Month 6, Week 2: The Unabridged Learning Guide

Introduction: The Architect's Advanced Toolkit

Last week, we laid the scientific foundation for our work as engineers. We learned the language of performance, Big O Notation, and used it to analyze the fundamental linear data structures that underpin so much of our code. We have now moved beyond simply making code *work*; we are learning to make it *work well*.

This week, we expand our toolkit into the world of **non-linear data structures** and master the **fundamental algorithms** for searching and sorting that are the bedrock of computer science. These are not abstract academic topics; they are the solutions to some of the most common and critical performance bottlenecks in backend development.

First, we will take a deep dive into **Hash Tables**, the data structure that powers JavaScript's Objects and Maps and provides the near-instant lookups that modern applications demand. We will then explore hierarchical data with **Trees**, focusing specifically on the **Binary Search Tree (BST)**, the structure that makes database indexes incredibly fast. We will also introduce the conceptual framework for **Graphs**, the ultimate data structure for modeling complex networks and relationships.

In the second half of our lesson, we will shift from organizing data to processing it. We will contrast the slow but simple **Linear Search** with the lightning-fast **Binary Search**. Finally, we will tackle one of the most classic computer science problems: **sorting**. We will implement and analyze four key sorting algorithms, from the simple but inefficient **Bubble Sort** and **Insertion Sort** to the powerful, "divide and conquer" strategies of **Merge Sort** and **Quick Sort**.

By the end of this lesson, you will have a deep, practical understanding of the trade-offs between these advanced tools, enabling you to make the intelligent, informed architectural decisions that define a senior engineer.

Table of Contents

1. Module 1: Hash Tables - The Engine of O(1)
    - The Hash Function and Collisions
    - Collision Resolution: Separate Chaining
    - JavaScript's Implementations: `Object` vs. `Map`
2. Module 2: Trees - The Hierarchical Structure
    - Tree Terminology
    - Binary Search Trees (BSTs)
    - Implementation and Performance
3. Module 3: Graphs - The Network Model
    - Graph Terminology
    - Representing Graphs: Adjacency List vs. Adjacency Matrix
4. Module 4: Fundamental Searching & Sorting Algorithms
    - Searching: Linear vs. Binary Search
    - Simple Sorting: Bubble Sort & Insertion Sort
    - Efficient Sorting: Merge Sort & Quick Sort

5. Take-Home Assessment: The Sorting Algorithm Analyzer

Module 1: Hash Tables - The Engine of O(1)

The Hash Function and Collisions A hash table is a combination of two things: an array (the underlying storage) and a **hash function**. The hash function is a "magic" function that takes a key of any type and deterministically converts it into an integer, which is then used as an index into the array.

A good hash function should be: * **Fast:** It needs to compute the index quickly. * **Deterministic:** The same key must always produce the same index. * **Uniform:** It should distribute keys as evenly as possible across the available array slots to minimize collisions.

A **collision** is the unavoidable event where two different keys produce the same hash index.

Collision Resolution: Separate Chaining The most common way to handle collisions is to have each slot in the underlying array point to another data structure, typically a **linked list**. When a collision occurs, you simply traverse the linked list at that index. If the key already exists, you update the value. If it doesn't, you add a new node to the end of the list.

This is why the worst-case complexity of a hash table is `O(n)`. If a terrible hash function maps every single key to the same index, the hash table effectively degrades into a single, slow linked list.

JavaScript's Implementations: `Object` vs. `Map`

- `Object`: The classic implementation. Keys are coerced to strings or symbols. It's highly optimized by the V8 engine for many use cases.
- `Map`: A more modern and often superior implementation.
  - **Key Types:** `Map` can use *any* data type as a key, including objects and functions, without string coercion.
  - **Performance:** For scenarios involving frequent additions and deletions of keys, `Map` can be significantly faster.
  - **Features:** `Map` has a built-in `.size` property, is directly iterable, and guarantees that it maintains the insertion order of its elements.

Module 2: Trees - The Hierarchical Structure

Tree Terminology

- **Root:** The single topmost node.
- **Parent:** A node that has a reference to other nodes.
- **Child:** A node that is referenced by a parent.
- **Leaf:** A node with no children.
- **Edge:** The connection between two nodes.

- **Height:** The length of the longest path from the root to a leaf.
- **Depth:** The length of the path from the root to a specific node.

Binary Search Trees (BSTs) A BST is a node-based binary tree data structure which has the following properties: * The left subtree of a node contains only nodes with keys lesser than the node's key. * The right subtree of a node contains only nodes with keys greater than the node's key. * The left and right subtree each must also be a binary search tree.

This ordering principle is what allows for `O(log n)` search times.

Implementation and Performance **Search (`find`):** 1. Start at the root. 2. Compare the target value with the current node's value. 3. If they match, you've found it. 4. If the target is *less*, move to the `left` child. 5. If the target is *greater*, move to the `right` child. 6. Repeat until you find the value or you hit `null` (meaning the value isn't in the tree).

Insertion (`insert`): You follow the exact same logic as `find`. You traverse the tree until you find a `null` spot where the new node should be placed, and then you insert it there.

The Catch: Balanced vs. Unbalanced Trees The `O(log n)` performance is only guaranteed if the tree is **balanced**. If you insert sorted data (`1, 2, 3, 4, 5`) into a simple BST, it will create a completely unbalanced tree that looks and performs exactly like a linked list, degrading all operations to `O(n)`. Self-balancing trees (like AVL trees or Red-Black trees) are more complex data structures that automatically restructure themselves on insertion to maintain a balanced state and guarantee `O(log n)` performance.

Module 3: Graphs - The Network Model

Graph Terminology

- **Vertex (or Node):** A point or entity in the graph.
- **Edge (or Link):** A connection between two vertices.
- **Directed Graph:** Edges have a direction (e.g., Twitter, where you can follow someone without them following you back).
- **Undirected Graph:** Edges are bidirectional (e.g., Facebook, where friendship is mutual).
- **Weighted Graph:** Edges have a "cost" or "weight" associated with them (e.g., a map where edges are roads and the weight is the distance).

Representing Graphs: Adjacency List vs. Adjacency Matrix

- **Adjacency Matrix:** A 2D array (a matrix) where `matrix[i][j] = 1` if there is an edge between vertex `i` and vertex `j`, and `0` otherwise. **Pros:** Fast to check if an edge exists. **Cons:** Uses a lot of space (V^2), which is inefficient for sparse graphs (graphs with few edges).
- **Adjacency List:** An array (or hash map) where the index corresponds to a vertex, and the value is a list of all vertices it is connected to. **Pros:** Space-efficient for sparse graphs. **Cons:** Slower to check if a specific edge exists (requires searching the list). This is the most common representation in practice.

Module 4: Fundamental Searching & Sorting Algorithms

Searching: Linear vs. Binary Search

- **Linear Search:** Simple but slow. It iterates through every element from start to finish. It has a time complexity of **O(n)** and works on any unsorted list.
- **Binary Search:** Fast but requires a sorted list. It repeatedly divides the search interval in half. It has a time complexity of **O(log n)**.

Simple Sorting: Bubble Sort & Insertion Sort

- **Bubble Sort:** The simplest, but most naive sorting algorithm. It makes multiple passes through the array, repeatedly swapping adjacent elements that are out of order. It has a time complexity of $\mathbf{O(n^2)}$ and is almost never used in practice.
- **Insertion Sort:** Builds the final sorted array one item at a time. It's like sorting a hand of playing cards. It iterates through the input and, for each element, finds its correct place in the already-sorted part of the array and inserts it there. It also has a time complexity of $\mathbf{O(n^2)}$, but it's very efficient for small or nearly-sorted arrays.

Efficient Sorting: Merge Sort & Quick Sort These are the "divide and conquer" algorithms that are used in real-world applications.

- **Merge Sort:**
  1. **Divide:** Recursively split the array in half until you have arrays of one element (which are inherently sorted).
  2. **Conquer:** Merge the sorted sub-arrays back together. A helper function merges two sorted arrays into a single sorted array.
  - **Performance:** It has a guaranteed time complexity of **O(n log n)** in all cases. Its main drawback is that it requires **O(n)** extra space for the temporary arrays used during the merge step.
- **Quick Sort:**
  1. **Divide:** Pick an element from the array (the "pivot").
  2. **Conquer:** Reorder the array so that all elements with values less than the pivot come before it, while all elements with values greater than the pivot come after it (the "partition" step).
  3. Recursively apply the above steps to the sub-arrays of elements with smaller and greater values.
  - **Performance:** Its average-case time complexity is **O(n log n)**. However, its worst-case complexity (which occurs with a poor pivot choice on already-sorted data) is $\mathbf{O(n^2)}$. In practice, with a good pivot strategy (like picking a random element), it is often faster than Merge Sort because it can sort "in-place" with only **O(log n)** space complexity for the recursion stack.

Take-Home Assessment: The Sorting Algorithm Analyzer

Objective: To demonstrate a deep understanding of sorting algorithms by implementing and analyzing their performance.

The Task: Create a TypeScript project to implement and compare two sorting algorithms: Insertion Sort and Merge Sort.

1. Implement the Algorithms: * Create a file `sorting.ts`. * Inside, write a function `insertionSort(arr: number[]): number[]`. * Write a function `mergeSort(arr: number[]): number[]`. You will likely need a helper function for the merging step.

2. Create the Analyzer: * Create a file `analyzer.ts`. * Import your sorting functions. * Create a large, unsorted array of random numbers (e.g., 10,000 items). * Use JavaScript's `performance.now()` or `console.time()` / `console.timeEnd()` to measure how long each sorting function takes to sort the large array. * Log the results to the console, comparing the execution times.

3. Write a `README.md` file: * In this file, briefly explain the results you observed. Why was one algorithm so much faster than the other? What are the Big O complexities of each, and how do your results reflect that?

Submission: Submit your entire project via a Pull Request to your personal assignments repository. ‘