

➤ PORTAFOLIO  
DE EVIDENCIAS  
-ESTRUCTURA  
DE DATOS  
➤ 3ER  
SEMESTRE

NOMBRE DEL  
ALUMNO:  
MARIN  
CORREO  
ANGEL

## Análisis - Semana 1: Introducción a JavaScript y Arreglos

### Descripción General

Esta carpeta contiene el material introductorio de la materia **Estructuras de Datos**, enfocándose en los conceptos fundamentales de **JavaScript** y la introducción a **estructuras de datos lineales**, específicamente **arreglos (arrays)**.

---

### Lenguaje

- **JavaScript (ES6+)**
- **HTML5**

El proyecto utiliza JavaScript vanilla sin frameworks externos, permitiendo aprender los conceptos fundamentales desde cero.

---

### Estructura de Archivos

Semana1/

```
|-- code.js          # Funciones básicas de JS (saludar, FizzBuzz)  
|-- index.html      # Página HTML principal  
|-- arrays/         # Implementación personalizada de arreglos  
|  |-- index.js     # Clase custom de Array con métodos  
`-- array/          # Temas avanzados de arreglos  
    |-- intro/        # Introducción a métodos de array  
    |-- map/          # Método map() - transformación de datos  
    |-- filter/        # Método filter() - filtrado de datos  
    |-- reduce/        # Método reduce() - reducción de datos  
    |-- delete/        # Eliminación de elementos  
    |-- immutability/  # Conceptos de inmutabilidad
```

└─ functionalProgramimng/ # Programación funcional con arrays

---

## Objetivos

1. **Introducción a JavaScript:** Aprender la sintaxis básica y cómo integrar JS con HTML
  2. **Resolver problemas lógicos:** Implementar algoritmos clásicos como FizzBuzz
  3. **Entender estructuras de datos:** Crear una clase Array personalizada desde cero
  4. **Métodos de array:** Dominar los métodos funcionales (map, filter, reduce)
  5. **Programación Funcional:** Introducir conceptos de programación funcional con arrays
  6. **Inmutabilidad:** Comprender la importancia de la inmutabilidad en JavaScript
- 

## Descripción Detallada

### Archivos Principales

#### **code.js**

Archivo introductorio que contiene tres funciones fundamentales:

1. **saludar():** Función básica que muestra un alert de bienvenida
2. **teSaludo(nombre):** Función con parámetro que personaliza el saludo
3. **fizzBuzz():** Implementación del problema clásico "FizzBuzz"
  - Itera números del 1 al 100
  - Imprime "Fizz" para múltiplos de 3
  - Imprime "Buzz" para múltiplos de 5
  - Imprime "FizzBuzz" para múltiplos de 3 y 5
  - Imprime el número si no cumple las condiciones anteriores

#### **index.html**

Página HTML que integra el código JavaScript, proporcionando:

- Interfaz visual con botones interactivos
- Referencias a funciones JavaScript mediante onclick
- Documentación en comentarios del desafío FizzBuzz

### **arrays/index.js**

Implementación personalizada de una clase Array con métodos básicos:

```
class array {
    // Métodos implementados:
    - get(index)      // Obtener elemento por índice
    - push(item)      // Agregar elemento al final
    - pop()          // Eliminar último elemento
    - methodDelete(index) // Eliminar elemento en posición específica
    - shiftIndex(index) // Reorganizar índices después de eliminar
}
```

### **Carpetas Temáticas**

<b>Carpeta</b>	<b>Contenido</b>
<b>intro/</b>	Conceptos básicos de arrays y acceso a elementos
<b>map/</b>	Transformación de elementos usando Array.prototype.map()
<b>filter/</b>	Filtrado de elementos con Array.prototype.filter()
<b>reduce/</b>	Reducción de arrays a valores únicos con Array.prototype.reduce()
<b>delete/</b>	Técnicas para eliminar y reorganizar elementos
<b>immutability/</b>	Operaciones sin modificar el array original

Carpeta	Contenido
<b>functionalProgramimng/</b>	Paradigmas de programación funcional

---

## Análisis Técnico

### Conceptos Clave Abordados

#### 1. Fundamentos de JavaScript

- Declaración y llamada de funciones
- Parámetros y argumentos
- Integración HTML-JavaScript

#### 2. Resolución de Problemas

El problema FizzBuzz es un clásico que enseña:

- Control de flujo (if/else)
- Operador módulo (%) para divisibilidad
- Iteración con bucles (for)
- Logging en consola

#### 3. Estructura de Datos Personalizada

La clase Array personalizada demuestra:

- Uso de objetos como contenedores de datos
- Gestión de índices
- Organización de métodos en una clase
- Manipulación de datos mediante referencias

#### 4. Métodos Funcionales de Array

- **map()**: Transformación de datos (1:1)
- **filter()**: Selección condicional de elementos
- **reduce()**: Agregación de datos a un único valor

## 5. Programación Funcional

- Funciones como ciudadanos de primera clase
  - Callbacks y funciones de orden superior
  - Evitar mutación de estado
- 

### Puntos de Aprendizaje Importantes

1. **El problema FizzBuzz** es un ejercicio fundamental para:

- Evaluar lógica de programación
- Practicar condicionales
- Entender iteración

2. **Crear un Array personalizado** muestra:

- Cómo funcionan internamente las estructuras de datos
- La importancia de la abstracción
- Complejidad de operaciones (push: O(1), delete: O(n))

3. **Métodos funcionales** permiten:

- Código más legible y declarativo
- Evitar mutación de datos
- Composición de operaciones

4. **Inmutabilidad** es crítica para:

- Predictibilidad del código
  - Debugging más fácil
  - Funciones puras y testables
- 

### Conclusión

Semana 1 establece las bases para el aprendizaje de estructuras de datos mediante:

- Introducción clara a JavaScript

- Implementación manual de estructuras comunes
- Exposición a paradigmas modernos (funcional, inmutable)
- Problemas prácticos y ejercicios interactivos

Este material es ideal para principiantes que desean comprender **cómo funcionan las estructuras de datos desde cero** antes de usar librerías y APIs de nivel superior.

## Análisis - Semana 2: HTML5, Navegación y Árbol Binario de Búsqueda

### Descripción General

Esta carpeta contiene material sobre **HTML5 fundamental, navegación entre páginas web** y una introducción a **estructuras de datos no lineales** como el **Árbol Binario de Búsqueda (BST)**. Representa la transición del curso desde JavaScript puro hacia aplicaciones web más complejas con interfaces HTML y estructuras de datos avanzadas.

---

### Lenguaje

- **HTML5**
- **JavaScript (ES6+)**
- **CSS3** (con Bootstrap 5.3.2)

El material combina HTML5 vanilla con JavaScript orientado a objetos, introduciendo también un framework CSS (Bootstrap) para diseño responsivo.

---

### Estructura de Archivos

Semana2/

```
|—— index.html          # Página principal - Tutorial HTML5
|—— acerca.html        # Página "Acerca de" - Ejemplo de navegación
|—— nosotros.html       # Página "Nosotros" - Ejemplo de navegación
```

```
|— HTML inicia con la etiqueta !DOCTYPE # Documento de referencia  
|— assets/          # Recursos estáticos (imágenes, etc.)  
|— binarySearchTree/    # Implementación de Árbol Binario de Búsqueda  
|   |— index.html      # Página HTML para BST  
|   |— main.js         # Clase Node y BinarySearchTree  
|— misitio/          # Proyecto de sitio web con Bootstrap  
|   |— index.html      # Página principal con Bootstrap  
|   |— code.js          # Funciones JavaScript para interactividad  
|— tarea/            # Tarea práctica sobre Rock Inglés  
|   |— index.html      # Página principal del tema  
|   |— conciertos.html # Página de conciertos  
|   |— fotos.html       # Página de fotos
```

---

## Objetivos

1. **Dominar HTML5:** Entender la estructura semántica completa de un documento HTML
  2. **Navegación web:** Crear sitios multi-página con enlaces internos efectivos
  3. **Elementos HTML básicos:** Trabajar con headings, párrafos, listas y formatos
  4. **Introducción a Bootstrap:** Usar frameworks CSS para diseño responsive
  5. **Árbol Binario de Búsqueda (BST):** Implementar una estructura de datos no lineal fundamental
  6. **Operaciones en BST:** Insert, search y recorridos (in-order, pre-order, post-order)
- 

## Descripción Detallada

### Archivos Principales

## **index.html (Tutorial HTML5)**

Página completa que demuestra elementos HTML fundamentales:

- **Headings:** Uso de <h1> a <h6> para jerarquía de títulos
- **Párrafos y saltos:** Etiquetas <p>, <br/> para estructura de contenido
- **Listas:**
  - Ordenadas (<ol>) para listas numeradas
  - Desordenadas (<ul>) para listas con viñetas
- **Formato preformateado:** <pre> para preservar espacios y saltos
- **Otras etiquetas:** Ejemplos de enlaces, imágenes, tablas y formatos (bold, italic, etc.)

## **acerca.html y nosotros.html**

Ejemplos de navegación básica con:

- Estructura completa de página HTML
- Menú de navegación con enlaces internos <a href="...">
- Meta tags para responsividad

## **HTML inicia con la etiqueta !DOCTYP.txt**

Documento de referencia sobre:

- Declaración DOCTYPE para HTML5
- Importancia de meta tags (charset, viewport)
- Estructura básica de un documento HTML válido

## **Carpeta binarySearchTree/**

### **main.js - Implementación Completa de BST**

#### **Clase Node:**

```
class Node {  
    constructor(value) {  
        this.value = value;  
    }  
}
```

```
this.left = null;  
this.right = null;  
}  
}
```

### Clase BinarySearchTree:

1. **insert(value)**: Inserta un nodo en la posición correcta
  - o Compara el valor con el nodo actual
  - o Navega hacia la izquierda si es menor
  - o Navega hacia la derecha si es mayor
  - o Complejidad:  $O(\log n)$  promedio,  $O(n)$  peor caso
2. **search(value)**: Busca un nodo por valor
  - o Retorna el nodo si lo encuentra
  - o Retorna false si no existe
  - o Complejidad:  $O(\log n)$  promedio
3. **Recorridos**:
  - o **showInOrder()**: Left → Root → Right (orden ascendente)
  - o **showInPreOrder()**: Root → Left → Right
  - o **showInPostOrder()**: Left → Right → Root (implícito en el código)

### index.html

Estructura básica para conectar la lógica de JavaScript:

```
<script defer src='main.js'></script>
```

### Carpeta misitio/

#### index.html - Proyecto Web con Bootstrap

Demuestra:

- Integración de Bootstrap 5.3.2 CDN
- Estructura responsiva con grid system

- Componentes: Header, navbar colapsable, secciones
- Uso de clases Bootstrap para estilos predefinidos
- Colores temáticos (#212429 para fondo oscuro)

### code.js

Contiene función botones() para interactividad (estructura base).

### Carpeta tarea/

#### Tarea Práctica: Historia del Rock Inglés

Archivos de ejemplo mostrando:

- Contenido educativo sobre rock inglés
  - **Listas no ordenadas:** Bandas iconicas (Beatles, Rolling Stones, Led Zeppelin, etc.)
  - **Listas ordenadas:** Álbumes esenciales con ranking
  - Estructura navegable entre index.html, conciertos.html, fotos.html
  - Buena práctica de organización de contenido temático
- 

### Análisis Técnico

#### 1. Conceptos de HTML5

##### Estructura Semántica

- DOCTYPE declaration: <!DOCTYPE html> (HTML5 simplificado)
- Meta tags críticos: charset, viewport para responsividad
- Headings para jerarquía de contenido (h1 > h6)

##### Elementos de Contenido

- <p>: Párrafos con espaciado automático
- <pre>: Preserva formato, útil para código
- <br/>: Salto de línea explícito
- <ol>/<ul>: Listas ordenadas/desordenadas

- <a>: Enlaces internos y externos
- <nav>: Navegación semántica

## 2. Navegación Web Multi-página

El patrón implementado:

```
<nav>
  <ul>
    <li><a href="index.html">Inicio</a></li>
    <li><a href="nosotros.html">Nosotros</a></li>
    <li><a href="acerca.html">Acerca de</a></li>
  </ul>
</nav>
```

Ventajas:

- Navegación consistente entre páginas
- Rutas relativas facilitan mantenimiento
- Estructura escalable para sitios más grandes

## 3. Árbol Binario de Búsqueda (BST)

### Propiedad Fundamental

Para cada nodo:

- Todos los nodos en el **subtree izquierdo** < valor del nodo
- Todos los nodos en el **subtree derecho** > valor del nodo

### Complejidad de Operaciones

Operación	Mejor Caso	Peor Caso	Promedio
Insert	$O(\log n)$	$O(n)$	$O(\log n)$
Search	$O(\log n)$	$O(n)$	$O(\log n)$

Operación	Mejor Caso	Peor Caso	Promedio
Delete	$O(\log n)$	$O(n)$	$O(\log n)$

El peor caso ocurre cuando el árbol es degenerado (lista enlazada).

## Recorridos

1. **In-Order:** Left → Root → Right
  - Produce elementos en orden ascendente
  - Útil para obtener datos ordenados
2. **Pre-Order:** Root → Left → Right
  - Procesa el nodo antes de sus hijos
  - Útil para copiar el árbol
3. **Post-Order:** Left → Right → Root
  - Procesa el nodo después de sus hijos
  - Útil para liberar memoria

## 4. Bootstrap 5.3.2

Framework CSS que proporciona:

- **Grid System:** Layouts responsivos con filas y columnas
- **Componentes:** Navbars, cards, buttons, etc.
- **Utilidades:** Clases para spacing, colores, tipografía
- **Responsividad:** Breakpoints para diferentes dispositivos

Ventajas:

- Desarrollo rápido sin CSS custom
- Consistencia visual
- Reducción de código redundante

## 5. Organización Pedagógica

La carpeta Semana 2 es un puente entre:

- **Conceptos básicos:** HTML5 y navegación
  - **Estructuras avanzadas:** BST
  - **Herramientas modernas:** Bootstrap
  - **Aplicaciones prácticas:** Proyectos temáticos (Rock, etc.)
- 

### Puntos de Aprendizaje Importantes

1. **HTML5 es la base:** Entender estructura semántica es crítico para accesibilidad y SEO
  2. **Navegación escalable:** El patrón multi-página facilita mantener sitios grandes
  3. **BST es fundamental:** Muchas estructuras avanzadas (AVL, Red-Black Trees) se construyen sobre BST
  4. **Recorridos recursivos:** Demuestran la elegancia de algoritmos recursivos vs iterativos
  5. **Frameworks CSS aceleren desarrollo:** Bootstrap ejemplifica cómo abstracciones facilitan la creación web
  6. **Proyectos prácticos refuerzan:** La tarea del Rock Inglés conecta teoría con aplicación real
- 

### Conclusión

Semana 2 es transicional y fundamental porque:

- Consolida HTML5:** Desde estructura básica hasta navegación compleja
- Introduce BST:** Primera estructura de datos no lineal, esencial para algoritmos
- Integra herramientas modernas:** Bootstrap prepara para desarrollo web profesional
- Aplica conocimiento:** Proyectos prácticos (Rock, sitios web) muestran utilidad real
- Sienta bases:** Prepara para semanas posteriores sobre estructuras más complejas (Grafos, Árboles N-arios, etc.)

Este material es ideal para estudiantes que ya conocen HTML básico y quieren profundizar en aplicaciones web reales y estructuras de datos fundamentales.

## Análisis - Semana 3: Bootstrap Avanzado, Listas Dblemente Enlazadas y Grafos

### Descripción General

Semana 3 profundiza en **Bootstrap 5 avanzado** con énfasis en **sistemas de grid responsivo** y presenta dos estructuras de datos fundamentales: **Listas Dblemente Enlazadas (Doubly Linked Lists)** y **Grafos (Graphs)**. Este material marca un salto significativo en complejidad de estructuras de datos no lineales.

---

### Lenguaje

- **HTML5**
- **CSS3** (Bootstrap 5 + Media Queries)
- **JavaScript (ES6+)** con Programación Orientada a Objetos

Se introduce por primera vez el uso extensivo de **media queries** para crear diseños verdaderamente responsivos.

---

### Estructura de Archivos

Semana3/

```
|—— index.html          # Tutorial Bootstrap Grid System
|—— mistyle.css         # Estilos responsivos con media queries
|—— doublyLinkedList/
|   |—— index.html      # Interfaz para navegación de películas
|   |—— main.js          # Clase DoublyLinkedList con métodos
```

```
| └─ images/          # Recursos de imágenes  
└─ graphs/          # Introducción a teoría de grafos  
   ├─ graphs.js      # Representaciones de grafos (edge list, adjacency  
   |   list/matrix)  
   └─ buildGraph.js  # Clase Graph con métodos básicos
```

---

## Objetivos

1. **Dominar Bootstrap Grid System:** Crear layouts responsivos con contenedores, filas y columnas
  2. **Media Queries avanzadas:** Implementar diseños que se adapten a múltiples breakpoints
  3. **Listas Dblemente Enlazadas:** Entender nodos con referencias bidireccionales (next y prev)
  4. **Operaciones en DLL:** Implementar add, delete, reverse, show y clear
  5. **Introducción a Grafos:** Comprender vértices, aristas y diferentes representaciones
  6. **Estructuras de Grafos:** Implementar lista de adyacencia y matriz de adyacencia
- 

## Descripción Detallada

### Archivos Principales

**index.html - Tutorial Completo de Bootstrap Grid**

Demuestra **6 tipos de contenedores Bootstrap:**

1. **container:**

- o 100% wide con breakpoints
- o Máximo ancho predefinido en cada breakpoint
- o Uso: contenedores tradicionales

2. **container-fluid:**

- 100% del ancho siempre
- Sin restricción de max-width
- Uso: layouts de ancho completo

### 3. **container-sm, container-md, container-lg, container-xl, container-xxl:**

- 100% wide hasta el breakpoint especificado
- Ejemplos:
  - container-sm:  $\geq 576\text{px}$  (small)
  - container-md:  $\geq 768\text{px}$  (medium)
  - container-lg:  $\geq 992\text{px}$  (large)
  - container-xl:  $\geq 1200\text{px}$  (extra large)
  - container-xxl:  $\geq 1400\text{px}$  (extra extra large)

### **Sistema Grid Row & Col:**

```
<div class="container">
  <div class="row">
    <div class="col">Columna 1</div>
    <div class="col">Columna 2</div>
  </div>
</div>
```

Características:

- **Filas (row):** Contenedores de 12 columnas
- **Columnas (col):** Se distribuyen automáticamente
- **col-sm-6, col-md-4:** Especificar tamaño en breakpoints

### **mistyle.css - Media Queries Responsivas**

Implementa cambios de color según el tamaño de pantalla:

```
/* xs (por defecto, <576px) - verde */
.responsive-bg { background-color: var(--bs-success); }
```

```
/* sm (≥576px) - azul primario */  
@media (min-width: 576px) { ... }
```

```
/* md (≥768px) - gris secundario */  
@media (min-width: 768px) { ... }
```

```
/* lg (≥992px) - rojo peligro */  
@media (min-width: 992px) { ... }
```

```
/* xl (≥1200px) - amarillo alerta */  
@media (min-width: 1200px) { ... }
```

```
/* xxl (≥1400px) - azul info */  
@media (min-width: 1400px) { ... }
```

#### **Breakpoints de Bootstrap 5:**

<b>Breakpoint</b>	<b>Dispositivo</b>	<b>Min-Width</b>
xs	Móvil pequeño	< 576px
sm	Móvil	≥ 576px
md	Tablet	≥ 768px
lg	Desktop pequeño	≥ 992px
xl	Desktop	≥ 1200px
xxl	Desktop grande	≥ 1400px

## Carpeta doublyLinkedList/

### main.js - Implementación de Lista Dblemente Enlazada

#### Clase Node:

```
class Node {  
  
    this.value = value; // Dato almacenado  
  
    this.next = null; // Referencia al siguiente nodo  
  
    this.prev = null; // Referencia al nodo anterior  
  
}
```

#### Clase DoublyLinkedList:

Método	Descripción	Complejidad
add(value)	Agregar elemento al final	O(1)
show()	Mostrar lista hacia adelante	O(n)
reverse()	Mostrar lista hacia atrás	O(n)
clear()	Limpiar toda la lista	O(1)
delete(value)	Eliminar nodo por valor	O(n)

#### Características principales:

1. **Head y Tail:** Apuntadores al inicio y fin
2. **Bidireccionalidad:** Navegación adelante y atrás
3. **Búsqueda optimizada:** Puede iniciar desde cualquier extremo
4. **Eliminación eficiente:** No requiere reorganizar índices

#### Ventajas sobre Singly Linked List:

- Navegación en ambas direcciones
- Búsqueda desde ambos extremos
- Inserción/eliminación más eficiente

- Usado en navegadores (historial back/forward)

### **index.html - Interfaz de Navegación de Películas**

Interfaz simple que demuestra uso práctico de DLL:

```
<button onclick="prevMovie();">Prev</button>
<button onclick="nextMovie();">Next</button>
<h4 id="title"></h4>
<img id="image" style="width: 100px;">
```

**Caso de uso:** Navegar entre películas con botones Anterior/Siguiente, típicamente usando DLL internamente.

### **Carpeta graphs/**

#### **graphs.js - Representaciones de Grafos**

Muestra 4 formas de representar el mismo grafo:

##### **1. Edge List (Lista de Aristas):**

```
const graph = [
  [0, 2],
  [2, 3],
  [2, 1],
  [1, 2]
];
```

- Simple, pero ineficiente para búsquedas
- Complejidad de búsqueda:  $O(E)$  donde  $E$  = aristas

##### **2. Adjacency List (Lista de Adyacencia) - Array:**

```
const graph = [[2], [2, 3], [0, 1, 3], [1, 2]];
```

- Índice = vértice, valor = lista de vecinos
- Complejidad de búsqueda:  $O(V + E)$

##### **3. Adjacency List - Objeto:**

```

const graph = {
    0: [2],
    1: [2, 3],
    2: [0, 1, 3],
    3: [1, 2]
};

```

- Más flexible, permite claves de cualquier tipo
- Complejidad:  $O(1)$  acceso a vértice

#### 4. Adjacency Matrix (Matriz de Adyacencia):

```

const graph = [
    [0, 0, 1, 0],
    [0, 0, 1, 1],
    [1, 1, 0, 1],
    [0, 1, 1, 0]
];

```

- $graph[i][j] = 1$  si existe arista entre i y j
- Complejidad de búsqueda:  $O(1)$
- Usa más memoria para grafos dispersos

#### Comparación de Representaciones:

Representación	Búsqueda de Arista	Espacio	Uso
Edge List	$O(E)$	$O(E)$	Algoritmos complejos
Adj. List (Array)	$O(V+E)$	$O(V+E)$	Grafos típicos
Adj. List (Objeto)	$O(1)$ promedio	$O(V+E)$	Grafos dinámicos
Adj. Matrix	$O(1)$	$O(V^2)$	Grafos densos

## **buildGraph.js - Clase Graph**

Implementa un grafo no dirigido usando lista de adyacencia:

```
class graph {  
  
    constructor() {  
  
        this.nodes = 0;      // Contador de vértices  
  
        this.adjacentList = {}; // Lista de adyacencia  
  
    }  
  
    addVertex(node) {  
  
        this.adjacentList[node] = [];  
  
        this.nodes++;  
  
    }  
  
    addEdge(node1, node2){  
  
        this.adjacentList[node1].push(node2);  
  
        this.adjacentList[node2].push(node1); // No dirigido  
  
    }  
}
```

### **Ejemplo de uso:**

```
const myGraph = new graph();  
  
myGraph.addVertex("1");  
  
myGraph.addVertex("3");  
  
myGraph.addVertex("4");  
  
myGraph.addVertex("5");  
  
myGraph.addVertex("6");  
  
myGraph.addVertex("8");
```

```
myGraph.addEdge("1", "6");
myGraph.addEdge("6", "3");
myGraph.addEdge("3", "5");
myGraph.addEdge("4", "5");
// ... más aristas
```

---

## Análisis Técnico

### 1. Bootstrap Grid System Responsivo

**Concepto Core:** Bootstrap divide el ancho en 12 columnas. Las clases col-\* especifican cuántas columnas ocupa cada elemento.

#### Ejemplo práctico:

```
<div class="row">
  <div class="col-md-4">33.33% en MD+</div>
  <div class="col-md-8">66.66% en MD+</div>
</div>
```

#### Ventajas:

- Mobile-first approach
- Breakpoints predefinidos
- Proporcional y flexible
- Facilita mantenimiento

### 2. Media Queries y Diseño Responsivo

Los media queries permiten aplicar estilos según condiciones:

```
/* Móvil (por defecto) */
```

```
.box { font-size: 12px; }
```

```
/* Tablet y superior */
```

```
@media (min-width: 768px) {  
    .box { font-size: 14px; }  
}  
  
/* Desktop y superior */
```

```
@media (min-width: 992px) {  
    .box { font-size: 16px; }  
}
```

### Filosofía Mobile-First:

1. Definir estilos base para móvil
2. Usar min-width para dispositivos más grandes
3. Progresiva mejora de experiencia

### 3. Listas Dblemente Enlazadas vs Simplemente Enlazadas

Característica	Singly	Doubly
Navegación	Adelante	Ambas direcciones
Memoria por nodo	1 puntero	2 punteros
Búsqueda	$O(n)$ desde inicio	$O(n/2)$ promedio
Eliminación	Requiere nodo anterior	Direct
Casos de uso	Pilas, colas	Navegadores, editores

### Complejidad de operaciones en DLL:

- Insert al inicio:  $O(1)$
- Insert al final:  $O(1)$
- Delete:  $O(n)$  búsqueda +  $O(1)$  eliminación
- Reverse iteration:  $O(n)$

## 4. Teoría de Grafos Fundamental

### Terminología:

- **Vértice (Nodo):** Punto en el grafo
- **Arista (Edge):** Conexión entre dos vértices
- **Grado:** Número de aristas conectadas a un vértice
- **Grafo Dirigido:** Las aristas tienen dirección
- **Grafo No Dirigido:** Las aristas son bidireccionales

### Aplicaciones Reales:

- Redes sociales (amigos = aristas)
- Navegación GPS (ciudades = vértices, caminos = aristas)
- Recomendaciones (usuarios → productos)
- Sistemas de transportes (estaciones = vértices)

## 5. Selección de Representación de Grafo

### Criterios:

- **Grafo disperso** (pocas aristas): Lista de adyacencia
- **Grafo denso** (muchas aristas): Matriz de adyacencia
- **Búsquedas frecuentes:** Matriz de adyacencia  $O(1)$
- **Espacio limitado:** Lista de adyacencia  $O(V+E)$

---

### 💡 Puntos de Aprendizaje Importantes

1. **Bootstrap facilita diseño responsive:** Abstacta complejidad de media queries
2. **Mobile-first es crítico:** Los dispositivos móviles son mayoría de usuarios
3. **DLL es puente entre arrays y grafos:** Introduce referencias bidireccionales
4. **Grafos son ubicuos:** La mayoría de problemas reales involucran grafos
5. **Elección de representación importa:** Afecta complejidad y uso de memoria

- 
6. **Practicalidad de DLL:** Aparecen en navegadores, editores de texto, reproductor multimedia
- 

## Conclusión

Semana 3 es **transicional y reveladora** porque:

- Consolida Bootstrap:** Desde grid básico hasta responsividad completa
- Introduce grafos:** Estructura fundamental para problemas complejos
- Conecta conceptos:** DLL prepara para traversals de grafos
- Expande horizonte:** Desde datos lineales a no lineales
- Aplicaciones prácticas:** Media queries, navegación, sistemas de recomendación

Este material es ideal para estudiantes preparándose para **algoritmos avanzados** (BFS, DFS, Dijkstra) que dependen de estructuras de grafos sólidas. La combinación de Bootstrap responsive y estructuras de datos crea una base para desarrollo web moderno.

## Análisis - Semana 4: Tipos de Datos Primitivos y Grafos Dirigidos

### Descripción General

Semana 4 combina **tipos de datos primitivos en JavaScript** (números, literales, constructores) con una **implementación avanzada de grafos dirigidos (Directed Graphs)**. Marca un punto de inflexión donde se profundiza en programación orientada a objetos aplicada a estructuras de datos complejas y se consolida la comprensión de números en JavaScript.

---

### Lenguaje

- **HTML5**
- **JavaScript (ES6+)** con enfoque en:

- Tipos primitivos (number)
  - Clases modernas
  - Estructuras de datos (Map, Arrays)
  - Programación Orientada a Objetos (POO)
- 

## Estructura de Archivos

Semana4/

```
|--- index.html      # Tutorial sobre tipos de datos numéricos  
|--- code.js        # Funciones con números y operaciones básicas  
└--- graph/  
    |--- index.html  # Implementación de grafo dirigido  
    |--- main.js     # Clases Graph y Node para grafos dirigidos
```

---

## Objetivos

1. **Dominar tipos numéricos en JavaScript:** Entender números primitivos vs objetos Number
  2. **Usar notación legible de números:** Aplicar separadores visuales (5\_000\_000)
  3. **Operaciones aritméticas:** Crear funciones para sumas, multiplicaciones y tablas
  4. **Grafos dirigidos:** Implementar grafos donde las aristas tienen dirección
  5. **Clases Node y Graph:** Crear estructuras orientadas a objetos para grafos
  6. **Traversal de grafos:** Mostrar nodos y sus conexiones dirigidas
- 

## Descripción Detallada

### Archivos Principales

**index.html - Tutorial de Tipos de Datos Numéricos**

Página educativa que explica:

### Tipos de datos numéricos en JavaScript:

1. **Números primitivos** (preferido):
  2. const number = 80;
  3. const decimal = 15.8;
    - o Creados literalmente
    - o Tipo primitivo eficiente
    - o Recomendado para uso general
4. **Objeto Number** (usando constructor):
  5. const num = new Number(42);
    - o Crea un objeto wrapper
    - o Menos eficiente
    - o Útil en casos específicos

### Notación Legible:

```
const legibleNumber = 5_000_000; // Equivalente a 5000000
```

- Separadores visuales (\_) para mejorar legibilidad
- No afecta el valor numérico
- Útil para números grandes

### Tabla HTML de referencia:

```
<table border="1">

<tr>
  <th>Constructor</th>
  <th>Descripción</th>
</tr>

<tr>
  <td>new Number(number)</td>
```

<td>Crea un objeto numérico a partir del número number</td>	
</tr>	
<tr>	
<td>&lt;td&gt;number&lt;/td&gt;</td>	<td>number</td>
<td>&lt;td&gt;Simplemente, el número en cuestión. Notación preferida.&lt;/td&gt;</td>	<td>Simplemente, el número en cuestión. Notación preferida.</td>
</tr>	
</table>	

## **code.js - Funciones Numéricas Básicas**

### **Variables globales:**

```
const number = 80;      // Variable numérica constante
const decimal = 15.8;   // Número decimal
const legibleNumber = 5_000_000; // Número con separador visual
```

### **Funciones implementadas:**

#### **1. `saldarNumber()`**

- Itera desde 1,000,000 hasta 5,000,000 (de 1M en 1M)
- Imprime "Saludo número: X" para cada iteración
- Demuestra: loops, números grandes, legibilidad

#### **2. `entornoSuma(num)`**

- Suma un parámetro con la variable global number (80)
- Retorna el resultado con console.log()
- Ejemplo: entornoSuma(20) → 100

#### **3. `tablaMultiplicar(num)`**

- Genera tabla de multiplicación de 1 a 10
- Itera ascendente (i = 1 a 10)
- Formato: "num x i = resultado"

#### **4. `tablaMultiplicarM(num)`**

- Similar a tablaMultiplicar() pero orden inverso
- Itera descendente ( $i = 10 \text{ a } 1$ )
- Demuestra: loops invertidos

### **Ejemplo de salida:**

$5 \times 1 = 5$

$5 \times 2 = 10$

$5 \times 3 = 15$

...

$5 \times 10 = 50$

### **Carpeta graph/**

#### **main.js - Implementación de Grafo Dirigido**

##### **Clase Node:**

```
class Node {
  constructor(value) {
    this.value = value; // Valor del nodo
    this.edges = []; // Array de nodos conectados
  }

  addEdge(node) {
    this.edges.push(node); // Agregar conexión saliente
  }
}
```

##### **Clase Graph:**

```
class Graph {
  constructor() {
    this.nodes = new Map(); // Map para almacenar nodos por valor
```

```

    }

addNode(value) {
    const node = new Node(value);
    this.nodes.set(value, node); // Almacenar por clave (value)
}

addEdge(startValueNode, endValueNode) {
    const startNode = this.nodes.get(startValueNode);
    const endNode = this.nodes.get(endValueNode);

    if (startNode && endNode) {
        startNode.addEdge(endNode); // DIRIGIDO: solo A→B, no B→A
    }
}

show() {
    for(const node of this.nodes.values()) {
        const edges = node.edges.map(edge => edge.value).join(', ');
        console.log(` ${node.value} -> ${edges}`);
    }
}

```

**Ejemplo de uso:**

```
const graph = new Graph();
```

```

// Agregar nodos
graph.addNode('A');
graph.addNode('B');
graph.addNode('C');
graph.addNode('D');

// Agregar aristas dirigidas (unidireccionales)
graph.addEdge('A', 'B'); // A → B
graph.addEdge('A', 'D'); // A → D
graph.addEdge('B', 'C'); // B → C
graph.addEdge('B', 'D'); // B → D
graph.addEdge('C', 'D'); // C → D
graph.addEdge('D', 'A'); // D → A

graph.show();

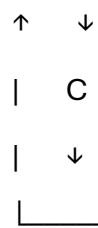
```

**Salida esperada:**

A -> B, D  
 B -> C, D  
 C -> D  
 D -> A

**Visualización del grafo:**

A ----→ B



---

## Análisis Técnico

### 1. Tipos Numéricos en JavaScript

#### Primitivo vs Objeto

Aspecto	Primitivo	Objeto
Creación	<code>let n = 42;</code>	<code>let n = new Number(42);</code>
Tipo	<code>number</code>	<code>object</code>
Eficiencia	 Alta	 Baja
Memoria	Menos	Más
Métodos	Acceso automático	Directo
Comparación	<code>5 === 5 → true</code>	<code>new Number(5) === new Number(5) → false</code>

**Recomendación:** Usar primitivos siempre. Los wrappers Object son legado de JavaScript antiguo.

#### Operaciones Numéricas

```
// Operaciones básicas  
const a = 10;  
const b = 3;  
  
console.log(a + b); // 13 (suma)  
console.log(a - b); // 7 (resta)  
console.log(a * b); // 30 (multiplicación)  
console.log(a / b); // 3.33... (división)  
console.log(a % b); // 1 (módulo)
```

```
console.log(a ** b); // 1000 (exponenciación)
```

**Complejidad temporal:** O(1) para todas las operaciones aritméticas básicas.

## 2. Separador Visual de Números (Numeric Separator)

### Característica ES2021:

// Legible

```
const million = 1_000_000;  
const billion = 1_000_000_000;  
const hex = 0xFF_FF_FF_FF;  
const binary = 0b1111_0000;
```

// Es equivalente a:

```
const million = 1000000;  
const billion = 1000000000;
```

### Ventajas:

- Mejora legibilidad en números grandes
- No afecta el valor
- Funciona en hex, binary, octal
- Ampliamente soportado en navegadores modernos

## 3. Grafos Dirigidos vs No Dirigidos

### Grafo No Dirigido (Semana 3)

```
addEdge(node1, node2){  
    this.adjacentList[node1].push(node2);  
    this.adjacentList[node2].push(node1); // Bidireccional  
}
```

Relación: A ↔ B (amistad en redes sociales)

### Grafo Dirigido (Semana 4)

```

addEdge(startValueNode, endValueNode) {
    const startNode = this.nodes.get(startValueNode);
    const endNode = this.nodes.get(endValueNode);
    startNode.addEdge(endNode); // Unidireccional
}

```

Relación: A → B (seguir en Twitter)

### **Aplicaciones de Grafos Dirigidos:**

- Redes sociales (seguidor → seguido)
- Sistemas de recomendación
- Flujos de trabajo
- Análisis de dependencias
- Compiladores (DAG - Directed Acyclic Graph)

## **4. Uso de Map en lugar de Objetos**

La clase Graph utiliza Map en lugar de objetos planos:

// Opción 1: Objeto plano

```
this.nodes = {};
```

// Opción 2: Map (utilizado)

```
this.nodes = new Map();
```

### **Comparación:**

Característica	Objeto {}	Map
Claves	Solo strings	Cualquier tipo
Iteración	for...in, Object.keys()	for...of, .values()
.size	Manual	Propiedad .size

Característica	Objeto {}	Map
Rendimiento	O(1)	O(1)
Métodos	Limitados	.has(), .delete(), .clear()
Prototipo	Hereda <b>proto</b>	Limpio

### Ventaja de Map aquí:

- Métodos más intuitivos (.get(), .set())
- Mejor rendimiento en iteraciones
- Más seguro (sin colisiones con propiedades proto)

## 5. Complejidad de Operaciones en Grafo Dirigido

Operación	Complejidad	Descripción
addNode(value)	O(1)	Insertar en Map
addEdge(u, v)	O(1)	Agregar a array edges
show()	O(V + E)	Visitar cada nodo y arista
Buscar si existe arista	O(grado del nodo)	Buscar en array edges

Donde:

- V = número de vértices (nodos)
- E = número de aristas (conexiones)

## 6. Patrones de Diseño Aplicados

### Builder Pattern (implícito):

```
const graph = new Graph();
graph.addNode('A');
graph.addNode('B');
```

```
graph.addEdge('A', 'B');  
// Construcción paso a paso
```

**Observer Pattern (potencial):** La estructura permite fácilmente agregar métodos como:

- `onNodeAdded(callback)`
  - `onEdgeAdded(callback)`
- 

### Puntos de Aprendizaje Importantes

1. **Números primitivos son eficientes:** Evitar wrappers Object innecesarios
  2. **Legibilidad importa:** Los separadores visuales hacen código más mantenible
  3. **Grafos dirigidos son comunes:** La mayoría de aplicaciones reales usan direcciónalidad
  4. **Map > Objetos para estructuras:** Especialmente en estructura de datos complejas
  5. **POO facilita mantenimiento:** Clases Node y Graph encapsulan comportamiento
  6. **Diferencia conceptual crítica:** Dirigido vs No-dirigido cambia aplicaciones completamente
- 

### Conclusión

Semana 4 es **fundamental en transición** porque:

- Consolida JavaScript básico:** Domina números y sus operaciones
- Introduce grafos dirigidos:** Extensión crucial de conceptos de Semana 3
- Mejora POO:** Implementación más sofisticada de clases
- Prepara para algoritmos:** BFS, DFS, topological sort dependen de grafos dirigidos
- Aplicaciones reales:** Redes sociales, workflows, compiladores usan esto
- Mejor performance:** Uso de Map vs objetos, números primitivos

Este material es ideal para estudiantes que dominan arrays y listas, y que están listos para estructuras no lineales más sofisticadas. La combinación de tipos primitivos fundamentales con grafos dirigidos avanzados crea un equilibrio entre lo básico y lo complejo, preparando para semanas posteriores sobre algoritmos de graph traversal (BFS, DFS) y problemas clásicos de grafos (camino más corto, componentes conectados, etc.).

## Análisis - Semana 5: JavaScript, JSON y Listas Enlazadas

### Descripción General

La carpeta Semana5 reúne ejercicios y ejemplos prácticos sobre JavaScript básico aplicado al DOM, uso de archivos JSON para datos, y una introducción a **listas enlazadas** (linked lists). El material está orientado a consolidar conocimientos de programación imperativa y estructuras de datos lineales.

---

### Lenguaje

- **JavaScript (ES6+)**
  - **HTML5**
  - **CSS3 (básico)**
  - **JSON** (para datos estáticos)
- 

### Estructura de Archivos

Semana5/

```
|--- index.html      # Página principal con ejemplos HTML + scripts  
|--- code.js        # Scripts de ejercicio (funciones y ejemplos)  
|--- inner.html     # Página adicional / ejemplo  
|--- introduccion/  # Material introductorio  
|   |--- funtions/    # Ejemplos de funciones
```

```
| └─variable/      # Conceptos de variables
|   └─json/
|     |   └─index.html    # Ejemplo de consumo de JSON en la página
|     |   └─main.js       # Código que carga y procesa JSON
|     |   └─people.json   # Datos de ejemplo en formato JSON
|   └─linkedList/
|     |   └─index.html    # Interfaz de demostración de linked list
|     |   └─main.js       # Implementación y métodos de la lista enlazada
|     └─images/         # Recursos multimedia
```

---

## Objetivos

1. Entender integración básica entre HTML y JavaScript.
  2. Aprender a cargar y procesar datos en formato JSON desde el cliente.
  3. Implementar operaciones básicas en una Linked List (agregar, mostrar, eliminar).
  4. Practicar funciones, scopes y flujo de control en JavaScript.
  5. Presentar ejemplos interactivos que refuerzen la teoría.
- 

## Descripción Detallada

### **index.html y code.js**

- Contienen ejemplos de elementos HTML enlazados a funciones en code.js.
- Muestran uso de eventos (onclick) y manipulación simple del DOM.
- code.js incluye funciones para operaciones básicas (bucles, salidas en consola, demostraciones).

### **Carpeta json/**

- people.json provee datos de ejemplo (listas de objetos) para practicar lectura y renderizado dinámico.

- main.js ilustra cómo usar fetch() (o XMLHttpRequest) para cargar JSON y transformar los datos a HTML (tablas o listas).
- Útil para entender asincronía básica y promesas.

## Carpeta linkedList/

- main.js contiene la implementación de una lista enlazada simple (clase Node y clase LinkedList o funciones equivalentes).
  - Métodos típicos: add, remove, show, clear y posiblemente find.
  - index.html demuestra la interfaz para interactuar con la estructura (botones para agregar/eliminar y un área para mostrar la lista).
- 

## Breve Análisis Técnico

### 1. Integración HTML ↔ JS

- Patrón básico: incluir `<script src="code.js"></script>` y llamar funciones desde atributos onclick o añadir event listeners en JS.
- Buenas prácticas: preferir addEventListener y evitar lógica inline en HTML para separar estructura y comportamiento.

### 2. Consumo de JSON

- `fetch('people.json').then(res => res.json()).then(data => render(data))` es la forma moderna.
- Considerar manejo de errores (catch) y estados de carga.
- JSON es ideal para datos estáticos en ejemplos y pruebas locales.

### 3. Implementación de Linked List

- Nodos con value y next (singly linked list) o prev/next si es doble.
- Complejidades: add O(1) si se mantiene tail, remove O(n) en el peor caso.
- Ventajas pedagógicas: entender punteros y gestión dinámica de memoria conceptual en JS.

### 4. Calidad y mejoras sugeridas

- Modularizar código: extraer la lógica de estructuras a módulos (linkedList.js) y mantener main.js para la interacción.
  - Añadir pruebas unitarias simples (por ejemplo con Jest) para validar métodos de la lista.
  - Mejorar la UX: mostrar estados (vacío, elemento agregado) y mensajes de error.
- 

### Puntos de Aprendizaje

- JSON + fetch() prepara para APIs reales.
  - Las linked lists permiten comprender por qué algunas operaciones en arrays son costosas.
  - Separación de responsabilidades (estructura vs UI) facilita mantenimiento del código.
- 

### Conclusión

Semana5 refuerza conceptos prácticos: manipulación del DOM con JavaScript, consumo de datos en JSON y la implementación manual de una Linked List. Es una semana orientada a traducción de teoría a práctica y sienta buenas bases para estudiar estructuras de datos más avanzadas y trabajar con APIs.

## Análisis - Semana 6: map() funcional, Programación Orientada a Objetos y Colas (Queue)

### Descripción General

La carpeta Semana6 integra conceptos de programación funcional aplicados a arrays (uso de map), fundamentos de **Programación Orientada a Objetos (POO)** en JavaScript y la implementación de **colas (queues)**. La semana busca consolidar la transición entre manipulación de colecciones y diseño de estructuras/abstracciones con clases.

---

## Lenguaje

- **JavaScript (ES6+)** — énfasis en funciones de orden superior, clases y módulos sencillos.
  - **HTML5** — páginas de ejemplo que integran los scripts.
  - **CSS3 (básico)** — estilos mínimos si aplica.
- 

## Estructura de Archivos (estimada)

Semana6/

```
|—— index.html      # Página principal de ejemplos  
|—— code.js        # Ejemplos y utilidades JS  
|—— map/           # Ejemplos del método `map` y transformaciones  
|   |—— index.html / main.js  
|—— poo/           # Ejemplos de clases, constructores y herencia  
|   |—— clases/ referencia /  
|—— queue/         # Implementación de colas (enqueue, dequeue, peek)  
|   |—— index.html / main.js
```

Nota: la estructura refleja los temas principales presentes en la carpeta Semana6 (map, poo, queue).

---

## Objetivos

1. Comprender y aplicar `Array.prototype.map()` para transformar colecciones de forma declarativa.
2. Dominar conceptos básicos de POO en JavaScript: class, constructor, métodos, this y composición/herencia básica.
3. Implementar una Queue (FIFO) con operaciones enqueue, dequeue, peek y isEmpty.

4. Conectar programación funcional y POO: usar transformaciones antes/después de procesar estructuras lineales.
  5. Mejorar prácticas: separar lógica (clases/estructuras) de la interfaz (HTML), y usar addEventListener en lugar de onclick inline.
- 

## Descripción Detallada

### Carpeta map/

- Ejemplos de uso de map() para transformar arrays de datos (por ejemplo, arrays de objetos a arrays de strings o números).
- Comparación con for y forEach: map devuelve un nuevo array, es inmutable y encaja con programación funcional.
- Buenas prácticas: evitar efectos secundarios dentro del callback de map.

### Carpeta poo/

- Demostraciones de class y creación de instancias en ES6.
- Ejemplos típicos: clases Person, Product, o estructuras que encapsulan comportamiento y estado.
- Posible inclusión de herencia simple y métodos estáticos.
- Recomendaciones: mantener métodos puros cuando sea posible y evitar dependencias globales.

### Carpeta queue/

- Implementación clásica de Queue (puede ser con Array o con LinkedList internamente):
  - enqueue(item) — agregar al final
  - dequeue() — eliminar del frente y retornar elemento
  - peek() — ver el frente sin eliminar
  - isEmpty() — true si la cola está vacía
- Complejidad: enqueue y dequeue idealmente O(1) si se usa una estructura adecuada (por ejemplo, punteros head/tail en lista enlazada o un buffer circular).

- Casos de uso: gestión de tareas, colas de impresión, breadth-first search (BFS) en grafos.
- 

## Breve Análisis Técnico

### 1. map() vs Mutación

- map() es apropiado para transformar datos sin mutar el original:
- const nums = [1, 2, 3];
- const squares = nums.map(x => x \* x); // [1,4,9]
- Evitar side-effects dentro del callback para preservar claridad y testabilidad.

### 2. POO en JavaScript (clases ES6)

- class es azúcar sintáctico sobre prototipos; entender this y binding es crucial.
- Ejemplo básico:
- class Person {
- constructor(name) { this.name = name; }
- greet() { return `Hola \${this.name}`; }
- }
- Considerar separar modelos (datos) de servicios (lógica que opera esos datos).

### 3. Implementación eficiente de Queue

- Usar Array.shift() es simple pero O(n) por reasignación de índices; para colas grandes preferir:
  - Linked list con head y tail (O(1) enqueue/dequeue).
  - Buffer circular con índice head y tail (O(1) y uso de espacio limitado).

### 4. Conexión entre paradigmas

- Flujo recomendado: recibir datos (JSON/array) → transformar con map/filter → encolar tareas con Queue → procesar con clases/servicios (POO).
- Esto muestra arquitectura simple: **ingest → transform → queue → process**.

## 5. Buenas prácticas y mejoras

- Modularizar: mover Queue y clases a módulos propios (queue.js, models/\*.js).
  - Añadir tests unitarios mínimos (por ejemplo con un runner simple) para validar enqueue/dequeue.
  - Documentar ejemplos en README.md dentro de la carpeta.
- 

### Puntos de Aprendizaje

- map() refuerza enfoque declarativo y evita errores comunes de mutación.
  - POO en JavaScript facilita encapsulación y reuso de comportamiento.
  - La elección de estructura para Queue impacta rendimiento; Array.shift() es aceptable para ejemplos pequeños, no para producción.
  - Entender cómo combinar transformaciones funcionales con estructuras orientadas a objetos es clave para arquitecturas limpias.
- 

### Conclusión

Semana6 busca cerrar la brecha entre tratamiento funcional de colecciones (map) y diseño orientado a objetos/estructuras (class, Queue). La semana prepara para algoritmos que requieren transformación previa de datos y procesamiento en cola (por ejemplo, BFS, pipelines de datos, procesamiento asíncrono). Aplicar las mejoras sugeridas (modularidad, tests, implementaciones O(1) para Queue) fortalecerá la calidad del código y la escalabilidad de los ejemplos.

## Análisis - Semana 7: Tablas Hash, Sets y Pilas (Stacks)

### Descripción General

La carpeta Semana7 agrupa ejercicios y ejemplos prácticos sobre **tablas hash (hash tables)**, **conjuntos (sets)** y **pilas (stacks)**. El material combina teoría (complejidad, colisiones) con implementaciones en JavaScript y ejemplos de uso para entender por qué estas estructuras son relevantes en problemas reales.

---

## Lenguaje

- **JavaScript (ES6+)**
- **HTML5**
- **CSS3 (básico)**

Las implementaciones usan JavaScript puro, con clases o funciones para ilustrar principios de diseño y rendimiento.

---

## Estructura de Archivos

Semana7/

```
|—— index.html      # Página de demostración y pruebas  
|—— code.js        # Ejemplos y utilidades generales  
|—— hashTables/    # Implementación y ejemplos de tablas hash  
|   |—— hashTables.js (o similar)  
|—— set/           # Ejemplos y uso de conjuntos (Set)  
|   |—— index.html / main.js  
└—— stack/         # Implementación de pilas (Stack)  
   |—— stack.js / index.html
```

Nota: los nombres de archivos concretos pueden variar; la estructura refleja las subcarpetas observadas en el repositorio.

---

## Objetivos

1. Entender el concepto de **función hash** y cómo se usa para indexar datos.
2. Implementar una **Hash Table** básica con manejo de colisiones (encadenamiento o open addressing).
3. Aprender el uso y ventajas de Set en JavaScript (unión, intersección, diferencia).

4. Implementar una **Stack** (LIFO) con operaciones push, pop, peek y isEmpty.
  5. Comparar complejidades y elegir la estructura adecuada según el problema.
- 

## Descripción Detallada

### Carpeta hashTables/

- Contiene una implementación educativa de una tabla hash.
- Elementos clave a revisar:
  - **Función hash:** conversión de claves a índices numéricos.
  - **Manejo de colisiones:** chaining (listas enlazadas por bucket) o open addressing.
  - **Operaciones:** set(key, value), get(key), has(key), delete(key).
- Recomendaciones: validar la calidad del hash y probar con colisiones intencionales.

### Carpeta set/

- Muestra el uso de Set nativo de ES6 y operaciones típicas:
  - new Set(iterable) — creación
  - add, delete, has — manipulación básica
  - Conversión a Array para aplicar map/filter
- Ejemplos típicos: eliminar duplicados, cálculos de unión/intersección.

### Carpeta stack/

- Implementación de pila para demostrar LIFO.
- Métodos esperados:
  - push(item) — agregar elemento
  - pop() — eliminar y retornar último elemento
  - peek() — ver el tope sin eliminar
  - isEmpty() — estado de la pila

- Casos de uso: evaluar expresiones (postfix), backtracking, navegación (historial)
- 

## Breve Análisis Técnico

### 1. Tablas Hash

- Operaciones promedio: get, set, delete → O(1) promedio.
- Peor caso: O(n) cuando muchas claves colisionan o mal diseño del hash.
- Estrategias de colisión:
  - **Encadenamiento:** cada bucket mantiene una lista de entradas; sencillo y eficiente para cargas moderadas.
  - **Open addressing:** sondaje lineal/ cuadrático/ doble hashing; requiere manejo cuidadoso del factor de carga.
- Factor de carga (load factor) debe monitorearse y, si es necesario, redimensionar la tabla (rehashing).

### 2. Set (ES6)

- Implementación nativa optimizada; uso recomendado para operaciones de pertenencia y eliminación de duplicados.
- Operaciones: add, delete, has → O(1) promedio.
- Ejemplo: const unique = [...new Set(array)]; elimina duplicados rápidamente.

### 3. Stack (Pila)

- Implementación simple con Array (push/pop) ofrece O(1) para operaciones de tope.
- Alternativa con lista enlazada cuando se desea control explícito de nodos y O(1) garantiza sin reasignaciones.
- Uso en algoritmos: evaluación de expresiones, recorridos recursivos simulados, undo/redo.

### 4. Casos de uso y recomendaciones

- Usar **Hash Table** para índices rápidos por clave (caches, tablas de símbolos, conteo de frecuencia).

- Usar **Set** para limpiar datos y operaciones de teoría de conjuntos.
  - Usar **Stack** donde el orden LIFO es necesario (paréntesis balanceados, DFS iterativo).
- 

### Puntos de Aprendizaje

- Las tablas hash ofrecen acceso muy rápido en la práctica, pero requieren buen manejo de colisiones y rehashing.
  - Set es una herramienta poderosa de ES6 para simplificar operaciones comunes con colecciones únicas.
  - Elegir Array vs LinkedList para Stack depende de la necesidad de rendimiento en escenarios extremos.
  - Es valioso complementar las implementaciones educativas con pruebas que simulen cargas y colisiones.
- 

### Conclusión

Semana7 consolida estructuras de datos esenciales para rendimiento en aplicaciones reales: hashing para acceso por clave, conjuntos para unicidad y pilas para control de flujo LIFO. Aplicar pruebas, medir el factor de carga y modularizar las implementaciones (módulos separados para hashTables, set, stack) mejorará la calidad del material y facilitará su reutilización en ejercicios posteriores.

## Análisis - Semana 8: Listas Enlazadas (Linked Lists)

### Descripción General

La carpeta Semana8 está centrada en **listas enlazadas** (principalmente singly linked lists) y ejemplos prácticos para entender nodos, punteros y operaciones dinámicas sobre colecciones. Es una semana orientada a manipulación de estructuras lineales sin índices numéricos fijos.

---

## Lenguaje

- **JavaScript (ES6+)**
  - **HTML5** (páginas de demostración)
  - **CSS3** (estilos mínimos para la interfaz)
- 

## Estructura de Archivos (estimada)

Semana8/

```
|—— linkedlist/      # Implementaciones y demos
|   |—— index.html    # Interfaz de prueba
|   |—— main.js / linkedlist.js# Clases Node y LinkedList
|   |—— images/       # Recursos (opcional)
```

---

## Objetivos

1. Entender la representación de una Node con value y next.
  2. Implementar operaciones básicas: add/append, prepend, remove, find, traverse.
  3. Analizar complejidad temporal de cada operación ( $O(1)$ ,  $O(n)$ ).
  4. Comparar LinkedList vs Array para seleccionar la estructura adecuada.
  5. Usar la lista en ejemplos prácticos (colas, recorrido, edición dinámica).
- 

## Descripción Detallada

### Implementación típica

- **Clase Node:** almacena value y next.
- **Clase LinkedList:** mantiene head, opcionalmente tail y length.
- **Métodos comunes:**
  - append(value) — agregar al final ( $O(1)$  si existe tail,  $O(n)$  si no)

- `prepend(value)` — agregar al inicio ( $O(1)$ )
- `remove(value)` — eliminar primer nodo con ese valor ( $O(n)$ )
- `find(predicate)` — buscar por condición ( $O(n)$ )
- `toArray() / fromArray()` — conversión para interoperabilidad

### **Interfaz (index.html)**

- Botones para agregar/eliminar/recorrer.
  - Área de visualización que representa nodos y flechas (ej. listas en DOM).
  - Ejemplo didáctico para mostrar cómo cambian head y next.
- 

### Breve Análisis Técnico

#### **Ventajas de LinkedList**

- Inserción/eliminación en extremos:  $O(1)$  con referencias adecuadas.
- Útil cuando las operaciones frecuentes son adiciones/eliminaciones dinámicas.
- No requiere reasignar memoria contigua.

#### **Desventajas**

- Acceso aleatorio por índice:  $O(n)$  (no es adecuado cuando se necesita acceso por índice frecuente).
- Overhead en memoria por referencias next.
- Implementaciones simples con Array pueden ser más rápidas para tamaños pequeños por optimizaciones internas.

#### **Complejidades clave**

- `append con tail: O(1)`
- `append sin tail: O(n)`
- `prepend: O(1)`
- `remove/find: O(n)`
- `toArray: O(n)`

## Buenas prácticas y mejoras

- Mantener tail y length para optimizar operaciones y simplificar tests.
  - Separar lógica de estructura (ej. linkedlist.js) de la UI (index.html, main.js).
  - Añadir pruebas unitarias simples para validar invariantes (p. ej. length, head nulo en lista vacía).
- 

### Puntos de Aprendizaje

- Las linked lists enseñan el concepto de punteros y cómo las estructuras dinámicas difieren de arrays indexados.
  - Son base para otras estructuras (stacks, queues, listas dobles) y para algoritmos que manipulan nodos.
  - Comparar siempre trade-offs: rendimiento práctico vs teórico según escenario.
- 

### Conclusión

Semana8 es esencial para comprender manejo dinámico de colecciones en memoria y para preparar temas posteriores (listas dobles, colas, y algoritmos que requieren manipulación de nodos). La implementación correcta (uso de tail, manejo de bordes) y pruebas simples harán que los ejemplos sean robustos y reutilizables.

## Análisis - Semana 9: Repaso de Estructuras Fundamentales (Arrays, Listas, Pilas, Colas, Grafos, Tablas Hash, Árboles)

### Descripción General

Semana9 reúne implementaciones y ejemplos de las estructuras de datos fundamentales: **arrays**, **listas enlazadas (singly/doubly)**, **pilas (stacks)**, **colas (queues)**, **grafos (graphs)**, **tablas hash (hash tables)** y **árboles (trees)**. Es una sesión de repaso y consolidación que compara implementaciones nativas de JavaScript con versiones educativas escritas desde cero.

---

## Lenguaje

- **JavaScript (ES6+)** — implementaciones con clases y funciones
  - **HTML5** — páginas de demostración y pruebas
  - **CSS3** — estilos mínimos (si aplica)
- 

## Estructura de Archivos (observada)

Semana9/

```
|—— arrays.js      # Ejemplos y utilidades con arrays  
|—— singly_linkedList.js  # Implementación de lista simplemente enlazada  
|—— doubly_linkedList.js  # Implementación de lista doblemente enlazada  
|—— stack.js       # Implementación de pila (LIFO)  
|—— queue.js       # Implementación de cola (FIFO)  
|—— queues/        # Ejemplos adicionales de colas  
|—— hash_table.js   # Implementación educativa de tabla hash  
|—— graph.js        # Implementación y ejemplos de grafos  
|—— tree.js         # Implementación de árbol (posible BST)  
|—— README.md       # Notas y ejemplos de la semana  
└—— LICENSE         # Licencia del repositorio
```

---

## Objetivos

1. Repasar la funcionalidad y coste de las estructuras de datos básicas.
2. Comparar implementaciones nativas (Array, Set, Map) con versiones manuales.
3. Practicar operaciones clave: inserción, eliminación, búsqueda y recorridos.
4. Entender trade-offs de memoria y tiempo entre implementaciones.

5. Conectar estructuras para resolver problemas compuestos (p. ej. BFS usando Queue).
- 

## Descripción Detallada

### **arrays.js**

- Ejemplos de manipulación de arrays: push, pop, shift, unshift, map, filter, reduce.
- Discusión sobre complejidad y cuándo emplear estructuras alternativas.

### **singly\_linkedList.js y doubly\_linkedList.js**

- Implementaciones educativas de Node y LinkedList.
- Métodos típicos: append/prepend, remove, find, toArray.
- doubly\_linkedList añade prev para navegación bidireccional y operaciones más eficientes en extremos.

### **stack.js y queue.js (+ queues/)**

- Stack: push, pop, peek, isEmpty — LIFO; uso con Array o lista enlazada.
- Queue: enqueue, dequeue, peek, isEmpty — FIFO; recomendaciones para O(1) (linked list o buffer circular).
- Ejemplos de uso: evaluación de expresiones (stack), BFS (queue), undo/redo.

### **hash\_table.js**

- Implementación de tabla hash con función hash básica y manejo de colisiones (encadenamiento o probing).
- Métodos: set, get, has, delete y consideraciones sobre rehashing/redimensionado.

### **graph.js**

- Representaciones y ejemplos: lista de adyacencia y posiblemente matrix.
- Algoritmos básicos incluidos o fácilmente añadibles: BFS, DFS.
- Uso práctico: modelar relaciones y traversals.

### **tree.js**

- Implementación de árbol, probablemente Binary Search Tree (BST).
  - Operaciones: insert, search, remove y recorridos (in-order, pre-order, post-order).
- 

## Breve Análisis Técnico

### Comparativa rápida

- Array (nativo): ideal para acceso por índice  $O(1)$ , menos eficiente para inserciones al frente ( $O(n)$ ).
- LinkedList: inserciones/eliminaciones  $O(1)$  en extremos, acceso aleatorio  $O(n)$ .
- Stack/Queue: patrones de acceso simples que se pueden implementar sobre Array o LinkedList según necesidades.
- Hash Table: acceso promedio  $O(1)$ , sensible a colisiones y factor de carga.
- Graph/Tree: estructuras no lineales; elección de representación afecta memoria y eficiencia de algoritmos.

### Complejidades (resumen)

- Acceso por índice (Array):  $O(1)$
- Insert/Remove en medio (Array):  $O(n)$
- Insert/Remove en extremos (LinkedList con tail/head):  $O(1)$
- Stack/Queue operations:  $O(1)$  (implementadas adecuadamente)
- HashTable get/set:  $O(1)$  promedio,  $O(n)$  peor caso
- BST operations:  $O(\log n)$  promedio,  $O(n)$  peor caso (degeneración)
- Graph traversals (BFS/DFS):  $O(V + E)$

### Recomendaciones pedagógicas

- Mantener implementaciones sencillas y bien comentadas para ilustrar comportamiento interno.
- Añadir tests simples y ejemplos de uso (ej. usar queue en BFS) para reforzar aprendizaje.
- Documentar limitaciones (por ejemplo, cuando usar Array vs LinkedList).

---

### Puntos de Aprendizaje

- Esta semana es ideal para consolidar la intuición sobre cost/beneficio de cada estructura.
  - Enseña cómo combinar estructuras para resolver problemas: p. ej. hash table para indexado rápido + lista para colisiones.
  - Refuerza algoritmos clásicos que dependen de estructuras (BFS con Queue, traversals con Stack o recursión).
- 

### Conclusión

Semana9 actúa como un checkpoint: reúne implementaciones esenciales y permite comparar directamente estrategias y complejidades. Reforzar con pruebas y ejemplos aplicados (pequeños retos) ayudará a internalizar cuándo y por qué usar cada estructura.

## Análisis - Semana 10: Árboles, Tablas y Estructuras Auxiliares

### Descripción General

La carpeta Semana10 agrupa implementaciones y ejercicios sobre **árboles (binary trees, tree)**, **tablas (hash/dictionary)** y estructuras auxiliares como **listas enlazadas y pilas**. El material mezcla implementaciones educativas (desde cero) con ejemplos prácticos para entender algoritmos de búsqueda, inserción y recorridos.

---

### Lenguaje

- **JavaScript (ES6+)**
- **HTML5** (páginas de ejemplo)
- **CSS3** (estilos básicos)

Las implementaciones usan JS puro, clases y estructuras nativas (Map, Set, Array) cuando conviene.

---

## Estructura de Archivos (observada)

Semana10/

```
|── alternative code.txt    # Notas / código alternativo  
|── binaryTree.js        # Implementación de árbol binario (insert, search, recorridos)  
|── code.js              # Ejemplos y utilidades generales  
|── dictionary.js        # Implementación/uso de diccionarios (key-value)  
|── hastTable.js         # (tipo) implementación de hash table educativa  
|── index.html           # Página de demostración  
|── linkedList.js        # Implementación de lista enlazada  
|── Set.js                # Ejemplo/implementación de Set personalizado  
|── style.css             # Estilos para las páginas  
|── stack/                # Implementación de pilas (stack.js)  
|   | └ stack.js  
└── tree/                 # Implementación de árboles (posible BST / utilities)  
   | └ tree.js
```

---

## Objetivos

1. Comprender y manipular **árboles binarios**: inserción, búsqueda y recorridos (in-, pre-, post-order).
2. Implementar y comparar **tablas hash / diccionarios** con estructuras nativas (Map) y manuales.
3. Repasar y aplicar **listas enlazadas** y **pilas** como apoyo en algoritmos de árboles.

4. Analizar complejidad y casos degenerados (árboles no balanceados, colisiones en hash).
  5. Entender cuándo usar estructuras nativas vs implementaciones educativas.
- 

## Descripción Detallada

### **binaryTree.js / tree/tree.js**

- Implementación de un árbol binario (posiblemente BST).
- Operaciones típicas:
  - insert(value) — insertar valor en posición correcta
  - search(value) — buscar un nodo
  - remove(value) — (si está implementado) eliminación con reasignación adecuada
  - Recorridos: inOrder, preOrder, postOrder
- Uso pedagógico: entender recursión y propiedades de árboles.

### **hashTable.js (hash table)**

- Implementación educativa de una hash table (nota: nombre del archivo tiene un typo).
- Elementos a revisar:
  - Función hash simple para convertir keys en índices
  - Manejo de colisiones (encadenamiento o probing)
  - Métodos: set, get, has, delete
- Comparación con dictionary.js (puede mostrar uso de Object o Map).

### **dictionary.js**

- Ejemplos de uso de diccionarios en JS: objetos planos o Map para key-value stores.
- Buenas prácticas: evitar usar objetos para claves no-string; preferir Map para claves arbitrarias.

### **linkedList.js y stack/stack.js**

- Soporte para operaciones auxiliares en árboles (p. ej. traversal iterativo requiere stack o queue).
- linkedList.js ofrece nodos y métodos (append, remove, find) útiles para colas/pilas personalizadas.
- stack.js implementa LIFO; útil para DFS iterativo o evaluaciones.

### **index.html y style.css**

- Interfaz para probar las implementaciones y visualizar recorridos o resultados.
  - Estilos mínimos para presentar nodos y estructuras.
- 

## **Breve Análisis Técnico**

### **Árboles (BST)**

- Operaciones promedio: insert, search  $\rightarrow O(\log n)$  si el árbol está balanceado.
- Peor caso:  $O(n)$  cuando el árbol está degenerado (inserciones ordenadas).
- Recomendación pedagógica: mostrar primero BST simple y luego comentar sobre balanceo (AVL/Red-Black).

### **Tablas Hash / Diccionarios**

- get/set promedio  $O(1)$ ; peor caso  $O(n)$  por colisiones.
- Importante: elegir o diseñar una función hash adecuada y controlar el factor de carga con rehashing.
- Para JS real, Map ofrece comportamiento robusto y manejo de claves no-string.

### **Estructuras auxiliares**

- Stack y Queue son esenciales para versiones iterativas de traversals (DFS/BFS).
- LinkedList puede ofrecer  $O(1)$  en enqueueues/dequeueues si se mantiene head y tail.

### **Calidad del código y mejoras**

- Corregir el nombre hastTable.js a hashTable.js para evitar confusión.
  - Añadir comentarios y ejemplos de uso en README.md para cada implementación.
  - Incluir tests simples que demuestren invariantes (por ejemplo, insert seguido de search debe encontrar el valor).
- 

### Puntos de Aprendizaje

- Los árboles muestran claramente la diferencia entre rendimiento promedio y degenerado.
  - Las hash tables requieren pensamiento sobre colisiones y redimensionamiento; Map es preferible en producción.
  - Combinar estructuras (por ejemplo, usar Stack para DFS) ilustra cómo componer soluciones.
- 

### Conclusión

Semana10 consolida conceptos avanzados de estructuras de datos: árboles y tablas son pilares para algoritmos eficientes. Fortalecer las implementaciones con limpieza de nombres, documentación y tests mejorará la usabilidad pedagógica y preparará al estudiante para estudiar balanceo de árboles, hashing robusto y algoritmos de búsqueda avanzados.

## Análisis - Semana 11: Aplicación Web de Productos (HTML, CSS, JS, JSON)

### Descripción General

La carpeta Semana11 contiene una pequeña aplicación web que muestra un catálogo de productos estático. Se centra en integrar **HTML5, CSS3 y JavaScript** para leer datos en formato **JSON** (products.json) y renderizar una vista interactiva en el navegador, con recursos en la carpeta assets.

---

## Lenguaje

- **HTML5** — estructura de la página (index.html)
  - **CSS3** — estilos en style.css ( posible uso de variables y clases)
  - **JavaScript (ES6+)** — lógica en app.js para cargar y renderizar products.json
  - **JSON** — datos de producto (products.json)
- 

## Estructura de Archivos

Semana11/

```
|—— index.html      # Página principal de la tienda / demo  
|—— app.js        # Lógica de la aplicación: carga y renderizado de productos  
|—— products.json   # Datos de ejemplo (array de objetos producto)  
|—— style.css      # Estilos de la UI  
└—— assets/        # Imágenes, íconos y otros recursos estáticos
```

---

## Objetivos

1. Aprender a consumir datos locales en JSON y renderizarlos dinámicamente en el DOM.
  2. Practicar manipulación del DOM y creación de componentes HTML vía JS (cards, listas).
  3. Implementar interactividad básica: filtrado, búsqueda, botones o handlers.
  4. Entender limitaciones de cargar JSON localmente (CORS / servidor vs file://).
  5. Mejorar estilos y disposición visual con style.css.
- 

## Descripción Detallada

**index.html**

- Plantilla principal que incluye estructura semántica (header, main, footer) y un contenedor donde app.js inyecta productos.
- Incluye referencia a style.css y app.js.

### **app.js**

- Funciones clave esperadas:
  - Cargar products.json mediante fetch() o XMLHttpRequest.
  - Parsear y transformar datos (map/filter) para preparar la representación.
  - Renderizar elementos (cards, listas) en el DOM.
  - Añadir listeners para interactividad (p. ej. filtros, botones "añadir al carrito").
- Posibles consideraciones: manejo de errores en fetch, estados de carga (skeleton/loading), y separación de responsabilidades (render vs data).

### **products.json**

- Array de objetos con campos típicos: id, title, description, price, image, category.
- Útil para practicar paginación, filtros por categoría y ordenamientos.

### **style.css y assets/**

- style.css define la estética: grid/flex layout para cards, tipografías, colores y responsive.
- assets/ contiene imágenes y recursos referenciados por products.json o index.html.

---

## Breve Análisis Técnico

### **1. Consumo de JSON local**

- fetch('products.json') funciona correctamente si la página se sirve por HTTP(S) (p. ej. localhost).

- Si se abre desde file://, algunos navegadores bloquean fetch por políticas CORS/archivo; usar un servidor local (p. ej. npx http-server o python -m http.server) para pruebas.

## 2. Renderizado y rendimiento

- Renderizar muchos productos puede impactar el DOM; usar fragmentos de documento (DocumentFragment) o render por lotes mejora rendimiento.
- Para render dinámico: crear plantillas (template literals) o <template> en HTML y clonarlo.

## 3. UX e interactividad

- Añadir estado de carga y manejo de errores mejora la experiencia.
- Funcionalidades recomendadas: búsqueda en vivo, filtros por categoría, ordenamiento por precio, paginación, y un pequeño carrito local (localStorage).

## 4. Accesibilidad y SEO

- Usar alt en imágenes, roles ARIA para componentes dinámicos y botones accesibles.
- Si el sitio se despliega estáticamente, considerar prerendering para SEO (no obligatorio para ejercicios).

## 5. Mejora del código y modularidad

- Separar la lógica en módulos: api.js (fetch), ui.js (render), utils.js.
- Evitar lógica inline en index.html; usar addEventListener en app.js.
- Añadir pruebas básicas (p. ej. comprobaciones unitarias de funciones puras).

---

### 💡 Puntos de Aprendizaje

- Integración práctica de fetch() + JSON con DOM es una habilidad central en desarrollo web.
- Diferencias entre desarrollo local y servido: siempre probar con un servidor local.

- Mejoras incrementales (filtrado, paginación, caching) transforman una demo en una mini-app real.
- 

## Conclusión

Semana11 ofrece un ejercicio práctico y completo para consolidar HTML/CSS/JS con datos JSON. Es excelente para practicar el flujo data → transform → render y para introducir buenas prácticas (modularidad, manejo de errores y accesibilidad). Para preparar despliegue o pruebas más reales, ejecutar la app en un servidor local y añadir controles de interacción avanzados.

## Análisis - Semana 12: Estructura de Proyecto Web (Archivos iniciales)

### Descripción General

La carpeta Semana12 contiene una plantilla/estructura inicial para proyectos web: HTML base, recursos estáticos (CSS, imágenes), scripts JavaScript y configuración de editor. Es útil como punto de partida para prácticas y despliegues sencillos.

---

### Lenguaje

- **HTML5** — estructura del proyecto
  - **CSS3** — estilos (carpeta css/)
  - **JavaScript (ES6+)** — lógica de cliente en js/
  - **JSON / configuración** — opcional en .vscode (settings)
- 

### Estructura de Archivos (observada)

Semana12/

└─ archivos-iniciales/

  └─ index.html # Plantilla HTML base

  └─ .vscode/ # Configuración de espacio de trabajo (opcional)

```
|— css/      # Estilos (main.css, reset, etc.)  
|— img/      # Imágenes y assets  
└— js/       # Scripts de interacción
```

---

## Objetivos

1. Proveer una plantilla mínima para comenzar proyectos web.
  2. Enseñar buenas prácticas de organización (separar css, js, img).
  3. Facilitar pruebas locales y despliegue estático (servir index.html).
  4. Permitir personalización rápida para ejercicios y demos.
- 

## Descripción Detallada

- index.html: Documento HTML5 básico con meta charset, viewport y enlaces a css/main.css y js/main.js.
  - css/: Hoja(s) de estilo para layout y componentes.
  - img/: Recursos gráficos organizados por tema o componente.
  - js/: Código de interacción (event listeners, manipulación DOM, pequeñas utilidades).
  - .vscode/ (si existe): Configs útiles para desarrollar (formatters, liveServer settings, snippets).
- 

## Breve Análisis Técnico

- Estructura clásica y apropiada para proyectos estáticos y prácticas didácticas.
- Recomendación para desarrollo local: usar un servidor estático (por ejemplo python -m http.server o npx http-server) para evitar problemas con fetch y rutas relativas.
- Buenas prácticas: usar index.html como punto de entrada, agrupar assets y mantener código modular (js/main.js, js/modules/).

- Considerar añadir un archivo README.md con instrucciones de arranque y dependencias (si las hubiera) y .gitignore para controlar versiones.
- 

### Puntos de Aprendizaje

- Separación de responsabilidades (html/css/js) facilita colaboración y pruebas.
  - Empezar desde una plantilla reduce fricción y acelera experimentación.
  - La configuración del editor en .vscode mejora consistencia del equipo (line endings, formato).
- 

### Conclusión

Semana12 proporciona una plantilla sólida para comenzar proyectos web prácticos. Para avanzar, se recomienda añadir ejemplos concretos (componentes, pequeños scripts de interacción), documentación (README.md) y scripts de desarrollo (package.json) si se incorporarán herramientas de build.

## Análisis - Semana 13: Recursión y Algoritmos de Ordenamiento

### Descripción General

La carpeta Semana13 se enfoca en dos temas clave de algoritmos: **recursión** y **ordenamiento**. Contiene ejemplos y utilidades para entender la pila de llamadas, distintos estilos de implementación (iterativa vs recursiva) y algoritmos de ordenamiento clásicos (mergesort, quicksort, selection sort).

---

### Lenguaje

- **JavaScript (ES6+)** — implementaciones de funciones recursivas y algoritmos de ordenamiento
- **HTML5** — interfaz de demostración (index.html)
- **Markdown** — documentación (README.md, database.md, processes.md, design.md)

---

## Estructura de Archivos

Semana13/

```
|— index.html          # Página de demostración / ejercicios  
|— main.js            # Código de interacción o ejemplos globales  
|— README.md          # Notas generales de la semana  
|— database.md         # Notas relacionadas (si aplica)  
|— processes.md        # Descripción de procesos/algoritmos  
|— design.md           # Diseño de soluciones o diagramas  
|— Recursion/          # Ejemplos y prácticas sobre recursión  
|   |— callStackExample.js  
|   |— countToZero.js  
|   |— fibonaccilterative.js  
|   |— fibonacciRecursive.js  
|   |— fibonacciRecursiveBetter.js  
|— Sorting/            # Implementaciones de ordenamiento  
|   |— mergesort.js  
|   |— quicksort.js  
|   |— selectionSort.js
```

---

## Objetivos

1. Comprender la recursión: cómo funciona la pila de llamadas y cuándo usarla.
2. Comparar implementaciones recursivas e iterativas (ej.: Fibonacci recursivo vs iterativo).
3. Implementar y analizar algoritmos de ordenamiento clásicos: mergesort, quicksort y selection sort.

4. Calcular y comparar complejidades temporales y espaciales de cada algoritmo.
  5. Practicar la transformación de una solución recursiva a una iterativa y viceversa.
- 

## Descripción Detallada

### Carpeta Recursion/

- Ejemplos que ilustran conceptos:
  - callStackExample.js: visualiza la pila de llamadas y la entrada/salida de funciones.
  - countToZero.js: ejemplo simple de recursión con condición base clara.
  - fibonacciRecursive.js vs fibonacciIterative.js: comparación directa de rendimiento y complejidad.
  - fibonacciRecursiveBetter.js: optimizaciones como memoización para mejorar recursión.

### Carpeta Sorting/

- mergesort.js: algoritmo divide y vencerás con complejidad  $O(n \log n)$  y espacio adicional  $O(n)$ .
  - quicksort.js: algoritmo promedio  $O(n \log n)$ , peor caso  $O(n^2)$ ; discusión sobre pivot y particionamiento.
  - selectionSort.js: algoritmo sencillo con complejidad  $O(n^2)$ , útil para entender principios básicos.
- 

## Breve Análisis Técnico

### Recursión

- Conceptos clave:
  - **Caso base:** condición que detiene la recursión.
  - **Llamada recursiva:** descomposición del problema en subproblemas.

- **Pila de llamadas:** cada llamada ocupa espacio en la pila; riesgo de stack overflow si la profundidad es grande.
- Optimización: **memoización** para evitar recalcular subproblemas (ej. Fibonacci), y **tail recursion** (cuando el motor JS lo optimiza).
- Trade-offs: la recursión suele ser más legible, pero puede usar más memoria; la versión iterativa suele ser más eficiente en espacio.

## Ordenamiento

- Complejidades y características:
  - **Mergesort:**  $O(n \log n)$  tiempo,  $O(n)$  espacio; estable; buen rendimiento en datos grandes.
  - **Quicksort:**  $O(n \log n)$  promedio,  $O(n^2)$  peor caso; in-place posible; elegir buen pivote (randomizado o median-of-three) reduce probabilidad de peor caso.
  - **Selection Sort:**  $O(n^2)$  tiempo,  $O(1)$  espacio; sencillo pero ineficiente para grandes  $n$ ; útil en contextos con memoria limitada y tamaños pequeños.
- Consideraciones prácticas: para datos parcialmente ordenados o datos grandes en memoria limitada, elegir el algoritmo adecuado según caso.

## Recomendaciones de práctica

- Añadir pruebas de rendimiento simples (timings) comparando implementaciones para distintos tamaños de entrada.
- Visualizar recursión y ordenamiento (pequeñas animaciones) ayuda a la comprensión.
- Implementar memoización en problemas recursivos con subproblemas repetidos.

### Puntos de Aprendizaje

- Entender la recursión conceptualmente es esencial para muchos algoritmos; la memoización convierte soluciones recursivas exponenciales en polinomiales.

- Los algoritmos de ordenamiento enseñan estrategias (divide & conquer, in-place, stable vs unstable) y son la base para algoritmos más avanzados.
  - Comparar implementaciones y medir con casos de prueba reales afianza la elección de algoritmo.
- 

## Conclusión

Semana13 es una semana clave para interiorizar principios algorítmicos: recursión y ordenamiento. Practicar con ejemplos concretos (Fibonacci, mergesort, quicksort) y medir su comportamiento hará que los conceptos teóricos se vuelvan aplicables en problemas reales de programación y optimización.

## Análisis - Semana 14: Proyecto Demo — Objetos, Interactividad y Estilos

### Descripción General

La carpeta Semana14 contiene un pequeño proyecto de demostración que integra **HTML**, **CSS** y **JavaScript** para practicar conceptos de objetos en JavaScript, manipulación del DOM y estilos. Incluye ejemplos y notas que sirven como referencia para aplicar patrones básicos de organización y diseño de UI.

---

### Lenguaje

- **HTML5** — estructura de la página (index.html).
  - **CSS3** — estilos en styles.css.
  - **JavaScript (ES6+)** — lógica en app.js y ejemplos en demo/objetos.js.
  - **Markdown** — notas.md con apuntes o recordatorios.
- 

### Estructura de Archivos

Semana14/

|— index.html # Página principal del demo

```
├── app.js          # Lógica principal / handlers de UI  
├── notas.md        # Apuntes y notas de la semana  
├── styles.css      # Estilos y layout del demo  
├── .vscode/         # Configuración del espacio (opcional)  
└── demo/  
    └── objetos.js   # Ejemplos prácticos con objetos JS (clases, literales)
```

---

## Objetivos

1. Practicar la creación y uso de **objetos** en JavaScript (literales y/o clases).
  2. Manipular el DOM para actualizar la interfaz desde app.js.
  3. Aplicar estilos responsivos y coherentes con styles.css.
  4. Separar responsabilidades: lógica (app.js) vs presentación (index.html/styles.css).
  5. Documentar aprendizajes y observaciones en notas.md.
- 

## Descripción Detallada

- index.html sirve como plantilla de la demo: contiene elementos interactivos (botones, formularios o tarjetas) que conectan con app.js.
  - app.js implementa la lógica de interacción: event listeners, manipulación del DOM (crear/actualizar nodos), y llamadas a funciones del demo.
  - demo/objetos.js contiene ejemplos didácticos sobre objetos: creación de objetos literales, uso de this, constructores o clases, métodos y ejemplos de mutación vs inmutabilidad.
  - styles.css define la apariencia: layout (grid/flex), variables de color, y estilos responsivos.
  - notas.md recopila observaciones, tareas pendientes o recordatorios del autor.
-

## Breve Análisis Técnico

### Organización y buenas prácticas

- La separación index.html (markup) / styles.css (presentación) / app.js (comportamiento) es correcta y facilita mantenimiento.
- demo/objetos.js es útil como módulo de ejemplos; sería ideal importarlo explícitamente desde app.js (ES Modules) para mejorar modularidad.

### Interactividad y rendimiento

- Para muchas operaciones DOM, usar DocumentFragment o actualizaciones por lotes reduce repaints.
- Evitar lógica inline (onclick) y preferir addEventListener en app.js mejora testabilidad.

### Mantenimiento y escalabilidad

- Convertir ejemplos en módulos (/demo, /lib) y usar type="module" en <script> prepara el proyecto para crecimiento.
- Añadir un README.md en la carpeta con instrucciones de ejecución (usar servidor local) ayuda a reproducir la demo.

### Accesibilidad

- Verificar alt en imágenes, foco en inputs, y atributos ARIA donde aplique.
- Asegurar contraste adecuado en styles.css para cumplir con WCAG básicos.

---

## Recomendaciones rápidas

- Usar módulos ES (import / export) para demo/objetos.js y app.js.
- Añadir un pequeño script serve (p. ej. npx http-server) en README.md para pruebas locales.
- Documentar ejemplos en notas.md con ejemplos de entrada/salida para cada función demostrada.

---

## Conclusión

Semana14 es una práctica compacta que integra objetos en JavaScript con manipulación del DOM y estilos CSS. Con pequeñas mejoras (modularización, documentación y accesibilidad) la demo puede evolucionar a una colección de componentes reutilizables y un material de referencia útil para estudiantes.

## Análisis - Semana 15: Grafos Avanzados, Tablas Hash y Listas Enlazadas

### Descripción General

Semana15 aborda implementaciones y ejemplos prácticos sobre **grafos, tablas hash y listas enlazadas** (dobles y simples). El material combina teoría y código para comprender representaciones de grafos, manejo de colisiones en tablas hash y operaciones en listas enlazadas.

---

### Lenguaje

- **JavaScript (ES6+)** — clases, módulos simples y manipulación de estructuras.
  - **HTML5** — páginas de demostración (si existen) para probar algoritmos.
  - **CSS3** — estilos mínimos para visualizar resultados (opcional).
- 

### Estructura de Archivos (observada)

Semana15/

```
|— code.js          # Ejemplos generales / utilidades  
|— graph/          # Implementaciones y ejemplos de grafos  
|   |— graph.js  
|   |— graph_2.js  
|— hashTable/      # Implementación educativa de hash table  
|   |— hashTable.js  
└— linkedList/    # Listas enlazadas (doble y simple)
```

```
|— dobly.js
```

```
└— singly.js
```

---

## Objetivos

1. Comprender representaciones de grafos (lista de adyacencia, matriz) y operaciones básicas.
  2. Implementar y probar algoritmos sobre grafos (BFS/DFS o utilidades similares).
  3. Revisar la implementación de una **hash table** y técnicas de colisión.
  4. Practicar operaciones en `linkedList` (`append`, `remove`, `find`) y en `dobly` (`prev/next`).
  5. Comparar rendimiento y casos de uso de cada estructura.
- 

## Descripción Detallada

### Carpeta graph/

- `graph.js` y `graph_2.js` contienen implementaciones y ejemplos de grafos.
- Deben mostrar: creación de vértices, adición de aristas, impresión de la lista de adyacencia y ejemplos de traversals.
- Utilidad didáctica: modelar relaciones y aplicar BFS/DFS para búsquedas y componentes conectados.

### Carpeta hashTable/

- `hashTable.js` implementa una tabla hash educativa con funciones básicas:
  - `set(key, value)`, `get(key)`, `delete(key)`, `has(key)`.
- Revisar manejo de colisiones (encadenamiento vs open addressing) y rehashing si aplica.

### Carpeta linkedList/

- `singly.js`: lista simplemente enlazada con `head`, `append`, `remove`, `find`.

- `dobly.js`: lista doblemente enlazada con `prev` y `next`, métodos para insertar/eliminar en ambos extremos, y recorrido inverso.
  - Importante mantener `length`, `head` y `tail` para eficiencia  $O(1)$  en operaciones de extremos.
- 

## Breve Análisis Técnico

### Grafos

- Representación recomendada: **lista de adyacencia** para grafos dispersos ( $E << V^2$ ).
- Operaciones importantes: agregar vértices/aristas  $O(1)$ , BFS/DFS  $O(V + E)$ .
- Verificar mutabilidad de la estructura al agregar/eliminar nodos.

### Tablas Hash

- Operaciones promedio  $O(1)$ ; el rendimiento depende de la calidad del hash y del factor de carga.
- Para enseñanza, el encadenamiento es más sencillo de explicar e implementar.
- Añadir pruebas con claves que provoquen colisiones para validar la robustez.

### Listas Enlazadas

- `singly`: insert/append  $O(1)$  con `tail`, búsqueda/removal  $O(n)$ .
  - `dobly`: soporta recorrido inverso y eliminación de un nodo en  $O(1)$  si se tiene referencia al nodo.
  - Casos de uso: implementación de colas, pilas, y estructuras intermedias.
- 

## Recomendaciones y Buenas Prácticas

- Modularizar código: exportar `Graph`, `HashTable`, `LinkedList` como módulos reutilizables.
- Añadir tests unitarios básicos para cada estructura (operaciones invariantes).

- Documentar limitaciones y complejidades en un README.md dentro de cada carpeta.
- 

## Conclusión

Semana15 integra estructuras críticas para problemas reales: grafos para relaciones complejas, hash tables para acceso rápido por clave y listas enlazadas para manipulación dinámica. Fortalecer las implementaciones con tests, ejemplos de uso (p. ej. BFS sobre datos reales) y modularidad mejorará la enseñanza y la reutilización del código.

## Análisis - Semana 16: Pilas, Colas y Árboles — Aplicaciones y Prácticas

### Descripción General

La carpeta Semana16 contiene implementaciones y ejemplos prácticos de **colas (queues)**, **pilas (stacks)** y **árboles (trees)**, además de una página de demostración index.html, estilos index.css y lógica front-end en index.js. Esta semana integra estructuras lineales y no lineales enfocadas en su uso práctico y aplicaciones simples.

---

### Lenguaje

- **JavaScript (ES6+)** — implementaciones y demo interactivas (index.js, queue/, stack/, tree/).
  - **HTML5** — interfaz de demostración (index.html).
  - **CSS3** — estilos en index.css.
- 

### Estructura de Archivos (observada)

Semana16/

```
|— index.html      # Página demo para interactuar con estructuras  
|— index.css       # Estilos de la demo
```

```
├── index.js      # Lógica de la demo (conexión UI ↔ estructuras)
├── queue/
│   └── queue.js    # Implementación de cola (enqueue, dequeue, peek)
└── stack/
    └── stack.js    # Implementación de pila (push, pop, peek)
└── tree/
    └── tree.js     # Implementación de árbol (posible BST o utilidades)
```

---

## Objetivos

1. Implementar y usar Queue (FIFO) y Stack (LIFO) correctamente en ejemplos prácticos.
  2. Entender cómo usar una Queue para BFS y una Stack para DFS (iterativo).
  3. Manejar operaciones básicas en Tree (inserción, búsqueda, recorridos) y conectar con la UI.
  4. Mostrar interactividad con index.html para experimentar con las estructuras.
  5. Aplicar buenas prácticas: modularidad, separación UI/logic y manejo de errores.
- 

## Descripción Detallada

### queue/queue.js

- Implementa los métodos básicos:
  - enqueue(item) — agregar al final
  - dequeue() — eliminar del frente
  - peek() — ver el elemento frontal
  - isEmpty() — comprobar si está vacía
- Recomendación: implementar con head/tail (lista ligada) o buffer circular para O(1) en dequeue.

### **stack/stack.js**

- Implementa push, pop, peek, isEmpty.
- Implementación simple con Array (push/pop) es aceptable para demos.

### **tree/tree.js**

- Implementación de árbol (probablemente BST) con insert, search y recorridos (inOrder, preOrder, postOrder).
- Usos prácticos: mostrar nodos en la UI, ordenar datos mediante inOrder si es BST.

### **index.html + index.js + index.css**

- index.html contiene controles (formularios/botones) para interactuar con las estructuras.
  - index.js conecta eventos de la UI con llamadas a los métodos de Queue/Stack/Tree y actualiza la vista.
  - index.css estiliza la visualización de nodos, colas o pilas para facilitar la comprensión.
- 

## **Breve Análisis Técnico**

### **Complejidades y recomendaciones**

- Queue y Stack bien implementadas tienen operaciones O(1) para encolar/desencolar y push/pop.
- Tree (BST) operaciones: insert/search O(log n) promedio, O(n) peor caso; documentar limitaciones.
- Para demos con muchos elementos, optimizar renderizado usando DocumentFragment y actualizaciones parciales del DOM.

### **Robustez y UX**

- Validar entradas en la UI y mostrar mensajes de error cuando las operaciones no sean válidas (por ejemplo, dequeue en cola vacía).
- Añadir visualizaciones simples (lista horizontal para Queue, vertical para Stack, diagrama básico para Tree) ayuda al aprendizaje.

## Pruebas y mantenibilidad

- Separar las implementaciones (queue/queue.js, stack/stack.js, tree/tree.js) y exportarlas como módulos facilita pruebas unitarias.
  - Añadir pruebas simples (p. ej. scripts que verifican invariantes) mejora la confiabilidad del material didáctico.
- 

## Puntos de Aprendizaje

- Entender cuándo usar Queue vs Stack según patrón de acceso (FIFO vs LIFO).
  - Relacionar Queue con algoritmos de recorrido por niveles (BFS) y Stack con DFS iterativo.
  - Conocer las limitaciones de un BST no balanceado y la necesidad de balanceo en escenarios reales.
  - La integración UI ↔ estructuras transforma conceptos teóricos en experiencia práctica.
- 

## Conclusión

Semana 16 cierra el ciclo práctico mostrando estructuras de acceso simple y árboles, y cómo exponerlas en una UI interactiva. Mejoras sugeridas: modularizar como ES modules, añadir tests y optimizar la visualización para datasets más grandes.