Month 6, Week 1: The Unabridged Learning Guide

Introduction: The Science of Engineering

For the past five months, we have been training as software architects. We have mastered the tools, the languages, the frameworks, and the patterns necessary to build complex, scalable, and secure backend applications. We have learned how to make our code *work*. This month, we learn how to make it *work well*. We transition from being skilled craftspeople to being true computer scientists.

This week, we dive into the foundational science of our profession: **Algorithms and Data Structures (DSA)**.

This is not an abstract academic exercise. The choice of a data structure and the design of an algorithm are the most fundamental performance decisions an engineer can make. A brilliant choice can allow an application to scale to millions of users; a poor choice can cause it to crumble under a fraction of that load.

We will begin by learning the universal language for analyzing performance: **Big O Notation**. This mathematical notation allows us to reason about an algorithm's efficiency and predict how it will behave as the input size grows. We will then use this new lens to analyze the linear data structures that form the bedrock of so many applications. We will revisit **Arrays** to understand their performance characteristics on a deeper level. We will then introduce **Linked Lists**, exploring their unique trade-offs.

Finally, we will explore two fundamental **Abstract Data Types**: **Stacks (LIFO)** and **Queues (FIFO)**. We will learn their rules, their real-world use cases (like the JavaScript call stack and event loop), and how to implement them efficiently using the primitive structures we've already mastered. Every concept will be implemented in TypeScript to maintain our professional standard of type safety.

Table of Contents

Module 1: The Language of Performance - Big O Notation

Why Time and Space Matter As a senior architect, your concerns go beyond simple functionality. You must consider how your application will perform under load. * **Time Complexity:** How does the execution time of an algorithm change as the input data grows? An algorithm that is fast for 100 users might be unacceptably slow for 1,000,000 users. * **Space Complexity:** How does the amount of memory an algorithm needs change as the input data grows? An algorithm that uses a few megabytes for a small input might consume gigabytes for a large one, crashing your server.

Big O notation is the standard for discussing these complexities. It describes the **asymptotic behavior** of a function—its rate of growth as the input size `n` approaches infinity. It focuses on the "big picture" and ignores constants and less significant terms.

Core Complexity Classes: O(1), O(log n), O(n), O(n log n), O(n²)

- **O(1) - Constant Time:** The best possible complexity. The number of operations does not change, no matter the size of `n`.
- **O(log n) - Logarithmic Time:** Excellent. The number of operations increases by one each time `n` doubles. This is characteristic of "divide and conquer" algorithms.
  - **Example: Binary Search.** To find an item in a sorted array of 1,000,000 items, you only need about 20 comparisons.
- **O(n) - Linear Time:** Good. The number of operations grows in a direct, one-to-one relationship with `n`.
- **O(n log n) - Log-Linear Time:** Very good. This is the complexity of the most efficient sorting algorithms (like Merge Sort and Quick Sort).
- **O(n²) - Quadratic Time:** Becomes slow very quickly. Often involves a nested loop over the same collection. An input of 10,000 items could mean 100,000,000 operations.
- **O(2) and O(n!) - Exponential and Factorial Time:** Extremely slow. These are generally unacceptable for anything but the smallest inputs.

Module 2: Linear Data Structures

Arrays: A Deep Dive into Performance

- **Memory Layout:** Arrays are a contiguous block of memory. This means `arr[0]`, `arr[1]`, `arr[2]`, etc., are all located right next to each other in RAM.
- **Time Complexity:**
  - **Access (`arr[i]`):** `O(1)`. Because of the contiguous memory, the computer can perform a simple mathematical calculation `(start_address + index * item_size)` to instantly find the location of any element.
  - **Search:** `O(n)`. Without knowing the index, you must check each element one by one in the worst case.
  - **Push/Pop (End):** `O(1)`. Adding or removing from the end is very fast.
  - **Unshift/Shift (Beginning):** `O(n)`. Adding or removing from the beginning requires re-indexing every single element in the array.

Linked Lists: The Chain of Nodes A linked list is made of `Node` objects that are not stored contiguously in memory. Each node has two parts: its `value` and a `next` pointer that holds the memory address of the next node.

TypeScript Implementation:

```
class ListNode<T> {
    public value: T;
    public next: ListNode<T> | null = null;

    constructor(value: T) {
        this.value = value;
    }
}

class LinkedList<T> {
    public head: ListNode<T> | null = null;

    // ... methods to add, remove, and find nodes
}
```

Arrays vs. Linked Lists: The Core Trade-off

Operation	Array Time Complexity	Linked List Time Complexity	Winner
Indexed Access	`O(1)`	`O(n)`	**Array**
Search (by value)	`O(n)`	`O(n)`	(Tie)
Insertion (start)	`O(n)`	`O(1)`	**Linked List**
Deletion (start)	`O(n)`	`O(1)`	**Linked List**
Insertion (end)	`O(1)`	`O(1)`*	**(Tie)**
Deletion (end)	`O(1)`	`O(n)`**	**Array**

* *O(1) for a linked list if you maintain a* `tail` *pointer.* ** *O(n) because to find the new tail, you must traverse from the head to the second-to-last node.*

Module 3: Abstract Data Types - Stacks and Queues

An Abstract Data Type (ADT) is a theoretical model. It defines a set of operations but does not specify how those operations are implemented.

Stacks: Last-In, First-Out (LIFO)

- **Operations:**
  - `push(item)`: Add to the top.
  - `pop()`: Remove from the top.
  - `peek()`: View the top item.
  - `isEmpty()`: Check if the stack is empty.

- **Implementation:** An array is an excellent choice because `push` and `pop` are both `O(1)`.

TypeScript Stack Implementation:

```typescript
class Stack<T> {
    private storage: T[] = [];

    push(item: T): void {
        this.storage.push(item);
    }

    pop(): T | undefined {
        return this.storage.pop();
    }
    // ... other methods
}
```

Queues: First-In, First-Out (FIFO)

- **Operations:**
  - `enqueue(item)`: Add to the back.
  - `dequeue()`: Remove from the front.
  - `peek()`: View the front item.
  - `isEmpty()`: Check if the queue is empty.
- **Implementation with an Array:** A naive implementation using `push` and `shift` is simple to write but inefficient. The `shift()` operation is `O(n)`, which can be a bottleneck.
- **A More Performant Implementation:** For a high-performance queue, a **Linked List** is the superior choice. Enqueueing is adding a new `tail`, and dequeueing is removing the `head`, both of which are `O(1)` operations.

Take-Home Assessment: The Infix to Postfix Converter

Objective: To demonstrate a deep understanding of the Stack data structure by implementing a classic computer science algorithm.

The Task: In mathematics, "infix" notation is the way we normally write expressions (e.g., `3 + 4`). "Postfix" or "Reverse Polish Notation" (RPN) is a way of writing expressions where the operator comes *after* the operands (e.g., `3 4 +`). It is easier for computers to parse. Your task is to write a function that converts an infix expression to a postfix expression using a Stack. This is a variation of Shunting-yard algorithm.

1. Create a `Stack` class in TypeScript.

2. Write the `infixToPostfix(expression: string): string` function. * You will need a stack to hold operators and a precedence map for `+`, `-`, `*`, `/`. * Loop through each token in the input expression string. * **If the token is a number:** Append it to your result string. * **If the token is an operator:** While the stack is not empty and the operator at the top of the stack has higher or equal precedence, pop from the stack and

append to your result. After the loop, push the current operator onto the stack. * **If the token is an open parenthesis `(`:** Push it onto the stack. * **If the token is a close parenthesis `)`:** Pop from the stack and append to your result until you find the matching open parenthesis. * After the main loop, pop any remaining operators from the stack and append them to the result.

Test Cases: * `"3 + 4 * 2"` should become `"3 4 2 * +"` * `"( 3 + 4 ) * 2"` should become `"3 4 + 2 *"`

Submission: Submit your TypeScript file containing the `Stack` class and the conversion function via a Pull Request to your personal assignments repository. ‘