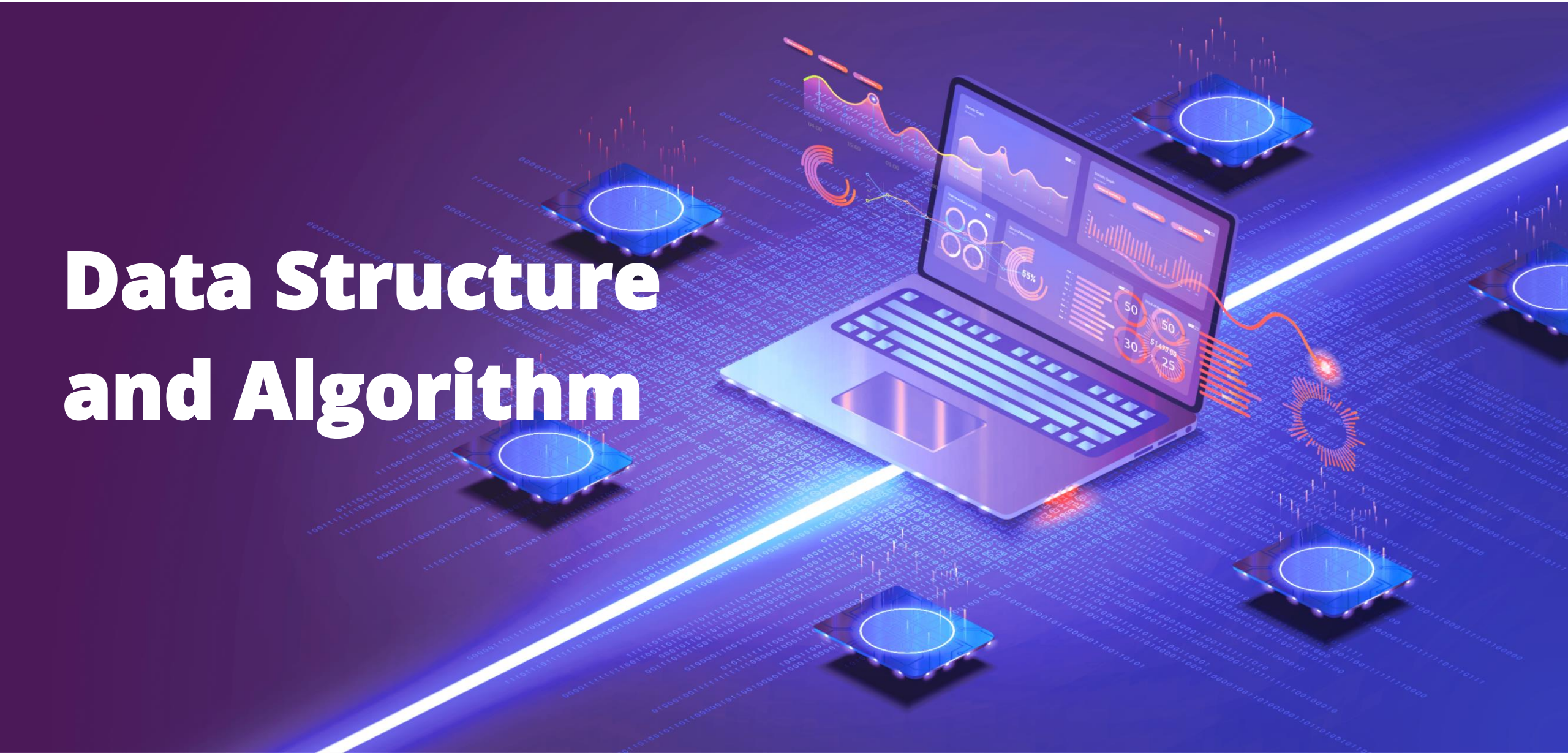
WILEY

ENABLING DISCOVERY | POWERING EDUCATION | SHAPING WORKFORCES

Data Structure and Algorithm



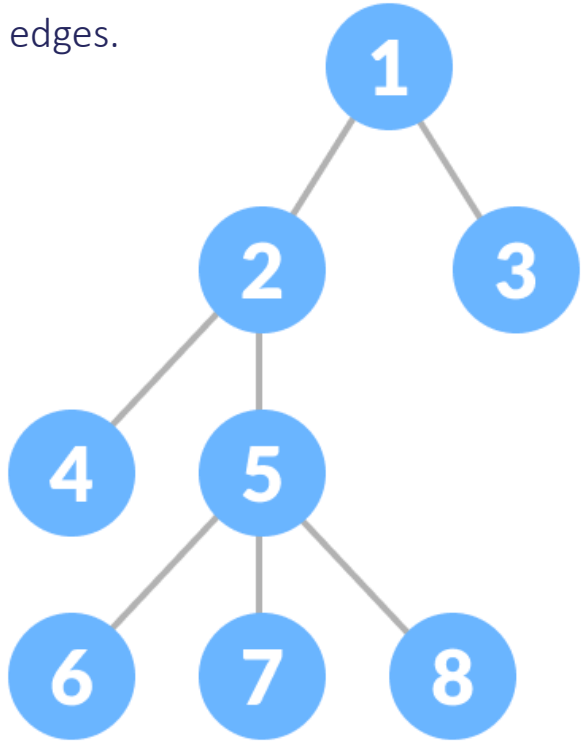
DATA STRUCTURES ALGORITHMS

TREES

Tree Data Structure

Let’s understand what about tree data structure. Also, you will learn about different types of trees and the terminologies used in tree.

A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



Why Tree Data Structure?

Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially. In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size. But, it is not acceptable in today's computational world.

Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

Tree Terminologies

Node

A node is an entity that contains a key or value and pointers to its child nodes.

The last nodes of each path are called **leaf nodes or external nodes** that do not contain a link/pointer to child nodes.

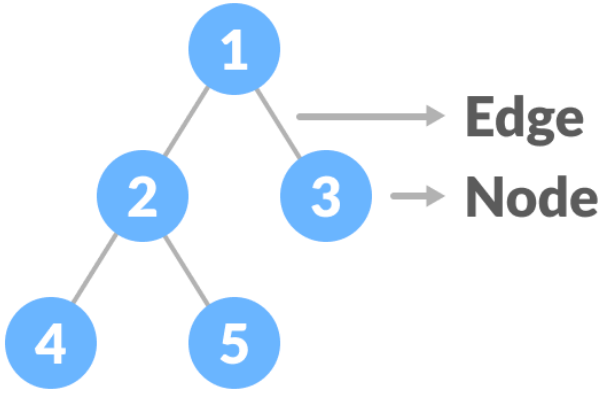
The node having at least a child node is called an **internal node**.

Edge

It is the link between any two nodes.

Root

It is the topmost node of a tree.



Height of a Node

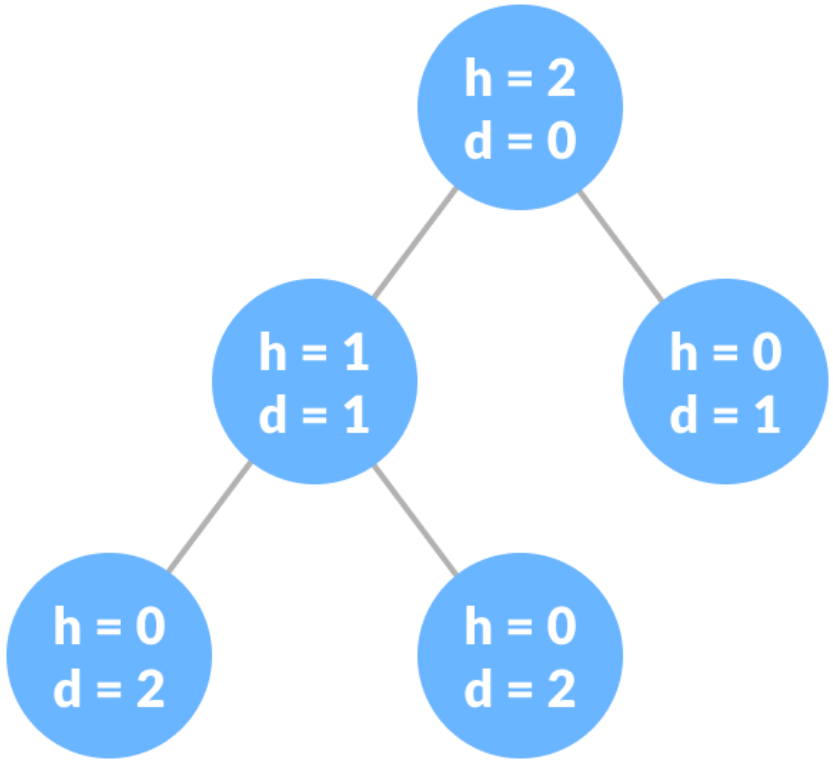
The height of a node is the number of edges from the node to the deepest leaf (ie. the longest path from the node to a leaf node).

Depth of a Node

The depth of a node is the number of edges from the root to the node.

Height of a Tree

The height of a Tree is the height of the root node or the depth of the deepest node.

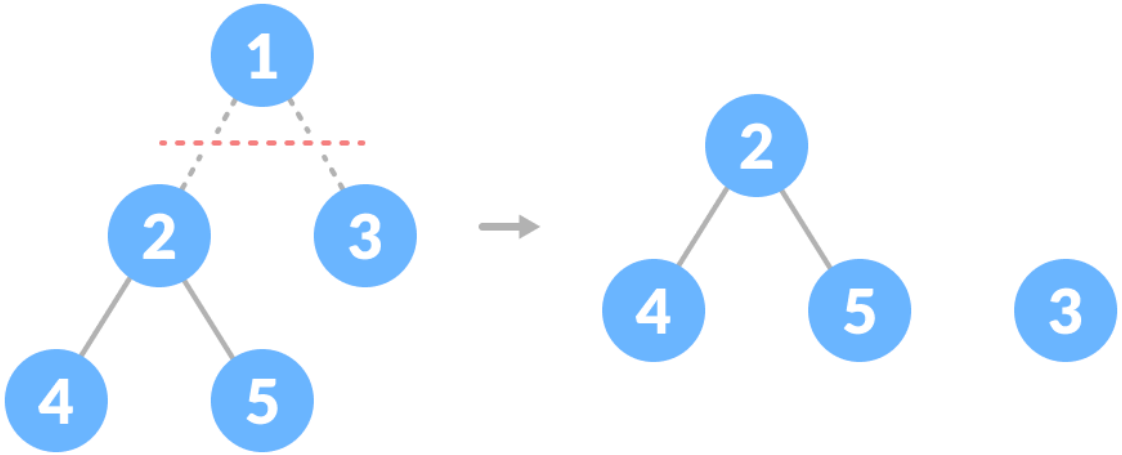


Degree of a Node

The degree of a node is the total number of branches of that node.

Forest

A collection of disjoint trees is called a forest.



Types of Tree

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree

Tree Traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

Tree Applications

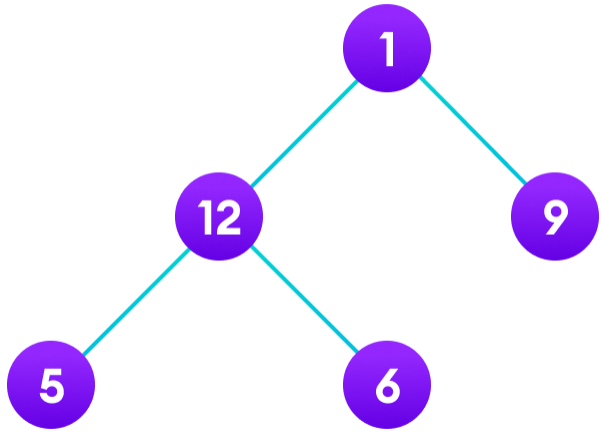
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure
- Compilers use a syntax tree to validate the syntax of every program you write.

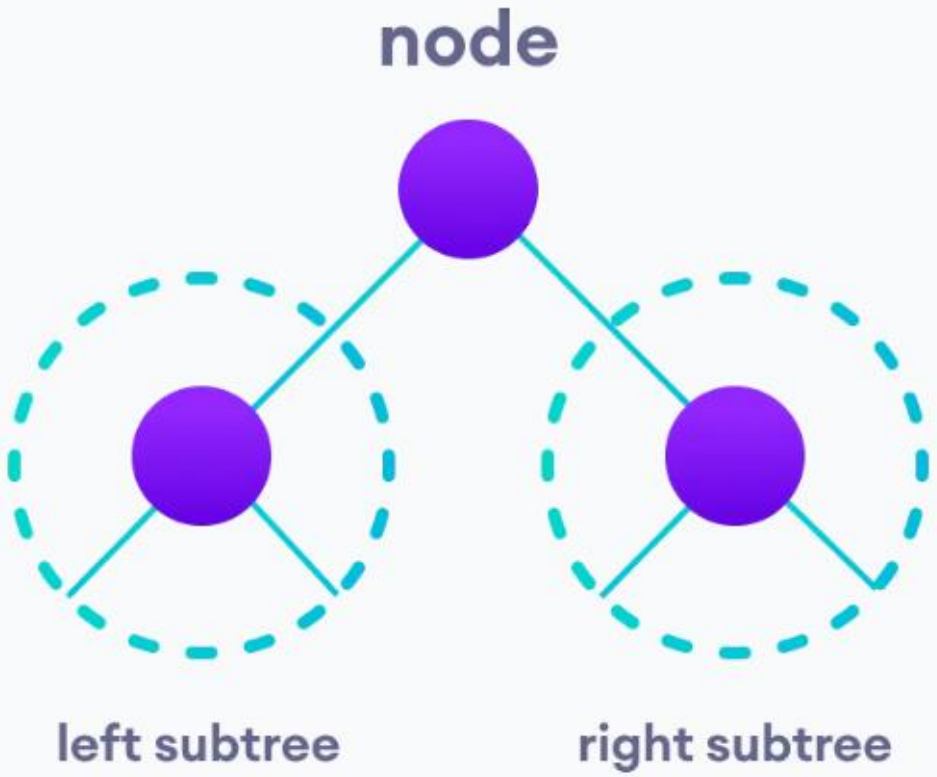
Tree Traversal - inorder, preorder and postorder

Different tree traversal techniques. Also, you will find working examples of different tree traversal methods in C, C++, Java and Python.

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.





Left and Right Subtree

Inorder traversal

```
inorder(root->left)
display(root->data)
inorder(root->right)
```

- First, visit all the nodes in the left subtree
- Then the root node
- Visit all the nodes in the right subtree

Preorder traversal

```
display(root->data)
preorder(root->left)
preorder(root->right)
```

- Visit root node
- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree

Postorder traversal

```
postorder(root->left)
postorder(root->right)
display(root->data)
```

- Visit all the nodes in the left subtree
- Visit all the nodes in the right subtree
- Visit the root node

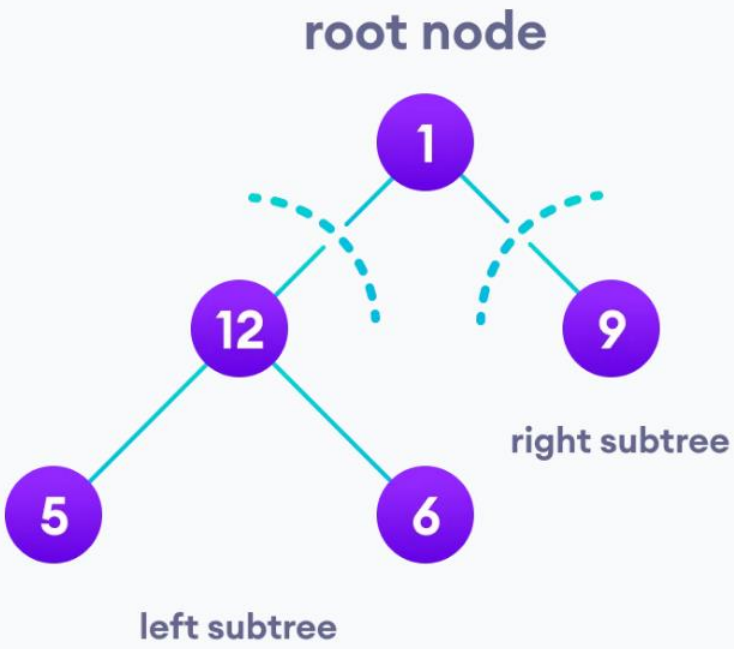
Starting from top, Left to right

```
1 -> 12 -> 5 -> 6 -> 9
```

Starting from bottom, Left to right

```
5 -> 6 -> 12 -> 9 -> 1
```

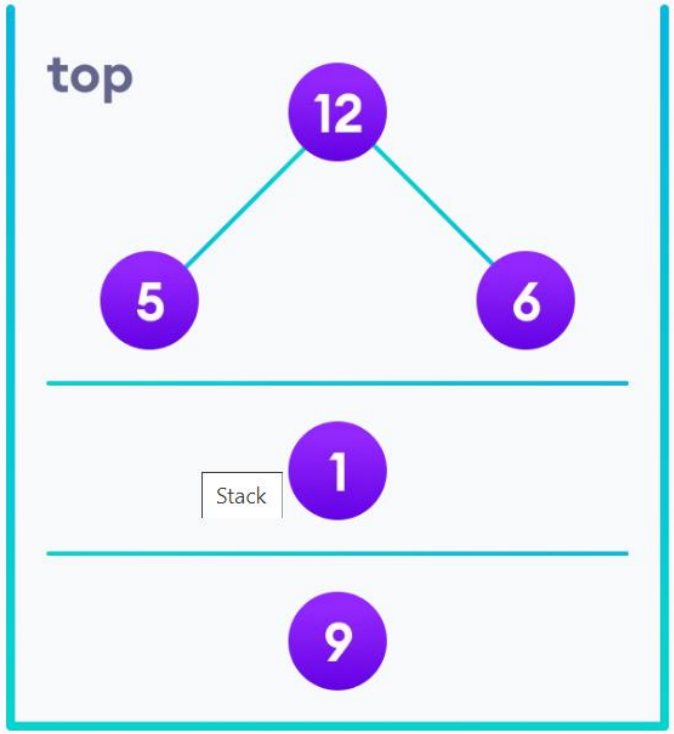
Let's visualize in-order traversal. We start from the root node.



Left and Right Subtree

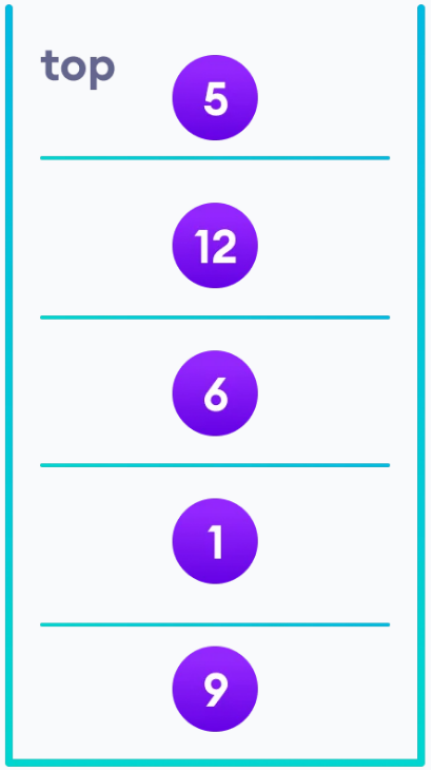
We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a stack so that we remember.



Stack

After traversing the left subtree, we are left with



Final Stack

Since the node "5" doesn't have any subtrees, we print it directly. After that we print its parent "12" and then the right child "6".

Putting everything on a stack was helpful because now that the left-subtree of the root node has been traversed, we can print it and go to the right subtree.

After going through all the elements, we get the inorder traversal as

```
5 -> 12 -> 6 -> 1 -> 9
```

We don't have to create the stack ourselves because recursion maintains the correct order for us.

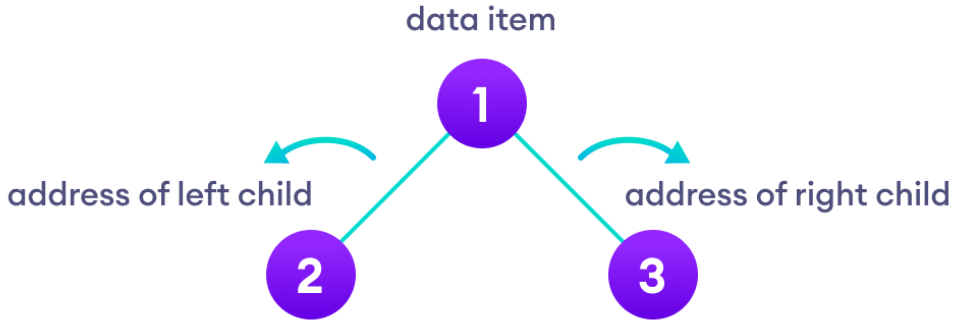
Tree and Tree Traversal Implementation

Binary Tree Data Structure

Let’s understand about binary tree and its different types. Also, you will find working examples of binary tree in C, C++, Java and Python.

A binary tree is a tree data structure in which each parent node can have at most two children. Each node of a binary tree
consists of three items:

- Data item
- Address of left child
- Address of right child

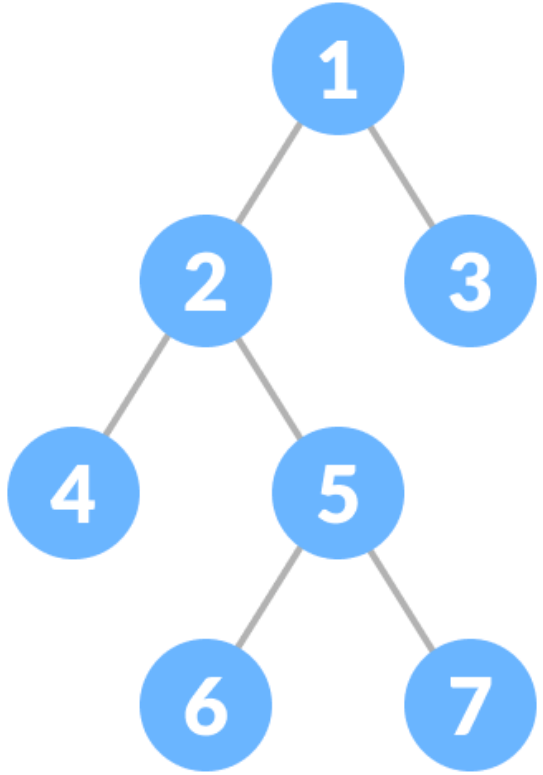


Types of Binary Trees

- Full Binary Tree
- Perfect Binary Tree
- Complete Binary Tree
- Degenerate or Pathological Tree
- Skewed Binary Tree
- Balanced Binary Tree

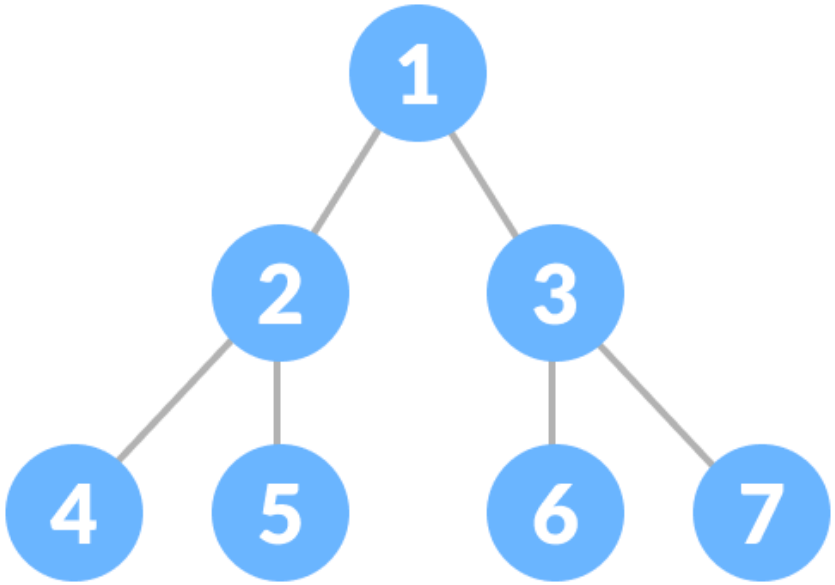
Full Binary Tree

A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children



Perfect Binary Tree

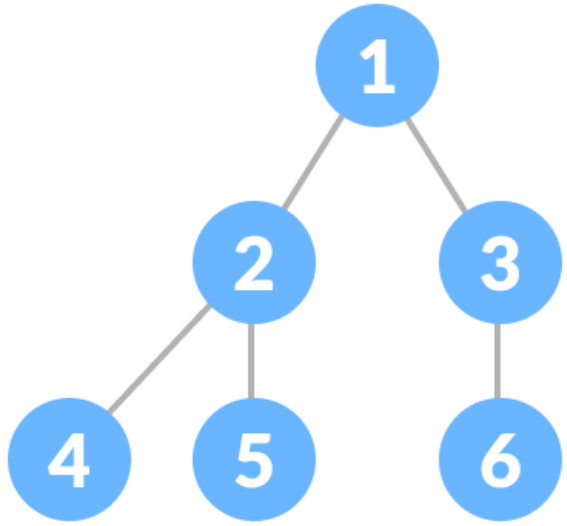
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Complete Binary Tree

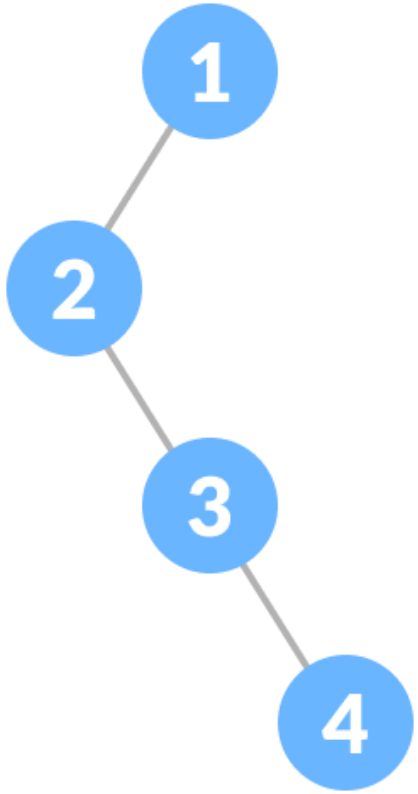
A complete binary tree is just like a full binary tree, but with two major differences

1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



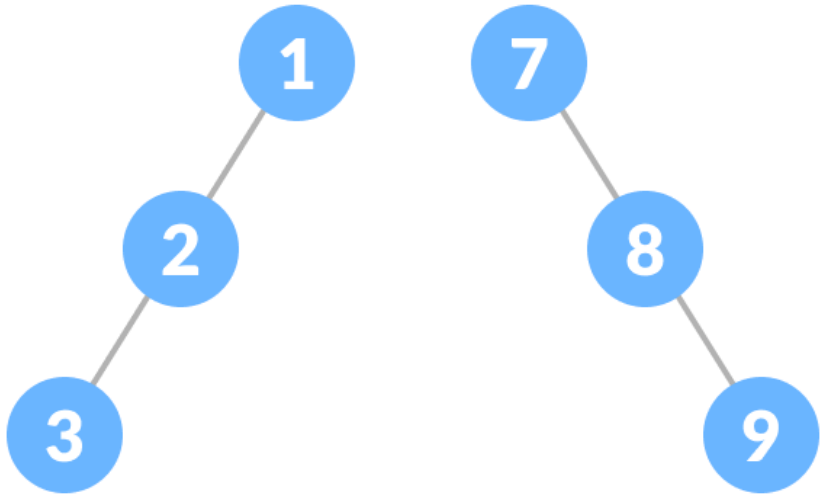
Degenerate or Pathological Tree

A degenerate or pathological tree is the tree having a single child either left or right.



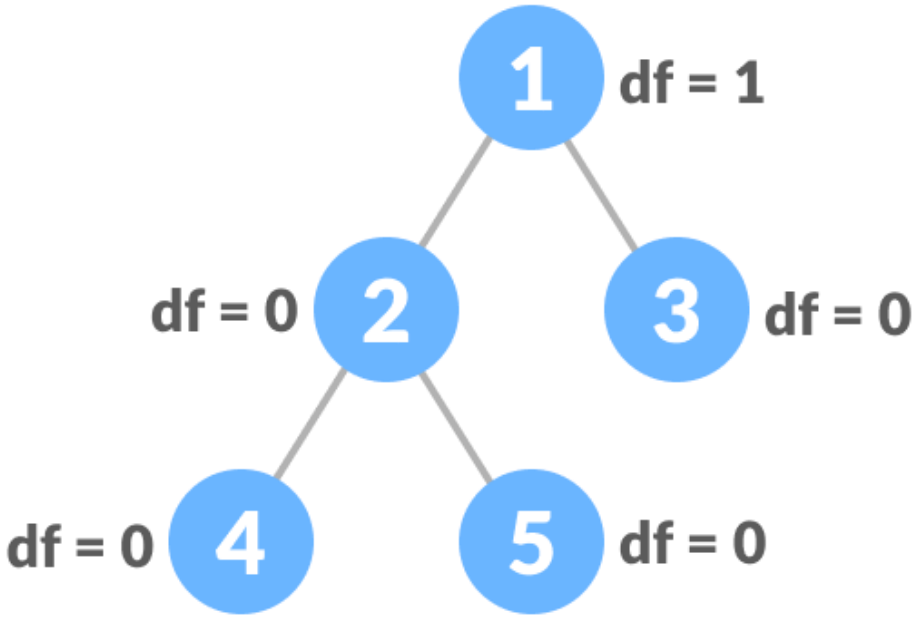
Skewed Binary Tree

A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes. Thus, there are two types of skewed binary tree: **left-skewed binary tree** and **right-skewed binary tree**.



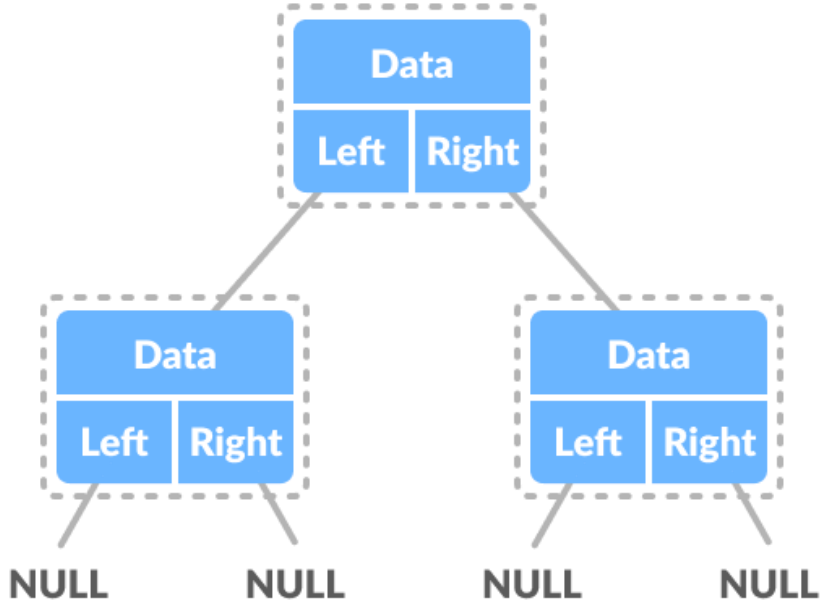
Balanced Binary Tree

It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.



Binary Tree Implementation

Binary Tree Applications

- For easy and quick access to data
- In router algorithms
- To implement heap data structure
- Syntax tree

Full Binary Tree

Let’s understand about full binary tree and its different theorems. Also, you will find working examples to check full binary tree in C, C++, Java and Python.

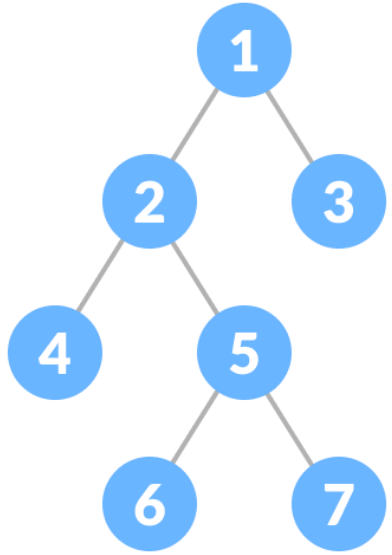
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.

It is also known as **a proper binary tree**.

Full Binary Tree Theorems

```
Let, i = the number of internal nodes
       n = be the total number of nodes
       l = number of leaves
      λ = number of levels
```

1. The number of leaves is `i + 1`.
2. The total number of nodes is `2i + 1`.
3. The number of internal nodes is `(n - 1) / 2`.
4. The number of leaves is `(n + 1) / 2`.
5. The total number of nodes is `2l - 1`.
6. The number of internal nodes is `l - 1`.
7. The number of leaves is at most $2^{\lambda - 1}$.

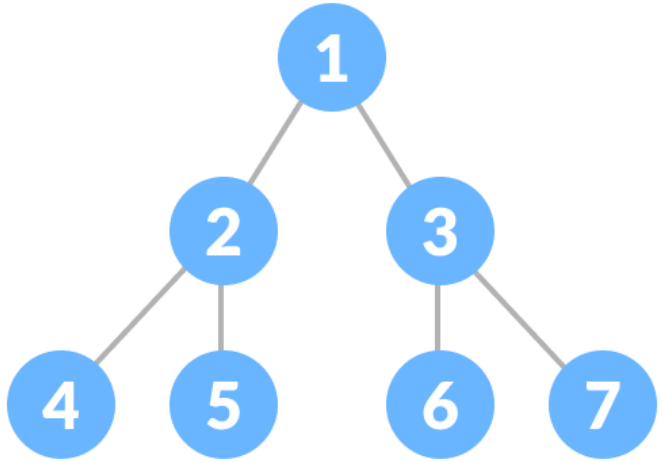


Full Binary Tree Implementation

Perfect Binary Tree

Let’s learn about the perfect binary tree. Also, you will find working examples for checking a perfect binary tree in C, C++, Java and Python.

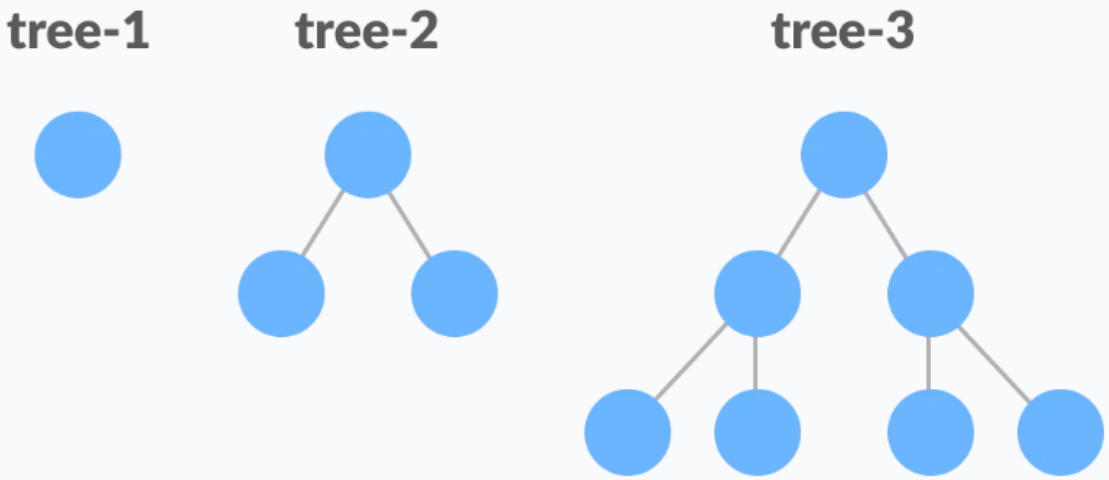
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



All the internal nodes have a degree of 2.

Recursively, a perfect binary tree can be defined as:

1. If a single node has no children, it is a perfect binary tree of height `h = 0`,
2. If a node has `h > 0`, it is a perfect binary tree if both of its subtrees are of height `h - 1` and are non-overlapping.



Perfect Binary Tree (Recursive Representation)

Perfect Binary Tree Implementation

Complete Binary Tree

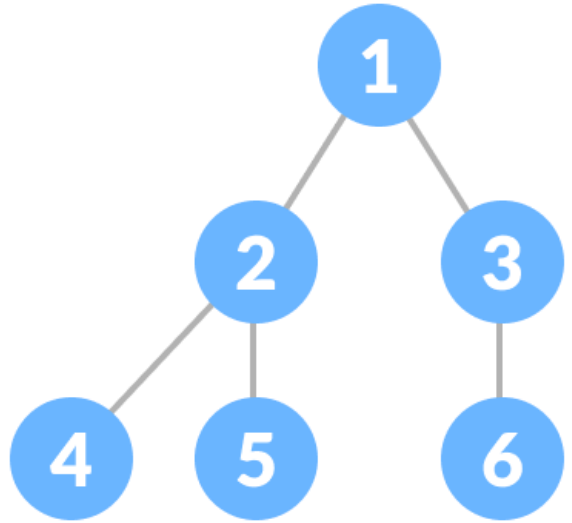
Let’s learn about a complete binary tree and its different types. Also, you will find working examples of a complete binary tree in C, C++, Java and Python.

A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

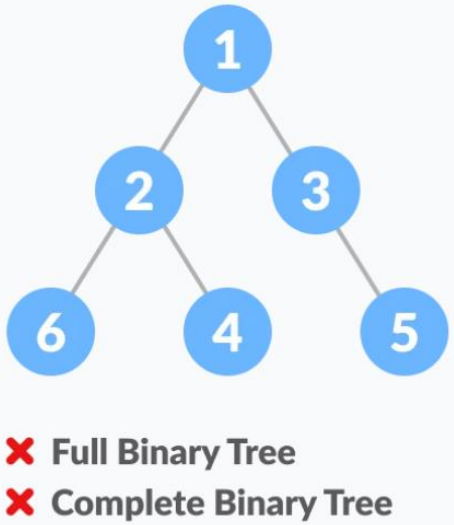
A complete binary tree is just like a full binary tree, but with two major differences

1. All the leaf elements must lean towards the left.
2. The last leaf element might not have a right sibling

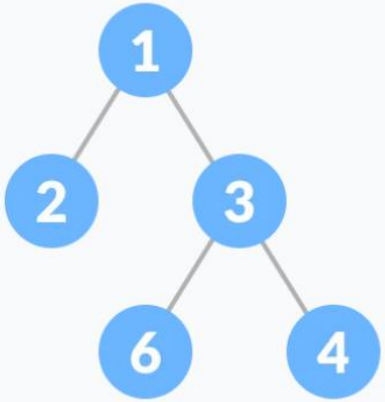
   i.e. a complete binary tree doesn't have to be a full binary tree.



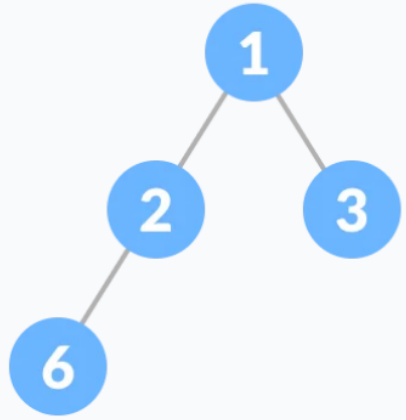
Full Binary Tree vs Complete Binary Tree



Comparison between full binary tree and complete binary tree

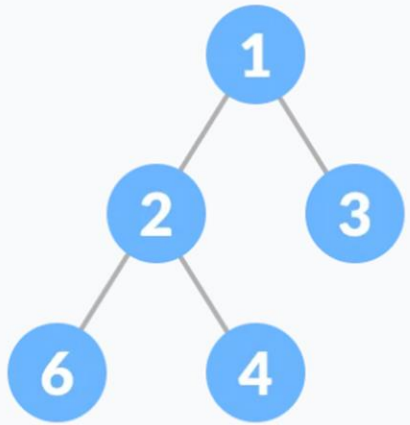


Comparison between full binary tree and complete binary tree



✗ Full Binary Tree
✓ Complete Binary Tree

Comparison between full binary tree and complete binary tree



✔ Full Binary Tree
✔ Complete Binary Tree

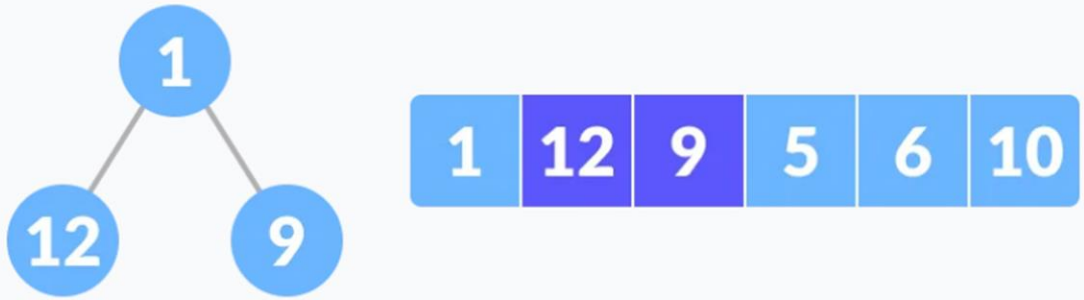
Comparison between full binary tree and complete binary tree

How a Complete Binary Tree is Created?

1. Select the first element of the list to be the root node. (no. of elements on level-I: 1)

Select the first element as root

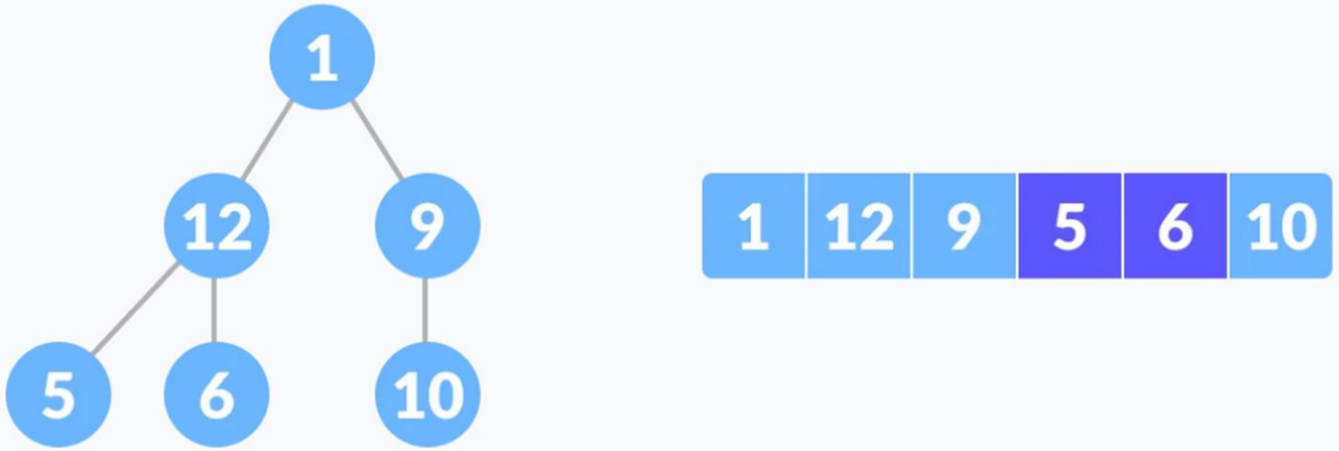
2. Put the second element as a left child of the root node and the third element as the right child. (no. of elements on level-II: 2)



12 as a left child and 9 as a right child

3. Put the next two elements as children of the left node of the second level. Again, put the next two elements as children of the right node of the second level (no. of elements on level-III: 4) elements).

4. Keep repeating until you reach the last element.



5 as a left child and 6 as a right child

Complete Binary Tree Implementation

Relationship between array indexes and tree element

A complete binary tree has an interesting property that we can use to find the children and parents of any node.

If the index of any element in the array is `i`, the element in the index `2i+1` will become the left child and element in `2i+2` index will become the right child. Also, the parent of any element at index `i` is given by the lower bound of `(i-1)/2`.

Let's test it out,

```
Left child of 1 (index 0)
= element in (2*0+1) index
= element in 1 index
= 12


Right child of 1
= element in (2*0+2) index
= element in 2 index
= 9

Similarly,
Left child of 12 (index 1)
= element in (2*1+1) index
= element in 3 index
= 5

Right child of 12
= element in (2*1+2) index
= element in 4 index
= 6
```

Let us also confirm that the rules hold for finding parent of any node

```
Parent of 9 (position 2)
= (2-1)/2
= ½
= 0.5
~ 0 index
= 1

Parent of 12 (position 1)
= (1-1)/2
= 0 index
= 1
```

Understanding this mapping of array indexes to tree positions is critical to understanding how the Heap Data Structure works and how it is used to implement Heap Sort.

Complete Binary Tree Applications

- Heap-based data structures
- Heap sort

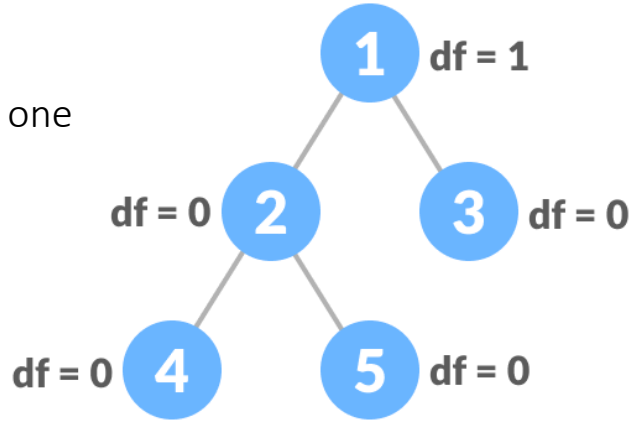
Balanced Binary Tree

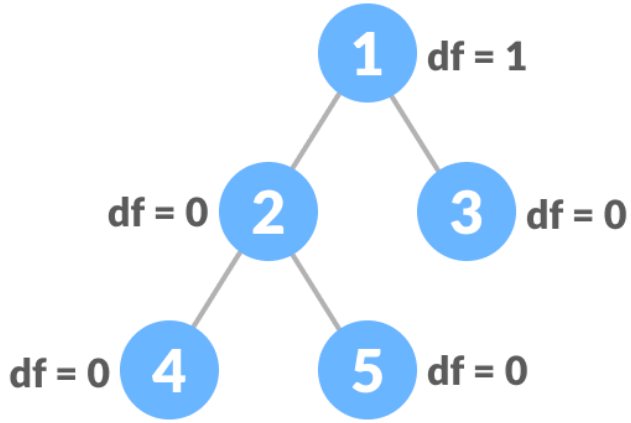
Let’s learn about a balanced binary tree and its different types. Also, you will find working examples of a balanced binary tree in C, C++, Java and Python.

A balanced binary tree, also referred to as a height-balanced binary tree, is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

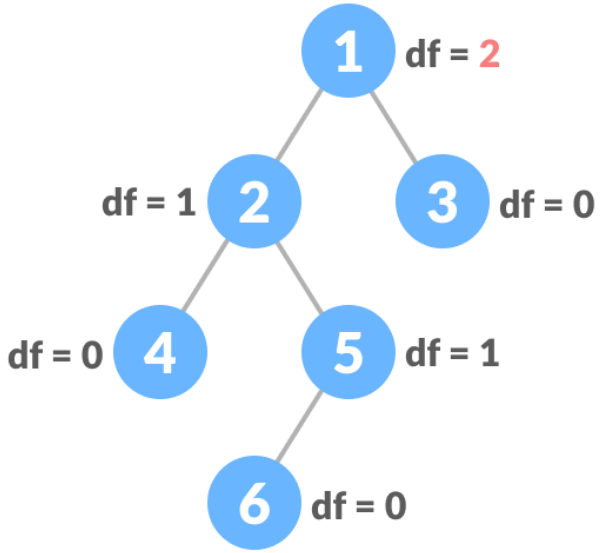
Following are the conditions for a height-balanced binary tree:

1. difference between the left and the right subtree for any node is not more than one
2. the left subtree is balanced
3. the right subtree is balanced





Balanced Binary Tree with depth at each level



df = |height of left child - height of right child|

Unbalanced Binary Tree with depth at each level

Balanced Binary Tree Implementation

Balanced Binary Tree Applications

- AVL tree
- Balanced Binary Search Tree

Binary Search Tree (BST)

Lets learn how Binary Search Tree works. Also, we will find working examples of Binary Search Tree in C, C++, Java and Python.

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

It is called a binary tree because each tree node has a maximum of two children.

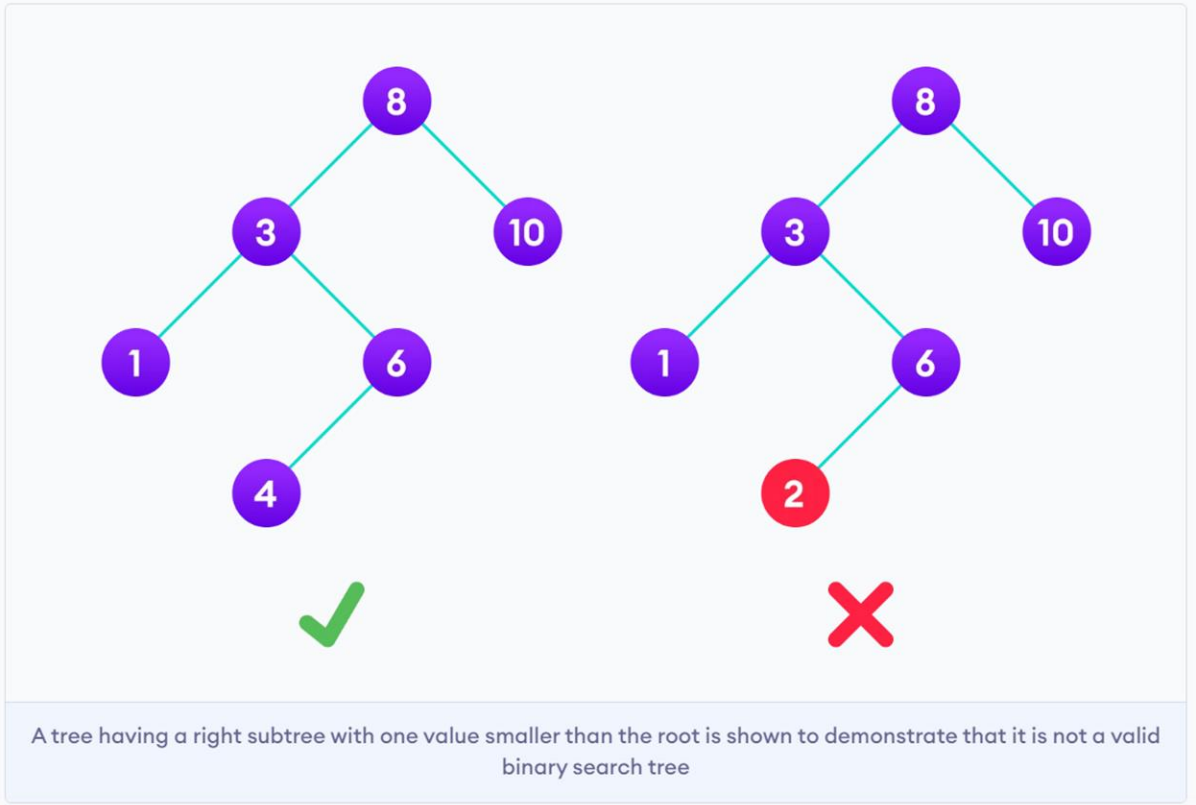
It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e. they have the above two properties

The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

There are two basic operations that you can perform on a binary search tree:



A tree having a right subtree with one value smaller than the root is shown to demonstrate that it is not a valid binary search tree

Search Operation

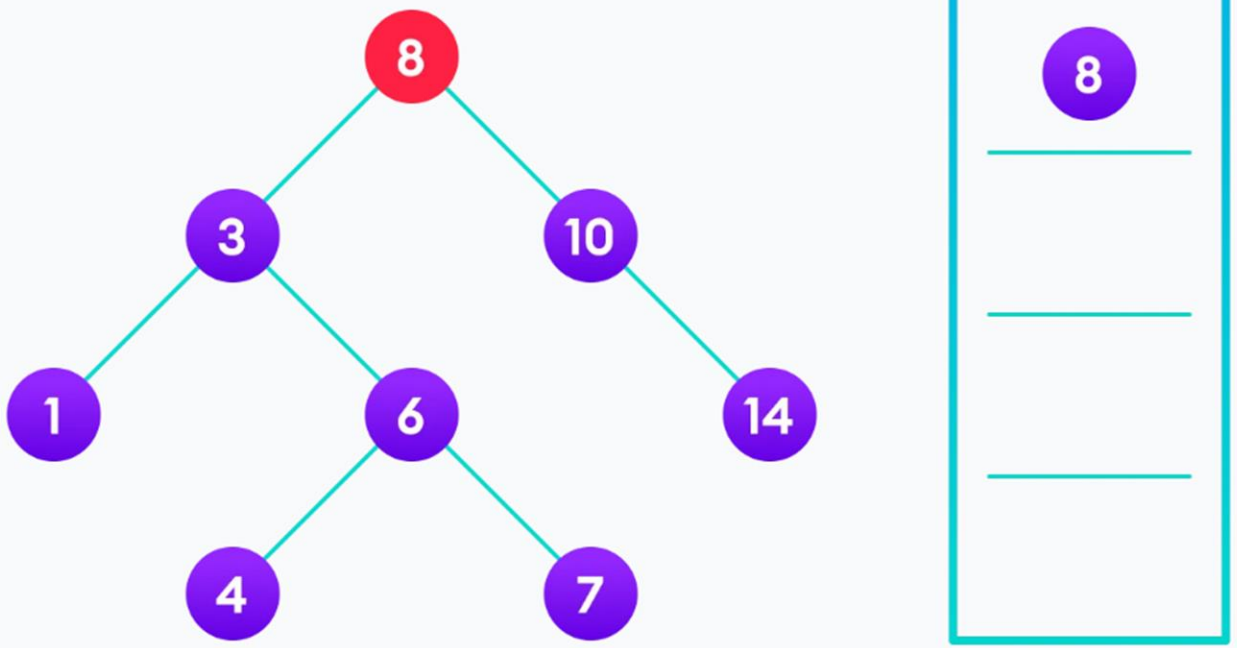
The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.

If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

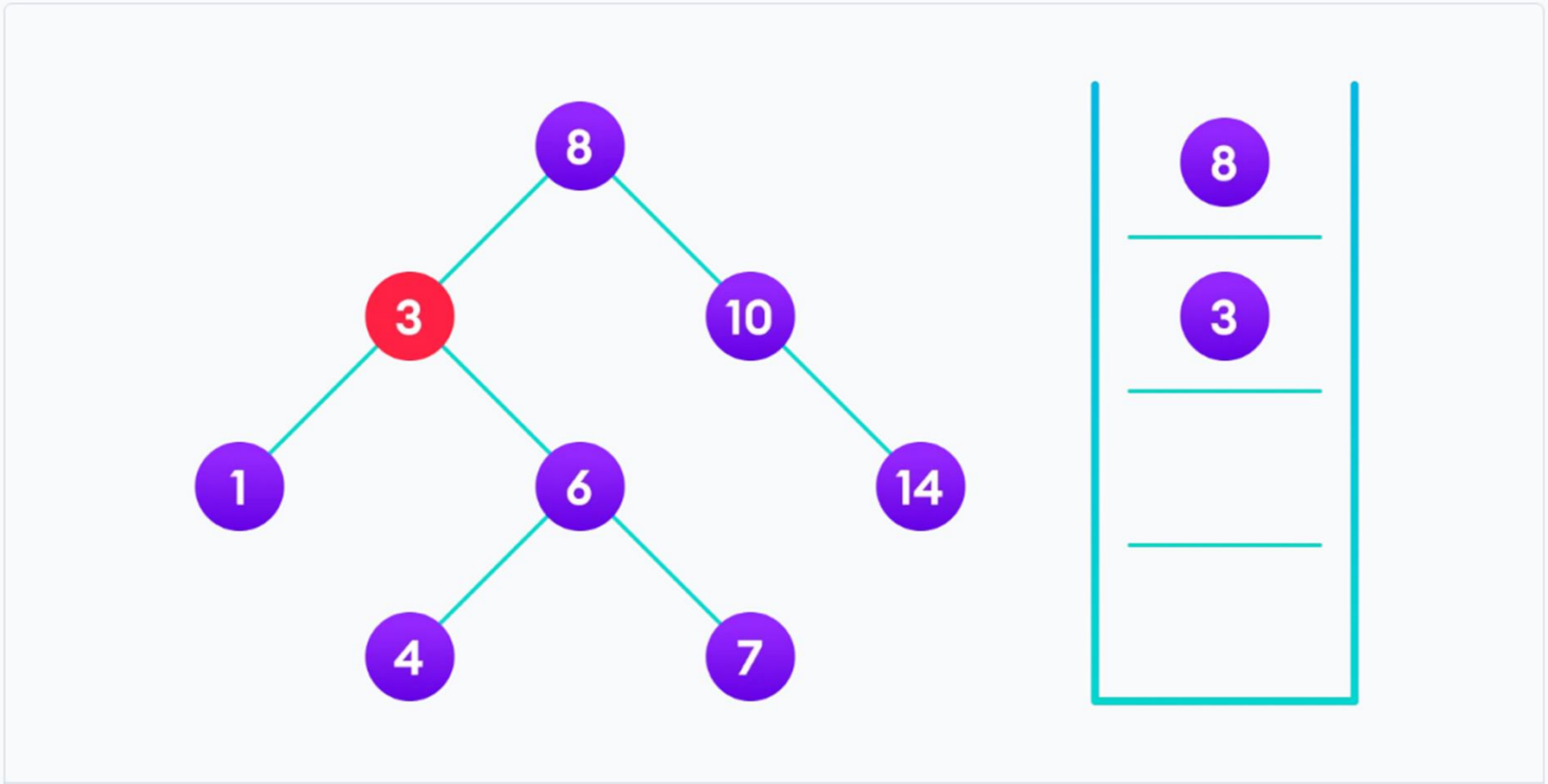
Algorithm:

```
If root == NULL
    return NULL;
If number == root->data
    return root->data;
If number < root->data
    return search(root->left)
If number > root->data
    return search(root->right)
```

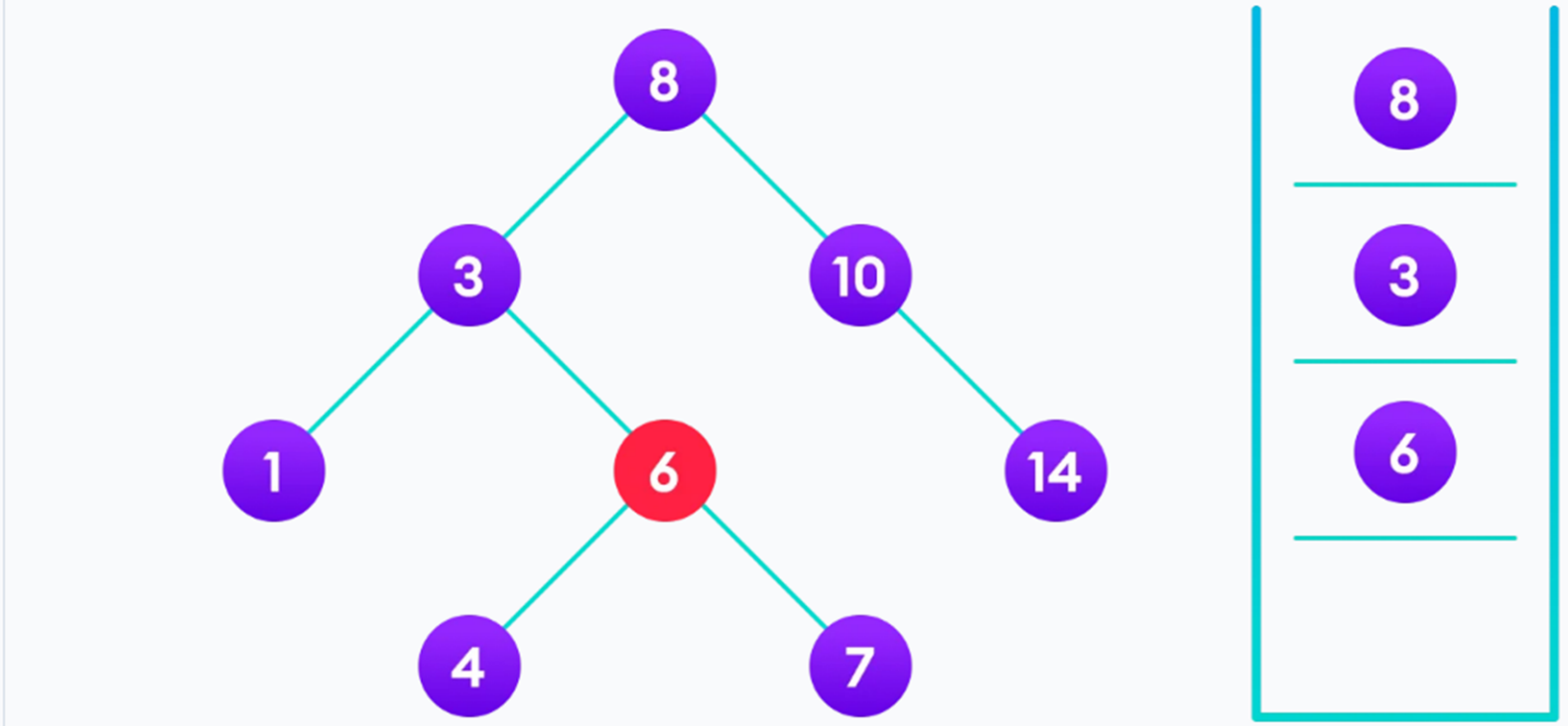
Let us try to visualize this with a diagram.



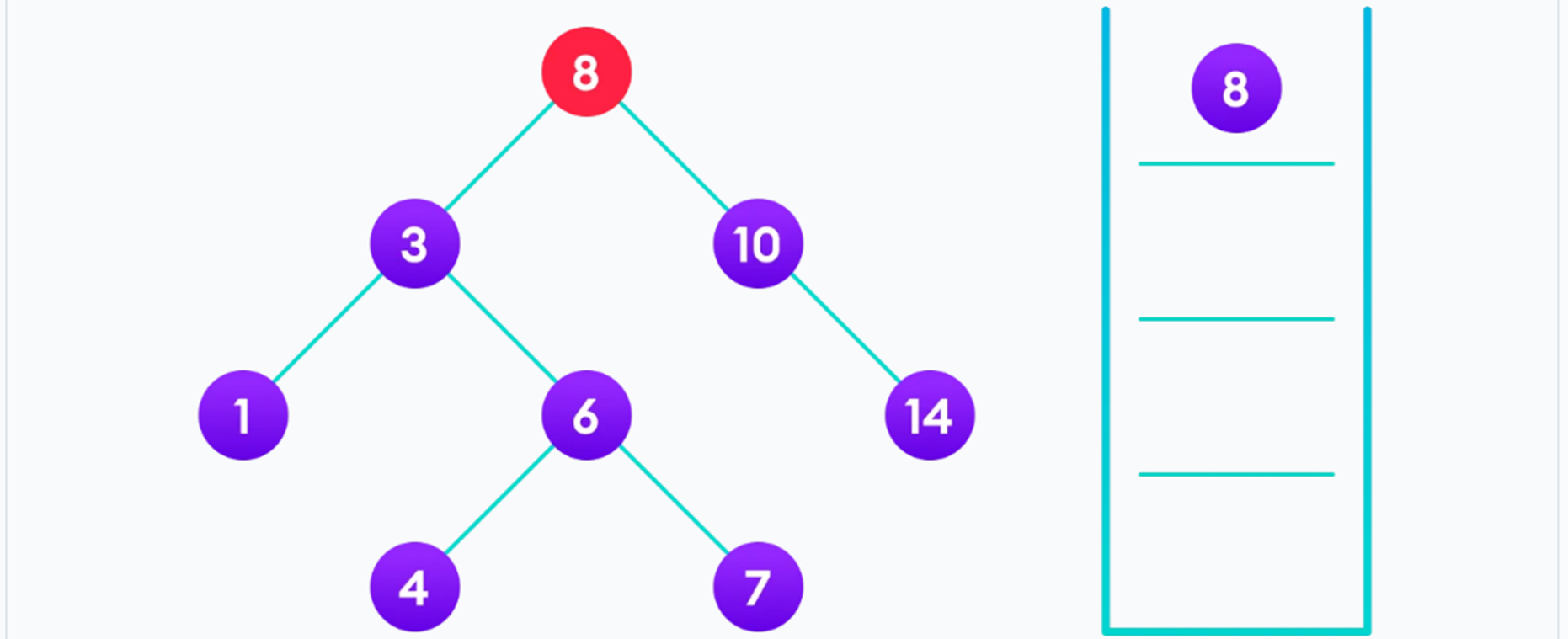
4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3



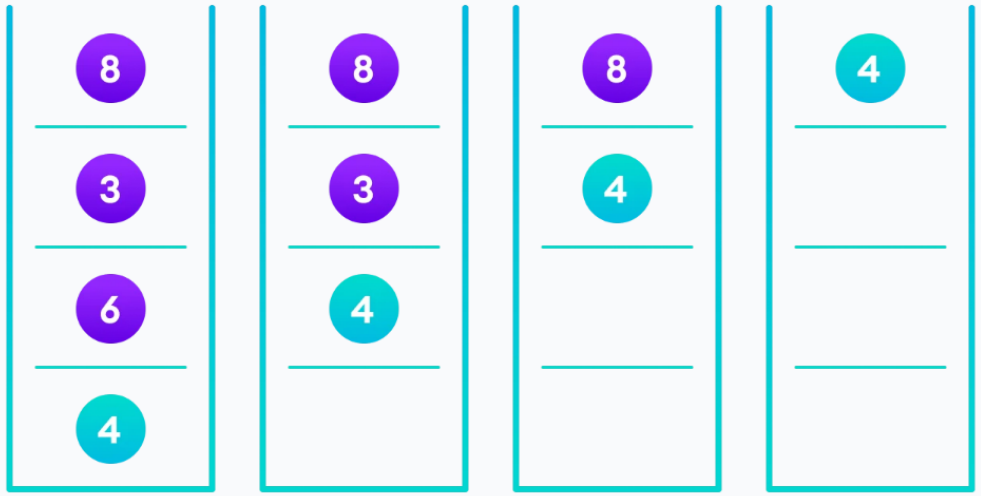
4 is not found so, traverse through the left subtree of 6



4 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

If you might have noticed, we have called return search(struct node*) four times. When we return either the new node or NULL, the value gets returned again and again until search(root) returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

Binary Search Tree Implementation

Binary Search Tree Complexities

Time Complexity

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	O(log n)	O(log n)	O(n)
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)

Here, `n` is the number of nodes in the tree.

Space Complexity

The space complexity for all the operations is `O(n)`.

Binary Search Tree Applications

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel