Trees

sushmakadge@somaiya.edu

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Tree – concept
- General tree
- Types of trees
- Binary tree: representation, operation
- Binary tree traversal
- Binary search tree
- BST- The data structure and implementation
- Threaded binary trees.
- Search Trees –
  - AVL tree, Multiway Search Tree, B Tree, B+ Tree, and Trie,
- Applications/Case study of trees.
- Summary
- Queries?

Somaiya TRUST

Trees

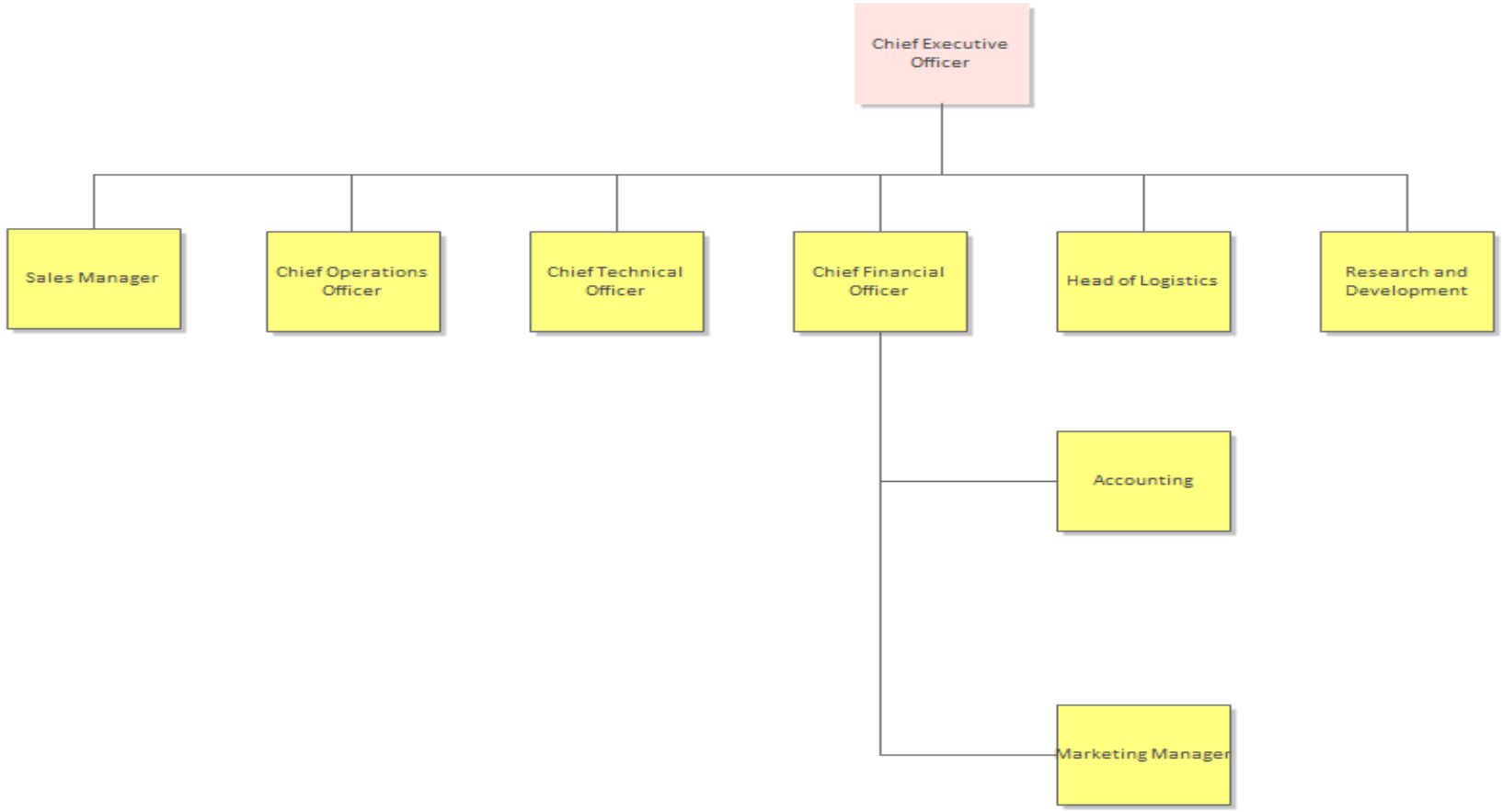
- **Learning Objective**
- Discussed linear data structures such as arrays, strings, stacks, and queues.
- Learn about a non-linear data structure called tree.
- A tree is a structure which is mainly used to store data.
- It has no loop and no circuit. It has no self-loop.
- Its hierarchical model.
- First discuss general trees and then binary trees. These binary trees are used to form binary search trees and heaps.
- They are widely used to manipulate arithmetic expressions, construct symbol tables, and for syntax analysis.

Trees

- A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.
- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the “children”).
- This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.

Trees

- Represents Hierarchy
- For eg- The organization structure of an Corporation



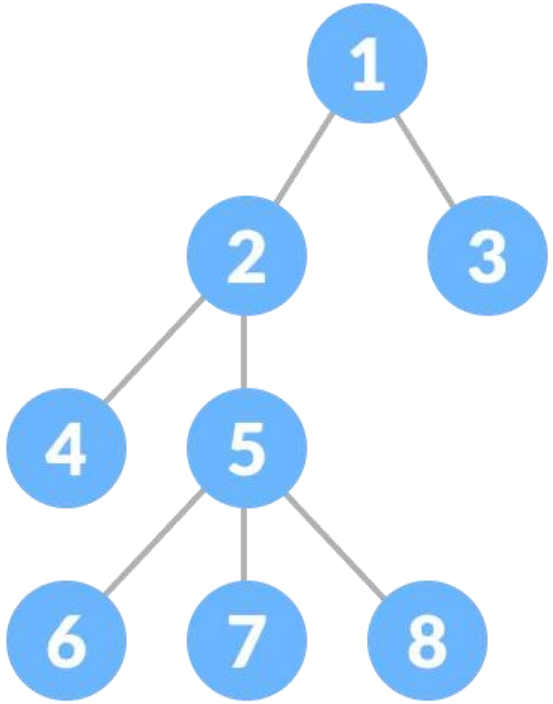
Why Tree is considered a non-linear data structure?

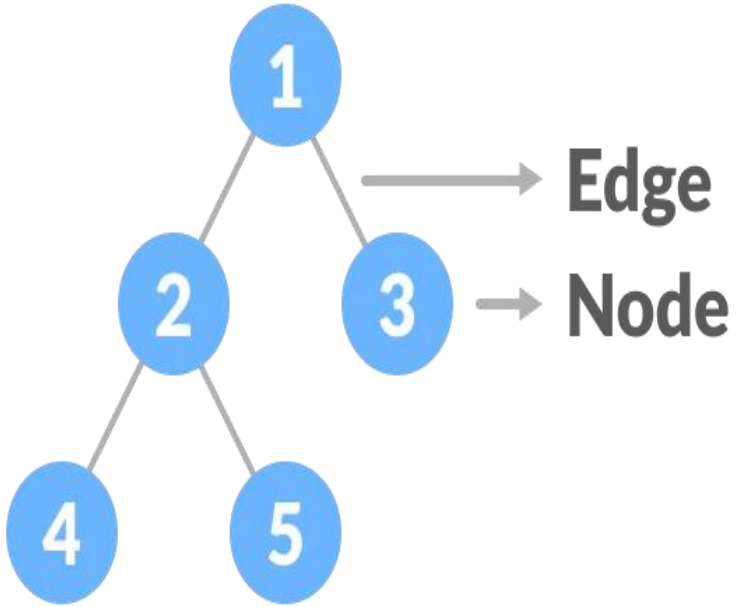
- The data in a tree are not stored in a sequential manner i.e, they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a non-linear data structure.

Why Tree Data Structure?

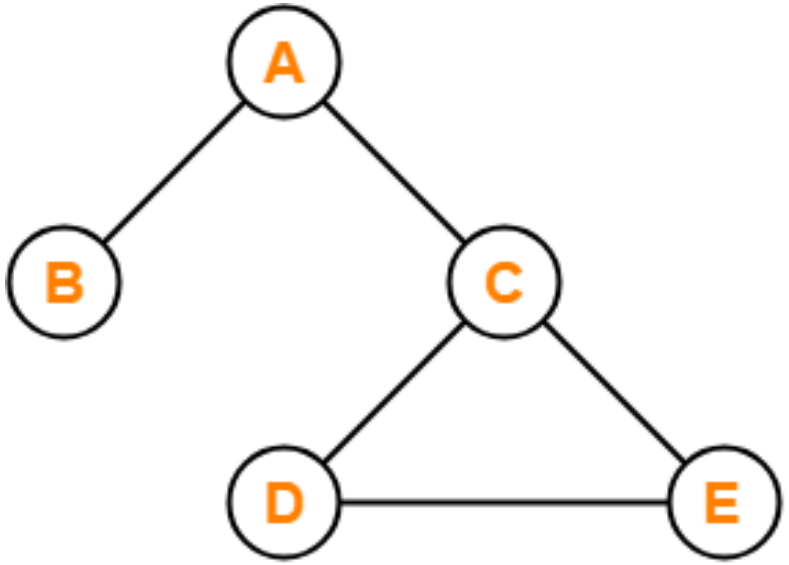
- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree



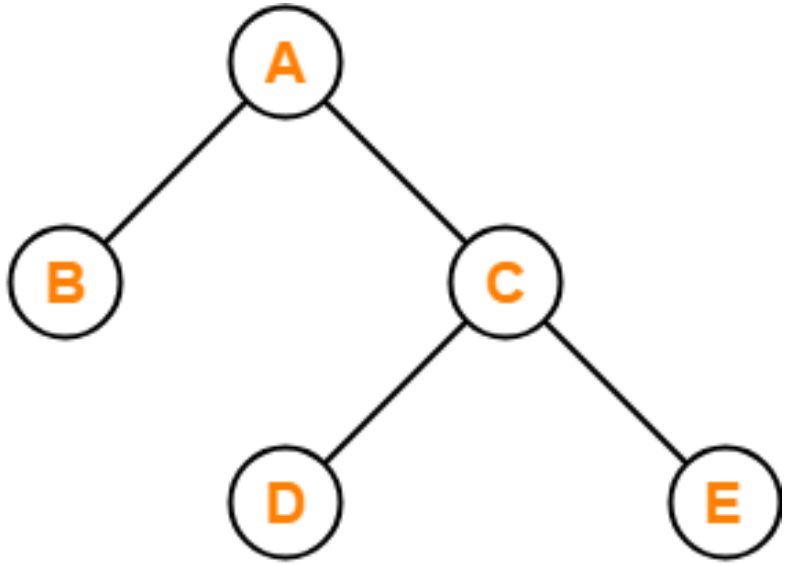


A tree is a connected graph without any circuits.



✗

This graph is not a Tree



✓

This graph is a Tree

Types of Tree

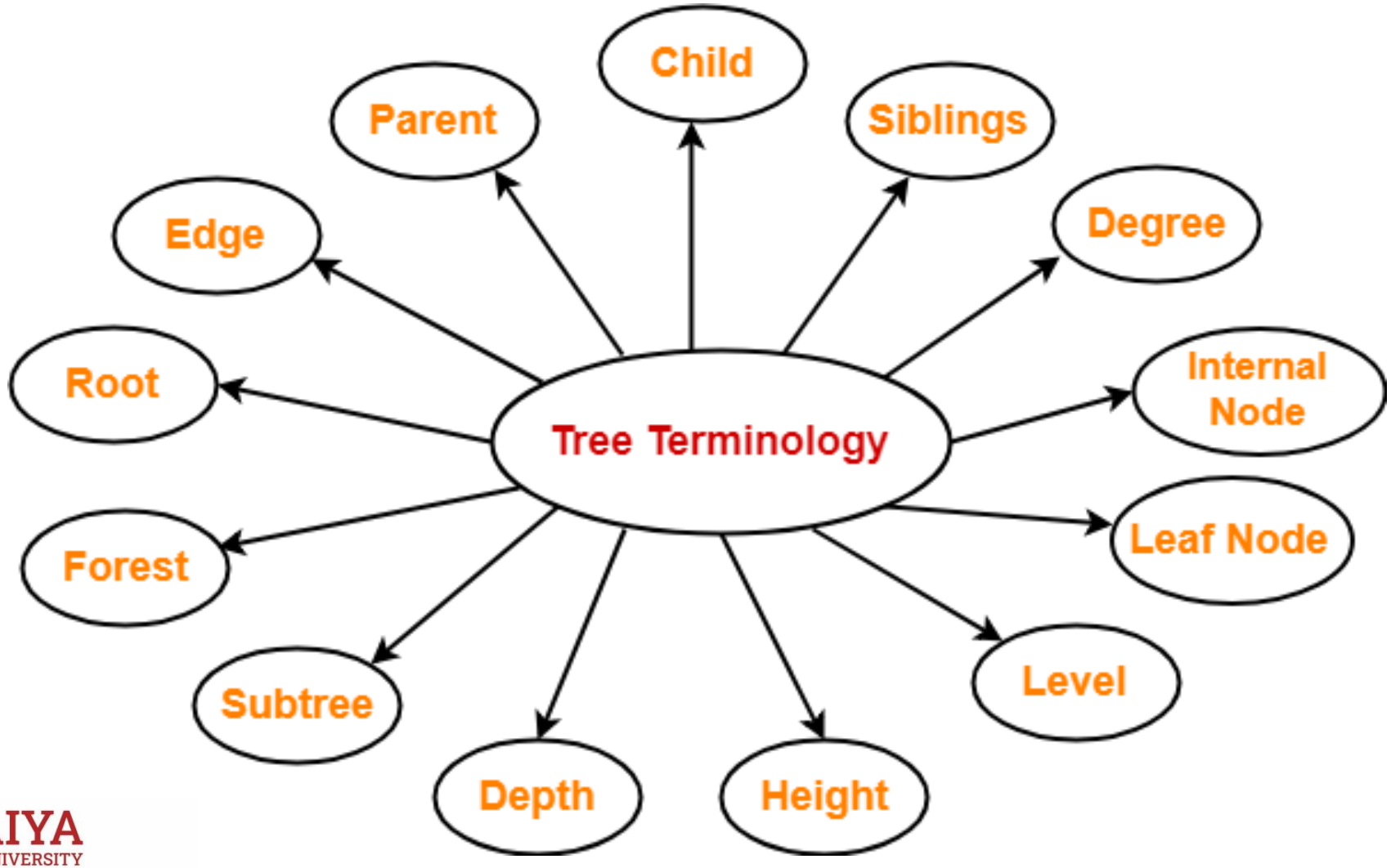
- General tree
- Binary tree
- Binary search tree
- Threaded binary tree
- AVL Tree
- B tree
- B+ Tree
- Trie
- Heap
- Red black tree
- Splay tree

Properties

The important properties of tree data structure are-

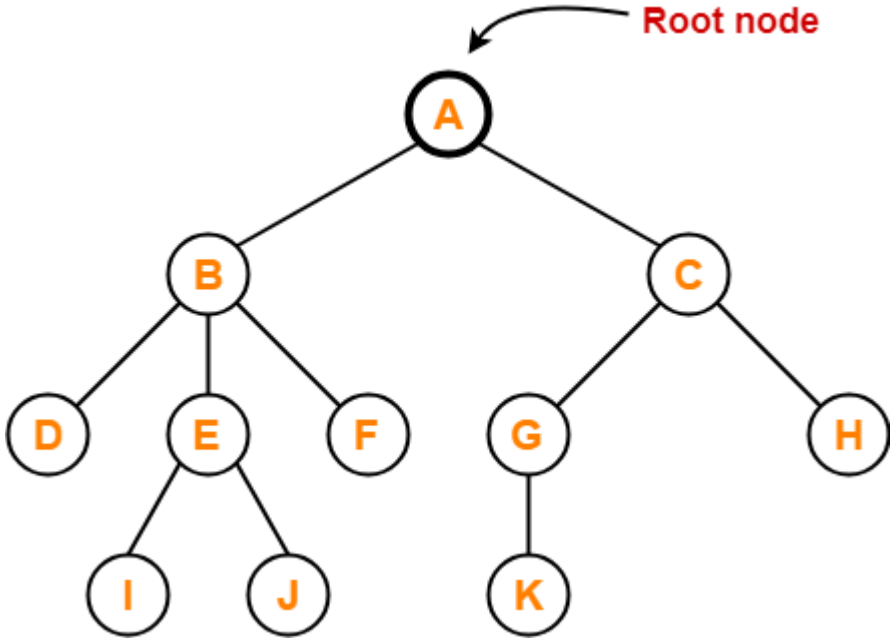
- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

Tree terminology



Root

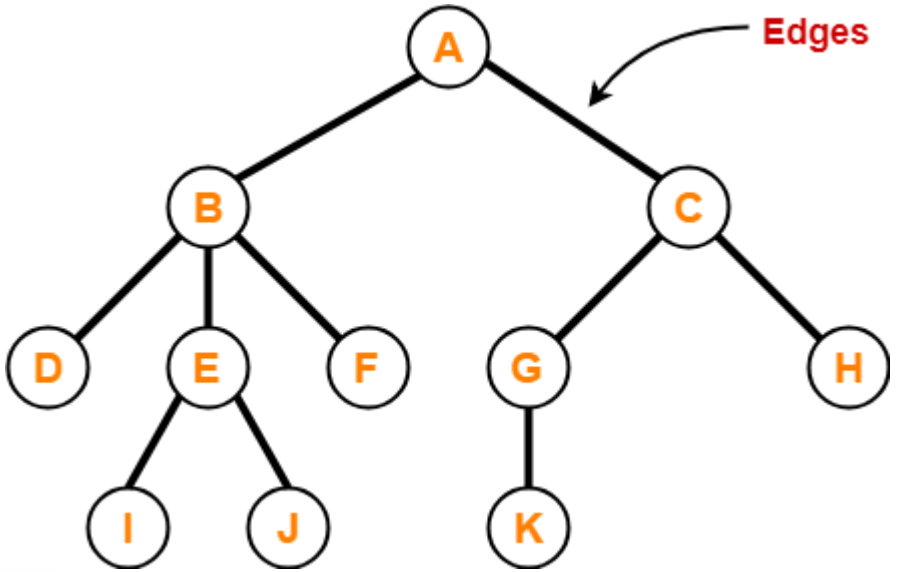
- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



Here, node A is the only root node.

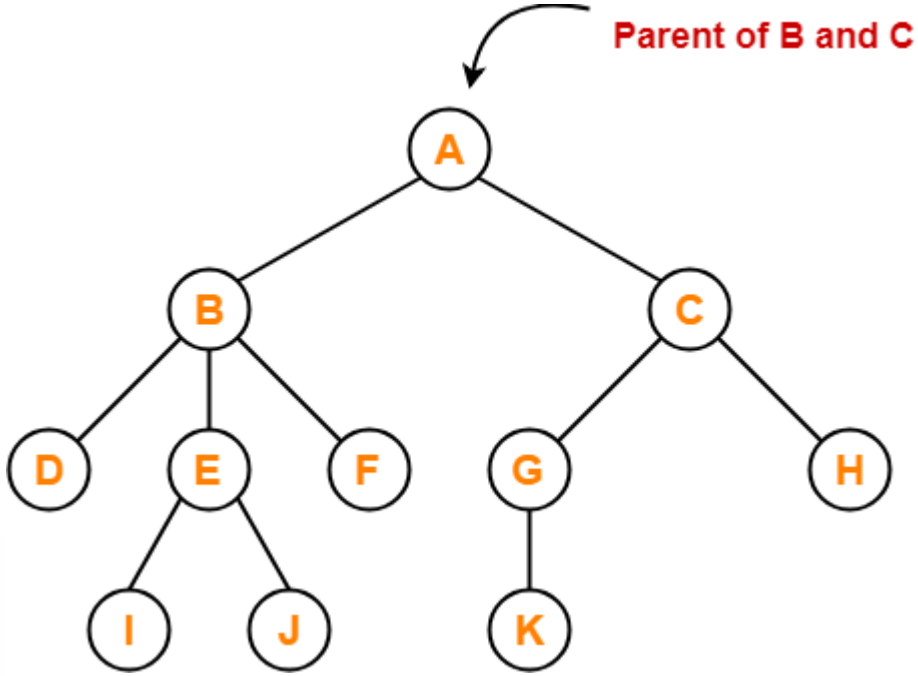
Edge

- 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.



Parent

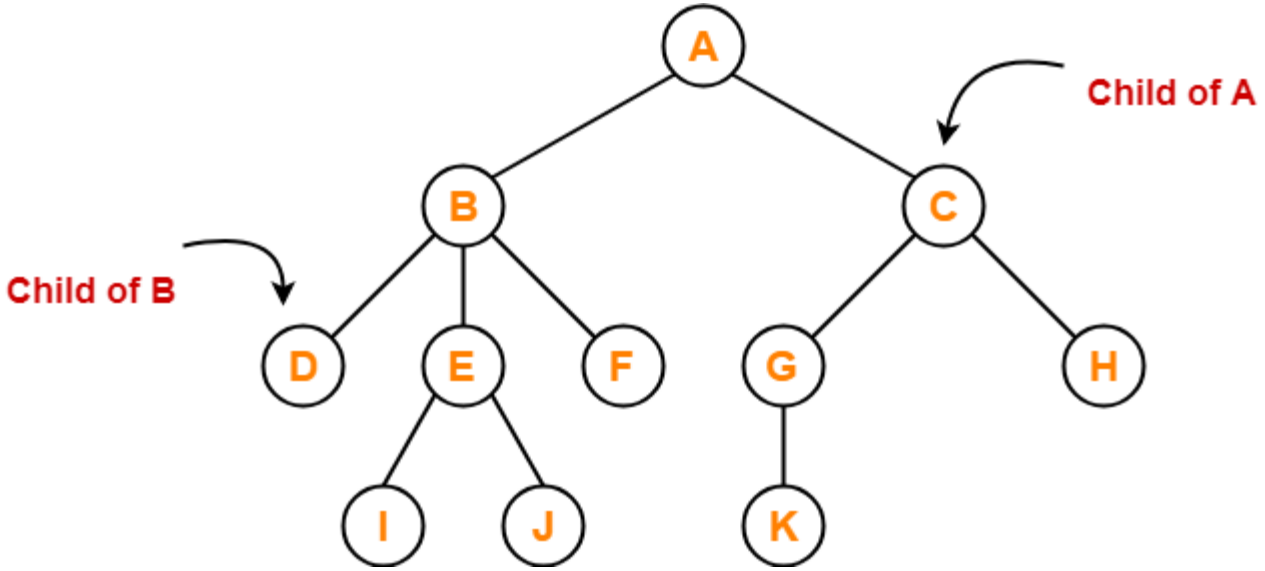
- The node which has a branch from it to any other node is called as a **parent node**.
- The node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.



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

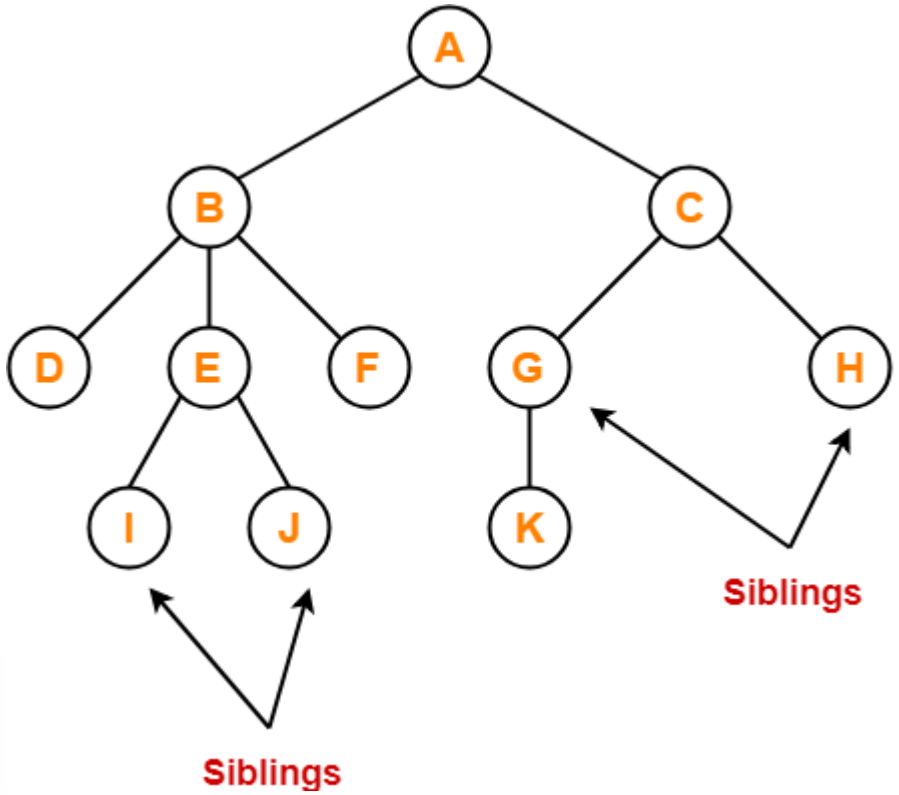
Child

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



Siblings

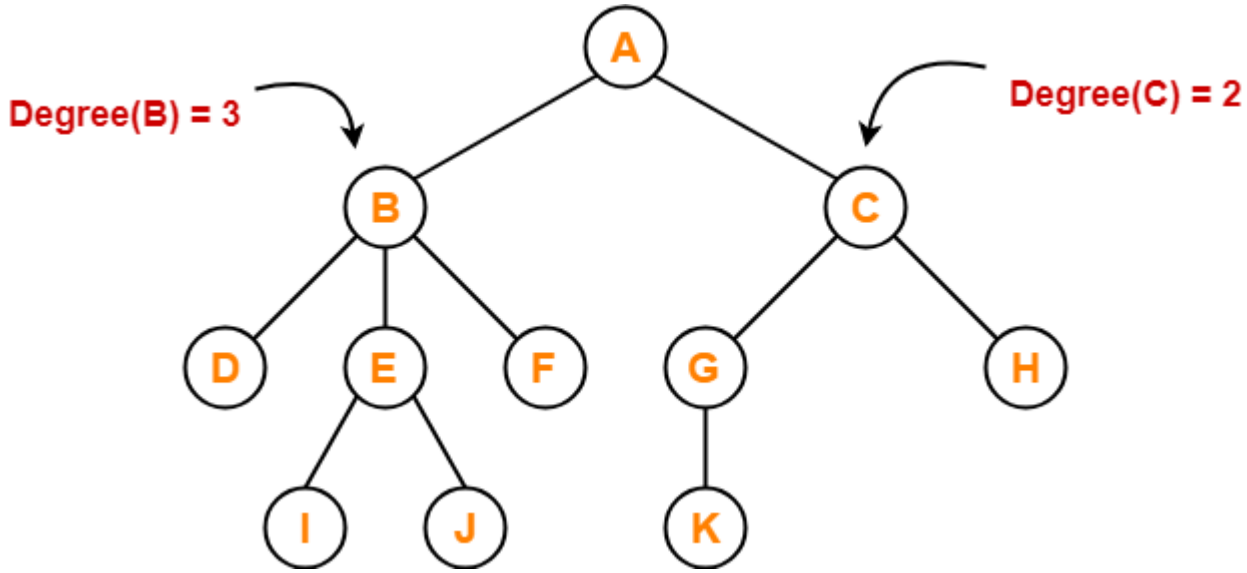
- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.



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

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.

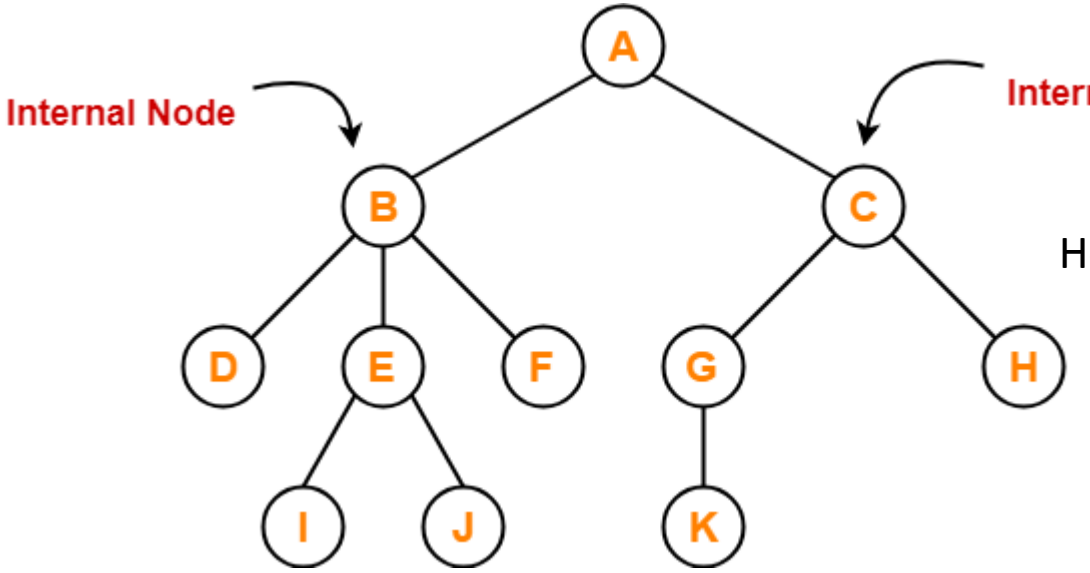


Here Degree of B is 3
Here Degree of A is 2
Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

Internal node

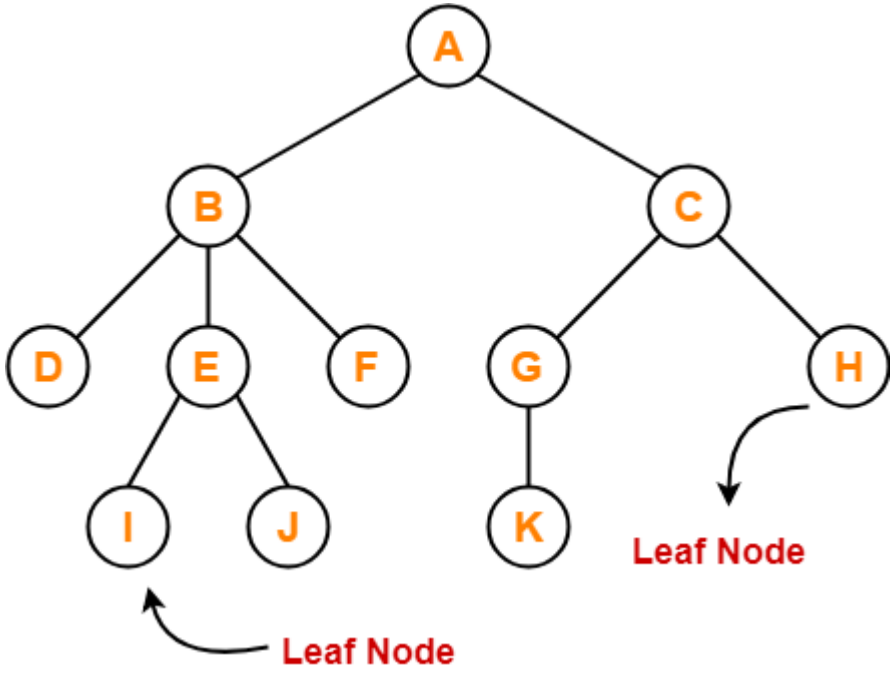
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.



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

Leaf node

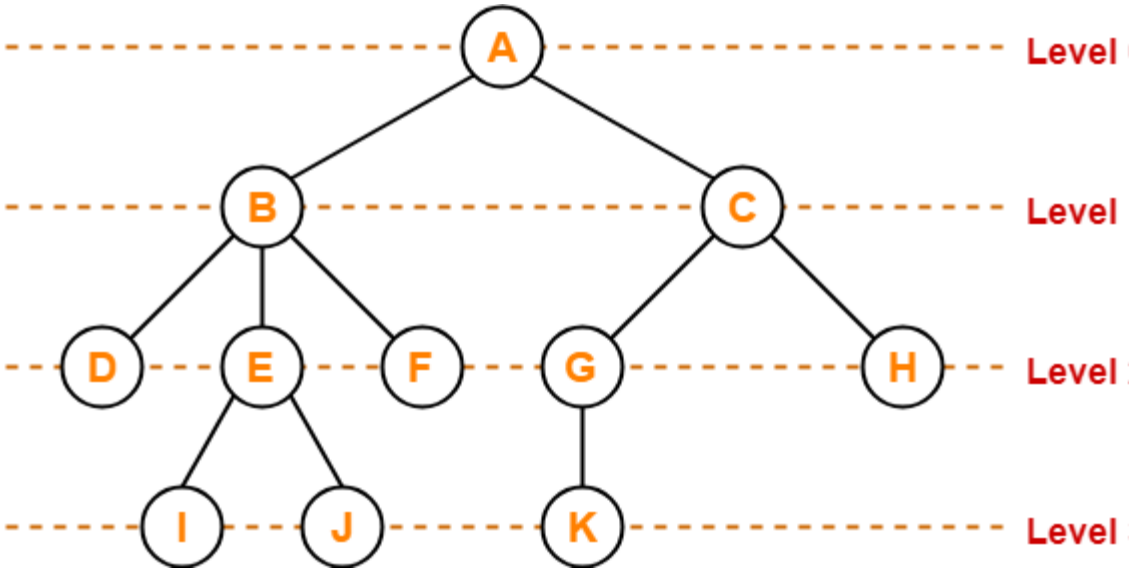
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.

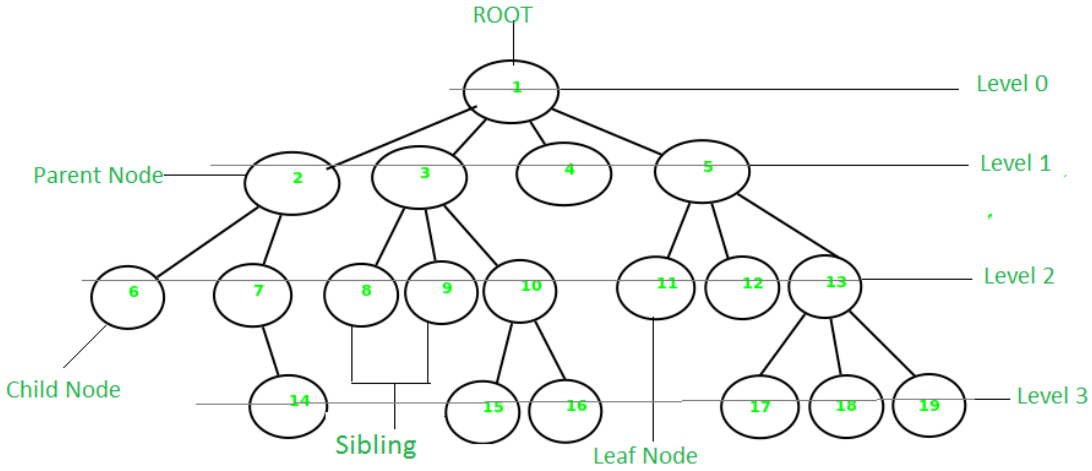


Here, nodes D, I, J, F, K and H are leaf nodes.

Level

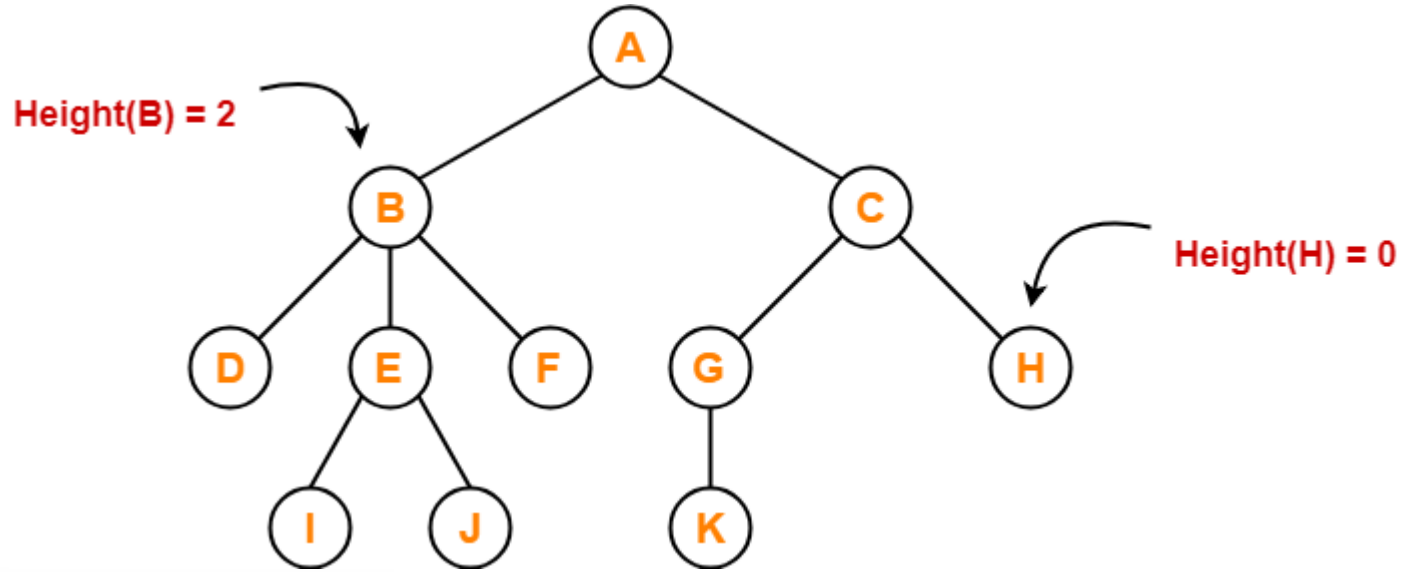
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.





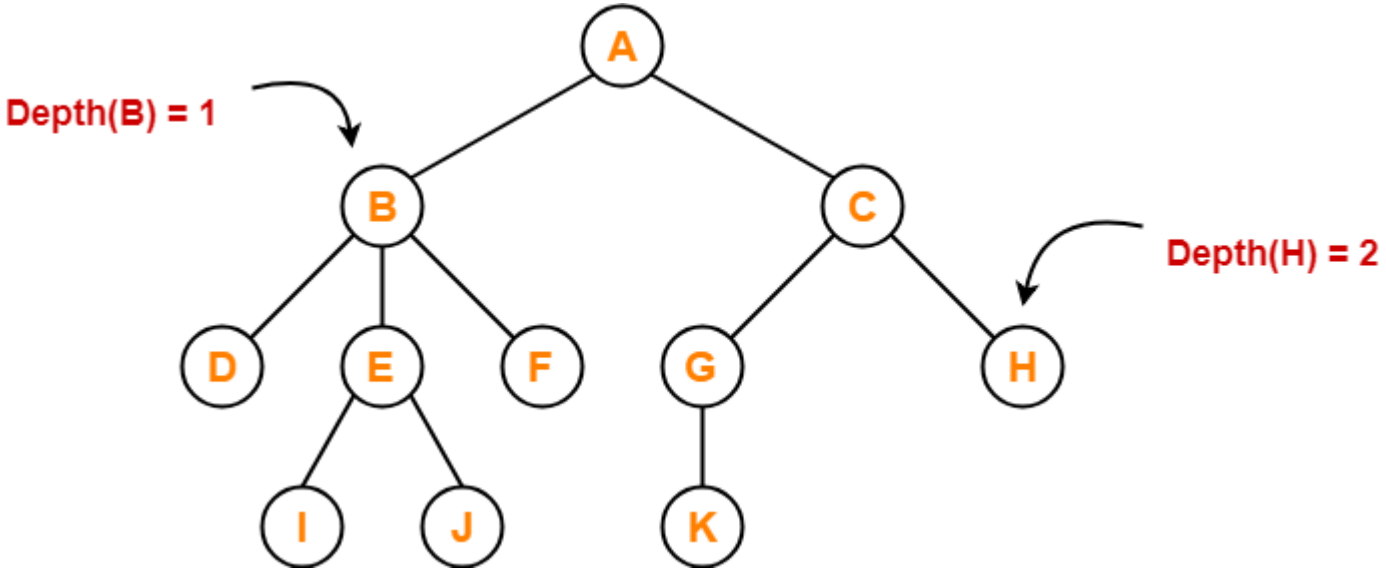
Height

- 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
-



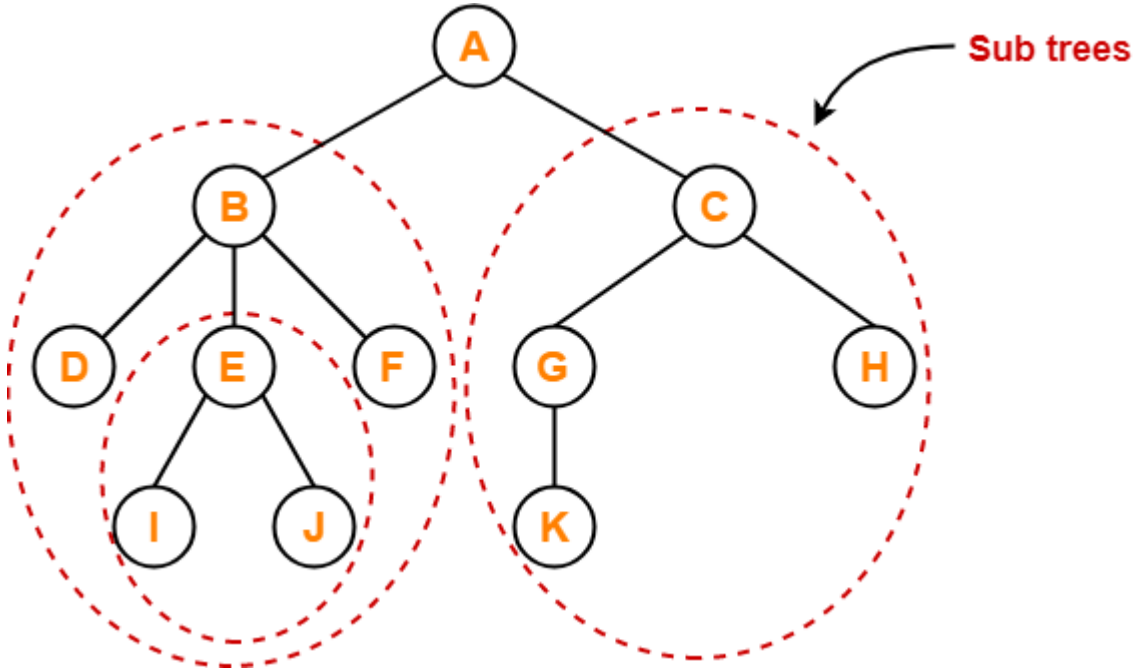
Depth

- 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.



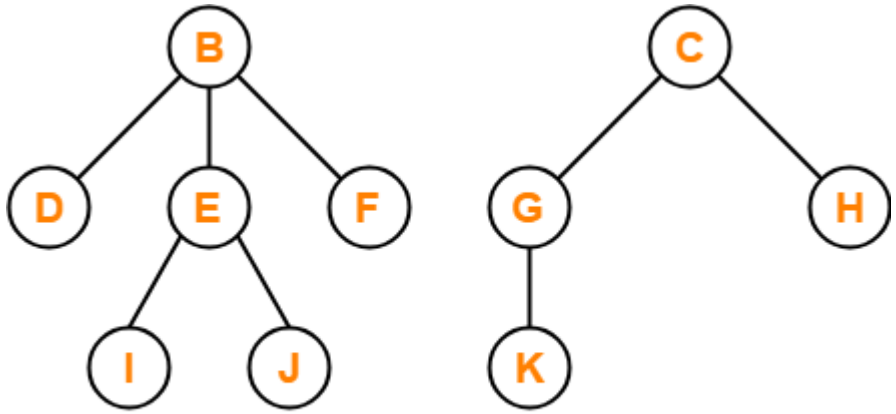
Subtree

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



Forest

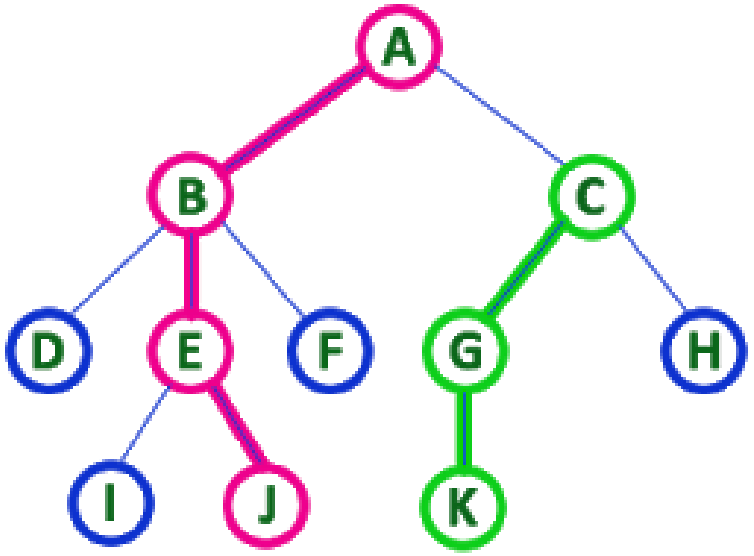
- A forest is a set of disjoint trees.



Forest

Path

- The sequence of consecutive edges from source node to destination node.



- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is

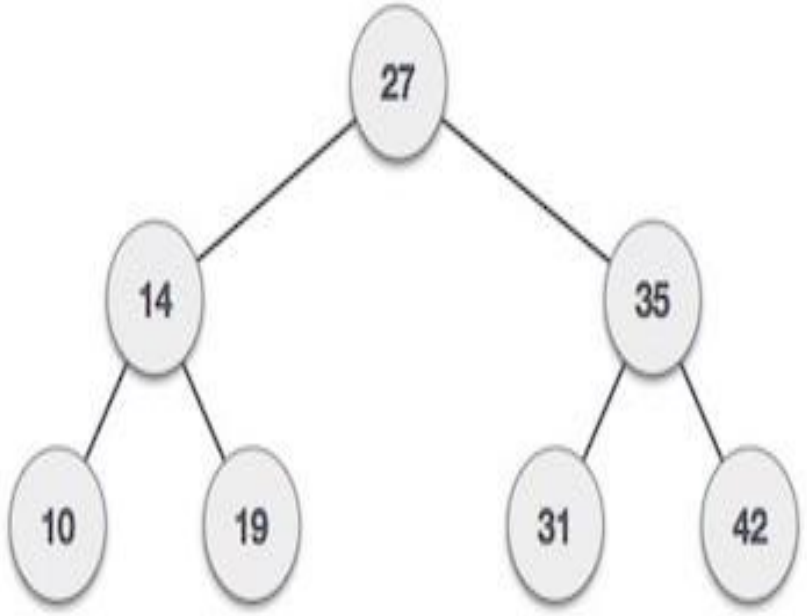
A - B - E - J

Here, 'Path' between C & K is

C - G - K

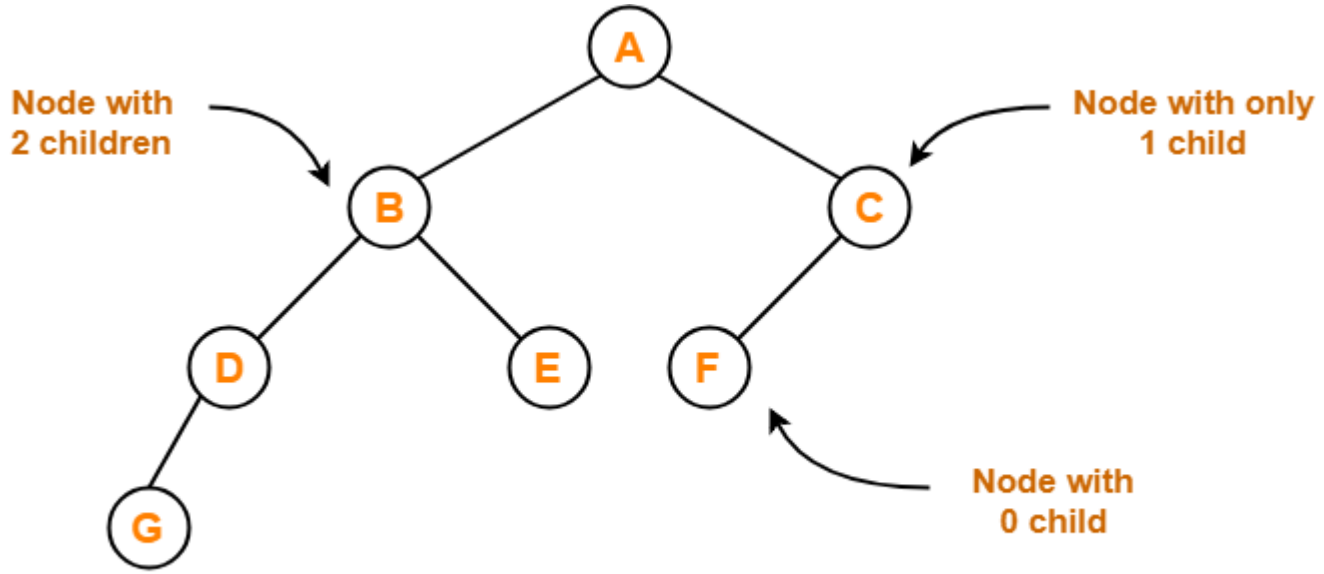
Key

- Key represents a value of a node based on which a search operation is to be carried out for a node.



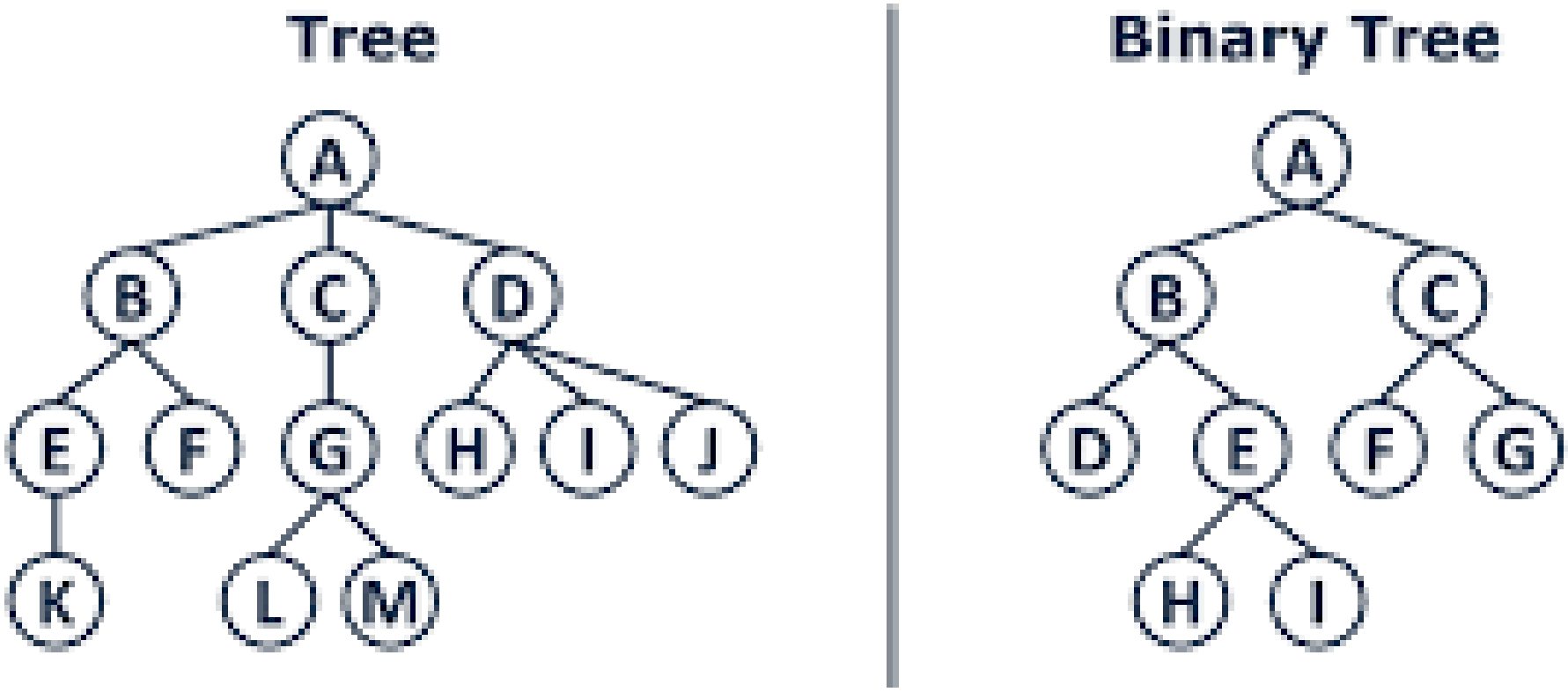
Binary Tree

- Binary tree is a special tree data structure in which each node can have at most 2 children.
- Thus, in a binary tree, Each node has either 0 child or 1 child or 2 children.



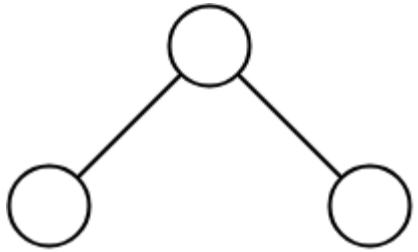
Binary Tree Example

Binary Tree



Unlabeled Binary Tree

- A binary tree is unlabeled if its nodes are not assigned any label.



Unlabeled Binary Tree

Number of different Binary Trees possible with 'n' unlabeled nodes $= \frac{{}^{2n}C_n}{n+1}$

Number of binary trees possible with 3 unlabeled nodes

$= {}^{2 \times 3}C_3 / (3 + 1)$

$= {}^{6}C_3 / 4$

$= 5$

Given ${}^{6}C_3$

Where $n = 6,\ r = 3$

$$\therefore {}^{n}C_r = \frac{n!}{r!(n-r)!}$$

$$= \frac{6!}{3!(6-3)!}$$

$$= \frac{6!}{3!(3)!}$$

$$= \frac{6 \times 5 \times 4 \times 3!}{3! \times 3 \times 2 \times 1}$$

$$= \frac{6 \times 5 \times 4}{3 \times 2 \times 1}$$

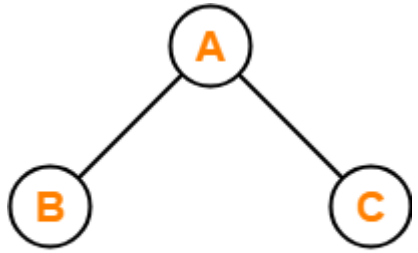
$$= \frac{120}{6}$$

$$= 20$$

Hence, ${}^{6}C_3$ is 20

Labeled Binary Tree

- A binary tree is labeled if all its nodes are assigned a label.



Labeled Binary Tree

Number of different Binary Trees possible with 'n' labeled nodes

$$= \frac{{}^{2n}C_n}{n+1} \times n!$$

Number of binary trees possible with 3 labeled nodes

$= \{ {}^{2 \times 3}C_3 / (3 + 1) \} \times 3!$

$= \{ {}^{6}C_3 / 4 \} \times 6$

$= 5 \times 6$

$= 30$

Given ${}^{6}C_3$

Where $n = 6,\ r = 3$

$$\therefore {}^{n}C_r = \frac{n!}{r!(n-r)!}$$

$$= \frac{6!}{3!(6-3)!}$$

$$= \frac{6!}{3!(3)!}$$

$$= \frac{6 \times 5 \times 4 \times 3!}{3! \times 3 \times 2 \times 1}$$

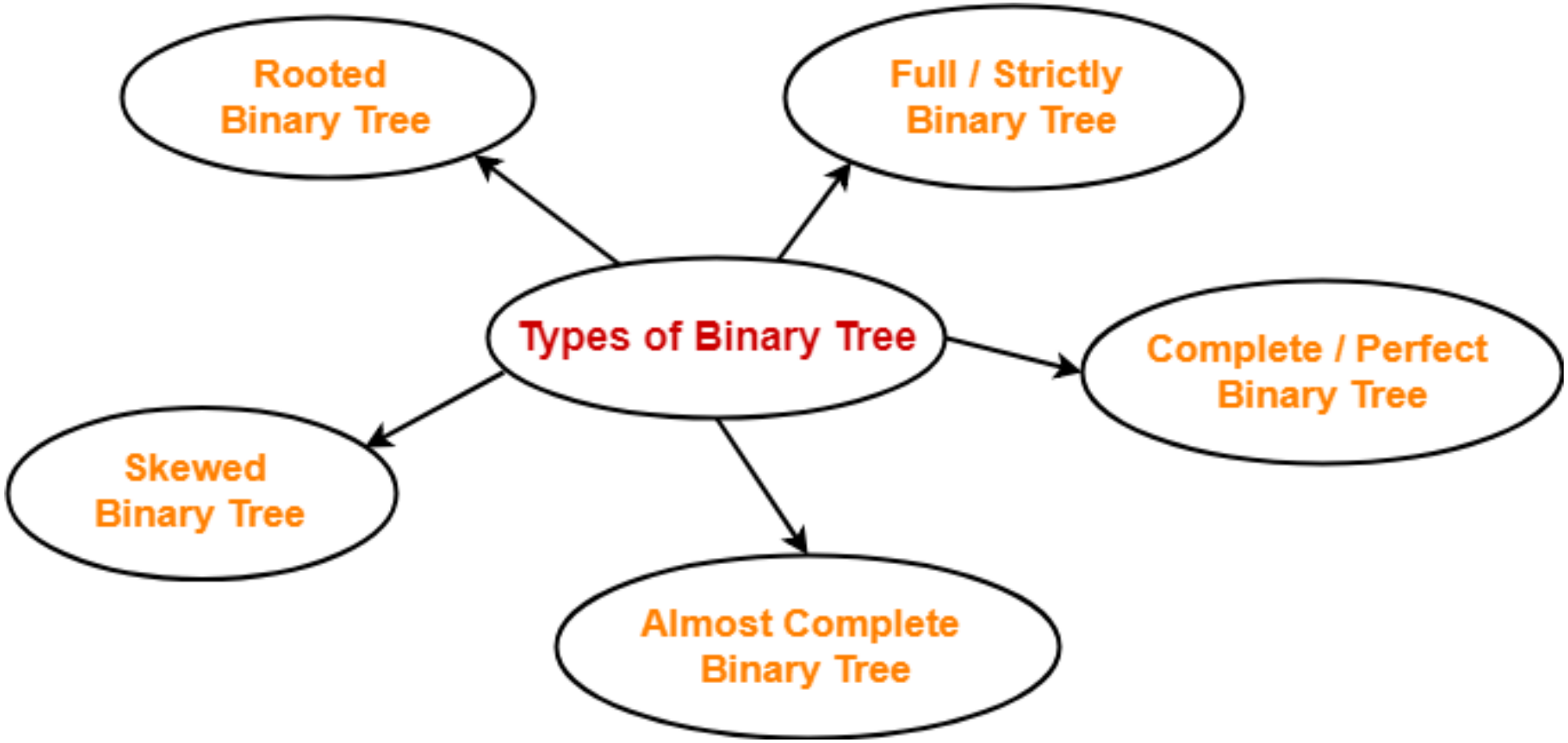
$$= \frac{6 \times 5 \times 4}{3 \times 2 \times 1}$$

$$= \frac{120}{6}$$

$$= 20$$

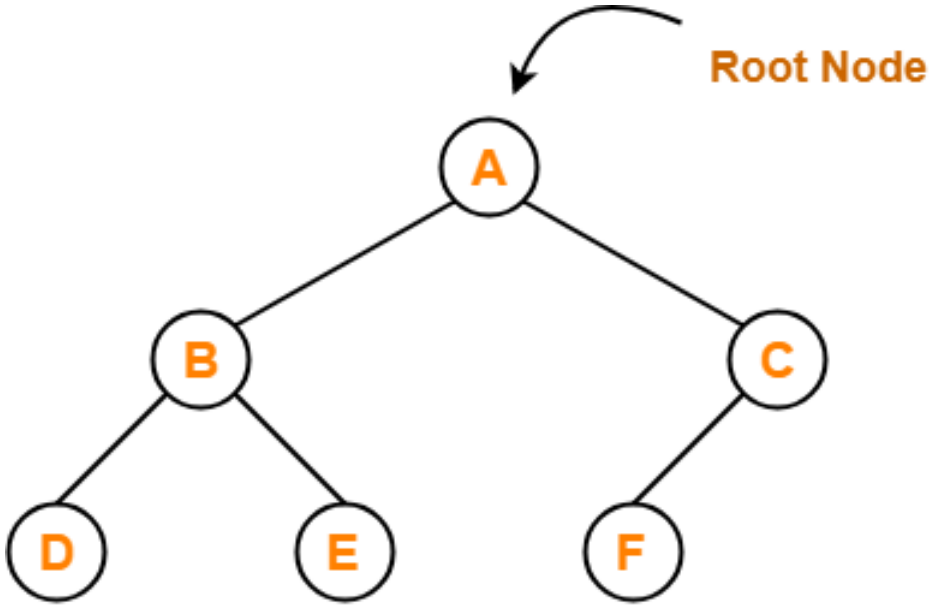
Hence, ${}^{6}C_3$ is 20

Types of Binary Tree



Rooted Binary Tree

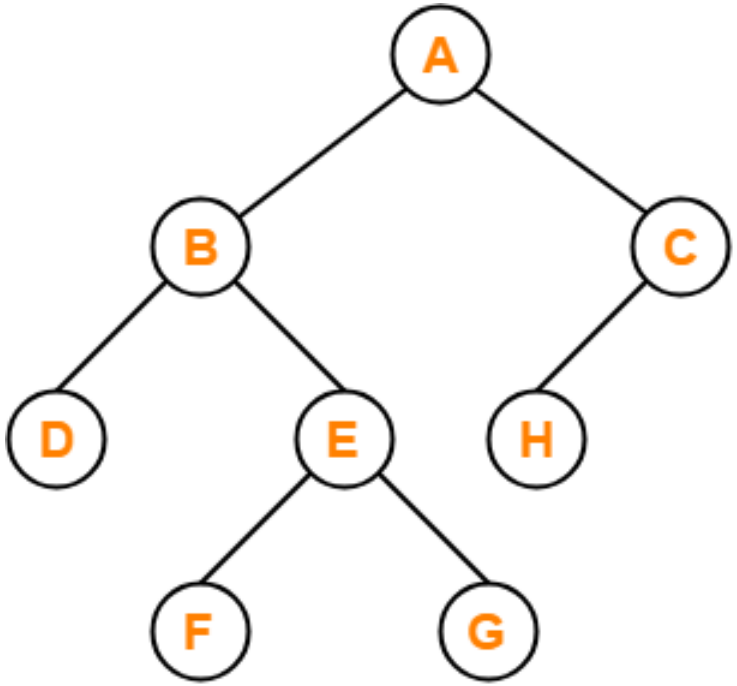
- A **rooted binary tree** is a binary tree that satisfies the following 2 properties-
- It has a root node.
- Each node has at most 2 children.

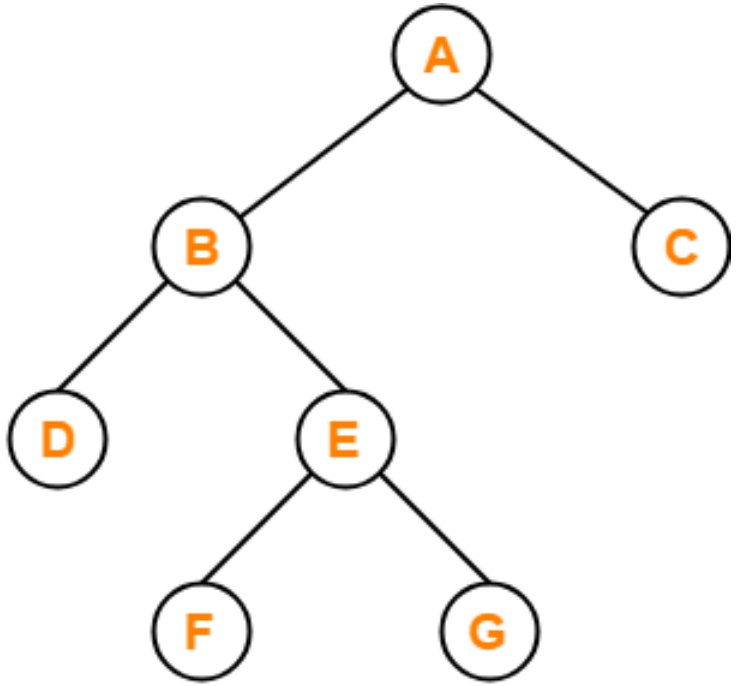


Rooted Binary Tree

Full/Strictly Binary Tree

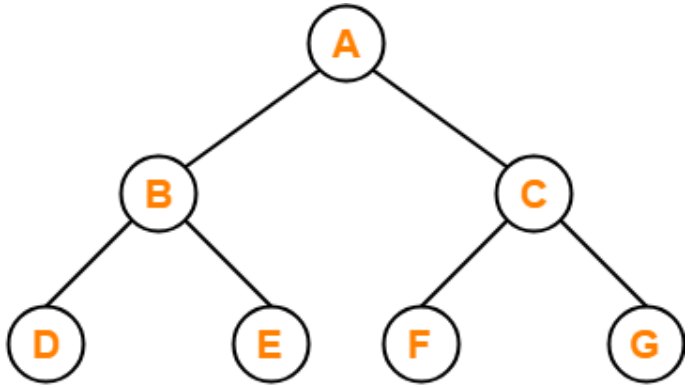
- Every node has either 0 or 2 children
- Full binary tree is also called as **Strictly binary tree**.





Complete/ Prefect Binary Tree

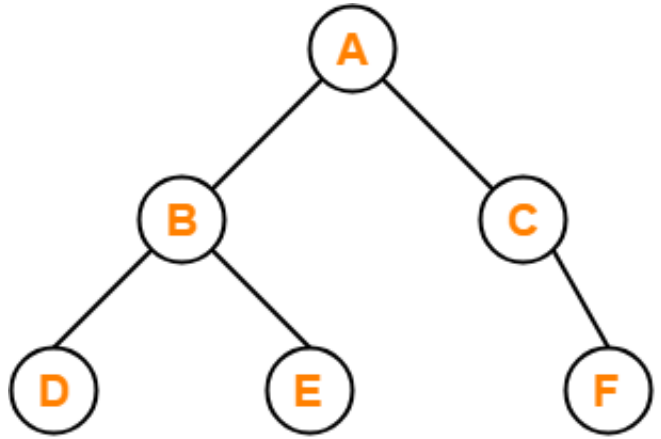
- In a complete binary tree, every level, except possibly the last, is completely filled.
- Second, all nodes appear as far left as possible.
- Complete binary tree is also called as **Perfect binary tree**.



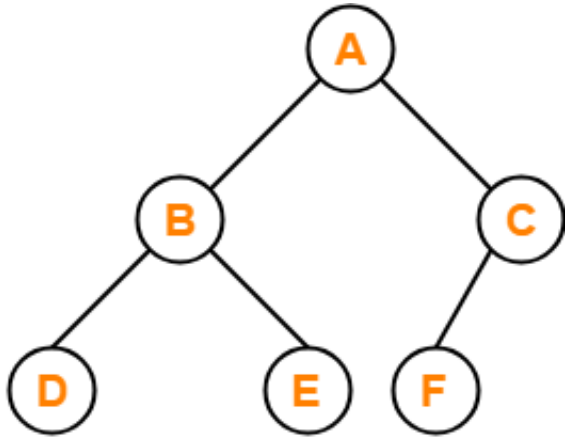
✓

Almost complete Types of Binary Tree

- An **almost complete binary tree** is a binary tree that satisfies the following 2 properties-
- All the levels are completely filled except possibly the last level.
- The last level must be strictly filled from left to right.



X



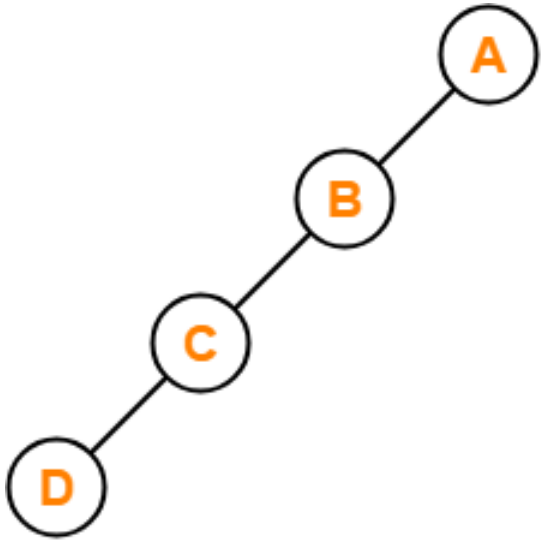
✓

Skewed Binary Tree

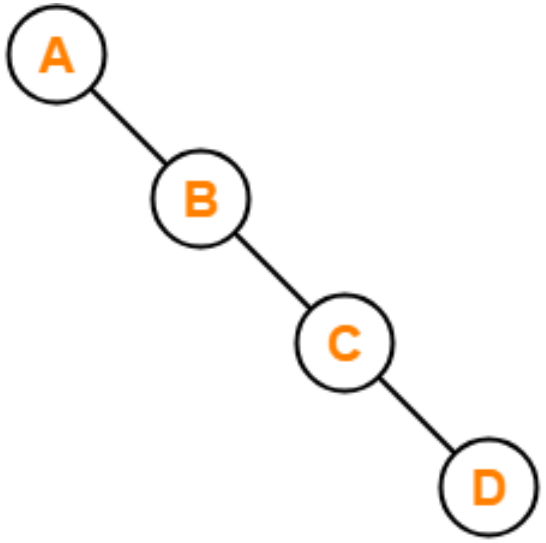
- All the nodes except one node has one and only one child.
- The remaining node has no child.

OR

- A **skewed binary tree** is a binary tree of n nodes such that its depth is (n-1).



Left Skewed Binary Tree



Right Skewed Binary Tree

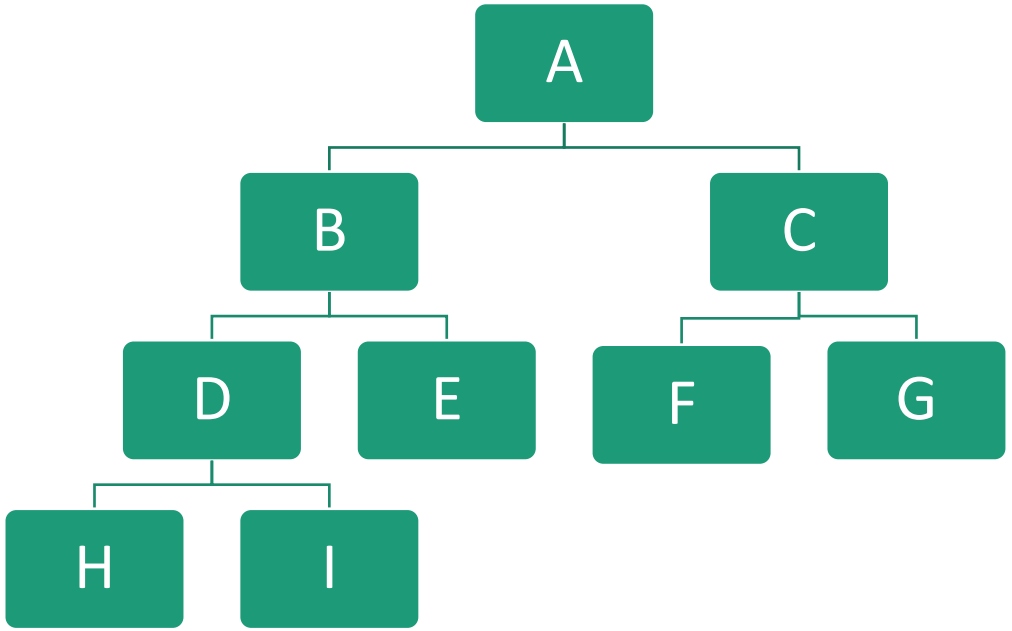
Binary Tree representation

- **Array Representation**
- **Linked List Representation**

Array Representation of a Binary Tree

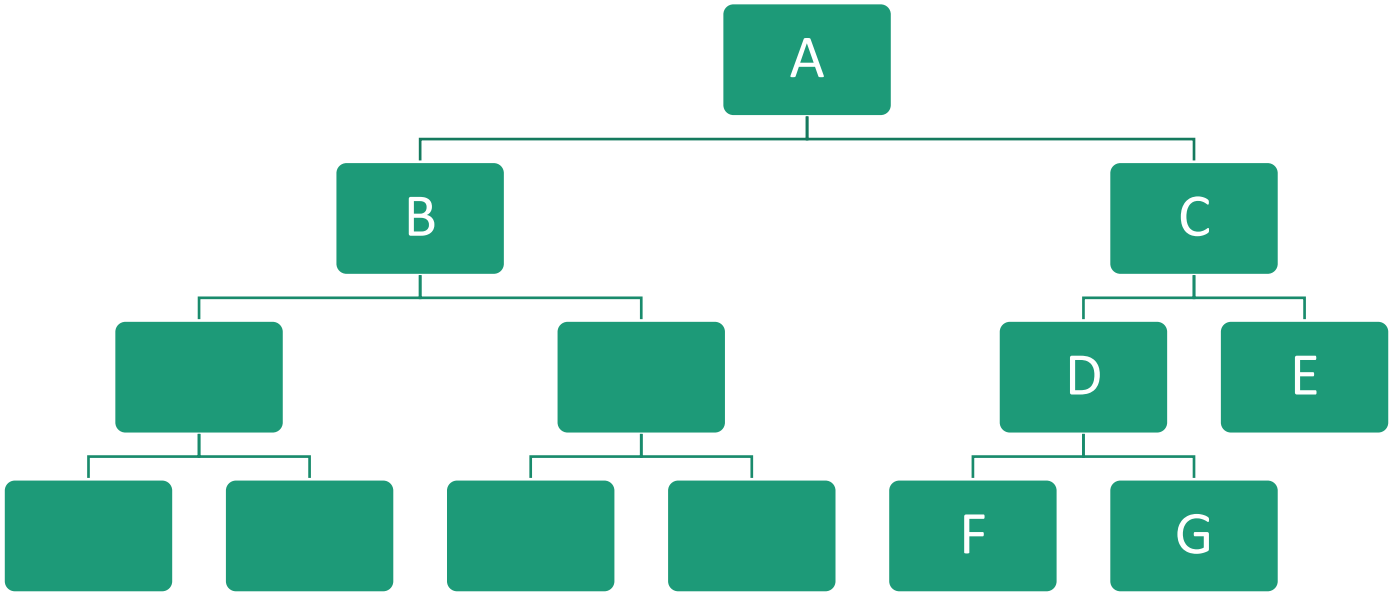
- Using 1-D Array
- **Nodes are numbered sequentially level by level left to right.**
- **Even empty nodes are numbered**
- When data of the tree is stored in an array then the number appearing against the node will work as indices of the node in the array

Array Representation of a Binary Tree



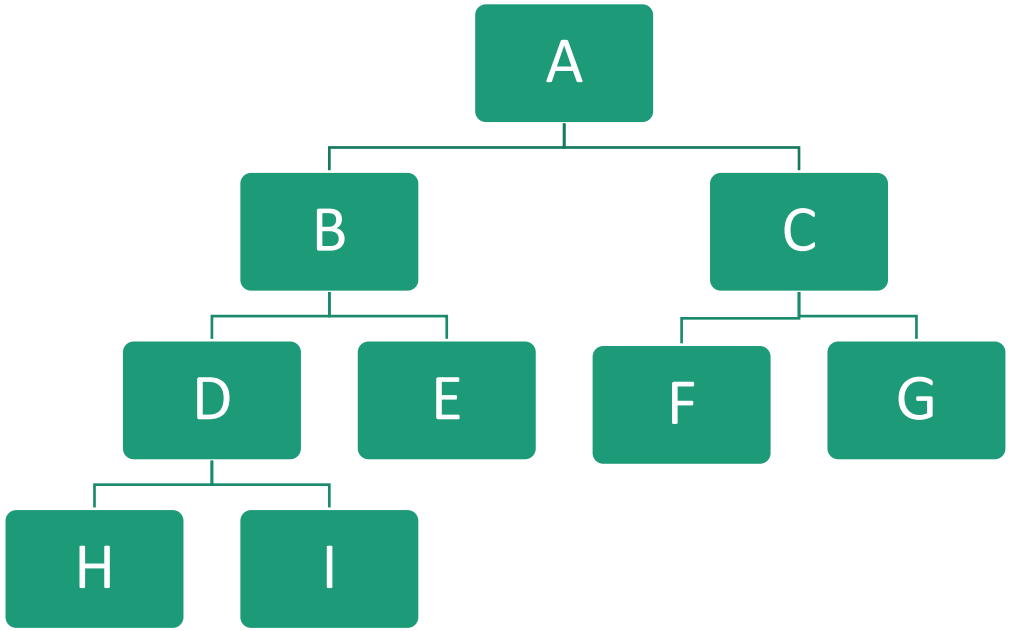
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
A	B	C	D	E	F	G	H	I

Array Representation of a Binary Tree



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
A	B	C			D	E					F	G

Array Representation of a Binary Tree



Node in position p

Left child=2p+1

Right child=2p+2

For C= 2,

LC = 2*2+1 = 5

RC = 2*2+2 = 6

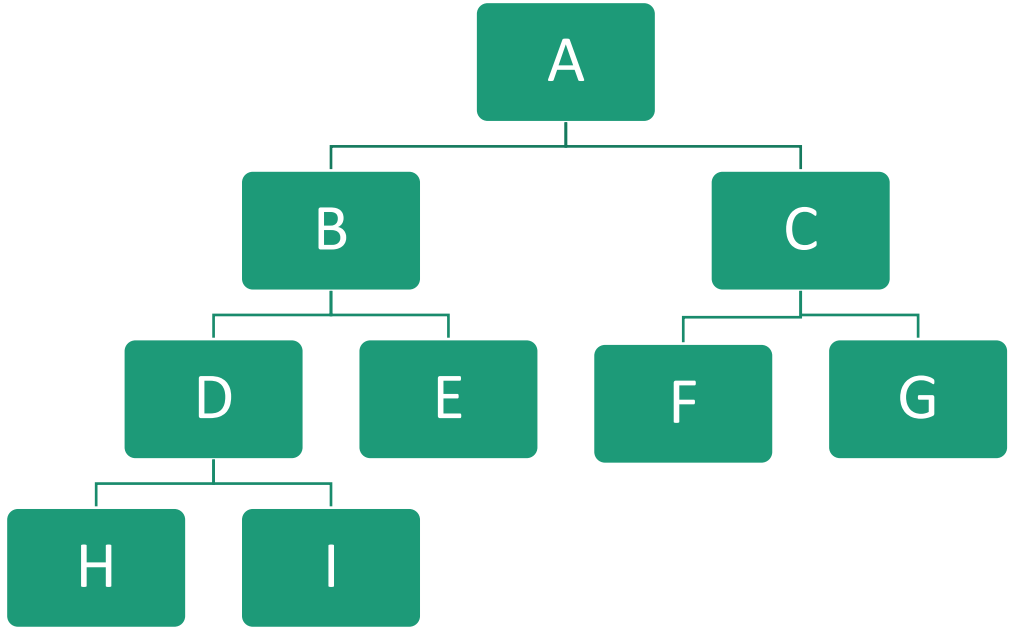
For D= 3,

LC = 2*3+1 = 7

RC = 2*3+2 = 8

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
A	B	C	D	E	F	G	H	I

Array Representation of a Binary Tree



[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
A	B	C	D	E	F	G	H	I

Node in position p = sibling

Given a Left child at position p then

Right Sibling = p+1

Given a Right child at position p then

Left Sibling = p-1

For C= I,
LS = 8-1 = 7 i.e. H

For D=, P= 3,
RS = 3+1 = 4 i.e. E

Linked representation of Binary Tree

- A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

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

Tree has 3 fields in a node-

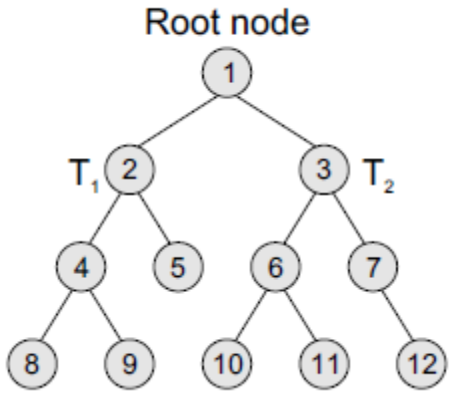
- Data field
- Address of left child
- Address of right child

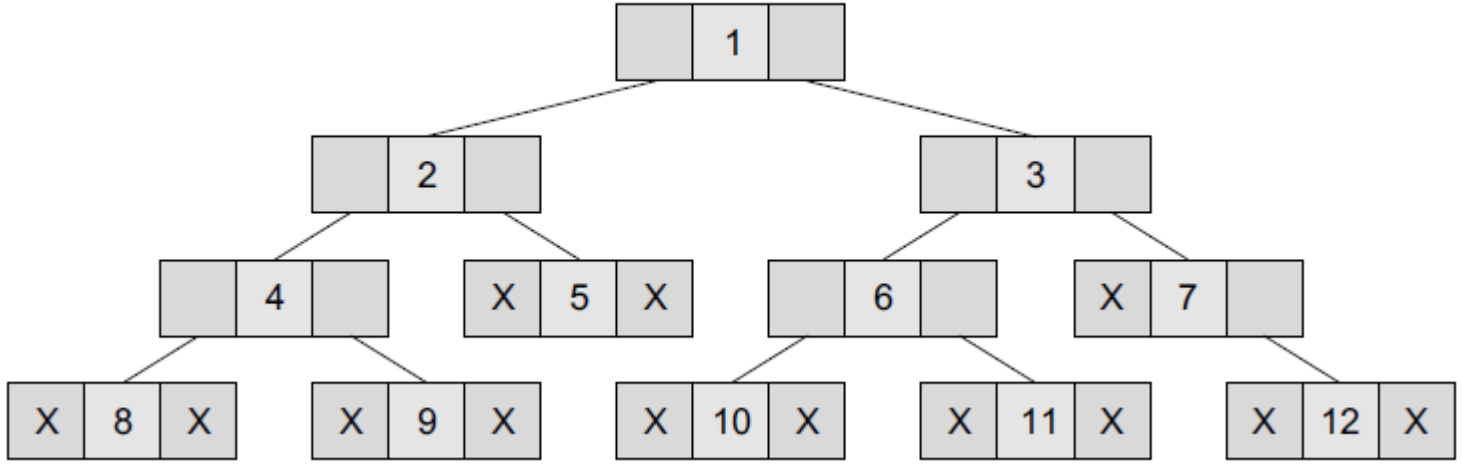
Linked representation of Binary Tree

- Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree.
- If ROOT = NULL, then the tree is empty.

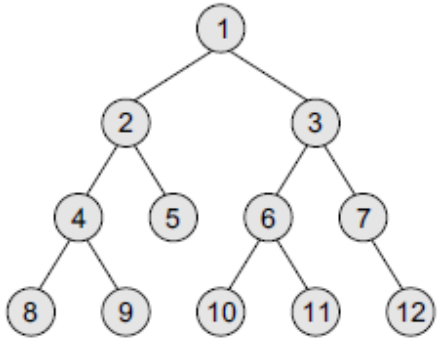
Linked representation of Binary Tree

- Consider the binary tree given and the schematic diagram of the linked representation of the binary tree shown in Fig.





Linked representation of Binary Tree

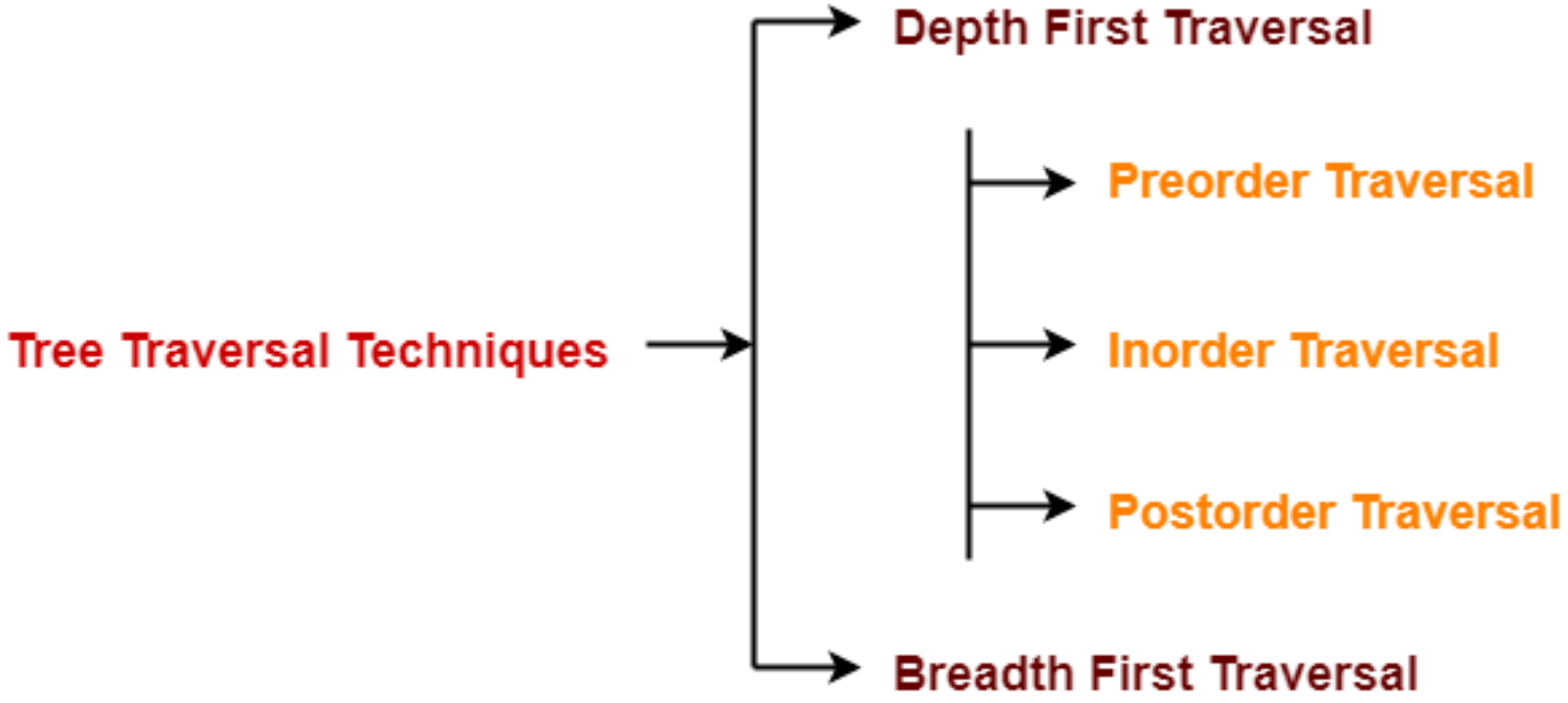


		LEFT	DATA	RIGHT
ROOT 3	1	–1	8	–1
	2	–1	10	–1
→	3	5	1	8
	4			
	5	9	2	14
	6			
	7			
	8	20	3	11
	9	1	4	12
	10			
	11	–1	7	18
	12	–1	9	–1
	13			
	14	–1	5	–1
→	15			
15 AVAIL	16	–1	11	–1
	17			
	18	–1	12	–1
	19			
	20	2	6	16

Linked representation of binary tree T

Tree Traversal

- Tree Traversal refers to the process of visiting each node in a tree data structure exactly once.



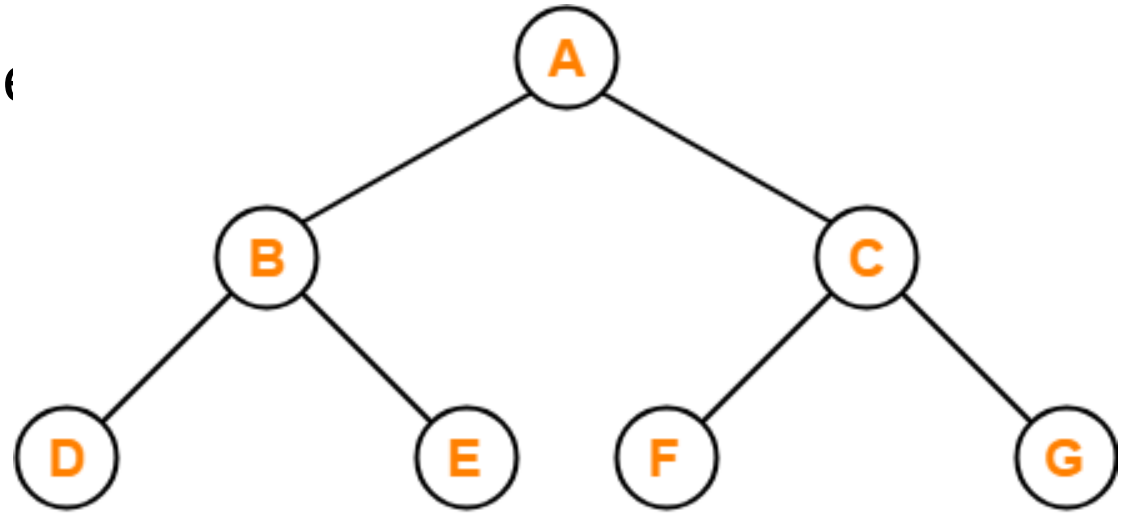
Depth First Traversal

Following three traversal techniques fall under Depth First Traversal-

- Preorder Traversal
- Inorder Traversal
- Postorder Traversal

Preorder Traversal

- Visit root node
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- **Root → Left → Right**
- display(root->data)
- preorder(root->left)
- preorder(root->right)



Preorder Traversal : A , B , D , E , C , F , G

Preorder Traversal

Algorithm for pre-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            Write TREE -> DATA
Step 3:            PREORDER(TREE -> LEFT)
Step 4:            PREORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

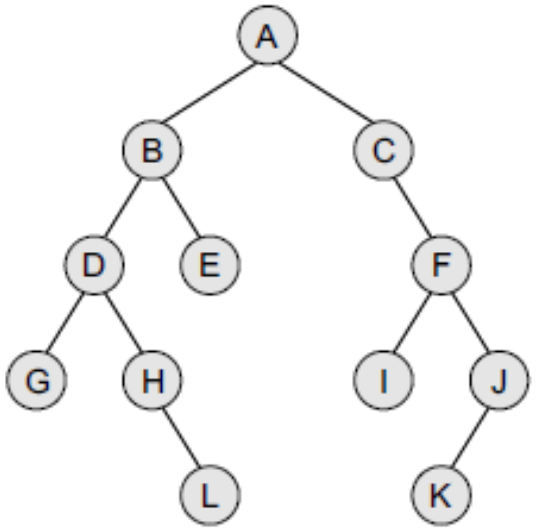
Pre-order traversal is also called as *depth-first traversal*.
Pre-order algorithm is also known as the NLR traversal algorithm (Node-Left-Right).

Preorder Traversal

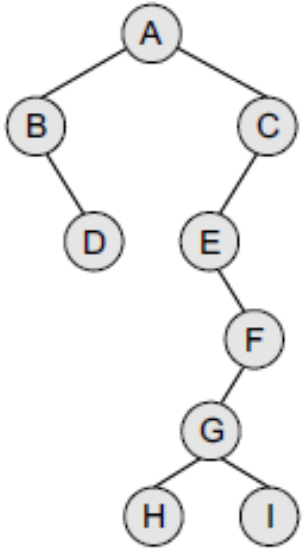
```
void preorder(struct node *tree)
{
        if(tree!=NULL)
         {
                printf("%d\n",tree->num);
                preorder(tree->left);
                preorder(tree->right);
         }
}
```

Preorder Traversal

- In Figs (a) and (b), find the sequence of nodes that will be visited using pre-order traversal algorithm.



(a)



(b)

Solution

(a)TRAVERSAL ORDER:

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

(b) TRAVERSAL ORDER:

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

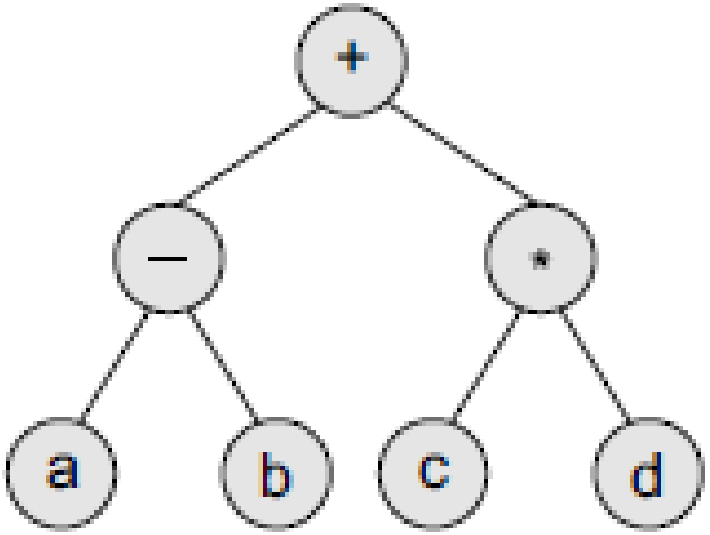
Preorder Traversal

Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

For example, consider the expressions given Exp = (a – b) + (c * d)

This expression can be represented using a binary tree as

When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression.

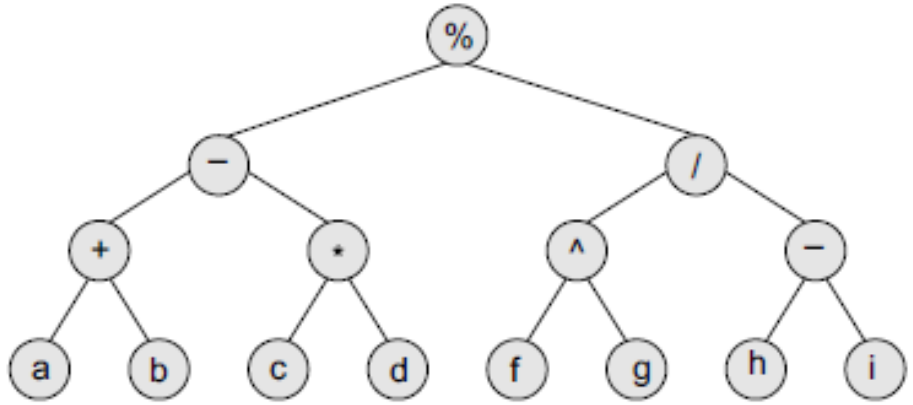


+ – a b * c d

Preorder Traversal

Given an expression, Exp = ((a + b) – (c * d)) % ((f ^g) / (h – i)), construct the corresponding binary tree.

This expression can be represented using a binary tree as

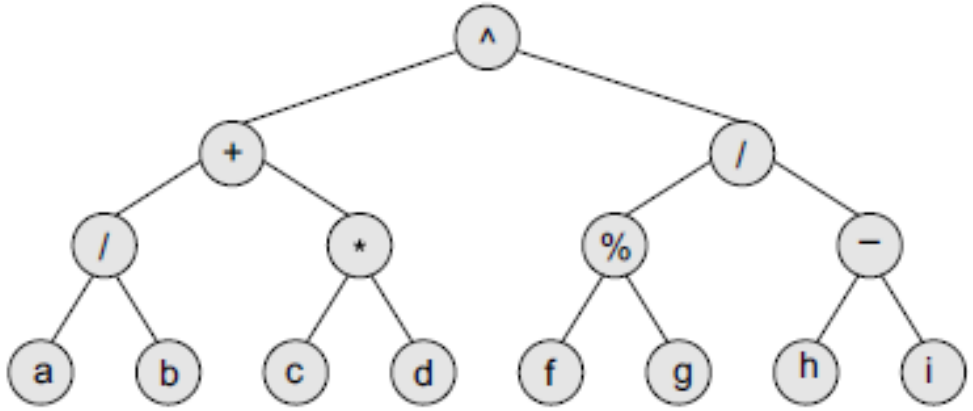


When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression.

% – + a b * c d / ^ f g – h i

Preorder Traversal

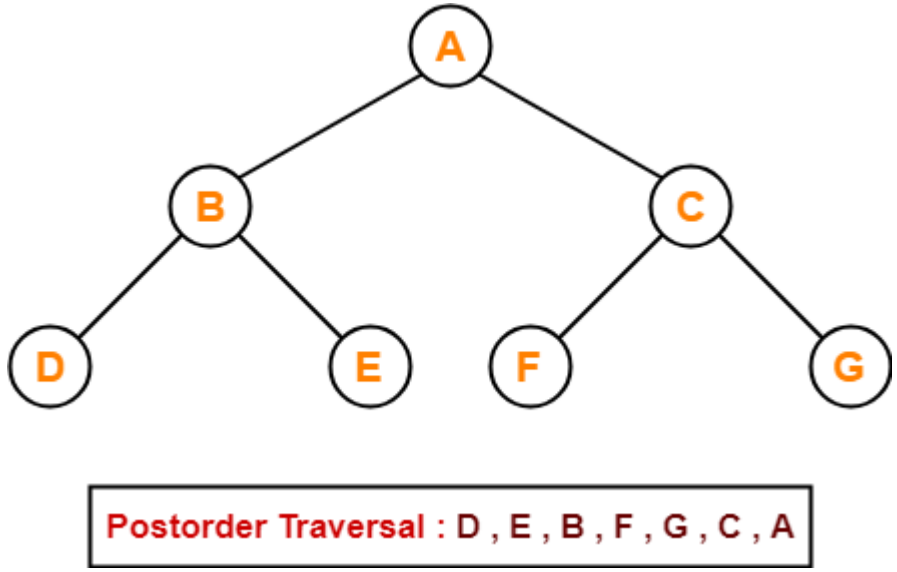
Given the binary tree, write down the expression that it represents.



Expression for the above binary tree is [{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]
When we traverse the elements of a tree using the pre-order traversal algorithm, the expression that we get is a prefix expression^ + / a b * c d / % f g – h i

Postorder Traversal

- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- Visit the root node
- **Left → Right → Root**
- postorder(root->left)
- postorder(root->right)
- display(root->data)



Post-order Traversal

- Algorithm for post-order traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
       [END OF LOOP]
Step 5: END
```

Post-order algorithm is also known as the LRN traversal algorithm (Left-Right-Node).

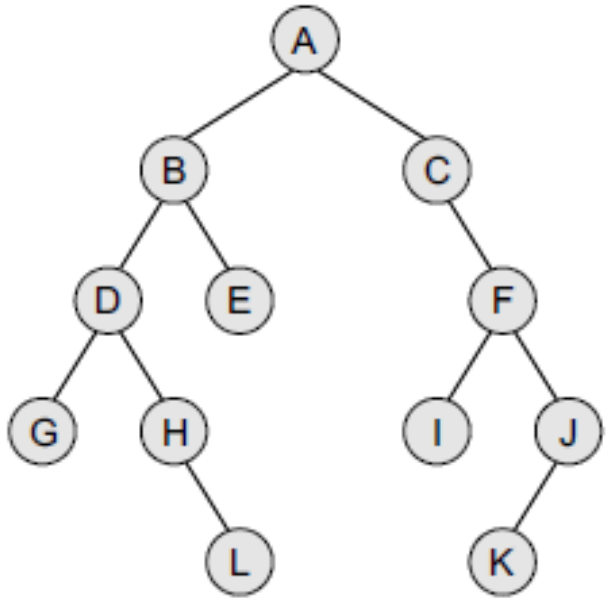
Post-order traversals are used to extract postfix notation from an expression tree.

Post-order Traversal

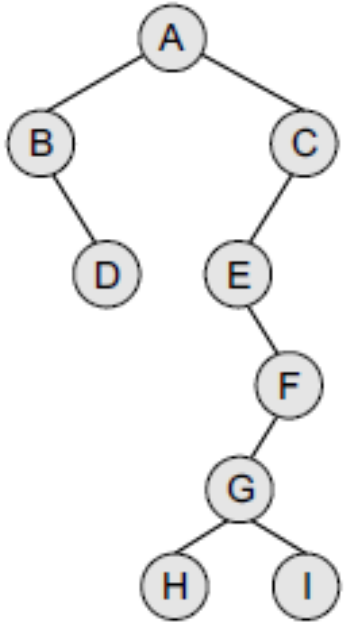
```
void postorder(struct node *tree)
{
        if(tree!=NULL)
         {
                postorder(tree->left);
                postorder(tree->right);
                printf("%d\n",tree->num);
         }

}
```

Post-order Traversal

- For the trees given, write the sequence of nodes that will be visited using post-order traversal algorithm.



(a)

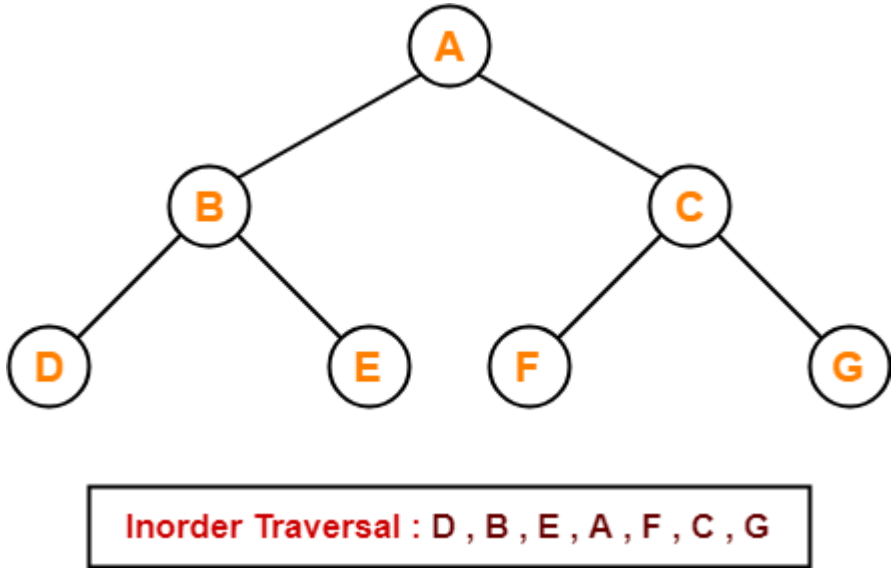


(b)

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

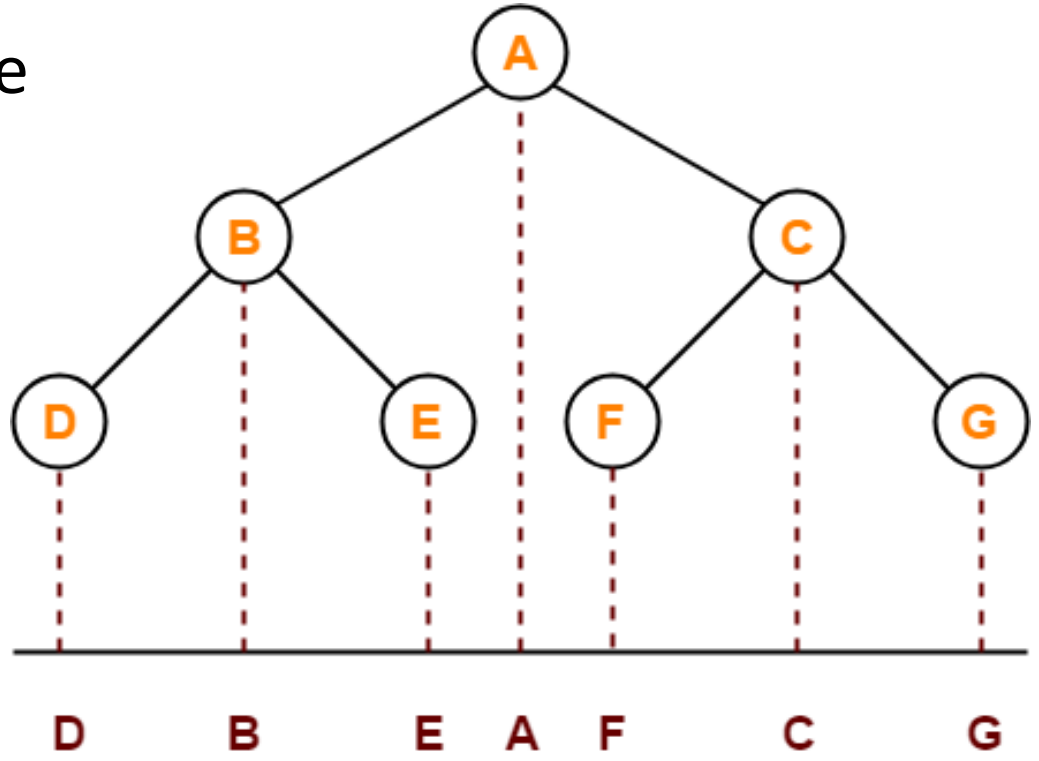
Inorder Traversal

- **Inorder traversal**
- First, visit all the nodes in the left subtree
- Then the root node
- Visit all the nodes in the right subtree
- **Left → Root → Right**
- inorder(root->left)
- display(root->data)
- inorder(root->right)



Inorder Traversal

Keep a plane mirror horizontally at the bottom of the tree and take the projection of all the nodes.



Inorder Traversal : D , B , E , A , F , C , G

Inorder Traversal

- Algorithm for Inorder Traversal

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
       [END OF LOOP]
Step 5: END
```

In-order traversal is also called as *symmetric traversal*.

In-order algorithm is also known as the LNR traversal algorithm (Left-Node-Right).

Inorder Traversal

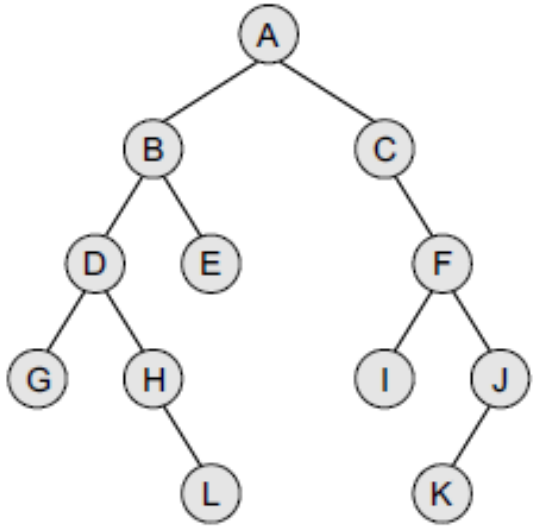
```
void inorder(struct node *tree)
{
        if(tree!=NULL)
            {
                inorder(tree->left);
                printf("%d\n",tree->num);
                inorder(tree->right);
            }
}
```

Inorder Traversal

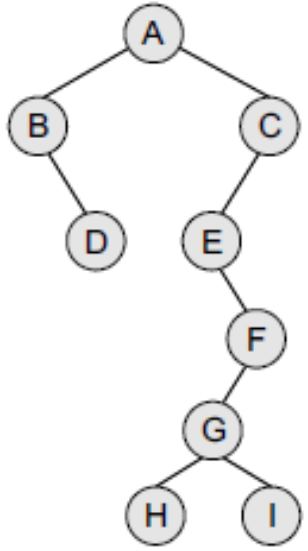
- In-order traversal algorithm is usually used to display the elements of a binary search tree.
- Here, all the elements with a value lower than a given value are accessed before the elements with a higher value.
- We will discuss binary search trees in detail

Inorder Traversal

- For the trees given find the sequence of nodes that will be visited using in-order traversal algorithm.



(a)



(b)

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

Applications

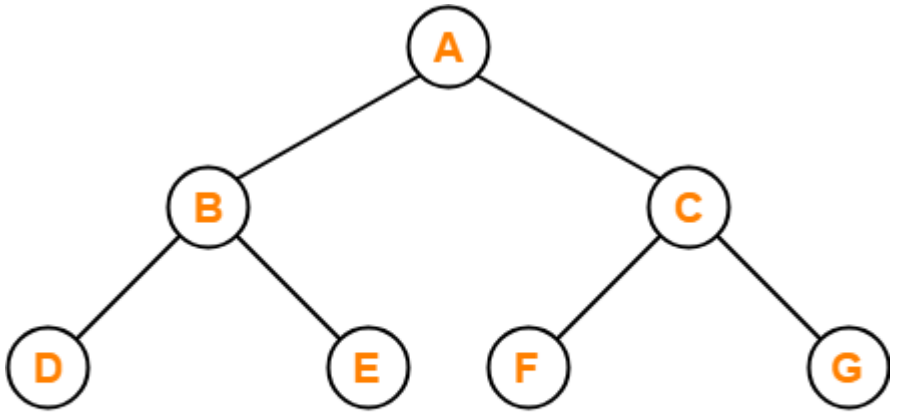
- Preorder traversal is used to get prefix expression of an expression tree.
- Preorder traversal is used to create a copy of the tree.
- Inorder traversal is used to get infix expression of an expression tree.
- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree. This is because it deletes the children first and then it deletes the parent.

Applications

- Preorder traversal is used to get prefix expression of an expression tree.
- Inorder traversal is used to get infix expression of an expression tree.
- Postorder traversal is used to get postfix expression of an expression tree.

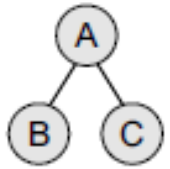
Breadth First Traversal

- Breadth First Traversal of a tree prints all the nodes of a tree level by level.
- Breadth First Traversal is also called as **Level Order Traversal**.
- All the nodes at a level are accessed before going to the next level.



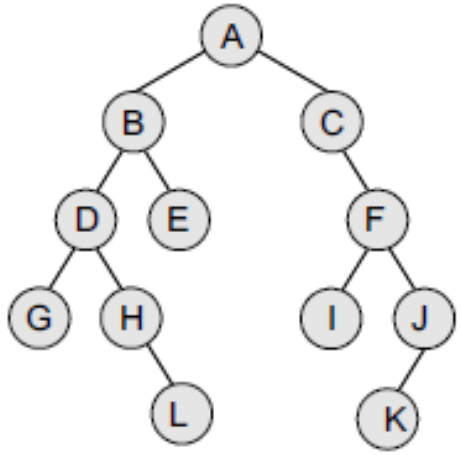
Level Order Traversal : A , B , C , D , E , F , G

Breadth First Traversal



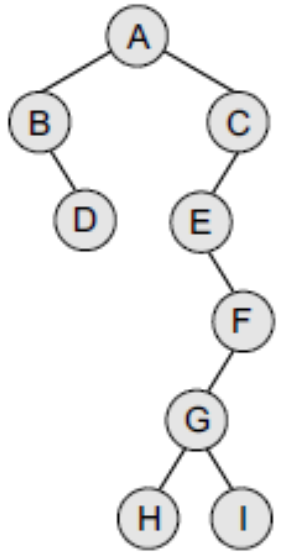
(a)

TRAVERSAL ORDER:
A, B, and C



(b)

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

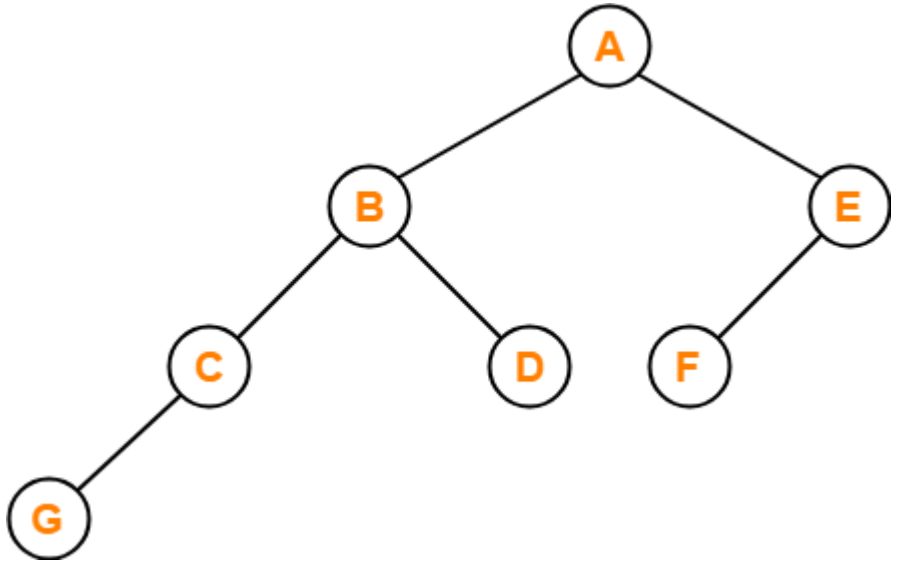


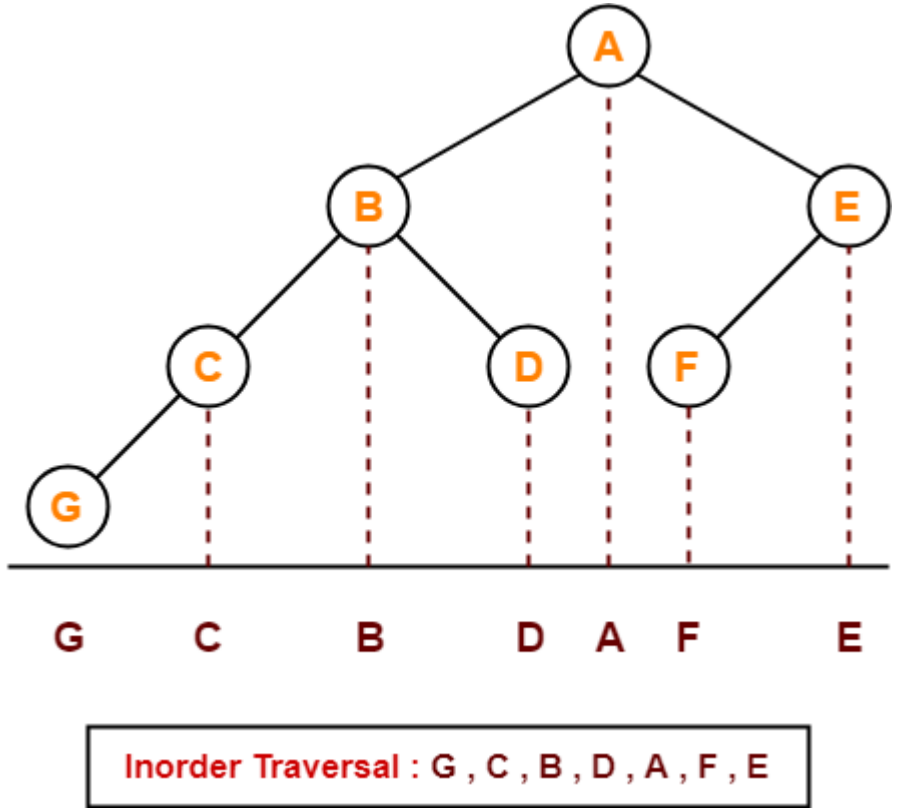
(c)

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

PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

- If the binary tree in figure is traversed in inorder, then the order in which the nodes will be visited is ____?



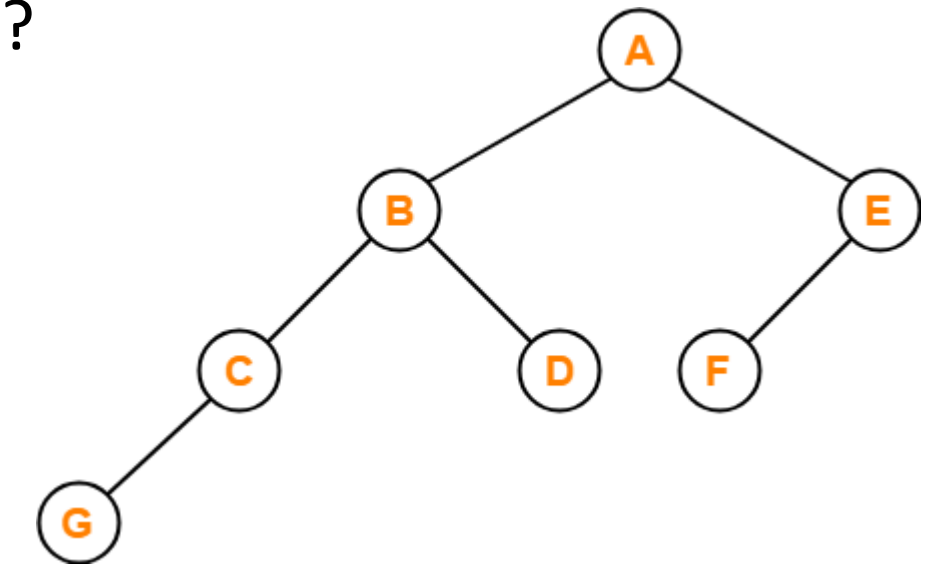


Inorder Traversal : G , C , B , D , A , F , E

PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

- Which of the following sequences denotes the postorder traversal sequence of the tree shown in figure?

FEGCBDBA
GCBDAFE
GCDBFEA
FDEGCBA



Postorder Traversal : G , C , D , B , F , E , A

Thus, Option (C) is correct.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

- Let LASTPOST, LASTIN, LASTPRE denote the last vertex visited in a postorder, inorder and preorder traversal respectively of a complete binary tree. Which of the following is always true?

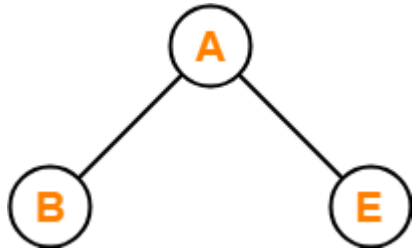
A) LASTIN = LASTPOST

B) LASTIN = LASTPRE

C) LASTPRE = LASTPOST

D) None of these

- Consider the following complete binary tree-



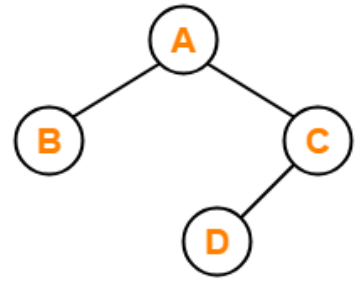
- Preorder Traversal : A , B, E
- Inorder Traversal : B , A , E
- Postorder Traversal : B , E , A
-
- Clearly, LASTIN = LASTPRE. Thus, Option (B) is correct.

PRACTICE PROBLEMS BASED ON TREE TRAVERSAL

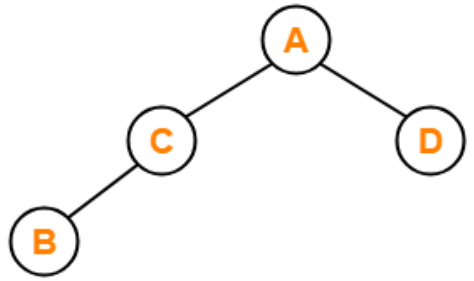
- Which of the following binary trees has its inorder and preorder traversals as BCAD and ABCD respectively

- Solution D

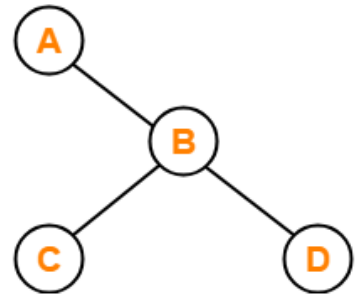
A)



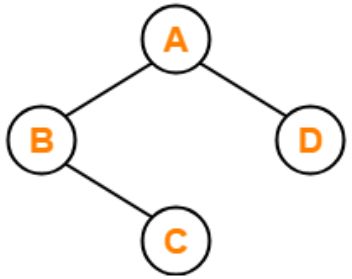
B)



C)

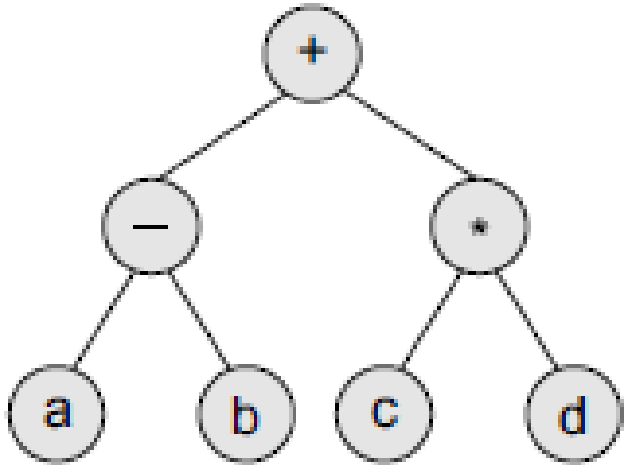


D)



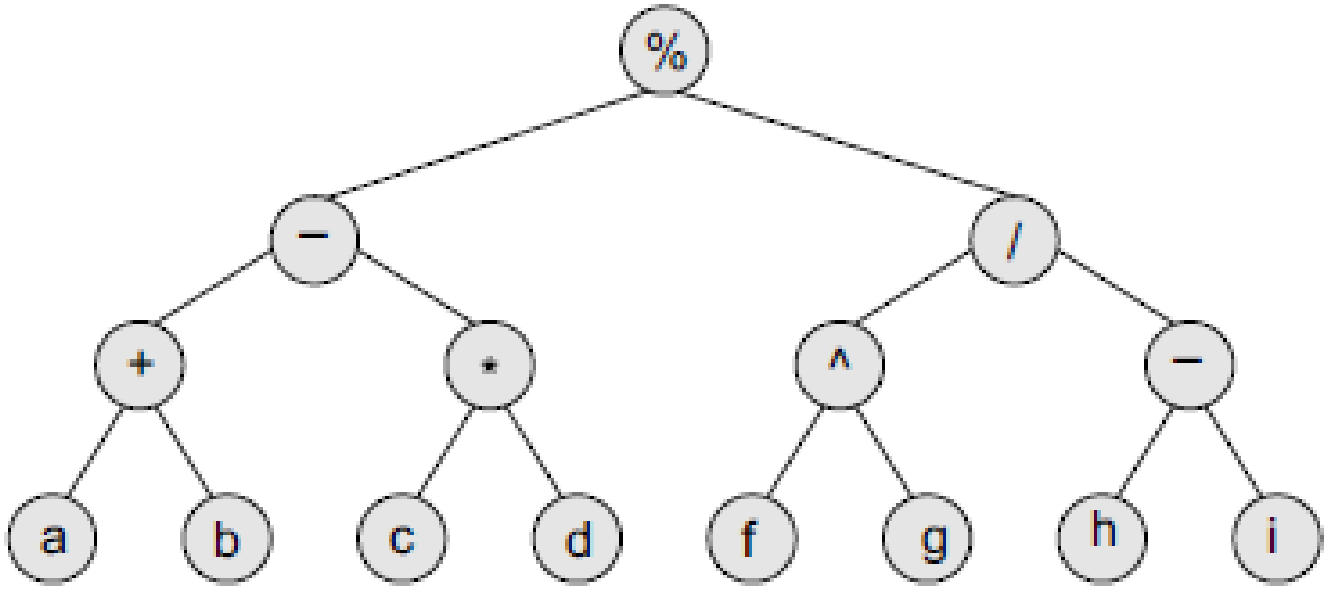
Binary Tree

- **Expression Trees**
- Binary trees are widely used to store algebraic expressions.
- For example, consider the algebraic expression given as:
- Exp = (a – b) + (c * d)
- This expression can be represented using a binary tree as shown in Fig



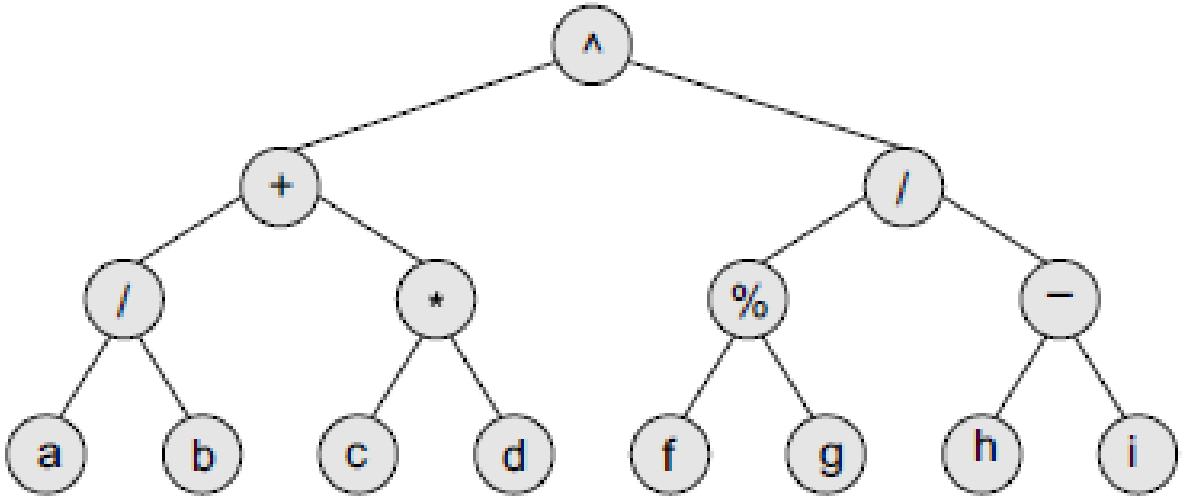
Binary Tree

- Given an expression, Exp = ((a + b) – (c * d)) % ((e ^f) / (g – h)), construct the corresponding binary tree.



Binary Tree

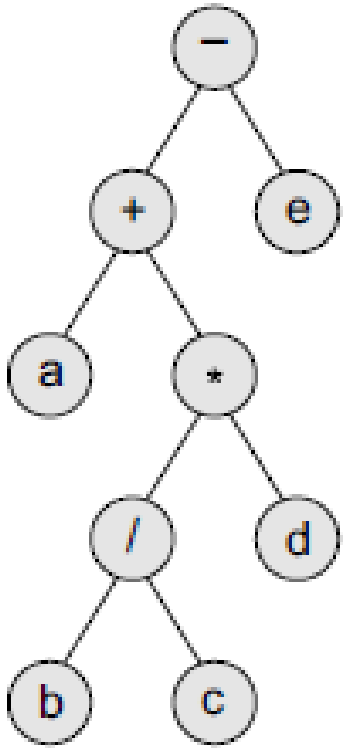
- Given the binary tree, write down the expression that it represents.



Expression for the above binary tree is
[{(a/b) + (c*d)} ^ {(f % g)/(h – i)}]

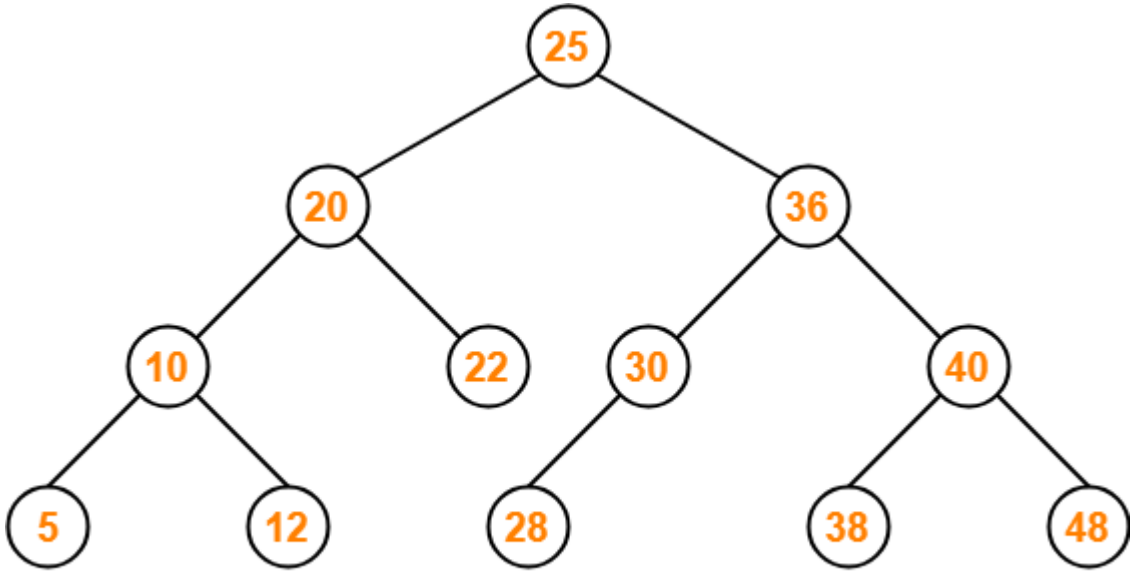
Binary Tree

- Given the expression, Exp = a + b / c * d – e, construct the corresponding binary tree.
- Use the operator precedence to find the sequence in which operations will be performed. The given expression can be written as
- Exp = ((a + ((b/c) * d)) – e)



Binary Search Tree(BST)

- Binary Search Tree is a special kind of binary tree in which nodes are arranged in a specific order.
- In a binary search tree (BST), each node contains-
- Only smaller values in its left sub tree
- Only larger values in its right sub tree



Binary Search Tree

Number of Binary Search Trees

Number of distinct Binary Search Trees possible with n distinct keys = $\frac{^{2n}C_n}{n+1}$

- No. of distinct binary search trees possible with 3 distinct keys

= $^{2\times3}C_3$ / 3+1

= $^{6}C_3$ / 4

= 5

$^nC_r = \frac{n!}{r!(n-r)!}$

Given $^6C_3$

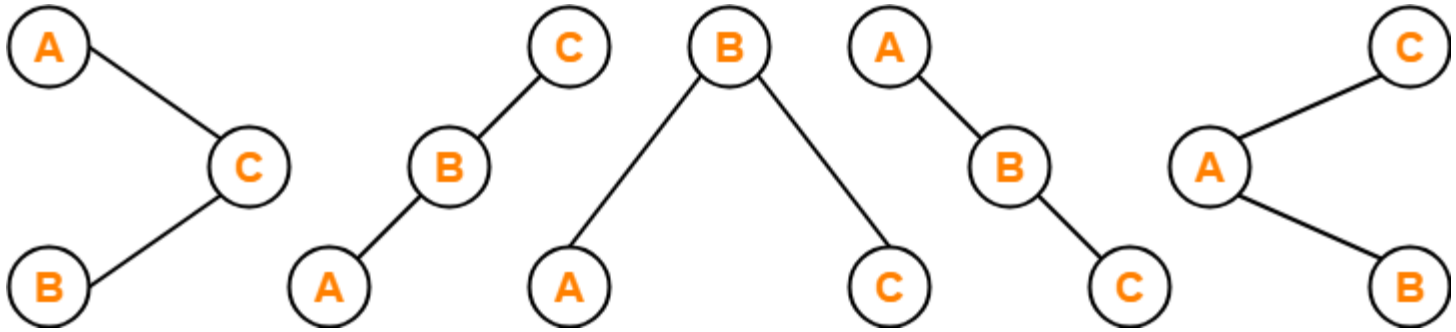
Where $n = 6,\ r = 3$

$$\therefore {}^nC_r = \frac{n!}{r!(n-r)!}$$
$$= \frac{6!}{3!(6-3)!}$$
$$= \frac{6!}{3!(3)!}$$
$$= \frac{6\times5\times4\times3!}{3!\times3\times2\times1}$$
$$= \frac{6\times5\times4}{3\times2\times1}$$
$$= \frac{120}{6}$$
$$= 20$$

Hence, $^6C_3$ is 20

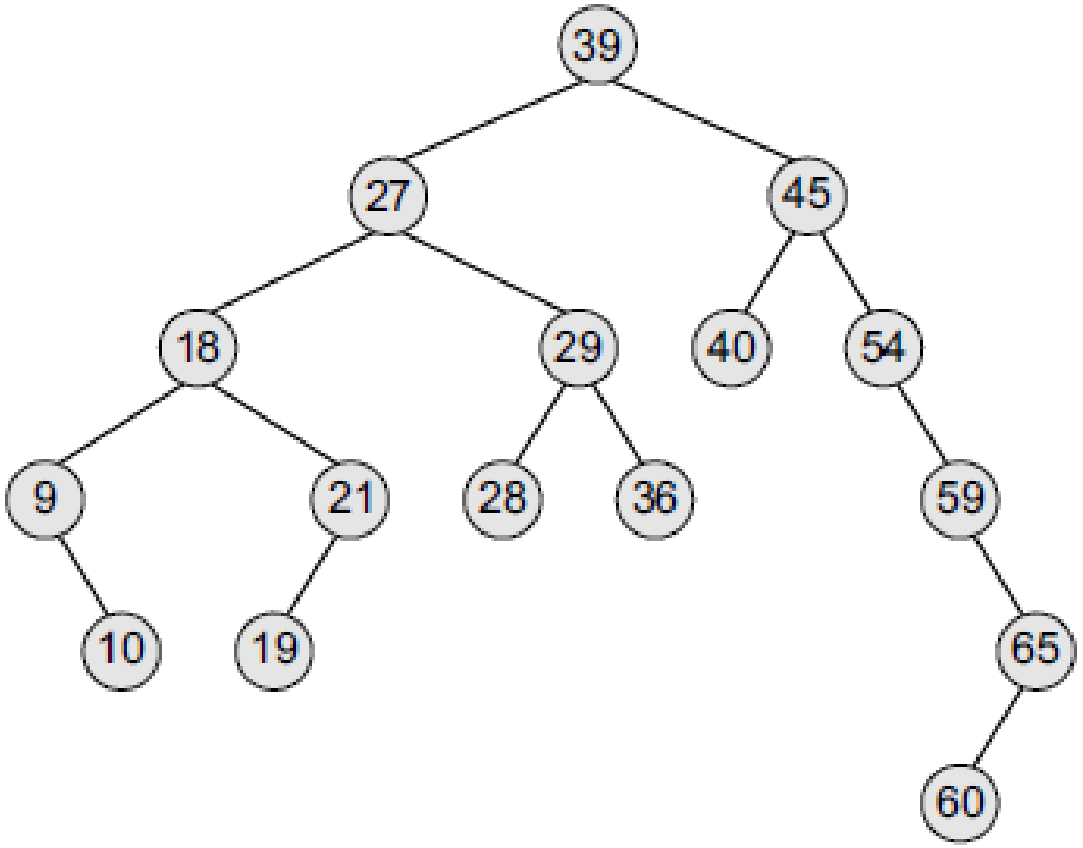
Number of Binary Search Trees

- If three distinct keys are A, B and C, then 5 distinct binary search trees are-



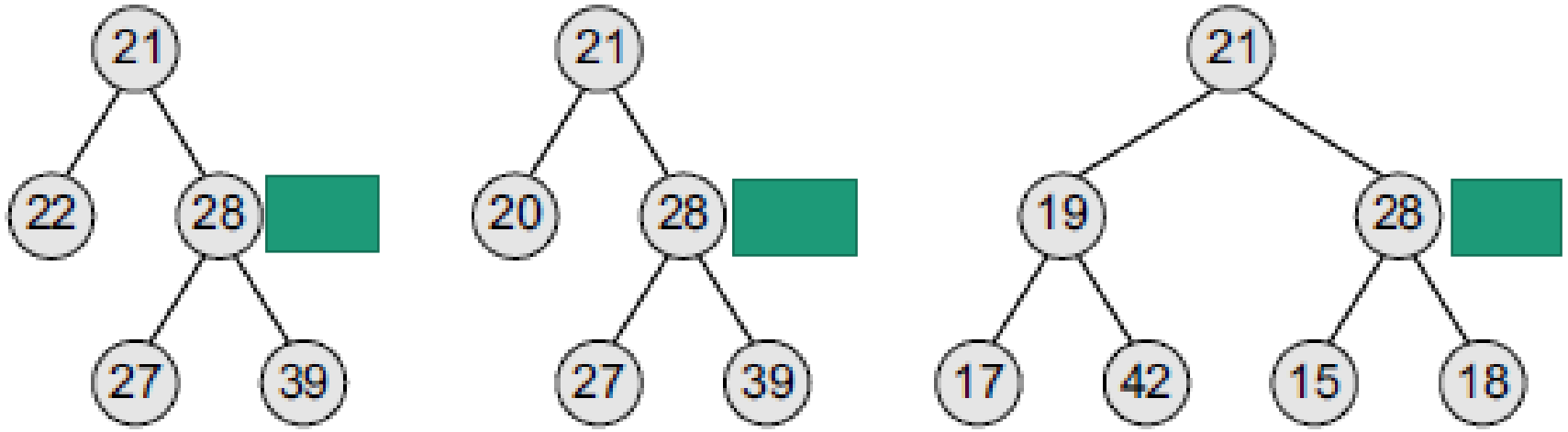
Binary Search Tree(BST)

- For example, in the given tree, if we have to search for 29.
- Then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value).
- The left sub-tree has a root node with the value 27. Since 29 is greater than 27, we will move to the right sub-tree, where we will find the element.



Binary Search Tree(BST)

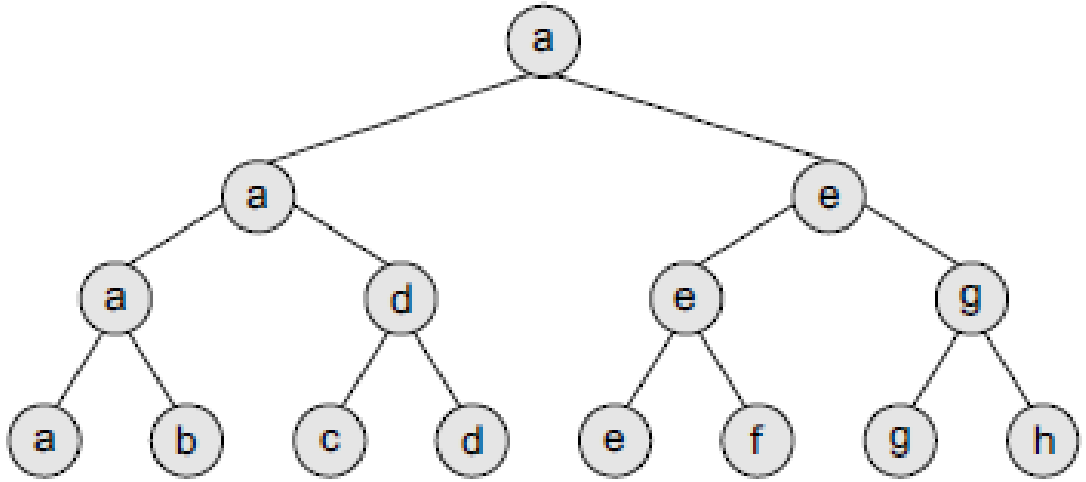
State whether the binary trees in Fig. are binary search trees or not.



Tournament Trees

- Chess tournament
- n number of players participate
- To declare the winner among all these players, a couple of matches are played and usually three rounds are played in the game.
- For ex. 8 players participate, then 4 matches will be played in round 1.
- Then in round 2, the winners of round 1 will play against each other.
- Similarly, in round 3, the winners of round 2 will play against each other
- The person who wins round 3 is declared the winner.
- Tournament trees are used to represent this concept.

Tournament Trees



Tournament trees are also called *winner trees*(a *selection tree*)

Constructing a Binary Tree from Traversal Results

- We can construct a binary tree if we are given at least two traversal results.
- The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal.
- In-order traversal result can used to determine the left and the right child nodes, & the pre-order/post-order can be used to determine the root node.

Steps to Construct a Tree from Traversal Results

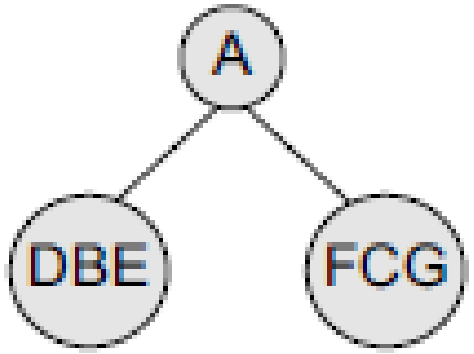
- ***Step 1*** Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.
- ***Step 2*** Elements on the left side of the root node in the in-order traversal sequence form the left sub-tree of the root node. Similarly, elements on the right side of the 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.

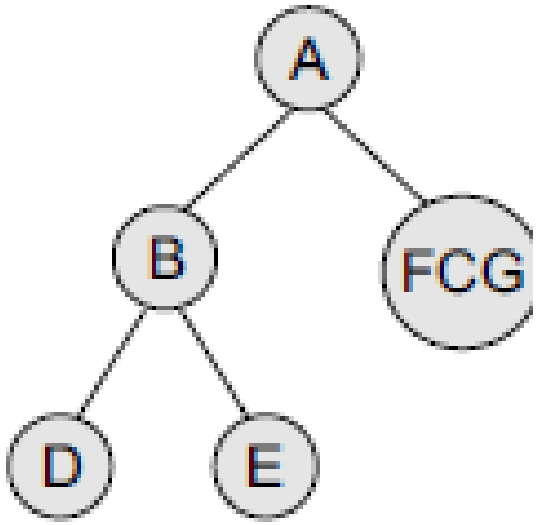
Constructing a Binary Tree from Traversal Results

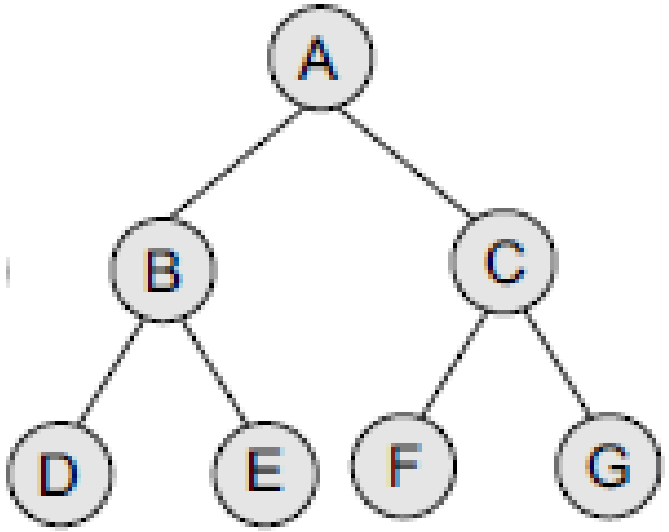
Consider the traversal results given below:

In–order Traversal: D B E A F C G

Pre–order Traversal: A B D E C F G





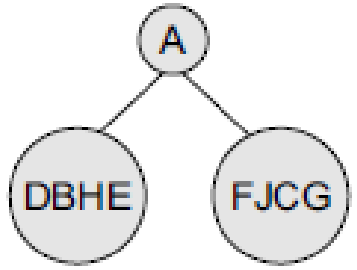


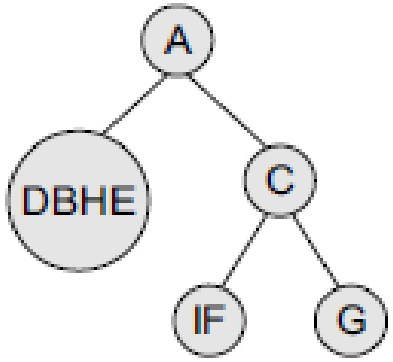
Constructing a Binary Tree from Traversal Results

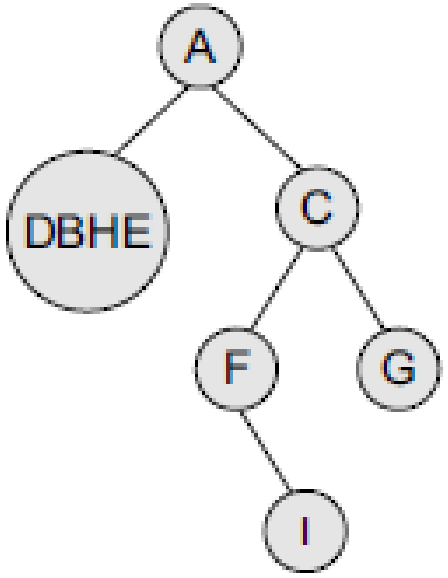
- Consider the traversal results given below:

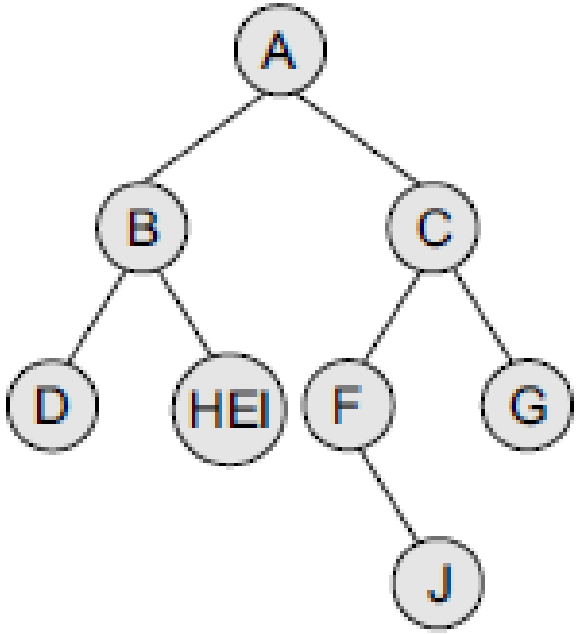
  In–order Traversal: D B H E I A F J C G

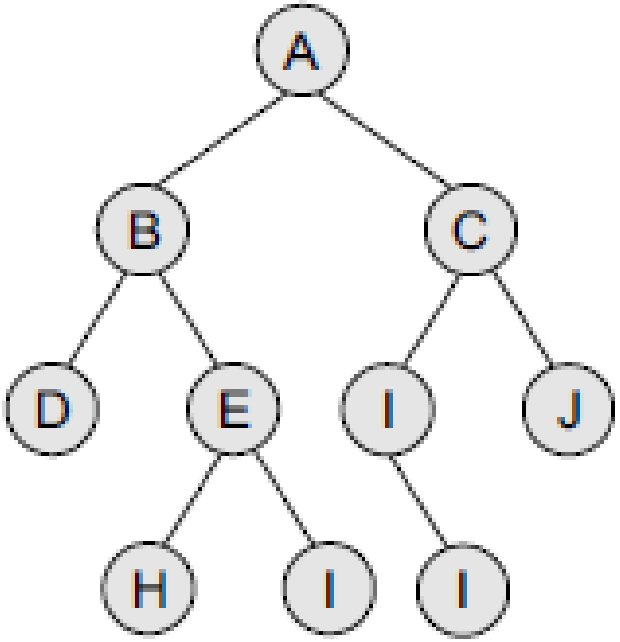
  Post order Traversal: D H I E B J F G C A











SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Tree Applications

- Trees are used to store simple as well as complex data. Here simple means an integer value, character value and complex data means a structure or a record.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system use.
- Another variation of tree, B-trees are prominently used to store tree structures on disc. They are used to index a large number of records.
- B-trees are also used for secondary indexes in databases, where the index facilitates a select operation to answer some range criteria.
- Trees are an important data structure 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.

Tree 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 a 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.

Applications

- **Spanning trees:** It is the shortest path tree used in the routers to direct the packets to the destination.
- **Binary Search Tree:** It is a type of tree data structure that helps in maintaining a sorted stream of data.
- Full Binary tree
- Complete Binary tree
- Skewed Binary tree
- Strictly Binary tree
- Extended Binary tree
- **Storing hierarchical data:** Tree data structures are used to store the hierarchical data, which means data is arranged in the form of order.
- **Syntax tree:** The syntax tree represents the structure of the program's source code, which is used in compilers.
- **Trie:** It is a fast and efficient way for dynamic spell checking. It is also used for locating specific keys from within a set.
- **Heap:** It is also a tree data structure that can be represented in a form of an array. It is used to implement priority queues.

Tree Applications

- Directory structure of a file storage
- Structure of an arithmetic expressions
- Used in almost every 3D video game to determine what objects need to be rendered.
- Used in almost every high-bandwidth router for storing router-tables.
- used in compression algorithms, such as those used by the .jpeg and .mp3 file formats
- Game trees

Directory structure of a file storage

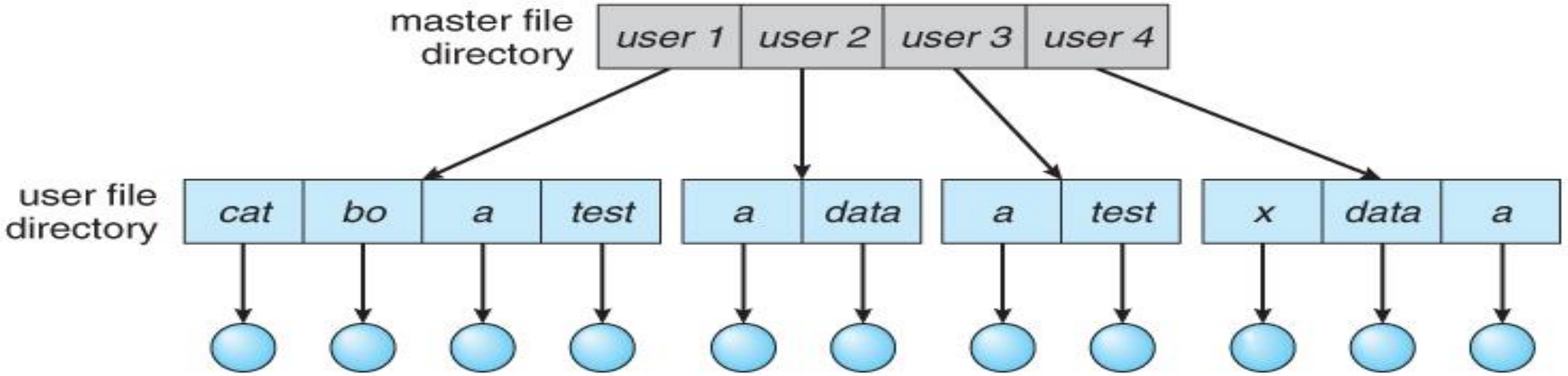


Image courtesy:
https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/images/Chapter11/11_10_TwoLevelStructure.jpg

Structure of an arithmetic expressions

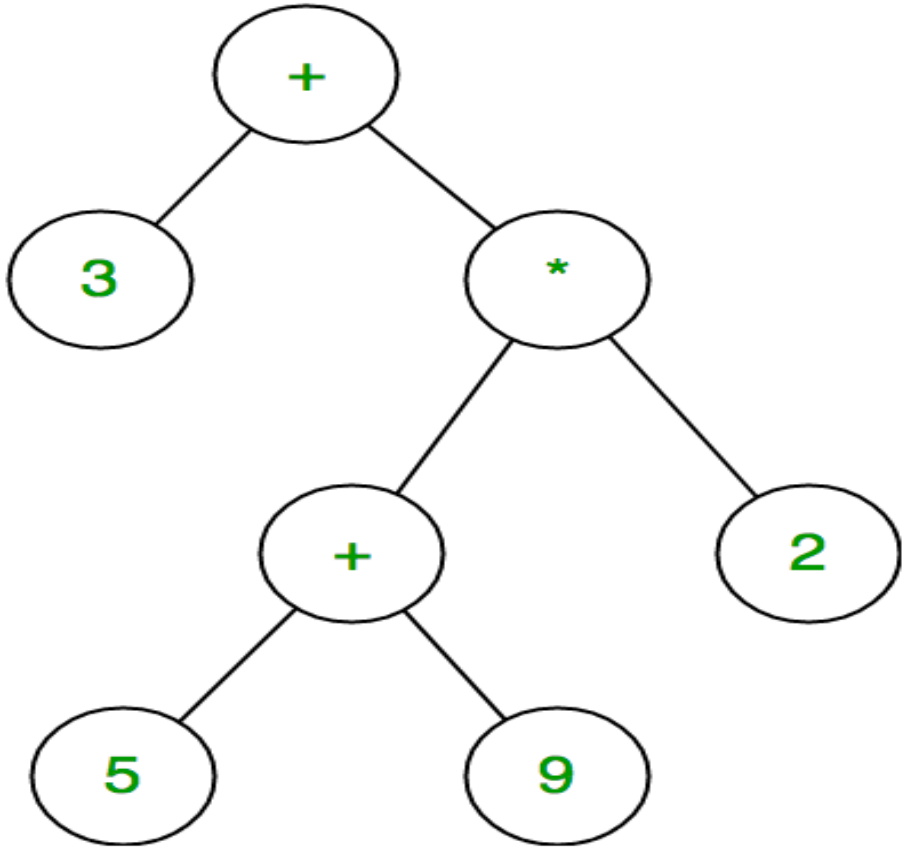


Image courtesy: https://media.geeksforgeeks.org/wp-content/uploads/expression-tree.png

Object rendering in 3-D game

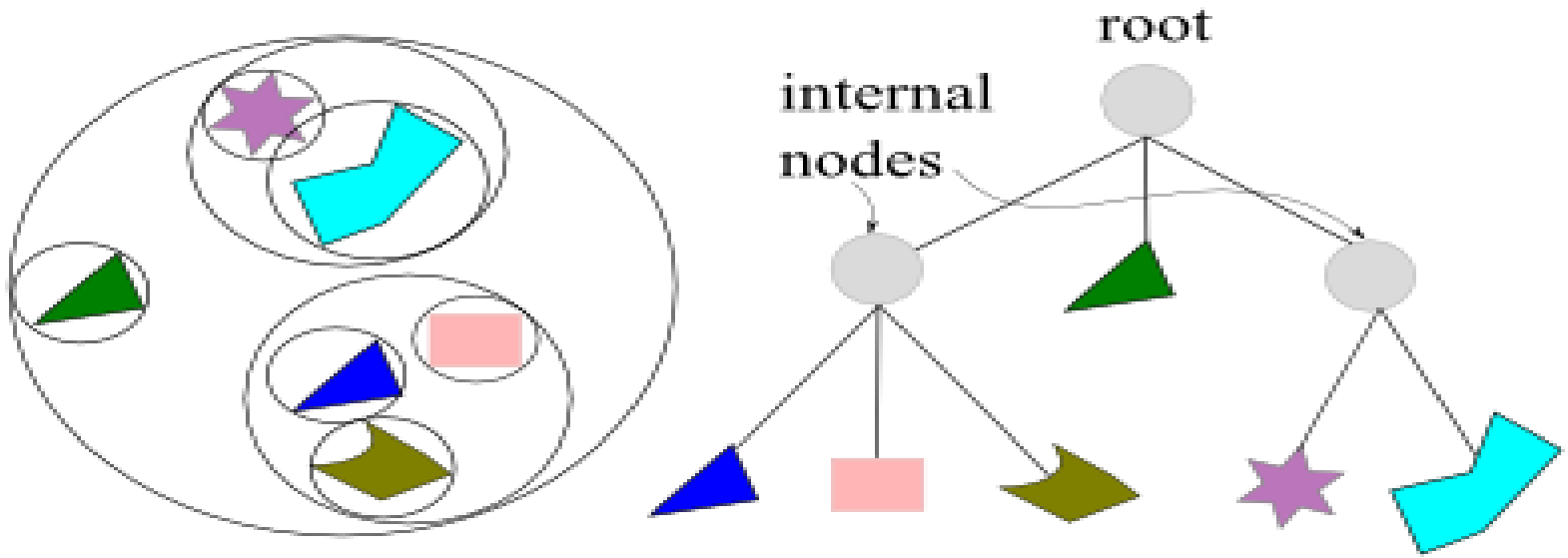


Image Courtesy: https://www.bogotobogo.com/Games/images/BVH.png

DNS Server entries

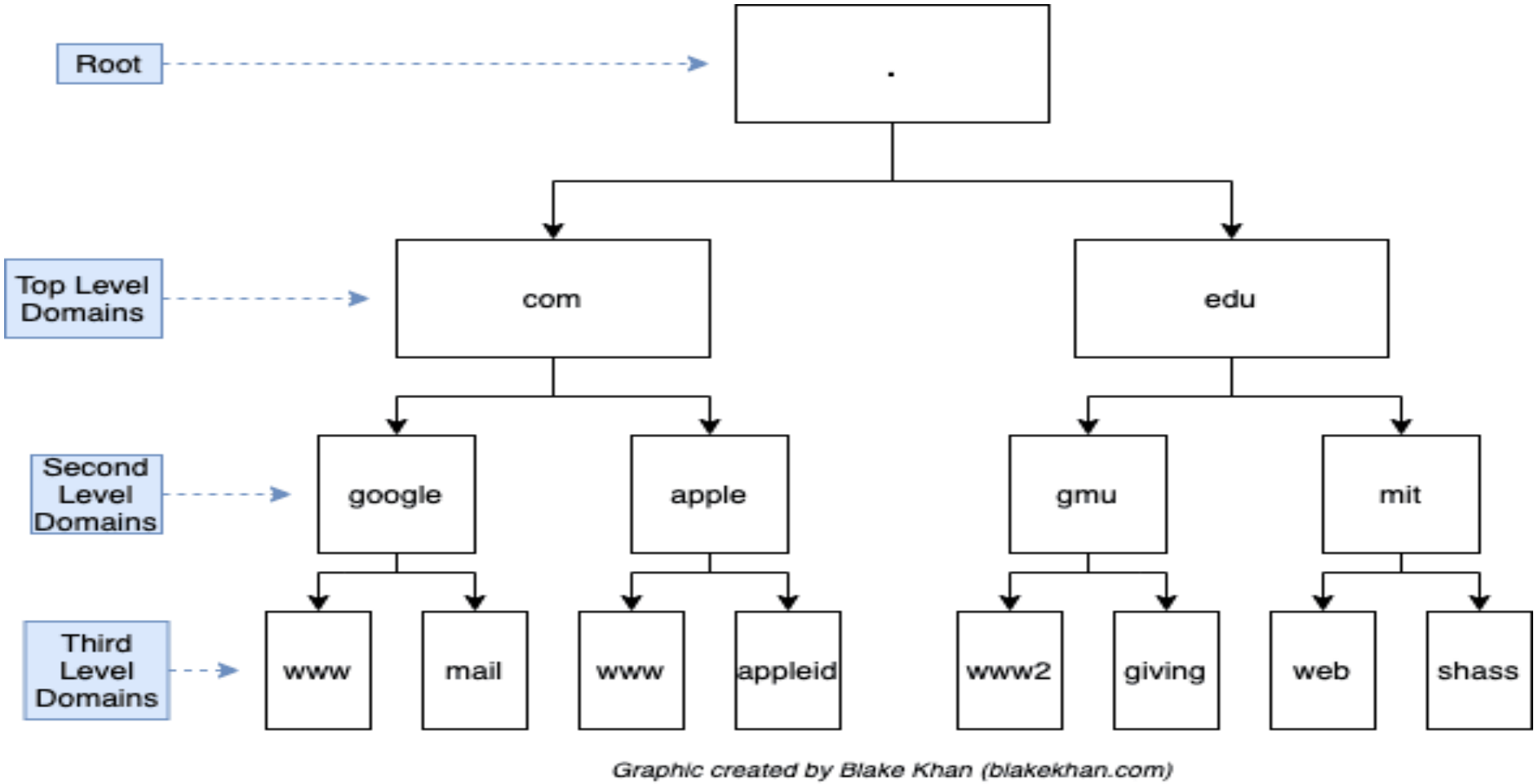
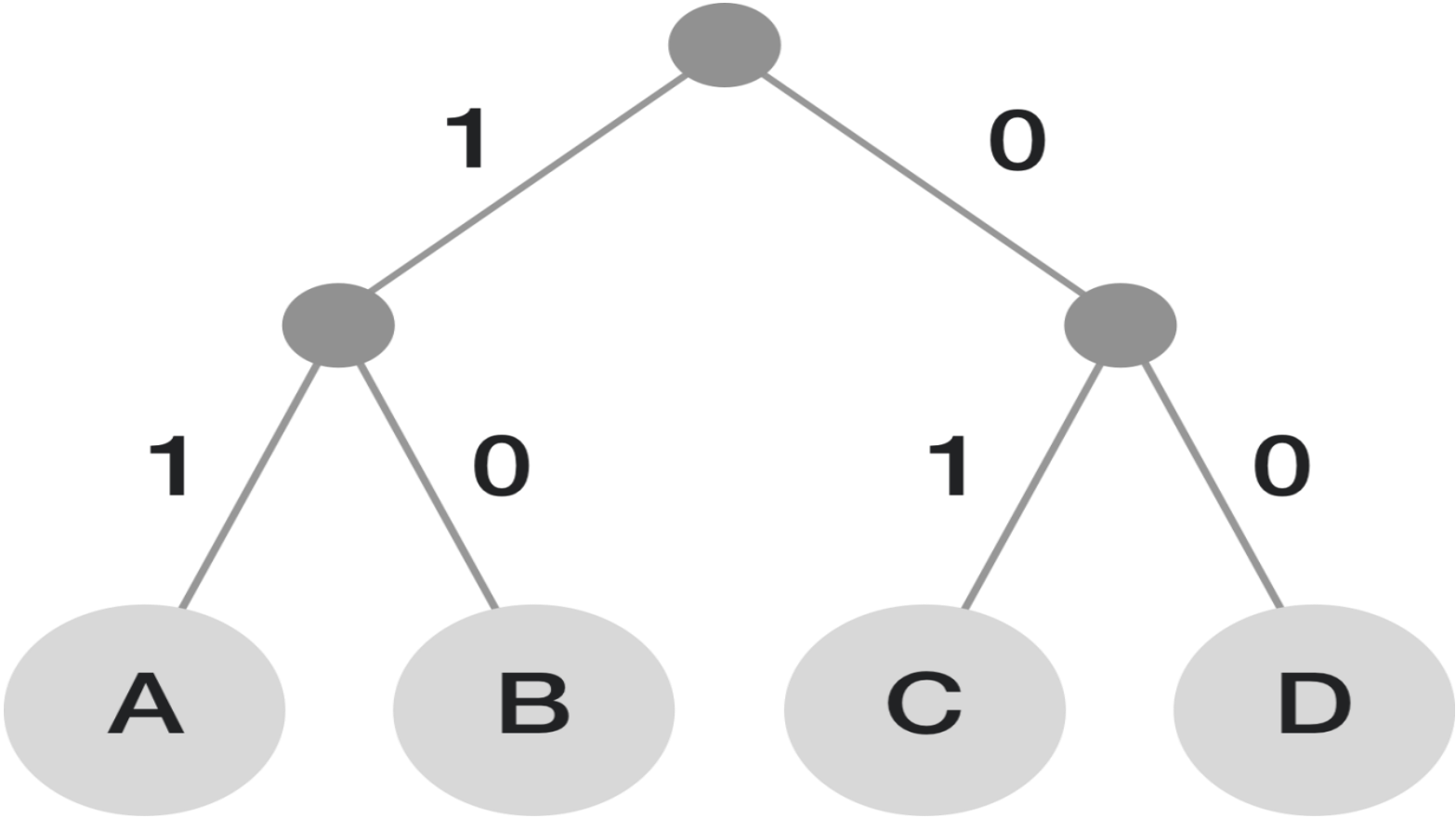


Image Courtesy: https://res.cloudinary.com/practicaldev/image/fetch/s--b9G6DenD--/c_limit%2Cf_auto%2Cfl_progressive%2Cq_auto%2Cw_880/https://i.imgur.com/xOdVIPZ.png

Compression Algorithm



https://brilliant-staff-media.s3-us-west-2.amazonaws.com/tiffany-wang/VEIWKBhSSc.png

Game Tree

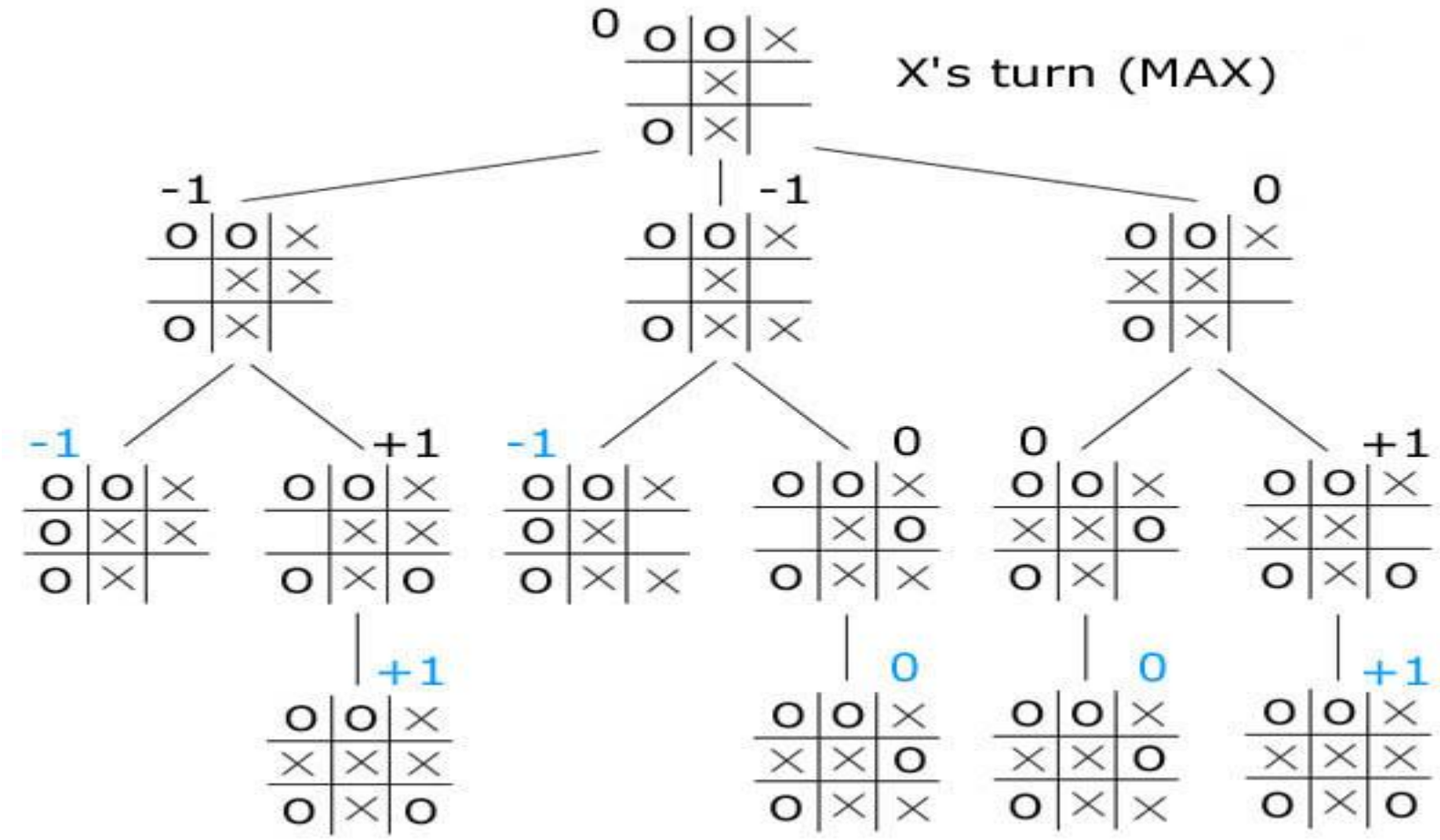
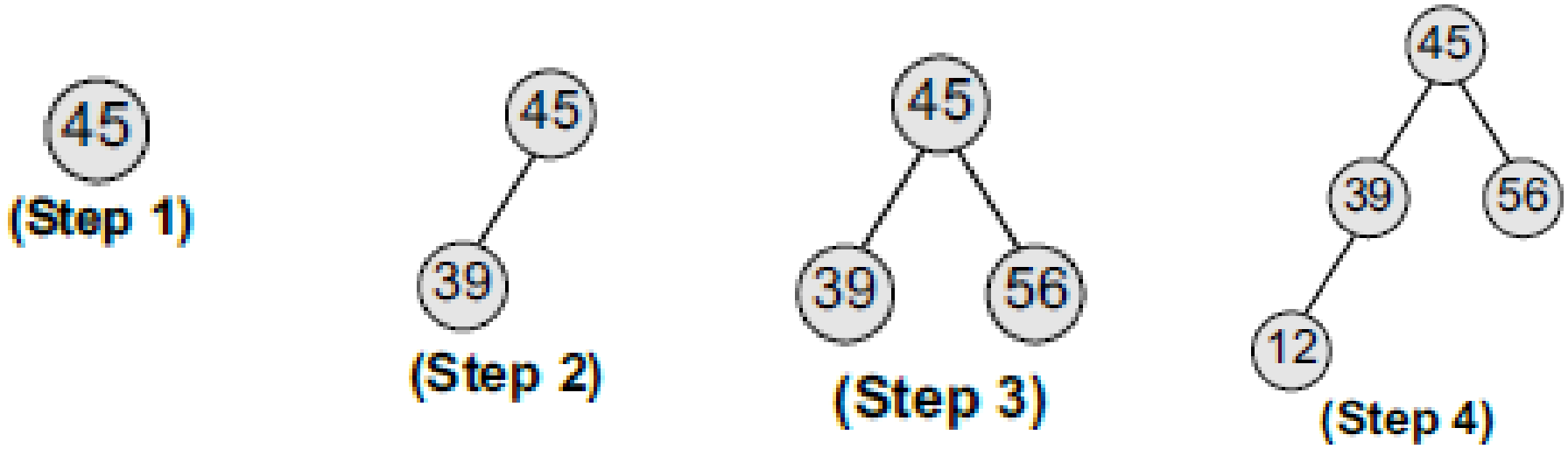


Image Courtesy: https://www.ocf.berkeley.edu/~yosenl/extras/alphabeta/alphabeta.jpg

Binary Search Tree

- Create a binary search tree using the following data elements:

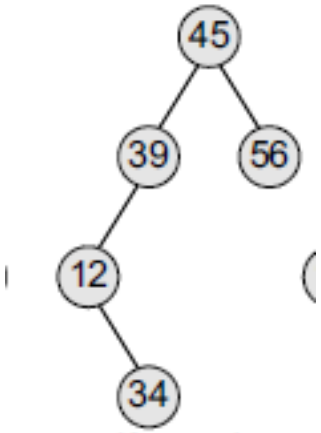
  45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



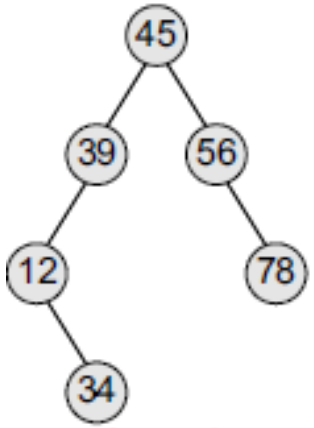
Binary Search Tree

- Create a binary search tree using the following data elements:

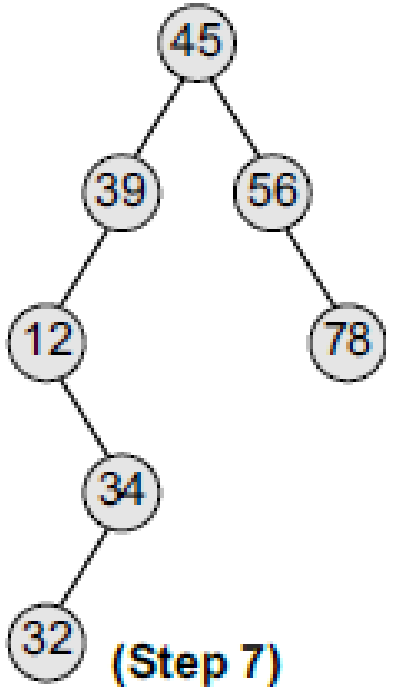
  45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



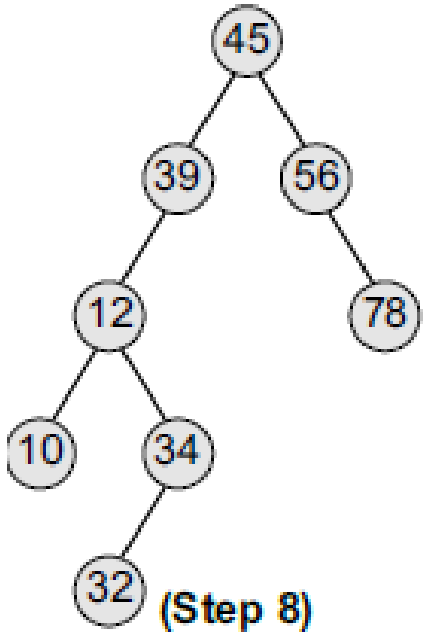
(Step 5)



(Step 6)



(Step 7)

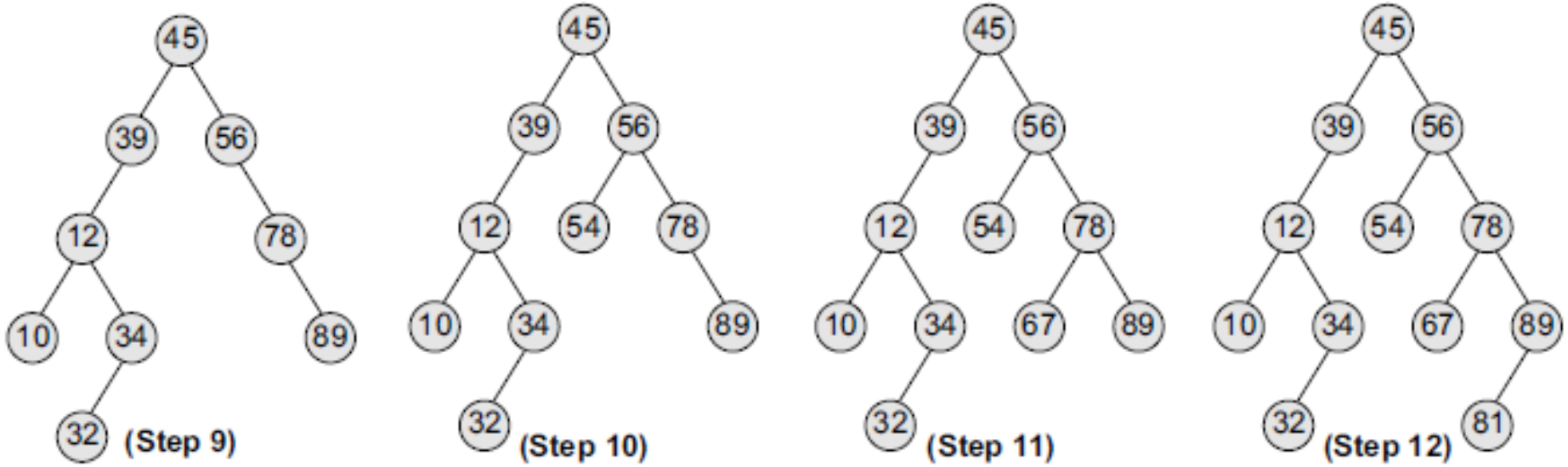


(Step 8)

Binary Search Tree

- Create a binary search tree using the following data elements:

  45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81



Binary Search implementation

```
Struct tree{
        int data;
        struct tree *left;
        struct tree *right;
        }
Struct tree *t;
```

Operations on BST : Searching for a Node

- The search function is used to find whether a given value is present in the tree or not.
- The searching process begins at the root node.
- The function first checks if the binary search tree is empty. If it is empty, then the value we are searching for is not present in the tree.
- So, the search algorithm terminates by displaying an appropriate message.
- However, if there are nodes in the tree, then the search function checks to see if the key value of the current node is equal to the value to be searched.
- If not, it checks if the value to be searched for is less than the value of the current node, in which case it should be recursively called on the left child node.
- In case the value is greater than the value of the current node, it should be recursively called on the right child node.

Operations on Binary Search Trees

- **Searching for a Node in a Binary Search Tree**

```
searchElement (TREE, VAL)

Step 1: IF TREE -> DATA = VAL OR TREE = NULL
          Return TREE
        ELSE
         IF VAL < TREE -> DATA
          Return searchElement(TREE -> LEFT, VAL)
         ELSE
          Return searchElement(TREE -> RIGHT, VAL)
         [END OF IF]
        [END OF IF]
Step 2: END
```

Operations on Binary Search Trees

- **Inserting a New Node in a Binary Search Tree**
- The insert function is used to add a new node with a given value at the correct position in the binary search tree.
- Adding the node at the correct position means that the new node should not violate the properties of the binary search tree.
- The initial code for the insert function is similar to the search function. This is because we first find the correct position where the insertion has to be done and then add the node at that position.
- The insertion function changes the structure of the tree. Therefore, when the insert function is called recursively, the function should return the new tree pointer.

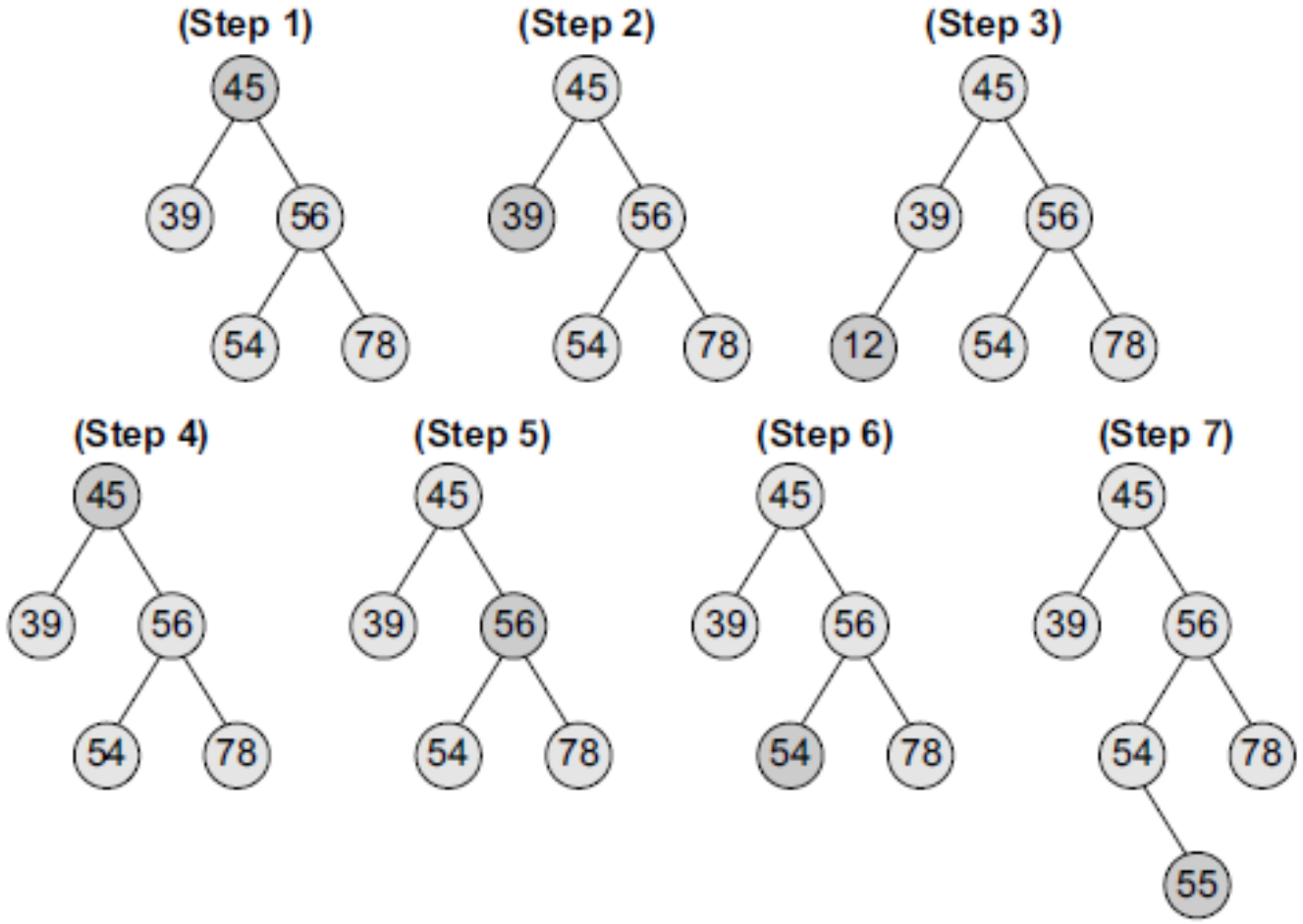
Operations on Binary Search Trees

- **Inserting a New Node in a Binary Search Tree**

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
            Allocate memory for TREE
            SET TREE -> DATA = VAL
            SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
            IF VAL < TREE -> DATA
                  Insert(TREE -> LEFT, VAL)
            ELSE
                  Insert(TREE -> RIGHT, VAL)
            [END OF IF]
        [END OF IF]
Step 2: END
```

Operations on Binary Search Trees

- We will take up the case of inserting 12 and 55.

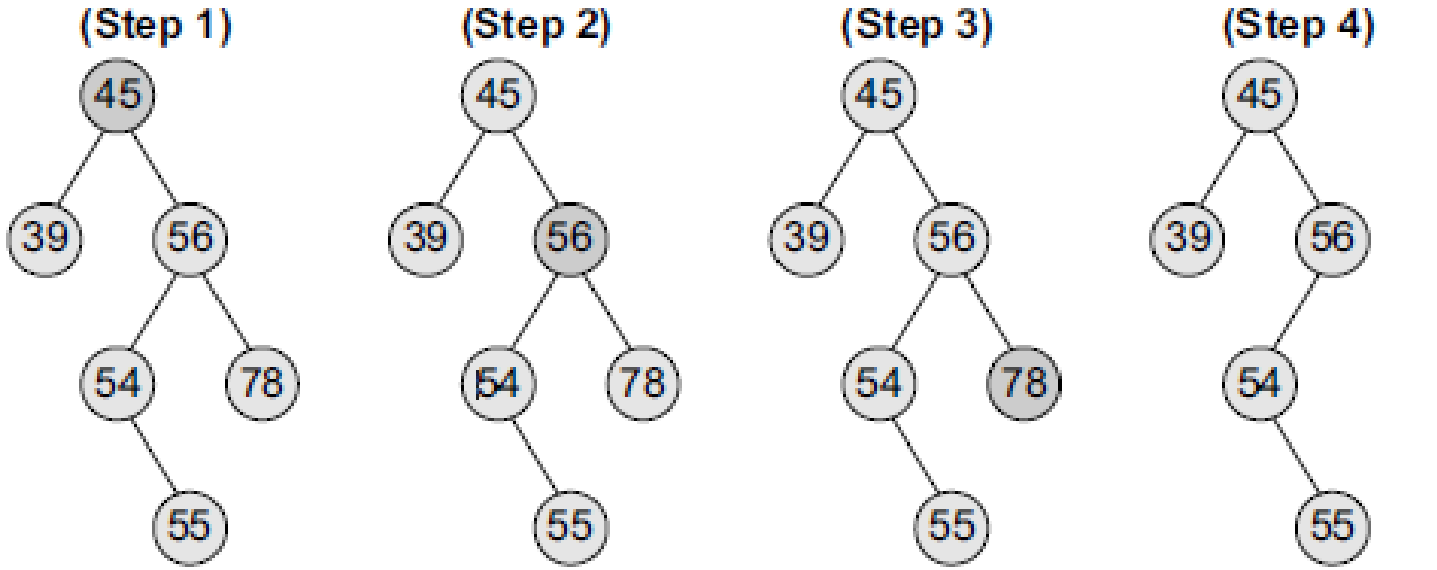


Operations on Binary Search Trees

- **Deleting a Node from a Binary Search Tree**
- The delete function deletes a node from the binary search tree. However, utmost care should be taken that the properties of the binary search tree are not violated and nodes are not lost in the process.
- We will take up three cases and discuss how a node is deleted from a binary search tree.
- ***Case 1: Deleting a Node that has No Children***
- ***Case 2: Deleting a Node with One Child***
- ***Case 3: Deleting a Node with Two Children***

Operations on Binary Search Trees

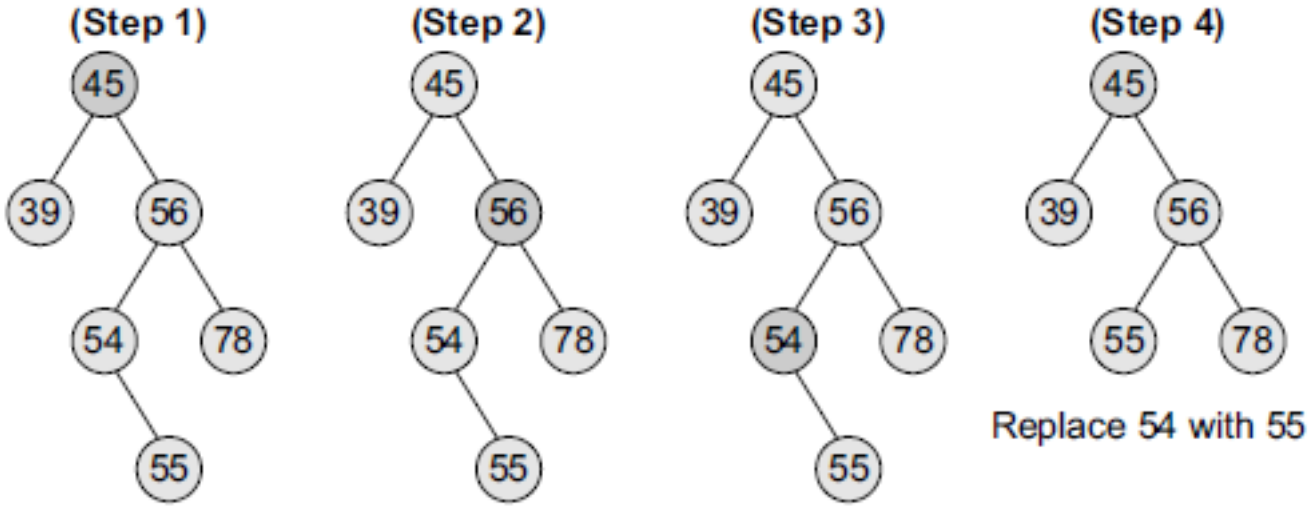
- ***Case 1: Deleting a Node that has No Children***
- Look at the binary search tree given. If we have to delete node 78, we can simply remove this node without any issue. This is the simplest case of deletion.



Delete node 78

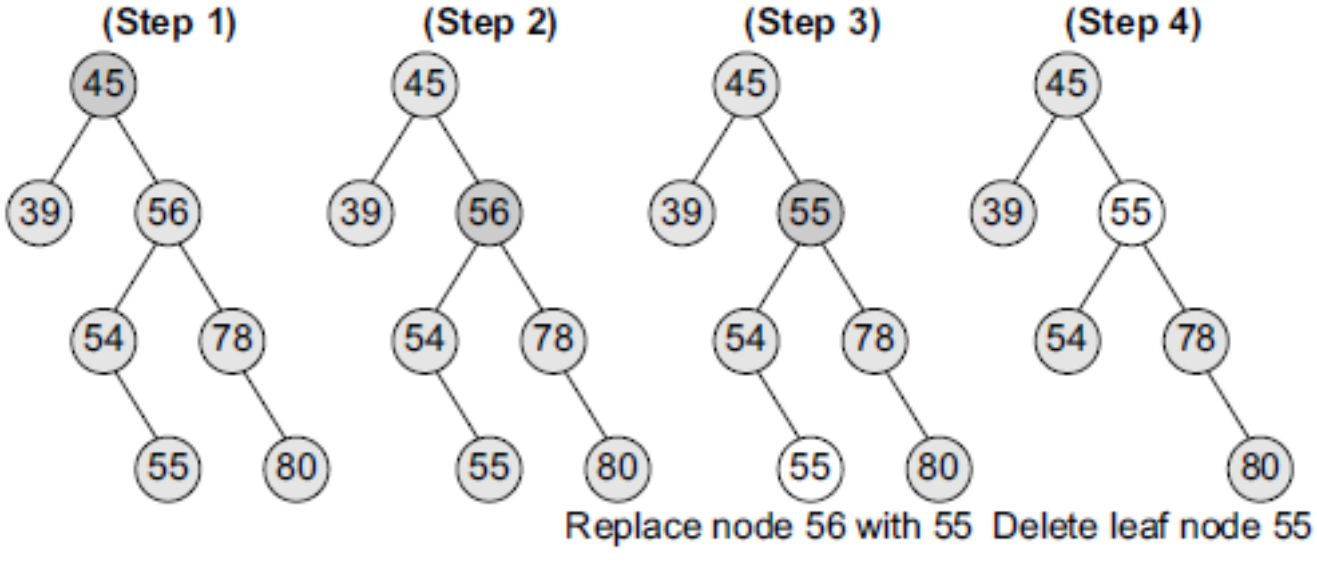
Operations on Binary Search Trees

- ***Case 2: Deleting a Node with One Child***
- The node's child is set as the child of the node's parent. In other words, replace the node with its child. Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent. Correspondingly, if the node is the right child of its parent, the node's child becomes the right child of the node's parent.



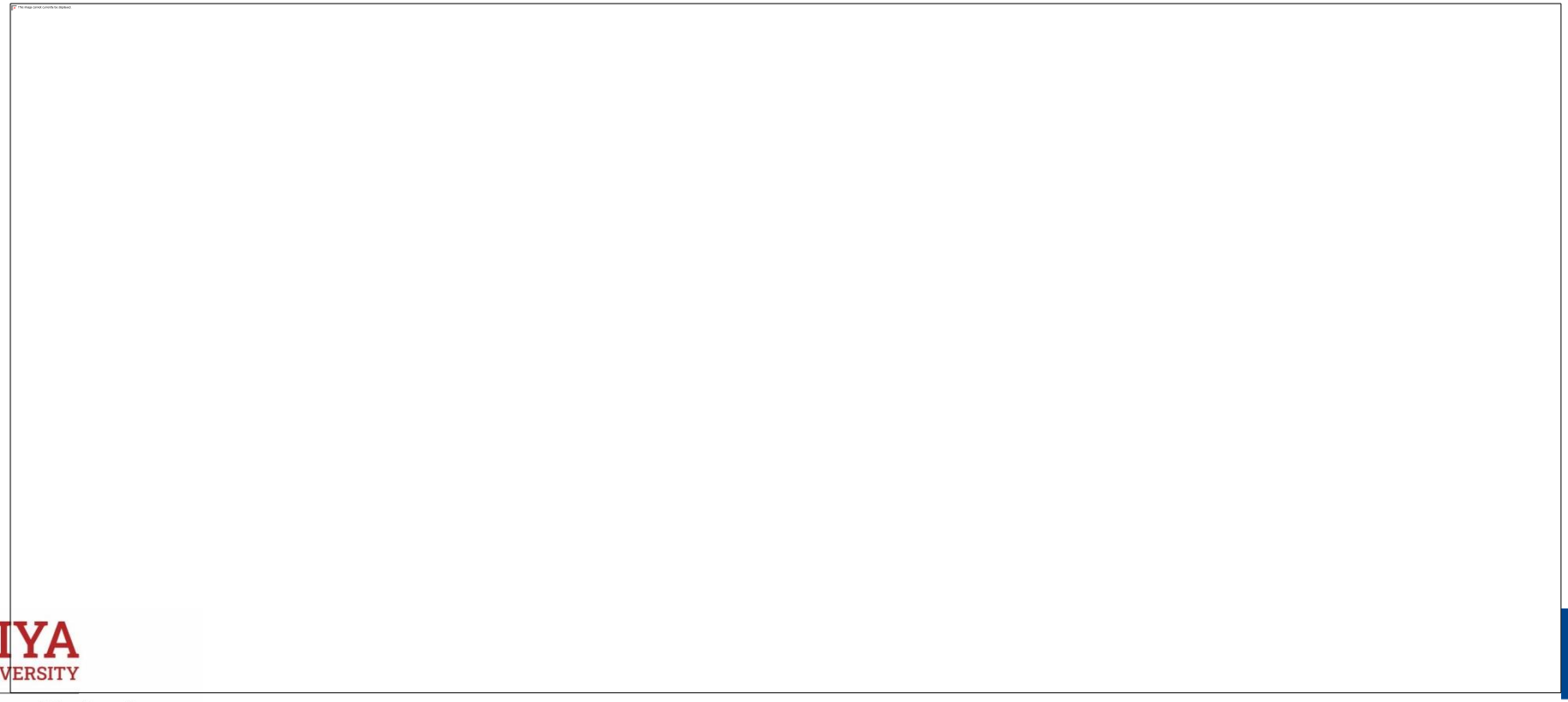
Operations on Binary Search Trees

- ***Case 3: Deleting a Node with Two Children***
- Replace the node's value with its *in-order predecessor* (largest value in the left sub-tree) or *in-order successor* (smallest value in the right sub-tree). The in-order predecessor or the successor can then be deleted using any of the above cases. Let us see how deletion of node with value 56 is handled.



Operations on Binary Search Trees

- ***Case 3: Deleting a Node with Two Children***
- This deletion could also be handled by replacing node 56 with its in-order successor



Operations on Binary Search Trees

- In Step 1 of the algorithm, we first check if TREE=NULL, because if it is true, then the node to be deleted is not present in the tree.
- However, if that is not the case, then we check if the value to be deleted is less than the current node's data. In case the value is less, we call the algorithm recursively on the node's left sub-tree, otherwise the algorithm is called recursively on the node's right sub-tree.
- Note that if we have found the node whose value is equal to VAL, then we check which case of deletion it is.

Operations on Binary Search Trees

- If the node to be deleted does not have any child, then we simply set the node to NULL.
- If the node to be deleted has either a left or a right child but not both, then the current node is replaced by its child node and the initial child node is deleted from the tree.
- If the node to be deleted has both left and right children, then we find the in-order predecessor of the node or in-order successor .

Operations on Binary Search Trees

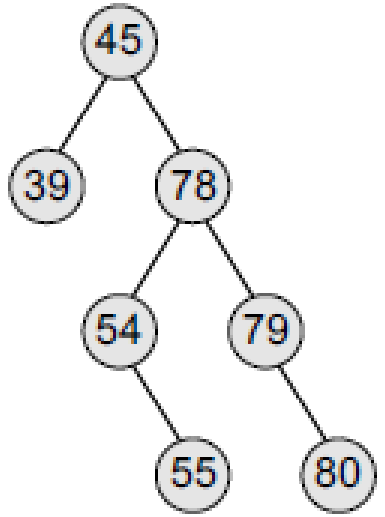
- If the node to be deleted has both left and right children, then we find the in-order predecessor of the node by calling
- findLargestNode(TREE ->LEFT) and
- replace the current node's value with that of its in-order predecessor.
- Then, we call Delete(TREE -> LEFT, TEMP -> DATA) to delete the initial node of the in-order predecessor.
- Thus, we reduce the case 3 of deletion into either case 1 or case 2 of deletion.

Operations on Binary Search Trees

```
Delete (TREE, VAL)

Step 1: IF TREE = NULL
          Write "VAL not found in the tree"
        ELSE IF VAL < TREE -> DATA
          Delete(TREE->LEFT, VAL)
        ELSE IF VAL > TREE -> DATA
          Delete(TREE -> RIGHT, VAL)
        ELSE IF TREE -> LEFT AND TREE -> RIGHT
          SET TEMP = findLargestNode(TREE -> LEFT)
          SET TREE -> DATA = TEMP -> DATA
          Delete(TREE -> LEFT, TEMP -> DATA)
        ELSE
          SET TEMP = TREE
          IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
              SET TREE = NULL
          ELSE IF TREE -> LEFT != NULL
               SET TREE = TREE -> LEFT
          ELSE
               SET TREE = TREE -> RIGHT
          [END OF IF]
          FREE TEMP
        [END OF IF]
Step 2: END
```

Determining the Number of Nodes

- To calculate the total number of elements/nodes the tree, we count the number of nodes in the left sub-tree and the right sub-tree.
- Number of nodes = totalNodes(left sub–tree)+ totalNodes(right sub–tree) + 1
- The total number of nodes in the tree can be calculated as:
- Total nodes of left sub–tree = 1
- Total nodes of left sub–tree = 5
- Total nodes of tree = (1 + 5) + 1
- Total nodes of tree = 7

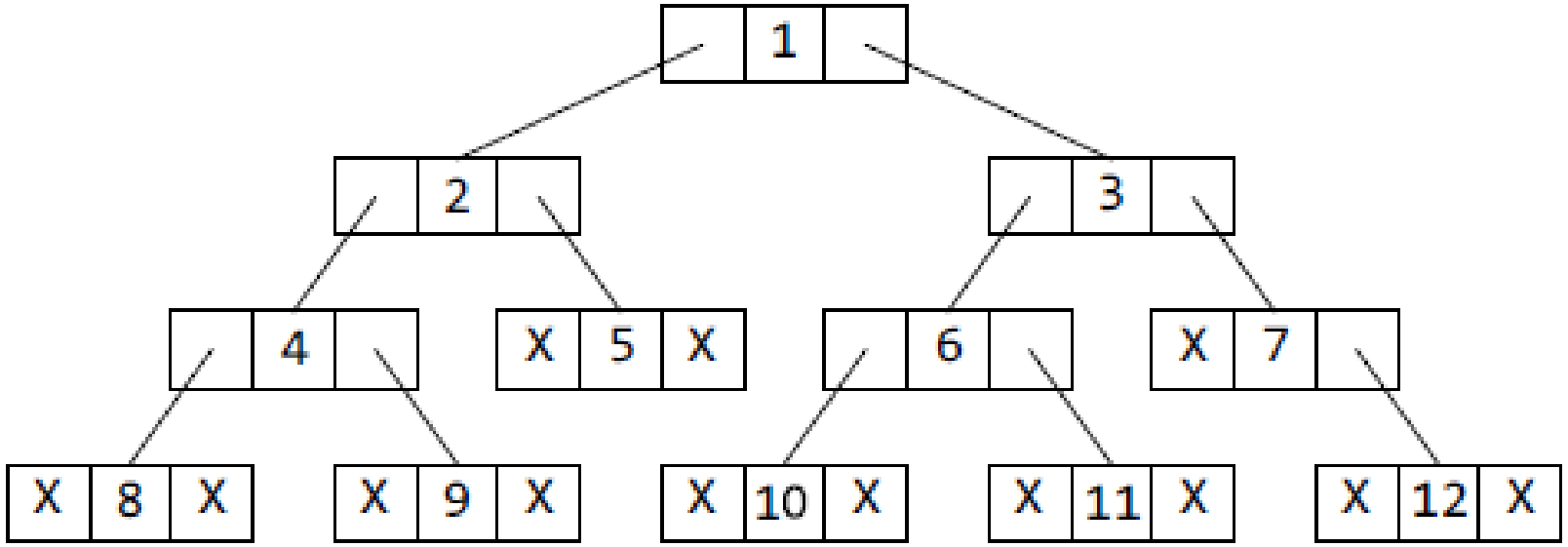


Determining the Number of Nodes

```
totalNodes(TREE)

Step 1: IF TREE = NULL
            Return 0
        ELSE
            Return totalNodes(TREE -> LEFT)
                + totalNodes(TREE -> RIGHT) + 1
        [END OF IF]
Step 2: END
```

THREADED BINARY TREEs

- A threaded binary tree is the same as that of a binary tree but with a difference in storing the NULL pointers.
- Consider the linked representation of a binary tree as given in Fig.



THREADED BINARY TREEs

- In the linked representation, a number of nodes contain a NULL pointer, either in their left or right fields or in both.
- The space is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information.
- For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node. These special pointers are called *threads* and binary trees containing threads are called *threaded trees*.
- In the linked representation of a threaded binary tree, threads will be denoted using arrows.
- A threaded binary tree may correspond to one-way threading or a two way threading.

THREADED BINARY TREEs

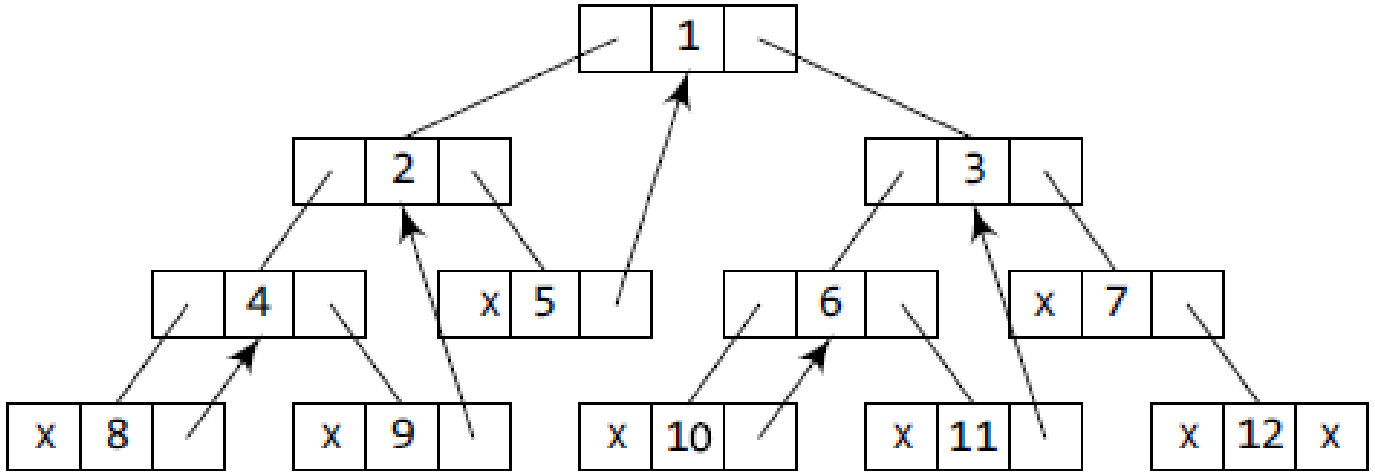
- In one-way threading, a thread will appear either in the right field or the left field of the node. A one-way threaded tree is also called a single-threaded tree.
- If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node. Such a one-way threaded tree is called a left-threaded binary tree.
- On the contrary, if the thread appears in the right field, then it will point to the in-order successor of the node. Such a one-way threaded tree is called a right threaded binary tree.

THREADED BINARY TREEs

- In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node. While the left field will point to the in-order predecessor of the node, the right field will point to its successor. A two-way threaded binary tree is also called a fully threaded binary tree.
- Figure shows a binary tree without threading and its corresponding linked representation.(previous slide)
- *The in-order traversal of the tree is given as 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12*

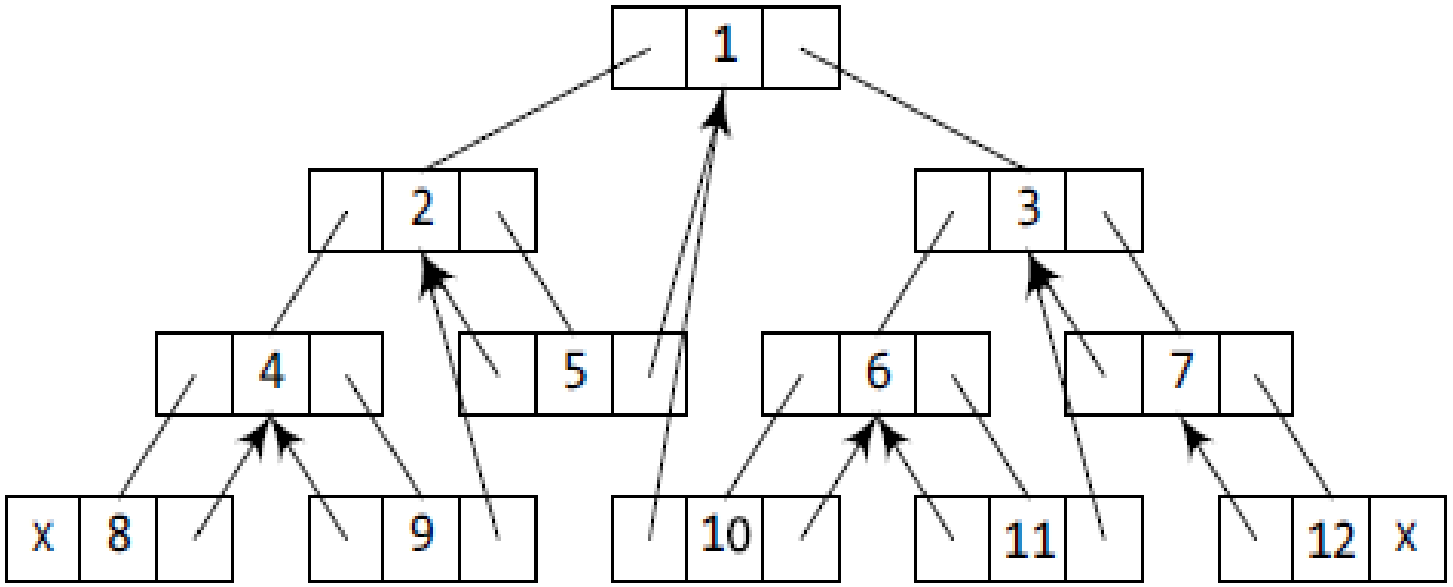
One-way Threading

- Node 5 contains a NULL pointer in its RIGHT field, so it will be replaced to point to node 1, which is its in-order successor. (*8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12*)
- Similarly, the RIGHT field of node 8 will point to node 4, the RIGHT field of node 9 will point to node 2, the RIGHT field of node 10 will point to node 6, the RIGHT field of node 11 will point to node 3, and the RIGHT field of node 12 will contain NULL because it has no in-order successor.

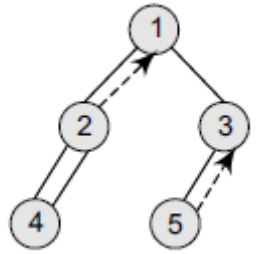


Two-way Threading *(8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12)*

- Node 5 contains a NULL pointer in its LEFT field, so it will be replaced to point to node 2, which is its in-order predecessor. Similarly, the LEFT field of node 8 will contain NULL because it has no in-order predecessor, the LEFT field of node 7 will point to node 3, the LEFT field of node 9 will point to node 4, the LEFT field of node 10 will point to node 1, the LEFT field of node 11 will contain 6, and the LEFT field of node 12 will point to node 7.



Traversing a Threaded Binary Tree



- Node 1 has a left child i.e., 2 which has not been visited. So, add 2 in the list of visited node make it as the current node.
- Node 2 has a left child i.e., 4 which has not been visited. So, add 4 in the list of visited node make it as the current node.
- Node 4 does not have any left or right child, so print 4 and check for its threaded link. It has a threaded link to node 2, so make node 2 the current node.
- Node 2 has a left child which has already been visited. Does not have a right child. Now, print 2 and follow its threaded link to node 1. Make node 1 the current node.
- Node 1 has a left child that has been already visited. So print 1. Node 1 has a right child 3 which has not yet been visited, so make it the current node.
- Node 3 has a left child (node 5) which has not been visited, so make it the current node.
- Node 5 does not have any left or right child. So print 5. However, it does have a threaded link which points to node 3. Make node 3 the current node.
- Node 3 has a left child which has already been visited. So print 3.Now there are no nodes left, so we end here. The sequence of nodes printed is—4 2 1 5 3.

Traversing a Threaded Binary Tree

Step 1: Check if the current node has a left child that has not been visited. If a left child exists that has not been visited, go to Step 2, else go to Step 3.

Step 2: Add the left child in the list of visited nodes. Make it as the current node and then go to Step 6.

Step 3: If the current node has a right child, go to Step 4 else go to Step 5.

Step 4: Make that right child as current node and go to Step 6.

Step 5: Print the node and if there is a threaded node make it the current node.

Step 6: If all the nodes have visited then END else go to Step 1.

Advantages of Threaded Binary Tree

- It enables linear traversal of elements in the tree.
- Linear traversal eliminates the use of stacks which in turn consume a lot of memory space and computer time.
- It enables to find the parent of a given element without explicit use of parent pointers.
- Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.
- Thus, we see the basic difference between a binary tree and a threaded binary tree is that in binary trees a node stores a NULL pointer if it has no child and so there is no way to traverse back.