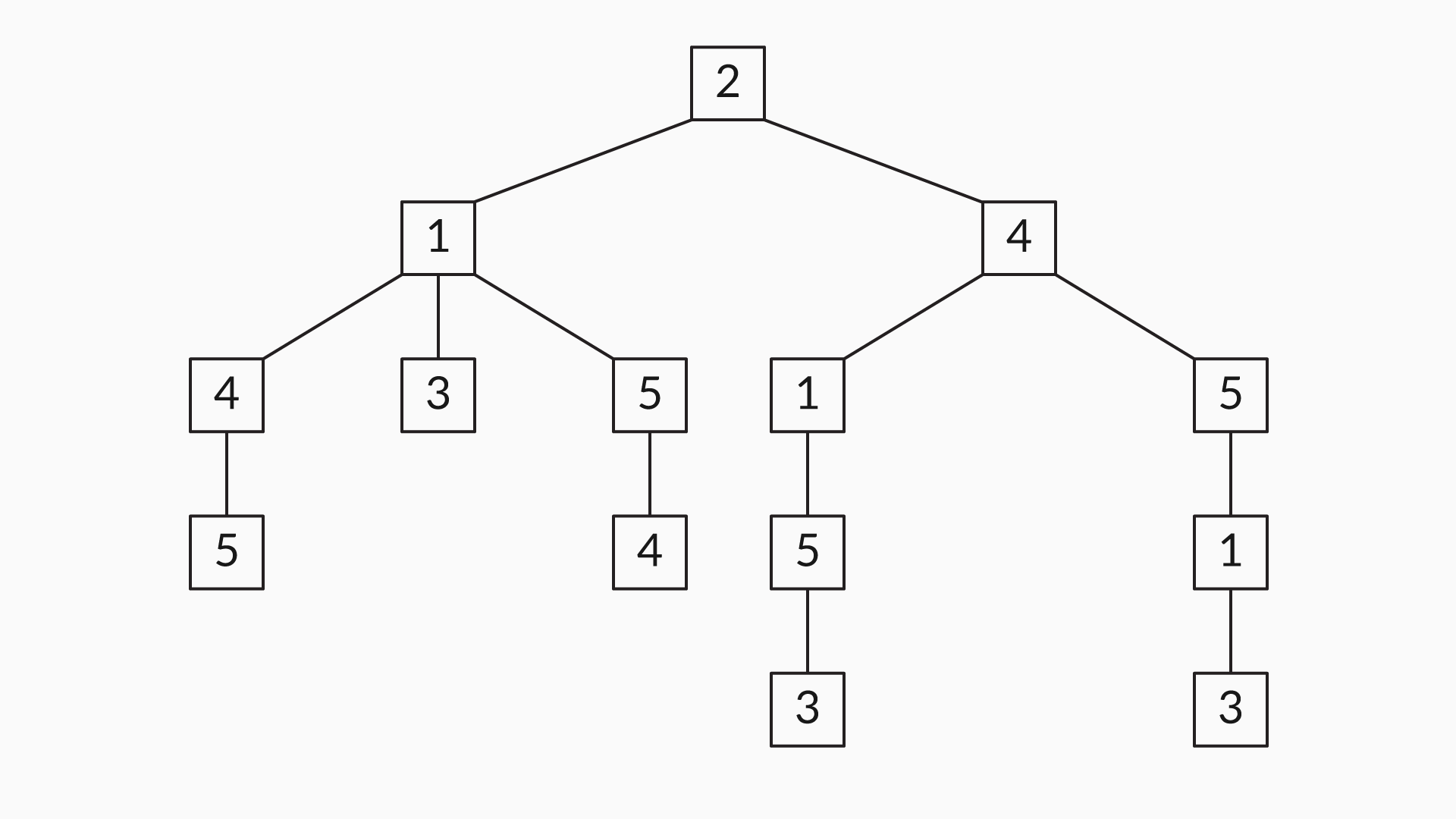
Degree in the directed graph is: In-degree + Out-degree. In the graph, In-degree of ‘B’ is 2 and Out-degree of ‘B’ is 2. Therefore, degree is 2+2 = 4.

# **Depth-First Search (DFS) – I**

Q15: List out all possible orders of visited nodes in a depth-first search for the graph below when the start node is 2.



Ans: To find all possible ways of traversing, you can depict all the nodes with a depth-first tree.



There are various possible ways to traverse starting from node 2. These include the following:

2 1 4 5 3

2 1 3 4 5

2 1 3 5 4

2 1 5 4 3

2 4 1 5 3

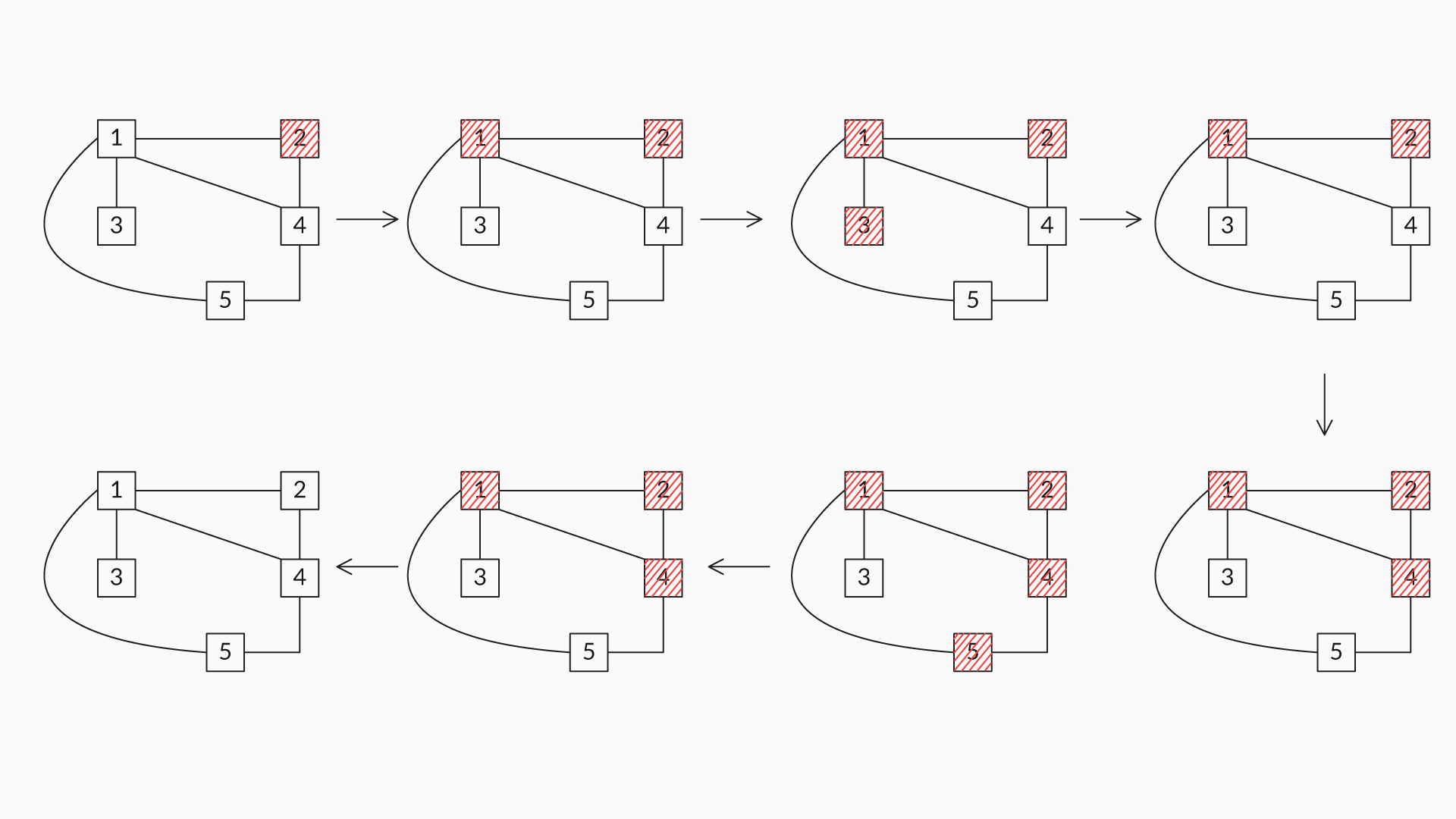
2 4 1 3 5

2 4 5 1 3

Depth-first search (DFS) is a traversal algorithm. From the start node, it traverses through any one of its neighbours and explores the farthest possible node in each branch before backtracking.

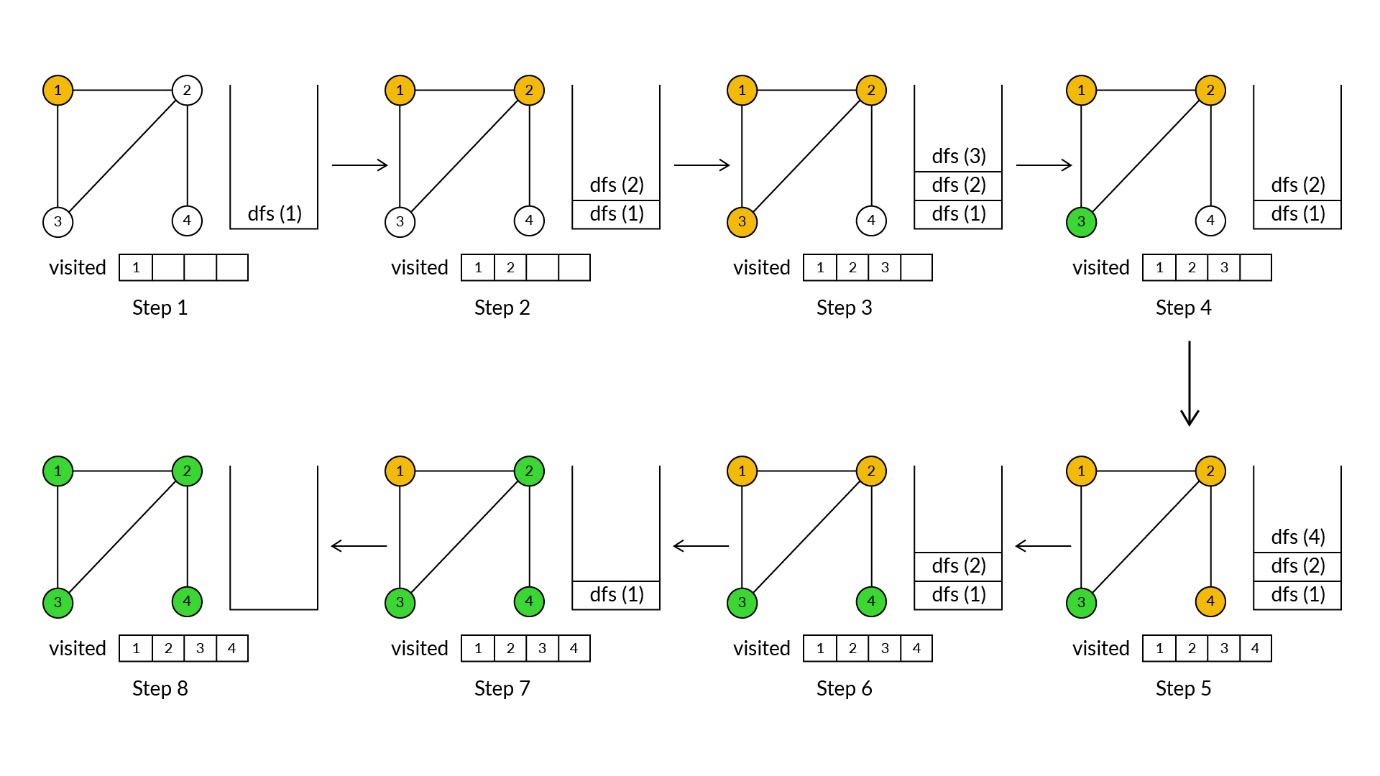
Backtracking happens when the search algorithm reaches a node where there are no neighbours to visit, or all the neighbours have been visited already. Then the DFS algorithm traces back to the previous node and traverses any neighbour that is left unvisited. In this way, backtracking helps with traversing through all the connected nodes in the graph and tracing back to the start node.

Here’s an image to explain the traversal of the DFS algorithm step-by-step on an example graph.



DFS

**Example**  
In the video, you saw the pseudocode for the depth-first search of a graph. Now, let’s take a look at the steps in finding the DFS traversal of the graph given in Step 1 in the image below, by taking node 1 as the starting node of the traversal.



Steps

Let us apply the pseudocode discussed in the video.

The steps in the image above are explained below:

**Step 1:**Run the dfs() method on node ‘1’ and add that node to the visited list

**Step 2:** The dfs(1) method recursively calls for all the unvisited neighbours of node ‘1’:

1. Here, the unvisited neighbours of node ‘1’ are {2, 3}.
2. Let us assume that dfs(1) recursively calls for node ‘2’ first and adds the node to the visited list.

**Step 3:**The dfs(2) method recursively calls for all the unvisited neighbours of node ‘2’:

1. Here, the unvisited neighbours of node ‘2’ are {3, 4}.
2. Let us assume that dfs(2) recursively calls for node ‘3’ first and adds the node to the visited list.

**Step 4:** The dfs(3) method recursively calls for all the unvisited neighbours of node ‘3’. Since there are no remaining unvisited neighbours of node ‘3’, it returns back.

**Step 5:** The dfs(2) method recursively calls for all the unvisited neighbours of node ‘2’:

1. Here, the unvisited neighbour of node ‘2’ is {4}. So, dfs(2) recursively calls for node ‘4’ and adds the node to the visited list.

**Step 6:** The dfs(4) method recursively calls for all the remaining unvisited neighbours of node ‘4’. Since there are no remaining unvisited neighbours of the node ‘4’, it returns back.

**Step 7:** The dfs(2) method recursively calls for all the remaining unvisited neighbours of node ‘2’. Since there are no remaining unvisited neighbours of the node ‘2’, it returns back.

**Step 8:** The dfs(1) method recursively calls for all the remaining unvisited neighbours of node ‘1’. Since there are no remaining unvisited neighbours of the node ‘1’, it returns back.

The visited list is the DFS of the graph.

Q16: For any given graph, explain what would happen when the following pseudocode or algorithm is used for depth-first traversal.

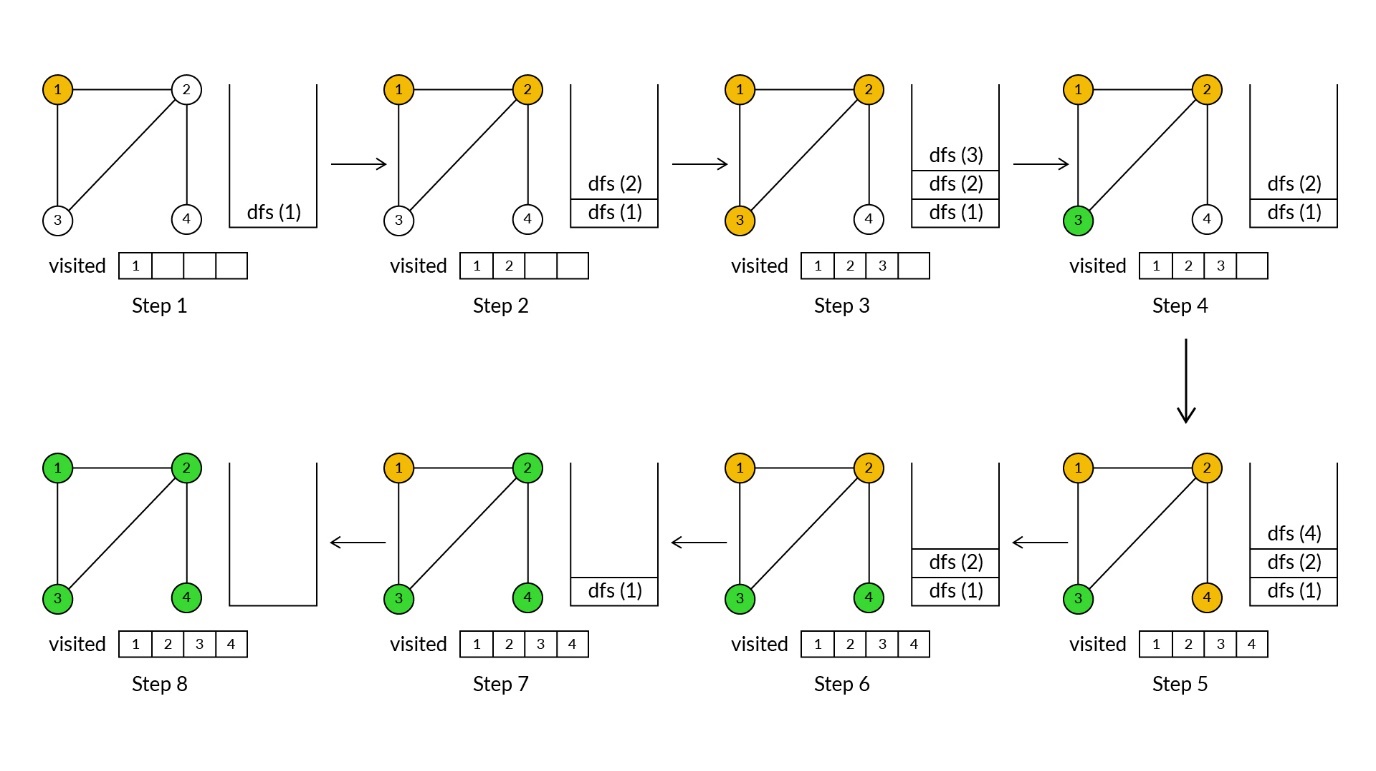
Procedure dfs(n)  
   for all n`∈ neighbours(n) do  
              dfs(n`)  
    end for  
end procedure

Ans: If the pseudocode given in the question is used for depth-first traversal, then for graphs containing cycles, it is possible that the traversal would go into an infinite recursive loop, because the visited nodes are not tracked. So, the pseudocode for depth-first traversal should be the following:

Visited ← { }  
Procedure dfs(n)  
 add n to visited set  
 for all n∈≠ighbours(n)do    if(n∈≠ighbours(n)do    if(n ∉ visited) then  
           dfs(n`)  
       end if  
 end for  
end procedure

The ‘if’ condition inside the ‘for’ loop checks whether a node has been visited previously or not. If the node is not visited, then it calls the depth-first search method recursively. In this way, the DFS traverses through all the connected nodes recursively and terminates at the end.

Q17: In the example above, you saw the DFS of the graph in Step 1 of the image given below. What are the steps in which backtracking happened?



Ans: 4

**✓ Correct**You missed this!

**Feedback:**

In Step 4, the DFS(3) method recursively calls for all the unvisited neighbours of node ‘3’. Since there are no remaining unvisited neighbours of node ‘3’, it returns back, i.e., backtracks.



6

**✓ Correct**

**Feedback:**

In Step 6, the DFS(4) method recursively calls for all the remaining unvisited neighbours of node ‘4’. Since there are no remaining unvisited neighbours of node ‘4’, it returns back, i.e., backtracks.



7

**✓ Correct**

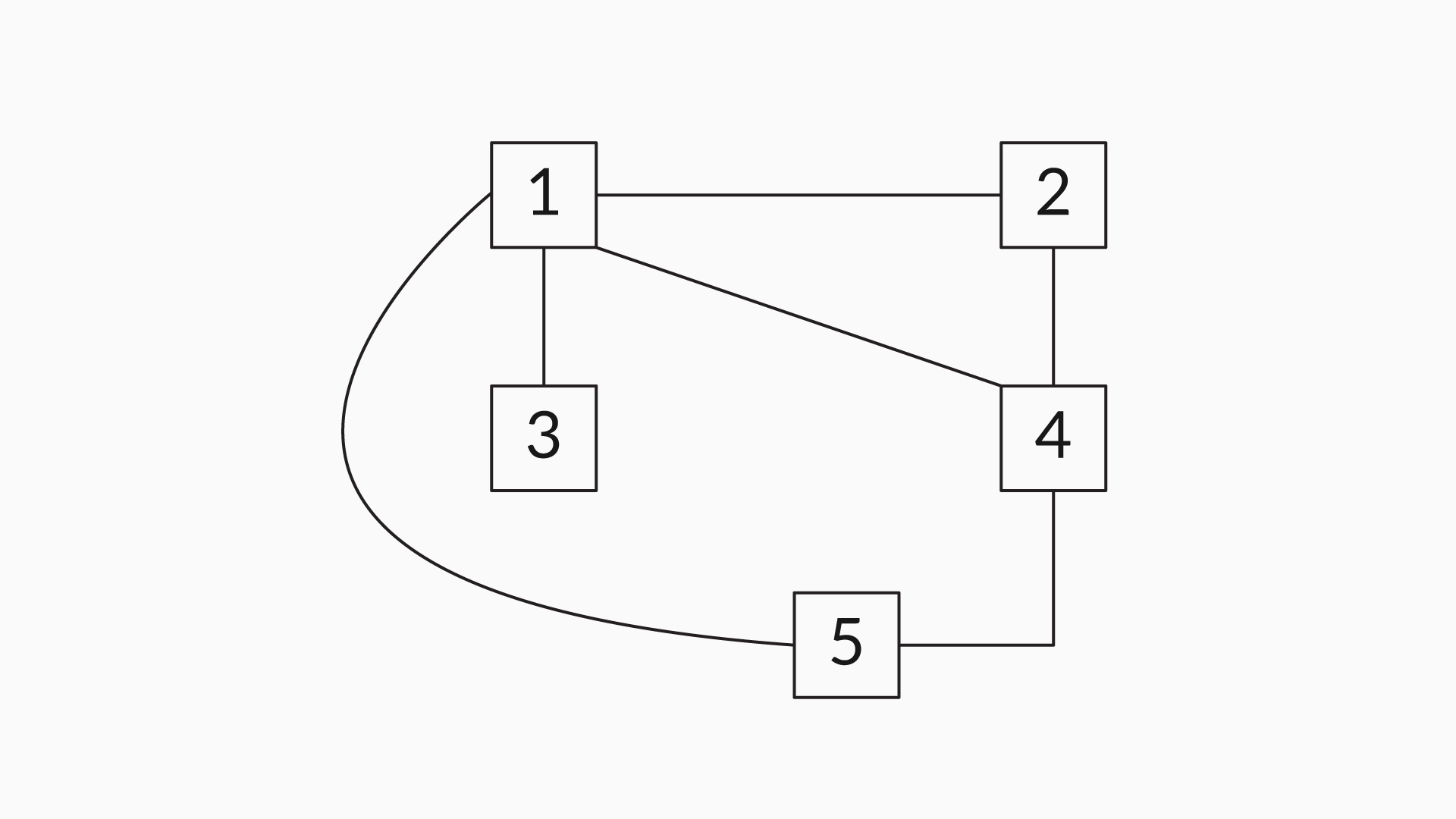
**Feedback:**

In Step 7, the DFS(2) method recursively calls for all the remaining unvisited neighbours of node ‘2’. Since there are no remaining unvisited neighbours of node ‘2’, it returns back, i.e., b

Q18: Explain how backtracking has been implemented in the pseudocode for DFS.

Ans: Backtracking is implemented using a stack, which keeps track of the nodes that are yet to be visited. As the search algorithm reaches a certain node (n) where there are no neighbours to be visited, or where all the neighbours have been visited already, the function call ‘dfs(n)’ pops out of the stack and traces back to the previous node in the traversal. This process of backtracking and the popping out of function calls from the stack continues until there are no more unvisited neighbours left to traverse or the search algorithm reaches the start node. This is how a recursion stack helps us perform backtracking in a DFS algorithm.

Q18 : Choose the correct possible order of the visited nodes in the depth-first traversal of the graph below when the start node is 4.



Ans: 4 1  2  3  5

**✓ Correct**

**Feedback:**

* Neighbours of the start node 4 are {1, 2, 5}, and the next node upon random selection can be node 1.
* Neighbours of node 1 are {2, 3, 5, 4}, and node 2 can be chosen from the unvisited neighbours.
* Neighbours of node 2 are {1, 4}, and both of them have been visited already. So, the algorithm recursively returns to node 1.
* Unvisited neighbours of node 1 are {3, 5}. So, the next possible node to be visited can be 3, and as you can observe, there are no neighbours of node 3. Therefore, the algorithm recursively returns to node 1.
* Now, the only node left is 5, and so, it would be our next node to visit.

This is one of the possible depth-first traversals where the start node is 4.

Q19: What is the most significant difference between the DFS algorithm of a graph and a tree?

Ans: The most significant difference is that the visited set is used by the graph DFS to keep track of all the visited nodes. This is done to prevent the traversal from falling into infinite recursive loops when encountering loops or cycles in a graph structure. On the other hand, trees have a hierarchical model and need not keep track of visited nodes in the DFS algorithm.

The pseudocode of the DFS algorithm is given below.

Visited ← { }

Procedure dfs(n)

  add n to visited set

  for all n` ∈ neighbours(n) do

        if (n` ∉ visited) then

            dfs(n`)

        end if

  end for

end procedure



**Step 1:** The start node of the DFS algorithm is added to the visited set.

**Step 2:** The ‘for’ loop instruction set is executed for all the neighbours of the start node.

**Step 3:** Then if the neighbour node is unvisited, only then it calls the dfs() method recursively.

**Step 4:** Now, the unvisited neighbour node becomes the start node and repeats the steps above.

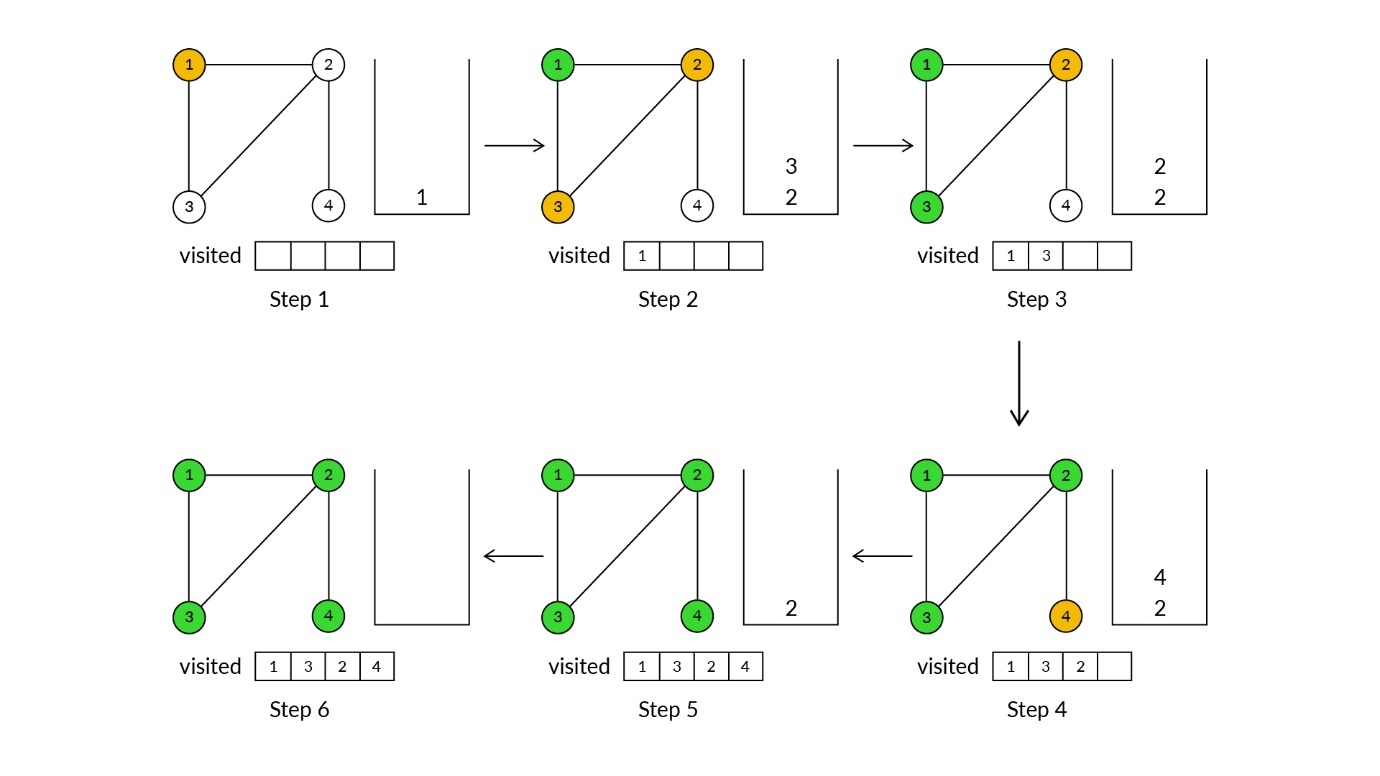
**Step 5:** The DFS traversal reaches a node where there are no unvisited neighbour nodes. Here, the recursive algorithm backtracks to the earlier unvisited neighbour nodes that are stored in the stack and visits the remaining nodes.

Thus, the DFS recursively visits all the nodes along one branch and then backtracks to the unvisited neighbour nodes. Once all the nodes connected from the start node are visited, the algorithm ends.

# **Depth-First Search (DFS) – II**

Now, since you have already seen the DFS of a graph using recursion, and we know that there is a relation between recursion and stack, let’s see the DFS of a graph using stacks with an example.

Let’s take a look at the steps in finding the DFS traversal of the graph given in Step 1 in the image below, by taking node 1 as the starting node of the traversal.



Steps

The steps in the image above are explained below:

**Step 1:**Push the starting node, which is node 1 here, to stack.

**Step 2:**Pop the stack; the popped element here is ‘1’:

1. If the popped element is not on the visited list, then add it to the visited list:
   1. So, ‘1’ is added to the visited list.
2. Now, push all the neighbours of the popped element that are not on the visited list:
   1. Therefore, 2 and 3 are pushed to the stack.

**Step 3:** Pop the stack; the popped element here is ‘3’:

1. If the popped element is not on the visited list, then add it to the visited list:
   1. So, ‘3’ is added to the visited list.
2. Now, push all the neighbours of the popped element that are not on the visited list:
   1. Therefore, 2 is pushed to the stack.

**Step 4:**Pop the stack; the popped element here is ‘2’:

1. If the popped element is not on the visited list, then add it to the visited list:
   1. So, ‘2’ is added to the visited list.
2. Now, push all the neighbours of the popped element that are not on the visited list:
   1. Therefore, 4 is pushed to the stack.

**Step 5:** Pop the stack; the popped element here is ‘4’:

1. If the popped element is not on the visited list, then add it to the visited list:
   1. So, ‘4’ is added to the visited list.
2. Now, push all the neighbours of the popped element that are not on the visited list:
   1. Since there are no neighbours of the popped element that are not on the visited list, do nothing.

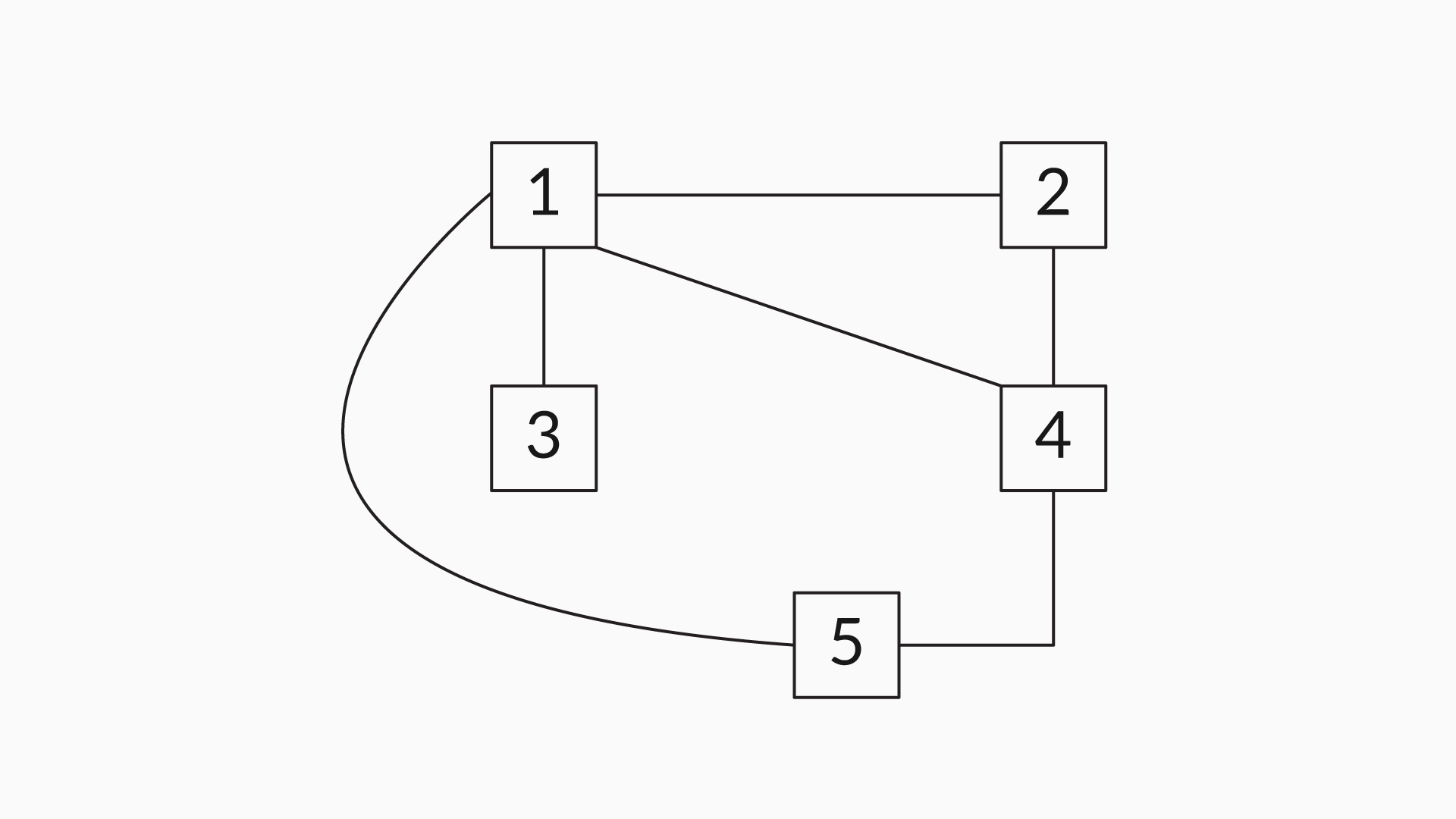
**Step 6:**Pop the stack; the popped element here is ‘2’:

1. If the popped element is not on the visited list, then add it to the visited list:
   1. Since ‘2’ is already on the visited list, do nothing.
2. Now, push all the neighbours of the popped element that are not on the visited list:
   1. Since there are no neighbours of the popped element that are not on the visited list, do nothing.

Since the stack is empty, the visited list is the DFS of the graph in the Step 1.

Q20: Explain how recursion helps perform a DFS traversal in a given graph.

Ans: In a depth-first search, recursion plays a critical role as it traverses to the last node of each branch in a given graph structure. It also helps with backtracking to previous unvisited nodes.

For the graph below, let us consider a scenario in which the depth-first traversal has reached the last visited node, node 5, from the start node 2. Then, the successive recursion calls are stored in the stack in the following manner when the start node is 2.  


|  |
| --- |
| →dfs(5) |
| dfs(4) |
| dfs(1) |
| dfs(2) |

After traversing to node 5, the graph will not have any unvisited neighbour nodes. So, it completes the function call of dfs(5) and returns to the function call of dfs(4). Thus, the recursion stack that maintains the record of all the successive function calls helps with backtracking and visiting all the connected nodes of a given graph during depth-first traversal.

Q21: Think and develop the iterative pseudocode for the depth-first search of a graph.

Recursive pseudocode

Visited <- {}  
Procedure dfs(n)  
   add n to visited set  
   for all n ∈ neighbours(n) do  
         if (n ∉ visited) then  
             dfs(n)  
         end if  
   end for  
end procedure

Ans: The iterative pseudocode of the depth-first search algorithm is as follows:

Procedure dfs(n){  
   S=empty stack  
   push n onto S  
   while(S is not empty){  
       n=pop(S)  
       if(n is unmarked)  
           add n to visited set  
       end if

       while (n` ∈ neighbours of n and n` is unmarked)

              push n` onto S

       end while  
   end while  
end procedure

Q22: Can you explain how a stack plays a critical role in the iterative pseudocode of the DFS algorithm?

Ans: A DFS traverses from the start node and so, the start node is pushed into the stack. Here, the ‘while(S is not empty)’ condition is satisfied and it enters the while loop.

The inner while loop looks like this:

**while** (n` ∈ neighbours of top(S) and n` is unmarked) {

mark(n`)

push n` onto S

}



In the first iteration, one of the neighbours of the start node is randomly marked and pushed into the stack:

* The neighbour node, which is marked in the first iteration, comes to the top of the stack for a second iteration of the while loop.
* Then the unvisited neighbour of the corresponding node is marked and pushed into the stack.

In this way, the LIFO property of a stack helps with traversing along a particular path as deeply as possible and then backtracking to the unvisited neighbour nodes of the given graph when there is no other node to visit along that path.

Q23: A recursive DFS uses \_\_\_\_\_\_, while an iterative DFS uses \_\_\_\_\_\_ .

Ans: a function call stack, a user-defined stack

**✓ Correct**

**Feedback:**

The only basic difference between a recursive DFS and an iterative DFS is that a recursive DFS uses a function call stack (used during recursion) and an iterative DFS uses a user-defined stack.

For the Java code implementation of the DFS algorithm, please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Additional Resources - Graphs and Graph Algorithms -> Depth-First Search (DFS) - II.

To know how the DFS algorithm can be used to understand whether all the nodes in a graph are connected or not, please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Additional Resources - Graphs and Graph Algorithms -> Depth-First Search (DFS) - III.

# **Breadth-First Search (BFS) – I**

Q24: What is the data structure that can be used to implement the BFS algorithm? Give reasons.

Ans: The FIFO property of the queue data structure helps with implementing the BFS algorithm. In a breadth-first search, while traversing from the start node, the start node is enqueued and marked as visited:

* Now the start node is dequeued and its immediate neighbours are enqueued and marked as visited.
* Since a queue provides FIFO property, each of the start node’s immediate neighbour is dequeued before enqueuing their corresponding next-level neighbour nodes and mark them as visited.

Breadth-first search is a traversing algorithm, where traversing begins from the start node and then explores the immediate neighbours of the start node. Then the traversing moves towards the next-level neighbours of the graph structure. As the name suggests, traversal across the graph happens breadthwise.

To implement a BFS, you need to consider the stage of each node. Nodes, in general, are considered to be in the following three different stages:

* Not visited,
* Visited and
* Completed.

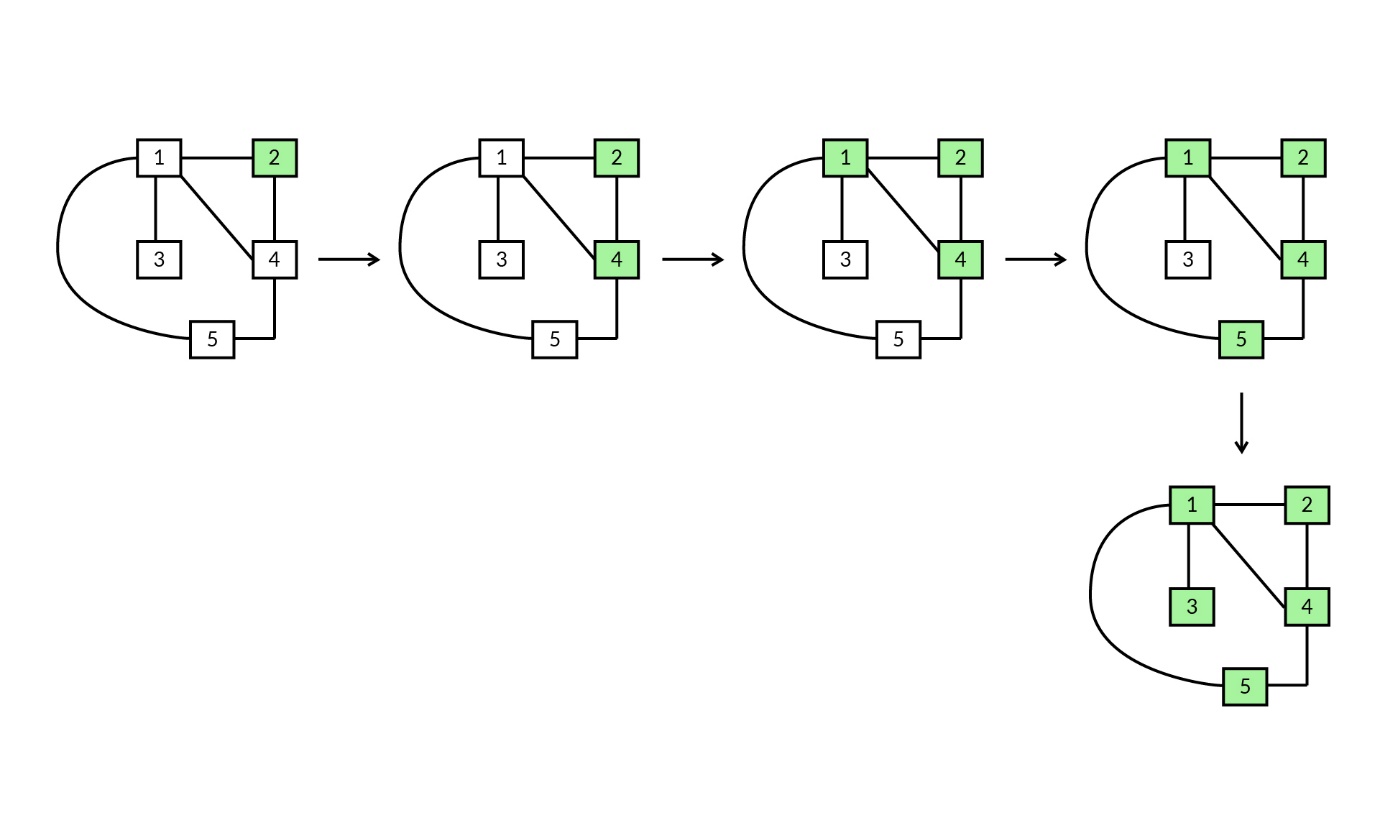
In the BFS algorithm, nodes are marked as visited during traversal, in order to avoid the infinite loops caused due to the possibilities of cycles in a graph structure.

# Breadth-First Search (BFS) – II

Q25: Explain the pseudocode of the breadth-first search in your own words.

1. Ans: Start from a vertex S. Let this vertex be at what is called ‘Level 0’
2. Find all the other vertices that are immediately accessible from this starting vertex S, i.e., they are only a single edge away
3. Mark these vertices to be at ‘Level 1’
4. Mark which is the parent vertex of the current vertex where you are, i.e., the vertex from which you accessed the current vertex. Do this for all the vertices at Level 1.
5. Now, find all those vertices that are a single edge away from all the vertices that are at ‘Level 1’. These new set of vertices will be at ‘Level 2’.
6. Repeat this process until all the nodes have been traversed.

Here is an image to explain the traversal of the BFS algorithm step by step on an example graph.

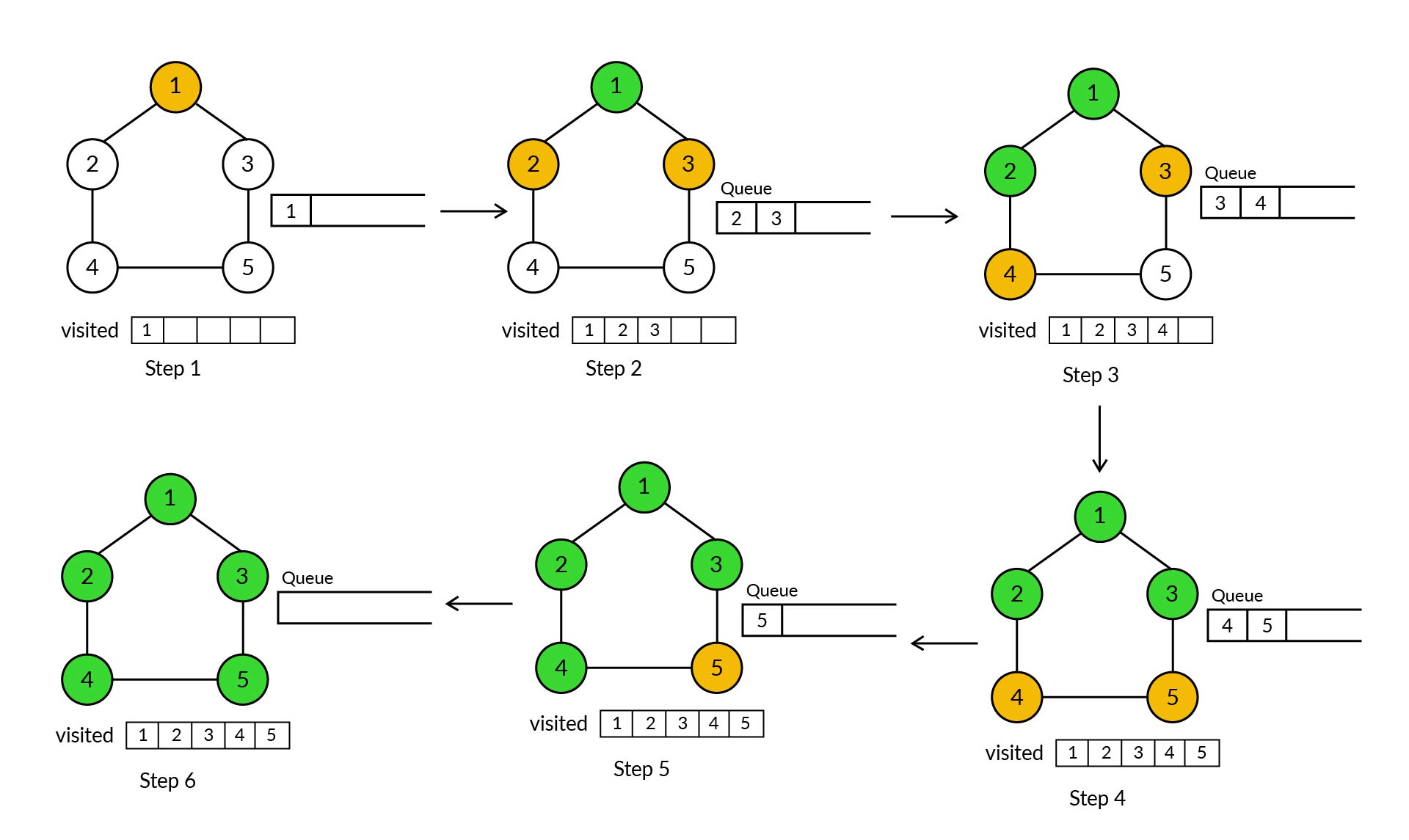


Example

#### Q26: Breadth-First Search (BFS)

You have seen pseudocode for a breadth-first search of a graph. Now, try to write the steps in finding the BFS traversal of the graph given in Step 1 in the image below, by taking node 1 as the starting node of the traversal.

**Hint:**



Ans: Let us apply the pseudocode discussed in the video.

The steps in the image above are explained below:

**Step 1:** Enqueue node ‘1’ to the queue and add it to the visited list

**Step 2:** Dequeue an element from the queue; here it is ‘1’:

* Enqueue all the neighbours of the popped element that are not on the visited list to the queue and also add them to the visited list:
  + Enqueue 2 and 3 to the queue and add them to the visited list.

**Step 3:**Dequeue an element from the queue; here it is ‘2’:

* Enqueue all the neighbours of the popped element that are not on the visited list to the queue and also add them to the visited list:
  + Enqueue 4 to the queue and add it to the visited list

**Step 4:** Dequeue an element from the queue; here it is ‘3’:

* Enqueue all the neighbours of the popped element that are not on the visited list to the queue and also add them to the visited list:
  + Enqueue 5 to the queue and add it to the visited list

**Step 5:** Dequeue an element from the queue; here it is ‘4’:

* Enqueue all the neighbours of the popped element that are not on the visited list to the queue and also add them to the visited list:
  + Since there are no neighbours of ‘5’ that are not on the visited list, do nothing.

**Step 6:** Dequeue an element from the queue; here it is ‘5’:

* Enqueue all the neighbours of the popped element that are not on the visited list to the queue and also add them to the visited list:
  + Since there are no neighbours of ‘5’ that are not on the visited list, do nothing.

Since the queue is empty, stop. The visited list is the BFS of the graph.

The pseudocode of the BFS algorithm is given below.

Procedure bfs(n)

 Q ← new Queue

 Visited ← { }

 enqueue (Q, n)

 Add n to visited set

 While Q is not empty

    n ← dequeue(Q)

    for all n` ∈ neighbours(n)

          if (n` ∉ visited) then

                enqueue(Q, n`)

               add n` to visited set

         end if

    end for

 end while

end procedure





**Step 1**: The start node is enqueued and also marked as visited in the following set of instructions:

 enqueue (Q, n)

 Add n to visited set

**Step 2**: The ‘while’ loop instruction set is executed when the queue is not empty.

**Step 3**: For each iteration of the ‘while’ loop, a node gets dequeued.

**Step 4**: Now the ‘for’ loop runs till all the unvisited neighbours of the dequeued node (n) are enqueued and also marked as visited.

**Step 5**: For the first iteration of the ‘while’ loop, all the neighbour nodes of the start node are enqueued. And for the second iteration, all the next-level unvisited neighbour nodes of one of the neighbour nodes are enqueued.

In this way, all the neighbour nodes are enqueued and visited level-wise from the start node. And after a certain number of iterations, all the nodes are dequeued and the algorithm ends.

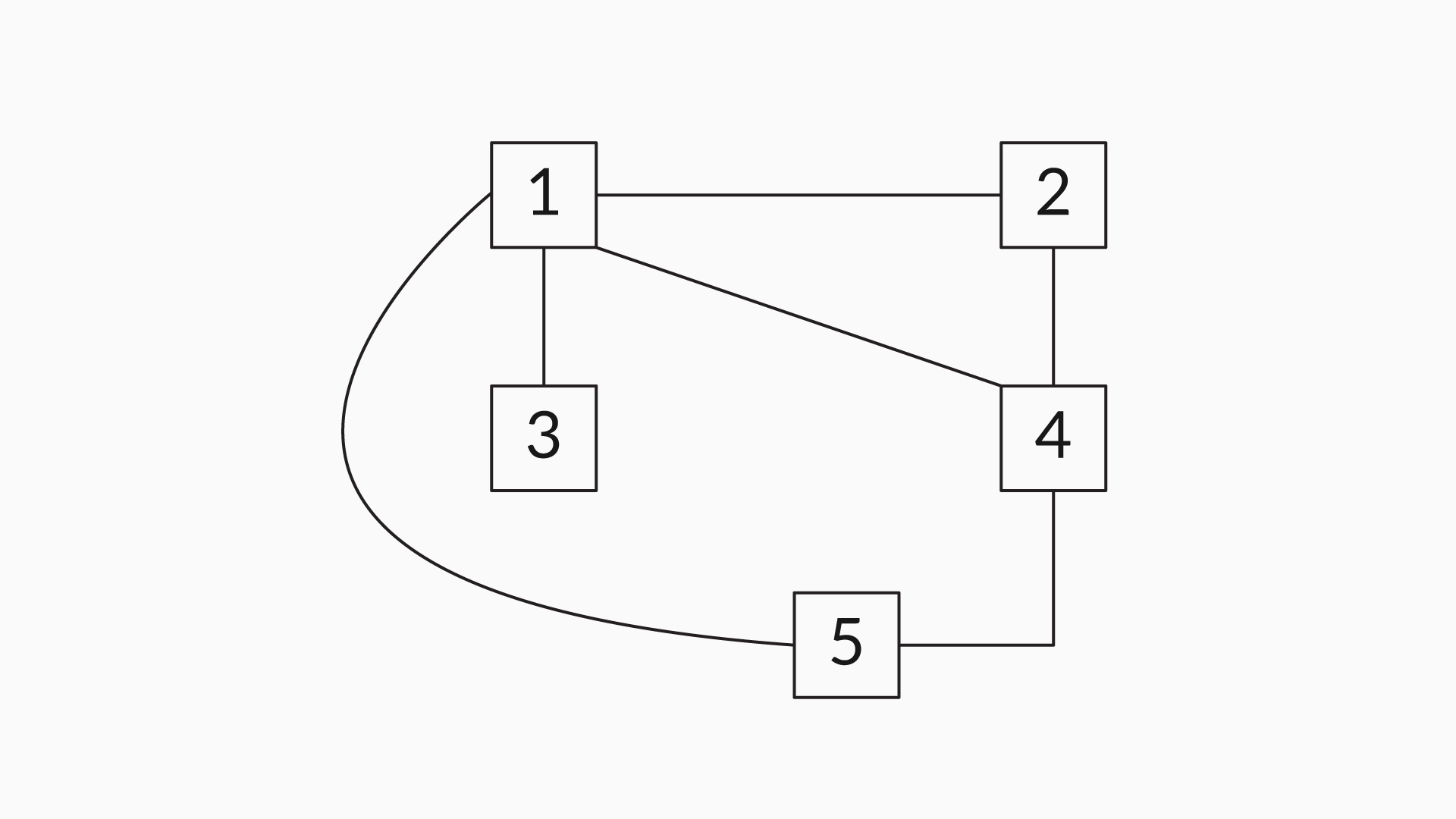
Q26: Explain how a breadth-first search is different from the depth-first search algorithm.

Ans: Depth-first search starts from a particular node and traverses through any one of its neighbours. Then it explores as far as possible along each branch before backtracking.

On the contrary, breadth-first search visits all the immediate neighbours at first and then explores the unvisited neighbour nodes of the immediate neighbours of the start node, and so on.

To implement the DFS algorithm, you can use the stack data structure, because its LIFO property allows the algorithm to explore the neighbour of its neighbours immediately. On the other hand, to implement the BFS algorithm, the queue data structure is used, because its FIFO property compels the algorithm to explore all the immediate neighbours of a node, before the algorithm is allowed to explore the  neighbours of its neighbours.

Q27: Choose the correct possible order of visited nodes in the breadth-first traversal of the graph below when the start node is 4.



Ans:   
4 5  2  1  3

**✓ Correct**

**Feedback:**

Neighbours of the start node 4 are {1, 2, 5}, and, in sequence, all of these neighbours are visited at first in a random order, which can possibly be 5, 2 and 1, and then the unvisited neighbours of {1, 2, 5} are visited, which is node 3 only.

So, {4, 5, 2, 1, and 3} is one of the possible orders of the breadth-first traversal of the given graph.

Q28: A drunk man is walking on the road and taking random turns when they are available, although he is conscious enough to remember the streets visited, and so, he does not follow the same path twice while turning. Which traversal is the man following?

Ans:   
DFS

**✓ Correct**

**Feedback:**

The drunk man taking random turns when they are available can be compared to choosing random neighbours while traversing in a graph. Also, from the starting point, he is not exploring all the adjacent paths and walking on a random path that he has not walked before. So, the traversal followed by the drunk man is a depth-first search.

Q29: Which of the following would always be the better option to find the shortest path between the source node and the destination node in an unweighted graph?

Ans: BFS

**✓ Correct**

**Feedback:**

Using a breadth-first search, you can traverse the graph level-wise and check all the nodes at that level with the destination node. However, using a depth-first search, you can traverse through the graph in one direction only at once. Go through the link given in the text below the question to know more about how to find the shortest path using a breadth-first search.

For the Java implementation of breadth-first search and to classify the different levels of nodes based on breadth-first traversal, please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Additional Resources - Graphs and Graph Algorithms -> Breadth-First Search (BFS) - II.

To know how to find the shortest path between the source node and the destination node in an unweighted graph, please visit [this](https://www.geeksforgeeks.org/shortest-path-unweighted-graph/)website.

For the industry demonstration of breadth-first traversal on a social network, please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Additional Resources - Graphs and Graph Algorithms -> Industry Demonstration - I.

For the industry demonstration of depth-first traversal on a social network,  please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Additional Resources - Graphs and Graph Algorithms -> Industry Demonstration - II.

# Summary

In this session, we covered the following topics:

* What is a graph abstract data type?
* Different types of graphs:
  + Undirected graphs
  + Directed graphs:
    - Directed acyclic graphs
* Differences between graphs and trees
* Depth-first search:
  + Pseudocode
* Breadth-first search:
  + Pseudocode

Q30: Consider a simple undirected and connected graph of 10 nodes and 10 edges. What is the maximum number of cycles possible in this graph?

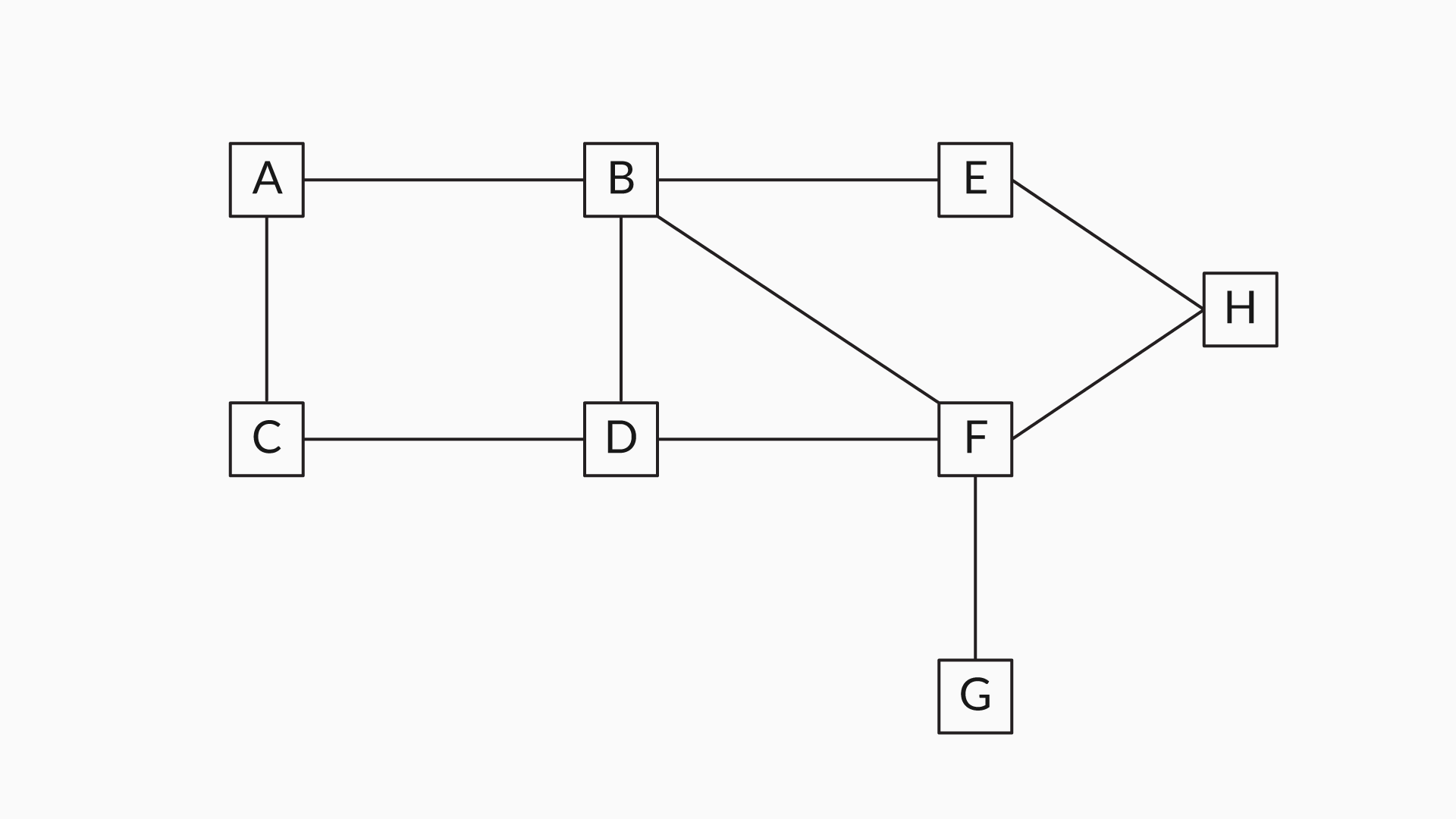
Ans:   
1

**✓ Correct**

**Feedback:**

While learning about the tree data structure, you have already learnt that when a tree has n nodes, it will have n-1 edges. In the question, it is mentioned that there are 10 nodes and 10 edges, and so, there is one more edge than in a normal tree. Therefore, only one cycle is possible in the graph structure.

Q31: How many different paths are available between nodes A and F in the graph given below?



Ans: 6

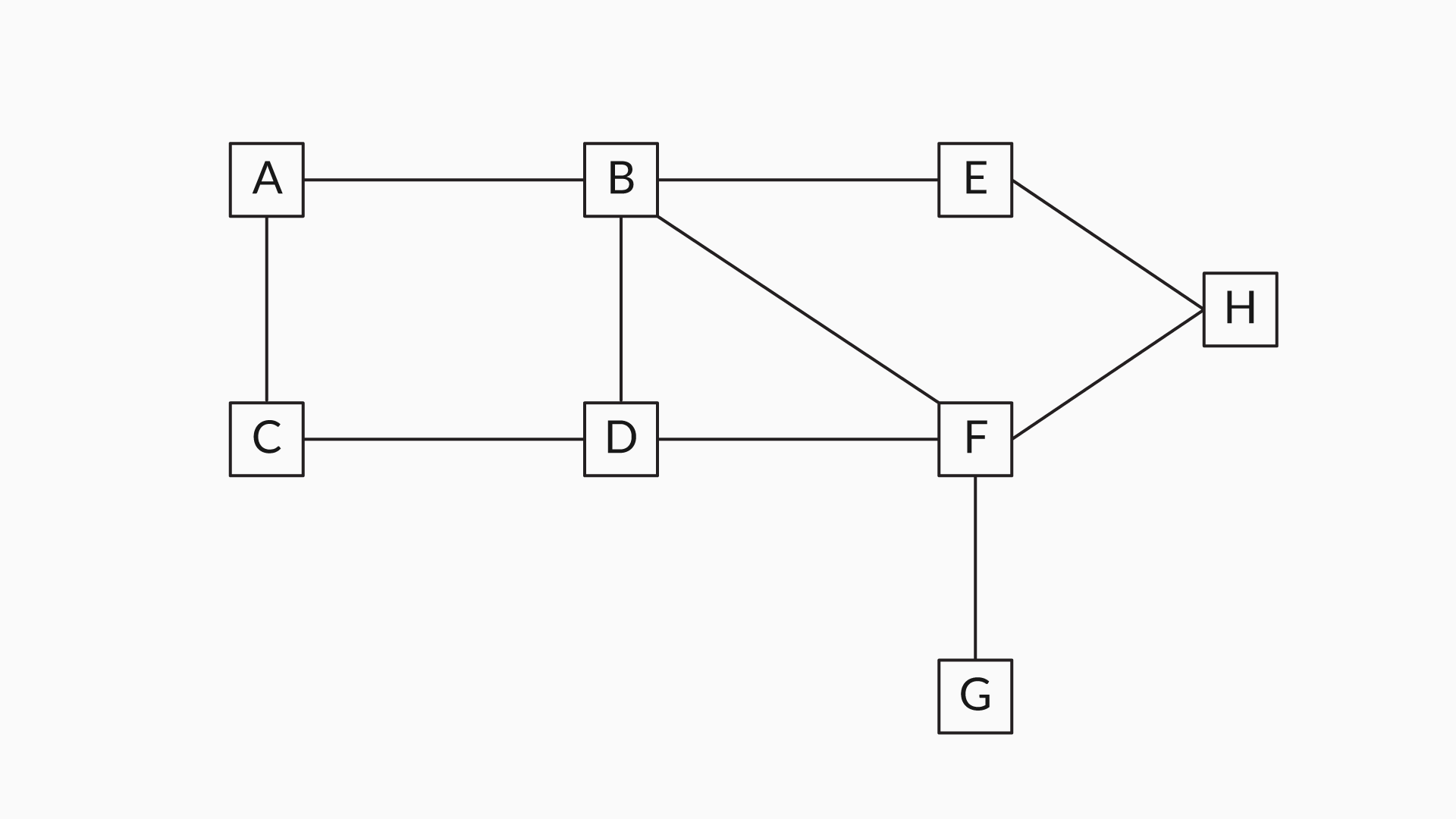
**✓ Correct**

**Feedback:**

In the given graph, node F can be reached from node A by a total of six different paths, which include the following:

1. A - > B -> F
2. A - > C - > D - > F
3. A - > B - > D - > F
4. A - > B - > E - > H - > F
5. A - > C - > D - > B - > F
6. A - > C - > D - > B - > E - > H - > F

Q32: Choose the correct possible order of visited nodes in the depth-first traversal of the graph below when the start node is A.



Ans: A C D F G B E H

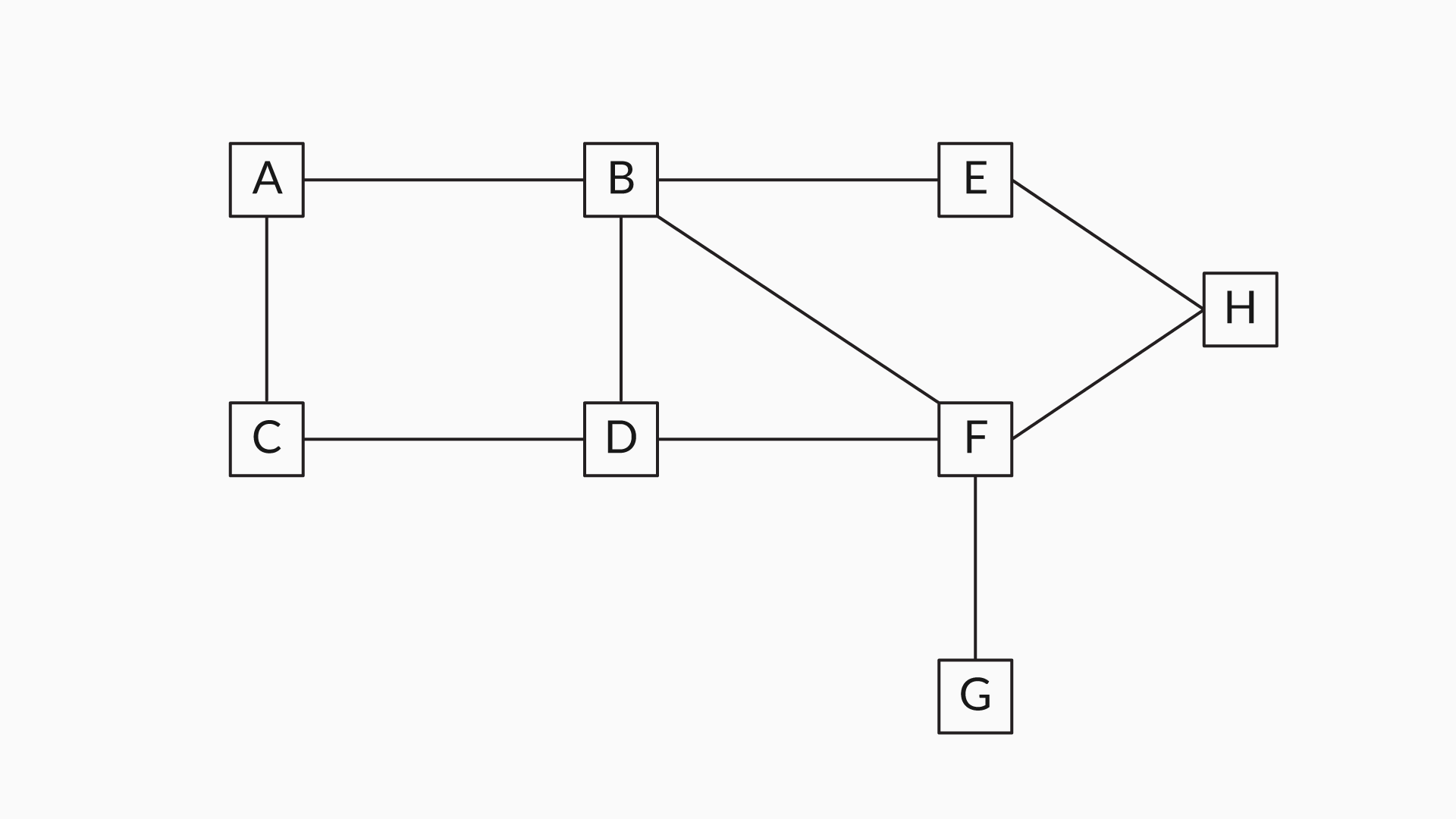
**✓ Correct**

**Feedback:**

* Neighbours of the start node A are {B, C}, and, at random, the next node to be visited can be chosen as node C.
* The unvisited neighbour of node C is {D} and so, so the next node to be visited is D.
* Unvisited neighbours of node D are {B, F}, and, at random, the next node to be visited can be node F.
* Unvisited neighbours of node F are {G, B, H}, and, at random, the next node to be visited can be node G.
* There are no unvisited neighbours of node G and so, it backtracks to node F.
* Unvisited neighbours of node F are {B, H}, and, at random, the next node to be visited can be node B.
* The unvisited neighbour of node B is {E}, and the next node to be visited is node E.
* The unvisited neighbour of node E is {H}, and the next node to be visited would be H.

This completes the depth-first search of this graph when the start node is A.

Q33: Choose the correct possible order of visited nodes in the breadth-first traversal of the graph given below when the start node is A.



Ans: A C B D E F H G

**✓ Correct**

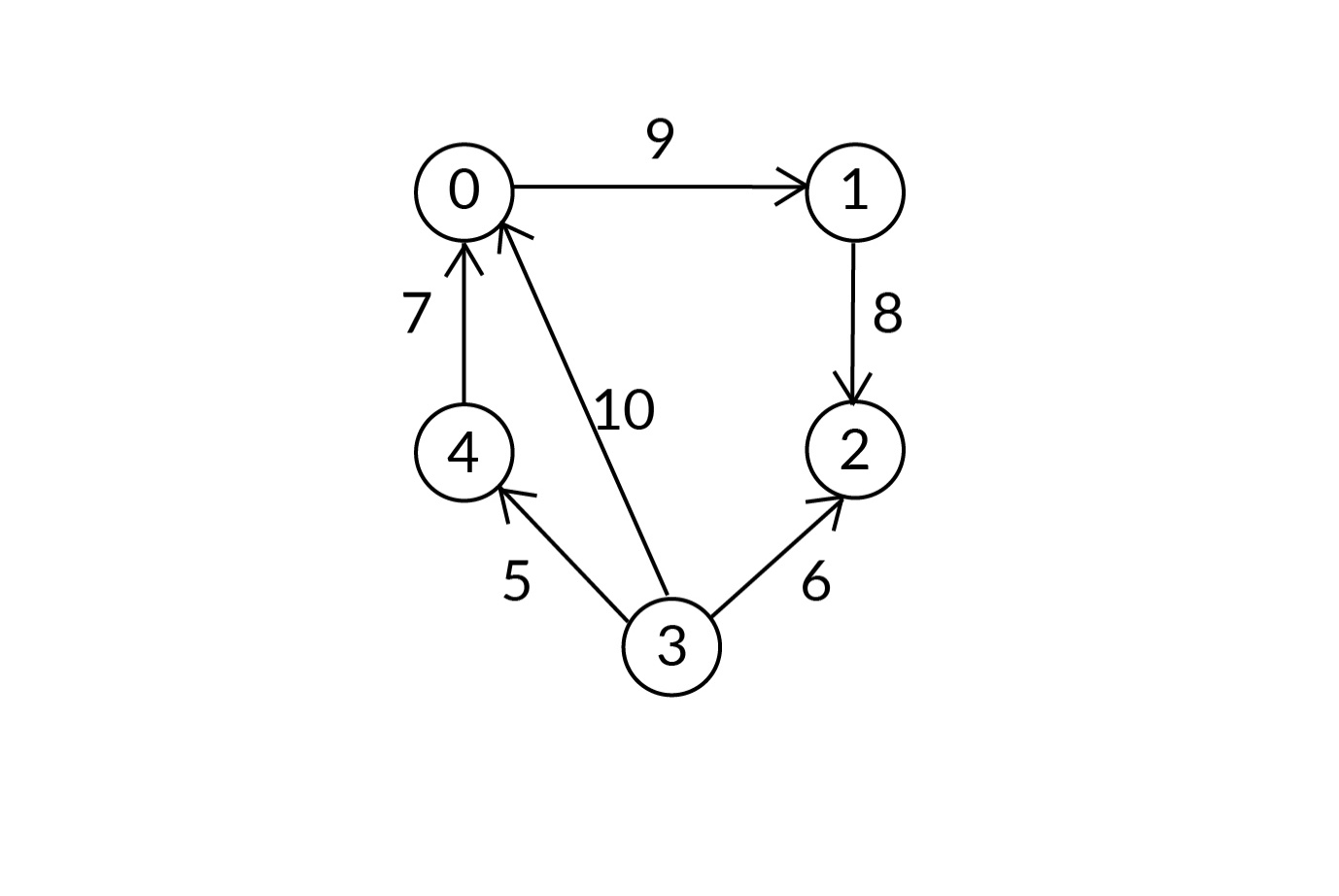
**Feedback:**

* Neighbours of the start node A are {B, C} and both the neighbours are visited first, which can possibly be in the order of A C B.
* Then it traverses first to the unvisited neighbour of node C, which is {D}, after which it traverses to the neighbours of node B, which are {E, F}. The order followed so far is A C B D E F.
* Next, it traverses to the neighbour node of E, which is {H}, after which it traverses to the neighbour node of F, which is {G}.

The possible order of breadth-first traversal is A  C  B  D  E  F  H  G.

# **Introduction to Edge Lists**

Q34: Can you think what the edge list representation of the graph below would be?



Ans: Set of nodes: {0, 1, 2, 3, 4}

Set of edges: {(0, 1, 9), (1, 2, 8), (3, 2, 6), (3, 0, 10), (4, 0, 7), (3, 4, 5)}, i.e., (0, 1, 9) indicates that there is an edge between ‘0’ and ‘1’ with edge weight as 9.

# **Introduction to Adjacency Matrix**

Q35: Why do you need to use a two-dimensional matrix in an adjacency matrix implementation?

 Ans: We use a two-dimensional matrix in an adjacency list implementation as we are representing the nodes in the rows as well as the columns. If there is an edge between the nodes, then we mark the intersecting cell as ‘true’.

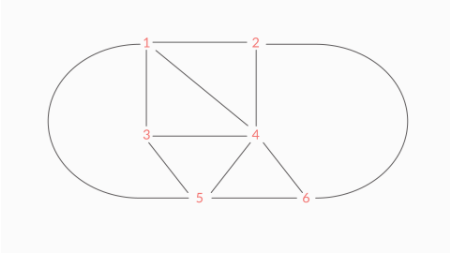
Q36: You can ignore the symmetrical half of the matrix in the case of an adjacency matrix implementation. Can you explain why?

Ans: We can ignore one half of the matrix, i.e., the symmetrical half, since there is an edge between x and y. Also, since we have marked cell (x, y) as ‘true’, we do not need to mark cell (y, x) the same because both the cells refer to the same edge in an undirected graph.

you saw how an adjacency matrix can be represented using a two-dimensional matrix of boolean values.

**Note:** Adjacency matrix is symmetric only in the case of an undirected graph, whereas in the case of a directed graph, it is non-symmetric.

#### Q37: Graph Algorithm



For the given graph, if the adjacency matrix is 1-indexed, then what would the values in the following be?

* Row 4
* Columns 1–6

**Note:** 1-indexed means the counting of rows or columns starts from the natural number 1 in the adjacency matrix.

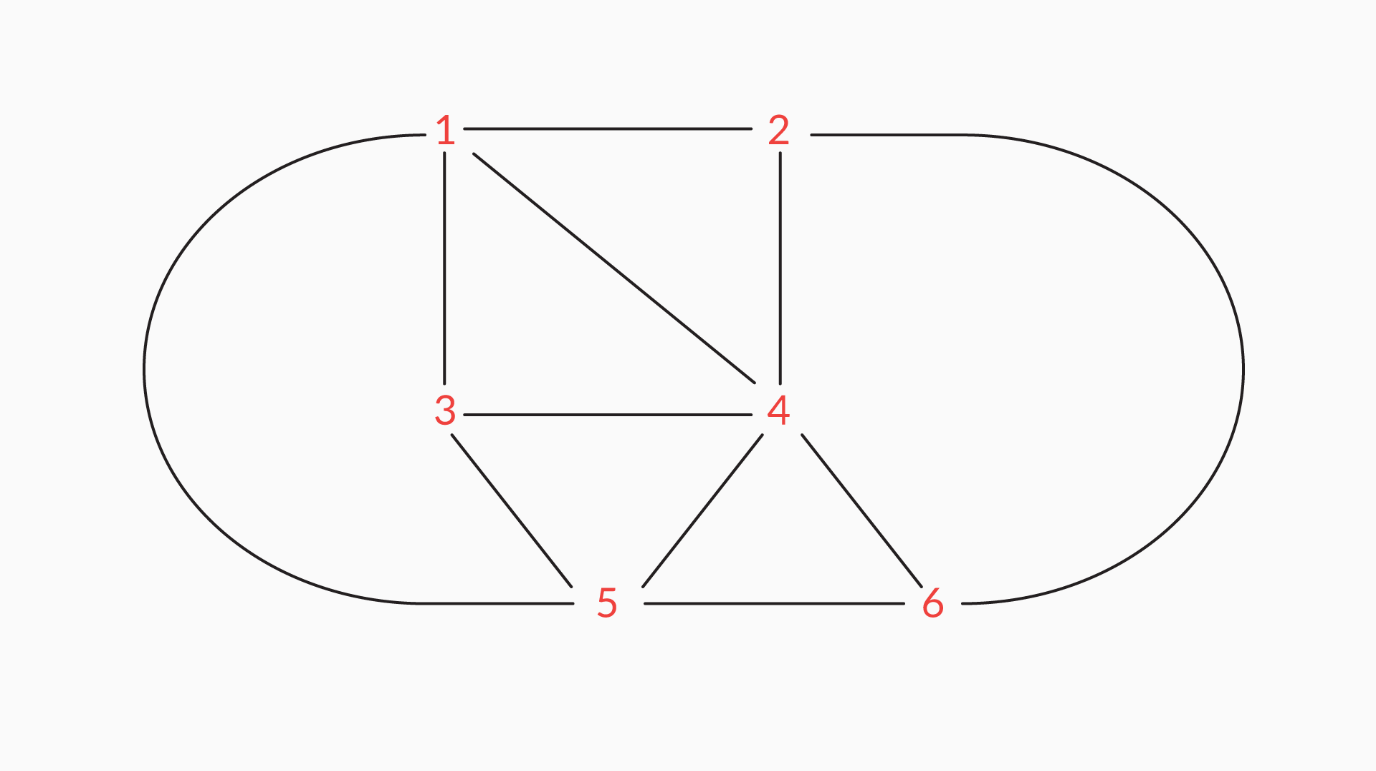
Ans: True True True False True True

**✓ Correct**

**Feedback:**

Here, since index 4 represents node 4 and 4 is connected to nodes 1, 2, 3, 5 and 6, columns 1, 2, 3, 5 and 6 of row 4 would have true boolean values in their respective cells.

#### Q37: Graph Algorithm



In the graph above, would the values in the columns of the row with the index 6 change if you remove the edge between 2 and 6?

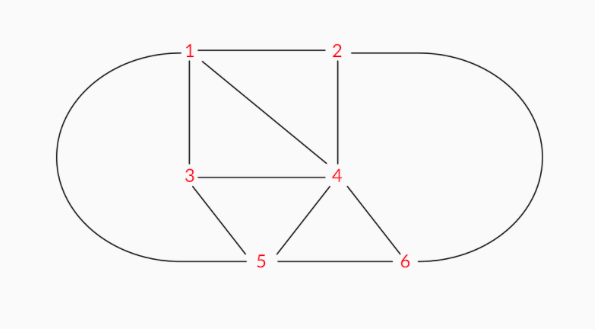
Ans: Yes

**✓ Correct**

**Feedback:**

If you remove the edge between 2 and 6, then the value in the cell at the intersection of the row indexed 6 and the column indexed 2 would change to ‘False’ since the edge no longer exists.

#### Q38: Graph Algorithm



In the graph above, how many true values would become false if you remove node 4?

Ans: 10

**✓ Correct**

**Feedback:**

If we remove node 4 from the graph, then all the true values in the row indexed 4 and the column indexed 4 should become false. So, there are three rows above 4 that contain true values: row 1, row 2 and row 3. There are two rows below 4 that contain true values: row 5 and row 6. This makes it five true values.

Then there are three columns to the left of row 4 that contain true values: column 1, column 2 and column 3. There are two columns to the right of column 4 that contain true values: column 5 and column 6. This makes it five true values again. So, you get a total of 10 true values, which would become false if you remove node 4 from the graph since node 4 is connected to all the other nodes.

#### Q39: Graph Algorithm

An adjacency matrix is always symmetric.

Ans: False

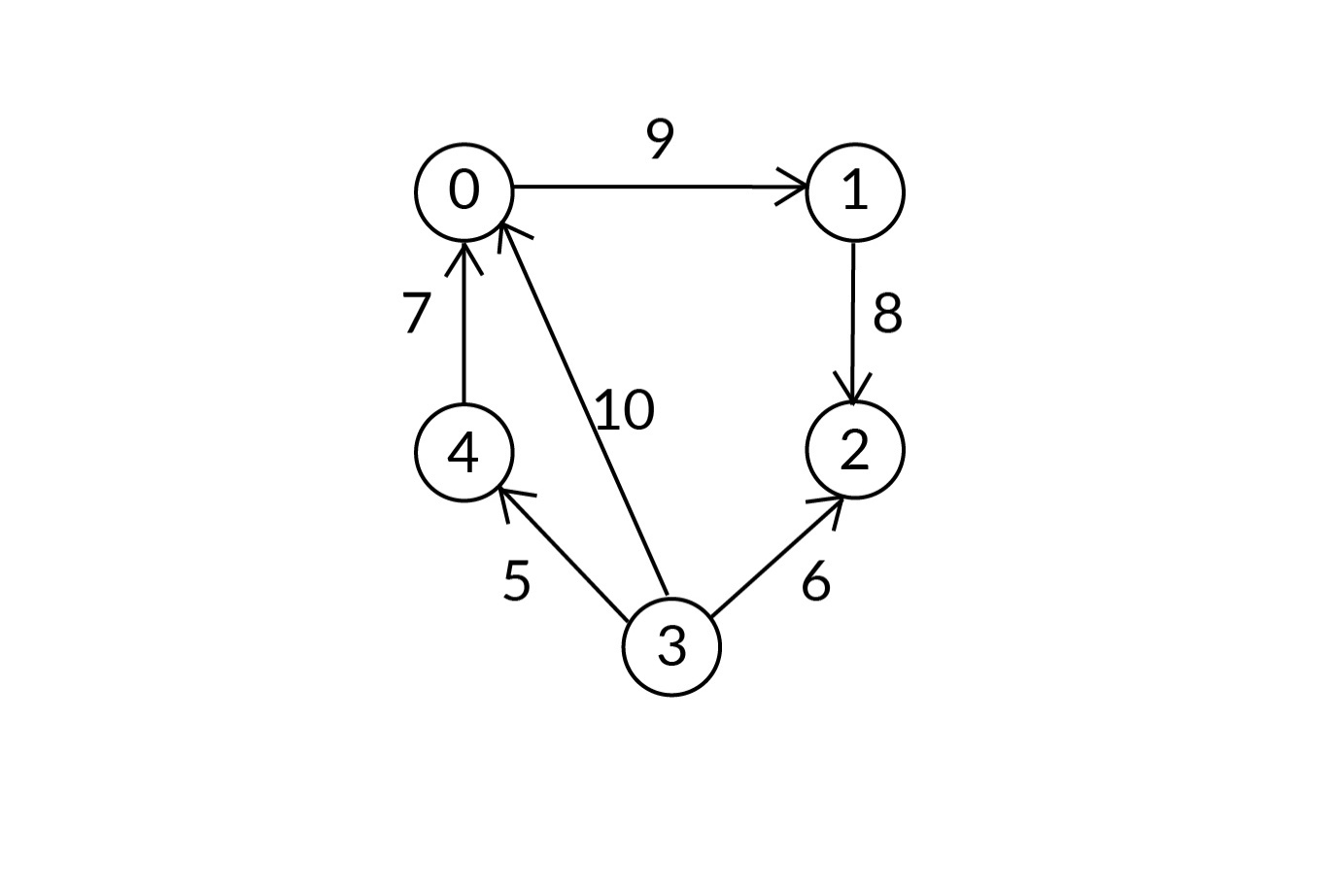
**✓ Correct**

**Feedback:**

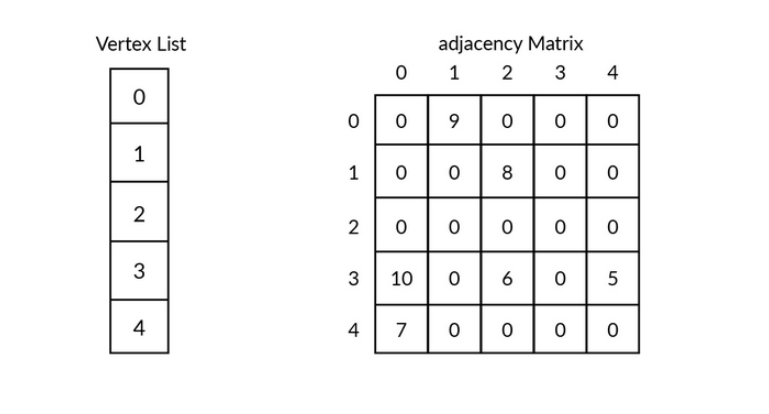
An adjacency matrix is symmetric only in the case of a undirected graph, whereas in the case of a directed graph, it is non-symmetric.

#### Q40: Graph Algorithm

Can you think what the adjacency matrix representation of the graph below would be?



Ans: **Suggested Answer**



In the adjacency matrix above, the (0, 1)th cell is ‘9’, which implies that there is an edge from node 0 to node 1, with edge weight as 9, and the (1, 0)th cell is ‘0’, which implies that there is no edge from node 1 to node 0.

To know how an adjacency matrix can be implemented in Java, please refer to the following path:

Resources -> Additional References - Data Structures and Algorithms -> Additional Resources - Graphs and Graph Algorithms -> Adjacency Matrix Implementation.

# **Performance Characteristics of an Adjacency Matrix**

The most important aspect of any algorithm is its performance characteristics, which determine how it would perform in a given situation or under given circumstances. Therefore, you need to take into account an algorithm’s performance when you make your choice.

So, in the forthcoming video, we will calculate the time complexity of certain operations on an adjacency matrix. Specifically, we will calculate the time complexity of the following four operations:

* getAllNodes: Get all the nodes of the graph
* addNode: Add a node to the graph
* addEdge: Add an edge between two specified nodes
* getAllNeighbours: Get all the neighbours of a specified node

Q41: Having understood the time complexity of the addEdge method, what do you think the time complexity of the getAllNeighbours method would be?

Ans: O(V)

**✓ Correct**

**Feedback:**

When you want to find out what the neighbours of a particular node are, you have to find its row in the matrix. This requires O(V) time, which is similar to the addEdge method. Then, in the same row, you have to check each column for true or false boolean values. If a row contains a true boolean value, then you have to fetch this node (value) and say that it is a neighbour of the given node. Again, this process will require you to traverse the entire row one by one. This is also an O(V) process. So, getAllNeighbours has an O(V) worst-case time execution.

Q42: Edge lists have better space complexities compared with adjacency matrixes. Justify this statement in your own words.

Ans: In the case of an edge list, the space needed to store the list is actually directly proportional to the number of edges. So, if the number of edges is E, then the space complexity would be O(E) or the number of edges in the graph. But when it comes to an adjacency matrix representation, the space required is directly proportional to the square of the number of nodes in the graph. Therefore, if the number of nodes is, say, N, then the space complexity would be O(N\*N). Hence, you can say that edge lists have better space complexities.

**Note:** At 4:15 the professor intends to say that Add Node method has a time complexity of O(V \* V) and Add Edge method has a time complexity of O(V).

So, in the video, you learnt:

* How to calculate the time complexity of certain important operations on an adjacency matrix.
* The practices that you can follow in order to improve the time complexities of the following methods: addEdge and getAllNeighbours.

**What are Dense Graphs and Sparse Graphs?**

Let us now discuss ‘dense graphs’ and ‘sparse graphs’ in detail.

**Dense graphs:** A dense graph is a graph in which the number of edges is close to the maximum possible edges for a given set of nodes. Given below is an example of a dense graph.

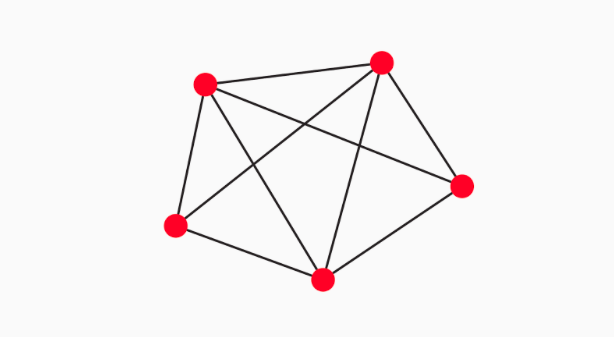


Figure 1

**Sparse graphs:** Sparse graphs are connected graphs with minimum or a small number of edges connecting the nodes. In sparse graphs, there may or may not be an edge between two nodes. Here, usually, the number of edges is n, which is also the number of vertices.

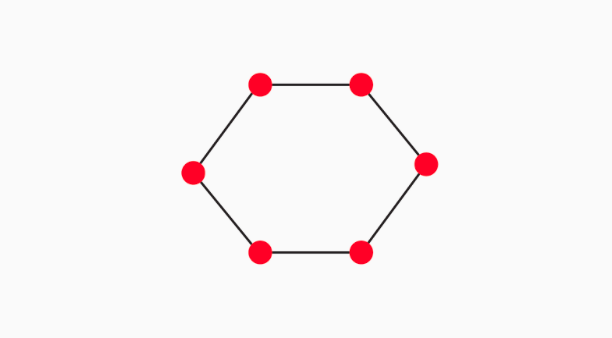


Figure 2

Q43: What is the time complexity of removing an edge from an adjacency matrix, given the vertices are integral values?

Ans: O(1)

**✓ Correct**

**Feedback:**

To remove an edge that is between two nodes in an adjacency matrix, you simply need to update the true boolean value to a false boolean value for the cell corresponding to the two nodes. For example, if you want to delete the edge between vertices v1 and v2 in an adjacency matrix, then you can do so by setting adjacency\_matrix[v1][v2] to ‘0’. Since the vertices are integral values, this takes a constant amount of time, i.e., O(1).

Q44: Which representation would you use to represent a sparse graph?

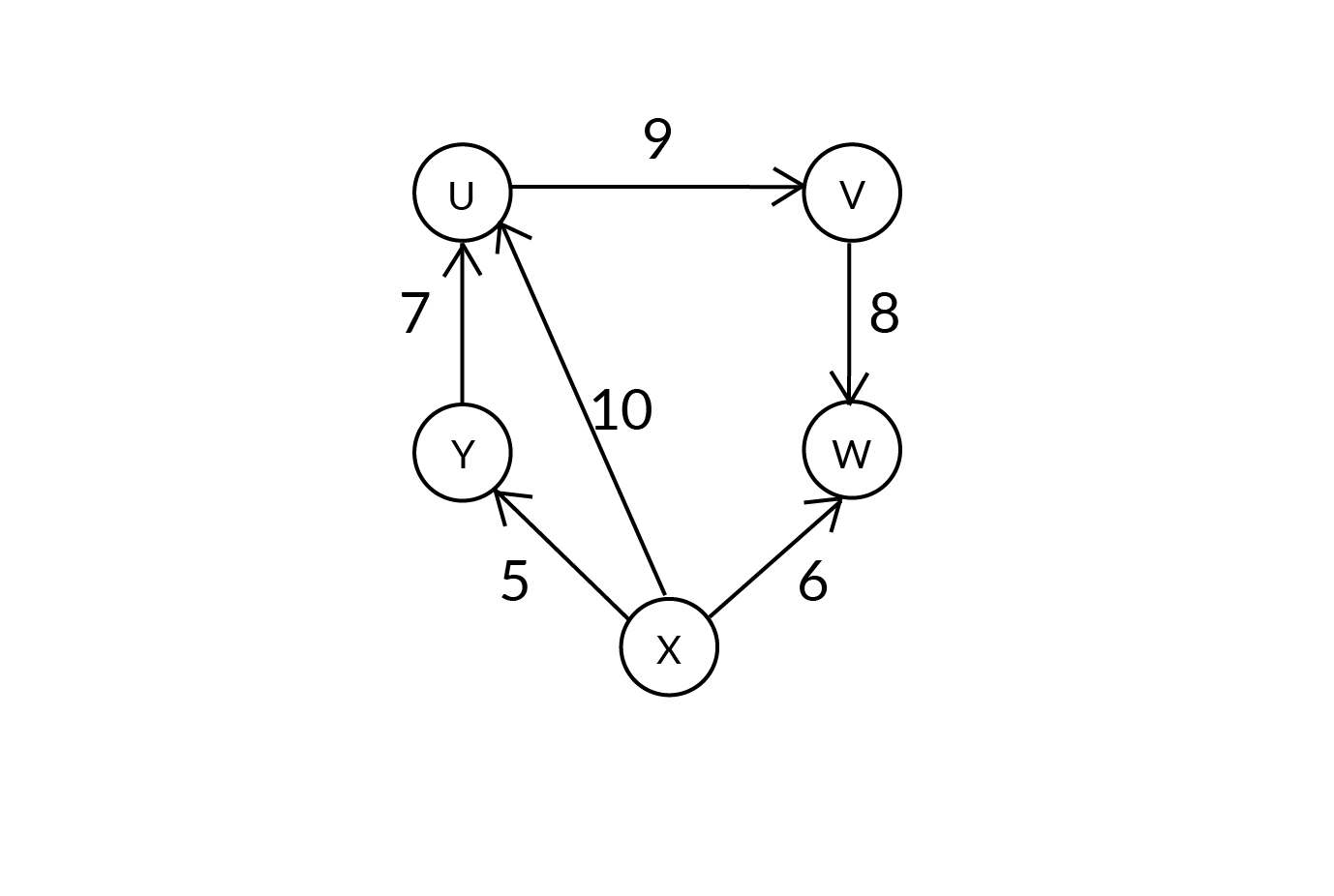
Ans: Edge list

**✓ Correct**

**Feedback:**

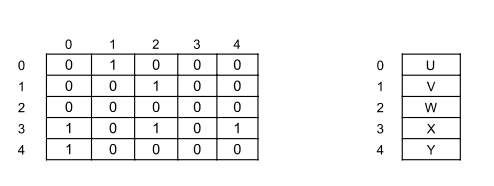
Adjacency matrix, in any situation, would take O(n\*n) space as it is represented using a two-dimensional matrix. Therefore, an adjacency matrix would waste a lot of space when used to represent a sparse graph. However, this is not the case with an edge list. Hence, you choose an edge list over an adjacency matrix to represent sparse graphs.

what the time complexity of removing an edge from an adjacency matrix is if all the vertices are not integral values.



Weighted graph

The adjacency matrix and vertex list of the above weighted directed graph are given below.



Adjacency Matrix and Vertex List

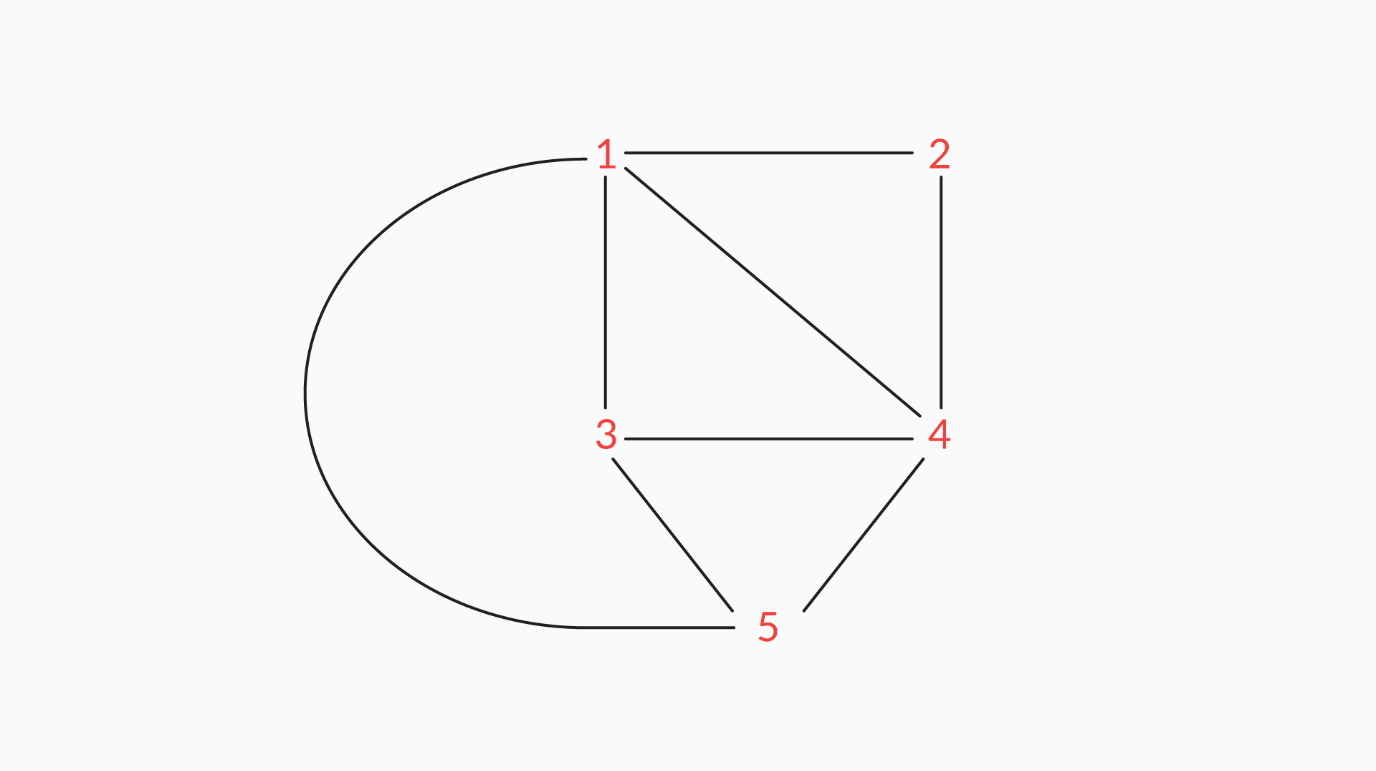
If you, for instance, want to remove an edge from X to U, then:

1. You have to traverse through the vertex list and find the indexes of X and U; here, they are 3 and 0, respectively:
   * If the size of the vertex list is V, then this step takes linear time, which is O(V) in the worst case.
2. Now, set that (3, 0)the cell to ‘0’:
   * This step takes constant time.

So, the total time taken to remove an edge is O(V).

# **Introduction to Adjacency Lists**

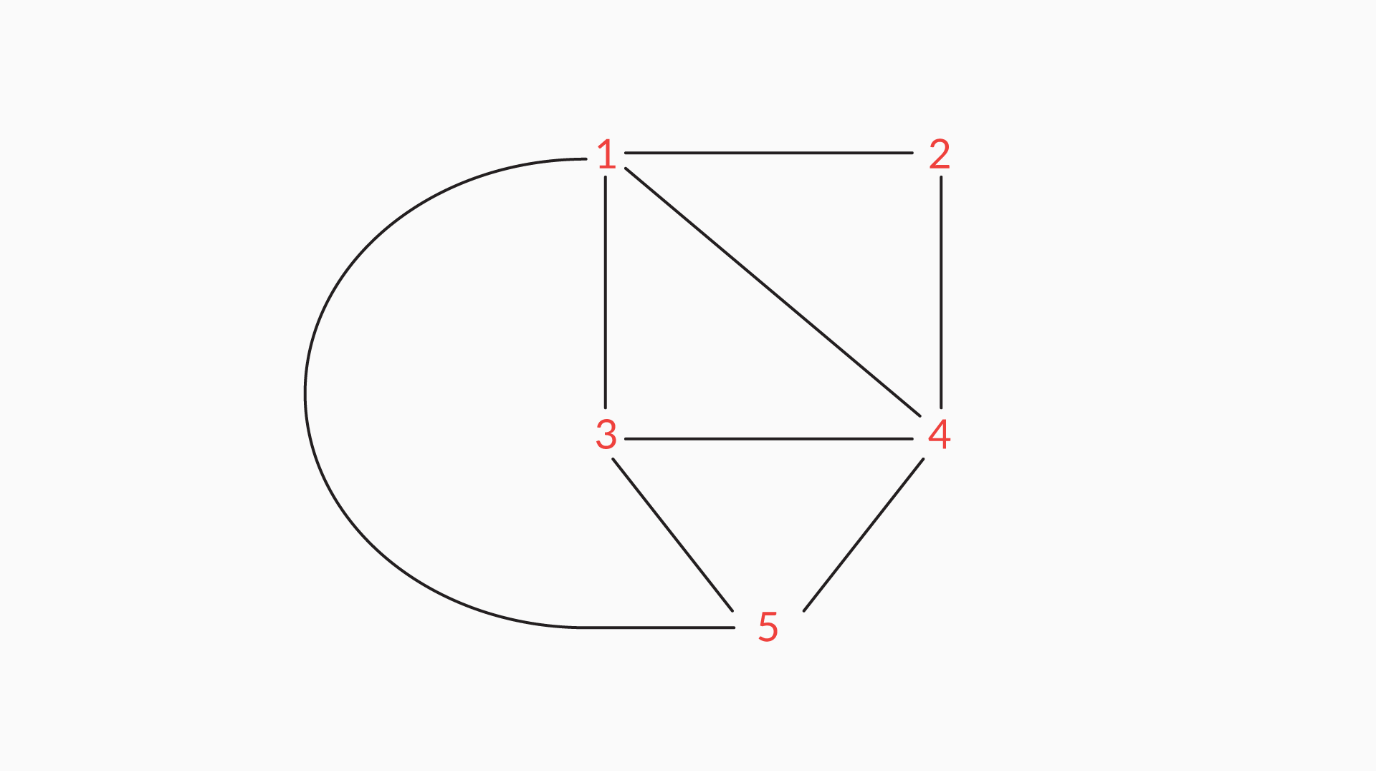
#### Q45: Graph Algorithm



In the given graph, which nodes are neighbours of 1, and why?

Ans: Nodes 2, 3, 4 and 5 are neighbours of 1, because there is a direct edge between nodes 1 and 2, nodes 1 and 3, nodes 1 and 4, and nodes 1 and 5.

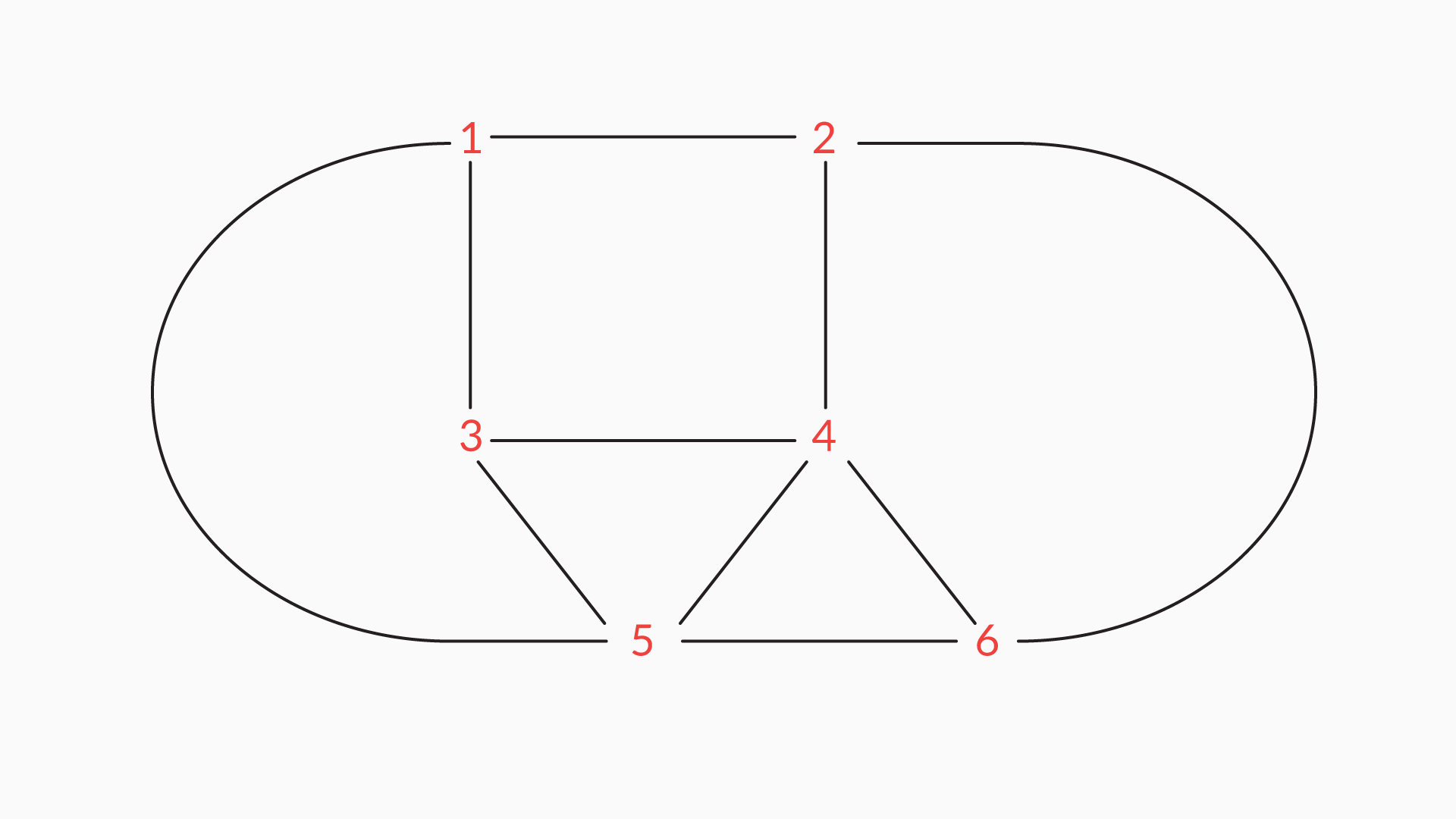
#### Q471: Graph Algorithm



In the given graph, how many neighbours does node 5 have and which are these neighbours?

Ans: Node 5 has three neighbours: 3, 4 and 1. This is because there is a direct edge between nodes 5 and 3, nodes 5 and 4, and nodes 5 and 1.

#### Q48: Graph Algorithm



For the given graph, write down all the values that will be present in the sets of all the nodes from 1 to 6. For example, for node 1, the set would be 2 3 5.

Ans: For node 1, the set would be 2 3 5.

For node 2, the set would be 1 6 4.

For node 3, the set would be 1 4 5.

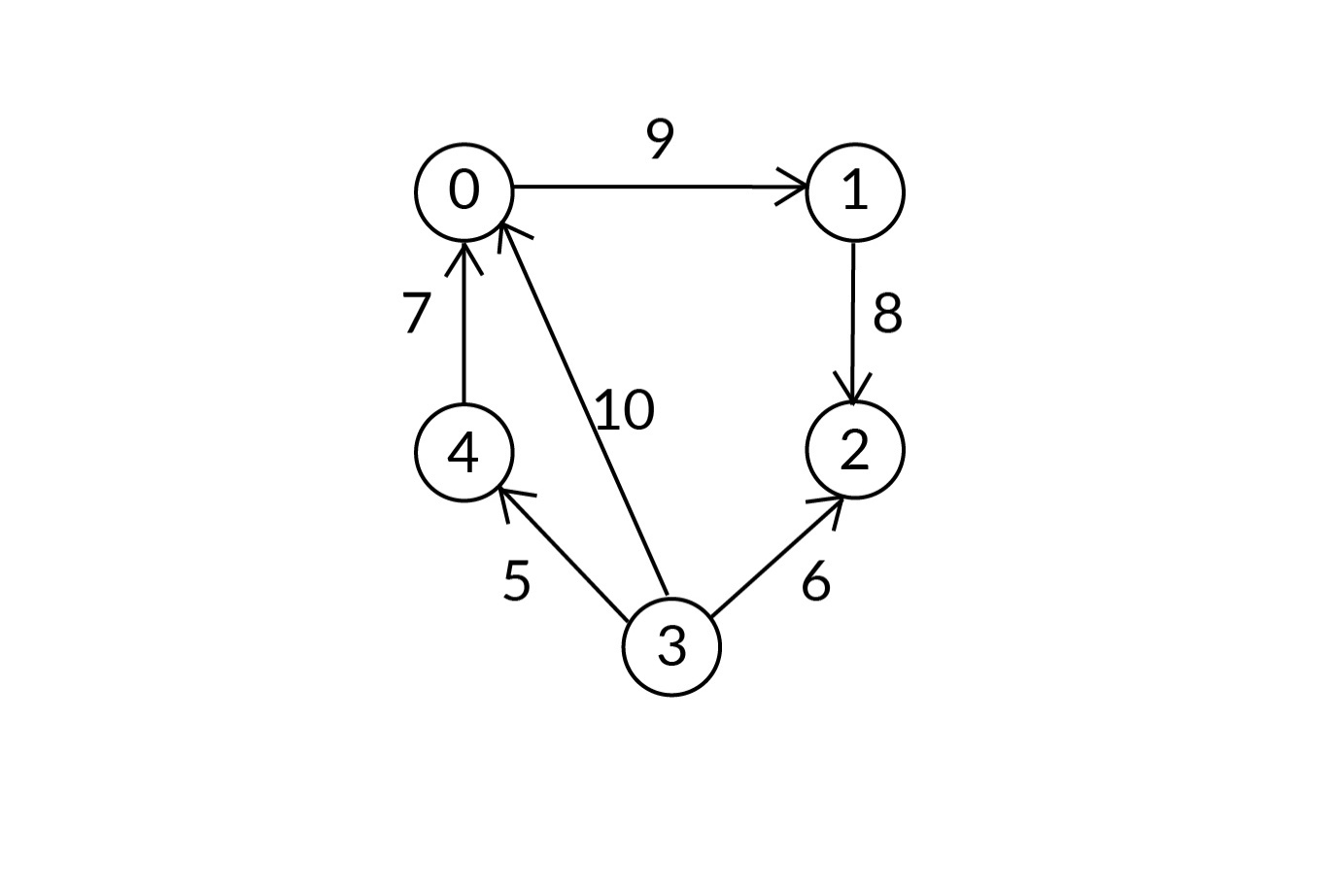
For node 4, the set would be 2 3 5 6.

For node 5, the set would be 1 3 4 6.

For node 6, the set would be 2 4 5.

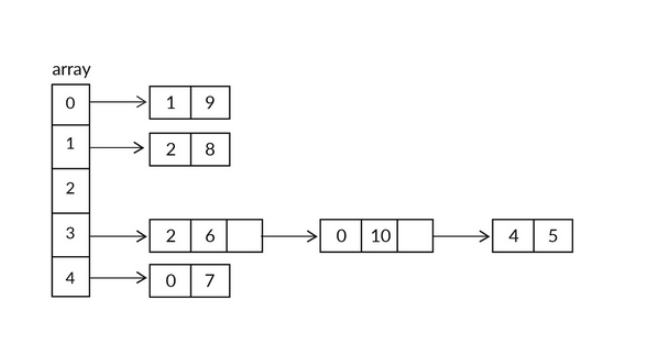
#### Q48: Graph Algorithm

Can you think what the adjacency list representation of the weighted directed graph below would be?



Ans: In the adjacency list above, we store only those nodes that are connected to a particular node. It is implemented using an array of lists, where the index of the array represents a node (since the vertices are integral values), and it contains a list of all the nodes connected to that node along with weight of the edge.

* In the above adjacency list, zeroth index of the array is mapped to node 1
  + This indicates that there is an edge from vertex ‘0’ to ‘1’
* The 9 beside 1 indicates the weight of the edge.



#### Q48: Graph Algorithm

In the case of a dense graph, i.e., a graph with many edges, which graph implementation should you choose?

Ans: Adjacency matrix

**✓ Correct**

**Feedback:**

Adjacency Matrix is more suited to represent dense graphs.

Q49: What is the time complexity of the getAllNodes method for the edge list, adjacency matrix and adjacency list implementations of a graph where V is the number of vertices or nodes?

Ans: O(1), O(V), O(1)

**✓ Correct**

**Feedback:**

The time complexity of the getAllNodes method for an edge list is O(1). For an adjacency matrix, it is O(V) and for an adjacency list, it is O(1) (where V is the number of vertices or nodes). In the case of an edge list, the method takes a constant time, i.e., O(1), for all the edges, because you can go to the cell corresponding to the nodes directly, retrieve its value, and check whether the value is true or false. In the case of an adjacency matrix, since we have to traverse for each row of the 2D array, and since the number of rows of the 2D arrays equals the number of vertices in the graph, it takes O(V) time. In the case of adjacency list, we are just returning a keyset of the hash map and not traversing it; so, it is just an O(1) operation because keySet() of hashMap is an O(1) operation.

Q49: For some sparse graphs, an adjacency list is more space-efficient compared with an adjacency matrix.

Ans: True

**✓ Correct**

**Feedback:**

Since a sparse graph has just a few edges and more nodes, it is better represented using an adjacency list, since the space complexity of an adjacency list, in this case, would be O(V). However, the space complexity of an adjacency matrix is always O(V\*V).

Q50: In which cases is it preferable to use the adjacency list implementation of a graph, and in which cases is it preferable to use the adjacency matrix implementation?

Ans: Adjacency lists use memory in proportion to the number of edges. This can help save a lot of memory if the adjacency matrix is sparse, i.e., if the graph does not have many edges. However, the trade-off is that finding the presence or absence of a specific edge is slightly slower in an adjacency list than in an adjacency matrix. An adjacency matrix has fast lookups to check for the presence or the absence of a specific edge, although it is slow when it comes to iterating over all the edges, and it requires O(n\*n) memory. So, it all depends on the kind of problem that you’re trying to solve or the graph that you’re working with.

Q51: Adding a vertex in an adjacency matrix is easier than doing so in an adjacency list.

Ans: False

**✓ Correct**

**Feedback:**

To add a vertex to an adjacency matrix, you need to call the addNode method, in which you need to create a new 2D array to account for the newly added vertex. However, in an adjacency list, you just need to push the nodes into the linked list of nodes.

Q52: For a website such as Facebook, which graph representation would you use and why?

Ans: Facebook has ~2.07 billion users. So, if we were to use an adjacency matrix, then it would take up too much memory. People generally have a few thousand friends (or edges) at most and so, an adjacency list representation is preferred in this case. Specifically, the space requirement for an adjacency list is O(V+E), whereas it is O(V^2) for an adjacency matrix. Therefore, as you can see, an adjacency list uses lot lesser space than an adjacency matrix in this case. Or, as a general rule of thumb, use adjacency lists for sparse graphs and adjacency matrices for dense graphs.

Q53: What would the time complexity be to find an edge between two elements in an adjacency list that has E edges and V vertices?

Ans: O(V)

**✓ Correct**

**Feedback:**

In order to check whether there is an edge between two elements, first you need to go to element 1 in the vector, which is O(1), and then traverse its linked list of connected nodes, which is O(V) in the worst case. Therefore, the total time complexity is O(V).

# **Introduction to Dijkstra's Algorithm**

Whenever you go from one place to another, you always wonder about the route that you must take in order to reach your destination in minimum time and at the least expense.

In this regard, Google Maps has become a part of our day-to-day lives. The application comes to the rescue whenever we want to travel.

However, millions of people use Google Maps, and so, it is not possible for the application to perform all the calculations to find the shortest possible route and display the same in a matter of a just a few seconds to each person who uses the tool and for each request. So, what makes it possible? Well, there is an algorithm at work behind each request that you send to Google Maps, and it is popularly known as ‘Dijkstra’s algorithm’.

Q54: Dijkstra’s algorithm can work with directed graphs only and not with undirected graphs.

Ans: False

**✓ Correct**

**Feedback:**

The algorithm can be applied to both, because a undirected graph is basically the same as a directed graph. A undirected graph just has two connections in opposite directions between the connected nodes. So, all you need to do is be aware of all the connections between all the nodes that can be reached from every given node, using, for example, an adjacency list.

Q55: Why do you store infinity or some high value in the table of distances initially?

Ans: We store some high value in the table of distances initially because we have to find the shortest path from one node to another. So, we update the table only when we get a path with a cost smaller than the cost that is already mentioned in the table. Therefore, by setting the initial values as infinity, we will ensure that the state table of the shortest path is updated with the correct values, no matter how large these values are.

**Note:** Dijkstra's algorithm can be used with both directed and undirected graphs.

So, in the video:

1. You learnt about a new algorithm, Dijkstra’s algorithm, which calculates the shortest path from a source to a destination.
2. You also learnt that you need two data structures to implement this algorithm: a one-dimensional array and a priority queue.

Q56: What does Dijkstra’s shortest path algorithm do?

Ans:   
Dijkstra’s shortest path algorithm calculates the shortest distance from a given node to all the other nodes in a graph.

**✓ Correct**

**Feedback:**

Correct. Dijkstra’s shortest path algorithm calculates the shortest distance from a given node to all the other nodes in a graph.

Q57: Dijkstra’s algorithm works with which two data structures?

Ans: Array of distances, priority queue

**✓ Correct**

**Feedback:**

Dijkstra’s algorithm uses the ‘array of distances’ to store the shortest distance and the ‘priority queue’ to sort the nodes in ascending order of edge weight.

# **Dijkstra’s Algorithm**

Q57: When should you apply the concept of ‘edge relaxation’ when using Dijkstra’s algorithm?

Ans: If we want to go from node n1 to n2, and if the distance value currently recorded in the table of n2 is greater than the sum of the distance value of n1 and the edge weight of the edge flowing from n1 to n2, then we would apply edge relaxation and replace the value of n2 in the table.

Q58: Elements from the priority queue are dequeued in the order of smallest to largest weight in Dijkstra’s algorithm.​​​​​​

Ans: True

**✓ Correct**

**Feedback:**

Since Dijkstra’s algorithm is a greedy algorithm, it will try to pick the node with the least edge weight, to find the shortest path. Hence, the priority queue is ordered from the smallest to the largest weight in the algorithm.

**Note:** In the video, at 7:17, Step 5 of Relaxation is given as C3 -> C4 = 70 + 10 = 80, but it should be C3 -> C5 = 70 + 10 = 80.

So, in the video, you learnt the following:

1. The concept of edge relaxation
2. How to make conditional updates to the table of distances, i.e., how relaxation works

The pseudocode of Dijkstra’s algorithm is given below.

Procedure **Dijkstra**(graph, node)

Perform edge relaxation **for** all the outgoing edges from the starting node

  While Q is not empty

      Dequeue the first element from priority queue

      nextNode ← front element of queue after previous deque

**if** nextNode is not **null**

        Relax the edges **if** necessary

      end **if**

   end **while**

end procedure



**Step 1:**First, perform edge relaxation for all the outgoing edges of the starting node.

**Step 2:**The ‘while’ loop instruction set is executed when the queue is not empty.

**Step 3:** For each iteration of the ‘while’ loop, the node at the front gets dequeued.

**Step 4:** nextNode will store the node at the front of the priority queue after the last deque.

**Step 5:** We will check whether the value of nextNode is null or not. If it is not null, then we will proceed and perform edge relaxation on the nodes where required.

In this way, we will end up getting the shortest distances from the given node to all the other nodes in our table of distances.

Q59: Which data structure should you use if you have to find the shortest path in a graph in which all the edges have the same weight?

Ans: Queue

**✓ Correct**

**Feedback:**

If you have to find the shortest path in a graph in which all the edges have the same weight, then you will not require any data structure to find the minimum weight each time. Hence, you will not require a priority queue; you can use a regular queue. Also, you use a queue because Dijkstra’s algorithm behaves exactly like a breadth-first search if all the edges in the graph have the same weight.

Q60: Can you perform Dijkstra’s algorithm without using a priority queue?

Ans: Yes

**✓ Correct**

**Feedback:**

Yes, it is possible to perform Dijkstra’s algorithm without using a priority queue. Specifically, instead of using a priority queue to find the next node with the shortest path, you will have to check all possible nodes manually and find out the one with the smallest value among these. As you can see, this process is more tedious than using a priority queue.

Q61: What are the criteria for edge relaxation?

Ans: If the distance value in the D table for node n2 > the distance value in the D table for node n1 + the edge weight between nodes n1 and n2

**✓ Correct**

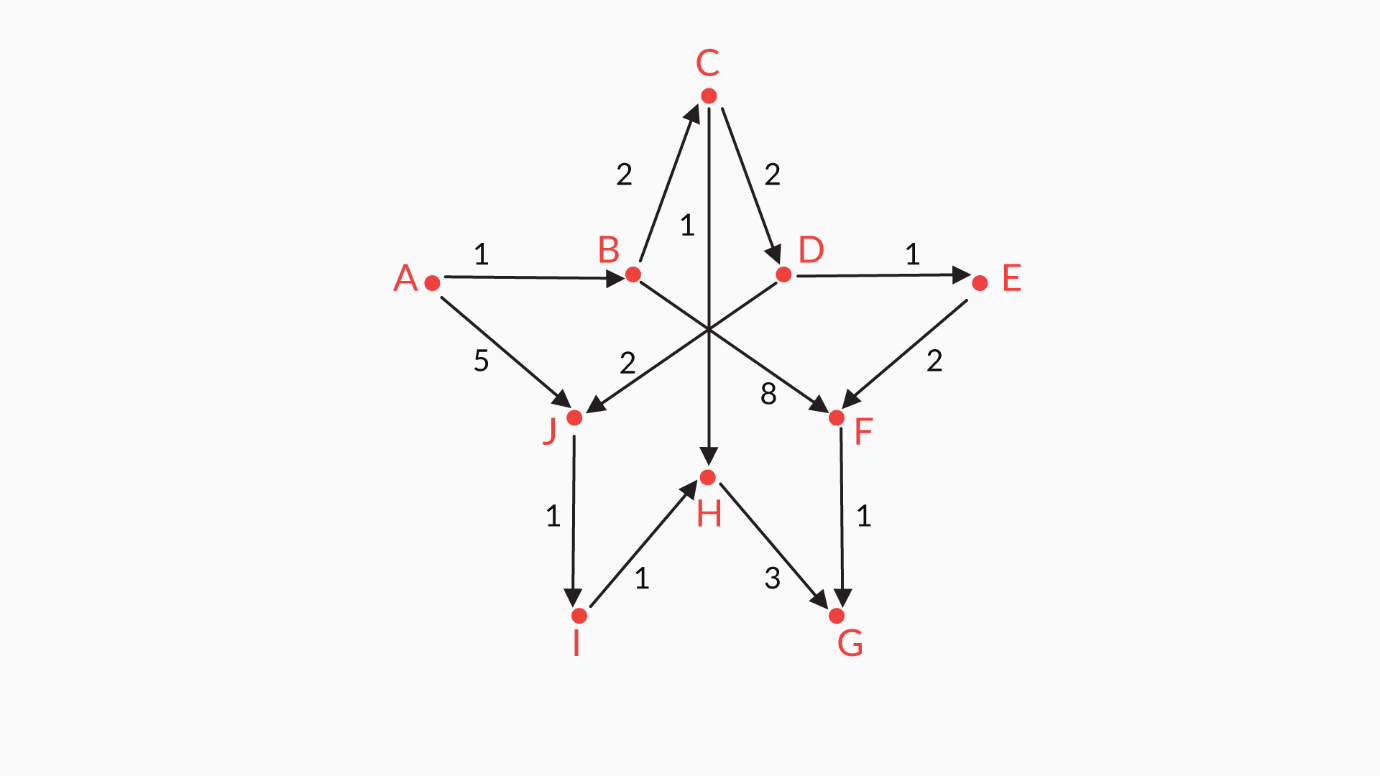
**Feedback:**

The criteria for edge relaxation is: if the distance value in the D table for node n2 > the distance value in the D table for node n1 + the edge weight between nodes n1 and n2.

# Summary

In this session, we covered the following:

* The three graph ADT implementations, which are as follows:
  + Edge list,
  + Adjacency matrix and
  + Adjacency list.
* You made a comparison between the time complexities of all the three graph implementations.
* You also learnt about Dijkstra’s algorithm:
  + Pseudocode
* You calculated the shortest path from one node to all the other nodes using Dijkstra's shortest path algorithm.



graph

Based on the graph above

Q62: Which graph ADT implementation would you use to represent this graph?

Ans: Adjacency list

**✓ Correct**

**Feedback:**

You would use an adjacency list to represent the graph above, since each node has a maximum of two neighbours, and you have 10 nodes. So, in order to have a better space complexity, you would use an adjacency list. Now, although edge lists are also space-efficient, adjacency lists are preferred because they are space-efficient and fast, i.e., O(1) in most cases.

Q63: What would the adjacency list representation for nodes A to E be?

Ans:   
A->

|  |  |
| --- | --- |
| B, 1 | J, 5 |

B->

|  |  |
| --- | --- |
| C, 2 | F, 8 |

C->

|  |  |
| --- | --- |
| D, 2 | H, 1 |

D->

|  |  |
| --- | --- |
| E, 1 | J, 2 |

E->

|  |
| --- |
| F, 2 |

**✓ Correct**

**Feedback:**

A is connected to B and J. The edge weight between A and B is 1, and between A and J, it is 5.

B is connected to C and F. The edge weight between B and C is 2, and between B and F, it is 8.

C is connected to D and H. The edge weight between C and D is 2, and between C and H, it is 1.

D is connected to E and J. The edge weight between D and E is 2, and between D and J, it is 2.

E is connected to F. The edge weight between E and F is 2.

Q64: What would the adjacency list representation for nodes F to J be?

Ans:   
F->

|  |
| --- |
| G, 1 |

G->

|  |
| --- |
|  |

H->

|  |
| --- |
| G, 3 |

I ->

|  |
| --- |
| H, 1 |

J ->

|  |
| --- |
| I, 1 |

**✓ Correct**

**Feedback:**

F is connected to G. The edge weight between F and G is 1.

G is connected to ‘no’ other node.

H is connected to G. The edge weight between H and G is 3.

I is connected to H. The edge weight between I and H is 1.

J is connected to I. The edge weight between J and I is 1.

Q65: How many paths are there from A to G?

Ans:   
5

**✓ Correct**

**Feedback:**

There are five paths between A and G, which are as follows:

* A -> B -> C -> D -> E -> F -> G
* A -> B -> F -> G
* A -> J -> I -> H -> G
* A -> B -> C -> D -> J -> I -> H -> G
* A -> B -> C -> H -> G

Q66: Which is the shortest path from A to G?

Ans: A -> B -> C -> H -> G

Out of the five paths, A -> B -> C -> H -> G becomes the shortest as it has a total edge weight of 1 + 2 + 1 + 3 = 7.

Q67: At the point when B1 is dequeued and its neighbours are pushed into the priority queue, what would the state of the table of distances be?

Ans:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H | I | J |
| 0 | 1 | 3 | 31 | 31 | 9 | 31 | 31 | 31 | 5 |

**✓ Correct**

**Feedback:**

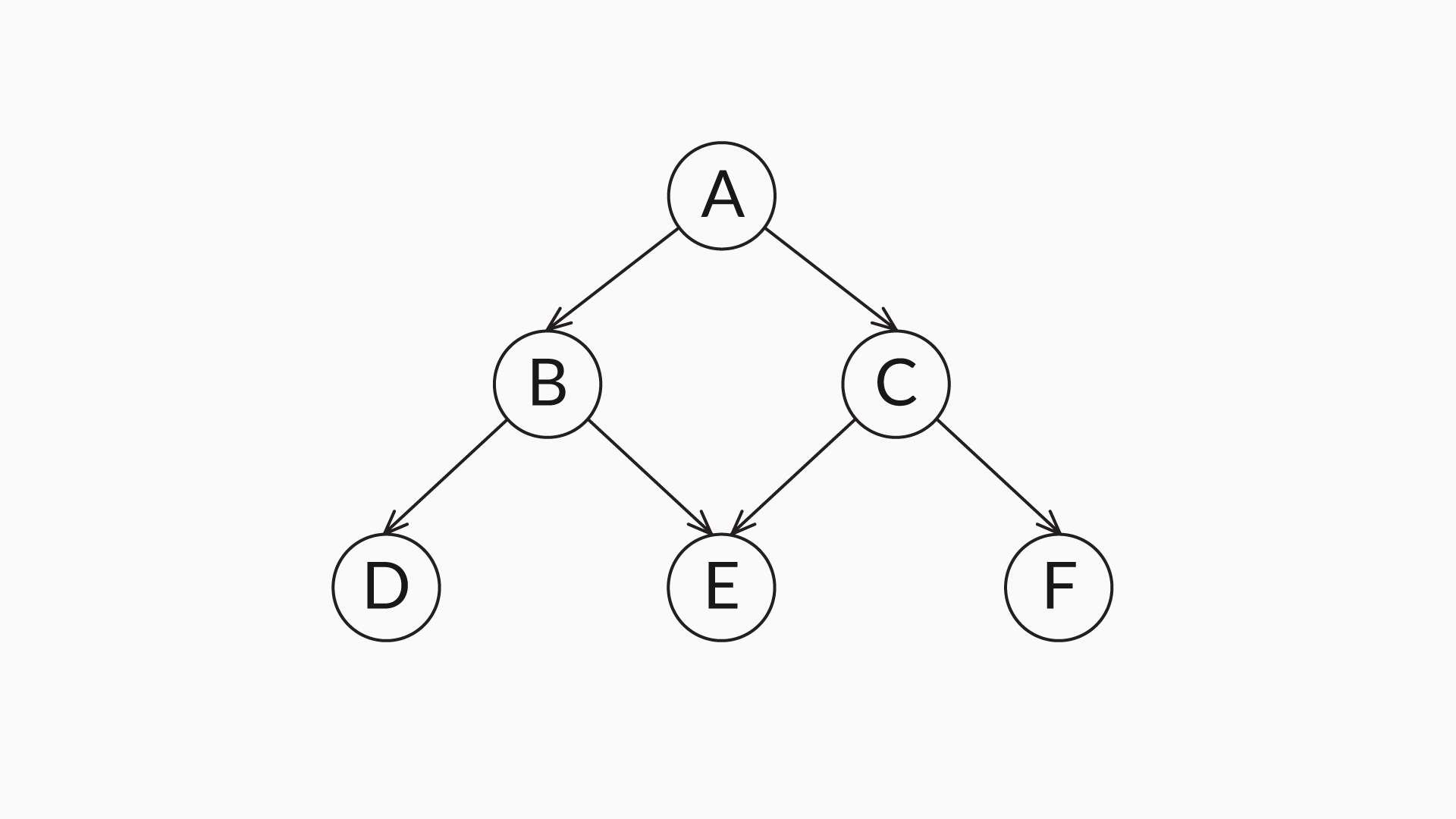
* You dequeue A from the priority queue.
* You push its neighbouring nodes into the same queue, based on the criteria for edge relaxation.
* The criteria state that if you are going from node A to node B and the present weight in the table of distances for node B is greater than the weight in the table of distances for node 1 + the edge weight between nodes A and B, then you update the table of distances for node B.
* Proceeding in the same manner, you make conditional updates on the neighbouring nodes of node A.
* Next, you have node B in front of the queue, and so, you dequeue it and make conditional updates on all of its neighbouring nodes.
* After doing all of this, the table of distances would look like this:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| A | B | C | D | E | F | G | H | I | J |
| 0 | 1 | 3 | 31 | 31 | 9 | 31 | 31 | 31 | 5 |

Graded Question:

#### Q1: Graphs

Identify the best possible category in which the following image can be included.



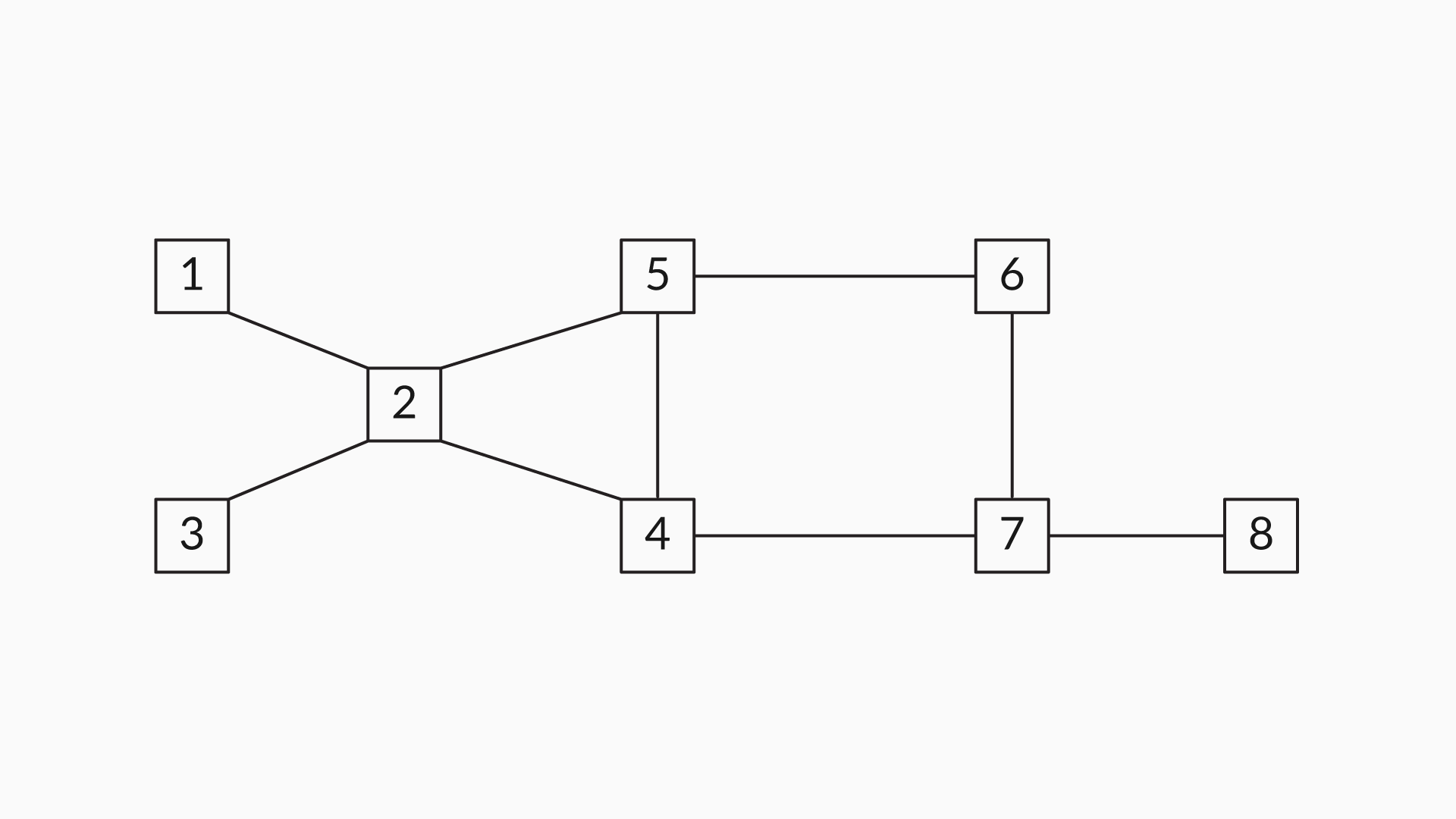
Ans: Directed acyclic graph

**✓ Correct**

**Feedback:**

In this graph, if you start from any particular node, then there is no path that takes you back to the same node. Hence, there are no cycles present in this directed graph. So, it best fits the category of directed acyclic graphs (DAGs).

#### Q2: Depth-First Search (DFS)

What is the possible order of the nodes when the graph below starts a depth-first search from node 4?

Ans: 4 2 5 6 7 8 1 3

**✓ Correct**

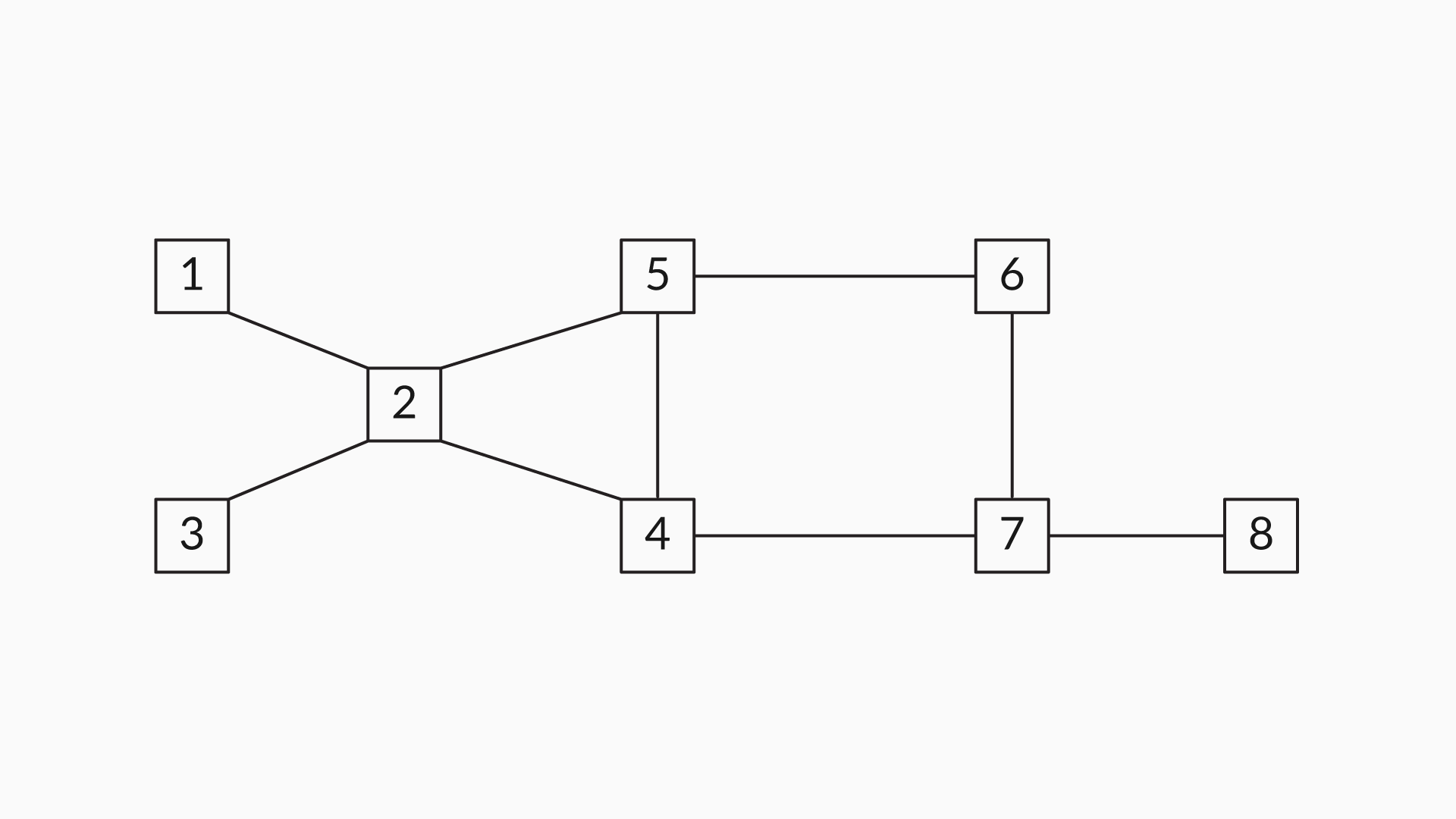
**Feedback:**

* Neighbours of node 4 are {2, 5, 7}, and, at random, the next node to be visited can be 2.
* Unvisited neighbours of node 2 are {1, 3, 5}, and, at random, the next node to be visited can be 5.
* The unvisited neighbour of node 5 is {6}, and the next node to be visited is 6.
* Node  6 consists of only one visited neighbour, node 7, and so, the next node to be visited can be 7.
* From node 7, it traverses to node 8 and it does not have any unvisited neighbours; so, it traces back to node 7.
* Again, node 7 does not have any visited neighbours, and so, it traces back to node 6, then to node 5, and then to node 2, which has two visited neighbour, nodes 1 and 3.
* Nodes 1 and 3 are visited individually and then it traces back to the start node, node 4.

This is one of the possible ways a depth-first traversal starts. It starts from node 4 and visits all the connected nodes.

#### Q3: Breadth-First Search (BFS)

What is the possible order of the nodes when the graph below starts a breadth-first search from node 4?



Ans:   
4 5 2 7 6 1 3 8

**✓ Correct**

**Feedback:**

* Neighbours of the start node 4 are {2, 5, 7} and all of these neighbours are visited at first; the possible order of the first four nodes can be 4, 5, 2 and 7.
* Since breadth-first search uses a queue data structure to store the neighbour nodes, the next nodes to be visited would be the visited neighbour node of 5, which is {6}.
* Then the unvisited neighbours of node 2, which include {1, 3}, are visited, possibly in the order 4 5 2 7 6 1 3.
* Then comes the unvisited neighbour of node 7, which is {8}.

In the order 4 5 2 7 6 1 3 8, breadth-first search visits all the nodes from the start node, node 4.

#### Q4: Graph Algorithm

Think of a undirected graph in which node A is connected to four other nodes, say, B, C, D and E, via an edge of weight 2. Which of the following statements is definitely false?

Ans: The shortest path from node C to node D is 30.

**✓ Correct**

**Feedback:**

In a graph in which node A is connected to all the other nodes via an edge of weight 2, the shortest path can never be 30, as you have the shortest path as C-A-D, which makes the shortest visible path 4. There is also a possibility of the shortest path being 2 if C and D are connected via an edge of weight 2.

#### Q5: Graph Algorithm

Think of a graph in which the shortest path from A to B is 73 and the shortest path from B to C is 38. Which of the following statements is definitely false?

Ans: The shortest path from A to C equals 146.

**✓ Correct**

**Feedback:**

In a graph, if the shortest path from A to B is 73 and the shortest path from B to C is 38, then the shortest path from A to C would never exceed 111 (the sum of 73 and 38). It can be less than 111 but not greater than that.

Q6: