**What is a Graph?**

The lesson explains multiple types and terminologies used in graphs (from basic to complex).

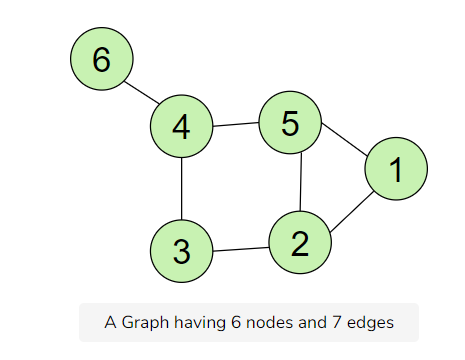
**We'll cover the following**

* + [Introduction](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Introduction-)
    - [🔍 How do you Define a Graph?](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#%F0%9F%94%8D-How-do-you-Define-a-Graph?-)
  + [Vertex:](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Vertex:-)
  + [Edge:](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Edge:-)
  + [Types of Graphs](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Types-of-Graphs-)
    - [1. Undirected Graph](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#1.-Undirected-Graph-)
    - [2. Directed Graph](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#2.-Directed-Graph-)
  + [Graph Terminologies](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Graph-Terminologies-)

**Introduction**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Introduction-)

**🔍 How do you Define a Graph?**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#%F0%9F%94%8D-How-do-you-Define-a-Graph?-)

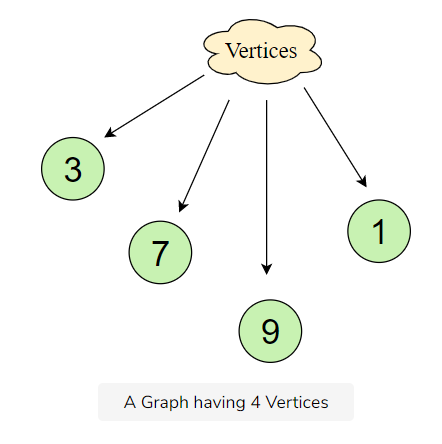
A graph is a set of **vertices** (nodes) that are connected to each other via **edges** in the form of a network.



A Graph having 6 nodes and 7 edges

**Vertex:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Vertex:-)

The structures for storing data in a graph, represented in the form of *Nodes* (1,3,7…), are also called *Vertices*

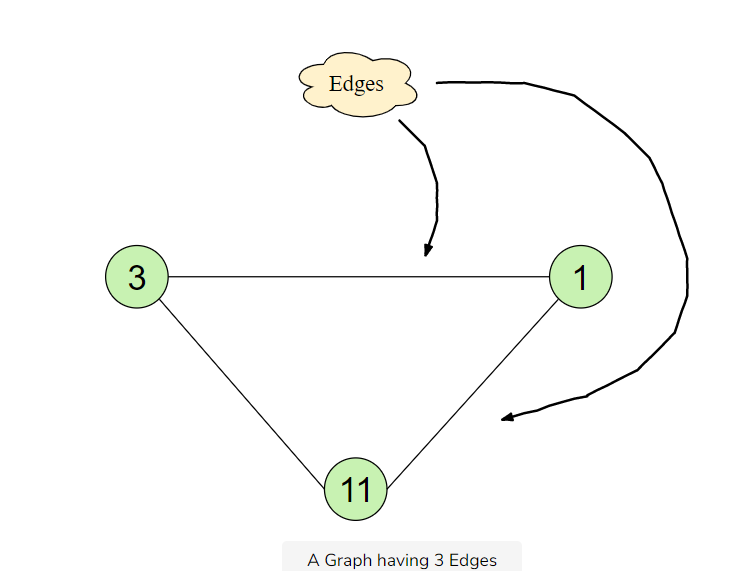


A Graph having 4 Vertices

**Edge:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Edge:-)

A **pair(x,y)** is called an **edge**, which indicates that vertex **x** is connected to vertex **y**. An edge may contain weight/cost, showing how much cost is required to traverse from vertex **x** to **y**.

Edges are usually represented using Straight lines.



A Graph having 3 Edges

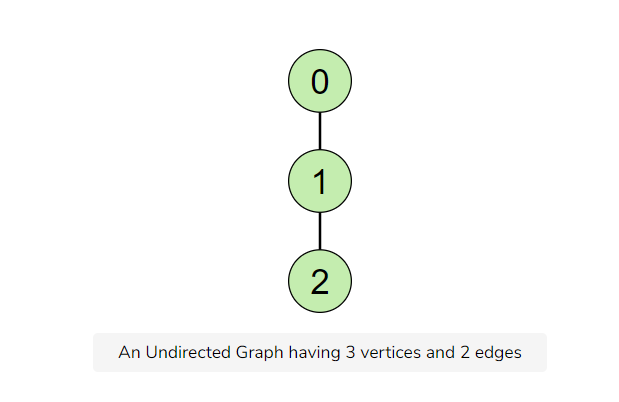
**Types of Graphs**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Types-of-Graphs-)

There are two common types of graphs:

1. Undirected
2. Directed

**1. Undirected Graph**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#1.-Undirected-Graph-)

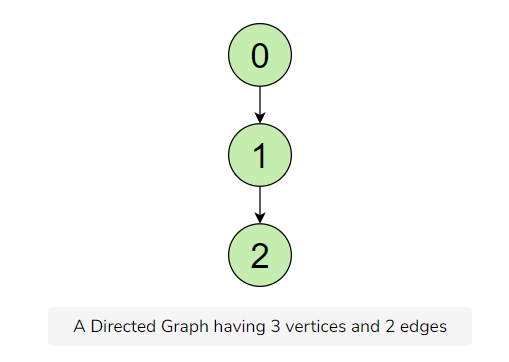
In an undirected graph, the edges are bi-directional by default; for example, with the **pair (0,1)**, it means there exists an edge between vertex **0** and **1** without any specific direction. You can go from vertex **0** to **1**, or vice versa.



An Undirected Graph having 3 vertices and 2 edges

**2. Directed Graph**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#2.-Directed-Graph-)

In a directed graph, the edges are unidirectional; for example, with *the pair (0,1)*, it means there exists an edge from vertex *0* towards vertex *1*, and the only way to traverse is to go from *0* to *1*.



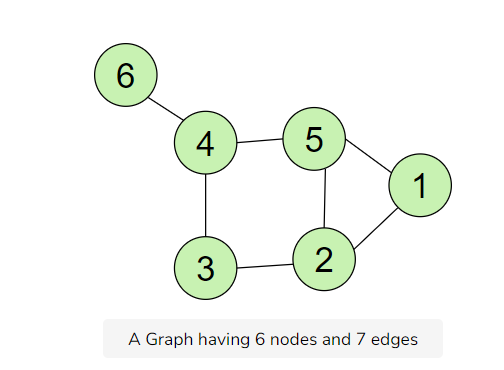
A Directed Graph having 3 vertices and 2 edges

**Graph Terminologies**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/gxQ283421q9#Graph-Terminologies-)

**1. Degree of Vertex**: Total Number of edges connected to a vertex.

Consider the figure given below:

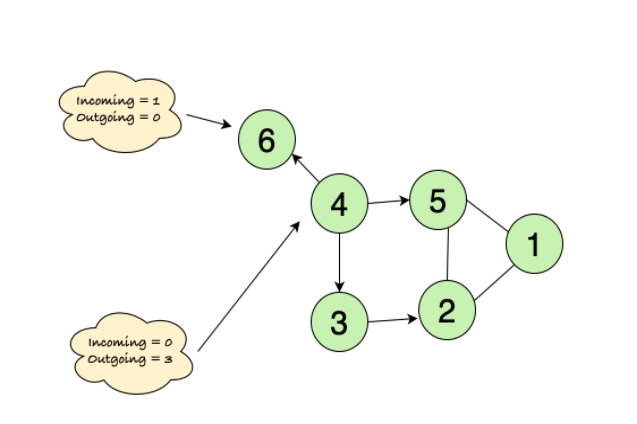
=> Degree of 5 = 3  
=> Degree of 1 = 2  
=> Degree of 6 = 1



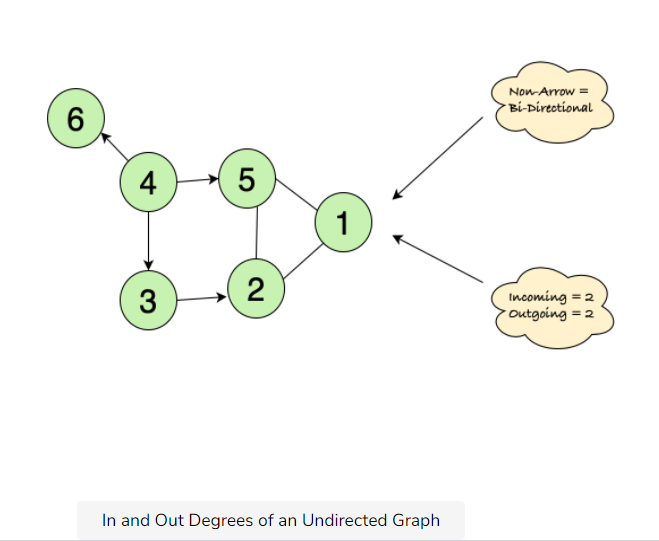
A Graph having 6 nodes and 7 edges

There are two types of degrees:

* **In-Degree of Vertex**: Total Number of incoming edges connected to a vertex.
* **Out-Degree of Vertex**: Total Number of outgoing edges connected to a vertex.

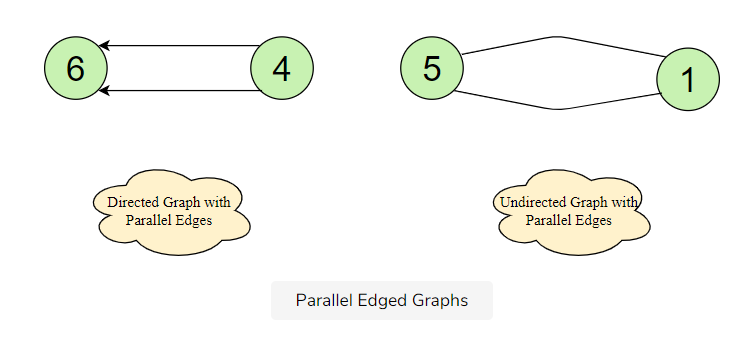
1 1

In and Out Degrees of a Directed Graph



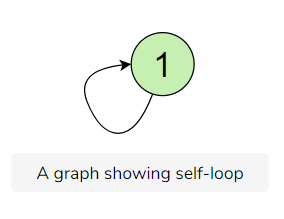
In and Out Degrees of an Undirected Graph

**2. Parallel Edges**: Two or more edges that are incident to the same two vertices.



Parallel Edged Graphs

**3. Self Loop**: Same endpoints of an edge, e.g., *pair (x,x)*.



A graph showing self-loop

In the next lesson, we’re going to learn ways to represent a graph data structure, which will help us in implementing it.

**Representation of Graphs**

The lesson elaborates on how a graph can be represented using the Adjacency List and Adjacency Matrix.

**We'll cover the following**

* + [Ways to Represent a Graph](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Ways-to-Represent-a-Graph-)
  + [Adjacency Matrix](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Adjacency-Matrix-)
  + [Explanation:](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Explanation:-)
    - [Undirected Graph:](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Undirected-Graph:-)
    - [Directed Graph:](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Directed-Graph:-)
  + [2. Adjacency List](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#2.-Adjacency-List-)
    - [Undirected Graph:](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Undirected-Graph:-)
    - [Directed Graph:](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Directed-Graph:-)

**Ways to Represent a Graph**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Ways-to-Represent-a-Graph-)

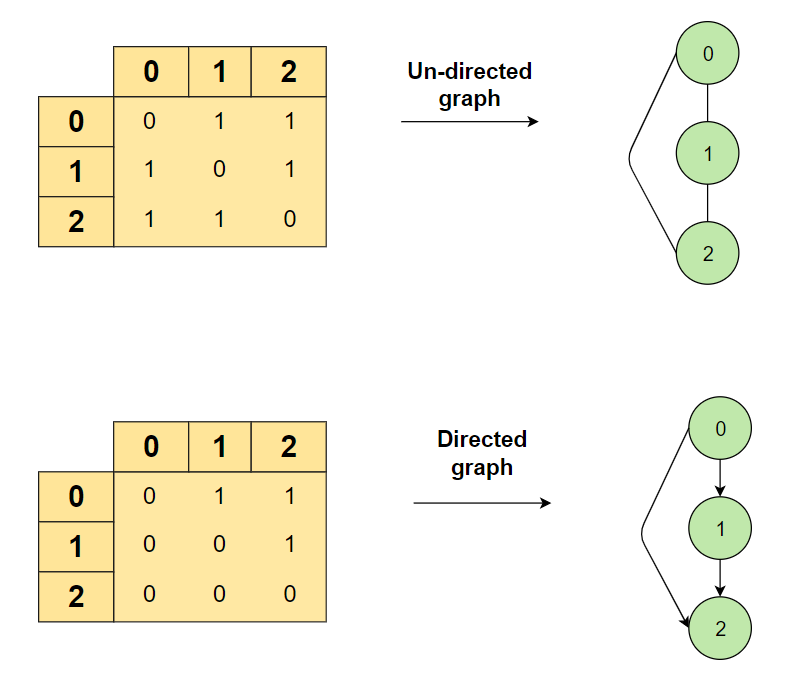
Here are the two most common ways to represent a graph :

1. Adjacency Matrix
2. Adjacency List

**Adjacency Matrix**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Adjacency-Matrix-)

The **adjacency matrix** is a two-dimensional matrix where each cell can contain a **0** or **1**. The row and column headings represent the vertices.

If a cell contains **1**, there exists an edge between the corresponding vertices, e.g., Matrix[0][1]=1*Matrix*[0][1]=1 shows that an edge exists between vertex **0** and **1**.



**Explanation:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Explanation:-)

Initially, the whole matrix is initialized with **0’s**, and as soon as we get an edge, we record the source (*row*) and destination (*col*) value inside our adjacency matrix.

**Undirected Graph:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Undirected-Graph:-)

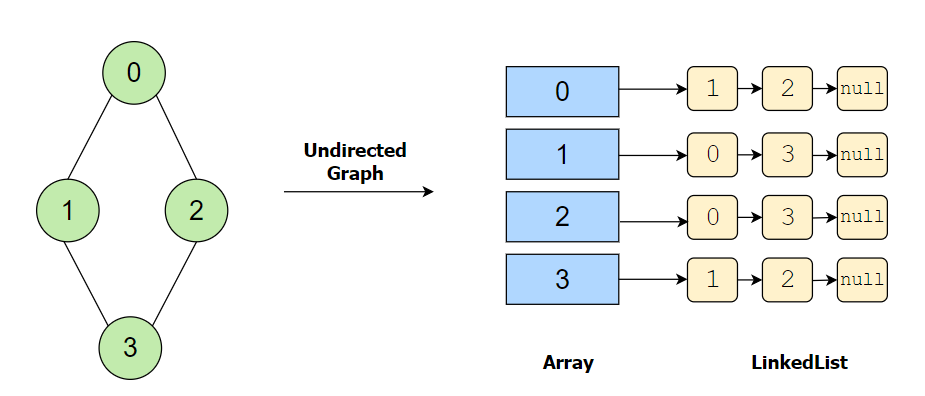
In the figure above, there is an **undirected graph**. If there exists an edge from the source to the destination, then we insert **1** for both cases ([row][col] and [col][row]) as we can traverse both ways.

**Directed Graph:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Directed-Graph:-)

In the case of a directed graph–which has an edge going from Node 0 to Node 1–, there is **1** for rowcol[0][1] in the *adjacency matrix*, and we similarly fill all the remaining parts of the matrix. If there is a node going from [row] to a [col], then we insert **1** for that particular [row][col]; otherwise, we put **0**.

**2. Adjacency List**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#2.-Adjacency-List-)

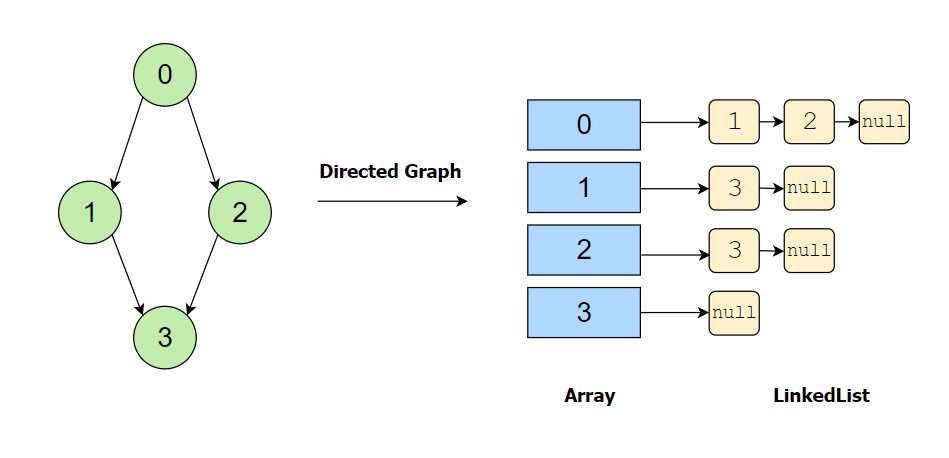
An array of Linked Lists is used to store edges between two vertices. The size of the array is equal to the number of vertices. Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.



**Undirected Graph:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Undirected-Graph:-)

For the undirected graph shown above, each edge connected to a vertex is mentioned in the linked list of the array index (representing that vertex). Since there is an edge from vertex **0** to vertex **1**, so the linked list of vertex **0** (at array index **0**) has a node with value **1**. Thus representing that vertex **1** is having an edge with vertex **0**.

A significant point to note here is that since this is an undirected graph, a linked list of vertex **1** will also contain a node with value **0**, thus representing a vertex from **1** to **0**.



**Directed Graph:**[#](https://www.educative.io/courses/data-structures-coding-interviews-java/RMP2RPWZZZq#Directed-Graph:-)

For the directed graph shown above, there are two outgoing edges from vertex **0**: first to vertex **1** and second to vertex **2**. So the linked list at array index **0** has nodes with value **1** and value **2**.

Since this is a directed graph and edges are unidirectional, hence no edge exists between vertex **1** and vertex **0** and vertex **2** and vertex **0**. Therefore **0** does not exist in a linked list of vertex **1** and vertex **2**.

In the next lesson, we are going to build a graph using the *Adjacency List* that we have covered above.

# Complexities of Graph Operations

Let's discuss the performance of the two graph representation approaches.

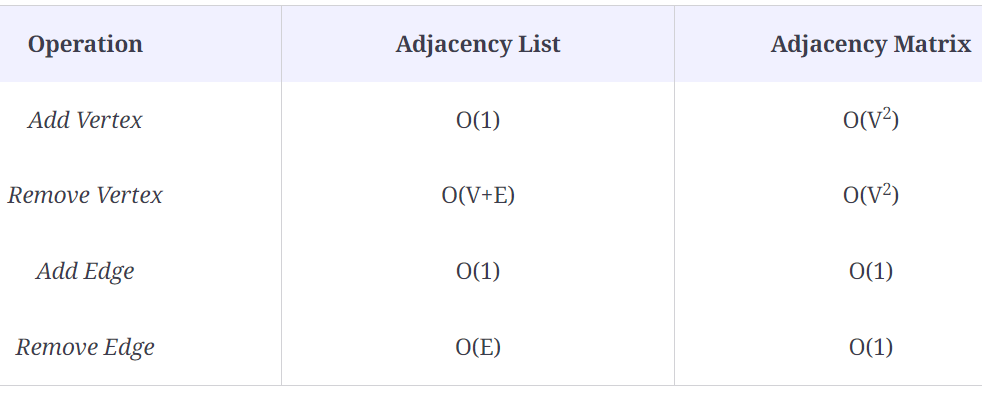
**We'll cover the following**

* + [Time Complexities](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Time-Complexities)
    - [Adjacency List](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Adjacency-List)
    - [Adjacency Matrix](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Adjacency-Matrix)
  + [Comparison](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Comparison)

## Time Complexities[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Time-Complexities)

Below, you can find the time complexities for the 4 basic graph methods.

Note that in this table, **V** means the total number of vertices, and **E** means the total number of edges in the Graph.



### Adjacency List[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Adjacency-List)

* The addition of edge in adjacency lists take constant time, as we only need to insert at the **tail** in the doubly-linked list with a tail pointer of the corresponding vertex.
* The addition of a vertex is not constant in the worst case. The adjacency list is an array of linked lists. So, we would need to allocate a bigger array, copy the contents from the previous array, which takes O(n). But this operation can be considered O(1) on average because we could do it smartly as follows. We start with a small array initially. Then, when there is a need to insert a vertex, instead of allocating one size bigger array, we allocate an array that is twice as big. That way, in the long run, the cost of re-allocation is averaged out over many other operations. So, O(1) is the average or amortized cost of add vertex in the adjacency list.
* Removing an edge takes O(E)*O*(*E*) time because–in the worst case–all the edges could be at a single vertex, and hence, we would have to traverse all **E** edges to reach the last one.
* Removing a vertex takes O(V + E) time because we have to delete all its edges, and then reindex the rest of the list one step back in order to fill the deleted spot.

### Adjacency Matrix[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Adjacency-Matrix)

* Edge operations are performed in constant time, as we only need to manipulate the value in the particular cell.
* Vertex operations are performed in O(V2) since we need to add rows and columns. We will also need to fill all the new cells.

## Comparison[#](https://www.educative.io/courses/data-structures-coding-interviews-java/NEYvz0nqZl6#Comparison)

Both representations are suitable for different situations. If your model frequently manipulates vertices, the adjacency list is a better choice.

If you are dealing primarily with edges, the adjacency matrix is the more efficient approach.

Keep these complexities in mind because they will give you a better idea about the time complexities of the several algorithms we’ll see in this section.

Let’s move towards a different type of ​graph called a Bipartite Graph in the upcoming lesson.

# What is a Bipartite Graph?

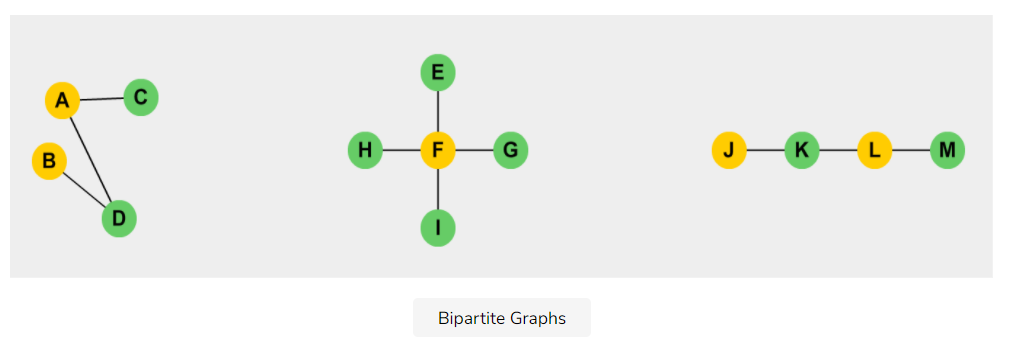
This lesson is about Bipartite Graphs and their key features. We will also take a look at Complete Bipartite graphs along with some examples.

**We'll cover the following**

* + [Introduction](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Introduction)
    - [Explanation](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Explanation)
    - [Can a Cycle Graph be Bi-Partite with Even Vertices?](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Can-a-Cycle-Graph-be-Bi-Partite-with-Even-Vertices?)
  + [Types of bipartite graphs](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Types-of-bipartite-graphs-)

## Introduction[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Introduction)

🔍 ***A Bipartite Graph*** is a special kind of Graph, in which the vertices can be divided into two disjoint sets U and V such that no vertex of U is adjacent to any other vertex in U and no vertex of V is adjacent to any other vertex in V. Vertices in U have edges that connect it to vertices in V.

**Note:** All the cycle graphs, i.e graphs that consist of a single cycle, or some number of vertices (at least 3) connected in a closed chain, are bipartite.

### Explanation[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Explanation)

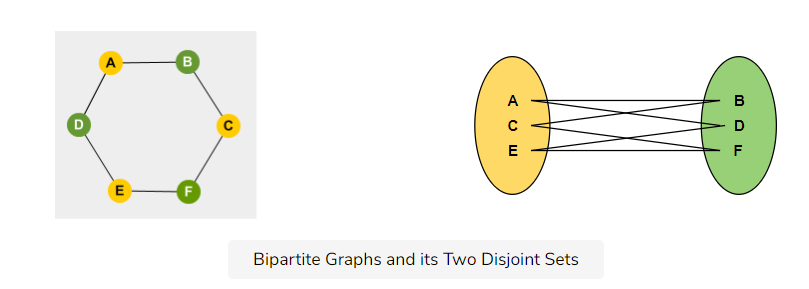
The above graphs are Bipartite.

1. Yellow-colored vertices belong to a disjoint set U.
2. Green colored vertices belong to a disjoint set V.

One can observe in the above graphs that no vertex of U is adjacent to any other vertex in U, and no vertex of V is adjacent to any other vertex in V.

### Can a Cycle Graph be Bi-Partite with Even Vertices?[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Can-a-Cycle-Graph-be-Bi-Partite-with-Even-Vertices?)

The illustration below shows a ***cycle graph*** with an even number of vertices.

But is the above graph Bipartite? **Yes!**

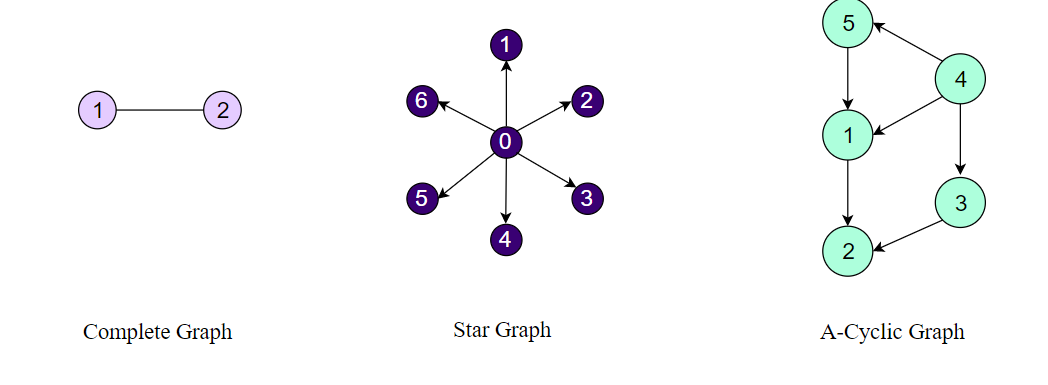
***How?*** You can see that the graph on the left has an even number of nodes, which means that they can be divided into two disjoint sets, shown on the right side, with non-adjacent vertices.

**Note:** A graph cannot be Bipartite if there are an odd number of vertices and has an odd cycle i.e., a cycle between the odd number of vertices.

## Types of bipartite graphs [#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQVGRZo23rn#Types-of-bipartite-graphs-)

Some other types of Bipartite graphs are listed below:

* **Complete Bipartite Graphs:** A complete Bipartite graph connects each vertex from set V1 to each vertex from set V2.
* **Star Graphs:** A star graph is a complete bipartite graph if a single vertex belongs to one set, and all the remaining vertices belong to the other set.
* **Acyclic Graphs:** A graph with no cycles is called an acyclic graph.



In the next lesson, we will discuss different types of Graph traversals and take a look at two famous Graph algorithms.

# Graph Traversal Algorithms

This lesson will cover the key points to traverse a graph, and will also briefly describe two famous graph algorithms— Breadth First Search and Depth First Search.

**We'll cover the following**

* + [Introduction](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#Introduction-)
  + [Types of Graph Traversals](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#Types-of-Graph-Traversals-)
    - [1. Breadth First Search](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#1.-Breadth-First-Search-)
    - [2. Depth First Traversal](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#2.-Depth-First-Traversal-)
  + [Time Complexity](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#Time-Complexity-)

## Introduction [#](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#Introduction-)

There are many applications for graphs, such as the GPS navigation system, shortest path finding, peer to peer networks, crawlers in the search engine, garbage collection (java), and even social networking websites.

Depending upon the problem under, the way we traverse a graph is important, since it can affect the time in which we reach the goal.

## Types of Graph Traversals [#](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#Types-of-Graph-Traversals-)

There are two basic techniques used for graph traversal:

1. Breadth First Search (BFS)
2. Depth First Search (DFS)

In order to understand these algorithms, we will have to view graphs from a slightly different perspective.

Any traversal needs a starting point, but a graph does not have a linear structure like lists or stacks. So how do we give the graph traversal a better sense of direction?

This is where the concept of **levels** is introduced. Take any vertex as the starting point. This is the lowest level in your search. The **next level** consists of all the vertices adjacent to your vertex. Therefore, a level higher would consist of the vertices adjacent to these nodes.

With this is in mind, let’s begin our discussion on the two Graph Traversal algorithms.

### 1. Breadth First Search [#](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#1.-Breadth-First-Search-)

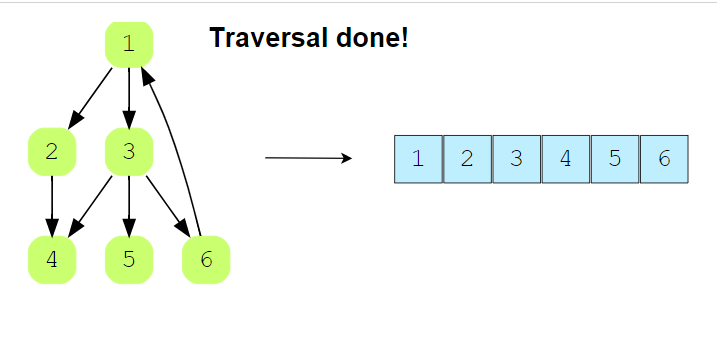
The BFS algorithm earns its name because it grows breadth-wise. All the nodes at a certain level are traversed before moving on to the next level.

The level-wise expansion ensures that you can reach every level of a starting vertex, one level at a time.

Now, let’s look at the BFS algorithm in action.

To build a graph based on BFS, start traversing from any vertex, call it *currentVertex*. If the adjacent vertices are yet already not visited, then print their values. Then, move on to children of the currentVertex.

Let’s look at BFS traversal in action!

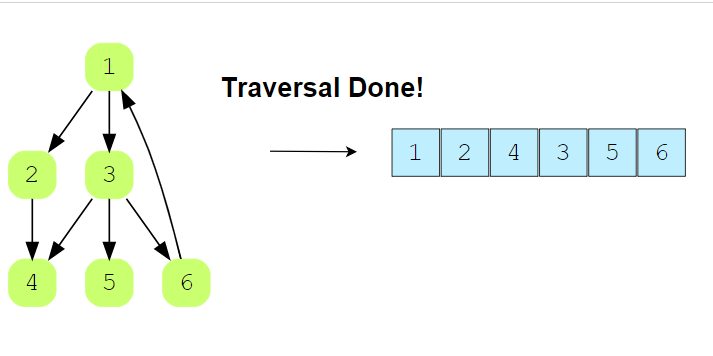
**8** of 8

### 2. Depth First Traversal [#](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#2.-Depth-First-Traversal-)

The DFS algorithm is the opposite of BFS in the sense that it grows depth-wise.

Starting from any node, we keep moving to an adjacent node until we reach the farthest level. Then we move back to the starting point and pick another adjacent node. Once again, we probe till the farthest level and move back. This process continues until all nodes are visited.

Let’s look at DFS traversal in action:

**10** of 10

## Time Complexity [#](https://www.educative.io/courses/data-structures-coding-interviews-java/gx8NKY3J326#Time-Complexity-)

The Time complexity of both **BFS** and **DFS**–where **V** is the number of vertices and **E** is the number of Edges–is given below.



In the upcoming lesson, we will implement Breadth-First Search and Depth-First Search algorithms based on our pre-implemented class of Graph.

# Challenge 1: Implement Breadth First Search

In this lesson, you have to implement the Breadth First Search algorithm discussed in the previous lesson. A solution is placed in the "solution" section to help you, but we would suggest trying to solve it on your own first.

**We'll cover the following**

* + [Problem Statement](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Problem-Statement)
    - [Input](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Input)
    - [Output](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Output)
    - [Sample Input](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Sample-Input)
    - [Sample Output](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Sample-Output)
  + [Coding Exercise](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Coding-Exercise-)

## Problem Statement[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Problem-Statement)

In this exercise, you have to implement Breadth First Search traversal in Java. It is a level-by-level searching algorithm for the graph, so we are going to use our already-implemented class of Graph for this task (since we have already covered the implementation of Graph).

**Note:** Your solution should work for both connected and unconnected graphs. For an unconnected graph, the order of output should depend upon indices in ascending order.

To solve this problem, all the previously implemented data structures will be available to us.

### Input[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Input)

A graph in the form of an adjacency list

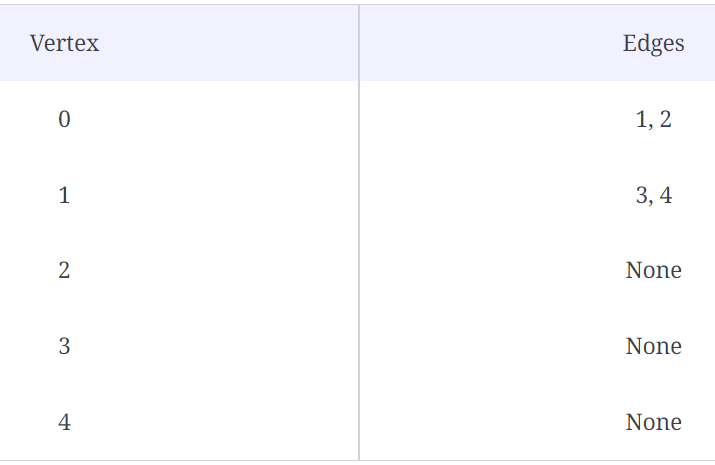
### Output[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Output)

A string containing the vertices of the graph listed in the correct order of traversal.

### Sample Input[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Sample-Input)

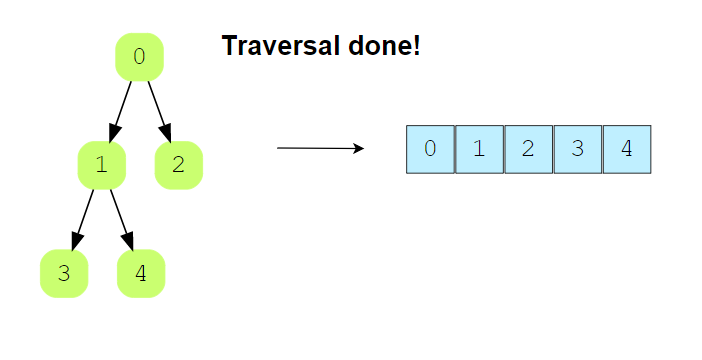
**Graph:**

In the **Input** column of Test cases, this graph will be represented as: |V|=5, E:[(0,1)(0,2)(1,3)(1,4)], where, |V| represents the number of vertices while E represents the edges.



### Sample Output[#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Sample-Output)

"01234"



**1** of 6

## Coding Exercise [#](https://www.educative.io/courses/data-structures-coding-interviews-java/YQ7jOBvY1LM#Coding-Exercise-)

First, take a close look and design a step-by-step algorithm before jumping to the implementation. This problem is designed for your practice, so initially try to solve it on your own. If you get stuck, you can always refer to the solution provided in the solution section. Good Luck!

# Solution Review: Implement Breadth First Search

## Solution: Using Queue

class CheckBFS {

    public static String bfs(Graph g){

        String result = "";

        //Checking if the graph has no vertices

        if (g.vertices < 1){

            return result;

        }

        //Boolean Array to hold the history of visited nodes (by default-false)

        boolean[] visited = new boolean[g.vertices];

        for(int i=0;i<g.vertices;i++)

        {

            //Checking whether the node is visited or not

            if(!visited[i])

            {

                result = result + bfsVisit(g, i, visited);

            }

        }

        return result;

    }

    public static String bfsVisit(Graph g, int source, boolean[] visited) {

        String result = "";

        //Create Queue for Breadth First Traversal and enqueue source in it

        Queue<Integer> queue = new Queue<>(g.vertices);

        queue.enqueue(source);

        visited[source] = true;

        //Traverse while queue is not empty

        while (!queue.isEmpty()) {

            //Dequeue a vertex/node from queue and add it to result

            int current\_node = queue.dequeue();

            result += String.valueOf(current\_node);

            //Get adjacent vertices to the current\_node from the array,

            //and if they are not already visited then enqueue them in the Queue

            DoublyLinkedList<Integer>.Node temp = null;

            if(g.adjacencyList[current\_node] != null)

                temp = g.adjacencyList[current\_node].headNode;

            while (temp != null) {

                if (!visited[temp.data]) {

                    queue.enqueue(temp.data);

                    visited[temp.data] = true; //Visit the current Node

                }

                temp = temp.nextNode;

            }

        }//end of while

        return result;

    }

    public static void main(String args[]) {

        Graph g = new Graph(5);

        g.addEdge(0,1);

        g.addEdge(0,2);

        g.addEdge(1,3);

        g.addEdge(1,4);

        System.out.println("Graph1:");

        g.printGraph();

        System.out.println("BFS traversal of Graph1 : " + bfs(g));

        System.out.println();

        Graph g2 = new Graph(5);

        g2.addEdge(0,1);

        g2.addEdge(0,4);

        g2.addEdge(1,2);

        g2.addEdge(3,4);

        System.out.println("Graph2:");

        g2.printGraph();

        System.out.println("BFS traversal of Graph2 : " + bfs(g2));

  }

}

### Explanation[#](https://www.educative.io/courses/data-structures-coding-interviews-java/mEoQP6xXQjp#Explanation)

The bfs() function is a wrapper for the bfsVisit() function which actually performs the traversal on one source vertex at a time and outputs all vertices reachable from the source. The reason for using the wrapper function is to make sure we traverse all vertices even when they are not reachable from any other vertex in the graph.

In the bfs() function, we insert the vertices into the queue, starting from the source vertex. To keep track of the nodes that have been traversed, every node inserted into the queue is marked visited in the visited array.

The result string is our output variable. The value of a node is appended to result when it is dequeued from the queue. For each node that is dequeued, its adjacent nodes are added to the queue if they have not been visited.

The **First In First Out** (FIFO) structure of the queue ensures that the graph is traversed one level at a time

### Time Complexity[#](https://www.educative.io/courses/data-structures-coding-interviews-java/mEoQP6xXQjp#Time-Complexity)

Since this algorithm traverses the whole graph once, its time complexity is O(V + E).