

Introduction to Trees

- Definitions of basic terms : Tree, Directed Tree, Root, Leaf, Branch, Level, Node, Forest
- Applications of a tree
- Binary trees : introduction, linear and linked representations
- Traversals of a binary tree: Preorder, Inorder and Postorder
- Types of binary trees : Full Binary Tree, Complete Binary Tree, Binary Search Tree

Tree Data Structure-

Tree data structure may be defined as-

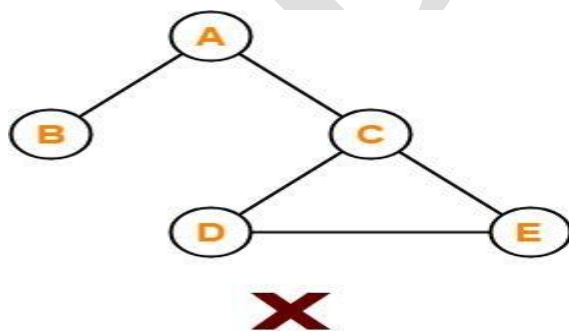
Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive definition.

OR

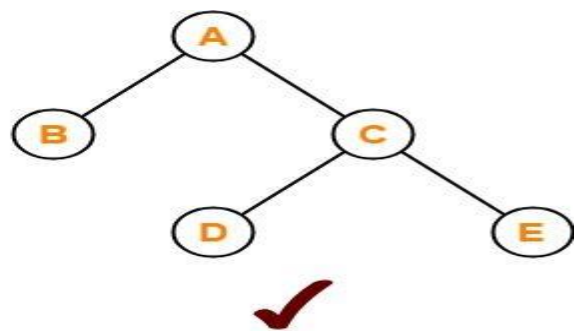
A tree is a connected graph without any circuits.

OR

If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.

Example-

This graph is not a Tree



This graph is a Tree

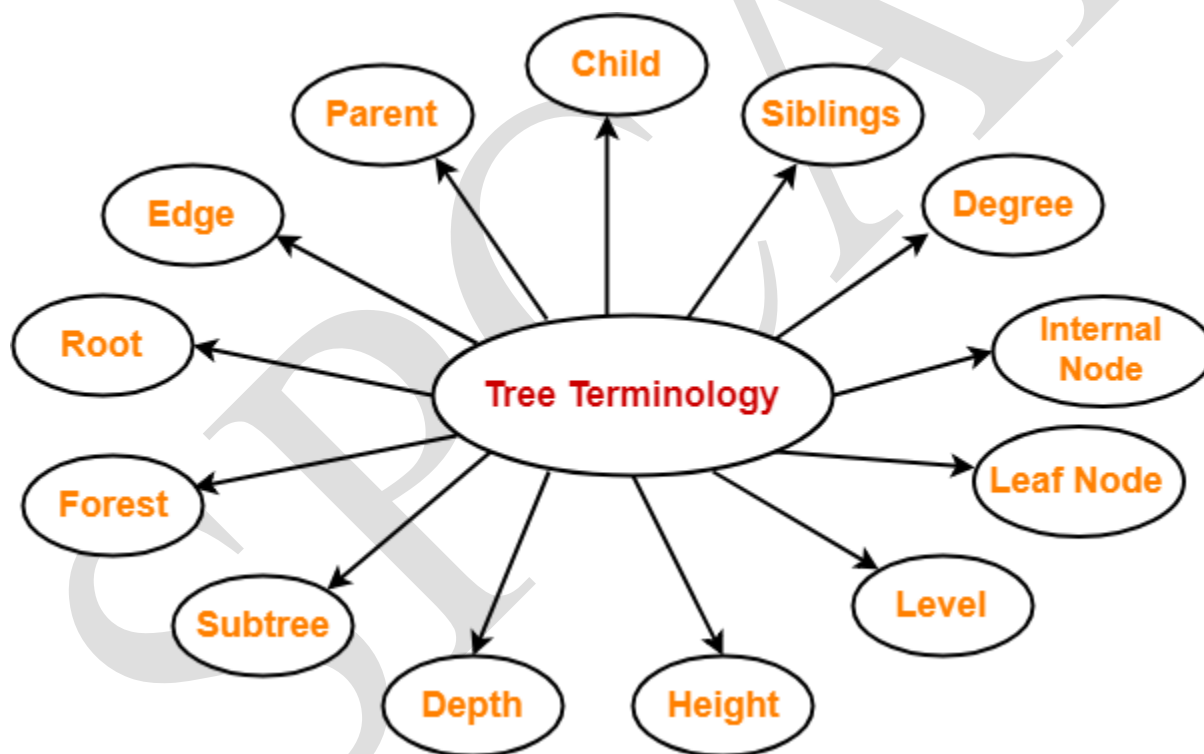
Properties-

The important properties of tree data structure are-

- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly $(n-1)$ edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and $(n-1)$ edges is a tree.

Tree Terminology-

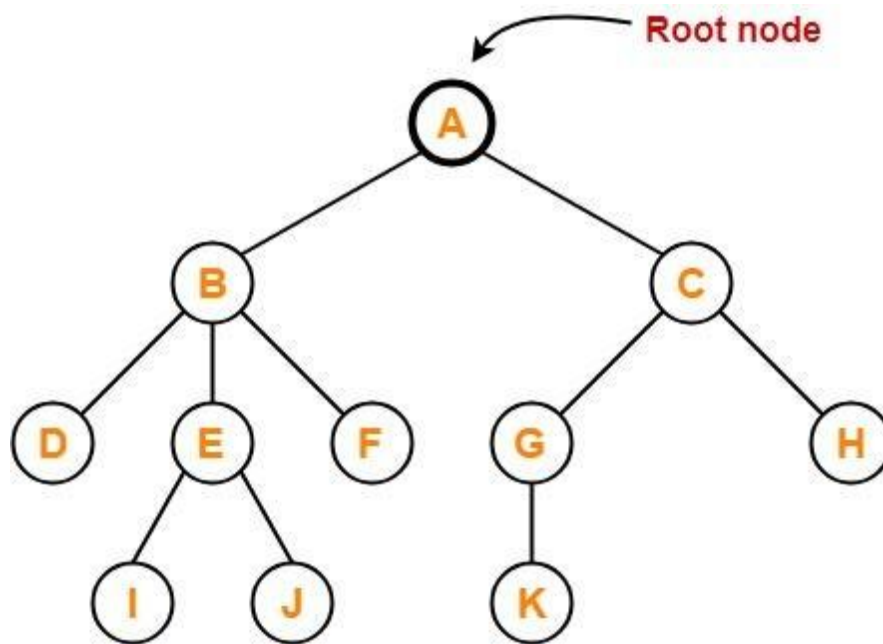
The important terms related to tree data structure are-



1. Root-

- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.

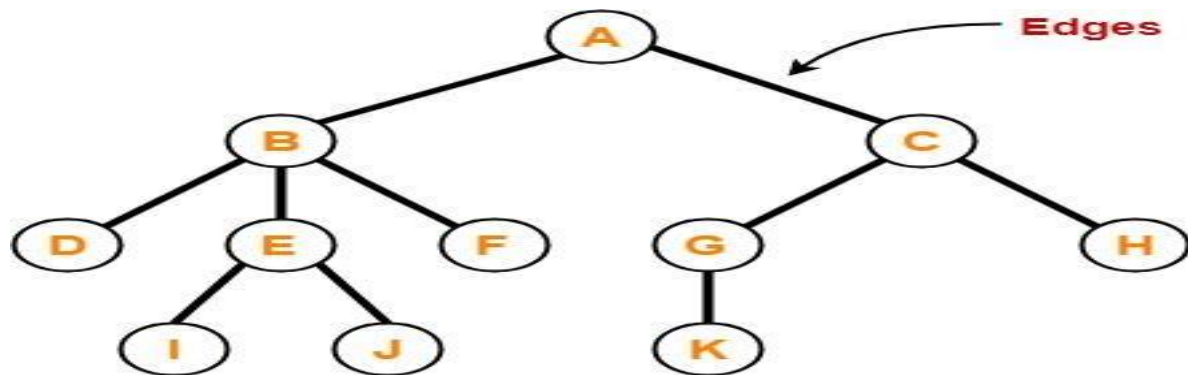
Example-



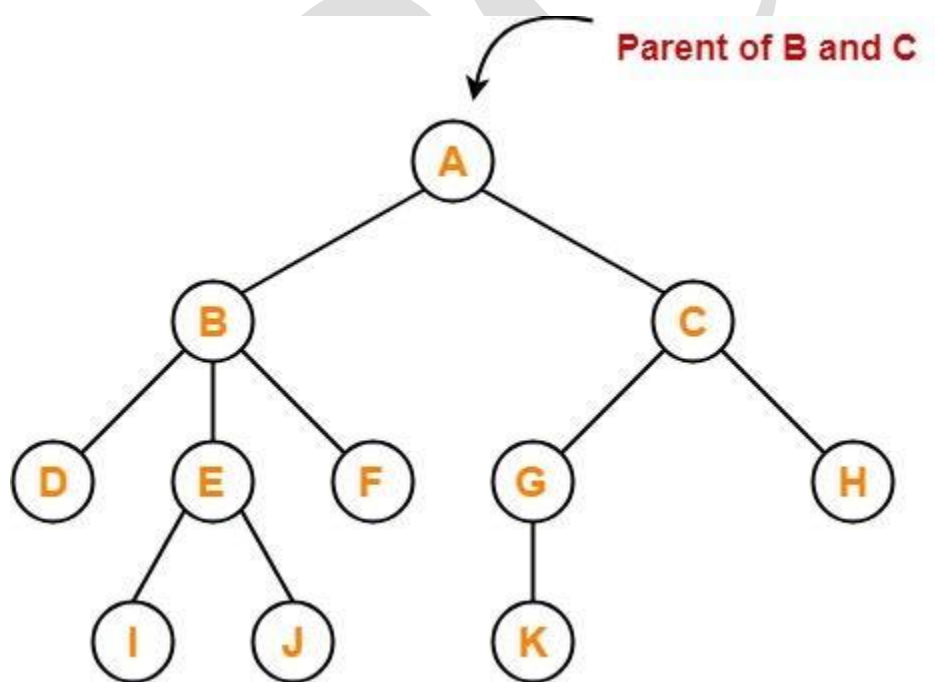
Here, node A is the only root node.

2. Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

Example-**3. Parent-**

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Example-

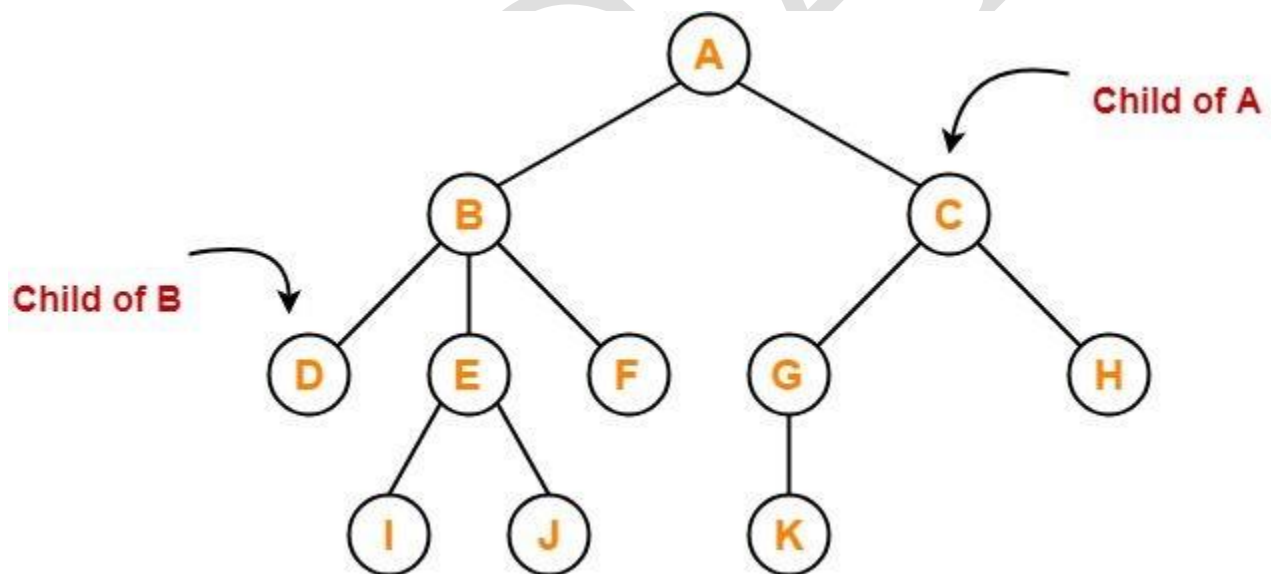
Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

Example-



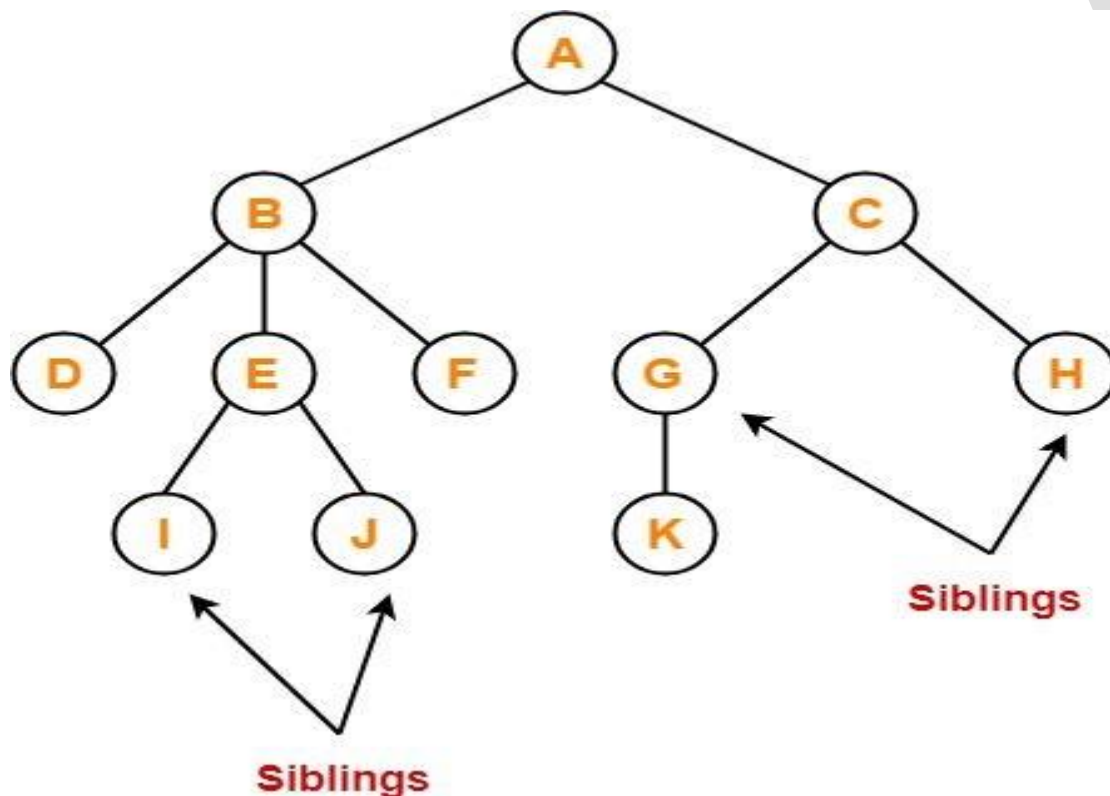
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings-

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

Example-



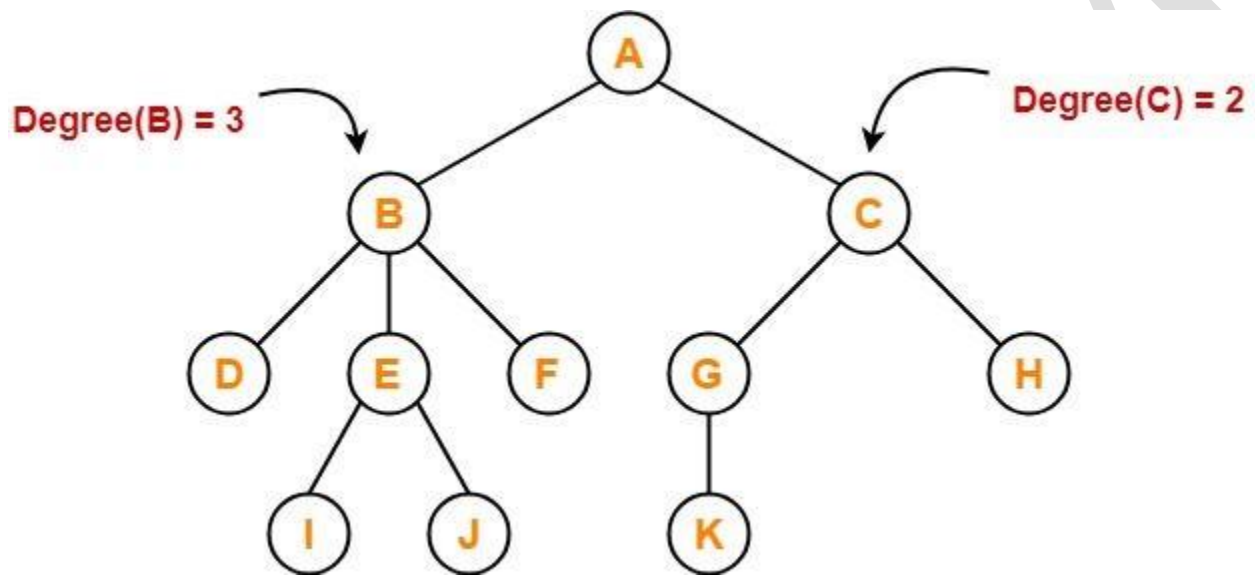
Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. Degree-

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Example-



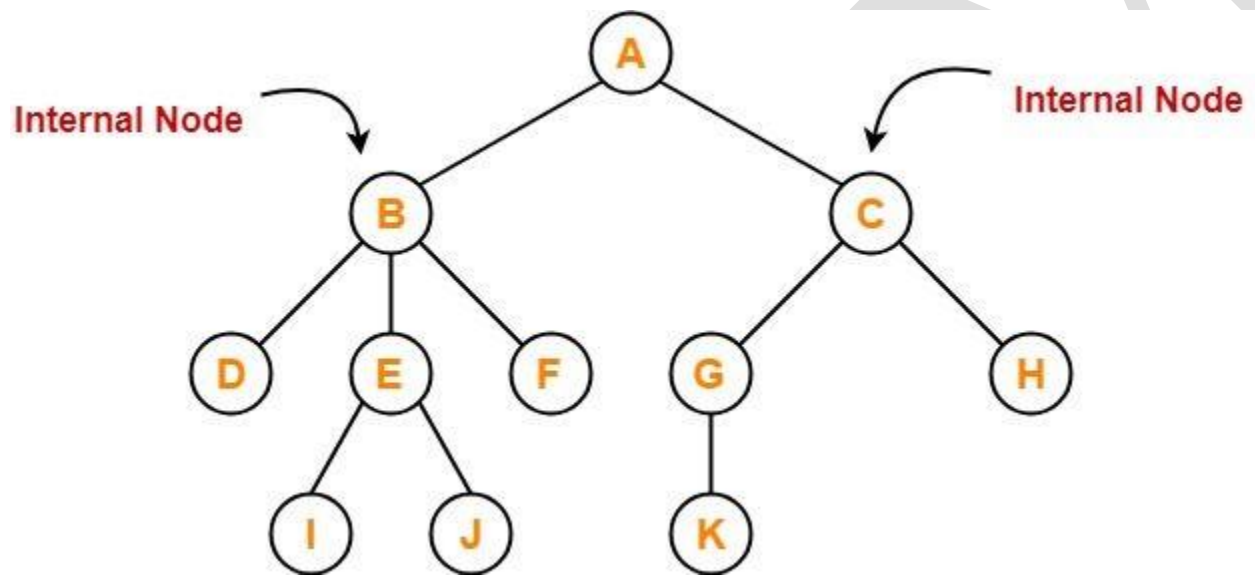
Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

7. Internal Node-

- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.

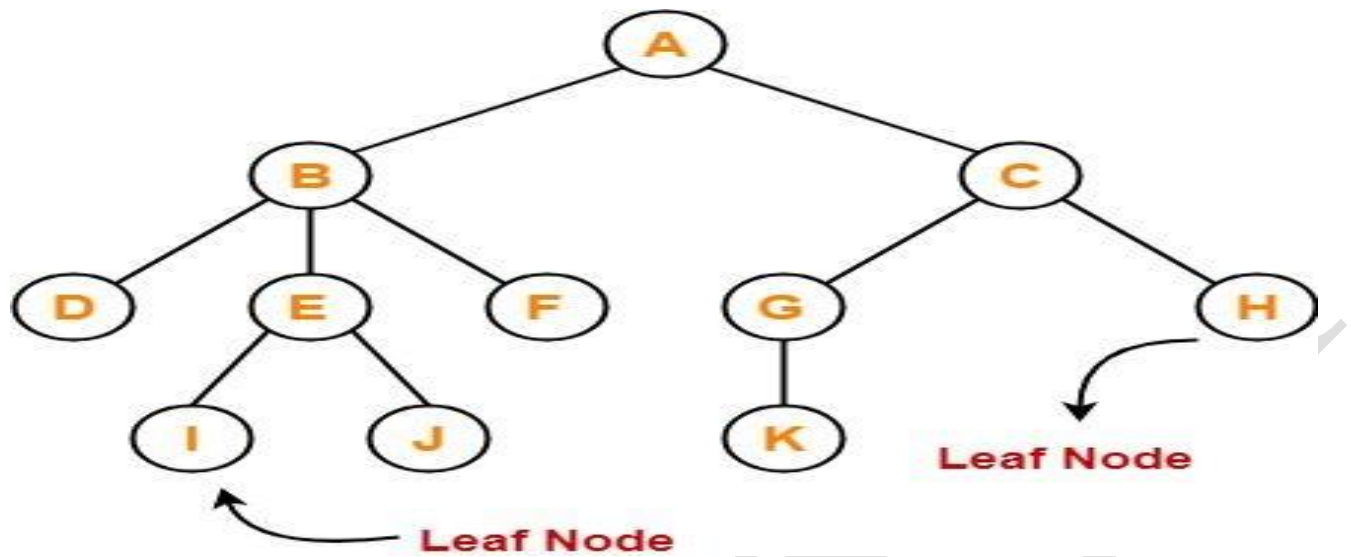
Example-



Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node-

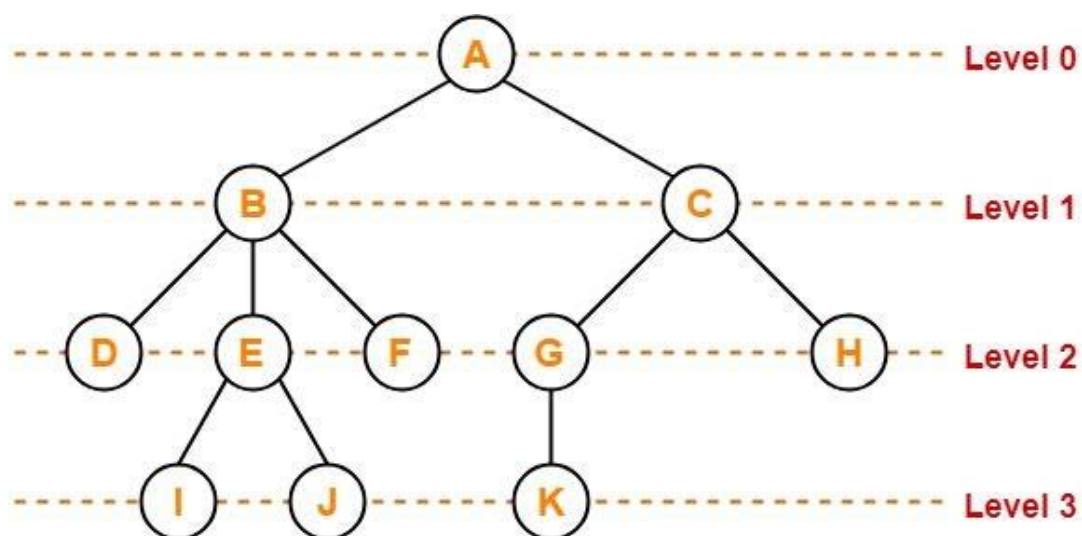
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.



Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level-

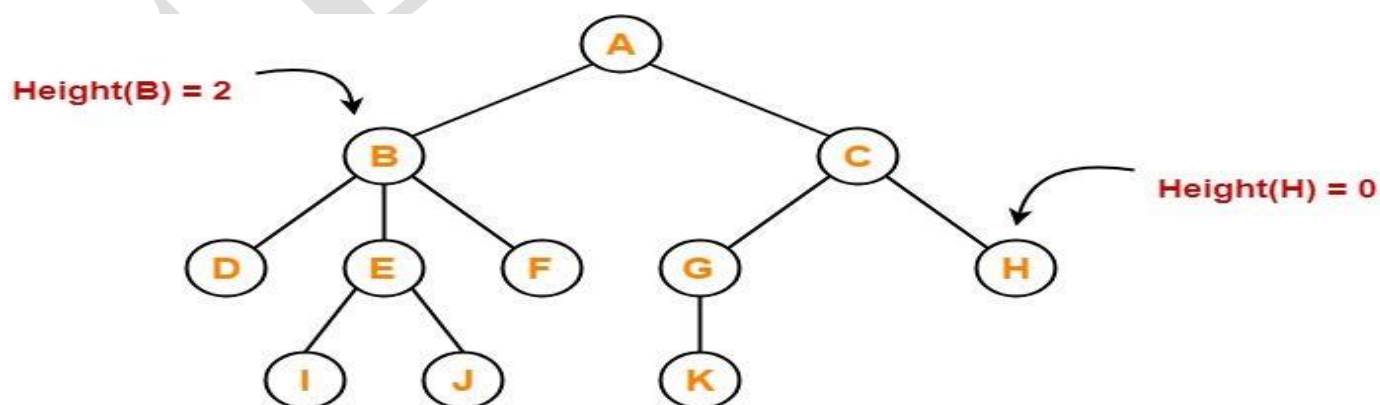
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



10. Height-

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0

Example-



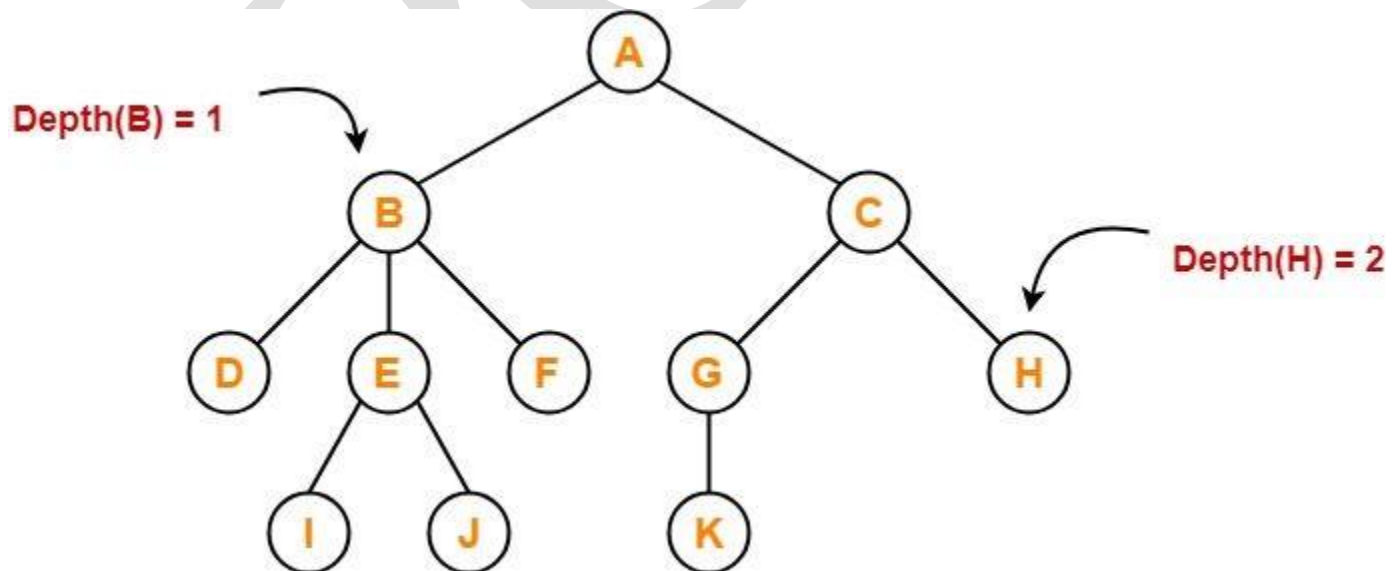
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

11. Depth-

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.

Example-



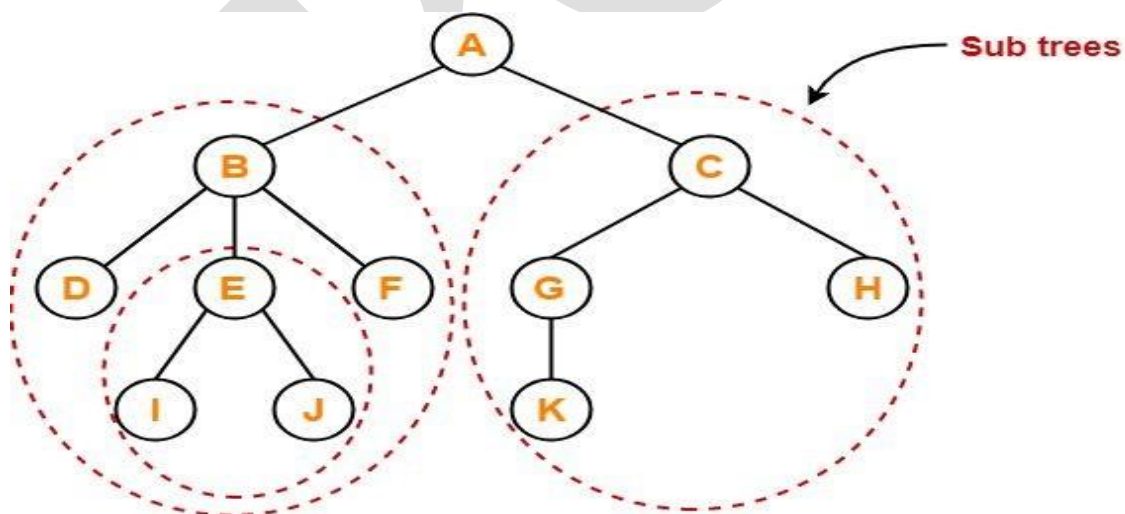
Here,

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

12. Subtree-

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.

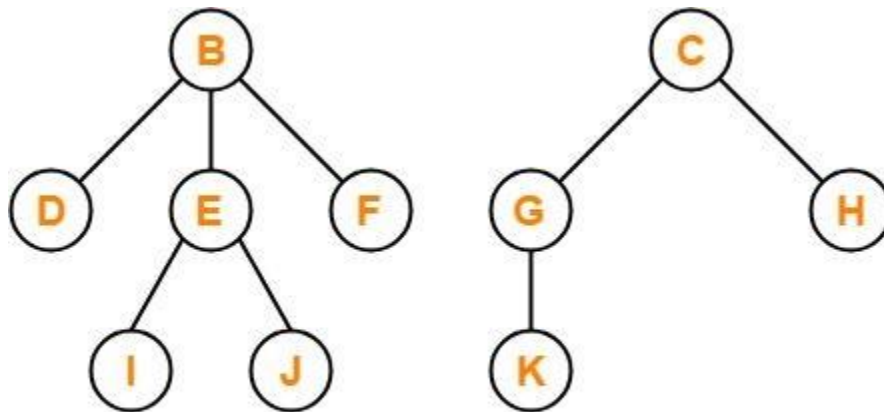
Example-



13. Forest-

A forest is a set of disjoint trees.

Example-



Forest

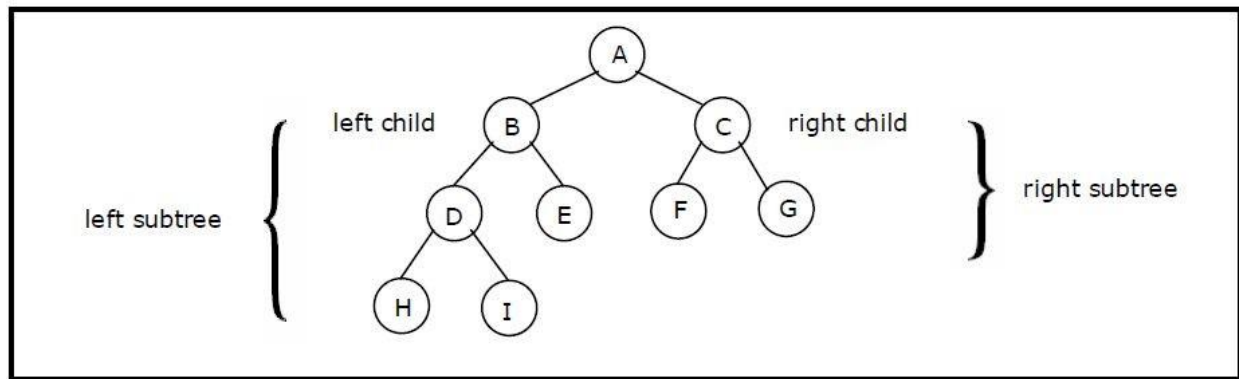
Applications of a tree:

- ❖ Trees and their variants are an extremely useful data structure with lots of practical applications.
- ❖ Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- ❖ Heap is a kind of tree that is used for heap sort.
- ❖ A modified version of tree called Tries is used in modern routers to store routing information.
- ❖ Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- ❖ Compilers use a syntax tree to validate the syntax of every program you write.

Binary Trees:

In general, tree nodes can have any number of children. In a binary tree, each node can have at most two children. A binary tree is either empty or consists of a node called the root together with two binary trees called the left subtree and the right subtree.

A tree with no nodes is called as a null tree. A binary tree is shown in figure 2.1



Binary trees are easy to implement because they have a small, fixed number of child links. Because of this characteristic, binary trees are the most common types of trees and form the basis of many important data structures.

Properties of Binary Trees:

Some of the important properties of a binary tree are as follows:

1. If h = height of a binary tree, then a. Maximum number of leaves = 2^h b. Maximum number of nodes = $2^{h+1} - 1$
2. If a binary tree contains m nodes at level l , it contains at most $2m$ nodes at level $l + 1$.
3. Since a binary tree can contain at most one node at level 0 (the root), it can contain at most 2^l nodes at level l .
4. The total number of edges in a full binary tree with n nodes is $n - 1$.

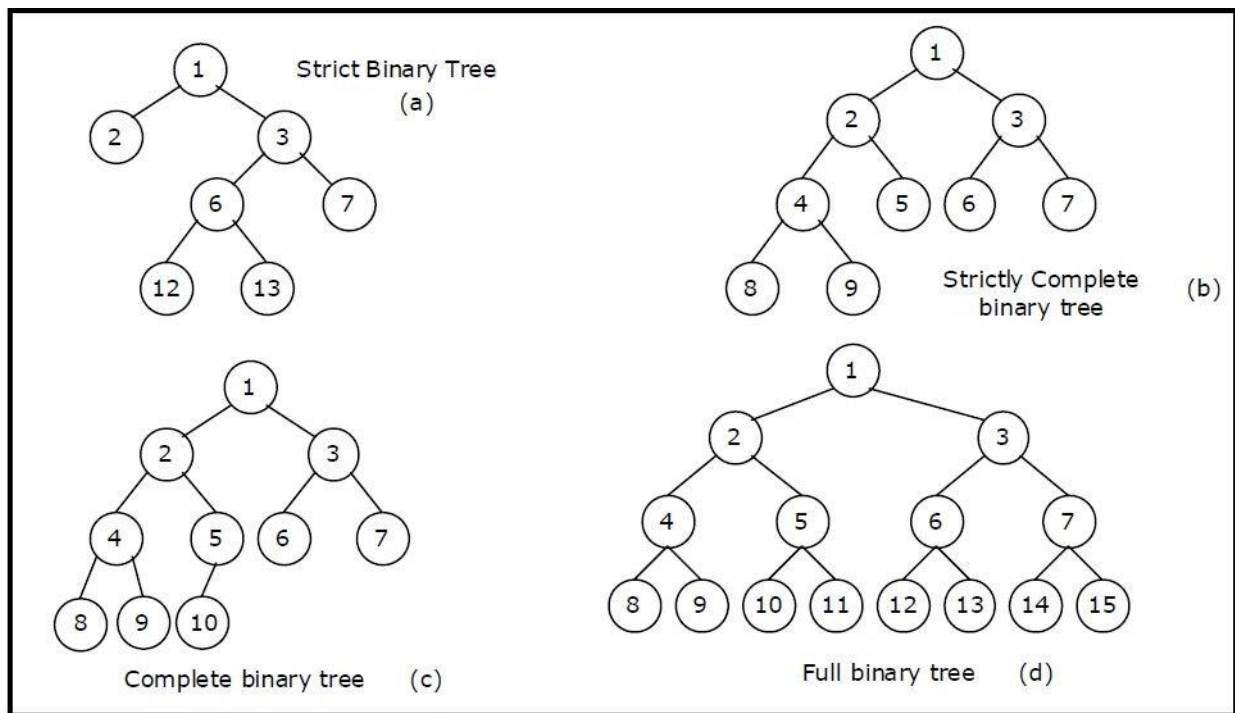
Full Binary tree:

A full binary tree of height h has all its leaves at level h . alternatively; All non leaf nodes of a full binary tree have two children, and the leaf nodes have no children.

Complete Binary tree:

A binary tree with n nodes is said to be complete if it contains all the first n nodes of the above numbering scheme. Below Figure shows examples of complete and incomplete binary trees.

A complete binary tree of height h looks like a full binary tree down to level $h-1$, and the level h is filled from left to right.



Strictly Binary Tree:

If every non-leaf node in a binary tree has nonempty left and right subtrees, the tree is termed as strictly binary tree. Thus the tree of above figure is strictly binary.

A strictly binary tree with n leaves always contains $2n - 1$ nodes.

IMPLEMENTATION OF BINARY TREES:

There are two methods of implementing a binary tree structure.

1. Linear Representations

2. Linked Representations

1. Linear Representations of Binary Tree:

In Linear representation of binary tree, a single array is used to represent a binary tree. For these, nodes are numbered/indexed according to a scheme giving 0 to root. Then all the nodes are numbered from left to right level by level from top to bottom, empty nodes are also numbered. Then each node having an index i is put into the array as its i^{th} element.

In the figure, shown below the nodes of binary tree are numbered according to the given scheme

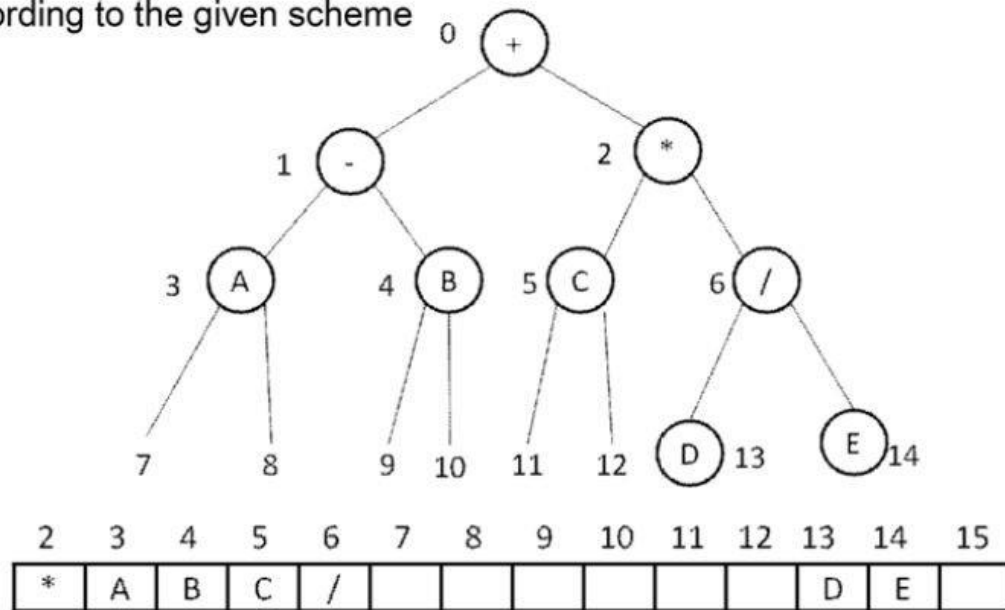


Fig. 8.6 : Array representation of binary tree

Above Figure shows how a binary tree is represented as an array. The root '+' is the 0th element while its left child '-' is the 1st element of the array.

Node 'A' does not have any child so its children that are 7th & 8th element of the array are shown as a Null value.

It is found that if n is the number or index of a node, then its left child occurs at $(2n + 1)^{\text{th}}$ position & right child at $(2n + 2)^{\text{th}}$ position of the array. If any node does not have any of its child, then null value is stored at the corresponding index of the array.

In general, in representation of binary tree: -

array Root is stored at position 0

Left child of the root is at position 1

Right child of the root is at position 2

For a node which array index is N

Left child of the node is at position $2 \times N + 1$

Right child of the node is at position $2 \times N + 2$

Advantages of Linear representation:

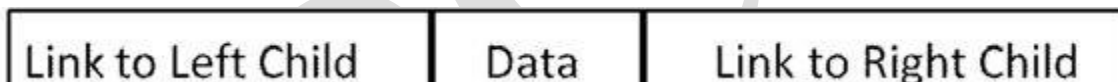
1. This method is very simple.
2. It is very easy to understand.
3. It is very easy to implement.
4. For a given a child node, its parent node can be determined immediately. If a child node is at location n then its parent node is at location $n/2$ (integer division).
5. This method can be implemented in old languages like BASIC and FORTRAN in which only static memory allocation is available.
6. If the tree structure does not change very much due to insertions, deletions etc. then sequential presentations of trees are efficient and convenient.

Disadvantages / Limitations of Linear representation:

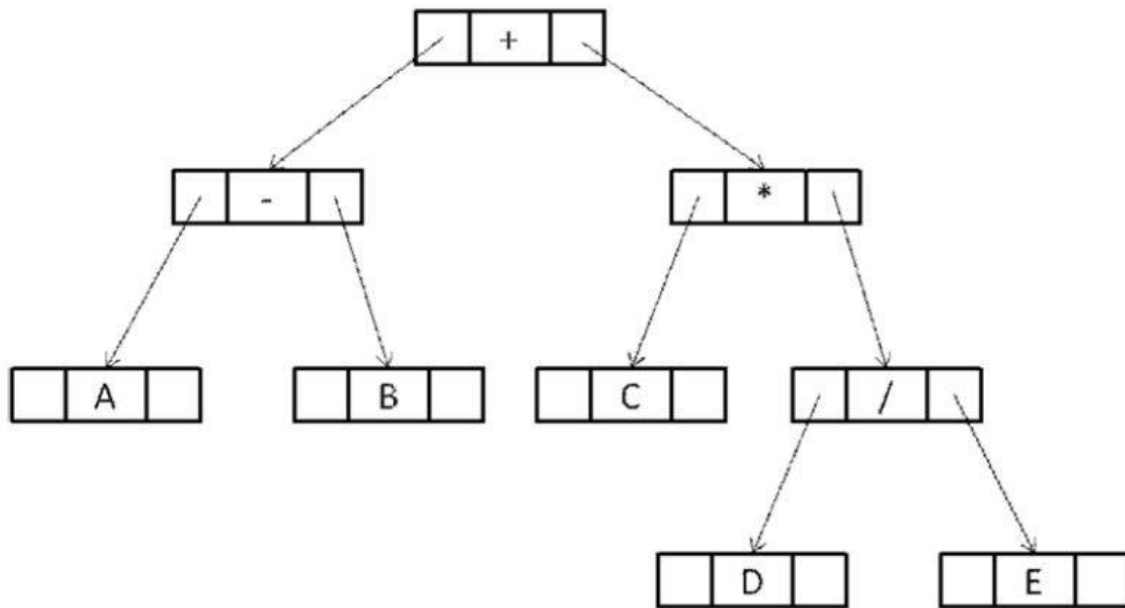
1. Data items are physically adjacent to each other in memory. So insertions or deletions require large data movement in array. This data movement is very time consuming.
2. Usually there is wastage of memory locations due to partially filled trees.
3. It allows only static representation. So it is not possible to enhance(extend) the tree structure if the array size is limited.

2. Linked Representations of Binary Tree:

In link representation, for representing each data one structure is used called “node”, each node contains tree field’s data part and two pointers to contain the address of the left child and the right child. If any node has its left or right child empty then it will have in its respective link field, a null value. A leaf node has null value in both of its links.



The binary tree of Arithmetic expression on below fig can be represented in link representation as follows



Advantages of Linked representation:

1. This method is more efficient in processing where frequent insertions and deletion are required.
2. Insertions / deletions do not require data movement. It needs only rearrangement of few pointers. So it is faster than linear representations.
3. It allows dynamic memory allocation. So the size of the tree can be changed without limitation. Limitation of availability of memory is the only problem.

Disadvantages / Limitations of Linked representation:

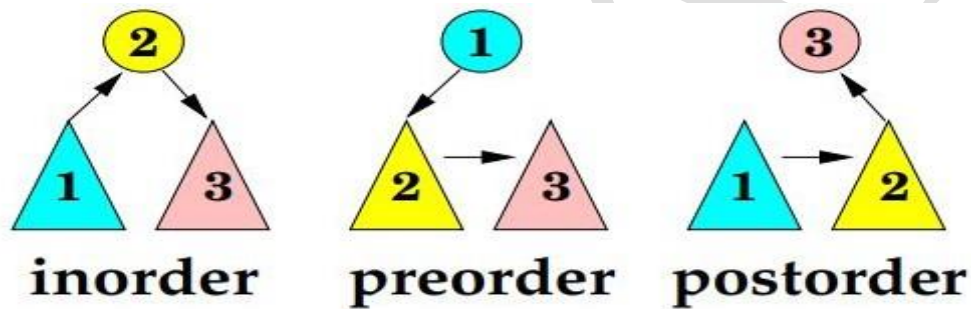
1. This method uses more memory than linear representation. It requires extra memory to maintain pointers.
2. There is wastage of memory in null pointers.
3. It is difficult to determine parent node for a given node.
4. Languages like FORTRAN, BASIC and COBOL does not support Dynamic memory allocation in such languages.

Binary Tree Traversal:

Traversal of a binary tree means to visit each node in the tree exactly once. In a linear list, nodes are visited from first to last, but since trees are nonlinear we need to define rules to visit the tree. There are a number of ways to traverse a tree. All of them differ only in the order in which they visit the nodes.

The three main methods of traversing a tree are:

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal



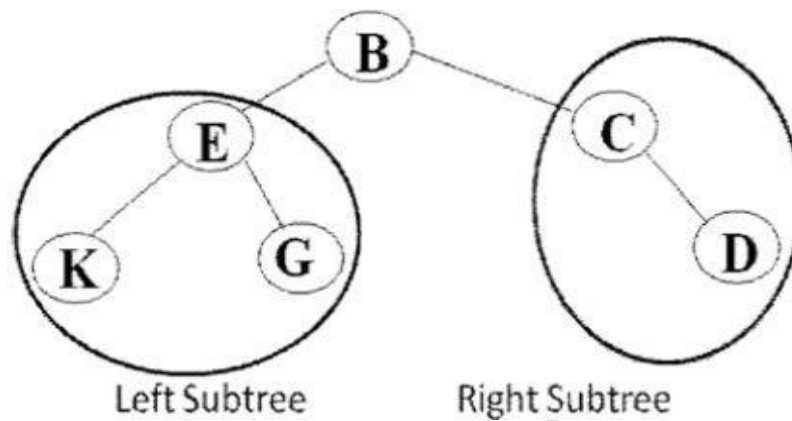
1. Preorder Traversal: In this traversal process, it visit every node as it moves left until it can move no further. Now it turns right to begin again or if there is no node in the right, retracts until it can move right to continue its traversal.

- ❖ Visit the Root
- ❖ Traverse the left subtree
- ❖ Traverse the right subtree

Algorithm preorder(node)

```
if (node==null) then return
Else
    visit(node)
    preorder(node->leftchild)
    preorder(node->rightchild)
```

The preorder traversal of the tree shown below is as follows.



In preorder traversal of the tree process B (root of the tree), traverse the left subtree and traverse the right subtree. However the preorder traversal of left subtree process the root E and then K and G. In the traversal of right subtree process the root C and then D. Hence B E K G C D is the preorder traversal of the tree.

2. Inorder Traversal : The traversal keeps moving left in the binary tree until one can move no further, process node and moves to the right to continue its traversal again. In the absence of any node to the right, it retracts backward by a node and continues the traversal.

- ❖ Traverse the left subtree
- ❖ Visit the Root
- ❖ Traverse the right subtree

Algorithm inorder(node)

```
if (node==null) then return  
Else  
    inorder(node->leftchild)  
    visit(node)  
    inorder(node->rightchild)
```

The inorder traversal of the tree (above fig) traverses the left subtree, process B(root of the tree) and traverse right subtree. However, the inorder traversal of left subtree processes K, E and then G and the inorder traversal of right subtree processes C and then D. Hence K E G B C D is the inorder traversal of the tree.

3.Postorder Traversal: The traversal proceeds by keeping to the left until it is no further possible, turns right to begin again or if there is no node to the right, processes the node and retraces its direction by one node to continue its traversal

- ❖ Traverse the left subtree
- ❖ Traverse the right subtree
- ❖ Visit the Root

Algorithm postorder(node)

```
if (node==null) then return
Else
    inorder(node->leftchild)
    inorder(node->rightchild)
    visit(node)
```

The postorder traversal of the tree (above fig) traverse the left subtree, traverse right subtree and process B(root of the tree). However, the postorder traversal of left subtree processes K, G and then E and the postorder traversal of right subtree processes D and then C. Hence K G E D C B is the postorder traversal of the tree.

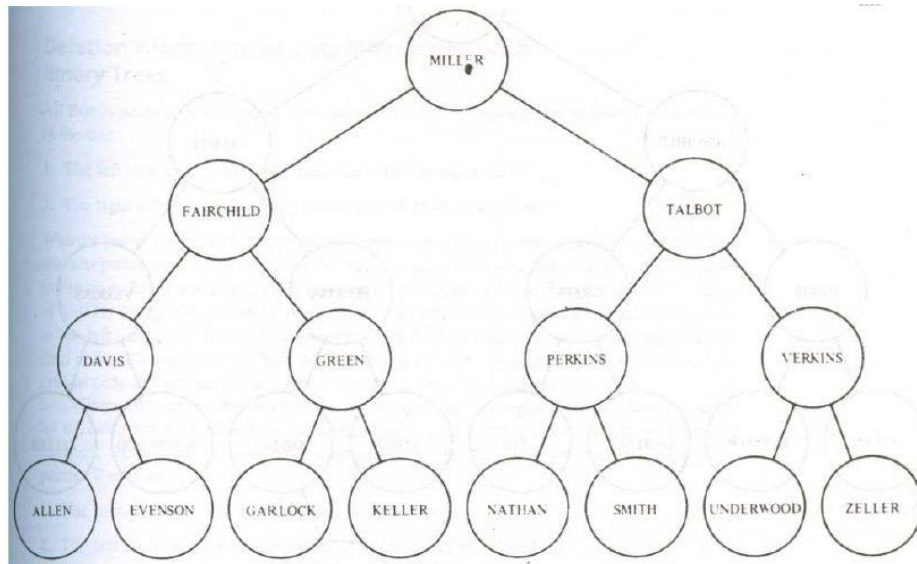
LEXICALLY ORDERED BINARY TREE (BINARY SEARCH TREE)

Definition:

A lexically ordered binary tree (binary search tree) is a binary tree which is either empty or satisfies the following rules:

1. The value of the key in the left child or left subtree is less than the value of the root.
2. The value of the key in the right child or right subtree is more than or equal to the value of the root.
3. All the subtrees of the left and right children observe the two rules.

This property is known as ordering property for binary trees. For example, tree in Figure explains this property for alphabetical ordering.



We can verify that an inorder traversal of this tree shows alphabetical list. ALLEN, DAVIS, EVENSON, FAIRCHILD, GARLOCK, GREEN, KELLER, MILLER, NATHAN, PERKINS, SMITH, TALBOT, UNDERWOOD, VERKINS, ZELLER

INSERTIONS IN A LEXICALLY ORDERED BINARY TREE

Insertion of a node into lexically ordered binary tree must maintain the ordering. Insertion operation is very simple. It requires fewer (less) comparisons than inserting a node into a linked list.

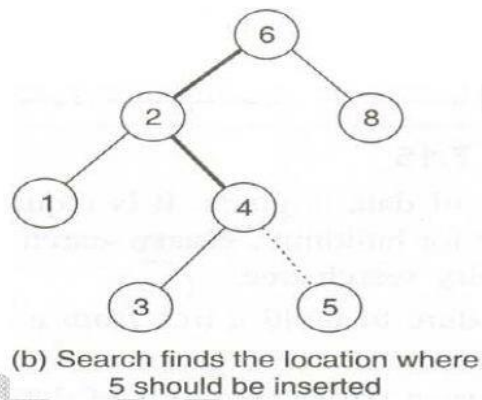
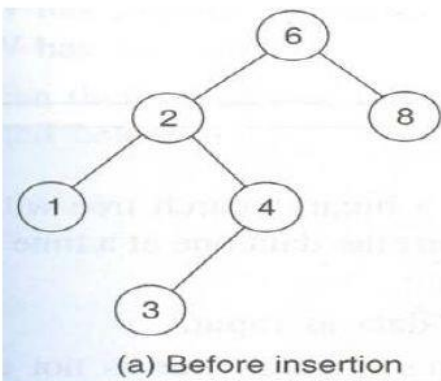
For example, number of nodes N is 1024. Insertion into linked list may require 1024 comparison in worst case. Full binary tree require at most 11 comparisons.

To insert a node with data say ITEM into a tree, the tree is required to be searched starting from the root node. If ITEM is found, do nothing. Otherwise ITEM is to be inserted at the dead end where the search halts.

Insertion rule:

1. If less, go left.
2. Otherwise, go right.

Figure shows the insertion of 5 into a binary tree. Here, search proceeds starting from root node as 6-2-4 then halts when it finds that right child is null (dead end). This means 5 will be inserted at right side of 4.

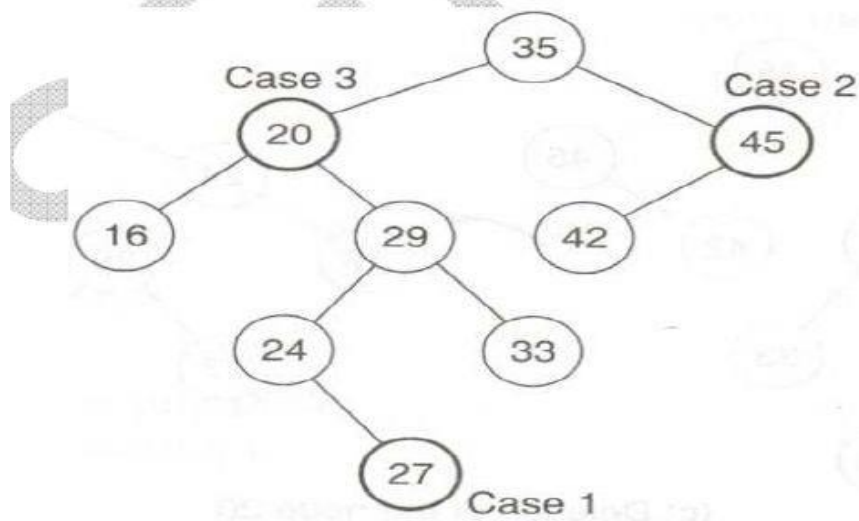


DELETIONS IN A LEXICALLY ORDERED BINARY TREE

Deletion of a node into lexically ordered binary tree must maintain the ordering. Deletion operation is slightly more complicated than insertion. Suppose T is a lexically ordered tree and N is the node which has to be deleted from T.

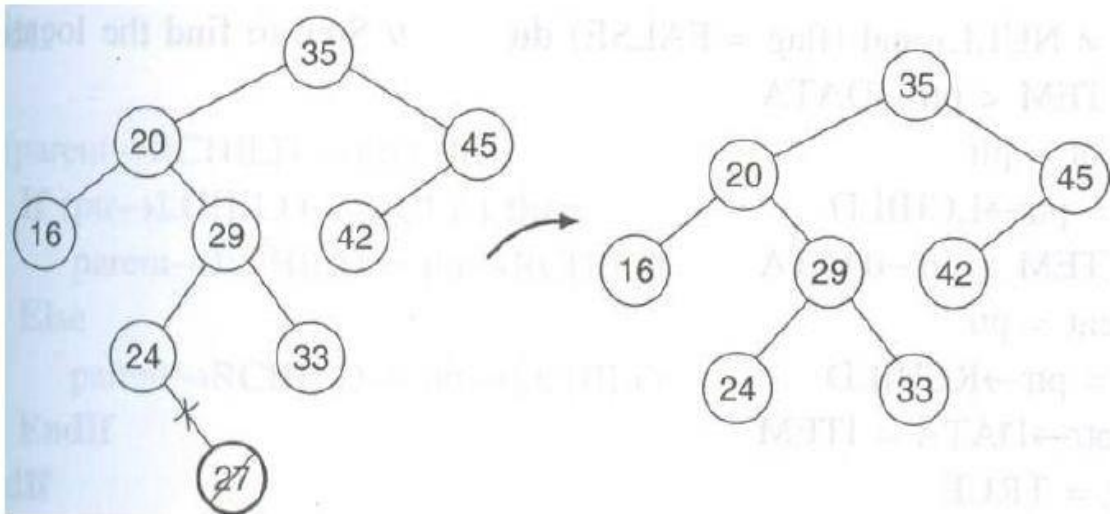
3 cases may arise for deletion:

1. N is the leaf node.
2. N has exactly one child (one subtree)
3. N has two child (both subtree)



Case 1: N is the leaf node.

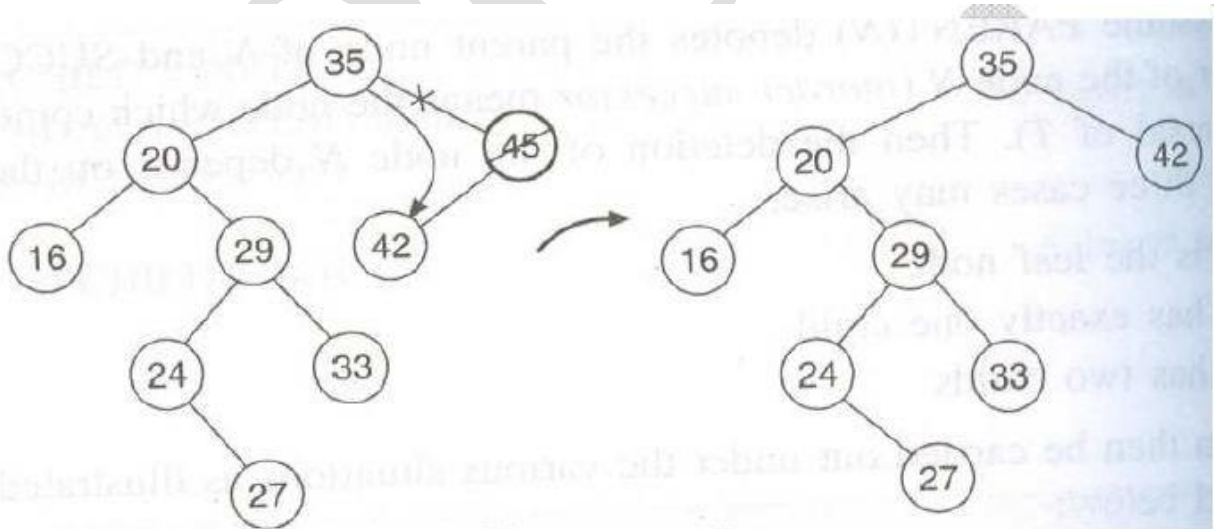
N is deleted from T and parent node pointer will be set to NULL.



Case 2: N has exactly one child (one subtree)

If the node N either a left or right subtree, the nonempty subtree can be appended to its grandparent node.

So, N is deleted. And parent of N now points to non empty subtree of N.

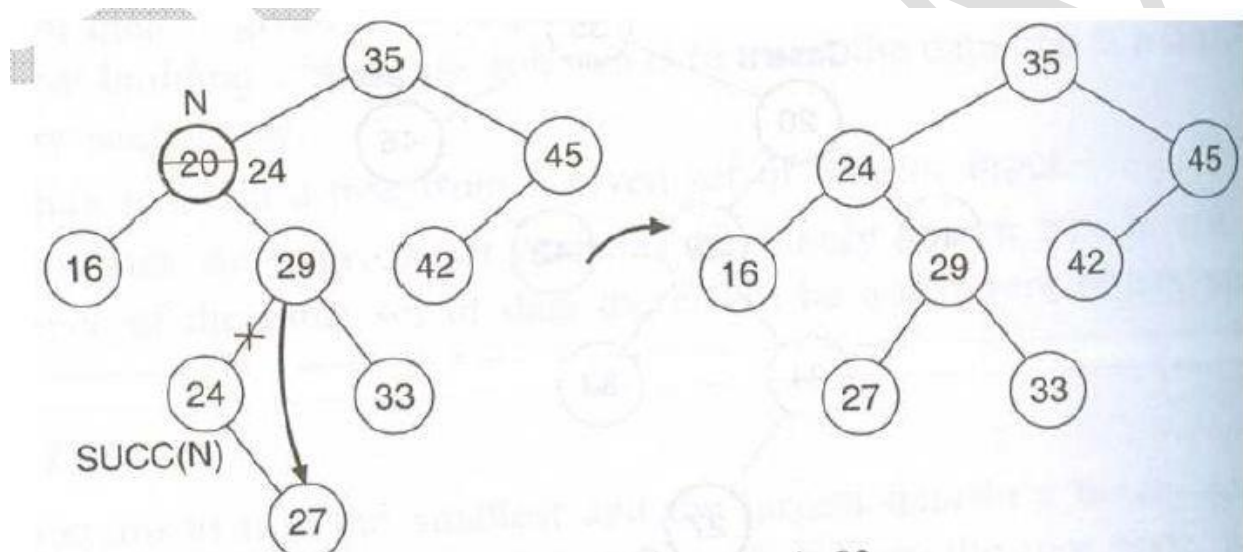


Case 3: N has two child (both subtree)

This situation is difficult when a node which is to be deleted has both the subtrees.

Follow below given steps:

1. First we can obtain inorder successor of the node to be deleted.
2. Then right subtree of this successor node is appended to its grandparent node. And the node to be deleted is replaced by its inorder Successor.



EXPRESSION TREE

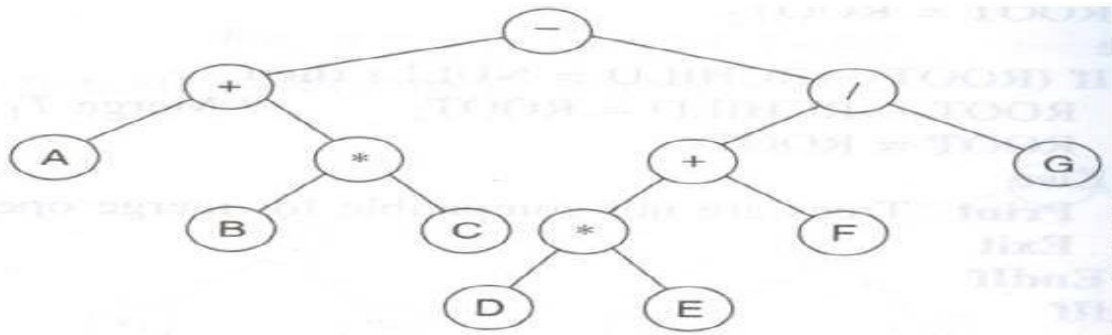
An expression tree is a binary tree which stores an arithmetic expression.

The leaves of expression tree are operands such as constants or variable names. All internal nodes are operators.

An expression tree is always a binary tree because an arithmetic expression contains binary operators (+, -, / * etc which needs two operands) or unary operators (e.g. “-“ which is used for negative value).

Figure shows an expression tree for arithmetic expression

$$(A + B * C) - ((D * E + F) / G)$$



Note: An expression tree does not contain parentheses because evaluation of such tree itself decides order of operations.

When we do Preorder traversal on Binary Expression Tree, we get prefix notation of the expression.

When we do Postorder traversal on Binary Expression Tree, we get postfix notation of the expression.

If the expression tree is constructed for expressions which do not contain parenthesis then Inorder traversal on such Binary Expression Tree, will give infix notation of the expression.

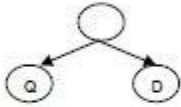
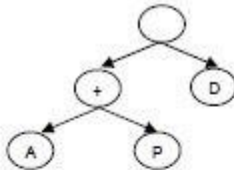
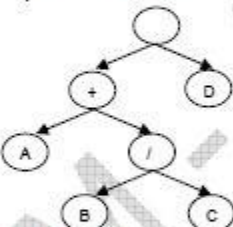
But if the expression tree is constructed for expressions which contain parenthesis then Inorder traversal on such Binary Expression Tree, will not give infix notation of the expression.

Construction of an expression tree

Interpret given expression as per the precedence of operators. Consider following rules for interpretation.

1. All expressions in parentheses are evaluated first.
2. Exponential operator (^) will come next.
3. Division and multiplication will be next in order of precedence.
4. Subtraction and addition will be processed last.

E.g. $A + B * C$ is interpreted as $A + (B * C)$

<p>Expression: $A + B / C - D$ $A + \underline{B / C} - D$ $\underline{A + P} - D \quad P = B / C$ $Q - D \quad Q = A + P$</p> <p>Now "-" is root Q and D are left and right child respectively.</p> 	<p>Replace Q with $A + P$</p>  <p>Replace P with B / C</p> 
<p>Expression: $(A + B) * C / D$ $\underline{(A + B)} * C / D$ $\underline{P * C} / D \quad P = (A + B)$ $Q / D \quad Q = P * C$</p>	<p>Expression: $A + (B * C - (D / E \wedge F) * G) * H$ $A + (B * C - (D / \underline{E \wedge F}) * G) * H \quad P = E \wedge F$ $A + (\underline{B * C - Q * G}) * H \quad Q = (D / P)$ $A + (R - \underline{Q * G}) * H \quad R = B * C$ $A + (\underline{R - X}) * H \quad X = Q * G$ $A + \underline{Y * H} \quad Y = R - X$ $A + Z \quad Z = Y * H$</p>