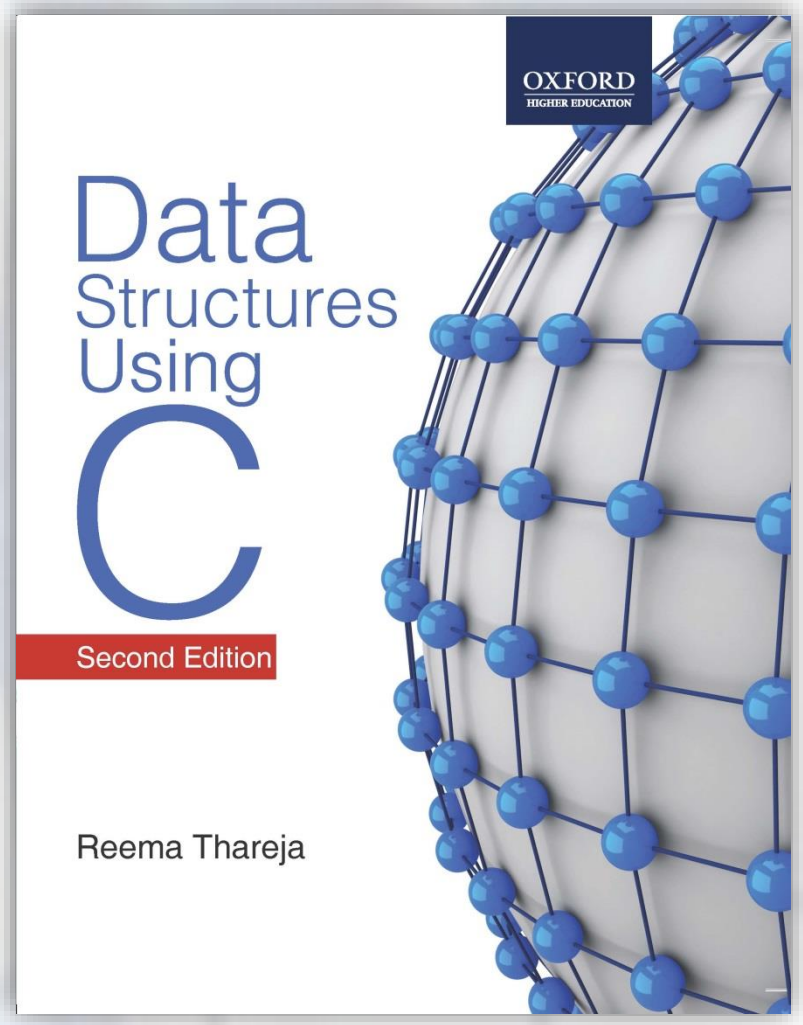
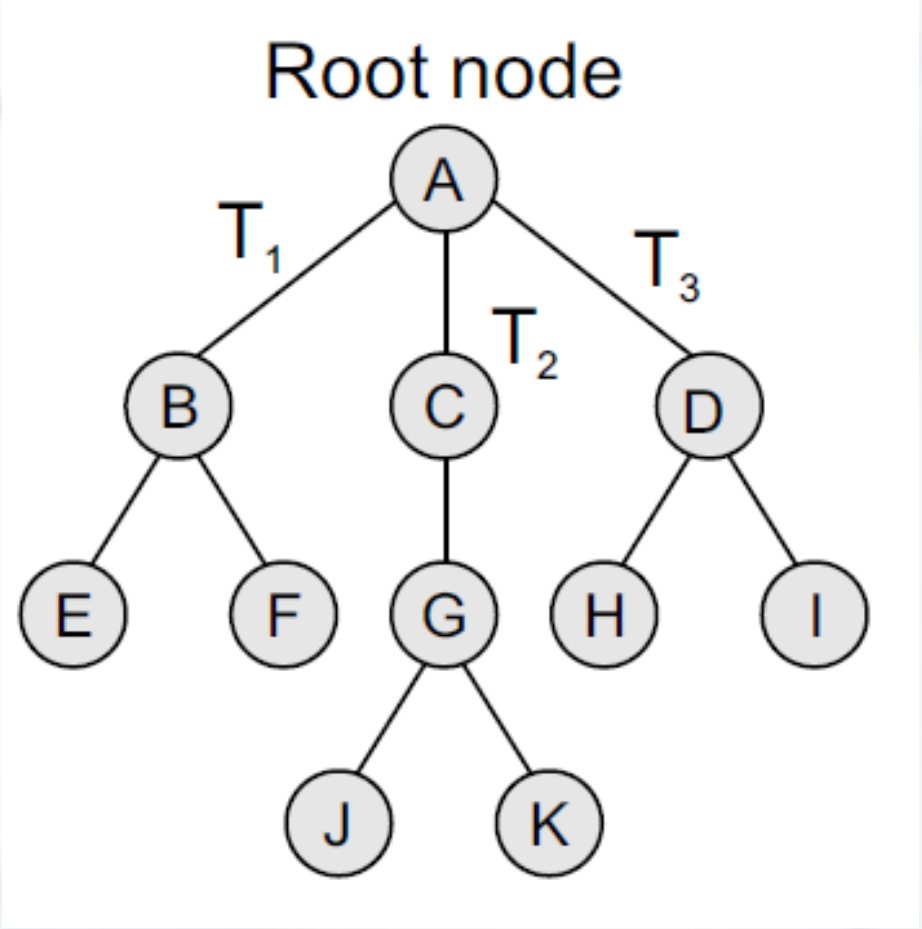

OXFORD
HIGHER EDUCATION
Data
Structures
Using
C
Second Edition
Reema Thareja

Trees

Trees

- A tree is recursively defined as a set of one or more nodes where one node is designated as the **root** of the tree and all the remaining nodes can be partitioned into non-empty sets each of which are in turn trees called as **sub-trees** of the root.
- A tree with no nodes is a *null tree*.



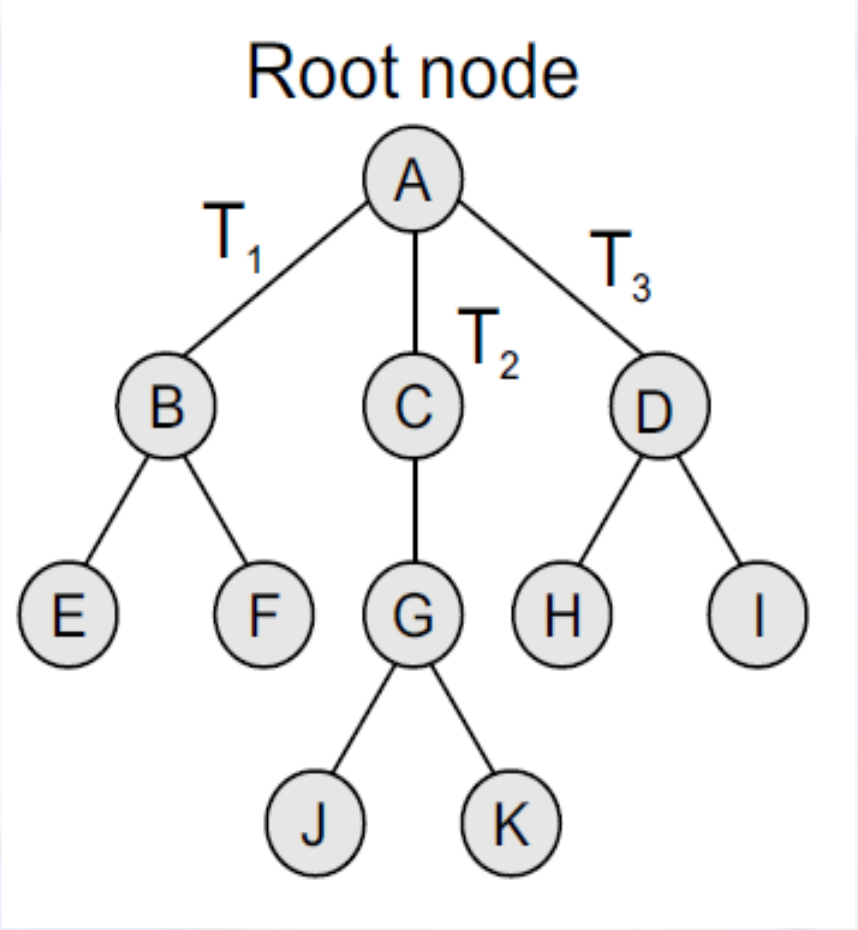
Trees (Wikipedia)

- In computer science, a **tree** is a widely used abstract data type that represents a hierarchical tree structure with a set of connected nodes. Each node in the tree can be connected to many children (depending on the type of tree), but must be connected to exactly one parent, except for the *root* node, which has no parent. These constraints mean there are no cycles or "loops" (no node can be its own ancestor), and also that each child can be treated like the root node of its own subtree, making recursion a useful technique for tree traversal.

Trees

Terminologies:

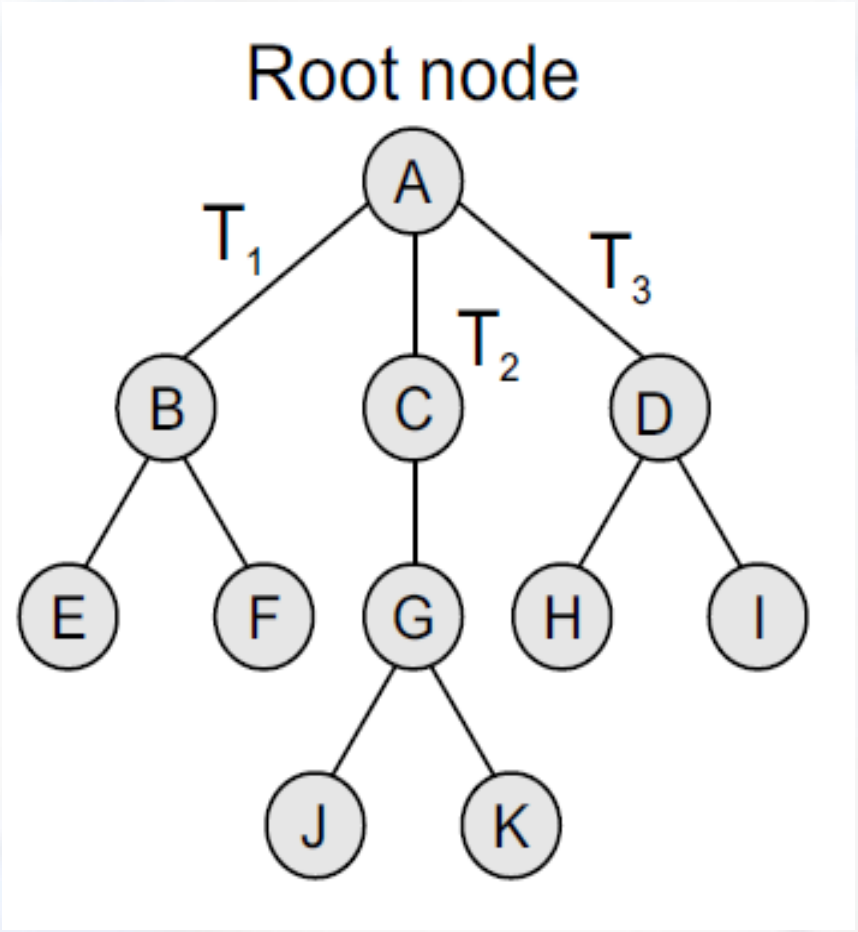
- ***Root node:*** The root node **A** is the topmost node in the tree.
- ***Sub-trees:*** If the root node ***A*** is not NULL, then the trees T1, T2, and T3 are called the sub-trees of ***A***.
- ***Leaf node:*** A node that has no children is called the leaf node or the terminal node.



Trees

Terminologies:

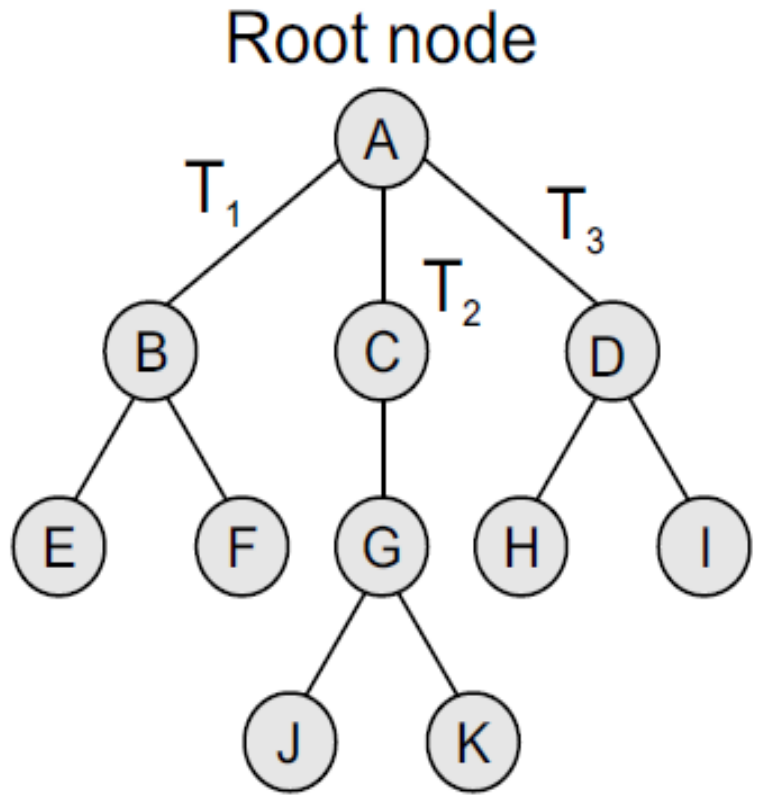
- ***Edge:*** *Connection between two nodes(Parent & child). Represented by the nodes at both ends (E.g. Edge CG)*
- ***Path*** **:** A sequence of consecutive edges is called a ***path****.*
- For example, the path from the root node **A** to node **I** is given as: A, D, and I.
- ***Ancestor node*** **:** An ancestor of a node is any predecessor node on the path from root to that node.
- The root node does not have any ancestor. In the tree given in Figure, nodes A, C, and G are the ancestors of node K.



Trees

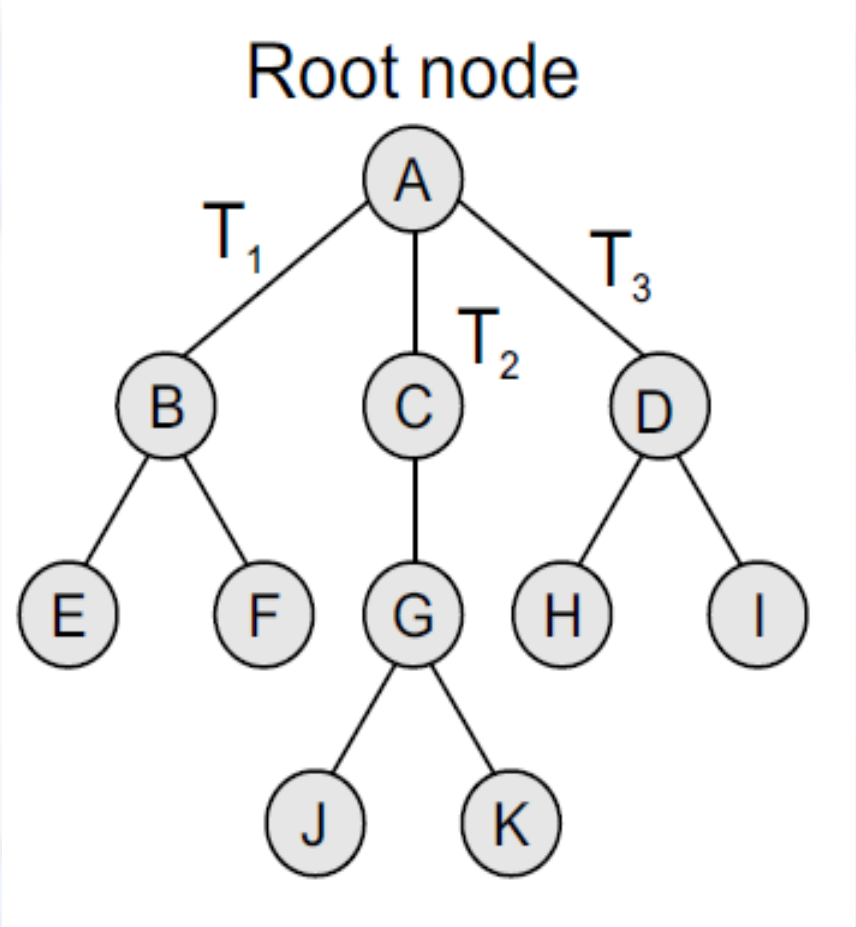
Terminologies:

- ***Descendant node:*** A descendant node of a given node is any successor node on the path from that node to a **leaf node**.
- **Leaf nodes** do not have any descendants.
- In the tree given, nodes C, G, J, and K are the descendants of node A.



Trees

- **Terminologies:**
- ***Degree:*** Degree of a node is equal to the number of children that a node has.
- The degree of a leaf node is zero.
- ***In-degree:*** In-degree of a node is the number of edges arriving at that node.
- In-degree of root node in Zero.
- ***Out-degree:*** Out-degree of a node is the number of edges leaving that node.

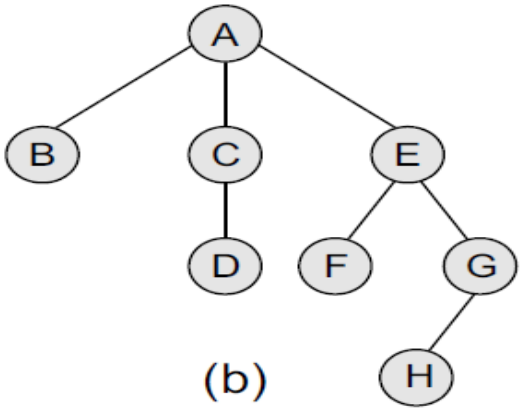


Types of Trees

Types of Trees

- General Trees
- Forests
- Binary Trees
- Expression Trees
- Tournament Trees

1. General Trees



- General trees are data structures that store elements hierarchically.
- The top node of a tree is the root node and each node, except the root, has a parent.
- A node in a general tree (except the leaf nodes) may have zero or more sub-trees.
- General trees which have at-most 2 sub-trees per node are called **binary trees**.
- General trees which have **at-most** 3 sub-trees **per node** are called **ternary trees**.
- However, the number of sub-trees for any node in a ternary tree may vary.
- For example, a node can have 1 sub-tree, whereas some other node can have 3 sub-trees.

2. Forests

- A **forest** is a disjoint union of trees.
- A set of disjoint trees (or forest) can be obtained by deleting the root and the edges connecting the root node to nodes at level 1.

- A forest can also be defined as an ordered set of zero or more general trees.

- We can convert a forest into a tree by adding a single node as the root node of the tree.

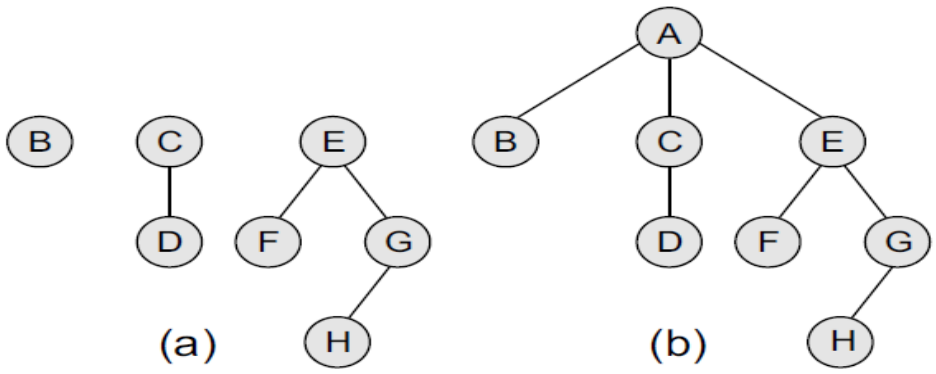
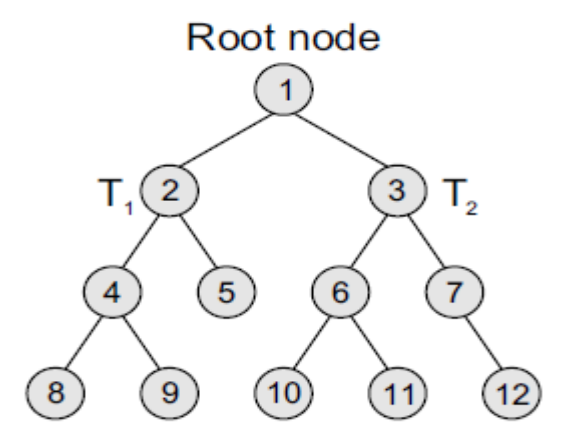


Figure 9.2 Forest and its corresponding tree

3. Binary Trees

- A **binary tree** is a tree where each nodes has **at most 2** child nodes.
- In a binary tree, the topmost element is called the root node, and each node has 0, 1, or at the most 2 children.
- Every node contains a data element, a "left" pointer which points to the left child, and a "right" pointer which points to the right child.
- The root element is pointed by a "root" pointer.
- If root = NULL, then it means the tree is empty.



R – Root node (node 1)

T_1- left sub-tree (nodes 2, 4, 5, 8, 9)

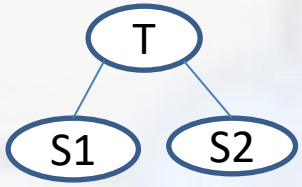
T_2- right sub-tree (nodes 3, 6, 7, 10, 11, 12)

Node structure of a binary tree

```
struct  nodetype{
        int key;
        struct nodetype *left;
        struct nodetype *right;
};
```

Binary Trees - Key Terms

- ***Parent:*** If N is any node in tree T that has *left node* S1 and/or *right node* S2, then N is called the *parent* of S1 and S2. Correspondingly, S1 and S2 are called the left child and the right child of N.
- Every node other than the root node has a parent.
- ***Sibling*****:** S1 and S2 are said to be *siblings*. In other words, all nodes that are at the same level and share the **same parent** are called *siblings*.
- ***Level number:*** Every node in the binary tree is assigned a *level number*. The root node is defined to be at level 0. The left and right child of the root node have a level number 1. Similarly, every node is at one level higher than its parents.
- ***Leaf node:*** A node that has no children.
- ***Degree:*** Degree of a node is equal to the number of children that a node has.

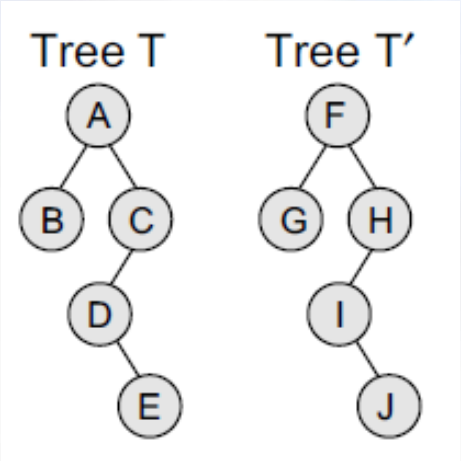


Binary Trees - Key Terms

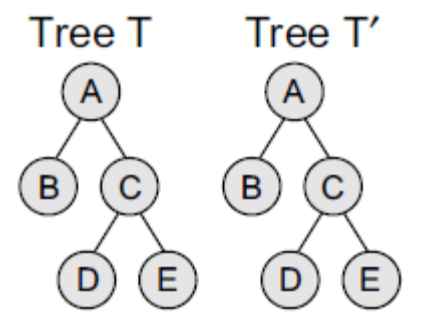
- ***In-degree*** of a node is the number of edges arriving at that node.
- ***Out-degree*** of a node is the number of edges leaving that node.
- ***Edge:*** It is the line connecting a node N to any of its successors
- ***Path:*** A **sequence of consecutive edges** is called a *path.*
- ***Depth:*** The *depth* of a node N is given as the **length of the path**(No. of edges) from the root to the node N. The **depth** of the **root node** is **zero**.
- ***Height:*** It is the total number of edges from the root node to the **deepest** node in the tree.
- **A tree with only a root node has a height of 0.**
- A binary tree of height **h** has at least **h+1** nodes and at most $\mathbf{2^{h+1} - 1}$ nodes.
- A binary tree with **n** nodes will have height at least $\mathbf{FLOOR(\log_2(n))}$ and at most **n-1**.

Binary Trees - Key Terms

- ***Similar binary trees:*** Given two binary trees T and T' are said to be similar if both these trees have the same structure.



- ***Copies of binary trees:*** Two binary trees T and T' are said to be *copies* if they have similar structure and same content at the corresponding nodes.

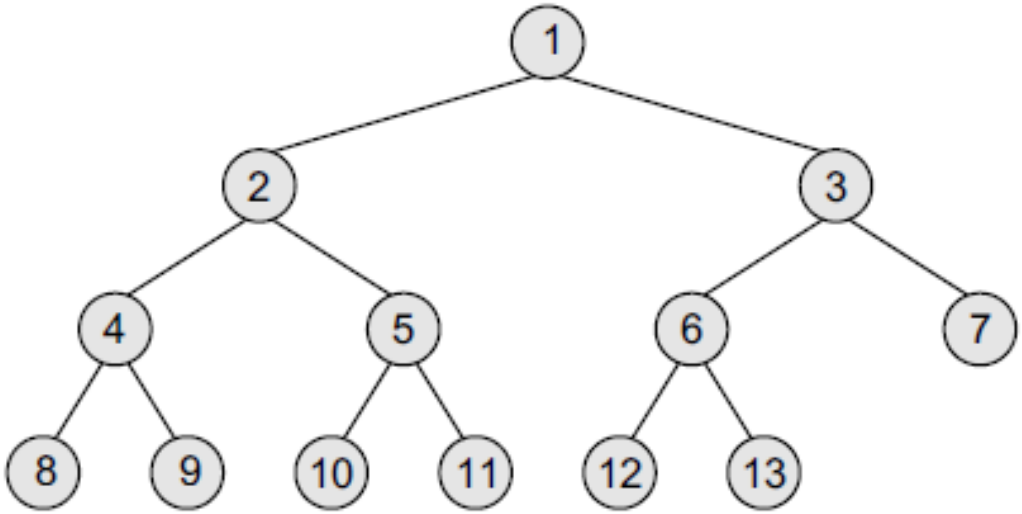


Complete Binary Trees

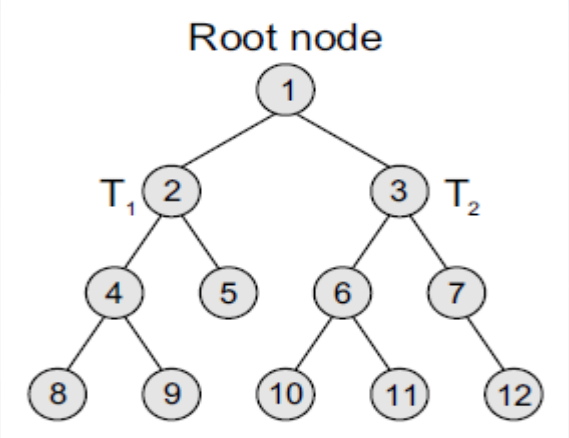
- A ***complete binary tree*** is a binary tree which satisfies two properties.

1. In a complete binary tree every level, except possibly the last, is completely filled.

2. All nodes appear as left as possible in the last level.



Complete binary tree



Not a complete binary tree

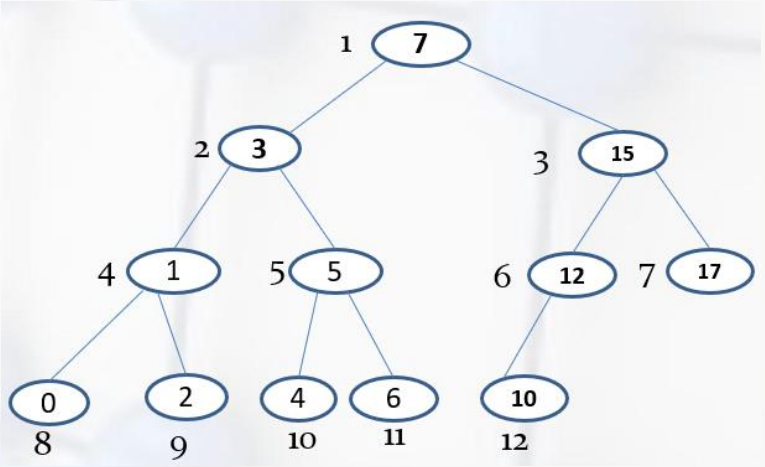
Complete Binary Trees

- In a **complete binary tree T,** a level l can have at most $\mathbf{2^l}$ **nodes**.
- The **height** of a complete binary tree T having **n** nodes is given as:
  FLOOR($\log_2(n)$)

  E.g., number of node n = 12

  $\log_2(12))$ = 3.584

  height = FLOOR(3.584) = 3



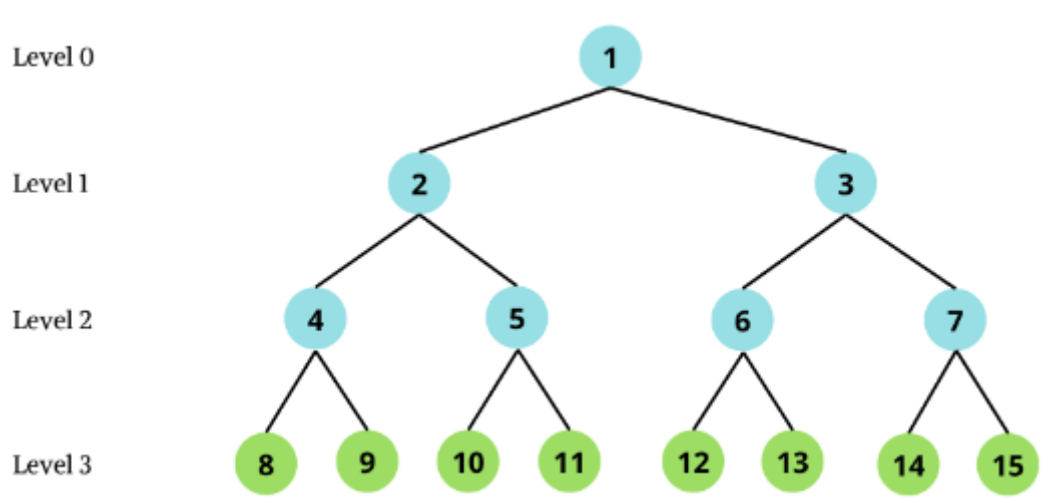
(Convention followed: Level of root node is 0, Height of tree with only node root node is 0)

Perfect Binary Trees

- A binary tree in which **all the leaf nodes are at the same depth/level**, and **all non-leaf nodes have exactly two children**.
- A perfect binary tree of height **h** will have:

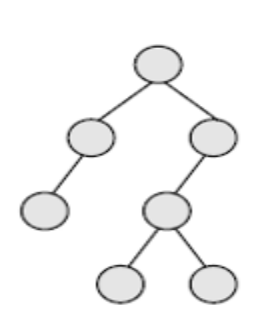
  ➔**Nodes n = $2^{h+1} - 1$ nodes**
- A perfect binary tree of **n** nodes will have:

  ➔ **Height h = FLOOR($\log_2(n)$)**
- Number of internal nodes with be 1 less than number of leaf nodes(external nodes)

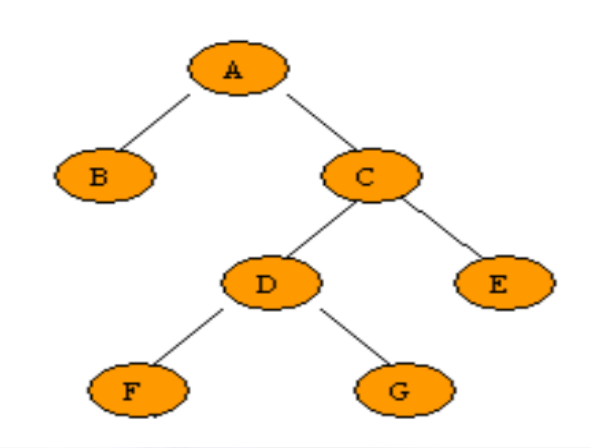


Strictly/Full Binary Trees

- A binary tree T is said to be a **strictly binary tree** (or a 2-tree) if each node in the tree has either **no child** or **exactly 2** children. Also known as **Full Binary Tree.**
- In a strictly binary tree nodes that have two children are called internal nodes and nodes that have no child or zero children are called external nodes(leaf nodes).
- If **strictly binary tree** has N nodes, the number leaf nodes = (N+1)/2
- The number non-leaf/internal nodes = FLOOR(N/2).
- **Number of leaf nodes(external nodes) with be 1 more than number of internal nodes.**
- If I is the number of internal nodes, total nodes will be I + (I+1) = 2*I+1
- A Perfect Binary Tree is a special type of Strictly Binary Tree.



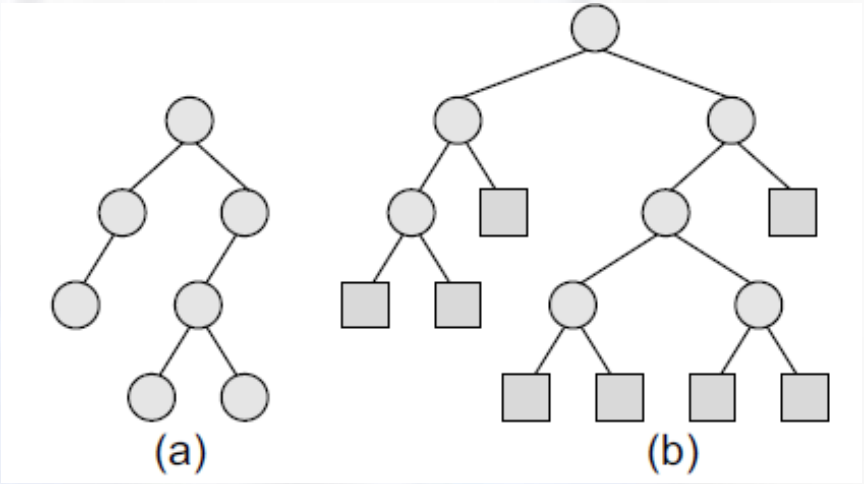
Not a strictly binary tree



A strictly binary tree

Extended Binary Trees

- Extended binary tree is a type of binary tree in which all the **null sub-tree** of the original tree are replaced with special nodes called **external nodes** whereas other nodes are called **internal nodes**
- In an extended binary tree, nodes that have two children are called **internal nodes** and nodes that have no children are called **external nodes**.
- In the figure internal nodes are represented using a circle and external nodes are represented using squares.



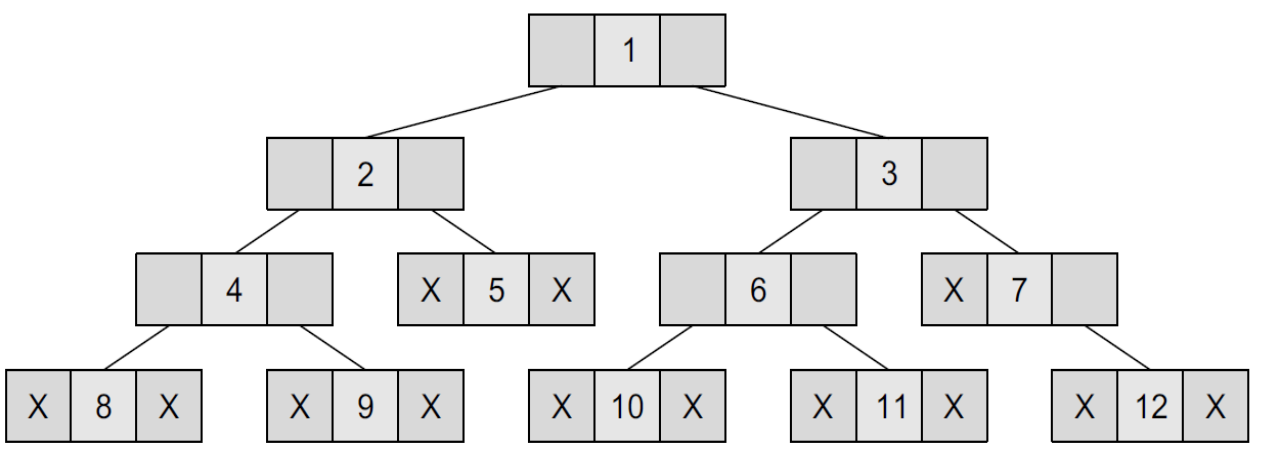
Binary tree

Corresponding Extended binary tree

Linked Representation of Binary Trees

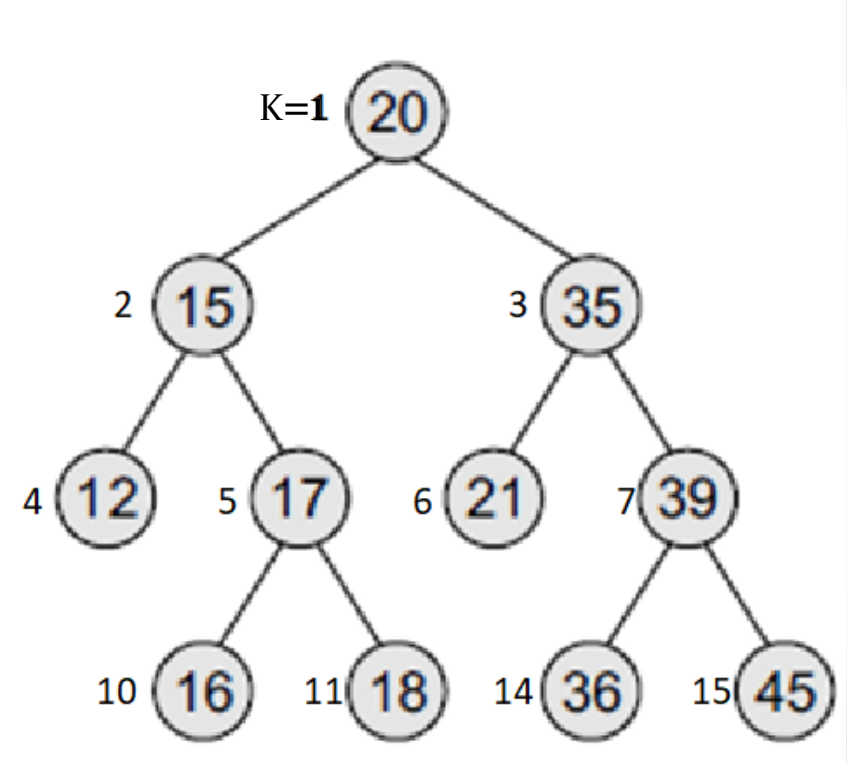
- In computer's memory, a binary tree can be maintained either using a linked representation or using sequential/array representation.
- In linked representation of binary tree, every node will have three parts: the data element, a pointer to the left node and a pointer to the right node. So in C, the binary tree is built with a node type given as below.

```
struct node
    {
       struct node* left;
       int data;
       struct node* right;
    };
```



Sequential Representation of Binary Trees

- Sequential representation of trees is done using a one-dimensional array (Assumes array indexing starts with 1).
- Children (if present) of **node number K** will be at index 2*K (left child) and 2*K+1 (right child).

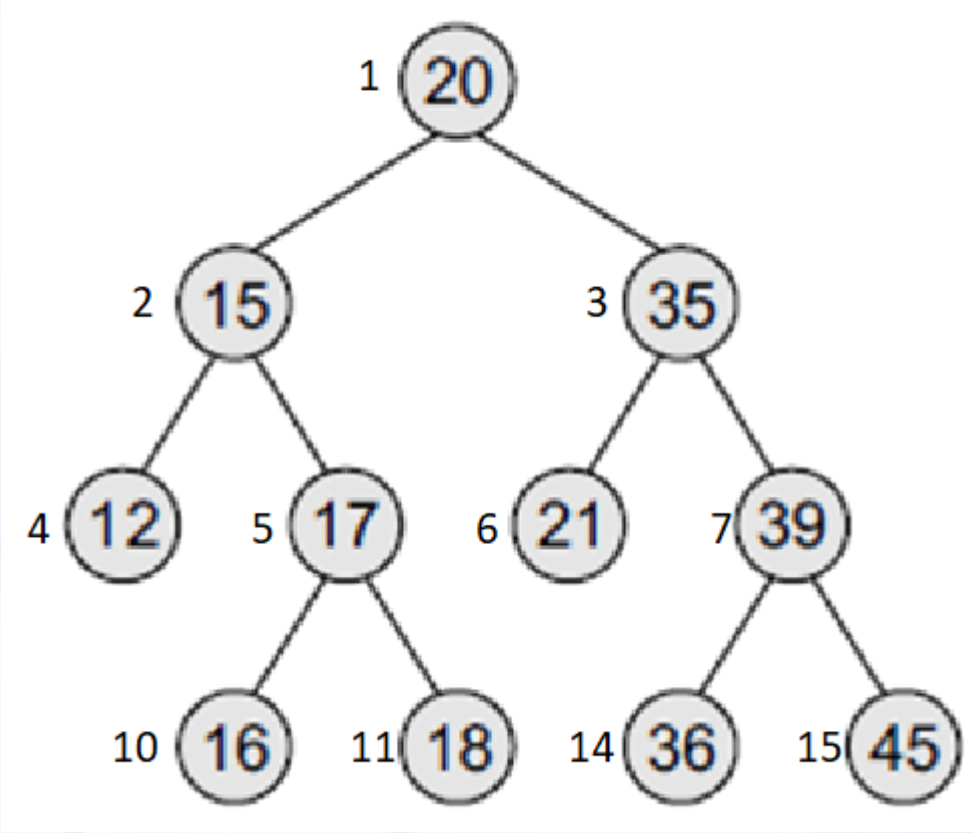


1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

- If array indexing start from 0, for node K, left child is at 2K+1 and right child at 2K+2.

Sequential Representation of Binary Trees

- Array-based sequential representation is inefficient as many locations may not be utilized.
- The maximum size of the array TREE is given as $(2^{h+1}-1)$, where h is the height of the tree. (Assuming height tree with only 1 node is 0)

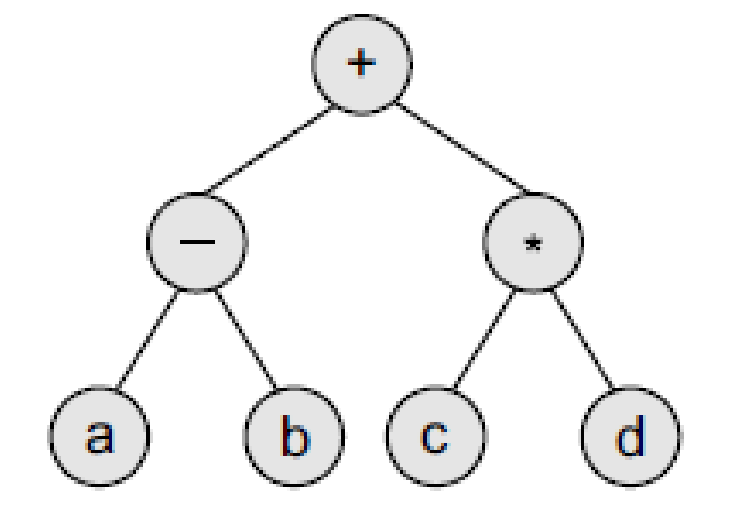


1	20
2	15
3	35
4	12
5	17
6	21
7	39
8	
9	
10	16
11	18
12	
13	
14	36
15	45

4. Expression Trees

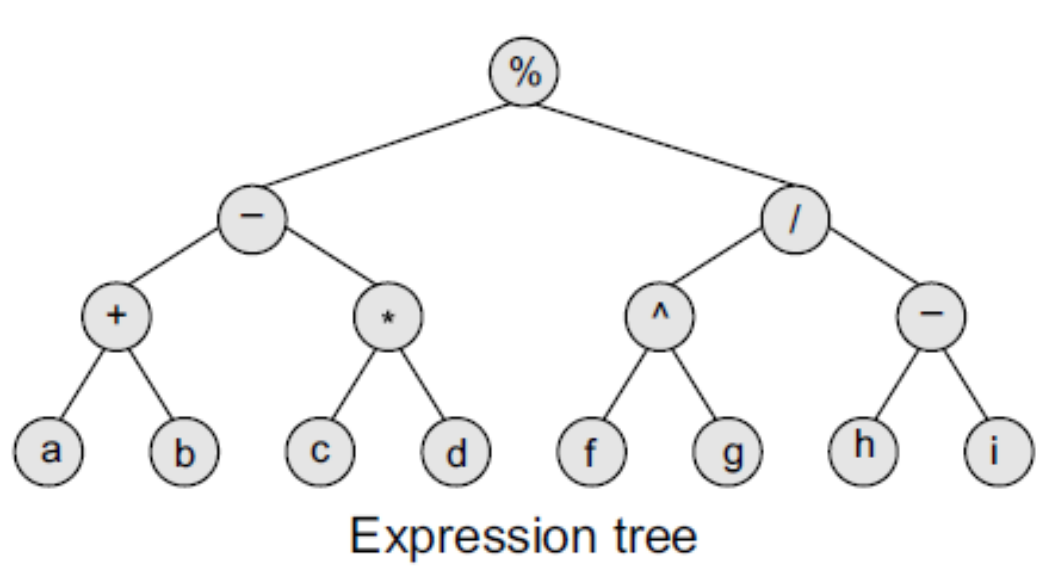
- Binary trees are widely used to store/represent algebraic expressions.
- For example, consider the algebraic expression Exp given as:

  Exp = (a – b) + (c * d)
- This expression can be represented using a binary tree as shown in figure:



4. Expression Trees

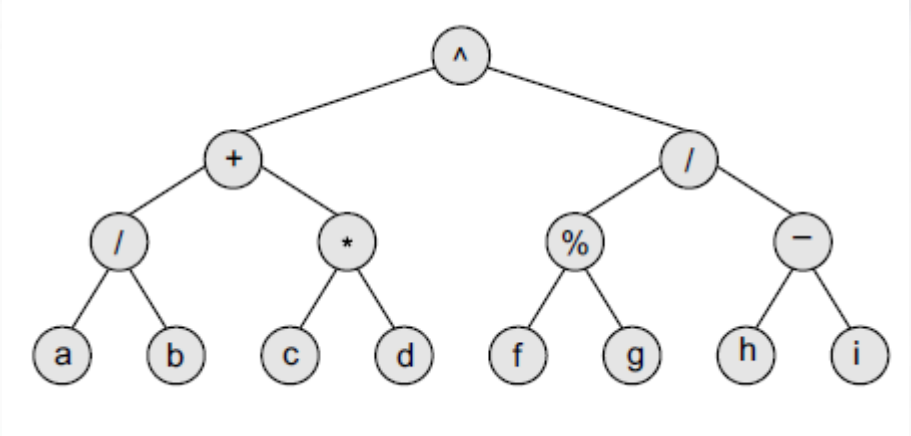
- Exp = `((a + b) - (c * d)) % ((e ^f) / (g - h))`
- This expression can be represented using a binary tree as shown in figure.
- All operands are in leaf nodes, operators in internal nodes.



Expression tree

4. Expression Trees

- For the expression tree is shown in following figure:



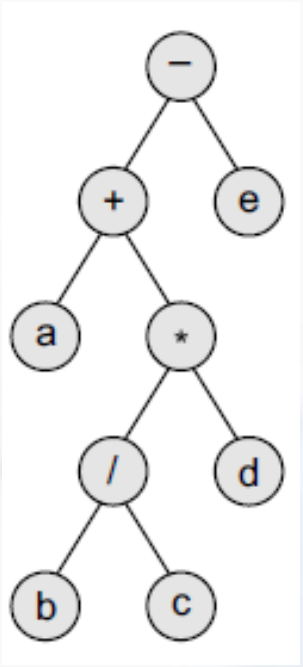
- Corresponding expression is: `((a/b) + (c*d)) ^ ((f % g)/(h – i))`

4. Expression Trees

- Exp = `a + b / c * d – e`
- Construct the expression tree.
- Use the operator precedence chart to find the sequence in which operations will be performed.
- Build tree in a top-down approach starting with last operation as the root.

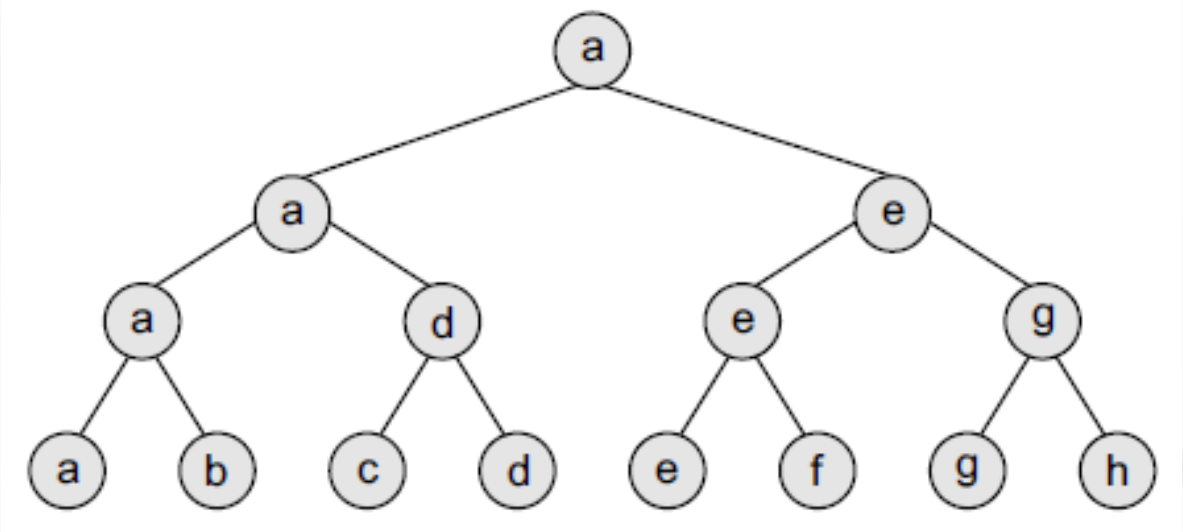
4. Expression Trees

- Exp = `a + b / c * d – e`
- Construct the expression tree.
- Use the operator precedence chart to find the sequence in which operations will be performed.
- Build tree in a top-down approach starting with last operation as the root.



Tournament Trees

- In a tournament tree (also called a selection tree), each leaf node represents a player
- Each internal node represents the winner of the match played between the players represented by its child nodes.
- These tournament trees are also called **winner trees** because they are used to record the winner at each level.



Traversing a Binary Tree

- Traversing a binary tree is the process of **visiting each node** in the tree **exactly once** in a systematic way.
- There are three most common algorithms for tree traversals, which differ in the order in which the nodes are visited.
- These algorithms are:
  - ✓ Pre-order traversal
  - ✓ In-order traversal
  - ✓ Post-order traversal

Pre-order Algorithm

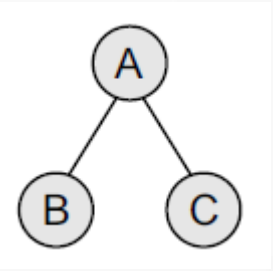
- To traverse a non-empty binary tree in preorder, the following operations are performed **recursively at each node**.
- The algorithm starts with the root node of the tree and continues by:

Preorder:

✓ Visiting the root node

✓ Traversing the left subtree in Preorder

✓ Traversing the right subtree in Preorder



Pre-order traversal: A, B, C

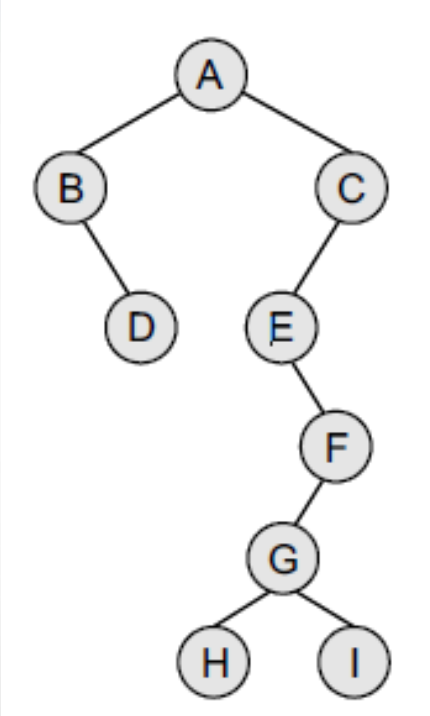
Pre-order Algorithm

- To traverse a non-empty binary tree in preorder, the following operations are performed recursively at each node.
- The algorithm starts with the root node of the tree and continues by:

✓ Visiting the root node

✓ Traversing the left subtree in preorder

✓ Traversing the right subtree in preorder

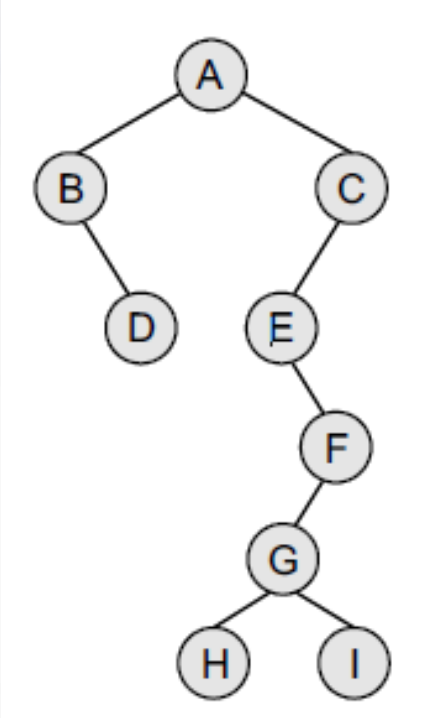


Preorder traversal:
A, B, D, C, E, F, G, H and I

Pre-order Algorithm

- Algorithm for pre-order traversal

```
PREORDER(TREE):
        IF TREE == NULL
                RETURN
        ELSE
                OUTPUT TREE->DATA
                PREORDER(TREE->LEFT)
                PREORDER(TREE->RIGHT)
```



A, B, D, C, E, F, G, H , I

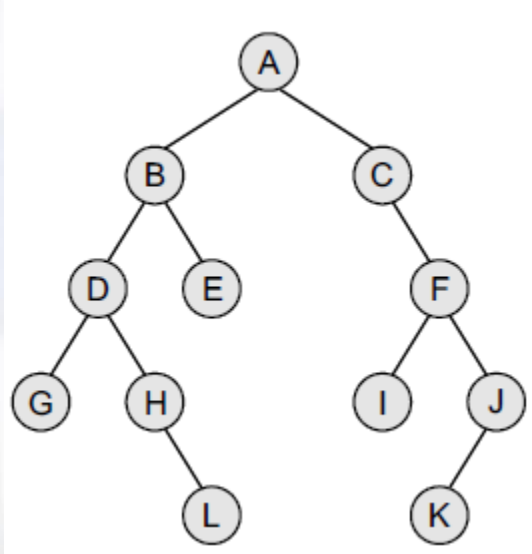
Pre-order Algorithm

- The algorithm starts with the root node of the tree and continues by:

✓ Visiting the root node

✓ Traversing the left subtree in preorder

✓ Traversing the right subtree in preorder



Preorder traversal:
??

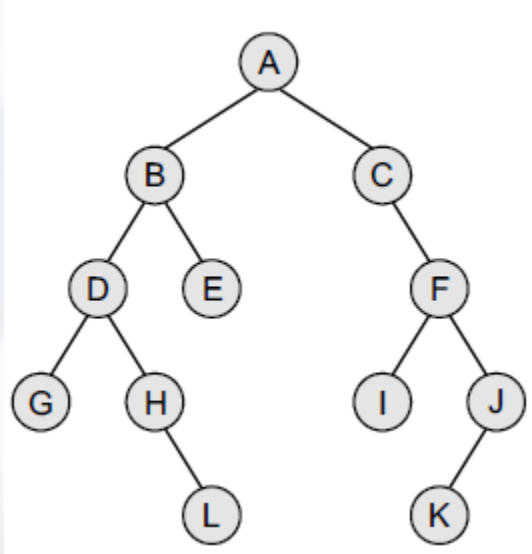
Pre-order Algorithm

- The algorithm starts with the root node of the tree and continues by:

✓ Visiting the root node

✓ Traversing the left subtree in preorder

✓ Traversing the right subtree in preorder



Preorder traversal:

A, B, D, G, H, L, E, C, F, I, J,K

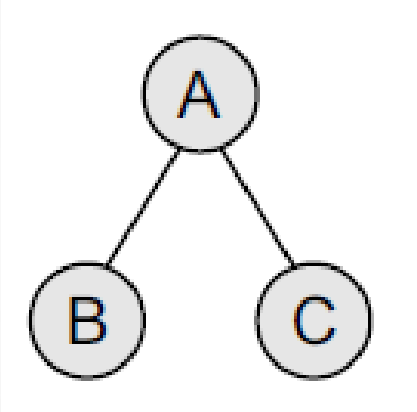
In-order Algorithm

- To traverse a non-empty binary tree in **in-order**, the following operations are performed **recursively at each node.**
- The algorithm starts with the root node of the tree and continues by,

✓ Traversing the left subtree in In-order

✓ Visiting the root node

✓ Traversing the right subtree in In-order



In-order Traversal: B, A ,C

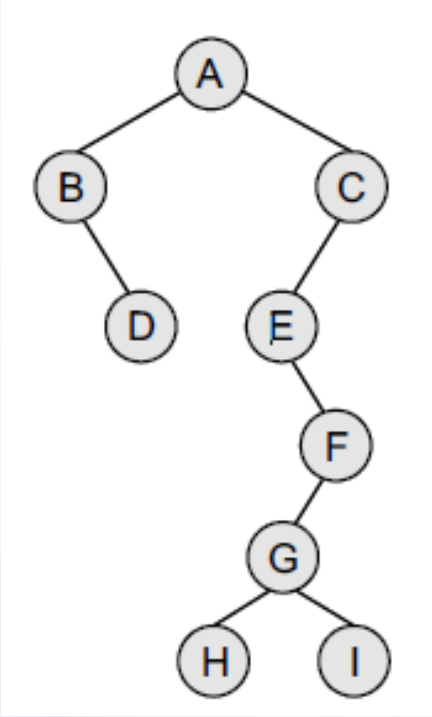
In-order Algorithm

- To traverse a non-empty binary tree in **in-order**, the following operations are performed **recursively** at each node.
- The algorithm starts with the root node of the tree and continues by,

✓ Traversing the left subtree in In-order

✓ Visiting the root node

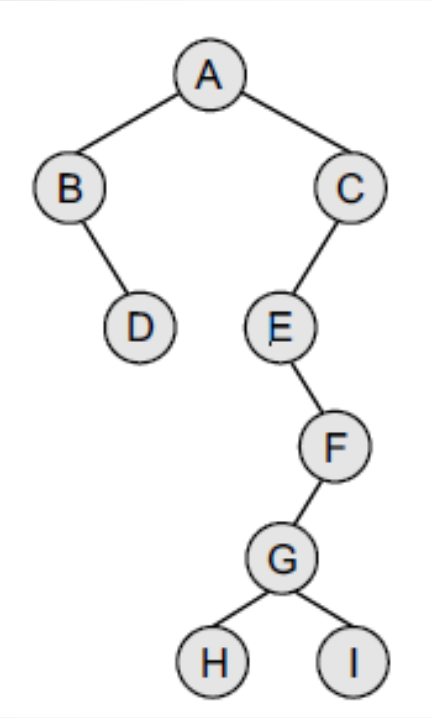
✓ Traversing the right subtree in In-order



In-order Algorithm

- Algorithm for in-order traversal

```
INORDER(TREE):
        IF TREE == NULL
                RETURN
        ELSE
                INORDER(TREE->LEFT)
                OUTPUT TREE->DATA
                INORDER(TREE->RIGHT)
```

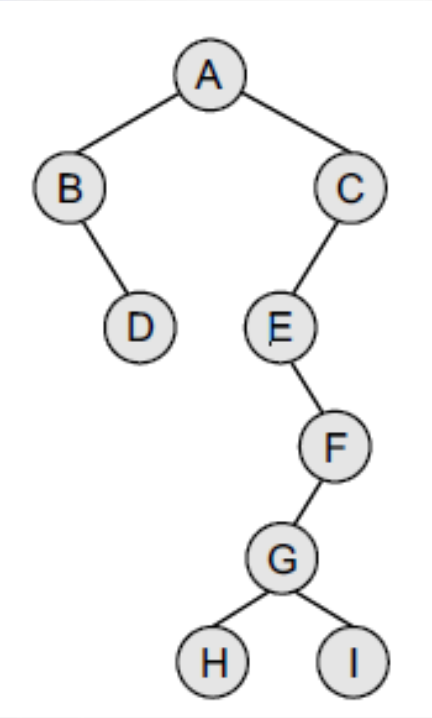


In-order traversal:
B, D, A, E, H, G, I, F, C

In-order Algorithm

- Algorithm for in-order traversal

```
INORDER(TREE):
        IF TREE == NULL
                RETURN
        ELSE
                INORDER(TREE->LEFT)
                OUTPUT TREE->DATA
                INORDER(TREE->RIGHT)
```



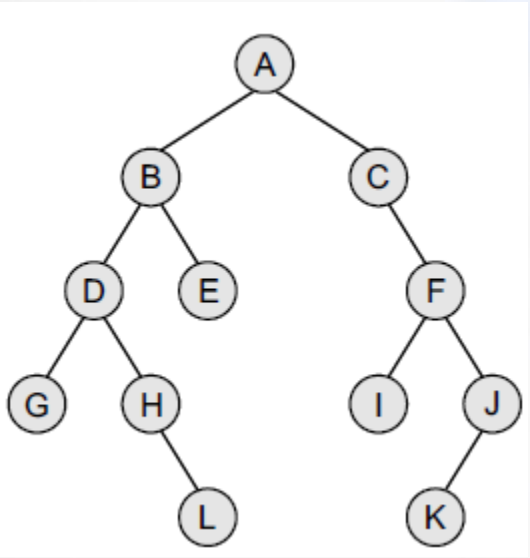
In-order traversal: ??

In-order Algorithm

- The algorithm starts with the root node of the tree and continues by:

- Algorithm for in-order traversal

```
INORDER(TREE):
        IF TREE == NULL
                RETURN
        ELSE
                INORDER(TREE->LEFT)
                OUTPUT TREE->DATA
                INORDER(TREE->RIGHT)
```



In-order traversal:
??

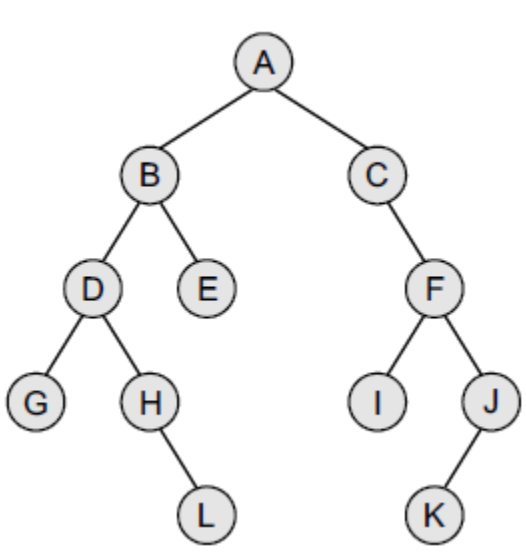
In-order Algorithm

- The algorithm starts with the root node of the tree and continues by:

✓ Traversing the left subtree in In-order

✓ Visiting the root node

✓ Traversing the right subtree in In-order



In-order traversal:
G, D, H,L, B, E, A, C, I, F, K, J

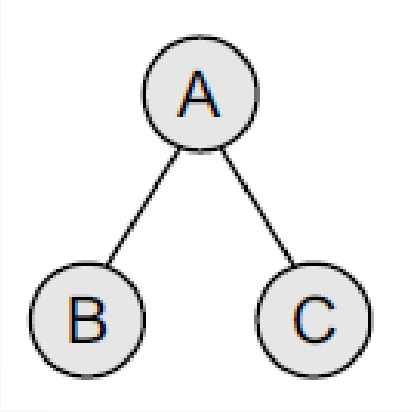
Post-order Algorithm

- To traverse a non-empty binary tree in **post-order**, the following operations are performed recursively at each node.
- The algorithm starts with the root node of the tree and continues by,

✓ Traversing the left subtree post-order

✓ Traversing the right subtree post-order

✓ Visiting the root node



Post-order: B,C,A

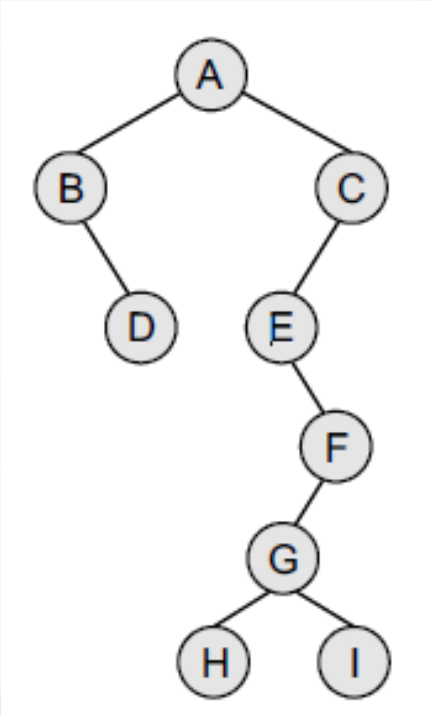
Post-order Algorithm

- To traverse a non-empty binary tree in **post-order**, the following operations are performed recursively at each node.
- The algorithm starts with the root node of the tree and continues by,

✓ Traversing the left subtree post-order

✓ Traversing the right subtree post-order

✓ Visiting the root node

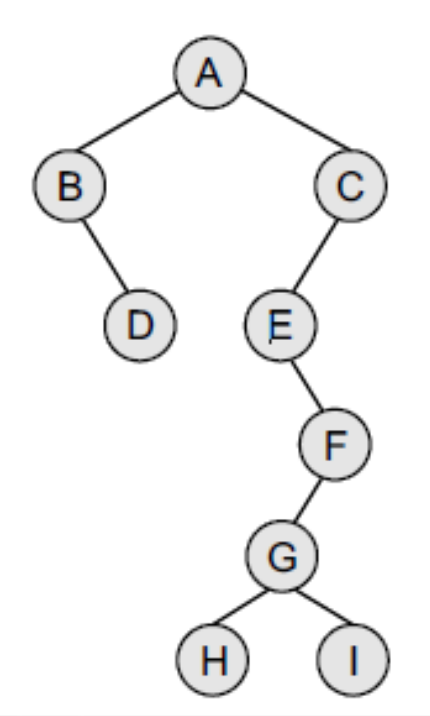


D, B, H, I, G, F, E, C and A

Post-order Algorithm

- Algorithm for post-order traversal

```
POSTORDER(TREE):
        IF TREE == NULL
                RETURN
        ELSE
                POSTORDER(TREE->LEFT)
                POSTORDER(TREE->RIGHT)
                OUTPUT TREE->DATA
```



In-order traversal:
D, B, H, I, G, F, E, C and A

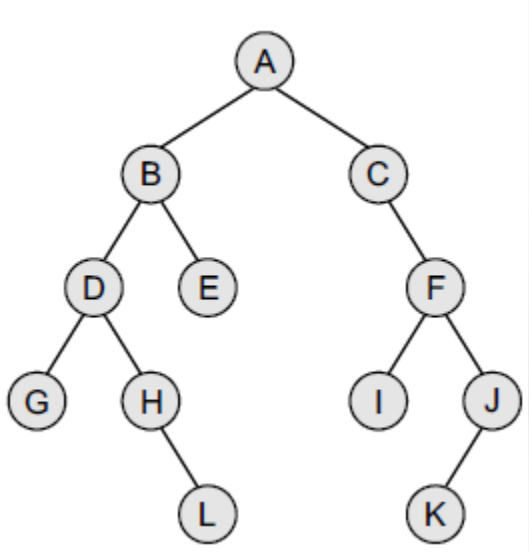
Post-order Algorithm

- The algorithm starts with the root node of the tree and continues by,

✓ Traversing the left subtree post-order

✓ Traversing the right subtree post-order

✓ Visiting the root node



Post-order traversal:
??

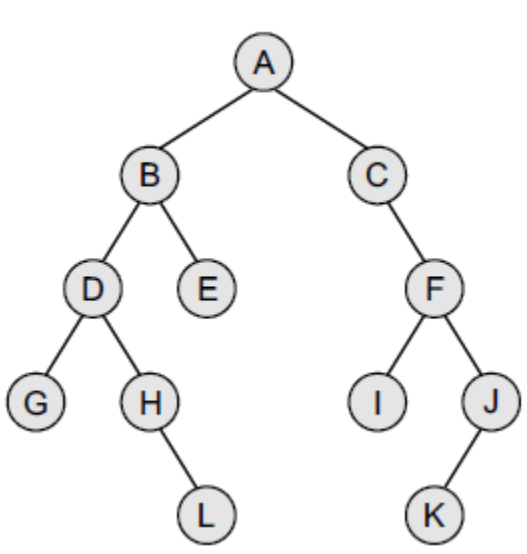
Post-order Algorithm

- The algorithm starts with the root node of the tree and continues by,

✓ Traversing the left subtree post-order

✓ Traversing the right subtree post-order

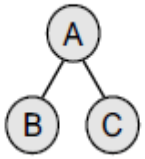
✓ Visiting the root node



Post-order traversal:
G, L, H, D, E, B, I, K, J, F, C, A

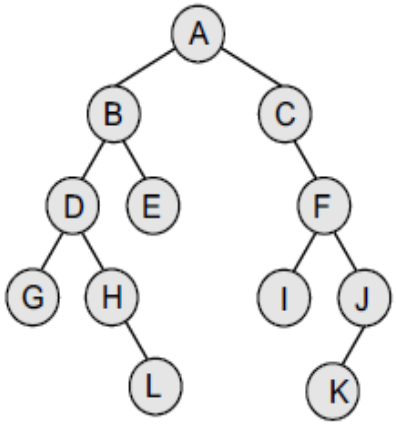
Level-order Algorithm

- In level-order traversal, all the nodes at a level are accessed before going to the next level.
- This algorithm is also called as the ***breadth-first traversal algorithm***.



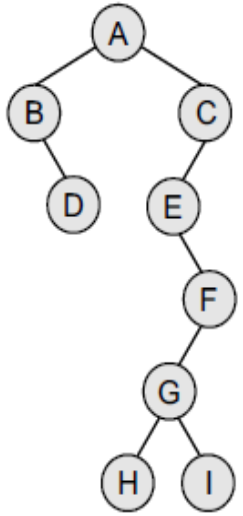
(a)

TRAVERSAL ORDER:
A, B, and C



(b)

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, I, J, L, and K

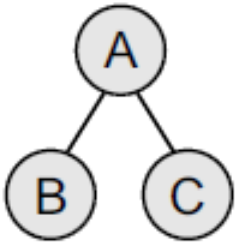


(c)

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, and I

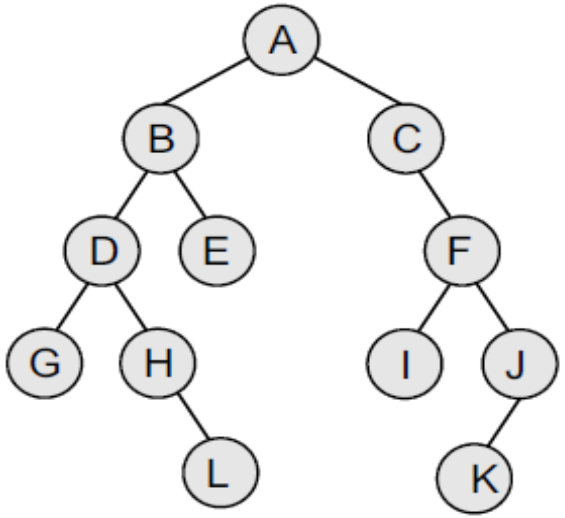
Level-order Traversal

- Also known as **Breadth-first algorithm**
- Visit the all nodes in a level from left-to-right starting from root, before going to next level.



(a)

TRAVERSAL ORDER:
A, B, and C



(b)

TRAVERSAL ORDER:
A, B, C, D, E, F, G, H, I, J, L, and K

Level-order Traversal

- Algorithm(An application of queue data structure):

1) Create an empty queue q

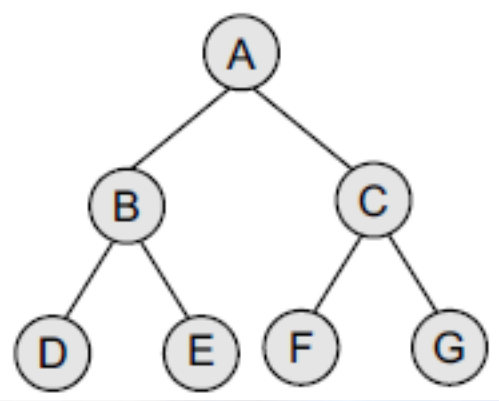
2) temp_node = root

3) Loop while temp_node is not NULL

   a) print temp_node->data.

   b) Enqueue temp_node's children 1st left then right child to q.

   c) Dequeue a node from q and assign it's value to temp_node.



Constructing Binary Tree from Traversal results

- In-order & Pre-order traversal of binary tree is given. Construct the tree and give port-order traversal.

Step 1 Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.

Step 2 Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the root node in the in-order traversal sequence form the right sub-tree of the root node.

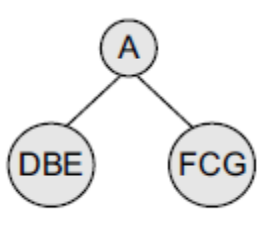
Step 3 Recursively select each element from pre-order traversal sequence and create its left and right sub-trees from the in-order traversal sequence.

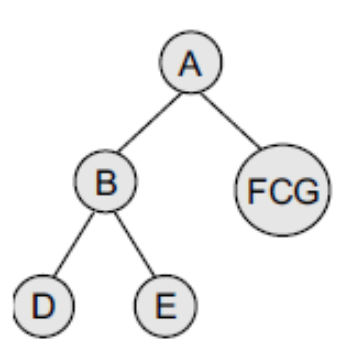
Constructing Binary Tree from Traversal result

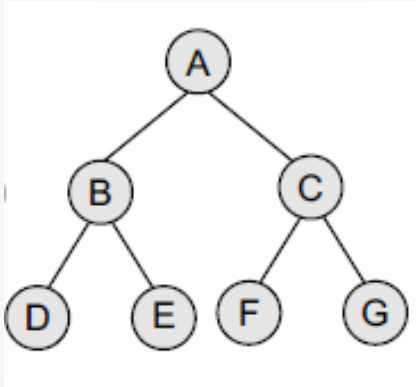
- In-order & Pre-order traversal of binary tree is given. Construct the tree and give post-order traversal.

In-order Traversal: D B E A F C G

Pre-order Traversal: A B D E C F G







Post-order: D, E, B, F, G, C, A

Constructing Binary Tree from Traversal result

- In-order & Pre-order traversal of binary tree is given. Construct the tree and give post-order traversal.

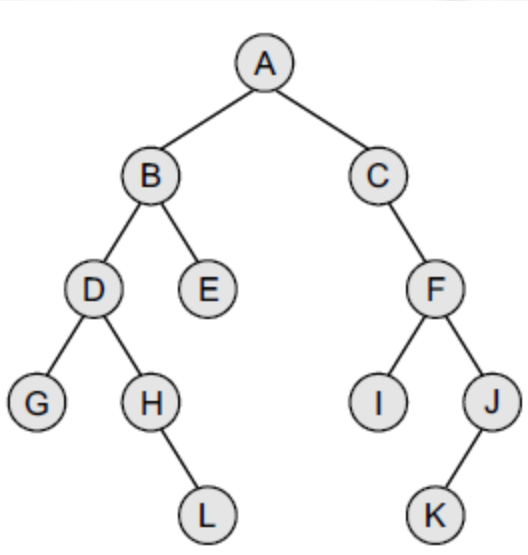
```
In-order:  G D H L B E A C I F K J
Pre-order: A B D G H L E C F I J K
```

Construct the tree???

Constructing Binary Tree from Traversal result

- In-order & Pre-order traversal of binary tree is given. Construct the tree and give post-order traversal.

```
In-order:  G D H L B E A C I F K J
Pre-order: A B D G H L E C F I J K
```



Constructing Binary Tree from Traversal result

- **In-order & Post-order** traversal of binary tree is given. Construct the tree and give Pre-order traversal.

In–order Traversal: D B H E I A F J C G

Post order Traversal: D H I E B J F G C A

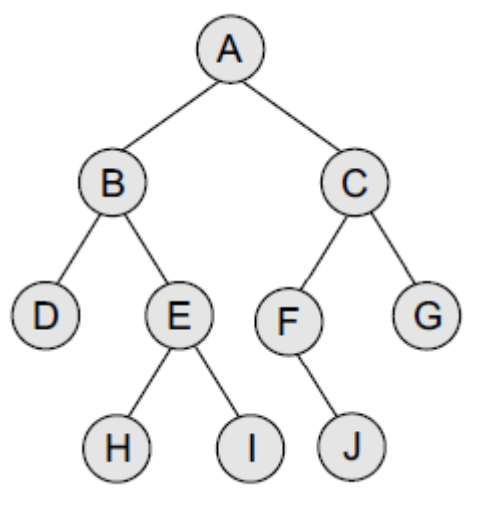
Build the tree ??

Constructing Binary Tree from Traversal result

- In-order & Post-order traversal of binary tree is given. Construct the tree and give Pre-order traversal.

In–order Traversal: D B H E I A F J C G

Post order Traversal: D H I E B J F G C A



Pre order Traversal: A B D E H I C F J G

Applications of Trees

- Trees are used to store simple as well as complex data.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- B-trees are used for indexes in databases.
- Trees are used for compiler construction.
- Trees are also used in database design.
- Trees are used in file system directories.
- Trees are also widely used for information storage and retrieval in symbol tables(within compiler implementations).