Data Structures and Algorithms

Non-Linear Data Structures

Engage and Think

Imagine you are working with Uber Eats and optimizing a food delivery app. The app must efficiently categorize restaurants by cuisine and ratings, retrieve search results instantly, and determine the fastest delivery routes. To manage this data effectively, the system needs the right structure, whether through hierarchical organization or key-value mappings.

What logic would you apply to organize restaurants into categories, plan the best delivery routes, and quickly retrieve user preferences?

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
- Apply and examine different tree traversal methods for efficient data handling
- Create and distinguish between different graph types to apply suitable algorithms for specific problems
- Analyze how HashMaps utilize hashing techniques to enable fast data retrieval and storage.

Introduction to Non-Linear Data Structures

Non-Linear Data Structures

Data does not always follow a straight path. Non-linear data structures are used to organize information efficiently in complex relationships. The following are its key benefits:

Enhance searching and retrieval speed

Optimize storage and minimize redundancy

Represent real-world structures

Support dynamic data organization

Types of Non-Linear Data Structures

The following are the key types of non-linear data structures, each designed for specific data organization and problem-solving needs:

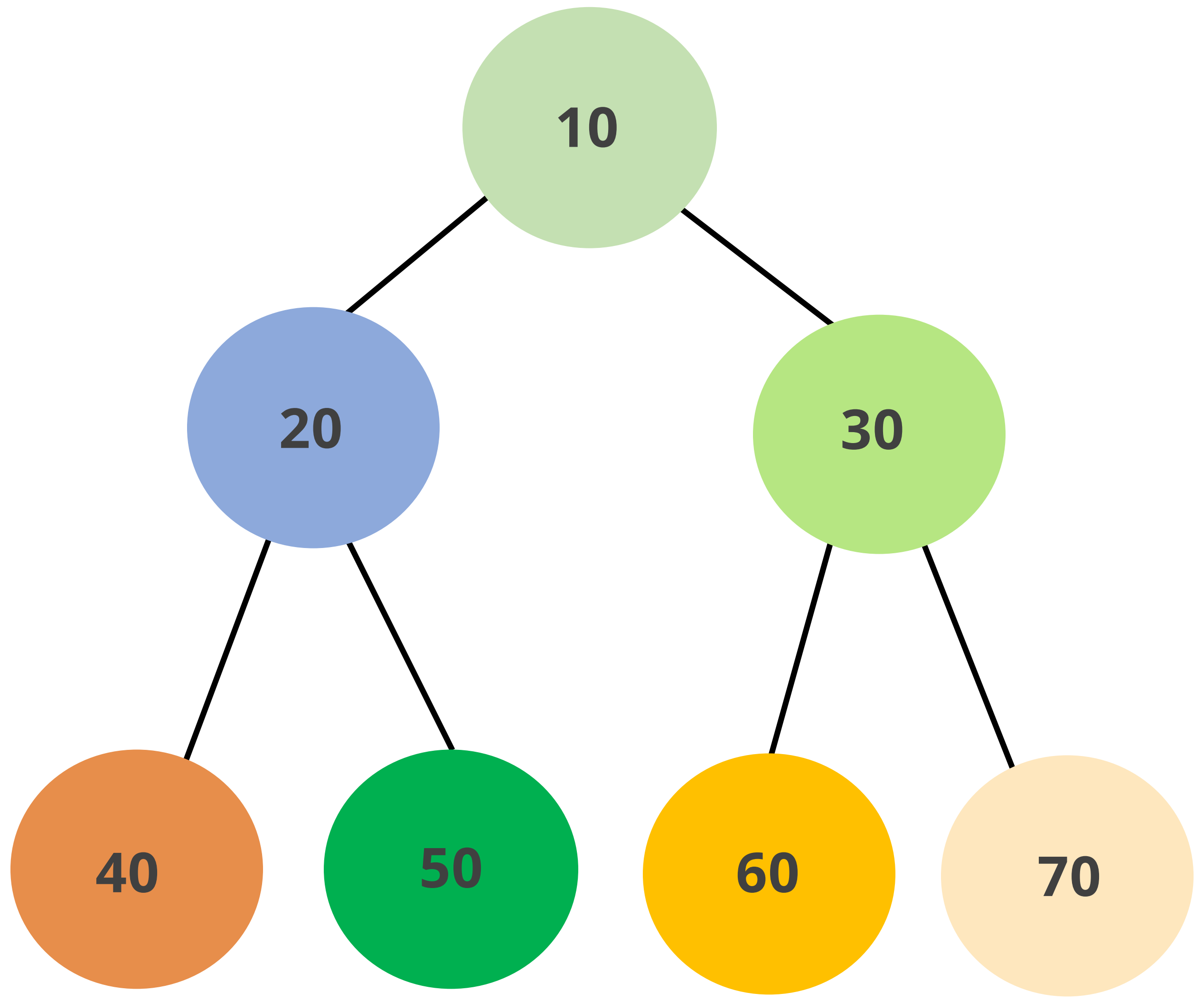
1. Trees
2. Graphs
3. HashMap

Choosing the right data structure enhances efficiency in solving computational and real-world challenges.

Introduction to Trees

Tree Data Structure

It is a hierarchical structure used to represent and organize data in the form of a parent-child relationship.



Tree Data Structure: Features

- **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.

Key Terminologies in Tree Data Structures

The key terminologies essential for analyzing and implementing tree-based algorithms efficiently are:

01 Nodes

02 Degree

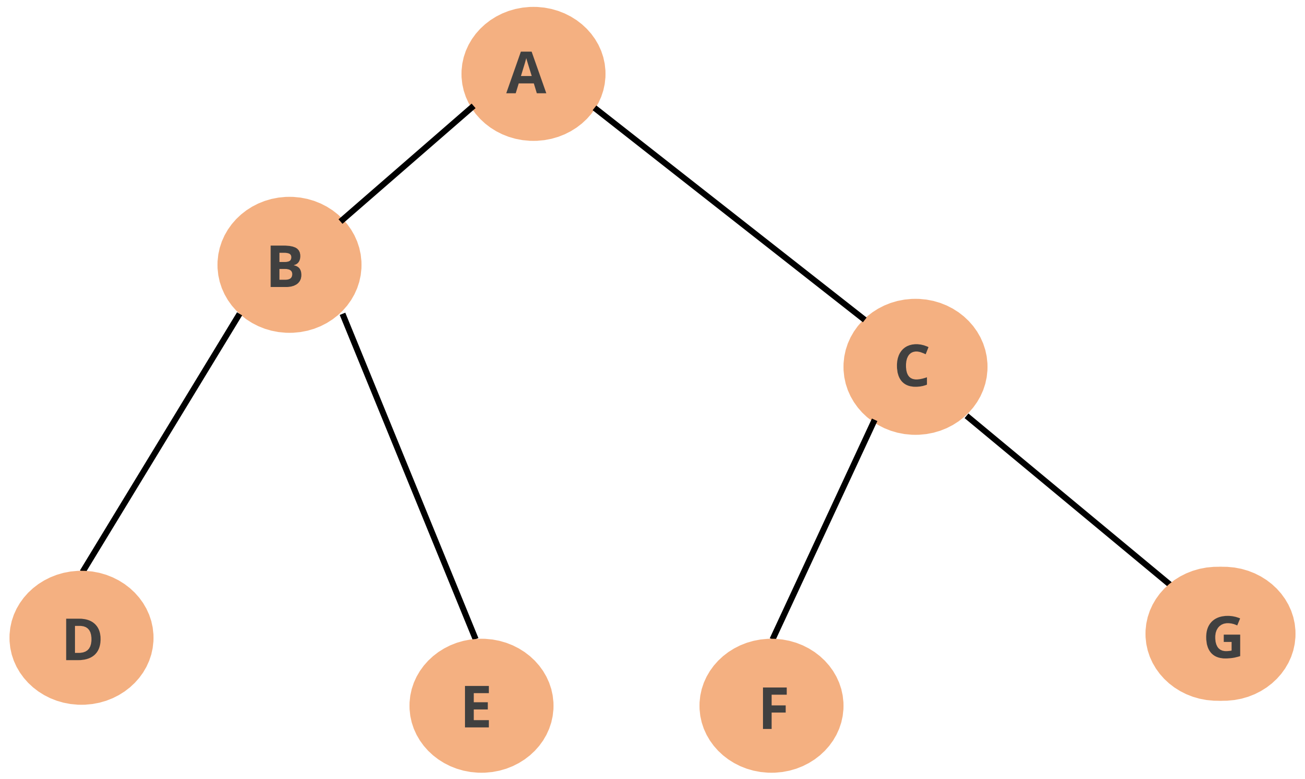
03 Levels

04 Height of a node

05 Depth of a node

Nodes in a Tree Data Structure

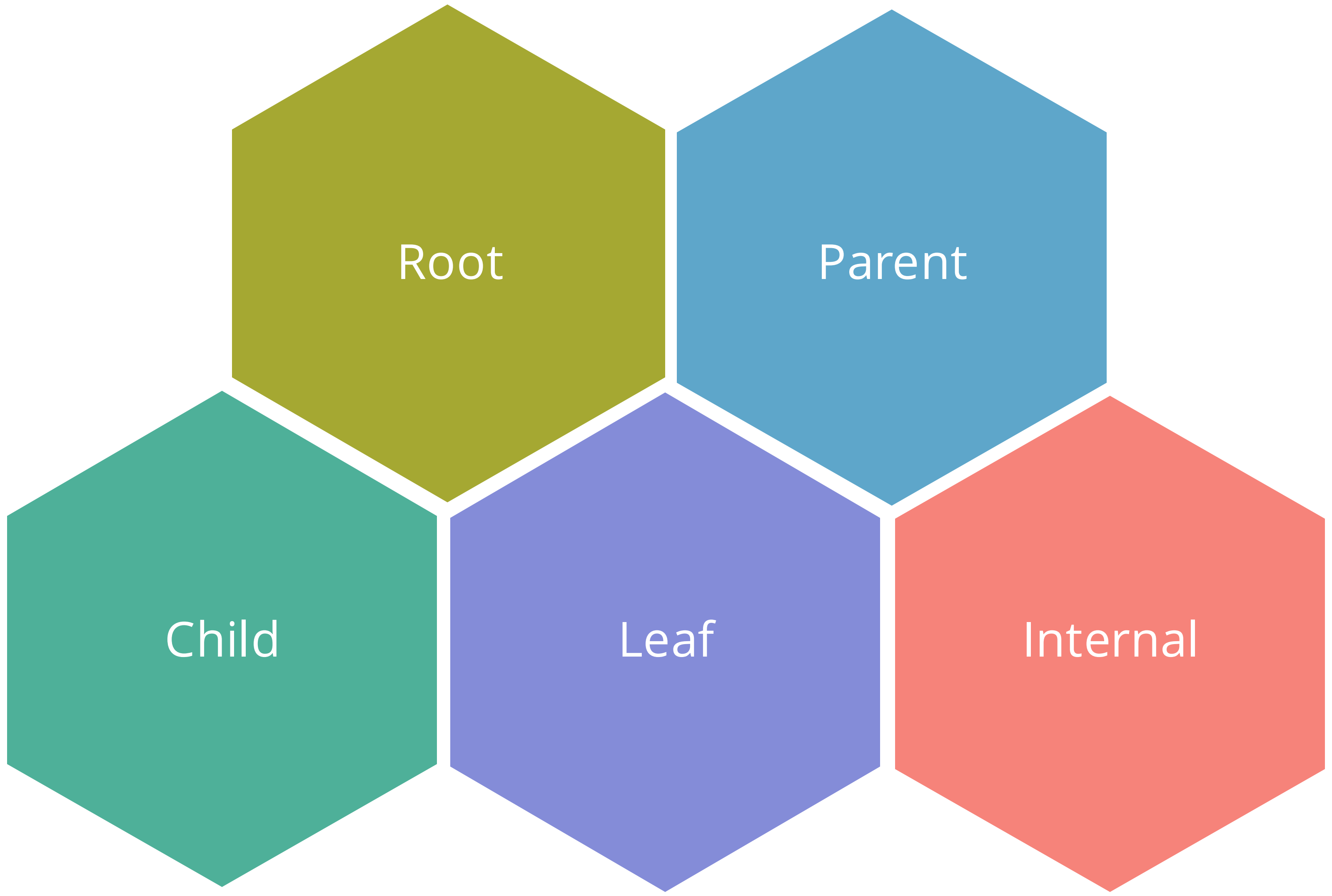
It is a fundamental unit in a tree that stores data and connects to other nodes through edges.



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

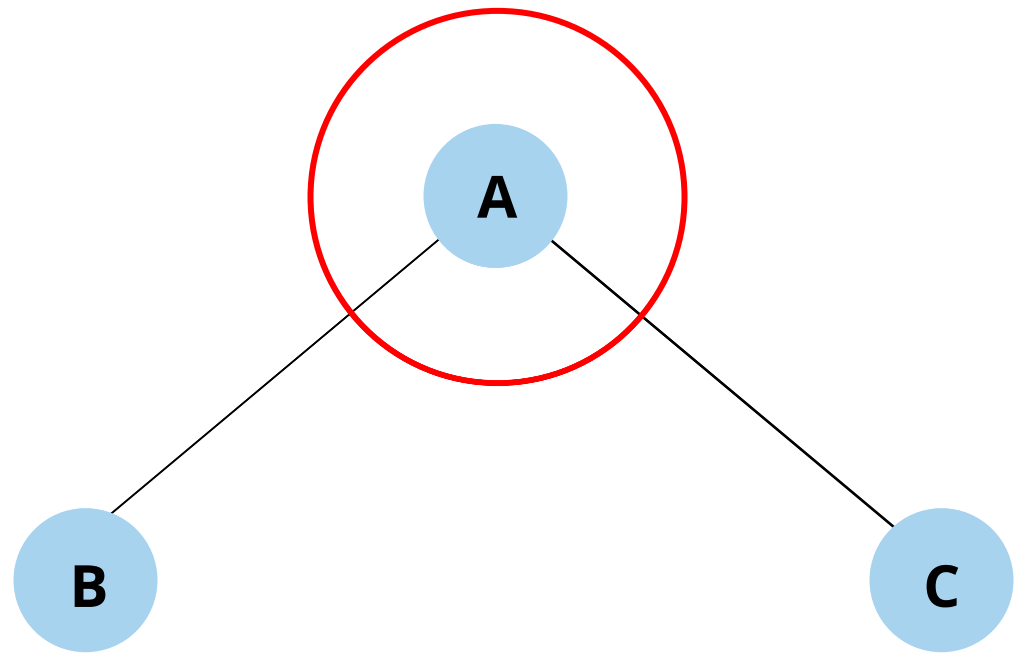
Types of Nodes

The following are the different types of nodes in a tree, categorized based on their position and relationships within the structure:



Root Node

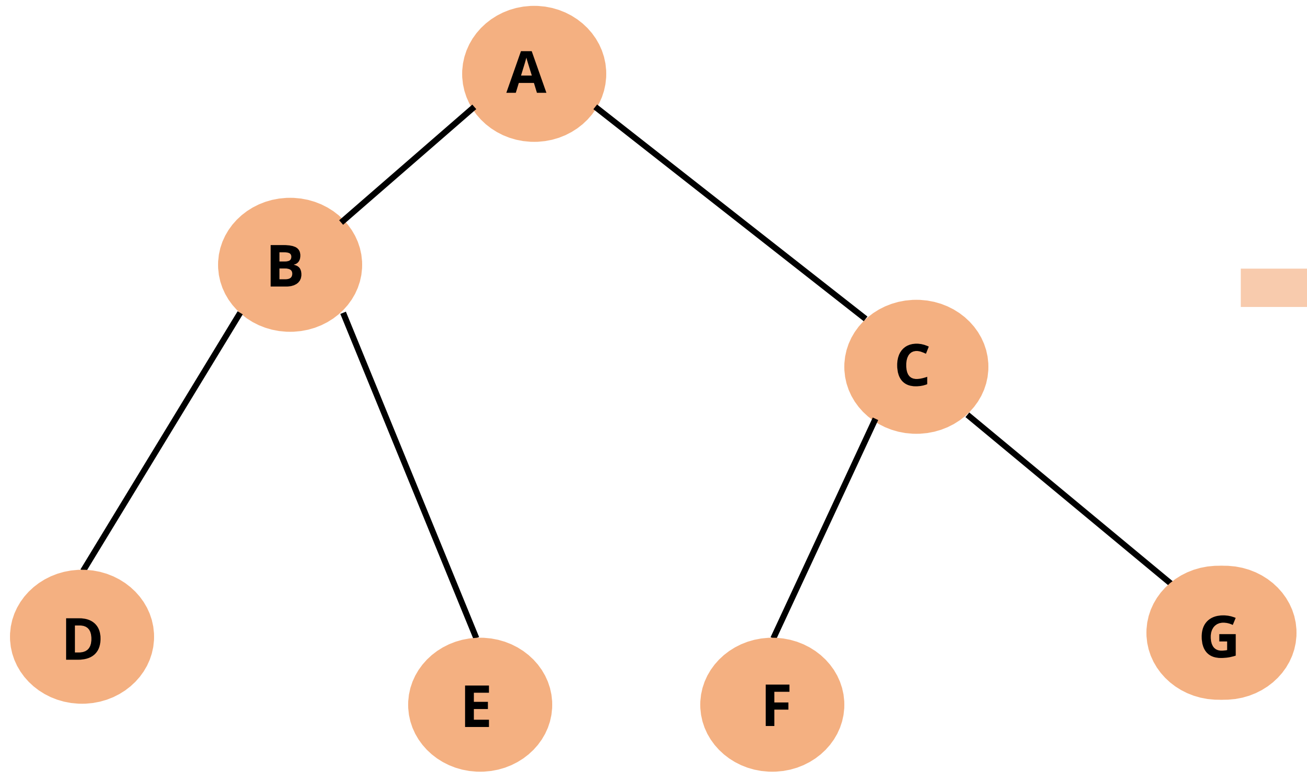
It is the topmost node in a tree data structure, serving as the starting point from which all other nodes branch out.



In the tree above, node A is the root node as it is the first and topmost node from which all other nodes originate.

Parent Node

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

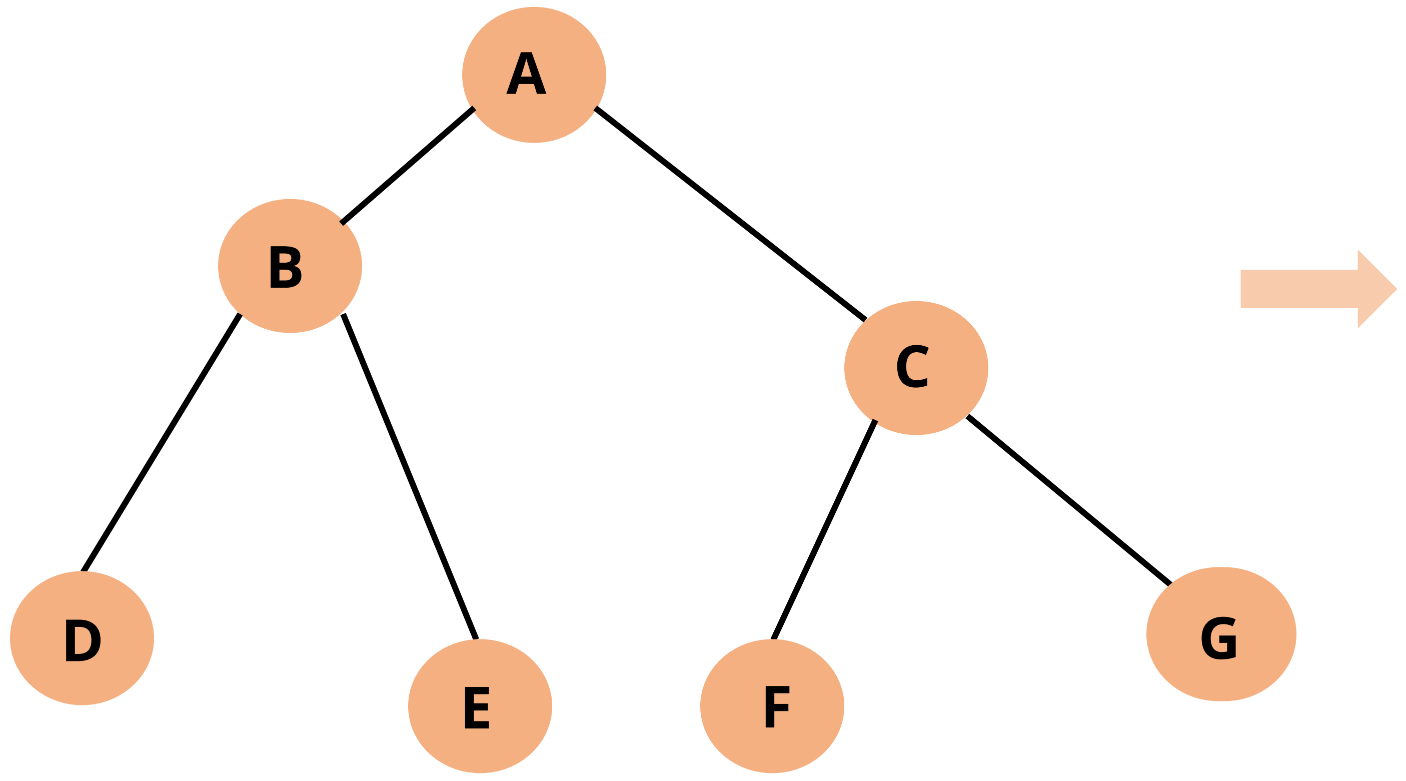


For example:

- 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.

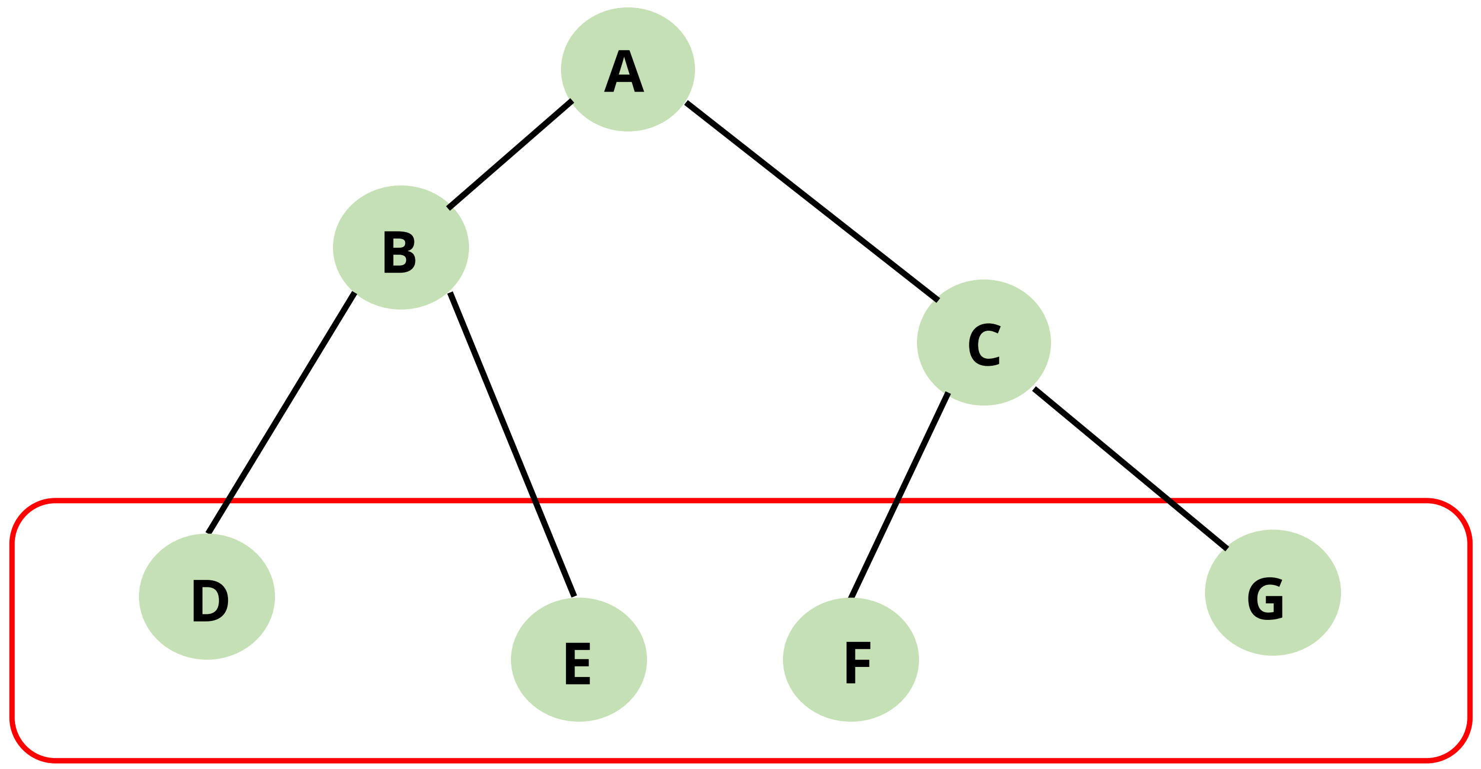


For example:

- 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

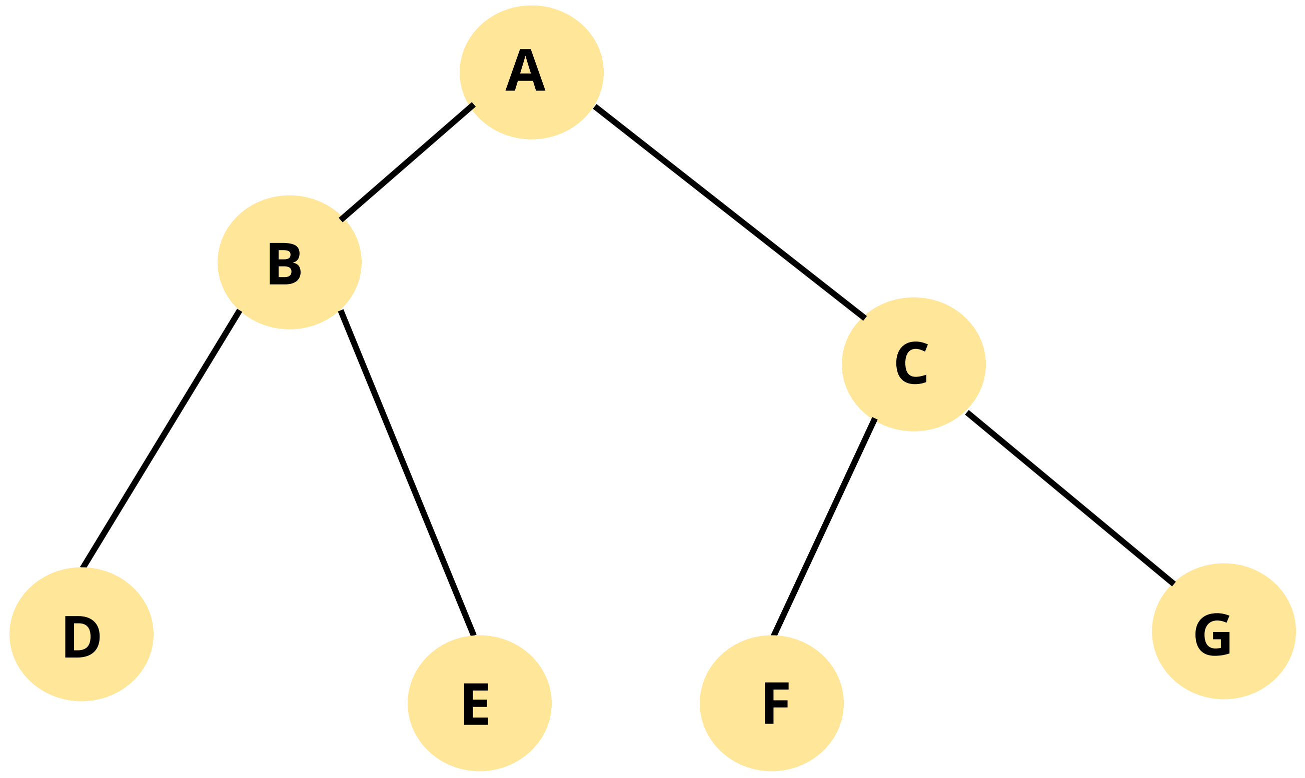
These are the nodes in a tree that do not have any further connections or branches.



In the tree above, D, E, F, and G are leaf nodes as they do not have any nodes connected below them.

Internal Node

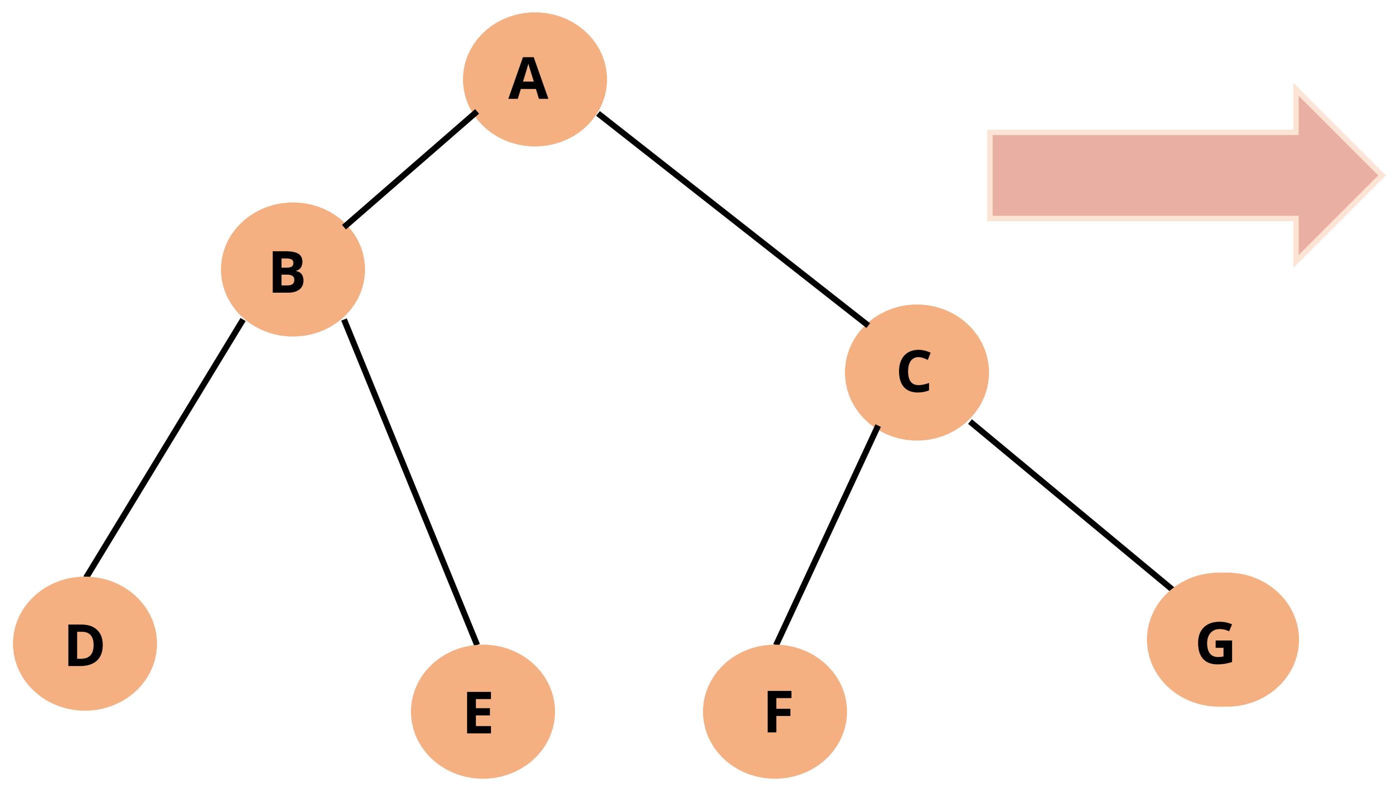
An internal node is any node with at least one child. All internal nodes are parents, but the root is not always considered an internal node.



In the tree above, A, B, and C are internal nodes as they have at least one connected node beneath them.

Degree in Tree Data Structure

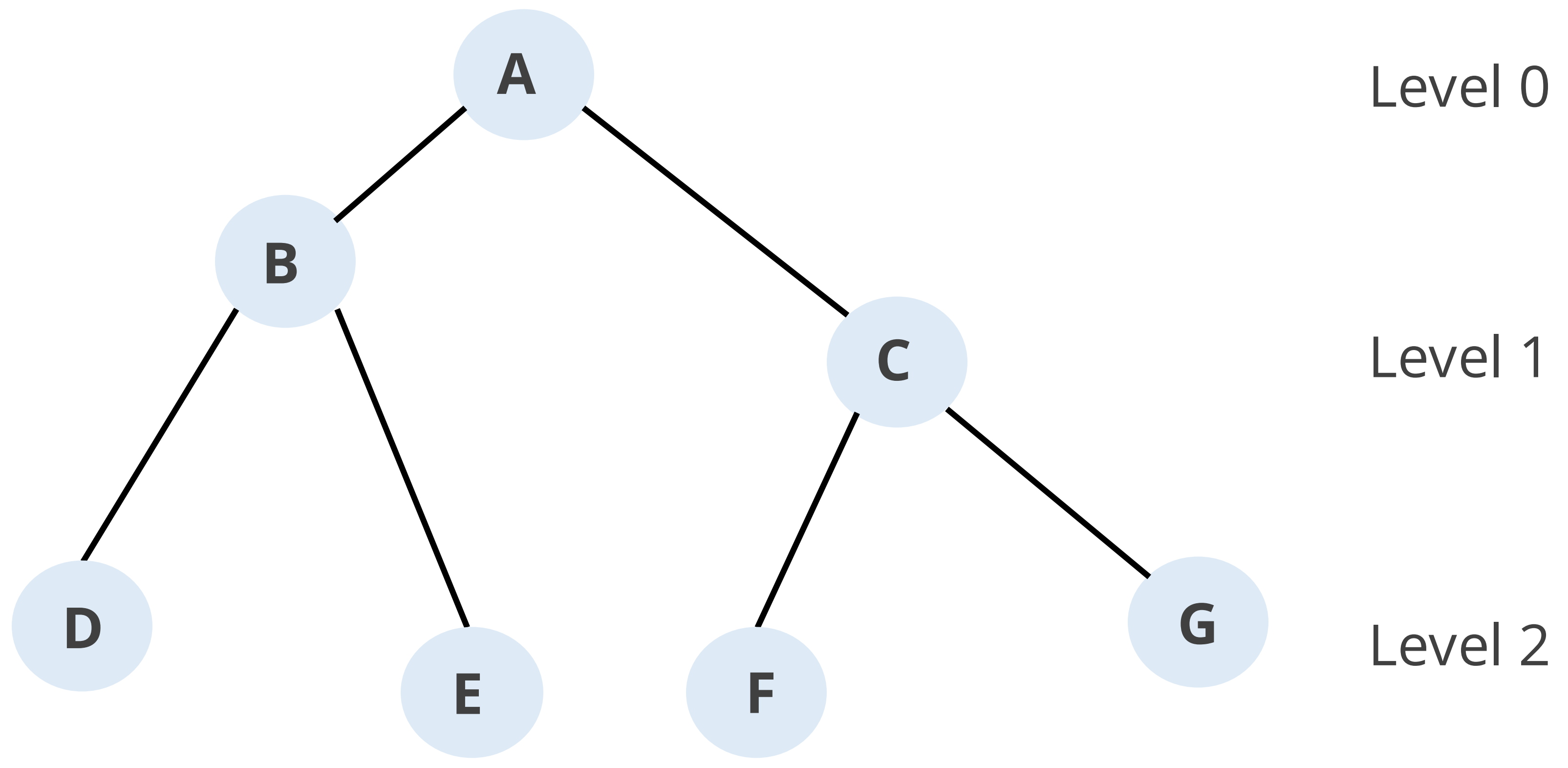
The degree of a node refers to the total number of child nodes it has. It helps determine the branching factor of a tree and varies for different nodes based on their position in the structure.



- **Node A** has two children (B and C), so its degree is **2**.
- **Node B** has two children (D and E), so its degree is **2**.
- **Node C** has two children (F and G), so its degree is **2**.
- **Nodes D, E, F,** and **G** have no children, so their degree is **0**.

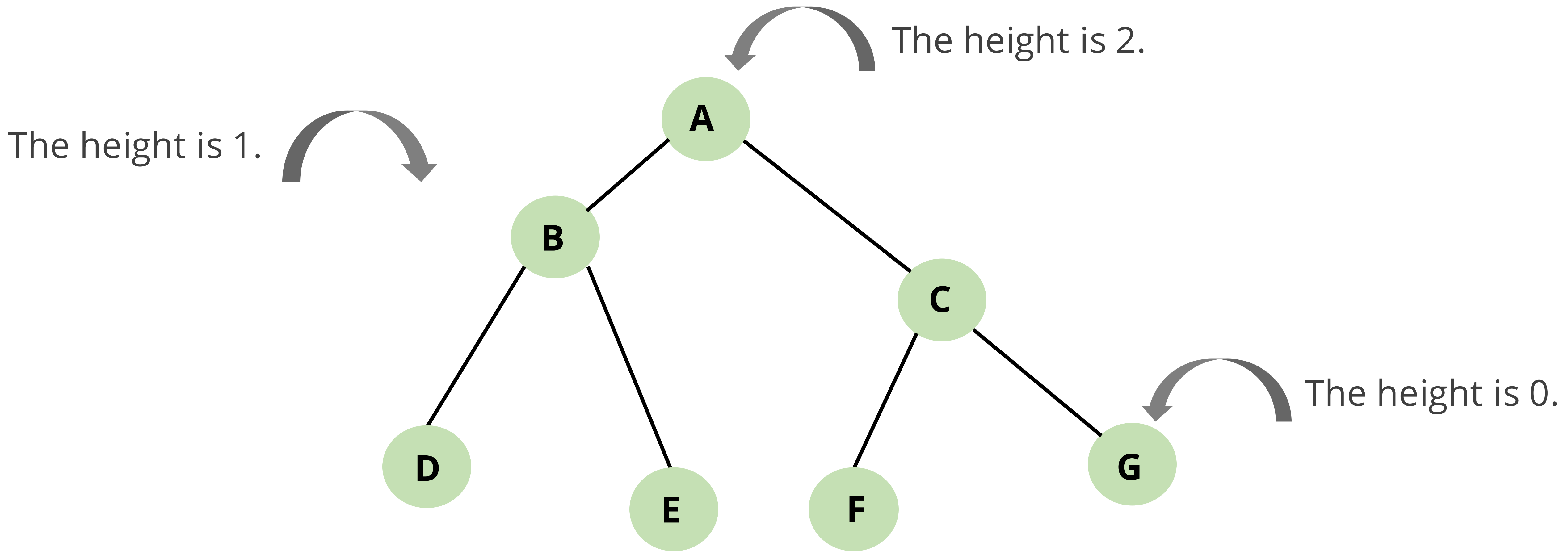
Levels in Tree Data Structure

In a tree structure, the level of a node represents its distance from the root node. The root node is always at level 0, and each subsequent level increases by 1 as it moves down the tree.



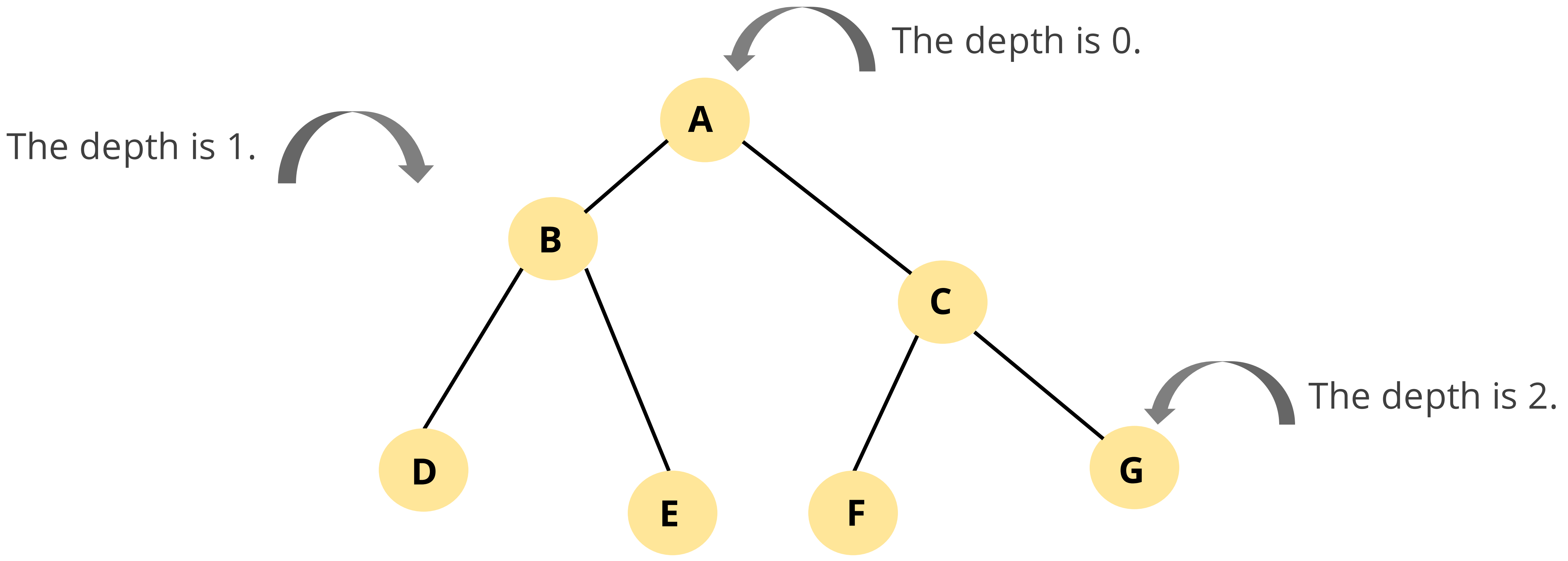
Height of a Node

The height of a node in a tree is the number of edges on the longest path from that node to a leaf node. It determines the depth of the subtree rooted at that node.



Depth of a Node

The depth of a node in a tree is the number of edges from the root node to that specific node. It represents how far a node is from the root.



Types of Tree Data Structures

Types of Trees

The following are the key types of trees in data structures, each designed for efficient data organization, searching, and storage:

01 Binary tree

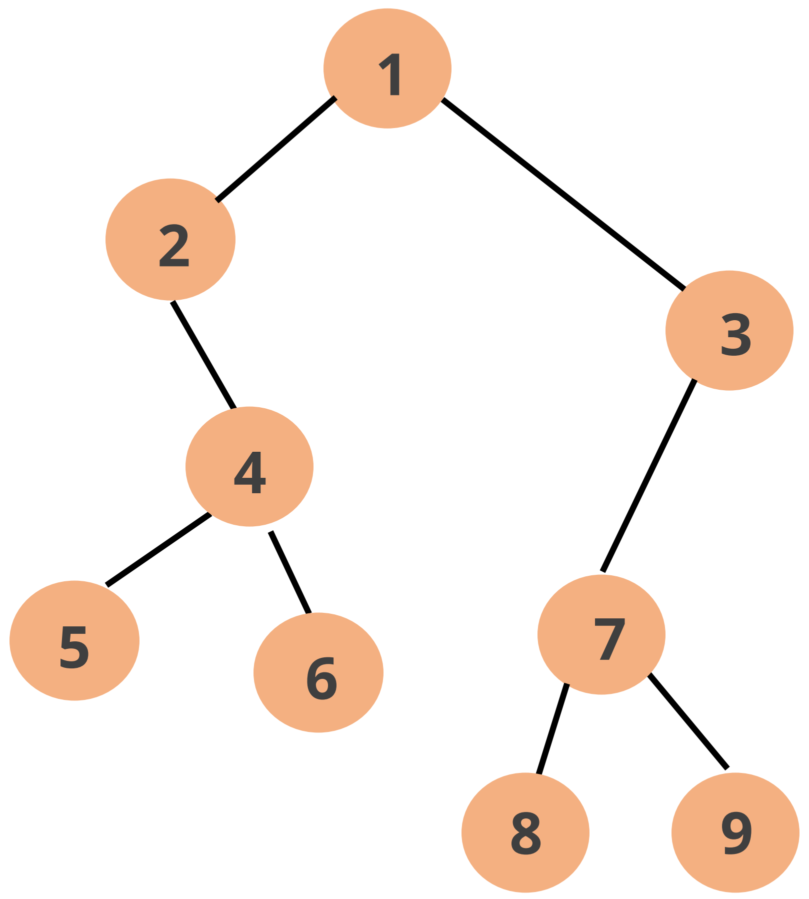
02 Binary search tree

03 AVL tree

04 B-tree

Binary Tree

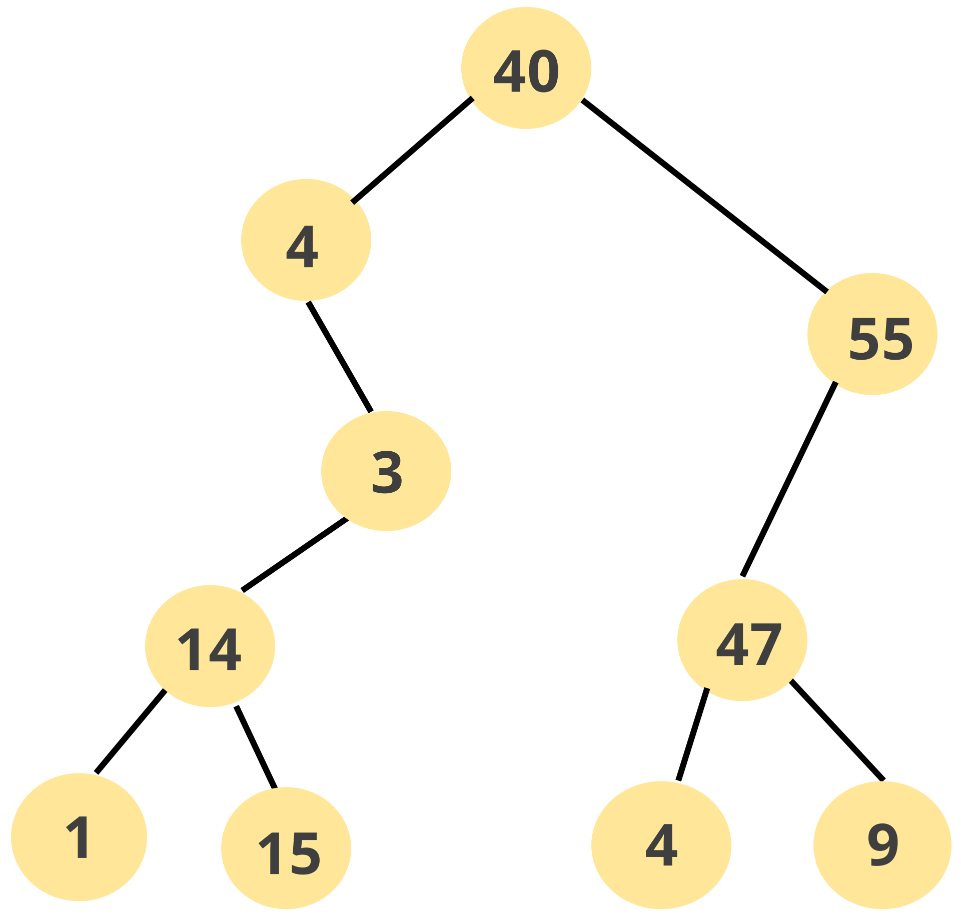
A binary tree is a tree data structure in which each parent node has at most two children.



They are used in hierarchical organization, such as family trees and decision-making processes.

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.



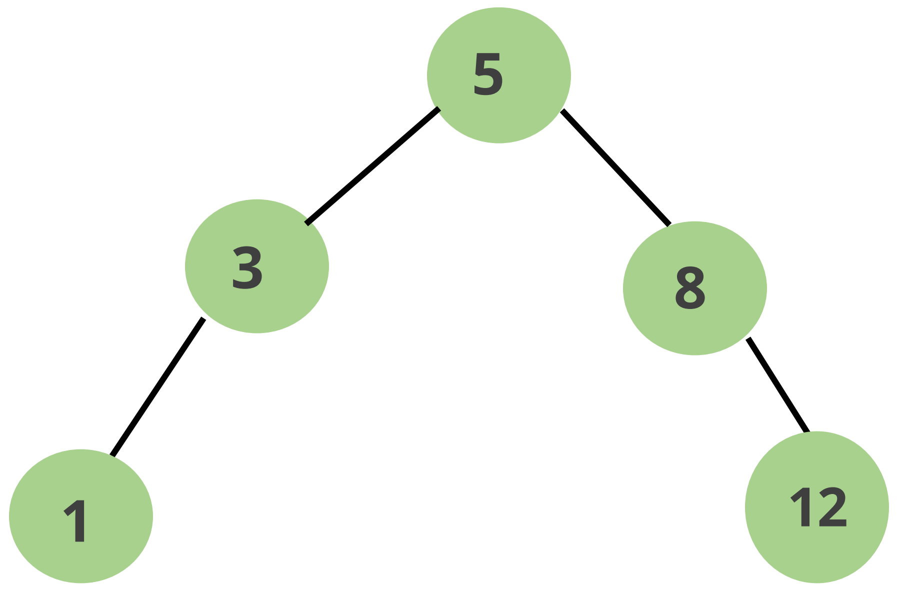
All nodes in the left subtree are less than the root node.

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

They are used in search engines and databases to store and retrieve sorted data efficiently.

AVL Tree

It 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 with time complexity of O(log n).



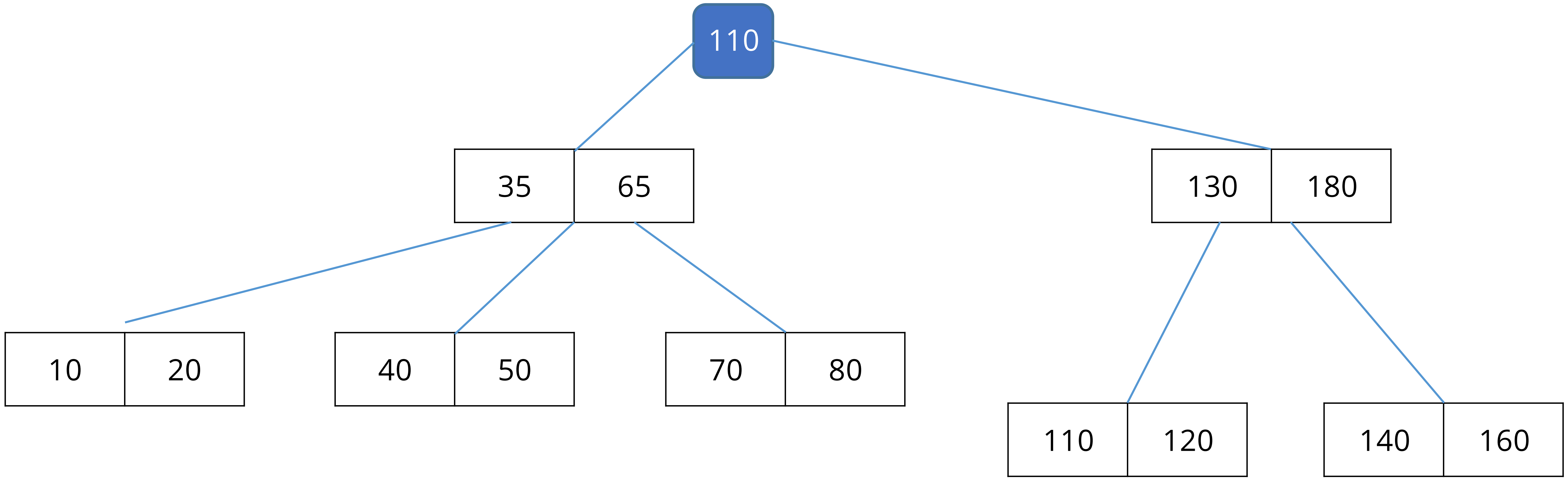
Balance factor = (Height of left subtree) - (Height of right subtree)

The tree remains balanced because the balance factor of each node is between -1 and 1, which is the condition for an AVL tree.

They are used in memory management and indexing, ensuring balanced and efficient data retrieval.

B-Tree

It is a self-balancing search tree in which each node can contain multiple values and have more than two children. It is also known as a height-balanced multi-way tree.



They are used in file systems and databases to manage large blocks of sorted data with fast access times.

Quick Check

A software engineer is building a real-time stock price tracking system that requires fast insertions and lookups. They consider using an AVL tree for efficient data storage.
Which of the following best explains why an AVL tree is a good choice?

A. AVL trees maintain balance, ensuring efficient insertions and lookups.

B. AVL trees do not allow duplicate values, preventing multiple stock prices for the same stock.

C. AVL trees store data in an unordered manner, making them ideal for unstructured data.

D. AVL trees do not balance themselves, which could slow down retrieval.

Tree Traversal

Tree Traversal

Traversing a tree data structure involves visiting nodes in a specific order to perform operations. The three types of tree traversal techniques are:

In-order traversal

Pre-order traversal

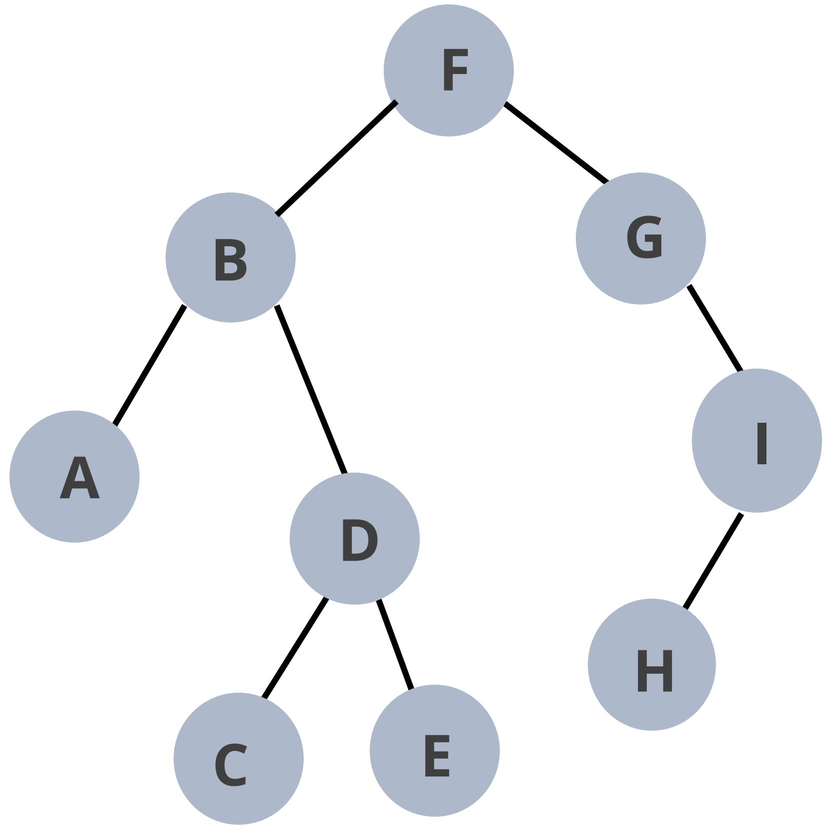
Post-order traversal

Pre-Order Traversal

It is a method of visiting all the nodes in a tree following a Root-Left-Right approach.

Steps to conduct pre-order traversal

1. Visit the root node
2. Visit all nodes on the left side
3. Visit all nodes on the right side



F	B	A	D	C	E	G	I	H

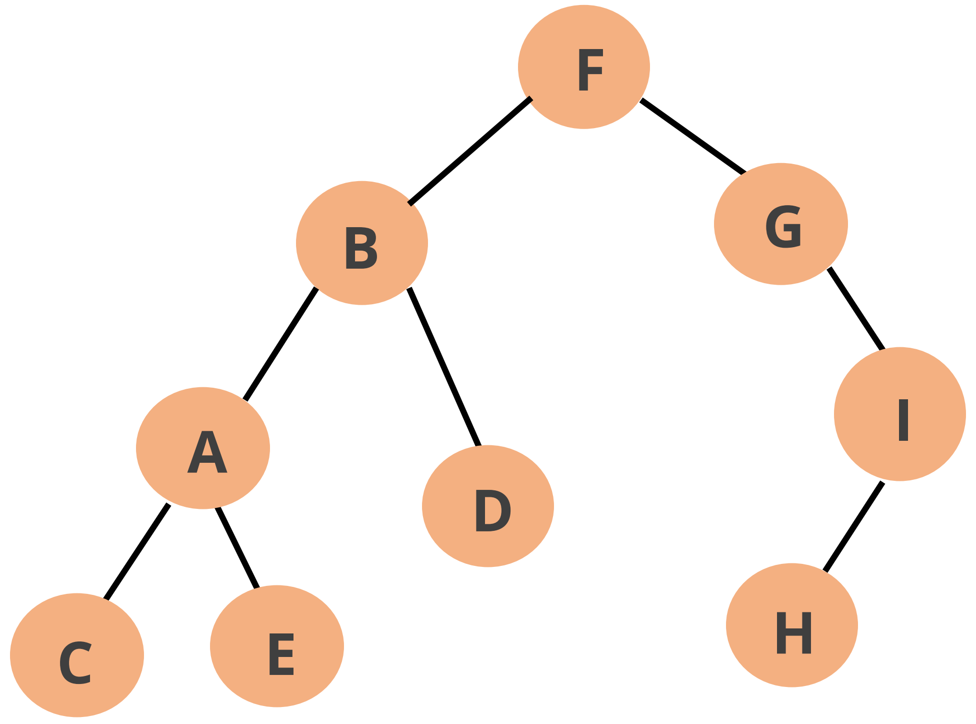
Example: It is used in file system navigation, where folders are accessed before their subfolders and files.

In-Order Traversal

It is a method to visit all the left nodes and come back to the root node and then visit all the right nodes. It follows a Left-Root-Right approach.

Steps to conduct in-order traversal

1. First, visit all nodes on the left side
2. Visit the root node
3. Visit all nodes on the right side



C	A	E	B	D	F	G	H	I

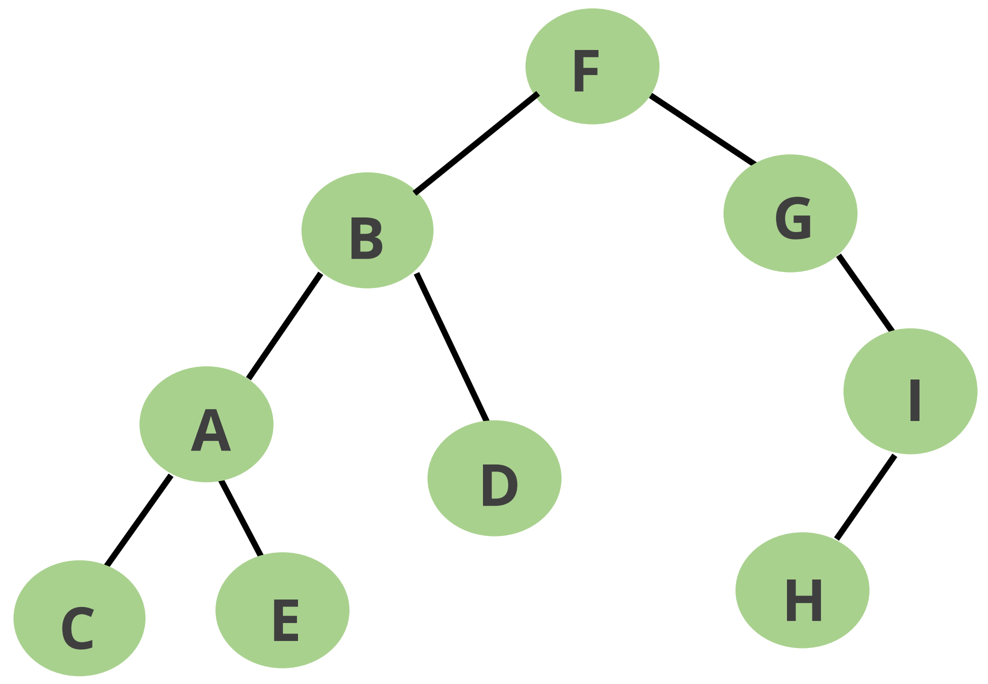
Example: It is used in binary search trees (BST) to retrieve sorted data in ascending order.

Post-Order Traversal

It is a method to visit the left and right children of a node before visiting the node itself.
It follows a Left-Right-Root approach.

Steps to conduct post-order traversal:

1. First, visit all nodes on the left side
2. Visit all nodes on the right side
3. Visit the root node



C	E	A	D	B	H	I	G	B

Example: It is used in deleting directories, where subfolders and files are deleted before the parent folder.

Quick Check

A file system organizes folders as a tree, where each folder (node) can contain subfolders (child nodes). Which traversal methods best fit the following operations?

- List folders before subfolders
- Retrieve files in sorted order (BST)
- Delete a folder after its contents

A. Pre-order, In-order, Post-order

B. Post-order, Pre-order, In-order

C. In-order, Post-order, Pre-order

D. Level-order, Pre-order, Post-order

Assisted Practice

Building and Traversing Binary Tree

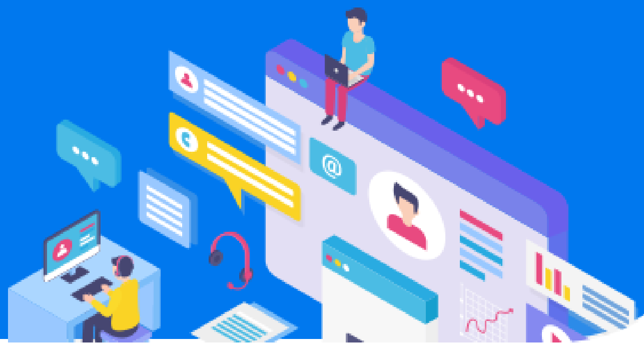
Duration: 20 Min.

Problem statement:
You have been assigned a task to demonstrate binary tree creation and traversal using JavaScript in the context of learning data structures, showcasing how insertion and in-order traversal work through interactive execution.

Outcome:
By the end of this demo, you will be able to implement and execute binary tree methods in JavaScript. This example demonstrates key techniques such as searching for a node and finding the smallest value in a binary tree.

Note: Refer to the demo document for detailed steps:
01_Building_and_Traversing_Binary_Tree

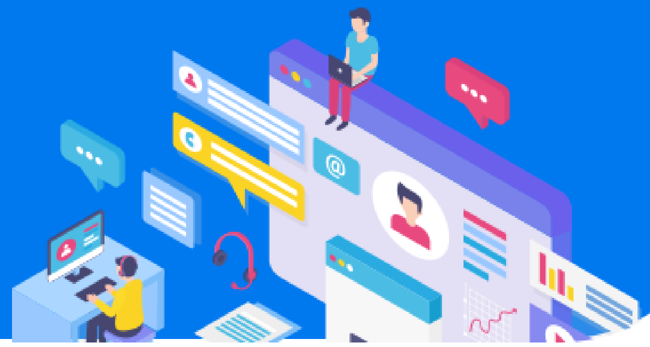
Assisted Practice: Guidelines



Steps to be followed:

1. Create a JavaScript file and execute it

Assisted Practice



Working with Binary Tree

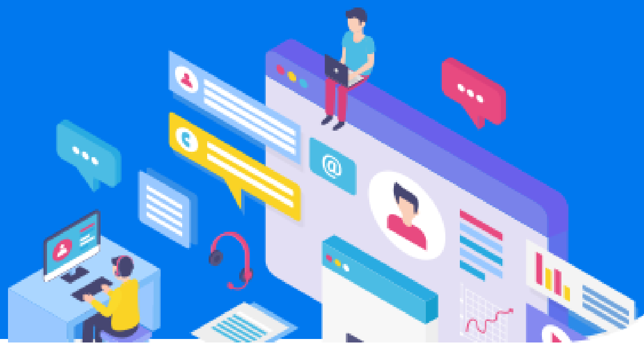
Duration: 20 Min.

Problem statement:
You have been assigned a task to demonstrate binary tree operations in JavaScript using node insertion, value searching, and minimum value retrieval to illustrate core algorithmic behavior in tree data structures.

Outcome:
By the end of this demo, you will be able to implement and execute binary tree operations in JavaScript. This example uses JavaScript to highlight key techniques such as node searching and identifying the smallest value in a binary tree.

Note: Refer to the demo document for detailed steps:
02_Working_with_Binary_Tree

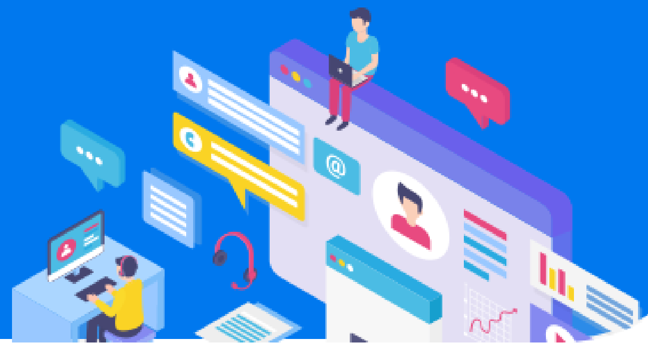
Assisted Practice: Guidelines



Steps to be followed:

1. Create a JavaScript file and execute it

Assisted Practice



Implementing AVL Tree

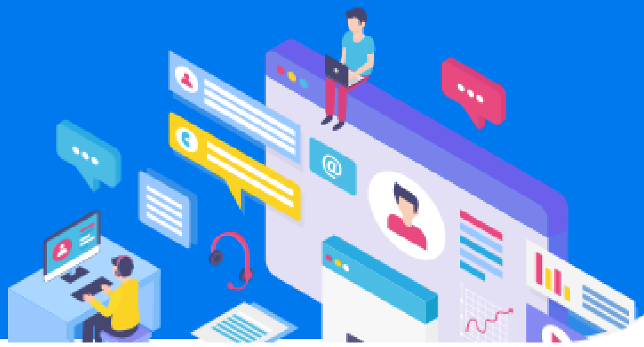
Duration: 20 Min.

Problem statement:
You have been assigned a task to demonstrate an AVL tree implementation in JavaScript using node insertion and self-balancing rotations to maintain height-balanced binary search tree properties.

Outcome:
By the end of this demo, you will be able to implement an AVL tree in JavaScript to maintain height-balanced binary search tree properties.

Note: Refer to the demo document for detailed steps:
03_Implementing_AVL_Tree

Assisted Practice: Guidelines



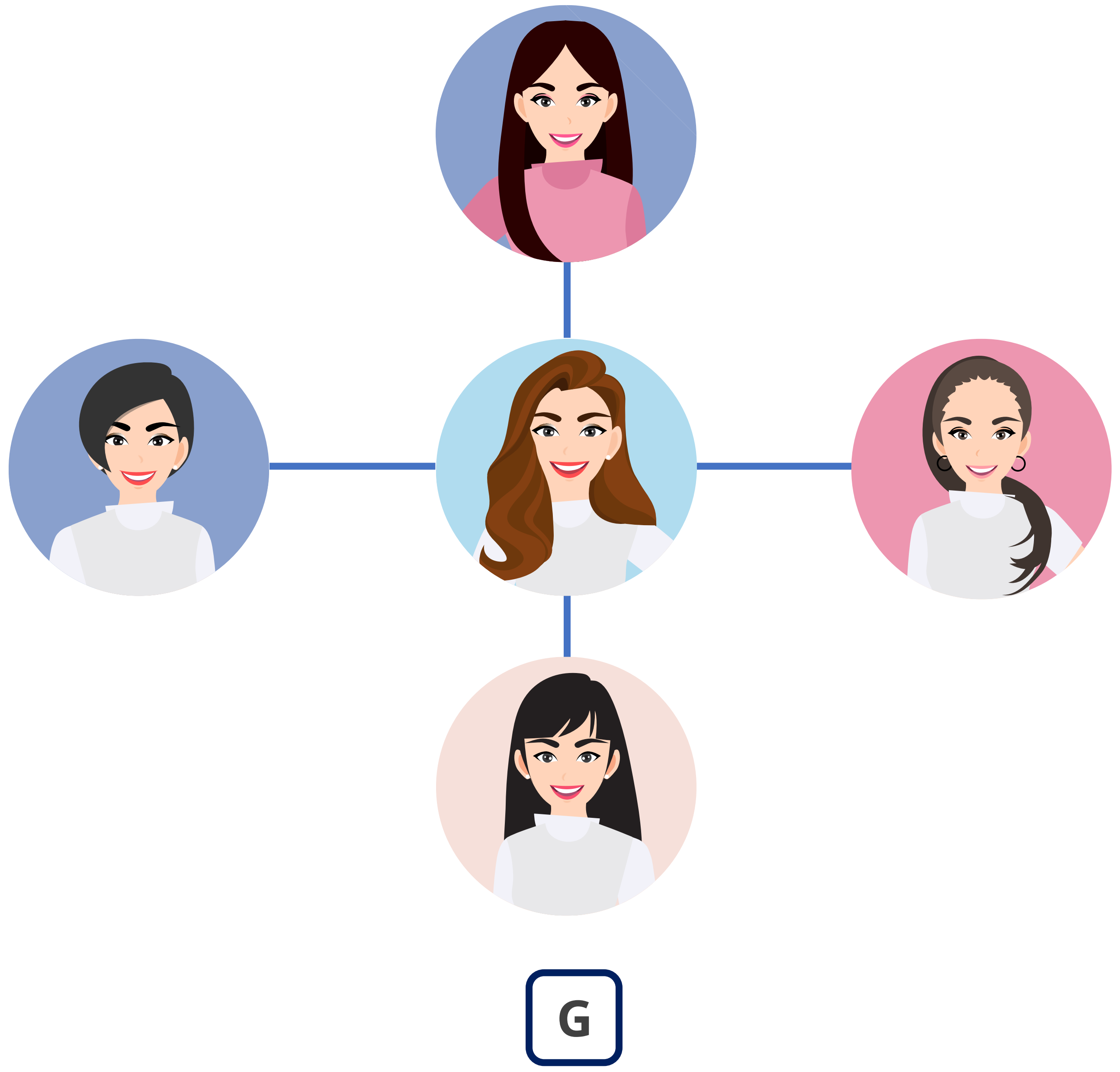
Steps to be followed:

1. Create a JavaScript file and execute it

Introduction to Graphs

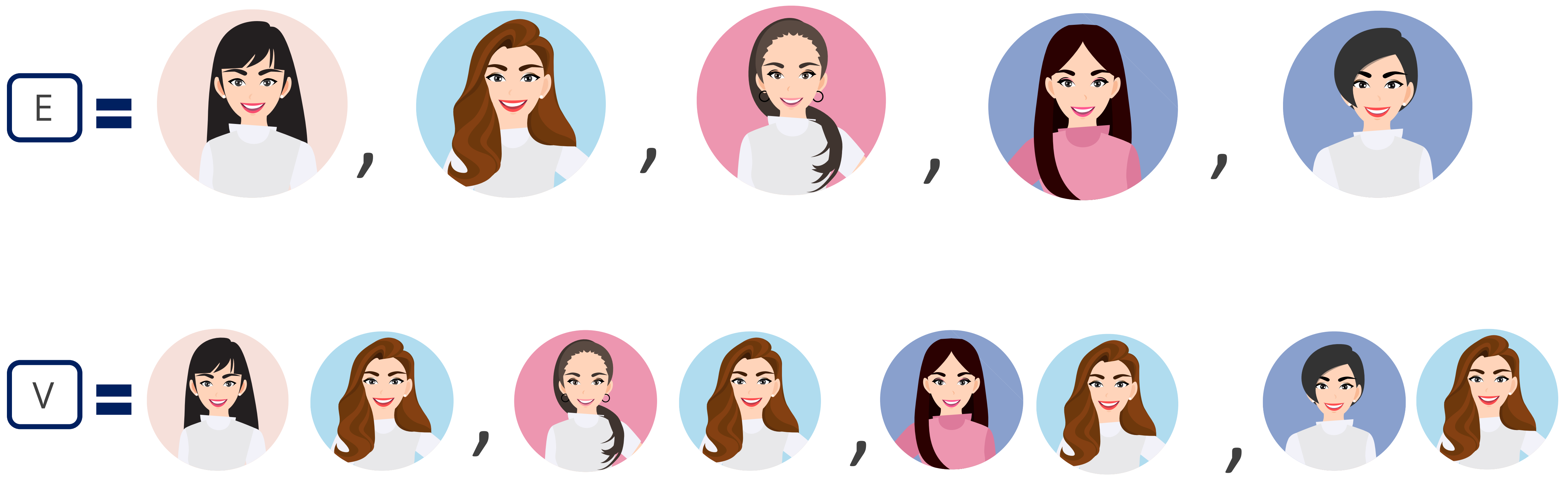
Graph Data Structure

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



Graph Data Structure

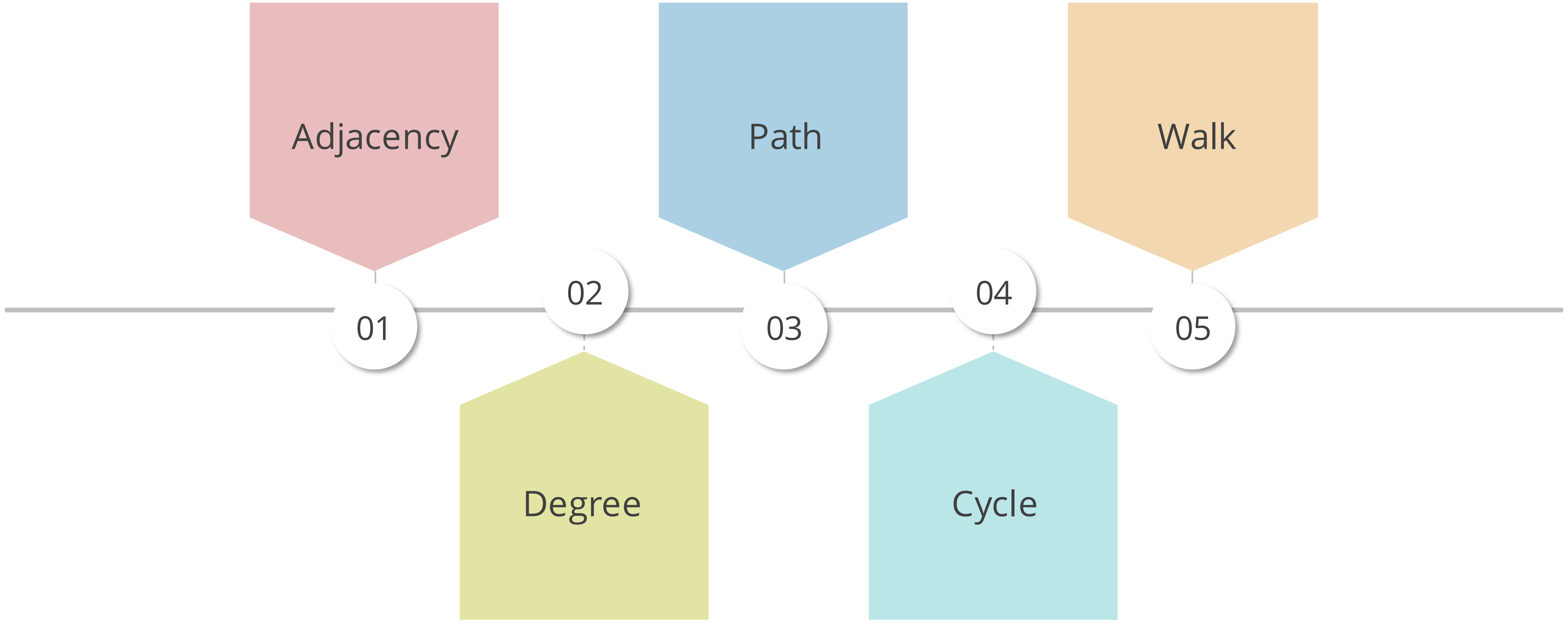
A graph, denoted as G, is defined as a pair of sets: (V, E), where V represents the set of vertices, and E represents the set of edges connecting these pairs of vertices.



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

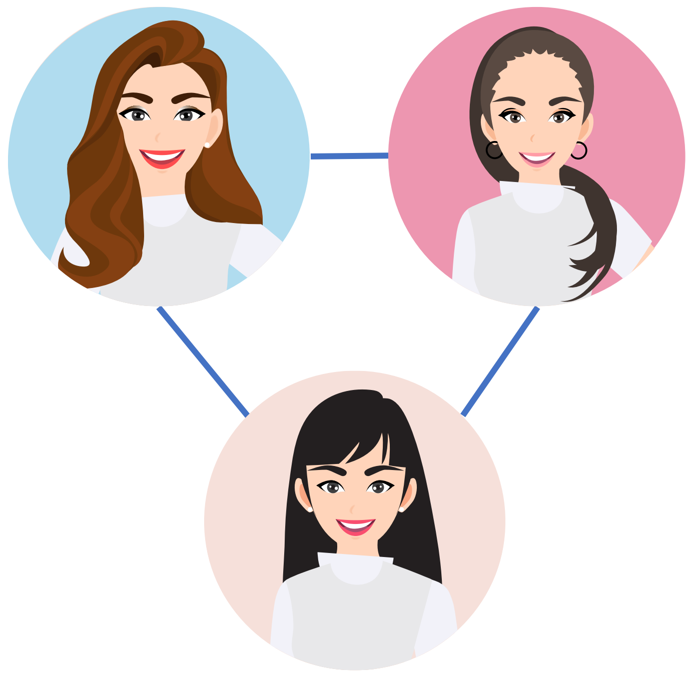
Graph Terminologies

The following are the key graph terminologies that define the structure, relationships, and properties of graphs in data structures:



Adjacency in Graph Data Structure

It represents relationships between vertices and edges, defining how elements are connected. The following are its types:

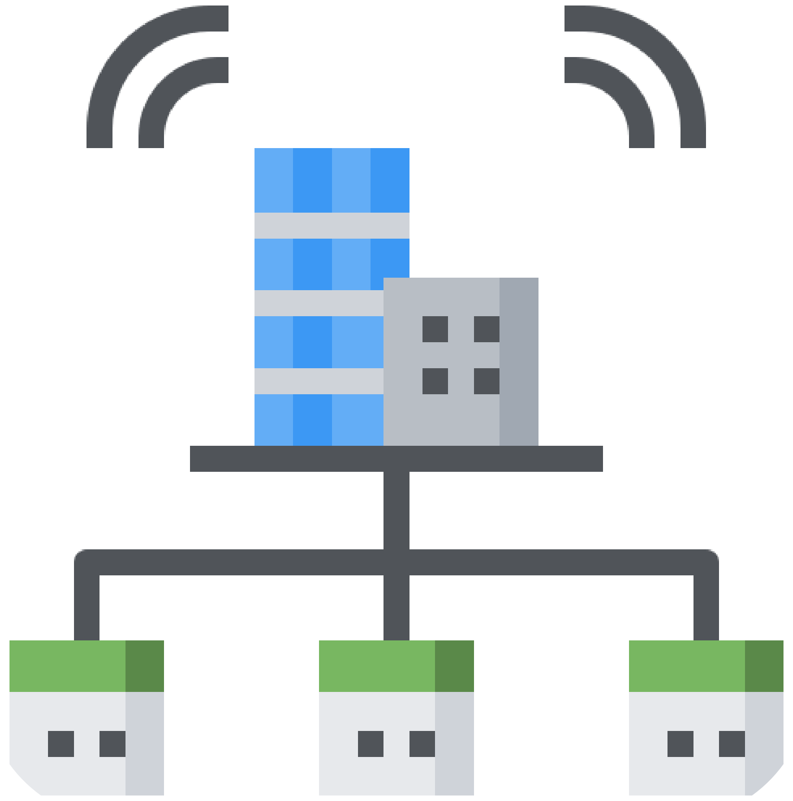


- **Adjacent vertices:** Refers to vertices that share an edge
- **Adjacent edges:** Describes edges that have a common vertex

Example: In social networks, two users are adjacent if they are directly connected as friends or followers.

Degree in Graph Data Structure

In a graph, the degree of a vertex represents the number of direct connections it has with other vertices.



Example: In a city road network, the degree of an intersection represents the number of roads connecting to it.

Path in Graph Data Structure

A path in a graph is a sequence of distinct vertices where an edge directly connects each consecutive pair.

Example: In navigation systems, the shortest path between two locations represents the optimal route on a map.

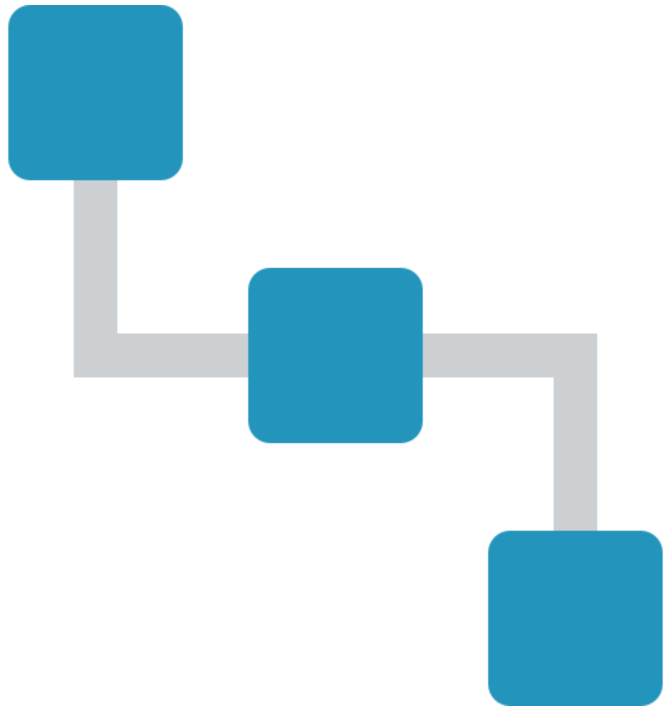
Cycle in Graph Data Structure

A cycle in a graph is a path that starts and ends at the same vertex, with all edges and vertices being distinct except for the starting and ending vertex.

Example: In transportation networks, circular train routes form cycles, allowing passengers to return to their starting station.

Walk in Graph Data Structure

A walk in a graph is a sequence of vertices where each consecutive pair is connected by an edge, and vertices or edges may be repeated.



Example: In social networks, browsing through friends' connections forms a walk, where users may revisit profiles multiple times.

Types of Graph

Types of Graph

A graph is denoted as G = (V, E). The different types of graph are:

01 Finite graph

02 Infinite graph

03 Trivial graph

04 Simple graph

05 Multi graph

06 Null graph

07 Complete graph

08 Pseudograph

09 Regular graph

Types of Graph

A graph is denoted as G = (V, E). The different types of graph are:

10 Weighted graph

11 Directed graph

12 Undirected graph

13 Connected graph

14 Disconnected graph

15 Cyclic Graph

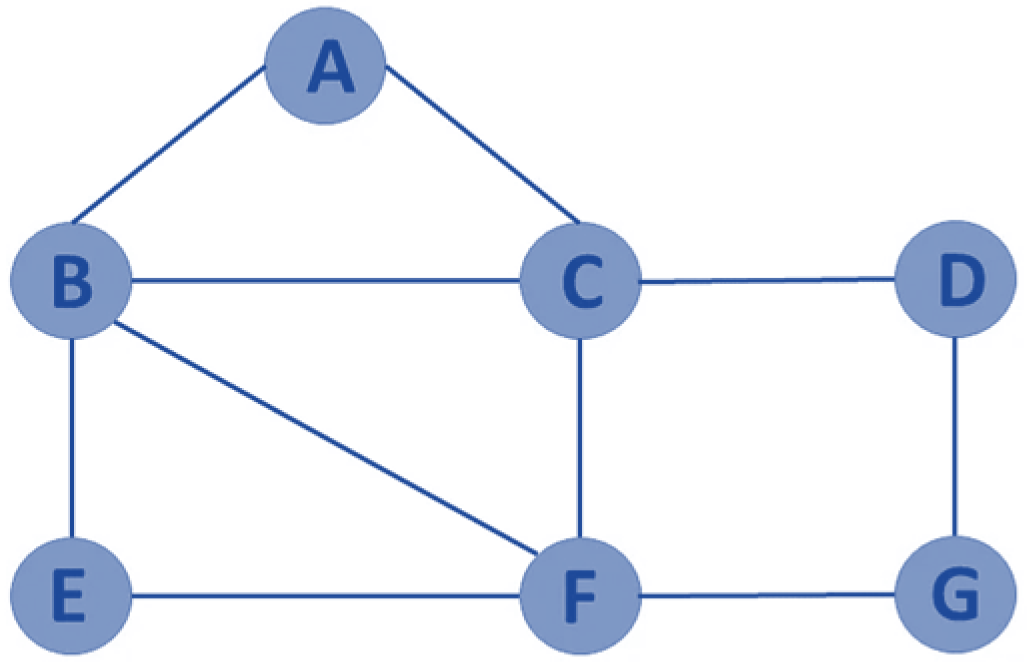
16 Acyclic graph

17 Directed acyclic graph

18 Subgraph

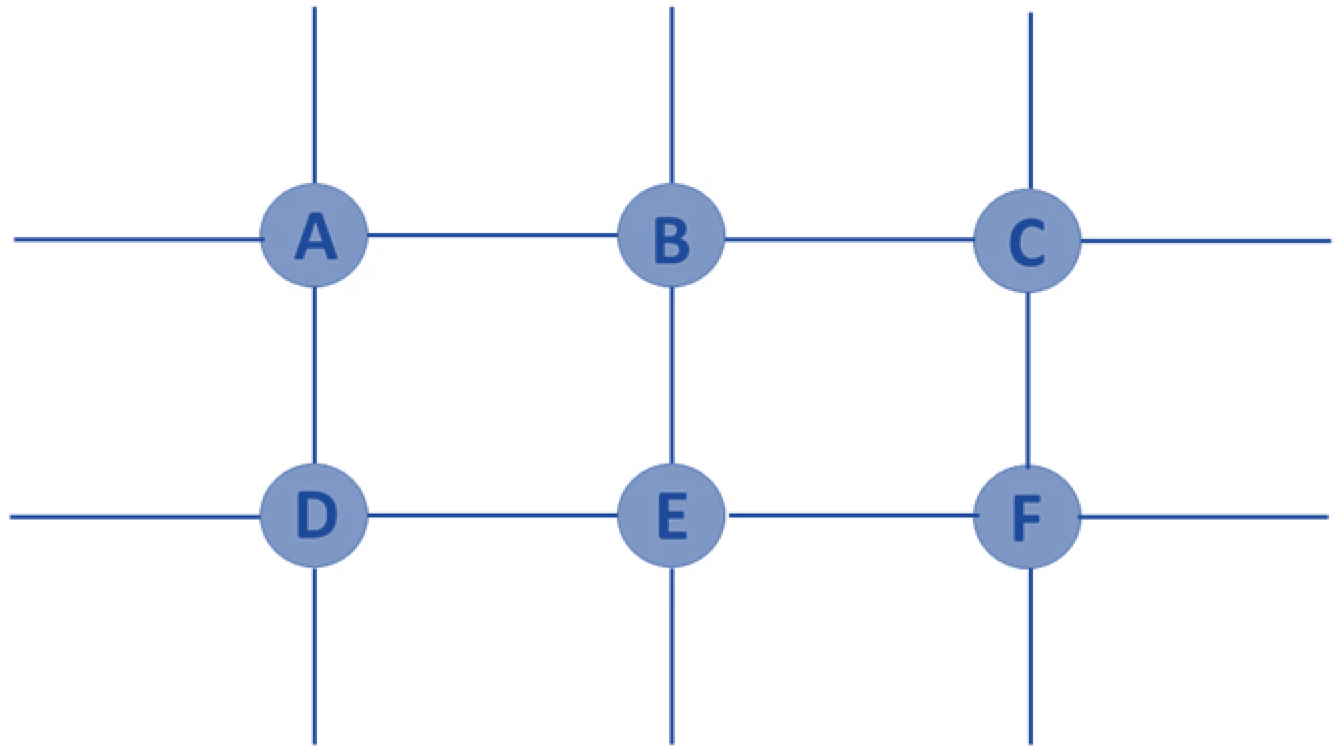
Finite Graph

A finite graph is a graph where the sets of vertices and edges are finite.



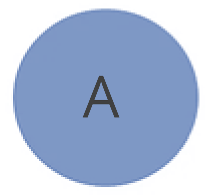
Infinite Graph

An infinite graph is a graph that has an unbounded or limitless number of vertices and edges.



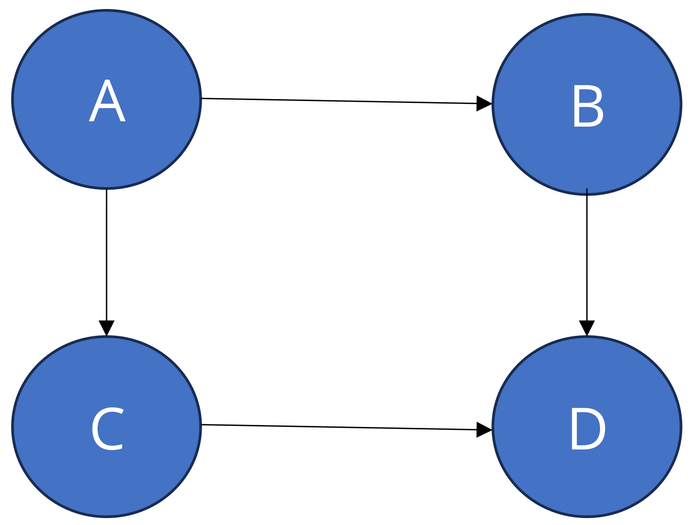
Trivial Graph

A trivial graph consists of just one vertex and no edges. It is the simplest valid graph and is often used as a base case in graph theory and algorithms.



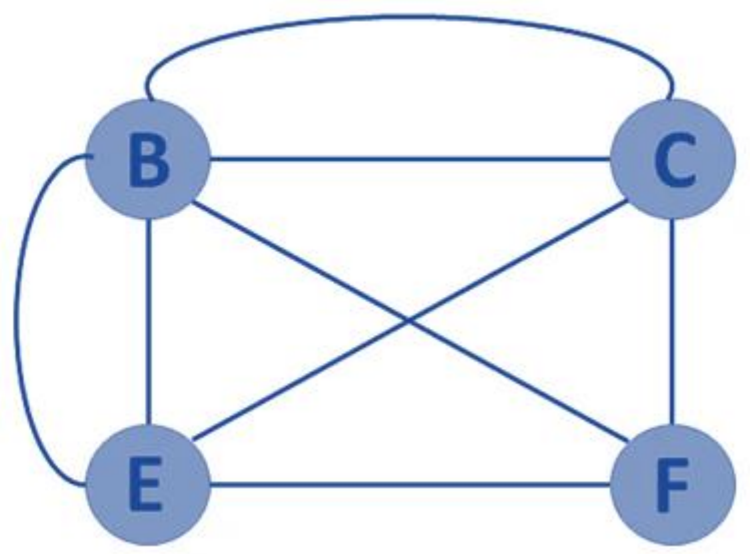
Simple Graph

A simple graph is a graph where each pair of distinct vertices is connected by exactly one edge.



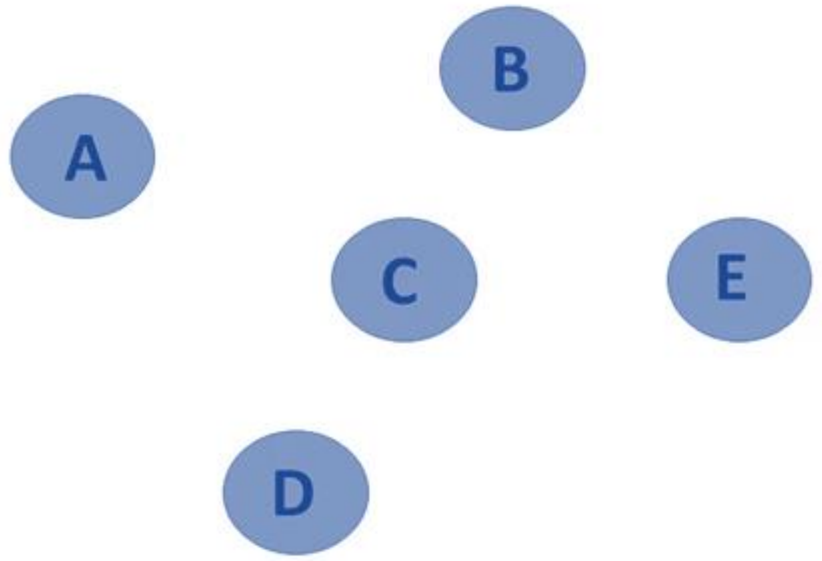
Multi Graph

A multigraph is a graph where multiple edges can connect a pair of vertices.



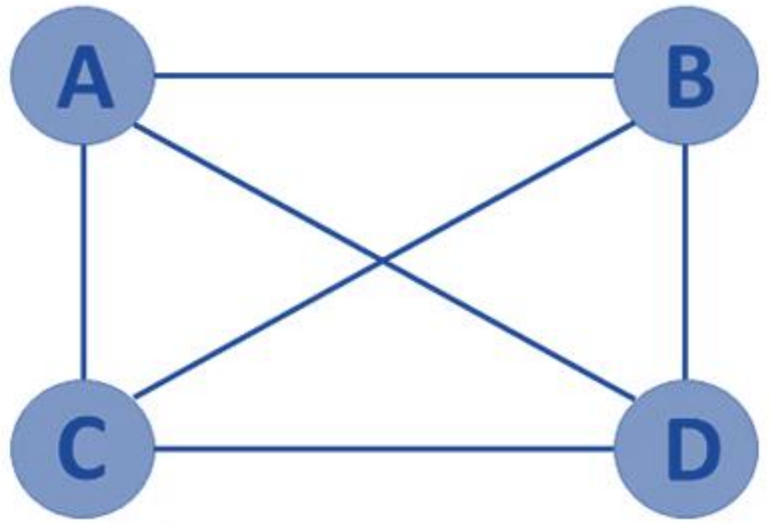
Null Graph

A null graph contains vertices but lacks edges, which means no vertices are connected.



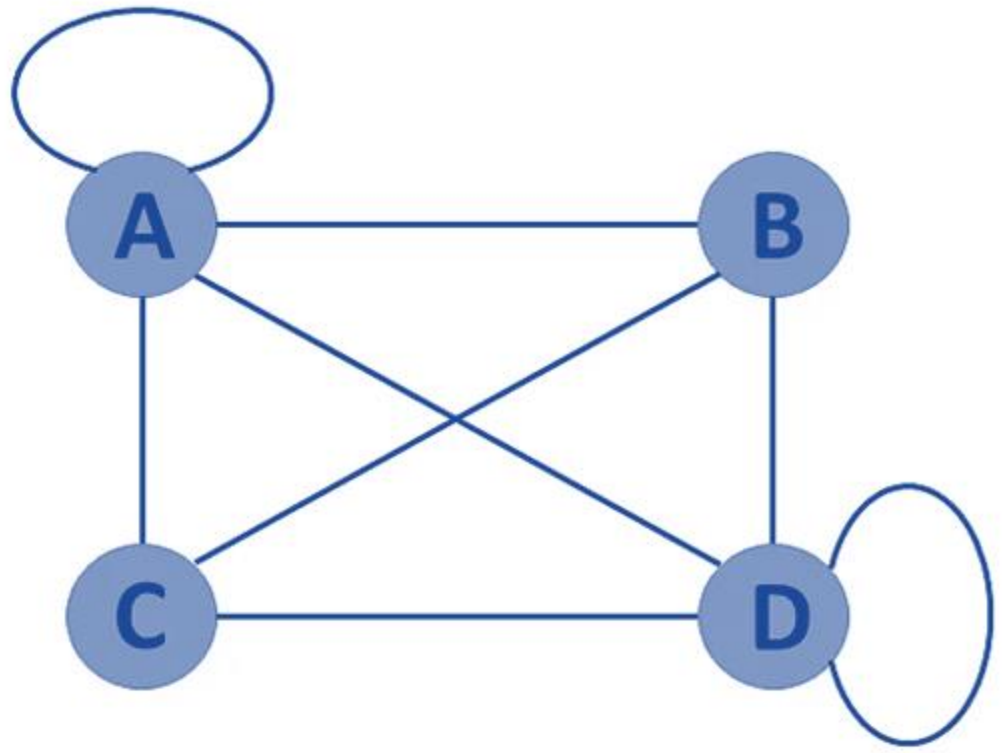
Complete Graph

A complete graph is a simple graph, where every pair of distinct vertices is connected by a unique edge.



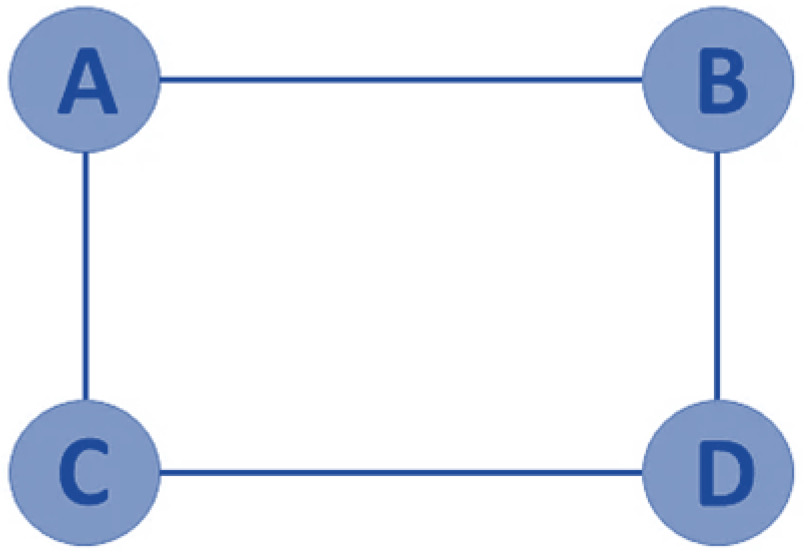
Pseudograph

It is a type of graph that includes self-loops, where a vertex can have an edge connecting back to itself, along with other regular edges.



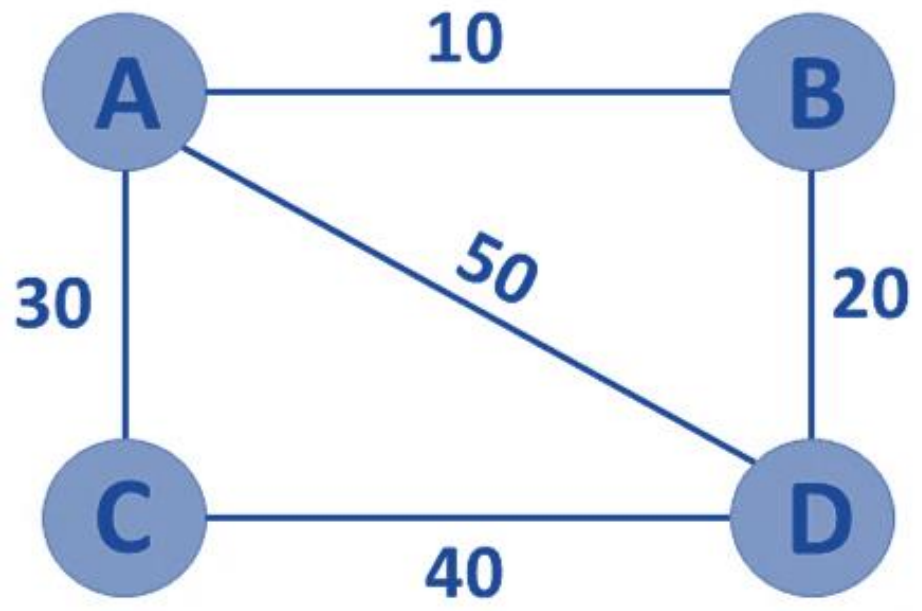
Regular Graph

A regular graph is where each vertex has the same number of edges or the same degree.



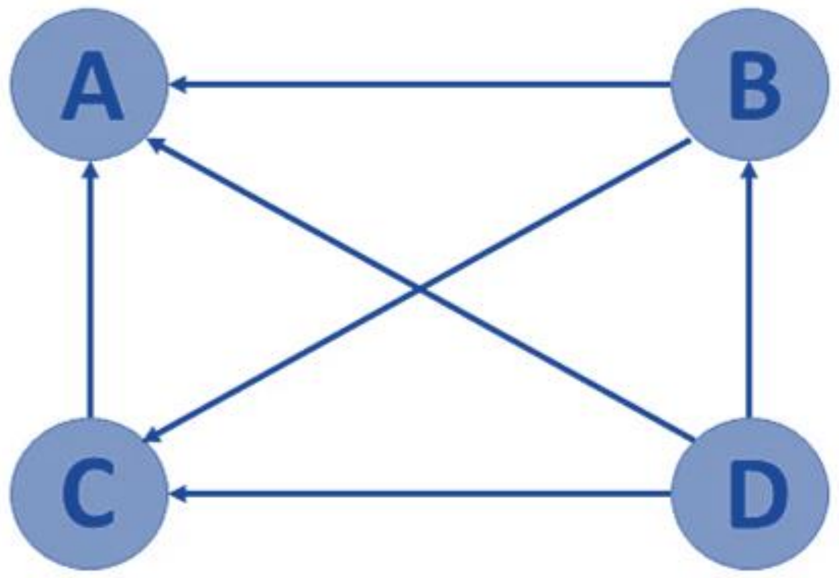
Weighted Graph

A weighted graph is a graph where each edge is assigned a specific value or weight, representing the cost of traversing that edge.



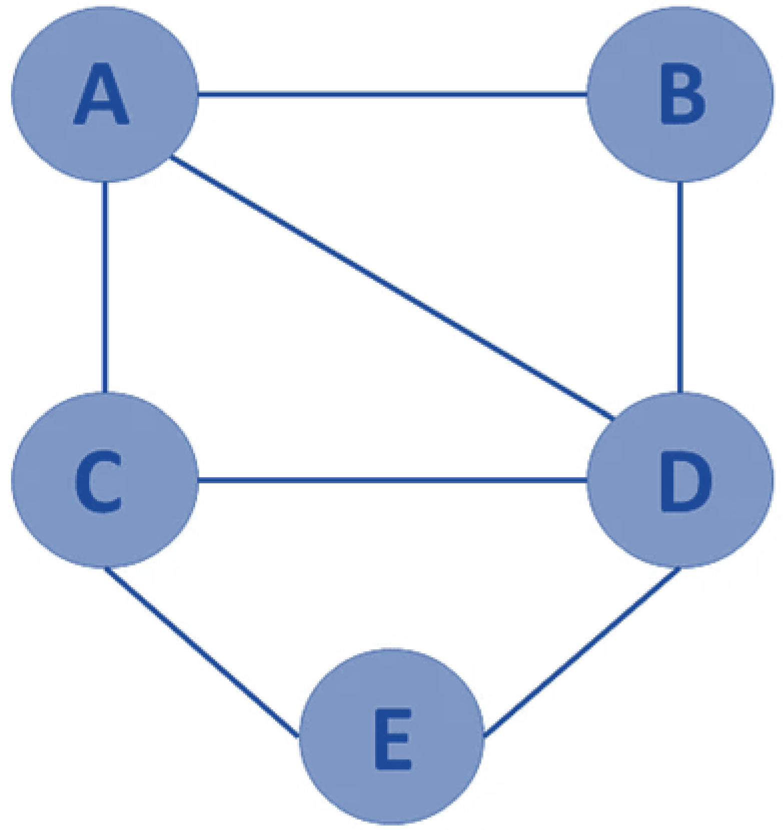
Directed Graph

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



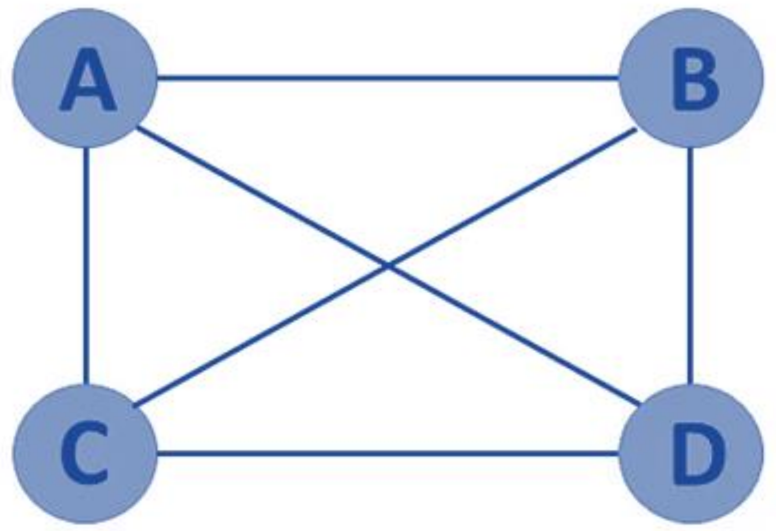
Undirected Graph

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



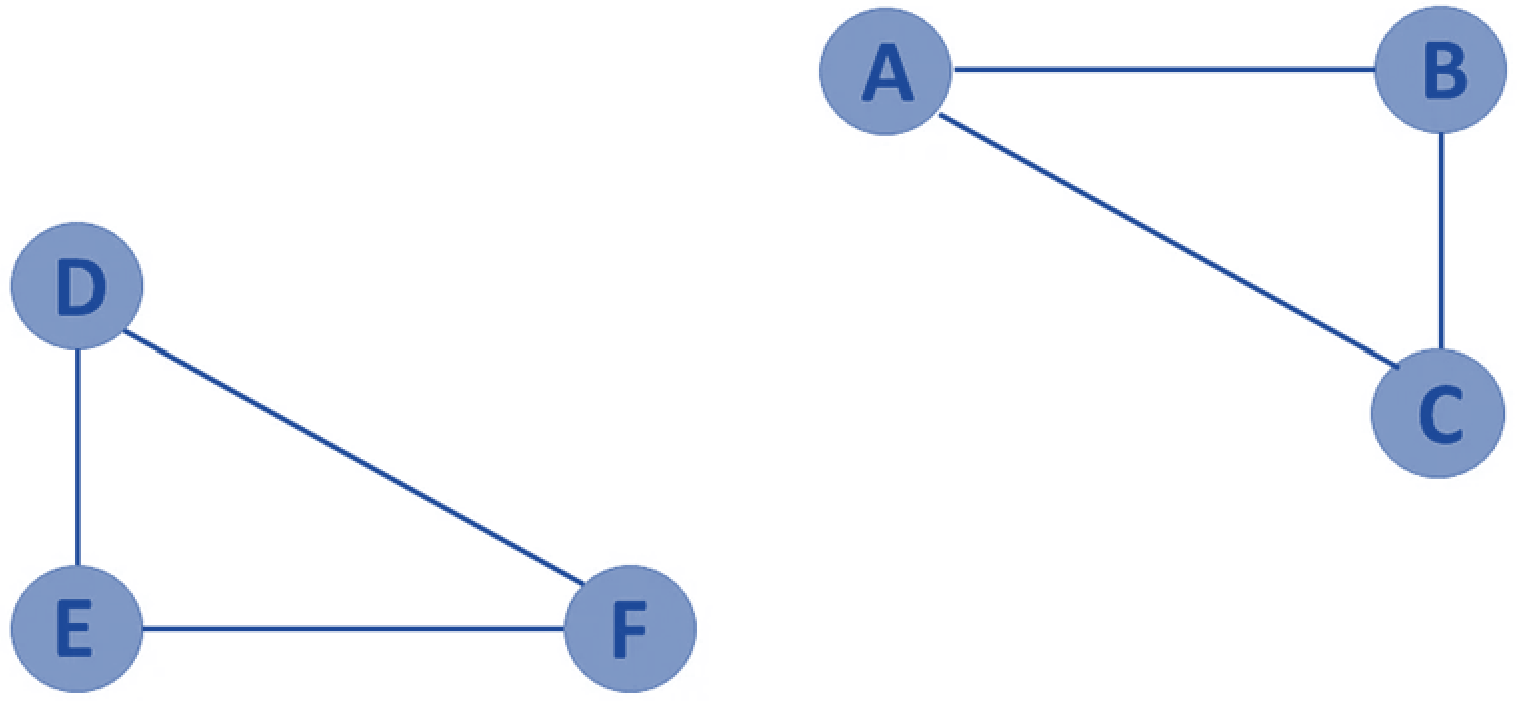
Connected Graph

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



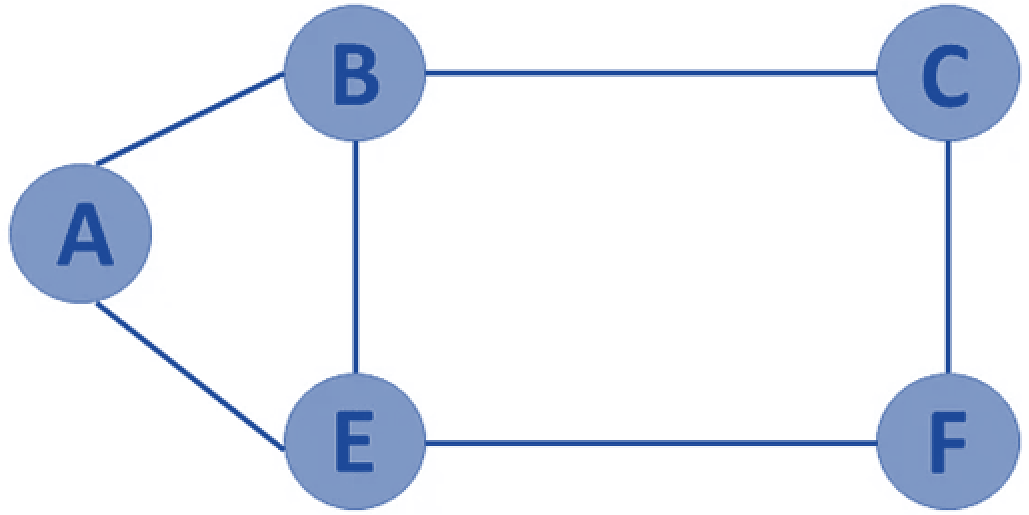
Disconnected Graph

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.



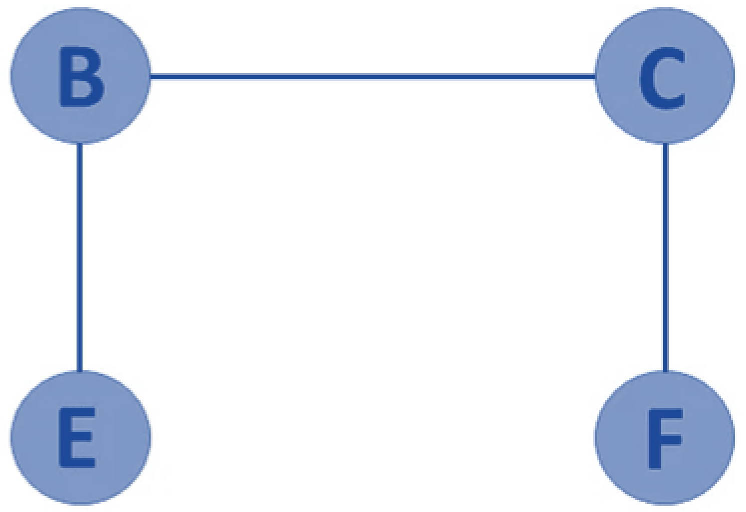
Cyclic Graph

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



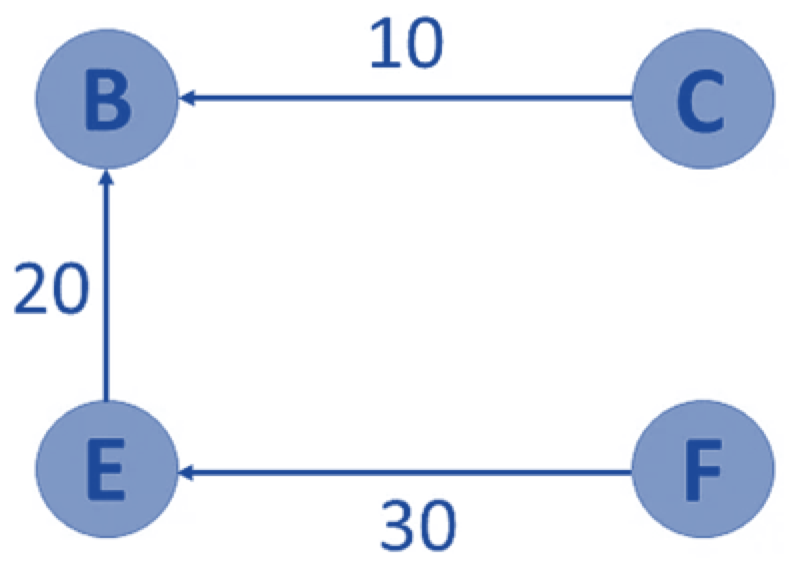
Acyclic Graph

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



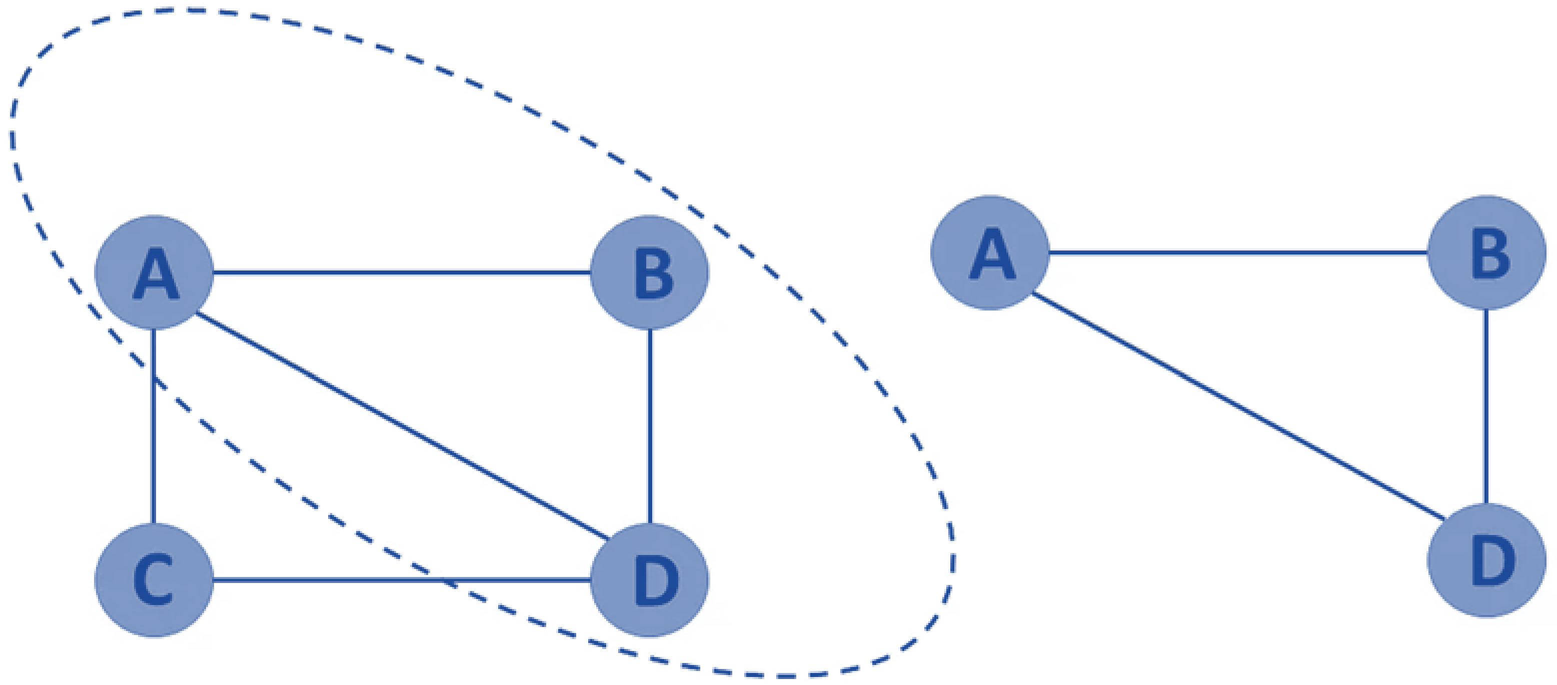
Directed Acyclic Graph

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.



Quick Check

A social networking platform wants to analyze user connections to recommend new friends. Each user is represented as a vertex, and their friendships as edges in a graph. The system needs to determine:

- Directly connected users
- The number of connections each user has
- The shortest path between two users

Which of the following graph terminologies best correspond to the above requirements?

A. Adjacency, degree, and path

B. Walk, cycle, and adjacency

C. Incidence, walk, and degree

D. Path, cycle, and adjacency

Graph Representation

Graph Representation

The two ways to represent a graph data structure are:

01

Adjacency matrix

02

Adjacency list

Adjacency Matrix

The features of representing a graph using the adjacency matrix are:

Fixed space usage:

Uses a 2D array of size V×V, even if the graph has very few edges.

Fast edge lookup:

Edge existence between two vertices is checked in constant time O(1).

Not ideal for large sparse graphs:

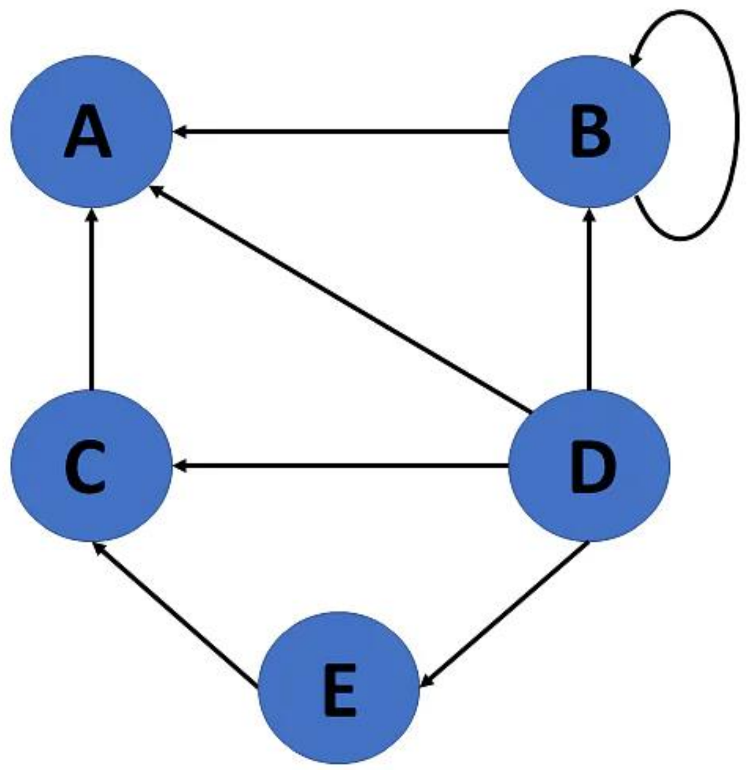
Wastes space when most entries are 0 (no connection).

Better for dense graphs:

Performs well when the graph has many edges (close to a complete graph).

Directed Graph Representation

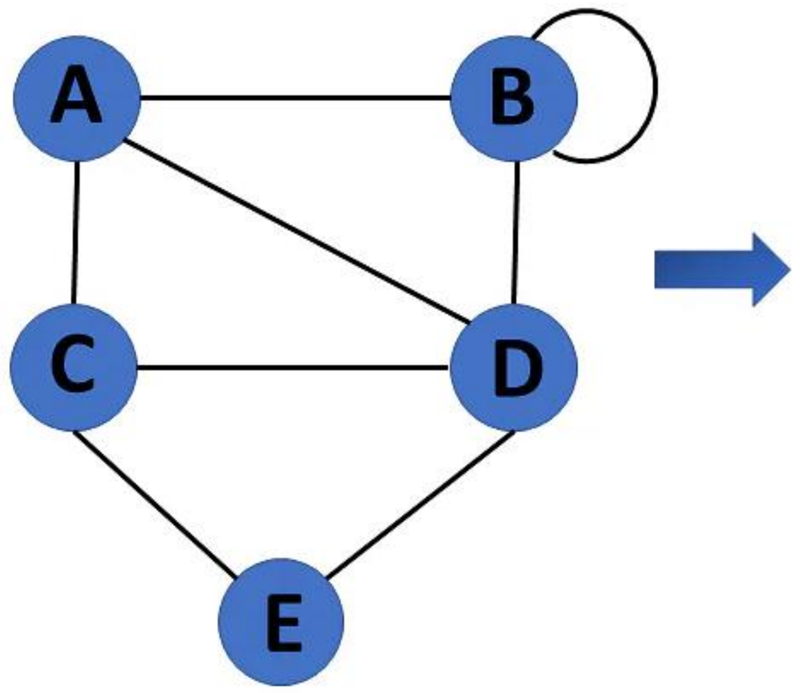
Here is the representation of a directed graph using an adjacency matrix, where each cell shows whether a directed edge exists between two vertices.



$$\begin{array}{c|ccccc}
 & A & B & C & D & E \\
\hline
A & 0 & 0 & 0 & 0 & 0 \\
B & 1 & 1 & 0 & 0 & 0 \\
C & 1 & 0 & 0 & 0 & 0 \\
D & 1 & 1 & 1 & 0 & 1 \\
E & 0 & 0 & 1 & 0 & 0
\end{array}$$

Undirected Graph Representation

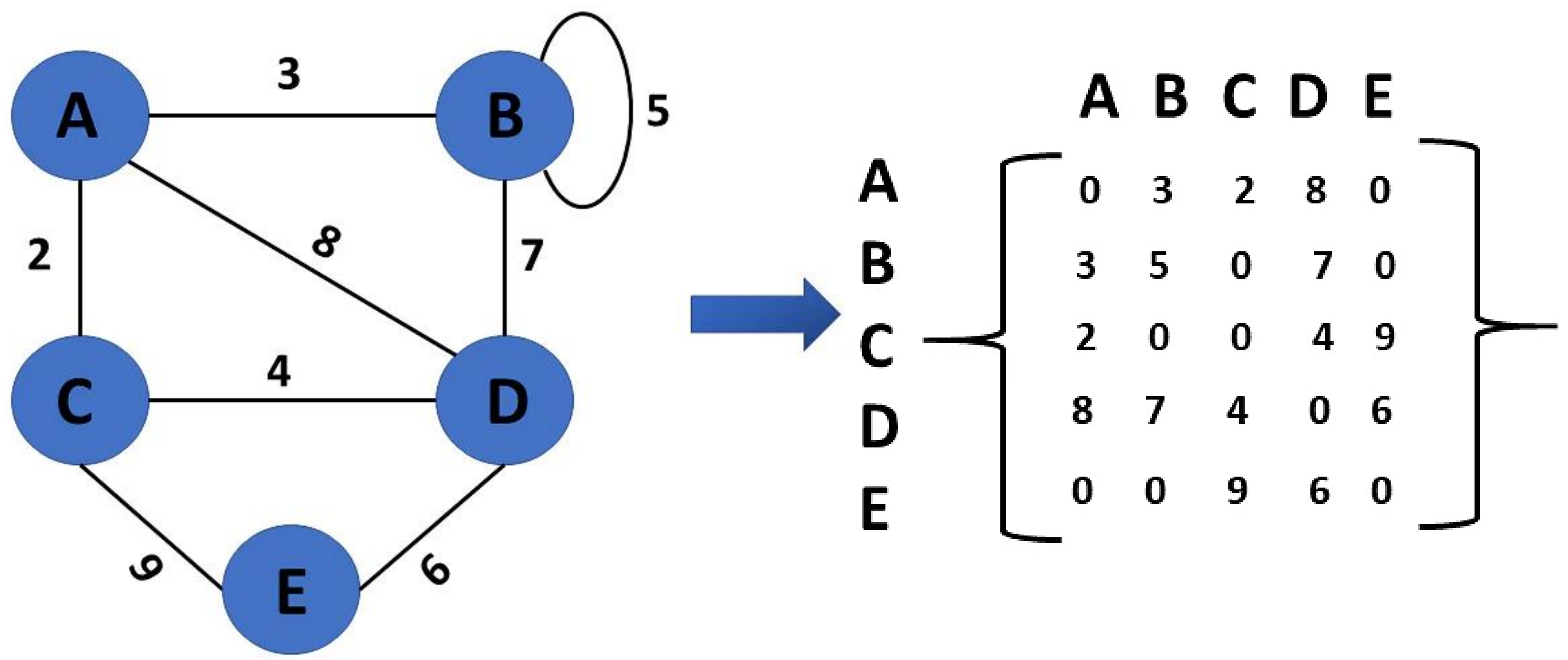
Here is the representation of an undirected graph using an adjacency matrix, where 1 indicates a connection between two vertices without direction.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 1 & 0 & 1 & 0 \\ C & 1 & 0 & 0 & 1 & 1 \\ D & 1 & 1 & 1 & 0 & 1 \\ E & 0 & 0 & 1 & 1 & 0 \end{array}$$

Weighted Undirected Graph Representation

Here is the representation of a weighted undirected graph using an adjacency matrix, where each cell shows the weight of the edge between two vertices.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 3 & 2 & 8 & 0 \\ B & 3 & 5 & 0 & 7 & 0 \\ C & 2 & 0 & 0 & 4 & 9 \\ D & 8 & 7 & 4 & 0 & 6 \\ E & 0 & 0 & 9 & 6 & 0 \end{array}$$

Adjacency List

The features of representing a graph using the adjacency matrix are:

Space efficient:

Uses less memory when the graph has fewer edges (not all nodes are connected)

Dynamic edge storage:

Easily supports adding or removing edges without resizing a structure

Faster iteration:

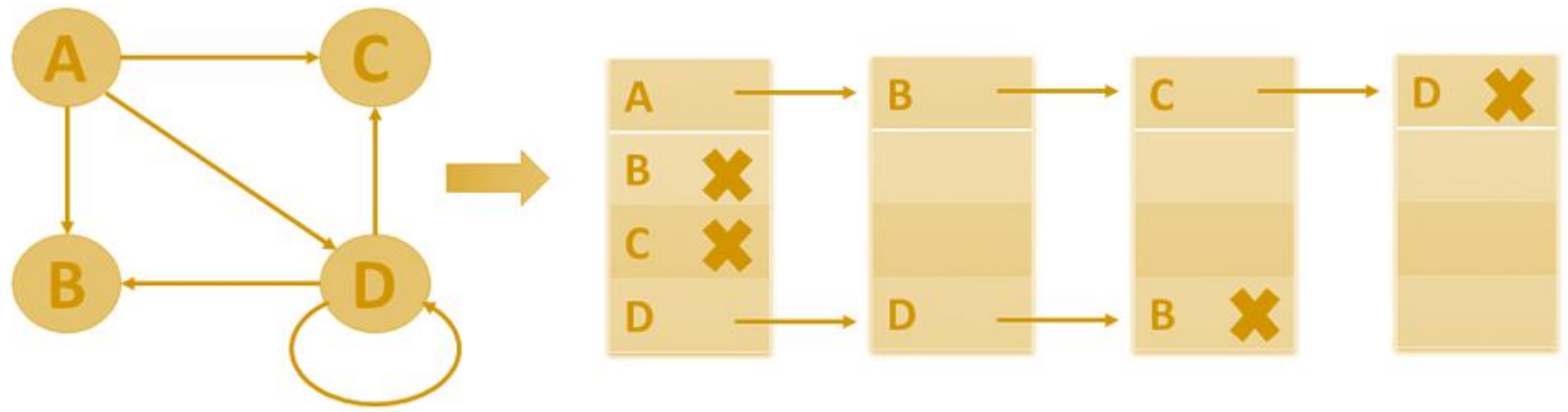
Traverses neighbors of a vertex faster and only visits existing connections

Slower edge existence check:

Checks if an edge exists between two nodes requires searching through a list

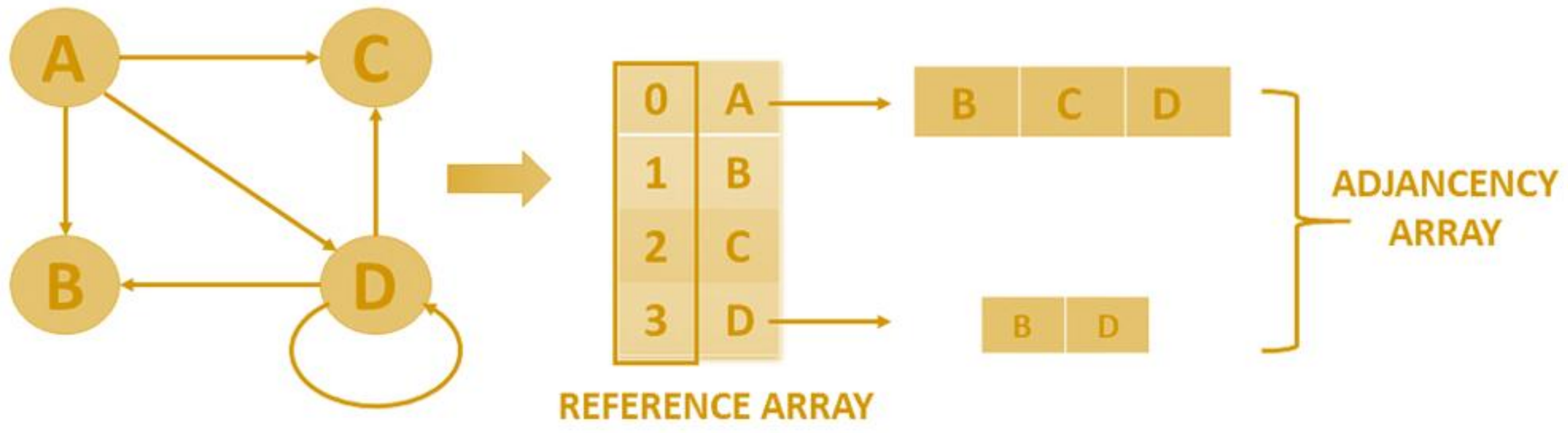
Graph Representation using Linked List

Here is the representation of a weighted undirected graph using an adjacency list, where each vertex points to a linked list of its connected neighbors with weights.



Graph Representation using Array

Here is the representation of a weighted undirected graph using an adjacency list stored in arrays, where each vertex references a list of its connected neighbors.



Quick Check

A city's bus network consists of bus stops (vertices) connected by roads (edges). The transport department needs an efficient way to store and retrieve direct connections between bus stops.
Which graph representation is best suited for this?

A. Adjacency matrix: It uses a 2D matrix to store connections.

B. Adjacency list: Each stop maintains a list of connected stops.

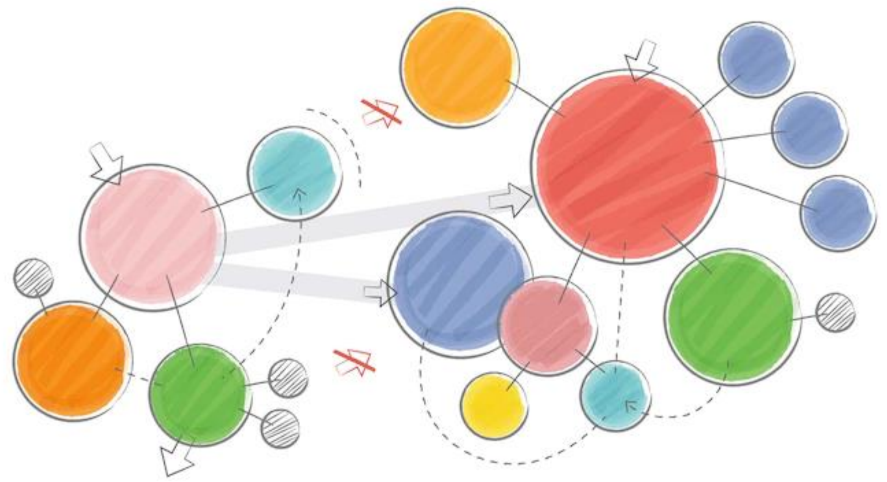
C. Incidence matrix: It represents connections between stops and roads.

D. Edge list: It stores direct roads as vertex pairs.

Graph Traversal

Graph Traversal

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

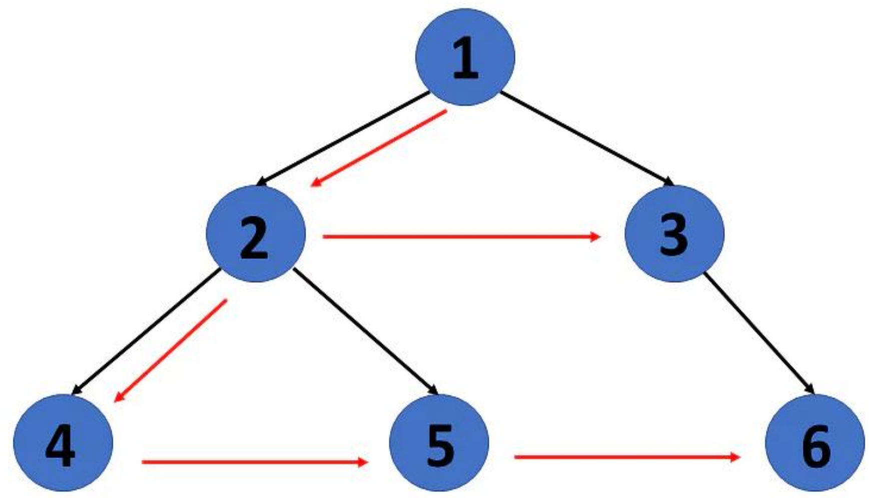


Graph traversing can be performed in two ways:

- Breadth-first search (BFS) algorithm
- Depth-first search (DFS) algorithm

Breadth-First Search (BFS) Algorithm

It is a graph traversal algorithm that explores all nodes at the present depth before moving to the next level.



BFS	1	2	3	4	5	6

Steps:

- 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.

BFS: Algorithm

The following are the steps of the Breadth-First Search (BFS) traversal algorithm:

Step 1: Start by selecting a starting vertex and mark it as visited

Step 2: Enqueue the starting vertex into a queue

Step 3: While the queue is not empty:

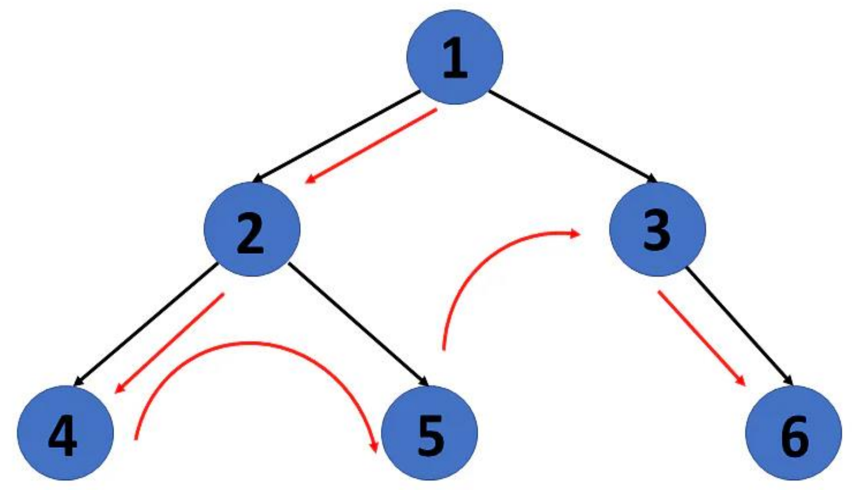
a. Dequeue a vertex from the queue

b. Visit all its unvisited adjacent vertices

c. Mark each as visited and enqueue them

Step 4: Repeat the process until all reachable vertices are visited

Depth-First Search (DFS) Algorithm



DFS						
	1	2	4	5	3	6

Steps:

- 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 toward the most recently unexplored nodes.
- The DFS algorithm utilizes a stack data structure for this purpose.

DFS: Algorithm

The following are the steps of the depth-first search (DFS) traversal algorithm:

Step 1: Start by selecting a starting vertex and mark it as visited

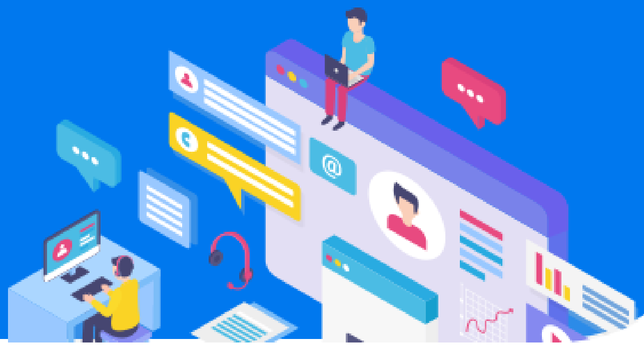
Step 2: Push the starting vertex onto a stack

Step 3: While the stack is not empty:

1. Pop a vertex from the stack
2. Visit one of its unvisited adjacent vertices
3. Mark it as visited and push it onto the stack

Step 4: Repeat the process until all reachable vertices are visited

Assisted Practice



Creating and Representing Graph

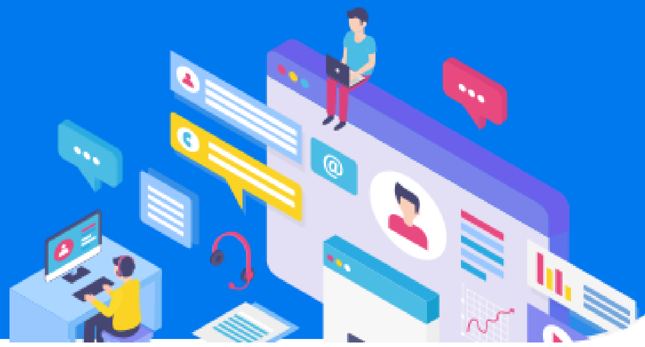
Duration: 20 Min.

Problem statement:
You have been assigned a task to demonstrate graph creation and representation in JavaScript using an adjacency list by adding vertices, connecting edges, and displaying the graph structure for clear visualization of relationships.

Outcome:
By the end of this demo, you will be able to create and represent a graph in JavaScript using an adjacency list to visualize relationships clearly.

Note: Refer to the demo document for detailed steps:
04_Creating_and_Representing_Graph

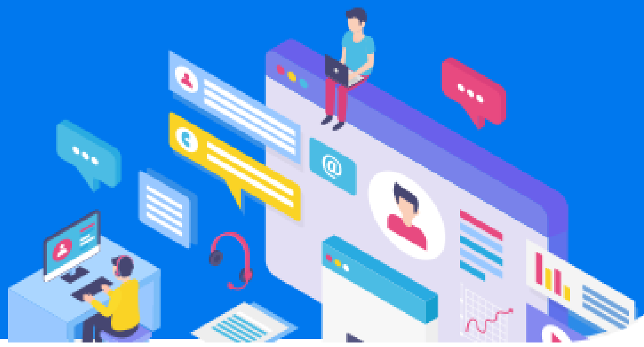
Assisted Practice: Guidelines



Steps to be followed:

1. Create a JavaScript file and execute it

Assisted Practice



Traversing a Graph

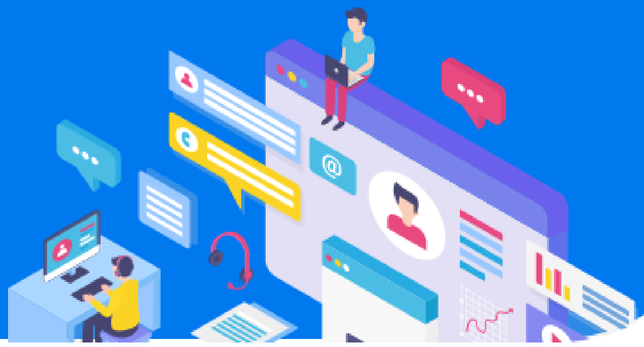
Duration: 20 Min.

Problem statement:
You have been assigned a task to demonstrate graph traversal using depth-first search (DFS) in JavaScript and an adjacency list to illustrate how graph exploration works programmatically.

Outcome:
By the end of this demo, you will be able to implement a graph and perform a depth-first traversal using JavaScript. By creating a simple yet effective graph representation through vertices and edges and utilizing the depth-first traversal method, you have explored how to navigate through the graph systematically.

Note: Refer to the demo document for detailed steps:
05_Traversing_a_Graph

Assisted Practice: Guidelines



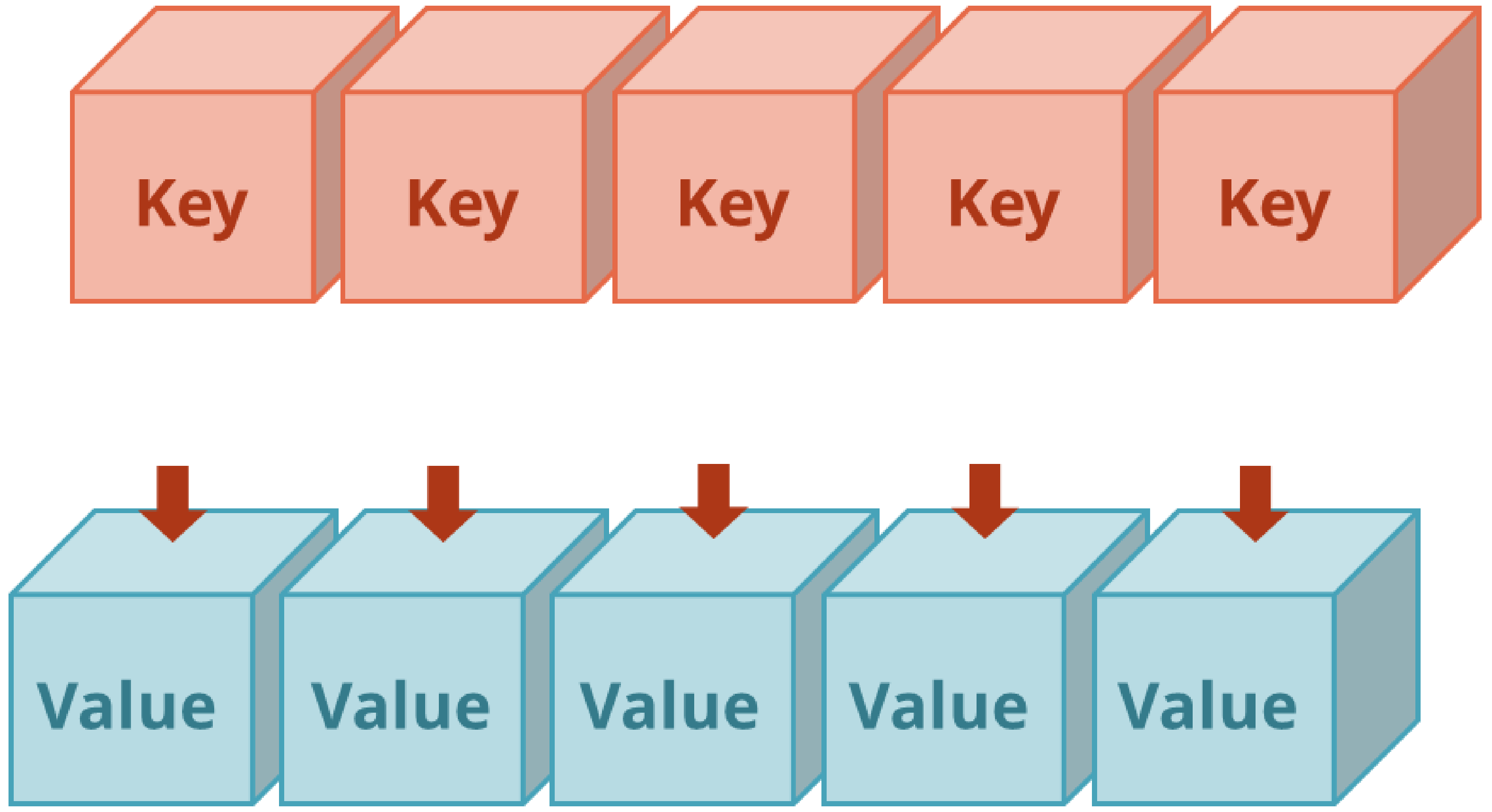
Steps to be followed:

1. Create a JavaScript file and execute it

HashMap

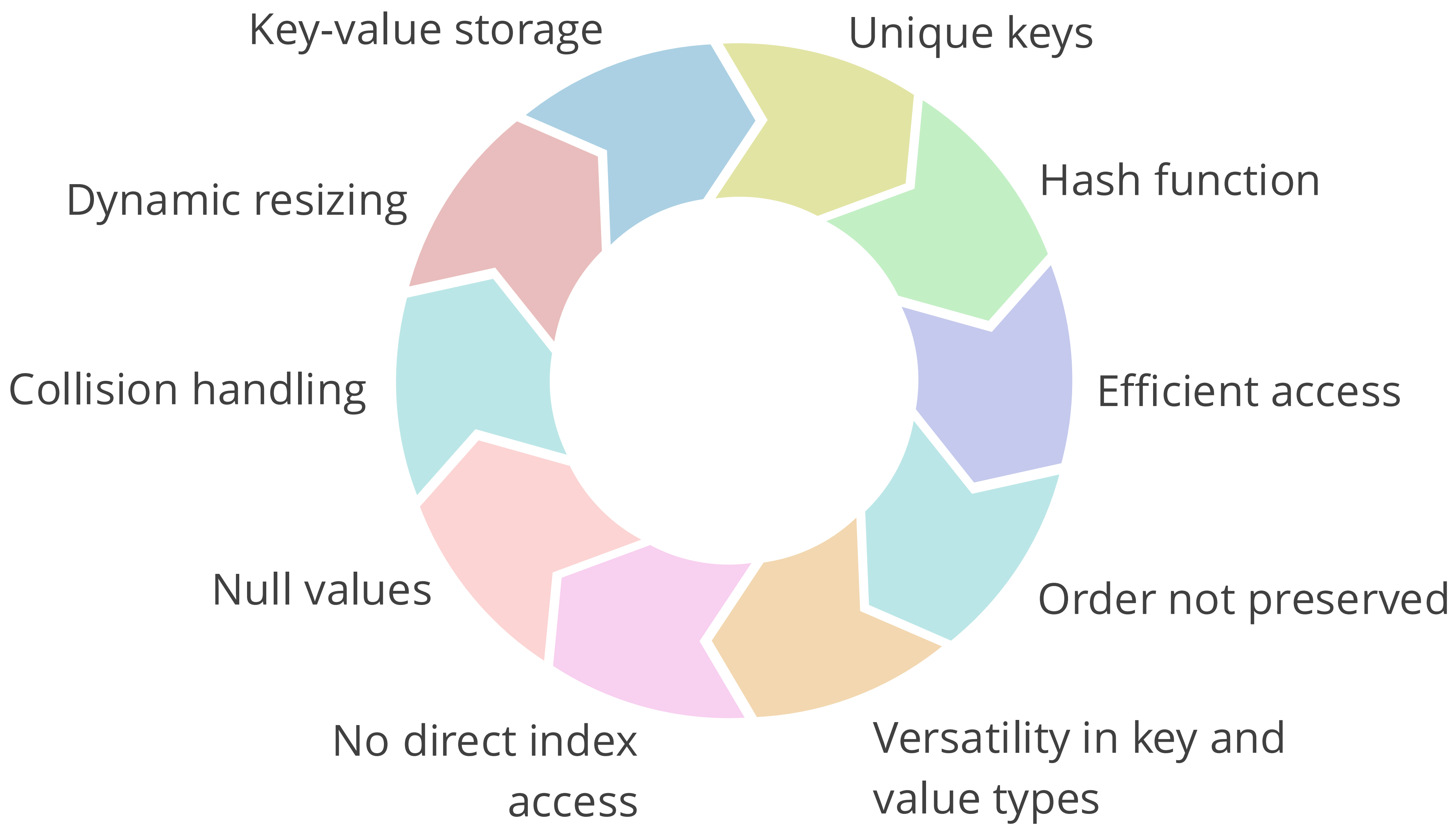
What Is a HashMap?

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 characterized by the following key characteristics:

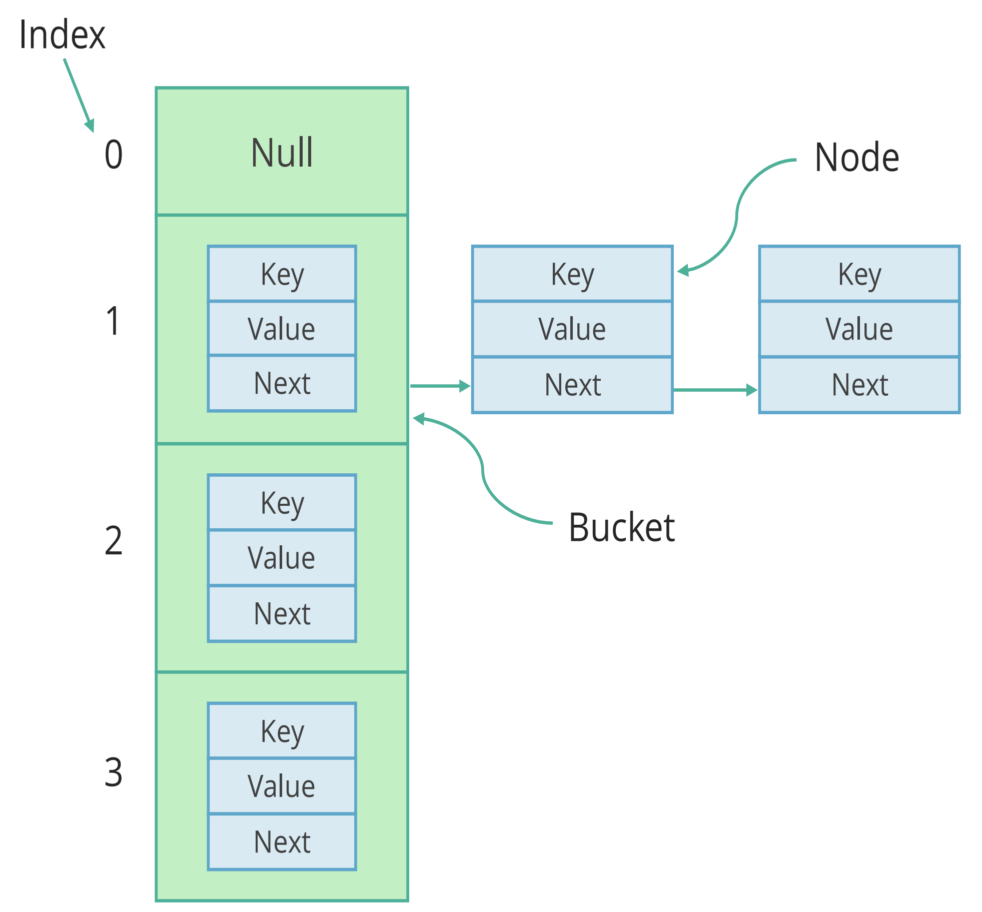


How Does HashMap Work?

How Does HashMap Work?

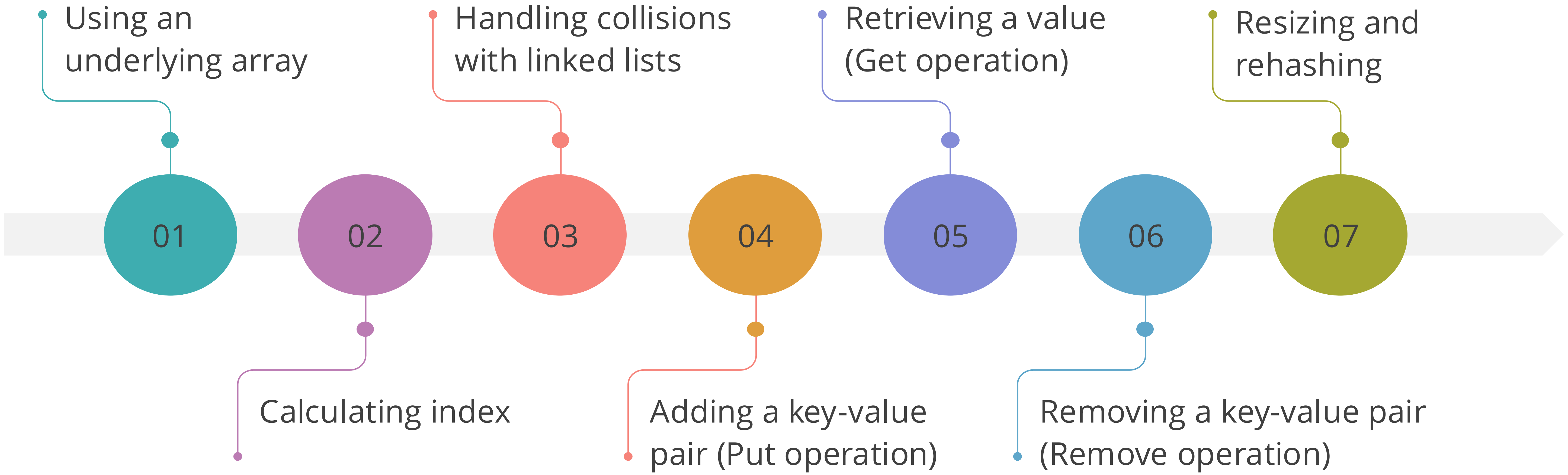
It works by storing data in a format that allows for quick retrieval, insertion, and deletion of key-value pairs.

Hash Map



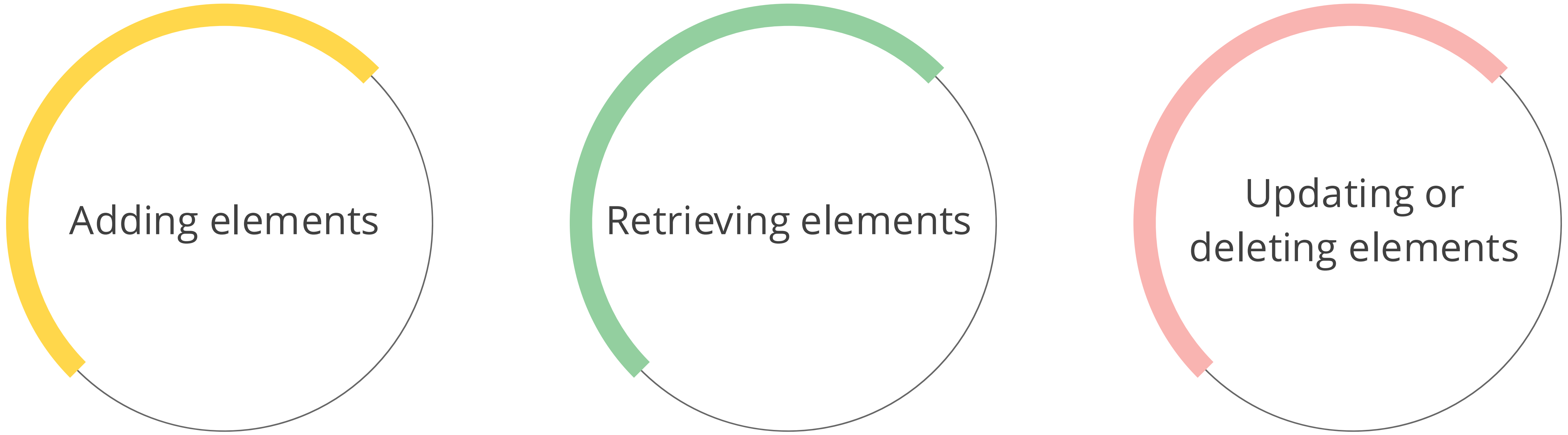
Implementing HashMaps

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


01
Using an underlying array
02
Calculating index
03
Handling collisions with linked lists
04
Adding a key-value pair (Put operation)
05
Retrieving a value (Get operation)
06
Removing a key-value pair (Remove operation)
07
Resizing and rehashing

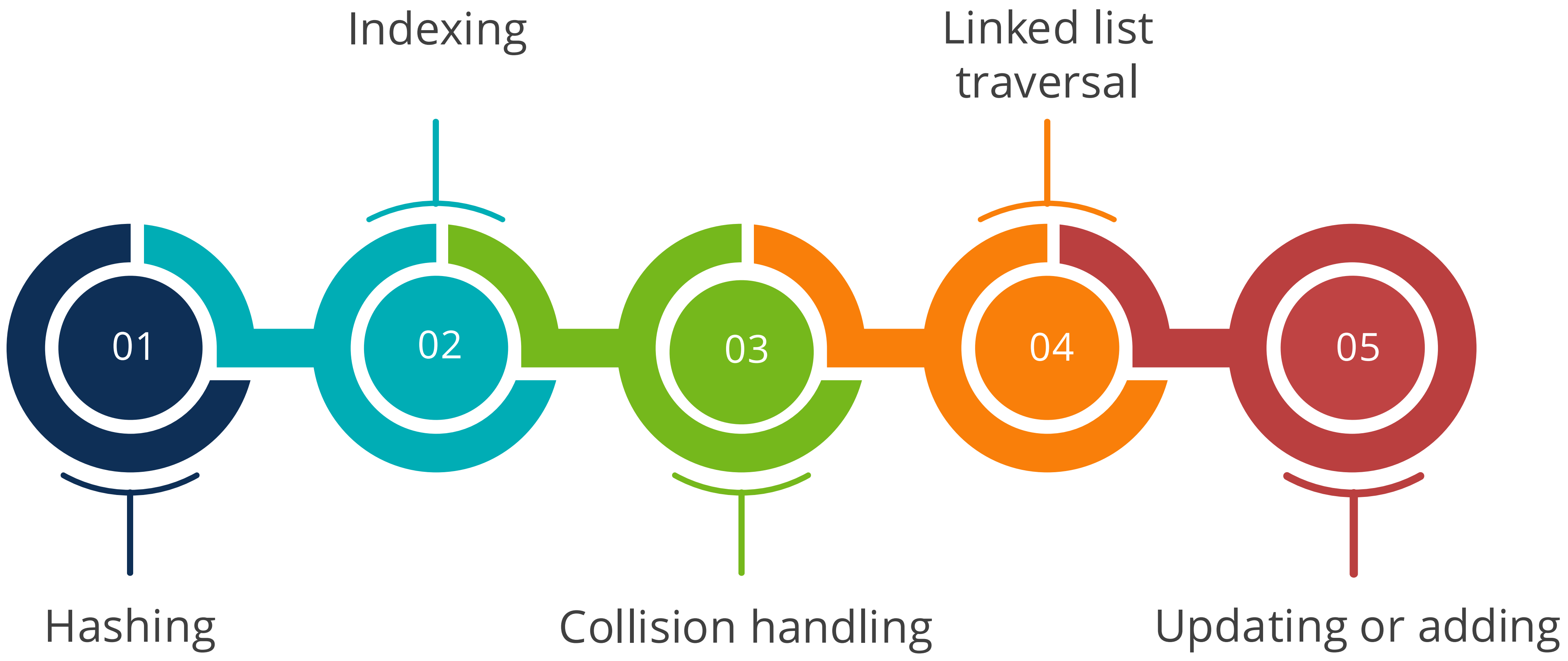
Operations on HashMap

A HashMap helps perform the following three essential operations:



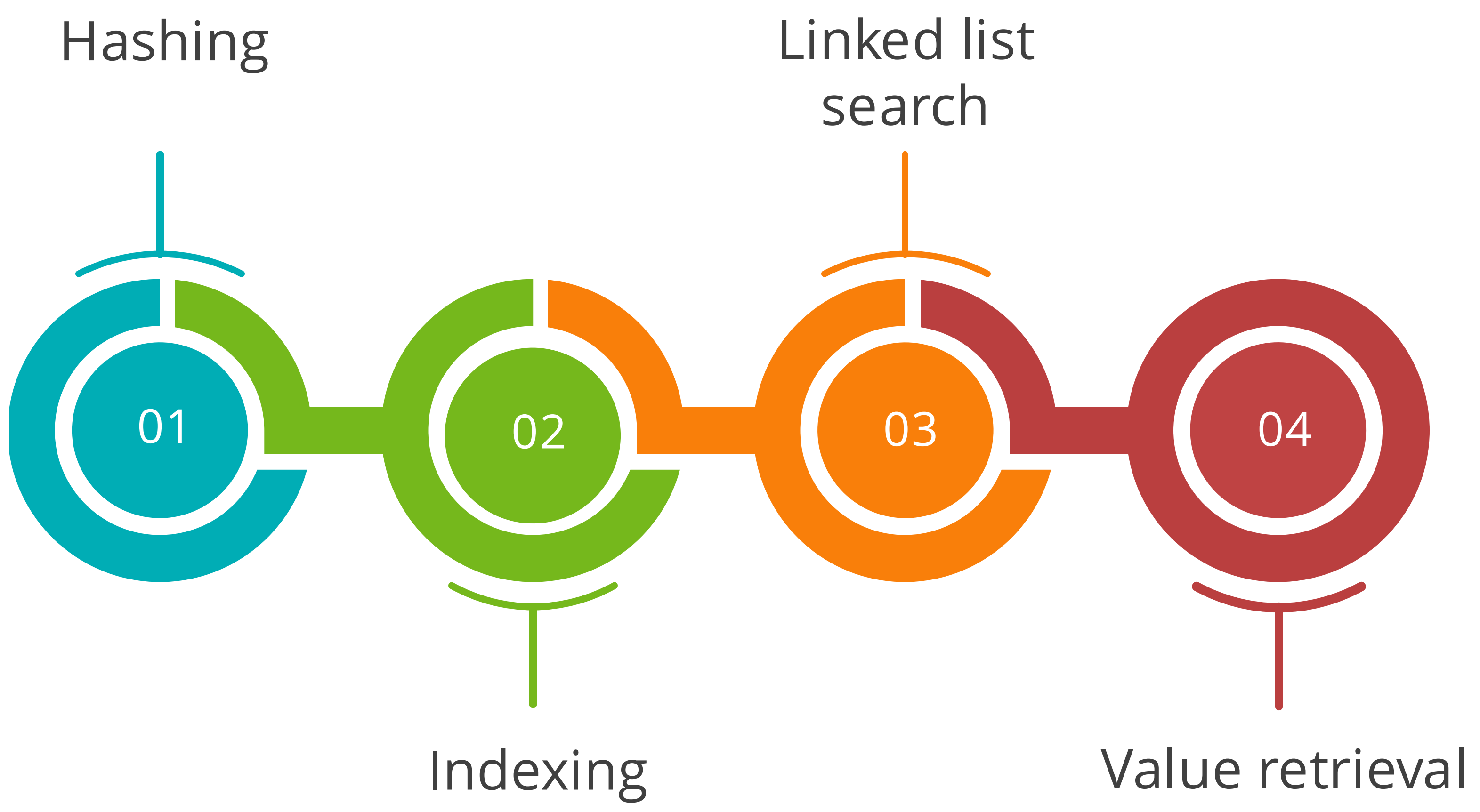
Adding Elements

Implementation using arrays and linked lists involves several steps for placing and locating key-value pairs. The key aspects involved in adding elements (Put operation) are:



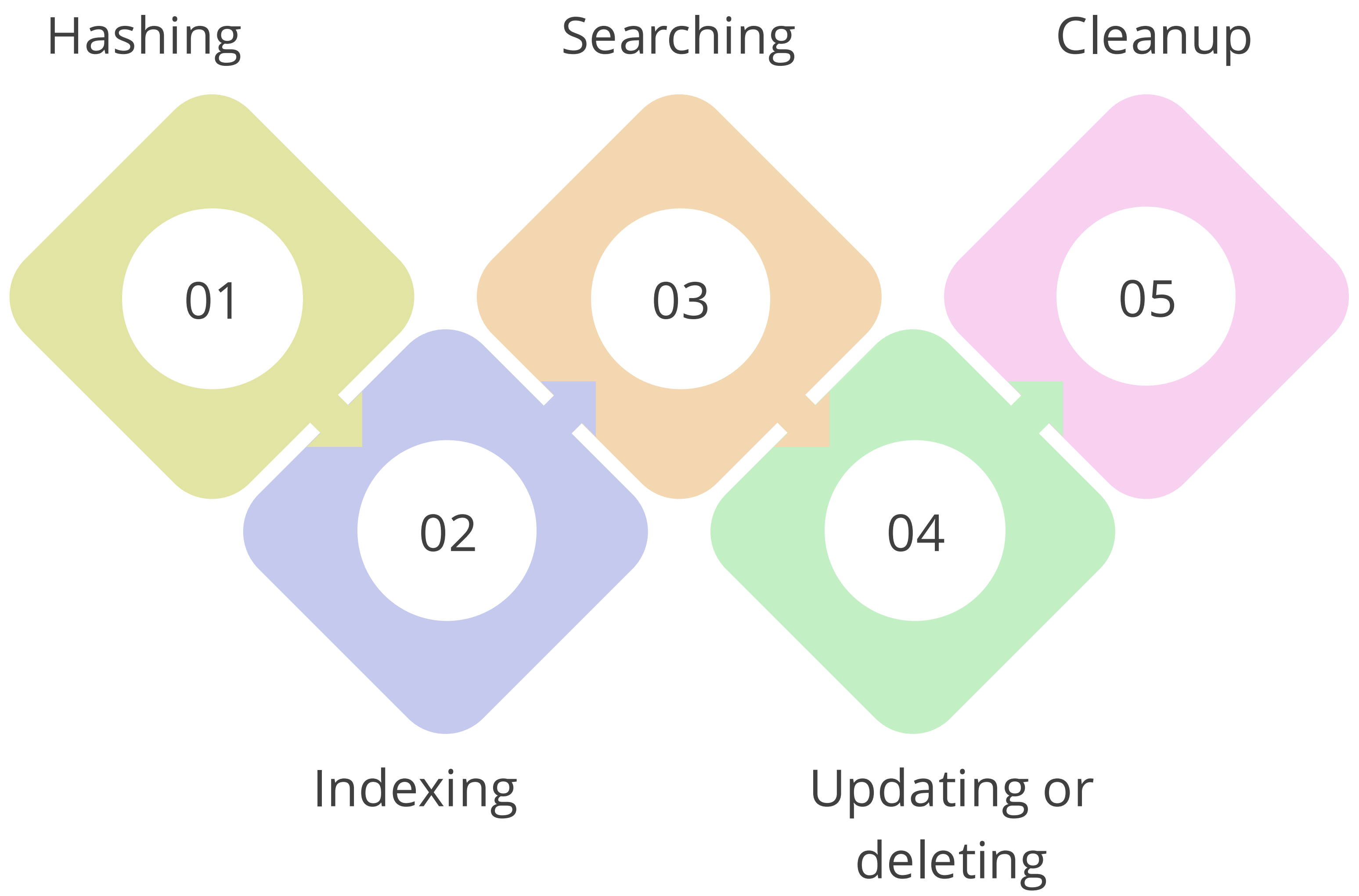
Retrieving Elements

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



Updating or Deleting Elements

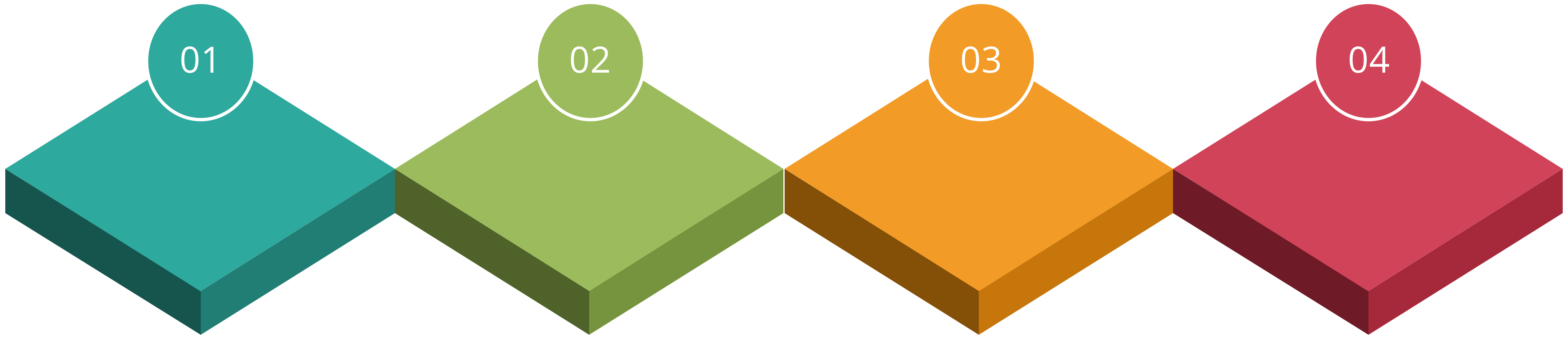
The following methods are used to update or delete an element in a HashMap by changing the value of a key-value pair:



Iterating over HashMaps

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

Here is a general description of the process:



Traverse the array

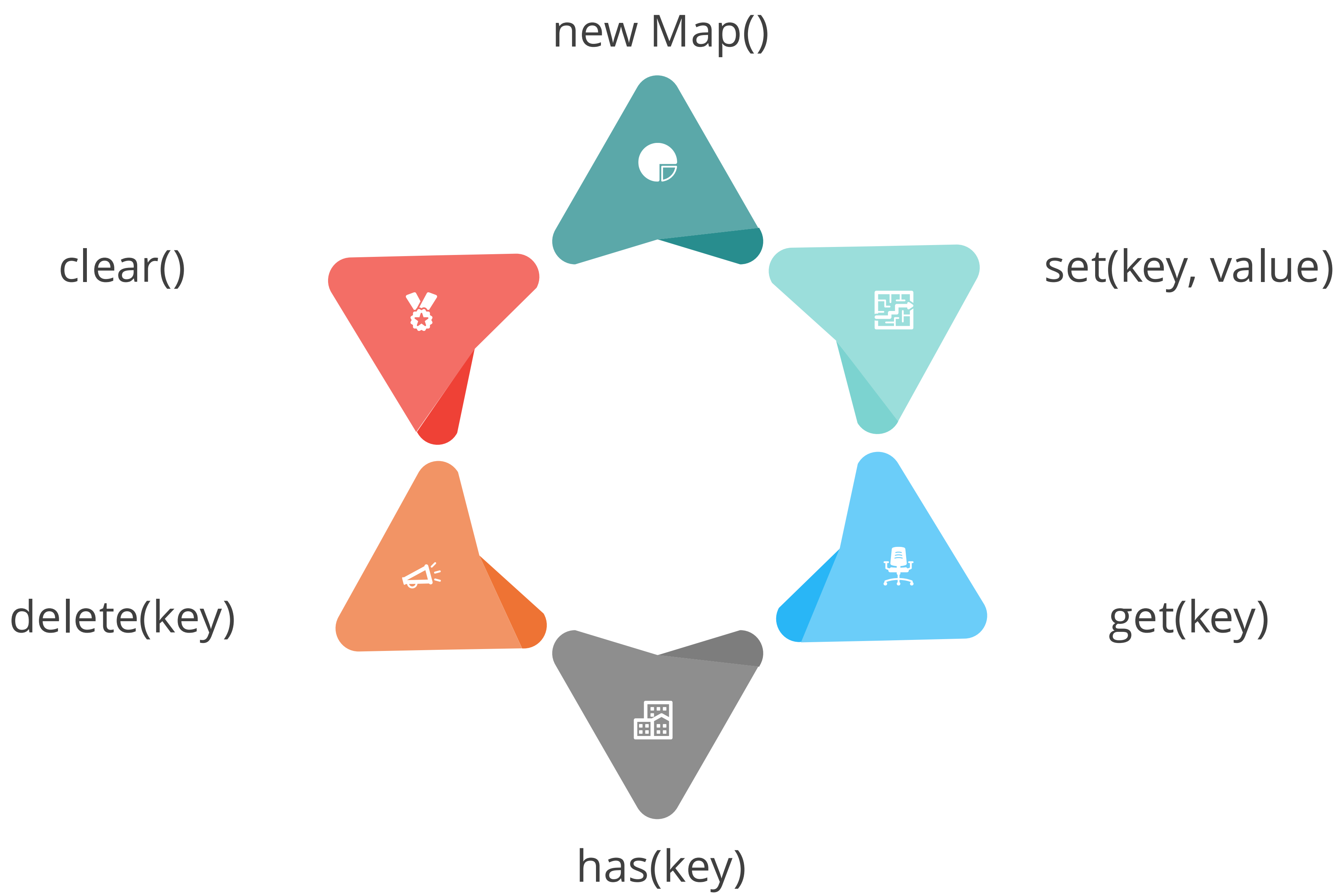
Traverse each linked list

Access key-value pairs

Move to the next index

Methods for HashMaps

Languages like Java and JavaScript offer diverse operations and methods for efficient data handling in HashMaps. The following are some common operations and methods:



Methods for HashMaps

size
keys()
values()
entries()
forEach (callbackFn[, thisArg])
replace(Key, Value)

Quick Check

An e-commerce platform stores product details using a HashMap, where each product ID (key) maps to product information (value). The system needs to quickly retrieve, update, and delete product details. Which feature of HashMaps makes them ideal for this use case?

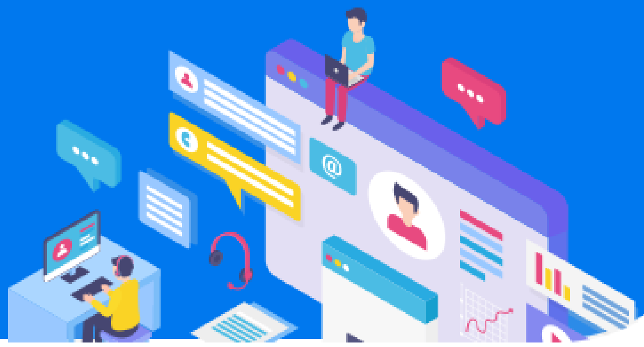
A. Direct index access

B. Key-value storage with fast lookups

C. Sorting elements in order

D. Sequential searching of keys

Assisted Practice



Declaring and Initializing HashMap

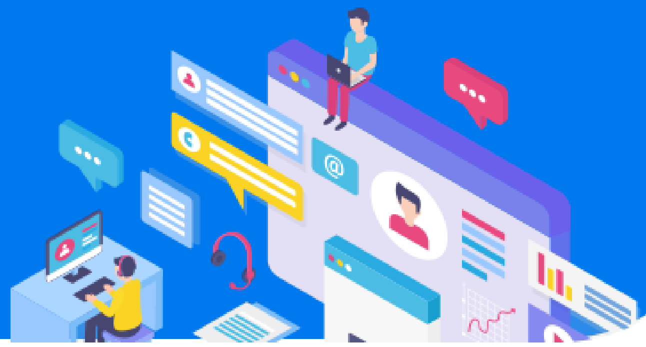
Duration: 20 Min.

Problem statement:
You have been assigned a task to demonstrate HashMap operations in JavaScript by declaring, initializing, and using key-value pairs to store and retrieve data efficiently for dynamic data access.

Outcome:
By the end of this demo, you will be able to create and manipulate a HashMap in JavaScript to store and retrieve data efficiently for dynamic data access, improving your ability to efficiently manage key-value pair data structures.

Note: Refer to the demo document for detailed steps:
06_Declaring_and_Initializing_HashMap

Assisted Practice: Guidelines



Steps to be followed:

1. Create a JavaScript file and execute it

Assisted Practice

Working with a HashMap

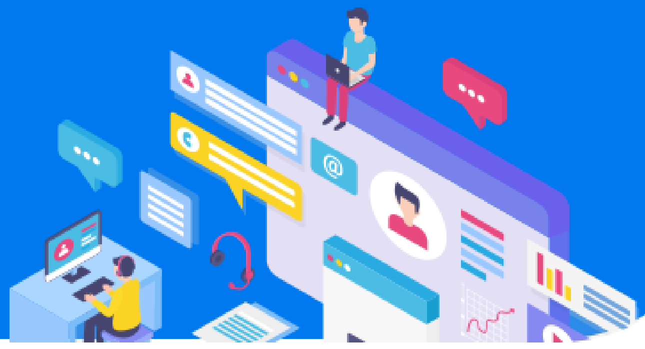
Duration: 15 Min.

Problem statement:
You have been assigned a task to demonstrate the HashMap functionality in JavaScript by building a key-value store, performing dynamic updates through add and delete operations, and managing map data using clear and display methods for efficient data handling.

Outcome:
By the end of this demo, you will be able to implement HashMap operations in JavaScript, including adding, deleting, clearing, and iterating elements. This example demonstrates declaring, initializing, accessing values, and checking key existence for efficient data handling.

Note: Refer to the demo document for detailed steps:
07_Working_with_a_HashMap

Assisted Practice: Guidelines



Steps to be followed:

1. Create a JavaScript file and execute it

Key Takeaways

- A 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.
- A 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 of representing a graph.
- HashMaps, typically using arrays and linked lists, efficiently handle data operations, such as addition, retrieval, update, and deletion, while managing collisions.

Thank You