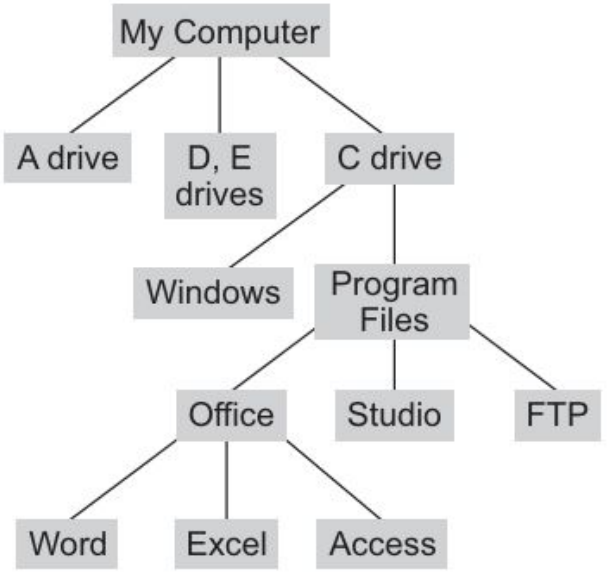
UNIT II

TREE

INTRODUCTION

- Non-linear data structures are used to represent the data containing hierarchical or network relationship between the elements.
- 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.
- Non-linear data structures are capable of expressing more complex relationships than linear data structures.
- In general, wherever the hierarchical relationship among data is to be preserved, the tree is used.

- Tree, a non-linear data structure, is a means to maintain and manipulate data in many applications.
- Consider the following example: The operating system of a computer system organizes files into directories and sub directories.
- Directories are also referred to as folders.
- The operating system organizes folders and files using a tree structure as in Fig.
- A folder contains other folders (subfolders) and files. This can be viewed as the tree drawn in Fig.
- The common uses of trees include the following:

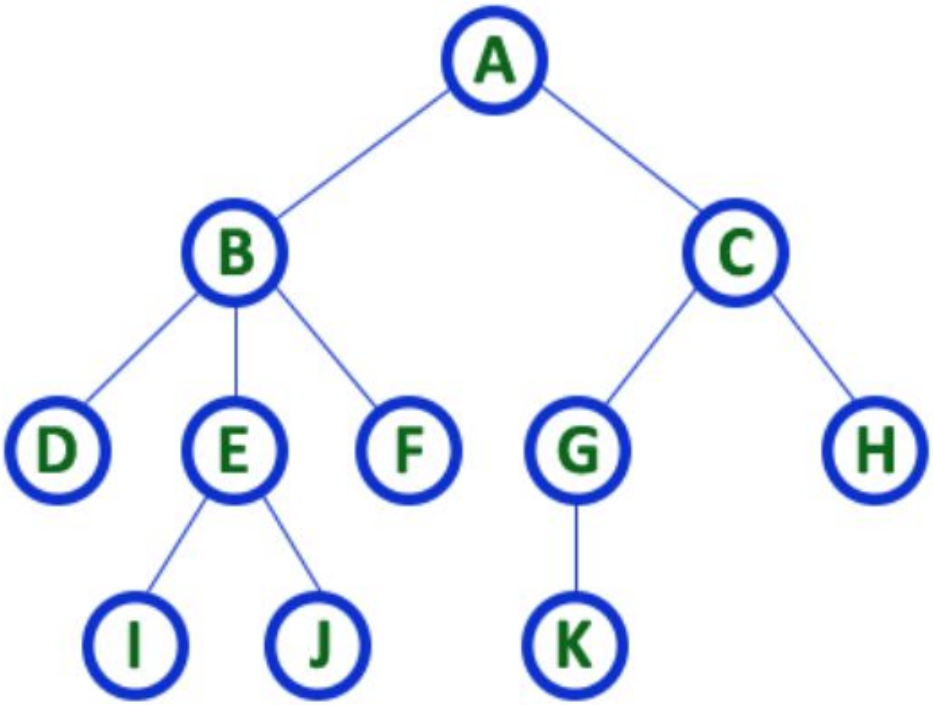


1. Manipulating hierarchical data
2. Making information easily searchable
3. Manipulating sorted lists of data

Basic Terminology

- **Tree**: A tree is a non-linear data structure that stores the information naturally in the form of a hierarchy style. It is a collection of nodes connected by directed or undirected edges.
- A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent a hierarchy.
- It's a non linear data structure as it does not store data in a sequential manner, but stores in a hierarchical fashion.
- In the Tree data structure, the first node is known as a root node i.e. from which the tree originates. Each node contains some data and also contains references to child nodes. A root node can never have a parent node.

Example

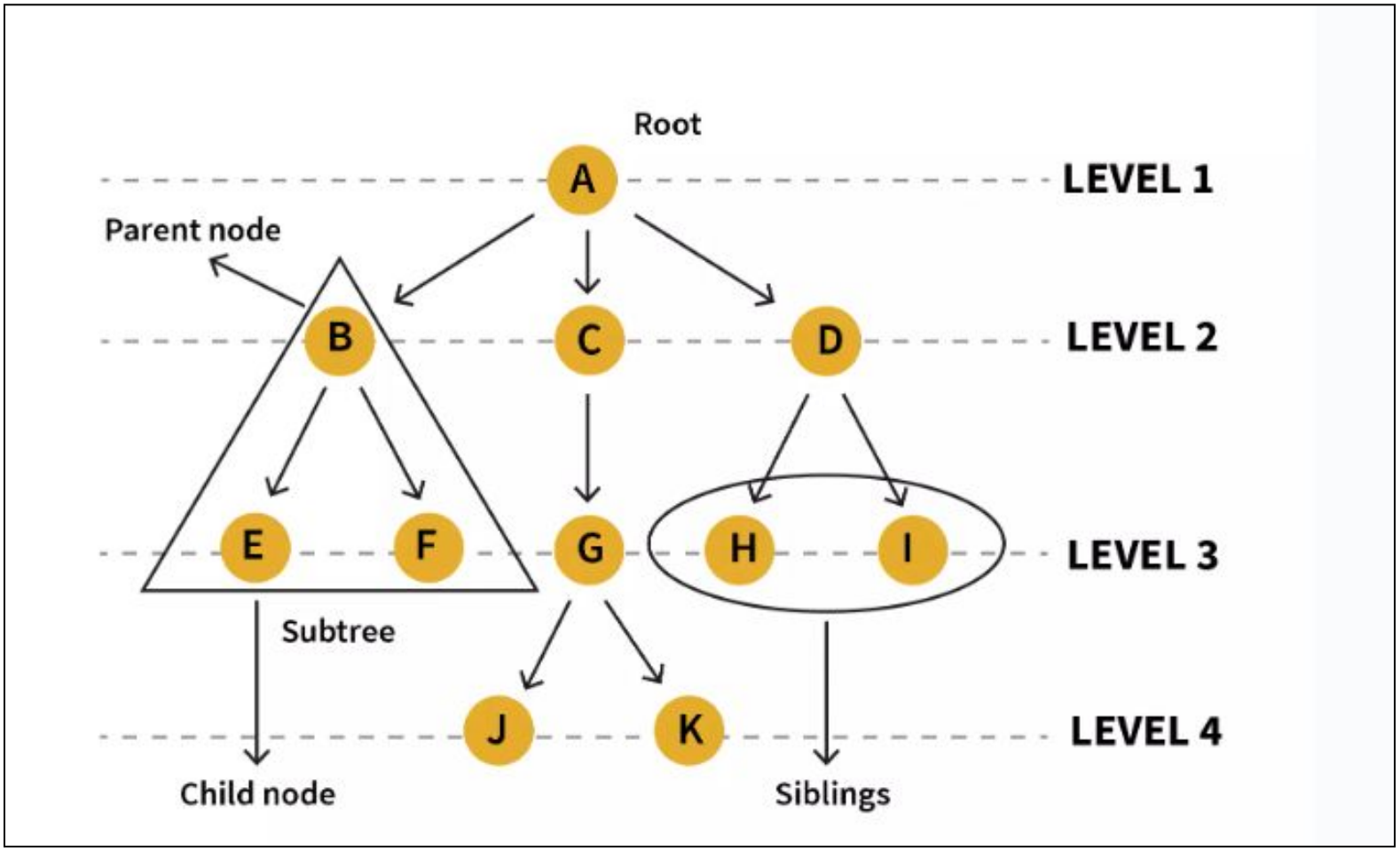


TREE with 11 nodes and 10 edges

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

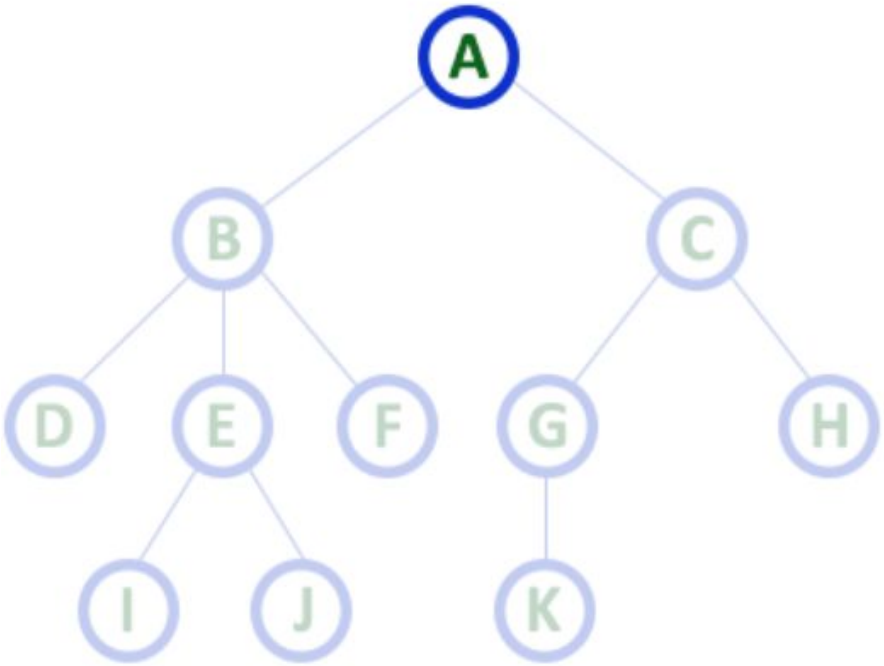
Tree Data Structure

Terminologies



1. Root

In a tree data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.

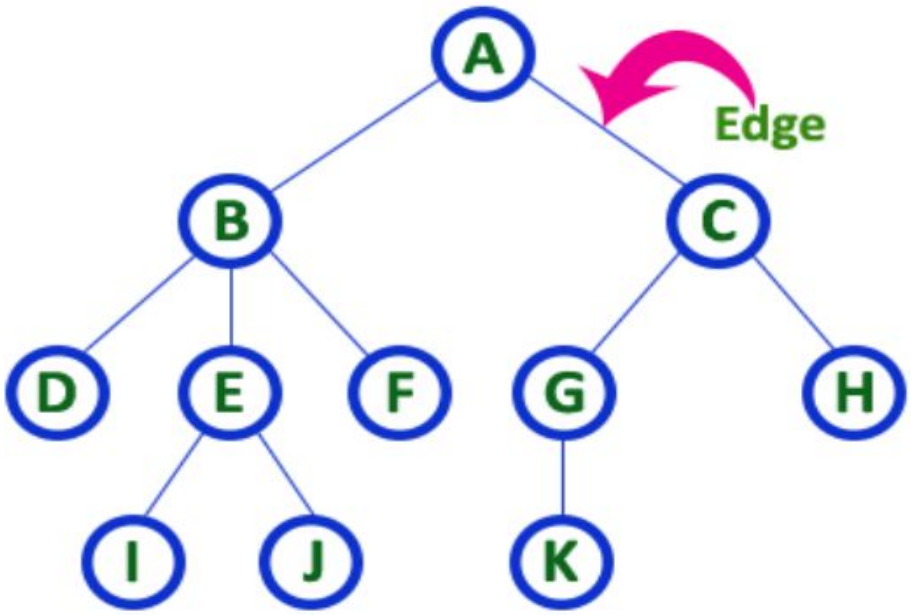


Here 'A' is the 'root' node

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

2. Edge

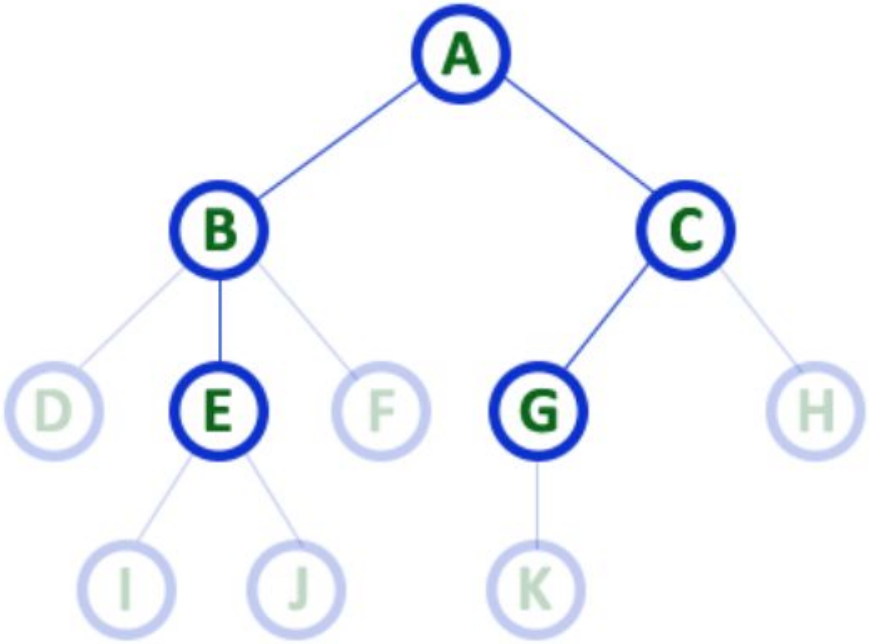
In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

3. Parent

In a tree data structure, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**".

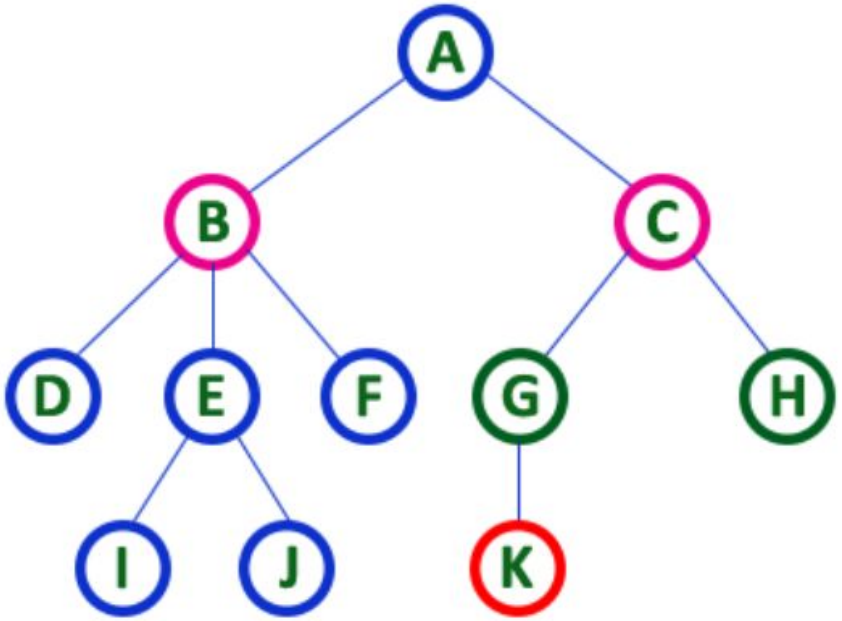


Here A, B, C, E & G are Parent nodes

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

4. Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



Here B & C are Children of A

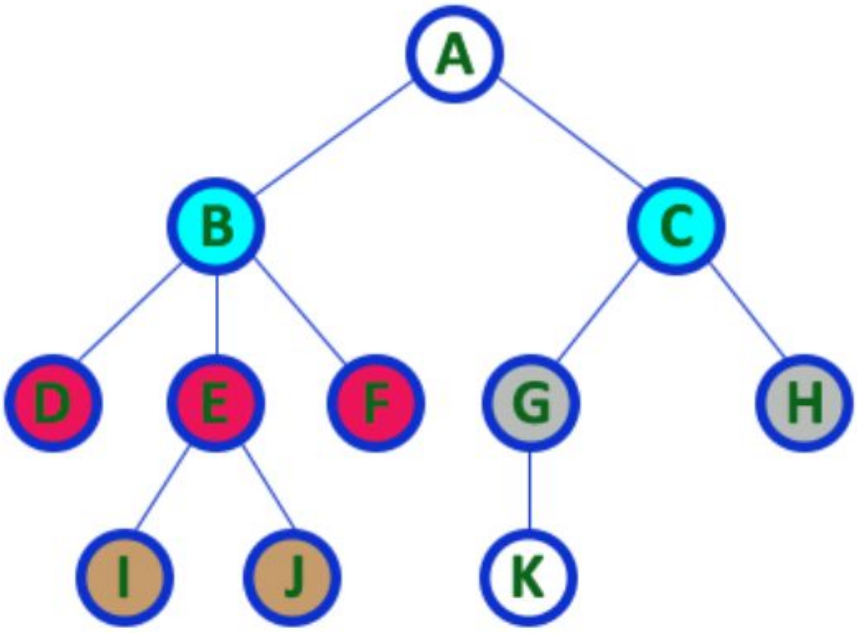
Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.

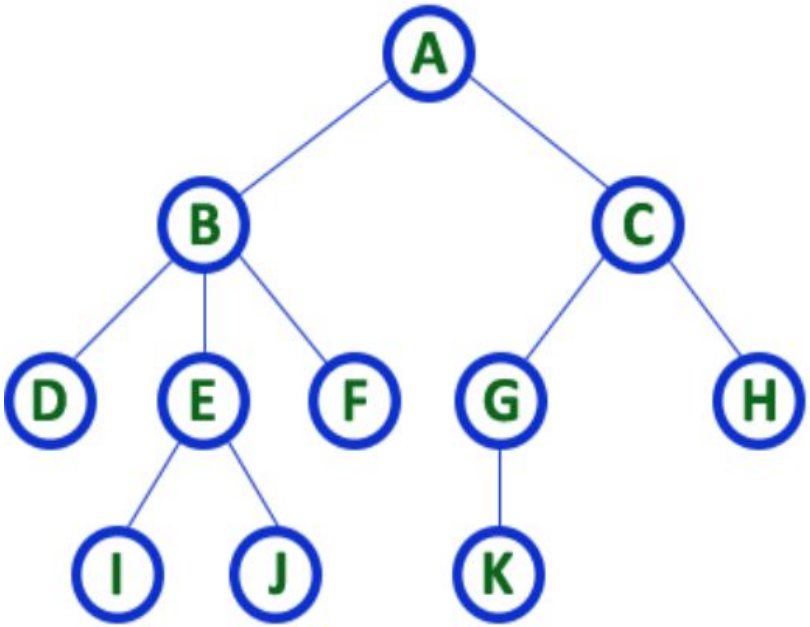


Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

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

6. Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.

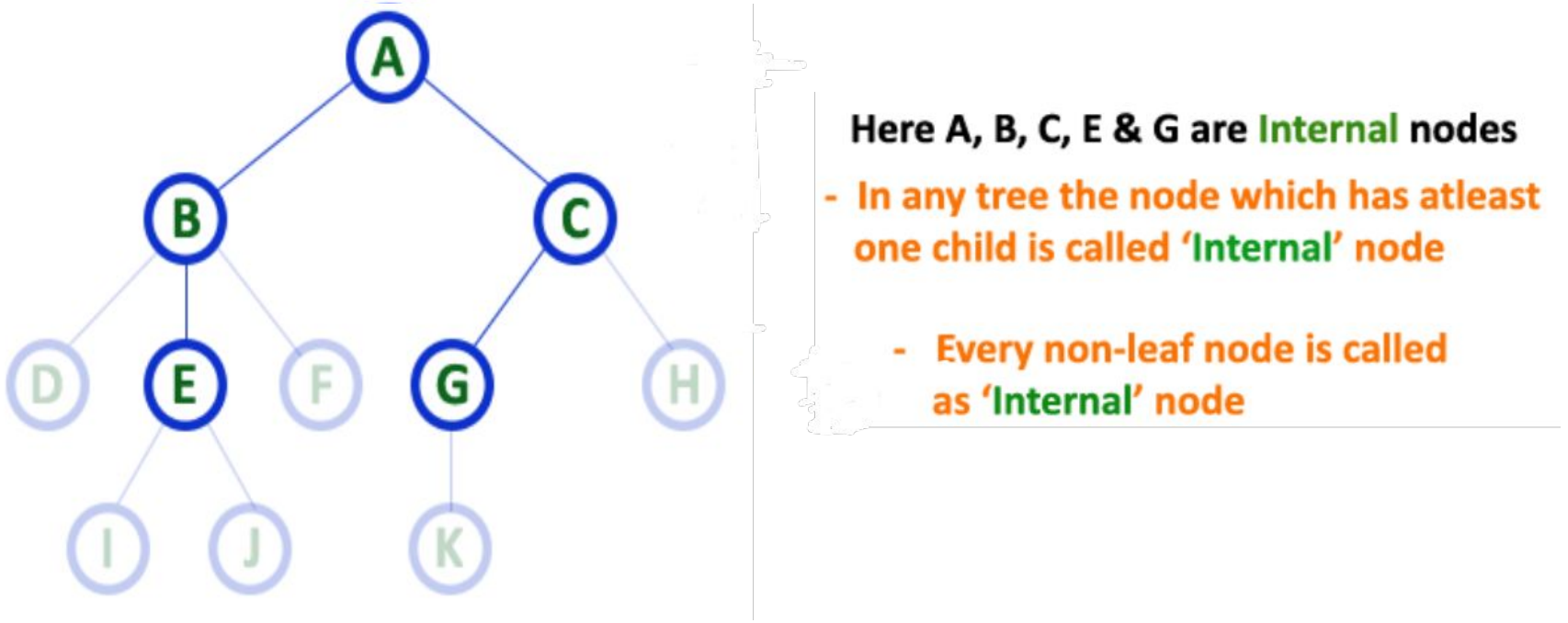


Here D, I, J, F, K & H are Leaf nodes

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

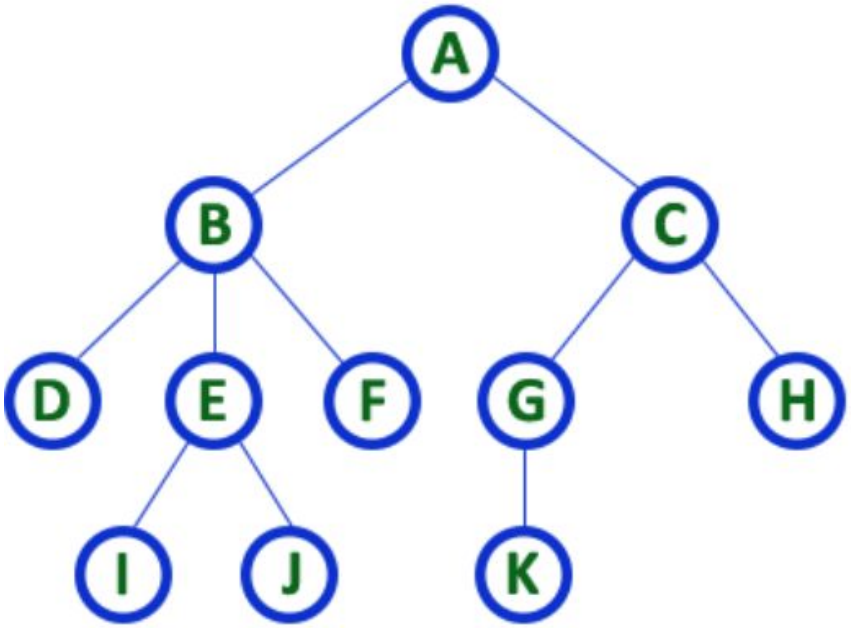
7. Internal Nodes

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



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

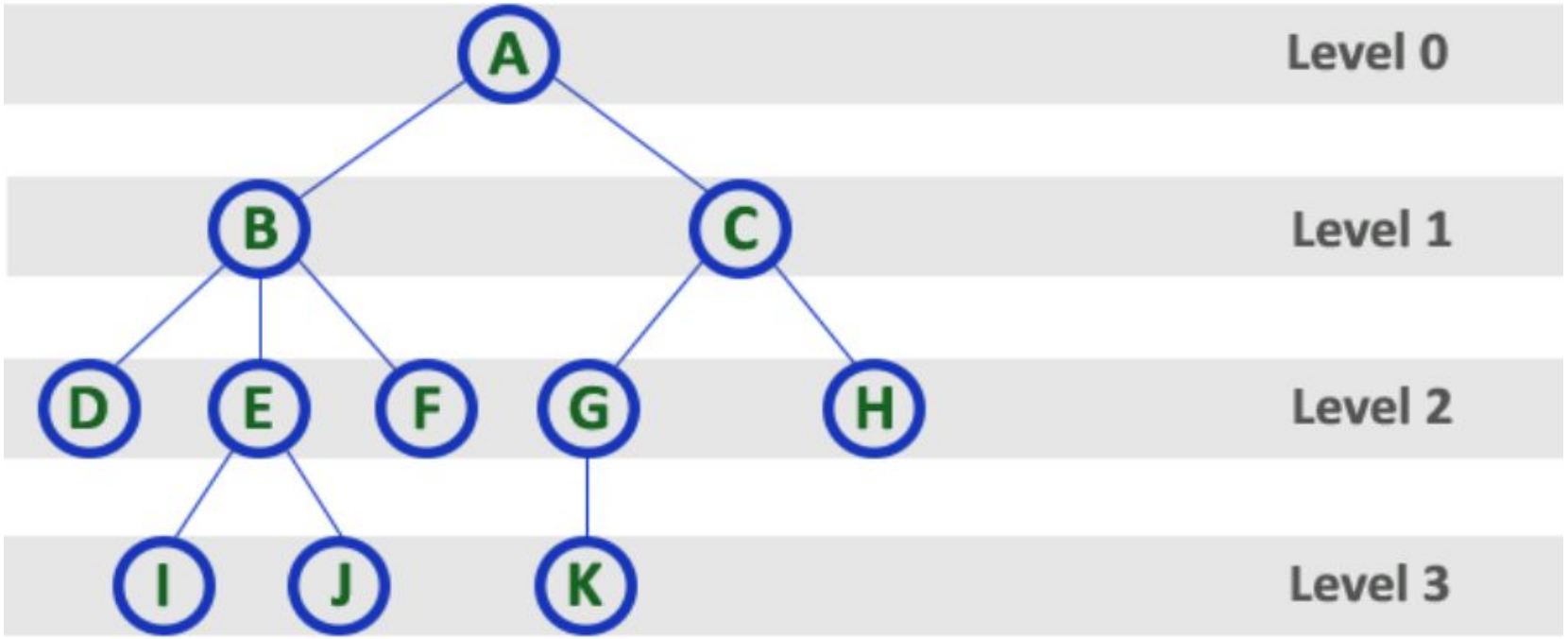
Here Degree of A is 2

Here Degree of F is 0

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

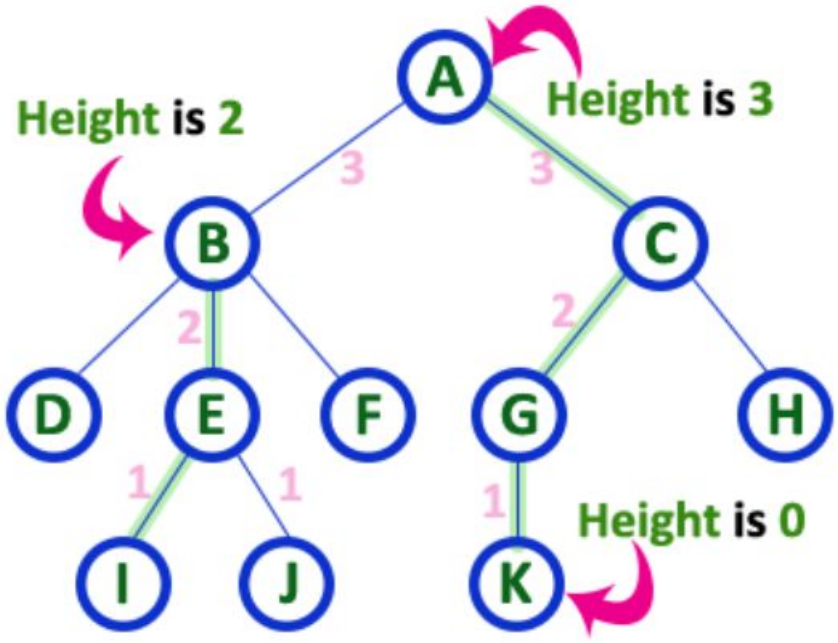
9. Level

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'.**

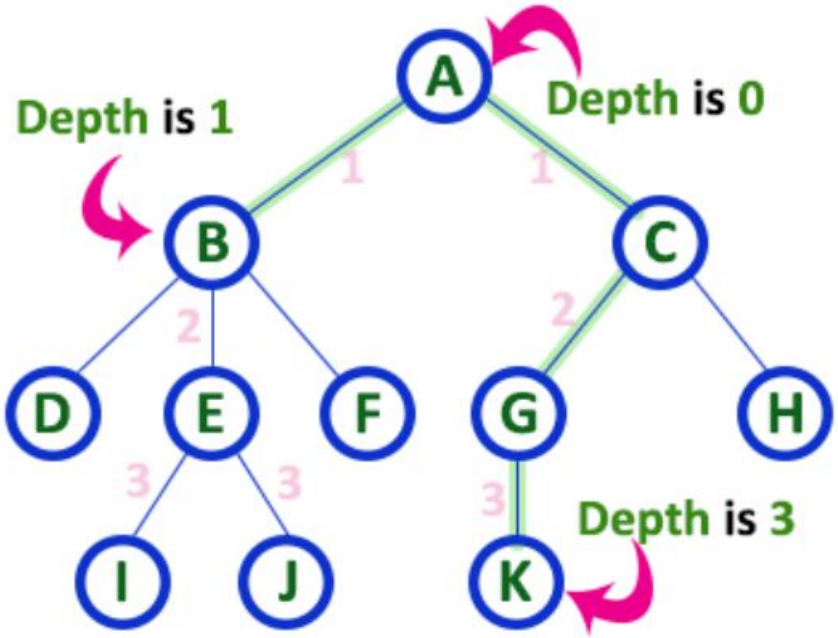


Here Height of tree is 3

- 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 edges 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'.**

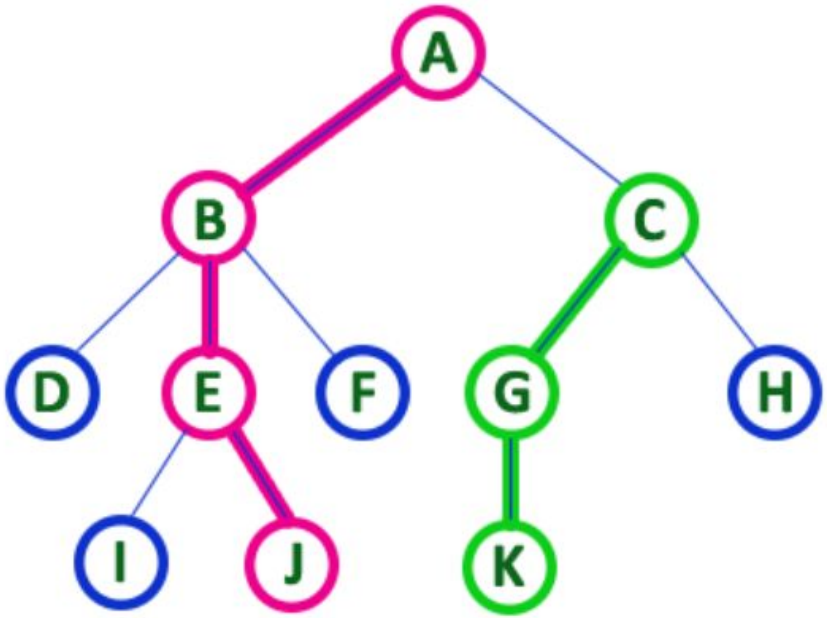


Here Depth of tree is 3

- 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.

12. Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



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

Here, 'Path' between A & J is

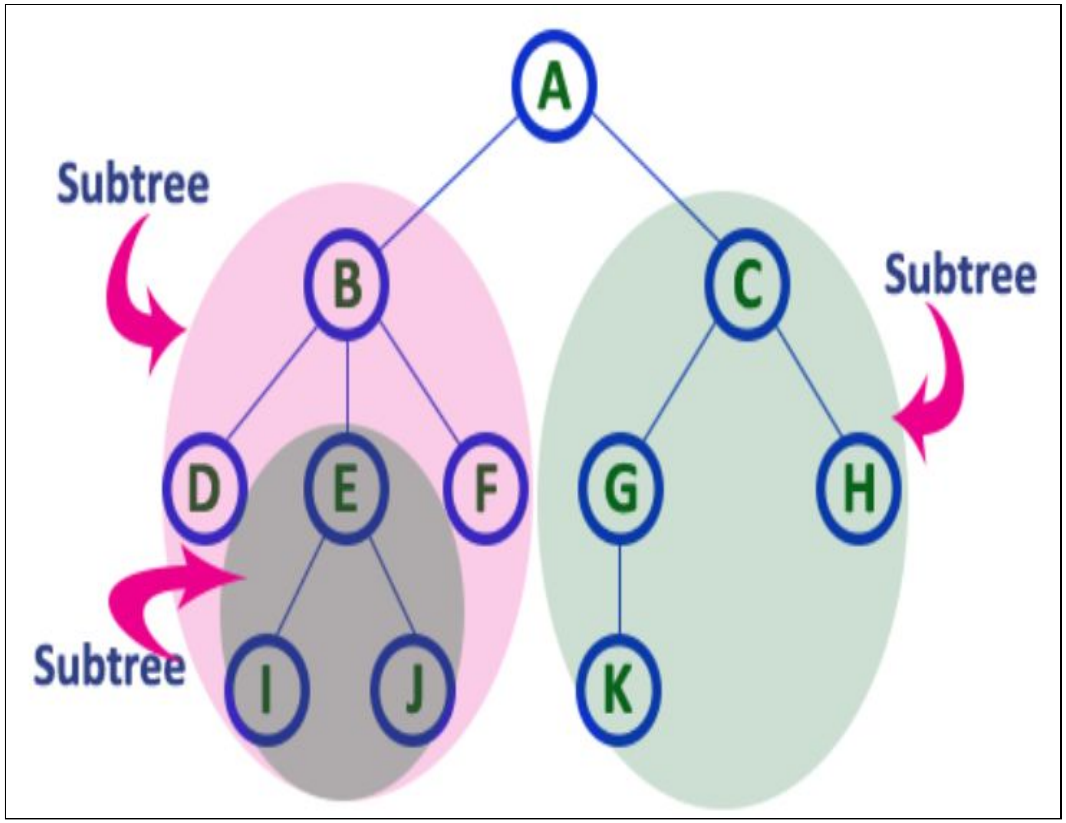
A - B - E - J

Here, 'Path' between C & K is

C - G - K

13. Sub Tree

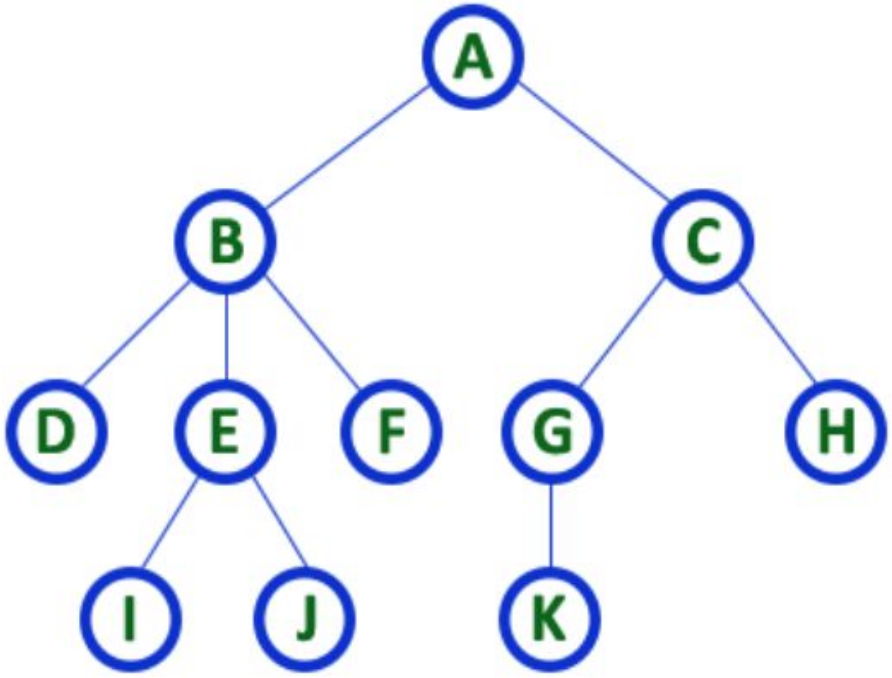
In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



Tree Representations

A tree data structure can be represented in two methods. Those methods are as follows...

1. **Sequential representation**
2. **List Representation**



TREE with 11 nodes and 10 edges

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

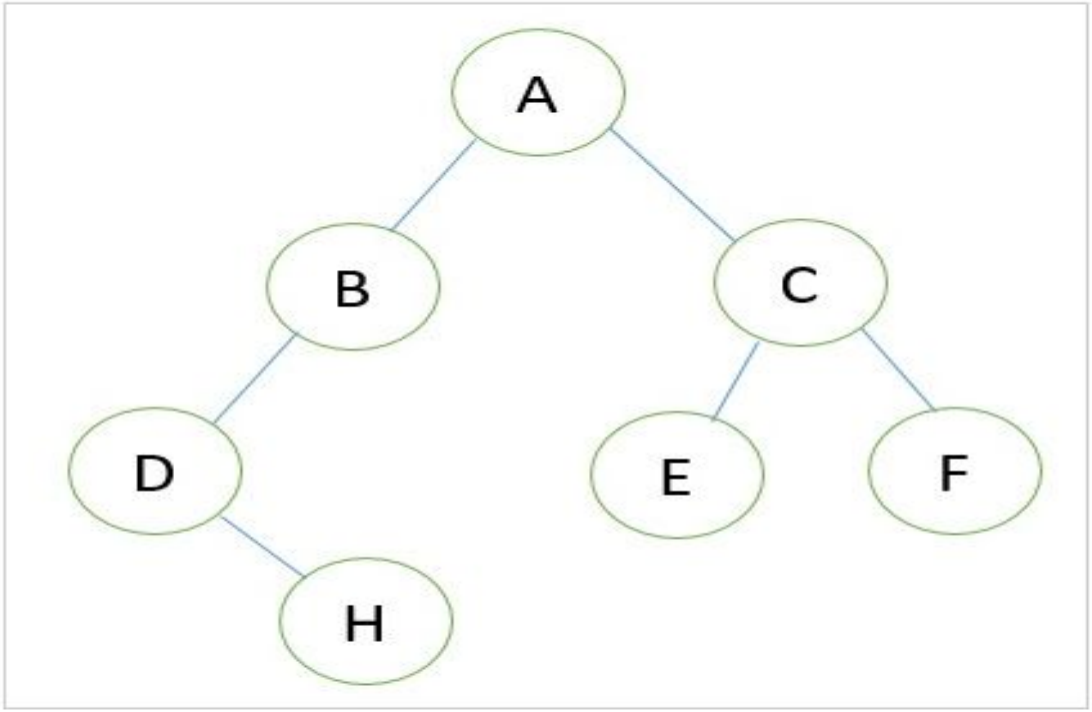
Sequential representation of Tree

- Let us consider that we have a tree **T**.
- let our tree **T** is a binary tree that us complete binary tree.
- Then there is an efficient way of representing **T** in the memory called the sequential representation or array representation of **T**. This representation uses only a linear array TREE as follows:
  - The root **N** of **T** is stored in **TREE [1]**.
  - If a node occupies **TREE [k]** then its left child is stored in **TREE [2 * k]** and its right child is stored into **TREE [2 * k + 1]**.

Tree Representations

For Example:

Consider the following Tree:

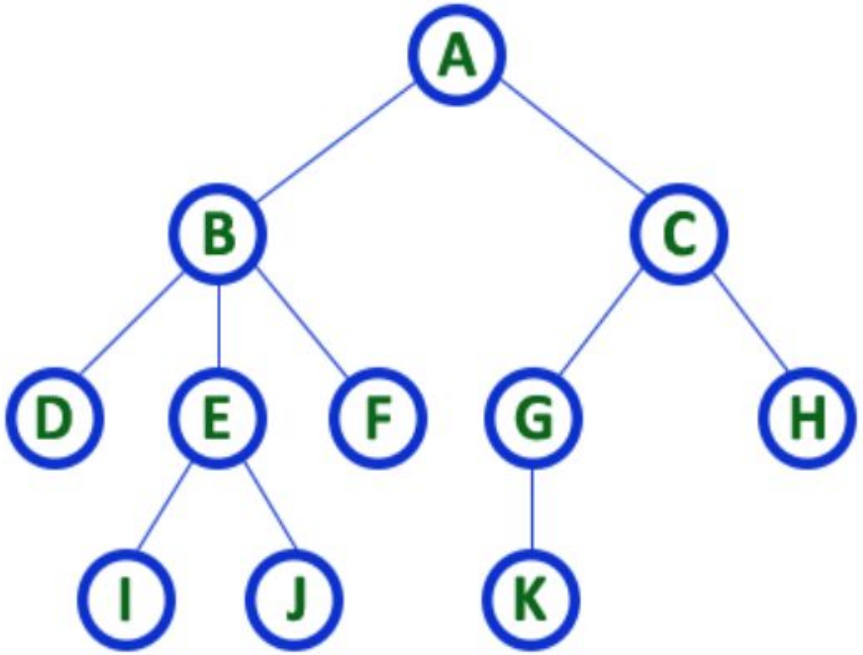


Its sequential representation is as follow:

A	B	C	D	-	E	F	-	H		

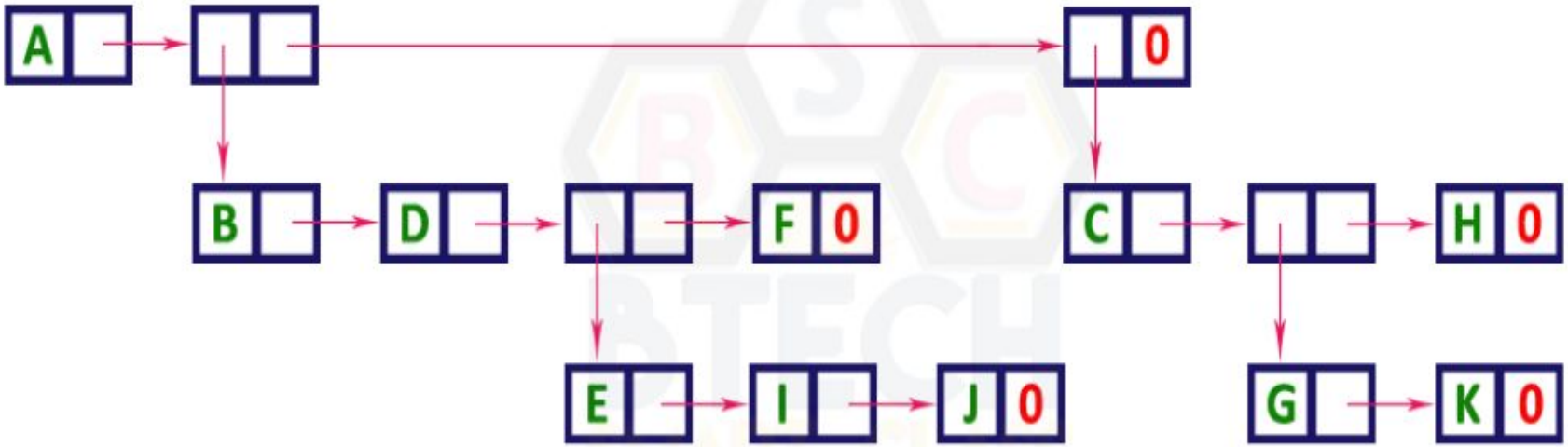
1. List Representation

- In this representation, we use two types of nodes one for representing the node with data called **'data node'** and another for representing only references called **'reference node'.**
- We start with a 'data node' from the root node in the tree.
- Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly.
- This process repeats for all the nodes in the tree.

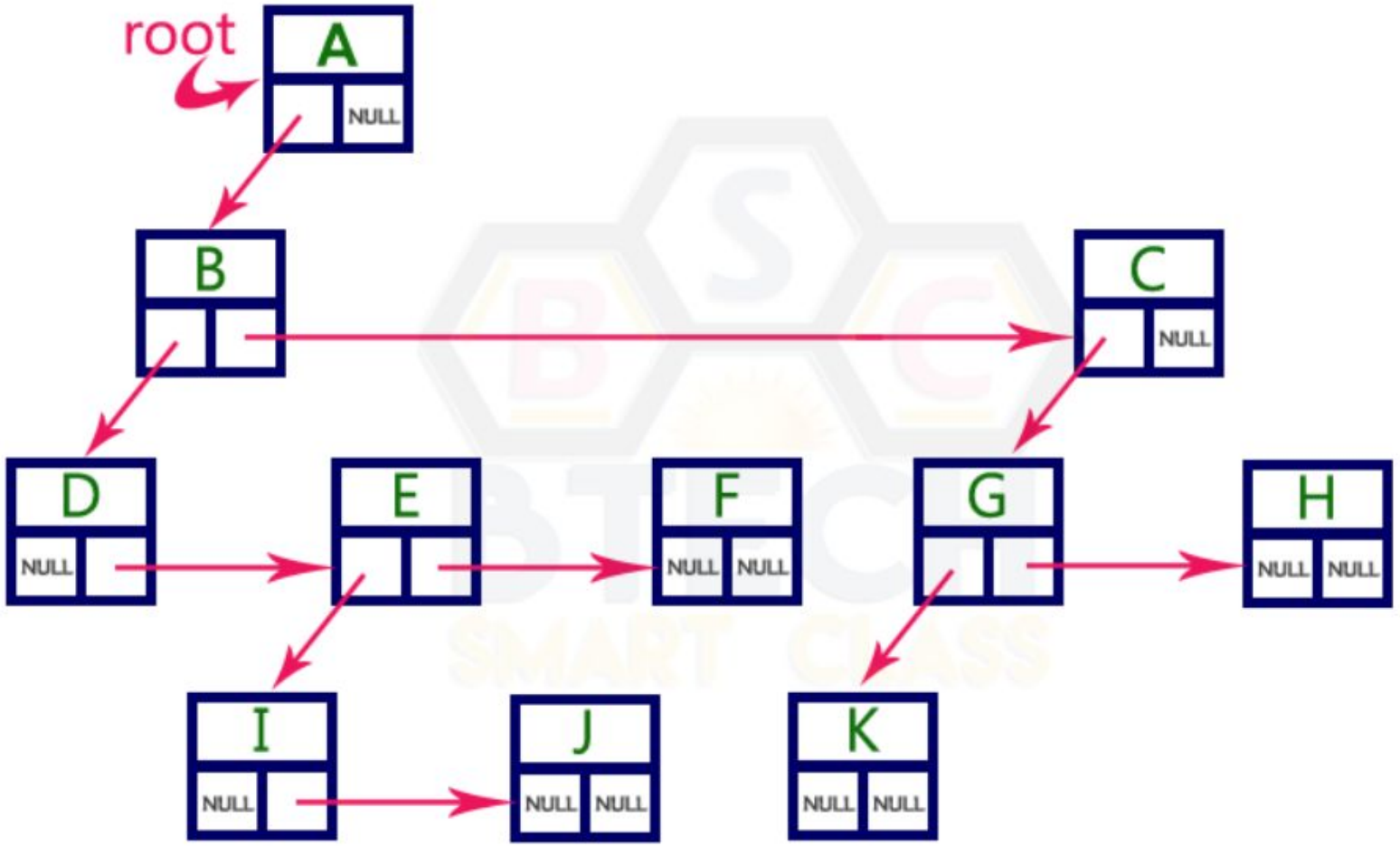


TREE with 11 nodes and 10 edges

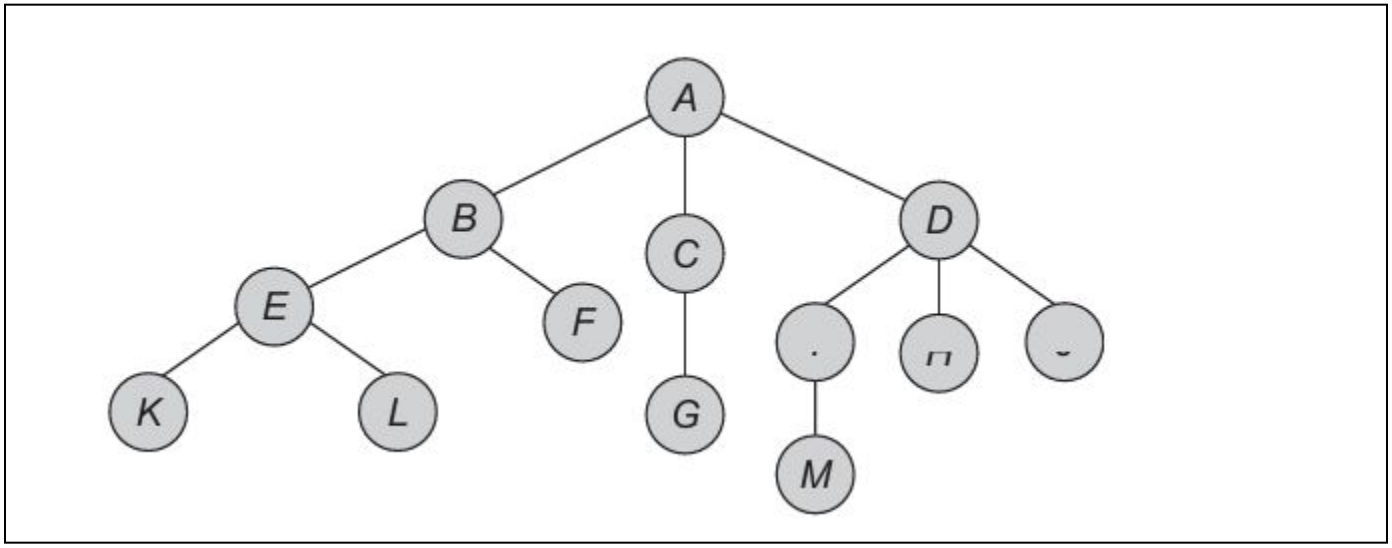
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

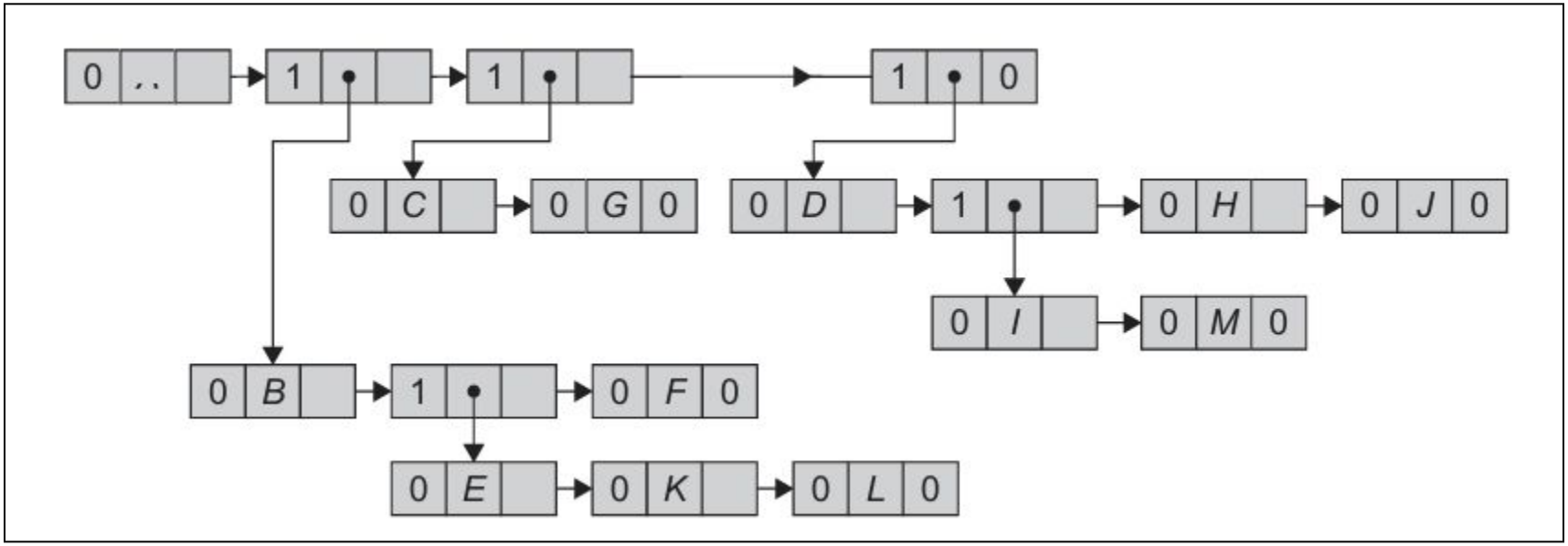


Left Child - Right Sibling Representation



Example - Draw the list representation of this tree





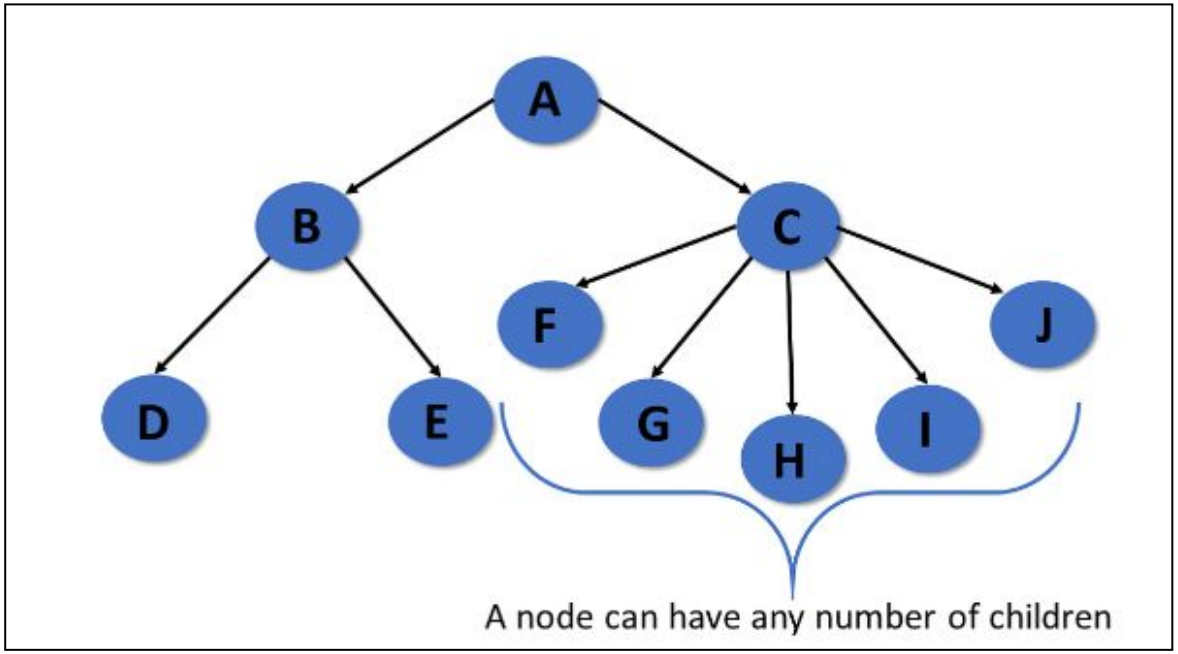
Types Of Trees

General Tree

The general tree is the type of tree where there are no constraints on the hierarchical structure.

Properties

1. The general tree follows all properties of the tree data structure.
2. A node can have any number of nodes.

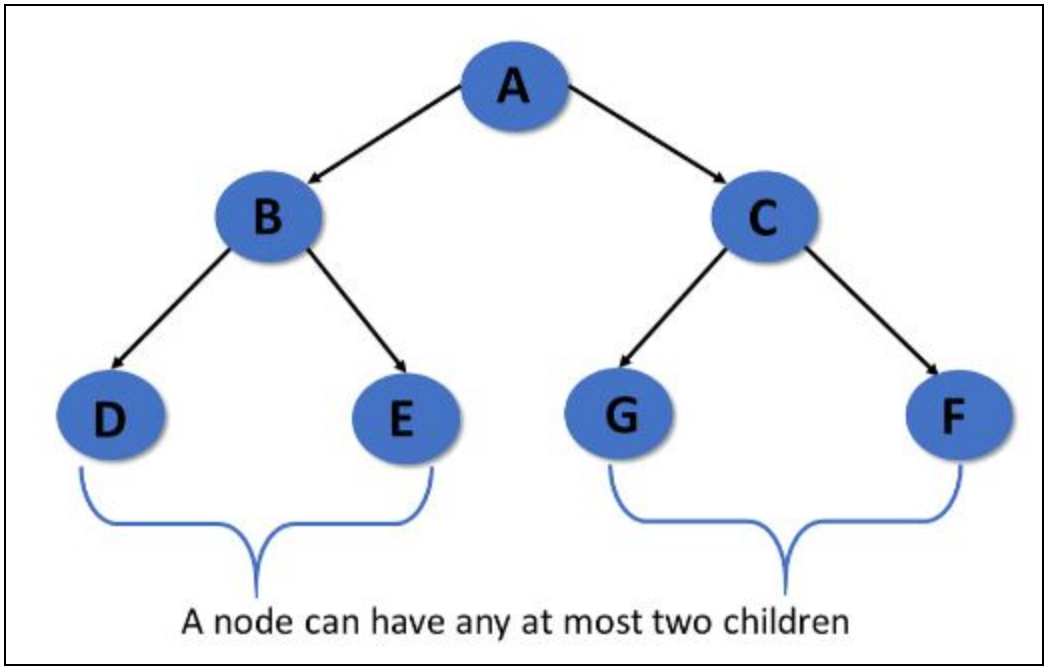


Binary Tree

A binary tree has the following properties:

Properties

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

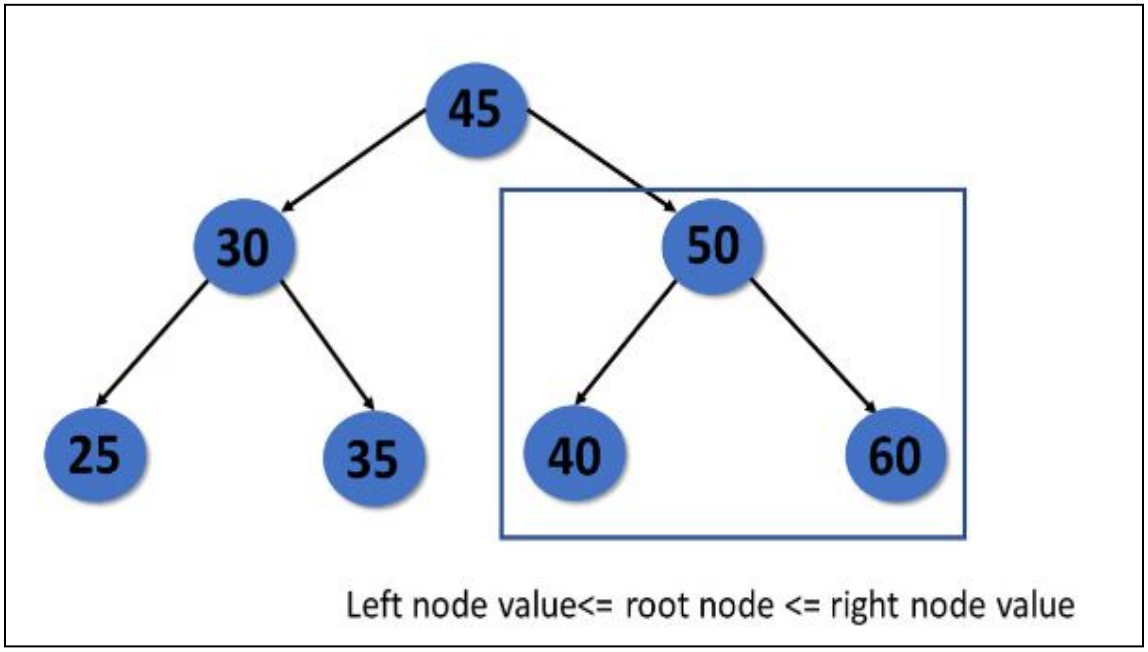


Binary Search Tree

A binary search tree is a type of tree that is a more constricted extension of a binary tree data structure.

Properties

1. Follows all properties of the tree data structure.
2. The binary search tree has a unique property known as the binary search property. This states that the value of a left child node of the tree should be less than or equal to the parent node value of the tree. And the value of the right child node should be greater than or equal to the parent value.

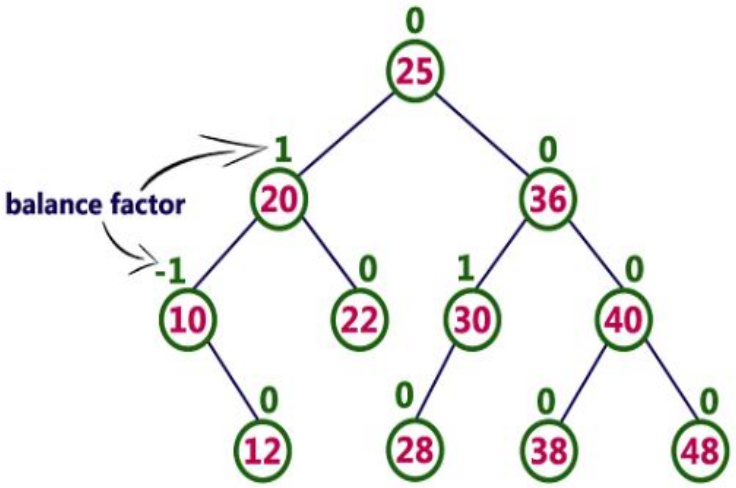


AVL Tree

An AVL tree is a type of tree that is a self-balancing binary search tree.

Properties

1. Follows all properties of the tree data structure.
2. Self-balancing.
3. Each node stores a value called a balanced factor, which is the difference in the height of the left sub-tree and right sub-tree.
4. All the nodes in the AVL tree must have a balance factor of -1, 0, and 1.



The balance factor of a node (N) in a binary tree is defined as the height difference.

BalanceFactor(N) = Height(RightSubtree(N)) — Height(LeftSubtree(N))

or

BalanceFactor(N) = Height(LeftSubtree(N)) — Height(RightSubtree(N))

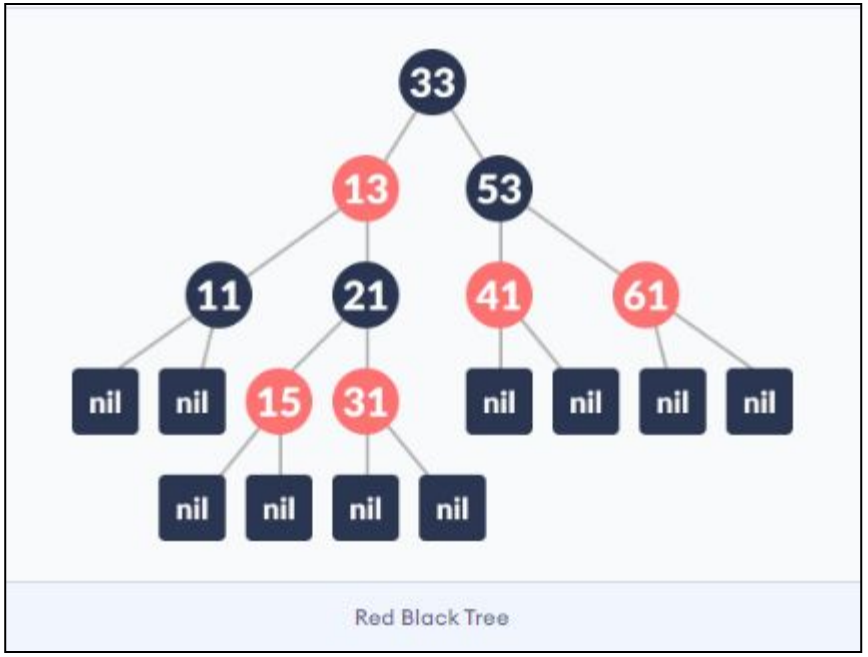
BalanceFactor(N) belongs to the set {-1,0,1}

Red-Black Tree

Red-Black tree is a self-balancing binary search tree in which each node contains an extra bit for denoting the color of the node, either red or black.

A red-black tree satisfies the following properties:

1. **Red/Black Property:** Every node is colored, either red or black.
2. **Root Property**: The root is black.
3. **Leaf Property:** Every leaf (NIL) is black.
4. **Red Property:** If a red node has children then, the children are always black.
5. **Depth Property:** Every simple path from a node to a descendant leaf contains the same number of black nodes.

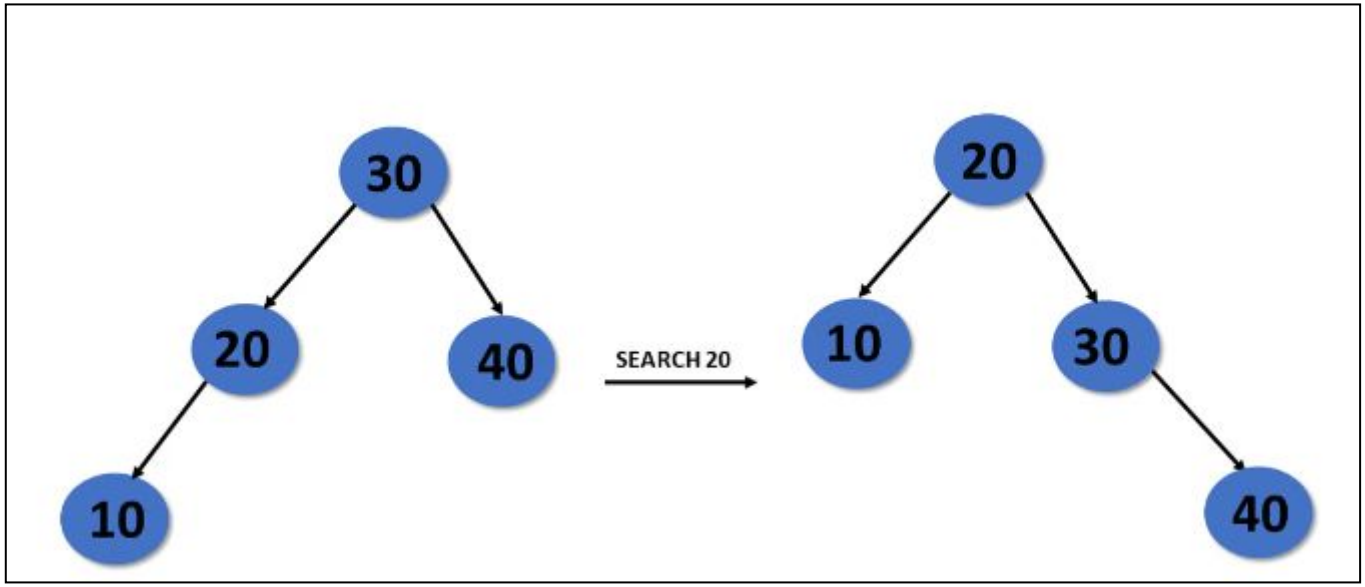


Splay Tree

A splay tree is a self-balancing binary search tree.

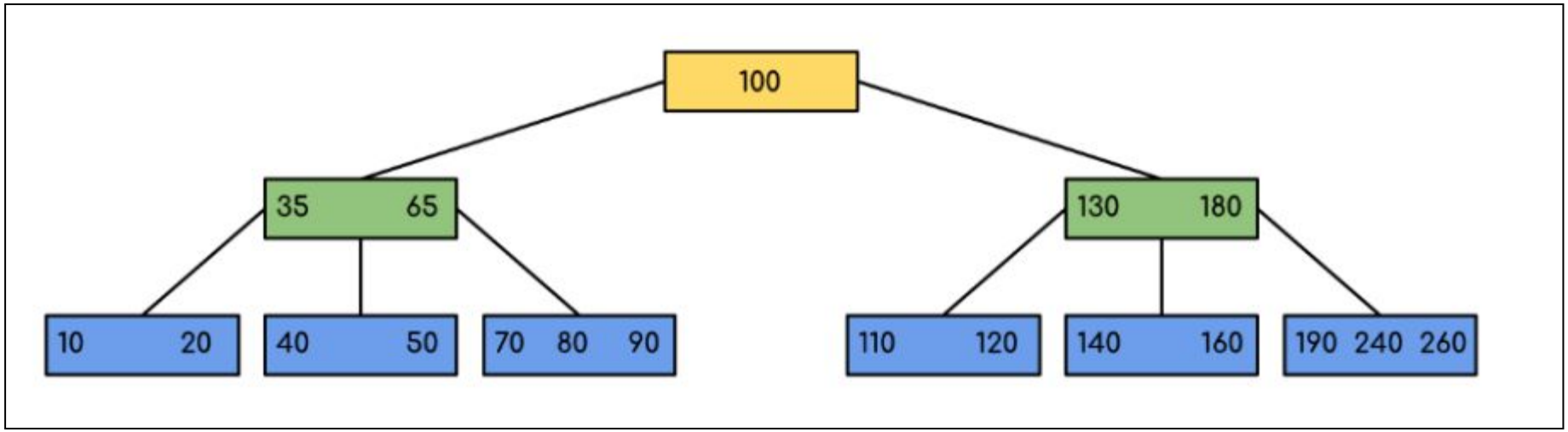
Properties

1. Follows properties of binary tree data structure.
2. Self-balancing.
3. Recently accessed elements are quicker to access again.
4. After you perform operations such as insertion and deletion, the splay tree acts, which is called splaying. Here it rearranges the tree so that the particular elements are placed at the root of the tree.



B-tree

- B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.
- It is also known as a height-balanced m-way tree.

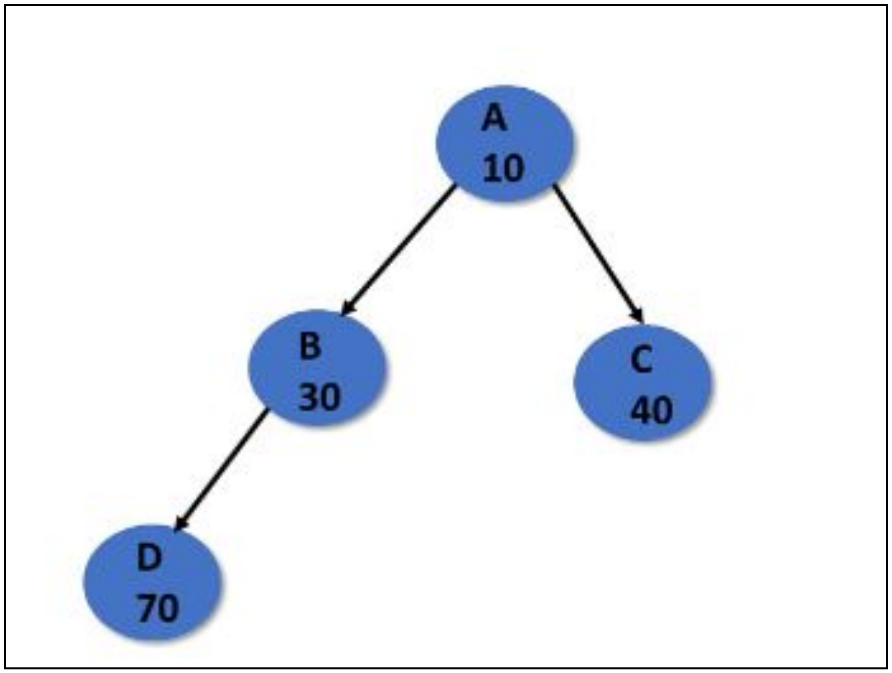


Treap Tree

The Treap tree is made up of a tree, and the heap is a binary search tree.

Properties

1. Each node has two values: a key and a priority.
2. Follows a binary search tree property.
3. Priority of the treap tree follows the heap property.



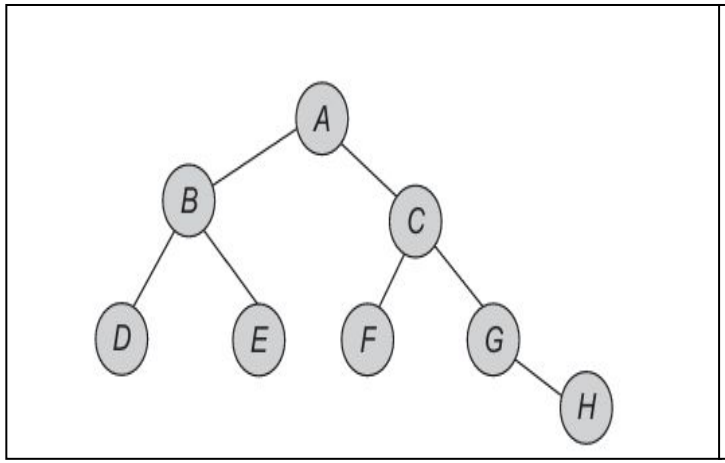
Here are some of the most common applications of trees in data structures:

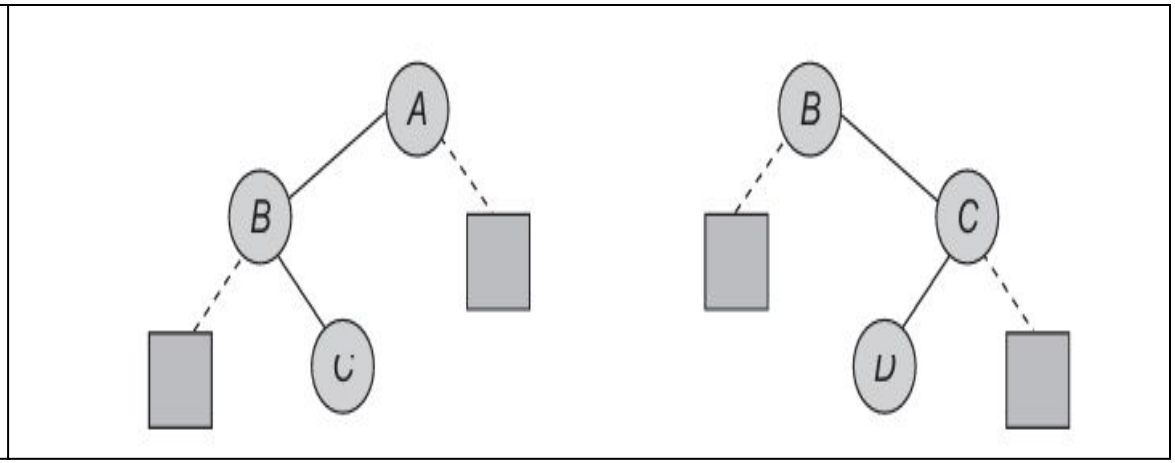
1. **Hierarchical Data Representation:** Trees are used to represent hierarchical data structures, such as file systems where files and directories have a parent-child relationship.
2. **Database Systems:** Trees, especially B-trees and B+ trees, are used in databases to enable quick searching, insertion, and deletion operations.
3. **Priority Queue Implementation:** Binary heaps, which can be visualized as trees, are used to implement priority queues.
4. **Syntax Tree:** In compilers, syntax trees are used to represent the structure of a program. They help in syntax checking and generating machine code.
5. **Network Routing Algorithms:** Trees are used in algorithms that find the shortest path in networking, like the OSPF (Open Shortest Path First) protocol.
6. **Decision Trees:** In machine learning, decision trees are used for classification and regression tasks. They split the data based on certain conditions to make decisions.
7. **Game Trees:** In artificial intelligence, trees are used to represent possible moves in a game. Games like chess or tic-tac-toe to decide the best move.
8. **Compression Algorithms:** Trees are used in algorithms like Huffman coding, which is used for data compression.

Binary tree

A binary tree has the degree two, with each node having at most two children.

Definition - A binary tree is either an empty tree or consists of a node, called root, and two children, left and right, each of which is itself a binary tree.



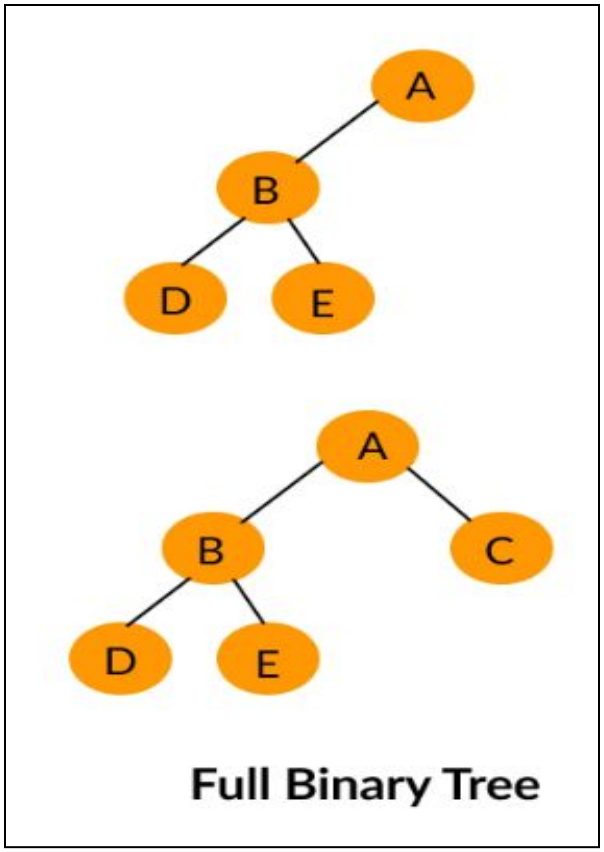


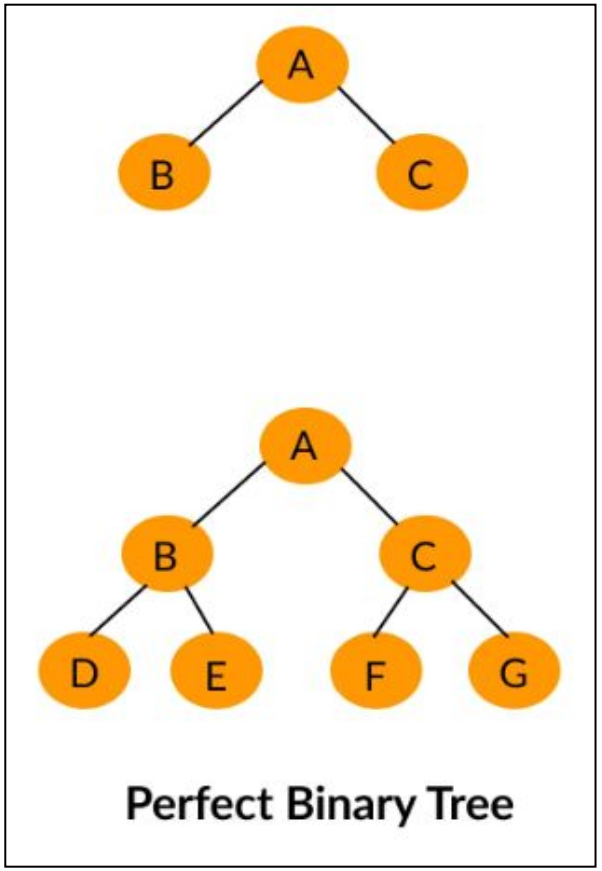
- A **binary tree** is a tree-type non-linear data structure with a maximum of two children for each parent.
- Every node in a **binary tree** has a left and right reference along with the data element.
- The node at the top of the hierarchy of a tree is called the **root node**.
- The nodes that hold other sub-nodes are the **parent nodes**.
- A parent node has two child nodes: **the left child and right child.**
- Hashing, routing data for network traffic, data compression, preparing binary heaps, and binary search trees are some of the applications that use a binary tree.

TYPES OF BINARY TREES:

1. **Full Binary Tree** - Every parent node has either two or no children.

2. **Perfect Binary Tree** - Every parent node has exactly two child nodes and all the external nodes (leaf nodes) are at the same level.

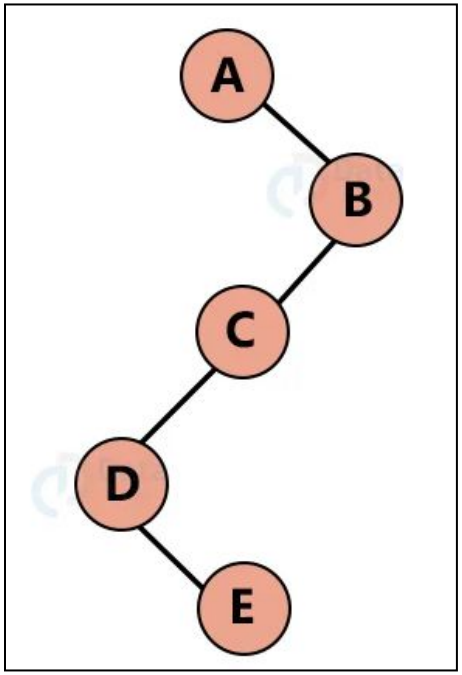




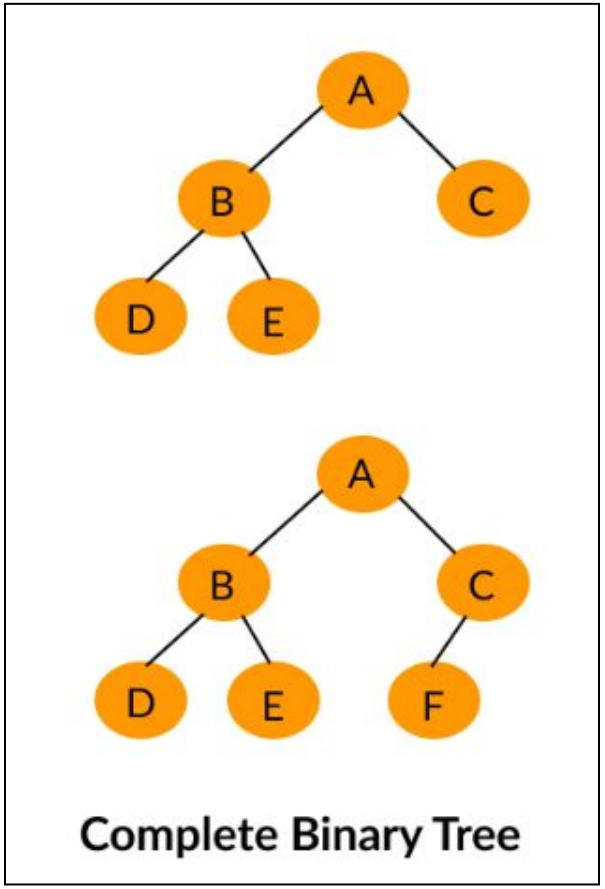
TYPES OF BINARY TREES:

3. Degenerate Tree - The tree in which the parent nodes have a single child, either left or right is called a degenerate tree.

4. Complete Binary Tree - In this case, every level must be filled, and all the leaf elements lean towards the left.



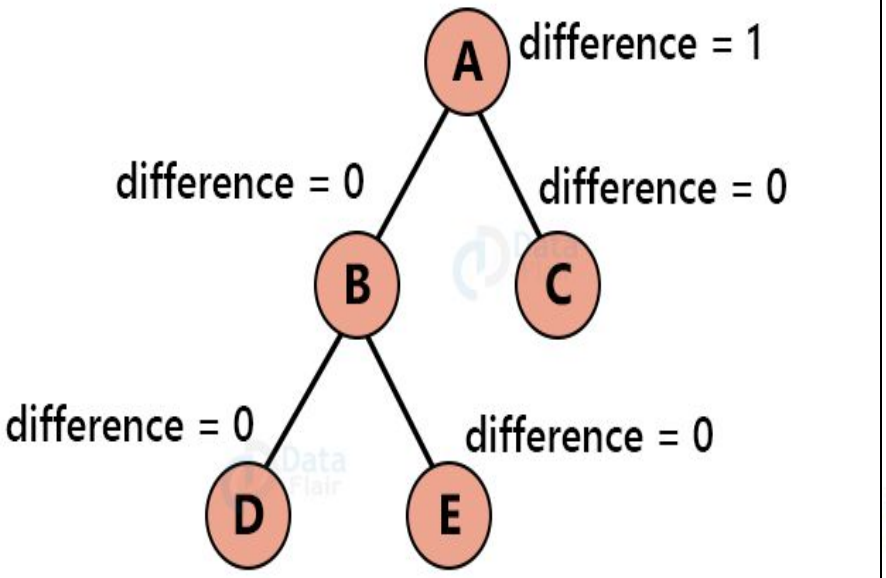
Degenerate Tree



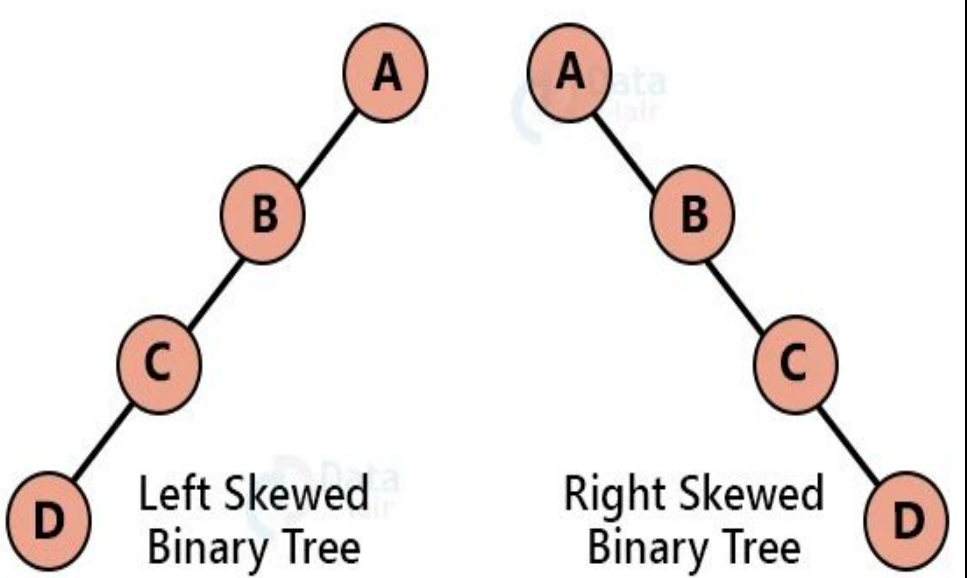
TYPES OF BINARY TREES:

5. Balanced Binary Tree - A binary tree in which the height difference between the left and the right child nodes is either 0 or 1.

6. Skewed Binary Tree - It is a binary tree in which new nodes can be added only to one side of the binary tree then it is a skewed binary tree.



Balanced Binary Tree



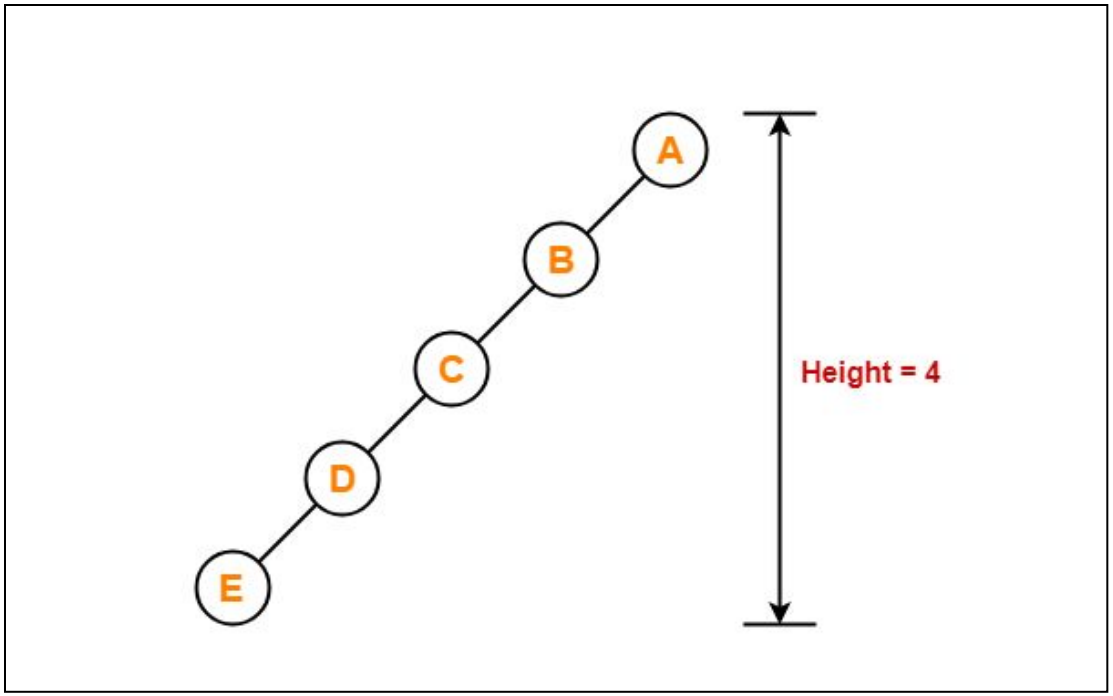
Skewed Binary Tree

PROPERTIES OF A BINARY TREE

1. **Minimum number of nodes in a binary tree of height H = H + 1**

Example-

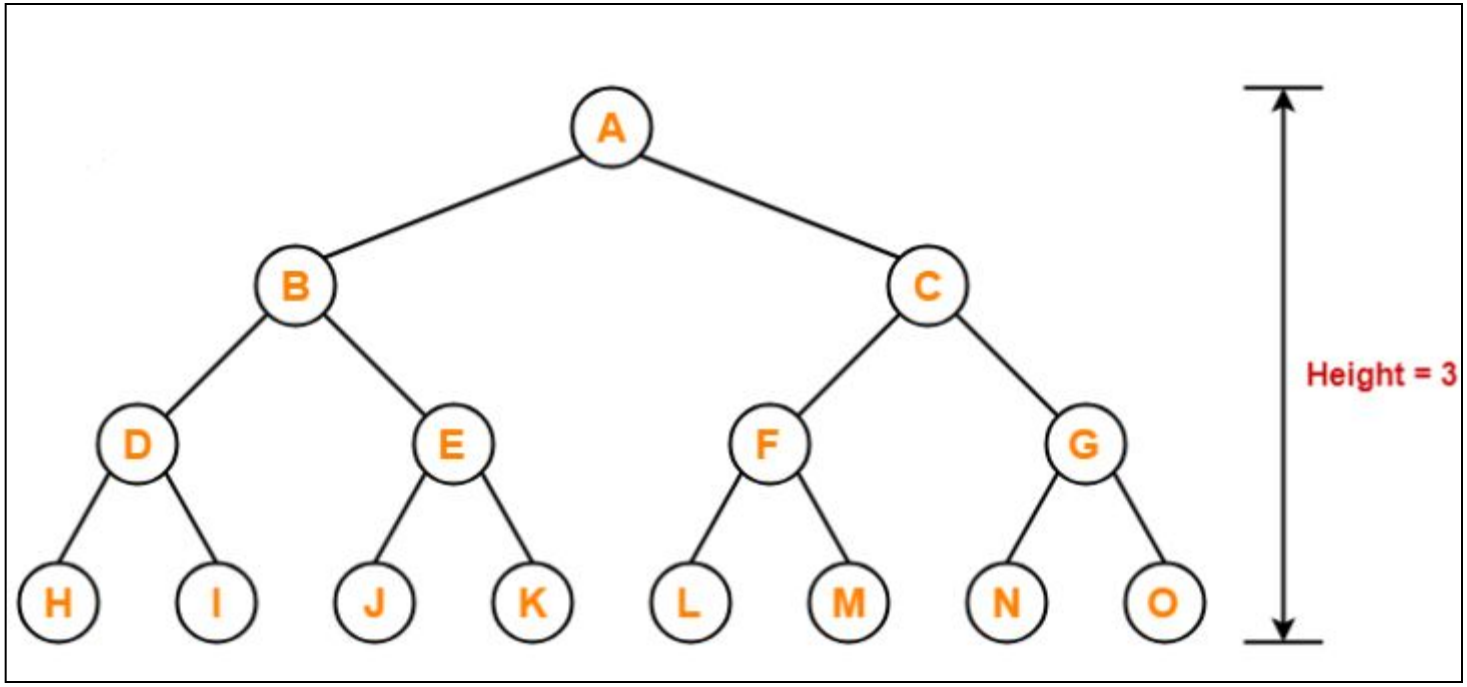
To construct a binary tree of height = 4, we need at least 4 + 1 = 5 nodes.



2. Maximum number of nodes in a binary tree of height H = $2^{H+1} - 1$

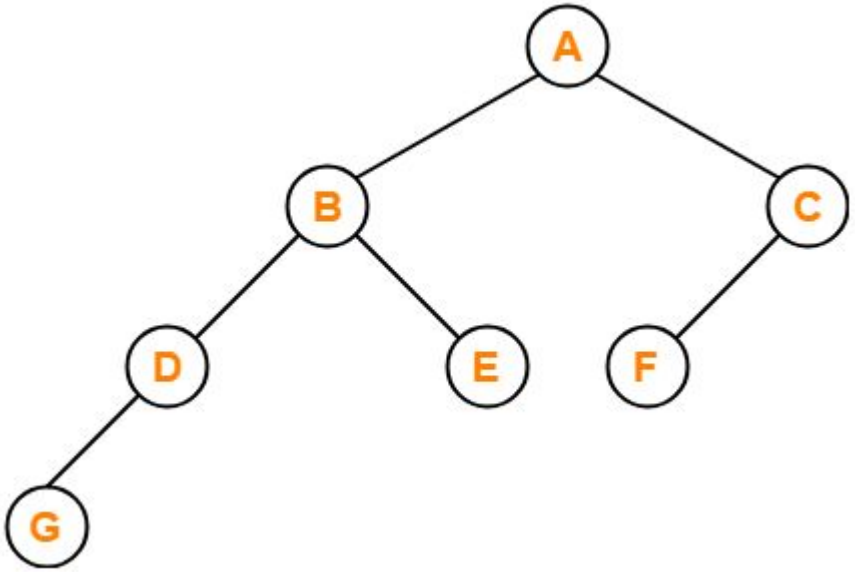
Example- Maximum number of nodes in a binary tree of **height 3**

$= 2^{3+1} - 1$
$= 16 - 1$
$= 15$ nodes
Thus, in a binary tree of height = 3, maximum number of nodes that can be inserted = 15.



3. Total Number of leaf nodes in a Binary Tree = Total Number of nodes with 2 children + 1

Example - Consider the following binary tree-



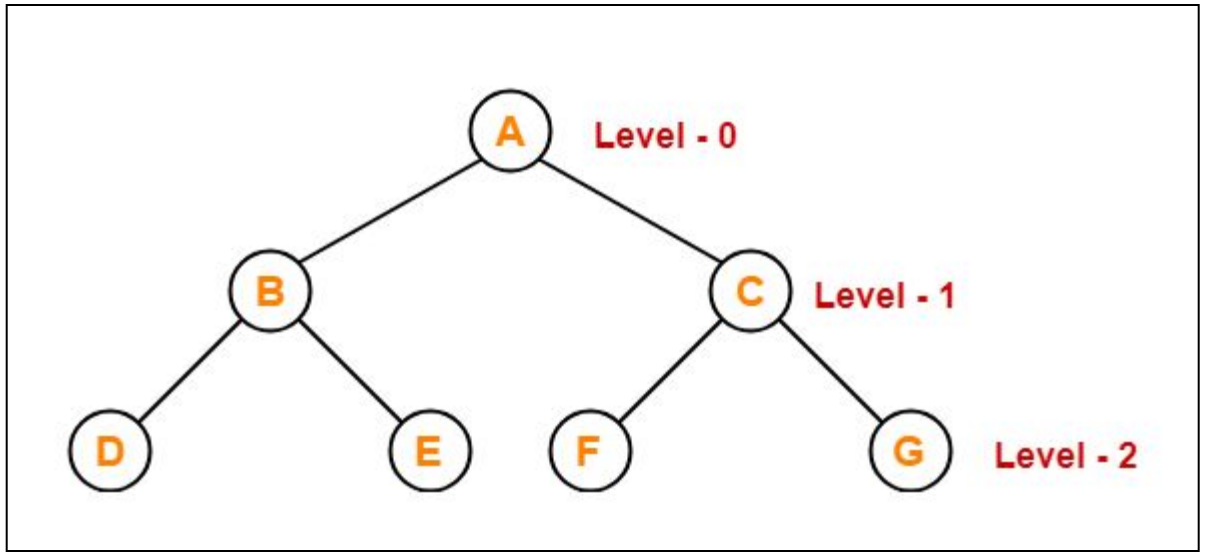
Here,
Number of leaf nodes = 3
Number of nodes with 2 children = 2

Here ,number of leaf nodes is one greater than number of nodes with 2 children.
This verifies the above relation

4. Maximum number of nodes at any level 'L' in a binary tree = 2^L

Example- Maximum number of nodes at level-2 in a binary tree = $2^2 = 4$

Thus, in a binary tree, maximum number of nodes that can be present at level-2 = 4.



Conversion of general trees into the binary trees

A general tree can be changed into an equivalent binary tree. This conversion process or technique is called the natural correspondence between general and binary trees.

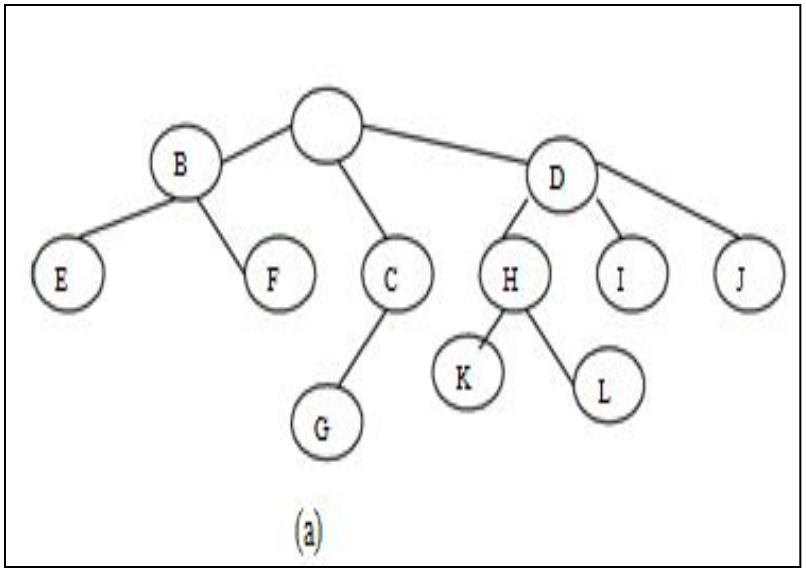
The algorithm is written below:

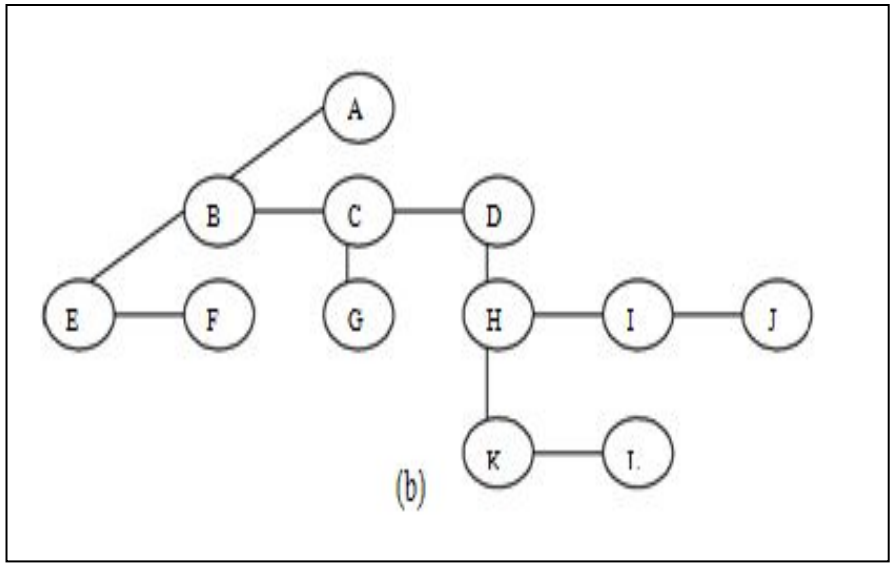
(a) Insert the edges connecting siblings from left to right at the same level.

(b Erase all edges of a parent to its children except to its left most offspring.

(c) Rotate the obtained tree 45^0 to mark clearly left and right subtrees.

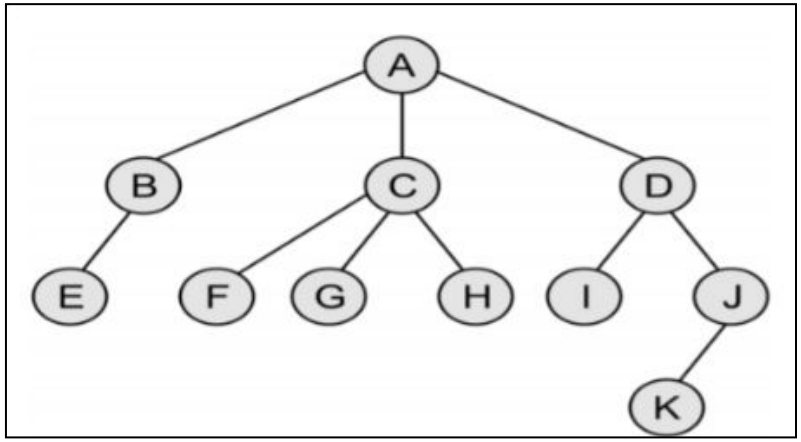
A general tree shown in figure (a) converted into the binary tree shown in (b)

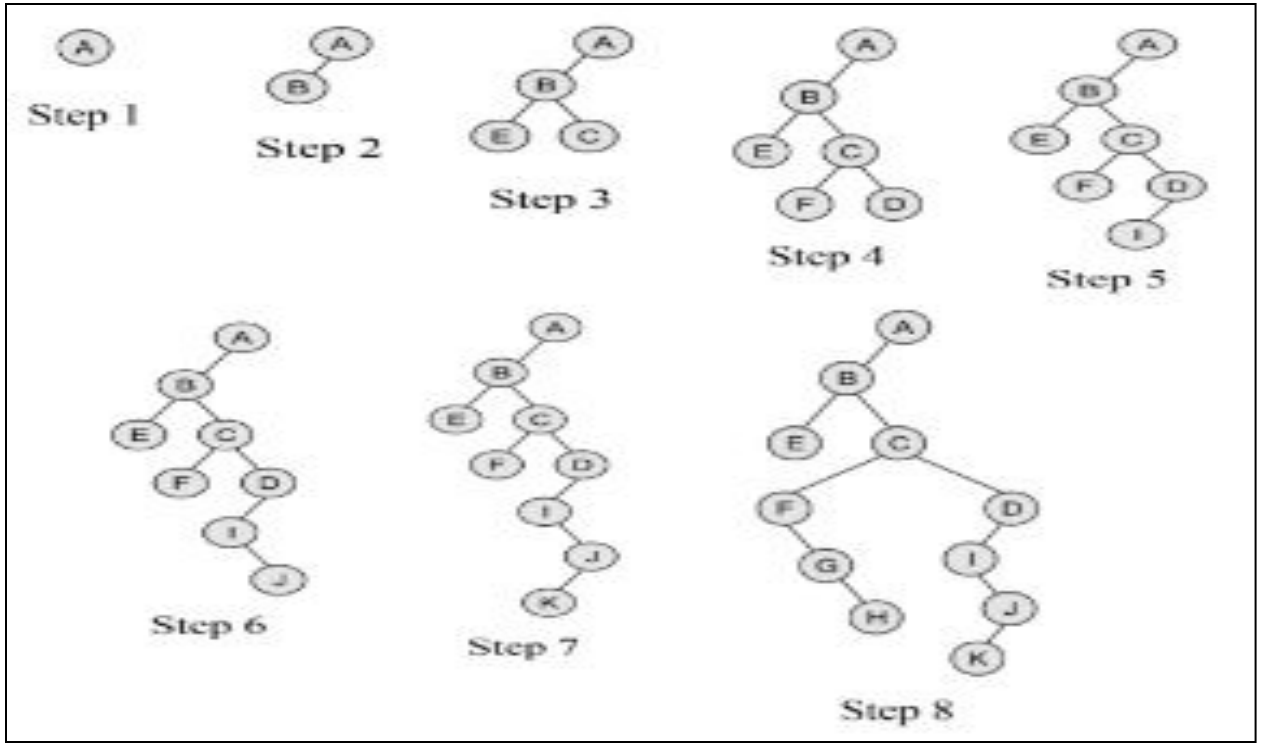




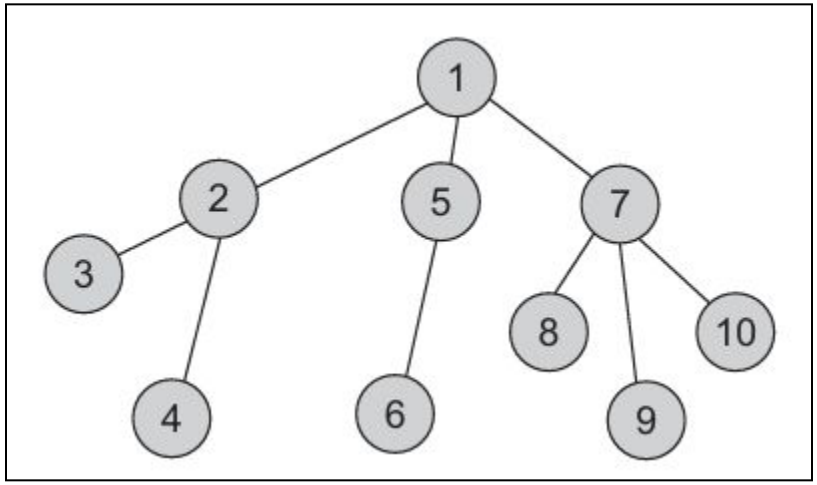
Examples:

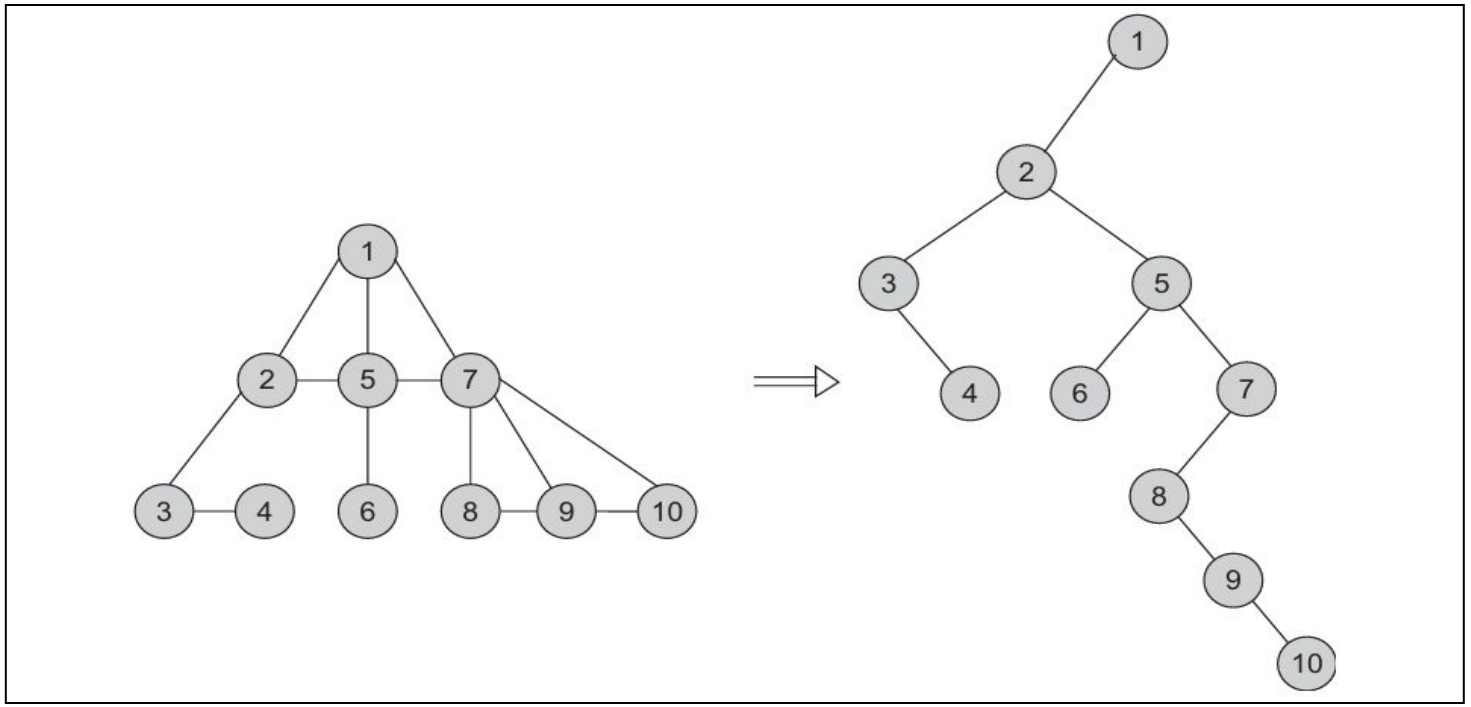
Convert the following Generic Tree to Binary Tree:



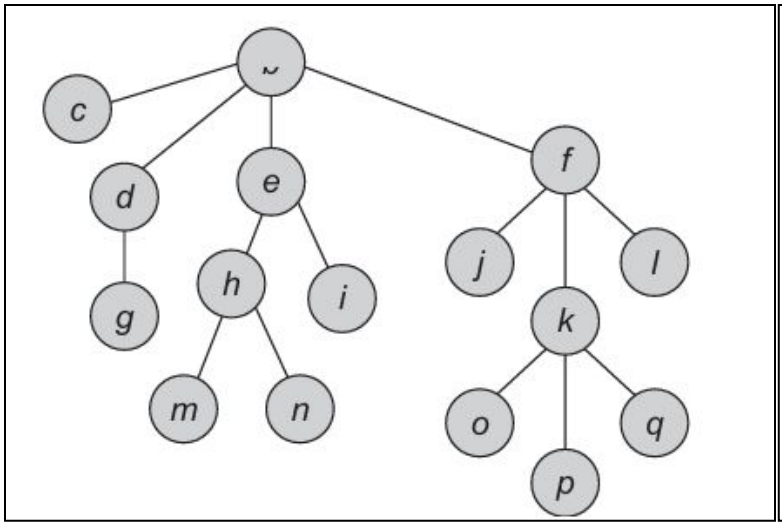


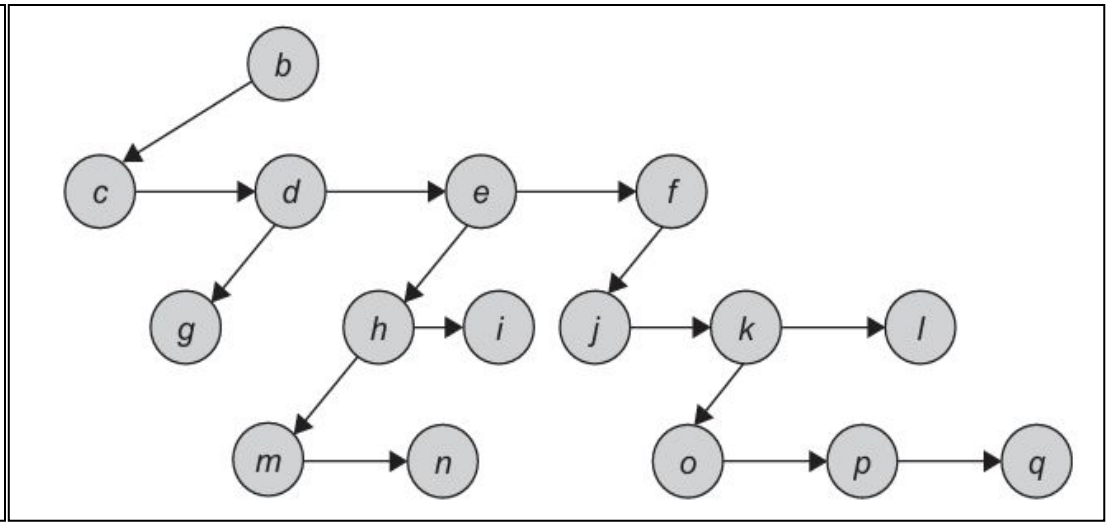
Convert the general tree in Fig. into its corresponding binary tree.

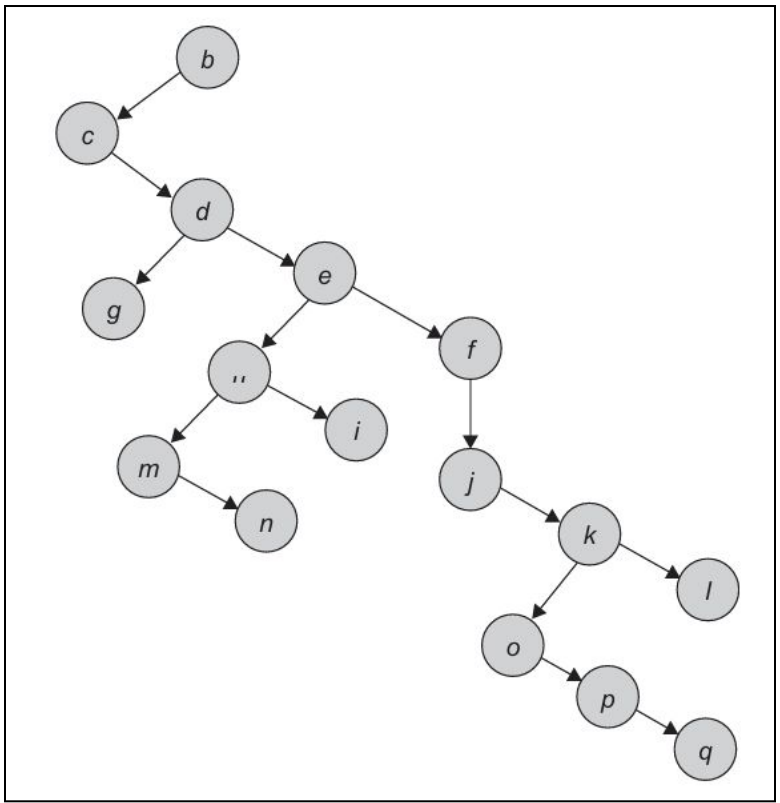




Convert the general tree in Fig. into its corresponding binary tree.







Operations on binary tree –

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

1. **Creation—**Creating an empty binary tree to which the 'root' points
2. **Traversal—**Visiting all the nodes in a binary tree
3. **Deletion—**Deleting a node from a non-empty binary tree
4. **Insertion—**Inserting a node into an existing (may be empty) binary tree
5. **Merge—**Merging two binary trees
6. **Copy—**Copying a binary tree
7. **Compare—**Comparing two binary trees 8. Finding a replica or mirror of a binary tree

ADT OF BINARY TREE

Abstract datatype Binary tree

{

instances:

a finite set of nodes either empty or consisting of a root node, left Binary_tree and right Binary_tree

operations:

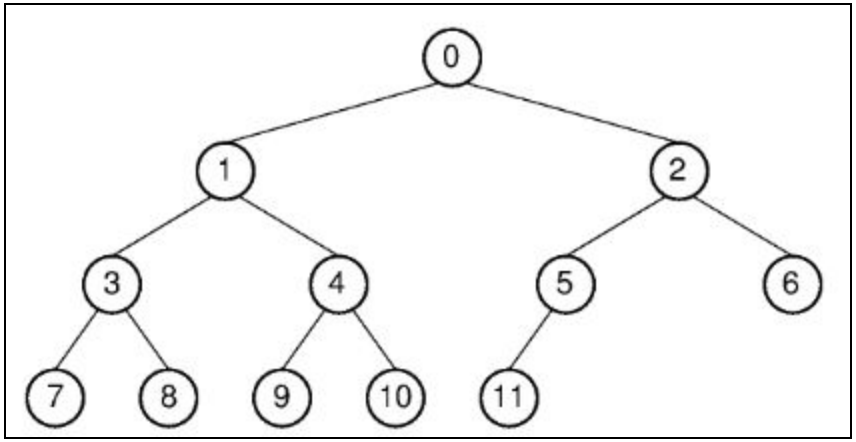
```
Binary_tree();  // create an empty binary tree
bool Isempty(bt); //return true iff the binary tree is empty
Binary_tree maketree((Binary_tree bt1,item,Binary_tree bt2) - return binary tree
whose left subtree is bt1 and whose right subtree is bt2 and whose
root node contains data item.
Binary_tree LeftSubtree(bt) -  return the left subtree of bt.
Binary_tree RightSubtree(bt) - return the right subtree of bt.
Binary_tree RootData(bt) - return the data in the root node of bt.
```

}

Array implementation of binary trees

- One of the ways to represent a tree using an array is to store the nodes level-by-level, starting from the level 0 where the root is present.
- Such a representation requires sequential numbering of the nodes, starting with the nodes on level 0, then those on level 1, and so on.
- The root node is stored in the first memory location as the first element in the array.
- The following rules can be used to decide the location of any i^{th} node of a tree:
    - For any node with index i, $0 \leq i \leq n - 1$

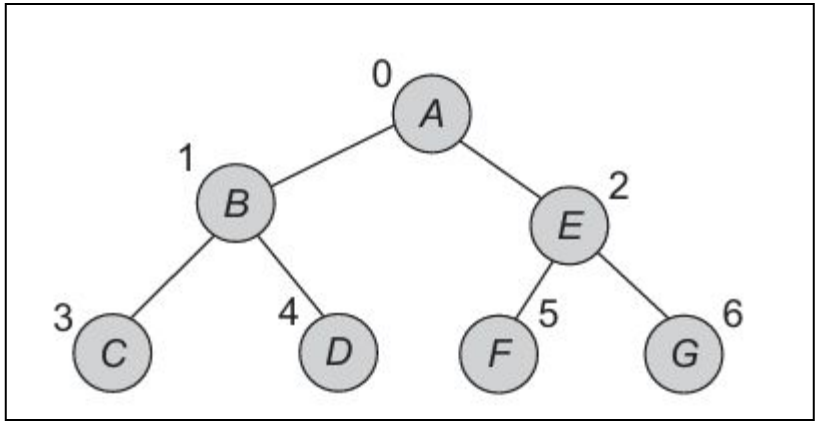
1. Parent(i) $= \lfloor (t-1)/2 \rfloor$ if $i \neq 0$; if $i = 0$, then it is the root that has no parent.
2. Lchild(i) $= 2 \times i + 1$ if $2i + 1 \leq n - 1$; if $2i \geq n$, then i has no left child.
3. Rchild(i) $= 2i + 2$ if $2i + 2 \leq n - 1$; if $(2i + 1) \geq n$, then i has no right child.



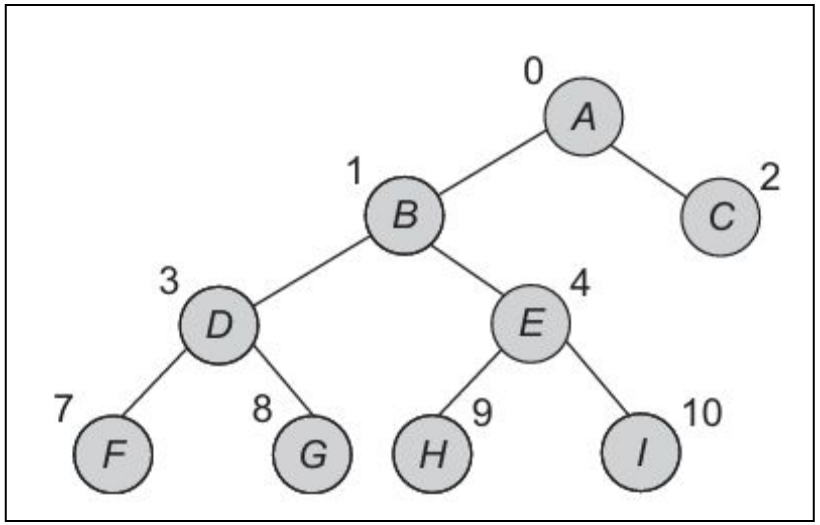
- Parent$(r) = \lfloor (r-1)/2 \rfloor$ if $r \neq 0$.
- Left child$(r) = 2r+1$ if $2r+1 < n$.
- Right child$(r) = 2r+2$ if $2r+2 < n$.

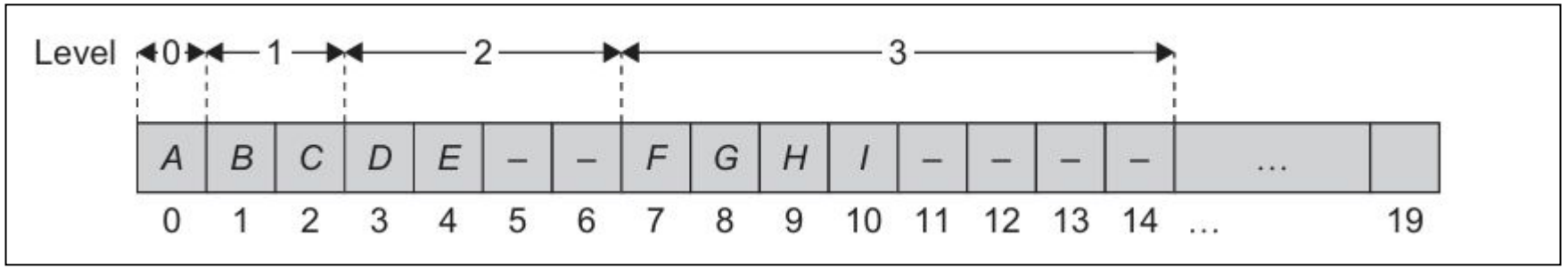
Position	0	1	2	3	4	5	6	7	8	9	10	11
Parent	– –	0	0	1	1	2	2	3	3	4	4	5
Left Child	1	3	5	7	9	11	– –	– –	– –	– –	– –	– –
Right Child	2	4	6	8	10	– –	– –	– –	– –	– –	– –	– –
Left Sibling	– –	– –	1	– –	3	– –	5	– –	7	– –	9	– –
Right Sibling	– –	2	– –	4	– –	6	– –	8	– –	10	– –	– –

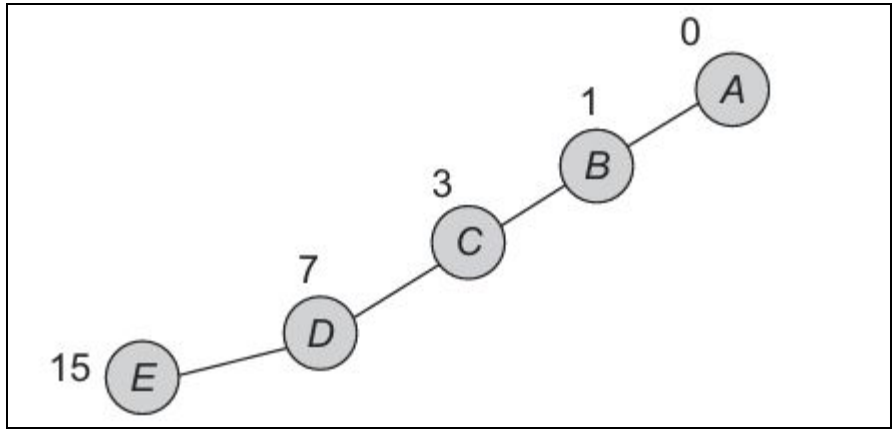
Let us consider the complete binary tree

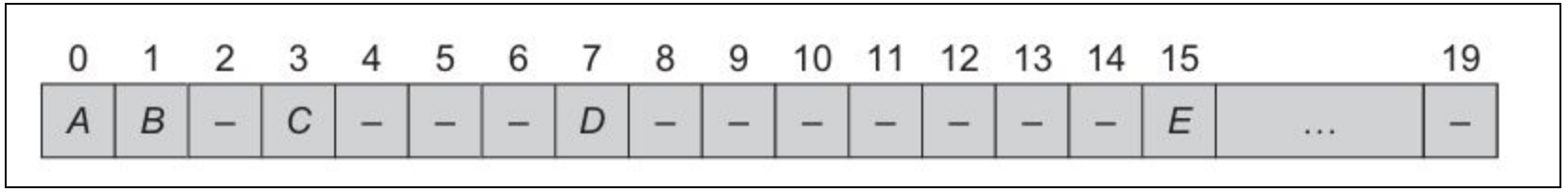


0	1	2	3	4	5	6	7	8
A	B	E	C	D	F	G	–	–






0
A
1
B
3
C
7
D
15
E


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 19
A B – C – – – D – – – – – – – E … –

```
#include<bits/stdc++.h>
using namespace std;
char tree[10];
int rootnode(char key)
{
   if(tree[0] != '\0')
      cout<<"Tree already had root";
   else
      tree[0] = key;
   return 0;
}
int leftchild(char key, int parent){
   if(tree[parent] == '\0')
     cout <<"\nCan't set child at"<<(parent * 2) + 1<<" , no
parent found";
   else
      tree[(parent * 2) + 1] = key;
   return 0;
}
```

```
int rightchild(char key, int parent){
   if(tree[parent] == '\0')
         cout<<"\nCan't set child at"<<(parent * 2) + 2<<" , no parent
found";
   else
      tree[(parent * 2) + 2] = key;
   return 0;
}
```

```
int traversetree(){
   cout << "\n";
   for(int i = 0; i < 10; i++){
      if(tree[i] != '\0')
         cout<<tree[i];
      else
         cout<<"-";
   }
   return 0;
}
int main(){
   rootnode('A');
   rightchild('C', 0);
   leftchild('D', 0);
   rightchild('E', 1);
   leftchild('P', 2);
   leftchild('O', 3);
   traversetree();
   return 0;
}
```

```
Can't set child at 7 , no parent found
ADC-EP----
```

Advantages

1. Any node can be accessed from any other node by calculating the index.

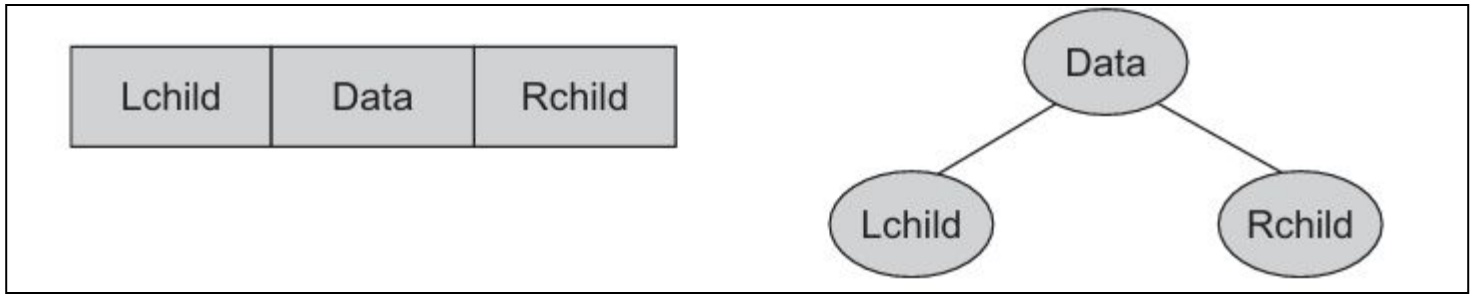
2. Here, the data is stored without any pointers to its successor or predecessor.

3. In the programming languages, where dynamic memory allocation is not possible (such as BASIC, FORTRAN), array representation is the only means to store a tree.

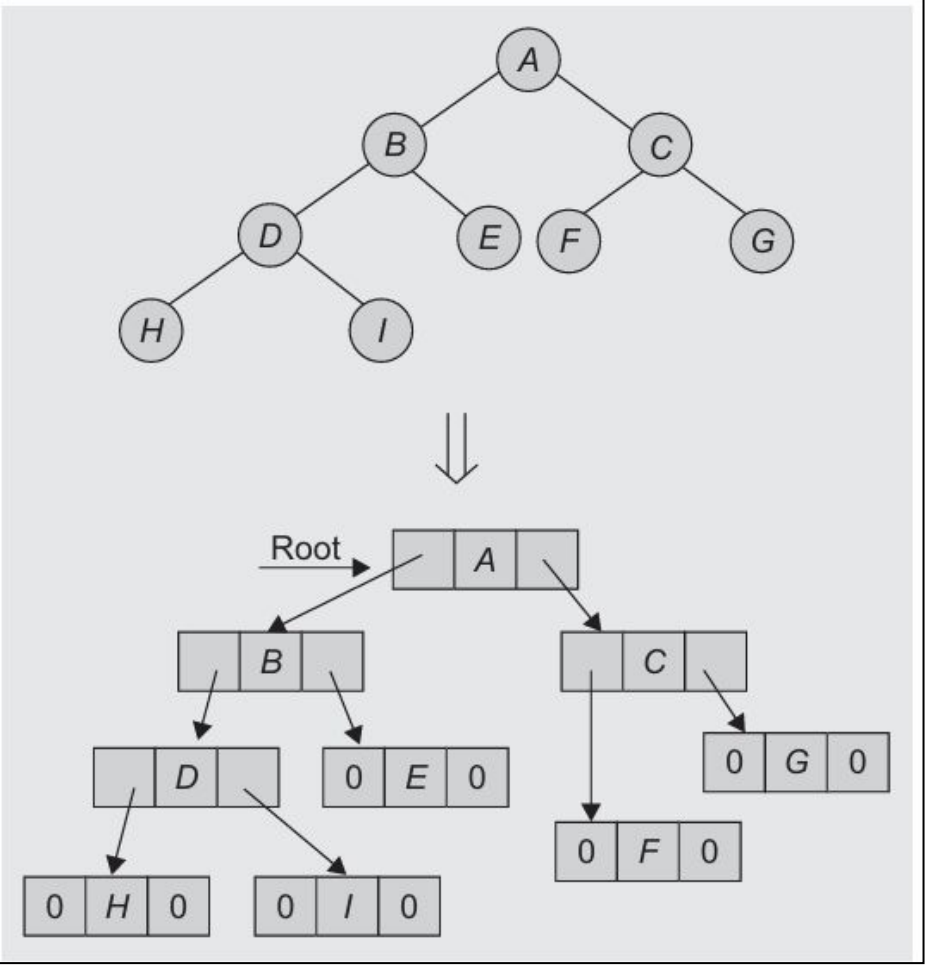
Disadvantages

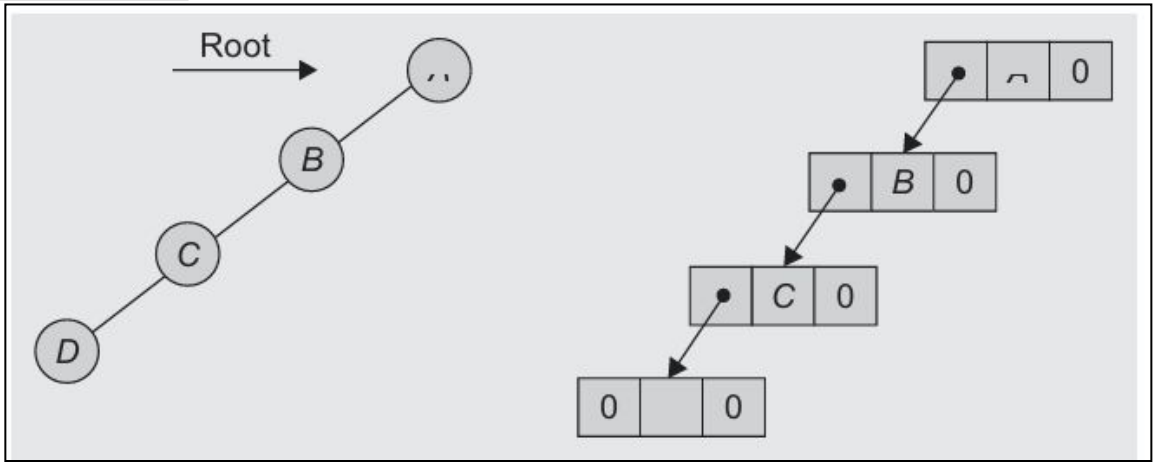
1. Other than full binary trees, majority of the array entries may be empty.
2. It allows only static representation. The array size cannot be changed during the execution.
3. Inserting a new node to it or deleting a node from it is inefficient with this representation, because it requires considerable data movement up and down the array, which demand excessive amount of processing time.

Linked implementation of binary trees



- The root of the tree is stored in the data member root of the tree. This data member provides an access pointer to the tree.
- In this node structure, Lchild and Rchild are the two link fields to store the addresses of left child and right child of a node; data is the information content of the node.
- With this representation, if we know the address of the root node, then using it, any other node can be accessed.
- Each node of a binary tree (as the root of some subtree) has both left and right subtrees.


A
B
C
D
E
F
G
H
I
Root
A
B
C
D
0 E 0
0 G 0
0 F 0
0 H 0
0 I 0

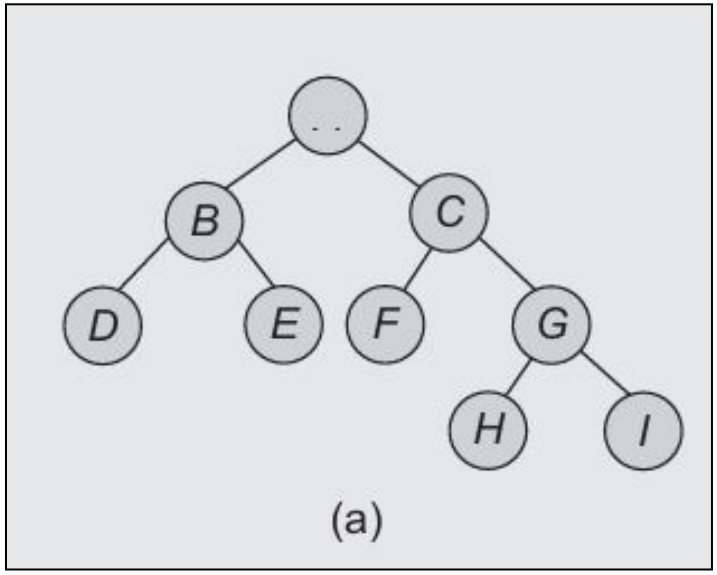

Root
B
C
D
0
B 0
C 0
0 0

Tree and its views

(a) Binary tree

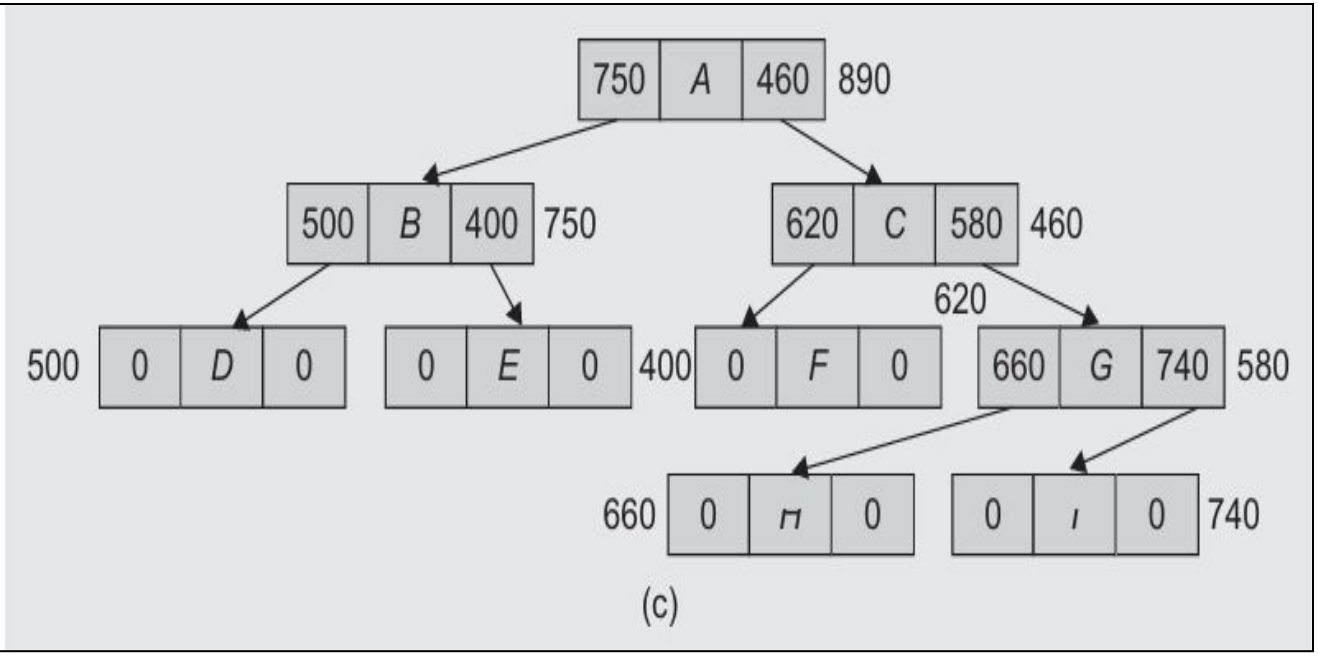
(b) Physical view

(c) Logical view



Address	Child	Data	Rchild
500	0	D	0
750	500	B	400
400	0	E	0
890	750	A	460
620	0	F	0
460	620	C	580
660	0	H	0
580	660	G	740
740	0	I	0

(b)



Structure for a Node

```
#include <iostream>


struct Node {
  int item;
  Node* left;
  Node* right;
};
```

Create a new Node

```
Node* create(int value) {
  Node* newNode = new Node;
  newNode->item = value;
  newNode->left = nullptr;
  newNode->right = nullptr;

  return newNode;
}
```

Advantages

1. The drawbacks of the sequential representation are overcome in this representation. We may or may not know the tree depth in advance. In addition, for unbalanced trees, the memory is not wasted.
2. Insertion and deletion operations are more efficient in this representation.
3. It is useful for dynamic data.

Disadvantages

1. In this representation, there is no direct access to any node. It has to be traversed from the root to reach to a particular node.
2. As compared to sequential representation, the memory needed per node is more. This is due to two link fields (left child and right child for binary trees) in the node.
3. The programming languages not supporting dynamic memory management would not be useful for this representation.

INSERTION OF A NODE IN BINARY TREE

- The **insert()** operation inserts a new node at any position in a binary tree.
- The node to be inserted could be a branch node or a leaf node.
- The branch node insertion is generally based on some criteria that are usually in the context of a special form of a binary tree.
- **The insertion procedure is a two-step process**.

1. Search for the node whose child node is to be inserted. This is a node at some level i, and a node is to be inserted at the level i + 1 as either its left child or right child. This is the node after which the insertion is to be made.
2. Link a new node to the node that becomes its parent node, that is, either the Lchild or the Rchild.

Insertion of a node in binary tree is represented in fig

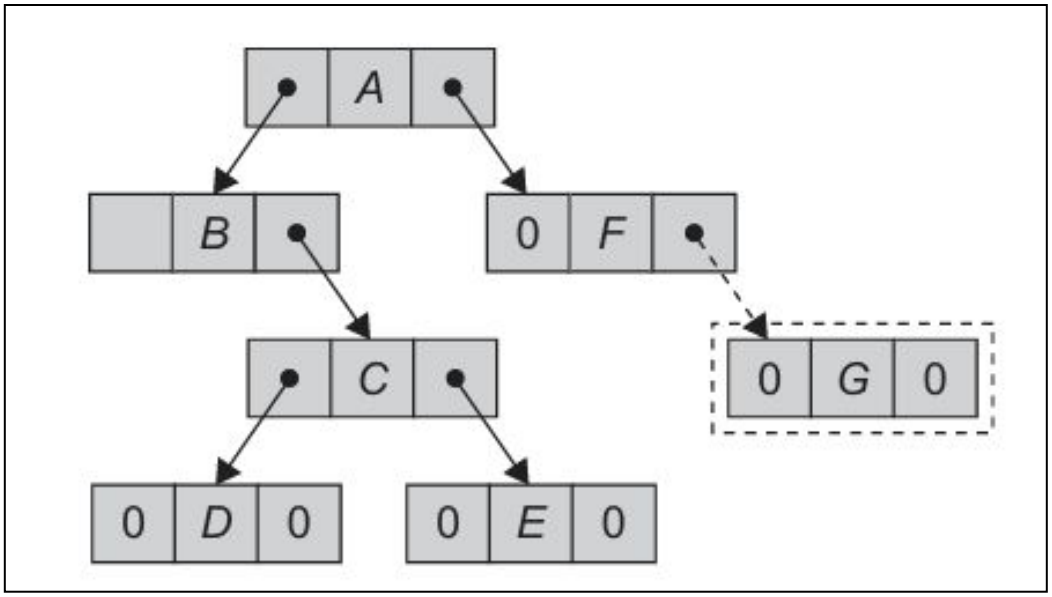


Fig. Insertion of node G as the Rchild of node F

Insert on the left of the node ->

```
Node* insertLeft(Node* root, int value)
{
  root->left = create(value);
  return root->left;
}
```

Insert on the right of the node ->

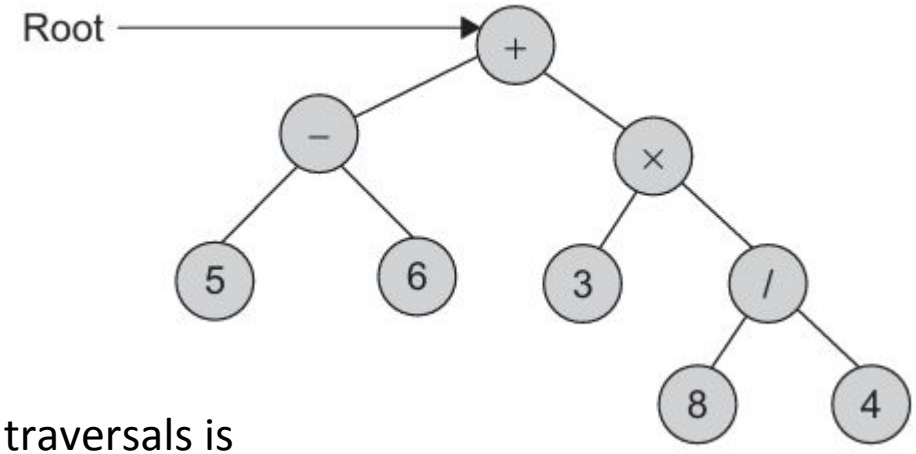
```
Node* insertRight(Node* root, int value)
 {
  root->right = create(value);
  return root->right;
}
```

```
int main()
{
  Node* root = create(1);
  insertLeft(root, 4);
  insertRight(root, 6);
  insertLeft(root->left, 42);
  insertRight(root->left, 3);
  insertLeft(root->right, 2);
  insertRight(root->right, 33);
 return 0;
}
```

BINARY TREE TRAVERSAL

- Traversal is a frequently used operation.
- Traversal means visiting every node of a binary tree.
- This operation is used to visit each node (exactly once).
- There are various traversal methods.
- When traversing, we need to treat each node and its subtree in the same fashion.
- If we let L, D, and R stand for moving left, data, and moving right, respectively when at a node, then there are six possible combinations—**LDR, LRD, DLR, DRL, RDL, and RLD.**

Consider the binary tree shown in Fig.



Result of each of the six traversals is

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 – +

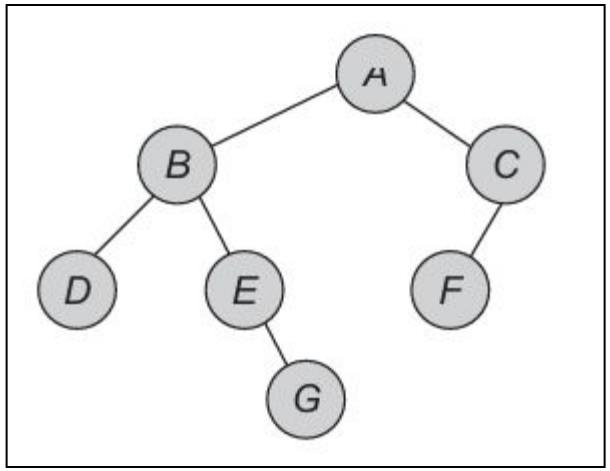
- DLR and RLD, LDR and RDL, and LRD and DRL are mirror symmetric. Three traversals, that is, LDR, LRD, and DLR, are fundamental.
- These are called as inorder, postorder, and Preorder traversals

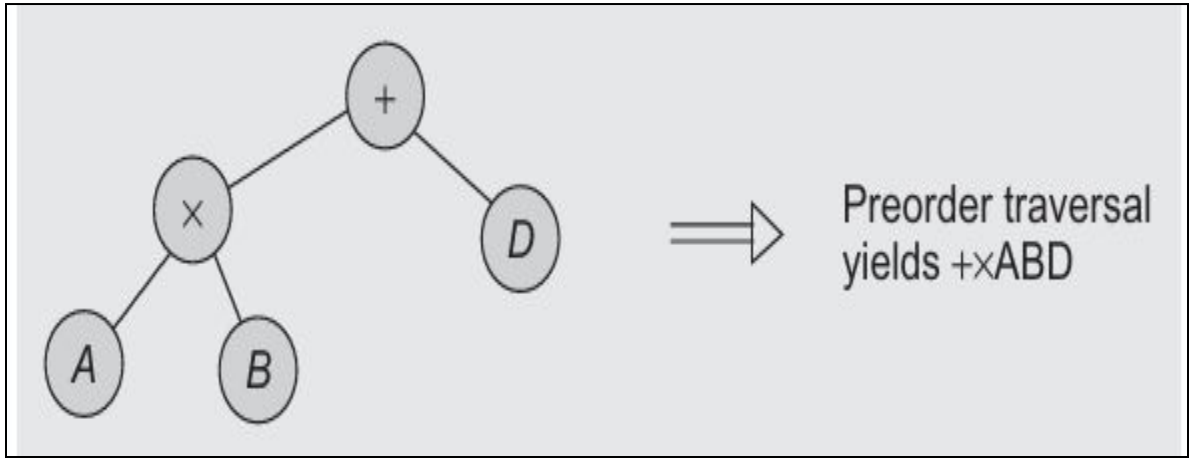
PREORDER TRAVERSAL (Root-L-R)

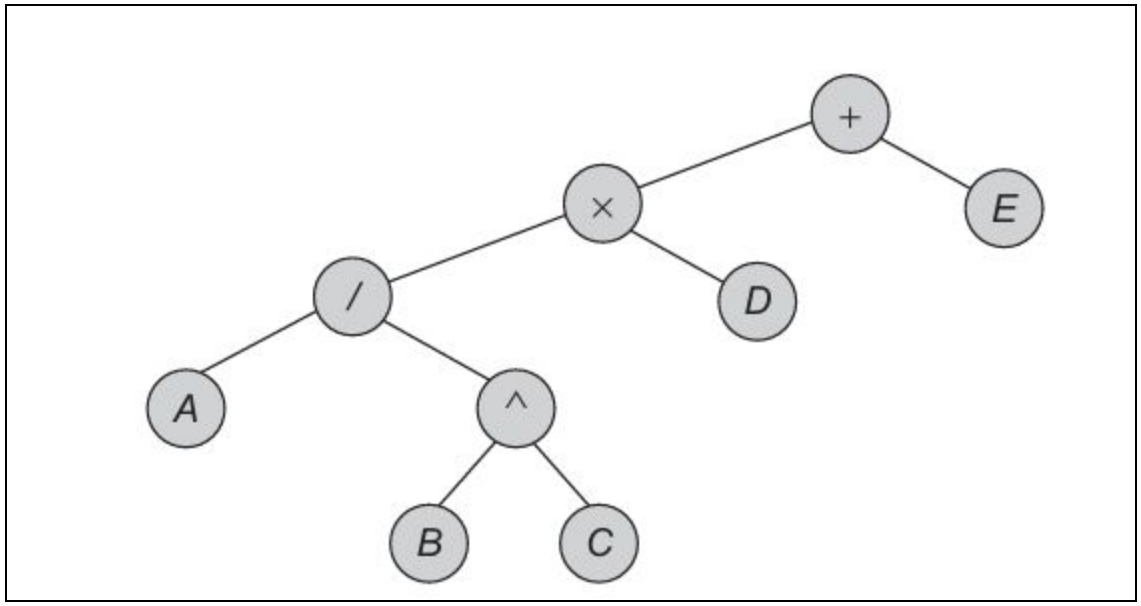
In this traversal, the root is visited first followed by the left subtree in preorder and then the right subtree in preorder. The tree characteristics lead to naturally implement the tree traversals recursively. It can be defined in the following steps:

Preorder (DLR) Algorithm

1. Visit the root node, say D.
2. Traverse the left subtree of the node in preorder.
3. Traverse the right subtree of the node in preorder.







- Preorder Traversal is **+x/A ^ BCDE**.
- The preorder traversal is also called as **depth-first traversal**.

Preorder traversal

```
void preorderTraversal(Node* root)
 {
    if (root == nullptr)
        return;
    std::cout << root->item << " ";
preorderTraversal(root->left);
preorderTraversal(root->right);
 }
```

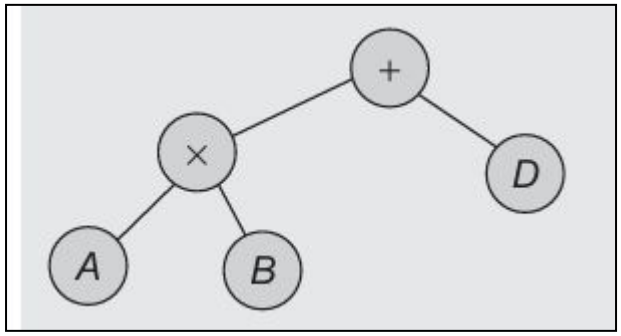
INORDER TRAVERSAL

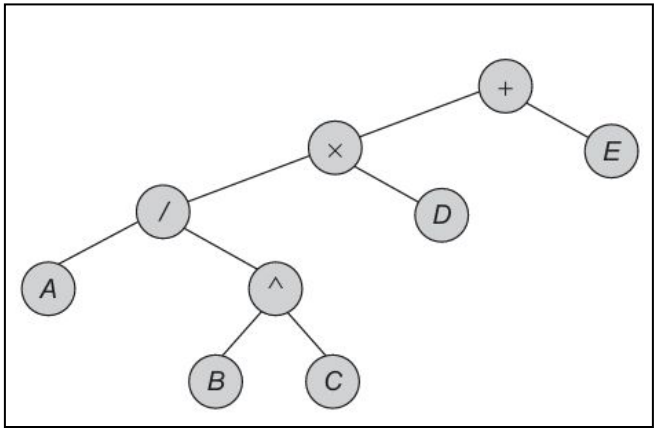
In this traversal, the left subtree is visited first in inorder followed by the root and then the right subtree in inorder. This can be defined as the following:

Inorder (LDR) Algorithm

1. *Traverse the left subtree of the root node in inorder.*
2. *Visit the root node.*
3. *Traverse the right subtree of the root node in inorder.*

Let us consider the binary tree in Fig.





An inorder traversal of a tree visits the node in the following sequence. which is an expression tree, yields an inorder expression as **A * B + D** for first Fig**.** and for second Fig. an inorder expression is **((A/B ^ C) * D) + E**.

Inorder traversal

```
void inorderTraversal(Node* root)

{

  if (root == nullptr)

    return;

  inorderTraversal(root->left);

  std::cout << root->item << " ";

  inorderTraversal(root->right);

}
```

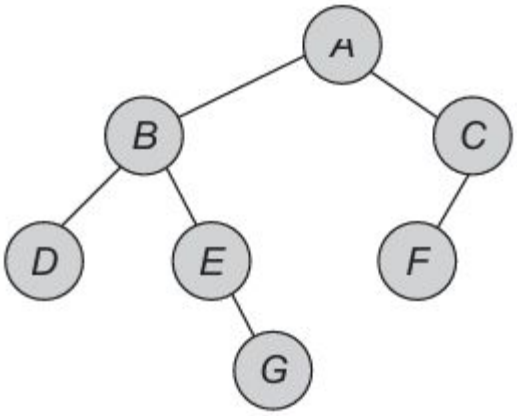
POSTORDER TRAVERSAL

In this traversal, the left subtree is visited first in postorder followed by the right subtree in postorder and then the root. This is defined as the following:

Postorder (LRD) Algorithm

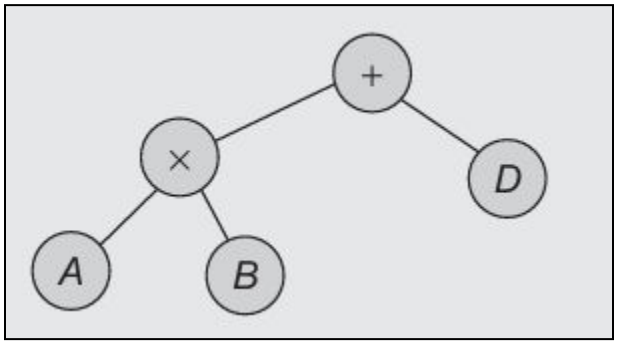
1. *Traverse the root's left child (subtree) of the root node in postorder.*
2. *Traverse the root's right child (subtree) of the root node in postorder.*
3. *Visit the root node.*

Let us consider the binary tree in Fig.

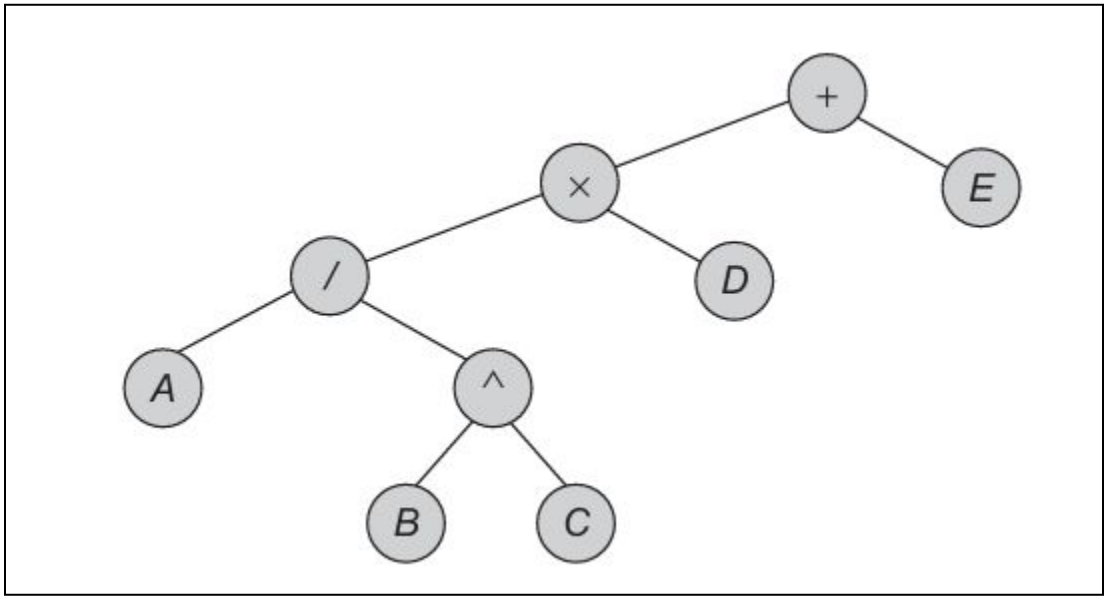


Inorder sequence: D B E G A F C

An inorder expression traversal of the tree in Figures.



= A B * D +

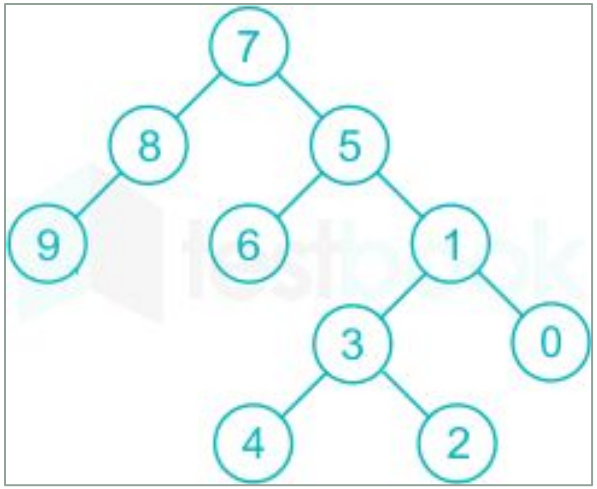


ABC**/ D*E+

Postorder traversal

```
void postorderTraversal(Node* root)

{

  if (root == nullptr)

        return;

  postorderTraversal(root->left);

  postorderTraversal(root->right);

  std::cout << root->item << " ";

}
```

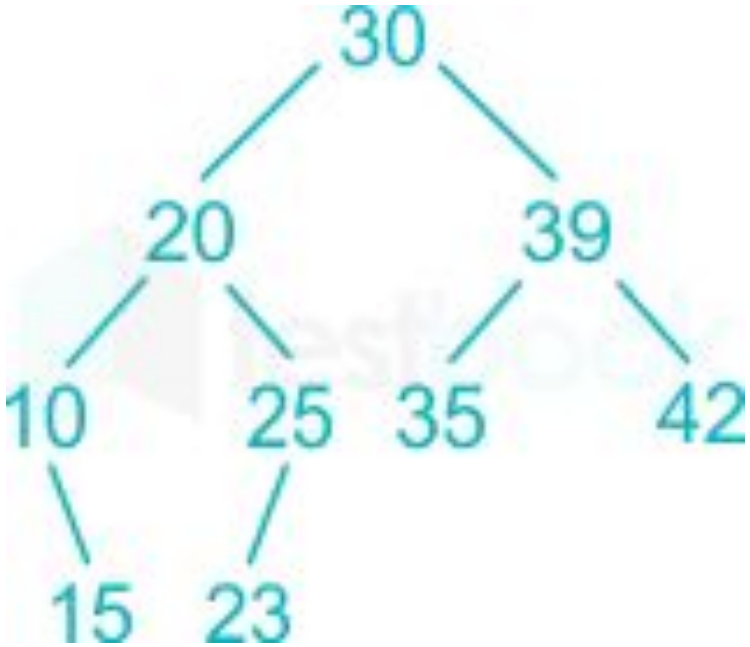
Suppose the numbers 7 , 5 , 1 , 8 , 3 , 6 , 0 , 9 , 4 , 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the inorder, Preorder and postorder traversal sequence of the resultant tree?



- The inorder traversal sequence is:
  - 9, 8, 7, 6, 5, 4, 3, 2, 1, 0
- The preorder traversal sequence is:
  - 7, 8, 9, 5, 6, 1, 3, 4, 2, 0
- The postorder traversal sequence is:
  - 9, 8, 6, 4, 2, 3, 0, 1, 5, 7

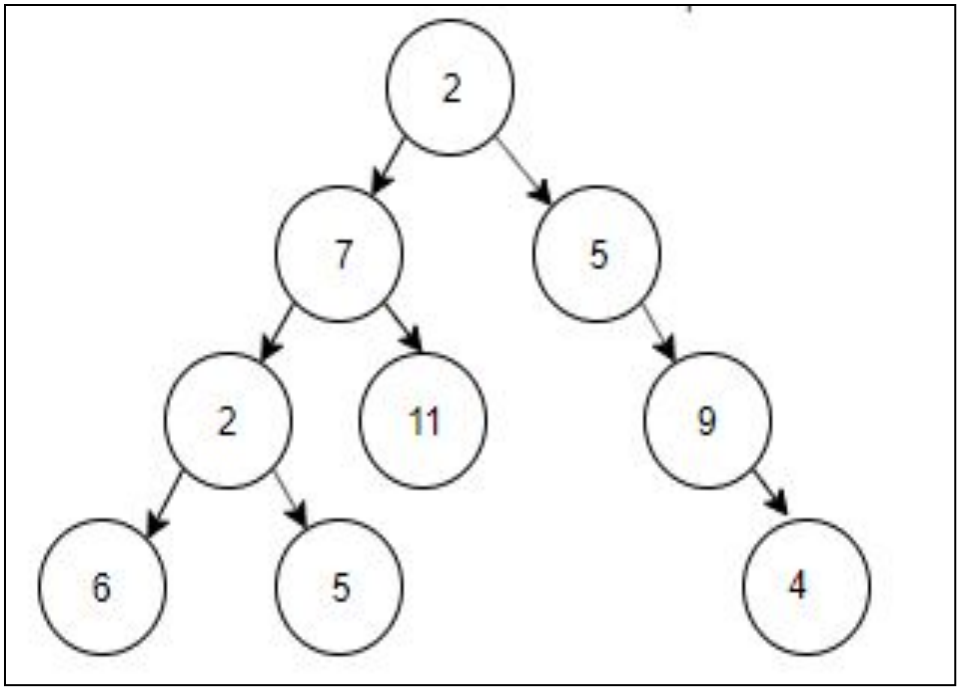
The preorder, Inorder and Postorder traversal sequence of a binary search tree is-

30 , 20 , 10 , 15 , 25 , 23 , 39 , 35 , 42



- Inorder traversal sequence:
  - 10, 15, 20, 23, 25, 30, 35, 39, 42
- Preorder traversal sequence:
  - 30, 20, 10, 15, 25, 23, 39, 35, 42
- Postorder traversal sequence:
  - 15, 10, 23, 25, 20, 35, 42, 39, 30

For the tree below, write the pre-order, post-order traversal.

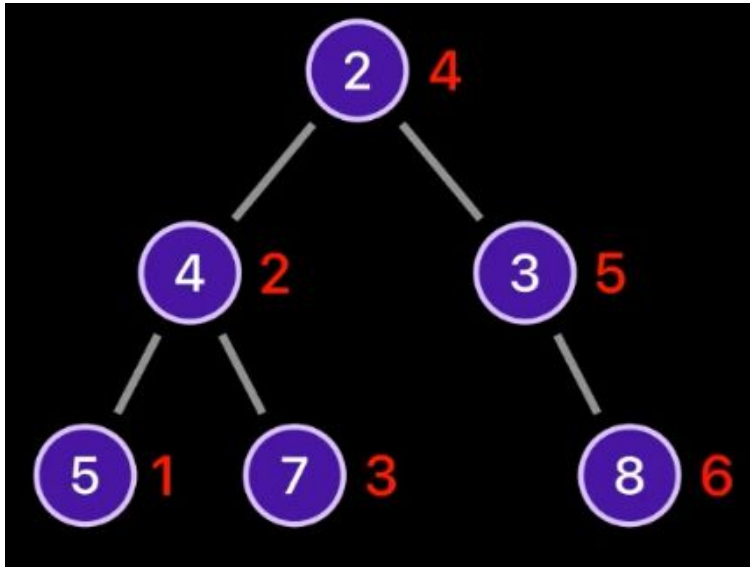


- pre-order - 2, 7, 2, 6, 5, 11, 5, 9, 4
- post-order - 6, 5, 2, 11, 7, 4, 9, 5, 2

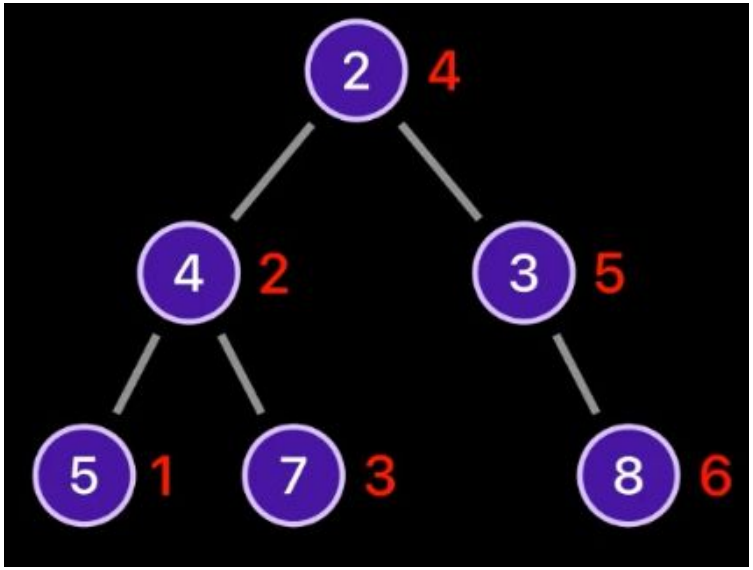
NON-RECURSIVE IMPLEMENTATION OF TRAVERSALS (Inorder Traversal)

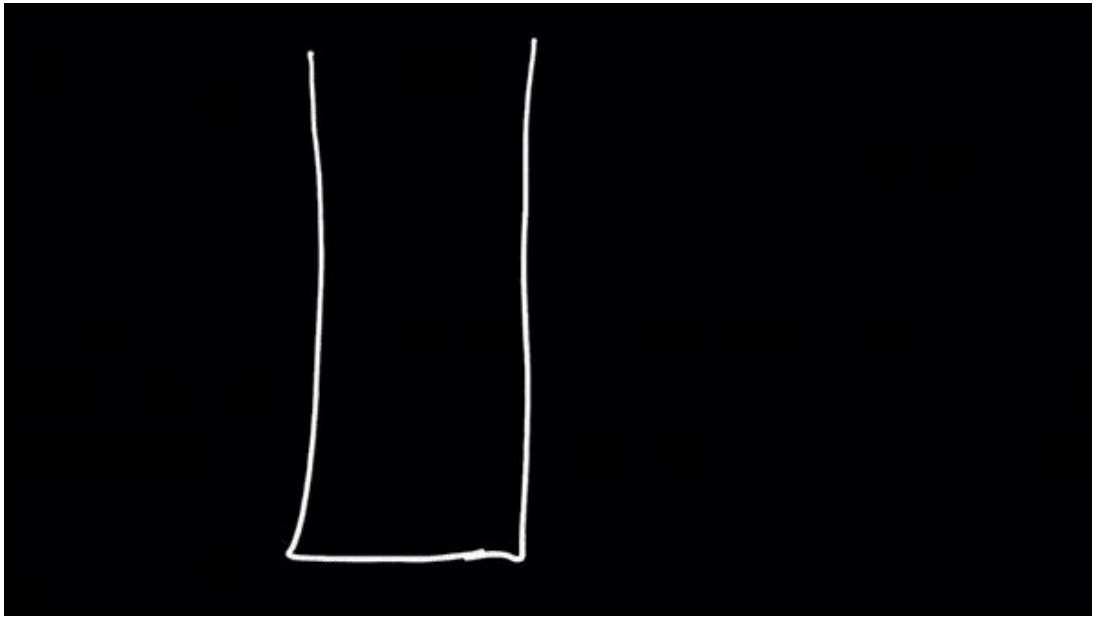
The inorder binary tree traversal algorithm can be described:

- Visit the left subtree
- Visit the root
- Visit the right-subtree



- During inorder traversal, nodes will be printed in the following order: 5, 4, 7, 2, 3, 8


2
4
4
2
3
5
5
1
7
3
8
6



Algorithm

To implement inorder traversal without recursive we will use Stack to store the traversal

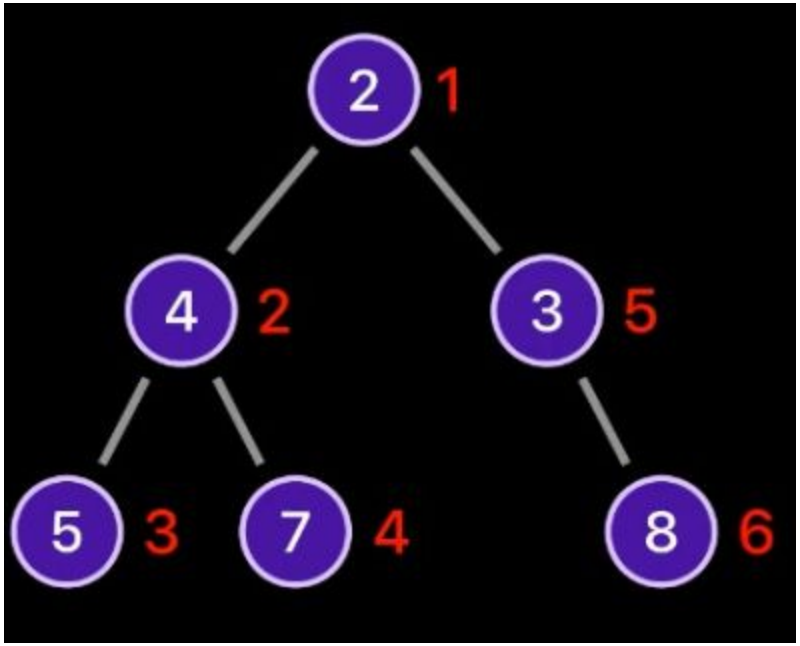
- **Step 1:** Create an empty stack
- **Step 2:** Set current node as root
- **Step 3:** Push the current node to the stack and set current = current.Left until the current is null
- **Step 4:** If the current is null and the stack is not empty then:
  - Pop a node from the stack
  - Print the poppedNode, set current = poppedNode.Right
  - Go to step 3
- **Step 5:** If the current is null and stack is empty then the traversal completed

```
void InorderTraversal(TreeNode* root)
 {
    if (root == nullptr) {
        return;
    }
    std::stack<TreeNode*> nodeStack;
    TreeNode* current = root;
    while (current != nullptr || !nodeStack.empty()) {
        while (current != nullptr) {
            nodeStack.push(current);
            current = current->left;
        }
        TreeNode* poppedNode = nodeStack.top();
        nodeStack.pop();
        // print value
        std::cout << poppedNode->value << std::endl;
        current = poppedNode->right;
    }
}
```

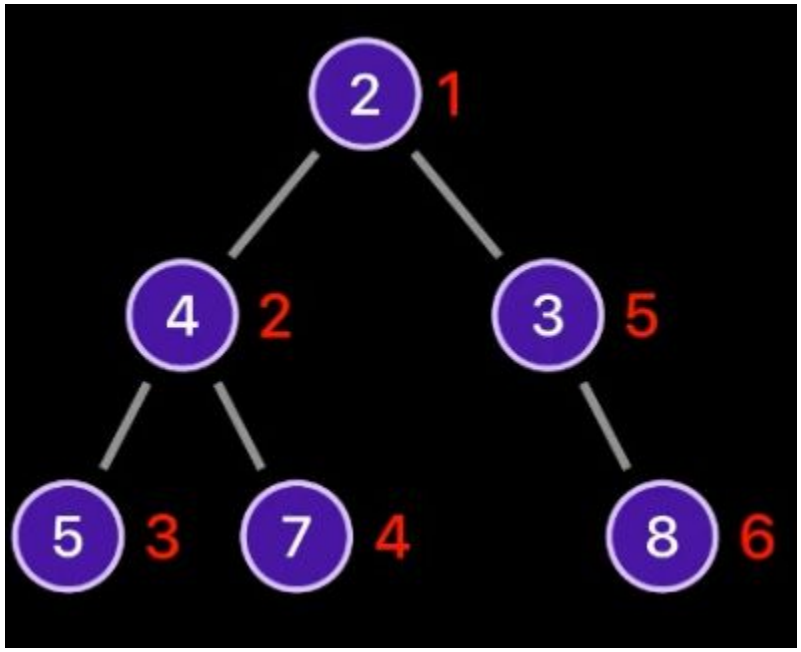
BINARY TREE PREORDER TRAVERSAL WITHOUT RECURSION

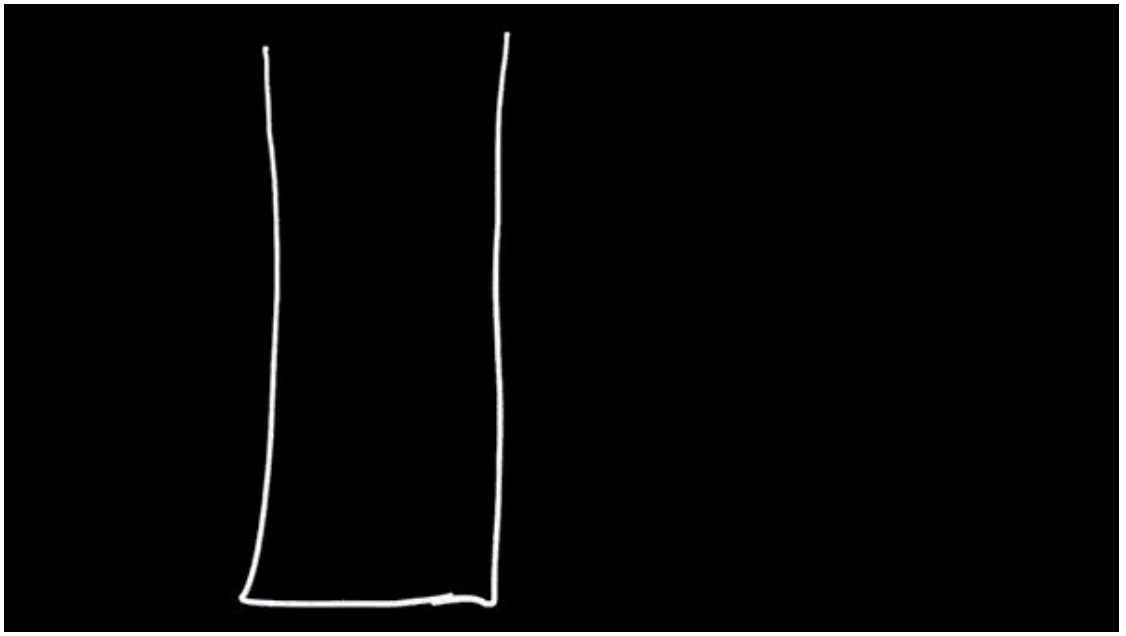
The preorder binary tree traversal algorithm can be described:

- Visit the root
- Visit the left subtree
- Visit the right subtree



During preorder traversal, nodes will be printed in the following order: 2, 4, 5, 7, 3, 8.


2 1
4 2
3 5
5 3
7 4
8 6



Algorithm

To implement preorder traversal without recursive we will use Stack to store the traversal

Step 1: Create an empty stack

Step 2: Set current node as root

Step 3: If the current is not null:

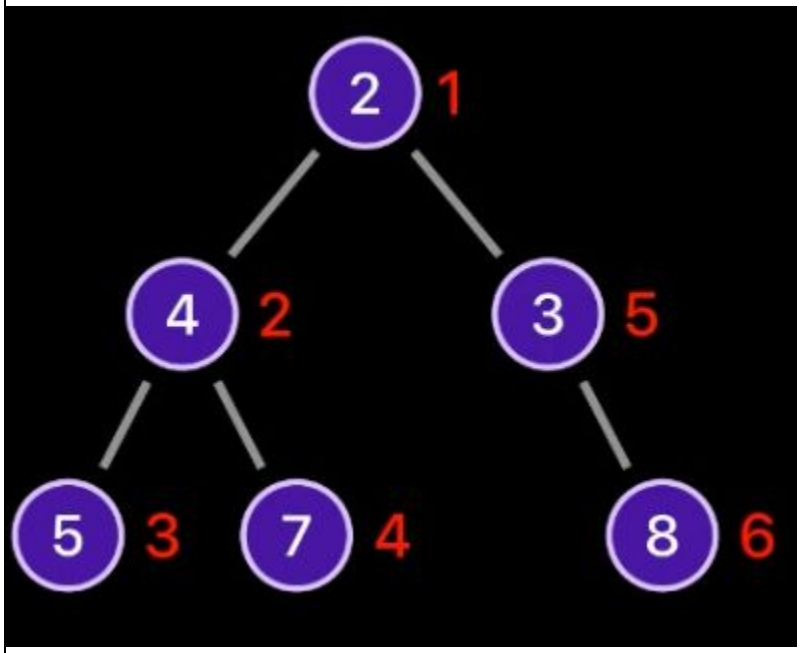
- Push the current node to the stack
- Print current
- Set current = current.Left

Step 4: If the current is null:

- Pop a node from the stack
- Set current = poppedNode.Right
- Go to step 3

Step 5: If the current is null and stack is empty then the traversal completed

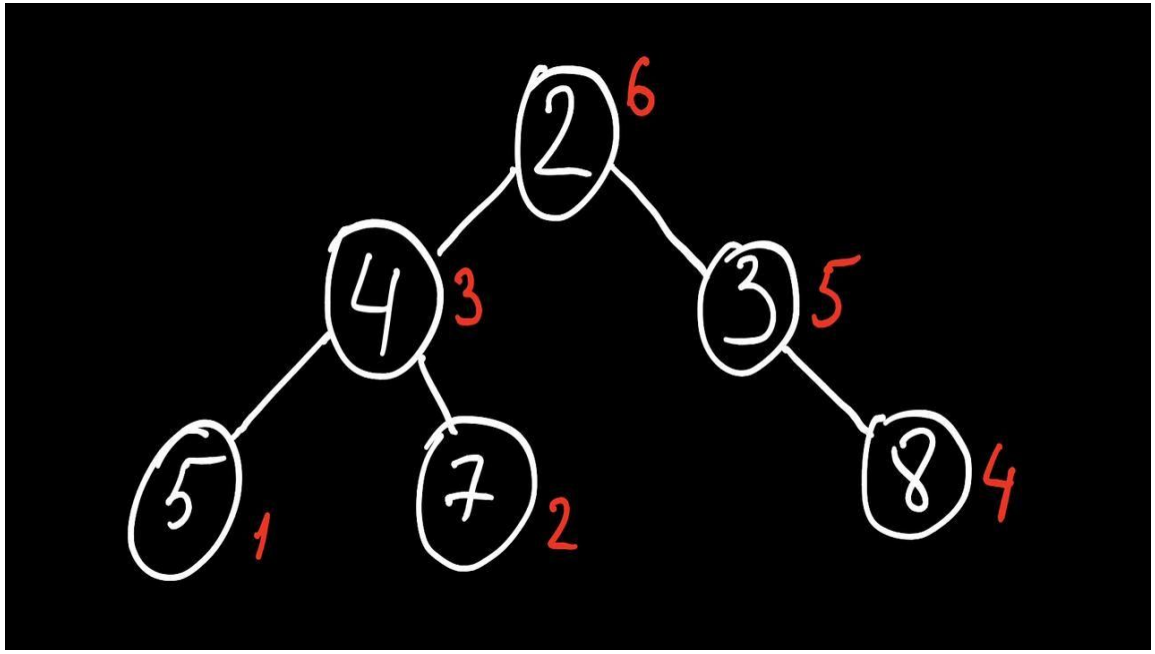
```
void PreorderTraversal(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    std::stack<TreeNode*> nodeStack;
    TreeNode* current = root;

    while (current != nullptr ||
!nodeStack.empty())
   {
        if (current != nullptr)
       {
            nodeStack.push(current);
            // print value
            std::cout << current->value <<
std::endl;
            current = current->left;
         }
         else
       {
            TreeNode* poppedNode =
nodeStack.top();
            nodeStack.pop();
            current = poppedNode->right;
        }
    }
}
```



Binary tree postorder traversal without recursion

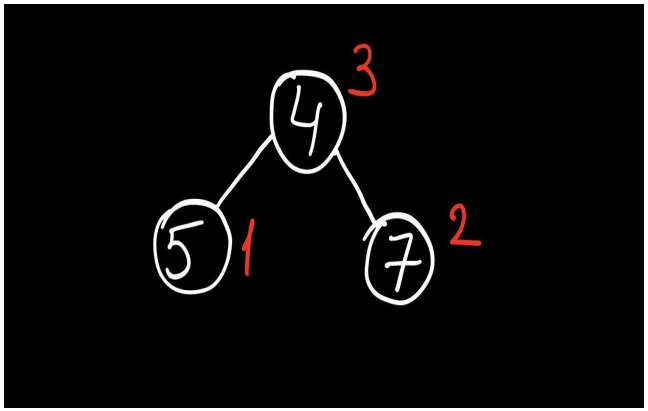
The postorder binary tree traversal algorithm can be described:

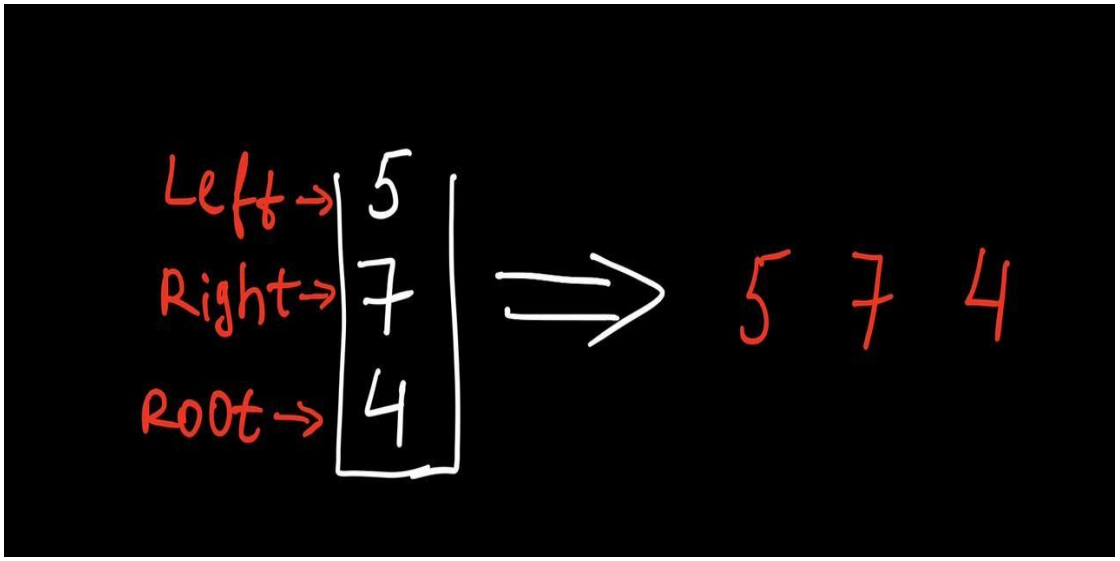
1. Visit the left subtree
2. Visit the right subtree
3. Visit the root



During postorder traversal, nodes will be printed in the following order: 5, 7, 4, 8, 3, 2.

Let's see on the simple example of how we can make the postorder traversal.





The nodes will be printed in the following order: 5, 7, 4. This traversal can be created for instance, if visit nodes in order:

- Visit the root
- Visit the right subtree
- Visit the left subtree

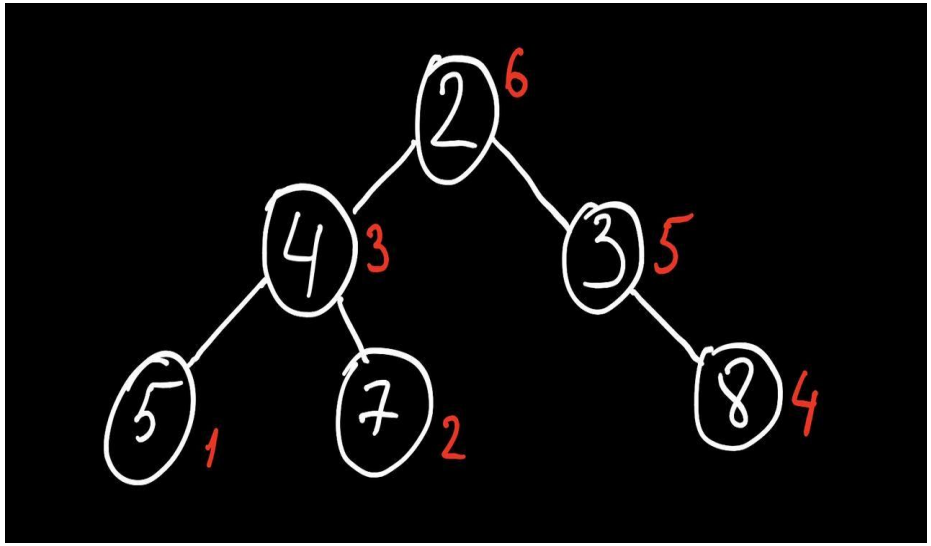
and put values into another result stack, like this:

Algorithm

1. **Create two stacks**
    1. result stack
    2. stack for the traversal
2. **While stack is not empty**
    1. pop a node from the stack and push into result stack
    2. push left child to the stack (if exists)
    3. push right child to the stack (if exists)
3. **Print traversal**

```
void PostorderTraversal(TreeNode* root)
{
    if (root == nullptr)
    {
        return;
    }

    std::stack<int> result;
    std::stack<TreeNode*> stack;

    stack.push(root);

    while (!stack.empty())
   {
        TreeNode* node = stack.top();

        stack.pop();

        result.push(node->value);

          if (node->left != nullptr)
          {

stack.push(node->left);
          }
```

```
        if (node->right != nullptr)
       {
            stack.push(node->right);
        }
}

while (!result.empty())
  {
    // print value
   std::cout << result.top()<<std::endl;
   result.pop();
  }
}
```

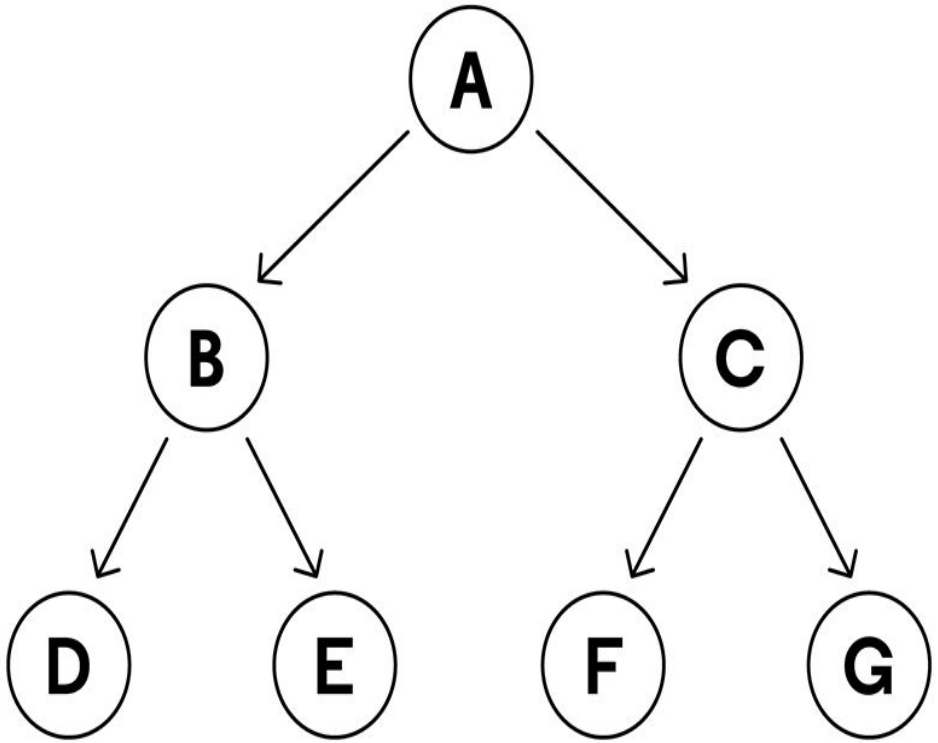


Breadth-first search

- A *breadth-first search* is when you inspect every node on a level starting at the top of the tree and then move to the next level.
- This algorithm also begins at the root node and then visits all nodes level by level.
- That means after the root, it traverses all the direct children of the root.
- After all direct children of the root are traversed, it moves to their children and so on.
- To implement BFS we use a queue.
- Storing the frontier nodes in a queue creates the level-by-level pattern of a breadth-first search.
- Child nodes are searched in the order they are added to the frontier.
- The nodes on the next level are always behind the nodes on the current level.
- Breadth-first search is known as a complete algorithm since no matter how deep the goal is in the tree it will always be located.

Tree with an Empty Queue

Frontier Queue
FIFO (First in First Out)

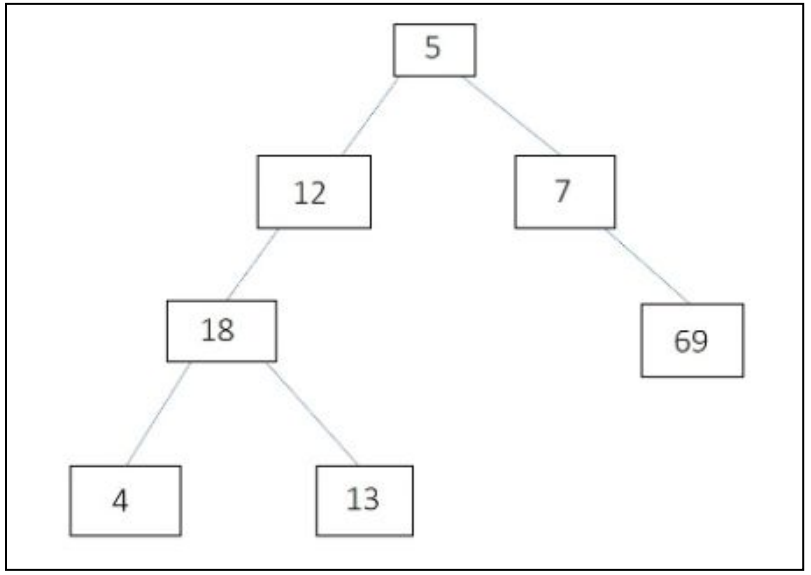


- Breadth-First Search is also known as Level Order Traversal.
- This algorithm starts with the root node and then visits all nodes level by level from left to right.
- Here we see every node on a level before going to a lower level.
- To implement BFS, we use a queue.
- The Breadth-First Search for trees can be implemented using level order traversal.

Algorithm

1. Start from root
2. Insert the root node into BFS and all of that node's neighbors into the queue.
3. If the node is not visited, pop it from the queue and add it to BFS, as well as all of its unvisited neighbors.
4. Repeat until the queue's size is not equal to zero(NULL).

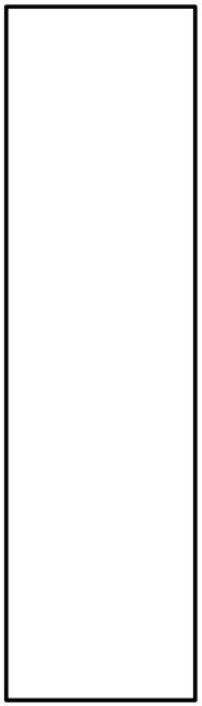
```
void BFSOrder(TreeNode* root) {
    if (root == nullptr) {
        return;
    }

    std::queue<TreeNode*> queue;
    queue.push(root);

    while (!queue.empty()) {
        TreeNode* temp = queue.front();
        queue.pop();
        std::cout << temp->data << " ";

        // Add left child to the queue
        if (temp->left != nullptr) {
            queue.push(temp->left);
        }

        // Add right child to the queue
        if (temp->right != nullptr) {
            queue.push(temp->right);
        }
    }
}
```



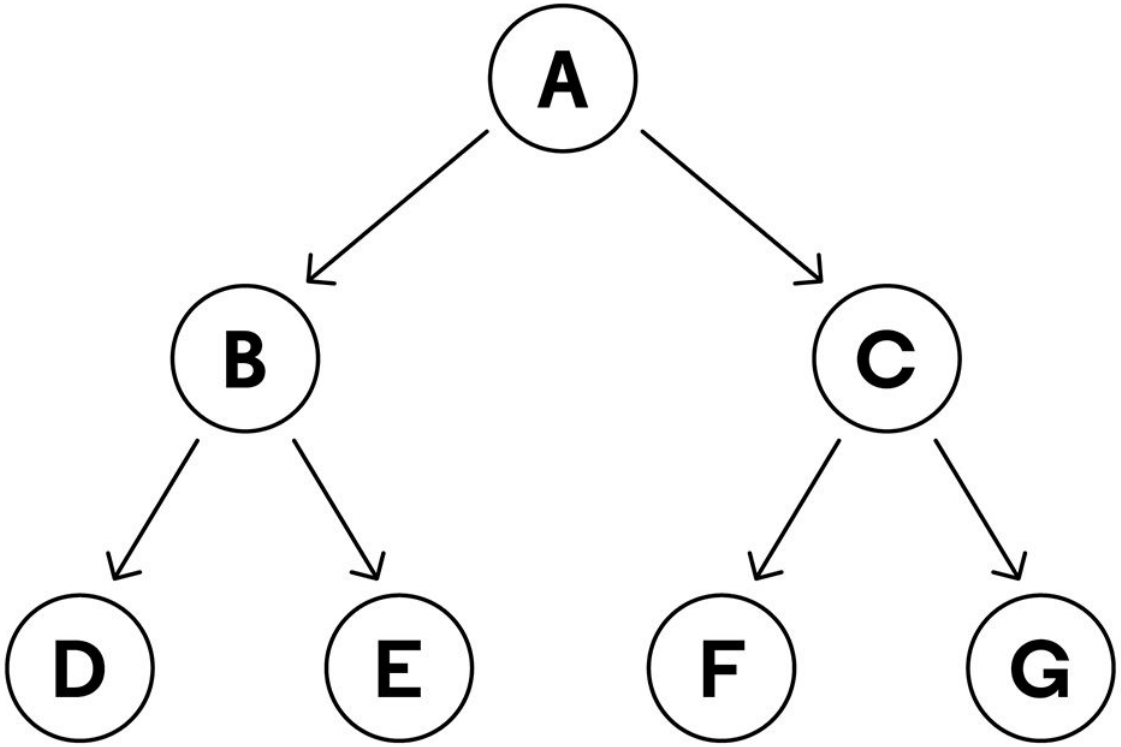
Depth-first search

- A *depth-first search* is where you search deep into a branch and don't move to the next one until you've reached the end.
- The algorithm begins at the root node and then it explores each branch before backtracking. It is implemented using stacks.
- Frontier nodes stored in a stack create the deep dive of a depth-first search.
- Nodes added to the frontier early on can expect to remain in the stack while their sibling's children (and their children, and so on) are searched.
- Depth-first search is not considered a complete algorithm since searching an infinite branch in a tree can go on forever.

Tree with an Empty Stack



Frontier Stack
LIFO (Last in First Out)



Algorithm :
Let us look at the algorithm to implement DFS.

In-Order Traversal

1. Inorder(root):
2. Move to the node's left side (traverse left-subtree).
3. Print the node's data.
4. Move to the node's right side (traverse right-subtree).

Pre-Order Traversal

5. Preorder(root):
6. Print the node's data.
7. Move to the node's left side (traverse left-subtree).
8. Move to the node's right side (traverse right-subtree).

Post-Order Traversal

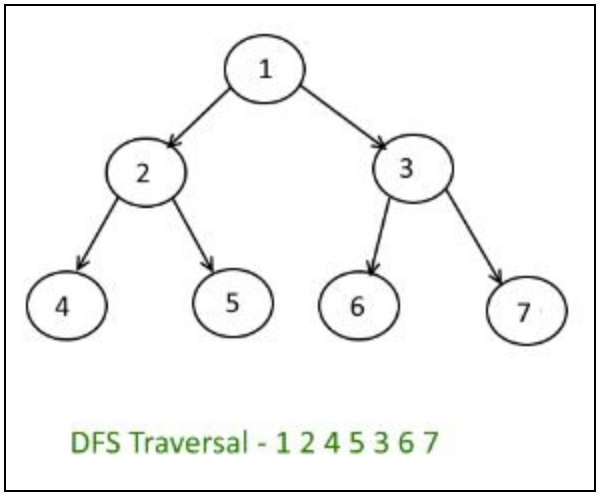
9. Postorder(root):
10. Move to the node's left side (traverse left-subtree).
11. Move to the node's right side (traverse right-subtree).
12. Print the node's data.

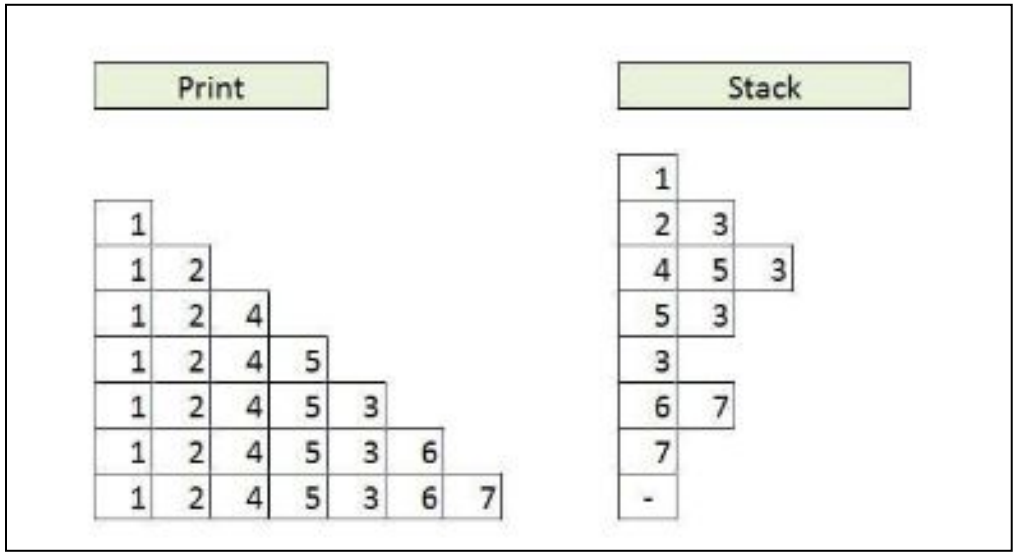
Approach:

Approach is quite simple, use Stack.

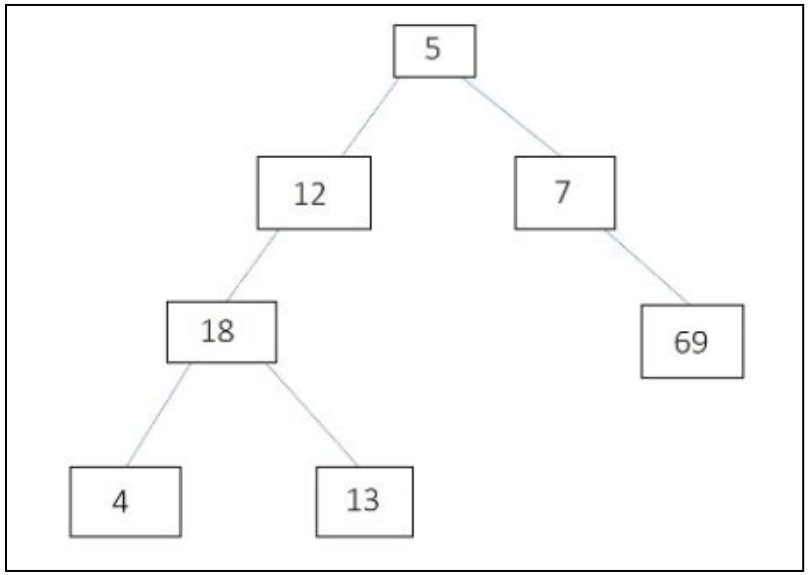
1. First add the add root to the Stack.
2. Pop out an element from Stack and add its right and left children to stack.
3. Pop out an element and print it and add its children.
4. Repeat the above two steps until the Stack id empty.

Example:





```
void DFS_Tree(TreeNode* Root) {
        if (Root == nullptr) {
            return;
        }

        std::stack<TreeNode*> S;
        TreeNode* Tmp = Root;

        do {
            std::cout << Tmp->Data << " ";

            if (Tmp->Rchild != nullptr)
                S.push(Tmp->Rchild);
            if (Tmp->Lchild != nullptr)
                S.push(Tmp->Lchild);

            if (S.empty())
                break;

            Tmp = S.top();
            S.pop();

        } while (true);
```



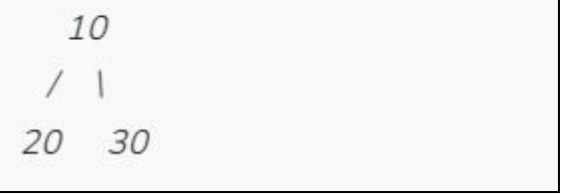
Operations on binary tree

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

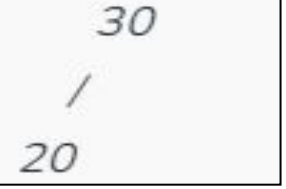
1. **Creation**—Creating an empty binary tree to which the 'root' points
2. **Traversal—**Visiting all the nodes in a binary tree
3. **Deletion—**Deleting a node from a non-empty binary tree
4. **Insertion—**Inserting a node into an existing (may be empty) binary tree
5. **Merge—**Merging two binary trees
6. **Copy—**Copying a binary tree
7. **Compare—**Comparing two binary trees
8. **Finding a replica** or mirror of a binary tree

- **Deletion—**Deleting a node from a non-empty binary tree

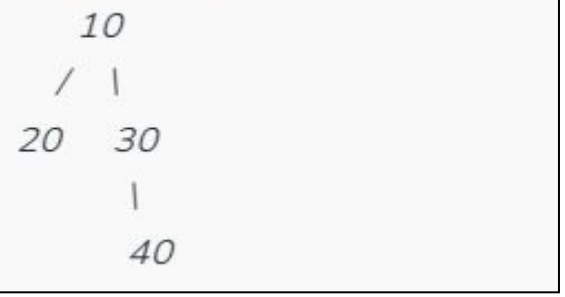
Input : *Delete 10 in below tree*



Output:



Input : *Delete 20 in below tree*



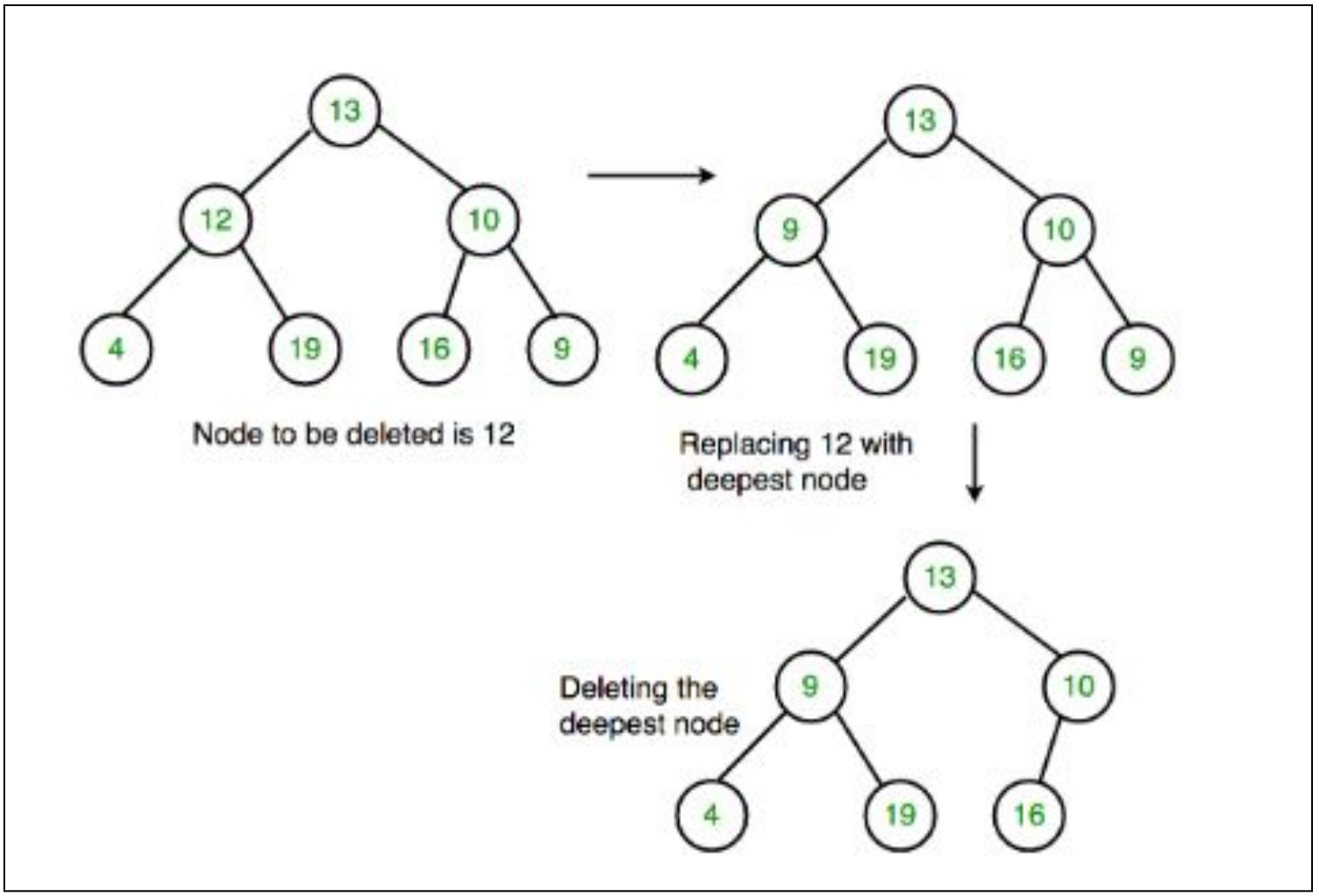
Output:

Algorithm:

1. Starting at the root, find the deepest and rightmost node in the binary tree and the node which we want to delete.
2. Replace the deepest rightmost node's data with the node to be deleted.
3. Then delete the deepest rightmost node.

Algorithm:

1. Initialize the tree with binary node struct.
2. Write a function (preorder, in order, and postorder) to print the nodes of the tree.
3. Write a function to delete the node.
4. Initialize a queue to iterate through the tree.
5. Iterate until the queue is empty.
6. Find the node with the given key and store it in a variable.
7. And the last node from the queue is the deepest node.
8. Delete the deepest node using another function.
9. Use the queue to traverse through the tree.
10. When we find the node delete it and return it.
11. Print the tree to see if the node is deleted or not.


13
12
10
4
19
16
9
Node to be deleted is 12
13
9
10
4
19
16
9
Replacing 12 with
deepest node
13
Deleting the
deepest node
9
10
4
19
16

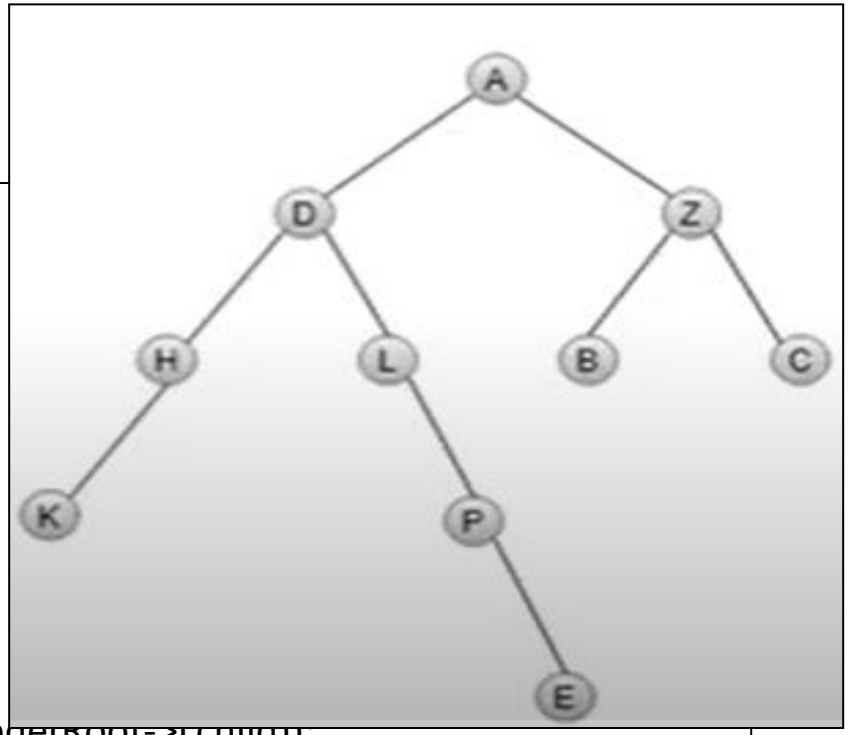
```
Node* deleteNode(struct Node* root, int key) {
   if (root == NULL){
      return NULL;
   }
   if (root->left == NULL && root->right == NULL) {
      if (root->data == key) {
         return NULL;
      }
      else {
         return root;
      }
   }
   queue<struct Node*> nodes;
   nodes.push(root);
   struct Node* temp;
   struct Node* key_node = NULL;
   while (!nodes.empty()) {
      temp = nodes.front();
      nodes.pop();
      if (temp->data == key) {
         key_node = temp;
      }
      if (temp->left) {
         nodes.push(temp->left);
      }
      if (temp->right) {
         nodes.push(temp->right);
      }
   }
```

```
 if (key_node != NULL) {
      int deepest_node_data = temp->data;
      deleteDeepestNode(root, temp);
      key_node->data = deepest_node_data;
   }
   return root;
}
```

```
void deleteDeepestNode(struct Node* root, struct Node* deleting_node){
   queue<struct Node*> nodes;
   nodes.push(root);
   struct Node* temp;
   while (!nodes.empty()) {
      temp = nodes.front();
      nodes.pop();
      if (temp == deleting_node) {
         temp = NULL;
         delete (deleting_node);
         return;
      }
      if (temp->right) {
         if (temp->right == deleting_node) {
            temp->right = NULL;
            delete deleting_node;
            return;
         }
         else {
            nodes.push(temp->right);
         }
      }
      if (temp->left) {
         if (temp->left == deleting_node) {
            temp->left = NULL;
            delete deleting_node;
            return;
         }
         else {
            nodes.push(temp->left);
         }
      }
   }
}
```

COUNTING NODES

CountNode() is the function that returns the total count of nodes in a linked binary tree.

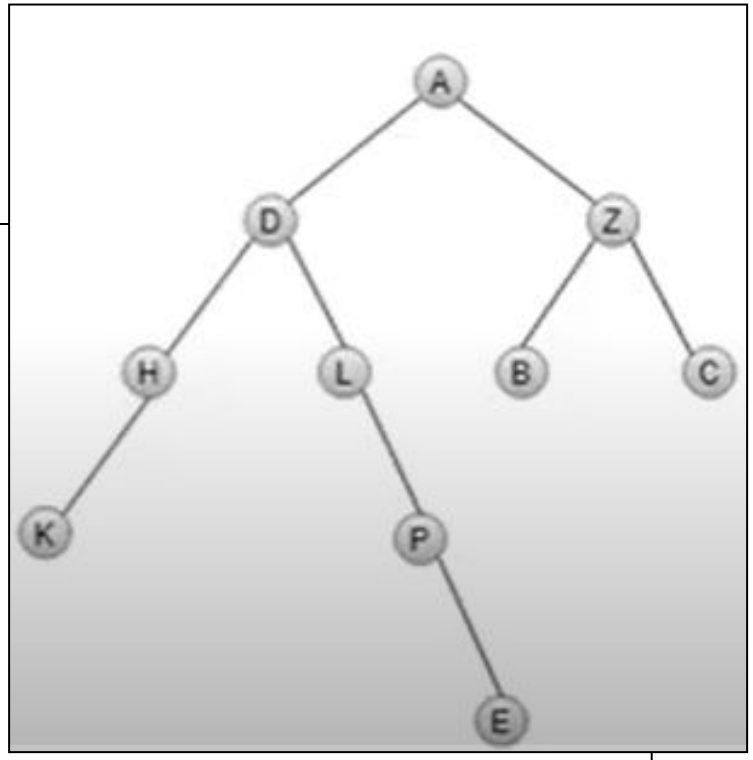



```
int  CountNode(TreeNode *Root)

{

  if(Root == Null)

     return 0;

  else

     return(1 + CountNode(Root->Rchild) + CountNode(Root->Lchild));

}
```

COUNTING LEAF NODES

The CountLeaf() operation counts the total number of leaf nodes in a linked binary tree.

Leaf nodes are those with no left or right children.

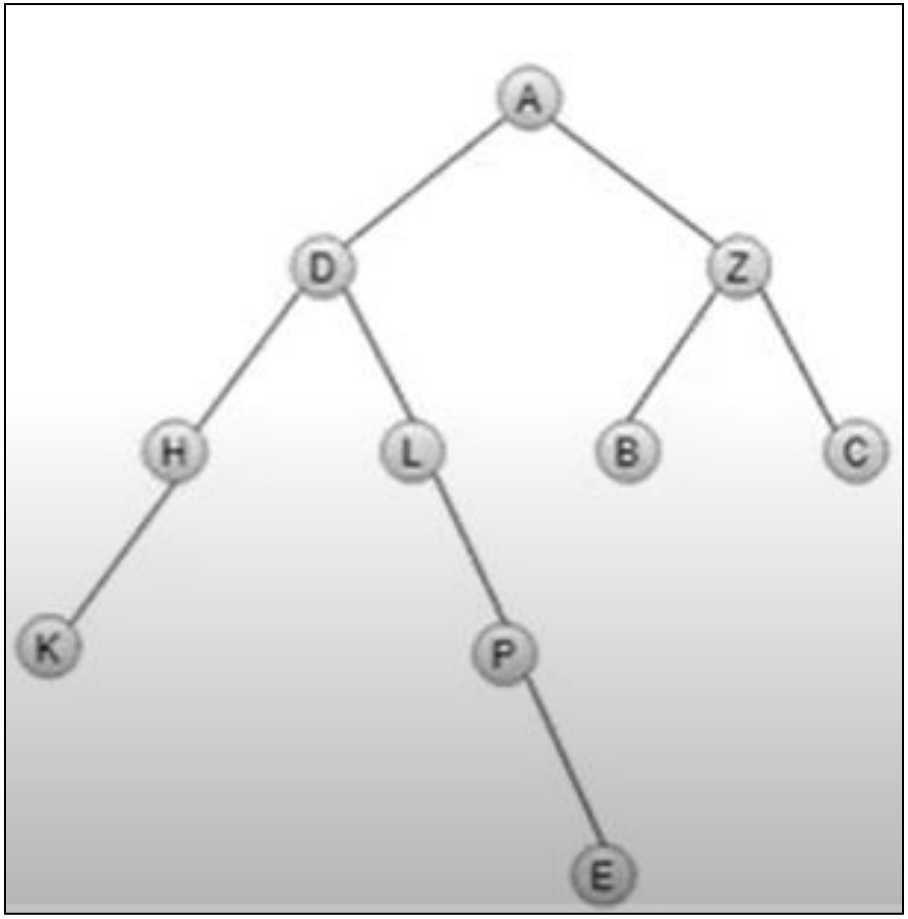



```
int CountLeaf(TreeNode *Root)
{
  if(Root == Null)
     return 0;
  else if ( (Root->Rchild == Null ) && ( Root->Lchild == Null ) )
     return(1);
  else
     return( CountLeaf( Root->Lchild ) + CountLeaf( Root->Rchild ) );
}
```

COMPUTING HEIGHT OF BINARY TREE

- The TreeHeight() operation computes the height of a linked binary tree.
- Height of a tree is the maximum path length in the tree.
- We can get the path length by traversing the tree depthwise.
- Let us consider that an empty tree's height is 0 and the tree with only one node has the height 1.

```
int height_recursive(TreeNode *node)
{
   if (node == NULL)
      return -1;
   int lheight = height_recursive(node->left);
                        int      rheight      =
height_recursive(node->right);
   return max(lheight, rheight) + 1;
}
```



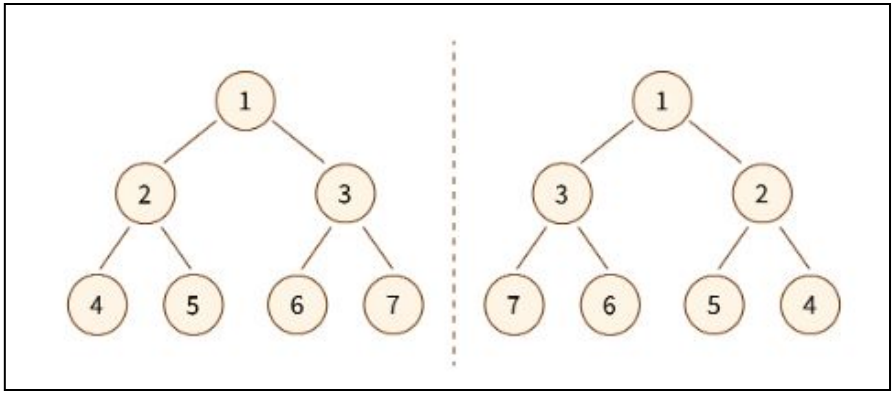
Getting Mirror, Replica, or Tree interchange of Binary Tree

The mirror() operation finds the mirror of the tree that will interchange all left and right subtrees in a linked binary tree.

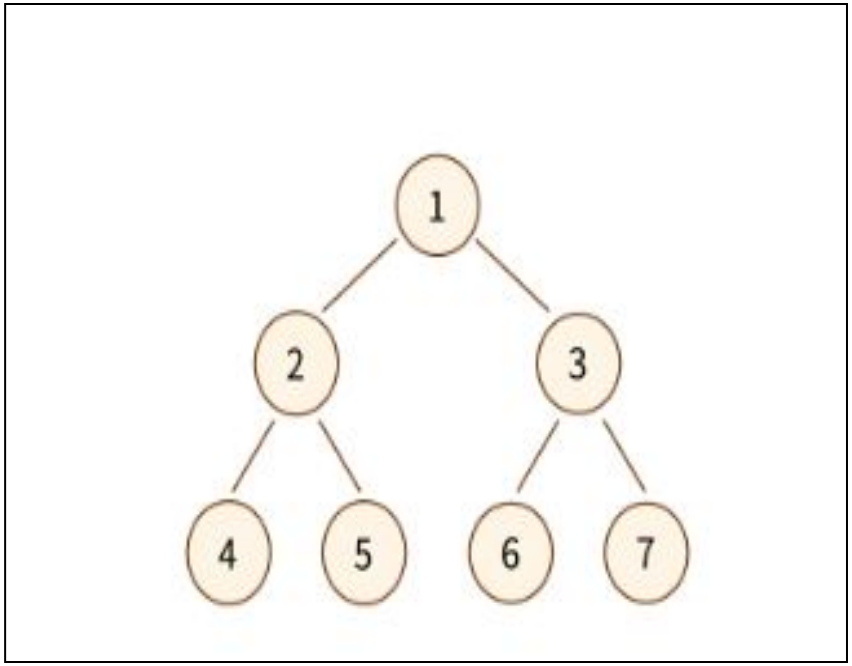
Algorithm

The algorithm for this approach is as follows:

1. Call the mirror function for the left sub-tree, i.e. Mirror(left sub-tree)
2. Call the mirror function for the right sub-tree, i.e. Mirror(right sub-tree)
3. Swap left and right sub-trees using,
    1. temp = left sub-tree
    2. left sub-tree = right sub-tree
    3. right sub-tree = temp



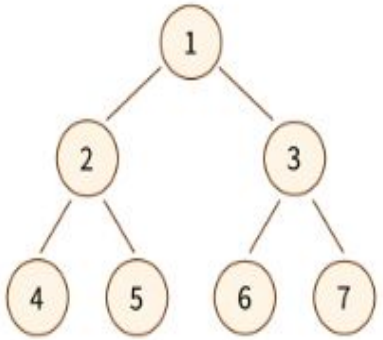
```
void mirror(TreeNode* node) {
    if (node == NULL)
        return ;
    else {
        struct Node* temp;
        // mirror the subtrees
        mirror(node->left);
        mirror(node->right);
        // swapping the pointers for this node
        temp = node->left;
        node->left = node->right;
        node->right = temp;
    }
}
```



Copying binary tree

The `cloneBinaryTree` () operation makes a copy of the linked binary tree. The function should allocate the necessary nodes and copy the respective contents into them.

```
Node* cloneBinaryTree(Node* root)
{
    // base case
    if (root == nullptr) {
        return nullptr;
    }

    // create a new node with the same data as the
root node
    Node* root_copy = new Node(root->data);

    // clone the left and right subtree
    root_copy->left = cloneBinaryTree(root->left);
    root_copy->right = cloneBinaryTree(root->right);

    // return cloned root node
    return root_copy;
}
```



Equality test

```
int isIdentical(Node* root1, Node* root2)
{
    // Check if both the trees are empty
    if (root1 == NULL && root2 == NULL)
        return 1;
    // If any one of the tree is non-empty and other is empty, return false
    else if (root1 == NULL || root2 == NULL)
        return 0;
    else {
     // Check if current data of both trees equal
        // and recursively check for left and right subtrees
        if (root1->data == root2->data && isIdentical(root1->left, root2->left)
            && isIdentical(root1->right, root2->right))
            return 1;
        else
            return 0;
    }
}
```

Huffman's Coding

- One of the most important applications of the binary tree is in communication.
- Huffman Coding is a technique that is used for compressing data to reduce its size without losing any of its details.
- Huffman Coding is a famous Greedy algorithm.
- It uses variable length encoding.
- It assigns variable length code to all the characters.
- The code length of a character depends on how frequently it occurs in the given text.
- The character which occurs most frequently gets the smallest code.
- The character which occurs least frequently gets the largest code.
- It is also known as Huffman Encoding.

Huffman's Coding - HOW IT WORKS

Let assume the string data given below is the data we want to compress -

- The length of the above string is 15 characters and each character occupies a space of 8 bits.
- Therefore, a total of 120 bits (8 bits x 15 characters) is required to send this string over a network.
- We can reduce the size of the string to a smaller extent using Huffman Coding Algorithm.

1. In this algorithm first we create a tree using the frequencies of characters and then assign a code to each character.
2. The same resulting tree is used for decoding once encoding is done.
3. Using the concept of prefix code, this algorithm avoids any ambiguity within the decoding process, i.e. a code assigned to any character shouldn't be present within the prefix of the opposite code.

STEPS TO HUFFMAN CODING

1. First, we calculate the count of occurrences of each character in the string.

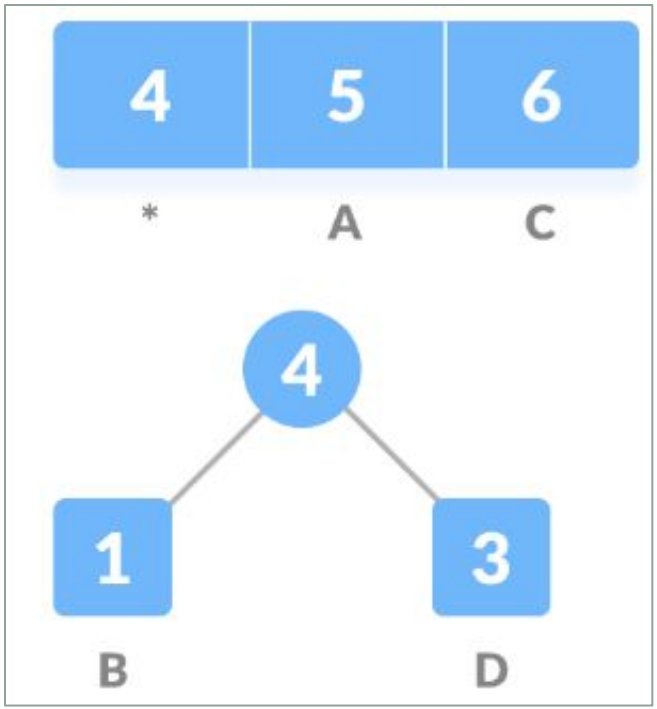
2. Then we sort the characters in the above string in increasing order of the count of occurrence. Here we use PriorityQueue to store.

3. Now we mark every unique character as a Leaf Node.

4. Let's create an empty node n. Add characters having the lowest count of occurrence as the left child of n and second minimum count of occurrence as the right child of n, then assign the sum of the above two minimum frequencies to n.

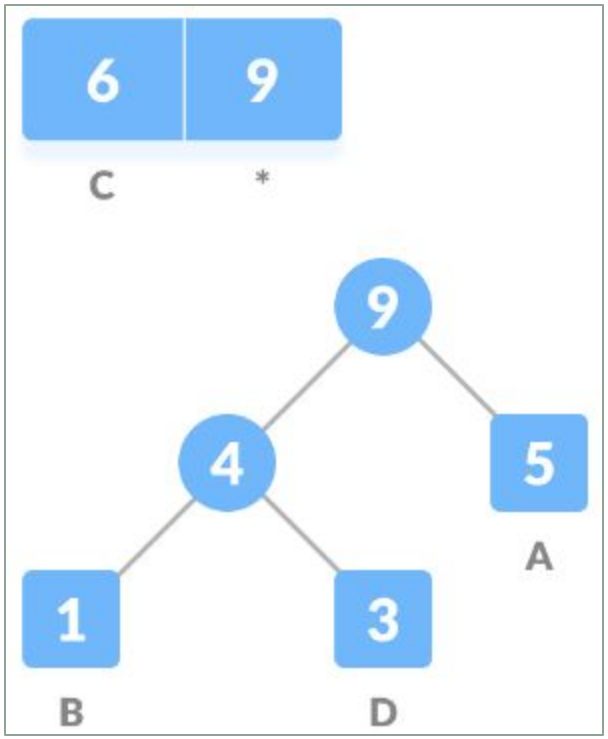
5. Now remove these two minimum frequencies from Queue and append the sum into the list of frequencies.

6. Add node n into the tree.

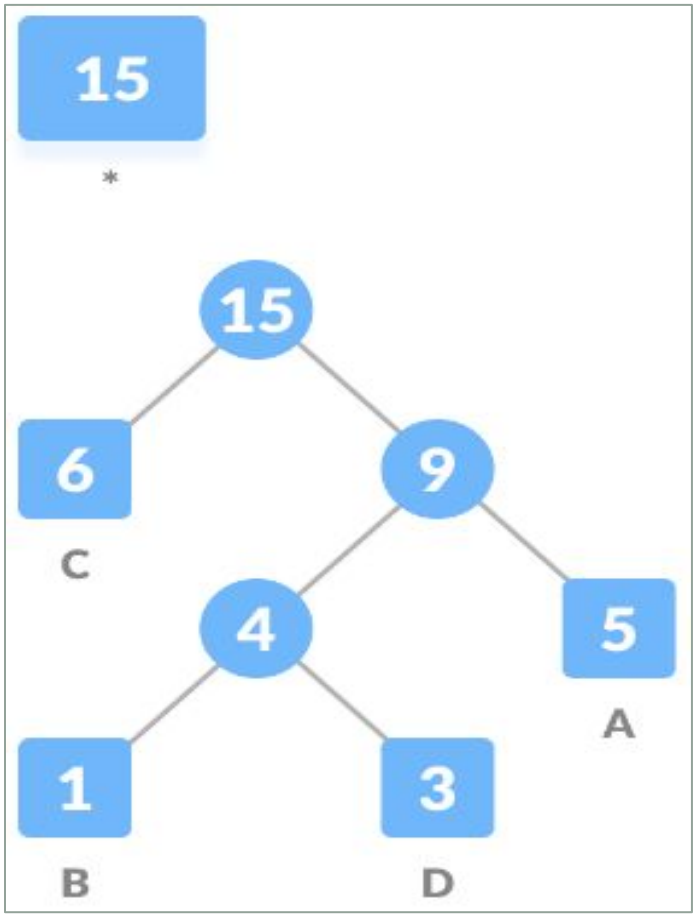


7. Just like we did for B and D, we repeat the same steps from 3 to 5 for the rest of the characters (A and C).

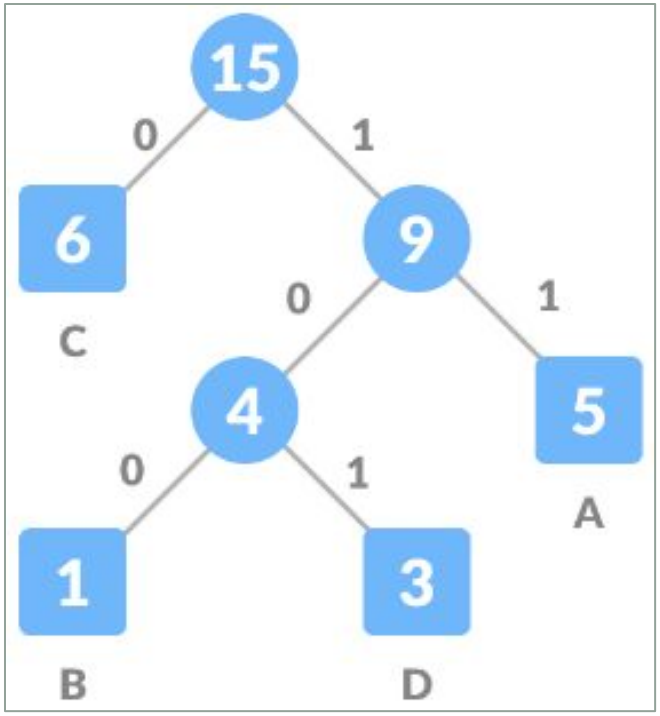
For A -



Repeat for C -

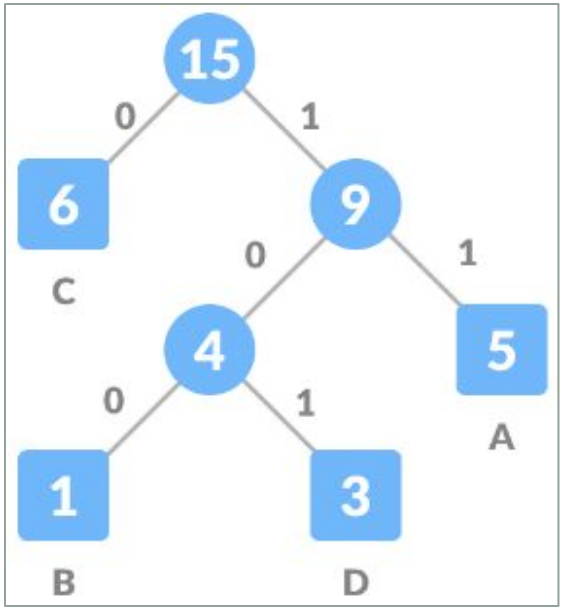


8. We got our resulting tree, now we assign 0 to the left edge and 1 to the right edge of every non-leaf node.



9. Now for generating codes of each character we traverse towards each leaf node representing some character from the root node and form code of it.

All the data we gathered until now is given below in tabular form -

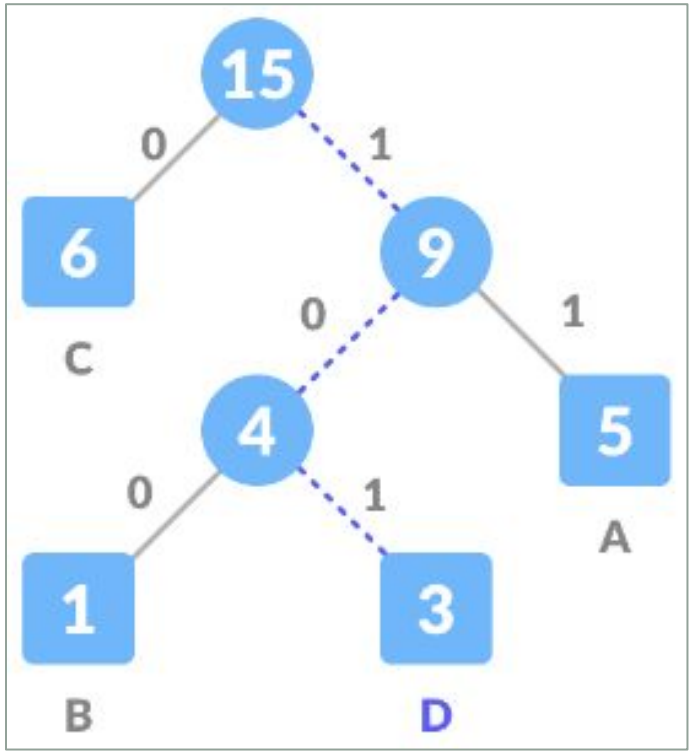


- Before compressing the total size of the string was 120 bits. After compression that size was reduced to 75 bits (28 bits + 15 bits + 32 bits).

Character	Frequency in string	Assigned Code	Size
B	1	**100**	1 x 3 = 3 bits
D	3	**101**	3 x 3 = 9 bits
A	5	**11**	5 x 2 = 10 bits
C	6	**0**	6 x 1 = 6 bits
4 x 8 = 32 bits	Total = 15 bits		Total = 28 bits

STEPS TO HUFFMAN DECODING

- To decode any code, we take the code and traverse it in the tree from the root node to the leaf node, each code will make us reach a unique character.
- Let assume code 101 needs to be decoded, for this we will traverse from the root as given below -



- As per the Huffman encoding algorithm, for every 1 we traverse towards the right child and for 0 we traverse towards the left one, if we follow this and traverse, we will reach leaf node 3 which represents D. Therefore, 101 is decoded to D.

ALGORITHM

1. create and initialize a PriorityQueue Queue consisting of each unique character.
2. sort in ascending order of their frequencies.
3. for all the unique characters :
    1. create a new_node
    2. get minimum_value from Queue and set it to left child of new_node
    3. get minimum_value from Queue and set it to right child of new_node
    4. calculate the sum of these two minimum values as sum_of_two_minimum
    5. assign sum_of_two_minimum to the value of new_node
    6. insert new_node into the tree
4. return root_node

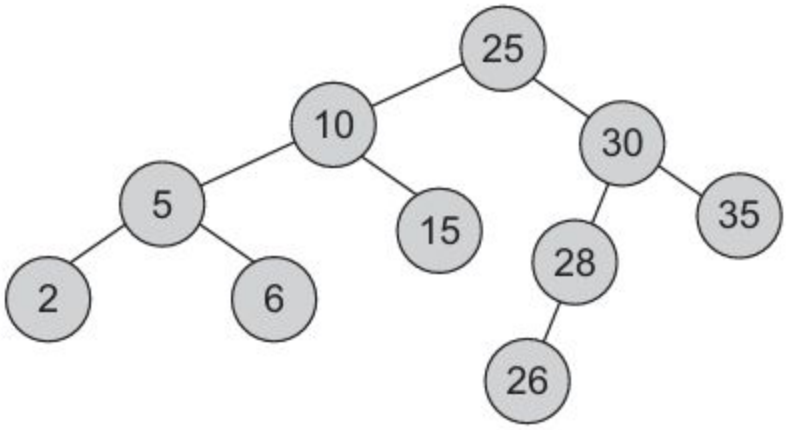
Binary Search Tree (BST)

- The binary search tree (BST) 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.
- The BSTs are classified as static trees and dynamic trees.
- **Static tree** is a BST where the set of values in the nodes is known in advance and never changes.
- **Dynamic tree** is a BST where the values in a tree may change over time.
- Complexity of binary search tree algorithm

Binary Search Tree (BST)

A BST is a binary tree that is either empty or where every node contains a key and satisfies the following conditions:

1. The key in the left child of a node, if it exists, is less than the key in its parent node.
2. The key in the right child of a node, if it exists, is greater than the key in its parent node.
3. The left and right subtrees of a node are again BSTs.



The following are the operations commonly performed on a BST:

1. Inserting a key
2. Searching a key
3. Deleting a key
4. Traversing the tree

Insertion

- A new key is always inserted at the leaf.
- We start searching a key from the root until we hit a leaf node.
- Once a leaf node is found, the new node is added as a child of the leaf node.

Inserting a node in a binary search tree - Algorithm

1. Create a new BST node and assign values to it.

2. insert(node, key)

i) If root == NULL,

return the new node to the calling function.

ii) if root data < key

call the insert function with root right and assign the return value in root right.

root right = insert(root right, key)

iii) if root data > key

call the insert function with root left and assign the return value in root left.

root left = insert(root left , key)

3. Finally, return the original root pointer to the calling function.

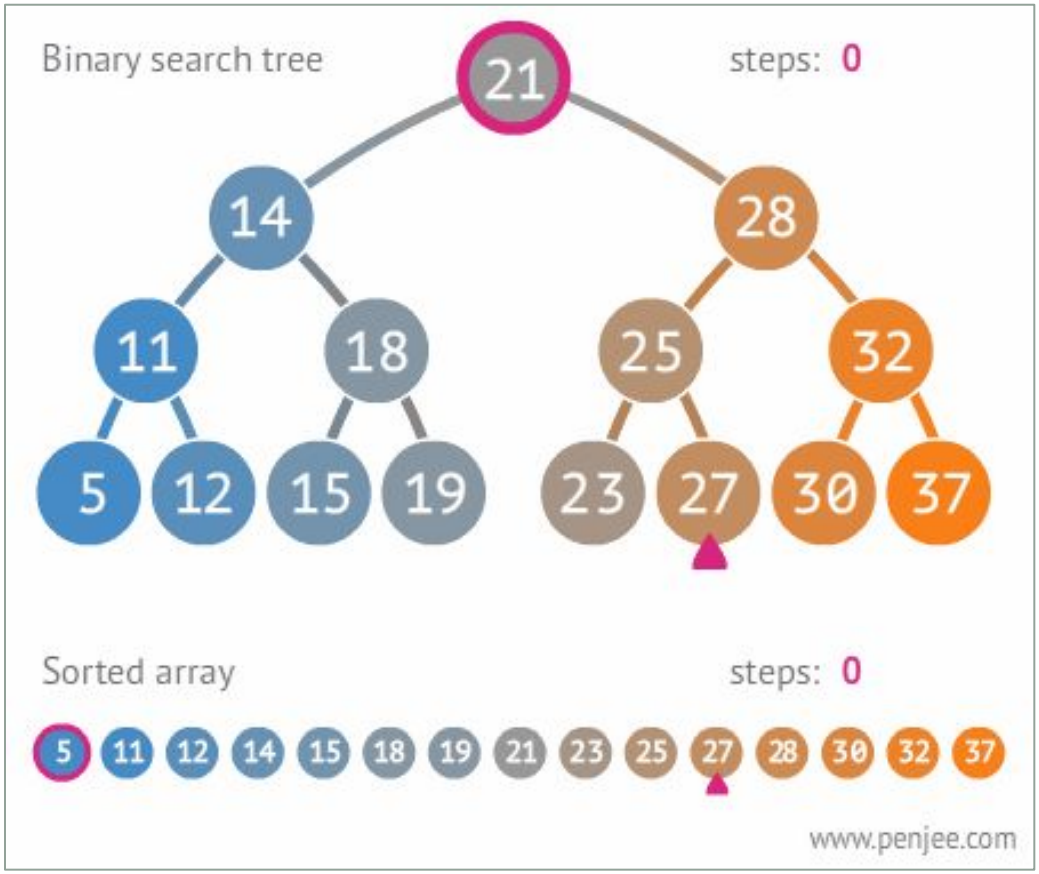
```
// Recursive function to insert a key into a BST
Node* insert(Node* root, int key) {
    // If the root is nullptr, create a new node and return it
    if (root == nullptr) {
        return new Node(key);
    }
    // If the given key is less than the root node, recur for the left subtree
    if (key < root->data) {
        root->left = insert(root->left, key);
    }
    // If the given key is more than the root node, recur for the right subtree
    else {
        root->right = insert(root->right, key);
    }

    return root;
}
```

```
struct TreeNode {
    int Data;
    TreeNode* Lchild;
    TreeNode* Rchild;
};

TreeNode* Insert(TreeNode* Root, int Key) {
    TreeNode* Tmp;
    TreeNode* NewNode = new TreeNode;
    NewNode->Data = Key;
    NewNode->Lchild = NewNode->Rchild =
     nullptr;

    if (Root == nullptr) {
        Root = NewNode;
        return Root;
    }
```

```
 Tmp = Root;
    while (Tmp != nullptr) {
        if (Tmp->Data < Key) {
            if (Tmp->Lchild == nullptr)
              {
                Tmp->Lchild = NewNode;
                return Root;
              }
            Tmp = Tmp->Lchild;
          }
        else
     {
            if (Tmp->Rchild == nullptr) {
                Tmp->Rchild = NewNode;
                return Root;
            }
            Tmp = Tmp->Rchild;
        }
    }
    return Root;
}
```

Searching

- To search for a target key, we first compare it with the key at the root of the tree.
- If it is the same, then the algorithm ends.
- If it is less than the key at the root, search for the target key in the left subtree, else search in the right subtree.



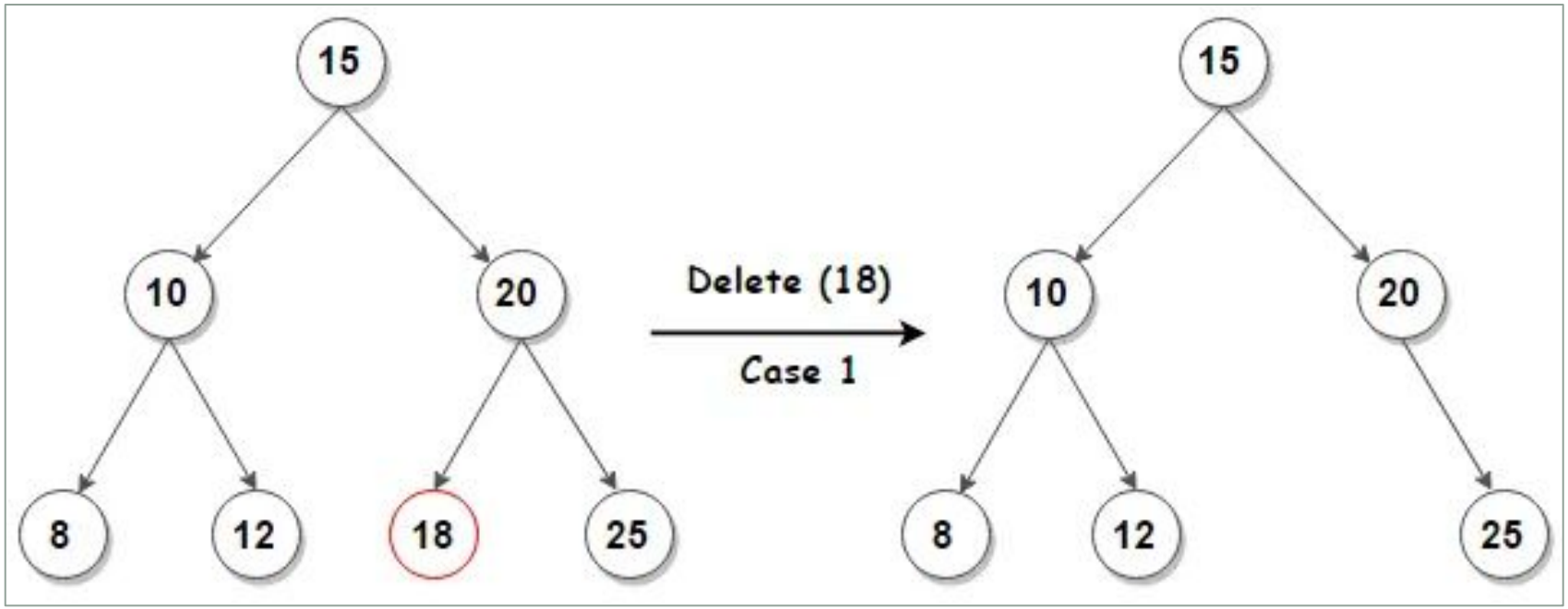
```
TreeNode* Rec_Search(TreeNode* root, int key)
{
    if (root == nullptr)
        return root;
    if (root->Data == key)
        return root;
    else if (root->Data < key)
        return Rec_Search(root->Lchild, key);
    else
        return Rec_Search(root->Rchild, key);
}
```

```
TreeNode* Search(TreeNode* Root, int Key)
 {
    TreeNode* Tmp = Root;
    while (Tmp)
    {
        if (Tmp->Data == Key)
            return Tmp;
        else if (Tmp->Data < Key)
            Tmp = Tmp->Lchild;
        else
            Tmp = Tmp->Rchild;
      }
    return nullptr;
}
```

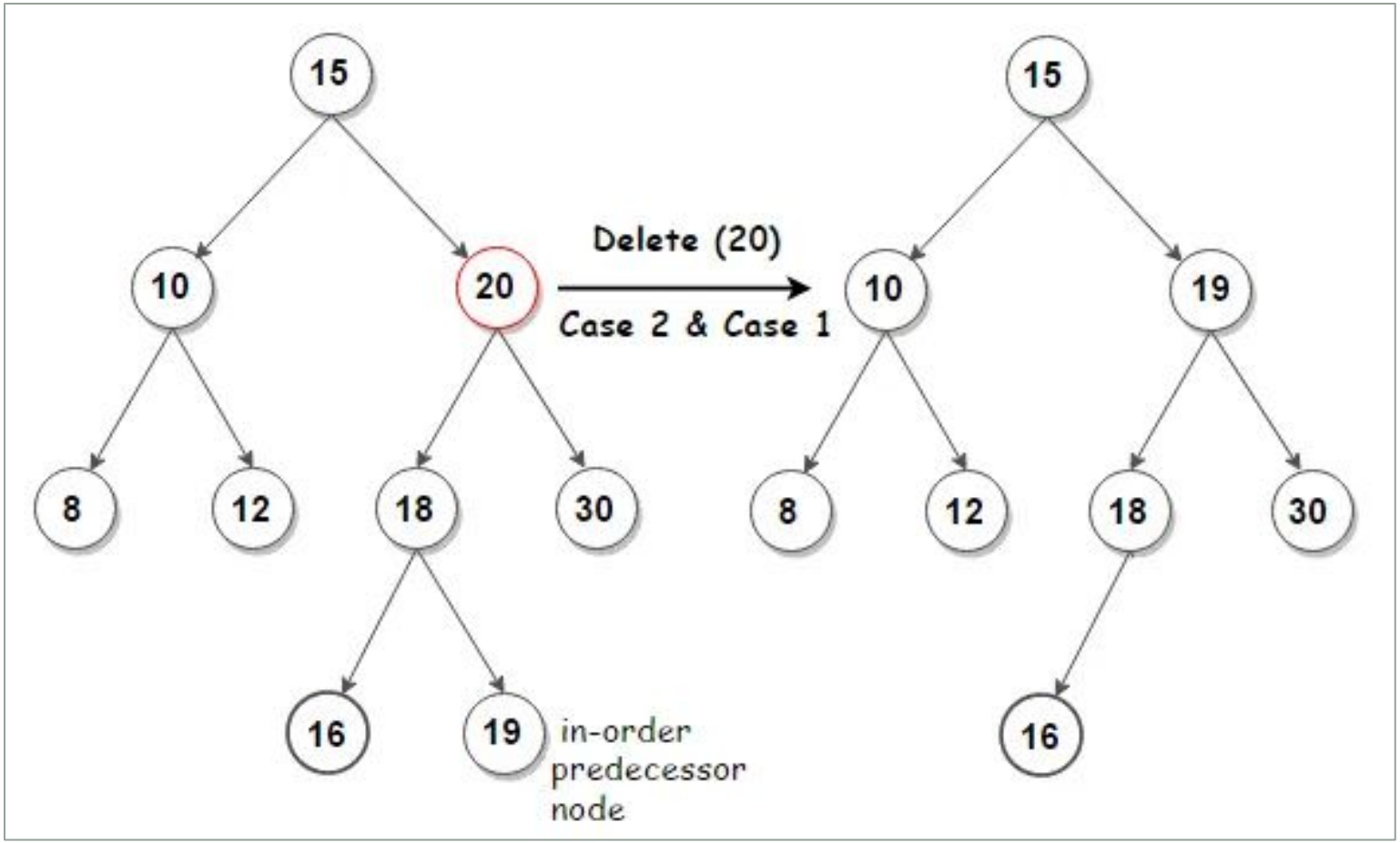
Deletion

Here are three possible cases to consider deleting a node from BST:

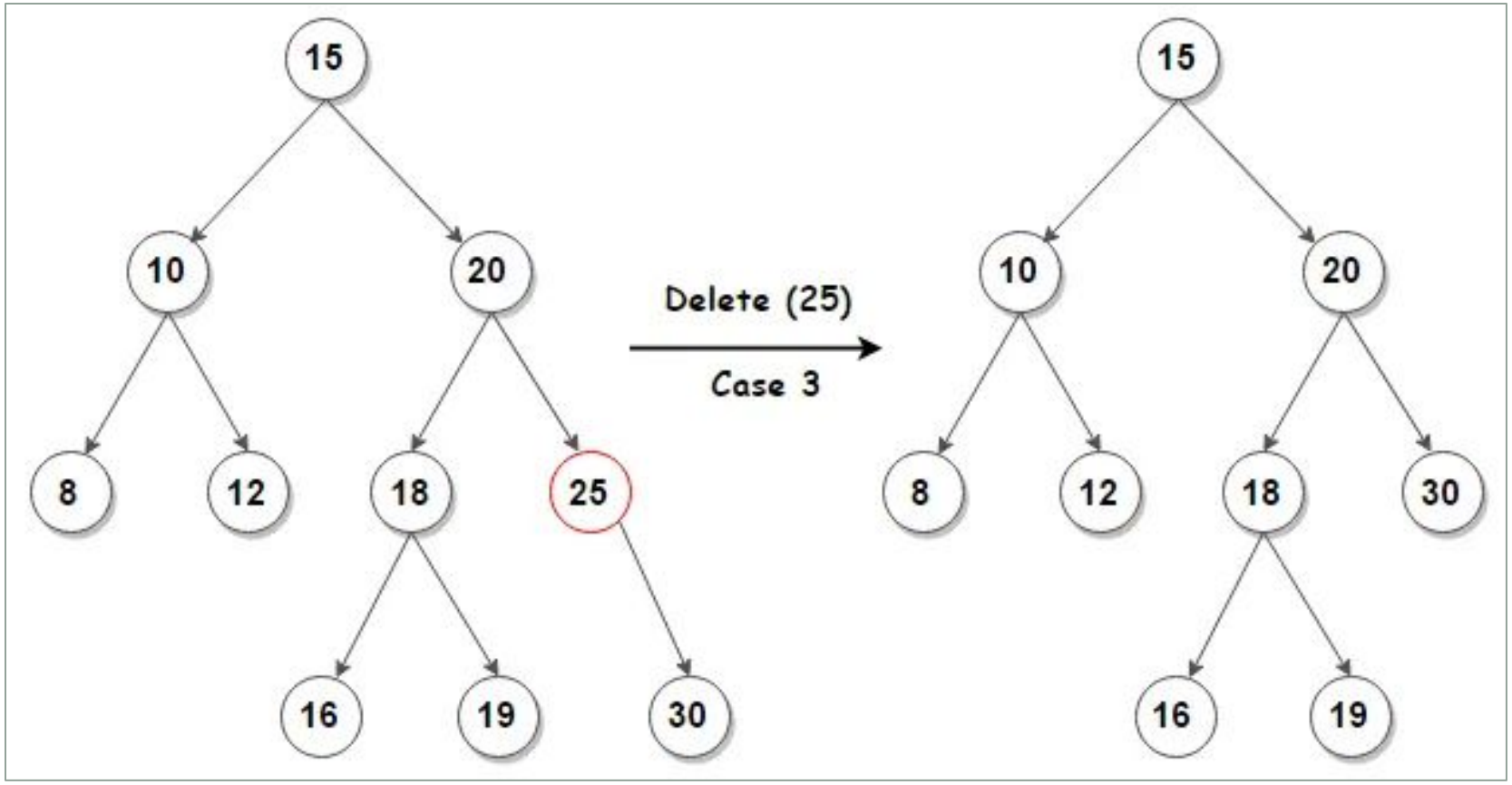
Case 1: Deleting a node with no children: remove the node from the tree.



Case 2: Deleting a node with two children



Case 3: Deleting a node with one child: remove the node and replace it with its child.



```
// Helper function to find minimum value node in the subtree rooted
at `curr` inorder Successor
TreeNode* inorder_Successor(TreeNode* curr)
{
    while (curr->left) {
        curr = curr->left;
    }
    return curr;
}
```

```
// Helper function to find maximum value node in the subtree rooted
at `curr` inorder Predecessor
TreeNode* inorder_Predecessor (TreeNode* curr)
{
    while (curr->right) {
        curr = curr->right;
    }
    return curr;
}
```

```
// Function to delete a node from a BST
TreeNode* deleteNode(TreeNode* root, int key) {
    // pointer to store the parent of the current node
    TreeNode* parent = nullptr;
    // start with the root node
    TreeNode* curr = root;
    // search key in the BST and set its parent pointer
    while (curr && curr->data != key) {
        // update the parent to the current node
        parent = curr;
// if the given key is less than the current node, go to the left
subtree;otherwise, go to the right subtree
        if (key < curr->data) {
            curr = curr->left;
        } else {
            curr = curr->right;
        }
    }
    // return if the key is not found in the tree
    if (curr == nullptr) {
        return root;
    }
```

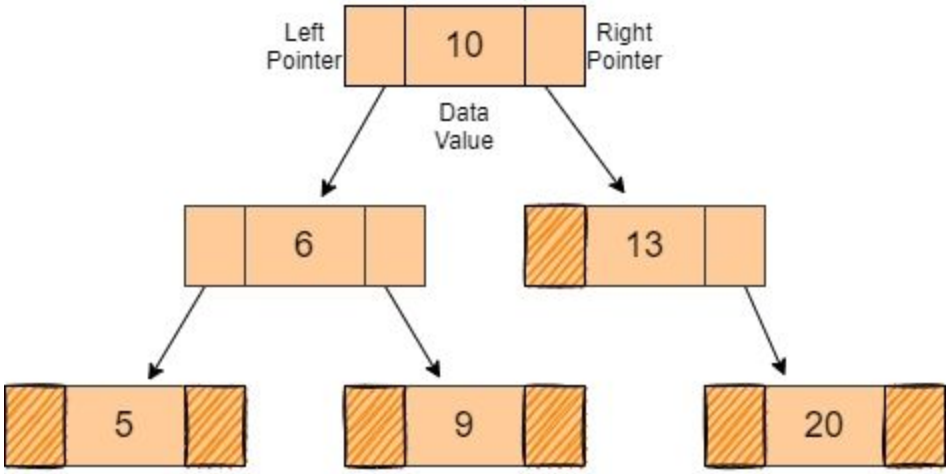
```
// Case 1: node to be deleted has no children, i.e., it is a leaf node
    if (curr->left == nullptr && curr->right == nullptr) {
        // if the node to be deleted is not a root node, then set its
        // parent left/right child to nullptr
        if (curr != root) {
            if (parent->left == curr) {
                parent->left = nullptr;
            } else {
                parent->right = nullptr;
            }
        }
        // if the tree has only a root node, set it to nullptr
        else {
            root = nullptr;
        }
        delete curr;
    }
```

```
// Case 2: node to be deleted has two children
   else if (curr->left && curr->right) {
       // find its inorder successor node
       TreeNode* successor = inorder_Successor(curr->right);
   //OR// find its inorder Predecessor node
      // TreeNode* successor = inorder_Predecessor(curr->left);
       // store successor value
       int val = successor->data;
       // recursively delete the successor. Note that the successor
       // will have at most one child (right child)
       deleteNode(root, successor->data);
       // copy value of the successor to the current node
       curr->data = val;
   }
```

```
 // Case 3: node to be deleted has only one child
    else {
        // choose a child node
        TreeNode* child = (curr->left) ? curr->left : curr->right;
        // if the node to be deleted is not a root node, set its parent
        // to its child
        if (curr != root) {
            if (curr == parent->left) {
                parent->left = child;
            } else {
                parent->right = child;
            }
        }
   // if the node to be deleted is a root node, then set the root to the child
        else {
            root = child;
        }
        delete curr;
    }
    return root;
}
```

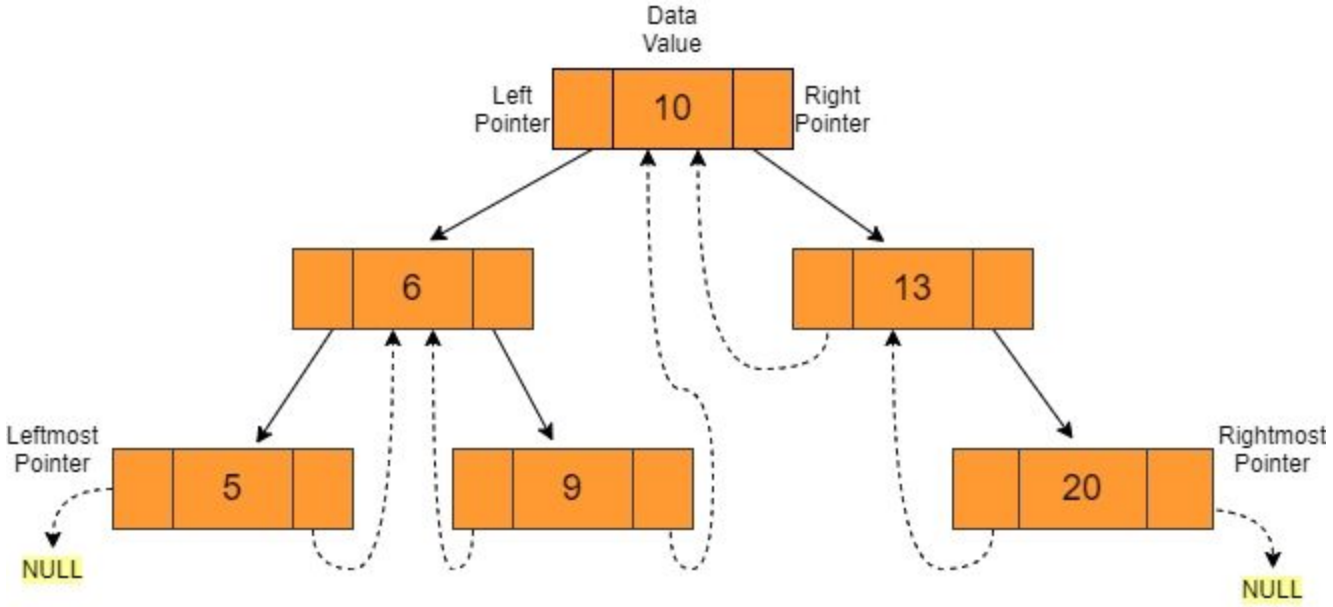
Threaded binary search tree

- A tree with a thread is called a threaded binary tree (TBT).
- In a threaded binary tree, either of the NULL pointers of leaf nodes is made to point to their successor or predecessor nodes.
- Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.
- If there is no in-order predecessor or in-order successor, then it points to the root node.



BINARY TREE

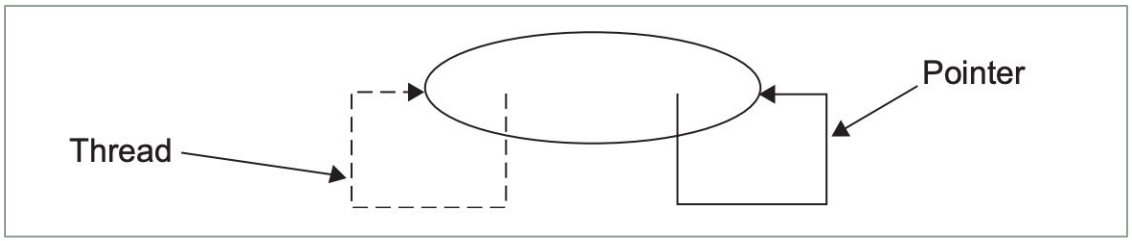
5 ,6 ,9 ,10 ,13 , 20



THREADED BINARY TREE

Threaded binary search tree-

- The difference is that the threads are not structural pointers of the tree.
- They can be removed but still the tree does not change.
- Tree pointers are the pointers that join and hold the tree together.
- Threads utilize the Null pointer's waste space to improve the processing efficiency.
- This allows us to traverse the tree both left to right and right to left without recursion.



Types of Threaded Binary tree

There are two types of Threaded Binary Trees:

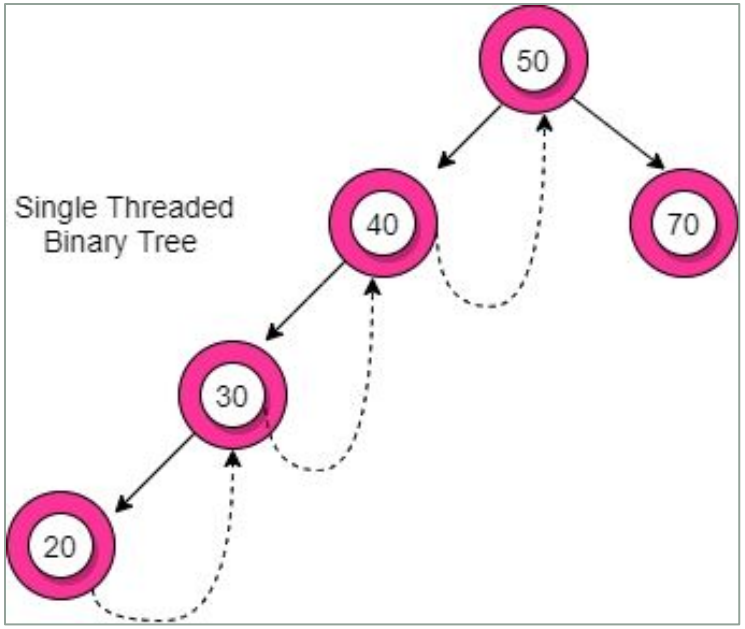
1. One-way Threaded Binary Tree
2. Two-way Threaded Binary Tree

1. One-way Threaded Binary Tree

In this type, if a node has a right null pointer, then this right pointer is threaded towards the in-order successor's node if it exists.

Node Structure of Single-Threaded Binary Trees

```
struct Node
{
        int value;
        Node* left;
        Node* right;
        bool rightThread;
 }
```

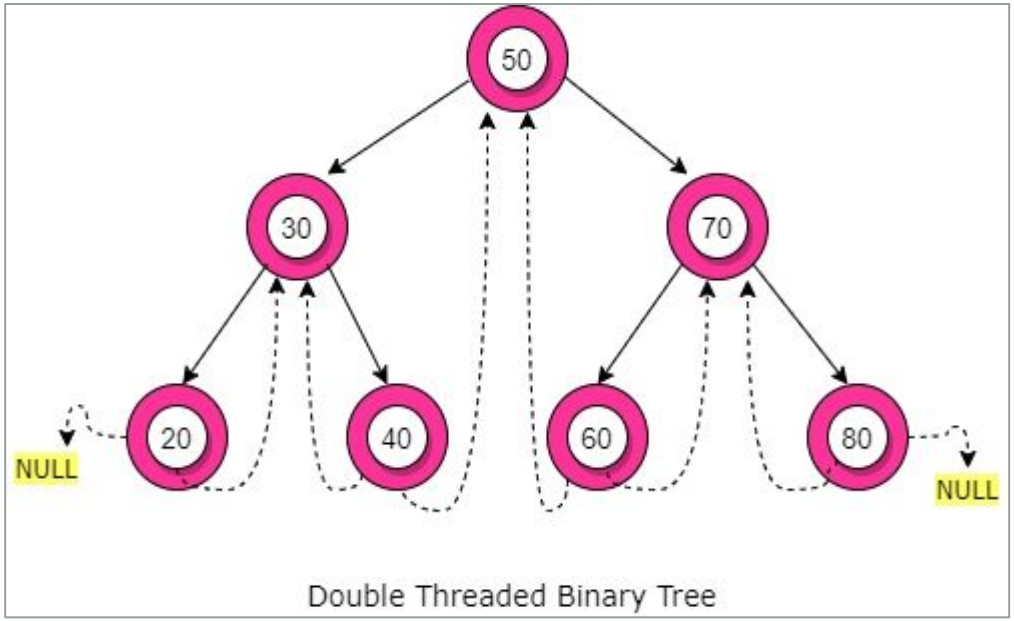


2. Two-way Threaded Binary Tree

In this type, the left null pointer of a node is made to point towards the in-order predecessor node and the right null pointer is made to point towards the in-order successor node.

Node Structure of Double-Threaded Binary Trees

```
struct Node
{
        int value;
        Node* left;
        Node* right;
        bool rightThread;
        bool leftThread;
 }
```



Double Threaded Binary Tree

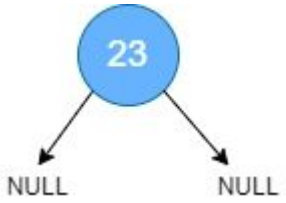
The Operations in a Threaded Binary Tree

Insert

Case 1: When a new node is inserted in an empty tree

The new node value becomes the root node, and both the left and right pointers of the value will be set to NULL.

```
root = Value;
newnode -> left = NULL;
newnode -> right = NULL;
```



The Operations in a Threaded Binary Tree

Insert

Case 2 : When a new node is inserted as a left child

- After inserting the node at its proper place, we will make its left and right child pointers point to in-order predecessor and successor, respectively. So the left and right threads of the new node will be:

```
newnode -> left = parent ->left;
newnode -> right = parent;
```

- Before insertion, the left pointer of the parent was a thread, but after insertion, it will be a link pointing to the new node.

```
parent -> leftThread = false;
parent -> left = newnode;
```

The Operations in a Threaded Binary Tree

Insert

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

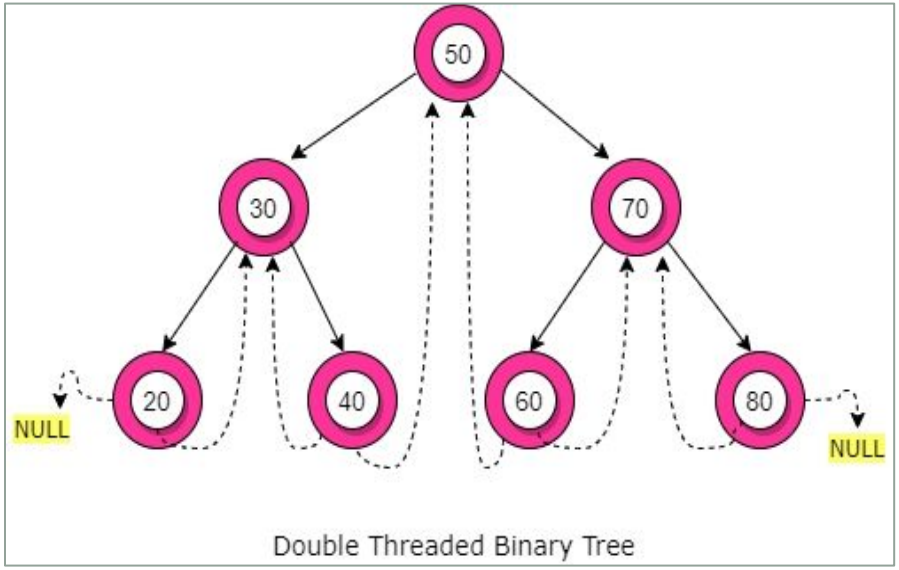
- The node that was the parent's in-order successor is now the in-order successor of this new node value. So the left and right threads of the new node will be:

```
newnode -> left = parent;
newnode -> right = parent -> right;
```

- Before insertion, the right pointer of the parent was a thread, but after insertion, it will be a link pointing to the new node.

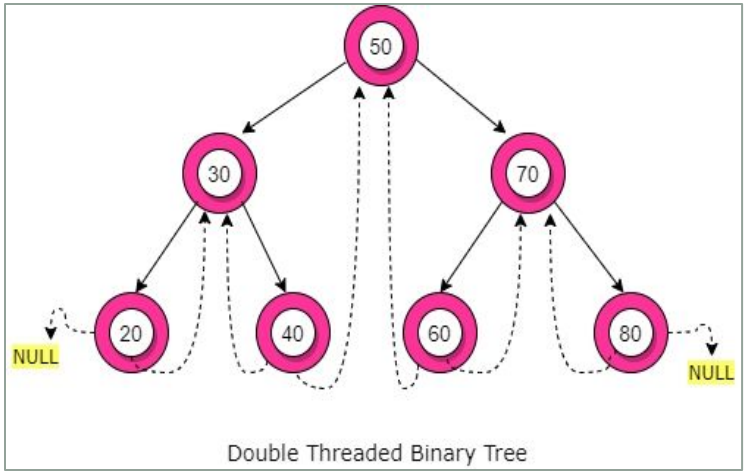
```
parent -> rightThread = false;
parent -> right = newnode;
```

```
using namespace std;

struct Node
{
    struct Node *left, *right;
    int info;
    // False if left pointer points to predecessor in Inorder Traversal
    bool lthread;
    // False if right pointer points to successor in Inorder Traversal
    bool rthread;
};
```



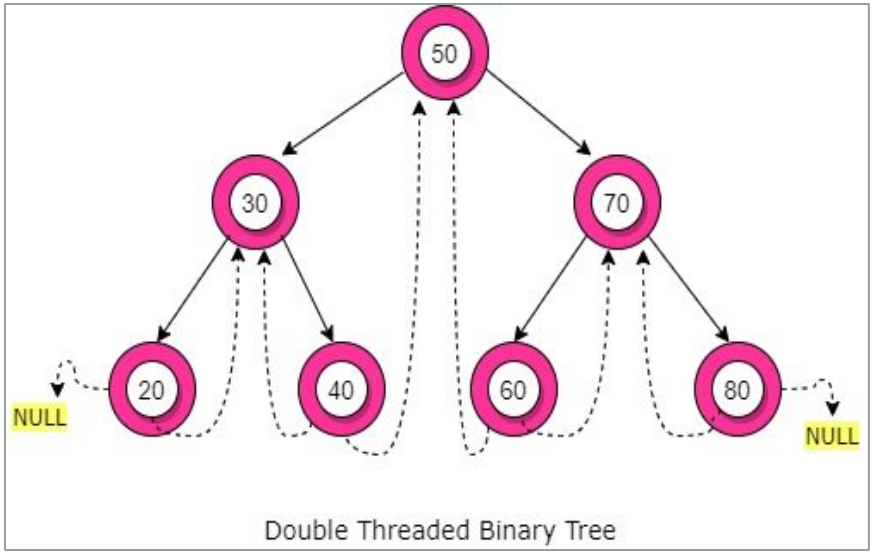
Double Threaded Binary Tree

```c
// Insert a Node in Binary Threaded Tree
struct Node *insert(struct Node *root, int ikey)
{
    // Searching for a Node with given value
    Node *ptr = root;
    Node *par = NULL; // Parent of key to be inserted
    while (ptr != NULL)
    {
        // If key already exists, return
        if (ikey == (ptr->info))
        {
            printf("Duplicate Key !\n");
            return root;
        }
         par = ptr; // Update parent pointer
        if (ikey < ptr->info) // Moving on left subtree.
        {
            if (ptr -> lthread == false)
                ptr = ptr -> left;
            else
                break;
        }
        else  // Moving on right subtree.
        {
            if (ptr->rthread == false)
                ptr = ptr -> right;
            else
                break;
        }
    }
```



Double Threaded Binary Tree

```
// Create a new node
    Node *tmp = new Node;
    tmp -> info = ikey;
    tmp -> lthread = true;
    tmp -> rthread = true;

    if (par == NULL)
    {
        root = tmp;
        tmp -> left = NULL;
        tmp -> right = NULL;
    }
    else if (ikey < (par -> info))
    {
        tmp -> left = par -> left;
        tmp -> right = par;
        par -> lthread = false;
        par -> left = tmp;
    }
    else
    {
        tmp -> left = par;
        tmp -> right = par -> right;
        par -> rthread = false;
        par -> right = tmp;
    }

    return root;
}
```



Double Threaded Binary Tree

The Operations in a Threaded Binary Tree

Deletion

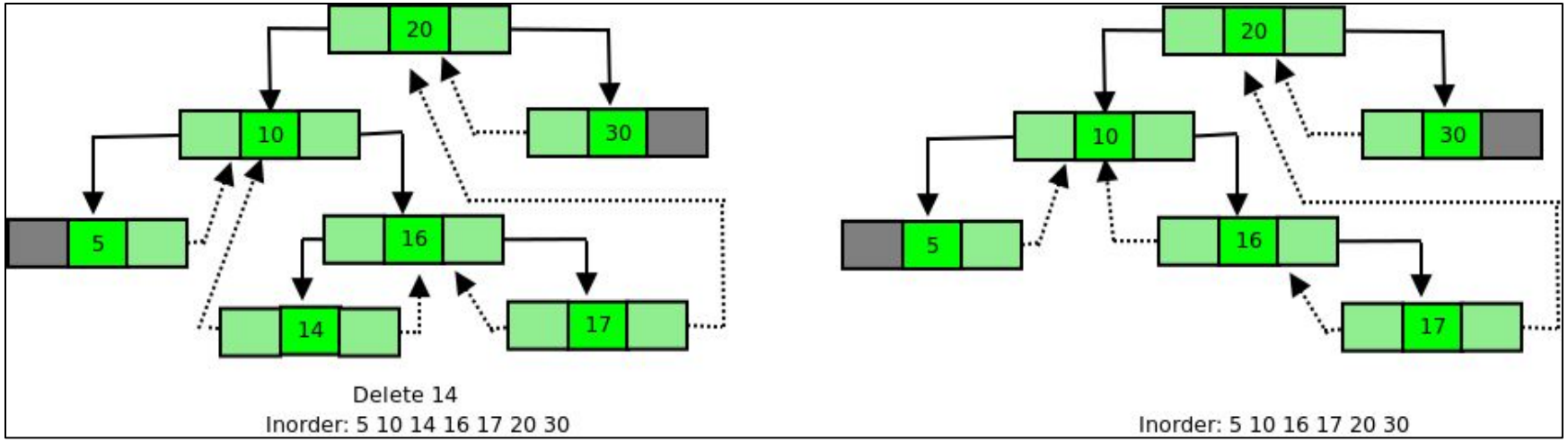
- In deletion, first the key to be deleted is searched, and then there are different cases for deleting the Node in which key is found.

Case 1: When a leaf node needs to be deleted (Left Child)

- When deleting a leaf Node in BST, the left or right pointer of the parent node is set to NULL. Whereas in Threaded binary search trees, it is turned into a thread instead of setting the pointer to NULL.
- If the leaf Node is the left child of its parent, then after deletion, the parent's left pointer should become a thread referring to its predecessor.

The Operations in a Threaded Binary Tree

Deletion



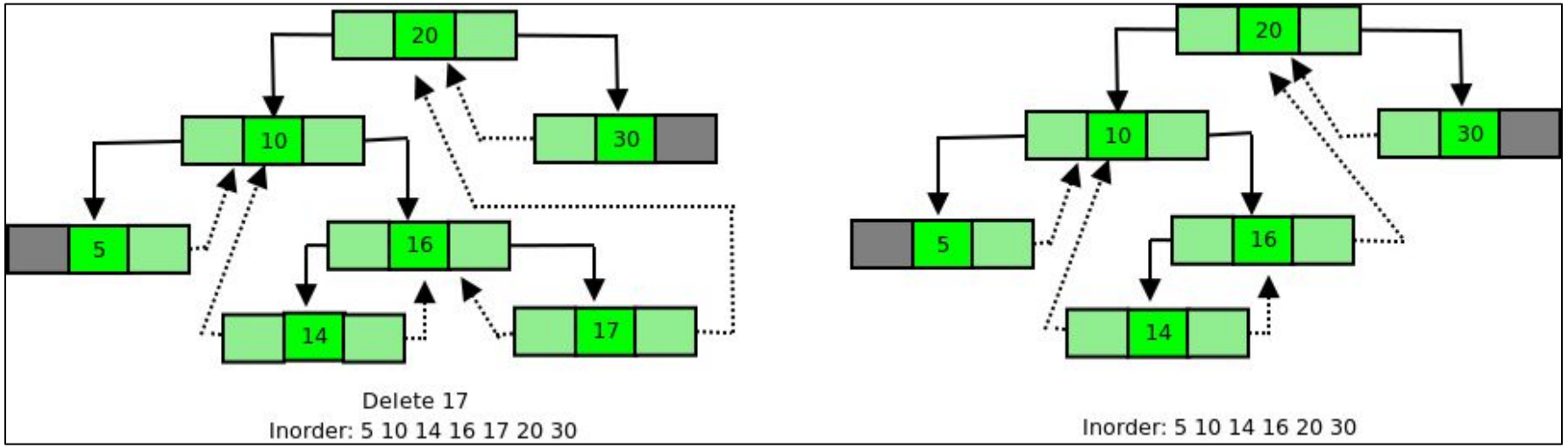

```
par -> lthread = true;

par -> left = ptr -> left;
```

The Operations in a Threaded Binary Tree

Deletion

Case 1: When a leaf node needs to be deleted (Right Child)

- If the leaf Node to be deleted is right child of its parent then after deletion, right pointer of parent should become a thread pointing to its successor.
- The Node which was inorder successor of the leaf Node before deletion will become the inorder successor of the parent Node after deletion.

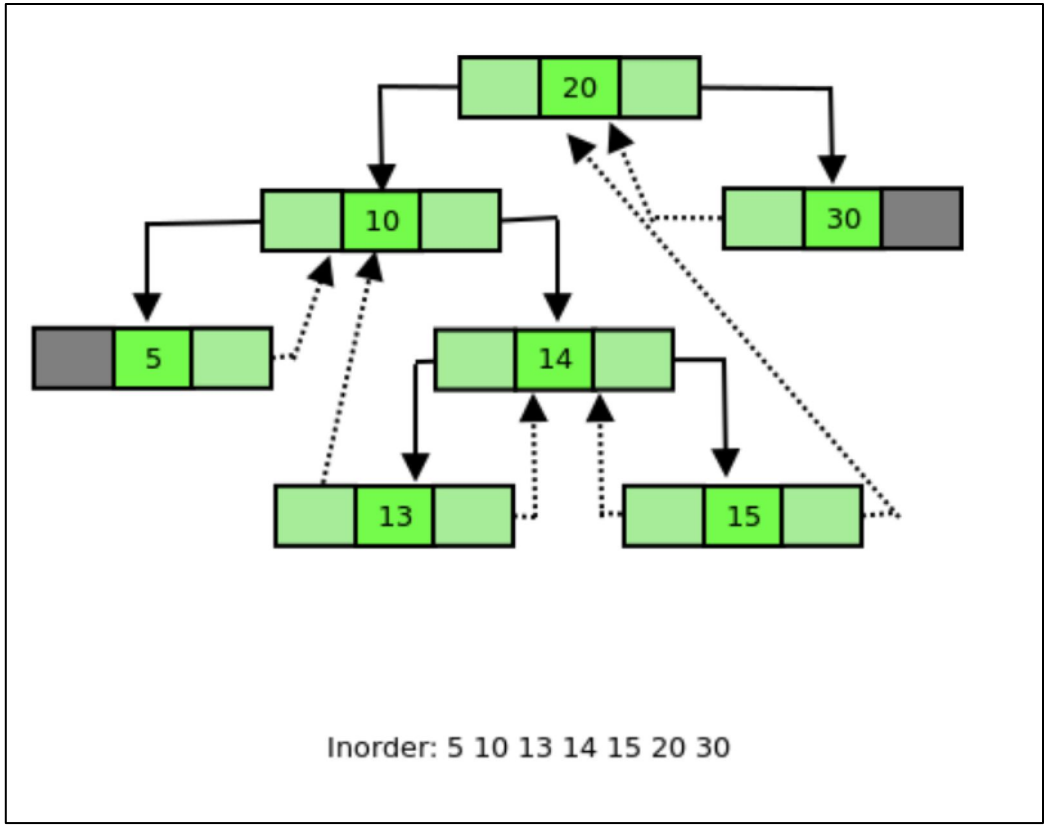



```
par -> rthread = true;
par -> right = ptr -> right;
```

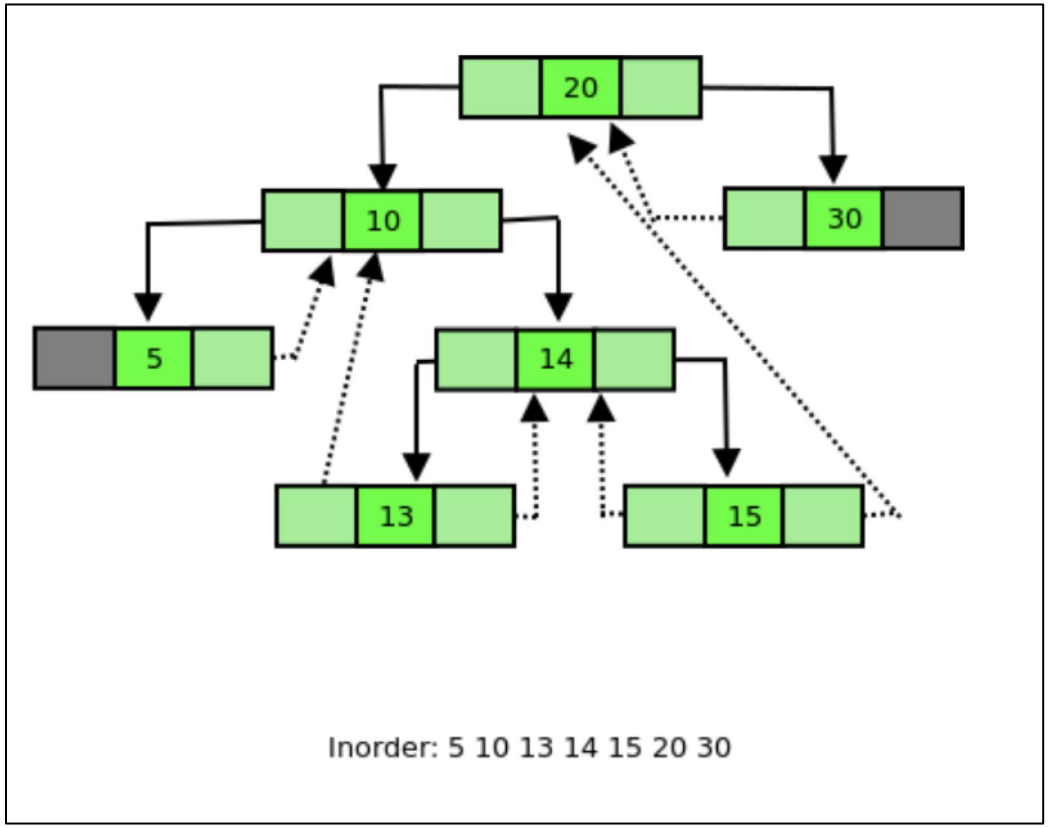
Deletion - Case A

```
// Here 'par' is pointer to parent Node and 'ptr' is pointer to current Node.
struct Node* caseA(struct Node* root, struct Node* par,
                   struct Node* ptr)
{
   // If Node to be deleted is root
   if (par == NULL)
      root = NULL;

   // If Node to be deleted is left of its parent
   else if (ptr == par->left) {
      par->lthread = true;
      par->left = ptr->left;
   }
   else {
      par->rthread = true;
      par->right = ptr->right;
   }

   // Free memory and return new root
   free(ptr);
   return root;
}
```

```
// Returns inorder successor using left and right children (Used in deletion)
struct Node* inSucc(struct Node* ptr)
{
    if (ptr->rthread == true)
        return ptr->right;

    ptr = ptr->right;
    while (ptr->lthread == false)
        ptr = ptr->left;

    return ptr;
}
```



```
struct Node* inPred(struct Node* ptr)
{
    if (ptr->lthread == true)
        return ptr->left;

    ptr = ptr->left;
    while (ptr->rthread == false)
        ptr = ptr->right;
    return ptr;
}
```



Case B: Node to be deleted has only one child

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

```
s = inSucc(ptr);
p = inPred(ptr);
```

- If Node to be deleted has left subtree, then after deletion right thread of its predecessor should point to its successor.

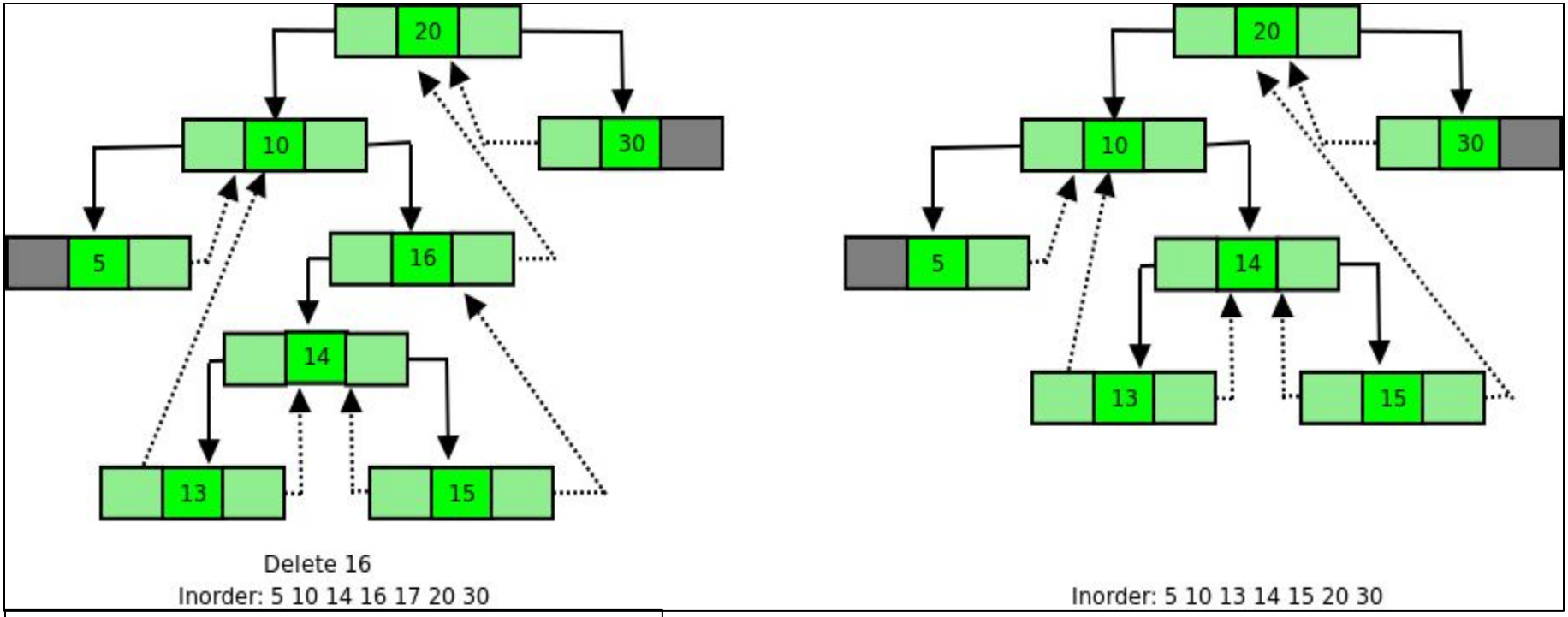
```
p->right = s;
```

The Operations in a Threaded Binary Tree

Deletion

```
s = inSucc(ptr);
p = inPred(ptr);
```

```
p->right = s;
```



inorder - 5 10 13 14 15 16 20 30

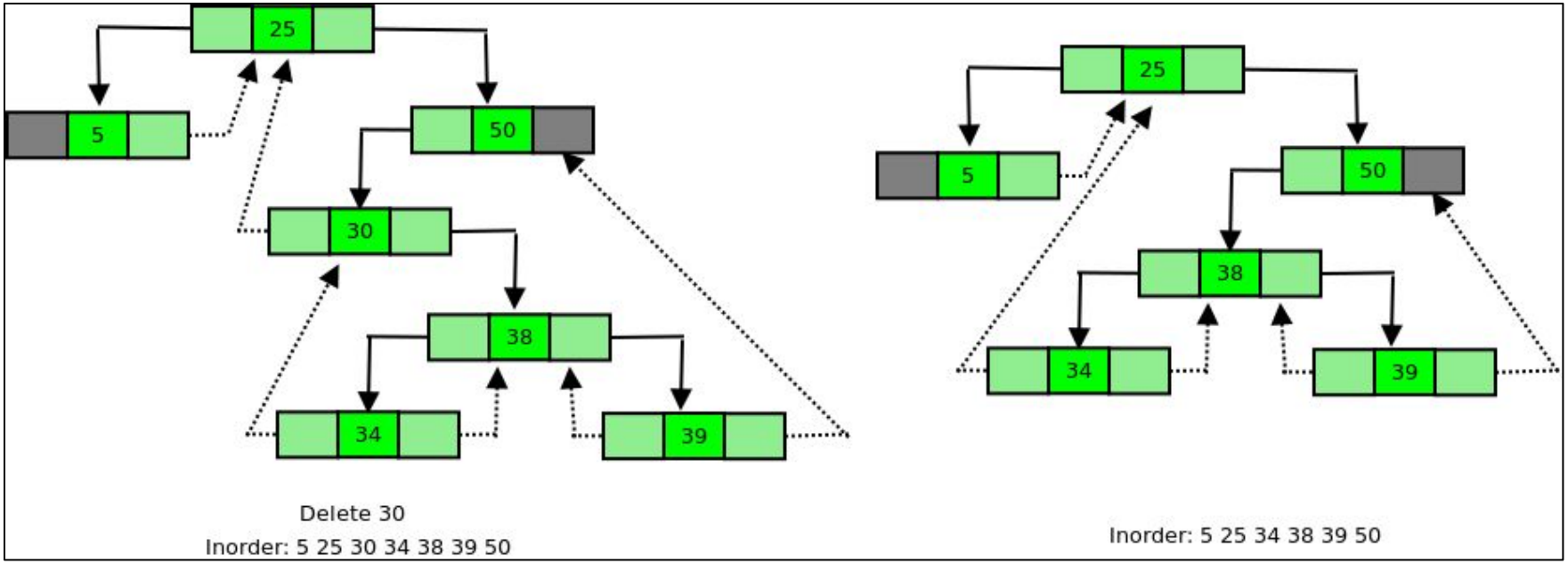
The Operations in a Threaded Binary Tree

Deletion

- If Node to be deleted has right subtree, then after deletion left thread of its successor should point to its predecessor.

```
s = inSucc(ptr);
p = inPred(ptr);
```

```
s->left = p;
```



Deletion- Case B

```
// Here 'par' is pointer to parent Node and 'ptr' is pointer to current Node.
struct Node* caseB(struct Node* root, struct Node* par, struct Node* ptr)
{
   struct Node* child;
   // Initialize child Node to be deleted has left child.
   if (ptr->lthread == false)
      child = ptr->left;
   // Node to be deleted has right child.
   else
      child = ptr->right;
    // Node to be deleted is root Node.
   if (par == NULL)
      root = child;
```

Deletion Case B

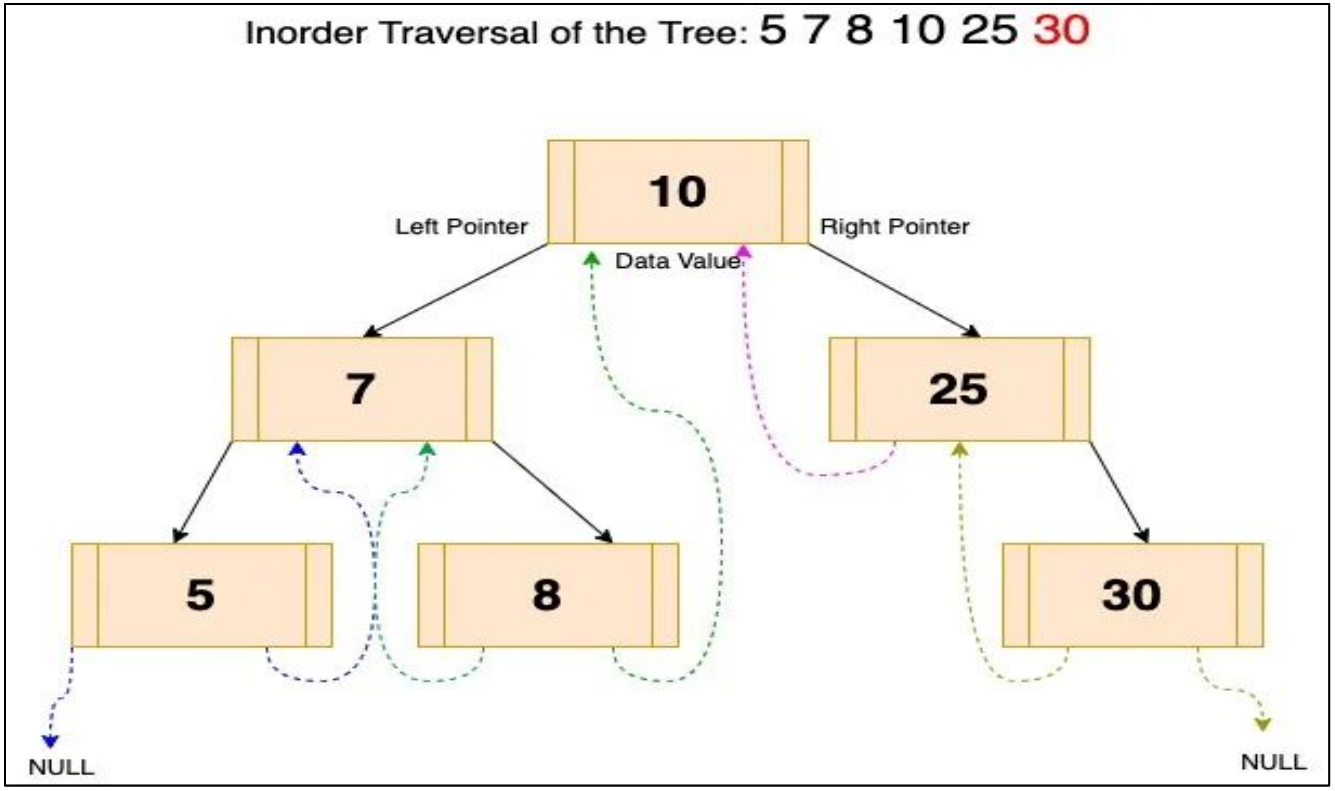
```
// Node is left child of its parent.
   else if (ptr == par->left)
      par->left = child;
   else
      par->right = child;

   // Find successor and predecessor
   Node* s = inSucc(ptr);
   Node* p = inPred(ptr);

   // If ptr has left subtree.
   if (ptr->lthread == false)
      p->right = s;

   // If ptr has right subtree.
   else {
      if (ptr->rthread == false)
         s->left = p;
   }

   free(ptr);
   return root;
}
```

The Operations in a Threaded Binary Tree

Deletion Case C

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.

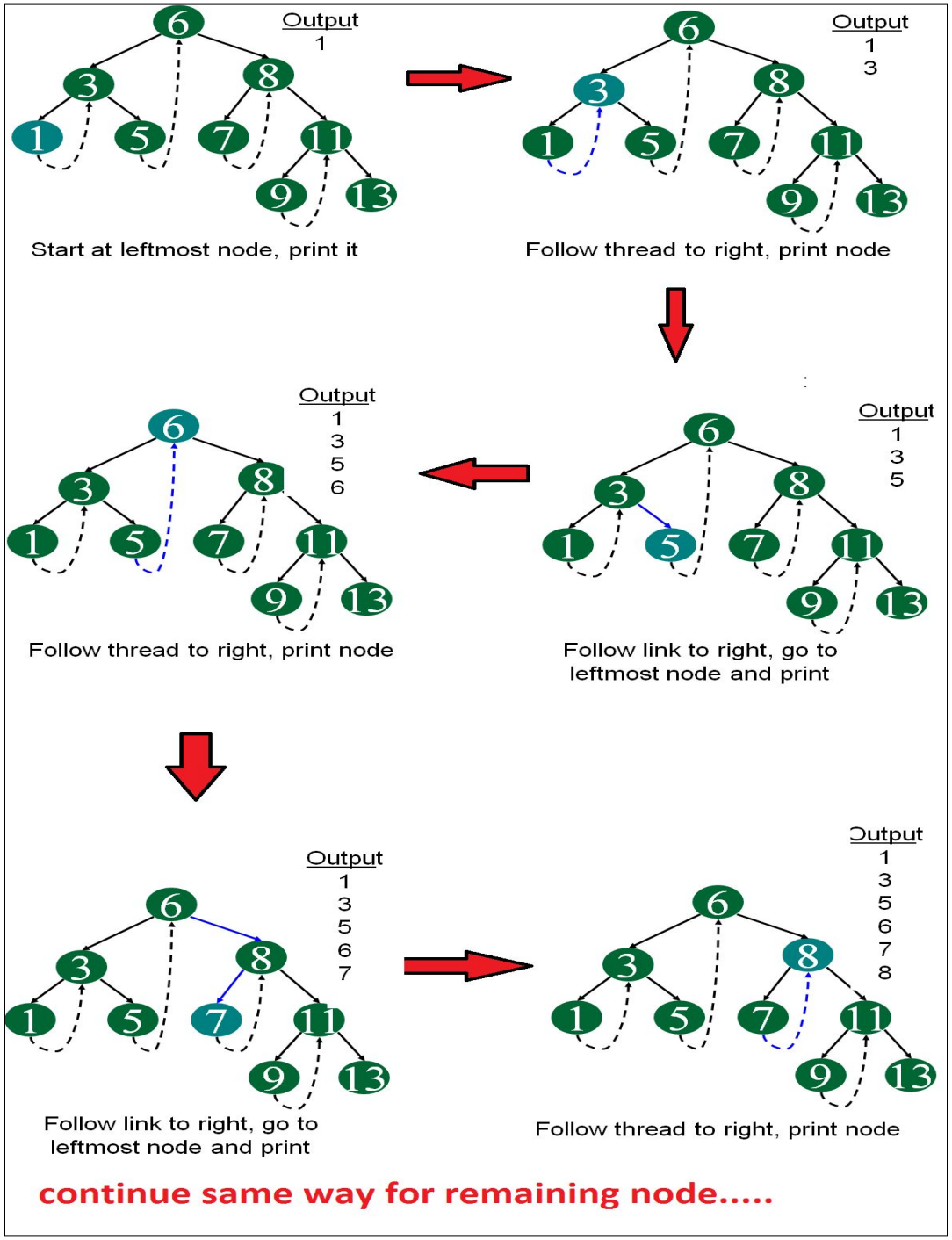


The Operations in a Threaded Binary Tree

Deletion

```
// Here 'par' is pointer to parent Node and 'ptr' is pointer to current Node.
struct Node* caseC(struct Node* root, struct Node* par , struct Node* ptr)
{
        // Find inorder successor and its parent.
        struct Node* par_succ = ptr;
        struct Node* succ = ptr->right;

        // Find leftmost child of successor
        while (succ->left != NULL)
         {
                par_succ = succ;
                succ = succ->left;
        }

        ptr->info = succ->info;

        if (succ->lthread == true && succ->rthread == true)

                root = caseA(root, par_succ, succ);
        else
                root = caseB(root, par_succ, succ);

        return root;
}
```

Inorder Traversal



```
// Utility function to find leftmost node in a tree rooted with n
Node* leftMost(Node* n)
{
    if (n == NULL)
        return NULL;

    while (n->left != NULL)
        n = n->left;

    return n;
}
```

The Operations in a Threaded Binary Tree

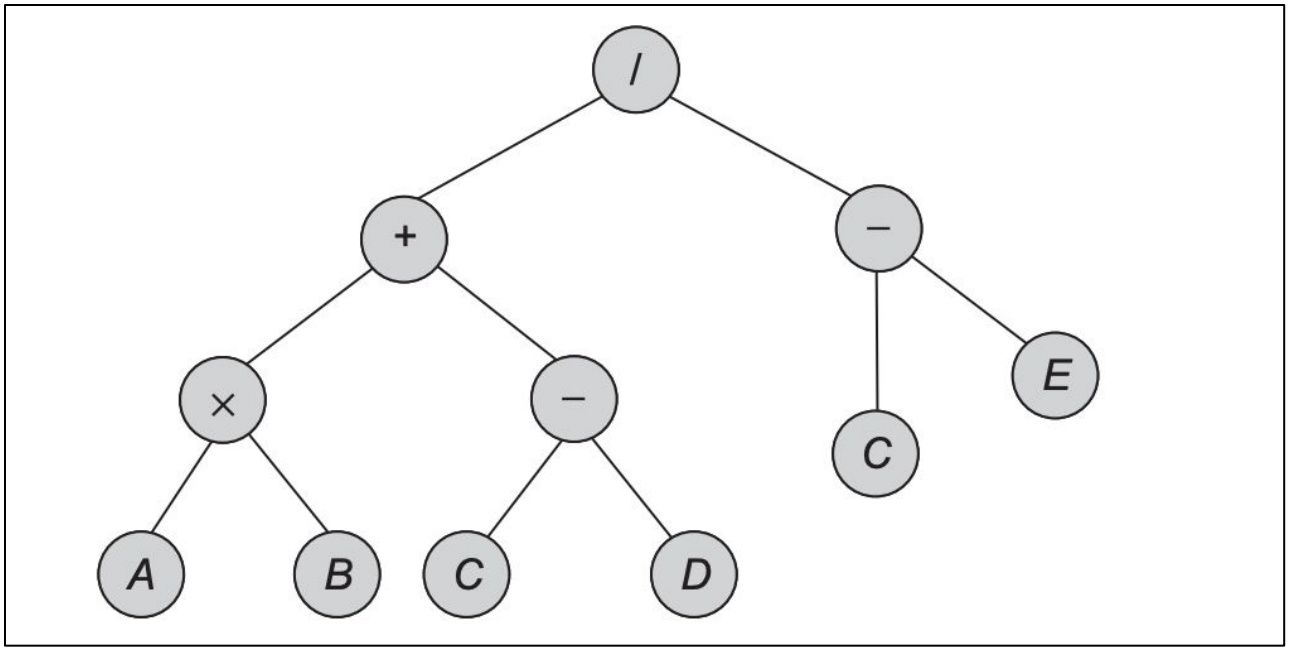
Inorder Traversal

```
// inorder traversal in a threaded binary tree
void inOrder(Node* root)
{
    Node* cur = leftMost(root);
    while (cur != NULL)
    {
        cout<<cur->data<<" ";
        // If this node is a thread node, then go to inorder successor
        if (cur->rightThread)
            cur = cur->right;
        else // Else go to the leftmost child in right subtree
            cur = leftmost(cur->right);
    }
}
```

Expression Tree

- A binary tree storing or representing an arithmetic expression is called as expression tree.
- The leaves of an expression tree are operands. Operands could be variables or constants.
- The branch nodes (internal nodes) represent the operators.
- A binary tree is the most suit_x0002_able one for arithmetic expressions as it contains either binary or unary operators.

Let $E = ((A \times B) + (C - D))/(C - E)$



- In the expression tree as in Fig. an infix expression is represented by representing the node as an operator, and the left and right subtrees are the left and right operands of that operator.

Construction of Expression Tree - algorithm

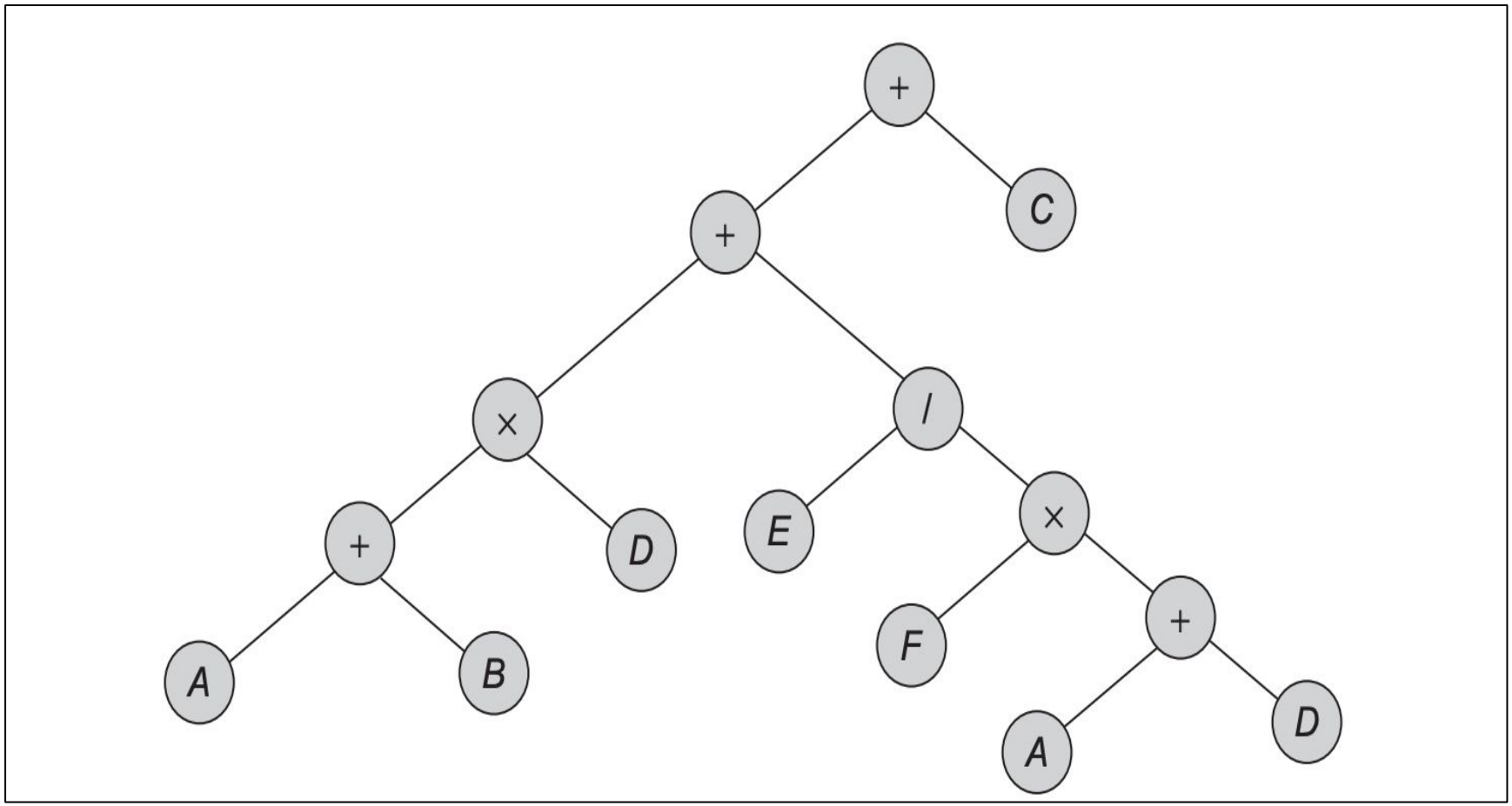
(**Scan the postfix expression from left to right**.)

1. Get one token from expression E.
2. Create a node say curr for it.
3. If (symbol is operand) then
   (a) push a node curr onto a stack.
4. else if (symbol is operator) then
   (a) T2 = pop()
   T1 = pop()
   Here T1 and T2 are pointers to left trees and right subtrees of the operator, respectively.
   (b) Attach T1 to left and T2 to the right of curr.
   (c) Form a tree whose root is the operator and T1 and T2 are left and right children, respectively.
   (d) Push the node curr having attached left and right subtrees onto a stack.
5. Repeat steps 1–4 till the end of expression.
6. Pop the node curr from the stack, which is a pointer to the root of expression tree.

Represent $AB + D \times EFAD \times + / + C +$ as an expression tree.

Solution - Figure represents the given expression in the form of a tree.

- First, the infix expression is converted to a postfix expression.



Building Expression tree from Prefix Expression

Algorithm: ConstructExpressionTreeFromPrefix

Input: Prefix expression as a string

***+ab-cd**

Output: Root of the expression tree

1. Initialize an empty stack of nodes.
2. Reverse the given prefix expression.
3. For each character 'ch' in the reversed prefix expression:
    a. Create a new node 'newNode' with 'ch' as data and both left and right children set to nullptr.
4. If 'ch' is an operand (variable), push 'newNode' onto the stack.
    - StackPush(stack, newNode)
5. If 'ch' is an operator:
    a. Pop the top two nodes 'op2' and 'op1' from the stack.
    b. Set 'newNode' as the root, 'op1' as the left child, and 'op2' as the right child.
    c. Push 'newNode' onto the stack.
    - StackPush(stack, newNode)
6. After processing all characters, the stack contains the root of the expression tree.
7. Return the root of the expression tree.

End Algorithm

Formation of Binary Tree from its Traversals

The basic principle for formulation is as follows:

1. If the preorder traversal is given, then the first node is the root node. If the postorder traversal is given, then the last node is the root node.

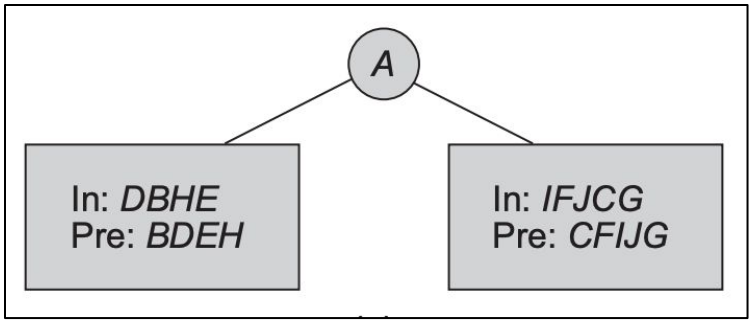
2. Once the root node is identified, all the nodes in all left and right subtrees of the root node can be identified.

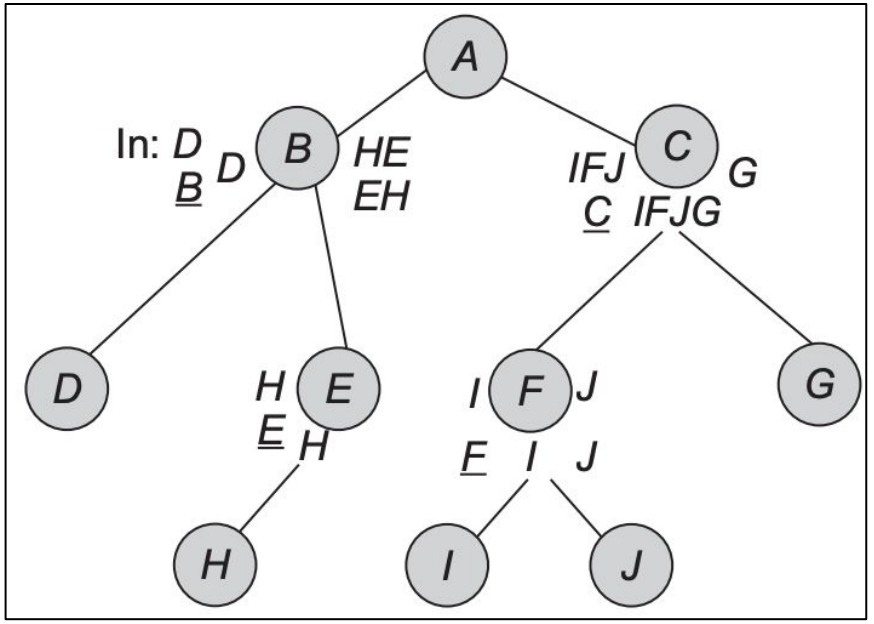
3. Same techniques can be applied repeatedly to form the subtrees.

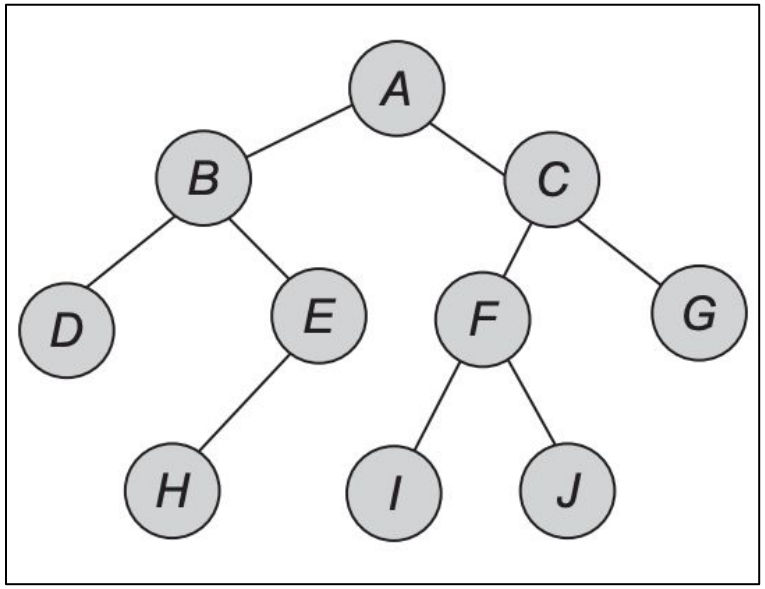
Construct a binary tree using the following two traversals:

Inorder : *D B H E A I F J C G*

Preorder: *A B D E H C F I J G*

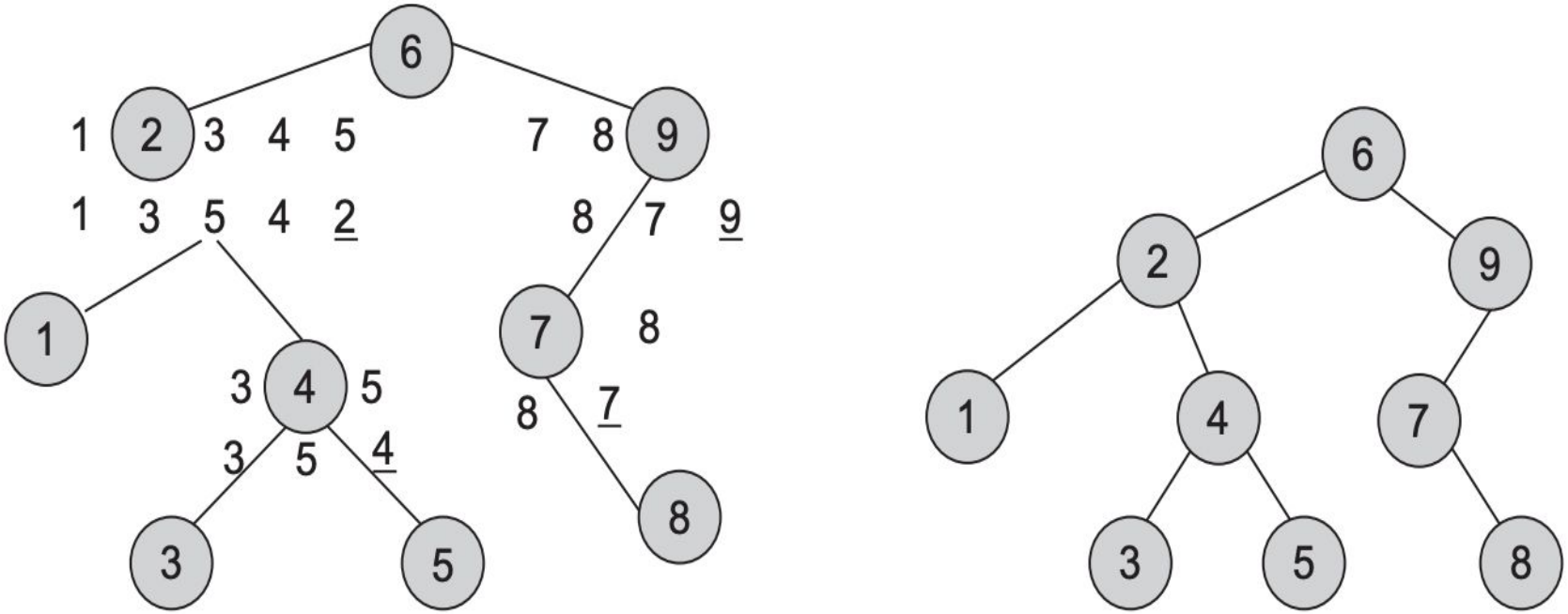






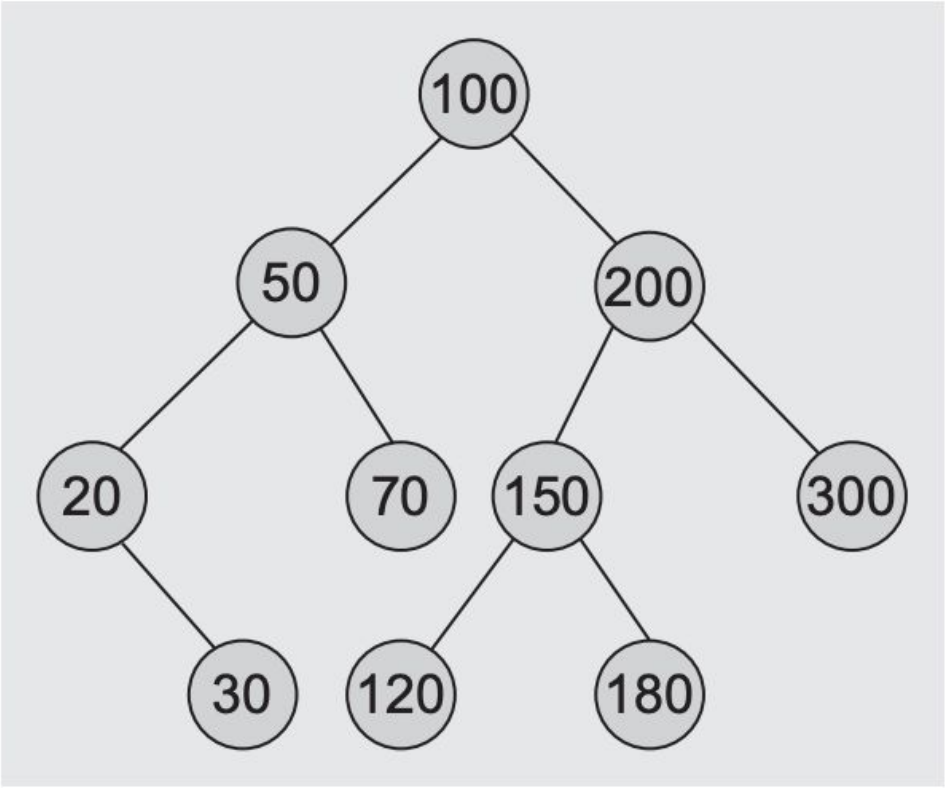
Construct a binary tree from its inorder and postorder traversals.

Inorder	:	1	2	3	4	5	6	7	8	9
Postorder:		1	3	5	4	2	8	7	9	6



Example - Build a BST from the following set of elements—100, 50, 200, 300,

20, 150, 70, 180, 120, 30—and traverse the tree built in inorder, postorder, and preorder.

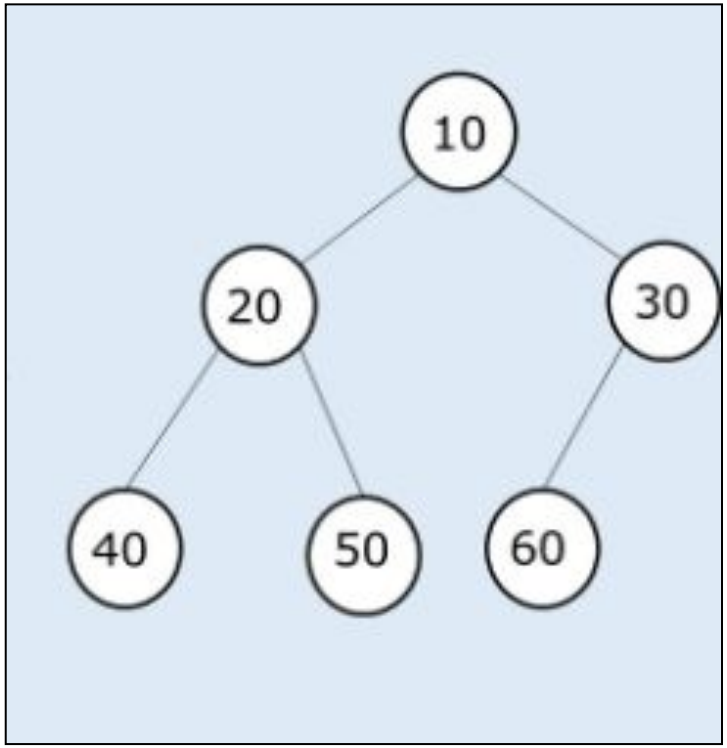


Preorder: 100 50 20 30 70 200 150 120 180 300
Inorder: 20 30 50 70 100 120 150 180 200 300
Postorder: 30 20 70 50 120 180 150 300 200 100

Construct A Binary Tree from Inorder and Preorder Traversal

Inorder :	40	20	50	10	60	30

Preorder :	10	20	40	50	30	60



InOrder(root) visits nodes in the following order:

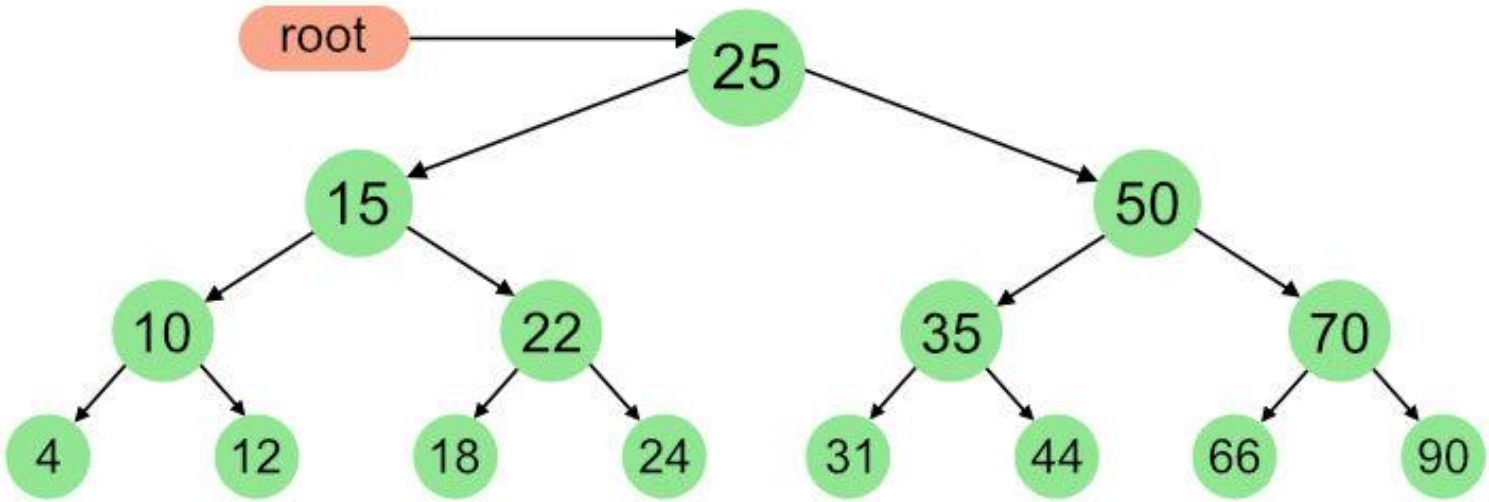
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

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

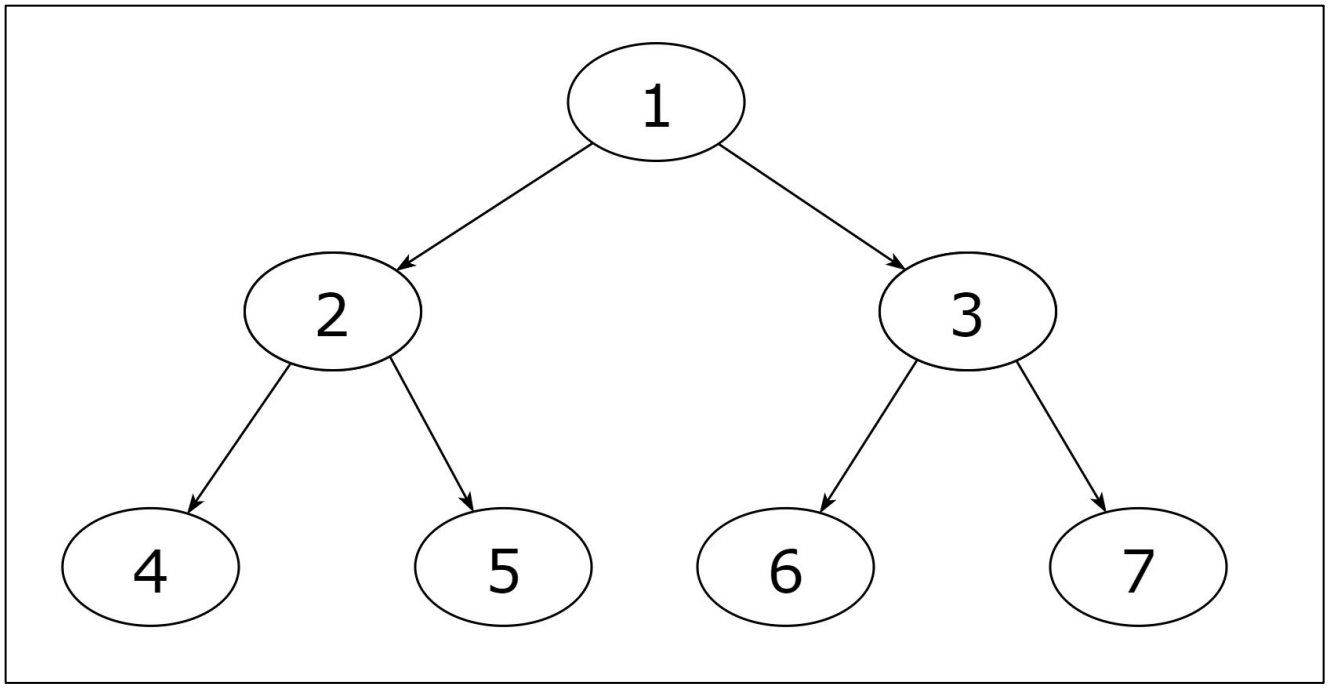


Example:

‘POSTORDER’ = [4, 5, 2, 6, 7, 3, 1]

‘PREORDER’ = [1, 2, 4, 5, 3, 6, 7]

A binary tree that matches the given ‘POSTORDER’ and ‘PREORDER’ traversal is:

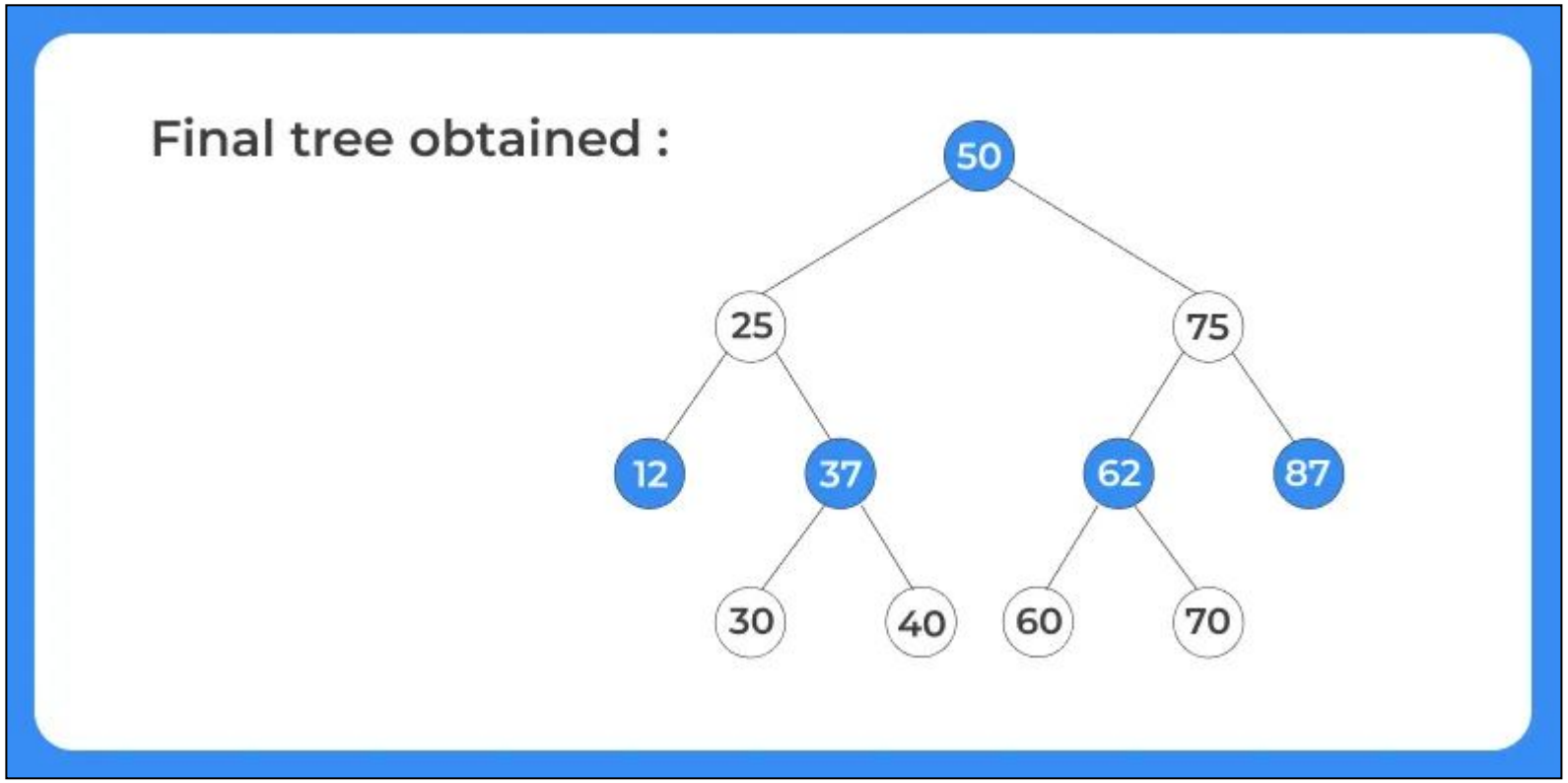


Construct Tree From Given Postorder and Preorder Traversal

Given traversals:

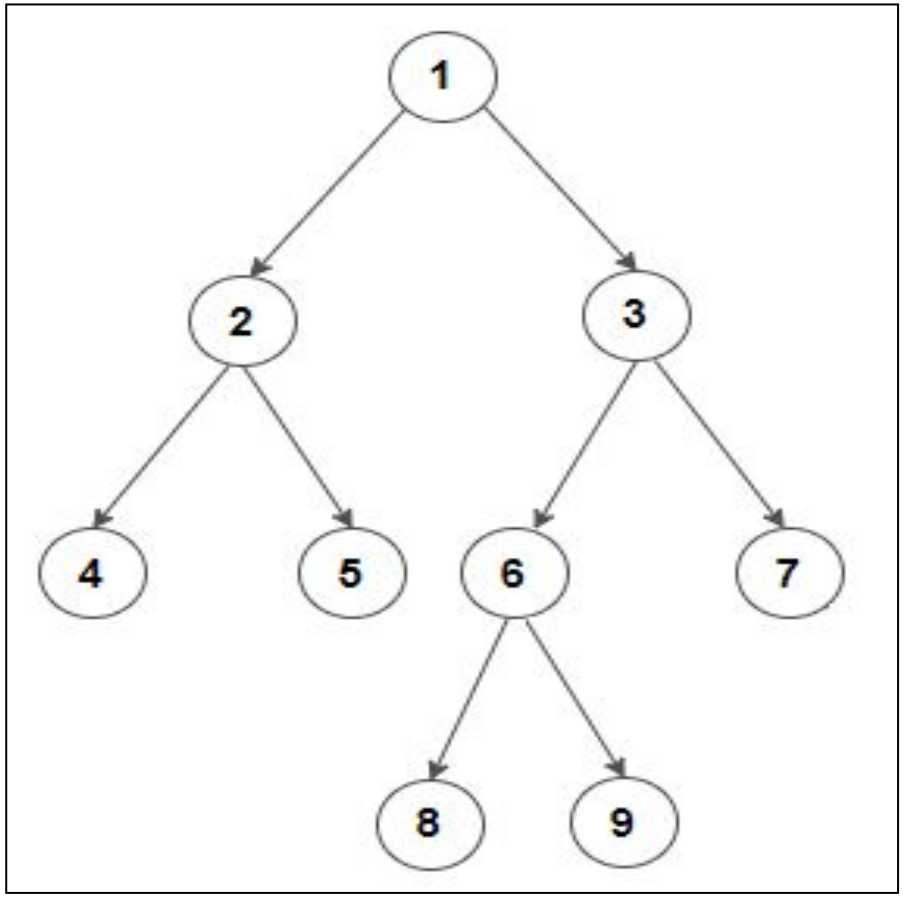
Preorder: 50, 25, 12, 37, 30, 40, 75, 62, 60, 70, 87

Postorder: 12, 30, 40, 37, 25, 60, 70, 62, 87, 75, 50

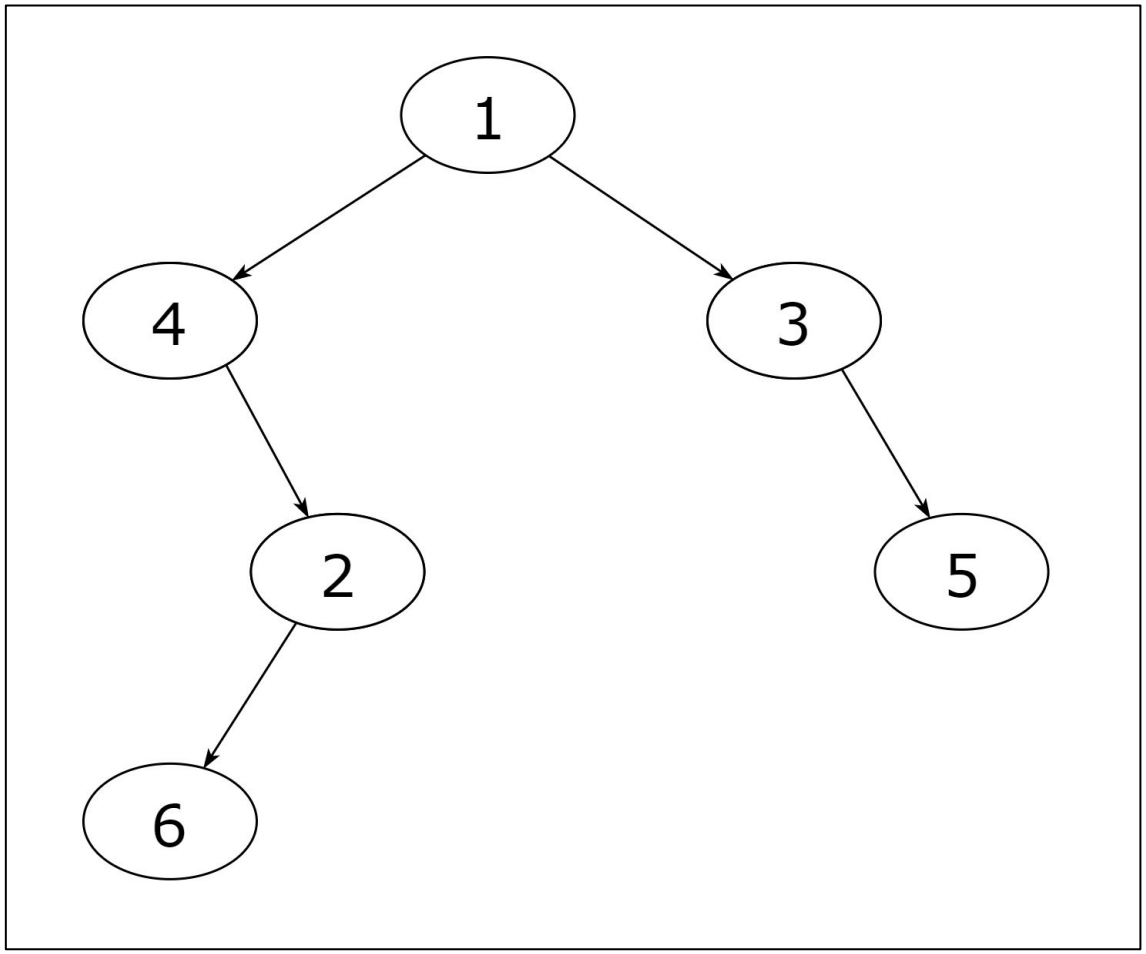


Construct Tree From Given Postorder and Preorder Traversal

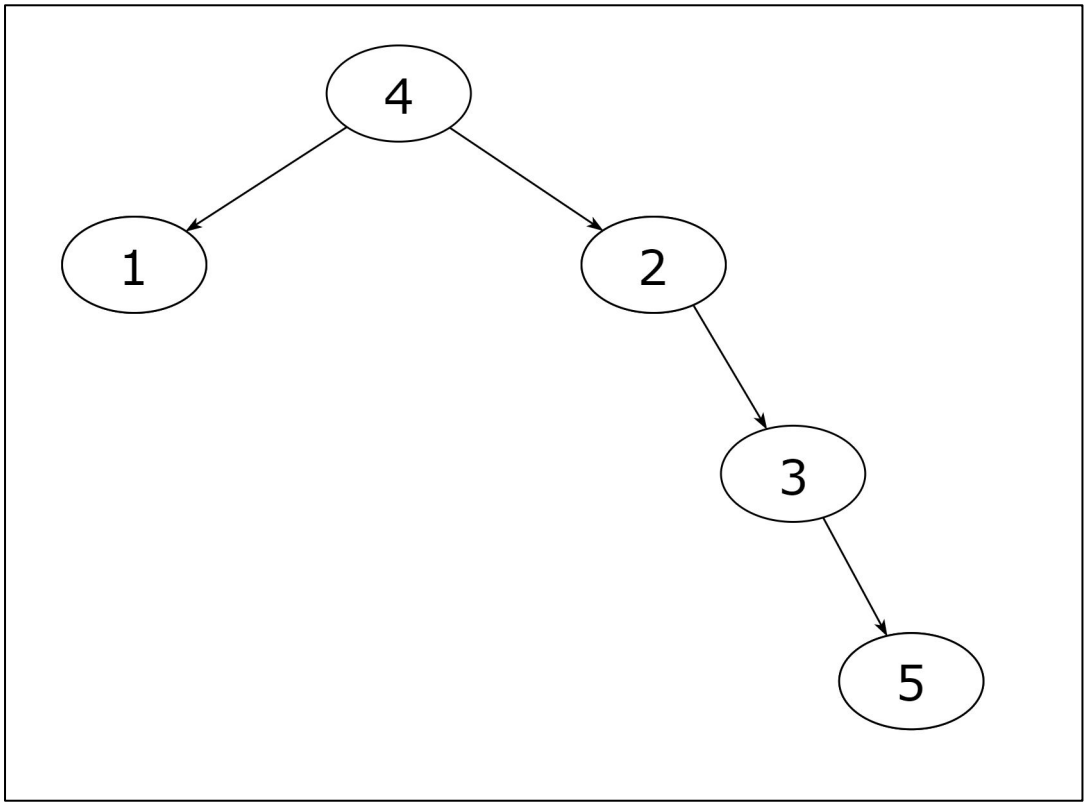
Preorder traversal : { 1, 2, 4, 5, 3, 6, 8, 9, 7 }
Postorder traversal: { 4, 5, 2, 8, 9, 6, 7, 3, 1 }



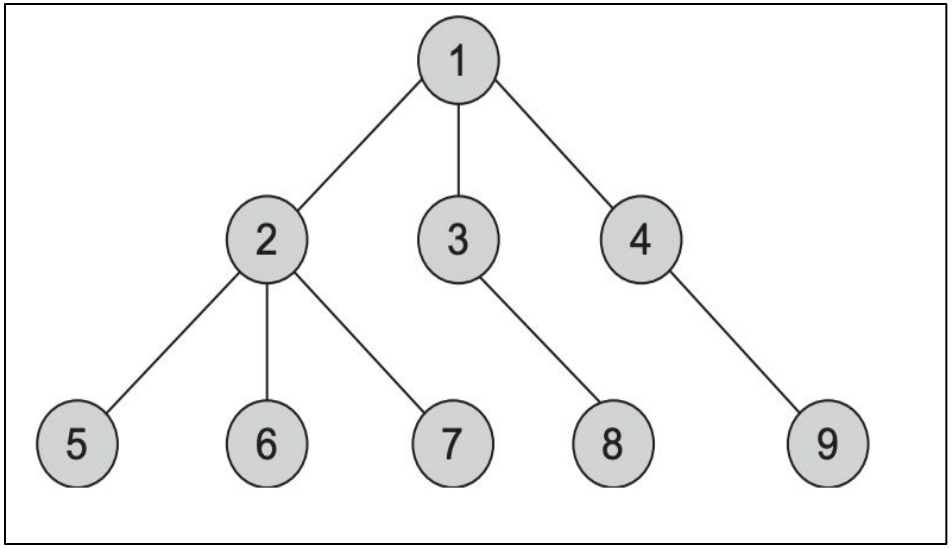
‘POSTORDER’ = [6, 2, 4, 5, 3, 1]
‘PREORDER’ = [1, 4, 2, 6, 3, 5]
A binary tree that matches the given ‘POSTORDER’ and ‘PREORDER’ traversal is:

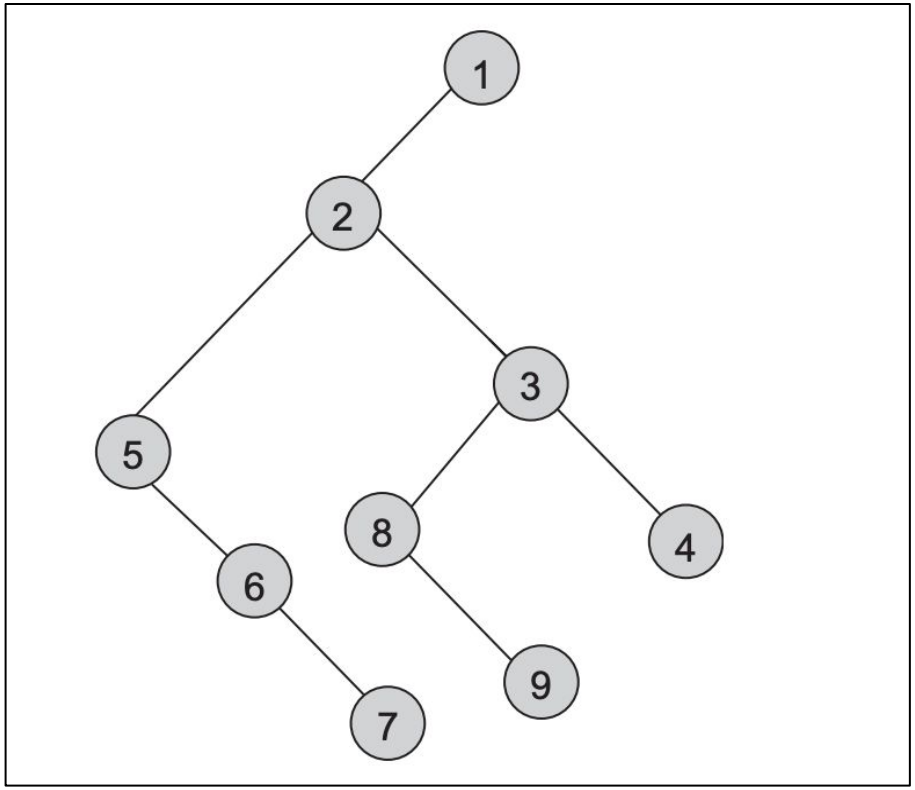


‘POSTORDER’ = [1, 5, 3, 2, 4]
‘PREORDER’ = [4, 1, 2, 3, 5]
A binary tree that matches the given ‘POSTORDER’ and ‘PREORDER’ traversal is:



Example - Convert the following tree in Figure into a binary tree.





Example - Convert the general tree in Figure into its corresponding binary tree.