



Welcome to this session: Coding Interview Workshop - Trees

The session will start shortly...

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



Safeguarding & Welfare

We are committed to all our students and staff feeling safe and happy; we want to make sure there is always someone you can turn to if you are worried about anything.

If you are feeling upset or unsafe, are worried about a friend, student or family member, or you feel like something isn't right, speak to our safeguarding team:



Ian Wyles
Designated Safeguarding
Lead



Simone Botes



Nurhaan Snyman



Rafiq Manan



Ronald Munodawafa



Tevin Pitts

Scan to report a
safeguarding concern



or email the Designated
Safeguarding Lead:
Ian Wyles

safeguarding@hyperiondev.com

Skills Bootcamp Coding Interview Workshop

- 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**

Skills Bootcamp Coding Interview Workshop

- 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](#)**
- If you are hearing impaired, please kindly use your computer's function through Google chrome to enable captions.

Learning Outcomes

- ❖ **Identify the basic components and properties of trees**, including terms like nodes, edges, roots, leaves, depth, and height.
- ❖ **Differentiate between various types of trees** such as binary trees, binary search trees, AVL trees, and heap trees, focusing on their unique characteristics, uses, and the operations they optimize.
- ❖ **Implement tree operations for BSTs** in Python and JavaScript, including insertion, deletion, and traversal to understand how trees manage data.
- ❖ **Construct and manipulate a heap** to understand its implementation in priority queues and improving the efficiency of sorting algorithms.



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

- A. 1
- B. 2
- C. 3
- D. 4



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. $O(n^2)$

Lecture Overview

- Tree Fundamentals
- Types of Trees
 - ◆ Binary Trees
 - ◆ AVL Trees
 - ◆ BST Trees
 - ◆ Heaps
- Interview Questions

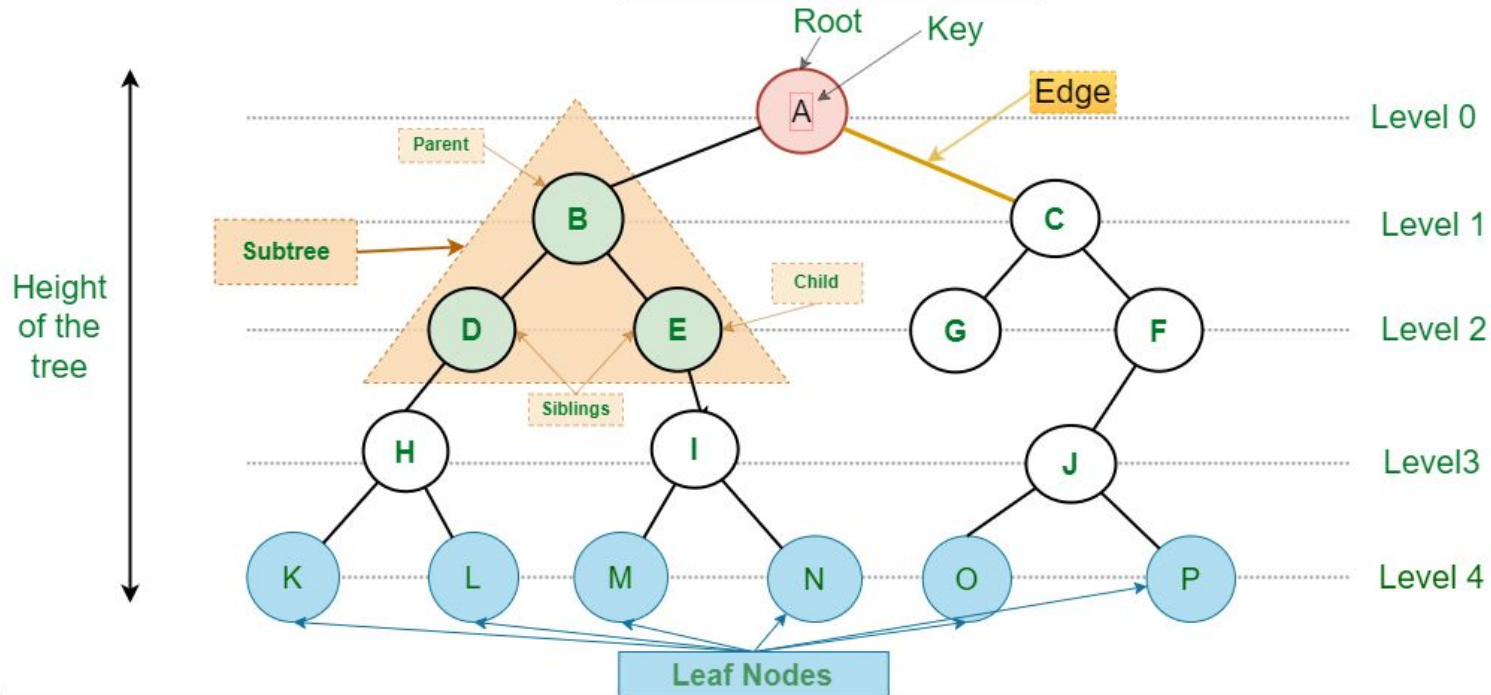


Tree Fundamentals

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.

Tree Data Structure



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



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.

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**.

Binary Tree Node

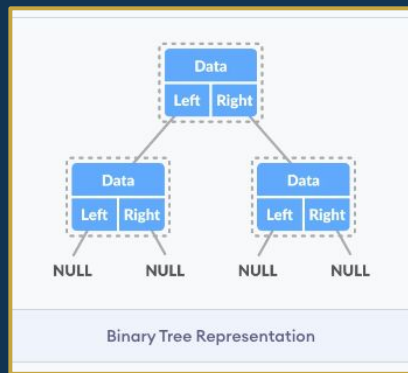
```
class TreeNode:
```

```
    def __init__(self, val=0, left=None, right=None):  
        self.val = val  
        self.left = left  
        self.right = right
```

// Binary Tree Node

```
class TreeNode {
```

```
    constructor(val = 0, left = null, right = null) {  
        this.val = val;  
        this.left = left;  
        this.right = right;
```



Source:

<https://www.programiz.com/dsa/binary-tree>

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.

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).

AVL Trees

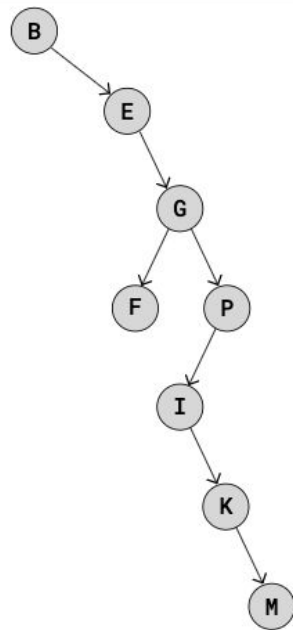
- ❖ The code for AVL trees could become quite long due to the balancing property, so feel free to check it out.

- **Python:**

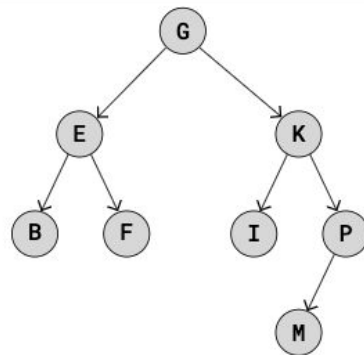
https://github.com/bfaure/Python3_Data_Structures/blob/master/AVL_Tree/main.py

- **Javascript:**

<https://github.com/gwtw/js-avl-tree/tree/master/src>

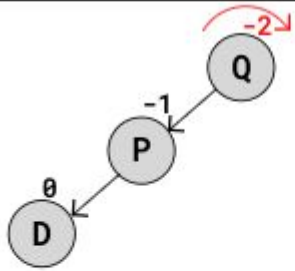


*Binary Search Tree
(unbalanced)
Height: 6*

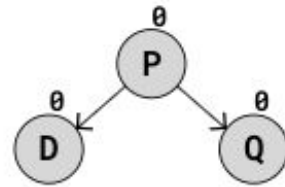


*AVL Tree
(self-balancing)
Height: 3*

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

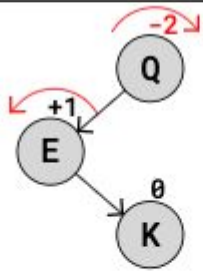


Rotate Right

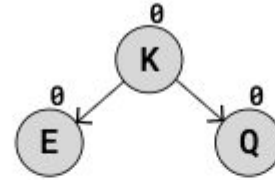


Insert L

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

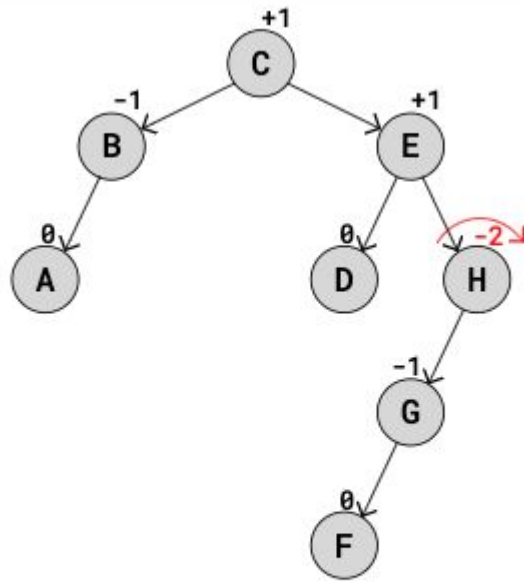


Rotate Left-Right

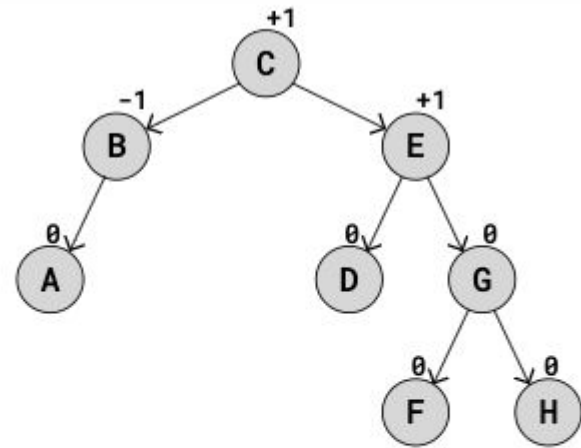


Insert C

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



Rotate Right



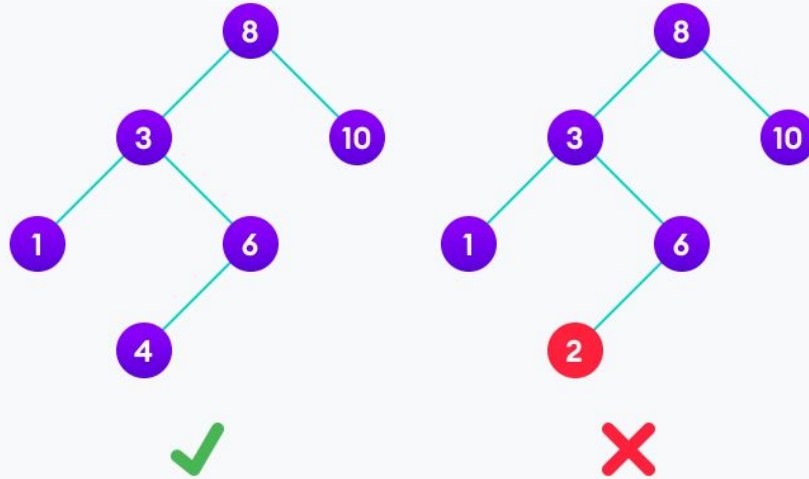
Reset

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



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 BSTs.



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>



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

- ❖ 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

```
# 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
```

```
// 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;
}
```

Let's Breathe!

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



Tree Traversal

- ❖ 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)**

Tree Traversal

❖ **In-order traversal** visits nodes in the following order:

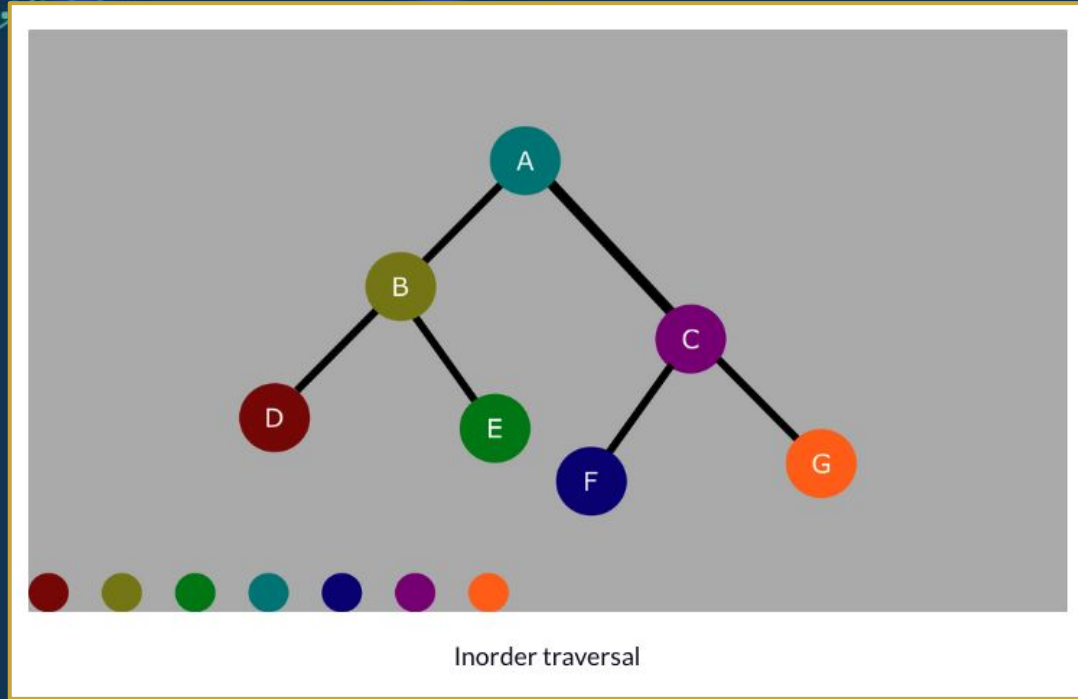
- **Left subtree**
- **Root**
- **Right subtree**

❖ Useful for:

- Visiting nodes in order
- Copying the tree

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

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



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

Tree Traversal

❖ **Pre-order traversal** visits nodes in the following order:

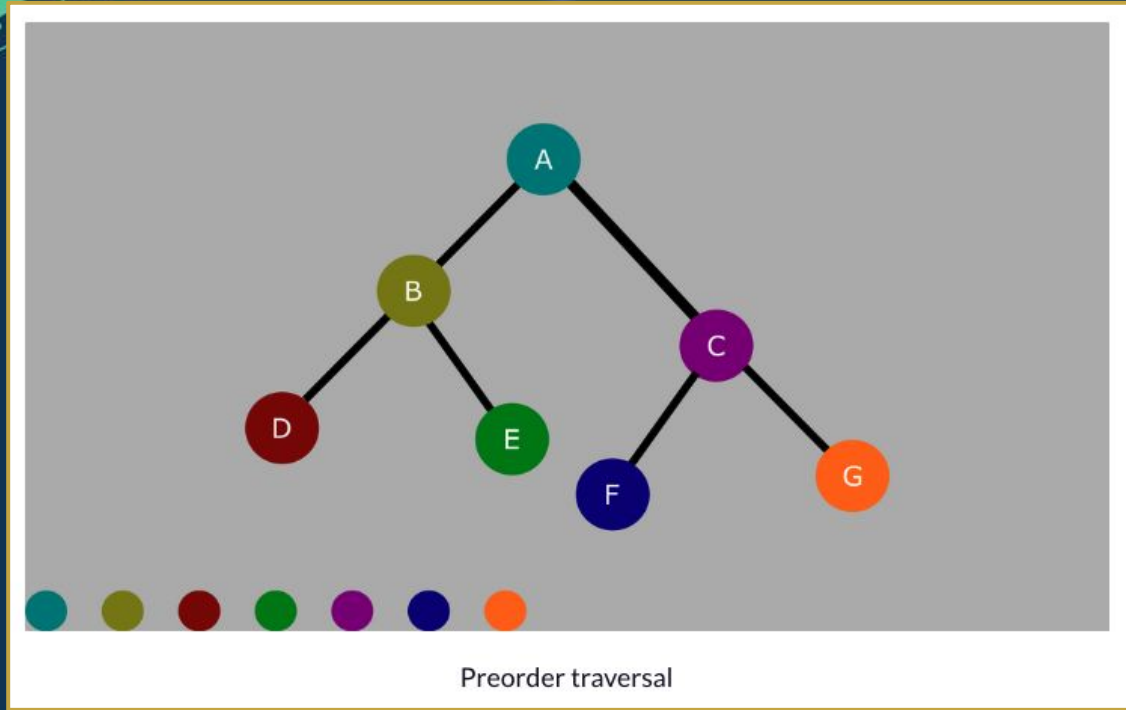
- **Root**
- **Left subtree**
- **Right subtree**

❖ Useful for:

- Creating a copy of the tree
- Prefix expression evaluation

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

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

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

Tree Traversal

❖ **Post-order traversal** visits nodes in the following order:

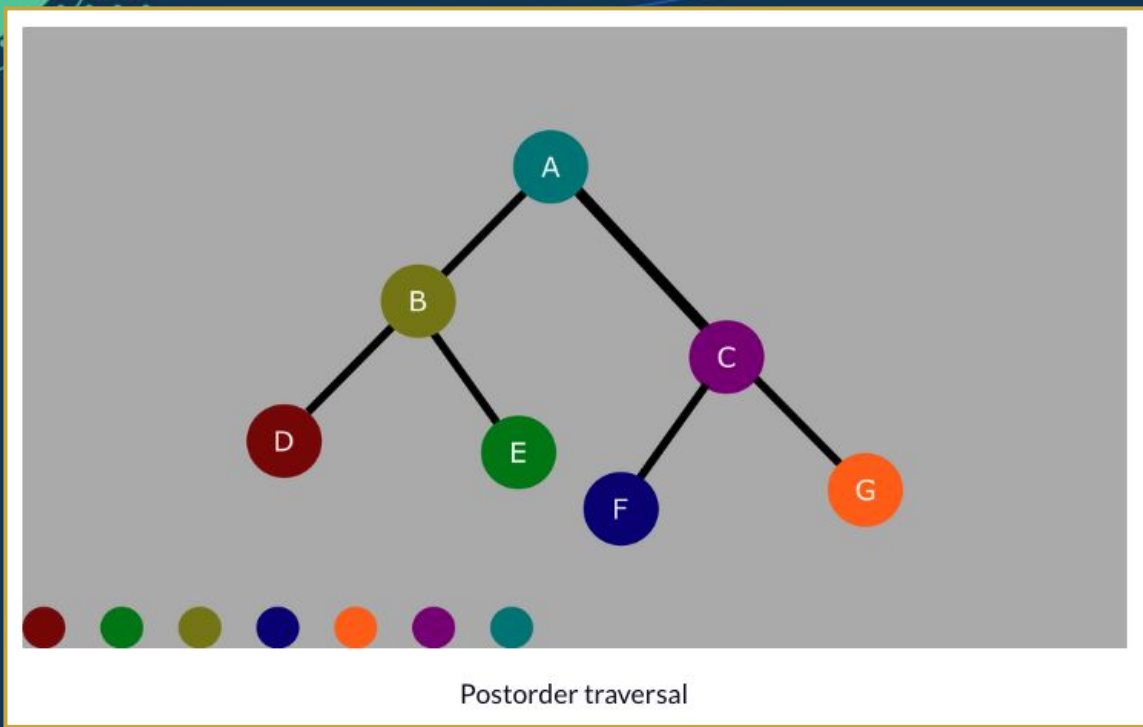
- **Left subtree**
- **Right subtree**
- **Root**

❖ Useful for:

- Deleting a tree
- Postfix expression evaluation

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

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



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

Tree Traversal

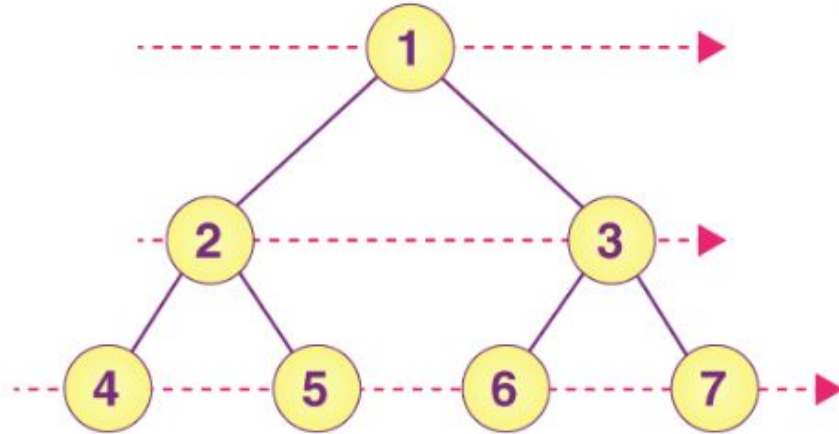
- ❖ **Level-order traversal visits nodes level by level, from left to right**
- ❖ Useful for:
 - Printing the tree level by level
 - Finding the shortest path between two nodes

Tree Traversal

```
# 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)
```

```
// 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

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

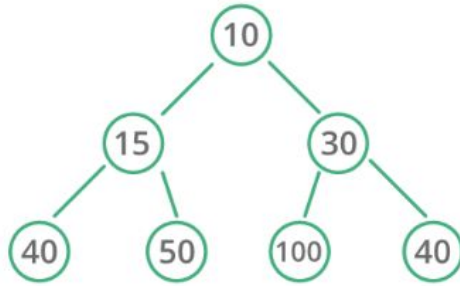
- ❖ **Heap property:**

- For a **min-heap**: $\text{parent}(i) \leq i$
- For a **max-heap**: $\text{parent}(i) \geq i$

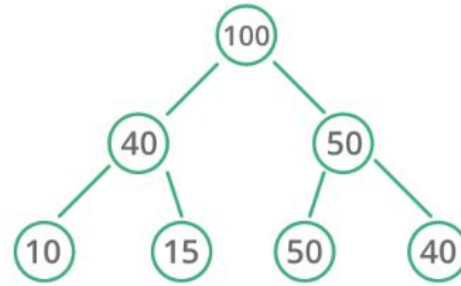
- ❖ **Applications:**

- Priority queues
- Heapsort algorithm

Heap Data Structure



Min Heap



Max Heap

GG

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



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**.



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

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

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 $n/2 * O(\log n) = O(n/2 \log n) = O(n \log n)$

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: $O(\log n)$**

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

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



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: **$O(\log n)$**

Heap Operations

```
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
```


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



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: $\text{parent}(i) \geq i$
- D. The heap property is: $\text{parent}(i) > i$





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



Homework

Practise the skills we've developed by completing the following problems:

- ❖ The next slide contains two questions to test your theoretical understanding of trees in a real world, cybersecurity scenario.
- ❖ We'll be going through two LeetCode examples in the lecture over the weekend, attempt them yourself and come ready with questions:
 - [Example 1](#)
 - [Example 2](#)
- ❖ Practice speaking through your solutions and explaining how you approached each problem.

Homework Example

You are building a program that uses tree structures to handle security incidents efficiently.

Incidents are stored based on their severity level.

1. How would you insert a new incident with a given severity into the tree using min heap?
2. After several incidents are added, how would you retrieve the most severe incident to be handled, and what is the space complexity of this operation?

Homework Example

You are building a program that uses tree structures to handle security incidents efficiently.

Incidents are stored based on their severity level.

1. How would you insert a new incident with a given severity into the tree using min heap?

You would use the insert function of the heap to add the new incident. Heapify would then ensure that the node is placed in the correct position to maintain the min heap property.

2. After several incidents are added, how would you retrieve the most severe incident to be handled, and what is the space complexity of this operation?

You would use the pop function to get the next task, which is the root of the min heap. The space complexity for this operation in an iterative approach is $O(1)$ as it does not require additional space beyond the heap itself.

Summary

- ★ Trees are fundamental in representing and managing hierarchical data structures in cyber security.
- ★ Various tree types (BSTs, AVL trees, heaps) optimize operations like searching, sorting, and prioritizing.
- ★ Tree operations such as insertion, deletion, and traversal are essential for efficient log analysis and threat prioritization.
- ★ Heaps improve efficiency in handling priority queues for critical security applications.

Further Learning

- ❖ [Programiz](#) - Binary Tree Tutorial
- ❖ [VISUALGO](#) - Tree Visualisation Tool
- ❖ [Simplilearn](#) - Data Structure and Algorithm Complexity Guide
- ❖ [Binari](#) - Interactive Binary Tree Learning

CoGrammar

Q & A SECTION

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

Thank you for attending



CoGrammar



Department
for Education