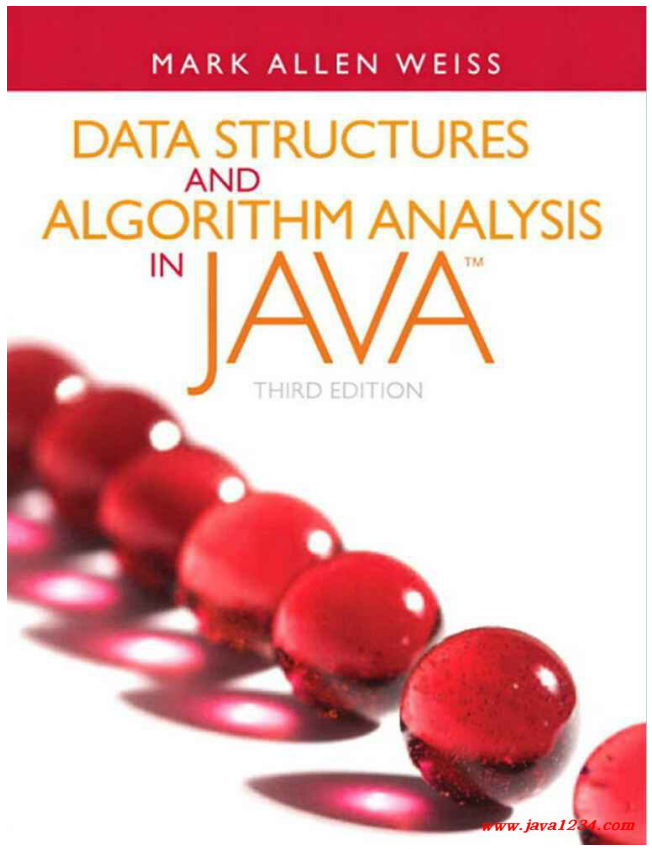
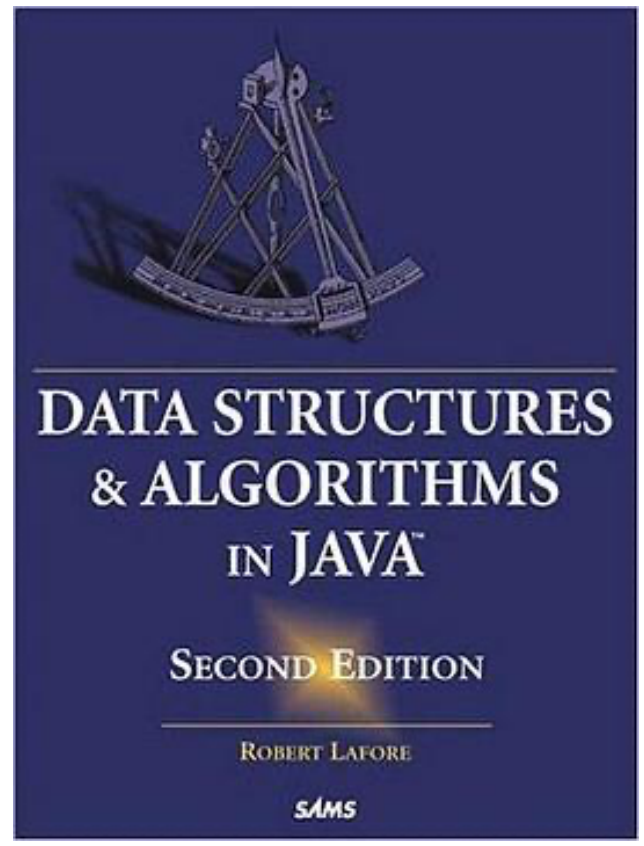
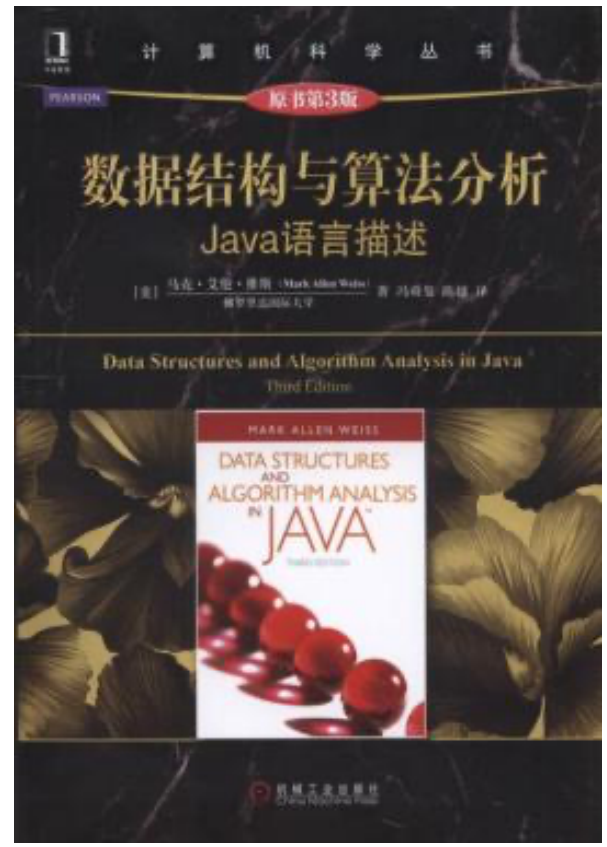
Topic 16 –
Graph Data Structure







- **Graph**
- Graph Data Structure
- Depth First Search
- Breadth First Search
- Topological Sorting

Graph

BFS DFS Topological

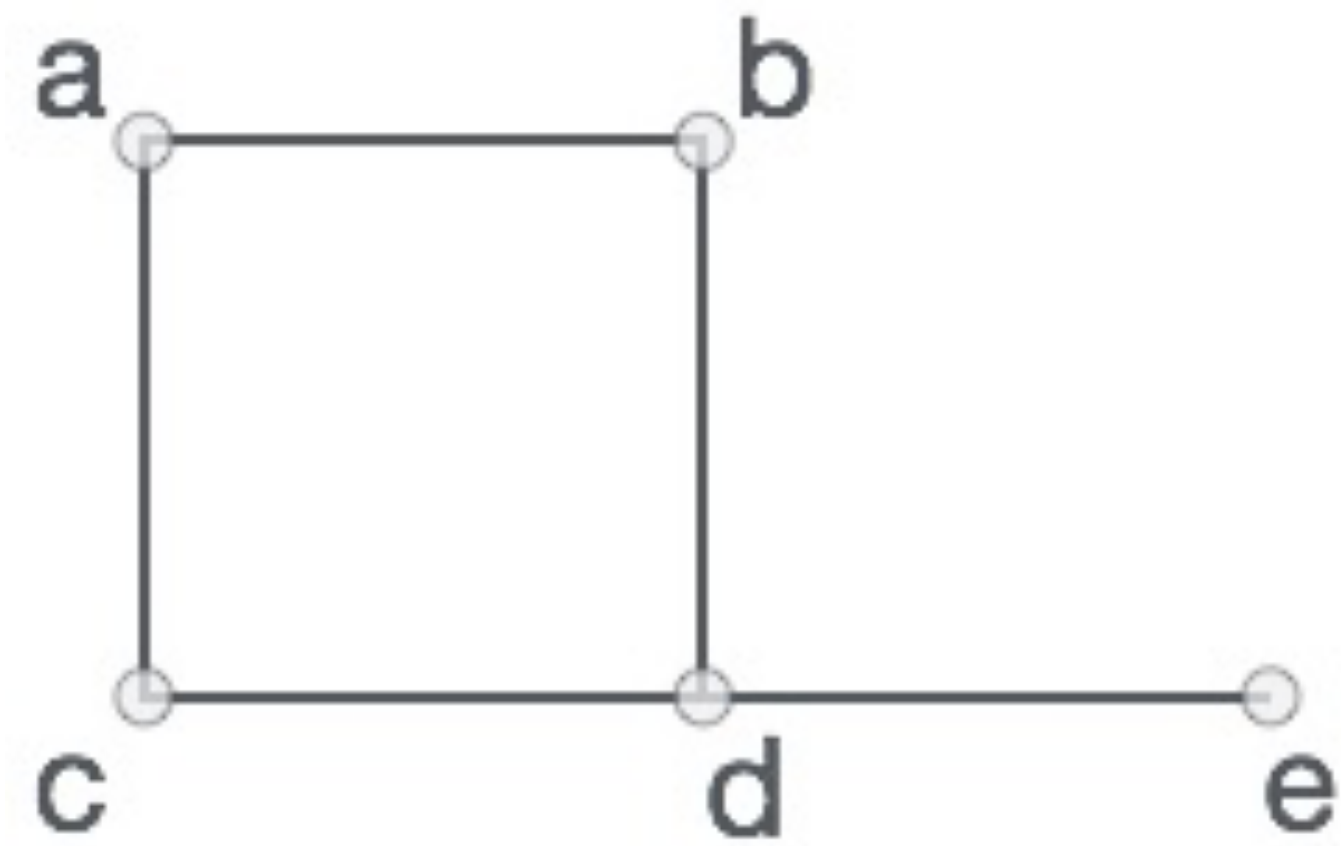
- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

顶点+边→图,

- Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −

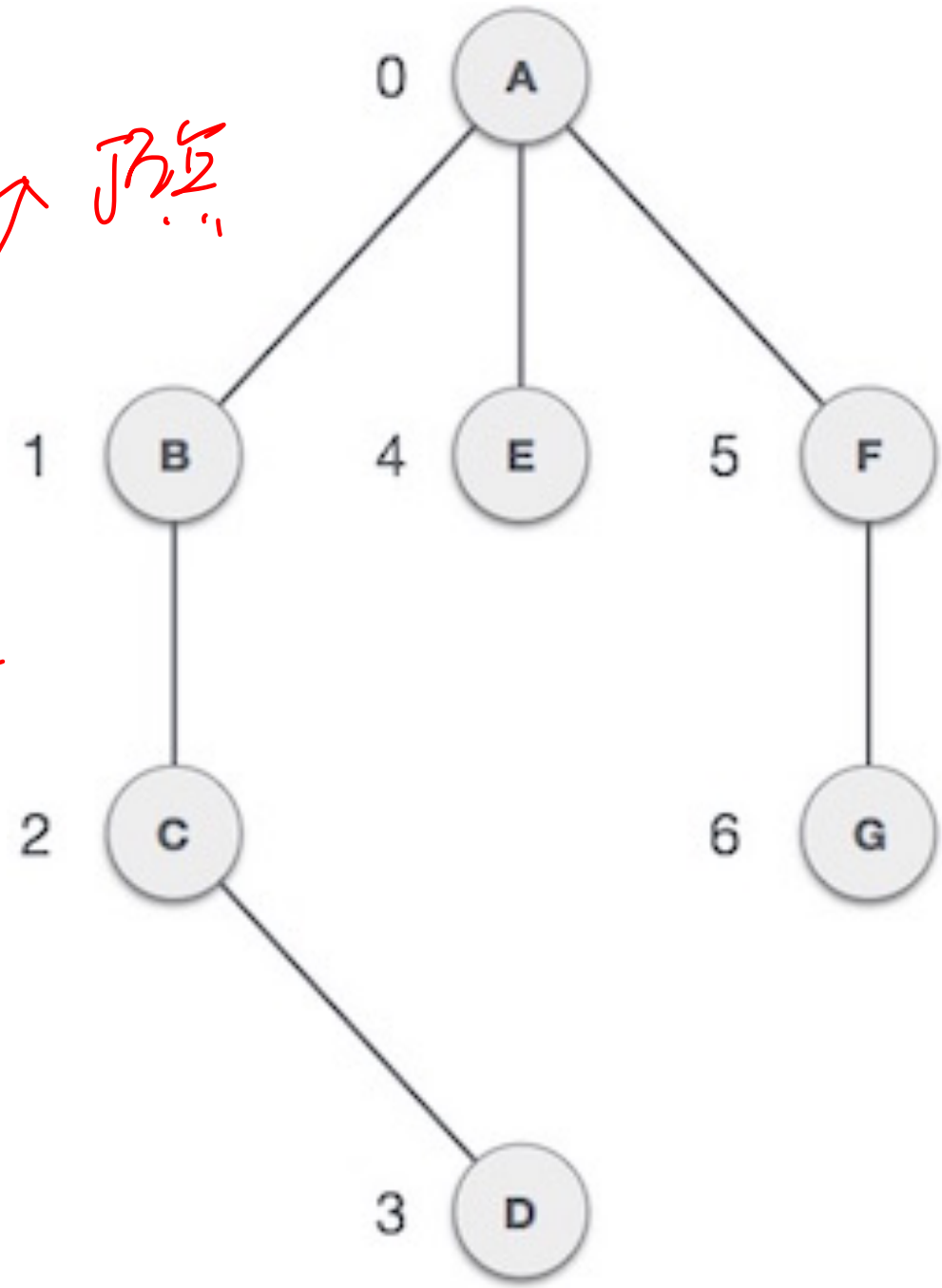
Graph

- In this graph,
  - V = {a, b, c, d, e}
  - E = {ab, ac, bd, cd, de}



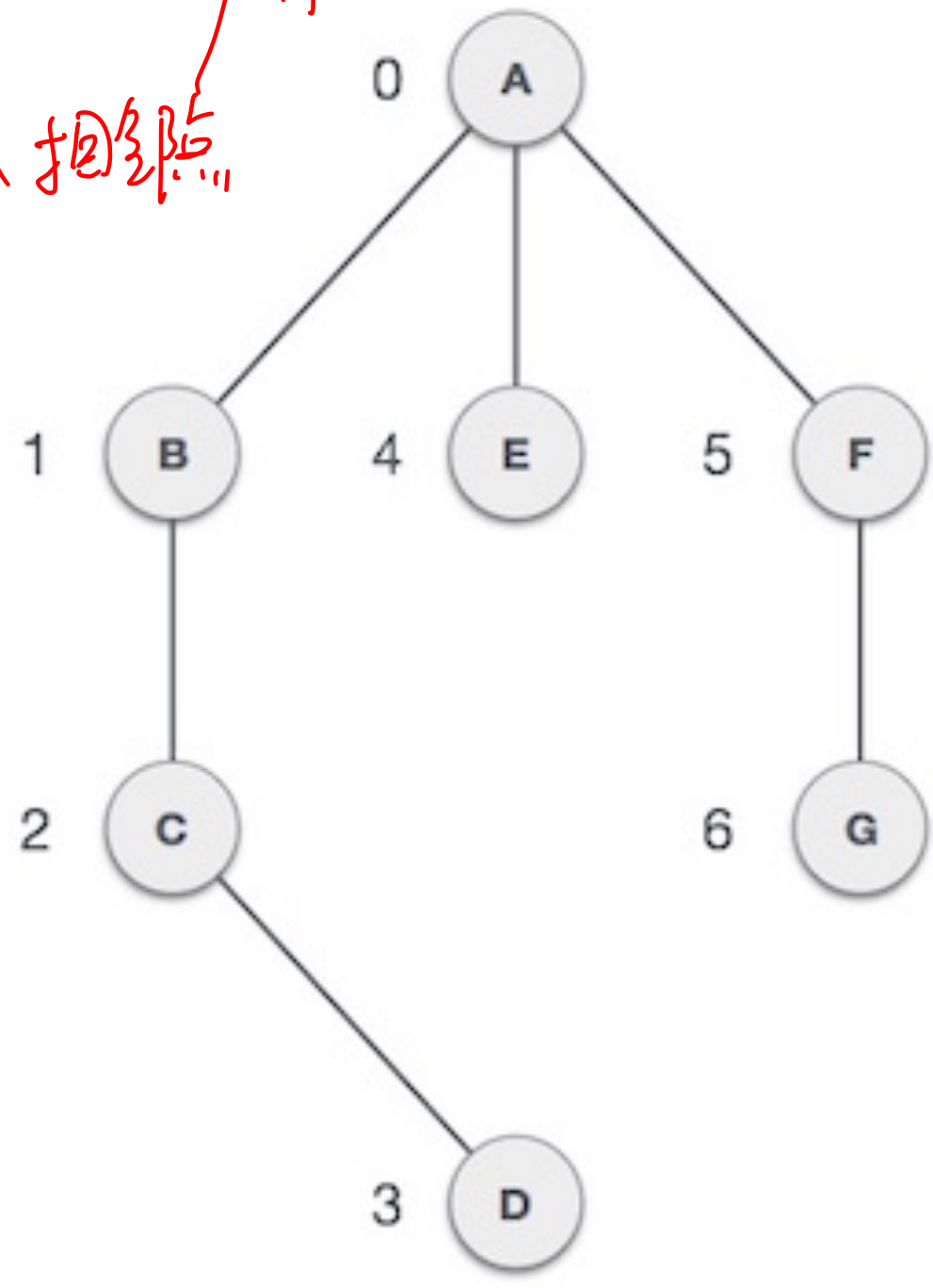
Graph

- **Vertex** − Each node of the graph is represented as a vertex.
  - A to G are vertices.
  - A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices.
  - We can use a two-dimensional array to represent edge set.
  - Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.



Graph

- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge.
  - B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices.
  - ABCD represents a path from A to D.



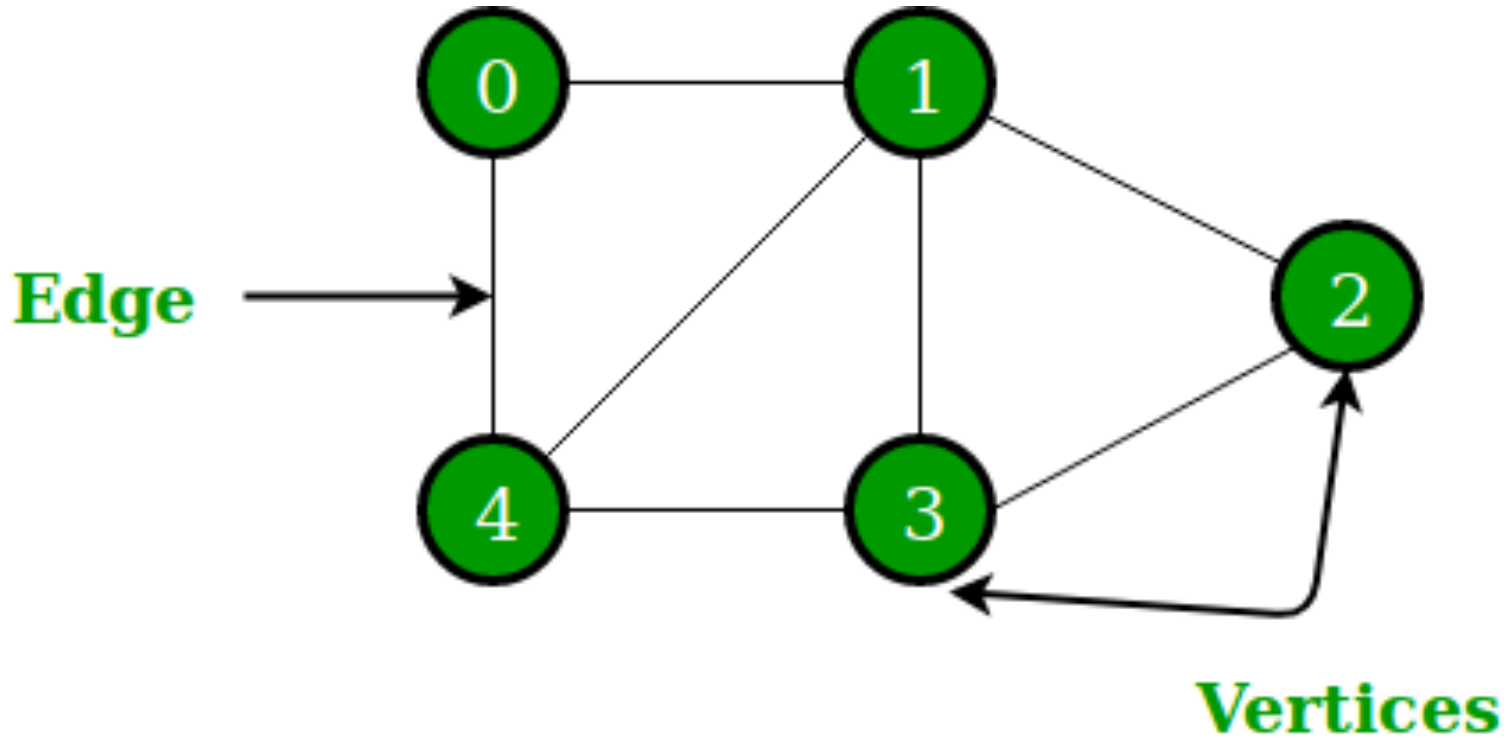
Basic Operations

- Following are basic primary operations of a Graph −
  - **Add Vertex** − Adds a vertex to the graph.
  - **Add Edge** − Adds an edge between the two vertices of the graph.
  - **Display Vertex** − Displays a vertex of the graph.

- Graph
- **Graph Data Structure**
- Depth First Search
- Breadth First Search
- Topological Sorting

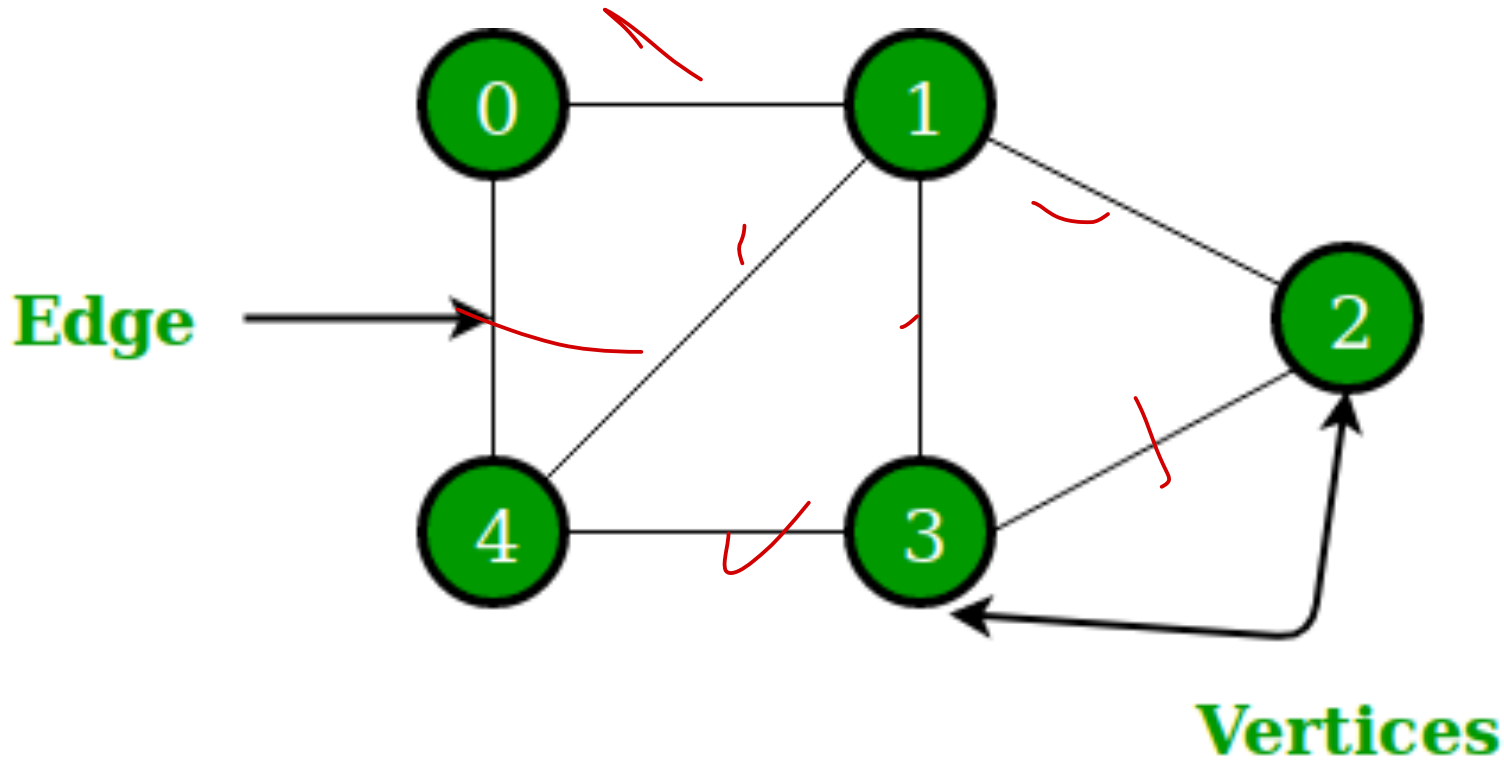
Graph Data Structure

- A Graph is a non-linear data structure consisting of nodes and edges.
- The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- More formally a Graph can be defined as,
    - A Graph consists of a finite set of vertices(or nodes) and set of Edges which connect a pair of nodes.



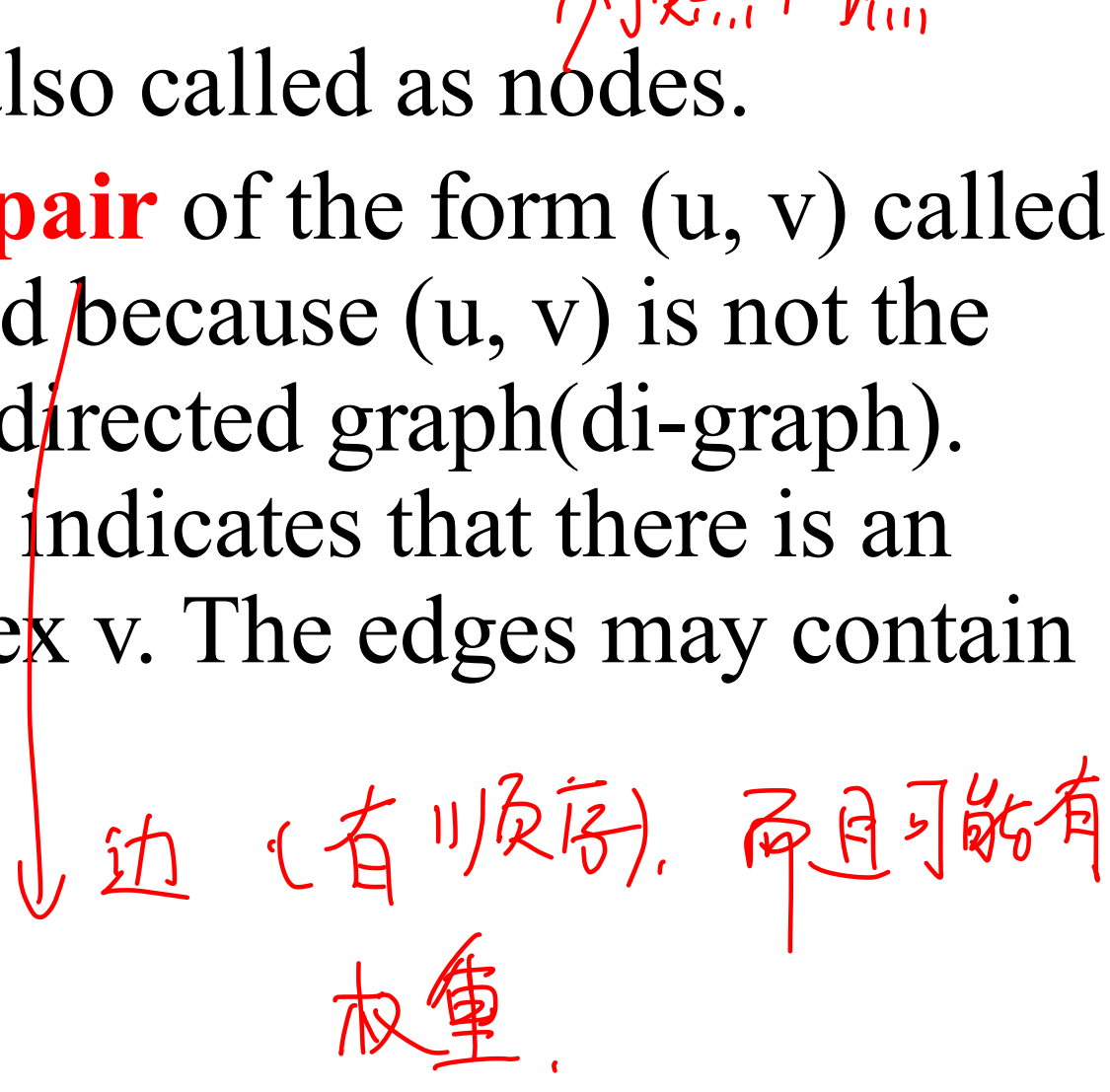
Example

- In this Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.



Graph and its representations

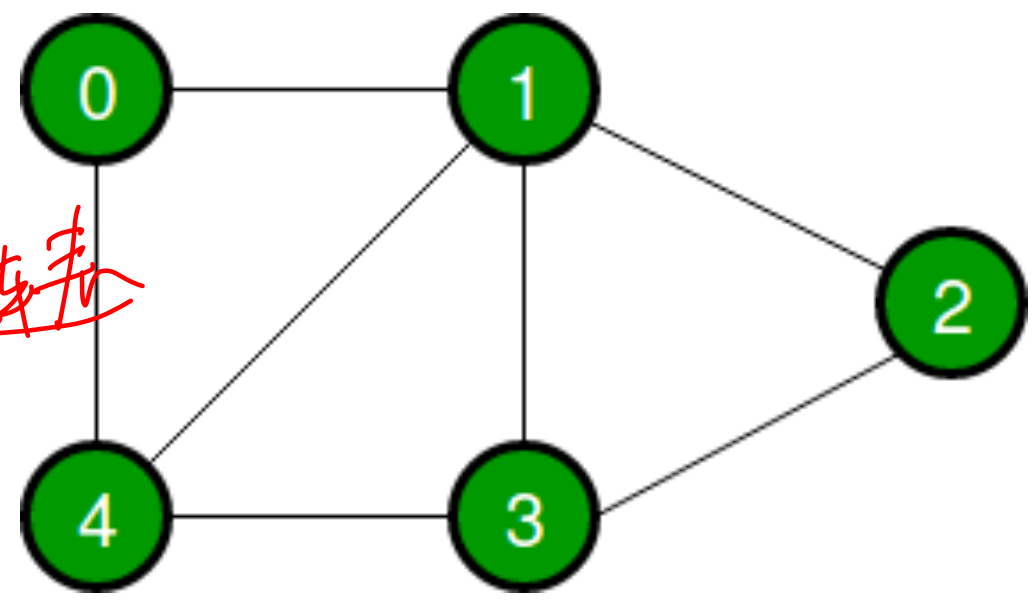
- A graph is a data structure that consists of the following two components:
  - **1.** A finite set of vertices also called as nodes.
  - **2.** A finite set of **ordered pair** of the form (u, v) called as edge. The pair is ordered because (u, v) is not the same as (v, u) in case of a directed graph(di-graph). The pair of the form (u, v) indicates that there is an edge from vertex u to vertex v. The edges may contain weight/value/cost.



Graph and its representations

- A graph is a data structure that consists of the following two components:
  - **1.** A finite set of vertices also called as nodes.
  - **2.** A finite set of **ordered pair** of the form (u, v) called as edge.

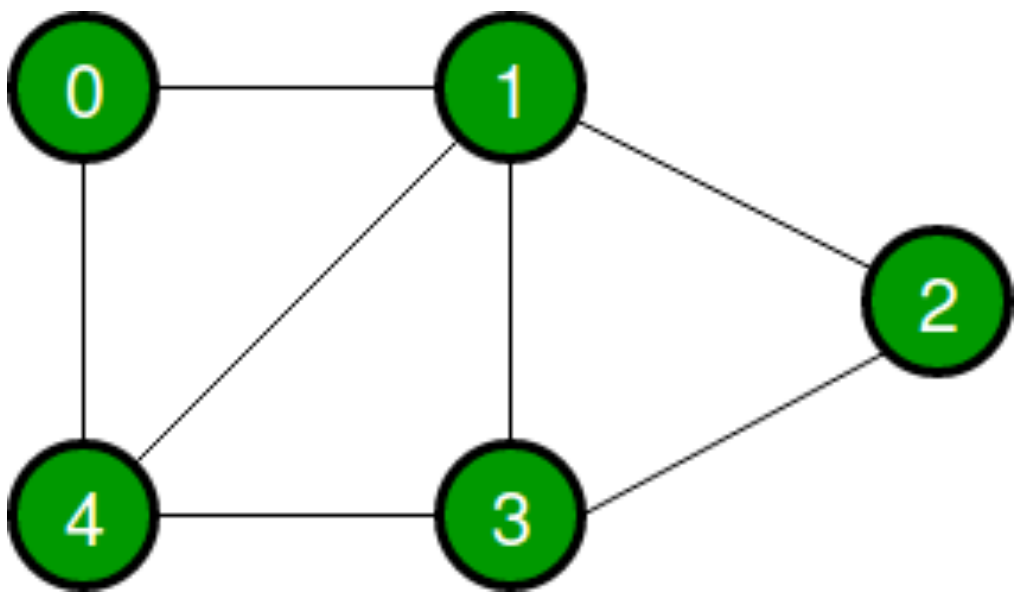
- Following is an example of an undirected graph with 5 vertices.
  1. Adjacency Matrix
  2. Adjacency List



Graph and its representations

- **Adjacency Matrix:**
  - It is a |V| x |V| array.
  - Let this 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j.
    - Adjacency matrix for undirected graph is always symmetric.
    - Adjacency Matrix is also used to represent weighted graphs.

If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



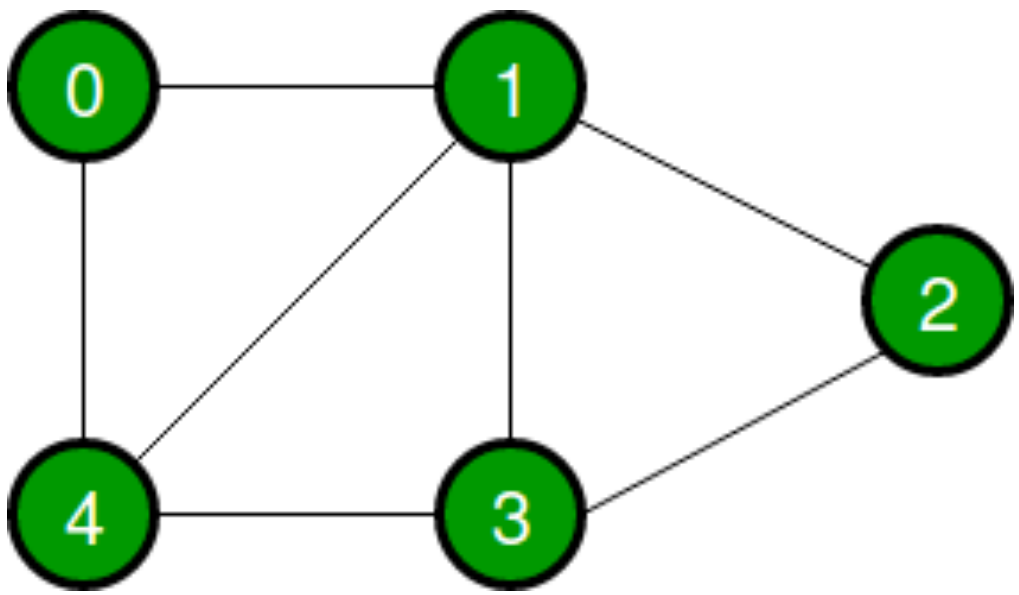
邻接矩阵.

Adjacency Matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Graph and its representations

- **Adjacency Matrix:**
  - *Pros:* Representation is easier to implement and follow. Removing an edge takes O(1) time. Queries like whether there is an edge from vertex ‘u’ to vertex ‘v’ are efficient and can be done O(1).
  - *Cons:* Consumes more space O(V^2). Even if the graph is sparse(contains less number of edges), it consumes the same space. Adding a vertex is O(V^2) time.

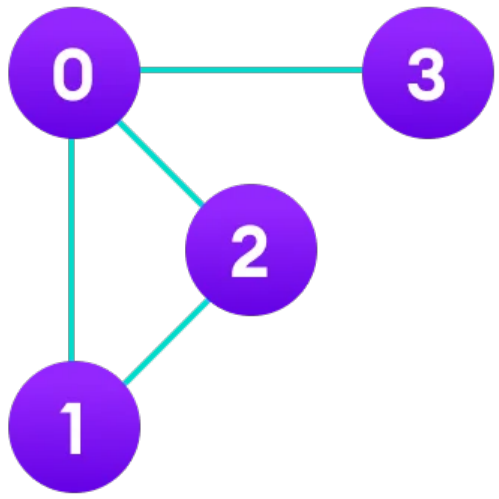


Adjacency Matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

Adjacency matrix representation

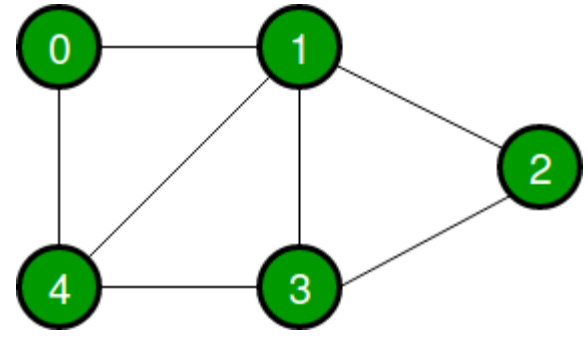
- An adjacency matrix is a way of representing a graph G = {V, E} as a matrix of booleans.
- The size of the matrix is VxV where V is the number of vertices in the graph and the value of an entry A_{ij} is either 1 or 0 depending on whether there is an edge from vertex i to vertex j.



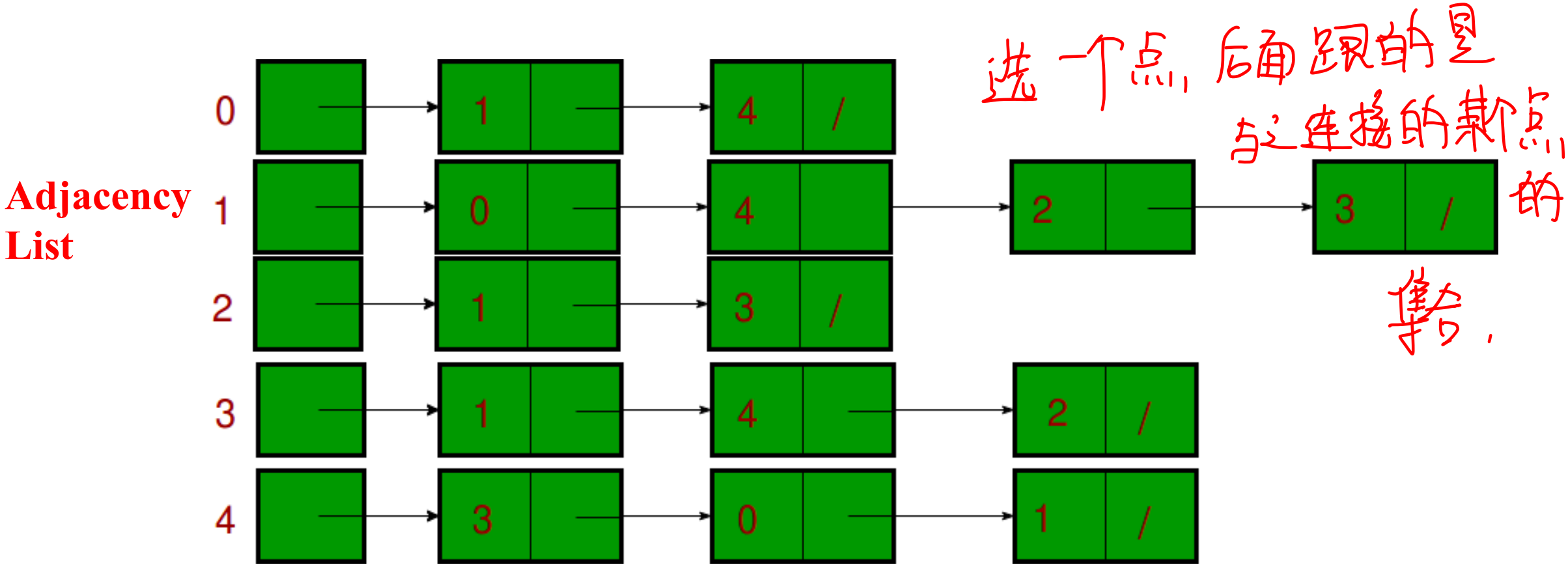
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	0
3	1	0	0	0

In case of undirected graphs, the matrix is symmetric about the diagonal because of every edge (i,j), there is also an edge (j,i).

Graph and its representations

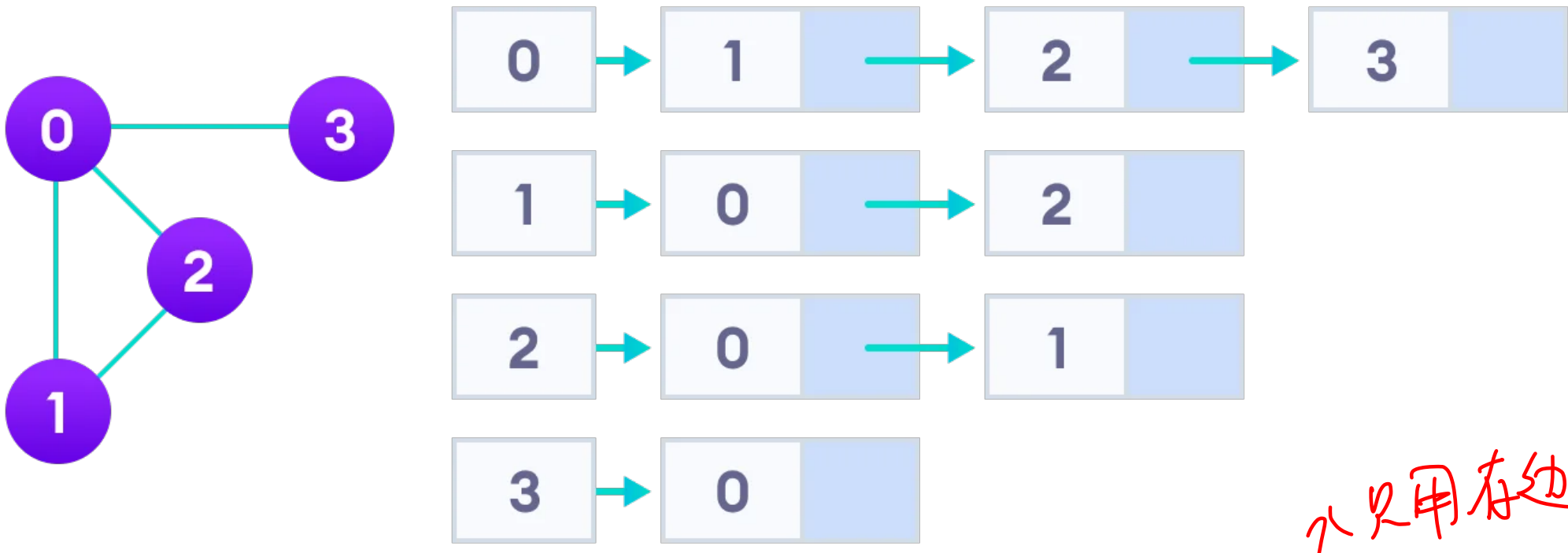


- **Adjacency List:**
  - An array of lists is used.
  - Let the array be an array[]. An entry array[i] represents the list of vertices adjacent to the ***i***-th vertex.
  - This representation can also be used to represent a weighted graph. The weights of edges can be represented as lists of pairs.



Adjacency List representation

- An adjacency list represents a graph as **an array of linked lists**.



- An adjacency list is efficient in terms of storage because we only need to store the values for the edges.
- For a sparse graph with millions of vertices and edges, this can mean a lot of saved space.

Adjacency List Java

- The type of LinkedList is determined by what data you want to store in it.
- For a labeled graph, you could store a dictionary instead of an Integer.

```
class Graph {
    private int numVertices;
    private LinkedList<integer> adjLists[];
}
```

ArrayList

```
// Driver Code
	public static void main(String[] args)
	{
		// Creating a graph with 5 vertices
		int V = 5;
		ArrayList<ArrayList<Integer> > adj = new ArrayList<ArrayList<Integer> >(V);

		for (int i = 0; i < V; i++)
			adj.add(new ArrayList<Integer>());

		// Adding edges one by one
		addEdge(adj, 0, 1);
		addEdge(adj, 0, 4);
		addEdge(adj, 1, 2);
		addEdge(adj, 1, 3);
		addEdge(adj, 1, 4);
		addEdge(adj, 2, 3);
		addEdge(adj, 3, 4);

		printGraph(adj);
}
```

ArrayList

```
// A utility function to print the adjacency list representation of graph
    static void printGraph(ArrayList<ArrayList<Integer> > adj)
    {
        for (int i = 0; i < adj.size(); i++) {
            System.out.println("\nAdjacency list of vertex" + i);
            System.out.print("head");
            for (int j = 0; j < adj.get(i).size(); j++) {
                System.out.print(" -> "+adj.get(i).get(j));
            }
            System.out.println();
        }
}
```

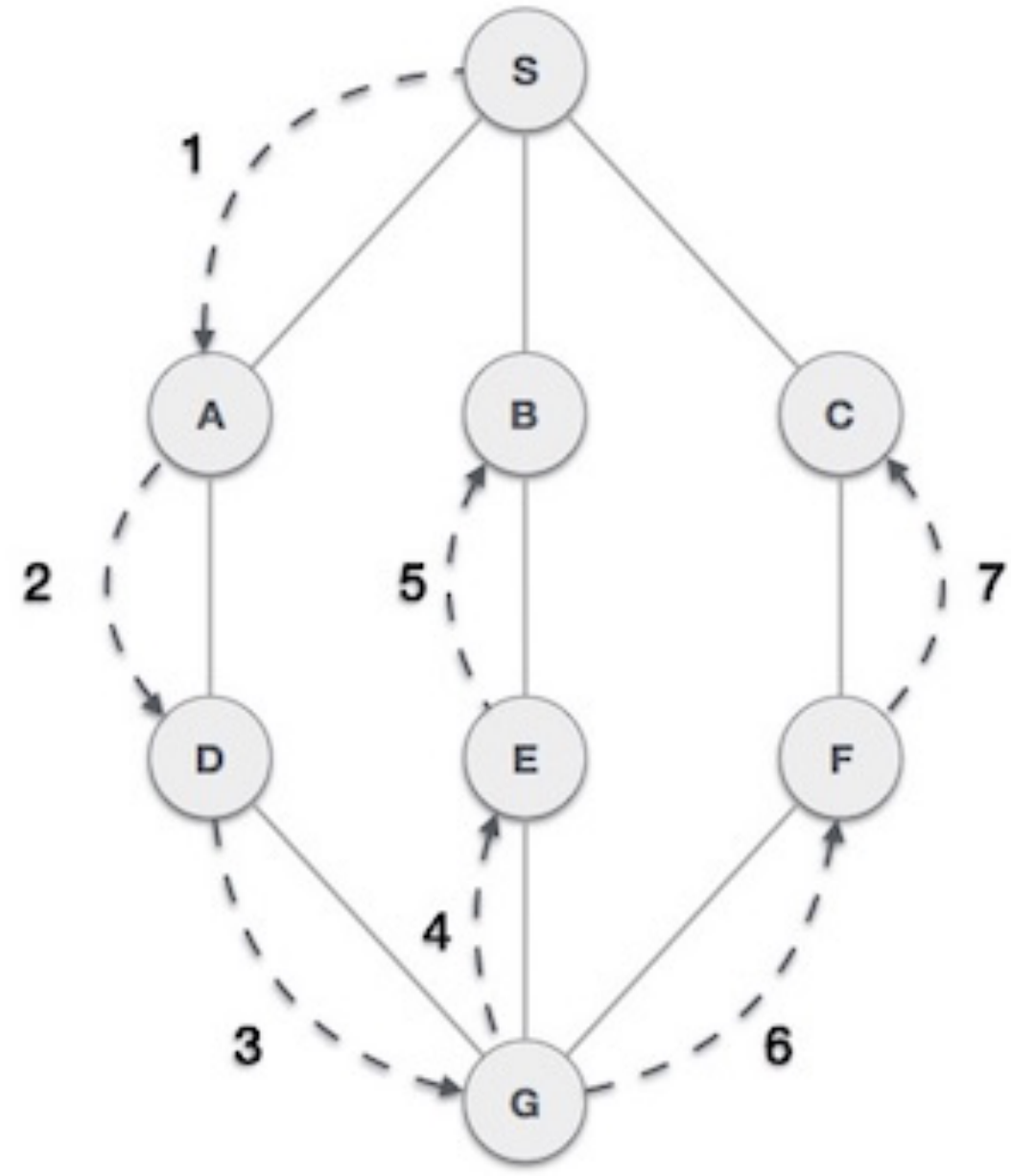
ArrayList

```
// Java code to demonstrate Graph representation using ArrayList in Java
import java.util.*;

class Graph {
	// A utility function to add an edge in an undirected graph
	static void addEdge(ArrayList<ArrayList<Integer> > adj, int u, int v)
	{
		adj.get(u).add(v);
		adj.get(v).add(u);
	}
	// A utility function to print the adjacency list representation of graph
	static void printGraph(ArrayList<ArrayList<Integer> > adj)
	{
		// See Pre-slide
	}

	// Driver Code
	public static void main(String[] args)
	{
		// See Pre-slide
	}
}
```

- Graph
- Graph Data Structure
- **Depth First Search**
- Breadth First Search
- Topological Sorting

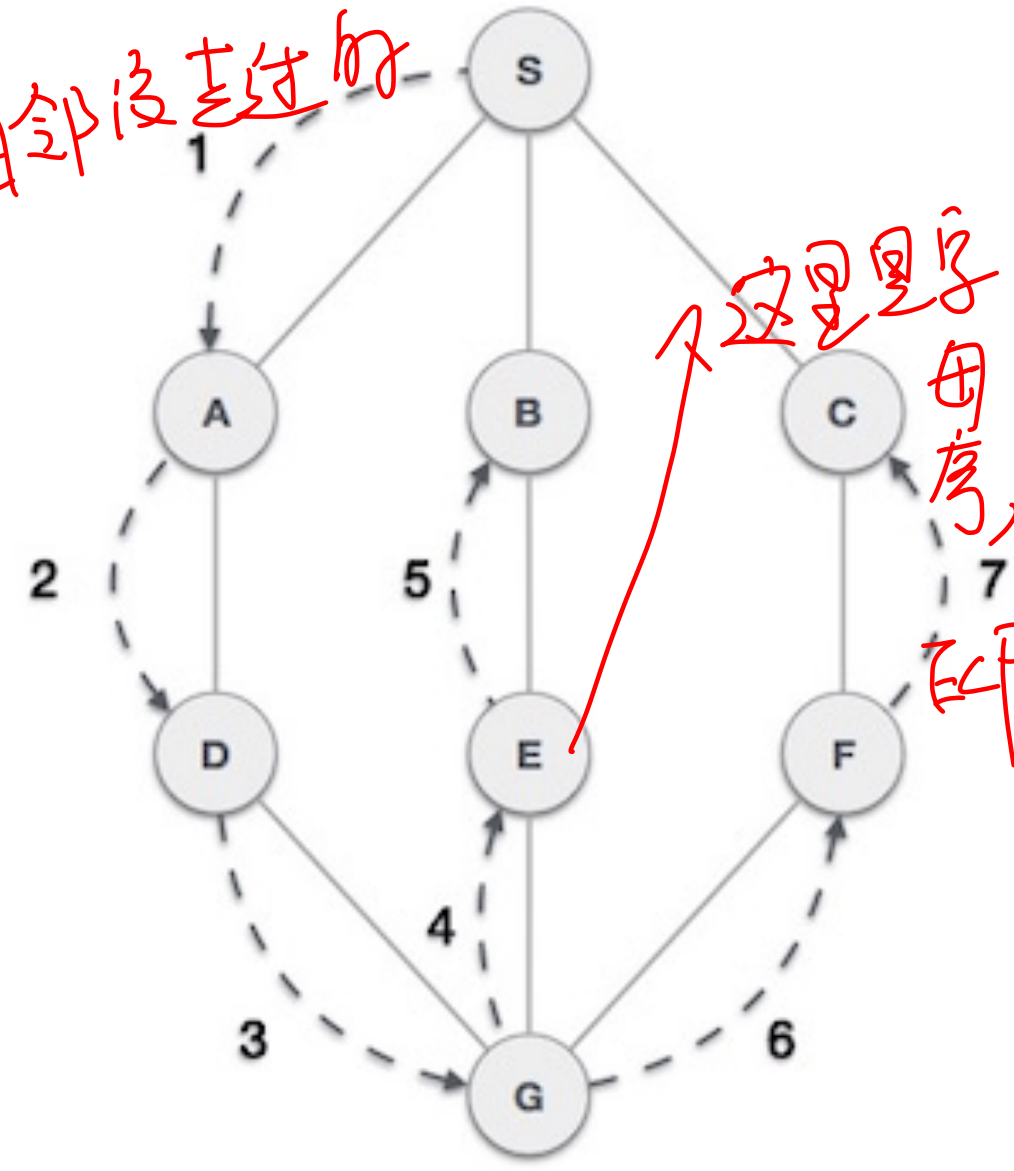
Depth First Search

- Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a **stack** to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

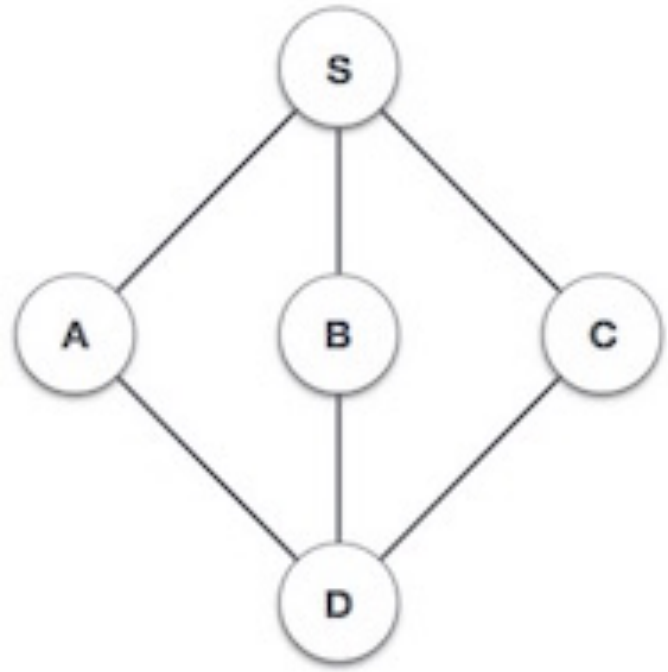


Depth First Search

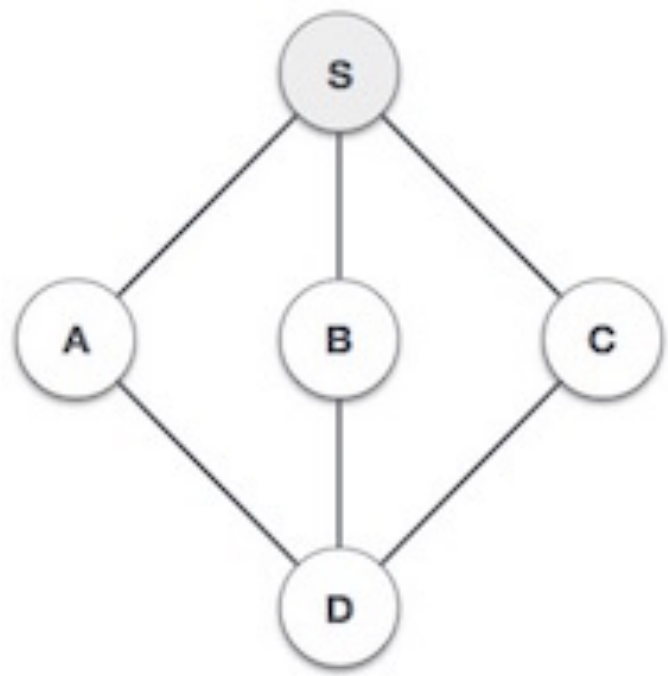
- As in the example, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C.
- It employs the following rules.
  - **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
  - **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
  - **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

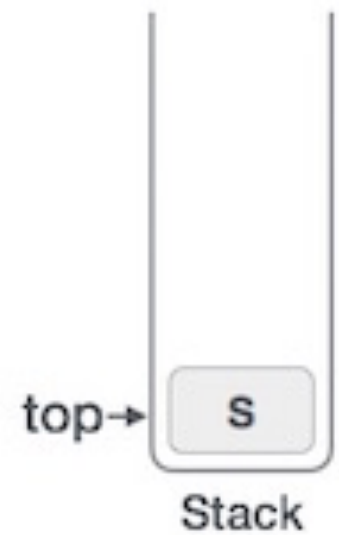


Example of DF



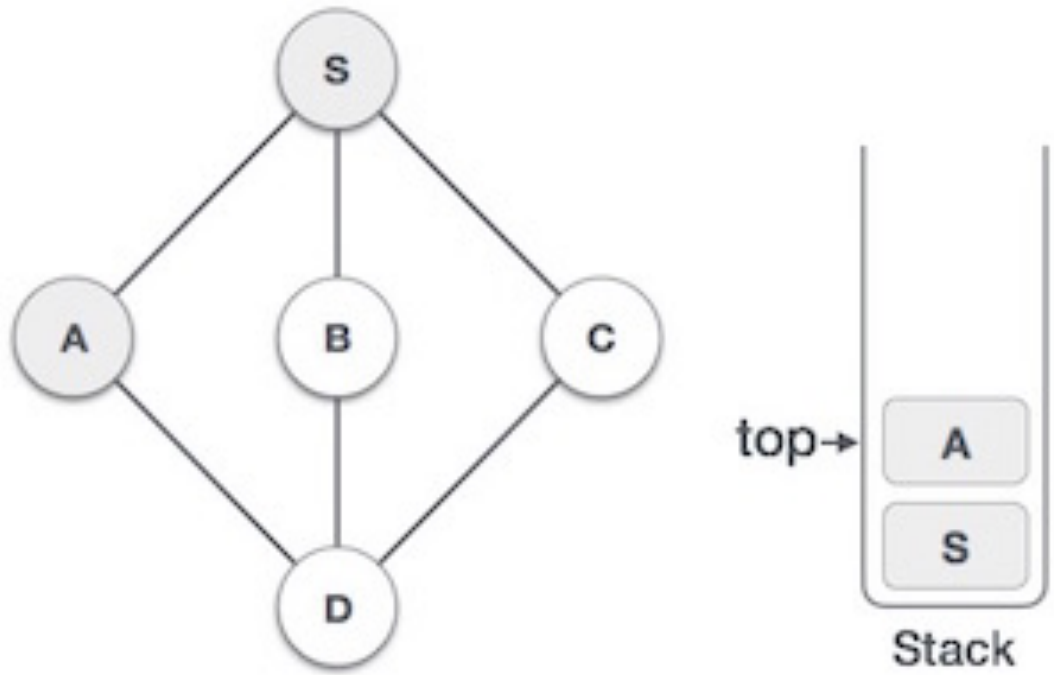
- **Step 0.**
  - initialize the stack.
- **Step 1.**
  - Mark **S** as visited and put it onto the stack.
  - Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them.
  - For this example, we shall take the node in an *alphabetical order*.





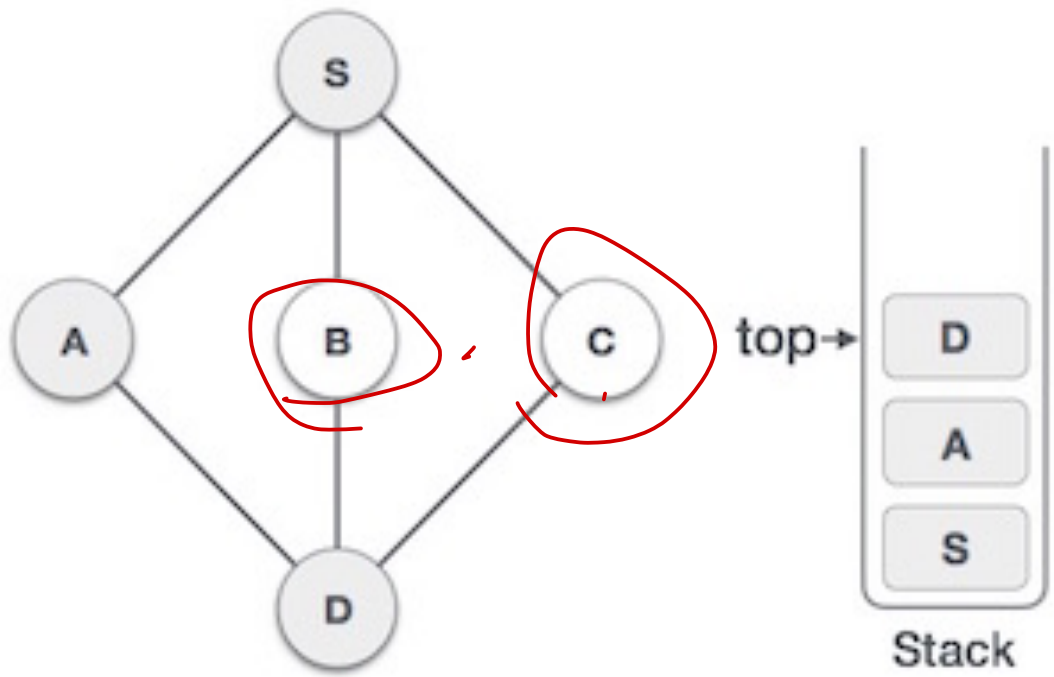
Example of DFS

- **Step 2.**
  - Mark **A** as visited and put it onto the stack.
  - Explore any unvisited adjacent node from A.
  - Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.



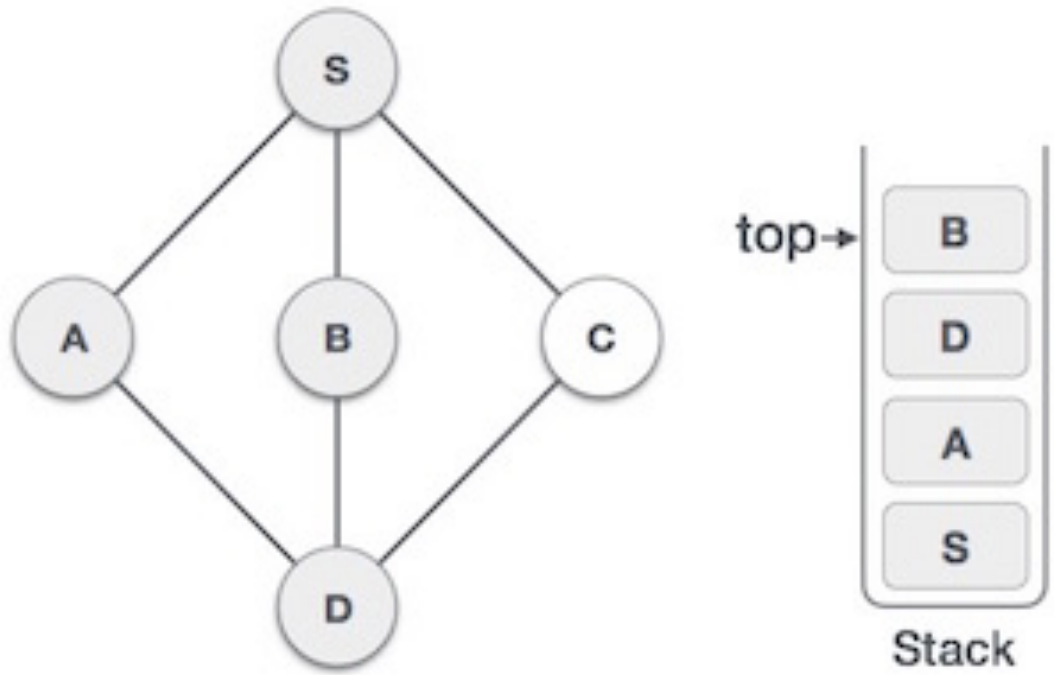
Example of DFS

- **Step 3.**
  - Visit **D** and mark it as visited and put onto the stack.
  - Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited.
  - However, we shall again choose in an alphabetical order.



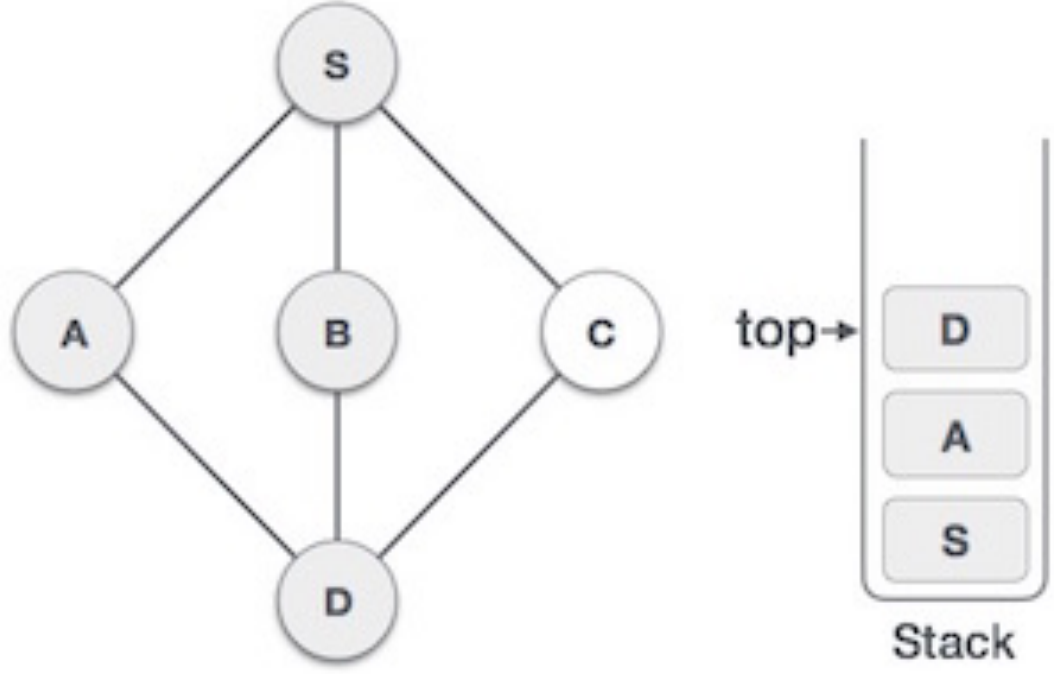
Example of DFS

- **Step 4.**
  - We choose **B**, mark it as visited and put onto the stack.
  - Here **B** does not have any unvisited adjacent node.
  - So, we pop **B** from the stack.



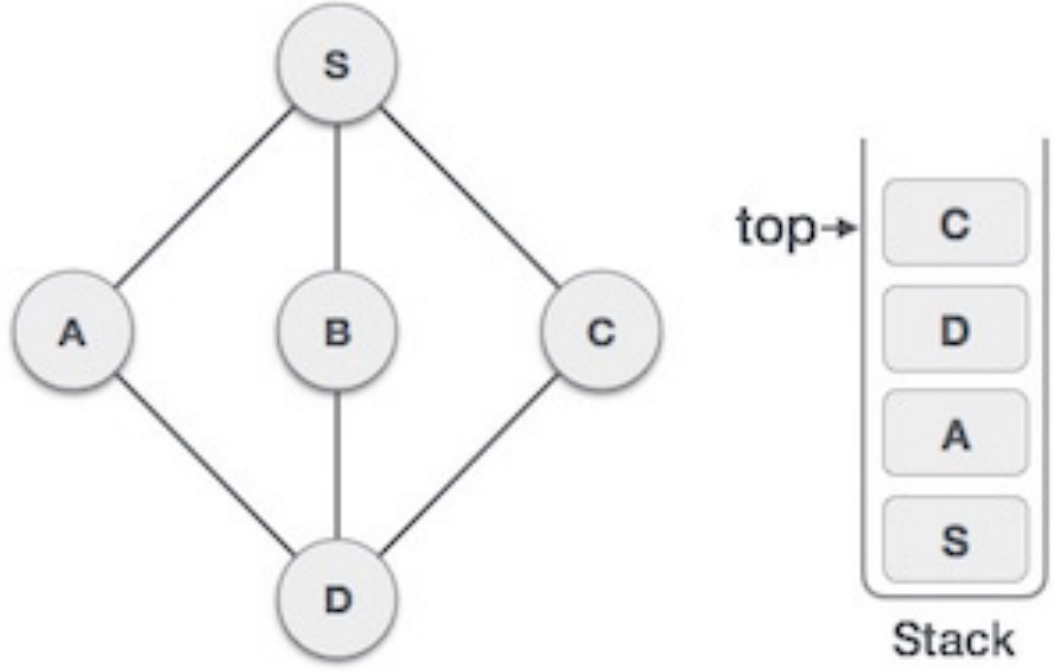
Example of DFS

- **Step 5.**
  - We check the stack top for return to the previous node and check if it has any unvisited nodes.
  - Here, we find **D** to be on the top of the stack.



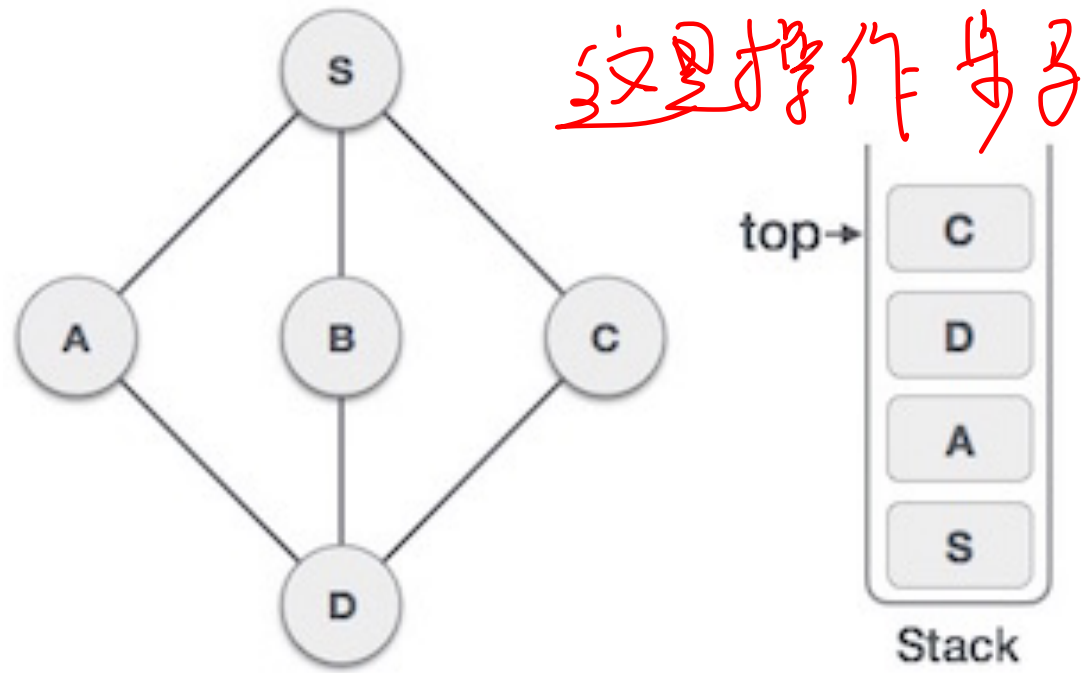
Example of DFS

- **Step 6.**
  - Only unvisited adjacent node is from **D** is **C** now.
  - So we visit **C**, mark it as visited and put it onto the stack.



Example of DFS

- **Step 7:**
  - As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node.
  - In this case, there's none and we keep **popping** until the stack is empty.



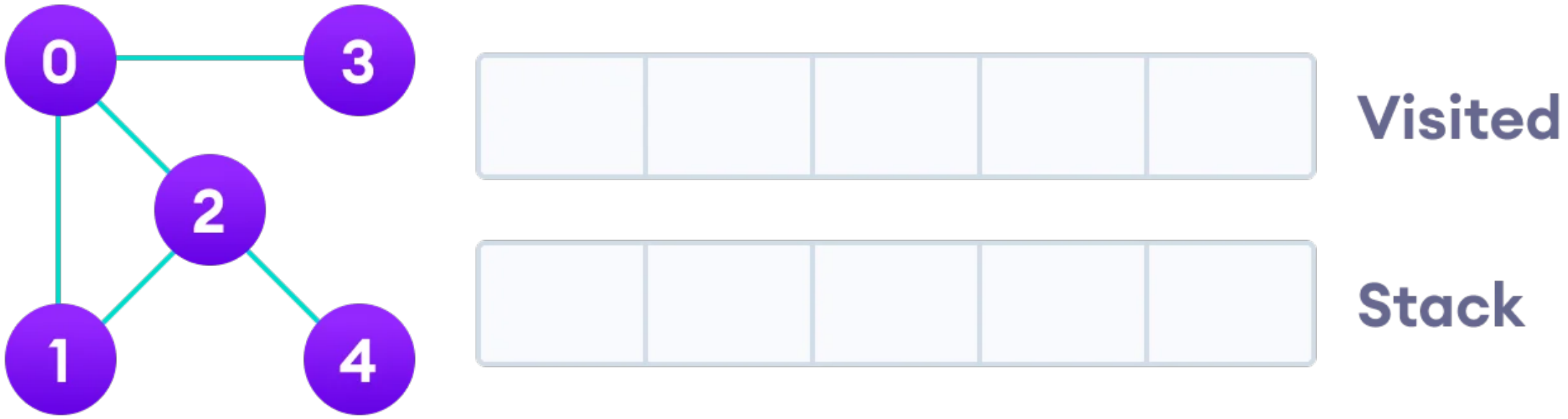
- **Output:**
  - **B, C, D, A, S**
  - Note. We had pop **B** at Step 4.

Depth First Search Algorithm

- A standard DFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The DFS algorithm works as follows:
  1. Start by putting any one of the graph's vertices on top of a stack.
  2. Take the top item of the stack and add it to the visited list.
  3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
  4. Keep repeating steps 2 and 3 until the stack is empty.

Depth First Search Example

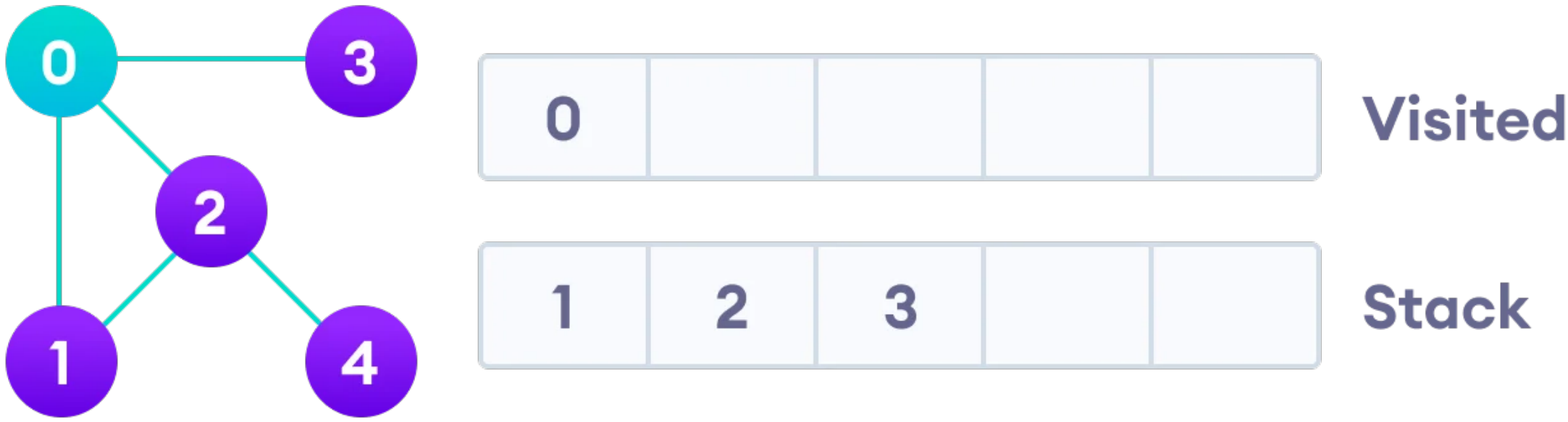
- Let's see how the Depth First Search algorithm works with an example.
- We use an undirected graph with 5 vertices.



Undirected graph with 5 vertices

Depth First Search Example

- We start from vertex 0, the DFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the **stack**.



Visit the element and put it in the visited list

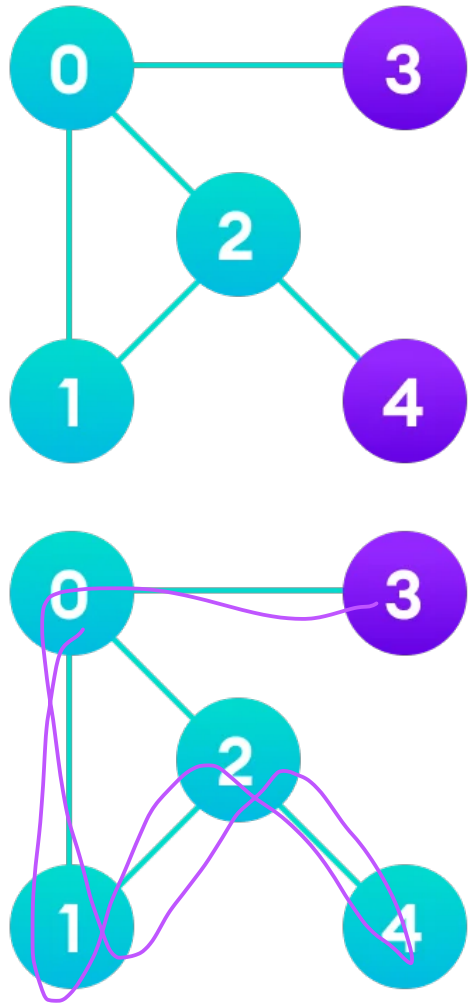
Depth First Search Example

- Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes.
- Since 0 has already been visited, we visit 2 instead.

Visit the element at the top of stack

Depth First Search Example

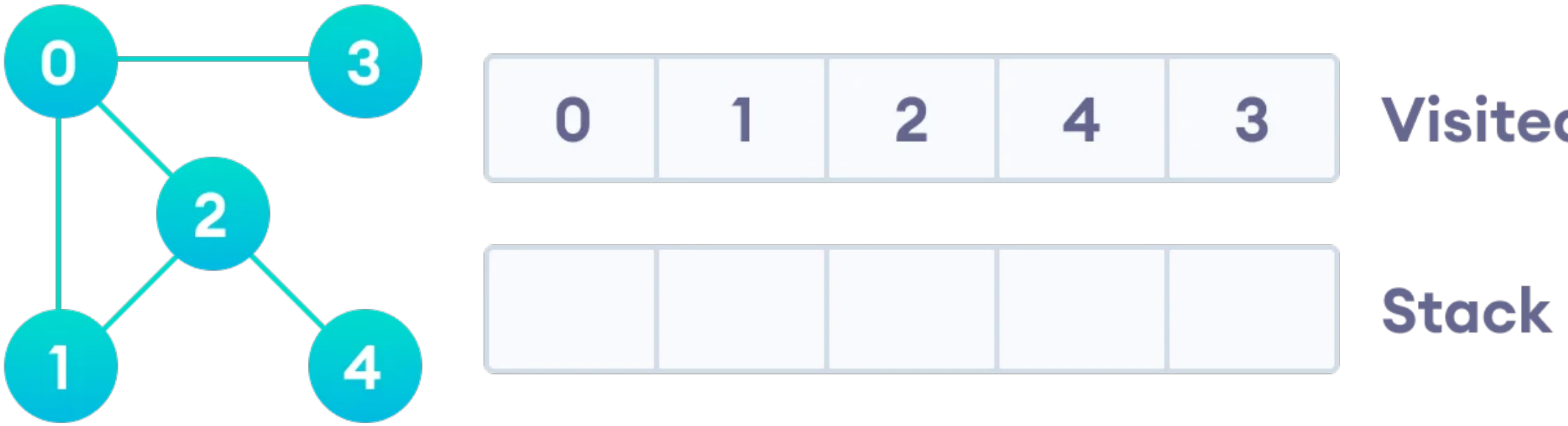
- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it.



Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the top of the stack and visit it

Depth First Search Example

- After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.



After we visit the last element 3, it doesn't have any unvisited adjacent nodes, so we have completed the Depth First Traversal of the graph.

DFS Pseudocode (recursive implementation)

- The pseudocode for DFS is shown below. In the init() function, notice that we run the DFS function on every node.
- This is because the graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the DFS algorithm on every node.

```
DFS(G, u)
        u.visited = true
        for each v ∈   G.Adj[u]
                if v.visited == false
                        DFS(G,v)

init() {
        For each u ∈   G
                u.visited = false
        For each u ∈   G
                DFS(G, u)
}
```

DFS Implementation in Java

```
// DFS algorithm in Java
import java.util.*;

class Graph {
      private LinkedList<Integer> adjLists[];
      private boolean visited[];

      // Graph creation
      Graph(int vertices) {
            adjLists = new LinkedList[vertices];
            visited = new boolean[vertices];
            for (int i = 0; i < vertices; i++)
                  adjLists[i] = new LinkedList<Integer>();
      }
```

DFS Implementation in Java

```
// Add edges
void addEdge(int src, int dest) {
    adjLists[src].add(dest);
}
```

DFS Implementation in Java

```java
// DFS algorithm
void DFS(int vertex) {
    visited[vertex] = true;
    System.out.print(vertex + " ");
    Iterator<Integer> ite = adjLists[vertex].listIterator();
    while (ite.hasNext()) {
        int adj = ite.next();
        if (!visited[adj])
            DFS(adj);
    }
}
```

DFS Implementation in Java

```
    public static void main(String args[]) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 3);
        System.out.println("Following is Depth First Traversal");
        g.DFS(2);
    }
}
```

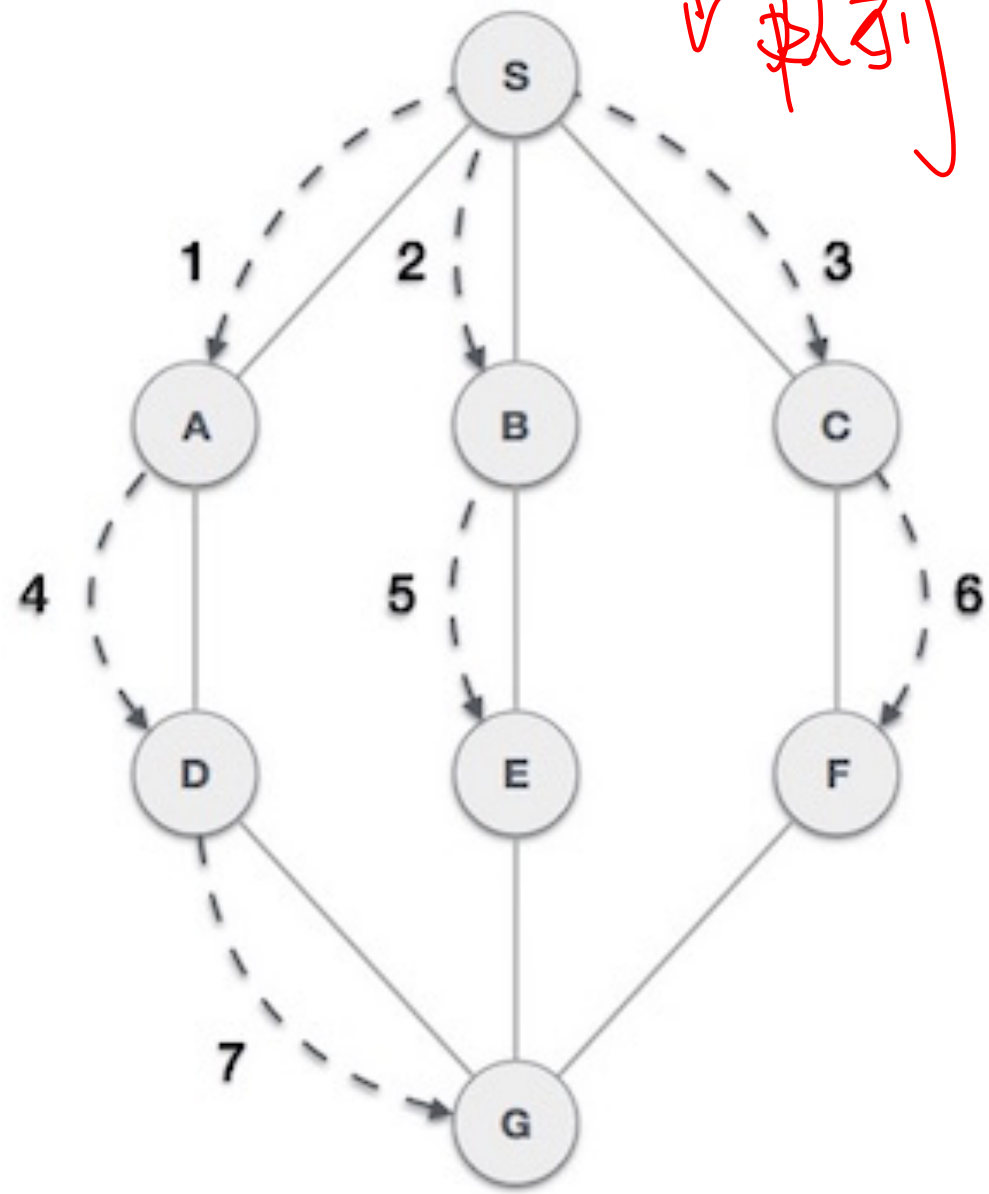
Complexity of Depth First Search

- The time complexity of the DFS algorithm is represented in the form of **O(V + E)**, where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is **O(V)**.

- Graph
- Graph Data Structure
- Depth First Search
- **Breadth First Search**
- Topological Sorting

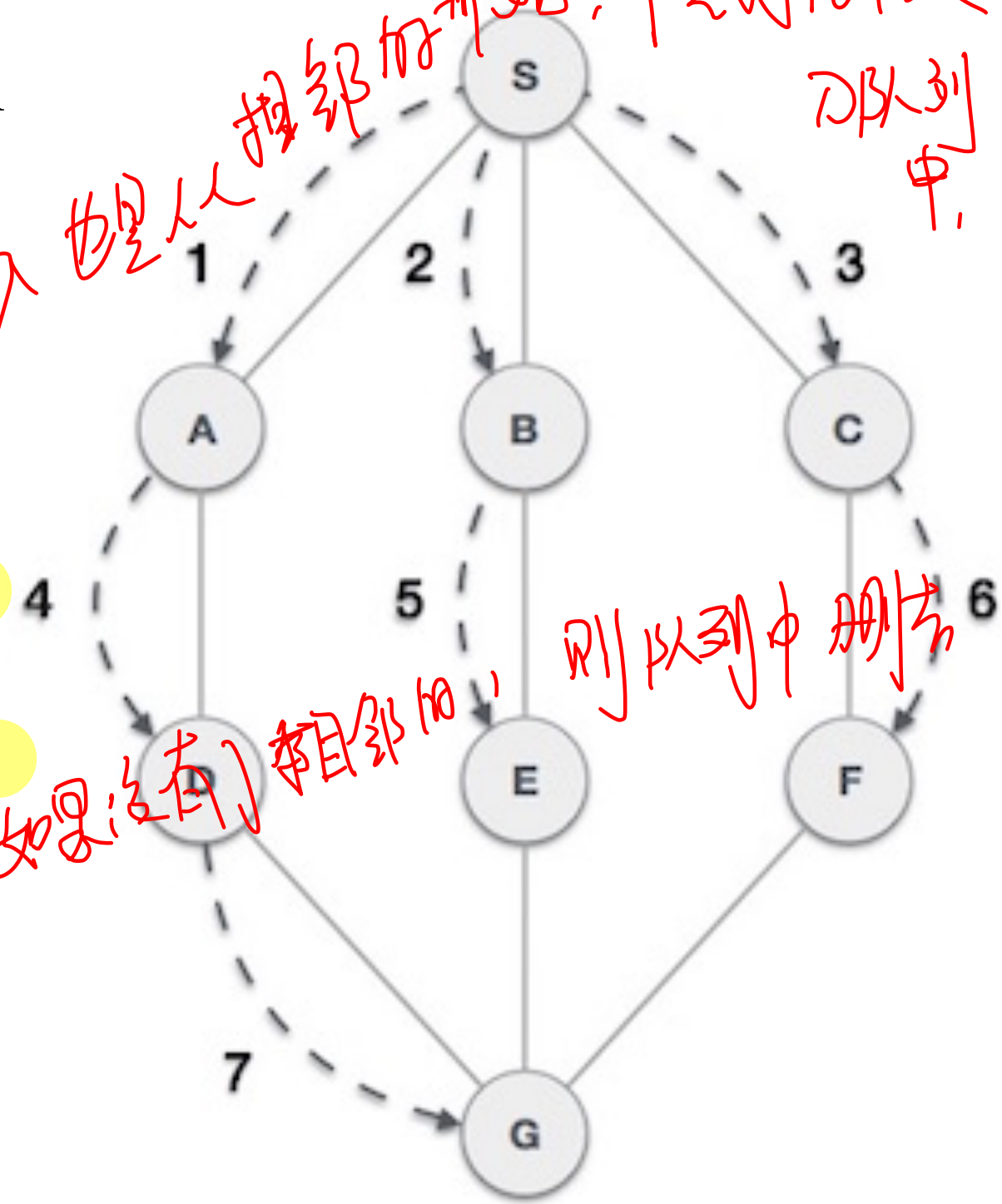
Breadth First Search

- Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a **queue** to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



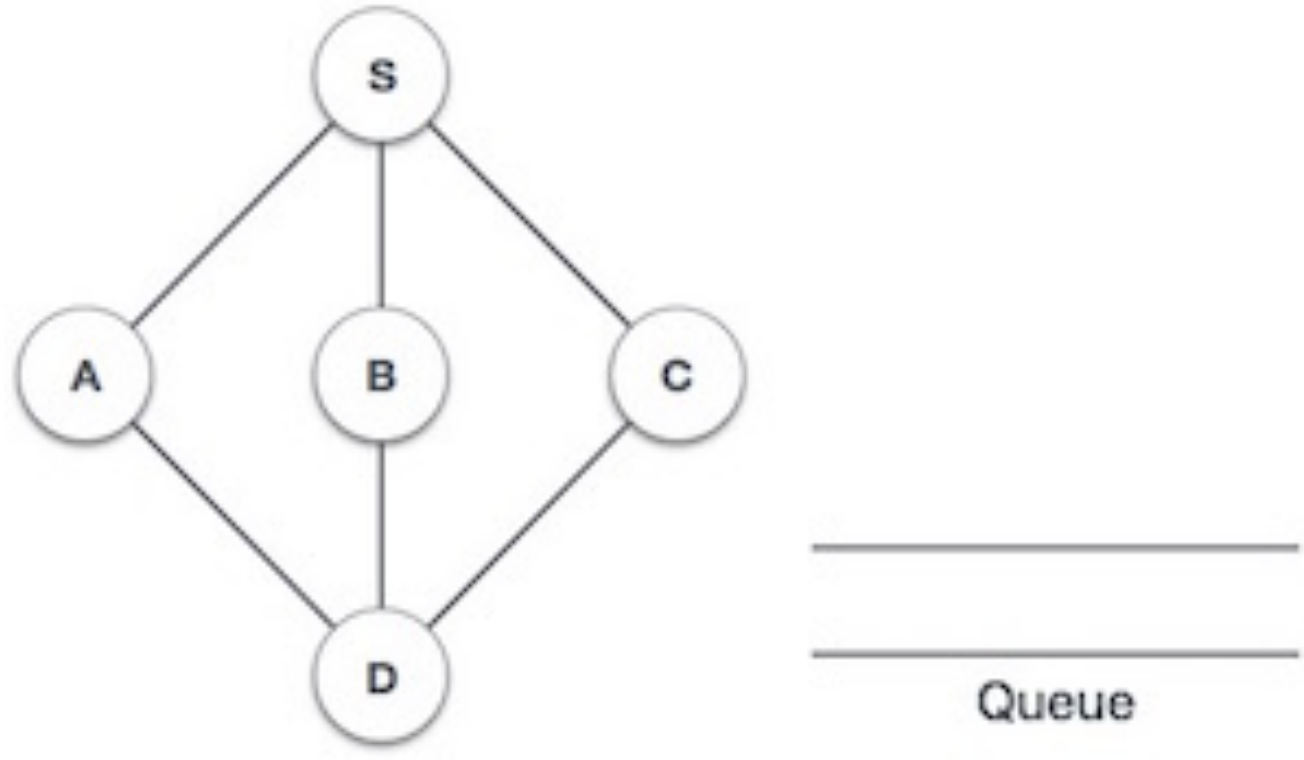
Breadth First Search

- As in the example, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D.
- It employs the following rules.
  - **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
  - **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
  - **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

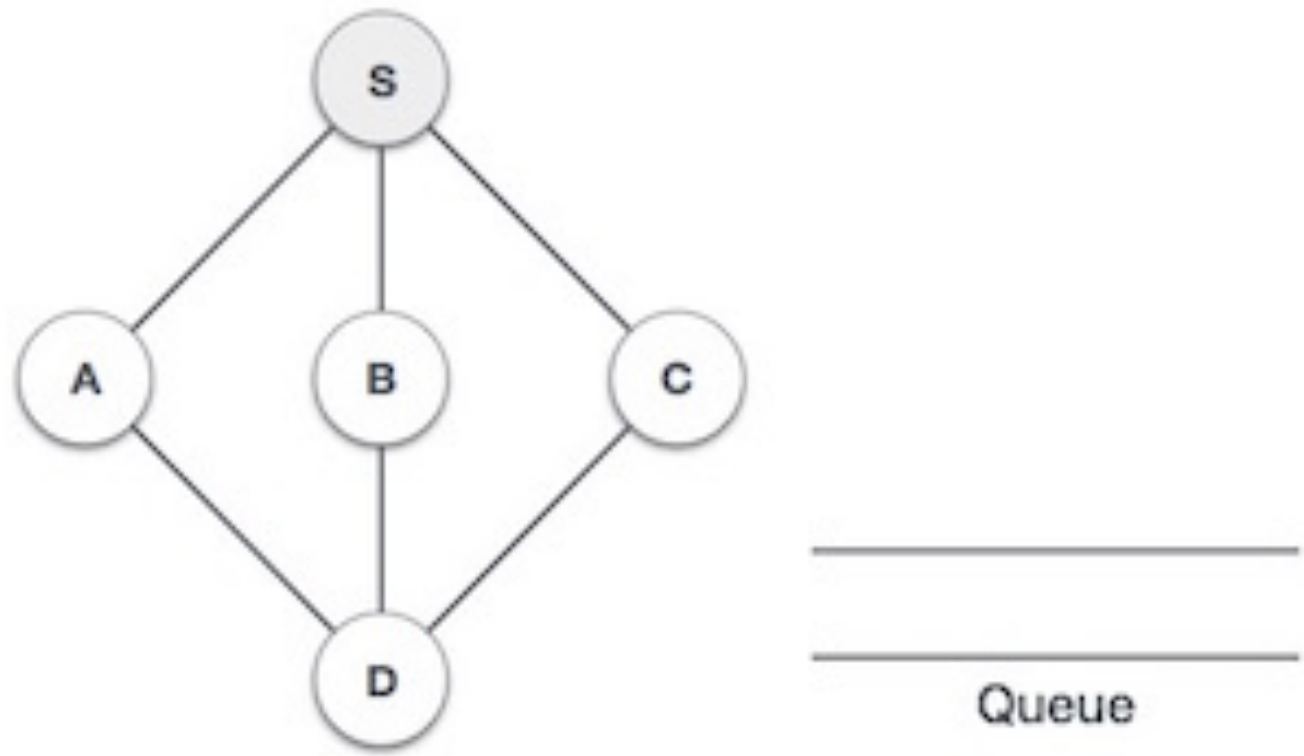


Example of BFS

- **Step 0.**
  - initialize the queue.

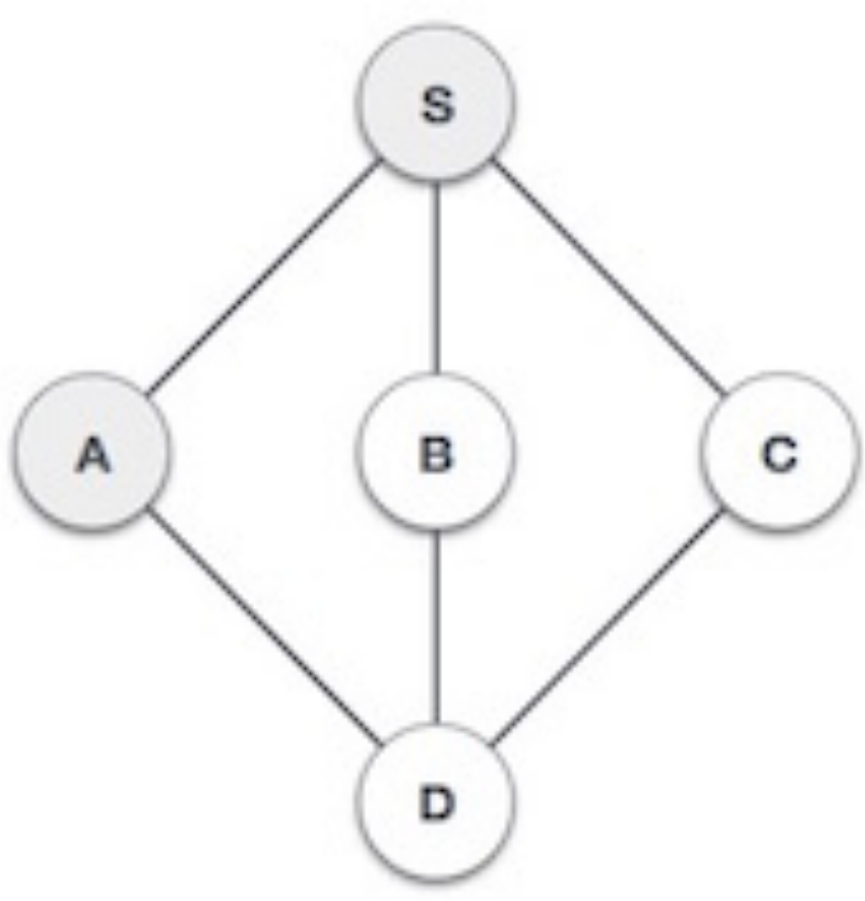


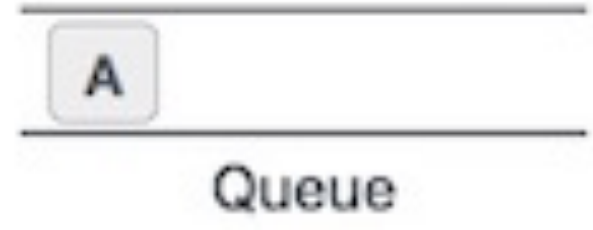
- **Step 1.**
  - We start from visiting **S** (starting node), and mark it as visited.



Example of BFS

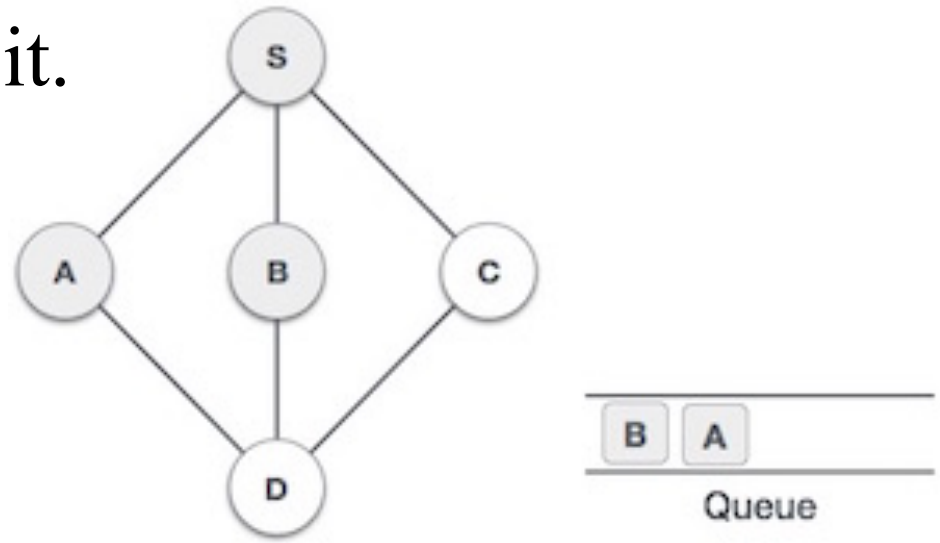
- **Step 2.**
  - We then see an unvisited adjacent node from **S**.
  - In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.



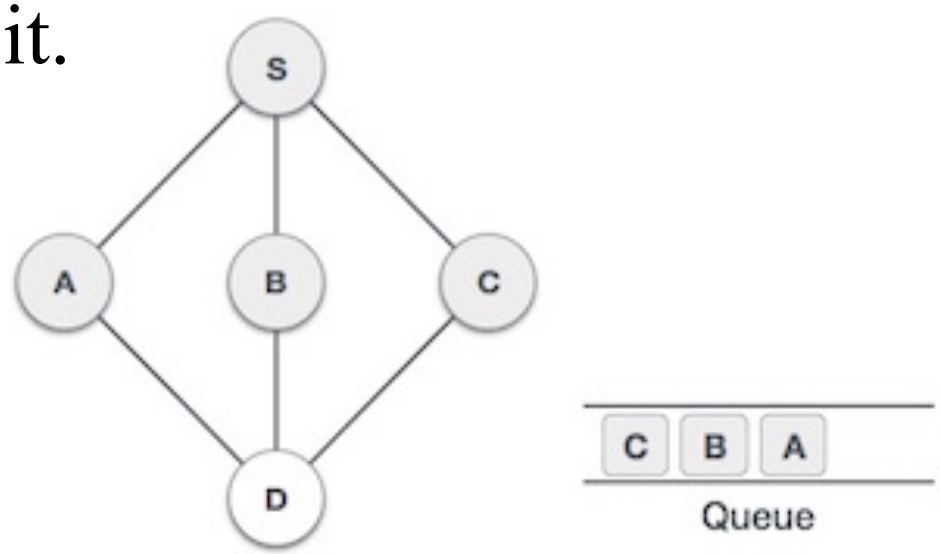


Example of BFS

- **Step 3.**
  - Next, the unvisited adjacent node from **S** is **B**.
  - We mark it as visited and enqueue it.

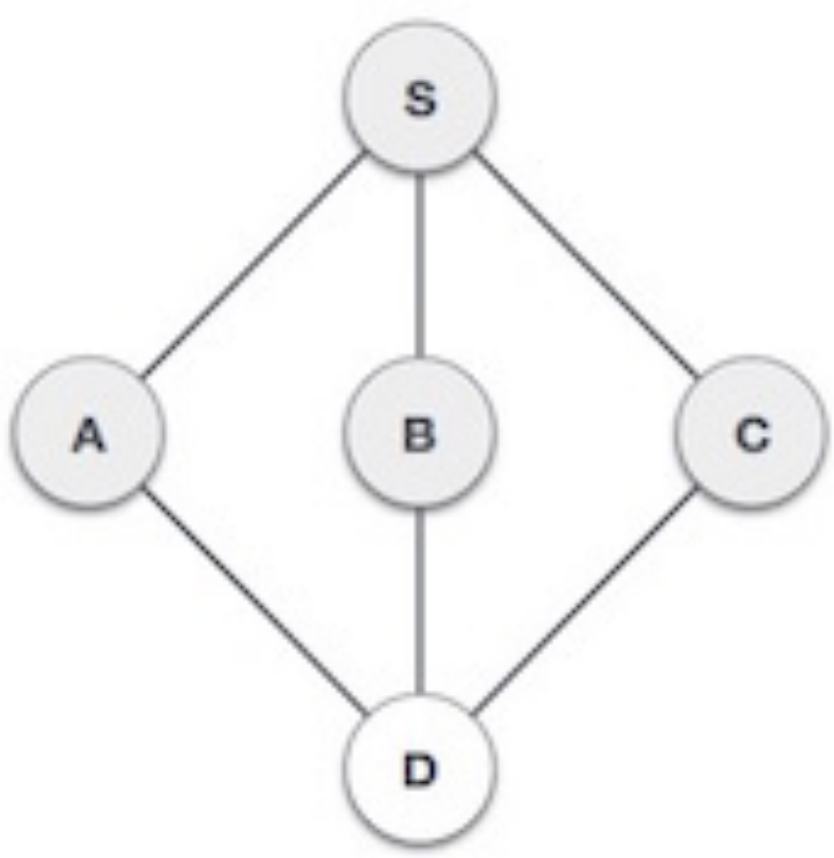


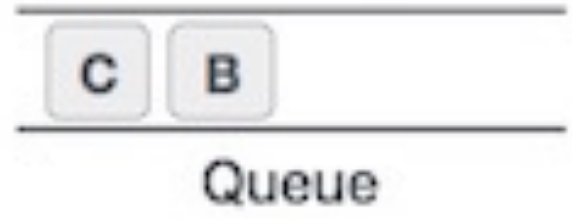
- **Step 4.**
  - Next, the unvisited adjacent node from **S** is **C**.
  - We mark it as visited and enqueue it.



Example of BFS

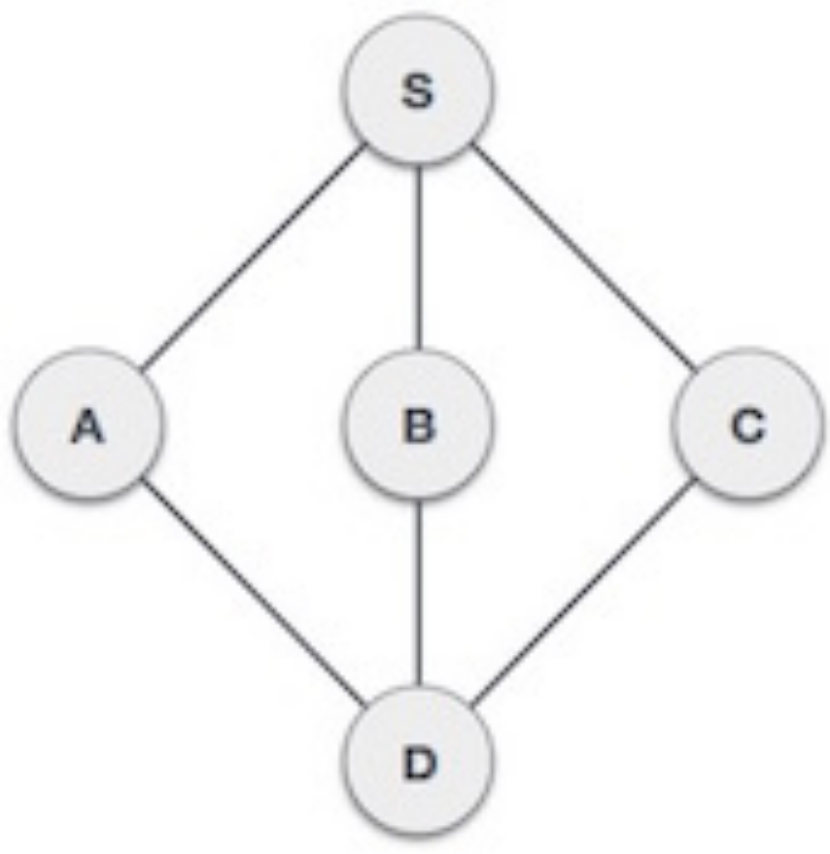
- **Step 5.**
  - Now, **S** is left with no unvisited adjacent nodes.
  - So, we **dequeue** and find **A**.

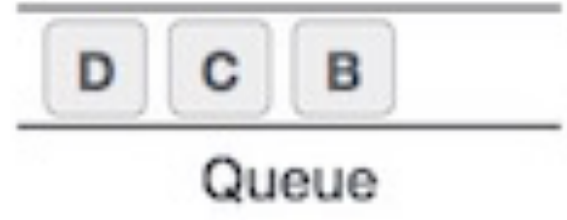




Example of BFS

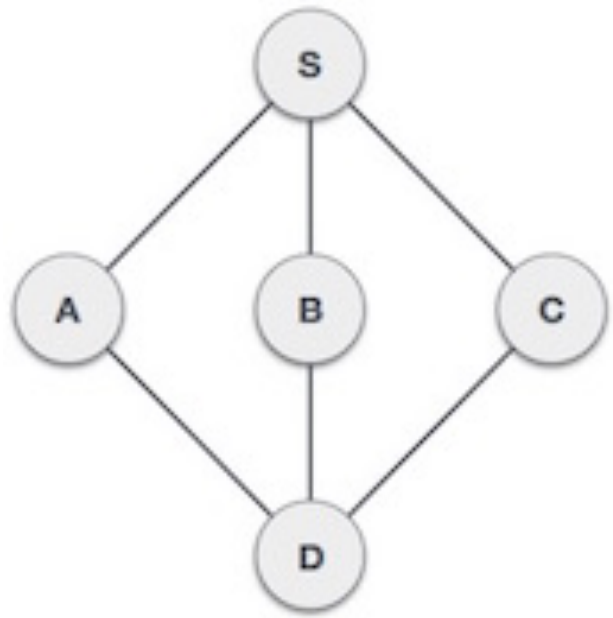
- **Step 6.**
  - From **A** we have **D** as unvisited adjacent node.
  - We mark it as visited and enqueue it.





Example of BFS

- **Step 7.**
  - At this stage, we are left with no unmarked (unvisited) nodes.
  - But as per the algorithm **we keep on dequeuing in order** to get all unvisited nodes.
    - When the queue gets emptied, the program is over.



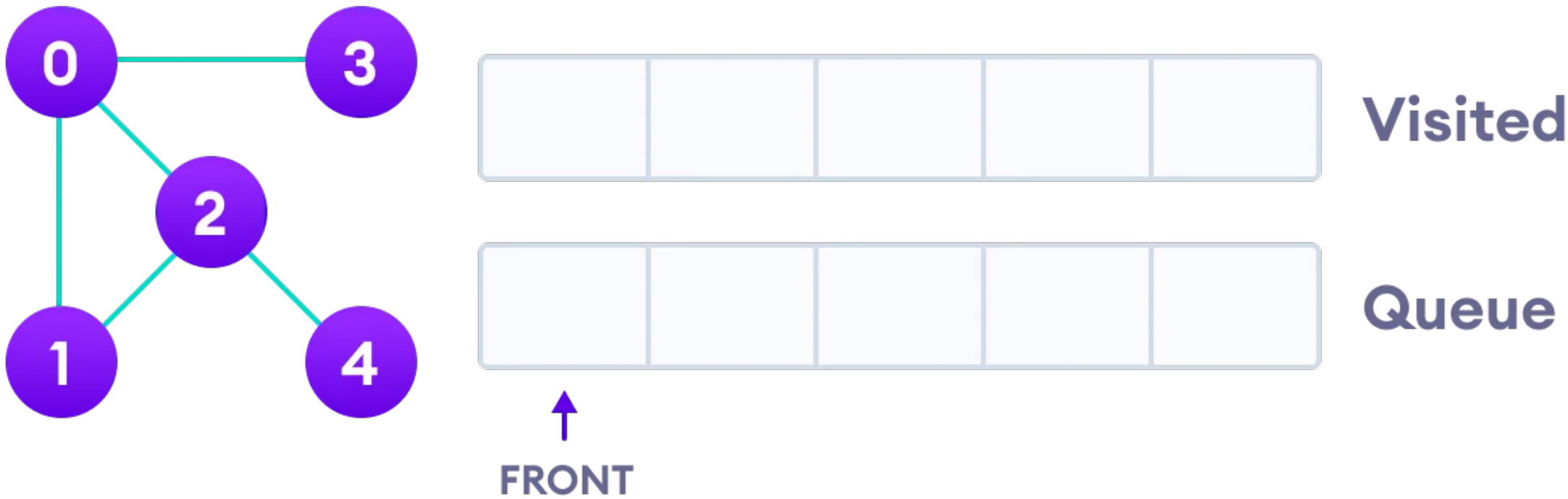
- **Output:**
  - **S, A, B, C, D**
    - We had mark **S** at Step 1 and dequeue **A** at Step 5.

BFS algorithm

- A standard BFS implementation puts each vertex of the graph into one of two categories:
  - Visited
  - Not Visited
- The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.
- The algorithm works as follows:
  1. Start by putting any one of the graph's vertices at the back of a queue.
  2. Take the front item of the queue and add it to the visited list.
  3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
  4. Keep repeating steps 2 and 3 until the queue is empty.

BFS example

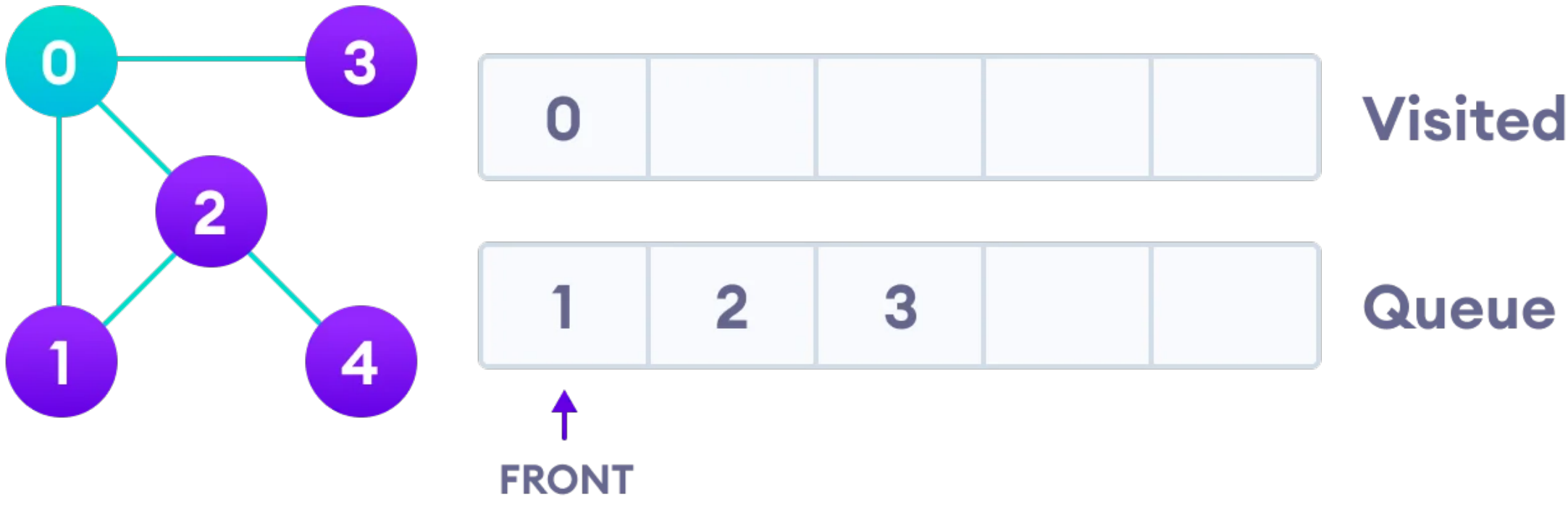
- Let's see how the Breadth First Search algorithm works with an example.
- We use an undirected graph with 5 vertices.



Undirected graph with 5 vertices

BFS example

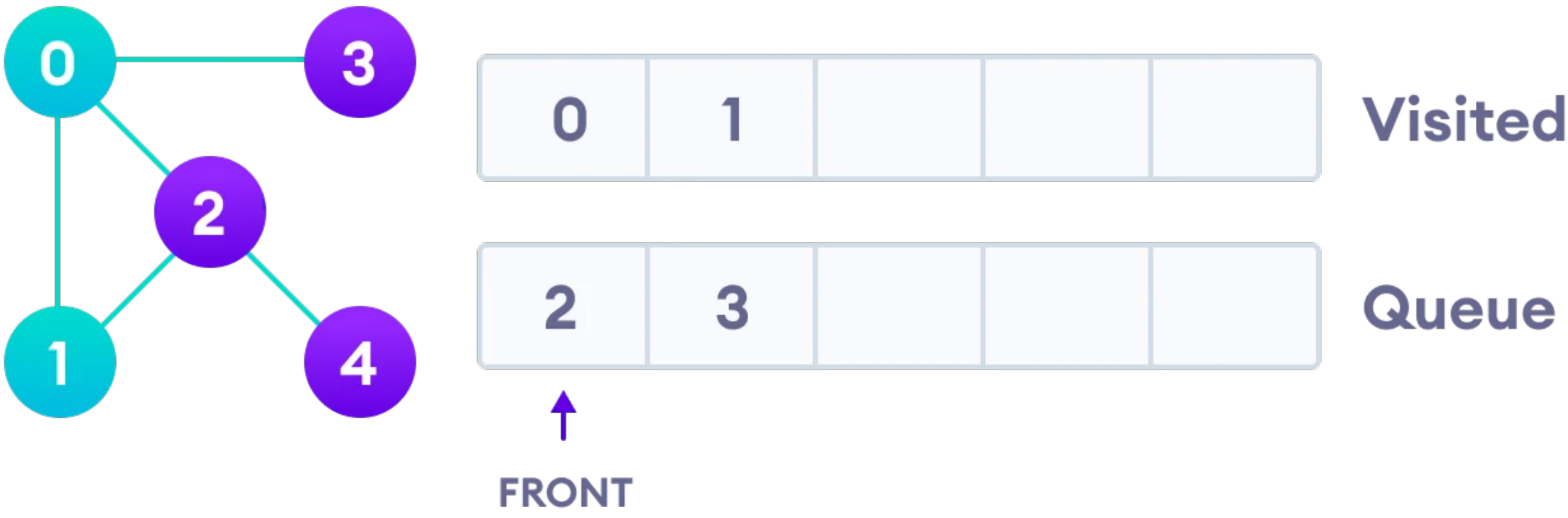
- We start from vertex 0, the BFS algorithm starts by putting it in the Visited list and putting all its adjacent vertices in the queue.



Visit start vertex and add its adjacent vertices to queue

BFS example

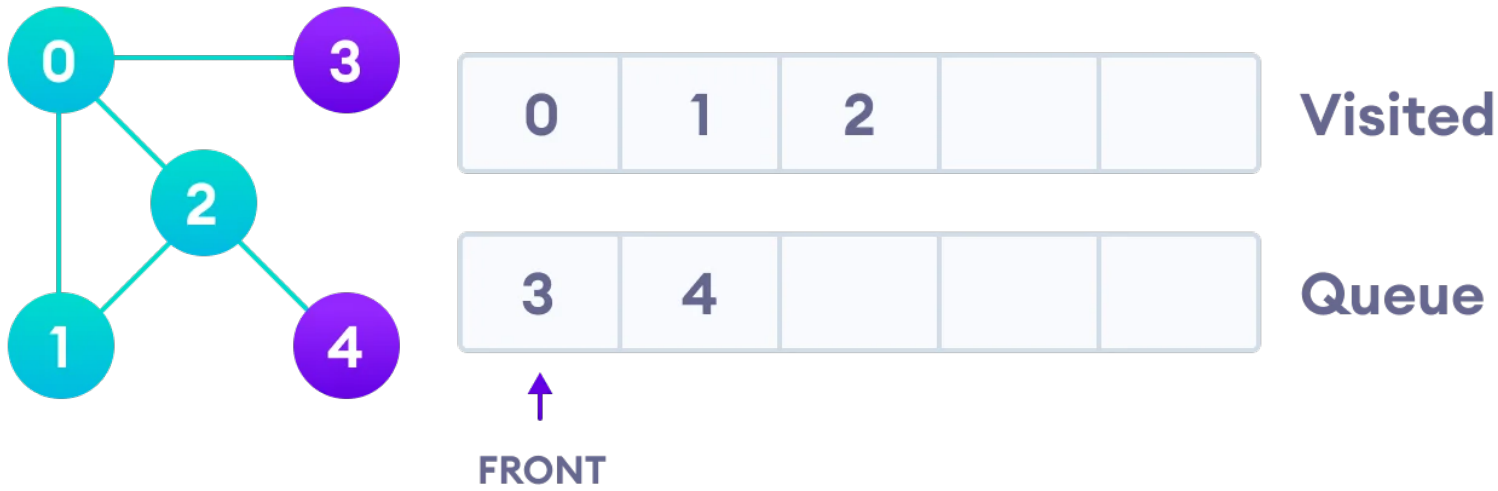
- Next, we visit the element at the front of queue i.e. 1 and go to its adjacent nodes.
- Since 0 has already been visited, we visit 2 instead.



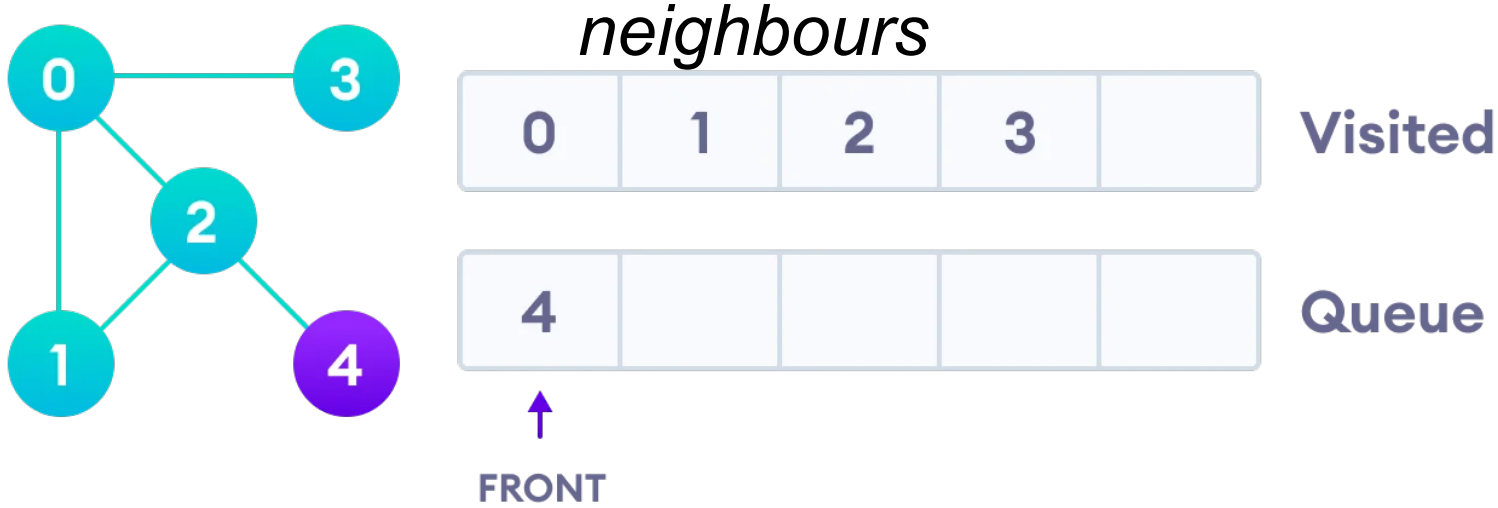
Visit the first neighbour of start node 0, which is 1

BFS example

- Vertex 2 has an unvisited adjacent vertex in 4, so we add that to the back of the queue and visit 3, which is at the front of the queue.



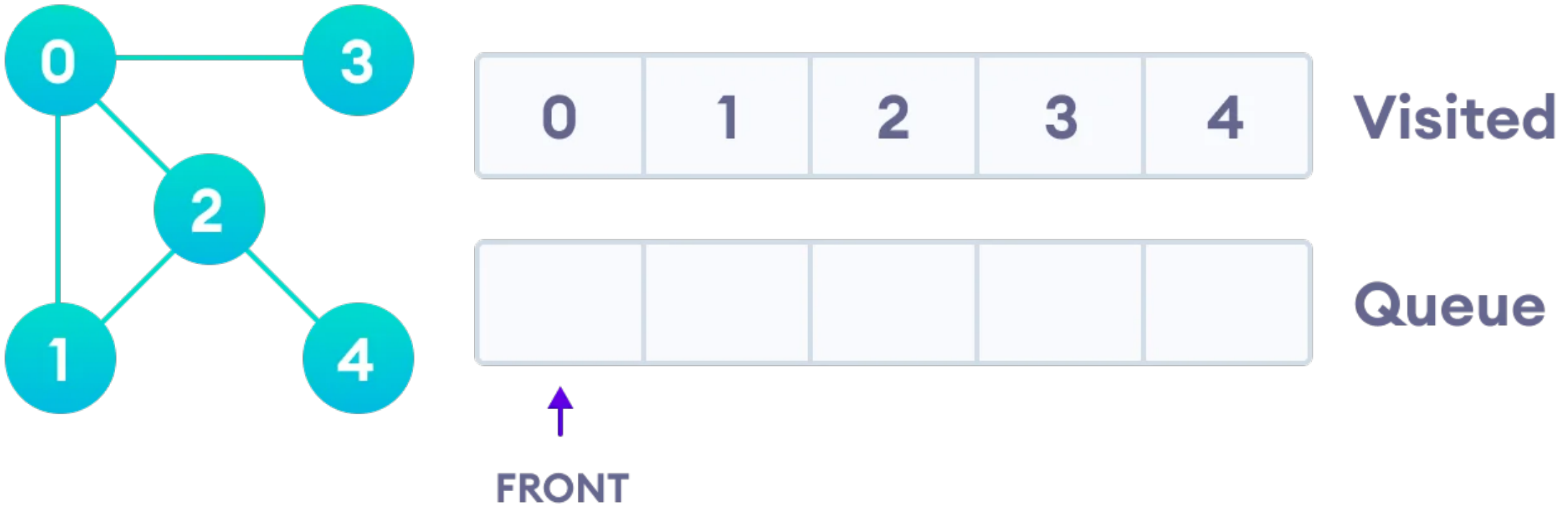
Visit 2 which was added to queue earlier to add its neighbours



4 remains in the queue

BFS example

- Only 4 remains in the queue since the only adjacent node of 3 i.e. 0 is already visited.
- We visit it.



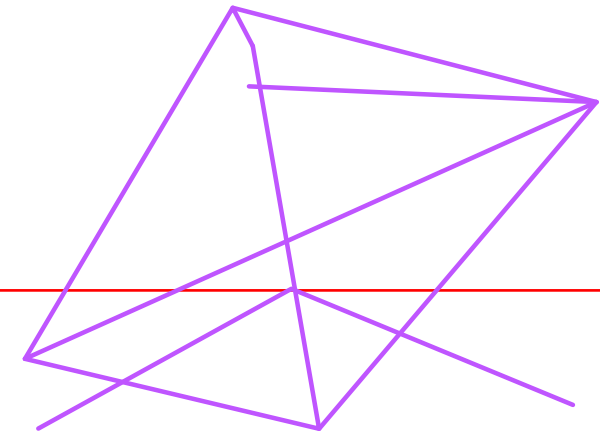
Visit last remaining item in the stack to check if it has unvisited neighbors

- Since the queue is empty, we have completed the Breadth First Traversal of the graph.

BFS pseudocode

```
create a queue Q
mark v as visited and put v into Q
while Q is non-empty
    remove the head u of Q
    mark and enqueue all (unvisited) neighbours of u
```

BFS in Java



```
// BFS algorithm in Java
import java.util.*;

public class Graph {
      private int V;
      private LinkedList<Integer> adj[];

      // Create a graph Graph(int v) {
            V = v;
            adj = new LinkedList[v];
            for (int i = 0; i < v; ++i) adj[i] = new LinkedList();
      }
```

BFS in Java

```
// Add edges to the graph
void addEdge(int v, int w) {
    adj[v].add(w);
}
```

BFS in Java

```
// BFS algorithm
void BFS(int s) {
        boolean visited[] = new boolean[V];
        LinkedList<Integer> queue = new LinkedList();
        visited[s] = true;
        queue.add(s);
        while (queue.size() != 0) {
                s = queue.poll();
                System.out.print(s + " ");
                Iterator<Integer> i = adj[s].listIterator();
                while (i.hasNext()) {
                        int n = i.next();
                        if (!visited[n]) {
                                visited[n] = true;
                                queue.add(n);
                        }
                }
        }
}
```

BFS in Java

```
public static void main(String args[]) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        System.out.println("Following is Breadth First Traversal
" + "(starting from vertex 2)");
        g.BFS(2);
    }
}
```

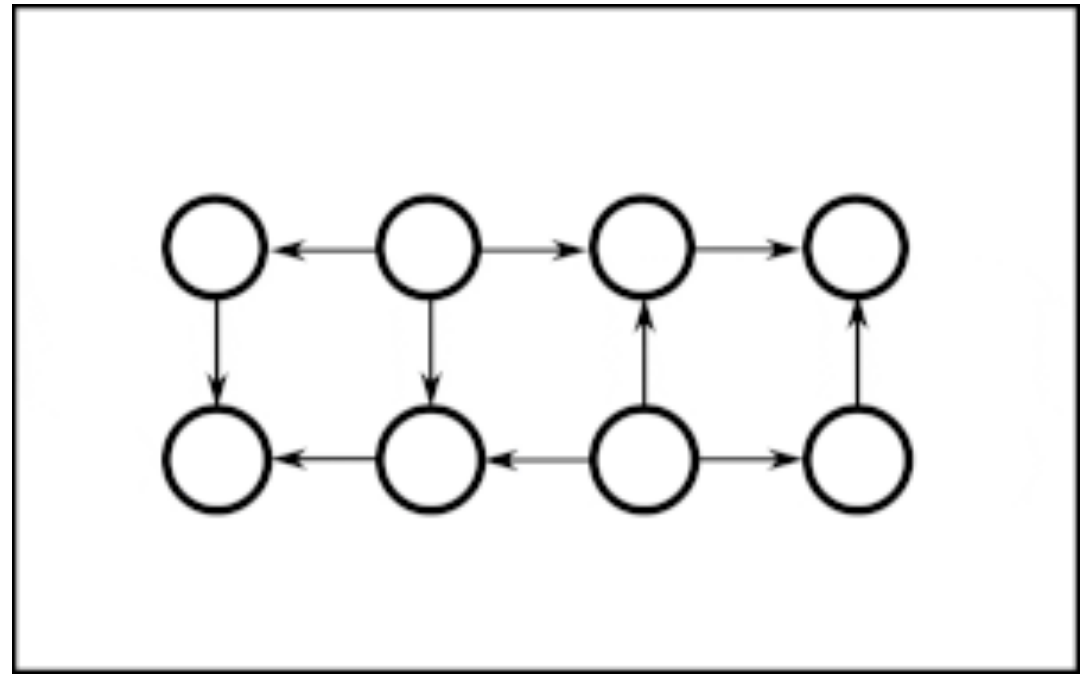
BFS Algorithm Complexity

- The time complexity of the BFS algorithm is represented in the form of **O(V + E),** where V is the number of nodes and E is the number of edges.
- The space complexity of the algorithm is **O(V)**.

- Graph
- Graph Data Structure
- Depth First Search
- Breadth First Search
- **Topological Sorting**

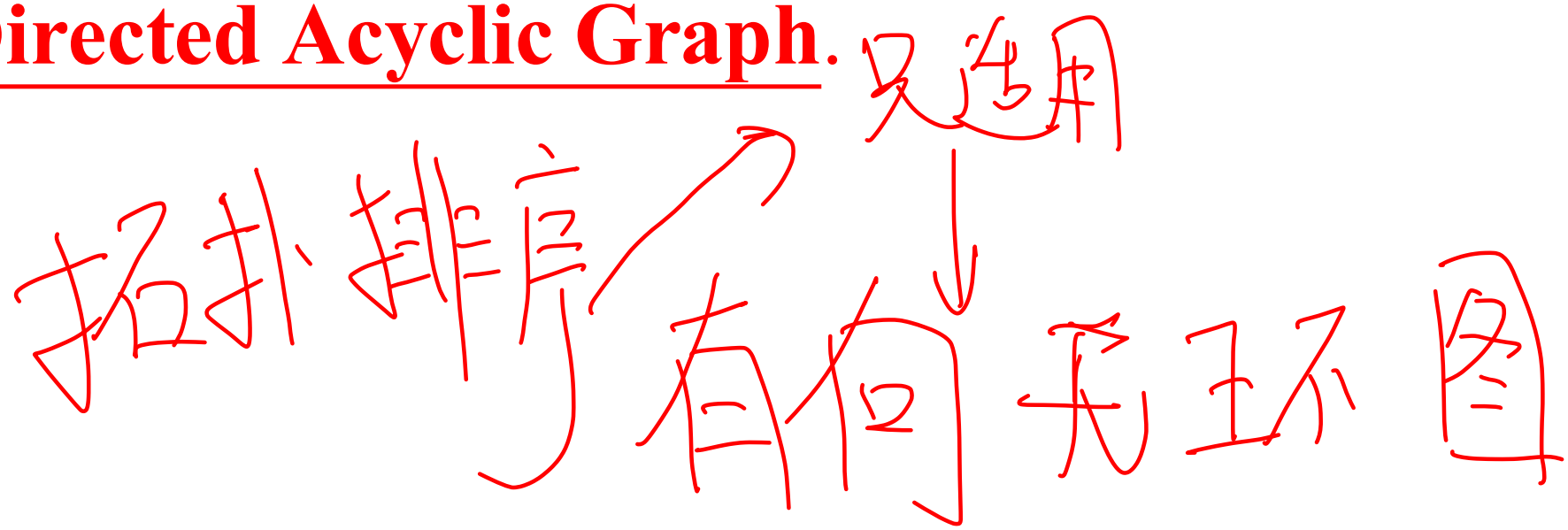
Topological Sorting

- Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs.
- In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers
- processing elements (PE)



Topological sort

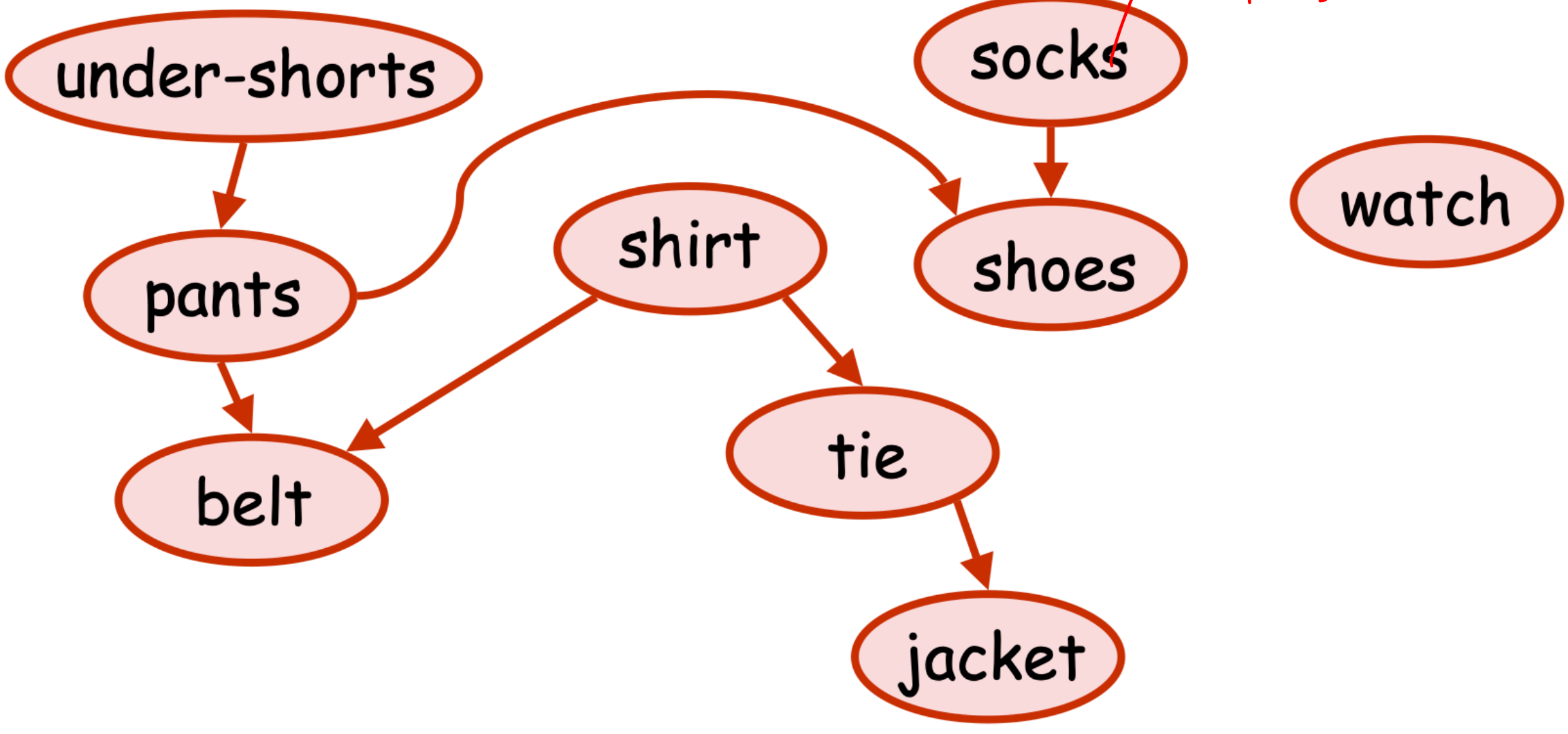
- Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering.
- Topological Sorting for a graph is not possible if the graph is not a DAG.
- Topological Sorting is possible if and only if the graph is a **Directed Acyclic Graph**.



Topological sort

- Directed graph can be used to indicate precedence among a set of events
- E.g., a possible precedence is dressing

有向无环图.



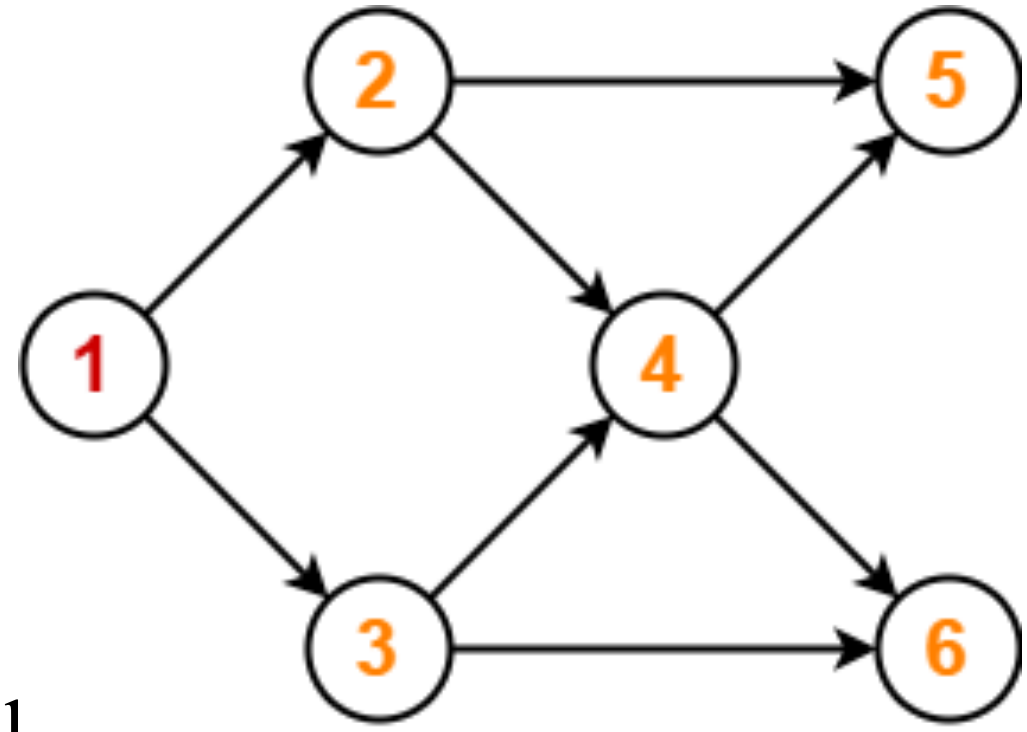
Topological Sort

- Fact:
  - If G contains a cycle, then it is impossible to find a desired ordering
    - (Prove by contradiction)

- However, if G is acyclic (contains no cycles) we shall give two algorithms that always find the desired ordering

Topological Sort Example

- Consider the following directed acyclic graph.



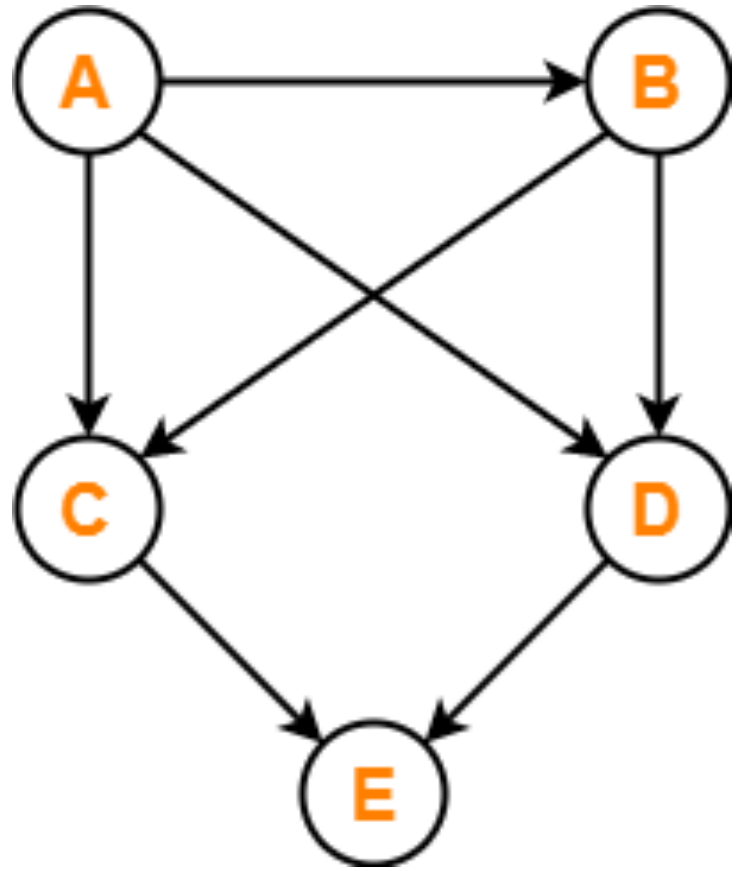
- For this graph, following 4 different topological orderings are possible.
  - **1 2 3 4 5 6**
  - **1 2 3 4 6 5**
  - **1 3 2 4 5 6**
  - **1 3 2 4 6 5**

规则:

- 图中每个顶点只出现 `一次` 。
- A在B前面，则不存在B在A前面的路径。(`不能成环！！！！`)
- 顶点的顺序是**保证所有指向它的下个节点在被指节点前面**！(例如A—>B—>C那么A一定在B前面，B一定在C前面)。所以，这个核心规则下只要满足即可，**所以拓扑排序序列不一定唯一**！

Problem-01

- Find the number of different topological orderings possible for the given graph-



A B C D E

A C B D E

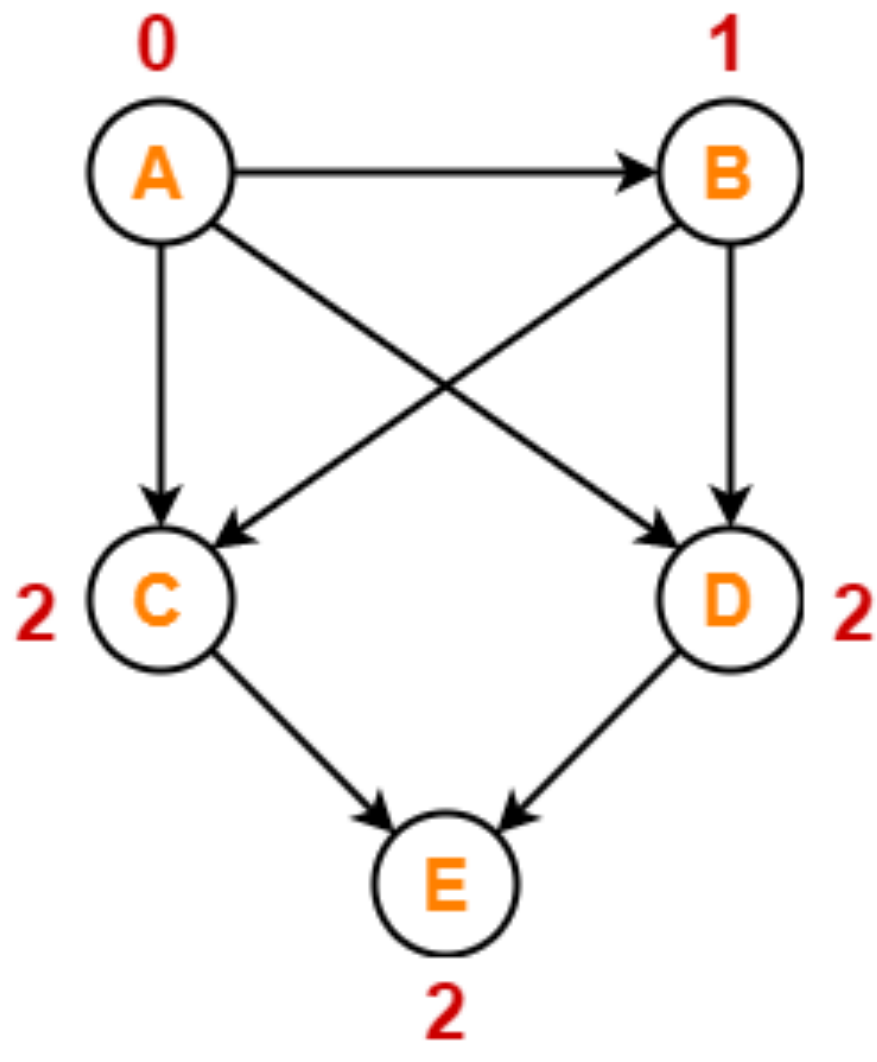
A D B C

A D C B

A

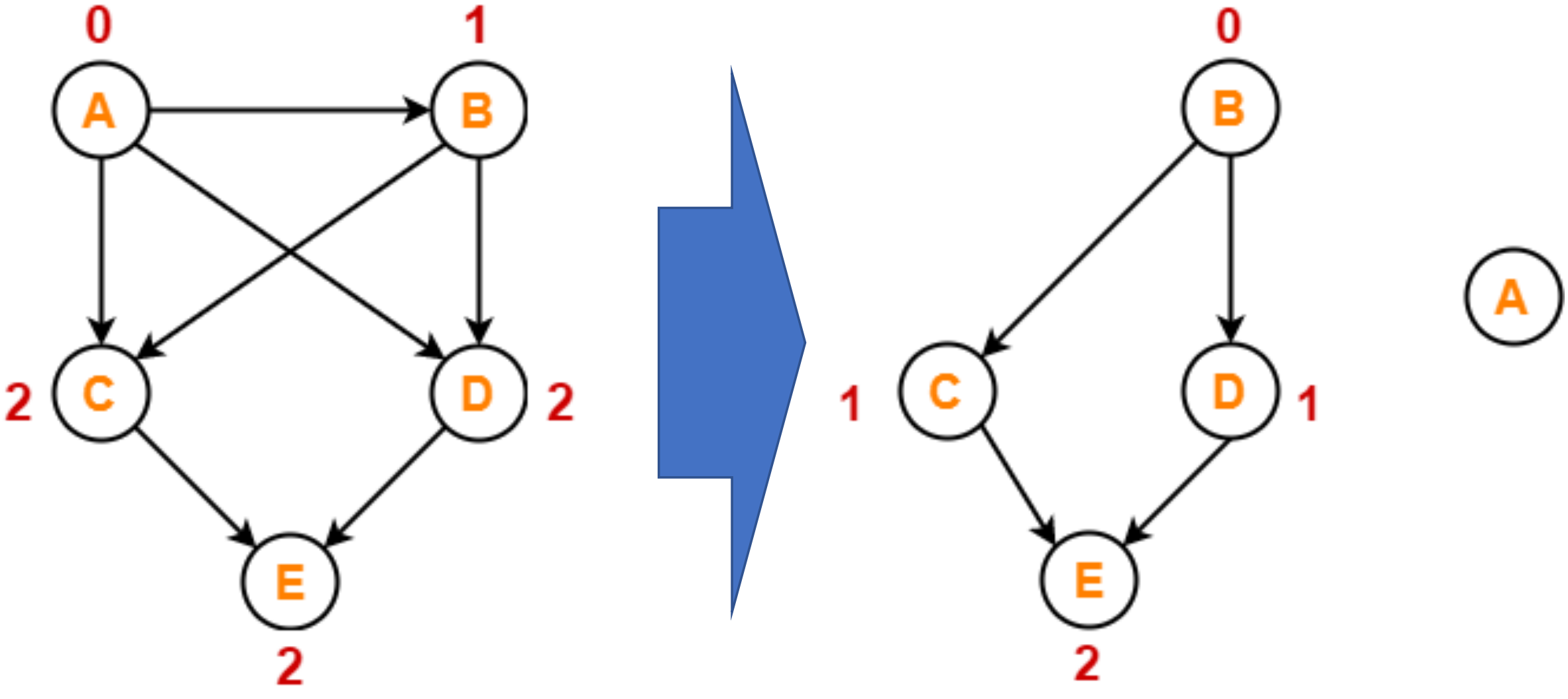
Problem-01

- **Solution.**
  - The topological orderings of the above graph are found in the following steps.
  - **Step-01:** Write **in-degree** of each vertex.



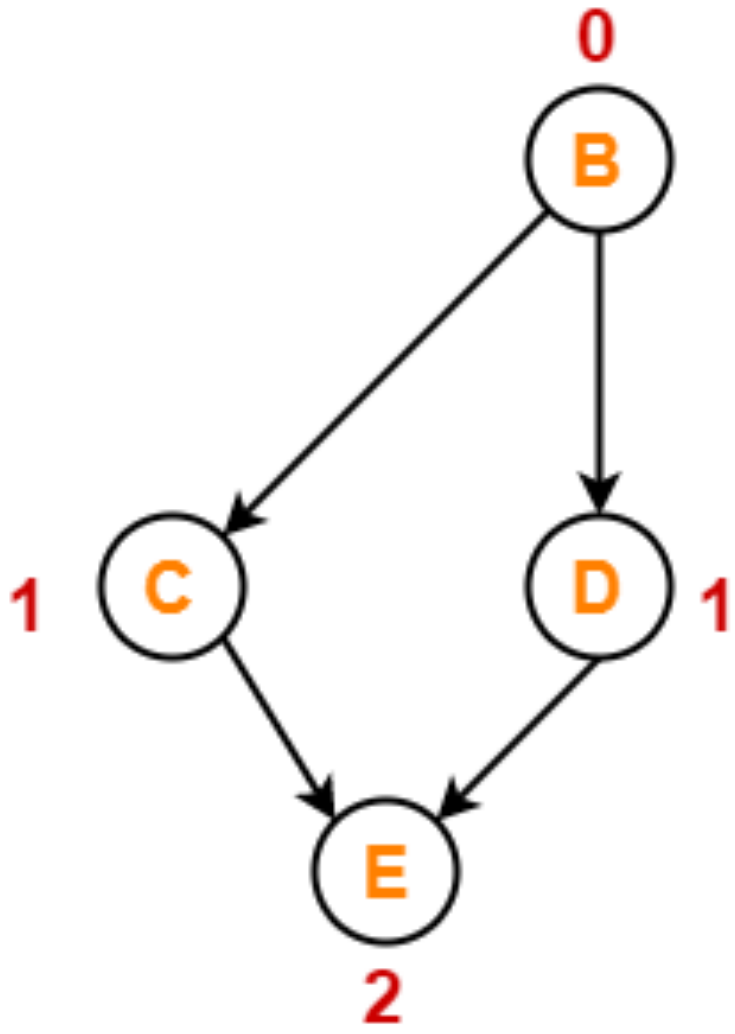
Problem-01

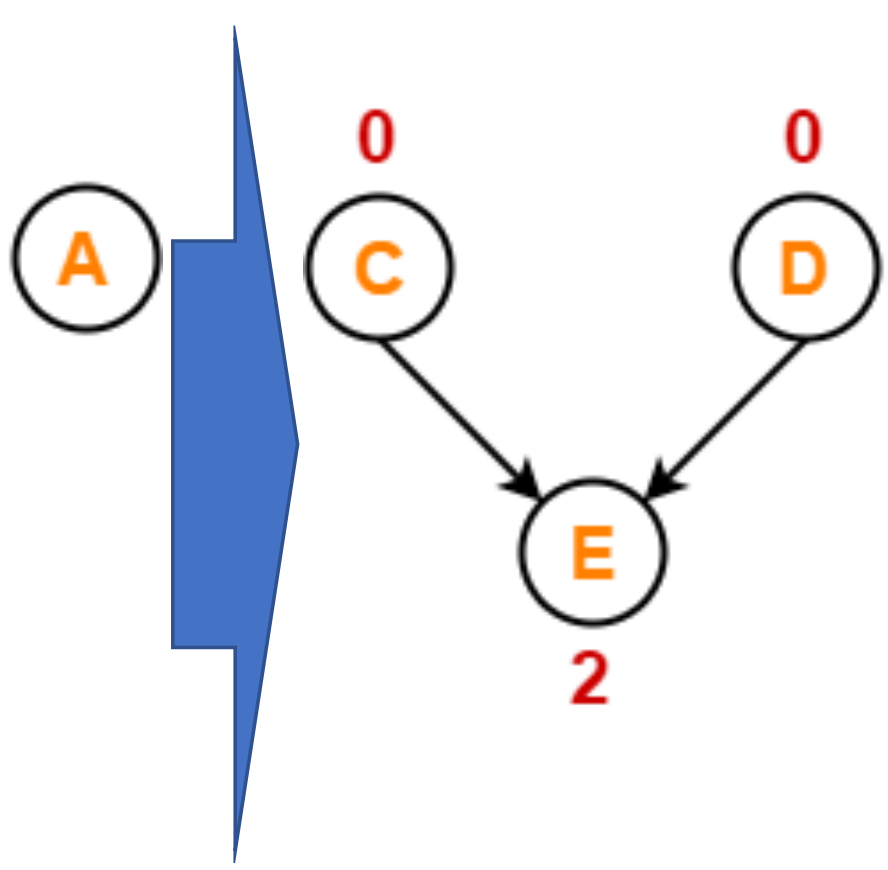
- **Step-02:**
  - Vertex-A has the least in-degree.
  - So, remove vertex-A and its associated edges.
  - Now, update the in-degree of other vertices.

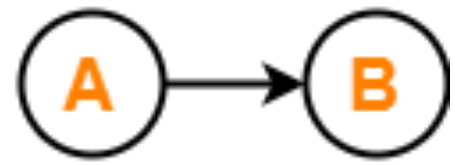


Problem-01

- **Step-03:**
  - Vertex-B has the least in-degree.
  - So, remove vertex-B and its associated edges.
  - Now, update the in-degree of other vertices.





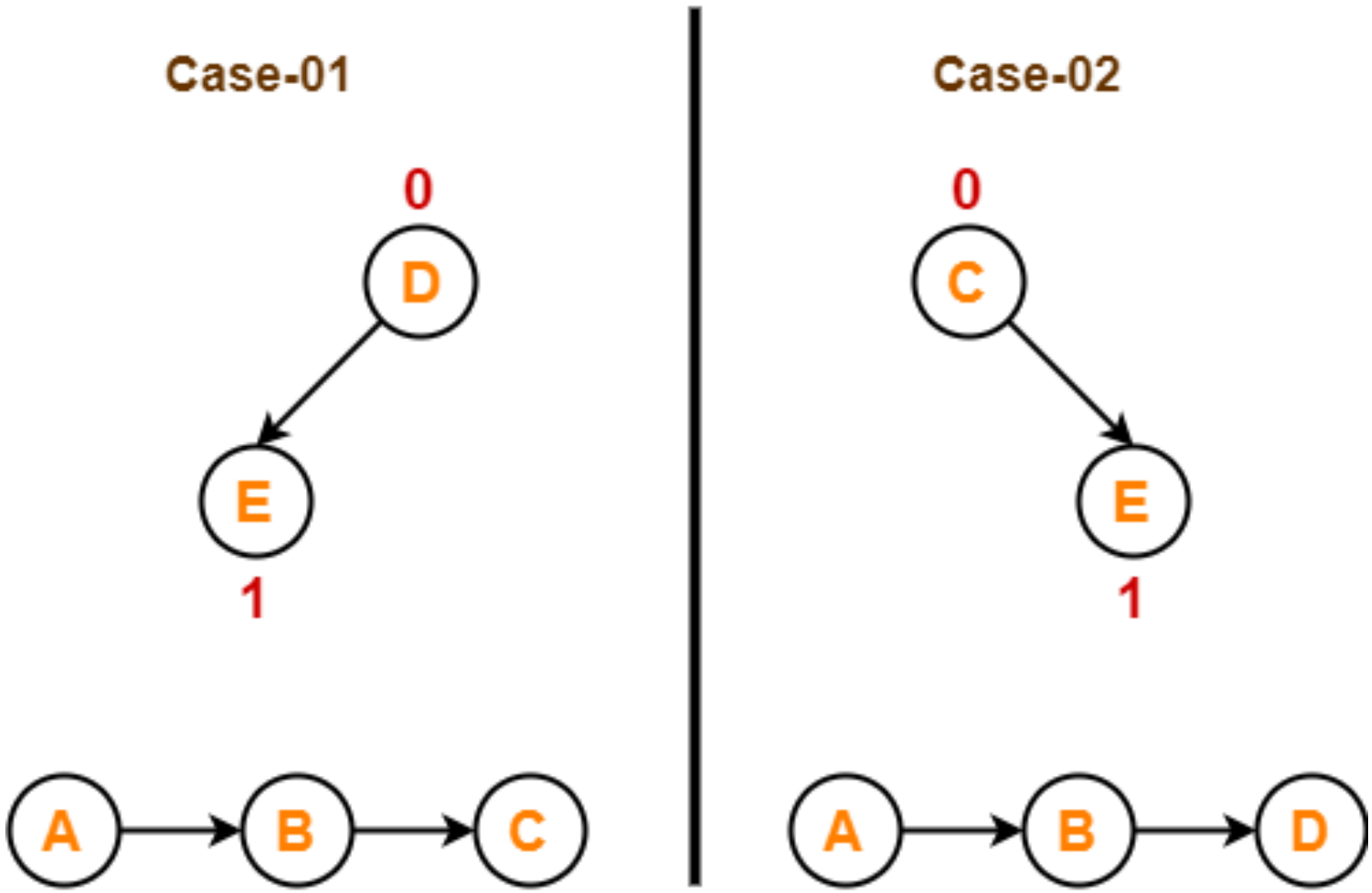


Problem-01

- **Step-04:**
  - There are two vertices with the least in-degree. So, following 2 cases are possible
  - In case-01:
    - Remove vertex-C and its associated edges.
    - Then, update the in-degree of other vertices.
  - In case-02:
    - Remove vertex-D and its associated edges.
    - Then, update the in-degree of other vertices.

Problem-01

- **Step-04:**
  - In case-01: Remove vertex-C and its associated edges.
  - In case-02: Remove vertex-D and its associated edges.

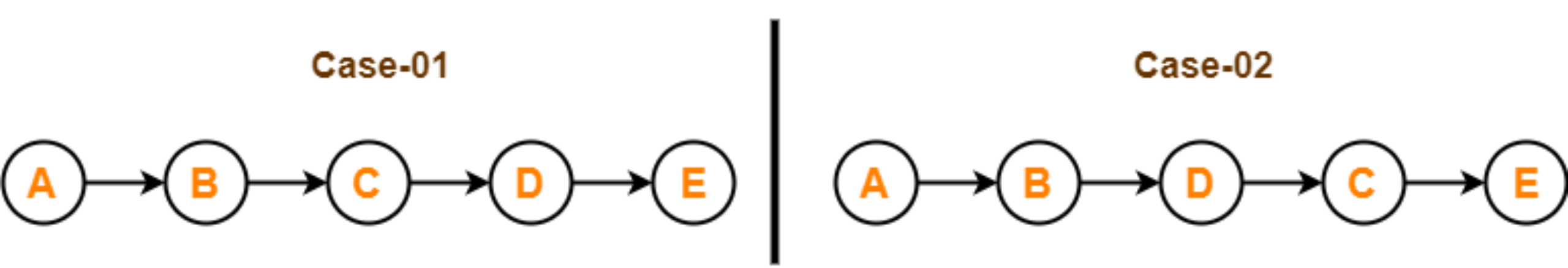


Problem-01

- **Step-05:**
  - Now, the above two cases are continued separately in the similar manner.
  - In case-01:
    - Remove vertex-D since it has the least in-degree.
    - Then, remove the remaining vertex-E.
  - In case-02:
    - Remove vertex-C since it has the least in-degree.
    - Then, remove the remaining vertex-E.

Problem-01

- **Conclusion:**
  - For the given graph, following **2** different topological orderings are possible.
    - **A B C D E**
    - **A B D C E**



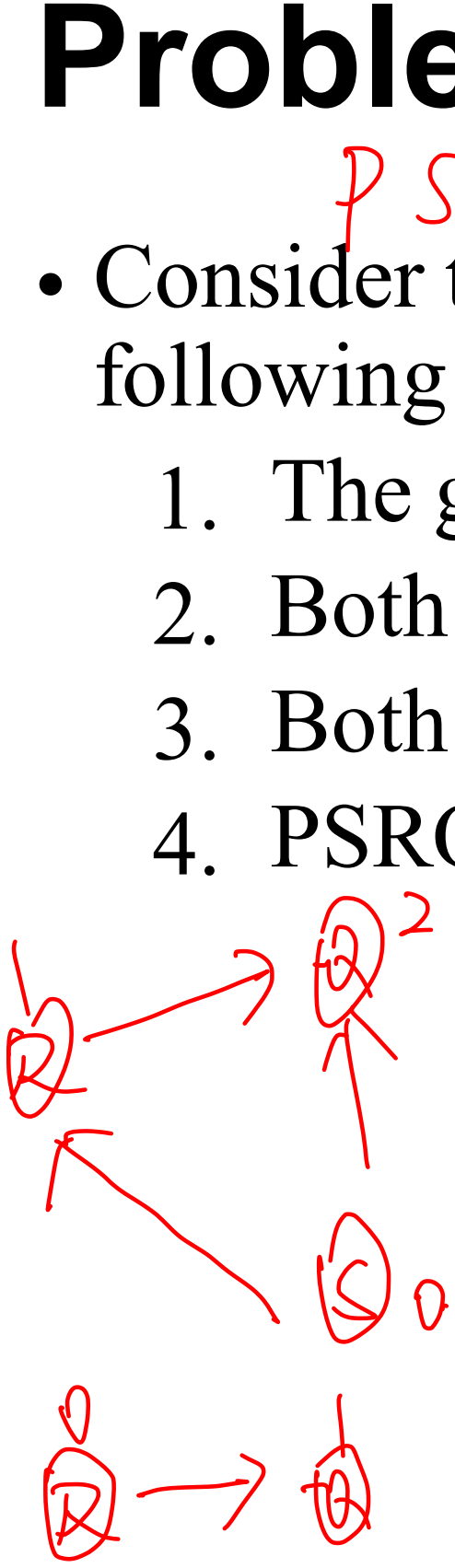
Problem-02

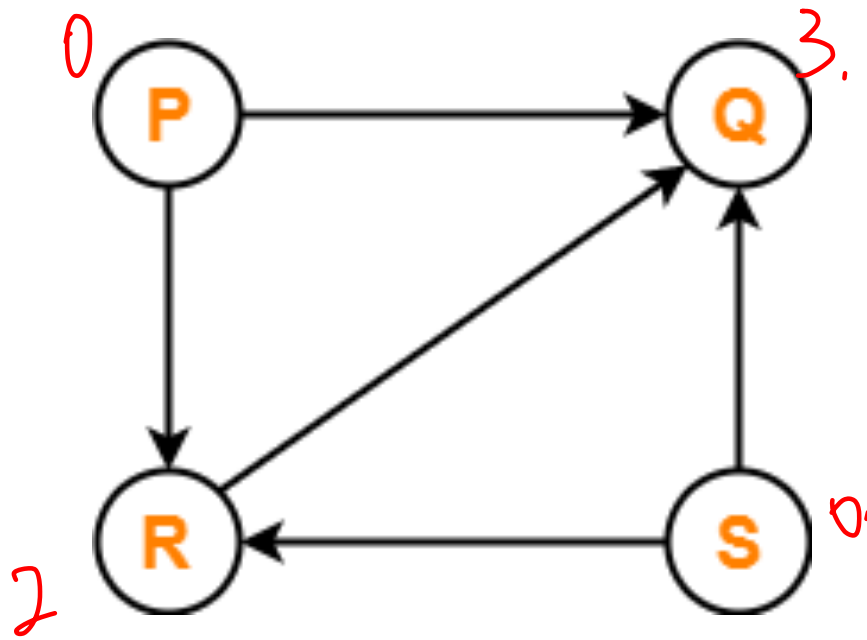
法一：定理(拓扑定义)

法二：删"0"法(0度)

PSRQ. SPRQ

- Consider the directed graph given below. Which of the following statements is true?
  1. The graph does not have any topological ordering.
  2. Both PQRS and SRPQ are topological orderings.
  3. Both PSRQ and SPRQ are topological orderings.
  4. PSRQ is the only topological ordering.





PSRQ

SPRQ

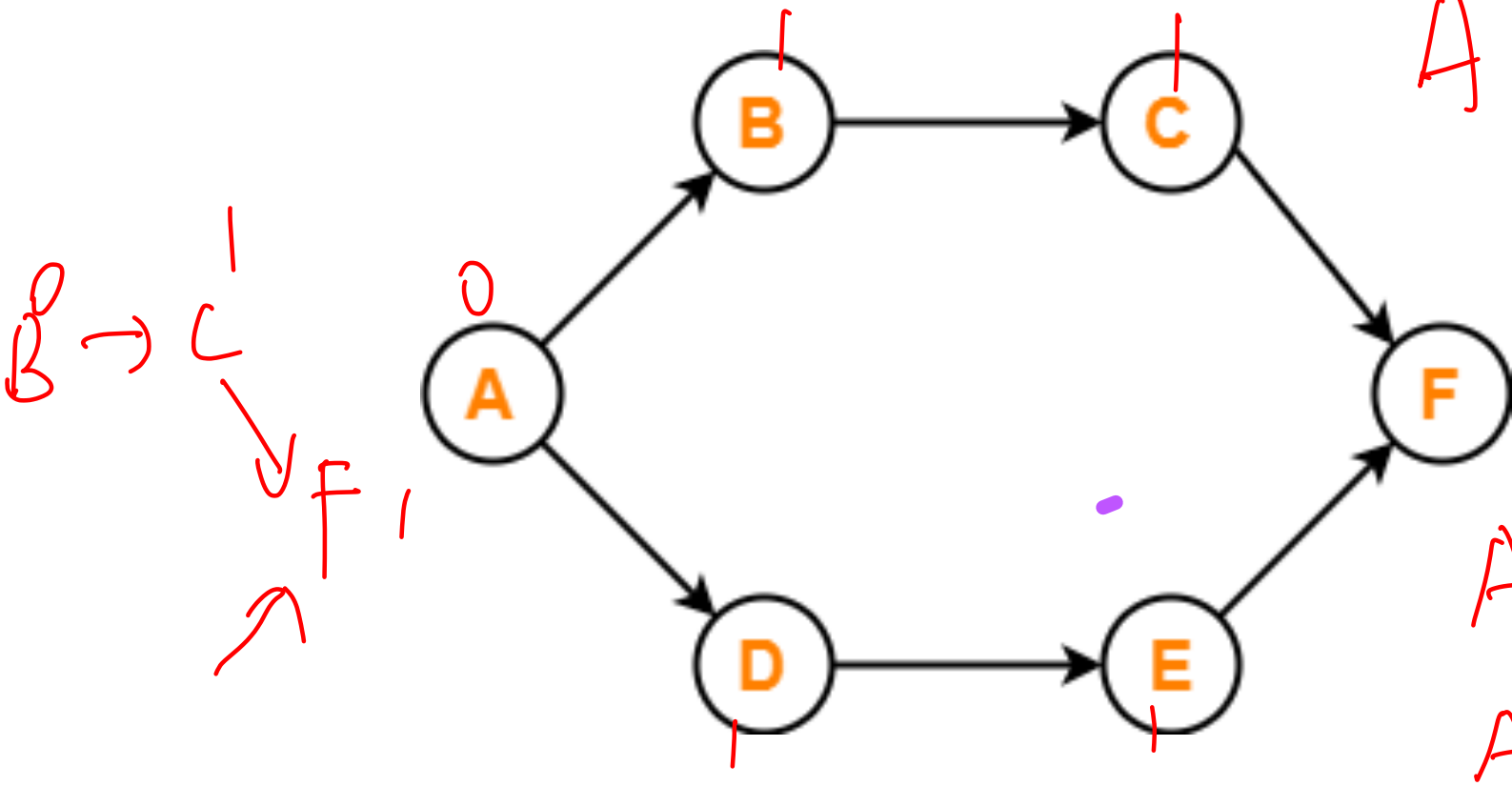
Problem-02

- **Solution.**
  - The given graph is a directed acyclic graph.
  - So, topological orderings exist.
  - P and S must appear before R and Q in topological orderings as per the definition of topological sort.

- Thus, Correct option is (**3**).

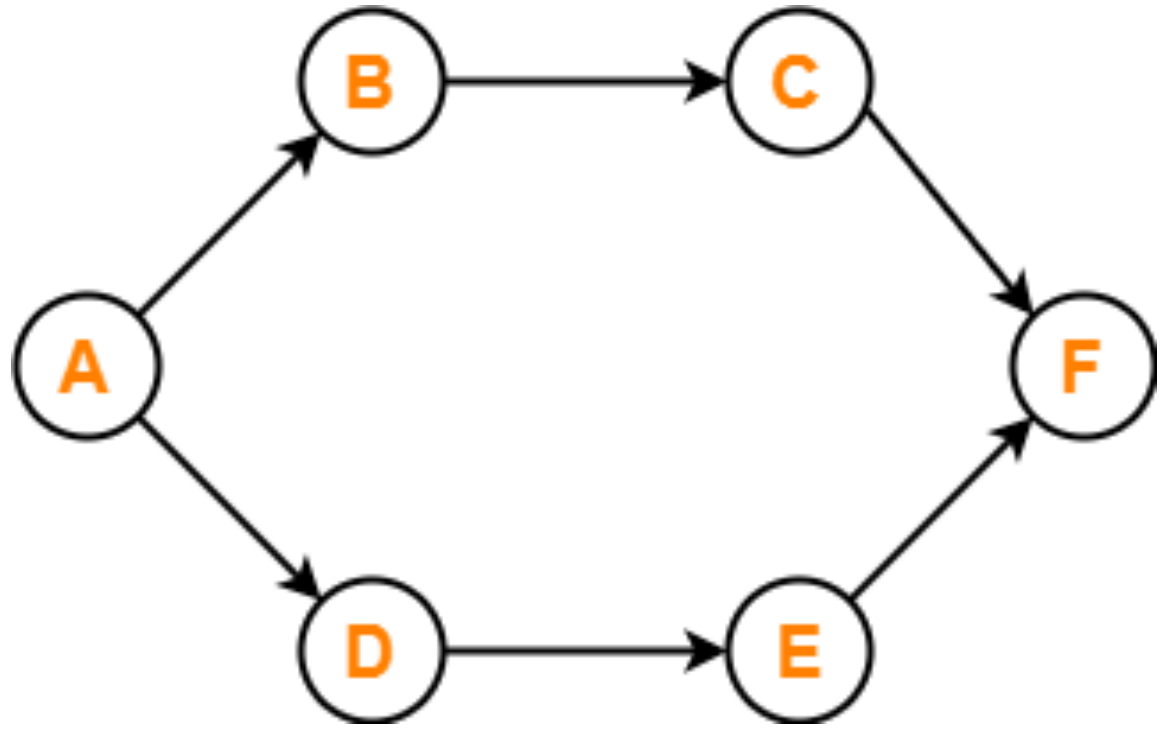
Problem-03

- Consider the following directed graph.
- The number of different topological orderings of the vertices of the graph is _____ ?



Problem-03

- **Solution.**
  - Number of different topological orderings possible = 6.
  - Thus, Correct answer is **6**.



Topological Sort

- We introduce the Topological Sort problem on directed acyclic graph (DAG)

- We give two linear-time algorithms :
  - (1) Using Queue
  - (2) Using Stack

Topological Sort (with Queue)

Topological-Sort(G) // given G is acyclic

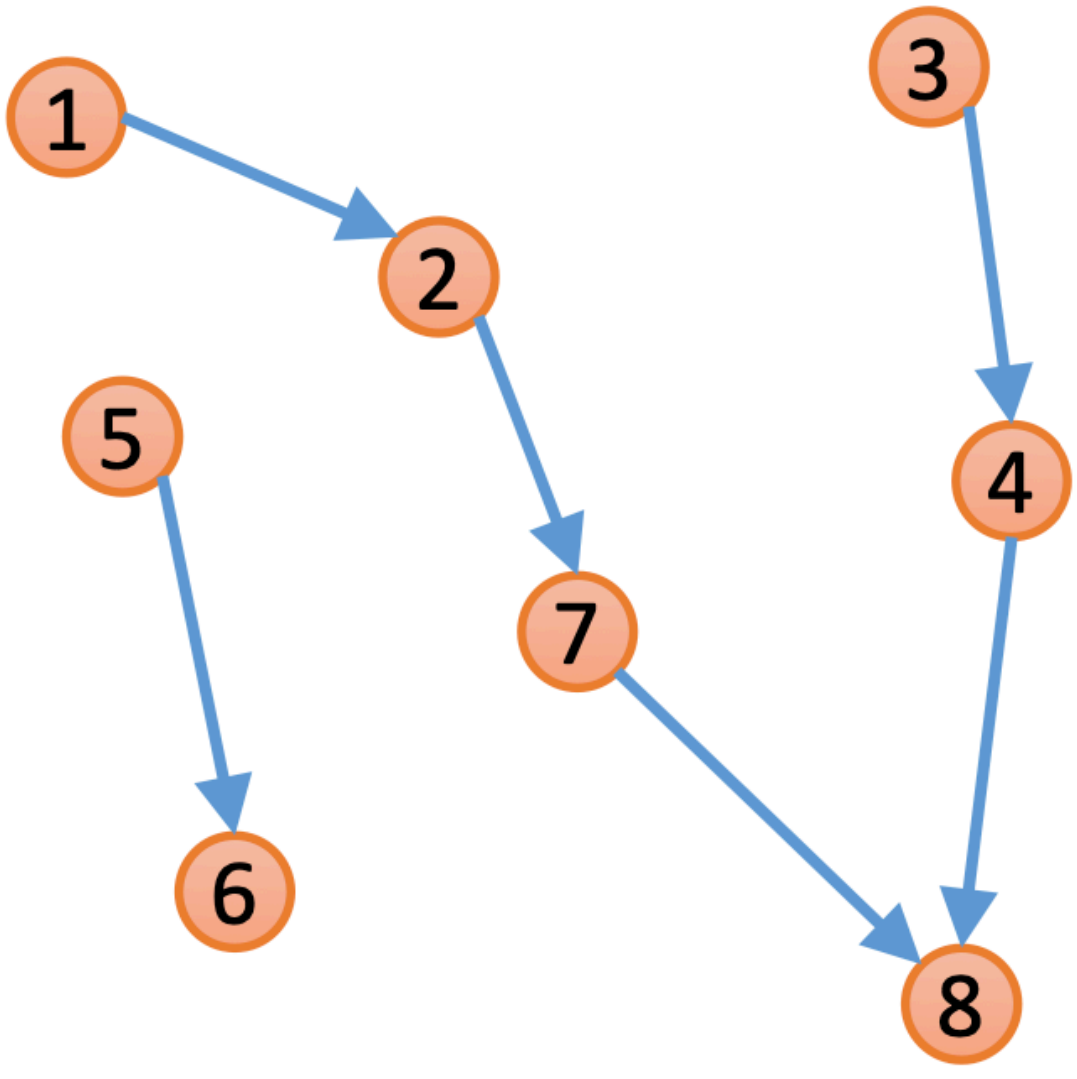
{

- while (G contains a vertex) {
    1. Pick a vertex v with in-degree = 0 ;
    2. Remove v and all its outgoing edges form the remaining graph;
    // That is we set G as G - {v}.
    3. Output v ;
  }

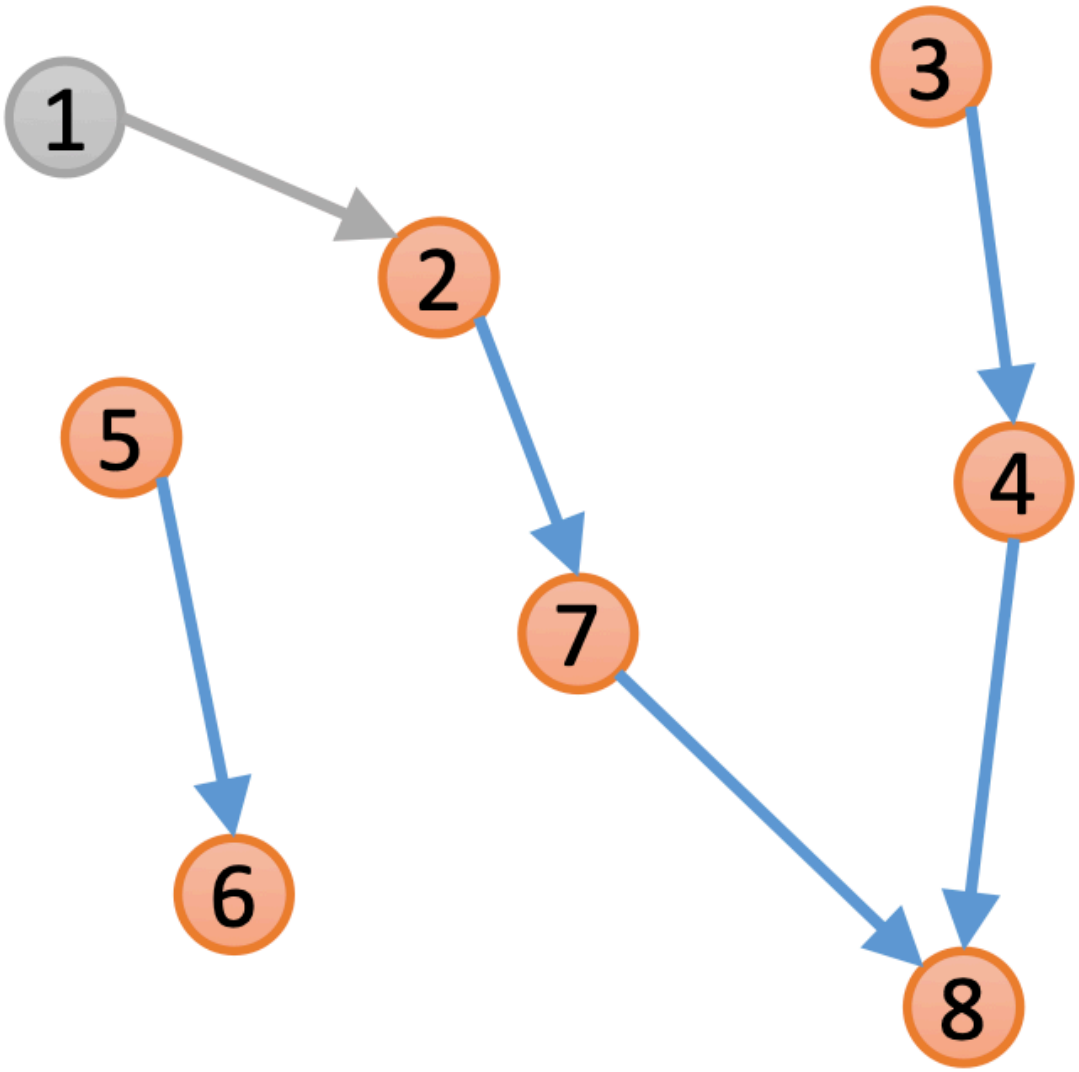
}

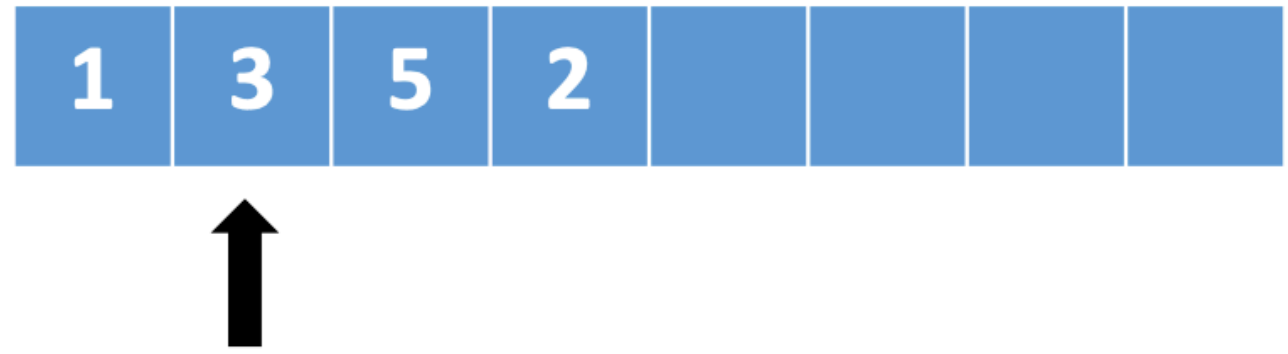
Why is the algorithm correct?

Example of Topological Sort (with Queue)

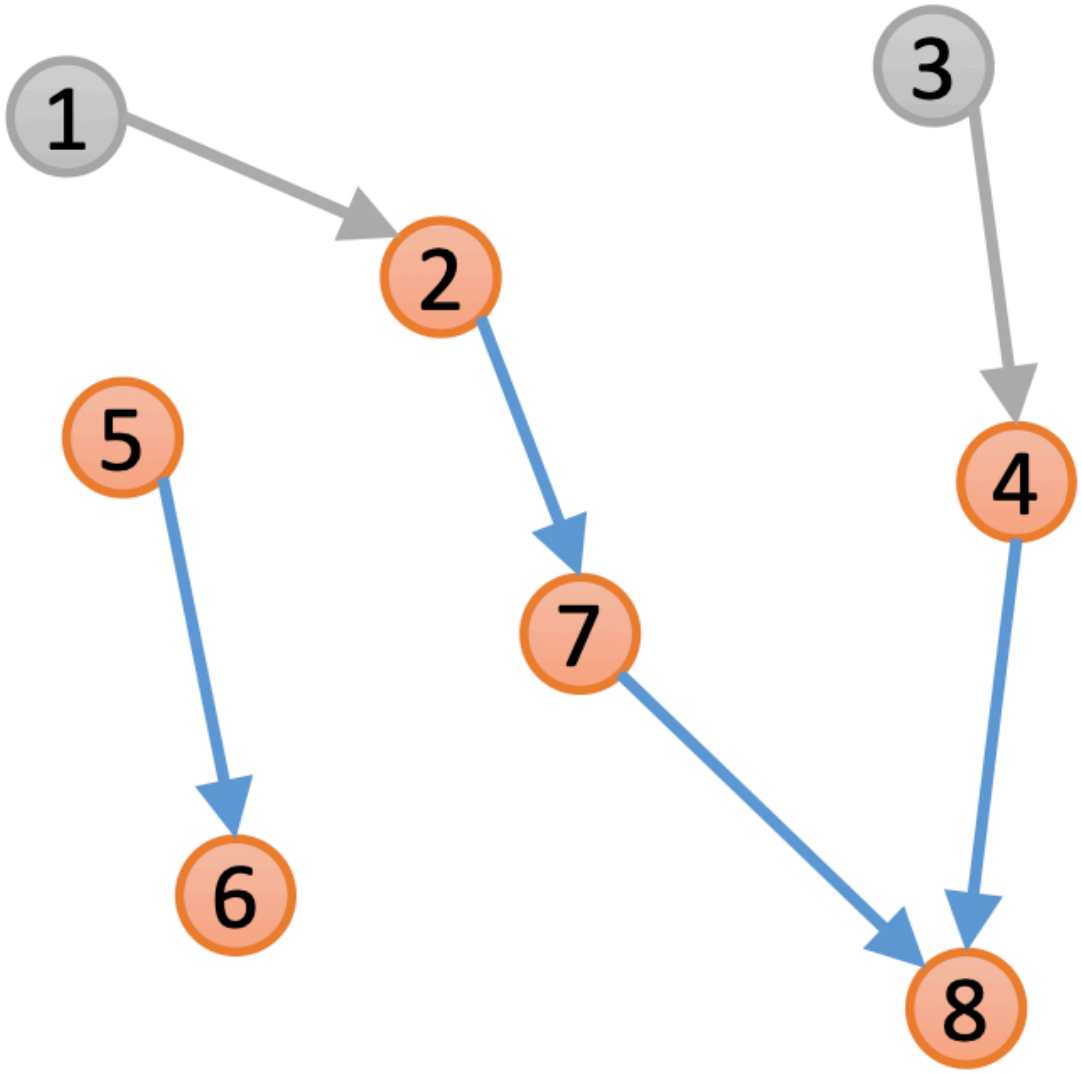


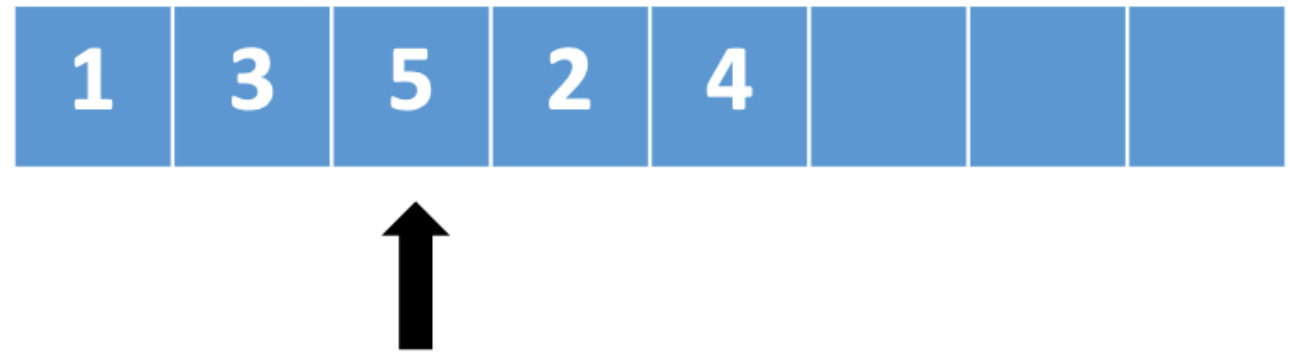
Example of Topological Sort (with Queue)



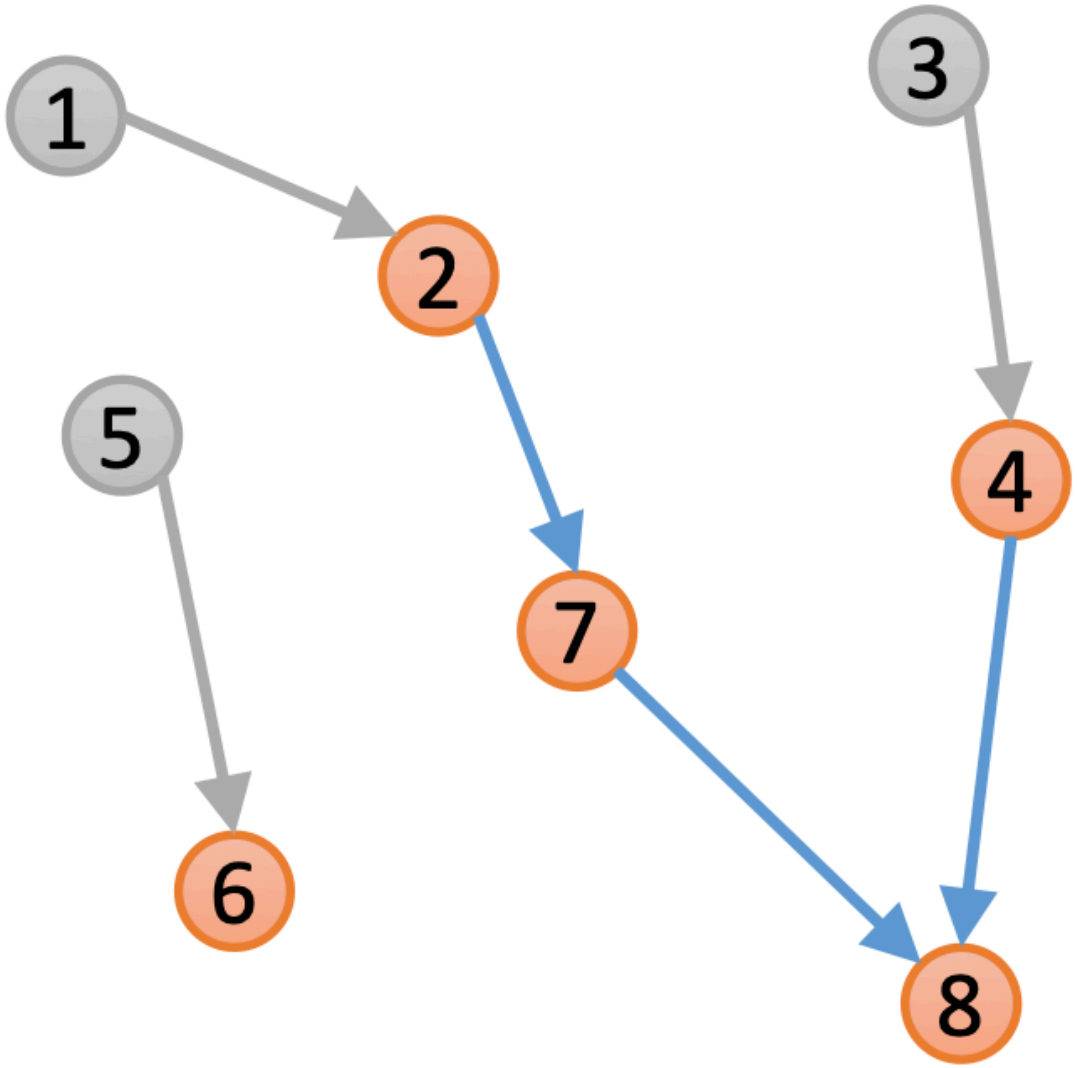


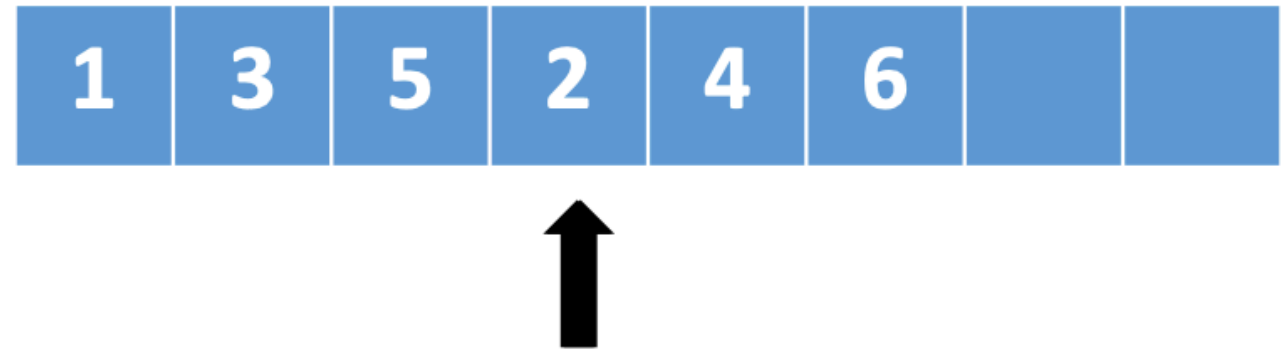
Example of Topological Sort (with Queue)



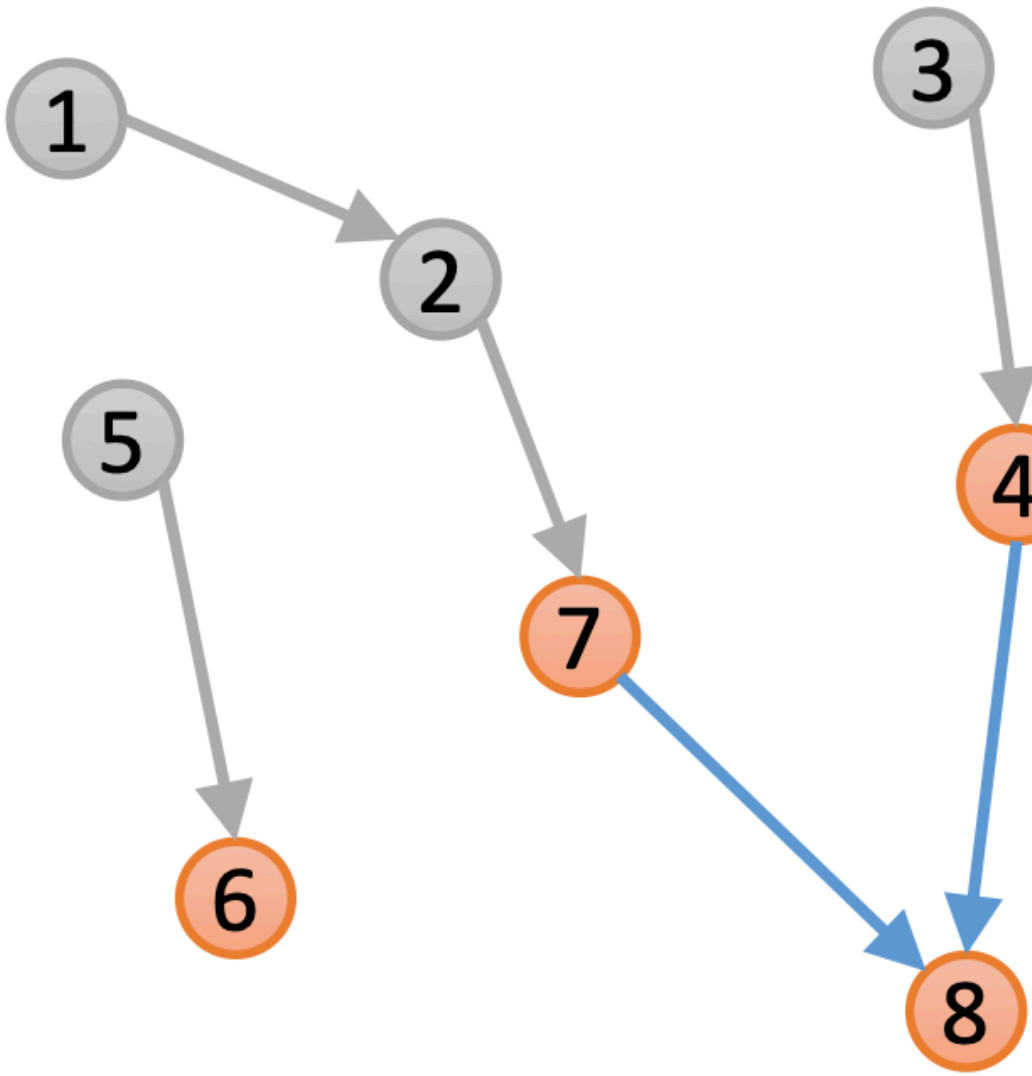


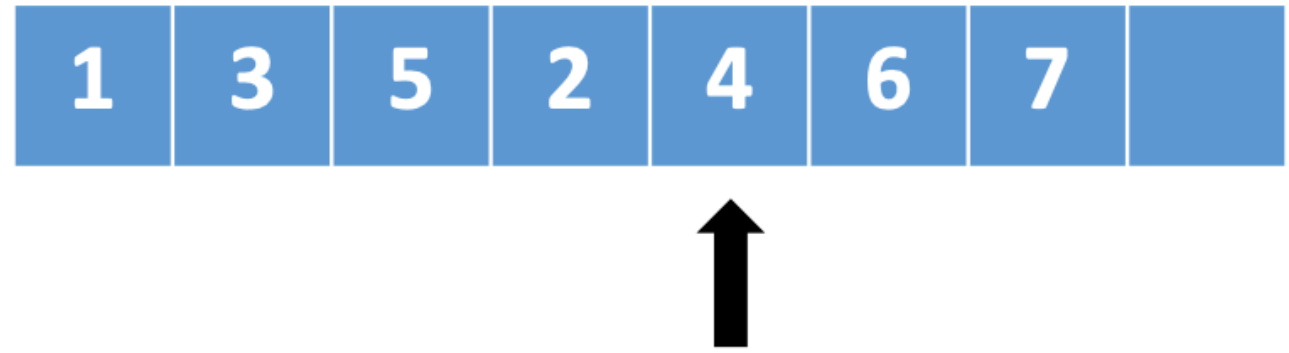
Example of Topological Sort (with Queue)



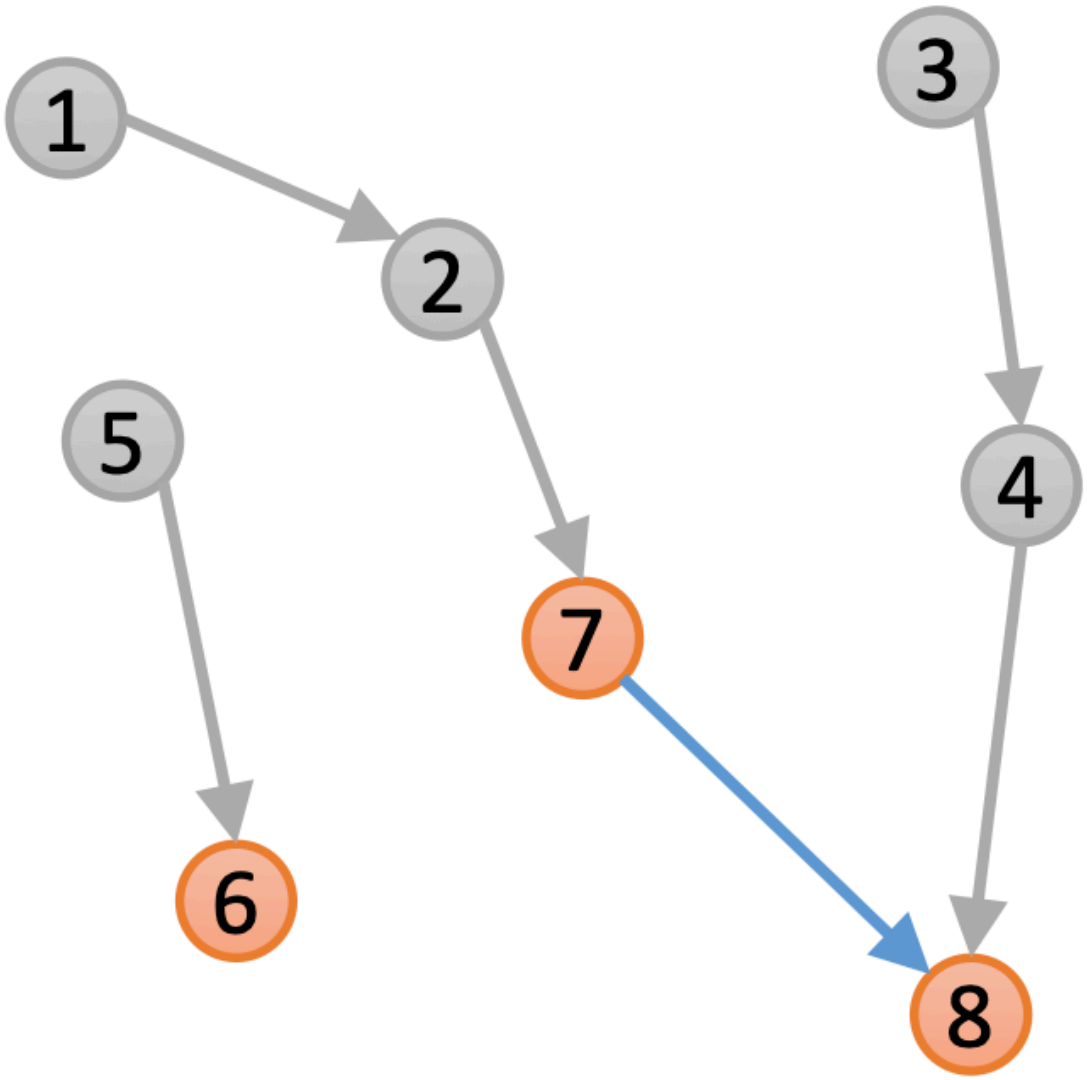


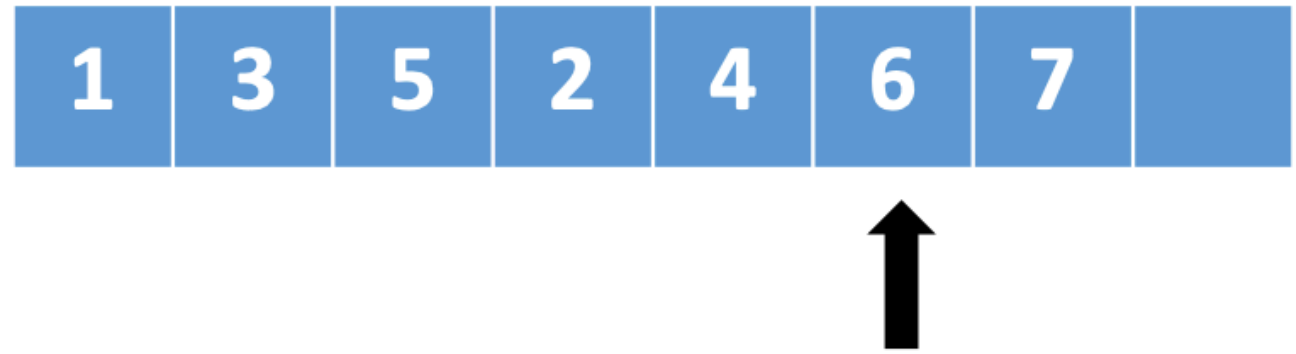
Example of Topological Sort (with Queue)



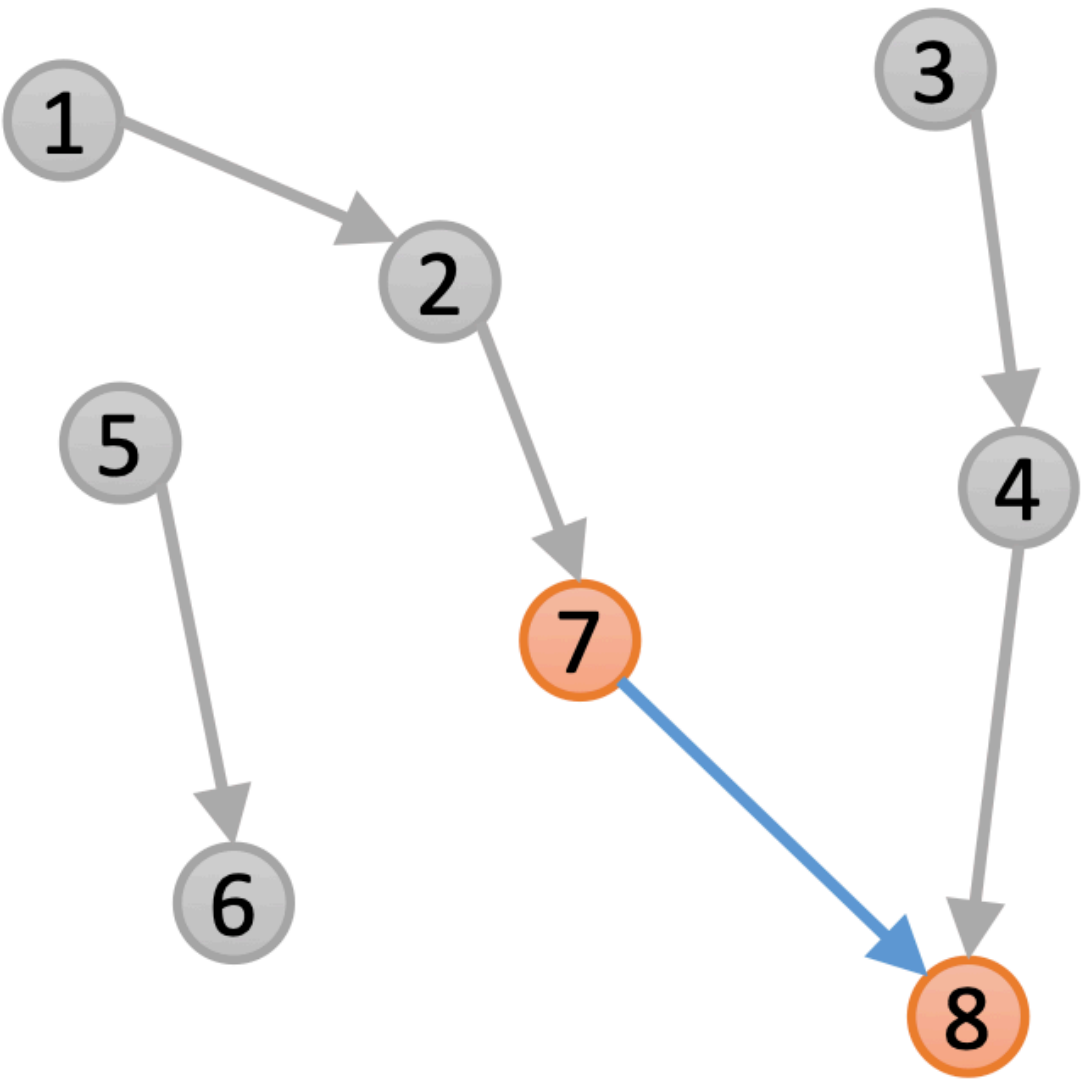


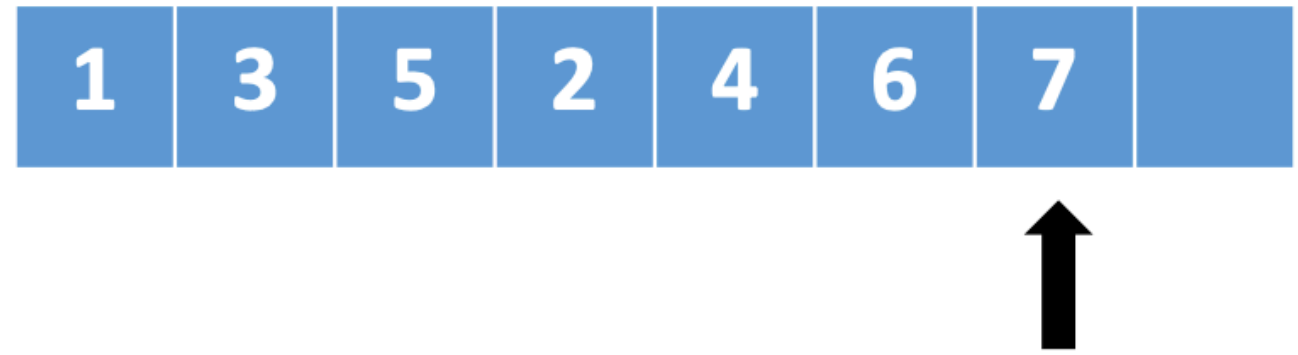
Example of Topological Sort (with Queue)



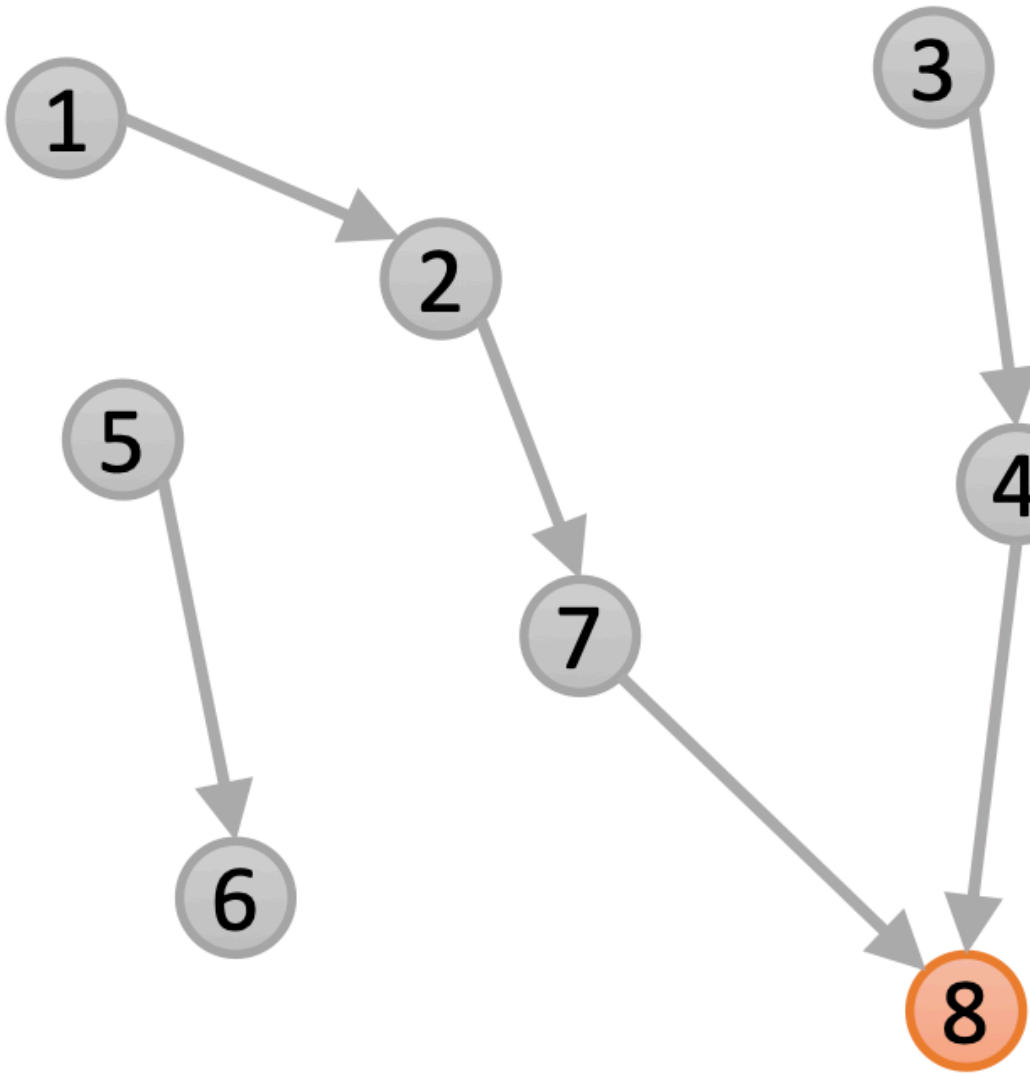


Example of Topological Sort (with Queue)

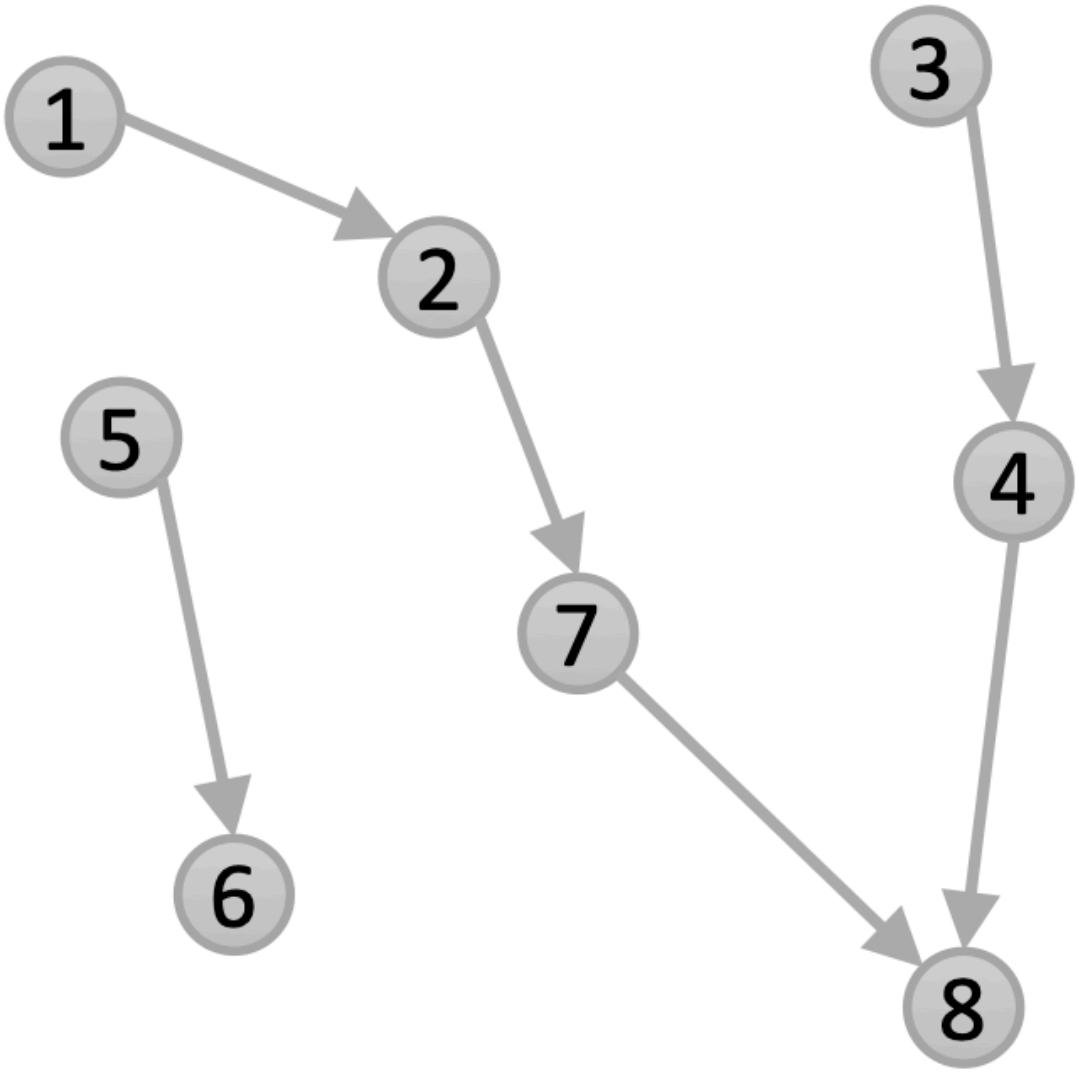




Example of Topological Sort (with Queue)



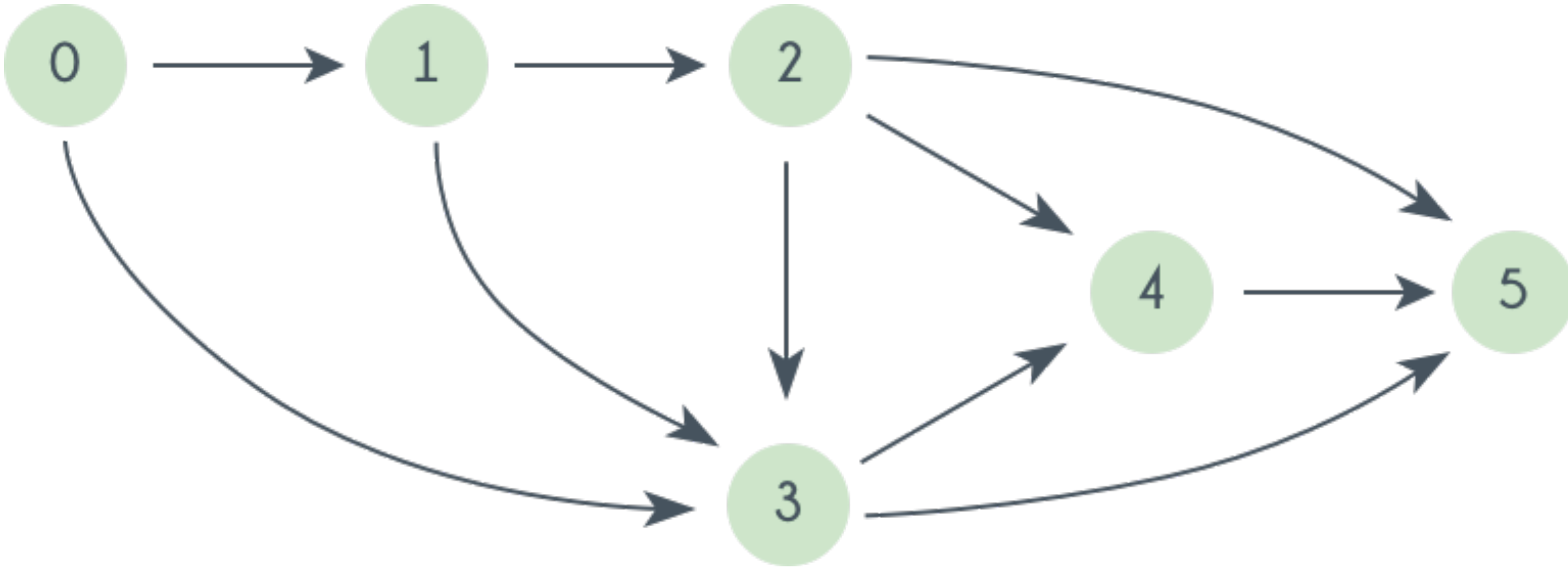
Example of Topological Sort (with Queue)



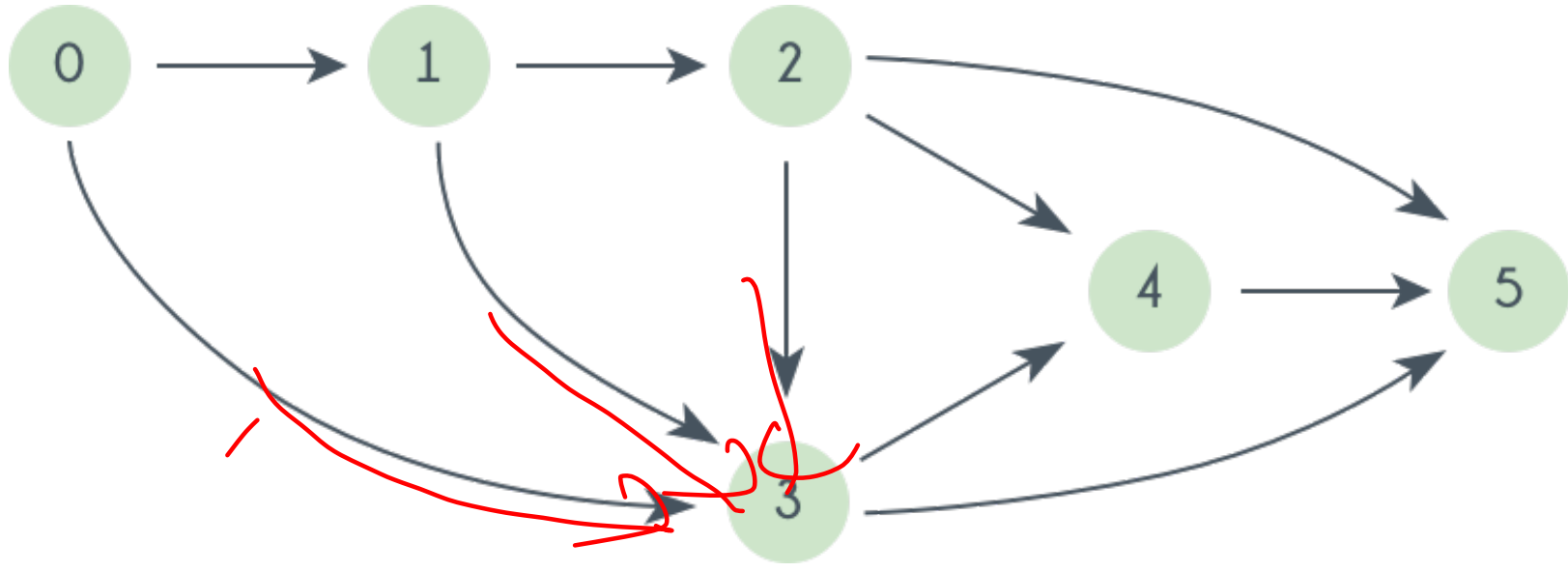
1	3	5	2	4	6	7	8

Example of Topological Sort - 2

- Let's take a graph and see the algorithm in action. Consider the graph given below:



Example of Topological Sort - 2



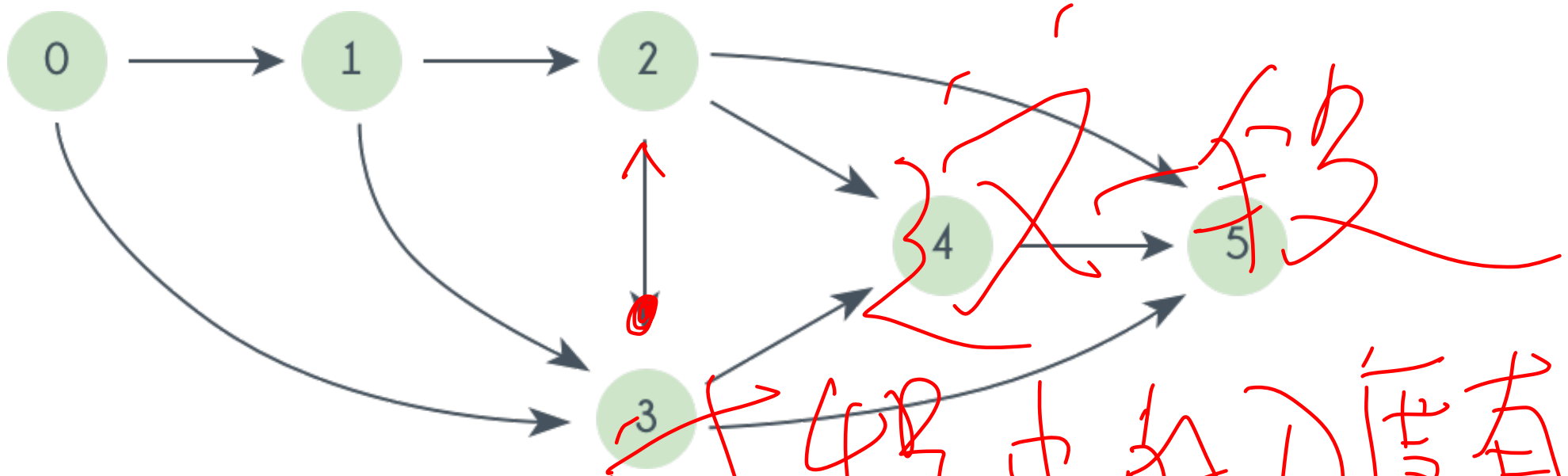
- Initially in_degree[0]=0 and T is empty

QUEUE : 0

in_degree	0	1	2	2	2	3
	0	*1*	*2*	*3*	*4*	*5*

T : --

Example of Topological Sort - 2



- So, we delete 0 from Queue and append it to T.
- The vertices directly connected to 0 are 1 and 2 so we decrease their in_degree[] by 1.
- So, now in_degree[1]=0 and so 1 is pushed in Queue.

QUEUE : 1

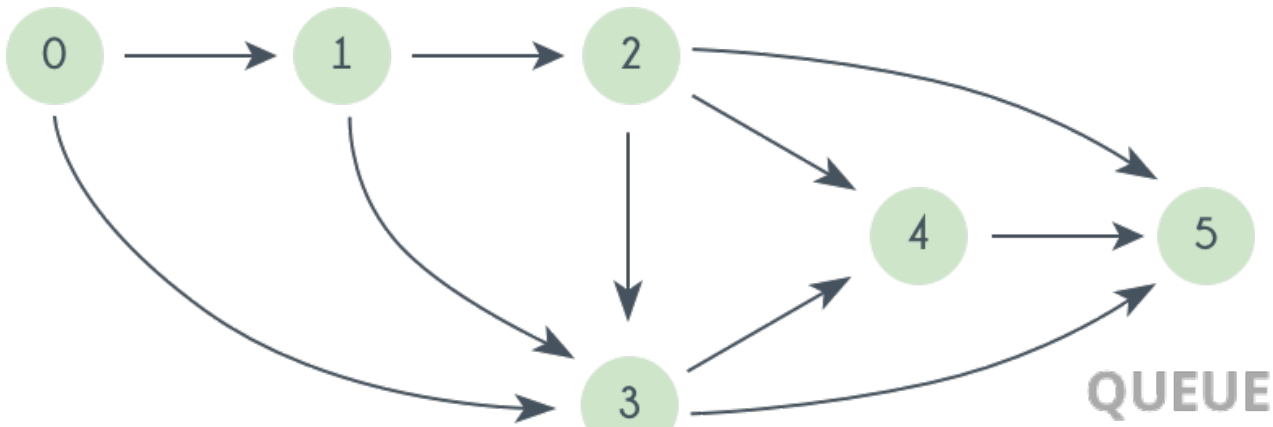
in_degree

0	0	1	1	2	3
0	*1*	*2*	*3*	*4*	*5*

T : 0

Example of Topological Sort - 2

- So, we continue doing like this, and further iterations looks like as follows:



QUEUE : 3

in_degree	0	0	0	0	1	2
	0	1	2	3	4	5

T : 0 1 2

这一段应该没有错误

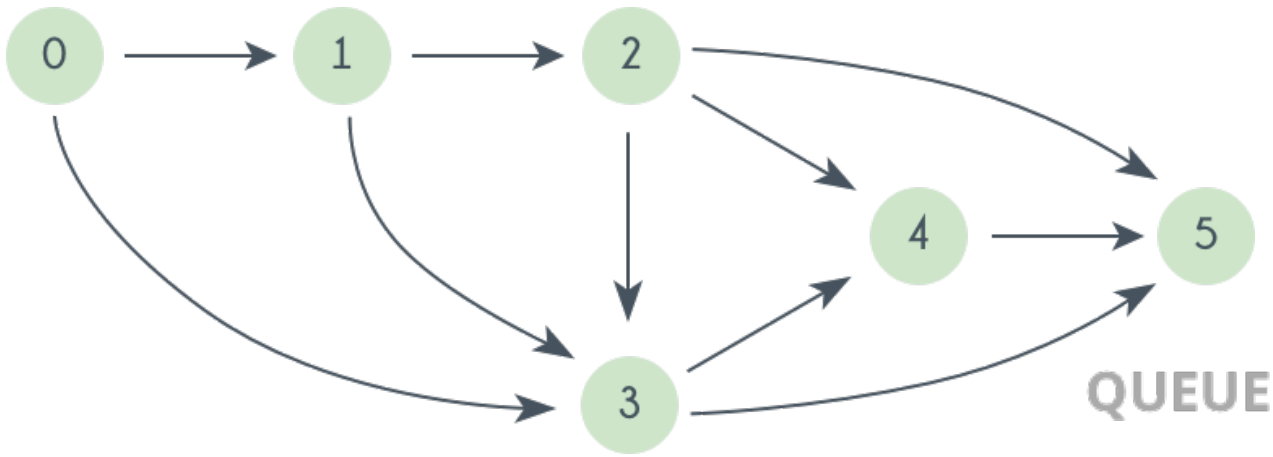
QUEUE : 4

in_degree	0	0	0	0	0	1
	0	1	2	3	4	5

T : 0 1 2 3

Example of Topological Sort - 2

- So, we continue doing like this, and further iterations looks like as follows:



QUEUE : 5

in_degree

0	0	0	0	0	0
0	*1*	*2*	*3*	*4*	*5*

T : 0 1 2 3 4

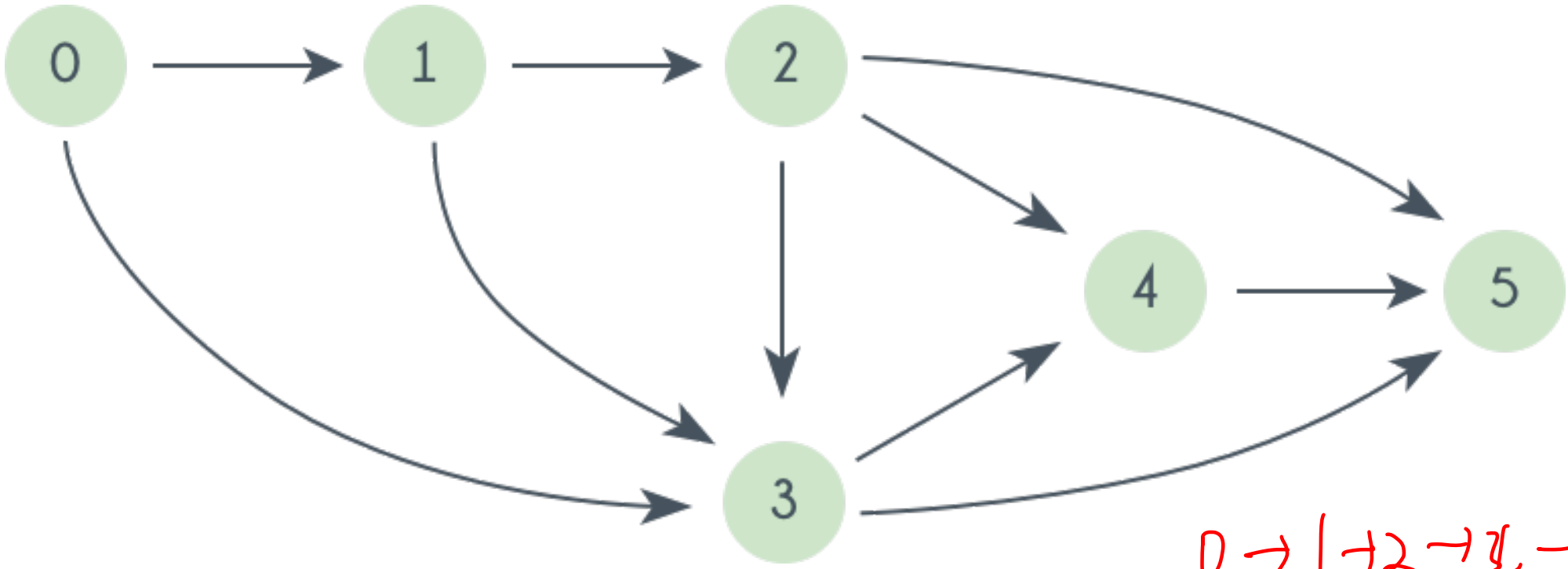
QUEUE : --

in_degree

0	0	0	0	0	0
0	*1*	*2*	*3*	*4*	*5*

T : 0 1 2 3 4 5

Example of Topological Sort - 2



0 → 1 → 2 → 4 → 5 → 3

- So at last we get our Topological sorting in T i.e. : **0, 1, 2, 3, 4, 5**

QUEUE : --

in_degree

0	0	0	0	0	0
0	*1*	*2*	*3*	*4*	*5*

T : 0 1 2 3 4 5

Topological Sort (with Queue)

- Theorem:
  - If G is acyclic, the previous algorithm produces a topological sort of G

如果G不含环，前面的操作会产生一个次序。

- Proof:
- Two cases may happen when we run the previous algorithm.
  - Case 1 : All vertices are output

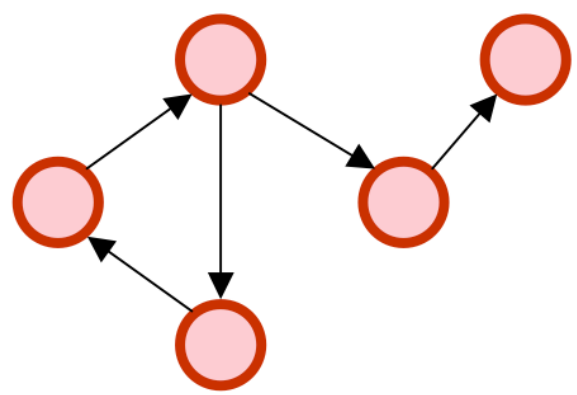
输出的有顶点。

  - Case 2 : Some vertex may not be output

一些顶点无法输出。

If G is acyclic, the previous algorithm produces a topological sort of G

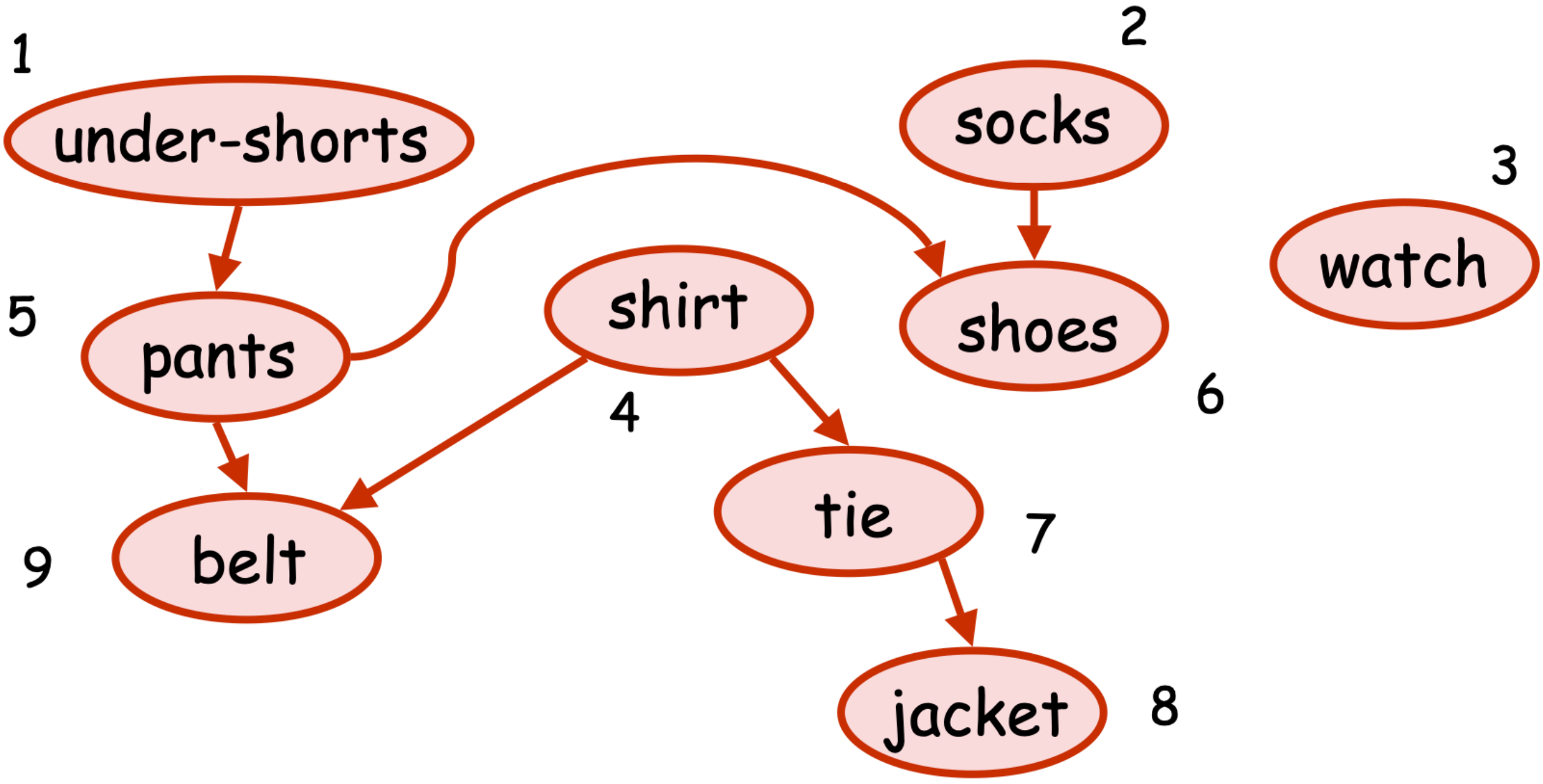
- Proof:
  - Two cases may happen when we run the previous algorithm.
    - Case 1 : All vertices are output
      - vertices are sorted correctly
        - (According to the the output strategy of this algorithm)
    - Case 2 : Some vertex may not be output
      - the remaining vertices must each have in-degree ≥ 1.
      - Now, we pick a vertex v in this group, repeatedly visit another vertex by tracing an incoming edge, some vertex will be visited twice (why?) ➔ a cycle is found !!



Performance

- Let G = (V,E) be the input directed graph •Running time for Topological-Sort with Queue :
  - 1. Each vertex keeps # incoming edges
  - 2. Finding vertices with in-degree = 0 :
  - Naive method: $O(|V|^2)$ total time
    - You check this by linear search
    - $|V| + (|V|-1) + \ldots + 1 = O(|V|^2)$
- Clever method: (use a queue Q)
- Enqueue vertex once its in-degree = 0
- Total time: O(|V|+|E|)

Topological Sort (Example)



When a vertex is output, its indegree is 0

Topological Sort (with Stack)

Topological-Sort(G) // given G is acyclic

{

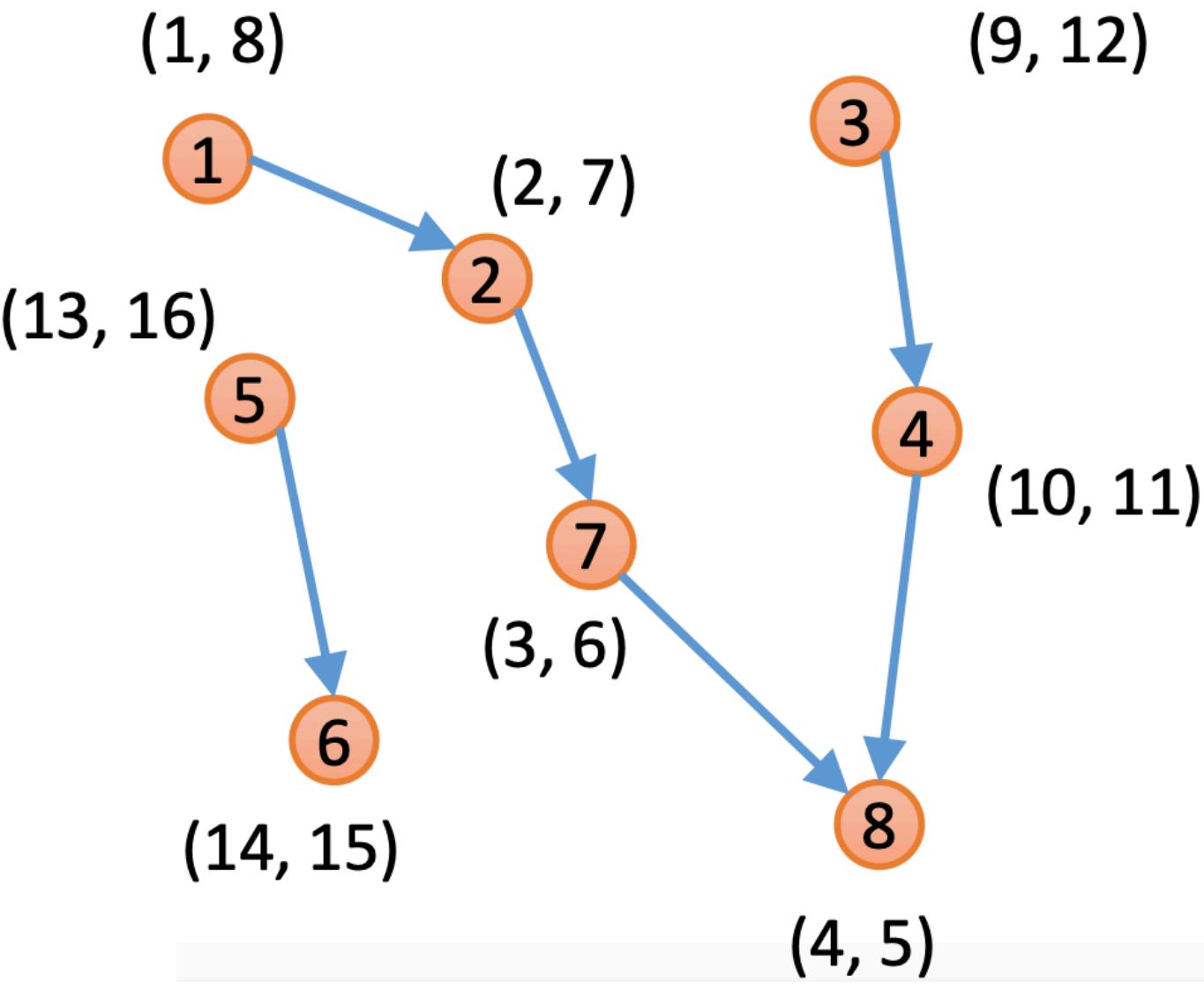
1. Call DFS on G
2. Output vertices in decreasing order of their finishing times ;

}

Why is the algorithm correct?

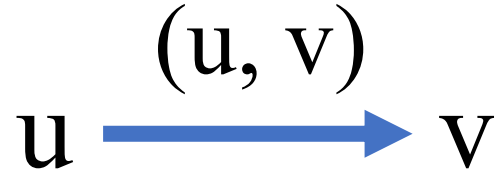
Topological Sort (with Stack)

- (check time, finishing time)



Topological Sort (with Stack)

- Theorem:
  - If G is acyclic, the previous algorithm produces a topological sort of G
- Proof:
  - Let (u, v) be an edge.



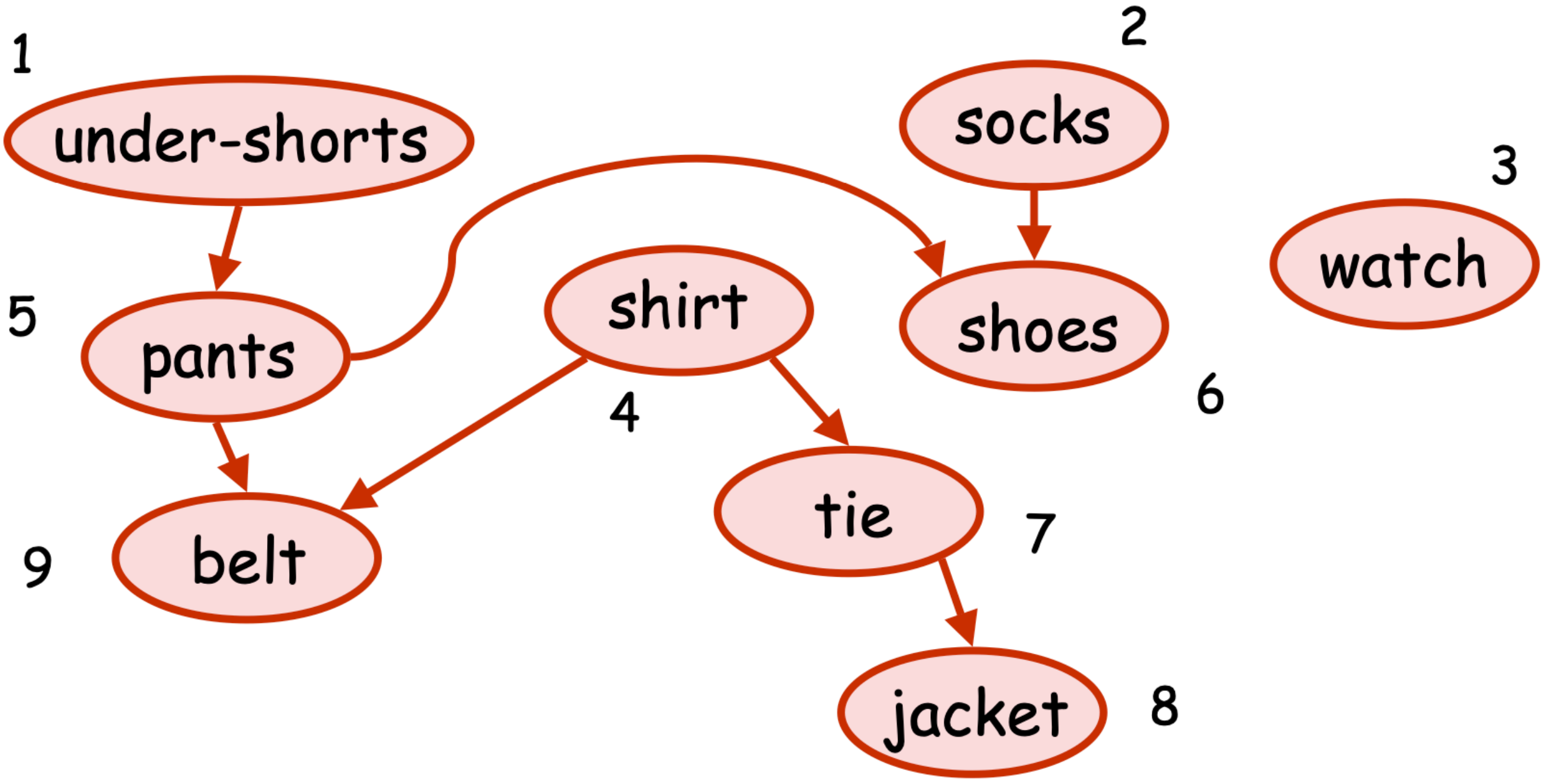
  - We shall show f(v) < f(u) so that the ordering is correct.
    - f(v) is the finish time of node v 完成时间.
  - Firstly, during DFS, there are two cases
    - Case 1 : u is visited before v
    - Case 2 : v is visited before u

u 先.
v 先.
两种可能性.

If G is acyclic, the previous algorithm produces a topological sort of G

- Case 1 : u is visited before v
  - u cannot finish before DFS is performed on all its neighbors.
  - Since v is a neighbor of u, we must have $d(u) < d(v) < f(v) < f(u)$
- Case 2 : v is visited before u
  - v must have finished before u starts (else, there will be a path from v to u and the graph contains a cycle.)
  - Thus, $f(v) < d(u) \rightarrow f(v) < f(u)$
- Both cases show $f(v) < f(u)$
- Done!

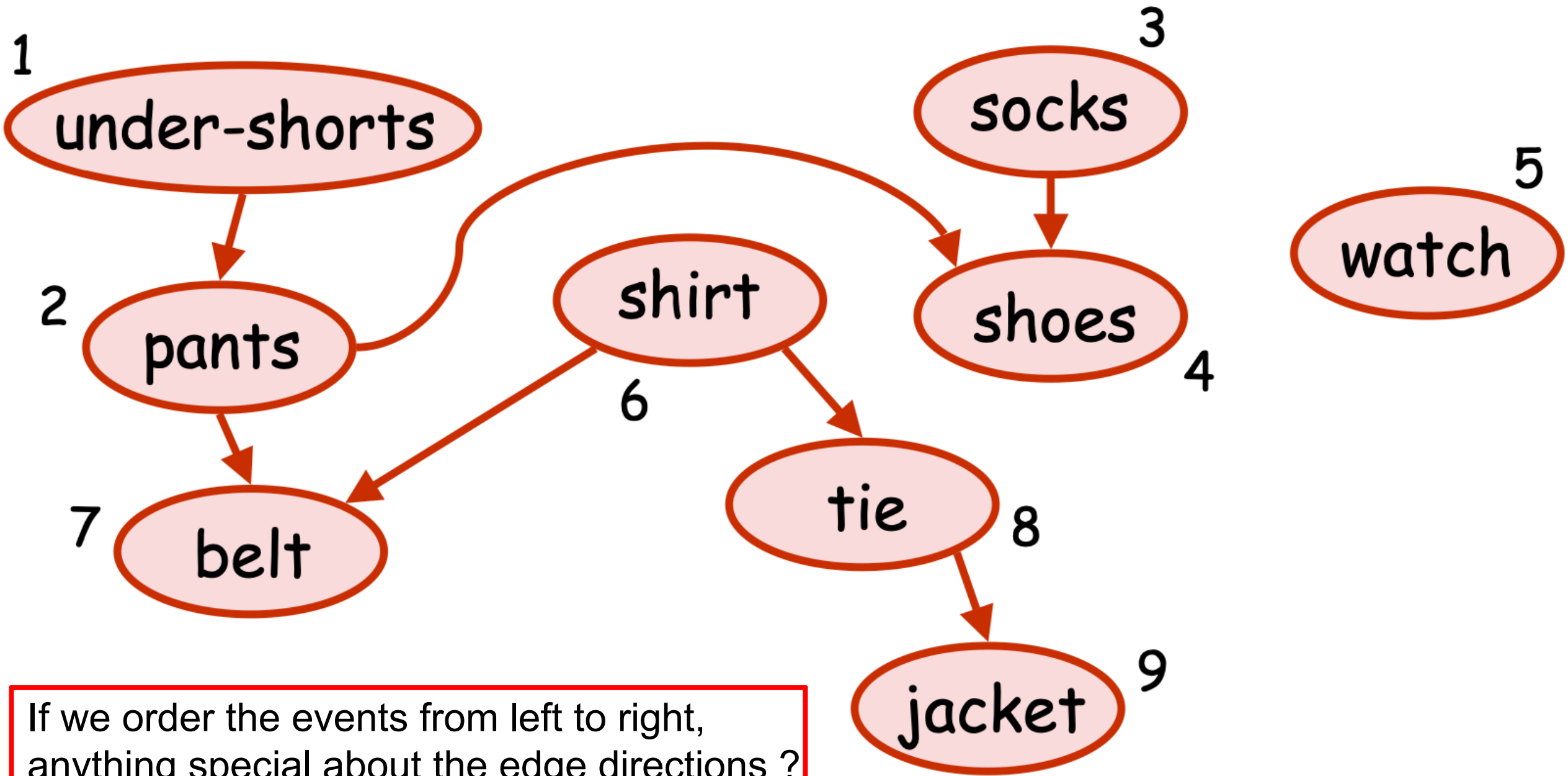
Topological Sort (Example)



Discovery and Finishing Times after a possible DFS

Topological Sort (Example)

- Ordering Finishing Times (in descending order)

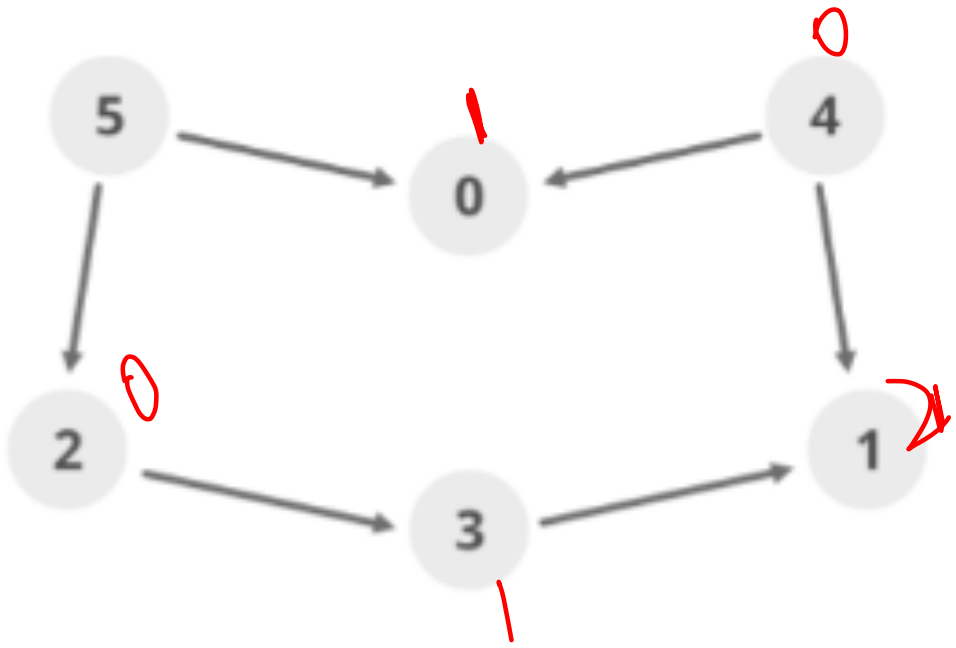


If we order the events from left to right, anything special about the edge directions ?

Performance

- Let G = (V,E) be the input directed graph
- Running time for Topological-Sort :
  - 1. Perform DFS : O(|V|+|E|) time
  - 2. Sort finishing times
    - Naive method: O(|V| log |V|) time
    - Clever method: (use an extra stack S)
      - During DFS, push a node into stack S once finished → no need to sort !!
- Total time: O(|V|+|E|)

Example



Adja cent list (G)

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

拓扑 临接列表
如果没有邻接
元素 / 邻接元素
已为
的优先放
入stack.

0 1 5 2 4 3

0 1 4 3 2 5

Step 1: Topological Sort(0), visited[0] = true

List is empty. No more recursion call.

Stack: 0

	0	1	2	3	4	5
visited	TRUE	false	false	false	false	false

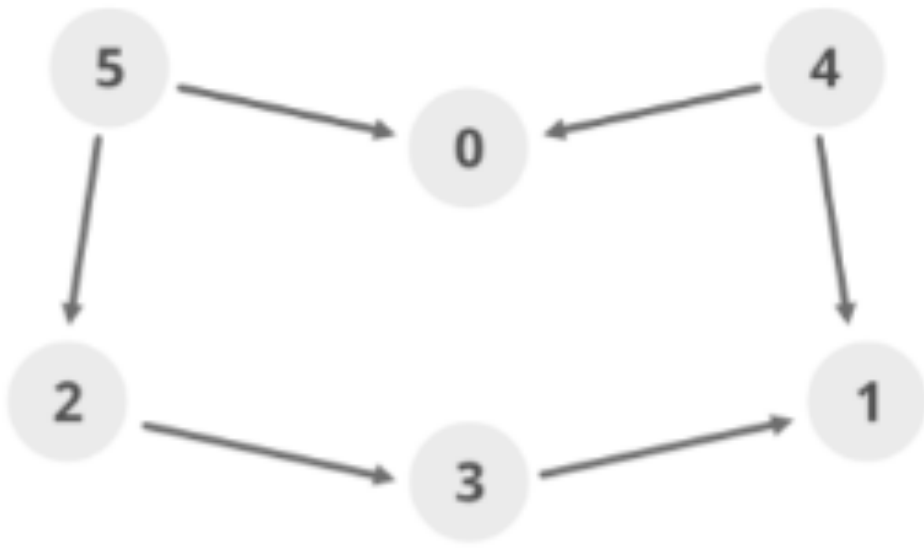
Step 2: Topological Sort(1), visited[1] = true

List is empty. No more recursion call.

Stack: 0 1

	0	1	2	3	4	5
visited	TRUE	TRUE	false	false	false	false

Example



Adja cent list (G)

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Step 3: Topological Sort(2), visited[2] = true

↓

Topological Sort(3), visited[3] = true

↓

'1' is already visited. No more recurrsion

Stack

0	1	3	2

	0	1	2	3	4	5
visited	TRUE	TRUE	TRUE	TRUE	false	false

Step 4: Topological Sort(4), visited[4] = true

↓

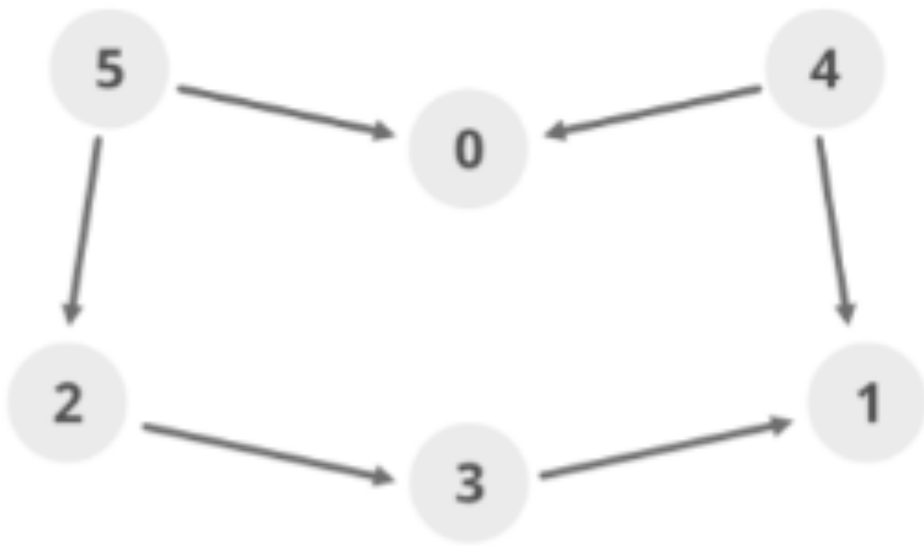
'0' , '1' are already visited. No more recurrsion call

Stack

0	1	3	2	4

	0	1	2	3	4	5
visited	TRUE	TRUE	TRUE	TRUE	TRUE	false

Example



Adja cent list (G)

0 →
1 →
2 → 3
3 → 1
4 → 0, 1
5 → 2, 0

	0	1	2	3	4	5
visited	false	false	false	false	false	false

Stack(empty)

Step 5: Topological Sort(5), visited[5] = true

'2' , '0' are already visited. No more recurrsion call

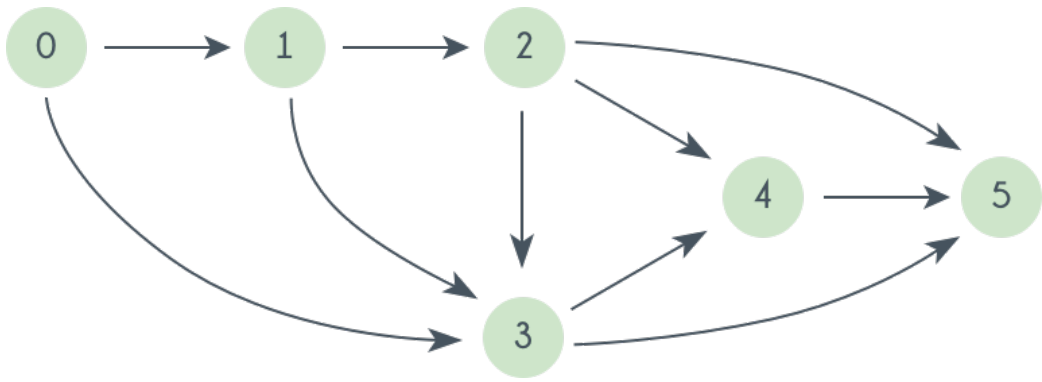
Stack	0	1	3	2	4	5

	0	1	2	3	4	5
visited	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE

Step 6: Print all elements of stack from top to bottom

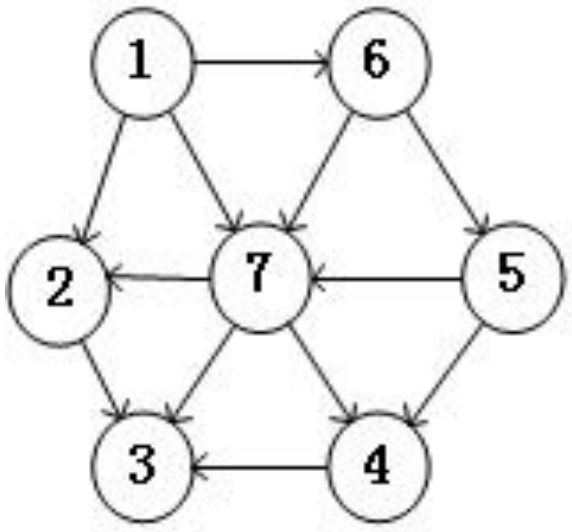
Output: 5, 4, 2, 3, 1, 0

Example 2



Stack	T
0	--
0 1	--
0 1 2	--
0 1 2 4	--
0 1 2 4 5	--
0 1 2 4	5
0 1 2	*4 5*
0 1 2 3	*4 5*
0 1 2	*3 4 5*
0 1	*2 3 4 5*
0	*1 2 3 4 5*
--	*0 1 2 3 4 5*

Example 3



Vertex	Indegree						
1	0	0	0	0	0	0	0
2	2	1	1	1	0	0	0
3	3	3	3	3	2	1	0
4	2	2	2	1	0	0	0
5	1	1	0	0	0	0	0
6	1	0	0	0	0	0	0
7	3	2	1	0	0	0	0

Enqueue	1	6	5	7	4, 2		3

```
// A Java program to print topological
// sorting of a DAG
import java.io.*;
import java.util.*;

// This class represents a directed graph using adjacency list representation
class Graph {
    private int V; // No. of vertices

    // Adjacency List as ArrayList of ArrayList's
    private ArrayList<ArrayList<Integer> > adj;

// Constructor
Graph(int v)
{
    V = v;
    adj = new ArrayList<ArrayList<Integer> >(v);
    for (int i = 0; i < v; ++i)
        adj.add(new ArrayList<Integer>());
}

// Function to add an edge into the graph
void addEdge(int v, int w) { adj.get(v).add(w); }
```

A recursive function used by topologicalSort

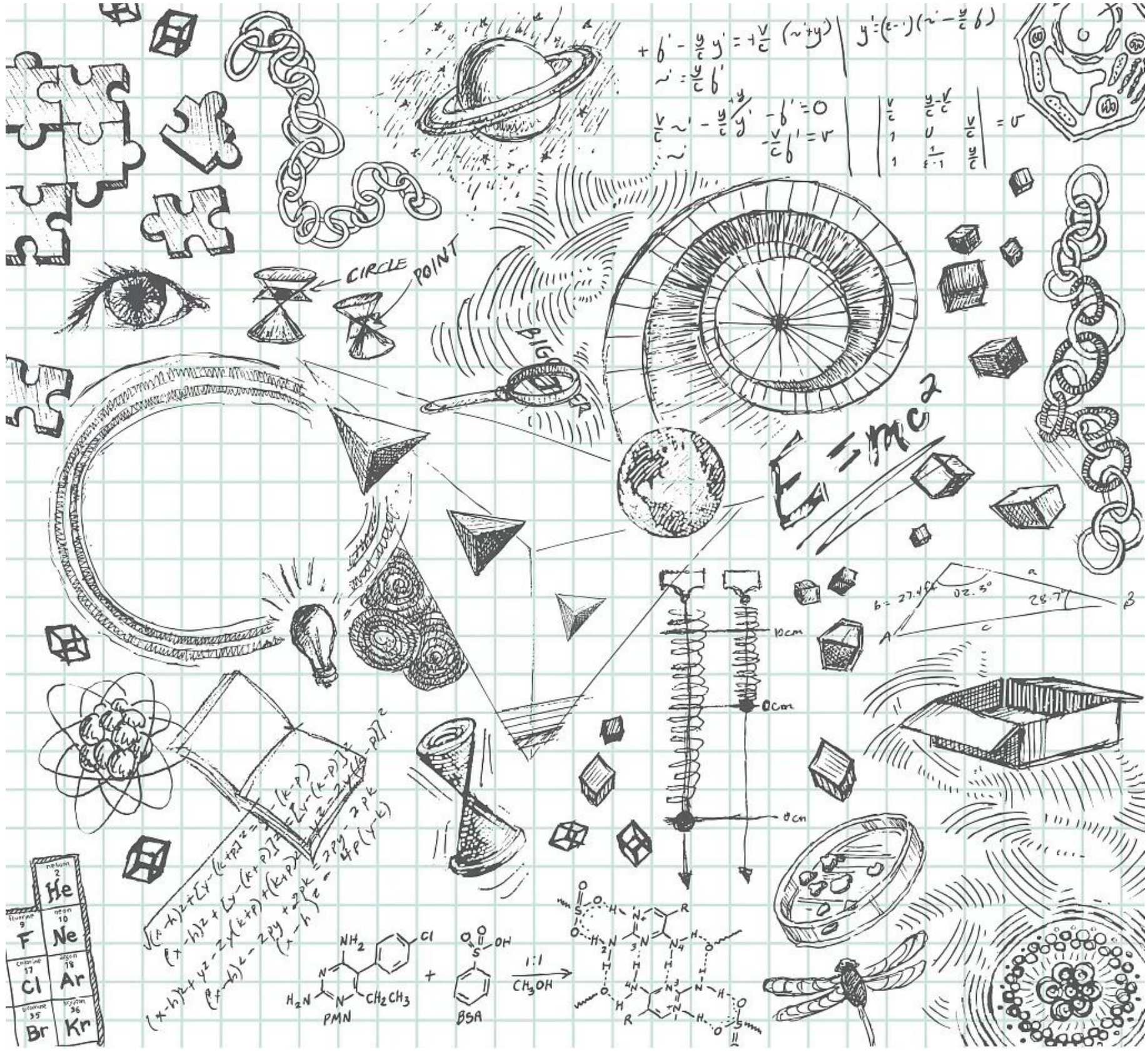
```
// A recursive function used by topologicalSort
   void topologicalSortUtil(int v, boolean visited[],
                            Stack<Integer> stack)
   {
      // Mark the current node as visited.
      visited[v] = true;
      Integer i;

      // Recur for all the vertices adjacent to this vertex
      Iterator<Integer> it = adj.get(v).iterator();
      while (it.hasNext()) {
         i = it.next();
         if (!visited[i])
            topologicalSortUtil(i, visited, stack);
      }

      // Push current vertex to stack which stores result
      stack.push(new Integer(v));
}
```

Topological Sort

```
// The function to do Topological Sort.
   // It uses recursive topologicalSortUtil()
   void topologicalSort()
   {
      Stack<Integer> stack = new Stack<Integer>();

      // Mark all the vertices as not visited
      boolean visited[] = new boolean[V];
      for (int i = 0; i < V; i++)
         visited[i] = false;

      // Call the recursive helper function to store
      // Topological Sort starting from all vertices one by one
      for (int i = 0; i < V; i++)
         if (visited[i] == false)
            topologicalSortUtil(i, visited, stack);

      // Print contents of stack
      while (stack.empty() == false)
         System.out.print(stack.pop() + " ");
}
```

Driver code

```
// Driver code
    public static void main(String args[])
    {
        // Create a graph given in the above diagram
        Graph g = new Graph(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);

        System.out.println("Following is a Topological "
                           + "sort of the given graph");
        // Function Call
        g.topologicalSort();
    }
}
```


CIRCLE
POINT
BIG
E=mc2
10cm
0cm
8cm
PMN
BSA
1:1
CH3OH