جامعة المستقبل
AL MUSTAQBAL UNIVERSITY

كلية العلوم

قسم الانظمة الطبية الذكية

Intelligent Medical Systems Department

Lap: (9)

Trees

Subject: Data Strectuer

Class: second

Lecturer: Prof.Dr. Mehdi Ebady Manaa

Programmer:- Fatima Hussein Jawad

Study Year: 2024-2025

Introduction to Trees

A **tree** is a hierarchical data structure consisting of nodes, where each node has:

1. **Data**: The value of the node.
2. **Children**: References to child nodes (if any).

The topmost node is called the **root**, and nodes without children are called **leaf nodes**. Trees are commonly used in areas like databases, file systems, and algorithms (e.g., search and sort).

Basic Terminologies

- **Root**: The topmost node in the tree.
- **Parent and Child**: Relationship between nodes where one is connected below another.
- **Leaf**: A node with no children.
- **Height**: The length of the longest path from the root to a leaf.
- **Depth**: The distance from the root to a specific node.

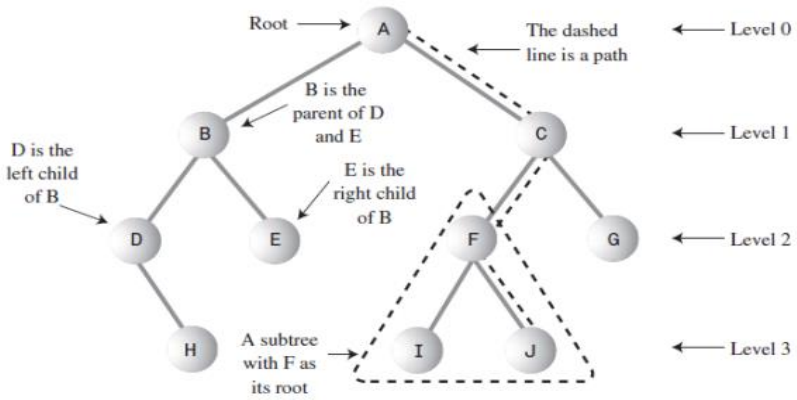

Figure 1: Tree terms. (H, E, I, J, and G are leaf nodes)

Binary Trees

A **binary tree** is a tree where each node has at most two children: **left** and **right**.

Python Implementation of Trees

1. Node Class:

```
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
```

2. Binary Tree Class:

```
class BinaryTree:
    def __init__(self):
        self.root = None
    # Insert a node into the binary tree
    def insert(self, data):
        if not self.root:
            self.root = Node(data)
        else:
            self._insert(self.root, data)
    def _insert(self, current, data):
        if data < current.data:  # Left subtree
            if current.left:
```

```
            self._insert(current.left, data)
        else:
            current.left = Node(data)
    else:  # Right subtree
        if current.right:
            self._insert(current.right, data)
        else:
            current.right = Node(data)
  # In-order traversal (Left -> Root -> Right)
  def in_order_traversal(self, node):
    if node:
        self.in_order_traversal(node.left)
        print(node.data, end=" ")
        self.in_order_traversal(node.right)
  # Pre-order traversal (Root -> Left -> Right)
  def pre_order_traversal(self, node):
    if node:
        print(node.data, end=" ")
        self.pre_order_traversal(node.left)
        self.pre_order_traversal(node.right)
  # Post-order traversal (Left -> Right -> Root)
  def post_order_traversal(self, node):
    if node:
```

```
        self.post_order_traversal(node.left)
        self.post_order_traversal(node.right)
        print(node.data, end=" ")
```

Tree Operations

1. Insert Data into Tree

```
# إنشاء الشجرة واختبار العمليات
tree = BinaryTree()
tree.insert(50)
tree.insert(30)
tree.insert(70)
tree.insert(20)
tree.insert(40)
tree.insert(60)
tree.insert(80)
```

2. Traversals

```
print("Tree created!")
print("In-order Traversal:")
tree.in_order_traversal(tree.root)  # Output: 20 30 40 50 60 70 80
print("\nPre-order Traversal:")
tree.pre_order_traversal(tree.root)  # Output: 50 30 20 40 70 60 80
print("\nPost-order Traversal:")
tree.post_order_traversal(tree.root)  # Output: 20 40 30 60 80 70 50
```

 # Output:

```
Tree created!
In-order Traversal:
20 30 40 50 60 70 80
Pre-order Traversal:
50 30 20 40 70 60 80
Post-order Traversal:
20 40 30 60 80 70 50
```

Additional Operations

Search for a Node

```
def search(node, key):
    if not node:
        return False
    if node.data == key:
        return True
    elif key < node.data:
        return search(node.left, key)
    else:
        return search(node.right, key)
print("\nSearch for 40 in the tree:", search(tree.root, 40))  # Output: True
print("Search for 90 in the tree:", search(tree.root, 90))  # Output: False
```

Find Minimum and Maximum

```
def find_min(node):
```

```
    current = node
    while current and current.left:
        current = current.left
    return current.data
def find_max(node):
    current = node
    while current and current.right:
        current = current.right
    return current.data
print("Minimum value in the tree:", find_min(tree.root))  # Output: 20
print("Maximum value in the tree:", find_max(tree.root))  # Output: 80
```

Delete a Node

```
def delete(node, key):
    if not node:
        return node
    if key < node.data:
        node.left = delete(node.left, key)
    elif key > node.data:
        node.right = delete(node.right, key)
    else:
        # Node with only one child or no child
        if not node.left:
            return node.right
```

```
    elif not node.right:
        return node.left
    # Node with two children: Get the inorder successor
    temp = find_min(node.right)
    node.data = temp
    node.right = delete(node.right, temp)
  return node
tree.root = delete(tree.root, 50)                # Deletes root (50)
print("\nIn-order traversal after deleting 50:")
tree.in_order_traversal(tree.root)                # Output: 20 30 40 60 70 80
```