Data Structures

Dilip Kumar Maripuri
Computer Applications

Data Structures

Session : Trees : Concepts, Classifications , Representations

Dilip Kumar Maripuri
Computer Applications

Trees

- A tree is a non-linear data structure that represents a hierarchical relationship between elements.
- It is a collection of nodes where each node stores a value and references to child nodes.
- A tree consists of nodes and edges and follows specific properties.
- Mathematically, a tree T is defined as a finite set of n nodes such that:
  - There is one special node called the **root**.
  - Each node (except the root) has exactly one **parent**.
  - A node can have zero or more **children**.
  - There is a unique path between any two nodes.
  - Trees do not contain **cycles** (i.e., they are *acyclic*).

Trees

- A tree T is a finite set of nodes such that:
  1. T is empty (i.e., $T = \emptyset$); or
  2. T consists of a distinguished node r, called the **root**, and zero or more non-empty subtrees $T_1, T_2, \ldots, T_k$, such that:
     - Each subtree is also a tree.
     - The root r is connected to the roots of all subtrees by edges.

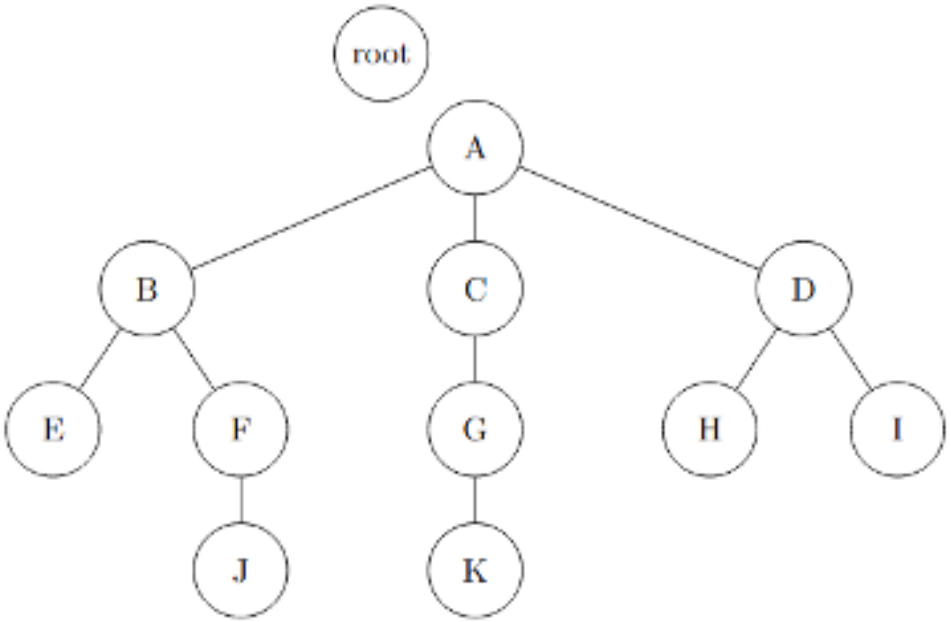
- Alternatively, a tree T can be represented as a graph $G = (V, E)$, where:
  - V is the set of vertices (nodes).
  - E is the set of edges connecting pairs of nodes.
- The graph G must satisfy the following properties:
  - **Connected**: There is a unique path between any two nodes in G.
  - **Acyclic**: G contains no cycles.

- **Characteristics of Trees**
  1. **Hierarchical Structure**: Unlike arrays and linked lists, trees organize data in a hierarchy.
  2. **Connected Graph without Cycles**: A tree is a connected acyclic graph.
  3. **Single Root**: There is exactly one root node.
  4. **Parent-Child Relationship**: Each node has exactly one parent (except the root) and can have multiple children.
  5. **Edges**: Connections between nodes are represented as edges.
  6. **No Loops**: Unlike graphs, trees cannot have cycles.

Basic Terminology in Trees

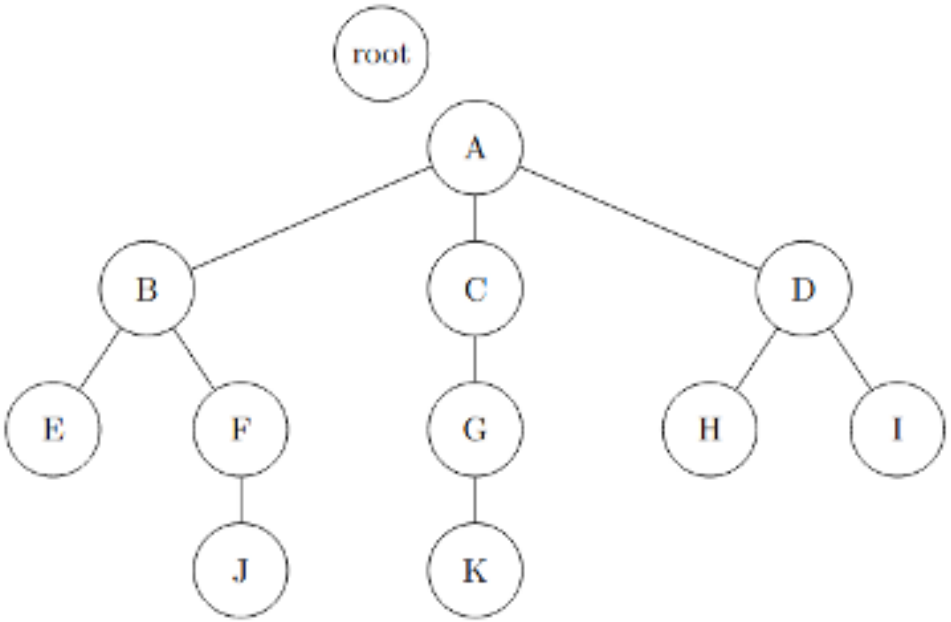
1. **Node**: The fundamental unit of a tree that contains data and links to child nodes.
2. **Root Node**: The topmost node in the tree. It has no parent.
3. **Parent Node**: A node that has children.
4. **Child Node**: A node that descends from another node (its parent).
5. **Leaf Node**: A node that has no children.
6. **Subtree**: A tree formed by a node and its descendants.
7. **Depth of a Node**: The number of edges from the root to that node.
8. **Height of a Node**: The longest path from that node to a leaf node.
9. **Degree of a Node**: The number of children a node has.
10. **Sibling Nodes**: Nodes that share the same parent.
11. **Path**: A sequence of nodes connected by edges.
12. **Ancestor of a Node**: Any node in the path from the root to the given node.
13. **Descendant of a Node**: Any node in the subtree rooted at the given node.

Trees



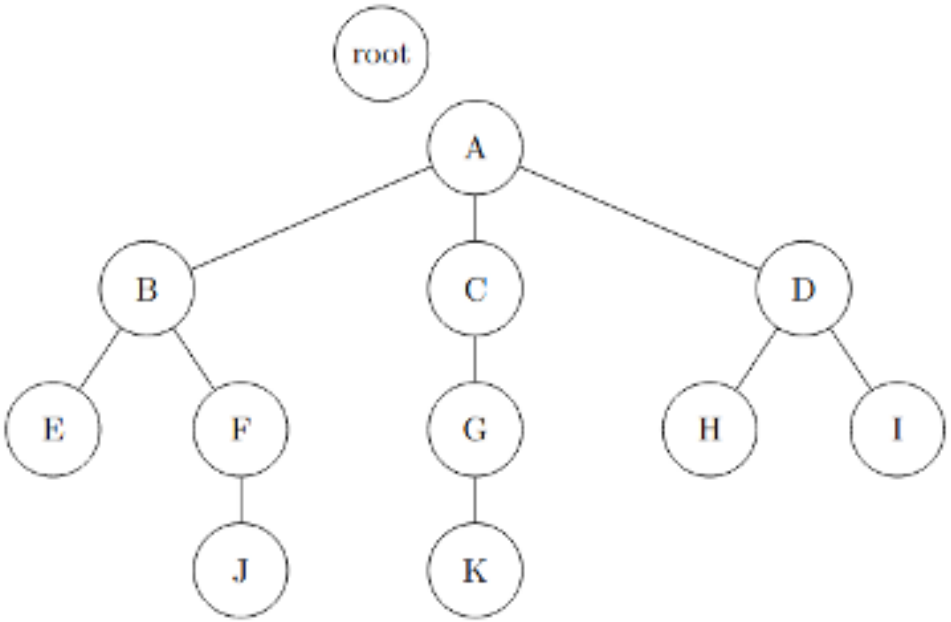
Node	Type(s)	Parent	Children	Siblings
A	Root, Internal, Parent	None	B, C, D	None
B	Internal, Parent, Child	A	E, F	C, D
C	Internal, Parent, Child	A	G	B, D

Trees



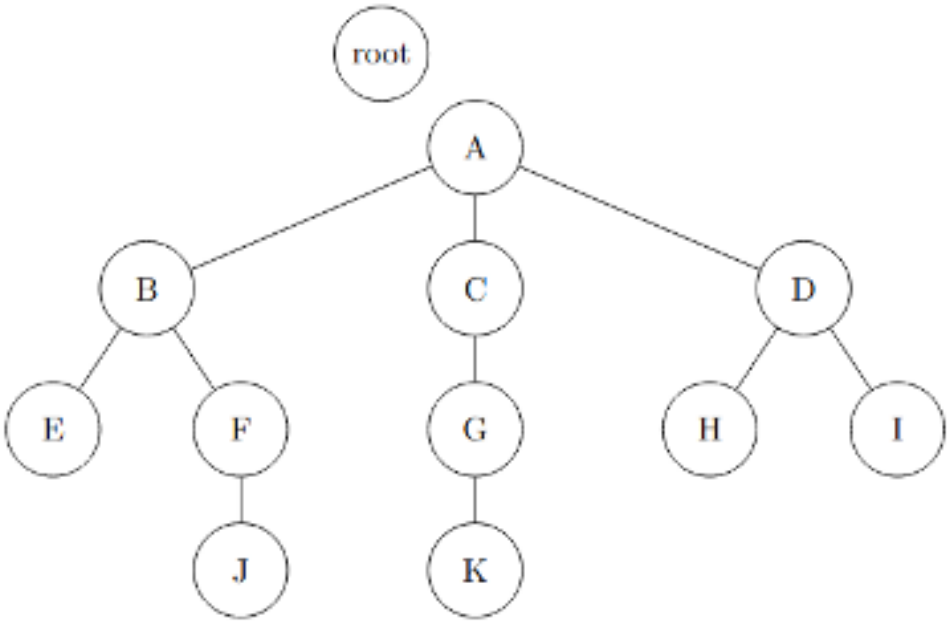
Node	Type(s)	Parent	Children	Siblings
D	Internal, Parent, Child	A	H, I	B, C
E	Leaf, Child	B	None	F
F	Internal, Parent, Child	B	J	E

Trees



Node	Type(s)	Parent	Children	Siblings
G	Internal, Parent, Child	C	K	None
H	Leaf, Child	D	None	I
I	Leaf, Child	D	None	H

Trees



Node	Type(s)	Parent	Children	Siblings
J	Leaf, Child	F	None	K
K	Leaf, Child	G	None	J

Trees

Node	Depth	Height	Degree	Ancestor(s)	Descendant(s)
A	0	3	3	None	B, C, D, E, F, G, H, I, J, K
B	1	2	2	A	E, F, J
C	1	2	1	A	G, K
D	1	2	2	A	H, I
E	2	0	0	A, B	None
F	2	1	1	A, B	J
G	2	1	1	A, C	K
H	2	0	0	A, D	None
I	2	0	0	A, D	None
J	3	0	0	A, B, F	None
K	3	0	0	A, C, G	None

Table: Node properties in the tree: depth, height, degree, ancestors, and descendants.

Property	Value	Explanation
Height of the Tree	3	The longest path from the root node (A) to any leaf node (J or K).
Depth of the Tree	3	The maximum depth of any node (e.g., J or K, which are 3 edges away from A).
Height of the Root Node	3	The height of the root node (A) is equal to the height of the tree.
Depth of the Root Node	0	The depth of the root node (A) is always 0 since it has no parent.

Table: Tree properties: height, depth, and explanation.

Trees : Depth vs. Height

- **For the Tree as a Whole:**
  - The height of the tree and the depth of the tree are equal.
  - This is because the longest path from the root to the deepest node (used for depth) is the same as the longest path from the root to the deepest leaf (used for height).
- **For Individual Nodes:**
  - Depth and height can differ.
  - For example:
    - Node B has:
    - Depth = 1 (distance from root A).
    - Height = 2 (distance to deepest leaf, J).

Trees : Depth vs. Height

- **Special Cases:**
  - For the root node:
    - Depth = 0 (no parent above it).
    - Height = Height of the tree (maximum height).
  - For leaf nodes:
    - Depth = Distance from root.
    - Height = 0 (no children below them).

Types of Trees

- **1. General Tree**

  Definition: A tree where each node can have an *unlimited number of children*.

  Examples: File systems, organizational hierarchies.

  Properties:

  - No restriction on the number of children.
  - The most flexible and generic tree structure.

Types of Trees

- **2. Skew Trees**

  A skew tree is a *degenerate tree* that behaves like a linked list because every node has only one child.
- **Left-Skewed Tree**

  - Every node has only a *left child*.
- **Right-Skewed Tree**

  - Every node has only a *right child*.

Types of Trees

- **3. Binary Tree**

  Definition: A tree where each node can have at most *two children*, referred to as *left* and *right*.

  Examples: Binary Search Tree (BST), AVL Tree.

  Properties:

  - Each node has 0, 1, or 2 children.
  - Commonly used for searching, sorting, and hierarchical representations.

Types of Trees

- **4. Ternary Tree**

  Definition: A tree where each node can have at most *three children*.

  Properties:

  - Children are referred to as *left*, *middle*, and *right*.
  - Commonly used in decision-making systems and Trinary Search Trees.

Types of Trees

- **5. N-ary Tree**

  Definition: A tree where each node can have at most *N children*.

  Examples: Quadtrees (4-ary trees), Octrees (8-ary trees).

  Applications:

  - Directory structures.
  - Spatial partitioning in computer graphics.

Types of Trees

- **6. Binary Search Tree (BST)**

  Definition: A binary tree where:

  - All nodes in the *left subtree* have values **less than** the root.
  - All nodes in the *right subtree* have values **greater than** the root.

  Examples: Used in databases and searching algorithms.

  Properties:

  - Efficient for searching, insertion, and deletion ($O(\log n)$) when balanced.
  - Inorder traversal yields a **sorted sequence**.

Types of Trees

- **Variants of Binary Trees**
  - **Strict Binary Tree:** Each node has either **exactly 2 children** or **no children**.
  - **Complete Binary Tree:** All levels except possibly the last are completely filled. Nodes in the last level are filled from left to right.
  - **Almost Complete Binary Tree:** Similar to a complete binary tree, but the last level may not satisfy the left-alignment property.
  - **Full Binary Tree:** Every node has either **0 children** or **2 children**.
  - **Balanced Binary Tree:** The height difference between left and right subtrees is at most 1 for every node.

Complete Vs Almost Complete Binary Trees

- **A complete binary tree follows these conditions:**
  - Every level except possibly the last is completely filled.
  - In the last level, all nodes must be as far left as possible (i.e., filling occurs from left to right without gaps).

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

- **Why is this a complete binary tree?**
  - Level 0 (Root): Filled (1).
  - Level 1: Filled (2,3).
  - Level 2 (Last Level): Nodes (4,5,6) are placed from left to right without gaps.

Complete Vs Almost Complete Binary Trees

- **An almost complete binary tree is similar to a complete binary tree, with the exception that:**
  - The last level may be incompletely filleds
  - Nodes are still added from left to right, but some spots may be skipped before the level is completely filled.

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

- **Why is this NOT a complete binary tree?**
  - The last level is not fully left-aligned. The expected position for node 6 should have been filled before moving right.
  - In a complete binary tree, node 6 should have been placed as the left child of node 3 before filling the right child.

Types of Trees

- **7. Balanced Trees**

  Balanced trees ensure that the tree height is logarithmic ($O(\log n)$) for n nodes, optimizing search and update operations.
- **AVL Tree:**
  - A self-balancing BST.
  - The height difference (balance factor) between the left and right subtrees is at most **-1, 0, or 1**.
  - Rotations (single or double) are used to maintain balance.
- **Red-Black Tree:**
  - A self-balancing BST with the following properties:
    - Every node is either **red** or **black**.
    - Root is always **black**.
    - No two consecutive red nodes.
  - Height is logarithmic, ensuring efficient operations.

Types of Trees

- **7. Balanced Trees**
- **B-Trees:**
  - A self-balancing **multiway search tree**.
  - Each node can have more than two children.
  - Commonly used in databases and file systems.

Types of Trees

- **8. Heap Trees Definition:** A binary tree with a **heap property**:
  - **Min-Heap:** Parent node's value is smaller than its children.
  - **Max-Heap:** Parent node's value is larger than its children.

Applications:

- Priority queues.
- Efficient sorting (Heap Sort).

- **1. Maximum Number of Nodes at a Given Level**

  Definition: In a binary tree, the maximum number of nodes at any level l is:2^l

  where l is the level number, starting from 0 for the root.

  Explanation:

  - At the root level ($l = 0$), there is only 1 node ($2^0 = 1$).
  - At level $l = 1$, there are $2^1 = 2$ nodes.
  - At level $l = 2$, there are $2^2 = 4$ nodes.
  - This exponential growth occurs because each node in a binary tree can have at most two children.
  - A binary tree with height 2 can have up to 7 nodes:
    - 1 node at level 0,
    - 2 nodes at level 1,
    - 4 nodes at level 2.

- **2. Maximum Number of Nodes in a Binary Tree of Height h**

  Definition: The maximum number of nodes in a binary tree of height h is given by: $2^{h+1} - 1$

  where h is the height of the tree, measured as the number of edges on the longest path from the root to the deepest leaf node, with the height of the root being considered 0.

  Explanation:

  - A binary tree with $h = 0$ (only the root) has $2^{0+1} - 1 = 1$ node.
  - A binary tree with $h = 1$ has $2^{1+1} - 1 = 3$ nodes.
  - A binary tree with $h = 2$ has $2^{2+1} - 1 = 7$ nodes.
  - A binary tree of height 3 can have a maximum of $2^{3+1} - 1 = 15$ nodes.

- **3. Minimum Possible Height or Levels in a complete Binary Tree**

  Definition: The minimum height h of a complete binary tree (or minimum number of levels) with n nodes is: $H_{min} = \lfloor \log_2(n) \rfloor$
- Minimum number of levels(Including Root Level)

  $L_{min} = \lfloor \log_2(n) \rfloor + 1$
  - For $n = 7$ nodes:

    $H_{min} = \lfloor \log_2(7) \rfloor = \lfloor 2.807 \rfloor = 2$

    $L_{min} = \lfloor \log_2(7) + 1 \rfloor = \lfloor 2.807 + 1 \rfloor = 3$

- **4. Perfect Binary Trees**

  Definition: A perfect binary tree is a tree that is both *full* and *complete*, meaning:

  - All interior (non-leaf) nodes have exactly two children.
  - All leaf nodes are at the same depth.
  - A perfect binary tree with k levels has exactly: $2^k - 1$ nodes.
  - All levels are completely filled.
  - A perfect binary tree with $k = 3$ levels has: $2^3 - 1 = 7$ nodes.
  - A perfect binary tree with $k = 4$ levels has: $2^4 - 1 = 15$ nodes.

Binary Tree Properties

Concept	Formula	Example for $h = 3$ or $n = 7$
Maximum Nodes at Level l	2^l	$2^2 =$ 4 nodes at level 2.
Maximum Nodes in Binary Tree of Height h	$2^{h+1} - 1$	$2^{3+1} - 1 = 15$.
Minimum Height for n Nodes	$\lceil \log_2(n+1) \rceil$	$\lceil \log_2(7+1) \rceil = 3$.
Nodes in a Perfect Binary Tree	$2^k - 1$	$2^3 - 1 = 7$.

Binary Tree Properties

- Relationship Between Leaf Nodes and Internal Nodes in a Full Binary Tree
- In a full binary tree, the number of leaf nodes (L) is always one more than the number of internal nodes (I): $L = I + 1$.

Trees - Array-Based Representation

- **Binary Trees** : A binary tree can be efficiently stored using arrays, especially when it is a *complete binary tree*.
- **Indexing Rules**
  - The root node is stored at **index 0** (or index 1 in some implementations for convenience).
  - For any node at index i:
    - **Left child** is at index $2i + 1$.
    - **Right child** is at index $2i + 2$.
    - **Parent** is at index $\lfloor \frac{i-1}{2} \rfloor$ (for $i > 0$).

Trees - Array-Based Representation

- **Advantages**
  - Fast access to nodes using indices.
  - Suitable for complete or almost complete binary trees.
- **Disadvantages**
  - Wastes space for skewed or sparse trees because the array may have many unused slots.
  - Hard to dynamically resize during insertion/deletion.

Trees - Array-Based Representation

- The array representation:

$$\text{Array: } [A, B, C, D, E]$$

  - A: Index 0.
  - B: Left child of A, index 1.
  - C: Right child of A, index 2.
  - D: Left child of B, index 3.
  - E: Right child of B, index 4.

Algorithm 1 InitializeTree

1. Start
2. Create an empty array *tree*
3. Return *tree*
4. Stop

Algorithm 2 Insert

1. Start
2. Input *tree*, *value*
3. Append *value* to the end of *tree*
4. Return updated *tree*
5. Stop

Algorithm 3 Find Left Child

1. Start
2. Input *tree*, *i*
3. Compute $leftIndex = 2 * i + 1$
4. If $leftIndex < \text{length}(tree)$, return $tree[leftIndex]$
5. Else, return NULL
6. Stop

Algorithm 4 Find Right Child

1. Start
2. Input *tree*, *i*
3. Compute $rightIndex = 2 * i + 2$
4. If $rightIndex <$ length(*tree*), return $tree[rightIndex]$
5. Else, return NULL
6. Stop

Algorithm 5 Find Parent

1. Start
2. Input *tree*, i
3. If $i = 0$, return NULL (root node has no parent)
4. Compute $parentIndex = (i - 1)/2$
5. Return $tree[parentIndex]$
6. Stop

Algorithm 6 Example Execution

1. Start
2. Call InitializeTree() to create *tree*
3. Insert values sequentially into *tree*: 1, 2, 3, 4, 5, 6
4. Find the left child of the root using LeftChild(*tree*, 0), should return 2
5. Find the right child of the root using RightChild(*tree*, 0), should return 3
6. Find the parent of the last inserted node using Parent(*tree*, 5), should return 2
7. Stop

```c
void insert(int tree[], int *size, int value
    )
{    tree[(*size)++] = value;     }

void display(int tree[], int size) {
    if (size == 0) {
        printf("Tree is empty!\n");
        return;
    }
    printf("Binary Tree (Level Order): ");
    for (int i = 0; i < size; i++)
        printf("%d ", tree[i]);
}
```

```
int parent(int index)
{    return (index - 1) / 2; }

int leftChild(int index)
{    return 2 * index + 1;    }

int rightChild(int index)
{    return 2 * index + 2;    }
```

```c
void displayNodeRelations(int tree[], int size, int index) {
    if (index < 0 || index >= size) {
        printf("Invalid index!\n");
        return;                    }
    printf("Node at index %d: %d\n", index, tree[index]);
    if (index > 0)
        printf("  Parent: %d\n", tree[parent(index)]);
    if (leftChild(index) < size)
        printf("  Left Child: %d\n", tree[leftChild(index)]);
    if (rightChild(index) < size)
        printf("  Right Child: %d\n", tree[rightChild(index)]);
}
```

Trees - Array-Based Representation

- **Non-Binary Trees** : For non-binary trees, array-based representation is less efficient because:
  - Nodes can have a variable number of children.
  - You would need to allocate a fixed number of slots for each node's children or use a separate array for each node.

  One alternative is to store the **first child** and **next sibling** of each node in the array:
  - **First child pointer:** Indicates the leftmost child.
  - **Next sibling pointer:** Indicates the next sibling of the current node.

Trees - Array-Based Representation

- Each entry in the array represents a node in the tree and contains the following:
  1. **Node Value:** Data of the node (e.g., A, B, C, etc.).
  2. **First Child Index:** The index of the first child of the node in the array. If the node has no children, this is `NULL` (or `-1`).
  3. **Next Sibling Index:** The index of the next sibling of the node in the array. If the node has no sibling, this is `NULL` (or `-1`).

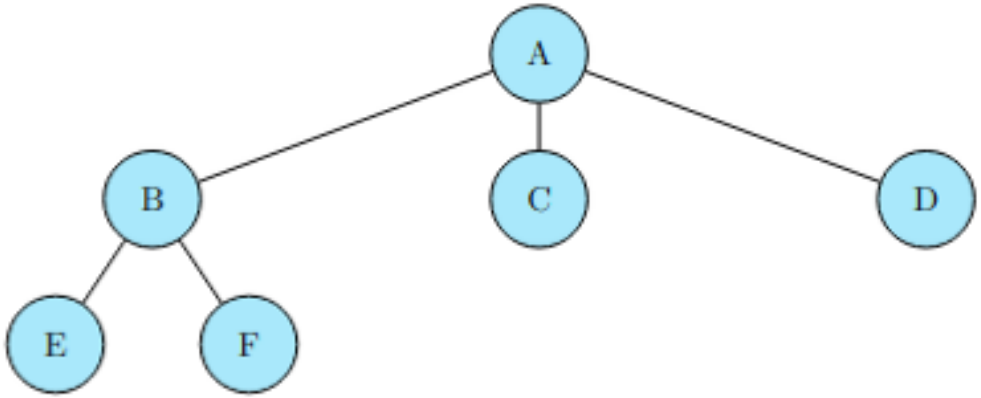
Linked List Representation

- **Binary Trees** Each node in a binary tree is represented as a structure containing:
  - Data or value of the node.
  - Pointer to the **left child**.
  - Pointer to the **right child**.
- **Advantages**
  - Space-efficient for sparse trees.
  - Easy to dynamically insert and delete nodes.
  - Flexible for various tree structures.

Linked List Representation

- **Non-Binary Trees** : For non-binary trees, each node needs:
  - Data or value of the node.
  - A pointer to its **first child**.
  - A pointer to its **next sibling**.

Linked List Representation



- Each node is represented as:
  - **Node A:** data $= A$, firstChild $= B$, nextSibling $= NULL$.
  - **Node B:** data $= B$, firstChild $= E$, nextSibling $= C$.
  - **Node C:** data $= C$, firstChild $= NULL$, nextSibling $= D$.
  - **Node E:** data $= E$, firstChild $= NULL$, nextSibling $= F$.

Binary Tree Representation

```
// Definition of a node in the binary tree using typedef
typedef struct mynode {
    int data;                    // Value of the node
    struct mynode* left;         // Pointer to the left child
    struct mynode* right;        // Pointer to the right child
} NODE;
```

Thank You

Dilip Kumar Maripuri
Associate Professor
Department of Computer Applications
dilip.maripuri@pes.edu
8073212026