**Module 4: Trees and Graphs**

**Lesson 1: Introduction to Trees**

**1. What Is a Tree?**

A **Tree** is a **non-linear data structure** in which elements (called **nodes**) are connected through **edges** in a hierarchical manner.  
Unlike arrays, stacks, or queues—which are linear—trees represent relationships that naturally form hierarchies (like family trees or organizational charts).

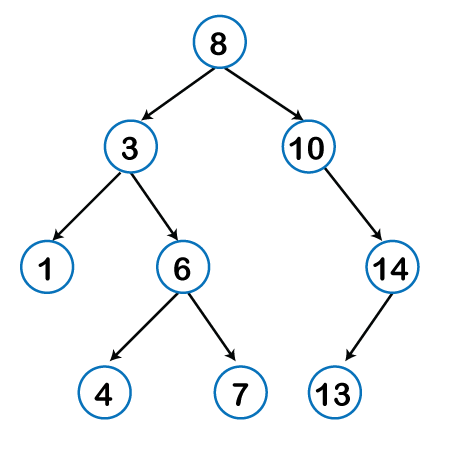
Example of a simple tree:

Figure. *TREE 1*

**2. Compo**

**nents of a Tree**

A tree is composed of **nodes** and **edges**:

* **Node:** Represents data or an entity (e.g., a person, file, or value).
* **Edge:** Represents the connection or relationship between two nodes.

**3. Real-Life Examples of Trees**

* **Family Trees:** Show parent–child relationships.
* **File Explorers:** Organize files and folders hierarchically.
* **Databases:** Use trees for indexing (like B-trees).
* **Domain Name Servers (DNS):** Use tree structures to resolve website names.
* **HTML DOM (Document Object Model):** The structure of a webpage is a tree.

**4. Types of Nodes**

| **Type of Node** | **Description** | **Example** |
| --- | --- | --- |
| **Root Node** | The topmost node; it has no parent. | 8 |
| **Leaf Node** | Nodes with no children (bottom of the tree). | 4 , 7 ,13 |
| **Branch Node** | Nodes that have both incoming and outgoing edges (middle nodes). | 3, 6, 10, 14 |
| **Parent Node** | A node that has children. | 8, 3, 6, 10, 14 |
| **Child Node** | A node that descends from a parent node. | All except 8 |
| **Siblings** | Nodes that share the same parent. | 3 and 10, 1 and 6, 4 and 7 |
| **Subtree** | A smaller tree contained within a larger tree. | Family of 3, family of 6 |

**Depth of a Node**

The **depth** of a node is the **number of edges from the root to that node**.

|  |  |  |
| --- | --- | --- |
| NODE | RELATION | DEPTH |
| 8 | *ROOT NODE/MAIN PARENT* | 0 |
| 3, 10 | CHILD/PARENT | 1 |
| 1,6,14 | CHILD/PARENT | 2 |
| 4, 7, 13 | CHILD | 3 |

**Height of a Node**

The **height** of a node is the **number of edges from that node down to the farthest leaf node**.

|  |  |  |
| --- | --- | --- |
| NODE | RELATION | HEIGHT |
| 8 | *ROOT NODE/MAIN PARENT* | 3 |
| 3, 10 | CHILD/PARENT | 2 |
| 1,6,14 | CHILD/PARENT | 1 |
| 4, 7, 13 | CHILD | 0 |

**Note:**

* The **height of a tree** is the height of the root node.
* The **depth** starts from the root (top-down).
* The **height** starts from the leaves (bottom-up).

**TRAVERSALS**

**Inorder Traversal (Left → Root → Right)**

**Process**: Visit left subtree → root → right subtree.(Read ->)

**Steps**:

* Left of 8 → 3
* Left of 3 → 1 → print **1**
* Back to 3 → print **3**
* Right of 3 → 6
* Left of 6 → 4 → print **4**
* Back to 6 → print **6**
* Right of 6 → 7 → print **7**
* Back to 8 → print **8**
* Right of 8 → 10
* Right of 10 → 14
* Left of 14 → 13 → print **13**
* Back to 14 → print **14**

**Output**: 1 3 4 6 7 8 10 13 14

**Preorder Traversal (Root → Left → Right)**

**Process**: Visit root → left subtree → right subtree.(Parent First)

**Steps**:

* Root → **8**
* Left → **3**
* Left of 3 → **1**
* Back to 3 → right → **6**
* Left of 6 → **4**
* Back to 6 → right → **7**
* Back to 8 → right → **10**
* Right of 10 → **14**
* Left of 14 → **13**

**Output**: 8 3 1 6 4 7 10 14 13

**Postorder Traversal (Left → Right → Root)**

**Process**: Visit left subtree → right subtree → root.(the CHILD is important to check first)

**Steps**:

* Left of 8 → 3
* Left of 3 → 1 → print **1**
* Right of 3 → 6
* Left of 6 → **4**
* Right of 6 → **7**
* Back to 6 → print **6**
* Back to 3 → print **3**
* Right of 8 → 10
* Right of 10 → 14
* Left of 14 → **13**
* Back to 14 → print **14**
* Back to 10 → print **10**
* Finally back to root 8 → print **8**

**Output**: 1 4 7 6 3 13 14 10 8

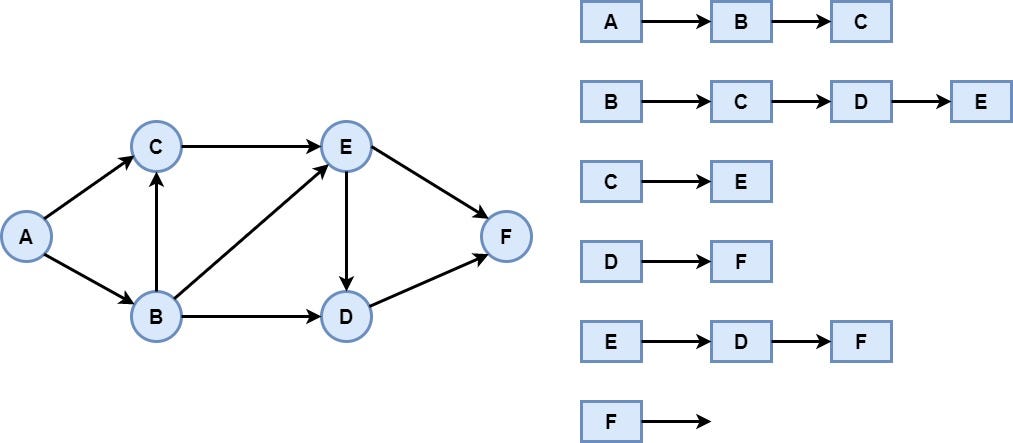
**Graphs**

**What is a Graph?**

A **graph** is like a map of **connections**. It’s not just a line (like an array or linked list), and it’s not strictly top-to-bottom like a tree. Instead, a graph is a **network of nodes and edges**:

* **Node (Vertex):** A point that holds some piece of data.
* **Edge:** A link (connection) between two nodes.

Think of graphs as the ultimate tool to represent **relationships**.



* Each circle (A, B, C, D, E, F) is a **node**.
* Each arrow is an **edge** (a directed connection).

**Graphs are EVERYWHERE:**

* **Social Networks (Undirected Graphs):**
  + Each person = a node.
  + A friendship = an edge.
  + If A and B are friends, you can say A ↔ B.
* **Navigation / Maps (Directed Graphs):**
  + Each intersection = a node.
  + Each road = an edge.
  + Some roads are **one-way** (directed).
* **Web Pages and Links:**
  + Each web page = a node.
  + A hyperlink from one page to another = directed edge.

👉 Your graph (A → F) could be interpreted as a **city map**:

* A = home
* B = coffee shop
* C = school
* D = office
* E = mall
* F = airport

The arrows = the allowed roads you can drive on.

**Directed vs Undirected**

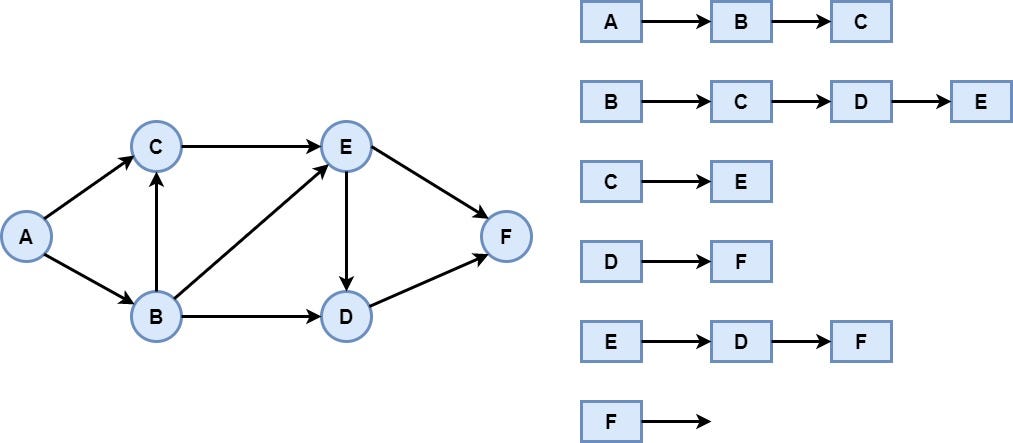
In your graph:

* All edges have arrows → **directed graph**.
* Example: A → B means you can go from **A to B**, but NOT necessarily back from B → A.

If this were **undirected**, the arrows would be two-sided, meaning roads work both ways.

**Adjacency (Who’s Connected?)**

build the **adjacency list** for your graph:



This means:

* A is connected to B and C.
* B is connected to C, D, and E.
* C is connected to E.
* E is connected to D and F.
* D only connects to F.
* F has no outgoing edges.

Adjacency basically answers: “Who are my neighbors?”

**Graph Representations**

**a) Adjacency Matrix**

For nodes A–F, we’d make a **6×6 table**.

Row = starting node, Column = destination node.  
1 = edge exists, 0 = no edge.

A B C D E F

A [ 0 1 1 0 0 0 ]

B [ 0 0 1 1 1 0 ]

C [ 0 0 0 0 1 0 ]

D [ 0 0 0 0 0 1 ]

E [ 0 0 0 1 0 1 ]

F [ 0 0 0 0 0 0 ]

Example:

* Row A, Column B = 1 → A → B exists.
* Row C, Column E = 1 → C → E exists.

**Graph Traversals**

Now, how do we **walk** this graph?  
We have two big strategies:

**🔸 Depth-First Search (DFS)**

Go as deep as possible before backtracking.

DFS from A:

1. Start at A → mark as visited.
2. Go to first neighbor B.
3. From B → go to C.
4. From C → go to E.
5. From E → go to D.
6. From D → go to F.
7. Backtrack until all neighbors are visited.

Order:

[ A → B → C → E → D → F ]

**🔸 Breadth-First Search (BFS)**

Go level by level using a queue.

BFS from A:

1. Start at A.
2. Visit neighbors → B, C.
3. Then visit neighbors of B → C, D, E (C already visited, skip).
4. Then neighbors of C → E (already visited, skip).
5. Then neighbors of D → F.
6. Then neighbors of E → D, F (already visited, skip).

Order:

[ A → B → C → D → E → F ]

**Real-World Application of This Graph**

Let’s say this graph is a **travel map**:

* A = your home.
* F = the airport.

Now:

* **DFS (A → B → C → E → D → F):** means you explore every possible road deeply before finally reaching the airport.
* **BFS (A → B → C → D → E → F):** means you try the shortest possible routes first, level by level, so you’ll reach the airport in fewer steps.

This is why BFS is often used in **shortest path problems** (like GPS apps).