Data Structures and Algorithm

Moazzam Ali Sahi

Lecture # 19-20

TREE Data Structure

Last Lecture

- Midterm Exam

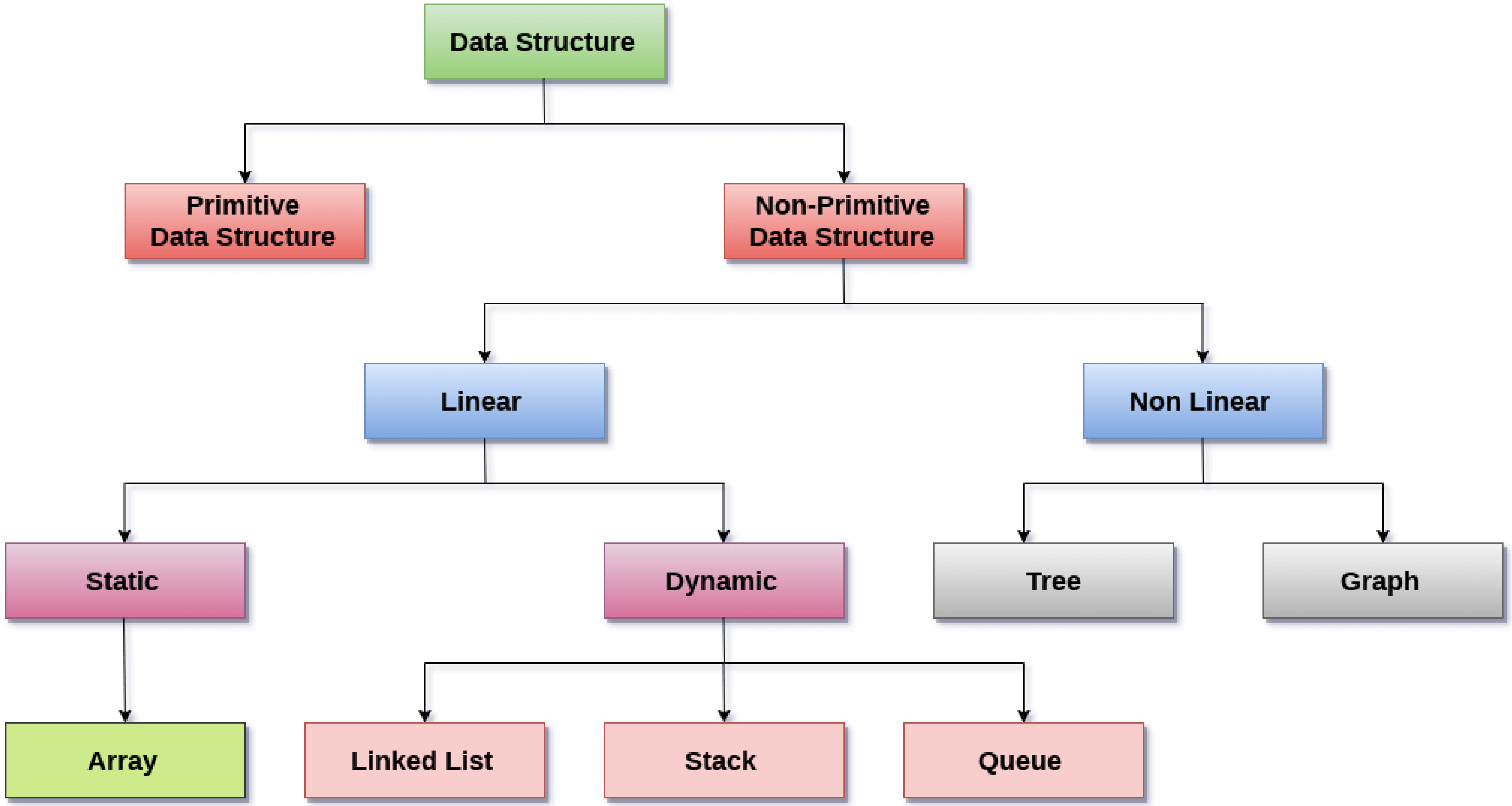
This Lecture

- Tree Data Structure

WHAT IS DATA STRUCTURE?

- A data structure is a data organization, management and storage format that enable efficient access and modification. a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data.
- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible
- Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

Data Structure Classifications



OPERATIONS ON DATA STRUCTURE

1) Traversing
2) Insertion
3) Deletion
4) Searching
5) Sorting
6) Merging

OPERATIONS ON DATA STRUCTURE

1) <u>Traversing</u>

Every data structure contains the set of data elements. Traversing the data structure means visiting each element of the data structure in order to perform some specific operation like searching or sorting.

- Example: If we need to calculate the average of the marks obtained by a student in 6 different subject, we need to traverse the complete array of marks and calculate the total sum, then we will divide that sum by the number of subjects i.e. 6, in order to find the average.

OPERATIONS ON DATA STRUCTURE

2) Insertion:

Insertion can be defined as the process of adding the elements to the data structure at any location. If the size of data structure is “n” then we can only insert “n” 1 data elements into it.

3) Deletion:

The process of removing an element from the data structure is called Deletion. We can delete an element from the data structure at any random location. If we try to delete an element from an empty data structure, then underflow occurs.

4) Searching:

The process of finding the location of an element within the data structure is called Searching. There are two algorithms to perform searching, Linear Search and Binary Search.

5) Sorting:

The process of arranging the data structure in a specific order is known as Sorting. There are many algorithms that can be used to perform sorting, for example, insertion sort, selection sort, bubble sort, etc.

6) Merging:

When two lists List A and List B of size M and N respectively, of similar type of elements, clubbed or joined to produce the third list, List C of size (M+N), then this process is called merging

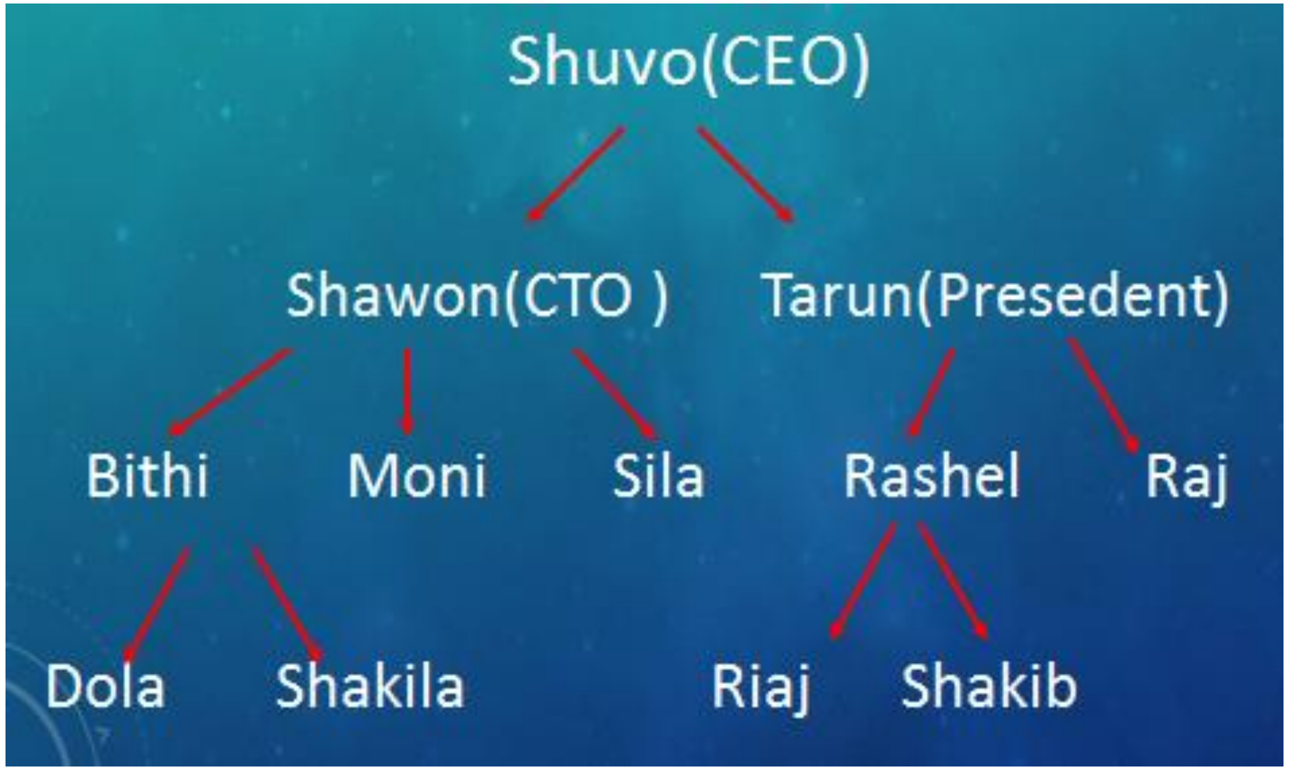
CHARACTERISTICS OF A DATA STRUCTURE

- <u>Correctness</u> Data structure implementation should implement its interface correctly

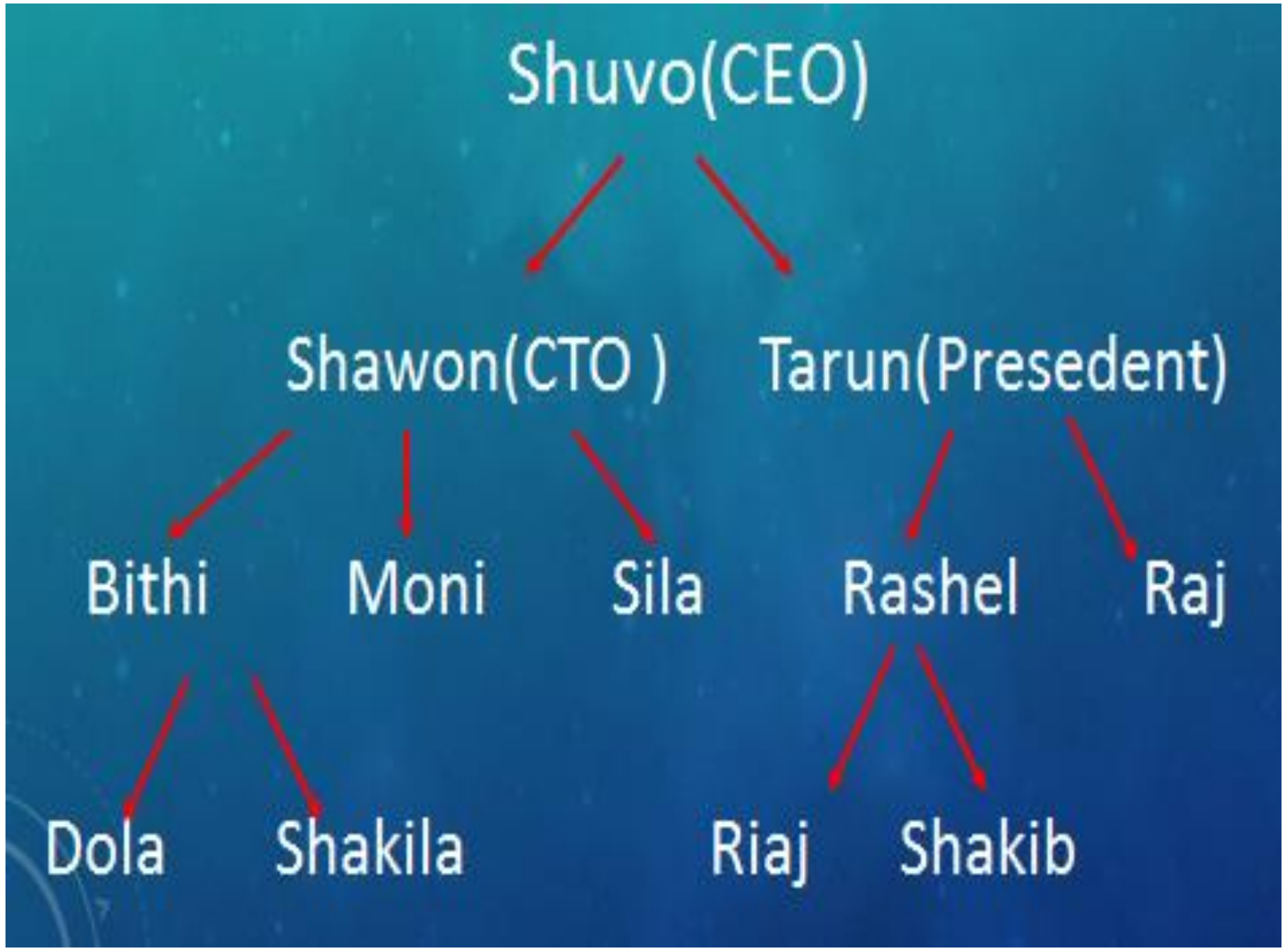
- <u>Time Complexity</u> Running time or the execution time of operations of data structure must be as small as possible

- <u>Space Complexity</u> Memory usage of a data structure operation should be as little as possible

Logic of Tree

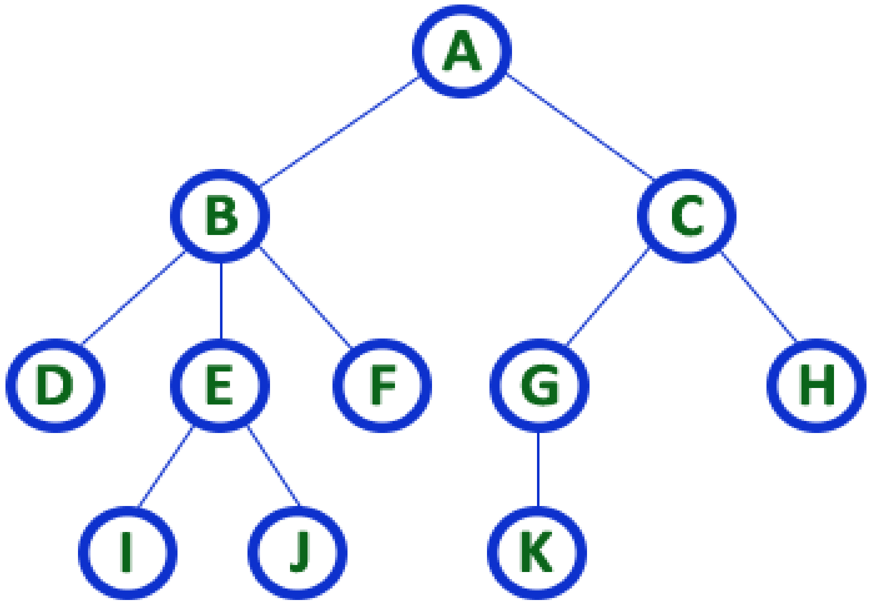


Logic of Tree



Tree Definition

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

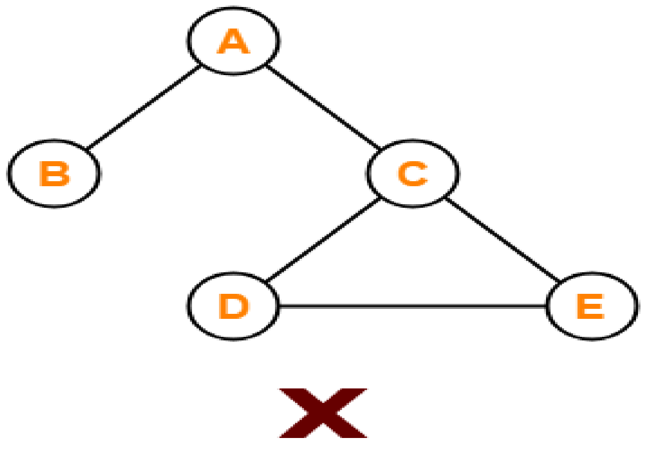


TREE with 11 nodes and 10 edges

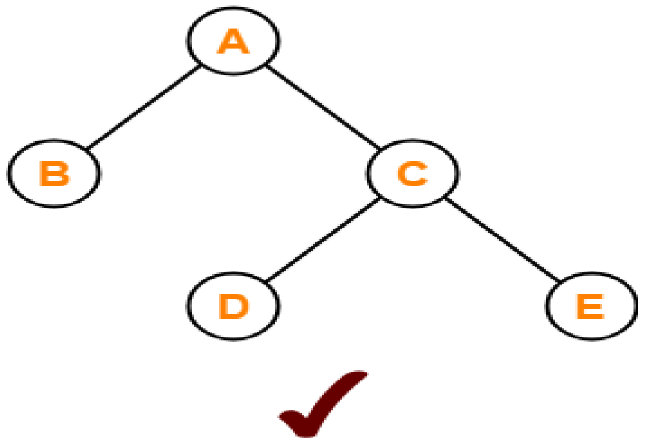
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Tree Definition

- Tree is a non-linear data structure which organizes data in a hierarchical structure and this is a recursive.
- A tree is a connected graph without any circuits.
- If in a graph, there is one and only one path between every pair of vertices, then graph is called as a tree.



This graph is not a Tree

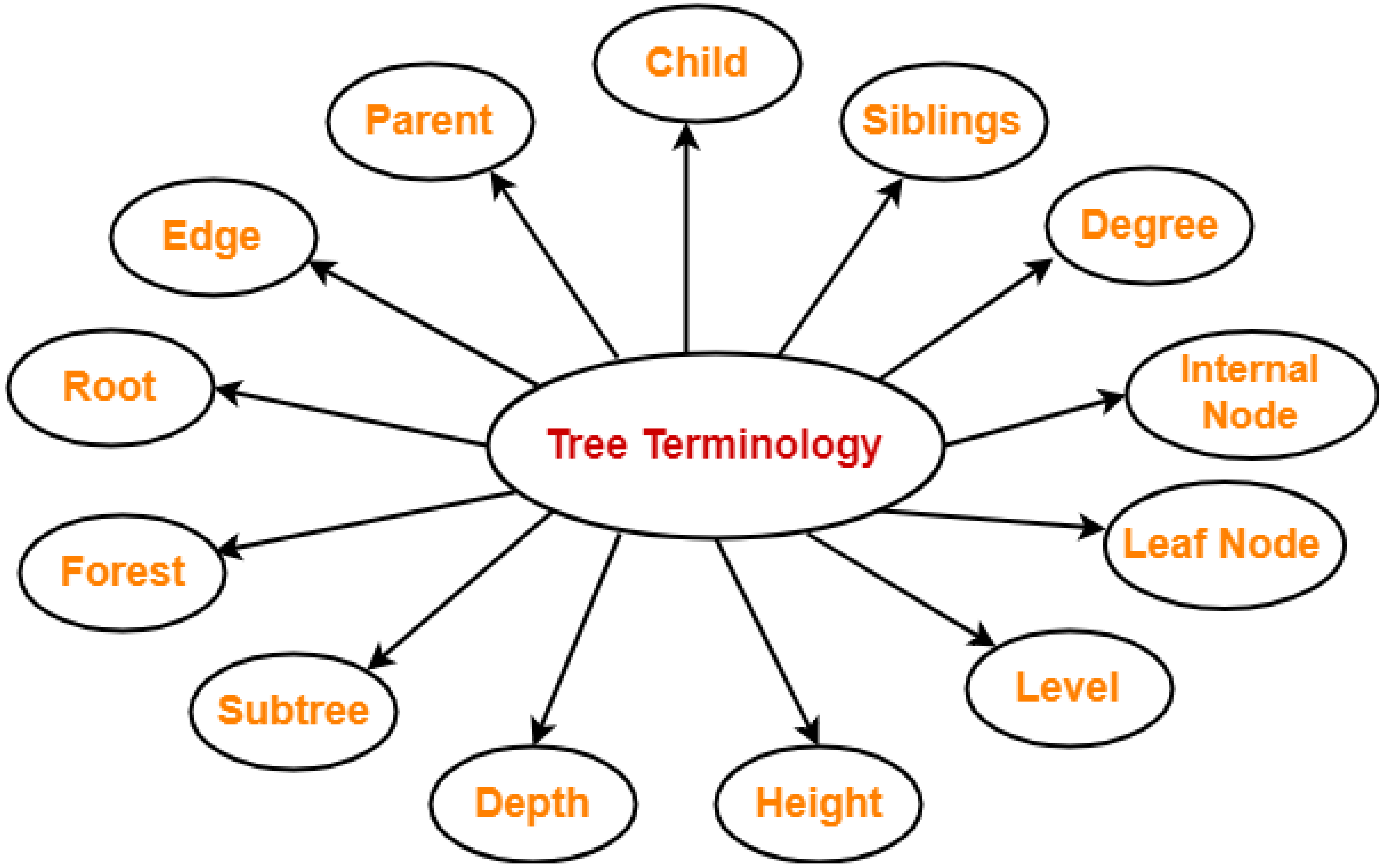


This graph is a Tree

Properties of Trees

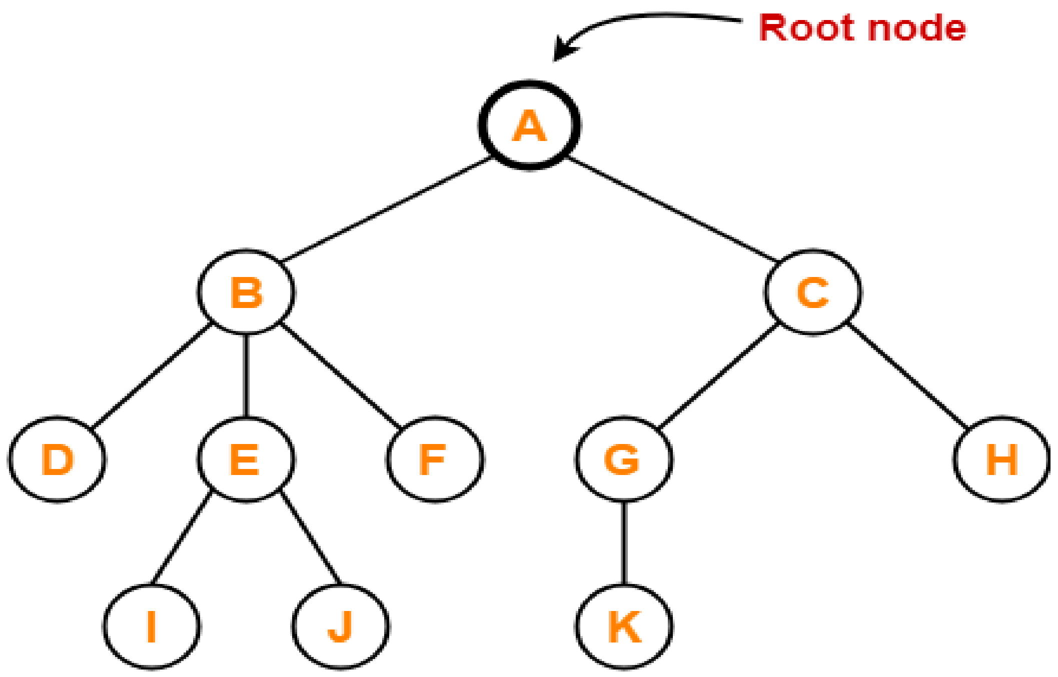
- There is one and only one path between every pair of vertices in a tree.
- A tree with n vertices has exactly (n-1) edges.
- A graph is a tree if and only if it is minimally connected.
- Any connected graph with n vertices and (n-1) edges is a tree.

Basic Tree Terminology



Root

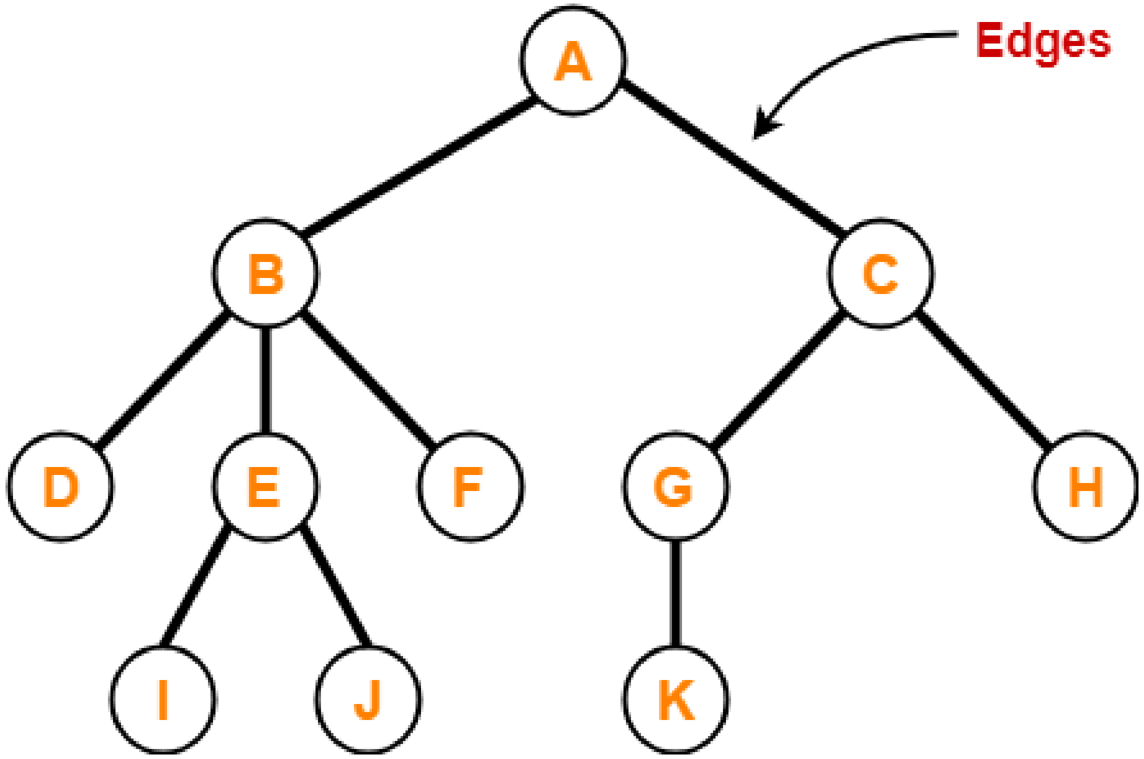
- The first node from where the tree originates is called as a root node.
- In any tree, there must be only one root node.
- We can never have multiple root nodes in a tree data structure.



Here, node **A** is the only root node

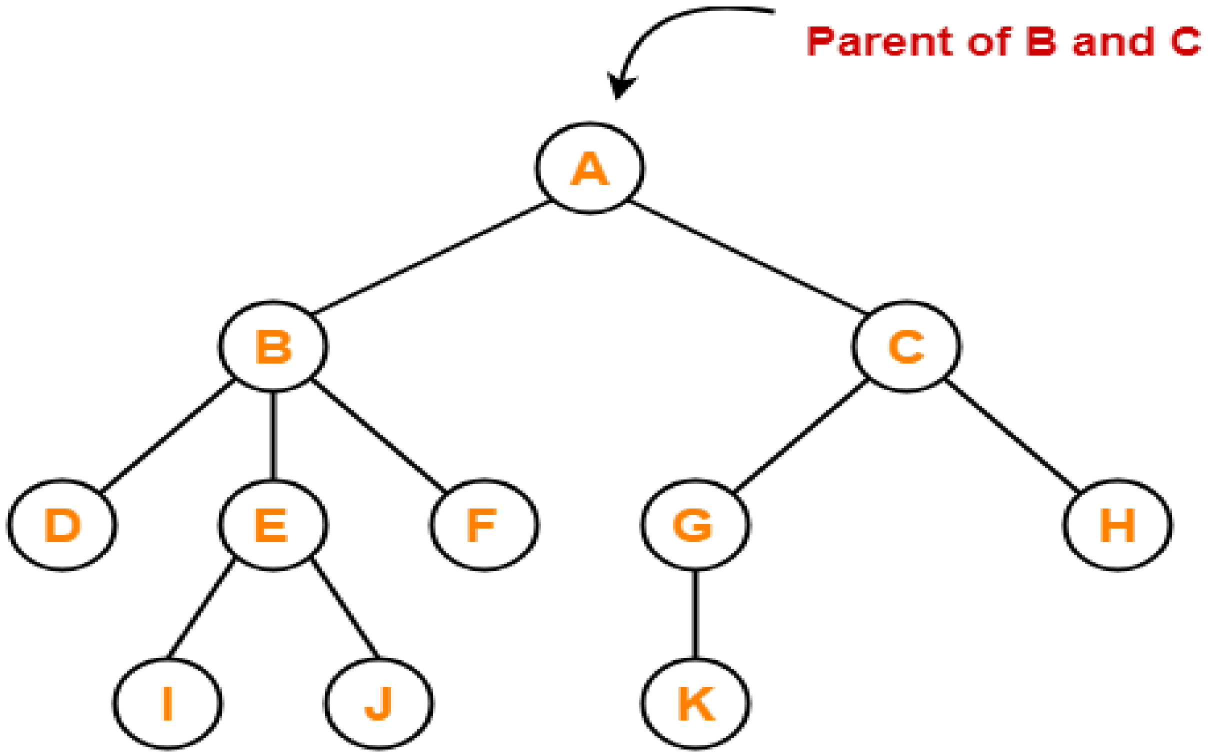
Edge

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



Parent Nodes

- 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.

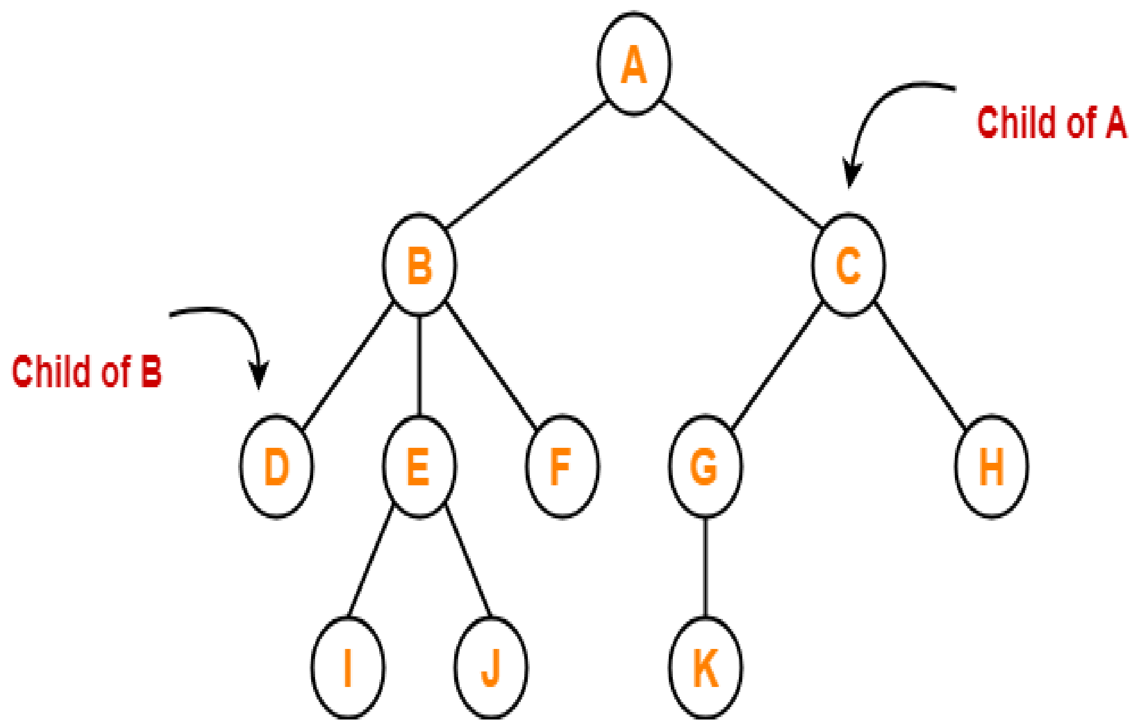


Here,

- 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

Child Nodes

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

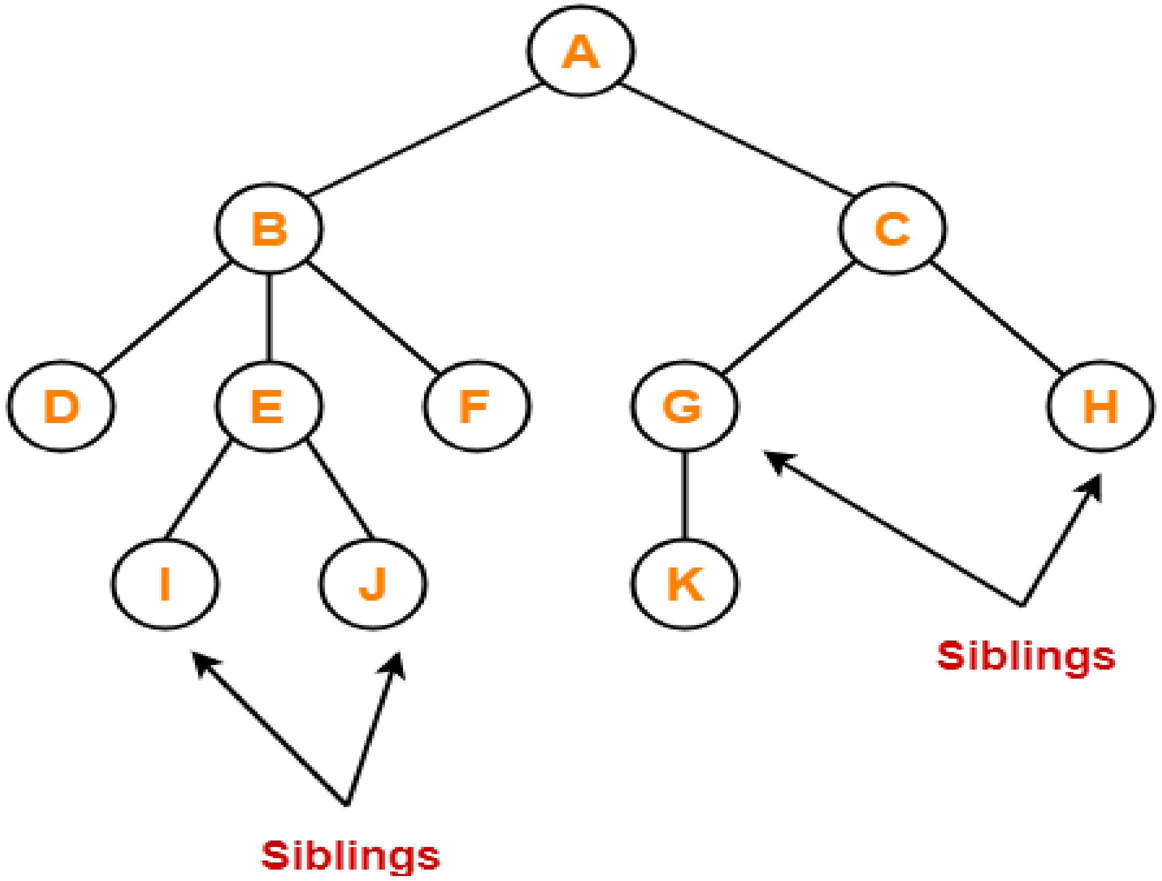


Here,

- 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

Siblings

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

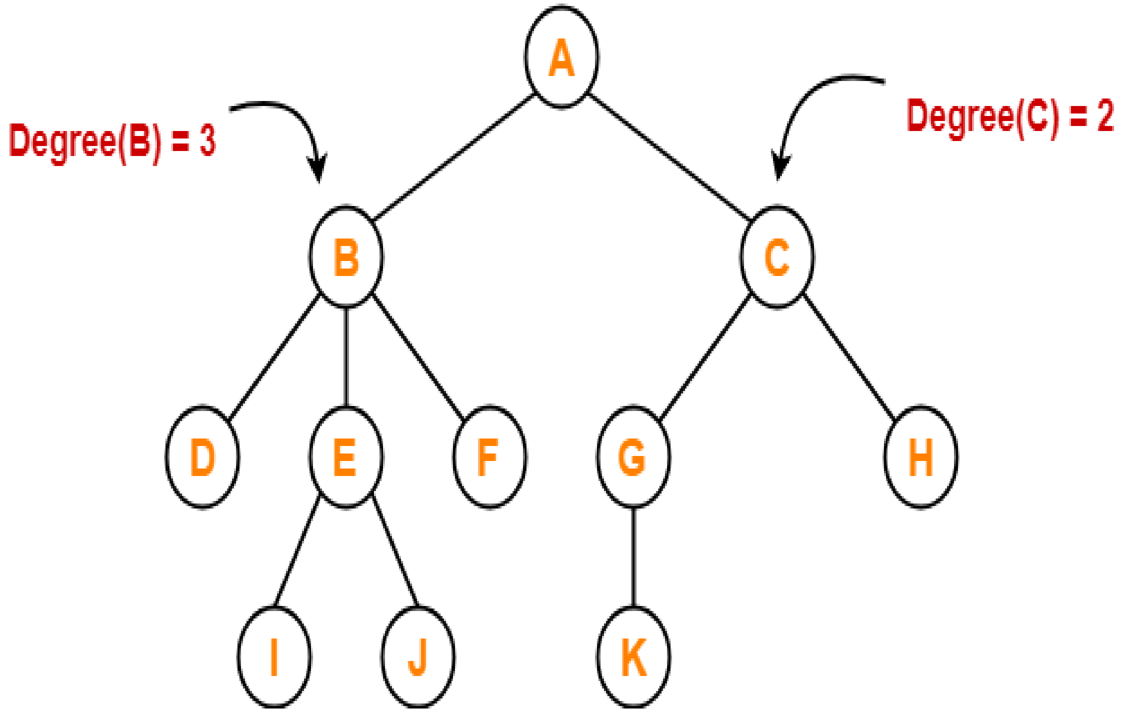


Here,

- 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

Degree of a Node

- 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.

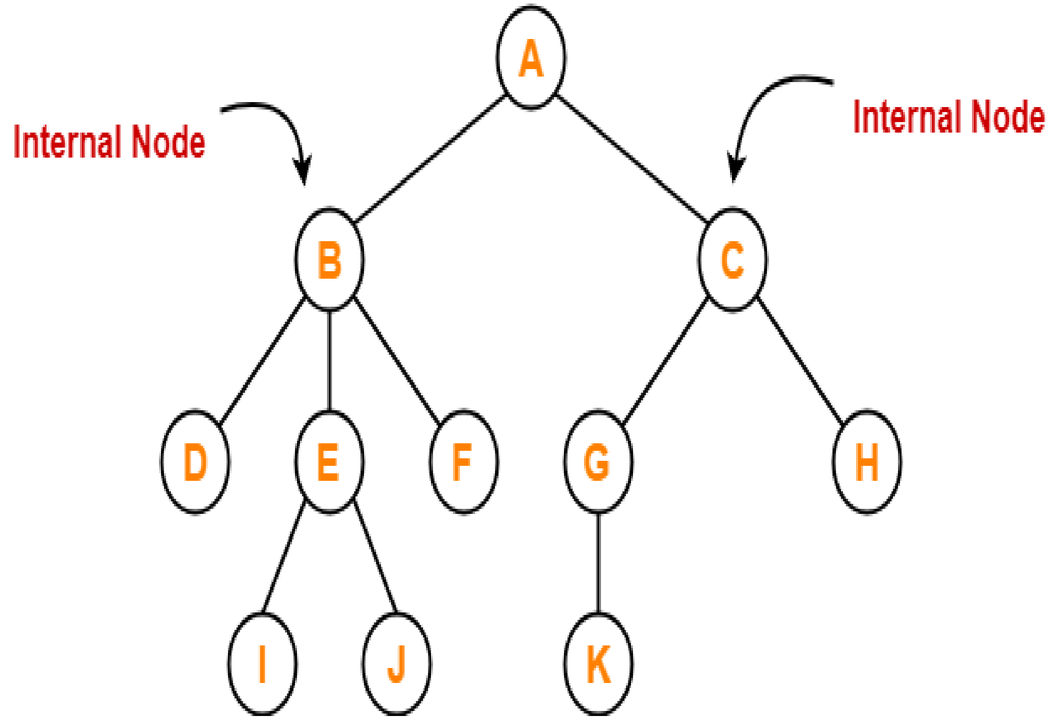


Here,

- Degree of 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

Internal Node

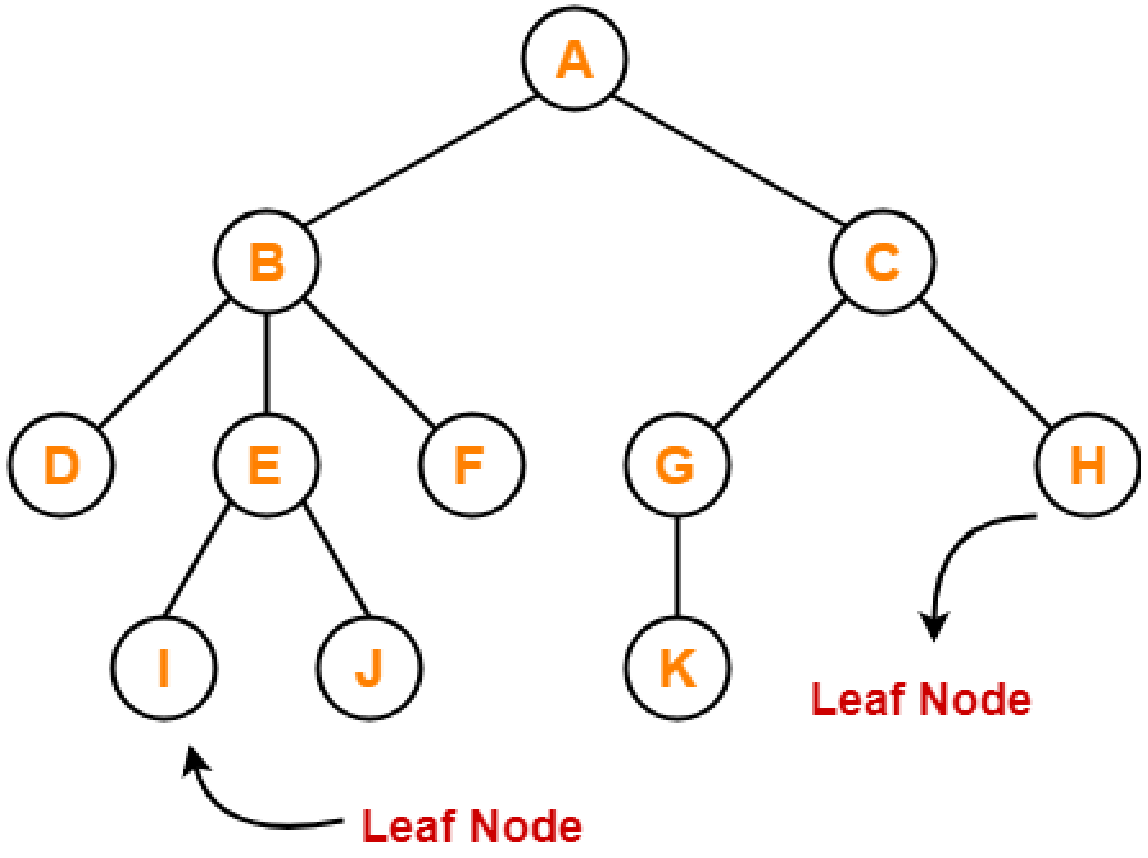
- The node which has at least one child is called as an internal node.
- Internal nodes are also called as non-terminal nodes.
- Every non-leaf node is an internal node.



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

Leaf Node

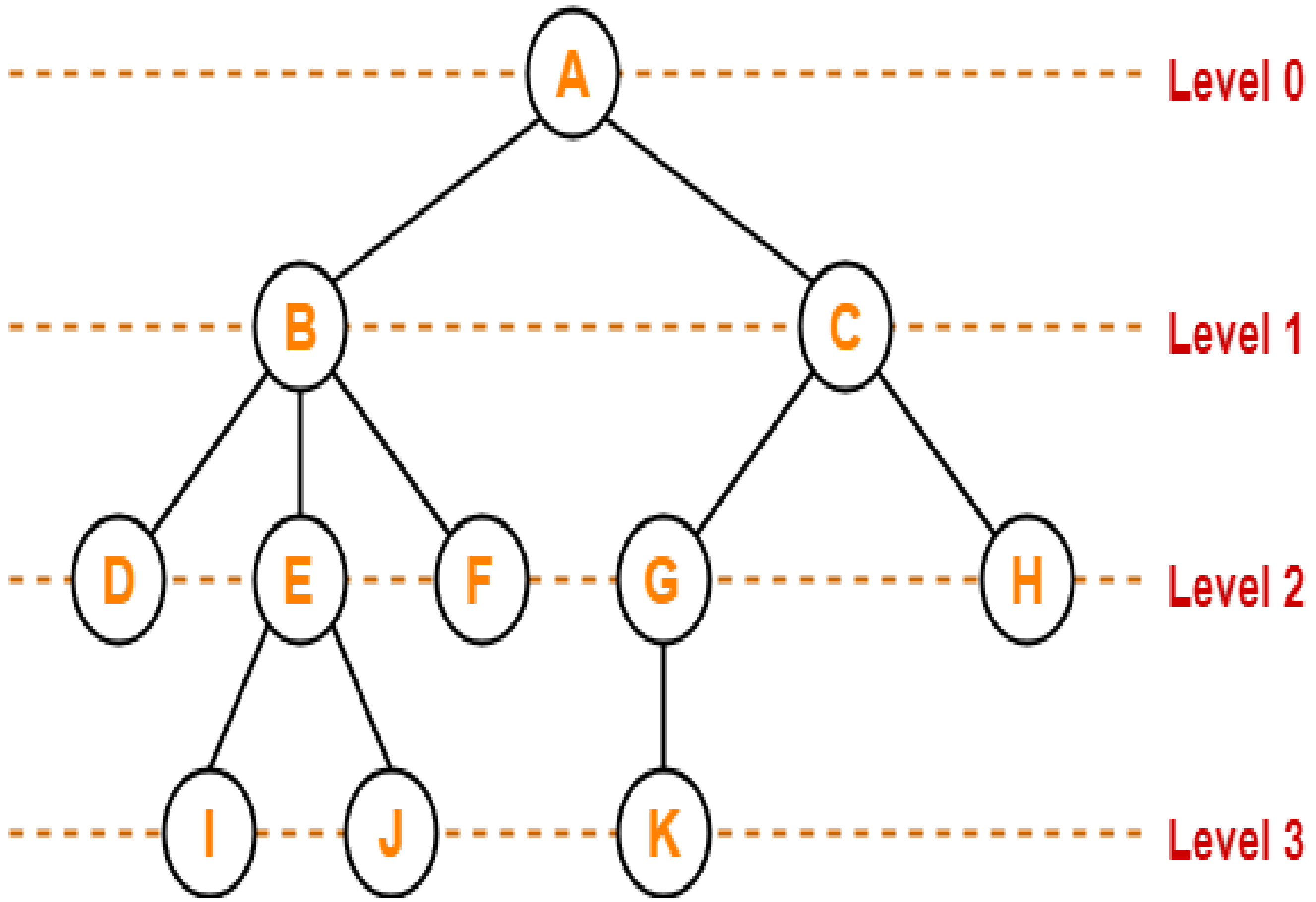
- The node which does not have any child is called as a leaf node.
- Leaf nodes are also called as external nodes or terminal nodes.



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

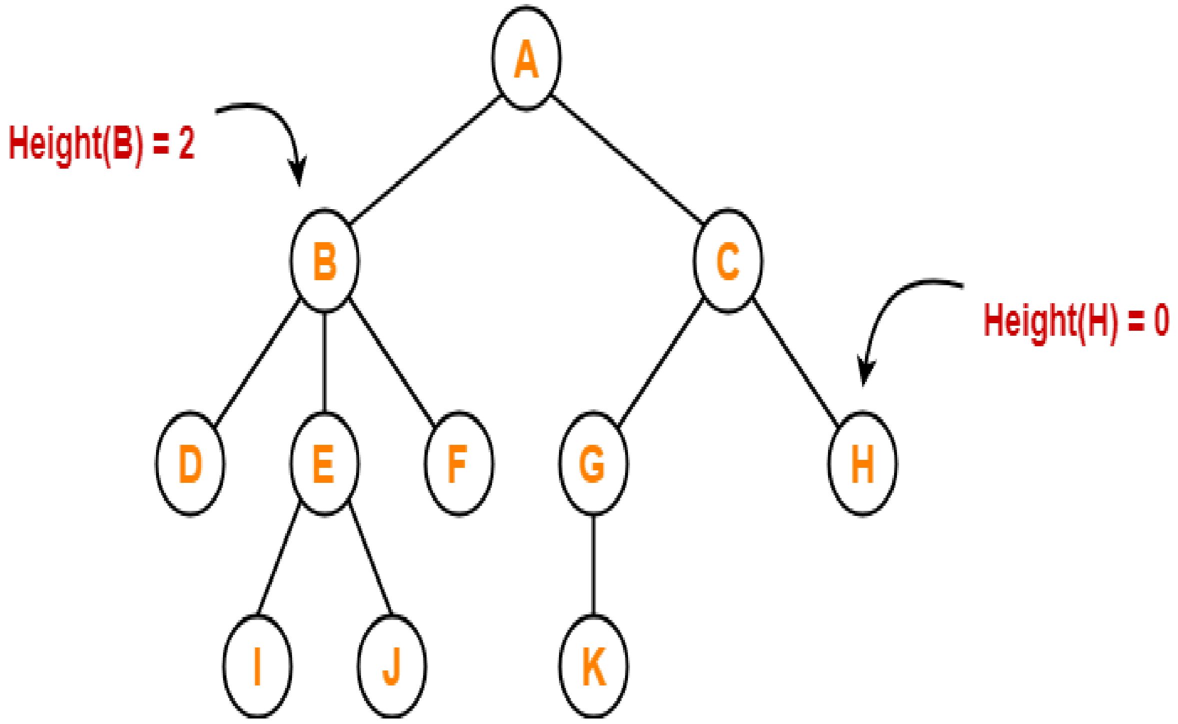
Level

- 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

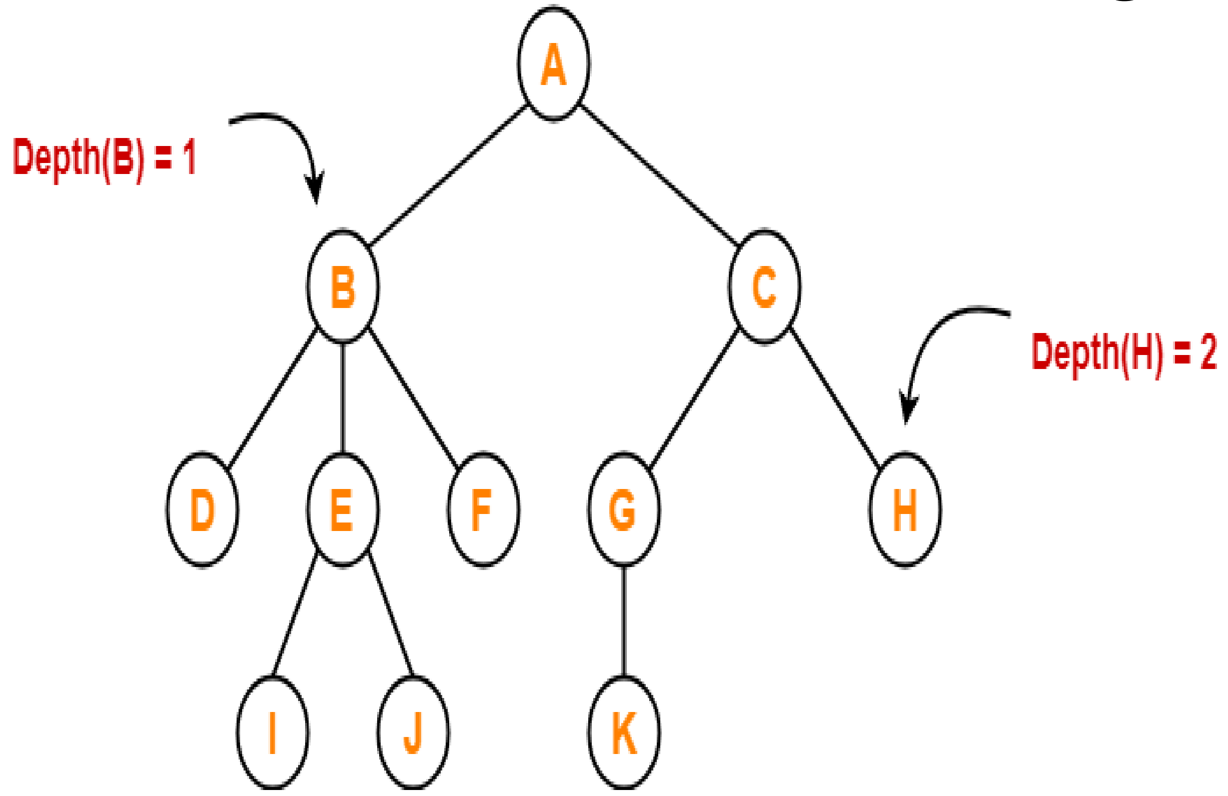


Here

- 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

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.

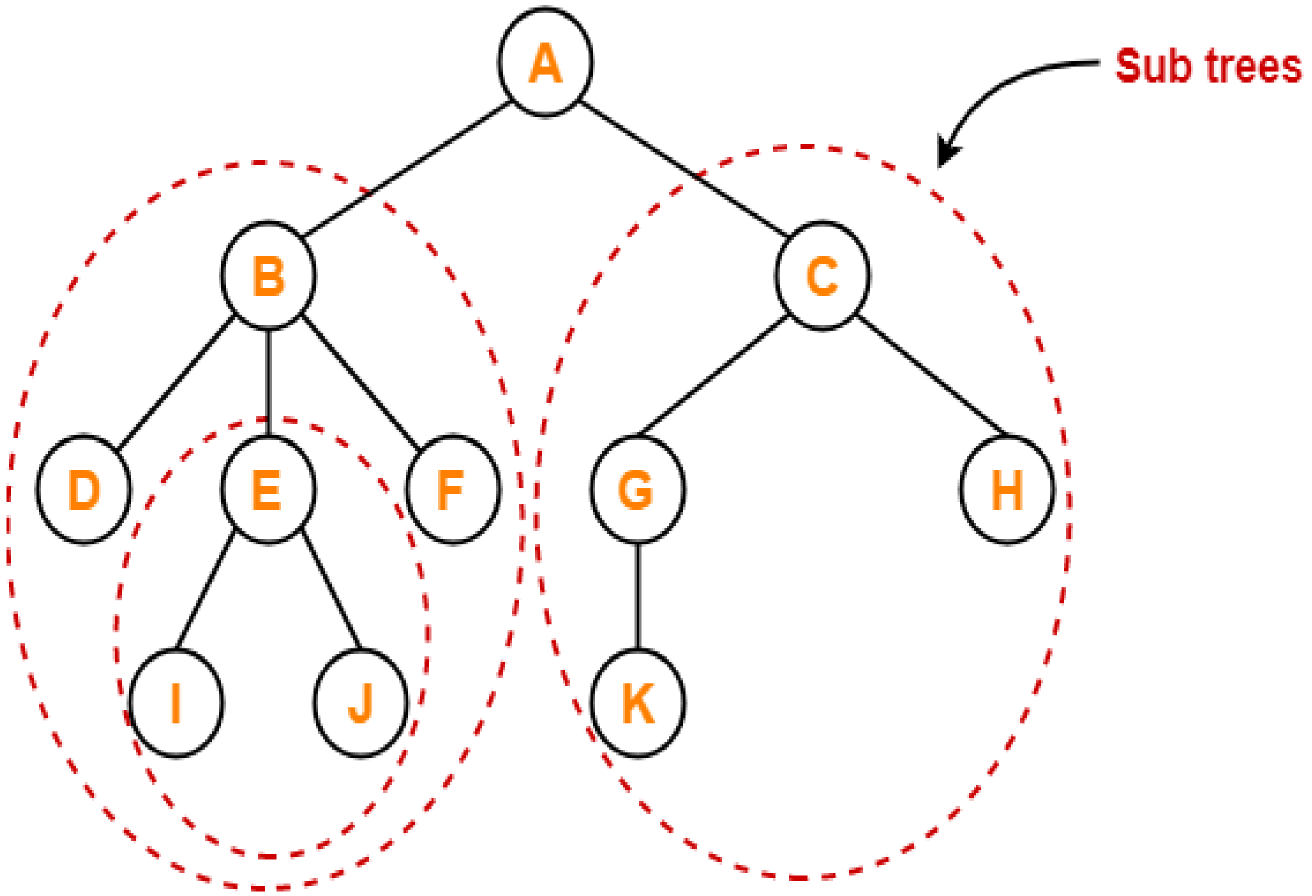


Here

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

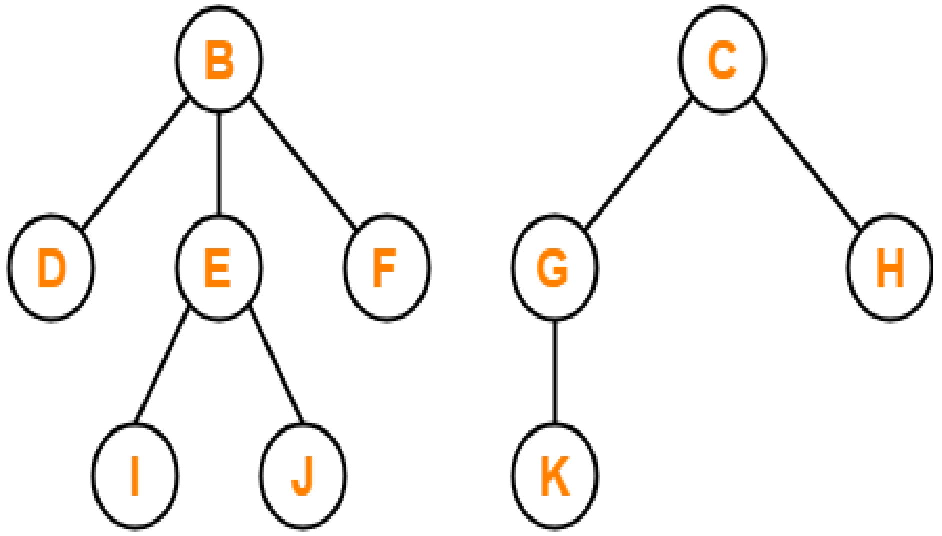
Sub-tree

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



Forest

- A forest is a set of disjoint trees.



Forest

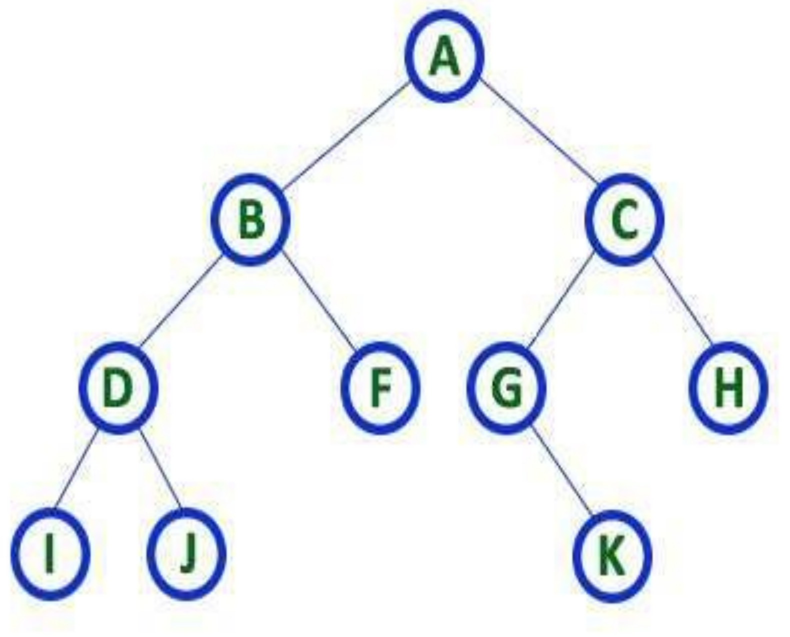
Advantages of Tree

- Tree reflects structural relationships in the data.
- It is used to represent hierarchies.
- It provides an efficient insertion and searching operations.
- Trees are flexible. It allows to move sub trees around with minimum effort.

Tree Representation

A tree data structure is represented using two methods.

- ✓Array Representation
- ✓Linked List Representation



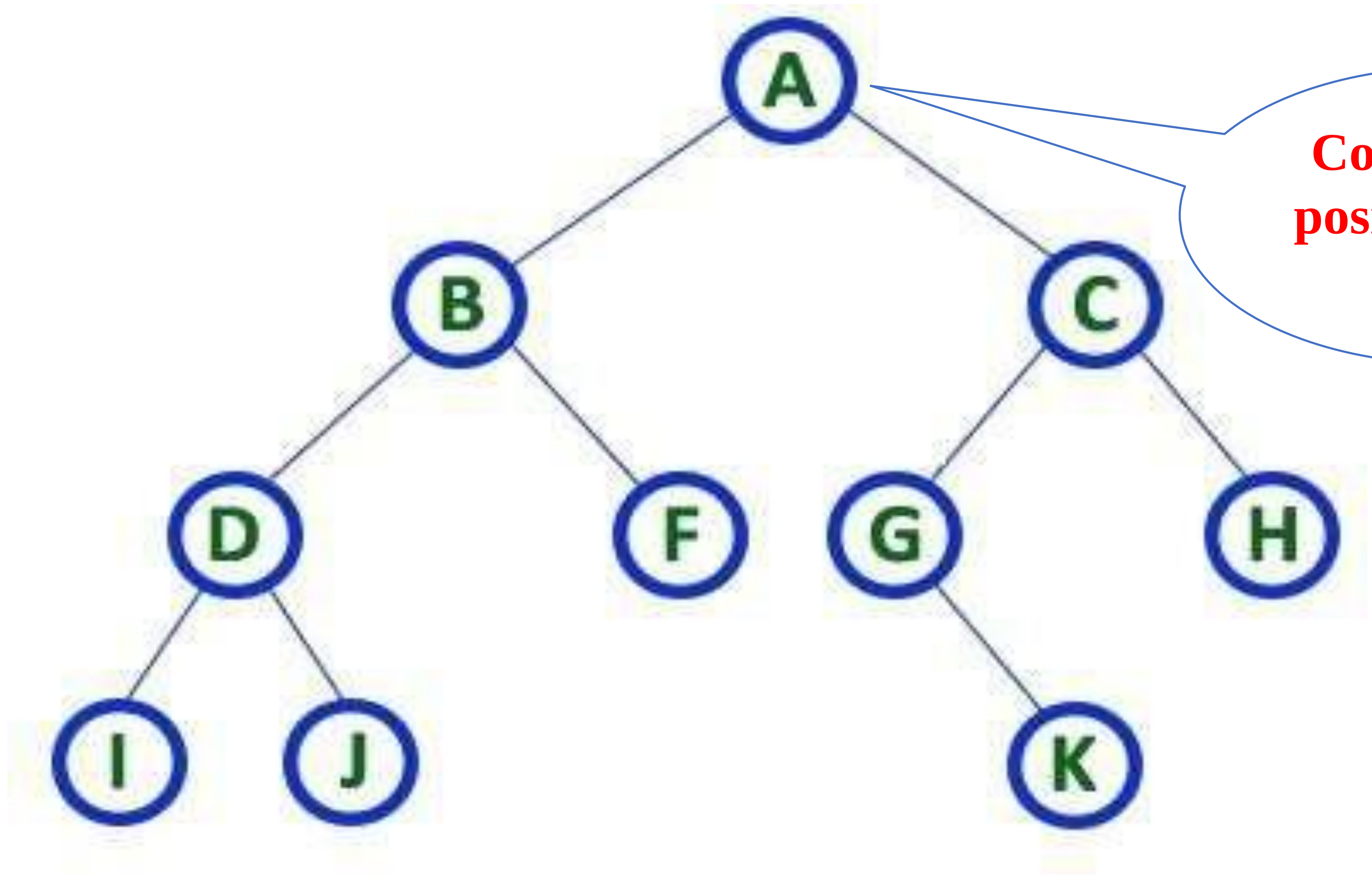
Array Representation

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

- Consider the root node at index ‘0’
- LEFT child is placed at 2i+1
- RIGHT child is placed at 2i+2

Where ‘i’ is the position of the parent

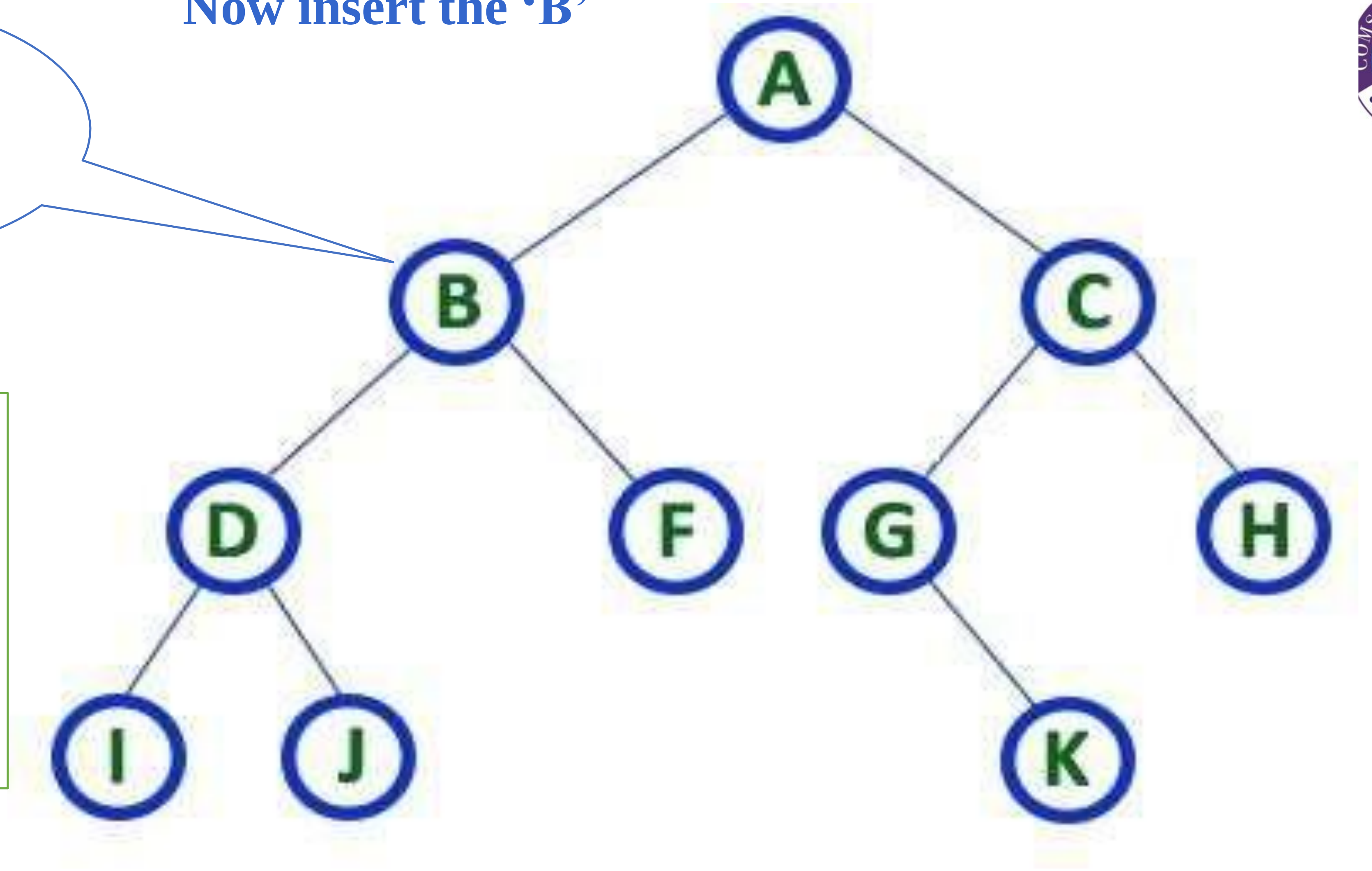
Example



Consider the position of 'A' is '0'

Array	A															
Index	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the ‘B’

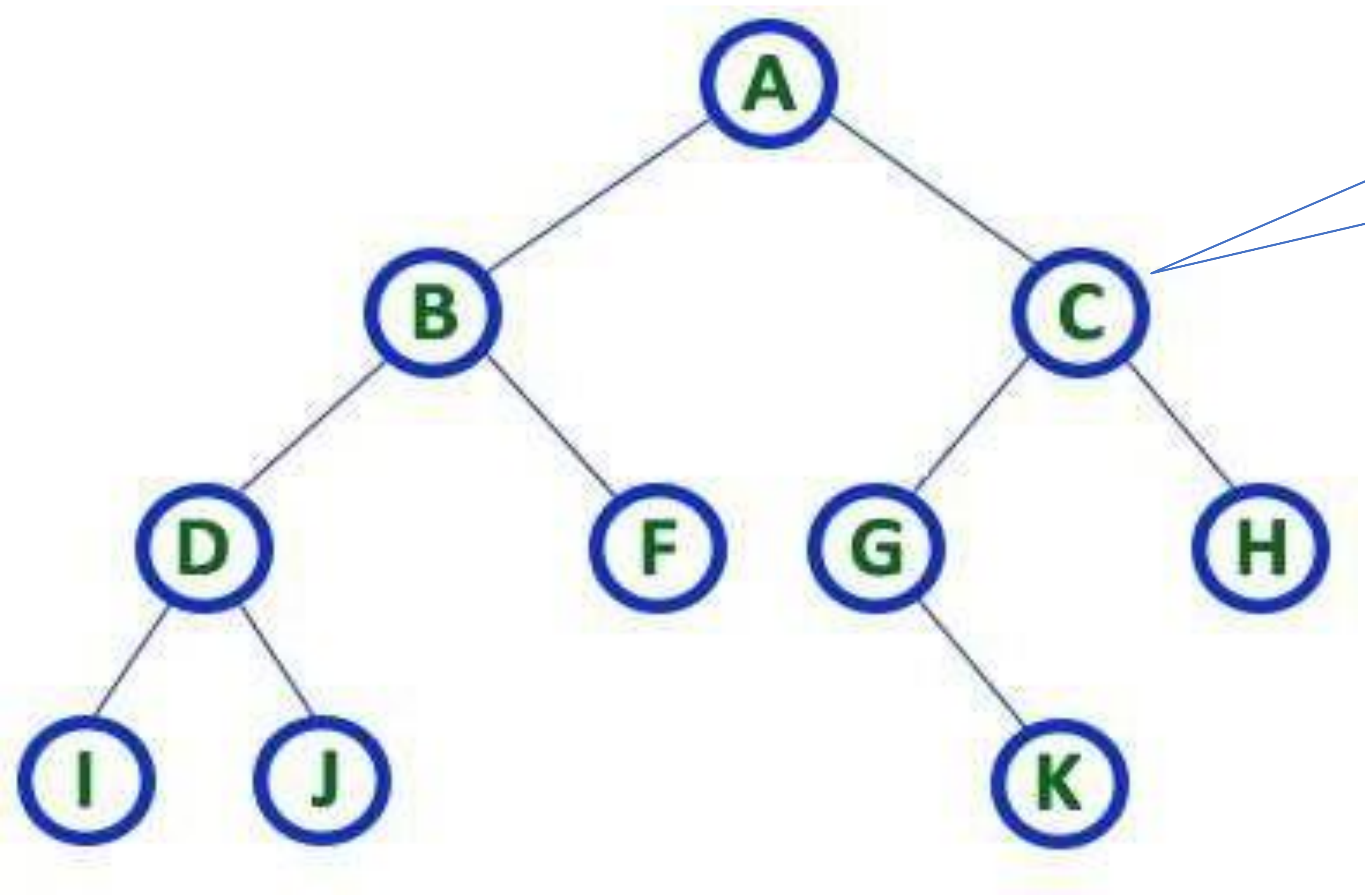


Now inserting the ‘B’

where Pos of ‘A’ is ‘0’
‘B’ is a left child of ‘A’
then formula is 2i+1
2(0)+1
=1

Array	**A**	**B**														
Index	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'C'



Now inserting the 'C'

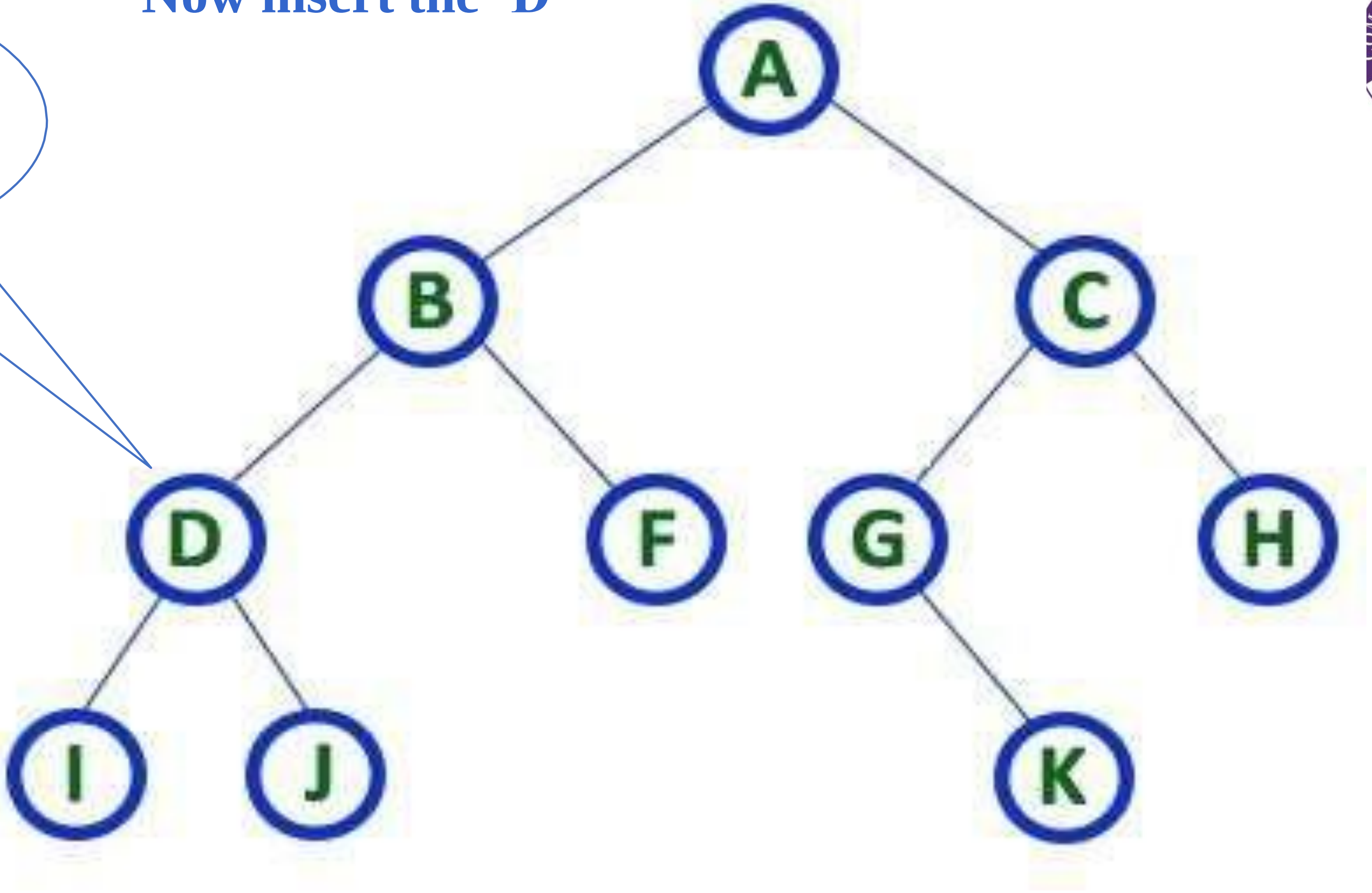
where Pos of 'A' is '0'
'C' is a Right child of 'A' then formula is
2i+2
= 2(0)+2
=2

Array	A	B	C													
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the ‘D’

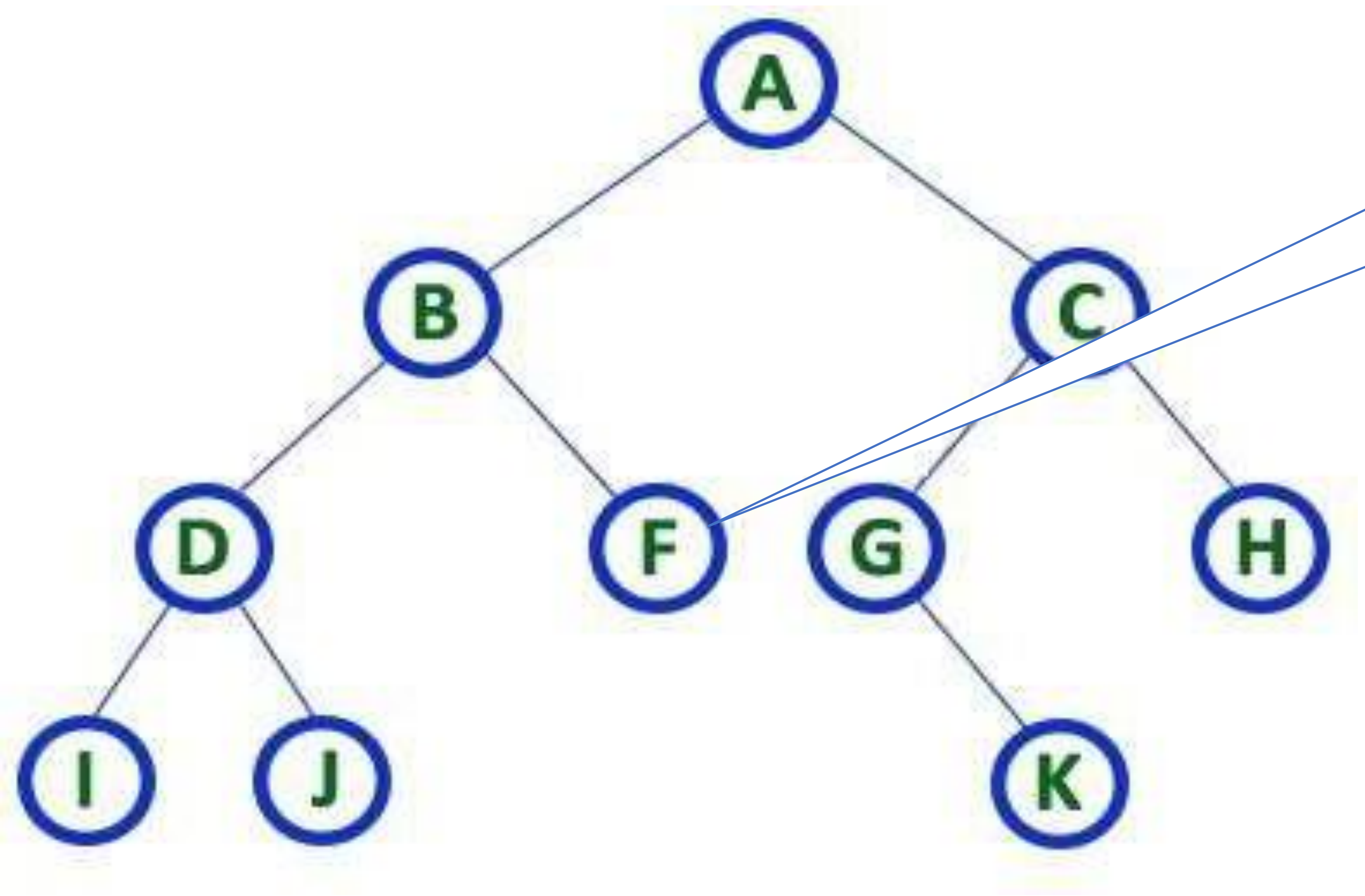
Now inserting the ‘D’

where Pos of ‘B’ is ‘1’
‘D’ is a left child of ‘B’
then formula is 2i+1
2(1)+1
= 3



Array	**A**	**B**	**C**	**D**												
Index	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

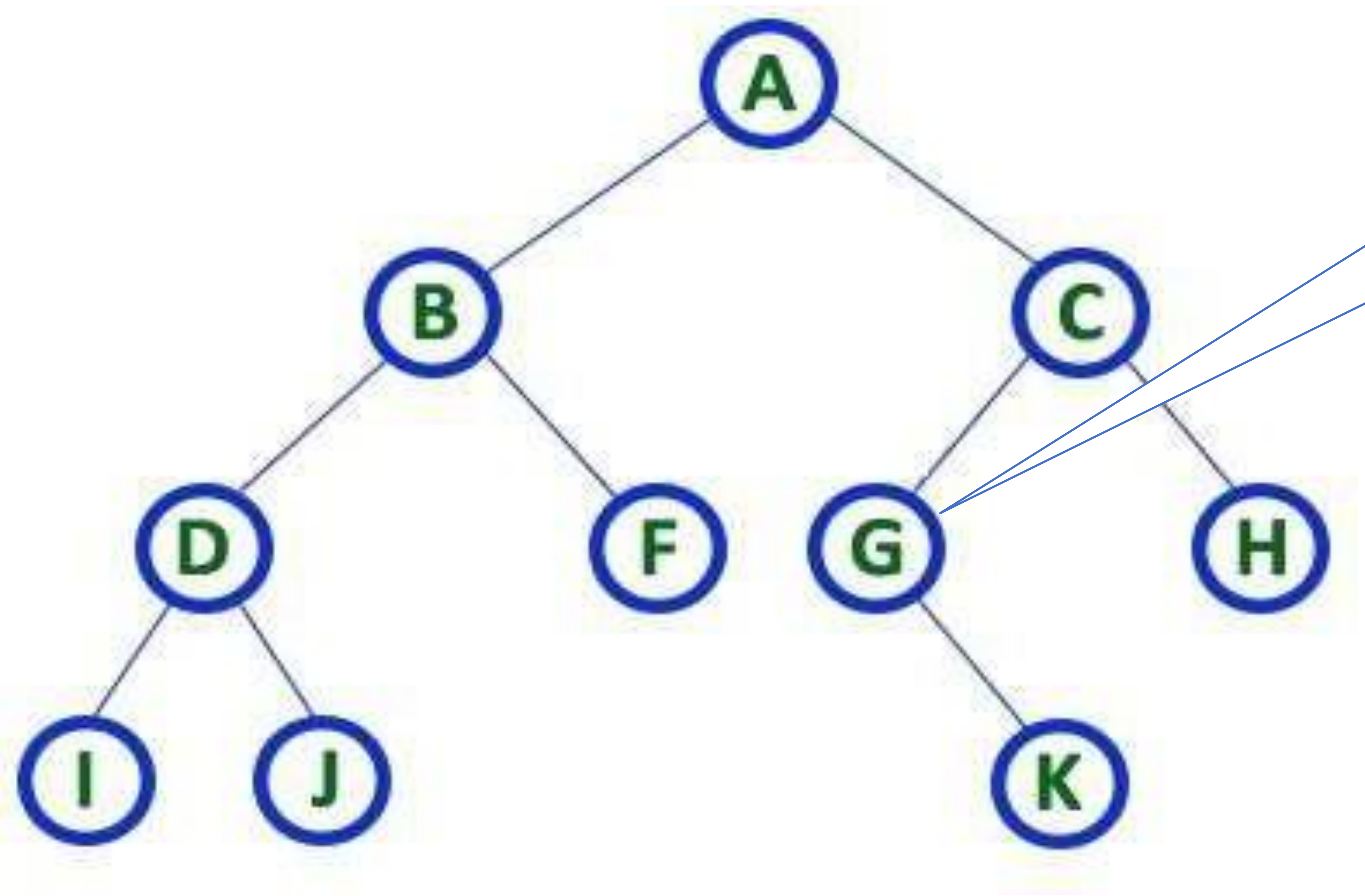
Now insert the ‘F’



where Pos of ‘B’ is ‘1’
‘F’ is a Right child of ‘B’ then formula is
2i+2
= 2(1)+2
= 4

Array	A	B	C	D	F											
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the ‘G’

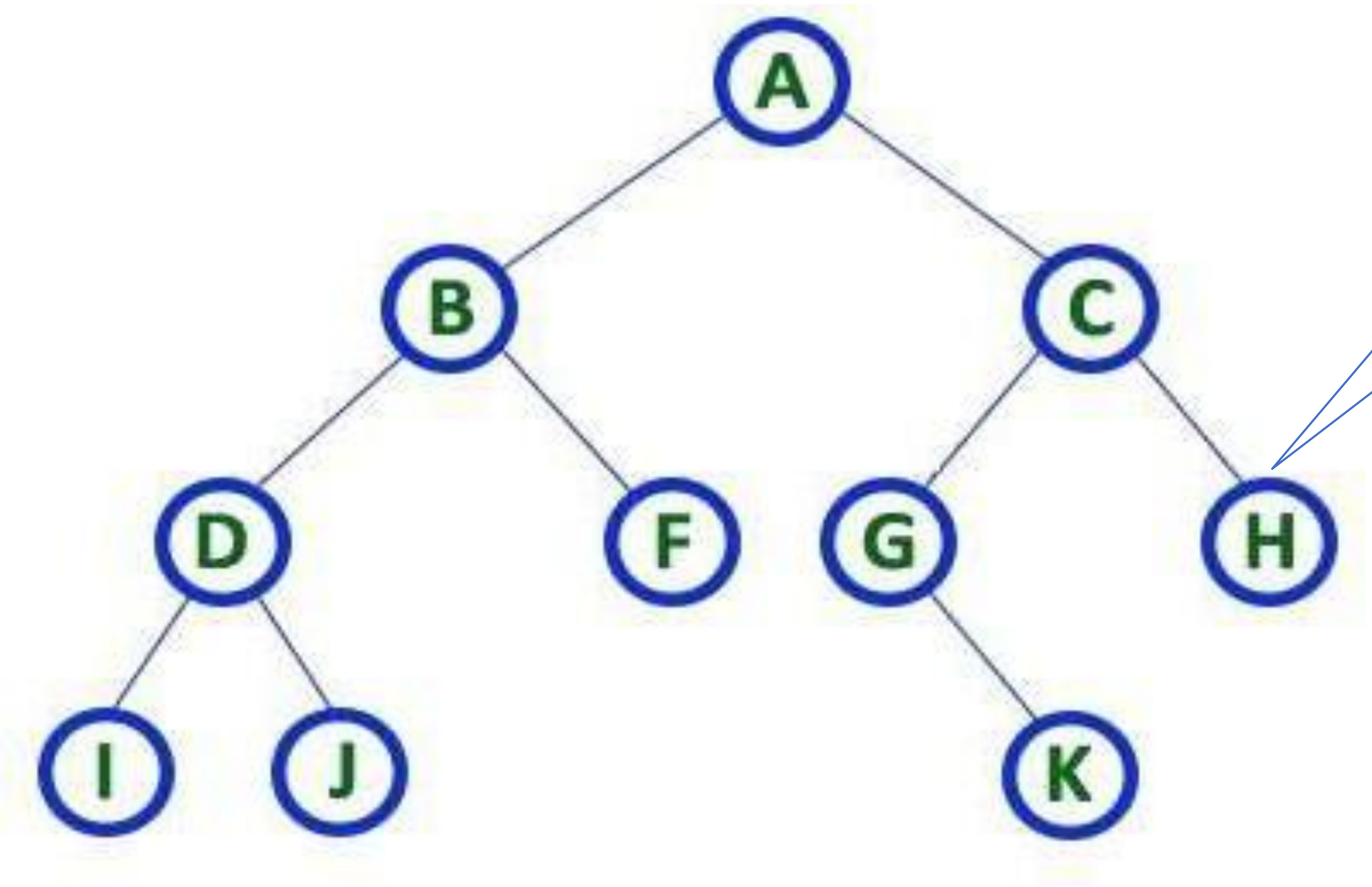


Now inserting the ‘G’

where Pos of ‘C’ is ‘2’
‘G’ is a left child of ‘C’
then formula is 2i+1
= 2(2)+1
= 5

Array	A	B	C	D	F	G										
Index	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'H'



Now inserting the 'H'

where Pos of 'C' is '2' 'H' is a Right child of 'C' then formula is

2i+2

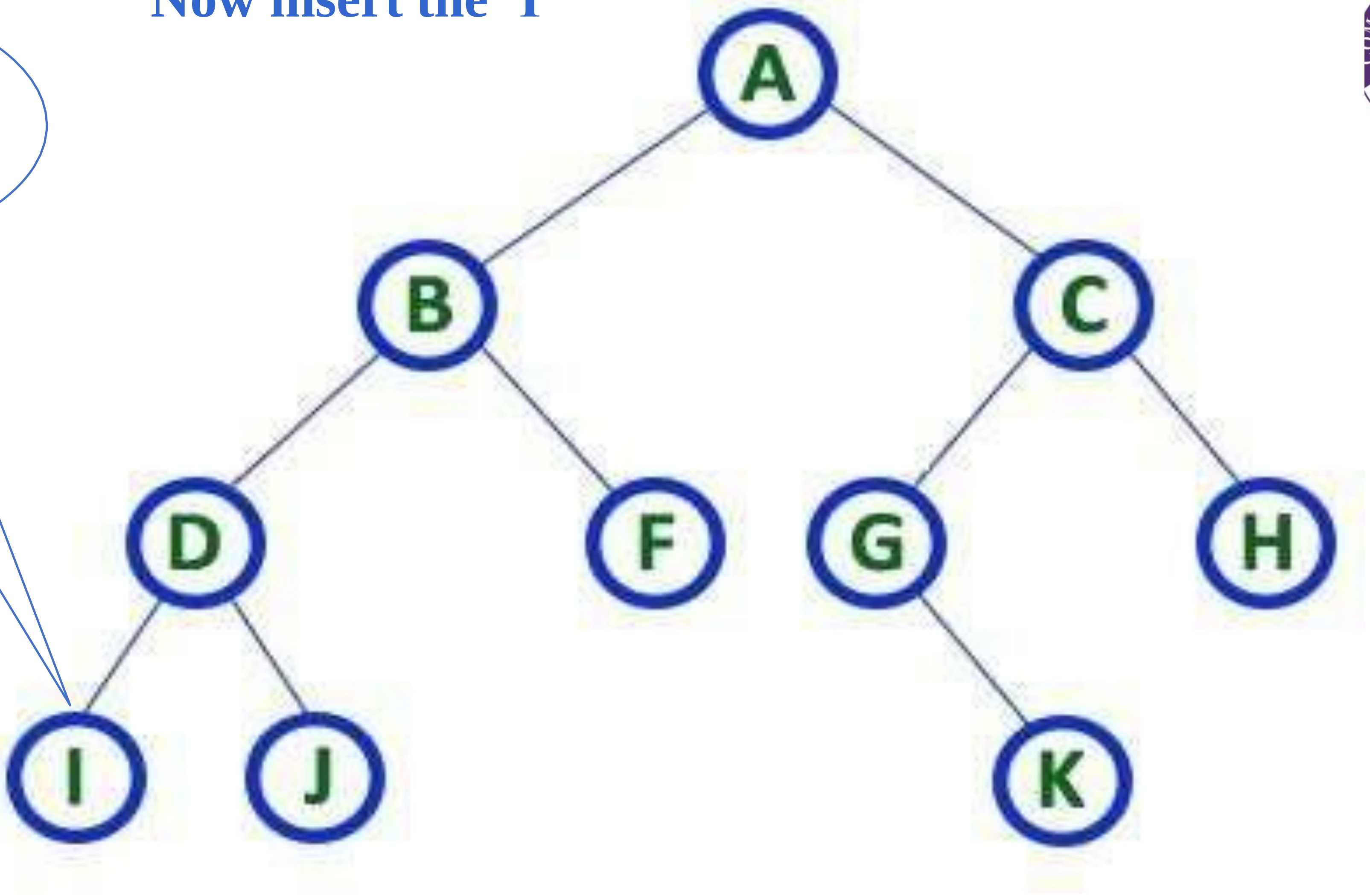
= 2(2)+2

= 6

Array	A	B	C	D	F	G	H									
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the ‘I’

Now inserting the ‘I’



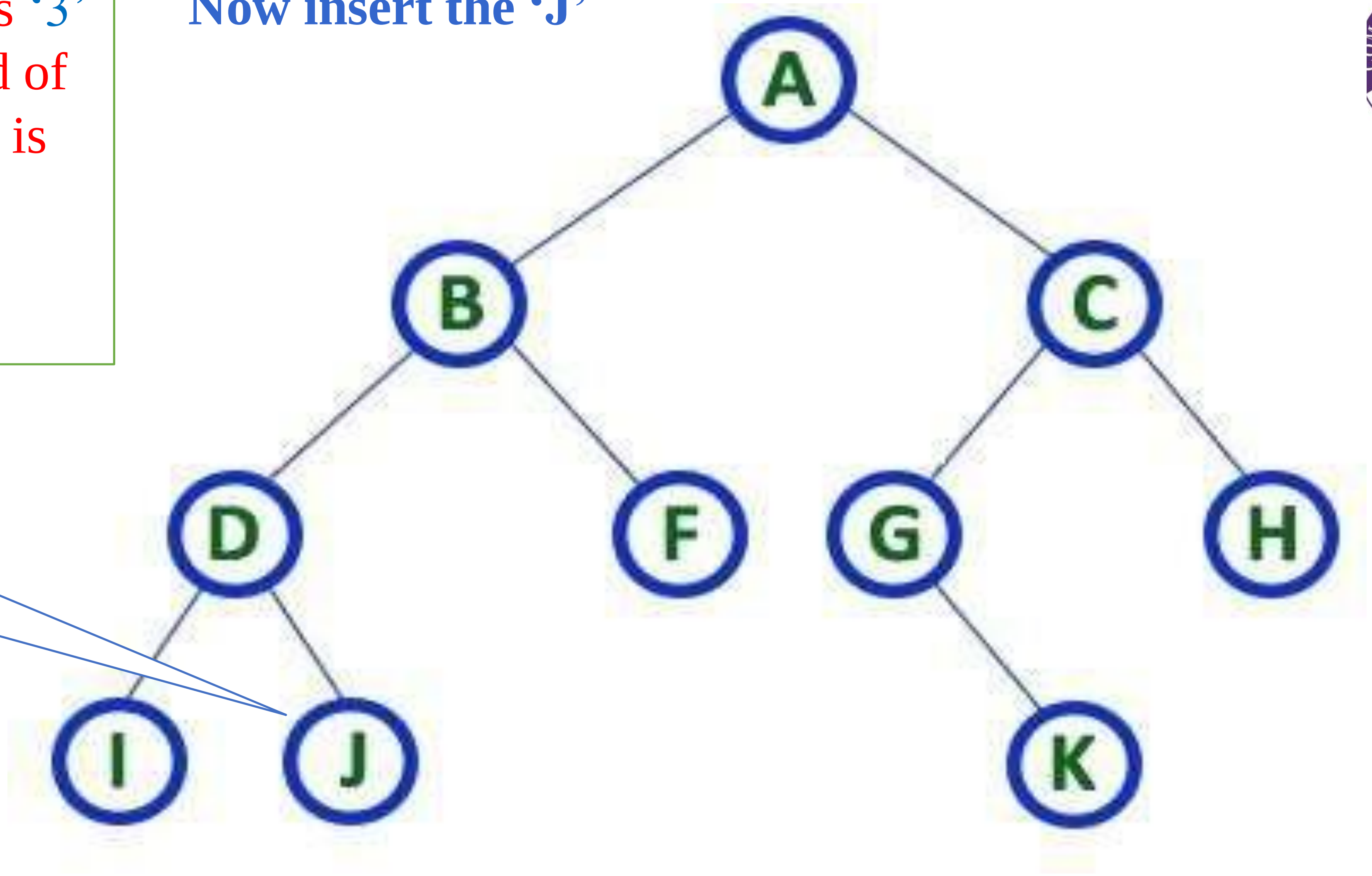
where Pos of ‘D’ is ‘3’
‘I’ is a left child of ‘D’
then formula is 2i+1
2(3)+1
= 7

Array	A	B	C	D	F	G	H	I								
Index	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the ‘J’

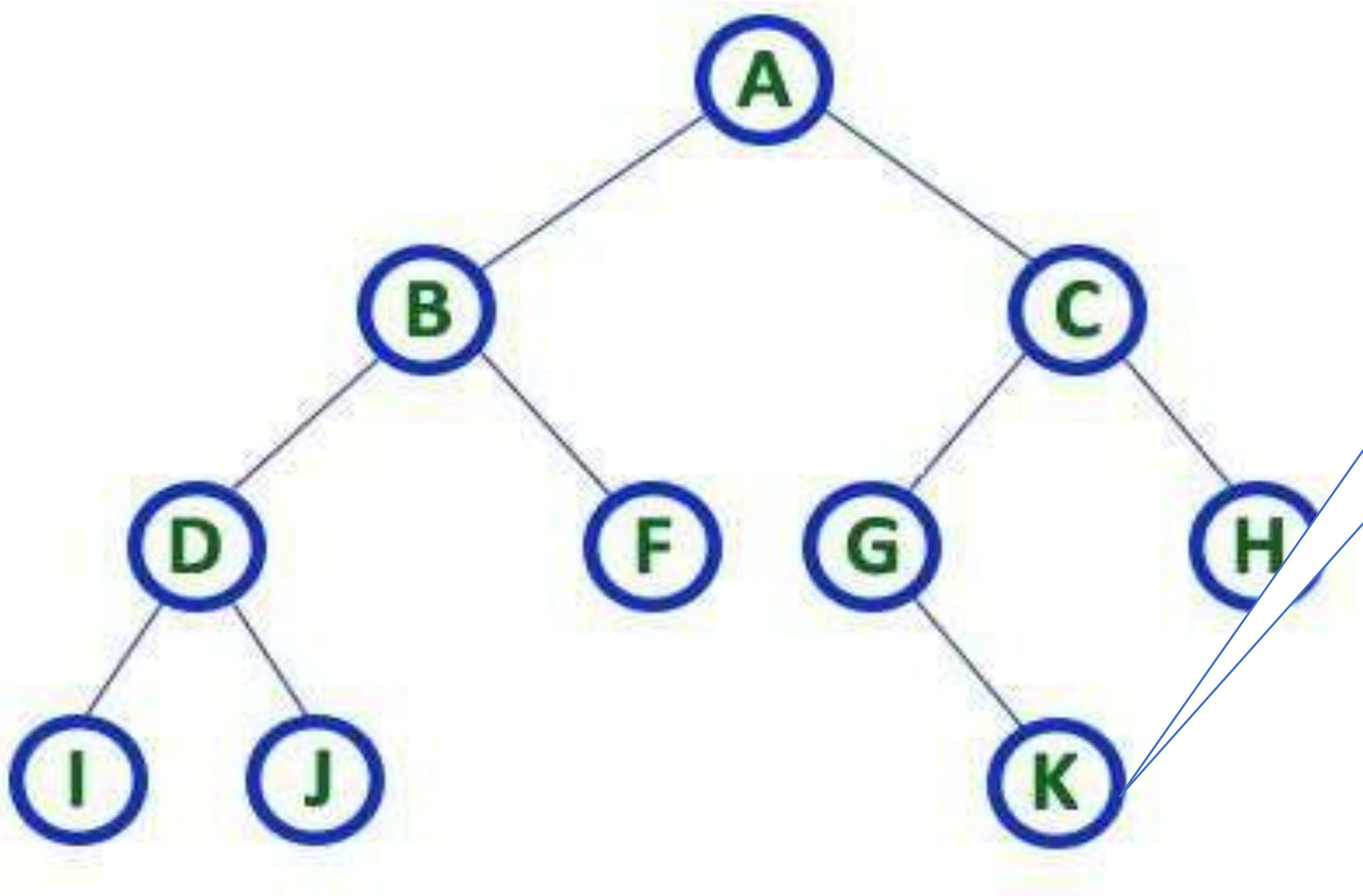
where Pos of ‘D’ is ‘3’
‘J’ is a Right child of ‘D’ then formula is
2i+2
2(3)+2
= 8

Now inserting the ‘J’



Array	**A**	**B**	**C**	**D**	**F**	**G**	**H**	**I**	**J**							
Index	**0**	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Now insert the 'K'



Now inserting the 'K'

where Pos of 'G' is '5' 'K' is a Right child of 'G' then formula is

2i+2

= 2(5)+2

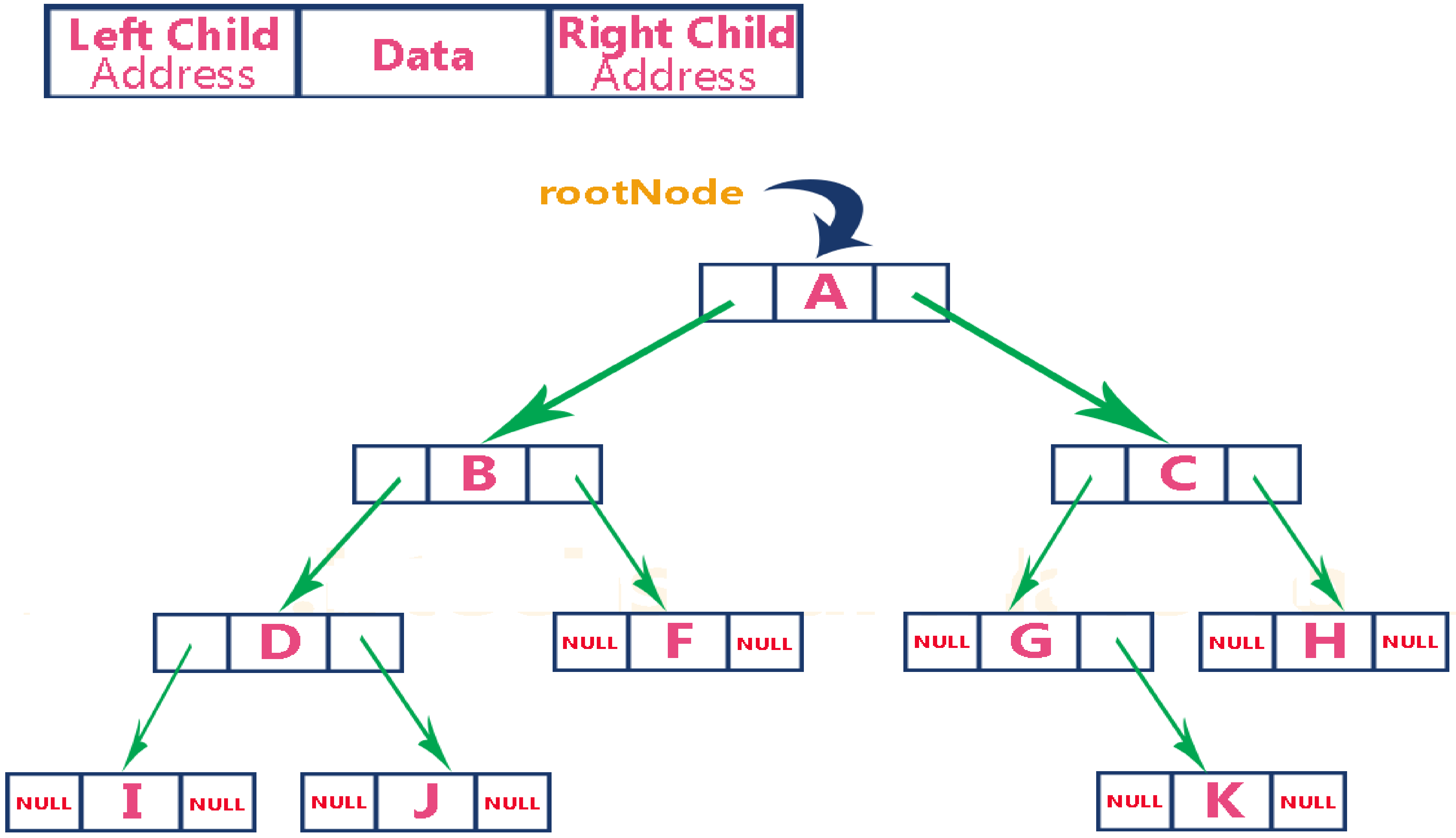
= 12

Array	A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-
Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Linked List Representation

We Use A Double Linked List To Represent A Binary Tree.

- In A Double Linked List, Every Node Consists Of Three Fields.
- First Field For Storing Left Child Address.
- Second For Storing Actual Data.
- Third For Storing Right Child Address.


Left Child Address
Data
Right Child Address
rootNode
A
B
C
D
NULL
F
NULL
NULL
G
NULL
H
NULL
NULL
I
NULL
NULL
J
NULL
NULL
K
NULL

Type of Trees

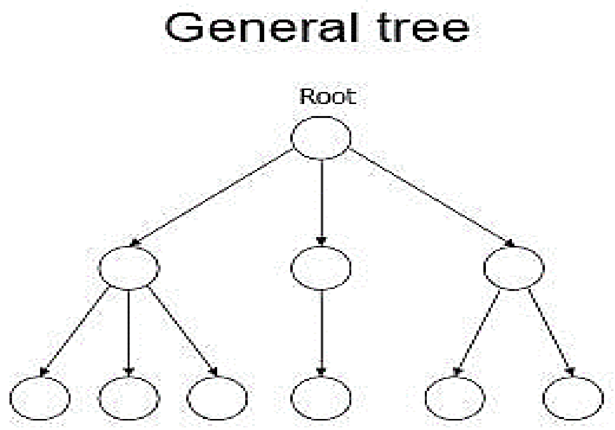
- General tree
- Binary tree
- Binary Search Tree

General Tree

- A general tree is a **data structure** in that each node can have infinite number of children .
- In general tree, root has **in-degree 0** and maximum **out-degree n.**
- **Height** of a general tree is the length of longest path from root to the leaf of tree.

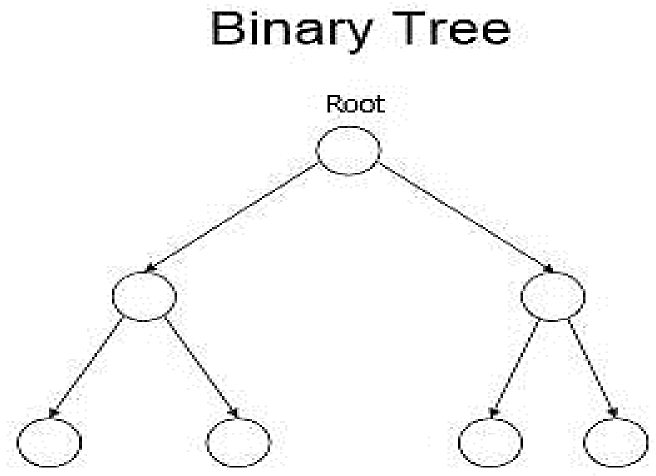
Height(T) = {**max**(height(child1) , height(child2) , ... height(child-n)) +1}

-



Binary tree

- A Binary tree is a data structure in that each node has at most two nodes left and right
- In binary tree, root has in-degree 0 and maximum out-degree 2.
- In binary tree, each node have in-degree one and maximum out-degree 2.
- Height of a binary tree is : Height(T) = { max (Height(Left Child) , Height(Right Child) + 1}
-

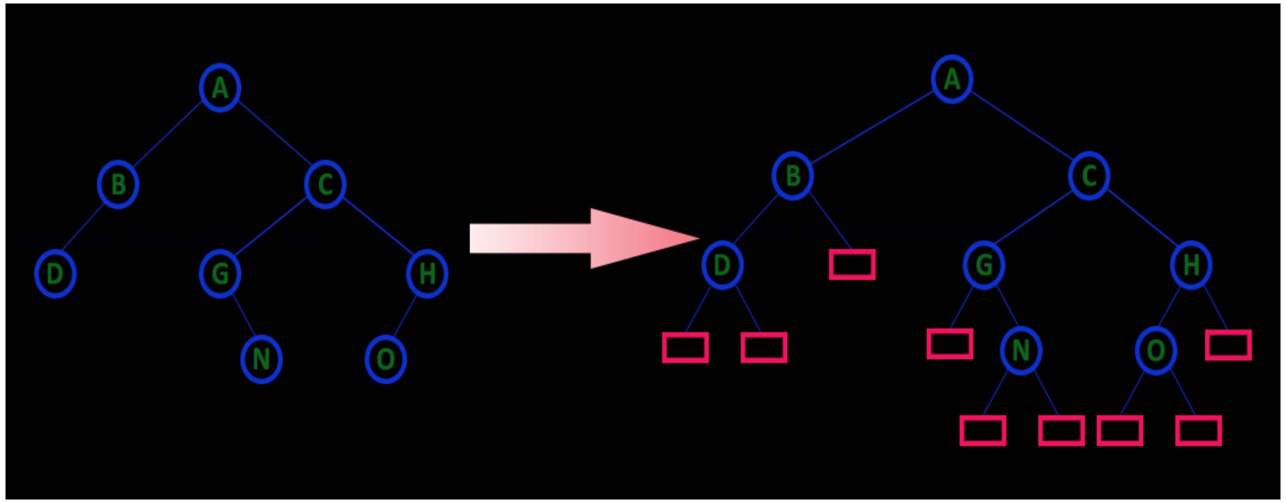


Binary Tree Types

1. Extended Binary Tree
2. Complete Binary Tree
3. Full Binary Tree
4. Skewed Binary Tree
5. Strictly Binary Tree
6. Expression Binary tree

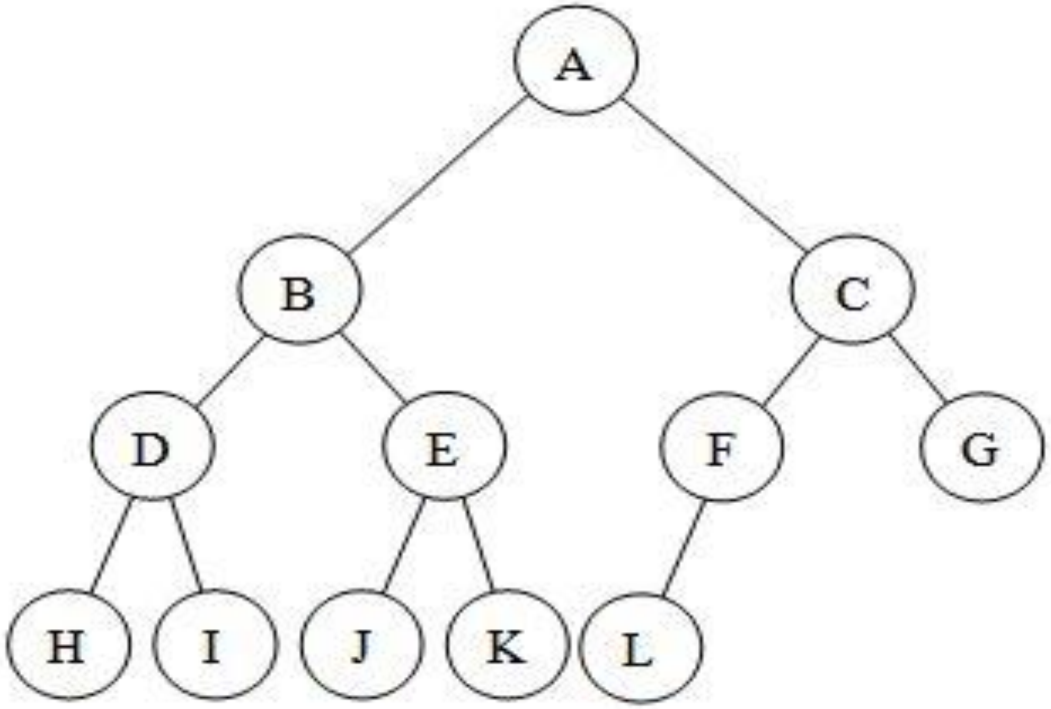
Extended Binary Tree

- An **extended binary tree** is a transformation of any **binary tree** into a complete **binary tree.**
- This transformation consists of replacing every null subtree of the original **tree** with “special nodes” or “failure nodes” .
- The nodes from the original **tree** are then internal nodes, while the “special nodes” are external nodes.



Complete Binary Tree

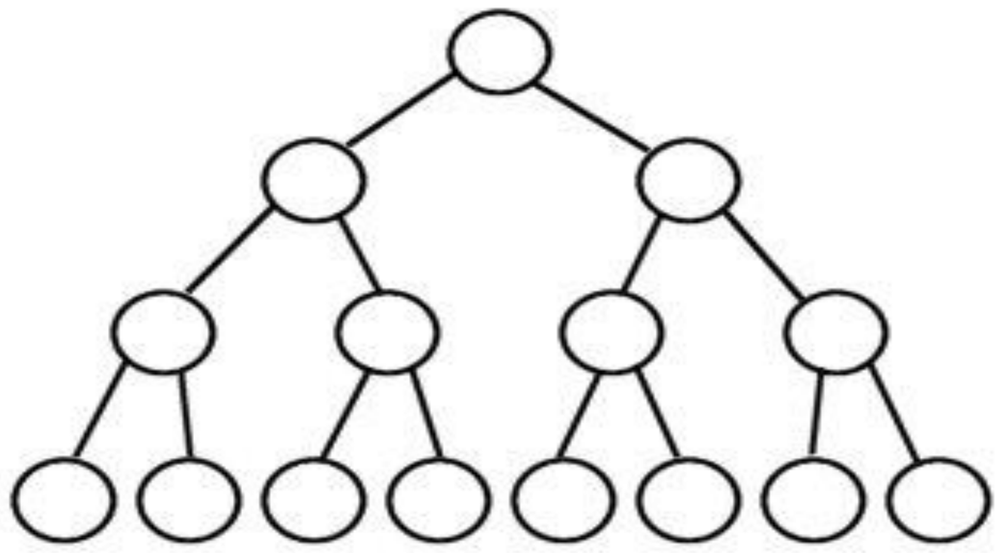
- A complete binary tree is a tree in which
- All leaf nodes are at n or n-1   level
- Levels are filled from left to right



Full Binary Tree

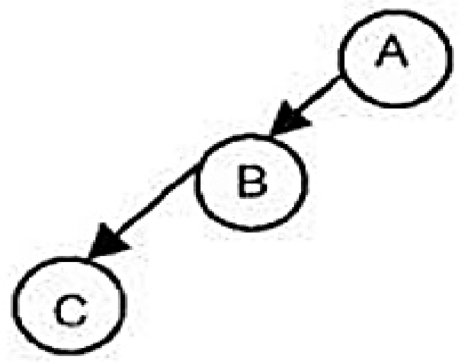
- A **full binary tree** (sometimes proper **binary tree** or 2-**tree**) is a **tree** in which every node other than the leaves has two children or no children. Every level is completely filled up.
- No of nodes= $2^{h+1} - 1$

Full Binary Tree

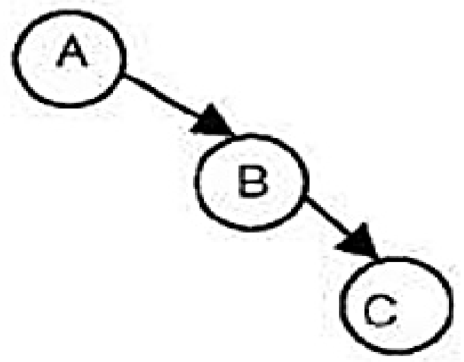


Skewed Binary Tree

- A binary tree is said to be *Skewed Binary Tree* if every node in the tree contains either only left or only right sub tree.
- If the node contains only left sub tree then it is called *left-skewed binary tree* and if the tree contains only right sub tree then it is called *right-skewed binary tree.*



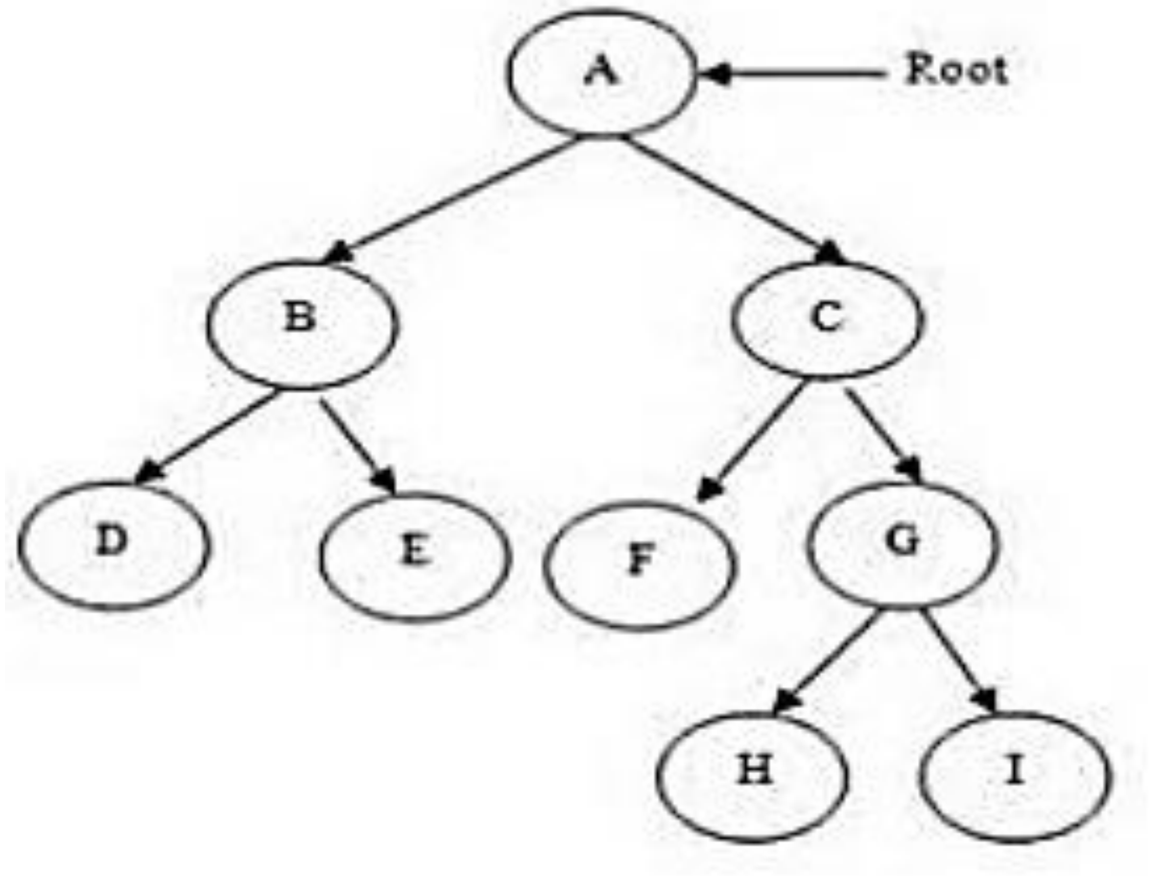
Left Skewed



Right Skewed

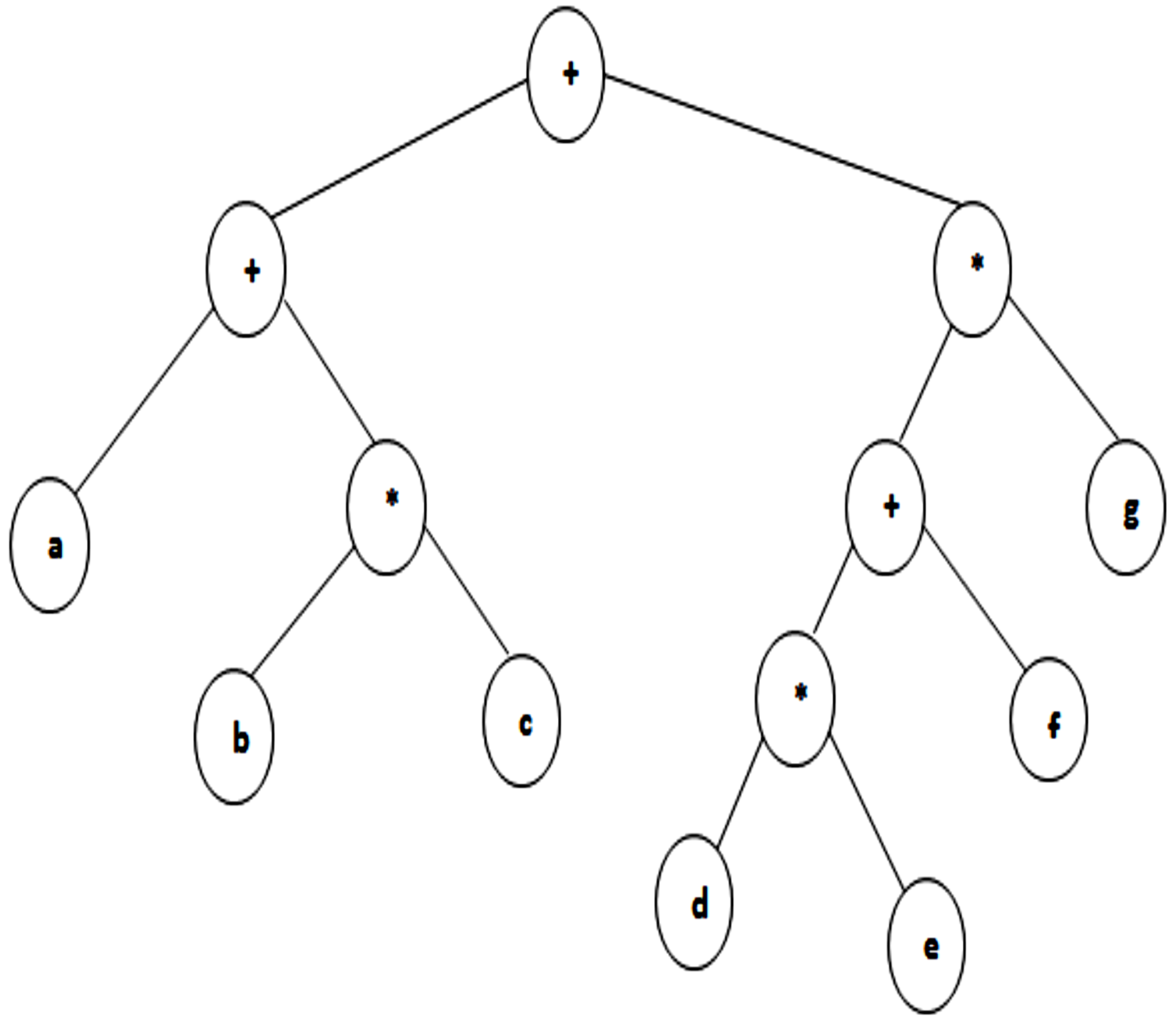
Strictly Binary Tree

- A node will have either two children or no child at all.



Expression Binary tree

- Expression trees are a special kind of binary tree used to evaluate certain expressions.
- Two common types of expressions that a binary expression tree can represent are algebraic and Boolean.
- These trees can represent expressions that contain both unary and binary operators.
- The leaves of a binary expression tree are operands, such as constants or variable names, and the other nodes contain operators.
- Expression tree are used in most compilers.


+
+
*
a
*
b
c
+
g
*
f
d
e

Binary Tree Traversal

1.Preorder traversal:- In this traversal method first process root element, then left sub tree and then right sub tree.

Procedure:-

Step 1: Visit root node

Step 2: Visit left sub tree in preorder

Step 3: Visit right sub tree in preorder

Binary Tree Traversal

<u>**2. Inorder traversal:-**</u> In this traversal method first process left element, then root element and then the right element.

Procedure:-

Step 1: Visit left sub tree in inorder

Step 2: Visit root node

Step 3: Visit right sub tree in inorder

Binary Tree Traversal

<u>**3. Postorder traversal:-**</u> In this traversal first visit / process left sub tree, then right sub tree and then the root element.

Procedure:-

Step 1: Visit left sub tree in postorder

Step 2: Visit right sub tree in postorder

Step 3: Visit root node

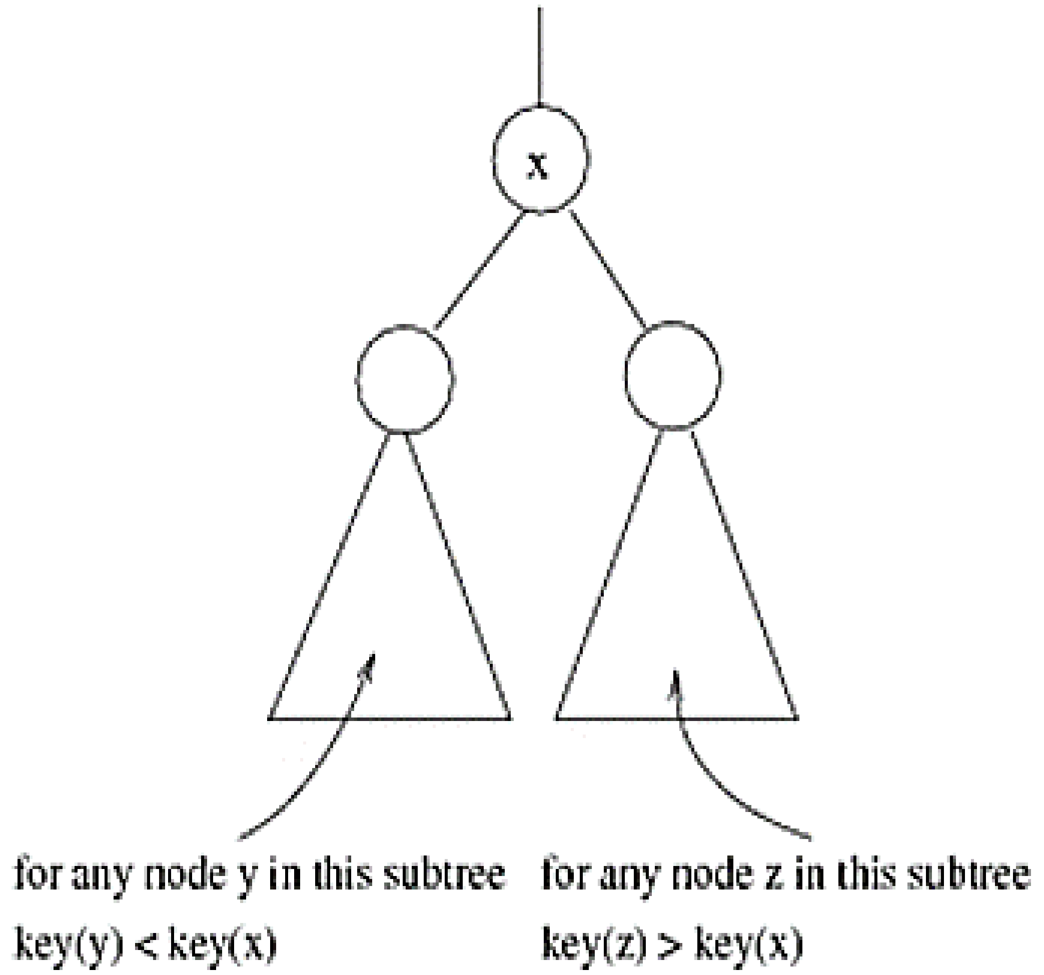
Binary Search Tree

- A **binary search tree** (**BST**) or "ordered **binary tree**" is a empty or in which each node contains a key that satisfies following conditions:
    - All keys are distinct.’
    - All elements in its left subtree are less to the node (<), **and** all the elements in its right subtree are greater than the node (>).

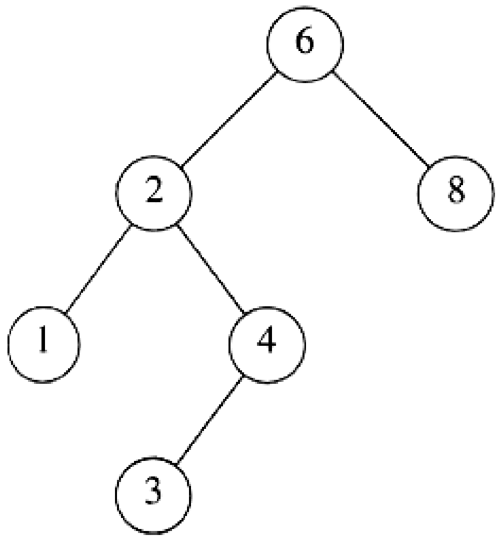
Binary search tree

Binary search tree property

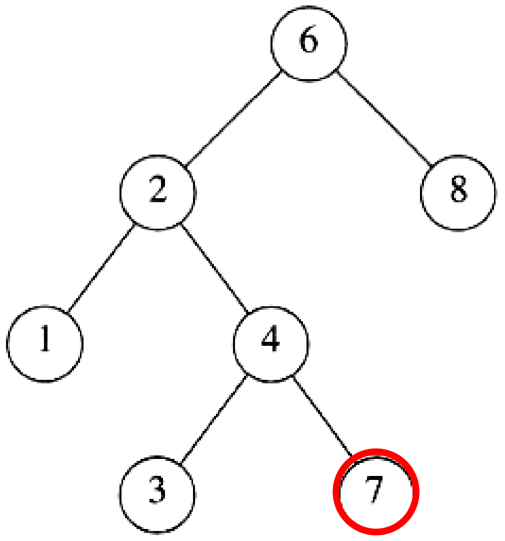
- For every node X
  - All the keys in its left subtree are smaller than the key value in X
  - All the keys in its right subtree are larger than the key value in X


x
for any node y in this subtree
key(y) < key(x)
for any node z in this subtree
key(z) > key(x)

Binary Search Trees



A binary search tree



Not a binary search tree

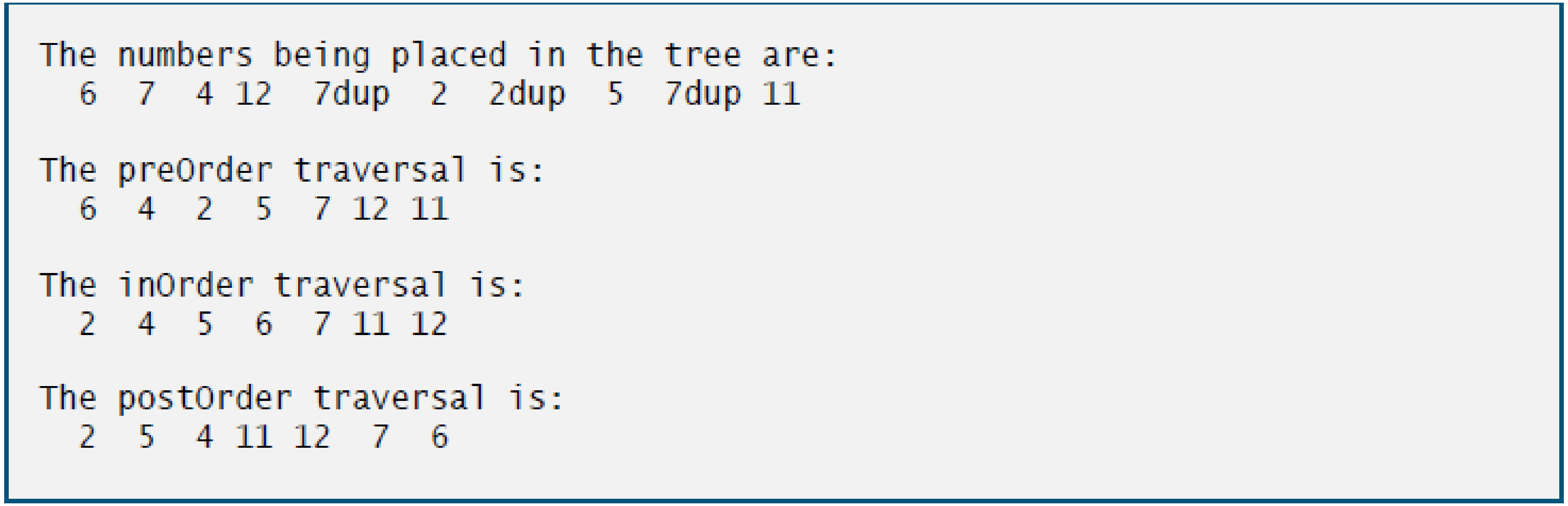
For more detail contact us

Linked List Implementation of Trees

```c
// Fig. 12.19: fig12_19.c
// Creating and traversing a binary tree
// preorder, inorder, and postorder
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// self-referential structure
struct treeNode {
   struct treeNode *leftPtr; // pointer to left subtree
   int data; // node value
   struct treeNode *rightPtr; // pointer to right subtree
};

typedef struct treeNode TreeNode; // synonym for struct treeNode
typedef TreeNode *TreeNodePtr; // synonym for TreeNode*

// prototypes
void insertNode(TreeNodePtr *treePtr, int value);
void inOrder(TreeNodePtr treePtr);
void preOrder(TreeNodePtr treePtr);
void postOrder(TreeNodePtr treePtr);

// function main begins program execution
int main(void)
{
   TreeNodePtr rootPtr = NULL; // tree initially empty

   srand(time(NULL));
   puts("The numbers being placed in the tree are:");

   // insert random values between 0 and 14 in the tree
   for (unsigned int i = 1; i <= 10; ++i) {
      int item = rand() % 15;
      printf("%3d", item);
      insertNode(&rootPtr, item);
   }

   // traverse the tree preOrder
   puts("\n\nThe preOrder traversal is:");
   preOrder(rootPtr);

   // traverse the tree inOrder
   puts("\n\nThe inOrder traversal is:");
   inOrder(rootPtr);

   // traverse the tree postOrder
   puts("\n\nThe postOrder traversal is:");
   postOrder(rootPtr);
}
```

```c
// insert node into tree
void insertNode(TreeNodePtr *treePtr, int value)
{
   // if tree is empty
   if (*treePtr == NULL) {
      *treePtr = malloc(sizeof(TreeNode));

      // if memory was allocated, then assign data
      if (*treePtr != NULL) {
         (*treePtr)->data = value;
         (*treePtr)->leftPtr = NULL;
         (*treePtr)->rightPtr = NULL;
      }
      else {
         printf("%d not inserted. No memory available.\n", value);
      }
   }
   else { // tree is not empty
      // data to insert is less than data in current node
      if (value < (*treePtr)->data) {
         insertNode(&((*treePtr)->leftPtr), value);
      }

      // data to insert is greater than data in current node
      else if (value > (*treePtr)->data) {
         insertNode(&((*treePtr)->rightPtr), value);
      }
      else { // duplicate data value ignored
         printf("%s", "dup");
      }
   }
}

// begin inorder traversal of tree
void inOrder(TreeNodePtr treePtr)
{
   // if tree is not empty, then traverse
   if (treePtr != NULL) {
      inOrder(treePtr->leftPtr);
      printf("%3d", treePtr->data);
      inOrder(treePtr->rightPtr);
   }
}

// begin preorder traversal of tree
void preOrder(TreeNodePtr treePtr)
{
   // if tree is not empty, then traverse
   if (treePtr != NULL) {
      printf("%3d", treePtr->data);
      preOrder(treePtr->leftPtr);
      preOrder(treePtr->rightPtr);
   }
}

// begin postorder traversal of tree
void postOrder(TreeNodePtr treePtr)
{
   // if tree is not empty, then traverse
   if (treePtr != NULL) {
      postOrder(treePtr->leftPtr);
      postOrder(treePtr->rightPtr);
      printf("%3d", treePtr->data);
   }
}
```

Code Output



```
The numbers being placed in the tree are:
  6  7  4 12  7dup  2  2dup  5  7dup 11

The preOrder traversal is:
  6  4  2  5  7 12 11

The inOrder traversal is:
  2  4  5  6  7 11 12

The postOrder traversal is:
  2  5  4 11 12  7  6
```