Trees are a commonly-encountered data structure in Front End development because the DOM is a tree, so if you have to prioritize, definitely focus on being good at tree data structures and common algorithms for trees, such as Breadth-first Search and Depth-first Search.

Data structure plays a really important role in managing your apps which could be frontend like HTML/ CSS/ JavaScript or in Backend e.g. Database as MongoDB/ SQL/ Firebase or dealing with load time of your application. Yes it’s not an easy task to manage your site/app because Think of a situation where you need to search someone on Instagram then you would not want your user to wait for lot of time and will implement backend in such a way that queries are faster.

First of all, algorithms also have many applications in the front-end project. For example:

* VirtualDOM is one of the core mechanisms in React and Vue. It needs to use the hash table data structure and diff algorithm.
* When parsing template syntax and generating AST, we need to use the tree data structure.
* The browser’s Browers History, as well as various undo and redo functions, need to use the stack.

So the algorithm is definitely useful to us front-end developers.

Core JavaScript is the basis of the JavaScript language. JavaScript is a dynamic computer programming language. It is lightweight and most commonly used as apart from web pages, whose implementations allow a client-side script to interact with the user and make dynamic pages.

The JavaScript language has the following components:

**Core JavaScript:**

Core JavaScript is the basic of the JavaScript language supports both the client and server side.

**Client-side javascript (CSJS)**

The client-side JavaScript has core JavaScript elements.  For functioning, it also has properties and methods, which helps developers.

**Server-side JavaScript (SSJS)**

Server-side JavaScript, as the name suggests runs on server-side. The core is part of client-side as well as server-side JavaScript. It includes the core JavaScript elements.

**Prototype-based programming**

**Prototype-based programming** is a style of [object-oriented programming](https://developer.mozilla.org/en-US/docs/Glossary/OOP) in which [classes](https://developer.mozilla.org/en-US/docs/Glossary/Class) are not explicitly defined, but rather derived by adding properties and methods to an instance of another class or, less frequently, adding them to an empty object.

In simple words: this type of style allows the creation of an [object](https://developer.mozilla.org/en-US/docs/Glossary/Object) without first defining its [class](https://developer.mozilla.org/en-US/docs/Glossary/Class).

### What is object-oriented programming?

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior.

The **event loop** is the secret behind JavaScript’s asynchronous programming. JS executes all operations on a single thread, but using a few smart data structures, it gives us the illusion of multi-threading. Let’s take a look at what happens on the back-end.

A **Promise** is a proxy for a value not necessarily known when the promise is created. It allows you to associate handlers with an asynchronous action's eventual success value or failure reason.

A Promise is in one of these states:

* *pending*: initial state, neither fulfilled nor rejected.
* *fulfilled*: meaning that the operation was completed successfully.
* *rejected*: meaning that the operation failed.

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables that makes it easier to compose asynchronous or callback-based code.

Following is a description of each lifecycle hook.

* **ngOnChanges** − When the value of a data bound property changes, then this method is called.
* **ngOnInit** − This is called whenever the initialization of the directive/component after Angular first displays the data-bound properties happens.
* **ngDoCheck** − This is for the detection and to act on changes that Angular can't or won't detect on its own.
* **ngAfterContentInit** − This is called in response after Angular projects external content into the component's view.
* **ngAfterContentChecked** − This is called in response after Angular checks the content projected into the component.
* **ngAfterViewInit** − This is called in response after Angular initializes the component's views and child views.
* **ngAfterViewChecked** − This is called in response after Angular checks the component's views and child views.
* **ngOnDestroy** − This is the cleanup phase just before Angular destroys the directive/component.

The different types of Angular directives are as follows:

| DIRECTIVE TYPES | DETAILS |
| --- | --- |
| [Components](https://angular.io/guide/component-overview) | Used with a template. This type of directive is the most common directive type. |
| [Attribute directives](https://angular.io/guide/built-in-directives#built-in-attribute-directives) | Change the appearance or behavior of an element, component, or another directive. |
| [Structural directives](https://angular.io/guide/built-in-directives#built-in-structural-directives) | Change the DOM layout by adding and rem |

1. The @[Directive](https://angular.io/api/core/Directive)() decorator's configuration property specifies the directive's CSS attribute selector, [appHighlight].
2. Import [ElementRef](https://angular.io/api/core/ElementRef) from @angular/core. [ElementRef](https://angular.io/api/core/ElementRef) grants direct access to the host DOM element through its nativeElement property.
3. Add [ElementRef](https://angular.io/api/core/ElementRef) in the directive's constructor() to [inject](https://angular.io/guide/dependency-injection) a reference to the host DOM element, the element to which you apply appHighlight.
4. Add logic to the HighlightDirective class that sets the background to yellow.

| CONTENT PROJECTION | DETAILS |
| --- | --- |
| [Single-slot content projection](https://angular.io/guide/content-projection#single-slot) | With this type of content projection, a component accepts content from a single source. |
| [Multi-slot content projection](https://angular.io/guide/content-projection#multi-slot) | In this scenario, a component accepts content from multiple sources. |
| [Conditional content projection](https://angular.io/guide/content-projection#conditional) | Components that use conditional content projection render content only when specific conditions are met. |

**Transclusion**more commonly known as **Content Projection** is a vastly powerful and useful directive within AngularJS and upwards. Directives are classes that add additional behaviour to elements in your Angular applications.

**Content projection** allows a directive to make use of templates while still being able to clone the original content and add it to the Document Object Model, also referred to as the DOM, which is a programming interface for HTML and XML documents and it represents a page where programs can change the document structure, style, and content.

To handle the navigation from one [view](https://angular.io/guide/glossary#view) to the next, you use the Angular [Router](https://angular.io/api/router/Router). The [Router](https://angular.io/api/router/Router) enables navigation by interpreting a browser URL as an instruction to change the view.

Dependency Injection is often more simply referred to as DI. The paradigm exists throughout Angular. It keeps code flexible, testable, and mutable. Classes can inherit external logic without knowing how to create it. Any consumers of those classes also do not need to know anything.

DI saves classes and consumers alike from having to know more than necessary. Yet the code is as modular as it was before thanks to the mechanisms supporting DI in Angular.