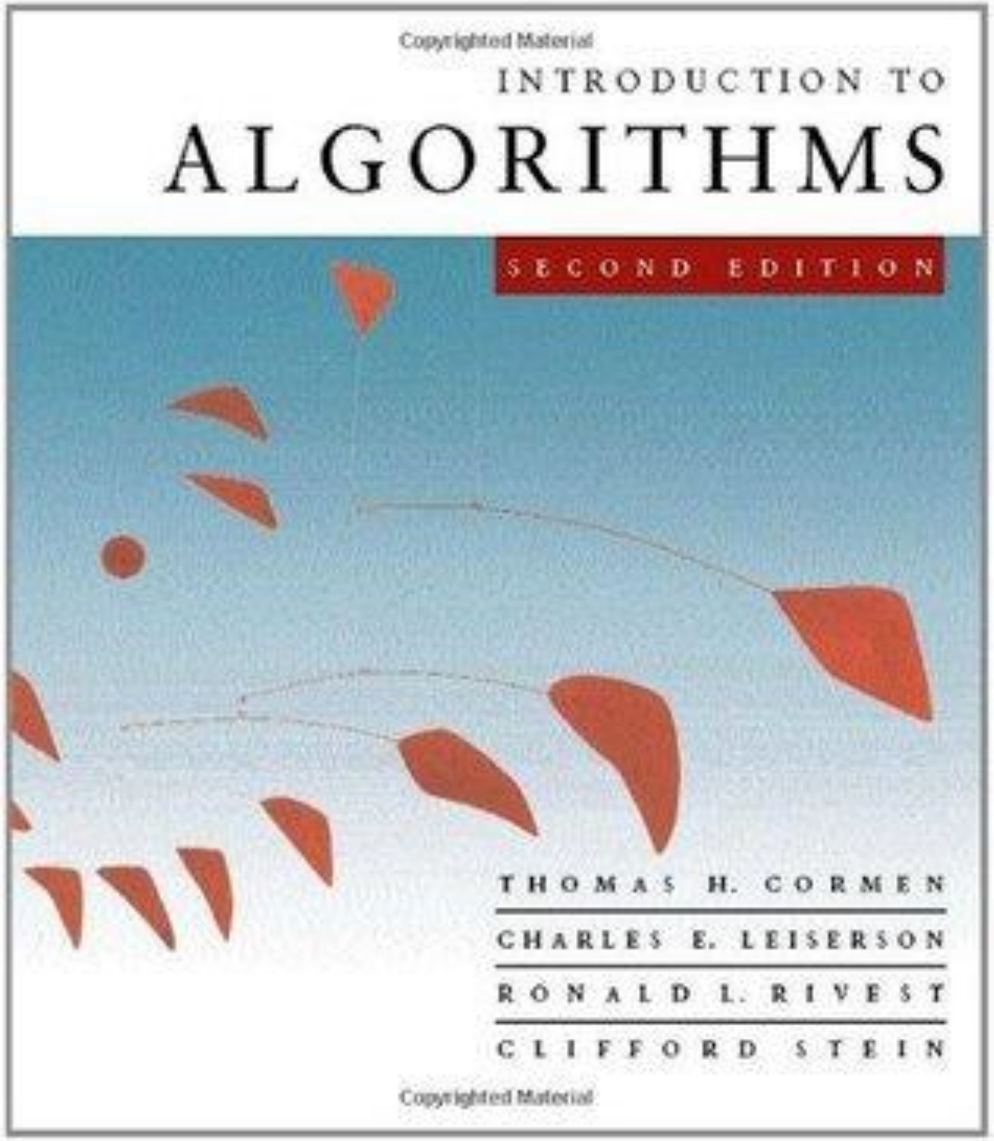
Fall 2024

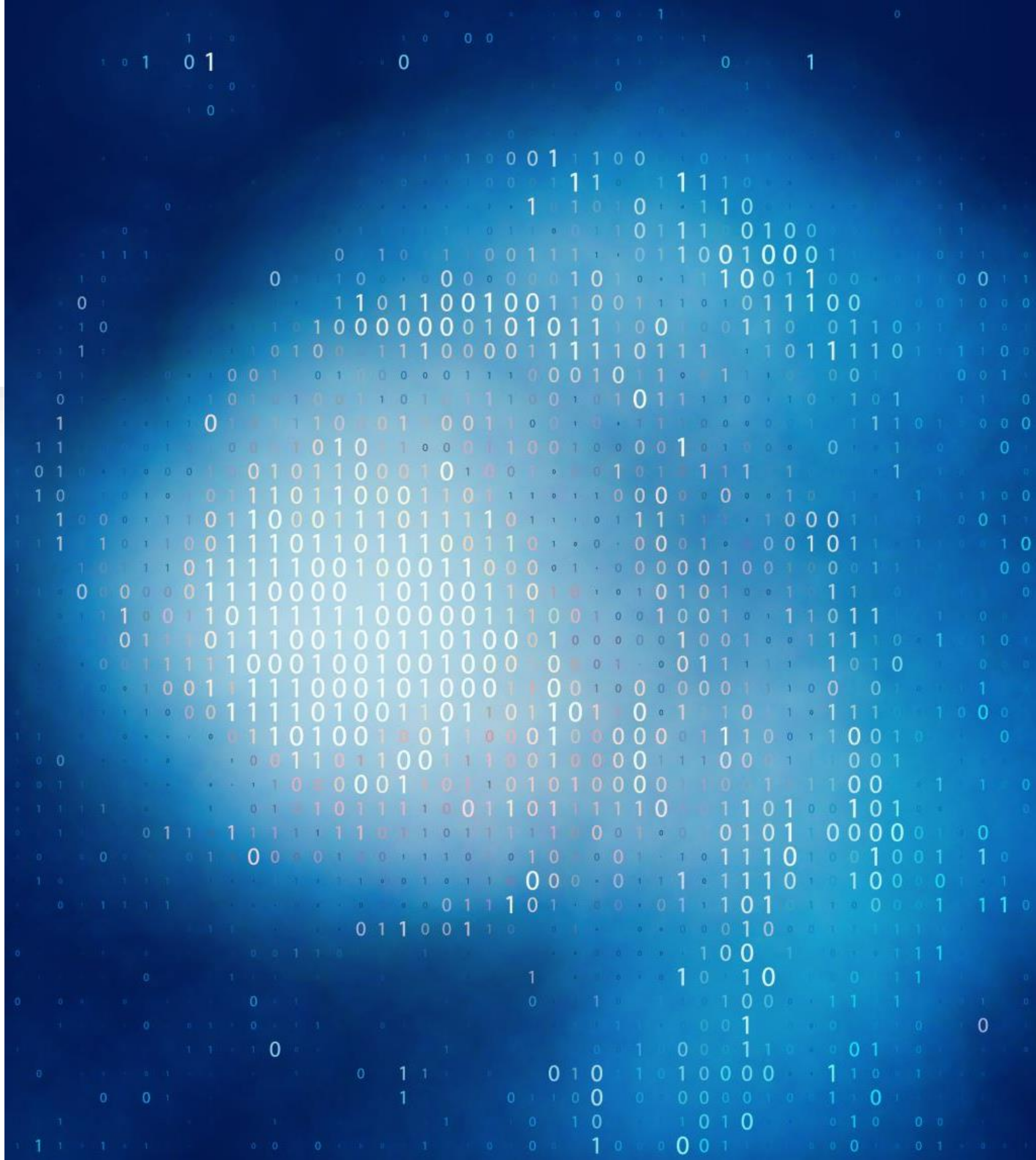
Prof. Alaa Sheta

Algorithms



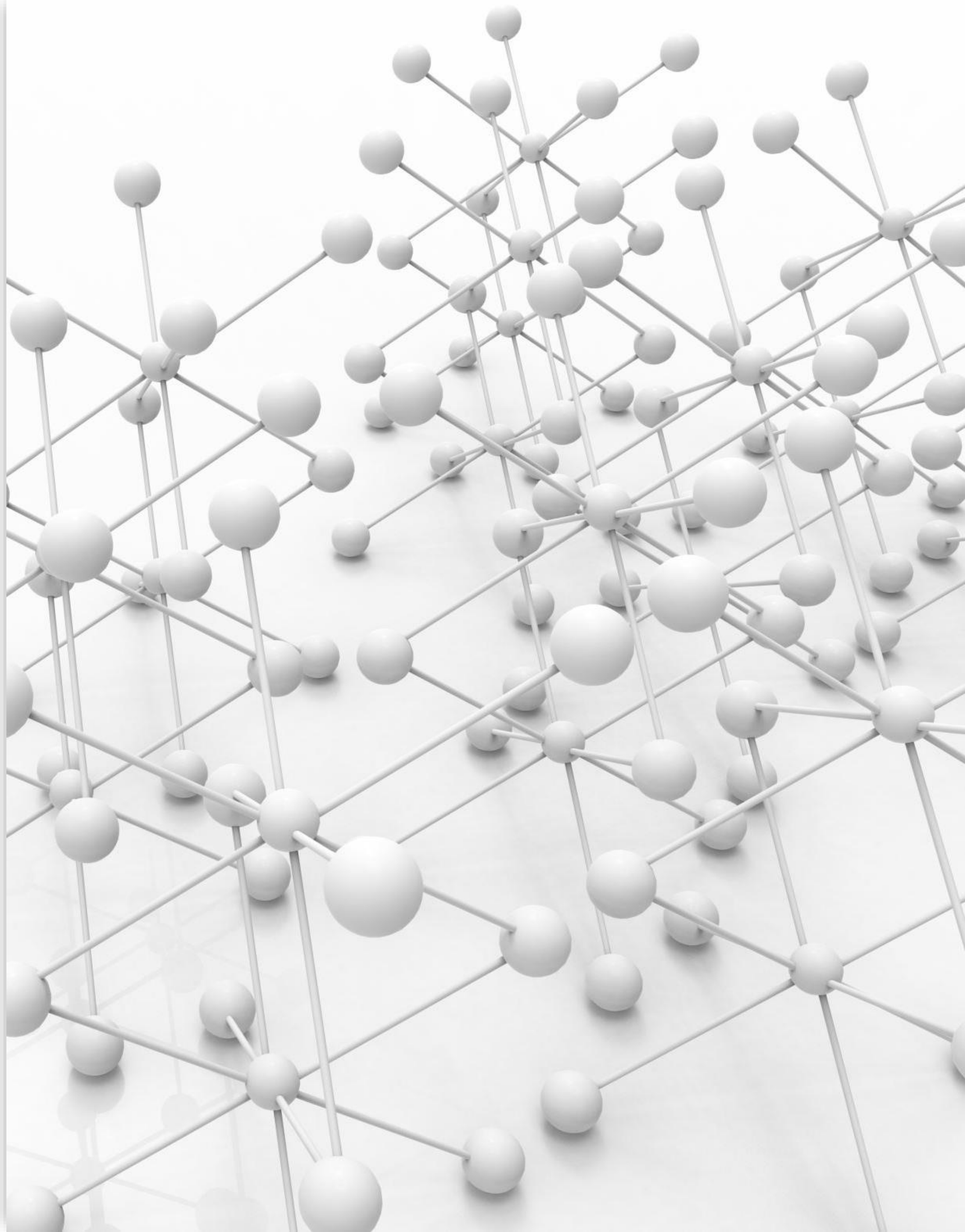
Outlines

- Binary Trees
- Huffman Coding Algorithm
- Graphs
- Dijkstra Algorithm
- Dijkstra Algorithm for TSP



What is a Binary Tree?

- A **binary tree** is a data structure in which each node has at most two children, the left and **right child**.
- It is a hierarchical structure used for various applications like search trees, expression trees, and in many algorithms.
- Each node in a binary tree contains:
    - **Data/Value**
    - **Left child** (points to the left subtree or is null)
    - **Right child** (points to the right subtree or is null)

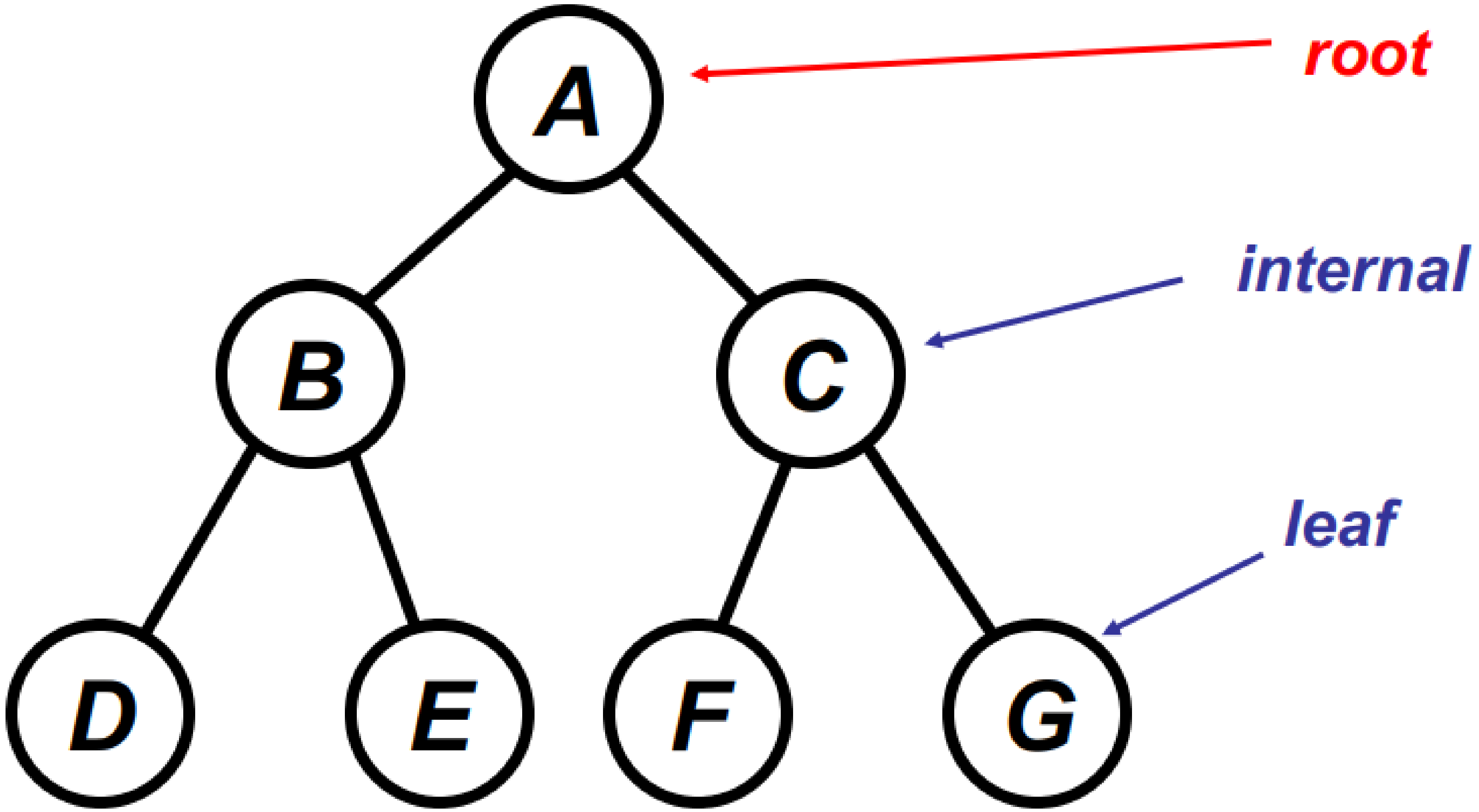


Binary Tree

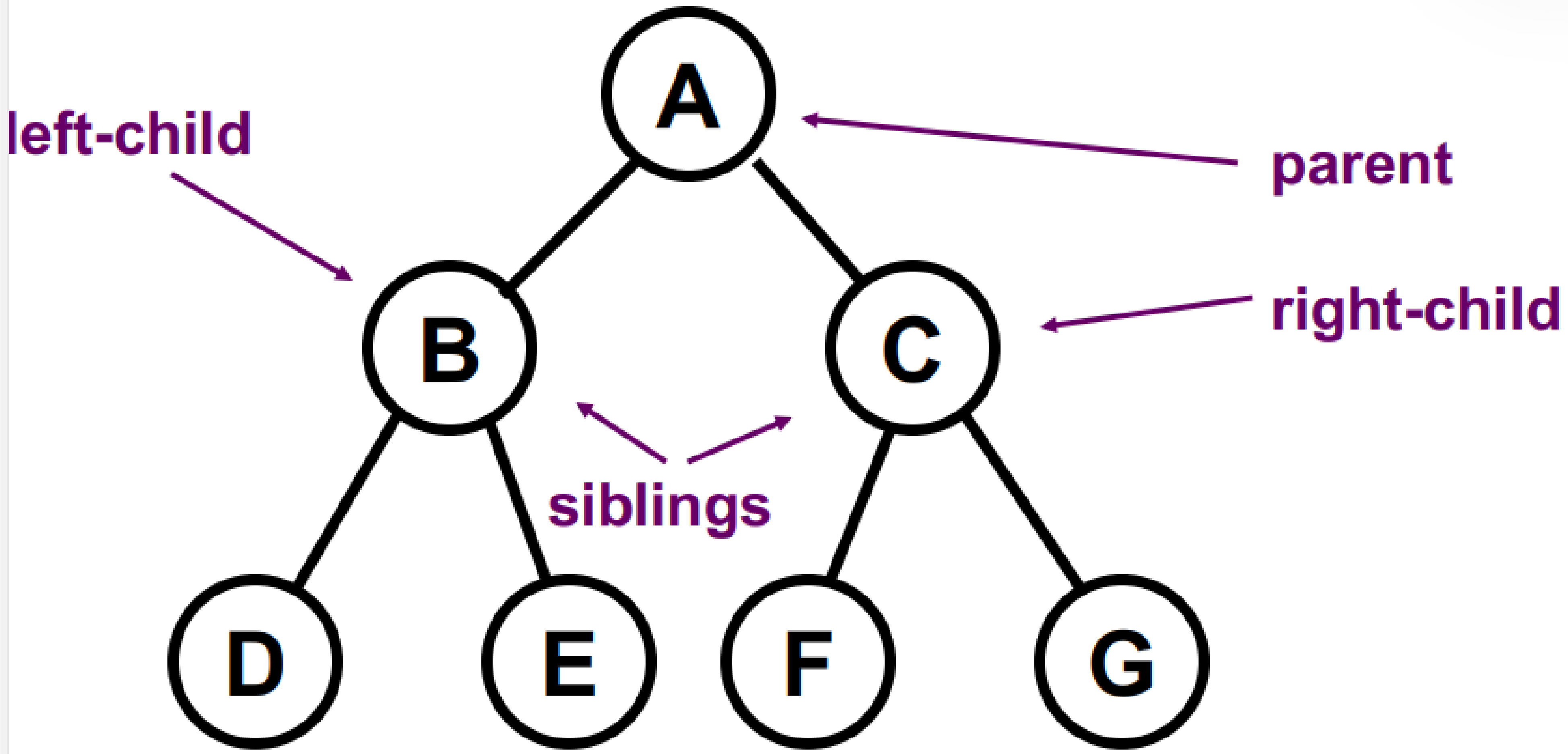
- A binary tree is either
  - empty or
  - has a root node and left- and right-subtrees that are also binary trees.

- The top node of a tree is called the root.
- Any node in a binary tree has at most 2 children.
- Any node (except the root) in a binary tree has exactly one parent node.

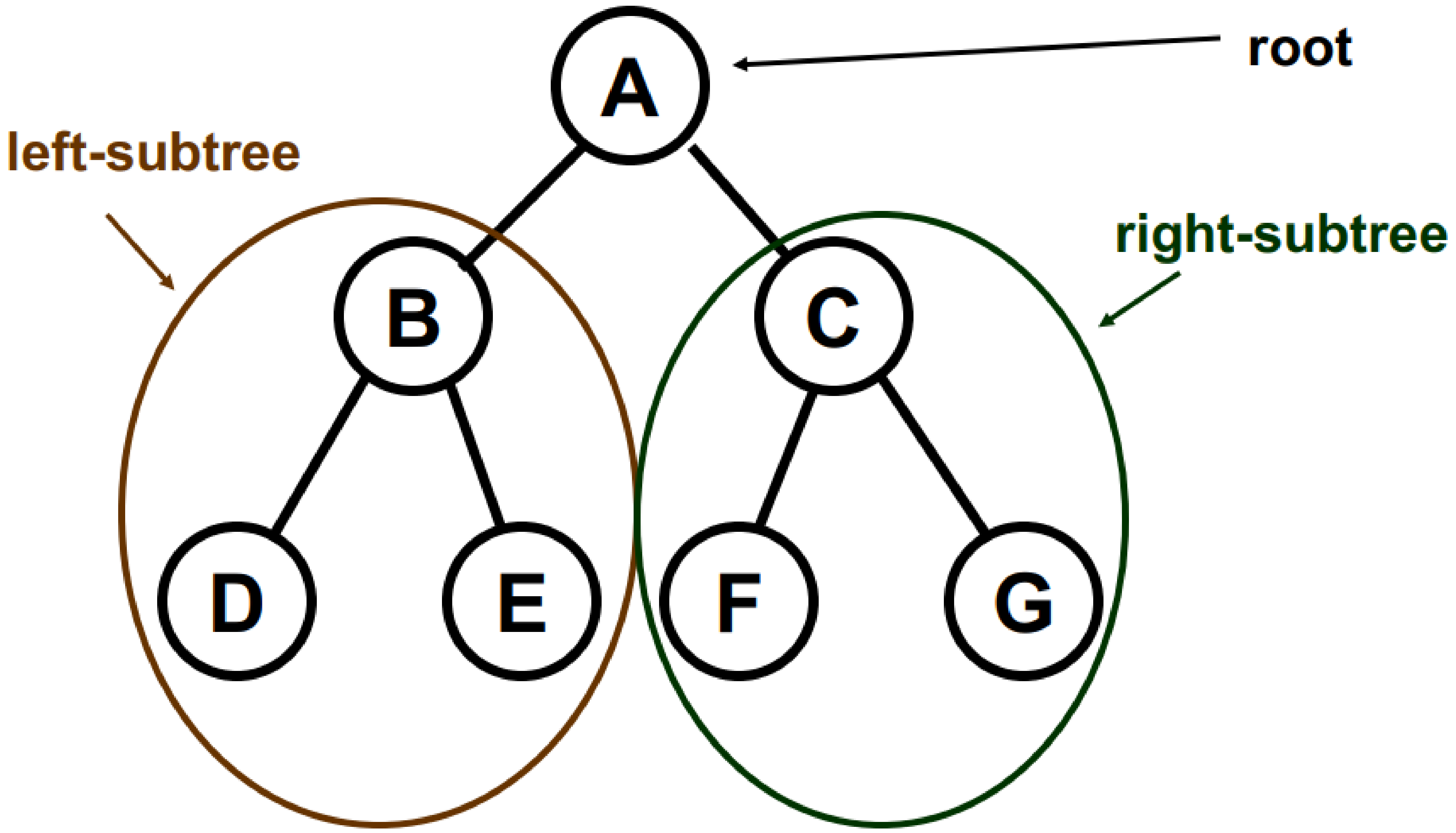
Tree Terminology



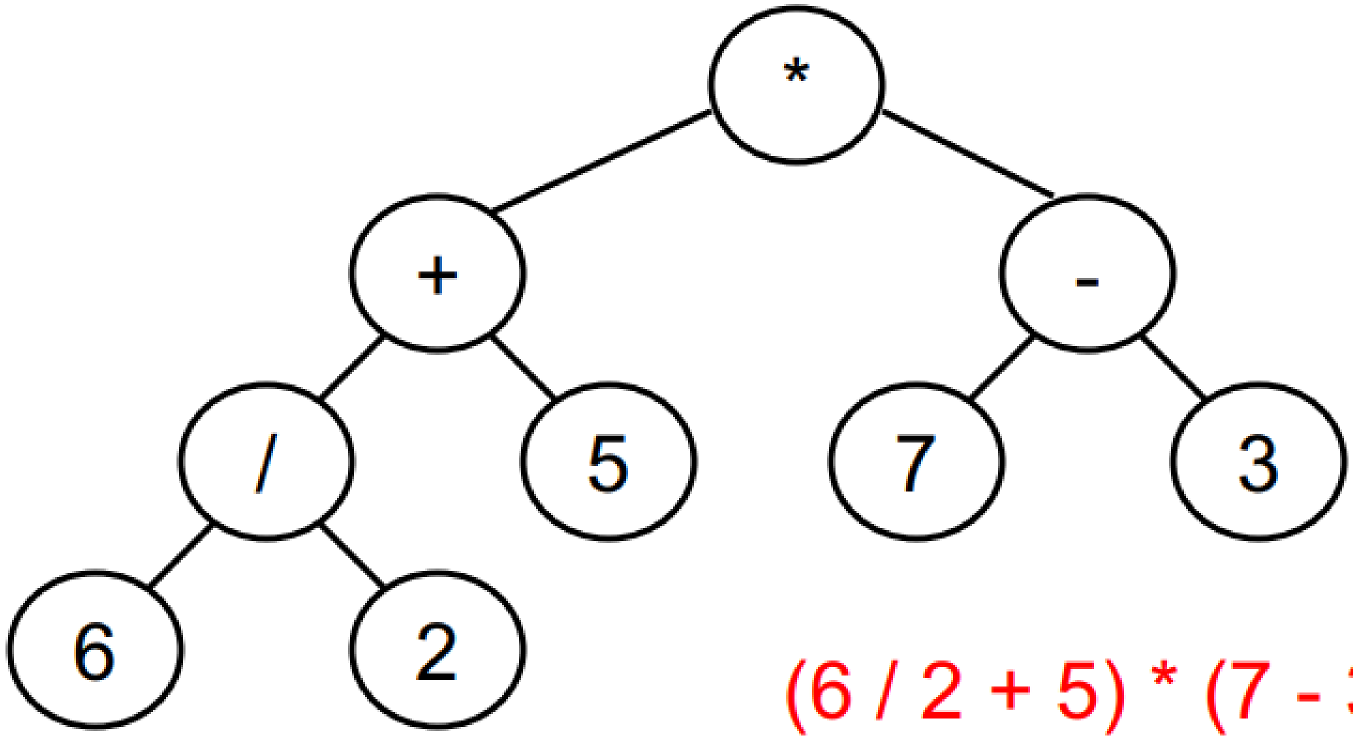
Tree Terminology



Tree Terminology



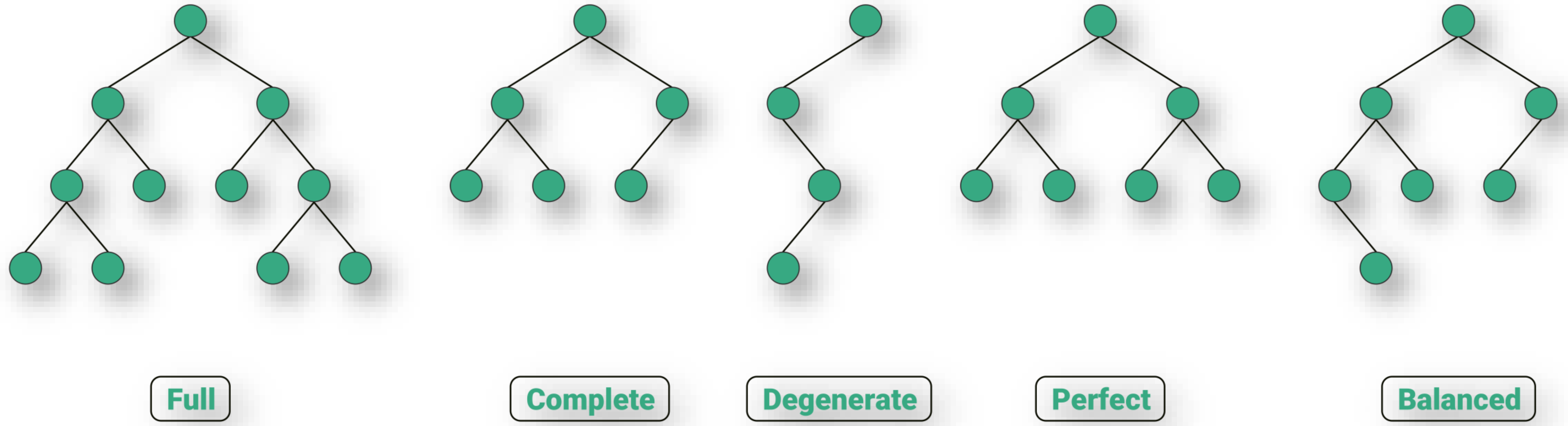
Example: Expression Trees



(6 / 2 + 5) * (7 - 3)

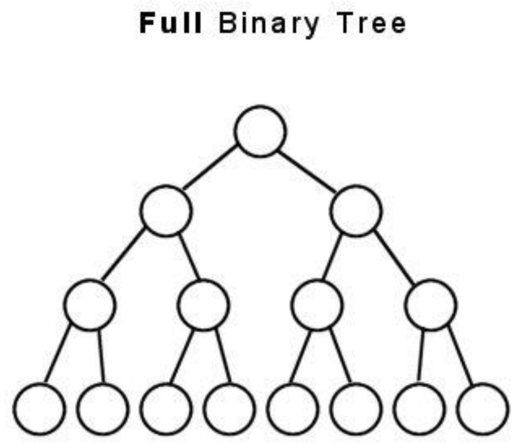
Types of Binary Trees

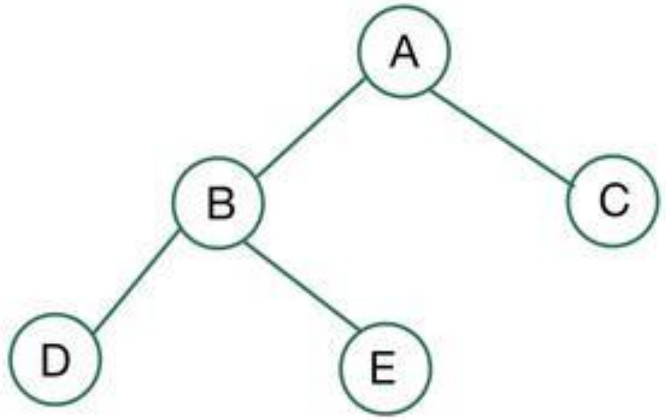
1. **Full Binary Tree**: A binary tree where every node has either 0 or 2 children.
2. **Complete Binary Tree**: A binary tree in which all levels, except possibly the last, are completely filled, and all nodes are as far left as possible.
3. **Perfect Binary Tree**: A binary tree that is full and complete. Every level is fully filled, and all leaf nodes are at the same level.
4. **Balanced Binary Tree**: A binary tree where the height of the left and right subtrees of any node differs by at most 1.
5. **Degenerate (Skewed) Binary Tree**: A tree where every node has only one child, making it behave like a linked list. It can be skewed left or right.

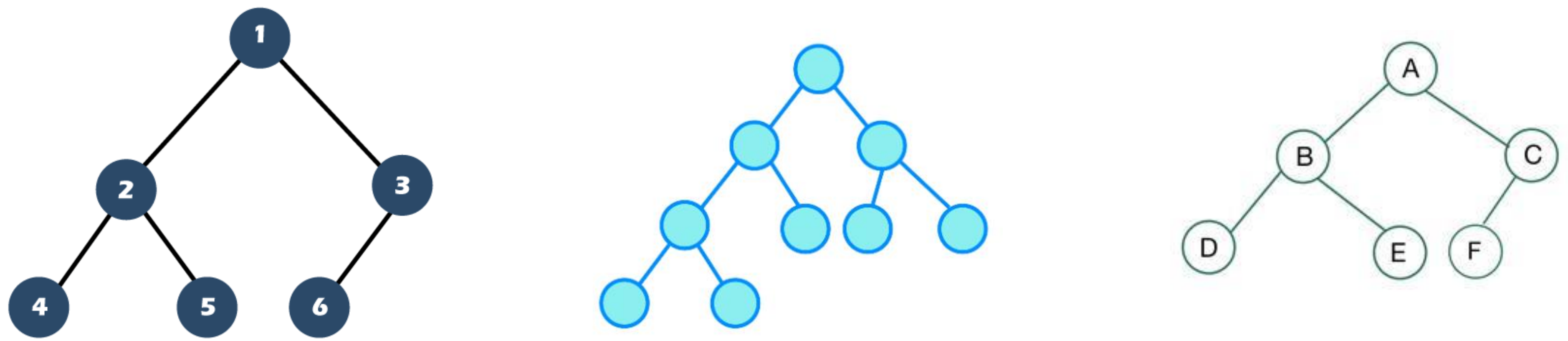

Full
Complete
Degenerate
Perfect
Balanced

Full Binary Tree:

- A full binary tree is one in which all nodes have either 0 or 2 offspring. In other words, a full binary tree is one in which all nodes, except the leaf nodes, have two offspring.

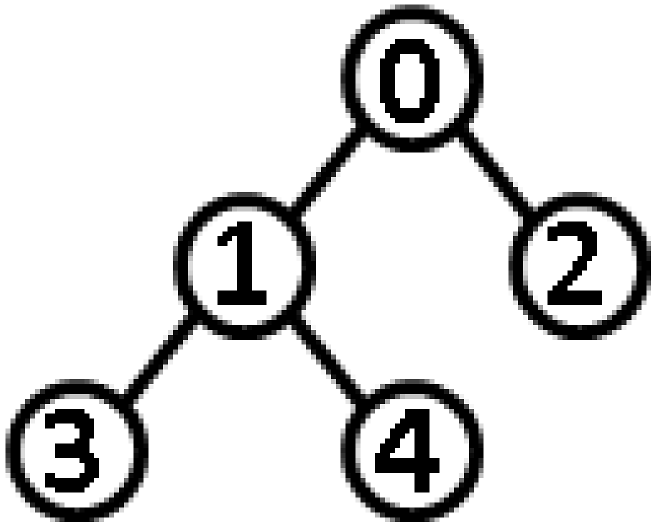


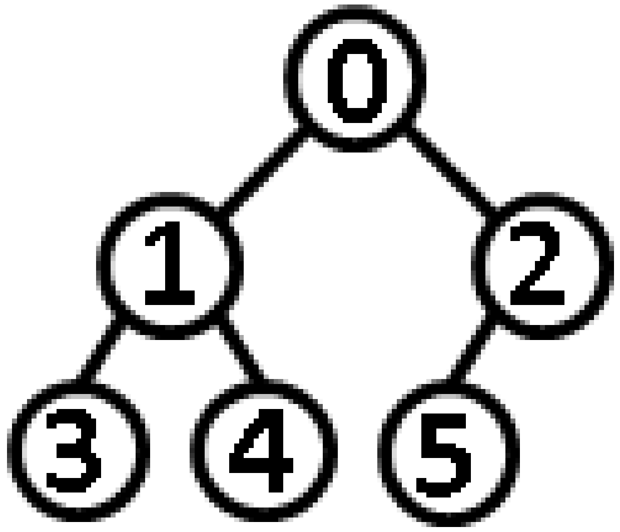


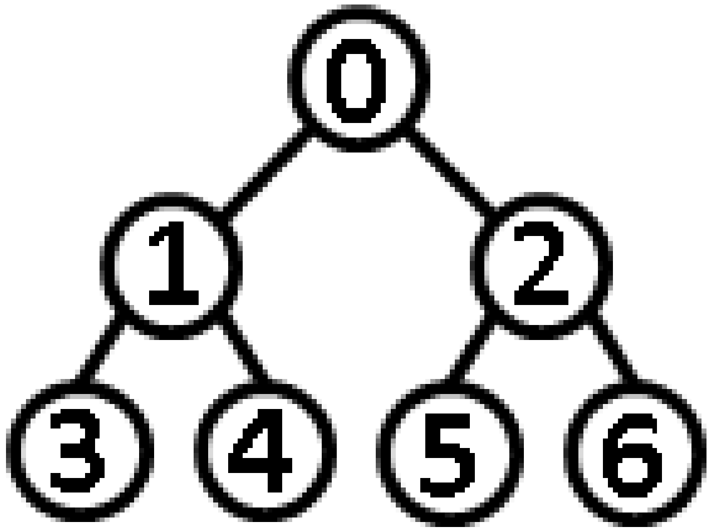


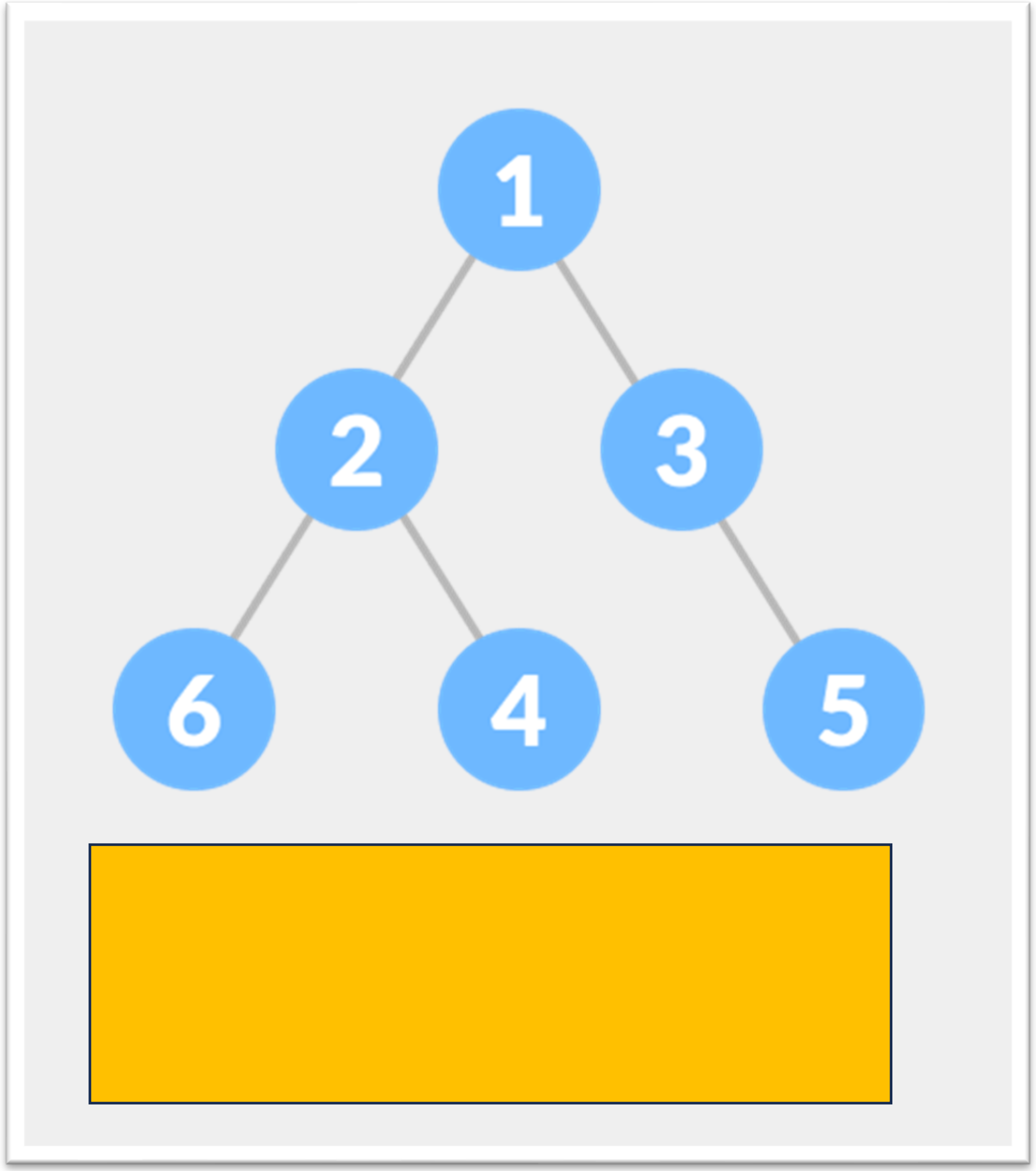
Complete Binary Tree

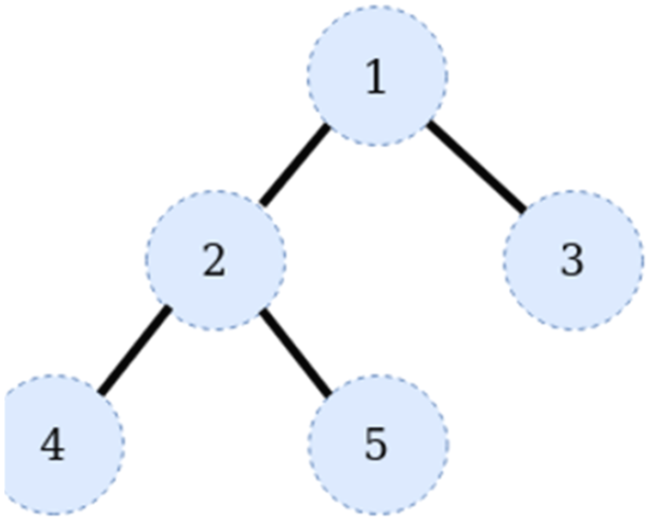
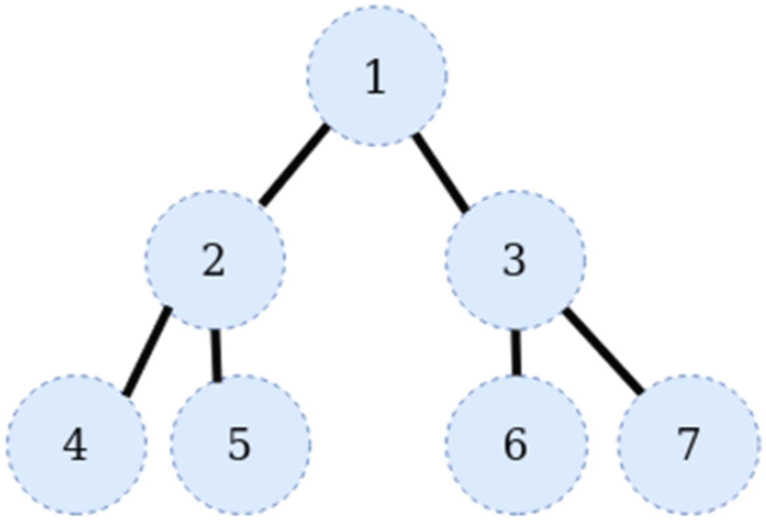
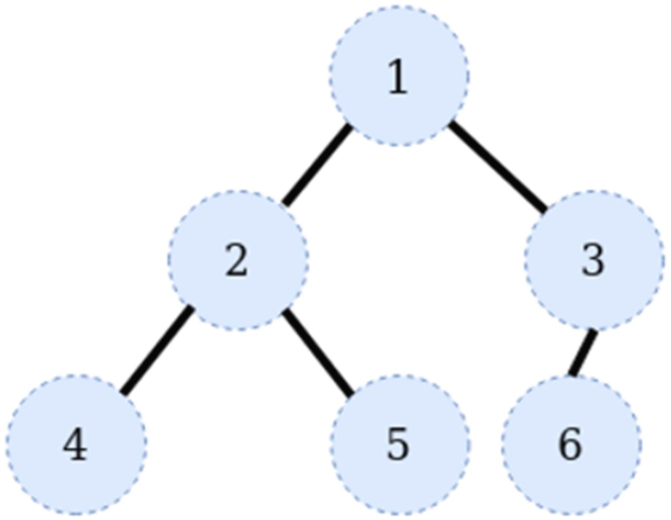
- A binary tree is said to be complete if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible.

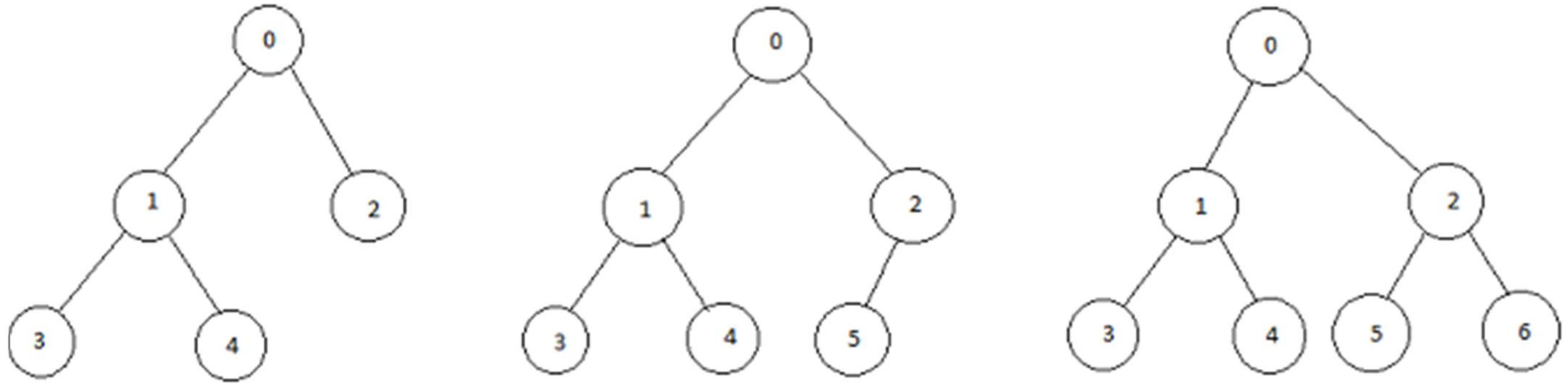

full
binary tree

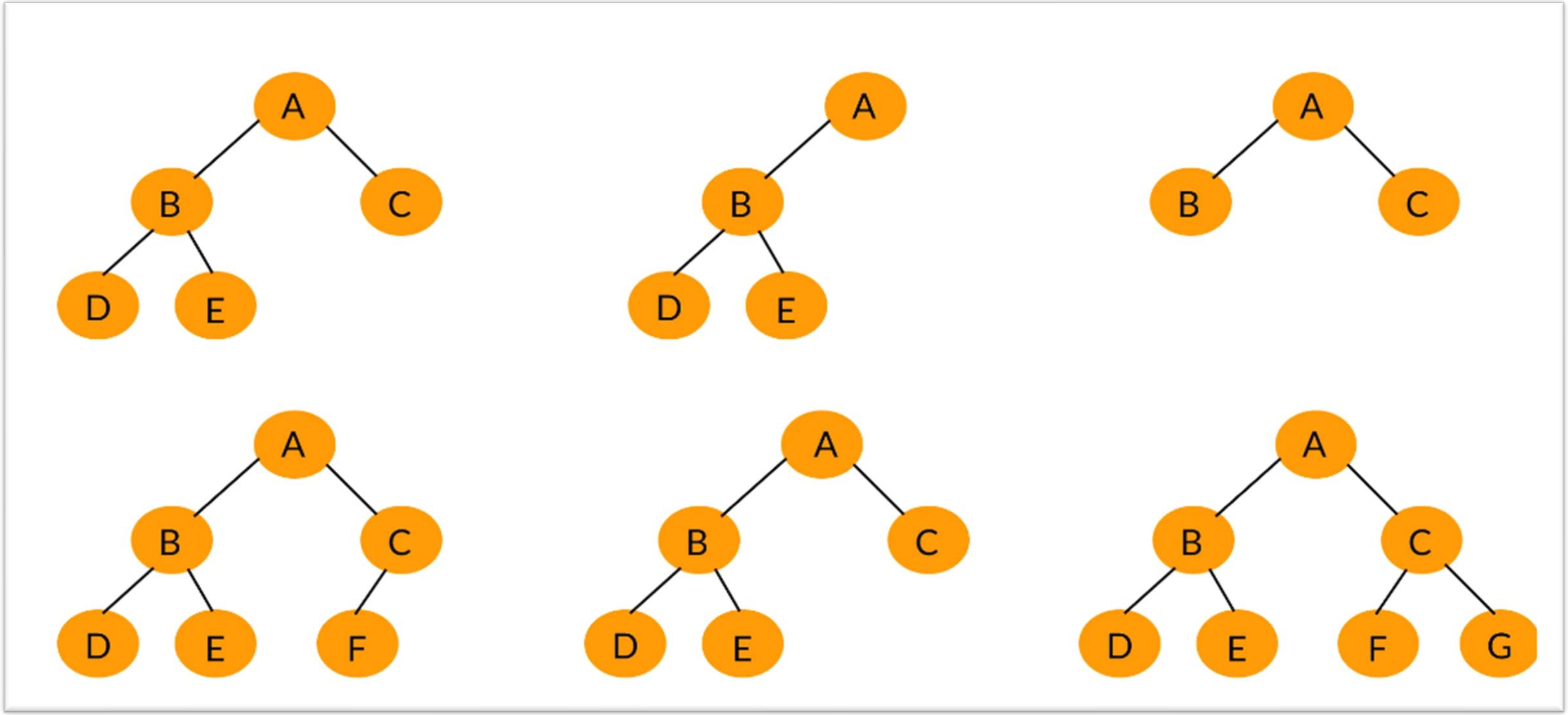

complete
binary tree


perfect
binary tree


1
2
3
6
4
5

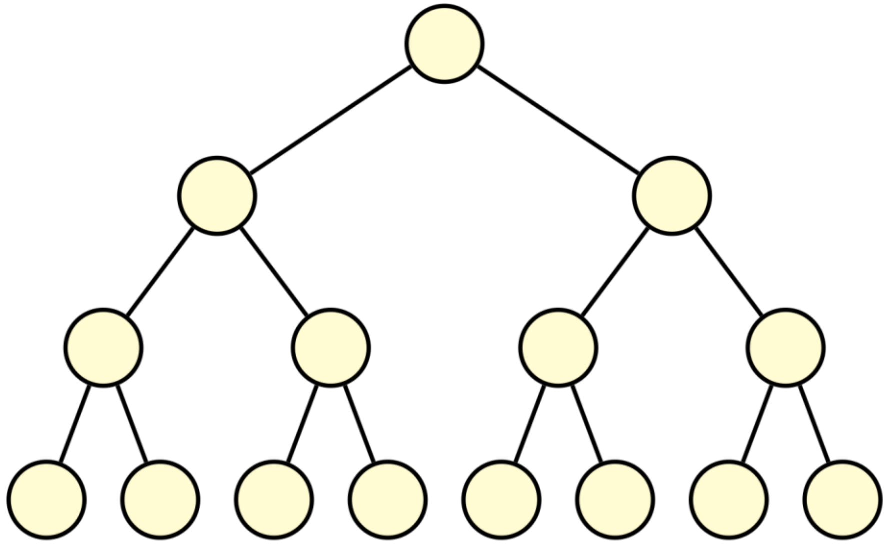

1
2
3
4
5

1
2
3
4
5
6
7

1
2
3
4
5
6


0
1
2
3
4
0
1
2
3
4
5
0
1
2
3
4
5
6

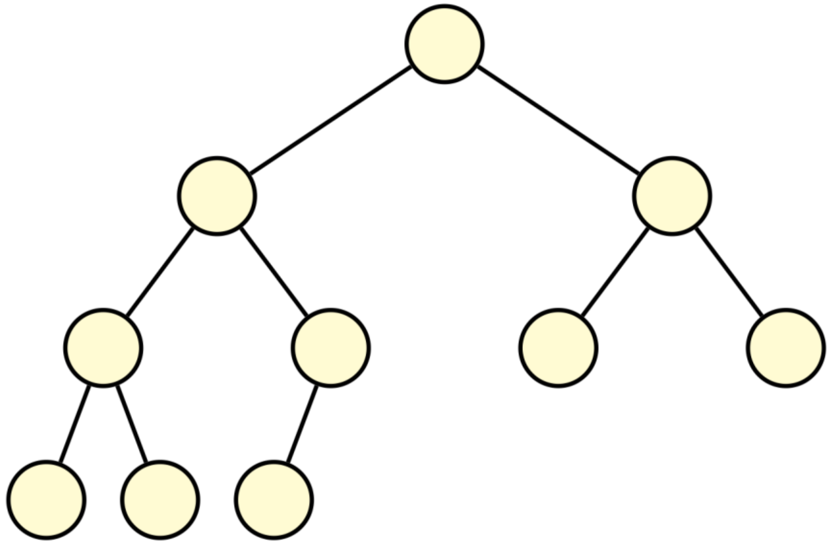


Complete Binary Tree **Full Binary Tree** **Perfect Binary Tree**

Perfect Binary Tree



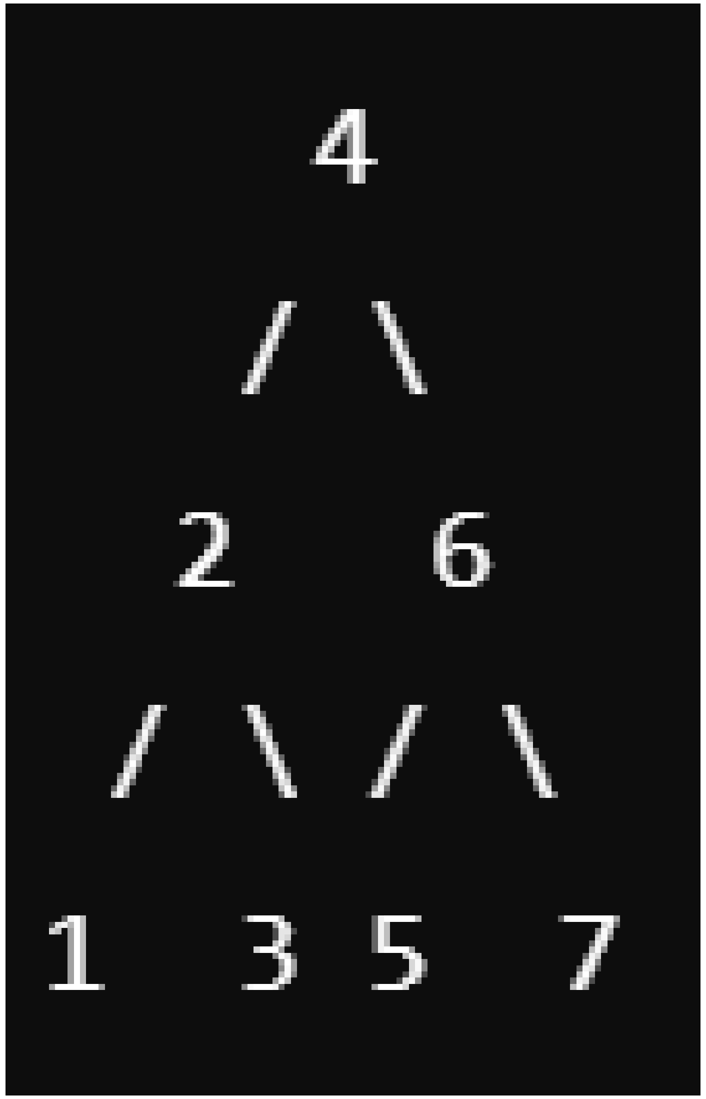
Complete Binary Tree



Tree Traversal Techniques

- Traversal is the process of visiting (checking or updating) each node in the tree.
- There are three common types of binary tree traversal methods:

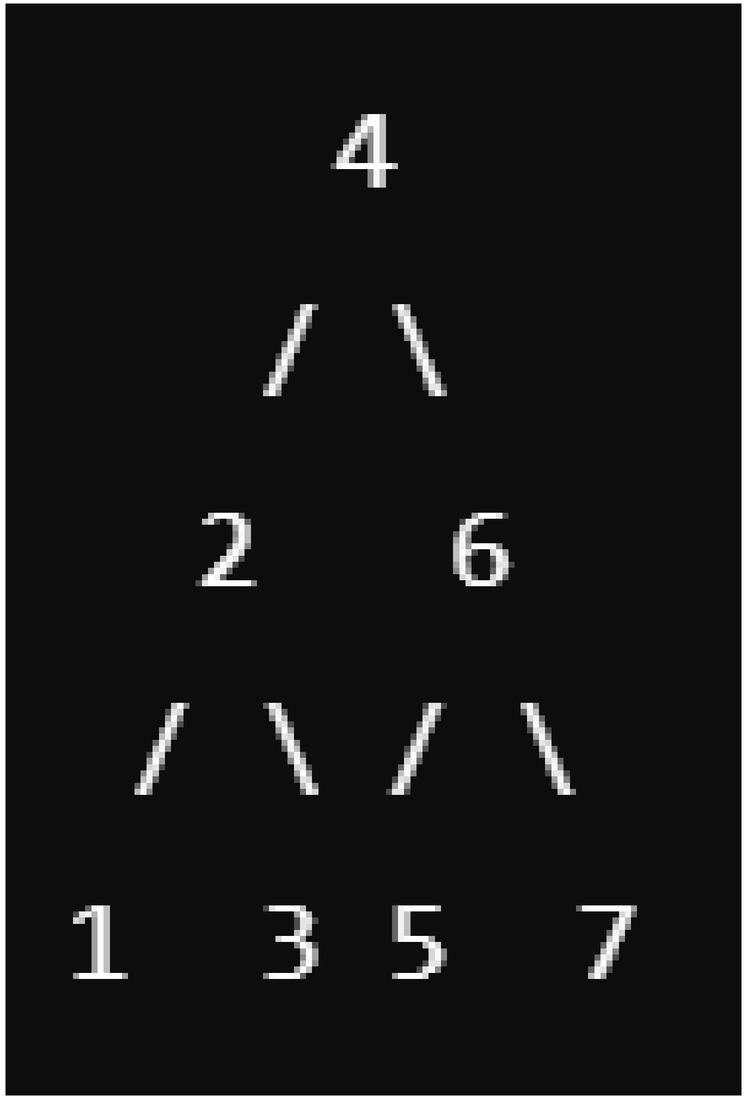
- **In-order traversal** (Left, Root, Right)
    1. Traverse the left subtree.
    2. Visit the root node.
    3. Traverse the right subtree.
- In-order traversal: 1, 2, 3, 4, 5, 6, 7



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

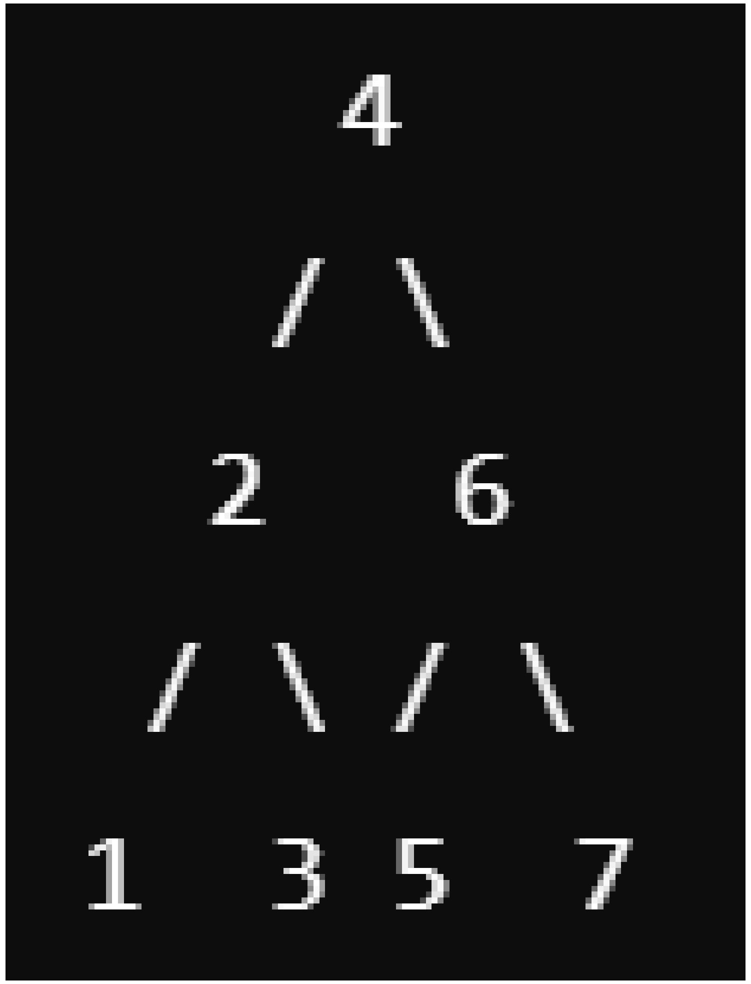
Tree Traversal Techniques

- **Pre-Order Traversal** (Root, Left, Right)
    - Visit the root node.
    - Traverse the left subtree.
    - Traverse the right subtree.
- Pre-Order Traversal: 4, 2, 1, 3, 6, 5, 7



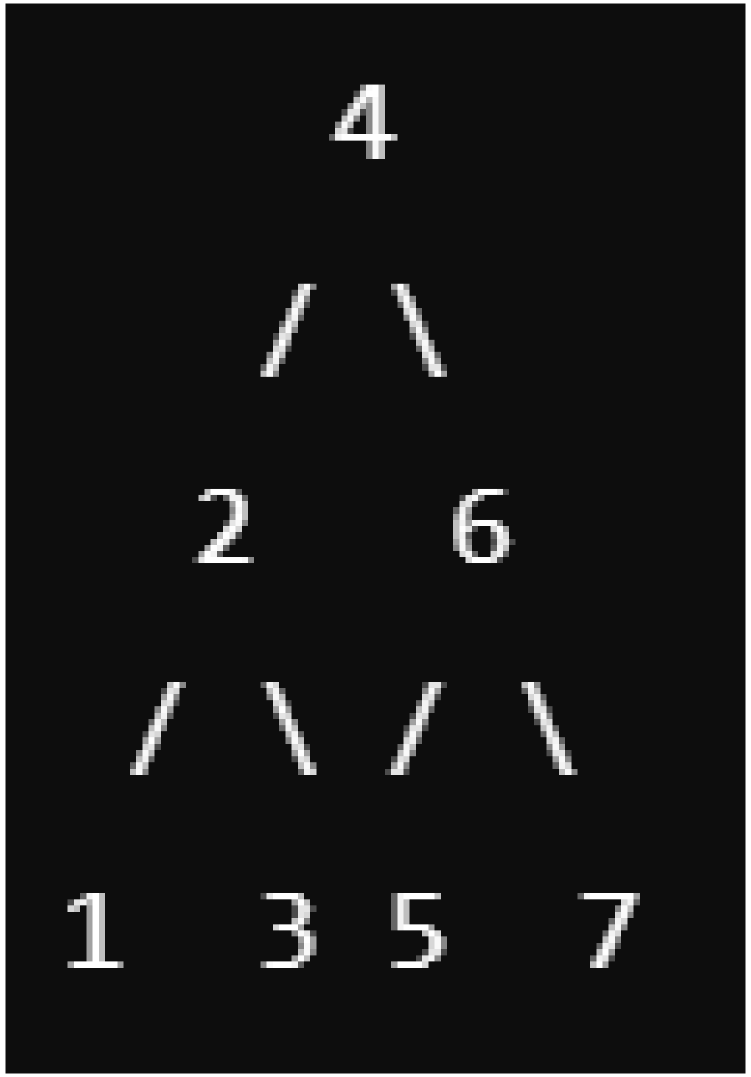
Tree Traversal Techniques

- **Post-Order Traversal** (Left, Right, Root)
  - Traverse the left subtree.
  - Traverse the right subtree.
  - Visit the root node.
- Post-Order Traversal: 1, 3, 2, 5, 7, 6, 4


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

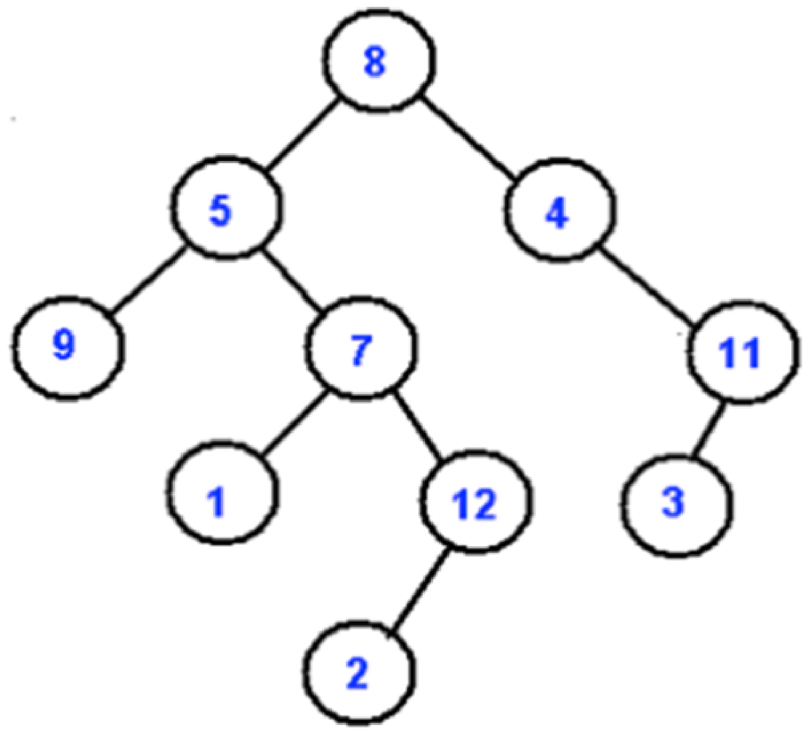
Tree Traversal Techniques

- **Level-Order Traversal** (Breadth-First Search)
  - Visit nodes level by level from top to bottom and from left to right.
- Level-Order Traversal: 4, 2, 6, 1, 3, 5, 7

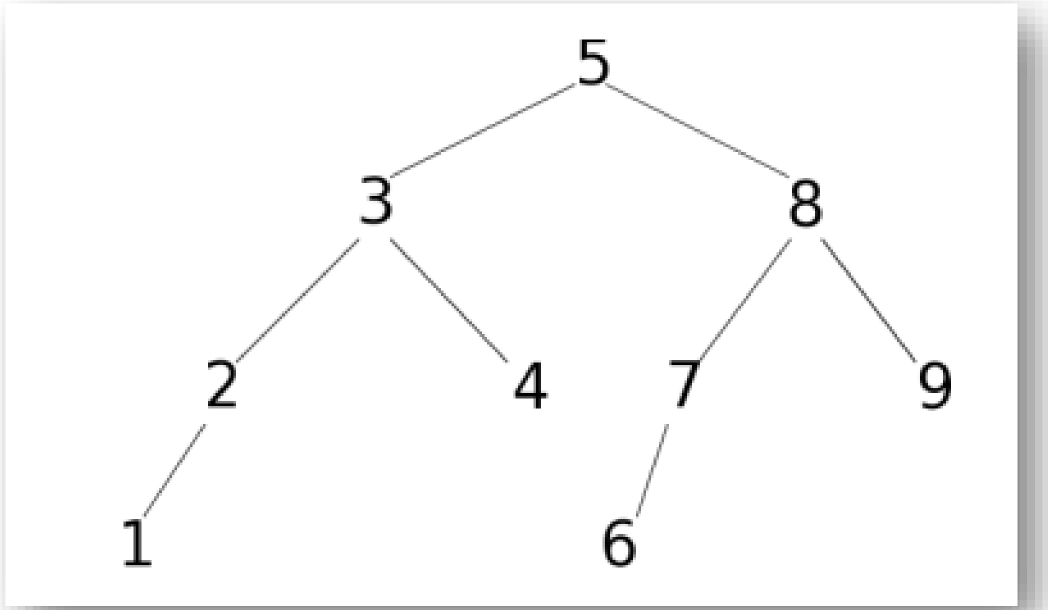


```
    4
   / \
  2   6
 / \ / \
1  3 5  7
```

- **In-order** traversal
  - Visit the left subtree first
  - Visit the node
  - Visit the right subtree
  - Example in-order traversal: 9, 5, 1, 7, 2, 12, 8, 4, 3, 11
- **Post-order** traversal
  - Visit the left subtree first
  - Visit the right subtree
  - Visit the node
  - Example post-order traversal: 9, 1, 2, 12, 7, 5, 3, 11, 4, 8
- **Pre-order** traversal
  - Visit the node first
  - Visit the left subtree
  - Visit the right subtree
  - Example pre-order traversal: 8, 5, 9, 7, 1, 12, 2, 4, 11, 3

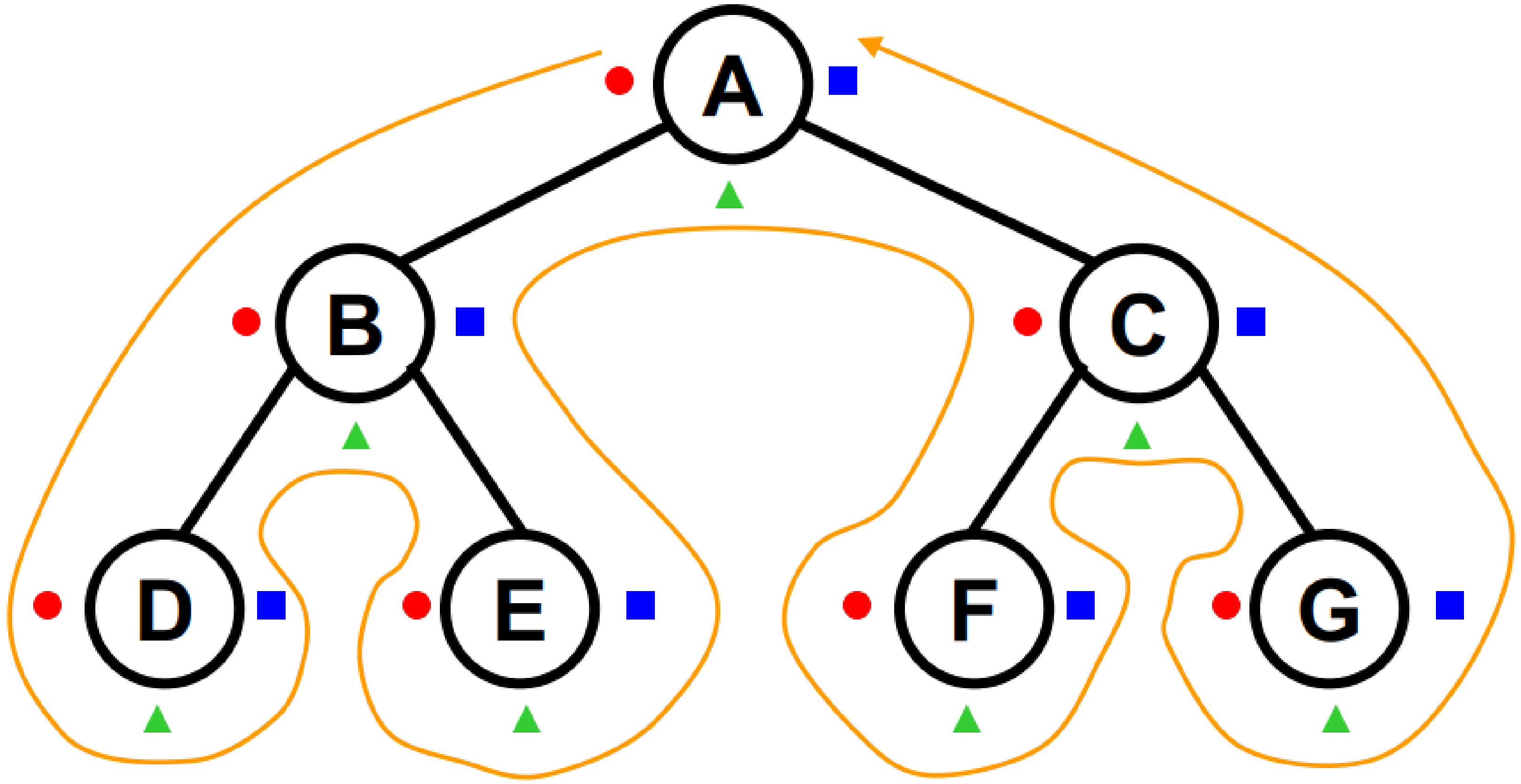


- In-order traverse: 1 2 3 4 5 6 7 8 9
- Post-order traverse: 1 2 4 3 6 7 9 8 5
- Pre-order traverse: 5 3 2 1 4 8 7 6 9



Binary Tree Traversals

●	PREORDER	ABDECFG
▲	INORDER	DBEAFCG
■	POSTORDER	DEBFGCA



Python Simulation

Class exercise

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# In-Order Traversal
def in_order_traversal(root):
    if root:
        in_order_traversal(root.left)
        print(root.val, end=" ")
        in_order_traversal(root.right)

# Pre-Order Traversal
def pre_order_traversal(root):
    if root:
        print(root.val, end=" ")
        pre_order_traversal(root.left)
        pre_order_traversal(root.right)

# Post-Order Traversal
def post_order_traversal(root):
    if root:
        post_order_traversal(root.left)
        post_order_traversal(root.right)
        print(root.val, end=" ")
```

```python
# Create the Binary Tree
root = Node(4)
root.left = Node(2)
root.right = Node(6)
root.left.left = Node(1)
root.left.right = Node(3)
root.right.left = Node(5)
root.right.right = Node(7)

# Print Traversals
print("In-Order Traversal: ")
in_order_traversal(root)

print("\nPre-Order Traversal: ")
pre_order_traversal(root)

print("\nPost-Order Traversal: ")
post_order_traversal(root)
```

Exercises

Q1) Which of the following is true about a Complete Binary Tree?

- A) Every node has exactly two children
- B) All levels, except possibly the last, are completely filled, and all nodes are as far left as possible
- C) All leaf nodes are at the same level
- D) Every node has only one child

- Q2) **What is the defining characteristic of a Full Binary Tree?**
  - A) All nodes have at most two children
  - B) Every node has either 0 or 2 children
  - C) Every node has exactly one child
  - D) Every node has at least one child

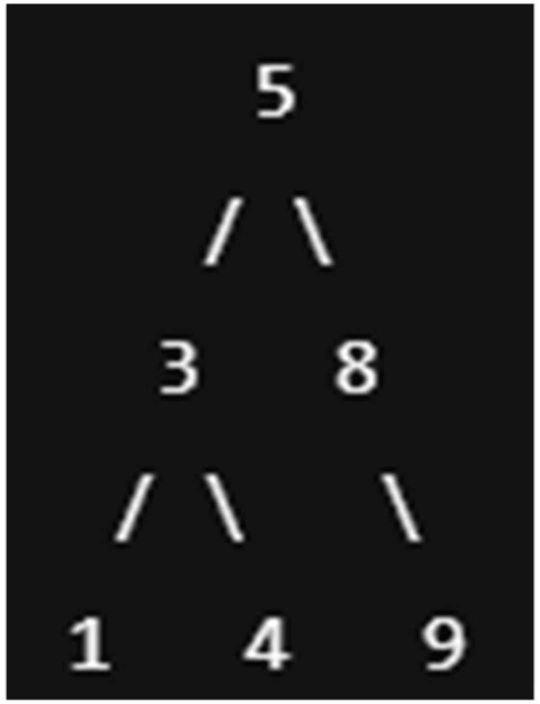
Exercises

- **Q3) In which traversal method is the root node visited before its left and right subtrees?**
    - A) In-Order Traversal
    - B) Pre-Order Traversal
    - C) Post-Order Traversal
    - D) Level-Order Traversal

- **Q4) In which type of binary tree traversal do you visit nodes level by level, from top to bottom and left to right?**
    - A) In-Order Traversal
    - B) Pre-Order Traversal
    - C) Level-Order Traversal
    - D) Post-Order Traversal

Q5) Which of the following is an In-Order traversal sequence of the given binary tree?

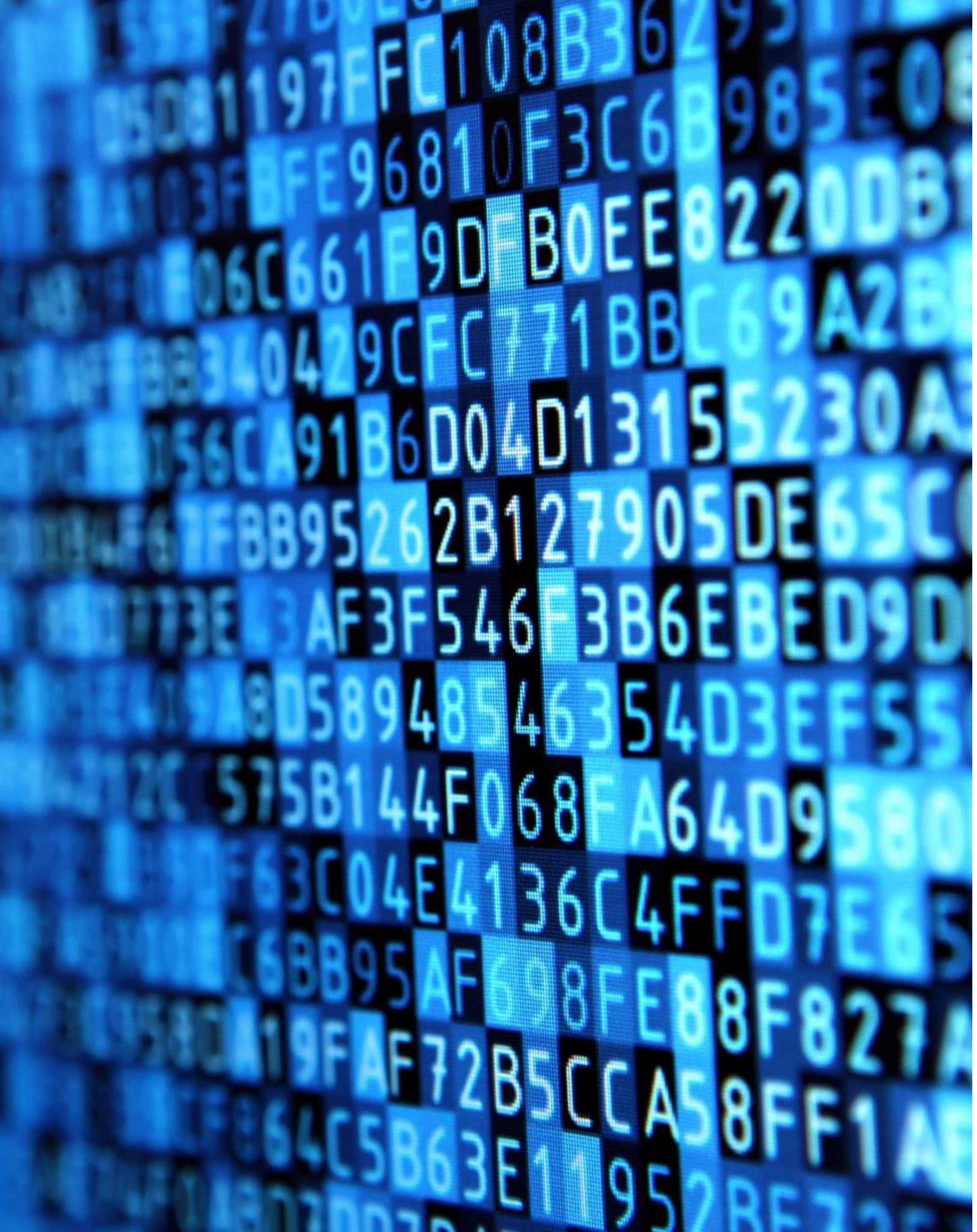
- A) 5, 3, 1, 4, 8, 9
- B) 1, 3, 4, 5, 8, 9
- C) 1, 4, 3, 9, 8, 5
- D) 9, 8, 5, 4, 3, 1



```
    5
   / \
  3   8
 / \   \
1   4   9
```

Huffman Tree

- A **Huffman Tree** is a binary tree used in **Huffman Coding**, a popular algorithm for lossless data compression.
- It is constructed based on the frequency of characters (or symbols) in a dataset and assigns variable-length binary codes to each character.
- Characters that occur more frequently are given shorter codes, while those that occur less frequently receive longer codes, minimizing the overall size of the compressed data.



What is Huffman coding?

- Huffman coding is a lossless data compression algorithm developed by David A. Huffman in 1952.
- It is used to minimize the total number of bits needed to represent a set of data, typically text, by assigning shorter codes to more frequent characters and longer codes to less frequent ones.
- Huffman coding is widely used in compression formats like ZIP files, JPEG images, and MP3 audio.

What is Huffman coding?

- **Huffman coding** assigns codes to characters such that the code's length depends on the corresponding character's relative frequency or weight.
- Huffman codes are variable length and prefix-free (no code is a prefix of any other).
- Any prefix-free binary code can be visualized as a binary tree with the encoded characters stored at the leaves.

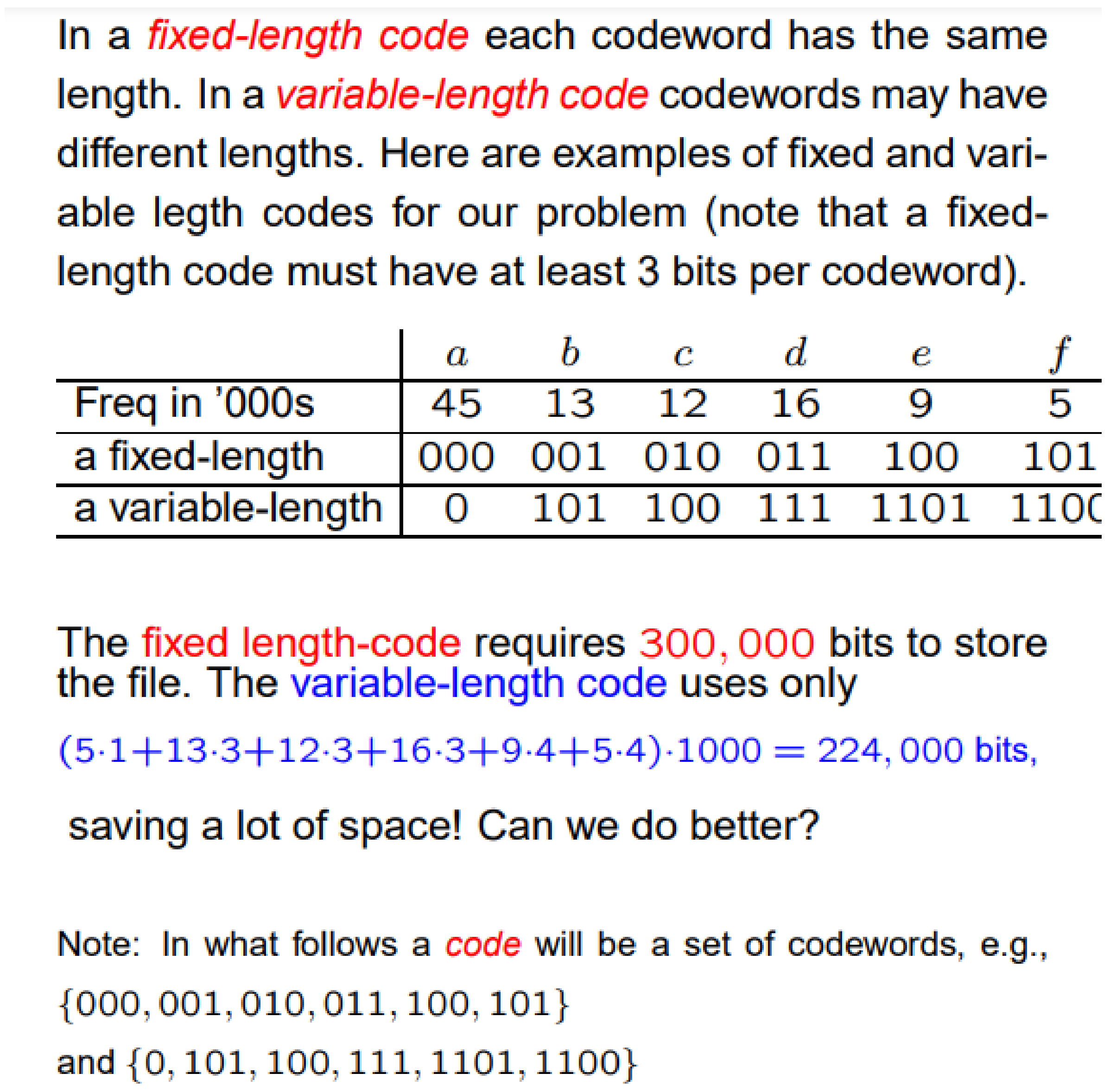
Regular coding for messages

Suppose that we have a $100,000$ character data file that we wish to store . The file contains only 6 characters, appearing with the following frequencies:

	a	b	c	d	e	f
Frequency in '000s	45	13	12	16	9	5

A *binary code* encodes each character as a binary string or *codeword*. We would like to find a binary code that encodes the file using as few bits as possible, ie., *compresses it* as much as possible.

Fixed-Length versus Variable-Length Codes



In a *fixed-length code* each codeword has the same length. In a *variable-length code* codewords may have different lengths. Here are examples of fixed and variable legth codes for our problem (note that a fixed-length code must have at least 3 bits per codeword).

	a	b	c	d	e	f
Freq in '000s	45	13	12	16	9	5
a fixed-length	000	001	010	011	100	101
a variable-length	0	101	100	111	1101	1100

The fixed length-code requires $300,000$ bits to store the file. The variable-length code uses only

$(5 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1000 = 224,000$ bits,

saving a lot of space! Can we do better?

Note: In what follows a *code* will be a set of codewords, e.g., $\{000, 001, 010, 011, 100, 101\}$ and $\{0, 101, 100, 111, 1101, 1100\}$

Fixed-Length versus Variable-Length Prefix Codes

Solution:

characters	a	b	c	d
frequency	60	5	30	5
fixed-length code	00	01	10	11
prefix code	0	110	10	111

To store 100 of these characters,
(1) the fixed-length code requires $100 \times 2 = 200$ bits,
(2) the prefix code uses only

$$60 \times 1 + 5 \times 3 + 30 \times 2 + 5 \times 3 = 150$$

a 25% saving.

Steps of the Huffman Coding Algorithm:

1. **Frequency Calculation**: Count the frequency of each character in the data.
2. **Priority Queue Creation**: Create a priority queue where each element is a node containing a character and its frequency. Nodes are sorted by frequency.
3. **Tree Construction**: Remove the two nodes with the lowest frequency from the priority queue.
   - Create a new internal node with these two nodes as children. The new node's frequency is the sum of the two nodes' frequencies.
   - Insert the new node back into the priority queue.
   - Repeat until only one node remains, which becomes the root of the Huffman tree.
4. **Code Assignment**: Traverse the tree to assign binary codes to each character. Left edges typically represent a 0, and right edges represent a 1. The path from the root to each leaf node gives the Huffman code for that character.

Example

- Let’s take an example to demonstrate Huffman coding for the string ABRACADABRA.

1. Frequency Calculation:
   - A: 5
   - B: 2
   - R: 2
   - C: 1
   - D: 1

2. Priority Queue Creation: (C, 1), (D, 1), (B, 2), (R, 2), (A, 5)

Tree Construction

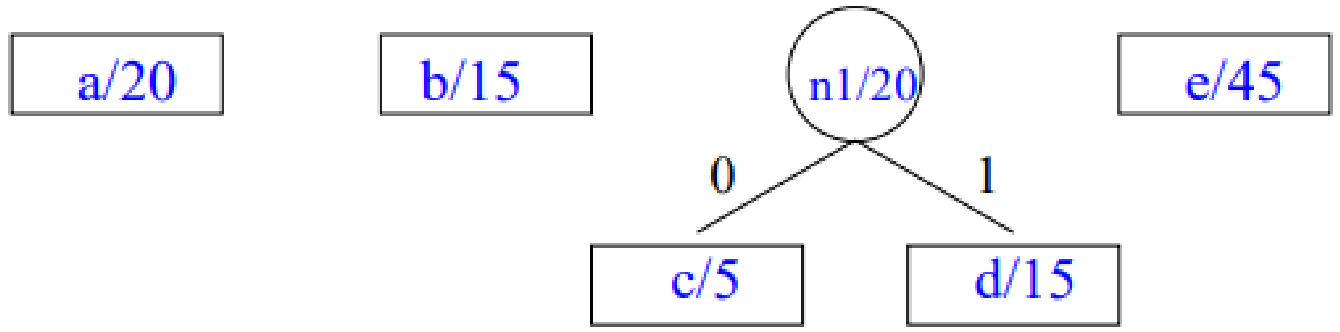
- Priority Queue Creation: (C, 1), (D, 1), (B, 2), (R, 2), (A, 5)
- Combine the two nodes with the lowest frequency: (C, 1) and (D, 1), resulting in a new node with frequency 2.
  - ((C,D), 2), (B, 2), (R, 2), (A, 5)
- Combine (B, 2) and (R, 2), resulting in a new node with frequency 4.
  - ((C,D), 2), ((B,R), 4), (A, 5)
- Combine ((C,D), 2) and ((B,R), 4) to get a node with frequency 6.
  - (((C,D), (B,R)), 6), (A, 5)
- Combine (((C,D), (B,R)), 6) and (A, 5) to form the root with frequency 11.

Code Assignment

- Assigning Codes: Traverse the tree to assign codes:
  - A: 0
  - (C,D): 10
  - (B,R): 11
  - For C: 100, D: 101
  - For B: 110, R: 111
- Huffman Codes for Each Character:
  - A: 0
  - B: 110
  - R: 111
  - C: 100
  - D: 101
- Encoding the String:
- The string ABRACADABRA is encoded as:
  - A B R A C A D A B R A
  - 0 110 111 0 100 0 101 0 110 111 0
- Encoded output: 01101110100010101101110

Example of Huffman Coding

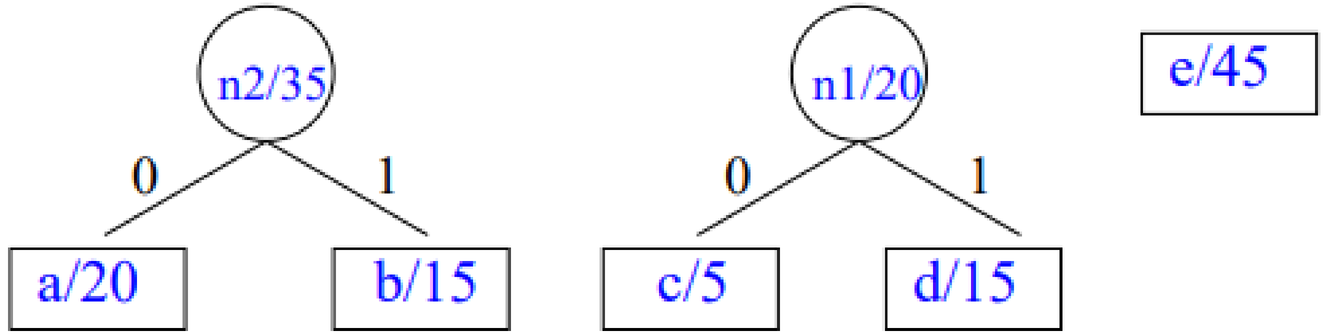
Let $A = \{a/20, b/15, c/5, d/15, e/45\}$ be the alphabet and its frequency distribution.
In the first step Huffman coding merges c and d.



Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.

Example of Huffman Coding – Continued

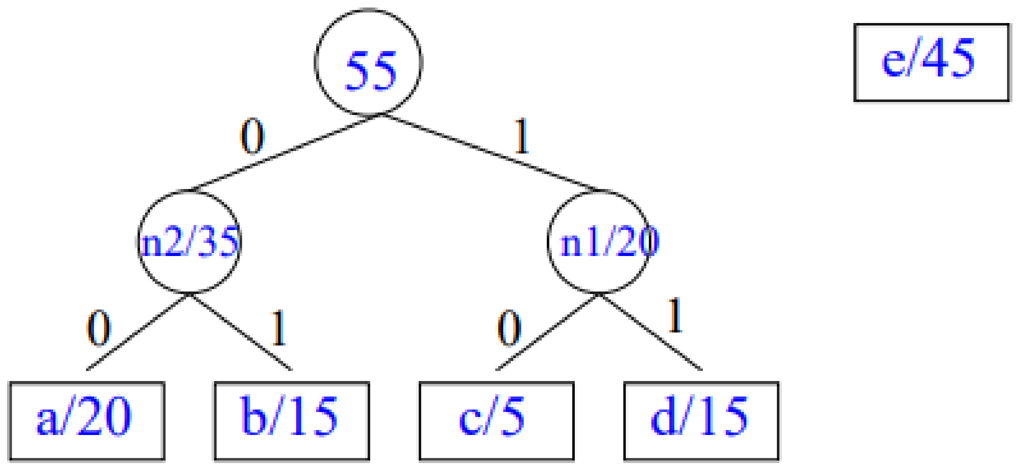
Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.
Algorithm merges a and b
(could also have merged $n1$ and b).



New alphabet is $A_2 = \{n2/35, n1/20, e/45\}$.

Example of Huffman Coding – Continued

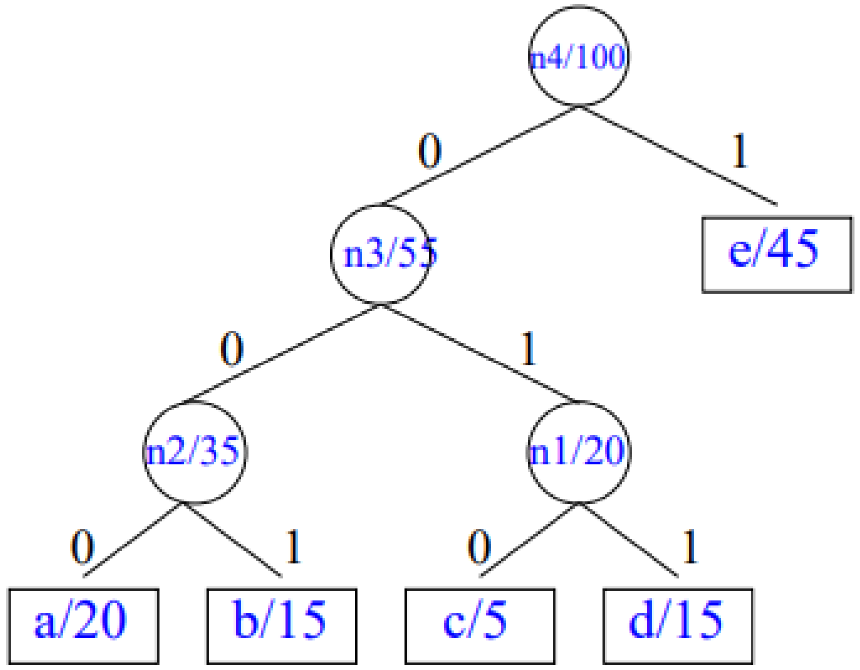
Alphabet is $A_2 = \{n2/35, n1/20, e/45\}$.
Algorithm merges $n1$ and $n2$.



New alphabet is $A_3 = \{n3/55, e/45\}$.

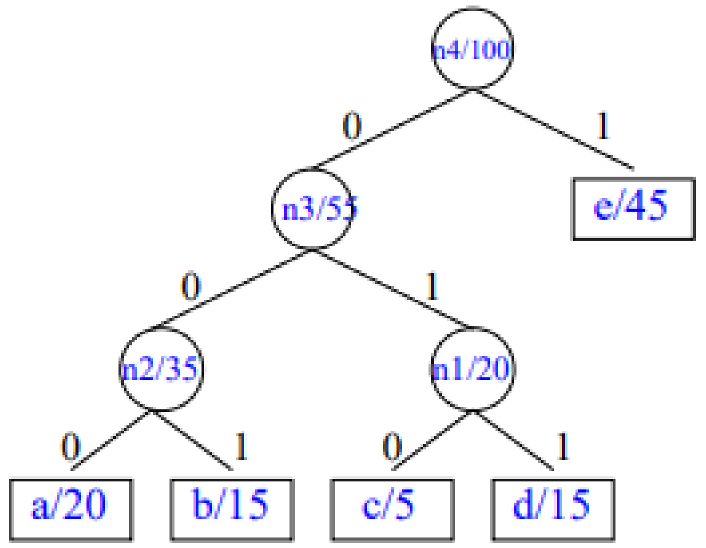
Example of Huffman Coding – Continued

Current alphabet is $A_3 = \{n3/55, e/45\}$.
Algorithm merges e and $n3$ and finishes.



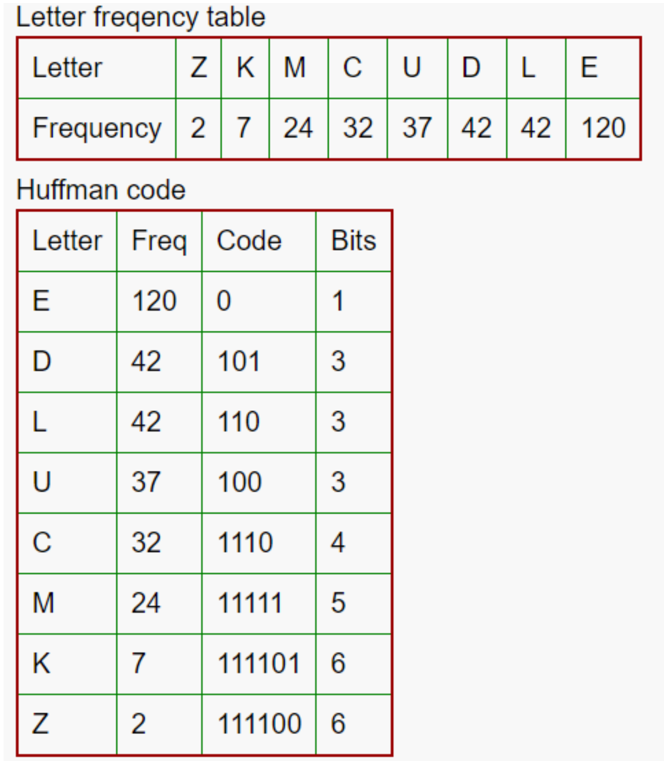
Example of Huffman Coding – Continued

Huffman code is obtained from the Huffman tree.



Huffman code is
$a = 000$, $b = 001$, $c = 010$, $d = 011$, $e = 1$.
This is the optimum (minimum-cost) prefix code for this distribution.

Please draw the Huffman tree structure



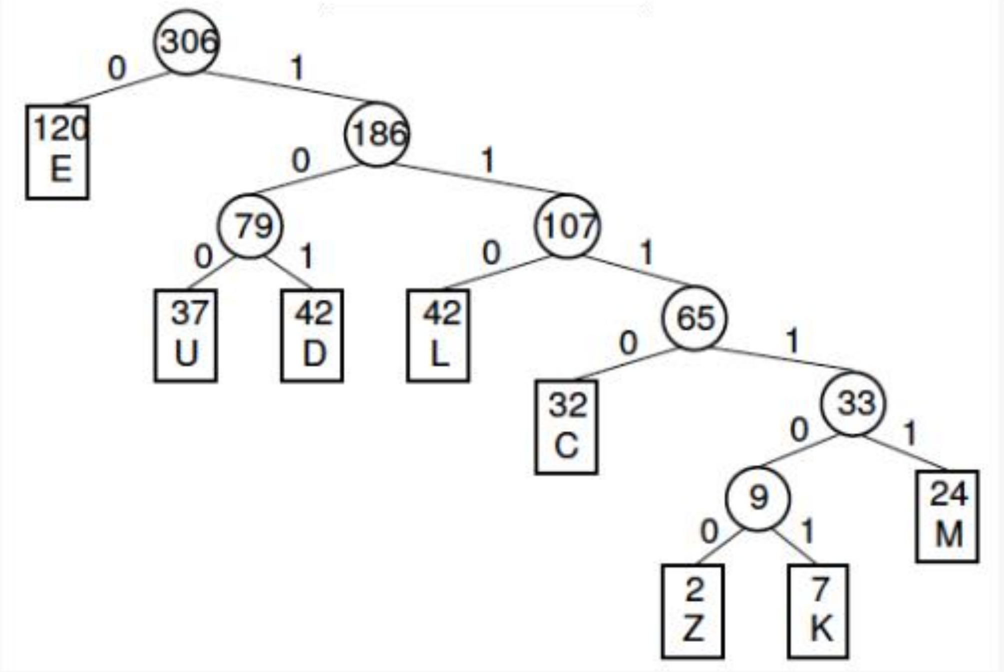
Letter freqency table

Letter	Z	K	M	C	U	D	L	E
Frequency	2	7	24	32	37	42	42	120

Huffman code

Letter	Freq	Code	Bits
E	120	0	1
D	42	101	3
L	42	110	3
U	37	100	3
C	32	1110	4
M	24	11111	5
K	7	111101	6
Z	2	111100	6

The Huffman tree



Time complexity

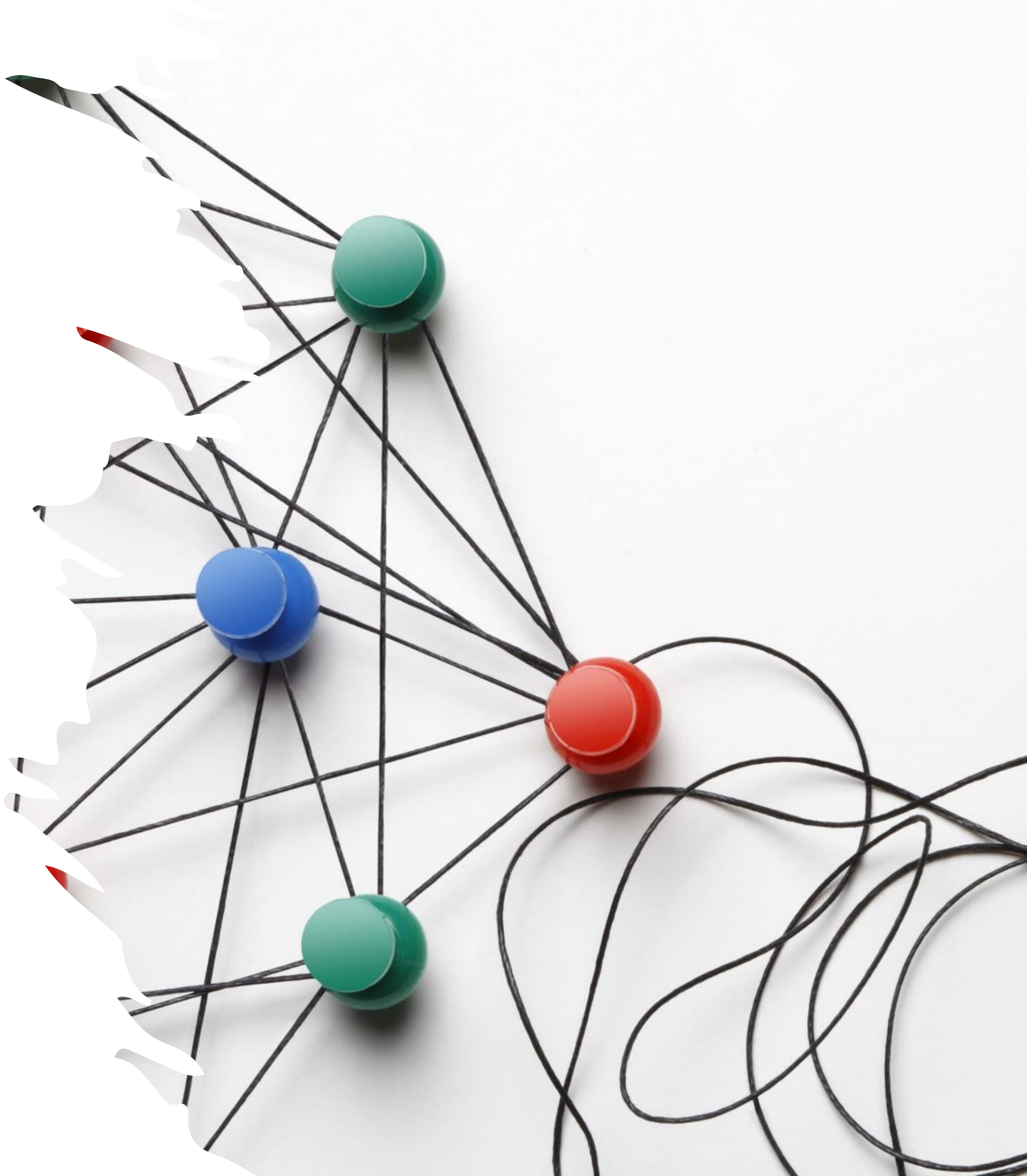
- **Building the Huffman Tree:**
  - To build a Huffman Tree, we need to calculate the frequency of each character in the input data, which takes **O(n)** time, where n is the size of the input.
  - Next, a priority queue (usually implemented as a binary heap) is used to repeatedly extract the two nodes with the smallest frequencies and combine them into a new node. This merging process happens for each character, and each insertion and extraction operation on the heap takes **O(log d)** time, where d is the number of distinct characters (symbols).
  - Since we perform d–1 such merge operations (for d distinct symbols), the total time for building the Huffman Tree is **O(d log d)**.

Time complexity

- **Generating the Codes:**
  - Once the Huffman Tree is built, we perform a traversal (typically a preorder traversal) of the tree to assign binary codes to each symbol. The time complexity for this step is **O(d)**, where d is the number of distinct characters (since we visit each node once).
- **Overall Time Complexity:**
- The overall time complexity of Huffman coding is **O(n + d log d)**, where:
  - n is the size of the input,
  - d is the number of distinct characters.
- In most cases, d is much smaller than n, so the dominant term is typically

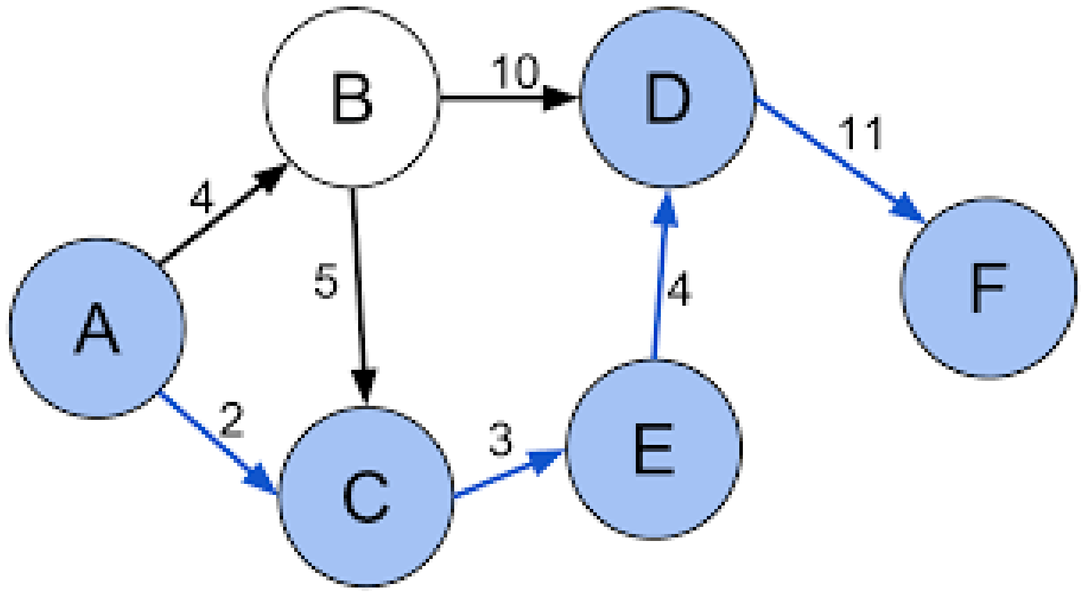
O(n log d)

Dijkstra's algorithm



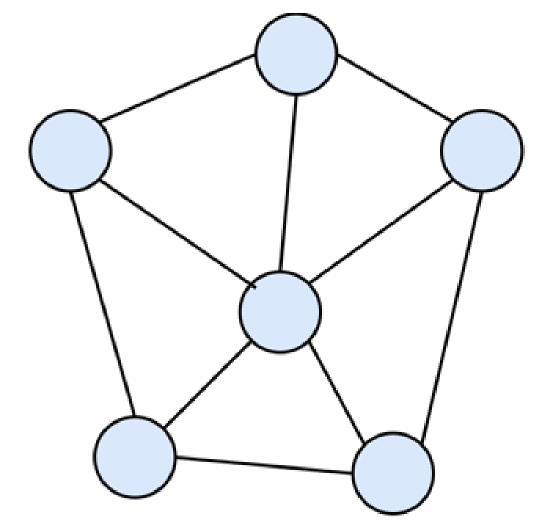
What is a graph?

- In mathematics and computer science, a **graph** is a collection of nodes (or vertices) and edges (or links) that connect pairs of nodes.
- Graphs are used to model relationships or connections between entities.
- Here are some key concepts related to graphs:
  - **Vertex (Node)**: An individual entity or point in a graph. For example, each person is a vertex in a social network graph.
  - **Edge (Link)**: A connection or relationship between two vertices. For instance, in a social network graph, an edge might represent a friendship or a follow relationship.

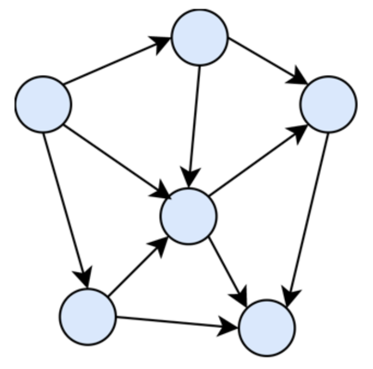


Types of graphs

- **Undirected Graph**: A graph where edges have no direction. The edge (u, v) is the same as (v, u). For example, in a friendship network, the friendship relationship is mutual.
- **Directed Graph (Digraph)**: A graph where edges have a direction. An edge (u, v) indicates a connection from vertex u to vertex v. For instance, in a web page link graph, a directed edge might represent a hyperlink from one page to another.



Undirected Graph

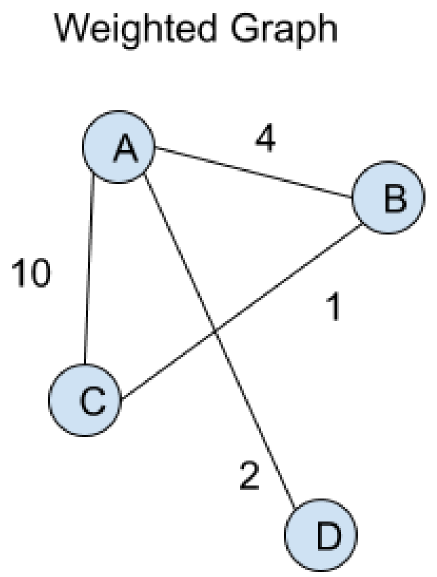


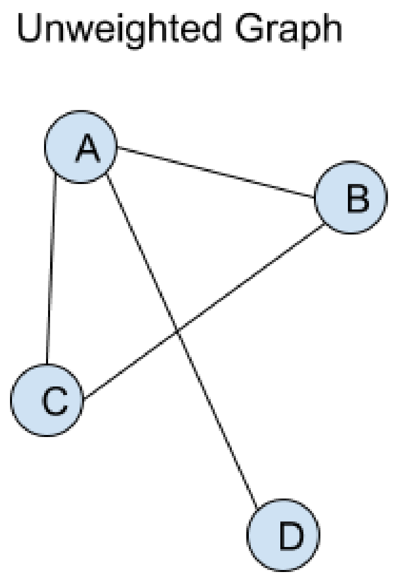
Directed Graph

Types of graphs

- **Weighted Graph**: A graph where edges have weights or costs associated with them. For example, in a road network, weights might represent distances or travel times.

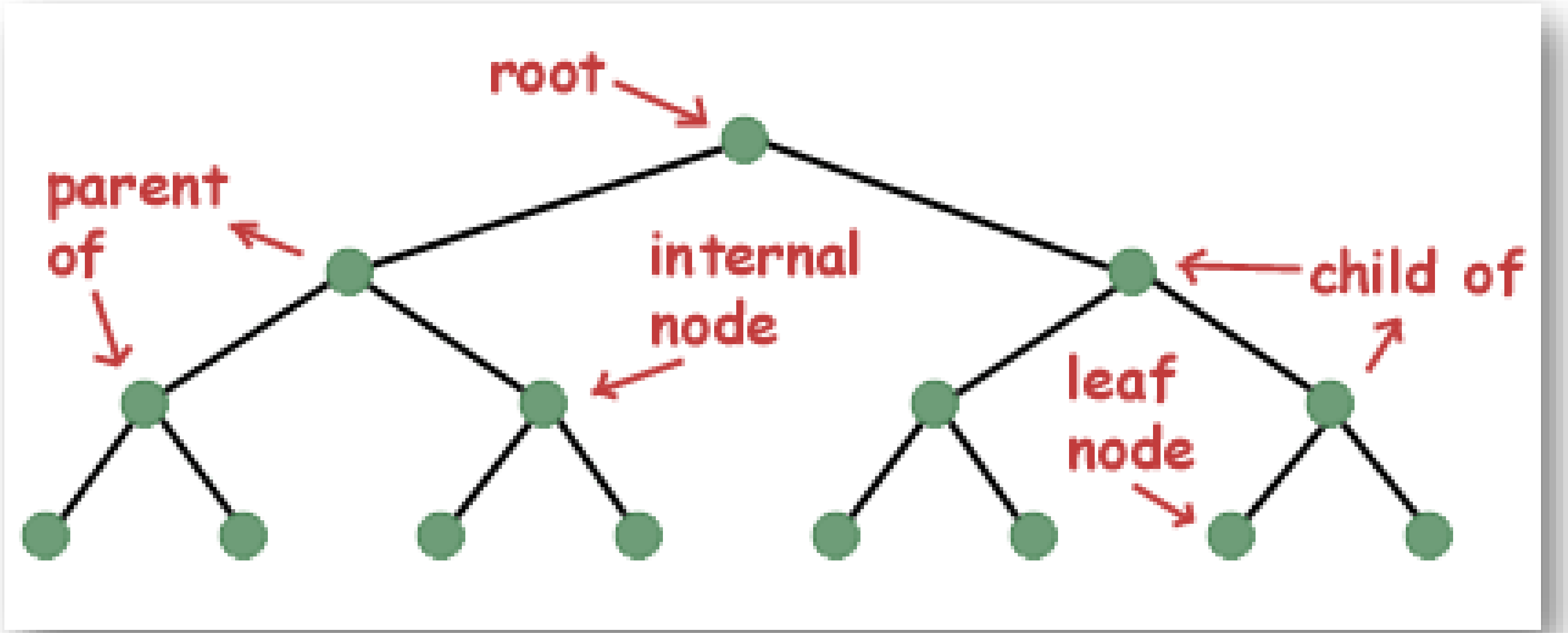
- **Unweighted Graph**: A graph where edges have no weights, or all edges are considered equal.





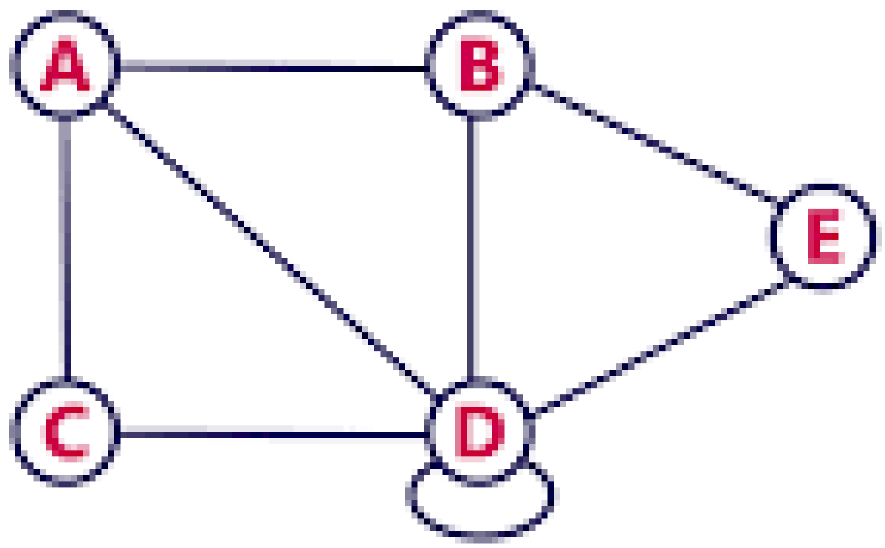
Types of graphs

- **Cycle**: A path in a graph that starts and ends at the same vertex without repeating any edges or vertices.
- **Connected Graph**: An undirected graph is connected if there is a path between every pair of vertices.
- **Tree**: A special type of connected acyclic graph. A tree has no cycles and has exactly one path between any pair of vertices.
- **Subgraph**: A graph formed from a subset of the vertices and edges of a larger graph.



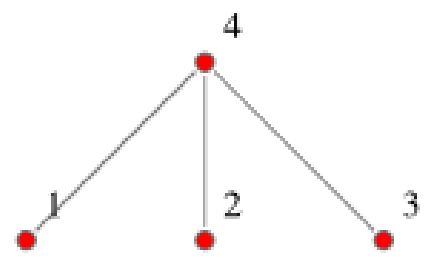
Convert a Graph to a Matrix

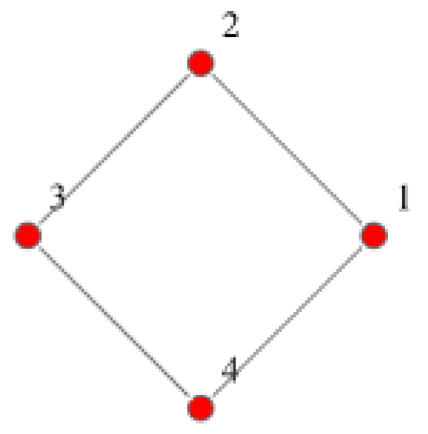
Convert a Graph to a Matrix

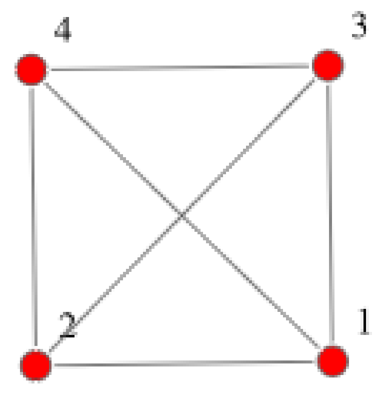


$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & 0 & 1 & 1 & 1 & 0 \\ B & 1 & 0 & 0 & 1 & 1 \\ C & 1 & 0 & 0 & 1 & 0 \\ D & 1 & 1 & 1 & 1 & 1 \\ E & 0 & 1 & 0 & 1 & 0 \end{array}$$

Convert the matrix to a graph







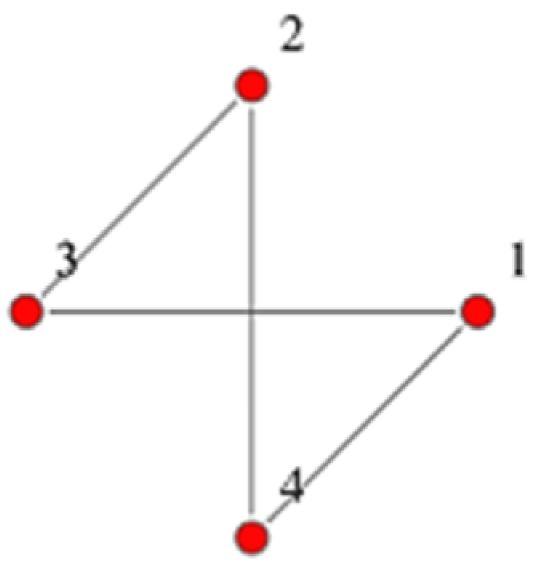
$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 & 1 \\ 3 & 0 & 0 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

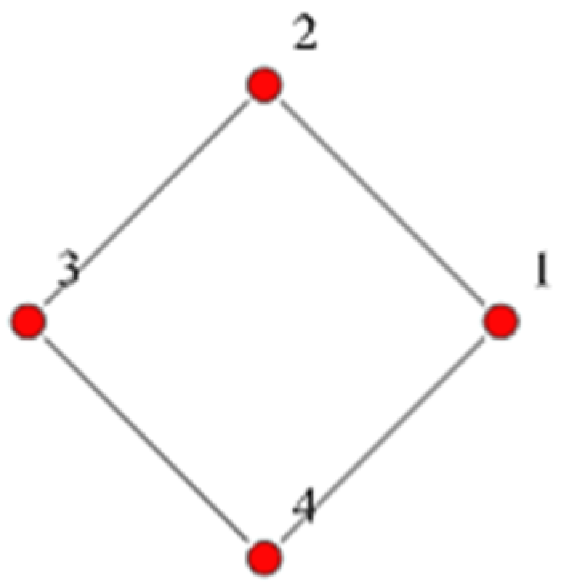
$$\begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

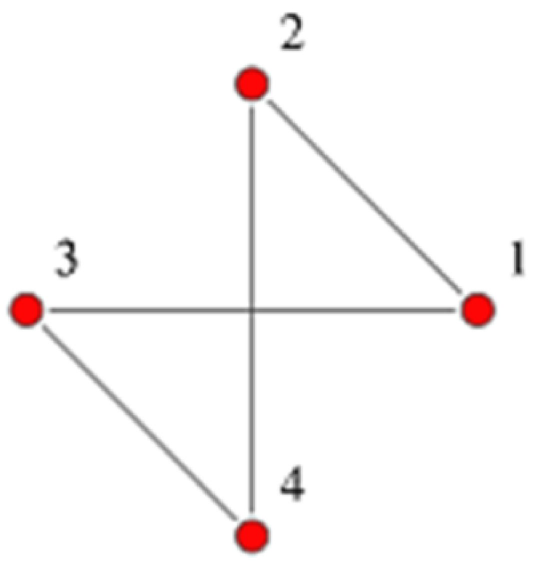
$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

In class exercise

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 0 & 1 & 1 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 0 \\ 4 & 1 & 1 & 0 & 0 \end{array} \qquad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$







What is Dijkstra's algorithm?

- Dijkstra's algorithm, a common approach in graph theory, can determine the shortest route between intersections on a road map.
- Dijkstra's algorithm efficiently finds the shortest path from a starting node (intersection) to a target node (another intersection) on a weighted graph (where weights represent distances).
- Here's a high-level explanation and a simple diagram to illustrate the concept:
  1. **Graph Representation**: Represent the intersections as nodes and the roads as edges with weights (distances).
  2. **Initialization**: Start from the initial intersection (source node). Set the distance to the source node to 0 and the distance to all other nodes to infinity.
  3. **Priority Queue**: First, use a priority queue to explore the nodes with the smallest known distance.
  4. **Relaxation**: For each node, update the distances to its adjacent nodes if a shorter path is found.
  5. **Repeat**: Continue until all nodes have been processed or the shortest distance to the target node is determined.

Dijkstra's Algorithm

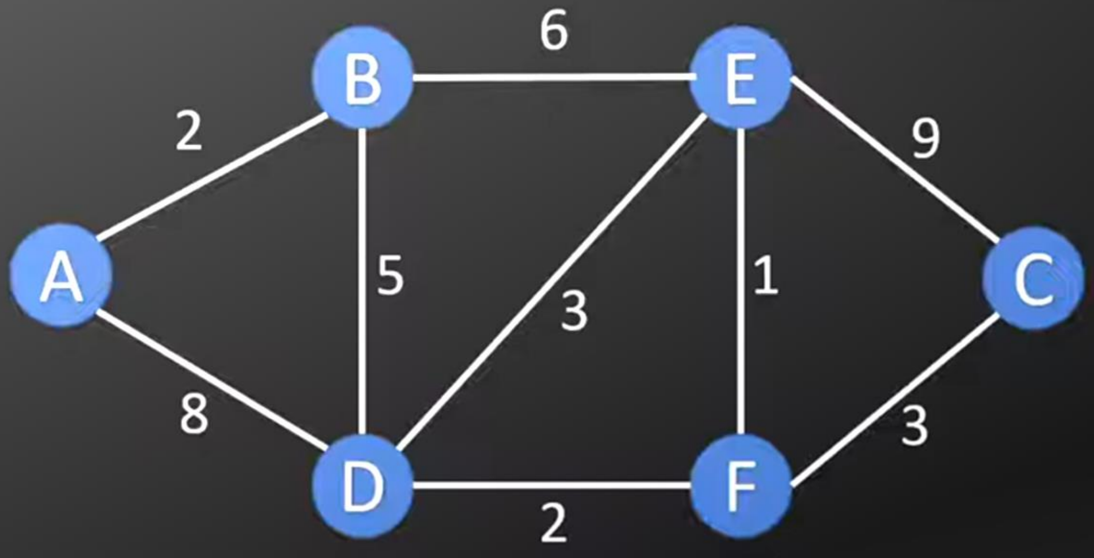
Input: A graph $G = (V, E)$ with non-negative edge weights, and a source vertex s.
Output: The shortest path from s to every vertex in V.

```
1. Initialize:
    • $d[v] \leftarrow \infty$ for all $v \in V$
    • $d[s] \leftarrow 0$
    • Create a priority queue $Q$ and insert all vertices $v \in V$ with priority $d[v]$
2. **While** $Q$ is not empty:
    • Extract the vertex $u$ with the minimum distance $d[u]$ from $Q$
    • For each neighbor $v$ of $u$:
        – **If** $d[u] + w(u, v) < d[v]$:
            ∗ $d[v] \leftarrow d[u] + w(u, v)$
            ∗ Update $v$'s priority in $Q$
```

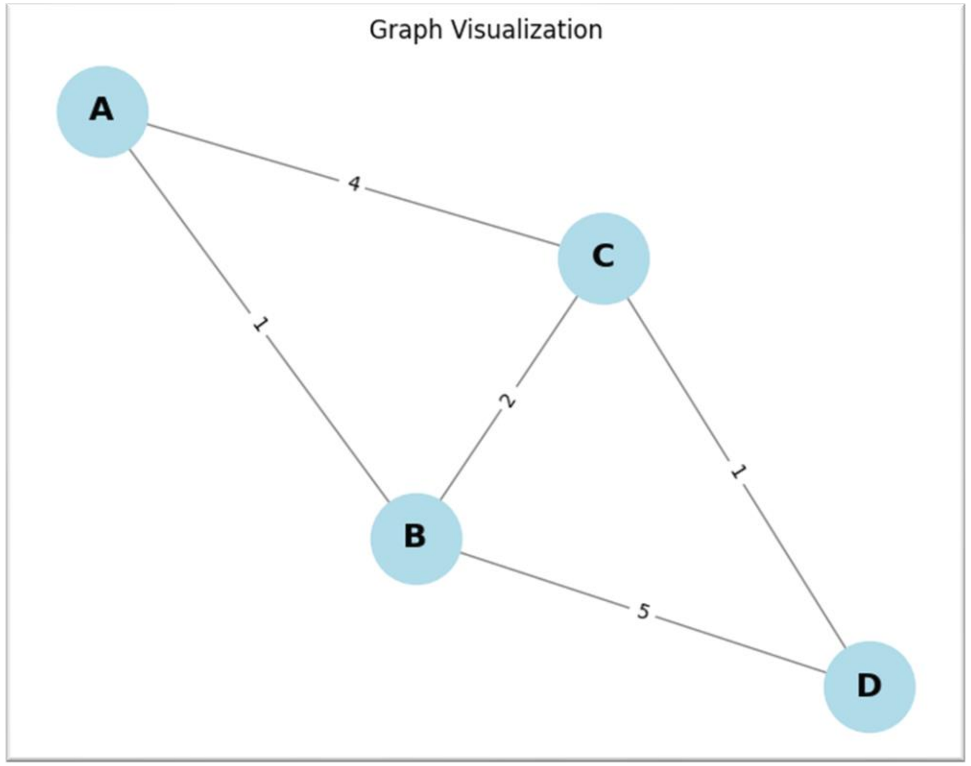
Return: The array d containing the shortest distances from s to each vertex.

Dijkstra’s Shortest Path Algorithm

Dijkstra's Shortest Path Algorithm



Simulation example



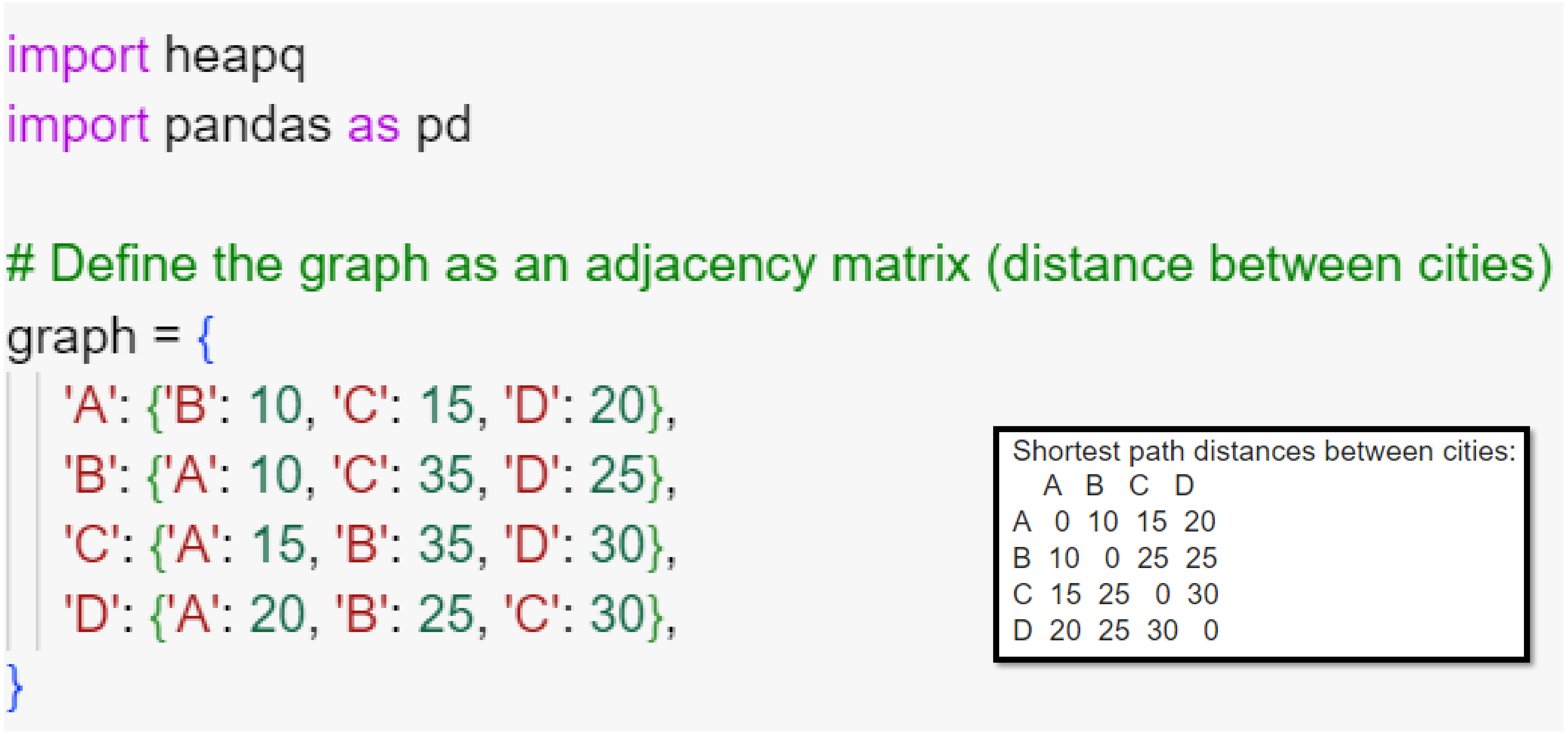
```
Shortest paths from node A:
```

	Node	Shortest Distance	Previous Node
0	A	0	None
1	B	1	A
2	C	3	B
3	D	4	C

Python Simulation

TSP using Dijkstra’s algorithm

- The Traveling Salesperson Problem (TSP) is typically solved using combinatorial optimization methods, such as dynamic programming, branch-and-bound, or approximation algorithms.
- Dijkstra’s algorithm is typically used to find the shortest path between two nodes in a graph. This involves finding the shortest route that visits every city exactly once and returns to the starting point.



```python
import heapq
import pandas as pd

# Define the graph as an adjacency matrix (distance between cities)
graph = {
    'A': {'B': 10, 'C': 15, 'D': 20},
    'B': {'A': 10, 'C': 35, 'D': 25},
    'C': {'A': 15, 'B': 35, 'D': 30},
    'D': {'A': 20, 'B': 25, 'C': 30},
}
```

```
Shortest path distances between cities:
    A   B   C   D
A   0  10  15  20
B  10   0  25  25
C  15  25   0  30
D  20  25  30   0
```

Python Simulation

python + colab

Thank You!