Introduction to Classes and Object Oriented Programming (OOP)

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - What Object Oriented Programming (OOP) is
> - What ES6 classes are, and why they're useful
> - The difference between OOP with ES6 classes and the old prototype-based approach
>
> ### Introduction
>
> ES6 classes are a significant feature of JavaScript that allows developers to create reusable, organized, and more manageable code. Prior to classes being introduced in ES6, developers primarily created similar functionality using constructor functions. Classes, however, provide a new syntax that tends to make the same code easier to reason about and manage. In this lesson, we will do a high-level overview of both classes and object oriented programming, and have a look at the benefits of this approach.

What is Object Oriented Programming?

> Object-oriented programming (OOP) is a programming paradigm that organizes data and behavior into objects that interact with one another. In OOP, objects are created from classes, which are essentially blueprints that define the properties and methods of the objects they generate. This allows for efficient and modular code, as objects can be reused and modified as needed.
>
> JavaScript also supports object oriented programming. In JavaScript, objects (in the more broad OOP sense, not the object data type which we covered in previous modules) can be created using either constructor functions or classes. Classes were introduced in ES6 as a new syntax for creating these objects and defining their properties and methods. Both constructor functions and classes allow for the creation of multiple object instances (a concept we'll come back to) with the same set of properties and methods, which can improve code organization and maintainability. Overall, OOP is a powerful and widely used programming paradigm that allows for efficient and modular code design.
>
> ### Overview of Classes
>
> Classes are essentially a "syntactic sugar" over JavaScript's existing prototype-based inheritance model. This means that while the underlying mechanics of JavaScript's version of an object oriented programming model remain the same, ES6 classes provide a simpler and more intuitive way to go about things.

ES6 classes consist of several components, including:
- **Declaration -** The class declaration is used to define a new class using the class keyword
- **Constructor -** The class constructor is a special method that is executed when a new instance of the class is created, and is used to initialize the object's properties.
- **Properties -** Properties are variables that belong to the class which store information important to that class
- **Methods -** Methods are functions that belong to the class and can be called on instances of the class
- **Inheritance -** Inheritance allows a new class to be based on an existing class, inheriting its properties and methods while also allowing for customization

>

When using ES6 classes, new instances can be created using the “new” keyword and the class name. This creates a new instance of the class, with its own set of properties and methods. As with prototype-based inheritance in JavaScript, properties and methods defined on the class are shared across all instances of the class. However, each instance has its own set of properties that can be modified independently.
>
Benefits of Using Classes
ES6 classes provide several benefits over traditional prototype-based OOP in JavaScript. Some of these benefits include:
- Simpler and cleaner syntax for defining classes and creating instance objects
- Clearer and more intuitive inheritance and subclassing
- Easier to understand and maintain code
- Better support for encapsulation and data privacy
>
Syntax of Classes
ES6 classes are defined using the **class** keyword, followed by the name of the class, and a set of curly braces containing the class definition. The class definition can include a constructor, properties, and methods.
>
The constructor method is called when a new instance of the class is created and is used to initialize the object’s properties. It is defined using the **constructor** keyword followed by a set of parentheses containing any arguments that need to be passed in.

```
// Syntax example
class ExampleClass {
  constructor(prop1, prop2) {
    this.prop1 = prop1
    this.prop2 = prop2
  }
}
```

Classes vs. ES5 Prototype-based OOP

> Before the introduction of ES6 classes, JavaScript used a prototype-based model for implementing OOP. This model was based on the concept of prototypes, which are essentially objects that serve as templates for other objects. Developers would create objects by cloning or extending existing prototypes, which could be a complex and error-prone process.

```
// ES5 prototype-based OOP example
function Car(make, model, year) {
  this.make = make
  this.model = model
  this.year = year
}


Car.prototype.drive = function() {
  return `The ${this.make} ${this.model} is driving...`
}


var myCar = new Car('Toyota', 'Corolla', 2020)
myCar.drive() // Output: 'The Toyota Corolla is driving...'
```

> You can see our example above is already in the javascript file in the IDE. If you add a `console.log()` statement, you can see the output; otherwise the output just “exists” but doesn’t go anywhere unless you have somewhere set up to display it.
>
> Try adding `console.log(myCar.drive())` at the bottom and refreshing the preview window.
>
> ES6 classes provide a more intuitive and structured approach to OOP that is similar to other object-oriented languages. One of the key differences between ES6 classes and ES5 prototype-based OOP is the syntax. ES6 classes provide a simpler and more organized syntax for defining classes and creating objects, making it easier to read and understand code.
>
> Another key difference is the support for inheritance and subclassing. In ES5, inheritance was implemented through a complex system of prototype chaining, which could be difficult to understand and maintain. In ES6, inheritance is implemented using the extends keyword, which provides a more structured and efficient way to implement inheritance and subclassing.

Knowledge Check

Conclusion & Takeaway

> ES6 classes provide a more intuitive and organized approach to object-oriented programming in JavaScript. They offer a simpler and more structured syntax for defining classes and creating objects, making it easier to read and understand code. Classes also offer better support for encapsulation and data privacy, and more efficient and modular code design. Understanding the benefits of classes and object-oriented programming in JavaScript can greatly improve a developer's ability to create reusable, organized, and more manageable code.

Classes and OOP: Deep Dive

> ### Goals
>
> By the end of this lesson, you should understand:
> - The basics of defining classes and constructors in ES6
> - How to create and access class properties and methods
> - How to implement inheritance, polymorphism and other OOP concepts
>
> ### Introduction
>
> Having gotten a basic understanding of classes and object oriented programming in the previous section, we're now going to explore in depth how the mechanics of all this works and how we can start implementing classes and OOP in our own projects. We will cover the different ways to define classes, how to create properties and methods on our classes and how to work with them, the concepts of encapsulation, polymorphism, abstraction and inheritance, as well as an alternative to inheritance called "composition".

Defining Classes Part 1

> In order to define classes, we use the **class** keyword. A class is a blueprint for creating objects which contain properties and methods that define the behavior of those objects. Often, these objects will have what is called a **constructor** method, which is a special method that runs when an instance of that particular object is created and initializes its properties.
>
> Once defined, we can create class instances by using the **new** keyword followed by the name of the class.
>
> Let's explore some examples of defining classes (with and without constructors) in more detail, as well as creating instances of those classes.
>
> ### Defining a class with a constructor
>
> Here is an example of defining a class with a constructor:

```
// Define a class with a constructor
class Person {
  constructor(name, age) {
    this.name = name
    this.age = age
  }


  greet() {
    console.log(`Hello, my name is ${this.name} and I am
        ${this.age} years old.`)
  }
}


const john = new Person('John', 30)
console.log('CLASS WITH CONSTRUCTOR - OUTPUT: \n')
john.greet() // Output: Hello, my name is John and I am 30 years
        old.
```

In the example above, we define a class called Person with a constructor that takes two arguments, name and age. We use the **this** keyword to initialize the name and age properties of the object. We also define a greet method that logs a message to the console.
>

Defining Classes Part 2

Defining a class without a constructor

Here is an example of defining a class without a constructor:

```
// Define a class without a constructor
class Person2 {
  name = ""
  age = 0
  gender = ""


  getDetails() {
    return `Name: ${this.name}, Age: ${this.age}, Gender:
        ${this.gender}`
  }
}
```

In this example, we have defined a class called “Person” that has three properties (name, age, and gender) and one method (**getDetails**). Notice that we did not define a constructor for this class, so the properties are initialized with default values of an empty string for “name”, zero for “age”, and an empty string for “gender”.
>
In order to change this, we would have to do so as follows:

```
const person = new Person2()
person.name = "Alice"
person.age = 30
person.gender = "female"


console.log('\n\nCLASS WITHOUT CONSTRUCTOR - OUTPUT: \n')
console.log(person.getDetails()) // Output: "Name: Alice, Age:
        30, Gender: female"
```

The **getDetails** method returns a string containing the values of all three properties. This method can be called on instances of the “Person2” class to get details about the specific person.
>
By omitting the constructor method, we have created a class with default property values that can be overridden when instances of the class are

created. This approach can be useful in cases where default property values are sufficient, and no additional setup is needed when instances are created.

Encapsulation

> Encapsulation is a fundamental concept in object-oriented programming (OOP) that refers to the bundling of data and methods that manipulate that data within a single unit, such that the internal workings of that unit are hidden from outside code. In JavaScript, encapsulation is achieved using various techniques, such as closures, IIFEs, and private properties and methods in classes.
>
> Since this lesson is about classes, here will will just go over an example of using private properties and methods:

```
// Private properties and methods
class Counter {
  #count = 0

  #increment(value) {
    if (value) {
      this.#count += value
    } else {
      this.#count++
    }
  }

  #decrement(value) {
    if (value) {
      this.#count -= value
    } else {
      this.#count--
    }
  }

  #getCount() {
    return this.#count
  }

  increment(value) {
    this.#increment(value)
  }

  decrement(value) {
    this.#decrement(value)
  }
```

```
  getCount() {
    return this.#getCount()
  }
}


const counter = new Counter()
counter.increment()
counter.increment(4)
counter.decrement(2)
counter.decrement()

console.log('\n\nENCAPSULATION EXAMPLE - OUTPUT: \n')
console.log('getCount result: ' + counter.getCount()) // Output:
        2
```

> One benefit of employing encapsulation is preventing certain inner workings of our code from being accessed or changed directly, outside of a specific controlled interface we define using public methods, which call help make our code more secure. Now, this is not foolproof, but it's definitely a help!

Inheritance Part 1

Inheritance is a fundamental concept in object-oriented programming that allows us to define new classes based on existing classes. The new class inherits all the properties and methods of the existing class and can also have its own properties and methods. This can be a powerful way to reuse code and create more complex class hierarchies.
>
JavaScript uses what's called **prototypal inheritance** which is based on the concept of delegation. This means an object delegates properties and methods to its prototype object. When a property or method is accessed on an object, JavaScript first checks if the property or method exists on the object itself. If it does not, JavaScript then looks for the property or method on the object's prototype object. If the property or method is found on the prototype object, it is returned. If it is not found on the prototype object, JavaScript continues the lookup process on the prototype object's prototype object, and so on, until the property or method is found or the end of the prototype chain is reached.
>
The basic form of inheritance when it comes to ES6 classes is simply by creating a new class that extends an existing class, using the **extends** keyword:

```
// Basic inheritance example
class Animal {
  constructor(name) {
    this.name = name
  }

  speak() {
    console.log(`${this.name} makes a noise.`)
  }
}


class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`)
  }
}


const dog = new Dog('Max')

console.log('\n\nBASIC INHERITANCE EXAMPLE - OUTPUT: \n')
dog.speak()  // Output: "Max barks."
```

In the example above, we define an Animal class with a constructor that initializes the name property of the object and a speak method that logs a message to the console. We then define a Dog subclass that extends the Animal class. The Dog class has its own speak method that logs a different message to the console.

Inheritance Part 2

Calling parent class methods

In a subclass, we can call methods of the parent class using the super keyword. This can be useful when we want to reuse code from the parent class or when we want to override a method but still call the parent method.
>
Here is an example of calling a parent class method:

```
// Parent class Animal2
class Animal2 {
  constructor(name) {
    this.name = name
  }

  speak() {
    console.log(`${this.name} makes a noise.`)
  }
}

// Subclass Dog2 extending Animal2
class Dog2 extends Animal2 {
  speak() {
    super.speak() // Calling the parent class method using super
        keyword
    console.log(`${this.name} barks.`)
  }
}

const dog2 = new Dog2('Lilly')
console.log('\n\nPARENT CLASS METHODS - OUTPUT: \n')
dog2.speak()  // Output: "Lilly makes a noise." "Lilly barks."
```

> In the example above, we define a Dog2 subclass that extends the Animal2 class. The Dog class has its own speak method that calls the speak method of the parent class using the **super** keyword and then logs a different message to the console.

Using multiple levels of inheritance

Inheritance can also be used to create multiple levels of class hierarchies. This allows us to reuse code across multiple classes and create more complex and specialized classes.
>
Here is a basic example of using multiple levels of inheritance:

```
class Animal3 {
  constructor(name) {
    this.name = name
  }

  speak() {
    console.log(`${this.name} makes a noise.`)
  }
}


class Dog3 extends Animal3 {
  speak() {
    console.log(`${this.name} barks.`)
  }
}


class Labrador extends Dog3 {
  speak() {
    console.log(`${this.name} barks loudly.`)
  }
}


const animal3 = new Animal3('Buddy')
const dog3 = new Dog3('Mitzie')
const labrador = new Labrador('Oliver')

console.log('\n\nMULTIPLE LEVELS OF INHERITANCE - OUTPUT: \n')
animal3.speak() // Output: Buddy makes a noise.
dog3.speak() // Output: Mitzie barks.
labrador.speak() // Output: Oliver barks loudly.
```

In the example above, we define an Animal3 class with a constructor that initializes the name property of the object and a speak method that logs a message to the console. We then define a Dog3 subclass that overrides the speak method of the parent class with its own implementation that logs a different message to the console. Finally, we define a Labrador subclass

that extends the Dog3 class and overrides its speak method with its own implementation that logs a different message to the console.
>

Inheritance Part 3

There's an issue with the example we just saw, however. What if we have multiple parent classes and we want to create a new class that extends a few of them but not all of them – meaning there are some classes whose properties and/or methods we don't really need? Well, using the current implementation this isn't possible.
>
Now, you might be thinking you can just use the **extends** keyword and list several classes that the new class should inherit from, along the lines of the following:

```
class Labrador extends Animal3, Dog3 {
  /* REST OF THE CODE HERE */
}
```

Well... this wouldn't work. JavaScript has no built-in support for implementing multiple inheritance this way, which is due to its use of **prototypal inheritance** under the hood.
>
This is in contrast to other programming languages like C++, where the above syntax *would* be possible.

That being said, if we want a given class to inherit properties and methods from multiple other classes, one way we can achieve similar functionality in JavaScript is to use **composition**.

Composition

In **composition**, or at least in one way of implementing it, rather than inheriting from multiple classes we instead create new classes which utilize instances of other classes in the constructor. This allows for greater flexibility and modularity in our code.
>
Here is an example of implementing composition in JavaScript:

```
// Define the Animal class that provides basic animal behavior
class Animal4 {
  constructor(name) {
    this.name = name
  }

  eat() {
    console.log(`${this.name} is eating.`)
  }

  sleep() {
    console.log(`${this.name} is sleeping.`)
  }
}


// Define the Flyer class that provides the ability to fly
class Flyer {
  constructor(name) {
    this.name = name
  }

  fly() {
    console.log(`${this.name} is flying.`)
  }
}


// Define the Swimmer class that provides the ability to swim
class Swimmer {
  constructor(name) {
    this.name = name
  }

  swim() {
```

```
    console.log(`${this.name} is swimming.`)
  }
}


// Define the Duck class that uses instances of Animal4, Flyer,
        and Swimmer to provide its functionality
class Duck {
  constructor(name) {
    this.name = name
    this.animal = new Animal4(name)
    this.flyer = new Flyer(name)
    this.swimmer = new Swimmer(name)
  }

  eat() {
    this.animal.eat()
  }

  sleep() {
    this.animal.sleep()
  }

  fly() {
    this.flyer.fly()
  }

  swim() {
    this.swimmer.swim()
  }
}


// Create a new Duck instance and call its eat(), fly(), swim(),
        and sleep() methods
const duck = new Duck('Donald')

console.log('\n\nCOMPOSITION EXAMPLE - OUTPUT: \n')
duck.eat() // Output: Donald is eating.
duck.fly() // Output: Donald is flying.
duck.swim() // Output: Donald is swimming.
duck.sleep() // Output: Donald is sleeping.
```

In this example, we created three classes (**Animal4**, **Flyer**, and **Swimmer**) which represent the different base functionalities. We then create a **Duck** class that does not inherit from these classes, but instead has instances of **Animal4**, **Flyer**, and **Swimmer** as properties.
>

We then define methods that delegate functionality to the respective instances of **Animal4**, **Flyer**, and **Swimmer**. This allows us to selectively use different functionality in different instances of our class. For example, we can easily create a new class that only has the **Animal4** and **Flyer** functionality, by omitting the **Swimmer** instance.
>

Polymorphism

> Polymorphism is a concept in object-oriented programming that refers to the ability of objects of different classes to be used interchangeably, even though they have different implementations of the same method or property. Polymorphism is important in programming because it helps to make code more flexible and adaptable to different situations.

In JavaScript, polymorphism can be implemented using method overriding and method overloading. Method overriding is when a subclass provides its own implementation of a method that is already defined in its superclass. Method overloading is when a class provides multiple implementations of a method with different parameter lists.
>
Method Overriding
We've actually already seen an example of method overriding previously where we created an **Animal** class with a **speak** method that outputs a generic message, then created a **Dog** subclass that overrides the **speak** method with its own implementation that outputs a specific message.
>
Since we've already seen an example of overriding methods, we'll move on to look at method overloading.
>
Method Overloading

```
// Now let's look at an example using method overloading:
class MyClass {
  myMethod() {
    if (arguments.length === 0) {
      console.log('No arguments passed')
    } else if (arguments.length === 1 && typeof arguments[0] ===
        'number') {
      console.log(`One number argument: ${arguments[0]}`)
    } else if (arguments.length === 1 && typeof arguments[0] ===
        'string') {
      console.log(`One string argument: ${arguments[0]}`)
    } else {
      console.log('Invalid arguments')
    }
  }
}

const instance = new MyClass()

console.log('\n\nPOLYMORPHISM (METHOD OVERLOADING) - OUTPUT:
        \n')
instance.myMethod()  // Output: "No arguments passed"
instance.myMethod(42)  // Output: "One number argument: 42"
instance.myMethod('hello')  // Output: "One string argument:
        hello"
instance.myMethod(1, 2, 3)  // Output: "Invalid arguments"
```

In this example, we define a **MyClass** class with a **myMethod** method that uses the **arguments** object to detect the number and types of arguments passed to the method.
>
If no arguments are passed to the method, it logs the message "No arguments passed". If one argument of type **number** is passed, it logs the message "One number argument: `[number]`". If one argument of type **string** is passed, it logs the message "One string argument: `[string]`". If any other combination of arguments is passed, it logs the message "Invalid arguments".

Abstraction

> Abstraction is a concept in object-oriented programming that refers to the process of hiding implementation details while showing only the necessary information to the user. Abstraction is important in programming because it helps to simplify complex systems and make them easier to understand and use.
>
> Typically, abstraction can be implemented using abstract classes and interfaces. Abstract classes are classes that cannot be instantiated and can only be extended by other classes. Interfaces are contracts that define the methods and properties that a class must implement.
>
> However, JavaScript does not have built-in support for interfaces like some other programming languages do. We can still implement the principle of abstraction, though, using a class that acts like an interface:

```
class Animal5 {
  speak() {
    throw new Error('You have to implement the speak() method in
        a new instance of this class!')
  }


  move() {
    throw new Error('You have to implement the move() method in
        a new instance of this class!')
  }
}


class Dog4 extends Animal5 {
  speak() {
    console.log('The dog barks.')
  }


  move() {
    console.log('The dog runs.')
  }
}


class Cat extends Animal5 {
```

```
  speak() {
    console.log('The cat meows.')
  }


  move() {
    console.log('The cat walks.')
  }
}


console.log('\n\nABSTRACTION EXAMPLE - OUTPUT: \n')
const dog4 = new Dog4()
dog4.speak()  // Output: "The dog barks."
dog4.move()  // Output: "The dog runs."


const cat = new Cat()
cat.speak()  // Output: "The cat meows."
cat.move()  // Output: "The cat walks."
```

In this example, we define an **Animal5** class that serves as an interface, with two abstract methods: **speak()** and **move()**. We don't provide an implementation for these methods in the **Animal5** class, because they will be implemented by the classes that extend **Animal5**.
>
We then define two classes that implement the **Animal5** interface: **Dog4** and **Cat**. These classes provide their own implementations of the **speak()** and **move()** methods.
>
Finally, we create instances of the **Dog4** and **Cat** classes and call their **speak()** and **move()** methods. Because these methods are defined in the **Animal5** interface, we know that they will be implemented by any class that claims to implement the **Animal5** interface. This allows us to use polymorphism to write code that can work with any class that implements the **Animal5** interface, without needing to know the specific implementation details of each class.

Conclusion and Takeaway

> In this lesson we went over several important topics concerning ES6 classes and object oriented programming in JavaScript. First we covered how to define classes, which act as blueprints for creating objects, as well as adding properties and methods to those classes. We also went over the important concepts and implementation of encapsulation, inheritance, composition, polymorphism and abstraction. In the next lesson, we'll practice using classes by building a small project.

Practice Time - Coding Exercises

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!
>
> 1. Create a class called **Person** that has a constructor taking two parameters: **name** and **age**. The class should have a method called **greet()** that returns a string with the person's name and age, like "Hello, my name is [NAME] and I am [NUMBER] years old." Create an instance of the **Person** class and call the **greet()** method on it.
> 2. Create a subclass of **Person** called Student that has an additional property called **major**. Add a method to the **Student** class called **study()** that returns a string saying "I am studying [SUBJECT]." Create an instance of the **Student** class and call the **greet()** and **study()** methods on it.
> 3. Create an abstract class called **Animal** that has a constructor taking a **name** parameter. The class should have an abstract method called **speak()** that returns a string with the animal's sound. Create two subclasses of **Animal**: **Dog** and **Cat**. Implement the **speak()** method for each subclass to return the appropriate sound. Create an instance of each subclass and call the **speak()** method on them.

Mini Project: Calculator App

Goals

By the end of this lesson, you will:

- Have practiced using ES6 classes in a project
- Have built a simple JavaScript calculator

Introduction

Thus far, we've learned a lot about ES6 classes – how to create them, how to add properties and methods to them, plus a lot more. Now we're going to get some practice actually using them to build something. In this case, what we will be building together is a simple calculator app. This mini project will not only give you practice implementing what you've learned about classes, but will also reinforce a lot of what you've learned about JavaScript in general by building something which by definition involves a bit more complex logic – which is what programming mostly consists of, in the end.

Getting Started

As in other projects, some starter code has been provided (all of the HTML and CSS) and we'll just be focussing on writing the JavaScript. Since we'll be selecting elements from the DOM to work with, open up the HTML file and take a moment to look over the structure. Also, open the browser preview to see what our calculator will look like. Right now, it doesn't do much – but that will soon change.

Building the Calculator Part 1

> Alright, let's start coding our calculator. The first thing we need to do here is to define a class called Calculator, which will contain all of our logic:

```
// Define a class named Calculator
 class Calculator {

 }
```

Next, we need to define the constructor method for our calculator class, which is where we will initialize properties for all the DOM elements we need to select, plus some additional ones:

> **NOTE:** The code below should be **inside** the curly-brackets of the Calculator class (just below `class Calculator {`)!

```
// Define the constructor method for the Calculator class
  constructor() {
    // Initialize the calculator display elements
    this.previousOperandElement =
        document.getElementById("previous-operand")
    this.currentOperandElement =
        document.getElementById("current-operand")
    this.display = document.getElementById("display")
    this.clearButton = document.getElementById("clear")
    this.signButton = document.getElementById("sign")
    this.percentButton = document.getElementById("percent")
    this.divideButton = document.getElementById("divide")
    this.sevenButton = document.getElementById("seven")
    this.eightButton = document.getElementById("eight")
    this.nineButton = document.getElementById("nine")
    this.multiplyButton = document.getElementById("multiply")
    this.fourButton = document.getElementById("four")
    this.fiveButton = document.getElementById("five")
    this.sixButton = document.getElementById("six")
    this.subtractButton = document.getElementById("subtract")
    this.oneButton = document.getElementById("one")
    this.twoButton = document.getElementById("two")
    this.threeButton = document.getElementById("three")
    this.addButton = document.getElementById("add")
    this.zeroButton = document.getElementById("zero")
    this.decimalButton = document.getElementById("decimal")
    this.equalsButton = document.getElementById("equals")

    // Set default values for the current and previous operands
        and operation
    this.currentOperand = "0"
    this.previousOperand = ""
    this.operation = undefined

    // Update the calculator display
    this.updateDisplay()

  }
```

This last method in the constructor is to initialize the calculator display. We haven't defined this method yet, but we will shortly.

Building the Calculator Part 2

Now, below the constructor, we're going to start defining some methods which will handle all our calculator logic.

We'll start with methods to clear the display, and to delete the last character or digit:

```
// Define the method to clear the calculator
  clear() {
    this.currentOperand = "0"
    this.previousOperand = ""
    this.operation = undefined
  }

  // Define the method to delete the last digit from the current
       operand
  delete() {
    this.currentOperand =
        this.currentOperand.toString().slice(0, -1)
  }
```

Next, we'll define methods to append a number to the current operand property (which will show up in the display) and to modify the operation and update the previous and current operands:

```
// Define the method to append a new number to the current
        operand
  appendNumber(number) {
    // Check for duplicate decimal points
    if (number === "." && this.currentOperand.includes("."))
        return

    // Replace 0 with the new number if the current operand is 0
    if (this.currentOperand === "0" && number !== ".") {
      this.currentOperand = number
    } else {
      // Append the new number to the current operand
      this.currentOperand += number
    }
  }

  // Define the method to choose an operation
  chooseOperation(operation) {
    // Don't allow operation selection if current operand is
        empty
    if (this.currentOperand === "") return

    // Perform the previous operation if one exists
    if (this.previousOperand !== "") {
      this.compute()
    }

    // Set the new operation and move the current operand to the
        previous operand
    this.operation = operation
    this.previousOperand = this.currentOperand
    this.currentOperand = ""
    // Update the calculator display
    this.updateDisplay()
  }
```

Building the Calculator Part 3

Next, we will define our **compute()** method, which is where the core of our calculation logic will live:

```
// Define the method to compute the result of the current
        operation
  compute() {
    // Initialize variables for the previous and current
        operands
    let computation
    const prev = parseFloat(this.previousOperand)
    const current = parseFloat(this.currentOperand)

    // Don't allow computation if either operand is NaN
    if (isNaN(prev) || isNaN(current)) return

    // Perform the appropriate operation based on the selected
        operation
    switch (this.operation) {
      case "+":
        computation = prev + current
        break
      case "-":
        computation = prev - current
        break
      case "*":
        computation = prev * current
        break
      case "/":
        computation = prev / current
        break
      default:
        return
    }

    // Update the current operand with the computed result and
        reset the operation and previous operand
    this.currentOperand = computation.toString()
    this.operation = undefined
    this.previousOperand = ""
  }
```

> Alright, a couple more methods to go and then we can create an instance of our class and start defining event listeners. The first of the remaining methods we need to create is one to properly format a number for the display, and second is our **updateDisplay()** which will, like the name says, update the calculator display:

```
// Define the method to format a number for display on the
       calculator
  getDisplayNumber(number) {
    const stringNumber = number.toString()
    const integerDigits = parseFloat(stringNumber.split(".")[0])
    const decimalDigits = stringNumber.split(".")[1]
    let integerDisplay

    if (isNaN(integerDigits)) {
      integerDisplay = ""
    } else {
      integerDisplay = integerDigits.toLocaleString("en", {
        maximumFractionDigits: 0,
      })
    }
    if (decimalDigits != null) {
      return `${integerDisplay}.${decimalDigits}`
    } else {
      return integerDisplay
    }
  }

  // Define the method to update the calculator display
  updateDisplay() {
    // Display the appropriate content based on whether an
       operation has been selected
    if (this.operation != null) {
      this.display.value =
        `${this.getDisplayNumber(this.previousOperand)}
        ${this.operation} ${this.currentOperand ?
        this.currentOperand : ''}`
    } else {
      this.display.value =
        this.getDisplayNumber(this.currentOperand)
    }
  }
```

Building the Calculator Part 4

Great! We finished defining our **Calculator** class! Now we need to create a new instance, so we can use it to define the event listeners:

> **NOTE:** The code below should be **outside** of the Calculator class!

```
// Create a new instance of the Calculator class
const calculator = new Calculator()


// Add event listeners for each button to update the calculator
        display based on user input
calculator.clearButton.addEventListener("click", () => {
  calculator.clear()
  calculator.updateDisplay()
})

calculator.signButton.addEventListener("click", () => {
  calculator.currentOperand = calculator.currentOperand * -1
  calculator.updateDisplay()
})

calculator.percentButton.addEventListener("click", () => {
  calculator.currentOperand = calculator.currentOperand / 100
  calculator.updateDisplay()
})

calculator.divideButton.addEventListener("click", () => {
  calculator.chooseOperation("/")
  calculator.updateDisplay()
})

calculator.multiplyButton.addEventListener("click", () => {
  calculator.chooseOperation("*")
  calculator.updateDisplay()
})

calculator.subtractButton.addEventListener("click", () => {
  calculator.chooseOperation("-")
  calculator.updateDisplay()
})

calculator.addButton.addEventListener("click", () => {
  calculator.chooseOperation("+")
```

```
  calculator.updateDisplay()
})

calculator.equalsButton.addEventListener("click", () => {
  calculator.compute()
  calculator.updateDisplay()
})

calculator.zeroButton.addEventListener("click", () => {
  calculator.appendNumber("0")
  calculator.updateDisplay()
})

calculator.oneButton.addEventListener("click", () => {
  calculator.appendNumber("1")
  calculator.updateDisplay()
})

calculator.twoButton.addEventListener("click", () => {
  calculator.appendNumber("2")
  calculator.updateDisplay()
})

calculator.threeButton.addEventListener("click", () => {
  calculator.appendNumber("3")
  calculator.updateDisplay()
})

calculator.fourButton.addEventListener("click", () => {
  calculator.appendNumber("4")
  calculator.updateDisplay()
})

calculator.fiveButton.addEventListener("click", () => {
  calculator.appendNumber("5")
  calculator.updateDisplay()
})

calculator.sixButton.addEventListener("click", () => {
  calculator.appendNumber("6")
  calculator.updateDisplay()
})

calculator.sevenButton.addEventListener("click", () => {
  calculator.appendNumber("7")
  calculator.updateDisplay()
})
```

```
calculator.eightButton.addEventListener("click", () => {
  calculator.appendNumber("8")
  calculator.updateDisplay()
})

calculator.nineButton.addEventListener("click", () => {
  calculator.appendNumber("9")
  calculator.updateDisplay()
})

calculator.decimalButton.addEventListener("click", () => {
  calculator.appendNumber(".")
  calculator.updateDisplay()
})
```

Try it out

Awesome, we are done creating our calculator app. Open the browser preview again and give it a try!

Conclusion and Takeaway

> In this lesson, we built out a simple JavaScript calculator app which is capable of all the basic calculator operations (addition, subtraction, multiplication, division, and calculating a percent). While building this mini project, we practiced working with ES6 classes – creating a class, defining a constructor, creating properties and methods, accessing class properties and calling methods, etc. In the coming lessons on data structures, we will practice working with classes even more!

Data Structures: Introduction

Goals

In this lesson, you will:

- Review the basics of Big O Notation
- See an overview of the data structures we will cover
- Gain an understanding of pointers and their use in establishing relationships between elements or nodes in data structures.

Introduction

Data structures are a fundamental concept in computer science that play a crucial role in solving complex problems efficiently. The term “data structures” refers to the way of organizing and storing data so that it can be accessed and manipulated quickly and easily. JavaScript provides several built-in data structures which we have seen in previous lessons – such as arrays, objects, sets and maps. However, there are other common custom data structures to suit specific needs and use cases, which is what we will cover in this lesson. Understanding data structures is key for progressing as a developer, and also in the context of technical interviews.

Quick Review of Big O Notation

> Before we start talking about data structures, we need to revisit a concept we covered previously.
>
> If you recall from a previous module, we discussed the topic of Big O Notation which is a critical concept in computer science and programming that transcends language. Whatever language in which you are writing code (in our case, in this course, it's JavaScript) knowing or at least being familiar with Big O Notation is important.
>
> As a refresher, Big O Notation is a way to describe how fast an algorithm grows as its input (or the size of the problem it solves) grows. It is commonly used in computer science to compare the efficiency of different algorithms. It looks at efficiency in terms of two categories: time complexity, and space complexity.
>
> Here are a few standard named complexities:
>
> - **Constant** - O(1)
> - **Logarithmic** - O(log n)
> - **Linear** - O(n)
> - **Quadratic** - O(n^2)
> - **Factorial** - O(n!)
>
> For example, if an algorithm has a time complexity of O(n) (e.g. - loops), it means that the time it takes to solve a problem increases linearly with the size of the input. If an algorithm has a time complexity of O(n^2) (e.g. - nested loops), it means that the time it takes to solve a problem increases quadratically with the size of the

input.

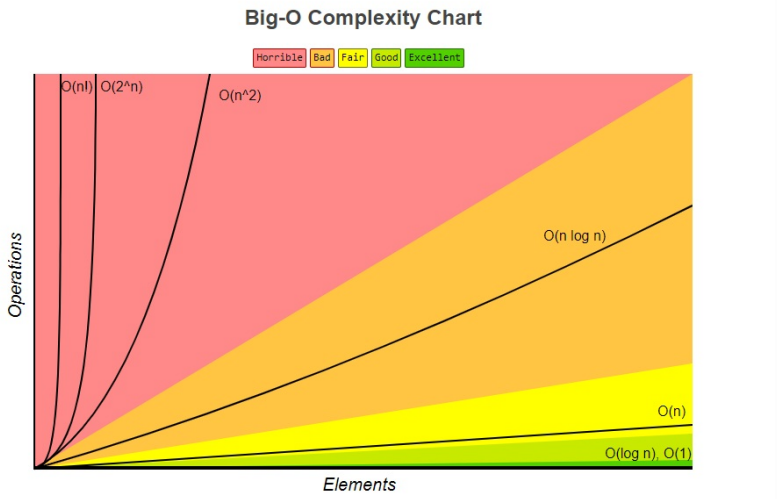


Image from https://www.bigocheatsheet.com

There are some rules of thumb when analyzing space and time complexity:
- Consider the number of iterations in loops.
- Consider the number of variables created, and data structures utilized
- Consider recursive calls
- Identify the number of operations performed
- Drop the constants (e.g. - O(**2**n) or O(**3**n^2))
- Drop the smaller terms (e.g. - O(n^2 + **n**))
- Analyze the worst-case scenario
>
All of these can help analyze the complexity in algorithms and determine the dominant factor affecting their performance. Keep all of this in mind as we progress through the following lessons covering data structures and algorithms.

Overview of data structures we will cover

Here is a list of the custom data structures we will look at in the following two lessons:

Singly linked lists: A data structure in which each node points to the next node, forming a linear sequence of elements that can be easily traversed but can only be accessed sequentially.

Stacks: A data structure in which elements are added and removed from only one end, known as the "top", following the Last-In-First-Out (LIFO) principle.

Binary search trees: A data structure in which each node has at most two child nodes, with the left child node having a smaller value and the right child node having a larger value, allowing for efficient searching and sorting of elements.

The Concept of Pointers

Pointers are an advanced concept in programming languages, and while JavaScript does not have pointers in the traditional sense because it doesn't give us the ability to manipulate memory directly, it does have similar concepts that achieve similar results.

In JavaScript, we have something called references. References are like pointers because they point to a location in memory where an object is stored. For example, when we assign a variable to an object, we are creating a reference to that object in memory.

Pointers are an essential concept when it comes to data structures like linked lists, stacks, queues, trees, and graphs. In these data structures, pointers are used to establish relationships between elements or nodes, enabling efficient access, traversal, and modification of data.

Linked lists are a great example. In a linked list, each node contains a data element and a pointer to the next node in the list. By using these pointers, we can traverse the linked list by starting at the head of the list and following the pointers to access each element in turn.

It's worth noting that some programming languages, like C and C++, do provide direct access to memory using pointers; however, when not managed correctly this access can lead to issues like memory leaks and segmentation faults. Higher-level (meaning more abstracted and not as "close to the metal") languages like JavaScript provide safer ways of using pointers, making it easier to work with complex data structures without worrying about low-level memory management.

Knowledge Check

Conclusion and Takeaway

> Data structures play a fundamental role in computer science, and understanding them is crucial for any developer. In this lesson, we reviewed Big O Notation and its importance in analyzing the efficiency of algorithms. We also covered a high-level overview of several custom data structures which we will soon look at in more depth, namely linked lists, stacks, queues, binary search trees, hash tables, and graphs. Finally, we discussed pointers and their use in establishing relationships between elements or nodes in data structures. In the next lesson, we will do a deep dive into the various data structures mentioned and see how we can implement them in JavaScript.

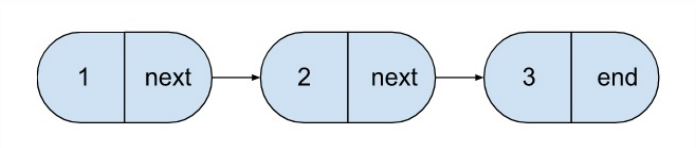
Attribution

Big-O Algorithm Complexity Cheat Sheet (Know Thy Complexities!) @ericdrowell. (n.d.). https://www.bigocheatsheet.com/

Data Structures: Deep Dive - Part 1

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - Understand the concepts and implementations of singly linked lists and stacks
> - Learn how to create and manipulate these data structures using JavaScript
>
> ### Introduction
>
> Singly linked lists and stacks are two fundamental data structures in computer science, and are the first we will go over. They provide a way to organize and store data so that it can be accessed and manipulated quickly and easily. In this lesson, we will cover the concepts and implementations of singly linked lists and stacks, as well as how these data structures work and their practical applications.

Singly Linked Lists Part 1

> ### Theory
>
> A singly linked list is a way of storing a collection of data items in memory, where each item is stored in a node object that contains a pointer to the next node in the list. The first node in the list is called the “head” of the list, while the last node is called the “tail”. Each node in the list has a “data” or “value” property that stores the actual data or value for that node.
>
> To understand how a linked list works, imagine you have a chain of beads, where each bead has a hole in it that can be threaded onto a string. The string represents the “pointers” in the linked list, connecting each bead to the next one. The first bead in the chain represents the head of the list, and the last bead represents the tail. Each bead represents a node in the linked list, and the hole in the bead represents the data stored in that node.
>
> Here’s a graphical representation of what a singly linked list with three nodes would look like, each node containing a single number:
>
> 
>
>
> linked list
>
> In this example, the head of the list points to the first node, which contains the number “1”. The “next” pointer in that node points to the second node, which contains the number “2”. The “next” pointer in the second node points to the third node, which contains the number “3”. Finally, the “next” pointer in the third node is null, indicating the end of the list and the tail of the list.

The advantage of using a linked list over other data structures, such as arrays, is that linked lists can be more efficient for certain operations, such as adding or removing items from the middle of the list. With an array, if you want to add or remove an item from the middle of the array, you have to shift all the items after that position up or down by one index, which can be slow and inefficient for large arrays. With a linked list, you only have to update the pointers in a few nodes to insert or remove a node from the middle of the list, which can be much faster.

For example, imagine you're writing a program that needs to keep track of a list of tasks to be done. You could use a linked list to represent this list, where each node in the list represents a single task, and the "next" pointer points to the next task in the list. This would allow you to add new tasks to the beginning or end of the list, or remove tasks from the middle of the list, without having to shift all the tasks up or down in the list like you would with an array.
>
Implementation
In the top left panel, in addition to our main JavaScript file, there is a completed example of a singly linked list. You can use it as a reference throughout this section.

Let's now code a singly linked list together piece by piece, by going through the following steps:
1. We first define a **SLLNode** class to represent each node in the linked list. Each node has a **value** property to store the data value for that node, and a **next** property that points to the next node in the list (initially null, since the node is not yet connected to any other nodes).

```
class SLLNode {
  constructor(value) {
    this.value = value
    this.next = null
  }
}
```

2. We then define a **LinkedList** class to represent the entire linked list. The **LinkedList** class has a **head** property that points to the first node in the list (initially null, since the list is empty), a **tail** property that points to the last node in the list (initially the same as **head**), and a **length** property that keeps track of the number of nodes in the list (initially 1, since we create the first node in the constructor).

```
class LinkedList {
  constructor(value) {
    const newNode = new SLLNode(value)
    this.head = newNode
    this.tail = this.head
    this.length = 1
  }

}
```

Singly Linked Lists Part 2

> **NOTE:** All of the code on this page should be written below the constructor method but still inside the curly brackets of our **LinkedList** class!

3. Next, we define a method **printList** that prints the contents of the list in order. First, if the length of the list is 0, we console log 'List is empty' and return. Otherwise, the method starts at the **head** node and traverses the list using the **next** pointers until it reaches the end of the list (i.e., the **next** property of the current node is null), printing each node's **value** property to the console.

```
printList() {
  if (this.length === 0) {
    console.log('List is empty')
    return
  }
  let temp = this.head
  while (temp !== null) {
    console.log(temp.value)
    temp = temp.next
  }
}
```

4. We define a method **getHeadNode** that prints the value of the **head** node to the console.

```
getHeadNode() {
  if (this.head === null) {
    console.log("Head: null")
  } else {
    console.log("Head: " + this.head.value)
  }
}
```

5. We define a method **getTailNode** that prints the value of the **tail** node to the console.

```
getTailNode() {
  if (this.tail === null) {
    console.log("Tail: null")
  } else {
    console.log("Tail: " + this.tail.value)
  }
}
```

6. We define a method **getLength** that prints the value of the **length** property to the console.

```
getLength() {
  console.log("Length: " + this.length)
}
```

7. We define a method **makeEmpty** that sets the **head**, **tail**, and **length** properties to null, null, and 0, respectively, effectively emptying the list.

```
makeEmpty() {
  this.head = null
  this.tail = null
  this.length = 0

  return this
}
```

8. We define a method **addNode** that adds a new node to the end of the list. The method takes a **value** parameter, creates a new **SLLNode** object with the given value, and then adds it to the end of the list by updating the **next** property of the current **tail** node to point to the new node, and updating **tail** to point to the new node. Finally, the **length** property is incremented to reflect the new node, and a message confirming the newly added node is returned.

```
addNode(value) {
  const newNode = new SLLNode(value)
  if (!this.head) {
    this.head = newNode
    this.tail = newNode
  } else {
    this.tail.next = newNode
    this.tail = newNode
  }
  this.length++
  return `New node added: ${newNode.value}`
}
```

Singly Linked Lists Part 3

> **NOTE:** All of the code on this page should be written below the constructor method but still inside the curly brackets of our **LinkedList** class!

9. We define a method **removeNode** that removes the last node from the list. The method starts at the **head** node and traverses the list using the **next** pointers until it reaches the second-to-last node, which becomes the new **tail** node. The method then sets the **next** property of the new **tail** node to null and returns the original **tail** node. If the list is empty, the method returns undefined.

```
removeNode() {
  if (this.length === 0) return null
  let temp = this.head
  let prev = this.head
  while (temp.next) {
    prev = temp
    temp = temp.next
  }
  this.tail = prev
  this.tail.next = null
  this.length--
  if (this.length === 0) {
    this.head = null
    this.tail = null
  }
  return `Node removed from end: ${temp.value}`
}
```

10. We define a method **insertAtBeginning** that adds a new node to the beginning of the list. The method takes a **value** parameter, creates a new **SLLNode** object with the given value, and then adds it to the beginning of the list by updating the **next** property of the new node to point to the current **head** node, and updating **head** to point to the new node. Finally, the **length** property is incremented to reflect the new node, and a message confirming the inserted node is returned.

```
insertAtBeginning(value) {
  const newNode = new SLLNode(value)
  if (!this.head) {
    this.head = newNode
    this.tail = newNode
  } else {
    newNode.next = this.head
    this.head = newNode
  }
  this.length++
  return `New node inserted at beginning: ${newNode.value}`
}
```

11. We define a method **removeFromBeginning** that removes the first node from the list. The method updates **head** to point to the second node in the list (i.e., the current **head.next** node), and then returns the original **head** node. If the list is empty, the method returns undefined.

```
removeFromBeginning() {
  if (this.length === 0) return null
  let temp = this.head
  this.head = this.head.next
  this.length--
  if (this.length === 0) {
    this.tail = null
  }
  temp.next = null
  return `Node removed from beginning: ${temp.value}`
}
```

12. We define a method **getNode** that returns the node at a given position/index in the list. The method takes an **index** parameter and traverses the list using a **for** loop, starting at the **head** node and following the **next** pointers until it reaches the node at the given index. If the index is out of bounds (i.e., less than 0 or greater than or equal to **length**), the method returns undefined.

```
getNode(index) {
  if (index < 0 || index >= this.length) return null
  let temp = this.head
  for (let i = 0; i < index; i++) {
    temp = temp.next
  }
  return temp
}
```

13. We define a method **setNode** that updates the **value** property for the node at a given position/index in the list. The method takes **index** and **value** parameters, and uses the **getNode** method to retrieve the node at the given index. If the node exists, the method updates its **value** property to the given value and returns message confirming the updated node. Otherwise, it returns a message stating there is no node at the specified index.

```
setNode(index, value) {
  let temp = this.getNode(index)
  if (temp) {
    temp.value = value
    return `New node value at index ${index}: ${temp.value}`
  }
  return 'No node at specified index'
}
```

Singly Linked Lists Part 4

> **NOTE:** All of the code on this page should be written below the constructor method but still inside the curly brackets of our **LinkedList** class!

14. We define a method **insertAtIndex** that inserts a new node at a given position/index in the list. The method takes **index** and **value** parameters, and uses the **getNode** method to retrieve the node before the given index. It then creates a new **SLLNode** object with the given value and updates its **next** property to point to the node after the given index. Finally, the method updates the **next** property of the node before the given index to point to the new node, and increments the **length** property to reflect the new node. If the index is less than 0 or greater than **length**, the method returns a message stating there is no node at the specified index.

```
insertAtIndex(index, value) {
   if (index < 0 || index > this.length) return 'No node at
       specified index'

   if (index === this.length) {
     this.addNode(value)
     return `Node inserted at index ${index}: ${value}`
   }
   if (index === 0) return this.insertAtBeginning(value)
   const newNode = new SLLNode(value)
   const temp = this.getNode(index - 1)
   newNode.next = temp.next
   temp.next = newNode
   this.length++
   return `Node inserted at index ${index}: ${temp.next.value}`
 }
```

15. We define a method **removeAtIndex** that removes a node from a given position/index in the list. The method takes an **index** parameter and uses the **get** method to retrieve the node before the given index. It then updates the **next** property of the node before the given index to point to the node after the given index, effectively removing the node at the given index from the list. Finally, the method decrements the **length** property to reflect the removed node, and returns a message confirming the removed node. If the index is less than 0 or greater than or equal to **length**, the method returns a message stating there is no node at the specified index.

```
removeAtIndex(index) {
   if (index < 0 || index >= this.length) return 'No node at
       specified index'

   if (index === 0) return this.removeFromBeginning()
   if (index === this.length - 1) return this.removeNode()
   const before = this.getNode(index - 1)
   const temp = before.next
   before.next = temp.next
   temp.next = null
   this.length--
   return `Node removed at index ${index}: ${temp.value}`
 }
```

16. We define a method **reverse** that reverses the order of the nodes in the list. The method starts by swapping the **head** and **tail** properties of the list, and then uses a **for** loop to traverse the list from the **head** node to the **tail** node, updating the **next** property of each node to point to the previous node instead of the next node. Finally, the **head.next** property is set to null to complete the reversal.

```
reverse() {
    let temp = this.head
    this.head = this.tail
    this.tail = temp
    let next = temp.next
    let prev = null
    for (let i = 0; i < this.length; i++) {
      next = temp.next
      temp.next = prev
      prev = temp
      temp = next
    }
 }
```

Singly Linked Lists Part 5

Now that we've finished implementing our singly linked list, let's add the following to test our code:

> **NOTE:** The code below should be **outside** of the LinkedList class!

```
// Create new instance of the LinkedList
const list = new LinkedList(1)

console.log('SINGLY LINKED LIST OUTPUT:\n\n')

// See the linked list in the console
const sllClone1 = _.cloneDeep(list)
console.log('The initial linked list object: ')
console.log(JSON.stringify(sllClone1, null, '\t'))

// Test the addNode method
console.log('\n')
console.log(list.addNode(2))
console.log('\n')
console.log(list.addNode(3))
console.log('\n')
console.log(list.addNode(4))
console.log('\nList after three nodes added: ')
list.printList() // 1 2 3 4
list.getLength() // Length: 4

// Test the getHeadNode method
console.log('\n')
list.getHeadNode() // Head: 1

// Test the getTailNode method
console.log('\n')
list.getTailNode() // Tail: 4

// Test the removeNode method
console.log('\n')
list.removeNode()

console.log('\nList after last node removed: ')
list.printList() // 1 2 3
list.getLength() // Length: 3
```

```
// Test the insertAtBeginning method
console.log('\n')
list.insertAtBeginning(0)
console.log('\nList after node added to beginning: ')
list.printList() // 0 1 2 3
list.getLength() // Length: 4

// Test the removeFromBeginning method
console.log('\n')
console.log(list.removeFromBeginning())
console.log('\nList after node removed from beginning: ')
list.printList() // 1 2 3
list.getLength() // Length: 3

// Test the getNode method
console.log('\nNode at index 1: ')
console.log(JSON.stringify(list.getNode(1), null, '\t')) //
        SLLNode {value: 2, next: SLLNode}
console.log('\nNode at index 3: ')
console.log(JSON.stringify(list.getNode(3), null, '\t'))//
        undefined

// Test the setNode method
console.log('\n')
console.log(list.setNode(2, 5)) // 'New node value at index 2:
        5'
console.log('\nList after node at index 2 changed: ')
list.printList() // 1 2 5

console.log('\n')
console.log(list.setNode(3, 6)) // 'No node at specified index'

// Test the insertAtIndex method
console.log('\n')
console.log(list.insertAtIndex(2, 4))
console.log('\nList after node inserted at index 2: ')
list.printList() // 1 2 4 5
list.getLength() // Length: 4
console.log('\n')
console.log(list.insertAtIndex(0, 0))
list.printList() // 0 1 2 4 5
list.getLength() // Length: 5
console.log('\n')
console.log(list.insertAtIndex(5, 6))
console.log('\nList after node inserted at index 5: ')
list.printList() // 0 1 2 4 5 6
list.getLength() // Length: 6

// Test the removeAtIndex method
```

```
console.log('\n')
console.log(list.removeAtIndex(3)) // Node removed at index 3: 4
console.log('\nList after node removed at index 3: ')
list.printList() // 0 1 2 5 6
list.getLength() // Length: 5
console.log('\n')
console.log(list.removeAtIndex(0)) // Node removed at index 0: 0
console.log('\nList after node removed at index 0: ')
list.printList() // 1 2 5 6
list.getLength() // Length: 4
console.log('\n')
console.log(list.removeAtIndex(3)) // Node removed at index 3: 6
console.log('\nList after node removed at index 3: ')
list.printList() // 1 2 5
list.getLength() // Length: 3

// Test the reverse method
console.log('\n')
list.reverse()
console.log('\nList after being reversed: ')
list.printList() // 5 2 1
console.log('\n')
list.getHeadNode() // Head: 5
console.log('\n')
list.getTailNode() // Tail: 1
```

Big O and Use Case Summary

- **Time complexity**: O(n) for searching, O(1) for insertion/deletion at the beginning, and O(n) for insertion/deletion at the end.
- **Space complexity**: O(n).
- **Best suited scenario**: When the primary operation is to insert or delete elements at the beginning, and frequent searching is not required. For example, when implementing a stack or a queue.

Stacks Part 1

Theory

A stack is a data structure in JavaScript that works like a stack of plates. Imagine you have a pile of plates, and you can only add or remove plates from the top of the pile. You can't add or remove plates from the middle or the bottom of the pile. This is similar to how a stack works.

In a stack data structure, you can add elements to the top of the stack, and you can remove elements from the top of the stack. The first element you add to the stack is the last element you can remove from the stack. This is called "last in, first out" or LIFO for short.

One example of a real-world scenario where using a stack would be advantageous is in a web browser's history. When you visit a website, it gets added to your browsing history. If you click the "back" button in the browser, it will remove the most recent website from the history and take you back to the previous website. This is a perfect example of a stack data structure, where the last website you visited is the first one that gets removed when you click "back".

Another example is in the implementation of the call stack in JavaScript. When you call a function in JavaScript, it gets added to the call stack. If that function calls another function, that function gets added to the top of the stack, and so on. When a function returns, it gets removed from the top of the stack. This is a way for JavaScript to keep track of which function is currently executing, and it's another example of a stack data structure.

Stacks are advantageous over other data structures in situations where the order of operations is important, and where you only need to add or remove elements from the top of the pile. For example, if you're writing a program that needs to keep track of a history of actions, a stack would be a good choice. You can add each new action to the top of the stack, and remove the most recent action from the top of the stack when you need to undo the last action.

Implementation

In the top left panel, in addition to our main JavaScript file, there is a completed example of a stack. You can use it as a reference throughout this section.

Let's code this together piece by piece, by going through the following steps:
>
1. We start by defining a **StackNode** class to represent each node of the linked list. This class has a **value** property to store the value of the node, and a **next** property to store the reference to the next node.

```
class StackNode {
  constructor(value) {
    this.value = value
    this.next = null
  }
}
```

2. We define a **Stack** class that uses a linked list to implement a stack data structure. This class has a **top** property to store the reference to the top node of the stack, and a **length** property to store the length of the stack. The constructor function initializes the stack with either a **null** top and 0 length if no value is provided, or a new **StackNode** with the provided value as the top and a **1** length.

```
class Stack {
  constructor(value) {
    if (value) {
      const newNode = new StackNode(value)
      this.top = newNode
      this.length = 1
    } else {
      this.top = null
      this.length = 0
    }
  }

}
```

3. We define the **push** method that takes a value and creates a new **StackNode** with that value. If the stack is empty, the new node is set as the top node. Otherwise, the new node is added to the top of the stack by setting its next property to the current top node, and setting the top node to the new node. The length of the stack is incremented, and the updated stack is returned.

```
push(value) {
    const newNode = new StackNode(value)
    if (this.length === 0) {
      this.top = newNode
    } else {
      newNode.next = this.top
      this.top = newNode
    }
    this.length++

    return this
  }
```

Stacks Part 2

4. We define the **pop** method that removes the top node from the stack. If the stack is empty, **undefined** is returned. Otherwise, the current top node is stored, the top node is set to the next node, and the next property of the removed node is set to **null**. The length of the stack is decremented, and the removed node is returned.

```
pop() {
    if (this.length === 0) {
      return undefined
    }
    let current = this.top
    this.top = current.next
    current.next = null


    this.length--
    return current
  }
```

5. We define the **seeTopNode** method that returns the top node of the stack. If the stack is empty, “Stack is empty” is returned. Otherwise, the top node is returned.

```
seeTopNode() {
    if (this.length === 0) {
      return "Stack is empty"
    } else {
      return this.top
    }
  }
```

6. We define the **isEmpty** method that returns **true** if the stack is empty and **false** otherwise by checking the length of the stack.

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

7. We define the **size** method that returns the length of the stack.

```
size() {
    return this.length
  }
```

Stacks Part 3

Now that we've finished implementing our stack, let's add the following to test our code:

```
// Create a new Stack instance with an initial value of 1
const stack = new Stack(1)

console.log('STACK OUTPUT:\n\n')

// See the stack in the console
const stackClone1 = _.cloneDeep(stack)
console.log('The initial stack object: ')
console.log(JSON.stringify(stackClone1, null, '\t'))

// Test the push method by adding some new nodes
stack.push(2)
stack.push(3)
stack.push(4)

const stackClone2 = _.cloneDeep(stack)
console.log('\nAfter pushing three nodes: ')
console.log(JSON.stringify(stackClone2, null, '\t'))

// Test the seeTopNode method to see the current top node of the
        stack
console.log('\nTop node: ')
console.log(JSON.stringify(stackClone2.seeTopNode(), null,
        '\t'))

// Test the size method to get the size of the stack
console.log('\nSize of the stack: ' + stackClone2.size())

// Test the pop method by removing the last node
console.log('\nPop: ')
console.log(JSON.stringify(stack.pop(), null, '\t'))

const stackClone3 = _.cloneDeep(stack)
console.log('\nAfter first pop: ')
console.log(JSON.stringify(stackClone3, null, '\t'))

// Test the isEmpty method to check if the stack is empty
console.log('\nStack empty?: ' + stackClone3.isEmpty())
```

```
// Test the push method again by adding a new node with the
        value 5
stack.push(5)

const stackClone4 = _.cloneDeep(stack)
console.log('\nAfter pushing node with value of 5: ')
console.log(JSON.stringify(stackClone4, null, '\t'))

// Test the pop method again by removing the last node
console.log('\n2nd Pop: ')
console.log(JSON.stringify(stack.pop(), null, '\t'))

const stackClone5 = _.cloneDeep(stack)
console.log('\nAfter 2nd pop: ')
console.log(JSON.stringify(stackClone5, null, '\t'))

// Test the seeTopNode method again to see the current top node
        of the stack
console.log('\nCurrent top node: ')
console.log(JSON.stringify(stack.seeTopNode(), null, '\t'))

// Test the size method again to get the size of the stack
console.log('\nCurrent stack size: ' + stack.size())
```

Big O and Use Case Summary

- **Time complexity**: O(1) for push, pop, and peek operations.
- **Space complexity**: O(n).
- **Best suited scenario**: When last-in-first-out (LIFO) ordering is required.

Conclusion and Takeaway

> Singly linked lists and stacks are essential data structures that every developer should be familiar with. In this lesson, we covered the basics of how these data structures work and how they can be implemented in JavaScript, as well as some of their practical applications. Next up, we will look at binary search trees, as well as the concept of recursion.

Data Structures: Deep Dive - Part 2

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The concepts and implementation of binary search trees in JavaScript
> - Recursion and how to write a recursive function
>
> ### Introduction
>
> In this lesson, we will cover the concepts and implementation of another important data structures – binary search trees. We will explore how this data structure works, and how to implement it in JavaScript. In addition, we will also cover the basics of recursion.

Binary Search Trees Part 1

Theory

A tree is a data structure that represents a hierarchy of data, similar to a family tree. It consists of nodes connected by edges. Each node in a tree can have zero or more child nodes, and one parent node (except for the root node, which has no parent).

There are many types of trees, but In this explanation we'll focus on the binary search tree.

A binary search tree is a type of tree where each node has at most two children, and the left child of a node has a value less than the node's value, while the right child has a value greater than the node's value. This makes searching and sorting the data in the tree very efficient, as we can use the values of the nodes to quickly navigate the tree and find the data we're looking for.

For example, let's say we have a list of names and ages, and we want to store them in a data structure so that we can quickly look up a person's age by their name. We could create a binary search tree where each node contains a name and an age, and the nodes are sorted by name. To find a person's age, we would start at the root node and compare the name we're looking for to the name of the root node. If the name we're looking for is less than the root node's name, we would go to the left child node. If it's greater than the root node's name, we would go to the right child node. We would continue this process until we find the node with the name we're looking for, and then we would return the age value of that node.

Binary search trees have many advantages over other data structures. One advantage is that they have a **small memory footprint**, as they only need to store the values of the nodes and their connections. Another advantage is that they allow for **efficient searching and sorting of data**, making them ideal for situations where we need to quickly find or sort data, such as in databases and search engines.

However, there are also some disadvantages to using binary search trees. One disadvantage is that they can become unbalanced if the data is not added in a balanced way, which can slow down searching and sorting. Another disadvantage is that they can be difficult to implement and maintain, especially if we need to perform complex operations such as inserting or deleting nodes.

Implementation

In the top left panel, in addition to our main JavaScript file, there is a completed example of a tree. You can use it as a reference throughout this section.

Let's code this together piece by piece, by going through the following steps:
>
1. Define the **TreeNode** class with the **constructor** function. In the **TreeNode** constructor function, define the properties **value**, **left**, and **right**. Set **value** to equal the value which is passed in, and set both **left** and **right** to null.

```
class TreeNode {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}
```

2. Define the **BinarySearchTree** class with the **constructor** function. In the **BinarySearchTree** constructor function, define the property **root**, and set it to null.

```
class BinarySearchTree {
  constructor() {
    this.root = null
  }

}
```

3. Define the **insertNode** method on the **BinarySearchTree** class. In the **insertNode** method, create a new **TreeNode** object with the given value. If the tree is empty (i.e., **root** is null), set the new node as the root of the tree and return the tree. Otherwise, traverse the tree from the root to the bottom, comparing the value of the new node with each node it encounters until it finds an appropriate place to insert the new node. If the value already exists in the tree, return a message stating that the tree already contains a node with that value.

```
insertNode(value) {
  const newNode = new TreeNode(value)
  if (this.root === null) {
    this.root = newNode
    return this
  }

  let current = this.root

  while (true) {
    if (newNode.value === current.value) {
      return `Tree already contains node with value of ${value}`
    }
    if (newNode.value < current.value) {
      if (current.left === null) {
        current.left = newNode
        return `New node with value of ${newNode.value} added to
        left of ${current.value}`
      }
      current = current.left
    } else {
      if (current.right === null) {
        current.right = newNode
        return `New node with value of ${newNode.value} added to
        right of ${current.value}`

      }
      current = current.right
    }
  }
}
```

Binary Search Trees Part 2

4. Define the **containsNode** method on the **BinarySearchTree** class. In the **containsNode** method, traverse the tree from the root to the bottom, comparing the given value with the value of each node it encounters until it finds the node with the given value or reaches a leaf node with no children. If the node with the given value is found, return true. Otherwise, return false.

```
containsNode(value) {
    if (this.root === null) {
      return false
    }


    let current = this.root


    while (current) {
      if (value < current.value) {
        current = current.left
      } else if (value > current.value) {
        current = current.right
      } else {
        return true
      }
    }
    return false
  }
```

5. Define the **minimumValue** method on the **BinarySearchTree** class. In the **minimumValue** method, traverse the tree from the root to the bottom, following the left child nodes until it reaches the node with the smallest value. Return the node with the smallest value.

```
 minimumValue(currentNode) {
    while (currentNode.left != null) {
      currentNode = currentNode.left
    }
    return currentNode
  }
```

Binary Search Trees Part 3

Now that we've finished implementing our binary search tree, let's add the following to test our code:

```
// Create a new BinarySearchTree instance
const tree = new BinarySearchTree()

console.log('BINARY SEARCH TREE OUTPUT:\n\n')

// See the binary search tree in the console
const bstClone1 = _.cloneDeep(tree)
console.log('The initial BST object: ')
console.log(JSON.stringify(bstClone1, null, '\t'))

// Insert 15 nodes with unique random values between 10 and 100
const values = []
while (values.length < 15) {
  const value = Math.floor(Math.random() * 91) + 10
  if (!values.includes(value)) {
    values.push(value)
    tree.insertNode(value)
  }
}


const bstClone2 = _.cloneDeep(tree)
console.log("\nTree after inserting 15 unique random nodes:")
console.log(JSON.stringify(bstClone2, null, '\t'))


// Test the containsNode method by checking for some random
        values in the tree
console.log("\nContains 50?: " + tree.containsNode(50))
console.log("\nContains 5?: " + tree.containsNode(5))
console.log("\nContains 105?: " + tree.containsNode(105))

// Test the minimumValue method by finding the minimum value in
        the tree
console.log("\nMinimum value in the tree: ")
console.log(JSON.stringify(tree.minimumValue(tree.root), null,
        '\t'))

// Test inserting a node with a value that already exists in the
        tree
```

```
console.log("\nInserting a node with a value that already
        exists:")
console.log(tree.insertNode(values[5]))
const bstClone3 = _.cloneDeep(tree)
console.log(JSON.stringify(bstClone3, null, '\t'))

// Test inserting a node with a new value
console.log("\nInserting a node with a new value:")

// Randomly generate new unique node value
let newTreeNode
while (!newTreeNode) {
  const value = Math.floor(Math.random() * 91) + 10
  if (!values.includes(value)) {
    newTreeNode = value
  }
}

console.log(tree.insertNode(newTreeNode))

const bstClone4 = _.cloneDeep(tree)
console.log('\nTree after inserting node with new value:')
console.log(JSON.stringify(bstClone4, null, '\t'))
```

Big O and Use Case Summary

- **Time complexity**: O(log n) for searching, insertion, and deletion in a balanced tree. Worst-case time complexity is O(n) for an unbalanced tree.
- **Space complexity**: O(n).
- **Best suited scenario**: When searching, insertion, and deletion operations are required frequently, and maintaining order is important.

A Primer on Recursion Part 1

Theory

A recursive function is a function that calls itself, either directly or indirectly, in order to solve a problem or perform a computation. A recursive function typically has two main components: the **base case** and the **recursive case**.

The **base case** is a condition that is used to terminate the recursion. If the base case is met, the function stops calling itself and returns a result. Without a base case, the function would continue to call itself indefinitely, leading to an infinite loop or a stack overflow error.

The **recursive case** is the part of the function that calls itself, usually with a modified input or state. Each time the function calls itself, it creates a new instance of the function on the call stack, which stores information about the function call, including the arguments passed to the function and the memory address of the next instruction to be executed.

As the function calls itself recursively, more and more instances of the function are added to the call stack. When the base case is finally met, the function starts returning values and popping instances off the call stack until the original call is reached and the final result is returned.

One common use case for recursion is to traverse and manipulate tree-like data structures. For example, you might use recursion to perform a depth-first search on a tree, or to compute the height of a binary tree.

Another common use case is to implement recursive algorithms, such as the famous Fibonacci sequence or a recursive binary search.

In terms of Big O notation, recursive algorithms can sometimes have exponential time complexity, which can make them impractical for large inputs. However, many recursive algorithms can be optimized to reduce their time complexity, and recursion can be a useful tool for writing code that is easy to read and understand.

Implementation

Let's now see a couple of actual examples in code:

> **Example 1: Factorial**
> The factorial of a non-negative integer n, denoted by n!, is the product of all positive integers less than or equal to n. For example, 5! = 5 * 4 * 3 * 2 * 1 = 120.
>
> We can compute the factorial of a number using recursion as follows:

```
function factorial(n) {
  if (n === 0) {
    return 1
  } else {
    return n * factorial(n - 1)
  }
}

console.log('Factorial of 5: ' + factorial(5)); // Output: 120
```

In this code, the **factorial** function takes an integer **n** as input and returns its factorial. The base case of the recursion is when **n** is equal to 0, in which case the function returns 1. Otherwise, the function multiplies **n** by the factorial of **n-1**.

Let's walk through how this works when we call **factorial(5)**:

factorial(5) returns **5 * factorial(4)**
factorial(4) returns **4 * factorial(3)**
factorial(3) returns **3 * factorial(2)**
factorial(2) returns **2 * factorial(1)**
factorial(1) returns **1 * factorial(0)**
factorial(0) returns **1**

So the final result is **5 * 4 * 3 * 2 * 1 = 120.**

A Primer on Recursion Part 2

Example 2: Fibonacci Sequence
The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding numbers. The sequence starts with 0 and 1, and the next number in the sequence is the sum of the two previous numbers. Therefore, the sequence goes 0, 1, 1, 2, 3, 5, 8, 13, 21, and so on. The sequence is named after the Italian mathematician Leonardo Fibonacci, who introduced it to the Western world in his book Liber Abaci, published in 1202. The Fibonacci sequence has many interesting properties and can be found in many natural phenomena, such as the branching patterns of trees and the spiral patterns of shells. It also has many applications in various fields, such as mathematics, computer science, and finance.

We can implement an algorithm that will give us the Fibonacci number corresponding to whichever index we pass in. For example, the function will be defined such that **fibonacci(5)** would return the sixth number in the Fibonacci sequence.

> Let's see the code for this:

```
function fibonacci(n) {
  if (n <= 1) {
    return n;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log('\nNumber at index 5 in the Fibonacci sequence: ' +
        fibonacci(5)) // Output: 5
```

In this code, the `fibonacci()` function takes a single argument `n`, which is the index of the Fibonacci number to calculate. If `n` is less than or equal to 1, the function returns `n` as the result. Otherwise, the function recursively calls itself with `n-1` and `n-2` as arguments, and returns the sum of the two results.
>
Note that using recursion to calculate the Fibonacci sequence can be inefficient for large values of `n`, as it will result in many duplicate function calls. In such cases, it's more efficient to use an iterative approach or memoization to avoid redundant calculations.
>
Big O and Use Case Summary
- **Time complexity**: Varies depending on the recursion depth and the operations performed.

- **Space complexity**: Varies depending on the recursion depth and the memory used by each recursive call.
- **Best suited scenario**: When solving problems that can be naturally expressed in terms of smaller instances of the same problem. For example, when implementing recursive algorithms such as binary search, quicksort, or tree traversal.

Conclusion and Takeaway

> In this lesson, we went over the basics of binary search trees, how they work and how they can be implemented in JavaScript. We also touched on recursion, which is another important concept in programming and is utilized by many of the algorithms we will look at later.

Data Structures: More Data Structures (AEL)

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The concepts and implementations of doubly linked lists, queues, hash tables, and graphs in JavaScript
> - How to create and manipulate these various data structures
>
> ### Introduction
>
> In this Asynchronous Elective Learning lesson, we will cover the concepts and implementations of additional data structures not included in the main lesson – namely, we will cover doubly linked lists, queues, hash tables, and graphs. We will explore how these data structures work, and how to implement them in JavaScript.

Doubly Linked Lists Part 1

> ### Theory
>
> A doubly linked list is a data structure that helps us store and manipulate data in a specific order. The order of the data is determined by the order of the nodes in the list. Each node contains a piece of data, as well as references to the previous node and the next node in the list.
>
> Think of it like a chain - each link in the chain is like a node in the doubly linked list. The links are connected to each other in a specific order, and each link has a connection to the link before it and the link after it. This allows us to easily move through the chain in either direction.
>
> One advantage of a doubly linked list is that it allows us to easily add or remove nodes from both the beginning and end of the list. This is useful in situations where we need to add or remove items from a list frequently, and where the order of the list is important. For example, imagine you're building a video game and you want to keep track of the player's score. A doubly linked list could be used to store the top 10 scores, and it would allow you to easily add or remove scores from the beginning or end of the list.
>
> Another advantage of a doubly linked list is that it allows us to easily traverse the list backwards. This is useful in situations where we need to move backwards through a list frequently. For example, imagine you're building a word processor and you want to allow the user to undo their last action. A doubly linked list could be used to keep track of the user's actions in order, and it would allow you to easily move back through the list to undo previous actions.
>
> ### Implementation
>
> In the top left panel, in addition to our main JavaScript file, there is a completed example of a doubly linked list. You can use it as a reference throughout this section.

Let's now code a doubly linked list together piece by piece, by going through the following steps:
>
1. The `DLLNode` class is a basic class used to create the individual nodes of the doubly linked list. It has three properties: `value`, `prev`, and `next`. The

`value` property holds the value of the node, while `prev` and `next` hold references to the previous and next nodes in the list respectively.
>

```
class DLLNode {
  constructor(value) {
    this.value = value
    this.next = null
    this.prev = null
  }
}
```

2. The `DoublyLinkedList` class is the main class that creates and manipulates the doubly linked list. It has three properties: `head`, `tail`, and `length`. `head` holds a reference to the first node of the list, `tail` holds a reference to the last node of the list, and `length` holds the number of nodes in the list.

```
class DoublyLinkedList {
  constructor(value) {
    const newNode = new DLLNode(value)
    this.head = newNode
    this.tail = newNode
    this.length = 1
  }


}
```

Doubly Linked Lists Part 2

3. We define a method **printList** that prints the contents of the list in order. First, if the length of the list is 0, we console log 'List is empty' and return. Otherwise, the method starts at the **head** node and traverses the list using the **next** pointers until it reaches the end of the list (i.e., the **next** property of the current node is null), printing each node's **value** property to the console.

```
printList() {
  if (this.length === 0) {
    console.log('List is empty')
    return
  }
  let temp = this.head
  while (temp !== null) {
    console.log(temp.value)
    temp = temp.next
  }
}
```

4. We define a method **getHeadNode** that prints the value of the **head** node to the console.

```
getHeadNode() {
  if (this.head === null) {
    console.log("Head: null")
  } else {
    console.log("Head: " + this.head.value)
  }
}
```

5. We define a method **getTailNode** that prints the value of the **tail** node to the console.

```
getTailNode() {
  if (this.tail === null) {
    console.log("Tail: null")
  } else {
    console.log("Tail: " + this.tail.value)
  }
}
```

6. We define a method **getLength** that prints the value of the **length** property to the console.

```
getLength() {
  console.log("Length: " + this.length)
}
```

7. We define a method **makeEmpty** that sets the **head**, **tail**, and **length** properties to null, null, and 0, respectively, effectively emptying the list.

```
makeEmpty() {
  this.head = null
  this.tail = null
  this.length = 0

  return this
}
```

8. The `addNode` method adds a node to the end of the list. It takes in a `value` parameter, creates a new node with the value, and adds it to the tail of the list. If the list is empty, both `head` and `tail` are set to the new node. If the list is not empty, the new node is connected to the previous node and `tail` is updated to point to the new node. Finally, the `length` property of the list is incremented by 1 aand a message confirming the newly added node is returned.

```
addNode(value) {
  const newNode = new DLLNode(value)
  if (this.length === 0) {
    this.head = newNode
    this.tail = newNode
  } else {
    this.tail.next = newNode
    newNode.prev = this.tail
    this.tail = newNode
  }
  this.length++
  return `New node added: ${newNode.value}`
}
```

Doubly Linked Lists Part 3

9. The `removeNode` method removes the last node of the list. It first checks if the list is empty and returns `null` if it is. If the list has only one node, both `head` and `tail` are set to `null`. If the list has more than one node, `tail` is updated to point to the second last node, the connection between the last node and the second last node is removed, and the `length` property of the list is decremented by 1. Finally, a message confirming the removed node is returned.

```
removeNode() {
  if (this.length === 0) return undefined
  let temp = this.tail
  if (this.length === 1) {
    this.head = null
    this.tail = null
  } else {
    this.tail = this.tail.prev
    this.tail.next = null
    temp.prev = null
  }
  this.length--
  return `Node removed from end: ${temp.value}`
}
```

10. The `insertAtBeginning` method adds a node to the beginning of the list. It takes in a `value` parameter, creates a new node with the value, and adds it to the head of the list. If the list is empty, both `head` and `tail` are set to the new node. If the list is not empty, the new node is connected to the next node and `head` is updated to point to the new node. Finally, the `length` property of the list is incremented by 1 and a message confirming the newly inserted node is returned.

```
insertAtBeginning(value) {
  const newNode = new DLLNode(value)
  if (this.length === 0) {
    this.head = newNode
    this.tail = newNode
  } else {
    newNode.next = this.head
    this.head.prev = newNode
    this.head = newNode
  }
  this.length++
 return `Node inserted at beginning: ${newNode.value}`
}
```

11. The `removeFromBeginning` method removes the first node of the list. It first checks if the list is empty and returns `null` if it is. If the list has only one node, both `head` and `tail` are set to `null`. If the list has more than one node, `head` is updated to point to the second node, the connection between the first node and the second node is removed, and the `length` property of the list is decremented by 1. Finally, a message confirming the removed node is returned.

```
removeFromBeginning() {
  if (this.length === 0) return undefined
  let temp = this.head
  if (this.length === 1) {
    this.head = null
    this.tail = null
  } else {
    this.head = this.head.next
    this.head.prev = null
    temp.next = null
  }
  this.length--
 return `Node removed from beginning: ${temp.value}`
}
```

12. The **getNode** method retrieves the node at a specified index of the list. It takes in an index parameter and returns the node at the specified **index**. If the index is less than 0 or greater than or equal to the length of the list, **null** is returned. The method then iterates through the list from the **head** node, using the **next** property of each node to move to the next node in the list until the specified index is reached. Finally, the node at the specified index is returned.

```
getNode(index) {
  if (index < 0 || index >= this.length) return undefined
  let temp = this.head
  if (index < this.length / 2) {
    for (let i = 0; i < index; i++) {
      temp = temp.next
    }
  } else {
    temp = this.tail
    for (let i = this.length - 1; i > index; i--) {
      temp = temp.prev
    }
  }
  return temp
}
```

13. The **setNode** method updates the value of a node at a specified index of the list. It takes in an **index** parameter and a **value** parameter, updates the value of the node at the specified index to the new value, and returns a message confirming the updated node. If the node at the specified index does not exist, a message to that effect is returned.

```
setNode(index, value) {
  let temp = this.getNode(index)
  if (temp) {
    temp.value = value
    return `New node value at index ${index}: ${temp.value}`
  }
  return 'No node at specified index'
}
```

Doubly Linked Lists Part 4

14. The `insertAtIndex` method inserts a node at a given index of the list. It takes in an index parameter and a value parameter, creates a new node with the value, and inserts it at the specified index. If the index is less than 0 or greater than the length of the list, a message is returned stating there is no node at the specified index. If the index is equal to the length of the list, the new node is added to the tail of the list using the **addNode** method. If the index is 0, the new node is added to the head of the list using the **insertAtBeginning** method. Otherwise, the node before the specified index is retrieved using the **getNode** method, the node after the specified index is retrieved by accessing the next property of the node before, and the new node is inserted between them. Finally, the length property of the list is incremented by 1 and a message confirming the newly added node is returned.

```
  insertAtIndex(index, value) {
    if (index < 0 || index > this.length) return 'No node at
specified index'
    if (index === this.length) {
      this.addNode(value)
      return `Node inserted at index ${index}: ${value}`
    }
    if (index === 0) return this.insertAtBeginning(value)

    const newNode = new DLLNode(value)
    const before = this.getNode(index - 1)
    const after = before.next
    before.next = newNode
    newNode.prev = before
    newNode.next = after
    after.prev = newNode
    this.length++
    return `Node inserted at index ${index}: ${newNode.value}`
  }
```

15. The **removeAtIndex** method removes a node at a given index of the list. It takes in an **index** parameter and removes the node at the specified index. If the index is less than 0 or greater than or equal to the length of the list, **undefined** is returned. If the index is 0, the first node is removed using the **removeFromBeginning** method. If the index is equal to the length of the list - 1, the last node is removed using the **removeNode** method. Otherwise, the node to remove is retrieved using

the **getNode** method, the node before the specified index is retrieved using the **prev** property of the node to remove, and the node after the specified index is retrieved using the **next** property of the node to remove. The node to remove is then disconnected from the list and the **length** property of the list is decremented by 1. Finally, a message confirming the removed node is returned.

```
removeAtIndex(index) {
  if (index === 0) return this.removeFromBeginning()
  if (index === this.length - 1) return this.removeNode()
  if (index < 0 || index >= this.length) return undefined

  const temp = this.getNode(index)

  temp.prev.next = temp.next
  temp.next.prev = temp.prev
  temp.next = null
  temp.prev = null

  this.length--
  return `Node removed at index ${index}: ${temp.value}`
}
```

Doubly Linked Lists Part 5

Now that we've finished implementing our doubly linked list, let's add the following to test our code:
>

```
// Create a new DoublyLinkedList instance with an initial value
of 1
const list = new DoublyLinkedList(1)

console.log('DOUBLY LINKED LIST:\n\n')

// See the doubly linked list in the console
const dllClone1 = _.cloneDeep(list)
console.log('The initial doubly linked list object: ')
console.log(dllClone1)

// Test the addNode method by adding some new nodes
console.log('\n')
console.log('addNode() method:')
console.log(list.addNode(2))
console.log(list.addNode(3))
console.log(list.addNode(4))

const dllClone2 = _.cloneDeep(list)
console.log('\nThe doubly linked list object after adding
nodes:')
console.log(dllClone2)

// Test the printList method to see the current state of the
linked list
console.log('\nprintList() output after three nodes added:')
list.printList() // Output: 1 2 3 4

// Test the getHeadNode method to get the head node of the
linked list
console.log('\n')
console.log('getHeadNode() method:')
list.getHeadNode() // Output: Head: 1

// Test the getTailNode method to get the tail node of the
linked list
console.log('\n')
console.log('getTailNode() method:')
```

```
list.getTailNode() // Output: Tail: 4

// Test the getLength method to get the length of the linked
list
console.log('\n')
console.log('getLength() method:')
list.getLength() // Output: Length: 4

// Test the removeNode method by removing the last node
console.log('\n')
console.log('removeNode() method:')
console.log(list.removeNode())

// Test the insertAtBeginning method by adding a new node at the
beginning
console.log('\n')
console.log('insertAtBeginning() method:')
console.log(list.insertAtBeginning(0))

// Test the removeFromBeginning method by removing the first
node
console.log('\n')
console.log('removeFromBeginning() method:')
console.log(list.removeFromBeginning())

// Test the getNode method by getting the node at index 2
const dllClone3 = _.cloneDeep(list)
console.log('\n')
console.log('getNode() method:')
console.log('\nNode at index 2: ')
console.log(dllClone3.getNode(2))
// Output: DLLNode { value: 3, next: null, prev: DLLNode {
value: 2, ... } }

// Test the setNode method by setting the value of the node at
index 2 to 10
console.log('\n')
console.log('setNode() method:')
console.log(list.setNode(2, 10))

// Test the insertAtIndex method by inserting a new node with
the value 5 at index 2
console.log('\n')
console.log('insertAtIndex() method:')
console.log(list.insertAtIndex(2, 5))

// Test the removeAtIndex method by removing the node at index 3
console.log('\n')
```

```
console.log('removeAtIndex() method:')
console.log(list.removeAtIndex(3))

const dllClone4 = _.cloneDeep(list)
console.log('\nList after node removed at index 3:')
dllClone4.printList()

// Test the makeEmpty method by emptying the linked list
console.log('\n')
console.log('List after makeEmpty() method called: ')
console.log(list.makeEmpty())
```

Be sure to refresh the preview window to see the output!

Big O and Use Case Summary

- **Time complexity**: O(n) for searching, insertion, and deletion.
- **Space complexity**: O(n).
- **Best suited scenario**: When searching, insertion, and deletion operations are required frequently.

Queues Part 1

Theory

A queue is a data structure in JavaScript (and other programming languages) that represents a collection of elements in which elements are added at one end, and removed from the other end, in a first-in-first-out (**FIFO**) order. This means that the first element added to the queue will be the first one to be removed.
>
To understand how a queue works, let's use a real-world example. Imagine you're waiting in line at a movie theater. The first person to arrive is the first in line, and they will be the first person to enter the theater when the doors open. The next person to arrive will join the back of the line, and will have to wait for everyone in front of them to enter the theater before they can go in. This is essentially how a queue works - elements are added to the end of the line, and removed from the front of the line.
>
Now, let's consider some real-world scenarios where using a queue data structure would be advantageous over other data structures:
>
- **Printer Queue**: When multiple users need to print documents to a single printer, a queue can be used to manage the print jobs. The documents are added to the end of the queue, and are printed in the order in which they were received. This ensures that everyone's documents are printed fairly, and that no one document jumps the queue.
>
- **Call Center Queue**: When customers call a call center, they are added to a queue until a customer service representative is available to speak with them. This ensures that customers are helped in the order in which they called, and that no one is skipped or overlooked.
>
- **Traffic Queue**: When multiple cars need to cross a bridge or pass through a toll booth, a queue can be used to manage the flow of traffic. Cars are added to the end of the queue and are allowed to cross the bridge or pass through the toll booth in the order in which they arrived. This helps to prevent traffic jams and accidents caused by cars trying to cut in front of each other.
>
Implementation
In the top left panel, in addition to our main JavaScript file, there is a completed example of a queue. You can use it as a reference throughout this section.

Let’s now code a queue together piece by piece, by going through the following steps:
>
1. We define the **QueueNode** class, which represents the nodes of the linked list. The class has a **value** property to store the element, and a **next** property to point to the next node in the list.

```
class QueueNode {
  constructor(value) {
    this.value = value
    this.next = null
  }
}
```

2. We define the **Queue** class, which uses the ES6 class syntax. The class takes a **value** parameter in its constructor, which is used to create the first node of the queue. If no value is passed in, the queue is initialized to an empty state with **this.first**, **this.last**, and **this.length** all set to **null** or **0**.

```
class Queue {
  constructor(value) {
    if (value) {
      const newNode = new QueueNode(value)
      this.first = newNode
      this.last = newNode
      this.length = 1
    } else {
      this.first = null
      this.last = null
      this.length = 0
    }
  }


}
```

3. We define the **enqueue()** method, which adds an element to the end of the queue. It takes a **value** parameter, which is used to create a new node with the **QueueNode** class. If the length of the queue is 0, the new node is both the first and last node in the queue. If the length is greater than 0, the **next** property of the current last node is set to the new node, and the **last** property is updated to point to the new node.

```
enqueue(value) {
    const newNode = new QueueNode(value)
    if (this.length === 0) {
      this.first = newNode
      this.last = newNode
    } else {
      this.last.next = newNode
      this.last = newNode
    }
    this.length++
    return this
  }
```

Queues Part 2

4. We define the **dequeue()** method, which removes the element at the front of the queue. If the length of the queue is 0, the method returns **undefined**. If the length is 1, both the **first** and **last** properties are set to **null**, and the length is decremented. If the length is greater than 1, the **first** property is updated to point to the next node in the list, the **next** property of the current first node is set to **null**, and the length is decremented. The current first node is then returned.

```
dequeue() {
    if (this.length === 0) {
      return undefined
    } else {
      let current = this.first
      if (this.length === 1) {
        this.first = null
        this.last = null
      } else {
        this.first = this.first.next
        current.next = null
      }
      this.length--
      return current
    }
  }
```

5. We define the **front()** method, which returns the element at the front of the queue without modifying the queue. If the queue is empty, the method returns a “No elements in Queue” message.

```
front() {
    if (this.isEmpty()) {
      return "No elements in Queue"
    }
    return this.first.value
  }
```

6. We define the **isEmpty()** method, which returns **true** if the **length** property is **0**, indicating an empty queue, and **false** otherwise.

```
isEmpty() {
    return this.size === 0
  }
```

7. We define the **getSize()** method, which returns the number of elements in the queue, which is stored in the **length** property.

```
getSize() {
    return this.size
  }
```

8. We define the **print()** method, which creates a string representation of the contents of the queue by iterating over the nodes in the list and appending their values to a string. If the queue is empty, a “No elements in Queue” message is returned.

```
print() {
    if (this.isEmpty()) {
      return "No elements in Queue"
    }
    let str = ""
    let current = this.first
    while (current !== null) {
      str += current.value + " "
      current = current.next
    }
    return str
  }
```

Queues Part 3

> Now that we've finished implementing our stack, let's add the following to test our code:

```
// Create a new queue object
const queue = new Queue()


console.log('QUEUE OUTPUT:\n\n')

// See the queue in the console
const queueClone1 = _.cloneDeep(queue)
console.log('The initial queue object: ')
console.log(JSON.stringify(queueClone1, null, '\t'))

// Add some elements to the queue
queue.enqueue(10)
queue.enqueue(20)
queue.enqueue(30)

const queueClone2 = _.cloneDeep(queue)
console.log('\nQueue after adding three nodes: ')
console.log(JSON.stringify(queueClone2, null, '\t'))

// Print the contents of the queue
console.log('\nResult of print() after enqueuing three nodes: '
+ queue.print()) // Output: "10 20 30"

// Remove an element from the queue
console.log('\nDequeue: ')
console.log(JSON.stringify(queue.dequeue(), null, '\t'))

const queueClone3 = _.cloneDeep(queue)
console.log('\nQueue after dequeuing first node: ')
console.log(JSON.stringify(queueClone3, null, '\t'))

// Print the contents of the queue
console.log('\nResult of print() after dequeuing a node: ' +
queue.print()) // Output: "20 30"
```

Be sure to refresh the preview window to see the output!

Big O and Use Case Summary

- **Time complexity**: O(1) for enqueue, dequeue, and peek operations.
- **Space complexity**: O(n).
- **Best suited scenario**: When first-in-first-out (FIFO) ordering is required. For example, when implementing a job scheduling system.

Hash Tables Part 1

> ### Theory
>
> A hash table is a data structure that allows for efficient insertion, retrieval, and deletion of key-value pairs. It works by using a hash function to map keys to indices in an array of buckets. Each bucket contains a linked list of key-value pairs that have the same hash value.
>
> When inserting a new key-value pair, the hash function is used to generate an index for the key, and the key-value pair is added to the linked list at that index. When retrieving a value for a key, the hash function is used to generate the index for the key, and the linked list at that index is searched for the key. When deleting a key-value pair, the linked list at the index for the key is searched for the key, and the pair is removed if found.
>
> Hash tables are often used when quick lookups of data are needed, such as in a search engine indexing the content of web pages or in a database indexing the location of records. They can also be used in situations where duplicates must be removed from a large dataset or when maintaining a set of unique values.
> ### Implementation
> Let's now code a hash table together piece by piece, by going through the following steps:

1. Define the **HashTable** class.

```
class HashTable {

}
```

2. In the constructor, add a parameter **size** that defaults to 7 if no value is provided. This value determines the size of the hash table. Also, we'll initialize the hash table with a new array of length **size** and assign it to the **dataMap** property.

```
constructor(size = 7) {
    this.dataMap = new Array(size)
    for (let i = 0; i < size; i++) {
      this.dataMap[i] = [];
    }
  }
```

3. Implement the **_hash** method to take a **key** as input and return a hash value. This method uses a simple hashing algorithm that multiplies the ASCII value of each character in the **key** by 23 and adds it to the **hash** variable. The result is divided by the length of the **dataMap** array, and the remainder is used as the hash value. This ensures that the hash value falls within the bounds of the array. The final hash value is returned.

```
_hash(key) {
    let hash = 0
    for (let i = 0; i < key.length; i++) {
      hash = (hash + key.charCodeAt(i) * 23) %
this.dataMap.length
    }
    return hash
  }
```

Hash Tables Part 2

4. Implement the *printTable* method that iterates over each element in the **dataMap** array and logs its index and value to the console.

```
printTable() {
    for (let i = 0; i < this.dataMap.length; i++) {
      console.log(i, ": ", this.dataMap[i])
    }
  }
```

5. Implement the **set** method that takes a **key** and **value** as input and stores them in the hash table. The method first calculates the hash value for the **key** using the **_hash** method. If the array element at the hash value is undefined, an empty array is created at that index. The **key** and **value** are then added to the array as a nested array. The **set** method returns the **HashTable** instance to allow for chaining.

```
set(key, value) {
    let index = this._hash(key)
    if (!this.dataMap[index]) this.dataMap[index] = []


    this.dataMap[index].push([key, value])
    return this
  }
```

6. Implement the **get** method that takes a **key** as input and returns the value associated with that key in the hash table. The method first calculates the hash value for the **key** using the **_hash** method. If the array element at the hash value is defined, the method iterates over the nested arrays at that index and returns the value associated with the input **key**. If the **key** is not found, the method returns **undefined**.

```
get(key) {
    let index = this._hash(key)
    if (this.dataMap[index]) {
      for (let i = 0; i < this.dataMap[index].length; i++) {
        if (this.dataMap[index][i][0] === key) {
          return this.dataMap[index][i][1]
        }
      }
    }
    return null;
  }
```

7. Implement the **keys** method that returns an array of all the keys in the hash table. The method initializes an empty array **allKeys** and then iterates over each element in the **dataMap** array. If an element is defined, the method iterates over the nested arrays at that index and pushes the first element of each nested array (the key) to the **allKeys** array. The **allKeys** array is returned.

```
keys() {
    let allKeys = []
    for (let i = 0; i < this.dataMap.length; i++) {
      if (this.dataMap[i]) {
        for (let j = 0; j < this.dataMap[i].length; j++) {
          allKeys.push(this.dataMap[i][j][0])
        }
      }
    }
    return allKeys
  }
```

Hash Tables Part 3

Test It Out

Let's create a new hash table and add some key-value pairs to it. We'll then use the get() method to retrieve the values associated with the keys.

```
// Create a new HashTable instance with a default size of 7
const table = new HashTable()
console.log('HASH TABLE OUTPUT:\n\n')

// See the initial hash table in the console
const tableClone1 = _.cloneDeep(table)
console.log('The initial hash table object: ')
console.log(JSON.stringify(tableClone1, null, '\t'))

// Test the set method by adding some key-value pairs
table.set("apple", 2)
table.set("banana", 4)
table.set("orange", 6)
console.log("\nTable after adding key-value pairs:")

const tableClone2 = _.cloneDeep(table)
console.log(JSON.stringify(tableClone2, null, '\t'))

// Test the get method by getting the value for a key
console.log("\nValue for 'banana': " +
tableClone2.get("banana"))

// Test the keys method by getting all the keys in the table
console.log("\nAll keys in the table: ")
console.log(JSON.stringify(tableClone2.keys(), null, '\t'))

// Test the printTable method to print the table in the console
console.log("\nprintTable result:")
JSON.stringify(table.printTable(), null, '\t')

// Test the set method again by updating the value for an
existing key
table.set("apple", 3)
console.log("\nTable after updating the value for 'apple':")

const tableClone3 = _.cloneDeep(table)
console.log(JSON.stringify(tableClone3, null, '\t'))
```

```
// Test the set method again by adding a new key-value pair
table.set("pear", 5)
console.log("\nTable after adding a new key-value pair:")

const tableClone4 = _.cloneDeep(table)
console.log(JSON.stringify(tableClone4, null, '\t'))

// Test the get method again by getting the value for a key that
doesn't exist in the table
console.log("\nReturn of get() for value 'grape' (doesn't exist
in table): " + table.get("grape"))
```

This should output the values associated with the keys ‘apple’, ‘banana’, and ‘cherry’. You can also try adding more key-value pairs and testing the get() method to make sure that the hash table is working correctly.

Be sure to refresh the preview window to see the output!

Big O and Use Case Summary

- **Time complexity**: O(1) for insertion, deletion, and searching on average. The worst-case time complexity is O(n).
- **Space complexity**: O(n).
- **Best-suited scenario**: When fast insertion, deletion, and searching operations are required, and order is not important. For example, when implementing a cache or a frequency counter.

Graphs Part 1

Theory

In computer science, a graph is a data structure that consists of a set of vertices (also called nodes) and a set of edges that connect those vertices. The edges can be directed or undirected, and they represent relationships between the vertices. Graphs can be used to model a wide variety of real-world systems, such as social networks, transportation networks, and computer networks.

In JavaScript, a graph can be implemented using an adjacency list, which is an object that maps each vertex to an array of its adjacent vertices. The Graph class above uses an adjacency list to store its vertices and edges. When you call **addVertex**, a new vertex is added to the adjacency list with an empty array for its adjacent vertices. When you call **addEdge**, two vertices are added to each other's adjacency lists. When you call **removeEdge**, two vertices are removed from each other's adjacency lists. When you call **removeVertex**, a vertex and all its adjacent vertices are removed from the adjacency list.

Graphs can be either weighted or unweighted. Weighted graphs have a value associated with each edge, representing some kind of cost or distance between the vertices. Unweighted graphs, like the one implemented in the Graph class above, simply indicate the presence or absence of an edge.

Graphs can also be either directed or undirected. Directed graphs have edges that only go in one direction, while undirected graphs have edges that go in both directions. The Graph class above implements an undirected graph.

In some cases, it may be more efficient to represent a graph using a matrix rather than an adjacency list. This is especially true for dense graphs with many edges, as matrix operations can be faster than array operations. However, for most practical purposes, an adjacency list is a good choice for implementing a graph in JavaScript.

There are many algorithms and techniques for working with graphs, including graph traversal (e.g. depth-first search, breadth-first search), shortest path algorithms (e.g. Dijkstra's algorithm), and minimum spanning tree algorithms (e.g. Kruskal's algorithm). These algorithms can be used to solve a wide variety of problems in fields like computer science, mathematics, and engineering.

> Real-world scenarios where using a graph would be advantageous over other data structures:
> 1. **Social networks**: In a social network like Facebook or Twitter, a graph can be used to represent users as vertices and relationships like friendships or follows as edges. This allows for easy traversal of the network to find connections between users, recommend new friends, and suggest content based on interests.
>
> 2. **Transportation networks**: In a transportation network like a subway or bus system, a graph can be used to represent stations as vertices and routes as edges. This allows for efficient route planning, finding alternative routes in case of delays or closures, and optimizing the placement of new stations or routes.
>
> 3. **Computer networks**: In a computer network, a graph can be used to represent devices as vertices and connections as edges. This allows for easy visualization of the network topology, detecting and resolving network issues, and optimizing the network for speed and reliability.
>
> In all of these scenarios, using a graph allows for an efficient and intuitive representation of complex systems and relationships between entities.
>
> ### Implementation
>
> Let's code this together piece by piece, by going through the following steps:
>
> 1. The **Graph** class is defined with a constructor that initializes an empty **adjacencyList** object. This object will store the vertices and their adjacent vertices in the graph.

```
class Graph {
  constructor() {
    this.adjacencyList = {}
  }


}
```

2. The **addVertex** method is defined, which takes a vertex as an argument and adds it to the **adjacencyList** if it doesn't already exist.

```
// Adds a vertex to the graph
  addVertex(vertex) {
    if (!this.adjacencyList[vertex]) this.adjacencyList[vertex]
= []
  }
```

Graphs Part 2

3. The **addEdge** method is defined, which takes two vertices as arguments and adds an undirected edge between them. It first checks that both vertices exist in the **adjacencyList**, and then adds each vertex to the other's adjacency list.

```
// Adds an undirected edge between two vertices
  addEdge(vertex1, vertex2) {
    // Make sure both vertices exist in the adjacency list
    if (!this.adjacencyList[vertex1] ||
!this.adjacencyList[vertex2]) {
      return
    }


    // Add vertex2 to the adjacency list of vertex1 and vice
versa
    this.adjacencyList[vertex1].push(vertex2)
    this.adjacencyList[vertex2].push(vertex1)
  }
```

4. The **removeEdge** method is defined, which takes two vertices as arguments and removes the undirected edge between them. It first checks that both vertices exist in the **adjacencyList**, and then removes each vertex from the other's adjacency list using the **filter** method.

```
// Removes an undirected edge between two vertices
  removeEdge(vertex1, vertex2) {
    // Make sure both vertices exist in the adjacency list
    if (!this.adjacencyList[vertex1] ||
!this.adjacencyList[vertex2]) {
      return
    }


    // Filter out vertex2 from the adjacency list of vertex1 and
vice versa
    this.adjacencyList[vertex1] =
this.adjacencyList[vertex1].filter(
      (v) => v !== vertex2
    )
    this.adjacencyList[vertex2] =
this.adjacencyList[vertex2].filter(
      (v) => v !== vertex1
    )
  }
```

5. The **removeVertex** method is defined, which takes a vertex as an argument and removes it and all its edges from the graph. It first checks that the vertex exists in the **adjacencyList**, and then removes all edges to the vertex using the **removeEdge** method. Finally, it removes the vertex itself from the **adjacencyList** using the **delete** keyword.

```
// Removes a vertex and all its edges from the graph
  removeVertex(vertex) {
    // Make sure the vertex exists in the adjacency list
    if (!this.adjacencyList[vertex]) return


    // Remove all edges to the vertex to be removed
    while (this.adjacencyList[vertex].length) {
      const adjacentVertex = this.adjacencyList[vertex].pop()
      this.removeEdge(vertex, adjacentVertex)
    }


    // Finally, remove the vertex itself
    delete this.adjacencyList[vertex]
  }
```

Graphs Part 3

Test It Out

Now that we've finished implementing our graph, let's add the following to test our code:

```
// Create a new Graph instance
const graph = new Graph()

console.log('GRAPH OUTPUT:\n\n')

// See the initial graph in the console
const graphClone1 = _.cloneDeep(graph)
console.log('The initial graph object: ')
console.log(JSON.stringify(graphClone1, null, '\t'))

// Test the addVertex method by adding some vertices
graph.addVertex("A")
graph.addVertex("B")
graph.addVertex("C")
console.log("\nGraph after adding vertices:")

const graphClone2 = _.cloneDeep(graph)
console.log(JSON.stringify(graphClone2, null, '\t'))

// Test the addEdge method by adding some edges
graph.addEdge("A", "B")
graph.addEdge("B", "C")
console.log("\nGraph after adding edges:")

const graphClone3 = _.cloneDeep(graph)
console.log(JSON.stringify(graphClone3, null, '\t'))

// Test the removeEdge method by removing an edge
graph.removeEdge("A", "B")
console.log("\nGraph after removing edge between A and B:")

const graphClone4 = _.cloneDeep(graph)
console.log(JSON.stringify(graphClone4, null, '\t'))

// Test the removeVertex method by removing a vertex
graph.removeVertex("B")
console.log("\nGraph after removing vertex B:")

const graphClone5 = _.cloneDeep(graph)
console.log(JSON.stringify(graphClone5, null, '\t'))
```

Big O and Use Case Summary

- **Time complexity**: O(V+E) for searching, where V is the number of vertices and E is the number of edges.
- **Space complexity**: O(V+E).

- **Best-suited scenario**: When modeling relationships between objects or entities, and performing traversal algorithms such as BFS or DFS.

NOTE: The **O(V+E)** complexity is a special case particular to graphs, and it doesn't easily translate to any of the other more common complexities simply because it depends on the context – in some contexts it could be O(n), in other contexts it could be O(n^2).

Conclusion and Takeaway

> In this AEL lesson, we covered more of the fundamental data structures: doubly linked lists, queues, hash tables, and graphs. We went over the basics of how these data structures work and how they can be implemented in JavaScript.

Practice Time - Exercises

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!
>
> 1. Implement a singly linked list in JavaScript with the following methods: insertAtHead(value), insertAtTail(value), removeFromHead(), removeFromTail(), search(value), and printList().
> 2. Implement a doubly linked list in JavaScript with the following methods: insertAtHead(value), insertAtTail(value), removeFromHead(), removeFromTail(), search(value), and printList().
> 3. Implement a stack in JavaScript with the following methods: push(value), pop(), peek(), isEmpty(), and printStack().
> 4. Implement a queue in JavaScript with the following methods: enqueue(value), dequeue(), peek(), isEmpty(), and printQueue().
> 5. Write a function in JavaScript that checks if a given string is a palindrome using a stack and a queue.

Practice Time - Coding Exercises

> Work alone or with a partner to answer the following practice questions. Try to come up with solutions first, but If you get stuck you can always refer back to the material in this lesson or by asking an instructor or TA for help. Happy coding!
>
> 1. Implement a binary search tree in JavaScript with the following methods: insert(value), search(value), and remove(value).
> 2. Implement a hash table in JavaScript with the following methods: set(key, value), get(key), and remove(key).
> 3. Implement a graph in JavaScript with the following methods: addVertex(vertex), addEdge(vertex1, vertex2), removeVertex(vertex), removeEdge(vertex1, vertex2), and depthFirstSearch(startingNode).
> 4. Write a recursive function in JavaScript that calculates the nth number in the Fibonacci sequence.
>   Write a recursive function in JavaScript that calculates the factorial of a given number.

When you're finished, submit your code.

Introduction to Algorithms

Goals

By the end of this lesson, you should understand:

- What algorithms are and their importance in computing
- The theory and implementation of the linear search and binary search algorithms

Introduction

Algorithms are essential to modern computing, providing a set of instructions that enable computers to perform specific tasks and solve complex problems efficiently. They are the foundation of computer science and are used in every aspect of modern technology, from search engines to social media platforms to self-driving cars. In this lesson, we will explore what algorithms are, their importance in computing, and their use in real-world scenarios. We will also focus on two essential search algorithms in computing: linear search and binary search, discussing their theory and implementation in JavaScript.

What are algorithms?

Firstly, what is an algorithm? An algorithm is a set of instructions that a computer follows to perform a specific task. It's like a recipe for a computer program, outlining the steps needed to solve a particular problem.

Why are algorithms important?

Algorithms are important in computing because they allow us to automate processes and solve complex problems efficiently. In fact, algorithms are the foundation of computer science and are used in every aspect of modern technology, from search engines to social media platforms to self-driving cars.

In JavaScript, algorithms play a crucial role in creating web applications and websites. JavaScript code, no less than code in any other programming language must be executed by the browser, and algorithms help to optimize the performance of the code.

Real World Scenarios

Let's explore some real-world scenarios where algorithms are used. One area where algorithms are particularly important is in search engines. When you enter a search query on Google or any other search engine, the engine uses complex algorithms to analyze millions of web pages and determine which ones are most relevant to your query. These algorithms take into account factors such as keywords, page rank, and user behavior to provide the best possible search results.

Another area where algorithms are crucial is in online shopping. When you add items to your cart on an e-commerce website, the site uses algorithms to calculate the total cost of your purchase, including any discounts or shipping fees. This process involves algorithms that perform calculations and update the display in real-time.

Overview of algorithms we will cover

Here is a list of the custom data structures we will look at in the following lessons:

- **Linear search**: A simple algorithm that sequentially searches for a target value in an array or list.
- **Binary search**: A more efficient algorithm that divides a sorted

array in half until the target value is found, or determined to not be present.
- **Bubble sort**: A sorting algorithm that repeatedly swaps adjacent elements until the list is sorted.
- **Merge sort**: A divide-and-conquer algorithm that recursively divides a list into halves, sorts each half, and merges the two sorted halves back into one list.
- **Breadth first search tree traversal**: A tree traversal algorithm that visits all nodes at a given level before moving on to the next level.
- **Depth first search tree traversal**: A tree traversal algorithm that explores as far as possible along each branch before backtracking.

Basic Search Algorithms: Linear Search

> ### Theory
>
> A linear search algorithm is a simple search algorithm that searches for a given value in a list or array, by iterating over each element in the list sequentially, until the target value is found or the entire list has been searched. This algorithm is also known as a sequential search, and it is used in cases where the list is not sorted or the list is small.
>
> The linear search algorithm has a worst-case time complexity of O(n), where n is the number of elements in the list. This means that the time it takes to search the list increases linearly with the number of elements in the list. However, in the best case scenario, if the target value is found in the first element of the list, the algorithm has a time complexity of O(1), which is constant time.
>
> ### Implementation
>
> Now, let's see how to implement the linear search algorithm in JavaScript. We will start by defining a function that takes an array and a target value as arguments and returns the index of the target value in the array.

```
function linearSearch(arr, target) {
  const lastIndex = arr.length - 1
  const lastElement = arr[lastIndex]
  arr[lastIndex] = target
  let i = 0
  while (arr[i] !== target) {
    i++
  }
  arr[lastIndex] = lastElement;
  if (i < lastIndex || arr[lastIndex] === target) {
    return i
  }
  return -1
}
```

The above code defines a function named **linearSearch** that takes an array named **arr** and a target value named **target** as arguments. The function iterates over each element in the array using a for loop and checks if the current element is equal to the target value using an if statement. If the target value is found, the function returns the index of the target value in the array. If the target value is not found, the function returns -1.
>
Now add the following to test our code:

```
const unsortedArray = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
        23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87,
        17, 8, 72, 70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53,
        66, 25, 56, 12, 4, 69, 27, 3, 83, 78, 55, 97]


const target = 39


console.log('LINEAR SEARCH:\n\n')


console.log('Search array: ' + '[' + unsortedArray + ']')
console.log('Target number: ' + target)


const startTime = performance.now()
const result = linearSearch(unsortedArray, target)
const endTime = performance.now()
const executionTime = endTime - startTime


console.log('Index of target: ' + result)
console.log('Execution time: ' + executionTime + '
        milliseconds')
```

> ### Big O and Use Case Summary
>
> - **Time complexity**: O(n).
> - **Space complexity**: O(1).
> - **Best suited scenario**: When searching for a single element in an unsorted list. For example, searching for a name in a phone book.

Basic Search Algorithms: Binary Search

Theory

A **binary search** is an efficient algorithm used to search for a specific value in a sorted array or list. The algorithm works by repeatedly dividing the search interval in half until the target value is found or the search interval is empty. The time complexity of the binary search algorithm is **O(log n)** which makes it faster than the linear search algorithm. This is because the algorithm reduces the search interval by half with each iteration, resulting in a logarithmic time complexity. The space complexity of the binary search algorithm is **O(1)** because it uses a constant amount of additional space.
>
Binary search algorithm follows the divide and conquer approach, which means that it divides the array into two parts, compares the target value with the middle element of the array and decides whether to search in the left or right half of the array. Here are the steps of the binary search algorithm:
>
- Set the low and high indices of the search interval, low = 0 and high = n-1, where n is the size of the array.
- Calculate the mid index of the search interval, mid = (low + high) / 2.
- Compare the target value with the middle element of the array. If they are equal, return the index of the middle element.
- If the target value is less than the middle element, discard the right half of the search interval and set high = mid - 1.
- If the target value is greater than the middle element, discard the left half of the search interval and set low = mid + 1.
- Repeat steps 2 to 5 until the target value is found or the search interval is empty.
>
Implementation
Here's how you can implement the binary search algorithm in JavaScript:

```
function binarySearch(arr, target) {
  let low = 0
  let high = arr.length - 1

  while (low <= high) {
    let mid = Math.floor((low + high) / 2)
    if (arr[mid] === target) {
      return mid
    } else if (arr[mid] < target) {
      low = mid + 1
    } else {
      high = mid - 1
    }
  }

  return -1
}
```

Let's break down the code step by step:

1. The function takes two arguments, arr (the sorted array to be searched) and target (the value to be searched for).
2. The function initializes the low and high indices of the search interval to 0 and arr.length - 1 respectively.
3. The function enters a while loop that continues as long as the low index is less than or equal to the high index. This ensures that the search interval is not empty.
4. Inside the while loop, the function calculates the mid index of the search interval using the formula (low + high) / 2. Math.floor is used to round down the result to the nearest integer.
5. The function compares the target value with the middle element of the array using the if statement. If they are equal, the function returns the index of the middle element.
6. If the target value is less than the middle element, the right half of the search interval is discarded and the high index is set to mid - 1.
7. If the target value is greater than the middle element, the left half of the search interval is discarded and the low index is set to mid + 1.
8. Steps 4 to 7 are repeated until the target value is found or the search interval is empty.
9. If the target value is not found, the function returns -1.
   >
   Let's test the function with an example:

```
const arr = [1, 3, 5, 7, 9]
const target = 7

console.log('BINARY SEARCH (1ST EXAMPLE):\n\n')

console.log('Search array: ' + '[' + arr + ']')
console.log('Target number: ' + target)

const startTime = performance.now()
const result = binarySearch(arr, target)
const endTime = performance.now()
const executionTime = endTime - startTime
console.log('Index of target: ' + result)
console.log('Execution time: ' + executionTime + '
        milliseconds')
```

In this example, we are searching for the value 7 in the array [1, 3, 5, 7, 9]. The function returns the index of the target value, which is 3.
>

Now, let's try it with a larger array and try to glean a little more information about the performance of our algorithm:

```
const sortedArray = [1, 3, 4, 5, 8, 10, 12, 14, 15, 17, 23, 24,
        25, 26, 27, 31, 35, 36, 37, 39, 41, 43, 48, 50, 52, 53,
        55, 56, 63, 64, 66, 69, 70, 72, 73, 74, 75, 78, 80, 81,
        83, 86, 87, 88, 91, 92, 94, 97, 99, 100]


const newTarget = 39

console.log('\n\n\nBINARY SEARCH (2ND EXAMPLE):\n\n')

console.log('Search array: ' + '[' + sortedArray + ']')
console.log('Target number: ' + newTarget)

const startTime2 = performance.now()
const result2 = binarySearch(sortedArray, newTarget)
const endTime2 = performance.now()
const executionTime2 = endTime2 - startTime2
console.log('Index of target: ' + result2)
console.log('Execution time: ' + executionTime2 + '
        milliseconds')
```

Big O and Use Case Summary

- **Time complexity**: O(log n).
- **Space complexity**: O(1).
- **Best suited scenario**: When searching for an element in a sorted list.

For example, searching for a word in a dictionary.

Knowledge Check

Conclusion and Takeaway

> Algorithms are sets of instructions that allow computers to automate processes and solve complex problems efficiently. They are the foundation of computer science and are used in every aspect of modern technology. In JavaScript, algorithms often play a crucial role in optimizing the performance of web applications and websites. Algorithms are also used in real-world scenarios, such as search engines and online shopping, to perform calculations and provide the best possible results for users. In this first lesson, we covered why algorithms are important, looked at the various algorithms we will explore in the next two lessons, and then kicked things off by going over two basic searching algorithms: linear search and binary search.

Algorithms: Deep Dive - Part 1

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The principles and differences between the bubble sort and merge sort algorithms
> - How to implement each of these sorting algorithms in JavaScript
>
> ### Introduction
>
> Sorting is a fundamental task in computer science, and there are many different algorithms for sorting data. In this lesson, we will cover two popular sorting algorithms: bubble sort and merge sort. We will discuss the theoretical background, implementation, and performance analysis of each algorithm, as well as their suitability for different scenarios.

Bubble Sort Part 1

Theory

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The pass of the list is repeated until the list is sorted. This algorithm is called bubble sort because the smaller elements bubble to the top of the list like bubbles rising to the surface of the water.

Bubble sort works by comparing adjacent elements in an array and swapping them if they are in the wrong order. The algorithm does this repeatedly, moving through the array multiple times until no more swaps are needed. Here's how it works:
1. We start by comparing the first two elements of the array. If the first element is larger than the second element, we swap them.
2. We then move to the next pair of elements and repeat the process until we have compared all the adjacent elements in the array.
3. After the first pass through the array, the largest element will have bubbled up to the end of the array.
4. We then repeat steps 1-3 for the remaining elements in the array, excluding the last element since it is already in the correct position.
5. The algorithm continues this process until no more swaps are needed, indicating that the array is sorted.

Implementation

Here is an implementation of the bubble sort algorithm in JavaScript:

```
// Define the bubbleSort function and pass in an array as a
       parameter
function bubbleSort(arr) {
  // Store the length of the array in a variable called len
  let len = arr.length;


  // Iterate through the array n times, where n is the length of
       the array
  for (let i = 0; i < len; i++) {
    // Iterate through the array up to the last unsorted element
    for (let j = 0; j < len - i - 1; j++) {
      // Compare adjacent elements and swap them if they are in
        the wrong order
      if (arr[j] > arr[j + 1]) {
        let temp = arr[j]
        arr[j] = arr[j + 1]
        arr[j + 1] = temp
      }
    }
  }

  // Return the sorted array
  return arr
}
```

The **bubbleSort** function takes an array as an argument and returns the sorted array. The **len** variable holds the length of the array. The outer loop iterates through the array, while the inner loop compares adjacent elements and swaps them if they are in the wrong order. The if statement checks if the current element is greater than the next element, and if so, swaps them using a temporary variable. The sorted array is then returned.
>
Let's walk through an example of using the bubble sort algorithm to sort the following array: [5, 3, 8, 4, 2]. Add the following code to our JavaScript file:

```
const unsortedArray = [5, 3, 8, 4, 2]

console.log('BUBBLE SORT:\n\n')

console.log('Unsorted array: ' + '[' + unsortedArray + ']')

const startTime = performance.now()
const result = bubbleSort(unsortedArray)
const endTime = performance.now()
const executionTime = endTime - startTime
console.log('Result: ' + '[' + result + ']')
console.log('Execution time: ' + executionTime + '
        milliseconds')
```

1. First pass through the array:
   a. Compare 5 and 3, swap them since 5 is larger than 3. Array now looks like [3, 5, 8, 4, 2].
   b. Compare 5 and 8, no swap needed. Array still looks like [3, 5, 8, 4, 2].
   c. Compare 8 and 4, swap them since 8 is larger than 4. Array now looks like [3, 5, 4, 8, 2].
   d. Compare 8 and 2, swap them since 8 is larger than 2. Array now looks like [3, 5, 4, 2, 8].
   >
   The largest element (8) has now “bubbled up” to the end of the array.
2. Second pass through the array:
   a. Compare 3 and 5, no swap needed. Array still looks like `[3, 5, 4, 2, 8]`
   b. Compare 5 and 4, swap them since 5 is larger than 4. Array now looks like [3, 4, 5, 2, 8].
   c. Compare 5 and 2, swap them since 5 is larger than 2. Array now looks like [3, 4, 2, 5, 8].
   d. The second largest element (5) has now “bubbled up” to the second to last position in the array.
   >
3. Third pass through the array:
   a. Compare 3 and 4, no swap needed. Array still looks like [3, 4, 2, 5, 8].
   b. Compare 4 and 2, swap them since 4 is larger than 2. Array now looks like [3, 2, 4, 5, 8].
   c. Compare 4 and 5, no swap needed. Array still looks like [3, 2, 4, 5, 8].
   >
   The third largest element (4) has now “bubbled up” to the third to last position in the array.

Bubble Sort Part 2

4. Fourth pass through the array:
   a. Compare 3 and 2, swap them since 3 is larger than 2. Array now looks like [2, 3, 4, 5, 8].
   b. Compare 3 and 4, no swap needed. Array still looks like [2, 3, 4, 5, 8].
   c. Compare 4 and 5, no swap needed. Array still looks like [2, 3, 4, 5, 8].

The fourth largest element (3) has now “bubbled up” to the fourth to last position in the array.
5. Fifth pass through the array:
a. Compare 2 and 3, no swap needed. Array still looks like [2, 3, 4, 5, 8].
b. Compare 3 and 4, no swap needed. Array still looks like [2, 3, 4, 5, 8].
c. Compare 4 and 5, no swap needed. Array still looks like [2, 3, 4, 5, 8].

The fifth largest element (2) has now “bubbled up” to the beginning of the array.
6. The algorithm is finished since no more swaps are needed. The sorted array is [2, 3, 4, 5, 8].

Optimizations:

One optimization that can be made to bubble sort is to add a flag variable that indicates whether any swaps were made during the current pass through the array. If no swaps were made, then the array is already sorted and the algorithm can terminate early. This can significantly improve the performance of the algorithm for nearly sorted arrays.

Here’s an implementation of the optimized bubble sort algorithm in JavaScript:

```
function bubbleSortOptimized(arr) {
  let len = arr.length;
  let swapped;
  do {
    swapped = false;
    for (let i = 0; i < len - 1; i++) {
      if (arr[i] > arr[i + 1]) {
        let temp = arr[i];
        arr[i] = arr[i + 1];
        arr[i + 1] = temp;
        swapped = true;
      }
    }
    len--;
  } while (swapped);
  return arr;
}


const unsortedArray2 = [5, 3, 8, 4, 2]

console.log('\n\n\nBUBBLE SORT (OPTIMIZED):\n\n')

console.log('Unsorted array: ' + '[' + unsortedArray2 + ']')

const startTime2 = performance.now()
const result2 = bubbleSortOptimized(unsortedArray2)
const endTime2 = performance.now()
const executionTime2 = endTime2 - startTime2
console.log('Result: ' + '[' + result2 + ']')
console.log('Execution time: ' + executionTime2 + '
        milliseconds')
```

The optimized `bubbleSortOptimized` function works similarly to the `bubbleSort` function, but it adds a `swapped` flag variable and a `do-while` loop that continues iterating through the array until no swaps are made during a pass through the array. The `len--` line is added to reduce the number of comparisons on each pass through the array since the largest element is guaranteed to be at the end of the array after each pass. This optimization reduces the worst-case time complexity of the algorithm to O(n) for nearly sorted arrays.

>

You could edit the code in the IDE to show every iteration as the bubble sort function runs through your array if you want to see it progress.

>

Measures of Time

Since we've been talking about time complexity, the code in the IDE also

adds a calculation on how long the bubble sort takes. Since the array is fairly small, when you refresh you will usually get zeros for both the regular and optimized bubble sort. The zero simply means the function has taken far less than a millisecond to run. *You may also sometimes see the optimized time show up as 1 ms and the unoptimized time as 0 ms.* This is due to inherent variability in the time it takes the processor to run functions. Best practices to test this out would involve running the same algorithm many times and then taking the average to get a more accurate value. If you were to use a very large array, you might start to see a real effect here. Often, programmers will measure the time their algorithms take. For your personal website, shaving off a few milliseconds is not a big deal. For an enterprise, saving a few milliseconds over millions of calculations can add up!

Big O and Use Case Summary

- **Time complexity**: $O(n^2)$.
- **Space complexity**: O(1).
- **Best suited scenario**: When sorting small lists or when simplicity of implementation is a priority. For example, sorting a small list of numbers.

Merge Sort Part 1

Theory

One of the most popular sorting algorithms is the merge sort algorithm. It is a divide-and-conquer algorithm that works by splitting a collection into smaller sub-collections, sorting them, and merging them back together in the correct order.

The merge sort algorithm has a time complexity of O(n log n), which makes it efficient for sorting large collections of items. The basic idea behind the merge sort algorithm is to divide the collection into smaller sub-collections until each sub-collection contains only one element. Then, the sub-collections are merged back together in the correct order. The merging process involves comparing the first elements of each sub-collection and selecting the smaller one. The selected element is removed from its sub-collection and added to a new collection. This process is repeated until all sub-collections have been merged.

Implementation

First let's the see the code:

```
function mergeSort(array) {
  if (array.length <= 1) {
    return array;
  }

  const middle = Math.floor(array.length / 2);
  const left = array.slice(0, middle);
  const right = array.slice(middle);

  return merge(mergeSort(left), mergeSort(right));
}

function merge(left, right) {
  const result = [];
  let leftIndex = 0;
  let rightIndex = 0;

  while (leftIndex < left.length && rightIndex < right.length) {
    if (left[leftIndex] < right[rightIndex]) {
      result.push(left[leftIndex]);
      leftIndex++;
    } else {
      result.push(right[rightIndex]);
      rightIndex++;
    }
  }

  return
        result.concat(left.slice(leftIndex)).concat(right.slice(rightI
        ndex));


}
```

Let’s walk through an example to see how the merge sort algorithm works. Go ahead and add the following code:

```
const unsortedArray = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
        23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87,
        17, 8, 72, 70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53,
        66, 25, 56, 12, 4, 69, 27, 3, 83, 78, 55, 97]



console.log('MERGE SORT:\n\n')

console.log('Unordered array: ' + '[' + unsortedArray + ']')

const start1 = performance.now()
const result = mergeSort(unsortedArray)
const end1 = performance.now()
time1 = end1 - start1
console.log('Result: ' + '[' + result + ']')
console.log('Execution time: ' + time1 +' milliseconds')
```

1. We call the mergeSort function with the unsorted array as input.
2. The mergeSort function checks if the length of the array is less than or equal to 1. In this case, the length is 8, so we proceed to the next step.
3. The function calculates the middle index of the array, which is 4. It then splits the array into two sub-collections: [5, 3, 8, 4] and [2, 7, 1, 6].
4. The function recursively calls itself on the left and right sub-collections.
5. The merge function is called with the two sorted sub-collections as input.
6. The merge function compares the first elements of the left and right arrays and selects the smaller one. It adds the selected element to the result array and increments the corresponding index variable. This process is repeated until one of the sub-collections is empty.
7. The merge function returns the sorted result array to the mergeSort function.
8. The mergeSort function returns the sorted array.
   >
   Optimizations:
   There are a few optimizations that can be made to the merge sort algorithm:
9. **In-Place Sorting**: The merge sort algorithm can be modified to sort the input array in-place instead of creating temporary arrays. This reduces the space complexity to O(1).
   >
   Here's what this would look like:

```
// This function sorts the input array in-place using merge sort
        algorithm
function mergeSortInPlace(array, left = 0, right = array.length
        - 1) {
  if (left >= right) {
    return
  }

  const middle = Math.floor((left + right) / 2)
  mergeSortInPlace(array, left, middle)
  mergeSortInPlace(array, middle + 1, right)
  const result = mergeInPlace(array, left, middle, right)

  return result
}

// This function merges two sorted sub-arrays within the input
        array in-place
function mergeInPlace(array, left, middle, right) {
  const result = []
  const leftArray = array.slice(left, middle + 1)
  const rightArray = array.slice(middle + 1, right + 1)
  let i = 0
  let j = 0

  for (let k = left; k <= right; k++) {
    if (i >= leftArray.length) {
      array[k] = rightArray[j]
      j++
    } else if (j >= rightArray.length) {
      array[k] = leftArray[i]
      i++
    } else if (leftArray[i] <= rightArray[j]) {
      array[k] = leftArray[i]
      i++
    } else {
      array[k] = rightArray[j]
      j++
    }
  }

  return result.concat(leftArray, rightArray)
}
```

Merge Sort Part 2

2. **Tail Recursion**: The merge sort algorithm can be implemented using tail recursion to reduce the call stack size. This can improve the performance and reduce the risk of stack overflow errors.
   >
   Here's what this would look like:

```
// This function sorts the input array using merge sort
        algorithm and tail recursion
function mergeSortTailRecursion(array) {
  // Define a helper function called mergeSortRec that takes two
        arguments: left index and right index
  function mergeSortRec(left, right) {
    // If the left index is greater than or equal to the right
        index, return
    if (left >= right) {
      return
    }

    // Calculate the middle index and recursively call
        mergeSortRec on the left and right sub-arrays
    const middle = Math.floor((left + right) / 2)
    mergeSortRec(left, middle)
    mergeSortRec(middle + 1, right)

    // Merge the two sorted sub-arrays within the input array
        using merge function
    mergeRec(array, left, middle, right)
  }

  function mergeRec(array, left, middle, right) {
    // Create two new arrays to hold the left and right sub-
        arrays
    const leftArray = array.slice(left, middle + 1)
    const rightArray = array.slice(middle + 1, right + 1)

    // Initialize pointers for the left and right sub-arrays and
        the merged array
    let leftIndex = 0
    let rightIndex = 0
    let mergedIndex = left

    // Merge the two sub-arrays into the merged array in
        ascending order
    while (leftIndex < leftArray.length && rightIndex <
        rightArray.length) {
      if (leftArray[leftIndex] < rightArray[rightIndex]) {
```

```
        array[mergedIndex] = leftArray[leftIndex]
        leftIndex++
      } else {
        array[mergedIndex] = rightArray[rightIndex]
        rightIndex++
      }
      mergedIndex++
    }

    // Add any remaining elements from the left sub-array to the
        merged array
    while (leftIndex < leftArray.length) {
      array[mergedIndex] = leftArray[leftIndex]
      leftIndex++
      mergedIndex++
    }

    // Add any remaining elements from the right sub-array to
        the merged array
    while (rightIndex < rightArray.length) {
      array[mergedIndex] = rightArray[rightIndex]
      rightIndex++
      mergedIndex++
    }
  }

  // Call mergeSortRec function with the input array and the
      left and right indices
  mergeSortRec(0, array.length - 1)

  // Return the sorted input array
  return array
}
```

Go ahead and add the following code to test the two optimizations:

```
const unsortedArray2 = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
        23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87,
        17, 8, 72, 70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53,
        66, 25, 56, 12, 4, 69, 27, 3, 83, 78, 55, 97]


console.log('\n\n\nMERGE SORT (IN PLACE):\n\n')

console.log('Unordered array: ' + '[' + unsortedArray2 + ']')

const start2 = performance.now()
const result2 = mergeSortInPlace(unsortedArray2)
const end2 = performance.now()
time2 = end2 - start2
console.log('Result: ' + '[' + result2 + ']')
console.log('Execution time: ' + time2 +' milliseconds')



const unsortedArray3 = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
        23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87,
        17, 8, 72, 70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53,
        66, 25, 56, 12, 4, 69, 27, 3, 83, 78, 55, 97]

console.log('\n\n\nMERGE SORT (TAIL RECURSION):\n\n')

console.log('Unordered array: ' + '[' + unsortedArray3 + ']')

const start3 = performance.now()
const result3 = mergeSortTailRecursion(unsortedArray3)
const end3 = performance.now()
time3 = end3 - start3
console.log('Result: ' + '[' + result3 + ']')
console.log('Execution time: ' + time3 + ' milliseconds')
```

Big O and Use Case Summary

- **Time complexity**: O(n log n).
- **Space complexity**: O(n).
- **Best suited scenario**: When sorting large lists or when stability of sorting is important. For example, sorting a list of student grades.

Conclusion and Takeaway

> By mastering sorting algorithms, you will be able to choose the most efficient algorithm for a given sorting task in JavaScript. In this lesson, we endeavored to understand the theoretical basis, implementation, and Big O performance of the bubble sort and merge sort algorithms, with a view toward developing critical thinking and analytical skills which can be applied to real-world scenarios and also help in technical interviews. In the next lesson, we will go over two more important algorithms for traversing the binary search tree data structure.

Algorithms: Deep Dive - Part 2

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The theoretical basis, implementation, and Big O performance of breadth-first search tree traversal, and depth-first search tree traversal
> - How to implement these algorithms in JavaScript
>
> ### Introduction
>
> We already covered a couple algorithms in the last lesson. In this lesson, we will focus on two more algorithms: breadth-first search tree traversal, and depth-first search tree traversal. Like we've done previously, we will discuss the theoretical basis, implementation, and Big O performance of each algorithm.

Tree Traversal: Breadth First Search (BFS) Part 1

> ### Theory
>
> Breadth-First Search (BFS) is an algorithm that is used to traverse or search a graph or a tree data structure. It starts at the root node and visits all the nodes at the current depth before moving on to the next depth level. BFS is useful for finding the shortest path between two nodes in an unweighted graph or tree, and it can also be used to generate a tree of all the possible paths from the root node to every other node in the graph or tree.
>
> BFS uses a queue data structure to keep track of the nodes that need to be visited. The algorithm works as follows:
>
> 1. Enqueue the root node to the queue.
> 2. While the queue is not empty:
>   **a.** Dequeue the node at the front of the queue.
>   **b.** Visit the node.
>   **c.** Enqueue all the node's children to the back of the queue.
> 3. Repeat step 2 until the queue is empty.
>
> This algorithm guarantees that nodes are visited in order of their distance from the root node, so it is guaranteed to find the shortest path between two nodes if one exists.
>
> ### Implementation
>
> Here's how we would implement breadth first search:

```
breadthFirstSearch() {
  let currentNode = this.root
  let queue = []
  let results = []
  queue.push(currentNode)

  while (queue.length) {
    currentNode = queue.shift()
    results.push(currentNode.value)

    if (currentNode.left) {
      queue.push(currentNode.left)
    }

    if (currentNode.right) {
      queue.push(currentNode.right)
    }
  }

  return results
}
```

This implementation uses a while loop to continue the BFS algorithm until the queue is empty. We start by enqueuing the root node to the queue, and then we dequeue the node at the front of the queue and visit it. We then enqueue all the node's children to the back of the queue. We repeat this process until the queue is empty.
>
Let's walk through the BFS algorithm to see how it works:
>
1. Initialize a currentNode variable to the root node of the tree.
2. Create an empty array called queue to serve as our queue data structure, and another empty array called results to store the visited nodes in the order that they were visited.
3. Add the currentNode to the back of the queue using the push() method.
4. Begin a while loop that continues as long as the queue is not empty. This loop will perform the BFS algorithm.
5. Inside the loop, dequeue the node at the front of the queue using the shift() method, and store its value in the currentNode variable.
6. Push the value of currentNode to the end of the results array using the push() method.
7. Check if currentNode has a left child. If it does, enqueue the left child to the back of the queue using the push() method.
8. Check if currentNode has a right child. If it does, enqueue the right child to the back of the queue using the push() method.
9. The while loop will repeat steps 5-8 until the queue is empty.

10. Return the results array containing the visited nodes in the order that they were visited.
>

Tree Traversal: Breadth First Search (BFS) Part 2

Now let's look at an example. Add the following code at the end of our JavaScript file:

```
const newTree = new BinarySearchTree()

console.log('BINARY SEARCH TREE (BFS) OUTPUT:\n\n')

// Insert 15 nodes with unique random values between 10 and 100
const values = []
while (values.length < 15) {
  const value = Math.floor(Math.random() * 91) + 10
  if (!values.includes(value)) {
    values.push(value)
    newTree.insertNode(value)
  }
}

console.log('Binary Search Tree: ' + JSON.stringify(newTree,
        null, '\t'))

// Result of breadth first search method
const start = performance.now()
const result = newTree.breadthFirstSearch()
const end = performance.now()
time = end - start
console.log('Result: [' + result + ']')
console.log('Execution time: ' + time + ' milliseconds')
```

Big O and Use Case Summary

- **Time complexity**: O(n)
- **Space complexity**: O(n).
- **Best suited scenario**: When traversing a tree or graph and finding the shortest path between two nodes is important. For example, finding the shortest path between two cities on a map.

Tree Traversal: Depth First Search (DFS) Part 1

> ### Theory
>
> Depth-first search (DFS) is a popular algorithm for traversing a graph or a tree data structure, and comes in three types: pre-order, in-order, and post-order. It can be used to search for a particular node, to count the number of nodes, to determine if the graph is connected, and to perform various other tree or graph-related operations.
>
> The DFS algorithm starts at the root node of the tree and visits each node in the tree, going as deep as possible before backtracking.
>
> In a pre-order traversal, the algorithm visits the root node, then recursively visits the left subtree, and finally recursively visits the right subtree. This traversal is useful for copying the structure of the tree.
>
> In an in-order traversal, the algorithm first visits the left subtree, then the root node, and finally the right subtree. This traversal is useful for printing out the tree in sorted order.
>
> In a post-order traversal, the algorithm first visits the left subtree, then the right subtree, and finally the root node. This traversal is useful for deleting the entire tree, as it visits the child nodes before the parent nodes.
>
> ### Implementation
>
> Let's look at how we can implement DFS tree traversal using recursive functions. We will look at each of the three types of depth first search.

Pre-Order Traversal

Add the following code inside our **BinarySearchTree** class where indicated:

```
depthFirstPreOrder() {
  let results = []


  function traverse(currentNode) {
    if (!currentNode) return


    results.push(currentNode.value)


    if (currentNode.left) {
      traverse(currentNode.left)
    }


    if (currentNode.right) {
      traverse(currentNode.right)
    }
  }


  if (this.root) {
    traverse(this.root)
    return results
  } else {
    throw new Error('Tree is empty - root node does not exist.')
  }
}
```

Now, let's go through this step by step to see how it works:

1. The function starts by creating an empty array called **results** to store the traversal results.
2. It defines a helper function called **traverse()** that takes a single argument, **currentNode**, which represents the current node being visited during the traversal.
3. **The traverse()** function first checks if the **currentNode** argument is null or undefined. If it is, the function immediately returns without doing anything.
4. If the currentNode argument is not null or undefined, the function pushes the value of the current node to the **results** array using the **push()** method.
5. The **traverse()** function then recursively calls itself on the left child of the current node, if it exists.
6. It then recursively calls itself on the right child of the current node, if it exists.

7. Once the **traverse()** function has finished visiting all nodes in the tree, it returns the results array.
8. If the tree has a root node, the **depthFirstPreOrder()** function calls the **traverse()** function with the root node to start the traversal.
9. Once the traversal is complete, the function returns the **results** array.
10. If the tree is empty (i.e., there is no root node), the function throws an error to indicate that the traversal cannot be performed.

Tree Traversal: Depth First Search (DFS) Part 2

In-Order Traversal

To change our DFS traversal type to in-order traversal, we actually only need to make a change to the **traverse** function inside the method. So, instead of:

```
function traverse(currentNode) {
  if (!currentNode) return

  results.push(currentNode.value)

  if (currentNode.left) {
    traverse(currentNode.left)
  }

  if (currentNode.right) {
    traverse(currentNode.right)
  }
}
```

We would move the **push** operation between the two recursive operations. Go ahead and add the following modified **traverse** function inside the **depthFirstInOrder** method where indicated:

```
 function traverse(currentNode) {
      if (!currentNode) return

      if (currentNode.left) {
        traverse(currentNode.left)
      }

      results.push(currentNode.value)

      if (currentNode.right) {
        traverse(currentNode.right)
      }
    }
```

This will now effectively return all our node values in numerical order from smallest to largest!
>

Post-Order Traversal

Now, for post-order traversal, we again only have to make a small change to the **traverse** function. We will now move the **push** operation to the end of the function. Add the following modified **traverse** function inside the **depthFirstPostOrder** method where indicated:

```
function traverse(currentNode) {
  if (!currentNode) return

  if (currentNode.left) {
    traverse(currentNode.left)
  }

  if (currentNode.right) {
    traverse(currentNode.right)
  }

  results.push(currentNode.value)
}
```

What's essentially happening is that the post-order traversal algorithm first visits all the nodes in the left subtree recursively, then the right subtree recursively, and finally the current node. This is in contrast to pre-order traversal, which visits the current node before visiting its child nodes.
>

Now let's look at an example. Add the following code to the end of our JavaScript file:

```
const newTree = new BinarySearchTree()

console.log('BINARY SEARCH TREE (DFS) OUTPUT:\n\n')

// Insert 15 nodes with unique random values between 10 and 100
const values = []
while (values.length < 15) {
  const value = Math.floor(Math.random() * 91) + 10
  if (!values.includes(value)) {
    values.push(value)
    newTree.insertNode(value)
  }
}

console.log('Binary Search Tree: ' + JSON.stringify(newTree,
        null, '\t'))

// Result of depth first search (preOrder) method
const start = performance.now()
const result = newTree.depthFirstPreOrder()
const end = performance.now()
time = end - start
console.log('\nDepthFirstPreOrder result: [' + result + ']')
console.log('Execution time: ' + time + ' milliseconds')


// Result of depth first search (inOrder) method
const start2 = performance.now()
const result2 = newTree.depthFirstInOrder()
const end2 = performance.now()
time2 = end2 - start2
console.log('\nDepthFirstInOrder result: [' + result2 + ']')
console.log('Execution time: ' + time2 + ' milliseconds')


// Result of depth first search (postOrder) method
const start3 = performance.now()
const result3 = newTree.depthFirstPostOrder()
const end3 = performance.now()
time3 = end3 - start3
console.log('\nDepthFirstPostOrder result: [' + result3 + ']')
console.log('Execution time: ' + time3 + ' milliseconds')
```

Big O and Use Case Summary

- **Time complexity**: O(n).
- **Space complexity**: O(n).

- **Best suited scenario**: When traversing a tree and finding the shortest path between two nodes is important. For example, finding the shortest path between two cities on a map.

Conclusion & Takeaway

> In this lesson, we finished our deep dive into some of the common searching and sorting algorithms in JavaScript. We went over two different algorithms for tree traversal: breadth-first search, and depth-first search using pre-order, in-order, and post-order traversal methods. In the next lesson, we'll do a brief overview of some common problem solving patterns, an example coding question that may come up in technical interviews and some various resources for getting practice with writing algorithms and solving coding challenges, as well as a couple for getting practice building different UI elements.

More Algorithms (AEL)

> ### Goals
>
> By the end of this lesson, you should understand:
>
> - The theoretical basis, and Big O performance of selection sort, insertion sort, and radix sort
> - How to implement these algorithms in JavaScript
>
> ### Introduction
>
> We already covered several algorithms in the main lesson. In this AEL lesson, we will focus on three more sorting algorithms: selection sort, insertion sort, and radix sort. Like we've done previously, we will discuss the theoretical basis, implementation, and Big O performance of each algorithm.

Selection Sort Part 1

Theory

Another one of the simplest sorting algorithms is selection sort. This algorithm sorts an array by repeatedly finding the minimum element from the unsorted part of the array and placing it at the beginning of the sorted part of the array.

Before diving into the implementation of the algorithm, let's take a closer look at how it works. The Selection Sort algorithm works as follows:

1. Set the first element of the array as the minimum element.
2. Compare the minimum element with the next element in the array.
3. If the next element is smaller than the minimum element, set the next element as the new minimum element.
4. Repeat step 3 for all the elements in the array.
5. Swap the minimum element with the first element of the unsorted part of the array.
6. Repeat steps 1 to 5 until the entire array is sorted.

The selection sort algorithm has a time complexity of $O(n^2)$, where n is the number of elements in the array. This means that the algorithm will take longer to sort large arrays. However, it is a simple algorithm to implement and is useful for sorting small arrays.

Implementation

Now that we have a good understanding of the Selection Sort algorithm, let's implement it in JavaScript. Here is the code:

```
function selectionSort(arr) {
  for (let i = 0; i < arr.length - 1; i++) {
    let minIndex = i;
    for (let j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[minIndex]) {
        minIndex = j;
      }
    }
    if (minIndex !== i) {
      let temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
    }
  }
  return arr;
}
```

Let's go through the code to understand how the algorithm works.

We define a function called **selectionSort** that takes an array as an argument. Then, we start a loop that will iterate through the array. The loop will run until i is less than **arr.length - 1**. We then set **minIndex** to i, which is the current element of the loop. Following this, we start another loop that will compare the current element with the rest of the elements in the array. If the element being compared is smaller than the current minimum element, we update **minIndex** to the index of the smaller element. After the inner loop has been completed, we check if **minIndex** has changed from its initial value. If it has, we swap the current element with the minimum element. Finally, we return the sorted array.
>
Now that we have implemented the Selection Sort algorithm, let's test it with some sample inputs. Here are some test cases:

```
console.log(selectionSort([3, 2, 1])); // [1, 2, 3]
console.log(selectionSort([5, 3, 6, 2, 10])); // [2, 3, 5, 6,
10]
console.log(selectionSort([1])); // [1]
console.log(selectionSort([])); // []
```

In the first test case, we pass an array with three unsorted elements. The expected output is an array with the same elements in sorted order. The output matches our expectations.
>
In the second test case, we pass an array with five unsorted elements. The expected output is an array with the same elements in sorted order. The output matches our expectations.

>
In the third test case, we pass an array with only one element. The expected output is the same array, as it is already sorted. The output matches our expectations.
>
In the fourth test case, we pass an empty array. The expected output is an empty array, as there is nothing to sort. The output matches our expectations.
>

Selection Sort Part 2

Test It Out

Now that we've implemented the selection sort algorithm, let's add the following to test our code:

```
console.log('\n\n\nSELECTION SORT:\n\n')

const unsortedArray = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87, 17, 8, 72,
70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53, 66, 25, 56, 12, 4,
69, 27, 3, 83, 78, 55, 97]

console.log('Unordered array: ' + '[' + unsortedArray + ']' +
'\n')

const start = performance.now()
const result = selectionSort(unsortedArray)
const end = performance.now()
time = end - start
console.log('Result: ' + '[' + result + ']')
console.log('Execution time: ' + time +' milliseconds')
```

Big O and Use Case Summary

- **Time complexity**: O(n^2).
- **Space complexity**: O(1).
- **Best suited scenario**: When sorting small lists or when simplicity of implementation is a priority. For example, sorting a small list of strings.

Insertion Sort Part 1

> ### Theory
>
> Insertion sort is another simple sorting algorithm that works well for small datasets. It is based on the idea of taking an element from an unsorted array and inserting it into its proper place in a sorted array. The algorithm sorts the array by comparing each element with the elements that come before it in the array, and inserting it into its proper place.
>
> The algorithm works by dividing the array into two sections: a sorted section and an unsorted section. Initially, the sorted section contains only the first element of the array, and the unsorted section contains all the remaining elements. The algorithm then takes the first element from the unsorted section and inserts it into its proper place in the sorted section. This process is repeated until all elements have been moved from the unsorted section to the sorted section.
>
> ### Implementation
>
> Here is the implementation of the insertion sort algorithm in JavaScript:

```
function insertionSort(arr) {
  for (let i = 1; i < arr.length; i++) {
    let j = i;
    while (j > 0 && arr[j-1] > arr[j]) {
      let temp = arr[j];
      arr[j] = arr[j-1];
      arr[j-1] = temp;
      j--;
    }
  }
  return arr;
}
```

Let's walk through the implementation of the insertion sort algorithm on the array [3, 5, 1, 4, 2], to get a better idea of how it works:

```
let arr = [3, 5, 1, 4, 2];
console.log(insertionSort(arr)); // Output: [1, 2, 3, 4, 5]
```

1. Start with the first element, which is already sorted. The sorted section contains [3], and the unsorted section contains [5, 1, 4, 2].
2. Take the first element from the unsorted section, which is 5, and compare it with the last element in the sorted section, which is 3. Since 5 is greater than 3, we do not need to move it. The sorted section contains [3, 5], and the unsorted section contains [1, 4, 2].
3. Take the first element from the unsorted section, which is 1, and compare it with the last element in the sorted section, which is 5. Since 1 is less than 5, we need to move 5 in one position to the right to make room for 1. The sorted section contains [3, 1, 5], and the unsorted section contains [4, 2].
4. Take the first element from the unsorted section, which is 4, and compare it with the last element in the sorted section, which is 5. Since 4 is less than 5, we need to move 5 in one position to the right to make room for 4. The sorted section contains [3, 1, 4, 5], and the unsorted section contains [2].
5. Take the first element from the unsorted section, which is 2, and compare it with the last element in the sorted section, which is 5. Since 2 is less than 5, we need to move 5 in one position to the right to make room for 2. Next, we compare 2 with the previous element in the sorted section, which is 4. Since 2 is less than 4, we need to move 4 in one position to the right to make room for 2. Finally, we compare 2 with the first element in the sorted section, which is 3. Since 2 is less than 3, we need to move 3 in one position to the right to make room for 2. The sorted section now contains [1, 2, 3, 4, 5], and the unsorted section is empty.
6. The algorithm has been completed, and the sorted array [1, 2, 3, 4, 5] is returned.

Insertion Sort Part 2

Test It Out

Now that we've implemented the insertion sort algorithm, let's add the following to test our code:

```
const unsortedArray = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87, 17, 8, 72,
70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53, 66, 25, 56, 12, 4,
69, 27, 3, 83, 78, 55, 97]

console.log('\n\n\nINSERTION SORT:\n\n')

console.log('Unordered array: ' + '[' + unsortedArray + ']' +
'\n')

const start = performance.now()
const result = insertionSort(unsortedArray)
const end = performance.now()
console.log('Result: ' + '[' + result + ']')
time = end - start
console.log('Execution time: ' + time +' milliseconds')
```

> #### Big O and Use Case Summary
>
> - **Time complexity**: O(n^2).
> - **Space complexity**: O(1).
> - **Best suited scenario**: When sorting small lists or when the input list is almost sorted. For example, sorting a small list of dates.

Radix Sort Part 1

> ### Theory
>
> Radix sort is a linear time sorting algorithm that works on integers, and it can handle large arrays with a large range of integers. The way this algorithm works is by dividing the input array into groups based on the values of the digits in each element. For example, if we have an array of integers, we can group them based on the units digit, the tens digit, the hundreds digit, and so on. We sort each group independently, and then combine them to get the final sorted array.
>
> ### Implementation
>
> Here's an example of implementing radix sort:

```
// 1. Define the radixSort function that takes an array as input
function radixSort(arr) {
  // 2. Define the helper function getMax to find the maximum
value in the array
  const getMax = (arr) => {
    let max = 0
    for (let i = 0; i < arr.length; i++) {
      if (arr[i] > max) {
        max = arr[i]
      }
    }
    return max
  }

  // 3. Define the helper function digitCount to count the
number of digits in a number
  const digitCount = (num) => {
    if (num === 0) return 1  // special case for 0
    return Math.floor(Math.log10(Math.abs(num))) + 1
    // returns the number of digits using log10 and adding 1 to
get the count
  }

  // 4. Define the helper function getDigit to get the digit at
a given position
  const getDigit = (num, i) => {
    return Math.floor(Math.abs(num) / Math.pow(10, i)) % 10
    // returns the digit by dividing the number by 10 to the
```

```
power of i,
    // taking the floor of the result, and getting the remainder
after dividing by 10
  }

  // 5. Get the maximum number of digits in the input array
  const maxDigits = digitCount(getMax(arr))

  // 6. Loop through each digit position, starting from the
least significant digit
  for (let k = 0; k < maxDigits; k++) {
    // 7. Create 10 buckets for each possible digit value (0-9)
    const buckets = Array.from({ length: 10 }, () => [])
    for (let i = 0; i < arr.length; i++) {
      // 8. Get the digit at the current position for the
current element
      const digit = getDigit(arr[i], k)
      // 9. Add the current element to the corresponding bucket
based on its digit value
      buckets[digit].push(arr[i])
    }
    // 10. Concatenate the buckets in order (0-9) to create a
new sorted array
    arr = [].concat(...buckets)
  }

  // 11. Return the sorted array
  return arr
}
```

Radix Sort Part 2

Let's walk through the implementation of radix sort, step by step:
1. Define the **radixSort** function that takes an array as input
2. Define the helper function **getMax** to find the maximum value in the array
3. Define the helper function **digitCount** to count the number of digits in a number
4. Define the helper function **getDigit** to get the digit at a given position
5. Get the maximum number of digits in the input array
6. Loop through each digit position, starting from the least significant digit
7. Create 10 buckets for each possible digit value (0-9)
8. Get the digit at the current position for the current element
9. Add the current element to the corresponding bucket based on its digit value
11. Concatenate the buckets in order (0-9) to create a new sorted array
12. Return the sorted array
>

Optimizations:
Radix sort is already a significantly optimized algorithm. However, one possible additional optimization that could be applied would be to use a hybrid sorting algorithm that combines radix sort with quick sort or merge sort to improve performance when the input array is partially sorted.

Test It Out

Now that we've written the radix sort algorithm, let's add the following to test our code:

```
const unsortedArray = [100, 41, 92, 36, 15, 86, 64, 35, 88, 26,
23, 43, 75, 63, 52, 14, 1, 24, 73, 5, 74, 50, 81, 87, 17, 8, 72,
70, 31, 10, 91, 39, 99, 48, 94, 80, 37, 53, 66, 25, 56, 12, 4,
69, 27, 3, 83, 78, 55, 97]

console.log('RADIX SORT:\n\n')

console.log('Unordered array: ' + '[' + unsortedArray + ']' +
'\n')


const start = performance.now()
const result = radixSort(unsortedArray)
const end = performance.now()
console.log('Result: ' + '[' + result + ']')
time = end - start
console.log('Execution time: ' + time +' milliseconds')
```

Big O and Use Case Summary

- **Time complexity**: O(nk), where n is the number of elements to sort, and k is the number of digits.
- **Space complexity**: O(n + k).
- **Best suited scenario**: When sorting large lists of integers with a fixed number of digits. For example, sorting a list of phone numbers.

Conclusion & Takeaway

> In this AEL lesson, we went over additional common sorting algorithms: selection sort, insertion sort, and radix sort. We covered a little of the theory behind them, how to implement them in JavaScript, and looked at a summary of their Big O complexities.

Common Algorithm Patterns

> There are nearly infinite variations of coding problems you could face in your career as a developer, in technical interviews, or when building your own projects. As stated, an algorithm is simply a defined set of steps to take in order to solve a particular problem, and strictly speaking we write algorithms at some point every time we build a project! Knowing different ways of approaching or framing problems, therefore, can be useful.

There are some common patterns for problems that you may encounter again and again, and can sometimes be a helpful source of inspiration for coming up with a solution when faced with coding problems in technical interviews. Let's look at some examples of these.
>
Some problem solving patterns in the wild
>
1. **Brute Force**: This involves trying all possible solutions until a correct one is found. This approach is simple but not efficient for large problems.
>
2. **Greedy Algorithm**: This involves making the locally optimal choice at each step, hoping that it will lead to a globally optimal solution. This approach can be fast and efficient but may not always lead to the optimal solution.
>
3. **Divide and Conquer**: This involves breaking a problem down into smaller subproblems, solving each subproblem recursively, and then combining the solutions. This approach can be very efficient for certain types of problems.
>
4. **Dynamic Programming**: This involves breaking a problem down into smaller subproblems and solving each subproblem only once, storing the solution to each subproblem in a table to avoid redundant computation. This approach can be very efficient for problems with overlapping subproblems.
>
5. **Backtracking**: This involves trying out all possible solutions, but as soon as a mistake is made, it backs up and tries a different solution. This approach is often used when the solution space is too large to be searched exhaustively.
>
6. **Sliding Window**: This pattern involves creating a "window" of a fixed size that slides over an array or string, allowing you to efficiently calculate values based on the elements within the window. This pattern can be useful for problems that involve finding subarrays or substrings that meet

certain criteria, such as having a certain sum or length.
>
7. **Two Pointers**: This pattern involves using two pointers to traverse an array or string in tandem, with one pointer moving faster than the other. This approach can be useful for problems that involve finding pairs or subarrays that meet certain criteria, such as having a certain sum or satisfying a certain condition.
>
8. **Memoization**: This involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. This approach can be useful for problems that involve computing the same value multiple times, such as in dynamic programming.
>
Note that this is by no means an exhaustive list. These are just a few common patterns, or approaches, to solving problems that have been given a name. If you want to learn more about any or all of these, feel free to look them up on Google!

Example Coding Question

The classic “FizzBuzz” problem

Everything we have been learning about data structures and algorithms in JavaScript is to help give a solid foundational knowledge for technical interviews. These topics come up quite often in these types of interviews – especially at the big tech companies.
>
One classic example of a coding question that may come up in technical interviews is some form of the FizzBuzz question. The question will be phrased something along the lines of:
>
“Create a function that takes a range of numbers as input and returns an array of strings. If a number in the range is **divisible by 3**, the corresponding element in the array should be ‘**Fizz**’. If a number is **divisible by 5**, the element should be ‘**Buzz**’. If the number is **divisible by both 3 and 5**, the element should be ‘**FizzBuzz**’. Otherwise, the element should be the number itself as a string.”
>
Try working on this problem for a bit, and feel free to ask your TA or instructor for help if you get stuck.

So, let’s look at an example solution to this problem (as stated above) and then walk through it step by step:

```
function fizzBuzz(start, end) {
  let result = []
  for (let i = start; i <= end; i++) {
    if (i % 3 === 0 && i % 5 === 0) {
      result.push("FizzBuzz")
    } else if (i % 3 === 0) {
      result.push("Fizz")
    } else if (i % 5 === 0) {
      result.push("Buzz")
    } else {
      result.push(i.toString())
    }
  }
  return result
}


console.log('FIZZBUZZ EXAMPLE OUTPUT:\n\n')

const fizzBuzzResults = fizzBuzz(1, 100)

console.log('Results: ' + '[' + fizzBuzzResults + ']')
```

This code defines a function **fizzBuzz** that takes two arguments, **start** and **end**, which represent the range of numbers to check. It creates an empty array **result** to store the output strings, and then uses a **for** loop to iterate through the numbers in the range. For each number, it checks whether it's divisible by 3, 5, or both, and appends the corresponding string to the **result** array. If the number isn't divisible by either 3 or 5, it appends the number as a string using the **toString()** method. Finally, the function returns the **result** array.

Technical Interviews and Practice

> In technical interviews, there could be any number of different problems like “FizzBuzz”, where the interviewer starts off a coding problem by saying “Create a function that…” or something similar.
>
> Oftentimes, in the pressure of an interview setting, it can be common to panic and freeze up – even if, under other circumstances, you would somewhat easily be able to reason through the problem and come up with a solution.
>
> There are a few things that are helpful to remember, however:
>
> - **First**, breathe and have confidence in yourself and your abilities!
>
> - **Second**, if you don’t understand something about what the interviewer is asking you to do, or just to verify that you do understand, it is OK to ask clarifying questions or to restate the problem in your own words.
> - **Third**, before you start writing a single line of code, try writing down in plain language the steps that come to mind to solve the problem, and even say those steps aloud as you write them.
> - **Fourth**, while it’s good to try to come up with the most efficient solution possible (think Big O) if you can, still don’t worry so much about this… the most important thing the interviewer is looking for a lot of the time is to see your thought process and how you approach challenging problems.
> - **Fifth**, if you get stuck at a certain point you can (believe it or not) ask your interviewer for any suggestions they may have for how to proceed.
>
> All that being said, if you want to walk into technical interviews with less anxiety, the absolute best thing you can do is practice! Keep learning and keep coding. In the next section, we’ll look at some resources where you can practice a multitude of coding questions to keep your skills sharp.
>
> ### Where to get further practice
>
> There are several resources for this, actually, but here we will focus on three of the most popular:
> - LeetCode
> - Hackerrank
> - Codewars

> For all three of these providers, you can sign up for a **free account** and solve problems whenever you have the time. This is a great way to beef up your skills as a developer, as well as your prowess and confidence in technical interviews!
>
> Remember, though, it's important to practice building **real-world projects** as well! If you're applying for a frontend developer job and you can reverse a linked list and write the merge sort algorithm blindfolded, but you're super shaky on building a real website UI (simple or complex) you're not likely to get very far. To that end, there are a couple of other free resources where you can get this kind of "real UI" practice:
> - Frontend Mentor
> - UI Design Daily

Project: Algo Practice & Add to Personal Website

Overview

For Project 13, you will get a little bit of practice tackling coding questions, upload the code for your attempts to GitHub, and also add to your website's project session!

Step One

Complete the following:
- Sign up for free accounts on LeetCode, Hackerrank, and CodeWars
- Create a GitHub repository with 3 folders (one for each of the above platforms), and a brief README file with at least 2 sentences describing your repo
- Work on 2 - 3 beginner level challenges from any of the above platforms (or one from each!)
- Push all of your code to the GitHub repo you created
- Submit a link to your repo and, if it's a *private* repo, add your TA as a collaborator

> **NOTE**: Try your best to solve the coding challenges, using comments to describe the thought process behind your approach. You can also ask your TA for help with these. But even if you can't arrive at a solution that's okay! The only requirement for these challenges, in terms of this project submission, is that it's evident that you employed critical thinking and attempted a solution.

Step Two

Add the calculator mini project (from Lesson 1.3), or another project you completed outside of class, to the projects section of your website. The styling is up to you, but you should include a screenshot, a title, a brief description, and a link to the code on GitHub. Finally, submit the GitHub link below (again, if it's a private repo, be sure to add your TA as a collaborator).

Happy coding!
>

> The only thing worse than not knowing why some code breaks is not knowing why it worked in the first place! It's the classic "house of cards" mentality: "it works, but I'm not sure why, so nobody touch it!" You may have heard, "Hell is other people" (Sartre), and the programmer meme twist, "Hell is other people's code."
> I believe truly: "Hell is not understanding my own code."
> — Kyle Simpson, You Don't Know JS: Async & Performance