Welcome to the CoGrammar Trees and Heaps

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.

CoGrammar

Coding Interview Workshop Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: **Questions**

Coding Interview Workshop Housekeeping cont.

- For all **non-academic questions**, please submit a query: www.hyperiondev.com/support

- Report a **safeguarding** incident: www.hyperiondev.com/safeguardreporting

- We would love your **feedback** on lectures: Feedback on Lectures

Skills Bootcamp
8-Week Progression Overview

Fulfil 4 Criteria to Graduation

✅ **Criterion 1: Initial Requirements**

Timeframe: First 2 Weeks
Guided Learning Hours (GLH):
Minimum of 15 hours
Task Completion: First four tasks

Due Date: 24 March 2024

✅ **Criterion 2: Mid-Course Progress**

60 Guided Learning Hours

Data Science - **13 tasks**
Software Engineering - **13 tasks**
Web Development - **13 tasks**

Due Date: 28 April 2024

CoGrammar

Skills Bootcamp Progression Overview

✅ Criterion 3: Course Progress

Completion: All mandatory tasks, including Build Your Brand and resubmissions by study period end
Interview Invitation: Within 4 weeks post-course
Guided Learning Hours: Minimum of 112 hours by support end date (10.5 hours average, each week)

✅ Criterion 4: Demonstrating Employability

Final Job or Apprenticeship Outcome: Document within 12 weeks post-graduation
Relevance: Progression to employment or related opportunity

CoGrammar

Lecture Objectives

- Identify components and properties of trees
- Differentiate between various types of trees
- Implement tree operations in Python and JavaScript
- Construct and manipulate heaps
- Analyse and apply tree and heap concepts to solve problems

CoGrammar

Introduction

- ❖ Trees are fundamental data structures in computer science
- ❖ They are used to represent hierarchical relationships between data elements
- ❖ Understanding trees is crucial for solving complex problems and optimizing algorithms

CoGrammar

Trees and Heaps

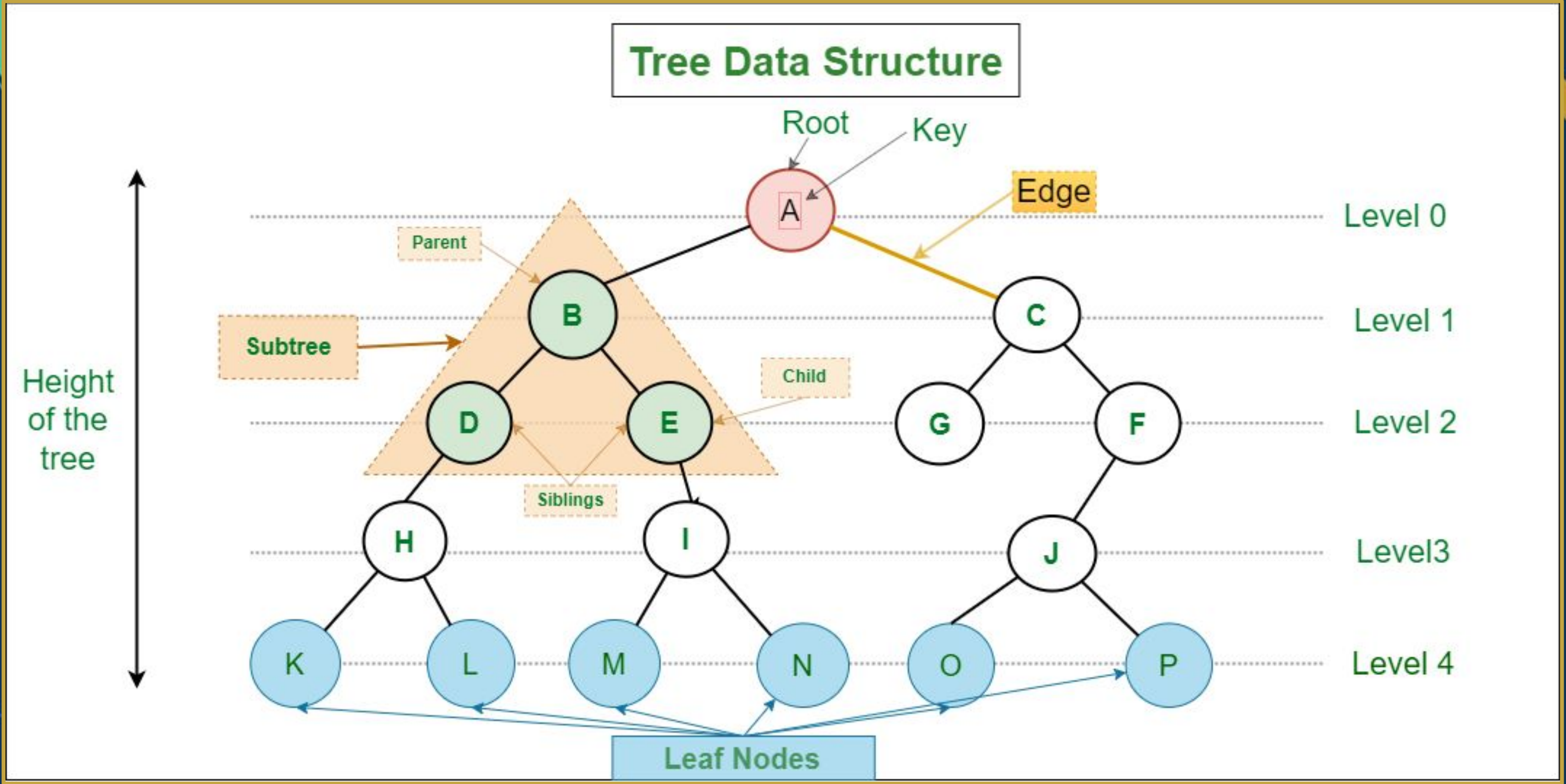
1. Tree Fundamentals
2. Binary Trees
3. Binary Search Trees (BSTs)
4. AVL Trees
5. Heap Trees
6. Tree Traversal

CoGrammar

Tree Fundamentals

A tree is a non-linear data structure consisting of nodes connected by edges

- Components of a tree:
  - **Nodes:** Data elements in the tree
  - **Edges:** Connections between nodes
  - **Root:** The topmost node in the tree
  - **Leaves:** Nodes without child nodes
  - **Depth:** Number of edges from the root to a node
  - **Height:** Maximum depth of the tree



Source: https://www.geeksforgeeks.org/introduction-to-tree-data-structure-and-algorithm-tutorials/

CoGrammar

Tree Fundamentals

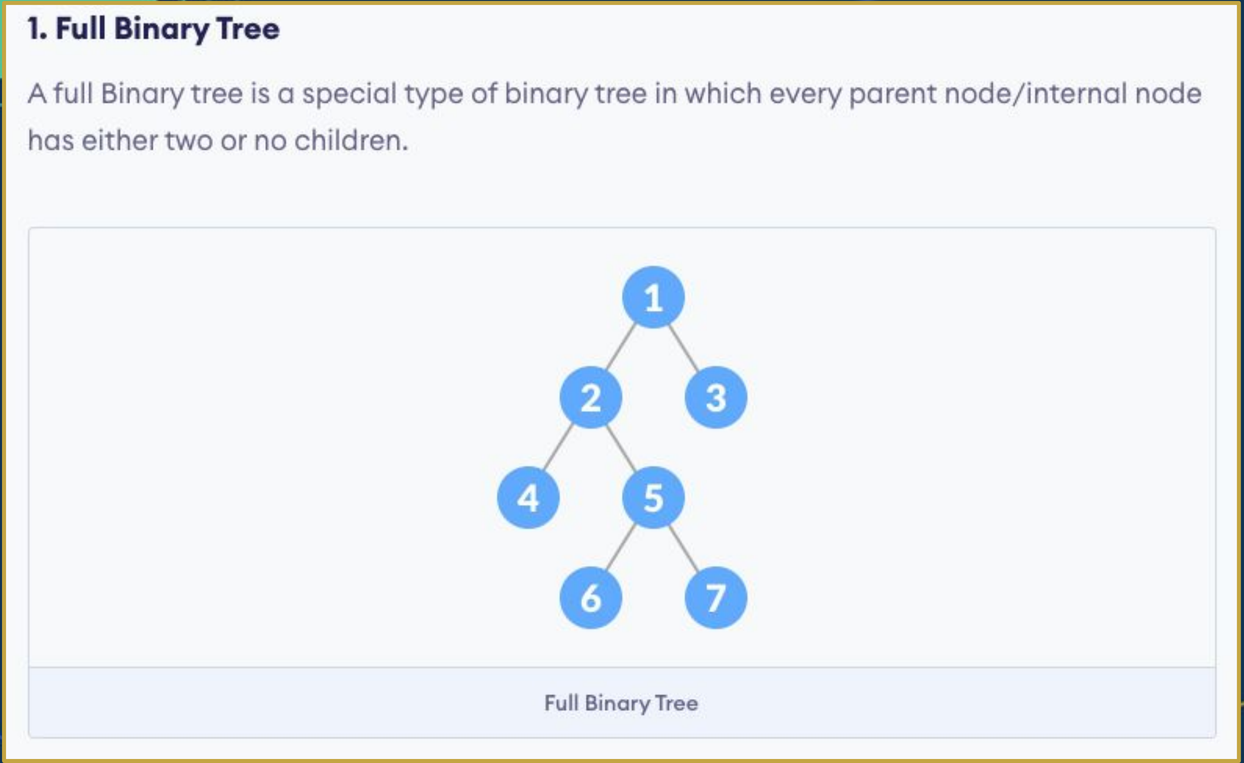
- Properties of trees:
  - Each node (except the root) has **exactly one parent node**
  - The tree is **acyclic (no cycles)**
  - There is a **unique path from the root to each node**
- Trees are **recursive data structures**:
  - Each node can be treated as the root of a subtree

CoGrammar

Binary Trees

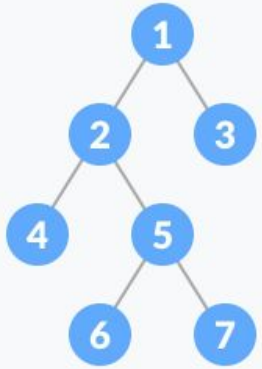
A binary tree is a tree in which each node has at most two child nodes (left and right)

- Properties:
  - There is **no specific ordering or relationship between the values of the nodes**.
  - The **left and right subtrees of a node can have any structure and are not necessarily balanced.**



1. Full Binary Tree

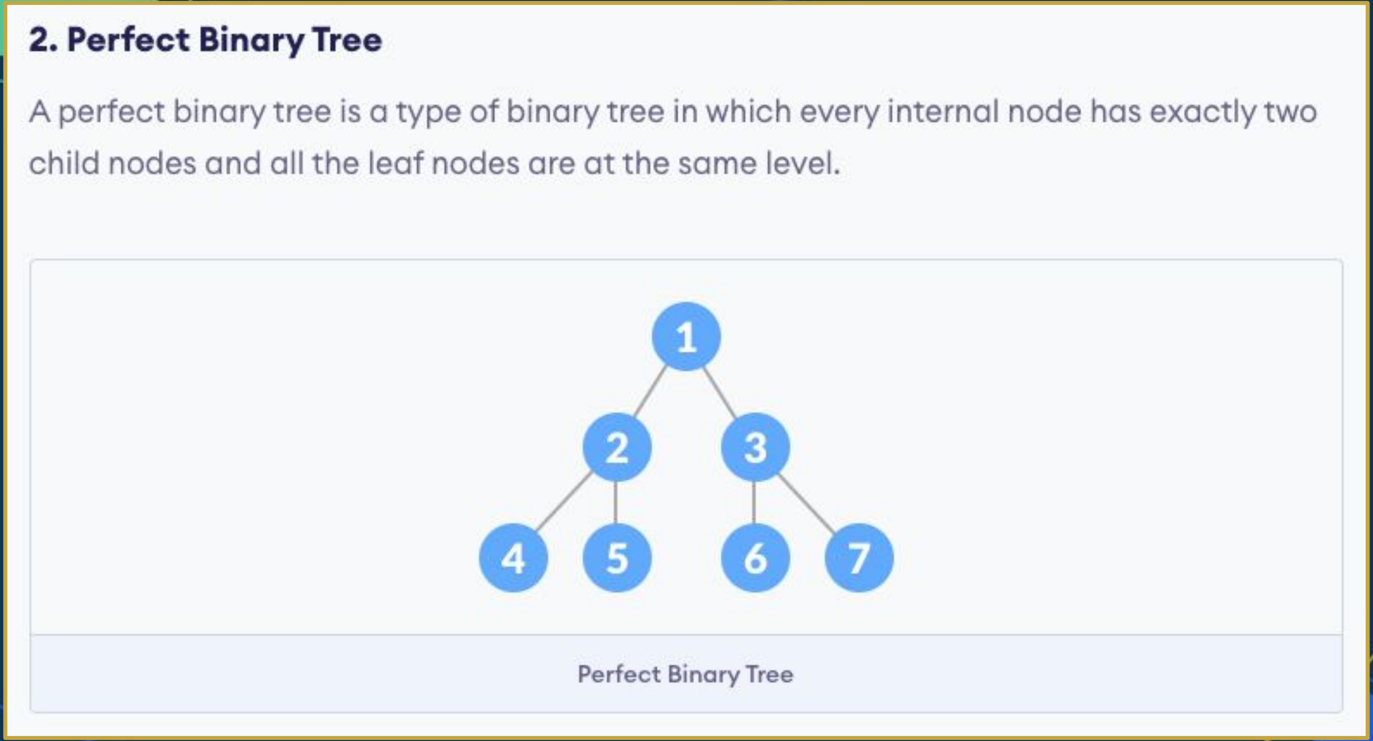
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.



Full Binary Tree

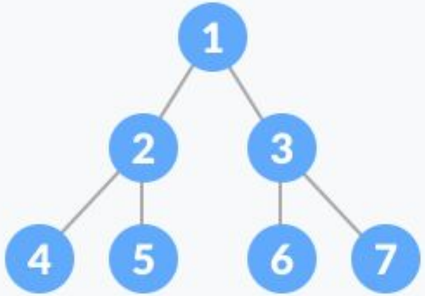
Source: https://www.programiz.com/dsa/binary-tree

CoGrammar



2. Perfect Binary Tree

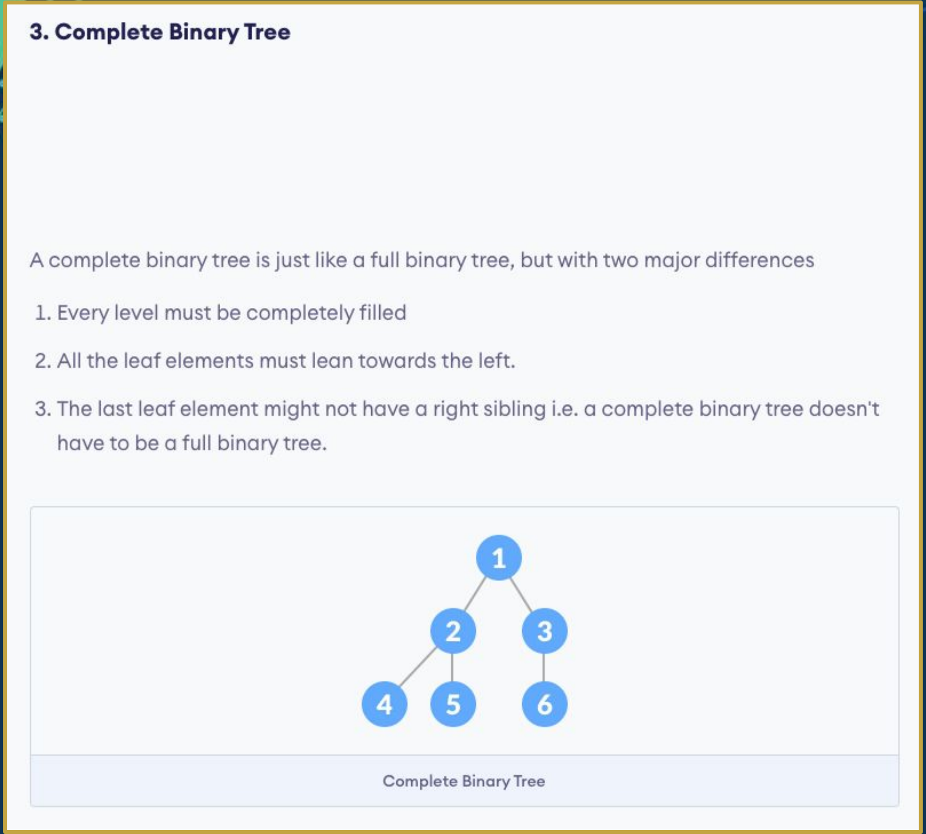
A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.



Perfect Binary Tree

Source: https://www.programiz.com/dsa/binary-tree

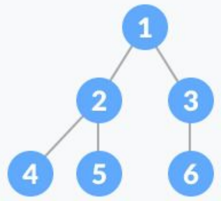
CoGrammar



3. Complete Binary Tree

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

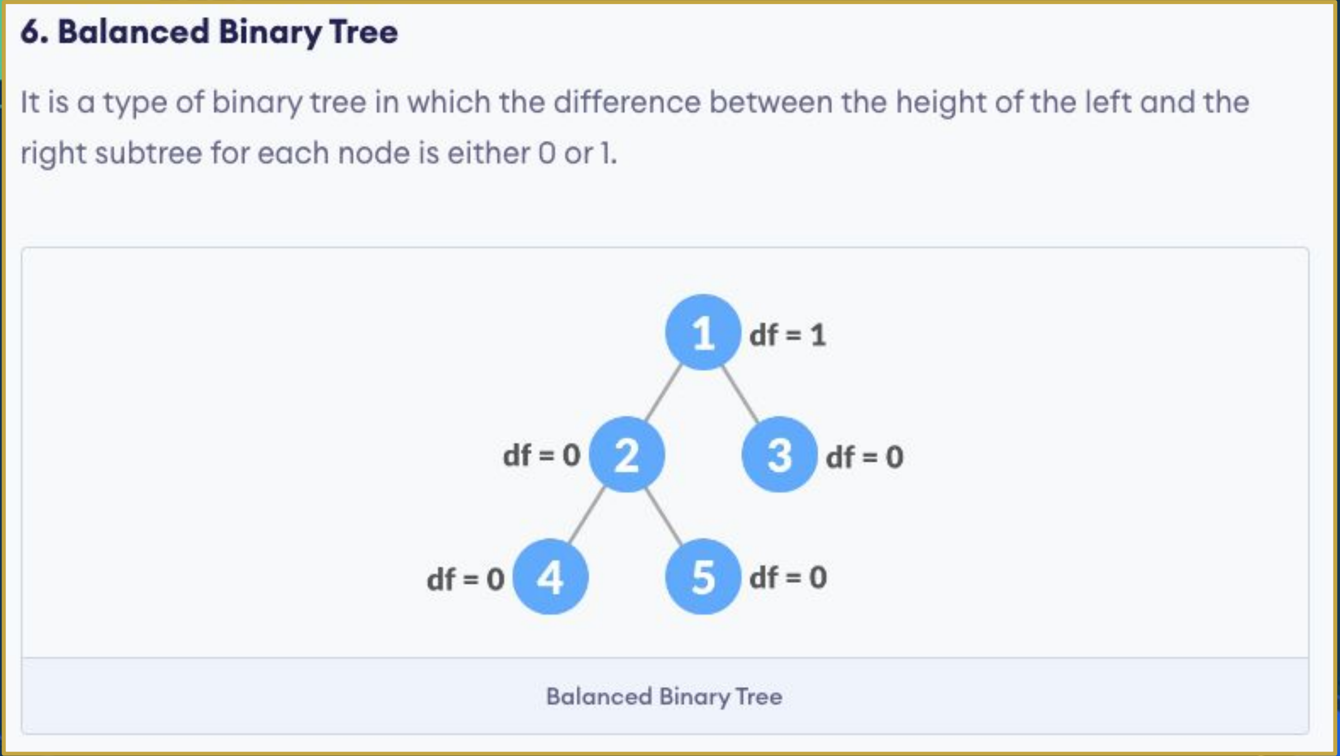
1. Every level must be completely filled
2. All the leaf elements must lean towards the left.
3. The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Complete Binary Tree

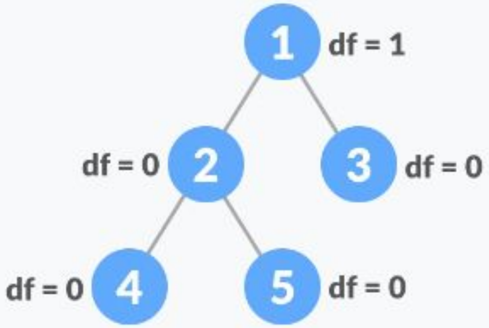
Source: https://www.programiz.com/dsa/binary-tree

CoGrammar



6. Balanced Binary Tree

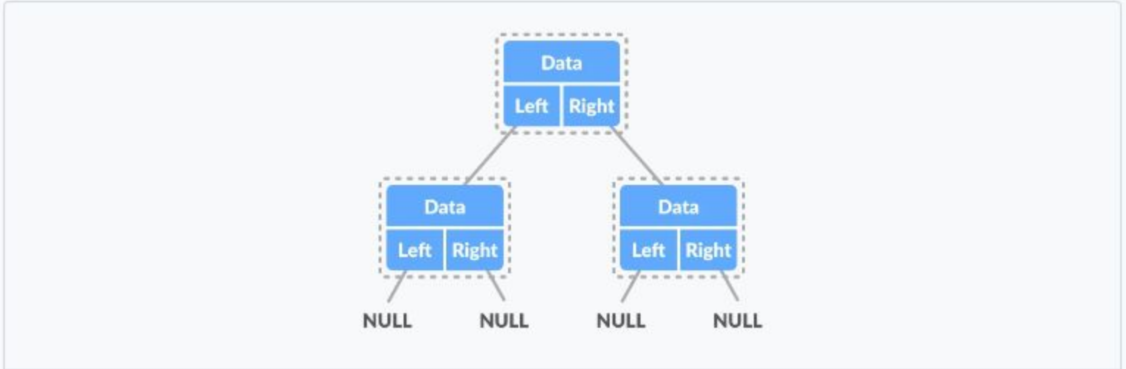
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Balanced Binary Tree

Source: https://www.programiz.com/dsa/binary-tree

CoGrammar



Binary Tree Representation

Source: https://www.programiz.com/dsa/binary-tree

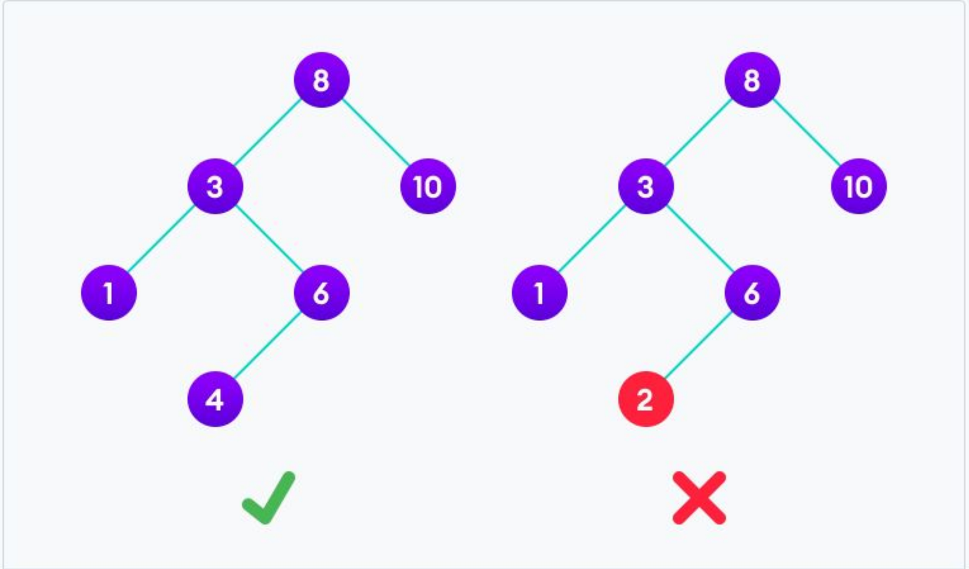
```python
# Binary Tree Node
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

```javascript
// Binary Tree Node
class TreeNode {
  constructor(val = 0, left = null, right = null) {
    this.val = val;
    this.left = left;
    this.right = right;
  }
}
```

CoGrammar

Binary Search Trees

- ❖ A binary search tree (BST) is a binary tree with the following properties:
  - ➢ The **left subtree of a node contains only nodes with keys less than the node's key**
  - ➢ The **right subtree of a node contains only nodes with keys greater than the node's key**
  - ➢ Both the left and right subtrees must also be binary search trees



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

Source: https://www.programiz.com/dsa/binary-search-tree

CoGrammar

Binary Search Trees

- ❖ Operations:
  - ➢ **Insertion:** Adding a new node to the tree while maintaining the BST properties
  - ➢ **Deletion:** Removing a node from the tree while maintaining the BST properties
  - ➢ **Search:** Finding a node with a specific key in the tree

CoGrammar

```python
# Binary Search Tree Insertion
def insert(root, val):
    if not root:
        return TreeNode(val)
    if val < root.val:
        root.left = insert(root.left, val)
    else:
        root.right = insert(root.right, val)
    return root
```

```javascript
// Binary Search Tree Insertion
function insert(root, val) {
  if (!root) {
    return new TreeNode(val);
  }
  if (val < root.val) {
    root.left = insert(root.left, val);
  } else {
    root.right = insert(root.right, val);
  }
  return root;
}
```

CoGrammar

Binary Search Trees

- Time complexity:
  - **Insertion, deletion, and search: O(h)**, where h is the height of the tree
  - In a balanced BST, **h = O(log n)**, where n is the number of nodes

CoGrammar

AVL Trees

An AVL tree is a self-balancing binary search tree

- ❖ Balancing property:
  - ➢ The heights of the **left and right subtrees of any node differ by at most one**
  - ➢ This property ensures that the tree **remains balanced**, preventing degeneration into a linked list

CoGrammar

AVL Trees

❖ Rotations:
  ➢ Used to **rebalance the tree when the AVL property is violated after an insertion or deletion**
  ➢ Left rotation and right rotation
❖ Applications:
  ➢ Efficient searching and sorting (since the tree is balanced)

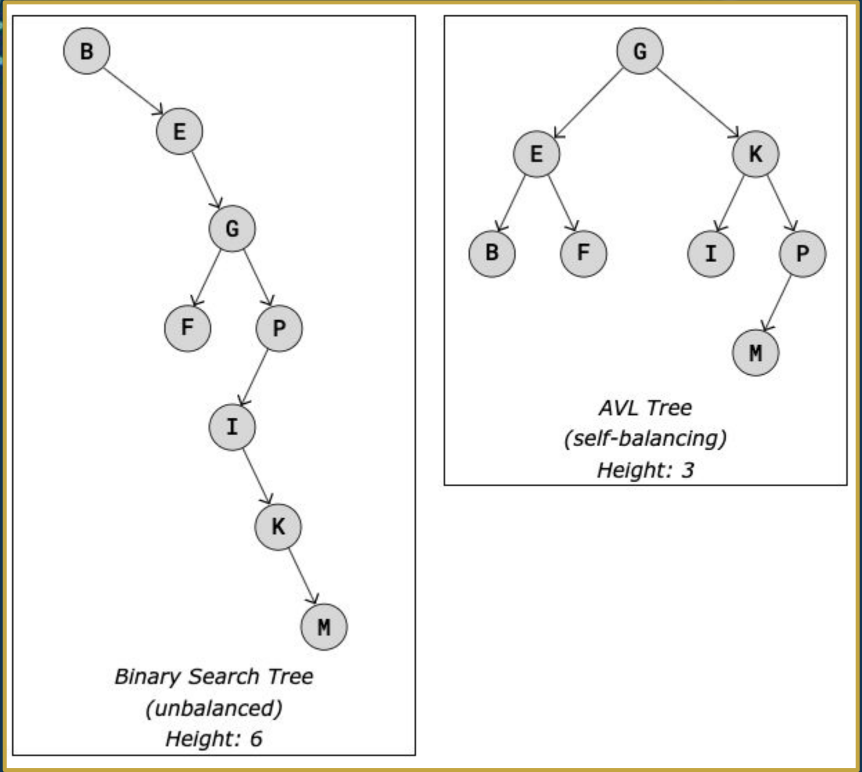
CoGrammar

AVL Trees

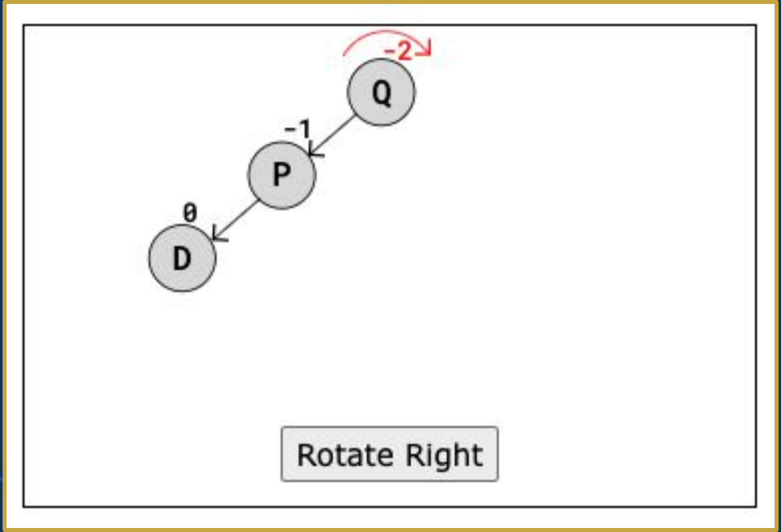
❖ The code for AVL trees could become quite long due to the balancing property, so feel free to check it out

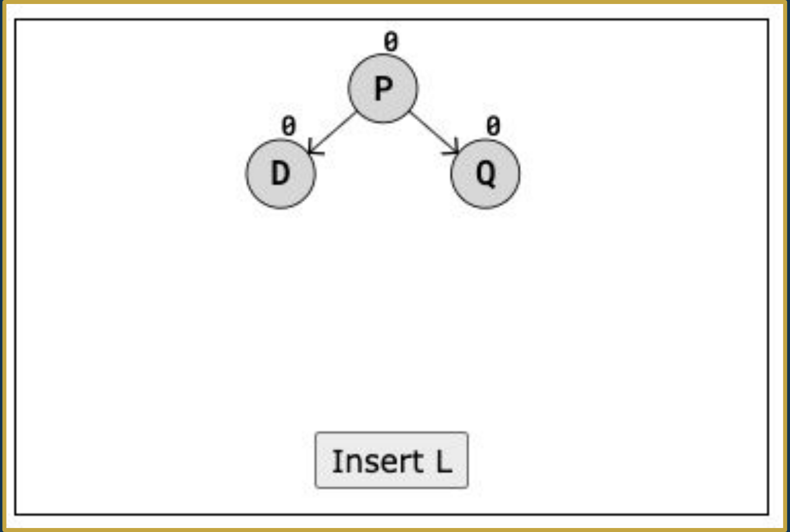
- ➢ Python: https://github.com/bfaure/Python3_Data_Structures/blob/master/AVL_Tree/main.py
- ➢ Javascript: https://github.com/gwtw/js-avl-tree/tree/master/src

CoGrammar



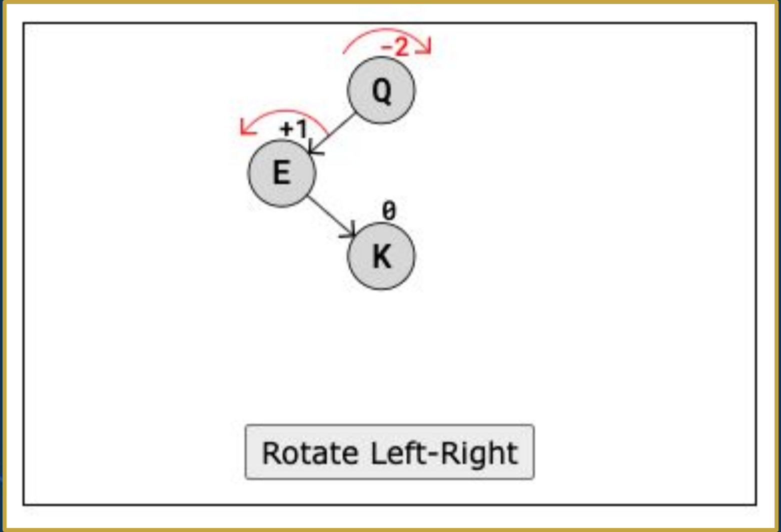
Source: https://www.w3schools.com/dsa/dsa_data_avltrees.php

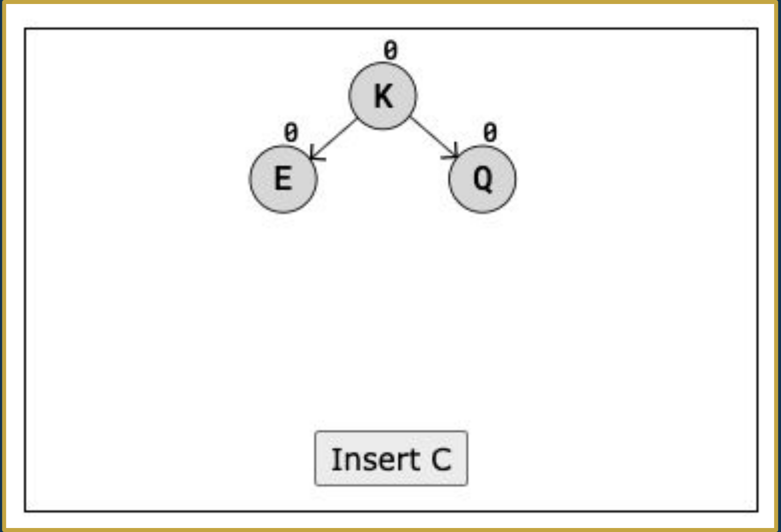




Source: https://www.w3schools.com/dsa/dsa_data_avltrees.php

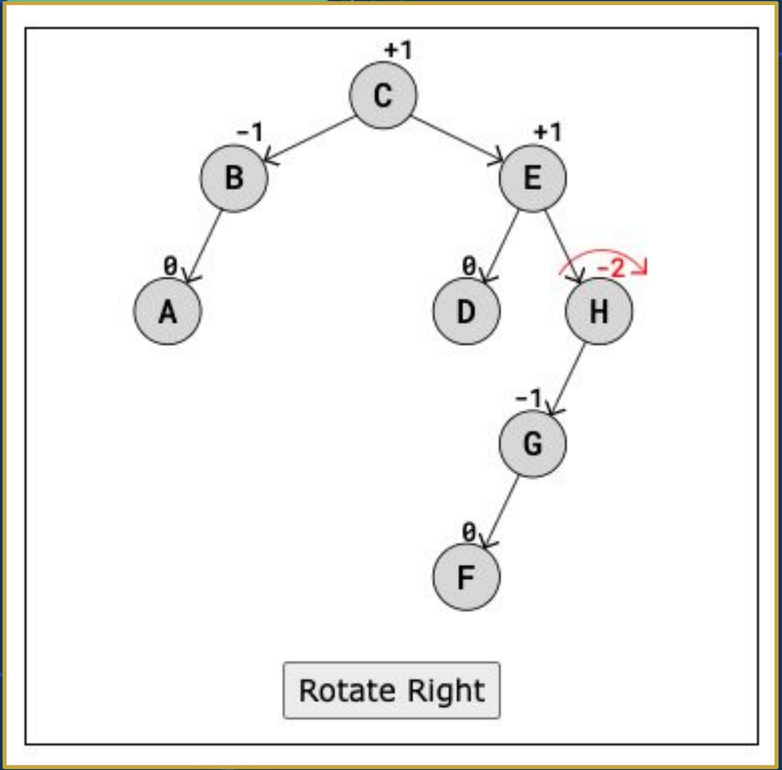
CoGrammar

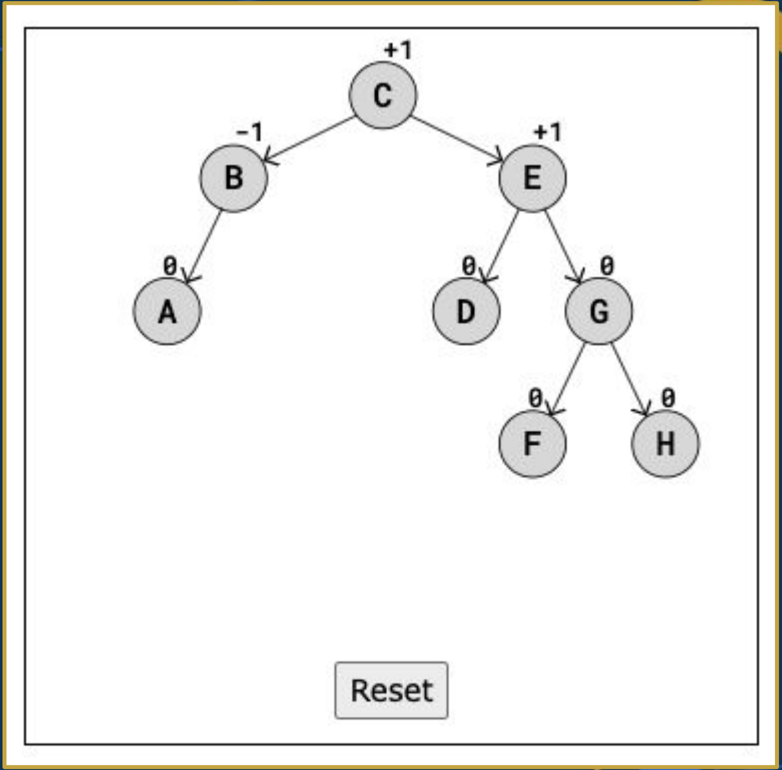




Source: https://www.w3schools.com/dsa/dsa_data_avltrees.php

CoGrammar





Source: https://www.w3schools.com/dsa/dsa_data_avltrees.php

Heap

A heap is a complete binary tree that satisfies the heap property

❖ Heap property:

  ➢ For a **min-heap: parent(i) <= i**

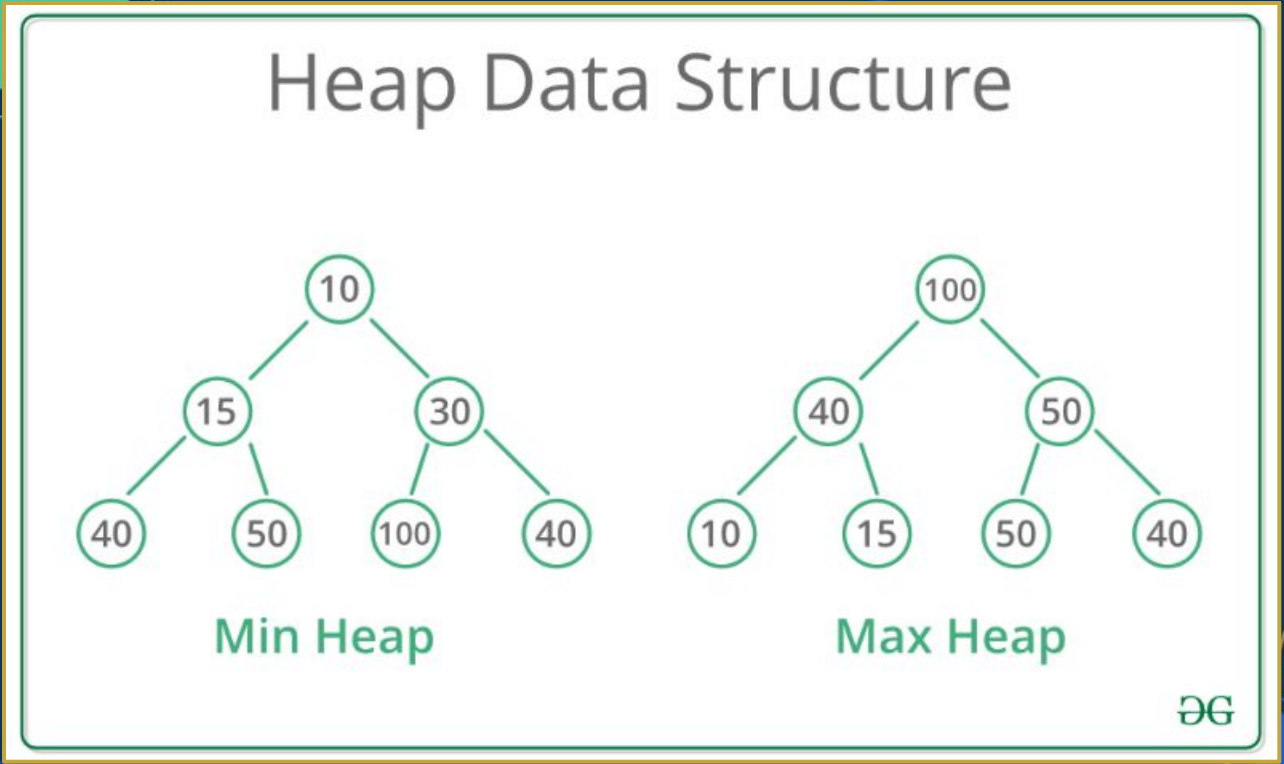
  ➢ For a **max-heap: parent(i) >= i**

CoGrammar

Heap

- ❖ Applications:
  - ➢ Priority queues
  - ➢ Heapsort algorithm

CoGrammar



Source: https://www.geeksforgeeks.org/heap-data-structure/

What is the maximum number of children a node can have in a binary tree?

A. 1

B. 2

C. 3

D. 4

CoGrammar

Answer: B

- In a binary tree, each node can have at most two children, referred to as the left child and the right child.

CoGrammar

What is the time complexity of searching for a value in a balanced binary search tree?

A. O(1)

B. O(log n)

C. O(n)

D. O(n^2)

CoGrammar

Answer: B

- In a balanced BST, the search operation has a time complexity of O(log n) because the tree's height is maintained as log n, and the search algorithm eliminates half of the remaining nodes at each step.

Which of the following is true about a min-heap?

A. The root node has the minimum value in the heap

B. The root node has the maximum value in the heap

C. The heap property is: parent(i) >= i

D. The heap property is: parent(i) > i

Answer: A

- ❖ In a min-heap, the heap property ensures that the value of each node is greater than or equal to the value of its parent node. Consequently, the root node always contains the minimum value in the heap.

CoGrammar

Let’s Breathe!

Let’s take a small break before moving on to the next topic.

CoGrammar

Tree Traversal

- ❖ Tree traversal is the process of **visiting each node in a tree exactly once**
- ❖ Types of tree traversal:
  - ➢ **In-order traversal (Depth-first Search)**
  - ➢ **Pre-order traversal (Depth-first Search)**
  - ➢ **Post-order traversal (Depth-first Search)**
  - ➢ **Level-order traversal (Breadth-first Search)**

CoGrammar

Tree Traversal

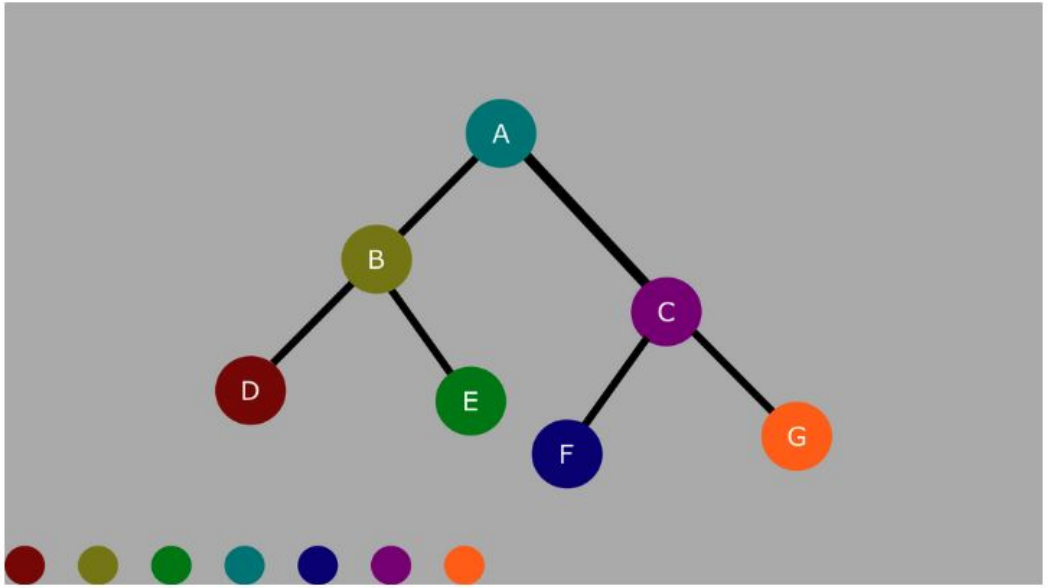
- **In-order traversal** visits nodes in the following order:
  - **Left subtree**
  - **Root**
  - **Right subtree**
- Useful for:
  - Visiting nodes in sorted order (for BSTs)
  - Creating a copy of the tree

CoGrammar

```python
# In-order Traversal
def inorder_traversal(root):
    if root:
        inorder_traversal(root.left)
        print(root.val)
        inorder_traversal(root.right)
```

```javascript
// In-order Traversal
function inorderTraversal(root) {
  if (root) {
    inorderTraversal(root.left);
    console.log(root.val);
    inorderTraversal(root.right);
  }
}
```

CoGrammar



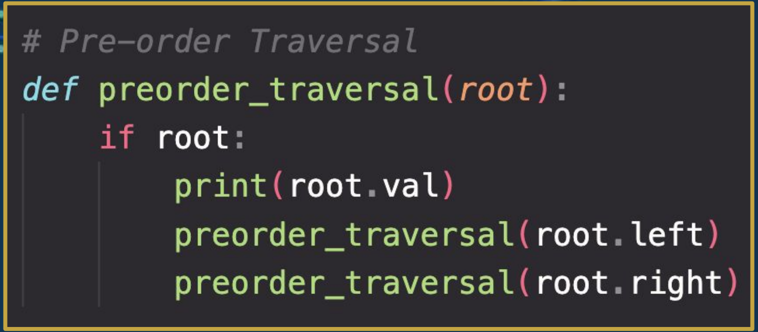
Inorder traversal

Source: https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/

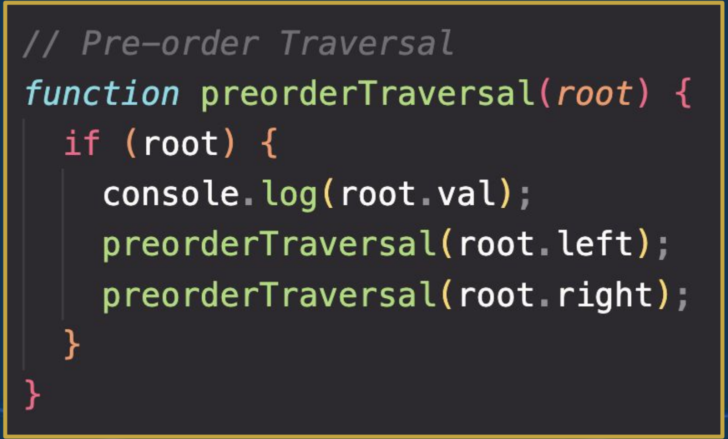
CoGrammar

Tree Traversal

- **Pre-order traversal** visits nodes in the following order:
  - **Root**
  - **Left subtree**
  - **Right subtree**
- Useful for:
  - Creating a copy of the tree
  - Prefix expression evaluation

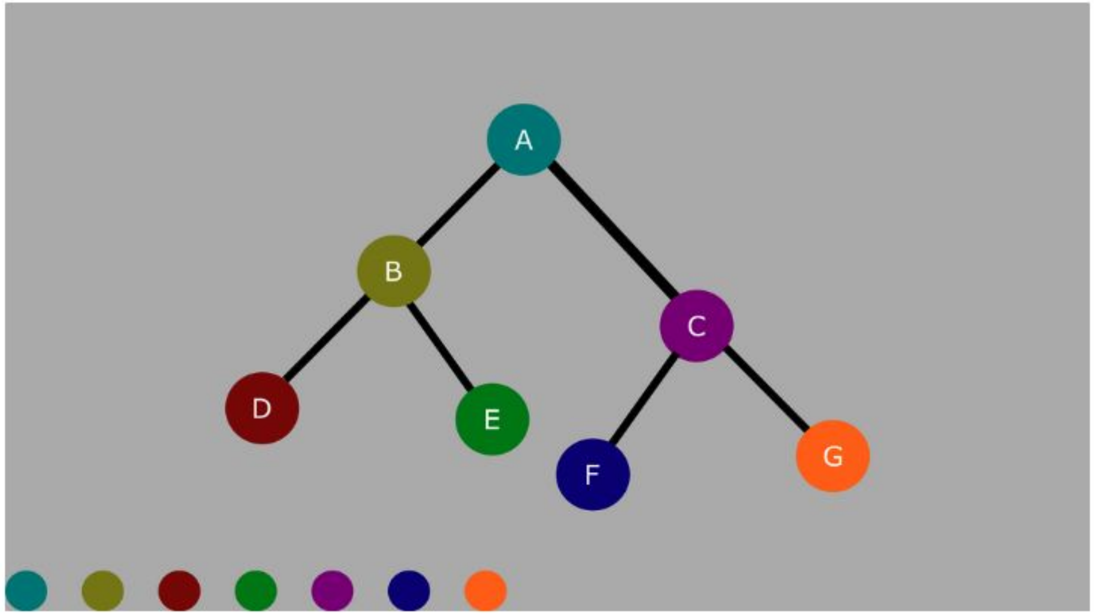


```python
# Pre-order Traversal
def preorder_traversal(root):
    if root:
        print(root.val)
        preorder_traversal(root.left)
        preorder_traversal(root.right)
```



```javascript
// Pre-order Traversal
function preorderTraversal(root) {
  if (root) {
    console.log(root.val);
    preorderTraversal(root.left);
    preorderTraversal(root.right);
  }
}
```

CoGrammar



Preorder traversal

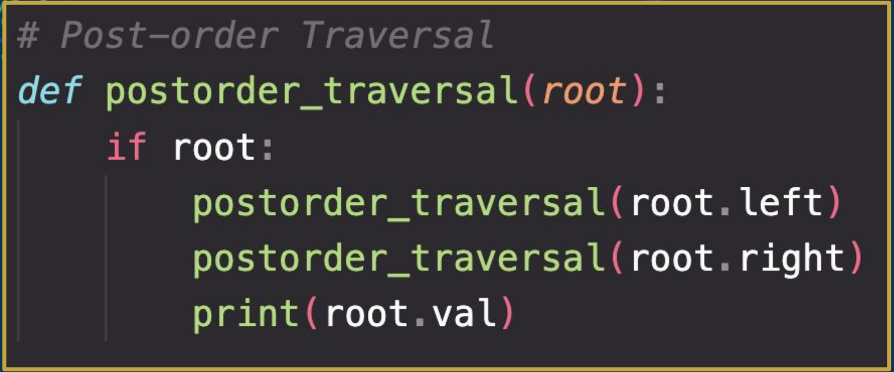
Source: https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/

CoGrammar

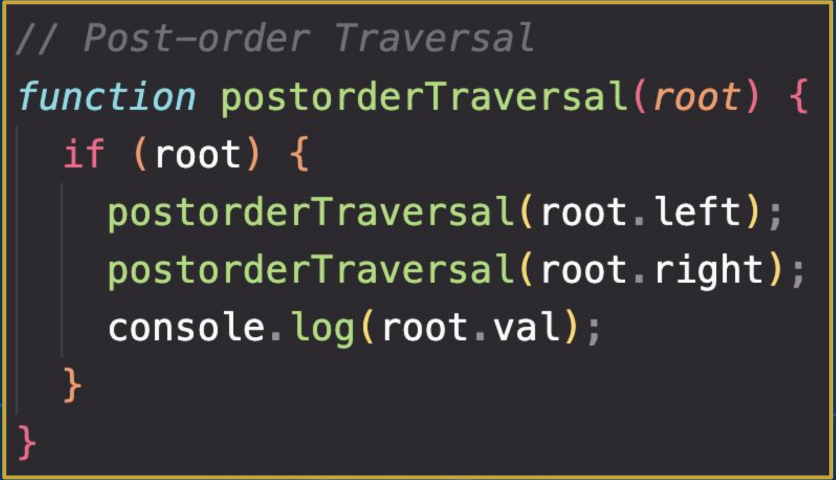
Tree Traversal

- **Post-order traversal** visits nodes in the following order:
  - **Left subtree**
  - **Right subtree**
  - **Root**
- Useful for:
  - Deleting a tree
  - Postfix expression evaluation

CoGrammar

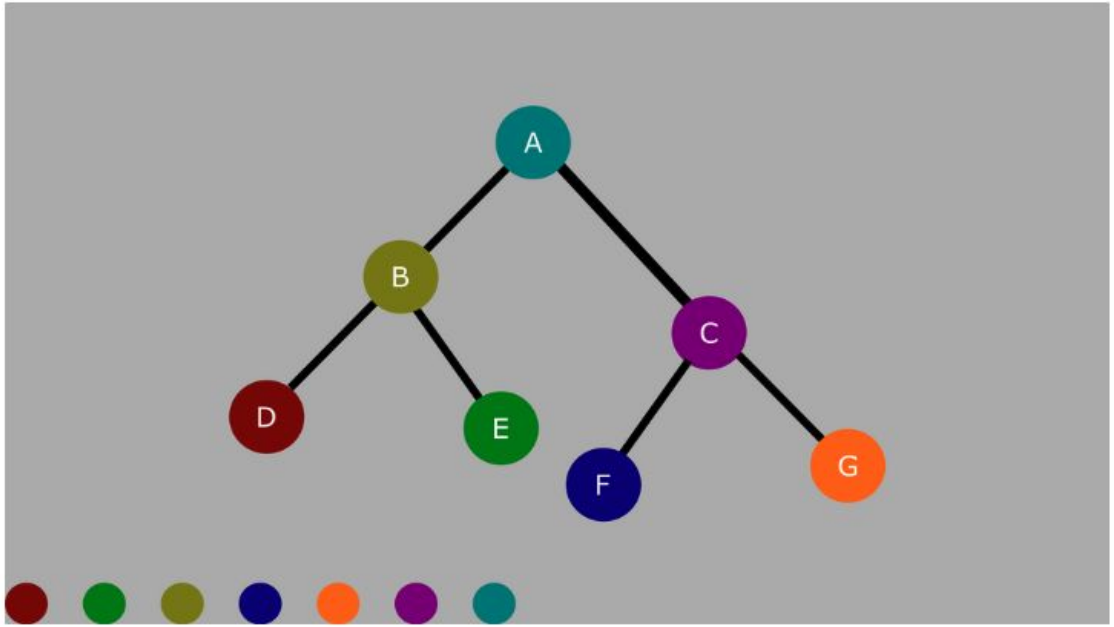


```python
# Post-order Traversal
def postorder_traversal(root):
    if root:
        postorder_traversal(root.left)
        postorder_traversal(root.right)
        print(root.val)
```



```javascript
// Post-order Traversal
function postorderTraversal(root) {
  if (root) {
    postorderTraversal(root.left);
    postorderTraversal(root.right);
    console.log(root.val);
  }
}
```

CoGrammar



Postorder traversal

Source: https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/

Tree Traversal

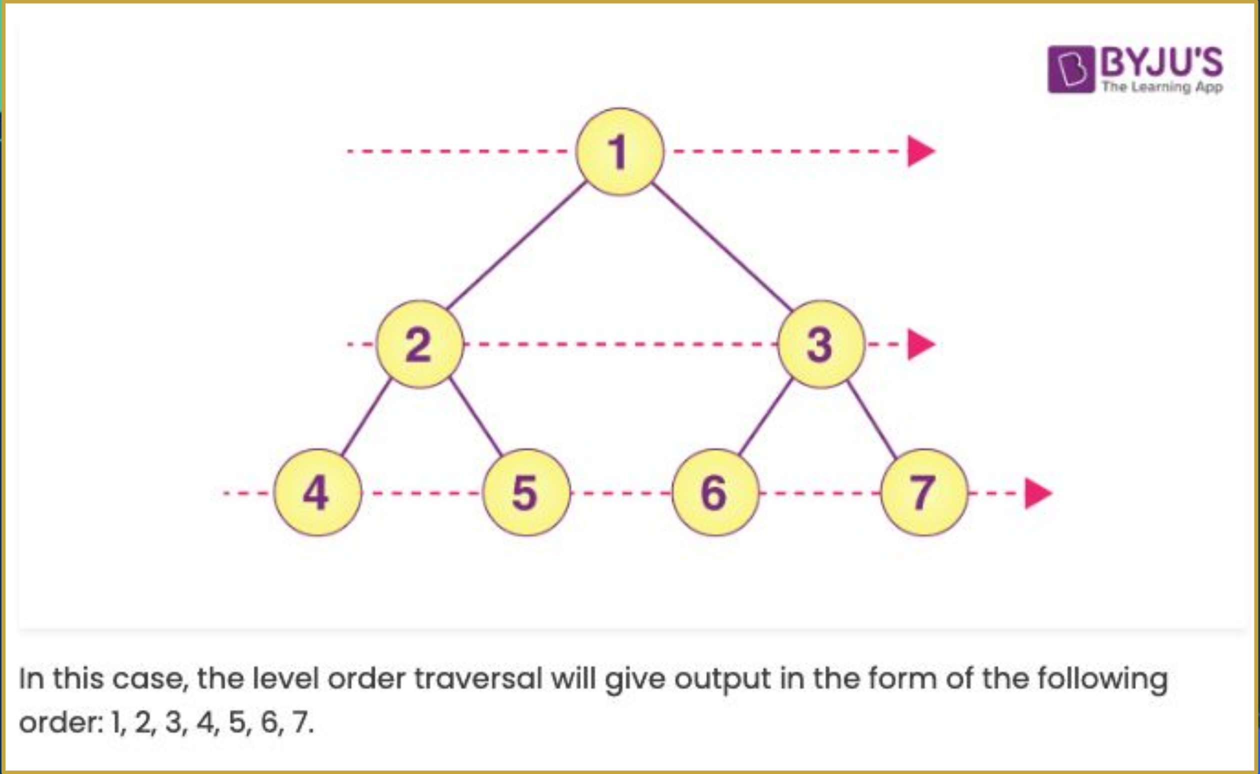
- **Level-order traversal visits nodes level by level, from left to right**
- Useful for:
  - Printing the tree level by level
  - Finding the shortest path between two nodes

CoGrammar

```python
# Level-order Traversal
from collections import deque

def level_order_traversal(root):
    if not root:
        return
    queue = deque([root])
    while queue:
        node = queue.popleft()
        print(node.val)
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
```

```javascript
// Level-order Traversal
function levelOrderTraversal(root) {
  if (!root) {
    return;
  }
  const queue = [root];
  while (queue.length) {
    const node = queue.shift();
    console.log(node.val);
    if (node.left) {
      queue.push(node.left);
    }
    if (node.right) {
      queue.push(node.right);
    }
  }
}
```



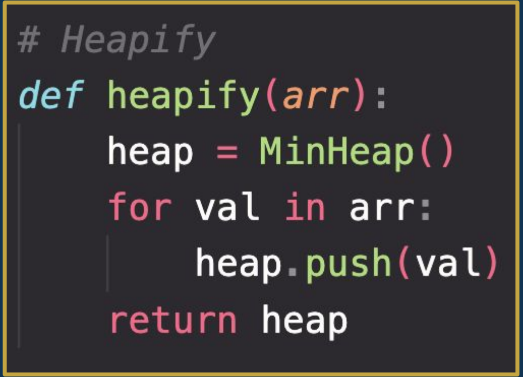
In this case, the level order traversal will give output in the form of the following order: 1, 2, 3, 4, 5, 6, 7.

Source: https://byjus.com/gate/tree-traversal-notes/

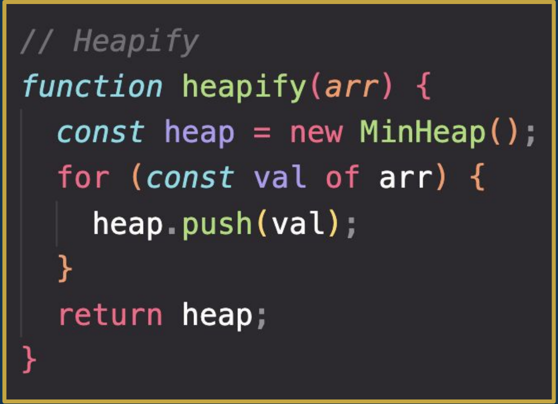
Heap Construction

- To construct a heap from an array of elements:
  - Insert each element into the heap one by one
  - After each insertion, sift up the new element to maintain the heap property
- This process is called **heapify**
- **Time complexity: O(n log n)** for binary heap since n/2 non-leaf nodes require the heapify operation in worst case and heapify is O(log n), so n/2 * O(log n) = O(n/2 log n) = O(n log n)

CoGrammar



```python
# Heapify
def heapify(arr):
    heap = MinHeap()
    for val in arr:
        heap.push(val)
    return heap
```

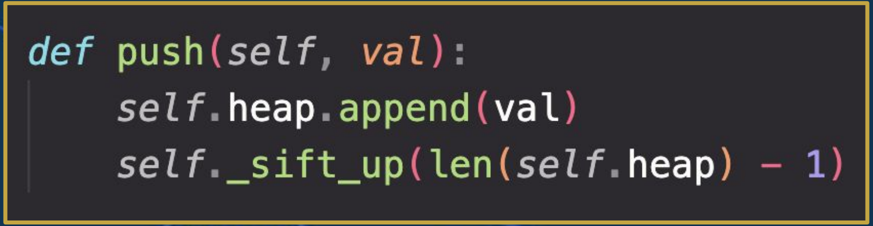


```javascript
// Heapify
function heapify(arr) {
  const heap = new MinHeap();
  for (const val of arr) {
    heap.push(val);
  }
  return heap;
}
```

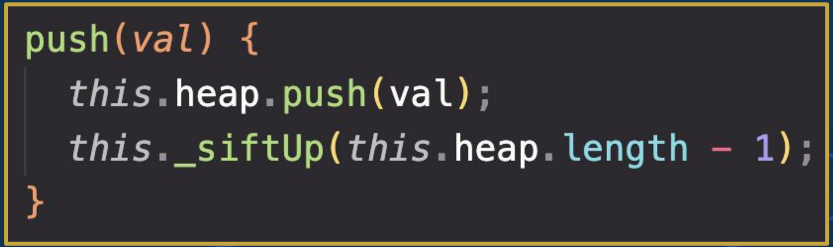
CoGrammar

Heap Operations

- **Insertion:**
  - Add a new element to the end of the heap
  - Sift up the new element to maintain the heap property
  - **Time complexity: O(log n)**



```python
def push(self, val):
    self.heap.append(val)
    self._sift_up(len(self.heap) - 1)
```

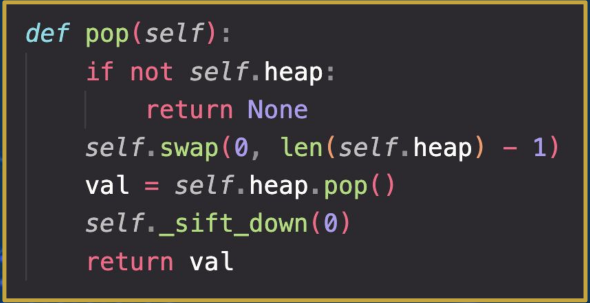


```javascript
push(val) {
  this.heap.push(val);
  this._siftUp(this.heap.length - 1);
}
```

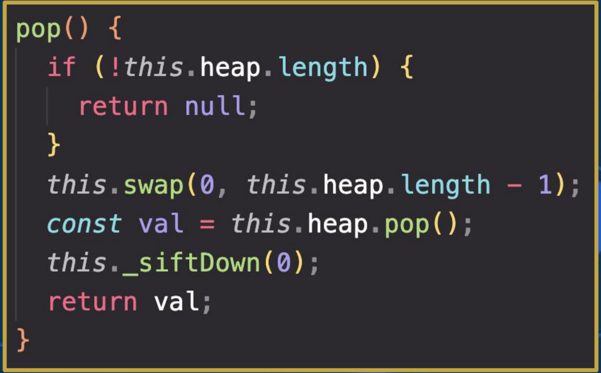
CoGrammar

Heap Operations

- **Deletion** (extracting the minimum or maximum element):
  - Replace the root with the last element in the heap
  - Remove the last element
  - Sift down the new root to maintain the heap property
  - Time complexity: **O(log n)**



```python
def pop(self):
    if not self.heap:
        return None
    self.swap(0, len(self.heap) - 1)
    val = self.heap.pop()
    self._sift_down(0)
    return val
```



```javascript
pop() {
  if (!this.heap.length) {
    return null;
  }
  this.swap(0, this.heap.length - 1);
  const val = this.heap.pop();
  this._siftDown(0);
  return val;
}
```

Interview-Style Questions: Social Network

On a social platform like Facebook, you can connect with friends, and each of your friends can also have their own friends. If we consider that both you and each of your friends can have at most 2 friends in total, how many levels of connections beyond your direct friends can you explore?

CoGrammar

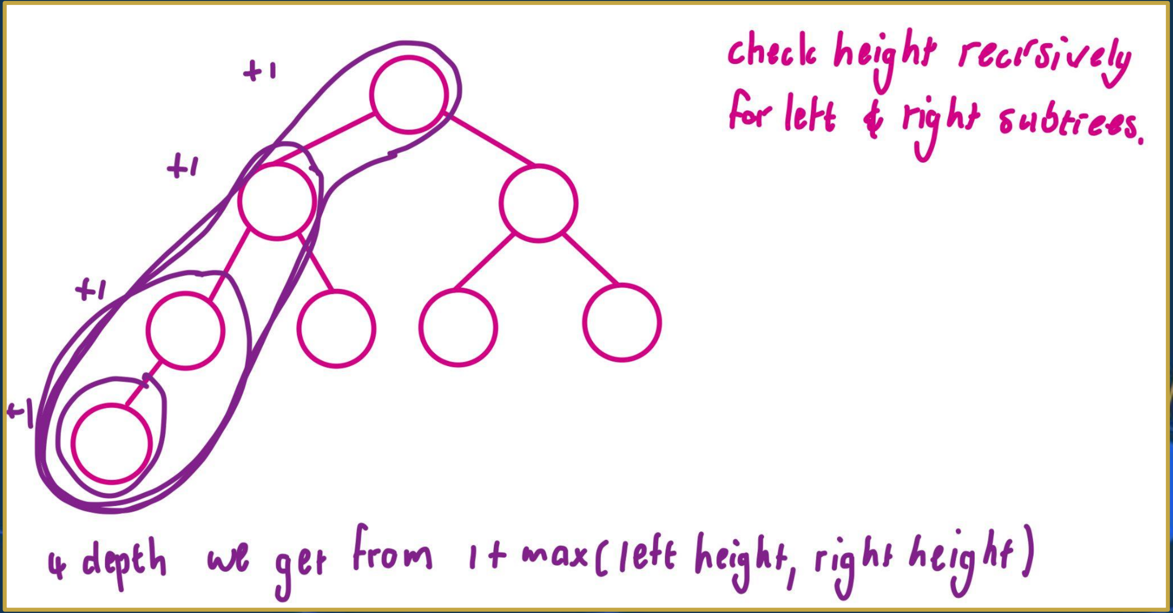
Interview-Style Questions

- ❖ Find the height of a binary tree
- ❖ Check if a binary tree is a BST
- ❖ Find the kth smallest element in a BST

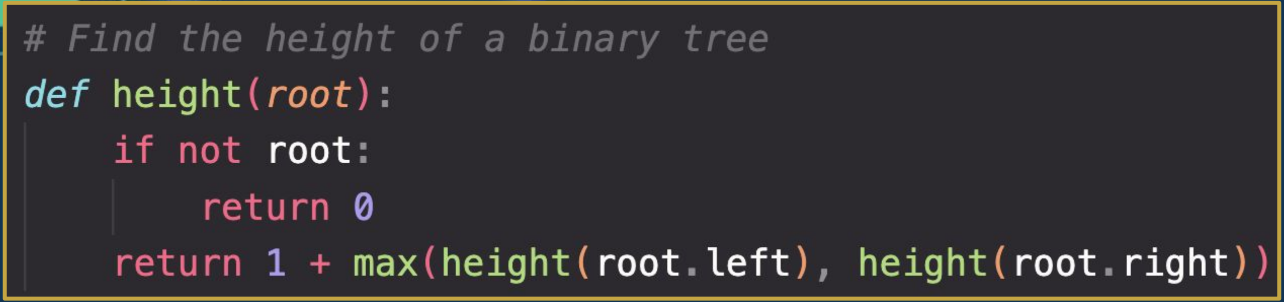
CoGrammar

Interview-Style Question 1

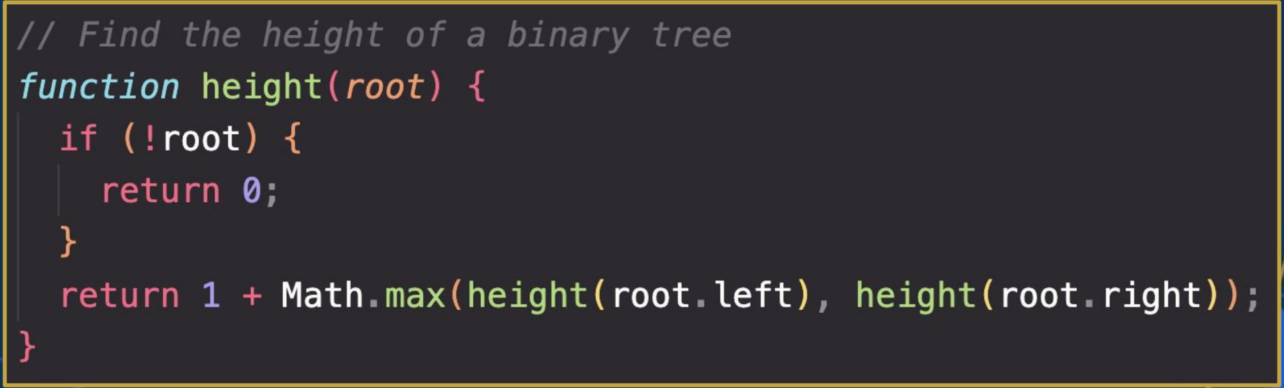
❖ Given the root of a binary tree, find its height (maximum depth)


check height recursively
for left & right subtrees.
+1
+1
+1
+1
4 depth we get from 1+max(left height, right height)

CoGrammar



```python
# Find the height of a binary tree
def height(root):
    if not root:
        return 0
    return 1 + max(height(root.left), height(root.right))
```



```javascript
// Find the height of a binary tree
function height(root) {
  if (!root) {
    return 0;
  }
  return 1 + Math.max(height(root.left), height(root.right));
}
```

Interview-Style Question 2

- ❖ Given the root of a binary tree, determine if it is a valid binary search tree (BST)
  - ➢ Basically, are all left nodes less in value than the right nodes
  - ➢ We can just check this for the left and right subtrees recursively

CoGrammar

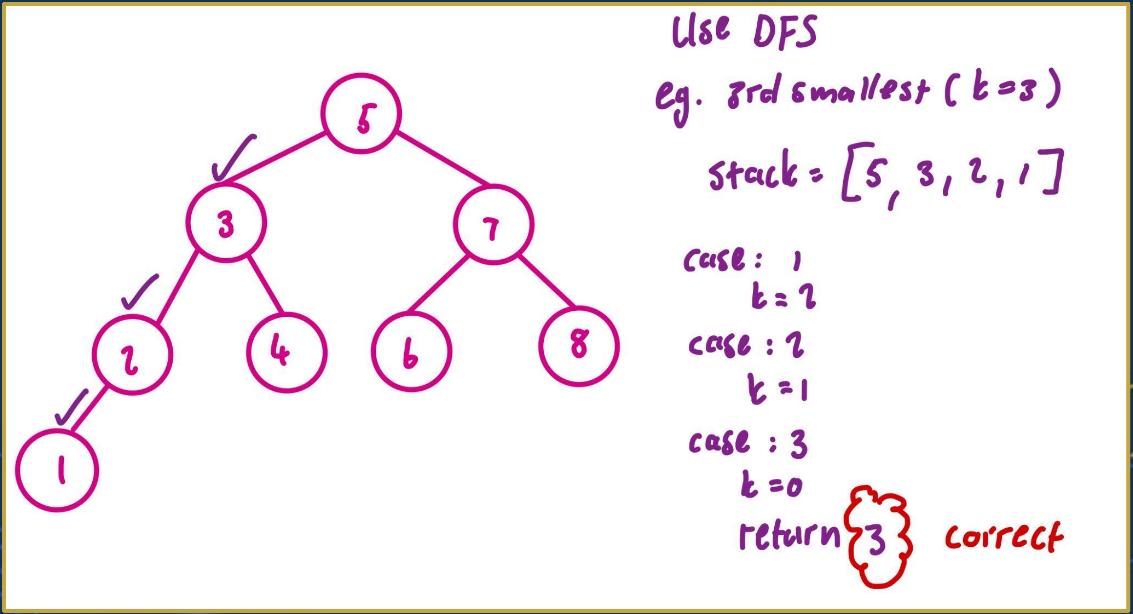
```python
# Check if a binary tree is a BST
def is_bst(root, min_val=float('-inf'), max_val=float('inf')):
    if not root:
        return True
    if root.val <= min_val or root.val >= max_val:
        return False
    return is_bst(root.left, min_val, root.val) and is_bst(root.right, root.val, max_val)
```

```javascript
// Check if a binary tree is a BST
function isBST(root, minVal = -Infinity, maxVal = Infinity) {
  if (!root) {
    return true;
  }
  if (root.val <= minVal || root.val >= maxVal) {
    return false;
  }
  return (
    isBST(root.left, minVal, root.val) && isBST(root.right, root.val, maxVal)
  );
}
```

CoGrammar

Interview-Style Question 3

- ❖ Given the root of a binary search tree and an integer k, find the kth smallest element in the BST



Use DFS

eg. 3rd smallest (k=3)

stack = [5, 3, 2, 1]

case : 1
k = 2

case : 2
k = 1

case : 3
k = 0

return 3 correct

CoGrammar

```python
# Find the kth smallest element in a BST
def kth_smallest(root, k):
    stack = []
    while True:
        while root:
            stack.append(root)
            root = root.left
        root = stack.pop()
        k -= 1
        if not k:
            return root.val
        root = root.right
```

```javascript
// Find the kth smallest element in a BST
function kthSmallest(root, k) {
  const stack = [];
  while (true) {
    while (root) {
      stack.push(root);
      root = root.left;
    }
    root = stack.pop();
    k--;
    if (!k) {
      return root.val;
    }
    root = root.right;
  }
}
```

CoGrammar

Which of the following is the correct order of nodes visited in a post-order traversal of a binary tree?

A. Root, Left, Right

B. Left, Right, Root

C. Right, Left, Root

D. Left, Root, Right

Answer: B

- ❖ In a post-order traversal, the left subtree is visited first, then the right subtree, and finally the root node.

Bonus Interview question

Let’s assume a that you would like to insert data quickly and retrieve it fast. What data structure would you use and why?

Bonus Interview question

What is the complexity of such data structure?

Bonus Interview question

Implement that in your language of choice.

CoGrammar

Bonus Interview question

Let’s add in this order: 2,1,3,4,5,6

Bonus Interview question

Is it still efficient?

Bonus Interview question

What would you do to optimise that?

CoGrammar

CoGrammar

Q & A SECTION

Please use this time to ask any questions relating to the topic, should you have any.

Thank you for attending

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

CoGrammar