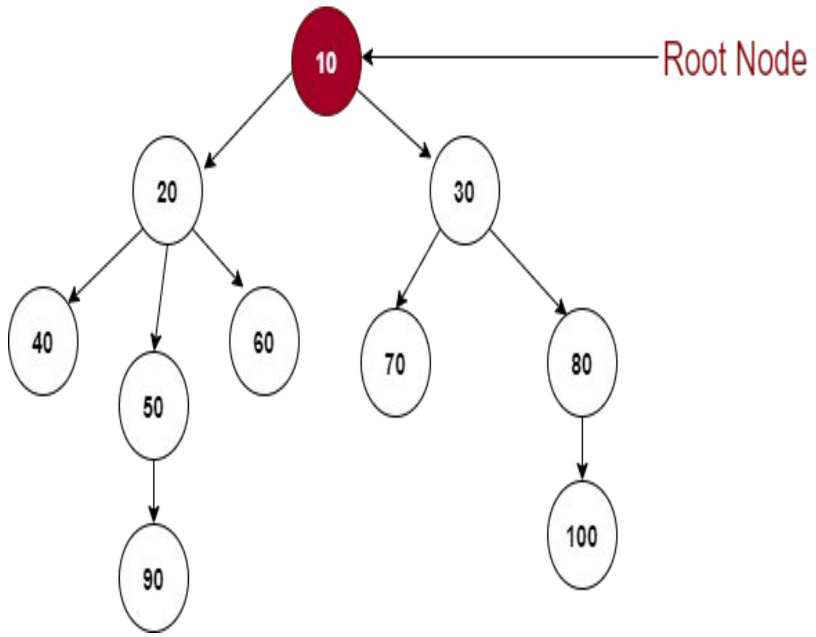
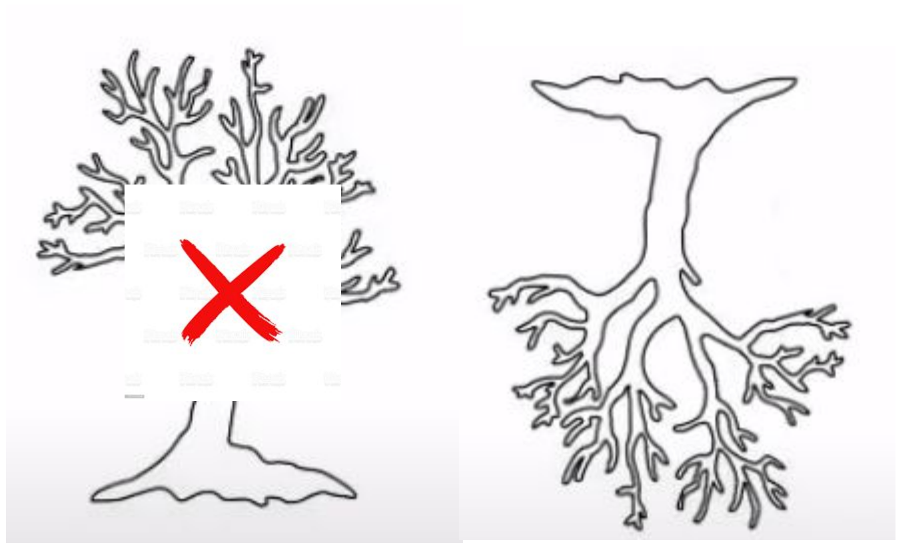
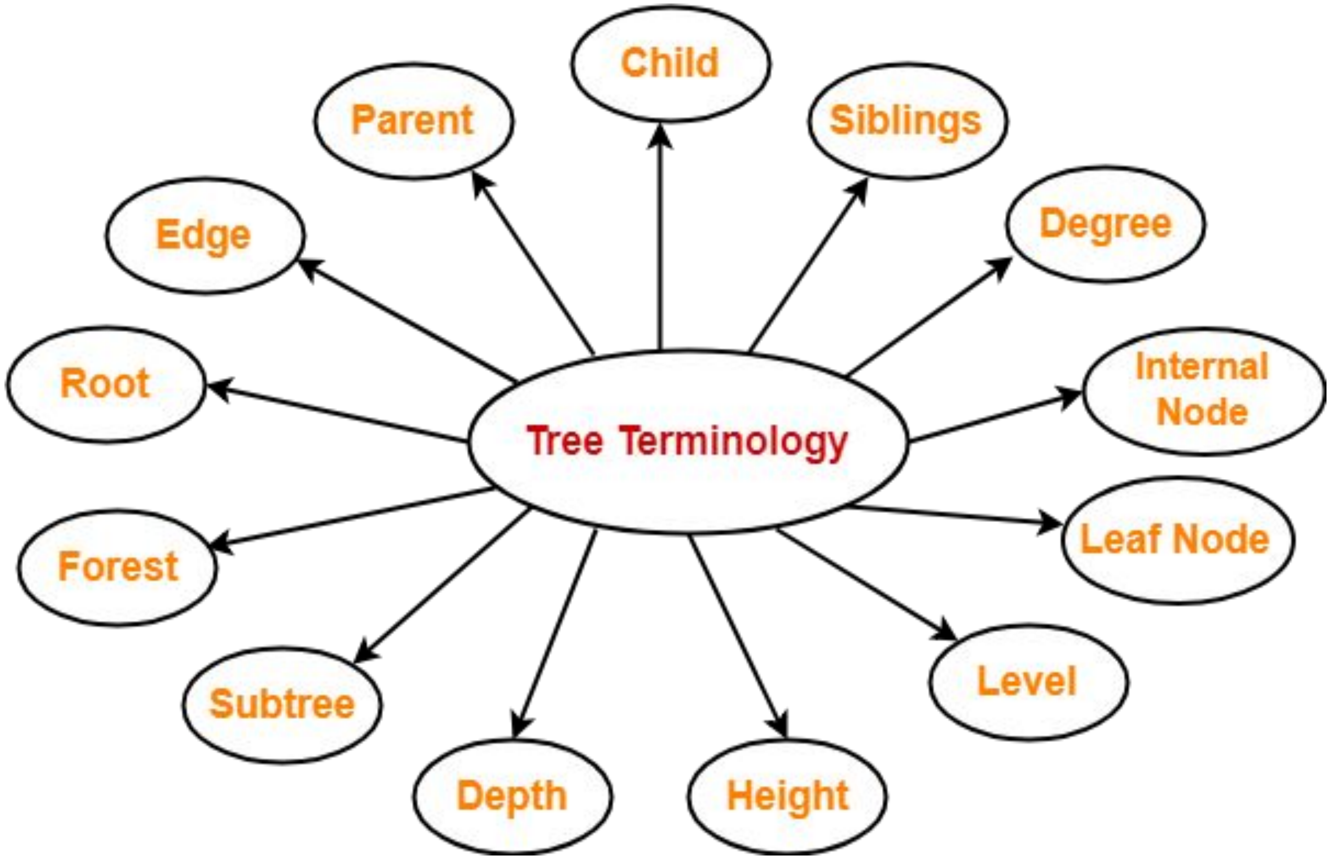
Tree

- A tree is a non-linear data structure. It is a collection of objects or entities known as nodes that are linked together by directed or undirected edges to represent or simulate hierarchy.
- A tree data structure is a non-linear data structure because it does not store in a sequential manner. It is a hierarchical structure as elements in a Tree are arranged in multiple levels.





Tree Terminologies



Tree Terminologies

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

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

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

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

- Siblings : Nodes which belong to the same parent are called as siblings.

Tree Terminologies

- Degree : Degree of a node is the total number of children of that node. Degree of a tree is the highest degree of a node among all the nodes in the tree.

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

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

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

Tree Terminologies

- Height : Total number of edges that lies on the longest path from any leaf node to a particular node is called as height of that node. Height of a tree is the height of root node. Height of all leaf nodes = 0

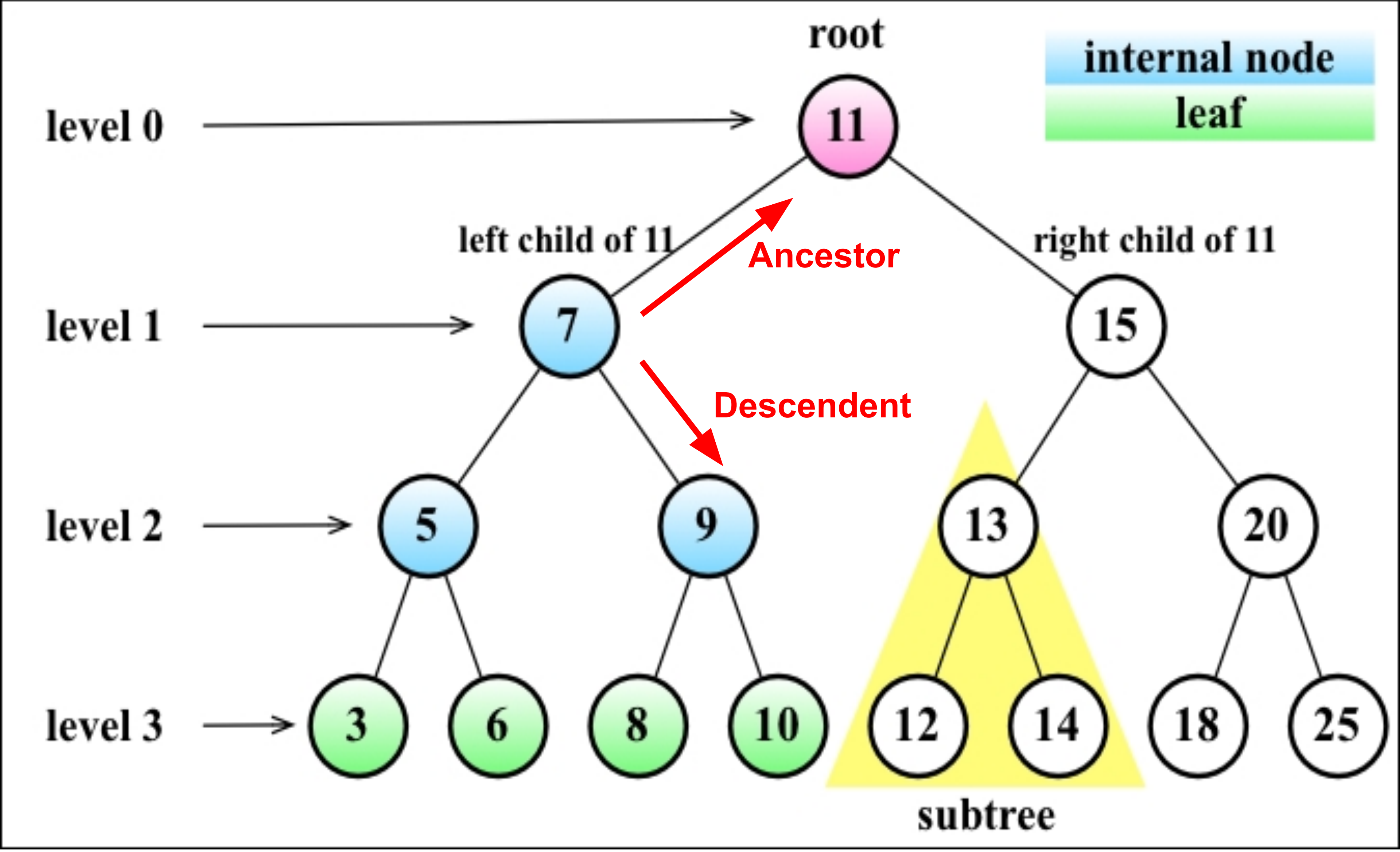
- Depth : Total number of edges from root node to a particular node is called as depth of that node. Depth of a tree is the total number of edges from root node to a leaf node in the longest path. Depth of the root node = 0. The terms “level” and “depth” are used interchangeably.

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

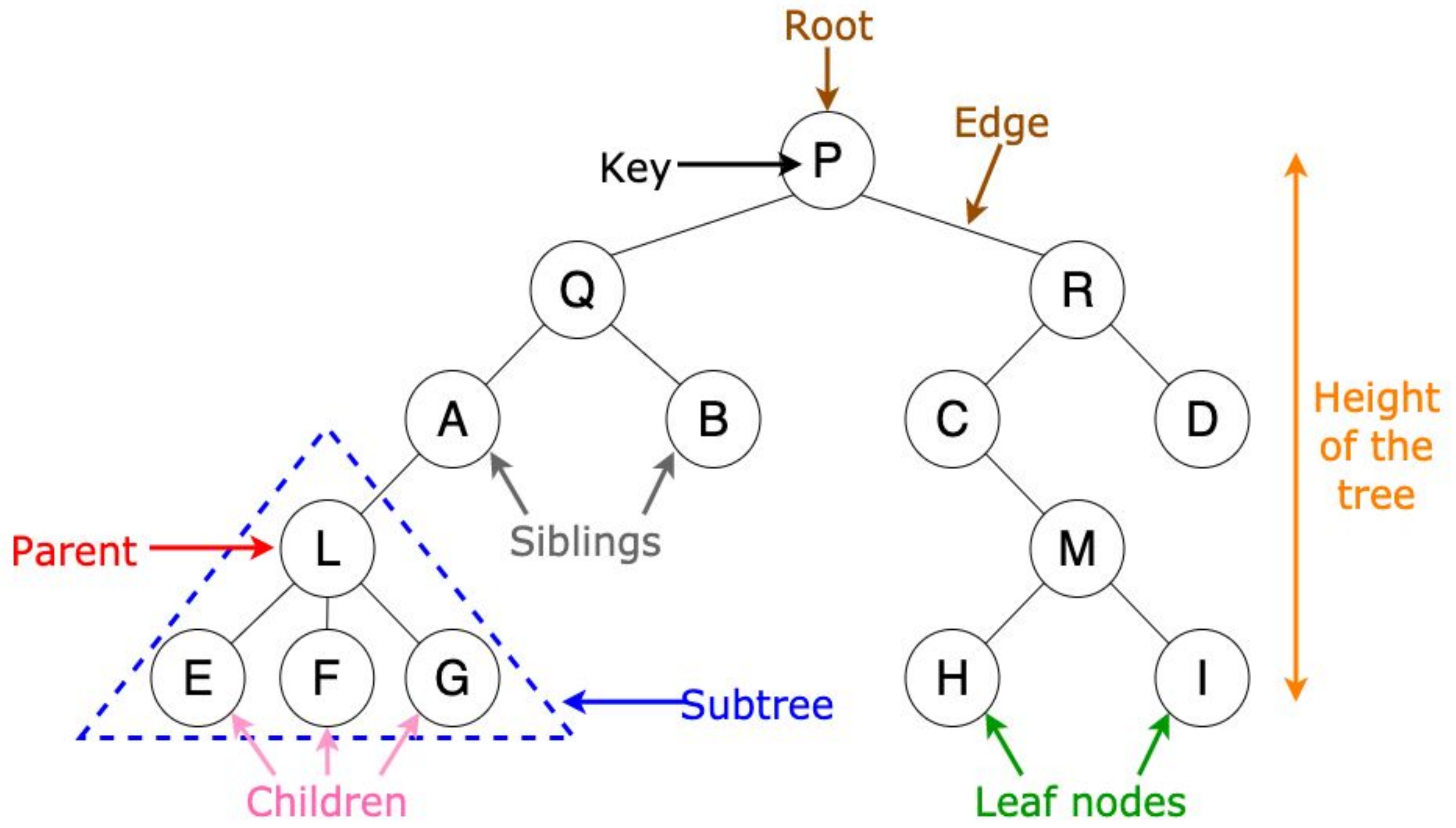
Tree Terminologies

- Ancestor : For any node n, its ancestors are the nodes which are on the path between roots to n.
- Descendent : The immediate successor of the given node is known as a descendant of a node.
- Keys : Key represents a value based on which a search operation is to be carried out for a node.
- Forest : A forest is a set of disjoint trees.

Tree Terminologies

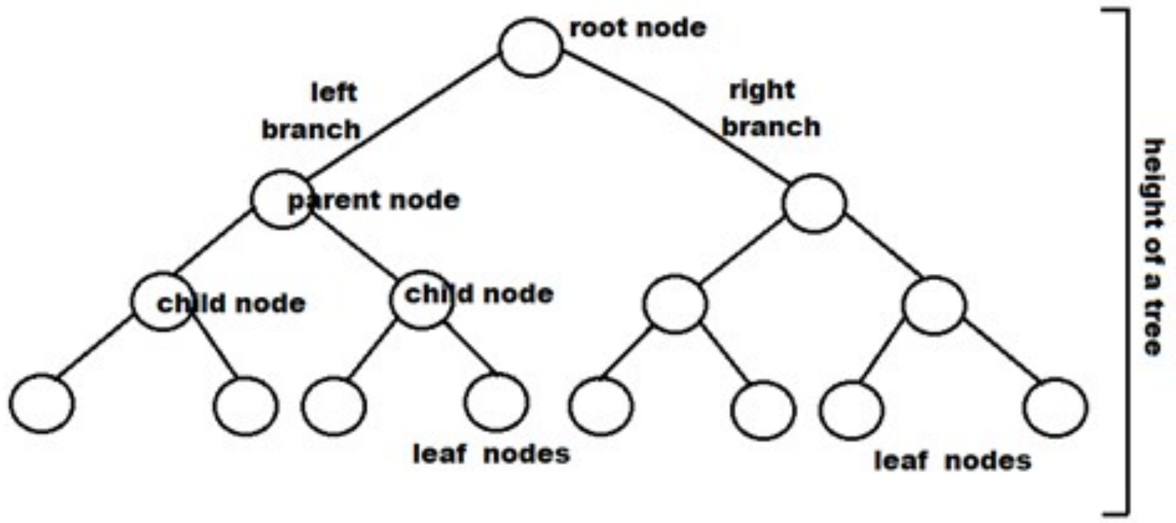


Tree Terminologies



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.

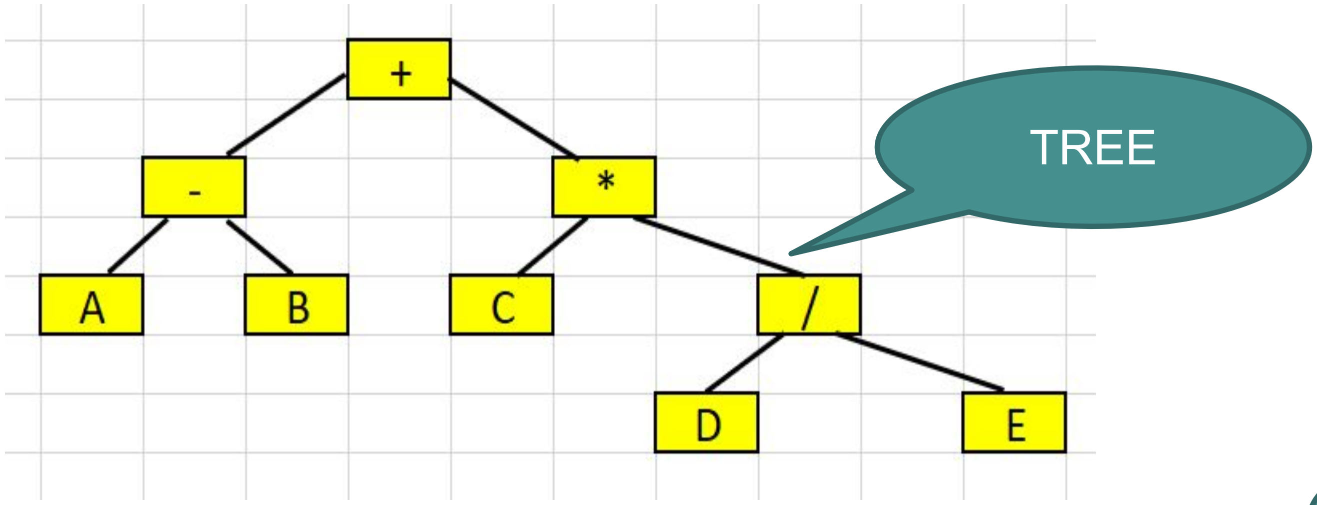


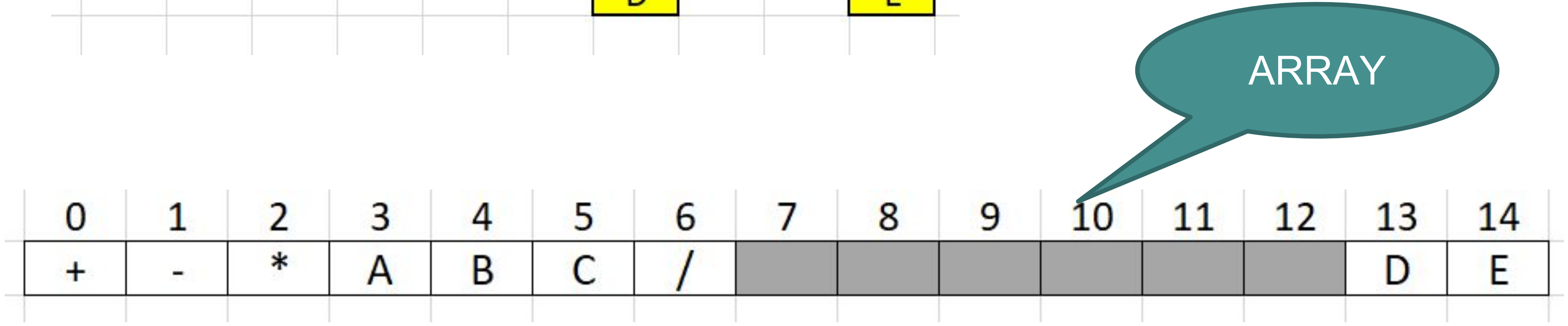
Binary Tree Properties

- In any binary tree, the maximum number of nodes on level 'l' is '2^l'.
- The maximum number of nodes possible in a binary tree of height h is 2^h-1 (2^{h+1}-1, if h is starting from 0).
- The minimum number of nodes possible in a binary tree of height h is h.
- For any non-empty binary tree, if n is the no. of nodes and e is the no. of edges then n=e+1
- Maximum no. of leaf nodes in binary tree = Total no. of nodes with 2 children + 1
- The height of complete binary tree with n number of nodes is $\log_2(n+1)$
- Total no. of binary trees possible with n nodes is $(1/(n+1)) * {}^{2n}C_n$

Binary Tree Representation

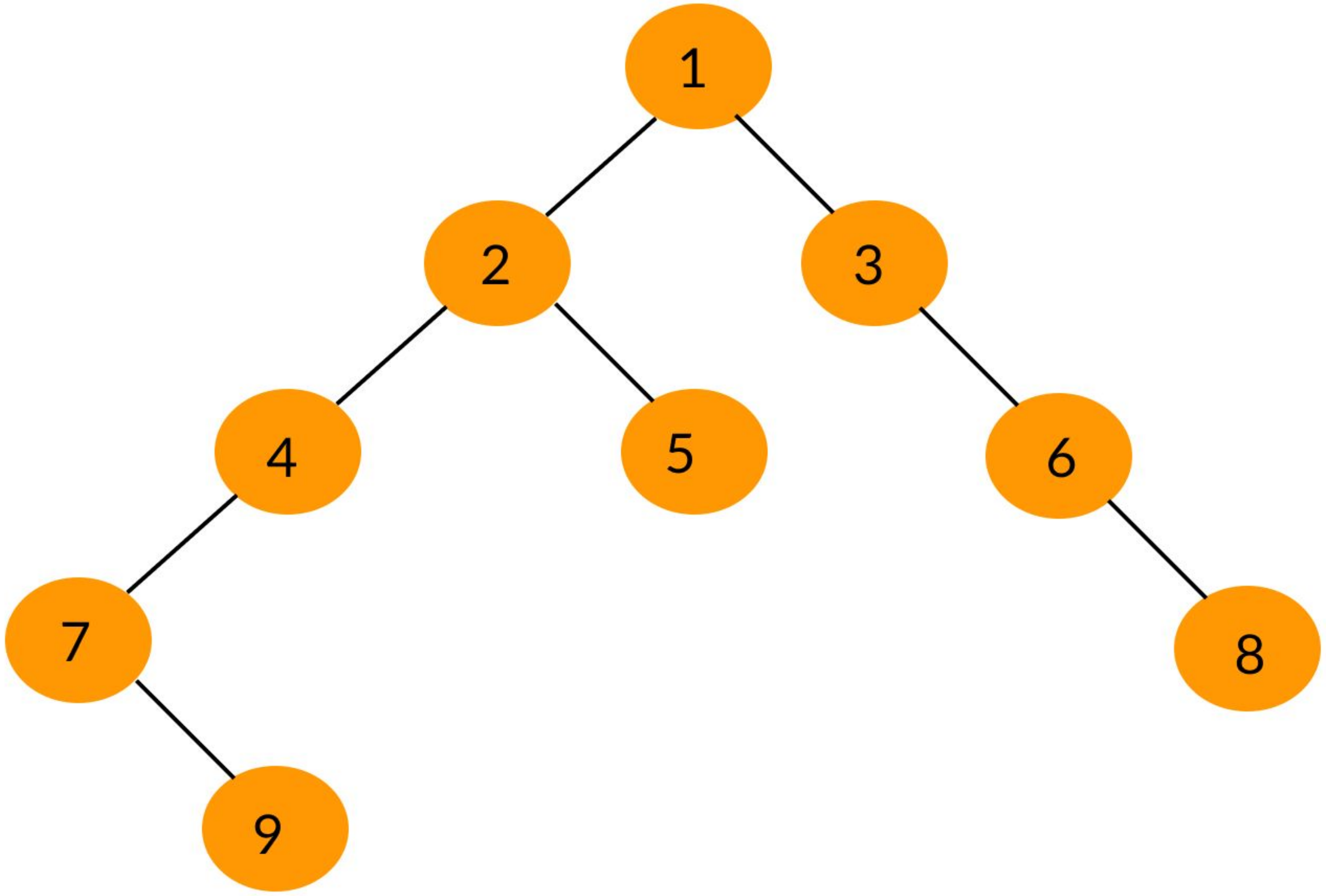
- **Linear Representation**
- Consider the expression (A-B) + C * (D/E)





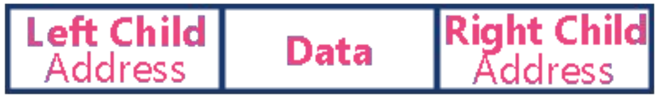
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
+	-	*	A	B	C	/	▨	▨	▨	▨	▨	▨	D	E

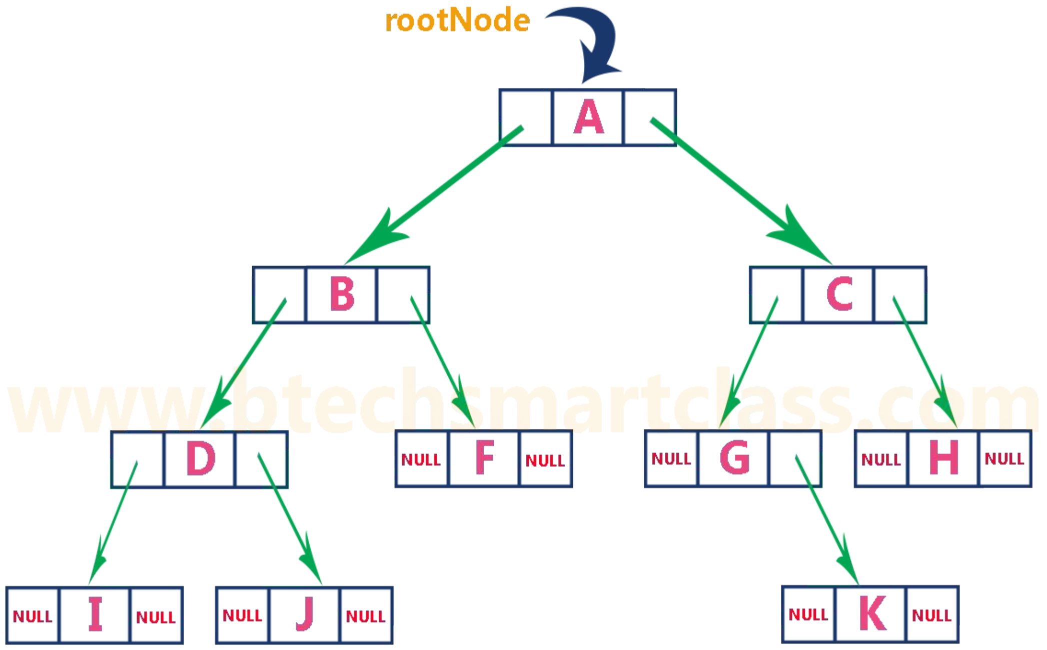
Represent the following tree using Array



Binary Tree Representation

- **Linked Representation :** We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.





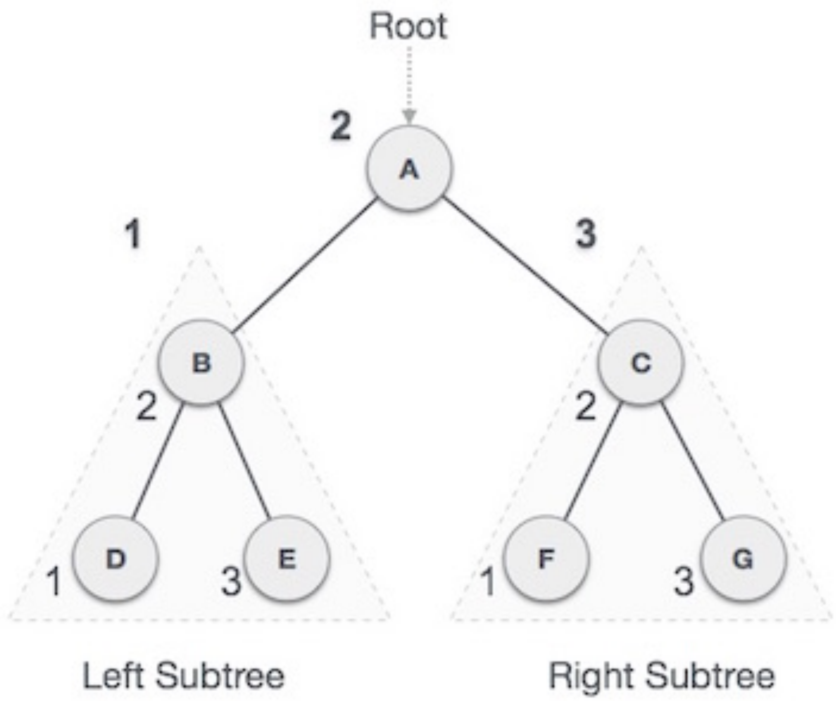
Binary Tree Traversal

- “In computer science, tree traversal (also known as tree search) is a form of graph traversal and refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.” — Wikipedia

- Traversal Methods
  - Inorder (Left Child – Root – Right Child)
  - Preorder (Root - Left Child – Right Child)
  - Postorder (Left Child – Right Child - Root)

Binary Tree Traversal

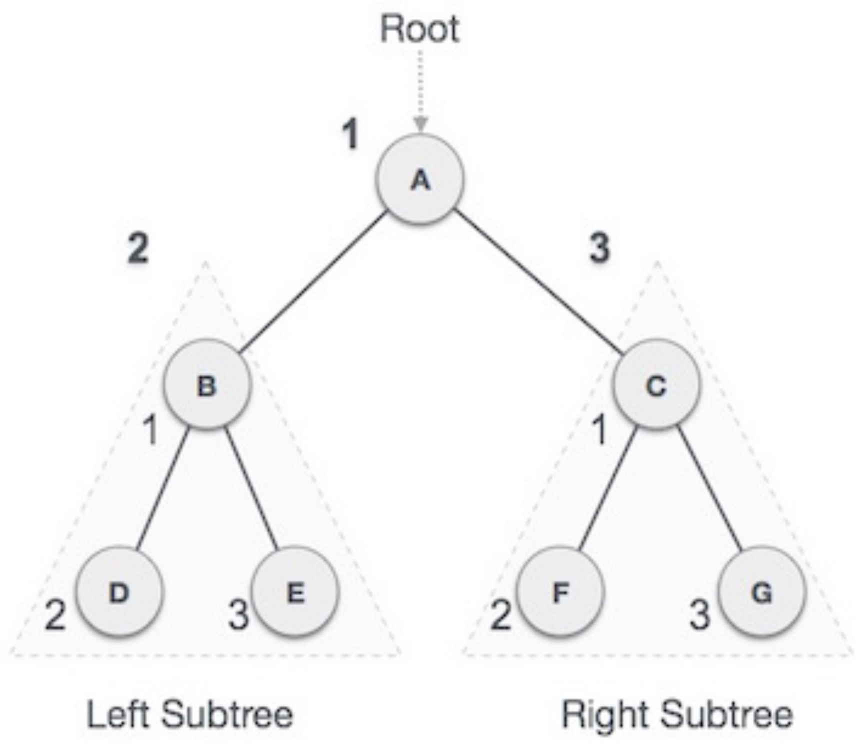
- Inorder Traversal : In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting the right child node. This in-order traversal is applicable for every root node of all subtrees in the tree. This is performed recursively for all nodes in the tree.



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

Binary Tree Traversal

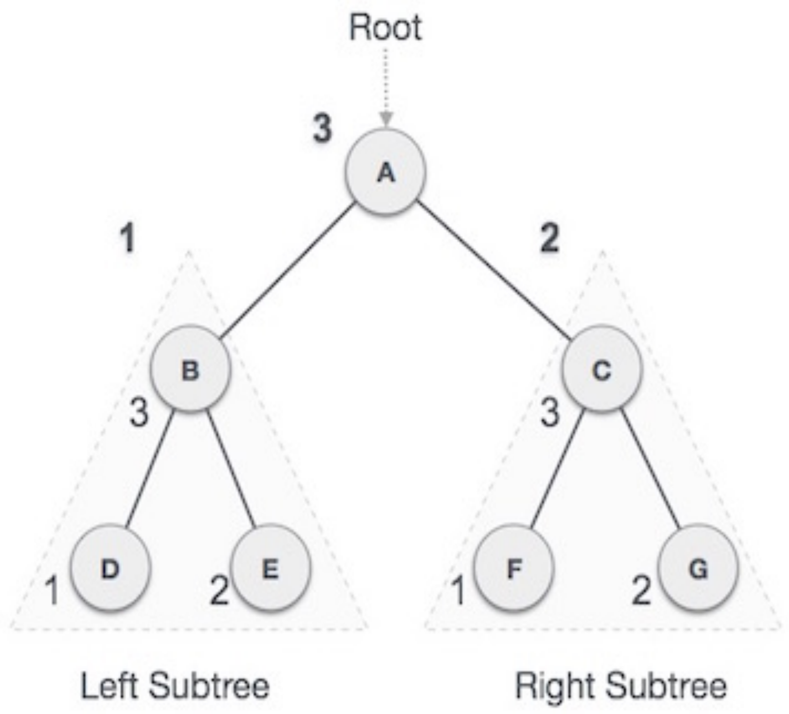
- Preorder Traversal : In this traversal, the root node is visited first, then its left child and later its right child. This pre-order traversal is applicable for every root node of all subtrees in the tree.



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

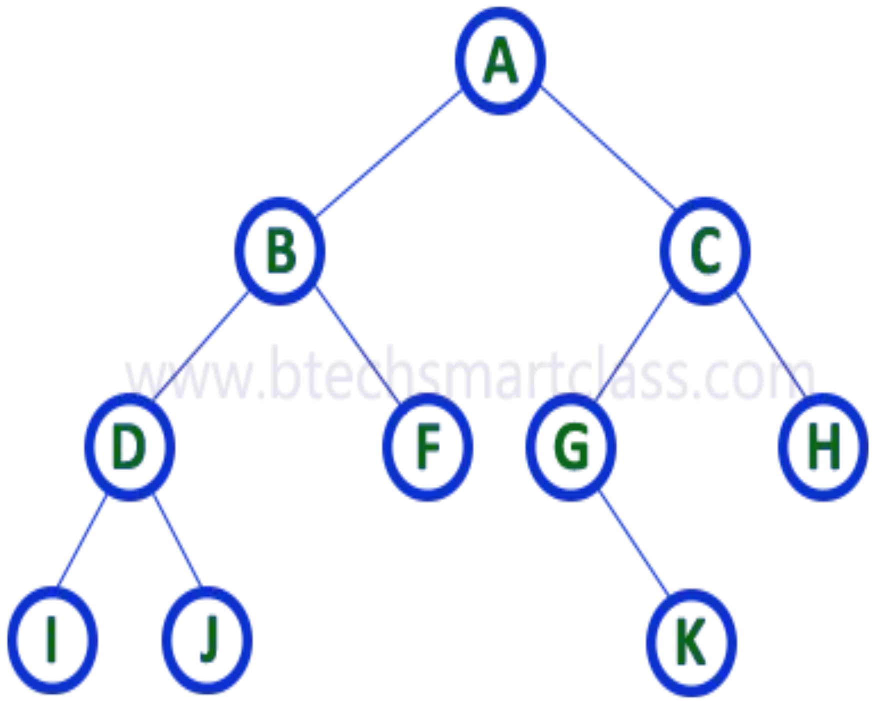
Binary Tree Traversal

- Postorder Traversal : In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.



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

Binary Tree Traversal



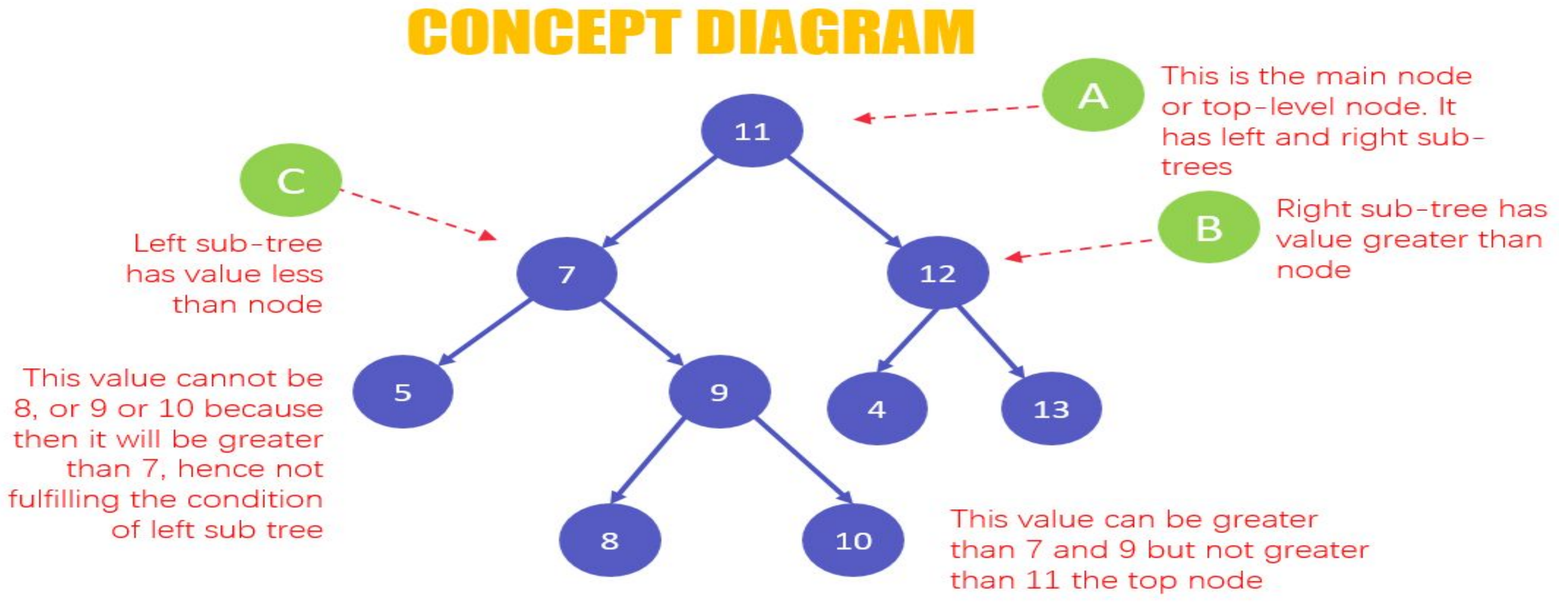
Inorder Traversal :
I - D - J - B - F - A - G - K - C - H

Preorder Traversal :
A - B - D - I - J - F - C - G - K - H

Postorder Traversal :
I - J - D - F - B - K - G - H - C - A

Binary Search Tree

- Binary Search Tree is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node’s key.
  - The right subtree of a node contains only nodes with keys greater than the node’s key.
  - The left and right subtree each must also be a binary search tree.

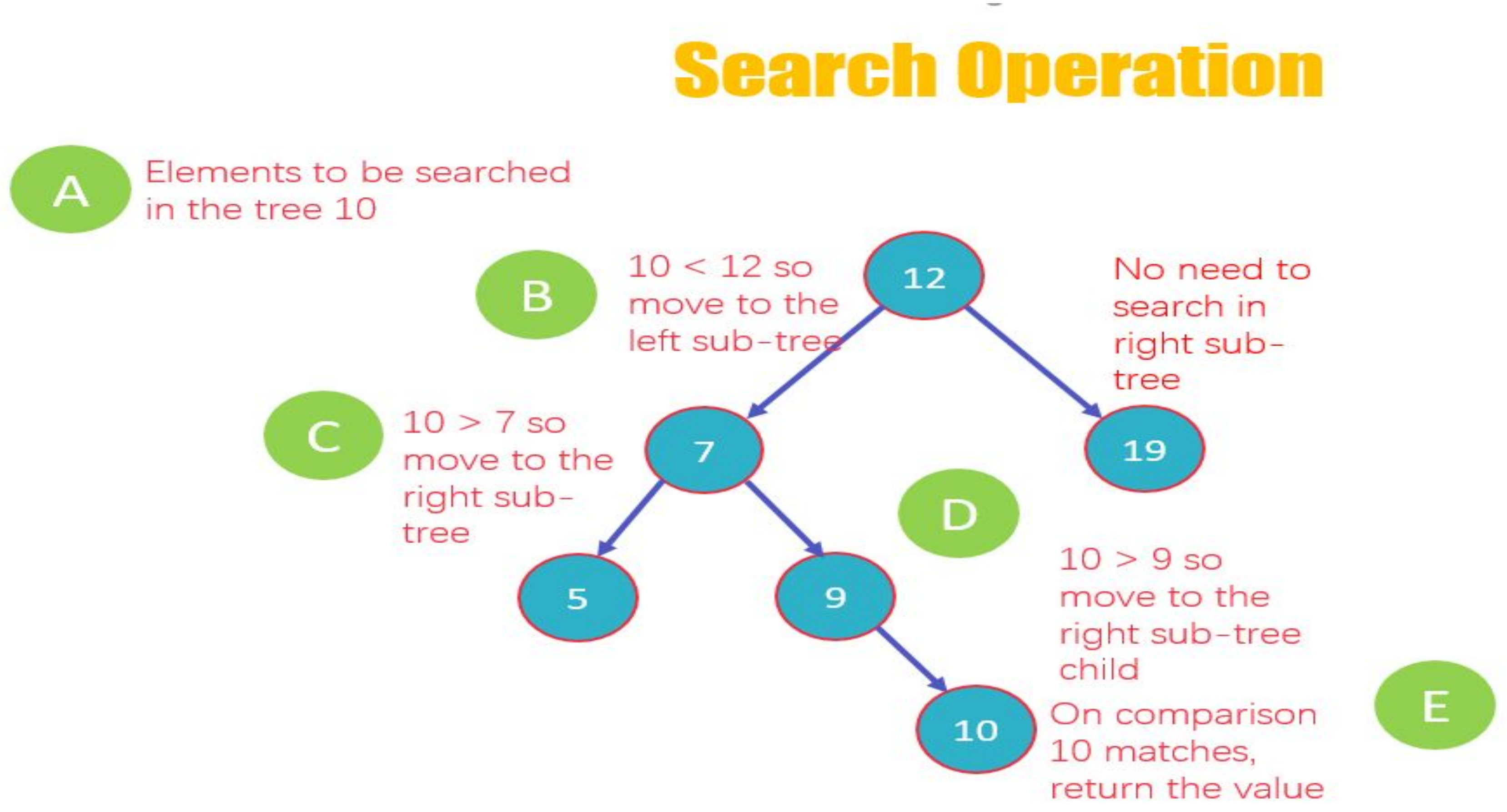

CONCEPT DIAGRAM
A
This is the main node or top-level node. It has left and right sub-trees
B
Right sub-tree has value greater than node
C
Left sub-tree has value less than node
This value cannot be 8, or 9 or 10 because then it will be greater than 7, hence not fulfilling the condition of left sub tree
This value can be greater than 7 and 9 but not greater than 11 the top node
11
7
12
5
9
4
13
8
10

Binary Search Tree

- Need of BST
  - The two major factors that make binary search tree an optimum solution to any real-world problems are Speed and Accuracy.
  - In case the element to be searched greater or less than the parent node, the node knows which tree side to search for. The reason is, the left sub-tree is always lesser than the parent node, and the right sub-tree has values always equal to or greater than the parent node.
  - BST is commonly utilized to implement complex searches, robust game logics, auto-complete activities, and graphics.
  - The algorithm efficiently supports operations like search, insert, and delete.

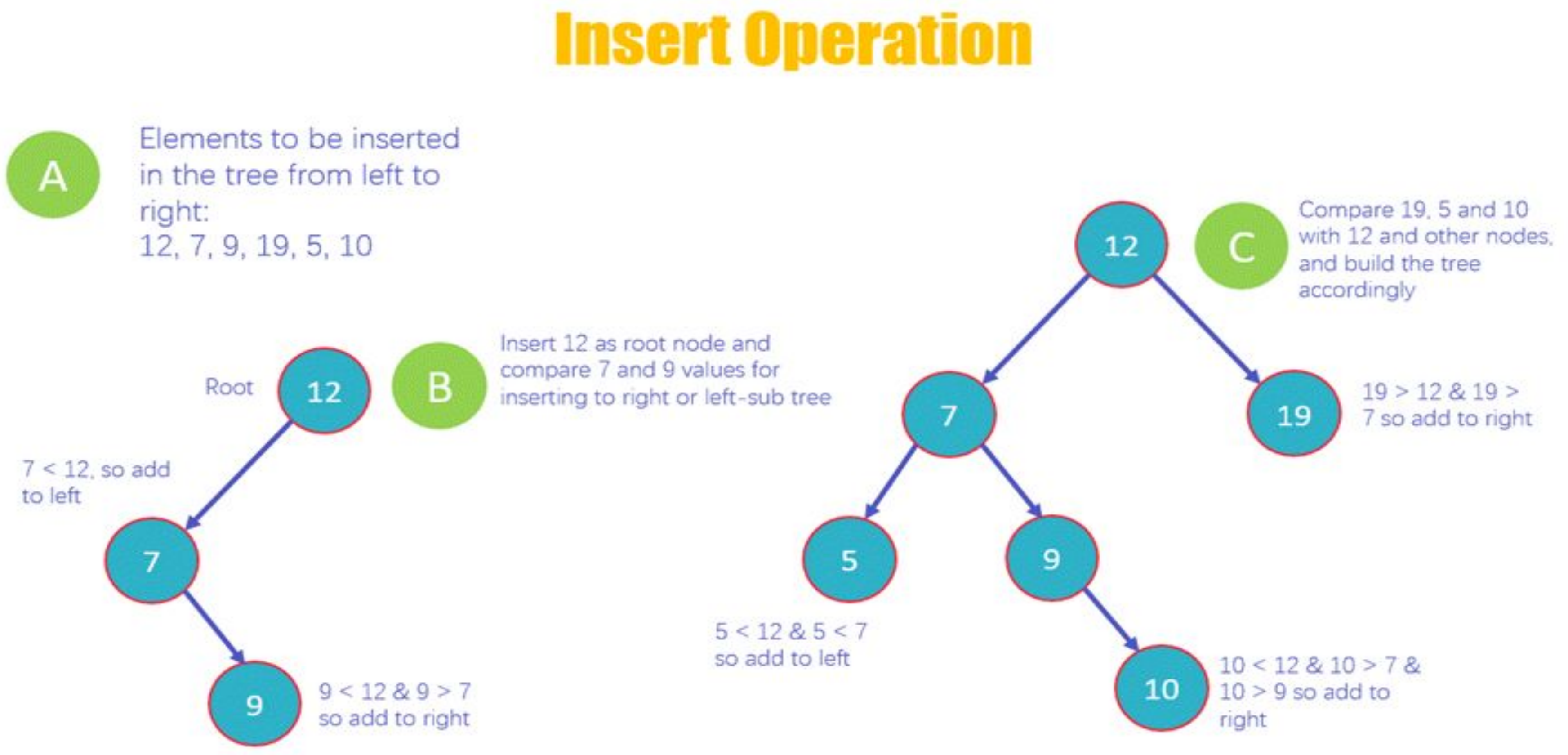
Binary Search Tree Operations

- Search : Always initiate analyzing tree at the root node and then move further to either the right or left subtree of the root node depending upon the element to be located is either less or greater than the root.


Search Operation
A
Elements to be searched in the tree 10
B
10 < 12 so move to the left sub-tree
12
No need to search in right sub-tree
C
10 > 7 so move to the right sub-tree
7
19
D
5
9
10 > 9 so move to the right sub-tree child
10
On comparison 10 matches, return the value
E

Binary Search Tree Operations

- Insert : This is a very straight forward operation. First, the root node is inserted, then the next value is compared with the root node. If the value is greater than root, it is added to the right subtree, and if it is lesser than the root, it is added to the left subtree.



Insert Operation
A
Elements to be inserted in the tree from left to right:
12, 7, 9, 19, 5, 10
B
Insert 12 as root node and compare 7 and 9 values for inserting to right or left-sub tree
Root
12
7 < 12, so add to left
7
9
9 < 12 & 9 > 7 so add to right
C
Compare 19, 5 and 10 with 12 and other nodes, and build the tree accordingly
12
7
19
19 > 12 & 19 > 7 so add to right
5
9
5 < 12 & 5 < 7 so add to left
10
10 < 12 & 10 > 7 & 10 > 9 so add to right

Binary Search Tree Operations

- Delete : Delete is the most advanced and complex among all other operations. There are multiple cases handled for deletion in the BST.

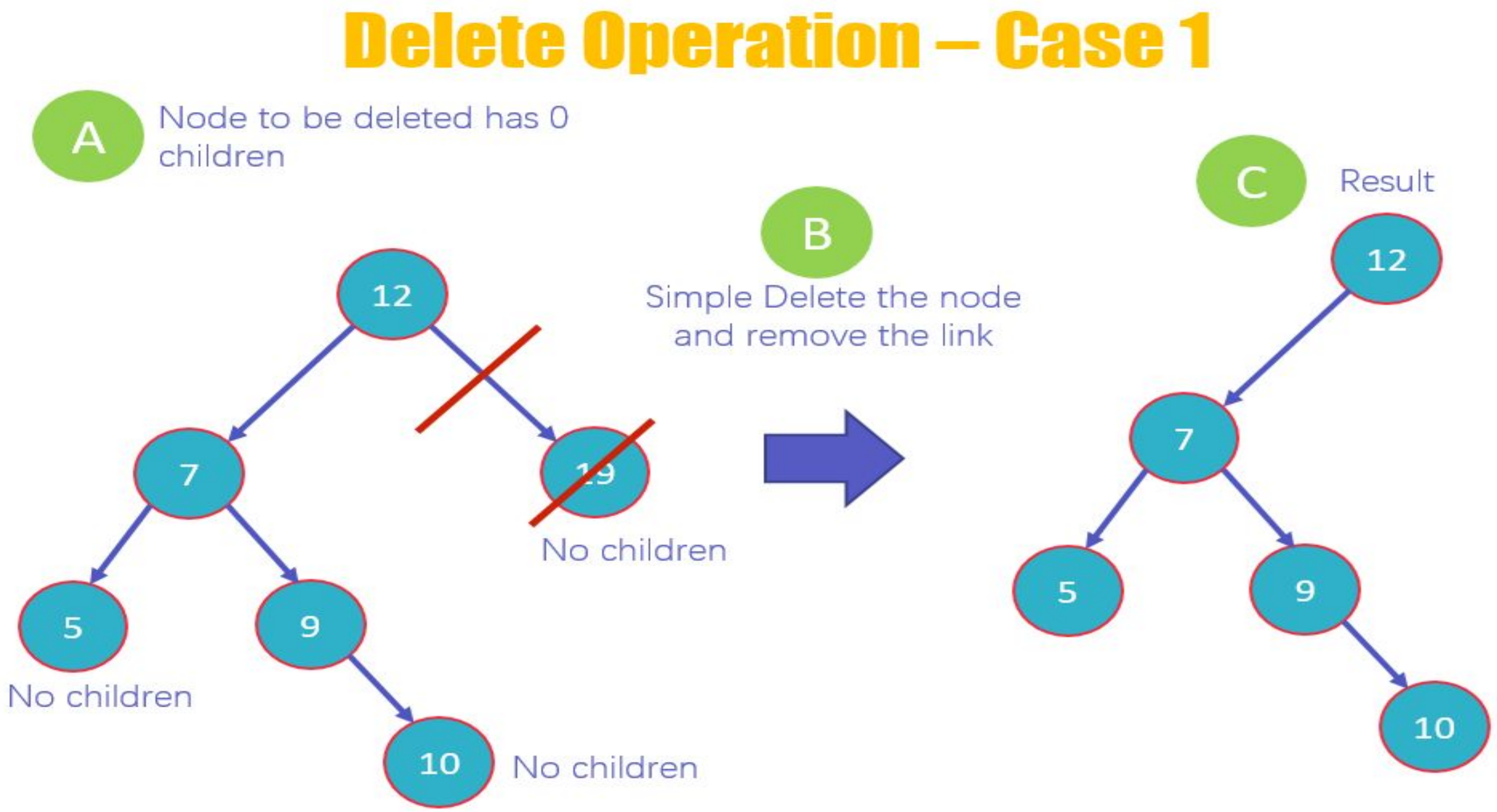
- Case 1- Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.

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

- Case 3 Node with two children: this is the most difficult situation, and it works on the following two rules
  - 3a - In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node
  - 3b - In Order Successor: you need to delete the node with two children and replace it with the smallest value on the right-subtree of the deleted node

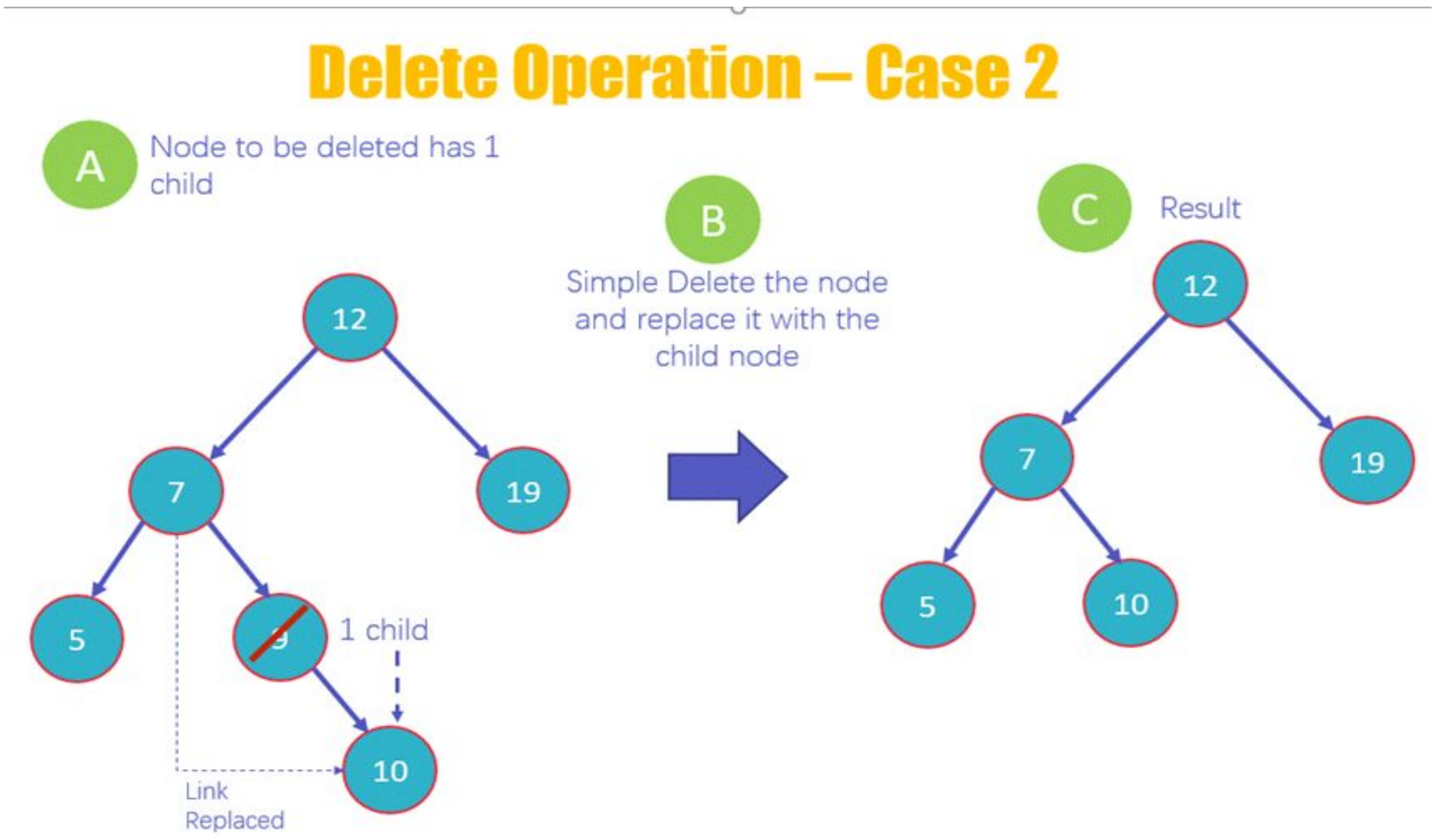
Binary Search Tree Operations

- Case 1:


Delete Operation – Case 1
A
Node to be deleted has 0 children
12
7
19
5
9
10
No children
No children
No children
B
Simple Delete the node and remove the link
C
Result
12
7
5
9
10

Binary Search Tree Operations

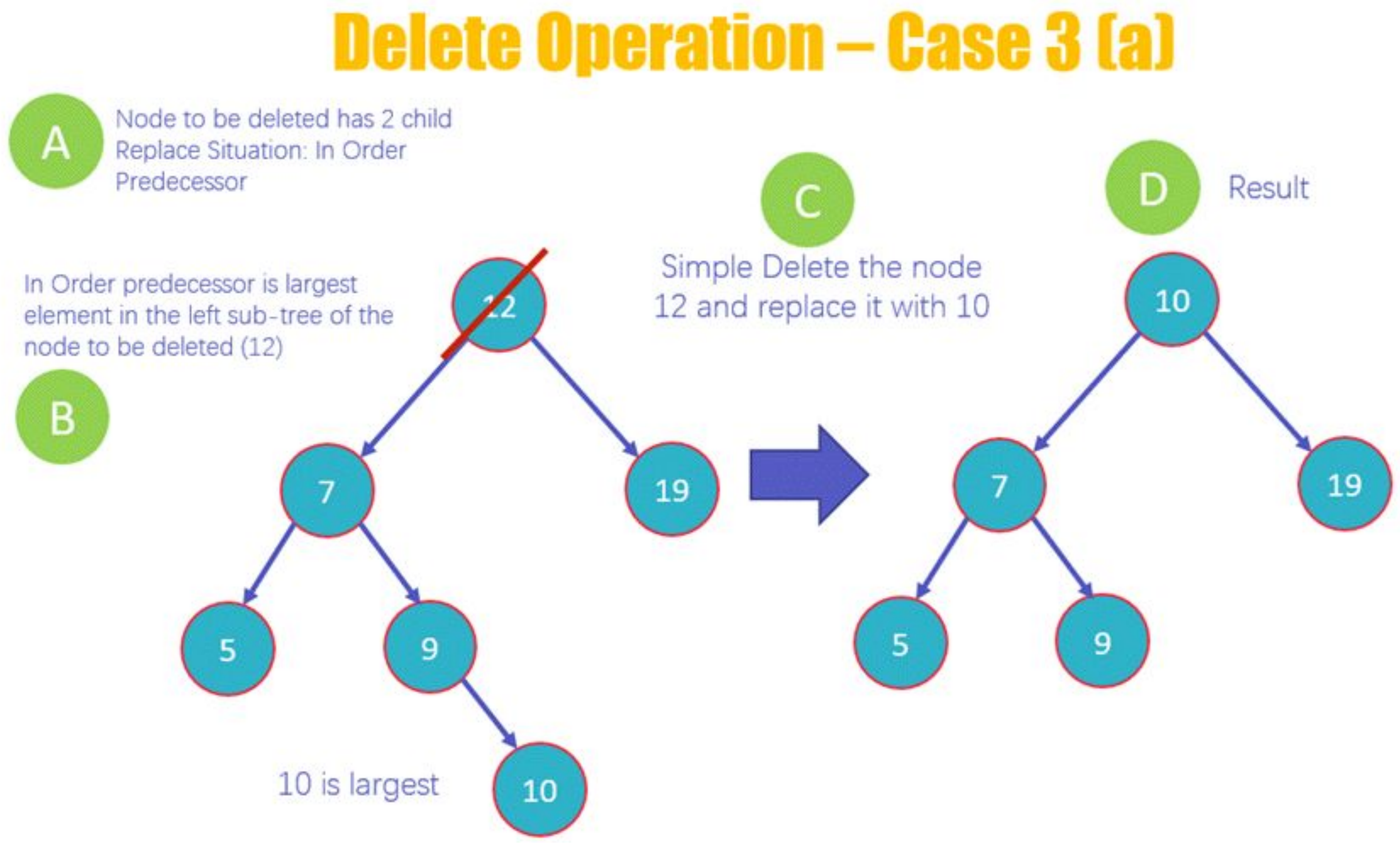
- Case 2:



Delete Operation – Case 2
A
Node to be deleted has 1 child
12
7
19
5
9
1 child
10
Link Replaced
B
Simple Delete the node and replace it with the child node
C
Result
12
7
19
5
10

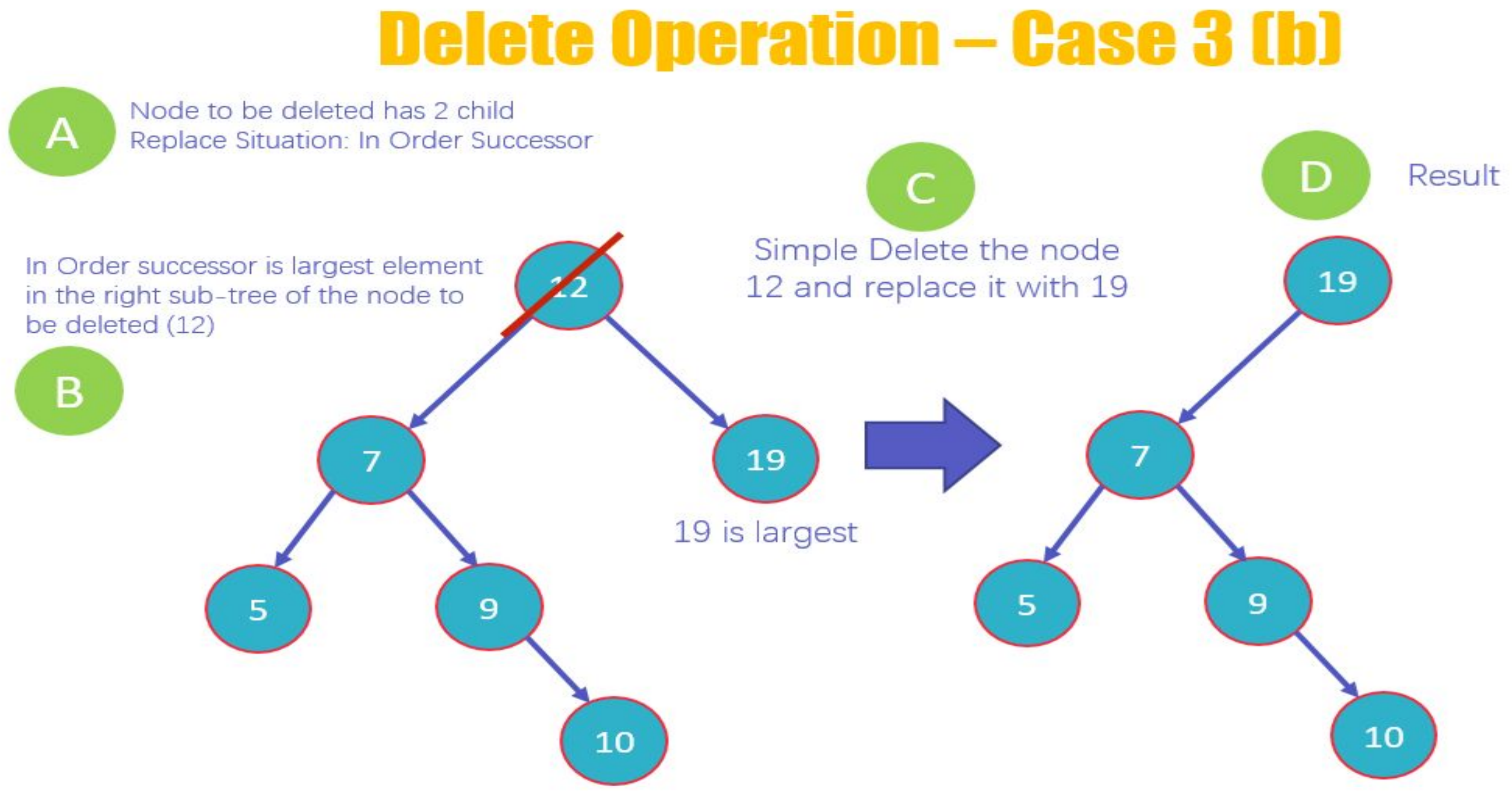
Binary Search Tree Operations

❖ Case 3a:


Delete Operation – Case 3 (a)
A
Node to be deleted has 2 child
Replace Situation: In Order Predecessor
B
In Order predecessor is largest element in the left sub-tree of the node to be deleted (12)
12
7
19
5
9
10
10 is largest
C
Simple Delete the node 12 and replace it with 10
D
Result
10
7
19
5
9

Binary Search Tree Operations

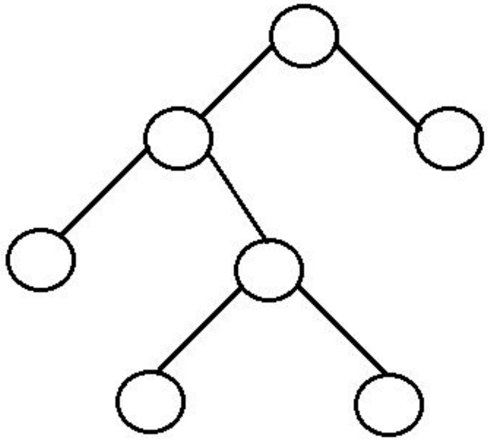
- Case 3b:



Delete Operation – Case 3 (b)
A
Node to be deleted has 2 child
Replace Situation: In Order Successor
In Order successor is largest element in the right sub-tree of the node to be deleted (12)
B
12
7
19
5
9
10
19 is largest
C
Simple Delete the node 12 and replace it with 19
D
Result
19
7
5
9
10

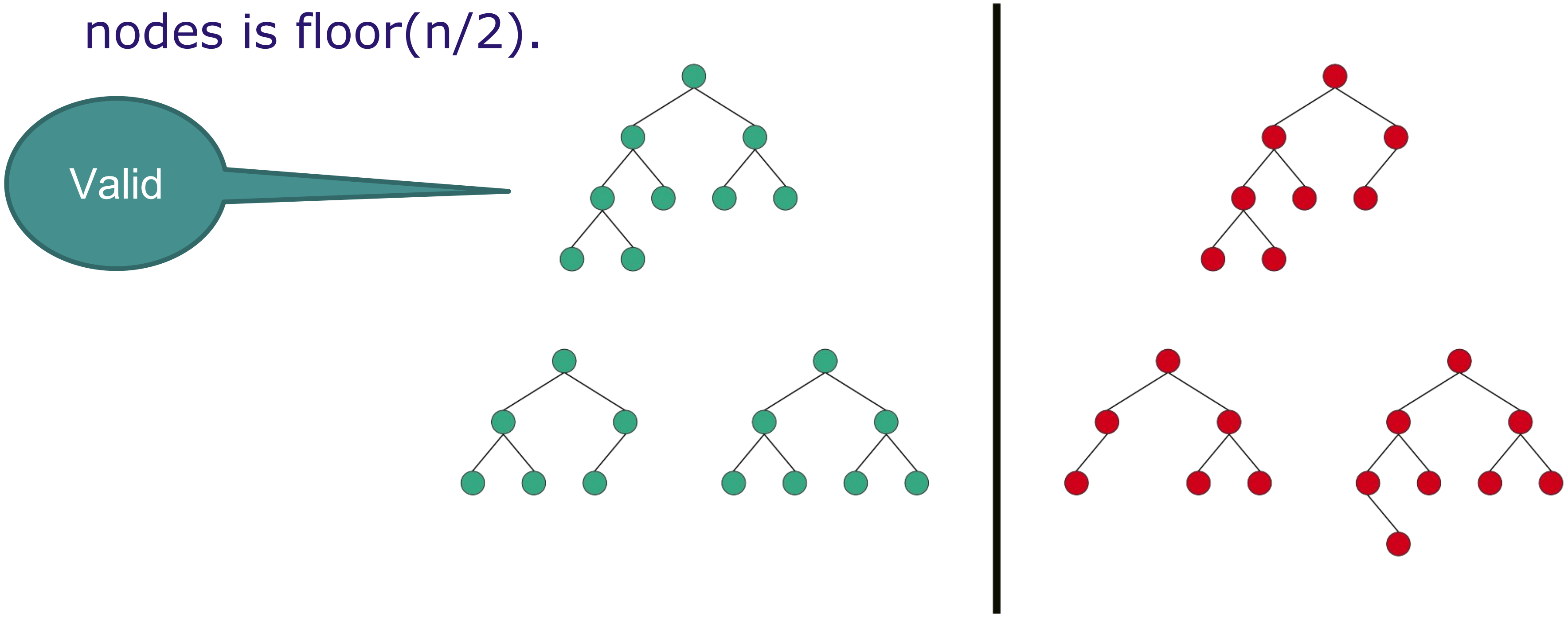
Binary Tree Types

- Full Binary Tree : Full Binary Tree is a Binary Tree in which every node has 0 or 2 children.
- Number of Leaf nodes = Number of Internal nodes + 1
- Minimum No. of Nodes = 2h-1, where h is height of tree.
- Maximum No. of Nodes = 2^{h+1}-1, where h is height of tree.



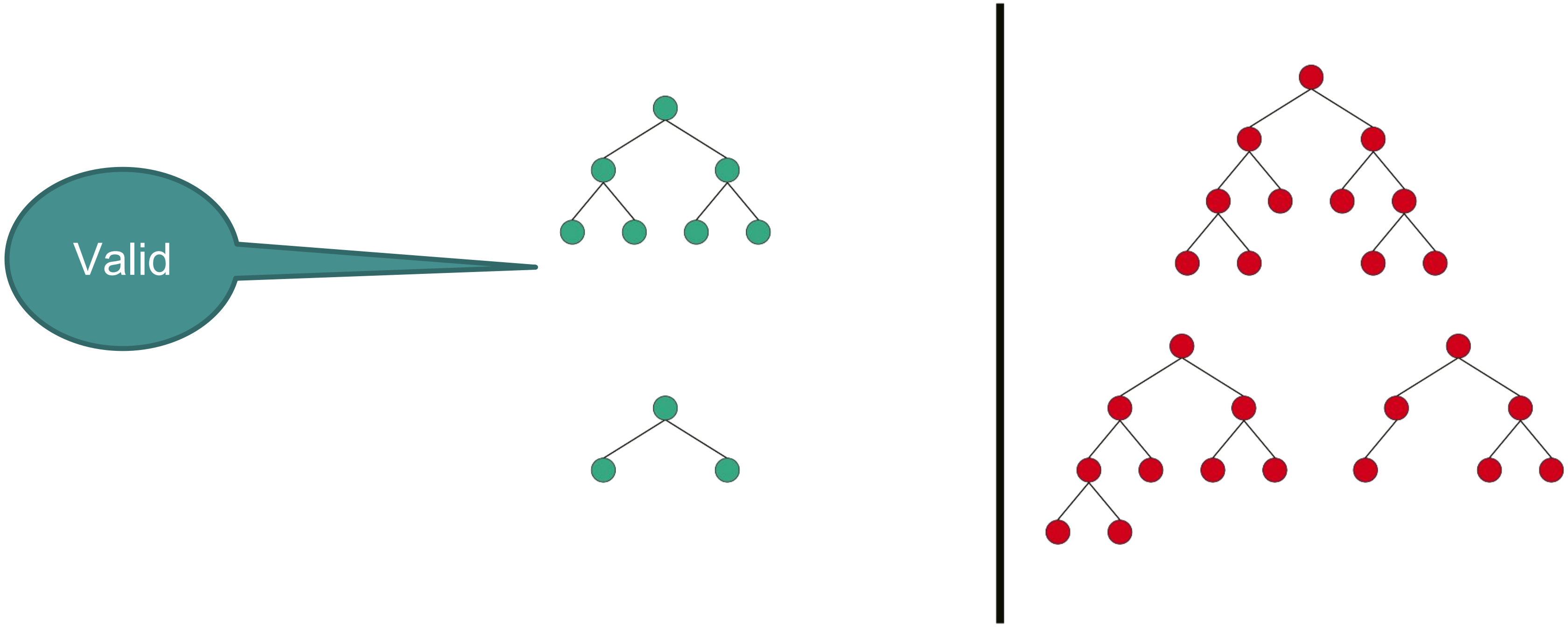
Binary Tree Types

- Complete Binary Tree : Complete Binary Tree has all levels completely filled with nodes except the last level and in the last level, all the nodes are as left side as possible.
- The number of internal nodes in a complete binary tree of n nodes is floor(n/2).



Binary Tree Types

- Perfect Binary Tree : Perfect Binary Tree is a Binary Tree in which all internal nodes have 2 children and all the leaf nodes are at the same depth or same level.
- A perfect binary tree with l leaves has n = 2l-1 nodes.

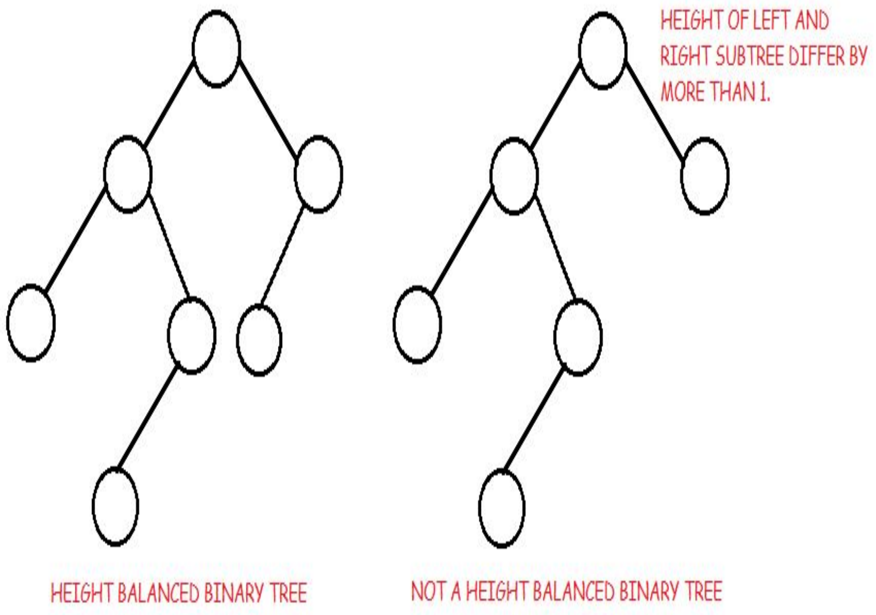


Binary Tree Types

- **Balanced Binary Tree :** Balanced binary tree: A binary tree is height balanced if it satisfies the following constraints:
  - The left and right subtrees' heights differ by at most one, AND
  - The left subtree is balanced, AND
  - The right subtree is balanced

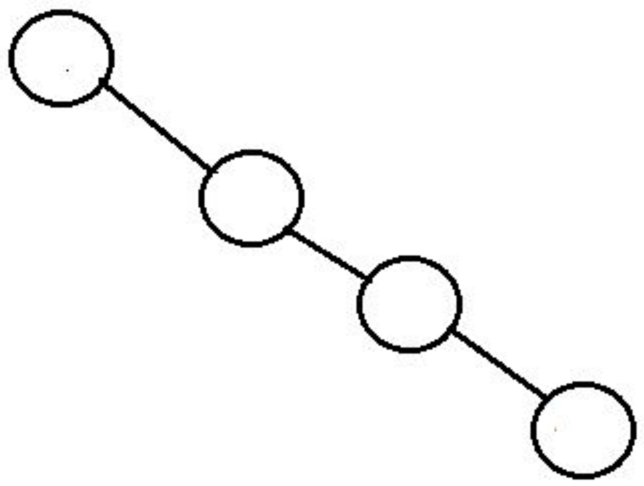
- An empty tree is height balanced.

- The height of a balanced binary tree is O(Log n) where n is number of nodes.


HEIGHT OF LEFT AND RIGHT SUBTREE DIFFER BY MORE THAN 1.
HEIGHT BALANCED BINARY TREE
NOT A HEIGHT BALANCED BINARY TREE

Binary Tree Types

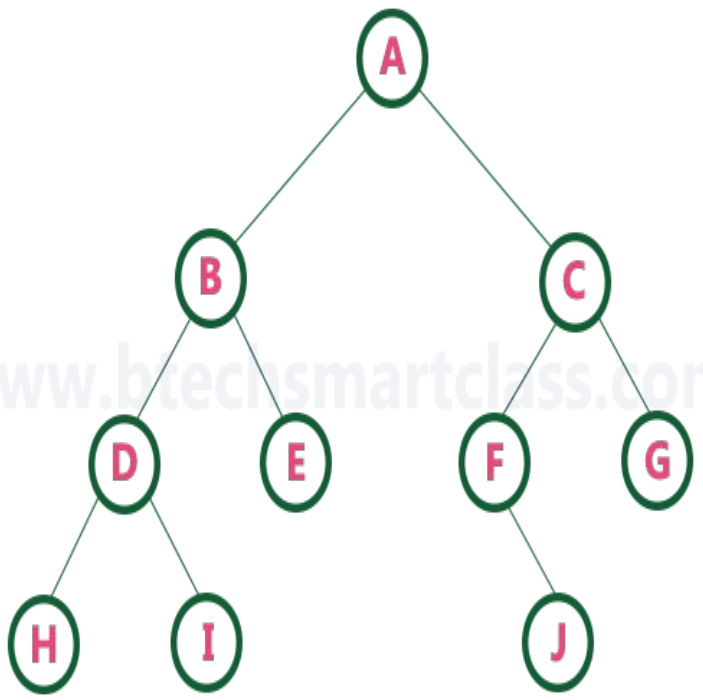
- Degenerate Tree : It is a tree is where each parent node has only one child node. It behaves like a linked list.
- Height of a Degenerate Binary Tree is equal to Total number of nodes in that tree.

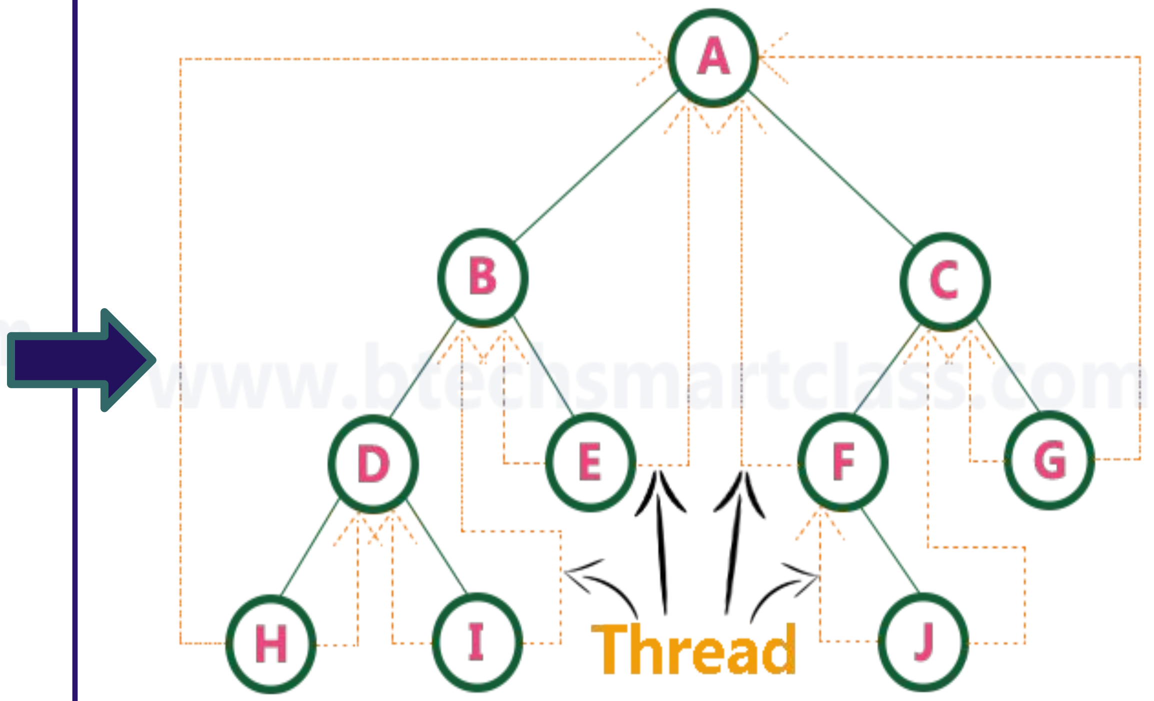


Threaded Binary Tree

- Threaded Binary Tree is a binary tree in which all left child pointers that are NULL points to its in-order predecessor, and all right child pointers that are NULL points to its in-order successor.
- If there are 2N number of reference fields, then N+1 number of reference fields are filled with NULL (N+1 are NULL out of 2N).
- Threaded Binary Tree makes use of NULL pointers to improve its traversal process. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as thread.
- Single Threaded Binary Tree : Right Null Pointers points to inorder successor
- Double Threaded Binary Tree : Left and Right Null Pointers points to inorder predecessor and inorder successor respectively.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

Threaded Binary Tree





❖ **In-order traversal :**

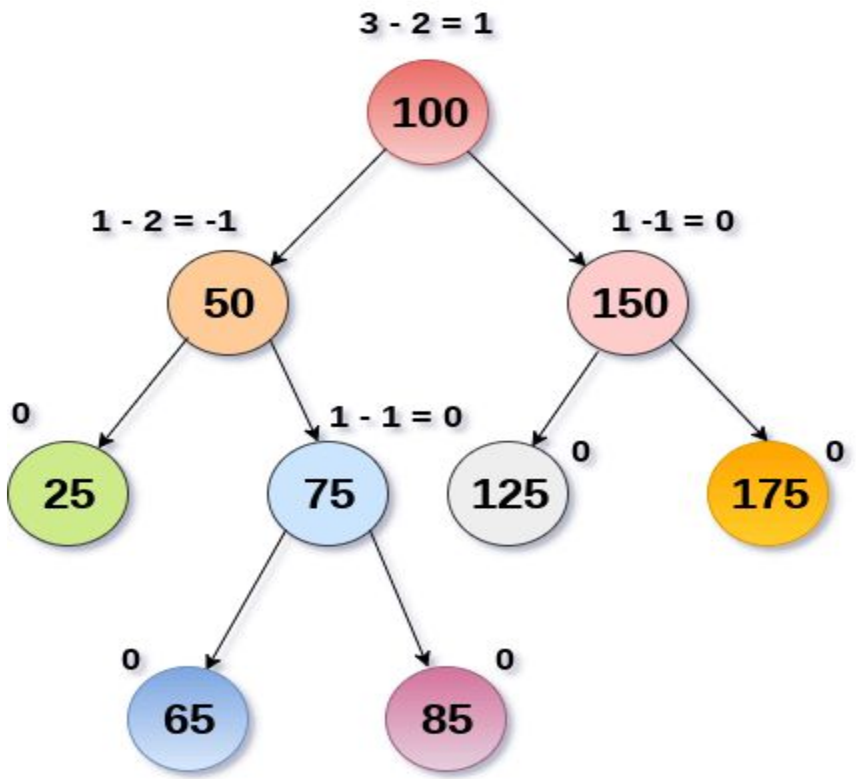
H - D - I - B - E - A - F - J - C - G

AVL Tree

- AVL (GM Adelson - Velsky and EM Landis, 1962) Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

- Balance Factor of Node = Height of Left Subtree – Height of Right Subtree

- If balance factor of every node is between -1 to 1, then the tree is AVL Tree.



AVL Tree

AVL Tree

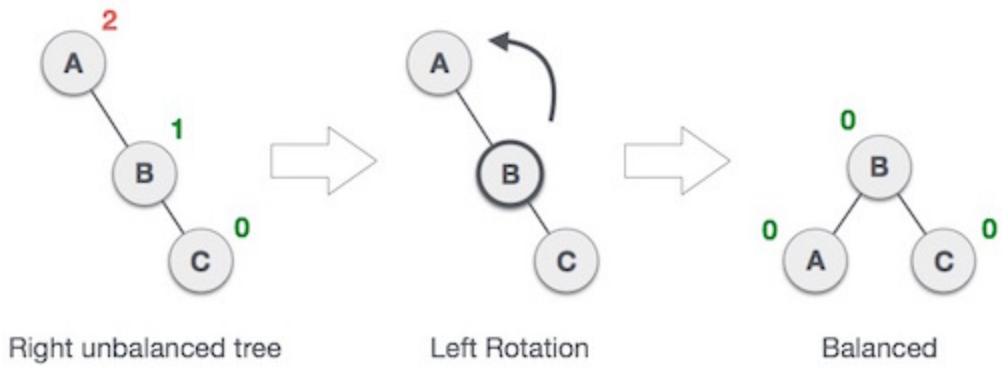
- Complexity of AVL Tree

Algorithm/Operation	Average case	Worst case
Space	o(n)	o(n)
Search	o(log n)	o(log n)
Insert	o(log n)	o(log n)
Delete	o(log n)	o(log n)

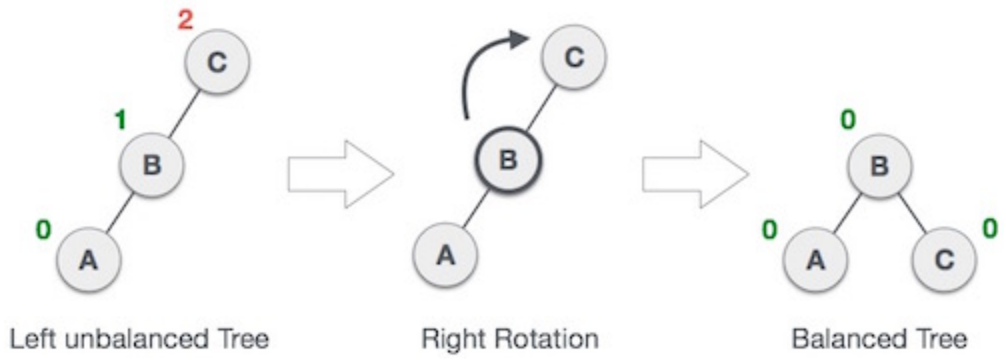
- AVL Rotations
  - Left rotation
  - Right Rotation
  - Left Right Rotation
  - Right Left Rotation
- The first two rotations Left and Right are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2

AVL Tree

- Left Rotation : If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.

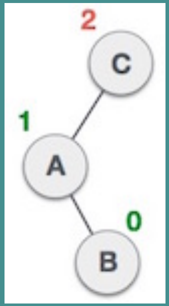
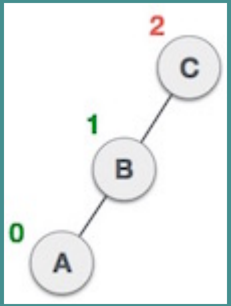
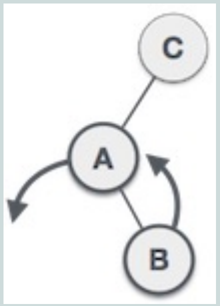
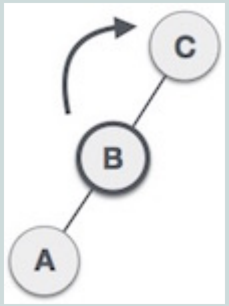


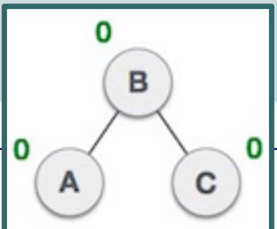
- Right Rotation : AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



AVL Tree

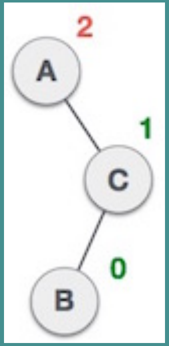
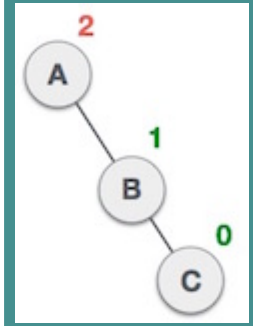
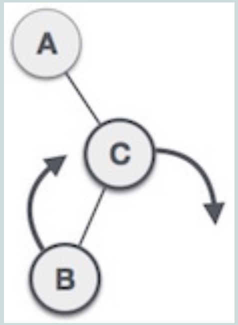
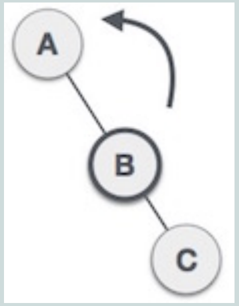
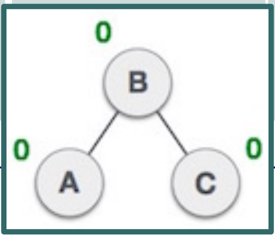
- Left - Right Rotation : A left-right rotation is a combination of left rotation followed by right rotation. It is performed when node is inserted into the right subtree of left subtree.

	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.		Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.		We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.



AVL Tree

- Right - Left Rotation : A right-left rotation is a combination of right rotation followed by left rotation. It is performed when node is inserted into the left subtree of right subtree.

	A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2.		Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation.
	First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**.	 	A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**.