UNIT – IV : TREES

- Introduction
  - Terminology
  - Representation Of Trees
- Binary Trees
- Binary Tree Traversals
- Additional Binary Tree Operations
- Heaps

INTRODUCTION

In computer science, a tree is a very general and powerful data structure that resembles a real tree.

It consists of an ordered set of linked nodes

in a connected graph, in which each node has at

most one parent node, and zero or more children nodes with a specific order.

Tree can be defined as either the empty

tree, or a node with a list of successor trees.

INTRODUCTION

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

OR

A tree is a connected graph without any circuits.

OR

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

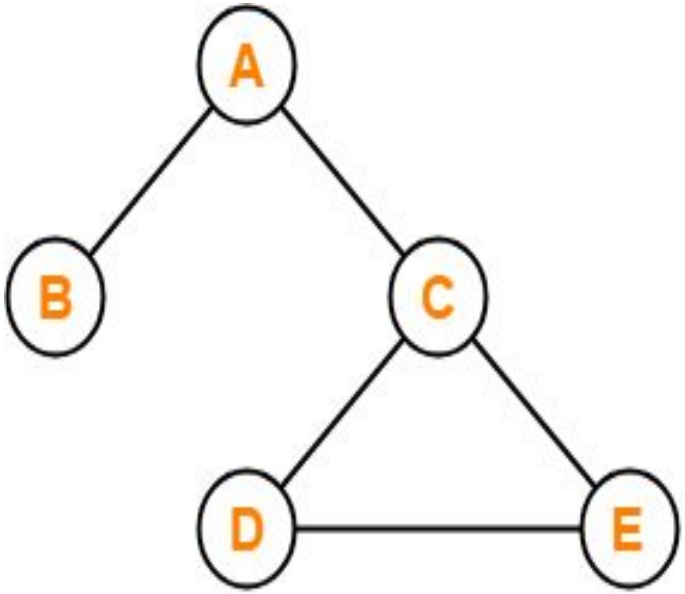
INTRODUCTION

A ***tree*** structure means that the data are organized so that items of information are related by branches.

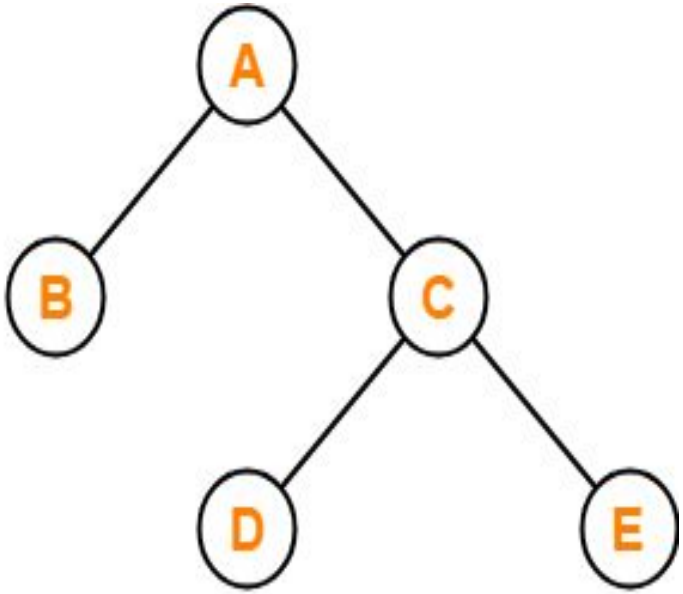
Definition (recursively): A *tree* is a finite set of one or more nodes such that

- There is a specially designated node called *root*.
- The remaining nodes are partitioned into n>=0 disjoint set $T_1,\ldots,T_n$, where each of these sets is a tree. $T_1,\ldots,T_n$ are called the *subtrees* of the

INTRODUCTION



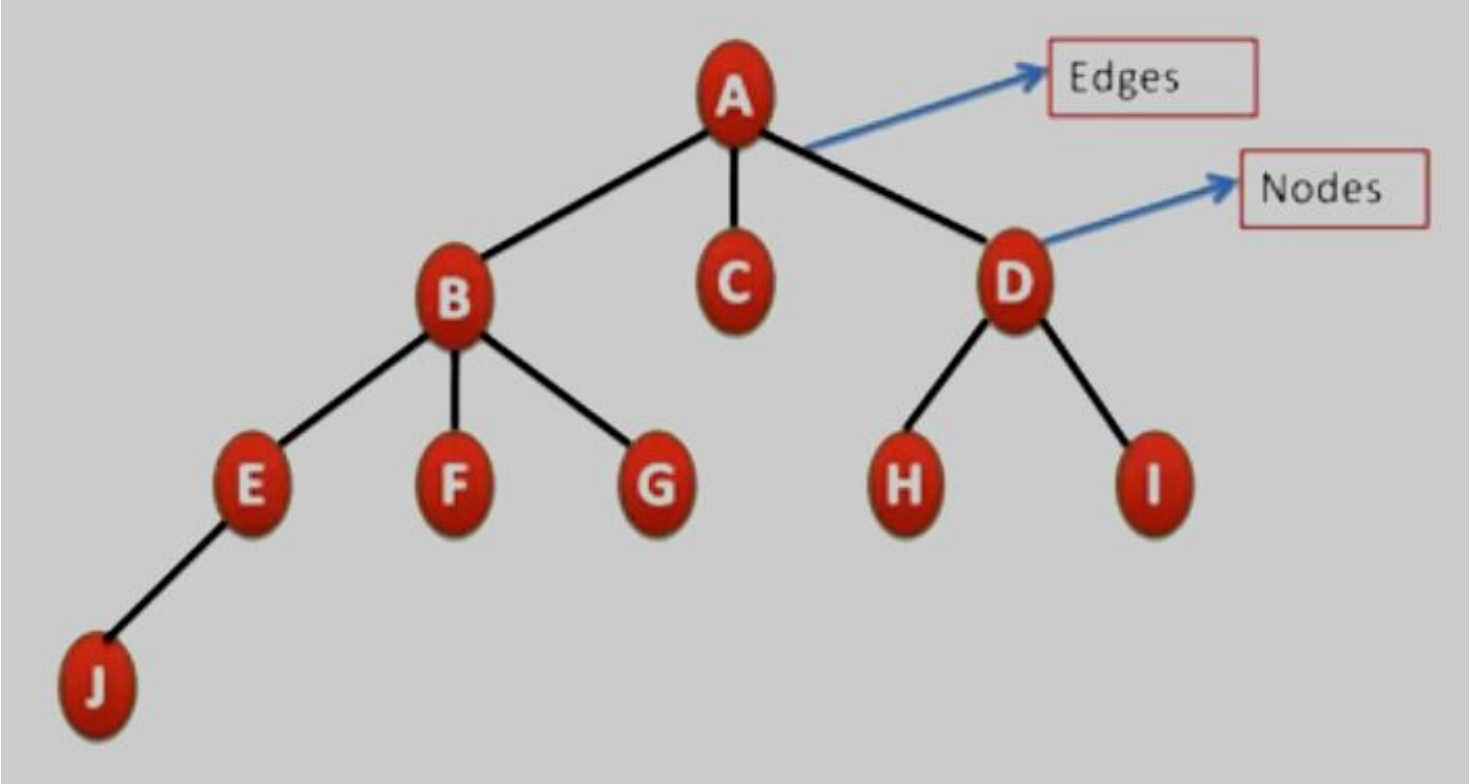
This graph is not a Tree



This graph is a Tree

INTRODUCTION

Tree contains ‘N’ number of nodes and ‘N-1’ number of edges.



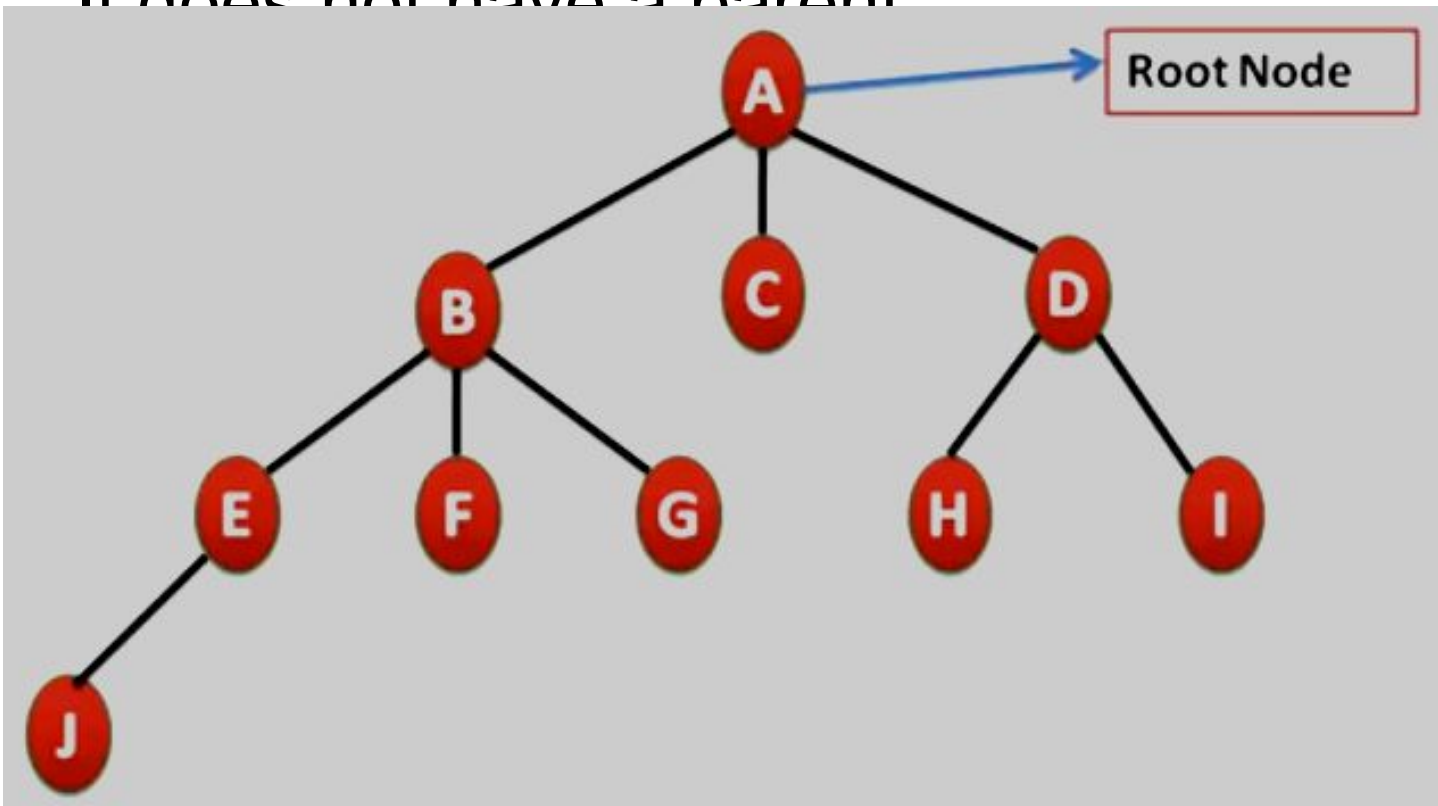
TERMINOLOGY

ROOT: Root is a special node in a tree.

It is first node of the tree.

The entire tree is referenced through it.

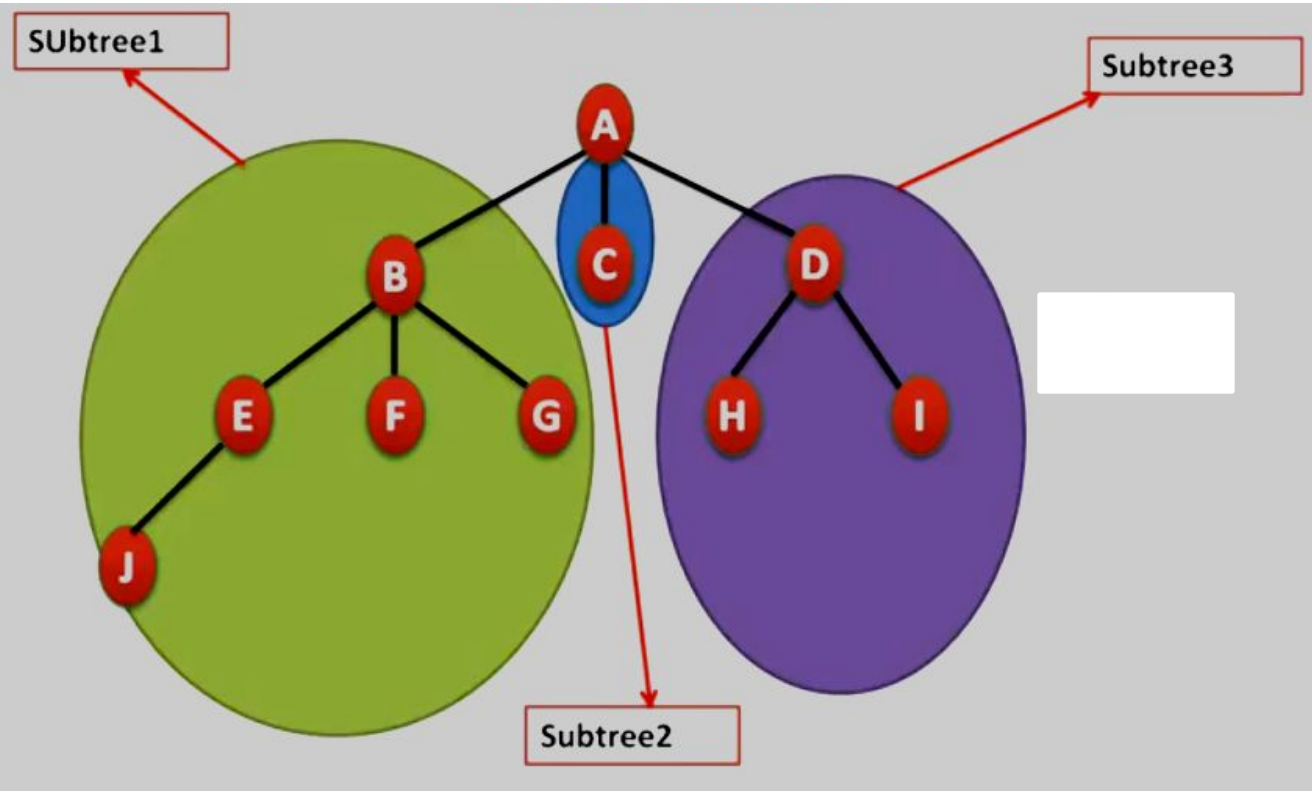
It does not have a parent



TERMINOLOGY

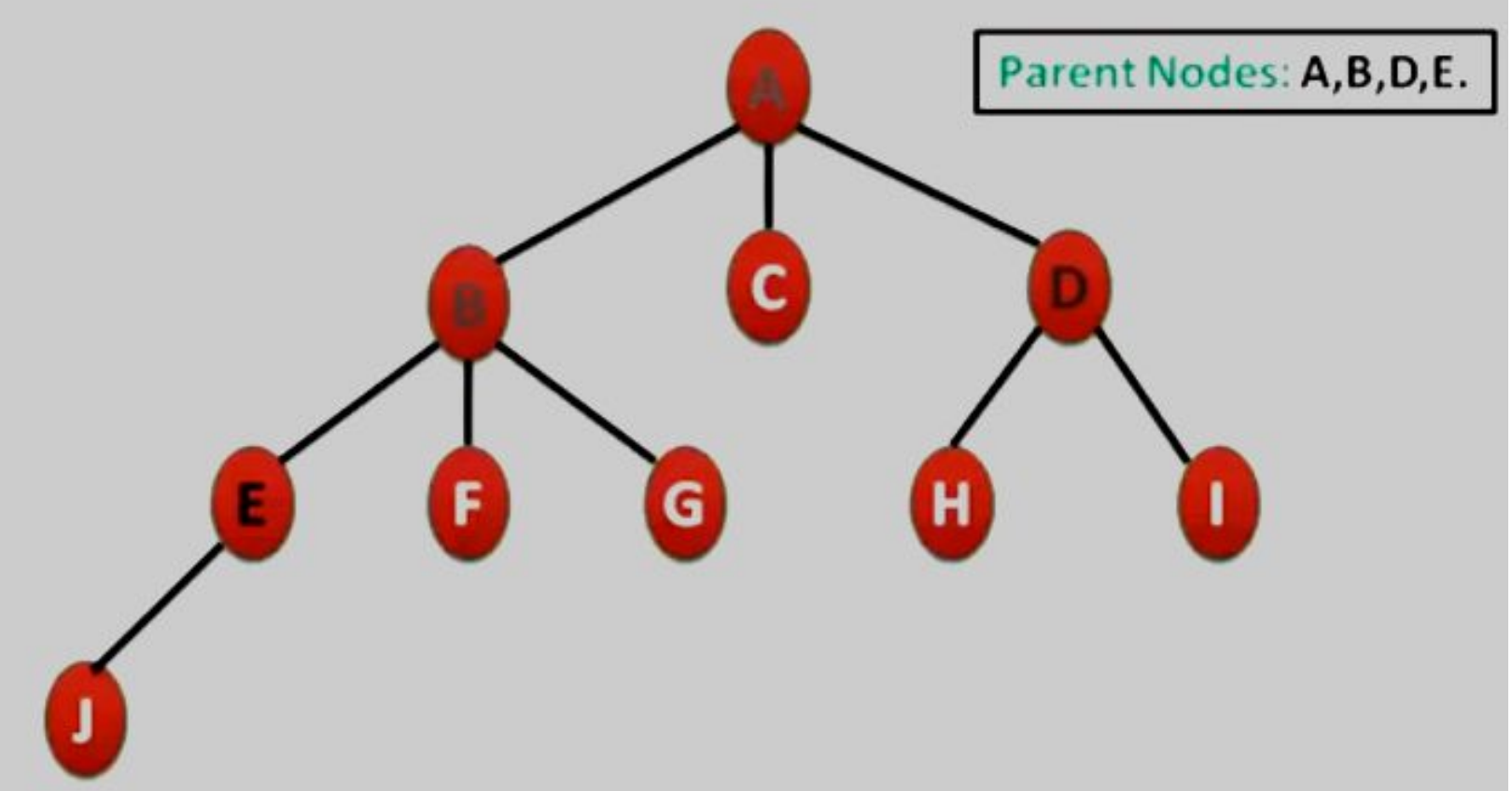
SUB TREE: Every edge from root node / parent node is called sub tree.

Sub tree is also one of the tree, it generates trees.



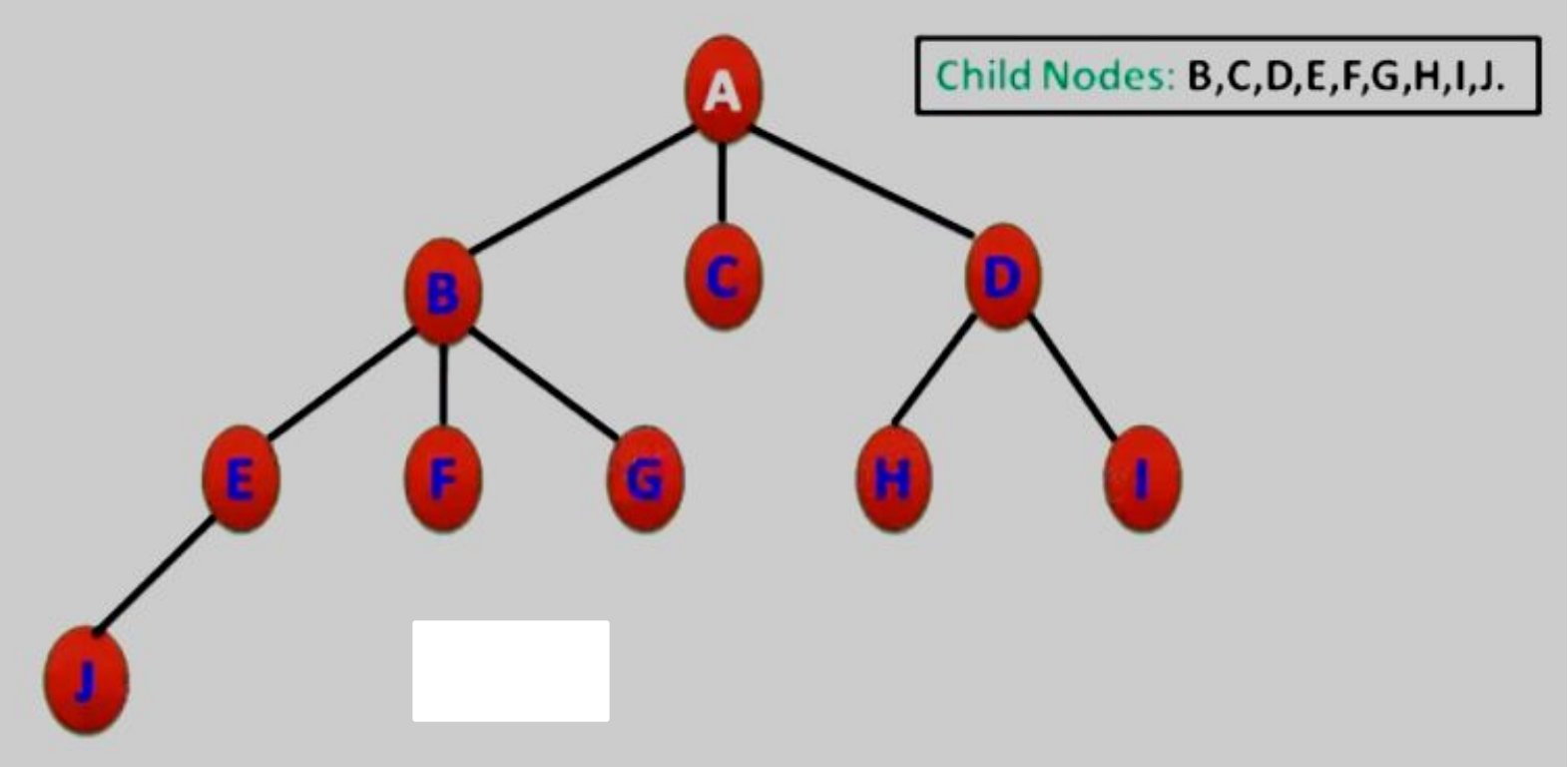
TERMINOLOGY

PARENT NODE: Parent node is an immediate predecessor of a node.



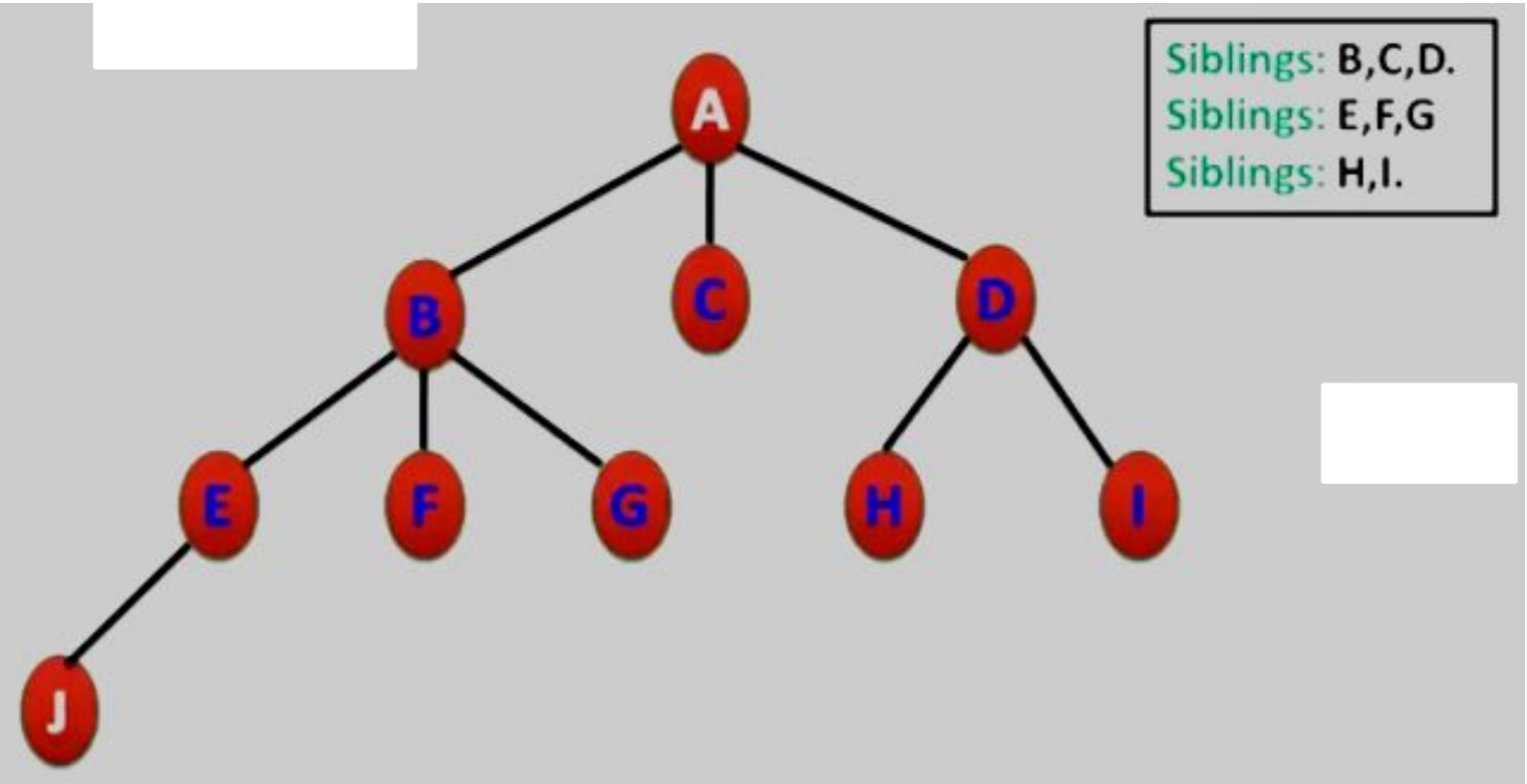
TERMINOLOGY

CHILD NODE: All immediate successors of a node are its children.



TERMINOLOGY

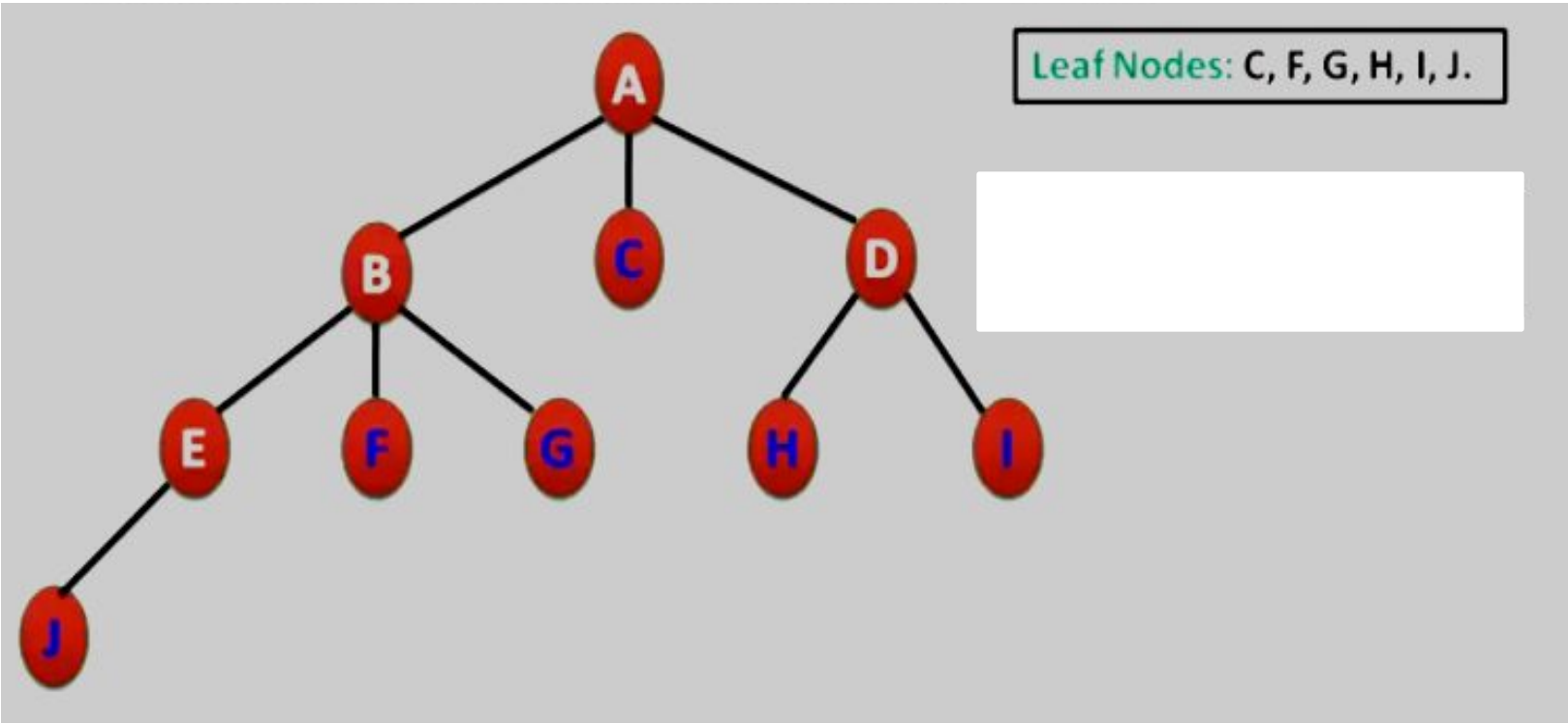
SIBLINGS: Nodes with the same parent are called Siblings.



TERMINOLOGY

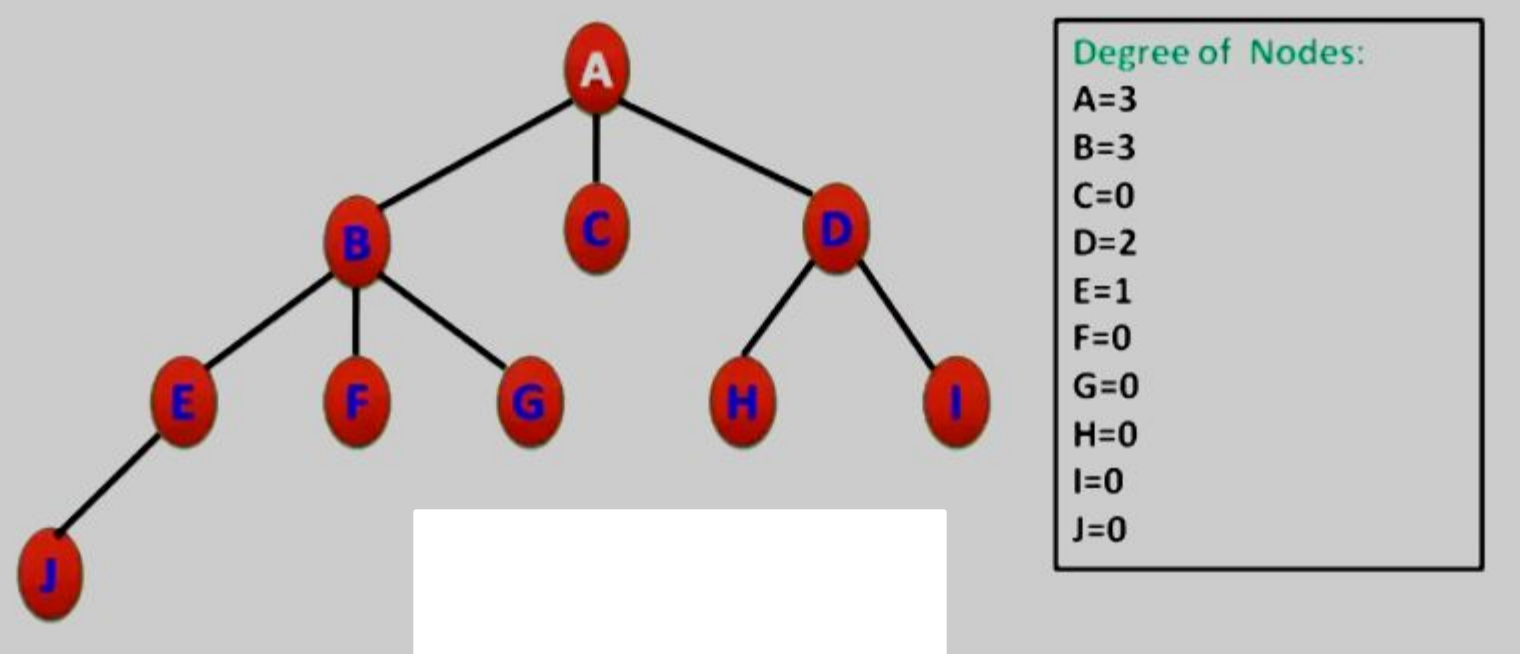
LEAF NODES: The node which does not have a child. A leaf is a node with no child.

- Leaf nodes are also called “External Nodes”.



TERMINOLOGY

DEGREE OF NODE: Degree of a node represents a number of children of a node.

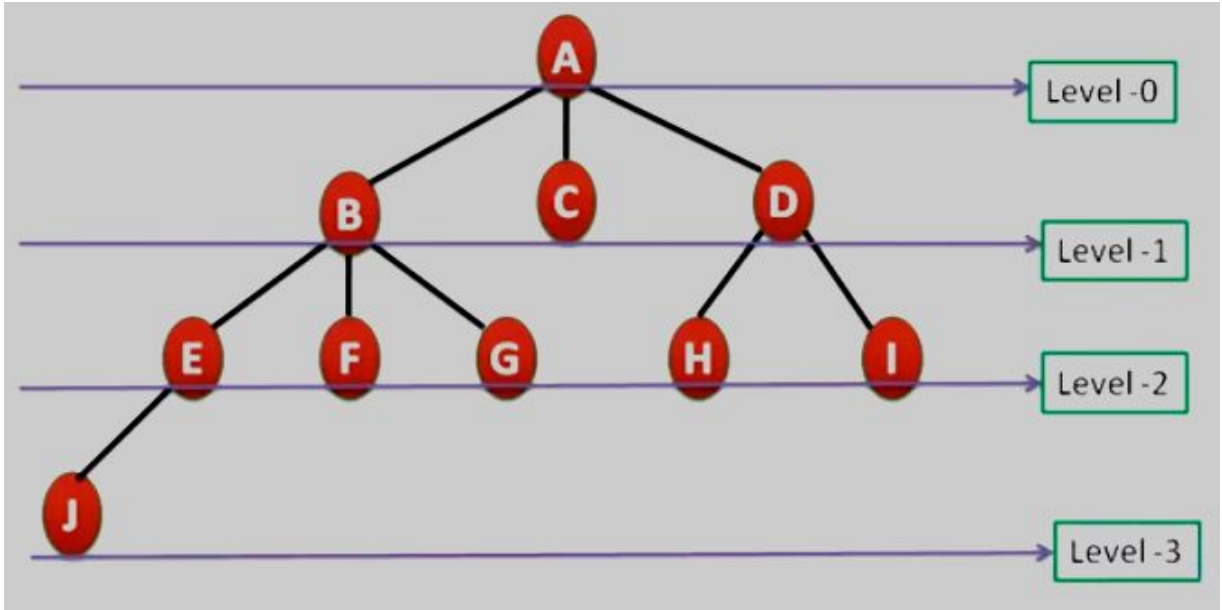


DEGREE OF TREE: Degree of a Tree is the maximum degree of a node.

TERMINOLOGY

LEVEL OF NODE: Levels of a node represents the number of connections between the node and the root.

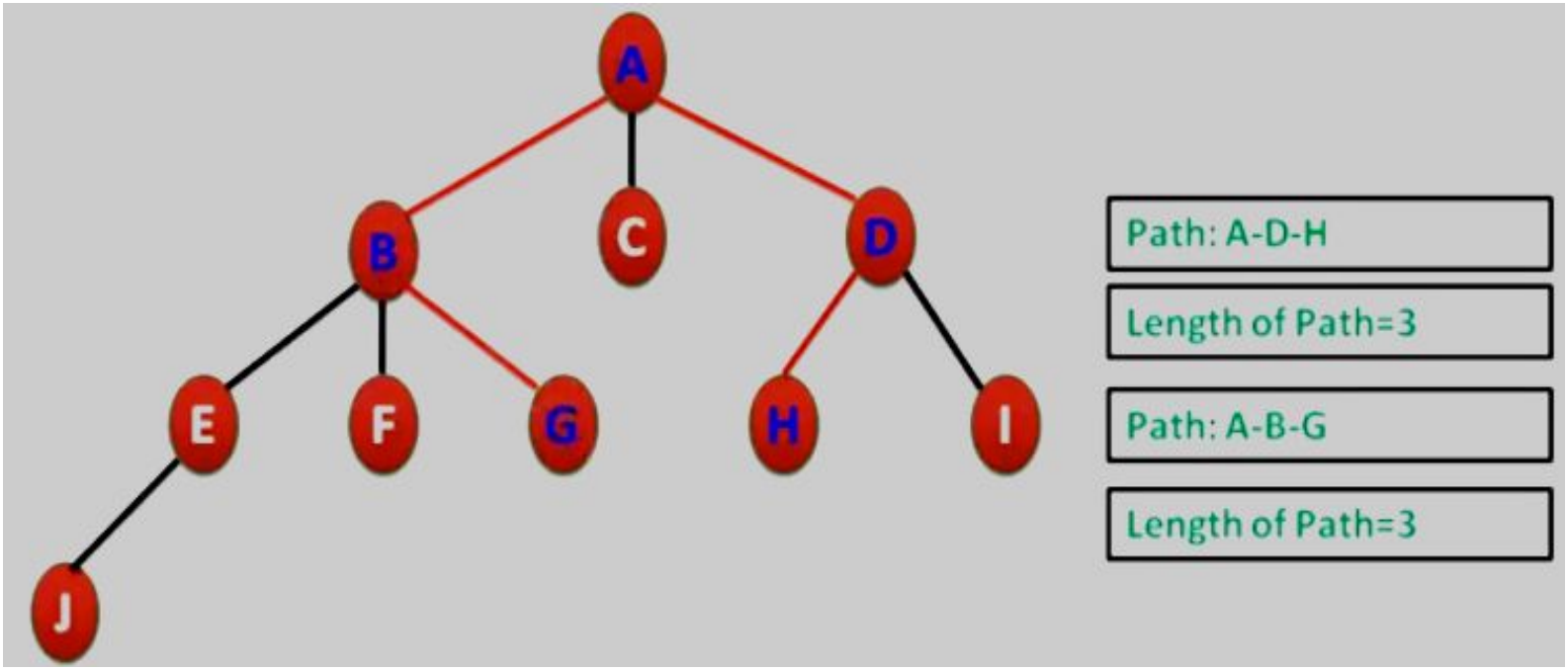
- It represents generation of a node.
- If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on…



TERMINOLOGY

PATH: The sequence of Nodes and Edges from one node to another node.

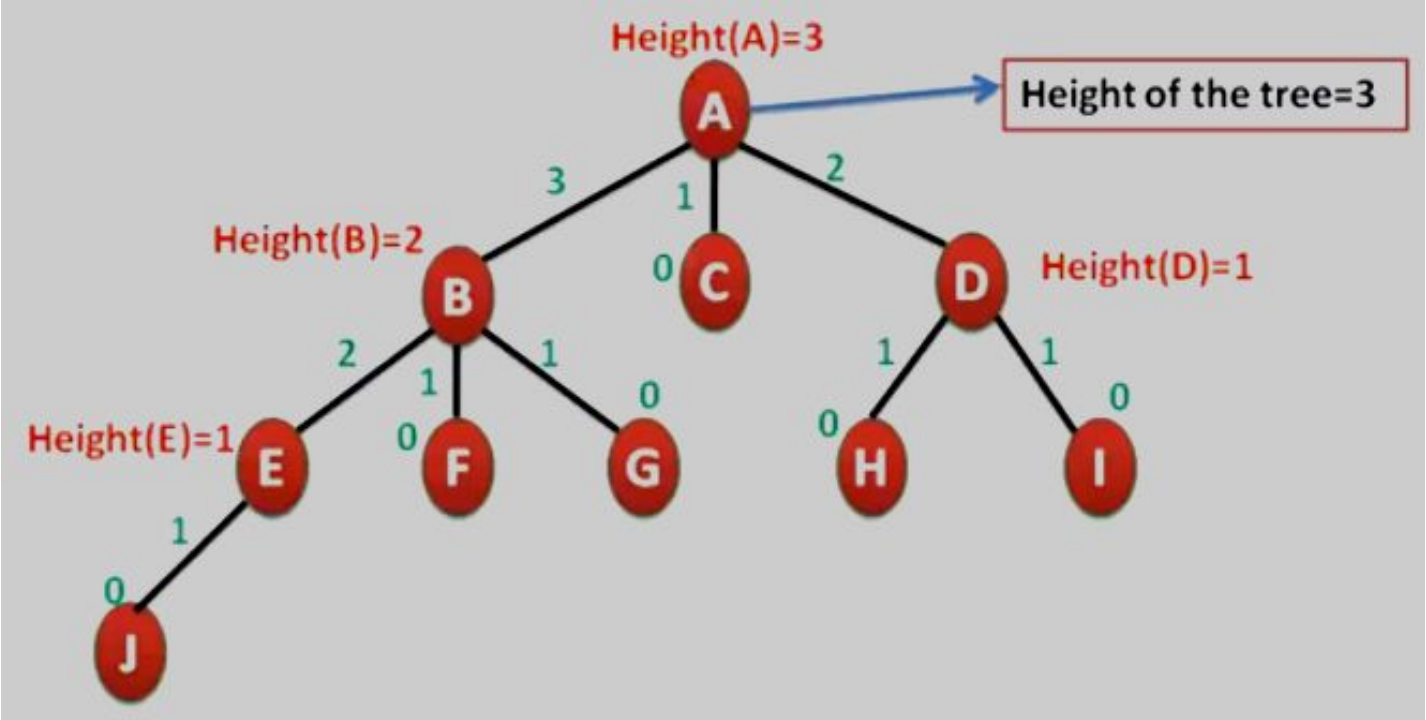
Length of a Path is total number of nodes in that path.



TERMINOLOGY

HEIGH: The total number of edges from leaf node to a particular node in the longest path.

- The height of the root node is equal to be Height of the Tree.
- In a tree, **height of all leaf nodes are "0".**



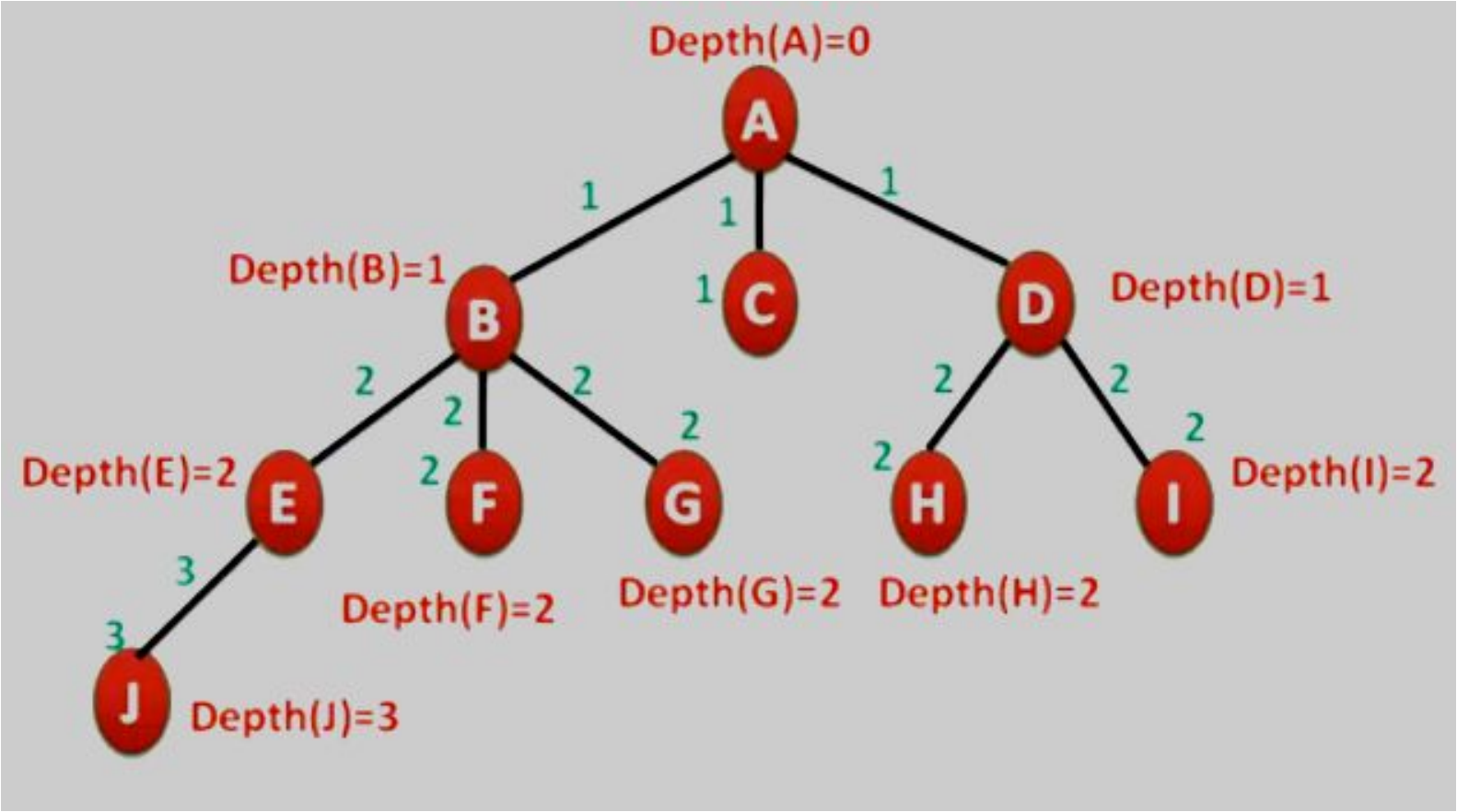
TERMINOLOGY

DEPTH OF THE NODE: The total number of edges from root node to a particular node.

The total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree..**

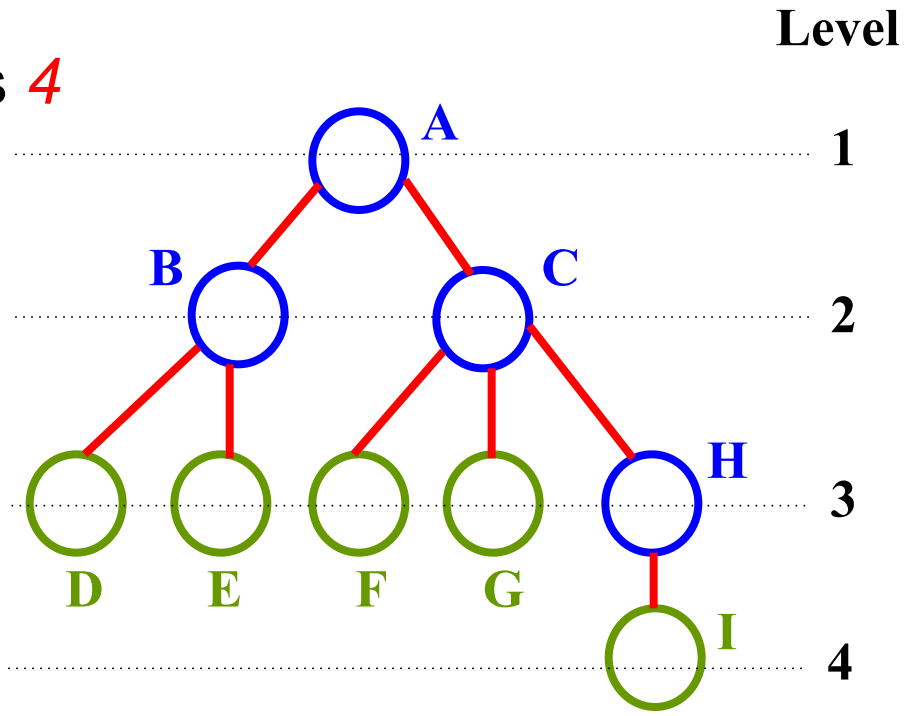
TERMINOLOGY

DEPTH OF THE NODE



TERMINOLOGY

A is the *root* node
B is the *parent* of D and E
C is the *sibling* of B
D and *E* are the *children* of B
D, E, F, G, I are *external nodes*, or *leaves*
A, B, C, H are *internal nodes*
The *level* of *E* is *3*
The *height (depth)* of the tree is *4*
The *degree* of node *B* is *2*
The *degree* of the tree is *3*
The *ancestors* of node *I* is *A, C, H*
The *descendants* of node *C* is *F, G, H, I*



TYPES OF TREES

The types of trees in a data structures are:

1. General Tree
2. Binary Tree
3. M-Way Search Tree
4. 2-3 Tree
5. B Tree
6. B+ Tree

Binary Search Tree.
Heap.
AVL Tree
Red-Black Tree
Splay Tree

ADVANTAGES OF TREE

- The tree reflects the data structural connections.
- The tree is used for hierarchy.
- It offers an efficient search and insertion procedure.
- The trees are flexible.
- This allows subtrees to be relocated with minimal effort.

APPLICATIONS OF TREE

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

BINARY TREE

A binary tree is either empty or consists of

a) a node called the root

b) left and right sub trees are themselves binary trees.

(OR)

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.

(OR)

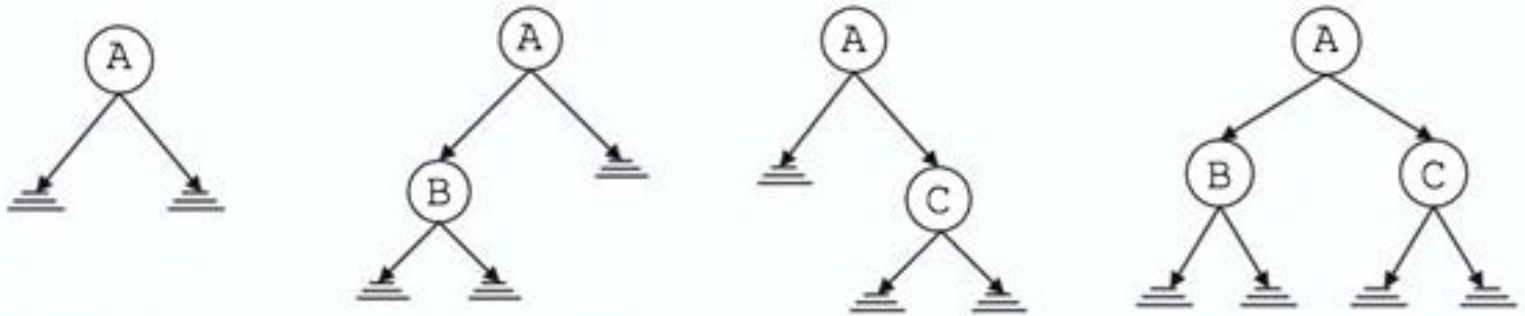
In binary tree each node will have one data field and two pointer fields for representing the sub-branches. The degree of each node in the binary tree will be at the most two.

BINARY TREE

Binary tree is a tree in which each node has at most two children, a left child and a right child.

(OR)

A Binary tree, is a tree in which no node can have more than two children



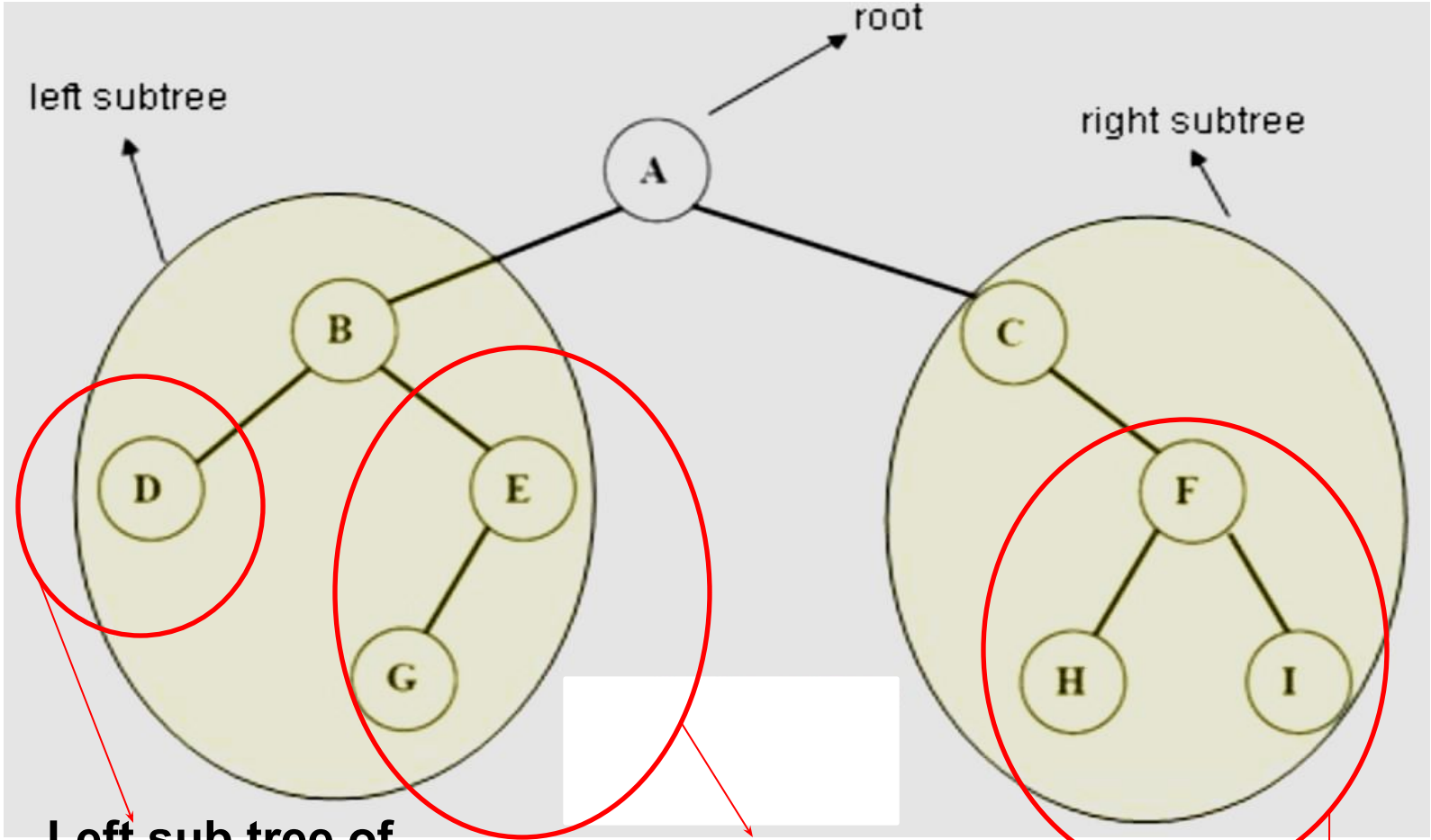
(a) Binary tree with one node

(b) Binary tree with two nodes

(c) Binary tree with two nodes

(d) Binary tree with three nodes

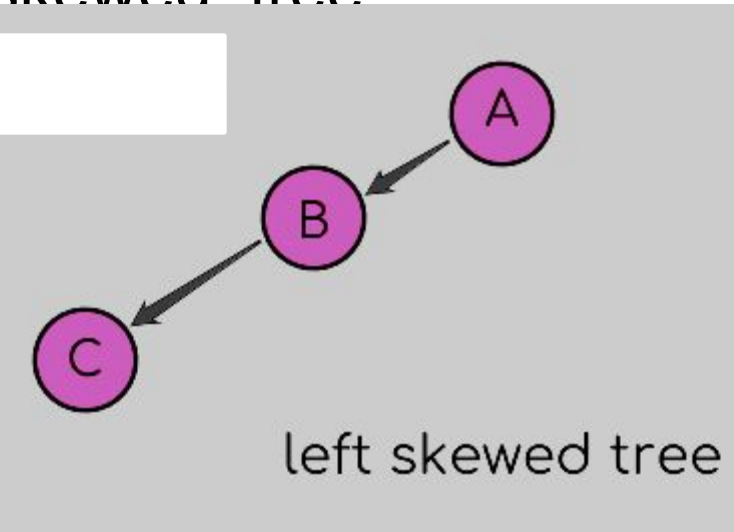
BINARY TREE


root
left subtree
right subtree
A
B
C
D
E
F
G
H
I
Left sub tree of B
Right sub tree of B
Right sub tree of C

TYPES OF BINARY TREES

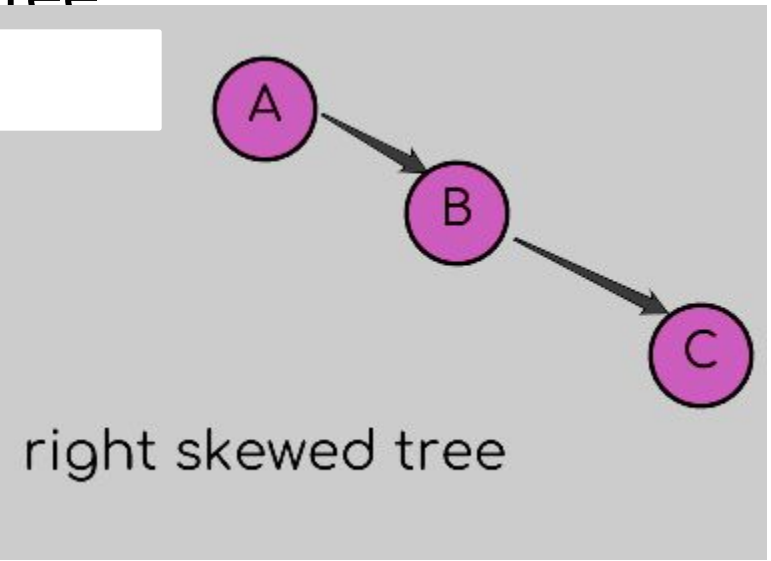
***1. Left skewed binary tree*:**

If the right sub - tree is missing in every node of a tree we call it as left skewed tree



***2. Right skewed binary tree*:**

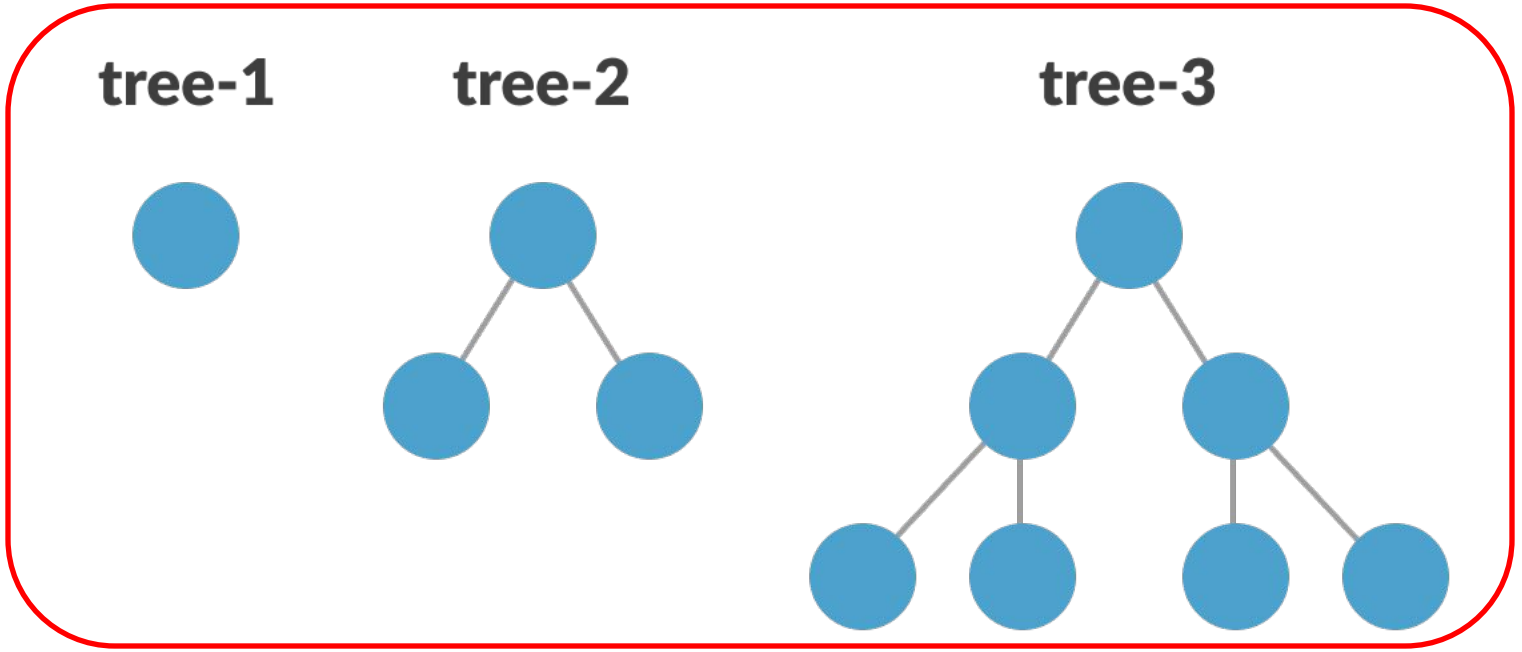
If the left sub – tree is missing in every node of a tree we call it is right sub - tree



TYPES OF BINARY TREES

***3. Perfect binary tree*:**

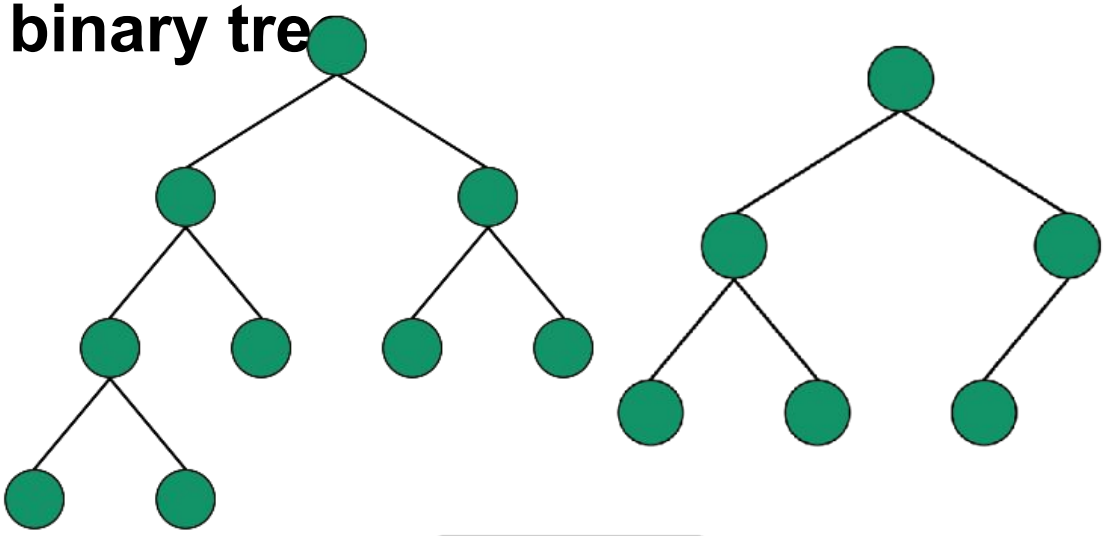
A perfect binary tree is a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.



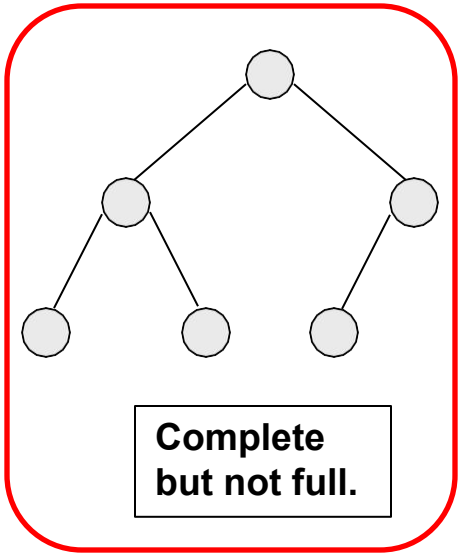
TYPES OF BINARY TREES

***4. Complete binary tree*:**

A **complete binary tree** is a **binary tree** in which all the levels are completely filled except possibly the lowest one, which is filled from the left. All the leaf elements must lean towards the left. The last leaf element might not have a right sibling i.e. a **complete binary tree**



Complete



COMPLETE BINARY TREE

Complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly *2l nodes at each level l, where l is from 0 to d.*

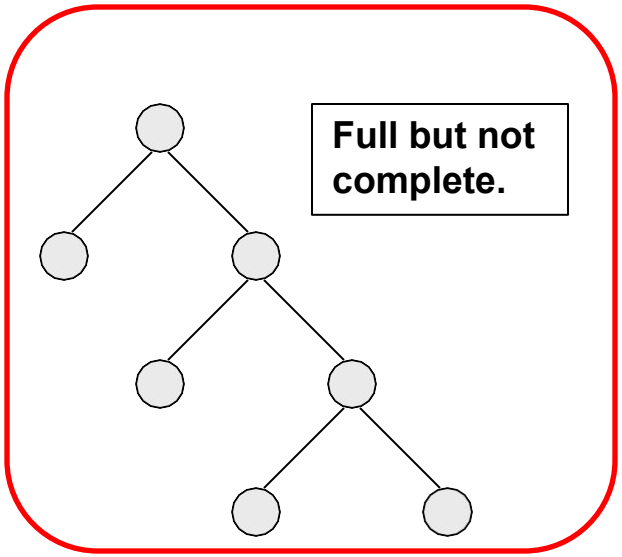
Note:

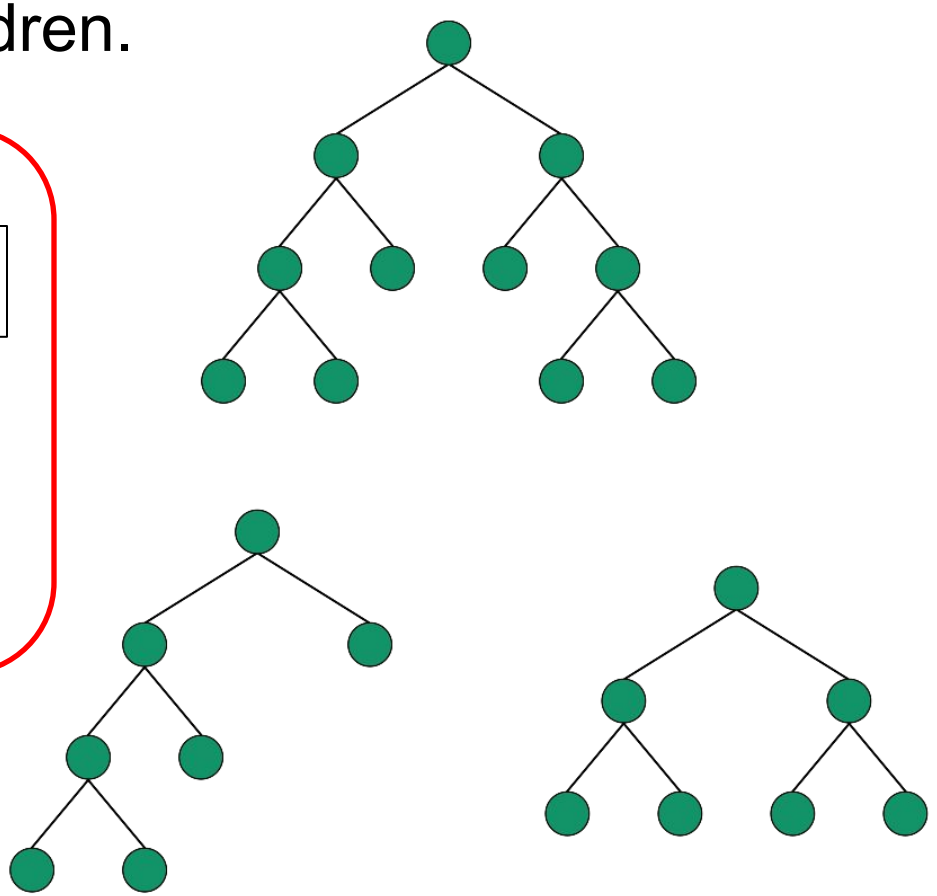
1. A binary tree of depth n will have maximum 2n -1 nodes.
2. A complete binary tree of level *l will have maximum 2l nodes at each level, where l starts* from 0.
3. Any binary tree with n nodes will have at the most n+1 null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

TYPES OF BINARY TREES

***5. Full binary tree*:**

Full Binary Tree is a Binary Tree in which every node has 0 or 2 children.

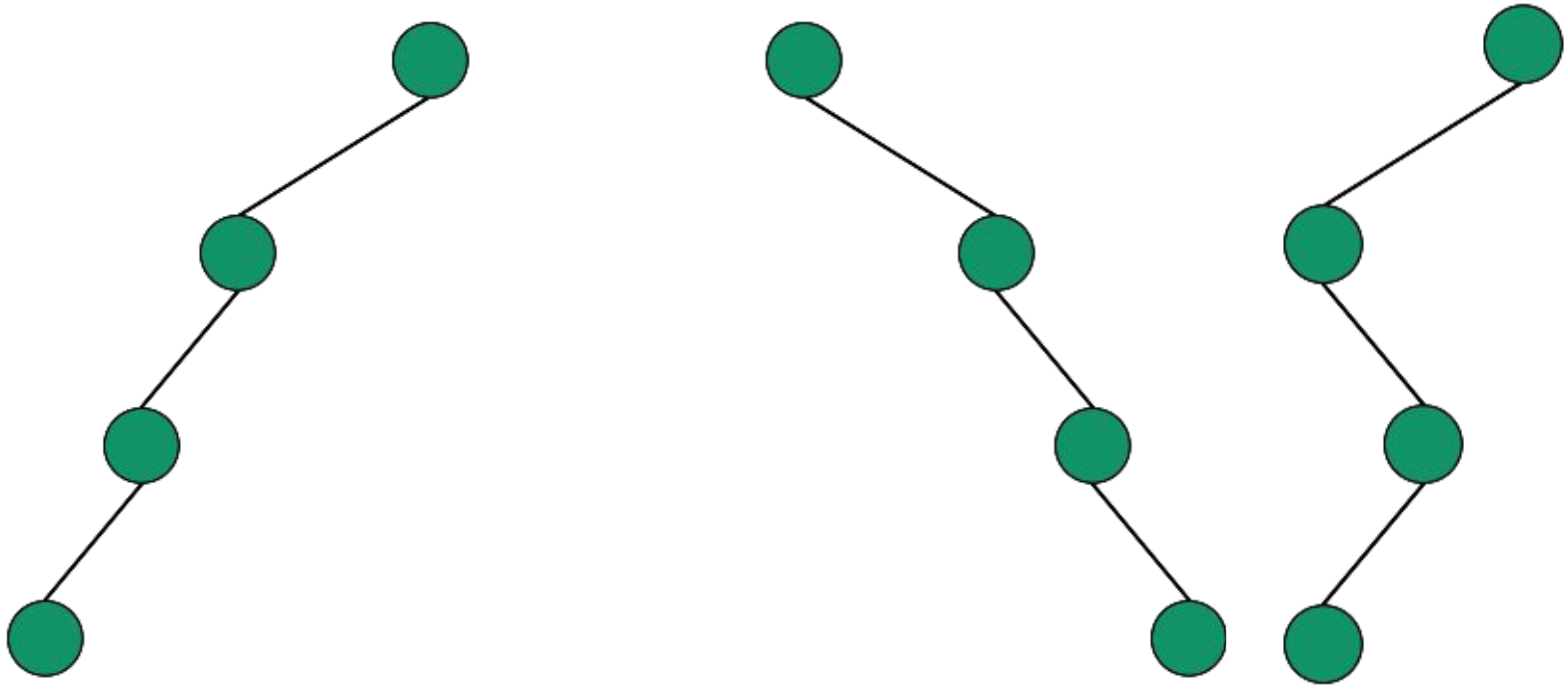




TYPES OF BINARY TREES

6. Degenerate Binary Tree :

Degenerate Binary Tree is a Binary Tree where every parent node has only one child node.



BINARY TREE REPRESENTATION

A binary tree can be represented mainly in 2 ways:

a) Sequential Representation

b) Linked Representation

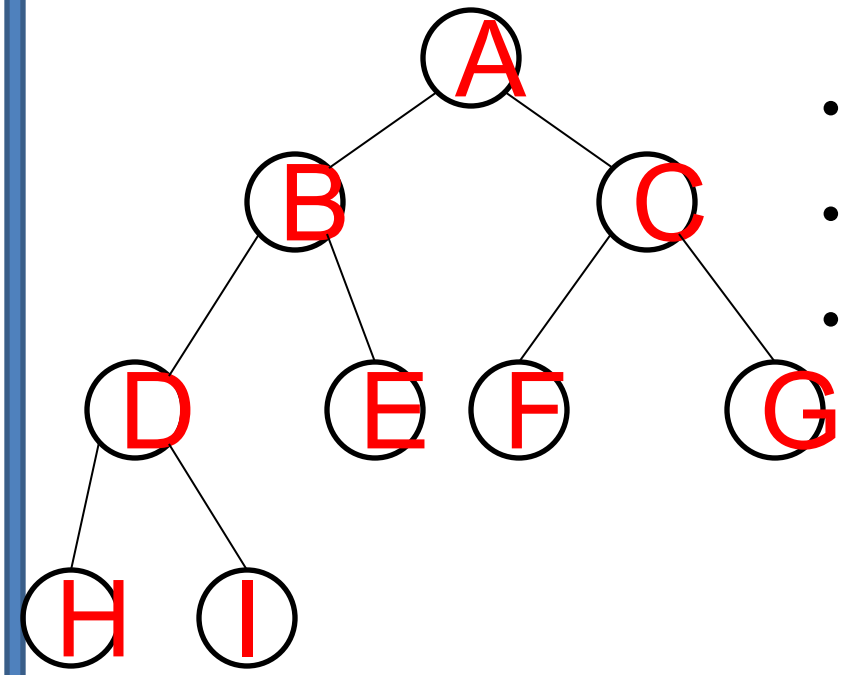
a) Sequential Representation

The simplest way to represent binary trees in memory is the sequential representation that uses one - dimensional array.

1) The root of binary tree is stored in the 1^{st} location of array.

2) If a node is in the $\mathbf{i}^{th}$ location of array, then its left child is in the location 2**i**+1 and its right child in the location 2**i**+2 The maximum size that is required for an array to store a tree is $2^{d+1}-1$, where d is the depth of the tree.

SEQUENTIAL REPRESENTATION

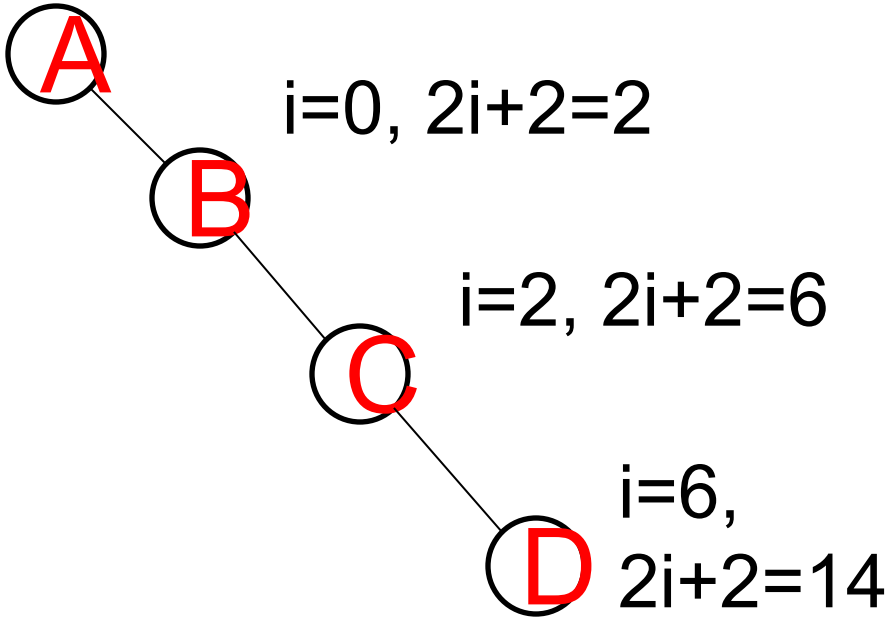


If node is at i^{th} index, then

- Left child would be at **[2i+1]**
- Right child would be at **[2i+2]**
- Parent node would be at **[(i-1)/2**

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

SEQUENTIAL REPRESENTATION



A	-	B	-	-	-	C	-	-	-	-	-	-	-	D
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

SEQUENTIAL REPRESENTATION

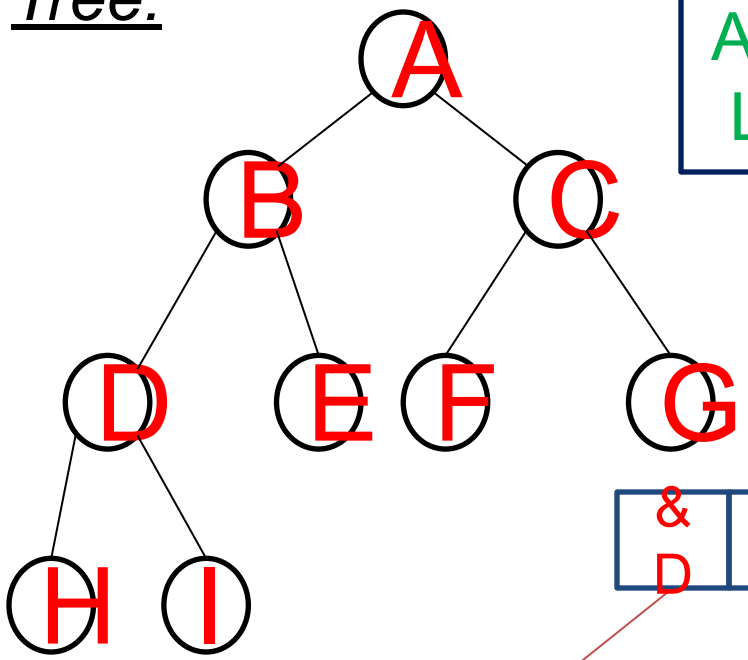
Advantages : The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular Node is fast because of the random access.

Disadvantages:

1. The major disadvantage with this type of representation
Is wastage of memory. For example in the skewed tree half of the array is unutilized.
2. In this type of representation the maximum depth of the tree has to be fixed.
3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary

LINKED REPRESENTATION

Double Linked list for representing each node in Binary Tree:



LINKED REPRESENTATION

Node Declaration and Initialization:

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

```
struct node* newNode(int data)
{
    struct node* node
        = (struct node*)malloc(sizeof(struct
node));
    node->data = data;
    node->left = NULL;
    node->right = NULL;

    return (node);
}
```

LINKED REPRESENTATION

Advantages :

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.
2. Insertions and deletions which are the most common operations can be done without moving the nodes.

Disadvantages :

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each

BINARY TREE OPERATION

The basic operations are,

1. Creation Operation
2. Insertion Operation
3. Deletion Operation
4. Traversal
5. Display.

BINARY TREE TRAVERSALS

Traversing a tree means that processing it so that each node is visited exactly once.

A binary tree can be traversed a number of ways.

The Most common tree traversals are

- In – order,
- Pre – order and
- Post – order.

BINARY TREE TRAVERSALS

Algorithms:

Pre-order:

RLR

1.Visit the root. Root | Left | Right 2.Traverse the left sub tree in pre-order.
3.Traverse the right sub tree in pre-order.

In-order:

LRR

1.Traverse the left sub tree in in-order. Left | Root | Right 2.Visit the root.
3.Traverse the right sub tree in in-order.

Post-order:

1.Traverse the left sub tree in post-order.
2.Traverse the right sub tree in post-order. LRR
3.Visit the root. Left | Right |

BINARY TREE TRAVERSALS

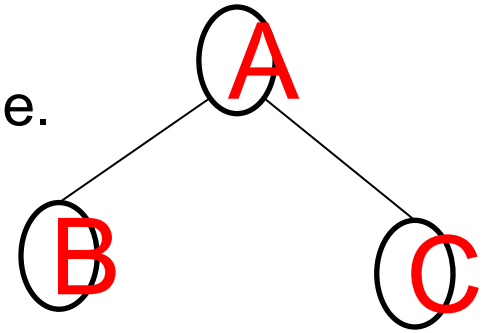
In-order: [Left | Root | Right]. ABC

Until all nodes are traversed −

Step 1 − Recursively traverse left subtree.

Step 2 − Visit root node.

Step 3 − Recursively traverse right subtree.



Pre-order: [Root | Left | Right] BAC

Until all nodes are traversed −

Step 1 − Visit root node.

Step 2 − Recursively traverse left subtree.

Step 3 − Recursively traverse right subtree.

Post-order: [Left | Right | Root] ABC

Until all nodes are traversed −

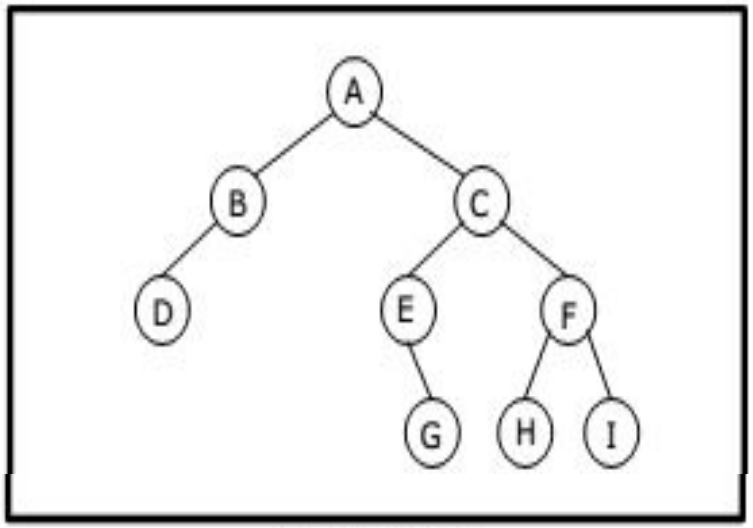
Step 1 − Recursively traverse left subtree.

Step 2 − Recursively traverse right subtree.

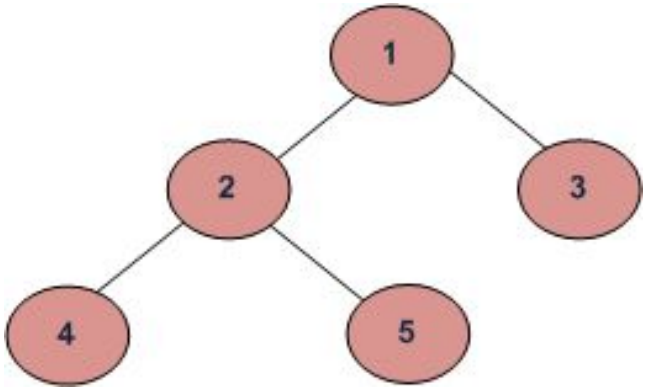
Step 3 − Visit root node.

BINARY TREE TRAVERSALS

Example s:



- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I



(a) Inorder : 4 2 5 1 3
(b) Preorder : 1 2 4 5 3
(c) Postorder : 4 5 2 3 1

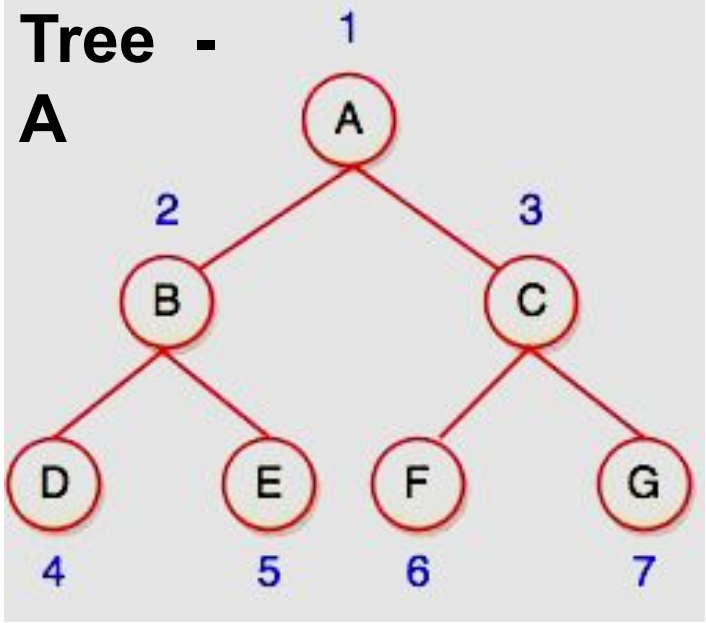
BINARY TREE TRAVERSALS

```
void postOrder(struct node* node)
{
    if (node == NULL)
        return;
      postOrder (node->left);  // recur. on left subtree
      postOrder (node->right);  // recur on right subtree
      printf("%d ", node->data); // now deal with the node
}
 void inOrder(struct node* node)
{
    if (node == NULL)
        return;
     inOrder (node->left); /* first recur on left subtree */
     printf("%d ", node->data); /* then print the data of node */
     inOrder (node->right); /* now recur on right subtree */
}
 void preOrder(struct node* node)
{
    if (node == NULL)
        return;
       printf("%d ", node->data);     /* first print data of node */
       printPreorder(node->left);     /* then recur on left subtree */
       printPreorder(node->right); /* now recur on right subtree */
}
```

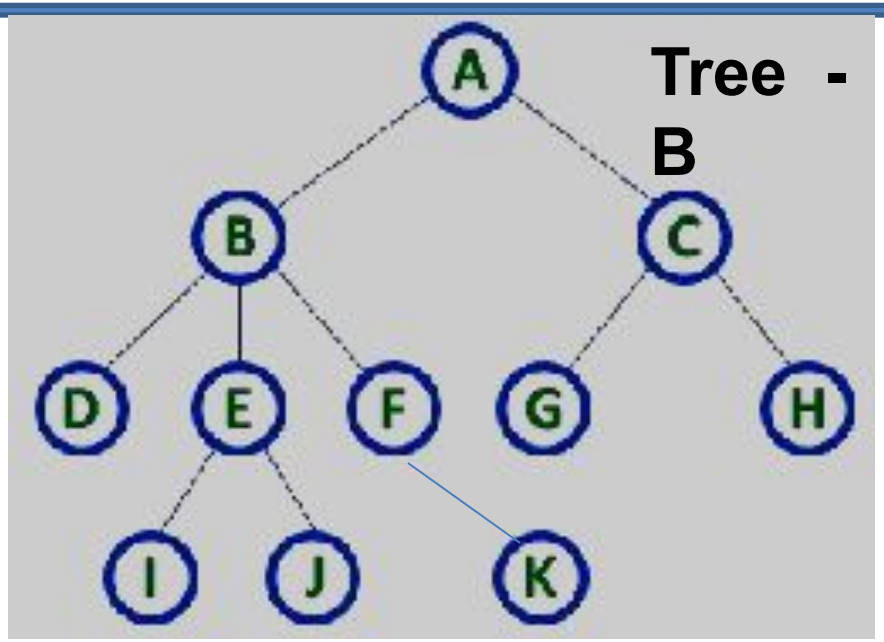
BINARY TREE TRAVERSALS

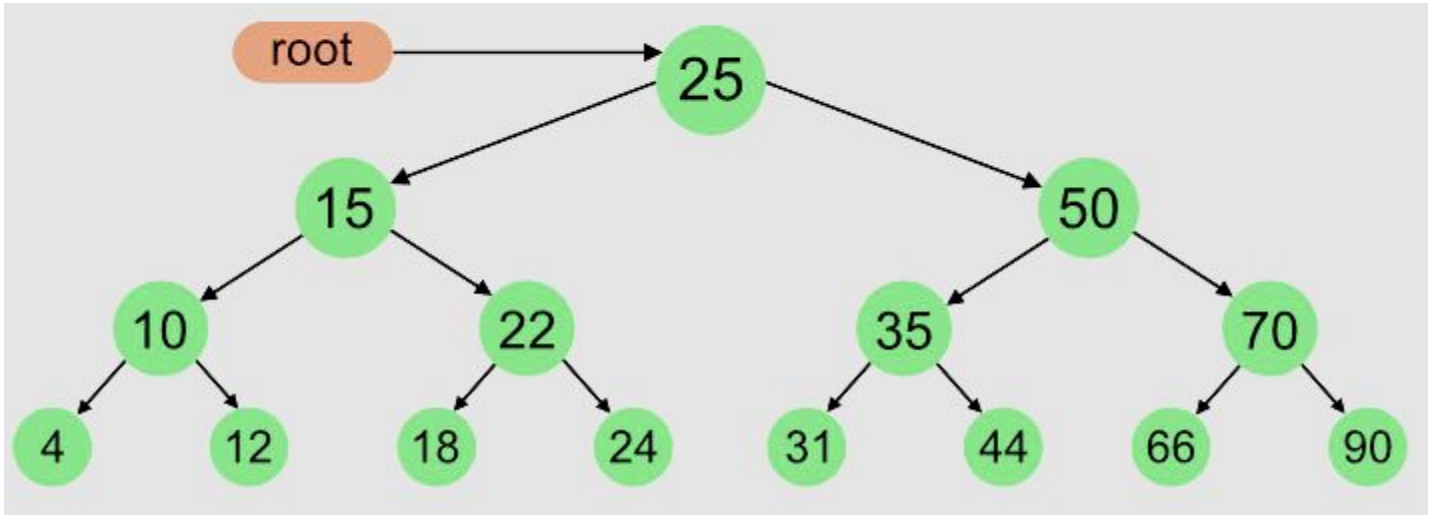
Examples Practice :

Tree - A



Tree - B





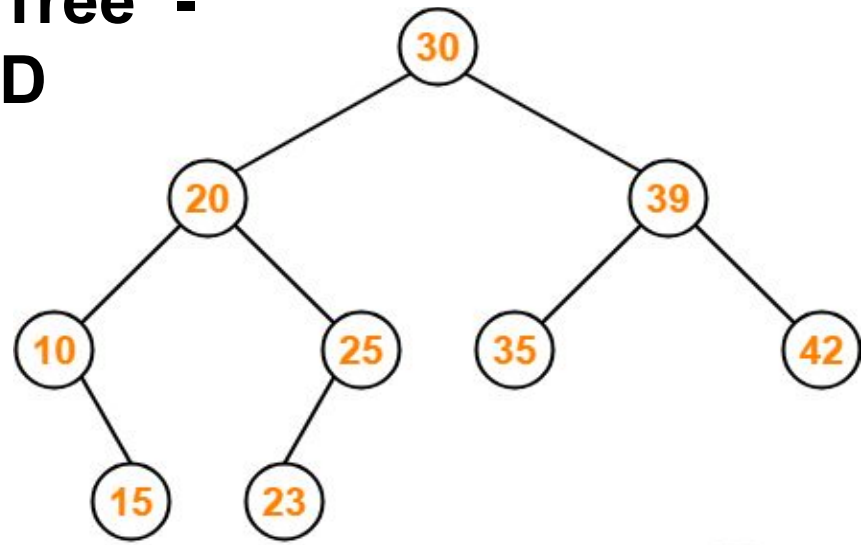
TREE -C

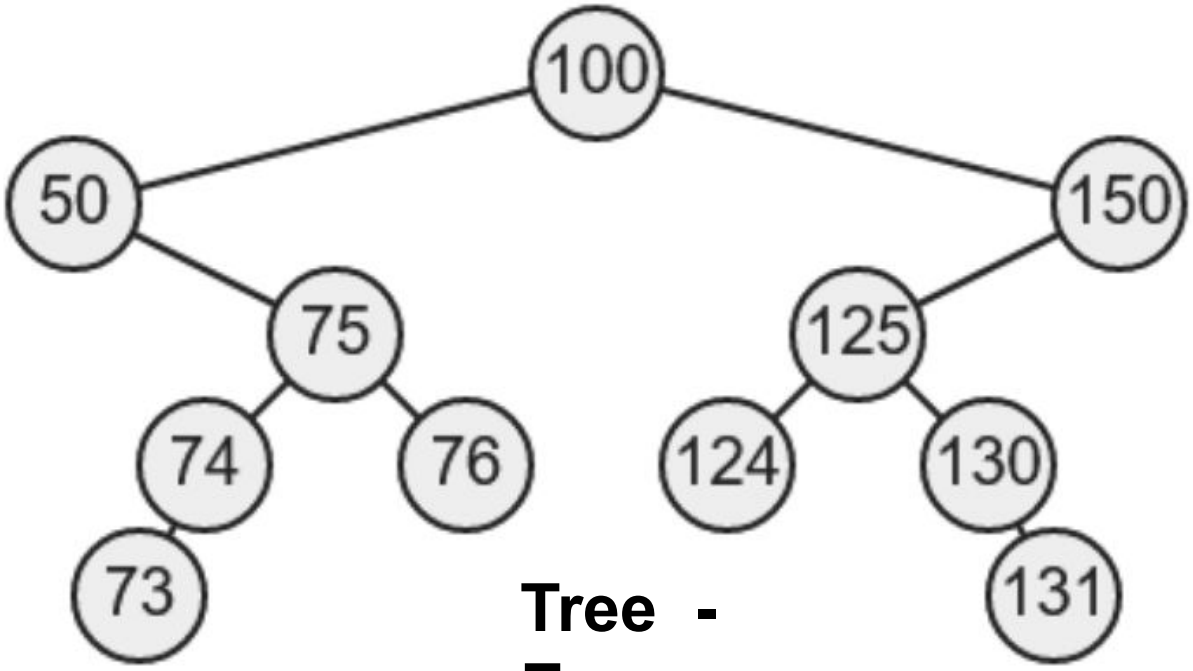
BINARY TREE TRAVERSALS

Dr. APJ ABDUL KALAM IIIT ONGOLE RGUKT AP

Examples Practice :

Tree - D



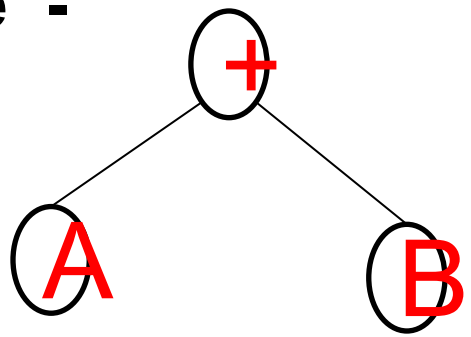


Tree - E

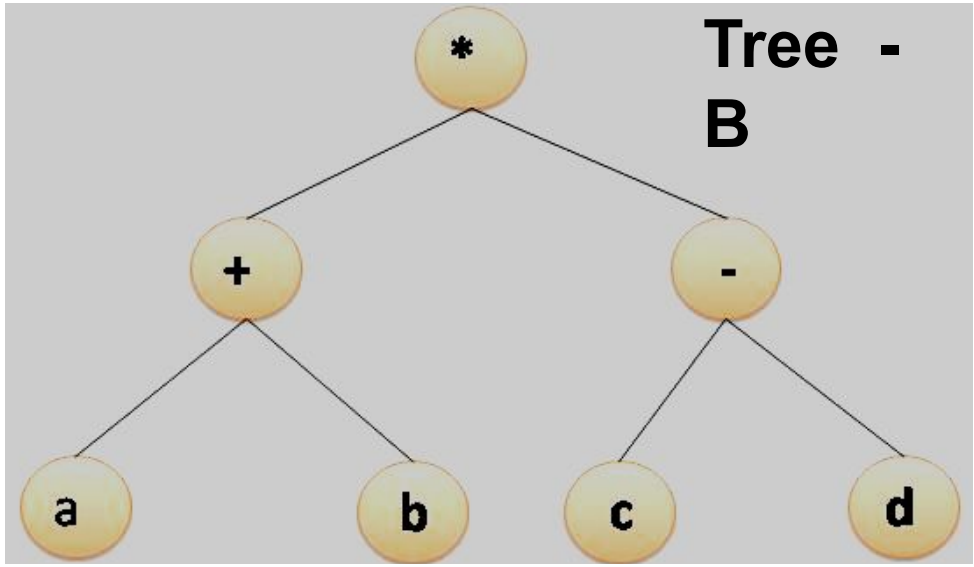
BINARY TREE TRAVERSALS

Examples Practice :

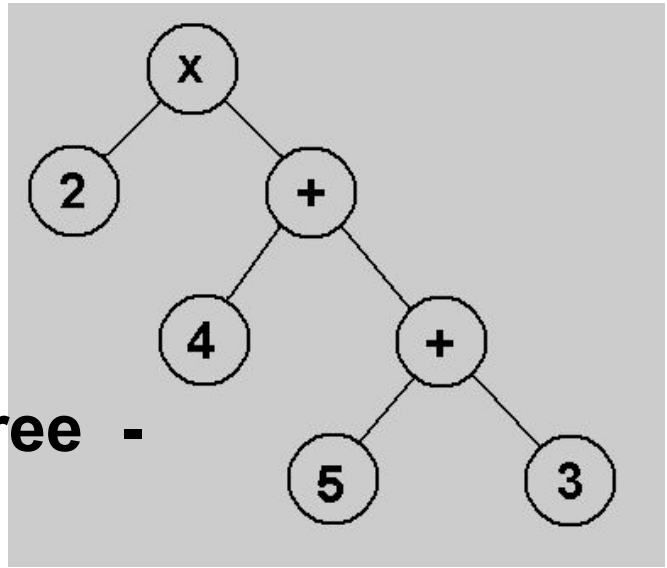
Tree - A



Tree - B



Tree - C

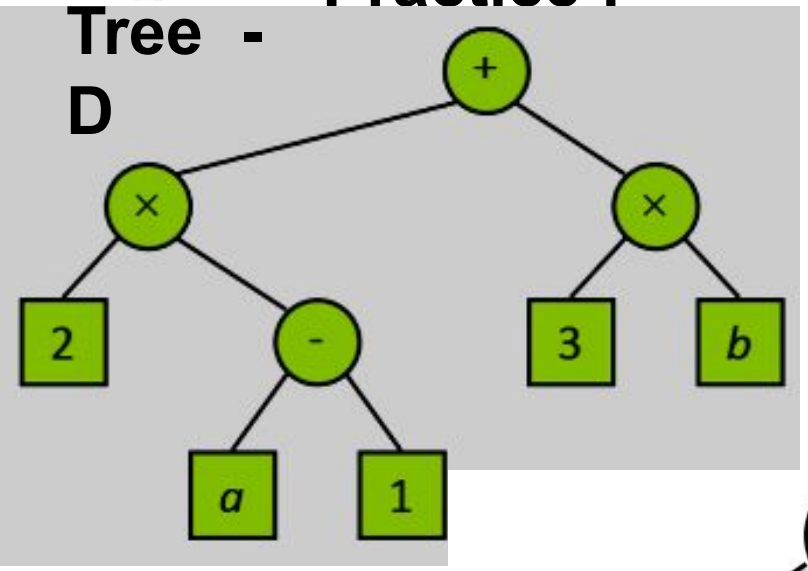


BINARY TREE TRAVERSALS

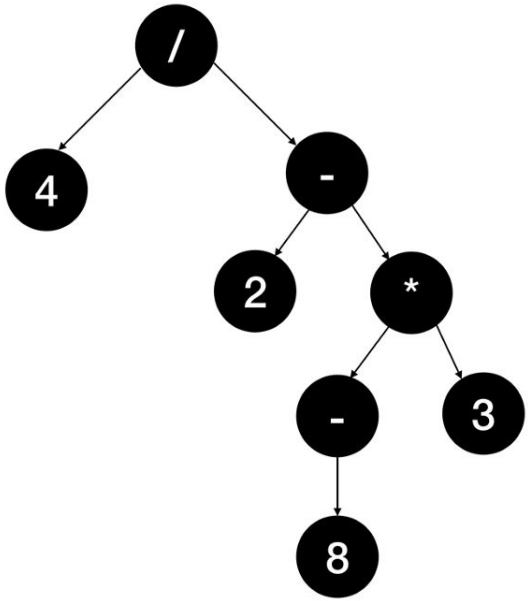
Dr. APJ ABDUL KALAM IIIT ONGOLE RGUKT AP

Examples Practice :

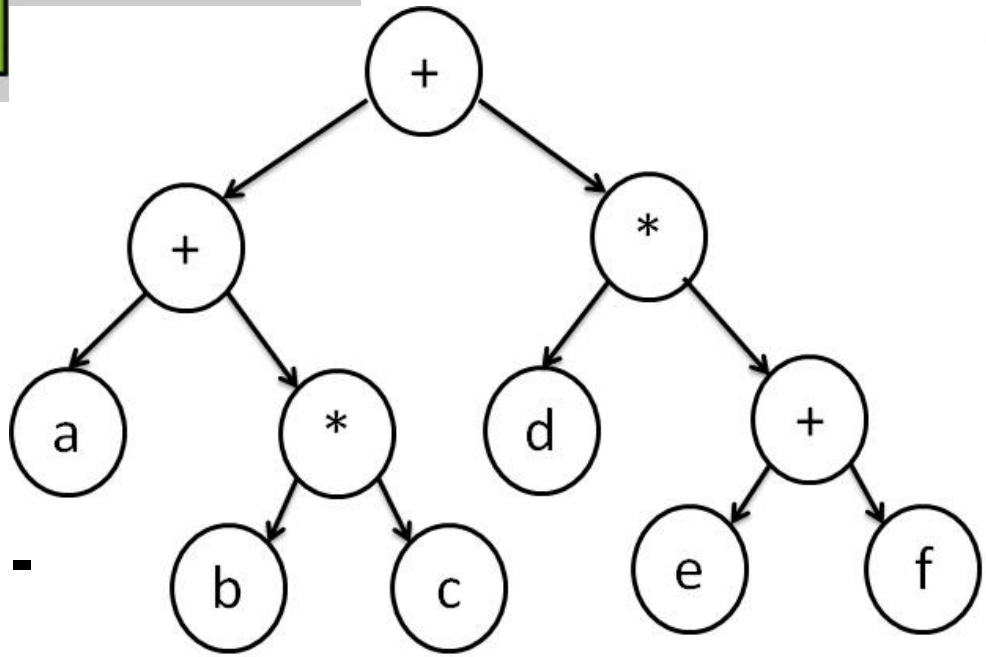
Tree - D



Tree - E



Tree - F



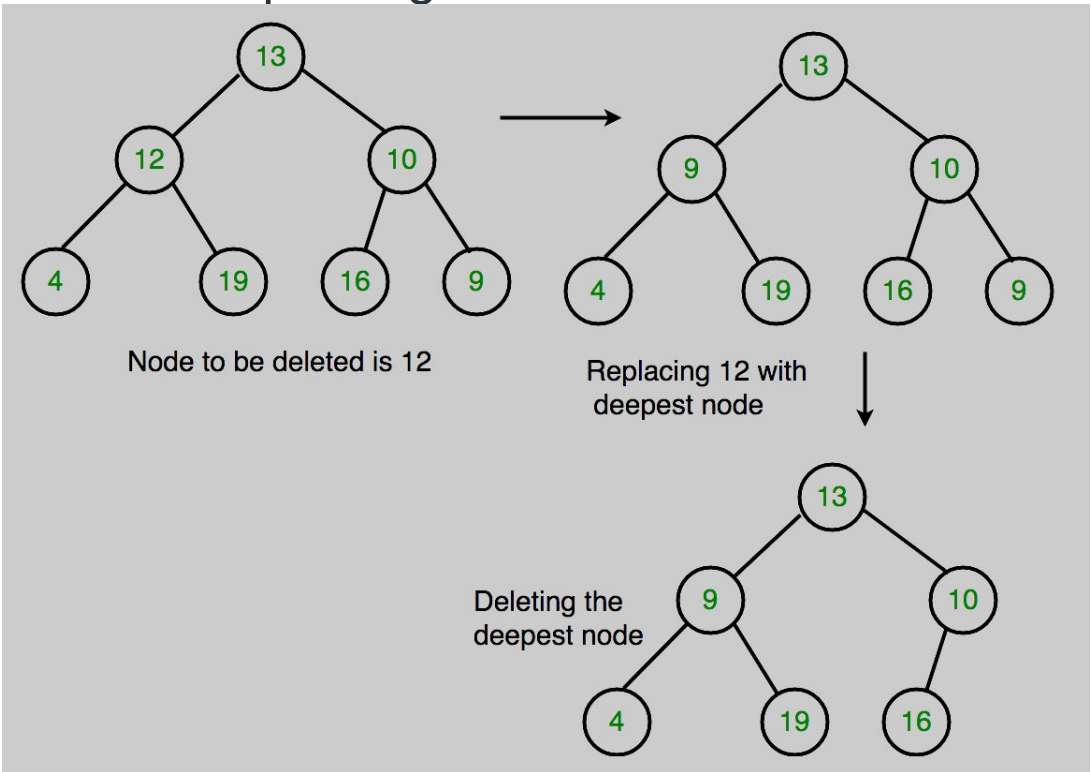
DELETING A NODE FROM COMPLETE BINARY TREE

Algorithm

1. Starting at the root, find the deepest and rightmost node in binary tree and 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.



PRIORITY QUEUE

Priority Queue is more specialized data structure than Queue.

A priority queue is **an abstract data type similar to a regular queue** or stack data structure in which each element additionally has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

In a queue, the **first-in-first-out rule** is implemented whereas, in a priority queue, the values are removed **on the basis of priority**. The element with the highest priority is removed first.

A *priority queue* is a data structure for maintaining a collection of items each with an associated *key* (or priority).

CHARACTERISTICS OF A PRIORITY QUEUE

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

A priority queue supports the following operations:

***insert* -** add an item and its key to the priority queue.
***maximum* (or *minimum*) -** return the item of highest priority.
***extractMax* (or *extractMin*) -** remove the item of

TYPES & APPLICATION OF PRIORITY QUEUES

There are two types of priority queues:

1. Max-priority queue.
2. Min-priority queue.

In both kinds, the priority queue stores a collection of elements and is always able to provide the most **"extreme"** element, which is the only way to interact with the priority queue.

PRIORITY QUEUE APPLICATIONS:

- *Operating systems.*
- *Time-activated events.*
- *Managing responsibilities.*
- *Sorting.*

IMPLEMENTATIONS OF PRIORITY QUEUES

The several possibilities for data structures we could use to implement a priority queue:

Unordered Array:- A simple, yet inefficient implementation, as retrieving the max element would require searching the entire array.

Sorted Array :- This is not a very efficient implementation either. Inserting a new element requires linearly searching the array for the correct position. Removing similarly requires a linear time: the rest of the elements need to be adjusted (shifted) into correct positions.

Hash table:- Although inserting into a hash table takes constant time (given a good hash function), finding the max element takes linear time. Therefore, this would be a poor choice for the underlying data structure.

Heap:- It turns out that that a heap makes an efficient priority queue.

IMPLEMENTATIONS OF PRIORITY QUEUES

A summary of **Complexities** of the Common operations on the Priority queue,

OPERATION	UNORDERED ARRAY	SORTED ARRAY	HASH TABLE	BINARY HEAP
Insert	O(1)	O(n)	O(1)	O(log(n))
maxElement	O(n)	O(1)	O(n)	O(1)
removeMax Element	O(n)	O(1)	O(n)	O(log(n))

BINARY HEAP

A heap is a tree-based data structure in which all the nodes of the tree are in a specific order.

For example, if X is the parent node of Y, then the value of X follows a specific order with respect to the value of Y and the same order will be followed across the tree.

The maximum number of children of a node in a heap depends on the type of heap. However, in the more commonly-used heap type, there are at most 2 children of a node and it's known as a Binary heap.

The (binary) **heap** data structure is an array that represents a *nearly complete* binary tree.

Heap is a special case of balanced binary tree data

TYPES OF HEAP

Two types of Heaps are,
Max Heap and MinHeap

Max Heap: In this type of heap, the value of parent node will always be greater than or equal to the value of child node across the tree and the node with highest value will be the root node of the tree.

Max Heap Property: The key of a node is ≥ than the keys of its children.

root of tree : first element in the array, corresponding to i = 1. parent(i) =i/2 : returns index of node's parent.

left(i)=2i : returns index of node's left child. right(i)=2i+1 : returns index of node's

HEAPIFY

Heapify rearranges the elements of an array where the left and right sub-tree of i^{th} element obeys the heap property.

Definition: Rearrange a heap to maintain the heap property, that is, the key of the root node is more extreme (greater or less) than or equal to the keys of its children. If the root node's key is not more extreme, swap it with the most extreme child key, then recursively heapify that child's subtree. The child subtrees must be heaps to start.

heapify(array)***MAX-HEAPIFY*** AND ***MIN-HEAPIFY***

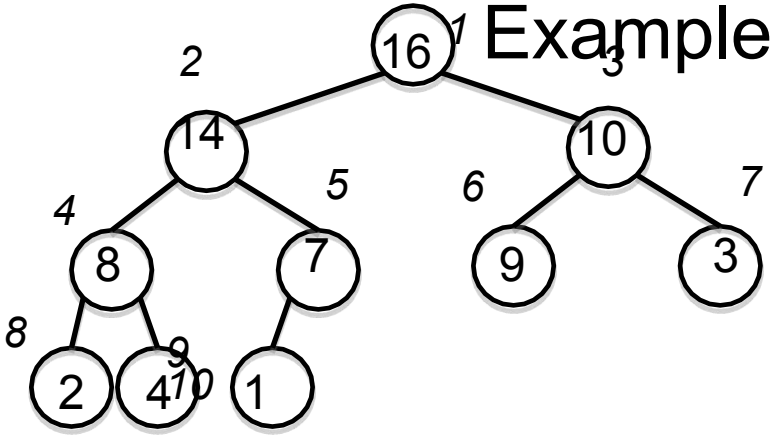
```
    Root = array[0]
      Largest = largest( array[0] , array [2*0 + 1]. array[2*0+2])
      if(Root != Largest)
         Swap(Root, Largest)
```

HEAPIFY

```
void heapify(int arr[], int n, int i)
{ // Find largest among root, left child and right child
    int largest = i;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < n && arr[left] > arr[largest])
       largest = left;
       if (right < n && arr[right] > arr[largest])
          largest = right; // Swap and continue heapifying if root is not
largest
        if (largest != i)
       {
            swap(&arr[i], &arr[largest]);
            heapify(arr, n, largest);
        }
}
```

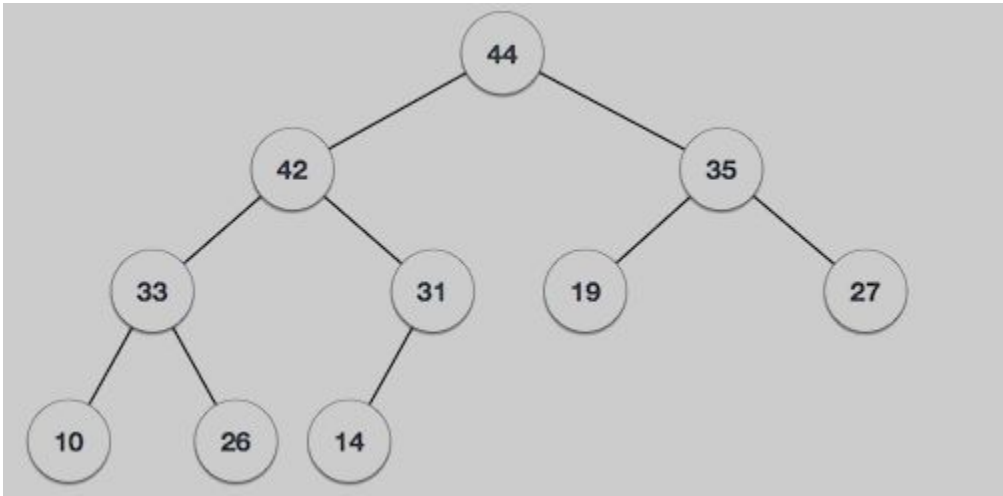
MAX HEAP

Max-Heap − Where the value of the root node is greater than or equal to either of its children.



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Input → 35 33 42 10 14 19 27 44 26 31



MAX HEAP CONSTRUCTION ALGORITHM

Step 1 − Create a new node at the end of heap.

Step 2 − Assign new value to the node.

Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

```
// Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);
```

BUILD MAX HEAP EXAMPLE

Input 35 33 42 10 14 19 27 44 26 31

MAX HEAP DELETION ALGORITHM

Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

Step 1 − Remove root node.

Step 2 − Move the last element of last level to root.

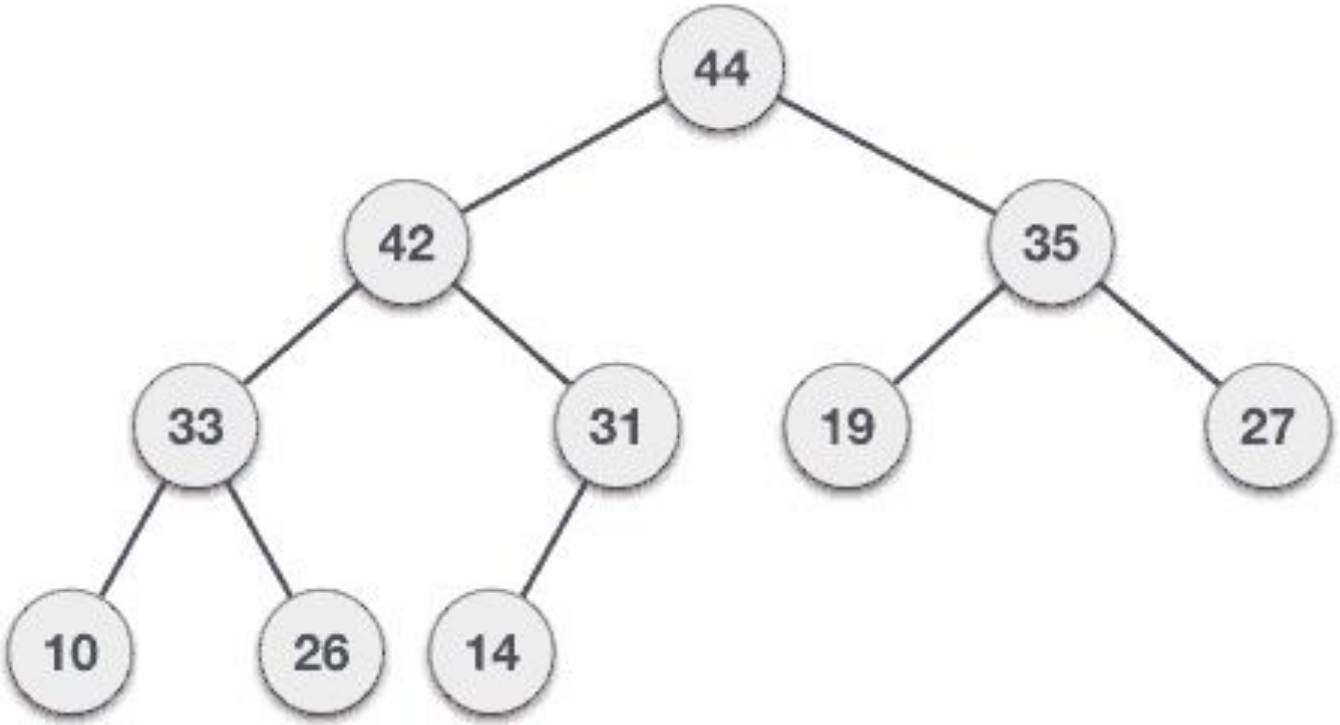
Step 3 − Compare the value of this child node with its parent.

Step 4 − If value of parent is less than child, then swap them.

Step 5 − Repeat step 3 & 4 until Heap property holds.

MAX HEAP DELETION EXAMPLE

Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value



MIN HEAP

Min-Heap − ***A Min-Heap is a complete binary tree in which the value in each internal node is smaller than or equal to the values in the children of that node.***

A min-heap has the property that every node at level 'n' stores a value that is less than or equal to that of its children at level 'n+1'. Because the root has a value less than or equal to its children, which in turn have values less than or equal to their children, the root stores the minimum of all values in the tree.

Operations on Min Heap:

getMin(): It returns the root element of Min Heap. Time Complexity of this operation is **O(1)**.

extractMin(): Removes the minimum element from MinHeap. Time Complexity of this Operation is **O(Log n)** as this operation needs to maintain the heap property (by calling heapify()) after removing root.

insert(): Inserting a new key takes **O(Log n)** time. We add a new key at the end of the tree. If new key is larger than its parent, then we don't need to do anything. Otherwise, we need to traverse up

BUILD MAX HEAP EXAMPLE

s.no	Min Heap	Max Heap
1.	In a Min-Heap the key present at the root node must be less than or equal to among the keys present at all of its children.	In a Max-Heap the key present at the root node must be greater than or equal to among the keys present at all of its children.
2.	In a Min-Heap the minimum key element present at the root.	In a Max-Heap the maximum key element present at the root.
3.	A Min-Heap uses the ascending priority.	A Max-Heap uses the descending priority.
4.	In the construction of a Min-Heap, the smallest element has priority.	In the construction of a Max-Heap, the largest element has priority.
5.	In a Min-Heap, the smallest element is the first to be popped from the heap.	In a Max-Heap, the largest element is the first to be popped from the heap.

HEAP SORT

Working of Heap Sort

Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

Swap: Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.

Remove: Reduce the size of the heap by 1.

Heapify: Heapify the root element again so that we have the highest element at root.

The process is repeated until all the items of the list are sorted.

HEAP SORT

Step 1 - Construct a **Binary Tree** with given list of Elements.

Step 2 - Transform the Binary Tree into **Max Heap.**

Step 3 - Delete the root element from Min Heap using **Heapify** method.

Step 4 - Put the deleted element into the Sorted list.

Step 5 - Repeat the same until Max Heap becomes empty.

Step 6 - Display the sorted list.

HEAP SORT

6 5 3 1 8 7 2 4