Data Structure

Dr. Ghanshyam Rathod, Assistant Professor
IT and Computer Science

Parul® University

DIGITAL LEARNING CONTENT

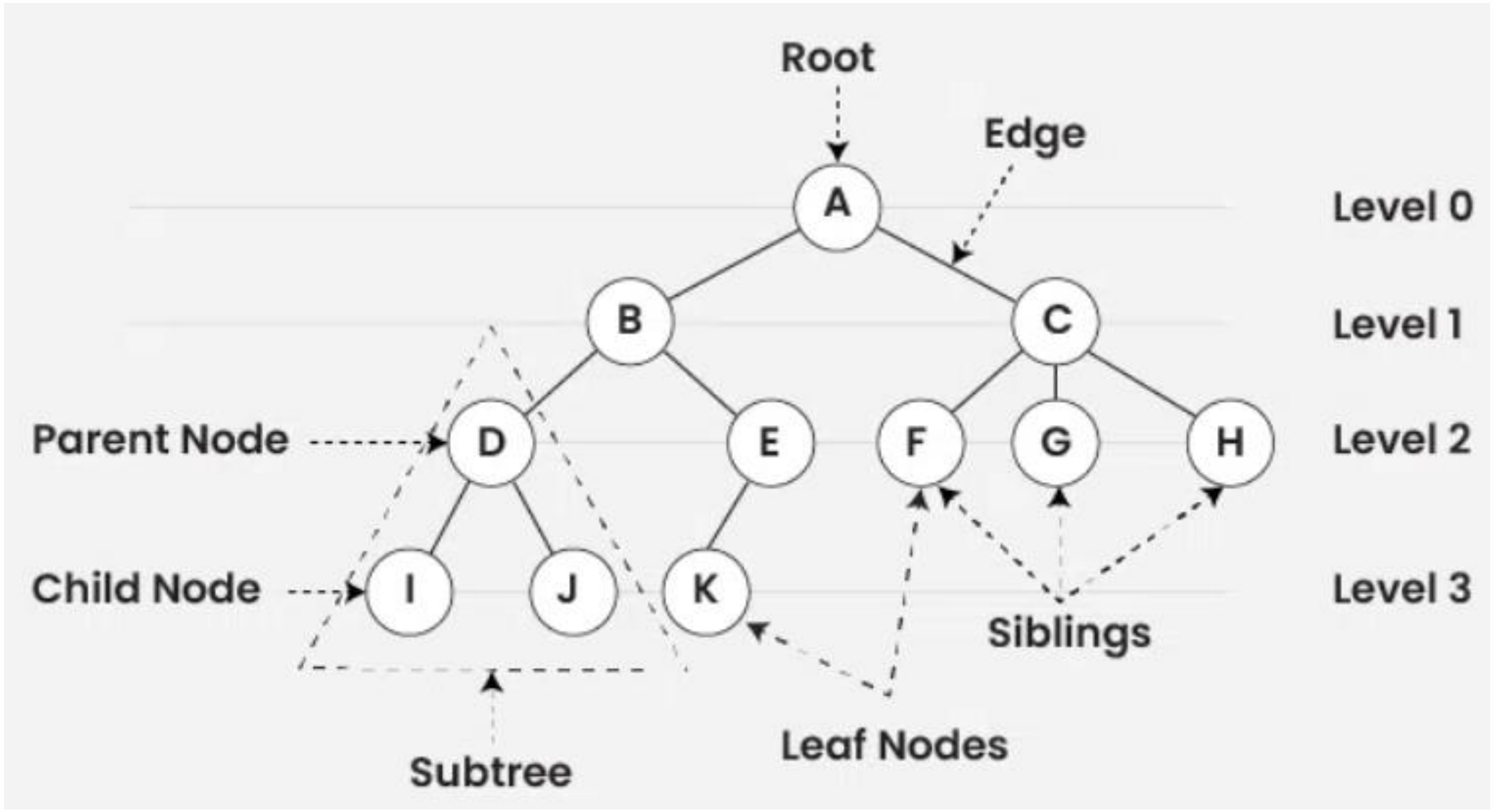
CHAPTER-4

Trees

Concept of Trees

- A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.
- Tree data structure is a specialized data structure to store data in hierarchical manner. Tree has roots, branches, and leaves connected to each other.
- A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more subtrees.
- A tree has following general properties:
  - One node is distinguished as a root;
  - Every node (exclude a root) is connected by a directed edge from exactly one other node; A direction is: parent -> children

Concept of Trees



Terminologies of Trees

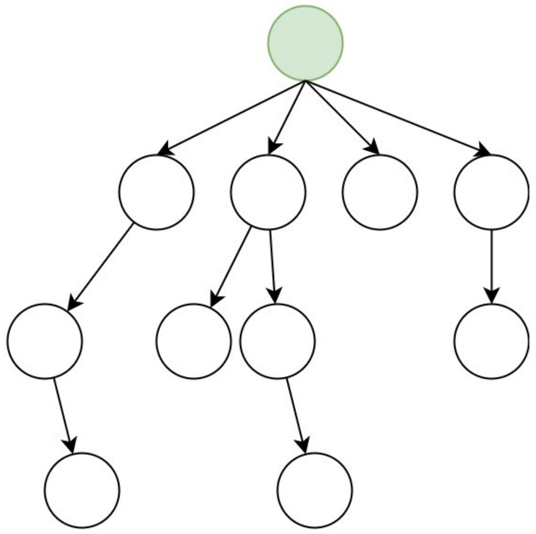
- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. {B} is the parent node of {D, E}.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: {D, E} are the child nodes of {B}.
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A} is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. {I, J, K, F, G, H} are the leaf nodes of the tree.
- **Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.

Terminologies of Trees

- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. {A,B} are the ancestor nodes of the node {E}
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. {D,E} are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level 0.
- **Internal node:** A node with at least one child is called Internal Node.
- **Subtree:** Any node of the tree along with its descendant.

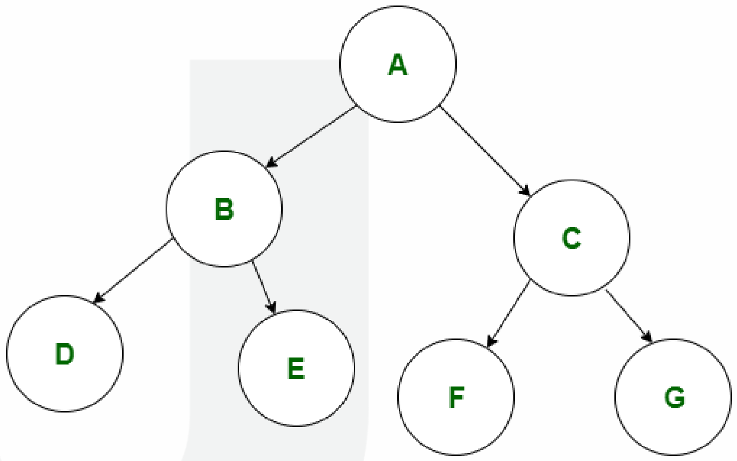
General Tree

- A general tree is a tree where each node may have zero or more children (a binary tree is a specialized case of a general tree).
- In general tree, there is no limitation on the degree of a node. The topmost node of a general tree is called the root node.
- There are many subtrees in a general tree. In a general tree, each node has in-degree(number of parent nodes) one and maximum out-degree(number of child nodes) n.
- General trees are used to model applications such as file systems.


General Tree

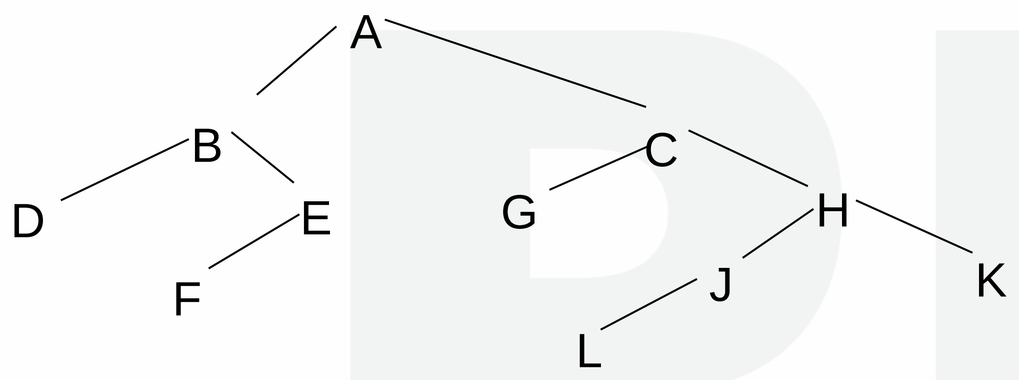
Binary Tree

- A binary tree is the specialized version of the General tree. A binary tree is a tree in which each node can have at most two nodes.
- In a binary tree, there is a limitation on the degree of a node because the nodes in a binary tree can't have more than two child node(or degree two).
- The topmost node of a binary tree is called root node and there are mainly two subtrees one is left-subtree and another is right-subtree.
- Unlike the general tree, the binary tree can be empty. Unlike the general tree, the subtree of a binary tree is ordered because the nodes of a binary tree can be ordered according to specific criteria.



Binary Tree

Example of Binary Tree



- **Root Node:** A
- **Left Sub Tree Nodes:** B, D, E, F
- **Right Sub Tree Nodes:** C, G, H, J, K, L
- **No of Nodes = 2:** A, B, C, H
- **No of Nodes = 1:** E, J
- **No of Nodes = 0:** D, G, F, K, L

- B is a left successor and C is a right successor of the node A.
- The left subtree of the root A consists of the nodes B, D, E and F, and the right subtree of A consists of the nodes C, G, H, J, K and L.

Difference between General Tree and Binary Tree

General Tree	Binary Tree
General tree is a tree in which each node can have many children or nodes.	Whereas in binary tree, each node can have at most two nodes.
The subtree of a general tree do not hold the ordered property.	While the subtree of binary tree hold the ordered property.
In data structure, a general tree can not be empty.	While it can be empty.
In general tree, a node can have at most **n (number of child nodes)** nodes.	While in binary tree, a node can have at most **2(number of child nodes)** nodes.
In general tree, there is no limitation on the degree of a node.	There is limitation on the degree of a node because the nodes in a binary tree can't have more than two child node.
There is either zero subtree or many subtree.	There are mainly two subtree: Left-subtree and Right-subtree.

Storage Representation of Binary Tree: Using Array

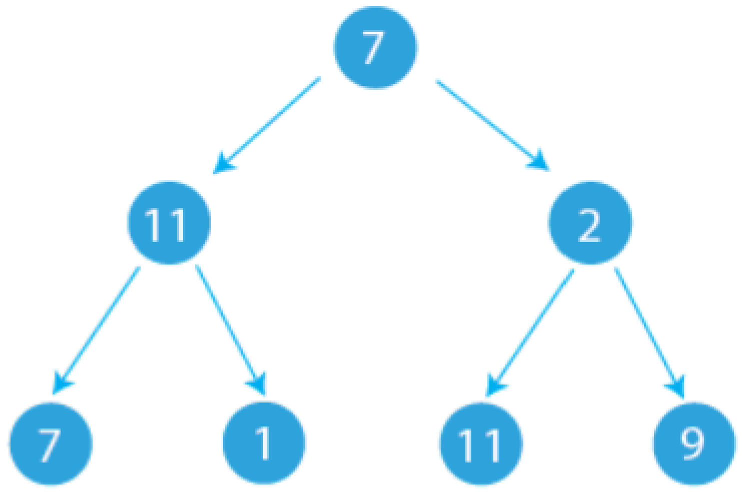
- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
- Rules to determine the locations of the parent, left child and right child of any node I in the binary tree using array.

1. Store the root node in the first location of the array TREE
2. If a node is present in the location i of the array TREE then you can find its left child in the location 2 * i + 1
3. If a node is present in the location i of the array TREE then you can find its right child in the location 2 * i + 2.
4. If a node is present in the location i of the array TREE then you can find its Parent in the location at floor((i-1)/2).

Storage Representation of Binary Tree: Using Array

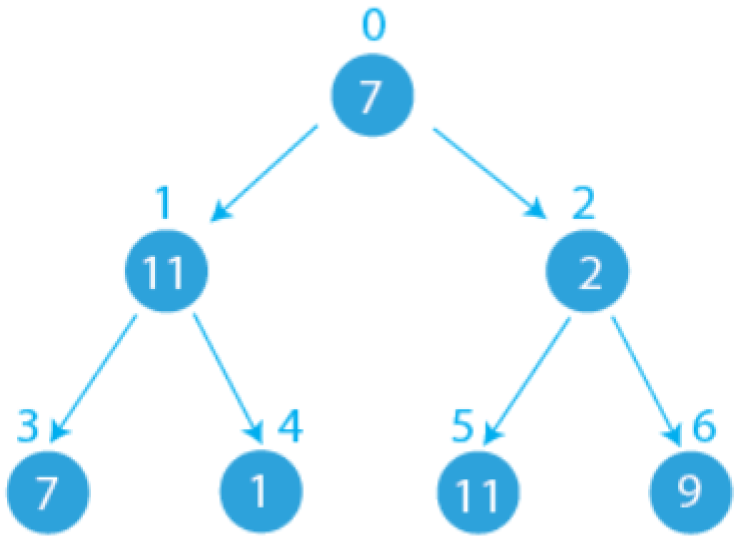
- For the array representation of binary tree, we will form an array of size 2*n+1 size where n is the number of nodes the binary tree. Now we will move step by step for the array representation of binary tree.

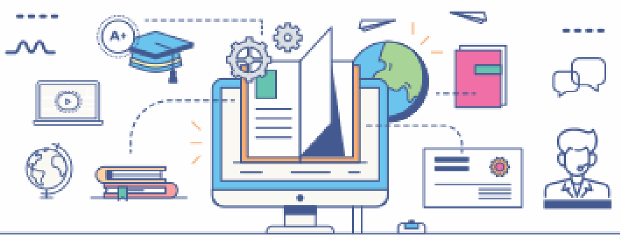
1. First, suppose we have a binary tree with seven nodes



Storage Representation of Binary Tree: Using Array

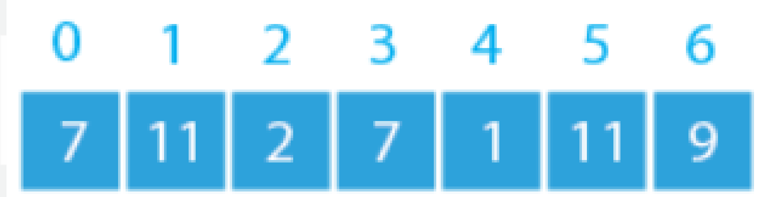
2. Now, for the array representation of binary tree, we have to give numbering to the corresponding nodes. The standard form of numbering the nodes is to start from the root node and move from left to right at every level. After numbering the tree and nodes will look like this:





Storage Representation of Binary Tree: Using Array

3. Now as we have numbered from zero you can simply place the corresponding number in the matching index of an array then the array will look like this:



0	1	2	3	4	5	6
7	11	2	7	1	11	9

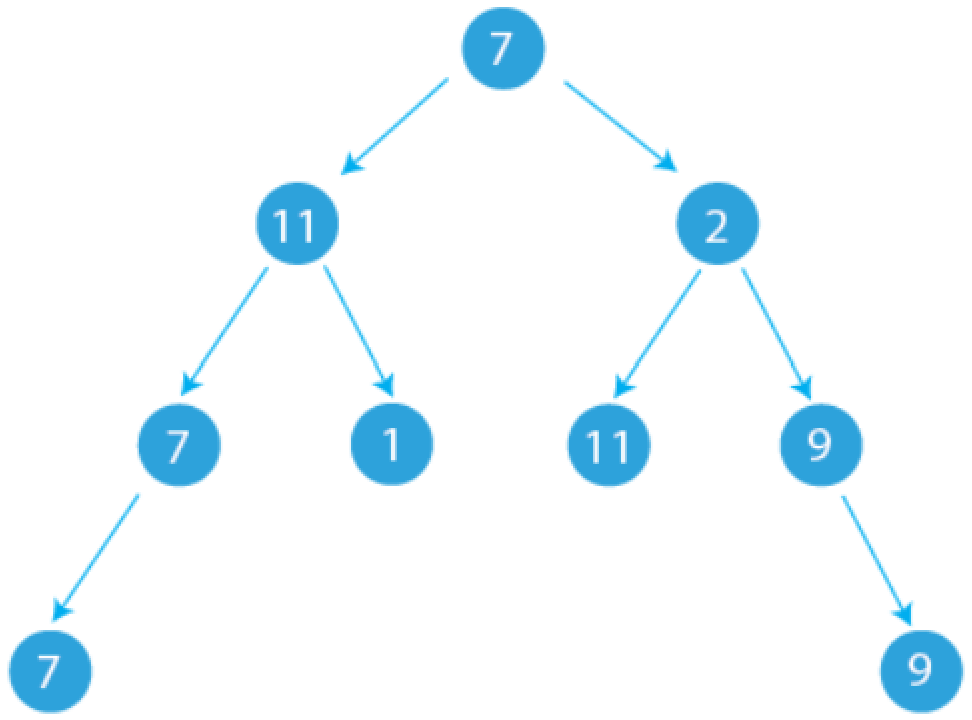
- If i = 1, then its left_child will be at (2 * 1) + 1 = 3 mens ‘7’, and its right_child will be at (2 * 1) + 2 = 4 means ‘1’.
- The Parent of i = 1 will be floor((1-1)/2) = 0 means ‘7’

4. That is the array representation of binary tree but this is the easy case as every node has two child so what about other cases? We will discuss other cases below.

Storage Representation of Binary Tree: Using Array

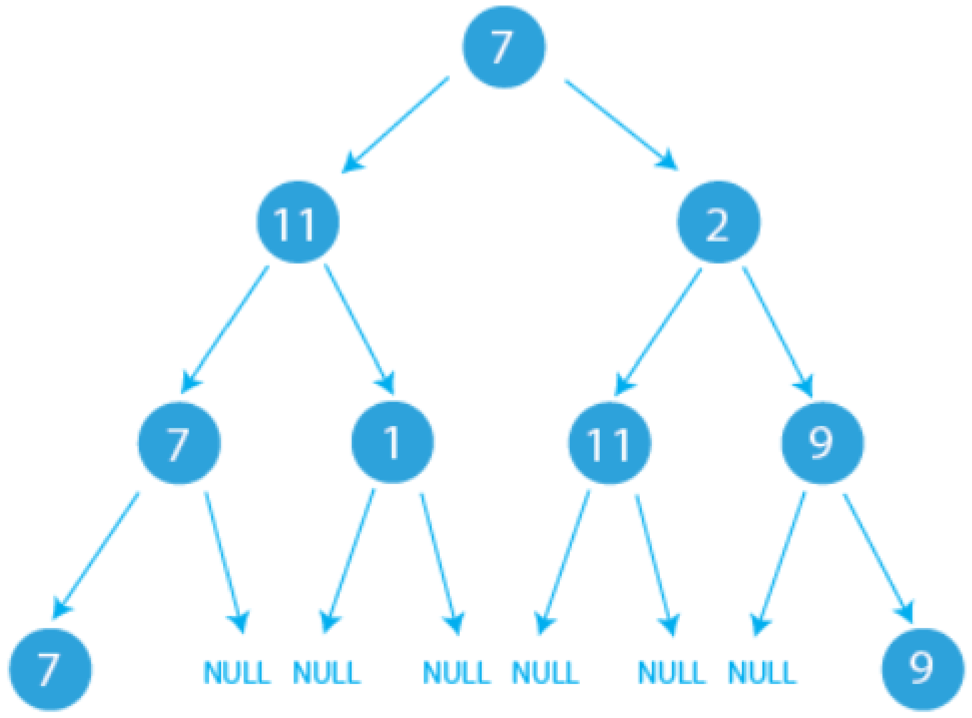
5. In these cases, we will discuss the scenarios for the array representation of binary tree where there the lead node is not always on the last level.

6. So, consider the binary tree given:



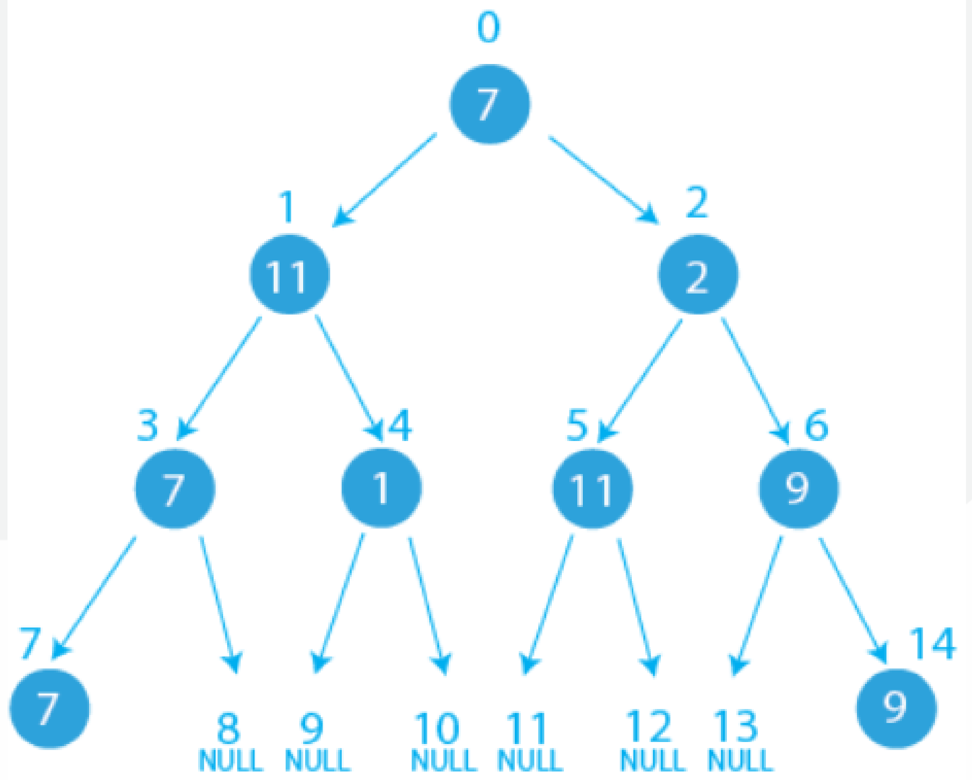
Storage Representation of Binary Tree: Using Array

7. While giving a number you are stuck with the cases where you encounter a leaf node so just make the child node of the leaf nodes as NULL then the tree will look like this:



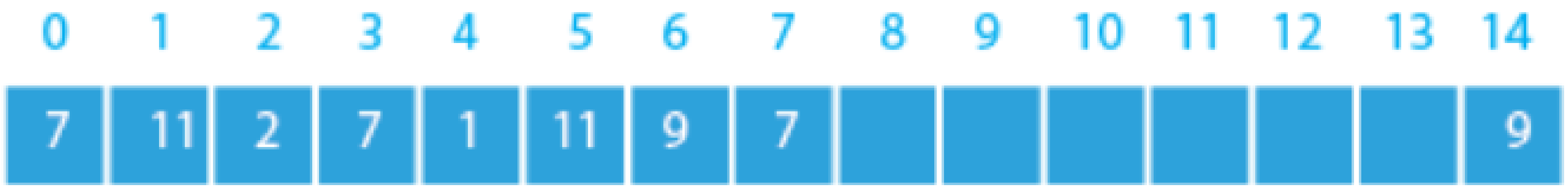
Storage Representation of Binary Tree: Using Array

8. Now just number the nodes as done above for the array representation of binary tree after that the tree will look like this:



Storage Representation of Binary Tree: Using Array

9. Now we have the number on each node we can easily use the tree for array representation of binary tree and the array will look like this:

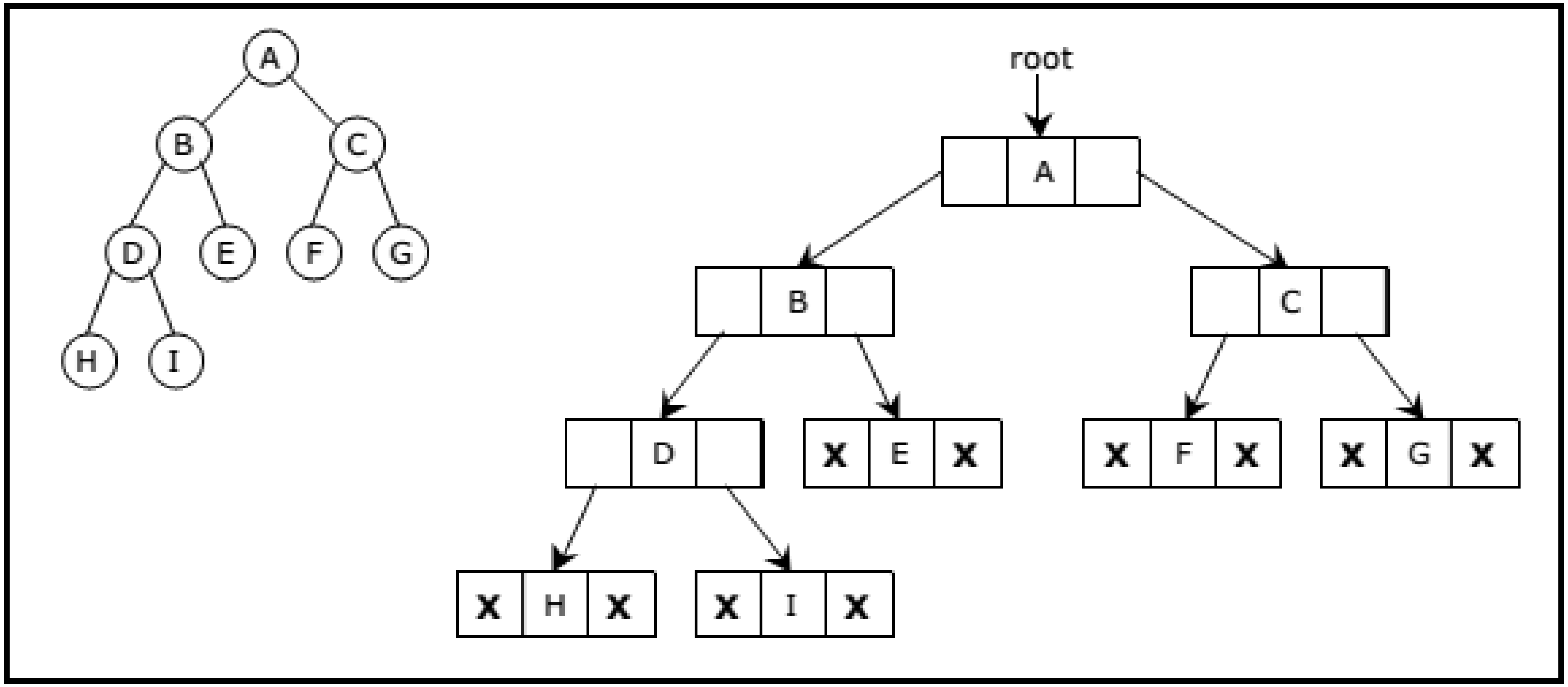


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
7	11	2	7	1	11	9	7							9

Storage Representation of Binary Tree: Using Linked List

- In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non contiguous memory locations and linked together by inheriting parent child relationship like a tree.
- Every node contains three parts : pointer to the left node, data element and pointer to the right node.
- Each binary tree has a root pointer which points to the root node of the binary tree.
- In an empty binary tree, the root pointer will point to null.
  - INFO[K] contains the data at the node N
  - LEFT[K] contains the location of the left child of node N
  - RIGHT[K] contains the location of the right child of node N.

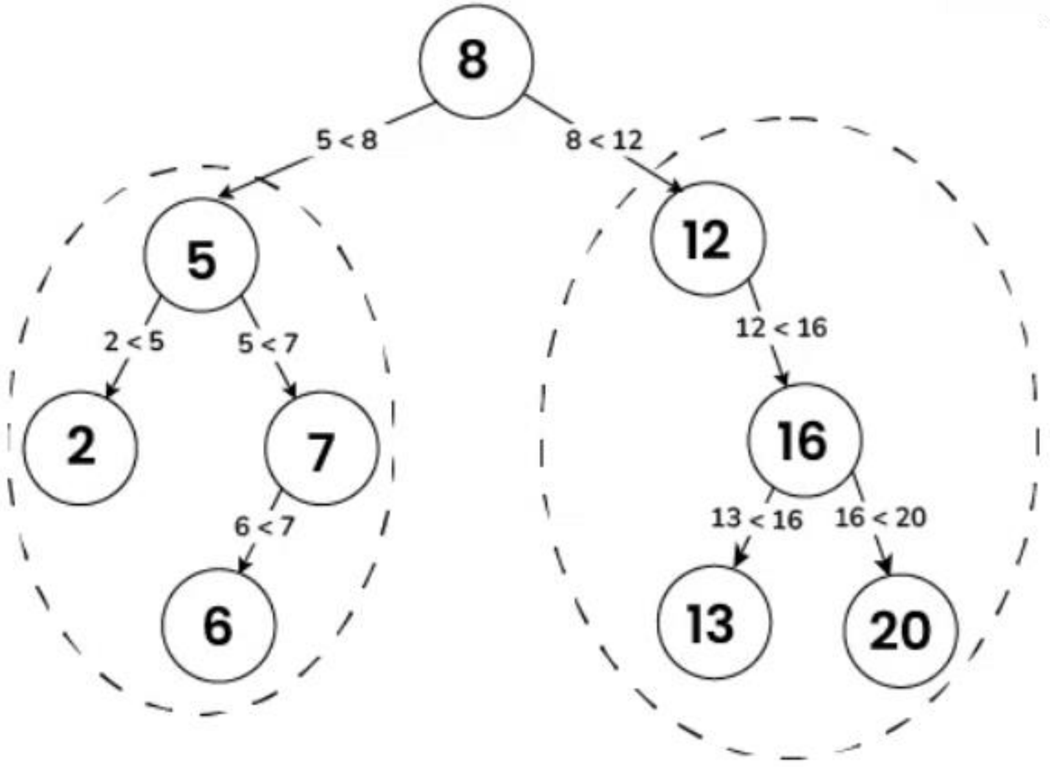
Storage Representation of Binary Tree: Using Linked List



Binary Search Tree:

- Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner.
- Binary search tree follows all properties of binary tree.
- Its left child contains values less than the parent (root) node and the right child contains values greater than the parent node.
- This hierarchical structure allows for efficient Searching, Insertion, and Deletion operations on the data stored in the tree.

Binary Search Tree:



Left subtree contains all elements less than 8

Right subtree contains all elements greater than 8

Operations on Binary Search Tree:

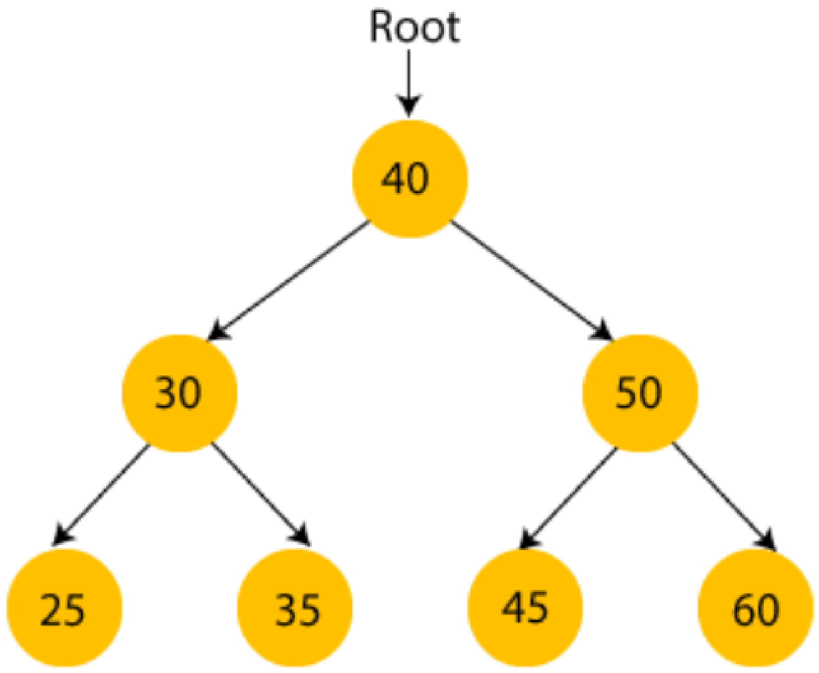
- Create: creates an empty tree.
- Search: to find or locate a specific element or node in a data structure.
- Insert: insert a node in the tree.
- Delete: deletes a node from the tree.
- Inorder: in-order traversal of the tree.
- Preorder: pre-order traversal of the tree.
- Postorder: post-order traversal of the tree.

Create Binary Search Tree:

- A binary search tree follows some order to arrange the elements.
- In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node.
- This rule is applied recursively to the left and right subtrees of the root.
- Let's understand the concept of Binary search tree with an example.

Create Binary Search Tree:

- In this figure, you can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.
- Similarly, you can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

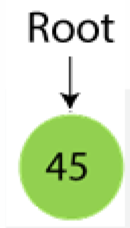


Example of Creating Binary Search Tree:

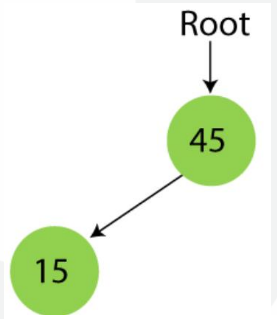
- Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50
  - First, we have to insert 45 into the tree as the root of the tree.
  - Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
  - Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.
- Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Example of Creating Binary Search Tree:

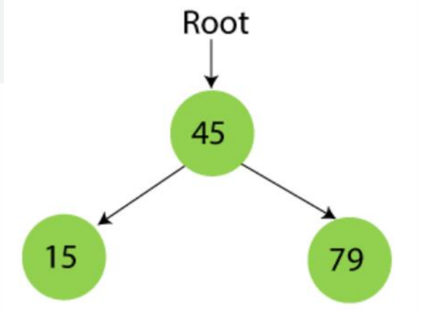
- Step-1: Insert 45



- Step-2: Insert 15
  - As 15 is smaller than 45, so insert it as the root node of the left subtree.



- Step-3: Insert 79
  - As 79 is greater than 45, so insert it as the root node of the right subtree.

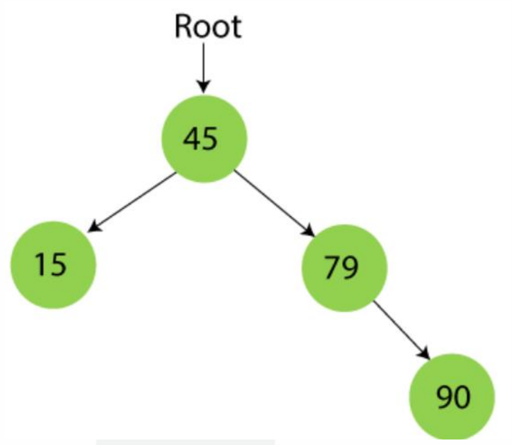


Parul® University

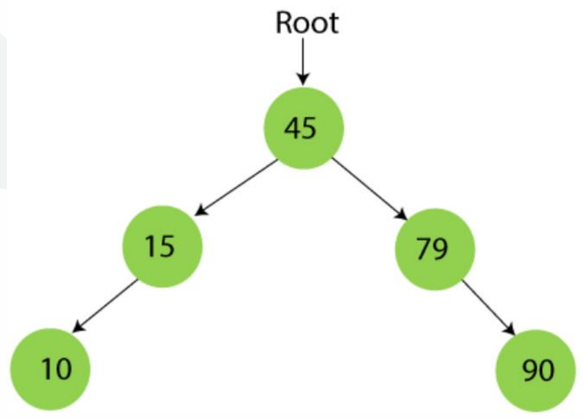
DIGITAL LEARNING CONTENT

Example of Creating Binary Search Tree:

- Step-4: Insert 90
  - 90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.

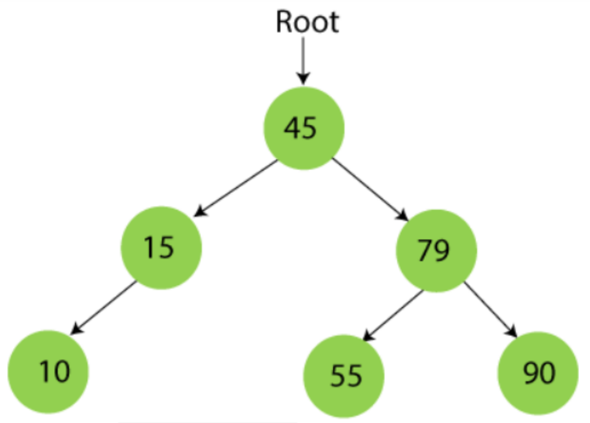


- Step-5: Insert 10
  - 10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.

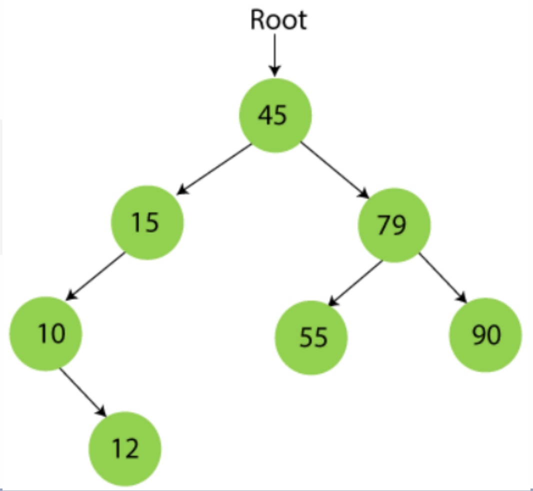


Example of Creating Binary Search Tree:

- Step-6: Insert 55
  - 55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



- Step-7: Insert 12
  - 12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.

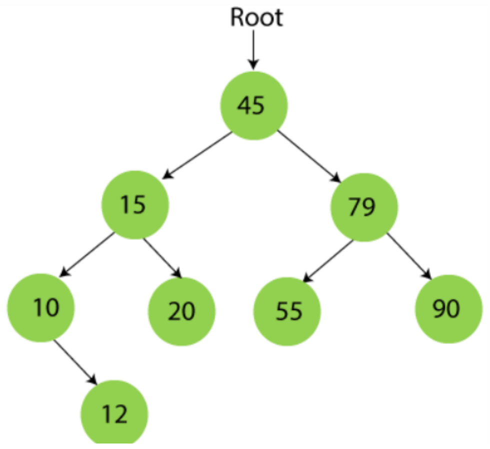


Parul® University

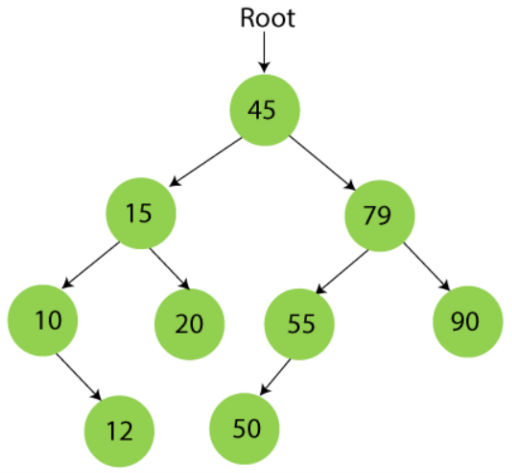
DIGITAL LEARNING CONTENT

Example of Creating Binary Search Tree:

- Step-8: Insert 20
  - 20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



- Step-9: Insert 50
  - 50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.

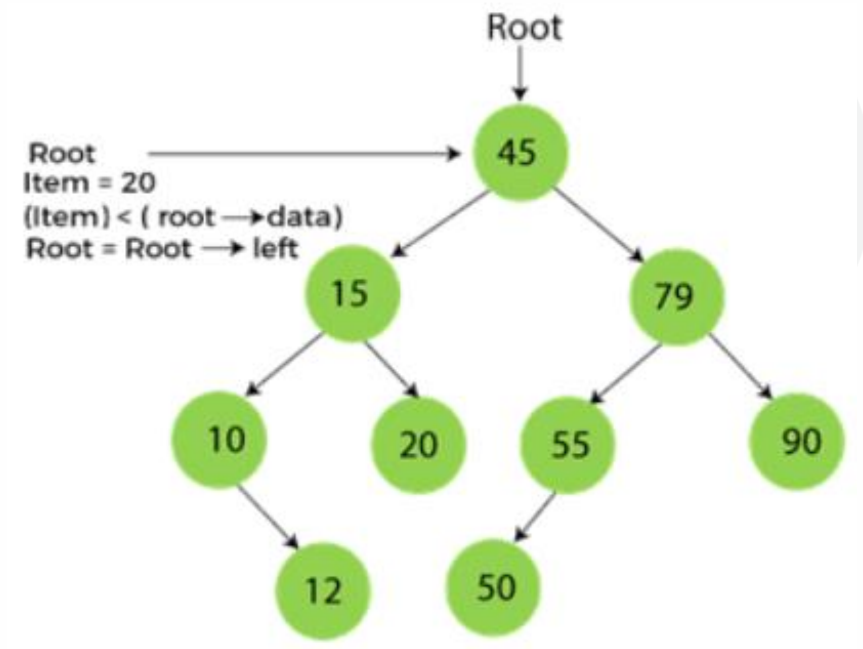


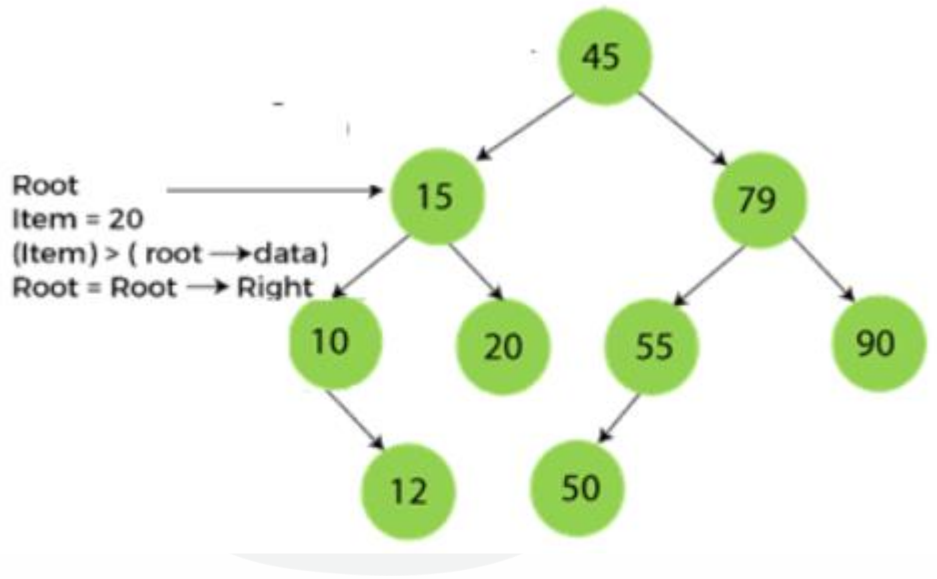
Searching in Binary search tree

- Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows

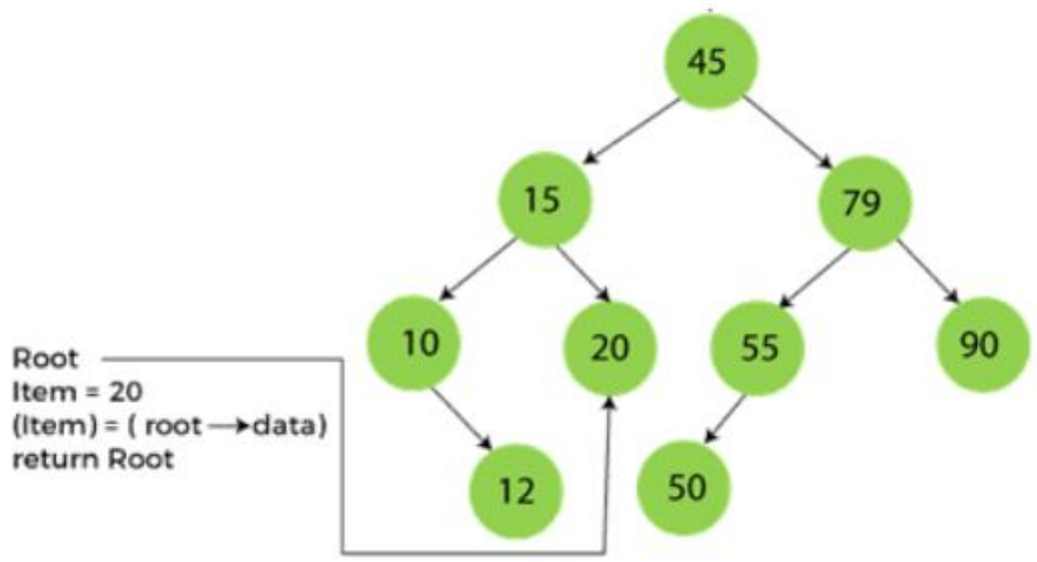
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL

Searching in Binary search tree





Searching in Binary search tree



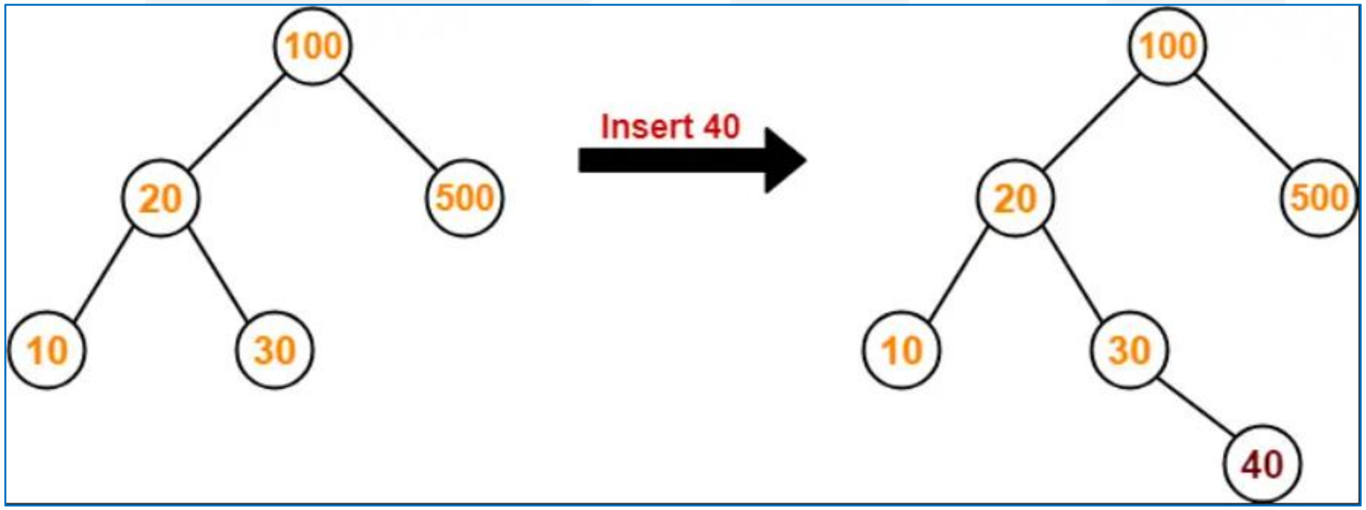
Insertion in Binary search tree

- Insertion Operation is performed to insert an element in the Binary Search Tree.
- The insertion of a new key always takes place as the child of some ***leaf node***.
- For finding out the suitable leaf node,

  1. Search the key to be inserted from the root node till some leaf node is reached.
  2. Once a leaf node is reached, insert the key as child of that leaf node.

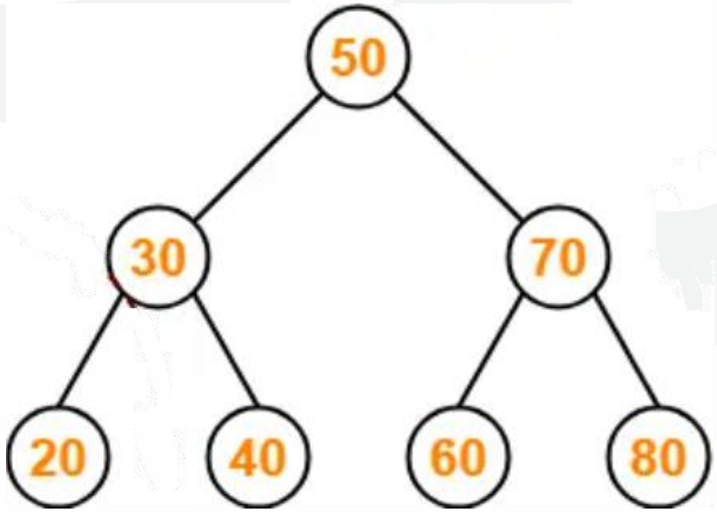
Insertion in Binary search tree

- Consider the following example where key = 40 is inserted in the given BST,
  1. We start searching for value 40 from the root node 100.
  2. As 40 < 100, so we search in 100's left subtree.
  3. As 40 > 20, so we search in 20's right subtree.
  4. As 40 > 30, so we add 40 to 30's right subtree.



Deletion in Binary search tree

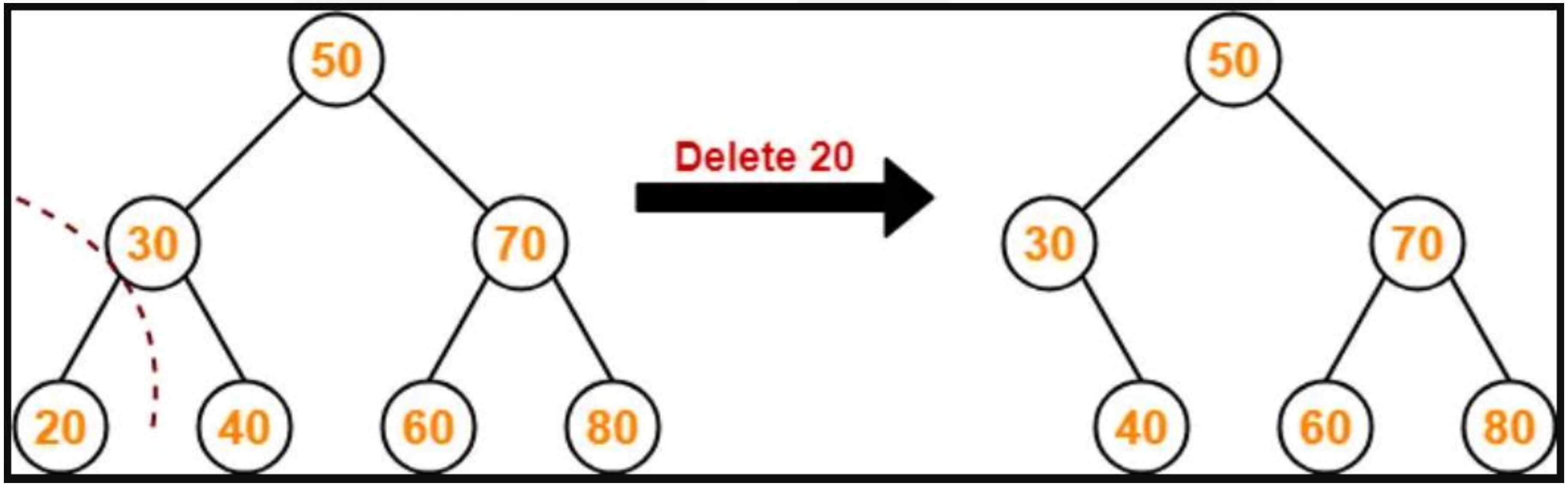
- Deletion Operation is performed to delete a particular element from the Binary Search Tree.
- When it comes to deleting a node from the binary search tree, following three cases are possible



Case-01: Deletion Of A Node Having No Child (Leaf Node):

- Just remove / disconnect the leaf node that is to deleted from the tree.

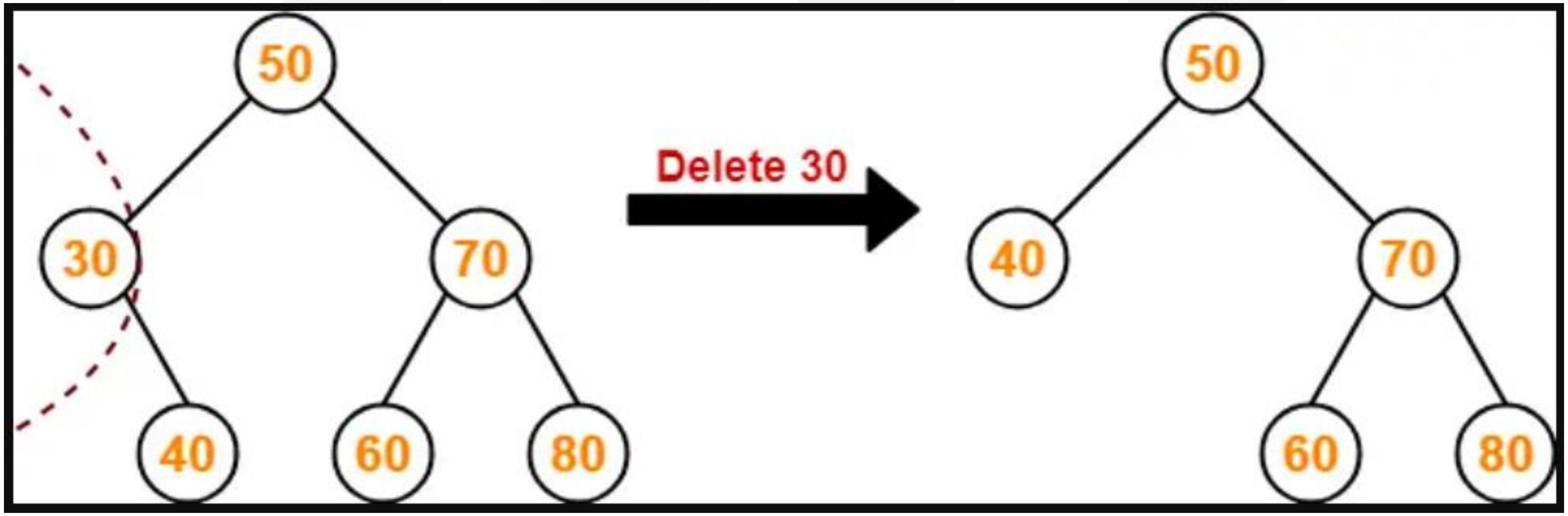
Example: Consider the following example where node with value = 20 is deleted from the BST



Case-02: Deletion Of A Node Having Only One Child:

- Just replace the child of the deleting node with the deleting node.

Example: Consider the following example where node with value = 30 is deleted from the BST

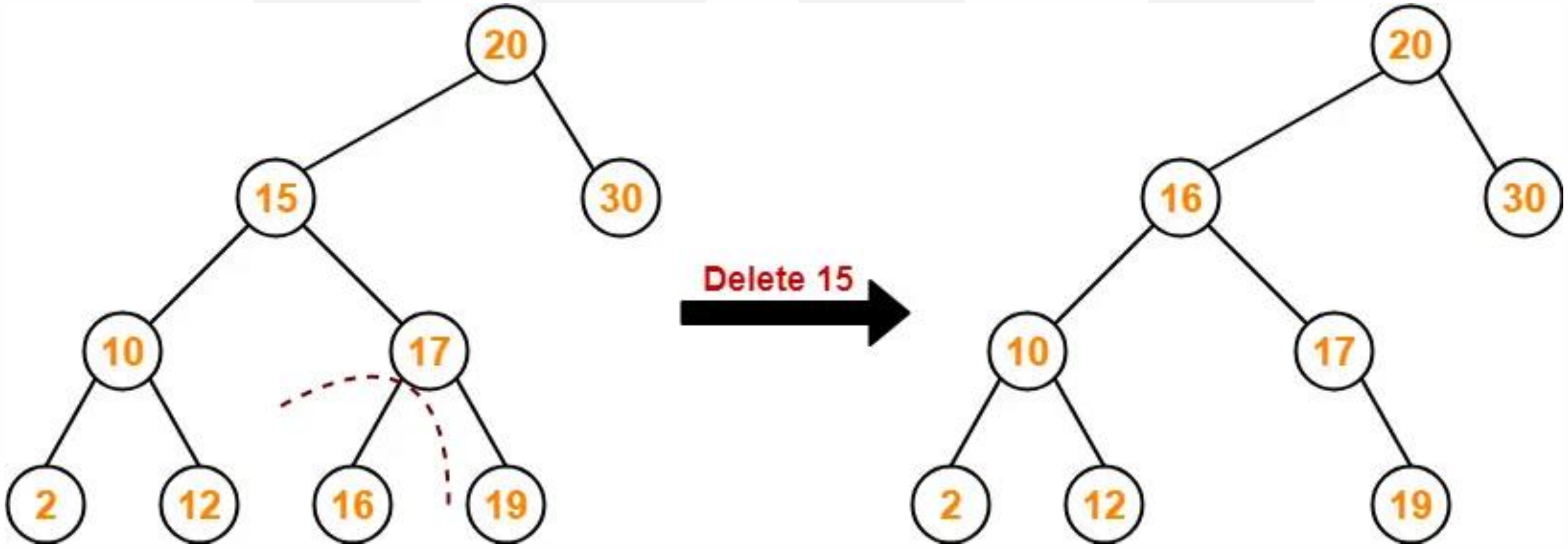


Case-03: Deletion Of A Node Having Two Children:

Method-01:

- Visit to the right subtree of the deleting node.
- Pluck the least value element called as Inorder successor.
- Replace the deleting element with its Inorder successor.

Example: Consider the following example where node with value = 15 is deleted from the BST

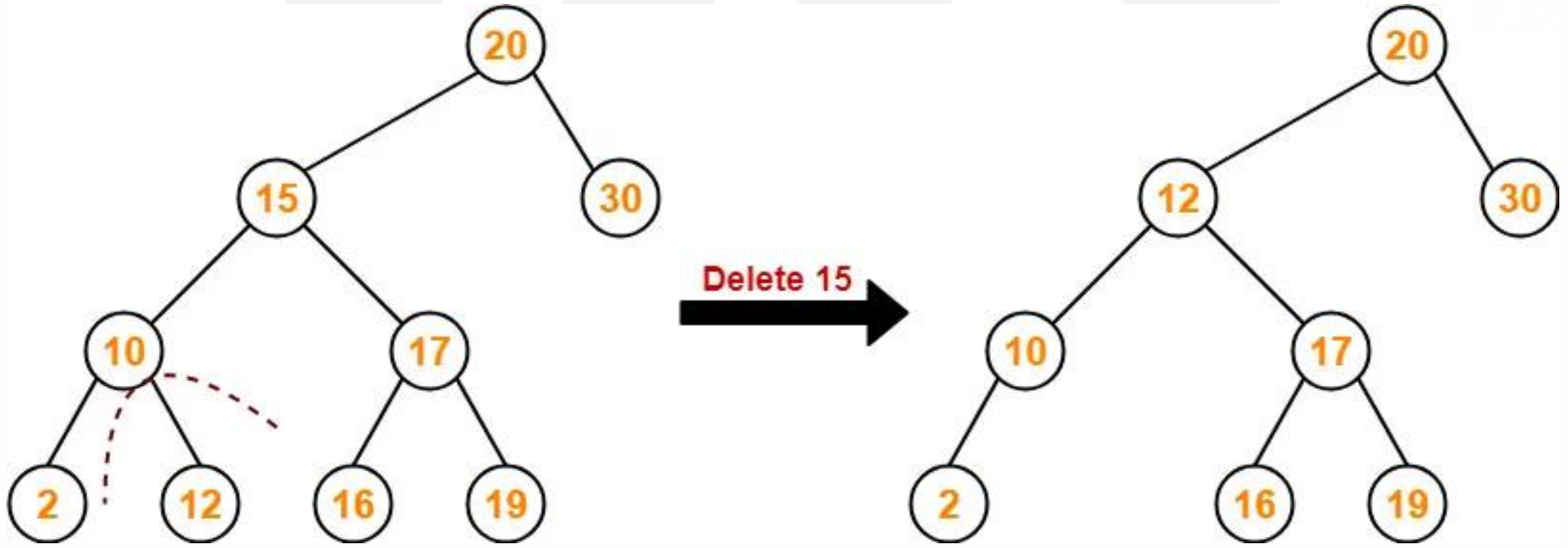


Case-03: Deletion Of A Node Having Two Children:

Method-02:

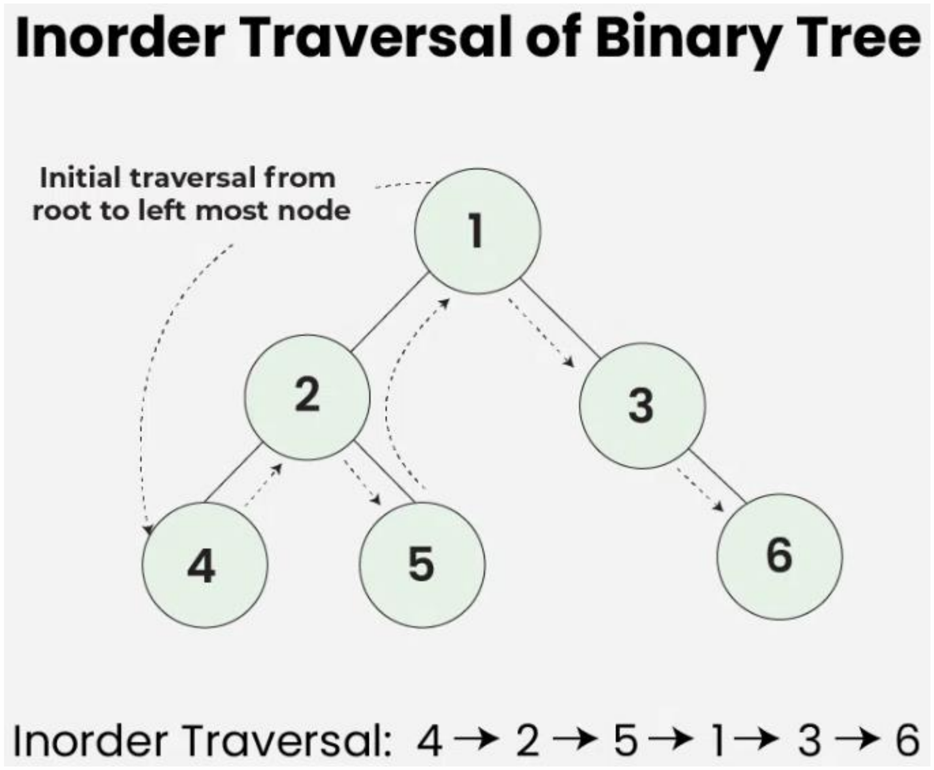
- Visit to the left subtree of the deleting node.
- Pluck the greatest value element called as Inorder predecessor.
- Replace the deleting element with its Inorder predecessor.

Example: In this example where node with value = 15 is deleted from the BST

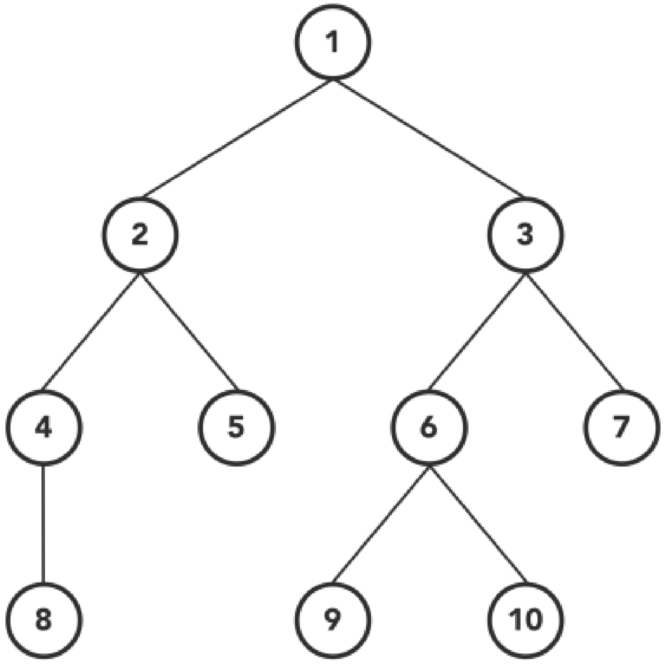


Binary Tree Traversal: In-order (Left-Root-Right)

- Visits all nodes inside the left subtree, then visits the current node before visiting any node within the right subtree..
- **Algorithm:**
  - Traverse the left subtree, i.e., call Inorder (left->subtree)
  - Visit the root.
  - Traverse the right subtree, i.e., call Inorder (right->subtree)

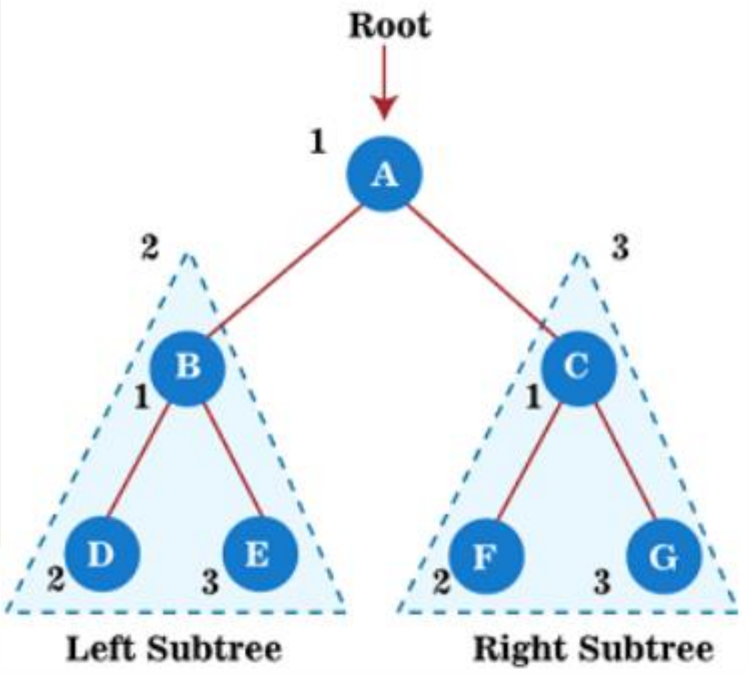

Inorder Traversal of Binary Tree
Initial traversal from root to left most node
1
2
3
4
5
6
Inorder Traversal: 4 → 2 → 5 → 1 → 3 → 6

Binary Tree Traversal: In-order (Left-Root-Right)



Binary Tree Traversal: In-order (Left-Root-Right)

- First, we traverse the left subtree B that will be traversed in inorder. After that, we will traverse the root node A. And finally, the right subtree C is traversed in inorder.
- So, for left subtree B, first, its left subtree D is traversed. Since node D does not have any children, so after traversing it, node B will be traversed, and at last, right subtree of node B, that is E, is traversed. Node E also does not have any children; therefore, the traversal of the left subtree of root node A is completed.

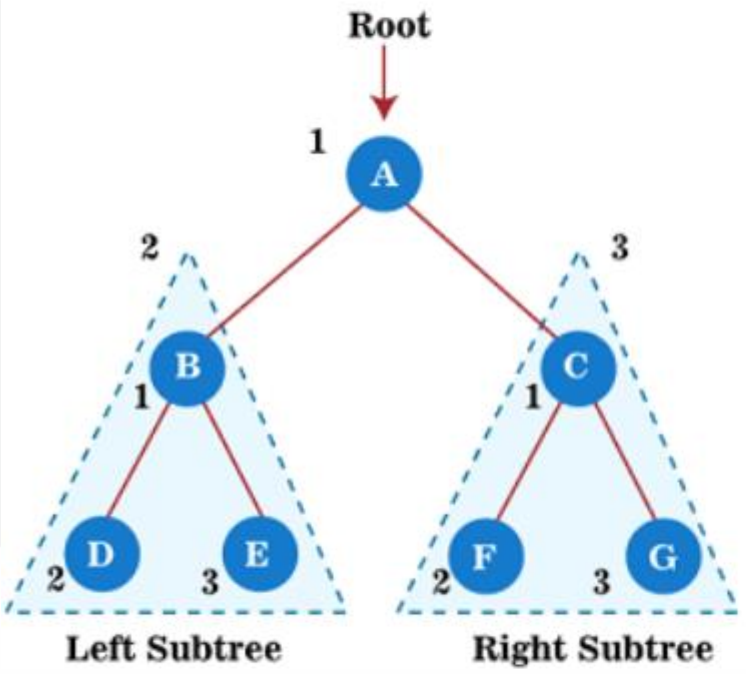


Parul® University

DIGITAL LEARNING CONTENT

Binary Tree Traversal: In-order (Left-Root-Right)

- At last, move towards the right subtree of root node A that is C. So, for right subtree C; first, its left subtree F is traversed. Since node F does not have any children, node C will be traversed, and at last, a right subtree of node C, that is, G, is traversed. Node G also does not have any children; therefore, the traversal of the right subtree of root node A is completed.

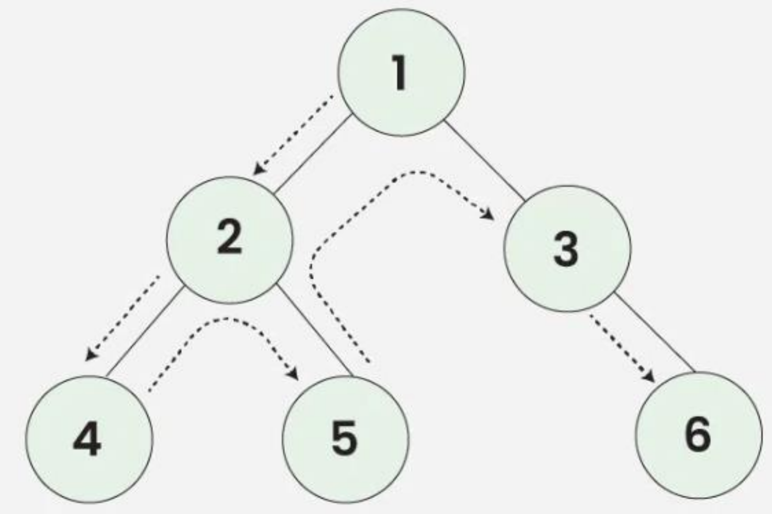


- D → B → E → A → F → C → G

Binary Tree Traversal: Pre-order (Root-Left-Right)

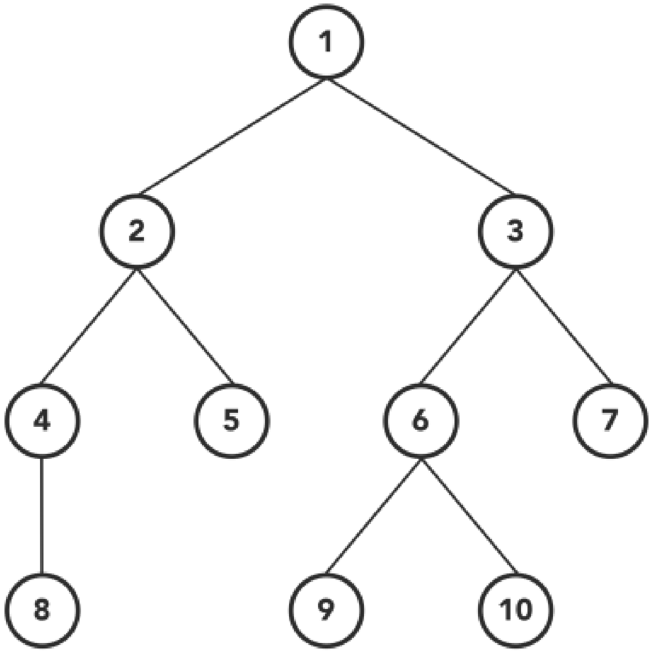
- Visits the current node before visiting any nodes inside left or right subtrees..
- **Algorithm:**
  - Visit the root.
  - Traverse the left subtree, i.e., call Preorder(left->subtree)
  - Traverse the right subtree, i.e., call Preorder(right->subtree)

Preorder Traversal of Binary Tree



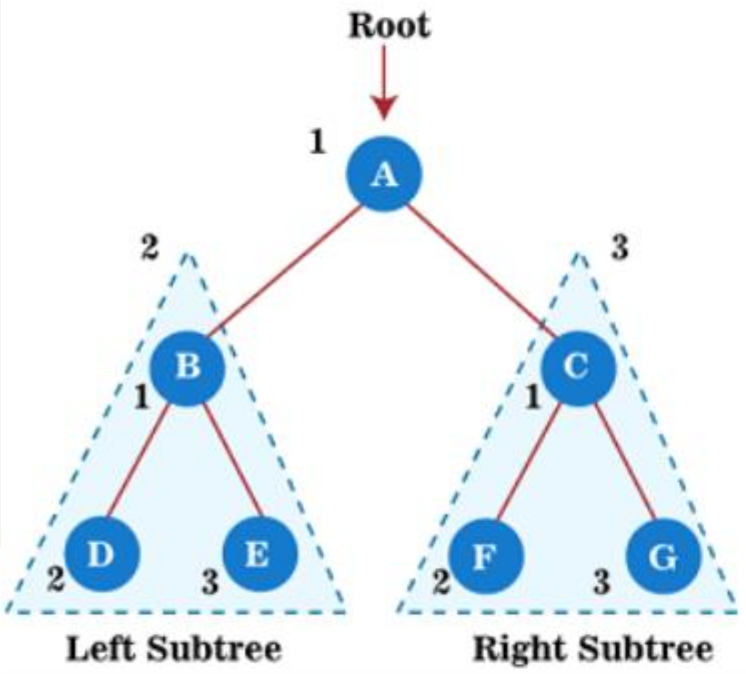
Preorder Traversal: 1 → 2 → 4 → 5 → 3 → 6

Binary Tree Traversal: Pre-order (Root-Left-Right)



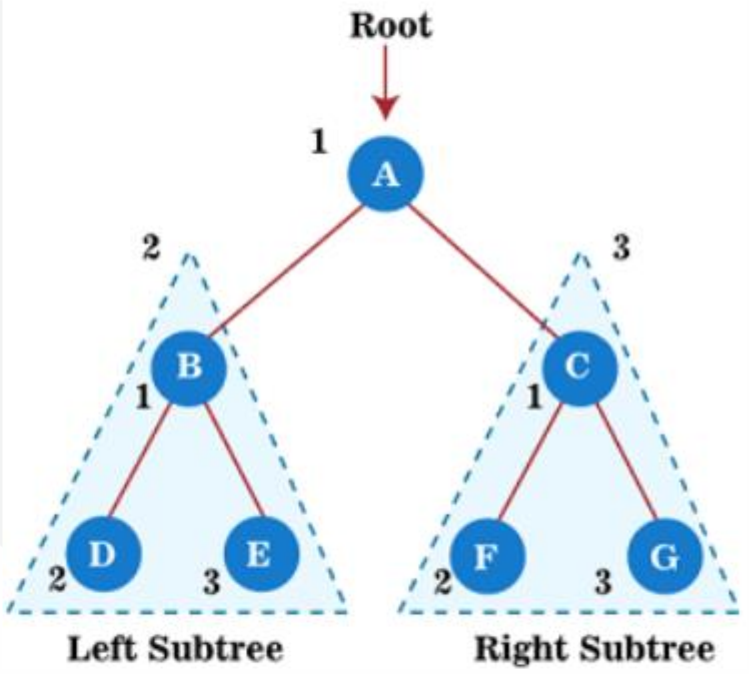
Binary Tree Traversal: Pre-order (Root-Left-Right)

- First, we traverse the root node A; after that, move to its **left subtree B**, which will also be traversed in preorder.
- So, for left subtree B, first, the root node B is traversed itself; after that, its left subtree D is traversed. Since node D does not have any children, move to right subtree E. As node E also does not have any children, the traversal of the left subtree of root node A is completed.



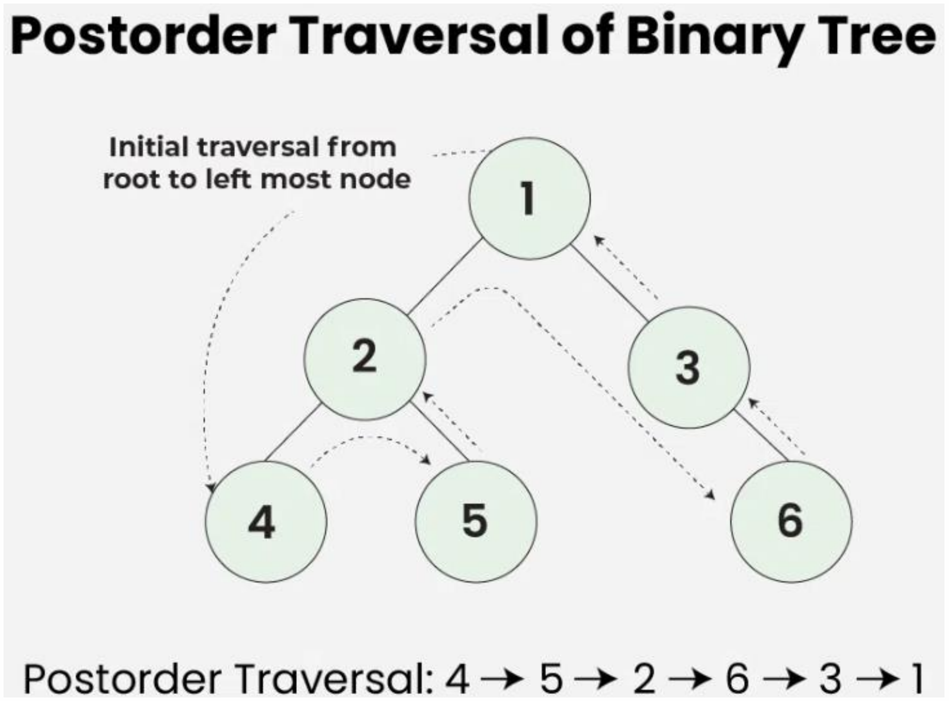
Binary Tree Traversal: Pre-order (Root-Left-Right)

- Now, move towards the right subtree of root node A that is C. So, for right subtree C, first the root node C has traversed itself; after that, its left subtree F is traversed. Since node F does not have any children, move to the right subtree G. As node G also does not have any children, traversal of the right subtree of root node A is completed.
- A → B → D → E → C → F → G

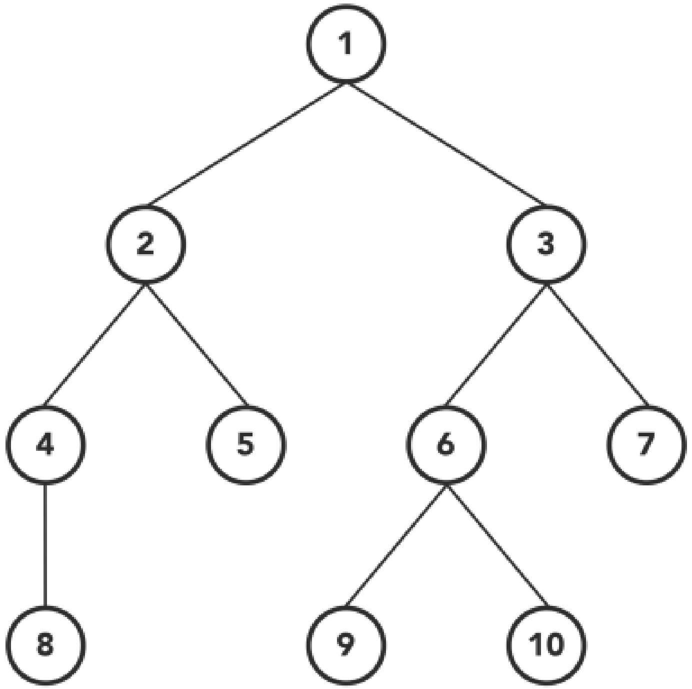


Binary Tree Traversal: Post-order (Left-Right-Root)

- Visits the current node before visiting any nodes inside left or right subtrees..
- Algorithm:
  - Traverse the left subtree, i.e., call Preorder(left->subtree)
  - Traverse the right subtree, i.e., call Preorder(right->subtree)
  - Visit the root.

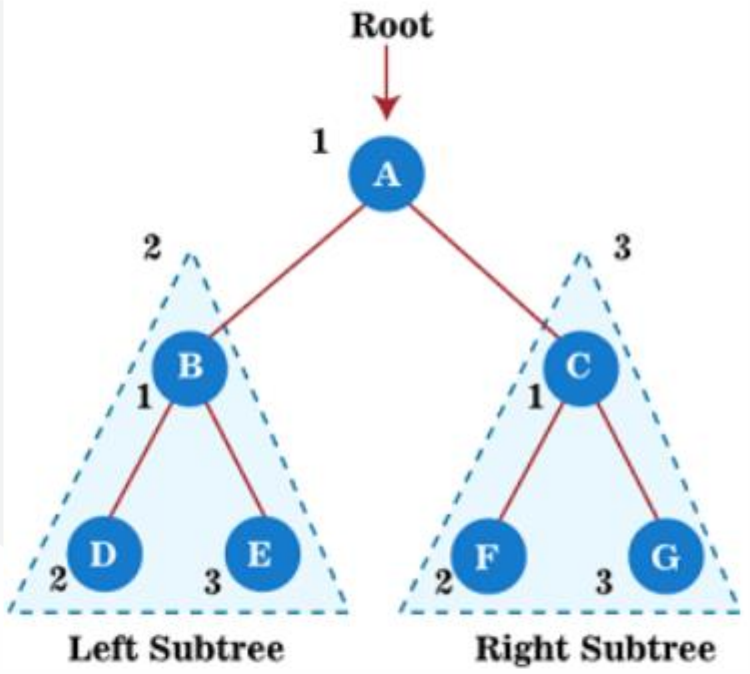

Postorder Traversal of Binary Tree
Initial traversal from root to left most node
1
2
3
4
5
6
Postorder Traversal: 4 → 5 → 2 → 6 → 3 → 1

Binary Tree Traversal: Post-order (Left-Right-Root)



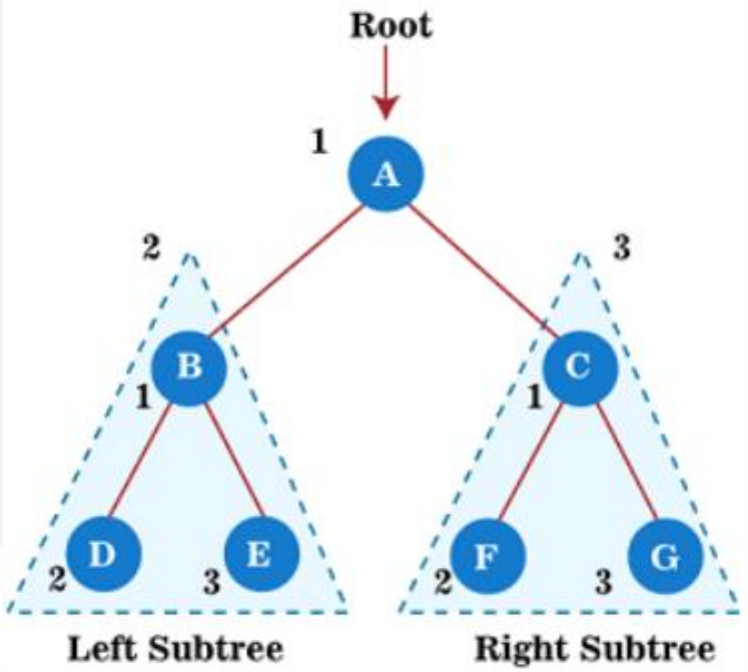
Binary Tree Traversal: Post-order (Left-Right-Root)

- So, for left subtree B, first, its left subtree D is traversed. Since node D does not have any children, traverse the right subtree E. As node E also does not have any children, move to the root node B. After traversing node B, the traversal of the left subtree of root node A is completed.
- Now, move towards the right subtree of root node A that is C. So, for right subtree C, first its left subtree F is traversed.



Binary Tree Traversal: In-order (Left-Right-Root)

- Since node F does not have any children, traverse the right subtree G. As node G also does not have any children, therefore, finally, the root node of the right subtree, i.e., C, is traversed. The traversal of the right subtree of root node A is completed.
- At last, traverse the root node of a given tree, i.e., A.
- D → E → B → F → G → C → A



Threaded Binary Trees

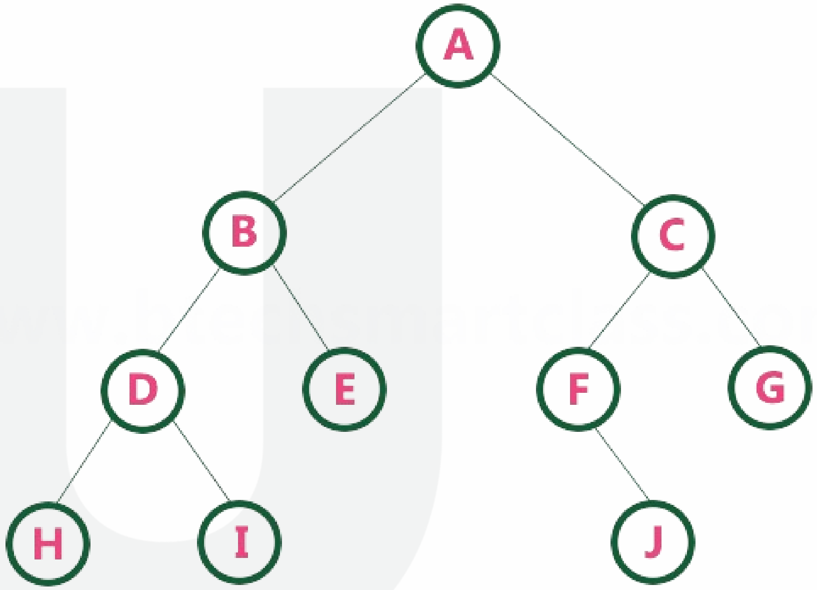
- A threaded binary tree modifies the standard binary tree structure that provides additional information about the tree's traversal order.
- In a threaded binary tree, some null pointers are replaced with references to predecessor or successor nodes.
- This modification enables us to navigate the tree efficiently without needing recursive algorithms or explicitly maintaining a stack.
- The threading of a binary tree simplifies its traversal by creating threads, or links, that point to the in-order predecessor and successor of each node.
- These threads effectively eliminate the need for backtracking during traversal, improving efficiency.

Threaded Binary Trees (Purpose)

- The primary purpose of using threaded binary trees is to optimize traversal operations, such as in-order traversal.
- By threading the tree, we eliminate the overhead of recursive function calls or stack-based iterations, making the traversal process more efficient.
- Moreover, threaded binary trees facilitate quick access to predecessor and successor nodes, which can be beneficial in various scenarios.

Threaded Binary Trees

- To convert this binary tree into a threaded binary tree, first find the in-order traversal of that tree...
- In-order traversal of above binary tree...
- H - D - I - B - E - A - F - J - C – G
- When we represent this binary tree using linked list representation, nodes H, I, E, F, J and G left child pointers are NULL.



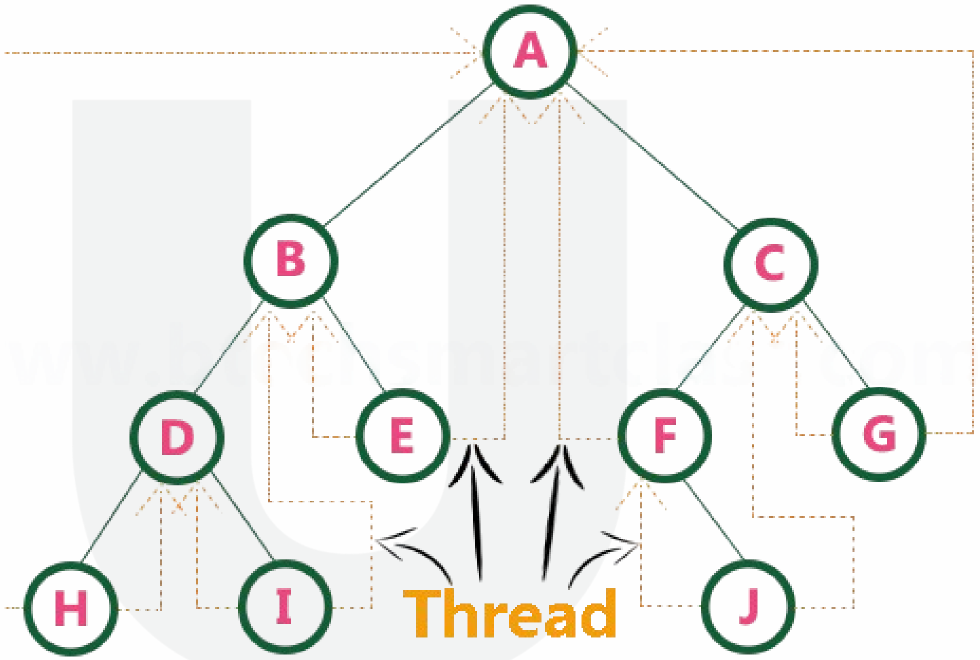
- This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A.

Parul® University

DIGITAL LEARNING CONTENT

Threaded Binary Trees

- And nodes H, I, E, J and G right child pointers are NULL.
- These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.



B – Trees

- The limitations of traditional binary search trees can be frustrating. Meet the B-Tree, the multi-talented data structure that can handle massive amounts of data with ease.
- When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage.
- B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.
- Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as “large key” trees.

B – Trees

- Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height.
- This shallow height leads to less disk I/O, which results in faster search and insertion operations.
- B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.
- B-Trees maintains balance by ensuring that each node has a minimum number of keys, so the tree is always balanced.
- This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Properties of B – Trees

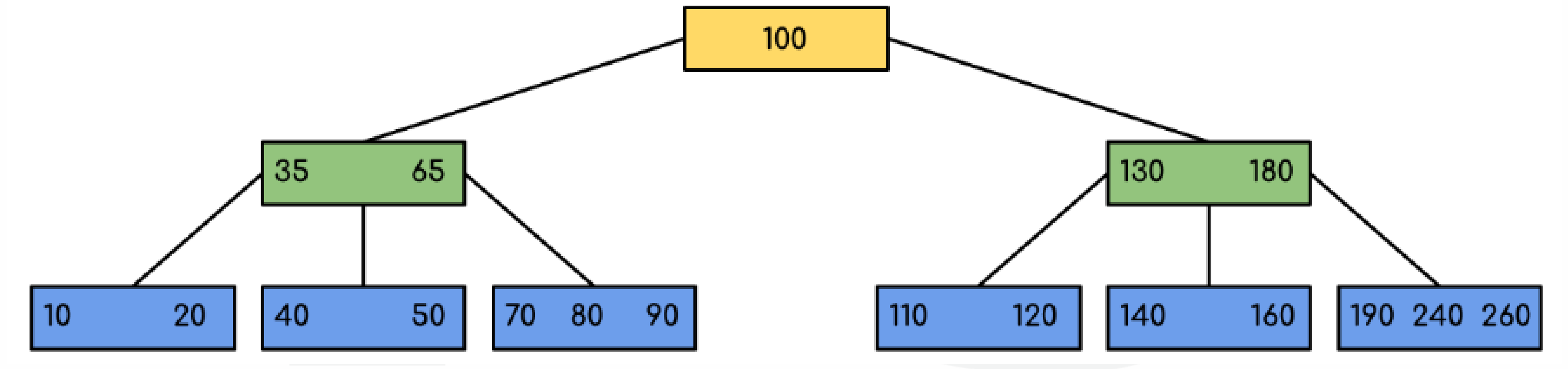
- All leaves are at the same level.
- B-Tree is defined by the term minimum degree ‘t‘. The value of ‘t‘ depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum of 1 key.
- All nodes (including root) may contain at most (2*t – 1) keys.
- Number of children of a node is equal to the number of keys in it plus 1.

Properties of B – Trees

- All keys of a node are sorted in increasing order. The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert, and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.

Example of B – Trees

- Following is an example of a B-Tree of minimum order 5
- Note: that in practical B-Trees, the value of the minimum order is much more than 5.

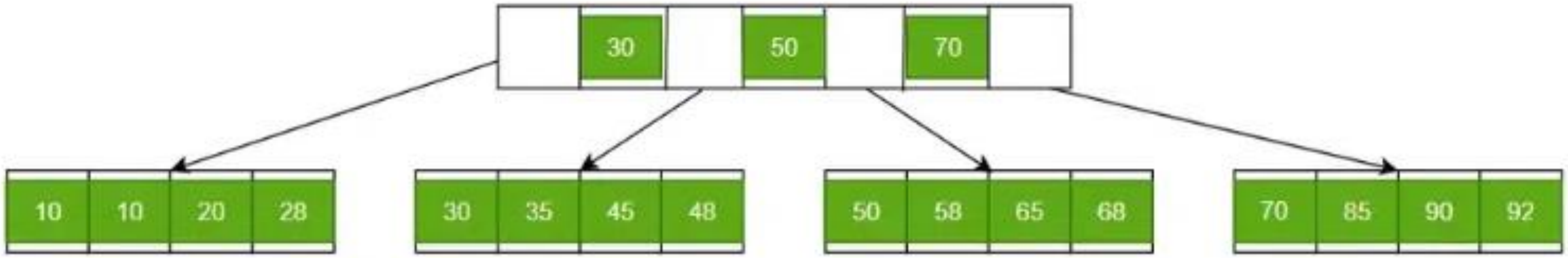


- We can see in the above diagram that all the leaf nodes are at the same level and all non-leafs have no empty sub-tree and have keys one less than the number of their children.

B+ Trees

- B+ Tree is a variation of the B-tree data structure. In a B+ tree, data pointers are stored only at the leaf nodes of the tree.
- In a B+ tree structure of a leaf node differs from the structure of internal nodes. The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
- The leaf nodes of the B+ tree are linked together to provide ordered access to the search field to the records. Internal nodes of a B+ tree are used to guide the search.
- Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree.

B+ Trees



- B+ Trees contain two types of nodes:
- Internal Nodes: Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
- Leaf Nodes: Leaf Nodes are the nodes that have n pointers.

Difference Between B+ Tree and B Tree

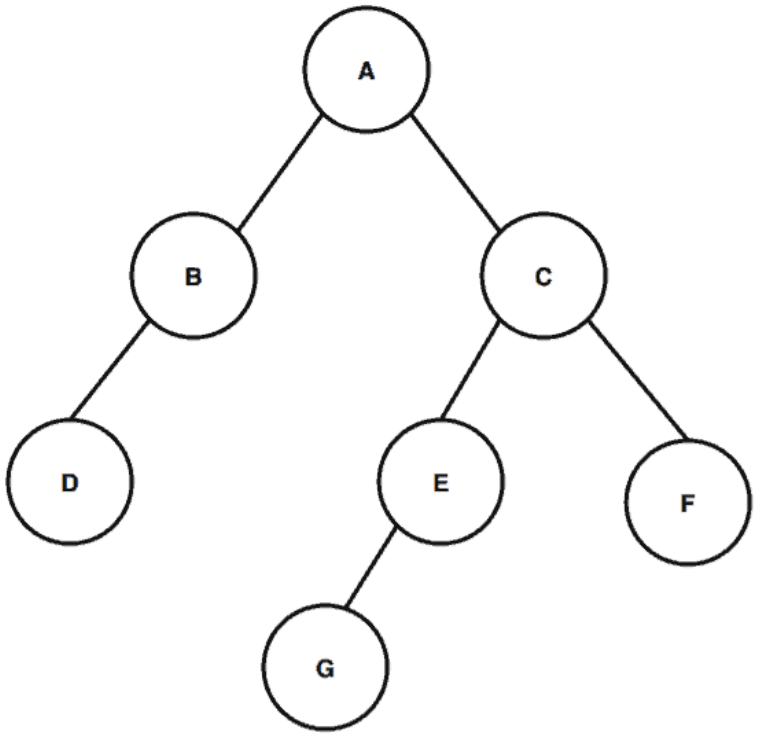
Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	Nodes store both keys and data values
Leaf Nodes	Leaf nodes form a linked list for efficient range-based queries	Leaf nodes do not form a linked list
Order	Higher order (more keys)	Lower order (fewer keys)
Key Duplication	Typically allows key duplication in leaf nodes	Usually does not allow key duplication
Disk Access	Better disk access due to sequential reads in a linked list structure	More disk I/O due to non-sequential reads in internal nodes

Difference Between B+ Tree and B Tree

Parameters	B+ Tree	B Tree
Applications	Database systems, file systems, where range queries are common	In-memory data structures, databases, general-purpose use.
Performance	Better performance for range queries and bulk data retrieval	Balanced performance for search, insert, and delete operations
Memory Usage	Requires more memory for internal nodes	Requires less memory as keys and values are stored in the same node

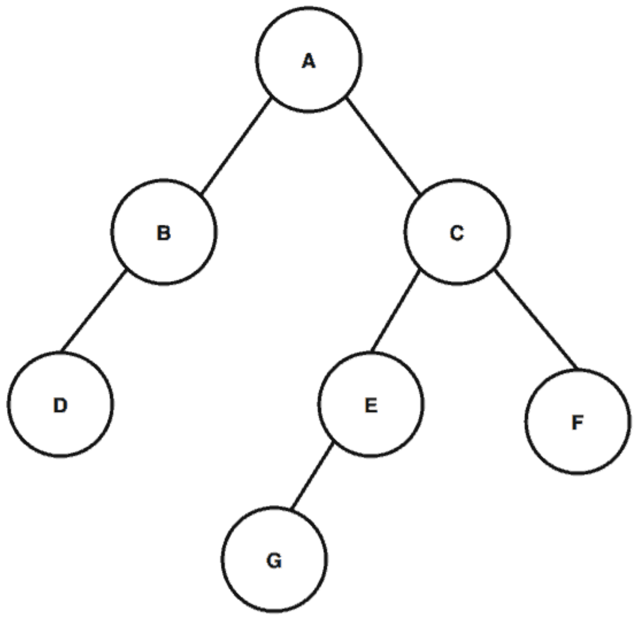
Height of a Binary Tree

- First, we'll calculate the height of node C. So, according to the definition, the height of node C is the largest number of edges in a path from the leaf node to node C.
- We can see that there are two paths for node C: C → E → G, and C → F. The largest number of edges among these two paths would be 2; hence, the height of node C is 2.



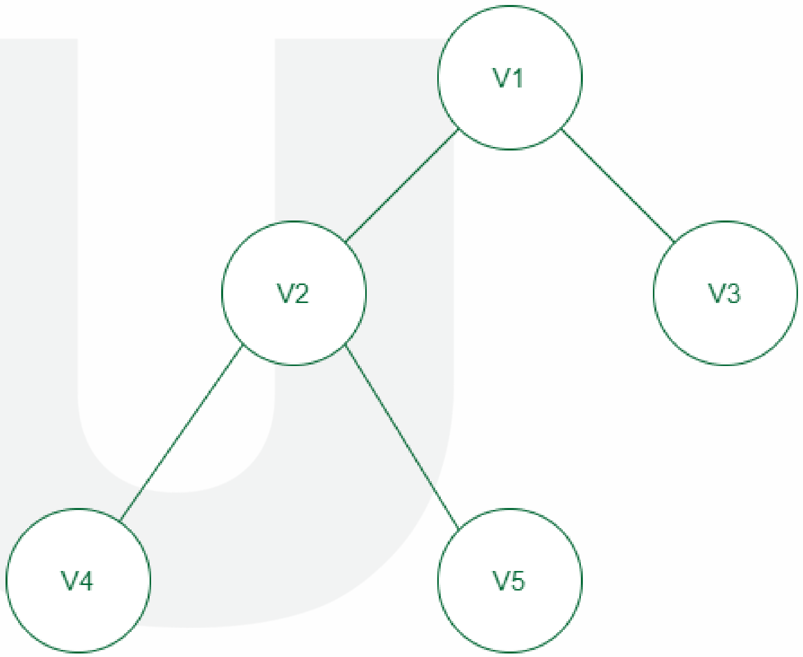
Height of a Binary Tree

- Now we'll calculate the height of the binary tree. From the root, we can have three different paths leading to the leaf nodes: A → C → F, A → B → D, and A → C → E → G. Among these three paths, the path A → C → E → G contains the largest number of edges, which is 3. Therefore, the height of the tree is 3.
- Next, we want to find the depth of node B. We can see that from the root, there's only one path to node B, and it has one edge. Thus, the depth of node B is 1.
- As we previously mentioned, the depth of a binary tree is equal to the height of the tree. Therefore, the depth of the binary tree is 3.



Height – Balanced Tree

- A height-balanced binary tree is defined as a binary tree in which the height of the left and the right subtree of any node differ by not more than 1.
- Conditions for Height-Balanced Binary Tree:
  - The difference between the heights of the left and the right subtree for any node is not more than one.
  - The left subtree is balanced.
  - The right subtree is balanced.

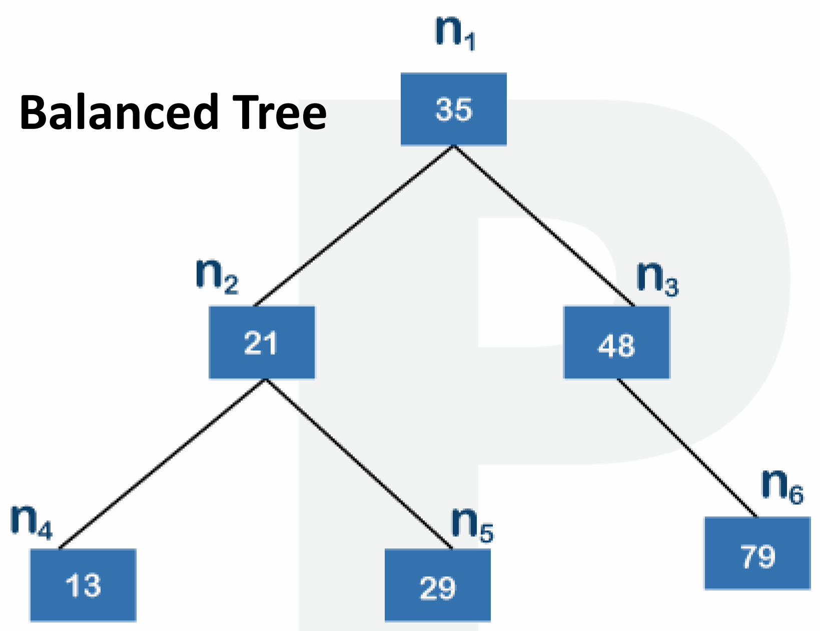


Height – Balanced Tree

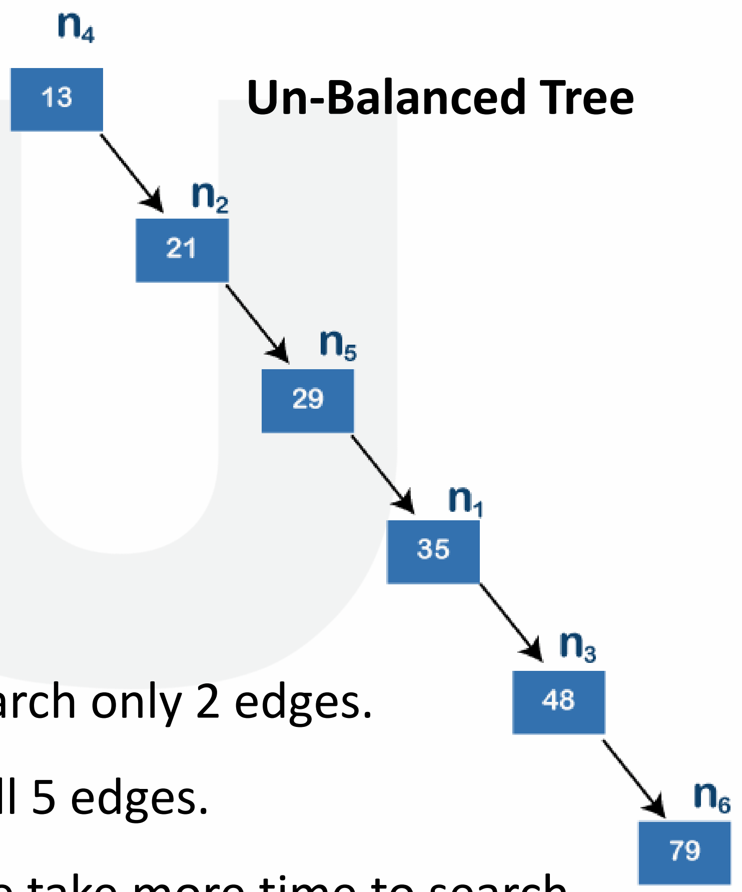
- To check if the binary tree is height-balanced or not, you have to check the height balance of each node.
- For this, you need to calculate the heights of the two subtrees.
- The height balance of a node is calculated as follows:
  - height balance of node = height of right subtree – height of left subtree

Why to use Height – Balanced Tree

Balanced Tree



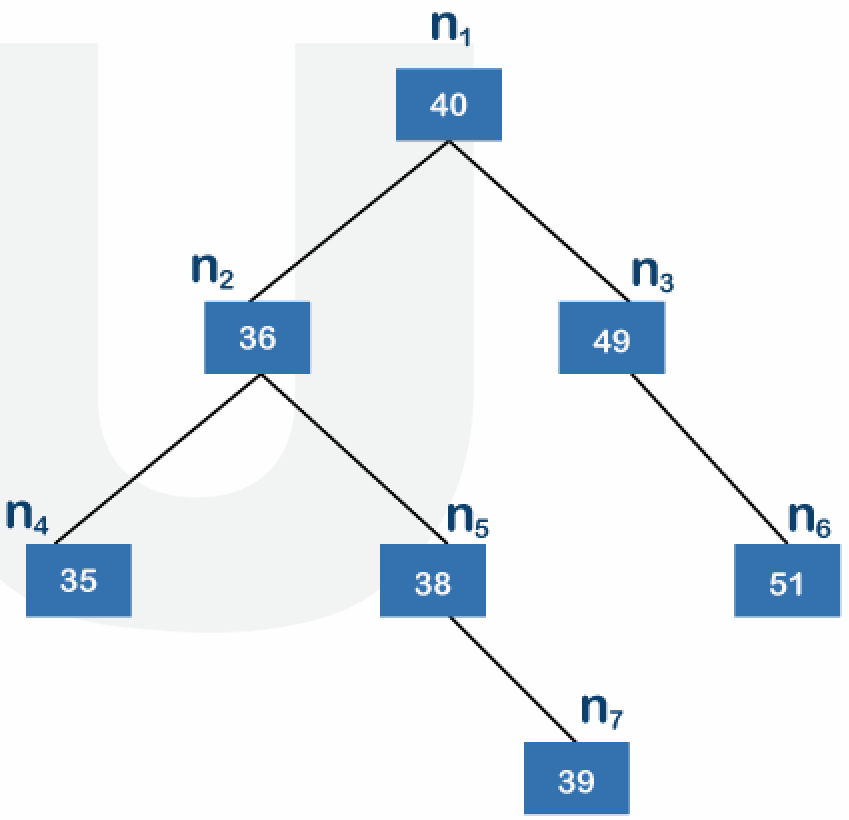
Un-Balanced Tree



- To search 79 in balanced tree, one have to search only 2 edges.
- But in un-balanced tree, one have to search all 5 edges.
- So, it may be possible that in un-balanced tree take more time to search.

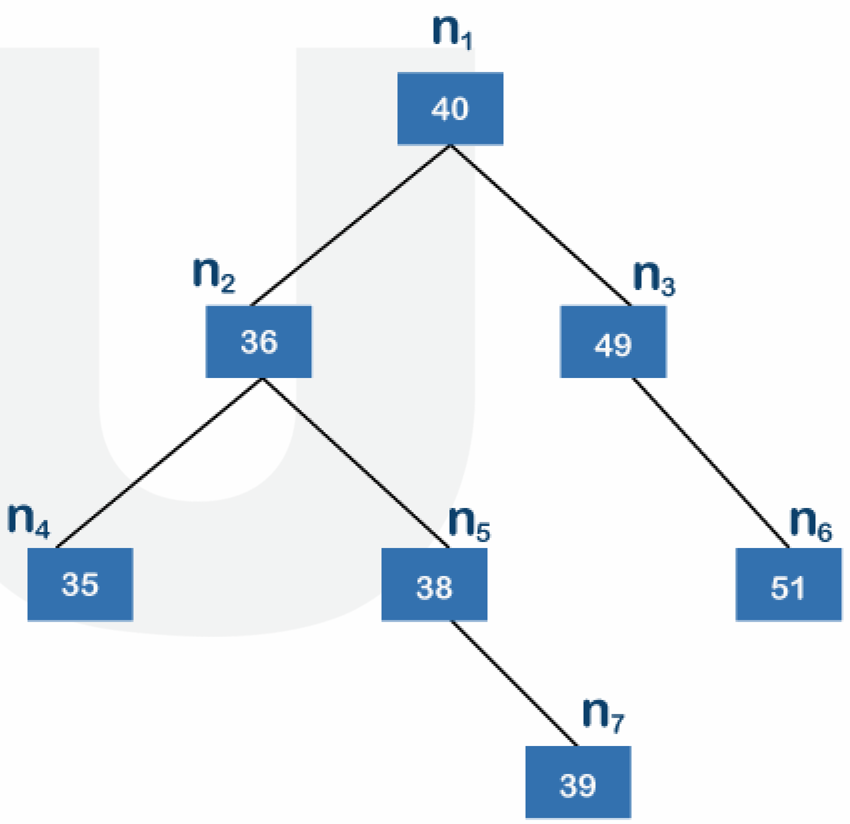
Height – Balanced Tree

- **Example of Balanced Binary Tree:**
- In this tree, n1 is a root node, and n4, n6, n7 are the leaf nodes.
- The n7 node is the farthest node from the root node.
- The n4 and n6 contain 2 edges and there exist three edges between the root node and n7 node.
- Since n7 is the farthest from the root node; therefore, the height of the above tree is 3.



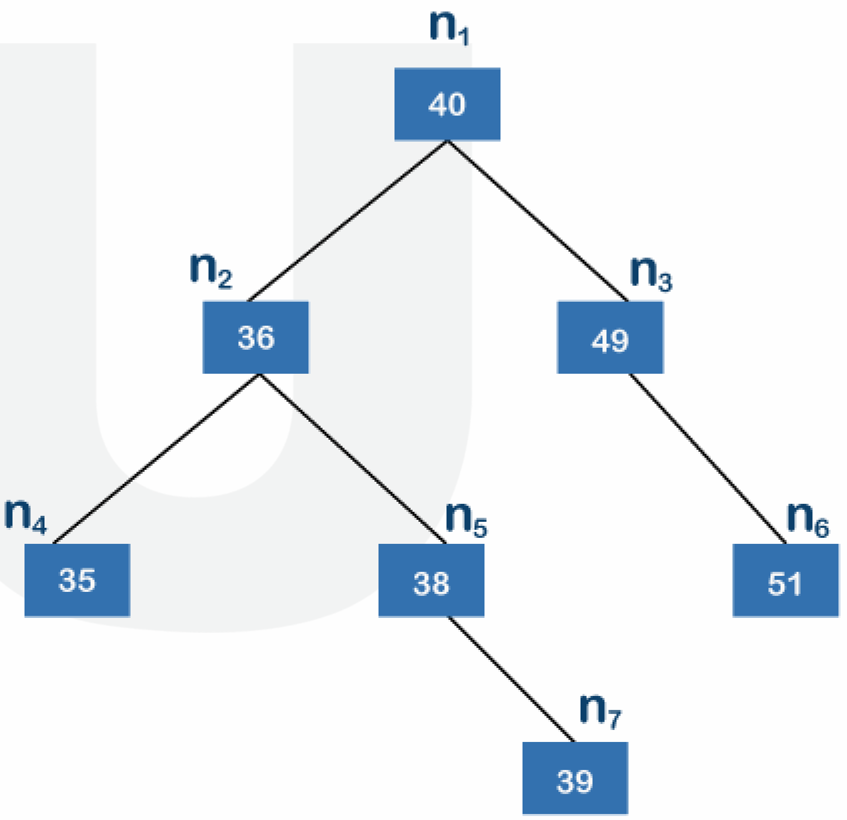
Height – Balanced Tree

- Now we will see whether the above tree is balanced or not. The left subtree contains the nodes n2, n4, n5, and n7, while the right subtree contains the nodes n3 and n6.
- The left subtree has two leaf nodes, i.e., n4 and n7. There is only one edge between the node n2 and n4 and two edges between the nodes n7 and n2; therefore, node n7 is the farthest from the root node. The height of the left subtree is 2.
- The right subtree contains only one leaf node, i.e., n6, and has only one edge; therefore, the height of the right subtree is 1.



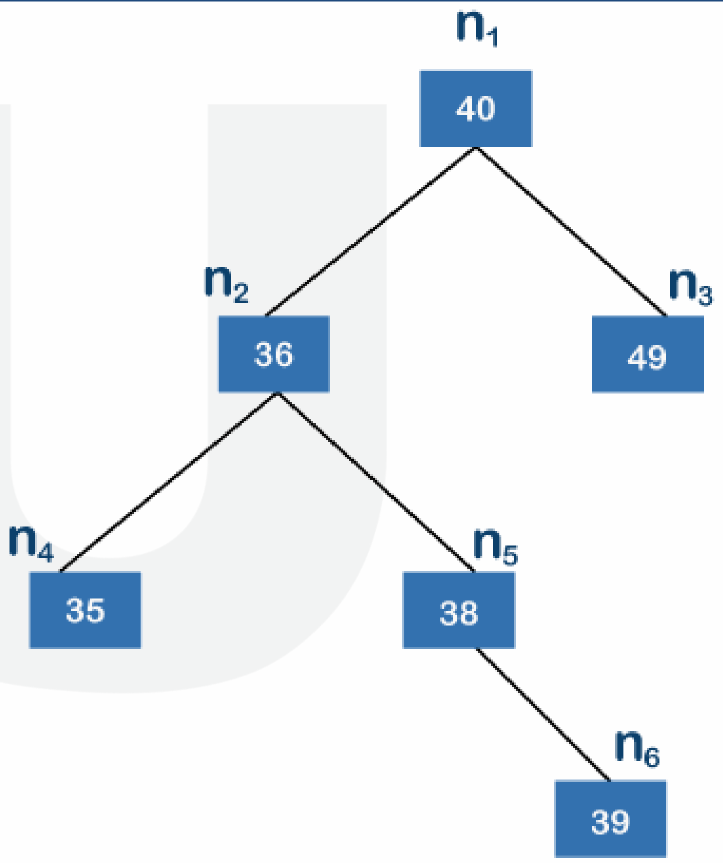
Height – Balanced Tree

- The difference between the heights of the left subtree and right subtree is 1. Since we got the value 1 so we can say that the above tree is a height-balanced tree.
- This process of calculating the difference between the heights should be performed for each node like n2, n3, n4, n5, n6 and n7.
- When we process each node, then we will find that the value of k is not more than 1, so we can say that the above tree is a ***balanced binary tree***.



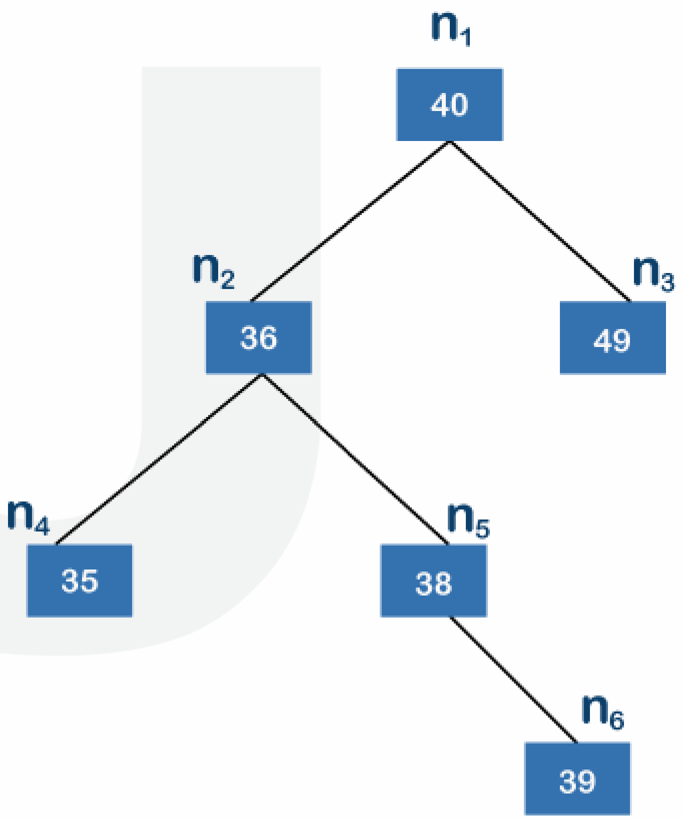
Height – Balanced Tree

- **Example of Un-Balanced Binary Tree:**
- In this tree, n6, n4, and n3 are the leaf nodes, where n6 is the farthest node from the root node.
- Three edges exist between the root node and the leaf node; therefore, the height of the above tree is 3.
- When we consider n1 as the root node, then the left subtree contains the nodes n2, n4, n5, and n6, while subtree contains the node n3.



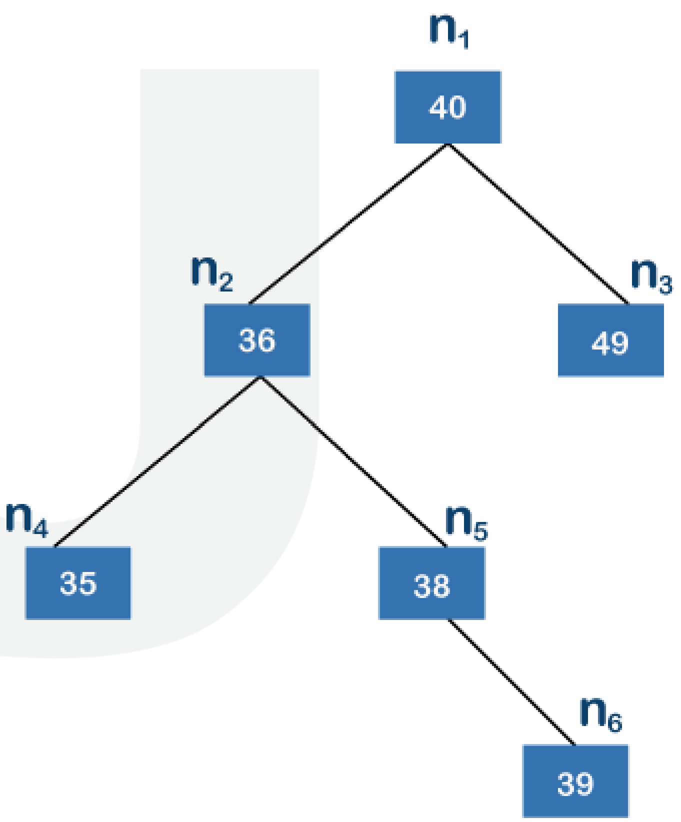
Height – Balanced Tree

- In the left subtree, n2 is a root node, and n4 and n6 are leaf nodes.
- Among n4 and n6 nodes, n6 is the farthest node from its root node, and n6 has two edges; therefore, the height of the left subtree is 2.
- The right subtree does have any child on its left and right; therefore, the height of the right subtree is 0.
- Since the height of the left subtree is 2 and the right subtree is 0, so the difference between the height of the left subtree and right subtree is 2.



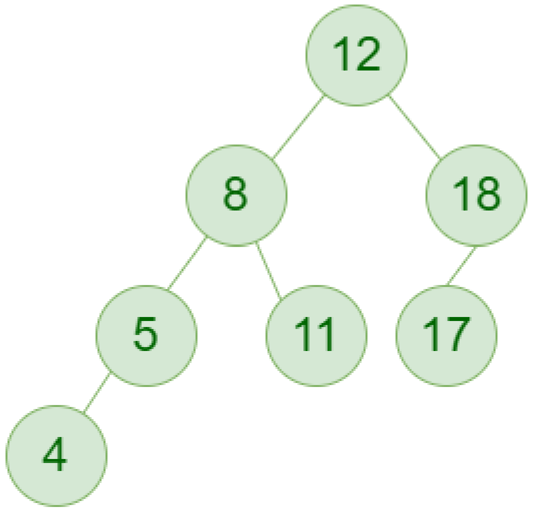
Height – Balanced Tree

- According to the definition, the difference between the height of left sub tree and the right subtree must not be greater than 1.
- In this case, the difference comes to be 2, which is greater than 1; therefore, this binary tree is an unbalanced binary search tree.



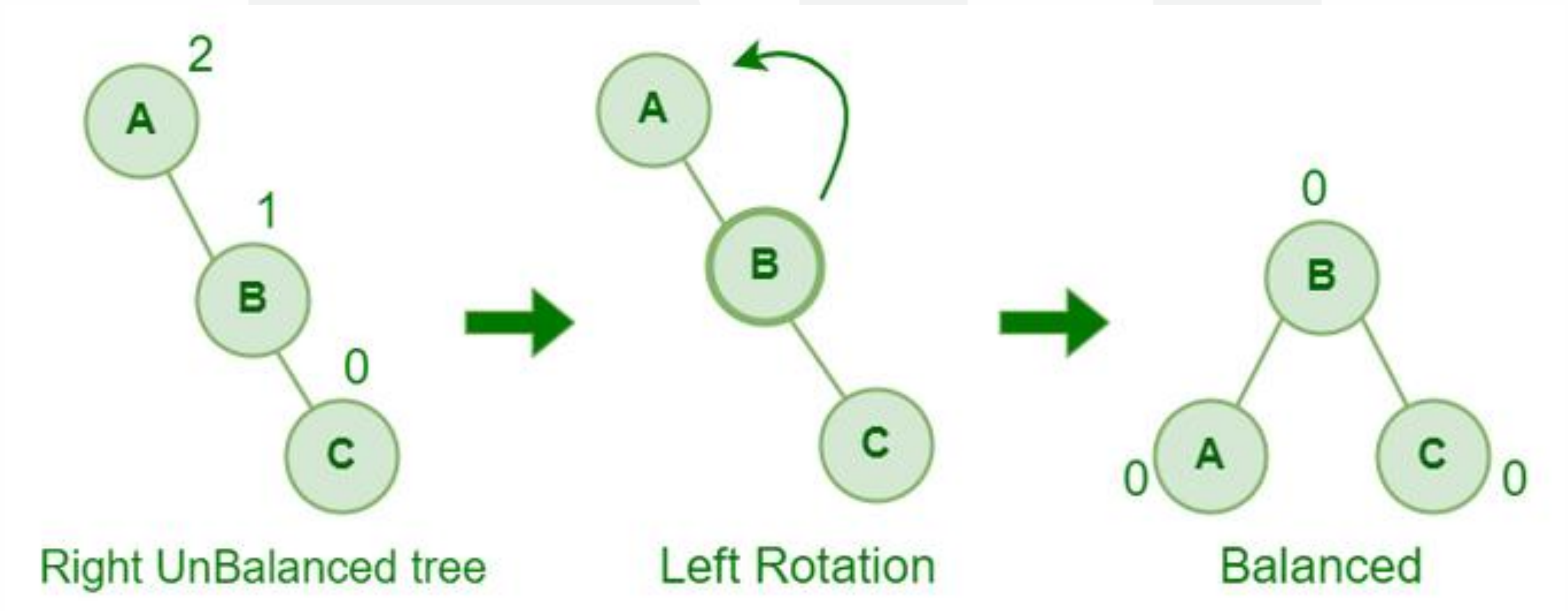
AVL Tree

- The name of the tree given as AVL as of the inventor's name (Georgy **A**delson-**V**elsky and Evgenii **L**andis)
- An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.
- This tree is AVL because the differences between the heights of left and right subtrees for every node are less than or equal to 1.
- **Operations on AVL Tree:** Insertion, Deletion, Searching



Rotating the subtrees in an AVL Tree

- **Left Rotation:** When a node is added into the right subtree of the right subtree, if the tree gets out of balance, we do a single left rotation.

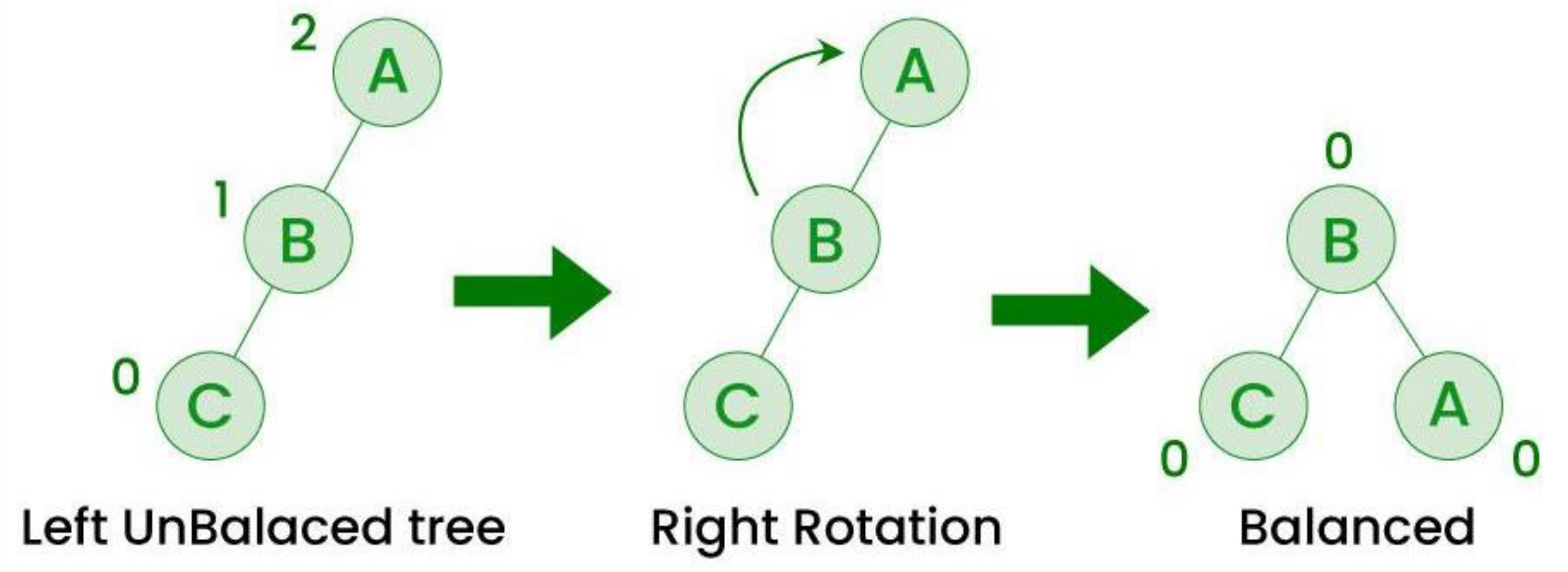


Parul® University

DIGITAL LEARNING CONTENT

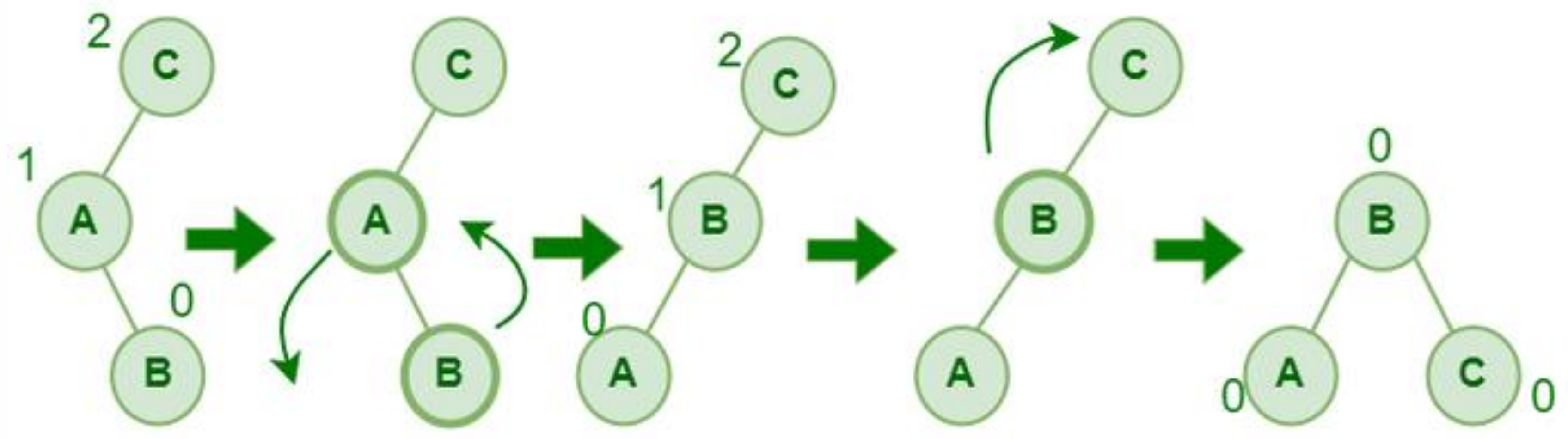
Rotating the subtrees in an AVL Tree

- **Right Rotation:** If a node is added to the left subtree of the left subtree, the AVL tree may get out of balance, we do a single right rotation.



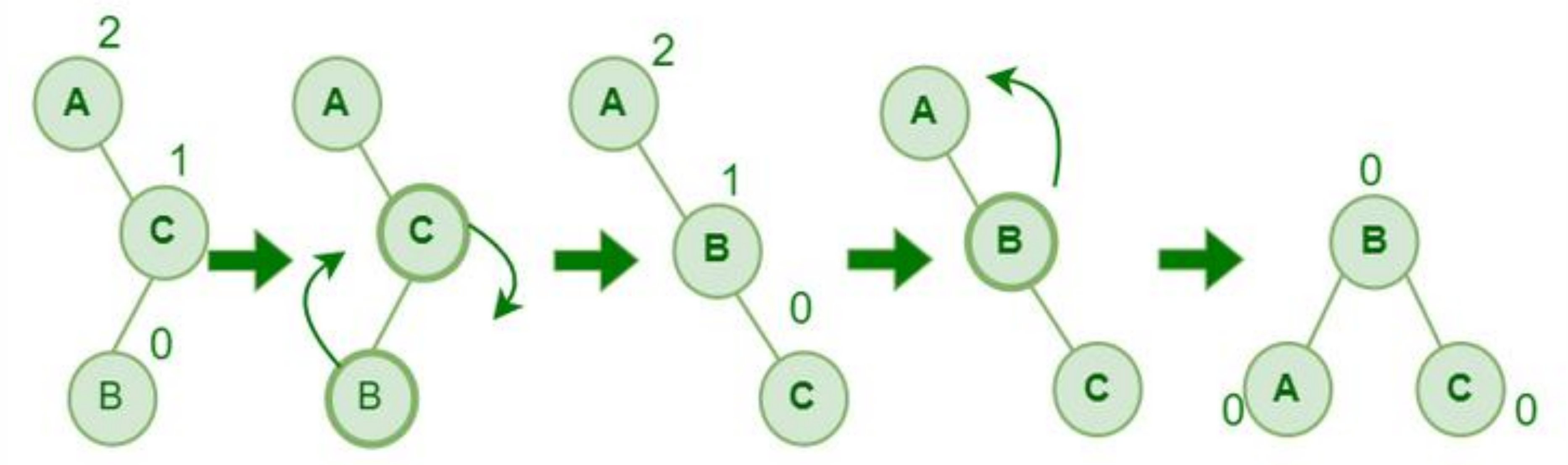
Rotating the subtrees in an AVL Tree

- **Left – Right Rotation:** A left-right rotation is a combination in which first left rotation takes place after that right rotation executes.



Rotating the subtrees in an AVL Tree

- **Right – Left Rotation:** A right-left rotation is a combination in which first right rotation takes place after that left rotation executes.



Applications of Heap Tree

1. **Priority Queues:** Heaps are commonly used to implement priority queues, where elements with higher priority are extracted first. This is useful in many applications such as scheduling tasks, handling interruptions, and processing events.

2. **Sorting Algorithms:** Heapsort, a comparison-based sorting algorithm, is implemented using the Heap data structure. It has a time complexity of O(n log n), making it ***efficient for large datasets***.

3. **Graph algorithms:** Heaps are used in graph algorithms such as Prim's Algorithm, Dijkstra's algorithm., and the A* search algorithm.

4. **Lossless Compression:** Heaps are used in data compression algorithms such as Huffman coding, which uses a priority queue implemented as a min-heap to build a Huffman tree.

5. **Medical Applications:** In medical applications, heaps are used to store and manage patient information based on priority, such as vital signs, treatments, and test results.

Applications of Heap Tree

6. **Load balancing:** Heaps are used in load balancing algorithms to distribute tasks or requests to servers, by processing elements with the lowest load first.

7. **Order statistics:** The Heap data structure can be used to efficiently find the kth smallest (or largest) element in an array. See method 4 and 6 of this post for details.

8. **Resource allocation:** Heaps can be used to efficiently allocate resources in a system, such as memory blocks or CPU time, by assigning a priority to each resource and processing requests in order of priority.

9. **Job scheduling:** The heap data structure is used in job scheduling algorithms, where tasks are scheduled based on their priority or deadline. The heap data structure allows efficient access to the highest-priority task, making it a useful data structure for job scheduling applications.

Applications of expression Tree

1. **Evaluation of Expression:** Calculators, spreadsheet applications, programming language interpreters, and compilers use this extensively.

2. **Compiler Design:** Design of compilers used in code creation and semantic analysis. The structure of expressions in source code, enabling analysis of operator precedence, associativity, and type checking. They also make it easier to generate intermediate code and optimize code.

3. **Handling of Mathematical Expressions:** Symbolic mathematical software and computer algebra systems (CAS) employ them.

4. **Interpreting Syntax in Programming Languages:** Programming language parsing and syntax analysis. Tools for code analysis, interpreters, and compilers need this.

5. **Processing Regular Expressions:** Regular expressions can be represented using expression trees, which makes text processing and pattern matching easier.

DIGITAL LEARNING CONTENT