# DATA STRUCTURES AND ALGORITHMS

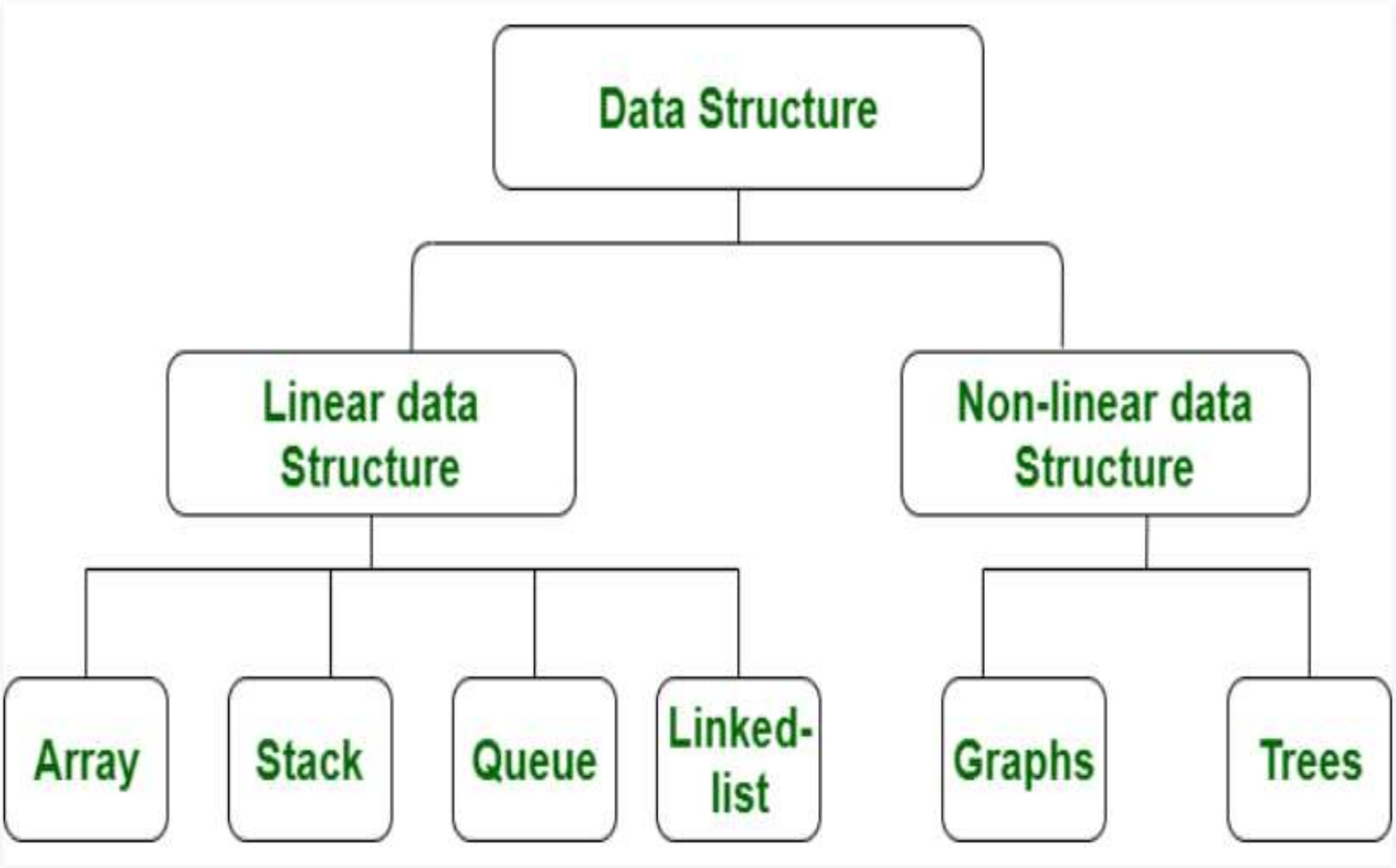
BINARYTREE, BINARY SEARCH TREE

# Linear Data Structures

- A Linear data structure have data elements arranged in sequential manner and each member element is connected to its previous and next element. This connection helps to traverse a linear data structure in a single level and in single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

# Non-linear Data Structures

- A non-linear data structure has no set sequence of connecting all its elements and each element can have multiple paths to connect to other elements. Such data structures supports multi-level storage and often cannot be traversed in single run. Such data structures are not easy to implement but are more efficient in utilizing computer memory. Examples of non-linear data structures are Tree, BST, Graphs etc


Data Structure
Linear data Structure
Non-linear data Structure
Array
Stack
Queue
Linked-list
Graphs
Trees

| Sr. No. | Key | Linear Data Structures | Non-linear Data Structures |
|---|---|---|---|
| 1 | Data Element Arrangement | In linear data structure, data elements are sequentially connected and each element is traversable through a single run. | In non-linear data structure, data elements are hierarchically connected and are present at various levels. |
| 2 | Levels | In linear data structure, all data elements are present at a single level. | In non-linear data structure, data elements are present at multiple levels. |
| 3 | Implementation complexity | Linear data structures are easier to implement. | Non-linear data structures are difficult to understand and implement as compared to linear data structures. |

| | | | |
|---|---|---|---|
| 4 | Traversal | Linear data structures can be traversed completely in a single run. | Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely. |
| 5 | Memory utilization | Linear data structures are not very memory friendly and are not utilizing memory efficiently. | Non-linear data structures uses memory very efficiently. |
| 6 | Time Complexity | Time complexity of linear data structure often increases with increase in size. | Time complexity of non-linear data structure often remain with increase in size. |
| 7 | Examples | Array, List, Queue, Stack. | Graph, Map, Tree. |

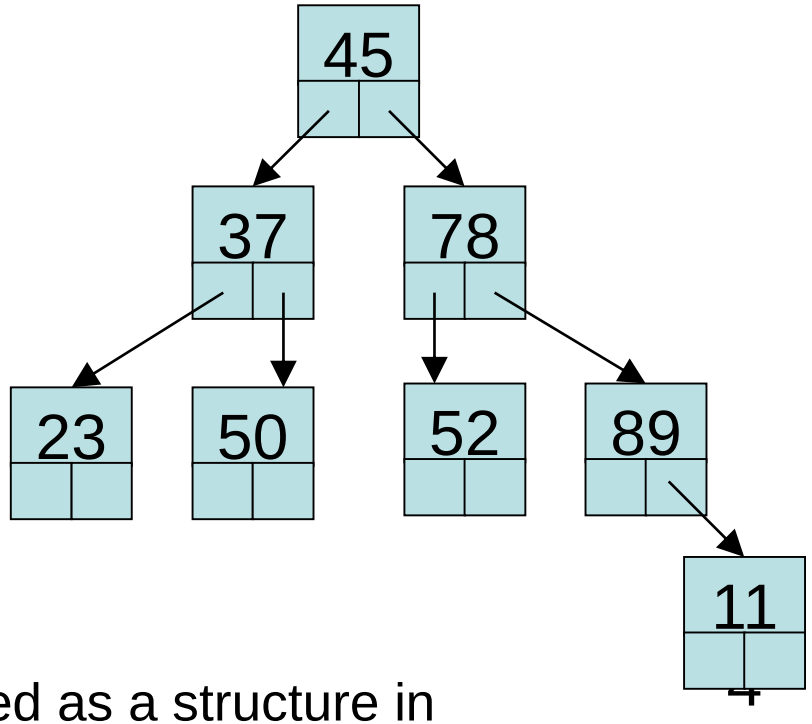
# Tree

- A tree can be defined as a finite set of data items (nodes) in which data items are arranged in a hierarchical order.
  - Trees represent the hierarchical relationship between various elements
  - Tree consists of nodes connected by an edge, the node represented by a circle and edge lives connecting to the circle.
-

# Tree Structures

.

The tree structure is naturally suited to many applications (e.g. databases, file systems, web sites, etc.), and can often allow algorithms to be significantly more efficient.

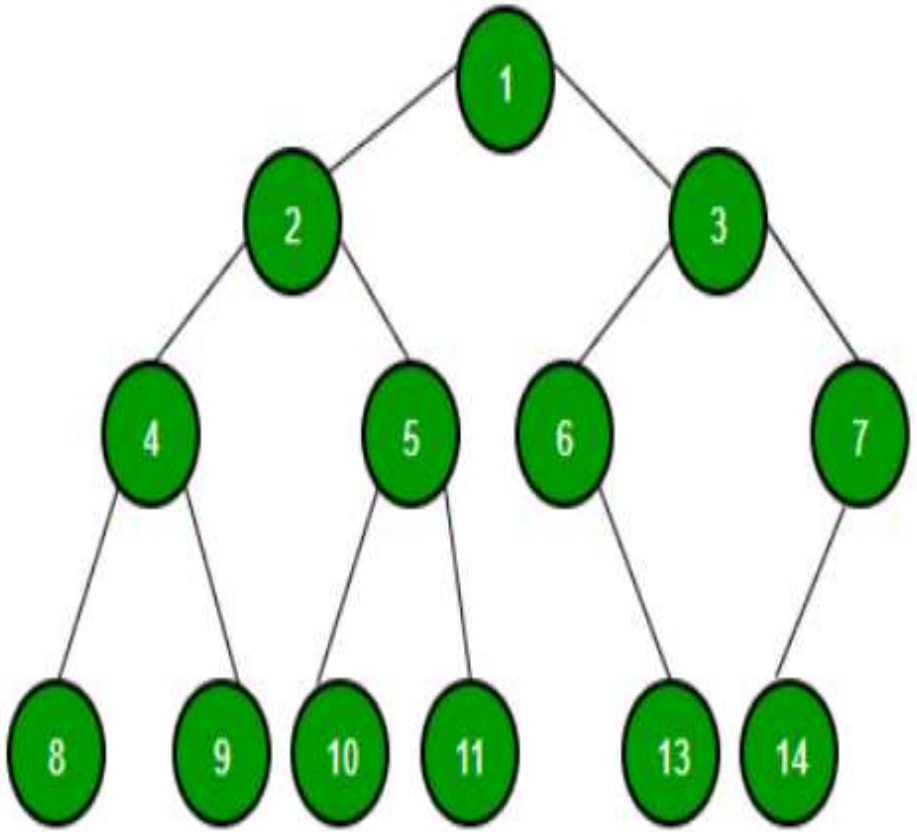



A tree (in data structure terms) is defined as a structure in which a set of *nodes* are connected together by their *edges*, in a parent-child relationship.

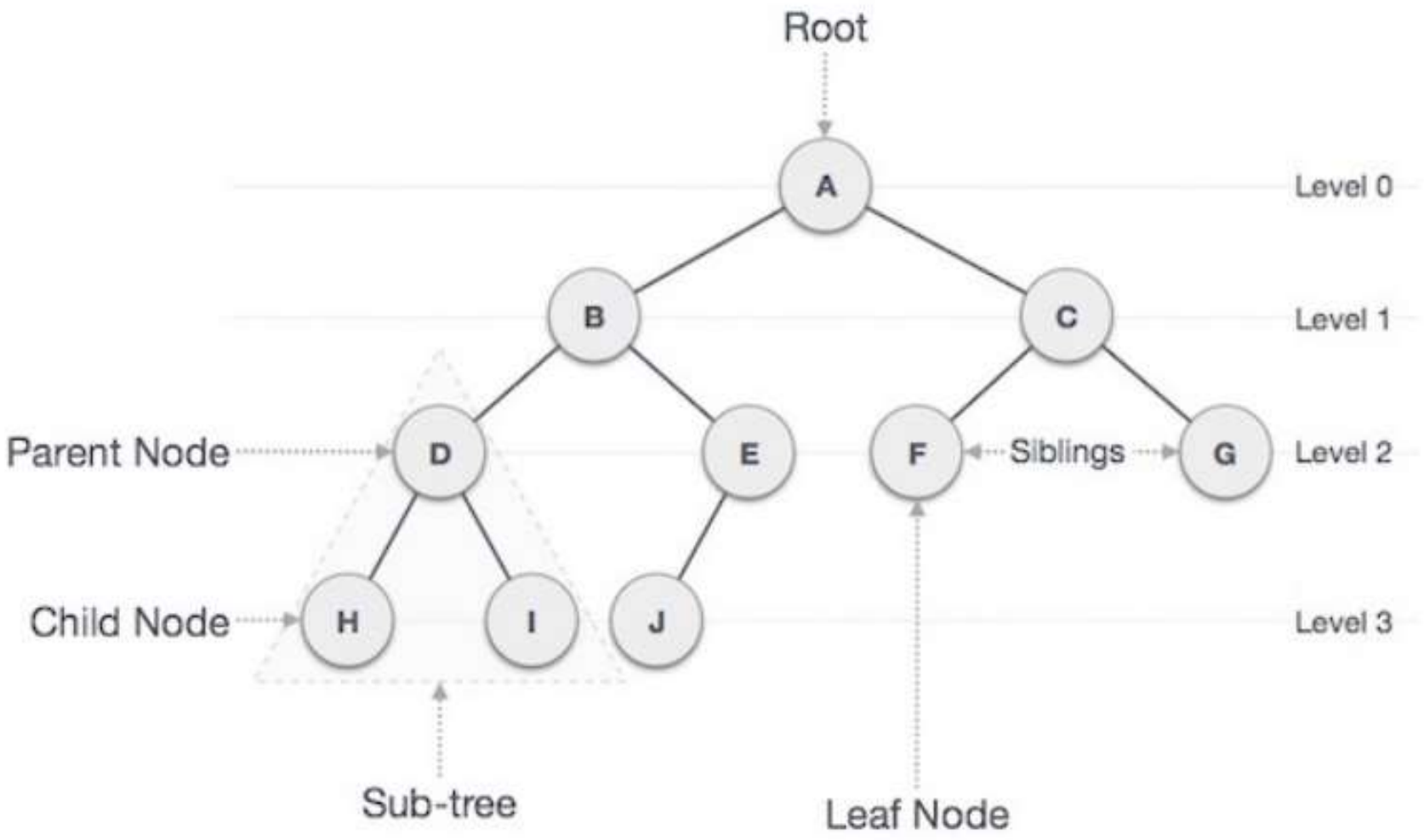
# Representing a tree

A Binary Tree node contains following parts.

- Data
- Pointer to left child
- Pointer to right child



# Binary Tree Terminoloy

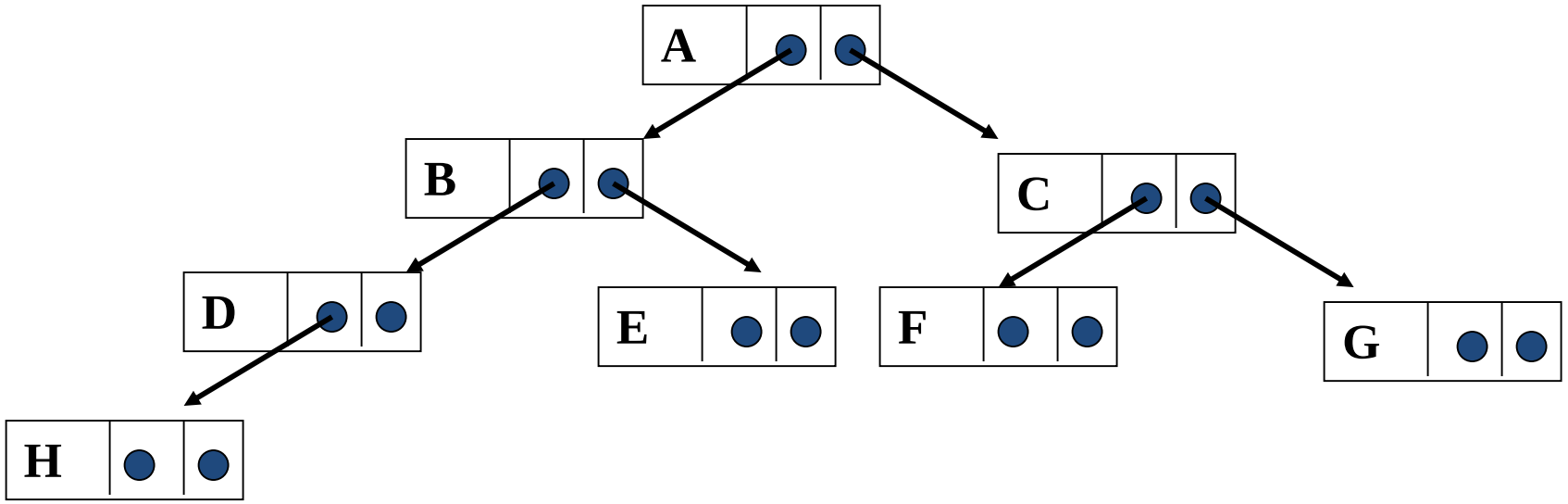


# Terminology

- The unique predecessor of a node is called its parent,
- A node's successors are its children
- The first not in a tree is known as the root. It signifies the beginning of a file. A root node has no parent.
- Children of the same parent are called siblings,

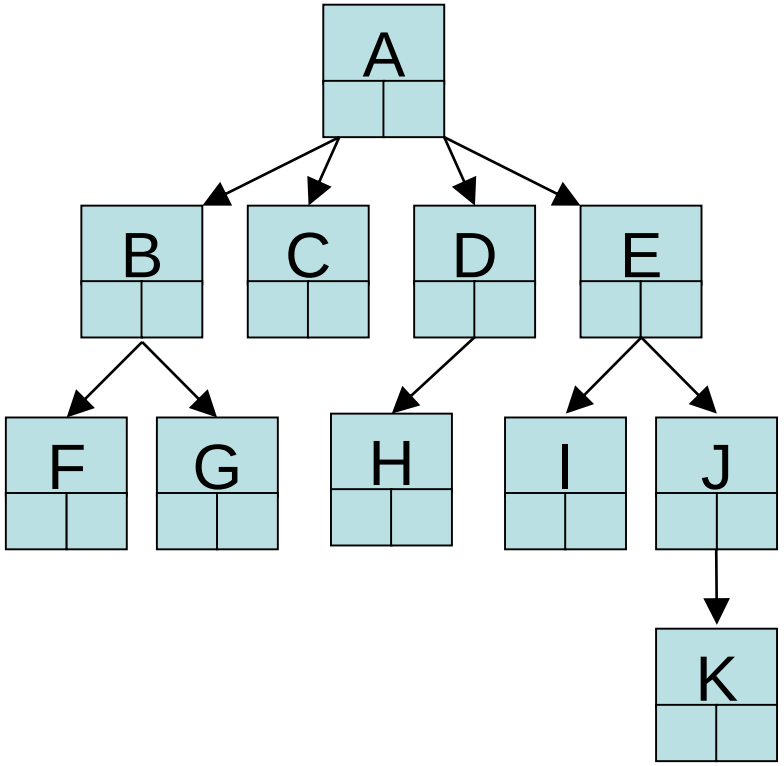
- The lines connecting the nodes are called branches
- A sequence of branches from one node to another node lower down the tree is called a path
- e.g. AB, BE, EJ
- The last node along any given path is known as a leaf
- A leaf has no children.

# Terminology




- Degree of a node: number of brances/subtrees of a node(0,1,2)
  - Nodes of degree 0 are leaf nodes
- Size of a tree: the number of nodes in the whole tree
- Depth of node: number of links leading to a node from the root node

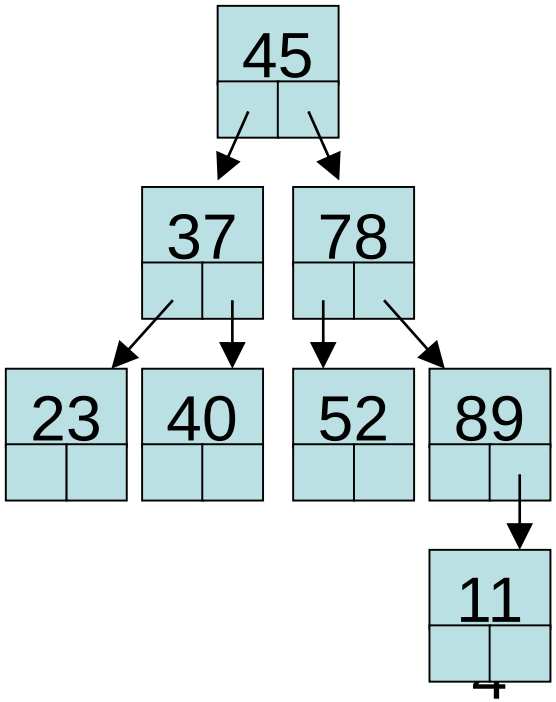
- the **depth** of a node is the length of the path from the root to the node. (The root has a depth of 0.)
- the **height** of a node is the length of the path from the node to the deepest leaf.
- the **size** of a node is the number of nodes in the subtree rooted at that node (including itself.)



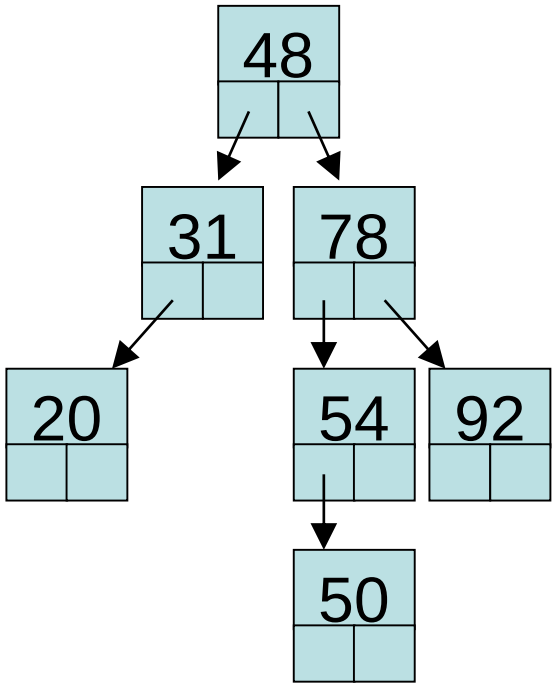

| | A | B | C | D | E | F | G | H | I | J | K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Depth** | 0 | 1 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 3 |
| **Height** | 3 | 1 | 0 | 1 | 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| **Size** | 11 | 3 | 1 | 2 | 4 | 1 | 1 | 1 | 1 | 2 | 1 |

A binary tree is balanced if and only if, for every node, the height of its left and right subtree differ by at most 1.

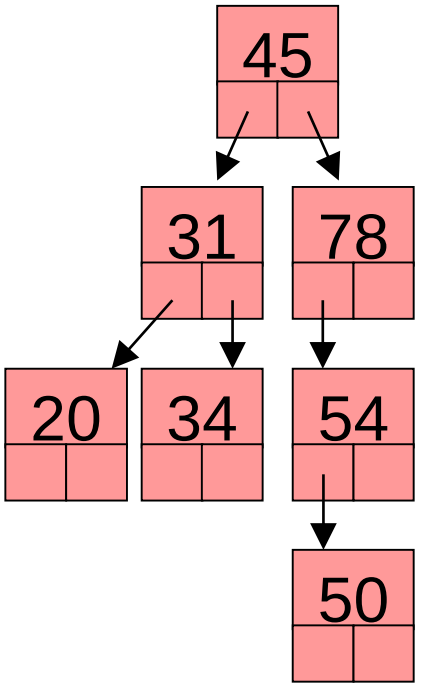
A balanced binary tree is also known as an AVL tree, after its inventors Adelson, Velskii, and Landis.




**balanced tree**



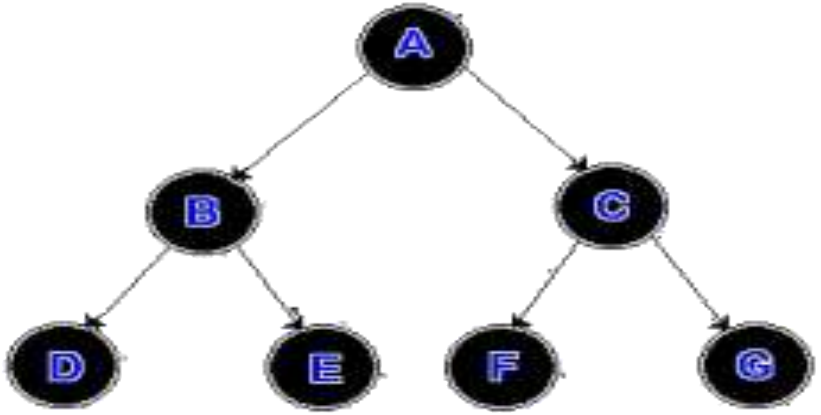

**balanced tree**

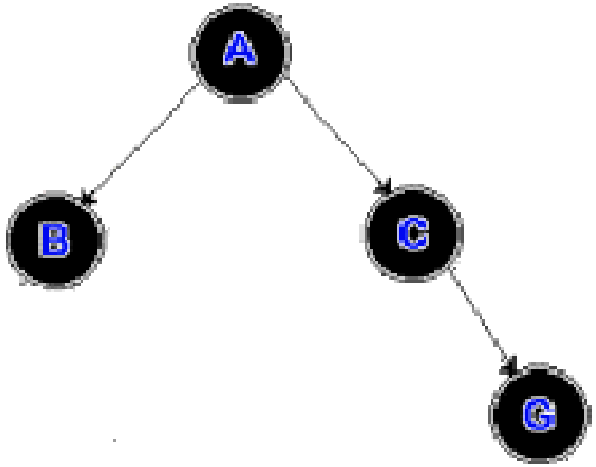



***unbalanced tree***

# Balanced binary trees

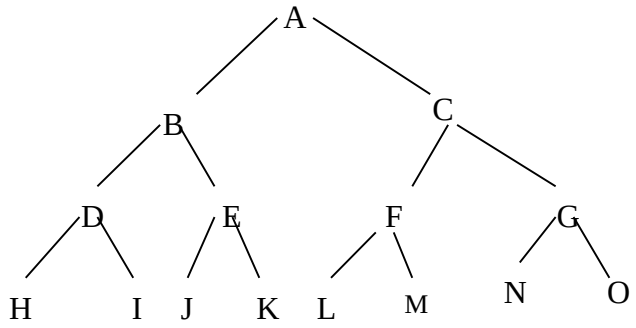
- An empty tree is balanced.
- A non-empty tree is balanced if and only if its subtrees are balanced. Where a balanced tree is a tree in which no leaf is much farther away from the root than any other leaf.



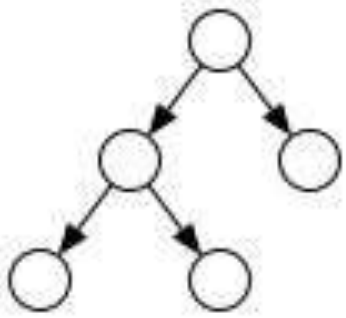



# Complete Binary tree

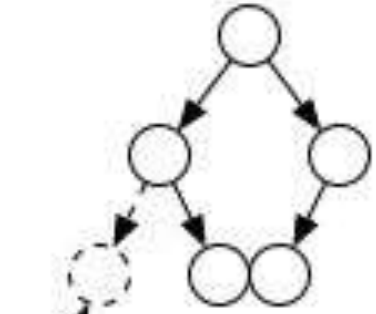
- **A complete tree** is one in which there are no gaps between leaves.
- For instance, a tree with a root node that has only one child must have its child as the left node.
- A complete tree is one that has every level filled in before adding a node to the next level, and one that has the nodes in a given level filled in from left to right, with no breaks.




*A complete binary tree of level 4.*



Complete Tree

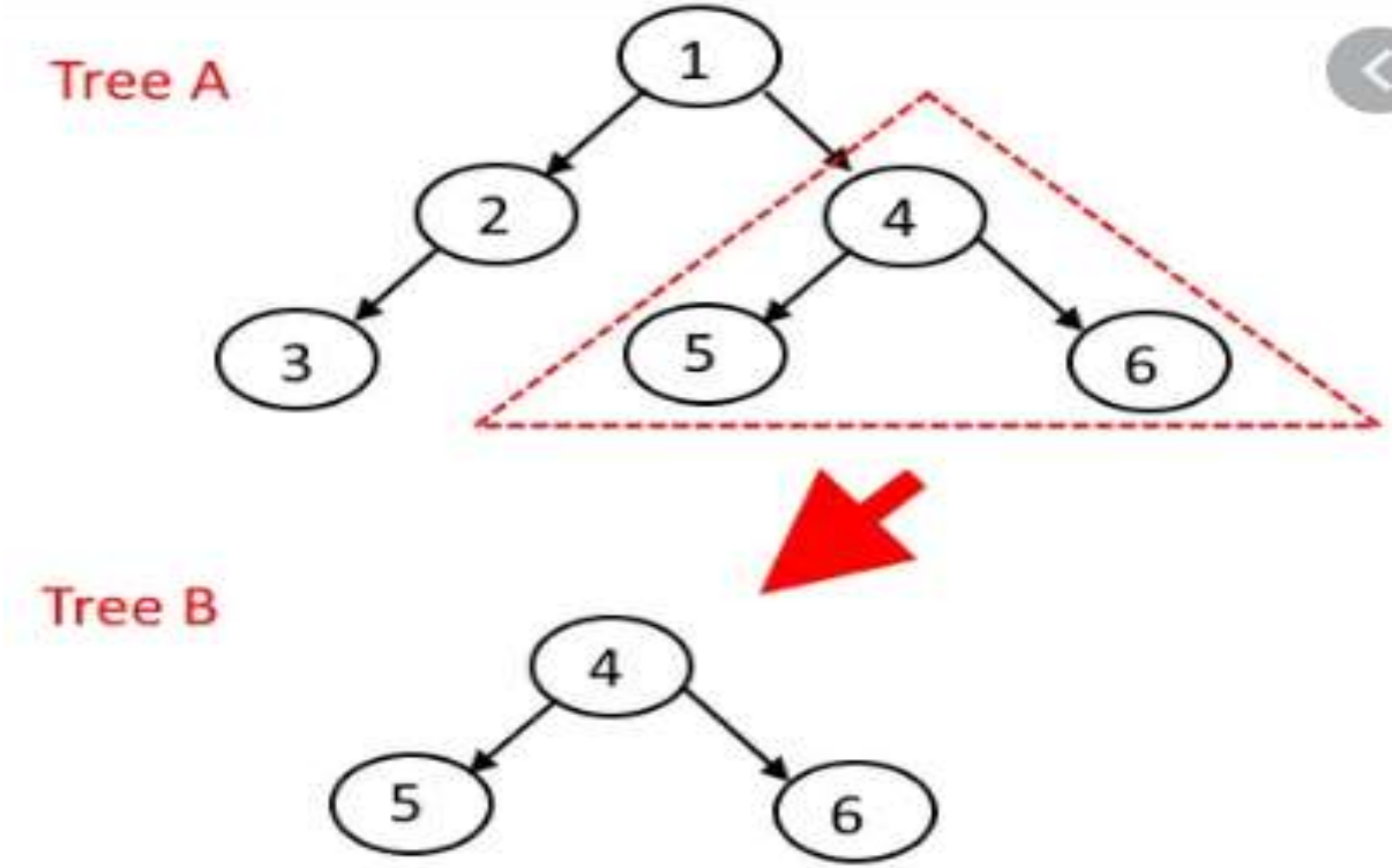


Incomplete Tree

Missing Node Here

# Sub Tree

- A subtree of a tree T is a tree S consisting of a node in T and all of its descendants in T.
  The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree.




Tree B is a subtree of Tree A

# Traversal methods

- Traversal method is a term used to denote the way data in a tree is accessed.
- Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

# Binary Tree Traversal Techniques:

A tree traversal is a method of visiting every node in the tree. By visit, we mean that some type of operation is performed. For example, you may wish to print the contents of the nodes.

There are four common ways to traverse a binary tree:d

1. Preorder
2. Inorder
3. Postorder
4. Level order

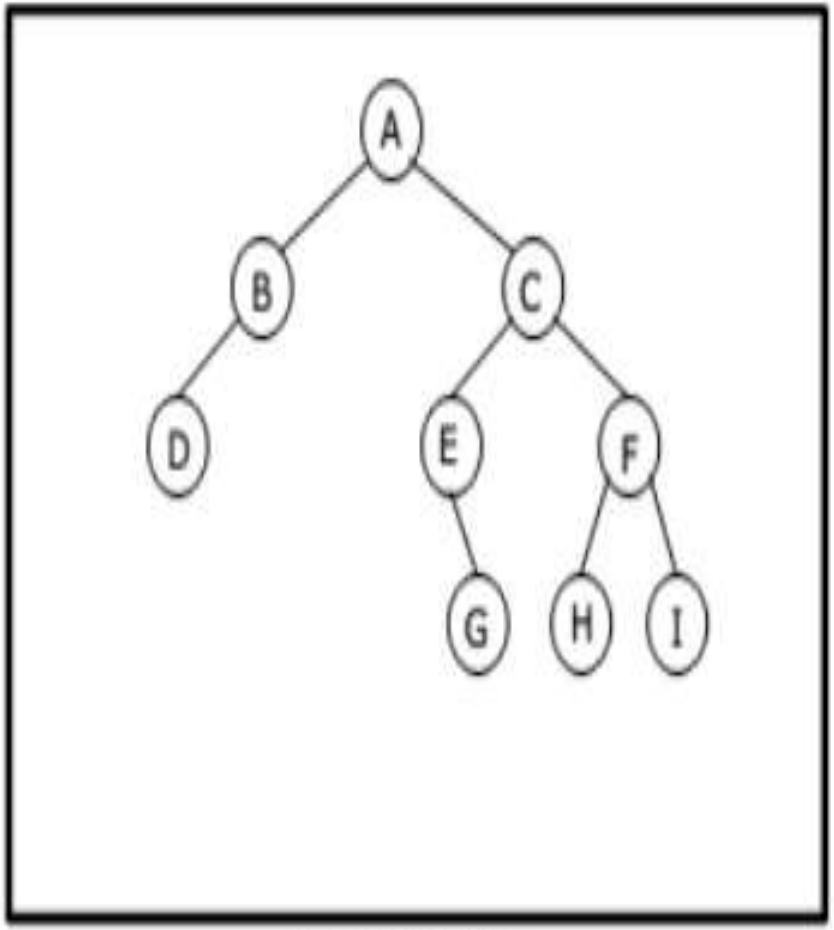
# Node Activities

- Process node (P) – perform desired operation eg. Read, write etc
- Visit left node (L) – access left child
- Visit right node (R) – access right child

The sequence of these activities distinguish the above traversal methods

# Binary trees traversal methods

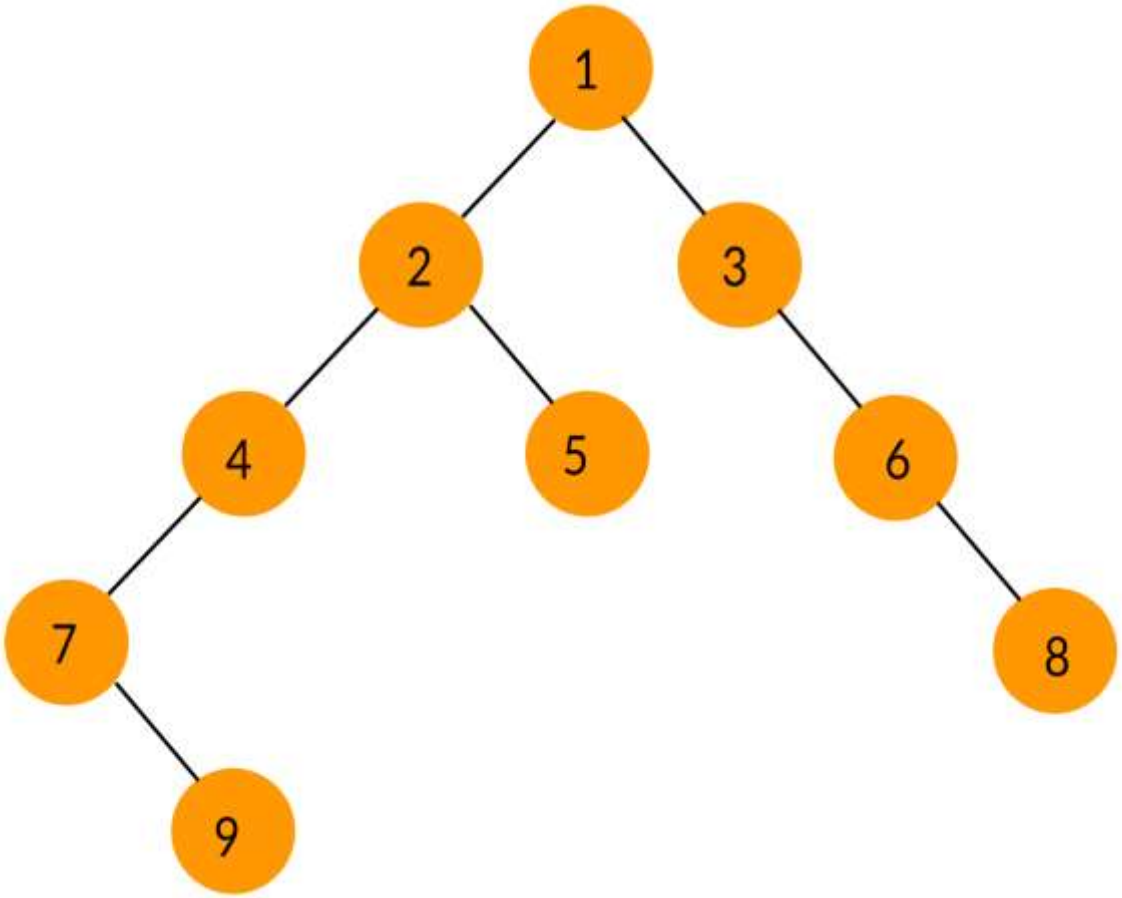
- *preorder : process* the node, then visit the left node (subtrees) , visit right node (subtrees) - PLR
- *inorder :* visit the left node (subtree), process the node, and then the right node (subtree) - LPR
- *postorder :* visit the left node (subtrees) , visit the right node (subtrees), process the node. LRP
- Level order(Breadth First). Nodes on a given level are accessed from left to right, then the next level downwards




Binary Tree

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

Pre, Post, Inorder and level order Traversing




**Inorder Traversal:** 7 9 4 2 5 1 3 6 8

**Preorder Traversal:** 1 2 4 7 9 5 3 6 8

**Postorder Traversal:** 9 7 4 5 2 8 6 3 1

Tree Traversal Algorithms can be classified broadly in the following two categories by the order in which the nodes are visited:

**Depth-First Search (DFS) Algorithm:** It starts with the root node and first visits all nodes of one branch as deep as possible of the chosen Node and before backtracking, it visits all other branches in a similar fashion. There are three sub-types under this, which we will cover in this article.

**Breadth-First Search (BFS) Algorithm:** It also starts from the root node and visits all nodes of current depth before moving to the next depth in the tree. We will cover one algorithm of BFS type in the upcoming section.

# BINARY SEARCH TREE (BST)

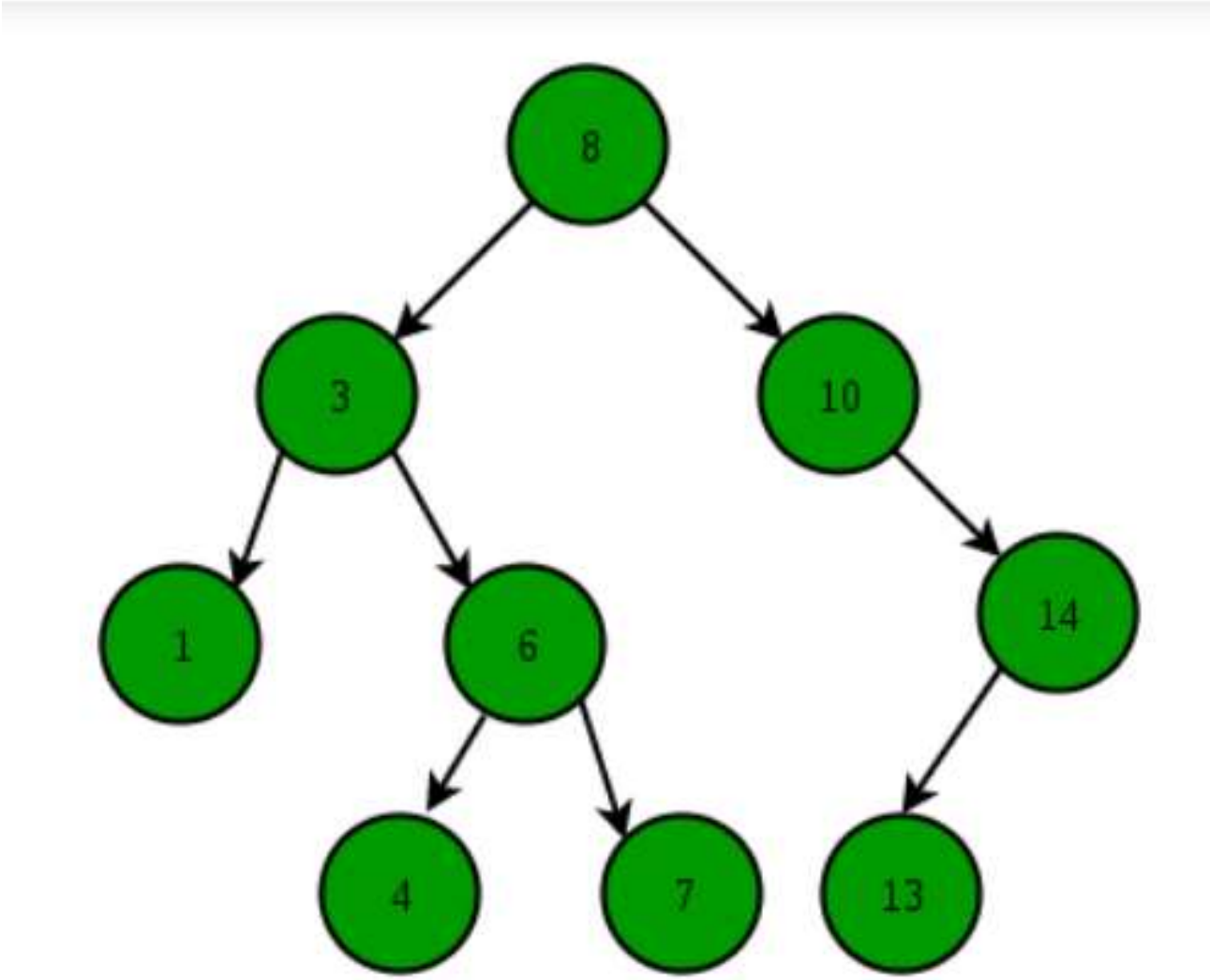
- One of the greatest challenges in non-linear data structures is searching for an element
- A BST is organized in such a way that it facilitates searching
- HOW?

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

# binary search tree

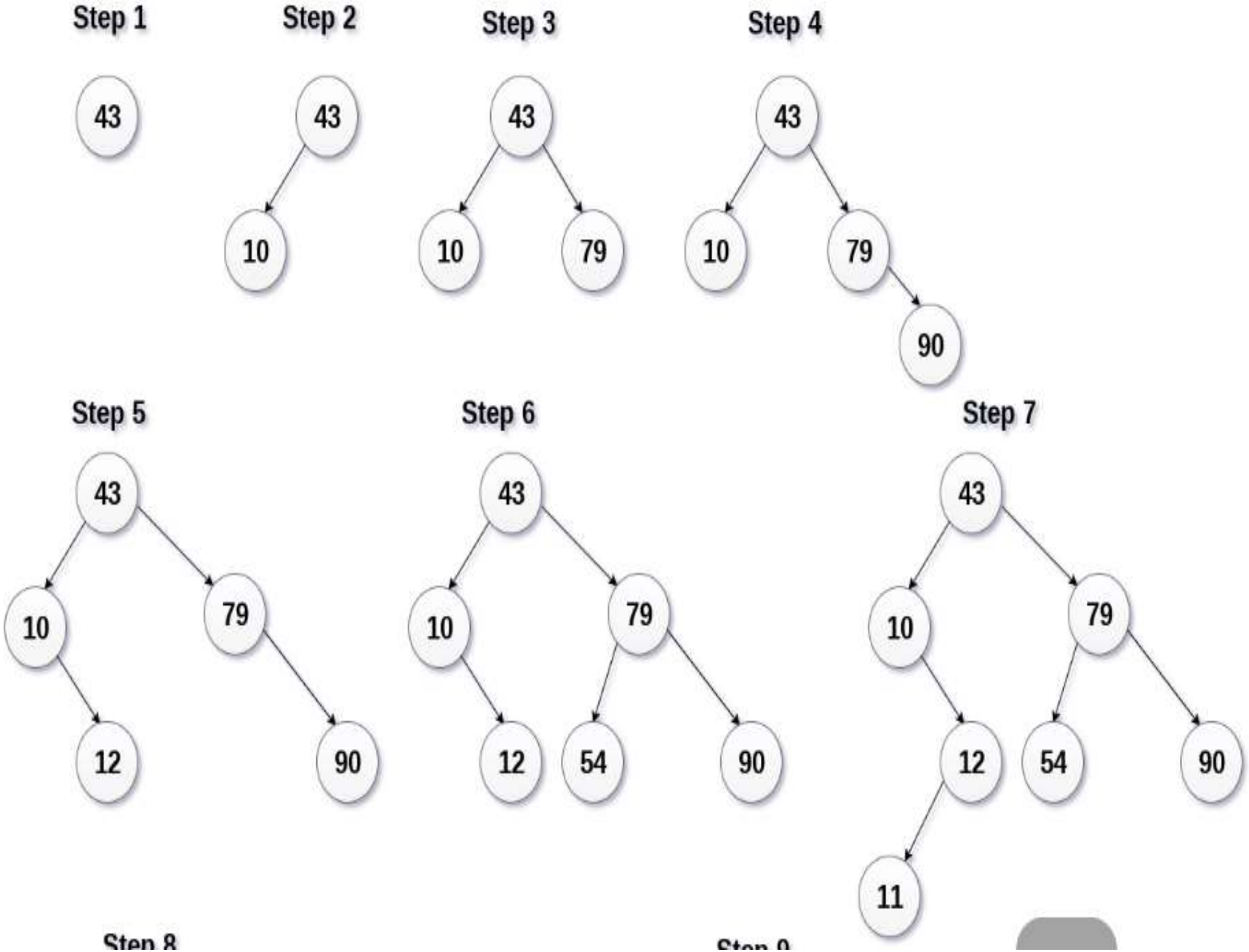
- Tree structures are used to store data because their organization enables efficient access to the data.
- A binary *search* tree labels each node in a binary tree with a single key such that for any node $x$, and nodes in the left subtree of $x$ have keys $\leq x$ and all nodes in the right subtree of $x$ have key's $> x$ .
- I.e. If k is the key value at any node, then $k \geq x$ for every key x in the node's left subtree and $k < y$ for every key $y$ in the node's right subtree.

# BST example

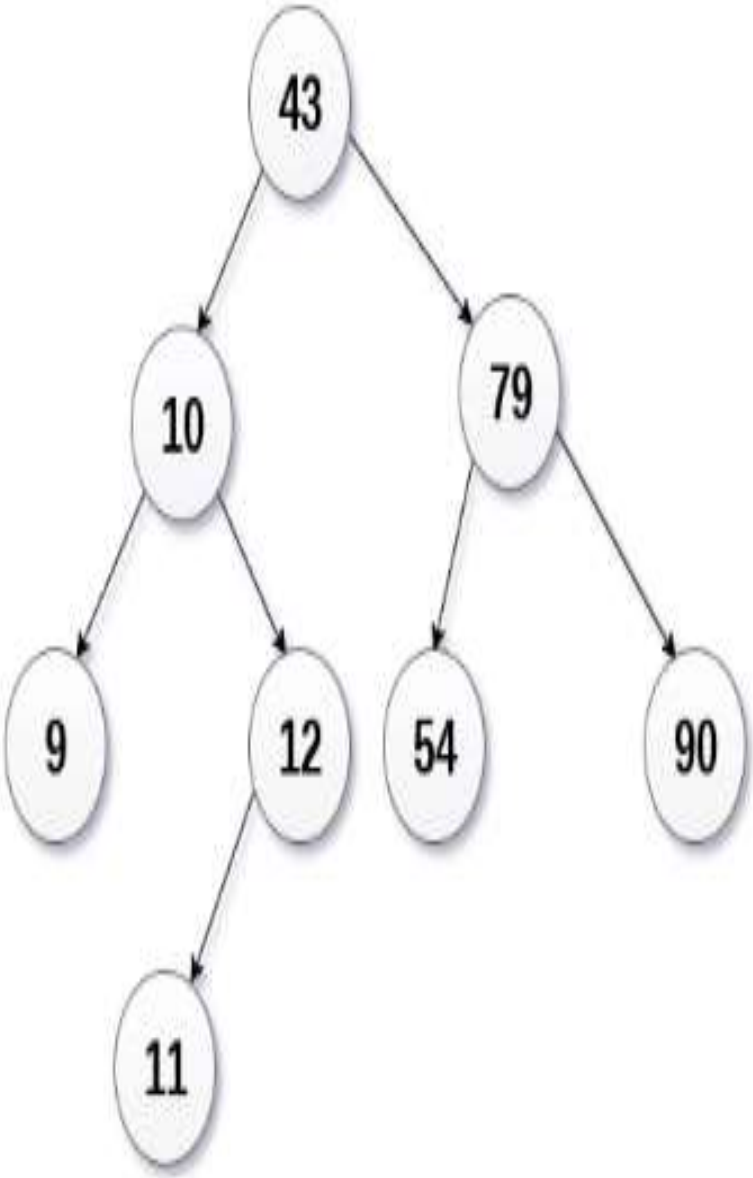


**Create the binary search tree using the following data elements.**
**43, 10, 79, 90, 12, 54, 11, 9, 50**

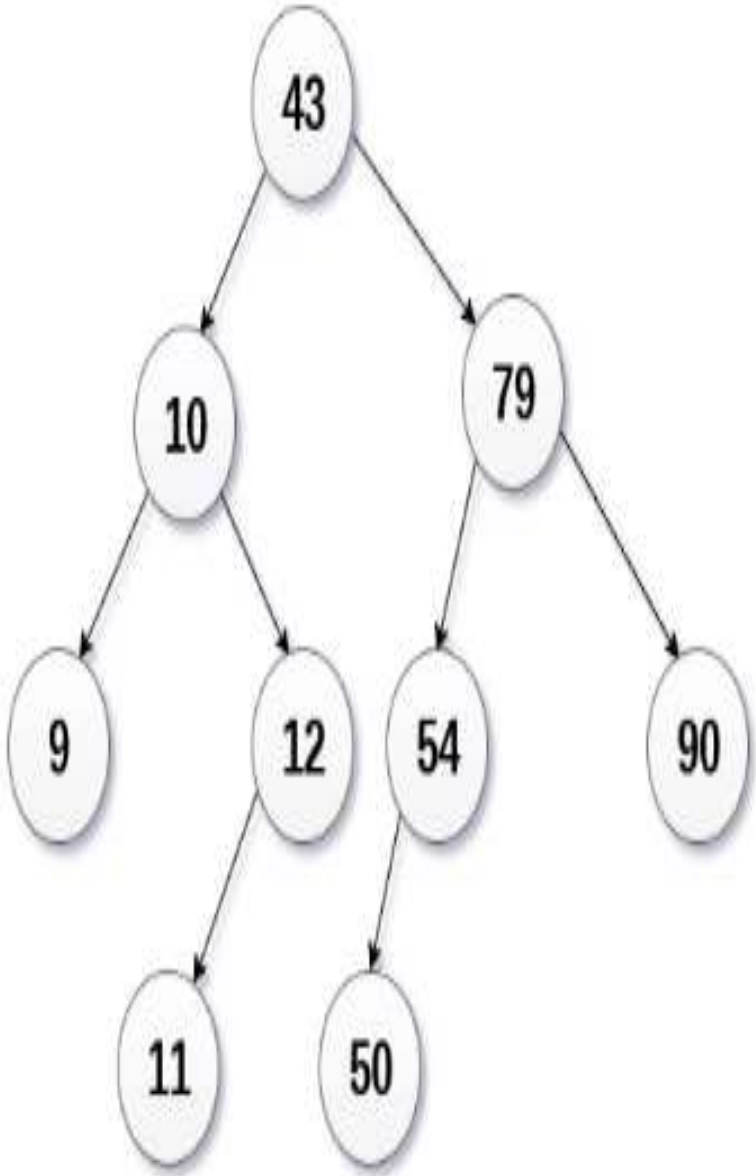
a) Insert 43 into the tree as the root of the tree.
b) Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
c) Otherwise, insert it as the root of the right of the right sub-tree.


Step 1
43
Step 2
43
10
Step 3
43
10
79
Step 4
43
10
79
90
Step 5
43
10
79
12
90
Step 6
43
10
79
12
54
90
Step 7
43
10
79
12
54
90
11
Step 8
Step 9

Step 8




Step 9



**END.**
**WAIRAGU G.R.**