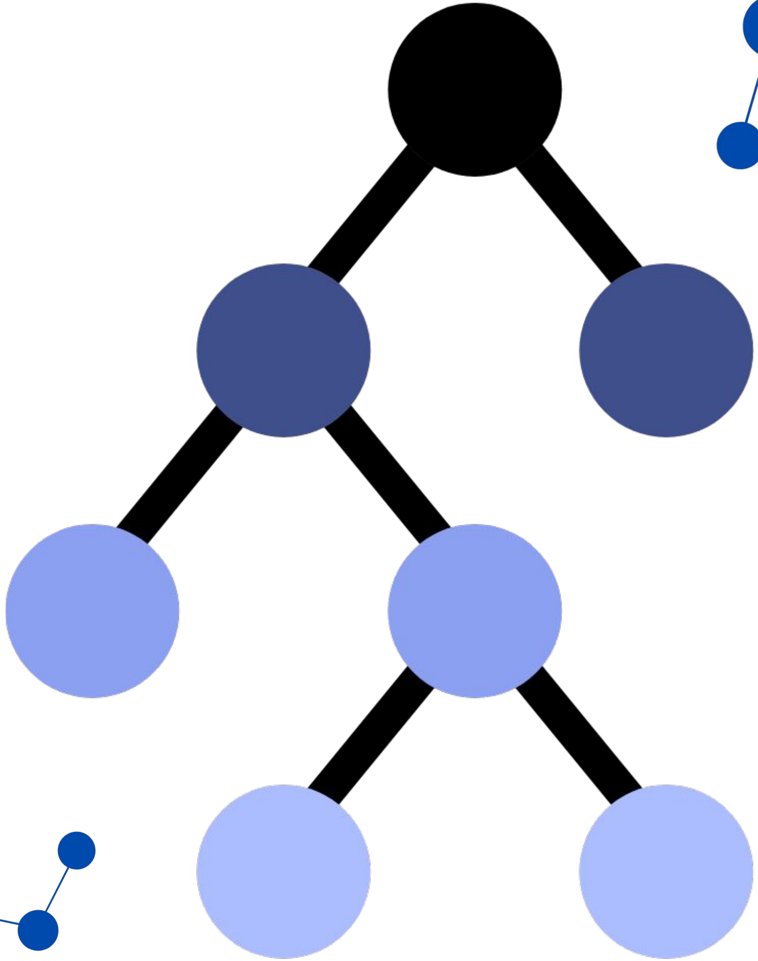
Trees

Binary Trees and Binary Search Trees



Lecture Flow

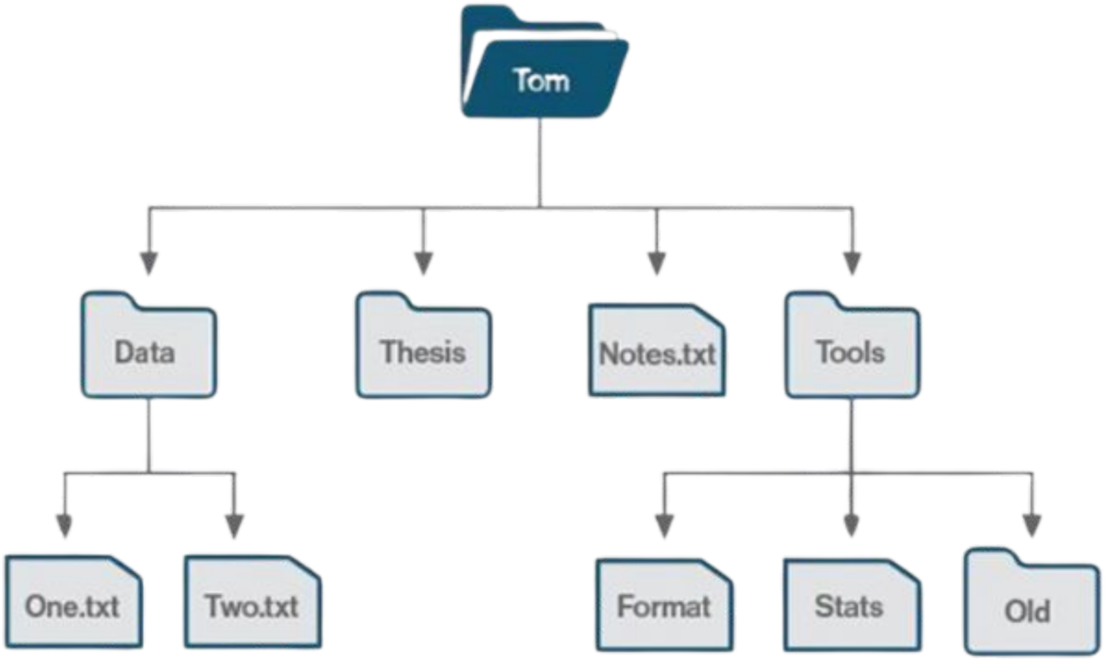
1. Prerequisites
2. Real life problem
3. Definition
4. Tree Terminologies
5. Types of Trees
6. Tree Traversal
7. Checkpoint
8. Basic BST Operations
9. Time and space complexity analysis
10. Things to pay attention to(common pitfalls)
11. Applications of a Tree
12. Practice Questions

Pre-requisites

- Basic Recursion
- Linked List
- Stacks
- Queues

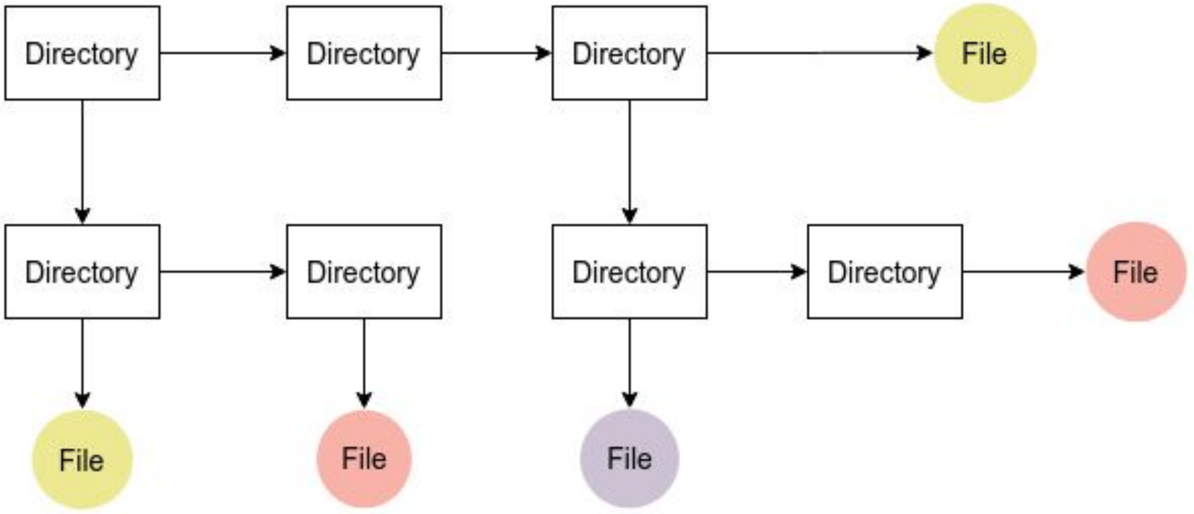
Real-Life Problem

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



Real-Life Problem

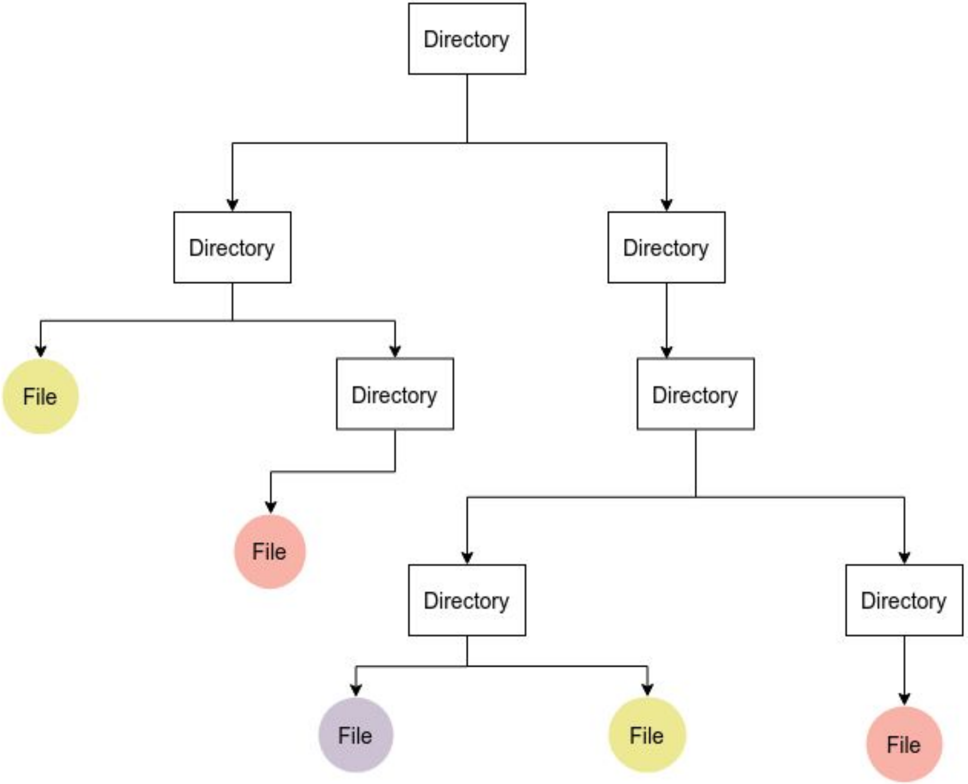
What about linked-lists? Why?



But a directory can contain multiple directories and/or files.

Real-Life Problem

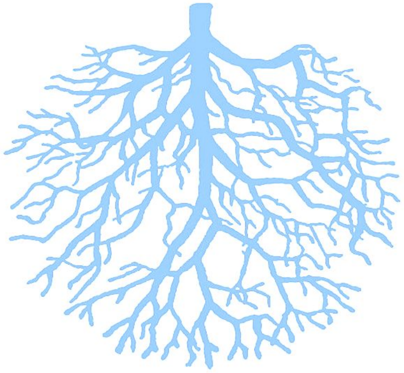
So a linked list with multiple ‘next nodes’?

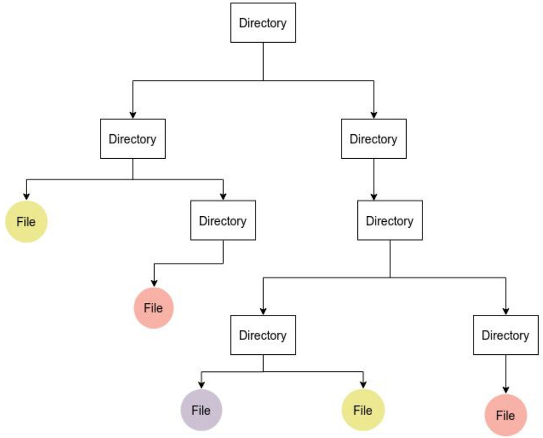


What are Trees?

Definition

- A hierarchical, non-linear data structure composed of zero or more nodes.
- Why the name tree? Such data structure branches out starting from the root, pretty much like the trees around us but upside down.

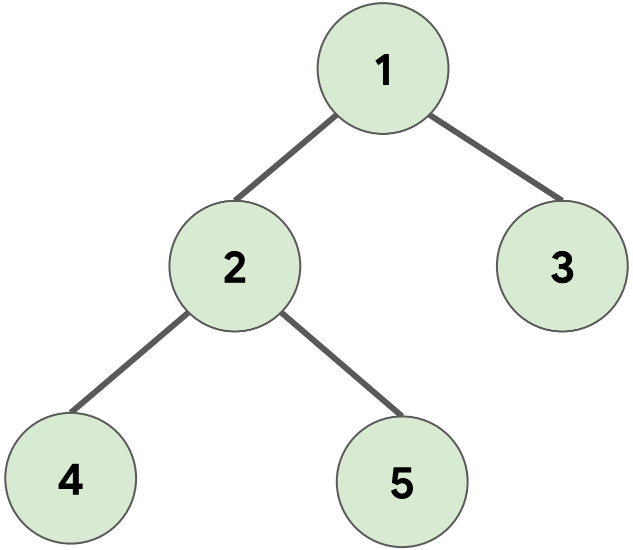




Terminologies in Trees

Terminology - Node

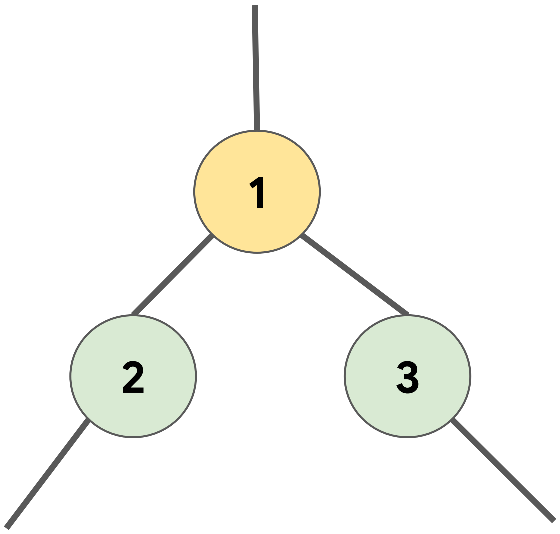
- A data structure that contains a value, a condition or a data structure (yes even trees)
- In trees, a node can have 0 or more children but at most one parent.




```python
class Node:
    def __init__(self, key:int):
        self.left = None
        self.right = None
        self.val = key
```

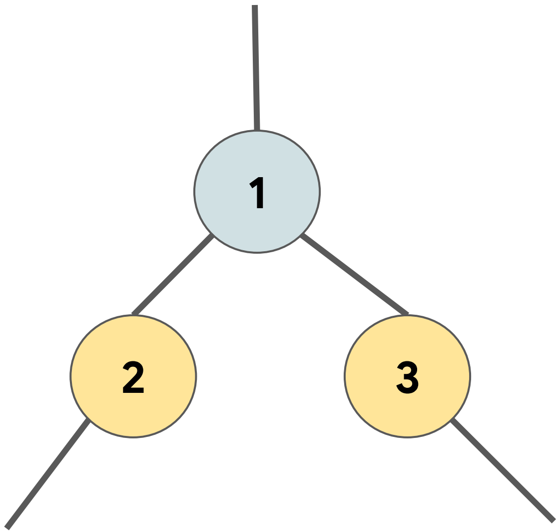
Terminology - Parent

- A node is called parent node to the nodes it's pointers point to.



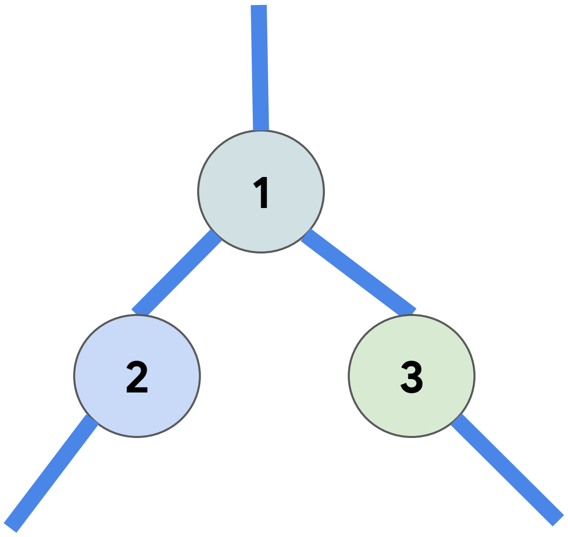
Terminologies - Child, Siblings

- The nodes a node's pointers point to are called child nodes of that node.
- Siblings: nodes that have the same parent node.



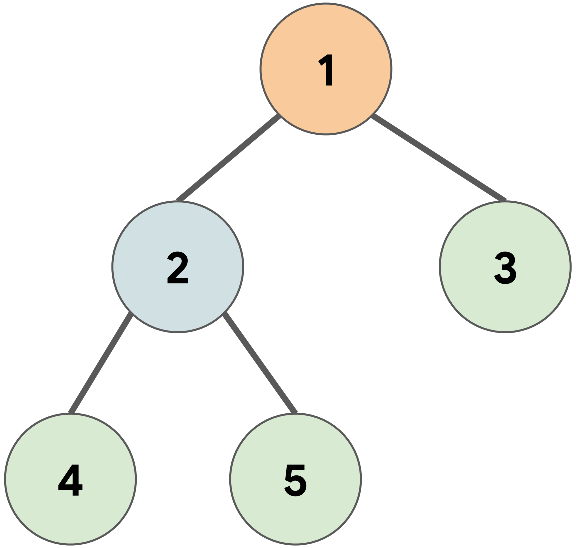
Terminology - Edge

- Edge: a connection between a child and parent node.



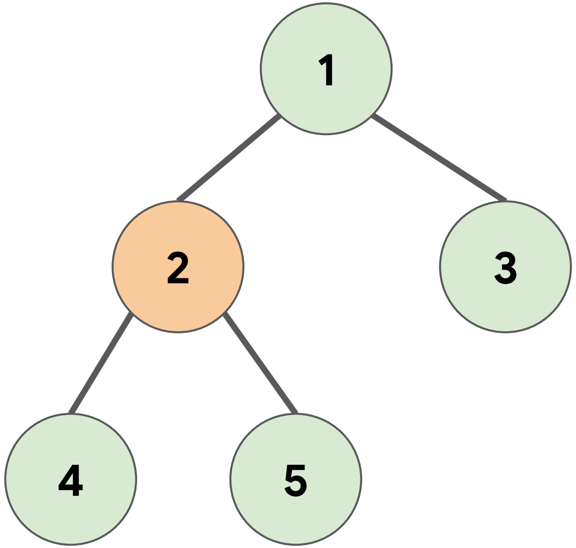
Terminology - Root Node

- A node with no parent is called a root node.



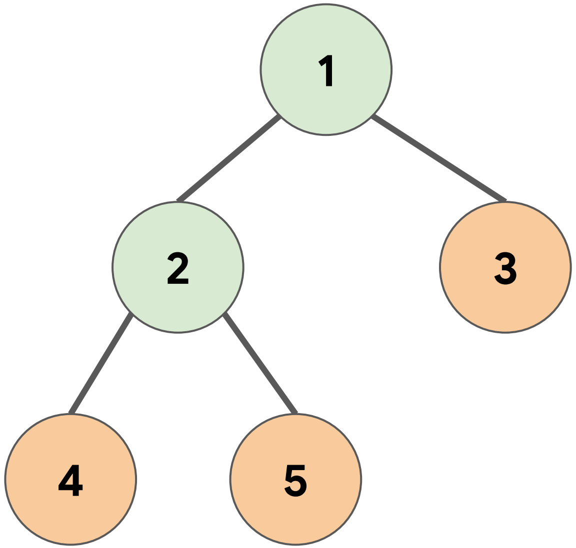
Terminology - Inner Node

- A node with the parent and the child is called an inner node or internal node



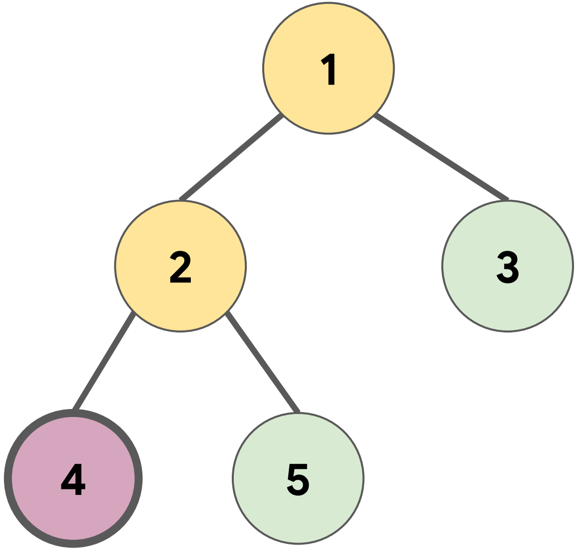
Terminology - Leaf Node

- A node with no children



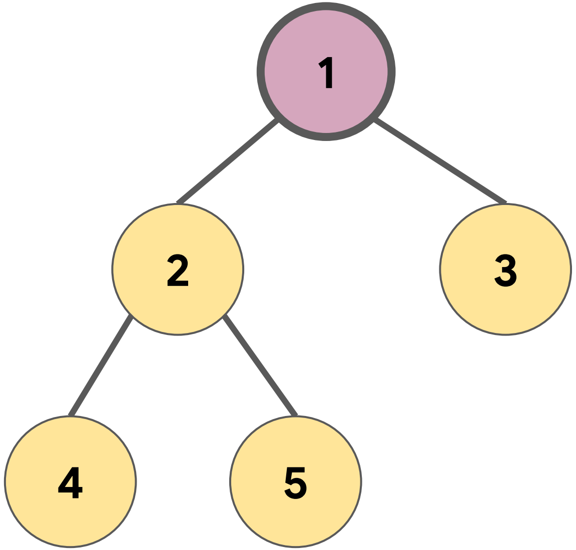
Terminology - Ancestor

- A node is called the ancestor of another node if it is the parent of the node or the ancestor of its parent node.
- In simpler terms, A is an ancestor of B if it is B's parent node, or the parent of B's parent node or the parent of the parent of B's parent node and so on.



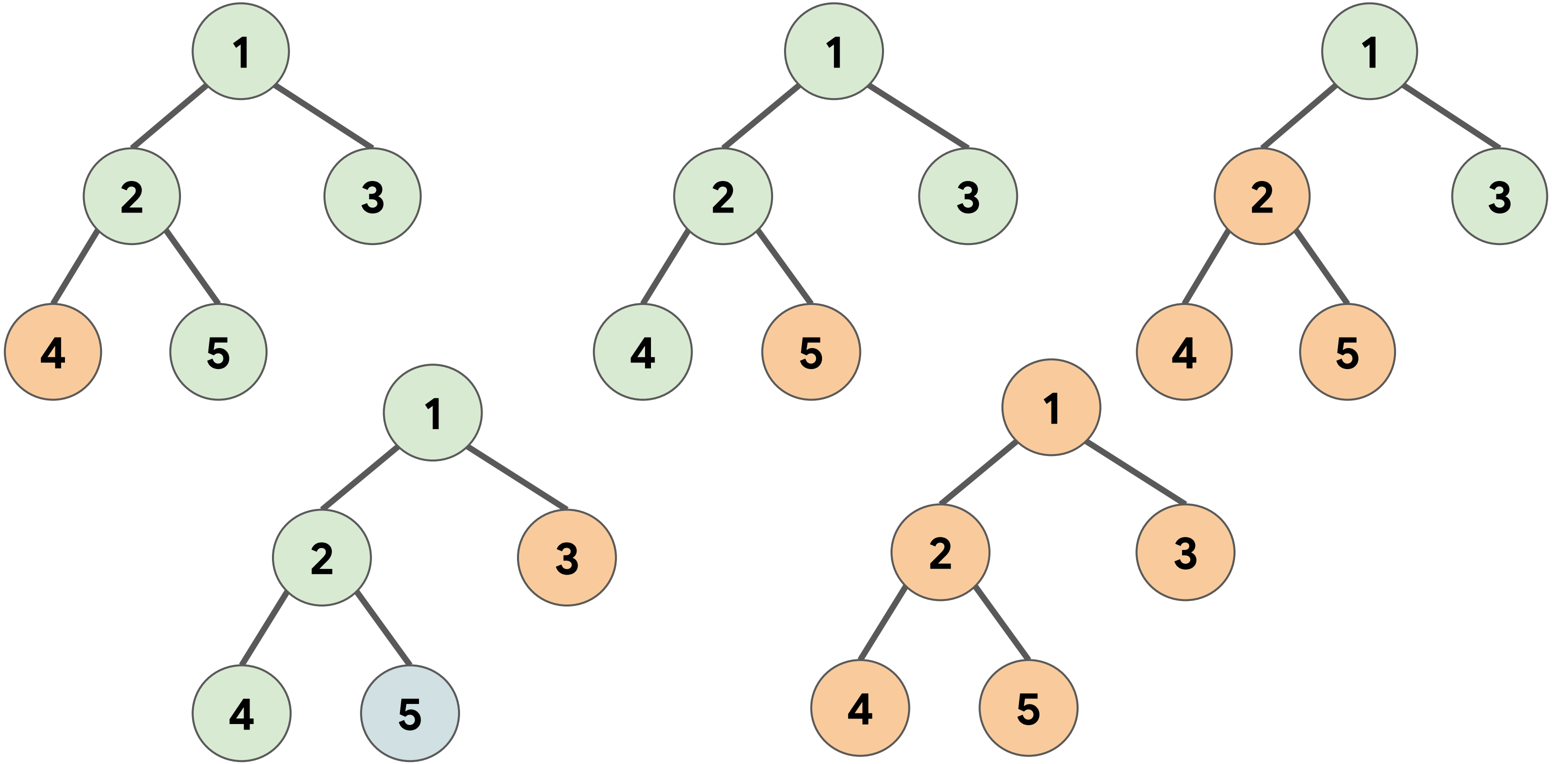
Terminology - Descendant

- A node A is called the descendant of another node B if B is the ancestor of A.
- In simpler terms, A is a descendant of B if A is the child node of B, or the child of the child node of B or the child of child of the child node of B and so on.



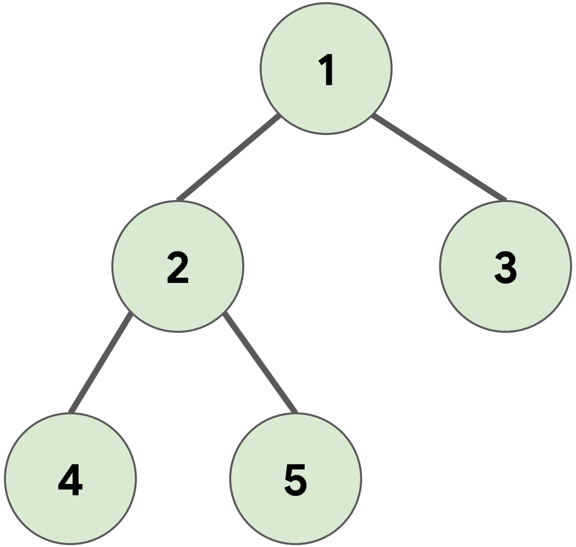
Terminology - Sub-tree

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



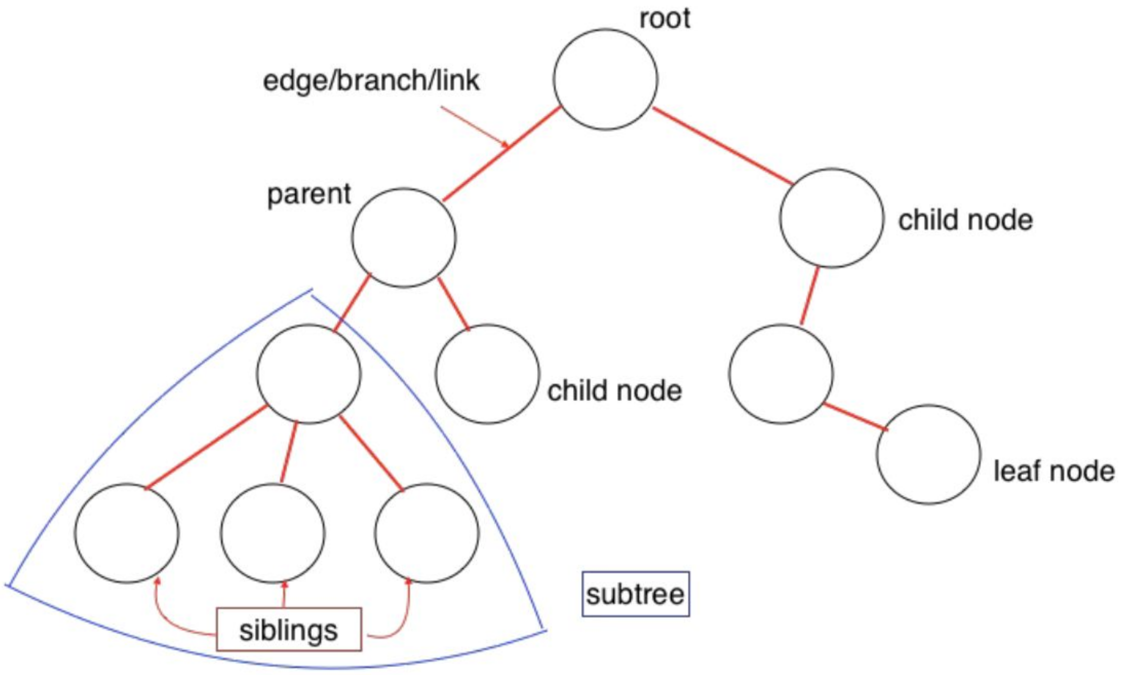
Terminology - Level, Height and Depth

- The level of a tree indicates how far you are from the root
- The height of a tree indicates how far you are from the farthest leaf
- The depth of the node is the total number of edges from the root to the current node. Level = depth + 1



Level 1	Height 2	Depth 0
Level 2	Height 1	Depth 1
Level 3	Height 0	Depth 2

Terminologies - Summary



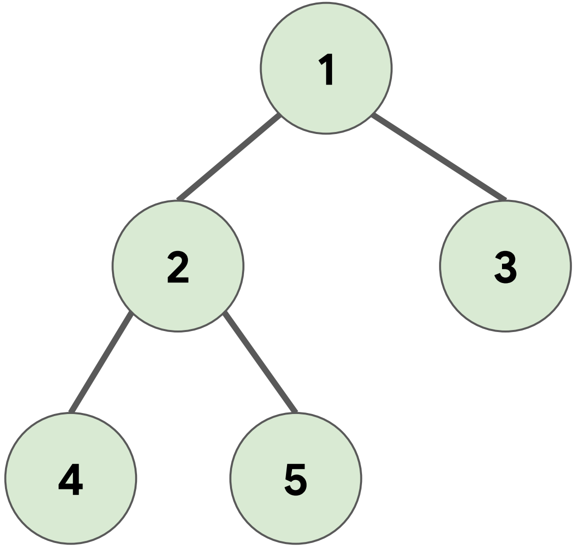
Types of Trees

Types

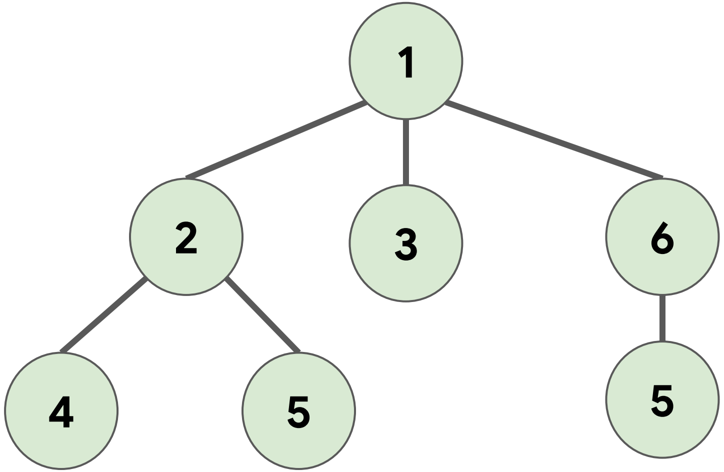
Depending on the number of children of every node, trees are generally classified as

1. Binary tree: Every node has at most two children
2. n-ary tree: Every node has at most n children

Binary trees



3-ary trees (a.k.a, Ternary Trees)



Types - Implementation

Binary tree

```python
class Node:
    def __init__(self, key:int):
        self.left = None
        self.right = None
        self.val = key
```

N-ary tree

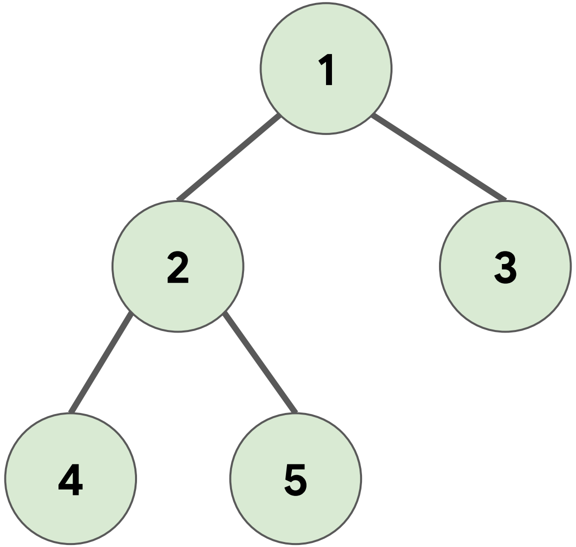
```python
class Node:
    def __init__(self, key:int):
        self.val = key
        self.children = []
        # len(children) <= N
```

Note: the N-ary tree implementation can be used for binary trees if `len(self.children) <= 2`

Types - Binary Trees

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.

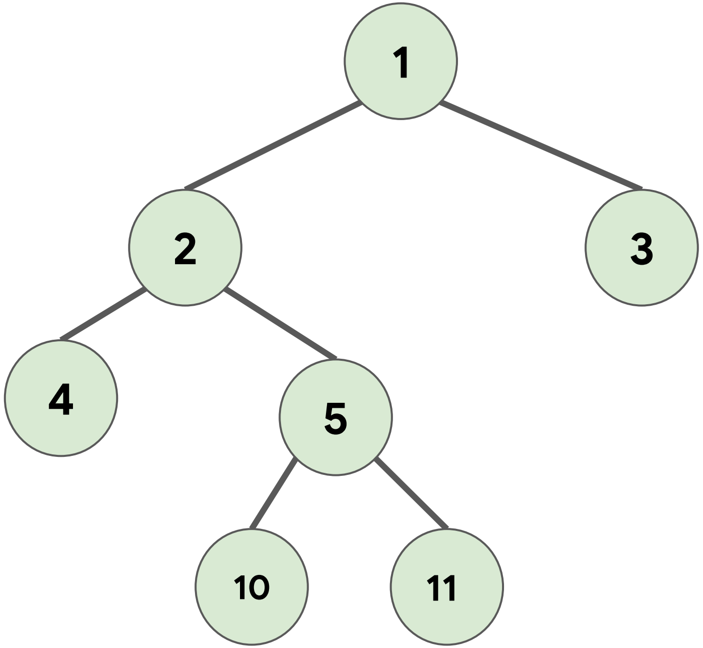
Binary trees



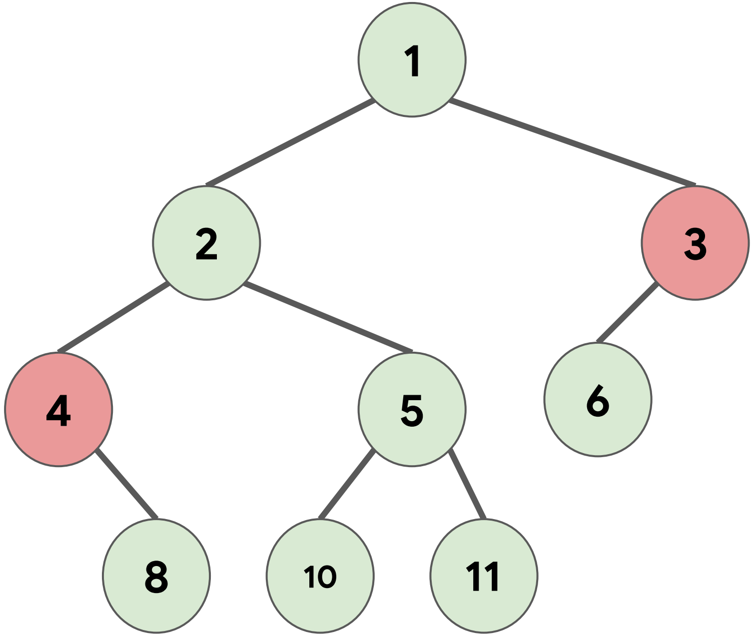
Types - Full Binary Trees

- A full binary tree is a special type of binary tree where each node has 0 or 2 children

Full Binary tree



Not a Full Binary tree



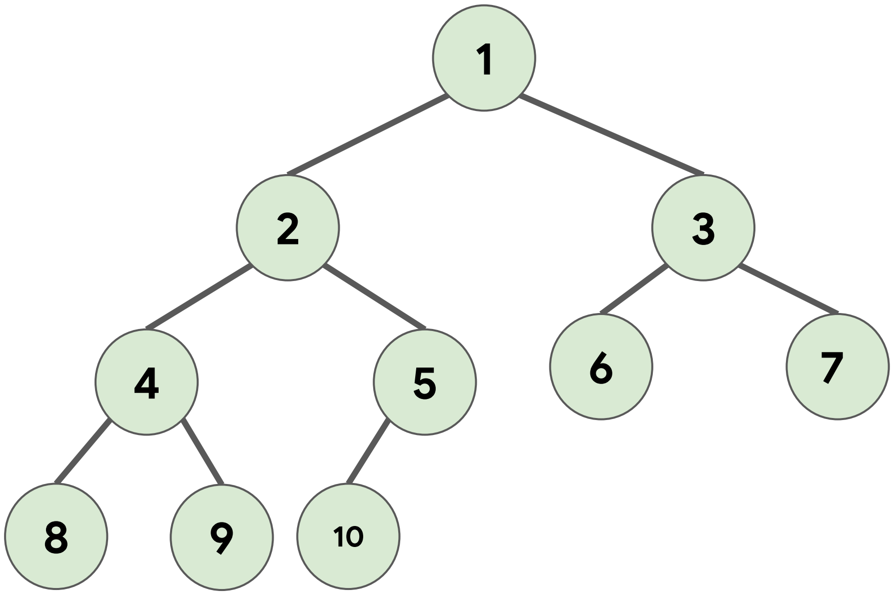
Types - Complete Binary Trees

- 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.

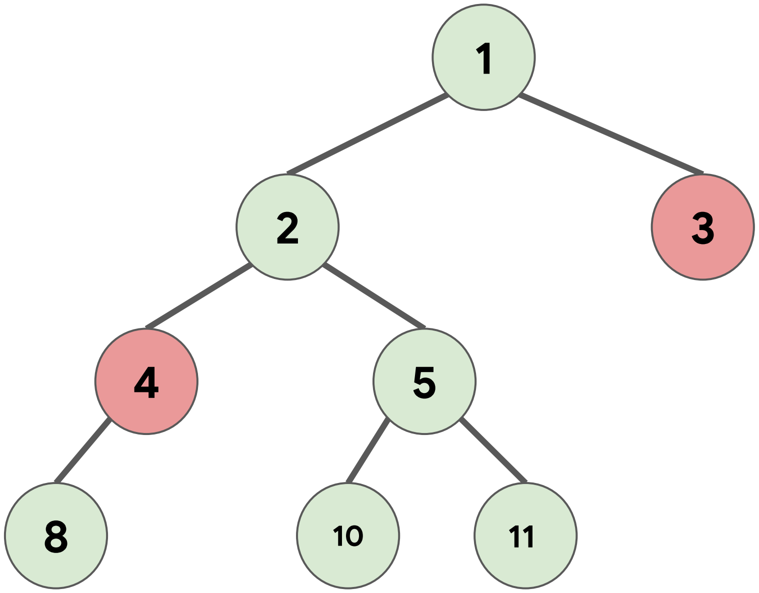
- A complete binary tree is just like a full binary tree, but with two major differences
  - All the leaf elements must lean towards the left.
  - The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.

Types - Complete Binary Trees

Complete Binary tree

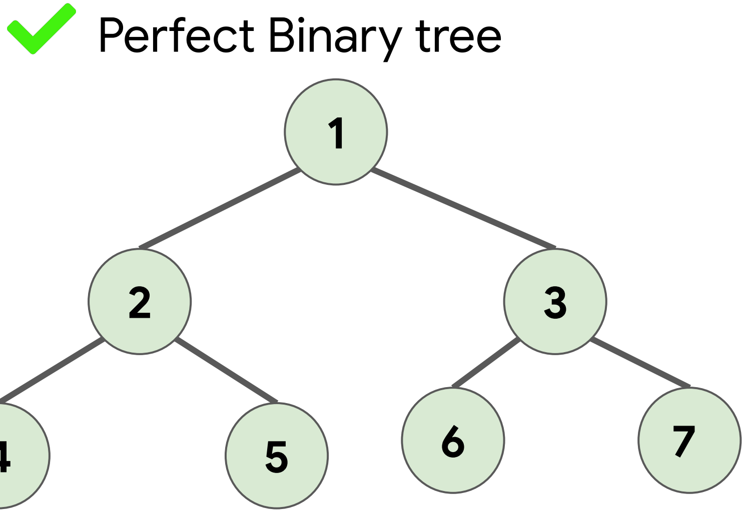


Not a Complete Binary tree



Types - Perfect Binary Trees

- 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.

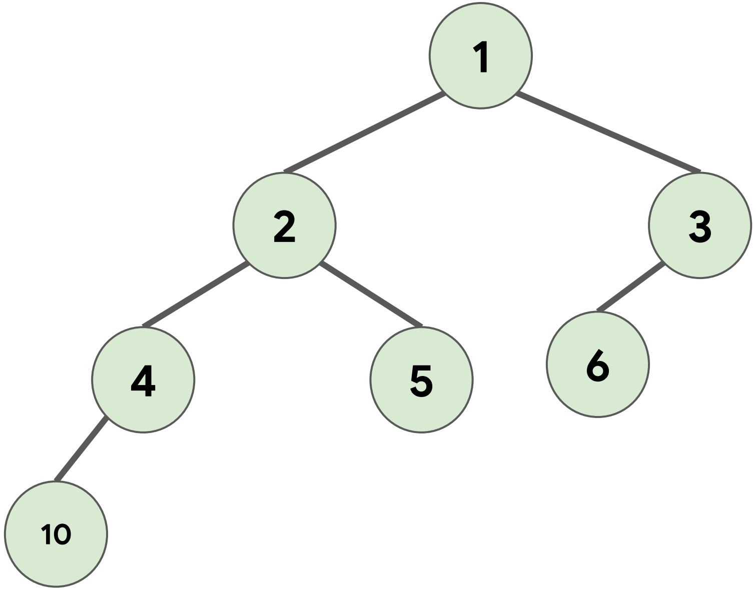


Types - Balanced Binary Trees

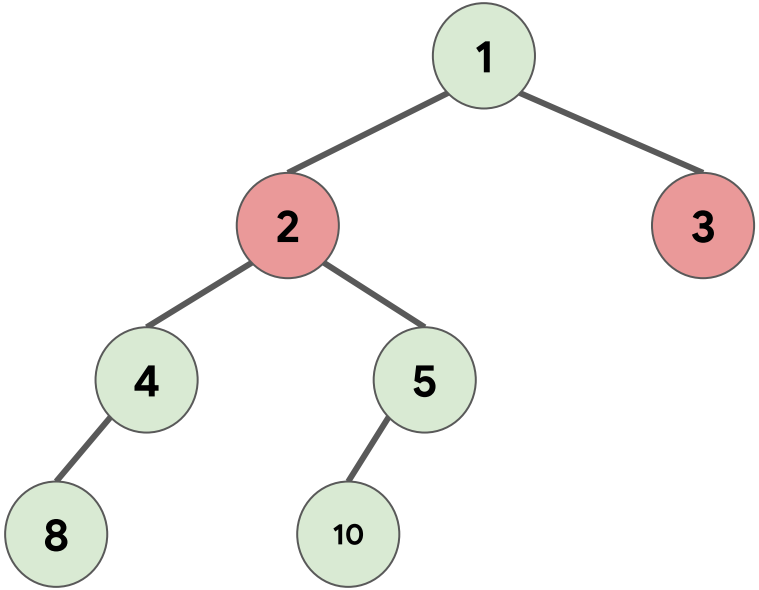
- A balanced binary tree is defined as a binary tree in which the height of the left and right subtree of any node differ by not more than 1.

Types - Balanced Binary Trees

✓ Balanced Binary tree

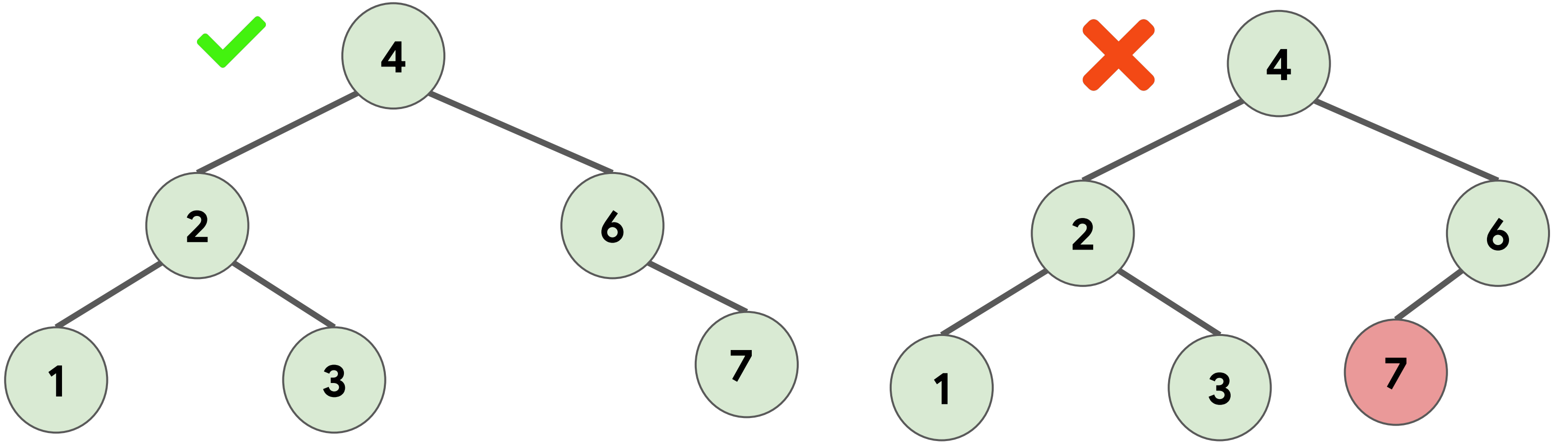


✗ Not a Balanced Binary tree



Types - Binary Search Trees

- 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 subtrees of the nodes should also be the binary search trees.



Types - Binary Search Trees

- Why binary search trees? Efficient search, insert and delete.
- Applications
  - Sorting large datasets
  - Maintaining sorted stream
  - Implementing dictionaries and priority queues

Tree Traversal

Tree Traversal

- Traversing a tree means visiting every value. Why would you need to do that?
  - To determine the a certain statistic, such as extremum value or average over all values
  - To sort the values

Tree Traversal - Depth First Search

- Depth-first search (DFS) is a method for exploring a tree or graph.
- In a DFS, you go as deep as possible down one path before backing up and trying a different one. You explore one path, hit a dead end, and go back and try a different one.
- There are basically three ways of traversing a binary tree:
  1. Preorder Traversal
  2. Inorder Traversal
  3. Postorder Traversal

Depth First Search - Preorder

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

- Problem Link

Depth First Search - Preorder

```python
# A function to do preorder tree traversal
def preOrder(root: Node):
    if root:
        # first add the data of node
        ans.append(root.val)
        # then recur on left child
        preOrder(root.left)
        # finally recur on right child
        preOrder(root.right)
```

Depth First Search - Inorder

- In inorder traversal, we traverse the left subtree first, then the the parent node and finally, the node's right subtree.
- Inorder traversal in BST results in a sorted order of the values

Problem Link

Depth First Search - Inorder

```python
# A function to do preorder tree traversal
def inOrder(root: Node):
    if root:
        # first recur on left child
        inOrder(root.left)
        # then add the data of node
        ans.append(root.val),
        # now recur on right child
        inOrder(root.right)
```

Depth First Search - Postorder

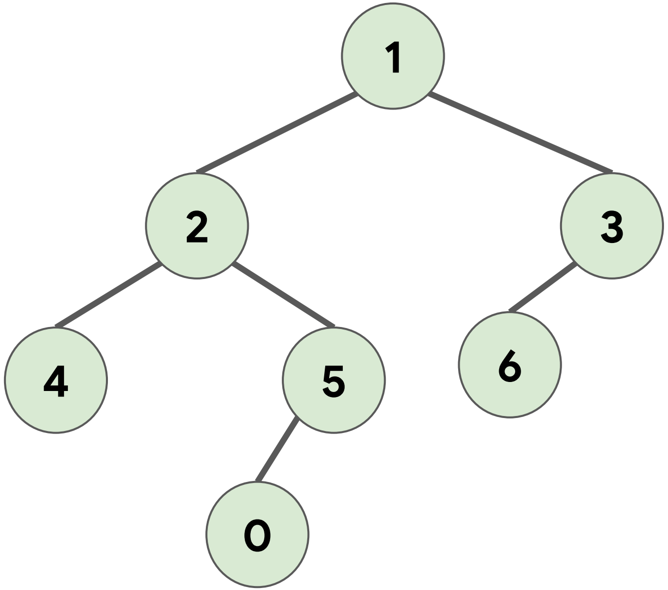
- In postorder traversal, we traverse the left subtree first, then the the right subtree of, and finally, the parent node.

[Problem Link]

Depth First Search - Postorder

```
# A function to do preorder tree traversal
def postOrder(root: Node):
    if root:
        # first recur on left child
        postOrder(root.left)
        # then recur on right child
        postOrder(root.right)
        # then add the data of node
        ans.append(root.val)
```

Depth First Search - Example

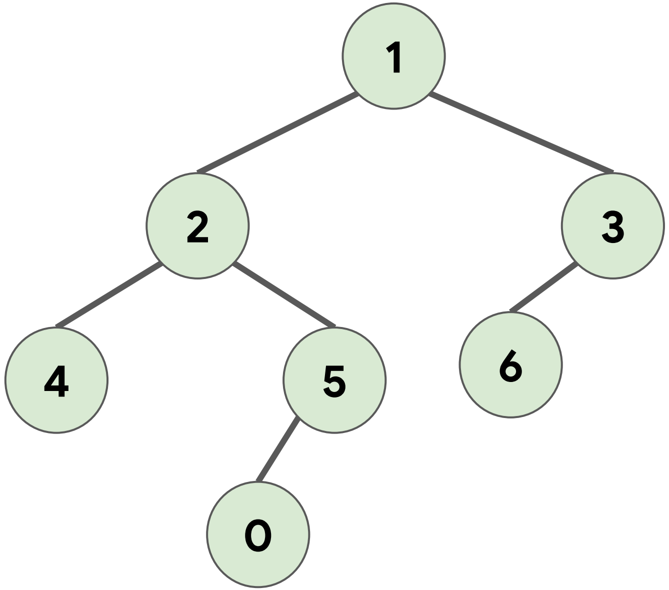


Preorder:

Inorder:

Postorder:

Depth First Search - Example



Preorder: 1 2 4 5 0 3 6

Inorder: 4 2 0 5 1 6 3

Postorder: 4 0 5 2 6 3 1

Checkpoint

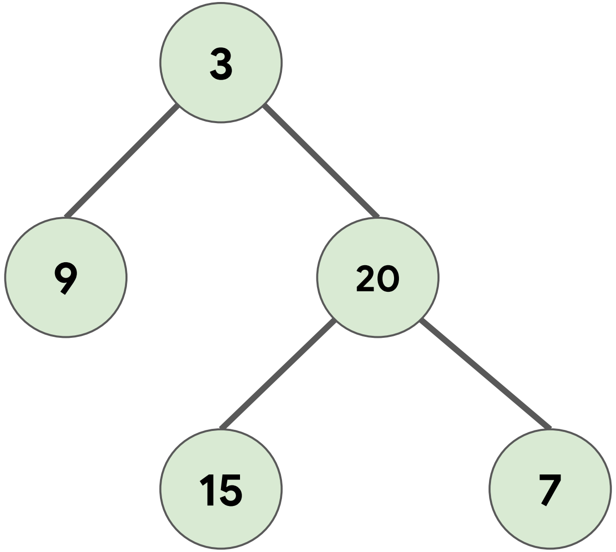
link

Practice Problem

[Link]

Simulation

Given the root of a binary tree, return its maximum depth. A binary tree maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.



Example 1:

Input: `root = [3,9,20,null,null,15,7]`

Output: 3

Example 2:

Input: `root = [1,null,2]`

Output: 2

Simulation - Solution

- What if the tree is empty?
  Answer: `max_depth = 0`

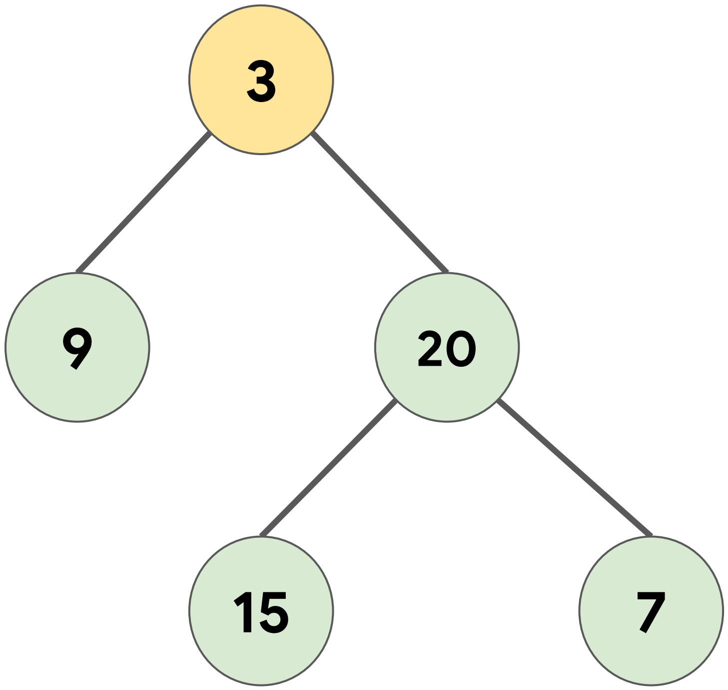
- What if we have just a node with no children?
  Answer: `max_depth = 1`

- What if the current node has left and/or right children?
  Answer: `max_depth = 1 + max(left_child_max_depth, right_child_max_depth)`

By recursively calculating the `max_depth` of each subtree

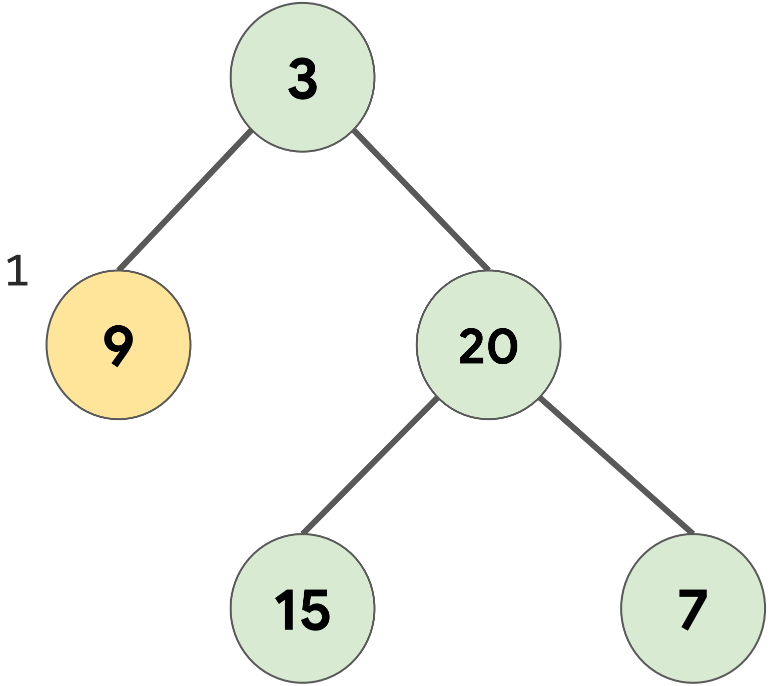
Simulation - Solution

```
1 + max(max_depth(root.left), max_depth(root.right))
```



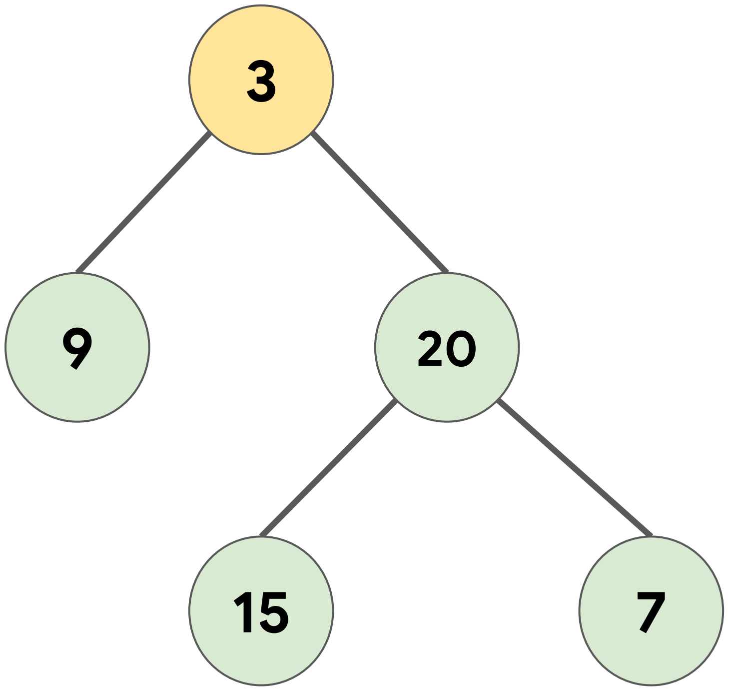
Simulation - Solution

```
1 + max(max_depth(root.left), max_depth(root.right))
```



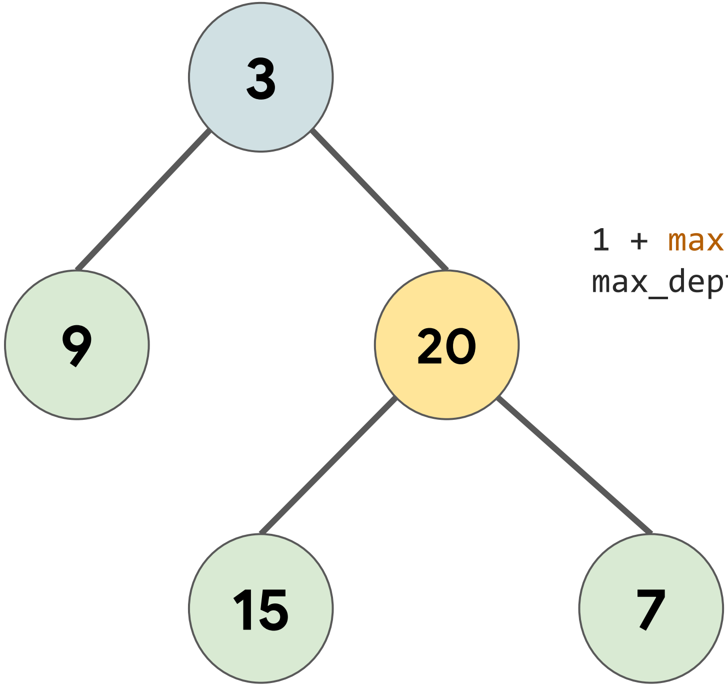
Simulation - Solution

```
1 + max(1, max_depth(root.right))
```



Simulation - Solution

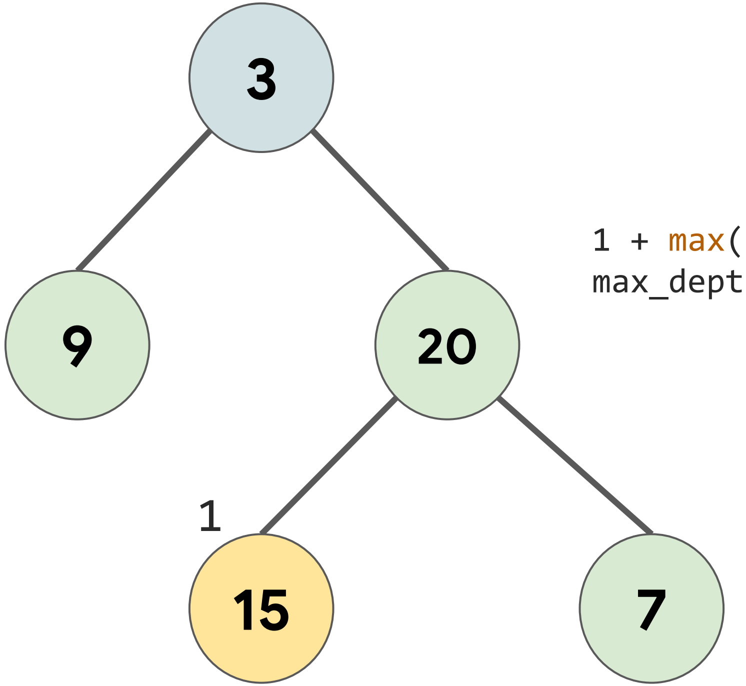
```
1 + max(1, max_depth(root.right))
```




```
1 + max(max_depth(root.left),
max_depth(root.right))
```

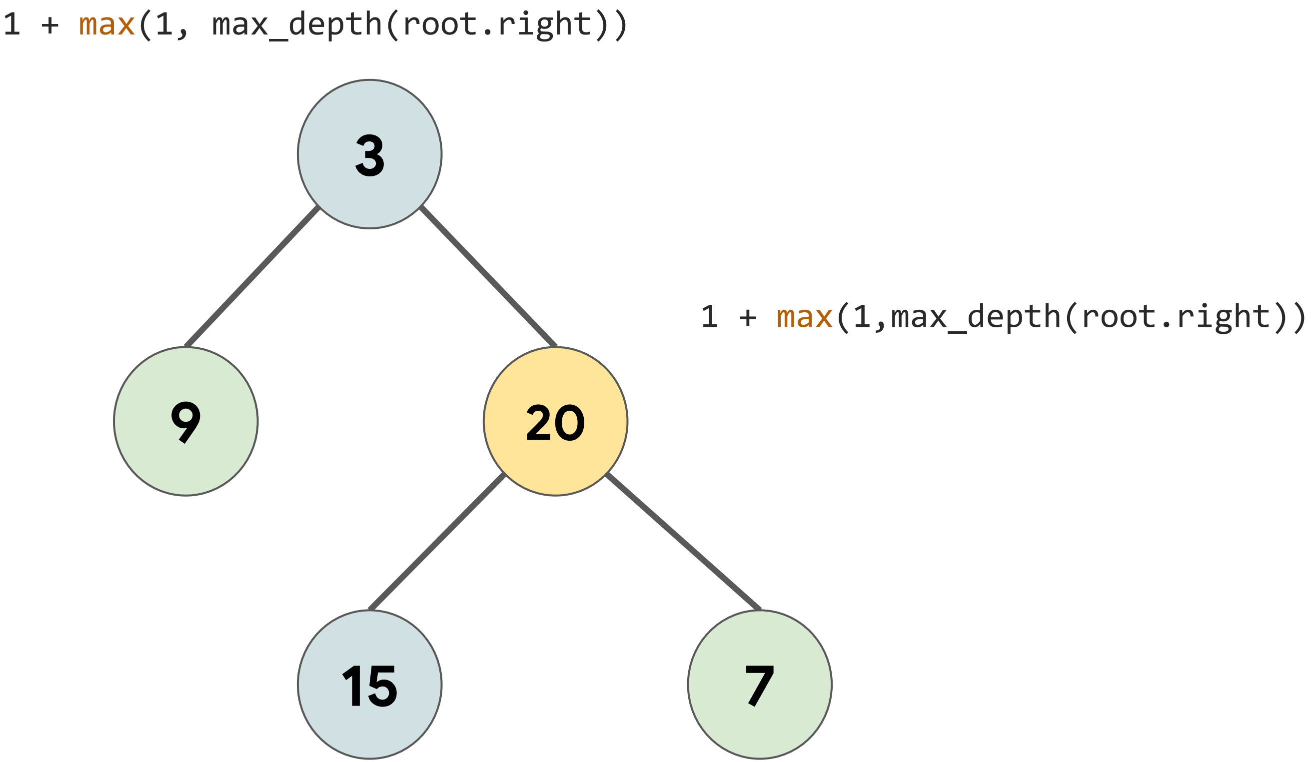
Simulation - Solution

```
1 + max(1, max_depth(root.right))
```

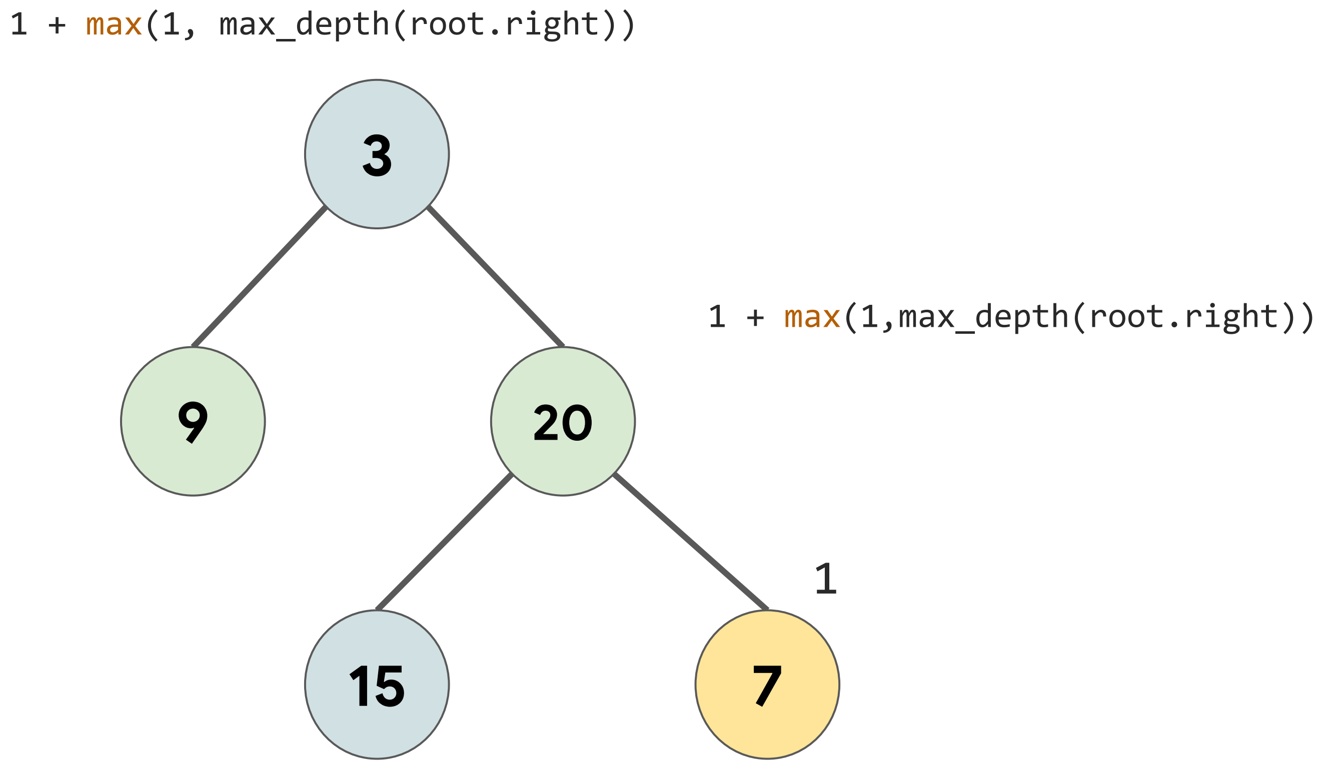



```
1 + max(max_depth(root.left),
max_depth(root.right))
```

Simulation - Solution



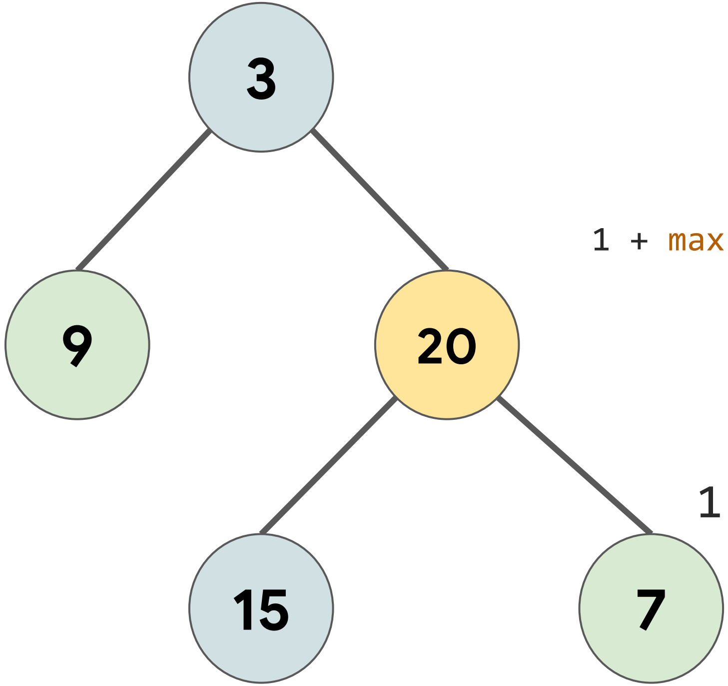
Simulation - Solution



Simulation - Solution

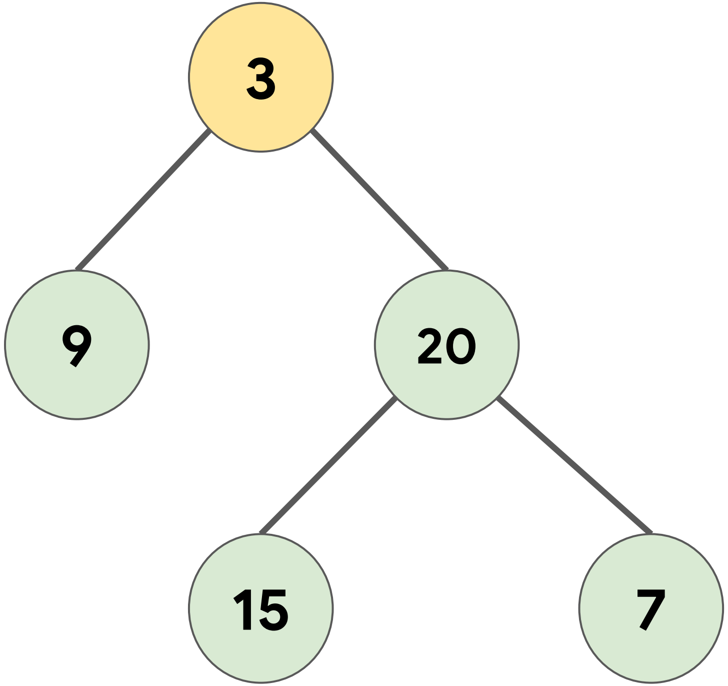
`1 + max(1, max_depth(root.right))`

`1 + max(1, 1) = 2`



Simulation - Solution

```
1 + max(1,2) = 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:
     def maxDepth(self, root:TreeNode):
          """
          :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(d)

Implementation - Iterative

```python
class Solution:
    def maxDepth(self, root:TreeNode):
        """
        :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(d)

Question

[Insert into a Binary Search Tree]

Basic Operation on Trees

Operation - Searching

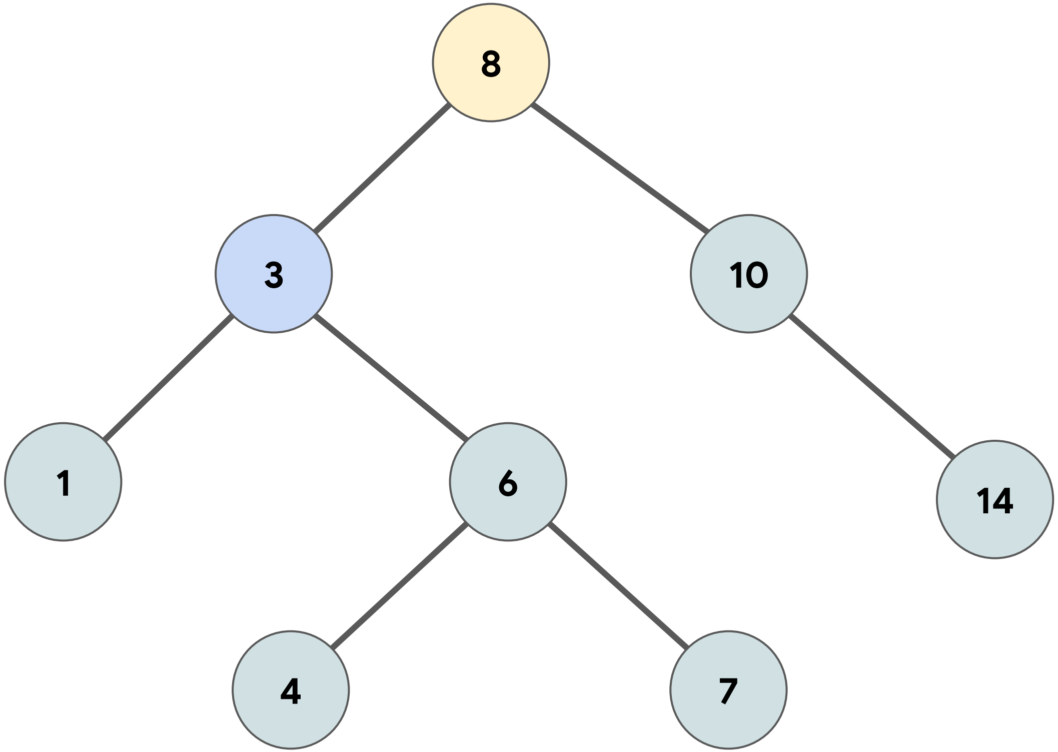
- The algorithm depends on the property of BST that if each left subtree has values below parent and each right subtree has values above the parent.

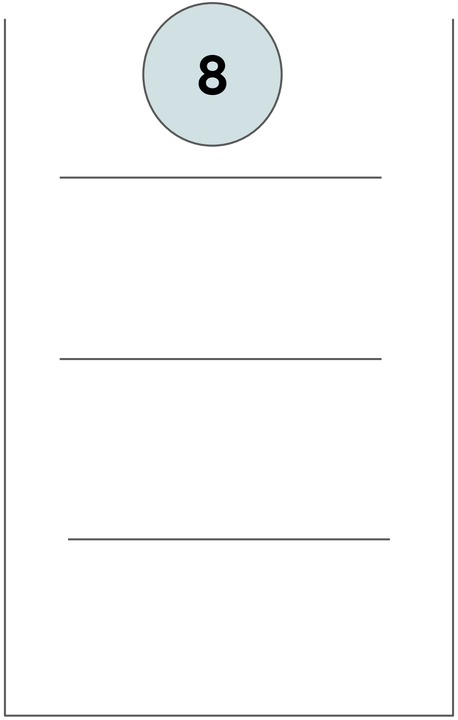
- If the value is below the parent, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree

- If the value is above the parent, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.

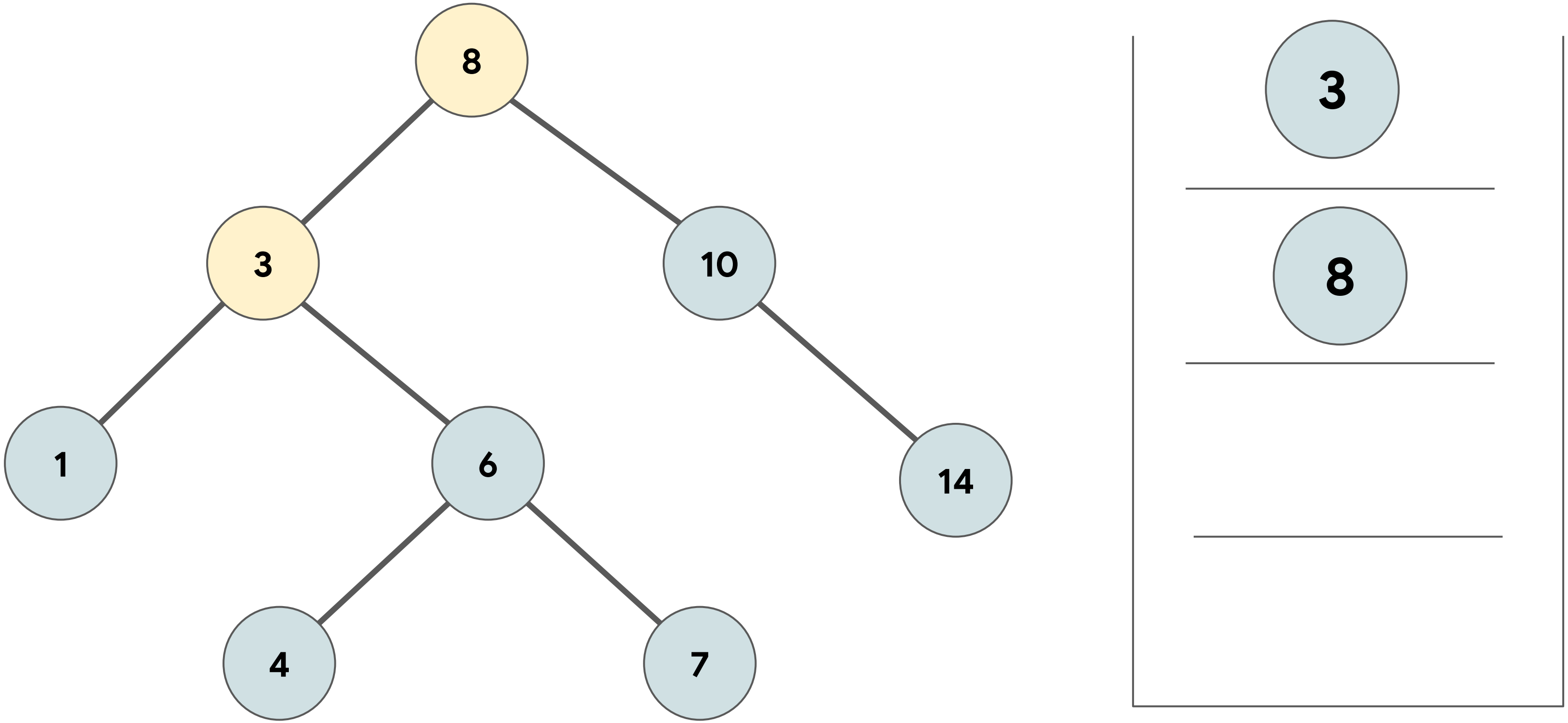
- Let us try to visualize this with a diagram searching for 4 in the tree:

Operation - Searching

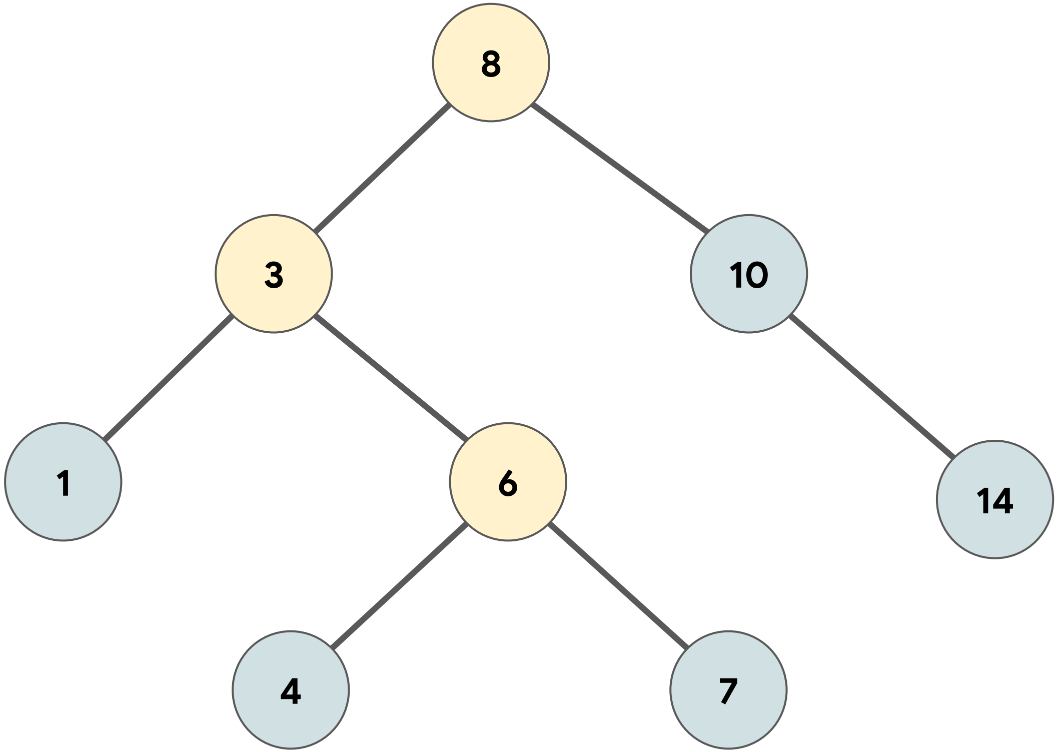


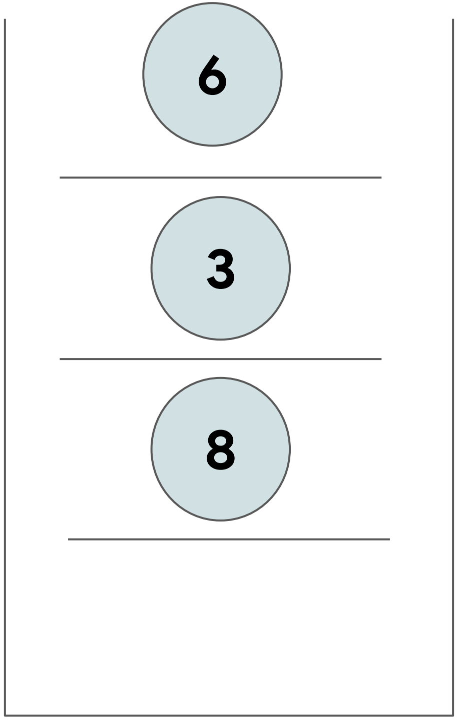


Operation - Searching

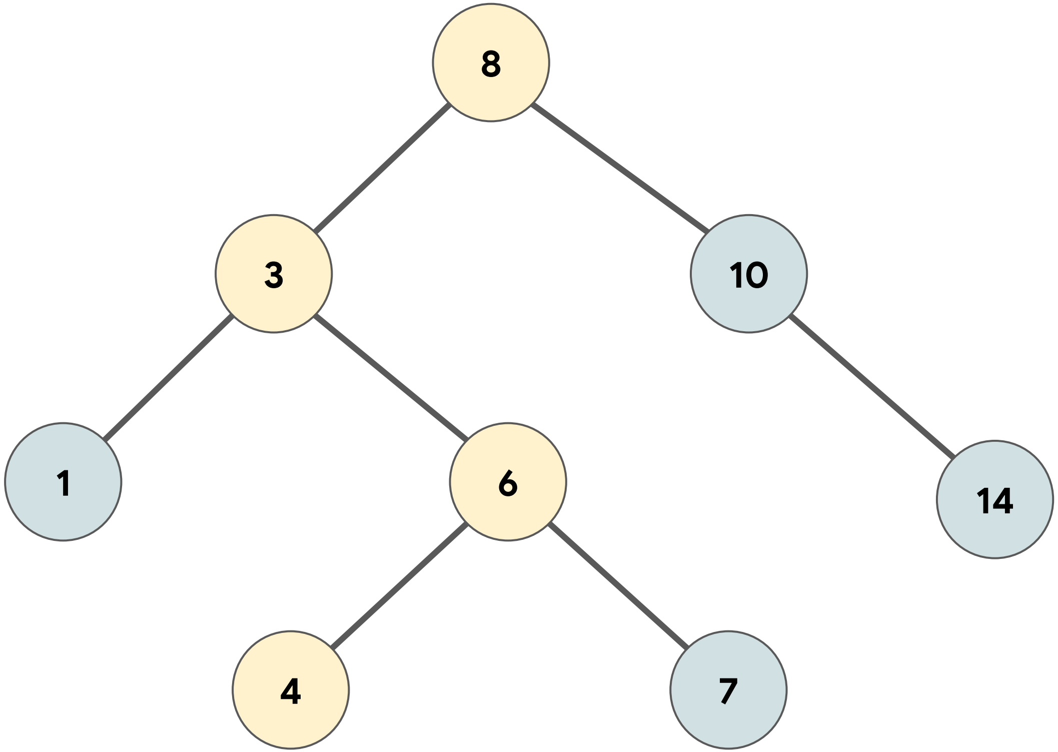


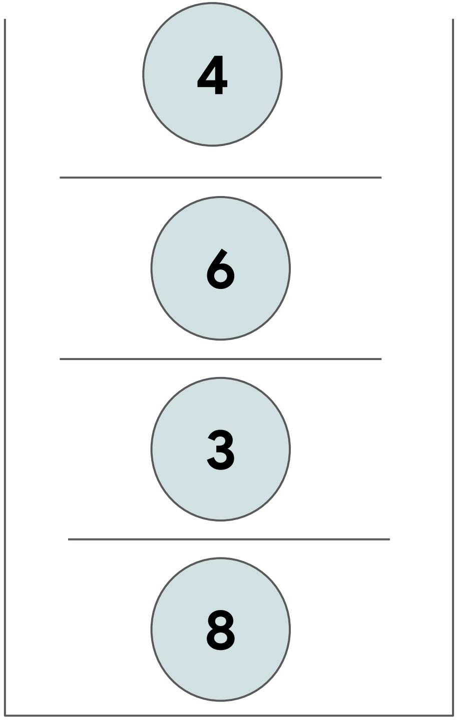
Operation - Searching





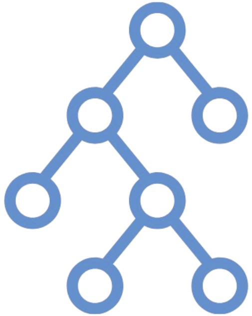
Operation - Searching





Operation - Searching Algorithm

```
def search(root):
    if root is 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.val)
```



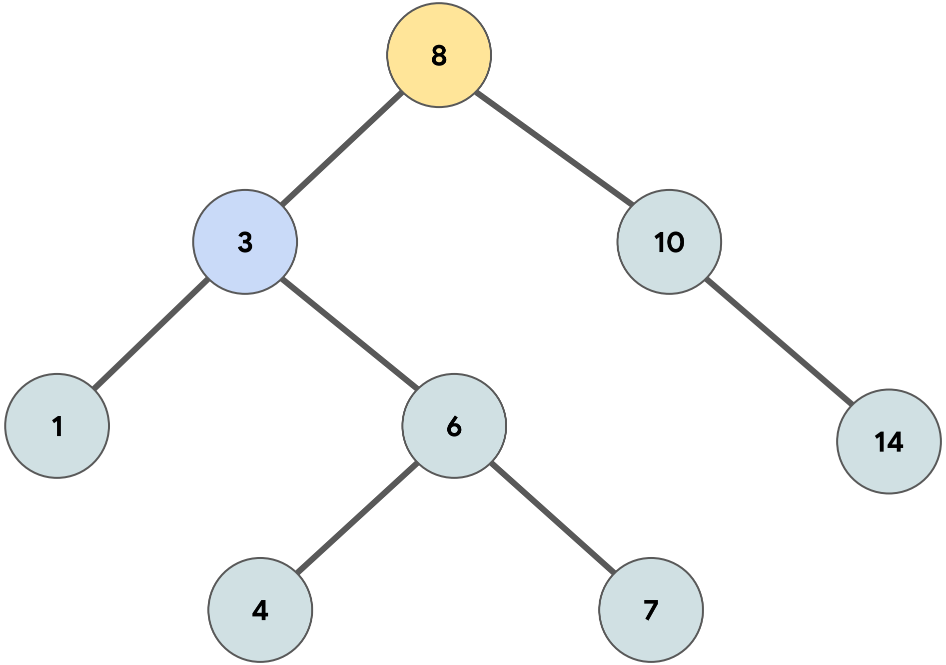
Operation - Insertion

- Inserting a value in the correct position is similar to searching because we try to maintain the BST 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 5 to an existing BST.

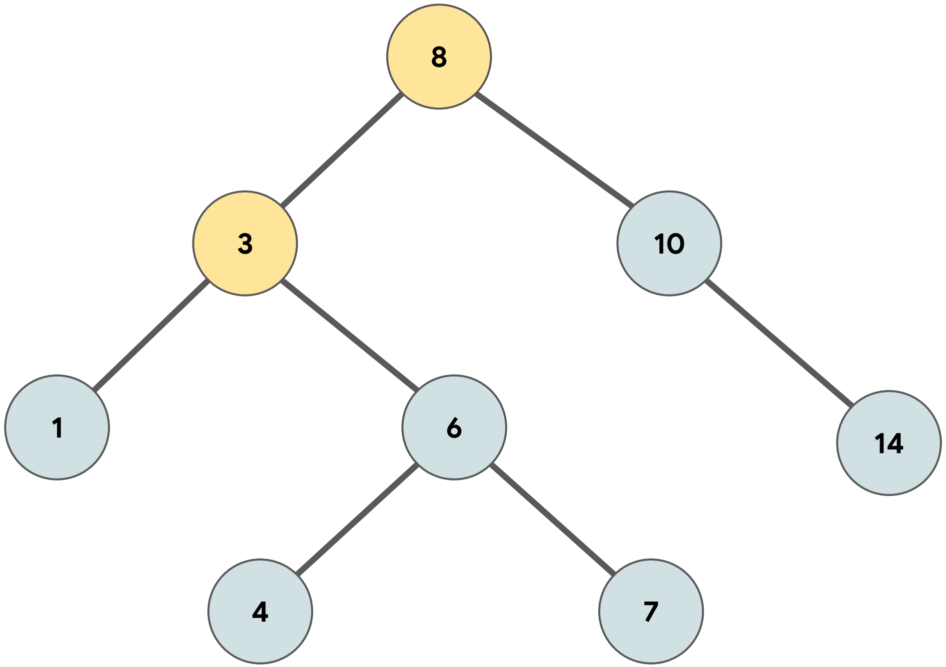
Operation - Insertion

Insert 5 in to the BST



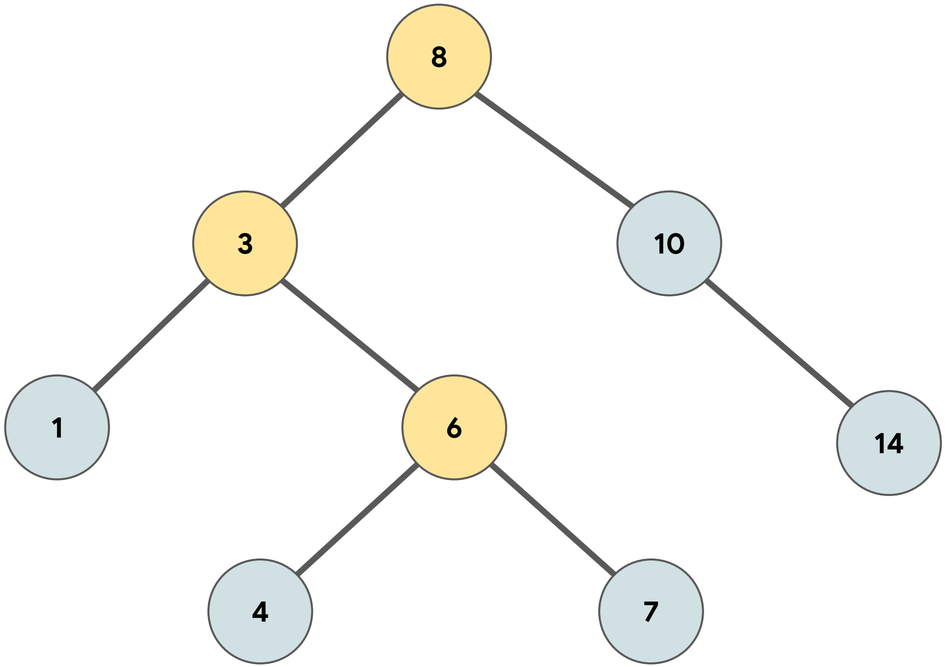
Operation - Insertion

Insert 5 in to the BST



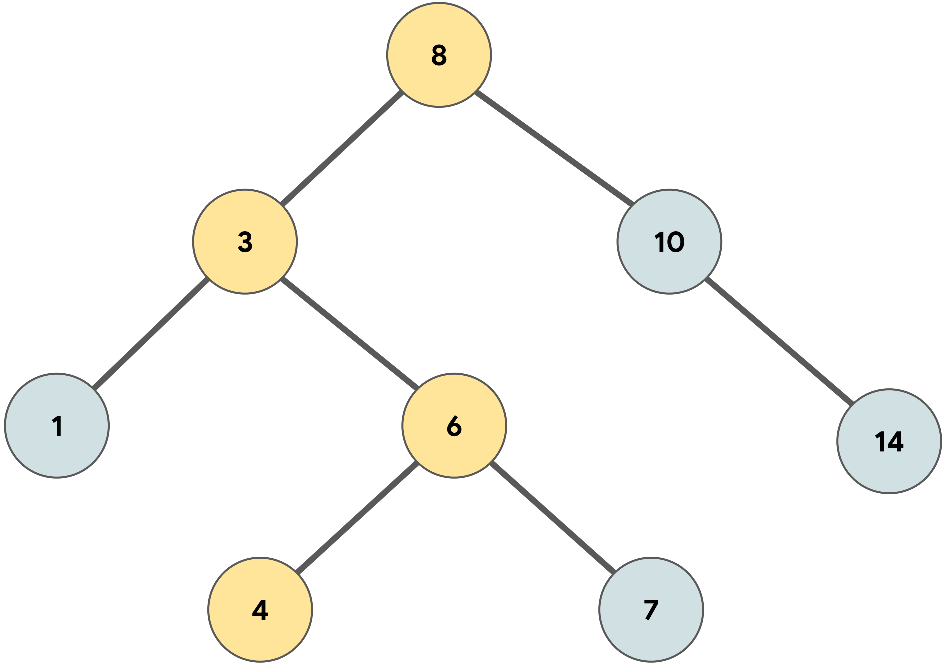
Operation - Insertion

Insert 5 in to the BST



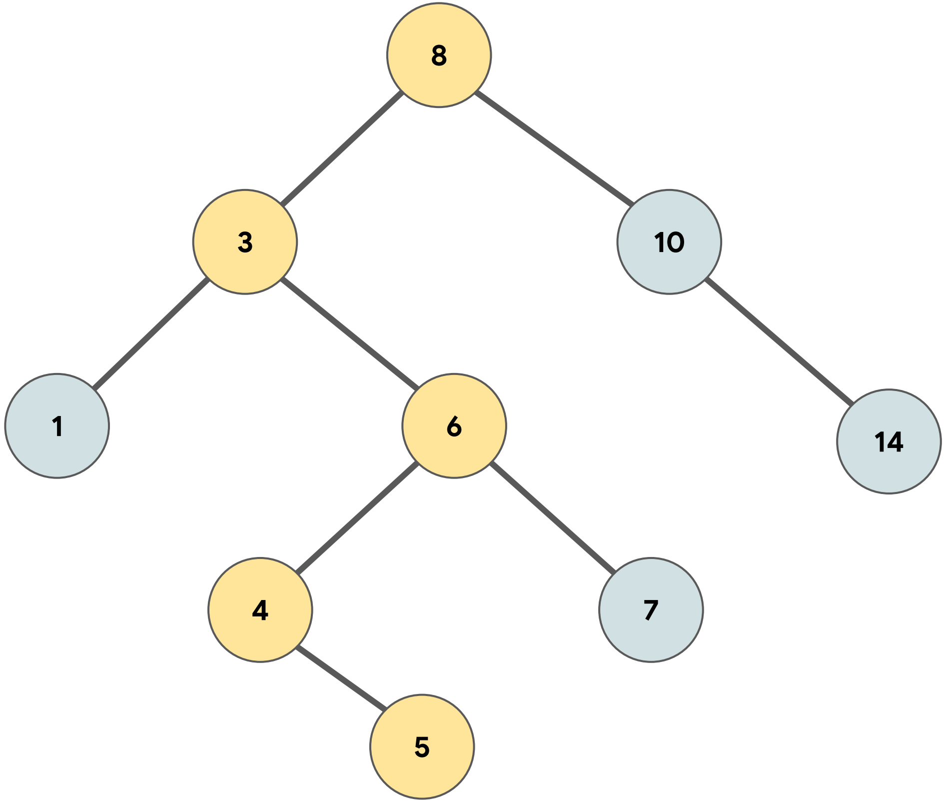
Operation - Insertion

Insert 5 in to the BST



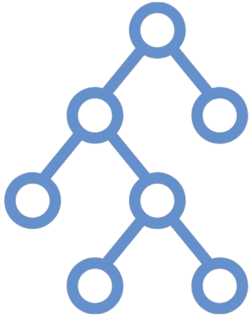
Operation - Insertion

Insert 5 in to the BST



Operation - Insertion Algorithm

```python
def insert(root, data):
    if root is None:
        return Node(data)
    if data < root.val:
        root.left = insert(root.left, data)
        return root
    if data > root.val:
        root.right = insert(root.right, data)
        return root
```



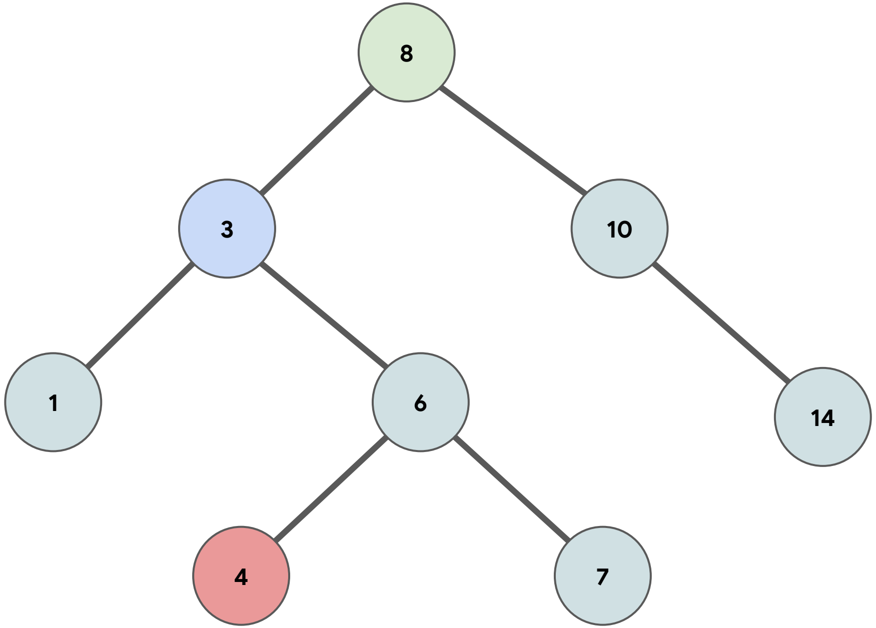
Practice Problem

Delete Node in a BST

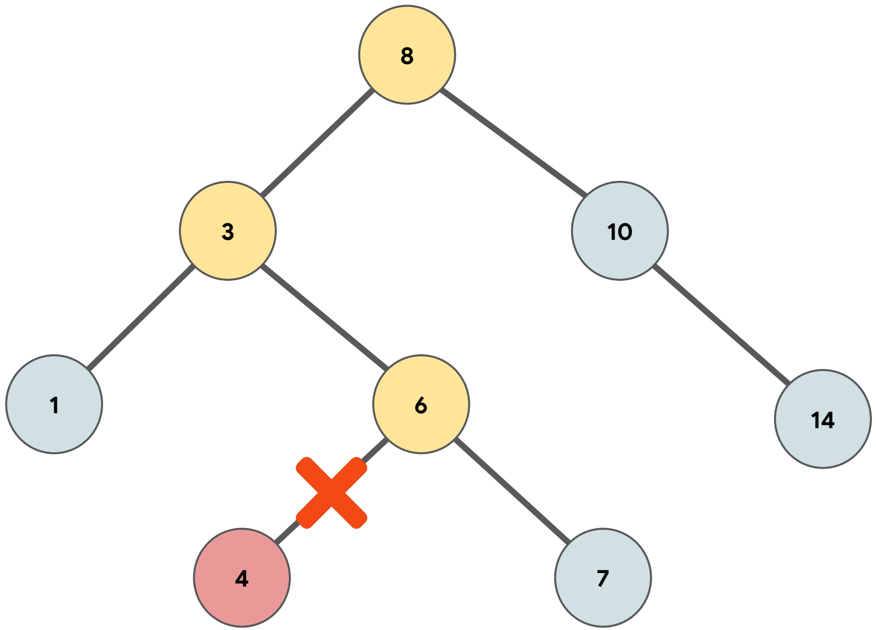
Operation - Deletion

- 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.



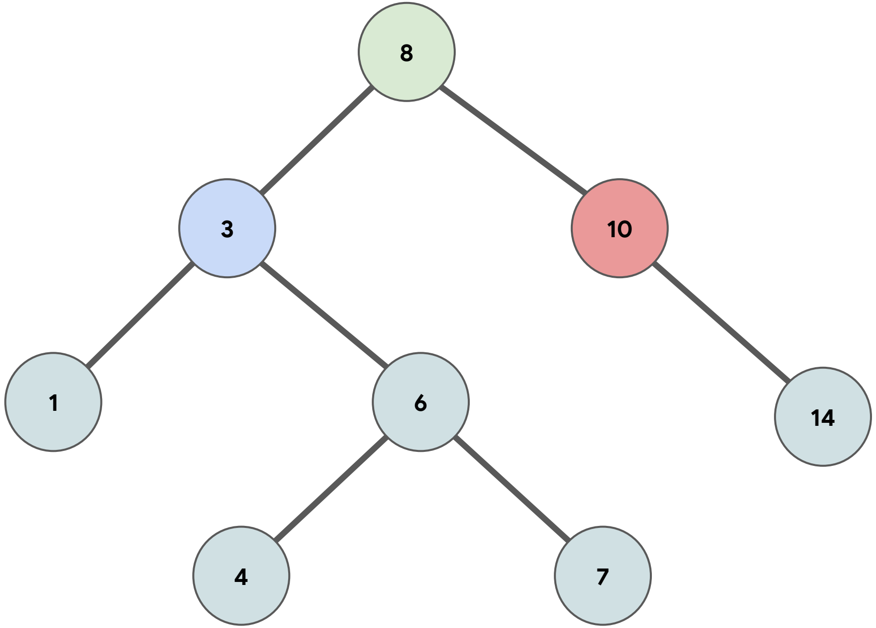
Operation - Deletion



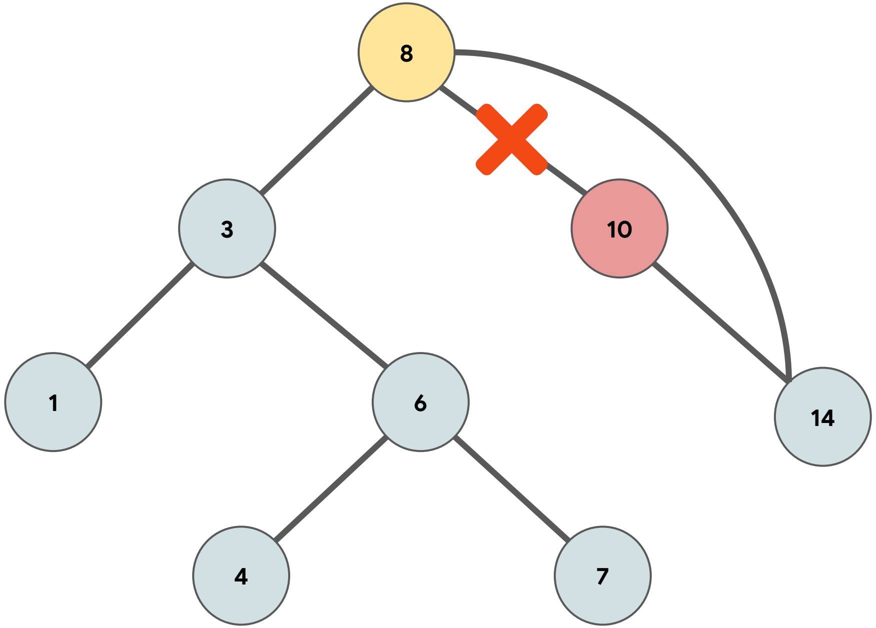
Operation - Deletion

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.



Operation - Deletion

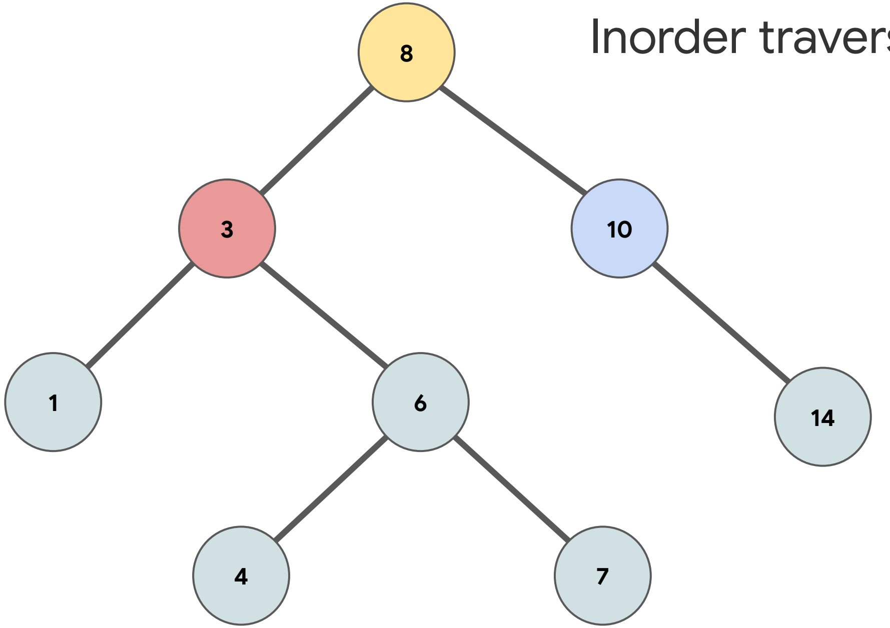


Operation - Deletion

Case Three: The node to be deleted has two children. In such a case follow the steps below:

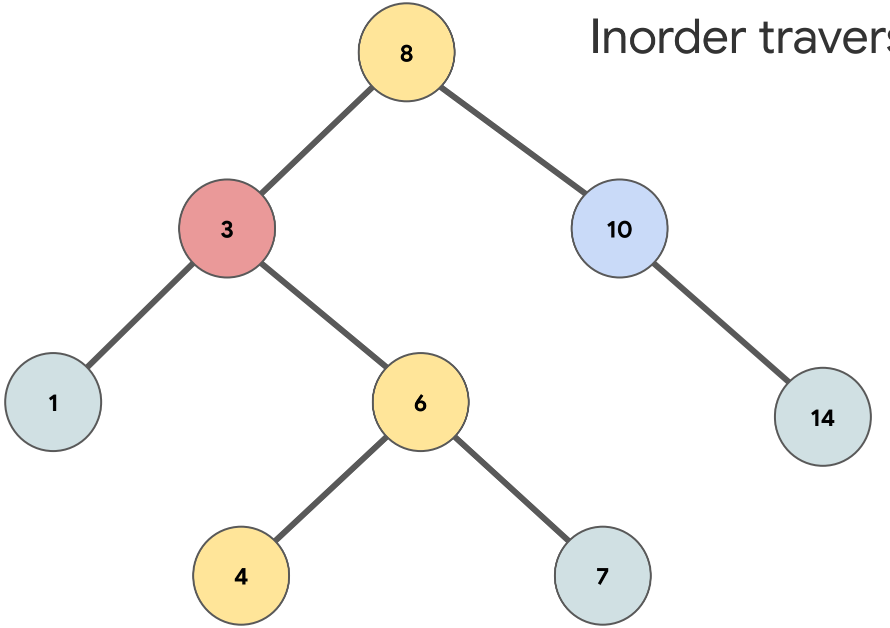
1. Get the **inorder successor** of that node. Why?
2. Replace the node with the inorder successor.
3. Remove the inorder successor from its original position.

Operation - Deletion



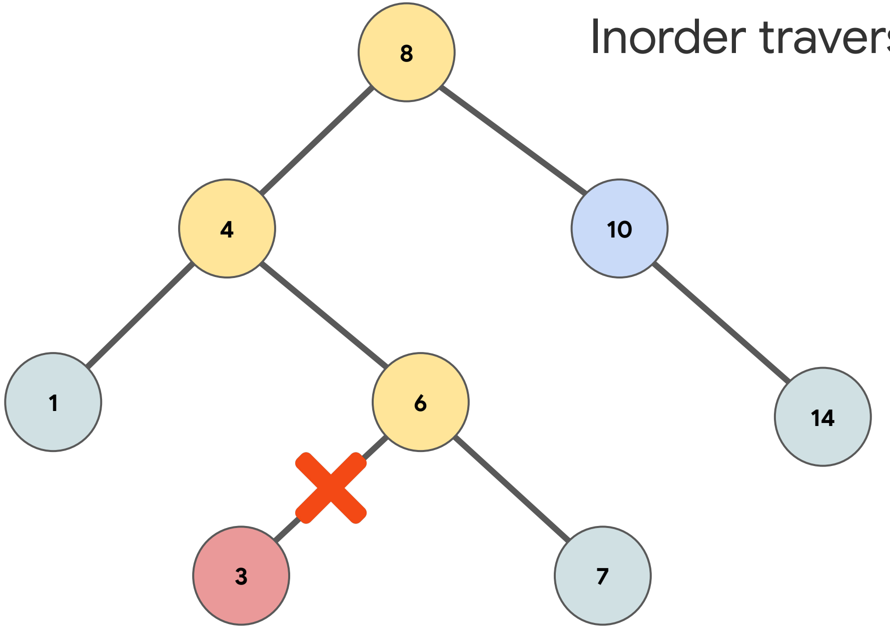
Inorder traversal: 1 3 4 6 7 8 10 14

Operation - Deletion



Inorder traversal: 1 3 **4** 6 7 8 10 14

Operation - Deletion



Inorder traversal: 1 **4** 6 7 8 10 14

Operation - Deletion Algorithm

```python
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)
    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,
        # place the inorder successor in position of the node to be deleted
        temp = minValueNode(root.right)
        root.key = temp.key

        # Delete the inorder successor
        root.right = deleteNode(root.right, temp.key)
    return root
```

Operation - Deletion Algorithm

```
`# Find the inorder successor
 def minValueNode(node):
     current = node

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

     return current
```

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

n = number of nodes
h = height of the binary tree

Binary Tree

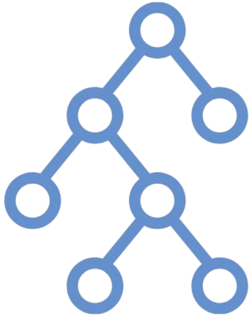
- Traversing
  - Time = `O(n)`
- Searching
  - Time = `O(n)`
- Insertion
  - Time = `O(n)`
- Deletion
  - Time = `O(n)`
- Space = `O(h)`

Binary Search Tree

- Traversing
  - Time = `O(n)`.
- Searching
  - Time = `O(h)`
- Insertion
  - Time = `O(h)`
- Deletion
  - Time = `O(h)`
- Space = `O(h)`

Common Pitfalls

- Null pointer exceptions
- Assuming the tree is balanced
- Wrong choice of traversal
- Wrong recurrence relations and base cases
- Stack overflow



Applications of Trees

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

Practice Questions

- Merge Two Binary Trees
- Search in Binary Search Trees
- Same Tree
- Lowest Common Ancestor of Binary Search Tree
- Validate Binary Search Trees
- Binary Tree Zigzag Level Order Traversal
- Maximum Difference Between Node and Ancestor
- Kth smallest Element in BST
- Maximum Sum BST in Binary Tree

Quote of the Day

"A tree with strong roots laughs at storms."
- Malay Proverb