DATA STRUCTURE ALGORITHMS AND APPLICATIONS (CT- 159)

Lecture – 8
Binary Tree, Types and Properties

Instructor: Engr. Nasr Kamal

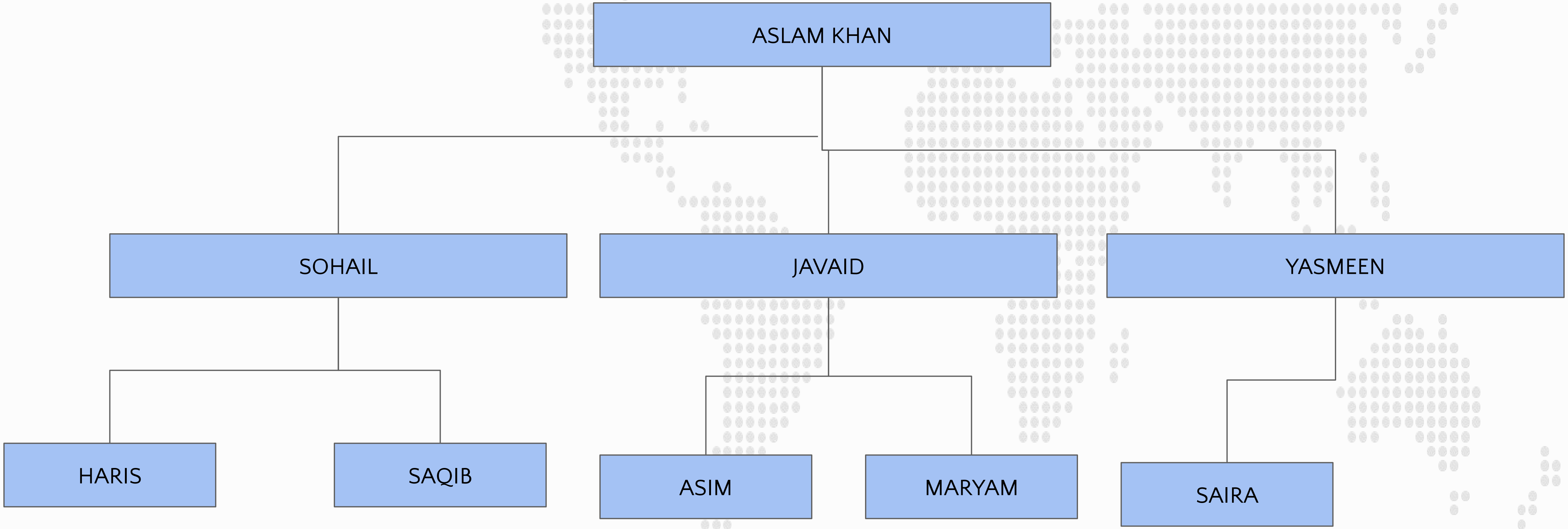
Department of Computer Science and Information Technology

Non-Linear Data Structure

- The data structures that we have discussed in previous lectures are linear data structures.
- The arrays, linked list, queue and stack are linear data structures.
- There are a number of applications where linear data structures are not appropriate. In such cases, there is need of some non-linear data structure.
- **Tree is one of the non-linear data structures.**

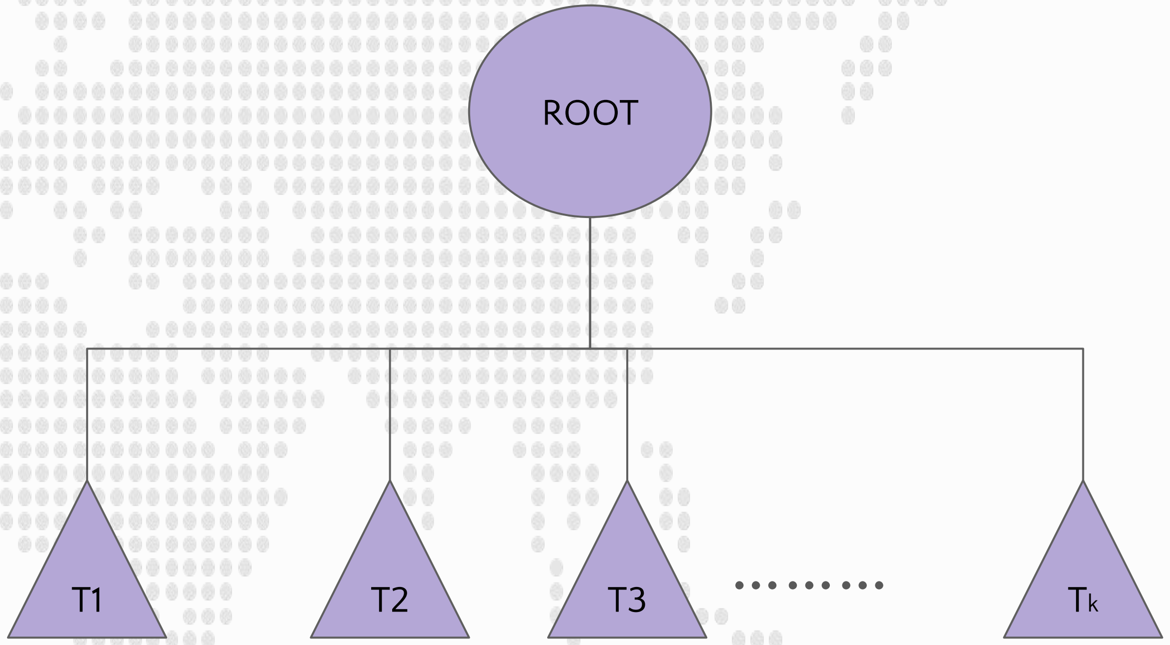
TREES

This figure is showing a genealogy tree of a family.

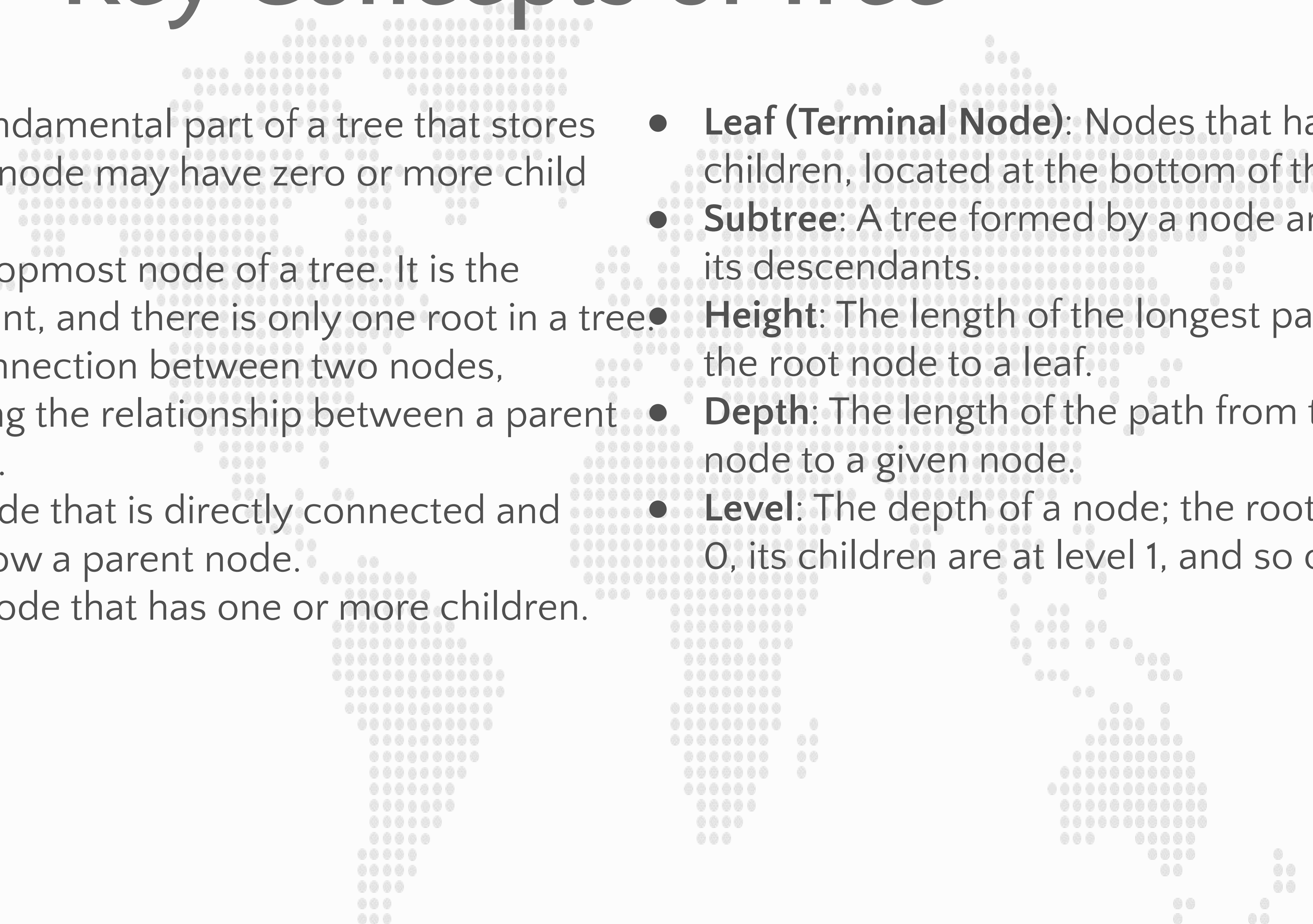


Tree

- In data structures, a **tree** is a hierarchical structure that represents a collection of elements called *nodes*, where each node is connected to one or more child nodes.
- The topmost node is known as the **root**, and every other node is connected via edges that define parent-child relationships.



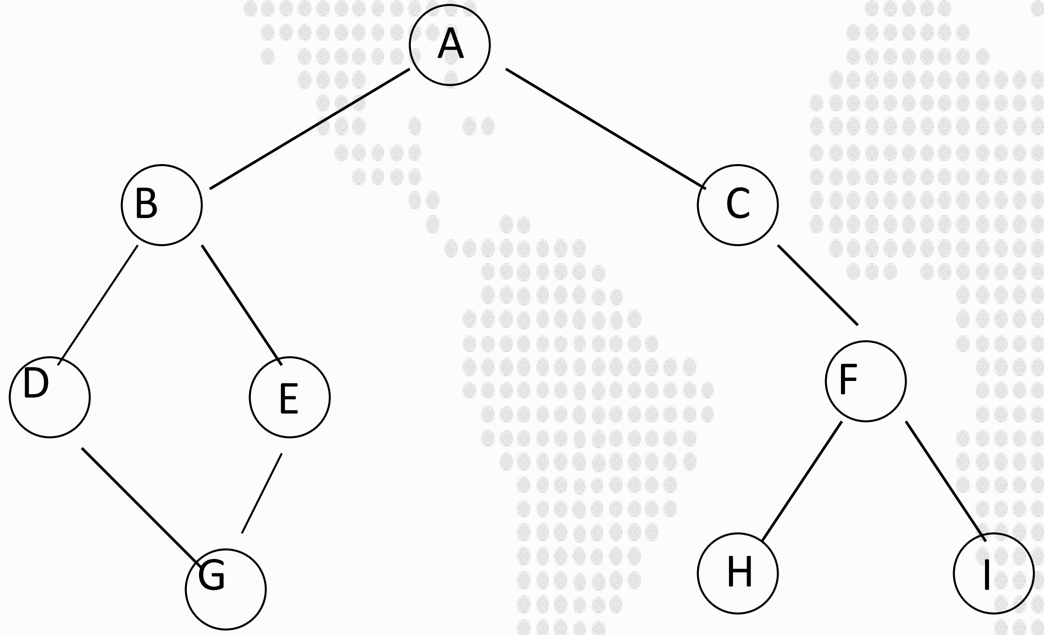
Key Concepts of Tree



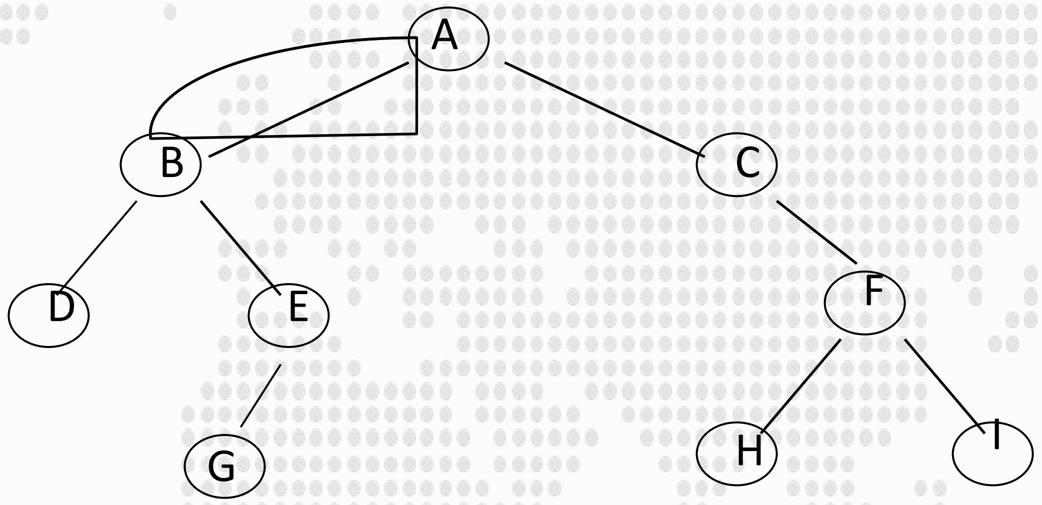
- **Node**: A fundamental part of a tree that stores data. Each node may have zero or more child nodes.
- **Root**: The topmost node of a tree. It is the starting point, and there is only one root in a tree.
- **Edge**: A connection between two nodes, representing the relationship between a parent and a child.
- **Child**: A node that is directly connected and comes below a parent node.
- **Parent**: A node that has one or more children.
- **Leaf (Terminal Node)**: Nodes that have no children, located at the bottom of the tree.
- **Subtree**: A tree formed by a node and all of its descendants.
- **Height**: The length of the longest path from the root node to a leaf.
- **Depth**: The length of the path from the root node to a given node.
- **Level**: The depth of a node; the root is at leve
  0, its children are at level 1, and so on.

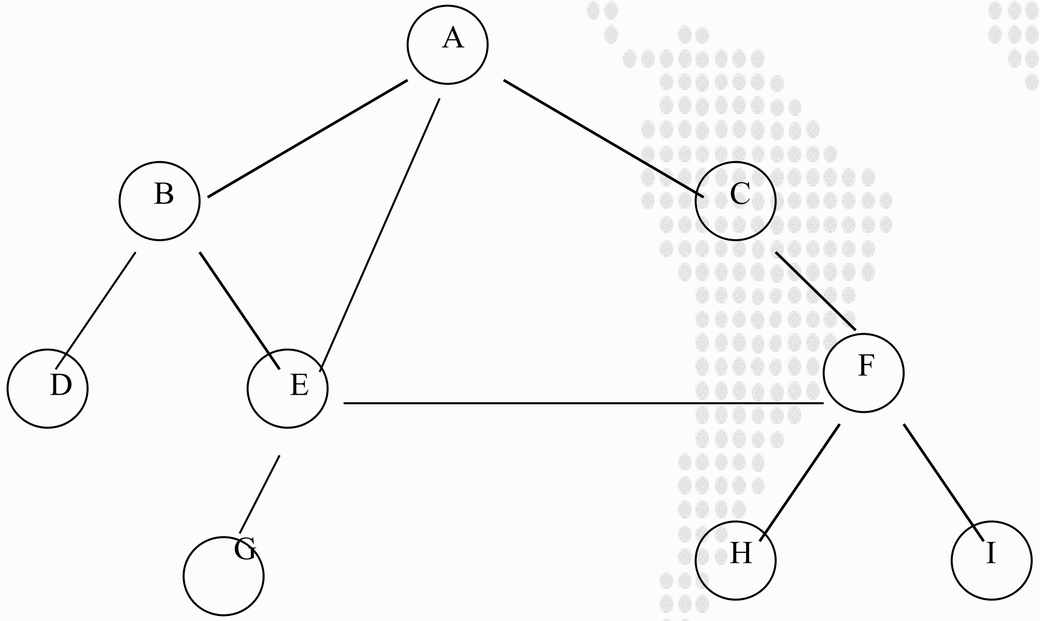
A Non-Tree Structure

In this figure we join the node G with D. Now this is not a tree. It is due to the reason that in a tree there is always one path to go (from root) to a node. But in this figure, there are two paths (tracks) to go to node G.



A Non-Tree Structure





Types of Trees

- **Binary Tree**: Each node has at most two children (left and right).
- **Binary Search Tree (BST)**: A binary tree where the left child contains values less than the parent node, and the right child contains values greater than the parent.
- **AVL Tree**: A self-balancing binary search tree where the heights of the left and right subtrees differ by at most one.
- **Heap**: A binary tree where the parent node is always greater (max-heap) or smaller (min-heap) than its child nodes.
- **Trie**: A tree-like data structure used to store a dynamic set of strings, often for autocomplete purposes.

Binary Tree

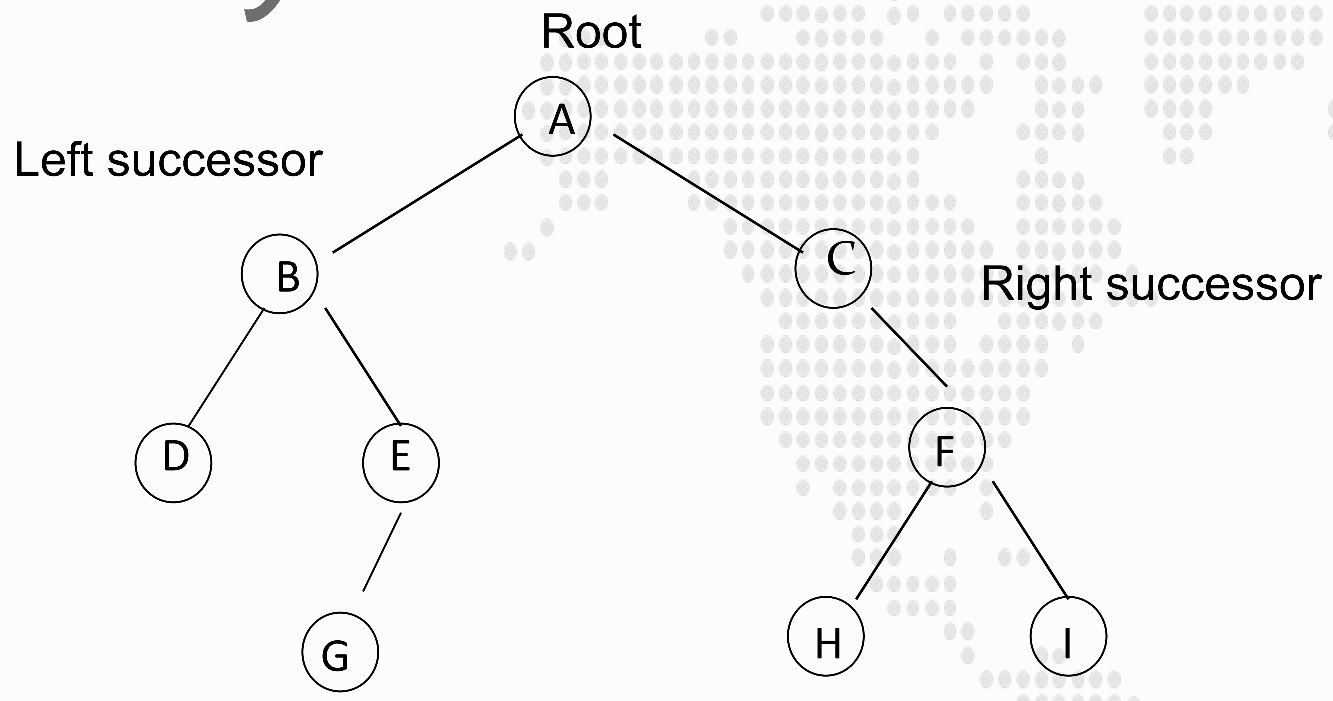
A binary tree is a data structure in which each node has at most two children, referred to as the left child and the right child.

```
    A
   / \
  B   C
 / \   \
D   E   F
```

Key Characteristics of a Binary Tree:

- **Node**: The element containing data.
- **Left Child**: The left branch connected to a node.
- **Right Child**: The right branch connected to a node.
- **Root**: The topmost node from which all nodes branch out.
- **Leaf**: A node with no children.
- **Height**: The number of edges on the longest path from the node to a leaf.

Binary Tree



- The node A, B, C and F have two successors, the node E have only one successor, and the node D,G,H, and I have no successor.
- The left sub-tree of root A consists of the nodes B,D,E, and G, and the right subtree of A consists of the node C,F,H, and I.

Structure of a Binary Tree:

In a binary tree, nodes are arranged such that:

- Each node can have **0, 1, or 2** children.
- The **left child** stores values or references to elements less than the parent (in the case of a binary search tree).
- The **right child** stores values or references to elements greater than the parent (in the case of a binary search tree).

Types of Binary Tree

- **Full Binary Tree**:
  - A **full binary tree** is a tree in which every node has either **0 or 2 children**—no nodes have only one child.
  - This means that each node is either a **leaf node** (no children) or an **internal node** with two children.

```
   A
  / \
 B   C
/   \
D   E
```

- **Complete Binary Tree**:
  - A **complete binary tree** is a tree where **all levels are fully filled**, except possibly the last level.
  - The last level must have nodes as **left as possible**.

```
   A
  / \
 B   C
/   \ /
D   E F
```

Types of Binary Tree

- **Perfect Binary Tree**:

  It is a special type of binary tree where:

  - All **internal nodes** have **exactly two children**.
  - All **leaf nodes** are at the **same level**.

```
   A
  / \
 B   C
/ \ / \
D  E F  G
```

- **Balanced Binary Tree**:
  - The **height difference between the left and right subtrees** of any node is at most **1**.
  - Examples of balanced binary trees include **AVL trees** and **Red-Black trees**.
  - They ensure that operations like insertion, deletion, and search are performed in **O(log n)** time.

```
  A
 / \
 B  C
/
D
```

Types of Binary Tree

- **Degenerate (or Pathological) Tree**:
    - A **degenerate tree** (also called a **skewed tree**) is a tree in which **each parent node has only one child**.
    - It behaves like a **linked list**, where the height of the tree is equal to the number of nodes.

Right Skewed:

```
A
 \
  B
   \
    C
```

Left Skewed:

```
    A
   /
  B
 /
C
```

Types of Binary Tree

Heaps

- **Binary Heap**: A complete binary tree that satisfies the **heap property**:
  - **Max-Heap**: Every parent node is greater than or equal to its child nodes.
  - **Min-Heap**: Every parent node is less than or equal to its child nodes.
- Binary heaps are used in implementing **priority queues** and for **heap sort**.

```
      10
     /  \
    9    5
   / \
  3   4
```

Applications of Binary Tree

- **Binary Search Trees (BSTs)**: Used for searching and sorting data efficiently.
- **Expression Trees**: Used to evaluate arithmetic expressions where leaves are operands and internal nodes are operators.
- **Decision Trees**: Commonly used in machine learning for decision-making processes.
- **Heaps**: Used in priority queues and sorting algorithms.

Binary trees provide a way to maintain a sorted hierarchy, making operations like insertion, deletion, and search relatively efficient compared to other data structures.

Type	Description	Use Cases
Full Binary Tree	Each node has 0 or 2 children.	Used when we need a well-defined structure.
Complete Binary Tree	All levels are filled except possibly the last, filled from left to right.	Suitable for heaps and priority queues.
Perfect Binary Tree	All internal nodes have 2 children; leaf nodes are at the same level.	Provides optimal depth and is ideal for balanced data.
Balanced Binary Tree	Height difference between subtrees is ≤1 for every node.	Ensures optimal time complexity for search, insert, and delete.
Degenerate Tree	Each node has only one child (like a linked list).	Considered inefficient; may need rebalancing.
Binary Search Tree	Left child < parent < right child.	Allows fast searching, insertion, and deletion.
Binary Heap	Complete binary tree with max or min properties.	Used in priority queues and sorting.

Breadth First Search BFS

BFS is an algorithm for traversing or searching through a graph or tree structure. It explores all the nodes at the present depth level before moving on to the nodes at the next depth level

How It Works:

- BFS uses a queue to keep track of the nodes to be visited.
- Start from a given node, mark it as visited, and enqueue it.
- Dequeue a node from the front of the queue, visit all its adjacent unvisited nodes, mark them as visited, and enqueue them.
- Repeat until the queue is empty.

EXAMPLE

```
    A
   / \
  B   C
 / \   \
D   E   F
```

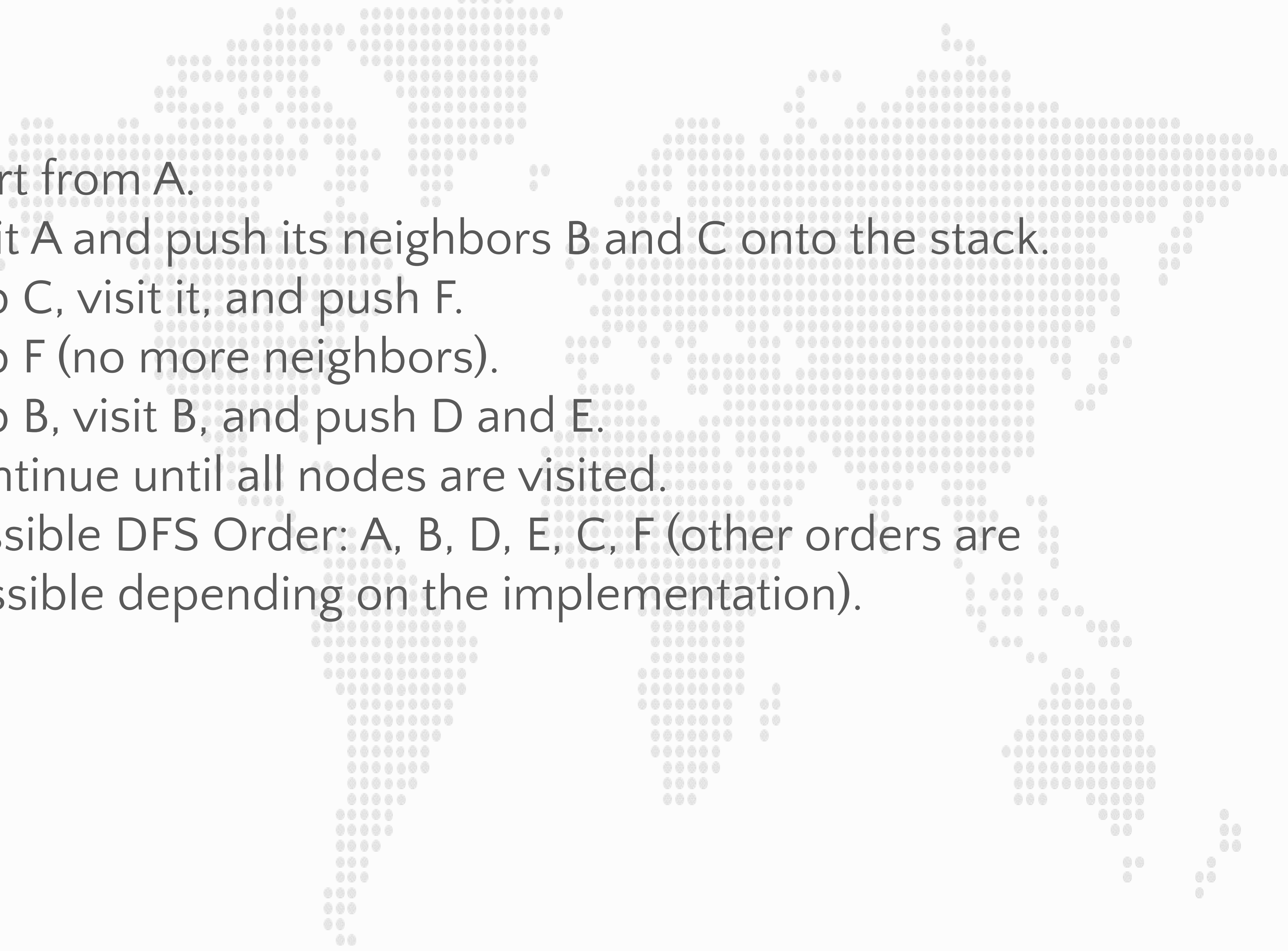
- Start from A.
- Visit A and enqueue its neighbors B and C.
- Dequeue A, visit B, and enqueue its neighbors D and E.
- Dequeue B, visit C, and enqueue its neighbor F.
- Continue until all nodes are visited.
- BFS Order: A, B, C, D, E, F

Depth First Search DFS

DFS is another algorithm for traversing or searching through a graph or tree structure. It explores as far as possible along each branch before backtracking.

How It Works:

- DFS uses a *stack* (or recursion) to keep track of the nodes to be visited.
- Start from a given node, mark it as visited, and push it onto the stack.
- Pop a node from the stack, visit an unvisited neighbor, mark it as visited, and push it onto the stack.
- Repeat until the stack is empty



EXAMPLE

```
    A
   / \
  B   C
 / \   \
D   E   F
```

- Start from A.
- Visit A and push its neighbors B and C onto the stack.
- Pop C, visit it, and push F.
- Pop F (no more neighbors).
- Pop B, visit B, and push D and E.
- Continue until all nodes are visited.
- Possible DFS Order: A, B, D, E, C, F (other orders are possible depending on the implementation).

Inorder, Preorder, Postorder Traversal

Inorder Traversal (Left, Root, Right):

- Visit the left subtree, then the root, then the right subtree.
- Order: D, B, E, A, C, F

Preorder Traversal (Root, Left, Right):

- Visit the root, then the left subtree, then the right subtree.
- Order: A, B, D, E, C, F

Postorder Traversal (Left, Right, Root):

- Visit the left subtree, then the right subtree, then the root.
- Order: D, E, B, F, C, A

Inorder, Preorder, Postorder Traversal

Inorder Traversal (Left, Root, Right):

- Visit the left subtree, then the root, then the right subtree.
- Order: D, B, E, A, C, F

Preorder Traversal (Root, Left, Right):

- Visit the root, then the left subtree, then the right subtree.
- Order: A, B, D, E, C, F

Postorder Traversal (Left, Right, Root):

- Visit the left subtree, then the right subtree, then the root.
- Order: D, E, B, F, C, A

Inorder, Preorder, Postorder Traversal

Inorder Traversal (Left, Root, Right):

- Visit the left subtree, then the root, then the right subtree.
- Order: D, B, E, A, C, F

Preorder Traversal (Root, Left, Right):

- Visit the root, then the left subtree, then the right subtree.
- Order: A, B, D, E, C, F

Postorder Traversal (Left, Right, Root):

- Visit the left subtree, then the right subtree, then the root.
- Order: D, E, B, F, C, A

Inorder, Preorder, Postorder Traversal

Given the following tree, write the traversal order for inorder, preorder, and postorder.

```
      1
     / \
    2   3
   /   / \
  4   5   6
       \
        7
```

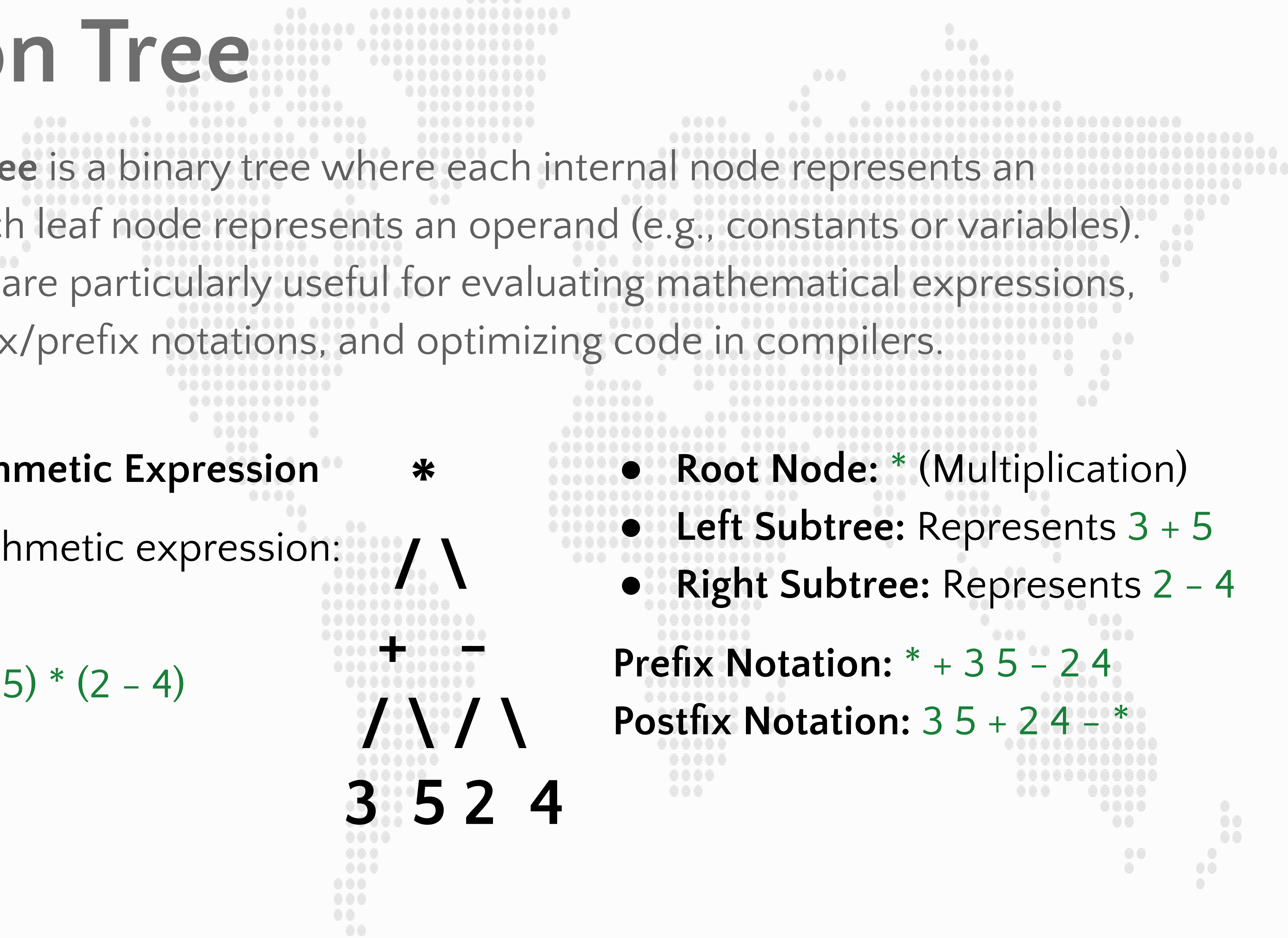
Inorder: 4, 2, 1, 5, 7, 3, 6

Preorder: 1, 2, 4, 3, 5, 7, 6

Postorder: 4, 2, 7, 5, 6, 3, 1

Expression Tree

An **Expression Tree** is a binary tree where each internal node represents an operator and each leaf node represents an operand (e.g., constants or variables). Expression trees are particularly useful for evaluating mathematical expressions, generating postfix/prefix notations, and optimizing code in compilers.



Example 1: Basic Arithmetic Expression

Let's consider the arithmetic expression:
(3 + 5) * (2 – 4)

Infix Expression: (3 + 5) * (2 – 4)

```
     *
    / \
   +   –
  /\  / \
 3  5 2  4
```

- **Root Node:** * (Multiplication)
- **Left Subtree:** Represents 3 + 5
- **Right Subtree:** Represents 2 – 4

Prefix Notation: * + 3 5 – 2 4

Postfix Notation: 3 5 + 2 4 – *

Expression Tree

Key Points About Expression Trees:

1. **Leaf Nodes** represent operands (e.g., constants or variables).
2. **Internal Nodes** represent operators (e.g., +, –, *, /, ˆ).
3. **Prefix Notation** is obtained by traversing the tree in **pre-order** (root-left-right).
4. **Postfix Notation** is obtained by traversing the tree in **post-order** (left-right-root).
5. **Infix Notation** corresponds to an **in-order traversal** with parentheses to respect the operator precedence.

Expression Tree

Example 2: Complex Expression with Multiple Operators

Consider the expression:
(a + b) * (c – d) / e

Infix Expression: (a + b) * (c – d) / e

- **Root Node:** / (Division)
- **Left Subtree:** Represents (a + b) * (c – d)
- **Right Subtree:** Represents e

Prefix Notation: / * + a b – c d e
Postfix Notation: a b + c d – * e /

```
       /
      / \
     *   e
    /\
   +  -
  / \ / \
 a  b c  d
```

Expression Tree

Example 4: Logical Expression

Consider the logical expression:
(a AND b) OR NOT c

Infix Expression: (a AND b) OR NOT c

- **Root Node:** OR
- **Left Subtree:** Represents a AND b
- **Right Subtree:** Represents NOT c

Prefix Notation: OR AND a b NOT c

Postfix Notation: a b AND c NOT OR

```
       OR
      /   \
   AND   NOT
   /  \      \
  a    b     c
```

Expression Tree

Example 5: Expression with Power

Consider the expression:

a ˆ b + c

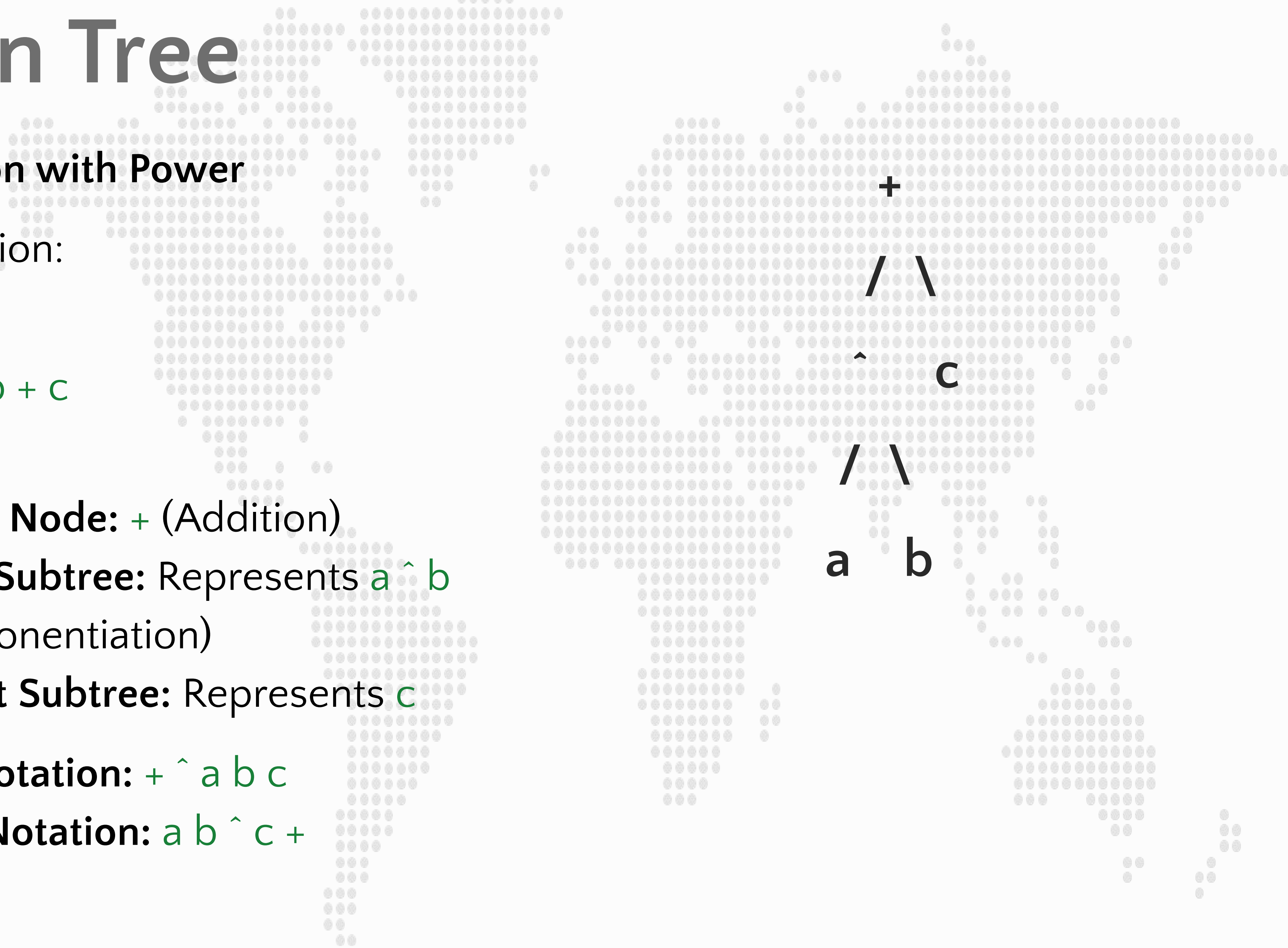
Infix Expression: a ˆ b + c

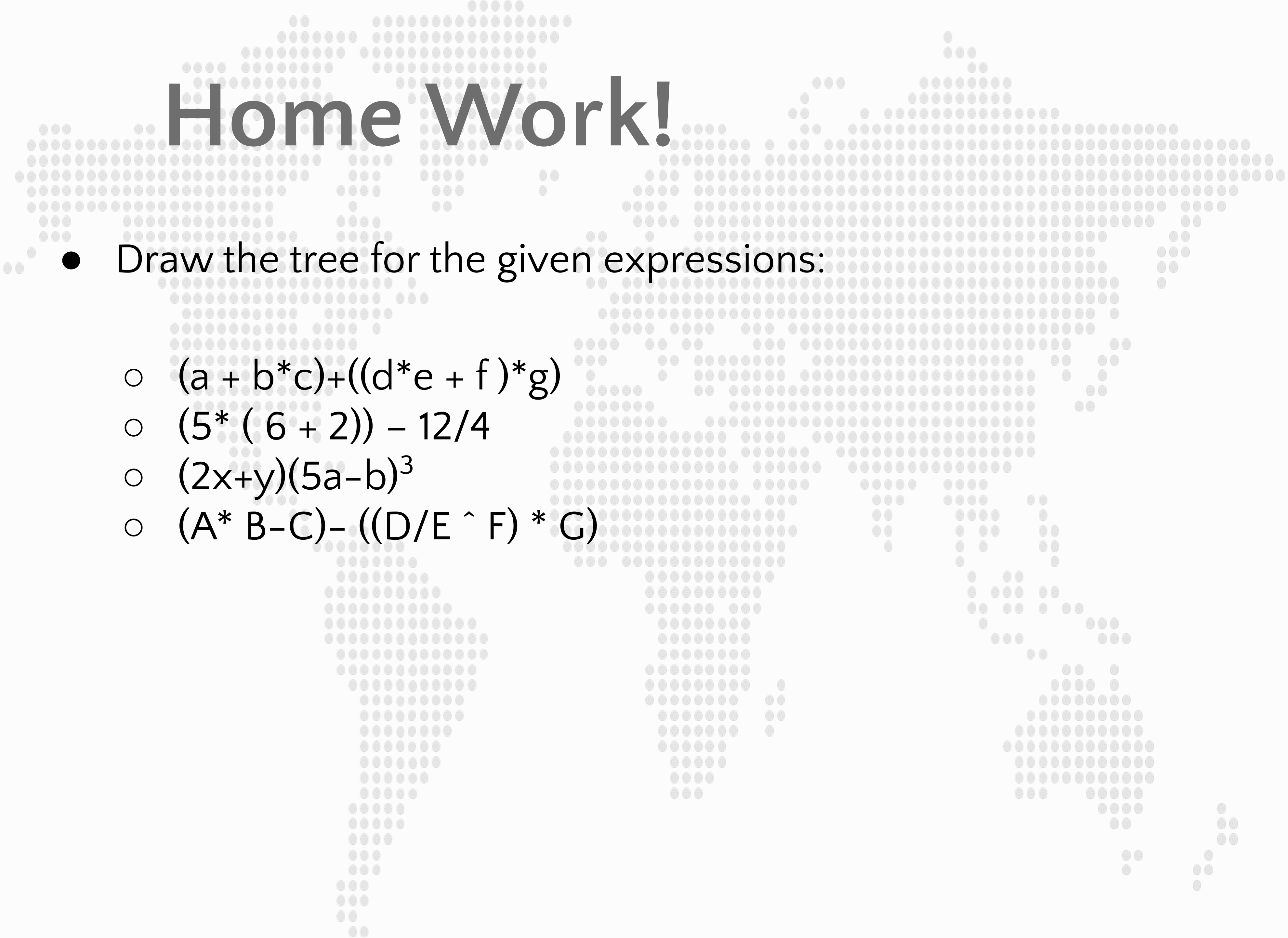
- **Root Node:** + (Addition)
- **Left Subtree:** Represents a ˆ b (Exponentiation)
- **Right Subtree:** Represents c

Prefix Notation: + ˆ a b c

Postfix Notation: a b ˆ c +

```
      +
     / \
    ˆ   c
   / \
  a   b
```





Home Work!

- Draw the tree for the given expressions:
  - (a + b*c)+((d*e + f)*g)
  - (5* (6 + 2)) – 12/4
  - $(2x+y)(5a-b)^3$
  - (A* B–C)– ((D/E ˆ F) * G)