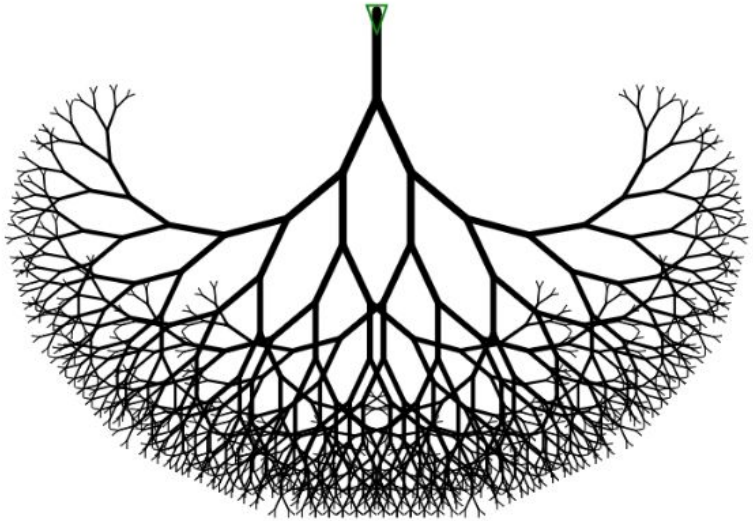
Trees

Binary Trees and Binary Search Trees



Lecture Flow

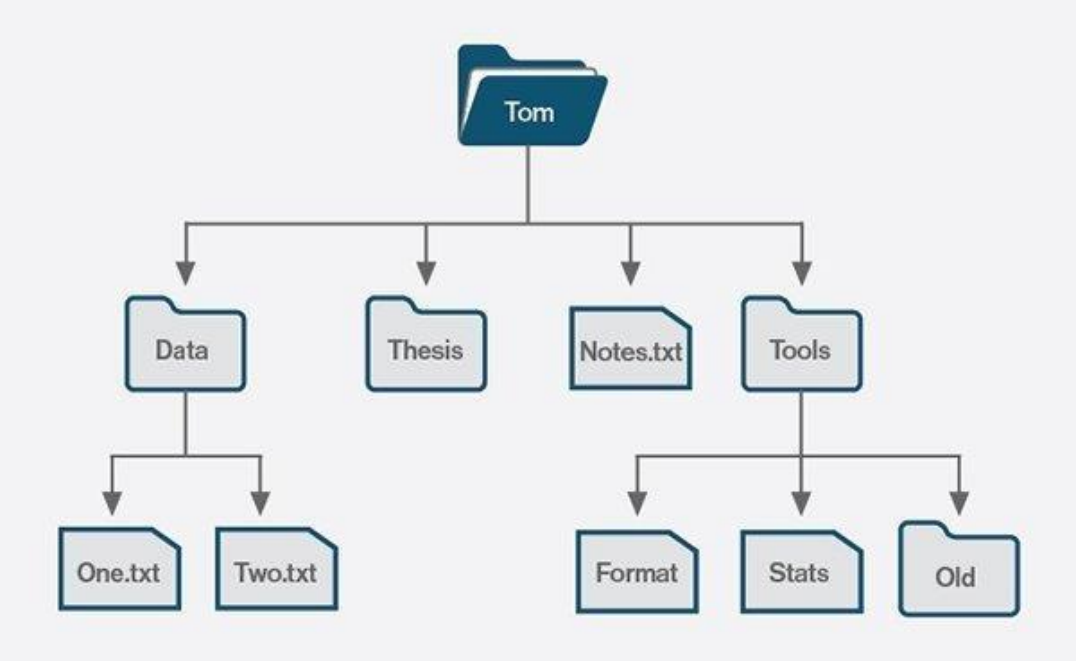
1. Prerequisites
2. Real life problem
3. Definition
4. Tree Terminologies
5. Types of Trees
6. Tree Traversal and basic Tree Operations
7. Time and space complexity analysis
8. Things to pay attention to(common pitfalls)
9. Applications of a Tree

Pre-requisites

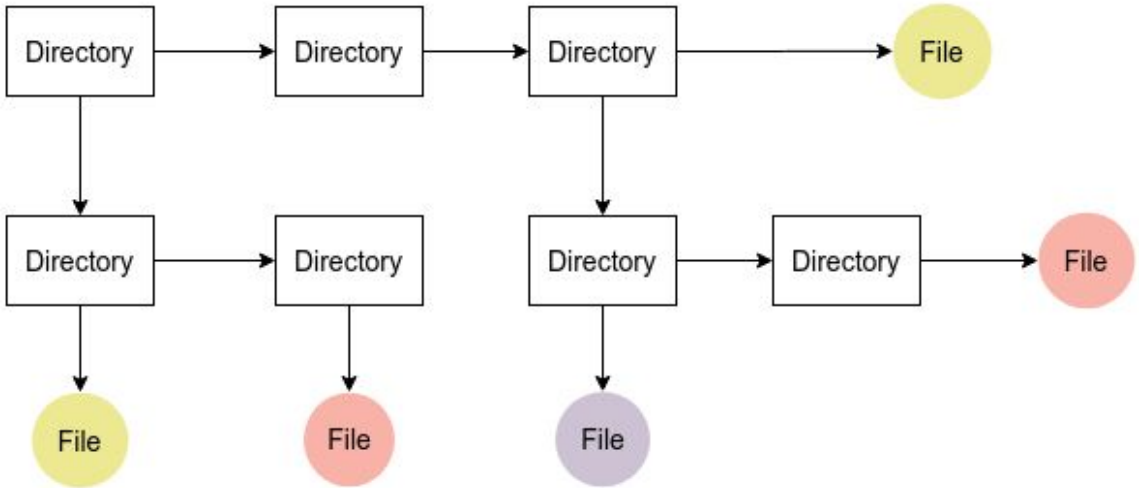
- Basic Recursion
- Linked List
- Stacks
- Queues

Real life problem

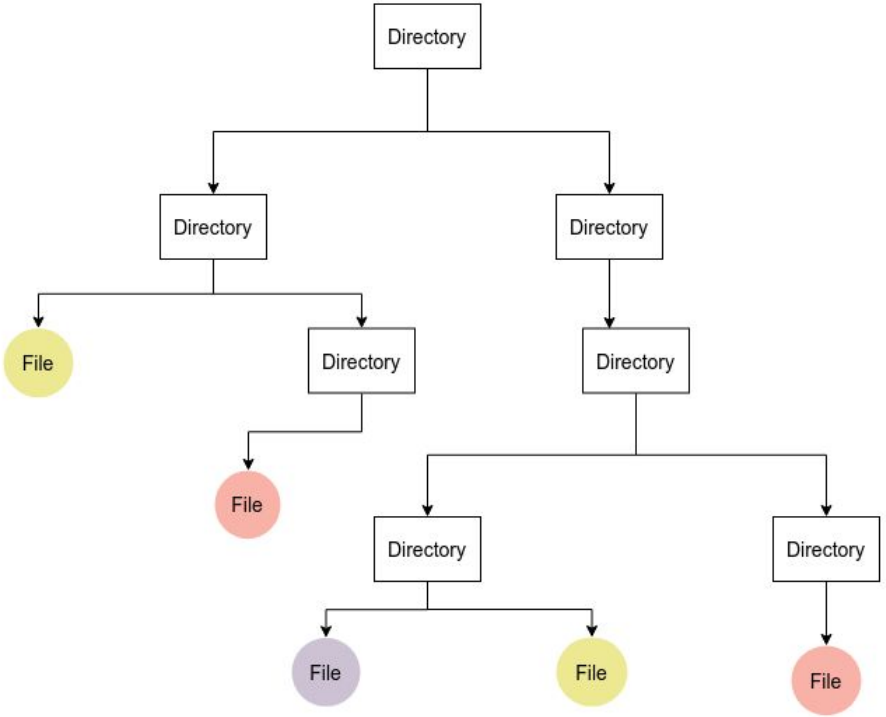
What data structure will be best to implement a file management system ?



- Considering all these cases, we know that an array, stack, or queue will not work. Can we implement this with the help of a linked list? We could make a linked list in such a way that if there is a directory at any particular node, we create a new linked list from here.
  What will this look like?



- It looks similar to a tree, right? So, we tried to use a linked list but ended up using a tree because the problem can easily be solved using **tree** data structure.
- Each internal node of the tree will be a directory, and each leaf node will be a file.



Problem Definition

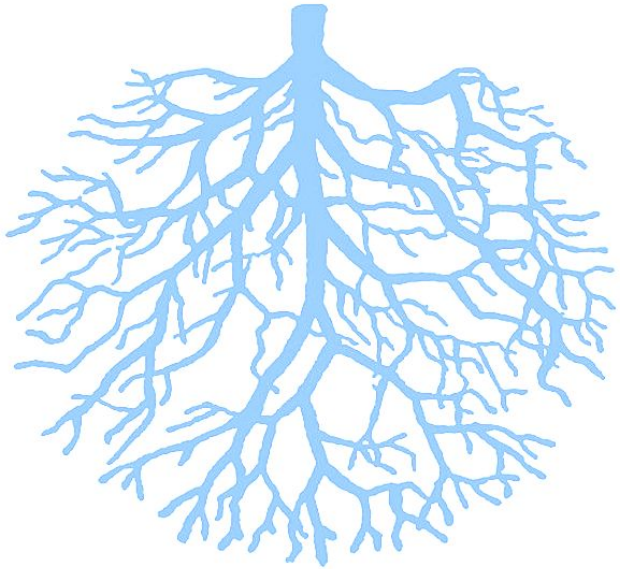
Can we have a better data structure which provides easier and quicker access than linear data structures such as arrays, linked lists, stacks and queues ?

Definition

- Unlike linear data structures (linked list, stack, queue), a tree is a hierarchical, non-linear data structure composed of one or more nodes (or no node at all).
- Trees are all around us. What does a typical tree look like?
    - Leaves
    - Branches

…definition

- Trees in data structures have their roots on the top and the leaves below. They are upside-down trees.



Tree Terminologies

Node

- A node is a structure that may contain a value or a condition or a separate data structure like an array, linked list, or even a tree. Each node of the tree can have zero or more children, but it can have only one parent, and if the node is a root node, it will not have any parent.

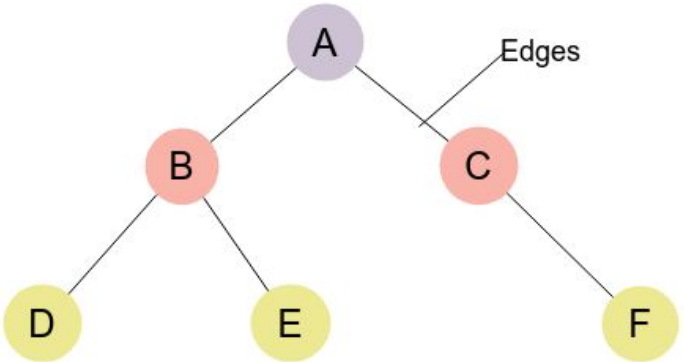
```
class Node:

    def __init__(self, key):

        self.left = None

        self.right = None

        self.val = key
```

Root Node(Root)

- A root is the first node of the tree, or we can say the node that does not have any parent. If the root node is connected to any other node, then that node is called the immediate child of the root node.

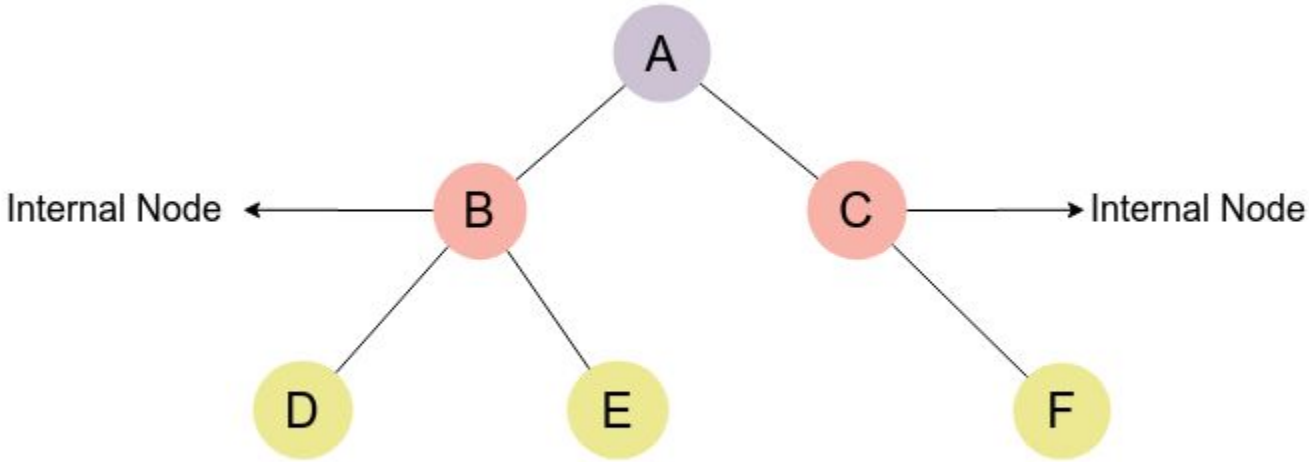
Edge

- An Edge acts as a link between the parent node and the child node.



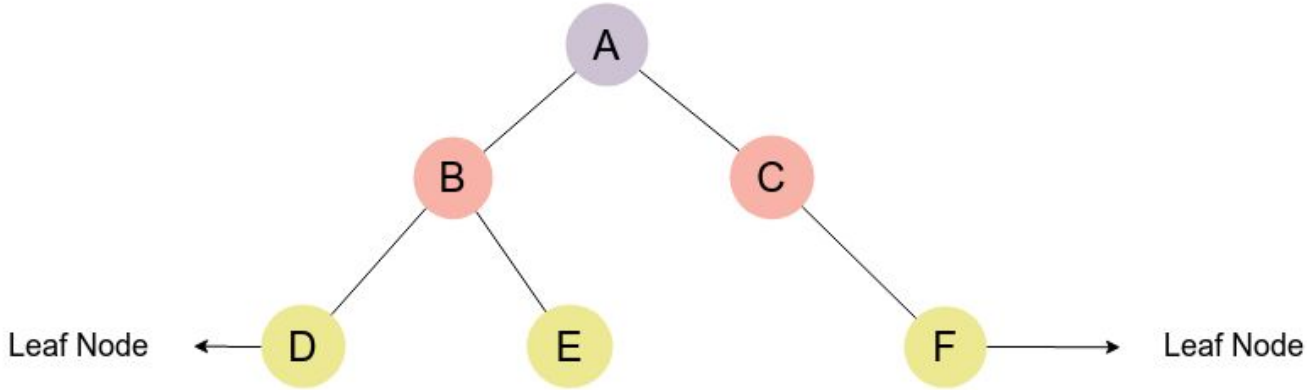
Internal Node or Inner Nodes

- Each node with a parent and a child is called the inner node or internal node of the tree.



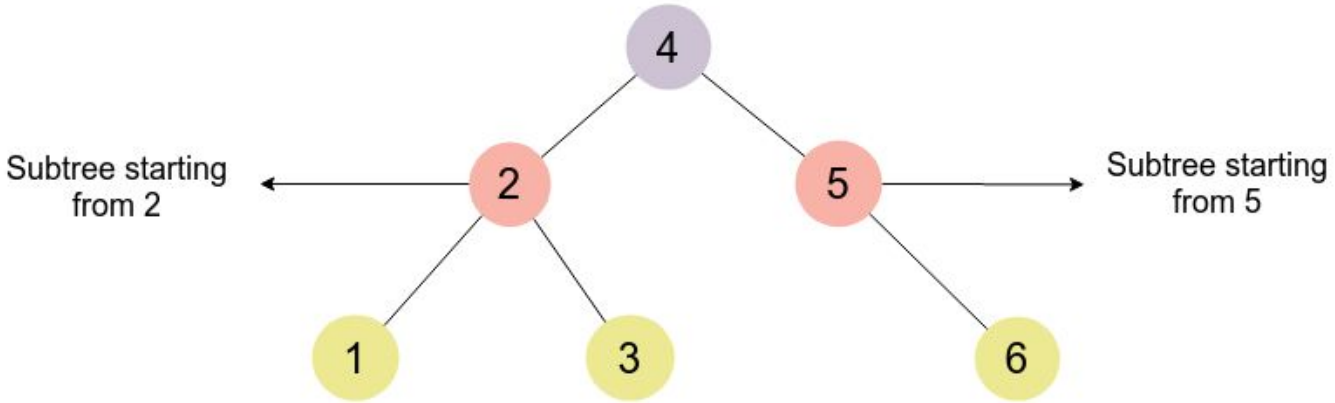
Leaf Node

- Nodes that do not have any further child nodes are called leaf nodes of the tree. A root node that does not have any children can also be called the only leaf node of the tree.



Subtree of a Tree

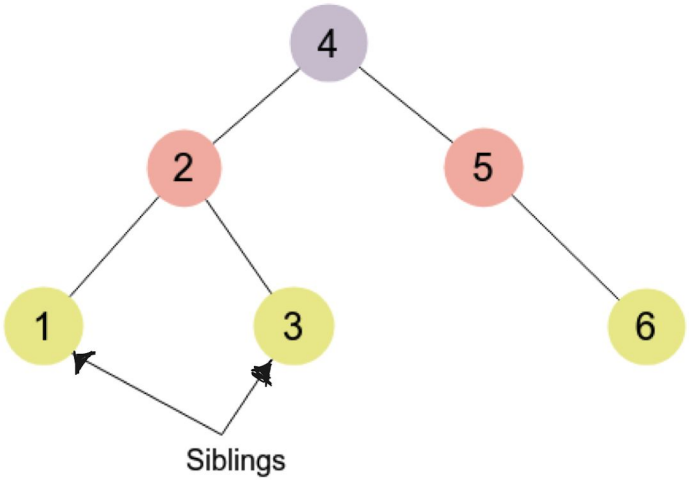
- Each node of a tree forms a recursive subtree of the tree. A subtree of a tree consists of a node **n** and all of the descendants of node **n**.



- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node.
- **An ancestor** of a node is any other node on the path from the node to the root.
- **A descendant** is the inverse relationship of ancestor: A node p is a descendant of a node q if and only if q is an ancestor of p.

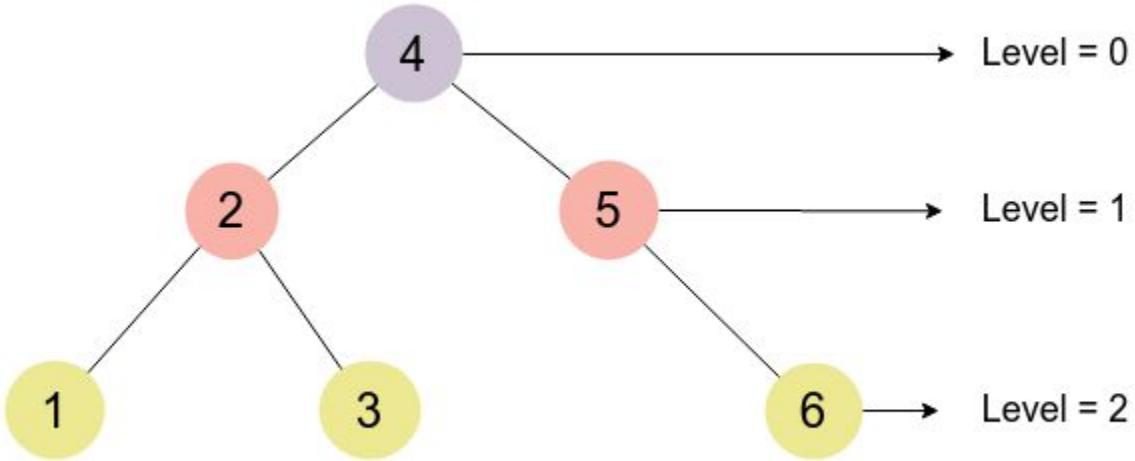
Siblings

- In a tree, two or more nodes that share the same parent node are called sibling nodes.



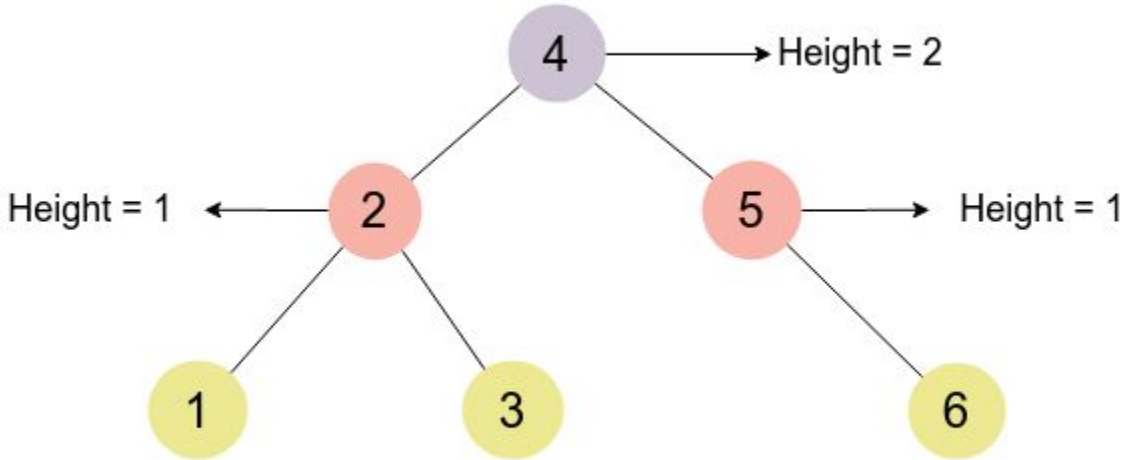
Level of a Tree

- Level of the tree indicates how far you have gone from the root node. The level of the tree starts with 0 and increments by one as you are going from top to bottom.



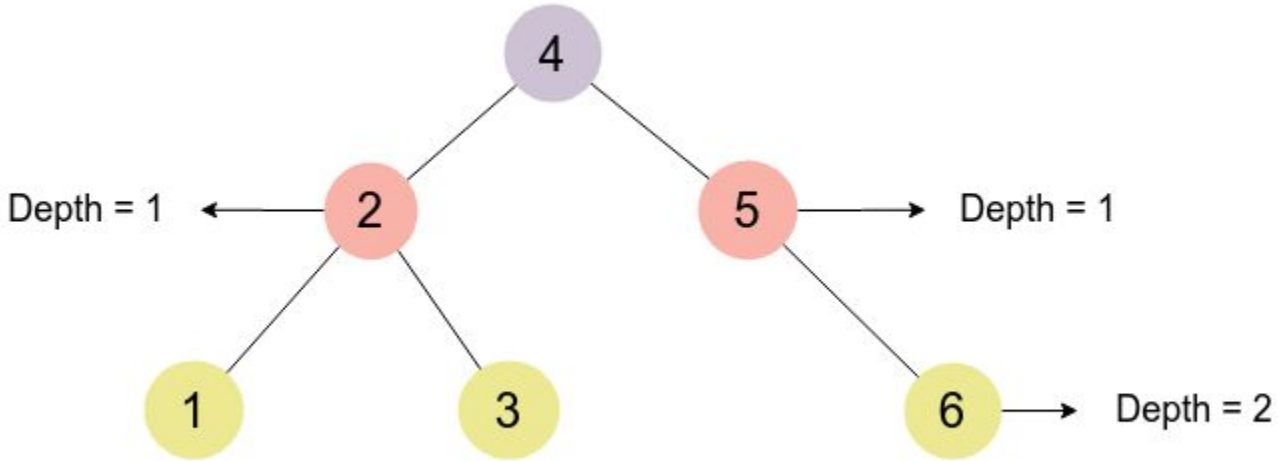
Height of a Tree

- The tree's height is the total number of edges from the root node to the farthest leaf node of the tree.

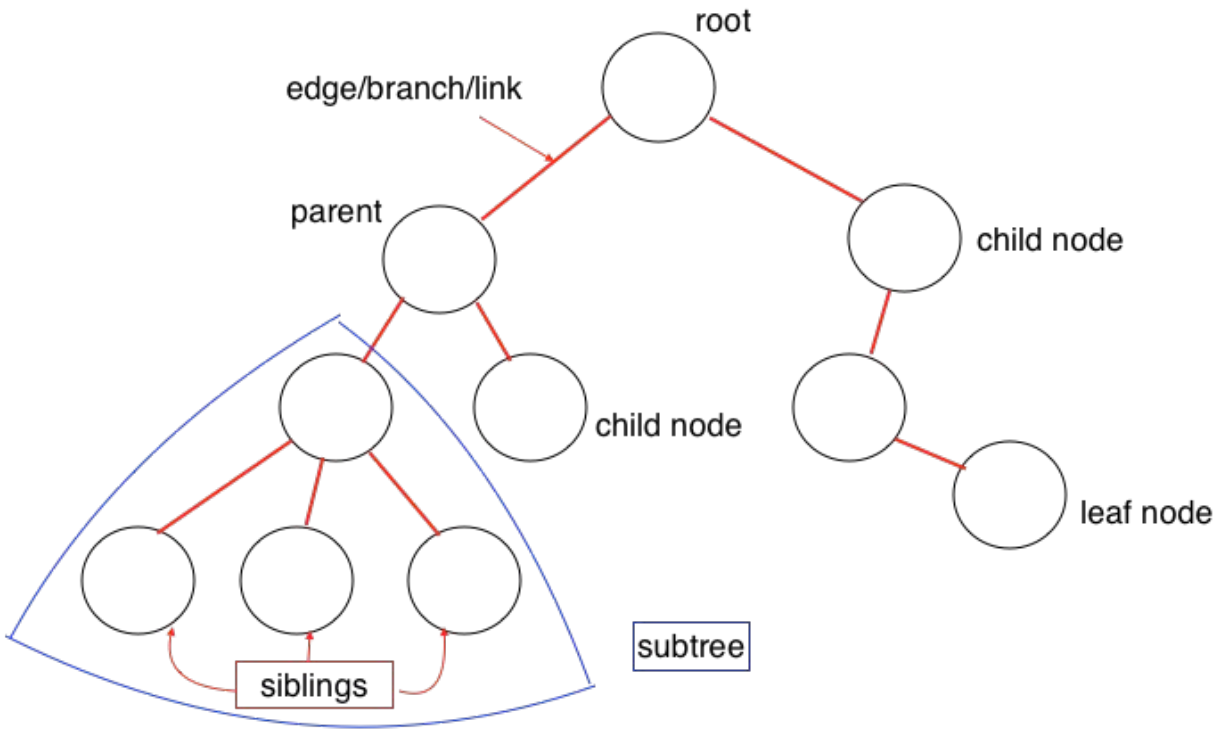


Depth of a Tree

- The depth of the tree is similar to the height; the only difference is that it goes backward. It means the depth of the node is the total number of edges from the root node to the current node. It is very similar to the level of the node.



Trees Terminologies Summary



Types of Trees

- In data structures, there are various types of trees. The selection of trees depends upon the nature of the problems we are trying to solve. These are the basic types of trees:
    - Binary Tree

        Tree with only two children
    - Ternary Tree

        Tree with only three children
    - N-ary Tree

        Tree with n children

Node Implementation of Binary and Nary Tree

Binary Tree

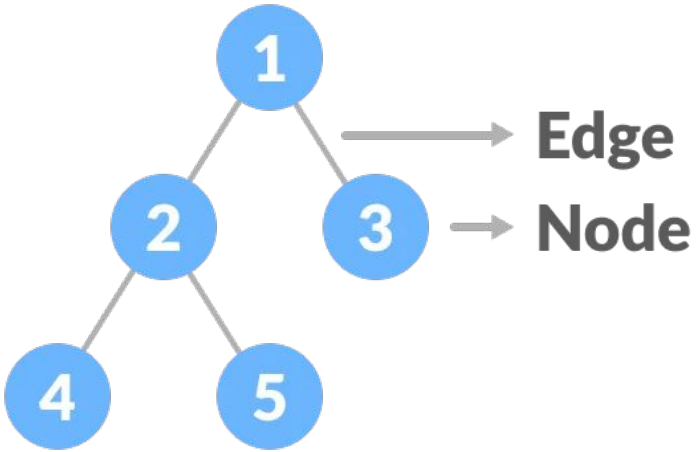
```
class Node:

    def __init__(self, key):

        self.left = None

        self.right = None

        self.val = key
```

N-ary Tree

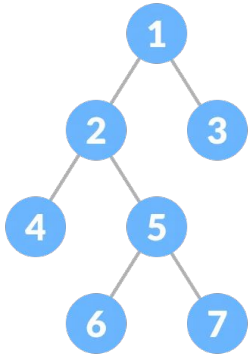
```
class Node:

    def __init__(self, val=None, children=[]):

        self.val = val

        self.children = children
```

Binary Tree

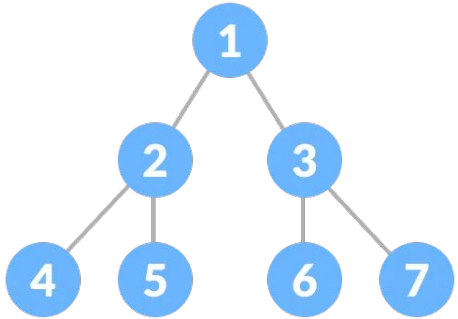
- A binary tree is a tree in which every internal node and root node has at most two children. These two child nodes are often called the left child node or right child node of the node.



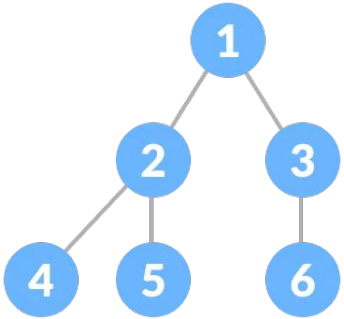
Types of Binary Tree



Full Binary Tree

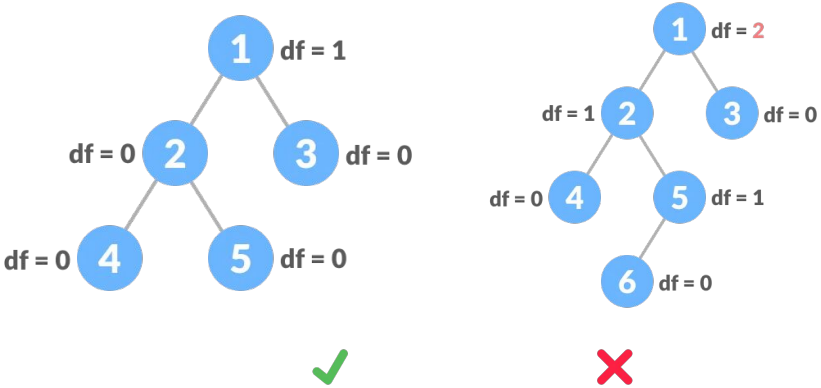


Perfect Binary Tree



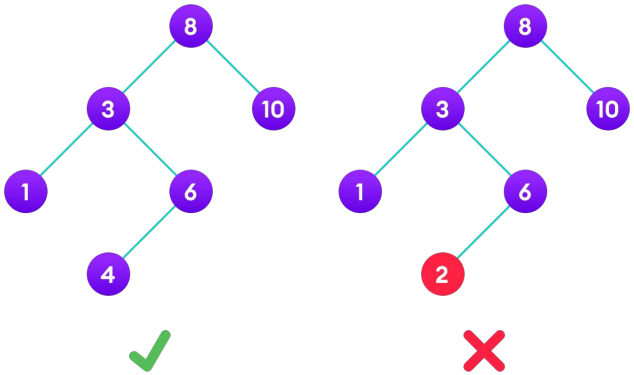
Complete Binary Tree

Types of Binary Tree



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

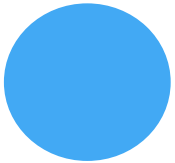
Balanced Binary Tree

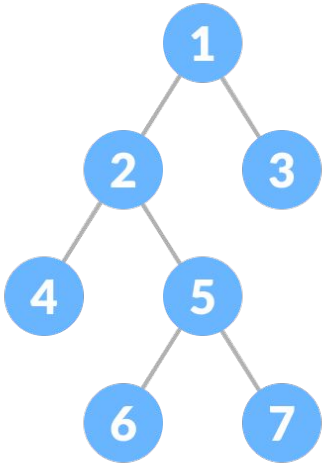


Binary Search Tree

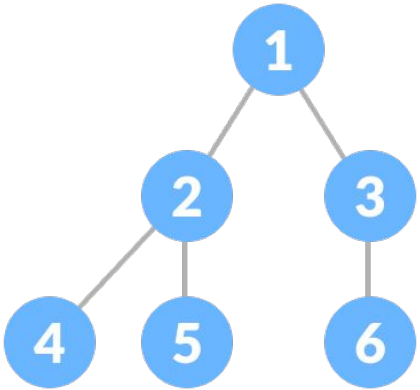
Full Binary Tree

- A Full Binary Tree is a binary tree in which every node has either 0 or 2 descendants. In other words, a full binary tree is a binary tree in which every node has two offspring, with the exception of the leaf nodes. It is also known as a proper binary tree.





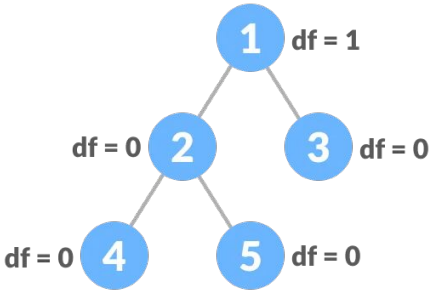
Complete Binary Tree

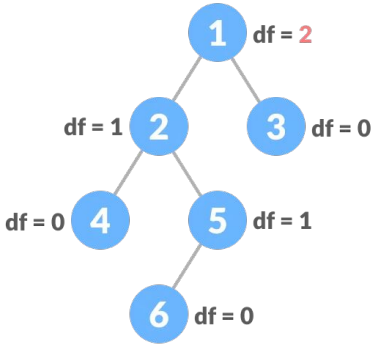


- A complete binary tree is a special type of binary tree where all the levels of the tree are filled completely except the lowest level nodes which are filled from as left as possible.
- A complete binary tree is just like a full binary tree, but with two major differences
  - All the leaf elements must lean towards the left.
  - 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.

Balanced Binary Tree

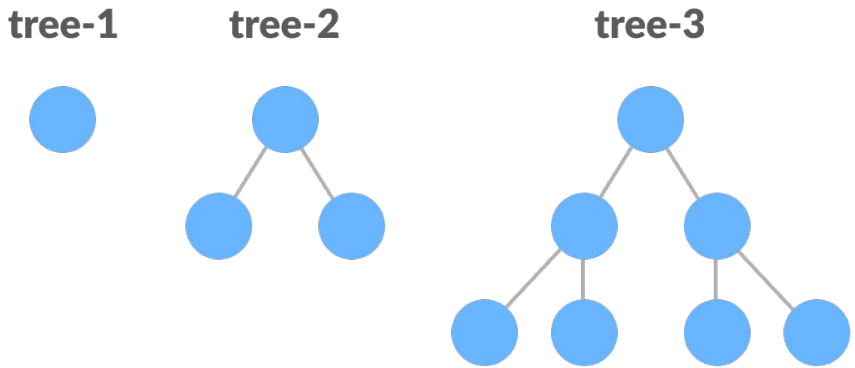
- A balanced binary tree is a binary tree that follows the 3 conditions: The height of the left and right tree for any node does not differ by more than 1. The left subtree of that node is also balanced. The right subtree of that node is also balanced.





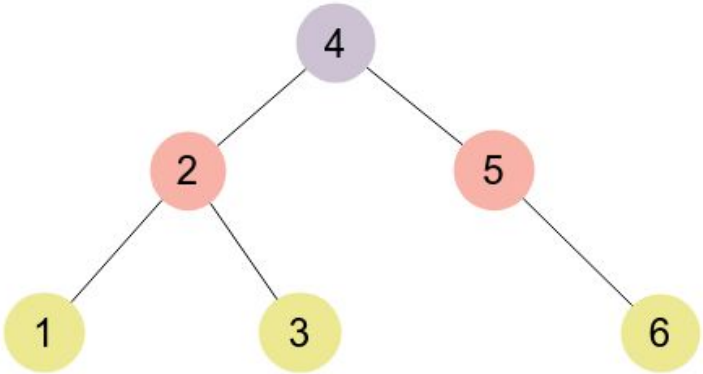
Perfect Binary Tree

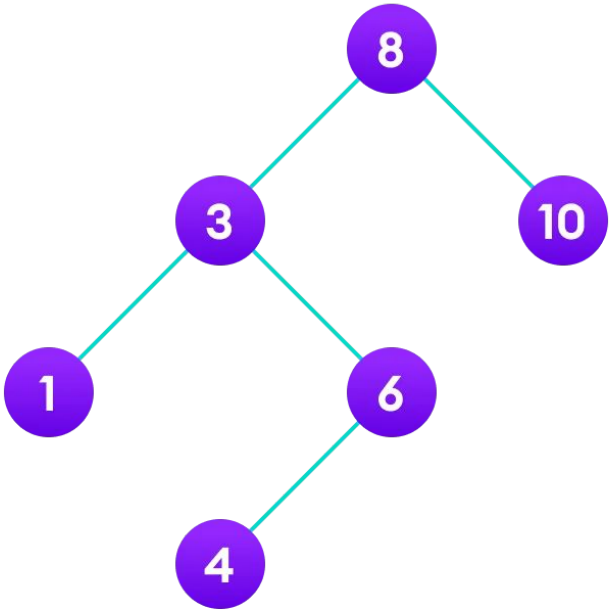
- A perfect binary tree is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In Simple terms
  - All leaf nodes are at the maximum depth of the tree
  - The tree is completely filled with no gaps.

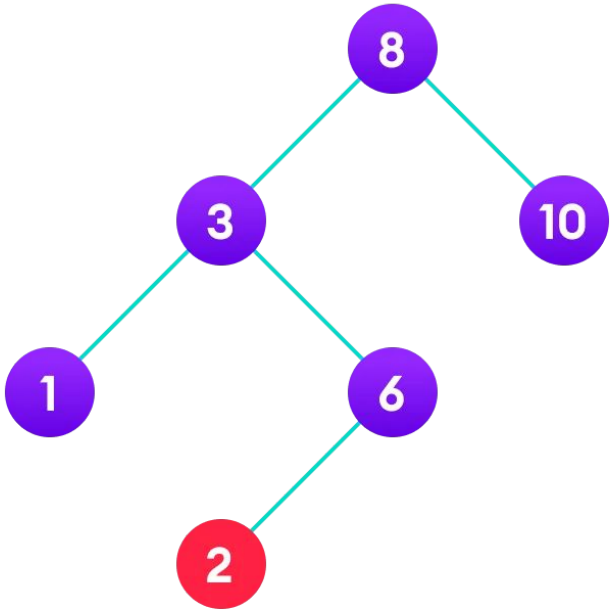


Binary Search Tree(BST)

- A binary search tree is a binary tree that has the following properties:
  - The left subtree of the node only contains values less than the value of the node.
  - The right subtree of the node only contains values greater than or equal to the value of the node.
  - The left and right subtree of the nodes should also be the binary search tree.




8
3
10
1
6
4


8
3
10
1
6
2

BST Continued

- 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

But Why BST?

Efficiently search, insert and delete of nodes from binary tree.

Applications

- Sorting large datasets
- Maintaining sorted stream of data.
- Used to implement Dictionaries and Priority queues.

Problem Pattern

BST problems usually state that the given tree is a binary search tree, which makes BST questions easy to identify.

Approach Pattern

Based on BST properties, we can greedily choose which half of the tree we can continue operation after every check.

Tree Traversal

Binary Tree Traversal

Tree Traversal

- 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 lists have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.

Tree Traversal

- If you implement a tree, you need a way to perform certain operations like finding the node's depth, height of the node, order each node of the tree (from left to right), etc.

- All these operations can be performed easily with the help of tree traversal algorithms.

Binary Tree Traversal

- There are basically three ways of traversing a binary tree:
    1. **Inorder Traversal**
    2. **Preorder Traversal**
    3. **Postorder Traversal**
- These traversal techniques are not only limited to the binary tree; they can easily be modified for an n-ary tree (a tree with a maximum of n children).

Preorder Traversal (Root, Left, Right)

- In preorder traversal, we traverse the root node first, then the left subtree of the binary tree, and finally, the node's right subtree.

```
# A function to do preorder tree traversal

def printPreOrder(root):

     if root:

          # First print the data of node

          print(root.val),

          # Then recur on left child

          printPreOrder(root.left)

          # Finally recur on right child

          printPreOrder(root.right)
```

Inorder Traversal (Left, Root, Right)

- Visit the current node before visiting any nodes inside the left or right subtrees. Here, the traversal is root – left child – right child. It means that the root node is traversed first then its left child and finally the right child.

- In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

```
# A function to do inorder tree
traversal

def printInorder(root):

    if root:

        # First recur on left child

        printInorder(root.left)

        # then print the data of node

        print(root.val),

        # now recur on right child

        printInorder(root.right)
```

Post Order Traversal (Left, Right, Root)

- In postorder traversal, we traverse the left subtree of the node and then the right subtree tree of the node, and finally, the root node of the tree.

```
# A function to do postorder tree traversal

def printPostOrder(root):

    if root:

        # First recur on left child

        printPostOrder(root.left)

        # the recur on right child

        printPostOrder(root.right)

        # now print the data of node

        print(root.val)
```

Binary Tree Traversal : Example

1. Inorder traversal

   Step 1: Visit all nodes in the left subtree

   Step 2: Visit the root node

   Step 3: Visit all nodes in the right subtree

2. Preorder traversal

   Step 1: Visit the root node

   Step 2: Visit all nodes in the left subtree

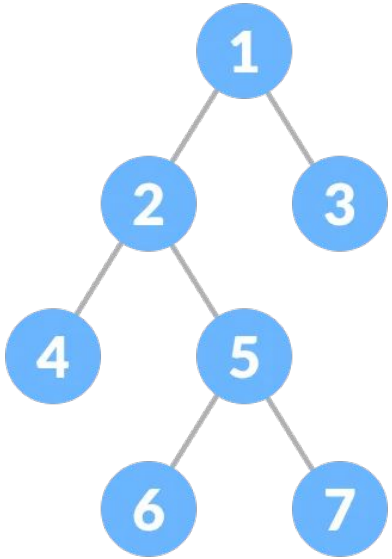
   Step 3: Visit all nodes in the right subtree

3. Postorder traversal

   Step 1: Visit all nodes in the left subtree

   Step 2: Visit all nodes in the right subtree

   Step 3: Visit the root node



Binary Tree Traversal : Example

1. Inorder traversal

Step 1: Visit all nodes in the left subtree

Step 2: Visit the root node

Step 3: Visit all nodes in the right subtree

4 2 6 5 7 1 3

2. Preorder traversal

Step 1: Visit the root node

Step 2: Visit all nodes in the left subtree

Step 3: Visit all nodes in the right subtree

1 2 4 5 6 7 3

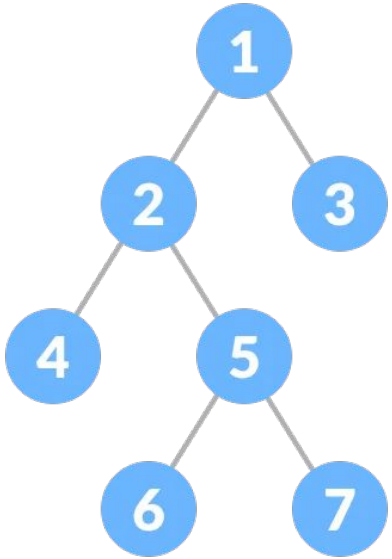
3. Postorder traversal

Step 1: Visit all nodes in the left subtree

Step 2: Visit all nodes in the right subtree

Step 3: Visit the root node

4 6 7 5 2 3 1



Checkpoint

Link

Sample Question

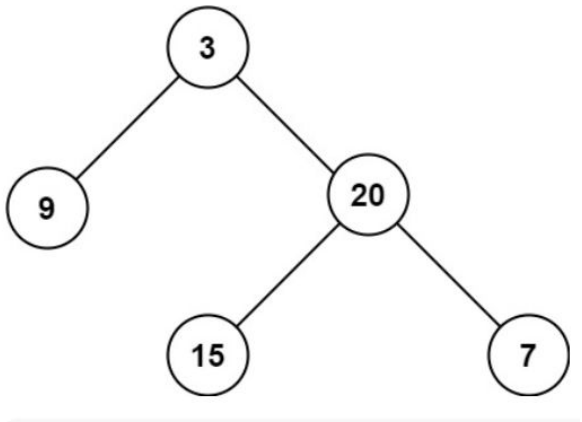
Link

Question

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:



Example 1:
Input: root = [3,9,20,null,null,15,7]
Output: 3

Example 2:
Input: root = [1,null,2]
Output: 2

Solution

- What if the tree is empty?

  Answer: max_depth = 0

- What if we have just a node with no children?

  Answer: max_depth = 1

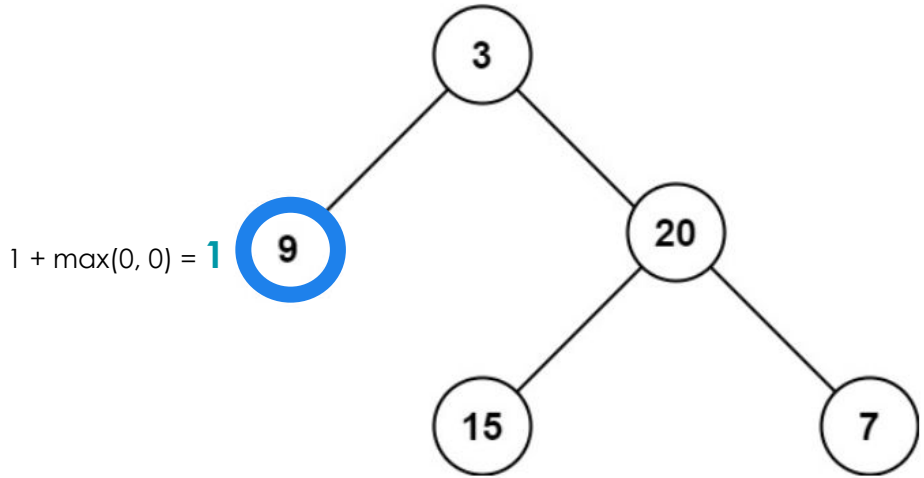
- What if the current node has left and right children?

  Answer: max_depth = 1 + max(left_child_max_depth, right_child_max_depth)

  By recursively calculating the max_depth of each subtree

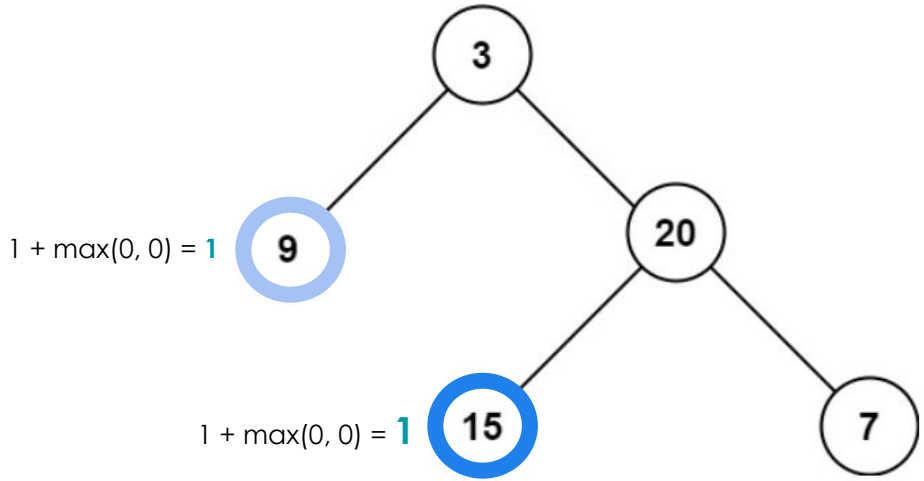
Step by Step Simulation (Recursive)

Step 1:



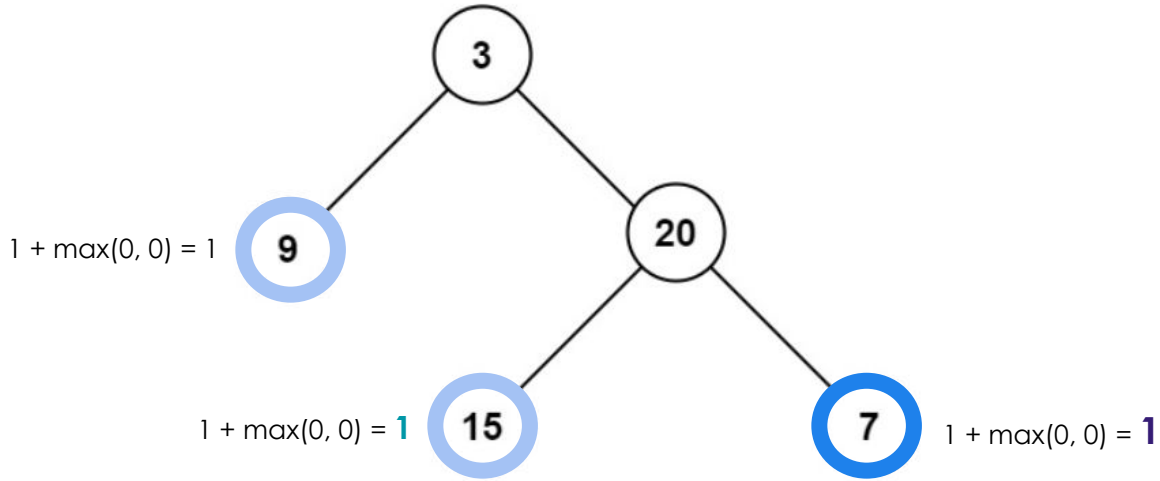
Step by Step Simulation (Recursive)

Step 2:



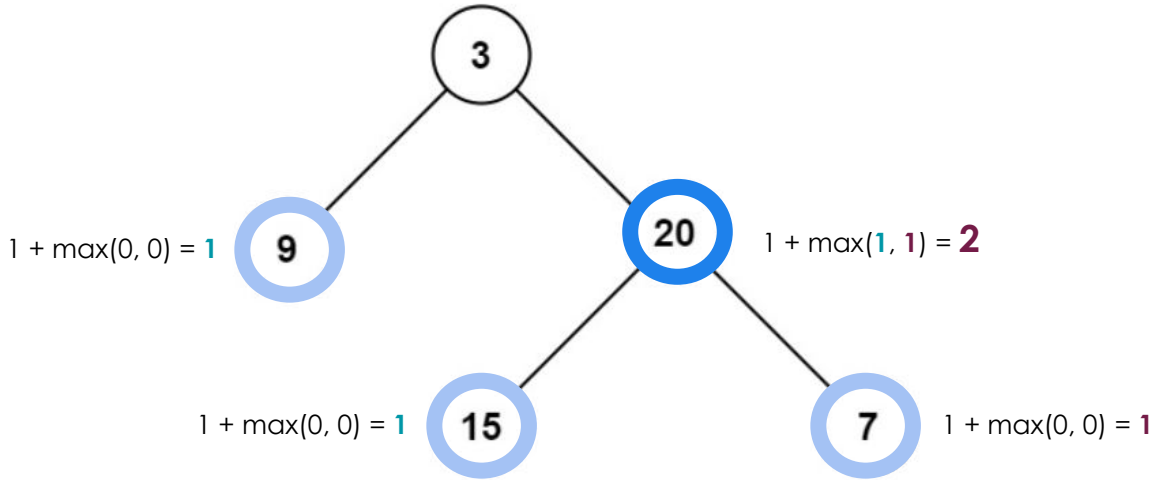
Step by Step Simulation (Recursive)

Step 3:



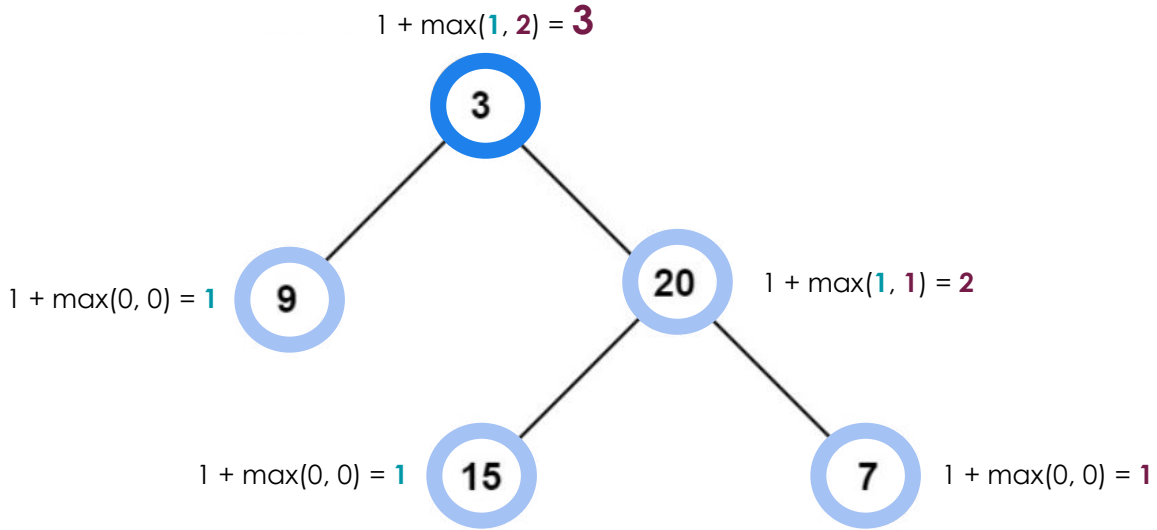
Step by Step Simulation (Recursive)

Step 4:



Step by Step Simulation (Recursive)

Step 5:



Therefore, max_depth = 3

Implementation (Recursive)

```
# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution(object):
   def maxDepth(self, root):
       """
       :type root: TreeNode
       :rtype: int
       """
       def find_max(node):
           if not node : return 0
           left = 1 + find_max(node.left)
           right = 1 + find_max(node.right)

           return max(left,right)
       return find_max(root)
```

Time complexity: O(n)
Space Complexity: O(n)

Implementation II (Iterative)

```
def maxDepth(self, root):
      """
       :type root: TreeNode
       :rtype: int
       """
       if root is None:
           return 0

       stack = []
       stack.append((root, 1))
       res = 0

       while stack:
           node, depth = stack.pop()
           if node:
               res = max(res, depth)
               stack.append((node.left, depth+1))
               stack.append((node.right, depth+1))
       return res
```

Time complexity: O(n)
Space Complexity: O(n)

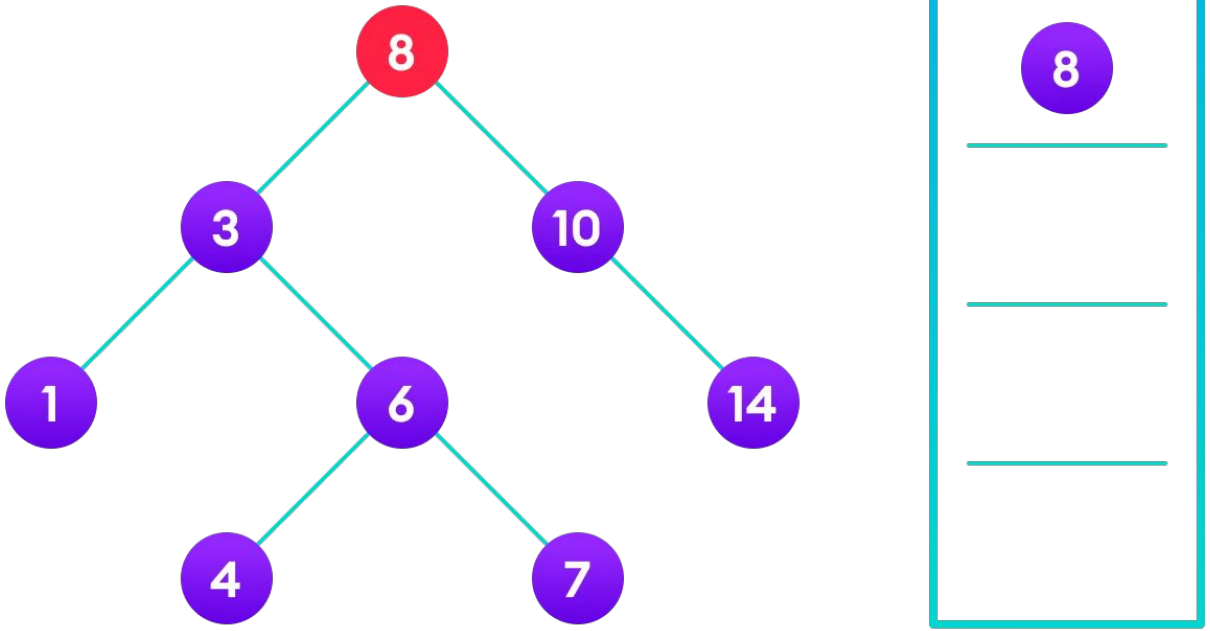
Basic Operations on Trees(BST)

Search Operation

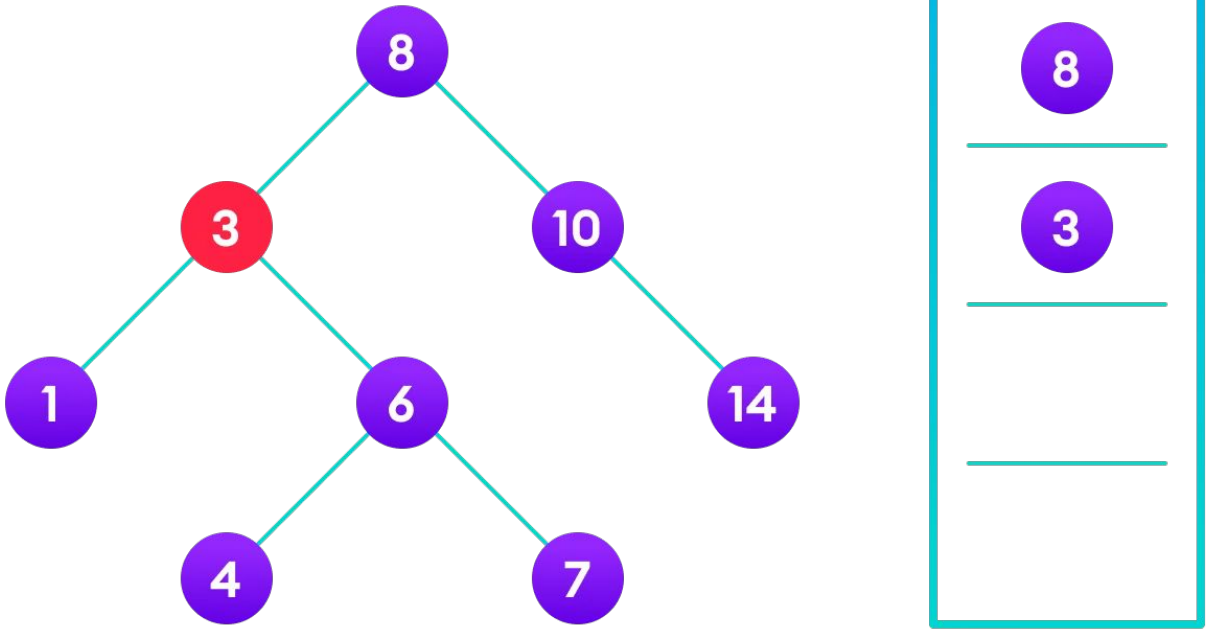
- The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.
- If the value is **below** the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree
- If the value 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.

  Let us try to visualize this with a diagram searching for 4 in the tree:

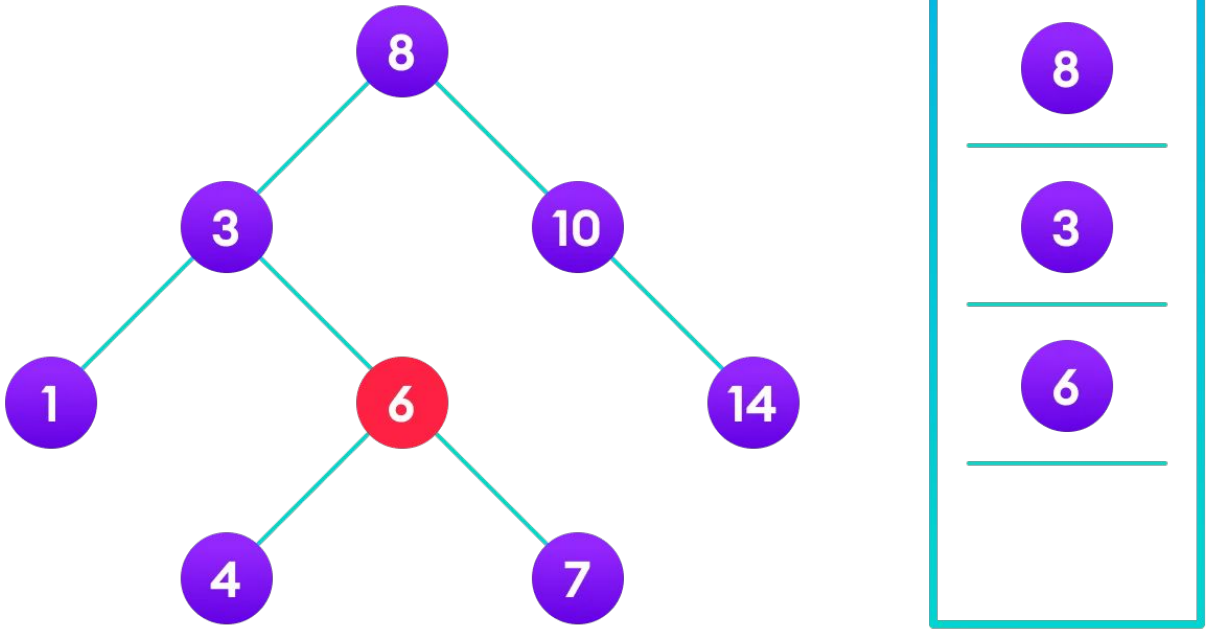
4 is not found so, traverse through the left subtree of 8



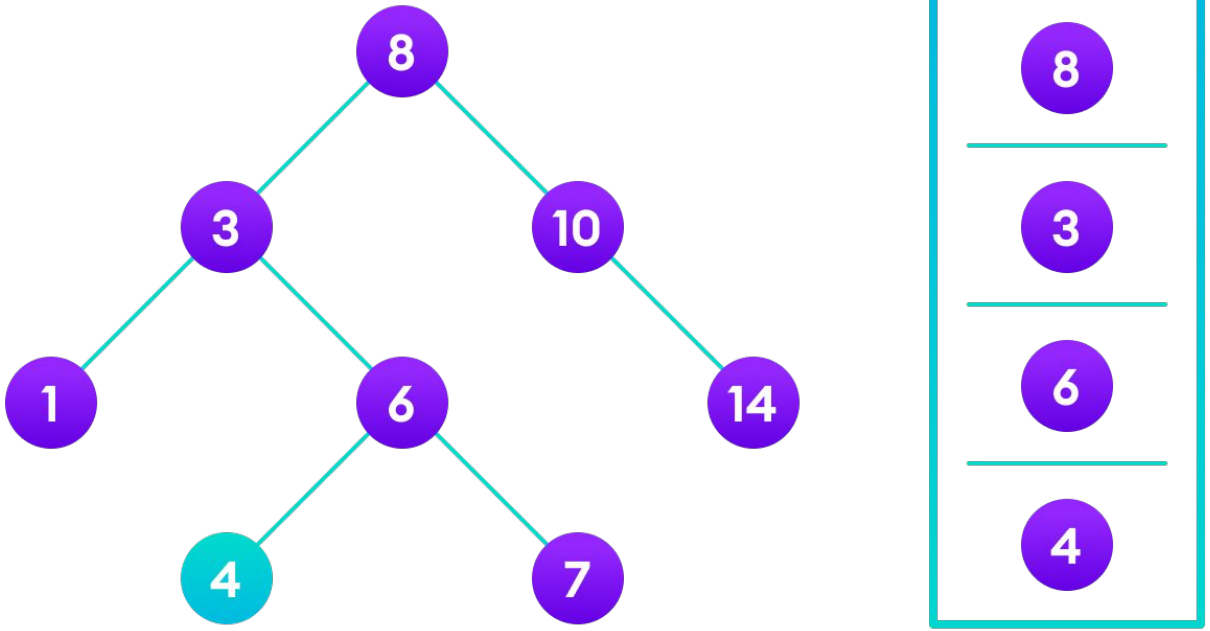
4 is not found so, traverse through the right subtree of 3



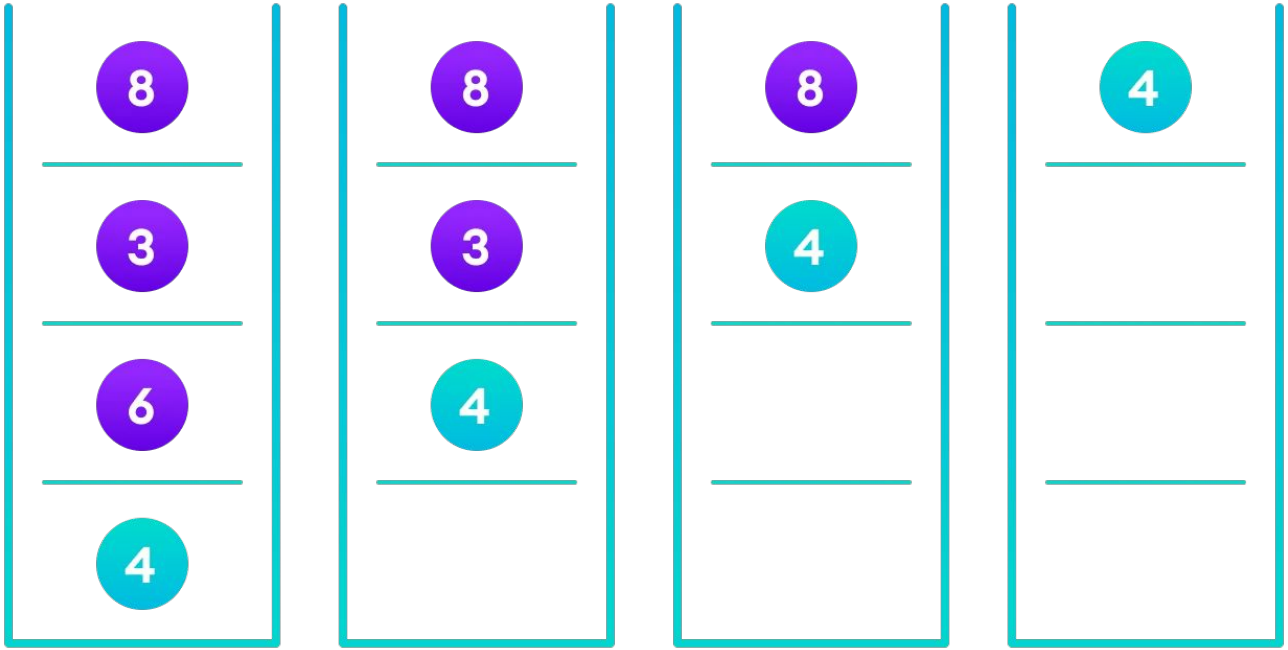
4 is not found so, traverse through the left subtree of 6



4 is found



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



Algorithm

```
def search(root):
        if root == None :
            return None
        if number == root.val :
            return root.val
        if number < root.val :
            return search(root.left)
        if number > root.val :
              return search(root.right)
```

Checkpoint

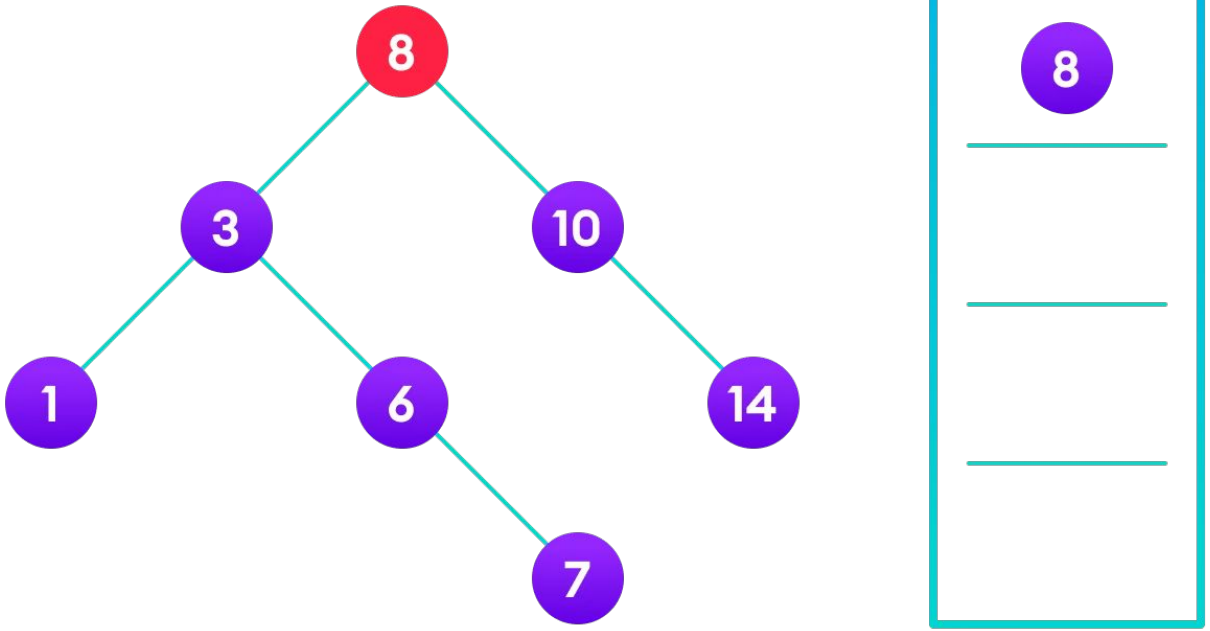
Search in a Binary Search Tree

Inserting Operation

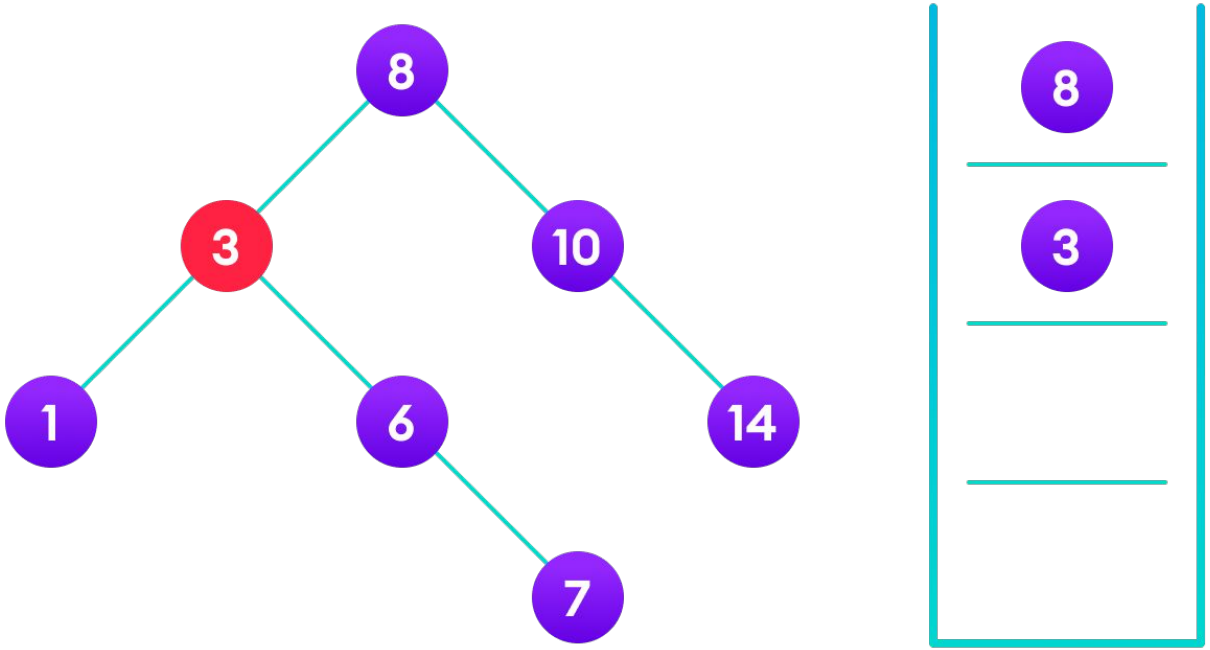
- Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.
- We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

  Let's try to visualize how we add a number 4 to an existing BST.

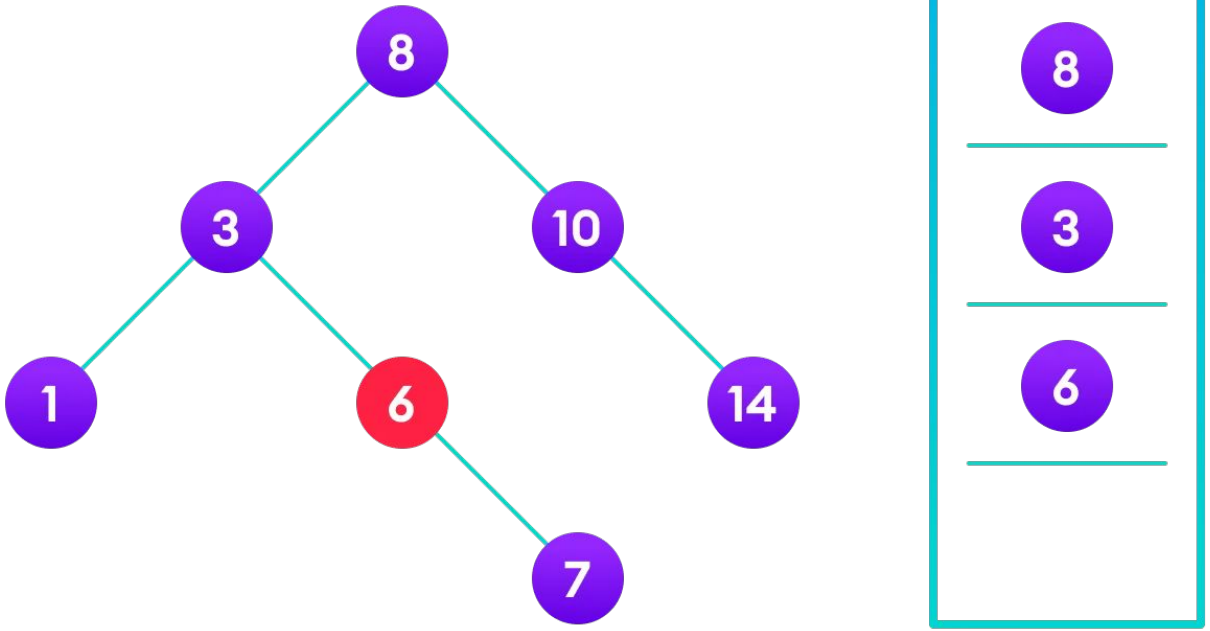
4<8 so, transverse through the left child of 8



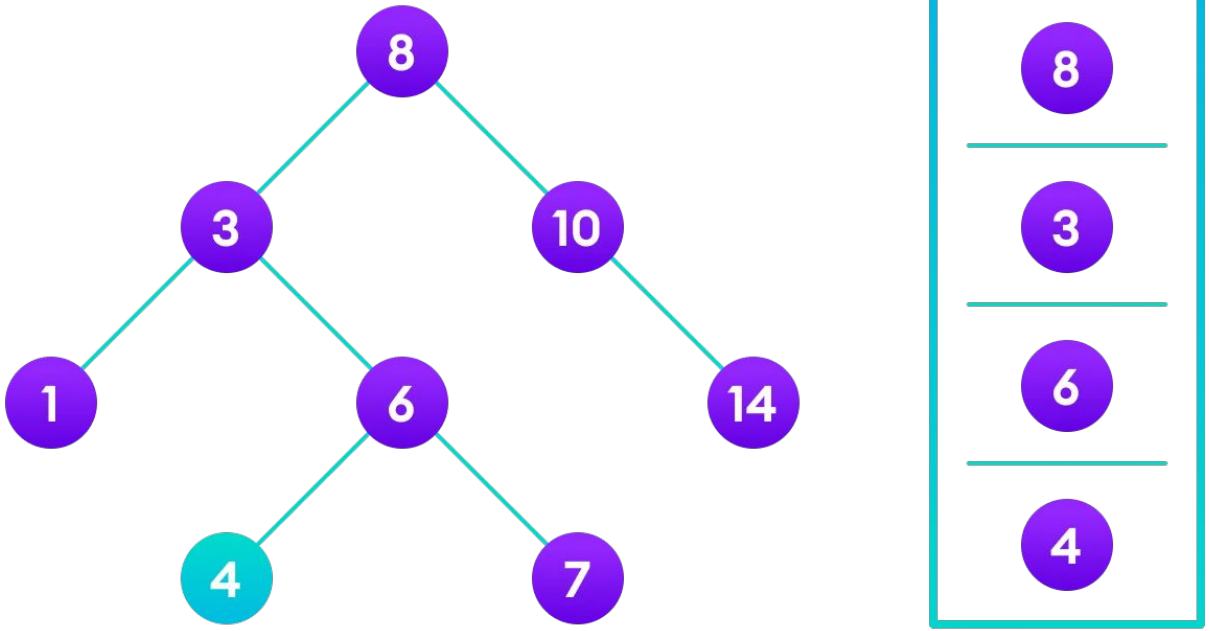
4>3 so, transverse through the right child of 8

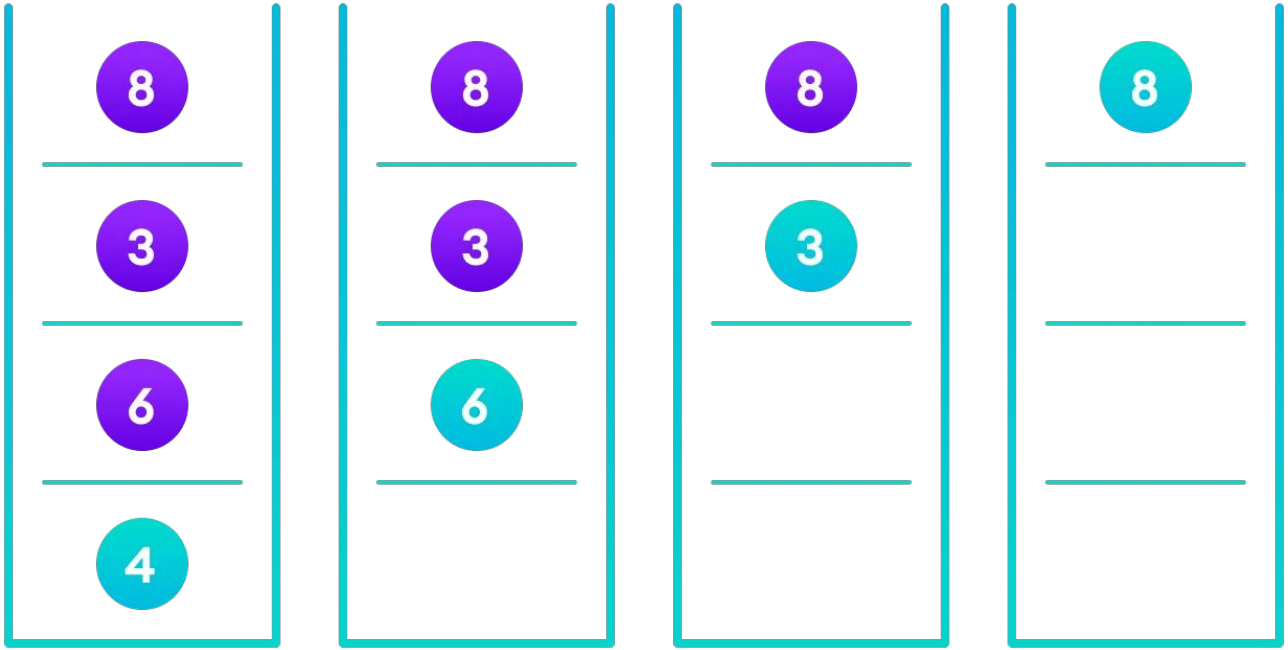


4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6




8
3
6
4
8
3
6
8
3
8

Algorithm

```
def insert(node,data):
    if node == None:
        return createNode(data) ## assume createNode creates a new node to be returned
    if data < node.data:
        node.left  = insert(node.left, data);
    elif data > node.data:
        node.right = insert(node.right, data);
    return node
```

Checkpoint

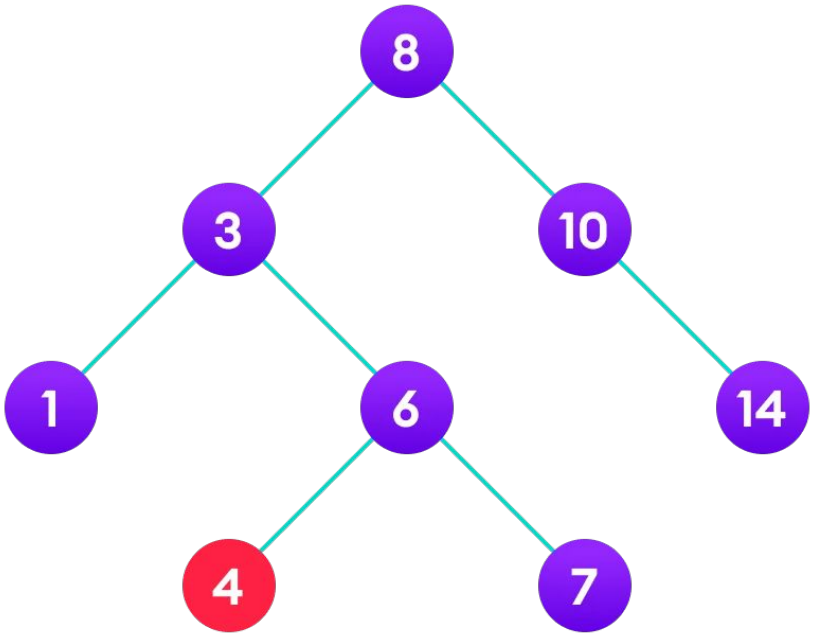
Insert into a Binary Search Tree

Deletion Operation

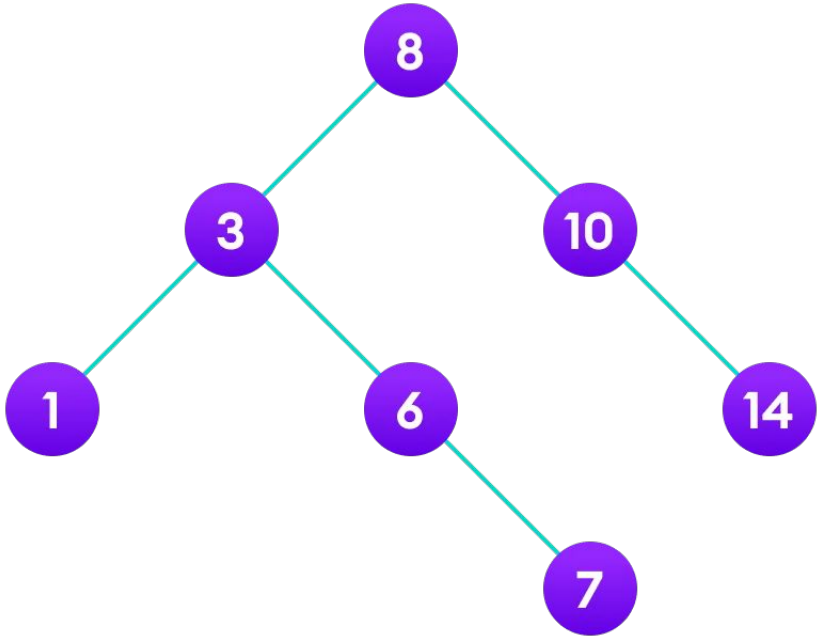
- There are three cases for deleting a node from a binary search tree.

Case One: In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

4 is to be deleted



Delete the node

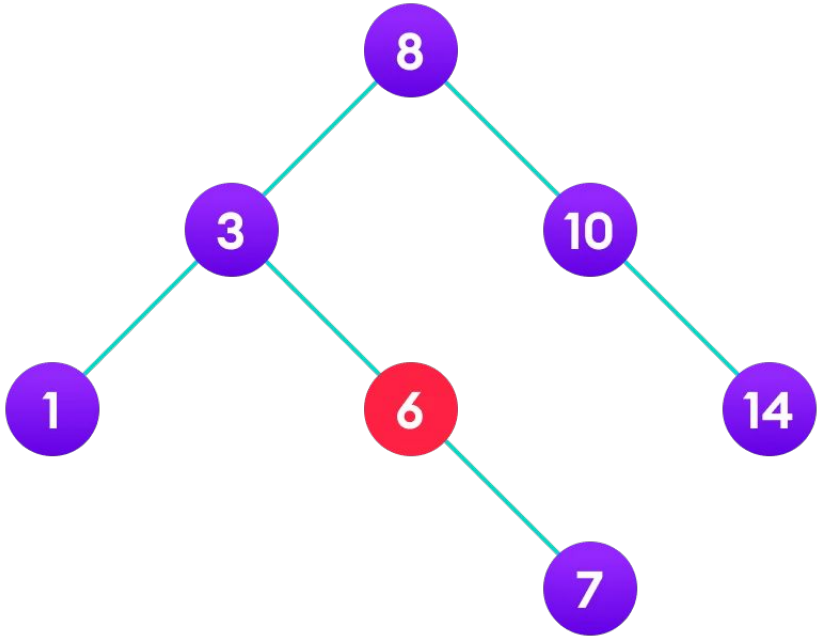


Case Two

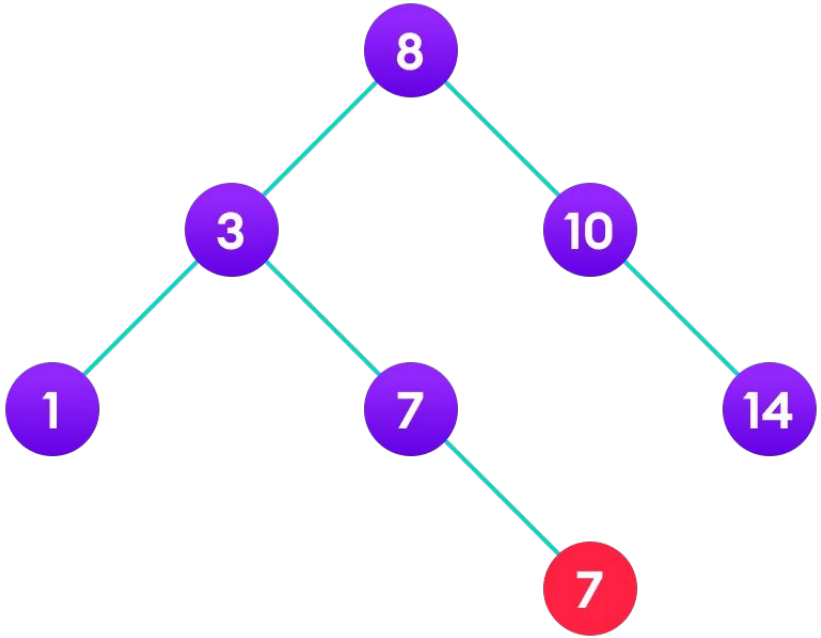
In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:

1. Replace that node with its child node.
2. Remove the child node from its original position.

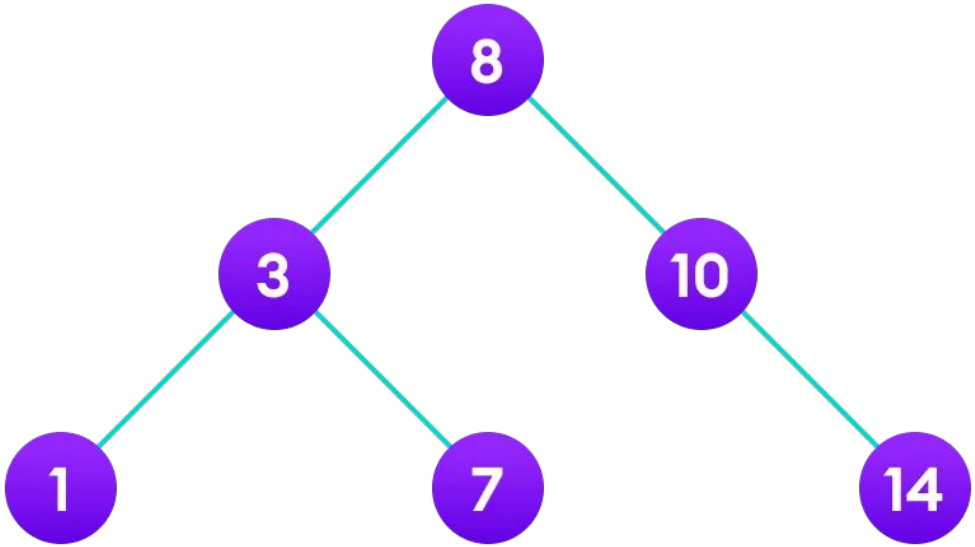
6 is to be deleted



copy the value of its child to the node and delete the child



Final tree

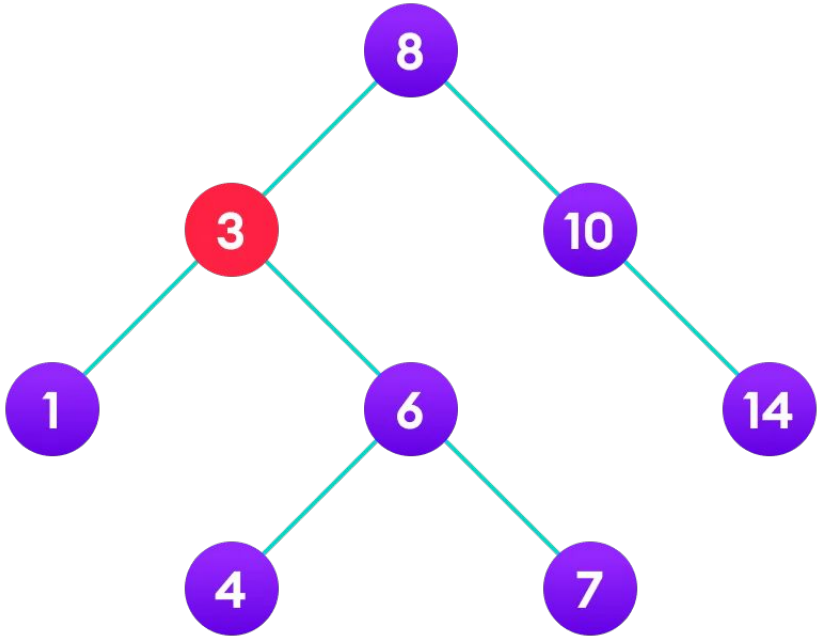


Case Three

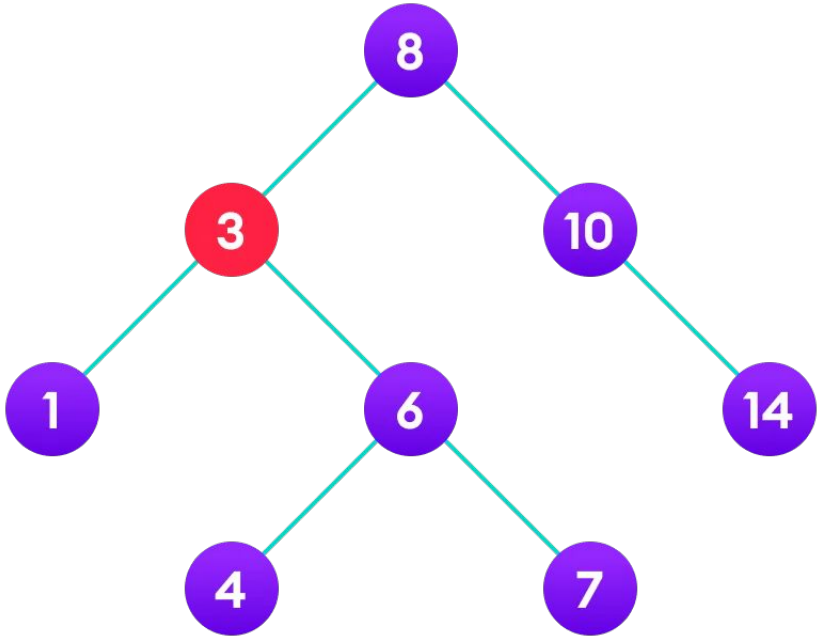
In the third case, the node to be deleted has two children. In such a case follow the steps below:

1. Get the inorder successor of that node.
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.

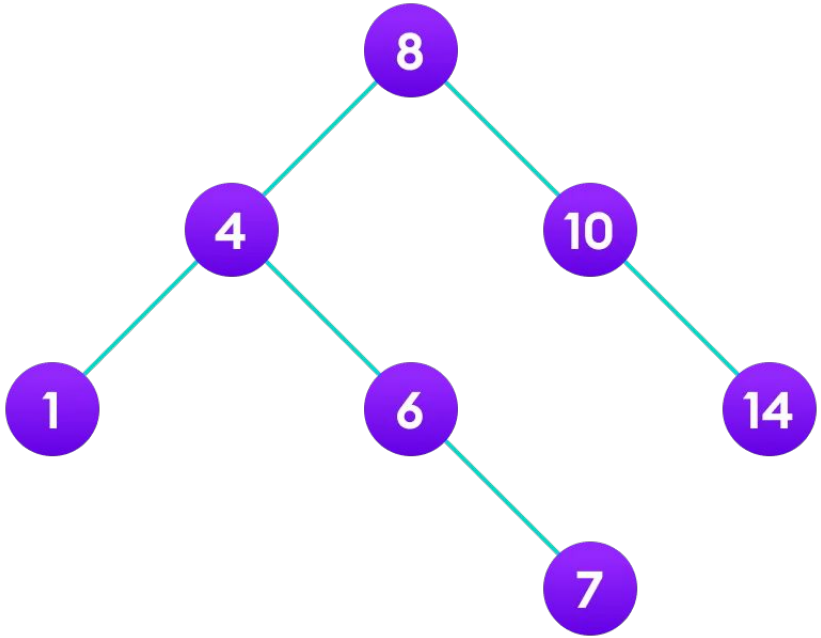
3 is to be deleted



Copy the value of the inorder successor (4) to the node



Delete the inorder successor



```
# Given a binary search tree and a key, this function
# delete the key and returns the new root

def deleteNode(root, key):
    # Return if the tree is empty
    if not root:
        return root
    # Find the node to be deleted
    if key < root.key:
        root.left = deleteNode(root.left, key)
    elif(key > root.key):
        root.right = deleteNode(root.right, key)
    # If key is same as root's key, then this is the node to be deleted
    else:
        # If the node is with only one child or no child
        if not root.left:
            return root.right
        elif not root.right:
            return root.left

        # If the node has two children,
        # get the inorder successor of the node to be deleted
        temp = minValueNode(root.right)

       # Copy the inorder successor's content to this node
        root.key = temp.key
        # Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)

    return root
```

```
# Find the inorder successor

 def minValueNode(node):
     current = node

     # Find the leftmost leaf
     while(current.left):
         current = current.left

     return current
```

CheckPoint

Delete Node in a BST

Time and Space Complexity Analysis

Binary Tree

- Traversing
  - Time = ?
- Searching
  - Time = ?
- Insertion
  - Time = ?
- Deletion
  - Time = ?
- Space = ?

Binary Search Tree

- Traversing
  - Time = ?
- Searching
  - Time = ?
- Insertion
  - Time = ?
- Deletion
  - Time = ?
- Space = ?

Time and Space Complexity Analysis

Binary Tree

- Traversing
  - Worst Time = **O(N)**
  - Best Time = **O(h)**
- Searching
  - Worst Time = **O(N)**
  - Best Time = **O(1)**
- Insertion
  - Worst Time = **O(N)**
  - Best Time = **O(h)**
- Deletion
  - Worst Time = **O(N)**
  - Best Time = **O(h)**
- Space = O(N) why?

Binary Search Tree

- Traversing
  - Worst Time = **O(N)**
  - Best Time = **O(h)**
- Searching
  - Worst Time = **O(N)**
  - Average Time = **O(h)**
  - Best Time = **O(1)**
- Insertion
  - Worst Time = **O(N)**
  - Average Time = **O(h)**
- Deletion
  - Worst Time = **O(N)**
  - Average Time = **O(h)**
- Space = O(N) why?

where N is the number of elements in the Binary Search Tree
h = log N for a balanced binary tree

Common Pitfalls

- node <> check existence of node
- Null pointer
    - node.right <> check existence of node
    - node.left <> check existence of node

Applications of a Tree

1. Representation structure in **File Explorer**. (Folders and Subfolders) uses N-ary Tree.
2. **Auto-suggestions** when you google something using Trie.
3. Used in d**ecision-based machine learning algorithms**.
4. Tree forms the backbone of other complex data structures like heap, priority queue, spanning tree, etc.
5. A binary tree is used in **database indexing** to store and retrieve data in an efficient manner.
6. To implement **Heap** data structure.
7. **Binary Search Trees (BST)** can be used in **sorting algorithms**.

Practice Questions

- Merge Two Binary Trees
- Search in a BST
- Same Tree
- Lowest Common Ancestor in a BST
- Validate Binary Search Tree
- Kth Smallest Element in a BST
- Maximum Sum BST in Binary Tree

Resources

- GeeksForGeeks
- Programiz
- JavaTpoint

Quote

"A journey of a thousand miles begins with a single step."

-Lao Tzu