# INTRODUCTION TO GRAPHS AND TREES

## Chapter 6 – Part1

**Prepared by: Enas Abu Samra**

# KEY POINTS OF CHAPTER 6

- Graphs and Trees.
- Graphs Terminology.
- Categories of Graphs.
- Types of Graphs.
- Trees Terminology.
- Key Differences Between Graphs and Trees.
- Representation of Graphs and Trees in Computers.

# TREES AND GRAPHS

- Trees and graphs are both abstract data structures. They are a **non-linear** collection of objects, which means that there is **no sequence between their elements** as it exists in a linear data structures like stacks and queues.
- Trees and graphs are data structures used to resolve various complex problems.

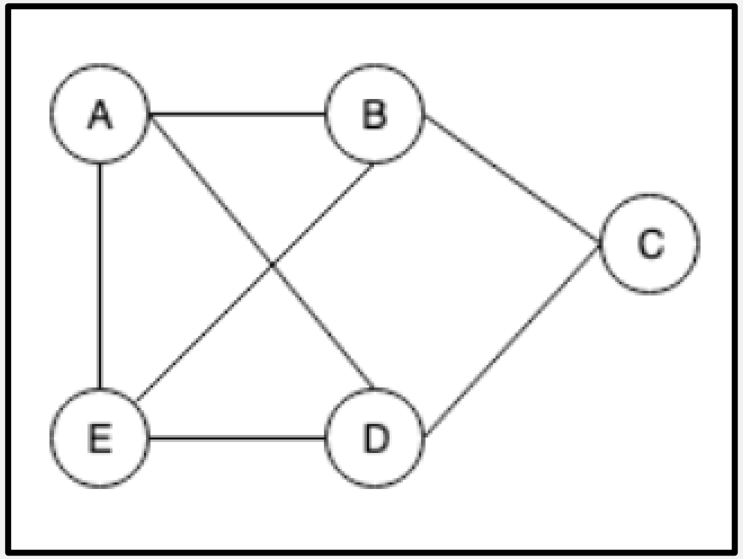
# Graphs

# GRAPHS

- A graph can also be defined as a collection of entities called **vertices** (nodes/points), connected to each other through a set of edges. The set of **edges** (lines/arcs) describes the relationships between the vertices.
- A graph G is defined as follows: G=(V, E)

  V(G): a finite, **nonempty** set of vertices.

  E(G): a set of edges.




- V={a, b, c, d, e}
- E={(ab),(ad),(ae),(bc),(be),(cd),(ed)}

# GRAPHS TERMINOLOGY

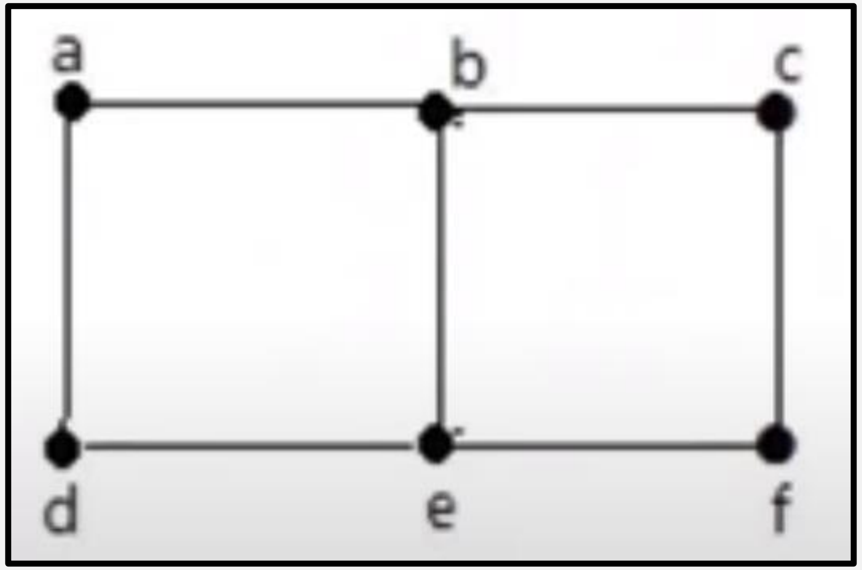
- **Vertex**: each node of the graph.
- **Edge**: a path or a line between two vertices.
- **Path**: a sequence of edges between the two vertices.
- **Cycle**: a path where the first and last vertices are the same.
- **Adjacency**: two nodes or vertices are adjacent if they are connected to each other through an edge.

# ADJACENCY

- In a graph, **two vertices are said to be adjacent**, if there is an edge between the two vertices. Here, the adjacency of vertices is maintained by the **single edge** that is connecting those two vertices.
- In a graph, **two edges are said to be adjacent**, if there is a common vertex between the two edges. Here, the adjacency of edges is maintained by the **single vertex** that is connecting two edges.

# ADJACENCY

- In the following graph:
- ✓ 'a' and 'd' are the **adjacent vertices**, as there is a common edge 'ad' between them.
- ✓ 'a' and 'b' are the **adjacent vertices**, as there is a common edge 'ab' between them.
- ✓ ' ab' and 'be' are the **adjacent edges**, as there is a common vertex 'b' between them.

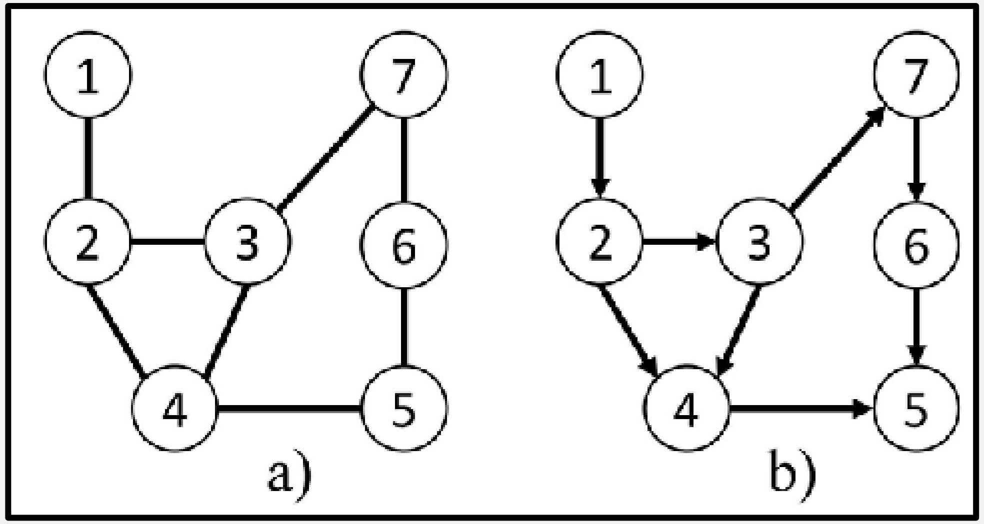


# CATEGORIES OF GRAPHS

- **Graphs can be:**
  - ✓ Directed  vs  Undirected
  - ✓ Weighted vs Unweighted
  - ✓ Connected vs Disconnected
  - ✓ Cyclic vs Acyclic.
  - ✓ Sparse vs Dense.

# UNDIRECTED AND DIRECTED GRAPHS

- When the edges in a graph have **no direction**, the graph is called **undirected**.
- When the edges in a graph have a **direction**, the graph is called **directed** (or digraph).

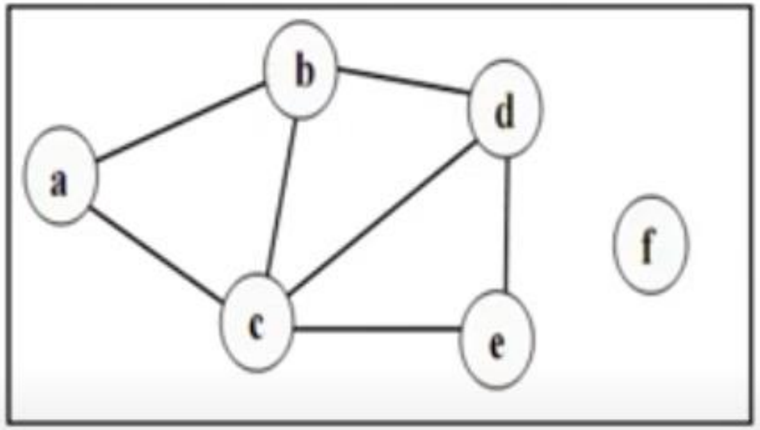


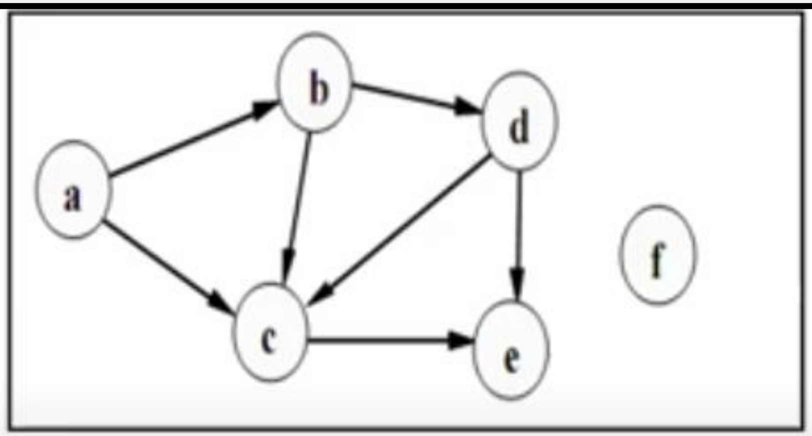
# DEGREE IN DIRECT AND UNDIRECT GRAPH

- **Degree in undirected graphs:**
- **Degree(V)** = # of adjacent (incident) edges to vertex v in G.
- **Σ degrees = 2 |E|**

- **Degree in directed graphs:**
- **In-Deg(V)** = # of incoming edges.
- **Out-Deg(V)** = # of outgoing edges.
- **Σ In-degree = Σ Out-degree = |E|**

# DEGREE IN DIRECT AND UNDIRECT GRAPH



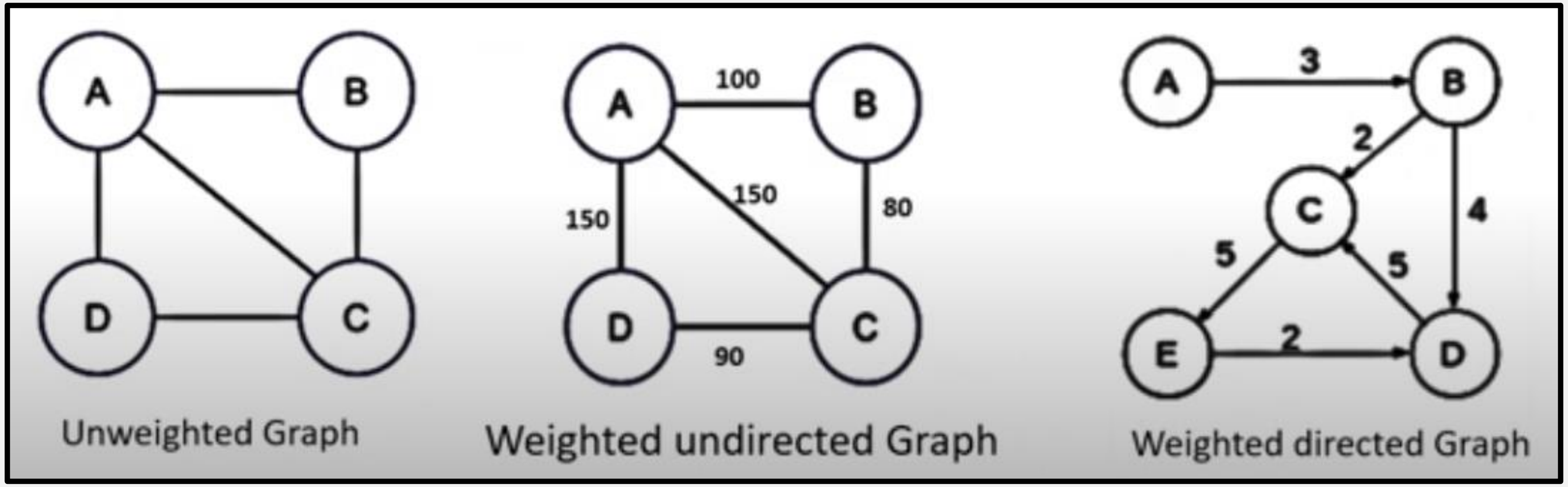




| vertex | degree |
|---|---|
| a | 2 |
| b | 3 |
| c | 4 |
| d | 3 |
| e | 2 |

| vertex | In-degree | Out-degree |
|---|---|---|
| a | 0 | 2 |
| b | 1 | 2 |
| c | 3 | 1 |
| d | 1 | 2 |
| e | 2 | 0 |

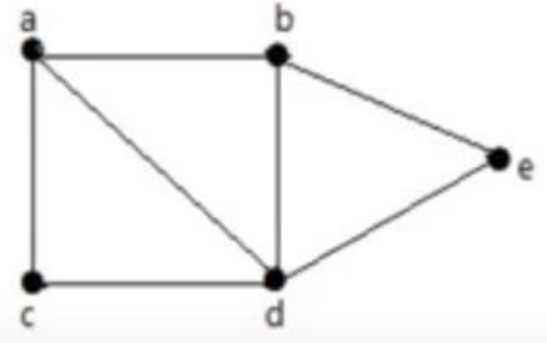
# WEIGHTED AND UNWEIGHTED GRAPH

- If edges in the graph have **weights**, then the graph is said to be a **weighted graph**, if the edges **do not have weights**, the graph is said to be **unweighted**.
- **A weight** is a numerical value attached to each individual edge.
- Weights may represent distance, cost, time etc.

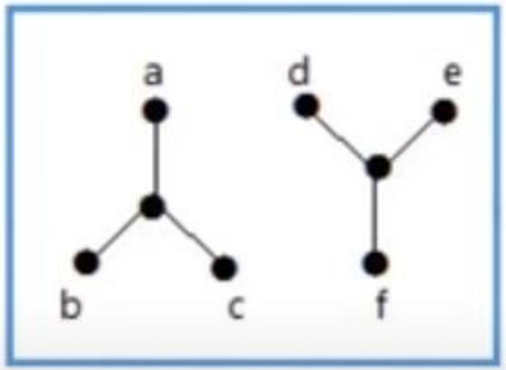


# CONNECTIVITY:
# CONNECTED AND DISCONNECTED GRAPH

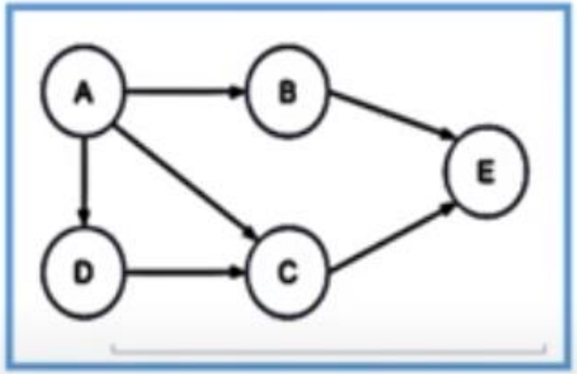
- A graph is said to be **connected** if there is a **path between every pair of vertices**. From every vertex to any other vertex, there should be some path to traverse.
- It is possible to travel from one vertex to any other vertex.




It can traverse from vertex 'a' to vertex 'e' using the path 'a-b-e'.




Traversing from vertex 'a' to vertex 'f' is not possible - No Path




Not possible to traverse to 'a'

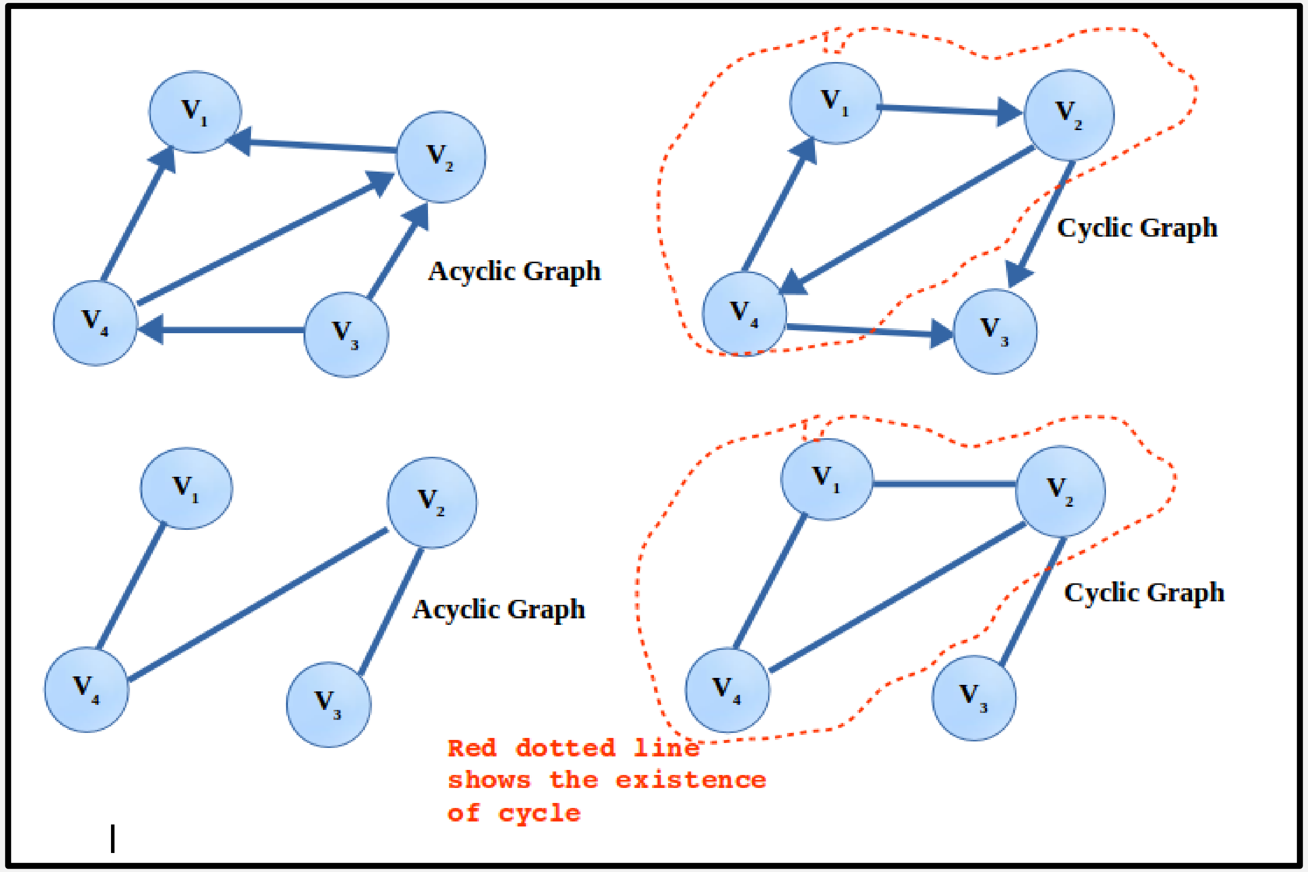
# CYCLIC AND ACYCLIC GRAPH

- A graph is said to have a **cycle** if you start from a node/vertex and after traversing some nodes, you come to the **same node**, then you can say that the graph is having a cycle.

- If there is a **cycle** in a graph, then that graph is called **Cyclic Graph**. If there is **no cycle** present in the graph, then that graph is called an **Acyclic Graph**.

- 

- **For a Cyclic Graph, at least one cycle is necessary.**



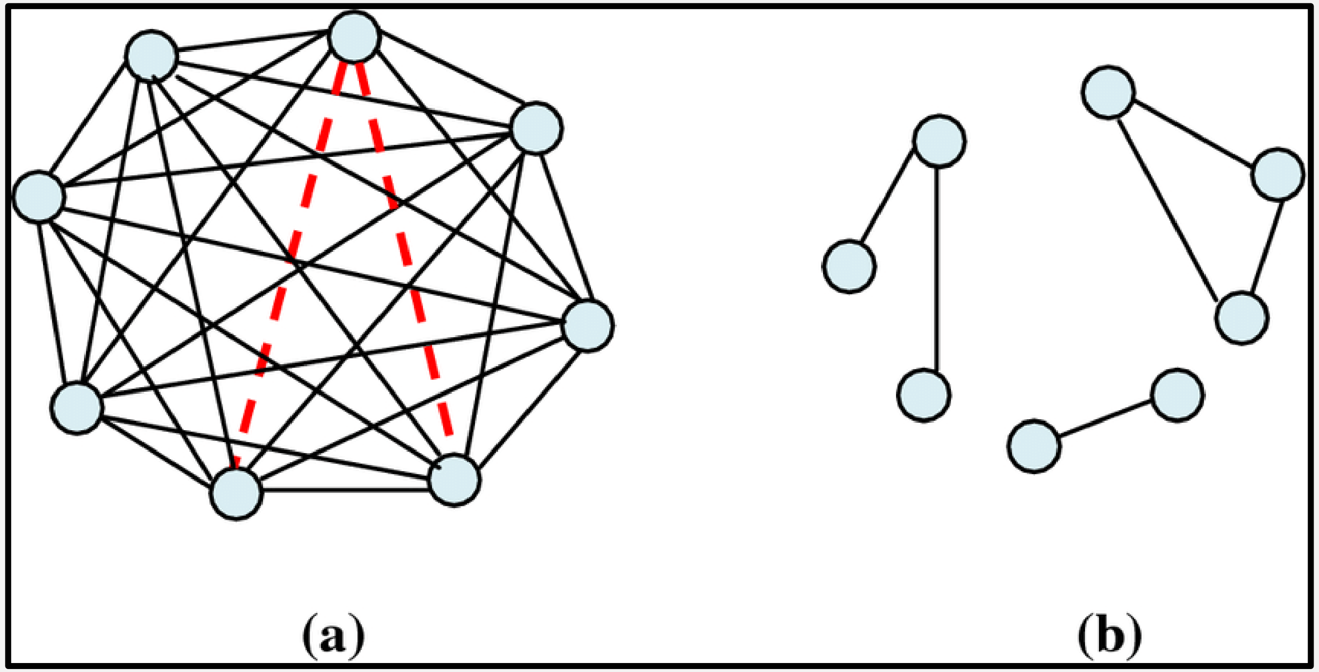
# SPARSE AND DENSE GRAPH

- **Sparse Graph:** A graph in which the **number of edges is much less than the possible number of vertices.**
- **Sparse Graph:** A sparse graph is a graph

  G = (V, E) in which |E| = O(|V|).

- **Dense Graph:** A graph in which the **number of edges is close to the possible number of vertices.**
- **Dense Graph:** A dense graph is a graph

  G = (V, E) in which $|E| = O(|V|^2)$.

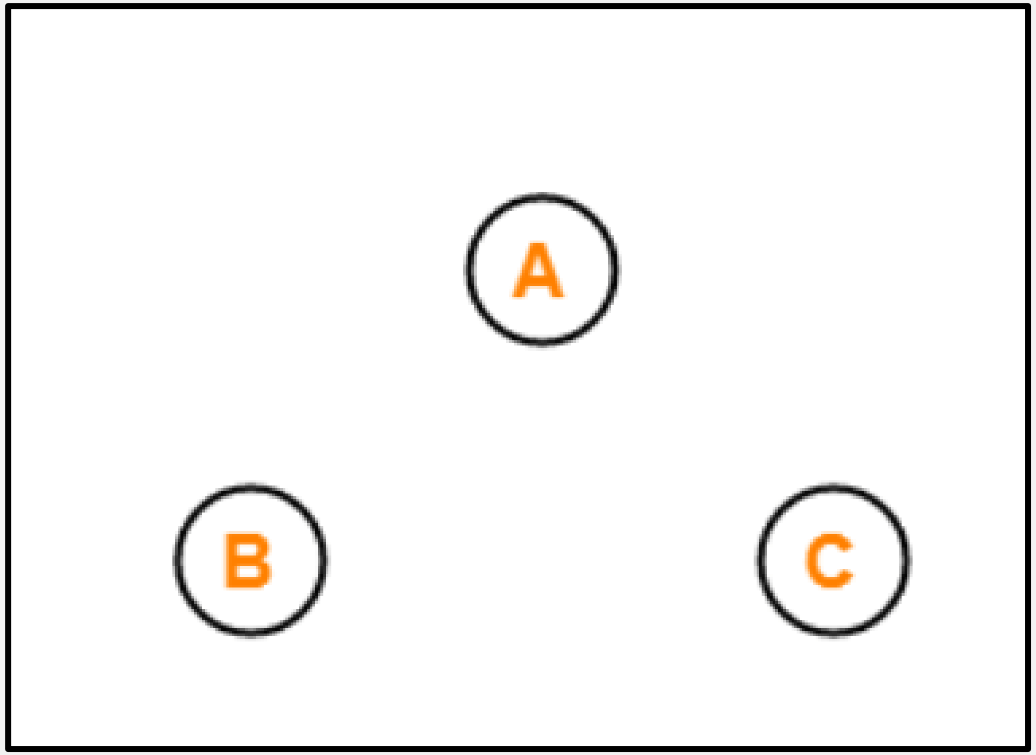


# TYPES OF GRAPHS

- ✓ Null Graph.
- ✓ Multi Graph.
- ✓ Regular Graph.
- ✓ Complete Graph.

# NULL GRAPH

- A null graph is a graph containing **no edges**.

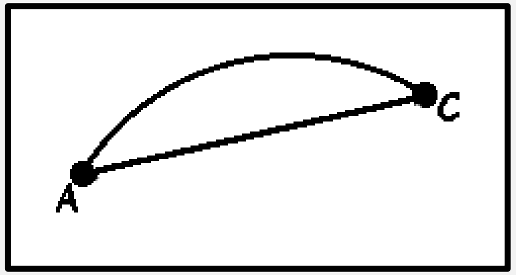


# PARALLEL EDGES AND MULTI GRAPH

- **Parallel Edges**

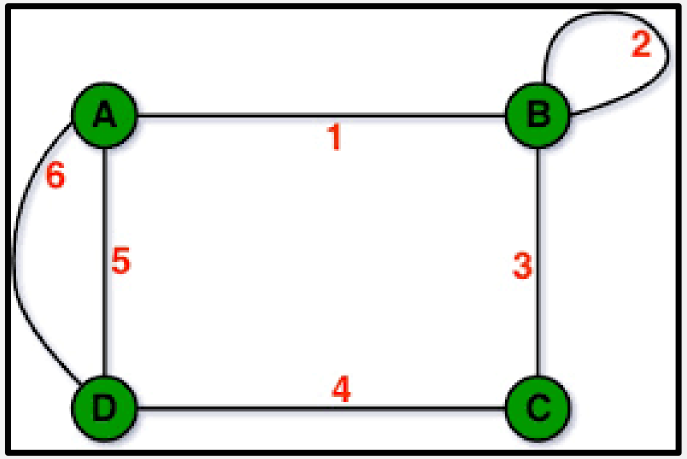
✓ In a graph, if a pair of vertices is connected by **more than one edge**, then those edges are called **parallel edges**.

✓ In the following example: 'a' and 'c' are the two vertices which are connected by two edges 'ac' and 'ca' between them.



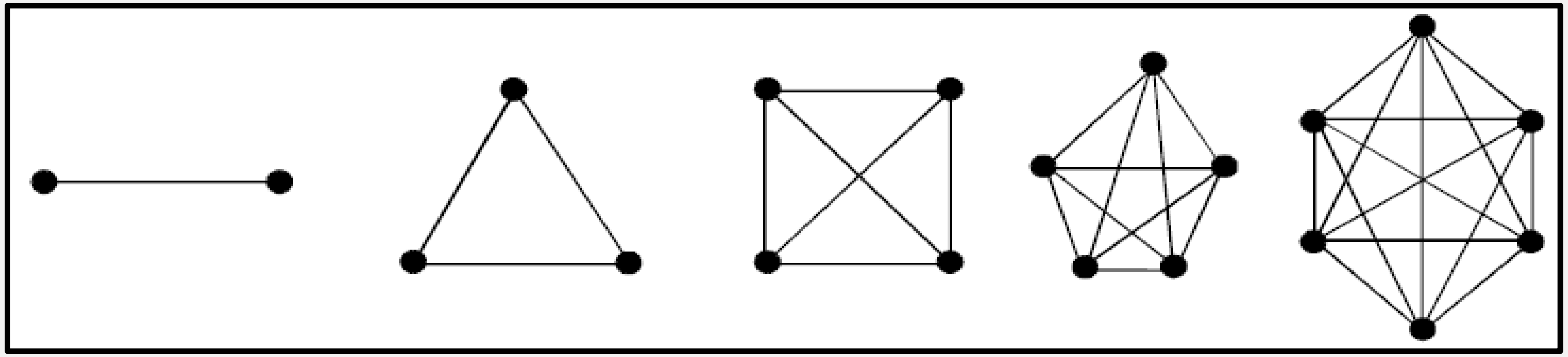

- **Multi Graph**

✓ A graph having **parallel edges** is known as a **Multigraph**.



# REGULAR GRAPH

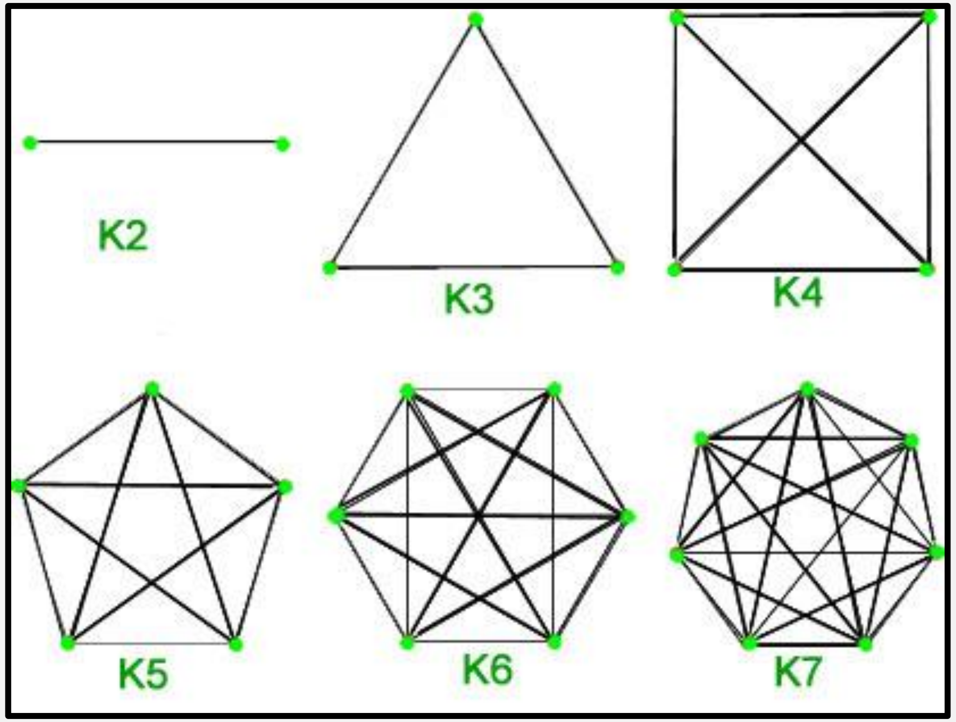
- **Regular graph** is a graph where each vertex has **the same number of neighbors (every vertex has the same degree).**



1-regular 2-regular 3-regular 4-regular 5-regular

# COMPLETE GRAPH

- A graph G is Complete Graph ($G_N$) if every node u in G is adjacent to **every** other node v in G.
- A complete graph is already **connected**.
- **# of edges** = n(n-1)/2



# Trees

# TREES

- A tree is a **nonlinear** data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.
- A tree can be **empty** with **no nodes**, or a **tree is a structure consisting of one node called the root and zero or one or more subtrees.**
- They **don’t have any cyclic** relations and there is **only one path to a particular node.**
- A tree must be **connected** which means **there must be a path from the root to all other nodes.**

# TREES TERMINOLOGY

- **Root**: the top (initial) node of the tree, where all the operations start.
- **Node**: each item in the tree, usually a key-value.
- **Edge**: a tree has n-1 edges (where n is the number of nodes) representing the connection between two nodes.
- **Parent**: a node which is a predecessor of any node.
- **Child**: a node which is descendant of any node.
- **Siblings**: a group of nodes which have the same parent.
- **Leaf (terminal) node**: a node without children.

# TREES TERMINOLOGY

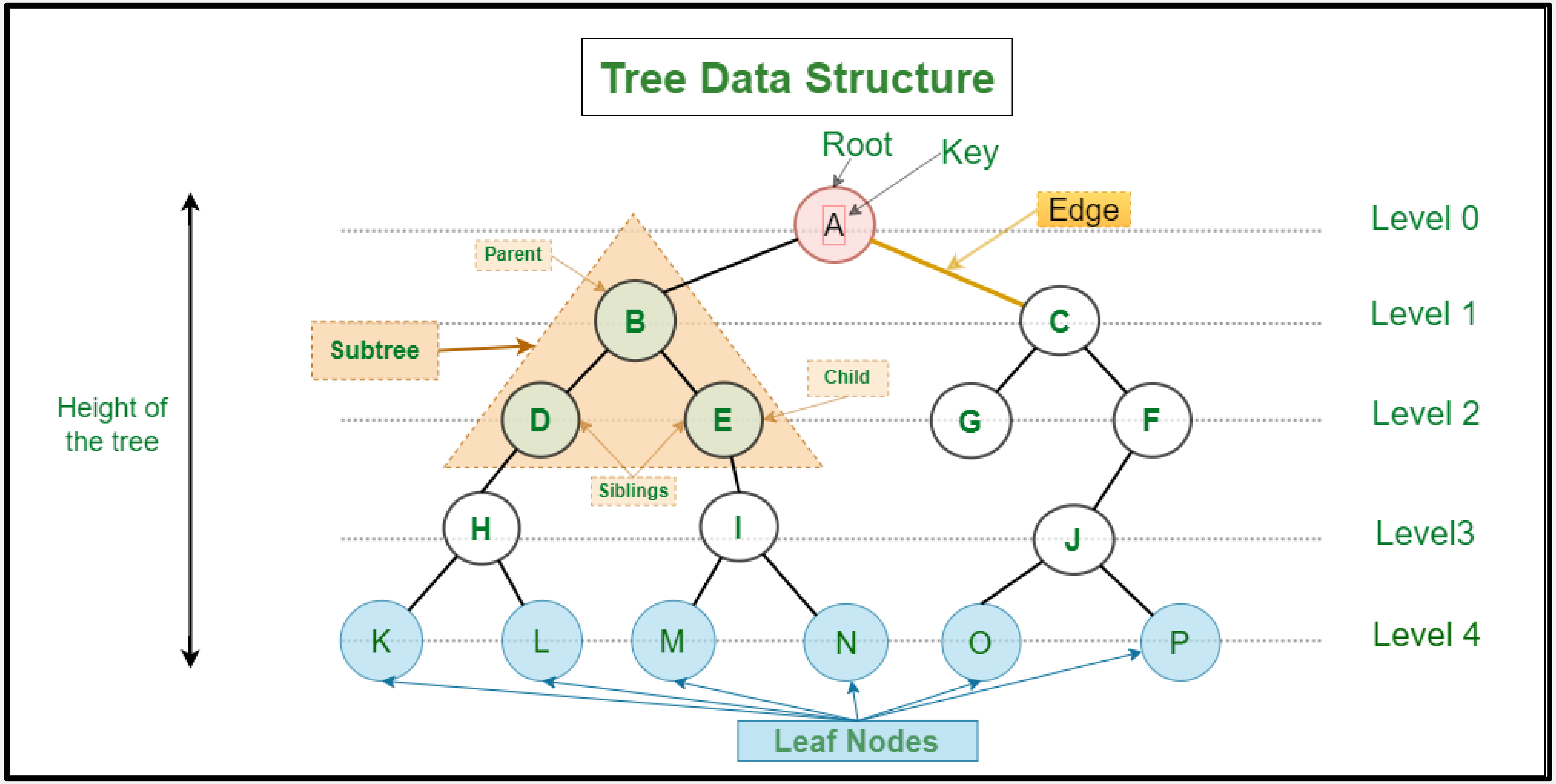
- **Level:** is the number of edges on the path from the root node to n.

  **The level of the root node is zero.**

  Also, it defined as 1 + the number of edges between the node and the root.
- **Height:** the number of edges from its root to the furthest leaf.
- **Sub-tree:** a portion of a tree data structure that can be viewed as a complete tree in itself

- There are different types of trees that you can work with, like Binary Tree, Binary Search Tree, Red-Black tree, AVL tree, Heap, etc. The deciding factor of which tree type to use is **performance**. Since trees are data structures, **performance is measured in terms of inserting and retrieving data.**

# TREES TERMINOLOGY



# Key Differences
# Between Graphs and Trees

# KEY DIFFERENCES BETWEEN GRAPHS AND TREES

- Trees and graphs are mainly differentiated by the fact that a tree structure must be **connected** and **can never have loops** while in the graph there are **no such restrictions.**

- **In trees**, all nodes must be reachable from the root and there must be **exactly one possible path from the root to a node. In graphs**, there are **no rules** dictating the connections among the nodes.

- **Main use of graphs is coloring and job scheduling**, on the other hand **main use of trees is for sorting and traversing.**

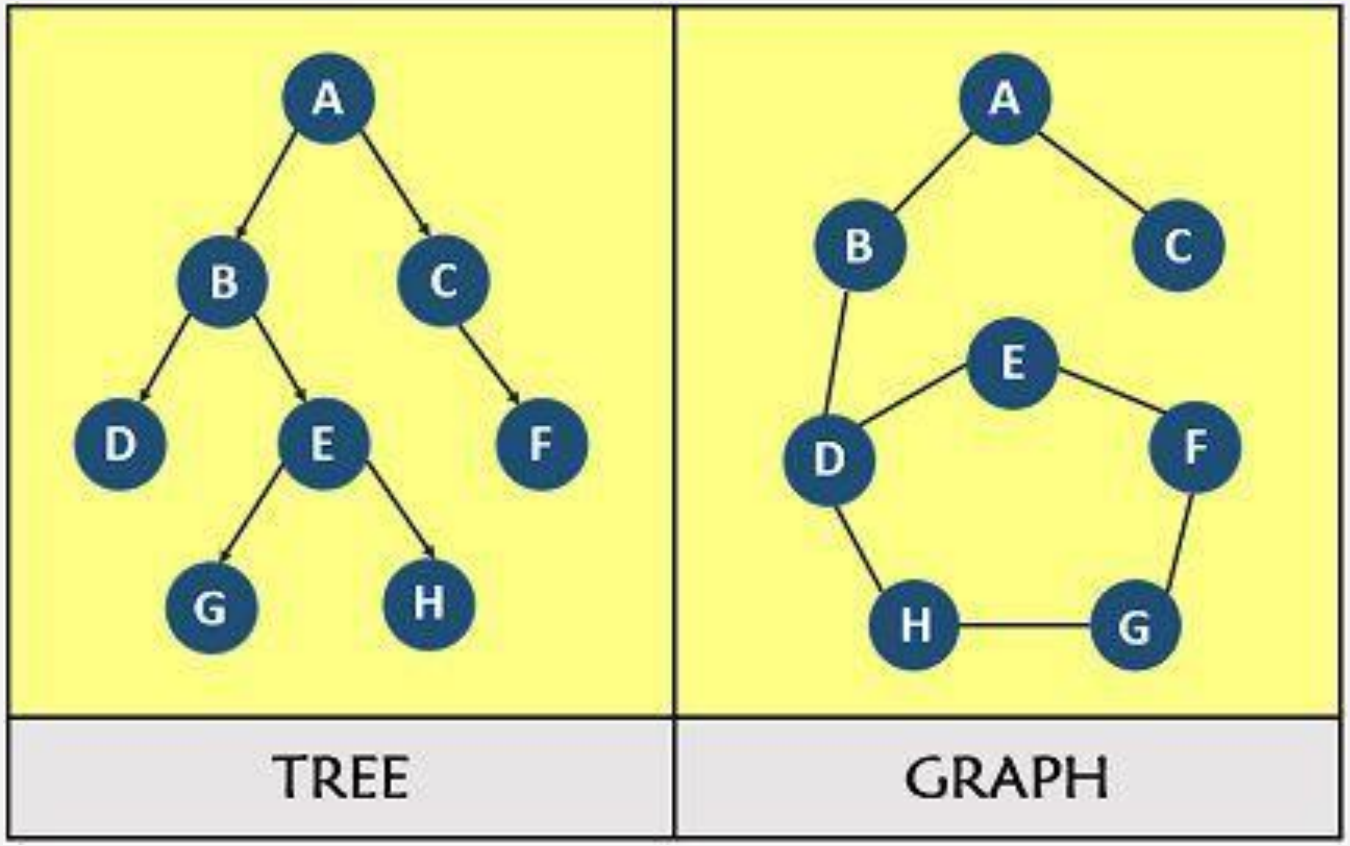
# KEY DIFFERENCES BETWEEN GRAPHS AND TREES

- **In graphs**, **the number of edges doesn't depend on the number of vertices**. On the contrary, if a **tree has "n" vertices (nodes) then it must have exactly "n-1" edges.**

- There **must** be a **root node in a tree** while there **is no such concept in a graph.**

- Basically speaking, **a tree is just a restricted form of a graph (connected acyclic graph).** Also known as a minimally connected graph. That makes **graphs more complex structures compared to the trees due to the loops and circuits**, which they may have.

# EXAMPLE OF A TREE AND A GRAPH



# Representation of Graphs and Trees in Computers

# Representation of Graphs in Computers

# GRAPH REPRESENTATION

- Different data structures for the representation of graphs are used in practice:
  1. Adjacency Matrix.
  2. Adjacency List.
  3. Incidence Matrix.

# GRAPH REPRESENTATION: ADJACENCY MATRIX

- **A two-dimensional matrix**, in which the **rows represent source vertices and columns represent destination vertices.** Only the cost for one edge can be stored between each pair of vertices.
- **Size:** $V \times V$, where is the number of vertices in the Graph.
- adjMatrix[i][j] = 1 when there is an edge b/w Vertex i and Vertex j, else 0.

# GRAPH REPRESENTATION: ADJACENCY MATRIX

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

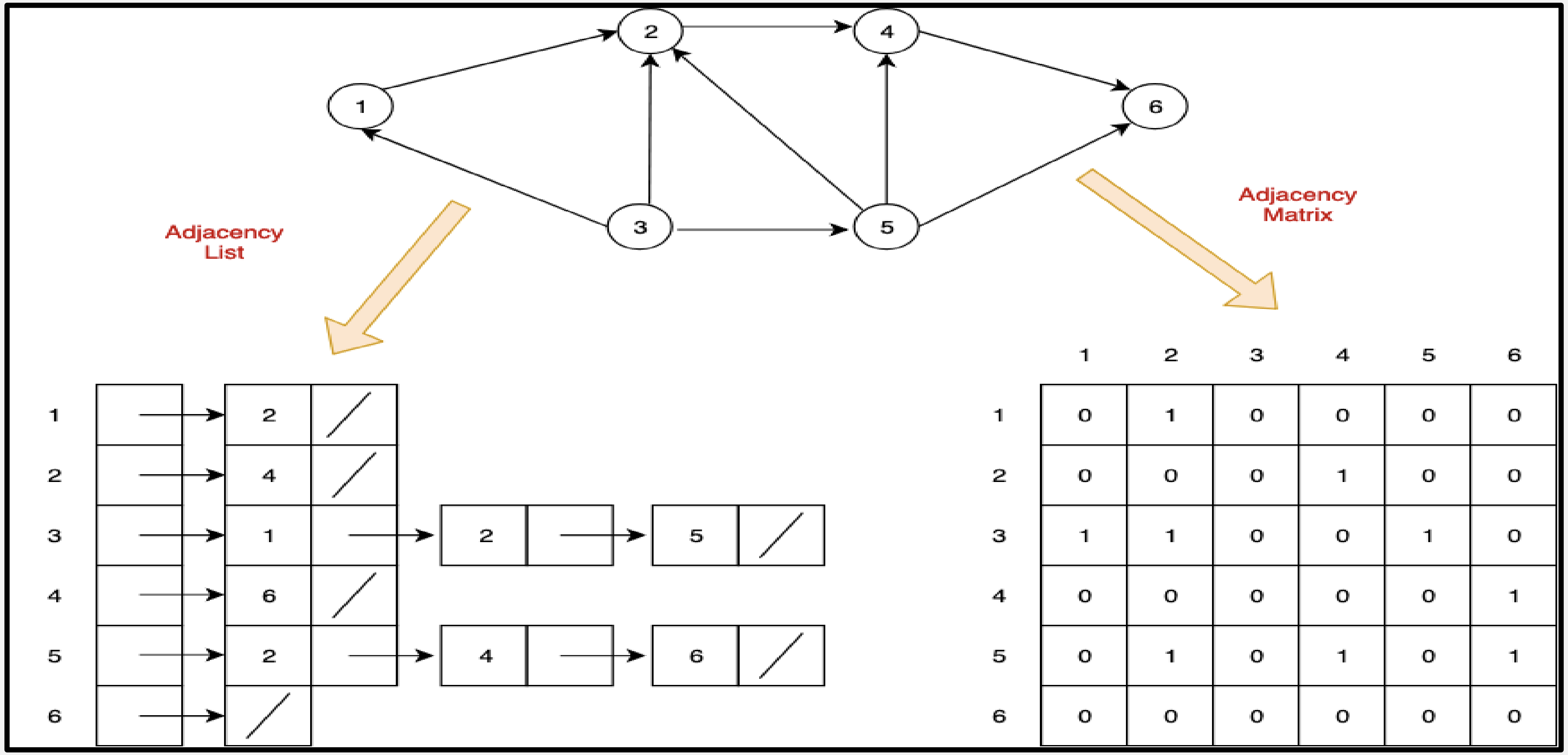
# GRAPH REPRESENTATION: ADJACENCY LIST

- Adjacency List is **the Array[] of Linked List**, where array size is same as number of vertices in the graph. Every Vertex has a Linked List.
- Each node in this linked list represents the **reference** to the other vertices which share an edge with the current vertex. **The weights can also be stored in the linked list node.**

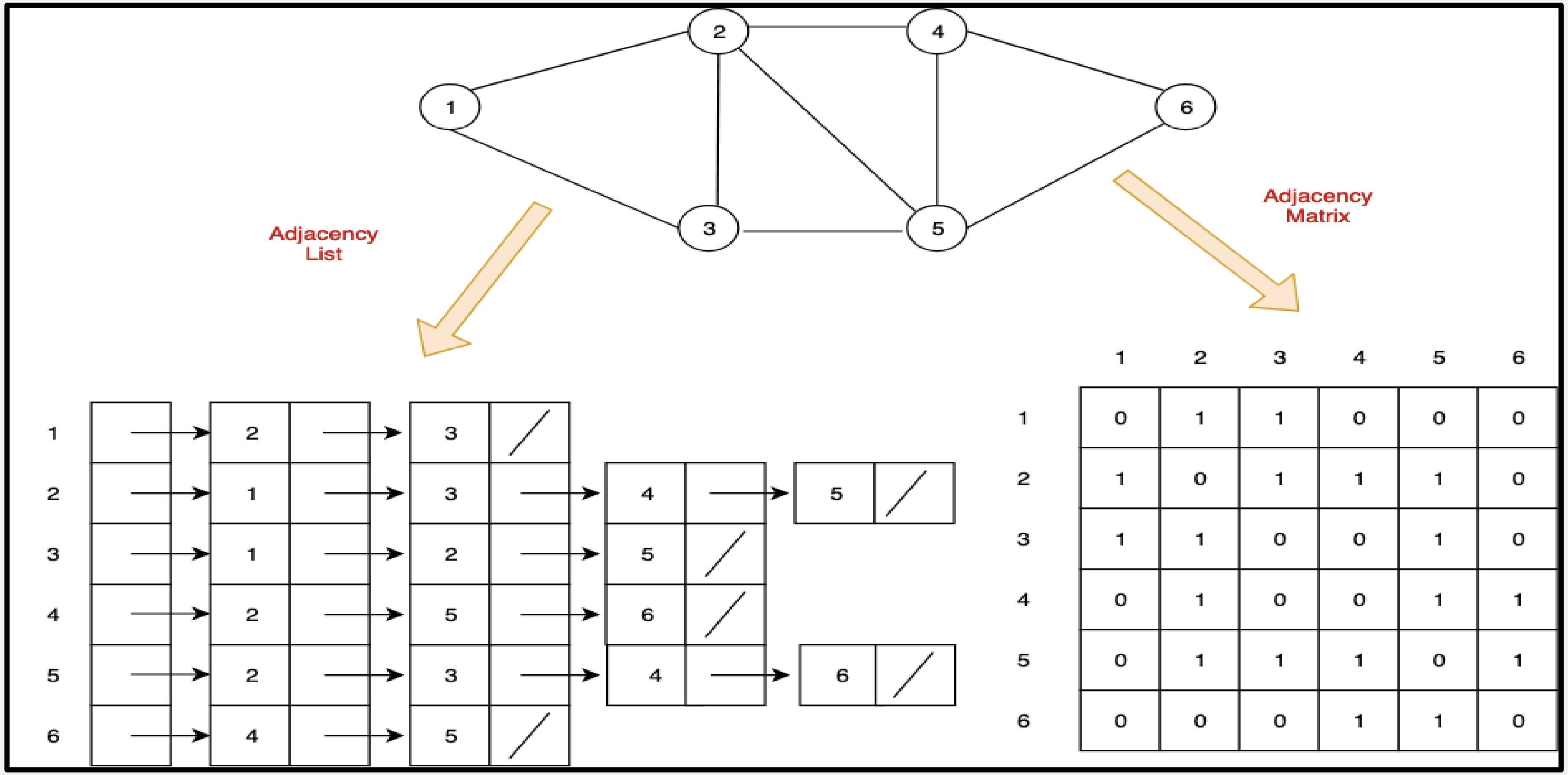
# GRAPH REPRESENTATION: ADJACENCY LIST

- Save space $O(|V|+|E|)$.
- Adding a vertex is easier.
- Support Sequential Search Only.

# GRAPH REPRESENTATION



# GRAPH REPRESENTATION




| | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 2 | 1 | 0 | 1 | 1 | 1 | 0 |
| 3 | 1 | 1 | 0 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 0 | 1 | 1 |
| 5 | 0 | 1 | 1 | 1 | 0 | 1 |
| 6 | 0 | 0 | 0 | 1 | 1 | 0 |

# Representation of Trees in Computers

# TREE REPRESENTATION

- Different data structures for the representation of trees are used in practice:

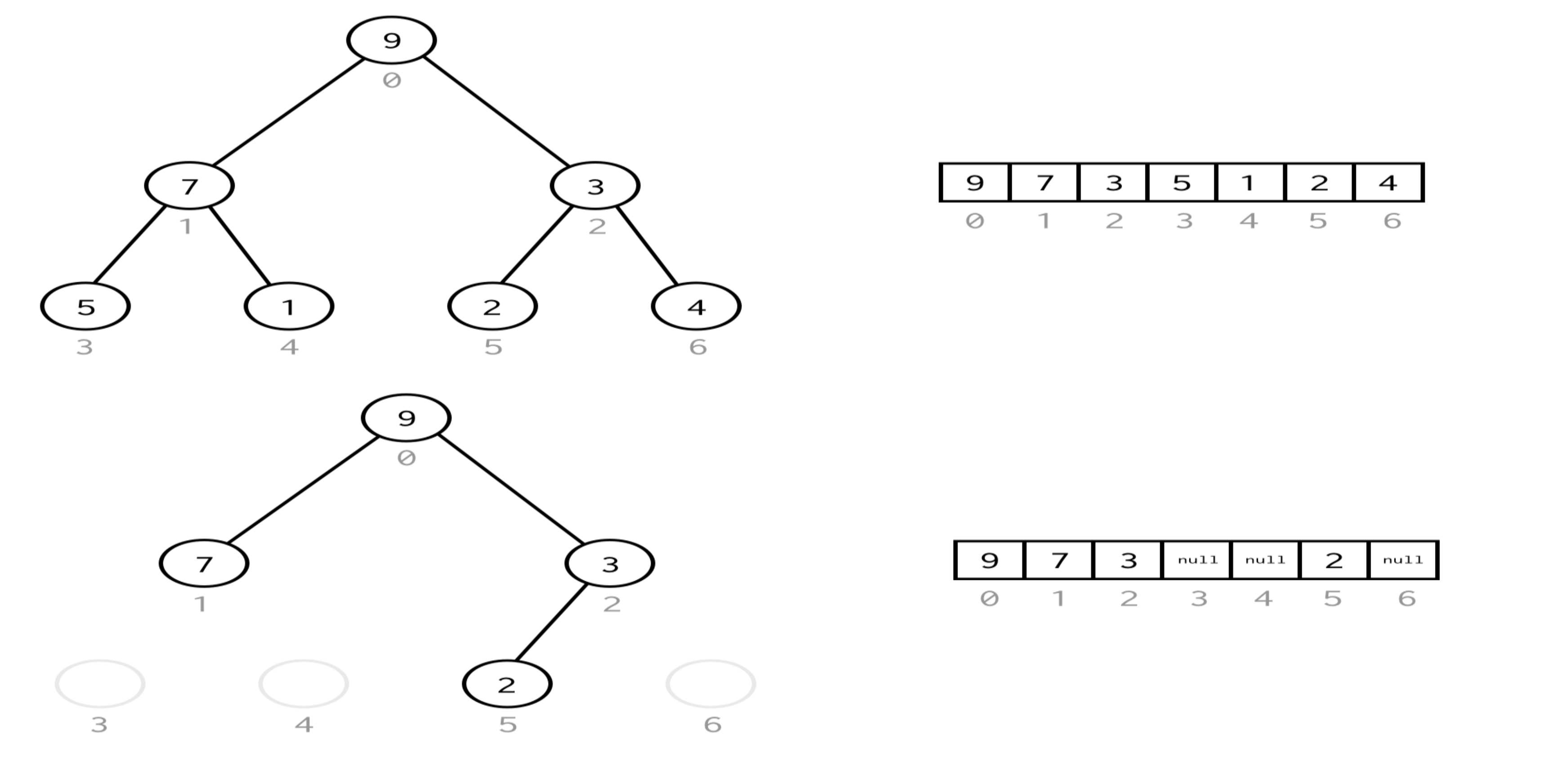
1. Arrays
2. Linked List
   - ✓ Single linked list
   - ✓ Double linked list

# TREE REPRESENTATION: ARRAY

- To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

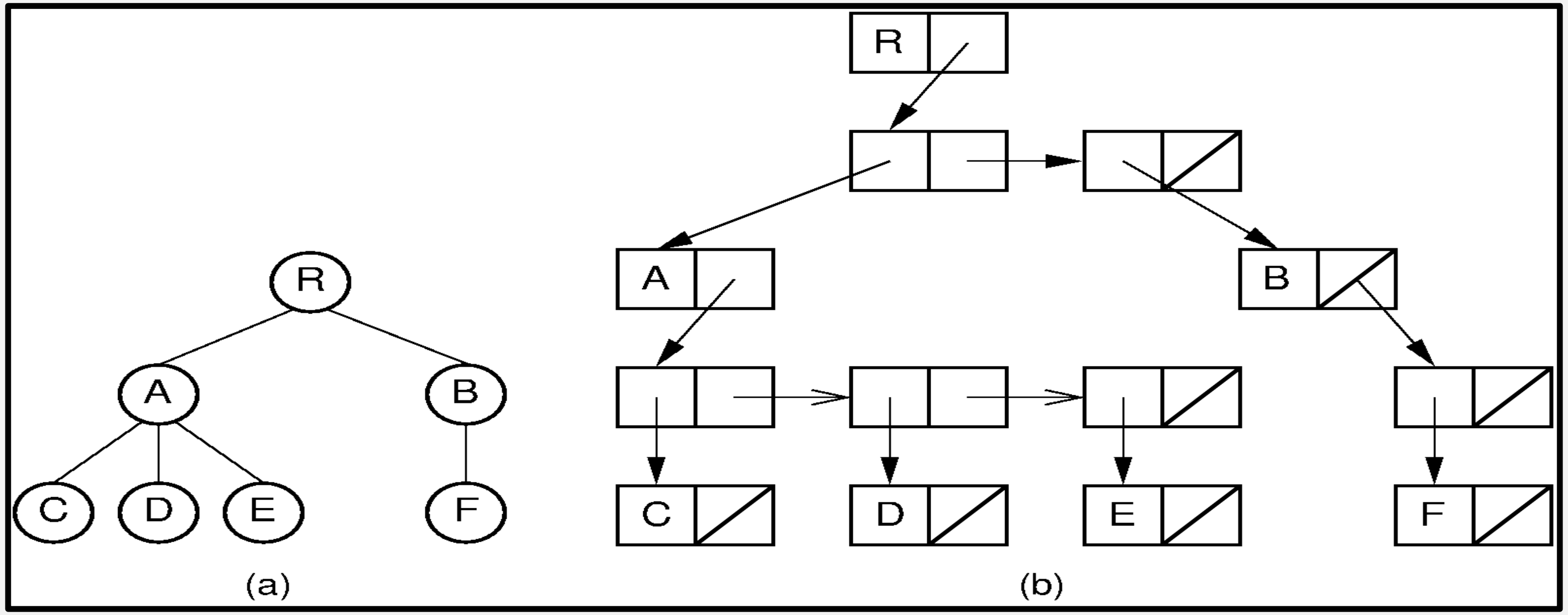
- If the node is at i-th index
  - ✓ Left child at: [(2*i) +1]
  - ✓ Right child at: [(2* i)+2]
  - ✓ Parent: floor [(i-1)/2]

# TREE REPRESENTATION: ARRAY



# TREE REPRESENTATION: SINGLE LINKED LIST

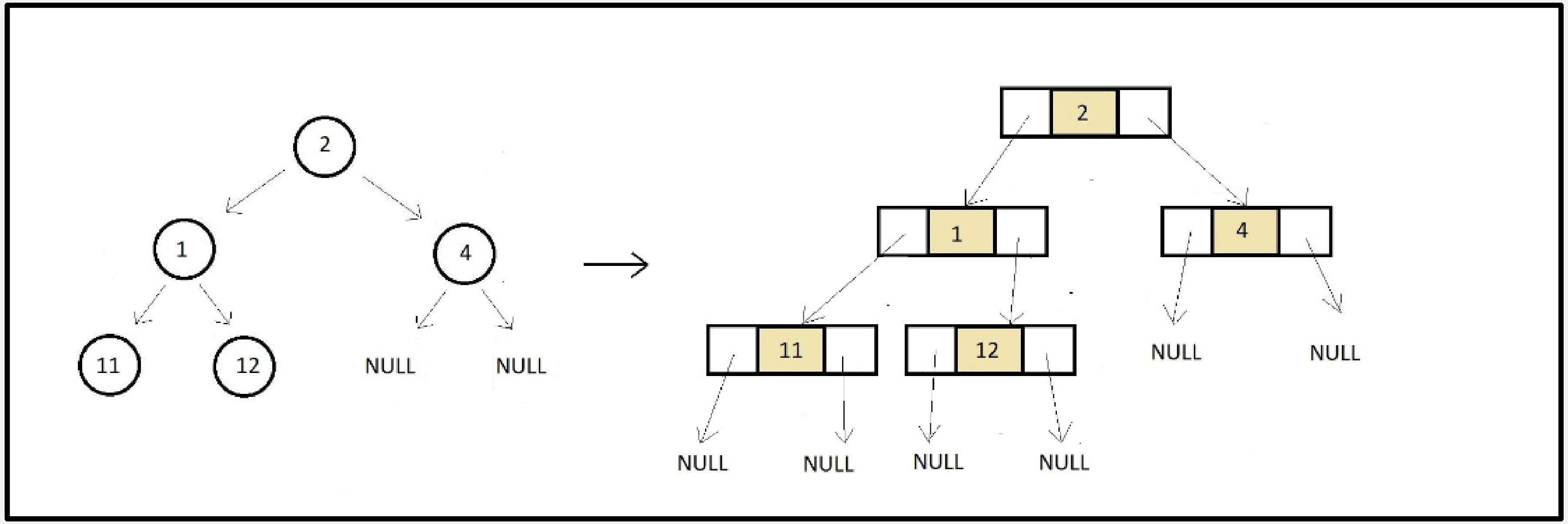
- **Two types of nodes are used**: one for representing the node with data called **'data node'** and another for representing only references called **'reference node'.**



# TREE REPRESENTATION: DOUBLE LINKED LIST

- In a doubly-linked list, every node consists of **three fields**. The **first field** is for storing **the left child address**, the **second** for storing **actual data**, and the **third** for storing **the right child address.**

| Left Child Address | Data | Right Child Address |
|---|---|---|



# END OF CHAPTER 6 – PART1