

# Tree

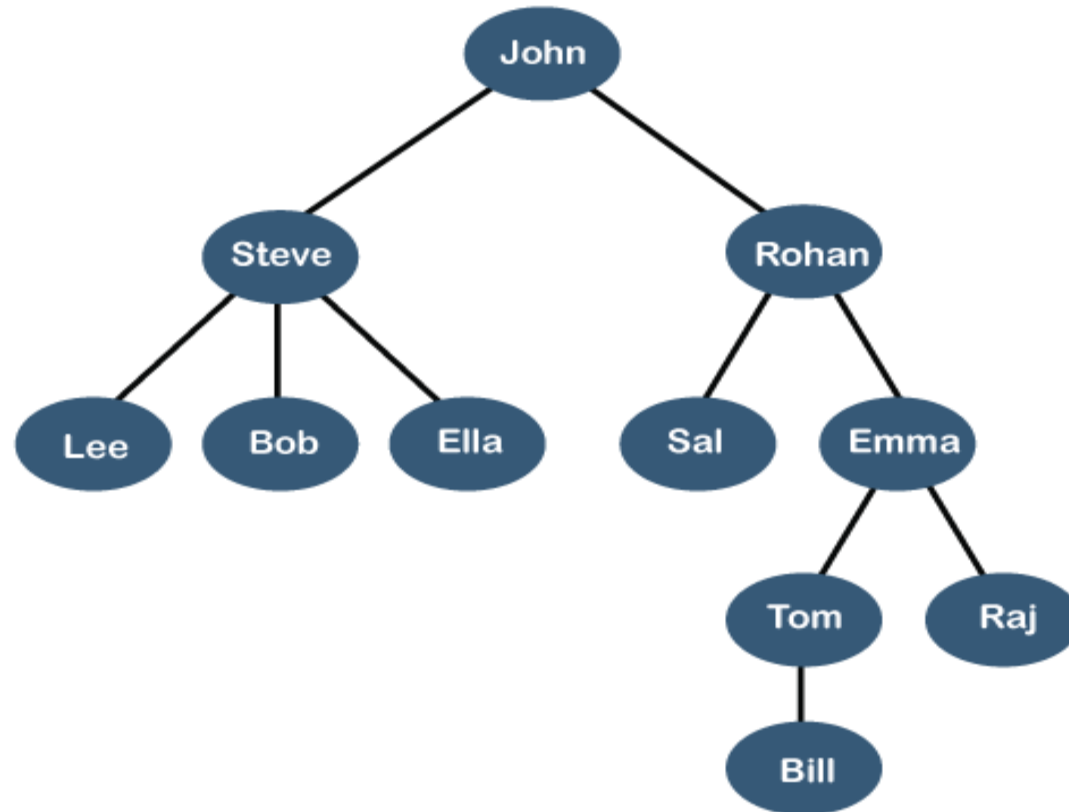
## Chapter 4

# Some factors are considered for choosing the data structure:

- **What type of data needs to be stored?:**
  - It might be a possibility that a certain data structure can be the best fit for some kind of data.
- **Cost of operations:**
  - If we want to minimize the cost for the operations for the most frequently performed operations.
  - For example, we have a simple list on which we have to perform the search operation; then, we can create an array in which elements are stored in sorted order to perform the *binary search*.
  - The binary search works very fast for the simple list as it divides the search space into half.
- **Memory usage:**
  - Sometimes, we want a data structure that utilizes less memory.

# Tree Data Structure

- A *tree* is also one of the data structures that represent hierarchical data.
- Suppose we want to show the employees and their positions in the hierarchical form then it can be represented as shown below:

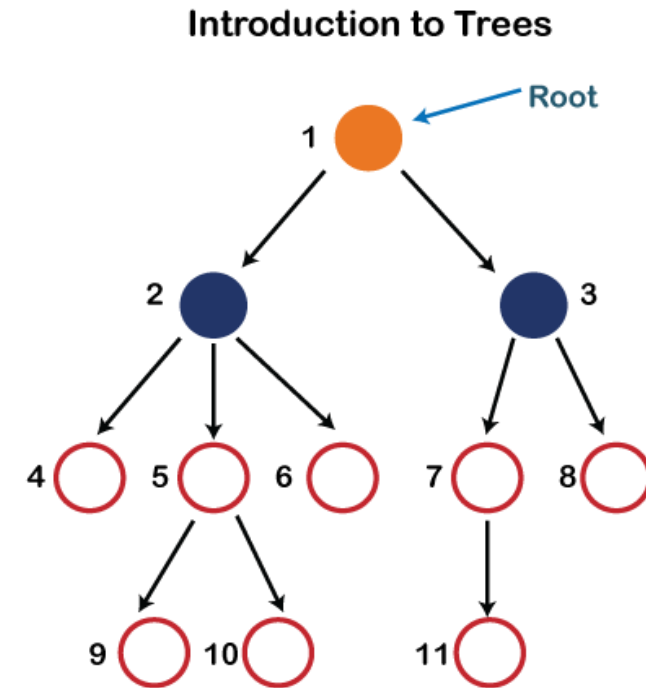


# Some key points of the Tree data structure.

- A tree data structure is defined as a **collection of objects** or **entities** known as **nodes** that **are linked together** 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**.
- In the Tree data structure, the **topmost node is known as a root node**. Each node contains some data, and data can be of any type.
- **Each node contains some data** and the link or reference of other nodes that can be called children.

# Some basic terms used in Tree data structure.

- **Root:**
  - The root node is the topmost node in the tree hierarchy.
  - In other words, the **root node is the one that doesn't have any parent**. In the above structure, node numbered 1 is **the root node of the tree**.
- **Child node:** If the node is a **descendant of any node**, then the node is known as a **child node**.
- **Parent:** If the node contains any **sub-node**, then that node is said to be the parent of that sub-node.
- **Sibling:** The nodes that have the **same parent** are known as **siblings**.
- **Leaf Node:-**
  - The node of the tree, which **doesn't have any child node**, is called a leaf node. A leaf node is the bottom-most node of the tree.
- **Internal nodes:** A node has **atleast one child node** known as an *internal node*.
- **Ancestor node:-**
  - An ancestor of a node is any predecessor node on a path from the root to that node.
  - The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.
- **Descendant:**
  - The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.



# Properties of Tree data structure

- **Recursive data structure:**

- The tree is also known as a *recursive data structure*.
- A tree can be defined as recursively because the distinguished node in a tree data structure is known as a *root node*.
- The root node of the tree contains a link to all the roots of its subtrees. **Recursion means reducing something in a self-similar manner**. So, this recursive property of the tree data structure is implemented in various applications.

- **Number of edges:**

- If there are  $n$  nodes, then there would  $n-1$  edges.
- Each arrow in the structure represents the link or path.
- **Each node, except the root node, will have atleast one incoming link known as an edge.**

- **Depth of node x:**

- The depth of node  $x$  can be defined as the **length of the path from the root to the node  $x$** . One edge contributes one-unit length in the path.
- So, the depth of node  $x$  can also be defined as **the number of edges between the root node and the node  $x$** . The root node has 0 depth.

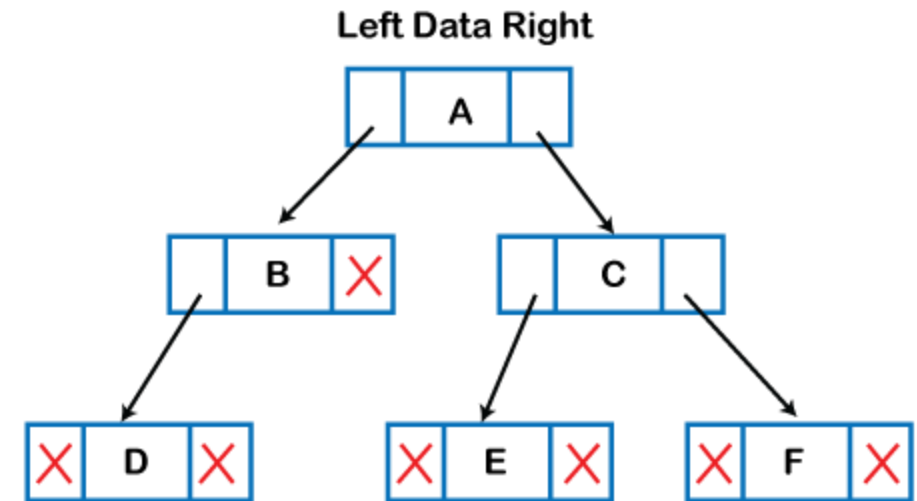
- **Height of node x:**

- The height of node  $x$  can be defined as the **longest path from the node  $x$  to the leaf node**.

# Implementation of Tree

- The tree data structure can be created by **creating the nodes dynamically with the help of the pointers.**
- The figure shows the representation of the tree data structure in the memory.
- In the given structure, the node contains three fields.
- The second field stores the data; the first field stores the address of the left child, and the third field stores the address of the right child.
- In programming, the structure of a node can be defined as:

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



# Applications of trees

- **Storing naturally hierarchical data:**
  - Trees are used to store the data in the **hierarchical structure**.
  - For example, the **file system**.
  - The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:**
  - It is used to organize data for **efficient insertion, deletion and searching**.
  - For example, a binary tree has a  $\log N$  time for searching an element.
- **Trie:**
  - It is a special kind of tree that is **used to store the dictionary**.
  - It is a fast and **efficient way for dynamic spell checking**.
- **Heap:**
  - It is also a **tree data structure implemented using arrays**.
  - It is used to implement priority queues.
- **B-Tree and B+Tree:**
  - B-Tree and B+Tree are the tree data structures **used to implement indexing in databases**.
- **Routing table:**
  - The tree data structure is also used to **store the data in routing tables in the routers**.

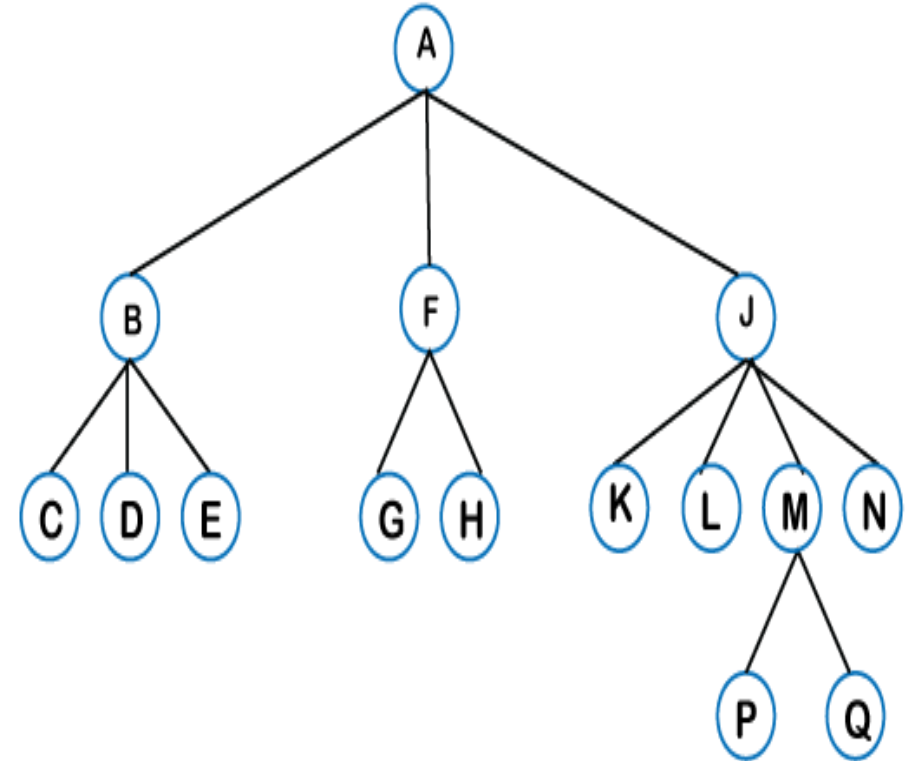


# Types of Tree data structure

- **General tree**
- **Binary tree**
- **Binary Search tree**
- **AVL tree**
- **Red-Black tree**
- **Splay tree**
- **Treap**
- **B-tree**

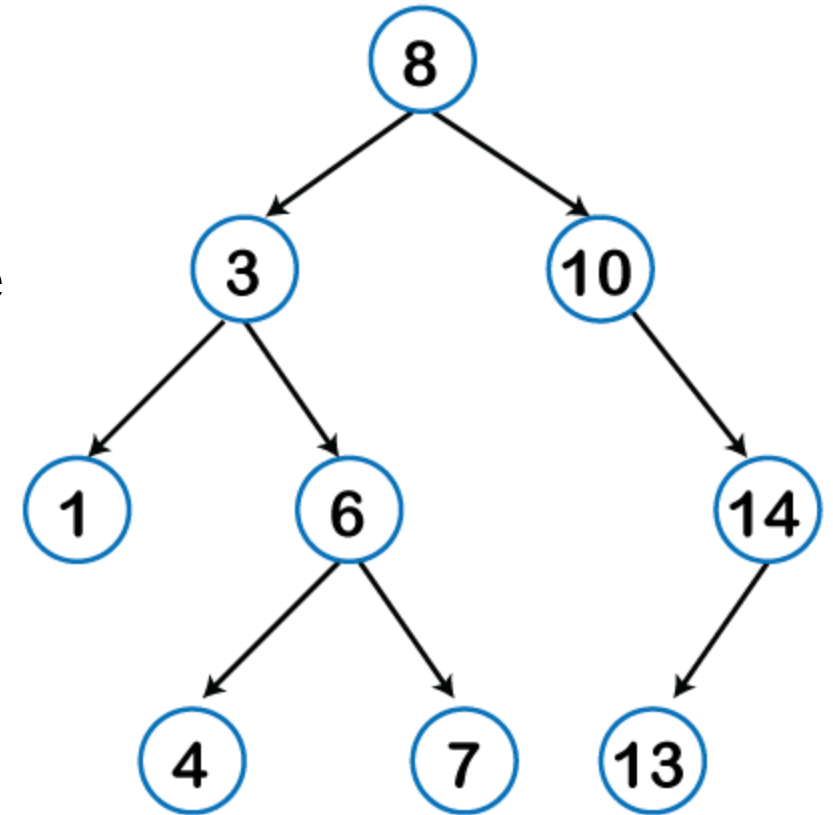
# General tree:

- The **general tree** is one of the types of tree data structure.
- In the general tree, a node can have **either 0 or maximum n number of nodes**.
- There is **no restriction imposed on the degree of the node** (the number of nodes that a node can contain).
- The topmost node in a general tree is known as a root node. The children of the parent node are known as subtrees.
- There can be **n number of subtrees in a general tree**.
- In the general tree, the **subtrees are unordered** as the nodes in the subtree cannot be ordered.
- Every non-empty tree has a downward edge, and these edges are connected to the nodes known as child nodes.
- The root node is labeled with level 0. The nodes that have the same parent are known as siblings.



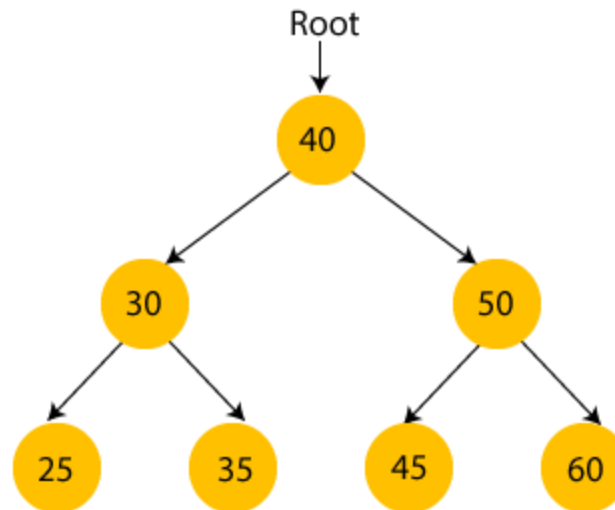
# Binary tree

- Here, binary name itself suggests two numbers, i.e., 0 and 1.
- In a binary tree, **each node in a tree can have utmost two child nodes.**
- Here, utmost means whether the node has 0 nodes, 1 node or 2 nodes.



# 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.

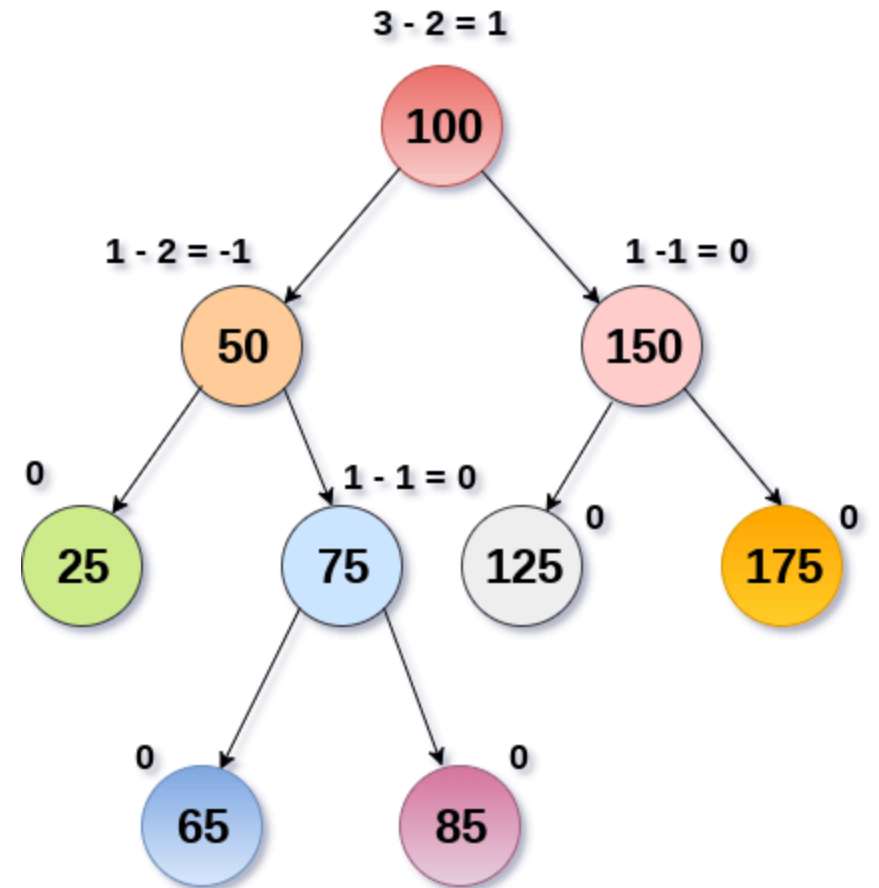


# AVL tree

- It is one of the types of the binary tree, or we can say that it is a variant of the binary search tree.
- AVL tree satisfies the property of the *binary tree* as well as of the *binary search tree*.
- It is a self-balancing binary search Here, self-balancing means that **balancing the heights of left subtree and right subtree**. This balancing is measured in terms of the *balancing factor*.
- We can consider a tree as an **AVL tree** if the tree obeys the binary search tree as well as a balancing factor.
- The balancing factor can be defined as the *difference between the height of the left subtree and the height of the right subtree*.
- The balancing factor's value must be either 0, -1, or 1; therefore, each node in the AVL tree should **have the value of the balancing factor either as 0, -1, or 1**.

# Balance Factor (k) = height (left(k)) - height (right(k))

- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

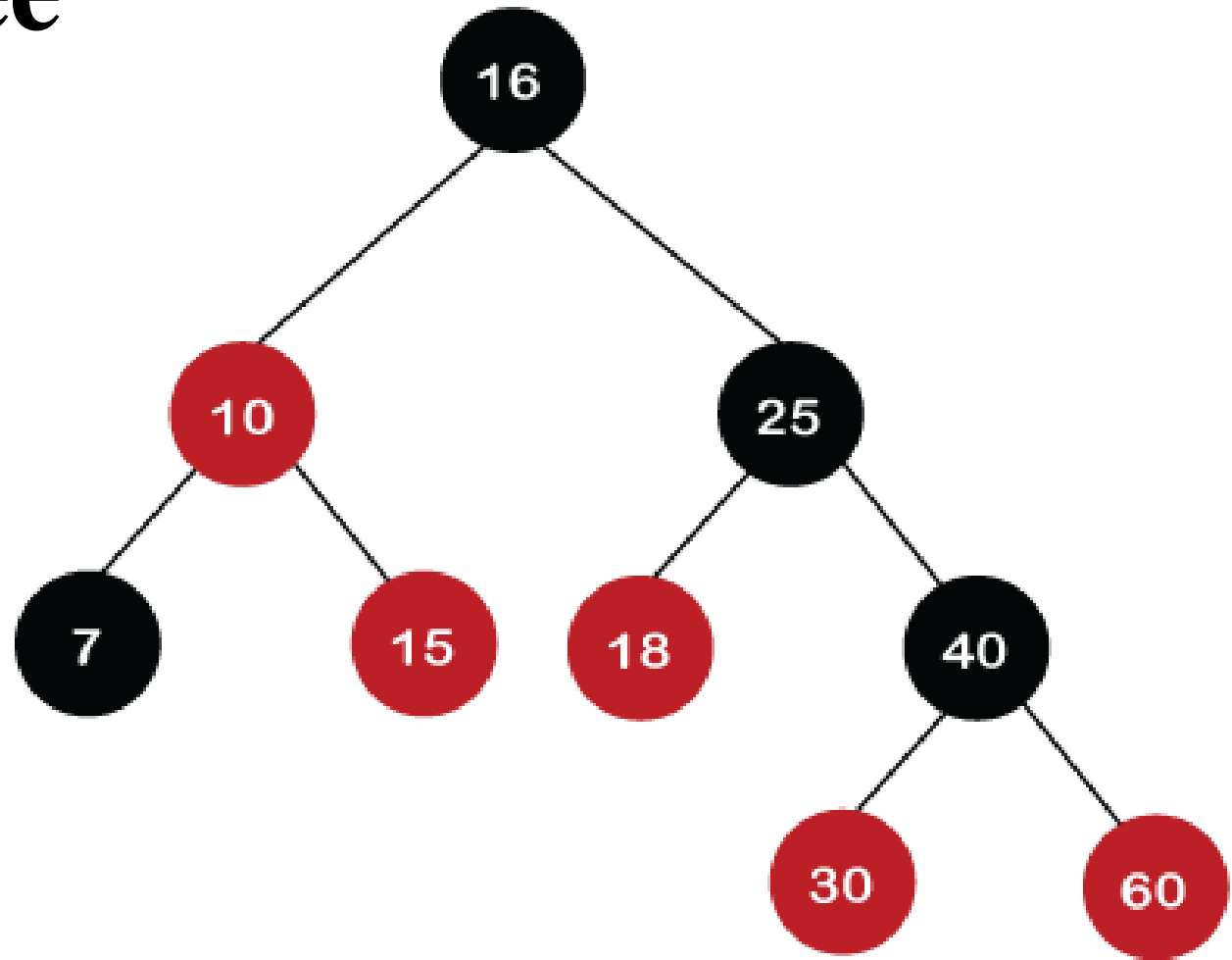


AVL Tree

# The red-Black tree

- It is the **binary search tree**.
- The prerequisite of the Red-Black tree is that we **should know about the binary search tree**.
- In a binary search tree, the value of the left-subtree should be less than the value of that node, and the value of the right-subtree should be greater than the value of that node.
- When any operation is performed on the tree, we want our tree to be balanced so that all the operations like searching, insertion, deletion, etc., take less time, and all these operations will have the time complexity of  $\log_2 n$ .

# The red-Black tree

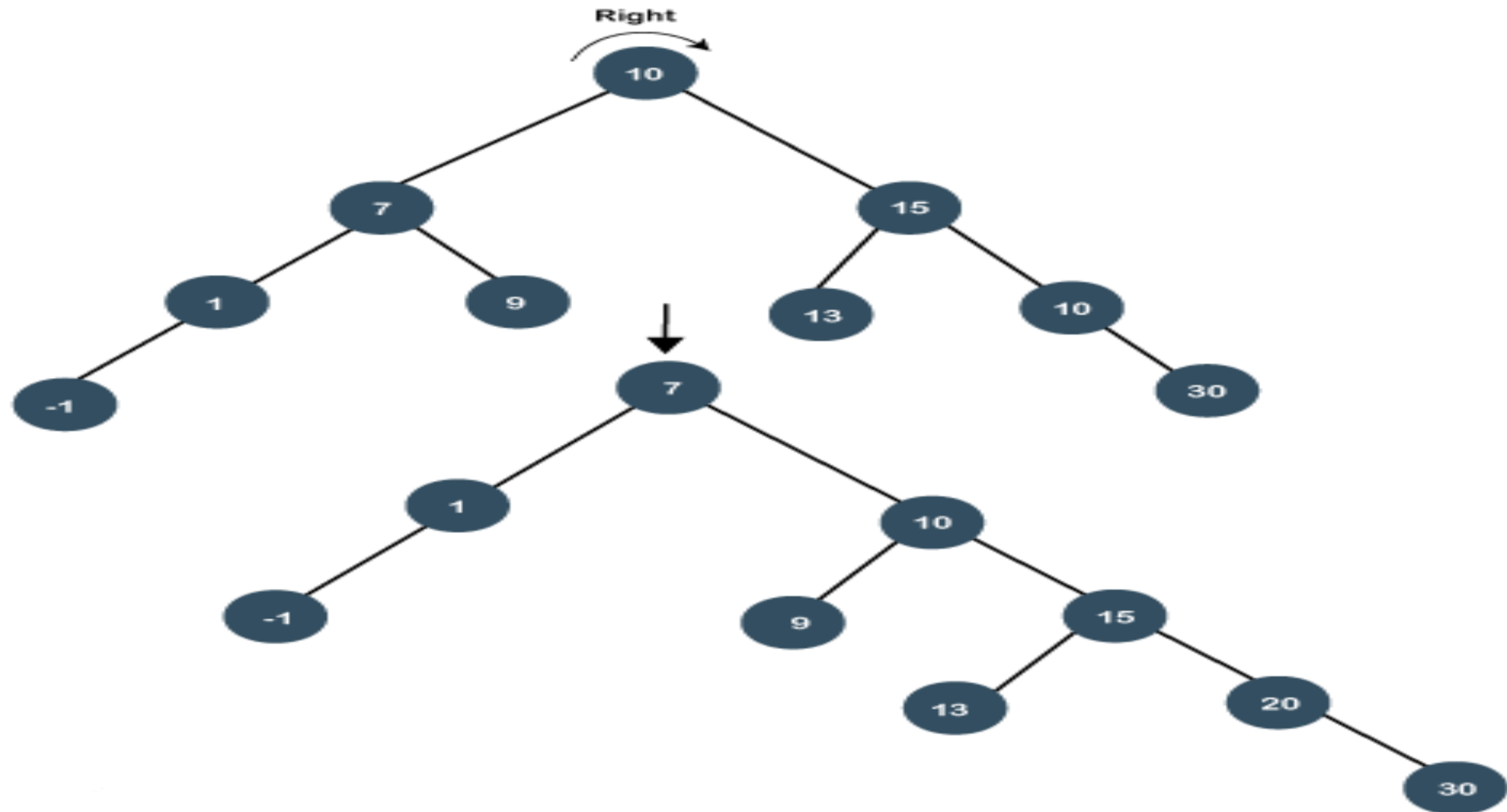




# Splay tree

- The splay tree data structure is also binary search tree in which **recently accessed element is placed at the root position of tree** by performing some rotation operations.
- Here, *splaying* means the **recently accessed node**.
- It is a *self-balancing* binary search tree having **no explicit balance condition** like AVL tree.
- It might be a possibility that height of the splay tree is not balanced, i.e., height of both left and right subtrees may differ, but the operations in splay tree takes order of  **$\log N$**  time where **n** is the number of nodes.

# Splay tree



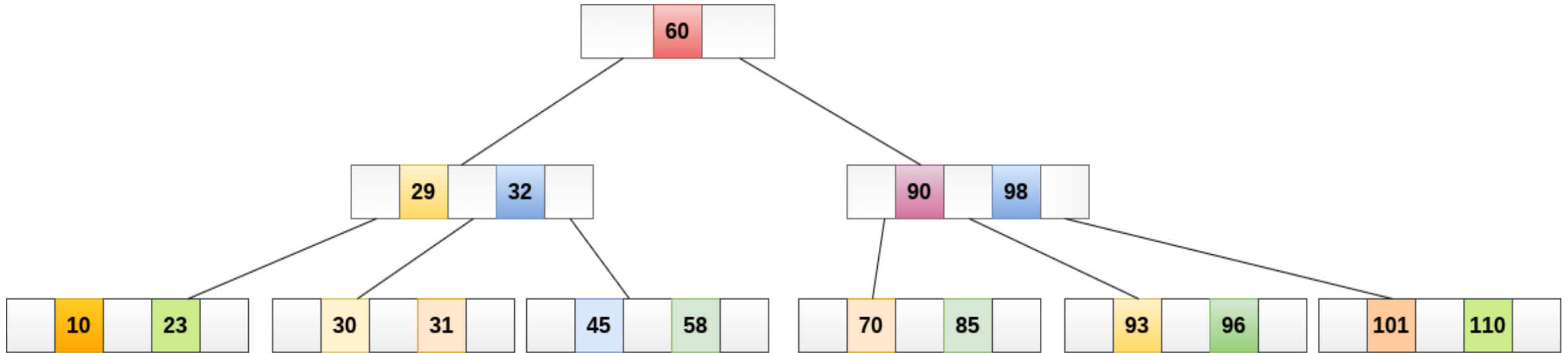
# Treap

- Treap data structure came from the **Tree and Heap data structure** So, it **comprises the properties of both** Tree and Heap data structures.
- In Binary search tree, each node on the left subtree must be equal or less than the value of the root node and each node on the right subtree must be equal or greater than the value of the root node.
- In heap data structure, both **right and left subtrees contain larger keys than the root**; therefore, we can say that the **root node contains the lowest value**.
- In treap data structure, each node has both *key* and *priority* where **key is derived from the Binary search tree** and **priority is derived from the heap data structure**.
- The **Treap** data structure follows two properties which are given below:
  - **Right child of a node  $\geq$  current node and left child of a node  $\leq$  current node (binary tree)**
  - **Children of any subtree must be greater than the node (heap)**

# B-tree

- B-tree is a balanced **m-way** tree where **m** defines the order of the tree.
- Till now, we read that the node contains only one key but **b-tree can have more than one key, and more than 2 children.**
- It always **maintains the sorted data.**
- In binary tree, it is possible that **leaf nodes can be at different levels**, but in b-tree, all the **leaf nodes must be at the same level.**
- **If order is m then node has the following properties:**
  - Each node in a b-tree can have maximum **m** children
  - For minimum children, a leaf node has 0 children, root node has minimum 2 children and internal node has minimum ceiling of  $m/2$  children. For example, the value of m is 5 which means that a node can have 5 children and internal nodes can contain maximum 3 children.
  - Each node has maximum  $(m-1)$  keys.
- The root node must contain minimum 1 key and all other nodes must contain at least **ceiling of  $m/2$  minus 1** keys.

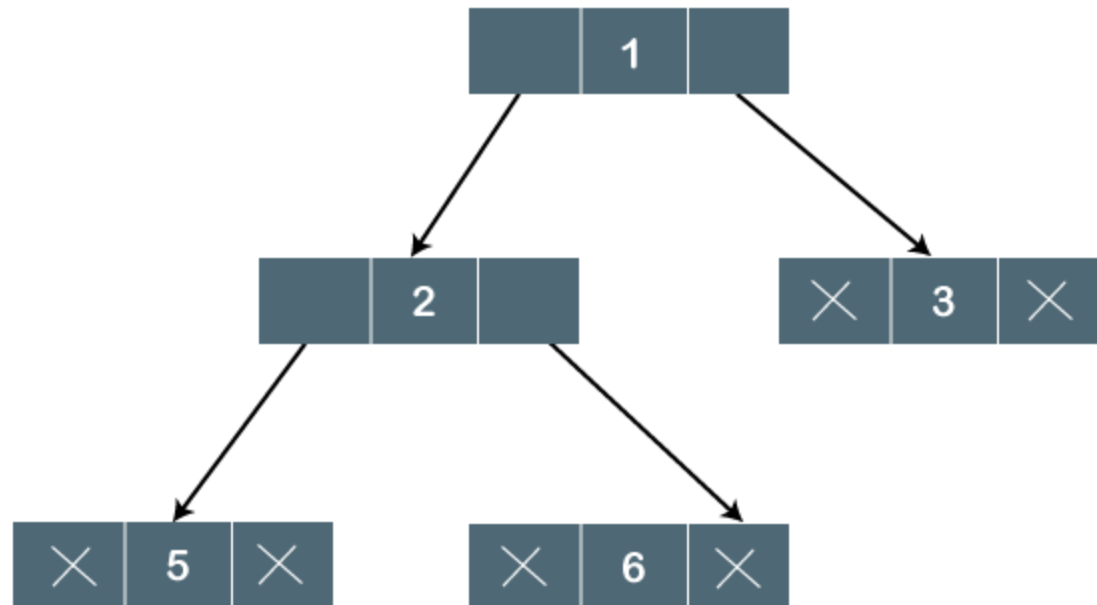
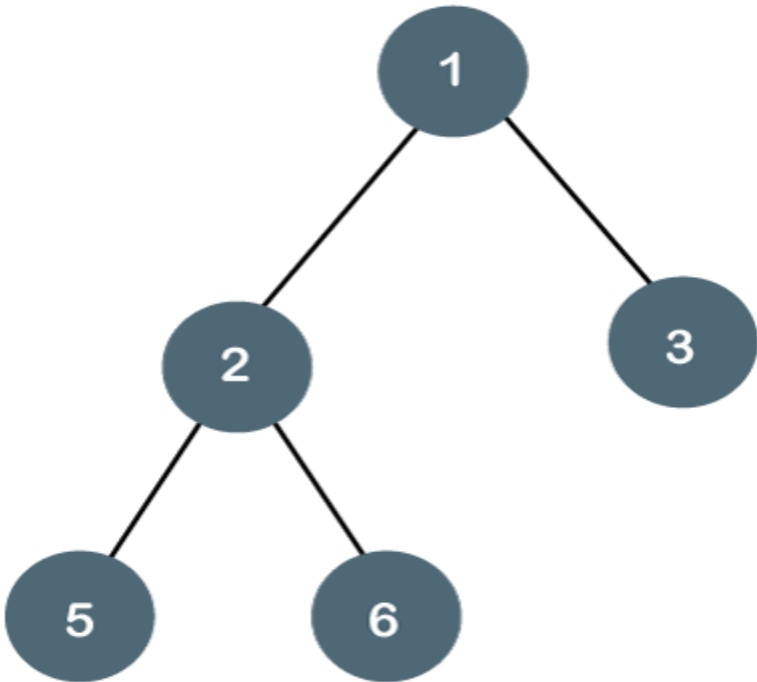
# B-tree



A B tree of order 4

# Binary Tree

- The Binary tree means that the node can have **maximum two children**.
- Here, binary name itself suggests that 'two'; therefore, each node can have either **0, 1 or 2 children**.



# Properties of Binary Tree

- At each level of  $i$ , the maximum number of nodes is  $2^i$ .
- The height of the tree is defined as the longest path from the root node to the leaf node.
- The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to  $(1+2+4+8) = 15$ .
- In general, the maximum number of nodes possible at height  $h$  is  $(2^0 + 2^1 + 2^2 + \dots + 2^h) = 2^{h+1} - 1$ .
- The minimum number of nodes possible at height  $h$  is equal to  $h+1$ .
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

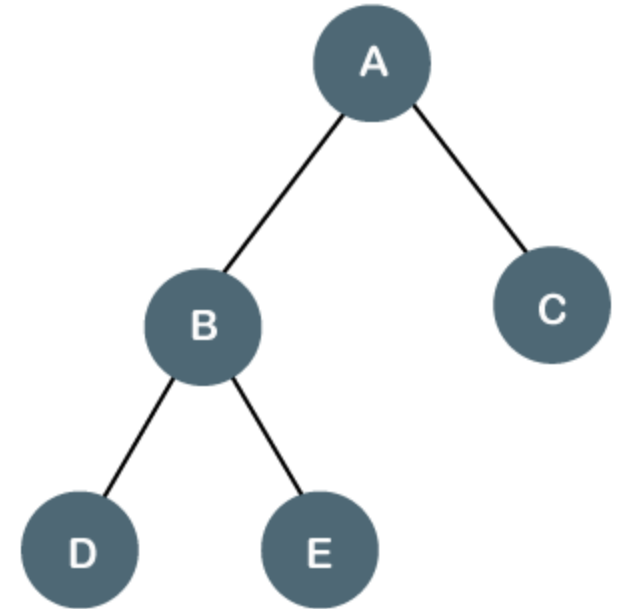
# Types of Binary Tree

- **There are four types of Binary tree:**
  - **Full/ proper/ strict Binary tree**
  - **Complete Binary tree**
  - **Perfect Binary tree**
  - **Degenerate Binary tree**
  - **Balanced Binary tree**



# 1. Full/ proper/ strict Binary tree

- The **full binary tree** is also known as a **strict binary tree**.
- The tree can only be considered as the full binary tree if **each node must contain either 0 or 2 children**.
- The full binary tree can also be defined as the tree in which each node **must contain 2 children except the leaf nodes**.

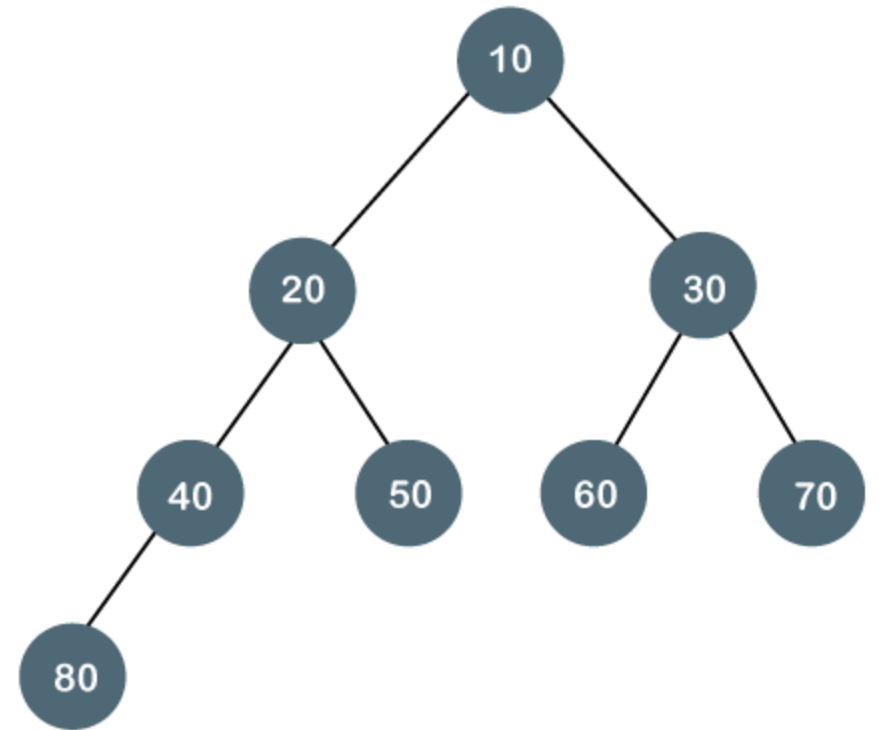


# Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e.,  $2^{h+1} - 1$ .
- The minimum number of nodes in the full binary tree is  $2*h-1$ .
- The minimum height of the full binary tree is  **$\log_2(n+1) - 1$** .
- The maximum height of the full binary tree can be computed as:
  - $n = 2*h - 1$
  - $n+1 = 2*h$
  - **$h = (n+1)/2$**

# Complete Binary Tree

- The complete binary tree is a tree in which all the nodes are **completely filled except the last level**.
- In the last level, all the nodes must be as **left** as possible.
- In a complete binary tree, the nodes **should be added from the left**.
- The beside tree is a complete binary tree because all the nodes are completely filled, and all the nodes in the last level are added at the left first.

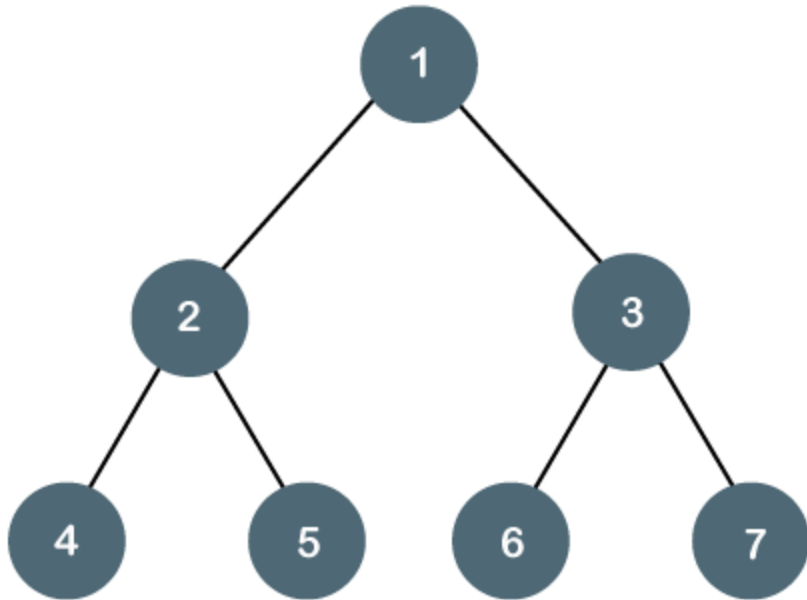


# Properties of Complete Binary Tree

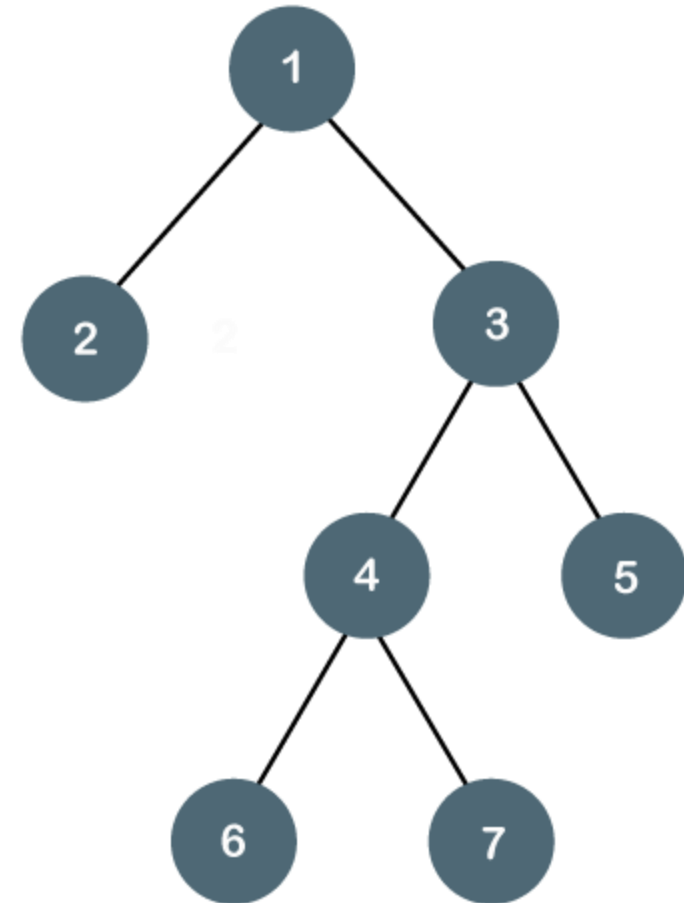
- The maximum number of nodes in complete binary tree is  $2^{h+1} - 1$ .
- The minimum number of nodes in complete binary tree is  $2^h$ .
- The minimum height of a complete binary tree is  $\log_2(n+1) - 1$ .

# Perfect Binary Tree

- A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.



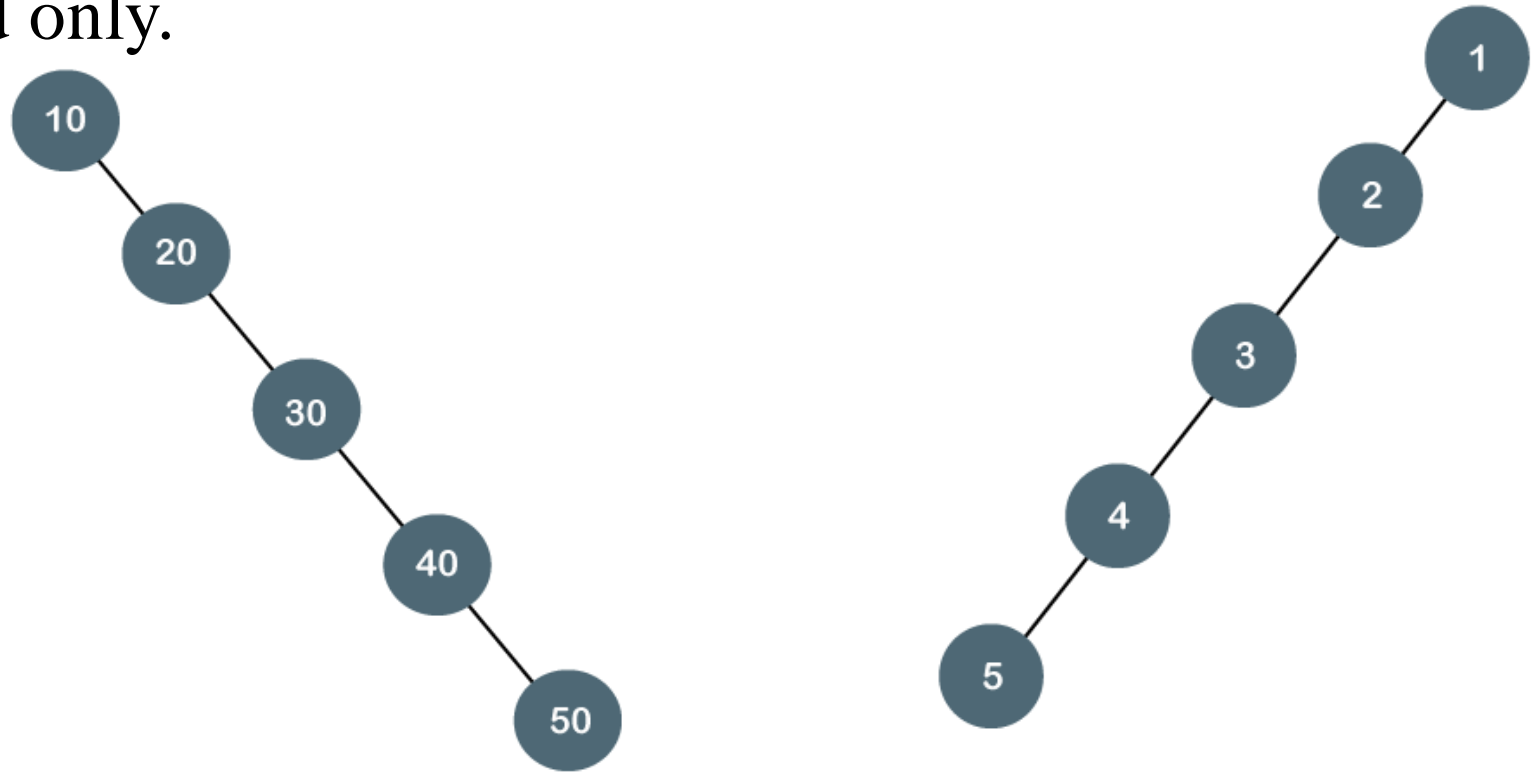
perfect binary tree.



tree is not a perfect binary tree because all the leaf nodes are not at the same level.

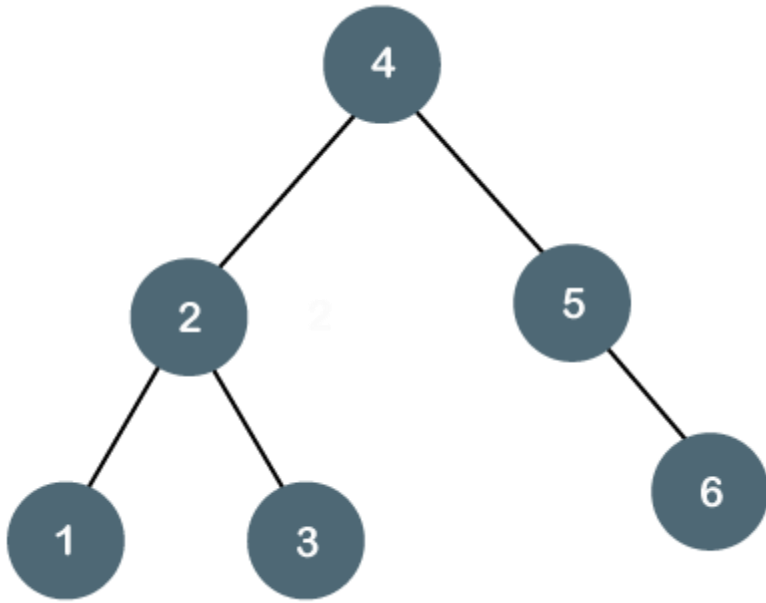
# Degenerate Binary Tree

- The degenerate binary tree is a tree in which all the internal nodes have only one children.
- The below tree is a degenerate binary tree because all the nodes have only one child. It is also known as a right-skewed tree as all the nodes have a right child only.

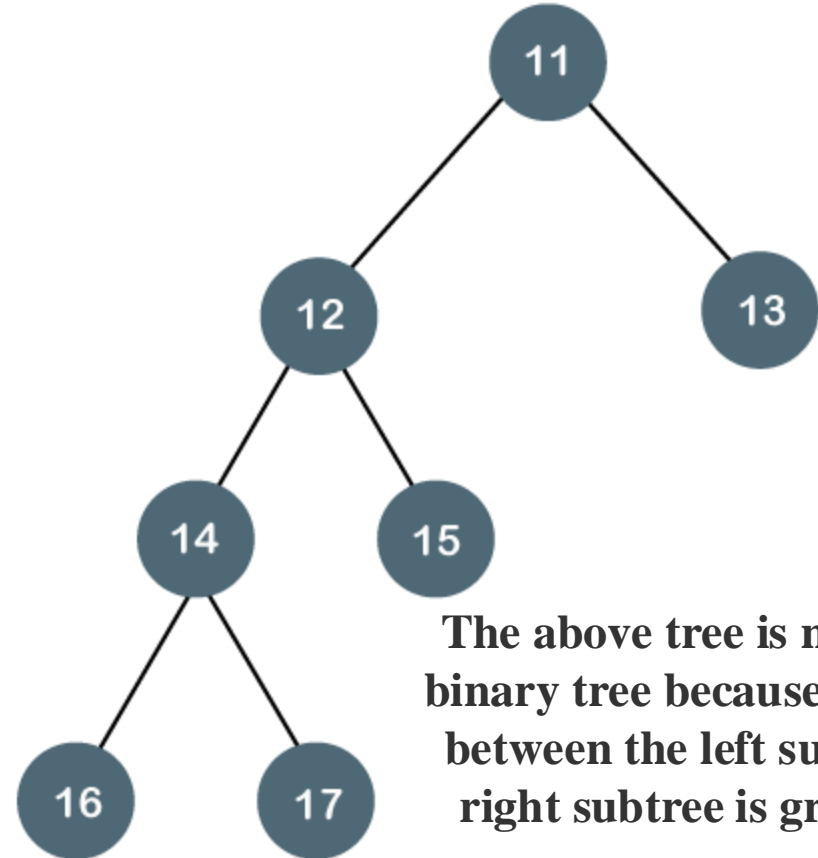


# Balanced Binary Tree

- The balanced binary tree is a tree in which both **the left and right trees differ by atmost 1**. For example, *AVL* and *Red-Black trees* are balanced binary tree.



The above tree is a balanced binary tree because the difference between the left subtree and right subtree is zero.



The above tree is not a balanced binary tree because the difference between the left subtree and the right subtree is greater than 1.

# Binary Tree Implementation

- A Binary tree is implemented **with the help of pointers**.
- The **first node** in the tree is represented **by the root pointer**.
- Each node in the tree consists of **three parts**, i.e., **data, left pointer and right pointer**.
- To create a binary tree, we first need to create the node. We will create the node of user-defined as shown below:

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

- In the above structure, **data** is the value, **left pointer** contains the address of the left node, and **right pointer** contains the address of the right node.



# Binary Tree program in C

```
#include<stdio.h>

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

void main()
{
    struct node *root;
    root = create();
}

struct node *create()
{
    struct node *temp;
    int data;
    temp = (struct node *)malloc(sizeof(struct node));
    printf("Press 0 to exit");
    printf("\nPress 1 for new node");
```

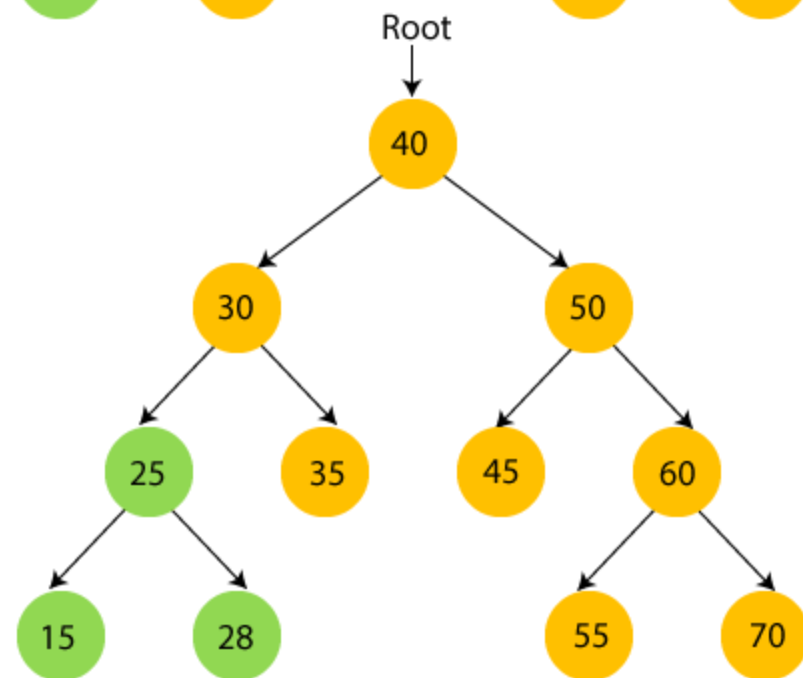
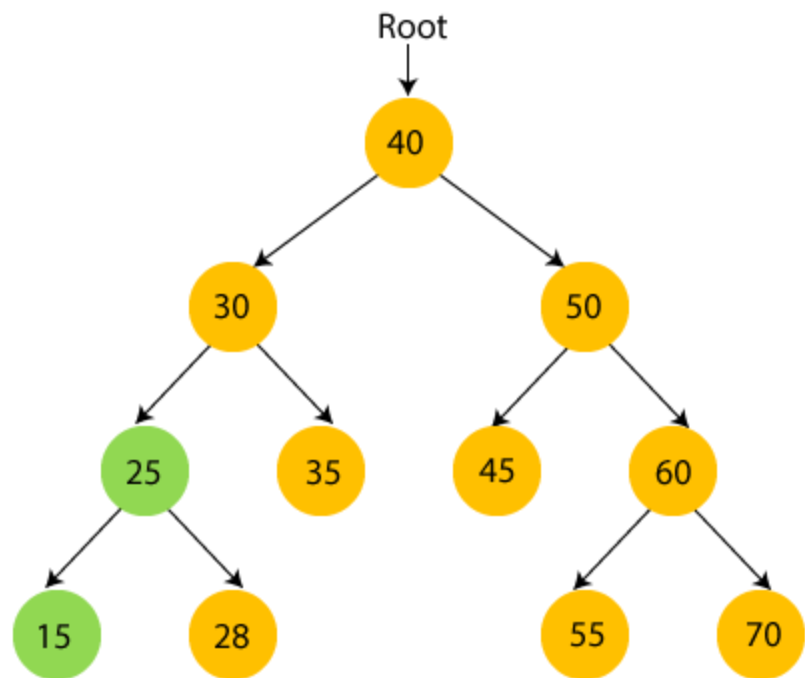
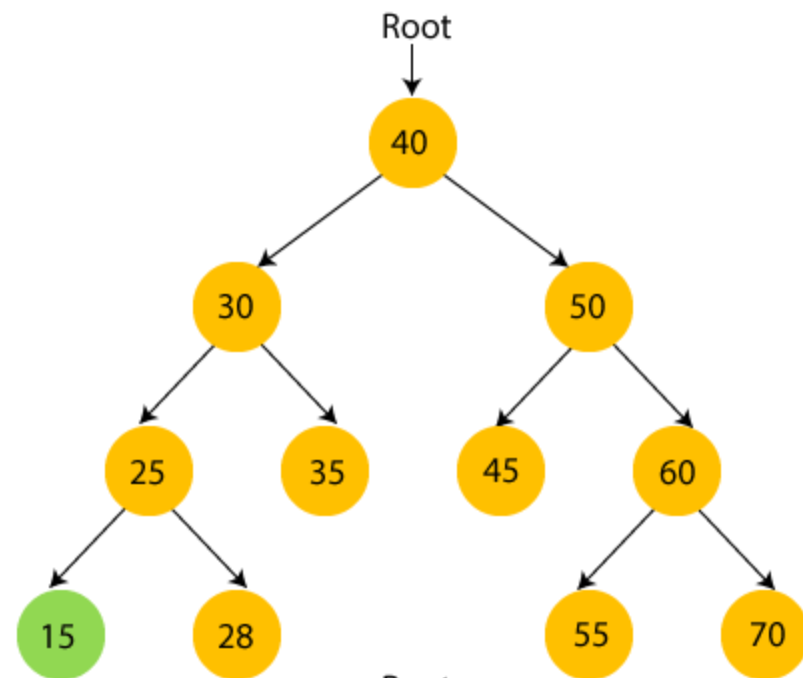
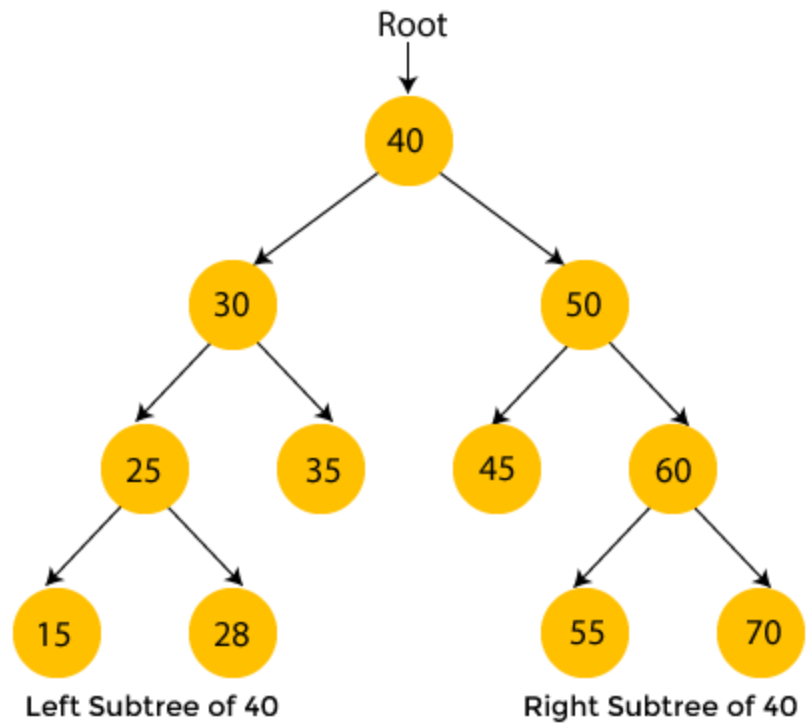
```
printf("Enter your choice : ");
    scanf("%d", &choice);
    if(choice==0)
    {
        return 0;
    }
    else
    {
        printf("Enter the data:");
        scanf("%d", &data);
        temp->data = data;
        printf("Enter the left child of %d", data);
        temp->left = create();
        printf("Enter the right child of %d", data);
        temp->right = create();
        return temp;
    } }
```

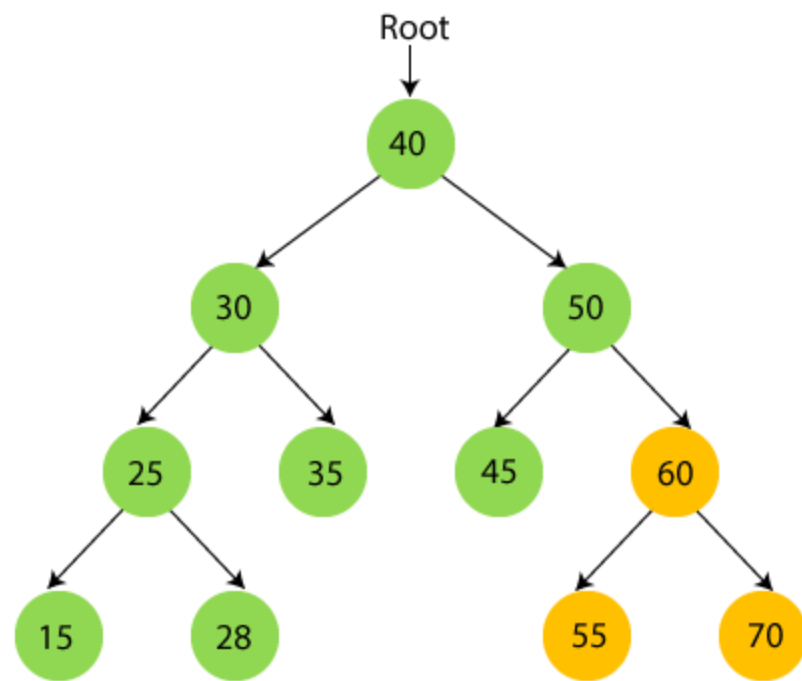
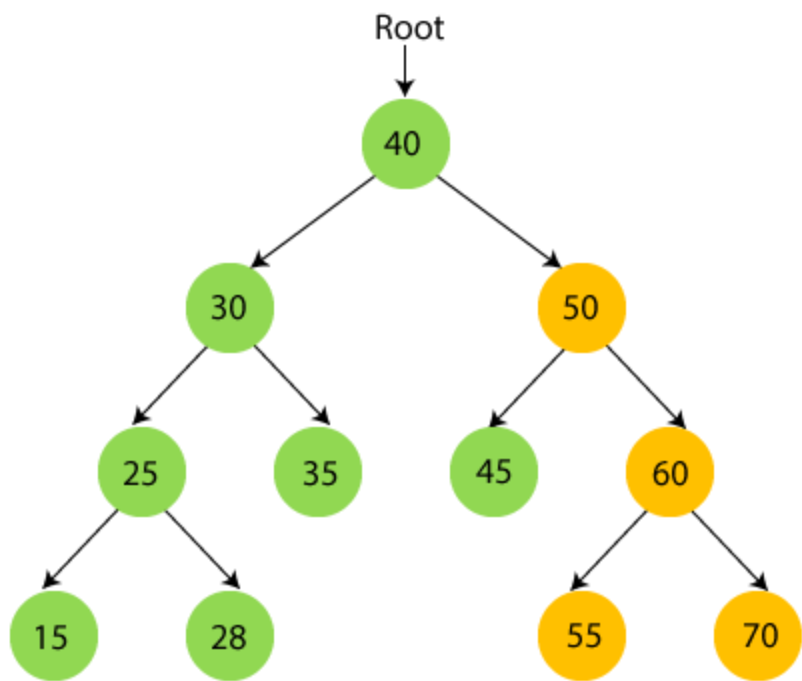
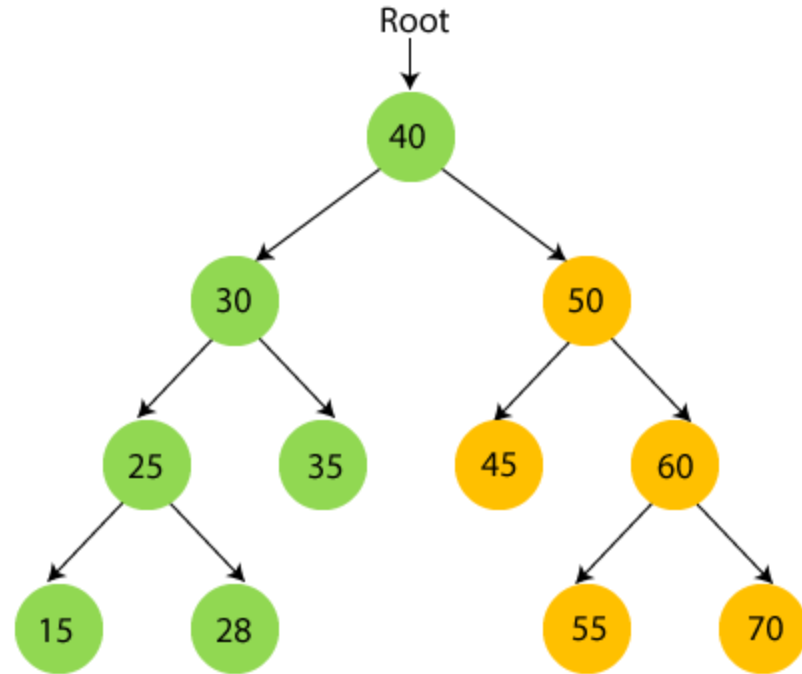
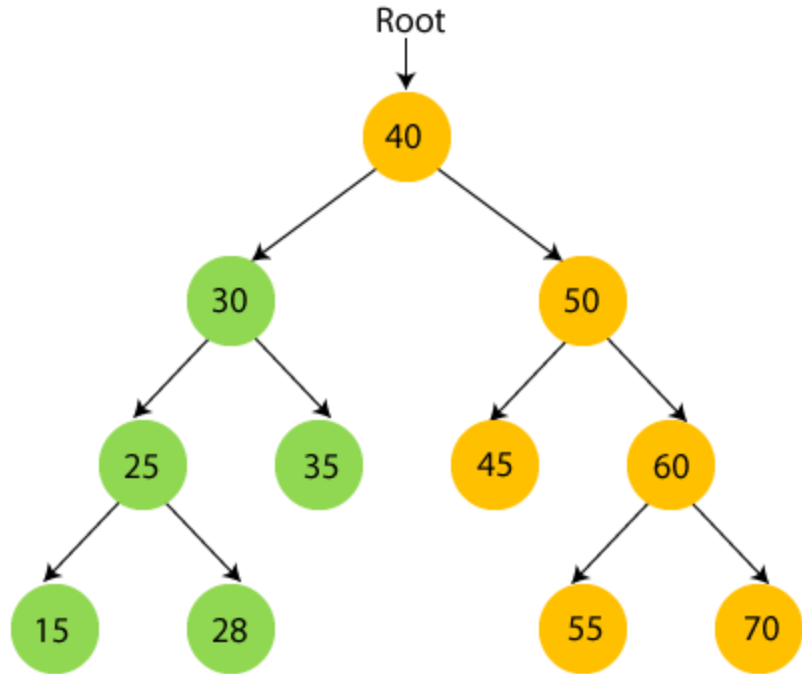
# Types of Traversal of Binary Tree

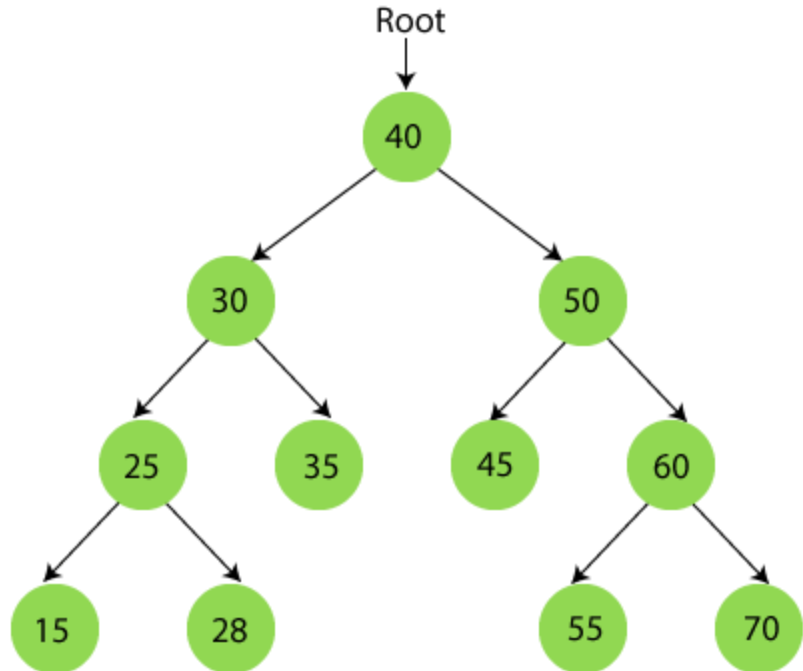
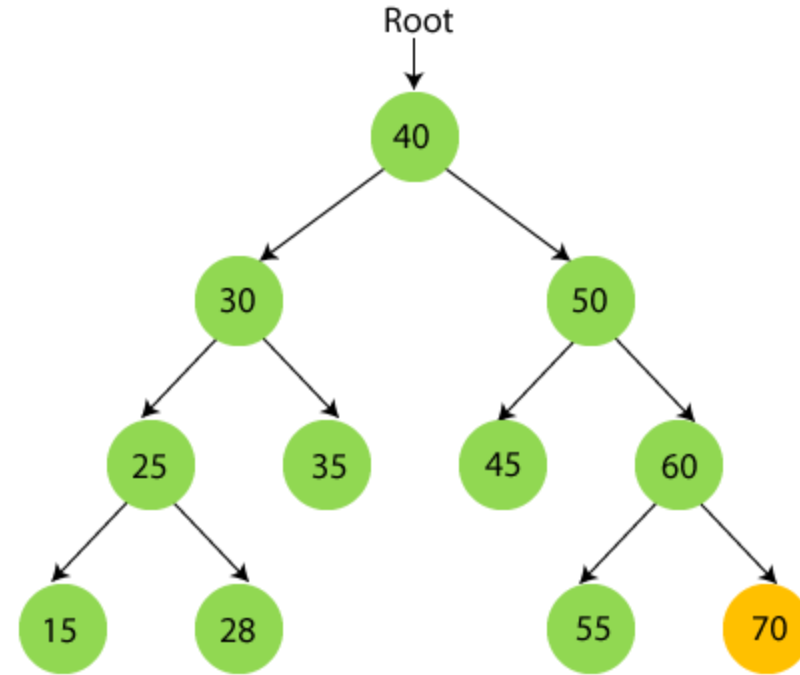
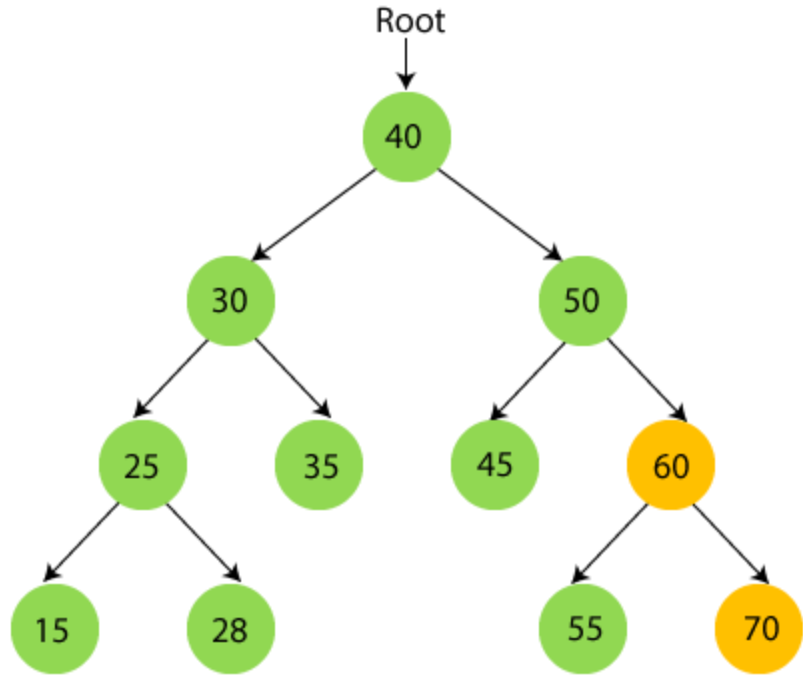
- There are three types of traversal of a binary tree.
  - Inorder tree traversal
  - Preorder tree traversal
  - Postorder tree traversal

# Inorder Tree Traversal(LEFT-ROOT- RIGHT)

- If we want to traverse the nodes in ascending order, then we use the inorder traversal. Following are the steps required for the inorder traversal:
  - Visit all the nodes in the left subtree
  - Visit the root node
  - Visit all the nodes in the right subtree
- Linear data structures such as stack, array, queue, etc., only have one way to traverse the data. But in hierarchical data structures such as **tree**, there are multiple ways to traverse the data.







After the completion of inorder traversal, the final output is –

**{15, 25, 28, 30, 35, 40, 45, 50, 55, 60, 70}**

# Algorithm

- *Inorder(root):*

1. *If root is NULL, then return*

2. *Inorder (root -> left)*

3. *Process root (For example, print root's data)*

4. *Inorder (root -> right)*

```

#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;

    struct node* left;
    struct node* right;
};

struct node* createNode(int val)
{
    struct node* Node = (struct node*)malloc(sizeof(struct
node));

    Node->data = val;
    Node->left = NULL;
    Node->right = NULL;

    return (Node); }

void traverseInorder(struct node* root)
{
    if (root == NULL)
        return;

```

```

traverseInorder(root->left);
    printf(" %d ", root->data);
    traverseInorder(root->right); }

int main()
{
    struct node* root = createNode(40);
    root->left = createNode(30);
    root->right = createNode(50);
    root->left->left = createNode(25);
    root->left->right = createNode(35);
    root->left->left->left = createNode(15);
    root->left->left->right = createNode(28);
    root->right->left = createNode(45);
    root->right->right = createNode(60);
    root->right->right->left = createNode(55);
    root->right->right->right = createNode(70);
    printf("\n The Inorder traversal of given binary tree is -\n");
    traverseInorder(root);

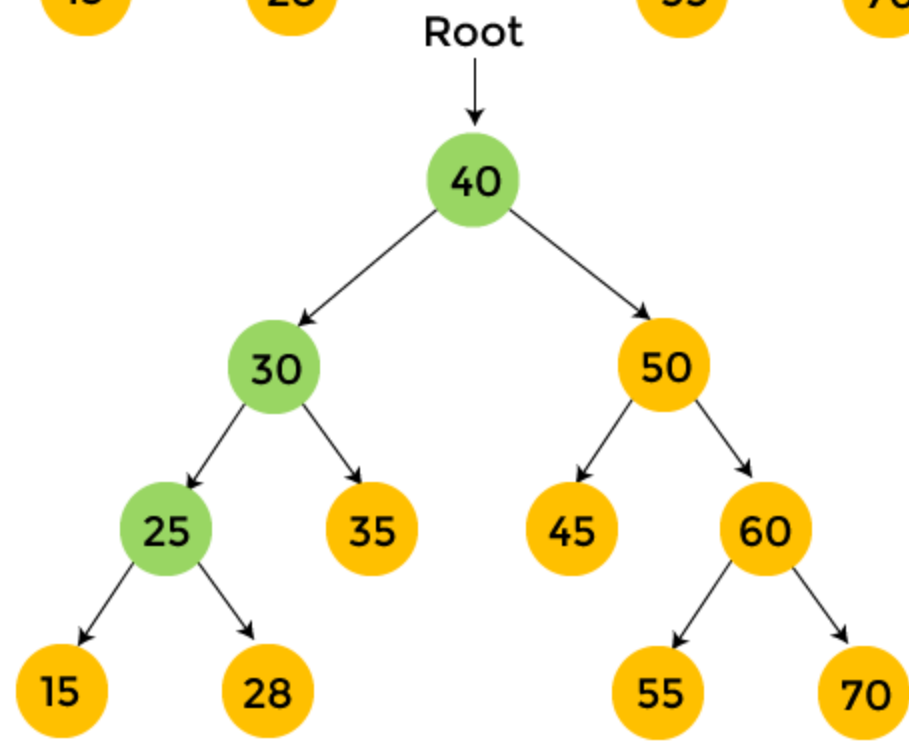
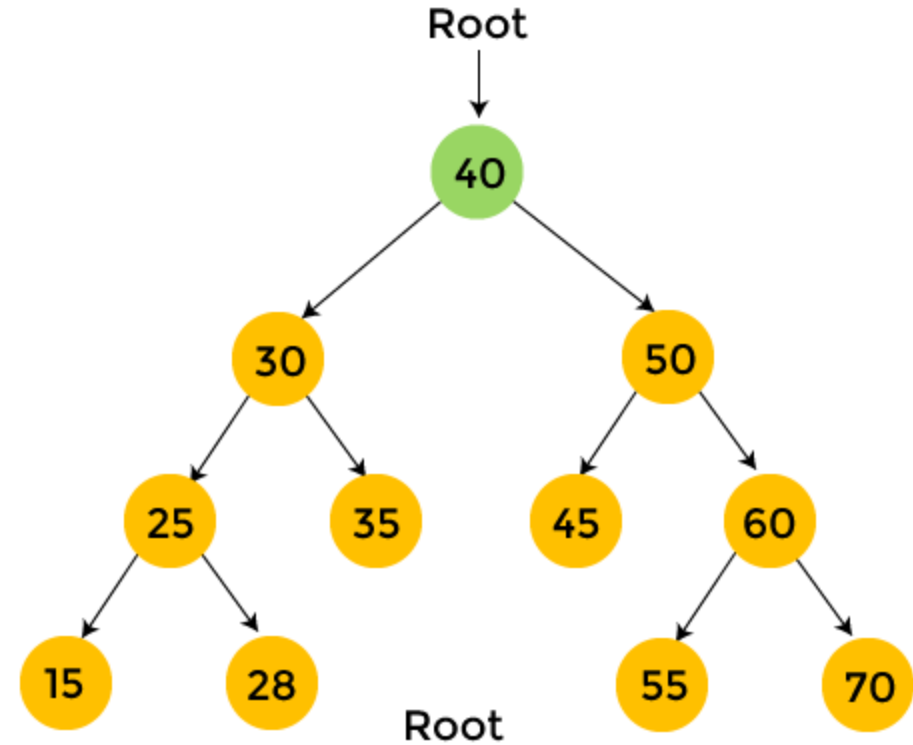
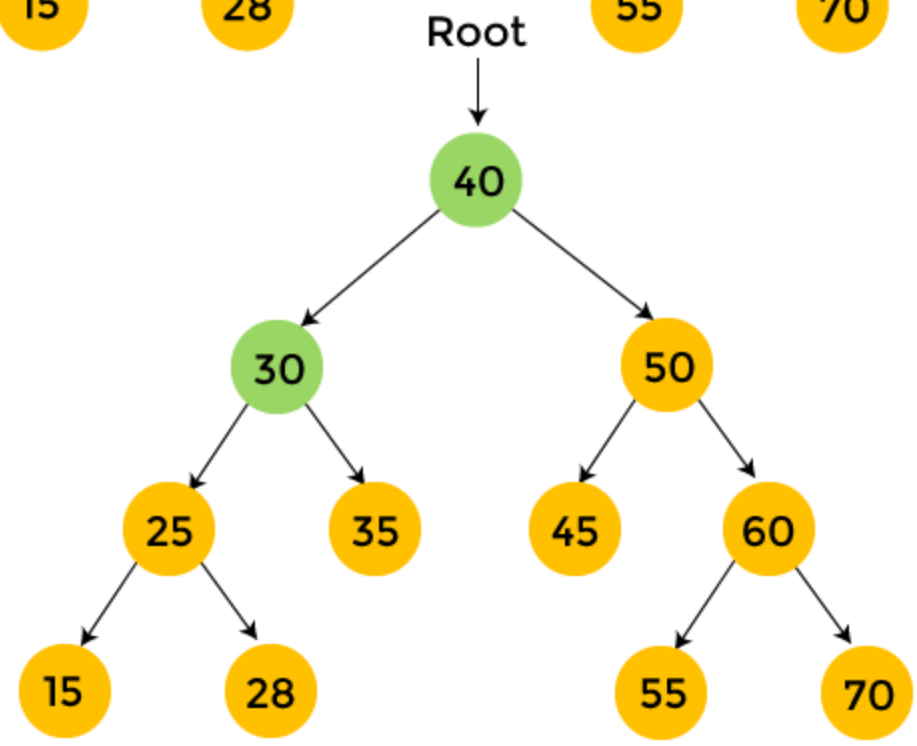
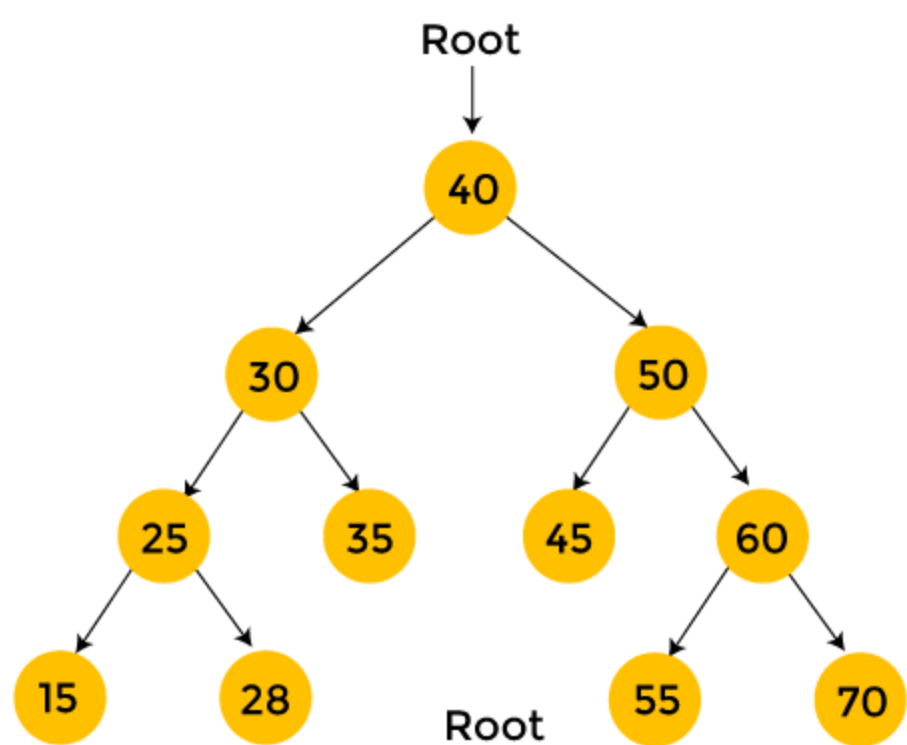
    return 0; }

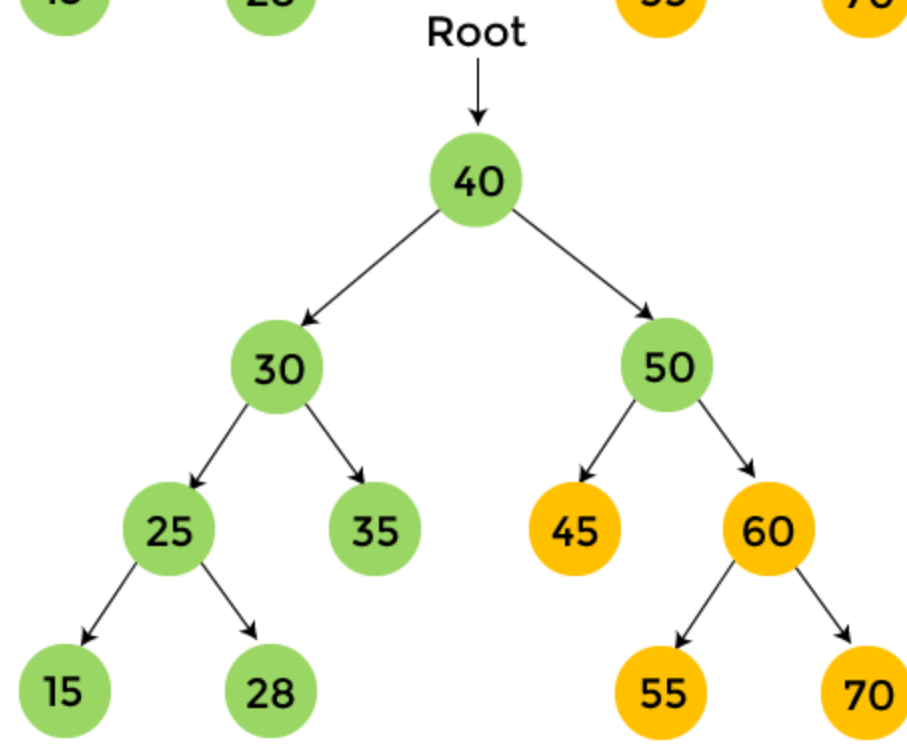
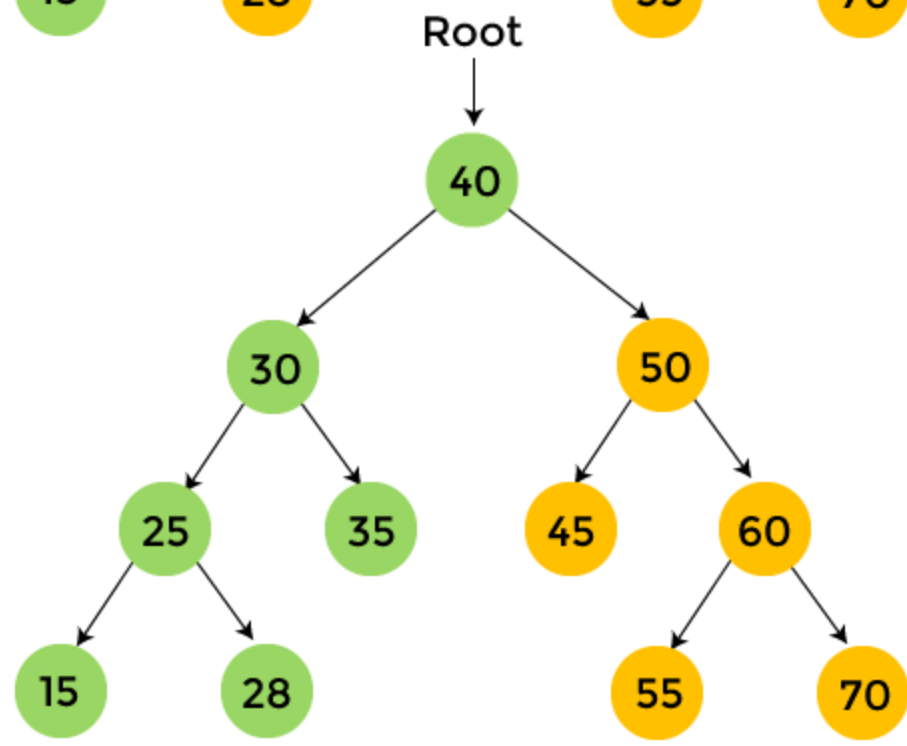
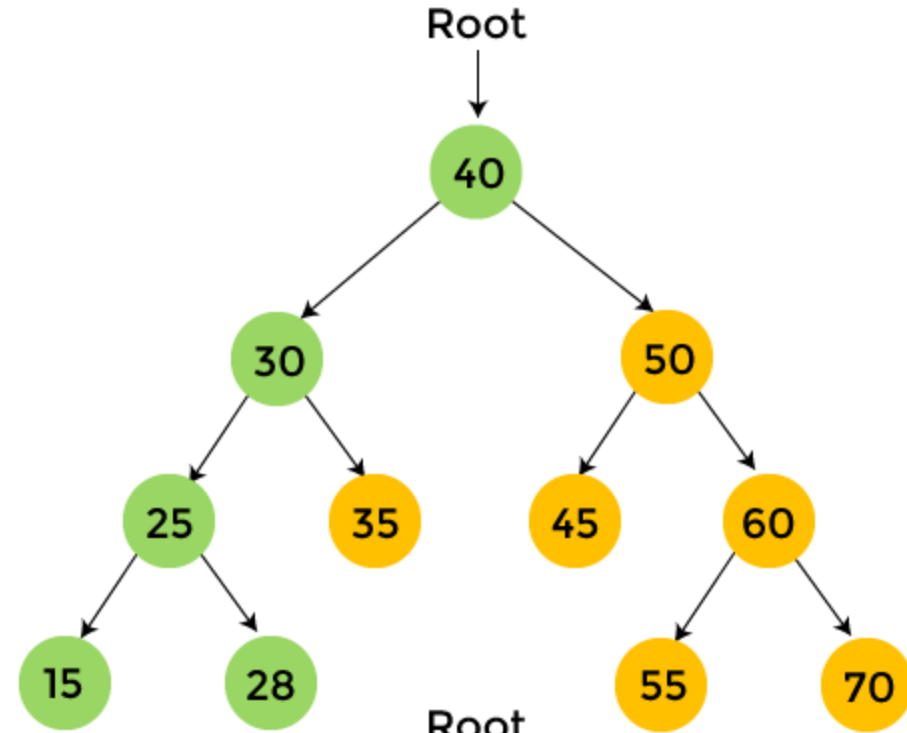
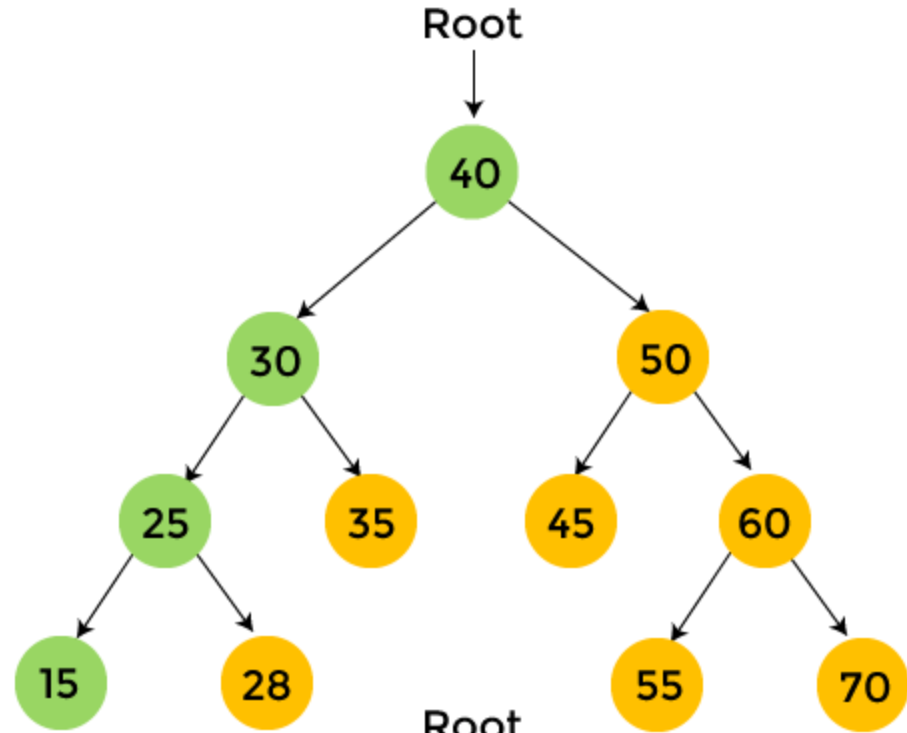
```

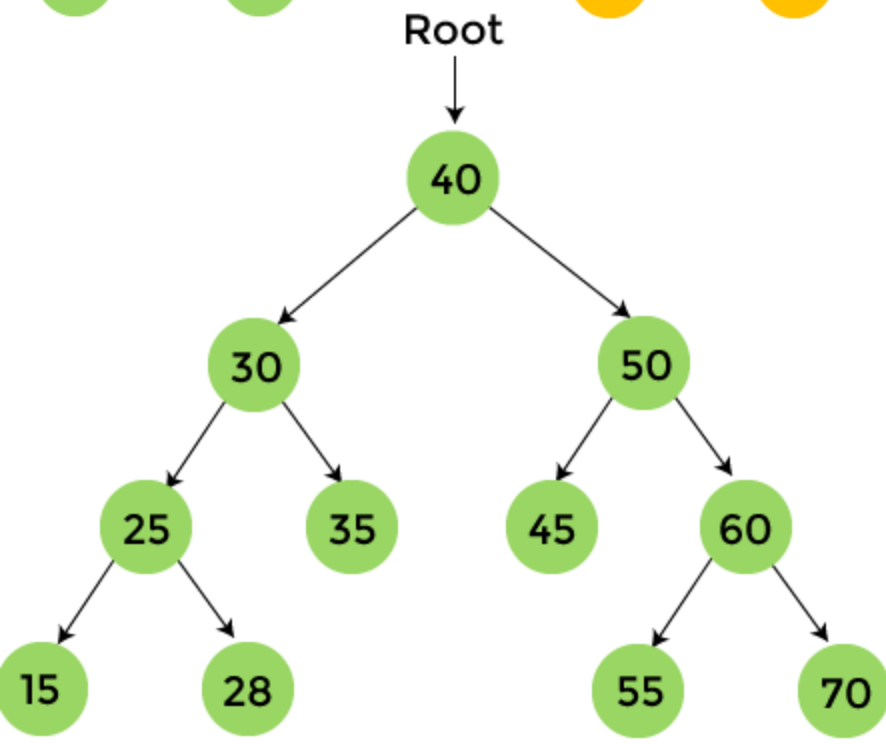
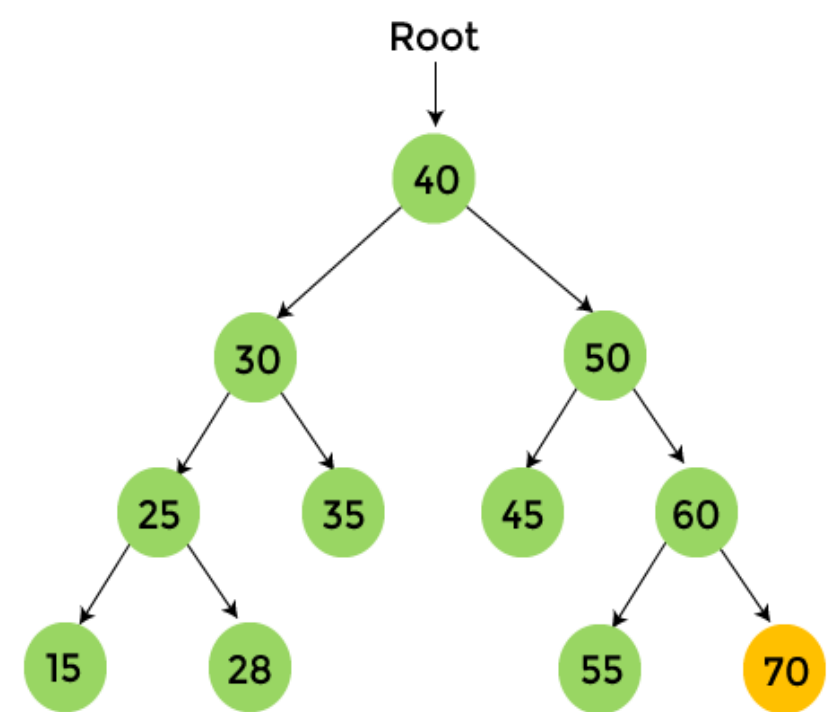
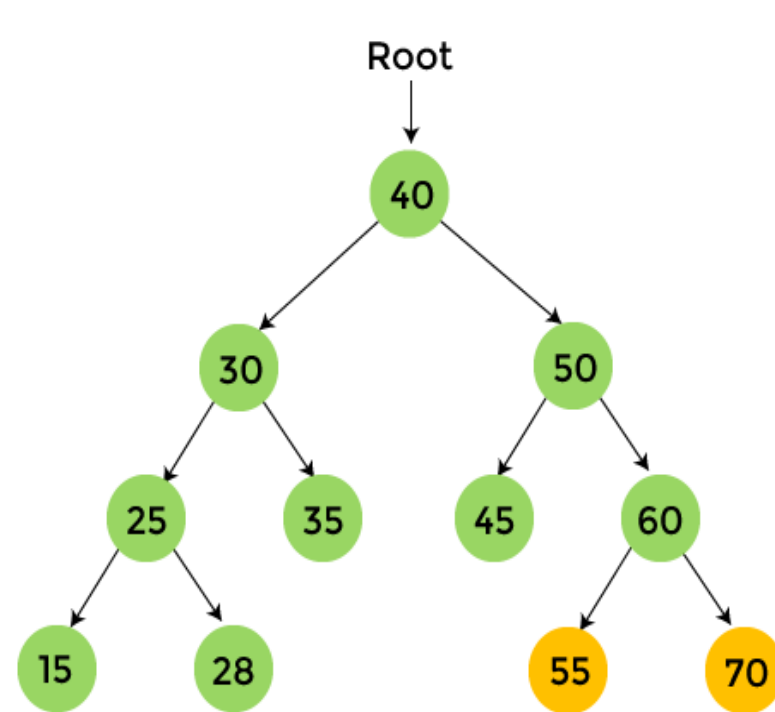
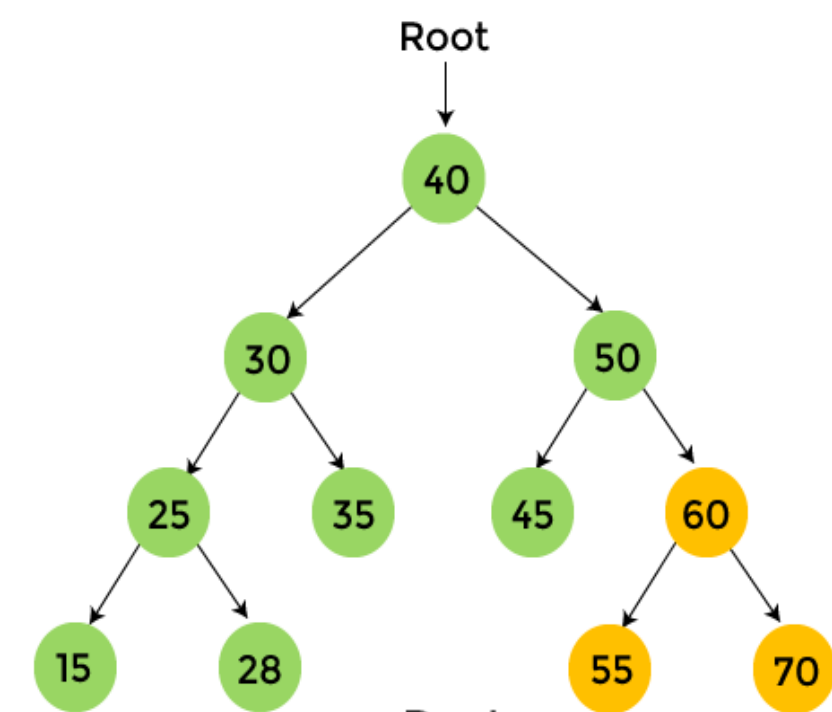


# Preorder Traversal (ROOT-LEFT-RIGHT)

- In preorder traversal, first, root node is visited, then left sub-tree and after that right sub-tree is visited. The process of preorder traversal can be represented as -
- **root  $\rightarrow$  left  $\rightarrow$  right**
- Root node is always traversed first in preorder traversal.
- Preorder traversal is **used to get the prefix expression of a tree.**
- The steps to perform the preorder traversal are listed as follows -
  - First, visit the root node.
  - Then, visit the left subtree.
  - At last, visit the right subtree.
- The preorder traversal technique follows the **Root Left Right** policy. The name preorder itself suggests that the root node would be traversed first.







After the completion of preorder traversal, the final output is -  
**40, 30, 25, 15, 28, 35, 50, 45, 60, 55, 70**

# Algorithm

- Now, let's see the algorithm of preorder traversal.
  - Step 1: Repeat Steps 2 to 4 **while** ROOT != NULL
  - Step 2: Write ROOT -> DATA
  - Step 3: PREORDER(ROOT -> LEFT)
  - Step 4: PREORDER(ROOT -> RIGHT)
  - [END OF LOOP]
  - Step 5: END

```

#include <stdio.h>
#include <stdlib.h>

struct node {
    int element;
    struct node* left;
    struct node* right; };

struct node* createNode(int val)
{
struct node* Node = (struct node*)malloc(sizeof(struct node));
    Node->element = val;

    Node->left = NULL;
    Node->right = NULL;

    return (Node); }

void traversePreorder(struct node* root)
{ if (root == NULL)
    return;

    printf(" %d ", root->element);
    traversePreorder(root->left);
    traversePreorder(root->right); }

```

```

int main()
{
    struct node* root = createNode(40);
    root->left = createNode(30);
    root->right = createNode(50);
    root->left->left = createNode(25);
    root->left->right = createNode(35);
    root->left->left->left = createNode(15);
    root->left->left->right = createNode(28);
    root->right->left = createNode(45);
    root->right->right = createNode(60);
    root->right->right->left = createNode(55);
    root->right->right->right = createNode(70);

    printf("\n The Preorder traversal of given binary tree is -
\n");

    traversePreorder(root);

    return 0;
}

```

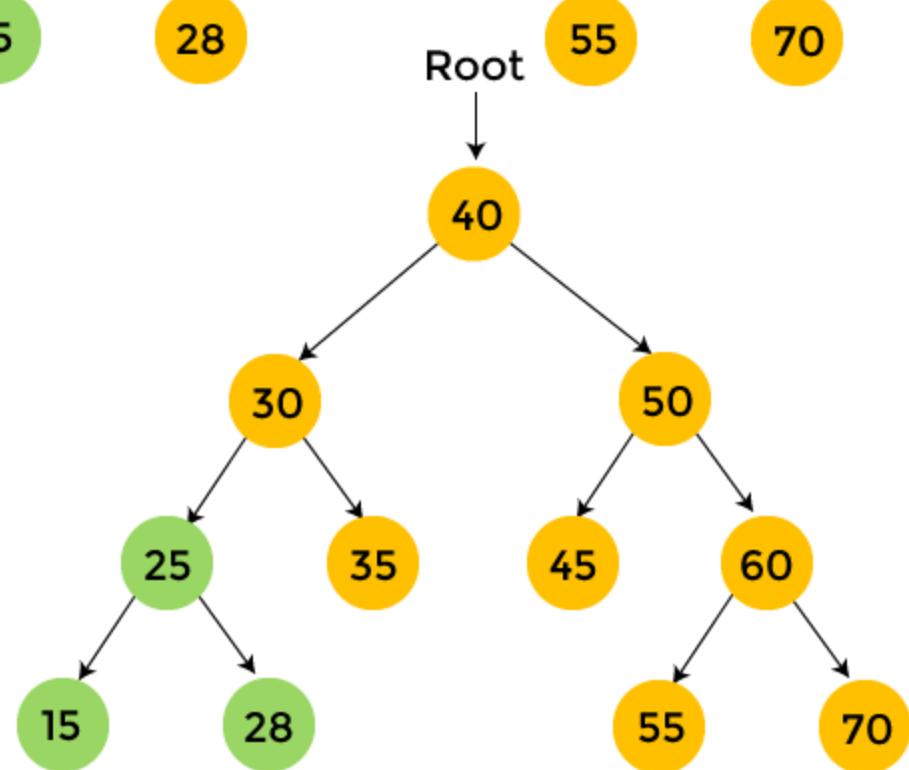
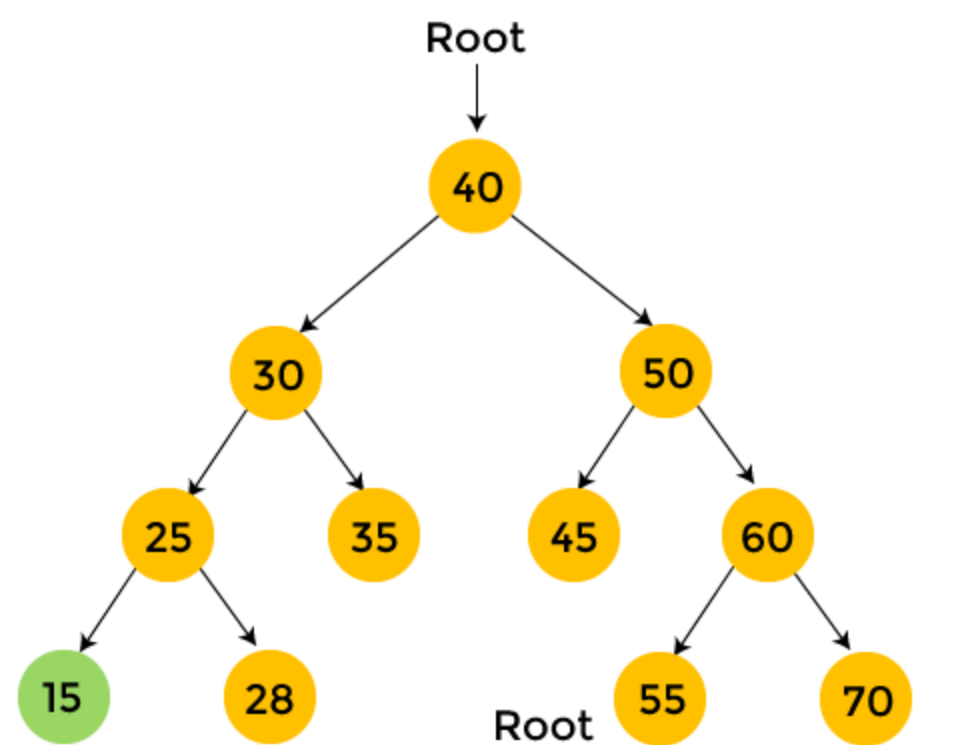
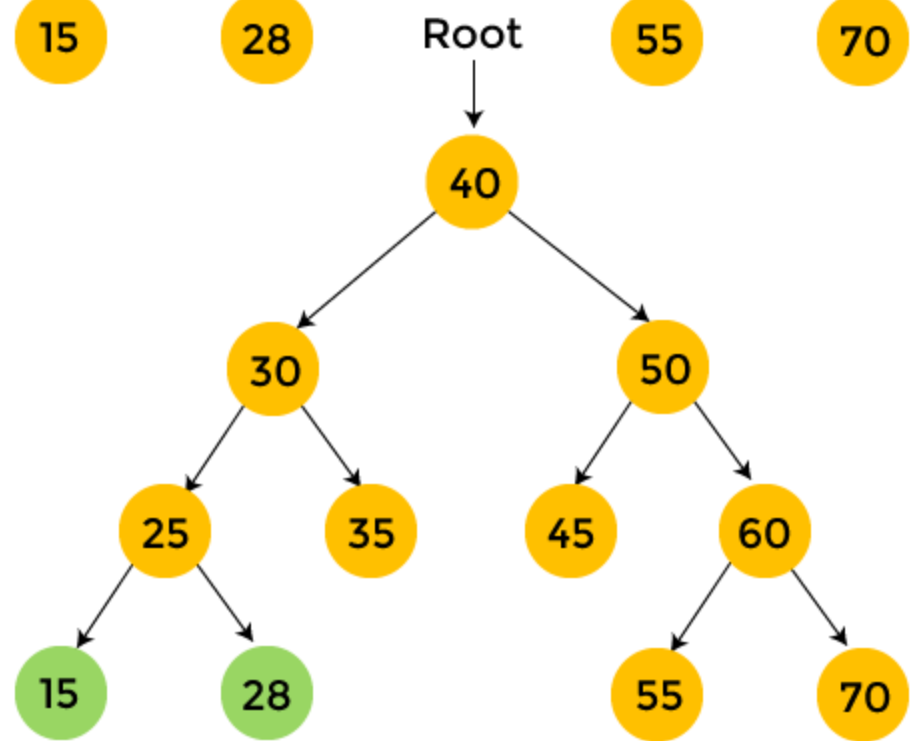
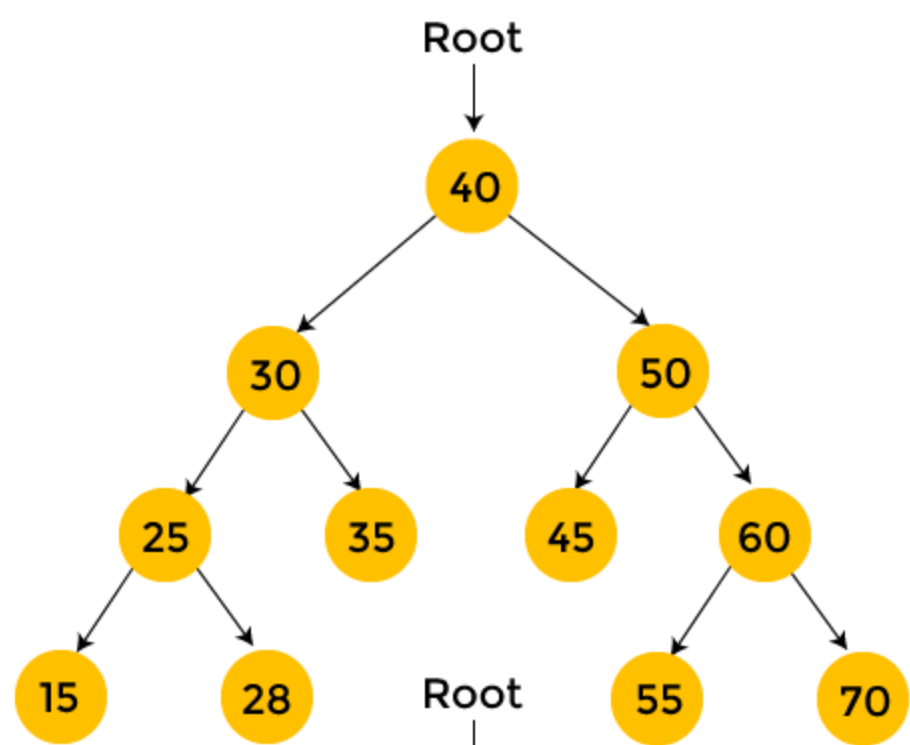
# Postorder Traversal (**Left-right-Root**).

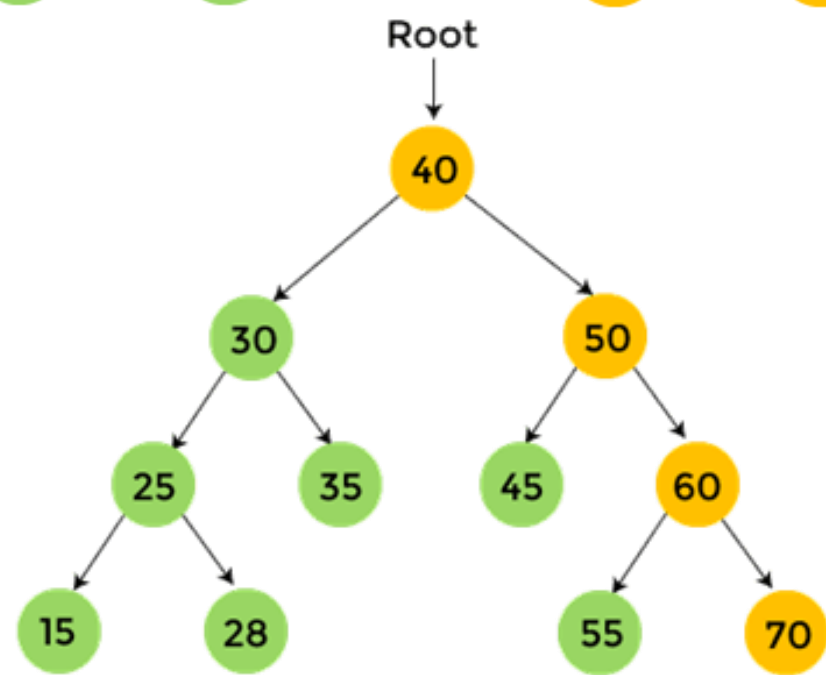
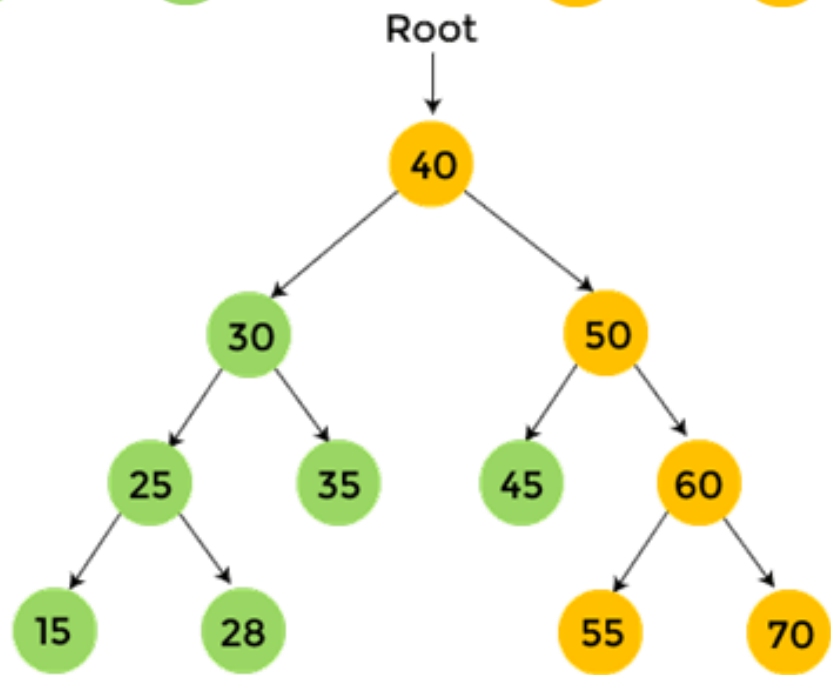
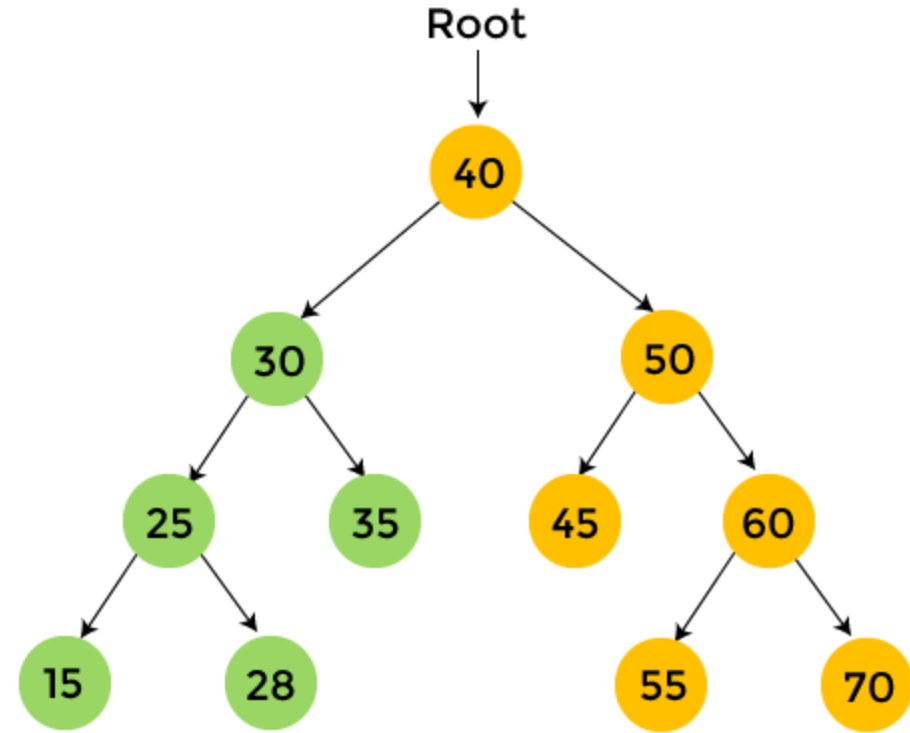
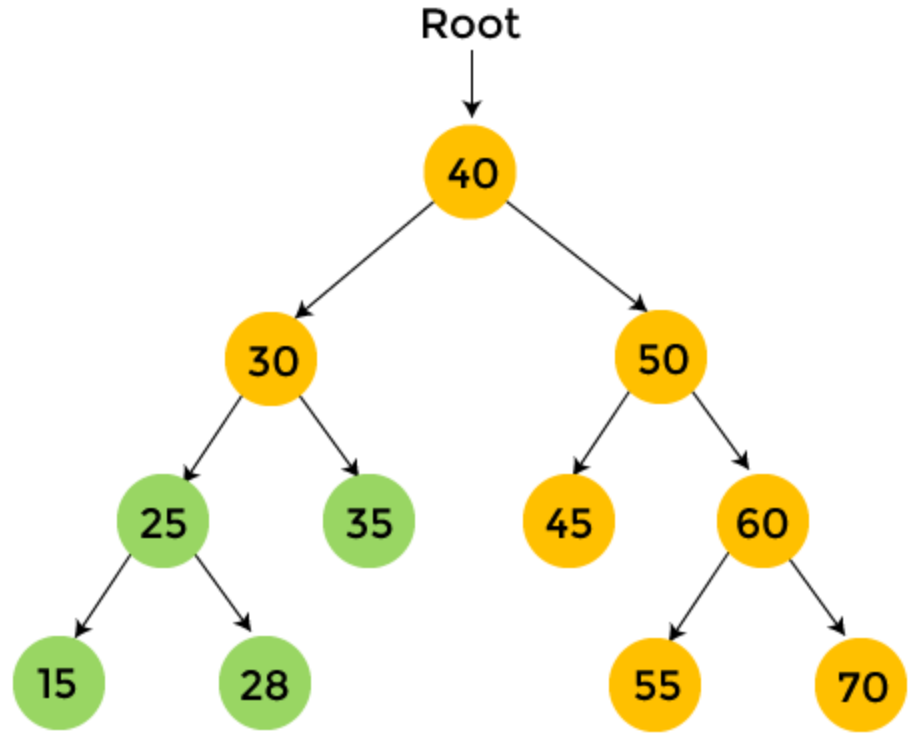
- The postorder traversal is one of the traversing techniques used for visiting the node in the tree. It follows the principle (**Left-right-Root**).
- Postorder traversal is .
- The following steps are used to perform the postorder traversal:
  - Traverse the left subtree by calling the postorder function recursively.
  - Traverse the right subtree by calling the postorder function recursively.
  - Access the data part of the current node.
- The post order traversal technique follows the **Left Right Root** policy.
- Here, Left Right Root means the left subtree of the root node is traversed first, then the right subtree, and finally, the root node is traversed.
- Here, the Postorder name itself suggests that the tree's root node would be traversed at last.

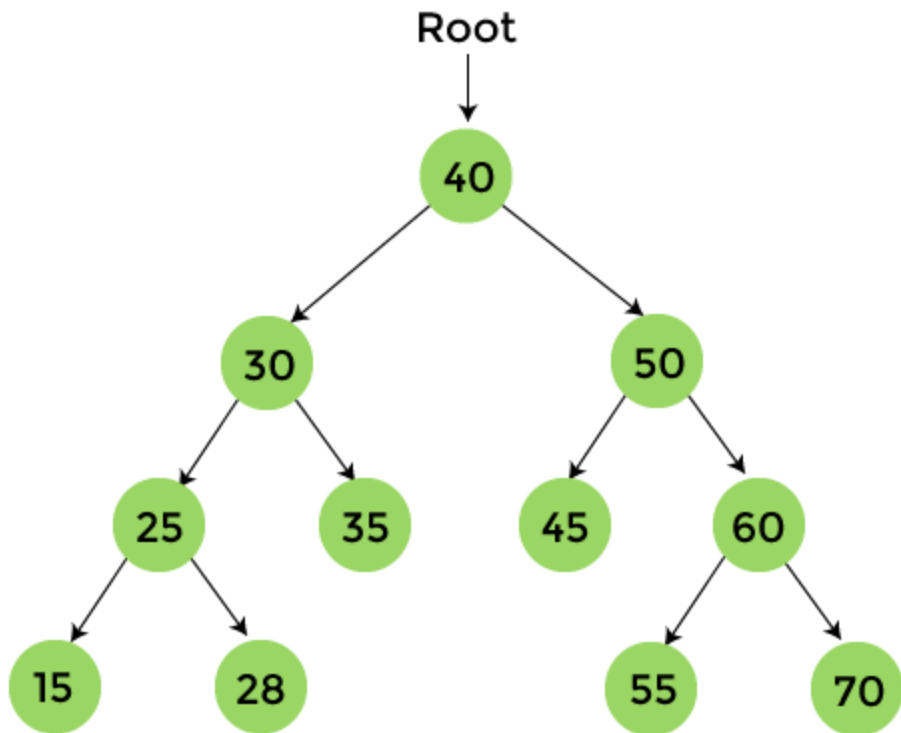
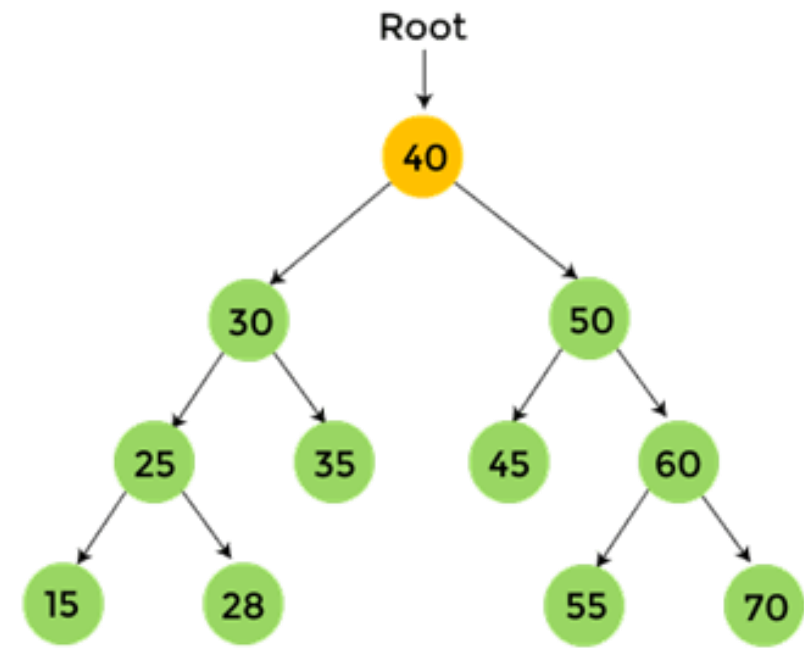
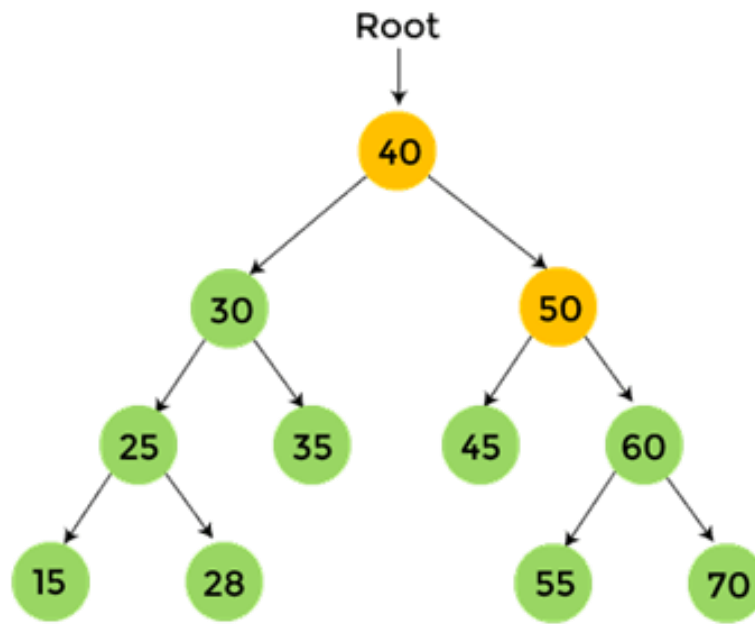
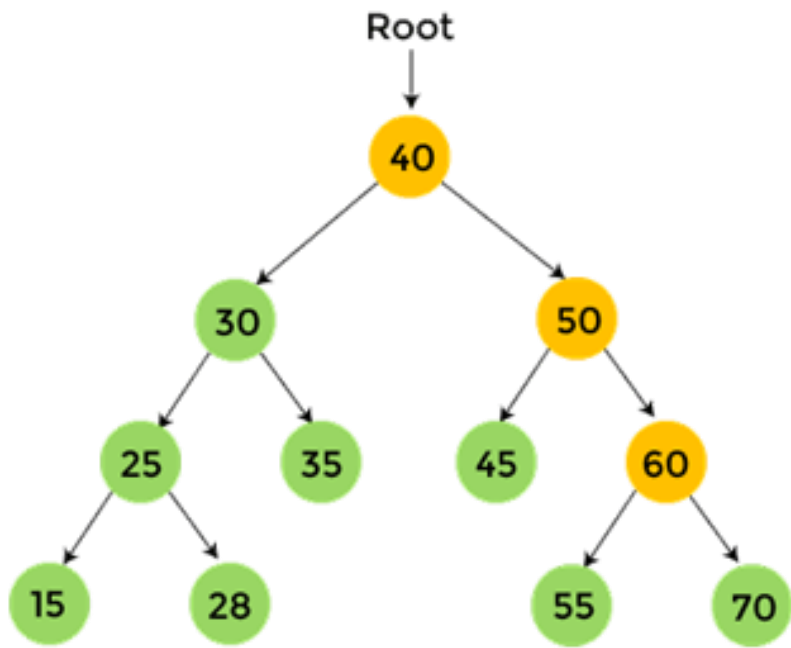
# Algorithm

- Now, let's see the algorithm of postorder traversal.
  - Step 1: Repeat Steps 2 to 4 **while** TREE != NULL
  - Step 2: POSTORDER(TREE -> LEFT)
  - Step 3: POSTORDER(TREE -> RIGHT)
  - Step 4: Write TREE -> DATA
  - [END OF LOOP]
  - Step 5: END









The final output that we will get after postorder traversal is -  
{15, 28, 25, 35, 30, 45, 55, 70, 60, 50, 40}

```

#include <stdio.h>
#include <stdlib.h>

struct node {
    struct node* left;
    struct node* right; };

struct node* createNode(int val)
{
    struct node* Node = (struct node*)malloc(sizeof(struct node));

    Node->element = val;
    Node->left = NULL;
    Node->right = NULL;
    return (Node); }

void traversePostorder(struct node* root)
{
    if (root == NULL)
        return;

    traversePostorder(root->left);
    traversePostorder(root->right);
    printf(" %d ", root->element); }

```

```

int main()
{
    struct node* root = createNode(40);
    root->left = createNode(30);
    root->right = createNode(50);
    root->left->left = createNode(25);
    root->left->right = createNode(35);
    root->left->left->left = createNode(15);
    root->left->left->right = createNode(28);
    root->right->left = createNode(45);
    root->right->right = createNode(60);
    root->right->right->left = createNode(55);
    root->right->right->right = createNode(70);

    printf("\n The Postorder traversal of given binary tree
is -\n");

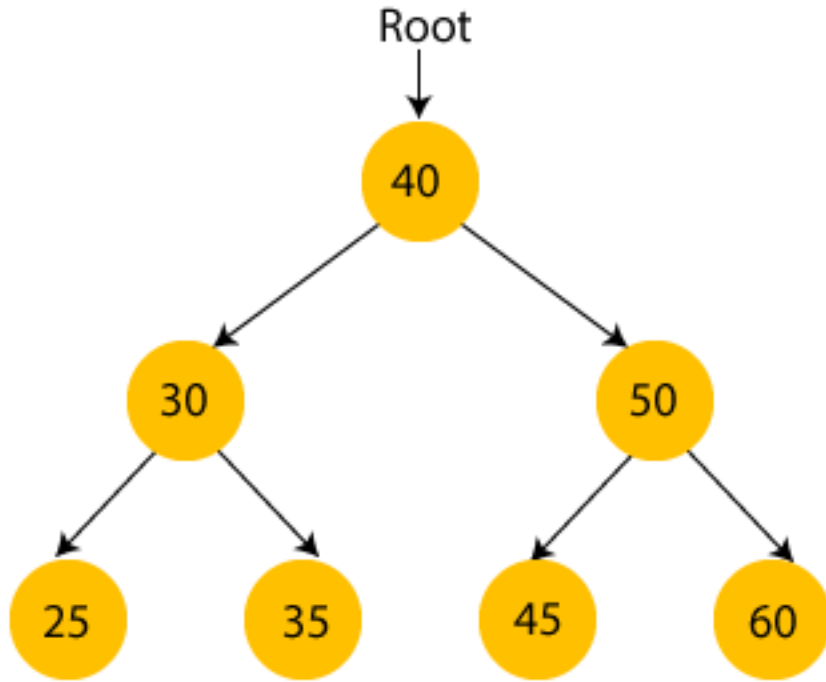
    traversePostorder(root);

    return 0;
}

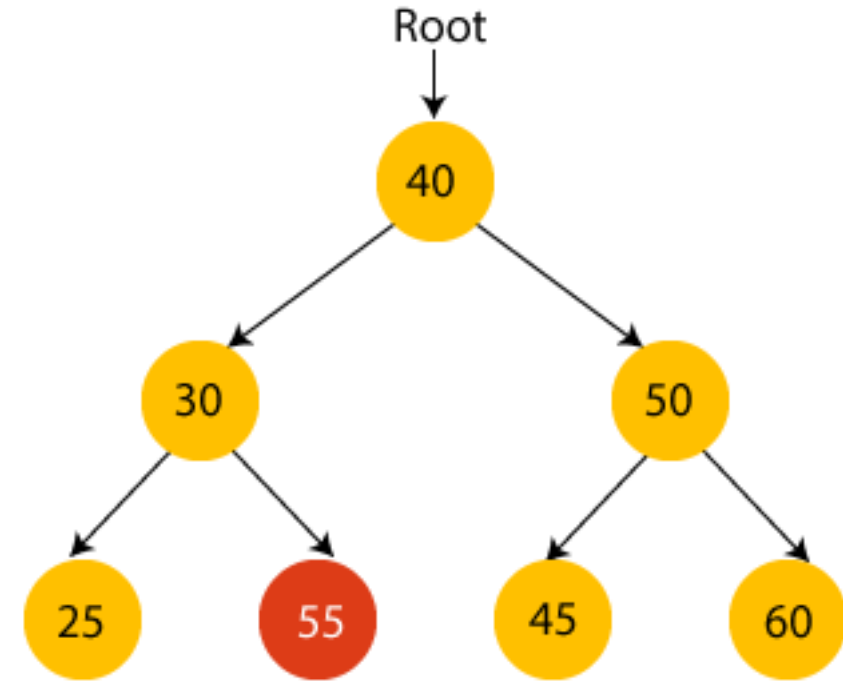
```

# Binary Search tree

- Binary search tree is a **non-linear data structure in which one node is connected to  $n$  number of nodes.**
- It is a node-based data structure.
- A node can be represented in a binary search tree with **three fields, i.e., data part, left-child, and right-child.**
- A node can be connected to the **utmost two child nodes in a binary search tree**, so the node contains two pointers (left child and right child pointer).
- Every node in the **left subtree** must contain a **value less than the value of the root** node, and the value of each node in the **right subtree** must be **bigger than the value of the root node.**



In the above figure, we 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.



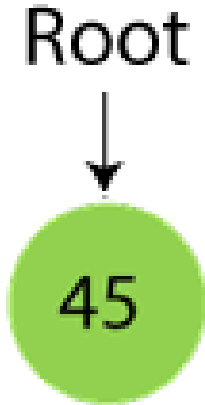
In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

# Example of creating a binary search tree

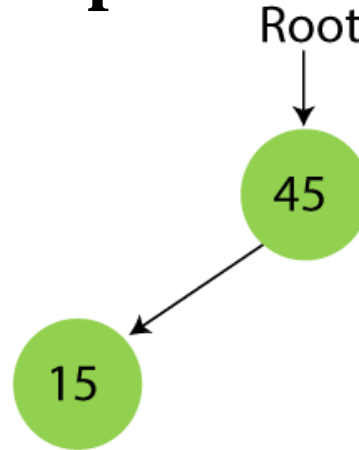
- Now, let's see the creation of binary search tree using an example.
  - **Suppose the data elements are :**
    - 45, 15, 79, 90, 10, 55, 12, 20, 50
1. First, we have to insert 45 into the tree as the root of the tree.
  2. Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
  3. Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

• 45, 15, 79, 90, 10, 55, 12, 20, 50

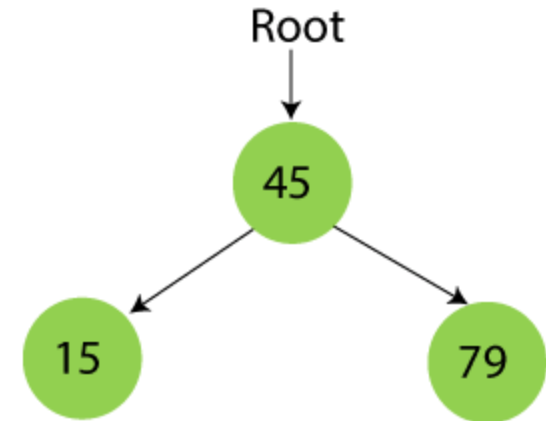
• **Step 1 - Insert 45.**



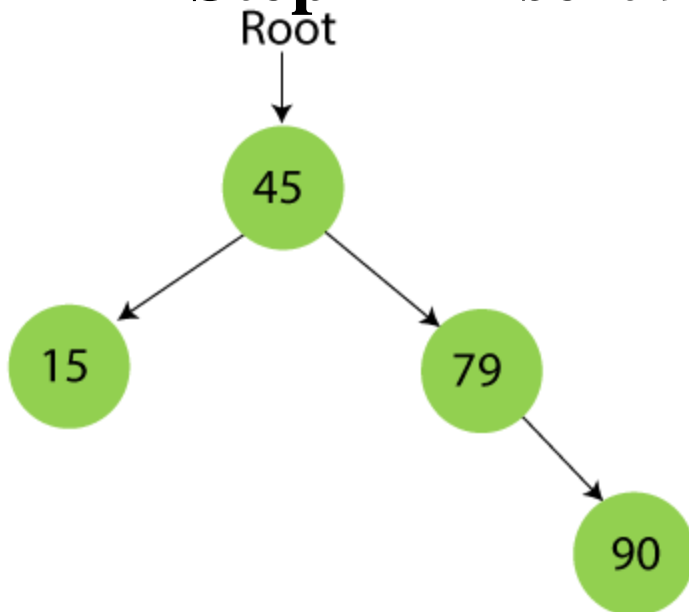
**Step 2 - Insert 15.**



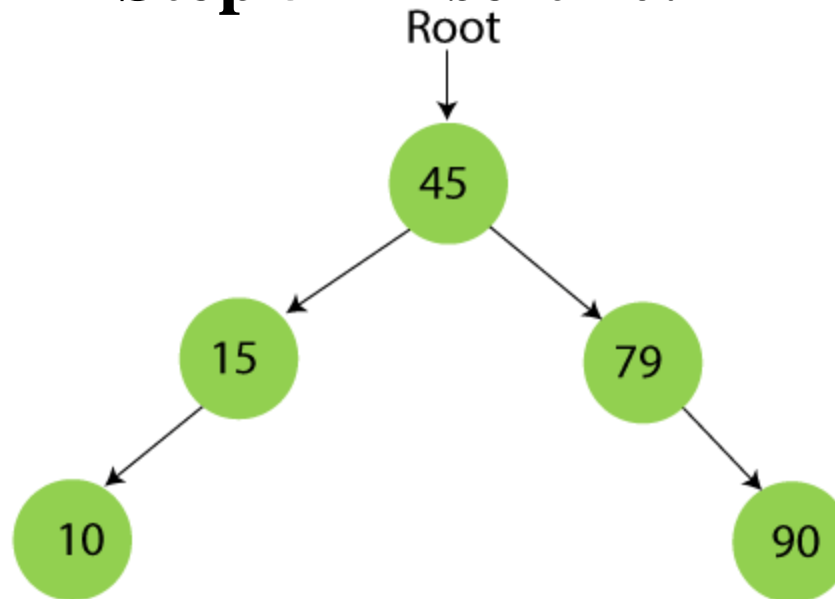
**Step 3 - Insert 79.**



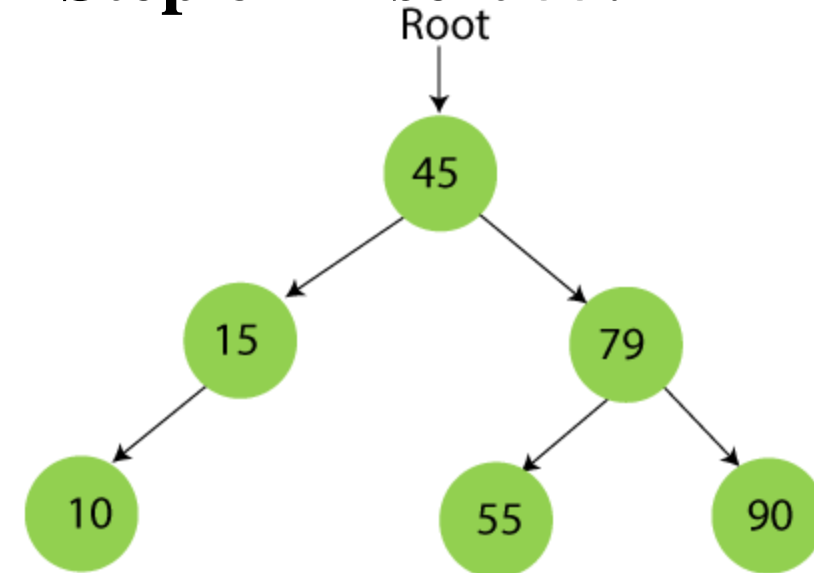
• **Step 4 - Insert 90.**



**Step 5 - Insert 10.**



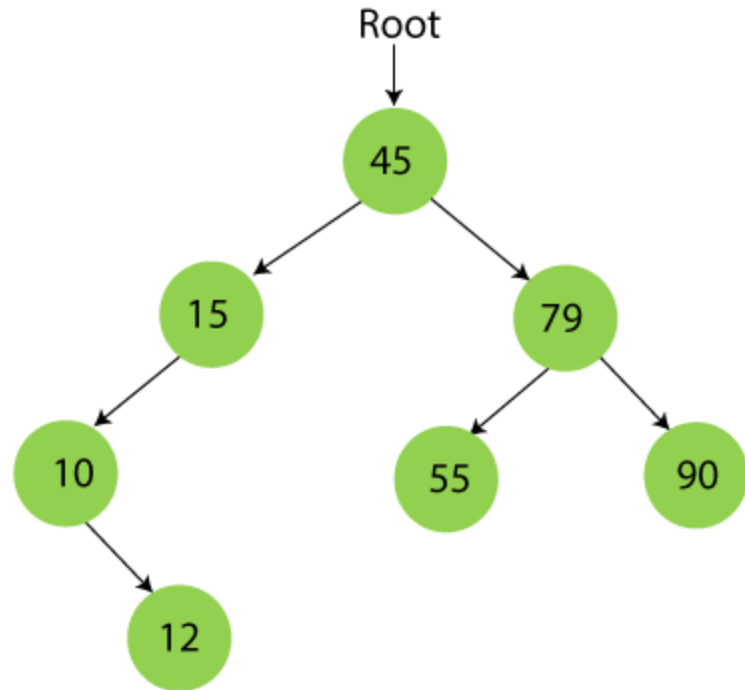
**Step 6 - Insert 55.**



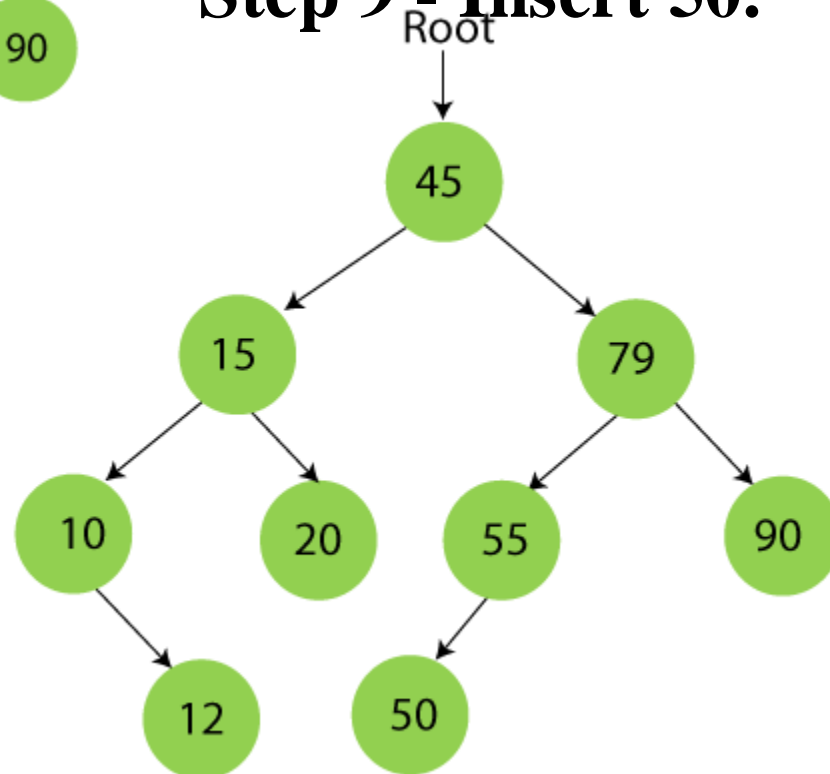


• 45, 15, 79, 90, 10, 55, 12, 20, 50

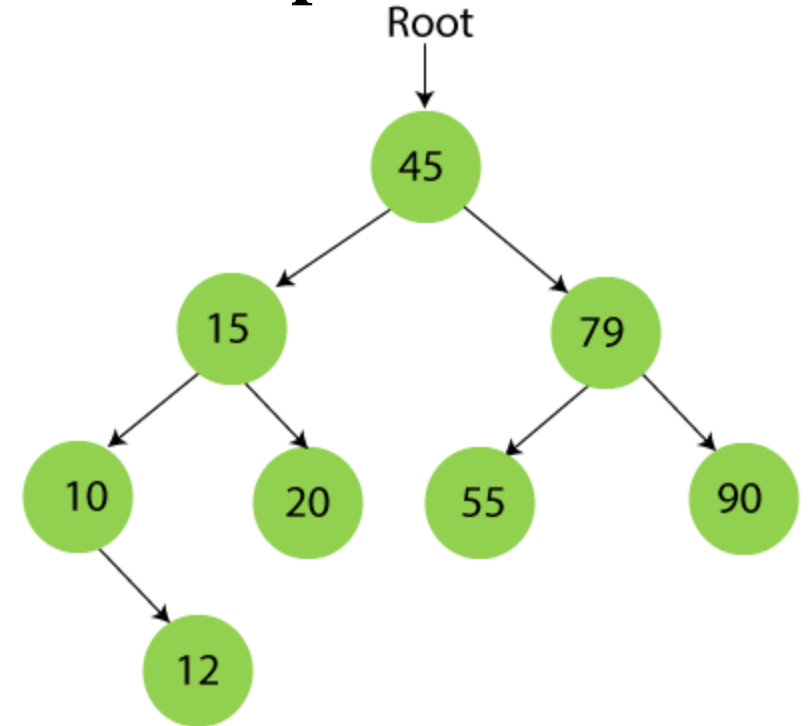
• **Step 7 - Insert 12.**



• **Step 9 - Insert 50.**



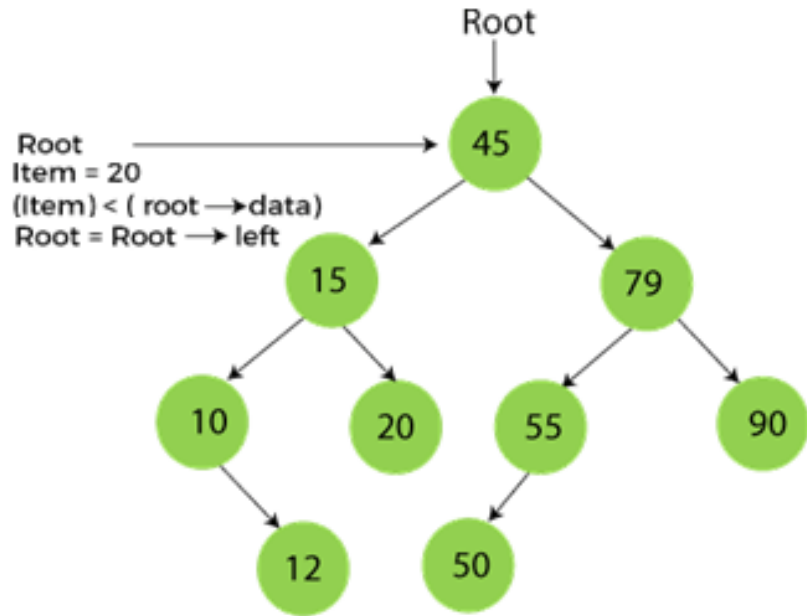
**Step 8 - Insert 20.**



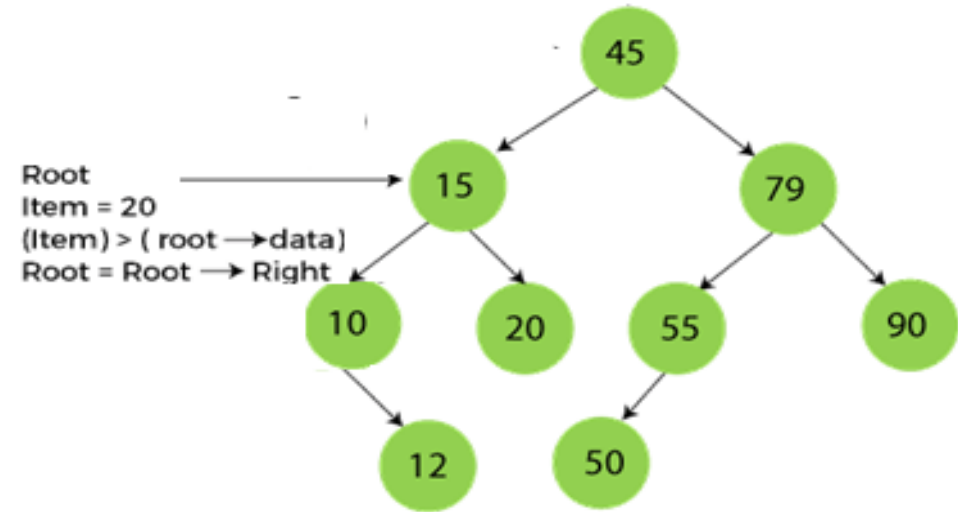
# Searching in Binary search tree

- The steps of searching a node in Binary Search tree are listed as follows -
  - First, compare the element to be searched with the root element of the tree.
  - If root is matched with the target element, then return the node's location.
  - 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.
  - If it is larger than the root element, then move to the right subtree.
  - Repeat the above procedure recursively until the match is found.
  - If the element is not found or not present in the tree, then return NULL.

## • Step1:



## Step2:



## Step3:



# Algorithm to search an element in Binary search tree

Search (root, item)

Step 1 - if (item = root  $\rightarrow$  data) or (root = NULL)

return root

else if (item < **root**  $\rightarrow$  data)

return Search(root  $\rightarrow$  left, item)

else

return Search(root  $\rightarrow$  right, item)

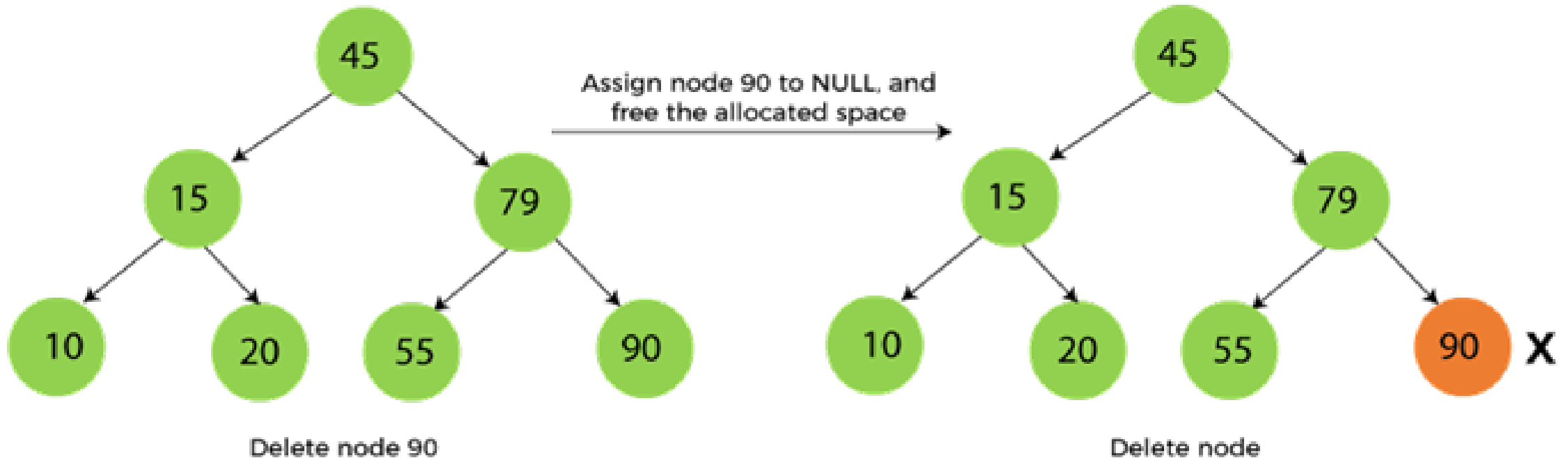
END if

Step 2 - END

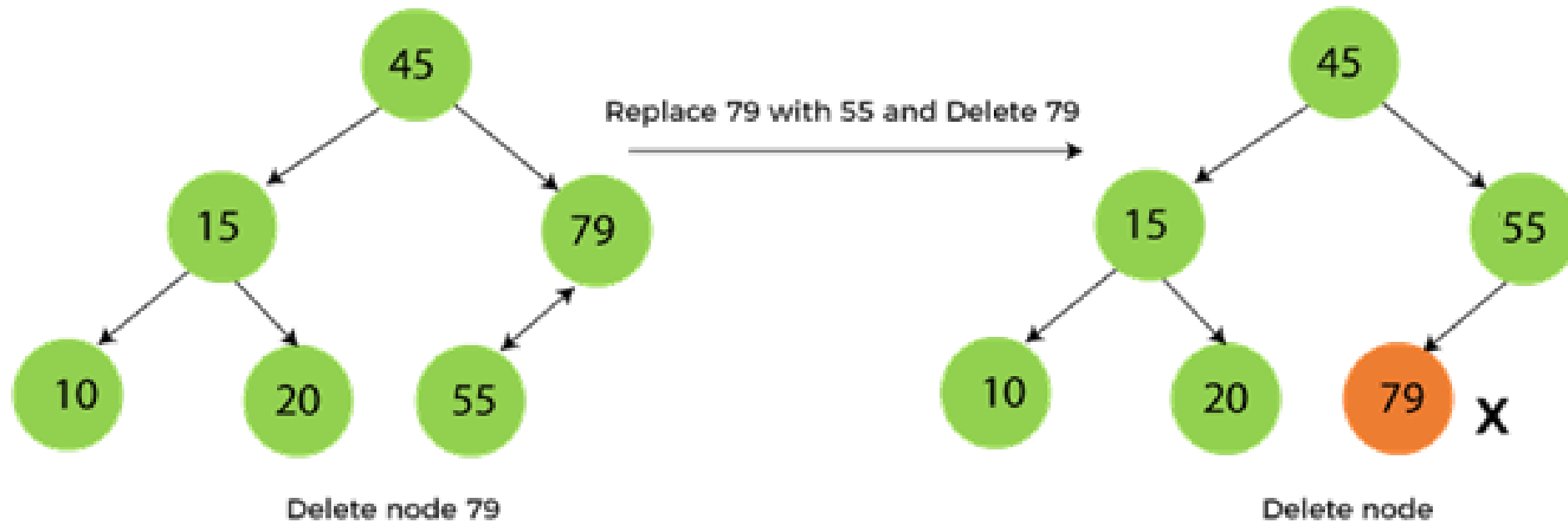
# Deletion in Binary Search tree

- In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated.
- To delete a node from BST, there are three possible situations occur -
  - The node to be deleted is the leaf node, or,
  - The node to be deleted has only one child, and,
  - The node to be deleted has two children

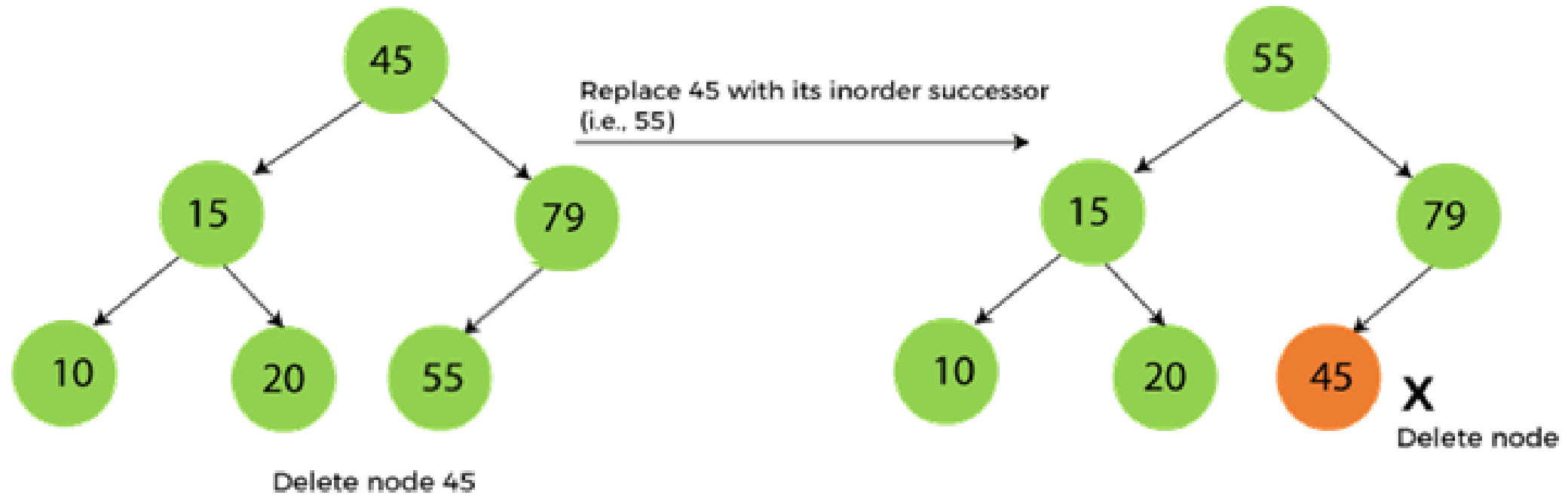
# When the node to be deleted is the leaf node



# When the node to be deleted has only one child

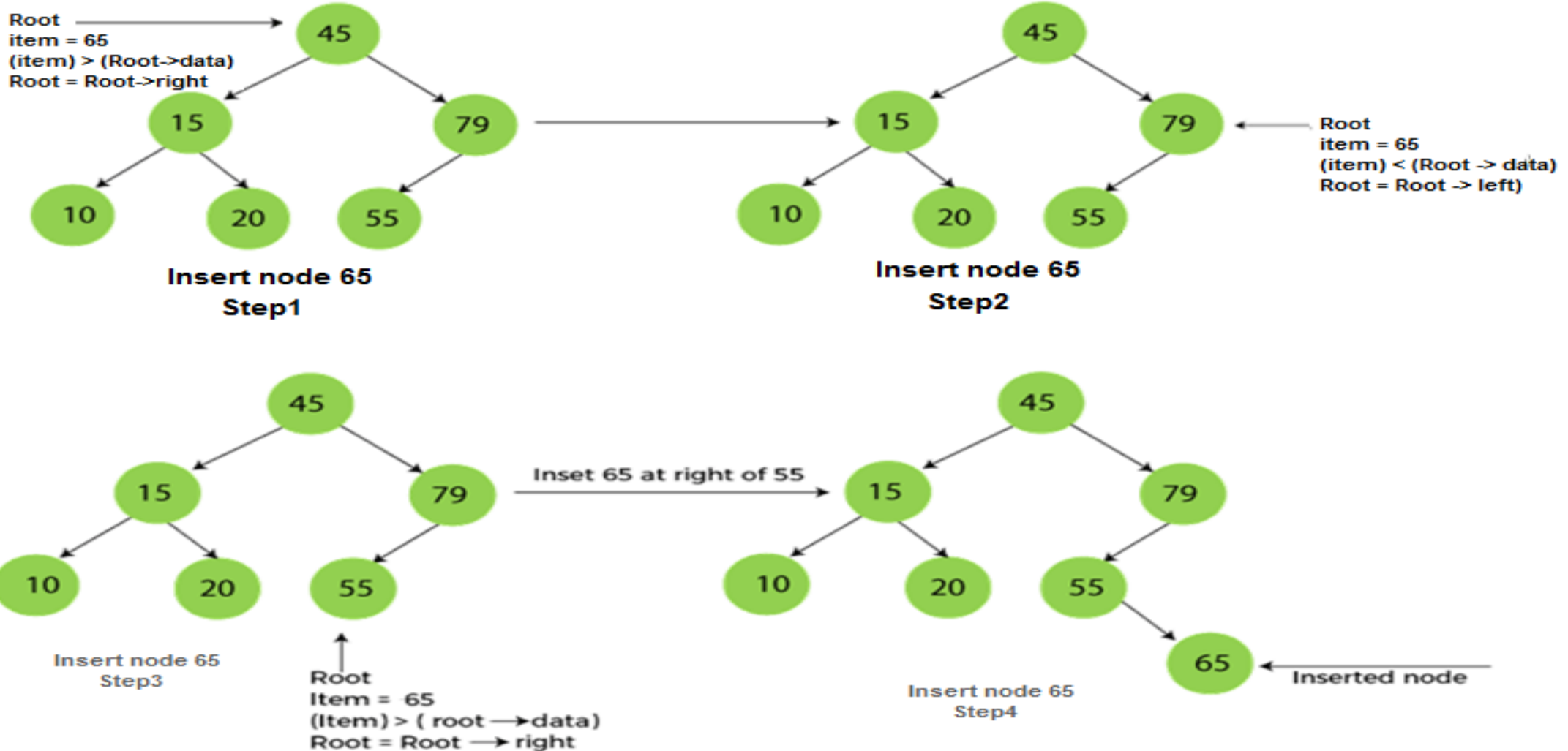


# When the node to be deleted has two children



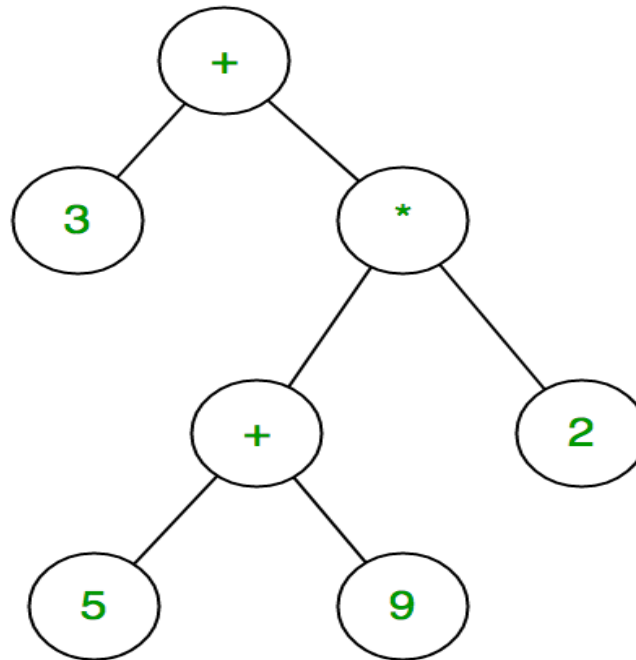


# Insertion in Binary Search tree



# Expression Tree

- The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for  $3 + ((5+9)*2)$  would be:



# Construction of Expression Tree:

- To construct an Expression Tree for the given expression, we generally use Stack Data Structure.
- First of all, we will do scanning of the given expression into left to the right manner, then one by one check the identified character,
  - If a scanned character is an operand, we will apply the push operation and push it into the stack.
  - If a scanned character is an operator, we will apply the pop operation into it to remove the two values from the stack to make them its child, and after then we will push back the current parent node into the stack.
- <https://youtu.be/NGEKaXpGAT8>

# Huffman Coding

- Huffman Coding is a technique of compressing data to reduce its size without losing any of the details.
- Huffman Coding is generally useful to compress the data in which there are frequently occurring characters.

# How Huffman Coding works?

- Suppose the string below is to be sent over a network.

B	C	A	A	D	D	D	C	C	A	C	A	C	A	C
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Initial string

- Each character occupies 8 bits. There are a total of 15 characters in the above string. Thus, a total of  $8 * 15 = 120$  bits are required to send this string.
- Using the Huffman Coding technique, we can compress the string to a smaller size.
- **Huffman coding first creates a tree using the frequencies of the character and then generates code for each character.**
- Once the data is encoded, it has to be decoded. Decoding is done using the same tree.

# Huffman coding is done with the help of the following

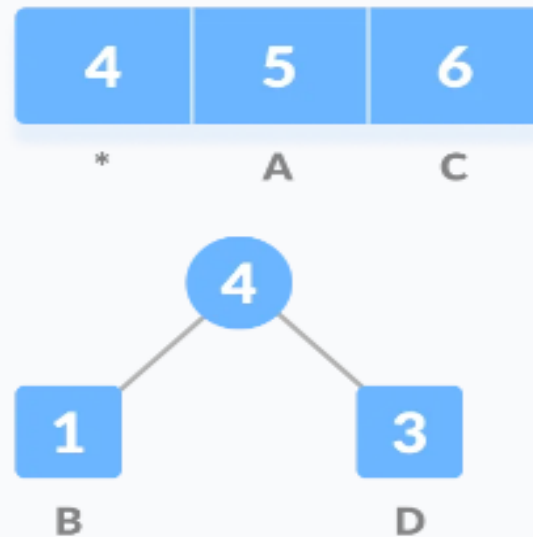
1. Calculate the frequency of each character in the string.

1	6	5	3
B	C	A	D
Frequency of string			

2. Sort the characters in increasing order of the frequency. These are stored in a priority queue `Q`.

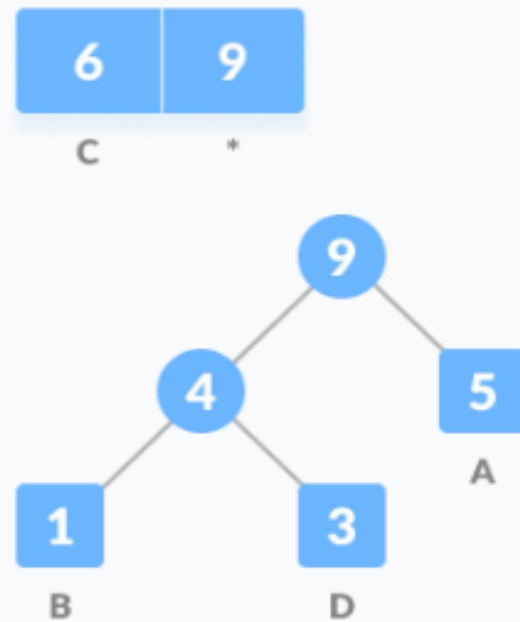
1	3	5	6
B	D	A	C
Characters sorted according to the frequency			

3. Make each unique character as a leaf node.
4. Create an empty node  $z$ . Assign the minimum frequency to the left child of  $z$  and assign the second minimum frequency to the right child of  $z$ . Set the value of the  $z$  as the sum of the above two minimum frequencies.



Getting the sum of the least numbers

5. Remove these two minimum frequencies from  $Q$  and add the sum into the list of frequencies (\* denote the internal nodes in the figure above).
6. Insert node  $z$  into the tree.
7. Repeat steps 3 to 5 for all the characters.

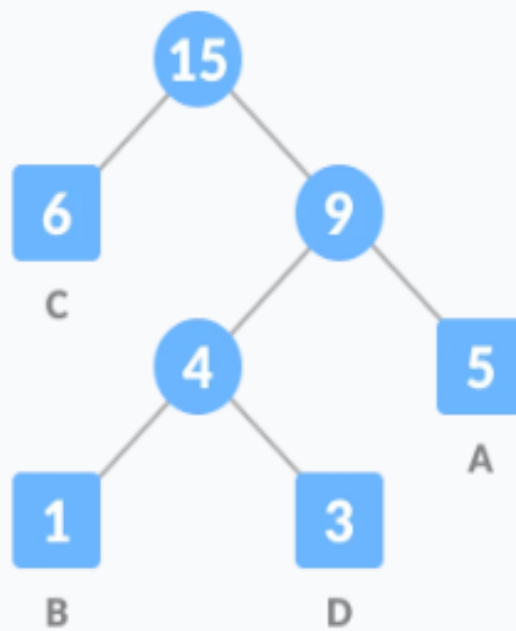


Repeat steps 3 to 5 for all the characters.



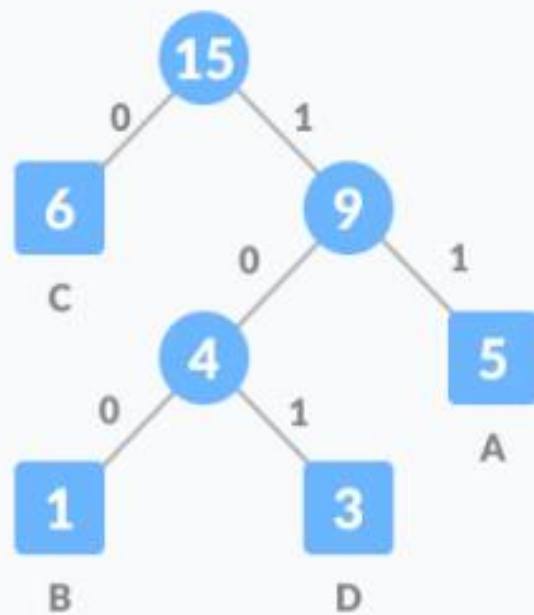
15

\*



Repeat steps 3 to 5 for all the characters.

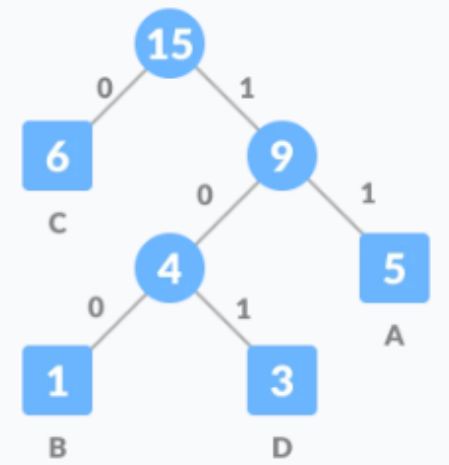
8. For each non-leaf node, assign 0 to the left edge and 1 to the right edge.



Assign 0 to the left edge and 1 to the right edge

For sending the above string over a network, we have to send the tree as well as the above compressed-code. The total size is given by the table below.

Character	Frequency	Code	Size
A	5	11	$5 \times 2 = 10$
B	1	100	$1 \times 3 = 3$
C	6	0	$6 \times 1 = 6$
D	3	101	$3 \times 3 = 9$
$4 \times 8 = 32$ bits	15 bits		28 bits

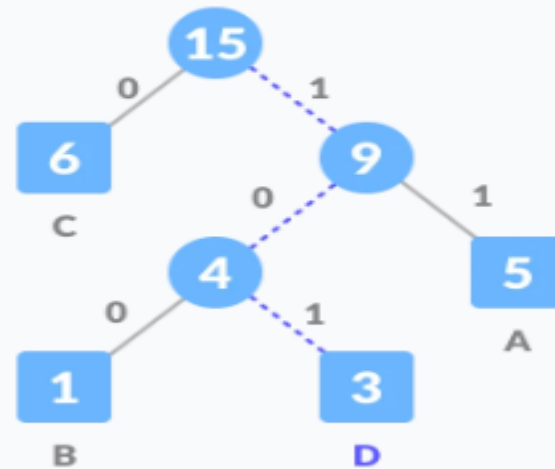


Without encoding, the total size of the string was 120 bits. After encoding the size is reduced to  $32 + 15 + 28 = 75$ .

# Decoding the code

For decoding the code, we can take the code and traverse through the tree to find the character.

Let 101 is to be decoded, we can traverse from the root as in the figure below.



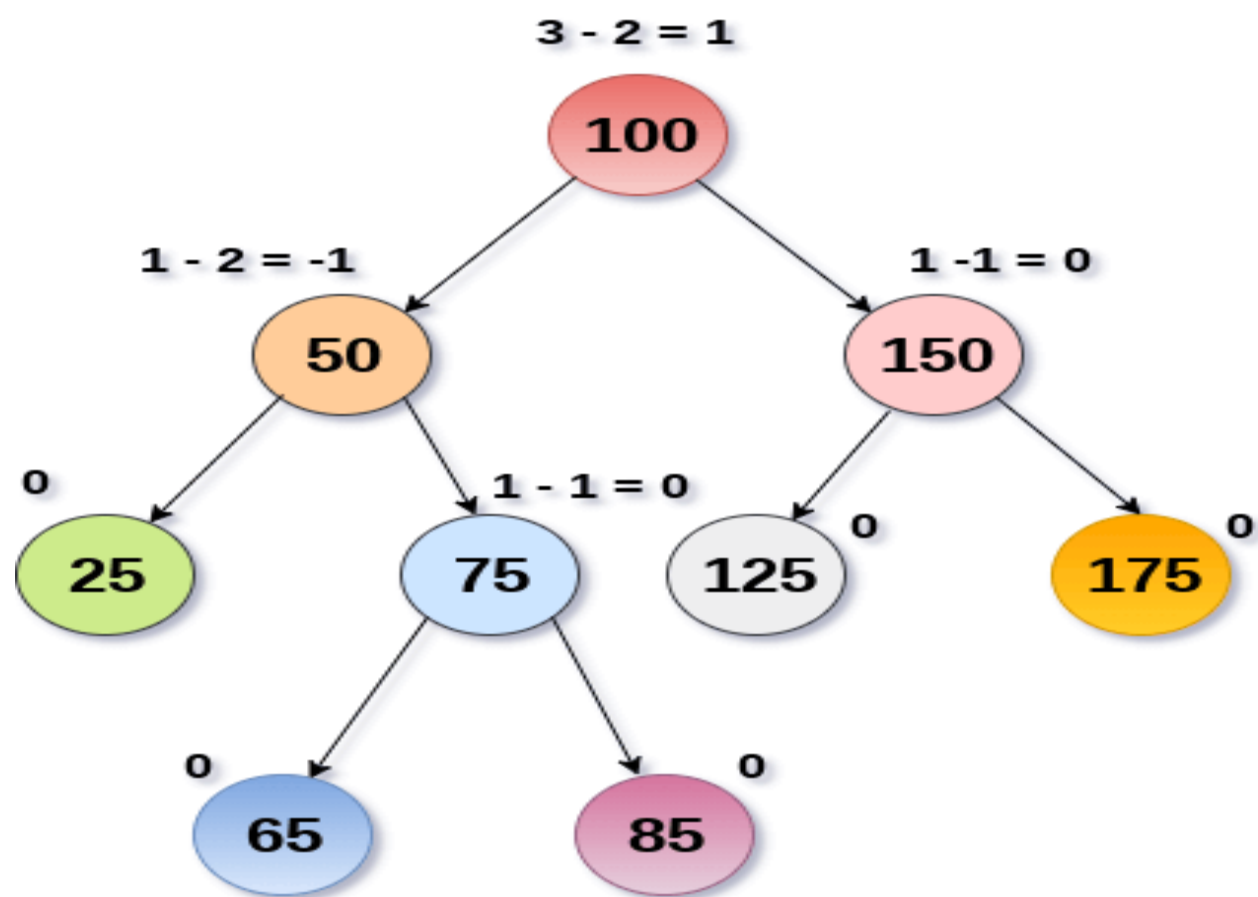
Decoding

# Huffman Coding Algorithm

- create a priority queue Q consisting of each unique character.
- sort then in ascending order of their frequencies.
- for all the unique characters:
  1. create a newNode
  2. extract minimum value from Q and assign it to leftChild of newNode
  3. extract minimum value from Q and assign it to rightChild of newNode
  4. calculate the sum of these two minimum values and assign it to the value of newNode
  5. insert this newNode into the tree
- return rootNode

# AVL Tree

- AVL 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.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.
- **Balance Factor (k) = height (left(k)) - height (right(k))**
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.



AVL Tree

# Operations on AVL tree

Sr. No	Operation	Discription
1	Insertion	Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations.
2	Deletion	Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree.

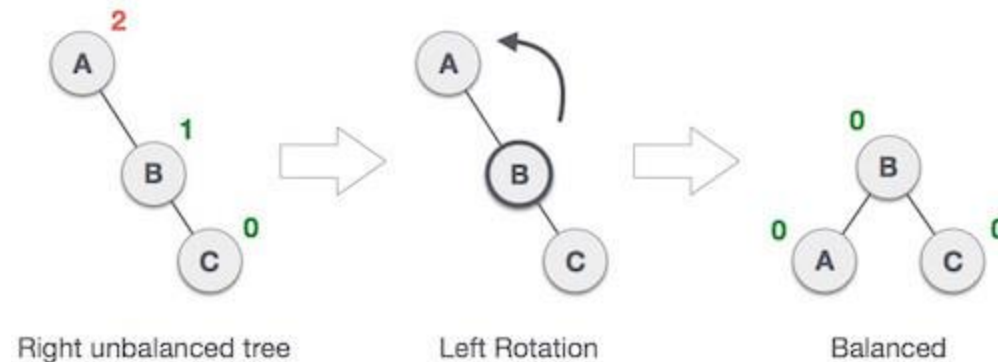


# AVL Rotations

- We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**.
- There are basically four types of rotations which are as follows:
  - L L rotation: Inserted node is in the left subtree of left subtree of A
  - R R rotation : Inserted node is in the right subtree of right subtree of A
  - L R rotation : Inserted node is in the right subtree of left subtree of A
  - R L rotation : Inserted node is in the left subtree of right subtree of A
- Where node A is the node whose balance Factor is other than -1, 0, 1.

# 1. RR Rotation

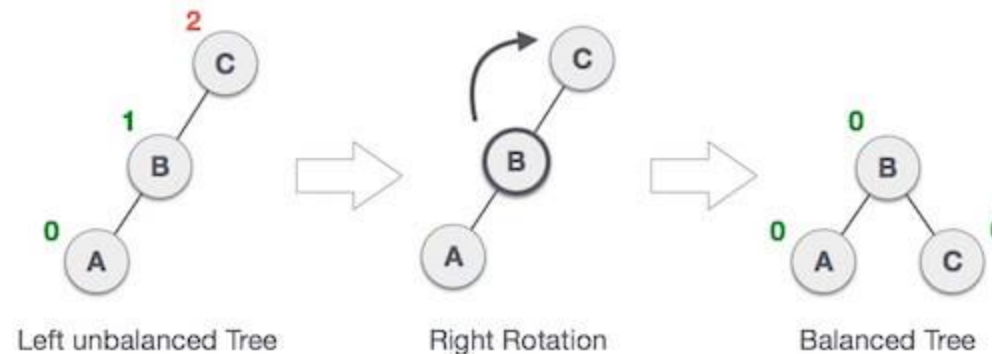
- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation,
- [RR rotation](#) is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



- In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree.
- We perform the RR rotation on the edge below A.

## 2. LL Rotation

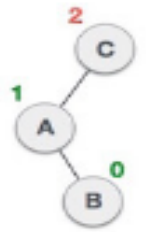
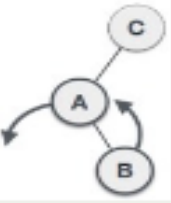
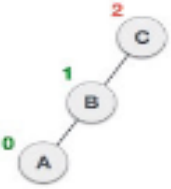
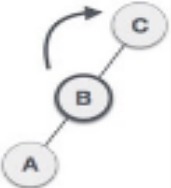
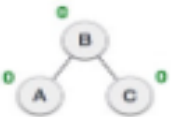
- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation,
- [LL rotation](#) is clockwise rotation, which is applied on the edge below a node having balance factor 2.



- In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree.
- We perform the LL rotation on the edge below A

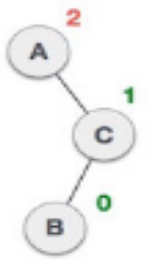
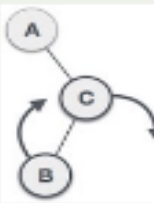
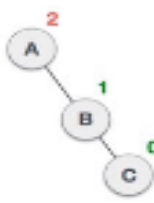
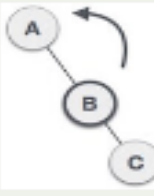
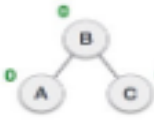
### 3. LR Rotation

- LR rotation = RR rotation + LL rotation,
- i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	<p>A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C</p>
	<p>As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.</p>
	<p>After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C</p>
	<p>Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.</p>

## 4. RL Rotation

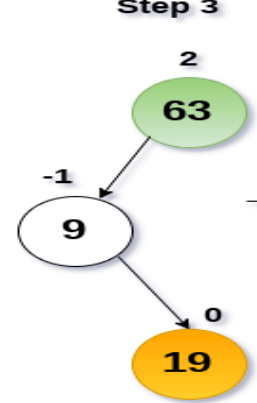
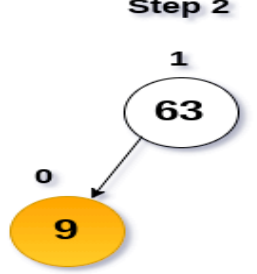
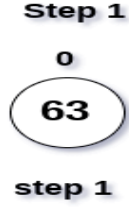
- R L rotation = LL rotation + RR rotation,
- i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

State	Action
	<p>A node <b>B</b> has been inserted into the left subtree of <b>C</b> the right subtree of <b>A</b>, because of which <b>A</b> has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of <b>A</b>.</p>
	<p>As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at <b>C</b> is performed first. By doing RR rotation, node <b>C</b> has become the right subtree of <b>B</b>.</p>
	<p>After performing LL rotation, node <b>A</b> is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node <b>A</b>.</p>
	<p>Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node <b>A</b>. node <b>C</b> has now become the right subtree of node <b>B</b>, and node <b>A</b> has become the left subtree of <b>B</b>.</p>
	<p>Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.</p>

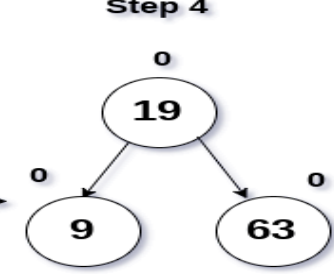
**Construct an AVL tree by inserting the following elements in the given order.**

- **63, 9, 19, 27, 18, 108, 99, 81**

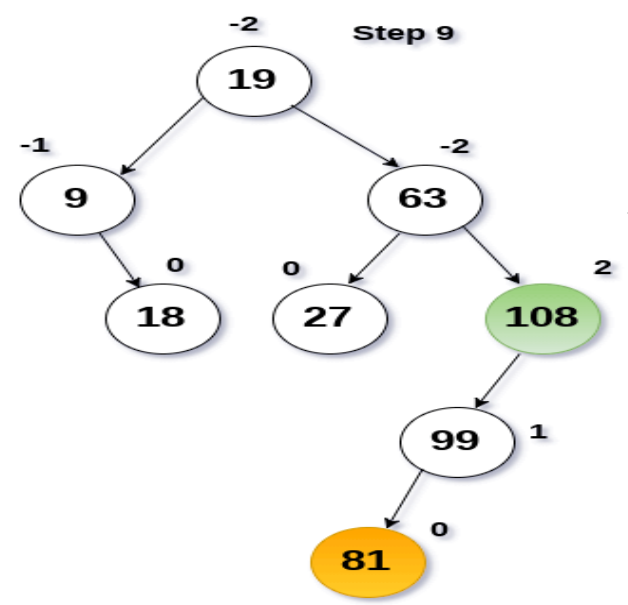
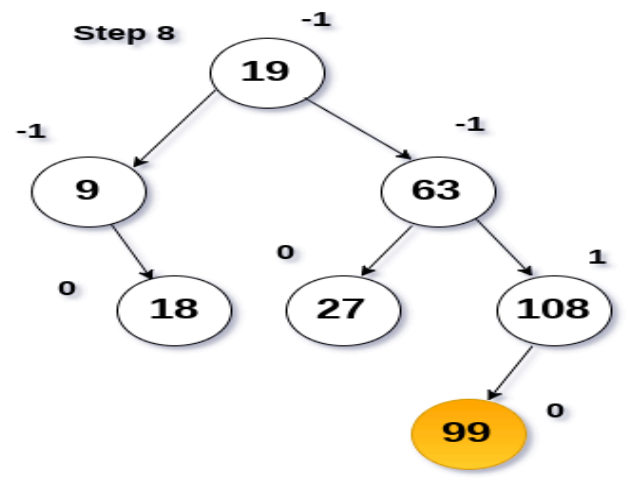
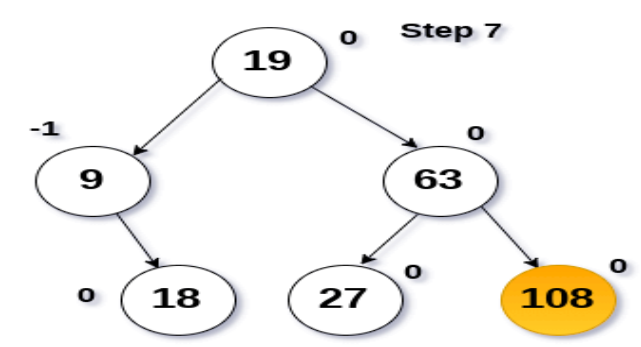
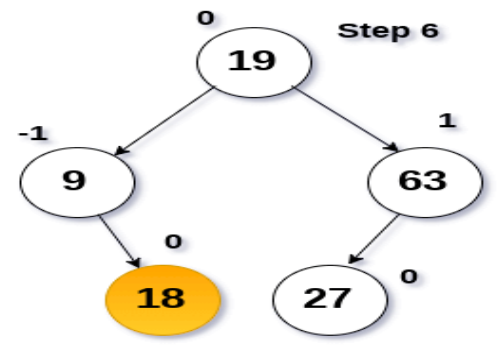
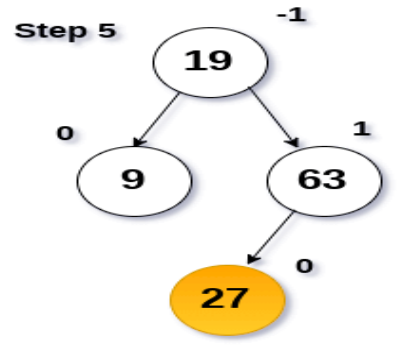




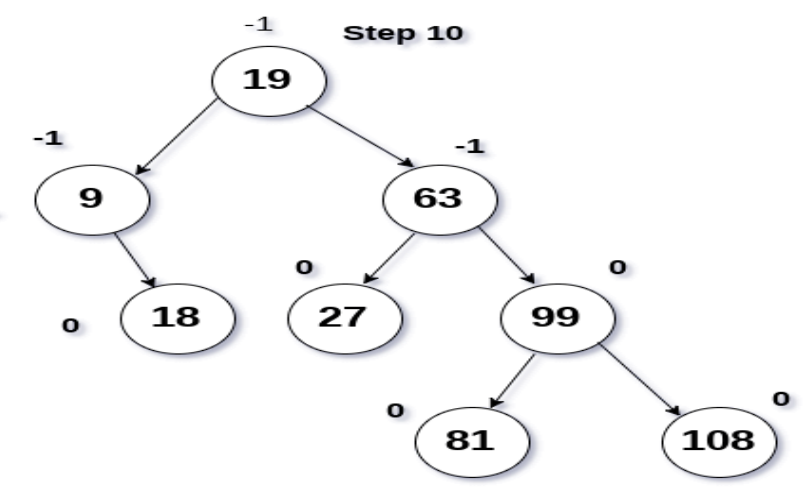
LR Rotation



63, 9, 19, 27, 18, 108, 99, 81



LL Rotation



AVL Tree