UNIT II

TREES

Institute & Department Vision Mission

Institute Vision :

"Satisfy the aspirations of youth force, who want to lead nation towards prosperity through techno-economic development."

Institute Mission :

"To provide, nurture and maintain an environment of high academics excellence, research and entrepreneurship for all aspiring students, which will prepare them to face global challenges maintaining high ethical and moral standards."

Departmental Vision :

"Empowering the students to be professionally competent & socially responsible for techno- economic development of society."

Departmental Mission :

A: To provide quality education enabling students for higher studies, research and entrepreneurship

B: To inculcate professionalism and ethical values through day to day practices.

Objectives

- Study how trees are used to represent data in hierarchical manner

- Define Binary Search Trees (BST), that allow both rapid retrieval by key and inorder traversal

- Learn use of trees as a flexible data structure for solving a wide range of problems

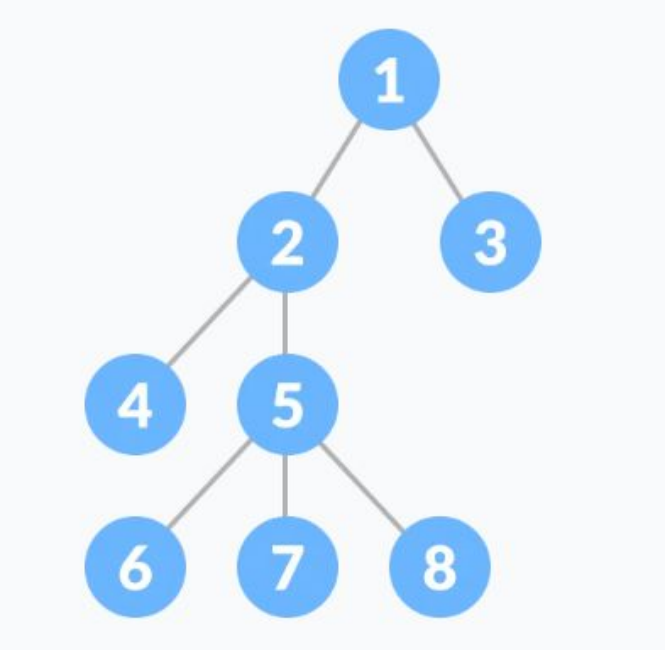
Topics

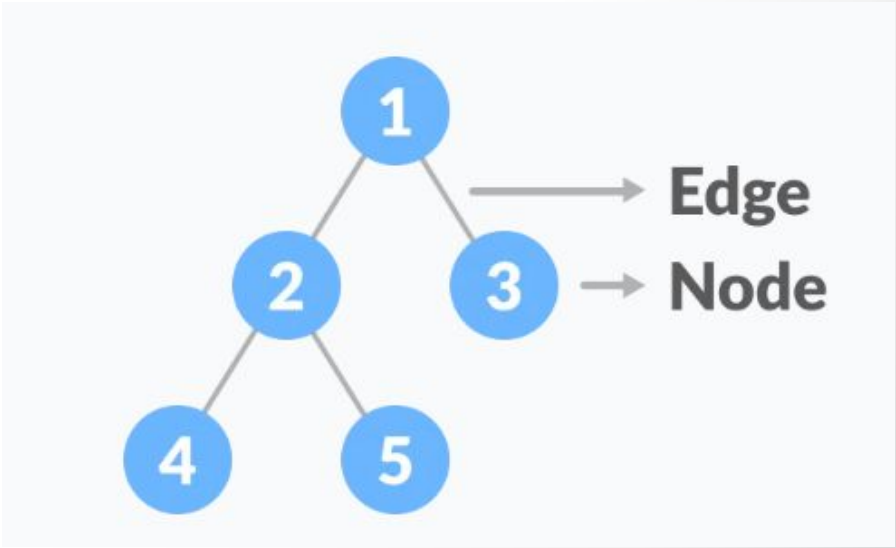
- **Tree- basic terminology, General tree and its representation, representation using sequential and linked organization, Binary tree- properties, converting tree to binary tree, binary tree traversals(recursive and non-recursive)- inorder, preorder, post order, depth first and breadth first, Operations on binary tree.** Huffman Tree (Concept and Use), **Binary Search Tree (BST), BST operations,** Threaded binary search tree- concepts, threading, insertion and deletion of nodes in in-order threaded binary search tree, in order traversal of in-order threaded binary search tree.

TERMINOLOGIES IN TREE

- **"A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges."**

- Tree can be used to maintain and manipulate data in many applications

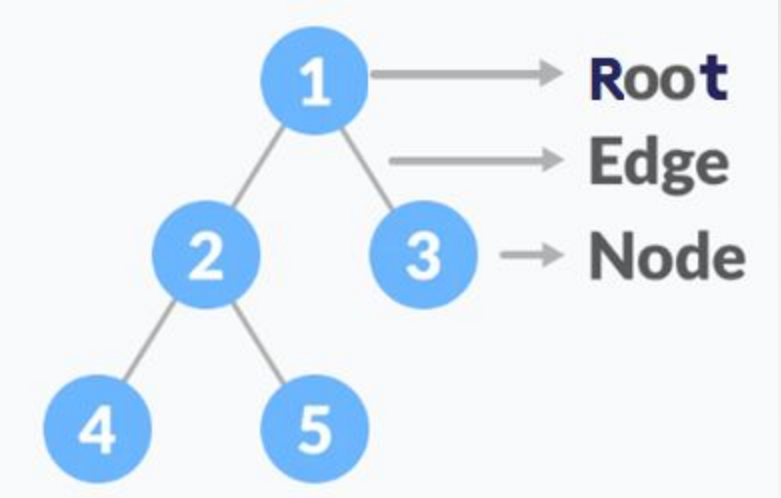




TERMINOLOGIES IN TREE

- **Root :**

It is the topmost node of a tree.



- **Edge :**

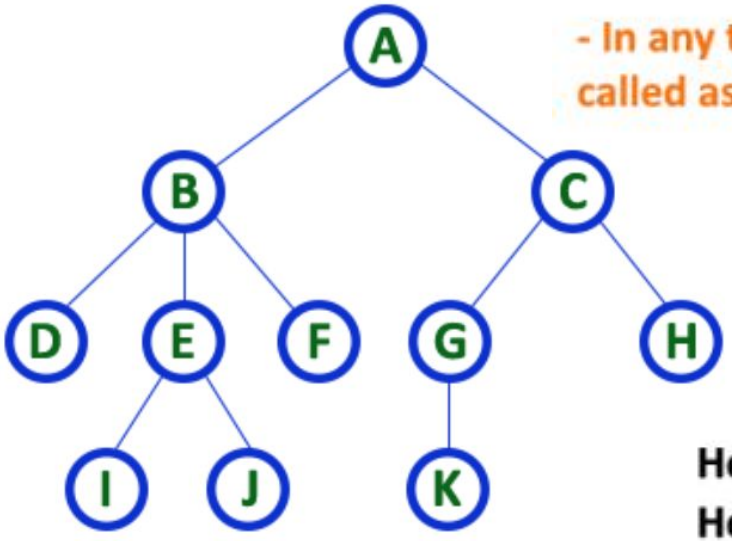
It is the link between any two nodes.

- **Node :**

A node is an entity that contains a key or value and pointers to its child nodes.

Here 'A' is the 'root' node

- In any tree the first node is called as ROOT node



Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

- descendant of any node is called as CHILD Node

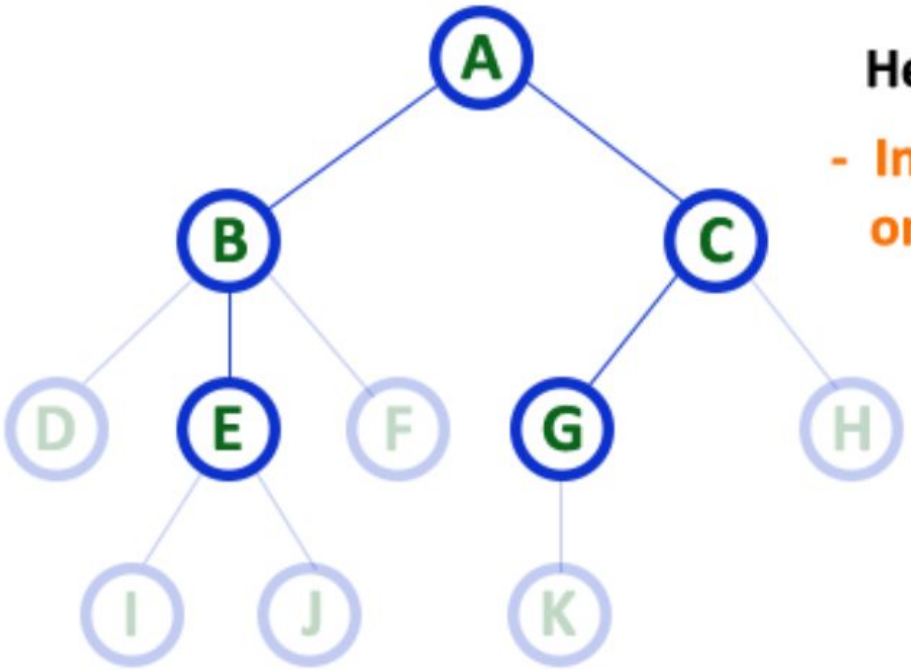
Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.

In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.

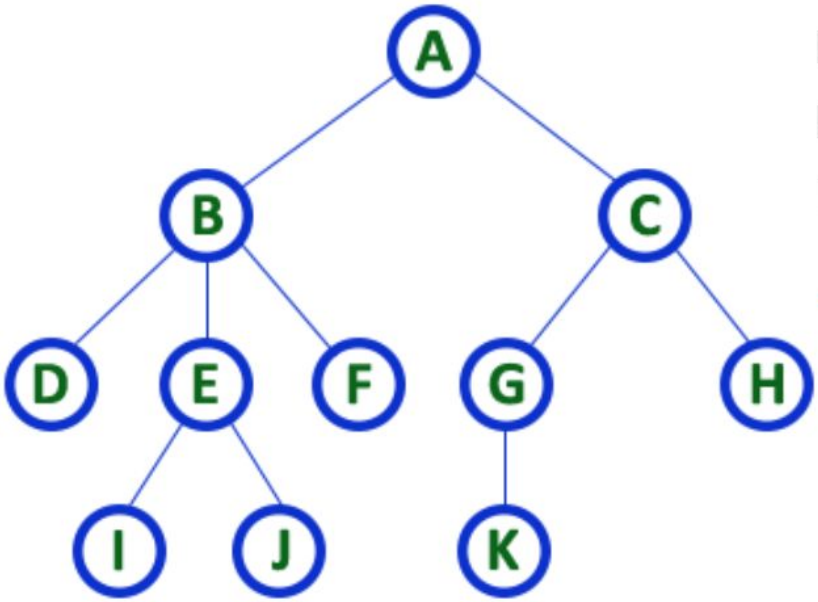


Here A, B, C, E & G are Internal nodes

- **In any tree the node which has atleast one child is called 'Internal' node**
- **Every non-leaf node is called as 'Internal' node**

8. Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as **'Degree of Tree'**



Here Degree of B is 3

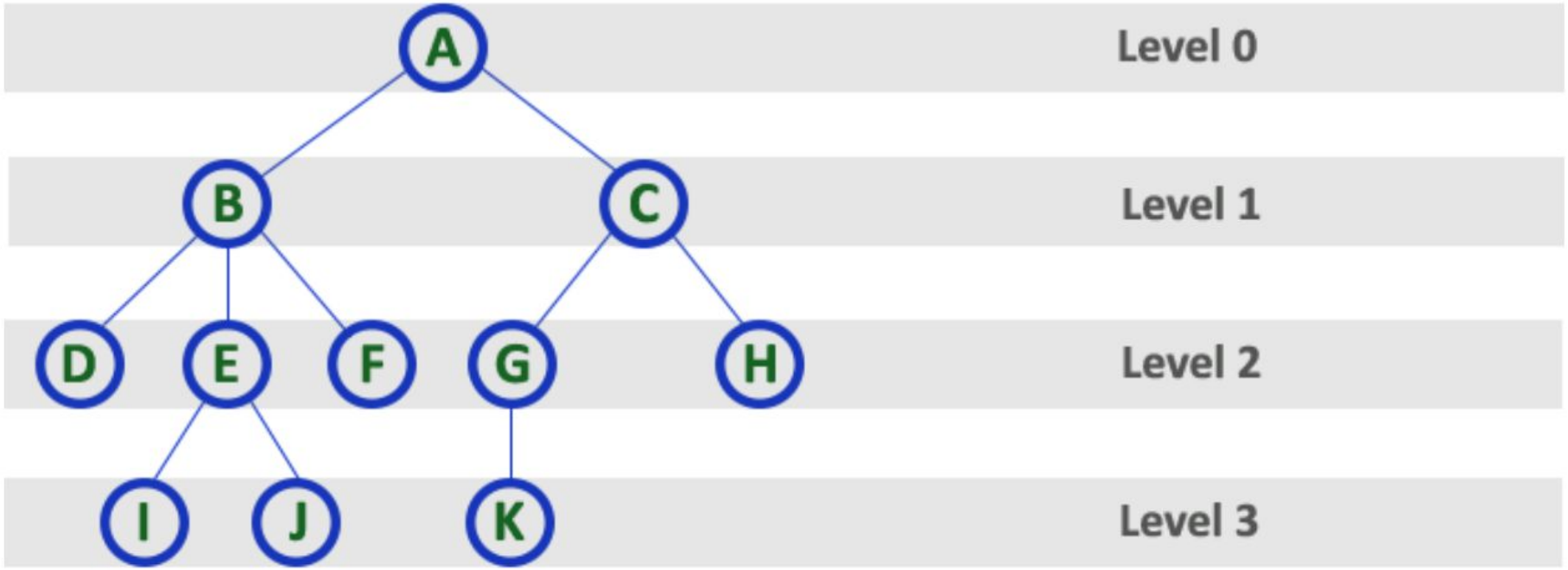
Here Degree of A is 2

Here Degree of F is 0

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

9. Level

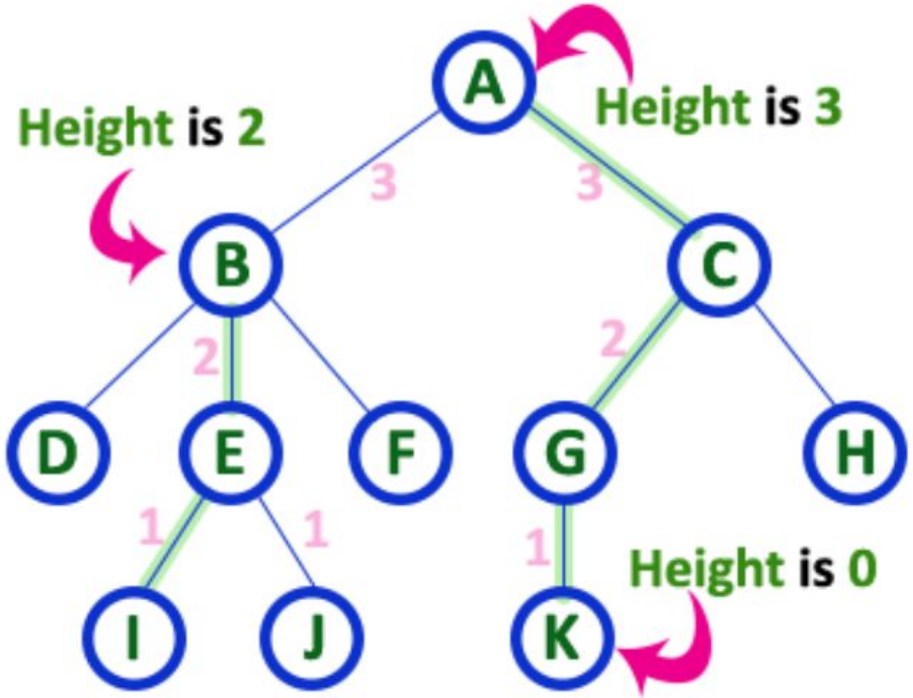
In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



10. Height

In a tree data structure, the total number of edges from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node.

In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**

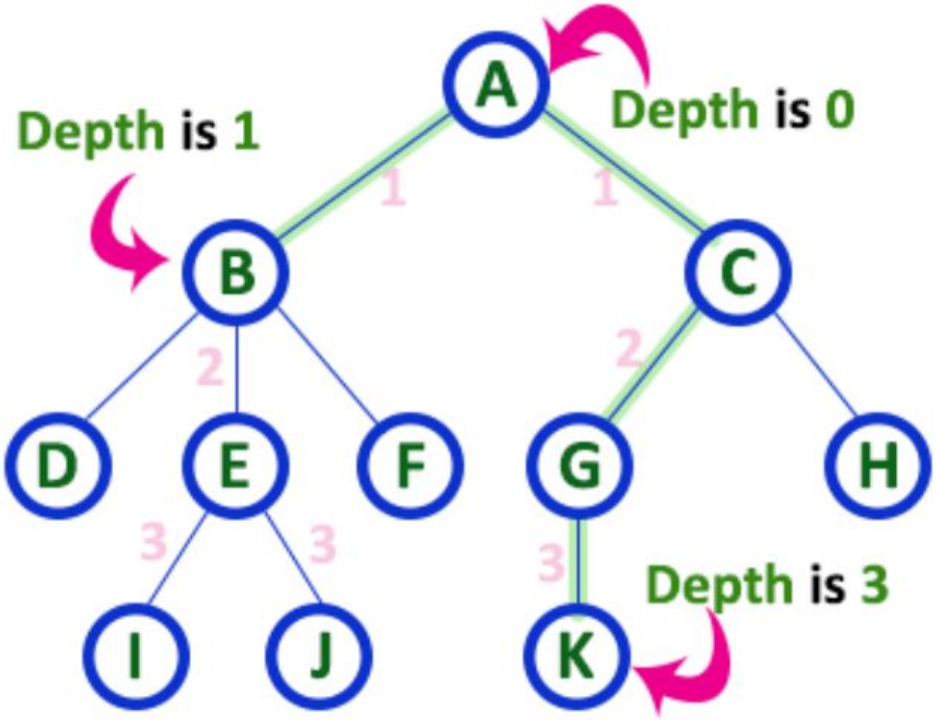


Here Height of tree is 3

- **In any tree, 'Height of Node' is total number of Edges from leaf to that node in longest path.**
- **In any tree, 'Height of Tree' is the height of the root node.**

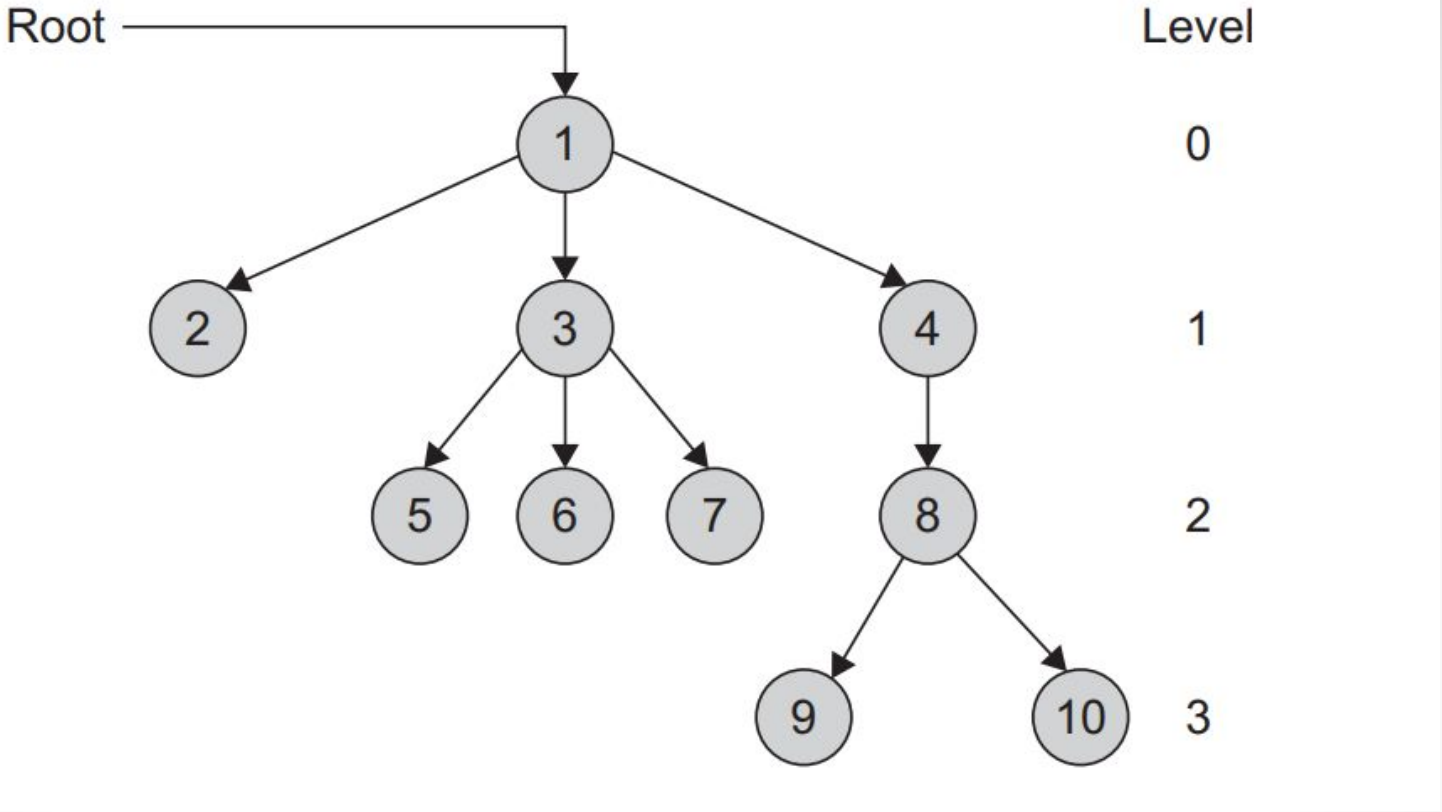
11. Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



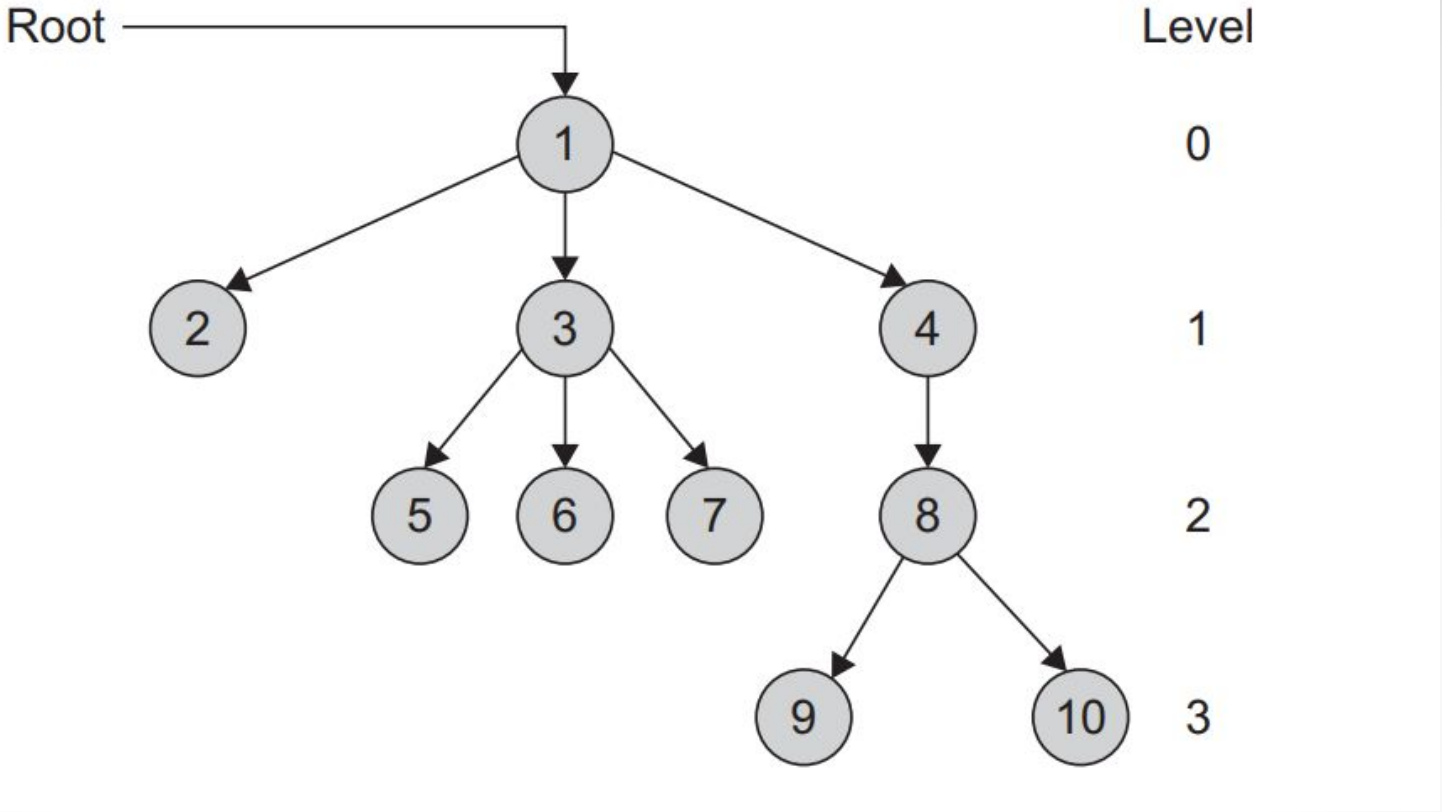
Here Depth of tree is 3

- **In any tree, 'Depth of Node' is total number of Edges from root to that node.**
- **In any tree, 'Depth of Tree' is total number of edges from root to leaf in the longest path.**



Height of the Tree =
Depth of the Node 5 =
Depth of the Node 6 =
Depth of the Node 7 =
Depth of the Node 8 =
Depth of the Node 9 =
Depth of the Node 10 =

The depth of nodes on the lowest level is always the same as height of the Tree



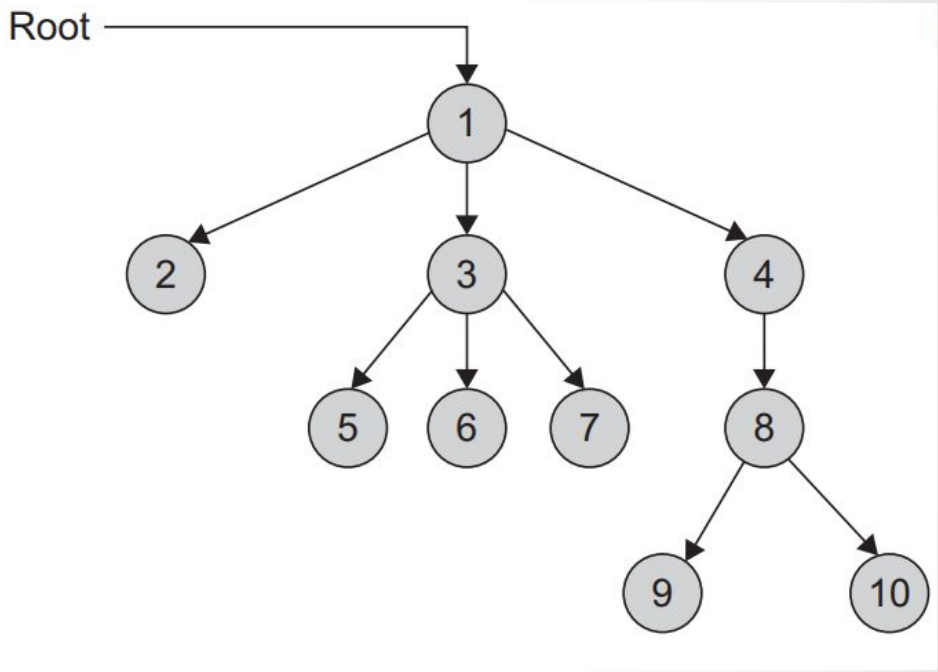
Height of the Tree = 3
Depth of the Node 5 = 2
Depth of the Node 6 = 2
Depth of the Node 7 = 2
Depth of the Node 8 = 2
Depth of the Node 9 = 3
Depth of the Node 10 = 3

The depth of nodes on the lowest level is always the same as height of the Tree

Paths & Path length

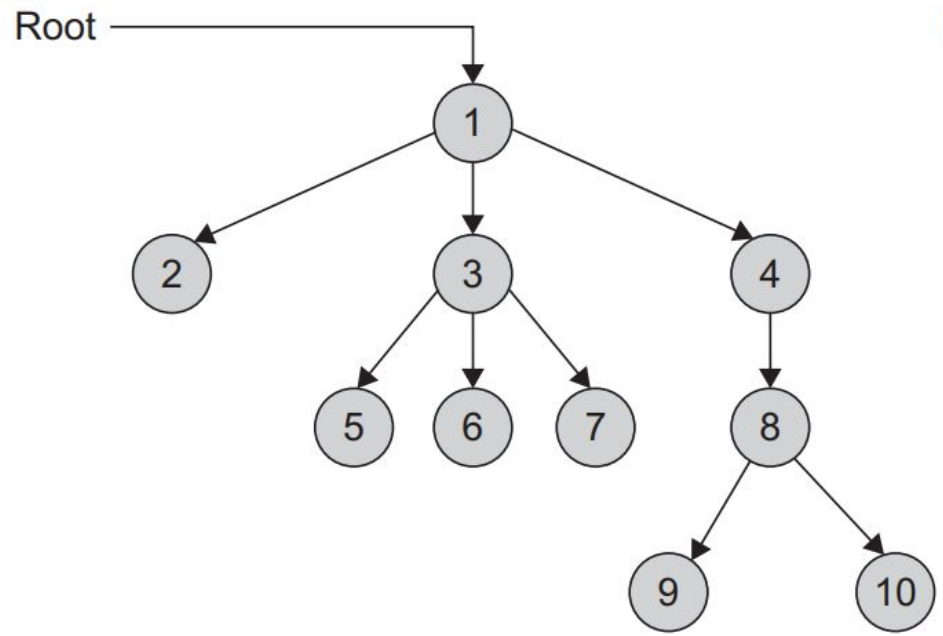
Paths exist from all parents to children. A unique path exists from the root to each leaf node as shown

$1 \rightarrow 2$	Length: 1
$1 \rightarrow 3 \rightarrow 5$	Length: 2
$1 \rightarrow 3 \rightarrow 6$	Length: 2
$1 \rightarrow 3 \rightarrow 7$	Length: 2
$1 \rightarrow 4 \rightarrow 8 \rightarrow 9$	Length: 3
$1 \rightarrow 4 \rightarrow 8 \rightarrow 10$	Length: 3



Orderings

The preorder, inorder, and postorder orderings of the nodes are given in the following sequence:



$1 \rightarrow 2 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 4 \rightarrow 8 \rightarrow 9 \rightarrow 10$ (preorder)

$2 \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 9 \rightarrow 8 \rightarrow 10 \rightarrow 4$ (inorder)

$2 \rightarrow 5 \rightarrow 6 \rightarrow 7 \rightarrow 3 \rightarrow 9 \rightarrow 10 \rightarrow 8 \rightarrow 4 \rightarrow 1$ (postorder)

Terminal node (Leaf node) : In a directed tree, any node which has out degree zero is a terminal node

Terminal node is also called as leaf node (or external node)

Branch node (Internal node) : All other nodes whose outdegree is not zero are called as branch nodes

Level of node : The level of any node is the path length of it from the root.

- **The** level of root of a directed tree is 0, while the level of any node is equal to its distance from the root
- Distance from the root is the number of edges to be traversed to reach the root

Trees

1. A class of graphs which that are acyclic are termed as trees
2. Trees are useful in describing any structure which that involves hierarchy
3. Familiar examples of such structures are family trees, the hierarchy of positions in organization, etc

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.

Forest and Trees

- In non-linear data structures, every data element may have more than one predecessor as well as successor
- Elements do not form any particular linear sequence
- A forest is a graph that contains no cycles and a connected forest is a tree
- **Forest :**A collection of disjoint trees is called a forest.

You can create a forest by cutting the root of a tree.

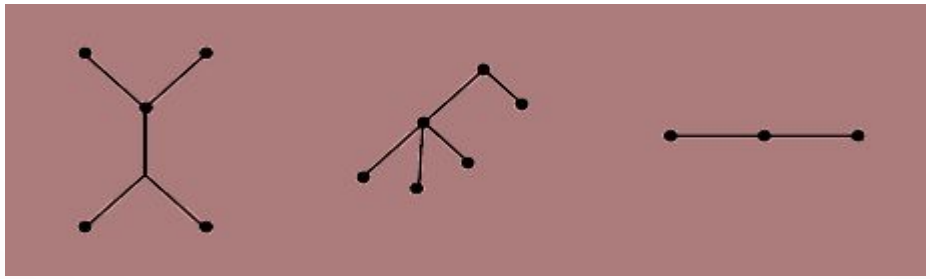


Figure 6: Forest with three trees

Applications of Tree

- **Binary Search Tree (BST)** is used to check whether elements present or not.

- **Heap** is a type of tree that is used to heap sort.
- **Trie** (also known as digital tree) are the modified version of the tree used in modem routing to information of the router.

- The widespread database uses **B-tree**.
- Compilers use **syntax trees** to check every syntax of a program.

Advantages of Tree

1. Trees reflect structural relationships in the data.

2. Trees are used to represent hierarchies.

3. Trees provide an efficient insertion and searching.
4. Treesare very flexible data, allowing to move

   subtrees around with minumum effort.

Types of Tree

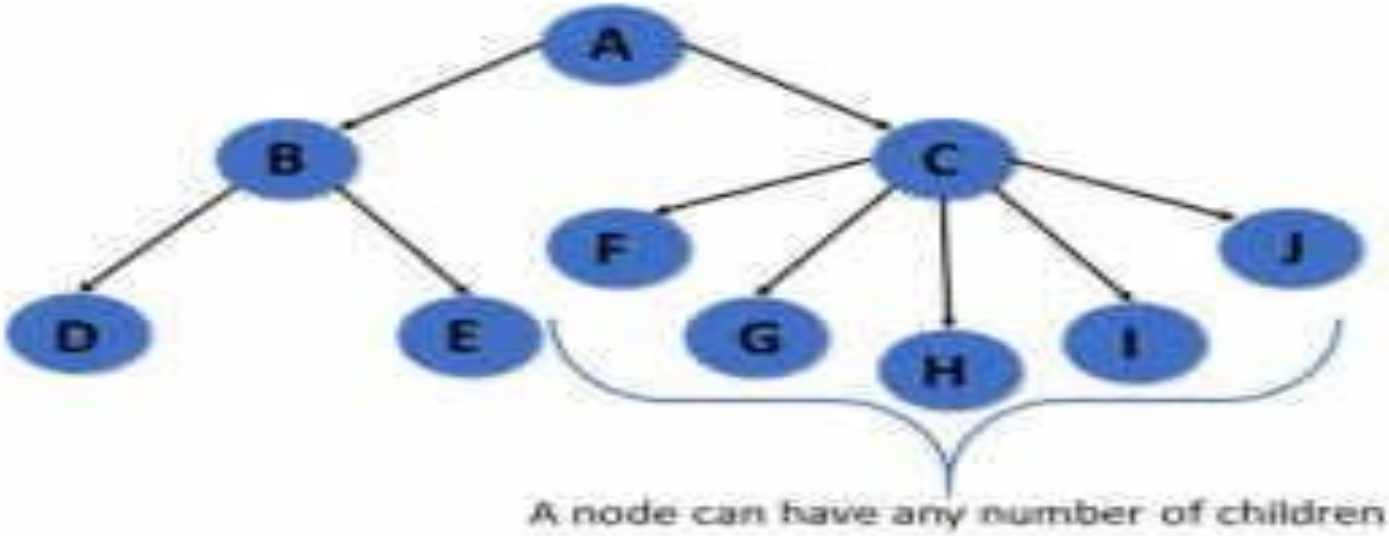
- **General Tree**
- **Binary Tree**
- **Binary Search Tree**
- **AVL Tree**
- **B-Tree**

General Tree

General Tree

- The general tree is the type of tree where there areno constraints on the hierarchical structure.

- **Properties**
- The general tree follows all properties of the tree data structure.
- A node can have any number of nodes.



General Trees

A tree is defined recursively, as follows

- A set of zero items is a tree, called the empty tree (or null tree)
- If T1, T2, - - - -,Tn are n trees for n > 0 and R is node, then the set T containing R and the trees T1, T2, - - - - ,Tn is a tree
- Within T, R is called the root of T and T1, T2, - - - -, Tn are called **subtrees**

Representation of a General Tree

- We can use either sequential organization or linked organization for representing a tree
- If we wish to use generalized linked list, then a node must have varying number of fields depending upon the number of branches
- However, it is simpler to use algorithms for data in which the node size is fixed

Figure 6: Node Structure

- Pseudo code Notations
- For a fixed size node, we can use a node with data and pointer fields as in generalized linked list

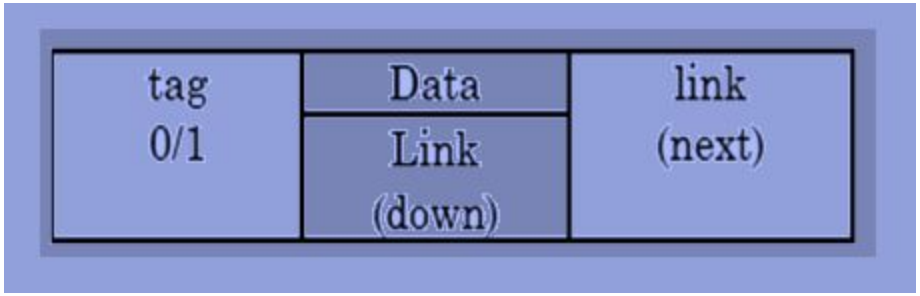


Figure 7: Node Structure in generalized linked list

Sample Tree

- Sequence
- Decision-Tree
- Repetition

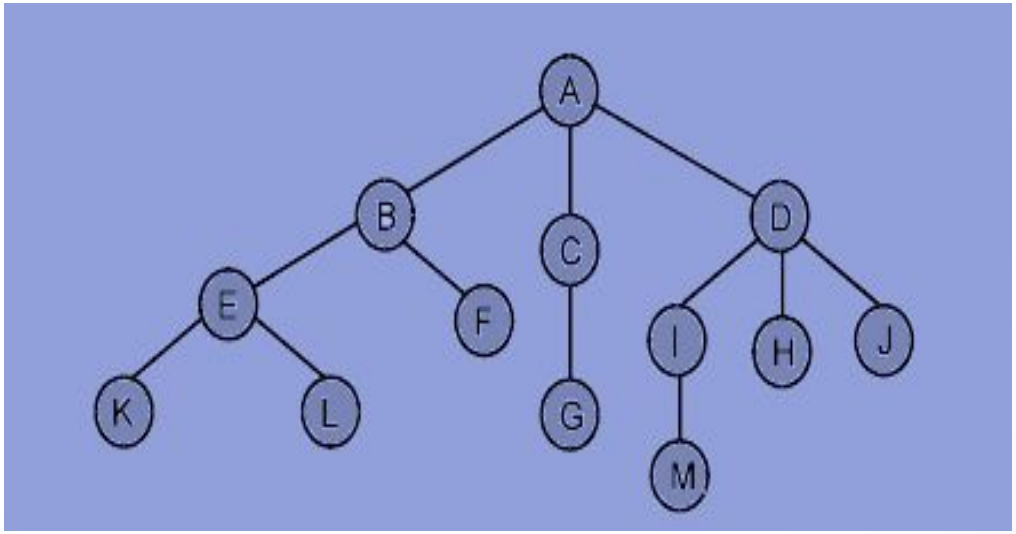


Figure 8: Sample Tree

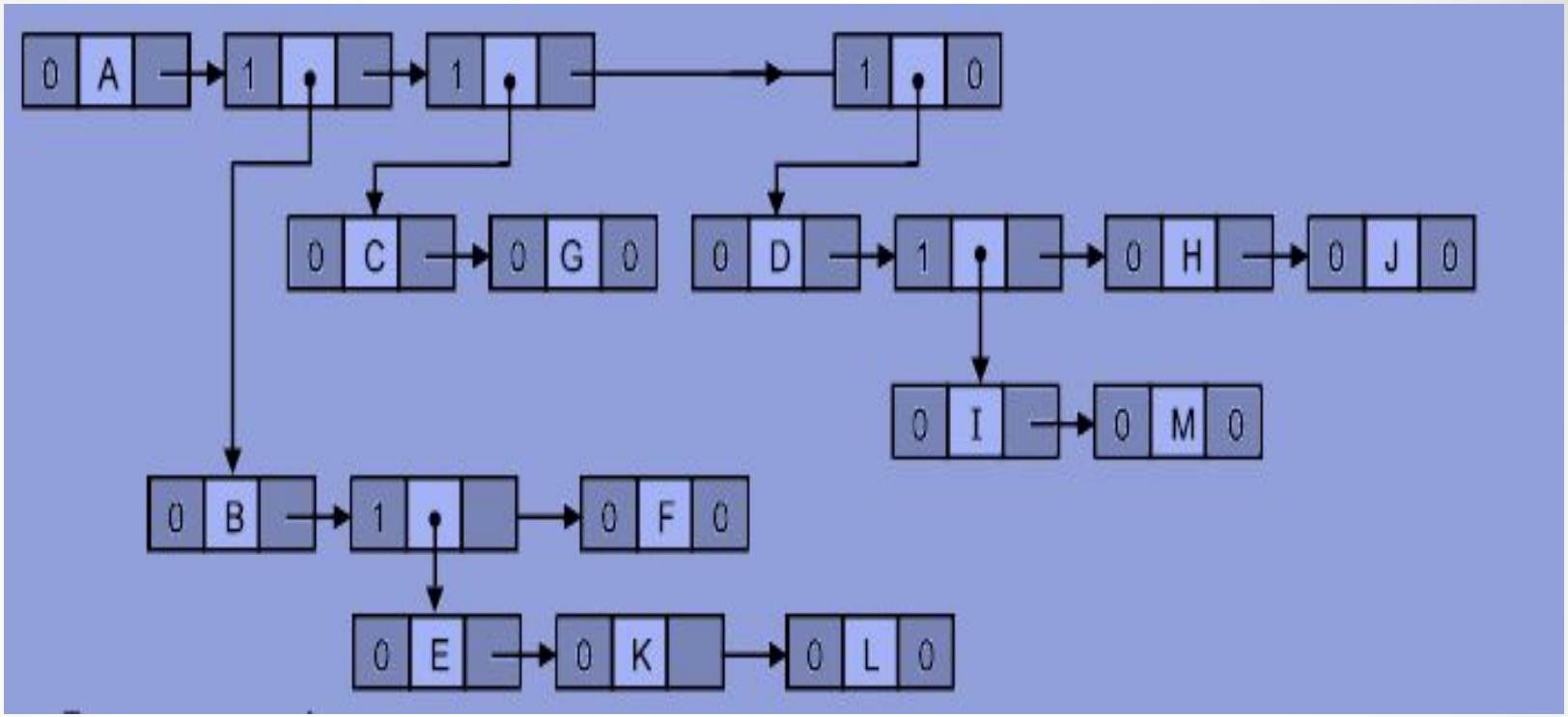
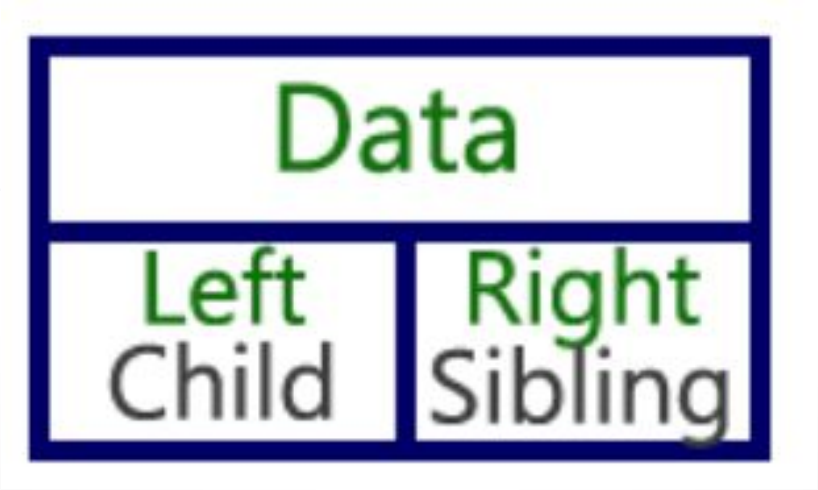


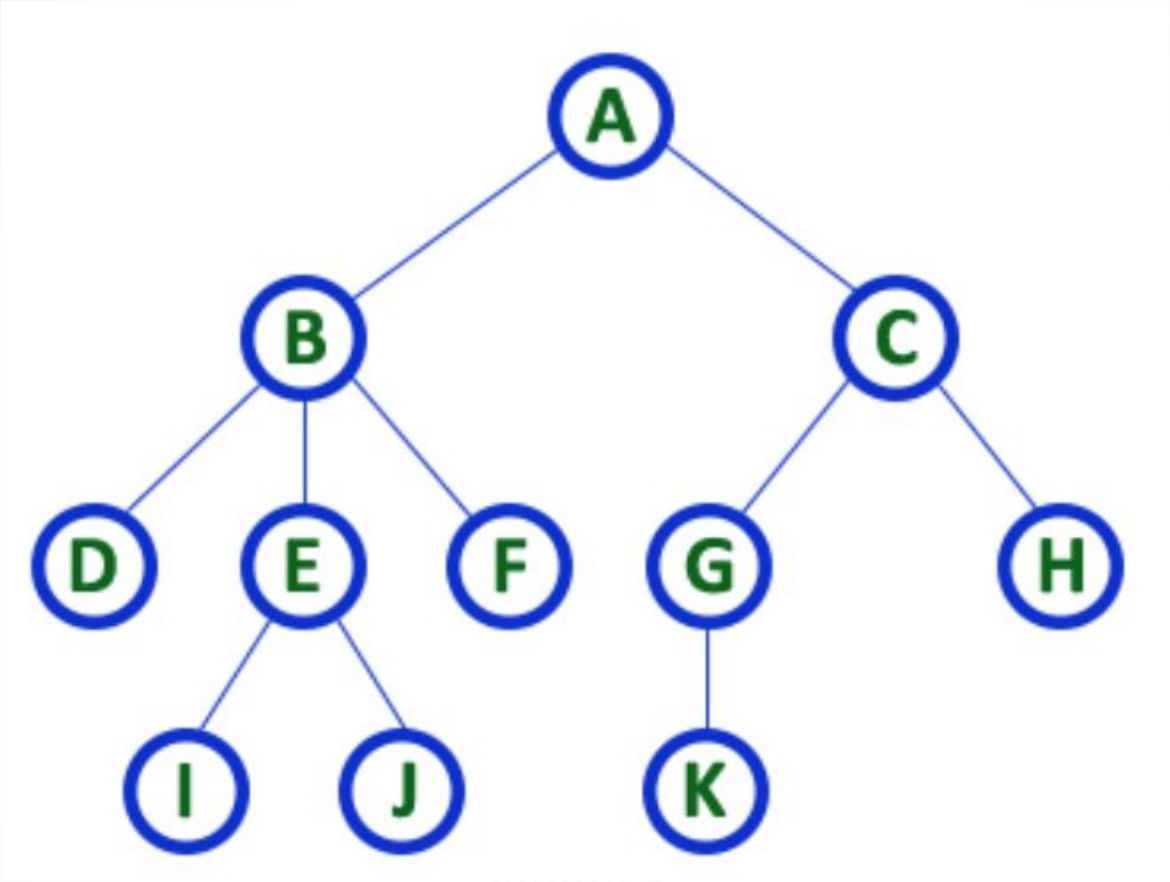
Figure 9: List Representation of Tree in Figure 8

Left child- Right Sibling Representation Method

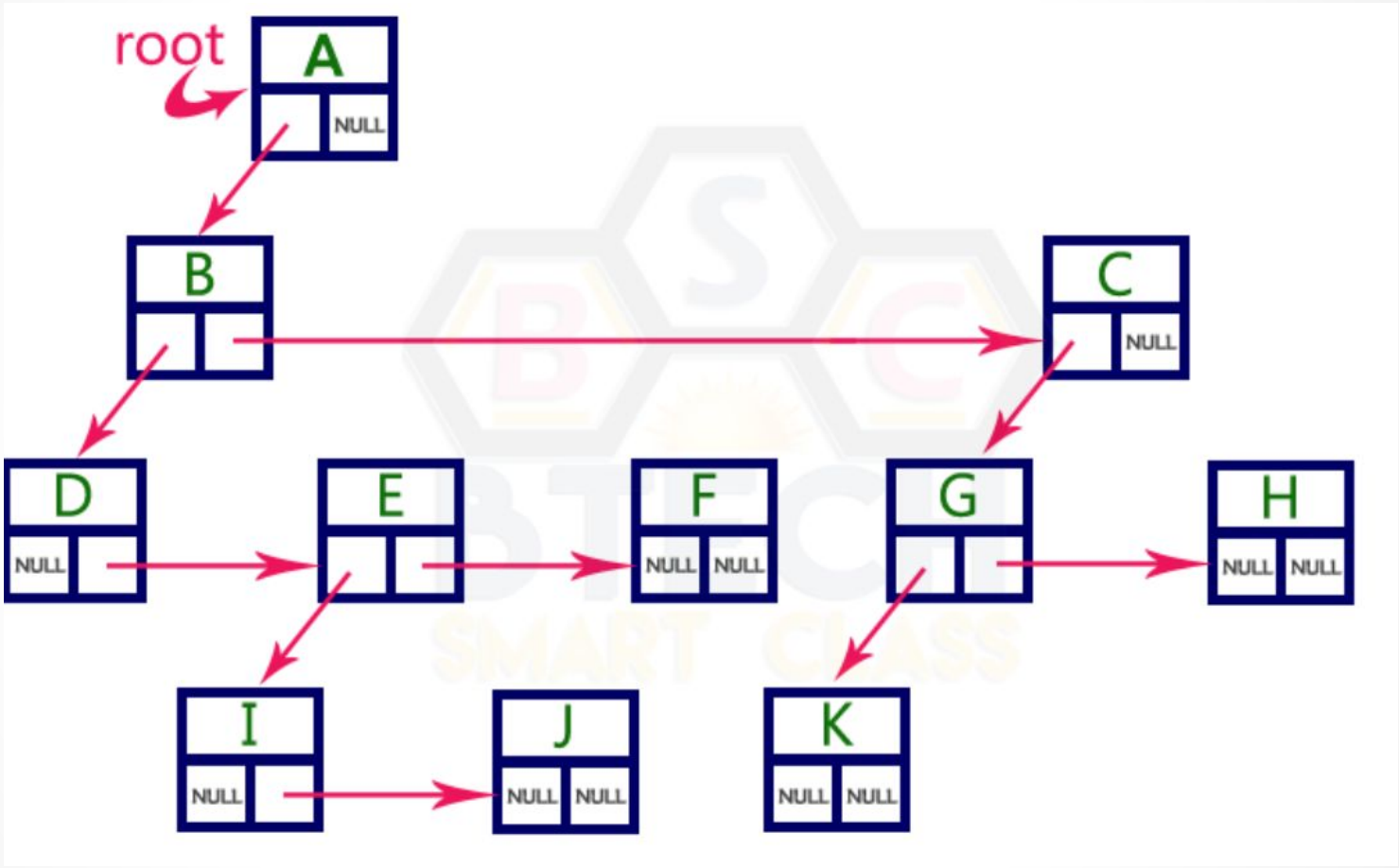
- In this representation, we use a **list** with one type of node which consists of **three fields** namely **Data field, Left child reference field and Right sibling reference field**.
- **Data field** stores the actual value of a node**, left child reference field** stores the address of the left child and **right reference field** stores the address of the right sibling node.



Left child- Right Sibling Method



Left child- Right Sibling Method



Binary Tree Representation Methods
Sequential & Linked List Representation

Figure 9: List Representation of Tree in Figure 8

Sequential & Linked List Representation (Binary Tree)

- Sequential Method uses only a single linear array Tree
- Sequential tree implementations can be used to **serialize** a tree structure.
- Serialization is the process of storing an object as a series of bytes, typically so that the data structure can be transmitted between computers.
- This capability is important when using data structures in a distributed processing environment.

Figure 9: List Representation of Tree in Figure 8

Sequential & Linked List Representation (Binary Tree)

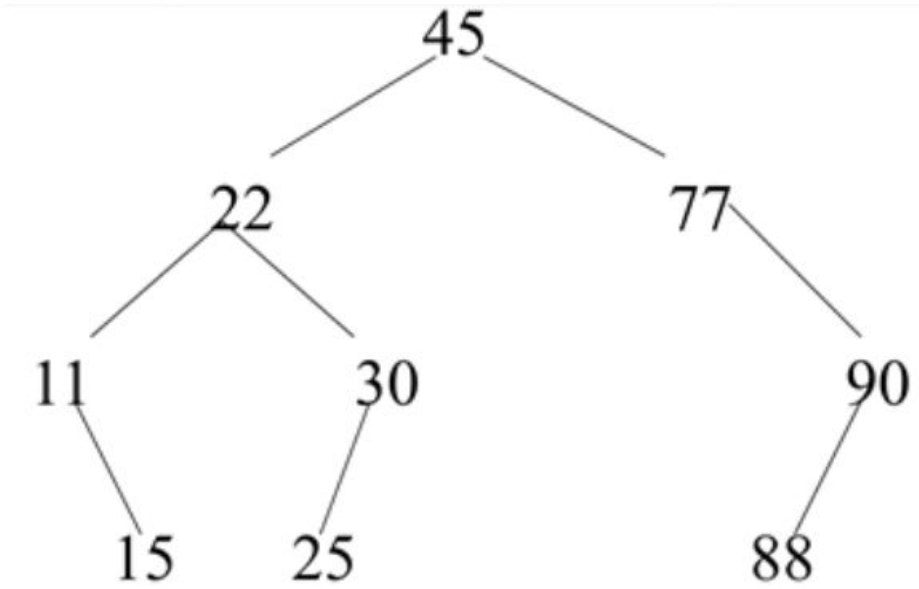
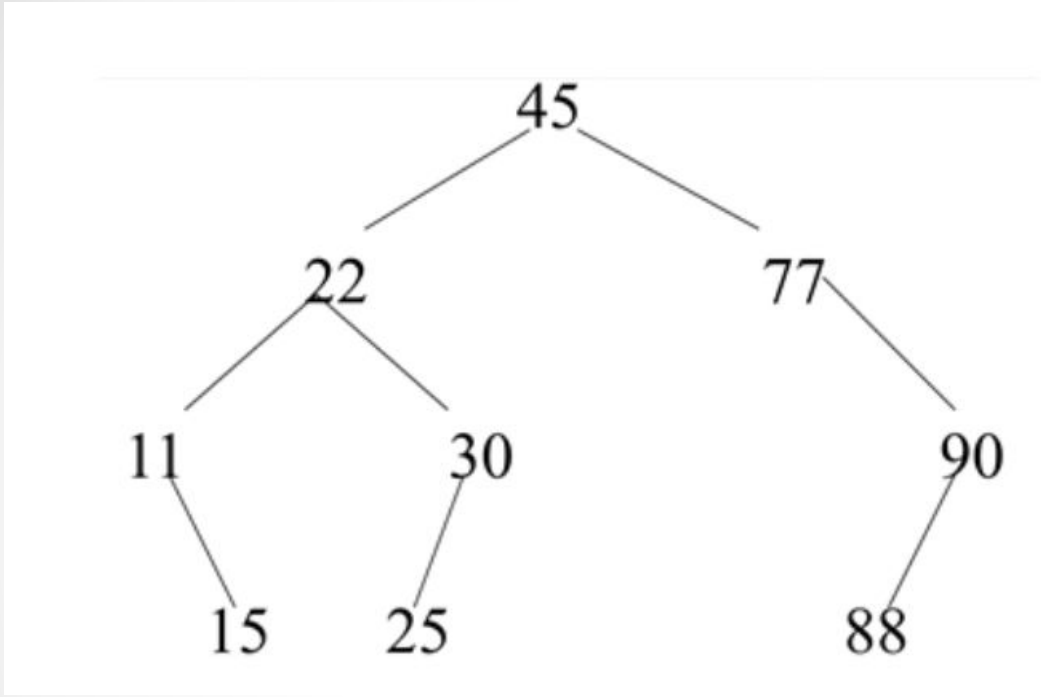


Figure 9: List Representation of Tree in Figure 8

Sequential & Linked List Representation (Binary Tree)



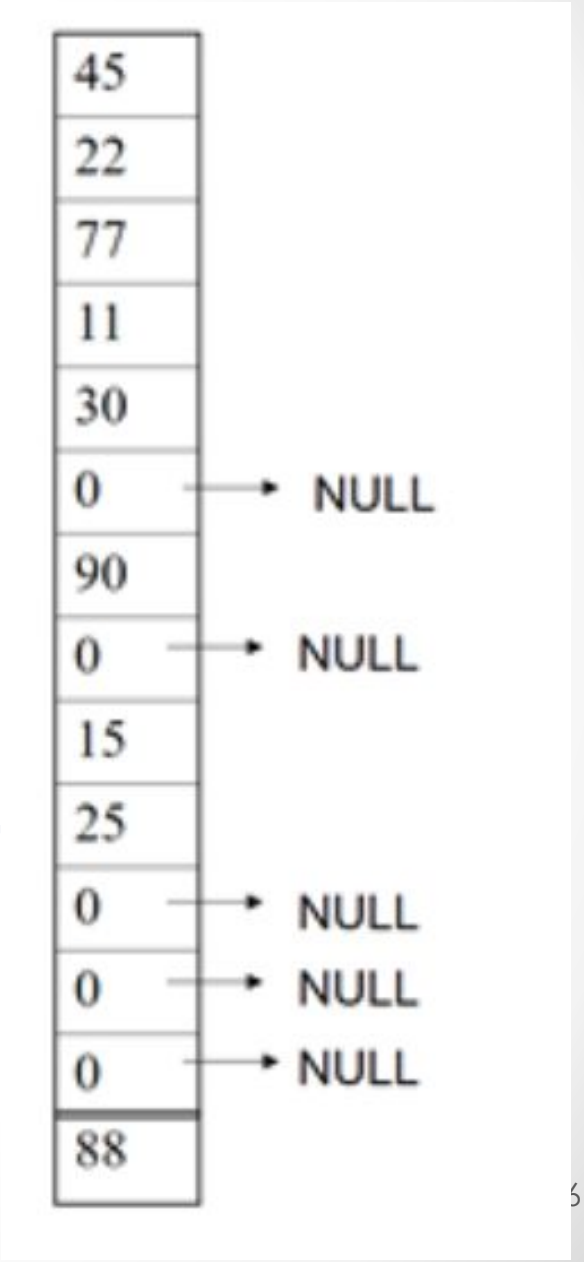
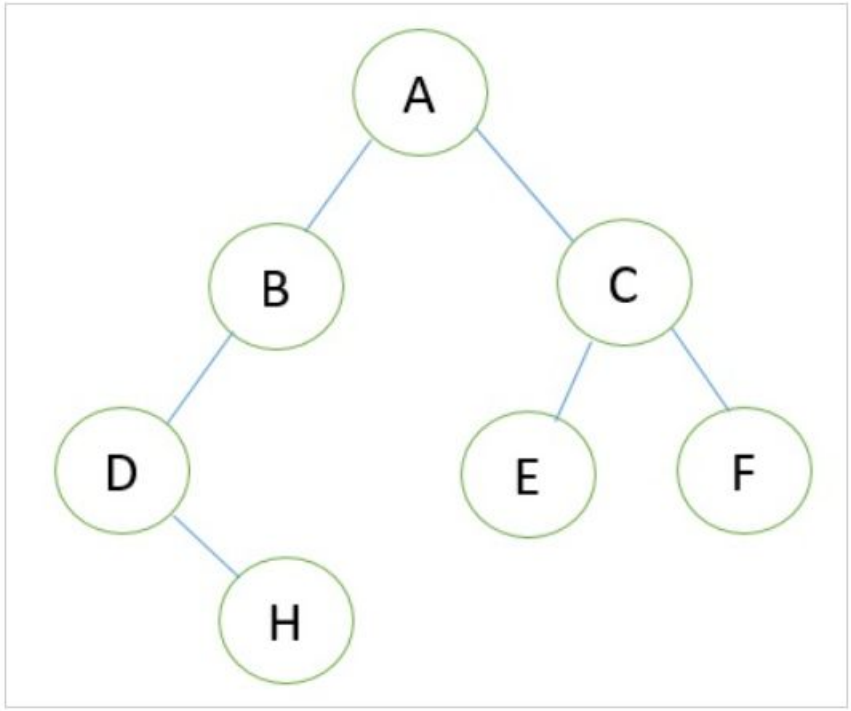


Figure 9: List Representation of T
Figure 8

Sequential & Linked List Representation (Binary Tree)



Its sequential representation is as follow:

A	B	C	D	-	E	F	-	H		

Sequential & Linked List Representation (Binary Tree)

In linked representation, Tree is maintained in memory by means of three parallel arrays, INFO, LEFT, and RIGHT, and a pointer variable ROOT.

Each node N of T will correspond to a location K such that **INFO[K]** contains data at node N. **LEFT[K]** contains the location of the left child of node N and **RIGHT[K]** contains the location of right child of node N. **ROOT** will contain the location of root R of Tree. If any subtree is empty, the corresponding pointer will contain a null value. If the tree T itself is empty, then ROOT will contain a null value

Figure 9: List Representation of Tree in Figure 8

Static Binary Tree Representation :

Array Representation :

- Binary tree using array represents a node which is numbered sequentially level by level from left to right. Even empty nodes are numbered.
- **Array index** is a value in tree nodes and **array value** gives to the parent node of that particular index or node. Value of the **root node index is always -1 as** there is no parent for root. When the data item of the tree is sorted in an array, the number appearing against the node will work as indexes of the node in an array.
- Location number of an array is used to store the size of the tree. The first index of an array that is 'o', stores the total number of nodes. All nodes are numbered from left to right level by level from top to bottom. In a tree, each node having an index i is put into the array as its i th element.

Binary Tree Representation using Array

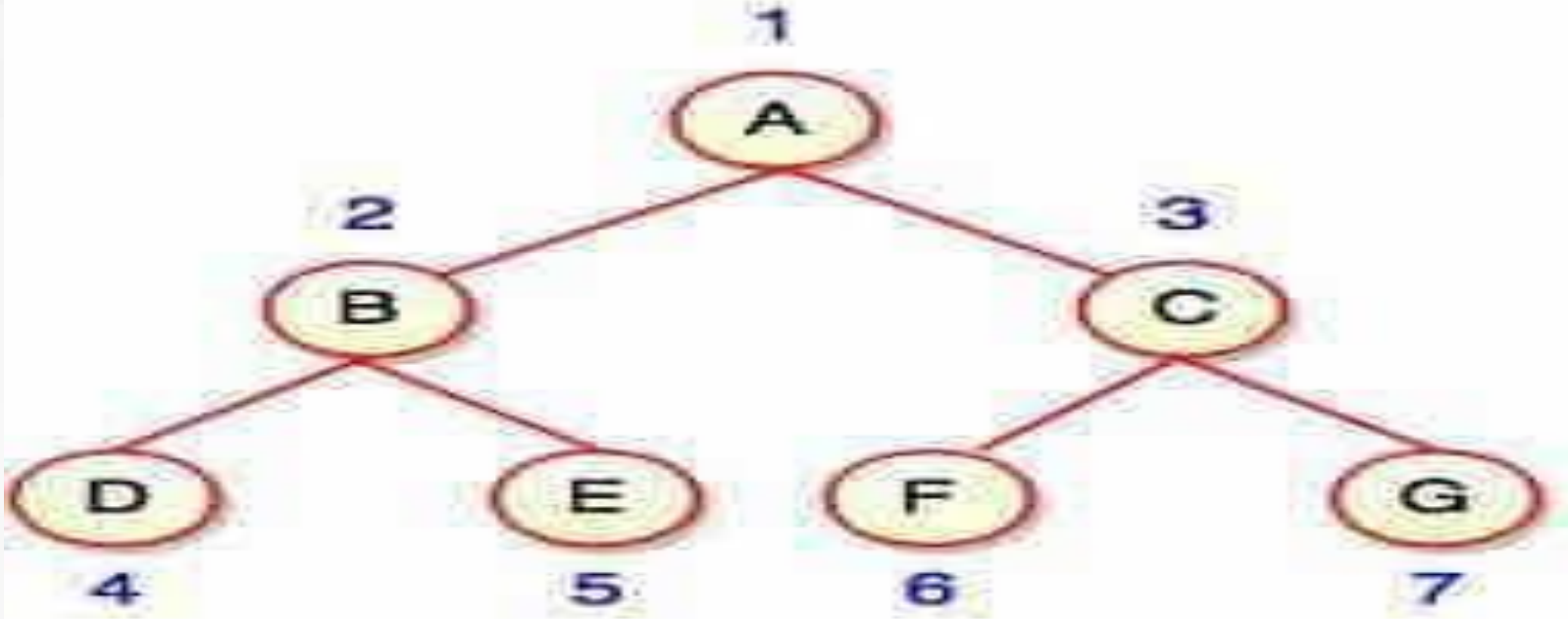


Fig. Binary Tree using Array

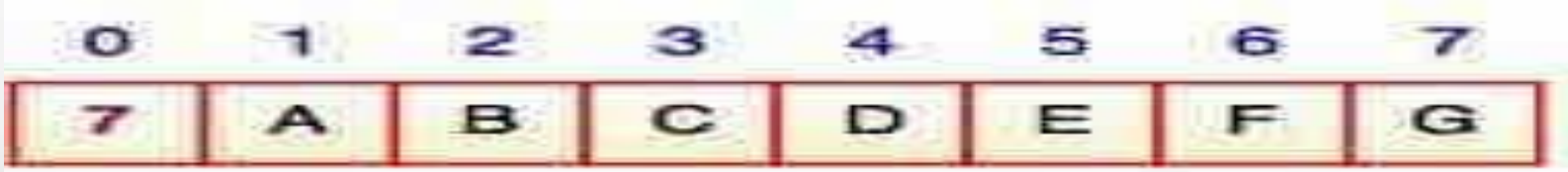
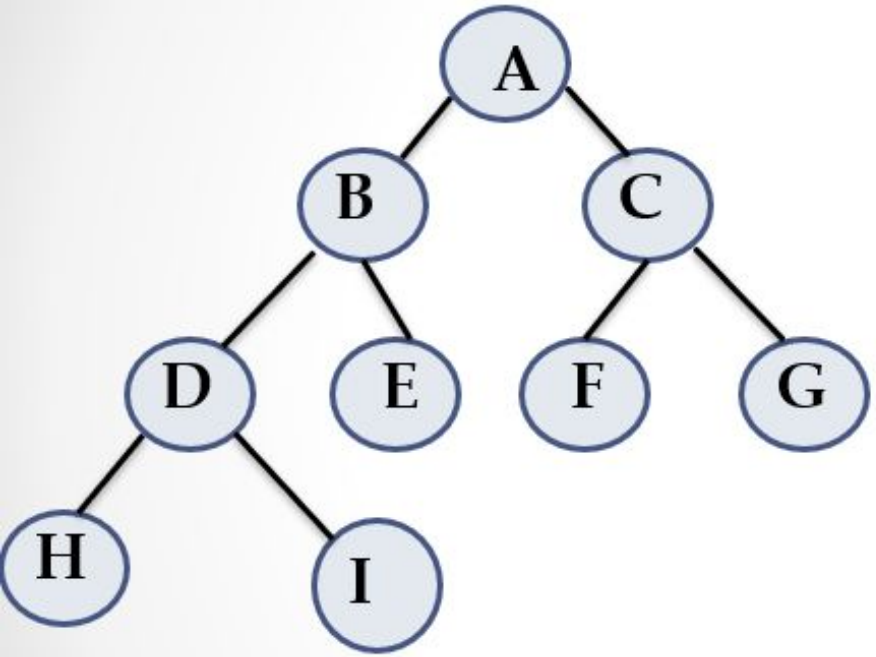


Fig. Location Number of an Array in a Tree

Array Implementation of a Binary Tree

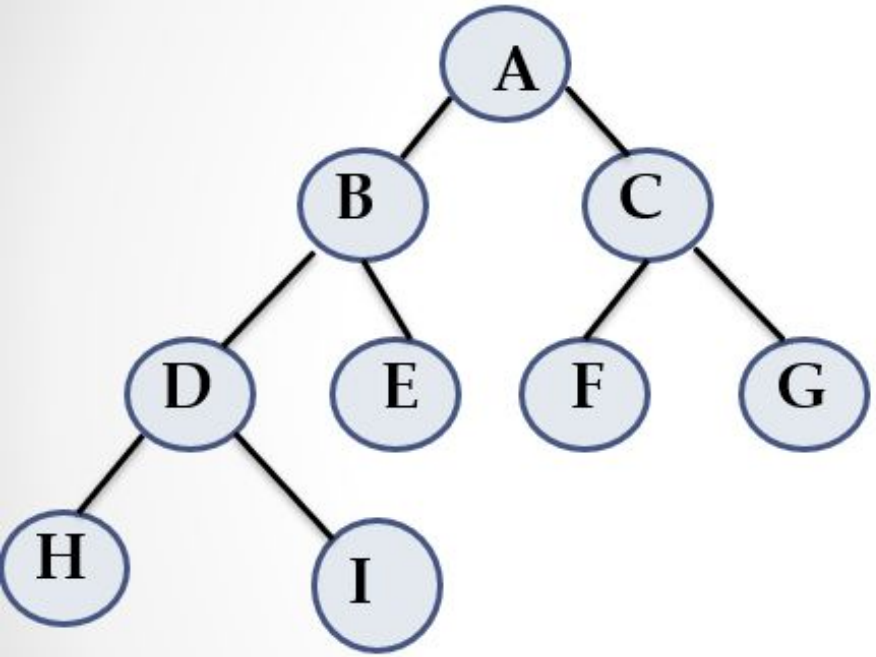


- If a node is at i^{th} index and Root is at index 0,
  - Height of the tree = 3
  - Size of array= $2^{h+1}-1=15$
  - -Parent= $\left\lfloor \frac{i-1}{2} \right\rfloor$
  - -Left_child=[(2*i)+1]
  - -Right_child =[(2*i)+2]

A	B	C	D	E	F	G	H	I	/	/	/	/	/	/
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

41

Array Implementation of a Binary Tree



- If root value starts from position 1, then:
- Height of the tree = 3
- Size of array= $2^{h+1}-1=15$
- Parent= $\left\lfloor \frac{i}{2} \right\rfloor$

Left_child=2 * i

Right_child =2 * i +1

A	B	C	D	E	F	G	H	I	/	/	/	/	/	/
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

42

Advantages of Array Representation of Binary Tree

- **Any node can be accessed from any other node by calculating the index**
- Here, data are stored without any pointers to their successor or ancestor
- **Programming languages, where dynamic memory allocation is not possible (such as BASIC, FORTRAN), array representation is the only mean to store a tree**

Disadvantages of Array Representation of Binary Tree

- Other than full binary trees, majority of the array entries may be empty
- Inserting a new node or deleting a node are inefficient with this representation, because these require considerable data, movement up and down the array which **demand** excessive Inserting amount of processing time

Dynamic -Binary Tree Representation

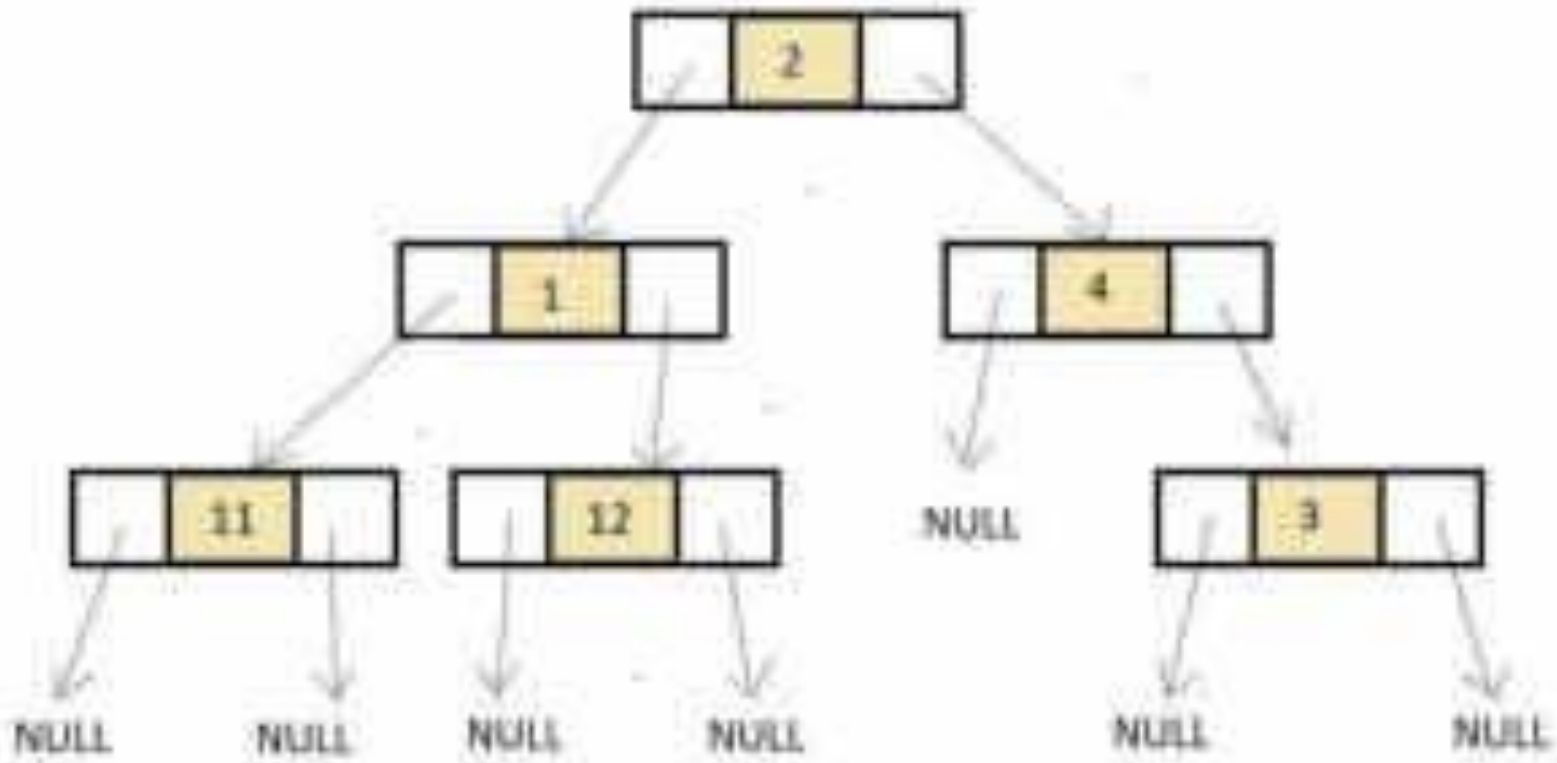
Linked representation :

Binary trees in linked representation are stored in the memory as linked lists. These lists have nodes that aren't stored at adjacent or neighboring memory locations and are linked to each other through the parent-child relationship associated with trees.

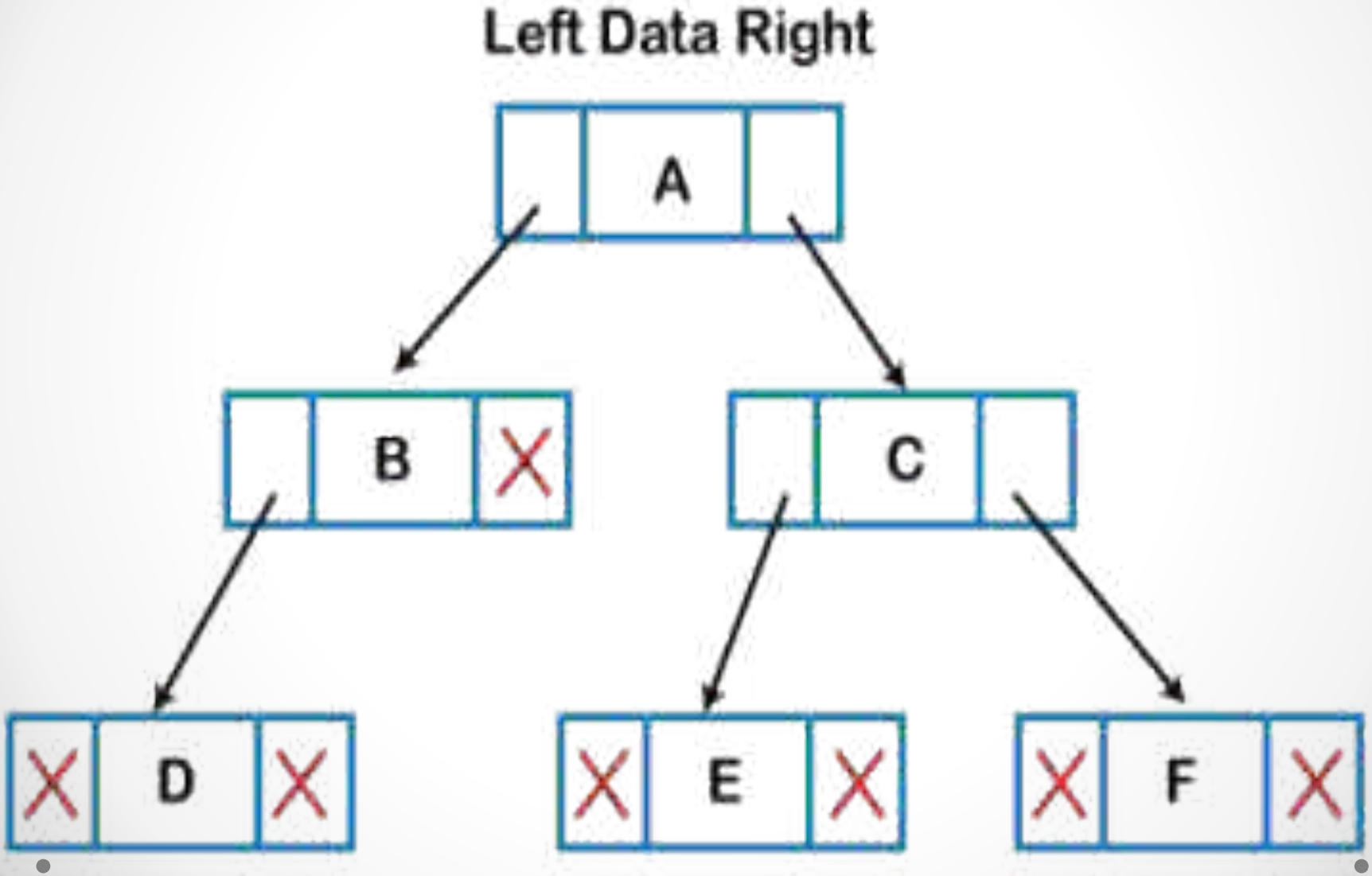
In this representation, each node has three different parts –

- pointer that points towards the right node,
- pointer that points towards the left node,
- data element.

Binary Tree Representation



Binary Tree Representation



Binary Tree Representation

Linked representation :

This is the more common representation. All binary trees consist of a root pointer that points in the direction of the root node. When you see a root node pointing towards null or 0, you should know that you are dealing with an empty binary tree. The right and left pointers store the address of the right and left children of the tree

Sequential & Linked List Representation (Binary Tree)

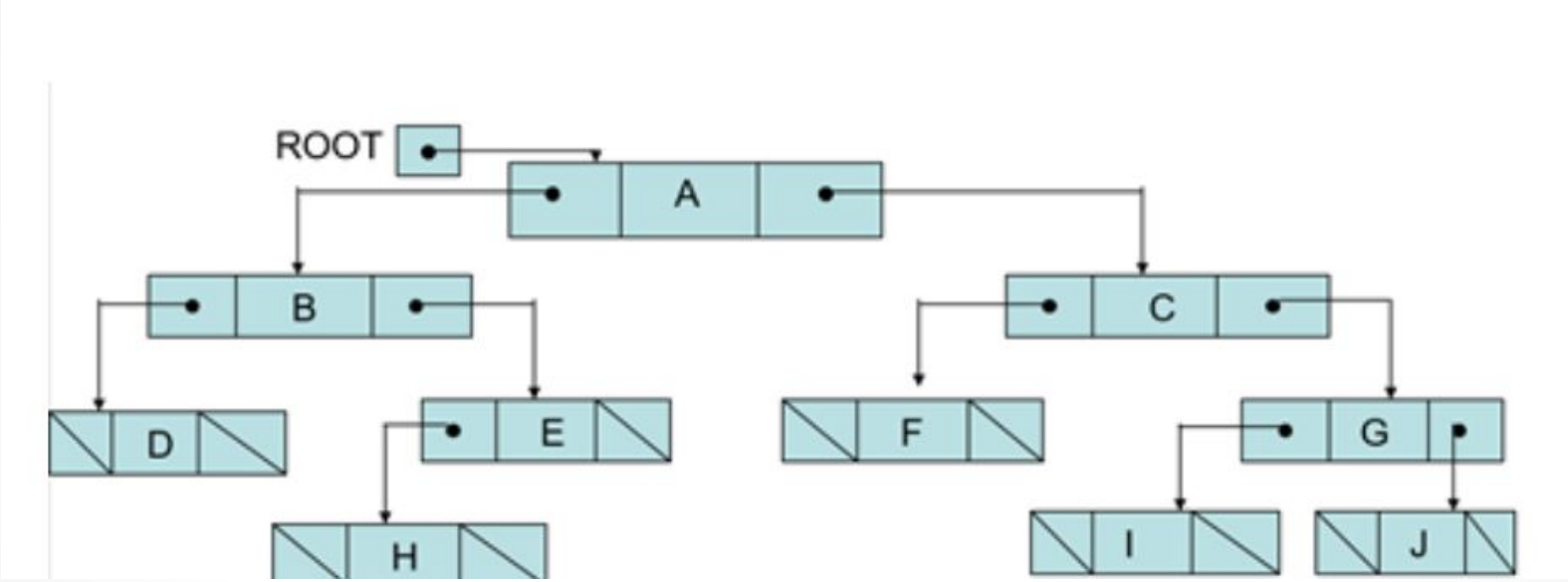
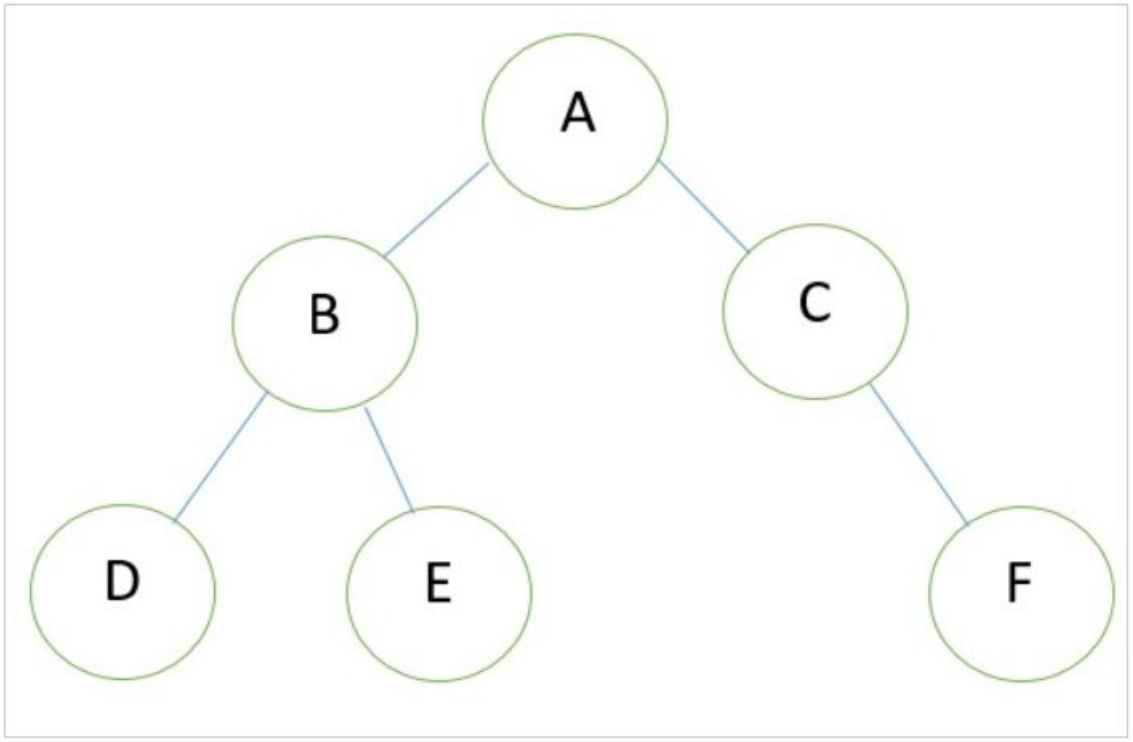


Figure 9: List Representation of Tree in Figure 8

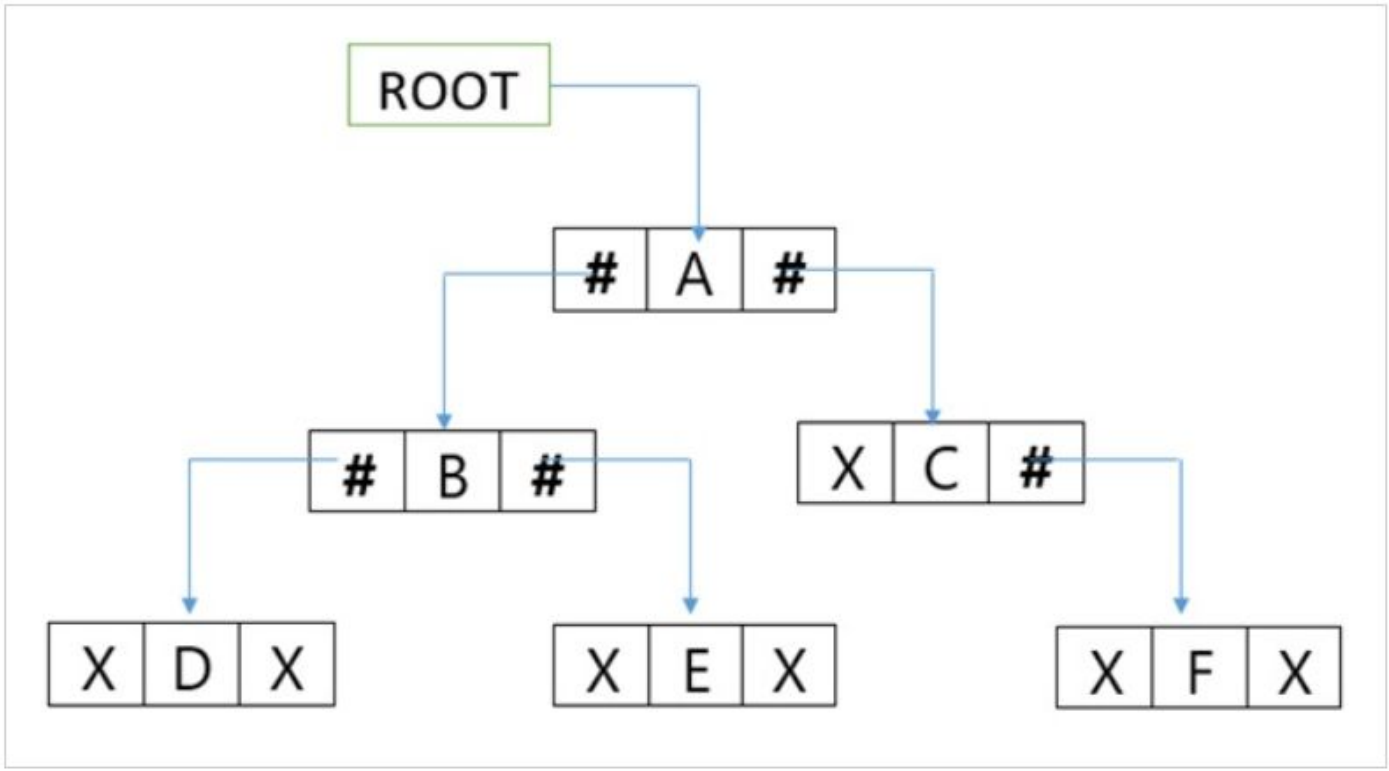
Sequential & Linked List Representation (Binary Tree)

Binary Tree



Sequential & Linked List Representation (Binary Tree)

Linked Representation of the Binary Tree



Advantages of Linked Representation of Binary Tree

- Efficient use of memory than that of sequential representation
- Insertion and deletion operations are more efficient in this representation.
- Use of dynamic memory allocation

Disadvantages of Linked Representation of Binary Tree

- In this representation, there is no direct access to any node. It has to be traversed from root to reach to a particular node
- As compared to sequential representation memory needed per node is more
- This is due to the presence of two additional pointers (left child and right child for binary trees) in node

Linked Implementation of Binary Trees

- Binary tree has natural implementation in linked storage. In linked organization we wish that all nodes should be allocated dynamically
- Hence we need each node with data and link fields
- Each node of a binary tree has both a left and a right subtree

Binary Tree Examples

Example 1: Consider the example of Perfect binary tree

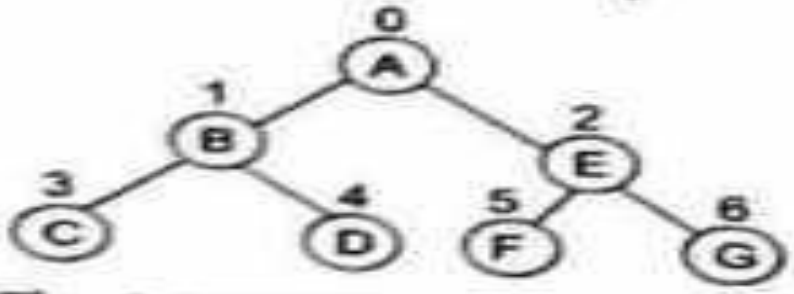


Fig. 6.13: Perfect binary tree

In Fig. 6.13,

Number of levels = 3 (0 to 2) and height = 2

Therefore, we need the array of size $2^{2+1} - 1$ is $2^3 - 1 = 7$.

The representation of the above binary tree using array is as followed:

1	2	3	4	5	6	7
A	B	C	D	E	F	G

We will number each node of tree starting from the root. Nodes on same level will be numbered left to right.

Example 2: Consider the example

Binary Tree Examples

Example 2: Consider the example of Complete binary tree.

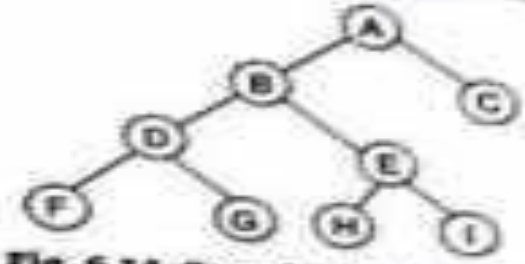


Fig. 6.14: Complete binary tree

Here, depth = 4 (level), therefore we need the array of size $2^4 - 1 = 15$. The representation of the above binary tree using array is as follows:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E	–	–	F	G	H	I	–	–	–	–

level 0 1 2 3

We can apply the above rules to find array representation.

1. Parent of node E (node 5) = $\frac{i}{2} = \frac{5}{2} = 2$ i.e. B.

   Hence, node B is at position 2 in the array.
2. Left child (i) = 2i.

   For example, left child of E = 2 × 5 = 10 i.e. H.

   Since, E is the 5th node of tree.
3. Right child (i) = 2i + 1

For example: Right child of D = 2 × 4 + 1 = 8 + 1 = 9 i.e. G.

Since, D is the 4th node of tree and G is the 9th element of an array.

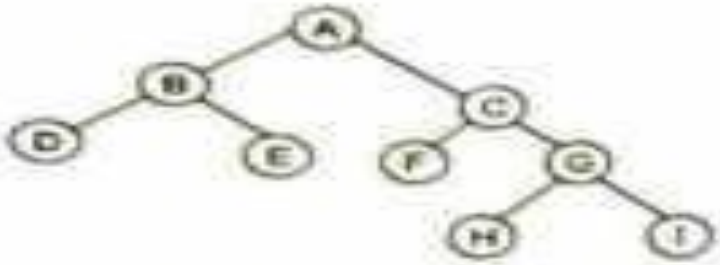
Binary Tree Examples

Example 3: Consider the example of skewed tree.

Fig. 6.15: Skewed binary tree

ee Examples

- Using this declaration for linked representation, the binary tree representation can be logically viewed as in Fig. 6.18 (c). In Fig. 6.18 (b) physical representation shows the memory allocation of nodes.

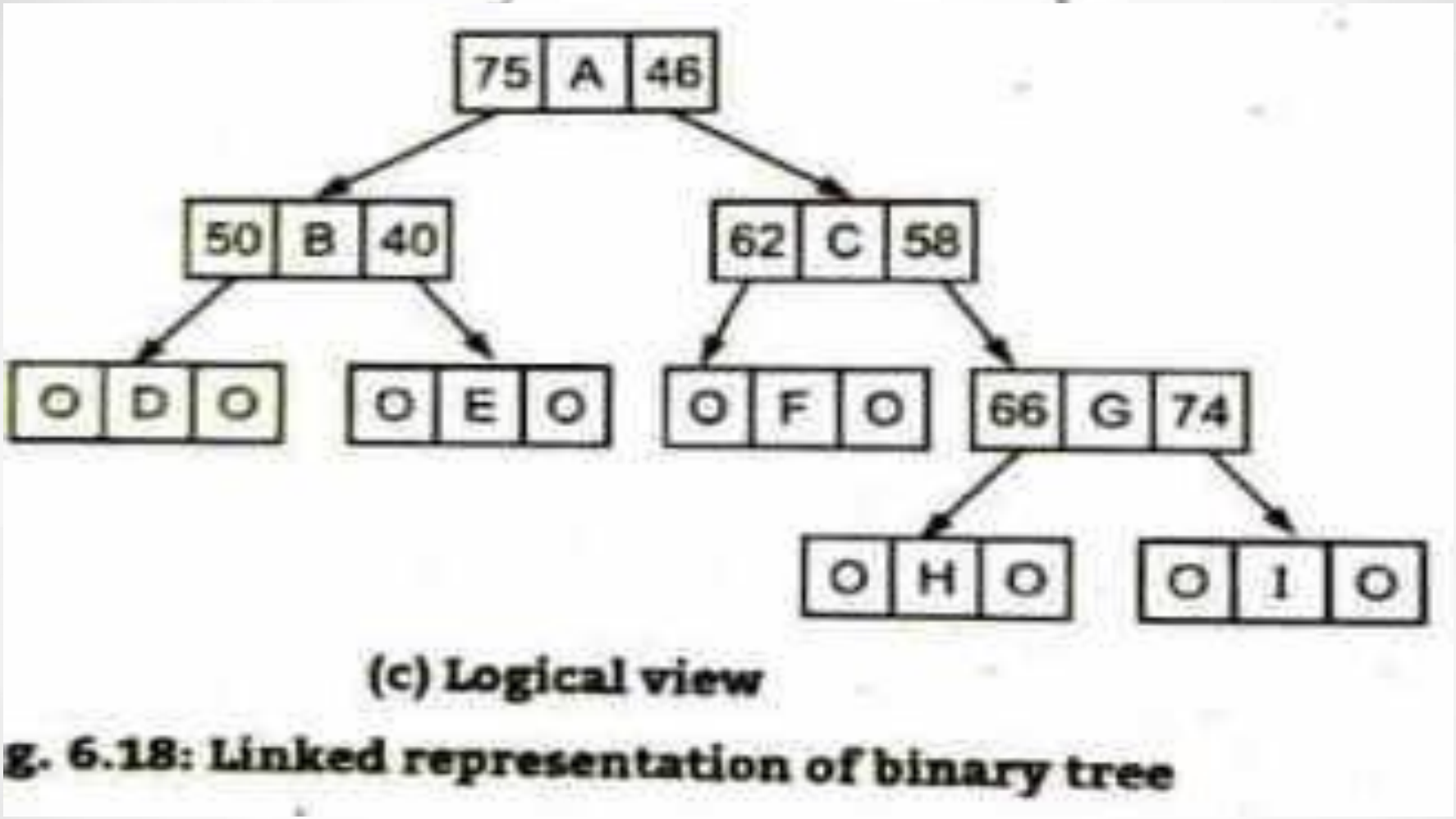


(a) A binary tree

Address	Node		
	Lchild	Data	Rchild
50	0	D	0
75	50	B	40
40	0	E	0
89	75	A	46
62	0	F	0
46	62	C	58
66	0	H	0
58	66	G	74
74	0	I	0

(b) A binary tree and its various nodes (physical view)

Binary Tree Examples



(c) Logical view

g. 6.18: Linked representation of binary tree

Types of Tree

- Free tree
- Rooted tree
- Ordered tree
- Position tree
- Complete tree
- Regular tree
- Binary tree

Free Tree

Specification

- A **free tree** is a **connected, acyclic graph**
- It is **undirected graph**
- It has no node designated as a root
- **As it is connected, any node can be reached from any other node by a unique path**

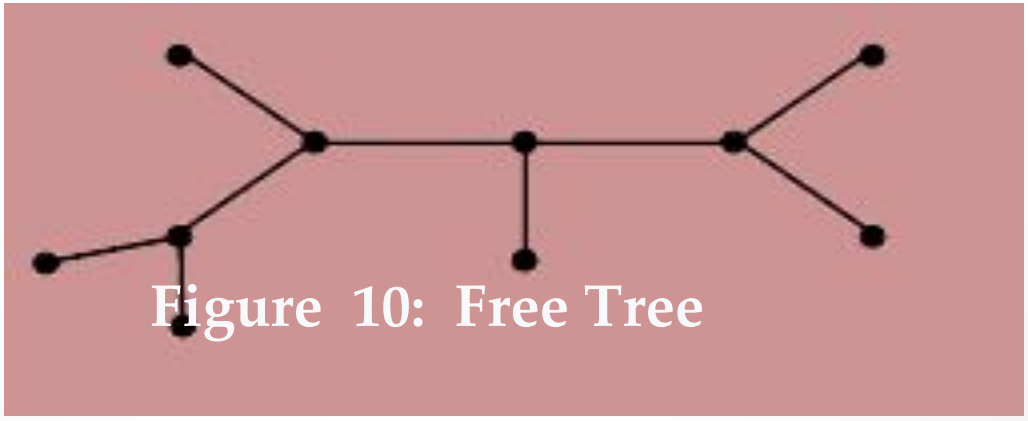
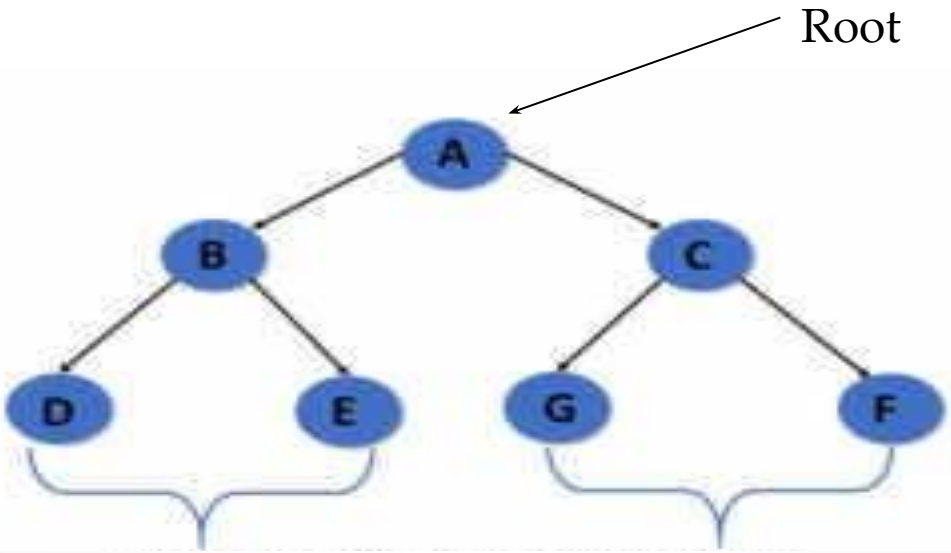


Figure 10: Free Tree

Rooted Tree

- Unlike **free tree, rooted tree** is a **directed graph** in which one node is designated as root, whose incoming degree is zero
- **And for all other nodes incoming degree is one**



Ordered Tree

- In many applications the relative order of the nodes at any particular level assumes some significance
- It is easy to impose an order on the nodes at a level by referring to a particular node as the first node, to another node as the second, and so on
- Such ordering can be done left to right

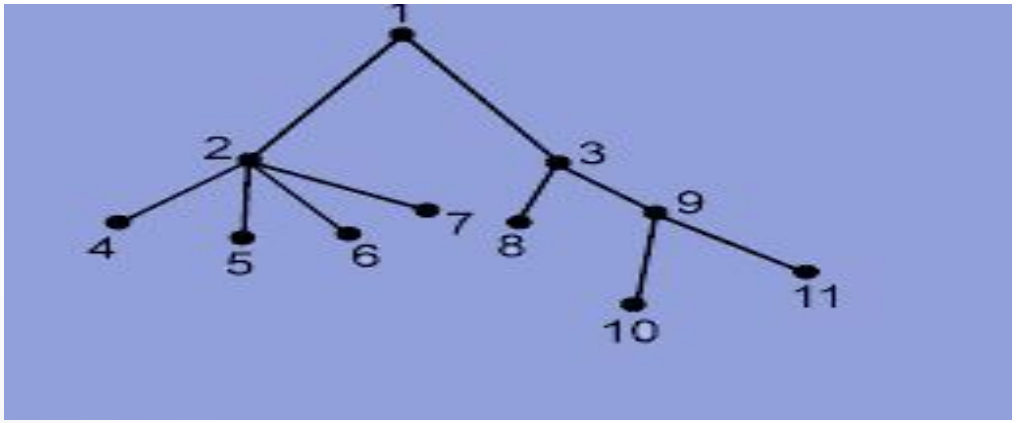
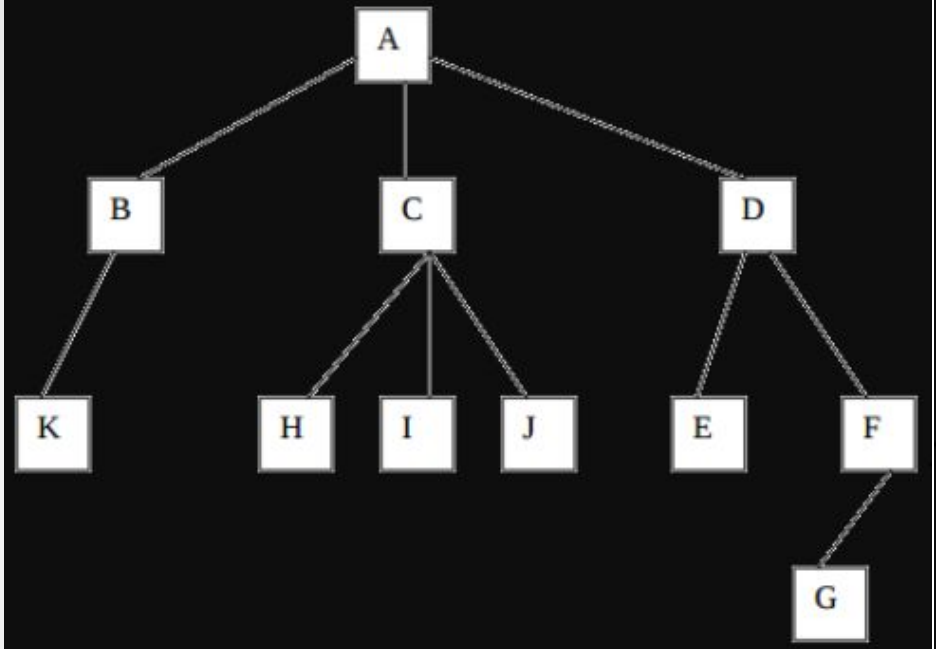
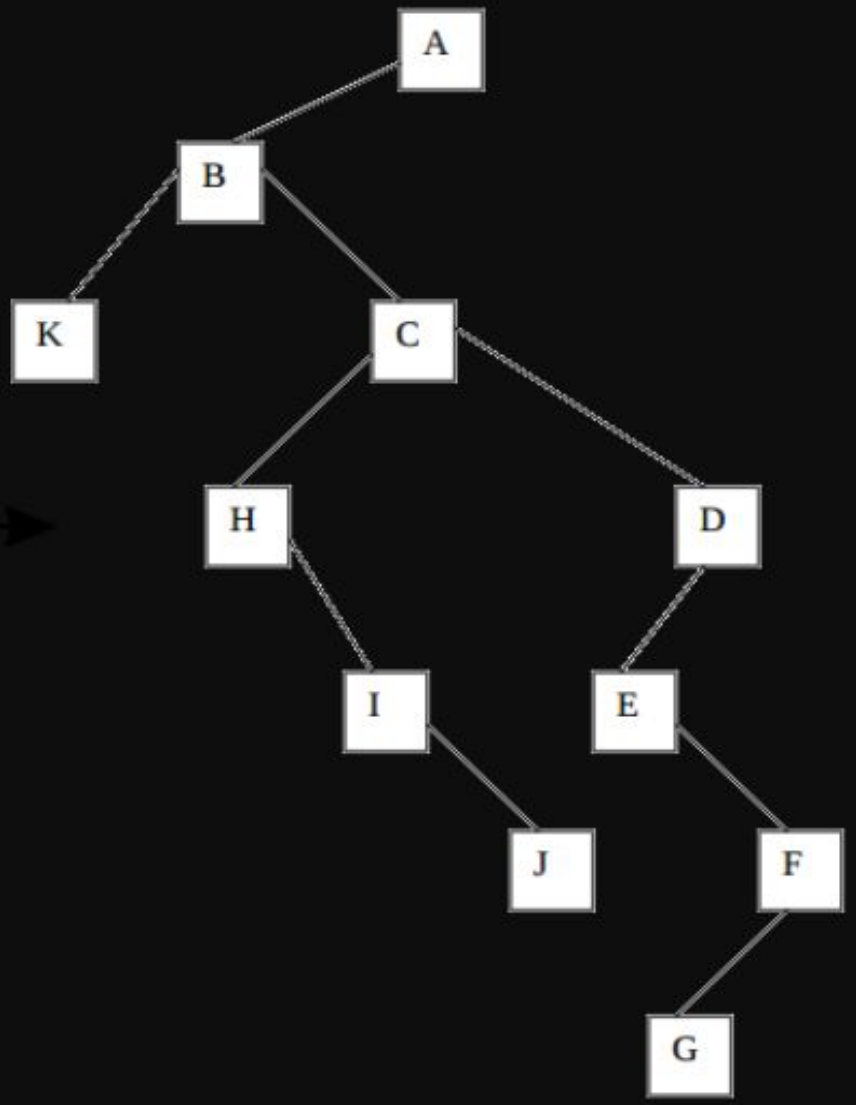


Figure 12: Ordered Tree

Ordered Tree



Unordered Tree



Regular Tree

- A tree in which each branch node vertex has the same out-degree is called as **Regular Tree**
- In a rooted tree, if every internal vertex has no more than m children(atmost m childeren), it is called a **m-ary tree**
- A rooted tree is called a **full m-ary** tree if every internal vertex has exactly **m** children. It is also called **regular m-ary tree**

Are the rooted trees in Figure 7 full m-ary trees for some positive integer m?

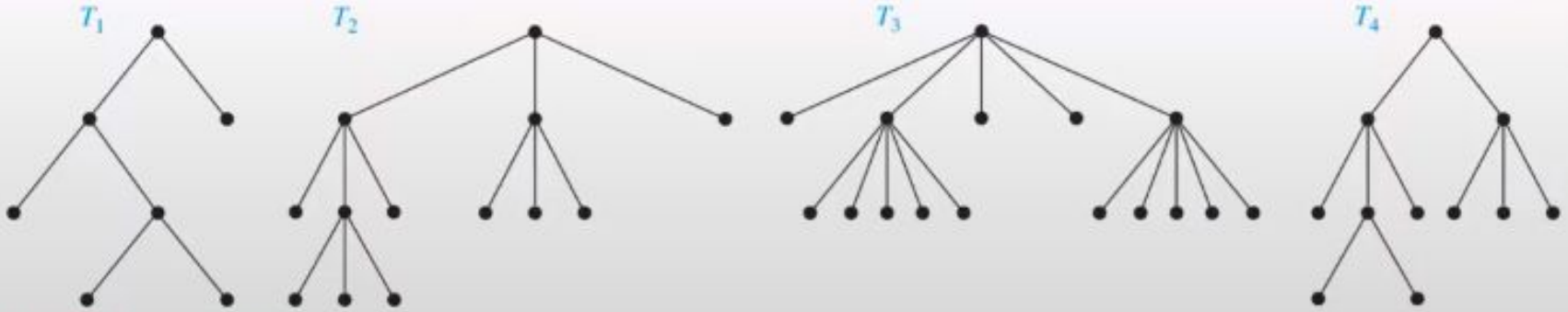


FIGURE 7 Four rooted trees.

Are the rooted trees in Figure 7 full m-ary trees for some positive integer m?

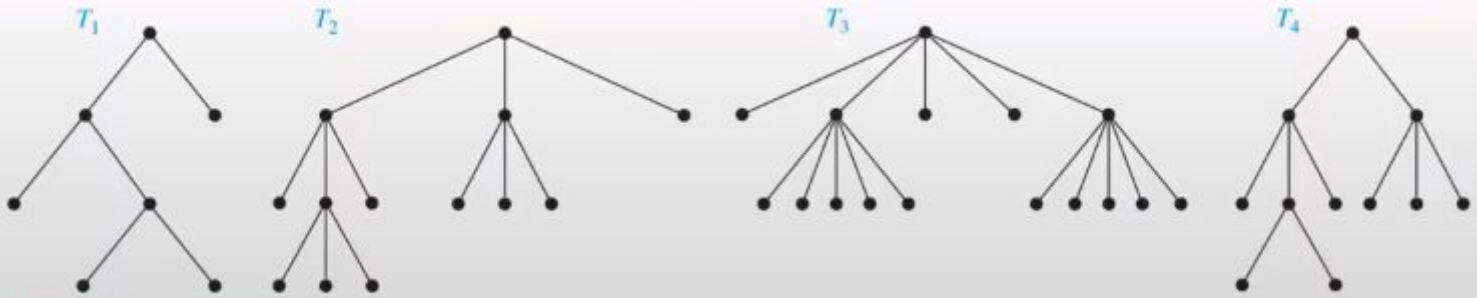


FIGURE 7 Four rooted trees.

EXAMPLE 3

Are the rooted trees in Figure 7 full m-ary trees for some positive integer m?

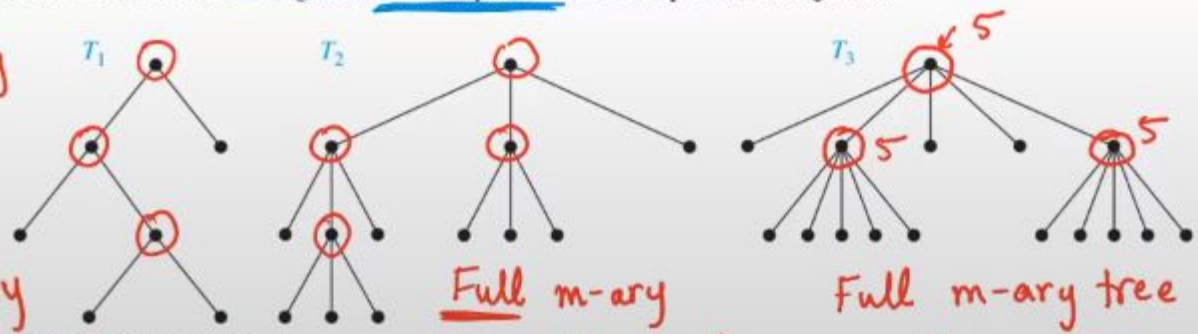


FIGURE 7 Four rooted trees.

T_1 Full m-ary tree with $m=2$ Full binary tree

Full m-ary tree with $m=3$

Full m-ary tree with $m=5$

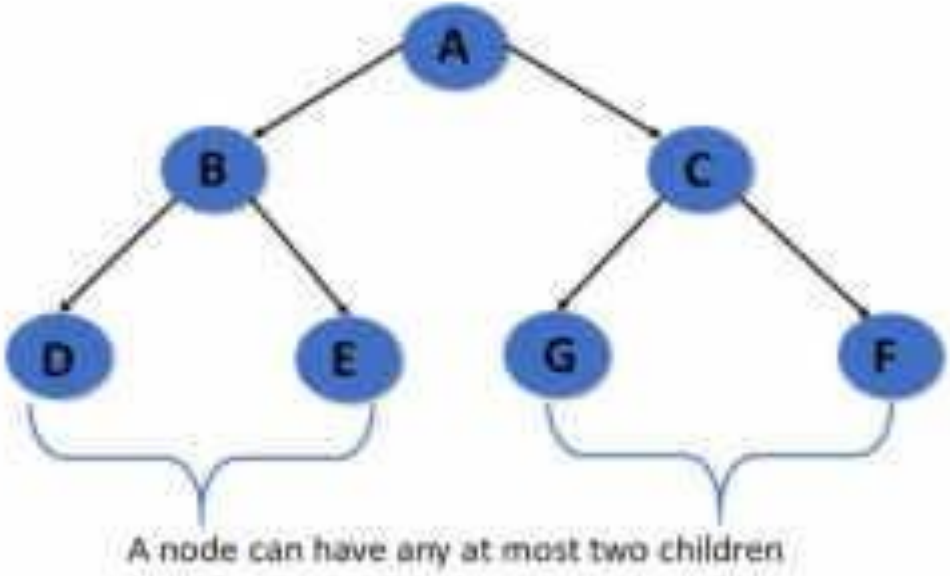
NO T_4 is not a full m-ary tree.

It is just an m-ary tree with $m=3$

Binary Tree

- **A binary tree has the following properties:**

- **Properties**
- Follows all properties of the tree data structure.
- Binary trees can have at most two child nodes.
- These two children are called the left child and the right child.



Binary Tree

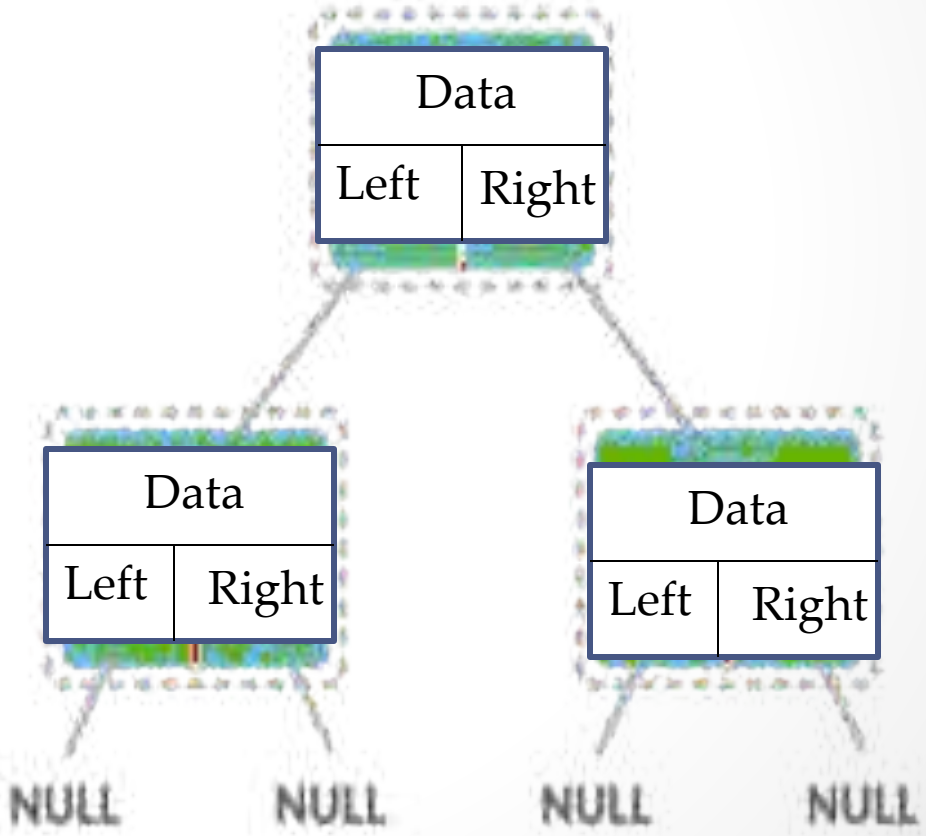
A binary tree has the following properties:

- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 2 (2 edges connecting 3 nodes). Therefore, the maximum number of nodes at height 2 is equal to (1+2+4) = 7. Maximum number of nodes for height, h is calculated as $2^{h+1} - 1$
- The minimum number of nodes possible at height h is equal to h+1.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum

Binary Tree Representation

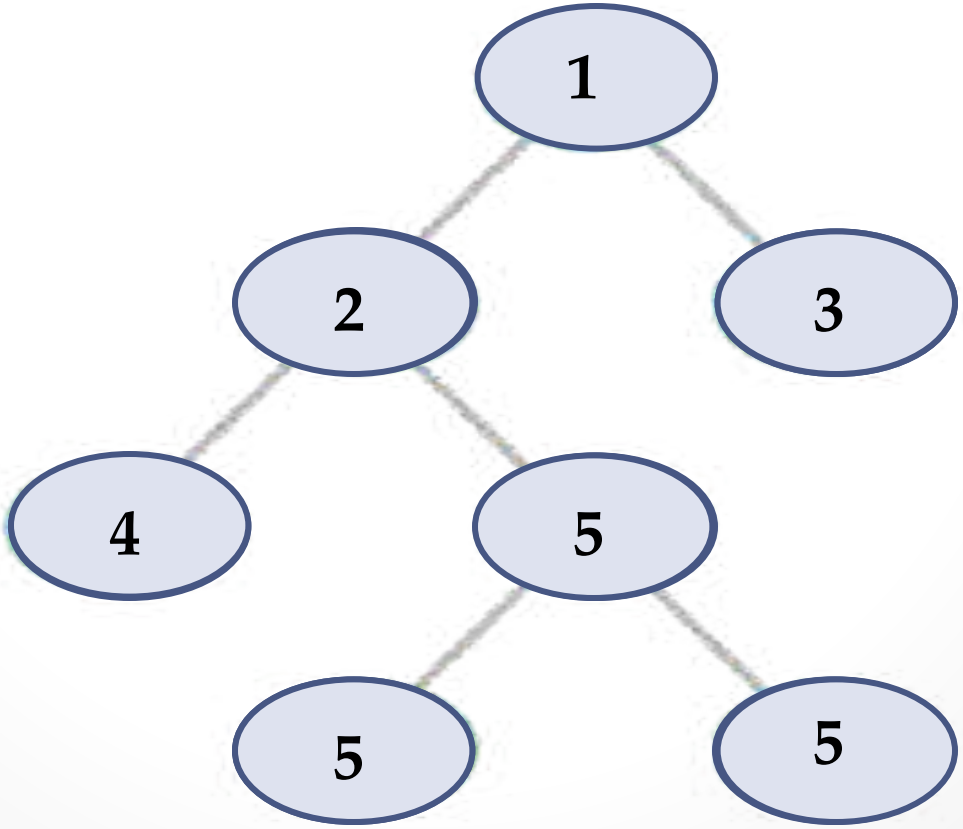
A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

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



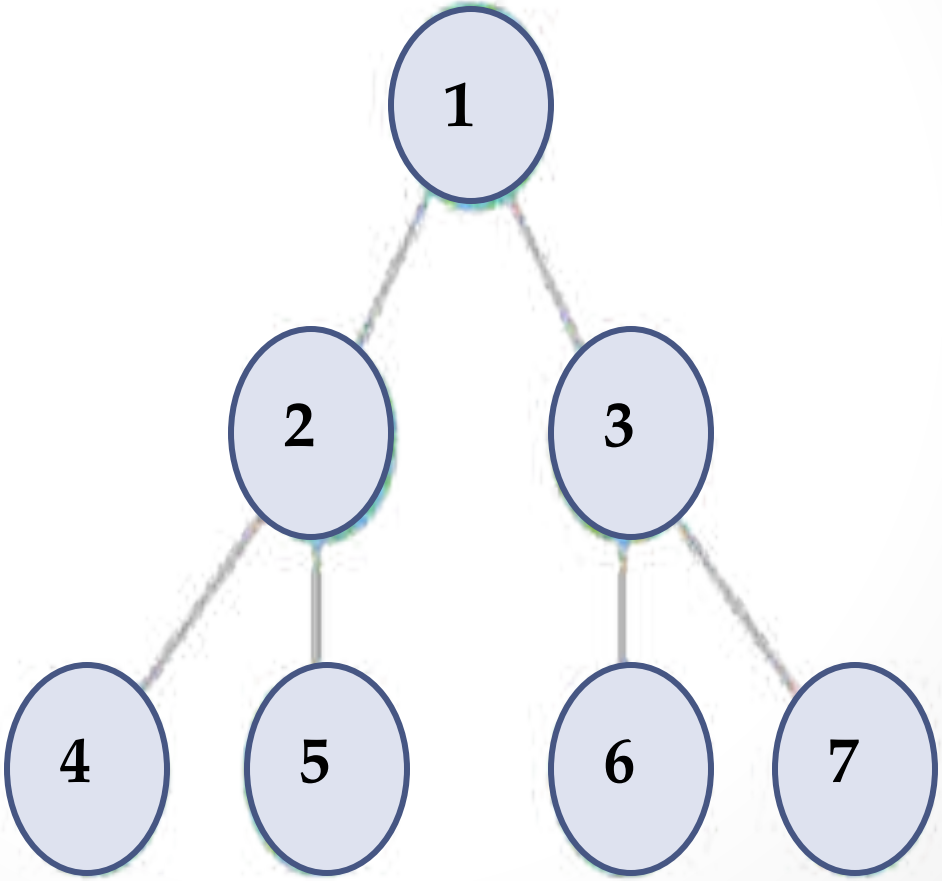
Types of Binary Tree

- **Full Binary tree/ Strictly Binary Tree:** It is a special type of binary tree. In this tree data structure, every parent node or an internal node has either two children or no child nodes.
- .



Types of Binary Tree

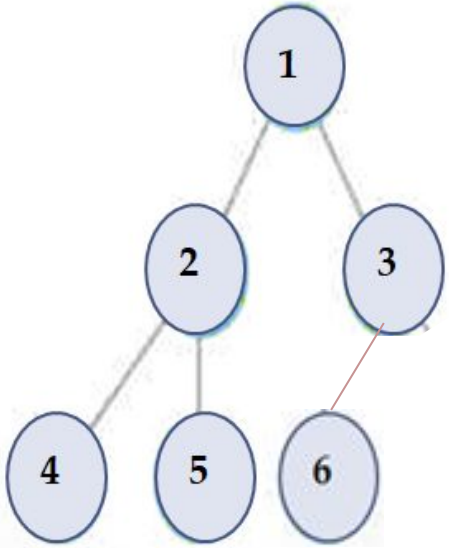
- **Perfect binary tree:** In this type of tree data structure, every internal node has exactly two child nodes and all the leaf nodes are at the same level.
- .



Binary Tree

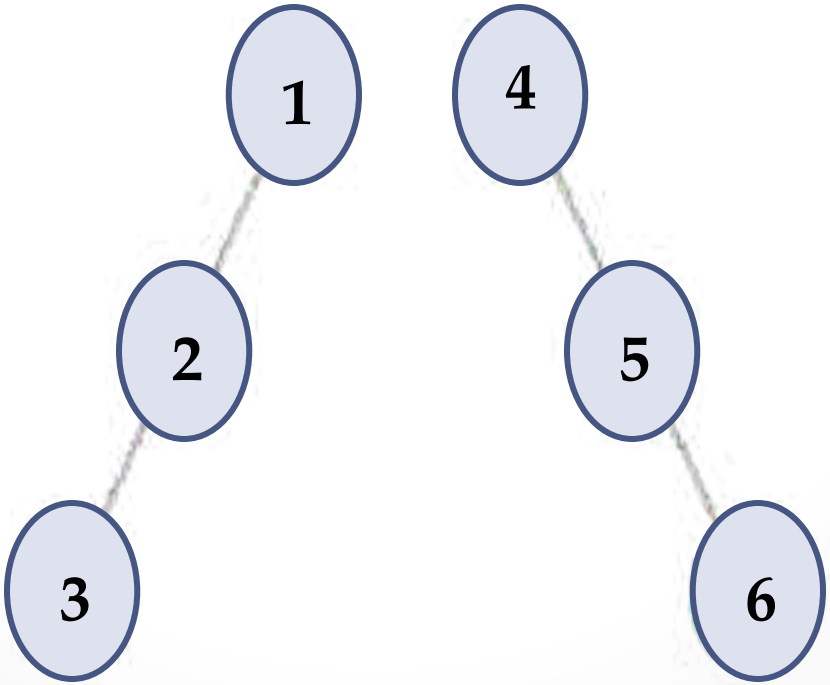
- **Complete binary tree:** It resembles that of the full binary tree with a few differences.

1. Every level is completely filled.
2. The leaf nodes lean towards the left of the tree.
3. It is not a requirement for the last leaf node to have the right sibling, i.e. a complete binary tree doesn't have to be a full binary tree.



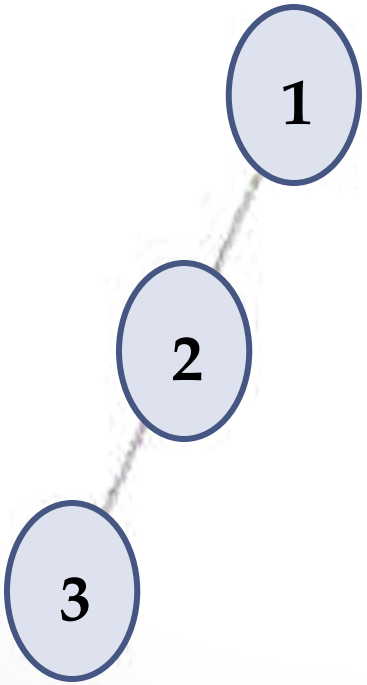
Binary Tree

- **Skewed binary tree:** It is a pathological or degenerate tree where the tree is dominated by either the left nodes or the right nodes. Therefore, there are two types of skewed binary trees, i.e. left- skewed or the right-skewed binary tree.



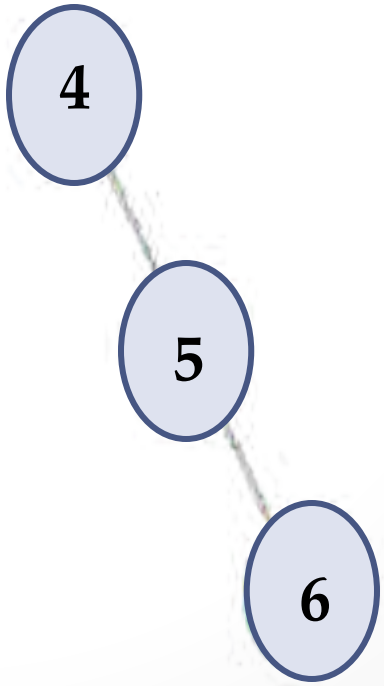
Binary Tree

- **Left Skewed binary tree:** These are those skewed binary trees in which all the nodes are having a left child or no child at all. It is a left side dominated tree. All the right children remain as null.



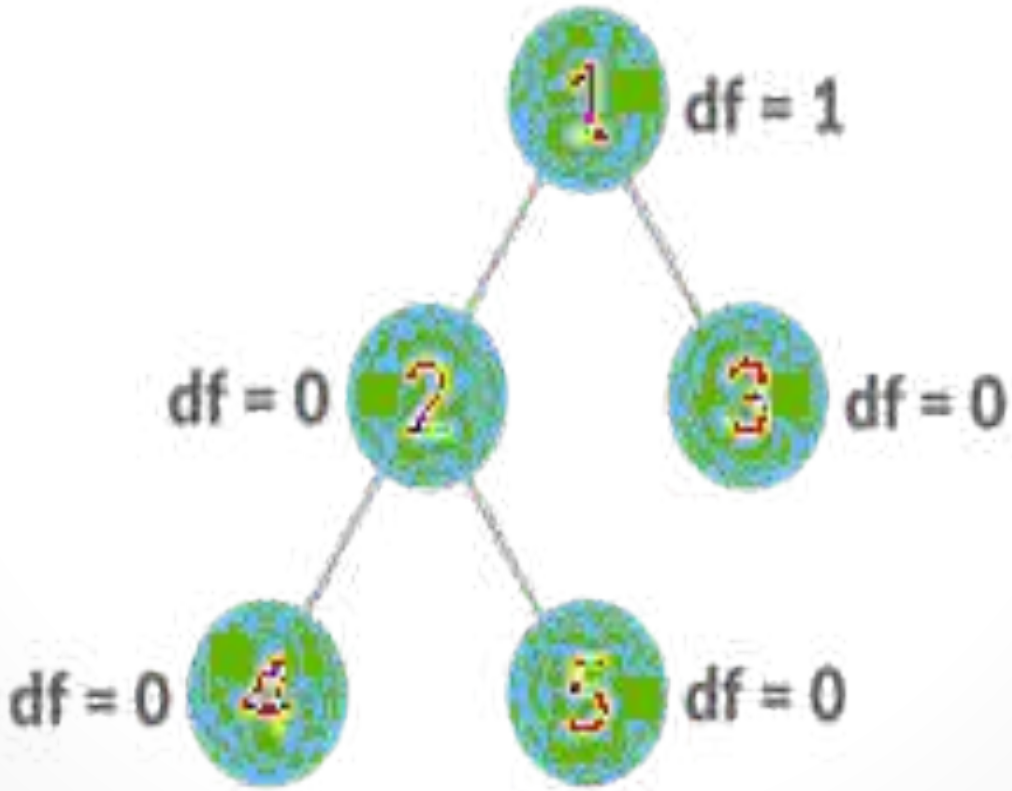
Binary Tree

- **Right Skewed binary tree:** These are those skewed binary trees in which all the nodes are having a right child or no child at all. It is a right side dominated tree. All the left children remain as null.



Binary Tree

- **Balanced binary tree:** The difference between the height of the left and right sub tree for each node is either 0 or 1.



Binary Tree

- A **Binary Tree** is a special form of a tree
- A tree in which each branch node vertex has the same out-degree **Binary tree** is important and frequently used in various applications of computer science

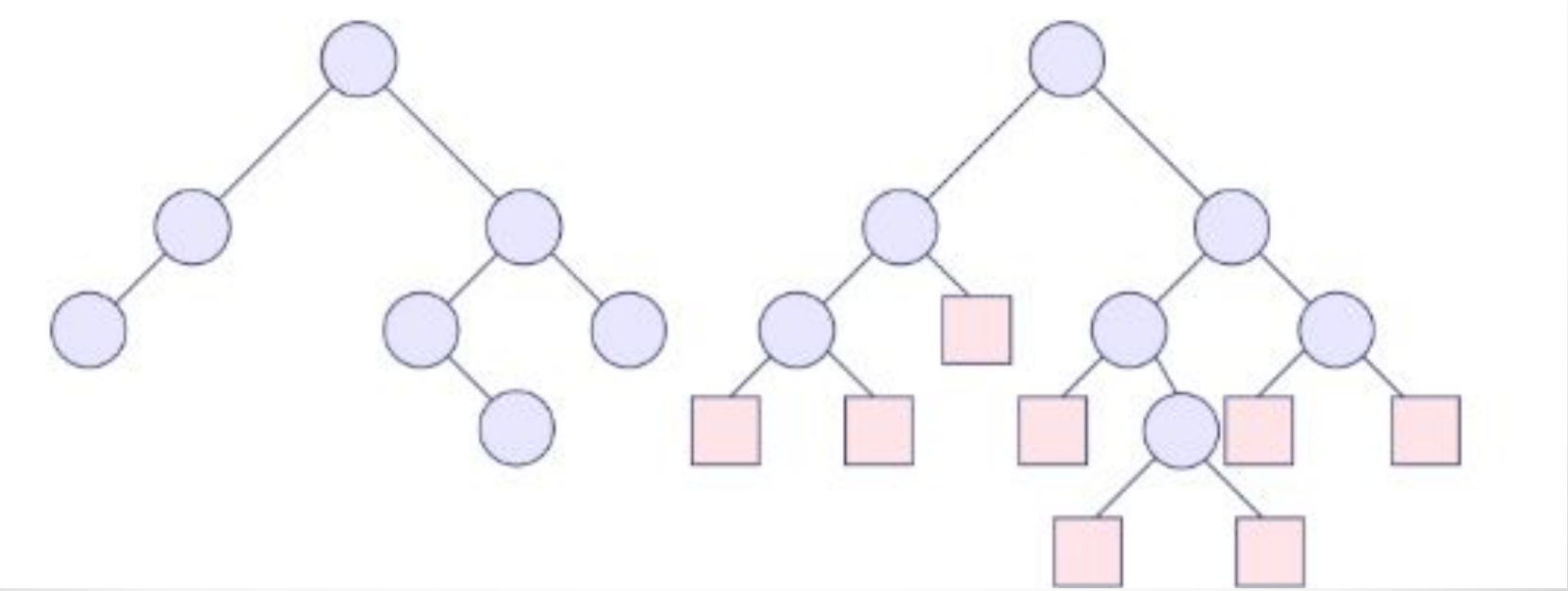
Binary Tree(cont….)

- Node with 2 children are called **internal nodes** and nodes with 0 children are called **external nodes**

- **Trees** can be converted into extended trees by adding a node
- A binary tree is said to be extended when it replaces all its null sub trees with special nodes.
- The **nodes from the original tree are internal nodes**, and **the special nodes are external nodes**.

Binary Tree(cont….)

- All external nodes are leaf nodes and the internal nodes are non-leaf nodes.
- Every internal node has exactly two children and every external node is a leaf.
- It displays the result which is a full binary tree.



Binary Tree(cont….)

- **<u>Definition :</u> A Binary Tree is either :** an empty tree; or consists of a node, called root and two children, left and right, each of which are themselves binary trees

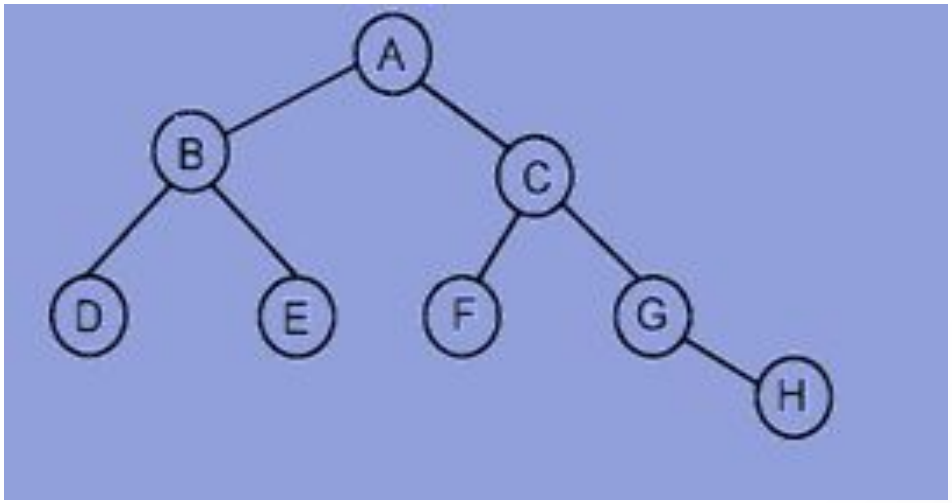


Figure 16 : Binary Tree

Binary Tree(cont….)

1. Declare CREATE( ) btree
2. MAKEBTREE(btree,element, btree) btree
3. ISEMPTY(btree) boolean
4. LEFTCHILD(btree) btree
5. RIGHTCHILD(btree) btree
6. DATA((btree) element

for all l,r ε btree, e ε element, Let

7. DATA(MAKEBTREE(l,e,r)) = e
8. ISEMPTY(CREATE) = true
9. ISEMPTY(MAKEBTREE(l,e,r)) = false
10. LEFTCHILD(CREATE( )) = error
11. RIGHTCHILD(CREATE( )) = error
12. LEFTCHILD(MAKEBTREE(l,e,r)) = l
13. RIGHTCHILD(MAKEBTREE(l,e,r)) = r
15. end

Operations on Binary Tree

The basic operations on a binary tree can be as listed below as follows:

- **Creation : Creating an empty binary tree to which 'root' points**
- **Traversal : To Visiting all the nodes in a binary tree**
- **Deletion : To Deleting a node from a non-empty binary tree**
- **Insertion : To Inserting a node into an existing (may be empty) binary tree**

Operations on Binary Tree

The basic operations on a binary tree can be as listed below as follows:

- Merge : To Merging two binary trees
- Copy : Copying a binary tree
- Compare : Comparing two binary trees
- Find replica or mirror

Realization Of A Binary Tree

The two main **reasons** are

- A **Binary Tree** has a natural implementation in linked storage
- The linked structure is more convenient for insertions and deletions

Operations of Binary Tree

Searching: For searching element, we have to traverse all elements (assuming we do breadth first traversal). Therefore, searching in binary tree has worst case complexity of O(n).

Insertion: For inserting element as left child , we have to traverse all elements. Therefore, insertion in binary tree has worst case complexity of O(n).

Deletion: For deletion of element, we have to traverse all elements to find (assuming we do breadth first traversal). Therefore, deletion in binary tree has worst case complexity of O(n).

Linked Implementation of Binary Trees

- Each node will have three fields Lchild, Data, and Rchild. Pictorially this node is shown in Fig.

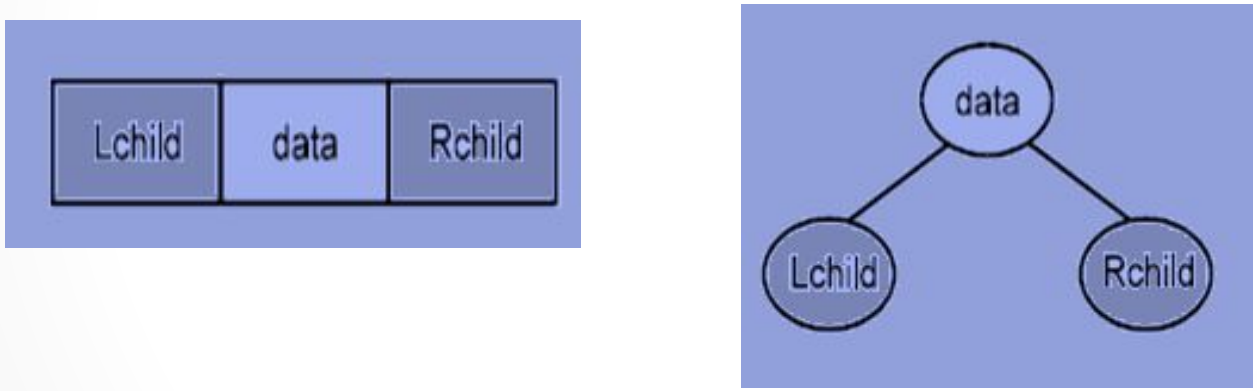


Figure 17 : Node Structure in Binary Tree

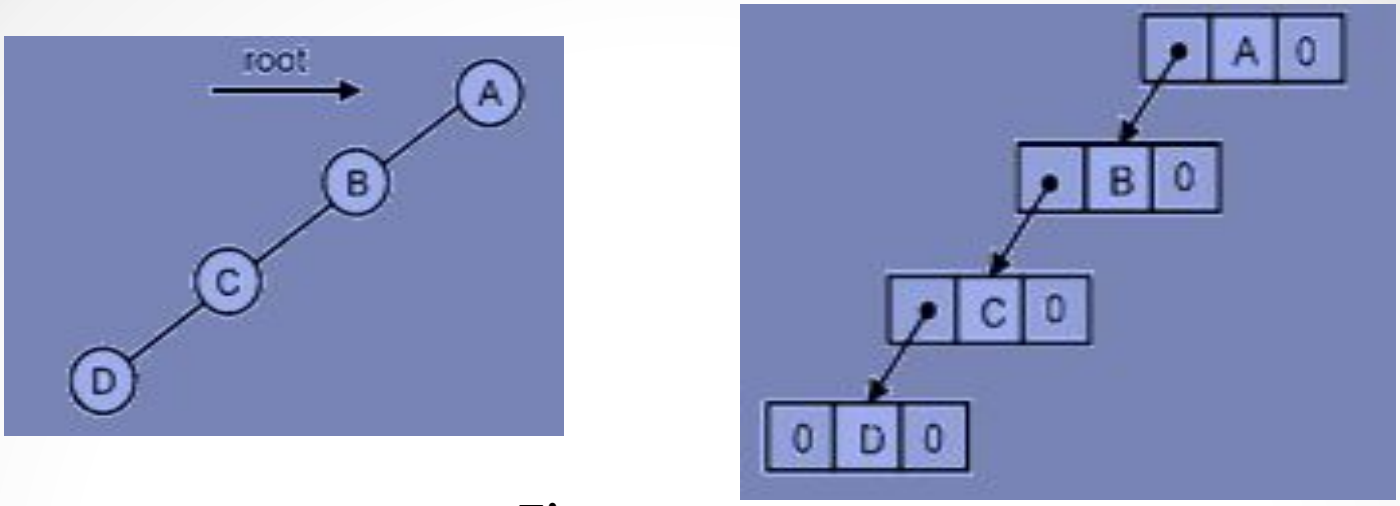


Figure ‘a’

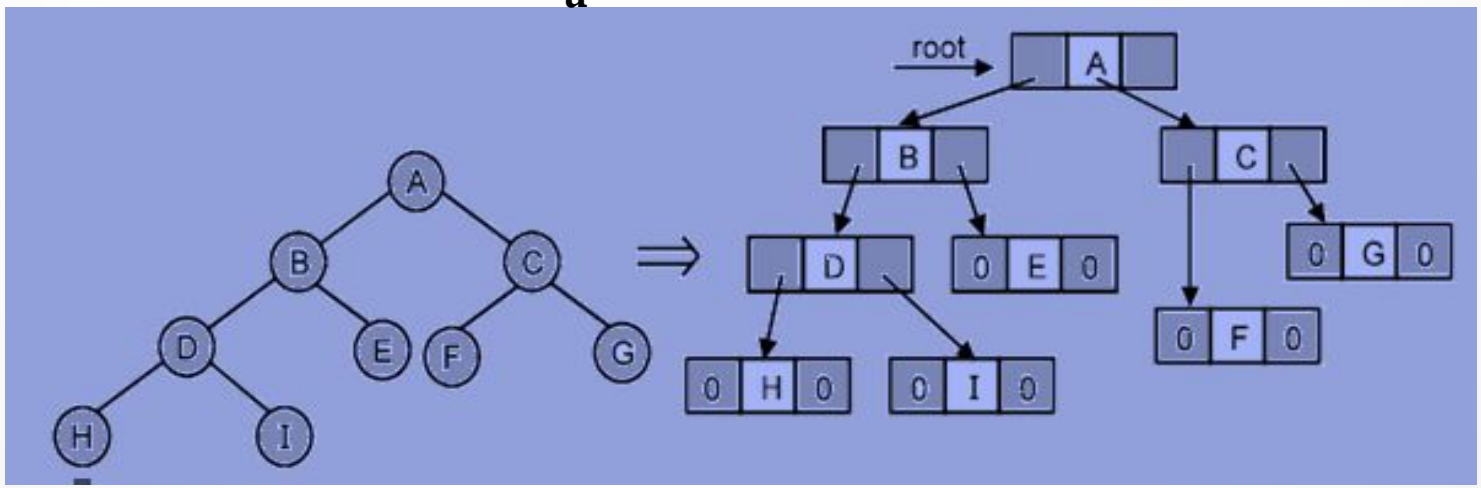


Figure ‘b’

Figure 18 : Linked Representation for Binary Tree

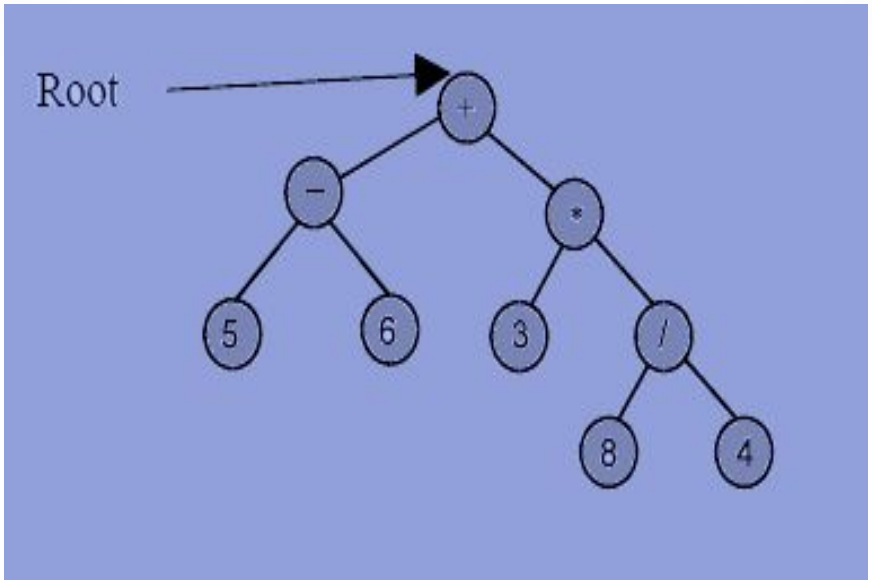


Figure 19: A Binary Tree Representing an Arithmetic Expression

Convert a Generic Tree (N-array Tree) to Binary Tree

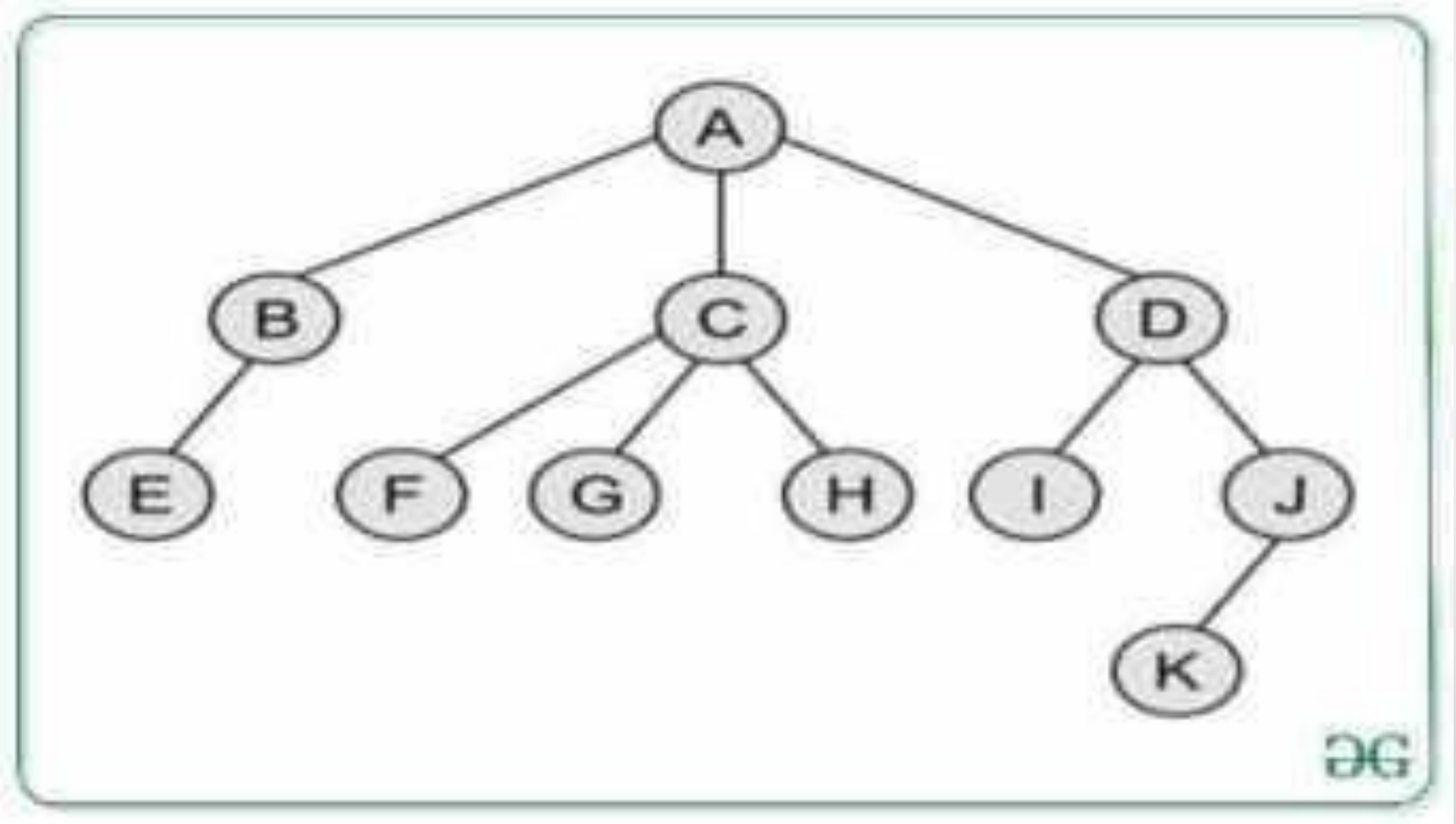
Following aretherules to convert a Generic(N-array Tree) to Binary Tree:

1. The root of theBinary Tree is theRoot of the Generic Tree.
2. The left child of a node in theGeneric Tree is the Left child of that node in the Binary Tree.
3. The right sibling of any node in theGeneric Tree is the Right child of that node in the Binary Tree.

Convert a Generic Tree(N-array Tree) to Binary Tree

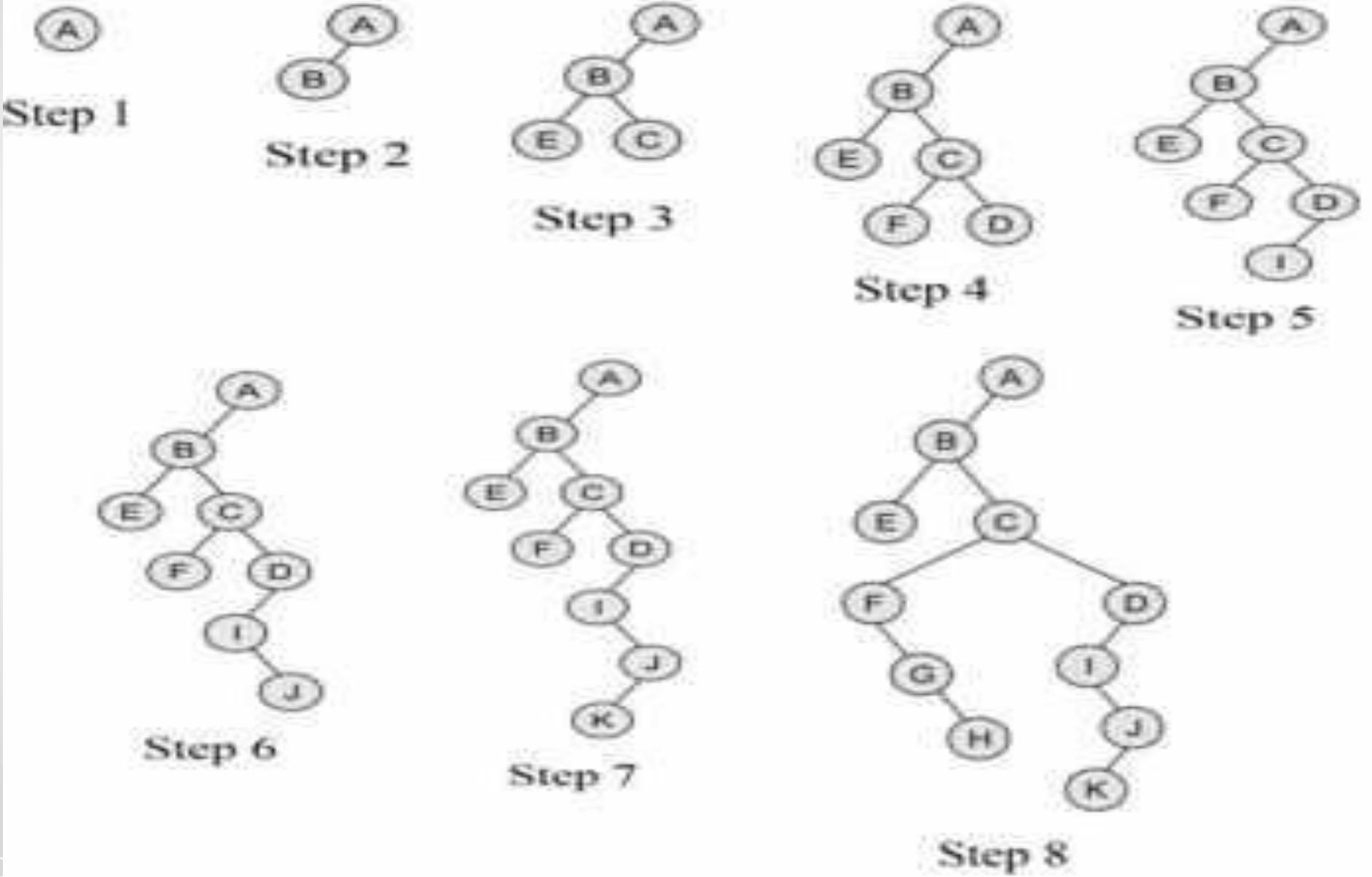
Examples:

Convert the following Generic Tree to Binary Tree:



Convert a Generic Tree(N-array Tree) to Binary Tree

Below is the Binary Tree of the above Generic Tree:



Convert a Generic Tree(N-array Tree) to Binary Tree

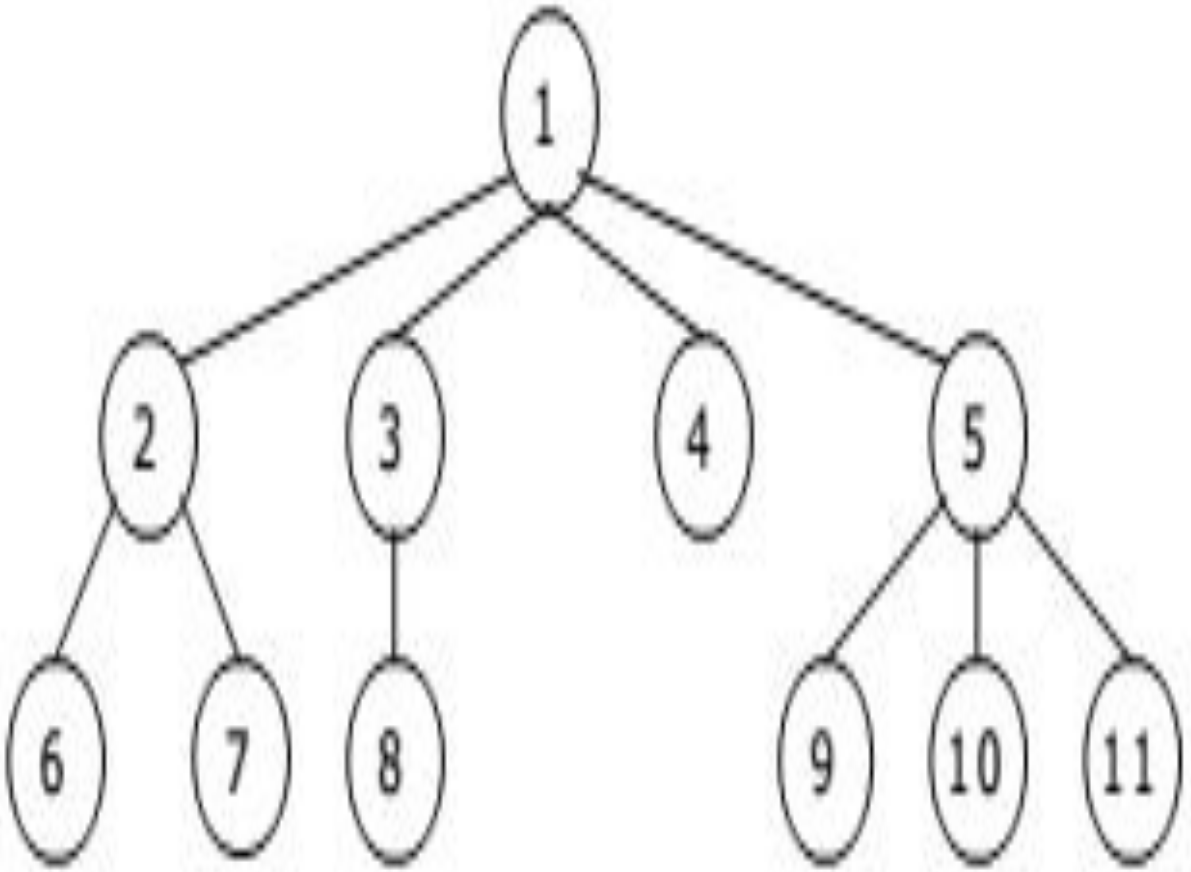
Note: If theparentnode has only theright child in the general tree then it becomes the rightmost child node of the last node following the parent node in the binary tree.

In the above example, if node B has the right child node L then in binary tree representation L would be the right child of node D.

Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

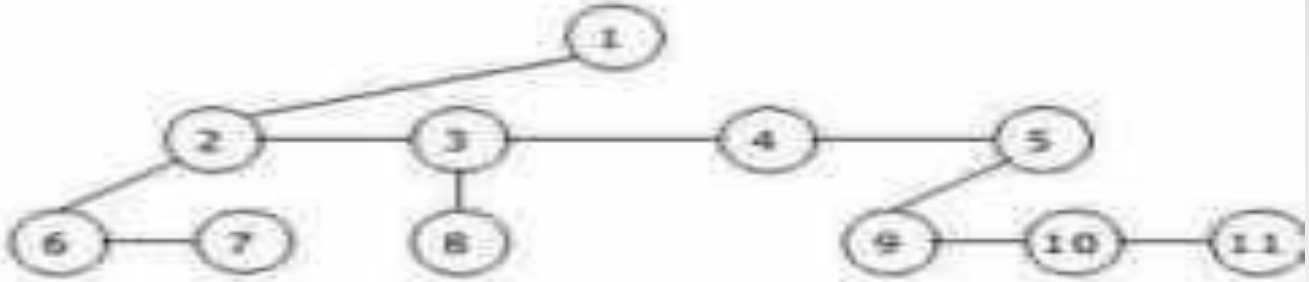
Convert the following Generic Tree to Binary Tree:



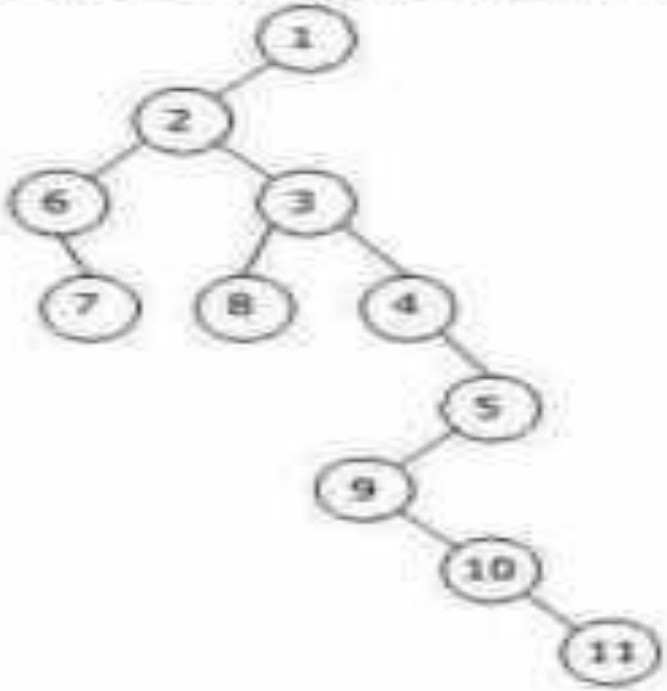
Convert a Generic Tree(N-array Tree) to Binary Tree

Solution:

Stage 1 tree by using the above mentioned procedure is as follows:



Stage 2 tree by using the above mentioned procedure is as follows:

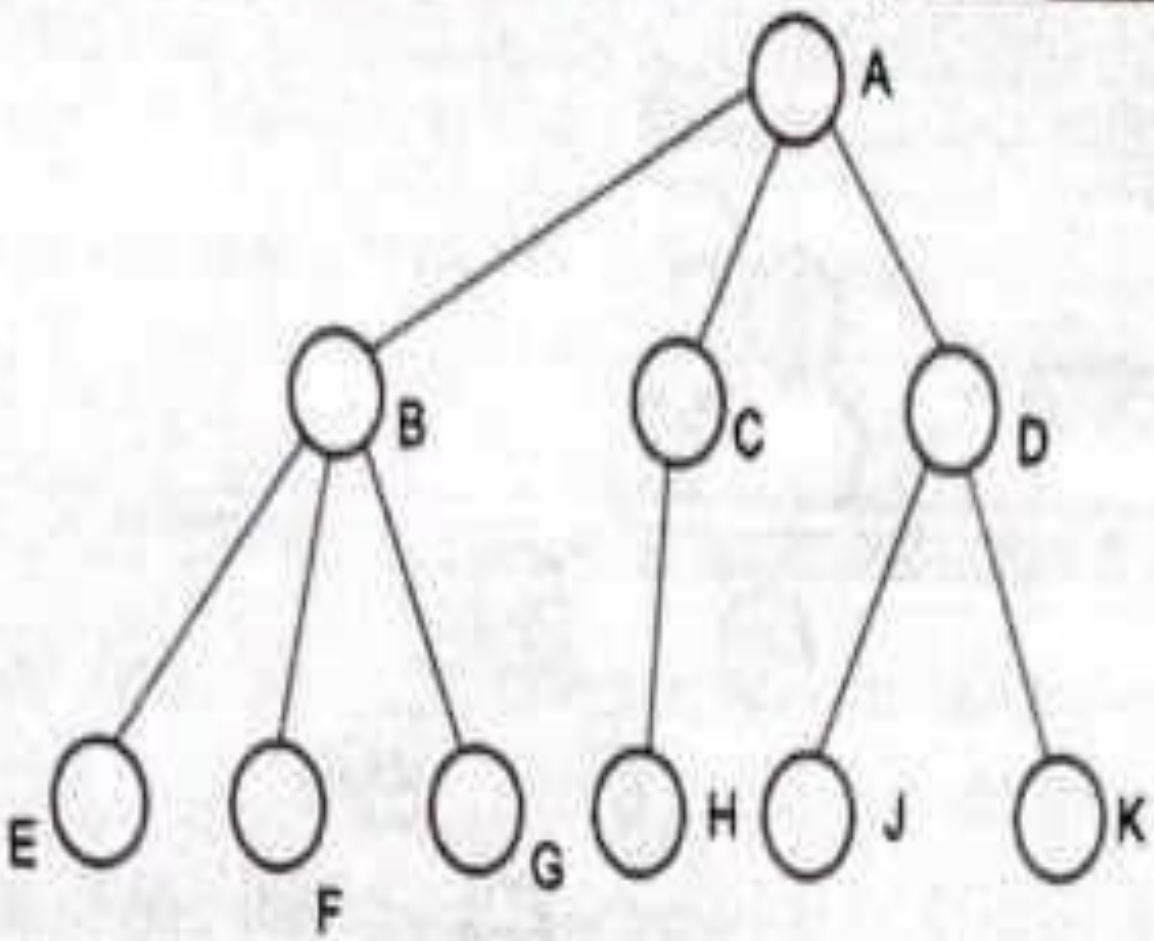


Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

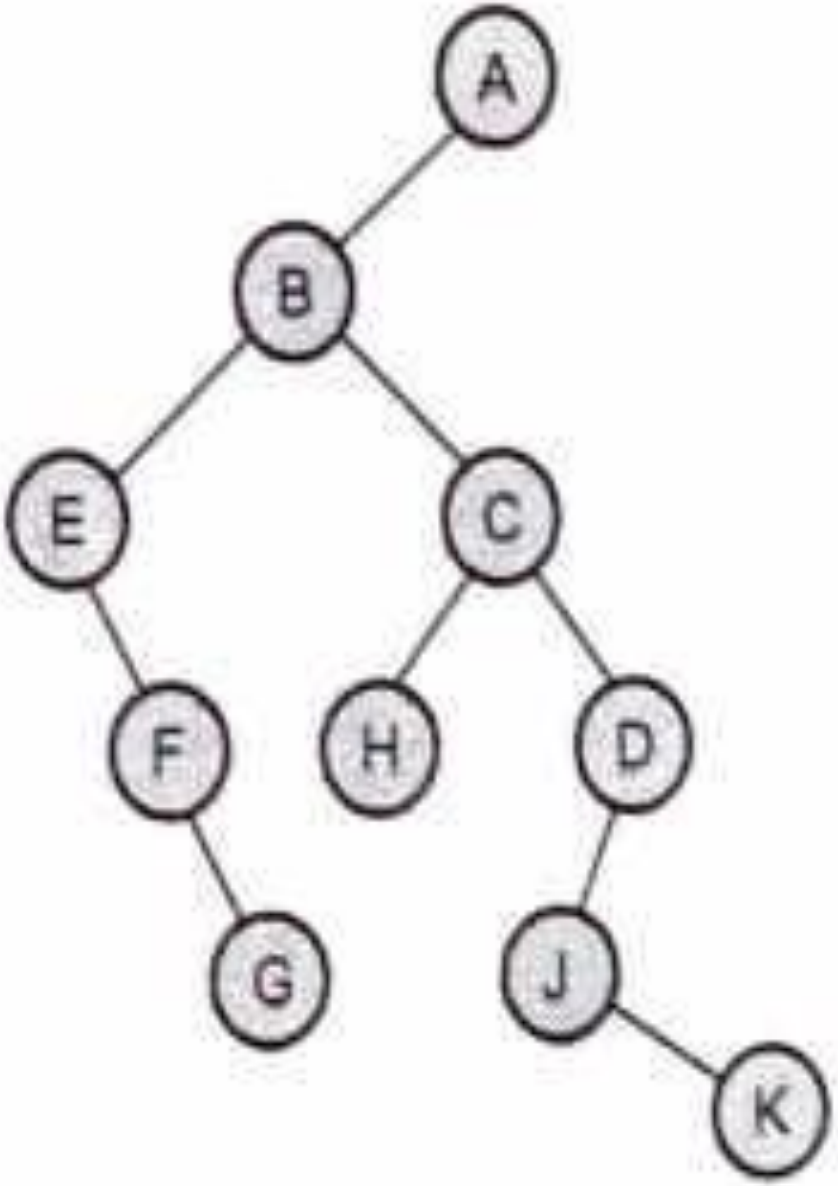
Convert the following Generic Tree to Binary Tree:

Convert following generalized tree into a inary tree. **SPPU : Dec.-13, Marks 3**



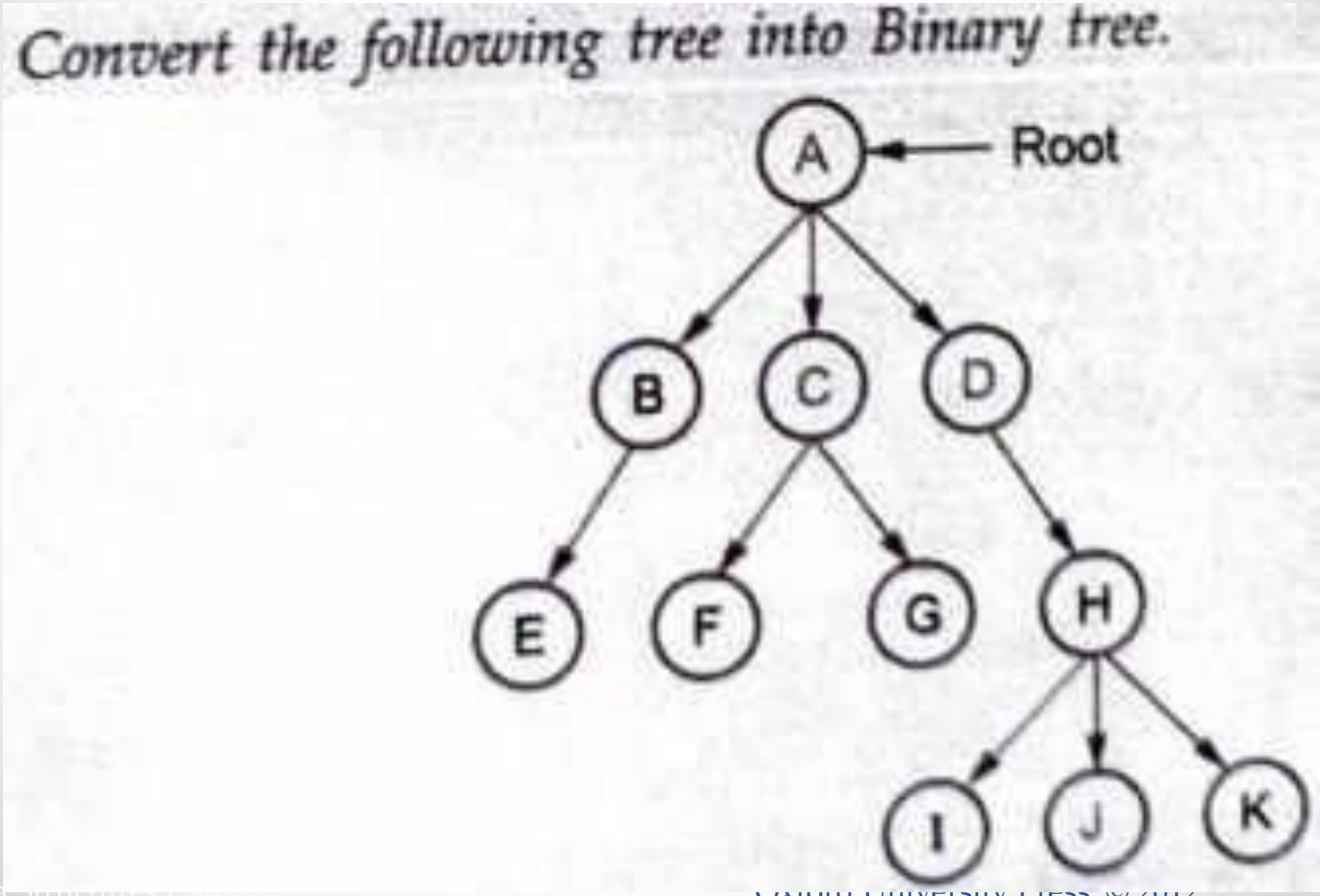
Convert a Generic Tree(N-array Tree) to Binary Tree

Solution :



Convert a Generic Tree(N-array Tree) to Binary Tree

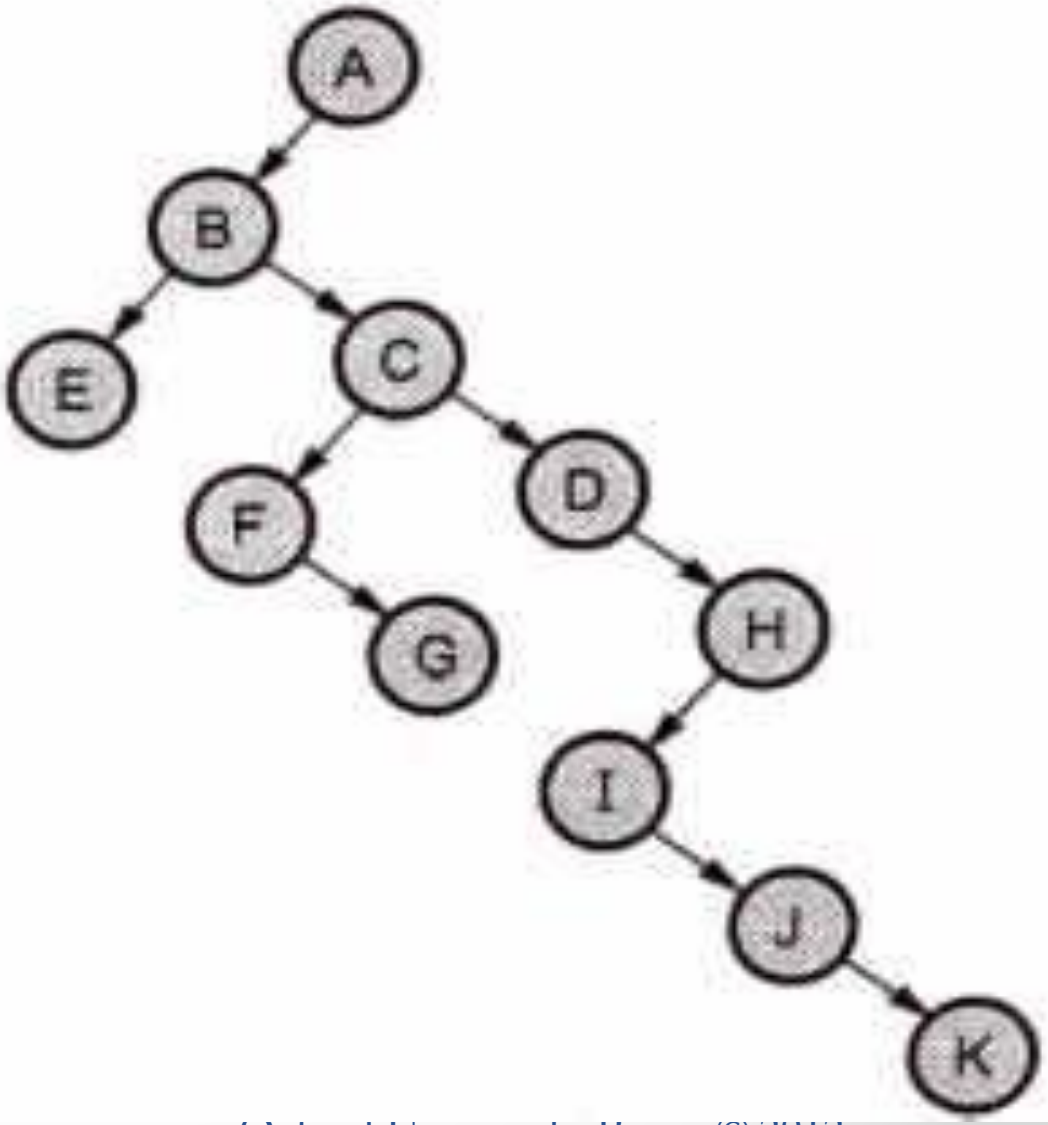
Examples:



Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

The binary tree will be -

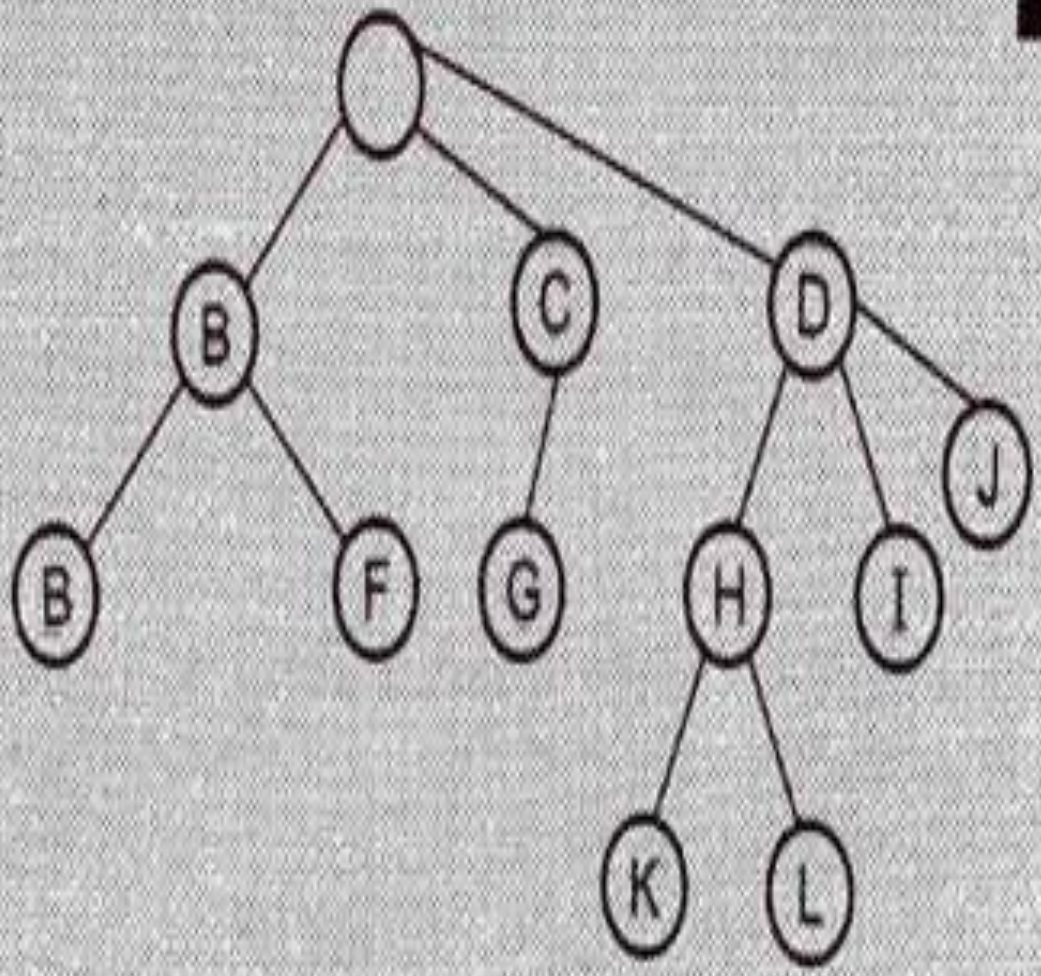


Convert a Generic Tree(N-array Tree) to Binary Tree

Examples:

Convert the given general tree to its equivalent binary tree.

SPPU : May-19, Marks 6



Tree Traversal

Traversal of the tree in data structures is a process of visiting each node and prints their value. There are three ways to traverse tree data structure.

1. Pre-order Traversal
2. In-Order Traversal
3. Post-order Traversal

Tree Traversal

Tree traversal means traversing or visiting each node of a tree. Linear data structures like Stack, Queue, linked list have only one way for traversing, whereas the tree has various ways to traverse or visit each node. The following are the three different ways of traversal:

1. Inorder traversal
2. Preorder traversal
3. Postorder traversal

Inorder Traversal

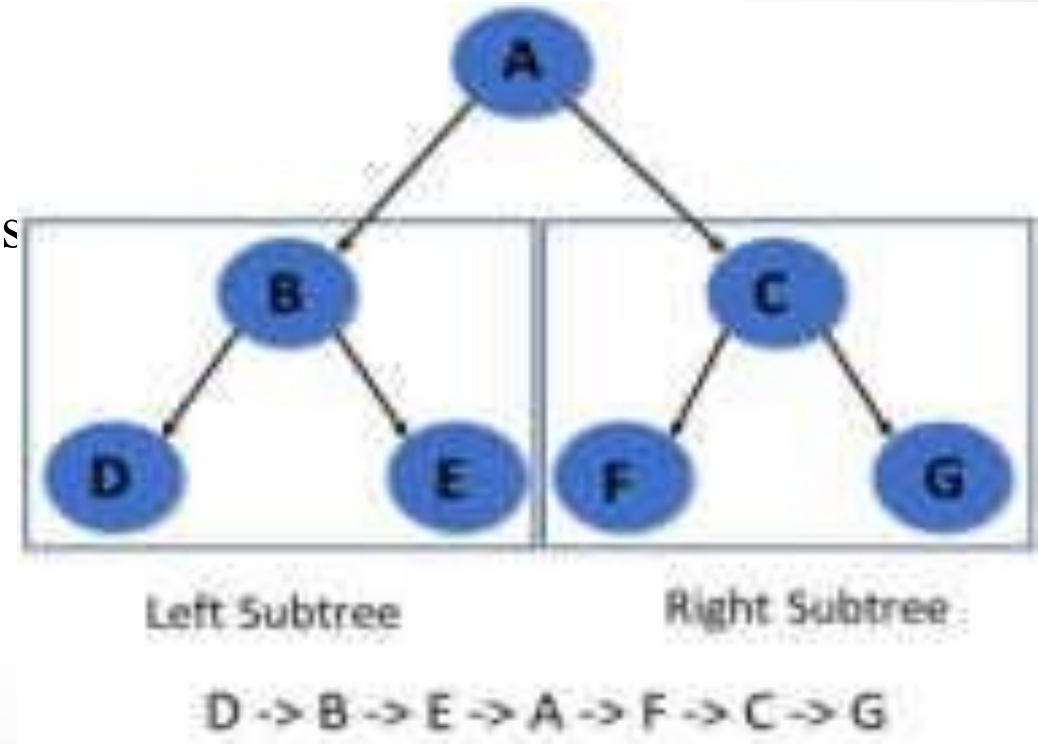
In the in-order traversal, the left subtree is visited first, then the root, and later the right subtree.

Algorithm:

Step 1- Recursively traverse the left subtree

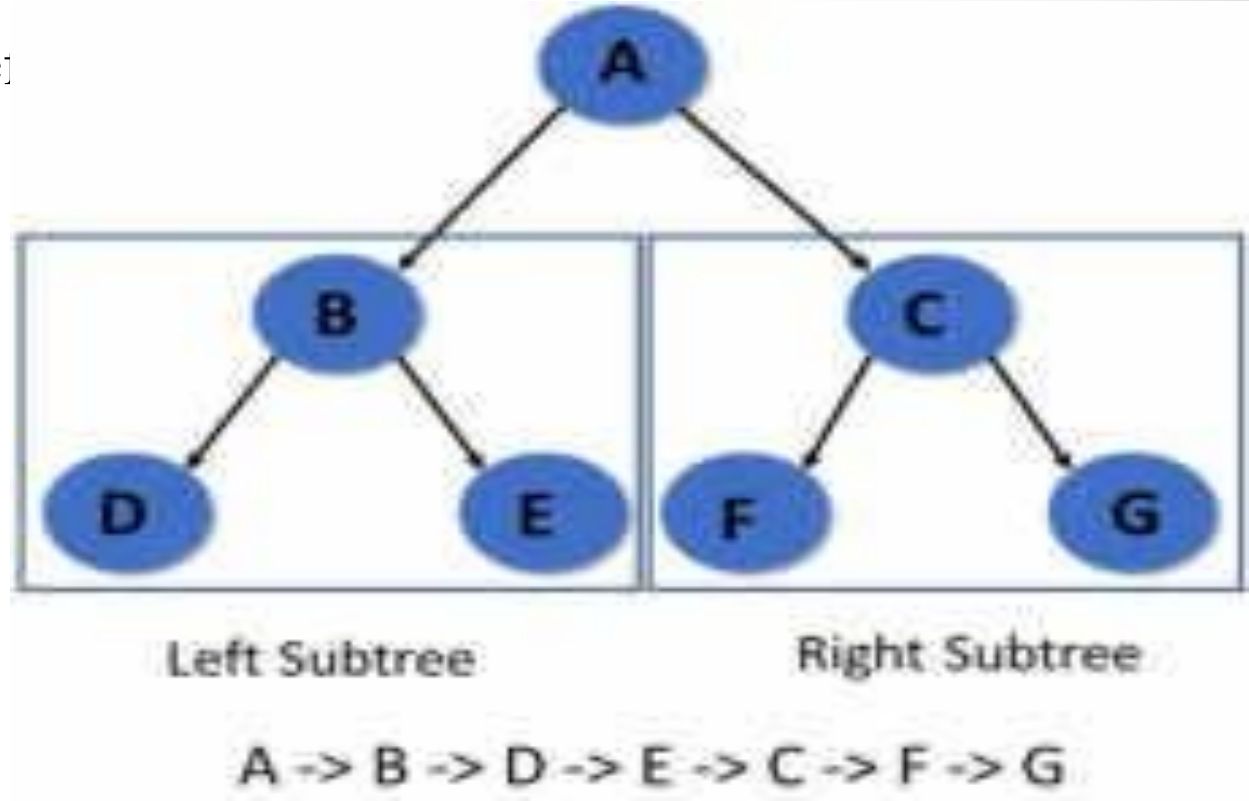
Step 2- Visit root node

Step 3- Recursively traverse right s



Pre-order Traversal

In pre-order traversal, it visits the root node first, then the le



Algorithm:

Step 1- Visit root node

Step 2- Recursively traverse the left subtree

Step 3- Recursively traverse right subtree

Post-order Traversal

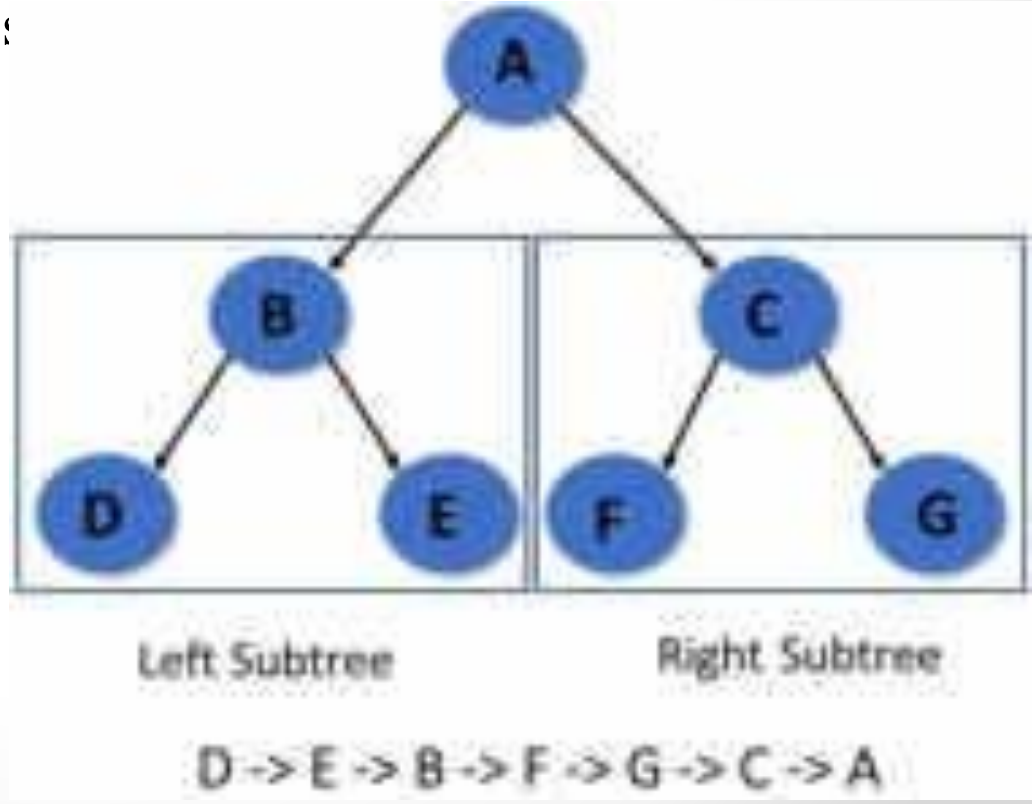
It visits the left subtree first in post-order traversal, then the right subtree, and finally the root node.

Algorithm:

Step 1- Recursively traverse the left subtree

Step 2- Recursively traverse right s

Step 3- Visit root node



Tree Traversal Example

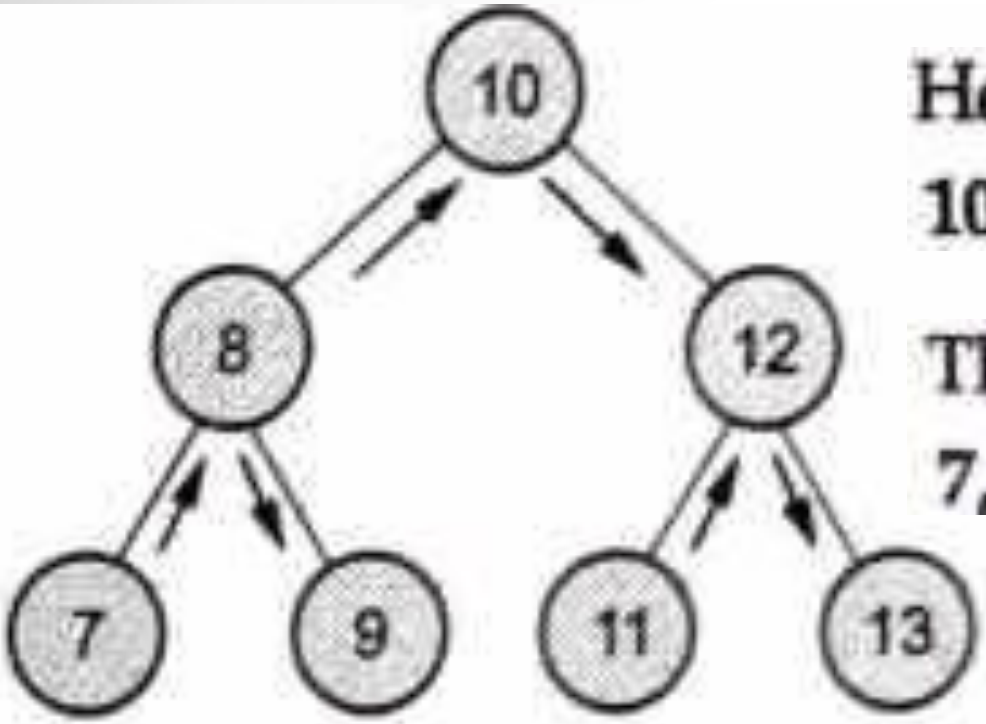


Fig. 2.6.1 Binary tree

Hence, Preorder traversal is

10, 8, 7, 9, 12, 11, 13.

The Postorder sequence is

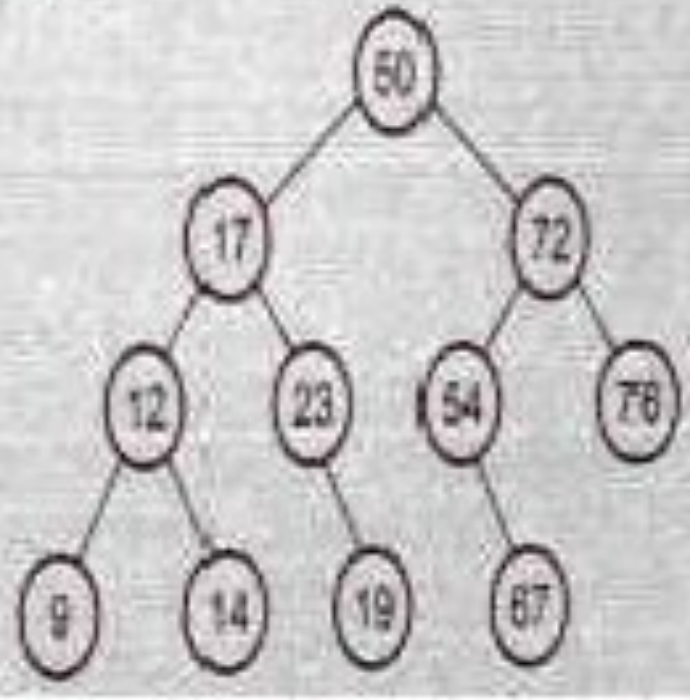
7, 9, 8, 11, 13, 12, 10.

Inorder sequence is 7, 8, 9, 10, 11, 12, 13

Tree Traversal Example

Example 2.6.1 *Consider the following tree given in the problem. Show a Postorder, Preorder and In order Traversal of tree.*

SPPU : May-14, Marks 3

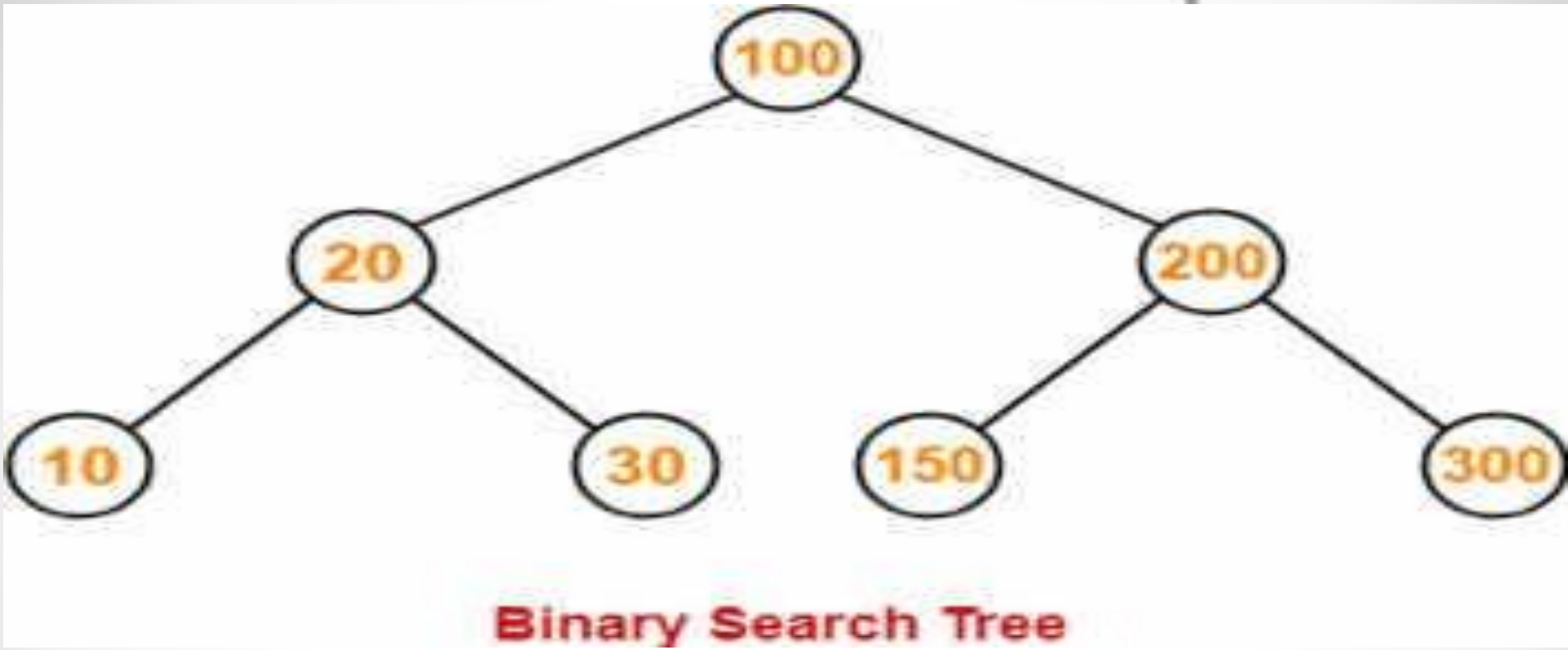


Solution : Preorder Sequence : 50, 17, 12, 9, 14, 23, 19, 72, 54, 67, 76.

Inorder Sequence : 9, 12, 14, 17, 23, 19, 50, 54, 67, 72, 76.

Postorder Sequence : 9, 14, 12, 19, 23, 17, 67, 54, 76, 72, 50.

Tree Traversal Example



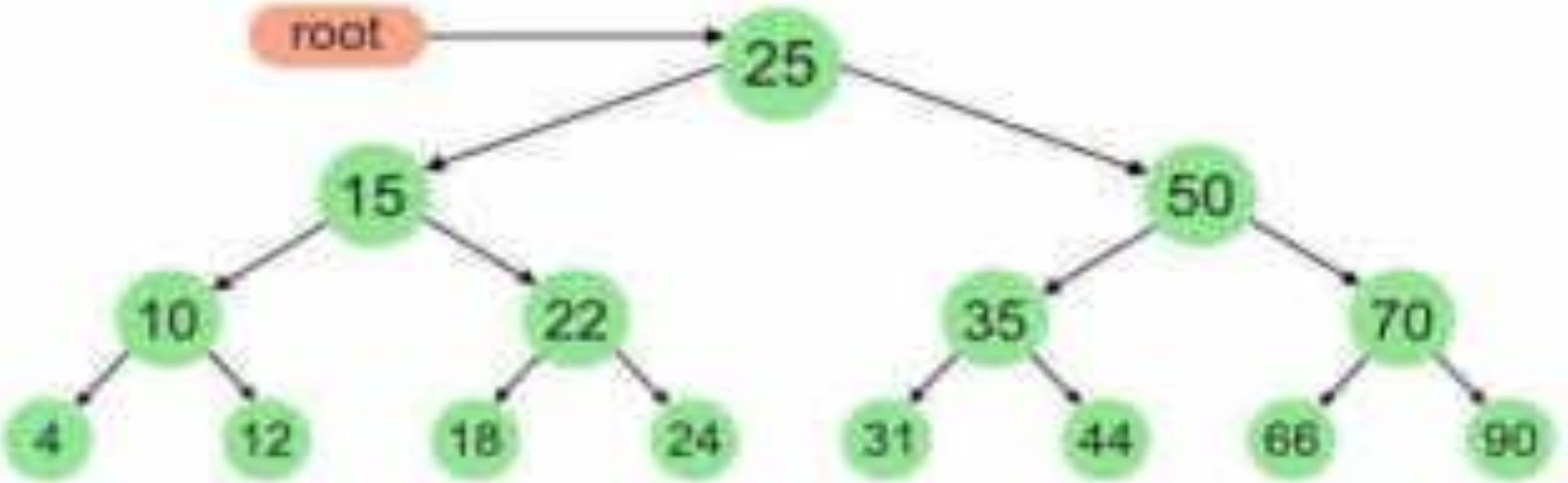
Binary Search Tree

Preorder Traversal - 100 , 20 , 10 , 30 , 200 , 150 , 300

Inorder Traversal- 10 , 20 , 30 , 100 , 150 , 200 , 300

Postorder Traversal : 10 , 30 , 20 , 150 , 300 , 200 , 100

Tree Traversal Example



InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Formation of Binary Tree from its Traversals

- Sometimes we need to construct a binary tree if its traversals are known
- From a single traversal a unique binary tree cannot be constructed
- However, if two traversals are known then the corresponding tree can be drawn uniquely

Formation of Binary Tree from its Traversals

- If the preorder traversal is given, then the first node is the root node
- If the preorder traversal is given, then the first node is the root node
- Once the root node is identified, all the nodes in all left subtrees and right subtrees of the root node can be identified
- Same techniques can be applied repeatedly to form subtrees
- We can conclude that, for the binary tree construction and traversals are essential out of which one should be inorder traversal and another preorder or postorder
- Alternatively given preorder and postorder traversals, binary tree cannot be obtained uniquely

Tree Traversal

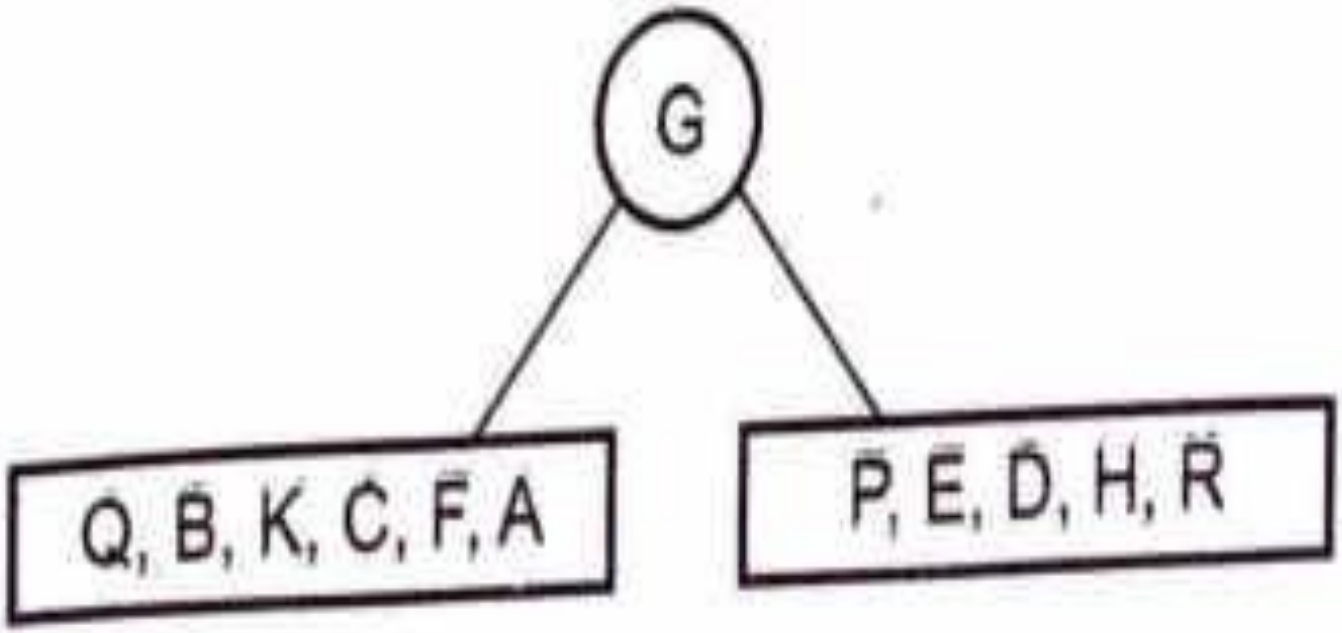
Tree Traversal Example

Example 2.6.2 *From the given traversals construct the binary tree.*

Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

In-order : Q, B, K, C, F, A, G, P, E, D, H, R

SPPU : Dec.-17, Marks 4



Tree Traversal

Tree Traversal Example

Example 2.6.2 *From the given traversals construct the binary tree.*

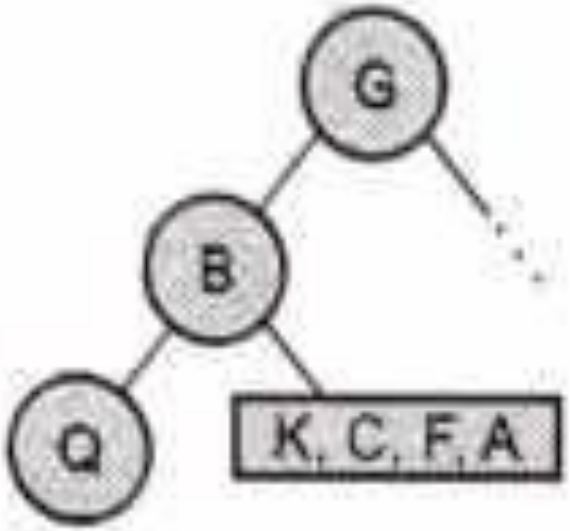
Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

In-order : Q, B, K, C, F, A, G, P, E, D, H, R

SPPU : Dec.-17, Marks 4

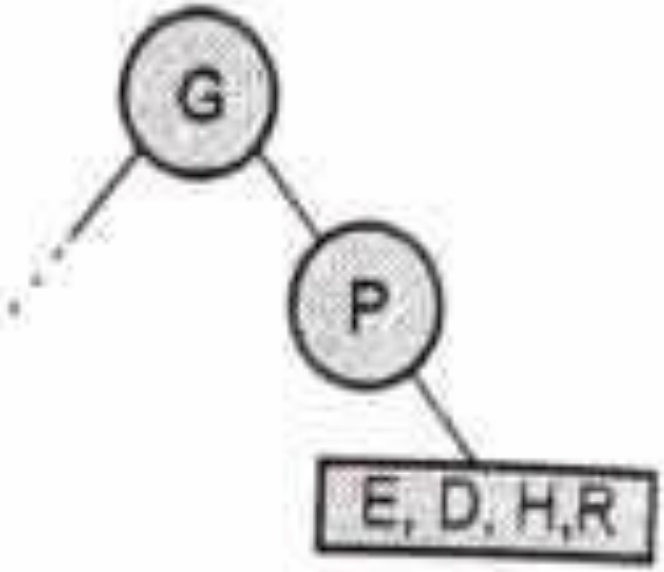
Preorder : [B], Q, A, C, K, F,

Inorder : Q, [B], K, C, F, A



Preorder : [P], D, E, R, H,

Inorder : [P], E, D, H, R,



Tree Traversal

Tree Traversal Example

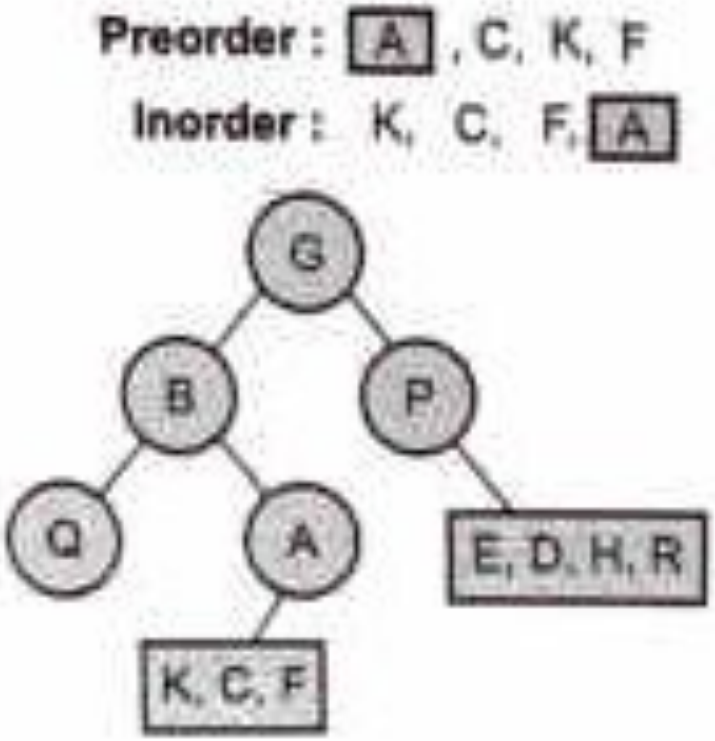
Example 2.6.2 *From the given traversals construct the binary tree.*

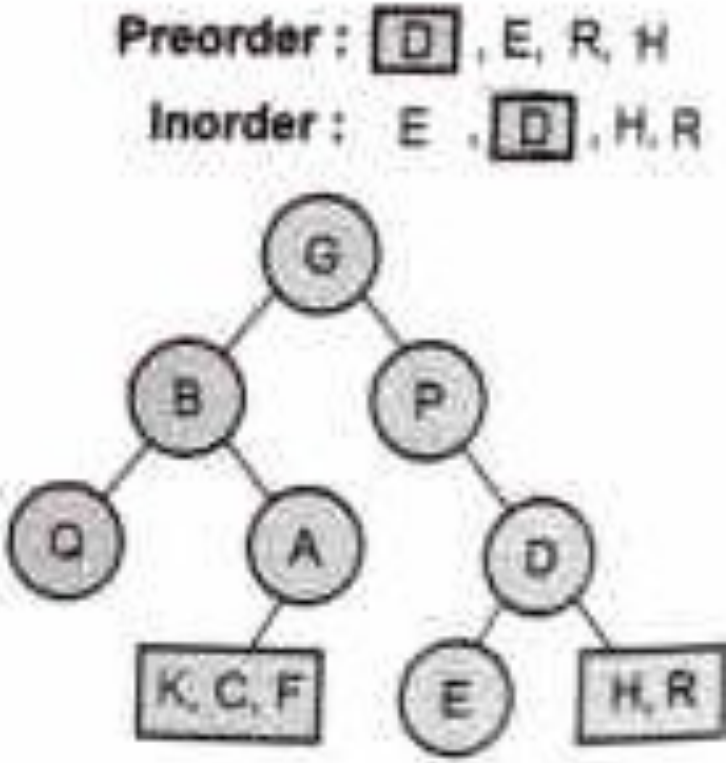
Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

In-order : Q, B, K, C, F, A, G, P, E, D, H, R

SPPU : Dec.-17, Marks 4

Step 3 :





Tree Traversal

Tree Traversal Example

Example 2.6.2 *From the given traversals construct the binary tree.*

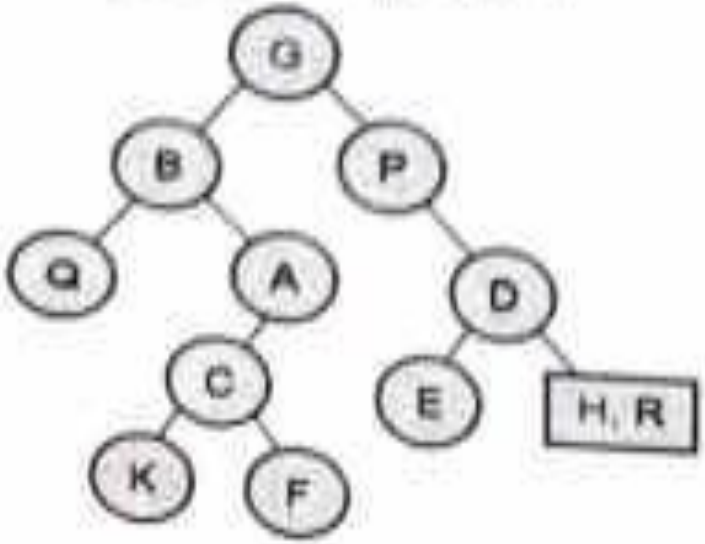
Pre-order : G, B, Q, A, C, K, F, P, D, E, R, H

In-order : Q, B, K, C, F, A, G, P, E, D, H, R

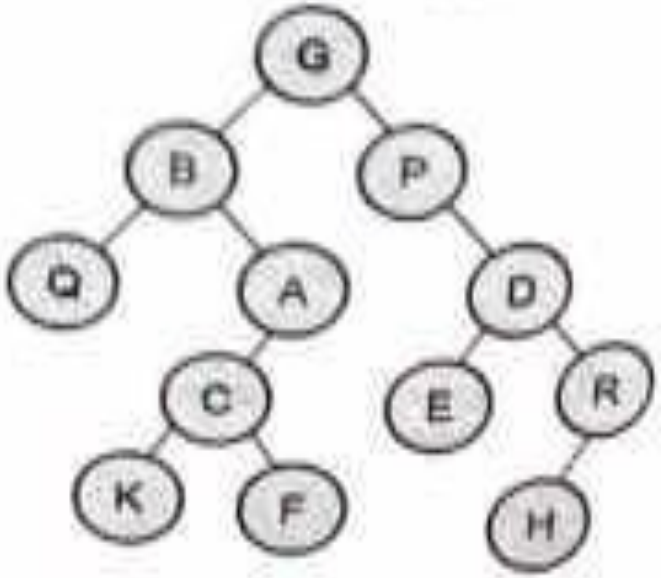
SPPU : Dec.-17, Marks 4

Step 4 :

Preorder : [C] , K, F,
Inorder : K, [C] , F,



Preorder : [R] , H,
Inorder : H, [R]



Inorder Traversal (recursive version)

```
void Binary_Tree::inorder(tree_pointer *ptr)
    {
        if(ptr!=NULL)
        {
        inorder(ptr->left_child);
        printf("%d", ptr->data);
        inorder(ptr->right_child);
        }

    }
```

Preorder Traversal (recursive version)

```
voidpreorder(tree_pointerptr)

{           if(ptr!=NULL)
            {
            printf("%d", ptr->data);
            preorder(ptr->left_child);
            predorder(ptr->right_child);
            }
}
```

Postorder Traversal (recursive version)

```
voidpostorder(tree_pointer  ptr)

{
        if (ptr!=NULL)
        {
        postorder(ptr->left_child);
        postdorder(ptr->right_child);
        printf("%d", ptr->data);
       }
}
```

Computing Time Complexity of Algorithm

LDR : 5 – 6 + 3 * 8 / 4
LRD : 5 6 – 3 8 4 / * +
DLR : + – 5 6 * 3 / 8 4
DRL : + * / 4 8 3 – 6 5
RDL : 4 / 8 * 3 + 6 – 5
RLD : 4 8 / 3 * 6 5 – +

Pre-order Traversal

- In this traversal, the root is visited first then left subtree in preorder and then right subtree in preorder
- **Tree characteristics lead to naturally implement tree traversals recursively**
- It can be defined in the following steps.

Preorder (DLR) Algorithm:

- Visit the root node
- **Traverse the left subtree of node in preorder and then**
- Traverse the right subtree of node in preorder

Pre-order Traversal

A preorder traversal of the tree in Fig. 2 visits node in a sequence: A B D E G C F

For an **Expression Tree, preorder traversal yields a prefix expression**

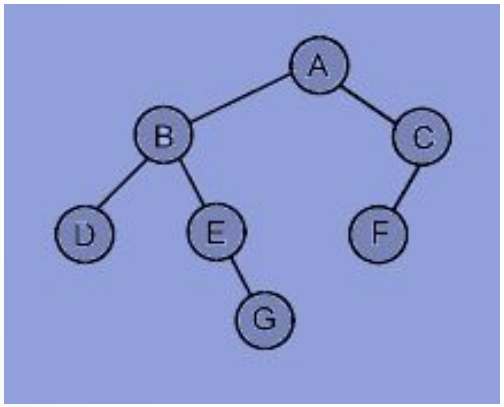


Figure 20: Binary Tree

Preorder traversal yields +*ABD

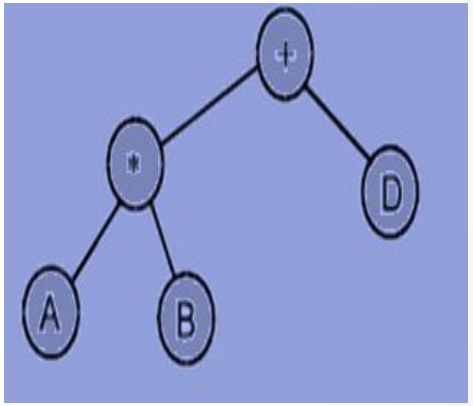


Figure 21: Expression Tree and its preorder traversal

In-order Traversal

In this traversal, the left subtree is visited first in inorder, then root and then the right subtree in inorder

In-order Traversal

Inorder ((LDR)) Algorithm

(a) Traverse the left subtree of root node in inorder
(b) Visit the root node
(c) Traverse the right subtree of root node in inorder

In-order Traversal

Inorder traversal is also called as **symmetric traversal**

Post-order Traversal

In this traversal, the left subtree is visited first in postorder, then the right subtree in postorder and then the root

Post-order Traversal Algorithm

(i) Traverse the root's left child (subtree) of root node in postorder

(ii) Traverse the root's right child (subtree) of root node in postorder

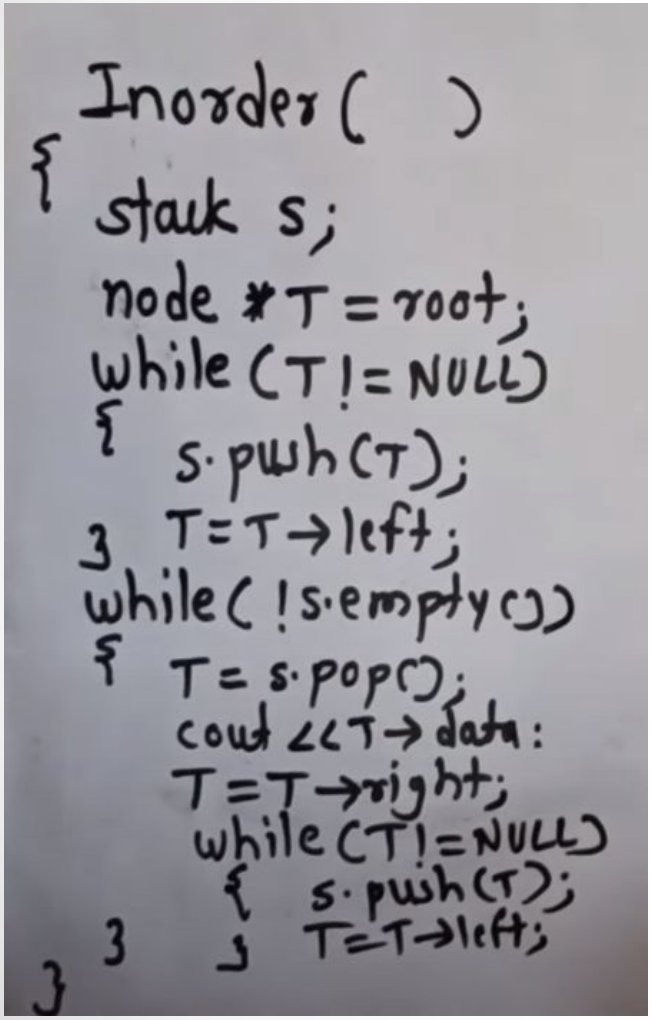
(iii) Visit the root node

Pre-order Traversal

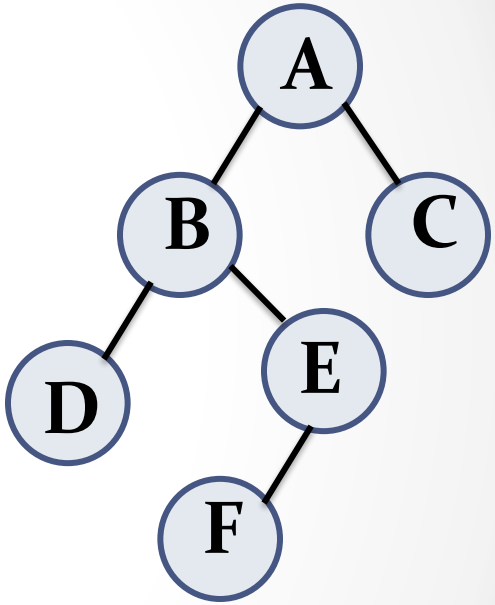
- For the expression tree in Fig. 21 the postorder traversal yields a postfix expression as : = A B * D +
- The postorder traversal says that traverse left and continue again
- When you cannot continue, move right and begin again or move back, until you can move right and visit the node

Non-recursive Implementations of Traversals

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.




```
Inorder()
{
      stack S;
      node *T=root;
      while(T!=NULL)
       {
        s.push(T);
        T -> left;
       }
       while(!s.empty())
       {
        T=s.pop();
              cout<<T ->data;
        T=T ->right;
        while(T!=NULL)
        {
              s.push(T);
              T=T->left
               }
       }
}
```

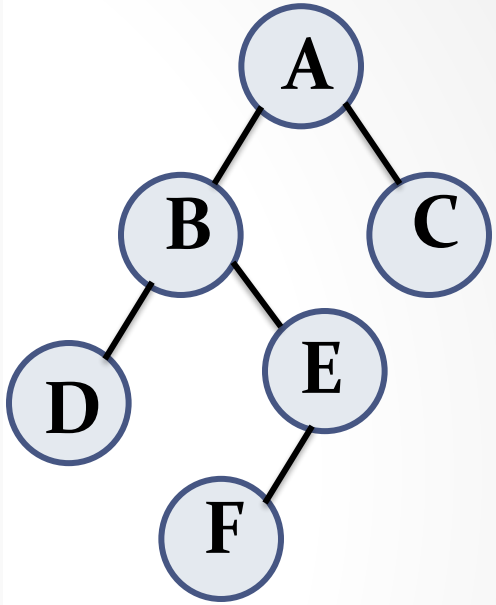


INORDER TREE TRAVERSAL WITHOUT RECURSION

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item, set current = popped_item->right
    c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

INORDER TREE TRAVERSAL WITHOUT RECURSION

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
    a) Pop the top item from stack.
    b) Print the popped item, set current = popped_item->right
    c) Go to step 3.
5) If current is NULL and stack is empty then we are done.



1. **BEGIN**
2. current=root
3. **WHILE**(1) **DO**

```
        WHILE(current!=NULL) THEN
           PUSH(current)
           current=current->left_child
        END WHILE
        IF(current==NULL and stack is not empty) THEN
        current=POP();
        VISIT(current->data)
        current=current->right
        END IF
    END WHILE
```

4. **STOP**

130

PREORDER TREE TRAVERSAL WITHOUT RECURSION

Following is a simple stack based iterative process to print Preorder traversal.

1. *Create an empty stack nodeStack and push root node to stack.*
2. *Do the following while nodeStack is not empty.*
    1. *Pop an item from the stack and print it.*
    2. *Push right child of a popped item to stack*
    3. *Push left child of a popped item to stack*

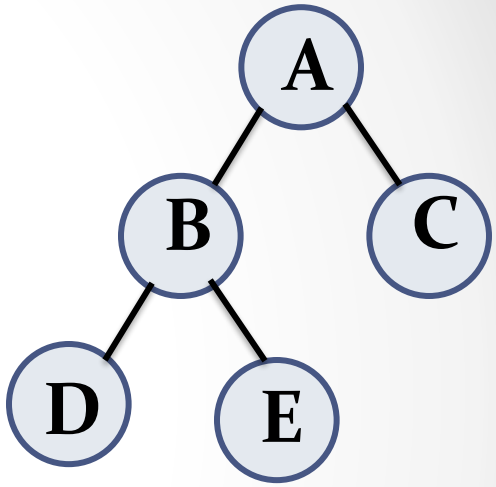
The right child is pushed before the left child to make sure that the left subtree is processed first.

PREORDER TREE TRAVERSAL WITHOUT RECURSION

Following is a simple stack based iterative process to print Preorder traversal.

1. *Create an empty stack nodeStack and push root node to stack.*
2. *Do the following while nodeStack is not empty.*
    1. *Pop an item from the stack and print it.*
    2. *Push right child of a popped item to stack*
    3. *Push left child of a popped item to stack*

The right child is pushed before the left child to make sure that the left subtree is processed first.



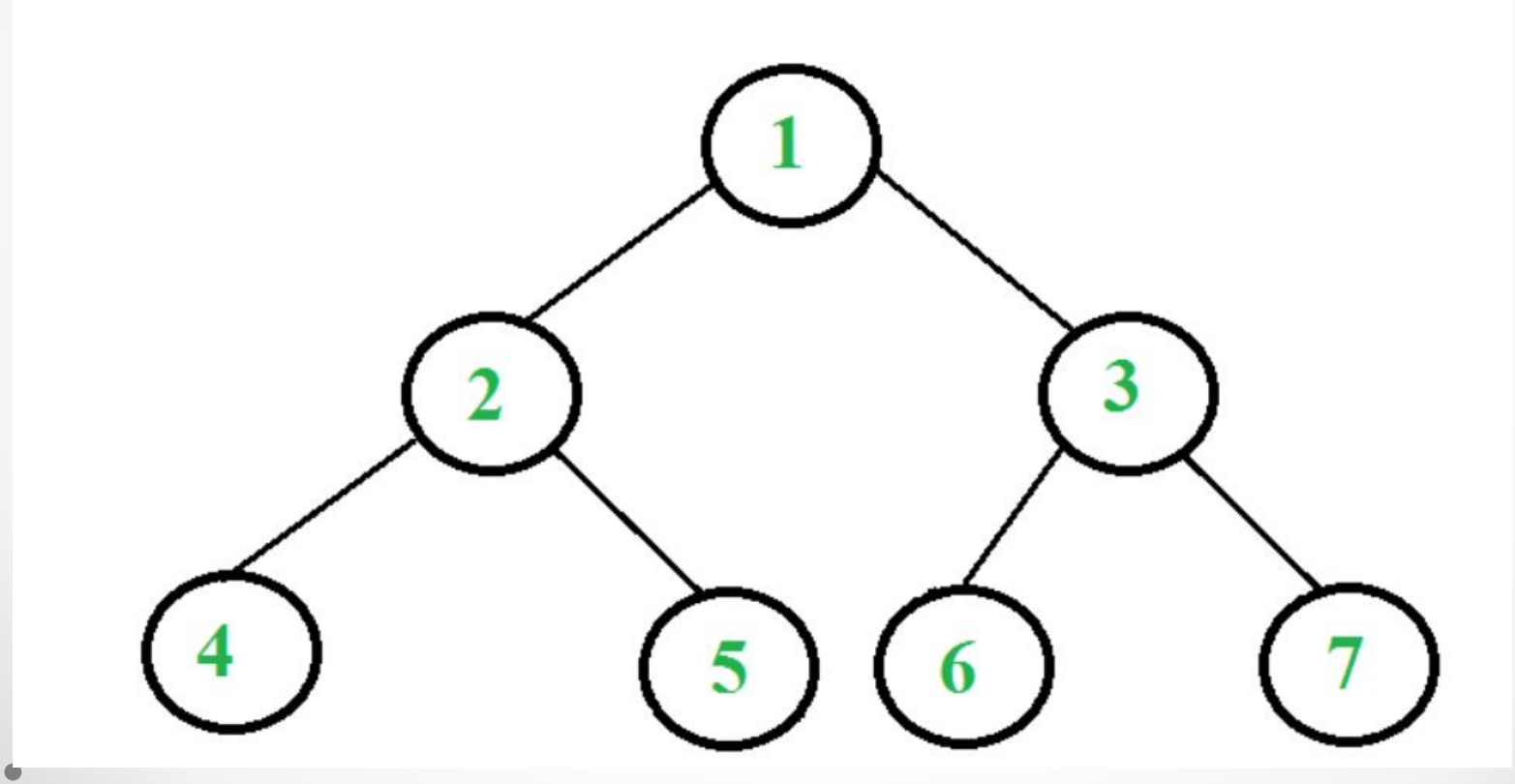
1. **BEGIN**
2. current=root
3. PUSH(current)
4. **WHILE**(stack not empty) **DO**
   current=POP();
   DISPLAY(current->data)
   PUSH(current->right)
   PUSH(current->left)
   END WHILE
5. **STOP**

POSTORDER TREE TRAVERSAL WITHOUT RECURSION

- Postorder traversal can easily be done using two stacks.
- The idea is to push reverse postorder traversal to a stack.
- Once we have the reversed postorder traversal in a stack, we can just pop all items one by one from the stack and print them;
- This order of printing will be in postorder because of the LIFO property of stacks.
- The second stack is used for this purpose.

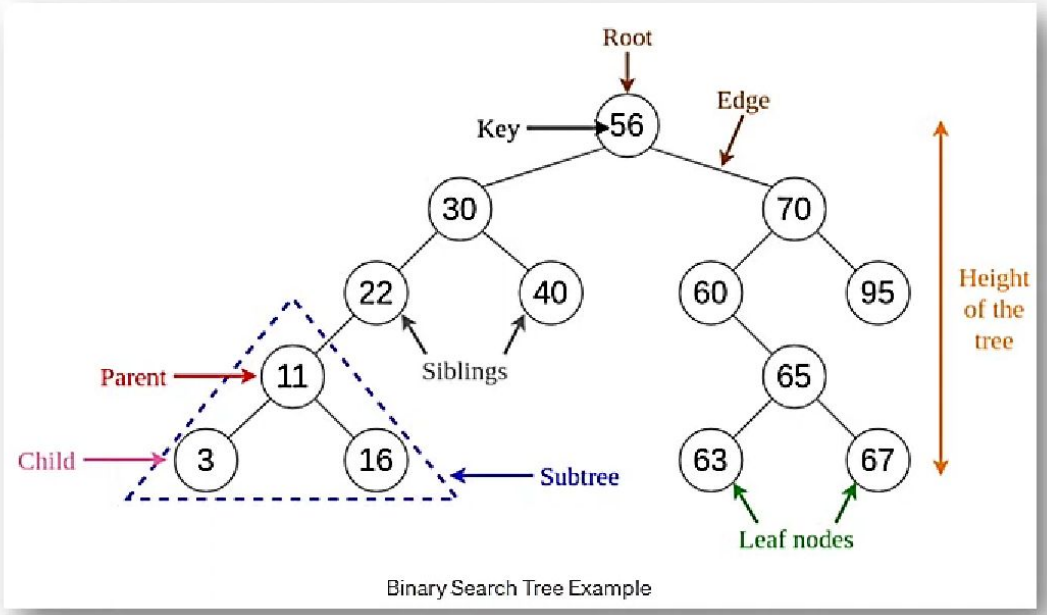
POSTORDER TREE TRAVERSAL WITHOUT RECURSION

1. Push root to first stack.
2. Loop while first stack is not empty
   2.1 Pop a node from first stack and push it to second stack
   2.2 Push left and right children of the popped node to first stack
3. Print contents of second stack



Course Outline

1. Why do we need Binary Search Tree?
2. Properties of Binary Search Tree
3. Difference Between BT & BST
4. Examples of BST
5. Operations on BST
6. Complexity of BST
7. Applications of BT
8. Important Questions



Binary Search Tree Example

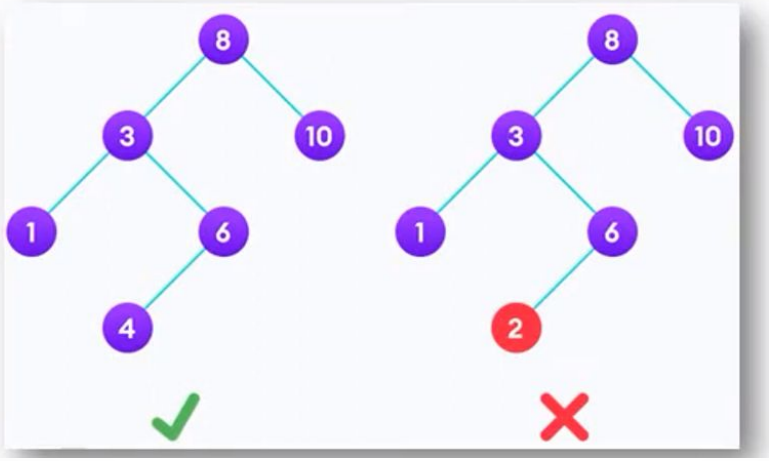
Why do we need Binary Search Tree?

- The two major factors that make binary search tree an optimum solution to any real-world problems are Speed and Accuracy.
- Binary search is in a branch-like format with parent-child relations, the algorithm knows in which location of the tree the elements need to be searched.
- This decreases the number of key-value comparisons the program has to make to locate the desired element.
- Additionally, in case the element to be searched greater or less than the parent node, the node knows which tree side to search for.
- The reason is, the left sub-tree is always lesser than the parent node, and the right sub-tree has values always equal to or greater than the parent node.
- BST is commonly utilized to implement complex searches, robust game logics, auto-complete activities, and graphics.
- The algorithm efficiently supports operations like search, insert, and delete.

Properties of Binary Search Trees

The properties that separate a binary search tree from a regular binary tree is:

1. **All nodes of left sub tree are less than the root node**
2. **All nodes of right sub tree are more than the root node**
3. Both sub trees of each node are also BSTs i.e. they have the above two properties



Difference Between BT & BST

A binary tree is simply a tree in which each node can have at most two children.

A binary search tree is a binary tree in which the nodes are assigned values, with the following restrictions :

1. No duplicate values.
2. The left subtree of a node can only have values less than the node
3. The right subtree of a node can only have values greater than the node and recursively defined
4. The left subtree of a node is a binary search tree.
5. The right subtree of a node is a binary search tree.

Binary tree vs Binary search

Points	Binary Tree	Binary Search Tree
Definit ion	A Binary Tree is a non-linear data structure in which a node can have 0, 1 or 2 nodes. Individually, each node consists of a left pointer, right pointer and data element.	A Binary Search Tree is an organized binary tree with a structured organization of nodes. Each subtree must also be of that particular structure.
Struc t ure	There is no required organization structure of the nodes in the tree.	The values of left subtree of a particular node should be lesser than that node and the right subtree values should be greater.

Binary tree vs Binary search tree

Points	Binary Tree	Binary Search Tree
Operations Performed	The operations that can be performed are deletion, insertion and traversal	As these are sorted binary trees, they are used for fast and efficient binary search, insertion and deletion.
Types	There are several types. Most common ones are the Complete Binary Tree, Full Binary Tree, Extended Binary Tree	The most popular ones are AVL Trees, Splay Trees, Tango Trees, T-Trees.

Example of Binary Search Tree

Construct a Binary Search Tree (BST) for the following sequence of numbers-

50, 70, 60, 20, 90, 10, 40, 100

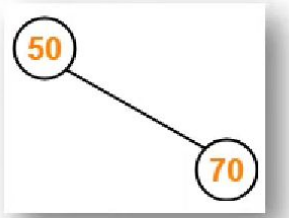
When elements are given in a sequence,

- Always consider the first element as the root node.
- Consider the given elements and insert them in the BST one by one.

Insert 50-

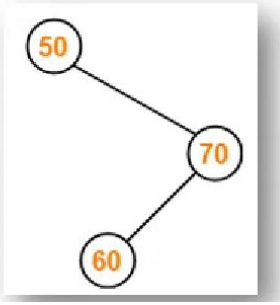
Insert 70-

- As 70 > 50, so insert 70 to the right of 50.



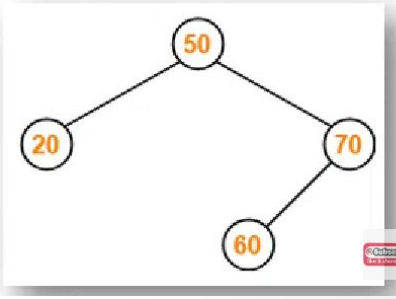
Insert 60-

- As 60 > 50, so insert 60 to the right of 50.
- As 60 < 70, so insert 60 to the left of 70.



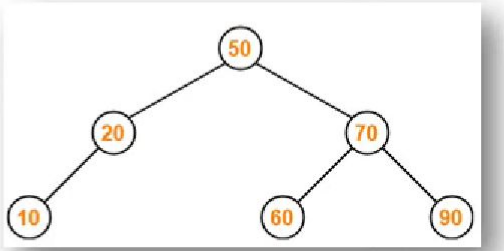
Insert 20-

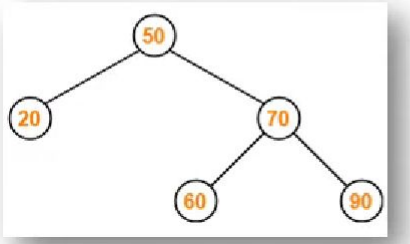
- As 20 < 50, so insert 20 to the left of 50.



Insert 90-

- As 90 > 50, so insert 90 to the right of 50.
- As 90 > 70, so insert 90 to the right of 70.



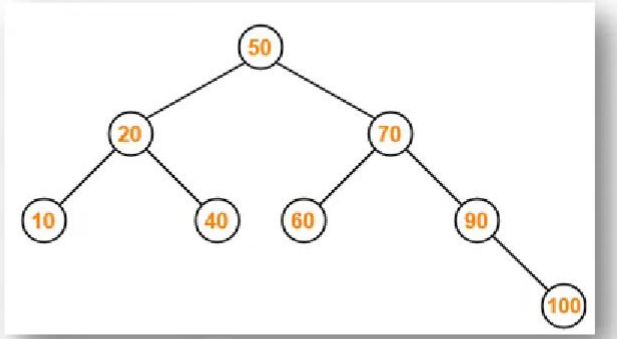


Insert 100-

- As 100 > 50, so insert 100 to the right of 50.
- As 100 > 70, so insert 100 to the right of 70.
- As 100 > 90, so insert 100 to the right of 90.

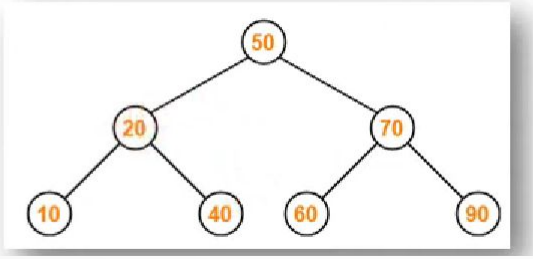
Insert 40-

- As 40 < 50, so insert 40 to the left of 50.
- As 40 > 20, so insert 40 to the right of 20.



Insert 10-

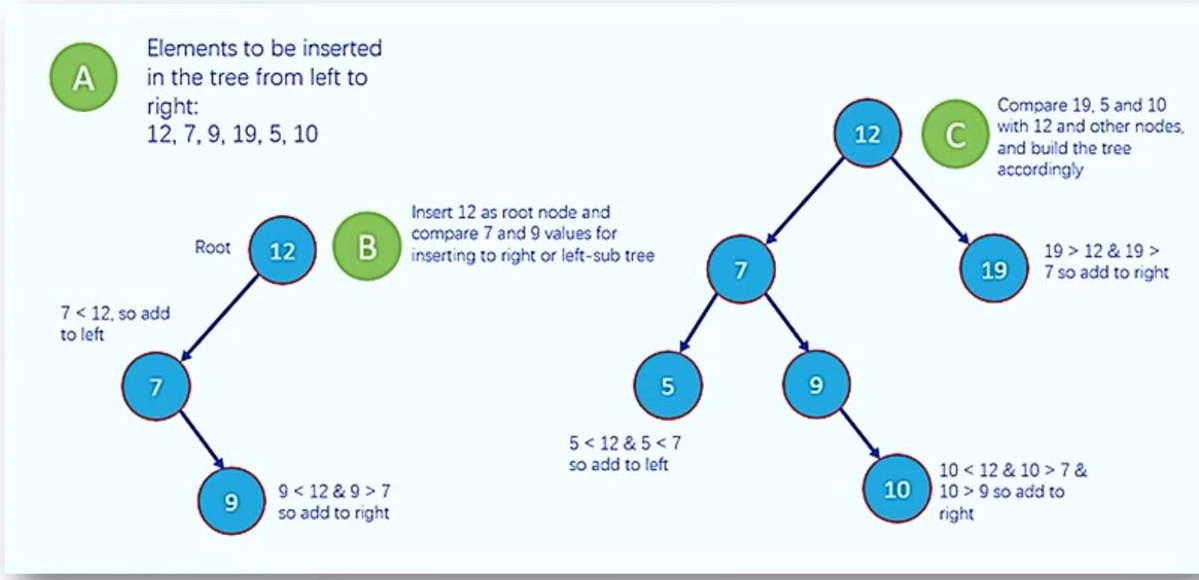
- As 10 < 50, so insert 10 to the left of 50.
- As 10 < 20, so insert 10 to the left of 20.



Operations Perform on Binary Search Tree

1. Insert Operation:

```
BST* BST ::Insert(BST* root, int value)
{
   if (!root)
      return new BST(value);
   }
   if (value > root->data)
   {
   root->right = Insert(root->right, value);
   }
   else
   {
      root->left = Insert(root->left, value);
   }
  return r
```

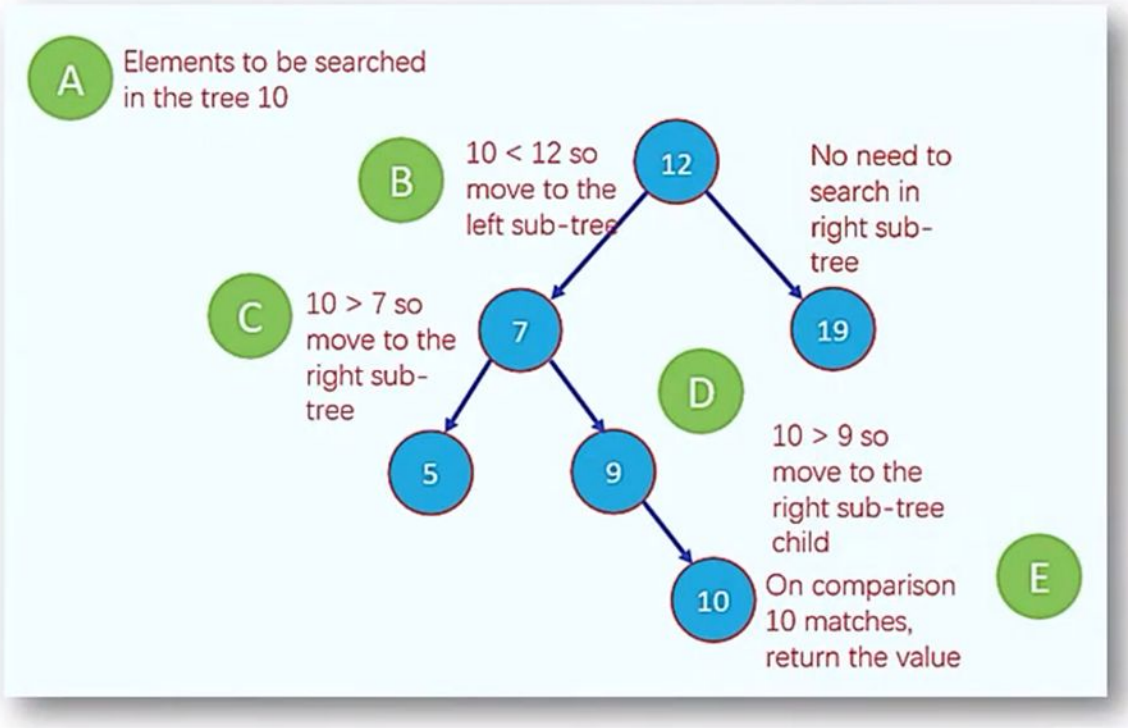


Binary Search Tree (BST)

2. Search Operation:

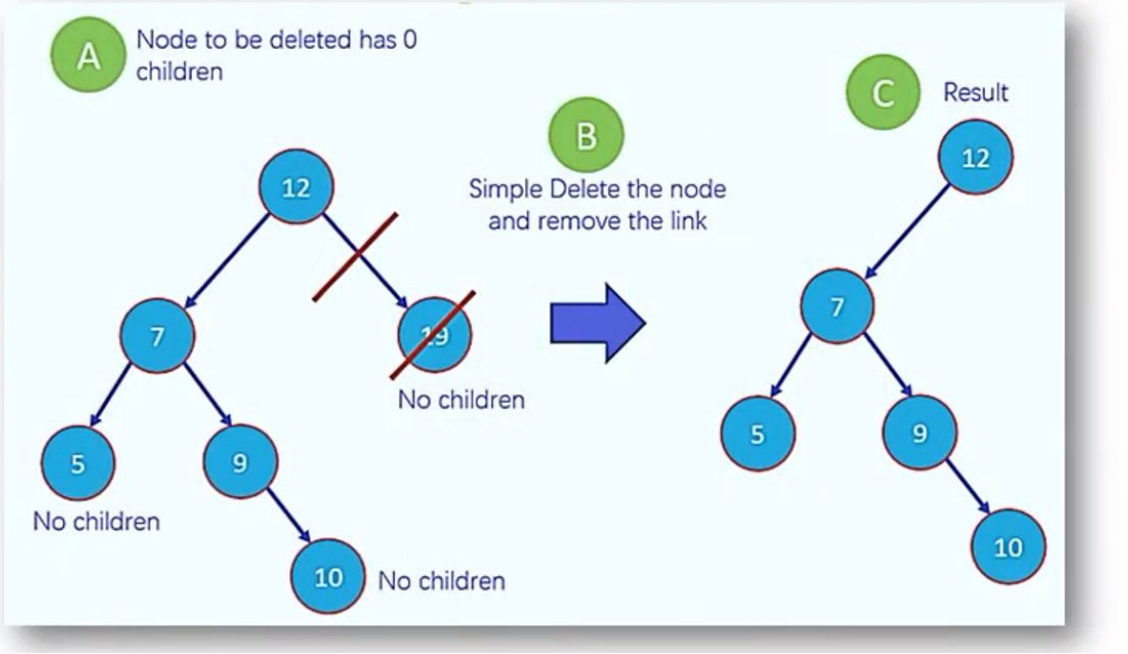
```
struct node* search(struct node* root, int key)
{
   if ( root->key == key)
   return root;

   if (root->key < key)
   return search(root->right, key);
  else
   return search(root->left, key);
}
```



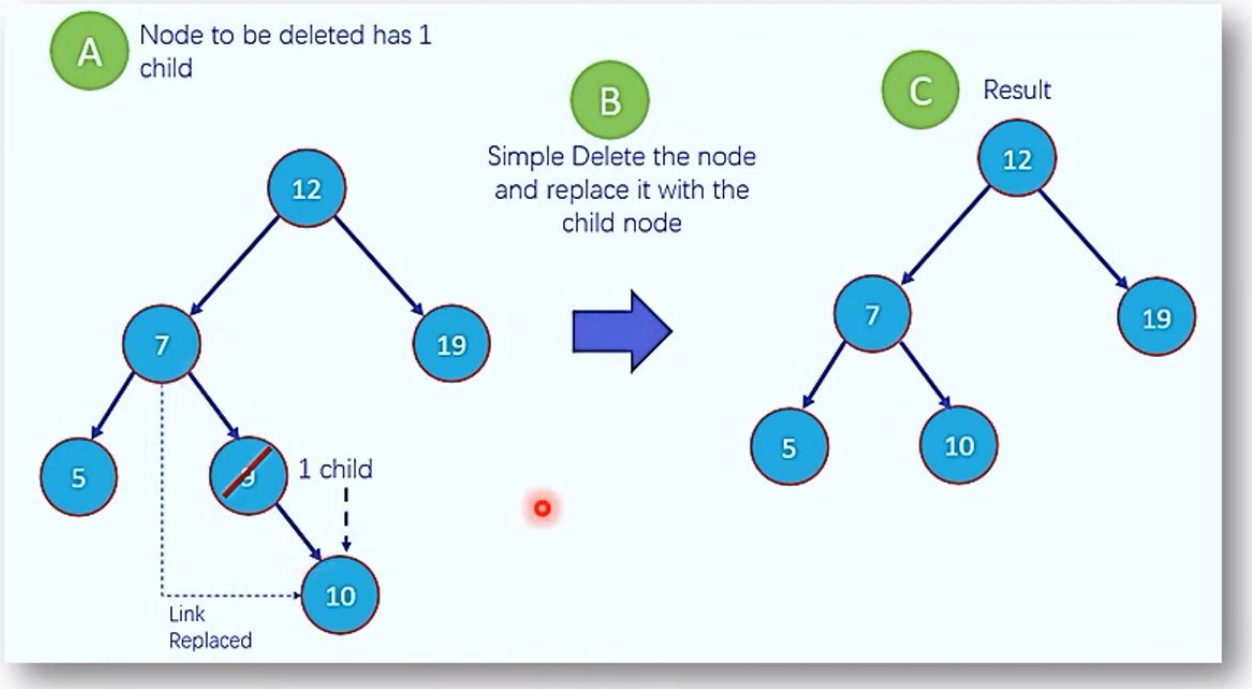
3. Delete Operation:

- **Case 1: Node with zero children:** This is the easiest situation, you just need to delete the node which has no further children on the right or left.



3. Delete Operation:

- **Case 2 - Node with one child:** Once you delete the node, simply connect its child node with the parent node of the deleted value.



Operations Perform on Binary Search Tree

3. Delete Operation:

- **Case 3: Node with two children:** This is the most difficult situation, and it works on the following two rules.

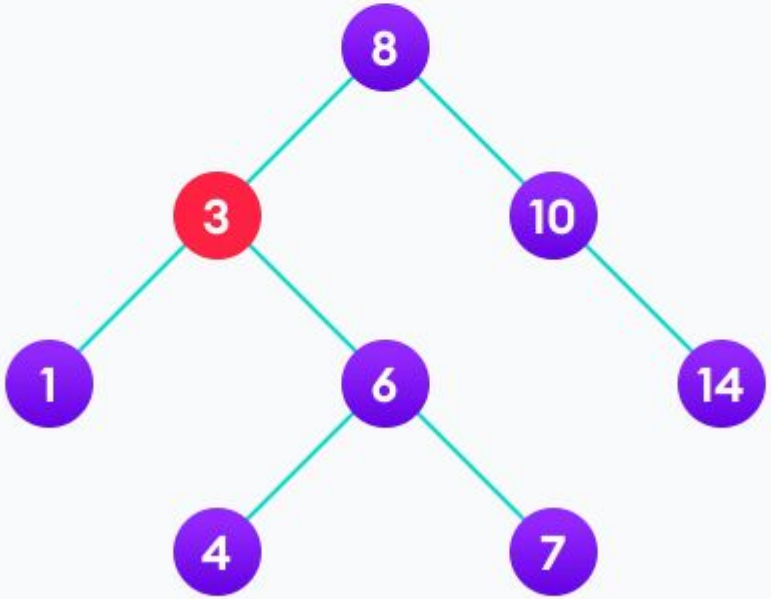
✓**3a - In Order Predecessor**:

✓**3b - In Order Successor**:

Case III

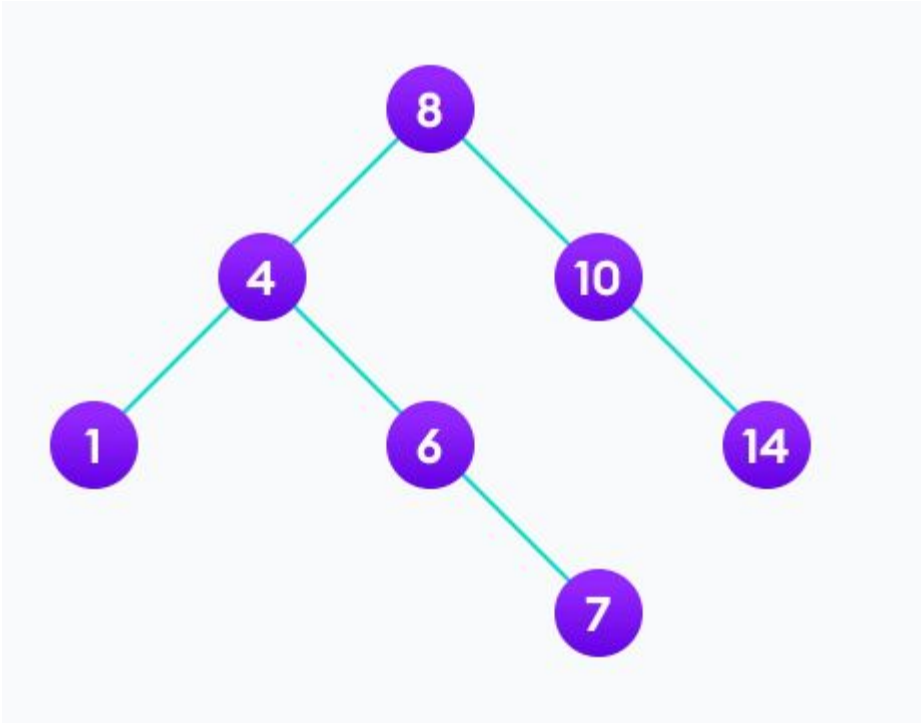
In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.



3 is to be deleted

Binary Search Tree (BST)



Complexity of BST

Operation	Average	Worst Case	Best Case
Search	O(log n)	O(n)	O(1)
Insertion	O(log n)	O(n)	O(1)
Deletion	O(log n)	O(n)	O(1)

Applications of Binary Search Tree

- Binary Search Tree: Used in *many* search applications where data is constantly entering/leaving, such as the map and set objects in many languages' libraries.
- Binary Space Partition: Used in almost every 3D video game to determine what objects need to be rendered.
- Binary Tries: Used in almost every high-bandwidth router for storing router-tables.
- Heaps: Used in implementing efficient priority-queues. Also used in heap-sort.
- Huffman Coding Tree:- used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.
- GGM Trees - Used in cryptographic applications to generate a tree of pseudo-random numbers.
- Syntax Tree: Constructed by compilers and (implicitly) calculators to parse expressions.

Equality Test

Following are the operations commonly performed on binary search tree

- Searching a key
- Inserting a key
- Deleting a key
- Traversing a tree

Breadth and Depth First Traversals

- As defined earlier, we know that traversal of tree means visiting through the nodes of a tree
- A traversal in which the node is visited before its children are visited is called a **breadth first traversal; a walk where the children are visited prior to** the parent is called a **depth first traversal**

STUDY EXAMPLES FROM THE NOTEBOOK

Depth-first Traversal

- We have already seen a few ways to traverse the elements of a tree
- A traversal in which children are visited (operated on) before the parent is called **Depth First Traversal**

DFS Algorithms

Depth first search (DFS) algorithm starts with the initial node of the graph G, and then goes to deeper and deeper until we find the goal node or the node which has no children. The algorithm, then backtracks from the dead end towards the most recent node that is yet to be completely unexplored.

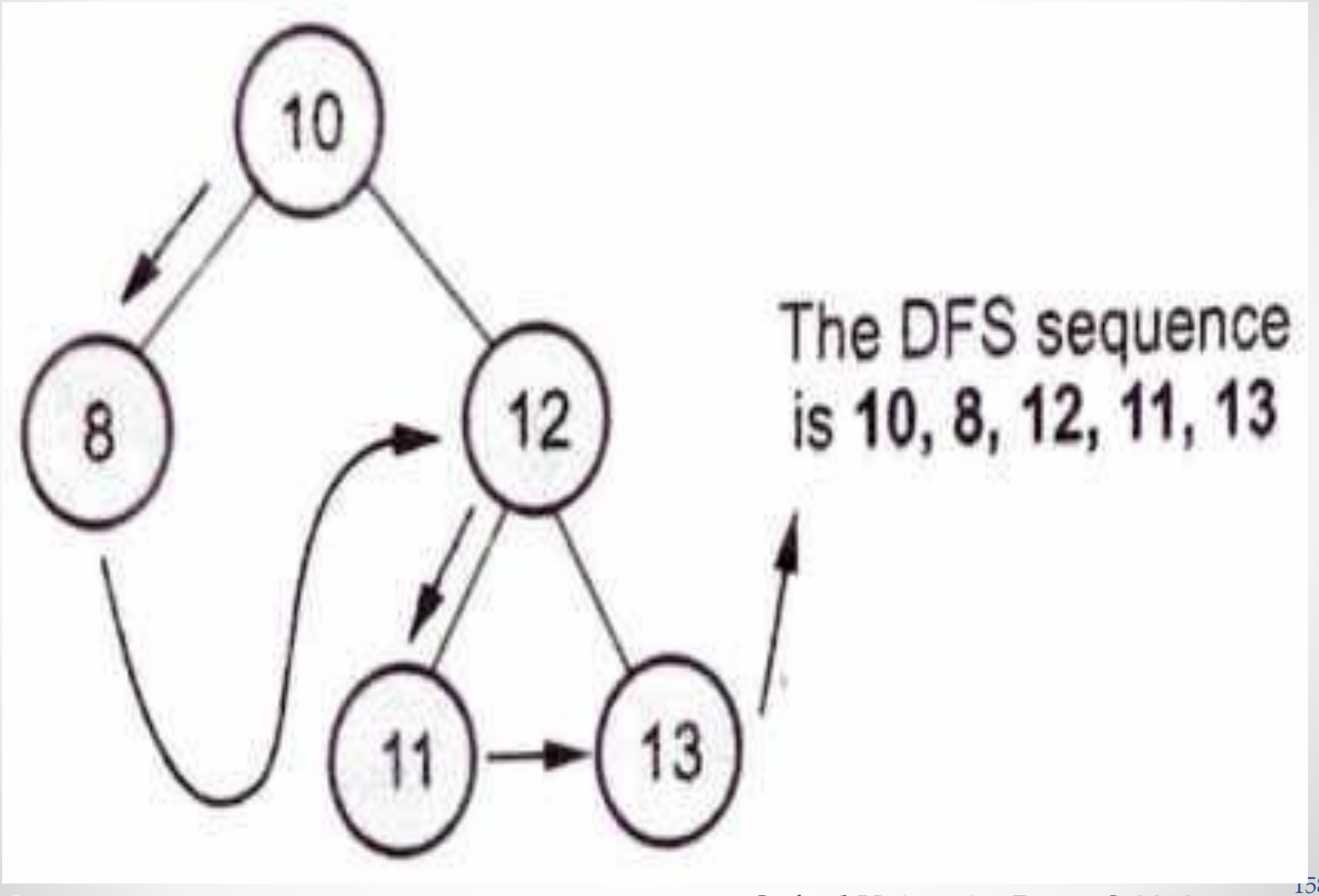
The data structure which is being used in DFS is stack. The process is similar to BFS algorithm. In DFS, the edges that leads to an unvisited node are called **discovery edge**s while the edges that leads to an already visited node are called **block edges**.

DFS Algorithms

DFS is an algorithm for finding or traversing graphs or trees in depth-ward direction. The execution of the algorithm begins at the root node and explores each branch before backtracking. It uses a stack data structure to remember, to get the subsequent vertex, and to start a search, whenever a dead-end appears in any iteration.

The full form of DFS is Depth-first search.

DFS Algorithms



DFS Algorithms

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

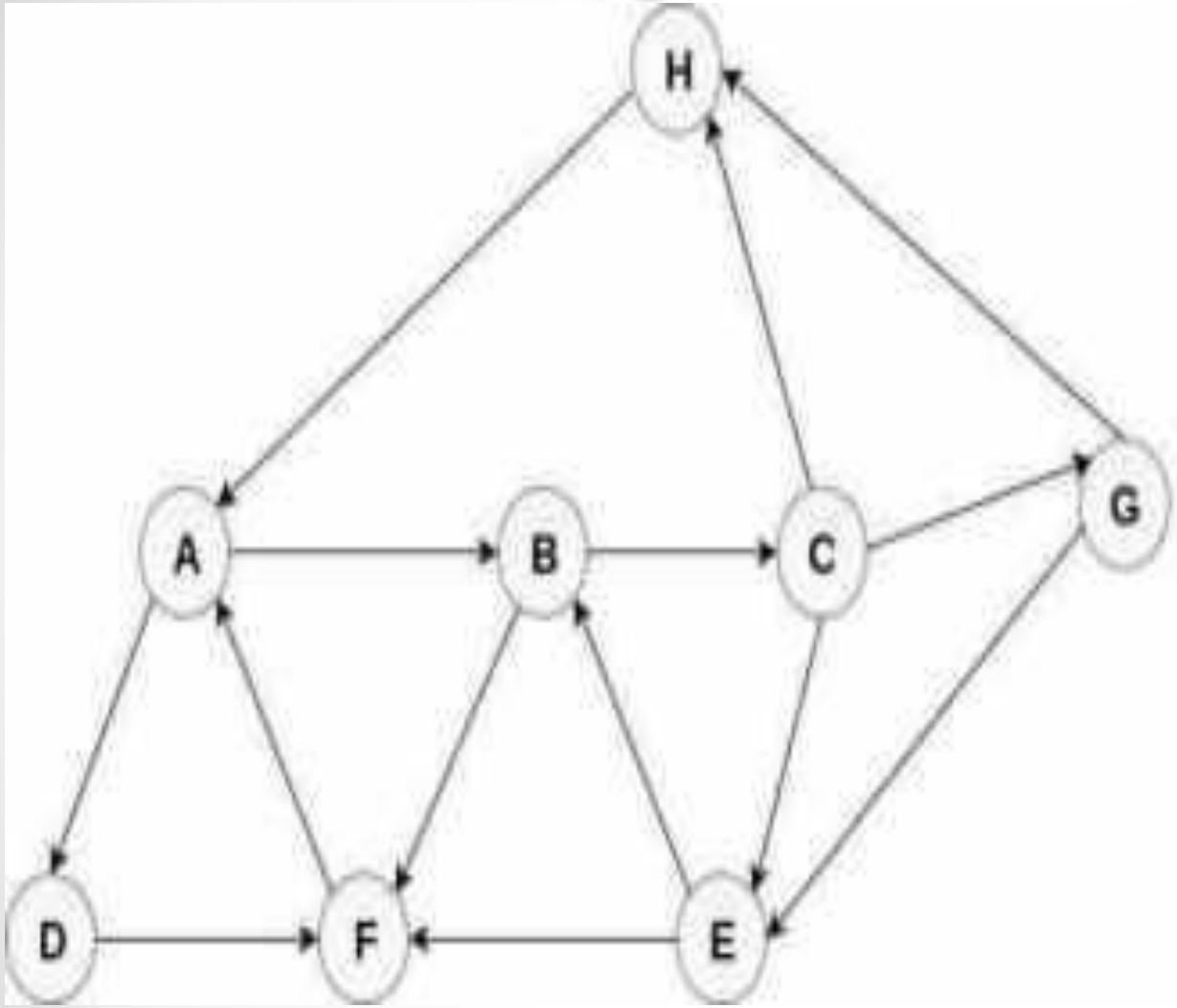
Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their
STATUS = 2 (waiting state)
[END OF LOOP]

Step 6: EXIT

DFS Algorithms Example



Adjacency Lists

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

The printing sequence of the graph will be :

H → A → D → F → B → C → G → E

Complexity of DFS Algorithms

Complexity of Depth First Search

The time complexity of the DFS algorithm is

represented in the form of O(V + E), where V is the

number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

Application of DFS Algorithms

1. For finding the path

2. To test if the graph is bipartite
3. For finding the strongly connected components of a graph

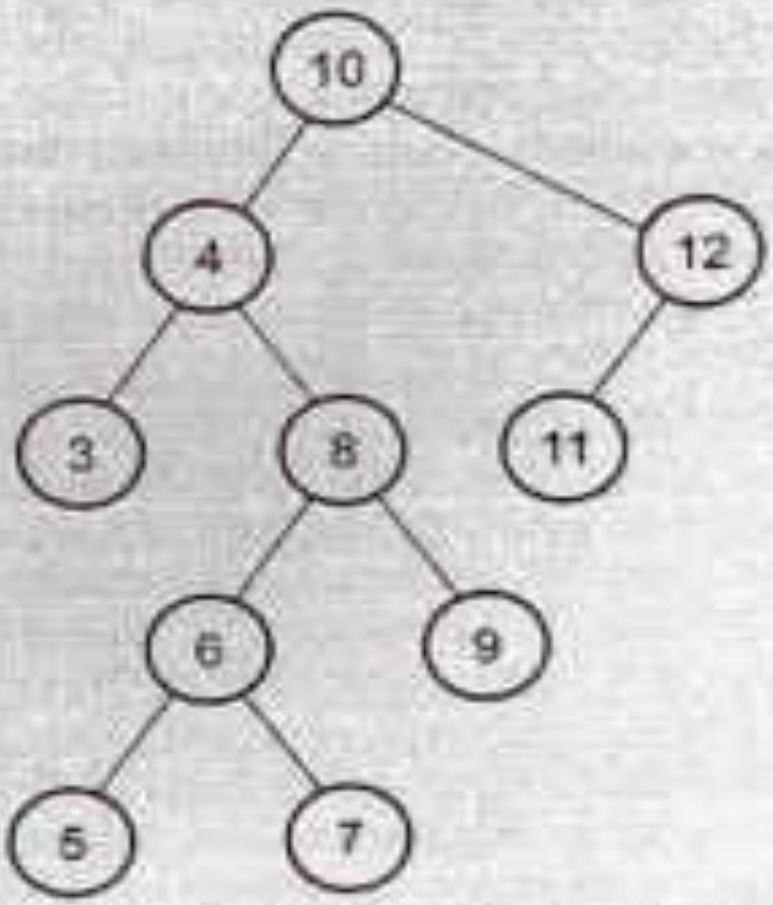
4. For detecting cycles in a graph

Application of DFS Algorithms

1. Path finding algorithms
2. Cycle detection in an undirected graph
3. Solving puzzle with only one solution, such as maze or sudoku

Example of DFS Algorithms

Example 2.7.1 *Consider following tree and obtain its DFS sequence.*



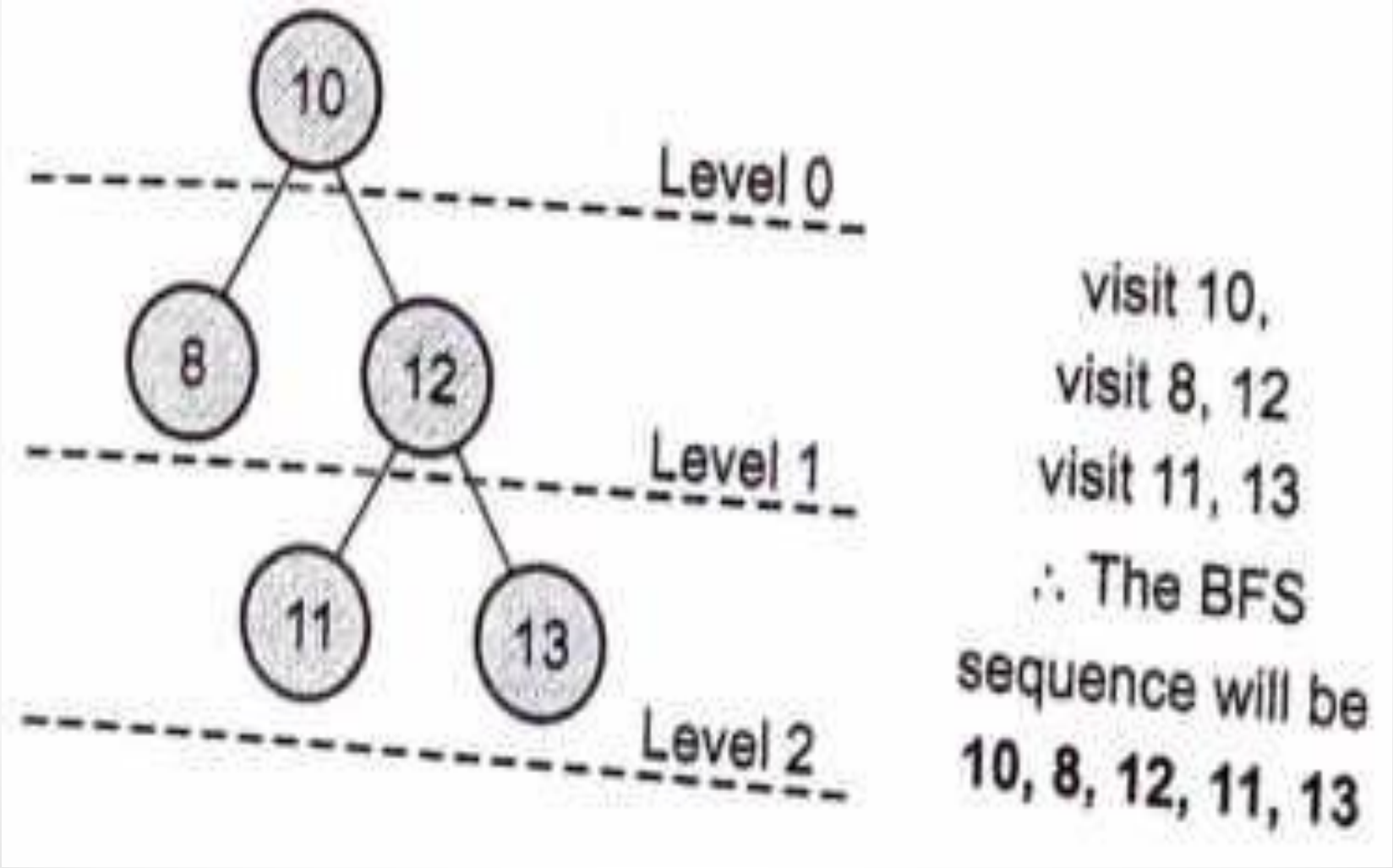
10, 4, 3, 8, 6, 5, 7, 9, 12, 11

BFS Algorithms

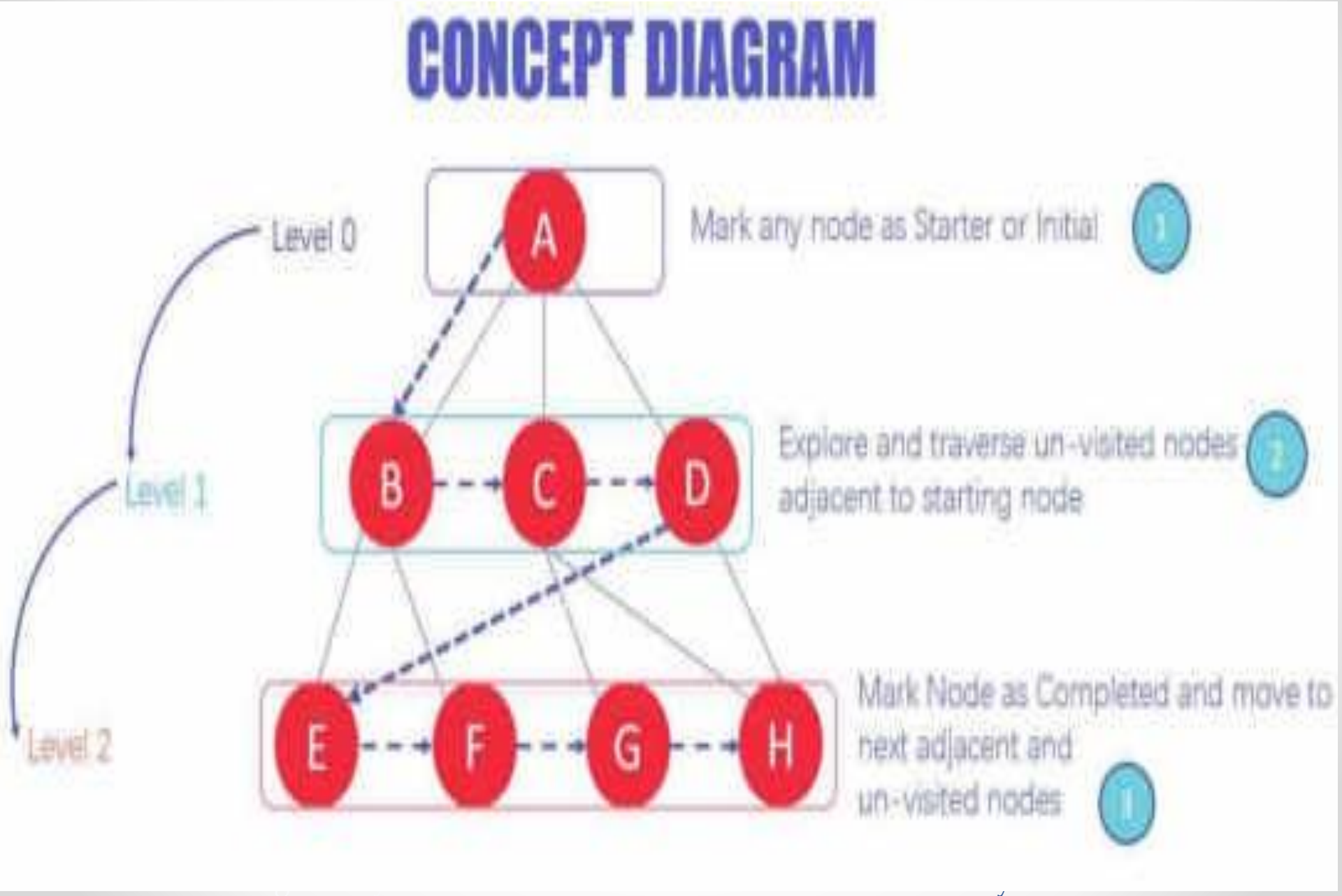
1. Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighboring nodes. Then, it selects the nearest node and explores all the unvisited nodes. While using BFS for traversal, any node in the graph can be considered as the root node.

2. BFS is the most commonly used approach. It is a recursive algorithm to search all the vertices of a tree or graph data structure. BFS puts every vertex of the graph into two categories - visited and non-visited. It selects a single node in a graph and, after that, visits all the nodes adjacent to the selected node

BFS Algorithms



BFS Algorithms



BFS Algorithms

Rule 1 − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 − If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 − Repeat Rule 1 and Rule 2 until the queue is empty.

BFS Algorithms

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

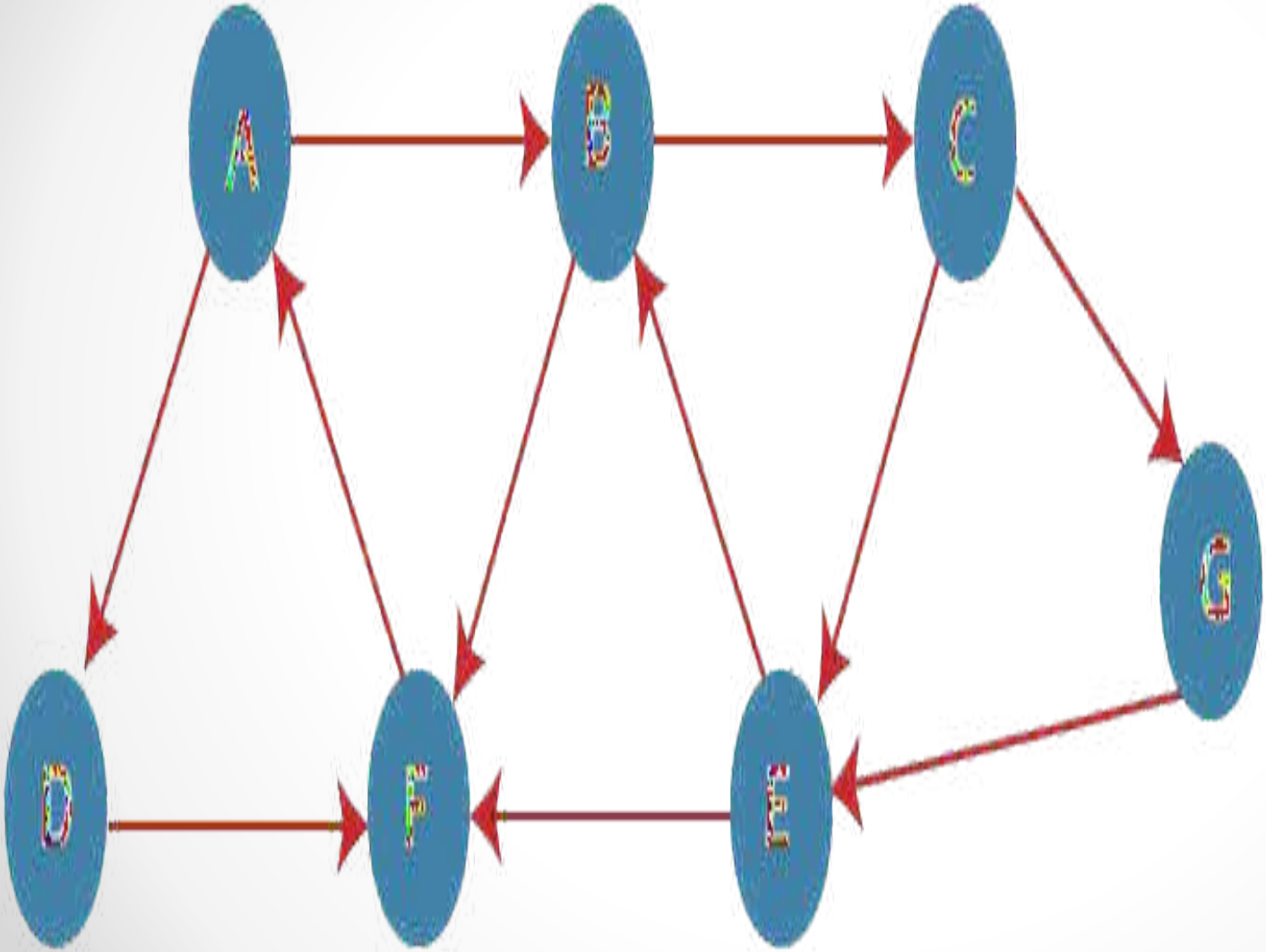
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Example of BFS Algorithms



Adjacency Lists

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

Complexity of BFS Algorithms

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the

number of nodes and E is the number of edges.

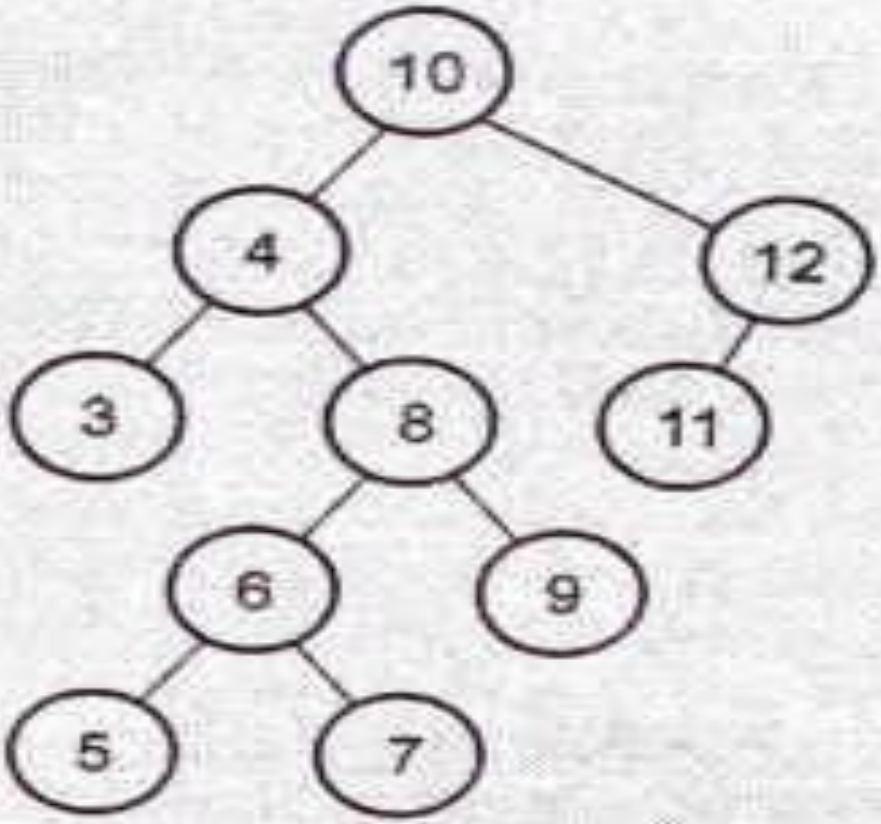
The space complexity of the algorithm is O(V).

Application of BFS Algorithms

1. BFS can be used to find the neighboring locations from a given source location, using GPS
2. In a peer-to-peer network, BFS algorithm can be used as a traversal method to find all the neighboring nodes. Most torrent clients, such as BitTorrent, uTorrent, etc. employ this process to find "seeds" and "peers" in the network.
3. BFS is used to determine the shortest path and minimum spanning tree.
4. BFS is also used in Cheney's technique to duplicate the garbage collection.
5. It can be used in ford-Fulkerson method to compute the maximum flow in a flow network.

Example of DFS Algorithms

Consider following tree and find its BFS sequence.



: 7, display it.

BFS sequence : 10, 4, 12, 3, 8, 11, 6, 9, 5, 7

BFS Vs DFS

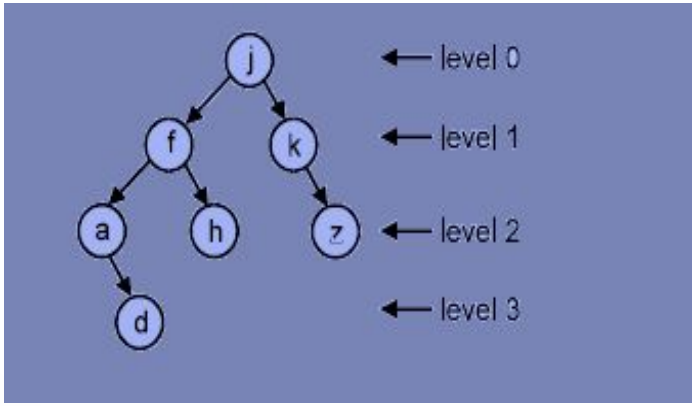
sn	BFS	DFS
1	BFS finds the shortest path to the destination.	DFS goes to the bottom of a subtree, then backtracks.
2	The full form of BFS is Breadth-First Search.	The full form of DFS is Depth First Search.
3	It uses a **queue** to keep track of the next location to visit.	It uses a **stack** to keep track of the next location to visit.
4	BFS traverses according to **tree level.**	DFS traverses according to **tree depth.**
5	It is implemented using **FIFO list**.	It is implemented using **LIFO list.**
6	It requires **more memory**	It requires **less memory**

BFS Vs DFS

sn	BFS	DFS
7	There is no need of backtracking in BFS.	There is a need of backtracking in DFS.
8	You can never be trapped into finite loops.	You can be trapped into infinite loops.
9	If you do not find any goal, you may need to expand many nodes before the solution is found.	If you do not find any goal, the leaf node backtracking may occur.

Breadth-first Traversal

- For **Depth-first** is not the only way to go through the elements of a tree
- Another way is to go through them **level-by-level**
- For example, each element exists at a certain level (or depth) in the tree



Computing Height of Binary Tree

- This operation computes the height of linked binary tree. Height of tree is maximum path length in tree
- We can get path length by traversing tree depth wise
- Let us consider that an empty tree's height is 0 and tree with only one node has height

Getting mirror or replica or Tree Interchange of Binary Tree

- This operation finds mirror of the tree that will interchange all left and right subtrees in linked binary tree

Copying Binary Tree

- TreeCopy operation will make a copy of linked binary tree
- The function should allocate necessary nodes and copy respective contents in it. as right child

Equality Test

- This operation checks whether two binary trees are equal
- Two trees are said to be equal if they have the same topology and all corresponding nodes are equal
- Same topology refers to fact that each branch in first tree corresponds to a branch in second tree in the same order and vice versa

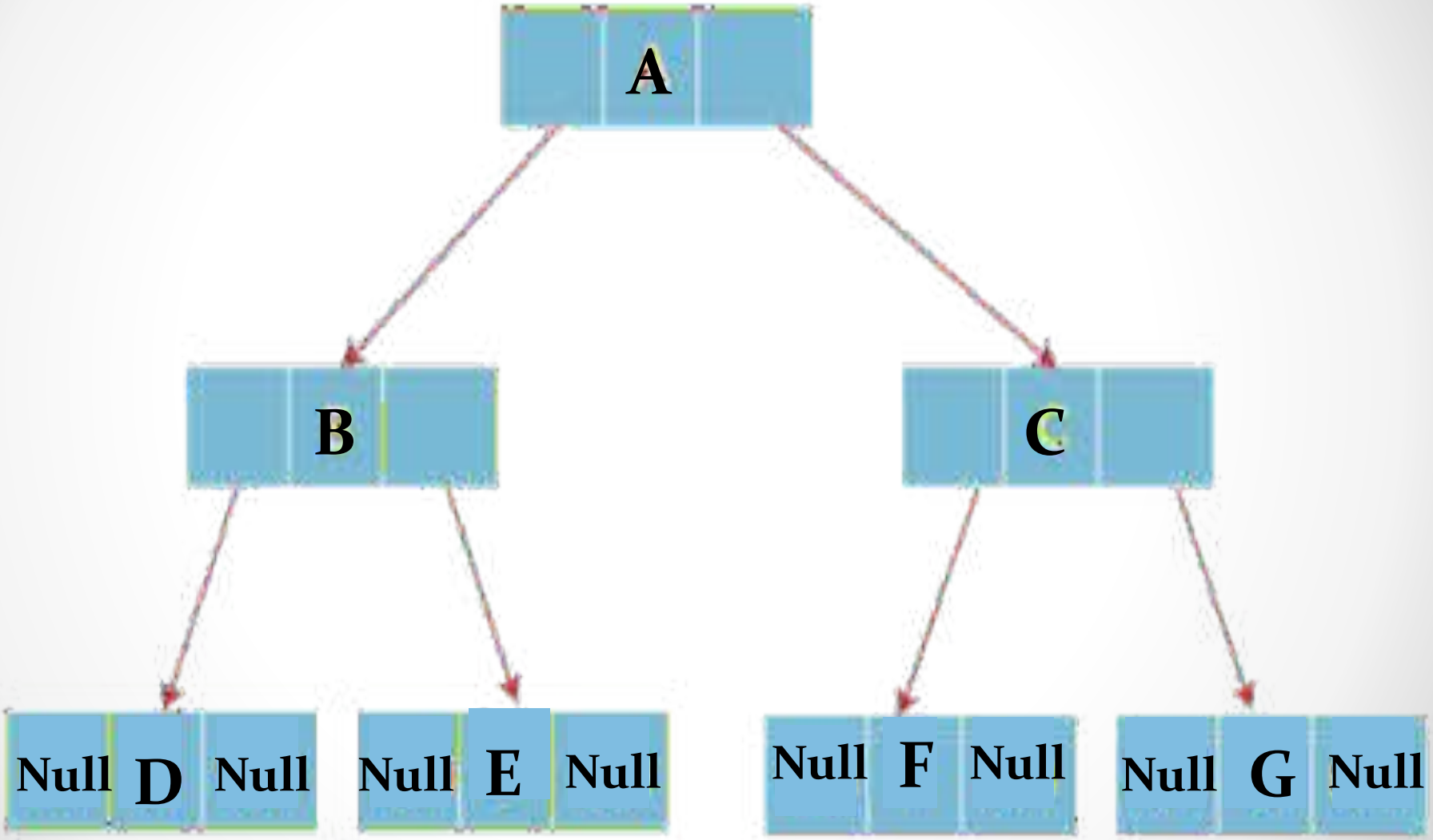
Threaded Binary Tree

1. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion.
2. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists)

Threaded Binary Tree

In the linked representation of binary trees, more than one half of the link fields contain NULL values which results in wastage of storage space. If a binary tree consists of **n** nodes then **n+1** link fields contain NULL values. So in order to effectively manage the space, a method was devised by Perlis and Thornton in which the NULL links are replaced with special links known as threads. Such binary trees with threads are known as **threaded binary trees**. Each node in a threaded binary tree either contains a link to its child node or thread to other nodes in the tree.

Threaded Binary Tree



Threaded binary tree

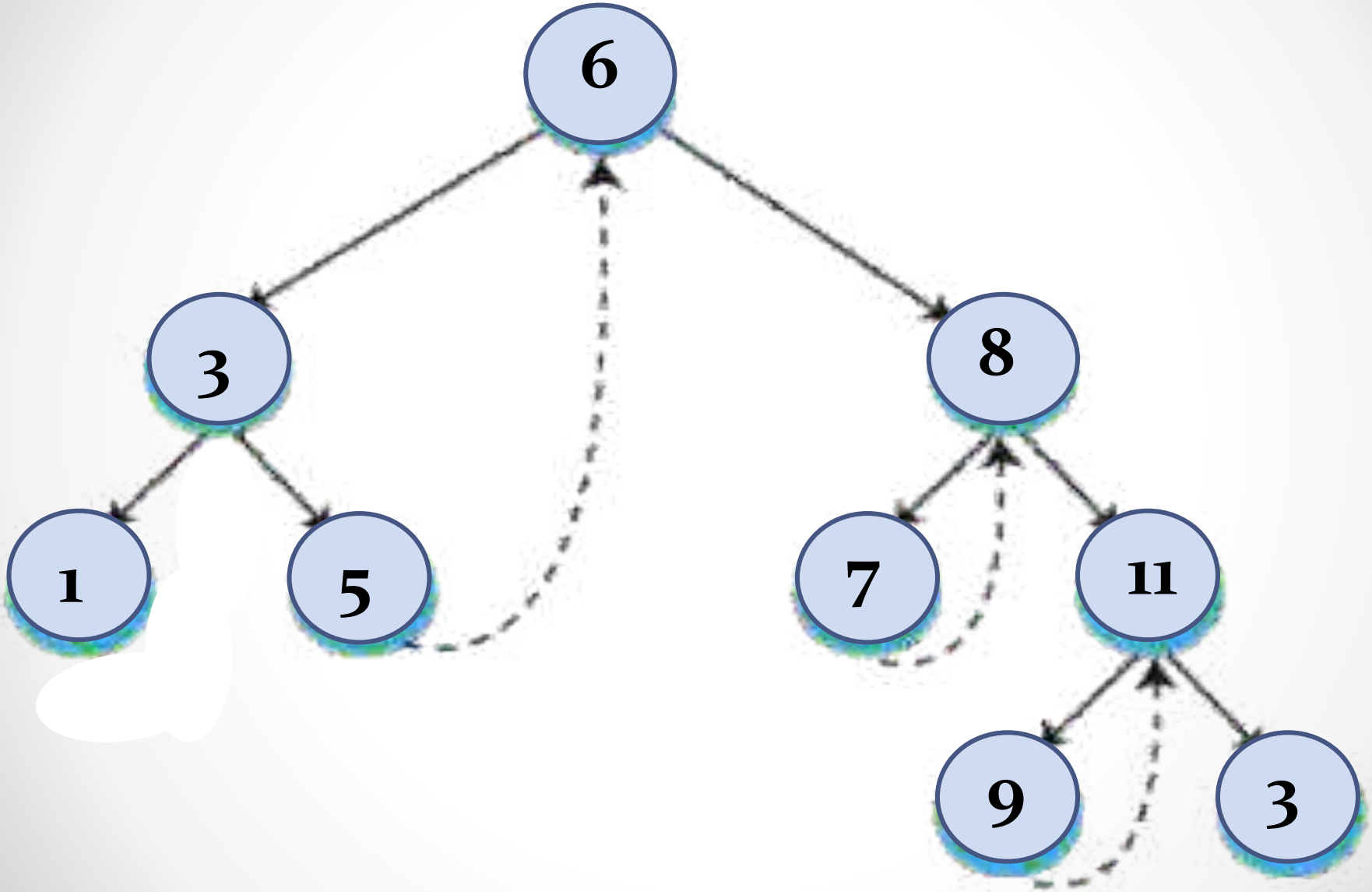
Threaded Binary Tree

There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

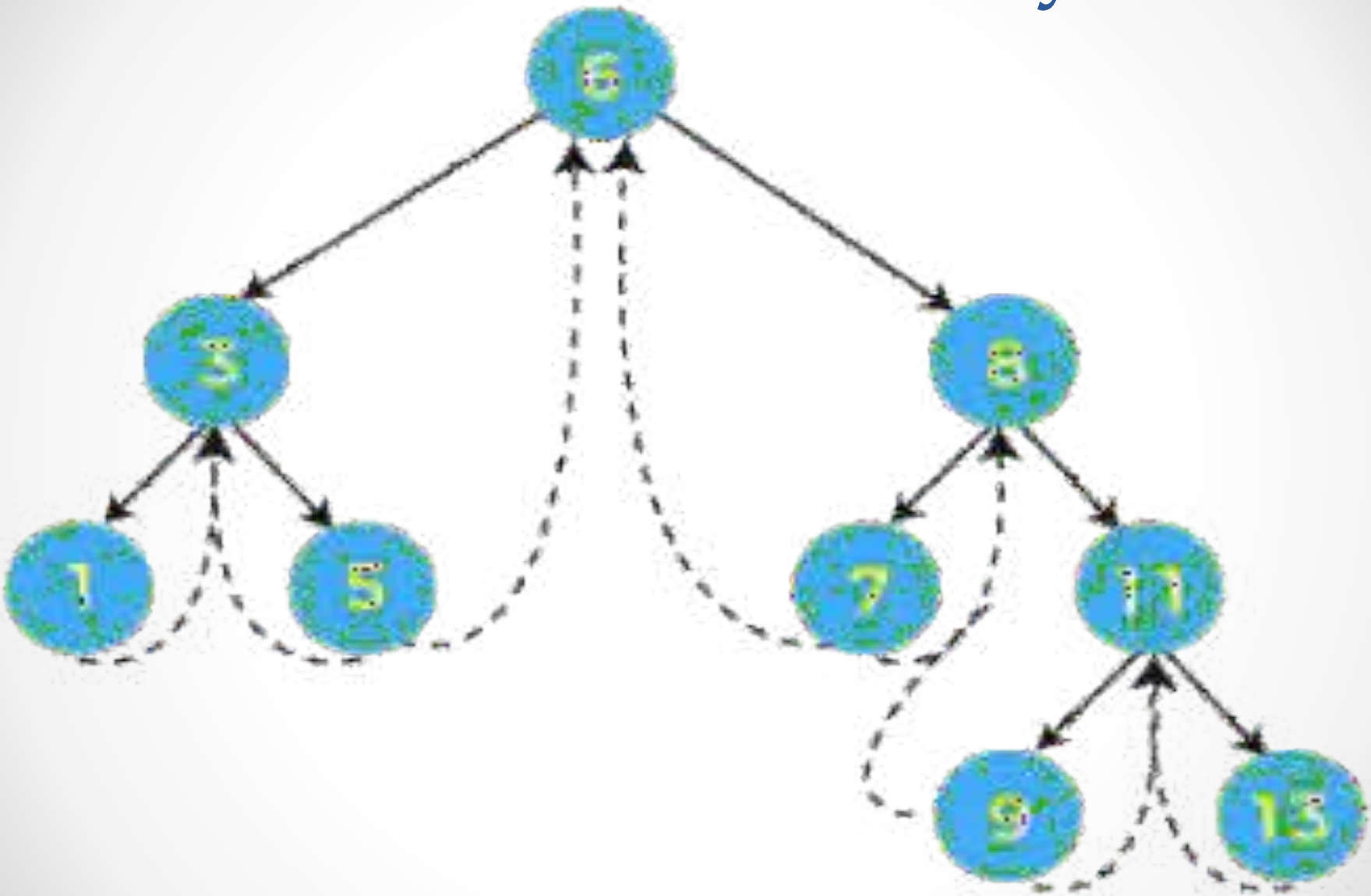
Double Threaded: Where both **left** and **right** NULL pointers are made to point to **inorder predecessor** and **inorder successor** respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

Threaded Binary Tree



Single Threaded Binary Tree

Threaded Binary Tree



Double Threaded Binary Tree

Insertion in Threaded Binary Tree

Insertion in Binary threaded tree is similar to insertion in binary tree but we will have to adjust the threads after insertion of each element.

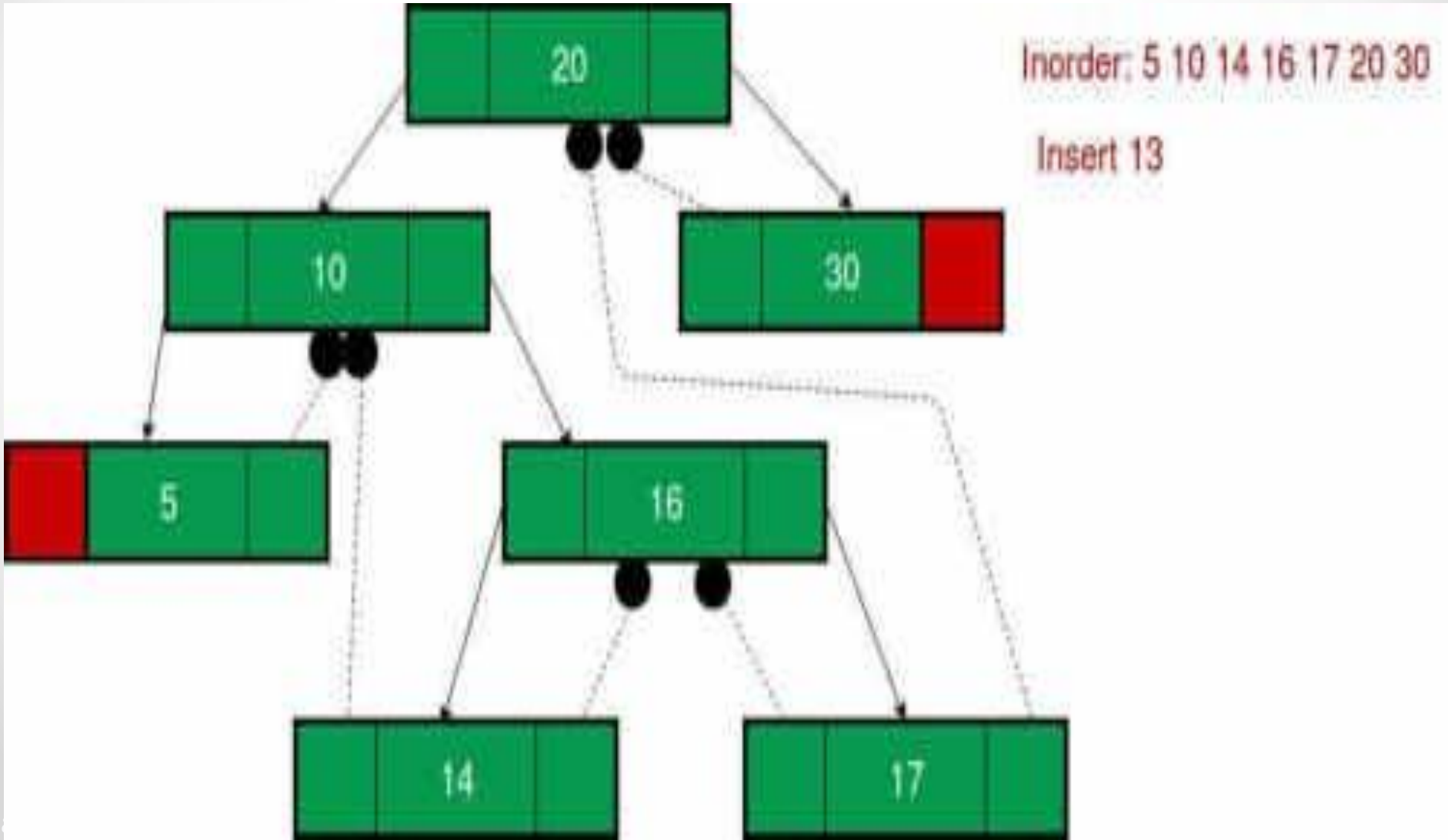
Case 1: Insertion in empty tree

Case 2: When new node inserted as the left child

Case 3: When new node is inserted as the right child

Insertion in Threaded Binary Tree

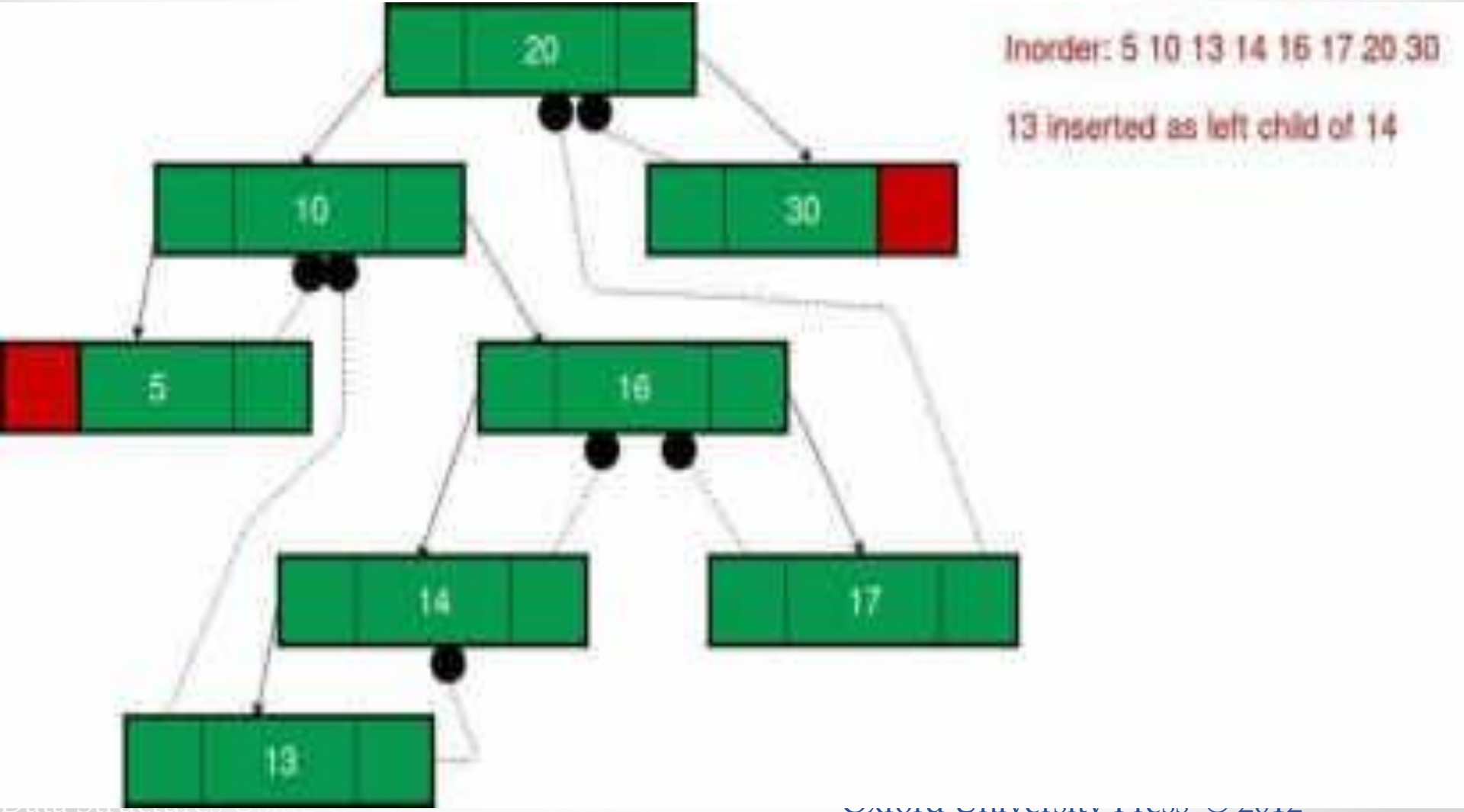
Following example show a node being inserted as left child of its parent.



Insertion in Threaded Binary Tree

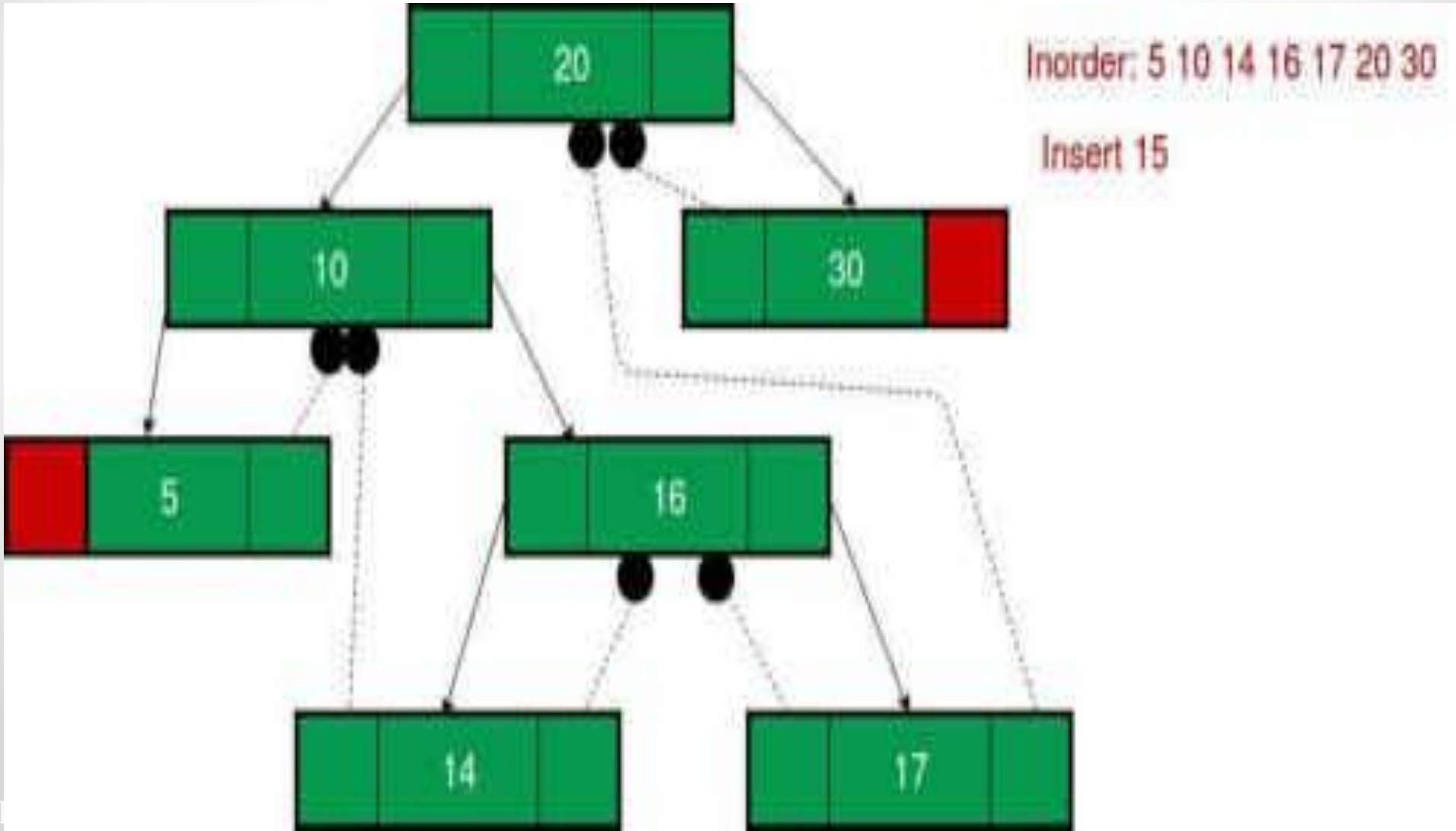
Tree

After insertion of 13 element



Insertion in Threaded Binary Tree

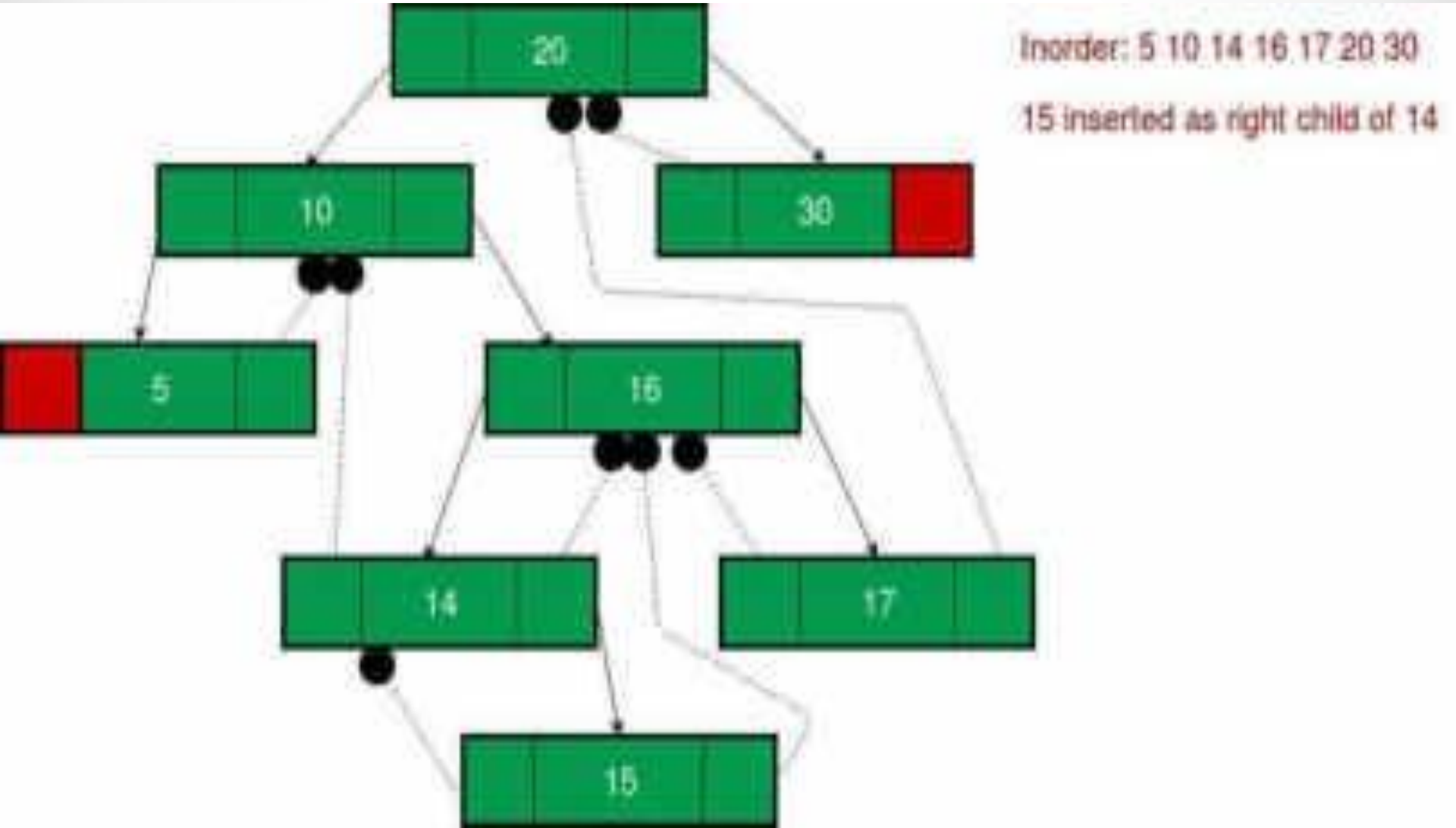
Following example shows a node being inserted as right child of its parent.



Insertion in Threaded Binary Tree

Tree

After 15 inserted



Deletion in Threaded Binary Tree

In deletion, first the key to be deleted is searched, and then there are different cases for deleting the Node in which key is found.

Case A: Leaf Node need to be deleted

Case B: Node to be deleted has only one child

Case C: Node to be deleted has two children

Deletion in Threaded Binary Tree

Case A: Leaf Node need to be deleted

In BST, for deleting a leaf Node the left or right pointer of parent was set to NULL. Here instead of setting the pointer to NULL it is made a thread.

If the leaf Node is to be deleted is left childs of its parent then after deletion, left pointer of parent should become a thread pointing to its predecessor of the parent Node after deletion.

Deletion in Threaded Binary Tree

Case B: Node to be deleted has only one child

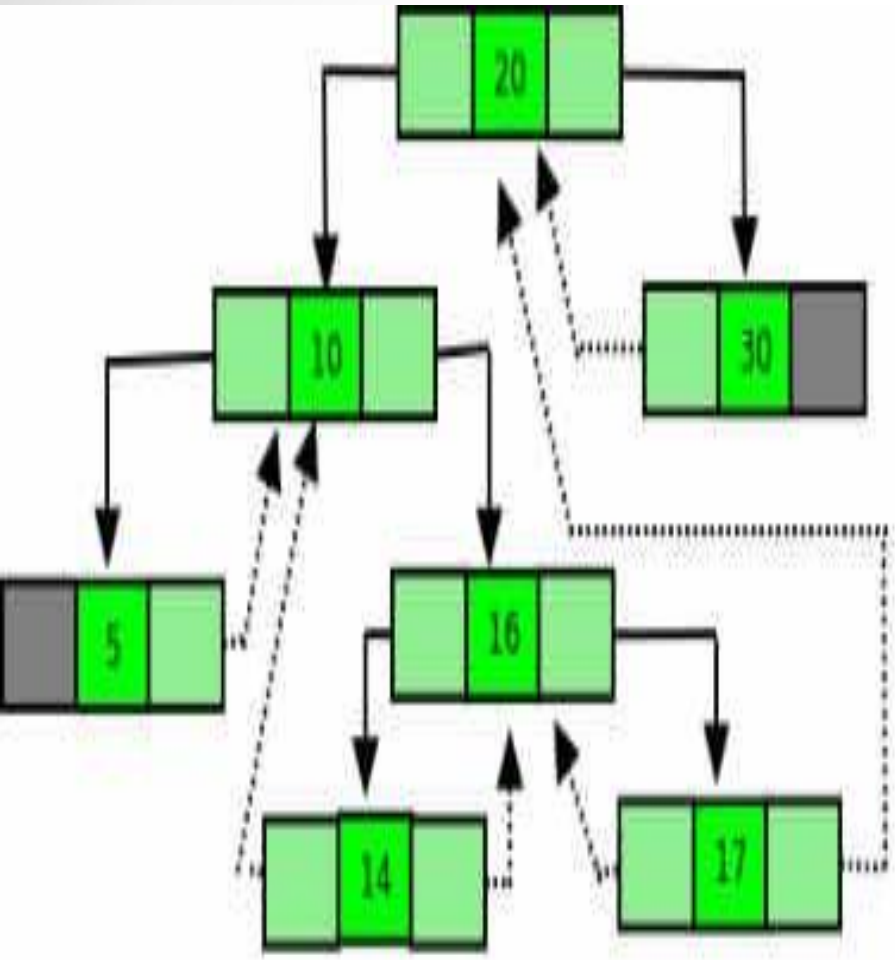
After deleting theNode as in a BST, theinorder successor and inorder predecessor of the Node are found out.

Deletion in Threaded Binary Tree

Case C: Node to be deleted has two children :

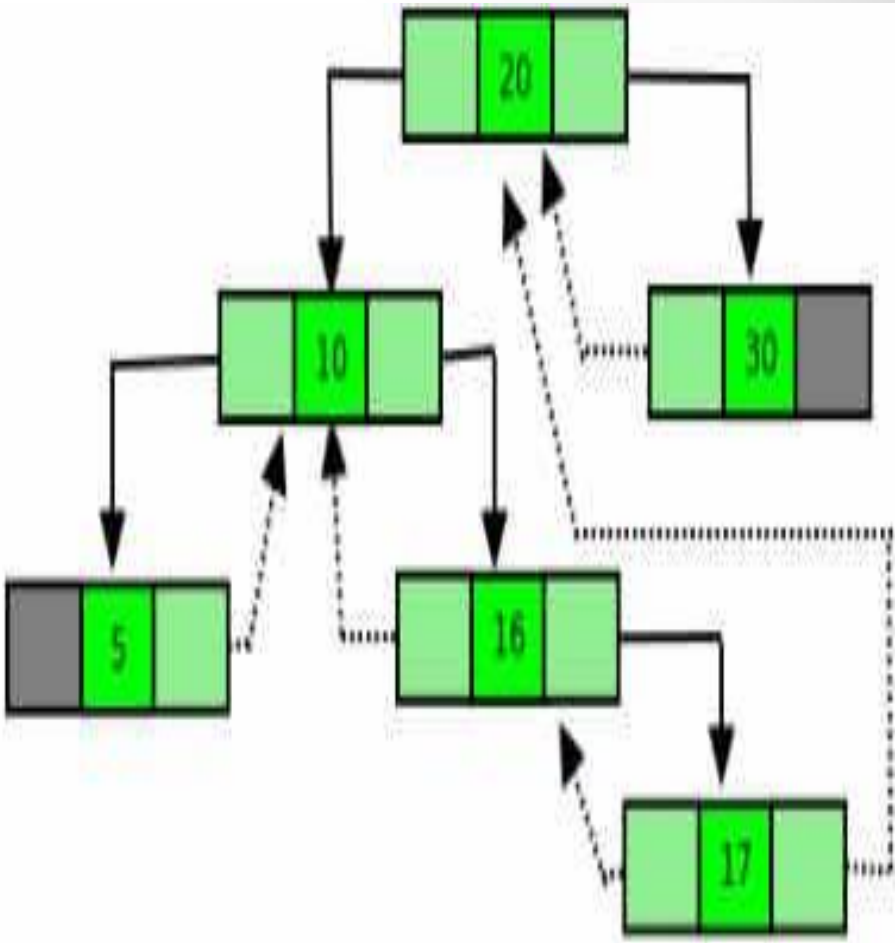
We find inorder successor of Node ptr (Node to be deleted) and then copy the information of this successor into Node ptr. After this inorder successor Node is deleted using either Case A or Case B

Deletion in Threaded Binary Tree



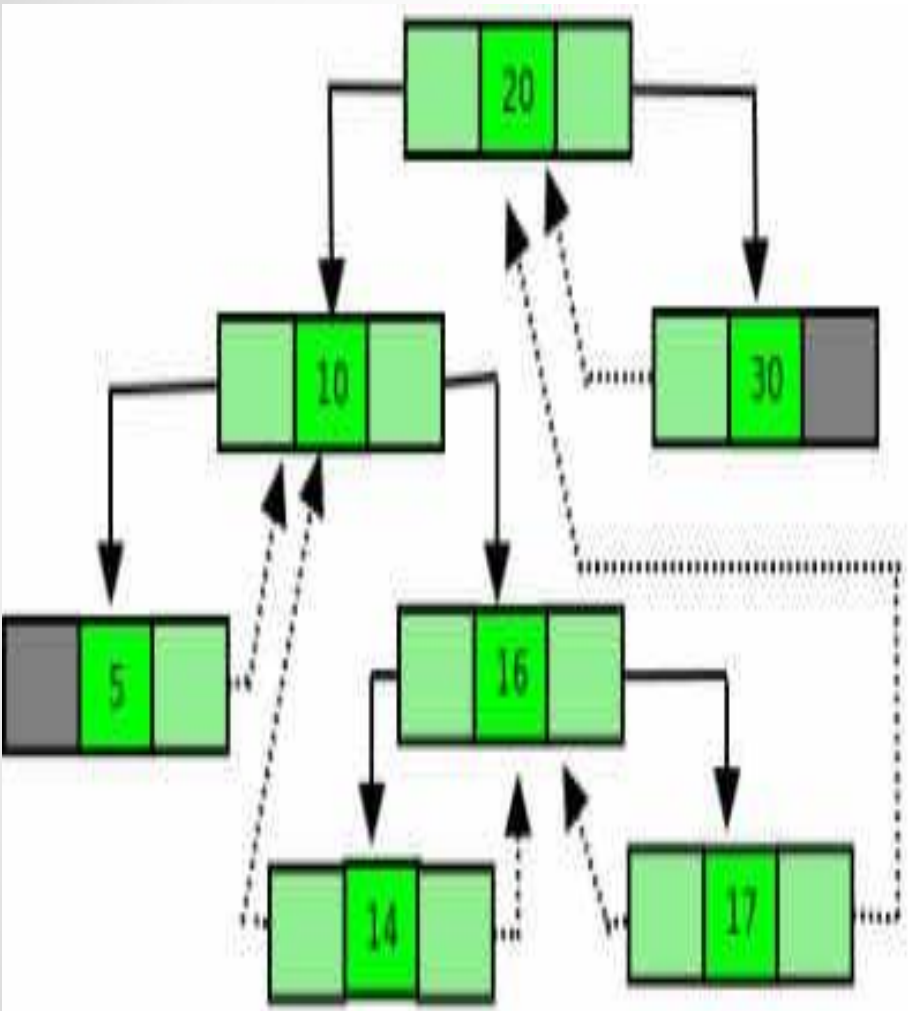
Delete 14

Inorder: 5 10 14 16 17 20 30



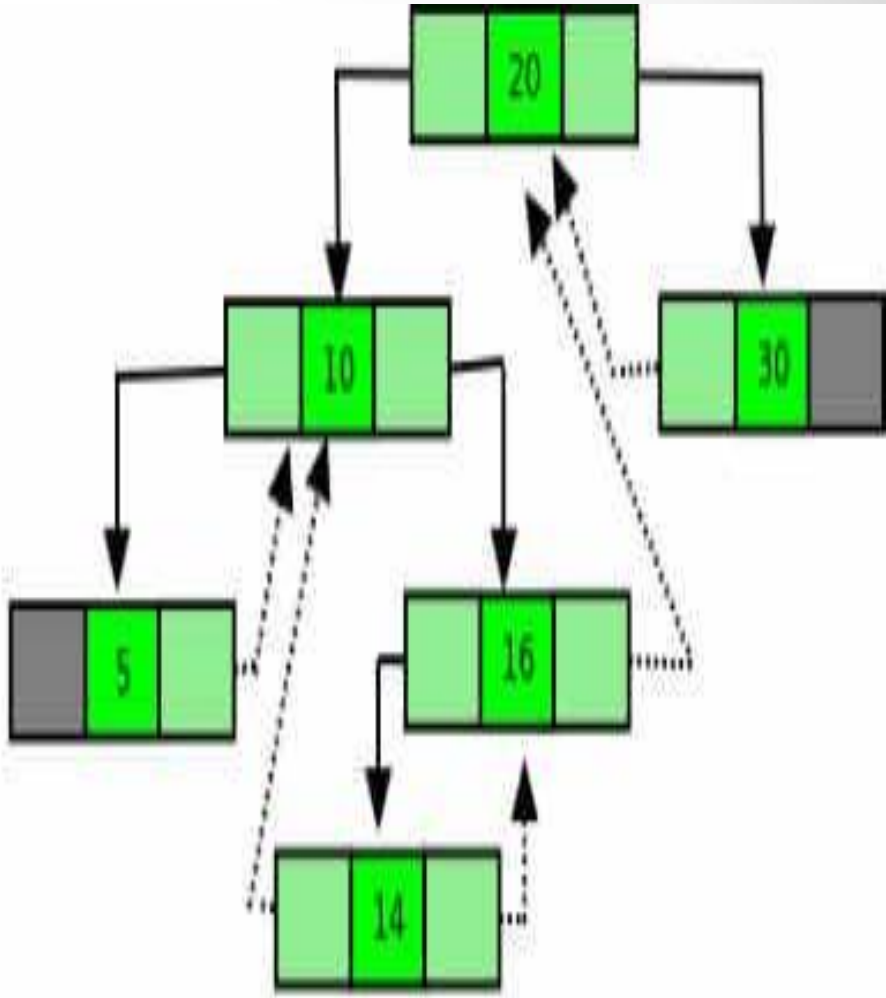
Inorder: 5 10 16 17 20 30

Deletion in Threaded Binary Tree



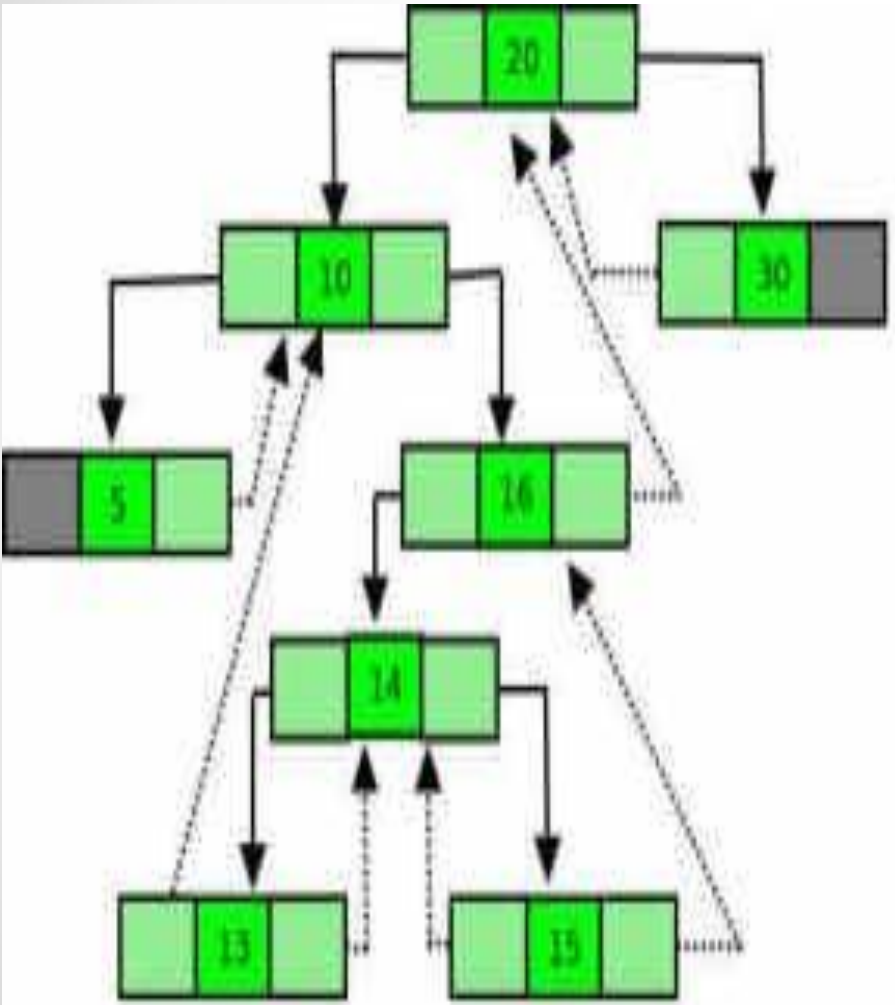
Delete 17

Inorder: 5 10 14 16 17 20 30



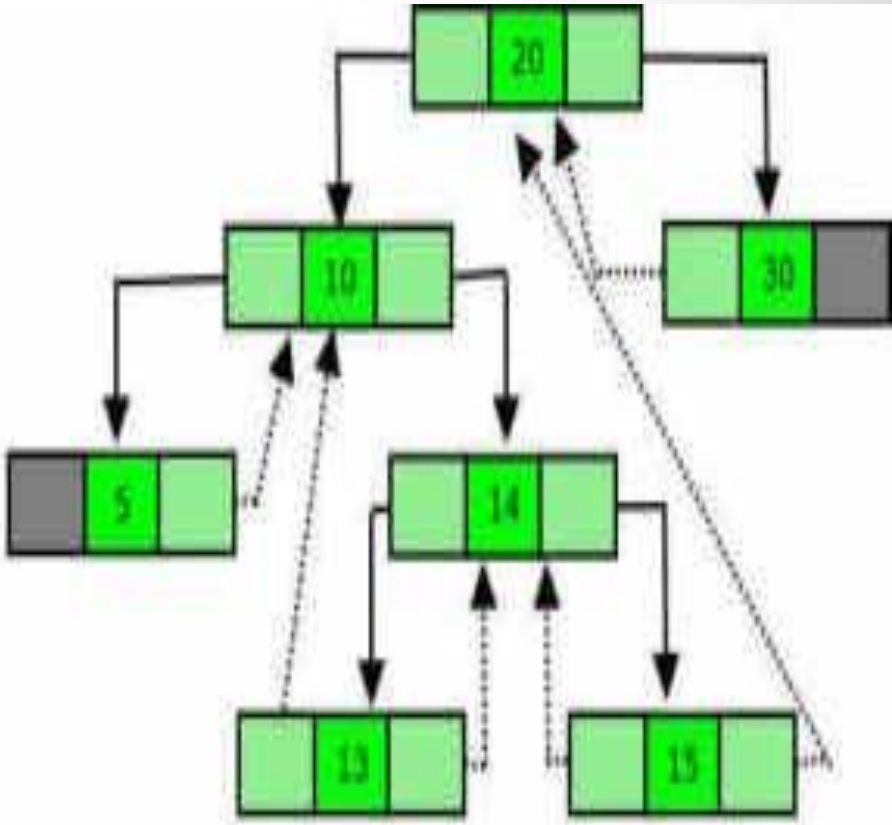
Inorder: 5 10 14 16 20 30

Deletion in Threaded Binary Tree



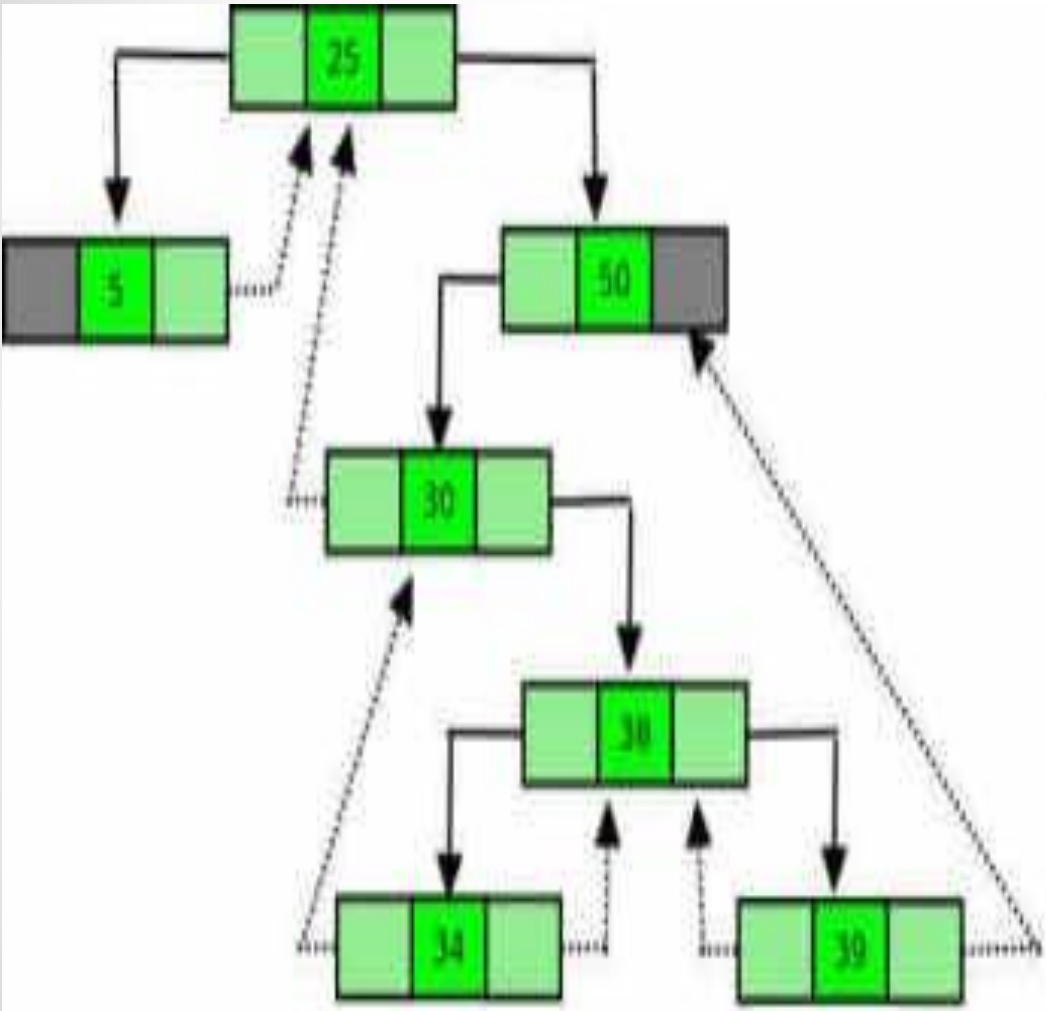
Delete 16

Inorder: 5 10 14 16 17 20 30



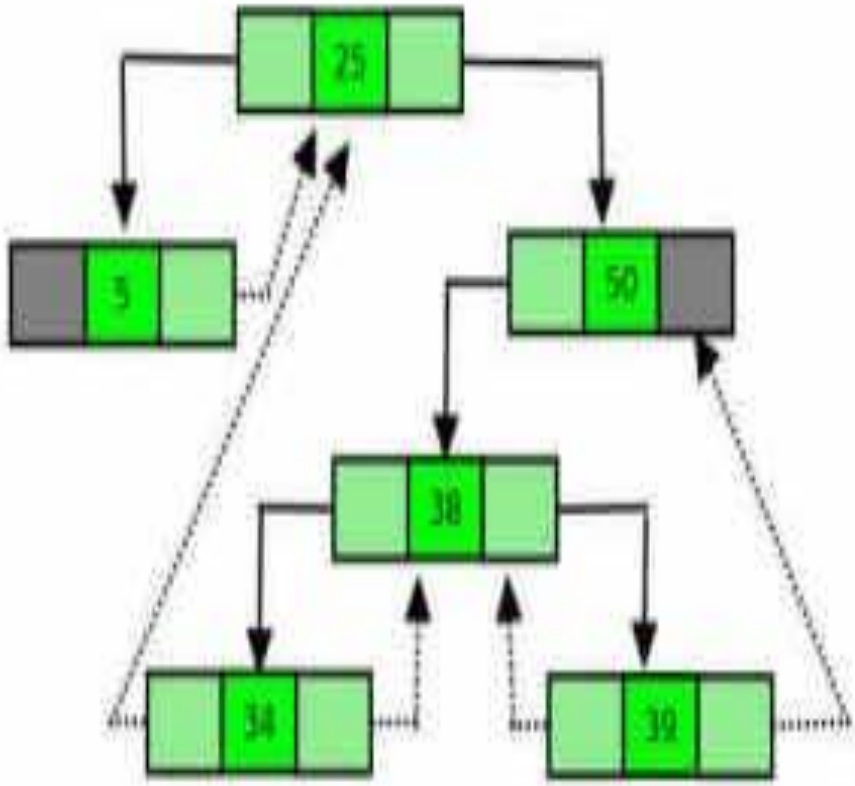
Inorder: 5 10 13 14 15 20 30

Deletion in Threaded Binary Tree



Delete 30

Inorder: 5 25 30 34 38 39 50



Inorder: 5 25 34 38 39 50

Advantages Threaded Binary Tree

1. In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
2. It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

Disadvantages Threaded Binary Tree

1. When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
2. Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

Applications Threaded Binary Tree

1. The idea of threaded binary trees is to make inorder traversal of the binary tree faster and do it without using any extra space, so sometimes in small systems where hardware is very limited we use of threaded binary tree for betterefficiency th software in a limited hardware space.

e

Threaded Binary Tree

Thornton has suggested to replace all the null links by pointers, called Threads

- A tree with thread is called as **Threaded Binary Tree (TBT)**
- Both threads and tree pointers are pointers to nodes in the tree
- **The difference is that threads are not structural pointers of the tree**
- They can be removed and the tree does not change
- **Tree poin**ters are the pointers that join and hold the tree together

Implementation of Threaded Binary Tree

```
17.   if (root==NULL)
18.     {
19.       root=temp;
20.       head=new node;
21.       head->data=999;
22.       head->right=head;
23.       head->rbit=0;
24.       head->left=root;
25.       head->lbit=0;
26.       root->left=head;
27.       root->right=head;
28.       temp->lbit=1;
29.       temp->rbit=1;
30.   }
31.   else{
32.   insert(root,temp)
33.   cout<<"Insert new Data";
34.   cin>>ch;}
35.   }while(ch=='y')
36.   }
```

Structure for TBT

Node creation

```
1.    struct node{
2.    int data;
3.    node *left;
4.    node *right;
5.    int lbit;
6.    int rbit;
7.    };
8.    create()
9.    {
10.   do{
11.     temp=new node;
12.     cin>>temp->data;
13.     temp->left=NULL;
14.     temp->right=NULL;
15.     temp->lbit=0;
16.     temp->rbit=0
```

Head & Root

Insertion in Threaded Binary Tree

```
if(temp->data > root->data)
  {
   if(root->right==NULL)
    {
      root->right=temp;
      temp->left=root;
      temp->right=root->right;
//thread to head
     root->rbit=0;
  }
 }
}
```

```
1.  insert(node *root, node *temp)
2.  {
3.    if(temp->data < root->data)
4.     {
5.       if(root->left==NULL)
6.       {
7.         root->left=temp; //root->left was
          //previously null now its points to temp that has
          //value 1
8.         temp->left= root-> left //it will go
        to the head
9.         temp->right=root //thread to root
10.        root->lbit=0;
11.          temp->lbit=1;
12.          temp->rbit=1;
13.       }
14.       else
15.       {
16.         insert(root->left, temp)
17.       }
18.  }
```

Inorder traversal in Threaded Binary Tree

```
1.  inorder(node *head)
2.    {
3.      node *t = head->left; // take a temporary pointer 't' and point it to left of head
4.     do
5.      {
6.        while(t->lbit!=1)
7.        {
8.          t=t->left;
9.        }
10.        cout<<t->data;
11.        while(t->rbit==1)
12.        {
13.          t=t->right; //go to right by thread
14.          if(t==head)
15.          return;
16.          cout<<t->data;
17.        }
18.          t=t->right;
19.       }while(t!=head)
20.    }
```

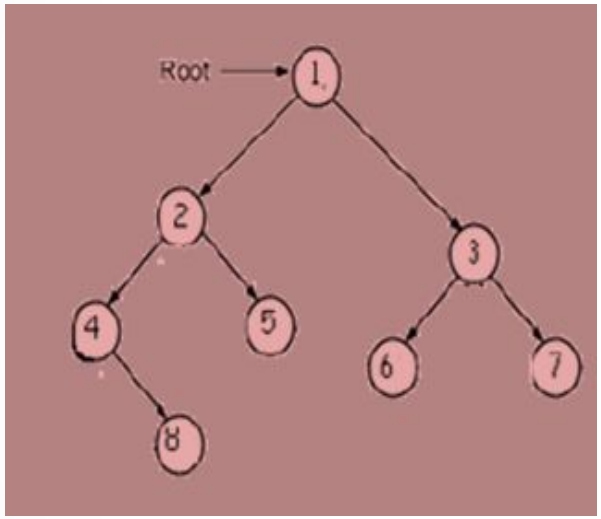
Preorder traversal in Threaded Binary Tree

```
1.  preorder(node *head)
2.    {
3.      node *t = head->left; // take a temporary pointer 't' and point it to left of head
4.     do
5.      {
6.        while(t->lbit!=1)
7.         {
8.          cout<<t->data;
9.          t=t->left;
10.        }
11.        cout<<t->data;
12.        while(t->rbit==1)
13.        {
14.          t=t->right; //go to right by thread
15.          if(t==head)
16.          return;
17.        }
18.          t=t->right;
19.        }while(t!=head);
20.  }
```

Threaded Binary Tree

- Threads utilize the NULL pointers waste space to improve processing efficiency
- One such application is to use these NULL pointers to make traversals faster
- In a left NULL pointer, we store a pointer to the node's inorder successor
- This allows us to traverse the tree both left to right and right to left without recursion

Threaded Binary Tree



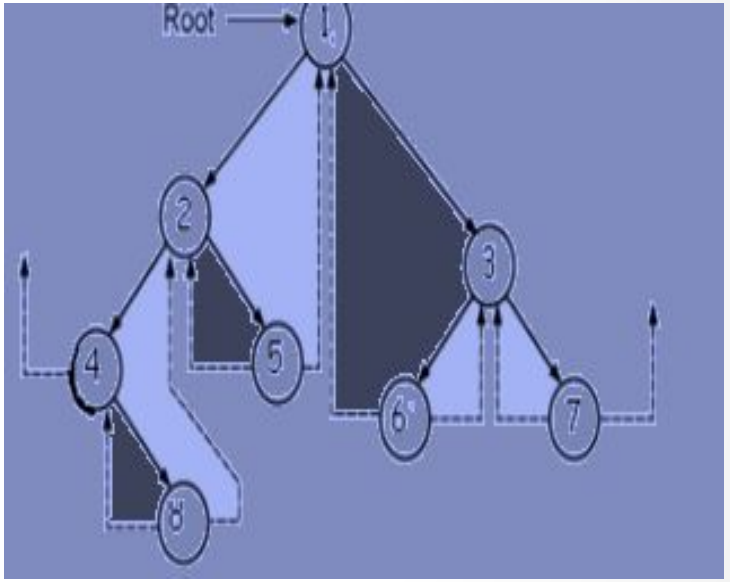


Figure 23: Threaded binary Tree

Pros and Cons of Threaded Binary Tree

- Threaded binary tree has some advantages over non-threaded binary tree.
- The traversal for threaded binary tree is straight forward. No recursion or stack is needed
- Once we locate the leftmost node, we loop, following the thread to next node
- When we find null thread, the traversal is complete

Pros and Cons of Threaded Binary Tree

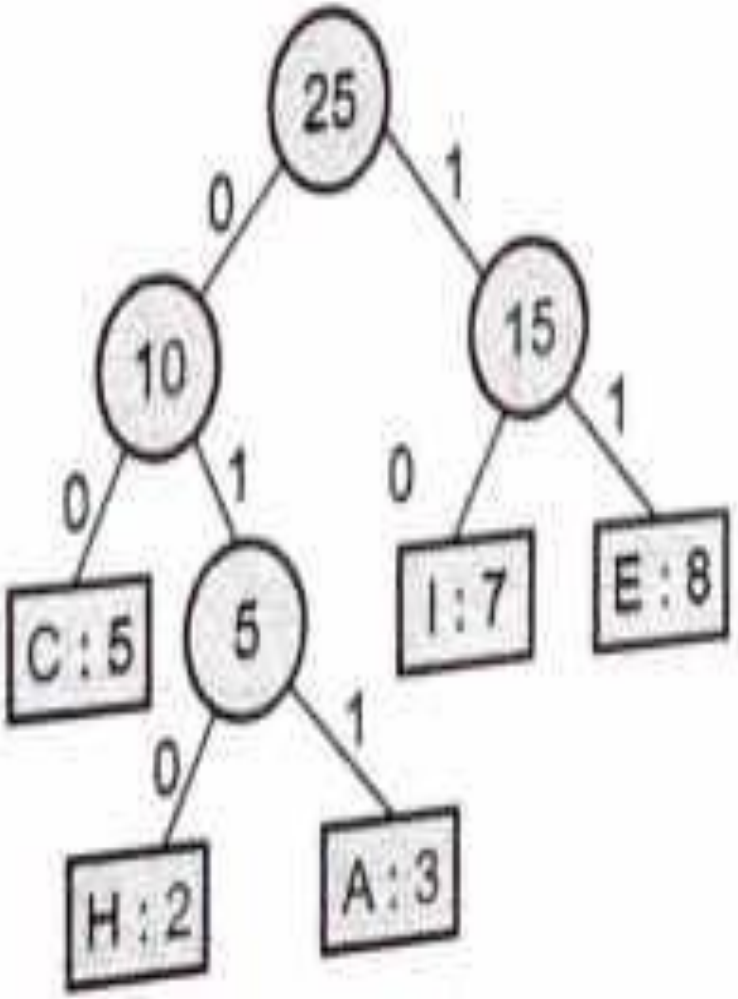
- In case of non threaded binary tree, this task is time consuming and difficult
- Also stack is needed for the same Threaded binary tree has some advantages over non-threaded binary tree.
- Threads are usually more to upward whereas links are downward
- Thus in a threaded tree we can traverse in either direction and nodes are infact circularity linked
- Hence, any node can be reached from any other node
- Insertions into and deletion from a threaded tree are although time consuming as the link and thread are to be man

Applications Of Binary Trees

- **Gaming**
- **Expression**
- **Huffman Code And**
- **Decision Trees**

Huffman's Tree Example

For encoding, the left branch is encoded as 0 and right branch as 1.



Huffman's Coding Complexity

1. The time complexity for encoding each unique character

   based on its frequency is O(nlog n).

2. Extracting minimum frequency from the priority queue takes place 2*(n-1) times and its complexity is O(log n).

   Thus the overall complexity is O(nlog n).

Applications of Huffman's Coding

1. Huffman coding is used in conventional compression formats like GZIP, BZIP2, PKZIP, etc.

2. For text and fax transmissions.

Game Trees

One of the most important applications of binary tree is in Communication

- One of the most important applications of binary tree
- Another Interesting application of trees is in the playing of games such as **tic-tac-toe, chess, nim, checkers, go, etc**
- **As an example let us consider the game of tic-tac-toe**

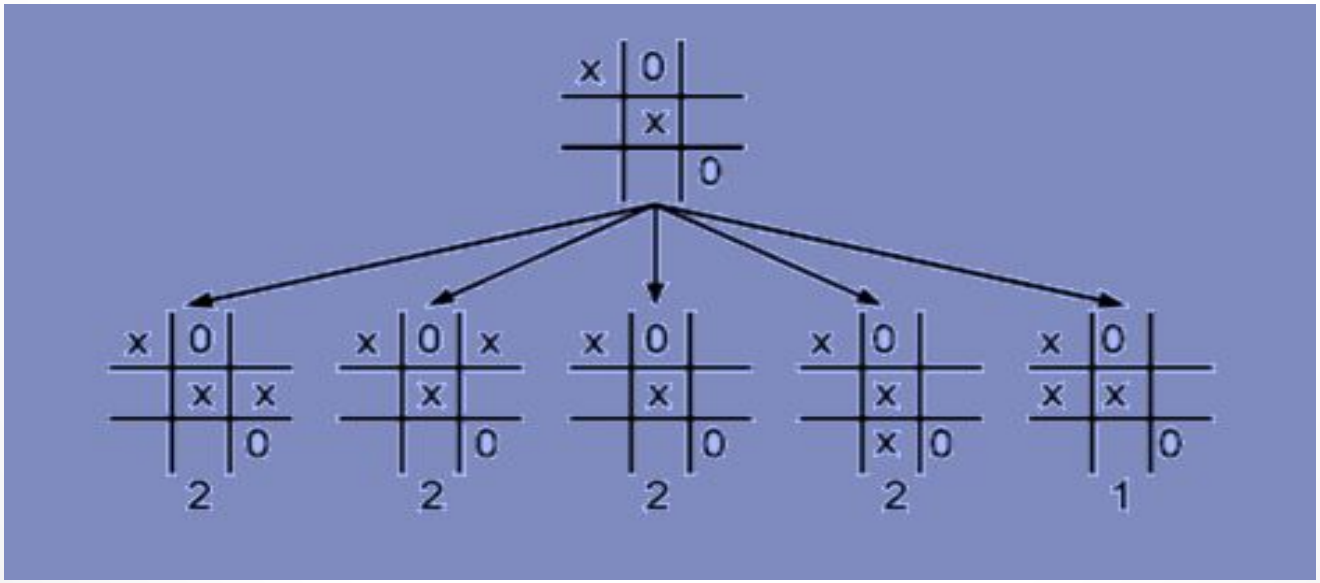


Figure 27 : An example game tree

Summary

- A binary tree is a special form of a tree. Binary tree is important and frequently used in various application of computer science. A binary tree has degree two, each node has at most two children. This makes the implementation of tree easier. Implementation of binary tree should represent hierarchical relationship between parent node and its left and right children.

- Tree, a non-linear data structure, is a mean to maintain and manipulate data in many applications as listed above. Where ever the hierarchical relationship among data is to be preserved, tree is used.

- A binary tree is a special form of a tree. Binary tree is important and frequently used in various application of computer science. A binary tree has degree two, each node has at most two children. This makes the implementation of tree easier. Implementation of binary tree should represent hierarchical relationship between parent node and its left and right children.

Summary

- Binary tree has natural implementation in linked storage. In linked organization we wish that all nodes should be allocated dynamically. Hence we need each node with data and link fields. Each node of a binary tree has both a left and a right subtree. Each node will have three fields Lchild, Data and Rchild.

- The operations on binary tree include- insert node, delete node and traverse tree. Traversal is one of the key operations. Traversal means visiting every node of a binary tree. There are various traversal methods. For a systematic traversal, it is better to visit each node (starting from root) and its both subtrees in same way.

Summary

- Let L represent left subtree, R represent right subtree and D be node data, three traversals are fundamental: LDR, LRD and DLR. These are called as inorder, postorder and preorder traversals because there is a natural correspondence between these traversals and producing the infix, postfix and preorder forms of an arithmetic expressions respectively. Also a traversal in which the node is visited before its children are visited is called a breadth first traversal; a walk where the children are visited prior to the parent is called a depth first traversal.
- The binary search tree is a binary tree with the property that the value in a node is greater than any value in a node's left subtree and less than any value in the node's right subtree. This property guarantees fast search time provided the tree is relatively balanced.·
- The key applications of tree include: expression tree, gaming, Huffman coding and decision tree.

End
of
UNIT 2….!