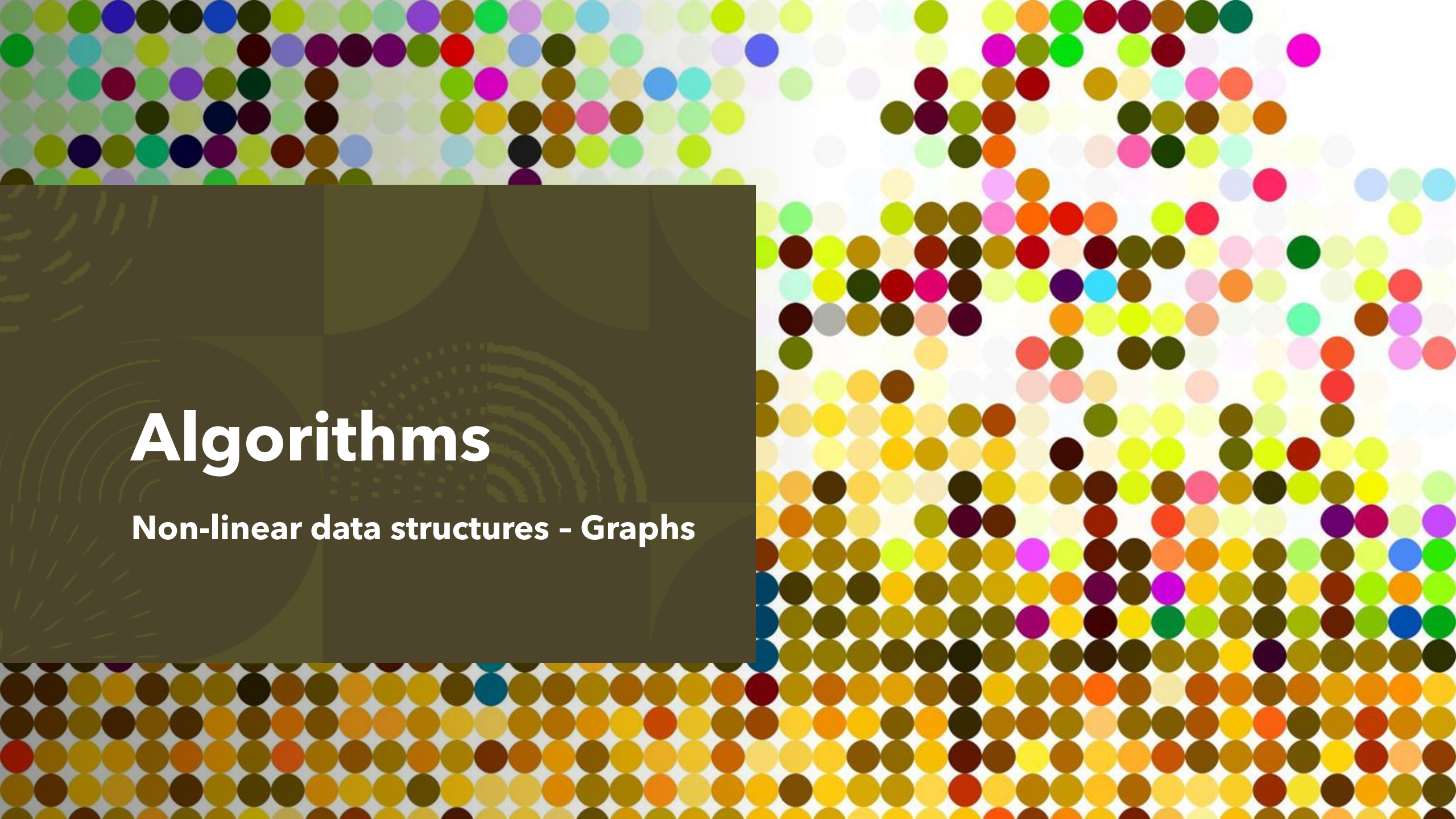
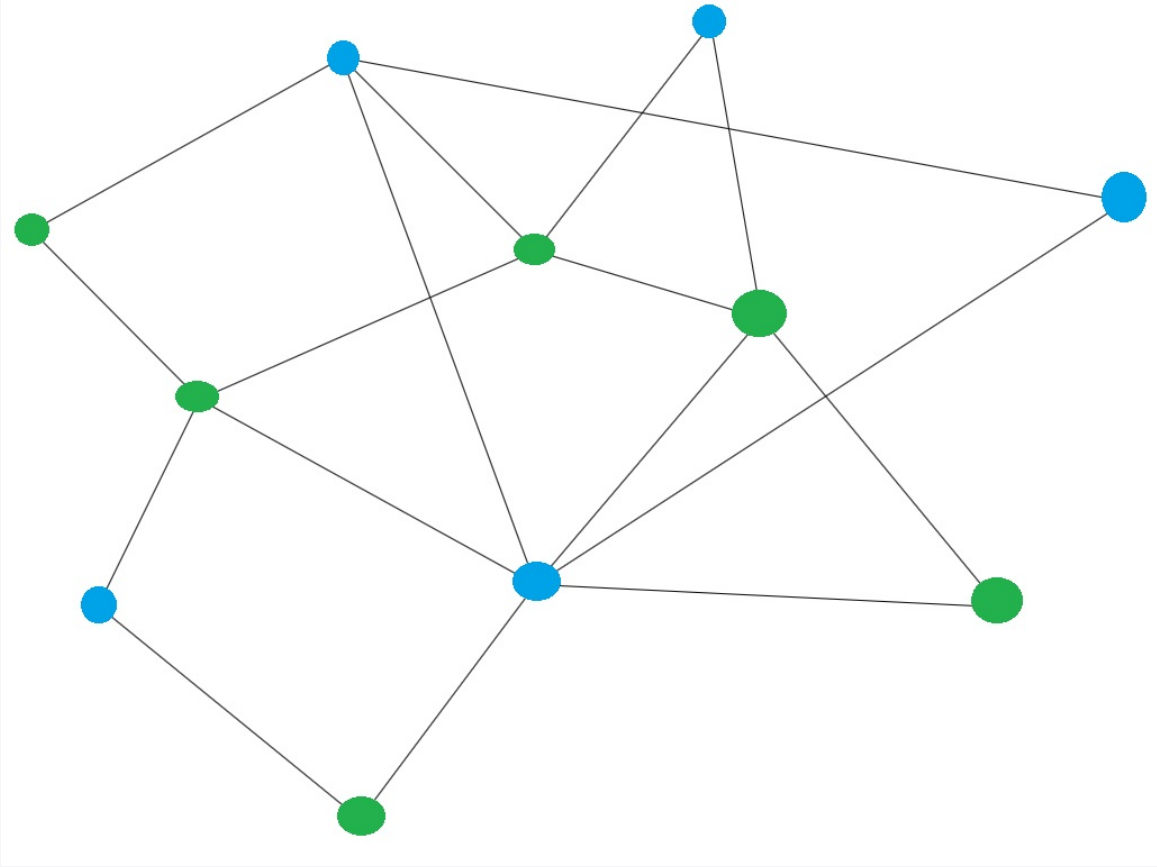


Algorithms

Non-linear data structures – Graphs

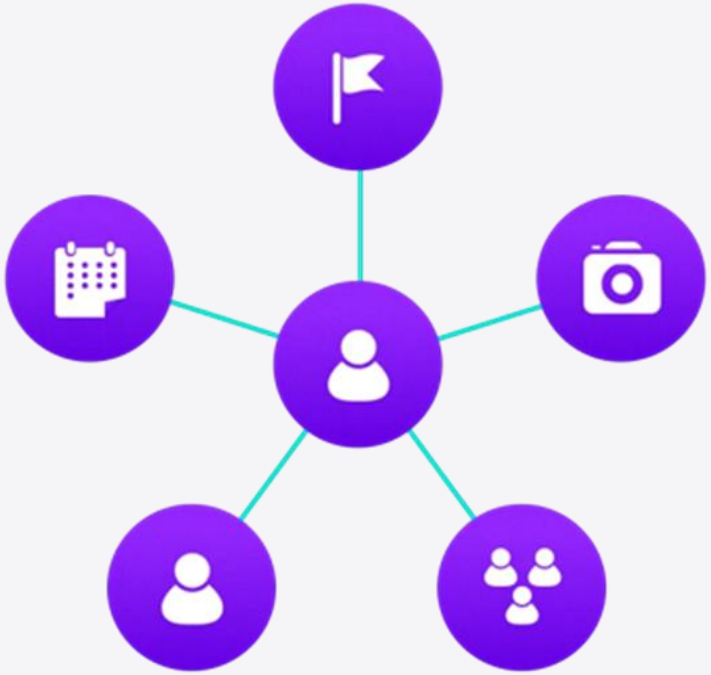
Introduction

In **computer science**, a graph data structure is a collection of nodes that have **data** and are connected to other nodes.



On Facebook, everything is a **node**, i.e., User, Photo, Album, Event, Group, Page, Comment, Story, Video, Link, Note...**anything with data is a node**.

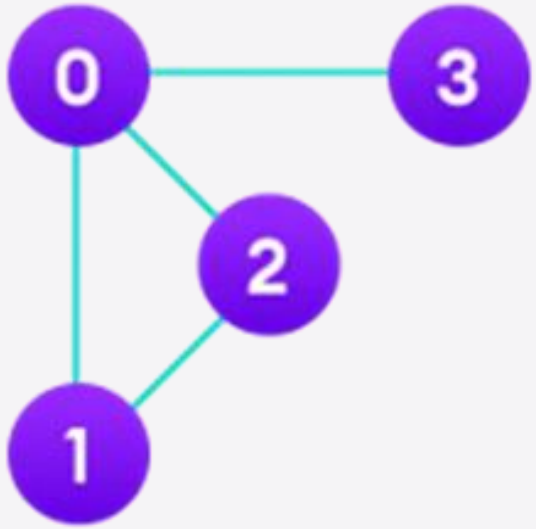
Every relationship is an **edge** from one node to another. Whether you post a photo, join a group, like a page, etc., a new edge is created for that relationship.



The **social networks** are a collection of these **nodes** and **edges**. In the case of **Facebook**, it uses a **graph data structure** to store its data.

More precisely, a graph is a data structure **(V, E)** that consists of

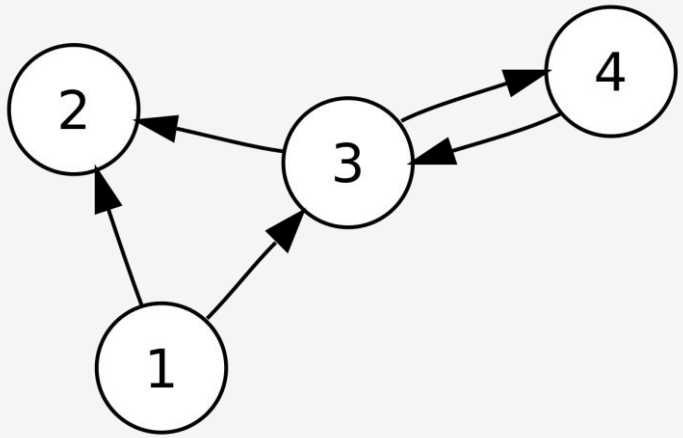
- A collection of vertices **V**
- A collection of edges **E**, represented as ordered pairs of vertices **(u,v)**

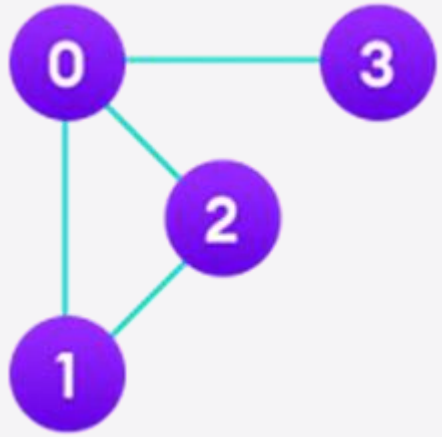


In the graph,

V = {**0**, **1**, **2**, **3**}
E = {**(0,1)**, **(0,2)**, **(0,3)**, **(1,2)**}
G = {**V**, **E**}

Graph Terminology





- **Adjacency:** A vertex is said to be adjacent to another vertex if there is an edge connecting them, e.g., vertices (0,1), (0,2), (0,3), (1,2).

- **Path:** A sequence of edges to go from *vertex A* to *vertex B* is called a path. 0-1, 1-2 and 0-2 are paths from vertex 0 to vertex 2.

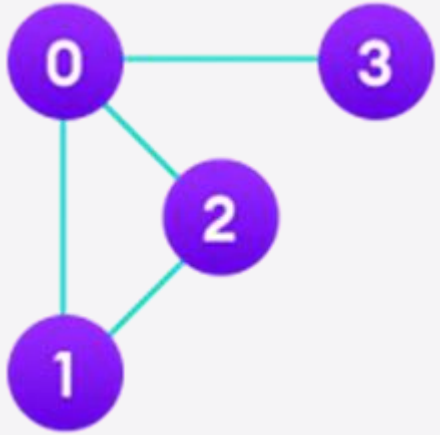
- **Directed Graph:** A graph in which an edge **(u,v)** doesn't necessarily mean that there is an edge **(v, u)** as well. The edges in such a graph are represented by arrows to show the direction of the edge.

Graph Representation

Graphs are commonly represented in two ways:

1. Adjacency Matrix

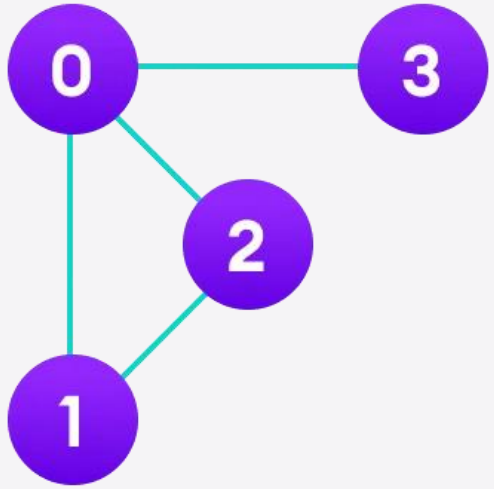
A **matrix of boolean values** (**0's** and **1's**). A finite graph can be represented in the form of a **square matrix** (**V x V**), where the **1's values** in the matrix indicates that there is a direct path between two vertices.



i → / 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

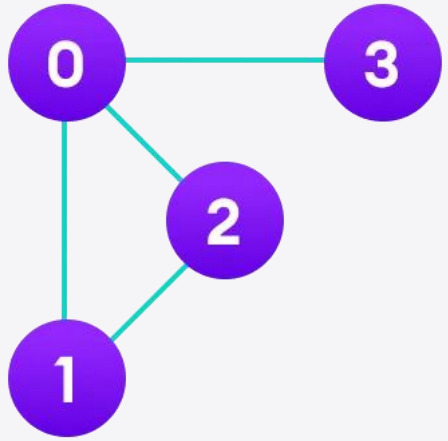
In the adjacency matrix, each row and column represent a vertex.

Each cell in the matrix, A_{ij}, where **i** and **j** are **vertices**. The value of A_{ij} is either **1** or **0** depending on whether there is an edge from vertex **i** to vertex **j**.



j \ i	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)**.



i → / 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

Since it is an **undirected graph**, for edge **(0,2)**, we also need to mark edge **(2,0)**; making **the adjacency matrix symmetric about the diagonal**.

Edge lookup (checking if an edge exists between ***vertex A*** and ***vertex B***) is extremely **fast** in adjacency matrix representation but we must **reserve space** for every possible link between all vertices (**V** x **V**).

Pros of Adjacency Matrix

- The operations like **adding** an edge, **removing** an edge, and **checking adjacency** are extremely time efficient, **constant time operations**.
- If the **graph is dense** (large number of edges), an adjacency matrix should be the first choice. Even if the matrix is sparse, we can use **sparse matrices**.
- **Parallel computing** enable us to perform even **expensive matrix operations**.
- We can get important insights into the nature of the graph and the relationship between its vertices.

Cons of Adjacency Matrix

- The **VxV space** requirement of the adjacency matrix makes it a memory hog. Graphs in the real life usually don't have too many connections and this is the major reason why **adjacency lists** are the better choice for most tasks.

- While basic operations are easy, operations like **inEdges** and **outEdges** are expensive when using the adjacency matrix representation.

Adjacency Matrix Applications

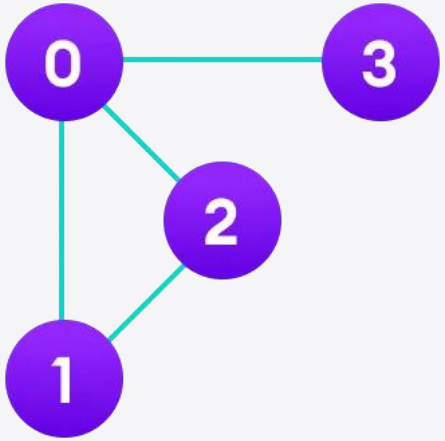
- Creating routing table in networks
- Navigation tasks

2. Adjacency List

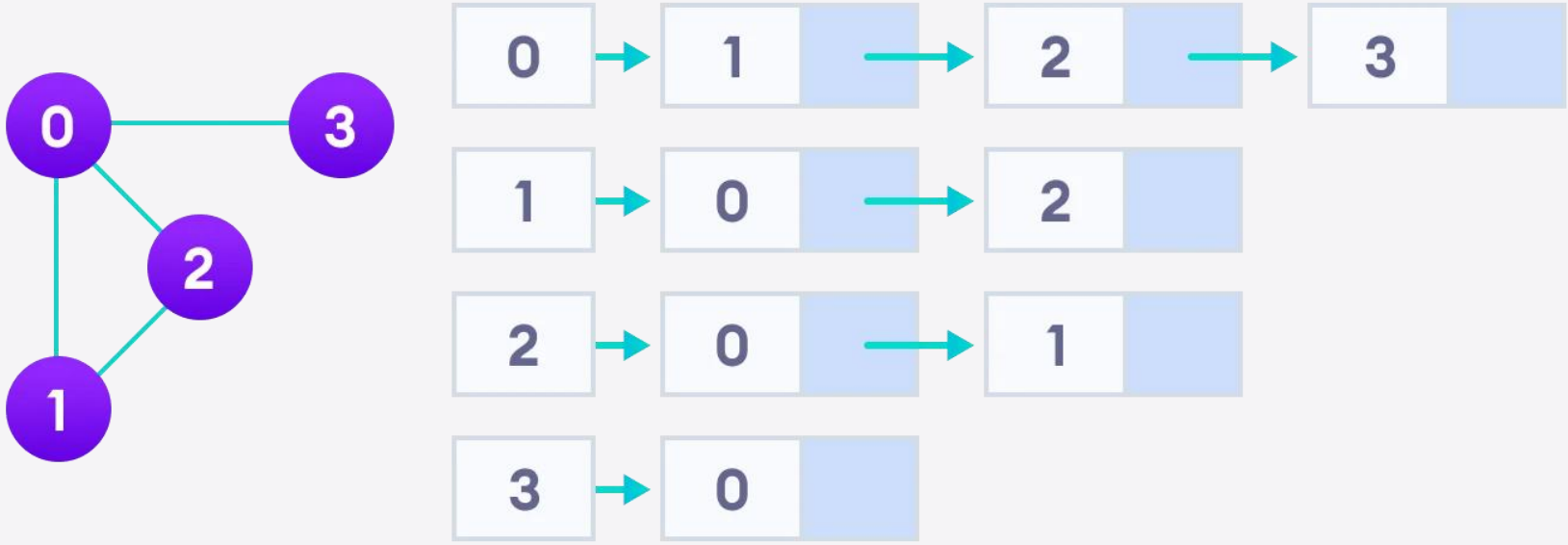
It represents a graph as an **array** of **linked lists**.

The **index of the array** represents a **vertex** and **each element in its linked list** represents the other vertices with which have an **edge**.

For example,

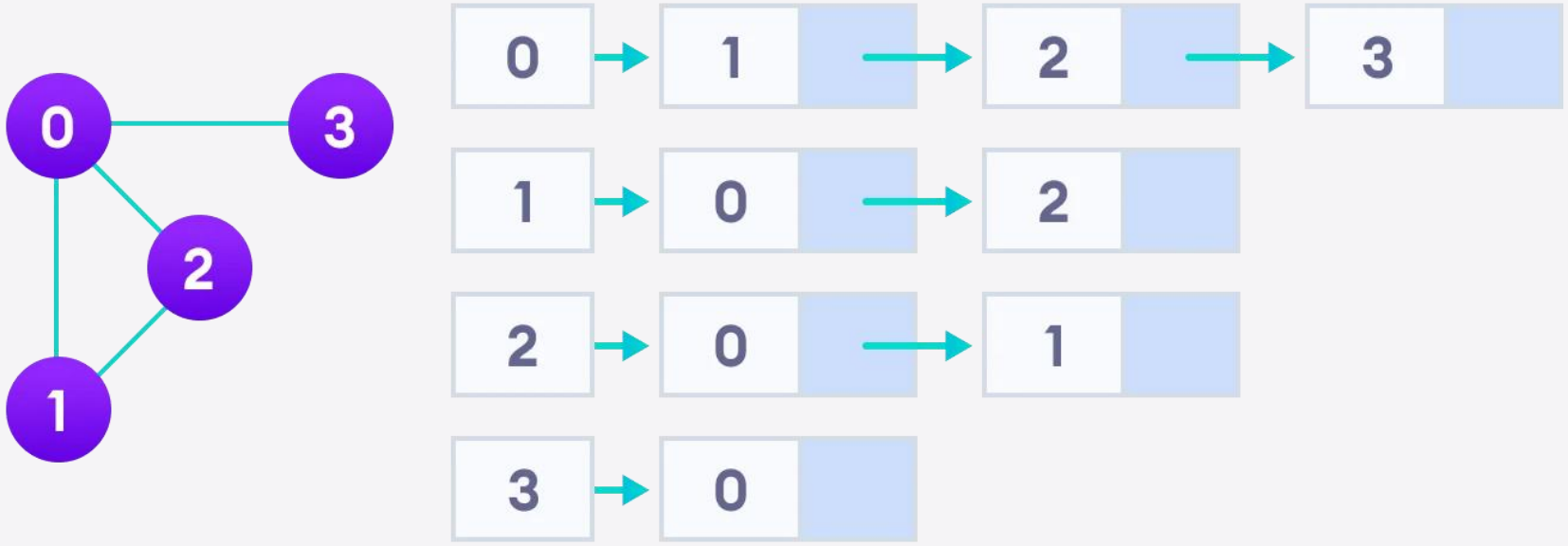


We can represent this graph in the form of a **linked list** as



Here, the vertices **0**, **1**, **2**, **3**, have a **linked list** with all its adjacent vertices.

For instance, vertex **1** has two adjacent vertices **0** and **2**. Therefore, **1** is linked with **0** and **2**.



An **adjacency list** is **efficient in terms of storage** because we only need to store the values for the edges.

For a graph with millions of vertices, this can mean a lot of saved space.

Pros of Adjacency List

- **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.
- It also helps to **find all the vertices adjacent to a vertex easily**.

Cons of Adjacency List

- Finding the **adjacent list** is not quicker than the **adjacency matrix** because **all the connected nodes must be first explored** to find them.

Adjacency List Applications

- It is faster to use **adjacency lists** for graphs having a smaller number of edges.

Adjacency List Structure

- The simplest **adjacency list** needs a **node data structure**, to store a **vertex** and a **graph data structure** to organize the nodes.

- We stay close to the basic definition of a graph - a collection of vertices and edges **{V, E}**. For simplicity, we use an unlabeled graph, i.e., the vertices are identified by their indices **0**,**1**,**2**,**3**.

Option #1

```
struct node
{
    int vertex;
    struct node* next;
};

struct Graph
{
    int numVertices;
    struct node** adjLists;
};
```

Don't let the **struct node** ** **adjLists** overwhelm you.

We want to store a **pointer to a pointer** to **struct node**, given that we don't know how many vertices the graph will have and so **we cannot create an array of Linked Lists at compile time**.

Option #2

```
class Graph
{
    private:
        int numVertices;
        list<int> *adjLists;
};
```

It is the same structure but by using the **built-in list STL data structures** of C++, we make the structure a bit cleaner. We are going to use this one.

Implementation

#include <list>

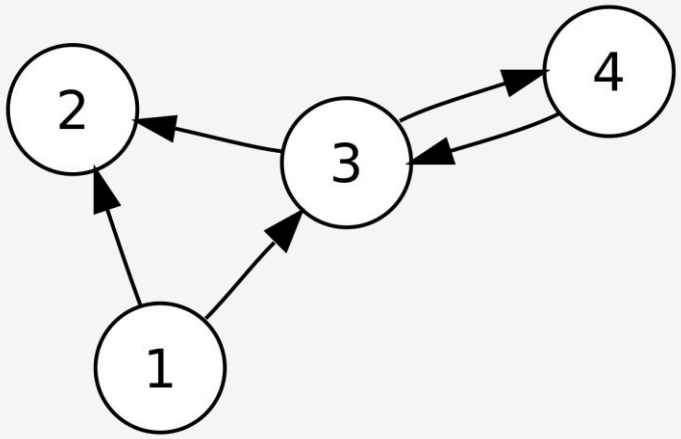
```
class Graph
{
    private:
        int numVertices;
        list<int> *adjLists;

    public:
        Graph(int V)
        {
            numVertices = V;
            adjLists = new list<int> [V];
        }
        ~Graph()
        {
            delete []adjLists;
        }
        void addEdge(int, int);
        void printGraph();
};
```

```
void Graph::printGraph()
{
    for (int i = 0; i < this->numVertices; i++)
    {
        cout << "\nVertex " << i << ":";

        for (auto x : this->adjLists[i])
            cout << "-> " << x;

        cout << endl;
    }
}
```

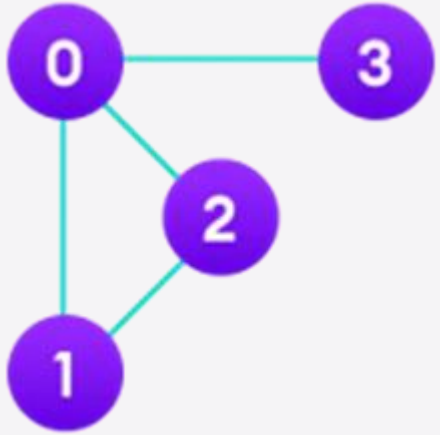
Implementation - addEdge

Depending on whether a **directed** or **undirected graph** is implemented, the **addEdge** function may vary slightly.

Directed Graph



Undirected Graph




```
void Graph::addEdgeDirect(int s, int d)
{
    this->adjLists[s].push_back(d);
}
```

```
void Graph::addEdge(int s, int d)
{
    this->adjLists[s].push_back(d);
    this->adjLists[d].push_back(s);
}
```

Program body

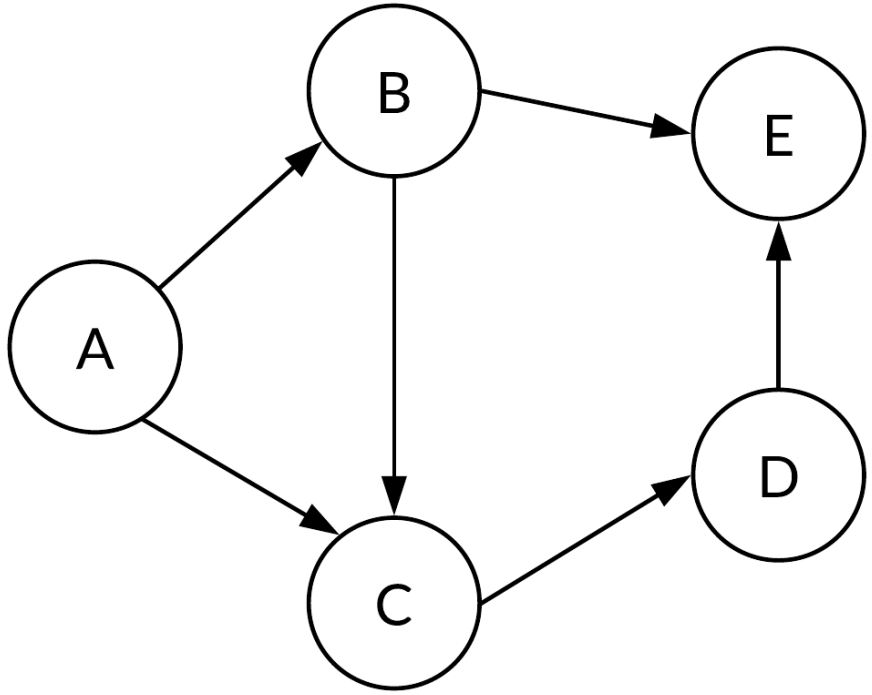
```
int main()
{
    // Create a graph
    Graph G(5);

    // Add edges
    G.addEdge(0, 1);
    G.addEdge(0, 2);
    G.addEdge(0, 3);
    G.addEdge(1, 2);

    // Print Adj List
    G.printGraph();
}
```

Result:

```
Vertex 0:-> 1-> 2-> 3

Vertex 1:-> 0-> 2

Vertex 2:-> 0-> 1

Vertex 3:-> 0

Vertex 4:
```

3. Edge List

An edge set simply represents a graph as a collection of all its edges.




```
Edge list:

graph = [('A', 'B'),
('A', 'C'),
('B', 'C'),
('B', 'E'),
('C', 'D'),
('D', 'E')
]
```

Graph Operations

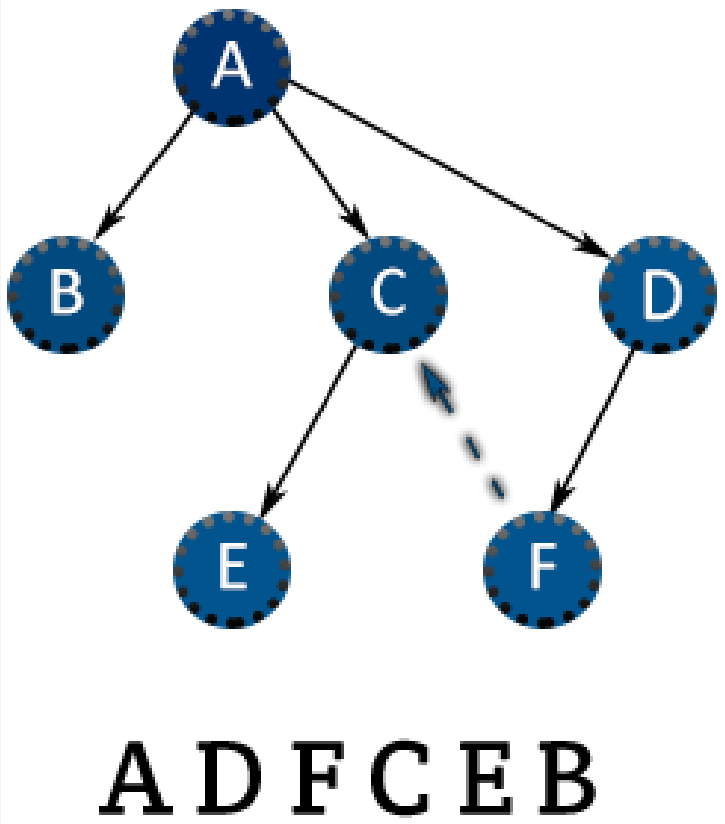
The most common graph operations are:

- Check if the element is present in the graph
- Graph Traversal
- Add elements(**vertex**, **edges**) to graph
- Finding the path from one vertex to another

Graph Traversal

Depth First Search (DFS)

Depth first Search or **Depth first traversal** is a **recursive algorithm** for searching all the vertices of a graph or tree data structure. Traversal means visiting all the **nodes** of a graph.



Depth First Search Algorithm

A standard **DFS** implementation mark each vertex of the graph into one of two categories:

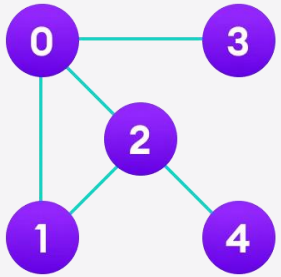
- **Visited**
- **Not Visited**

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

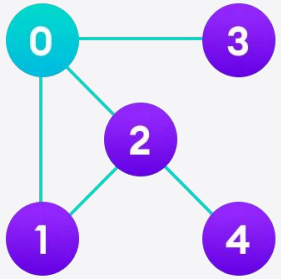
The **DFS** algorithm works as follows:

1. Start with any graph's vertices on **top of a stack**.
2. Add the top vertex of the **stack** to the **visited list**.
3. Add to the **top of the stack** the vertex's adjacent nodes that aren't in **visited** yet.
4. Keep repeating **steps 2** and **3** until the stack is **empty**.

We start from vertex 0.



					Visited
					Stack

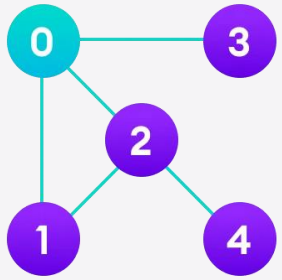


0					Visited
1	2	3			Stack

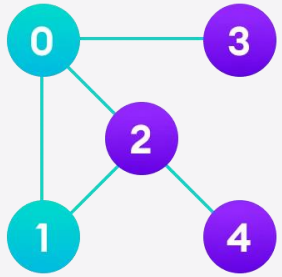
The **DFS** algorithm works as follows:

1. Start with any graph's vertices on **top of a stack**.
2. Add the top vertex of the **stack** to the **visited list**.
3. Add to the **top of the stack** the vertex's adjacent nodes that aren't in **visited** yet.
4. Keep repeating **steps 2** and **3** until the stack is **empty**.

Next, we visit the element at the top of stack i.e. 1 and go to its adjacent nodes.



0					Visited
1	2	3			Stack



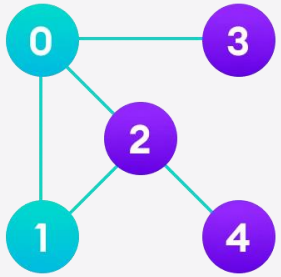
0	1				Visited
2	3				Stack

Since 0 has already been visited, we visit 2 instead.

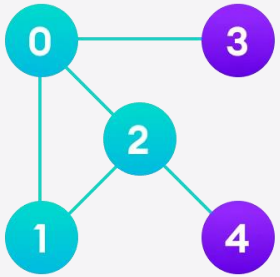
The **DFS** algorithm works as follows:

1. Start with any graph's vertices on **top of a stack**.
2. Add the top vertex of the **stack** to the **visited list**.
3. Add to the **top of the stack** the vertex's adjacent nodes that aren't in **visited** yet.
4. Keep repeating **steps 2** and **3** until the stack is **empty**.

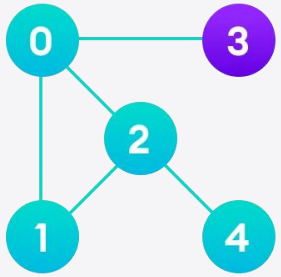
Vertex 2 has an unvisited adjacent vertex, 4, so we add that to the top of the stack and visit it.



0	1				Visited
2	3				Stack



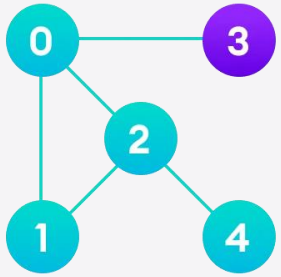
0	1	2			Visited
4	3				Stack



0	1	2	4		Visited
3					Stack

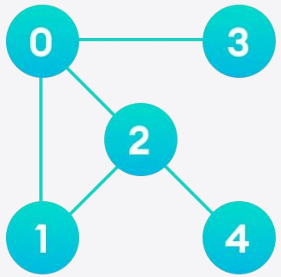
The **DFS** algorithm works as follows:

1. Start with any graph's vertices on **top of a stack**.
2. Add the top vertex of the **stack** to the **visited list**.
3. Add to the **top of the stack** the vertex's adjacent nodes that aren't in **visited** yet.
4. Keep repeating **steps 2** and **3** until the stack is **empty**.



0	1	2	4		Visited
3					Stack

After we visit the last element 3, it doesn't have any unvisited adjacent nodes.

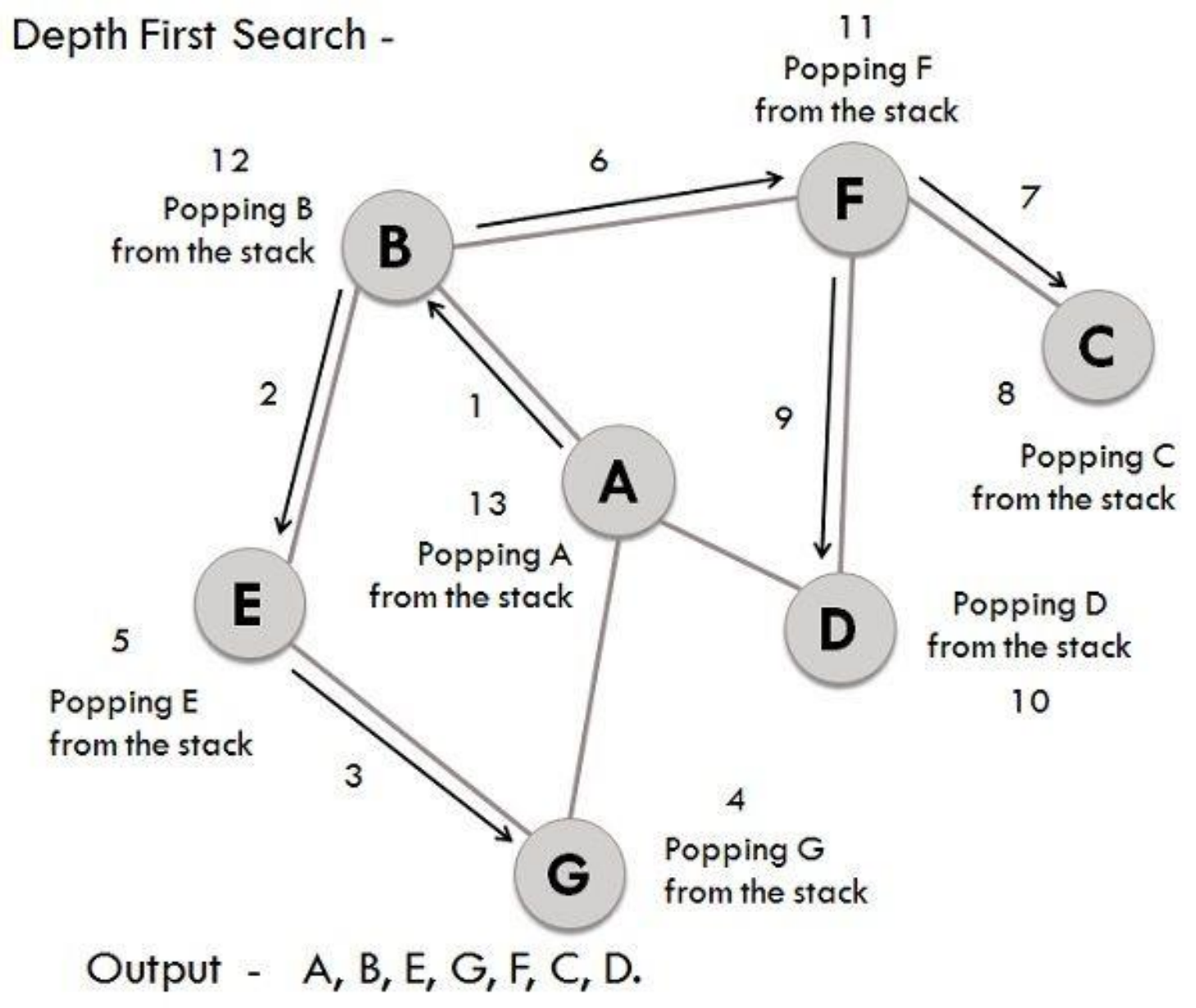


0	1	2	4	3	Visited
					Stack

So, we have completed the DFS of the graph.

Another Example

We start from vertex 'A'.



DFS Pseudocode (recursive implementation)

In the **init()** function, notice that we run the DFS function on every node.

This is done because the graph might have two or more **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)
}
```

Implementation

It is the same structure than before but updated to keep track of the unvisited nodes in a DFS traversal.

```
void Graph::ResetVisited()
{
    for (int i = 0; i < numVertices; i++)
        visited[i] = false;
}
```

```
class Graph
{
    private:
        int numVertices;
        list<int> *adjLists;
        bool *visited;

    public:
        Graph(int V)
        {
            numVertices = V;
            adjLists = new list<int> [V];
            visited = new bool[V];
        }
        ~Graph()
        {
            delete []adjLists;
            delete []visited;
        }
        void addEdge(int, int);
        void printGraph();
        void ResetVisited();

        void DFS(int);
};
```

Program body

```
int main()
{
    // Create a graph
    Graph G(5);

    G.addEdge(0, 1);
    G.addEdge(0, 2);
    G.addEdge(0, 3);
    G.addEdge(1, 2);
    G.addEdge(2, 4);

    // Print Adj List
    G.printGraph();

    cout << "\nDFS: ";
    G.DFS(0);

    return 0;
}
```

Result:

```
Vertex 0:-> 1-> 2-> 3

Vertex 1:-> 0-> 2

Vertex 2:-> 0-> 1-> 4

Vertex 3:-> 0

Vertex 4:-> 2

DFS: 0 1 2 4 3
```

¿What is the time complexity?

DFS Implementation

1. Start with any graph's vertices on **top of a stack**.
2. Add the top vertex of the **stack** to the **visited list**.
3. Add to the **top of the stack** the vertex's adjacent nodes that aren't in **visited** yet.
4. Keep repeating **steps 2** and **3** until the stack is **empty**.

```
void Graph::DFS(int vertex)
{
    this->visited[vertex] = true;
    list<int> adjVertex = this->adjLists[vertex];

    cout << vertex << " ";

    for(auto i: adjVertex)
        if (!this->visited[i])
            DFS(i);
}
```

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**.

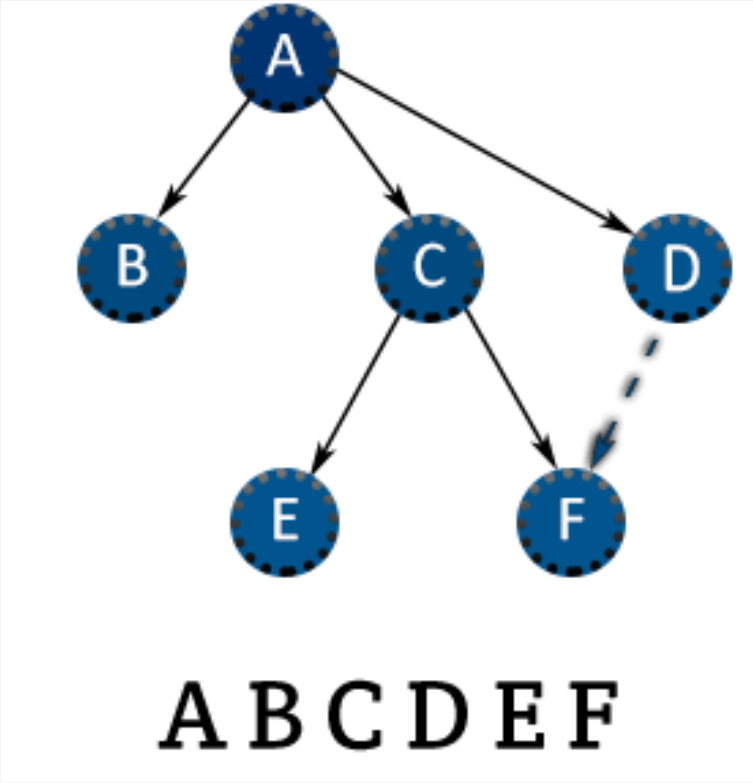
- The **space complexity** of the algorithm is **O(V)**.

Applications of DFS Algorithm

- For finding the path
- To test if the graph is bipartite
- For finding the strongly connected components of a graph
- For detecting cycles in a graph

Breadth First Search (BFS)

Breadth First Traversal or **Breadth First Search** is a **recursive algorithm** for searching all the **vertices** of a graph or tree data structure.



Breadth First Search 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.

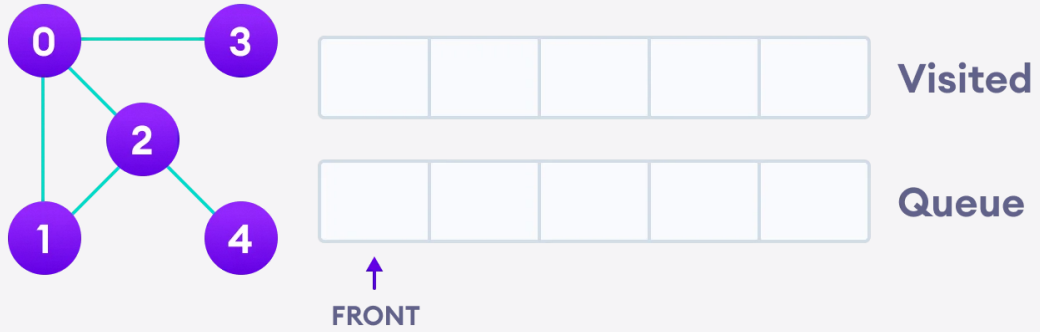
The algorithm works as follows:

- Start by putting **any one of the graph's vertices** at the back of a **queue**.
- Take the **front item** of the queue and add it to the **visited list**.
- 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.
- Keep repeating **steps 2** and **3** until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node.

BFS Example

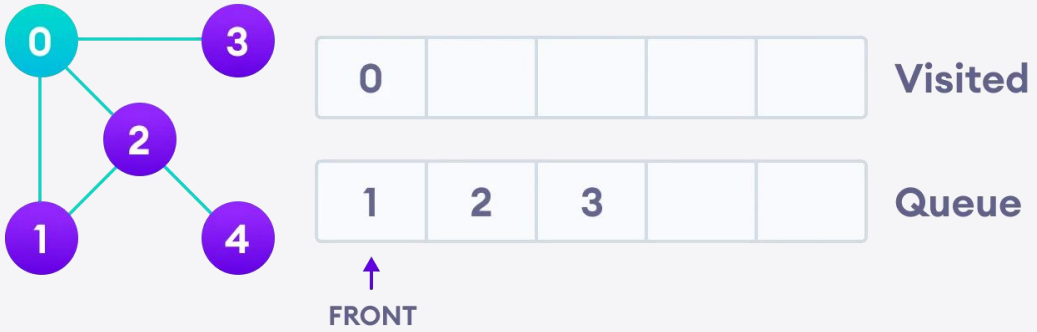
We start from **vertex 0**, the BFS algorithm starts by putting it in the **visited list** and putting all its adjacent vertices in the **stack**.



BFS Example

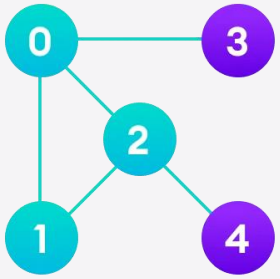
Next, we visit the element at the front of **queue** i.e. **1** and go to its adjacent nodes.

Since **0** has already been visited, we visit **2** instead.



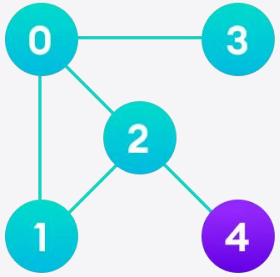
BFS Example

Vertex **2** has an unvisited adjacent vertex in **4**, so we add it to the back of the **queue** and visit **3**, **front of the queue**.



0	1	2			Visited
3	4				Queue

FRONT



0	1	2	3		Visited
4					Queue

FRONT

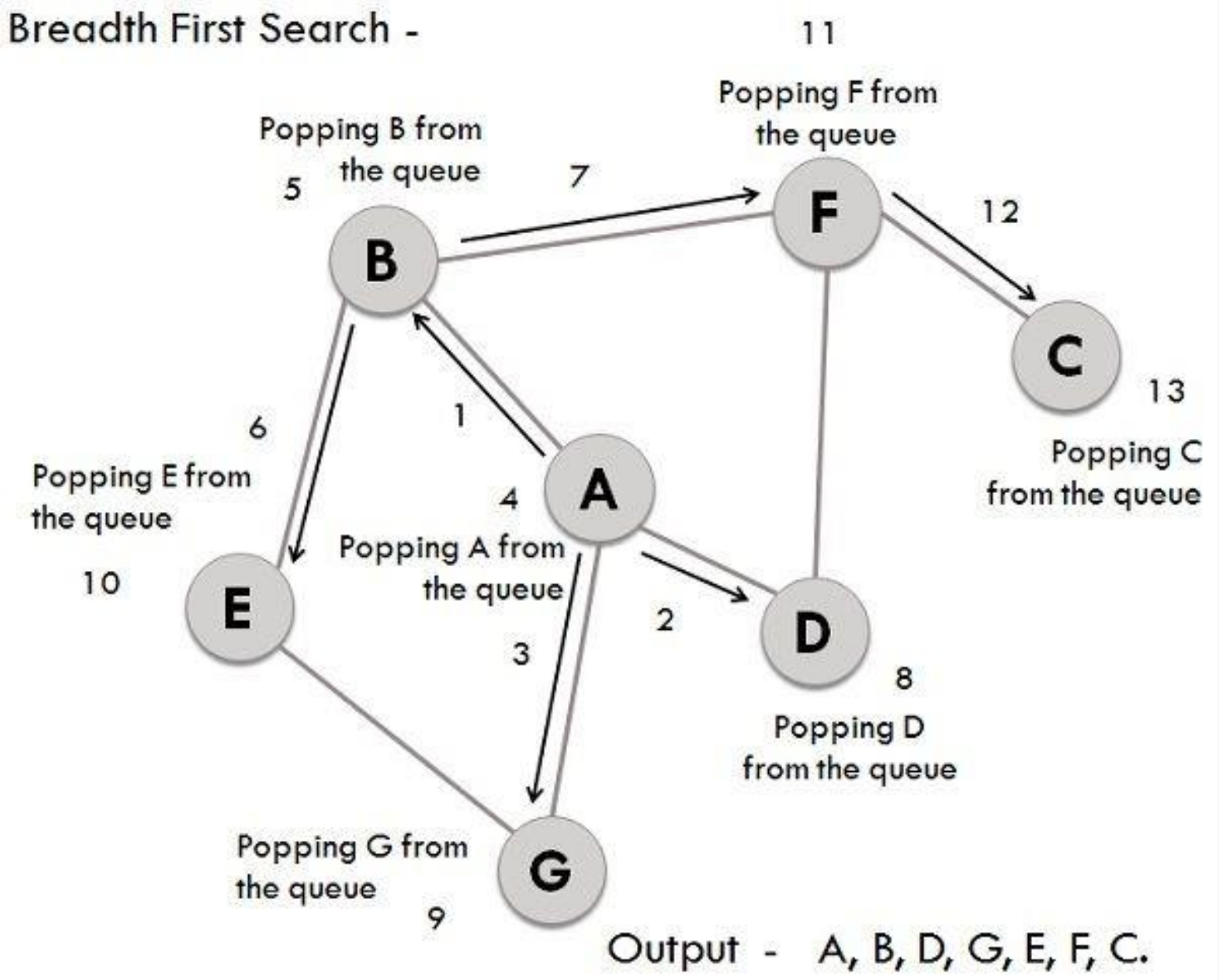
BFS Example

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

Since the **queue is empty**, we have completed the **BFS** of the graph.

Another Example

We start from vertex 'A'.



BFS Implementation

The BFS algorithm works as follows:

1. Start with anyone of the graph's vertices at the **back of a queue**.
2. Add the **front item** of the **queue** to the **visited list**.
3. Add to the **back of the queue** the vertex's adjacent nodes not visited before from the **queue**.
4. Keep repeating **steps 2** and **3** until the **queue is empty**.

```
void Graph::BFS(int startVertex)
{
    visited[startVertex] = true;

    list<int> queue;
    queue.push_back(startVertex);

    while (!queue.empty())
    {
        int currVertex = queue.front();

        cout << currVertex << " ";

        queue.pop_front();

        for (auto i: adjLists[currVertex])
        {
            if (!visited[i])
            {
                visited[i] = true;
                queue.push_back(i);
            }
        }
    }
}
```

Program body

```
int main()
{
    // Create a graph
    Graph G(5);

    G.addEdge(0, 1);
    G.addEdge(0, 2);
    G.addEdge(0, 3);
    G.addEdge(1, 2);
    G.addEdge(2, 4);

    // Print Adj List
    G.printGraph();

    cout << "\nBFS: ";
    G.BFS(0);

    return 0;
}
```

Result:

```
Vertex 0:-> 1-> 2-> 3

Vertex 1:-> 0-> 2

Vertex 2:-> 0-> 1-> 4

Vertex 3:-> 0

Vertex 4:-> 2

BFS: 0 1 2 3 4
```

¿What is the time complexity?

BFS Algorithm Applications

- To build index by search index
- For GPS navigation
- Path finding algorithms
- In Ford-Fulkerson algorithm to find maximum flow in a network
- Cycle detection in an undirected graph
- In minimum spanning tree

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)**.

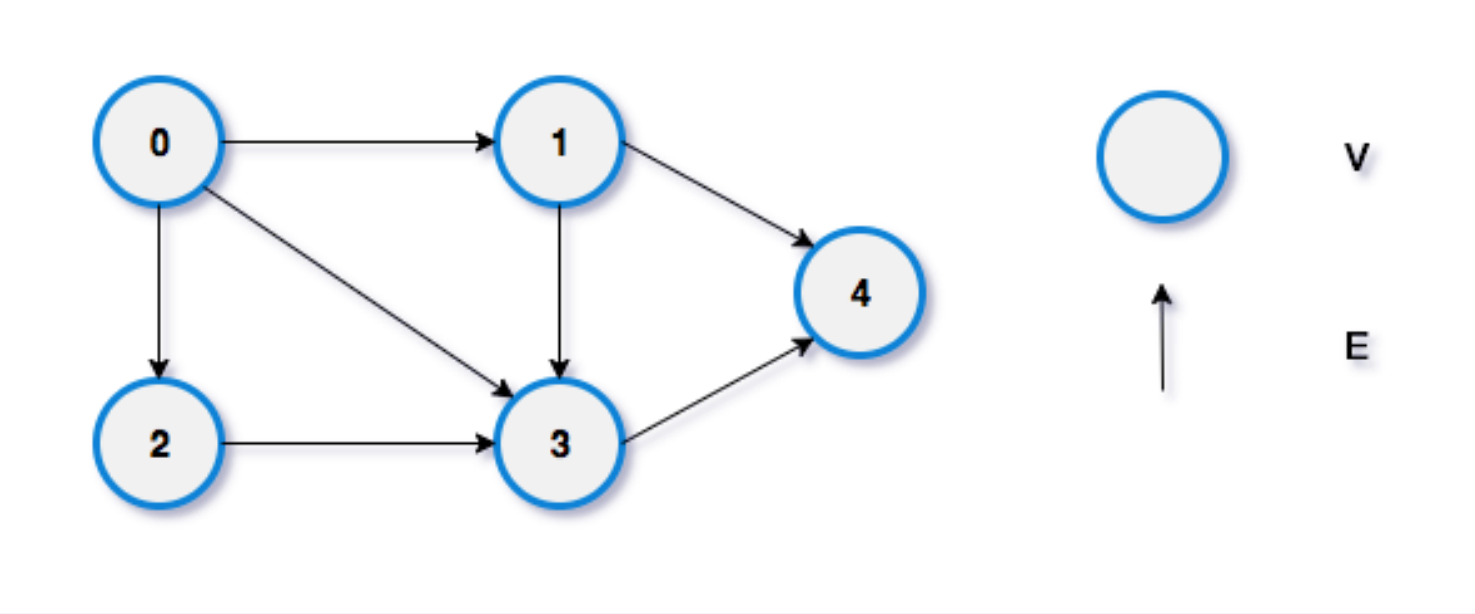
Degree (In-degree)
&
Out-degree

Degree of each vertex in the graph

Degree (vertex) = The number of edges incident to the vertex(node).

In other words, the number of relations a particular node makes with the other nodes in the graph.

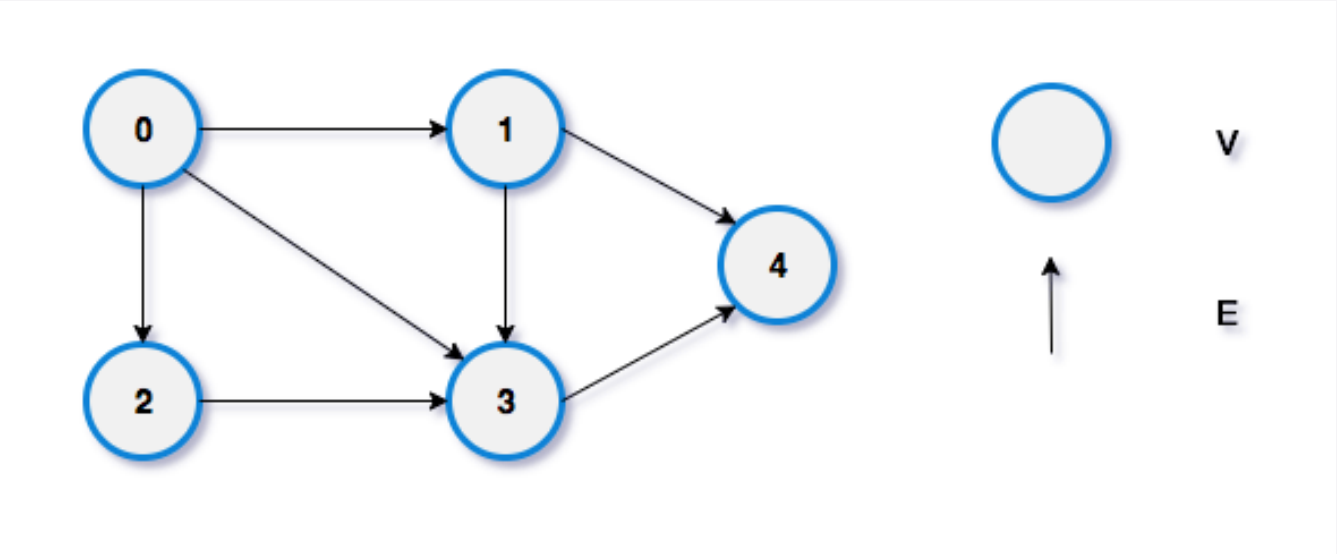
Example



In-degree

In-degree of a vertex is the number of edges coming to the vertex.

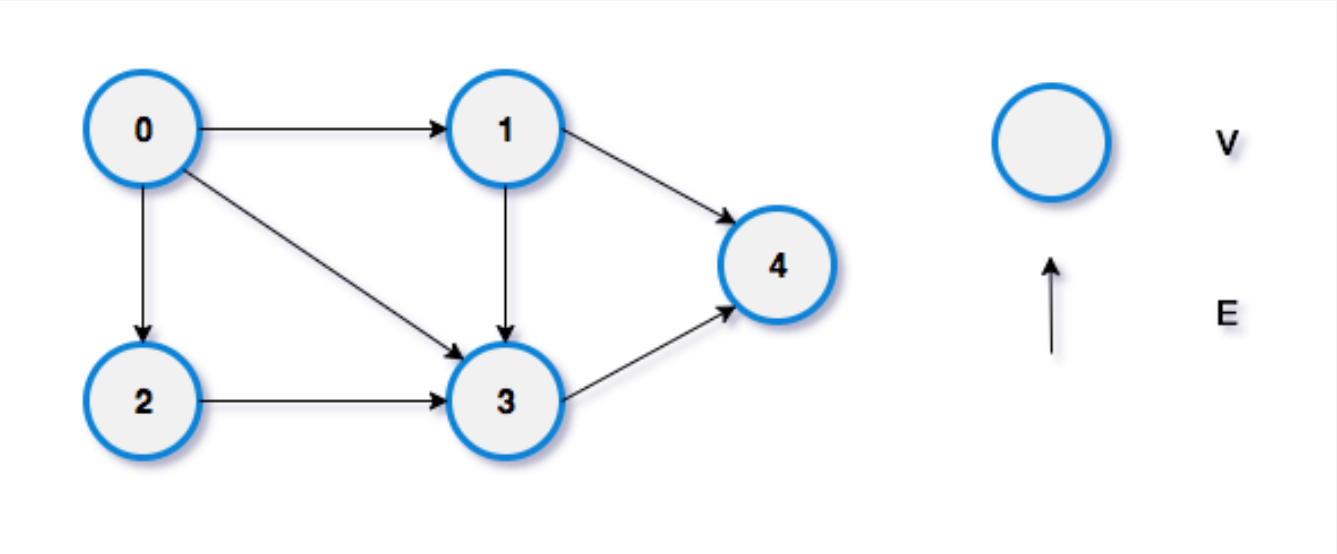
- **In-degree** of **vertex 0** = 0
- **In-degree** of **vertex 1** = 1
- **In-degree** of **vertex 2** = 1
- **In-degree** of **vertex 3** = 3
- **In-degree** of **vertex 4** = 2



Out-degree

Out-degree of a vertex is the number edges which are coming out from the vertex.

- **Out-degree** of **vertex 0** = 3
- **Out-degree** of **vertex 1** = 2
- **Out-degree** of **vertex 2** = 1
- **Out-degree** of **vertex 3** = 1
- **Out-degree** of **vertex 4** = 0

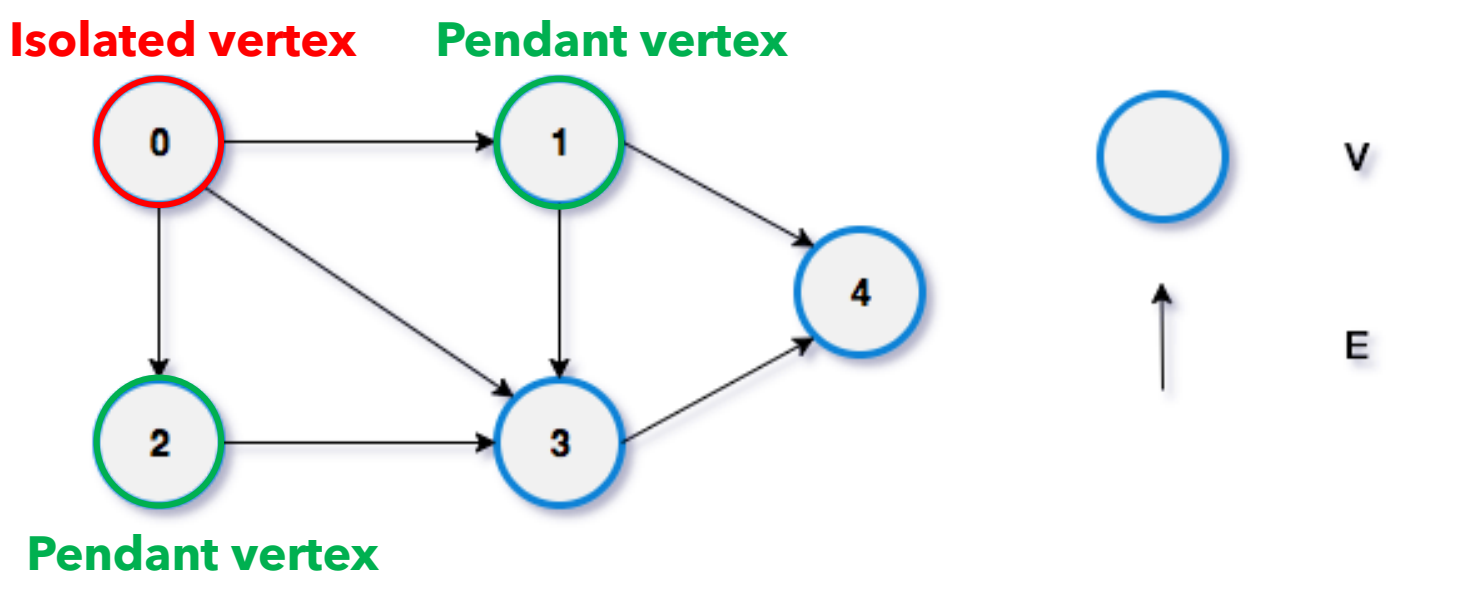


- **Pendant Vertex**

A vertex with **degree one** is called a pendant vertex.

- **Isolated Vertex**

A vertex with **degree zero** is called an isolated vertex.



Act 4.1
- Graph: Representations and traversals

All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

What do I have to do?

Individually, create a **Graph** class with the variables number of vertices, number of edges, Adj Matrix, Adj list

and the next class function:

- **LoadGraph** (Randomly fill the Adjacency Matrix / list)

Input: number of vertices (n), number of edges (m)

Adjacency Matrix / list with **m** connections **must match**.

The application must ask the user for the number of vertices (**n**), the number of edges (**m**)

- **DFS** (Print the Depth-Fist Search Path)

Input: Initial node

Apply to both Adjacency matrix and list, **must match** (same initial node). **All vertex must be shown**.

- **BFS** (Print the Breadth-Fist Search Path)

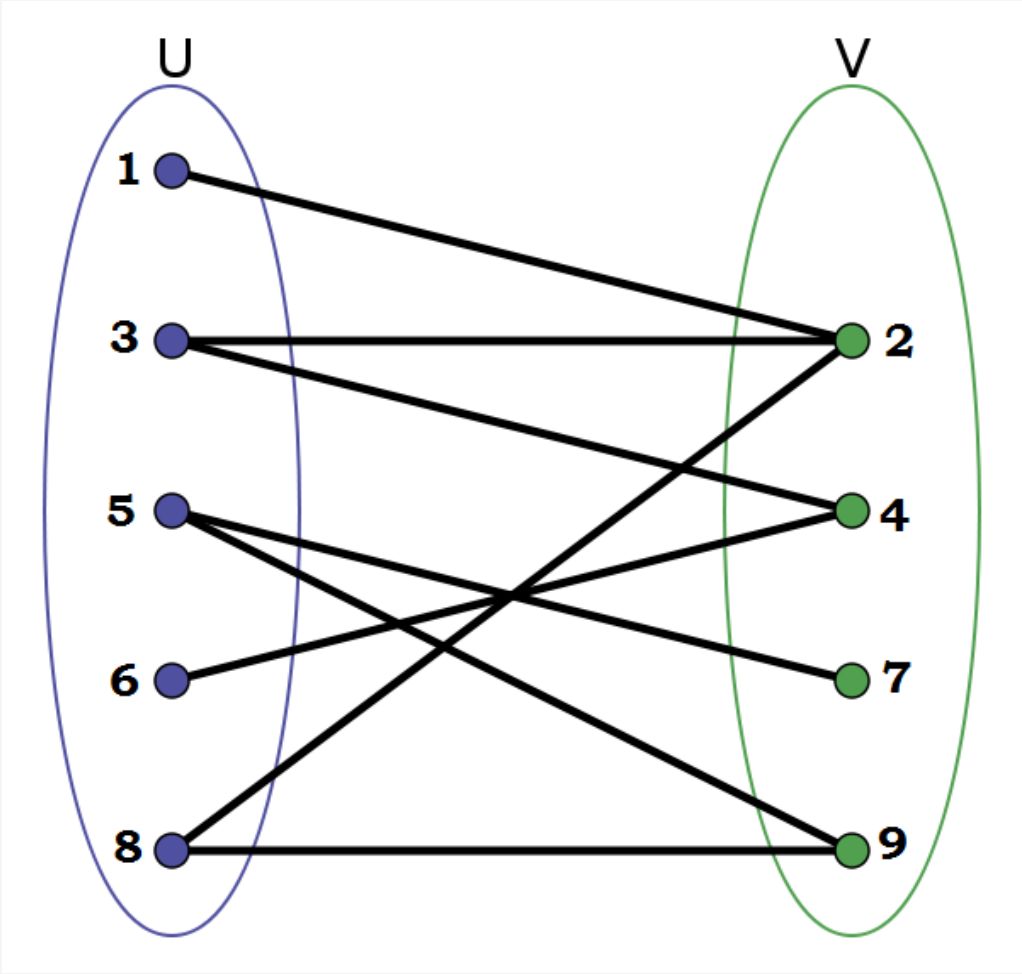
Input: Initial node

Apply to both Adjacency matrix and list, must match (same initial node). **All vertex must be shown**.

All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

Bipartite graph

A **bipartite graph** (or bigraph) is a graph whose vertices can be divided into two **disjoint sets** **U** and **V** such that every **edge** connects a **vertex** in **U** to one in **V**.

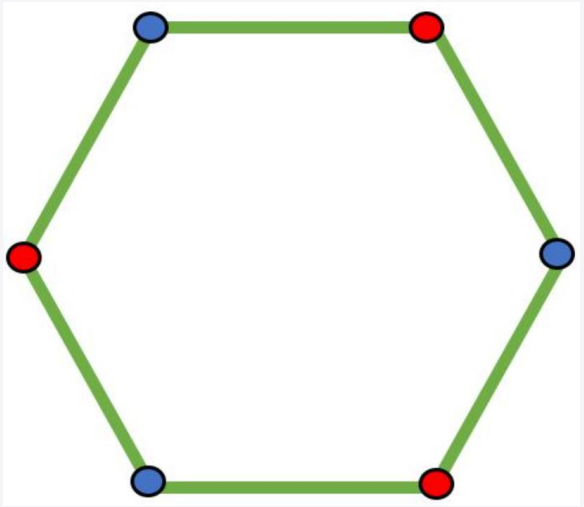


It is possible to test whether a graph is **bipartite** or **not** using a **Breadth-first search (BFS)** algorithm. There are two ways to check for a bipartite graph:

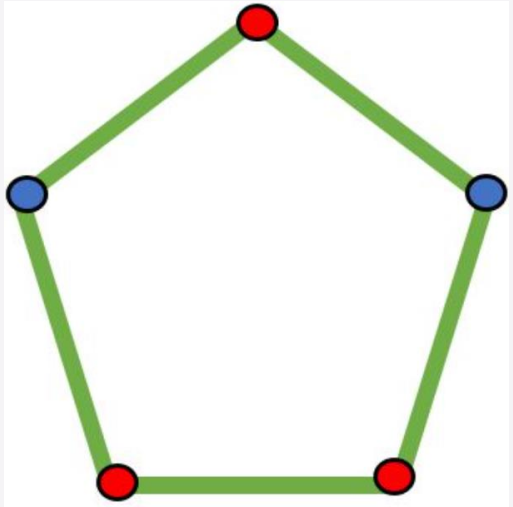
1. A graph is a **bipartite graph** if and only if it is 2-colorable.

2. A graph is a **bipartite graph** if and only if it does not contain an odd cycle.

While doing **BFS traversal**, each node in the BFS tree is given its parent's opposite color. If there exists an **edge** connecting the **current vertex** to a previously colored **vertex with the same color**, then **the graph is not bipartite**.



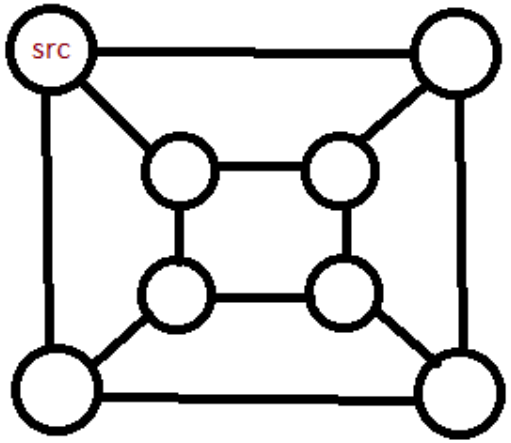
Cycle graph of length 6



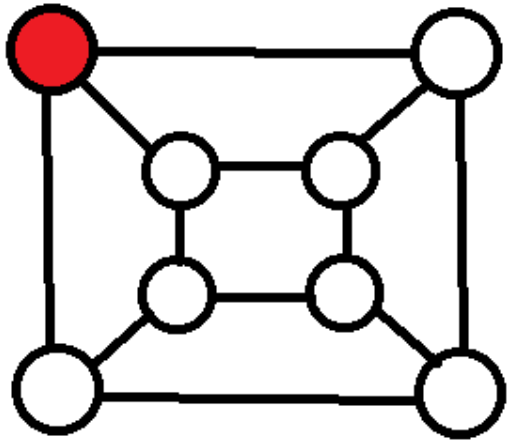
Cycle graph of length 5

It is not possible to color a cycle graph with an **odd cycle** using **two colors**.

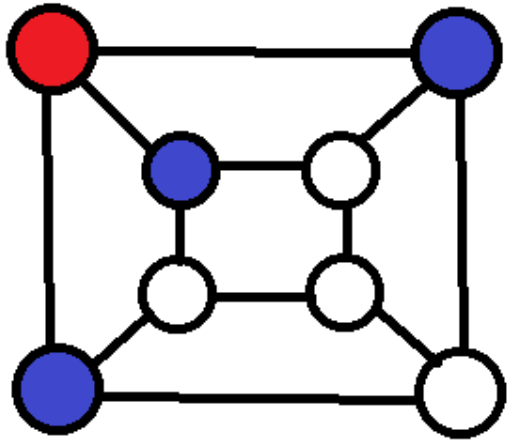
IsBipartite Algorithm Example



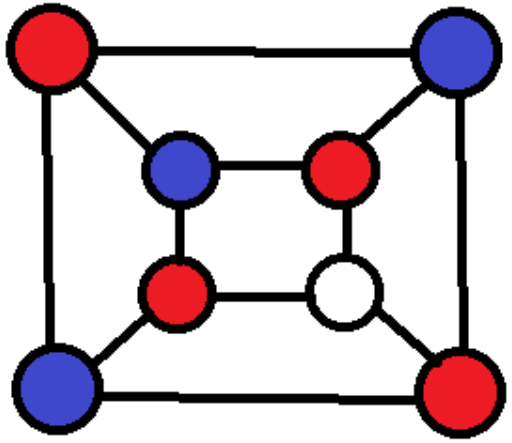
given a graph with source vertex



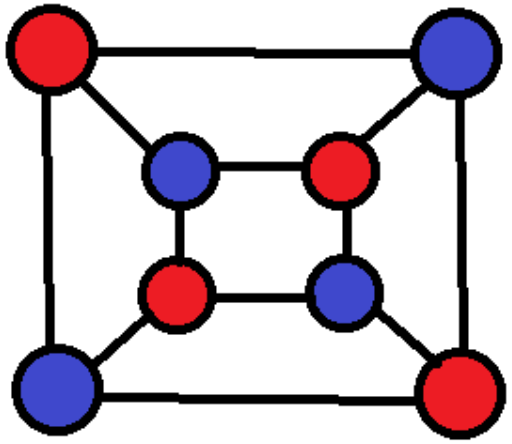
colour src vertex, say red



assign another colour to the neighbours, say blue



assign the neighbours of the vertices of the previous step the colour red



repeat till all vertices are coloured, or a conflicting colour assignment occurs.

set U: red colour
set V: blue colour

Implementation

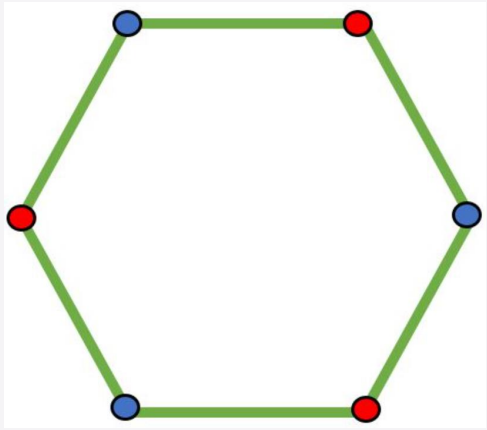
Like the BFS algorithm, the algorithm works as follows:

1. Start by putting the start Vertice at the back of a queue and marked as visited and set its color to 0.
2. 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 and set its color to the opposite of its parent.
3. Keep repeating until the queue is empty or the color of a node and its parent is the same.

```
bool Graph::isBipartite(int startVertex)
{
    vector<bool> color(numVertices);

    visited[startVertex] = true;
    color[startVertex] = 0;

    list<int> queue;
    queue.push_back(startVertex);

    while(!queue.empty())
    {
        int v = queue.front();
        queue.pop_front();

        for (auto u: adjLists[v])
        {
            if (!visited[u])
            {
                visited[u] = true;
                color[u] = !color[v];

                queue.push_back(u);
            }
            else if (color[v] == color[u])
            {
                return false;
            }
        }
    }
    return true;
}
```

Program body

```
Graph G(6);

G.addEdge(0, 1);
G.addEdge(1, 2);
G.addEdge(2, 3);
G.addEdge(3, 4);
G.addEdge(4, 5);
G.addEdge(5, 0);

// Print Adj List
G.printGraph();

cout << "\nDFS: ";
G.DFS(0);

G.ResetVisited();

cout << "\nBFS: ";
G.BFS(0);

G.ResetVisited();

cout << "\n\nIs Bipartite? ";
if (G.isBipartite(0))
    cout << "Graph is Bipartite";
else
    cout << "Graph is not Bipartite";
```

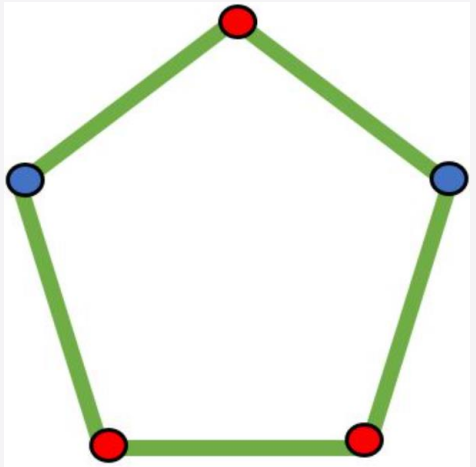


Result:

```
Vertex 0:-> 1-> 5

Vertex 1:-> 0-> 2

Vertex 2:-> 1-> 3

Vertex 3:-> 2-> 4

Vertex 4:-> 3-> 5

Vertex 5:-> 4-> 0

DFS: 0 1 2 5 4
BFS: 0 1 5 2 4 3

Is Bipartite? Graph is Bipartite
```

Program body

```
Graph G(5);

G.addEdge(0, 1);
G.addEdge(1, 2);
G.addEdge(2, 3);
G.addEdge(3, 4);
G.addEdge(4, 0);

// Print Adj List
G.printGraph();

cout << "\nDFS: ";
G.DFS(0);

G.ResetVisited();

cout << "\nBFS: ";
G.BFS(0);

G.ResetVisited();

cout << "\n\nIs Bipartite? ";
if (G.isBipartite(0))
    cout << "Graph is Bipartite";
else
    cout << "Graph is not Bipartite";
```



Result:

```
Vertex 0:-> 1-> 4

Vertex 1:-> 0-> 2

Vertex 2:-> 1-> 3

Vertex 3:-> 2-> 4

Vertex 4:-> 3-> 0

DFS: 0 1 2 4
BFS: 0 1 4 2 3

Is Bipartite? Graph is not Bipartite
```

¿What is the time complexity?

IsBipartite Algorithm Complexity

If **Adjacency matrix** is used, then:

- Worst time complexity case: **O(V^2)**
- Average time complexity case: **O(V^2)**
- Best time complexity case: **O(V^2)**
- Space complexity: **O(V^2)**

where **V** is the number of vertices.

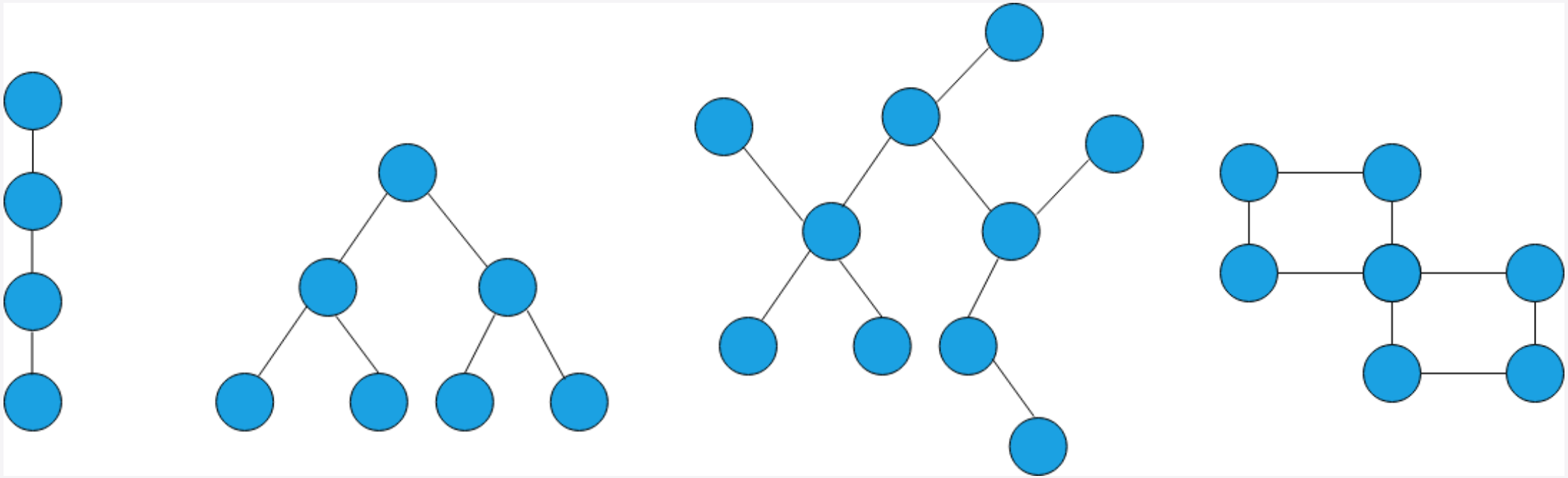
If **Adjacency list** is used, then:

- Worst time complexity case: **O(V+E)**
- Average time complexity case: **O(V+E)**
- Best time complexity case: **O(V+E)**
- Space complexity: **O(V+E)**

where **V** is the number of vertices.

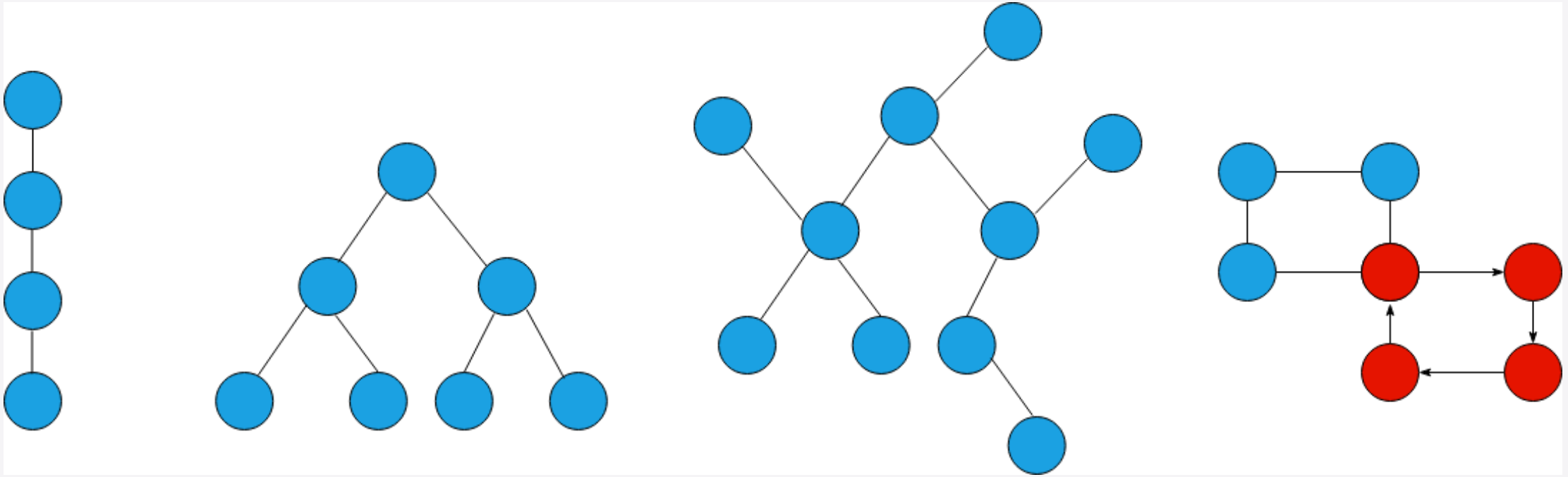
Acyclic graph (tree)
&
Directed Acyclic Graph (DAG)

What is a Tree?



We have a couple of graphs above; **can you spot the odd one that cannot be a tree?**

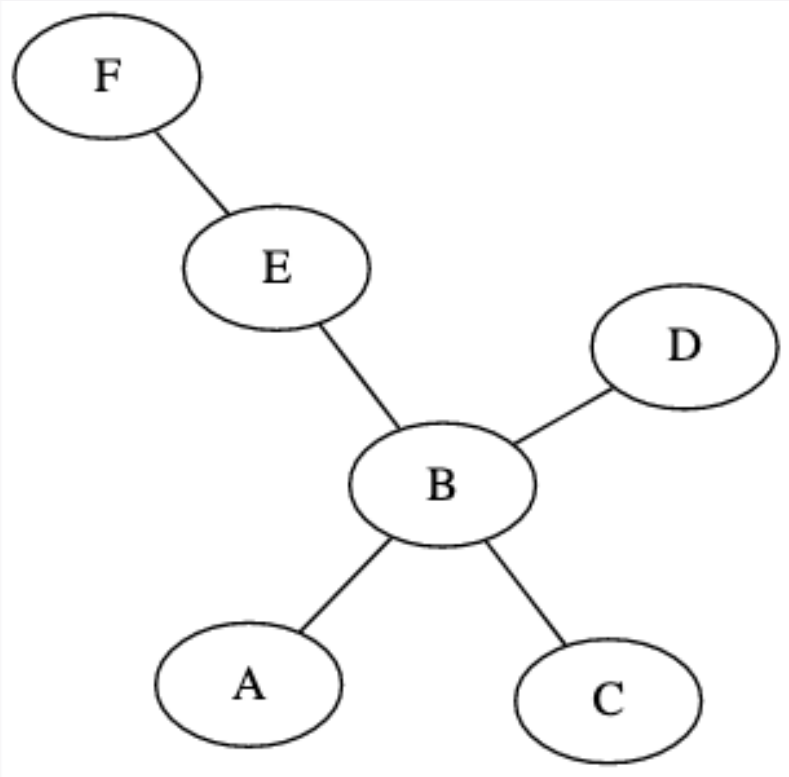
What is a Tree?

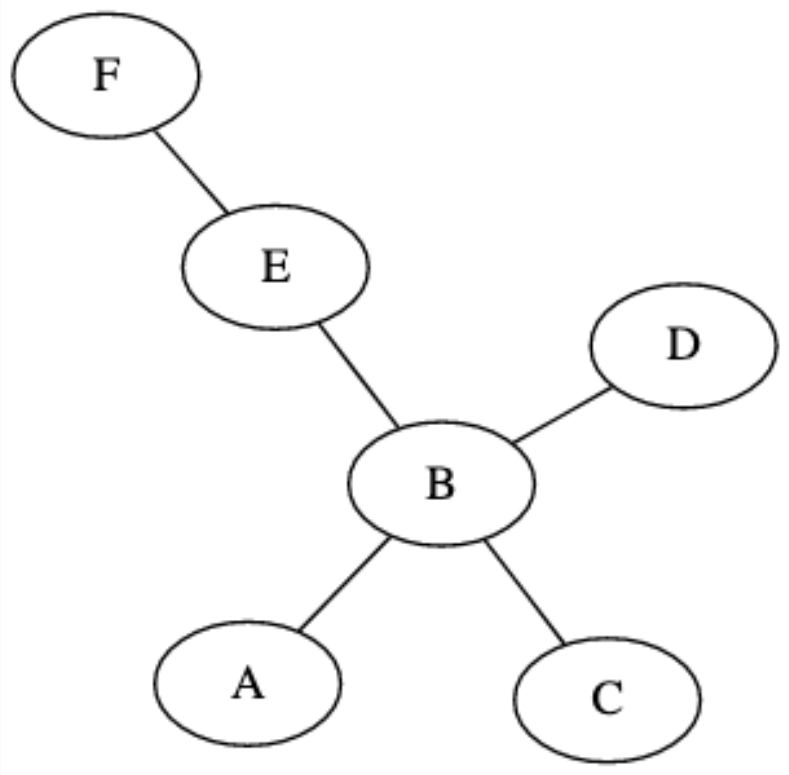


Because a **tree** is an **(un)directed graph with no cycles**. The key thing to remember is trees aren't allowed to have cycles in it.

What is an Acyclic Graph?

An acyclic graph is a **graph without cycles** (a cycle is a complete circuit). When following the graph from node to node, **you will never visit the same node twice**.





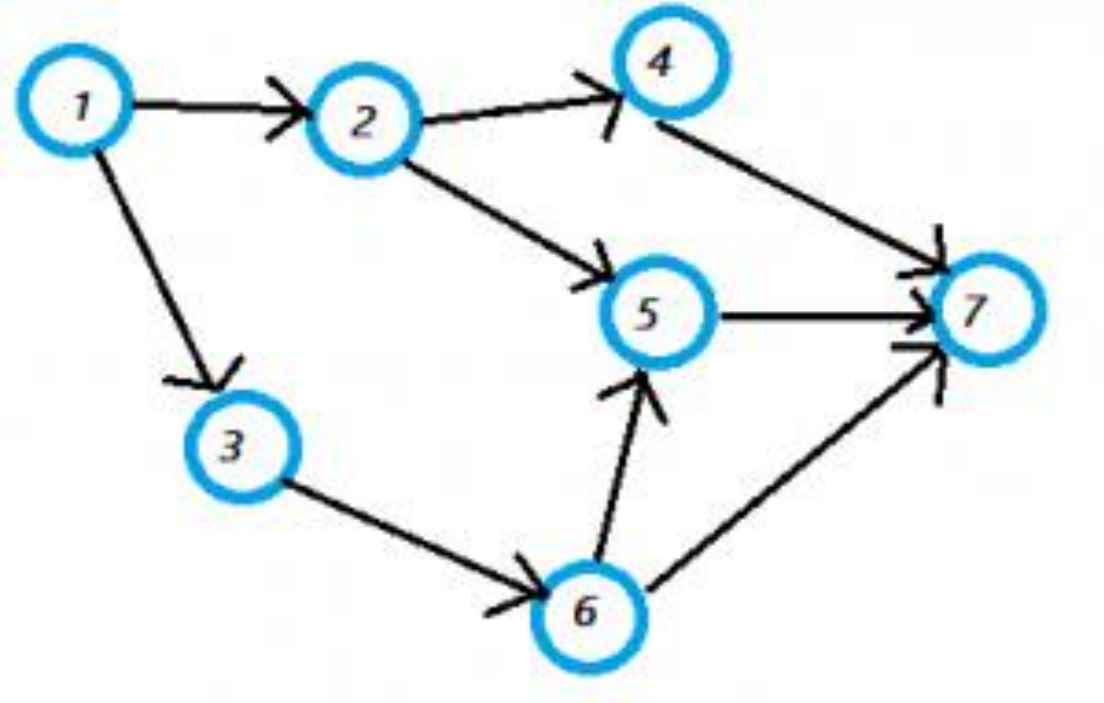
A **connected acyclic graph**, like the one above, is called a **tree**.

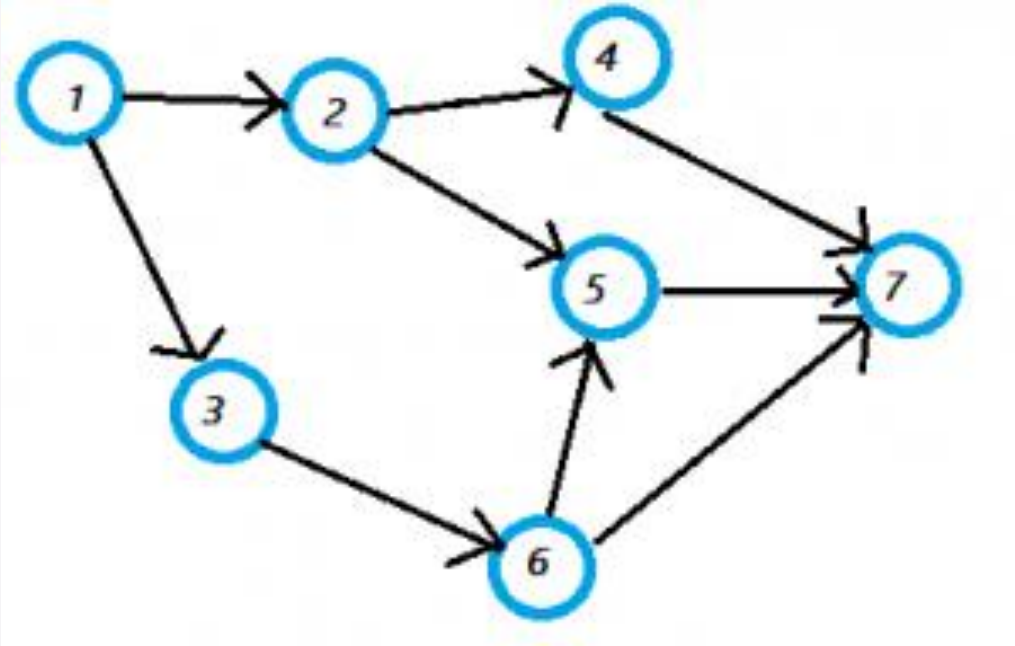
If one or more of the tree "*branches*" is **disconnected**, the acyclic graph is a called a **forest**.

What is a Directed Acyclic Graph?

A **directed acyclic graph** (**DAG**) is an **acyclic graph** that has a **direction** as well as a **lack of cycles**.

That is, each edge directed from one vertex to another, such that following those directions **will never form a closed loop**.



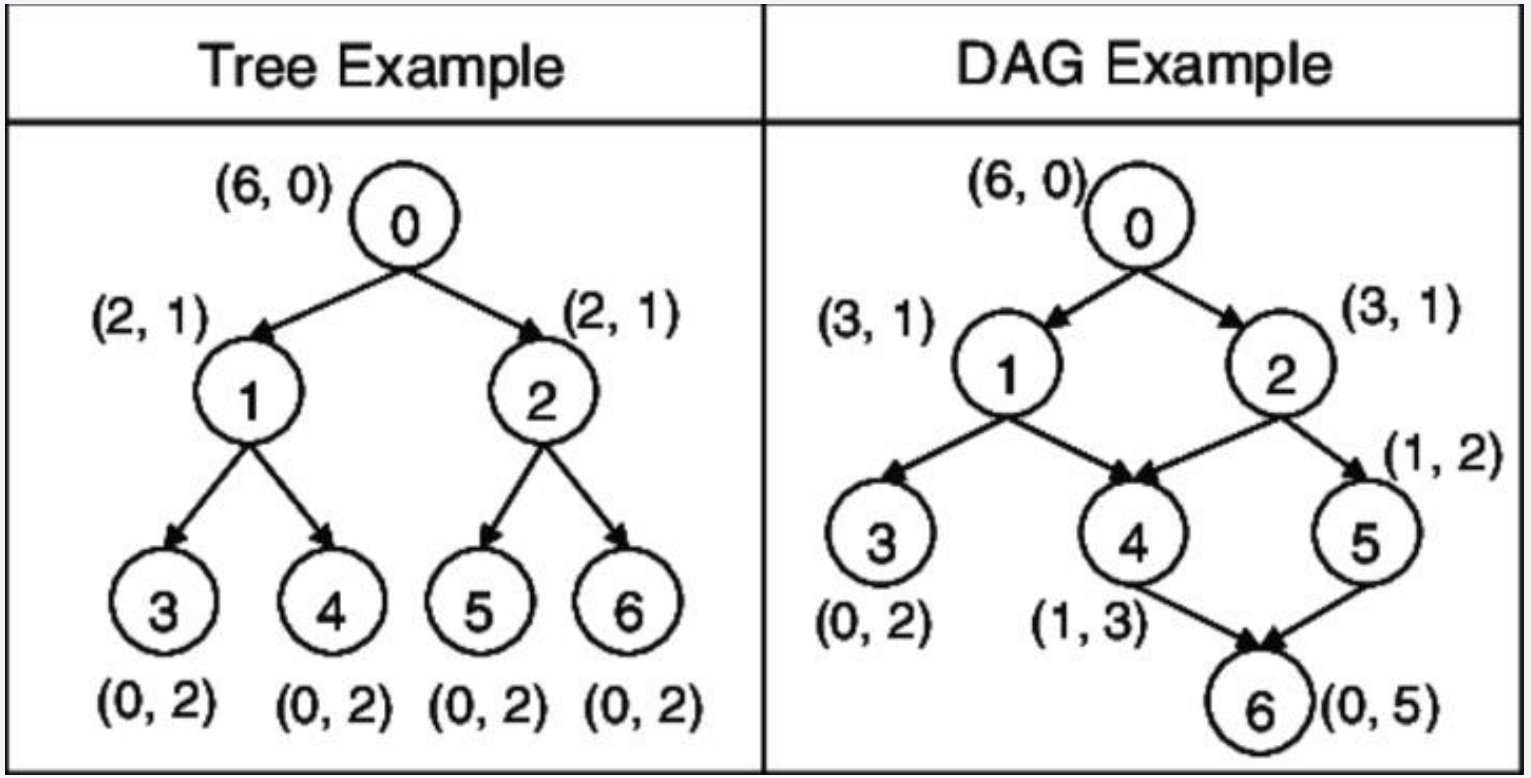


The parts of the above graph are:

- **Vertices set** = {1,2,3,4,5,6,7}.
- **Edge set** = {(1,2), (1,3), (2,4), (2,5), (3,6), (4,7), (5,7), (6,7)}.

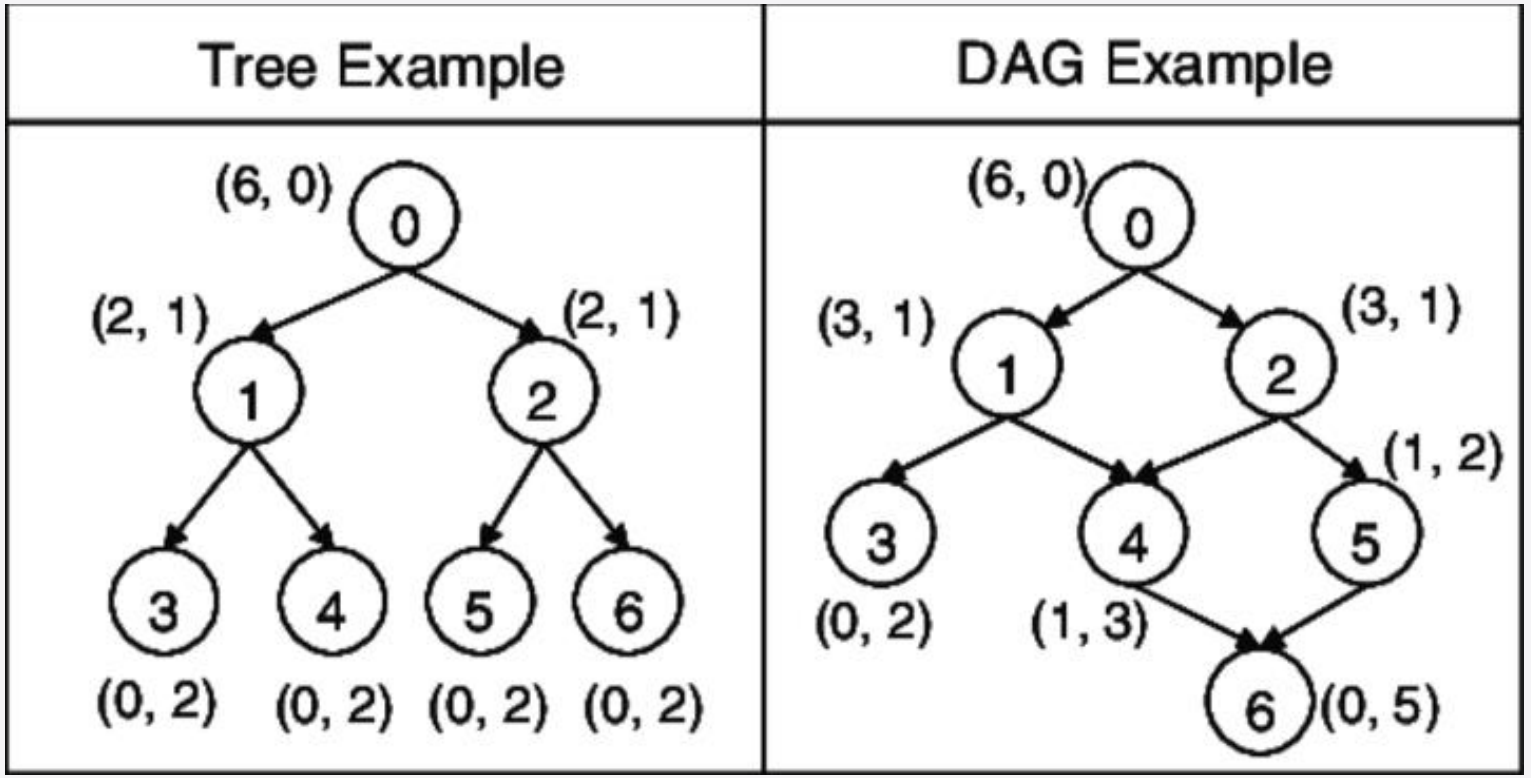
A **directed acyclic graph** has a **topological ordering**. This means that the nodes are ordered so that the **starting node has a lower value than the ending node**.

DAG = Tree?



Trees and **DAGs** are **connected**, **directed**, **rooted**, and have **no cycles**. Starting from any node and going up the parents you will eventually work your way up to the top (***root***).

DAG = Tree?



However,

- **DAG nodes have multiple parents**, there will be multiple paths on the way up (that eventually merge).
- **Trees** are **DAGs** with the **restriction that a child can only have one parent**.

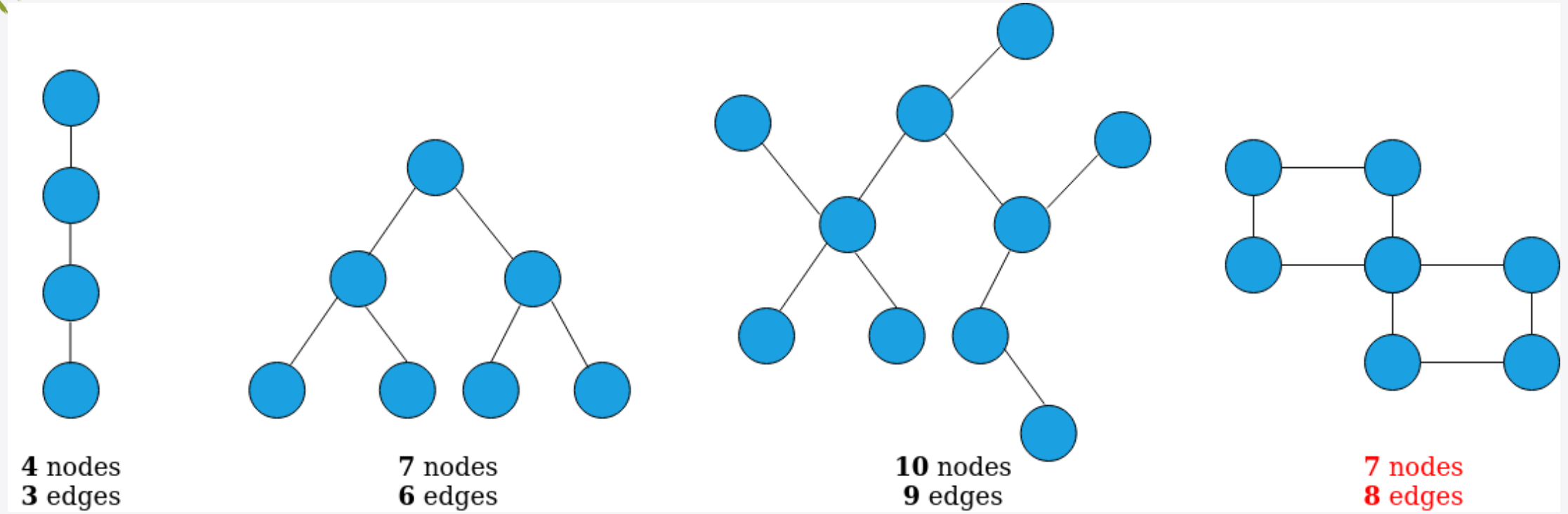
An undirected graph is a tree if it has the following properties.

- There is no cycle.
- The graph is connected.

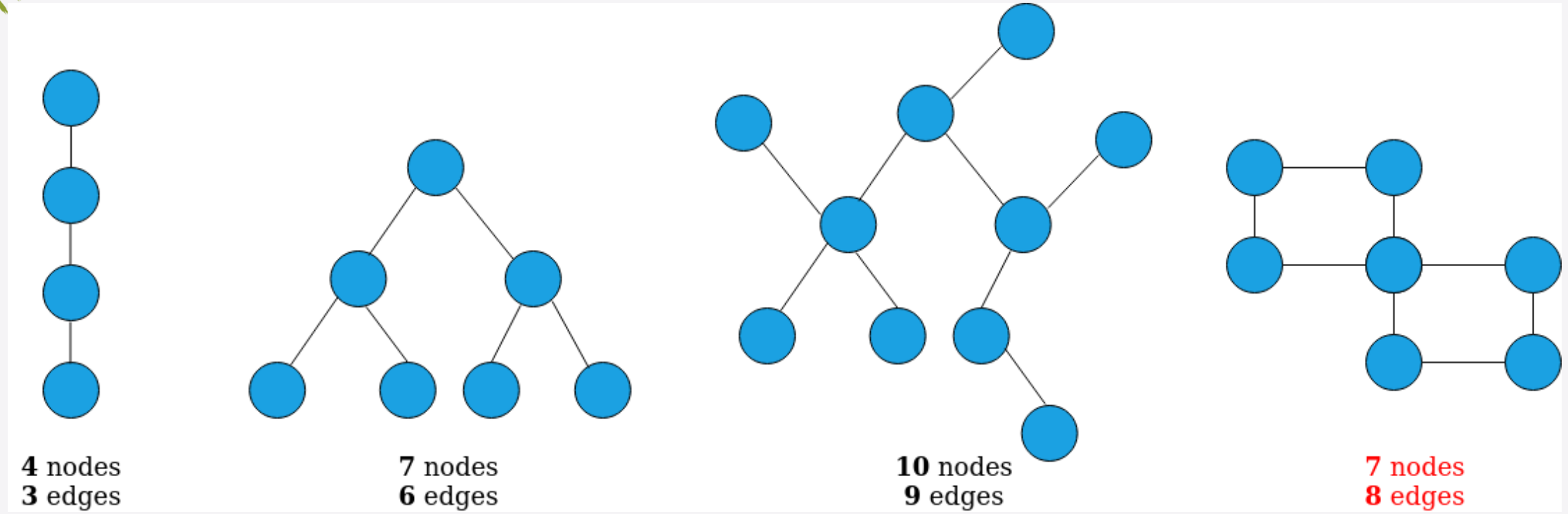
For an undirected graph, we can either use BFS or DFS to detect the above two properties.

How to check for connectivity?

In an undirected graph, we can start **BFS** or **DFS** from any vertex and **check if all vertices are reachable or not**. If all vertices are reachable, then the **graph is connected**, otherwise not.

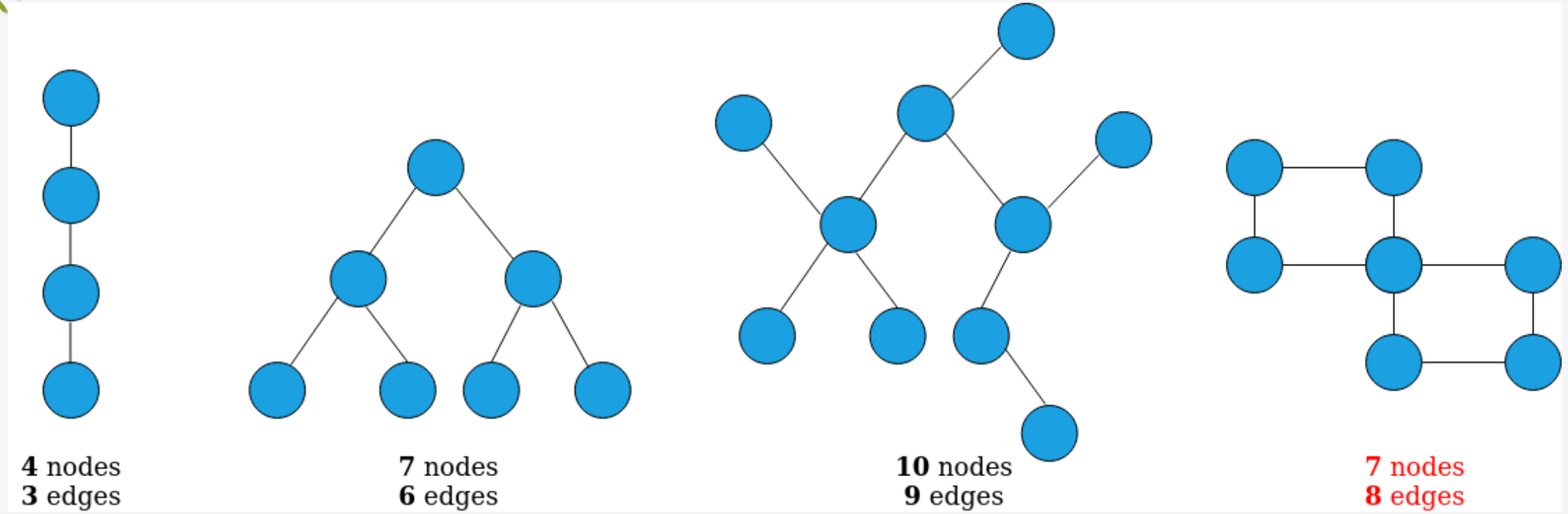


However, if we observe carefully the definition of tree and its structure, we will deduce that: All trees have **N - 1 edges**, where **N** is the **number of nodes**.



Since we have assumed our graph of **n nodes** to be **connected**, it must have at least **n – 1 edges** inside it.

Now **if we try to add one more edge than the n – 1 edges already the graph will end up forming a cycle** and thus will not satisfy the definition of tree.



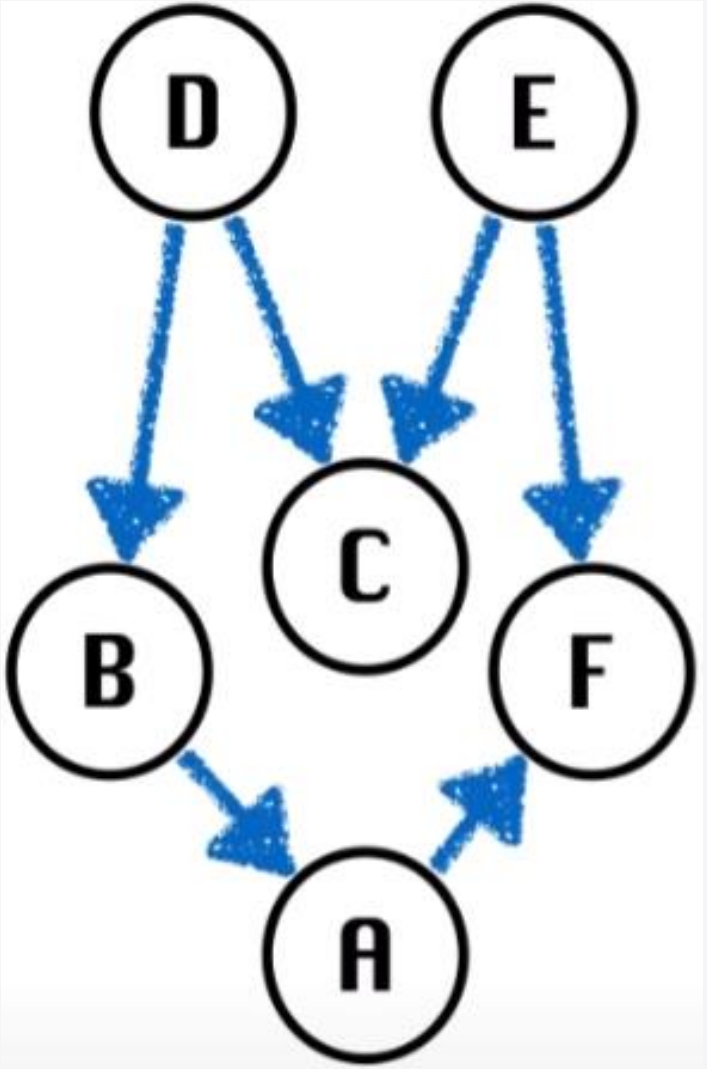
In this case, we must check only two thinks:

- **The graph is connected.**
- **The number of edges**

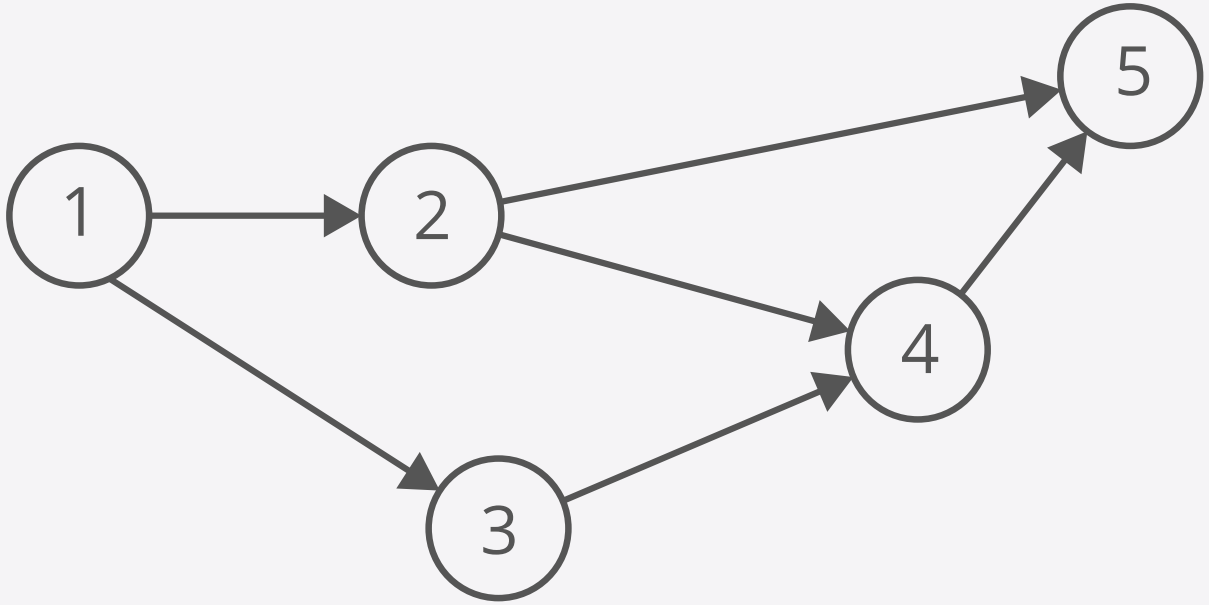
Topological Sorted

Imagine the **vertices** of the graph may represent **tasks** to be performed, and the edges may represent constraints that **one task must be performed before another**; in this application, a **topological ordering** is just a valid sequence for the tasks.

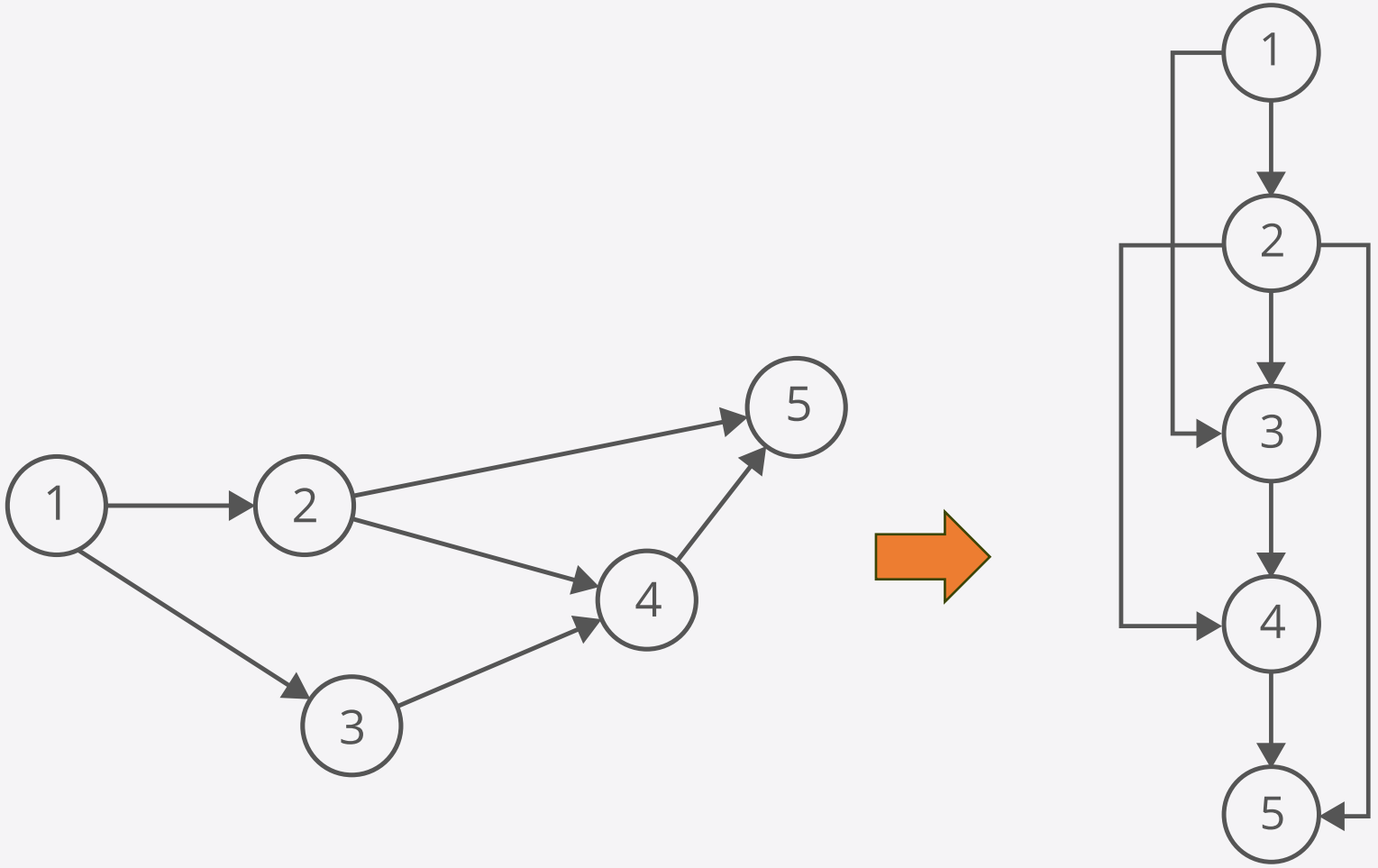
A topological sort of a **DAG = (V,E)** is a **linear ordering** of all its **vertices** such that if **DAG** contains an edge **(u,v)** then **u** always appears before **v** in the ordering.



The **topological sort algorithm** takes a **directed graph** and returns an **array of the nodes** where each node appears before all the nodes it points to.

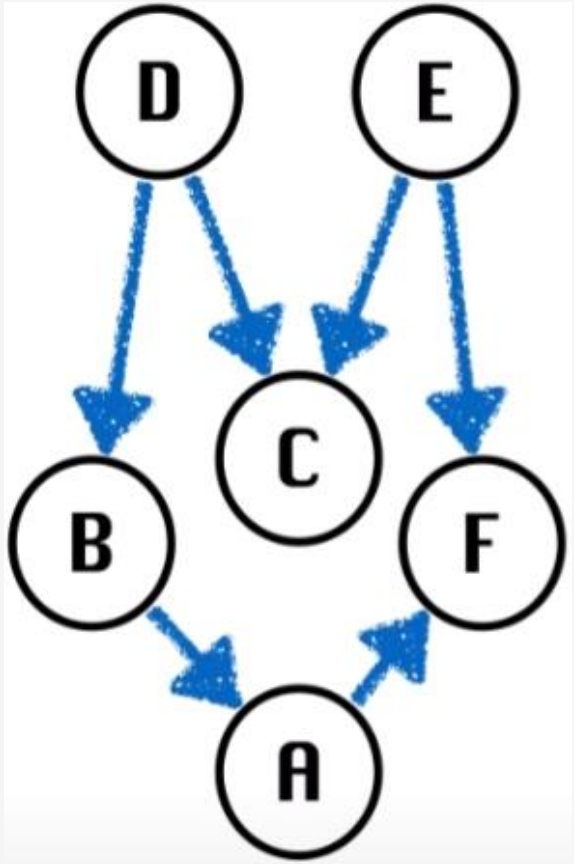


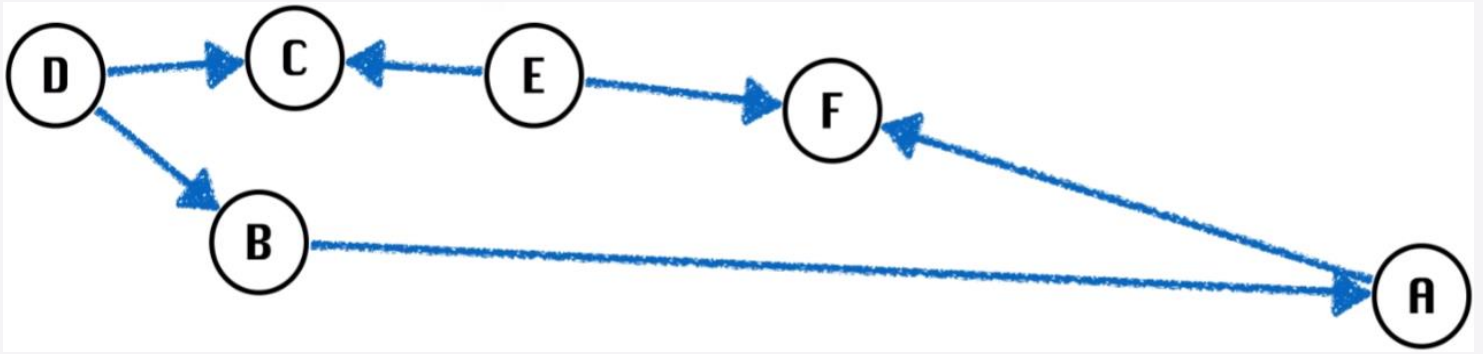
Since **node 1** points to **nodes 2** and **3**, **node 1** appears before them in the ordering. And, since **nodes 2** and **3** both point to **node 4**, they appear before it in the ordering.



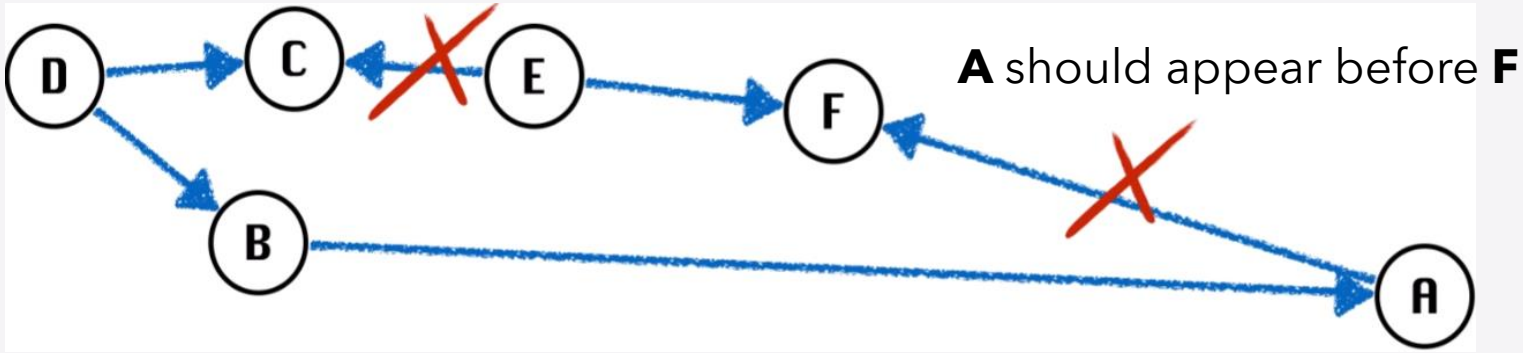
So **[1, 2, 3, 4, 5]** would be a topological ordering of the graph. **Can a graph have more than one valid topological ordering?** **Yes!** In the example above, **[1, 3, 2, 4, 5]** works too.

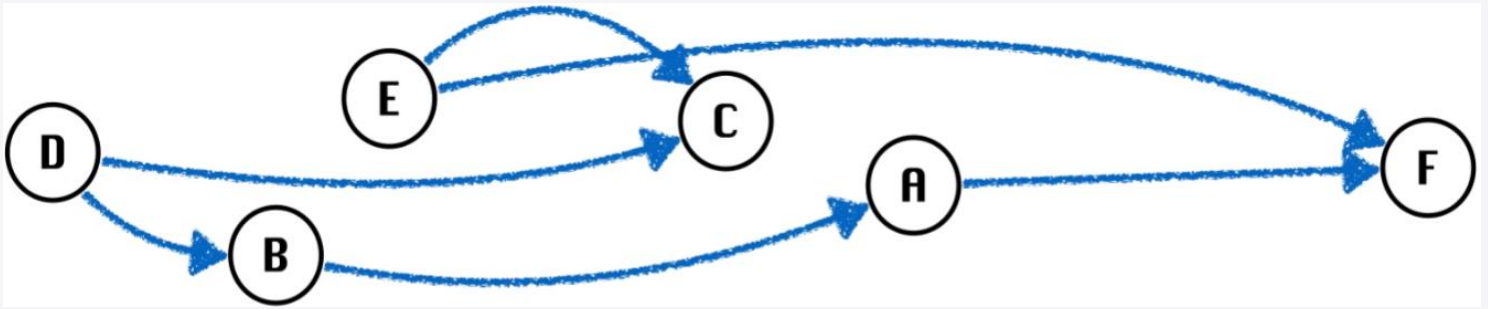
Example





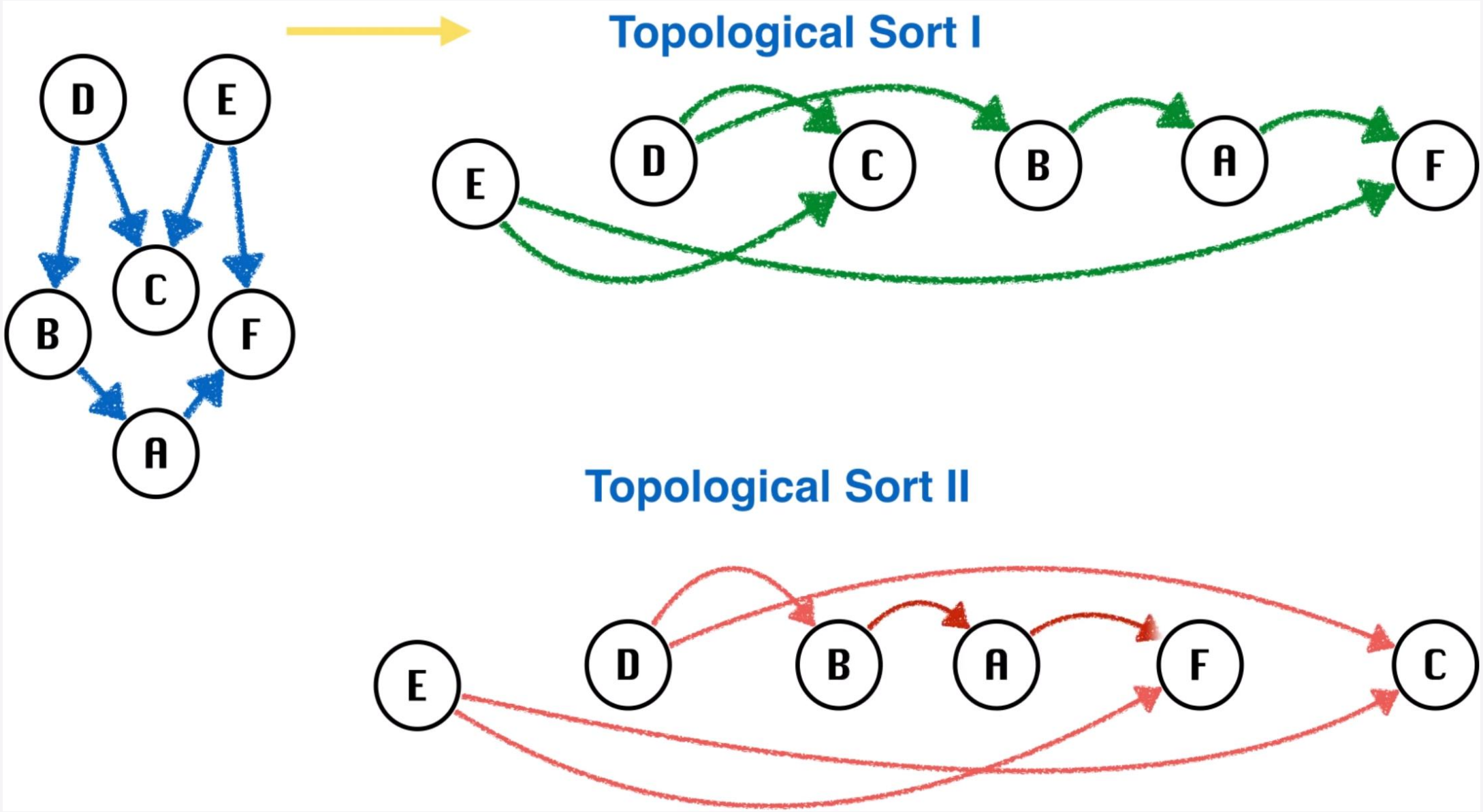
E should appear before **C**





Node F can be a task that has **A** and **E** as prerequisites.
Node A can be a task that has **B** as a prerequisite.

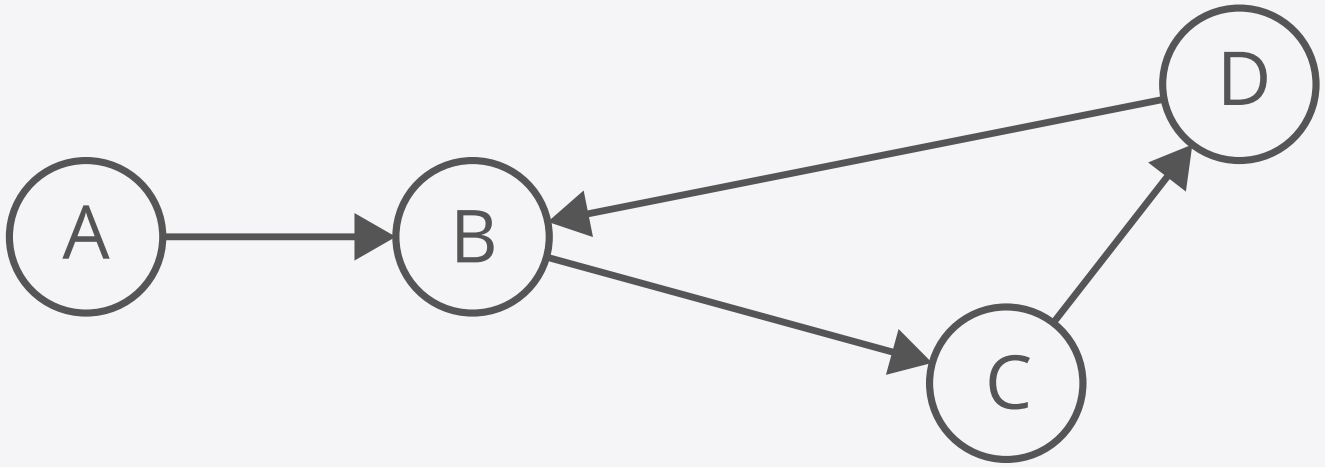
Example


D
E
C
B
F
A
Topological Sort I
E
D
C
B
A
F
Topological Sort II
E
D
B
A
F
C

Cyclic Graphs

Topological sort **is only for DAG**. A topological ordering is possible if and only if the graph has **no directed cycles**.

Look at this **directed graph with a cycle**:

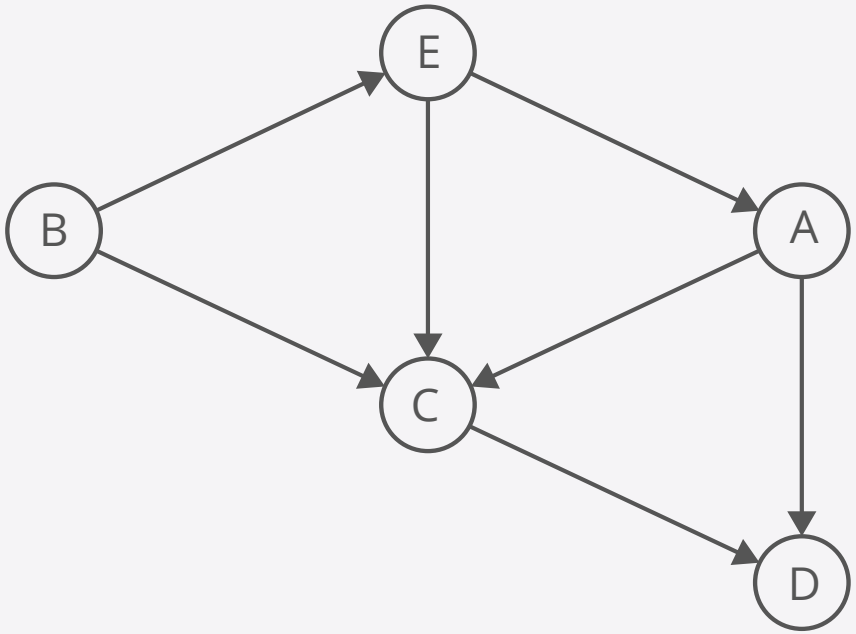


The cycle creates an impossible set of constraints — **B** must be before and after **D** in the ordering.
As a rule, **cyclic graphs don't have valid topological orderings**.

The algorithm

Topological sort is simply a modification of **DFS**.

let's focus on the first node in the topological ordering. That node can't have any incoming directed edges; it must have an **indegree of zero**.

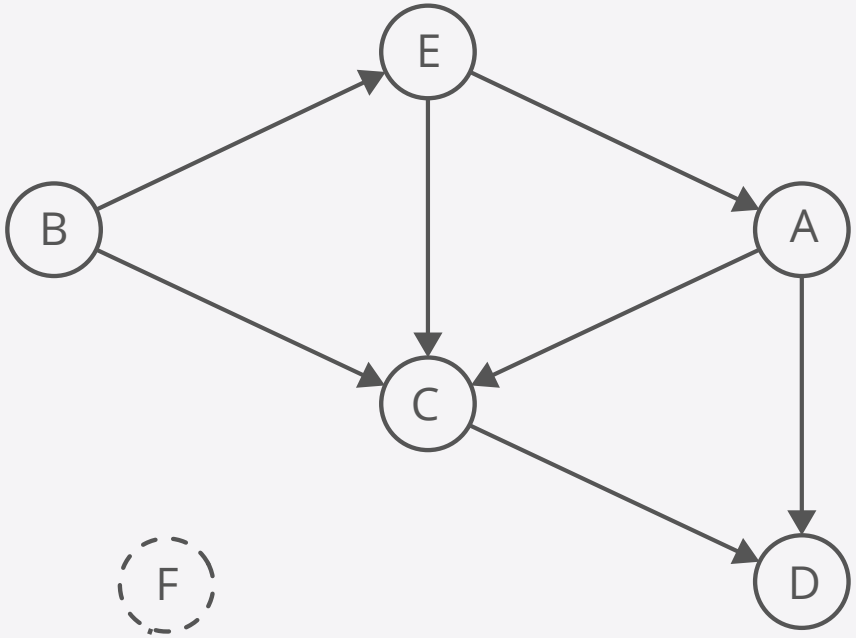


Why?

The algorithm

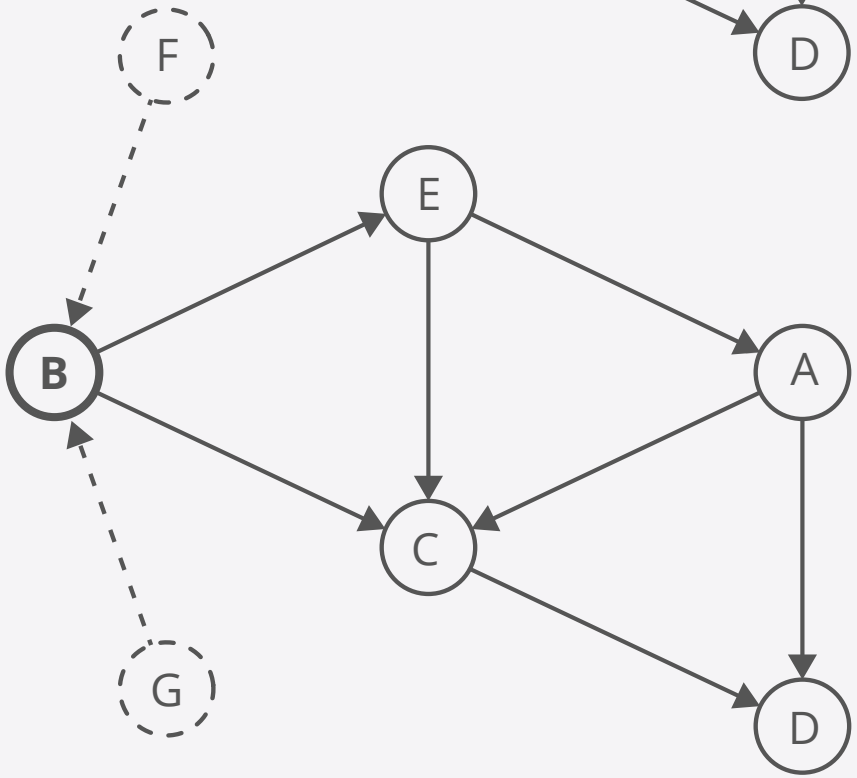
Topological sort is simply a modification of **DFS**.

let's focus on the first node in the topological ordering. That node can't have any incoming directed edges; it must have an **indegree of zero**.



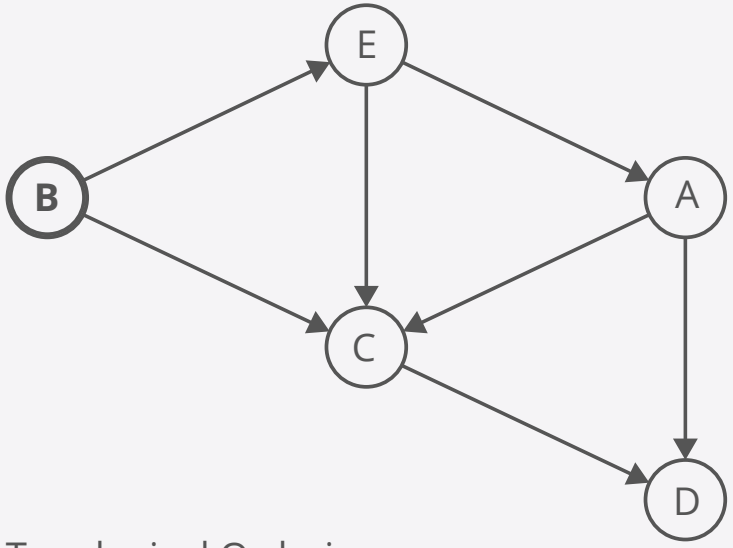
Why?

Because if it had incoming directed edges, then the nodes pointing to it would have to come first.



So, we'll find a **node with an indegree of zero** and add it to the **topological ordering**.

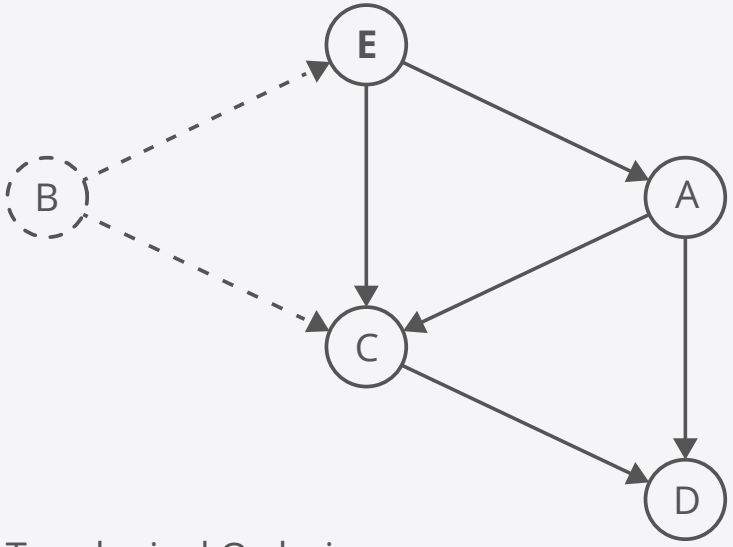
That covers the first node in our topological ordering. **What about the next one?**



Topological Ordering:

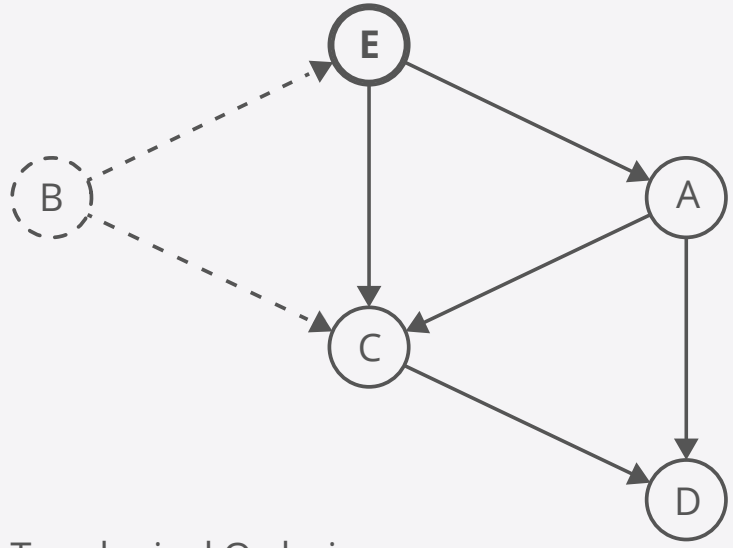
Once a node is added to the topological ordering, **we can take the node, and its outgoing edges, out of the graph**.

Then, we can repeat: look for any node with an indegree of zero and add it to the ordering.



Topological Ordering:

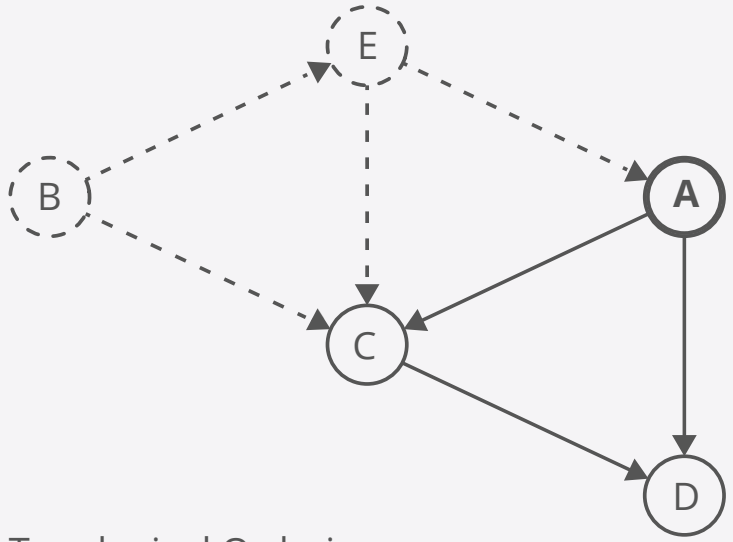
Here's what this looks like on our graph. We'll grab a **node with an indegree of 0**, add it to our **topological ordering** and **remove it from the graph**:



Topological Ordering:

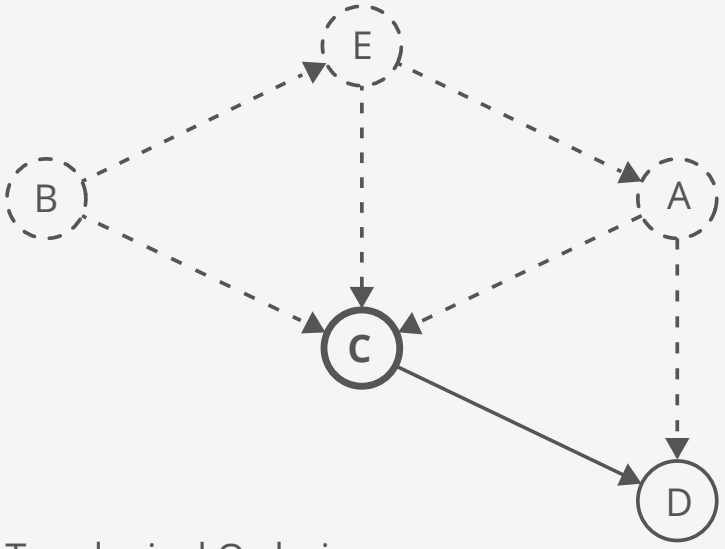
---

and repeat



Topological Ordering:

and repeat

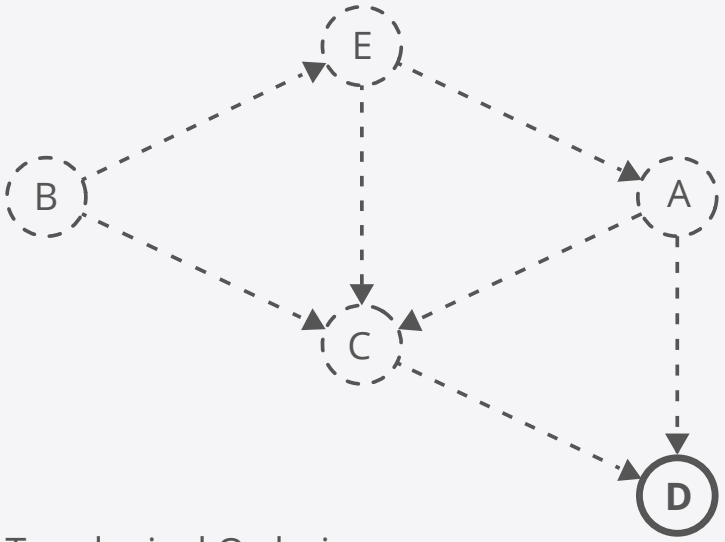


Topological Ordering:

B	E	A	**C**	

---

until we are out of nodes.



Topological Ordering:

B	E	A	C	**D**

Note: *this isn't the only way to produce a topological ordering.*

Implementation

We'll use the strategy we outlined above:

1. **Identify a node with no incoming edges.**
2. **Add that node to the ordering.**
3. **Remove it from the graph.***
4. **Repeat.**

* Instead of actually removing the nodes from the graph (and destroying our input), **we can track each node's indegree**. When we add a node to the topological ordering, we'll **decrement the indegree of that node's neighbors**, representing that those nodes have one fewer incoming edges.

Implementation

We'll use the strategy we outlined above:

1. **Identify a node with no incoming edges.**
2. **Add that node to the ordering.**
3. **Remove it from the graph.***
4. **Repeat.**

We'll keep looping until there aren't any more nodes with indegree zero. This could happen for two reasons:

- **There are no nodes left.** We've taken all of them out of the graph and added them to the topological ordering.
- **There are some nodes left, but they all have incoming edges.** This means **the graph has a cycle**, and no topological ordering exists.

Time Complexity

- **Determine the indegree for each node.** This is **O(E)** time (where **E** is the number of edges), since this involves looking at each directed edge in the graph once.

- **Find nodes with no incoming edges.** This is a simple loop through all the nodes with some number of constant-time appends. **O(V)** time (where **V** is the number of nodes).

- **Add nodes until we run out of nodes with no incoming edges.** This loop could run once for every node—**O(V)** times.

- **Check if we included all nodes or found a cycle.** This is a fast **O(1)** comparison.

All together, the time complexity is **O(V + E).**

Space Complexity

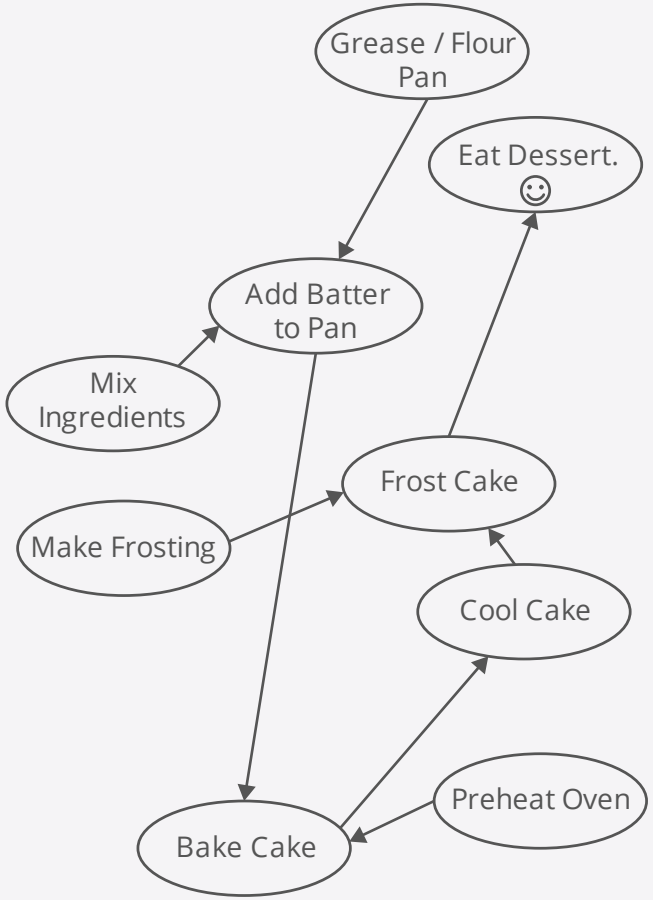
Here are the data structures we created:

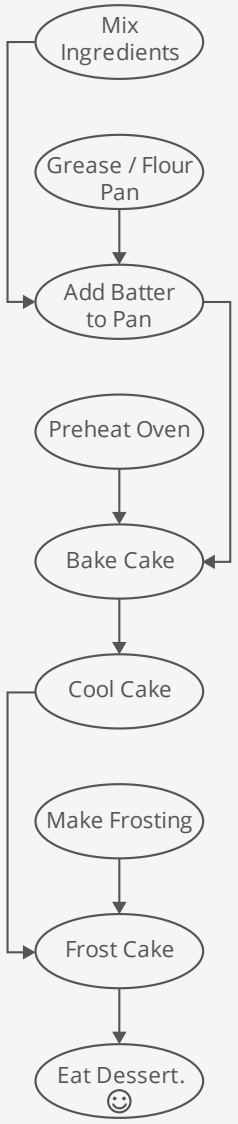
- **indegrees**—this has one entry for each node in the graph, so it's **O(V)** space.
- **nodesWithNoIncomingEdges**—in a graph with no edges, this would start out containing every node, so it's **O(V)** space in the worst case.
- **topologicalOrdering**—in a graph with no cycles, this will eventually have every node. **O(V)** space.

All in all, we have three structures and they're all **O(V)** space. Overall space complexity: **O(V)**

Uses

The most common use for topological sort is ordering steps of a process where some the steps depend on each other.





Act 4.2
- Graph: Complementary algorithms

All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

What do I have to do?

Individually, create a **Graph** class with the variables number of vertices, number of edges, Adj Matrix, Adj list

and the next class function:

- **LoadGraph** (Randomly create a connected DAG)

Input: number of vertices (n)

It must guarantee the creation of an adjacency list of a **connected Direct Acyclic Network (DAG)**. **You cannot use any of the other functions of this activity for this purpose**.

The application must ask the user for the number of vertices (**n**)

All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

- **IsTree**

Output: A boolean value that tells whether the graph is a tree or not.

Say if the DAG is a tree or not

- **TopologicalSort**

Print the data in topological order.

- **BipartiteGraph**

Output: A boolean value that tells whether the graph is a bipartite graph or not.

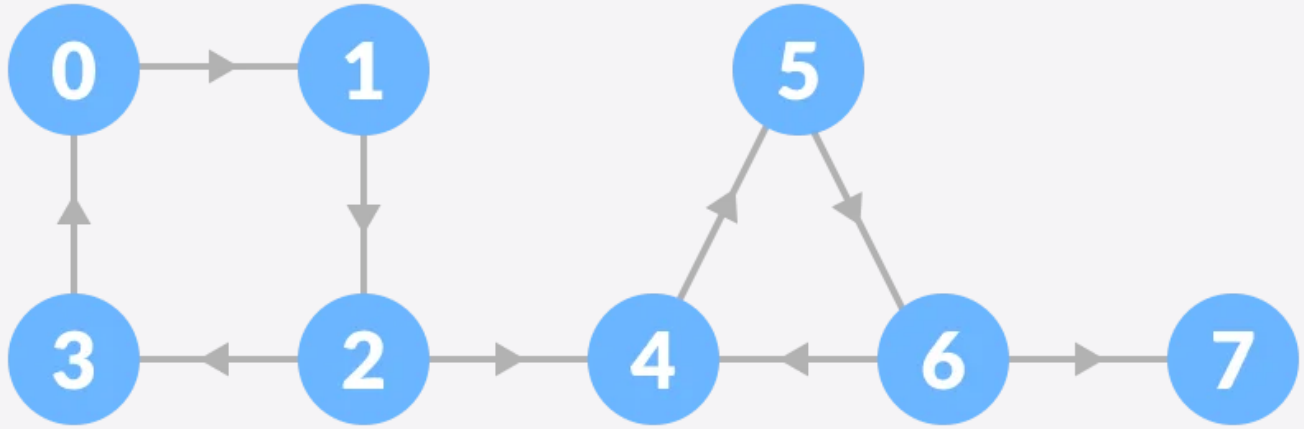
Say if DAG is a bipartite graph or not

Strongly Connected Components

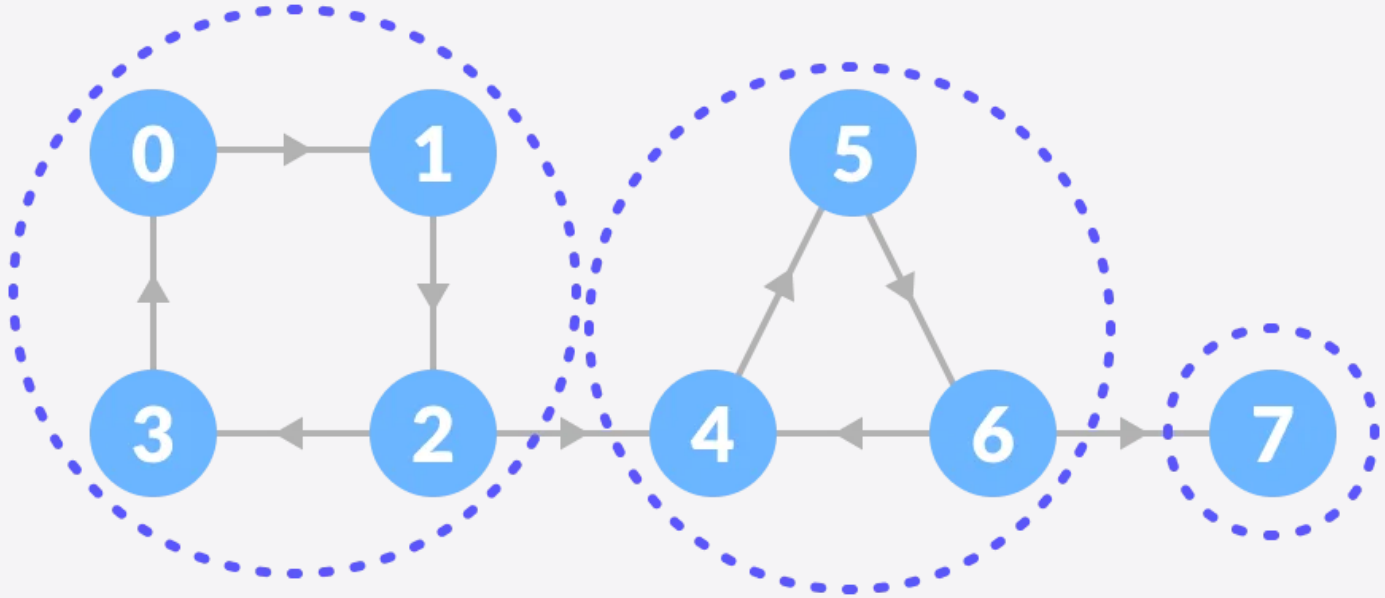
A strongly connected component is the **portion of a directed graph** in which there is a path from each vertex to another vertex. **It is applicable only on a directed graph**.

For example:

Let us take the graph below.



The strongly connected components of the above graph are:



You can observe that in the first strongly connected component, every vertex can reach the other vertex through the directed path.

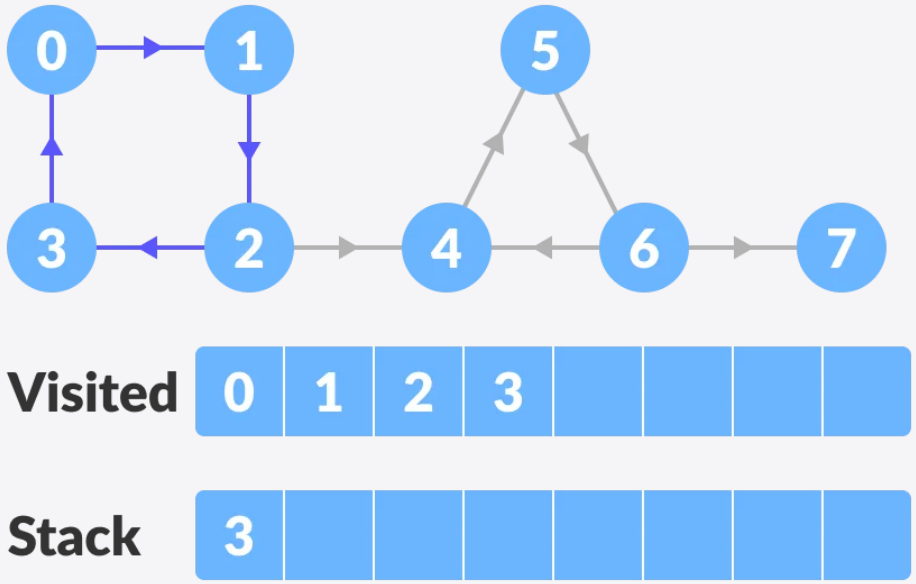
These components can be found using **Kosaraju's Algorithm**.

Kosaraju's Algorithm

1. Perform a depth first search (DFS) on the whole graph.

Let us start from **0**, visit all its child vertices, and **mark the visited vertices** as done.

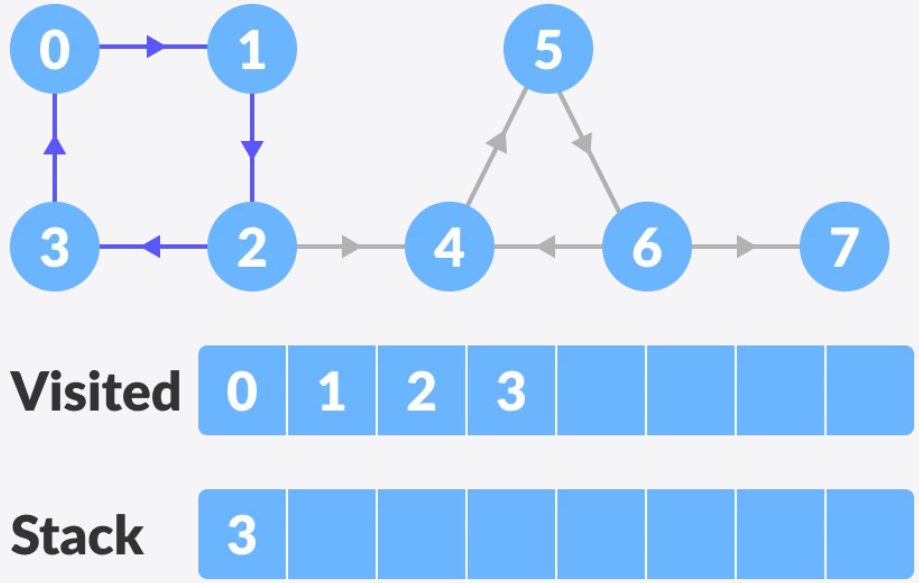
If a **vertex is already visited**, then push this vertex to the stack.



Kosaraju's Algorithm

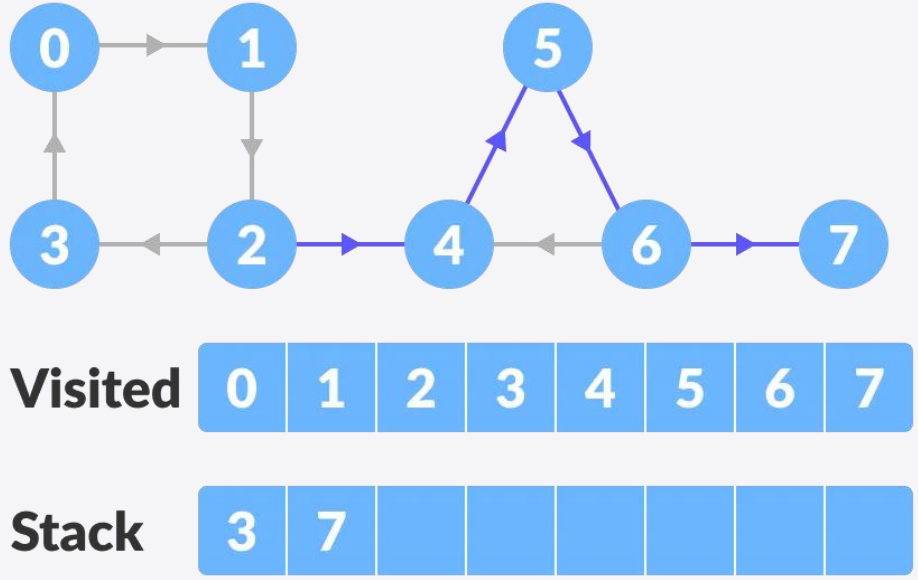
1. Perform a depth first search (DFS) on the whole graph.

Let us start from **0**, visit all its child vertices, and **mark the visited vertices** as done.

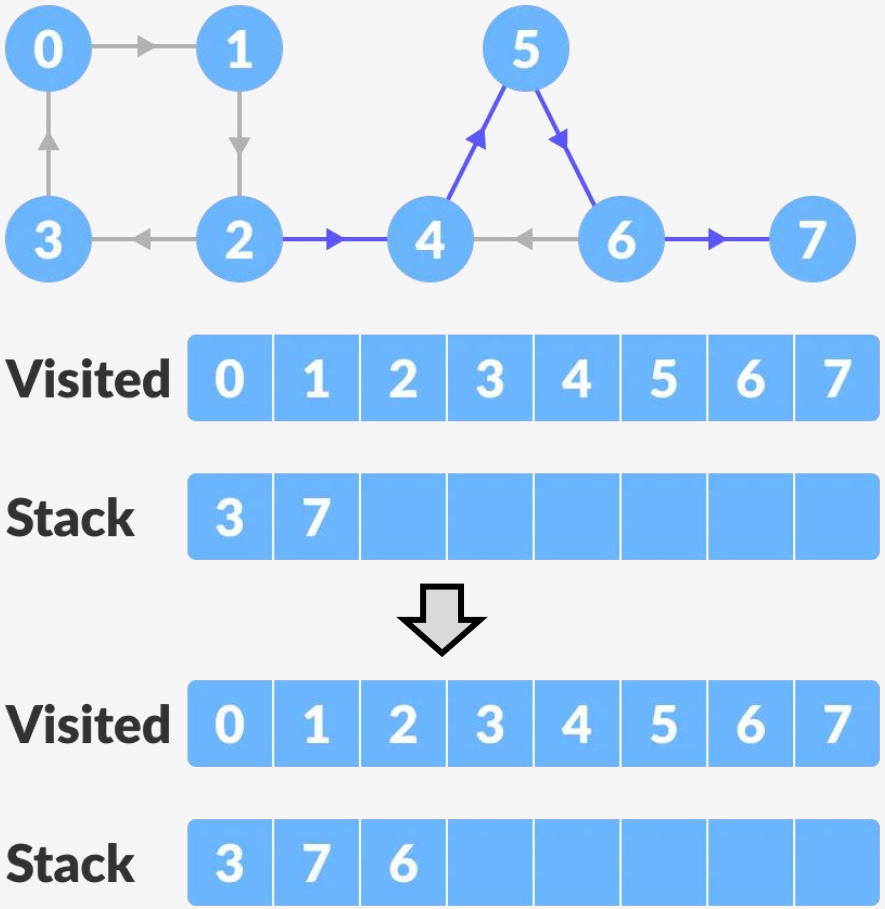


If a **vertex is already visited**, then push this vertex to the stack.

Go to the previous vertex (**2**) and visit its child vertices (4, 5, 6 and 7) sequentially. Since there is nowhere to go from **7**, **push it into the stack**.

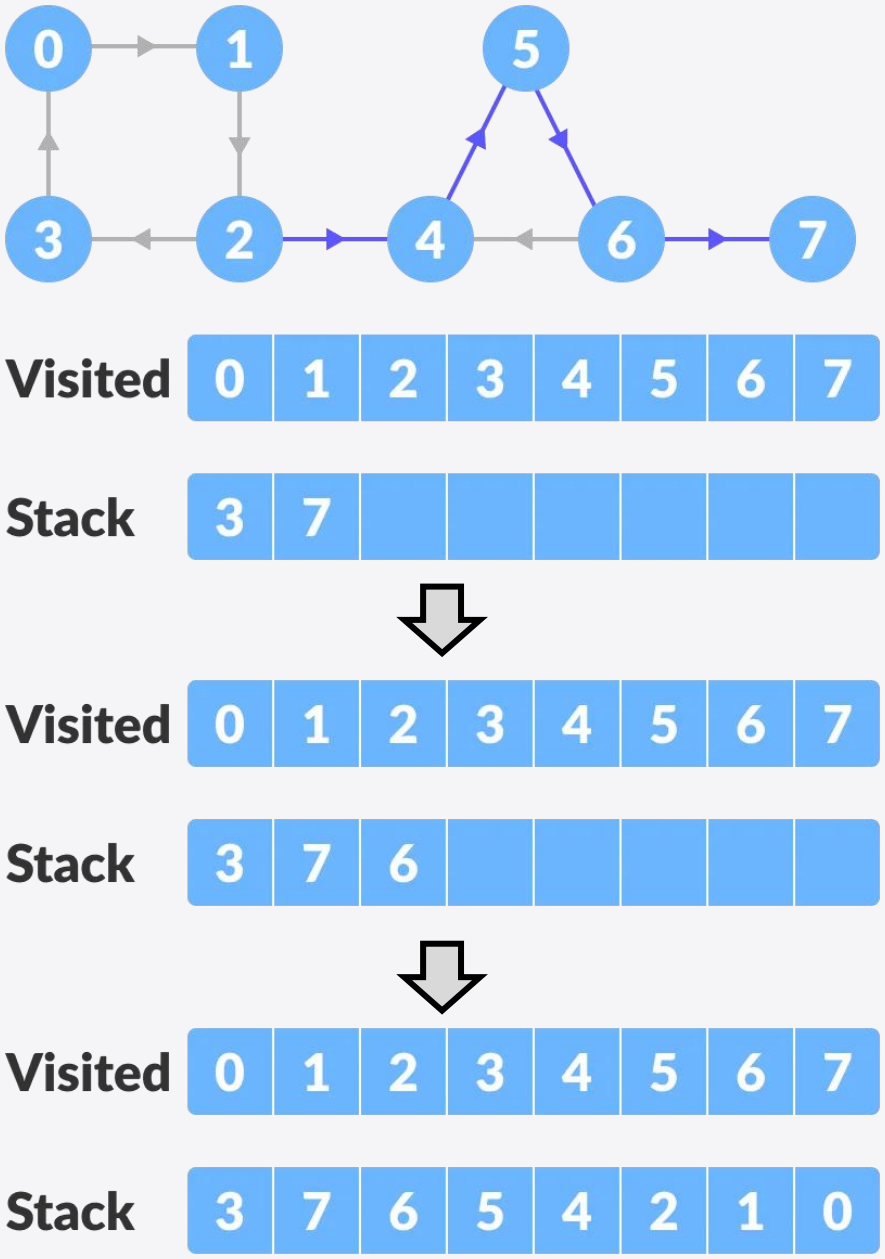


Go to the previous vertex (**6**) and visit its child vertices. But all its child vertices are visited, so push it into the stack.

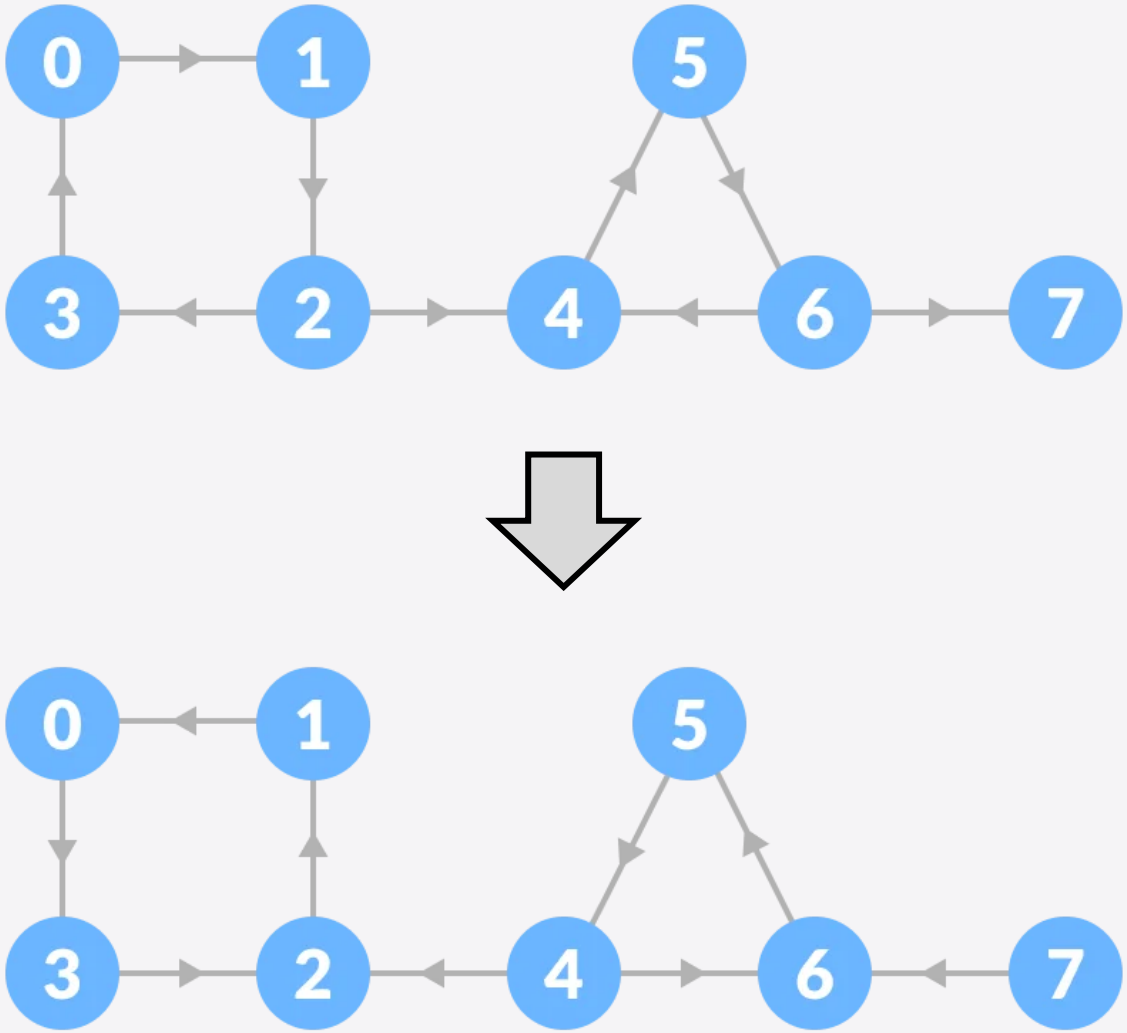


Go to the previous vertex (**6**) and visit its child vertices. But all its child vertices are visited, so push it into the stack.

Similarly, a final stack is created.



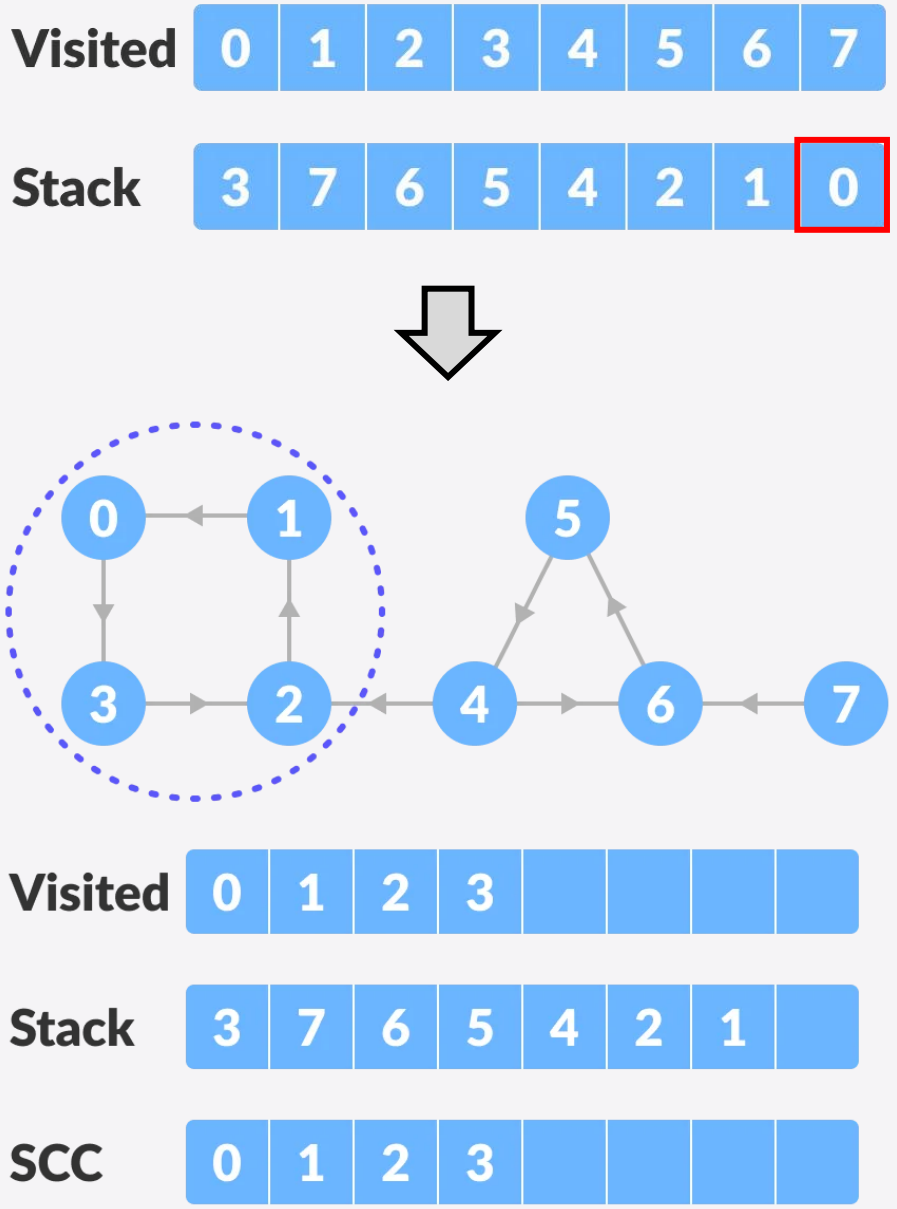
2. Reverse the original graph.



3. Perform depth-first search on the reversed graph.

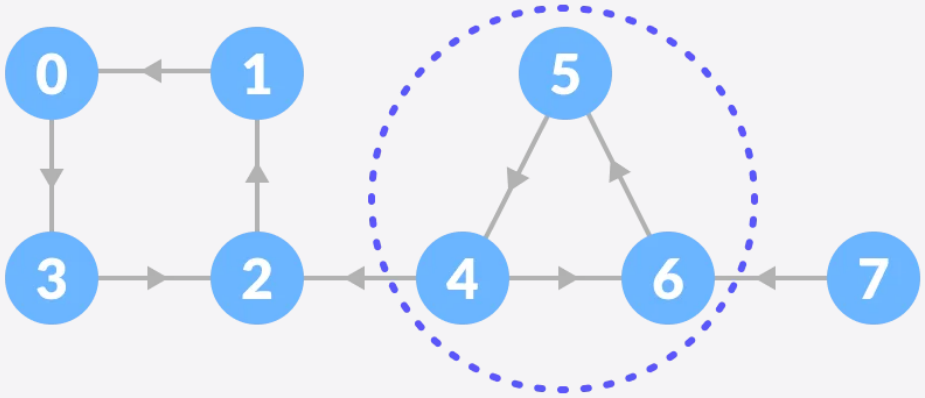
Start from the top vertex of the stack. Traverse through all its child vertices. **Once the already visited vertex is reached**, one strongly connected component is formed.

For example: Pop **vertex 0** from the stack. Starting from 0, traverse through its child vertices (0, 1, 2, 3 in sequence) and **mark them as visited**. The child of **3 is already visited**, so these visited vertices form one strongly connected component.

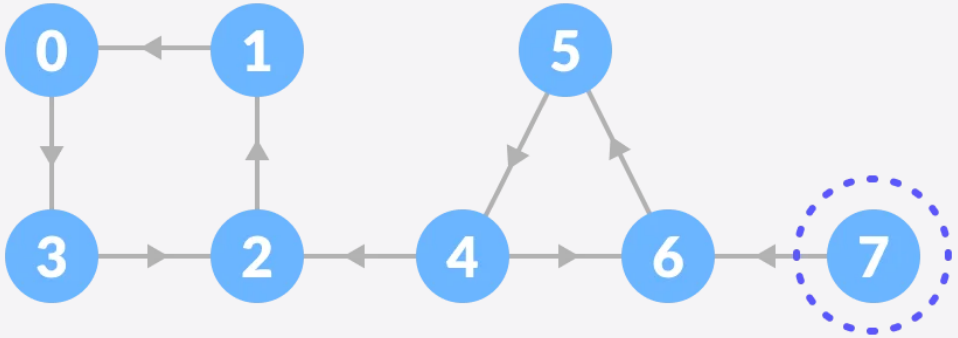


Go to the stack and pop the top vertex if already visited.

Otherwise, choose the top vertex from the stack and traverse through its child vertices as presented.

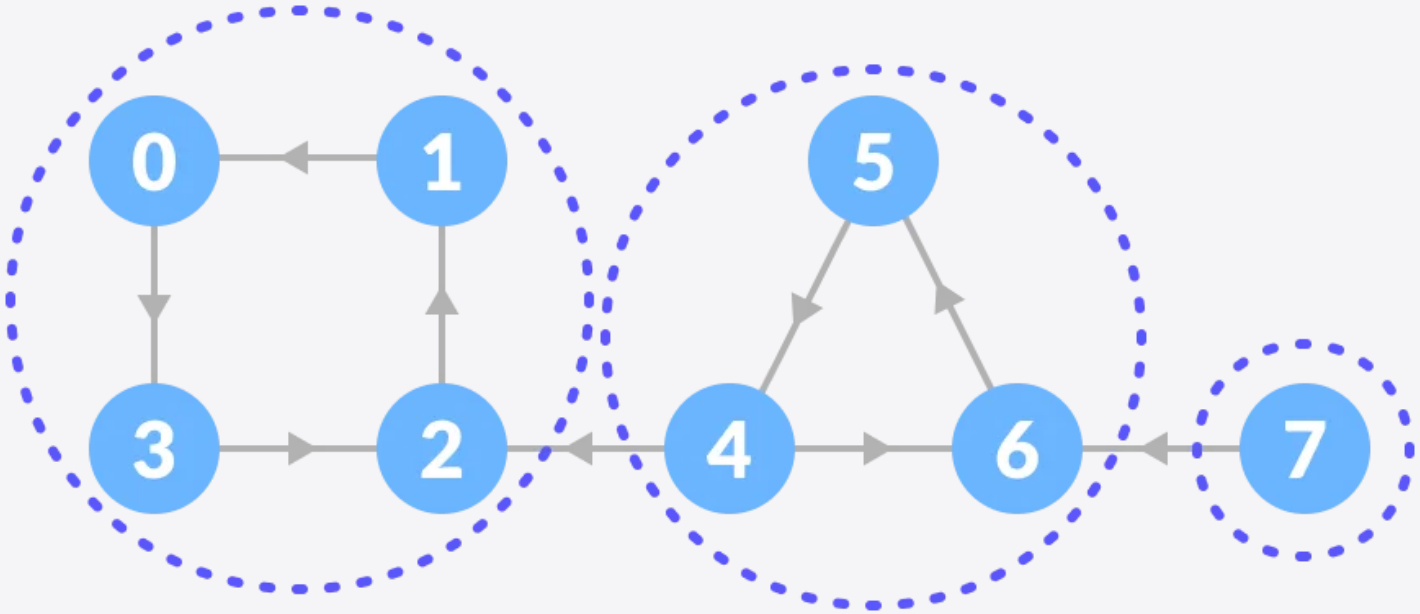


Visited	0	1	2	3	4	5	6	
Stack	3	7	6	5				
SCC	4	5	6					



Visited	0	1	2	3	4	5	6	
Stack								
SCC	7							

4. Thus, the strongly connected components are:



Kosaraju's Algorithm Complexity

Kosaraju's algorithm runs in linear time i.e., **O(V+E)**.

Strongly Connected Components Applications

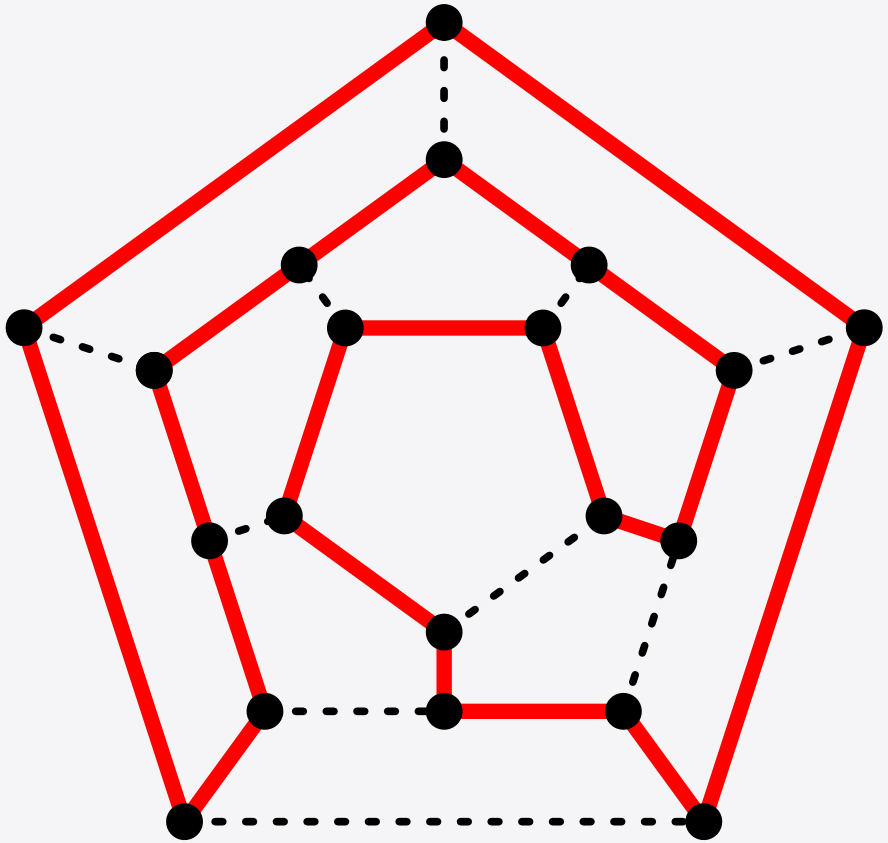
- Vehicle routing applications
- Maps
- Model-checking in formal verification

Hamiltonian & Euler cycle

Hamiltonian path

A Hamiltonian path is a path that visits each vertex of the graph exactly once. A Hamiltonian path can exist both in a **directed** and **undirected graph**.

So, if a path covers all the vertices of a given graph without repeating any vertex, then it's a **Hamiltonian path**.

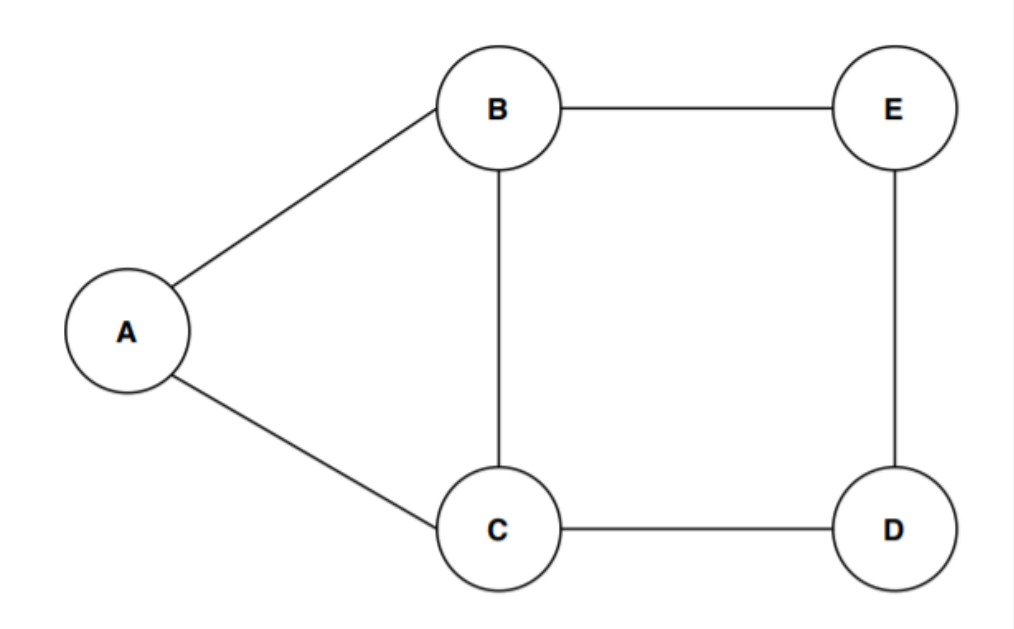


Hamiltonian path - Example

Let’s take some graphs:

First, we’ll try empirically by taking a random path in **G** above, say **ABCDE**.

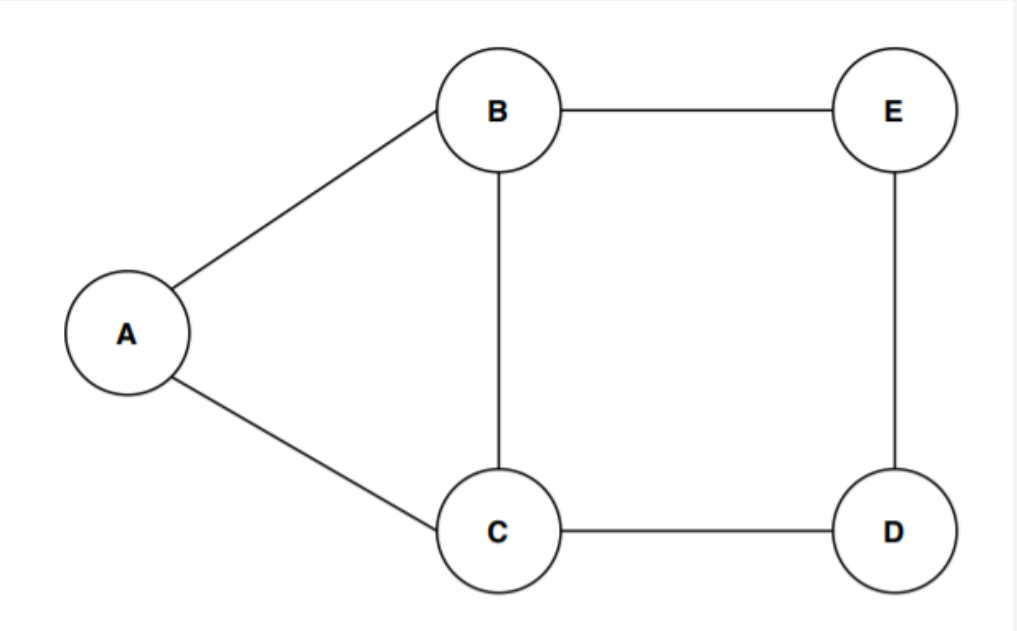
But is it Hamiltonian?



Hamiltonian path - Example

Let's take some graphs:

First, we'll try empirically by taking a random path in **G** above, say **ABCDE**.



But is it Hamiltonian?

Following the definition of a Hamiltonian path, our path covers all the vertices of our graph without visiting any vertex more than once. **Hence, we can say that the path is a Hamiltonian path**.

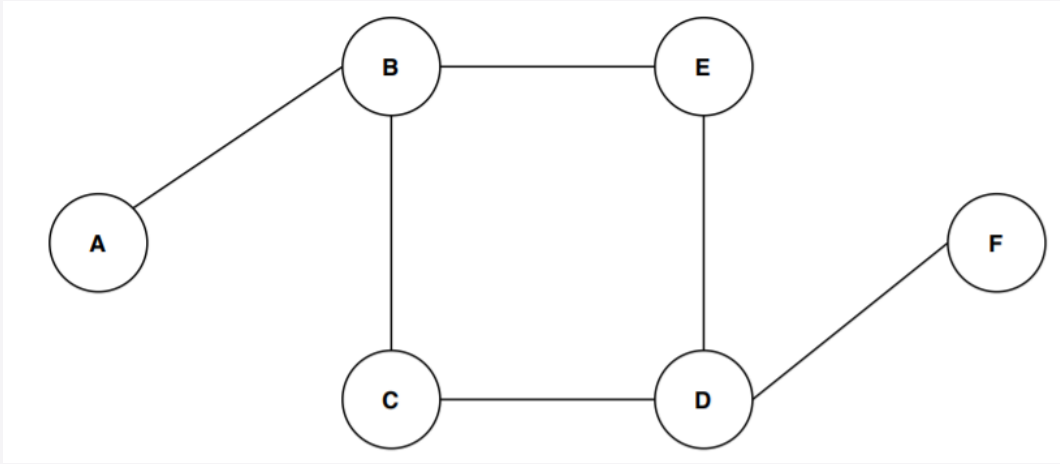
What about **ACBED**?

Hamiltonian path – Example 2

Let's take another graph and call it **G2**:

Let's try our method again and take the random path **ABCDEDF**.

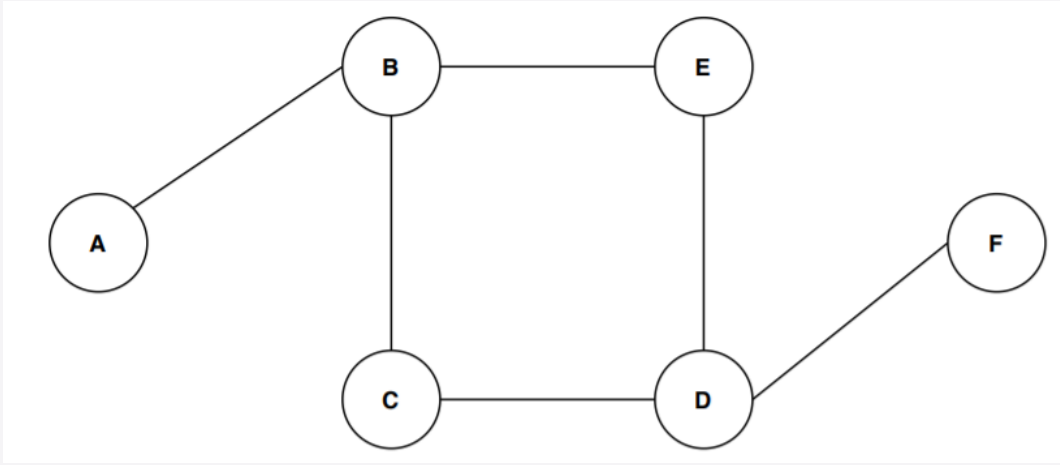
Is it Hamiltonian?



Hamiltonian path – Example 2

Let's take another graph and call it **G2**:

Let's try our method again and take the random path **ABCDEDF**.



Is it Hamiltonian?

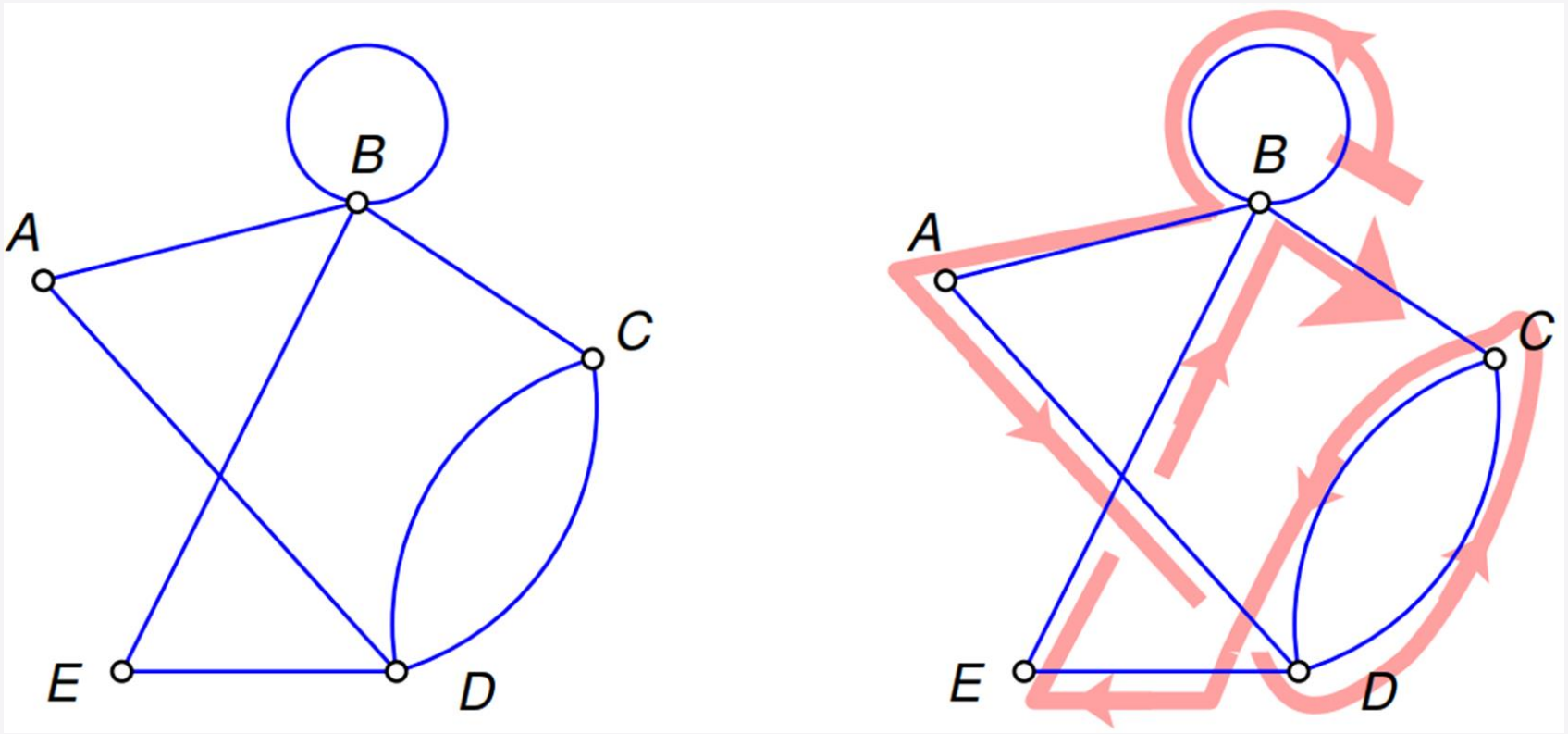
According to the definition, a path shouldn't contain a vertex more than once. Here we can see the vertex is repeated twice. So, the **path ABCDEDF** is **not a Hamiltonian path**.

In fact, there can't be any Hamiltonian path for this graph.

Eulerian path

Eulerian path is a walk* where **we must visit each edge only once, but we can revisit vertices**. An Euler path can be found in a directed as well as in an undirected graph.

* A **walk** simply consists of a sequence of vertices and edges.

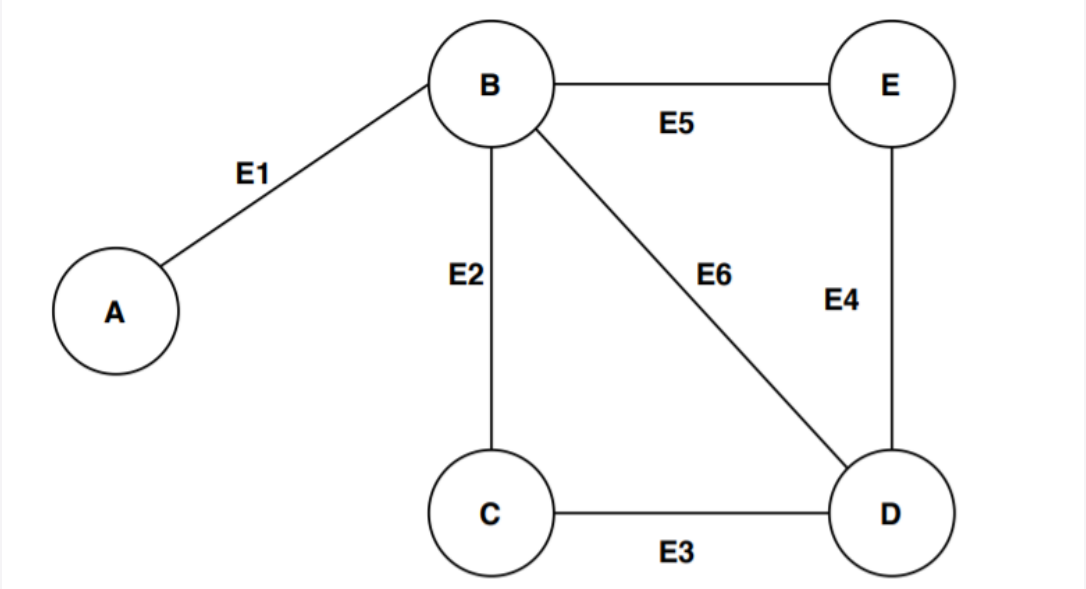


Euler path - Example

Now, let's try and do the same for Euler paths.

Let's define a walk in the graph **G3**. Our randomly picked sample walk in **G3** is **ABCDBED**.

Is it follows the Eluer path definition?



Euler path - Example

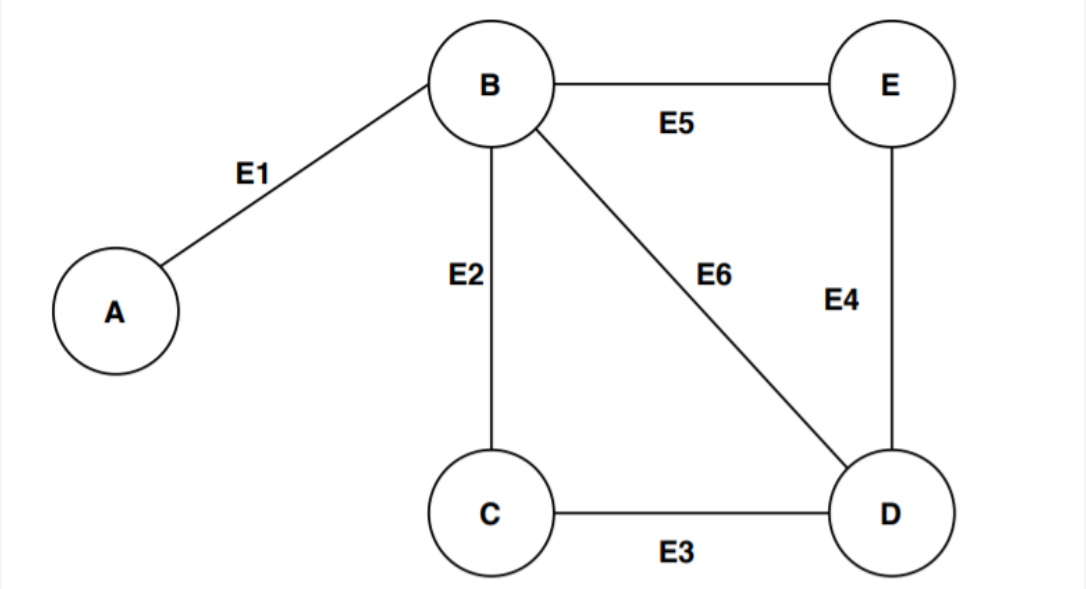
Now, let's try and do the same for Euler paths.

Let's define a walk in the graph **G3**. Our randomly picked sample walk in **G3** is **ABCDBED**.

Is it follows the Eluer path definition?

The walk covers the edges **E1-E2-E3-E6-E5-E4**.

We can see **our walk covers all the edges of the graph without repeating** any edges.

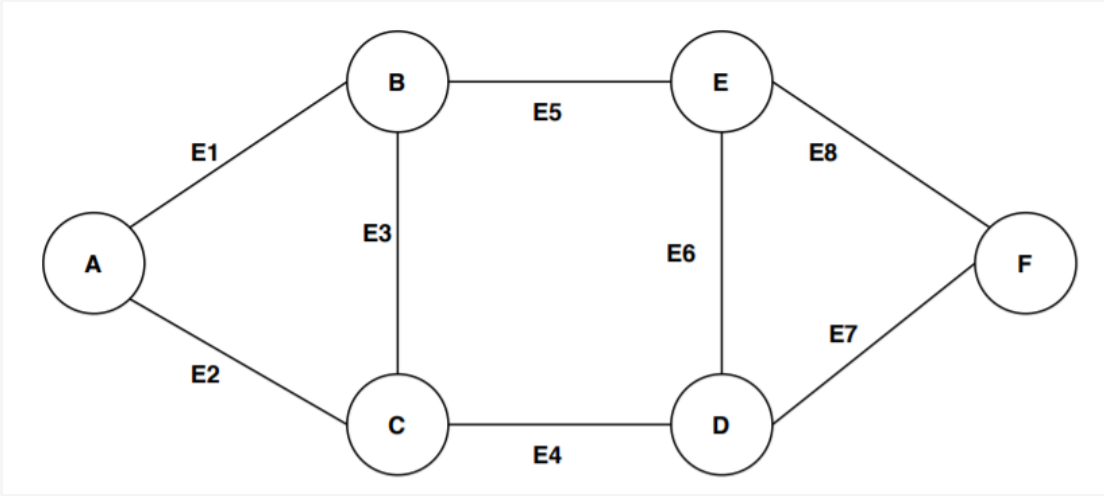


Euler path – Example 2

It’s time to consider a different graph:

We’re picking a random walk **ABCDEFDCABE** of the graph **G4**.

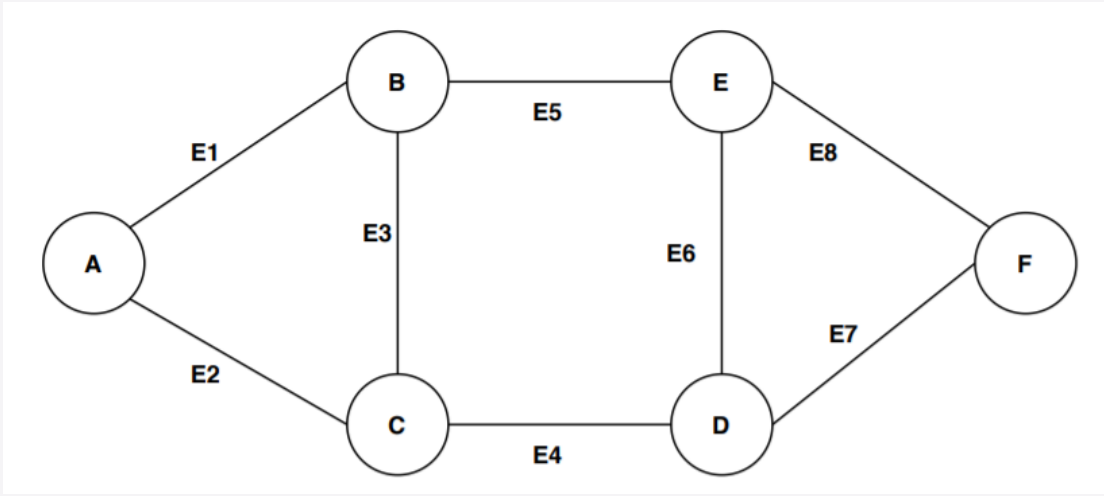
Is it Eulerian?



Euler path – Example 2

It’s time to consider a different graph:

We’re picking a random walk **ABCDEFDCABE** of the graph **G4**.



Is it Eulerian?

The edges it covers are **E1-E3-E4-E6-E8-E7-E4-E2-E1-E5**.

The walk covers all the edges of the graph but there is repetition of the edges **E1** and **E2**, **against the definition of Eulerian path**.

It is not possible to cover all the edges without repeating vertices in **G4**.

Applications (1/2)

A business traveler leaves every morning from his home and needs to visit a few customers and then go back home.

How should he go about minimizing the total distance he travels?

(We are supposing that the distances are known.)

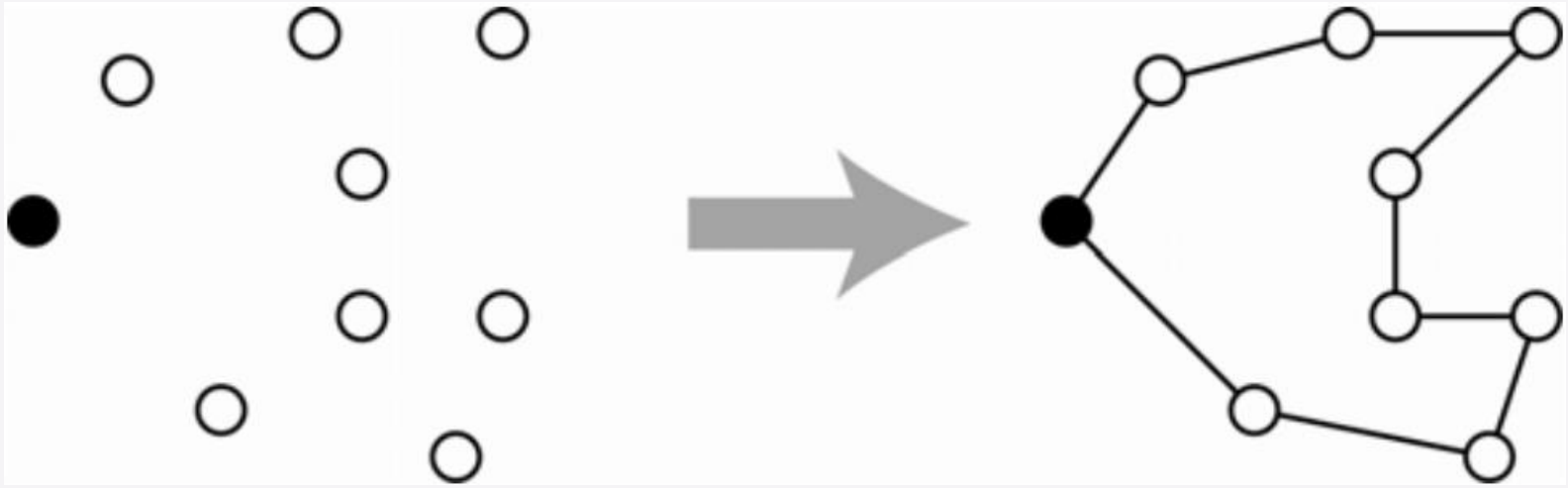
Applications (1/2)

A business traveler leaves every morning from his home and needs to visit a few customers and then go back home.

How should he go about minimizing the total distance he travels?

(We are supposing that the distances are known.)

Here we are looking for a **Hamiltonian cycle** of the minimum possibly length.



Applications (2/2)

A garbage collection truck leaves from the depot and needs to travel along each street to carry out its garbage collection.

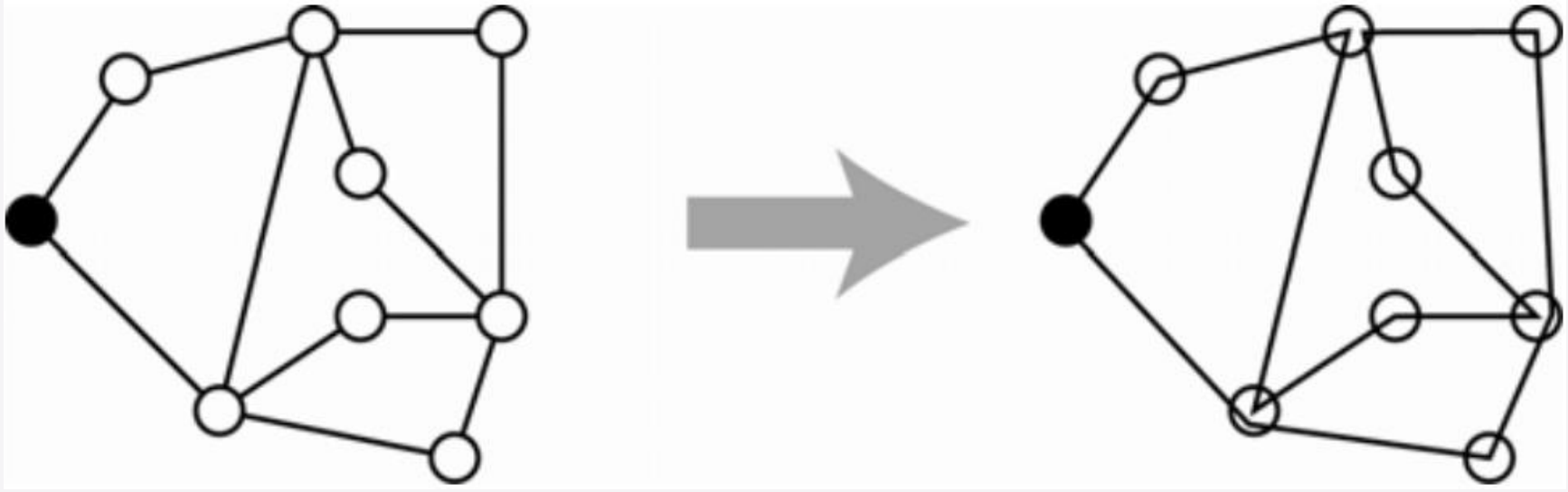
How should it do that?

Applications (2/2)

A garbage collection truck leaves from the depot and needs to travel along each street to carry out its garbage collection.

How should it do that?

Here, we are looking for a **Eulerian cycle**.



Act 4.3 - Comprehensive Graph Activity (Competence evidence)

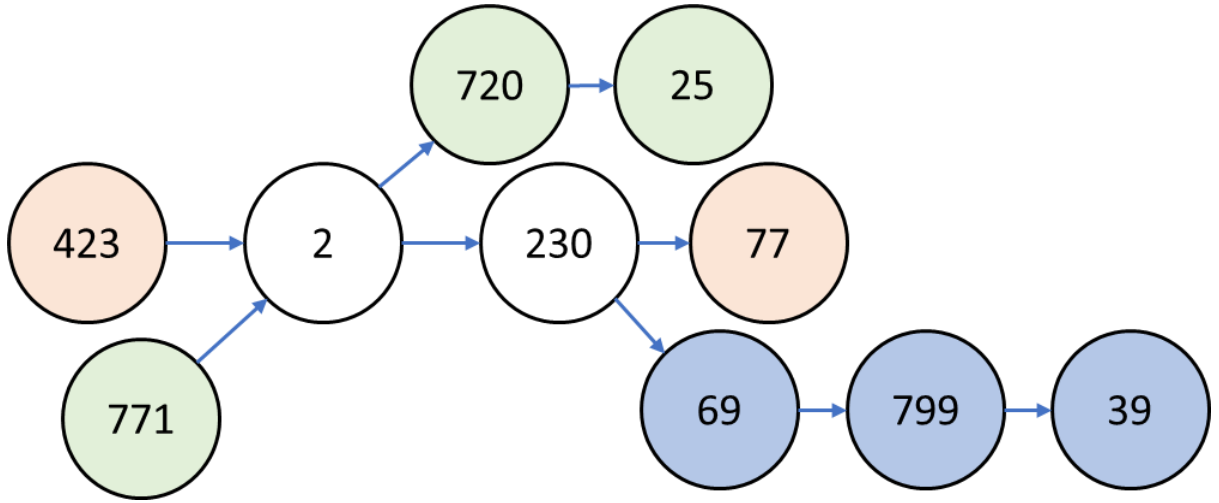
All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

What do we have to do?

In Teams of three, create a **Graph** class:

Open the input file **bitacora.txt**, use the adjacency Matrix / list to organized the IP addresses like a directed graph (each section of the IP address corresponds to a vertex). For example,

```
Oct 9 10:32:24 423.2.230.77:6166 Failed password for illegal user guest
Sep 5 06:00:34 771.2.720.25:4731 Failed password for root
Sep 5 00:41:32 230.69.799.39:5288 Failed password for admin
```



All functionalities must be documented. As part of the documentation, the complexity of each of them must be included.

The document must describe the importance and efficiency of the use of graphs to get important insights of the relationship between its vertices

- **Determine the out-degree of each node.**
- **Display the TOP-10 vertices (nodes) with more out-degree in the format:**
  - **IP segment: _, Out-degree value:_**
- **On which IP address is the boot master presumably located? Display the input File entries with the IP addresses that contain the vertex (IP address segment) with the highest out-grade.**