DSViz Listas enlazadas: diseño

Arquitectura y decisiones de diseño

Gustavo Gutiérrez-Sabogal
Jovanny Bedoya-Guapacha
Nancy Janet Castillo-Rodríguez

Universidad Tecnológica de Pereira

Índice

1. Introducción

Este documento describe las decisiones de diseño y la arquitectura del visualizador de listas enlazadas **DSViz: Listas enlazadas**. Su objetivo es servir como referencia técnica para desarrolladores y colaboradores del proyecto.

1.1 Motivación

DSViz: Listas enlazadas es una herramienta pedagógica diseñada para la enseñanza de conceptos en el curso de *Estructuras de datos* en la Universidad Tecnológica de Pereira dentro el programa de Ingeniería de sistemas y computación. Las decisiones de diseño están guiadas por los siguientes principios:

- **Cero instalación**: la aplicación se ejecuta completamente en el navegador como una *Single Page Application* (SPA) desplegada en GitHub Pages.
- **Interactividad**: los estudiantes tienen acceso a la estructura interna con los nodos y apuntadores de la estructura de datos.
- **Visualización paso a paso de operaciones**: cada operación sobre la estructura de datos se anima de manera secuencial para facilitar la comprensión del algoritmo subyacente.
- **Separación conceptual**: los elementos de programación (punteros, direcciones de memoria) se mantienen separados de la representación conceptual de la estructura.

1.2 Tecnologías principales

La aplicación está construida sobre el siguiente conjunto de tecnologías:

Categoría	Tecnología	Versión
Framework	Next.js (App Router)	16
Lenguaje	TypeScript (modo estricto)	5.9
Biblioteca UI	React	19
Visualización	React Flow (`@xyflow`)	12
Disposición	ELK.js (Sugiyama)	0.11
Estilos	Tailwind CSS	3.4
Componentes	shadcn/ui (Radix UI)	—
Tour guiado	driver.js	1.4
Exportación	html-to-image	1.11

1.3 Alcance

El presente documento cubre la versión actual del módulo. Se describen las tecnologías utilizadas, la estructura del proyecto, el modelo de datos, el motor de visualización, la interfaz de usuario y el proceso de despliegue.

1.4 Audiencia

Este documento está dirigido a desarrolladores que deseen contribuir al proyecto, así como a docentes y estudiantes interesados en comprender la implementación interna del simulador.

1.5 Estructura del documento

El documento se organiza en los siguientes capítulos:

1. **Arquitectura General** (Capítulo 2.): describe las tecnologías, la estructura del proyecto y el flujo de datos.
2. **Modelo de Datos** (Capítulo 3.): detalla las interfaces de tipos, las fábricas de nodos y aristas, y la gestión de estado.
3. **Motor de Visualización** (Capítulo 4.): explica la integración con xyflow, el sistema de animaciones y la disposición automática con ELK.
4. **Interfaz de Usuario** (Capítulo 5.): presenta los componentes principales, el sistema de menús y la estilización.
5. **Despliegue y CI/CD** (Capítulo 6.): documenta el proceso de construcción, despliegue en GitHub Pages y el flujo de trabajo con GitHub Actions.

2. Arquitectura General

2.1 Tecnologías principales

2.1.1 Framework y construcción

La aplicación utiliza **Next.js 16** con el *App Router* como framework principal. Next.js fue seleccionado por las siguientes razones:

- Soporte nativo para TypeScript y compilación incremental.
- Sistema de rutas basado en el sistema de archivos (`src/app/`).
- Exportación estática (`output: 'export'`) que permite desplegar en GitHub Pages sin necesidad de un servidor.

El proyecto se compila con **TypeScript 5.9** en modo estricto (`strict: true`), lo que garantiza seguridad de tipos en todo el código fuente. La configuración de TypeScript utiliza alias de rutas (`@/*` apunta a `./src/*`) para simplificar las importaciones.

2.1.2 Visualización

El motor de visualización se basa en dos bibliotecas complementarias:

- **`@xyflow/react`** (React Flow v12): biblioteca para la visualización interactiva de grafos basados en nodos y aristas. Proporciona el lienzo (*canvas*), el sistema de nodos arrastrables, las conexiones entre nodos y los controles de navegación.
- **ELK.js** (v0.11): biblioteca de disposición automática de grafos que implementa el algoritmo de Sugiyama (*layered layout*). Se utiliza para organizar automáticamente los nodos de la lista en disposición horizontal.

2.1.3 Interfaz de usuario

La interfaz se construye con las siguientes tecnologías:

- **Tailwind CSS** (v3.4): framework CSS *utility-first* para la estilización.
- **shadcn/ui**: colección de componentes reutilizables construidos sobre primitivas de **Radix UI**. Incluye botones, entradas de texto, *sliders*, *switches* y selectores.
- **Lucide React**: biblioteca de iconos SVG.
- **driver.js**: biblioteca para tours guiados interactivos con soporte para localización en español.

2.2 Estructura del proyecto

El código fuente se organiza bajo el directorio `src/` siguiendo la convención del *App Router* de Next.js:

```
src/
├── app/
│   ├── layout.tsx                # Layout raíz (fuentes, estilos globales)
│   ├── page.tsx                  # Página principal (redirección a /sll)
│   ├── globals.css               # Estilos CSS globales y Tailwind
│   ├── about/page.tsx            # Página "Acerca de"
│   └── sll/                      # Módulo de lista enlazada simple
│       ├── page.tsx              # Componente principal (orquestador)
│       ├── LinkedListNode.tsx    # Componente de nodo de lista
│       ├── CircleNode.tsx        # Componente de nodo circular (punteros)
│       ├── constants.ts          # Constantes de configuración
│       ├── hooks/                # Custom hooks
│       │   ├── useListOperations.ts
│       │   ├── useListVisualization.ts
```

```
│       │   └── useListInitialization.ts
│       ├── operations/            # 16 operaciones individuales
│       │   ├── index.ts
│       │   ├── insertAtHead.ts
│       │   ├── insertAtTail.ts
│       │   ├── insertAtTail01.ts
│       │   ├── insertAtPosition.ts
│       │   ├── deleteAtHead.ts
│       │   ├── deleteAtTail.ts
│       │   ├── deleteAtPosition.ts
│       │   ├── traverseList.ts
│       │   ├── reverseList.ts
│       │   ├── searchValue.ts
│       │   ├── getLength.ts
│       │   ├── findMiddle.ts
│       │   ├── accessFront.ts
│       │   ├── accessBack.ts
│       │   ├── accessNth.ts
│       │   └── removeDuplicates.ts
│       └── utils/                 # Utilidades
│           ├── nodeFactory.ts
│           ├── edgeFactory.ts
│           ├── nodeFilters.ts
│           ├── pointerHelpers.ts
│           ├── listHelpers.ts
│           ├── elkLayout.ts
│           └── helpers.ts
├── components/                    # Componentes compartidos
│   ├── menu/
│   │   ├── Menu.tsx               # Panel lateral de controles
│   │   ├── MenuHeader.tsx         # Encabezado animado del menú
│   │   ├── Section.tsx            # Sección colapsable reutilizable
│   │   ├── ColorPickerInput.tsx   # Selector de color
│   │   └── sections/
│   │       ├── ListCreation.tsx   # Creación de listas
│   │       ├── Operations.tsx     # Operaciones sobre la lista
│   │       ├── DisplayOptions.tsx # Opciones de visualización
│   │       └── TestSection.tsx    # Sección experimental
│   ├── navbar.tsx                 # Barra de navegación
│   ├── footer.tsx                 # Pie de página
│   ├── hero.tsx                   # Sección hero de la landing
│   ├── ui/                        # Componentes atómicos (shadcn/ui)
│   │   ├── button.tsx
│   │   ├── input.tsx
│   │   ├── slider.tsx
│   │   ├── select.tsx
│   │   ├── switch.tsx
│   │   └── card.tsx
│   ├── custom-slider.tsx          # Slider con etiqueta
│   ├── custom-select.tsx          # Selector personalizado
│   ├── custom-toggle.tsx          # Toggle con etiqueta
│   ├── custom-input.tsx           # Input personalizado
│   └── algorithm-cards.tsx        # Tarjetas de algoritmos
├── types/                         # Definiciones de tipos
│   ├── linked-list.ts             # Tipos del modelo de datos
│   ├── common.ts                  # Tipos compartidos (DOM, selección)
│   ├── hooks.ts                   # Tipos de hooks
│   ├── utils.ts                   # Tipos de utilidades
│   └── index.ts                   # Exportaciones centralizadas
└── lib/
    ├── utils.ts                   # Función cn() para clases CSS
```

```
    └── helpers/
        └── array_helpers.ts       # Utilidades de arrays
```

2.2.1 Organización modular

La estructura del módulo de visualización está organizada de manera modular con una estructuraconsistente:

- **`page.tsx`**: componente principal que actúa como orquestador del estado y la visualización.
- **`hooks/`**: *custom hooks* que encapsulan la lógica del visualizador. Estos permiten separar la lógica de la visualización de los componentes.
- **`operations/`**: funciones puras que implementan cada operación sobre la estructura de datos.
- **`utils/`**: fábricas y funciones auxiliares.

Esta separación permite agregar nuevas operaciones y algoritmos siguiendo el mismo patrón.

2.3 Flujo de datos

El flujo de datos de la aplicación sigue un patrón unidireccional que puede describirse en las siguientes etapas:

1. **Acción del usuario**: el usuario interactúa con el menú lateral (por ejemplo, hace clic en «At Head» para insertar un elemento).
2. **Manejador de evento**: el componente `Menu` invoca el *callback* `onVisualize(opIndex, value, position)` que está definido en `page.tsx`.
3. **Despacho de operación**: la función `handleVisualize` en `page.tsx` determina la operación correspondiente mediante un `switch` y la delega al *hook* `useListOperations`.
4. **Ejecución de la operación**: el *hook* construye un `OperationContext` con el estado actual y lo pasa a la función de operación correspondiente (por ejemplo, `insertAtHead`).
5. **Actualización de estado**: la operación modifica los nodos y aristas llamando a `setState()`, que actualiza el estado de React en `page.tsx`.
6. **Procesamiento visual**: el *hook* `useListVisualization` filtra y transforma los nodos y aristas según las opciones de visualización activas (punteros visibles, resaltado de cabeza/cola, nodo bajo el cursor).
7. **Renderizado**: React Flow recibe los nodos y aristas procesados (`highlightedNodes`, `highlightedEdges`) y los renderiza en el lienzo.

El siguiente diagrama de secuencia ilustra este flujo de datos desde la acción del usuario hasta el renderizado de la visualización:

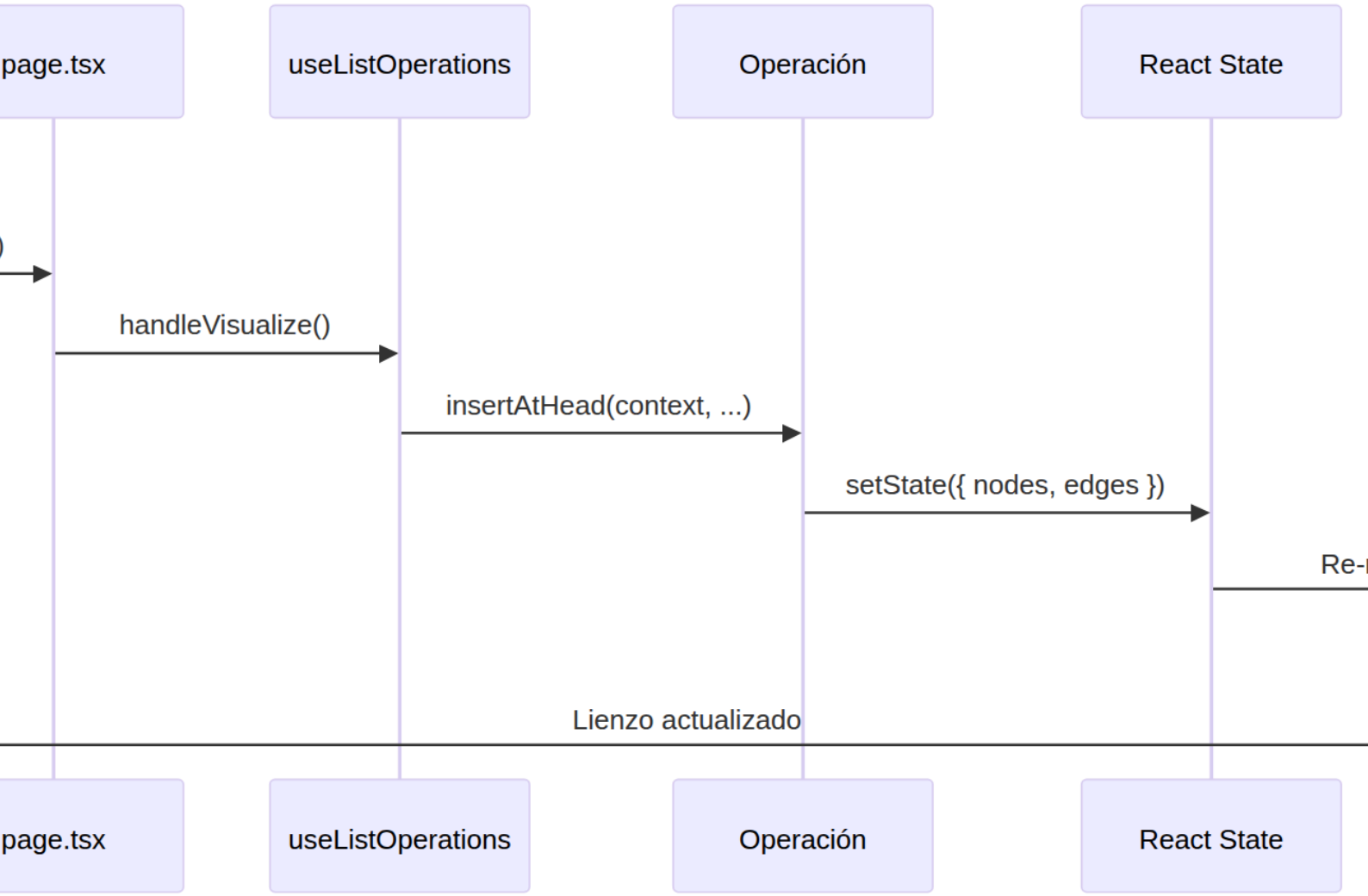

Figura 2.1: Flujo de datos desde la acción del usuario hasta el renderizado.

2.3.1 Animaciones

Las animaciones se logran mediante un patrón asíncrono: cada operación es una función `async` que alterna entre llamadas a `setState()` (para actualizar la visualización) y `await sleep(speed)` (para pausar entre fotogramas). Esto permite que el usuario observe el progreso paso a paso de cada algoritmo. La velocidad de animación es configurable a través del *slider* de velocidad en el menú de opciones de visualización.

3. Modelo de Datos

Este capítulo describe las estructuras de datos internas utilizadas para representar la lista enlazada en el motor de visualización. Todas las definiciones de tipos se encuentran en el directorio `src/types/`.

3.1 Representación de nodos

La visualización utiliza dos tipos de nodos, ambos basados en la interfaz `Node` de xyflow:

3.1.1 Nodo de lista (`LinkedListNode`)

Representa un elemento de la lista enlazada. Su interfaz de datos es:

```
interface ListNodeData {
  label: string;                              // Valor del nodo
  nodeId: string;                             // Identificador único
  onPointerHover: (nodeId: string | null) => void; // Callback de hover
}
```

Cada nodo de lista tiene cuatro *handles* (puntos de conexión):

- **Right** (*source*): punto de salida para la arista al siguiente nodo.
- **Left** (*target*): punto de entrada desde el nodo anterior.
- **Top** (*target*): punto de conexión para punteros desde arriba (e.g. *head*).
- **Bottom** (*target*): punto de conexión para punteros desde abajo (e.g. *tail*).

La creación de nodos se realiza mediante la función fábrica `createListNode` definida en `src/app/sll/utils/nodeFactory.ts`, que recibe un identificador, valor, posición y color de fondo.

3.1.2 Nodo circular (`CircleNode`)

Representa un puntero o referencia. Se utiliza para visualizar los punteros `head`, `tail` y los punteros adicionales creados por el usuario.

```
interface CircleNodeData {
  label: string;                                  // Etiqueta (H, T, C)
  nodeId: string;                                 // Identificador único
  onPointerHover: (nodeId: string | null) => void;  // Callback de hover
  onLabelChange: (nodeId: string, newLabel: string) => void; // Edición
}
```

Los nodos circulares permiten edición de la etiqueta mediante doble clic (máximo 3 caracteres). Se crean con `createCircleNode` de `nodeFactory.ts`.

3.2 Representación de aristas

Las aristas representan las conexiones entre nodos (representando el concepto de puntero). La aplicación utiliza tres tipos de aristas, todas creadas por funciones fábrica en `src/app/sll/utils/edgeFactory.ts`:

3.2.1 Aristas de lista

Conectan nodos consecutivos de la lista (del *handle* derecho de un nodo al *handle* izquierdo del siguiente). Utilizan el tipo `smoothstep` con una flecha `ArrowClosed` como marcador.

```
function createListEdge(
  sourceId: string,
  targetId: string
): Edge
```

3.2.2 Aristas de puntero

Conectan los nodos circulares (*head*, *tail*) con los nodos de la lista:

- **`createHeadPointerEdge`**: conecta el puntero *head* al *handle* superior del primer nodo de la lista.
- **`createTailPointerEdge`**: conecta el puntero *tail* al *handle* inferior del último nodo de la lista.

Estas aristas se reconstruyen automáticamente después de cada operación estructural mediante la función `createPointerEdges` de `src/app/sll/utils/pointerHelpers.ts`.

3.3 Gestión de estado

El estado de la aplicación se gestiona enteramente en el componente principal `page.tsx` mediante *hooks* de estado de React. No se utiliza un gestor de estado externo (como Redux o Zustand), ya que el alcance del estado es local al módulo de visualización.

3.3.1 Variables de estado

El componente principal mantiene las siguientes variables de estado:

Variable	Tipo	Descripción
`nodes`	`Node[]`	Todos los nodos (lista + punteros)
`edges`	`Edge[]`	Todas las aristas
`count`	`number`	Cantidad de nodos para creación aleatoria
`speed`	`number`	Velocidad de animación (ms entre fotogramas)
`isRunning`	`boolean`	Indica si una operación está en ejecución
`operation`	`number`	Índice de la operación seleccionada
`hoveredNodeId`	`string | null`	ID del nodo bajo el cursor
`highlightHead`	`boolean`	Resaltar nodo cabeza
`highlightTail`	`boolean`	Resaltar nodo cola
`nodeColor`	`string`	Color base de los nodos
`newNodeColor`	`string`	Color de nodos recién insertados
`iterateColor`	`string`	Color del nodo activo durante animación
`showPointers`	`boolean`	Mostrar/ocultar punteros head y tail
`autoAdjust`	`boolean`	Ajustar lienzo automáticamente
`selectedNodes`	`string[]`	IDs de nodos seleccionados
`laserPointerEnabled`	`boolean`	Activar puntero láser
`lengthResult`	`number | null`	Resultado de la operación *Get Length*
`searchResult`	`object | null`	Resultado de la operación *Search*

3.3.2 Patrón `OperationContext`

Para evitar pasar múltiples parámetros a cada operación, se utiliza un objeto de contexto que encapsula todo lo necesario:

```
interface OperationContext {
  state: OperationState;
  setState: (updates: NodeEdgeUpdate | StateUpdater,
             callback?: () => void) => void;
  reactFlowInstance?: ReactFlowInstance;
  handlePointerHover: (nodeId: string | null) => void;
```

```
}

interface OperationState {
  nodes: Node[];
  edges: Edge[];
  speed: number;
  iterateColor: string;
  newNodeColor?: string;
}
```

Este patrón permite que las operaciones sean funciones puras que reciben un contexto y lo modifican, sin depender de variables externas. El *hook* `useListOperations` se encarga de construir el contexto a partir del estado actual de React.

3.4 Constantes de configuración

Todas las constantes de configuración están centralizadas en el archivo `src/app/sll/constants.ts`. Las principales categorías son:

3.4.1 Disposición (`LAYOUT`)

Constante	Valor	Descripción
`NODE_HORIZONTAL_SPACING`	150px	Separación horizontal entre nodos
`POINTER_VERTICAL_OFFSET`	100px	Distancia vertical de los punteros
`INITIAL_X`	50px	Posición X inicial
`INITIAL_Y`	100px	Posición Y inicial
`FIT_VIEW_PADDING`	0.3	Padding del ajuste de vista

3.4.2 Colores (`COLORS`)

Constante	Valor	Uso
`NODE_DEFAULT`	`#2196F3`	Color base de los nodos (azul)
`NODE_NEW`	`#4CAF50`	Nodos recién insertados (verde)
`NODE_ITERATE`	`#FF5722`	Nodo activo en animación (naranja)
`HEAD_HIGHLIGHT`	`#9C27B0`	Resaltado de cabeza (púrpura)
`TAIL_HIGHLIGHT`	`#E91E63`	Resaltado de cola (rosa)
`EDGE_DEFAULT`	`#333`	Color de las aristas

3.4.3 Animación (`ANIMATION`)

Constante	Valor	Descripción
`DEFAULT_SPEED`	500ms	Velocidad predeterminada
`FIT_VIEW_DURATION`	500ms	Duración de la transición del lienzo
`FIT_VIEW_DELAY`	100ms	Retardo antes del ajuste de vista

3.4.4 Operaciones (`OPERATIONS`)

Las 16 operaciones disponibles están indexadas numéricamente:

Índice	Nombre	Categoría
0	`INSERT_HEAD`	Inserción
1	`DELETE_HEAD`	Borrado
2	`INSERT_TAIL`	Inserción
3	`DELETE_TAIL`	Borrado
4	`TRAVERSE`	Algoritmo
5	`REVERSE`	Algoritmo
6	`INSERT_TAIL_O1`	Inserción
7	`INSERT_AT_POSITION`	Inserción
8	`GET_LENGTH`	Algoritmo
9	`SEARCH_VALUE`	Búsqueda
10	`FIND_MIDDLE`	Algoritmo
11	`DELETE_AT_POSITION`	Borrado
12	`REMOVE_DUPLICATES`	Algoritmo
13	`ACCESS_FRONT`	Acceso
14	`ACCESS_BACK`	Acceso
15	`ACCESS_NTH`	Acceso

4. Motor de Visualización

Este capítulo describe cómo la aplicación transforma el modelo de datos (descrito en Capítulo 3.) en una representación visual interactiva.

4.1 Integración con xyflow (React Flow)

La visualización se basa en la biblioteca `@xyflow/react` (React Flow v12), que proporciona un lienzo interactivo para grafos de nodos y aristas.

4.1.1 Registro de tipos de nodos

La aplicación registra dos tipos de nodos personalizados:

```
const nodeTypes = {
  linkedListNode: LinkedListNode,
  circleNode: CircleNode,
};
```

Estos componentes se definen en `src/app/sll/LinkedListNode.tsx` y `src/app/sll/CircleNode.tsx` respectivamente. Cada uno implementa su propia lógica de renderizado y manejo de eventos.

4.1.2 Sistema de *handles*

Los *handles* son los puntos de conexión que permiten trazar aristas entre nodos. El componente `LinkedListNode` define cuatro *handles*:

- **Right** (`Position.Right`, tipo *source*): representa el puntero `next` del nodo. Es el punto de salida para la arista hacia el siguiente elemento.
- **Left** (`Position.Left`, tipo *target*): punto de entrada desde el nodo anterior.
- **Top** (`Position.Top`, tipo *target*): utilizado por el puntero *head* para conectarse desde arriba.
- **Bottom** (`Position.Bottom`, tipo *target*): utilizado por el puntero *tail* para conectarse desde abajo.

4.1.3 Validación de conexiones

La función `isValidConnection` en `page.tsx` controla qué conexiones son válidas:

- Los nodos circulares solo pueden conectarse a los *handles* `top` o `bottom` de un nodo de lista.
- Se previene la conexión múltiple: si un nodo circular ya tiene una arista saliente, no se permite una segunda (excepto durante reconexión).

4.1.4 Interacción con el lienzo

ReactFlow se configura con las siguientes opciones de interacción:

- **Nodos arrastrables**: el usuario puede reposicionar cualquier nodo.
- **Selección parcial**: permite seleccionar nodos arrastrando un rectángulo.
- **Desplazamiento con scroll**: el lienzo se desplaza con la rueda del ratón.
- **Reconexión de aristas**: las aristas existentes pueden ser redirigidas a otros nodos.

4.2 Sistema de animaciones

Las animaciones son el componente central de la experiencia pedagógica. Cada operación sobre la lista se visualiza paso a paso para que el estudiante pueda seguir la ejecución del algoritmo.

4.2.1 Patrón de animación

Todas las operaciones siguen el mismo patrón asíncrono:

```
async function operacion(context: OperationContext, ...params) {
  const { state, setState } = context;
  const listNodes = getListNodes(state.nodes);

  // Paso 1: modificar nodos/aristas
  const updatedNodes = listNodes.map(node => /* ... */);
  setState({ nodes: updatedNodes });
  await sleep(state.speed);

  // Paso 2: siguiente modificación
  // ...

  // Paso final: restaurar colores
  setState({ nodes: resetColors(updatedNodes) });
}
```

La función `sleep` (definida en `src/app/sll/utils/helpers.ts`) es una promesa que se resuelve después de un número configurable de milisegundos, creando la pausa visual entre fotogramas.

4.2.2 Ejemplo: inserción en la cabeza

La operación `insertAtHead` (`src/app/sll/operations/insertAtHead.ts`) ilustra el patrón completo:

1. Filtra los nodos de lista mediante `getListNodes()`.
2. Calcula la posición X del nuevo nodo: `primerNodo.x - 150`.
3. Crea el nuevo nodo con el color `newNodeColor` usando `createListNode()`.
4. Inserta el nodo al inicio del arreglo de nodos.
5. Reconstruye todas las aristas con `createEdgesForList()`.
6. Actualiza las posiciones y aristas de los punteros *head* y *tail*.
7. Llama a `setState()` para actualizar la visualización.
8. Ejecuta `sleep(speed)` para la pausa animada.

4.2.3 Ejemplo: recorrido de la lista

La operación `traverseList` (`src/app/sll/operations/traverseList.ts`) muestra la animación iterativa:

1. Restablece todos los nodos al color predeterminado.
2. Itera sobre cada nodo de la lista:
   - Resalta los nodos visitados (del primero al actual) con `iterateColor`.
   - Llama a `setState()` con los nodos actualizados.
   - Espera `sleep(speed)` milisegundos.
3. Al finalizar, restablece todos los colores al valor predeterminado.

4.2.4 Ejemplo: inversión de la lista

La operación `reverseList` (`src/app/sll/operations/reverseList.ts`) es más compleja y consta de tres fases:

1. **Inversión de punteros**: para cada nodo, se elimina la arista saliente y se crea una nueva apuntando al nodo anterior. Se resaltan los nodos involucrados.
2. **Reposicionamiento**: se invierte el orden del arreglo de nodos y se recalculan las posiciones X.
3. **Reconstrucción de aristas**: se crean las aristas finales y se actualizan los punteros *head* y *tail*.

4.2.5 Patrones comunes en las operaciones

Todas las operaciones comparten estos patrones:

- **Filtrado de nodos**: `getListNodes()` para separar nodos de lista de punteros circulares.
- **Reconstrucción de aristas**: `createEdgesForList()` después de cambios estructurales.

- **Actualización de punteros**: `updatePointers()` y `createPointerEdges()` para mantener los punteros sincronizados.
- **Animación por color**: uso de `.map()` sobre el arreglo de nodos para aplicar colores de resaltado según el índice de iteración.

4.3 Disposición automática con ELK

La disposición automática de los nodos se implementa mediante la biblioteca **ELK.js**, que proporciona algoritmos de disposición de grafos.

4.3.1 Algoritmo utilizado

Se utiliza el algoritmo **layered** (Sugiyama), configurado con dirección `RIGHT` para una disposición horizontal de izquierda a derecha. Este algoritmo es especialmente adecuado para estructuras lineales como las listas enlazadas.

La configuración se encuentra en `src/app/sll/utils/elkLayout.ts`:

```
const elk = new ELK();

const graph = {
  id: 'root',
  layoutOptions: {
    'elk.algorithm': 'layered',
    'elk.direction': 'RIGHT',
    'elk.spacing.nodeNode': '150',
  },
  children: nodes.map(node => ({
    id: node.id,
    width: nodeWidth,
    height: nodeHeight,
  })),
  edges: edges.map(edge => ({
    id: edge.id,
    sources: [edge.source],
    targets: [edge.target],
  })),
};
```

4.3.2 Posicionamiento de punteros

Después de calcular las posiciones de los nodos de lista, los nodos punteros (*head* y *tail*) se posicionan relativamente:

- **Head**: se ubica 100px por encima del primer nodo de la lista.
- **Tail**: se ubica 100px por debajo del último nodo de la lista.

Este cálculo se realiza mediante las funciones `getHeadPointerPosition` y `getTailPointerPosition` de `src/app/sll/utils/pointerHelpers.ts`.

4.3.3 Activación

La disposición automática se activa de dos maneras:

- **Botón *Auto Layout***: en el menú *Display Options*, el usuario puede ejecutar manualmente la disposición.
- **Después de operaciones**: si la opción `autoAdjust` está habilitada, el lienzo se ajusta automáticamente después de cada operación mediante `reactFlowInstance.fitView()`.

4.4 *Hook* de visualización

El *hook* `useListVisualization` (`src/app/sll/hooks/useListVisualization.ts`) actúa como capa intermedia entre el estado bruto y lo que se renderiza en ReactFlow. Sus responsabilidades son:

1. **Filtrar punteros**: cuando `showPointers` es `false`, se eliminan los nodos circulares y sus aristas del conjunto renderizado.
2. **Resaltar cabeza y cola**: cuando las opciones `highlightHead` o `highlightTail` están activas, se modifican los estilos de los nodos correspondientes.
3. **Resaltar aristas al pasar el cursor**: cuando el usuario posiciona el cursor sobre la sección de puntero de un nodo, la arista saliente se resalta con el color de iteración.

El *hook* utiliza `useMemo` para evitar recálculos innecesarios y retorna los arreglos `highlightedNodes` y `highlightedEdges` que se pasan directamente a ReactFlow.

5. Interfaz de Usuario

Este capítulo describe el diseño de los componentes de la interfaz de usuario, el sistema de menús y la estrategia de estilización.

5.1 Componentes principales

La interfaz se divide en tres áreas principales, como se muestra en la Figura 5.1:

1. **Barra de navegación** (parte superior): enlaces a documentación, repositorio y tour guiado.
2. **Panel de menú** (izquierda): controles para la creación, operaciones y configuración de la visualización.
3. **Lienzo** (derecha): área de visualización interactiva con ReactFlow.

Figura 5.1: Partes principales de la interfaz de usuario.

5.1.1 Componente orquestador (`page.tsx`)

El archivo `src/app/sll/page.tsx` es el componente principal del módulo de listas enlazadas. Sus responsabilidades son:

- Gestionar las 18 variables de estado (ver Sección 3.3).
- Coordinar los tres *custom hooks* (`useListOperations`, `useListVisualization`, `useListInitialization`).
- Definir los 22 manejadores de eventos que conectan el menú con las operaciones.
- Renderizar la estructura de la página: `Navbar`, `Menu` y el lienzo `ReactFlow`.

5.1.2 Nodo de lista (`LinkedListNode`)

El componente `LinkedListNode` (`src/app/sll/LinkedListNode.tsx`) renderiza cada elemento de la lista enlazada como un rectángulo dividido en dos secciones:

```
┌──────────────────────┬──────────────┐
│  Sección de datos    │  Puntero     │
│  (valor del nodo)    │  (next →)    │
└──────────────────────┴──────────────┘
```

- **Sección de datos** (70% del ancho): muestra el valor del nodo con un fondo semitransparente blanco sobre el color de fondo del nodo.
- **Sección de puntero** (30% del ancho): representa el puntero `next`. Al posicionar el cursor sobre esta sección, el fondo cambia a rojo (`#F44336`) y la arista saliente se resalta.

Las dimensiones mínimas son 72x36 píxeles, con bordes redondeados y un divisor vertical entre las dos secciones. Figura 5.2 muestra un ejemplo de nodo de lista enlazada renderizado.

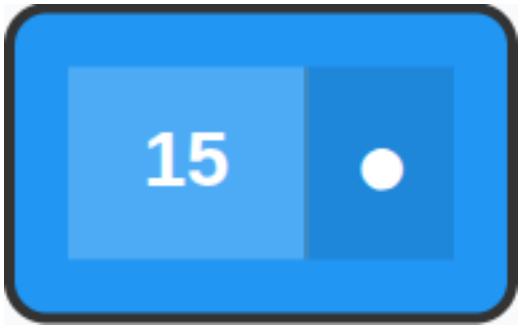

Figura 5.2: Ejemplo de nodo de datos en el lienzo.

5.1.3 Nodo circular (`CircleNode`)

El componente `CircleNode` (`src/app/sll/CircleNode.tsx`) renderiza los nodos de tipo puntero como círculos de 45x45 píxeles con fondo naranja (`#FF5722`).

Características interactivas:

- **Doble clic**: activa el modo de edición de la etiqueta.
- **Entrada de texto**: máximo 3 caracteres. Se confirma con Enter o al perder el foco, y se cancela con Escape.
- **Hover**: al posicionar el cursor sobre el nodo, se invoca `onPointerHover` para resaltar la arista conectada.

Figura 5.3: Ejemplo de nodo circular para señalar el primer elemento de la lista.

5.2 Sistema de menús

El panel de menú se implementa como un componente jerárquico compuesto por varias capas.

5.2.1 Estructura del menú

```
Menu (panel lateral, 288px de ancho)
├── MenuHeader (encabezado animado, fijo)
├── ListCreation (sección colapsable)
│   ├── Pestaña Empty    → botón "Create Empty List"
│   ├── Pestaña Random   → slider de cantidad + "Generate Random"
│   └── Pestaña Custom   → entrada JSON + "Create From Sequence"
├── Operations (sección colapsable)
│   ├── Insert           → At Head, At Position, At Tail, At Tail (O1)
│   ├── Remove           → Head, Tail, Position
│   ├── Access           → Front, Back, Nth
│   ├── Search           → entrada de valor + botón
```

```
│   └── Algorithms       → Traverse, Reverse, Find Middle,
│                            Get Length, Remove Duplicates
├── Programmer Tools (sección colapsable)
│   ├── Show head and tail pointers
│   └── Add pointer
└── Display Options (sección colapsable)
    ├── Speed             → slider (10-100, mapeado a 10-1000ms)
    ├── Colors            → selectores de color (Standard, New, Active)
    ├── Highlights        → Head / Tail toggles
    ├── Layout            → Scramble + Auto Layout
    └── Canvas            → Auto Adjust toggle
```

5.2.2 Componente `Section`

El componente reutilizable `Section` (`src/components/menu/Section.tsx`) implementa una sección colapsable con:

- Un encabezado que muestra un icono (Lucide), un título y un indicador de chevron que rota al expandir/colapsar.
- Animación suave de altura máxima (*max-height*) para la transición.
- Soporte para estado controlado y no controlado mediante las props `isOpen` y `defaultOpen`.

5.2.3 Componente `MenuHeader`

El encabezado del menú (`src/components/menu/MenuHeader.tsx`) incluye animaciones basadas en el progreso de desplazamiento (*scroll*):

- Tamaño dinámico del icono que disminuye al desplazarse.
- Opacidad y rotación que cambian según la posición de scroll.
- Una barra de progreso con gradiente que indica la posición actual.

5.2.4 Sección de operaciones

La sección `Operations` (`src/components/menu/sections/Operations.tsx`) organiza las 16 operaciones en 5 subsecciones colapsables. Cada subsección sigue un patrón consistente:

1. **Entrada de valor**: campo de texto con botón de valor aleatorio (para operaciones que requieren un valor).
2. **Entrada de posición**: campo numérico con validación de rango (para operaciones que requieren una posición).
3. **Botones de acción**: uno por cada operación disponible.

Los botones de acción invocan `onVisualize(opIndex, value, position)`, que se propaga hasta el componente principal `page.tsx`.

5.3 Estilización con Tailwind CSS y Radix UI

5.3.1 Tailwind CSS

La aplicación utiliza **Tailwind CSS** con un enfoque *utility-first*. Las clases de utilidad se aplican directamente en los componentes JSX, evitando la necesidad de hojas de estilo separadas.

Para la gestión de clases condicionales se utiliza la función `cn()` definida en `src/lib/utils.ts`, que combina `clsx` (para lógica condicional) y `tailwind-merge` (para resolver conflictos entre clases de Tailwind):

```
import { clsx, type ClassValue } from "clsx";
import { twMerge } from "tailwind-merge";

export function cn(...inputs: ClassValue[]) {
  return twMerge(clsx(inputs));
}
```

5.3.2 Sistema de colores

El sistema de colores se basa en variables CSS con valores HSL, lo que permite una personalización global del tema. Las variables se definen en `src/app/globals.css`:

- `--background`, `--foreground`: colores base del fondo y texto.
- `--primary`, `--secondary`, `--accent`: colores de énfasis.
- `--destructive`: color para acciones destructivas.
- `--border`, `--ring`: colores de bordes y anillos de foco.

5.3.3 Componentes shadcn/ui

Los componentes atómicos de la interfaz (botones, entradas, *sliders*, *switches*, selectores) provienen de **shadcn/ui** y se encuentran en `src/components/ui/`. Estos componentes están construidos sobre primitivas de **Radix UI**, que proporcionan accesibilidad y comportamiento estandarizado.

Los componentes personalizados (`custom-slider.tsx`, `custom-select.tsx`, `custom-toggle.tsx`, `custom-input.tsx`) envuelven los componentes de shadcn/ui con etiquetas, valores de visualización y estilos específicos del visualizador.

5.3.4 Diseño adaptativo

La interfaz utiliza un diseño de dos columnas con `flex`:

- El panel de menú tiene un ancho fijo de 288px con scroll vertical independiente.
- El lienzo ocupa el espacio restante (`flex-1`) y se adapta al tamaño de la ventana.

La barra de navegación es fija en la parte superior y el contenido principal ocupa la altura restante de la ventana (`h-screen`).

6. Despliegue y CI/CD

Este capítulo documenta el proceso de construcción, despliegue y el flujo de trabajo de integración continua del proyecto.

6.1 GitHub Pages

La aplicación se despliega como un sitio estático en **GitHub Pages**. Esto es posible gracias a la configuración de exportación estática de Next.js.

6.1.1 Configuración de Next.js

El archivo `next.config.mjs` define las siguientes opciones clave para el despliegue:

- **`output: 'export'`**: indica a Next.js que genere una exportación estática (aplicación de una sola página o SPA) en lugar de un servidor Node.js.
- **`basePath: '/AlgorithmVisualizer'`**: prefijo de ruta necesario para GitHub Pages, ya que el sitio se sirve desde un subdirectorio del dominio `github.io`.
- **`assetPrefix: '/AlgorithmVisualizer'`**: asegura que los recursos estáticos (JavaScript, CSS, imágenes) se carguen con la ruta correcta.
- **`distDir: './build'`**: directorio de salida de la construcción.

6.1.2 Enrutamiento del lado del cliente

Al ser una aplicación de una sola página (SPA), toda la navegación se maneja en el navegador. El archivo `public/_redirects` contiene la regla:

```
/* /index.html 200
```

Esta regla redirige todas las solicitudes al archivo `index.html`, permitiendo que el enrutador de Next.js maneje las rutas del lado del cliente.

6.2 Flujo de trabajo con GitHub Actions

El despliegue automatizado se configura en el archivo `.github/workflows/nextjs.yml`. El flujo se activa con cada *push* a la rama `master`.

6.2.1 Etapas del flujo

El *workflow* consta de dos *jobs* secuenciales:

6.2.1.1 1. Construcción (`build`)

1. **Checkout**: descarga el código fuente del repositorio.
2. **Detección del gestor de paquetes**: identifica automáticamente si el proyecto usa `npm` o `yarn` verificando la existencia de archivos de bloqueo.
3. **Configuración de Node.js**: instala Node.js v20 y configura el caché de dependencias.
4. **Configuración de Pages**: inyecta automáticamente el `basePath` en la configuración de Next.js y desactiva la optimización de imágenes del lado del servidor.
5. **Restauración de caché**: utiliza el caché de `.next/cache` para acelerar construcciones incrementales.
6. **Instalación de dependencias**: ejecuta `npm ci` para una instalación determinista.
7. **Construcción**: ejecuta `next build`, que compila TypeScript y genera la exportación estática en el directorio `./build`.
8. **Subida del artefacto**: empaqueta el directorio `./build` como artefacto de GitHub Pages.

6.2.1.2 2. Despliegue (`deploy`)

1. Descarga el artefacto generado en la etapa anterior.
2. Lo despliega en el entorno de GitHub Pages.
3. Genera la URL pública del sitio.

6.2.2 Control de concurrencia

El *workflow* utiliza un grupo de concurrencia (`pages`) que permite solo un despliegue simultáneo. Los despliegues en curso no se cancelan para evitar estados intermedios.

6.3 Proceso de construcción

6.3.1 Construcción de la aplicación

El comando de construcción principal es:
`npm run build`
Este ejecuta dos pasos secuenciales:

1. **`tsc`**: compila TypeScript y verifica que no existan errores de tipos.
2. **`next build`**: genera la exportación estática de la aplicación.

El resultado es el directorio `./build` que contiene todos los archivos HTML, JavaScript y CSS necesarios para servir la aplicación.

6.3.2 Construcción de la documentación

La documentación se construye por separado mediante Quarto:
`npm run docs:build`
Este comando ejecuta:

1. `cd doc/manual && quarto render`: genera la documentación del manual de usuario en HTML y PDF.
2. `cd doc/design && quarto render`: genera el documento de diseño en HTML y PDF.
3. `npm run docs:sync`: copia los resultados a `public/docs/manual/` y `public/docs/design/` respectivamente.

La documentación debe construirse **antes** de la construcción de la aplicación, ya que los archivos generados en `public/docs/` se incluyen en la exportación estática de Next.js.

6.3.3 Flujo completo de despliegue

Para un despliegue completo que incluya documentación actualizada:

```
# 1. Construir la documentación
npm run docs:build

# 2. Construir la aplicación (incluye public/docs/)
npm run build

# 3. Confirmar y enviar los cambios
git add .
git commit -m "Update build"
git push
```

El *push* a `master` activa automáticamente el *workflow* de GitHub Actions que despliega el sitio.

6.4 Estructura de URLs desplegadas

Una vez desplegada, la aplicación es accesible en las siguientes rutas:

Ruta	Contenido
`/AlgorithmVisualizer/sll`	Visualizador de listas
`/AlgorithmVisualizer/about`	Página «Acerca de»

Ruta	Contenido
`/AlgorithmVisualizer/docs/manual/index.html`	Manual de usuario (HTML)
`/AlgorithmVisualizer/docs/design/index.html`	Documento de diseño (HTML)