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.

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

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

Portfolio Assignment Reviews

Submit you solutions here!

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

Trees and Heaps

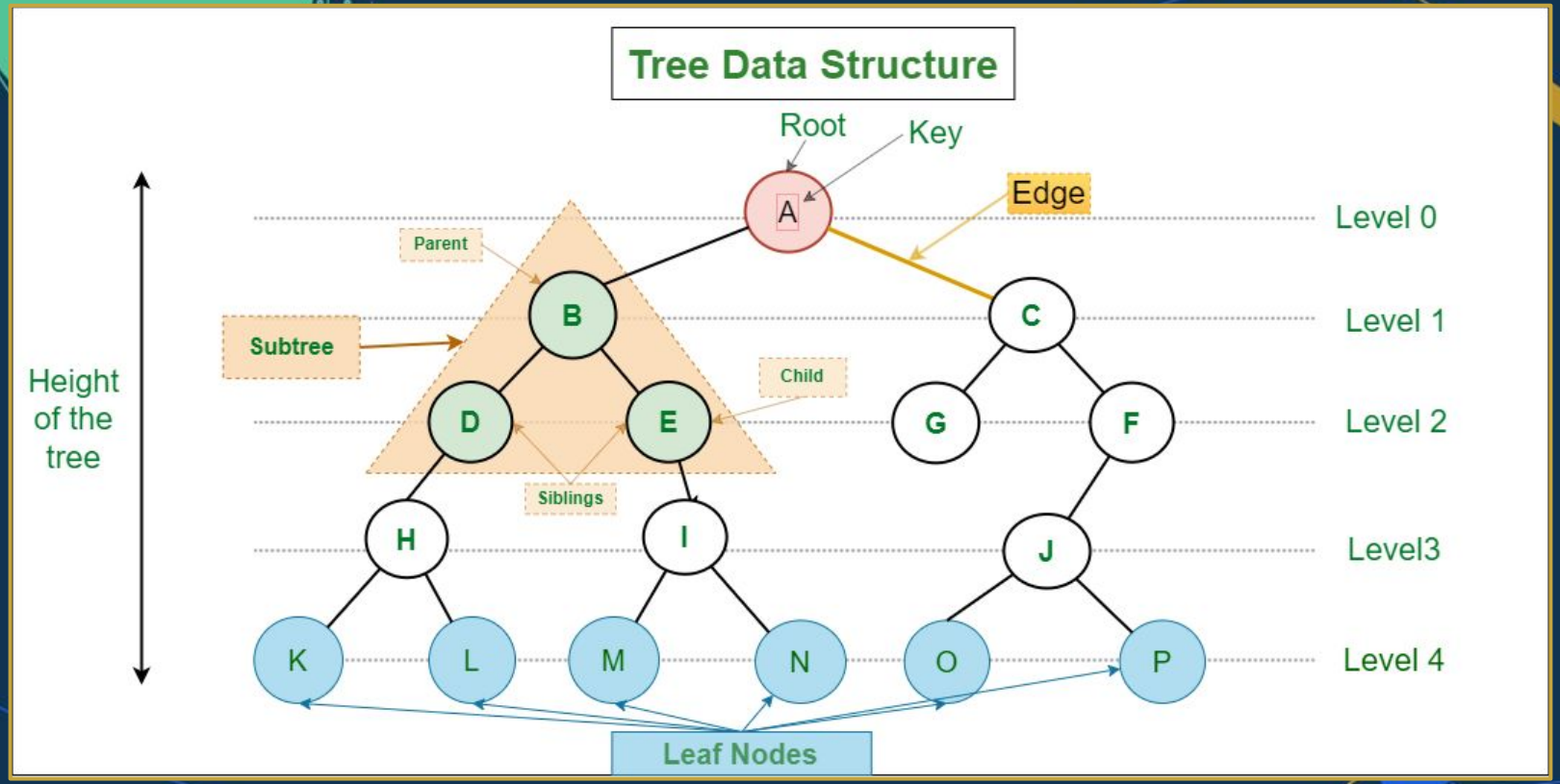
1. Tree Fundamentals
2. Binary Trees
3. Binary Search Trees (BSTs)
4. AVL Trees
5. Heap Trees
6. Tree Traversal
7. Heaps and the Heapsort algorithm

CoGrammar

Tree Fundamentals

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

- Components of a node:
  - 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/

Tree Fundamentals

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

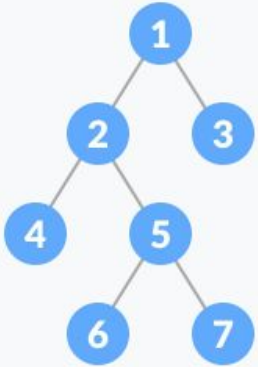
Binary Trees

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

- Properties:
  - 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

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.



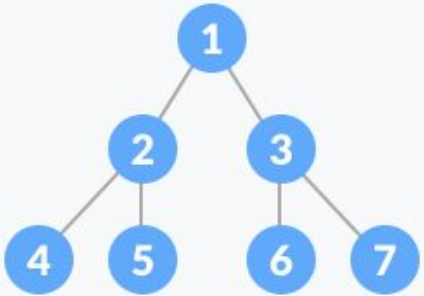
Full Binary Tree

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.



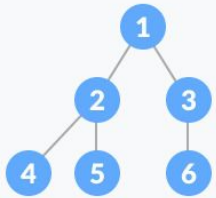
Perfect Binary Tree

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

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.



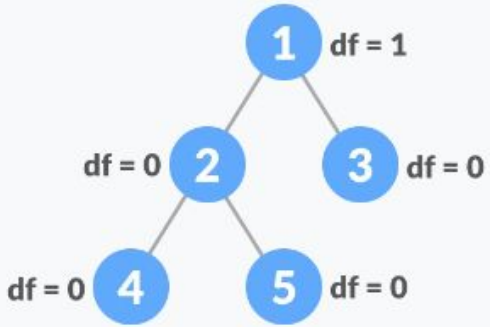
Complete Binary Tree

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

CoGrammar

6. Balanced Binary Tree

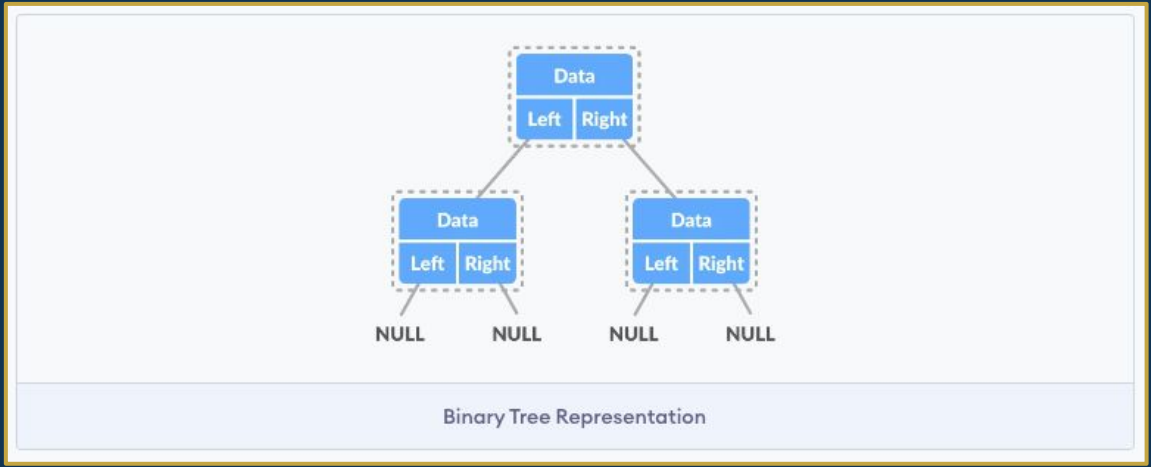
It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 or 1.



Balanced Binary Tree

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

CoGrammar



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

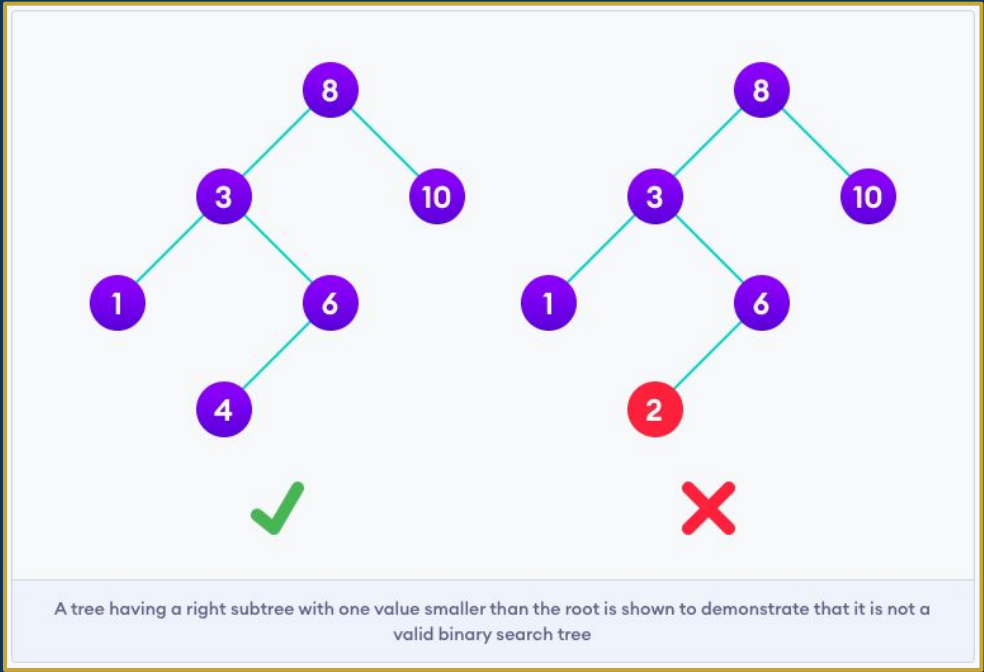
CoGrammar

```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;
  }
}
```

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

```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

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

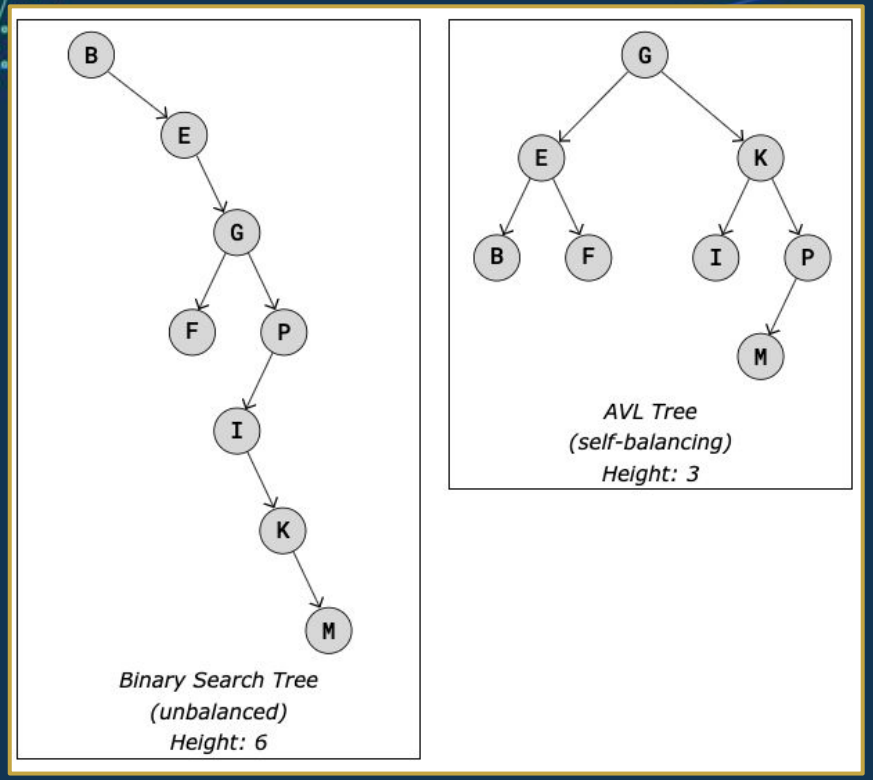
AVL Trees

- 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

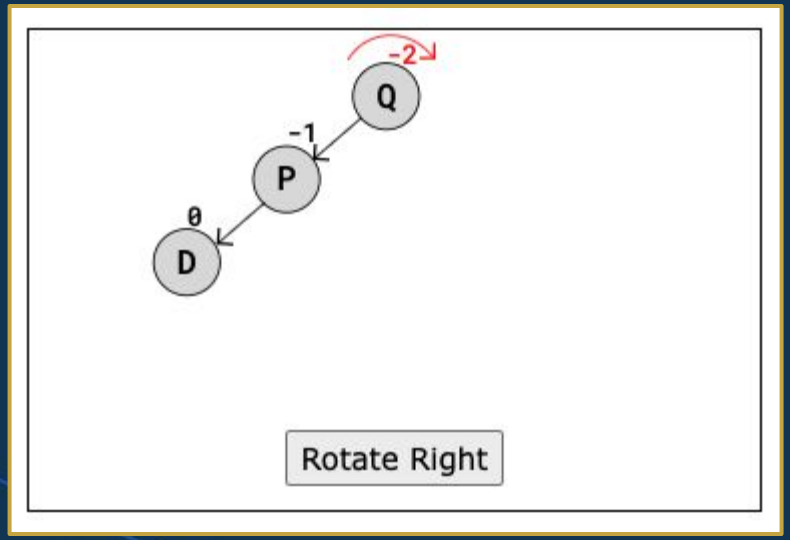
AVL Trees

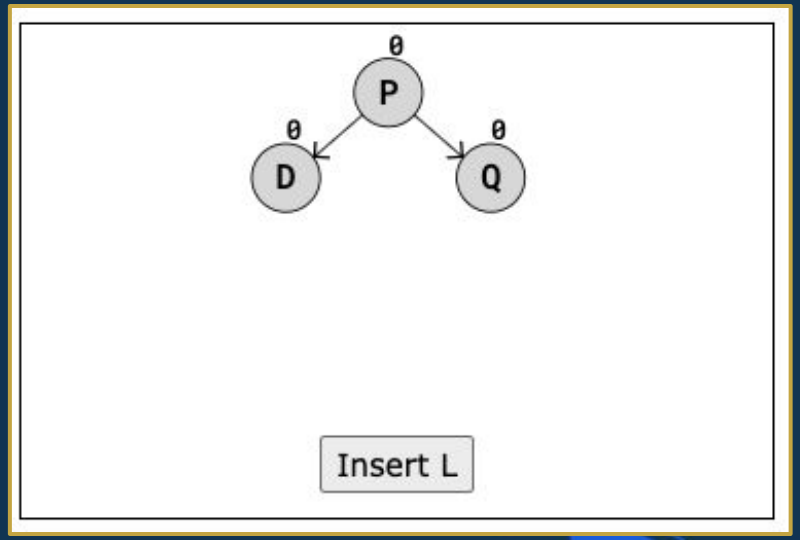
- The code for AVL trees could become quite long due to the balancing property, so feel free to check it out
  - Python: https://github.com/bfaure/Python3_Data_Structures/blob/master/AVL_Tree/main.py
  - Javascript: https://github.com/qwtw/js-avl-tree/tree/master/src

CoGrammar

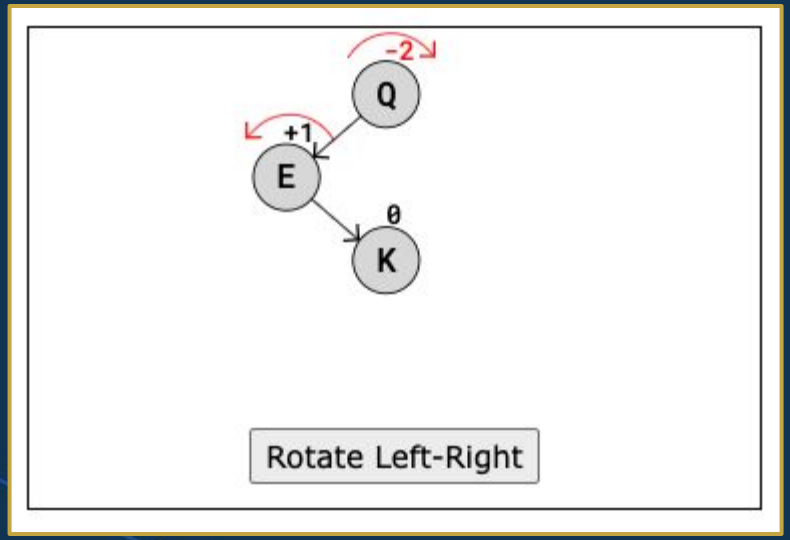


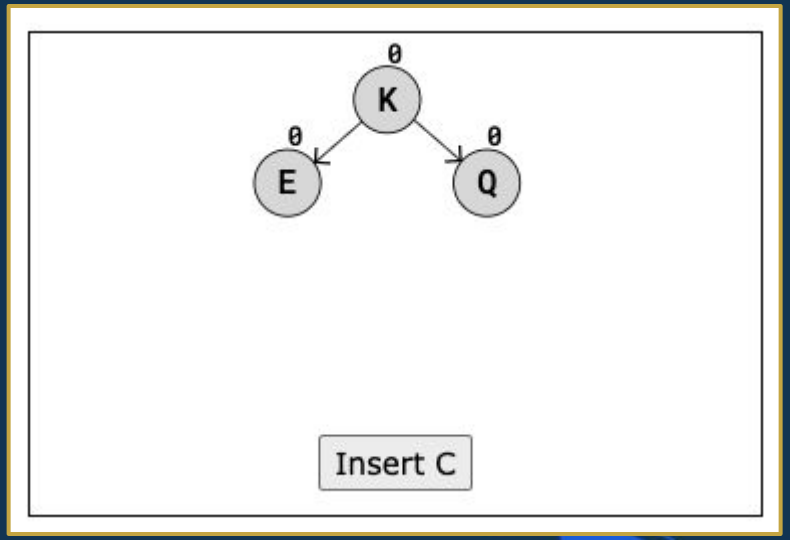
Source: https://www.w3schools.com/dsa/dsa_data_avltrees.php



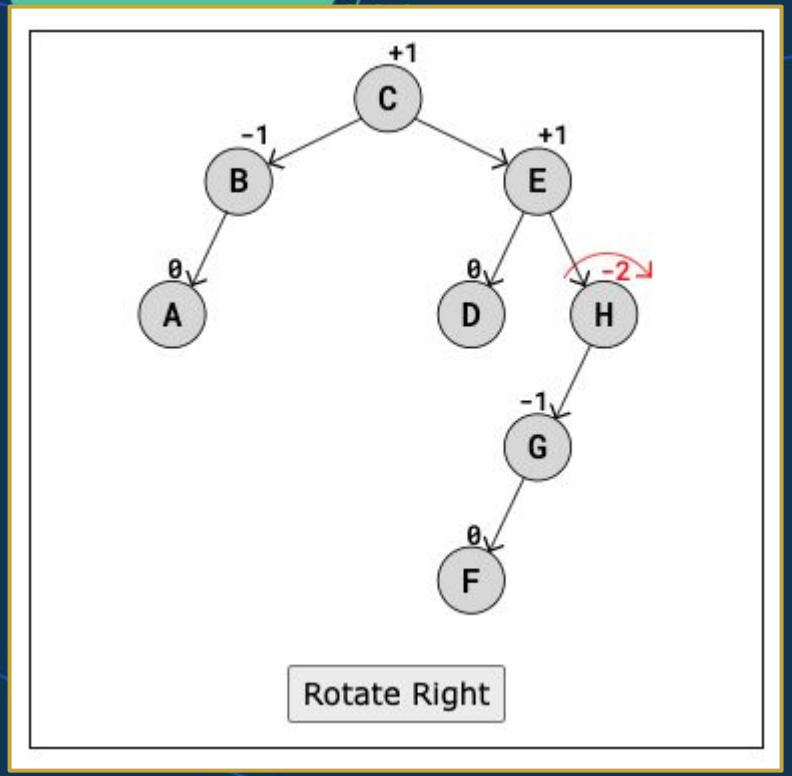


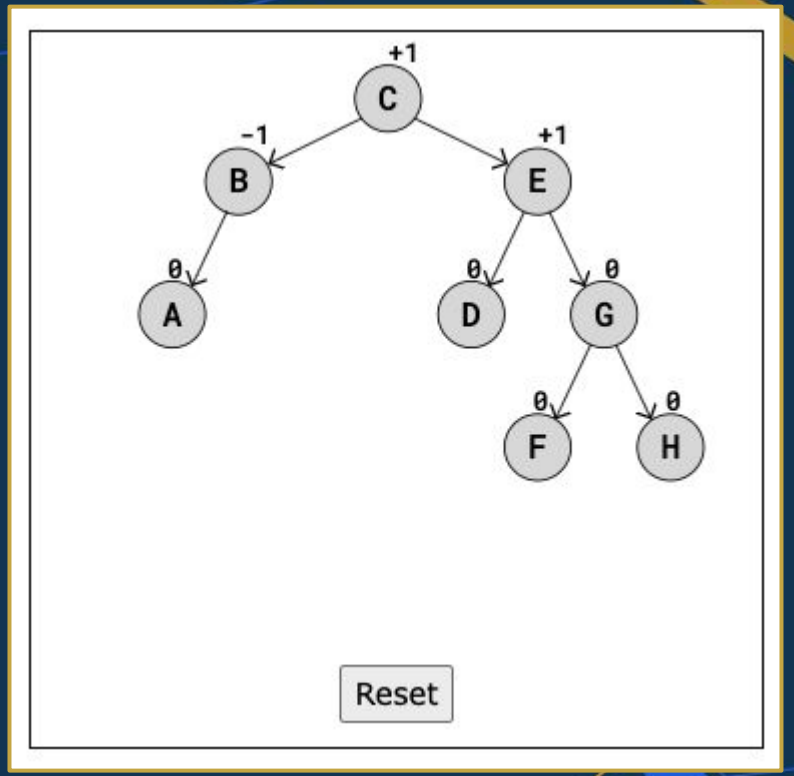
Source: https://www.w3schools.com/dsa/dsa_data_avltrees.php





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





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

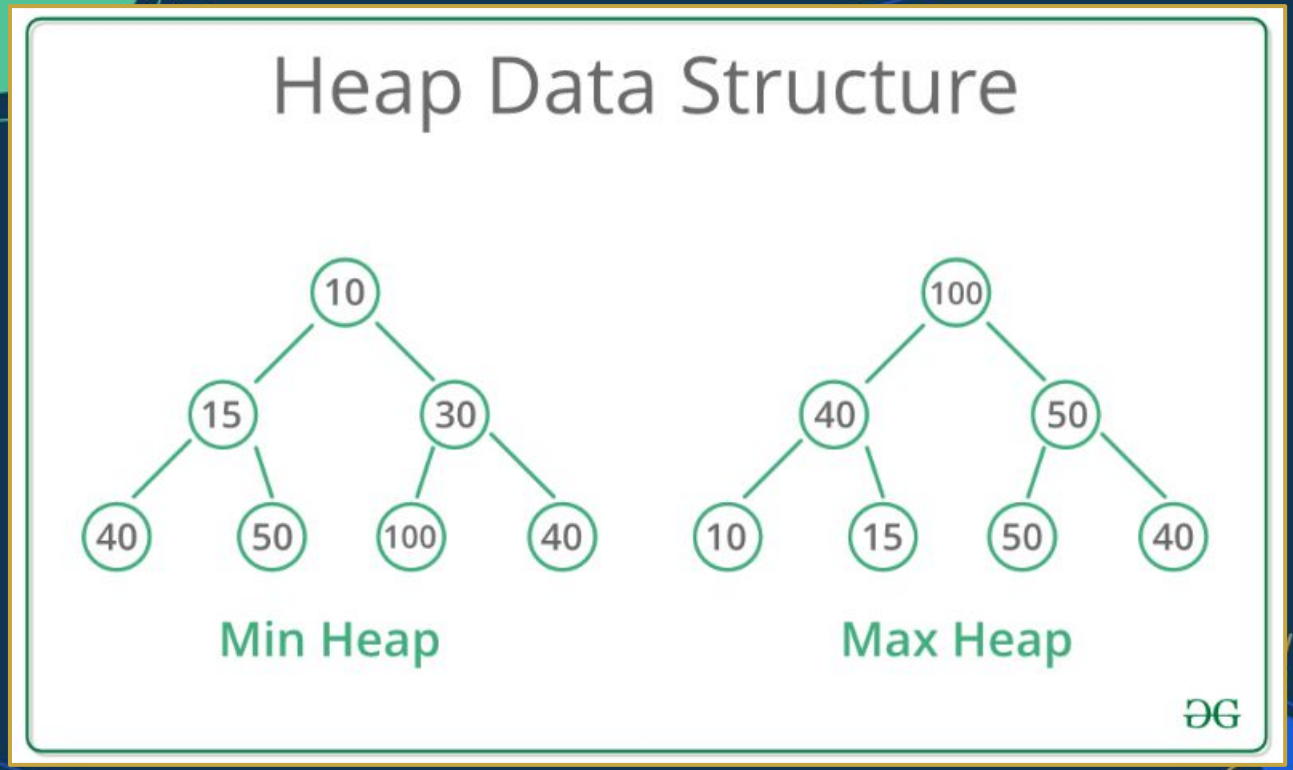
Heap

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
- Types of heaps:
  - Min-heap: The value of each node is greater than or equal to the value of its parent
  - Max-heap: The value of each node is less than or equal to the value of its parent

Heap

- Applications:
  - Priority queues
  - Heapsort algorithm



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

CoGrammar

```python
# Min Heap
class MinHeap:
    def __init__(self):
        self.heap = []

    def parent(self, i):
        return (i - 1) // 2

    def left_child(self, i):
        return 2 * i + 1

    def right_child(self, i):
        return 2 * i + 2

    def swap(self, i, j):
        self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

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

    def _sift_up(self, i):
        while i > 0 and self.heap[self.parent(i)] > self.heap[i]:
            self.swap(i, self.parent(i))
            i = self.parent(i)
```

```python
def pop(self):
    if not self.heap:
        return None
    if len(self.heap) == 1:
        return self.heap.pop()
    min_val = self.heap[0]
    self.heap[0] = self.heap.pop()
    self._sift_down(0)
    return min_val

def _sift_down(self, i):
    while True:
        l = self.left_child(i)
        r = self.right_child(i)
        smallest = i
        if l < len(self.heap) and self.heap[l] < self.heap[smallest]:
            smallest = l
        if r < len(self.heap) and self.heap[r] < self.heap[smallest]:
            smallest = r
        if smallest == i:
            break
        self.swap(i, smallest)
        i = smallest
```

CoGrammar

```
// Min Heap
class MinHeap {
  constructor() {
    this.heap = [];
  }

  parent(i) {
    return Math.floor((i - 1) / 2);
  }

  leftChild(i) {
    return 2 * i + 1;
  }

  rightChild(i) {
    return 2 * i + 2;
  }

  swap(i, j) {
    [this.heap[i], this.heap[j]] = [this.heap[j], this.heap[i]];
  }

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

```
_siftUp(i) {
  while (i > 0 && this.heap[this.parent(i)] > this.heap[i]) {
    this.swap(i, this.parent(i));
    i = this.parent(i);
  }
}

pop() {
  if (this.heap.length === 0) {
    return null;
  }
  if (this.heap.length === 1) {
    return this.heap.pop();
  }
  const minVal = this.heap[0];
  this.heap[0] = this.heap.pop();
  this._siftDown(0);
  return minVal;
}
```

```
_siftDown(i) {
  while (true) {
    const l = this.leftChild(i);
    const r = this.rightChild(i);
    let smallest = i;
    if (l < this.heap.length && this.heap[l] < this.heap[smallest]) {
      smallest = l;
    }
    if (r < this.heap.length && this.heap[r] < this.heap[smallest]) {
      smallest = r;
    }
    if (smallest === i) {
      break;
    }
    this.swap(i, smallest);
    i = smallest;
  }
}
```

CoGrammar

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

A. 1

B. 2

C. 3

D. 4

Answer: B

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

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. $O(n^2)$

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.

Let’s Breathe!

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

CoGrammar

Tree Traversal

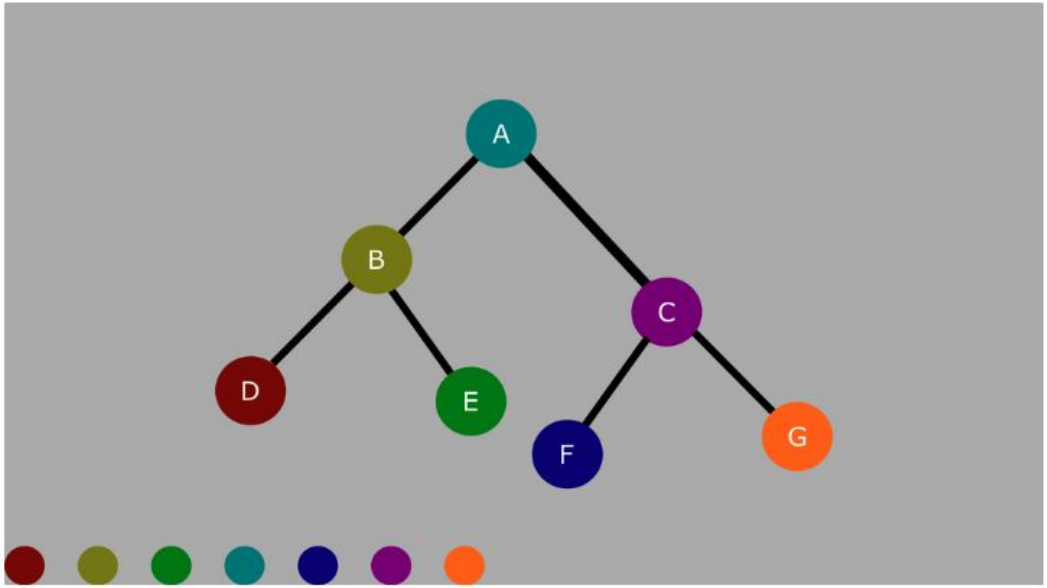
- Tree traversal is the process of visiting each node in a tree exactly once
- Types of tree traversal:
  - In-order traversal
  - Pre-order traversal
  - Post-order traversal
  - Level-order traversal

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

```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);
  }
}
```



Inorder traversal

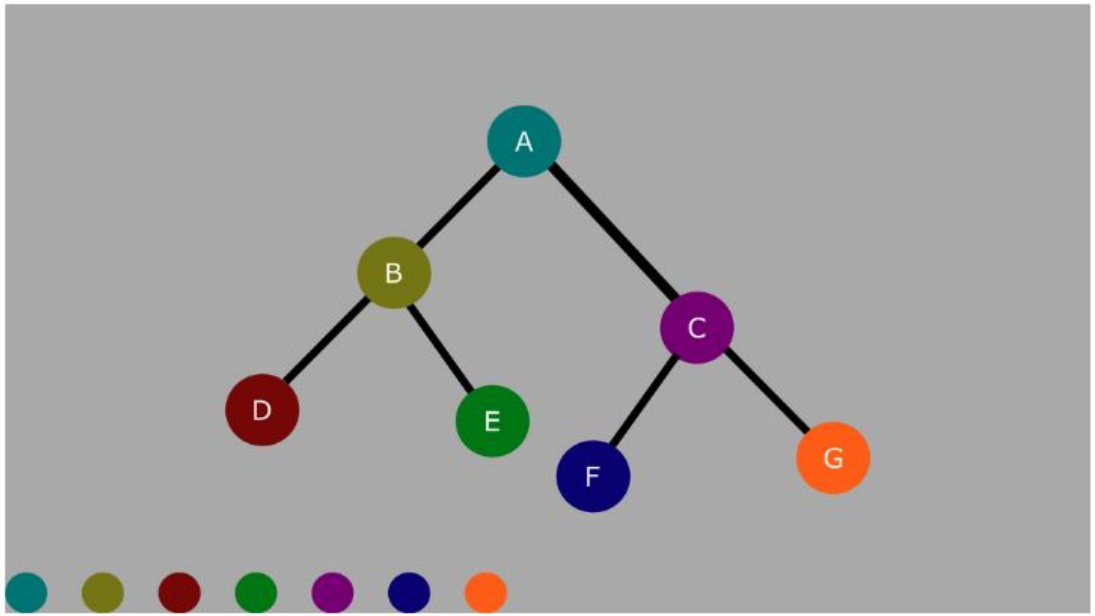
Source: https://www.freecodecamp.org/news/binary-search-tree-traversal-inorder-preorder-post-order-for-bst/

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

```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);
  }
}
```



Preorder traversal

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

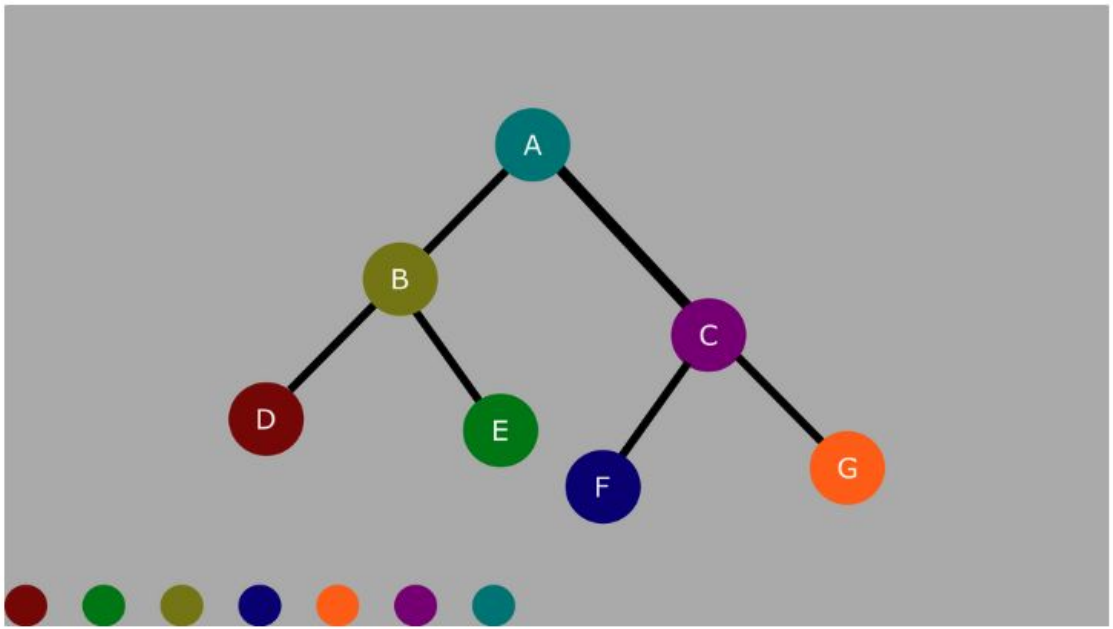
CoGrammar

Tree Traversal

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

```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);
  }
}
```



Postorder traversal

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

CoGrammar

Tree Traversal

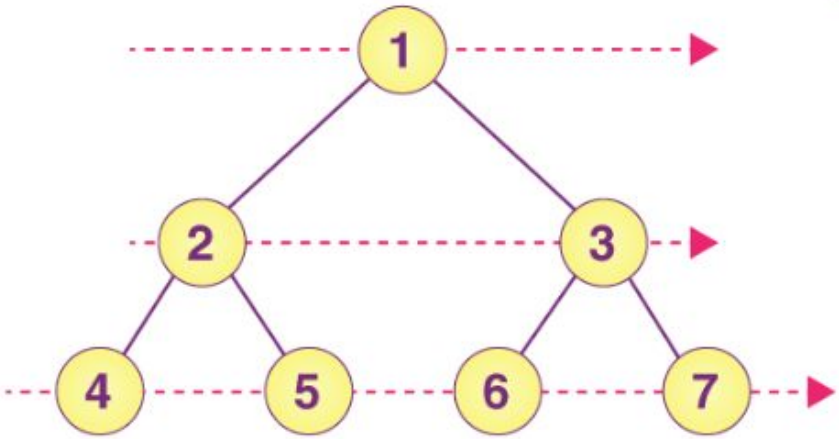
- Level-order traversal visits nodes level by level, from left to right
- Useful for:
  - Printing the tree level by level
  - Finding the shortest path between two nodes

```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);
    }
  }
}
```

CoGrammar



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)

```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;
}
```

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);
}
```

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;
}
```

Heap Operations

- Peek (getting the minimum or maximum element):
  - Return the root of the heap
  - Time complexity: O(1)

```python
def peek(self):
    if not self.heap:
        return None
    return self.heap[0]
```

```javascript
peek() {
  if (!this.heap.length) {
    return null;
  }
  return this.heap[0];
}
```

Priority Queue using Heap

A priority queue is an abstract data type that allows inserting elements with priorities and extracting the element with the highest (or lowest) priority

- Can be efficiently implemented using a heap
  - Min-heap for a min-priority queue
  - Max-heap for a max-priority queue

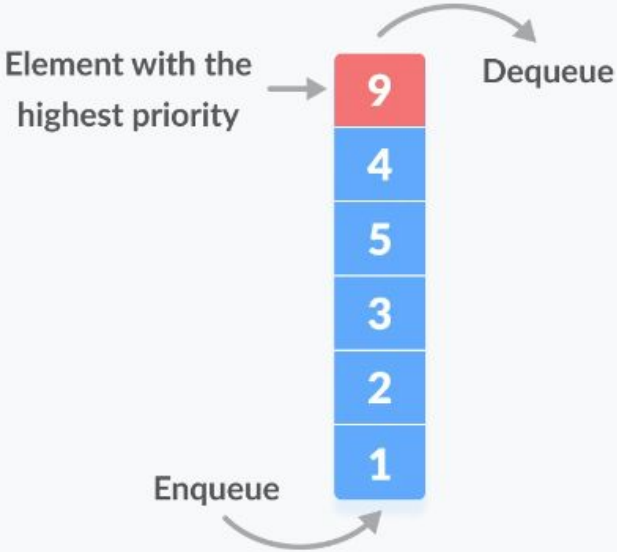
Priority Queue using Heap

- Operations:
  - Insertion: O(log n)
  - Extraction: O(log n)
  - Peek: O(1)
- Applications:
  - Dijkstra's shortest path algorithm
  - Task scheduling

A comparative analysis of different implementations of priority queue is given below.

Operations	peek	insert	delete
Linked List	`O(1)`	`O(n)`	`O(1)`
Binary Heap	`O(1)`	`O(log n)`	`O(log n)`
Binary Search Tree	`O(1)`	`O(log n)`	`O(log n)`

Source: https://www.programiz.com/dsa/priority-queue



Removing Highest Priority Element

Source: https://www.programiz.com/dsa/priority-queue

CoGrammar

```python
class PriorityQueue:
    def __init__(self):
        self.heap = MinHeap()

    def enqueue(self, val):
        self.heap.push(val)

    def dequeue(self):
        return self.heap.pop()

    def peek(self):
        return self.heap.peek()

    def is_empty(self):
        return len(self.heap.heap) == 0
```

```javascript
class PriorityQueue {
  constructor() {
    this.heap = new MinHeap();
  }

  enqueue(val) {
    this.heap.push(val);
  }

  dequeue() {
    return this.heap.pop();
  }

  peek() {
    return this.heap.peek();
  }

  isEmpty() {
    return this.heap.heap.length === 0;
  }
}
```

CoGrammar

Heapsort Algorithm

Heapsort is a comparison-based sorting algorithm that uses a heap data structure

- Steps:
  - Build a max-heap from the input array
  - Swap the root (maximum element) with the last element
  - Remove the last element (now the maximum) from the heap
  - Repeat steps 2-3 until the heap is empty

Heapsort Algorithm

- Time complexity: O(n log n)
- Space complexity: O(1)

```python
# Heapsort
def heapsort(arr):
    def sift_down(start, end):
        root = start
        while root * 2 + 1 <= end:
            child = root * 2 + 1
            if child + 1 <= end and arr[child] < arr[child + 1]:
                child += 1
            if arr[root] < arr[child]:
                arr[root], arr[child] = arr[child], arr[root]
                root = child
            else:
                break

    # Build max-heap
    for start in range(len(arr) // 2 - 1, -1, -1):
        sift_down(start, len(arr) - 1)

    # Heap sort
    for end in range(len(arr) - 1, 0, -1):
        arr[0], arr[end] = arr[end], arr[0]
        sift_down(0, end - 1)
    return arr
```

```javascript
// Heapsort
function swap(arr, i, j) {
  [arr[i], arr[j]] = [arr[j], arr[i]];
}

function heapifySort(arr, n, i) {
  let largest = i;
  let left = 2 * i + 1;
  let right = 2 * i + 2;

  if (left < n && arr[left] > arr[largest]) {
    largest = left;
  }

  if (right < n && arr[right] > arr[largest]) {
    largest = right;
  }

  if (largest !== i) {
    swap(arr, i, largest);
    heapifySort(arr, n, largest);
  }
}
```

```javascript
function heapsort(arr) {
  const n = arr.length;

  // Build max heap
  for (let i = Math.floor(n / 2) - 1; i >= 0; i--) {
    heapifySort(arr, n, i);
  }

  // Extract elements from the heap one by one
  for (let i = n - 1; i > 0; i--) {
    swap(arr, 0, i);
    heapifySort(arr, i, 0);
  }

  return arr;
}
```

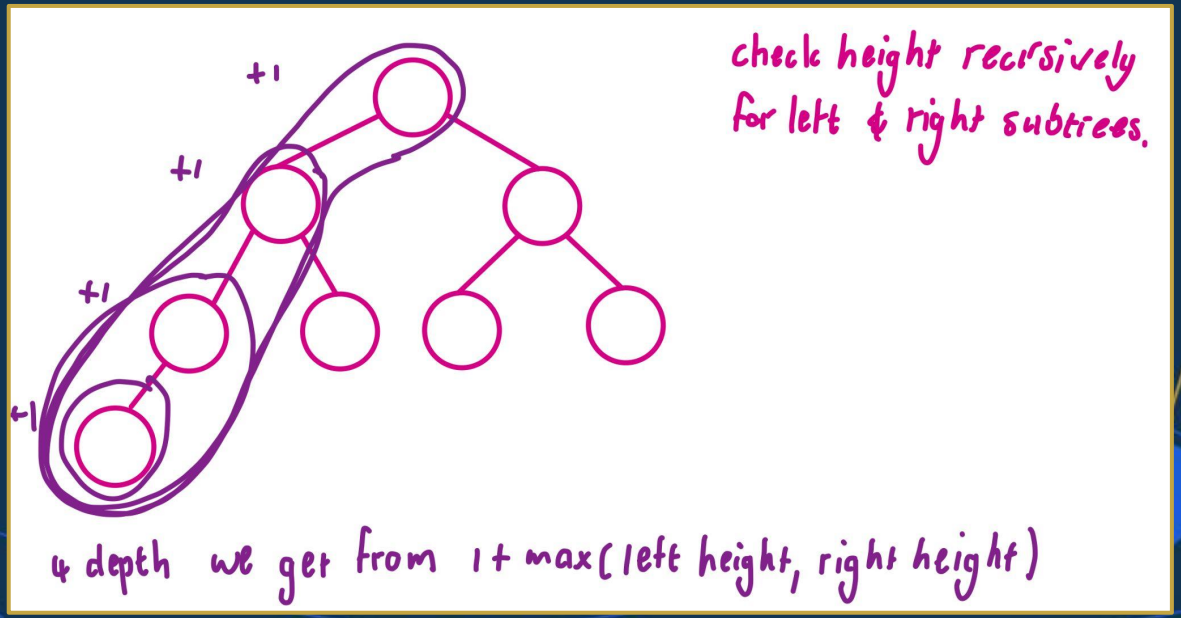
CoGrammar

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

Interview-Style Question 1

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



```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));
}
```

CoGrammar

Interview-Style Question 2

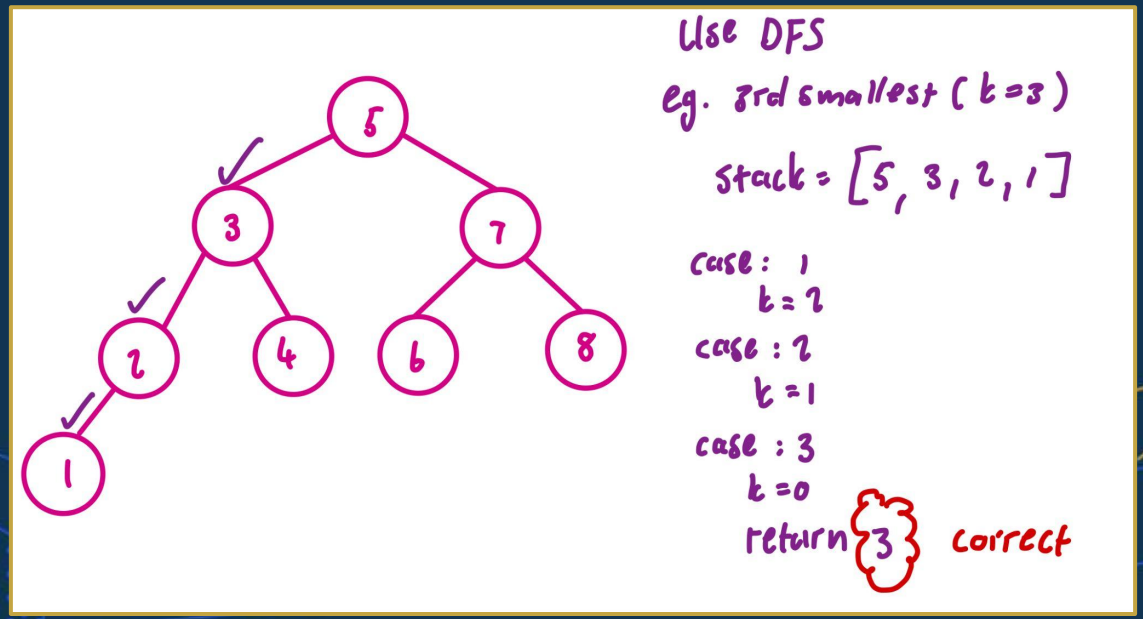
- Given the root of a binary tree, determine if it is a valid binary search tree (BST)
  - Basically, are all left nodes less in value than the right nodes
  - We can just check this for the left and right subtrees recursively

```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)
  );
}
```

Interview-Style Question 3

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



```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.

What is the space complexity of the heapsort algorithm?

A. O(1)

B. O(log n)

C. O(n)

D. O(n log n)

Answer: A

- Heapsort is an in-place sorting algorithm, meaning it does not require any additional space beyond the input array. Therefore, its space complexity is O(1).

Which of the following is NOT a valid use case for a priority queue?

A. Task scheduling

B. Dijkstra's shortest path algorithm

C. Sorting a list of numbers

D. Huffman coding

Answer: C

- While priority queues are useful for task scheduling, graph algorithms like Dijkstra's shortest path, and Huffman coding, they are not the most efficient choice for sorting a list of numbers. Algorithms like heapsort, quicksort, or mergesort are better suited for general-purpose sorting.

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