CoGrammar

Welcome to this session:

Coding Interview Workshop - Non-Linear Data Structures

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

CoGrammar HyperionDev

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

- **Understand the key differences** between trees , heaps, and graphs, and their use cases.
- **Implement binary search trees (BST), heaps, and graph adjacency lists** for efficient data handling.
- **Perform DFS and BFS traversals** and apply them to problems like shortest path and cycle detection.
- **Understand the various problems** which benefit from using a non-linear data structure in the solution.
- **Use heaps for priority queues** and understand their significance in scheduling and shortest path algorithms.

Which data structure would be best for efficiently storing and searching hierarchical data like a file system?

A. Stack
B. Tree
C. Queue
D. Hash Table

In a social network, which data structure would best represent friend connections?

A. Stack
B. Queue
C. Tree
D. Graph

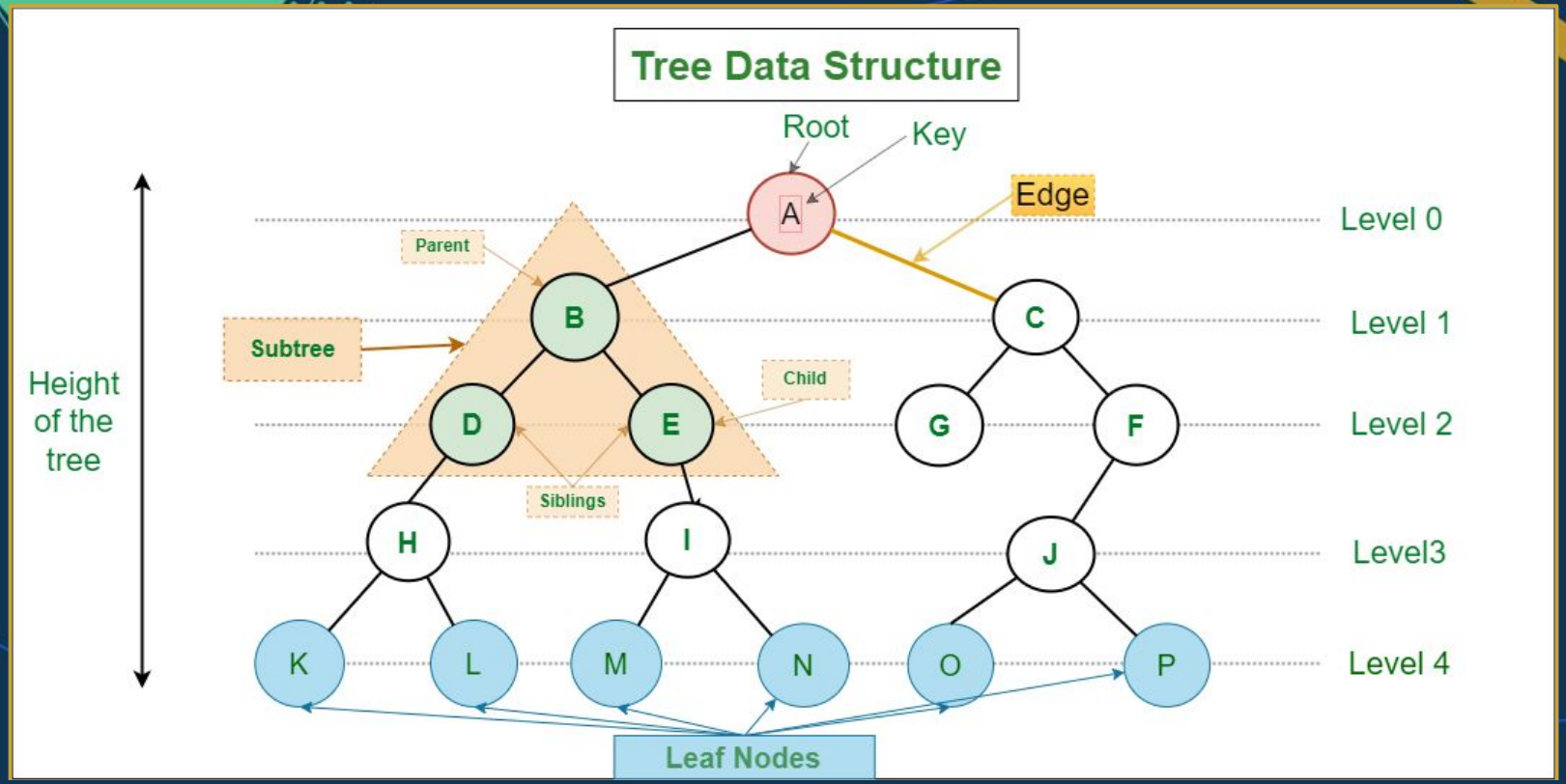
Lecture Overview

- Trees
- Heaps
- Graphs

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/

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.**

AVL Trees

An AVL tree is a self-balancing binary search tree

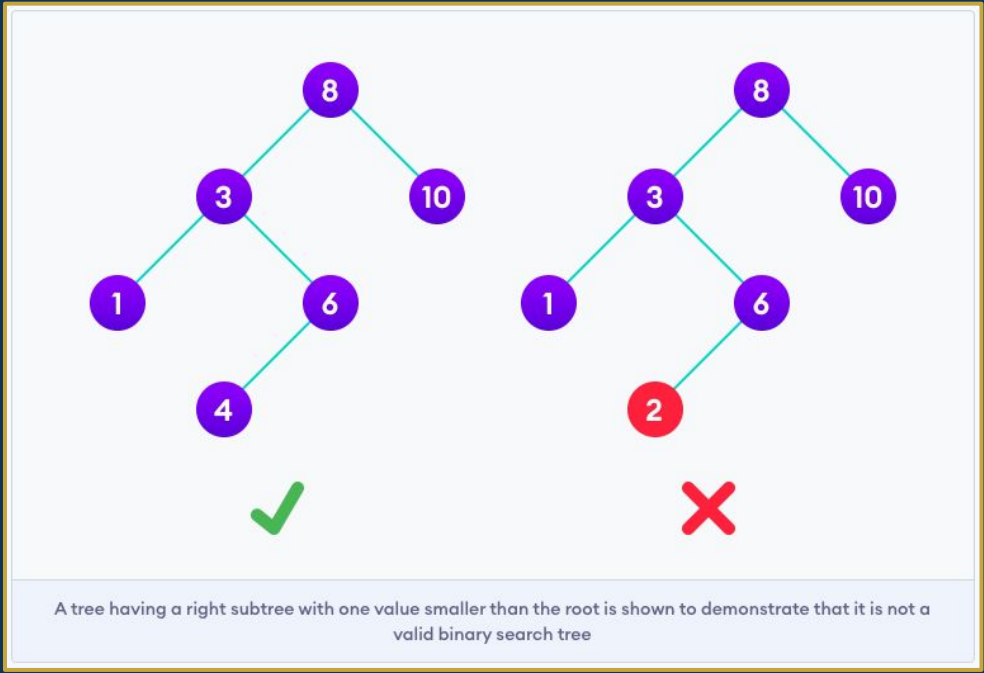
- Balancing property:
  - The **heights** of the left and right subtrees of any node **differ by at most one.**
  - This property ensures that the tree **remains balanced**, preventing degeneration into a linked list.

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).

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.



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

CoGrammar

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

```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);
```

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);
```

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);
```

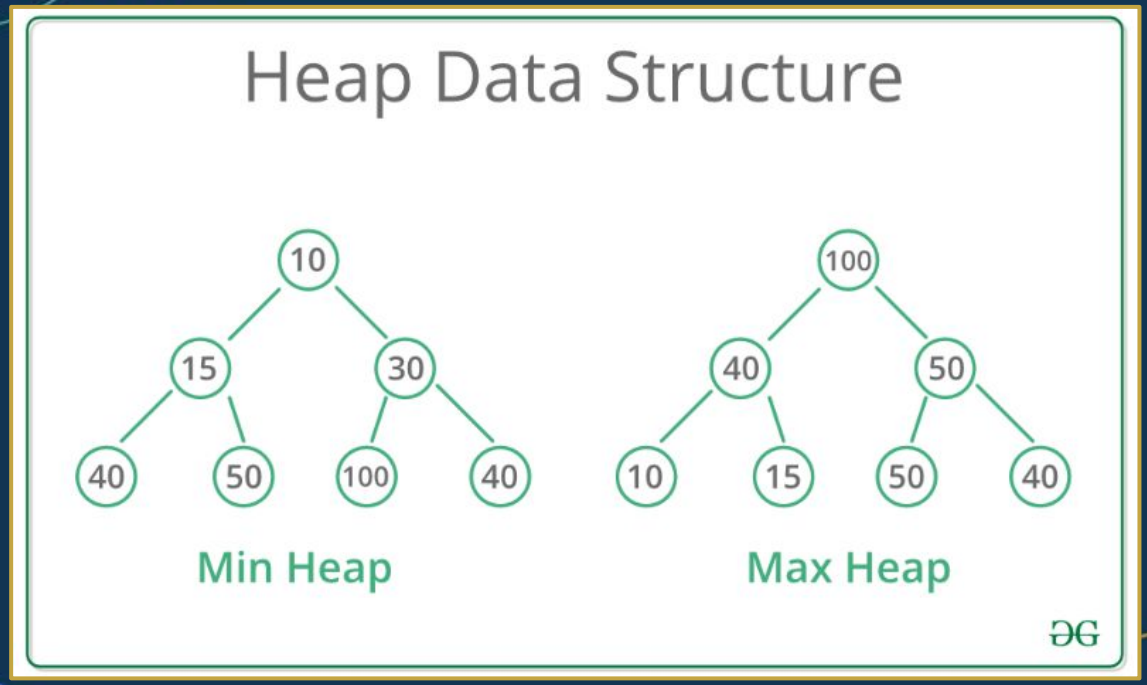
Tree Traversal

- **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

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**
- **Applications:**
  - Priority queues
  - Heapsort algorithm



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

CoGrammar

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.**

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

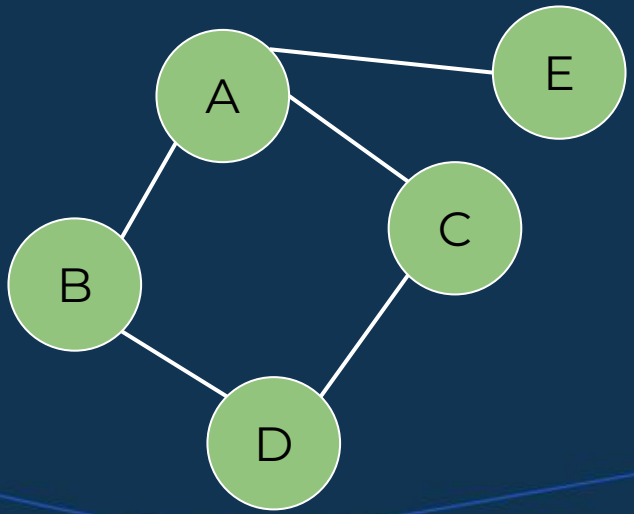
- **Deletion** (extracting the minimum or maximum element):
  - Replace the root with the last element in the heap
  - Remove the last element
  - Sift down the new root to maintain the heap property
  - Time complexity: **O(log n)**

Graphs

A non-linear data structure made up of vertices or nodes and connected by edges or arcs, that is used to represent complex relationships between objects.

- Graphs are made up of two sets, the **Vertices (*V*)** and **Edges (*E*)**.
- Each element of *E* is a **pair** consisting of two elements from *V*.
- Vertices can be **labelled** and may be a **reference** to an external entity with additional information known as **attributes**.
- Edges can be **labelled** and the pairs can be **ordered** depending on the type of the graph.

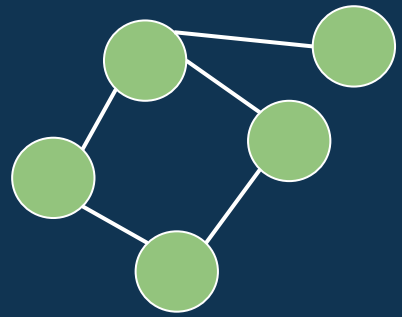
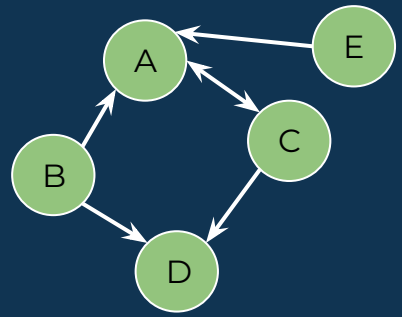
- Graphs are depicted visually using circles or boxes to represent vertices, and lines (or arrows) between the circles or boxes to represent edges. This can only be done for small datasets.



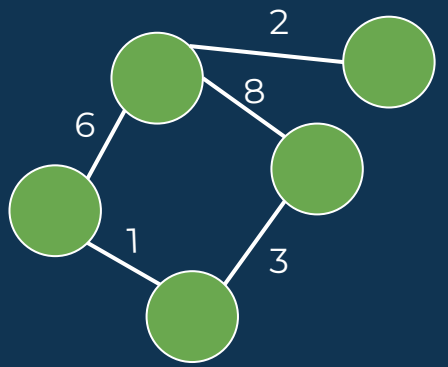
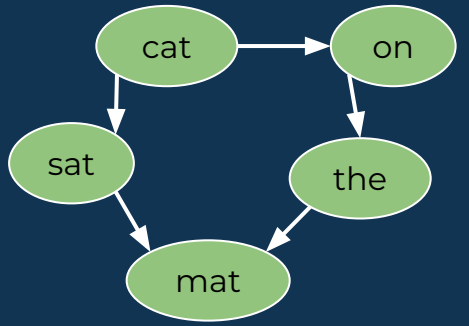
- **Neighbours:** Two nodes that are connected by an edge. This property is also known as **adjacency**.
  - *E.g. A is adjacent to B, A and B are neighbours*
- **Degree:** The number of other nodes that a node is connected to (i.e. the number of neighbours a node has).
  - *E.g. The degree of A is 3*
- **Loop:** An edge that connects a node to itself.

- **Path:** A sequence of nodes that are connected by edges.
  - ➢ *E.g. (A, B, D) denoted using **sequence notation ()***
- **Cycle:** A closed path which starts and ends at the same node and no node is visited more than once.
  - ➢ *E.g. {A, C, D, B, A}*

Types of Graphs

Undirected Graphs Edges connecting nodes have no direction. For this graph, the order of the pairs of vertices in the edge set does not matter. **Applications:** Social networks, Recommendation Systems	
Directed Graphs Edges connecting nodes have specified directions. Edges may also be bidirectional. This graph cannot contain any loops. **Applications:** Maps, Network Routing, WWW	

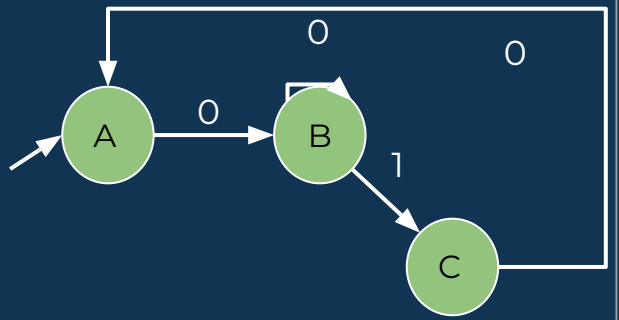
CoGrammar

Weighted/Labelled Graphs Directed/Undirected graphs that have values associated with each of its edges. These values can record any information relating to the edge e.g. distance between nodes. **Applications:** Transportation Networks, Financial/Transactional Networks	
Vertex Labelled Graphs Directed/Undirected graphs where the vertices/nodes in the graph are labelled with information which identifies the vertex. **Applications:** Biological Networks (molecular structures), Semantic Networks	

Cyclic Graphs

A directed graph which contains at least one cycle.

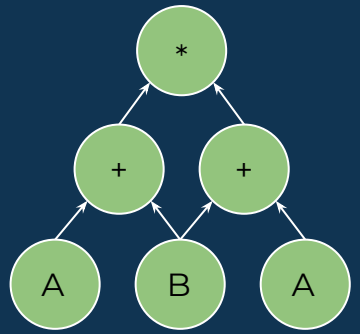
Applications: Task Scheduling, Manufacturing Processes, Finite State Machines (a mathematical model of computation)



Directed Acyclic Graphs

Also known as a DAG. A directed graph with no cycles. Various use-cases across fields.

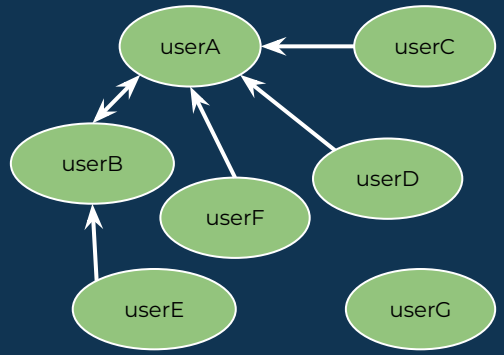
Applications: Dependency Resolution Systems e.g. package management, Project Management, Compiler Design



Disconnected Graphs

The graph contains nodes which are not connected via an edge to any other nodes in the graph.

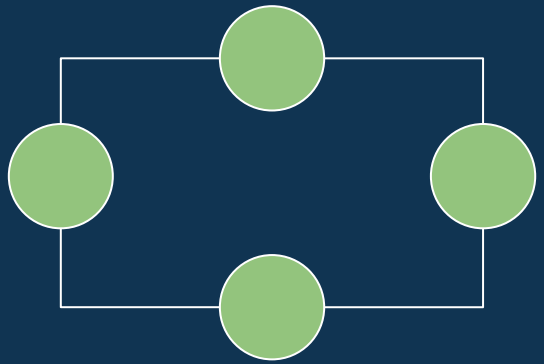
Applications: Social Networks, Transportation Networks, Component Analysis



Connected Graphs

There is a path from any node in the graph to any other node in the graph.

Applications: Communication Networks, Routing Algorithms, Circuit Design, Data Analysis



CoGrammar

Graph Traversal

- The process of **visiting each node in a graph exactly once**
- Types of graph traversal algorithms:
  - **Depth-First Search (DFS):** How to explore as deep as possible for each branch in a graph, before backtracking to explore an alternative branch.
  - **Breadth-First Search (BFS):** Level-by-level exploration of nodes.

Practice the Structures

Let's practice using different Non-Linear Data Structures by solving some classic problems for each paradigm.

Then we'll do the following problems together:

- Binary Tree Inorder Traversal
- Symmetric Tree
- Course Schedule

Which data structure is best for finding the shortest path in a city map?

A. Tree
B. Heap
C. Graph
D. Stack

If you need to prioritize tasks in a job scheduler, which data structure is best?

A. Stack
B. Queue
C. Heap
D. Linked List

Homework

Practise the skills we've developed by completing the rest of the LeetCode questions:

- Practise speaking through your solutions and explaining how you approached each problem.
- In the next lecture we'll be covering the topic: "String Manipulation"
- You can have a look at the following LeetCode questions to prepare:
  - Example 1
  - Example 2
  - Example 3

Summary

- ★ **Trees**: Used for hierarchical data like file systems.
- ★ **Heaps**: Used for priority scheduling.
- ★ **Graphs**: Used for complex relationships like social networks.
- ★ **BFS & DFS**: Used for pathfinding and cycle detection.
- ★ **Dijkstra's Algorithm**: Uses a heap for finding shortest paths efficiently.

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

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education