Data Structures and Algorithms

Non-Linear Data Structures

A Day in the Life of a MERN Stack Developer

You are working as a MERN stack developer at a burgeoning tech startup, focusing on developing a complex social networking application. The application requires handling intricate connections between users, groups, and content, demanding a sophisticated approach to data management and retrieval.

To meet this challenge, you realize the necessity of mastering non-linear data structures: trees, graphs, and HashMaps.

In this lesson, you will dive into the world of Trees, utilizing them to manage hierarchical user data and content categorization, enabling efficient data traversal and management.

You will explore Graphs to represent and manipulate the complex network of user relationships and interactions within the social network.

A Day in the Life of a MERN Stack Developer

Furthermore, you will use HashMaps for rapid data access and storage, crucial for the real-time features of the application.

To achieve all of the above, along with some additional features, you will learn a few concepts in this lesson that will help you find a solution to the above scenario.

This knowledge is essential for you to effectively handle the data structure challenges of the social networking application, ensuring its performance, scalability, and user satisfaction.

Learning Objectives

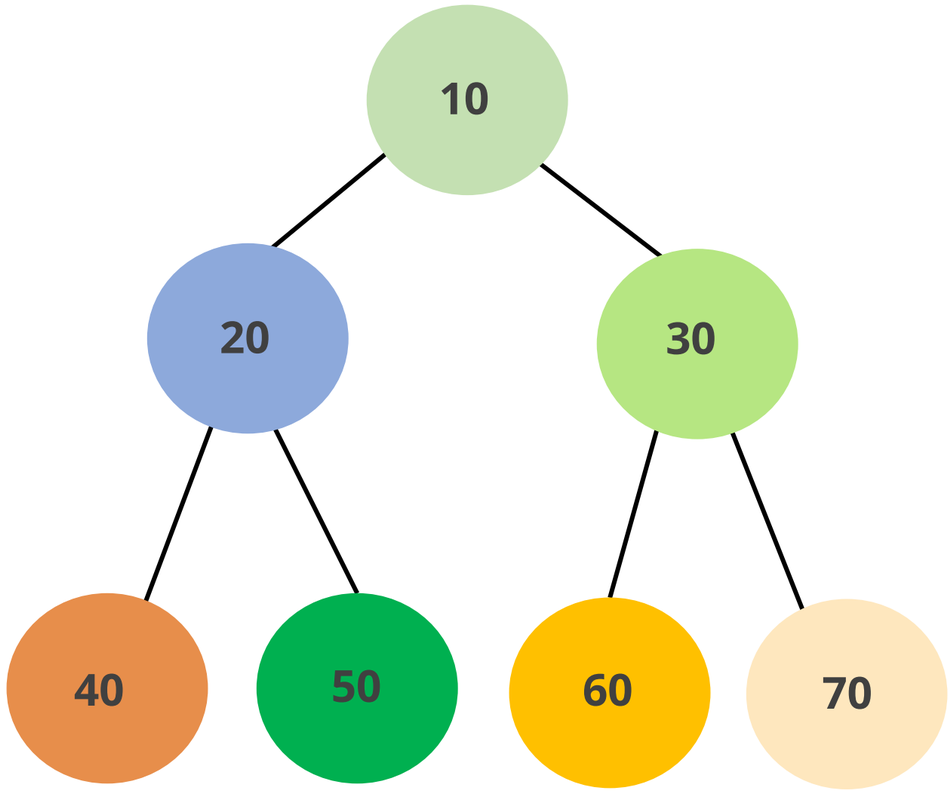
By the end of this lesson, you will be able to:

- Build various tree types and analyze their properties to understand their structural differences and performance
- Practice and assess different tree traversal methods for efficient data handling
- Create and distinguish between different graph types to apply suitable algorithms for specific problems
- Utilize traversal algorithms to solve graph-based problems for comprehensive exploration and optimal solutions

Introduction to Trees

What Is a Tree?

A tree is a non-linear data structure comprised of nodes connected by edges.



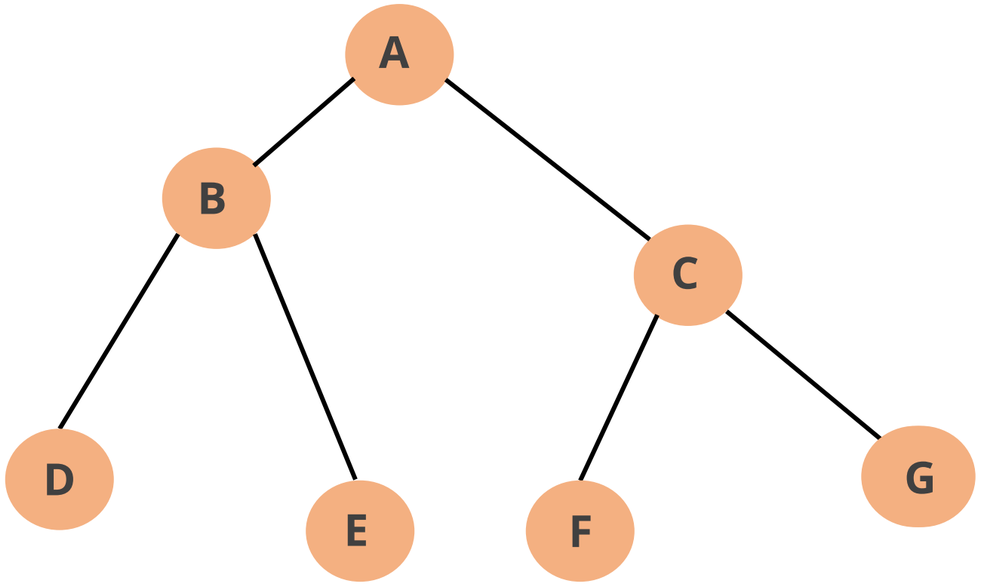
Why Tree Data Structure?

- **Hierarchical structure:** Trees effectively represent hierarchical data, ideal for applications like file systems and organizational charts.
- **Efficient search:** Balanced trees provide fast search operations, often with logarithmic time complexity.
- **Ordered access:** Trees, such as binary search trees, maintain elements in a sorted order, facilitating ordered data retrieval and range queries.
- **Dynamic nature:** Trees are dynamic, allowing for flexible growth and modification, unlike fixed-size arrays.

Tree Terminologies

Node

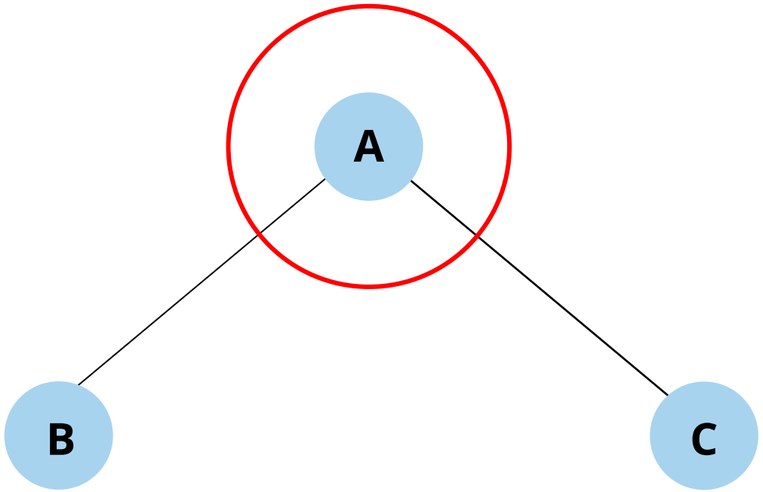
A node is a basic unit or element in data structures such as trees and linked lists.



In a tree, every element is a node, and a node can have multiple child nodes or none.

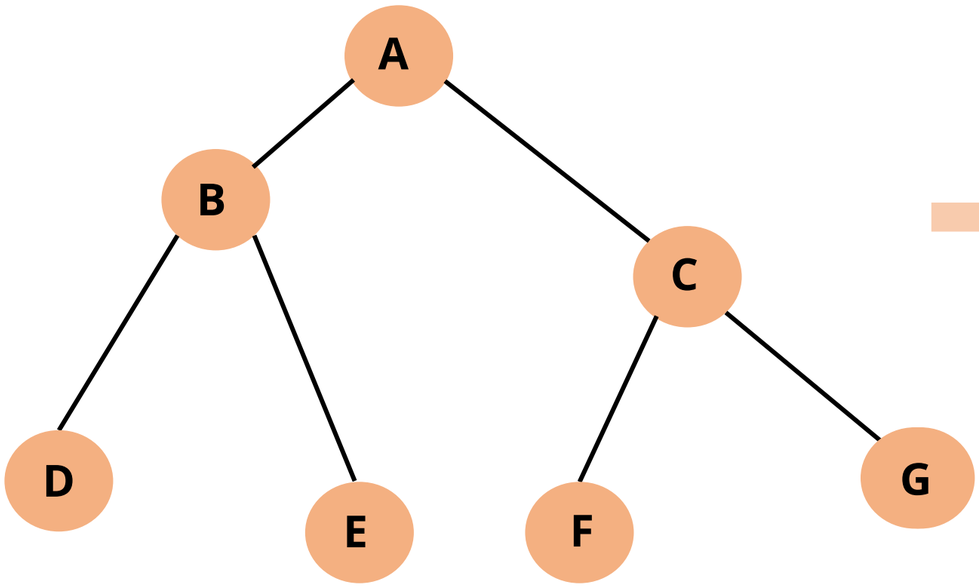
Root Node

The root node is the first and topmost node in a tree data structure.



Parent Node

A node that serves as the predecessor to another node is referred to as a parent node.

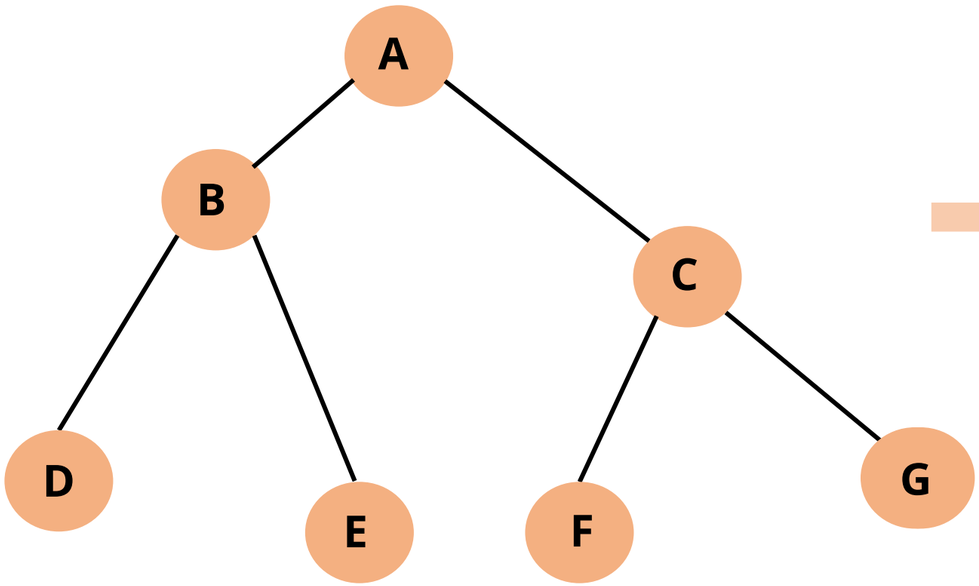


For example:

- Node A is the parent of nodes B and C.
- Node B is the parent of nodes D and E.
- Node C is the parent of nodes F and G.

Child Node

A node that is a descendant of another node is referred to as a child node.

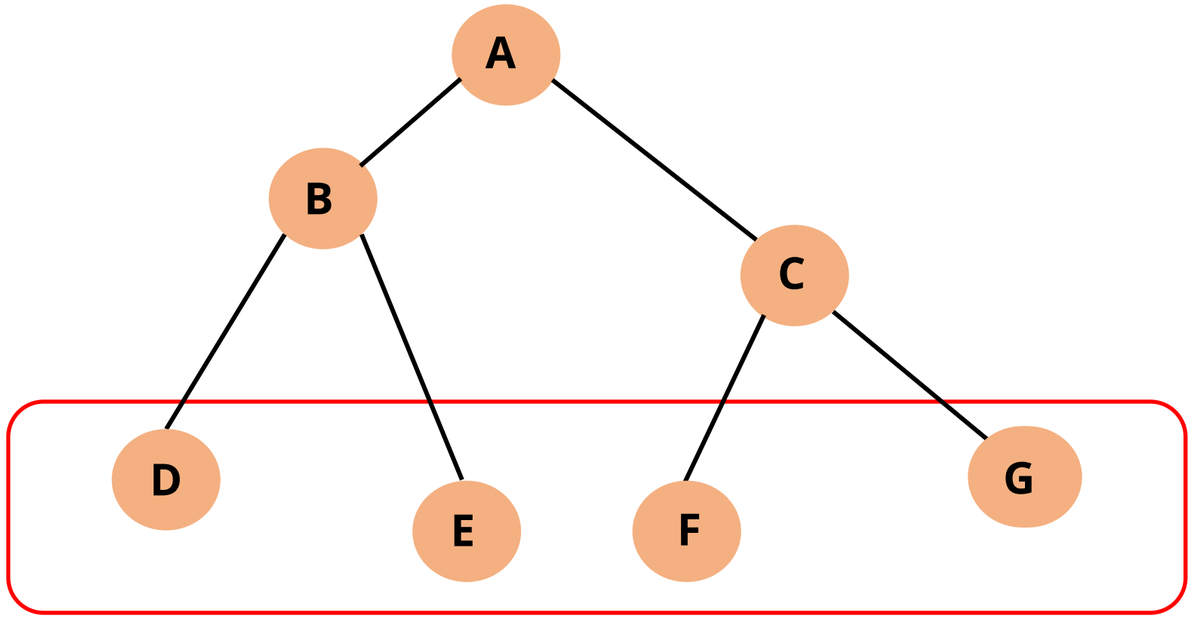


For example:

- Nodes B and C are the children of node A.
- Nodes D and E are the children of node B.
- Nodes F and G are the children of node C.

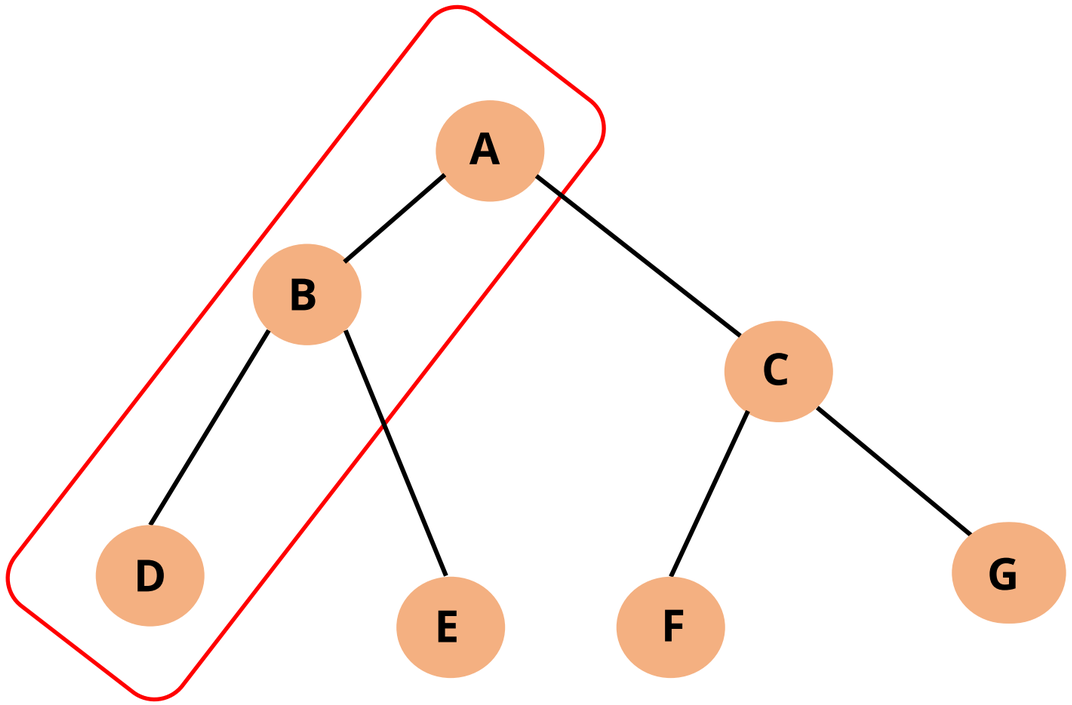
Leaf Node

A leaf node is a node in a tree that does not have any children.



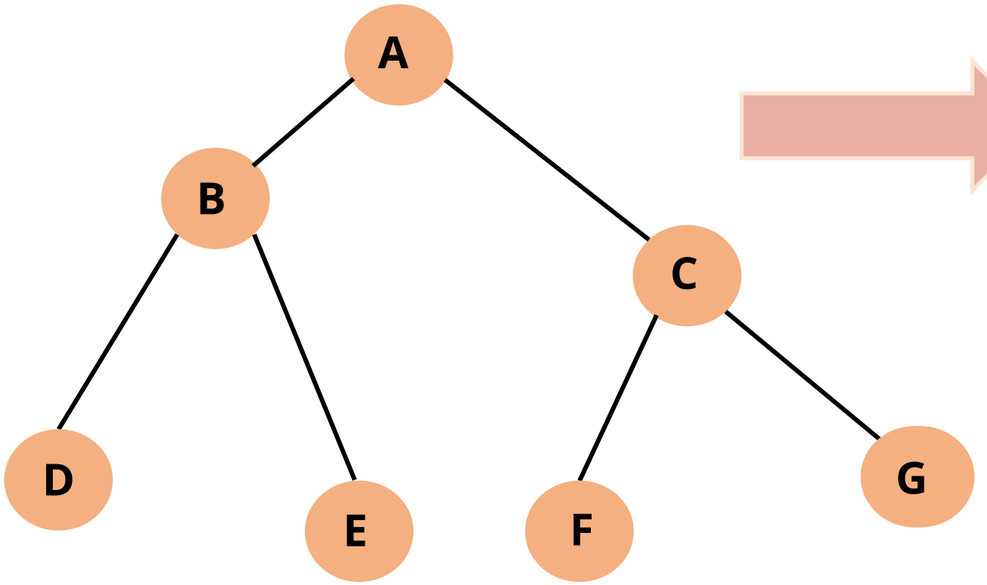
Internal Node

An internal node is a node in a tree that has at least one child node.



Degree

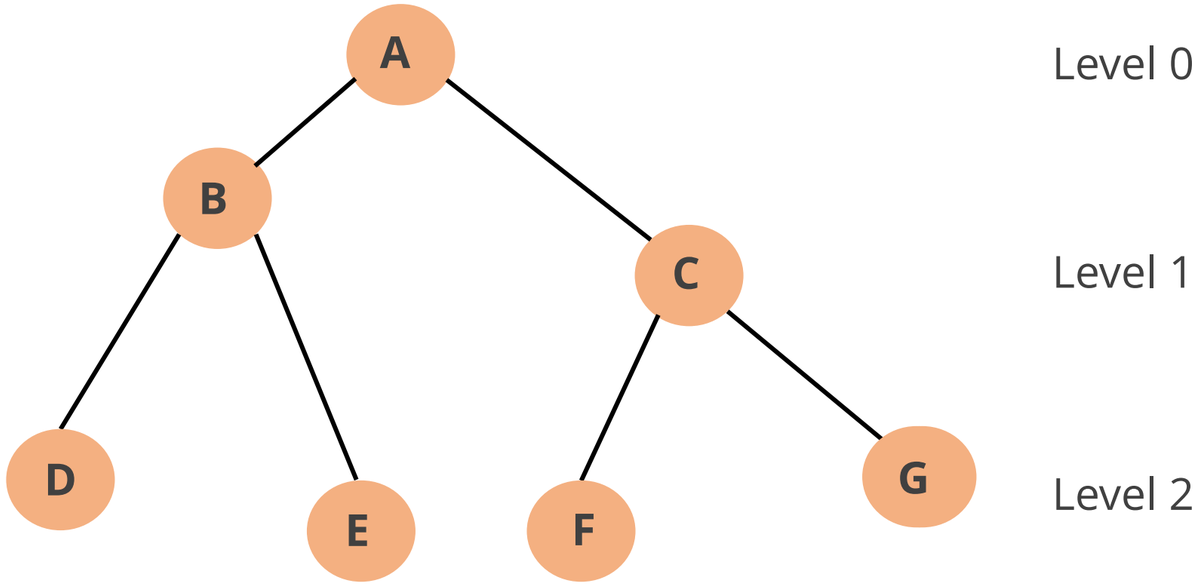
The degree of a node is the total number of its children.



- The degree of nodes A, B, and C is 2.
- The degree of nodes D, E, F, and G is 0.

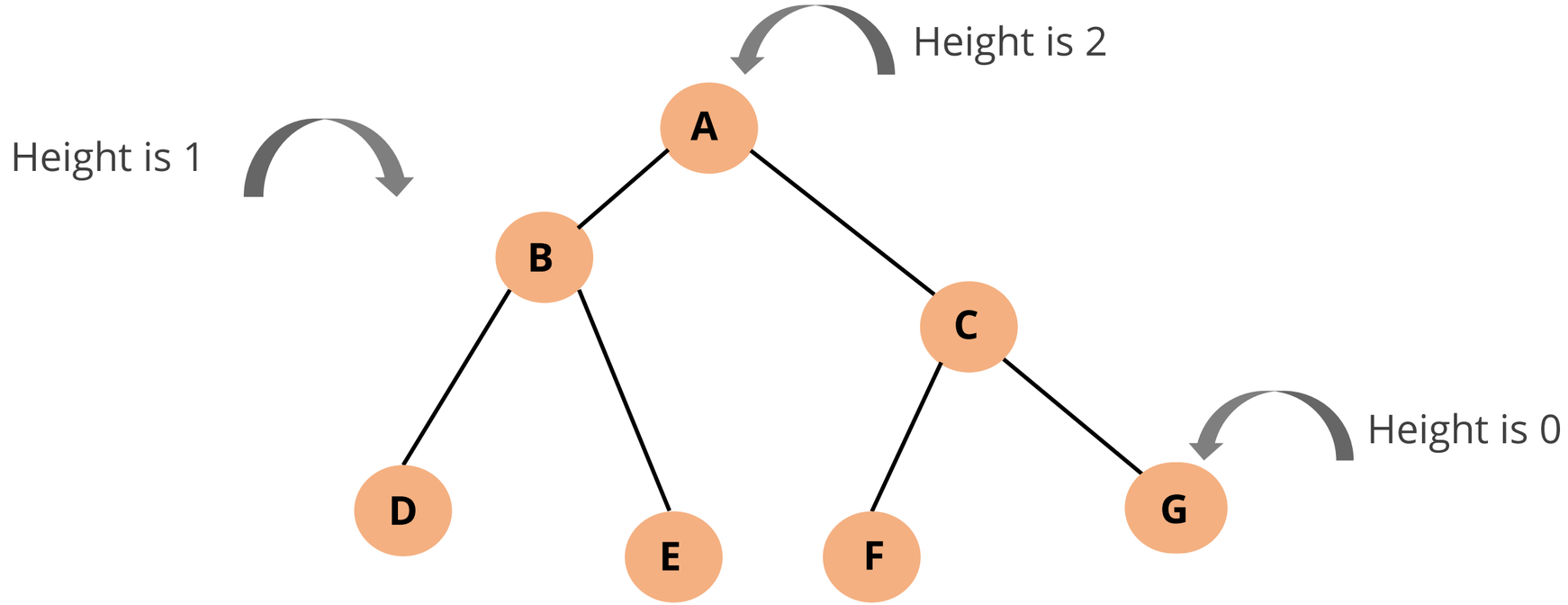
Levels

In a tree structure, the root node is at level 0. The children of the root node are at level 1, and similarly, the children of nodes at level 1 are at level 2.



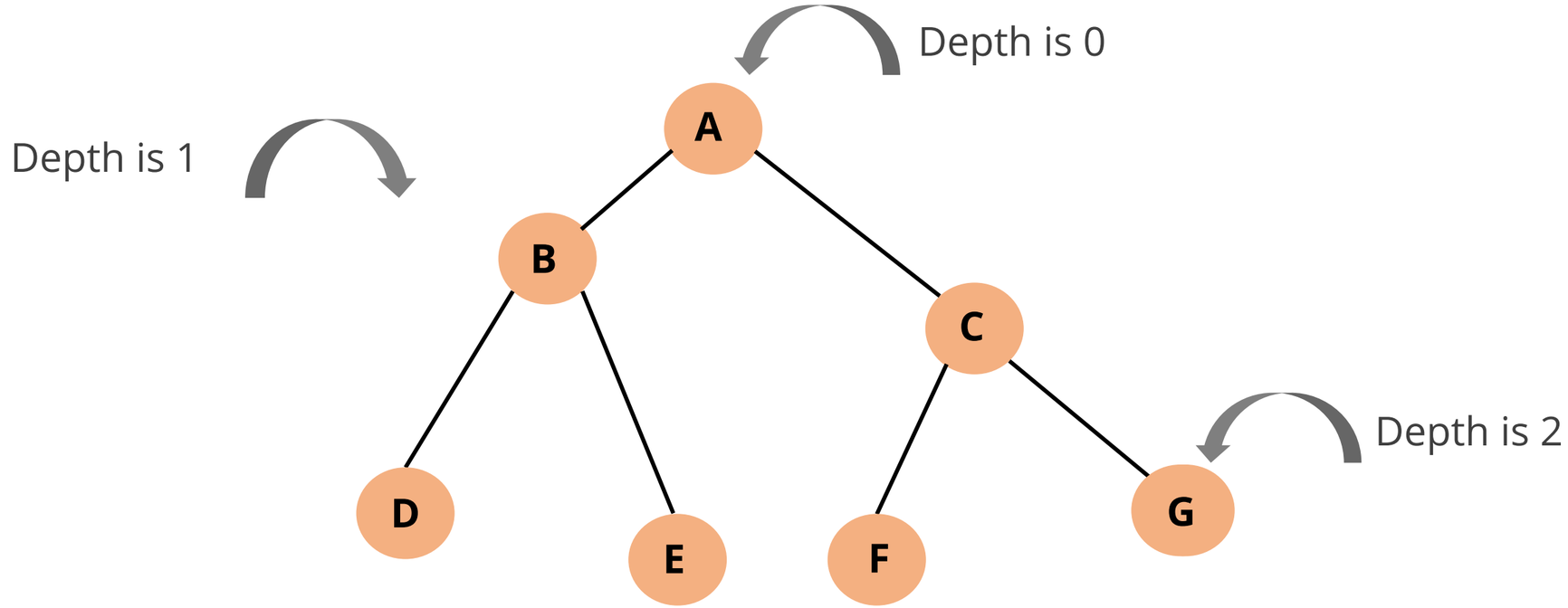
Height of a Node

The height of a node in a tree is the number of edges on the longest path from the node to a leaf node.



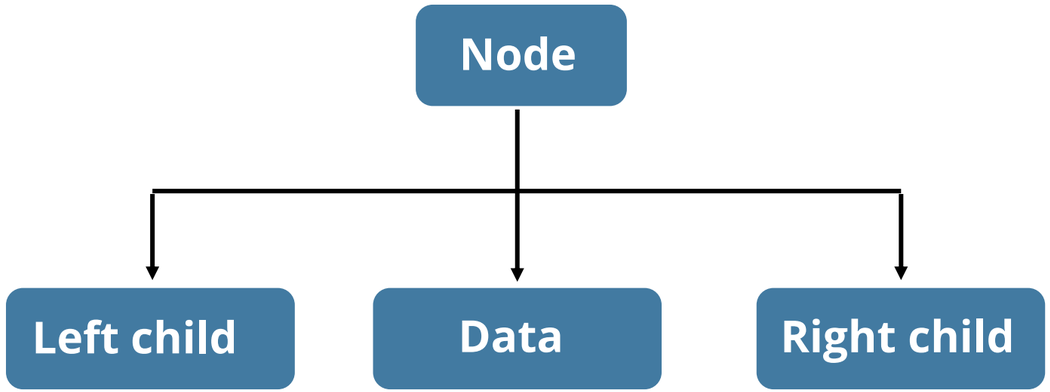
Depth of a Node

The depth of a node in a tree is the number of edges from the root node to that particular node.



Components of a Tree Node

Node Components

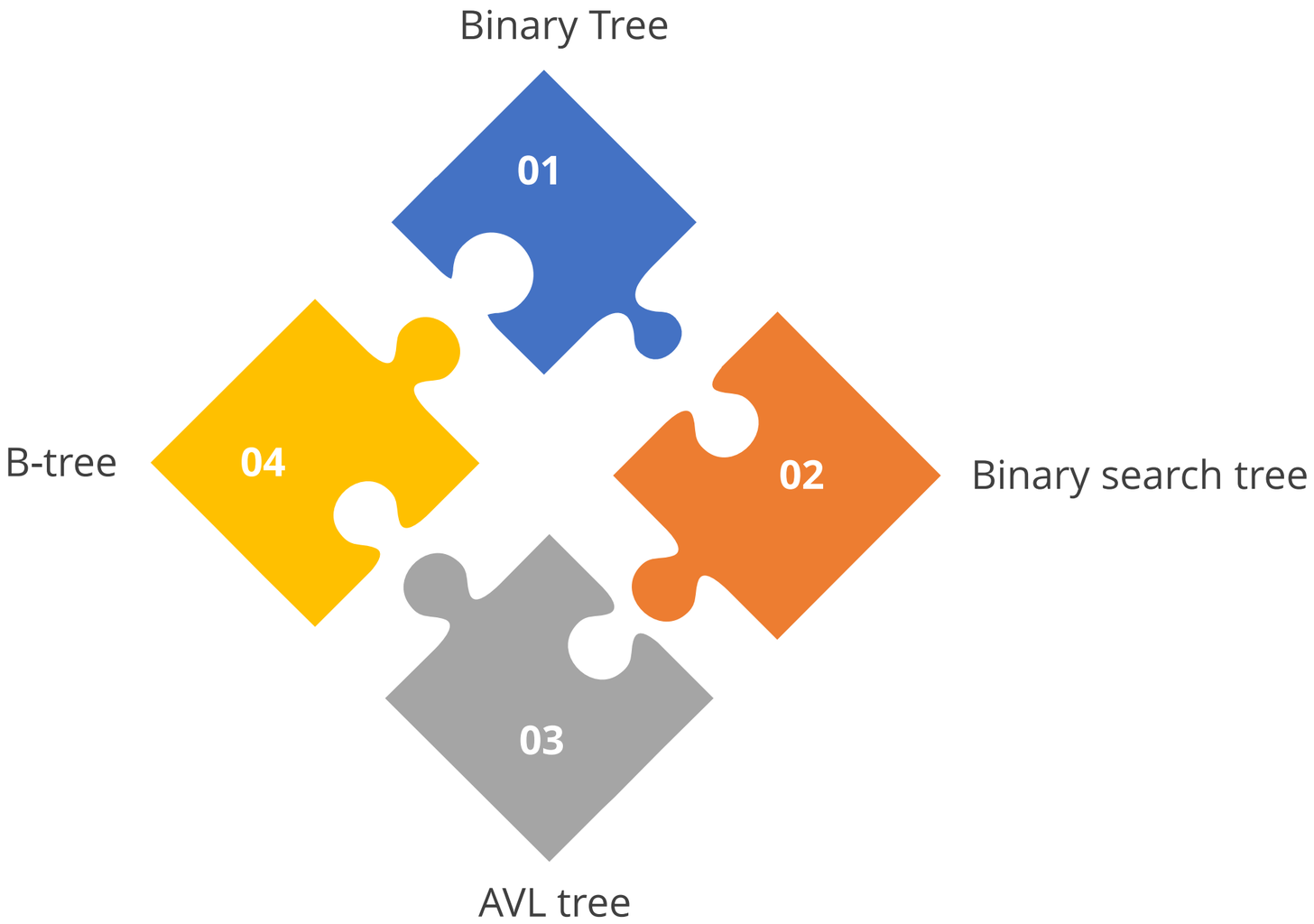


- **Data:** Each node contains data or a value. This can be a single value, such as a number or a character, or more complex data, like an object.

- **References:** They are used to establish connections between nodes in a tree. References are essentially links that indicate the parent-child relationships within the tree.

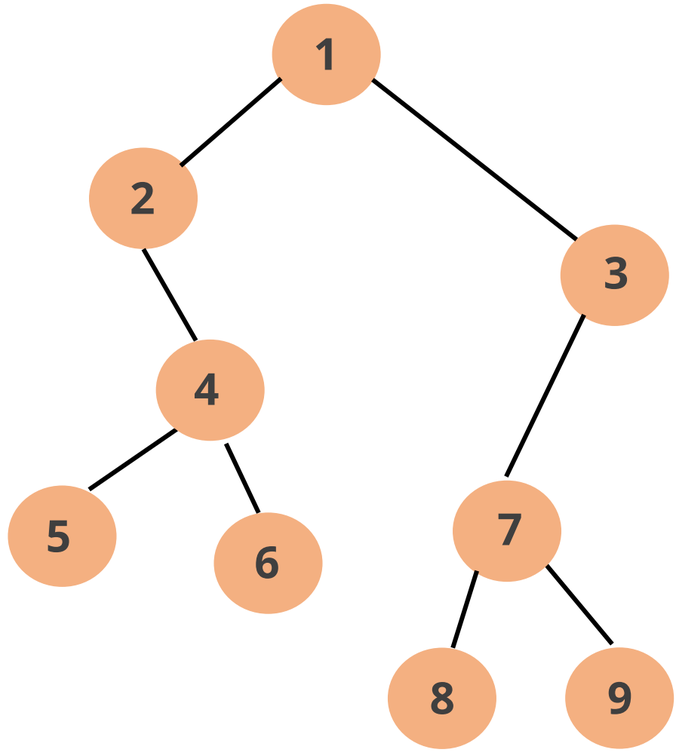
Types of Tree Data Structures

Types of Tree



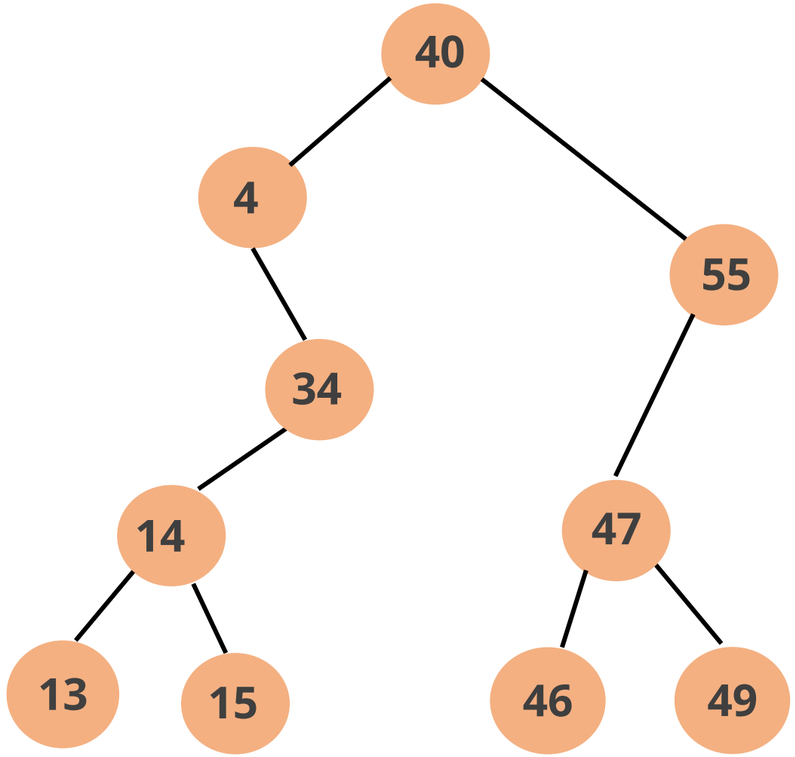
Binary Tree

A binary tree is a tree data structure in which each parent node has at most two children.



Binary Search Tree

A binary search tree, or BST, is a tree data structure in which each node can have a maximum of two children.

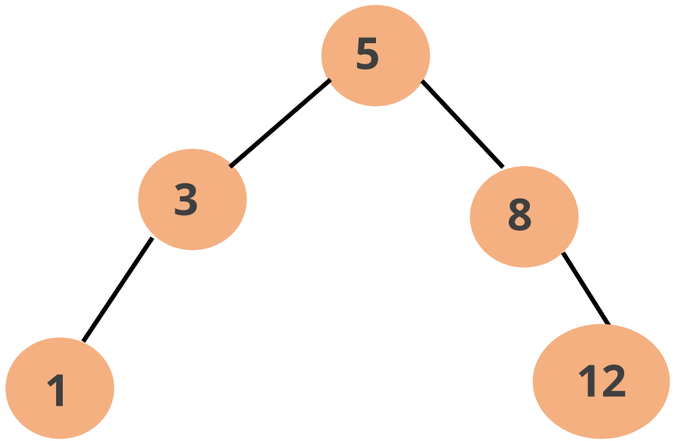


All nodes in the left subtree are less than the root node.

All nodes in the right subtree are greater than the root node.

AVL Tree

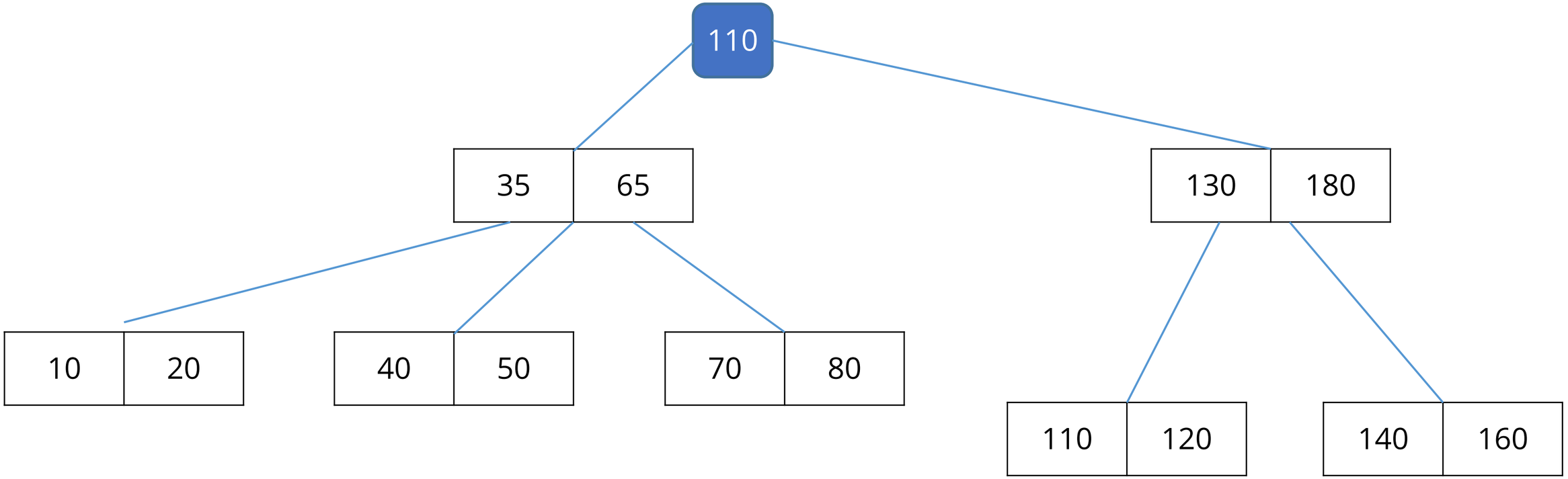
An AVL tree is a self-balancing binary search tree where the height difference between the left and right subtrees of any node is kept at most one. This property ensures that searches, insertions, and deletions have a time complexity of O(log n).



Balance factor = (Height of left subtree) - (Height of right subtree) or (Height of right subtree) - (Height of left subtree).

B-Tree

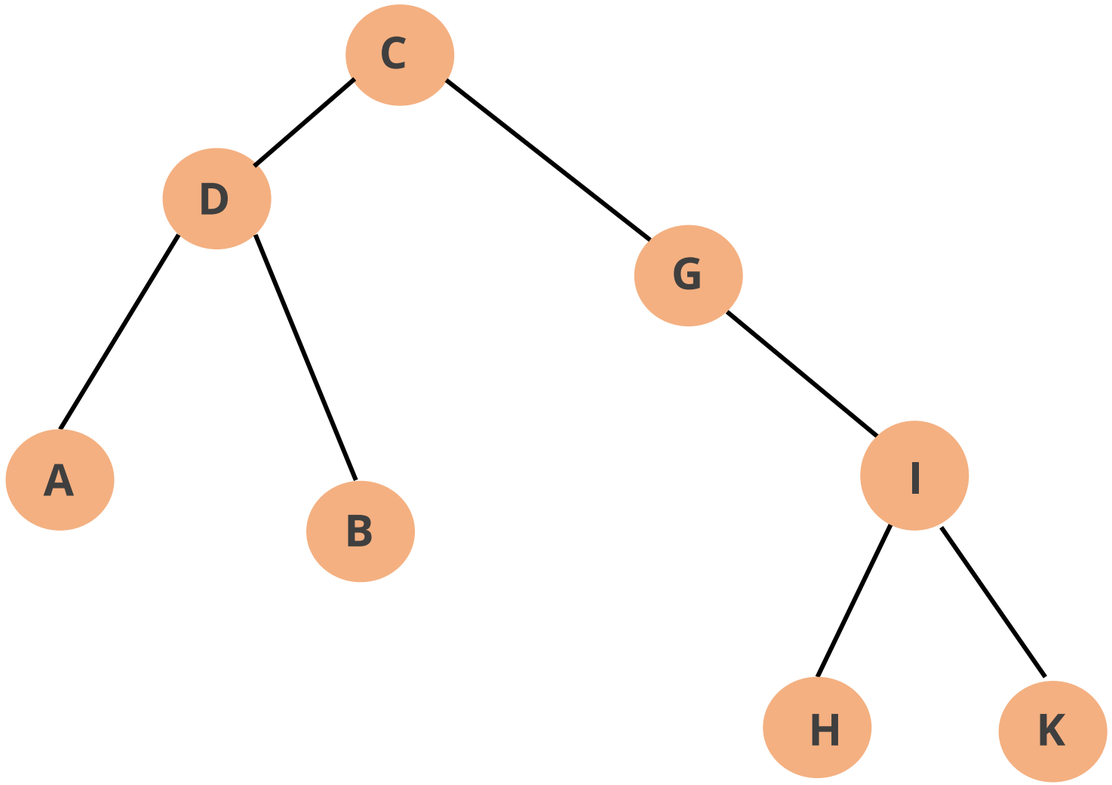
A B-tree is a special type of self-balancing search tree in which each node can hold more than one key and can have more than two children.



A B tree is also known as a height-balanced m-way tree.

Tree Traversal

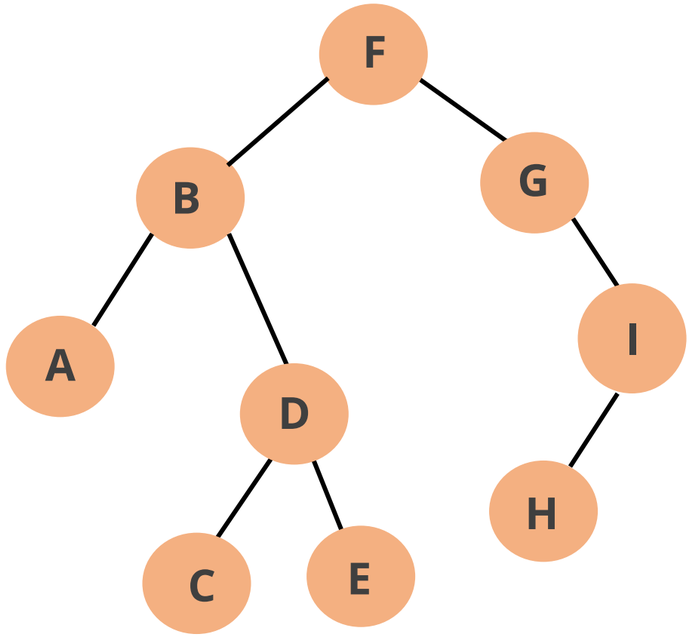
Traversing a tree data structure helps in visiting the required nodes in the tree to perform specific operations.



Tree Traversal

Pre-order traversal:

- Visit the root node
- Visit all nodes on the left side
- Visit all nodes on the right side

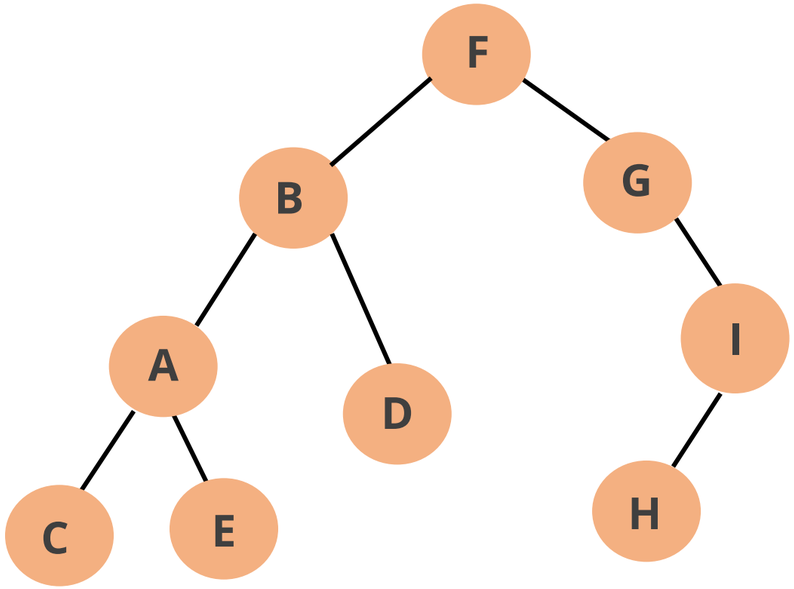


F	B	A	D	C	E	G	I	H

Tree Traversal

In-order traversal:

- First, visit all nodes on the left side
- Visit the root node
- Visit all nodes on the right side

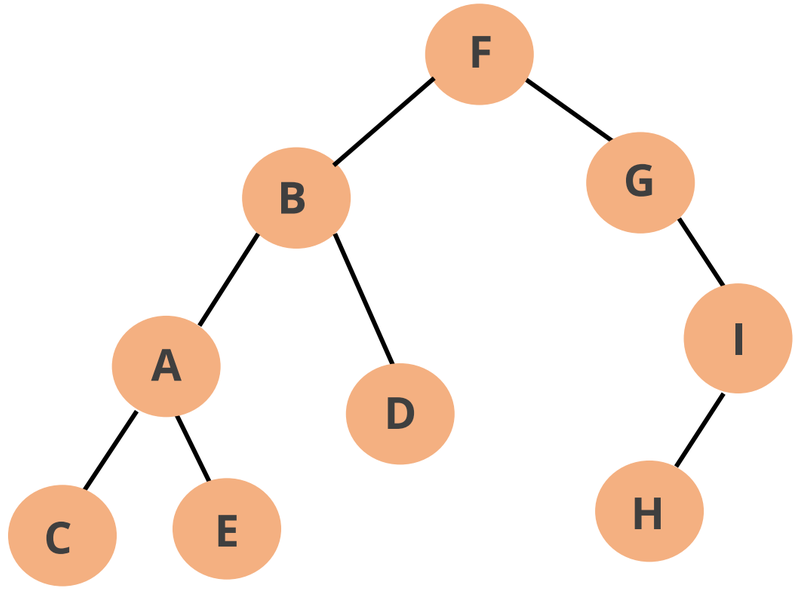


C	A	E	B	D	F	G	H	I

Tree Traversal

Post-order traversal:

- First, visit all nodes on the left side
- Visit all nodes on the right side
- Visit the root node



C	E	A	D	B	H	I	G	B

Application of Tree

01 Binary search trees are used to quickly check whether an element present in a set or not.

02 The most popular database uses B-tree, which is a variant of tree data structure.

03 The modified version of trees is called Tries is used in the router to store routing information.

04 The compiler uses a syntax tree to validate the syntax of every program written.

Assisted Practice

Building and Traversing Binary Tree **Duration: 20 Min.**

Problem Statement:

You have been assigned a task to demonstrate the creation and traversal of a binary tree using JavaScript

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

Assisted Practice

Working with Binary Tree **Duration: 20 Min.**

Problem Statement:

You have been assigned a task to demonstrate the important methods for a binary tree in JavaScript

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

Assisted Practice

Implementing AVL Tree **Duration: 20 Min.**

Problem Statement:

You have been assigned a task to demonstrate the implementation of a AVL tree in JavaScript

Assisted Practice: Guidelines

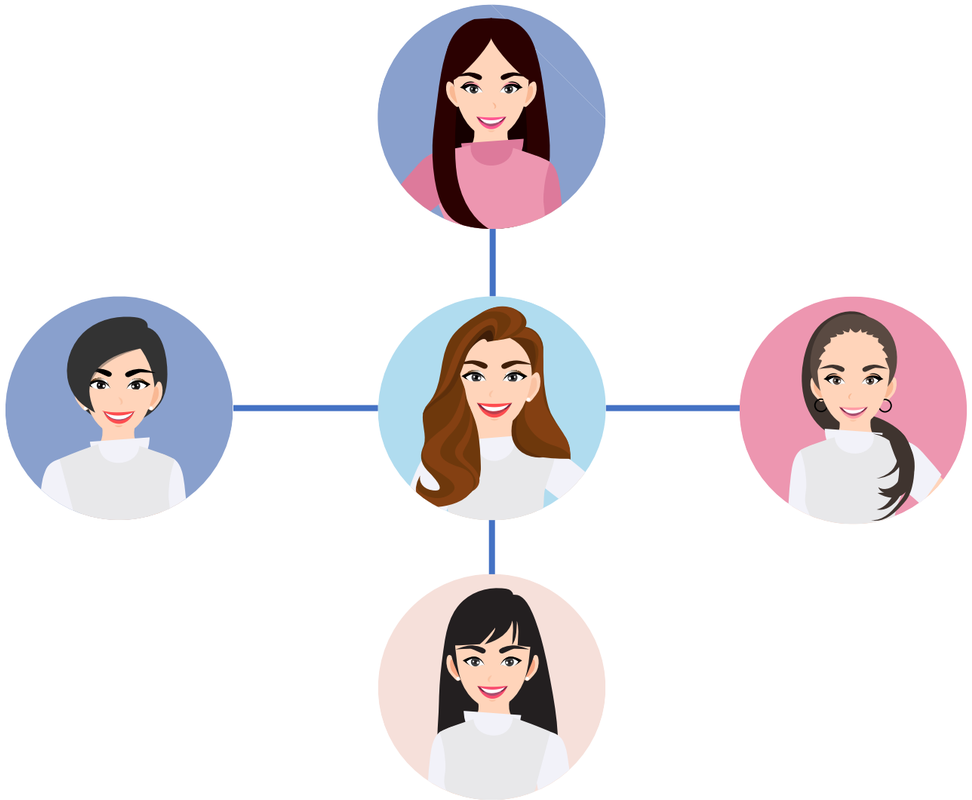
Steps to be followed:

1. Create and execute the JS file

Introduction to Graphs

Graph

A graph is a non-linear data structure consisting of finite sets of vertices connected by a collection of edges.

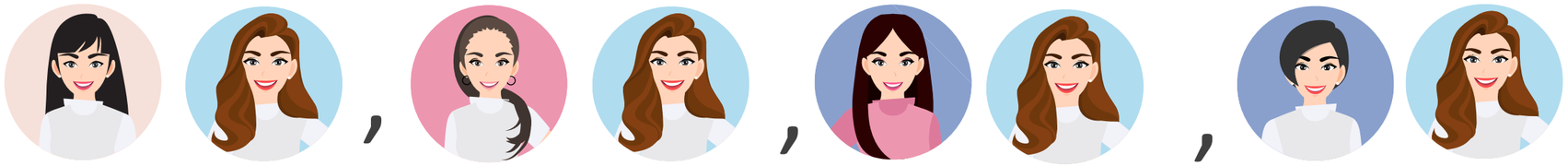


G

Graph

A graph, denoted as **G**, is defined as a pair of sets: **(V, E)**, where **V** represents the set of vertices, and **E** represents the set of edges connecting these pairs of vertices.

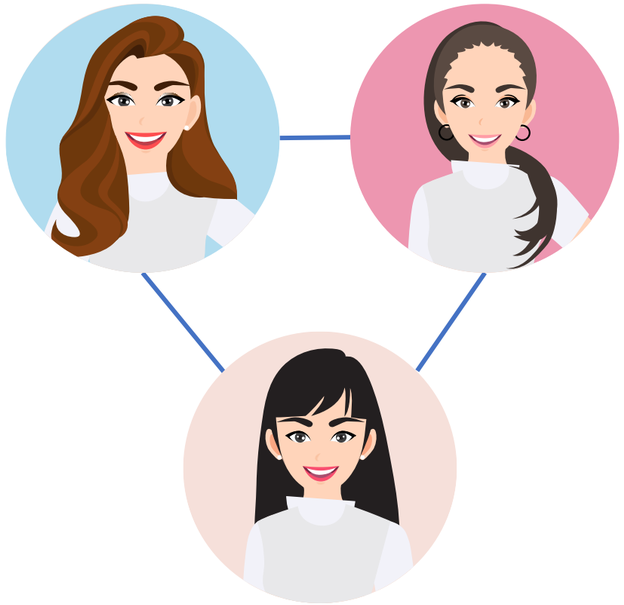
E = 

V = 

In mathematical notation: **G = (V, E)**

Graph Terminologies

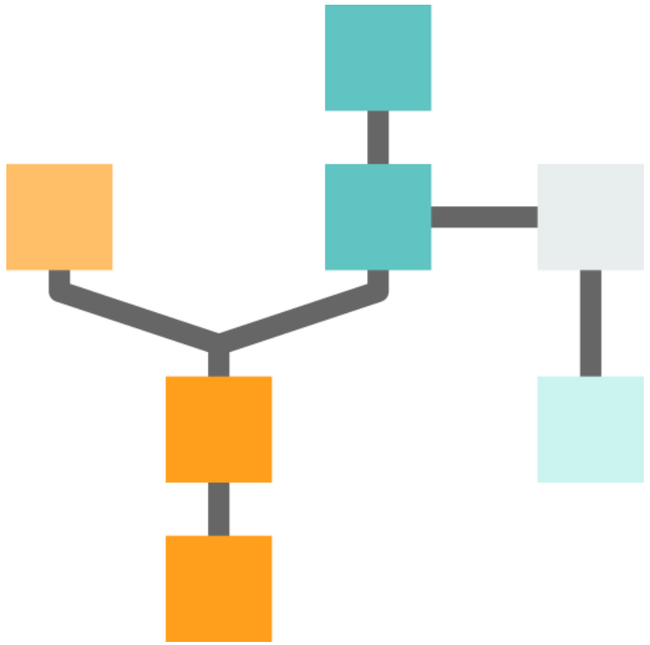
Graph Terminologies



Adjacency

- **Adjacent vertices:** Refers to vertices that share an edge.
- **Adjacent edges:** Describes edges that have a common vertex.

Graph Terminologies



Degree

In an undirected graph, the degree is defined as the number of vertices adjacent to a given vertex.

Graph Terminologies

Path

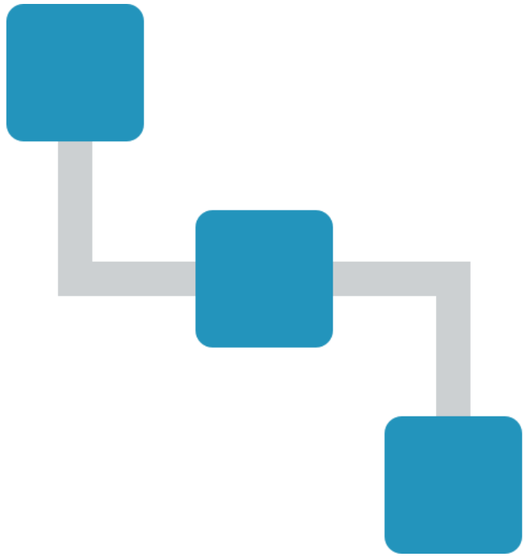
A path is defined as a sequence of distinct vertices, where two consecutive vertices are adjacent to each other.

Graph Terminologies

Cycle

A cycle is a path in which only one vertex is repeated, and that repeated vertex is both the first and last vertex.

Graph Terminologies



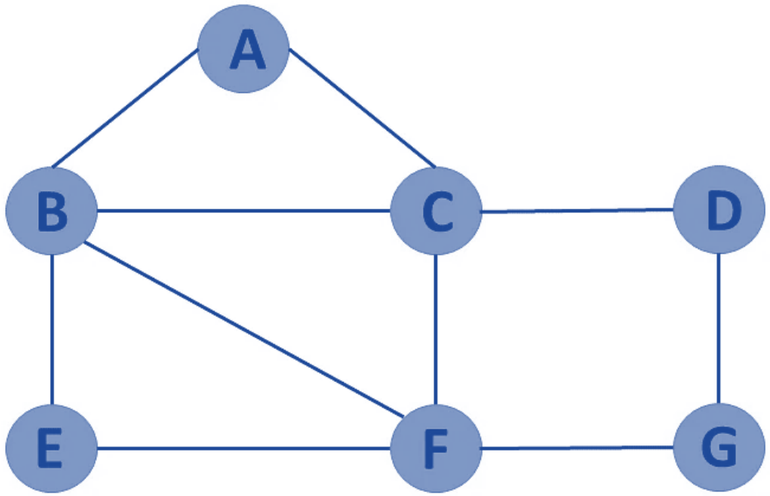
Walk

A walk is the sequence of vertices and edges of the graph.

Types of Graph

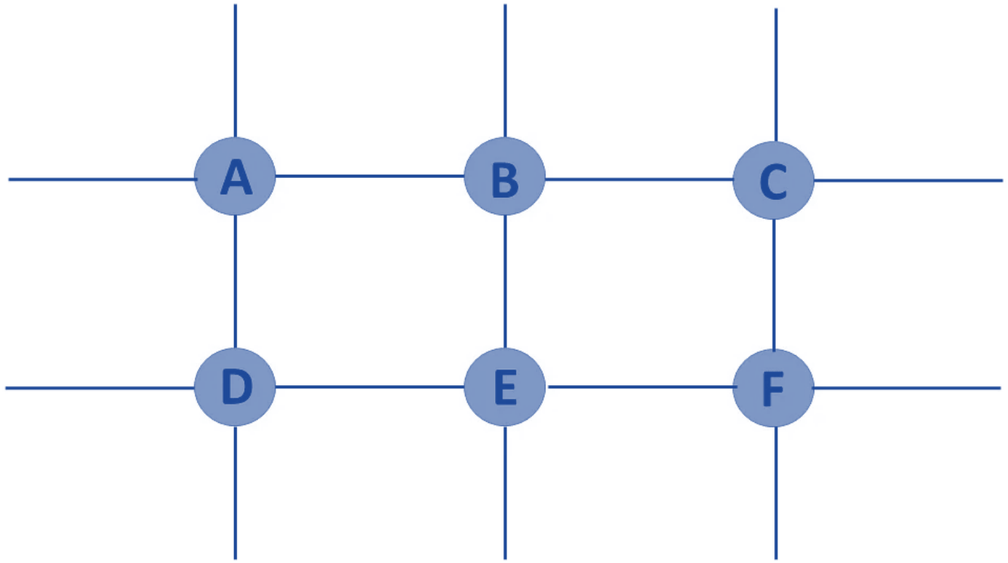
Finite Graph

A finite graph, denoted as **G = (V, E)**, is a graph where the sets of vertices **(V)** and edges **(E)** are both finite.



Infinite Graph

An infinite graph, represented as **G = (V, E)**, is a graph with an unbounded, or limitless, number of vertices **(V)** and edges **(E)**.

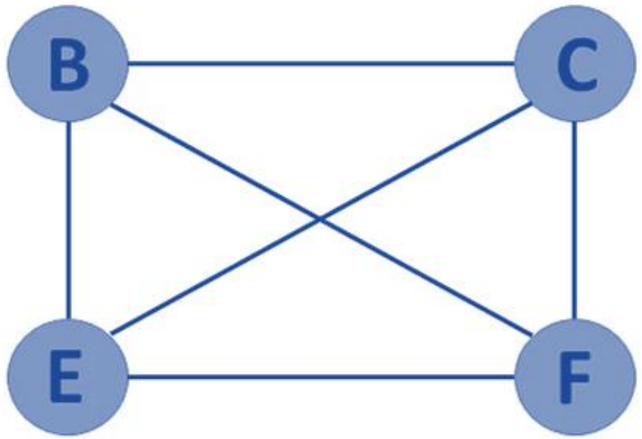


Trivial Graph

A trivial graph, denoted as **G = (V, E)**, consists of a single vertex with no edges.

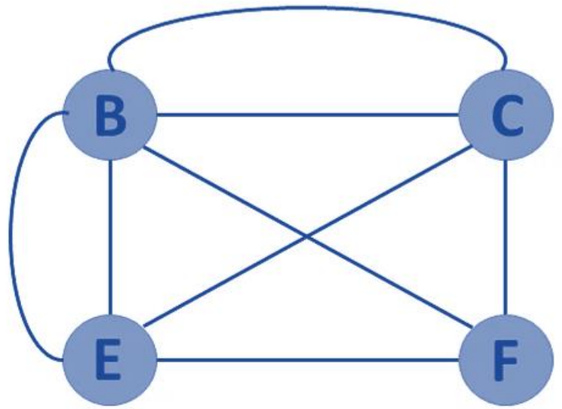
Simple Graph

A simple graph, represented as **G = (V, E)**, is a graph where each pair of distinct vertices is connected by exactly one edge.



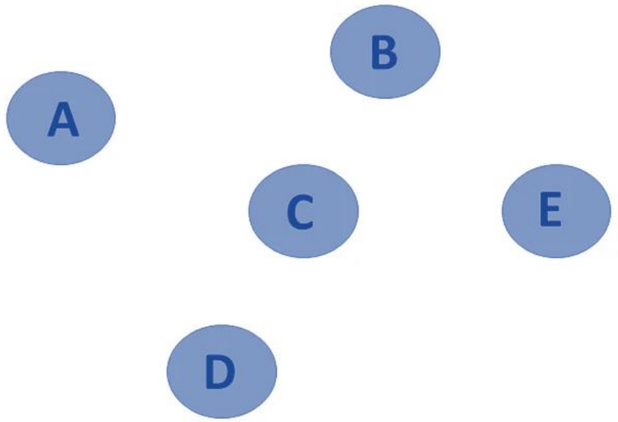
Multi Graph

A multigraph, denoted as **G = (V, E)**, is a graph where multiple edges can connect a pair of vertices.



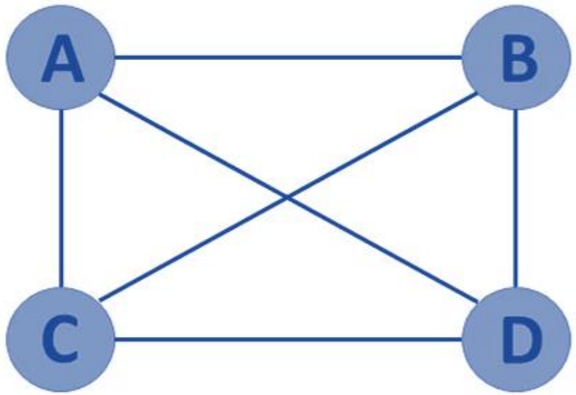
Null Graph

A null graph is a graph **G = (V, E)** that contains vertices but lacks edges, meaning no vertices are connected.



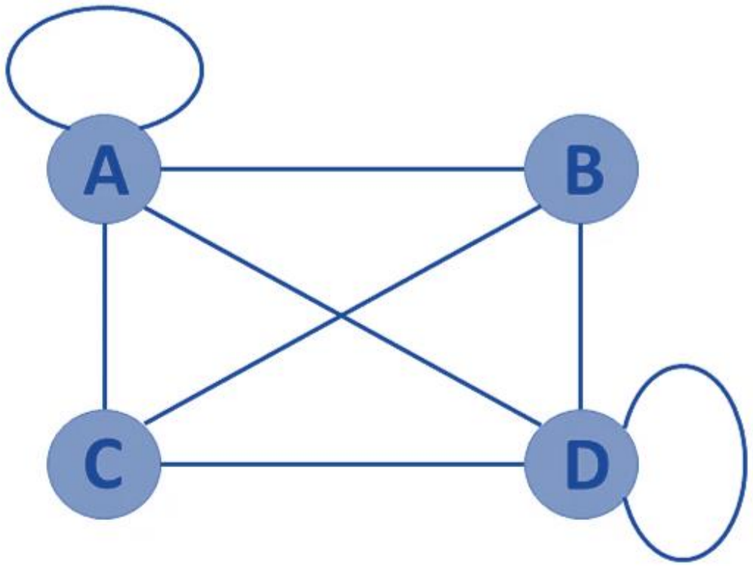
Complete Graph

A complete graph is a simple graph **G = (V, E)** where every pair of distinct vertices is connected by a unique edge.



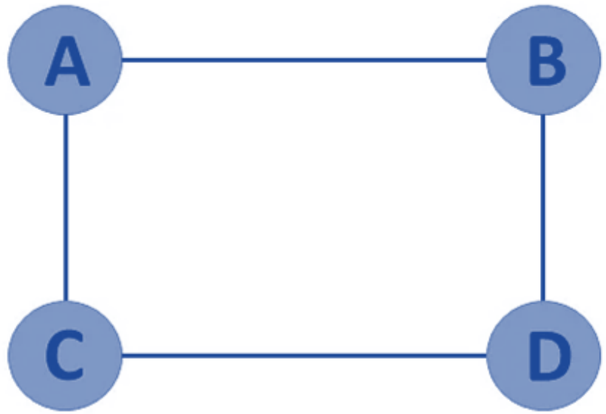
Pseudograph

A graph **G = (V, E)** is considered a pseudograph if, in addition to other edges, it contains a self-loop.



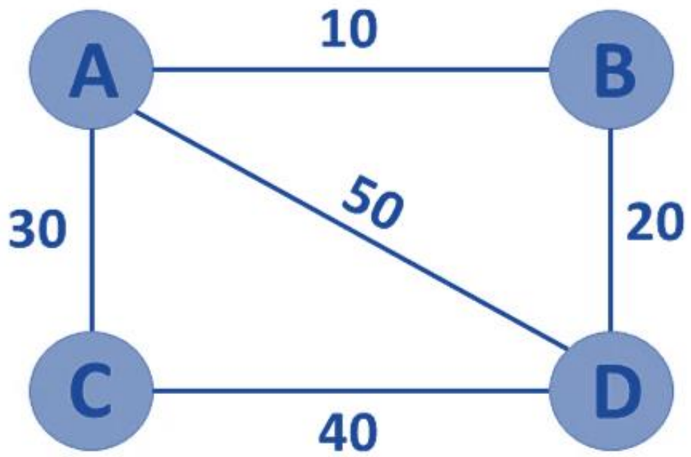
Regular Graph

A regular graph is a simple graph **G = (V, E)** where each vertex has the same number of edges, or the same degree.



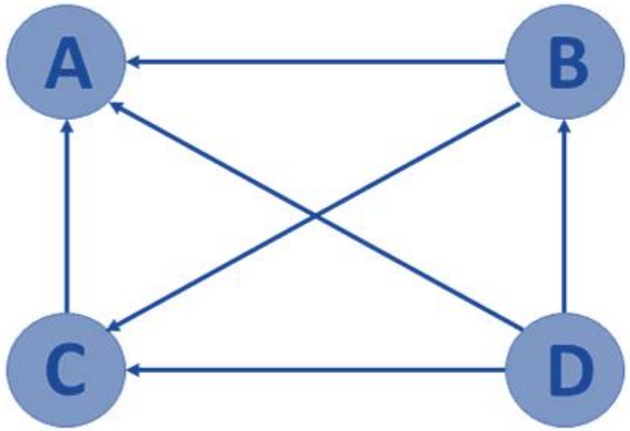
Weighted Graph

A weighted graph, denoted as **G = (V, E)**, is a graph where each edge is assigned a specific value or weight, representing the cost of traversing that edge.



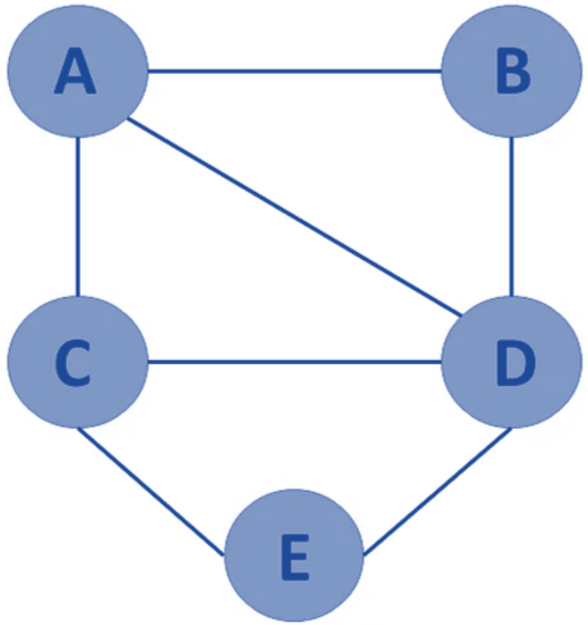
Directed Graph

A directed graph, or digraph, is a graph where the edges between nodes have a specific direction.



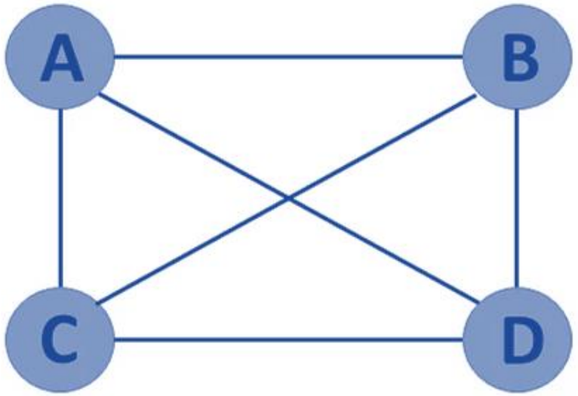
Undirected Graph

An undirected graph is a graph composed of a set of nodes connected by links, where these links have no directional orientation.



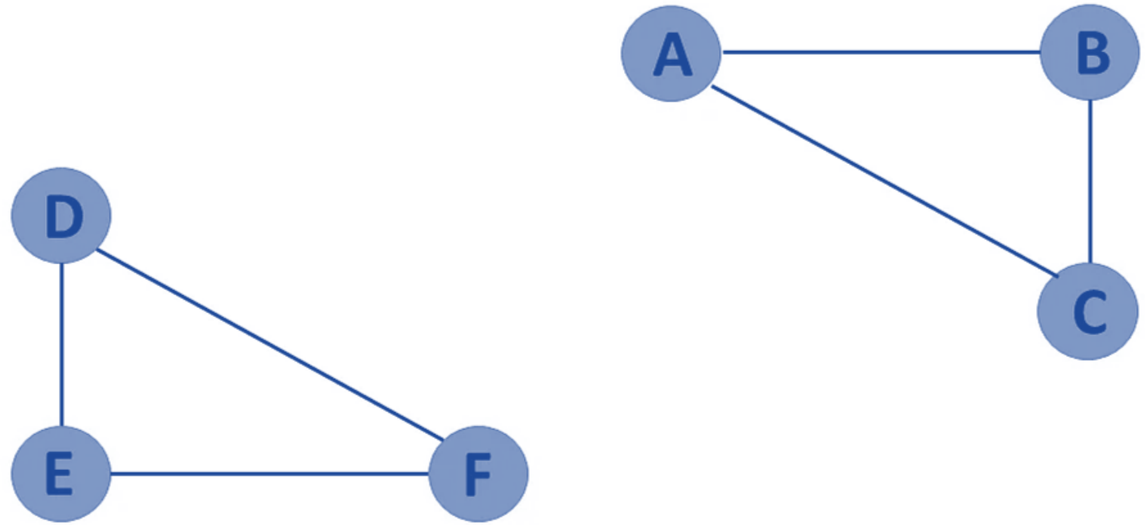
Connected Graph

A graph is considered connected if a path exists between every pair of vertices in the graph.



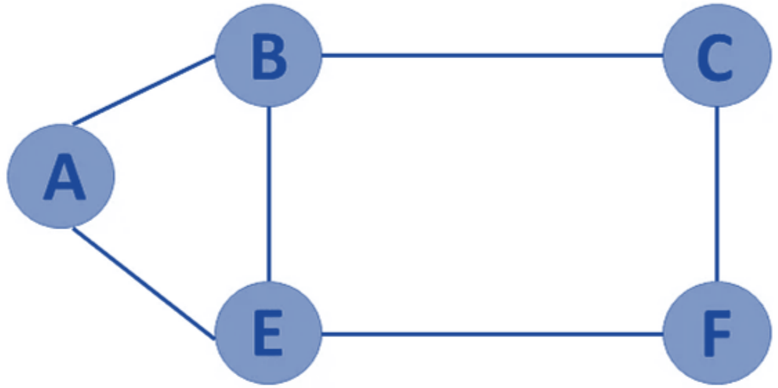
Disconnected Graph

A graph is considered disconnected when there are no edges connecting its vertices. In this case, it is often referred to as a null graph.



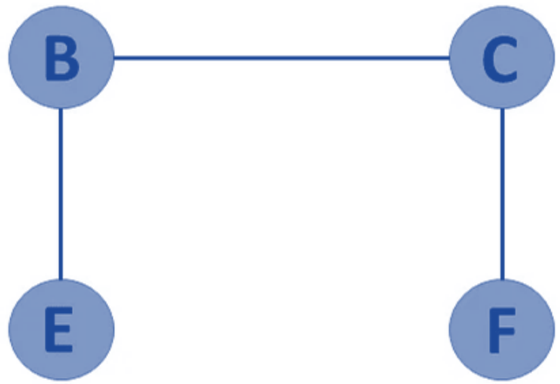
Cyclic Graph

A graph is considered cyclic if it contains at least one graph cycle.



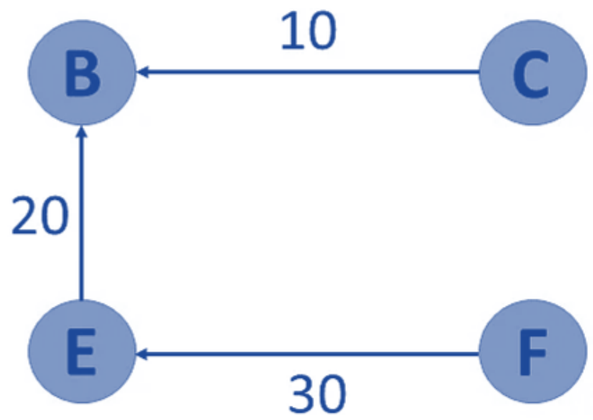
Acyclic Graph

An acyclic graph is defined as a graph that contains no cycles.



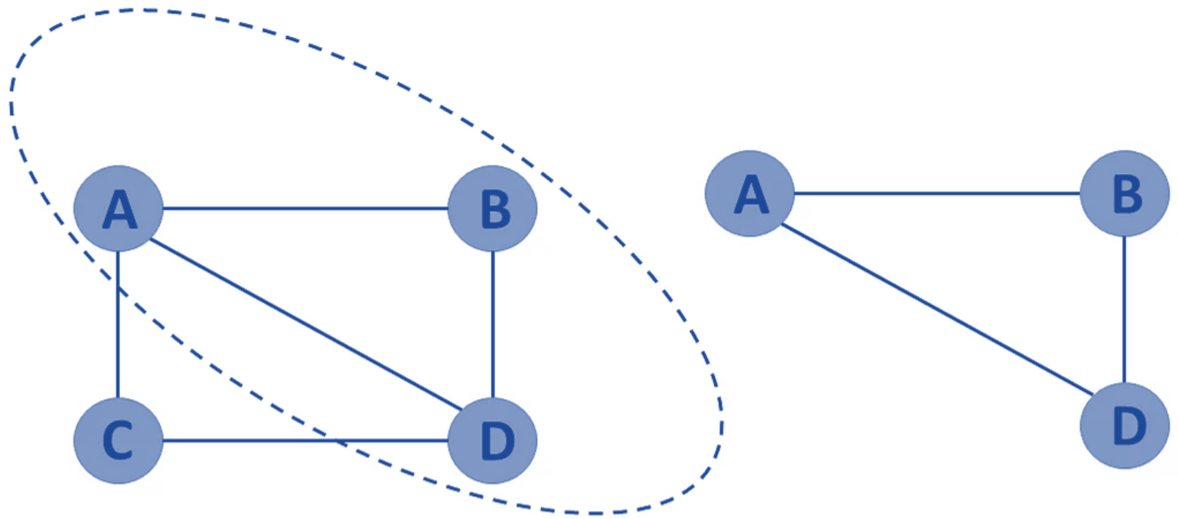
Directed Acyclic Graph

A directed acyclic graph (DAG) is a type of graph characterized by directed edges and the absence of cycles.



Subgraph

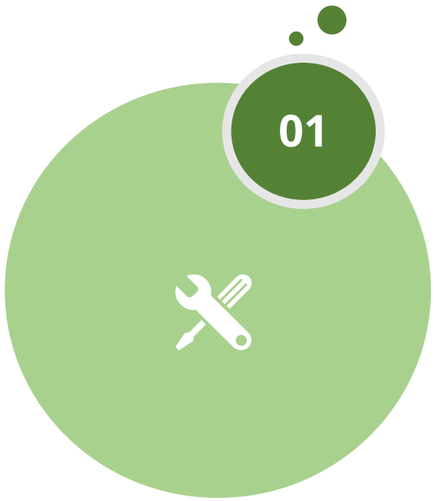
A subgraph consists of vertices and edges that form a subset of another graph.



Graph Representation

Graph Representation

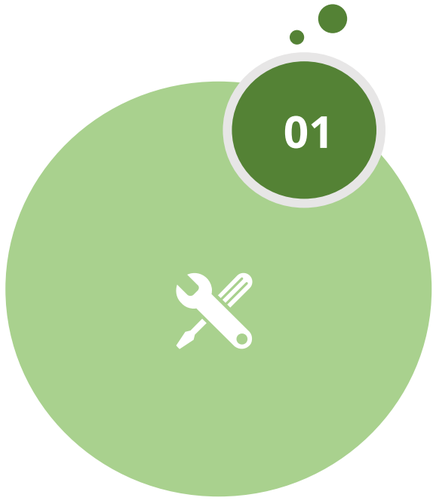
There are two ways to represent a graph data structure.



Adjacency matrix

Adjacency list

Graph Representation

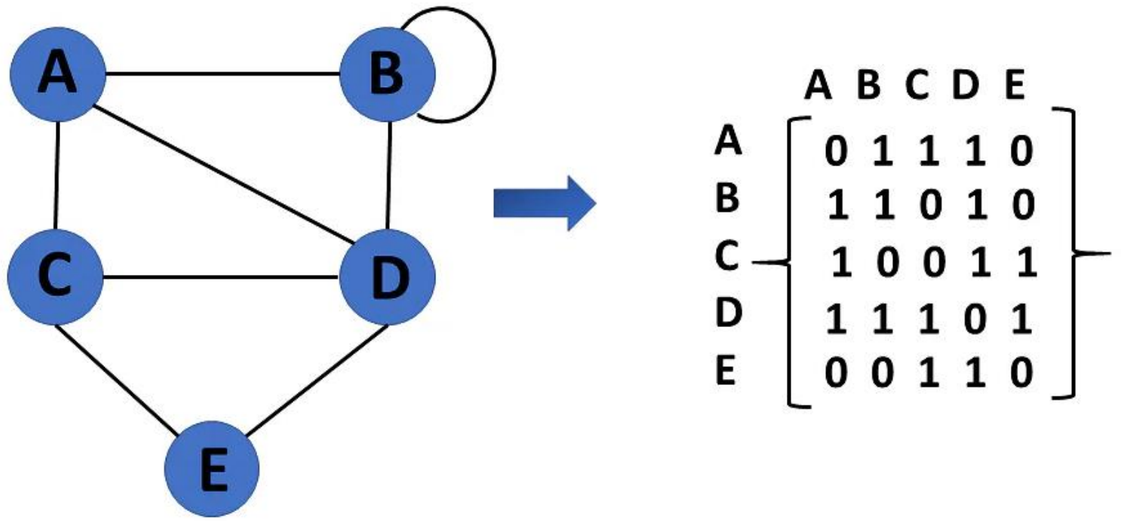


Adjacency matrix

- Adjacency matrix is a sequential representation.
- Adjacency matrix is used to represent which nodes are adjacent to each other.
- If there is an edge between two vertices, the value of the corresponding element of the graph is 1, otherwise 0.
- In a weighted graph with edge weights differing from simply 0s and 1s, storage of the actual weight for each edge is possible.

Graph Representation

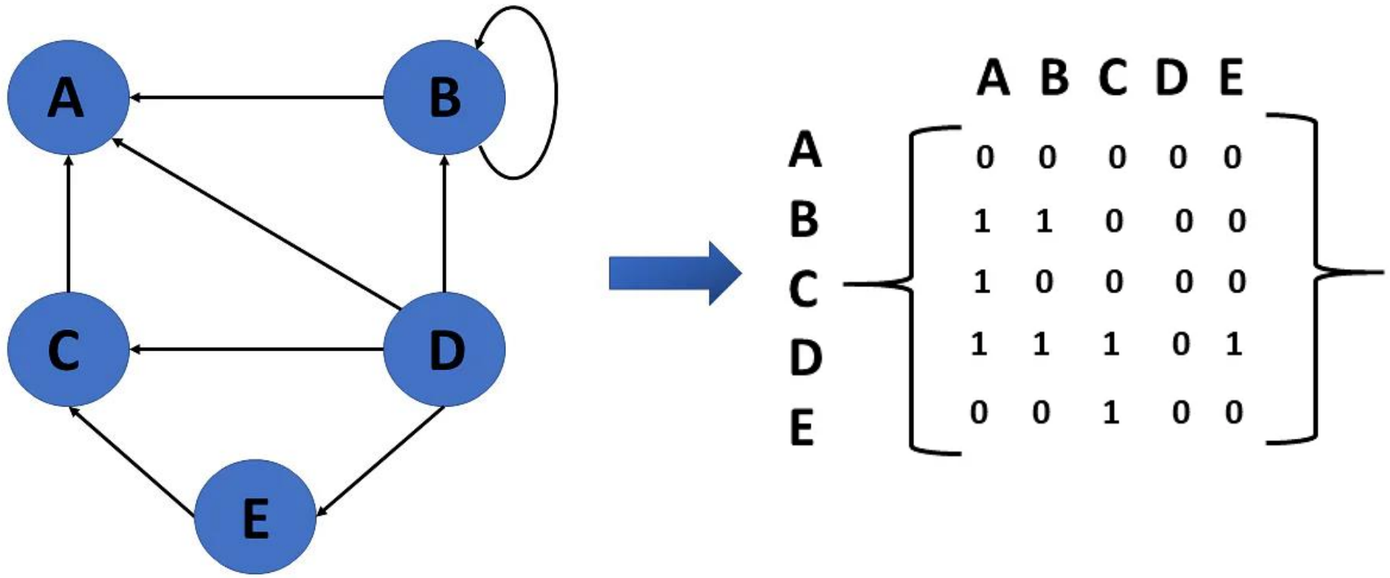
The image given below shows an undirected graph representation:



	A	B	C	D	E
A	0	1	1	1	0
B	1	1	0	1	0
C	1	0	0	1	1
D	1	1	1	0	1
E	0	0	1	1	0

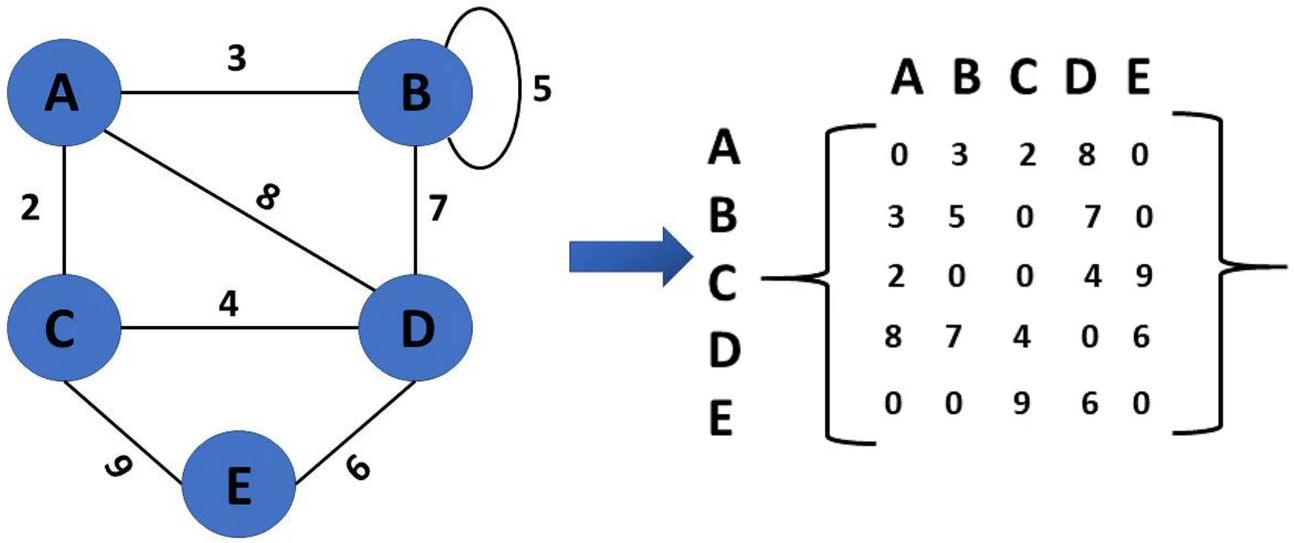
Graph Representation

The image given below shows a directed graph representation:



Graph Representation

The image given below shows a weighted undirected graph representation:



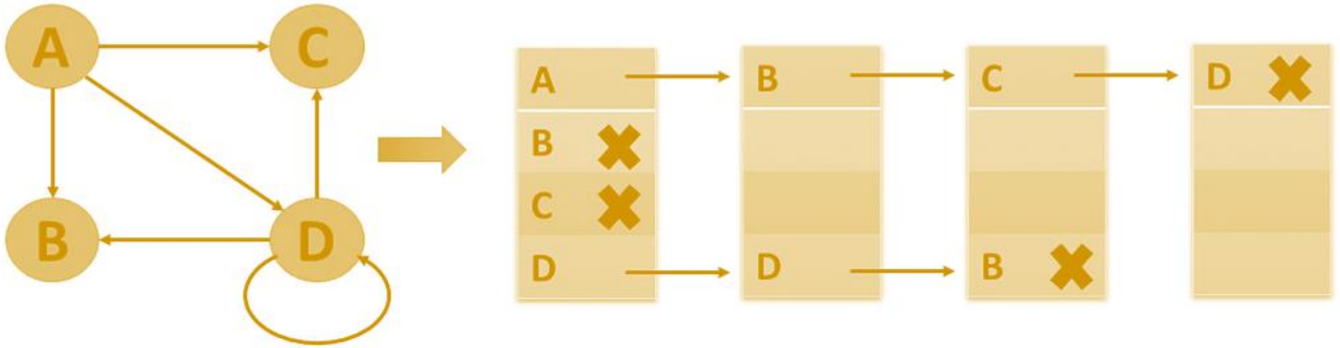
Graph Representation

Adjacency list

- Adjacency list is a linked representation.
- In this representation, we maintain the list of its neighbors for each vertex in the graph.
- Every vertex of the graph contains the list of its adjacency vertices.
- Array of vertices, each indexed by vertex number, linking to its neighbors' list.

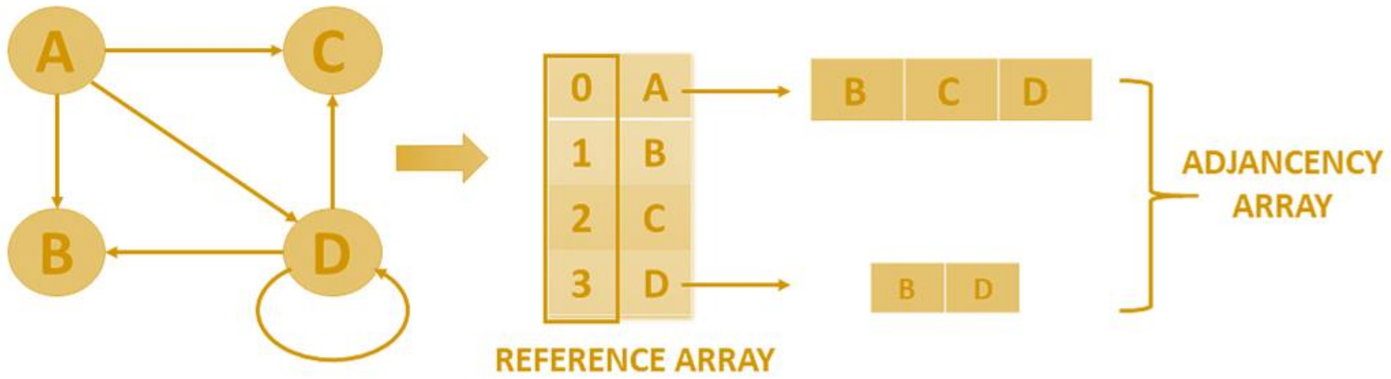
Graph Representation

The image given below shows a weighted undirected graph representation using a linked list:



Graph Representation

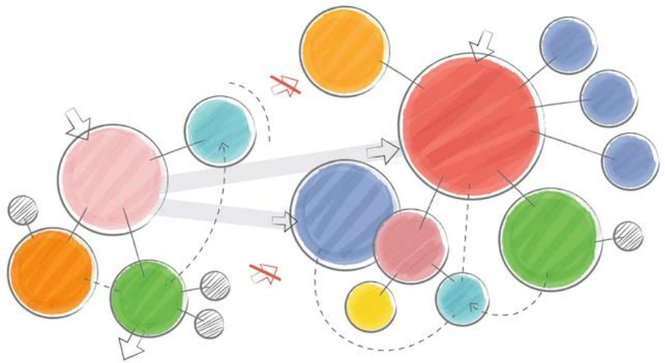
The image given below shows a weighted undirected graph representation using an array:



Graph Traversal

Graph Traversal

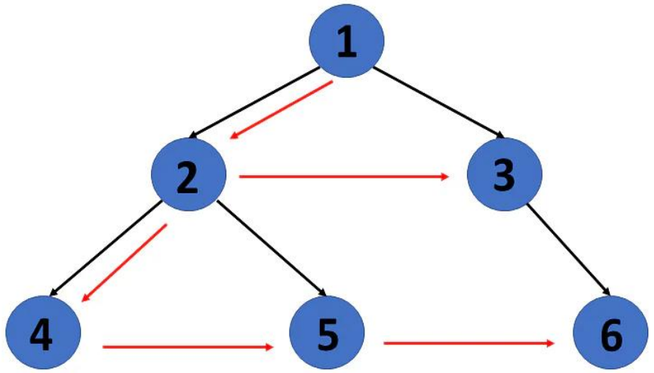
Graph traversal refers to the process of examining each edge and vertex once in a graph.



Graph traversing can be performed in two ways:

- Breadth-First Search (BFS) algorithm
- Depth-First Search (DFS) algorithm

Graph Traversal



BFS

1	2	3	4	5	6

Breadth-First Search (BFS) Algorithm

- BFS begins the traversal of the graph from the root node and explores all adjacent nodes.
- It then selects the closest node and explores its neighboring nodes.
- The BFS algorithm employs a queue data structure for this purpose.

Graph Traversal

Algorithm of Breadth-First Search (BFS)

Consider the graph you want to navigate

Select any vertex from graph, say v1

Examine any two data structure for traversing

Graph Traversal

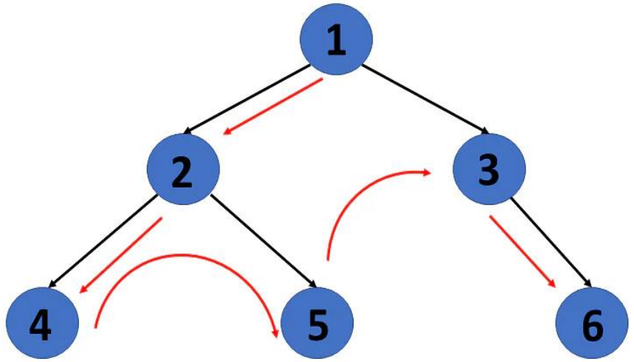
Algorithm of Breadth-First Search (BFS)

Starting from the vertex, add v1’s to the queue data structure

Now, using the FIFO concept, remove a element from the queue

Repeat the previous step until the queue is not empty

Graph Traversal



DFS

1	2	4	5	3	6

Depth-First Search (DFS) Algorithm

- DFS commences its traversal of the graph from the initial node and delves deeper and deeper until it encounters a node with no children.
- It then backtracks from the dead end towards the most recently unexplored nodes.
- The DFS algorithm utilizes a stack data structure for this purpose.

Graph Traversal

Algorithm of Depth-First Search (DFS)

Consider the graph you want to navigate

Select any vertex from the graph, say v1

Examine any two data structures for traversing the graph

Insert v1 into the array’s first block

Graph Traversal

Algorithm of Depth-First Search (DFS)

Now, using the FIFO principle pop the topmost element

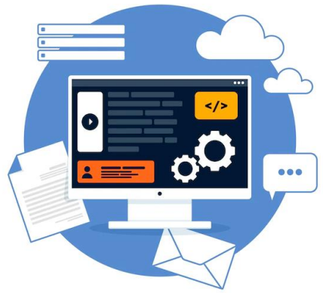
If the topmost element of the stack is already present, discard and insert

Repeat the previous step until the stack data structure isn’t empty

Applications of Graphs

Applications of Graphs

Following are some applications of graphs in data structure:



Computer Science

Facebook

Database

World Wide Web

Assisted Practice

Creating and Representing Graph **Duration: 20 Min.**

Problem Statement:

You have been assigned a task to demonstrate the creation and representation of a graph using JavaScript.

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

Assisted Practice

Traversing a Graph **Duration: 20 Min.**

Problem Statement:

You have been assigned to demonstrate the graph traversal using JavaScript

Assisted Practice: Guidelines

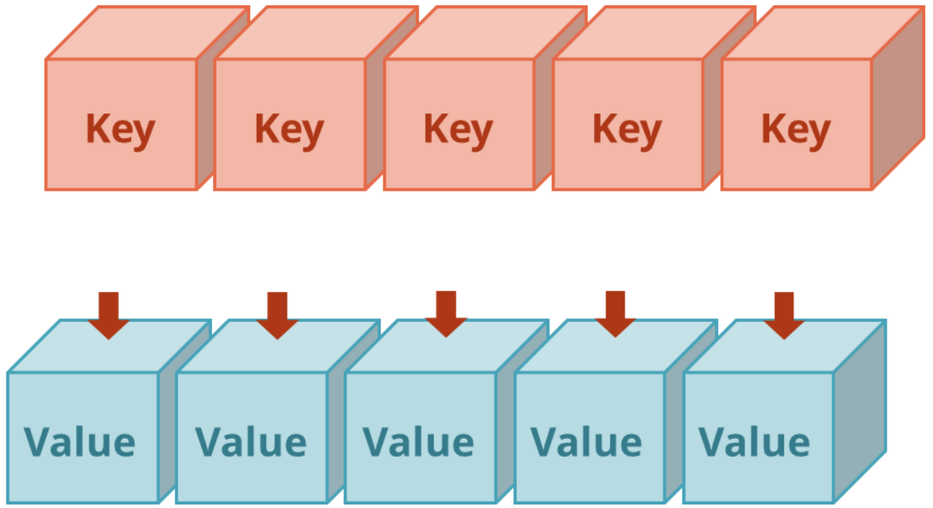
Steps to be followed:

1. Create and execute the JS file

HashMap

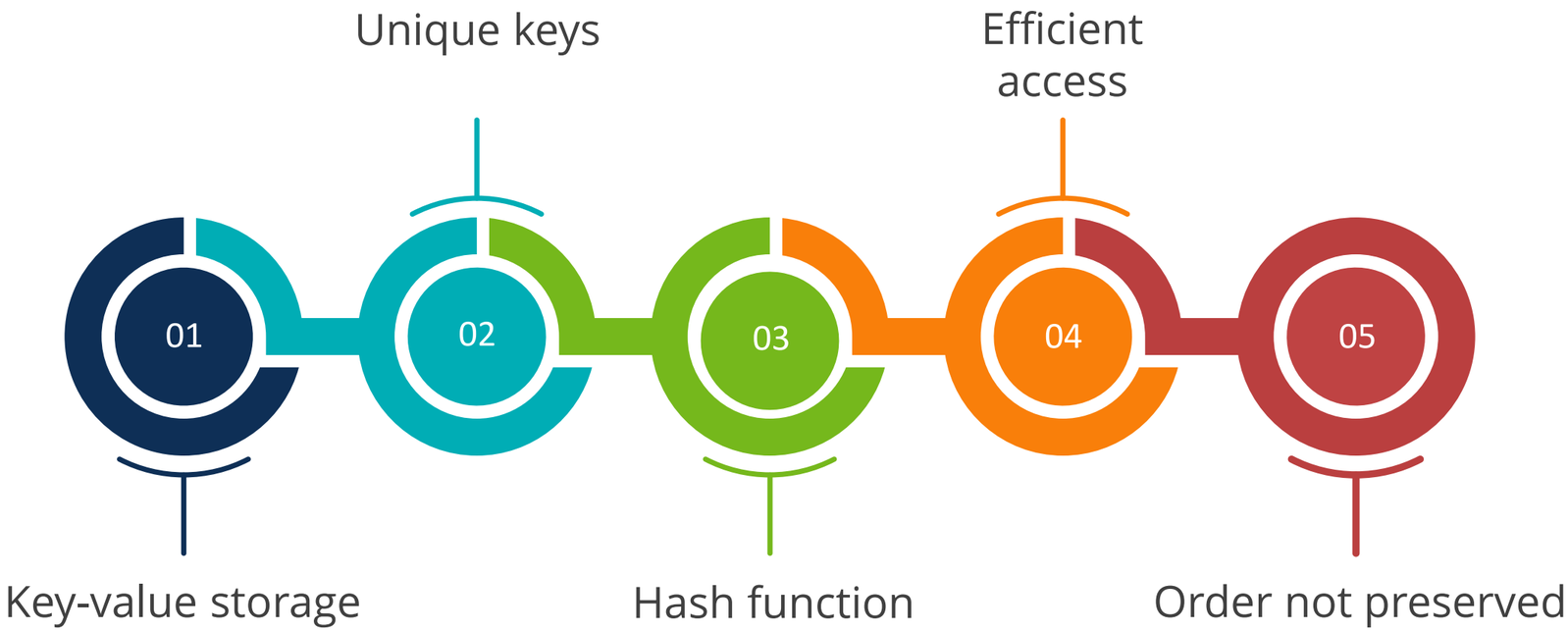
What Is a HashMap?

It is a data structure that stores key-value pairs, allowing for efficient retrieval of values based on their corresponding keys.



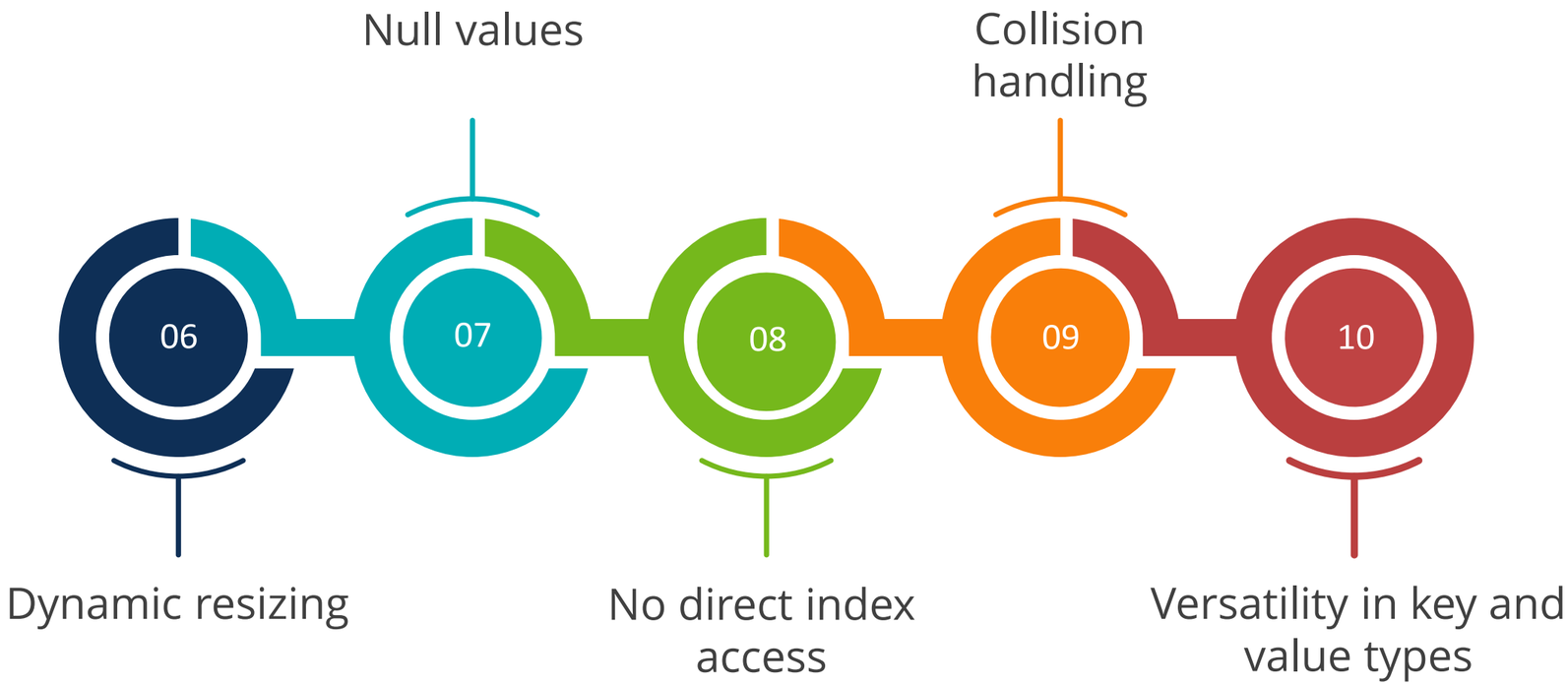
Characteristics of a HashMap

A HashMap is distinguished by several key characteristics:



Characteristics of a HashMap

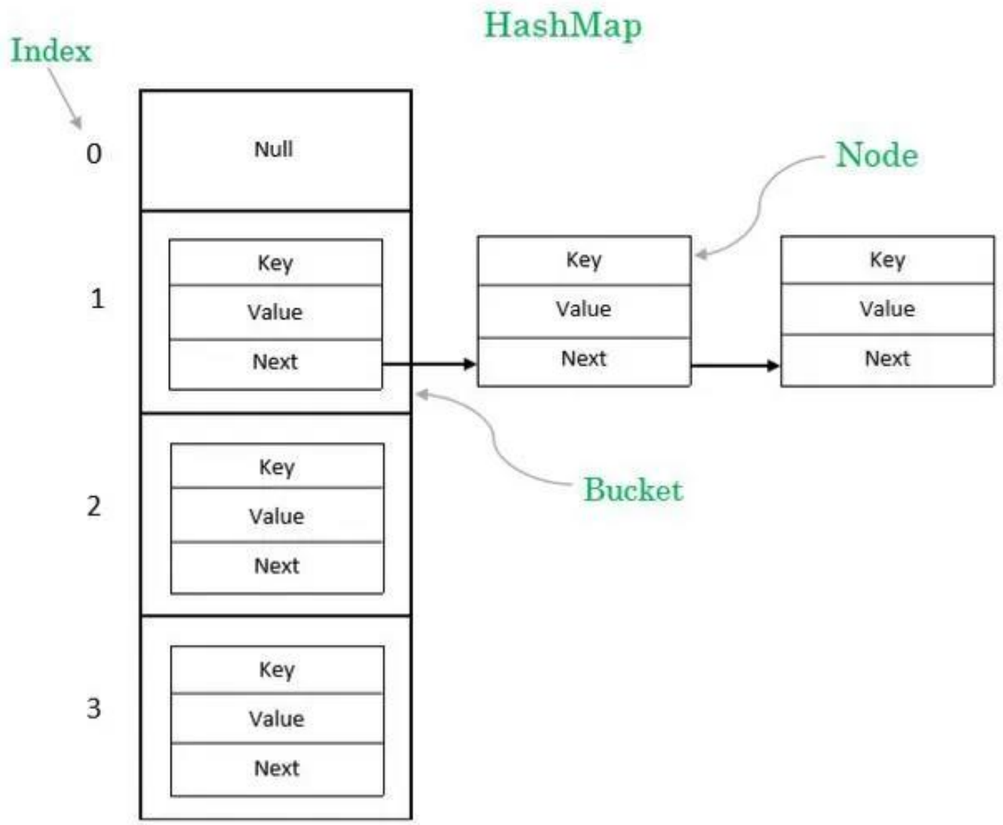
A HashMap is distinguished by several key characteristics:



How HashMap Works?

How HashMap Works?

It works by storing data in a format that allows for quick retrieval, insertion, and deletion of key-value pairs.

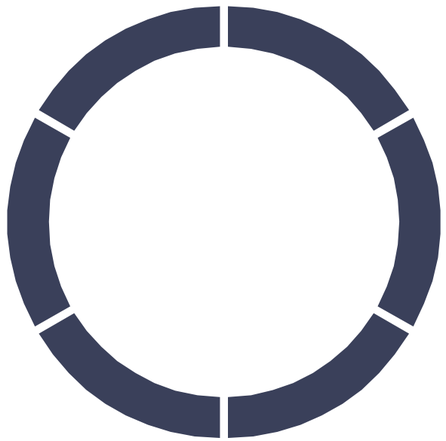


Implementing HashMaps Using Arrays and LinkedList

It involves creating a data structure that maps keys to values, with a unique value for each key. Here's a simplified explanation without using code:

1. Underlying array

2. Index calculation

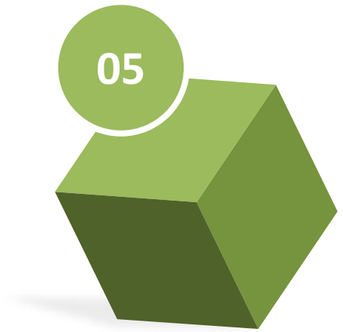


3. Handling collisions with linked lists

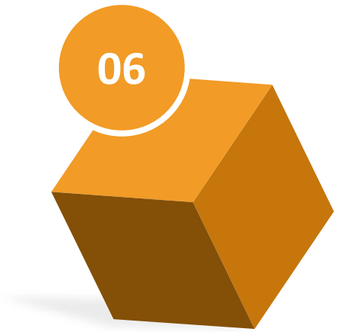
4. Adding a key-value pair (Put operation)

Implementing HashMaps Using Arrays and LinkedList

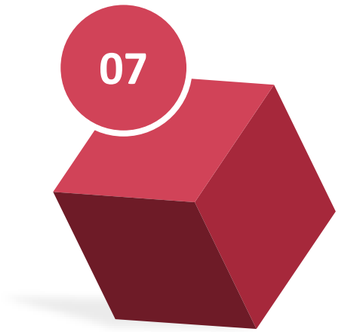
It involves creating a data structure that maps keys to values, with a unique value for each key. Here's a simplified explanation without using code:



Retrieving a value (Get operation)



Removing a key-value pair (Remove operation)

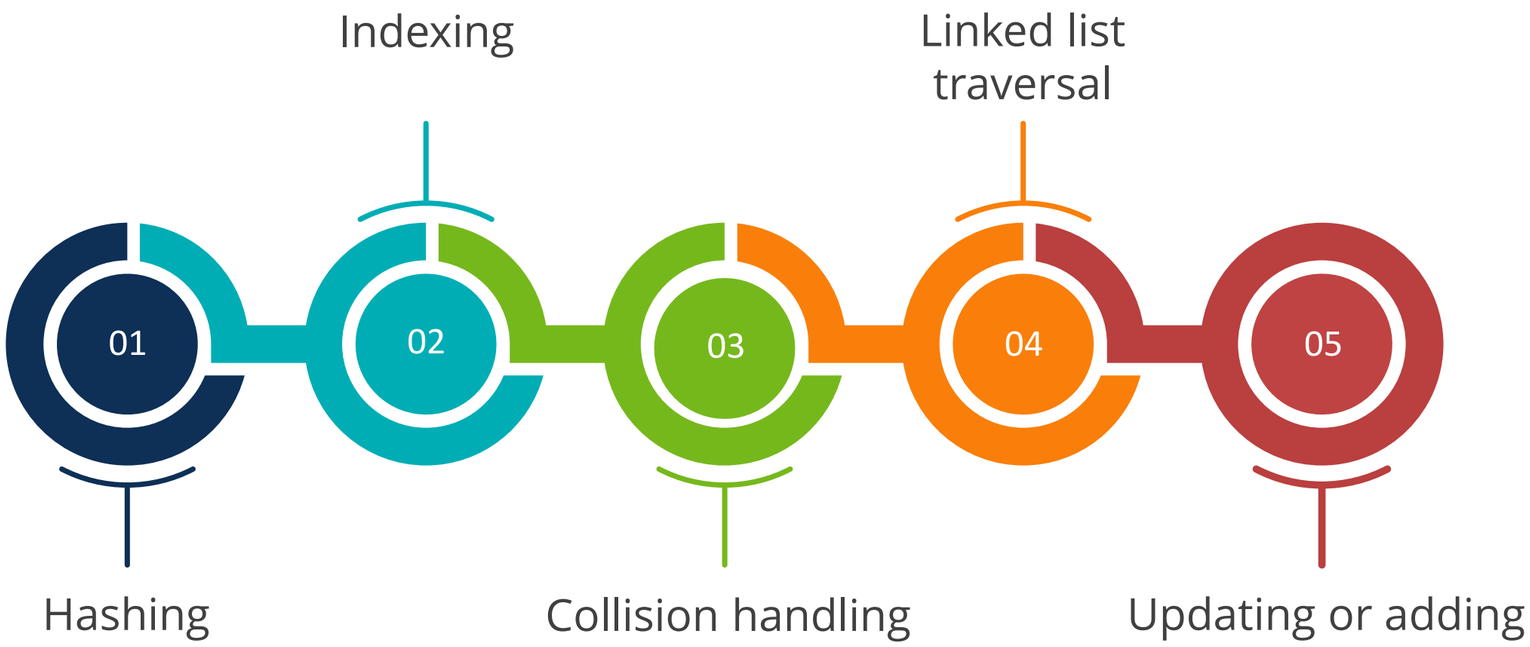


Resize and rehash

Adding and Retrieving Elements

Implementation using arrays and linked lists involves several steps for placing and locating key-value pairs.

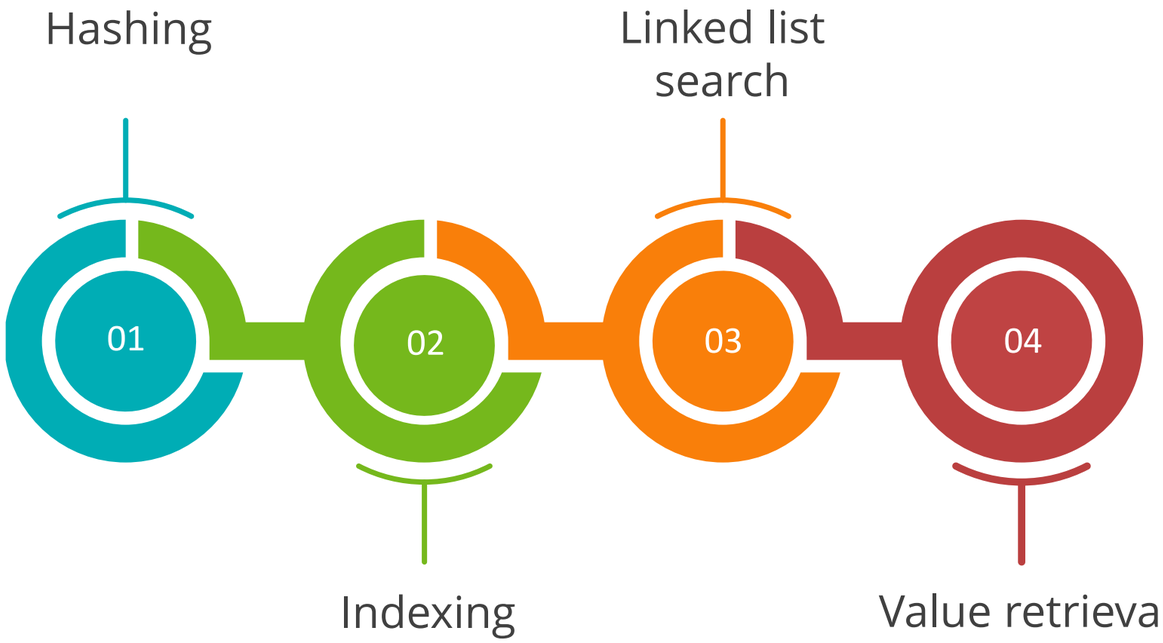
Here are the key aspects involved in adding elements (Put operation):



Adding and Retrieving Elements

Implementation using arrays and linked lists involves several steps for placing and locating key-value pairs.

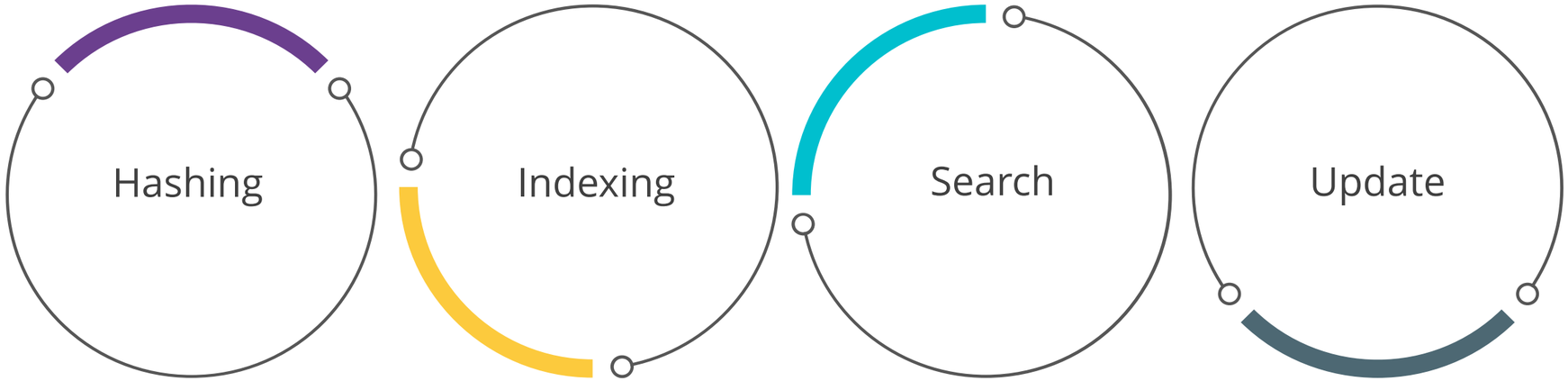
Here are the key aspects involved in retrieving elements (Get operation):



Updating and Deleting Elements

A HashMap utilizes arrays and linked lists to identify a key for updating its value or deleting the key-value pair.

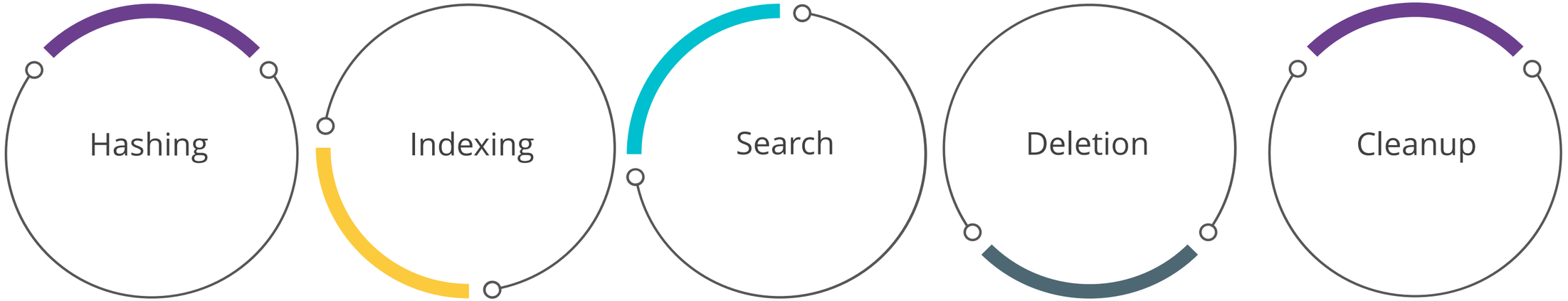
To update an element in a HashMap, the process involves changing the value associated with a given key. The following methods are used:



Updating and Deleting Elements

A HashMap utilizes arrays and linked lists to identify a key for updating its value or deleting the key-value pair.

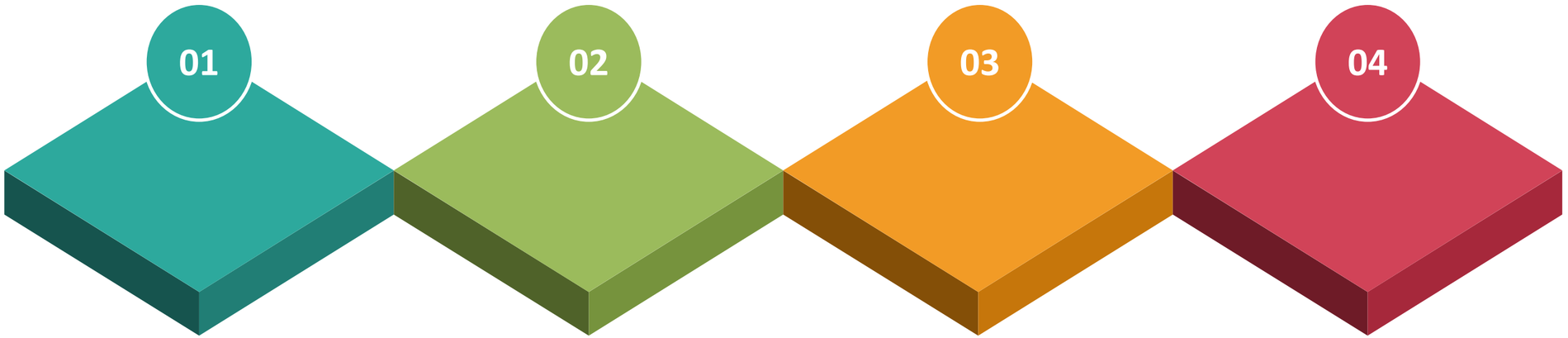
Deleting an element involves removing the key-value pair from the HashMap.
The process is as follows:



Iterating over HashMaps

Traversal of hashmaps, implemented with arrays and linked lists, requires iterating over each array element and its corresponding linked list.

Here's a general description of the process:



Traverse the array

Traverse each linked list

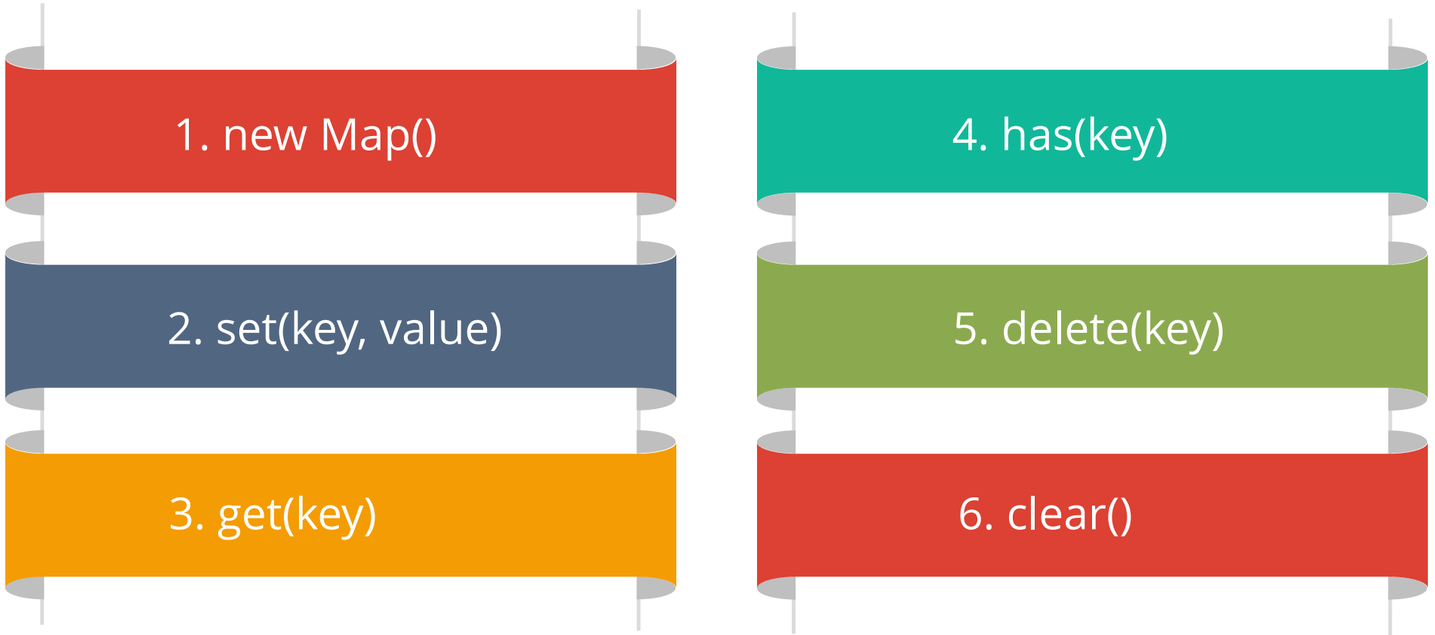
Access key-value pairs

Move to the next index

HashMaps Operations and Methods

Languages like Java and JavaScript offer diverse operations and methods for efficient data handling in hashmaps.

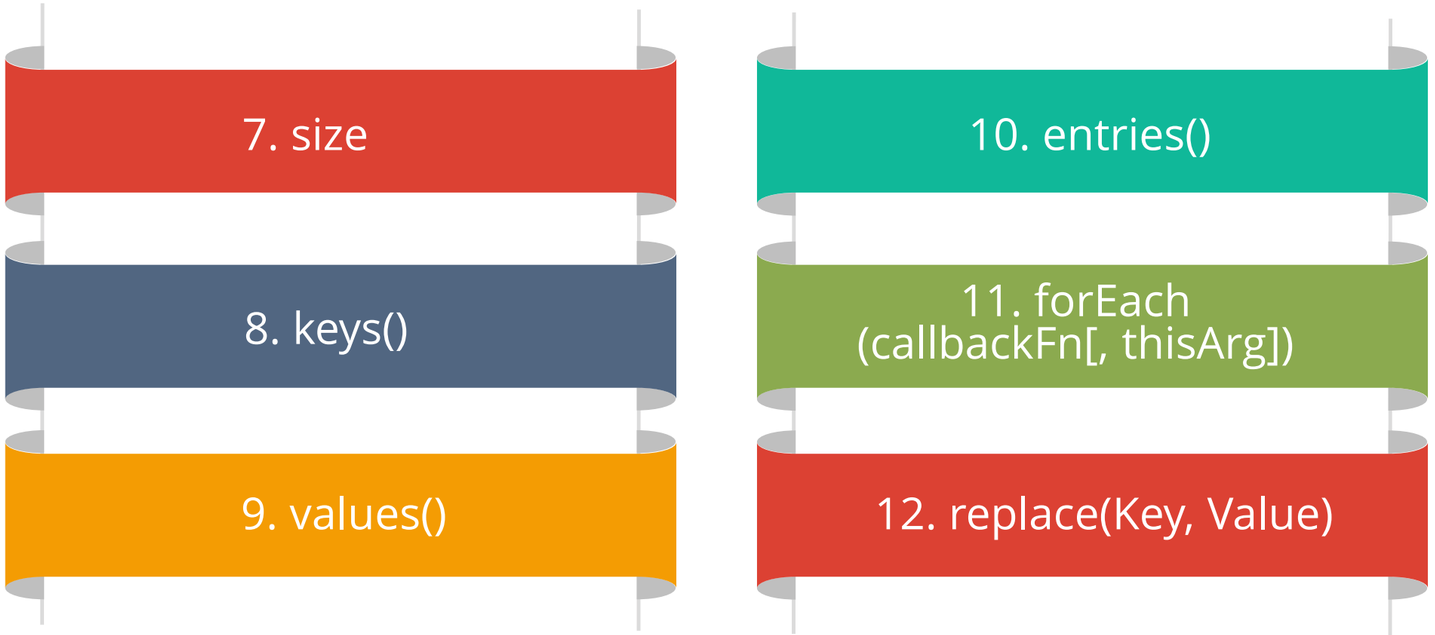
Here are some common operations and methods associated with HashMaps:



HashMaps Operations and Methods

Languages like Java and JavaScript offer diverse operations and methods for efficient data handling in hashmaps.

Here are some common operations and methods associated with HashMaps:



Assisted Practice

Declaring and Initializing HashMap **Duration: 15 Min.**

Problem Statement:

You have been assigned a task to demonstrate the process of creating, initializing, and utilizing a HashMap in JavaScript to effectively store and retrieve data.

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

Assisted Practice

Working with HashMap **Duration: 15 Min.**

Problem Statement:

You have been assigned a task to demonstrate the usage of a HashMap in JavaScript, covering the creation of a HashMap, addition and deletion of key-value pairs, and the methods to clear and display its contents.

Assisted Practice: Guidelines

Steps to be followed:

1. Create and execute the JS file

Key Takeaways

- The tree is a non-linear data structure that consists of nodes and is connected by edges.
- Traversing tree data structure helps to visit the required node in the tree to perform a specific operation.
- The graph is a non-linear data structure that consists of finite sets of vertices and a bunch of edges connecting with them.
- Adjacency matrix and Adjacency list are two ways by which graph is represented.
- HashMaps, typically using arrays and linked lists, efficiently handle data operations like addition, retrieval, update, and deletion, while managing collisions.

Thank You