Manual Técnico

A continuación, se detalla la especificación técnica de las estructuras de datos fundamentales implementadas para el Sistema de Red de Bibliotecas Mágicas.

Estructuras de Almacenamiento Lineal (Filas y Pilas)

Cola <T> (Clase Cola)

El TDA **Cola** gestiona el flujo de trabajo de la biblioteca (Ingreso, Traspaso, Salida) de forma concurrente, siguiendo la política **FIFO** (First-In, First-Out).

Aspecto	**Descripción**
Implementación	Utiliza una **Lista Enlazada Simple** interna, definida por el nodo frente (cabeza) y el nodo fin (cola).
Mecanismo de Procesamiento	Cada instancia de Cola opera en un **hilo** separado (hiloProcesamiento). El método iniciarProcesamiento() ejecuta un bucle que espera el tiempo definido por tiempoProcesamiento antes de invocar un **callback** a la interfaz **Procesador<T>**.
Métodos Clave	encolar(T dato): Inserta un nuevo nodo al final de la lista. desencolar(): Remueve y retorna el nodo en frente.
Concurrencia	El uso de un **Procesador** desacopla la lógica de movimiento (CoordinadorEnvios) de la lógica de tiempo (Cola), permitiendo un flujo asíncrono.

Pila <T> (Clase Pila)

El TDA **Pila** se utiliza para registrar el historial de préstamos en cada Biblioteca, aplicando la política **LIFO** (Last-In, First-Out).

Aspecto	Descripción
Implementación	Utiliza una **Lista Enlazada Simple** interna, donde la única referencia mantenida es al nodo superior (tope).
Métodos Clave	apilar(T dato): Inserta un nuevo nodo en la posición del tope, reasignando el tope al nuevo nodo. desapilar(): Remueve y retorna el nodo en tope, reasignando el tope al nodo siguiente.
Propósito	Almacenar objetos **RegistroPrestamo** de forma eficiente para la recuperación rápida de la acción más reciente.

Estructura de Acceso Directo (Hash)

TablaHash <K, V> (Clase TablaHash)

La **Tabla Hash** es la estructura principal de acceso directo y se utiliza para el **Catálogo de la Biblioteca** (indexación por ISBN) y como el **Almacén de Vértices** en el GrafoBibliotecas (indexación por ID).

Aspecto	Descripción
Implementación	Utiliza un **arreglo de cubetas** (Entrada<K, V>[]). Cada cubeta soporta la colisión mediante **Encadenamiento** (lista enlazada simple de objetos Entrada).
Función Hash	calcularHash(K clave): Implementa un **Hash Polinomial** manual, utilizando un número primo (31) para calcular un valor entero, el cual se modula por la capacidad de la tabla para obtener el índice final.

Manejo de Colisiones	Si dos claves caen en la misma cubeta, el nuevo elemento se añade al **final de la lista enlazada** ya existente en ese índice del arreglo.
Redimensionamiento	La tabla se redimensiona (duplica su capacidad) cuando el **Factor de Carga** (elementos / capacidad) excede el umbral de **0.75**. Todos los elementos deben ser **rehasheados** con la nueva capacidad.

Estructuras de Grafo (Red Logística)

ListaAdyacencia (Clase ListaAdyacencia)

La **Lista de Adyacencia** se utiliza dentro del Vertice para almacenar todas las conexiones salientes.

Aspecto	Descripción
Implementación	Es una **Lista Enlazada Simple** que solo almacena objetos **Arista**.
Propósito	Permite a cada Vertice saber a qué otros vértices puede enviar libros, almacenando las aristas con sus pesos (tiempo, costo).
Iterador	Proporciona una **Clase Interna IteradorLista** que permite recorrer las aristas sin exponer la estructura interna de los nodos de la lista.

Vertice (Clase Vertice)

El **Vértice** es el nodo del grafo que envuelve el objeto del modelo de negocio.

Aspecto	Descripción
Implementación	Es una clase contenedora que envuelve una **Biblioteca** y posee una **ListaAdyacencia** con las conexiones salientes.

Identificación	Obtiene su identidad (id) y la lógica de negocio directamente del objeto **Biblioteca** que contiene.
Propósito	Sirve como el puente entre el modelo (Biblioteca) y la estructura del grafo, permitiendo que el GrafoBibliotecas lo almacene por ID.

Estructuras de Índice Avanzado (Árboles)

ArbolAVL (Clase ArbolAVL)

El **Árbol AVL** es el índice primario para la búsqueda rápida por **Título** de los libros en la biblioteca.

Aspecto	Descripción
Implementación	Árbol Binario de Búsqueda **Auto-balanceado** (AVL). La raíz es de tipo **NodoAVL**.
Balanceo	Mantiene el **Factor de Equilibrio (FE)** de cada nodo en $\{-1, 0, 1\}$ mediante **Rotaciones Simples y Dobles** (delegadas a InsertarAVL y EliminarAVL).
Búsqueda	Permite una búsqueda eficiente ($O(\log n)$) por título. Implementa buscarTodosPorTitulo para manejar múltiples copias del mismo título (o títulos con el mismo prefijo).

ArbolB (Clase ArbolB)

El **Árbol B** es un índice multi-vía y multi-nivel diseñado para la búsqueda eficiente por **Rango de Fechas**.

Aspecto	Descripción
Implementación	Estructura de árbol de disco, optimizada para manejar grandes volúmenes de claves en cada **NodoB**.

Claves Compuestas	Cada clave (ClaveB) en un nodo B almacena un **valor entero (fecha/año)** y un índice secundario (IndiceISBN).
Índice Secundario	Permite que muchos libros compartan la misma fecha. La clave solo almacena la fecha, y un objeto **IndiceISBN** asociado gestiona todos los libros con esa fecha específica (una forma de manejar colisiones de claves).
Búsqueda	El método buscarPorRango recorre eficientemente solo las ramas relevantes del árbol para devolver libros entre una fechaInicio y fechaFin.

ArbolBPlus (Clase ArbolBPlus)

El **Árbol B+** es la estructura utilizada para la indexación por **Género**, optimizada para el recorrido secuencial.

Aspecto	**Descripción**
Implementación	Separa los **NodoInterno** (solo claves para navegación) de los **NodoHoja** (claves y datos).
Secuencialidad	Los **NodoHoja** están enlazados secuencialmente mediante el puntero **siguienteHoja** a través de la variable primeraHoja.
Propósito	Permite la búsqueda rápida por género y, lo más importante, facilita el **recorrido secuencial** de todos los libros dentro de un mismo rango o categoría (todos los libros de un género, todos los géneros, etc.).
Dato Almacenado	Las hojas almacenan una lista o índice (IndiceLibrosEnGenero) que referencia a los libros que pertenecen a ese género.

Componentes Esenciales del Grafo (Red Logística)

Arista (Clase Arista)

La **Arista** representa una conexión dirigida entre dos bibliotecas, con dos criterios de "peso" para el algoritmo de Dijkstra.

Aspecto	Descripción
Implementación	Clase simple que encapsula los atributos de la conexión.
Atributos Clave	**idOrigen** y **idDestino** (identificadores de las bibliotecas conectadas). **tiempo** (costo en segundos) y **costo** (costo monetario o logístico).
Propósito	Almacena la información necesaria para el cálculo de la ruta óptima por el **RutaDijkstra**.

RutaDijkstra (Clase RutaDijkstra)

Esta clase estática implementa el algoritmo de búsqueda de ruta más corta.

Aspecto	Descripción
Implementación	Clase estática que contiene la lógica del **Algoritmo de Dijkstra**.
Funcionalidad	Calcula la ruta más corta entre un origen y un destino en el GrafoBibliotecas, utilizando como peso la **Arista.tiempo** o la **Arista.costo**, según el Criterio especificado.
Proceso Clave	Itera sobre todos los nodos, seleccionando repetidamente el nodo **no visitado** con la distancia mínima actual para relajar a sus vecinos.

GrafoBibliotecas (Clase GrafoBibliotecas)

El **Grafo Dirigido** principal que modela la red logística de las bibliotecas.

Aspecto	Descripción
Implementación	Utiliza el patrón de **Lista de Adyacencia**. El almacenamiento principal es una **TablaHash<String, Vertice>**.
Manejo de Nodos	Las bibliotecas se almacenan como **Vertice**, indexadas por su ID en la TablaHash, permitiendo el acceso O(1) (promedio).
Operaciones Clave	**agregarBiblioteca**, **conectarBibliotecas**, **eliminarBiblioteca** (requiere eliminar conexiones salientes y entrantes), y **getConexionesSalientes** (obtenido del Vertice).

Clases de Rendimiento y Algoritmos de Ordenamiento

Catalogo (Clase Catalogo)

El **Catálogo** es un TAD simple diseñado para representar la colección de libros sin indexar, sirviendo como la **Línea Base de Rendimiento** contra las estructuras de índice avanzadas.

Aspecto	Descripción
Implementación	Clase contenedora simple que gestiona una lista interna de objetos **Libro**.
Búsqueda Secuencial	Implementa de forma manual los métodos **buscarTituloSecuencial** y **buscarISBNSecuencial**. Estos recorren la lista elemento por elemento (búsqueda $O(n)$) para medir la diferencia de rendimiento con **ArbolAVL** (O(\log n)) y **TablaHash** (O(1)).

Propósito	Garantizar que las pruebas de rendimiento en **PruebaRendimiento** tengan un punto de comparación justo para demostrar la eficiencia de los TADs avanzados.

Ordenamientos (Clase Ordenamientos)

Clase estática que contiene la implementación de los principales algoritmos de ordenamiento, usados para comparar eficiencia.

Aspecto	Descripción
Implementación	Clase estática con métodos que reciben una List<Libro> y un **Comparator<Libro>** (interfaz del lenguaje, análogo a una función comparadora implementada manualmente).
Algoritmos	Implementa versiones manuales y eficientes de: **bubbleSort** (O(n^2)), **selectionSort** (O(n^2)), **insertionSort** (O(n^2)), **shellSort** (O(n^{3/2}) a O(n\log^2 n)), y **quickSort** (O(n \log n) promedio).
Función ordenar	Cada método devuelve el **tiempo total en nanosegundos** que tardó la ejecución del algoritmo, permitiendo la comparación directa en PanelOrdenamientos.

Vistas Avanzadas y Patrones de Control (GUI/Validación)

Clase de Control/Vista: PanelTraficoLibros2

Vista de la GUI especializada en el monitoreo en tiempo real de la simulación logística.

Aspecto	Descripción
Patrón Implementado	**Observador (Listener)**. PanelTraficoLibros2 implementa la interfaz **EnvioListener** y se registra en el **CoordinadorEnvios** (el sujeto).

Flujo de Datos	El CoordinadorEnvios notifica al PanelTraficoLibros2 cada vez que un libro cambia de estado (cola de entrada, cola de traspaso, en tránsito).
Lógica Interna	Utiliza un **Map** interno para correlacionar los mensajes de texto del evento con el objeto **Libro** en tránsito y actualizar las coordenadas de ubicación.
Propósito	Monitorear el estado de los objetos **Libro** mientras se mueven a través del **GrafoBibliotecas** y las colas internas de cada Biblioteca.

Clase de Control/Vista: MainWindowGrafo

Vista especializada para la visualización de la topología del grafo.

Aspecto	**Descripción**
Algoritmo Implementado	**Layout de Fuerza Dirigida (Force-Directed Layout)**. Este algoritmo no es una estructura de datos, sino un método de visualización.
Lógica	Simula fuerzas físicas: **Repulsión** entre todos los nodos para que no se superpongan, y **Atracción** a lo largo de las Aristas para que los nodos conectados se mantengan cerca.
TADs Usados	Hace uso intensivo de la **TablaHash** del grafo y el **Iterador** de la ListaAdyacencia para calcular las fuerzas en cada iteración.
Objetivo	Proporcionar una representación visual clara de la **GrafoBibliotecas** donde la posición de los nodos refleja las conexiones estructurales.

Clase de Control/Vista: PruebaRendimiento

Módulo de validación diseñado para comparar la eficiencia de las estructuras de índice.

Aspecto	Descripción
Propósito	Medir y comparar el tiempo de ejecución (en microsegundos) de la búsqueda en los TADs avanzados (ArbolAVL, TablaHash) contra la **Búsqueda Secuencial** proporcionada por el **Catálogo**.
Metodología	Utiliza la función **medirTiempo** para envolver la ejecución de la función de búsqueda y obtener el tiempo transcurrido en nanosegundos.
Validación	Demuestra de manera práctica la ganancia de rendimiento (complejidad O(\log n) y O(1)) sobre los métodos de complejidad O(n) en colecciones no indexadas.

Clase de Control/Vista: PanelOrdenamientos

Módulo de validación diseñado para comparar la eficiencia de los algoritmos de ordenamiento.

Aspecto	Descripción
Propósito	Ejecutar una comparativa de tiempo de los algoritmos **bubbleSort**, **selectionSort**, **insertionSort**, **shellSort** y **quickSort** (implementados en la clase **Ordenamientos**).
Ejecución Concurrente	Las pruebas se ejecutan en un **hilo separado** (Thread) para evitar que la interfaz de usuario se congele (bloqueo de UI), utilizando SwingUtilities.invokeLater para actualizar los resultados de forma segura.
Criterio	Utiliza la interfaz **Comparator** del lenguaje para abstraer el criterio de ordenamiento (Título, Autor, Año), permitiendo que los mismos algoritmos se usen para cualquier tipo de dato.

Análisis Big-O y justificación de estructuras

Justificación Estructural por Contexto de Uso

El diseño de la arquitectura se basa en dos pilares: la **Logística de la Red (Grafo)** y la **Eficiencia de la Búsqueda (Índices)**. Cada estructura de datos implementada a mano fue elegida por su superioridad operativa en el contexto específico de la biblioteca o la red.

TDA Implementado	Contexto de Uso en el Sistema	Justificación de la Elección
ArbolAVL	Índice primario de **Búsqueda por Título** (Biblioteca).	Garantiza el **balanceo automático** tras inserciones/eliminaciones, asegurando que la búsqueda siempre se mantenga en **O(\log n)** (logarítmica), incluso en el peor caso. Es fundamental para búsquedas internas.
TablaHash	**Catálogo/Índice por ISBN** (Biblioteca) y **Almacén de Vértices** (GrafoBibliotecas).	Proporciona el **acceso más rápido** (O(1) promedio) a los nodos o libros usando sus claves únicas (ID de Biblioteca, ISBN). Es ideal para la recuperación directa sin necesidad de orden.
ArbolB	Índice de **Búsqueda por Rango de Fechas** (Biblioteca).	Estructura optimizada para manejar **grandes volúmenes de claves** en un entorno que simula un acceso lento (lectura de disco). Permite la búsqueda y el **recorrido eficiente por rango** (fechas).
ArbolBPlus	Índice de **Búsqueda por Género** (Biblioteca).	Similar al Árbol B, pero con la característica clave de que **todas las claves de datos están en las hojas**, y estas hojas están **enlazadas secuencialmente**. Esto es vital para el recorrido eficiente de todos los libros dentro de un mismo género.

Cola	**Flujo Logístico** (colaIngreso, colaTraspaso, colaSalida).	Su política **FIFO** modela perfectamente el flujo de trabajo secuencial, donde los libros deben ser procesados en el orden en que llegan a la estación de trabajo.
GrafoBibliotecas	**Red Logística** (Topología de Bibliotecas).	El TDA fundamental para modelar la relación compleja de transporte (nodos/bibliotecas, aristas/conexiones) y aplicar algoritmos de ruta como **Dijkstra**.

Análisis de Complejidad Big-O (Algoritmos y Operaciones)

El siguiente análisis detalla la complejidad temporal de las operaciones clave implementadas manualmente en el sistema, clasificándolas en sus respectivos TADs.

AVL (Búsqueda por Título)

Operación	Complejidad Promedio	Complejidad Peor Caso	Justificación
Búsqueda (buscarPorTitulo)	O(log n)	O(log n)	La propiedad de **auto-balanceo** (altura log n) garantiza una búsqueda rápida y consistente.
Inserción (insertar)	O(log n)	O(log n)	La inserción requiere **O(\log n)** para encontrar la posición, más un tiempo constante para el **re-balanceo** (rotaciones simples o dobles).
Recorrido Inorden	O(n)	O(n)	Debe visitar cada uno de los n nodos exactamente una vez para obtener el listado alfabético.

Tabla Hash (Búsqueda por ISBN)

Operación	Complejidad Promedio	Complejidad Peor Caso	Justificación
Búsqueda (obtener)	O(1)	O(n)	En promedio, el acceso es constante. El peor caso ocurre si hay una **colisión extrema** y todos los n elementos caen en la misma cubeta (lista enlazada).
Inserción (insertar)	O(1)	O(n)	Depende del factor de carga. Si el **redimensionamiento** es necesario, puede tomar O(n) para re-hashear todos los elementos.
Eliminación	O(1)	O(n)	Similar a la inserción y búsqueda, depende de la dispersión de los elementos.

Árboles B y B+ (Búsqueda por Fecha/Género)

Sea n el número total de claves y m el orden del árbol.

Operación	Complejidad	Justificación
Búsqueda (buscarPorRango)	O(log_m n)	La altura de un árbol B o B+ es extremadamente baja (log_m n), haciendo que la búsqueda sea muy eficiente.
Inserción	O(\log_m n)	El número de accesos a disco (o nodos en memoria) es logarítmico. Las divisiones (splits) se propagan hacia arriba, manteniendo el límite logarítmico.

Recorrido Secuencial (B+)	O(k)	Una vez que se encuentra la primera hoja, el recorrido de k elementos subsecuentes es extremadamente rápido (lineal O(k)) debido al enlace de las hojas.

Grafo y Algoritmos (Logística)

Operación	**TAD / Algoritmo**	**Complejidad**	**Justificación**
Búsqueda de Ruta	RutaDijkstra	O(V^2) o O(log V)	Usando una matriz de adyacencia o un **Heap de Prioridad** (si la implementación lo usa), donde V son los vértices (Bibliotecas) y E son las aristas (Conexiones). Dado que es una red pequeña, O(V^2) es aceptable.
Agregar/Eliminar Vértice	GrafoBibliotecas	O(1)$(promedio)	La operación se delega a la **TablaHash** interna para acceder rápidamente al vértice por ID.
Agregar Arista	Vertice / ListaAdyacencia	O(1)	La inserción en una **Lista Enlazada Simple** (ListaAdyacencia) es constante al insertar al principio.

Ordenamientos (Algoritmos de Ordenamientos)

Esta tabla detalla la complejidad temporal de los algoritmos de ordenamiento implementados, que son utilizados por el módulo **PanelOrdenamientos** para medir el rendimiento. Sea n el número de libros a ordenar.

Algoritmo	**Complejidad Promedio**	**Complejidad Peor Caso**	**Funcionamiento y Justificación**
bubbleSort	O(n^2)	O(n^2)	**Funcionamiento:** Recorre repetidamente la lista, comparando elementos adyacentes e intercambiándolos si están en el orden incorrecto. Los elementos "flotan" a su posición final. **Justificación:** Es un algoritmo simple, pero ineficiente debido a las muchas comparaciones e intercambios, lo que lo hace ideal como línea base **ineficiente** para la comparativa.
selectionSort	O(n^2)	O(n^2)	**Funcionamiento:** Divide la lista en una parte ordenada y una desordenada. En cada iteración, encuentra el elemento **mínimo** en la parte desordenada y lo **intercambia** con el primer elemento de esa parte. **Justificación:** Aunque siempre toma O(n^2) comparaciones, minimiza el número de **intercambios** (solo n intercambios totales), lo que podría hacerlo ligeramente mejor que Bubble Sort en ciertos escenarios de alto costo de intercambio.

insertionSort	O(n^2)	O(n^2)	**Funcionamiento:** Construye la lista ordenada elemento por elemento. Recorre la lista, tomando un elemento y **desplazando** los elementos ya ordenados que son mayores que él, para insertarlo en su posición correcta. **Justificación:** Muy eficiente para listas **casi ordenadas** (O(n)) o para listas muy pequeñas. En el peor caso (orden inverso) es O(n^2).
shellSort	O(n \log^2 n)	O(n^2)	**Funcionamiento:** Es una mejora de insertionSort. Ordena elementos que están lejos entre sí (separados por un gap decreciente) antes de hacer el ordenamiento final con gap=1. **Justificación:** Es significativamente **más rápido** que los tres anteriores para listas de tamaño moderado, ya que permite que los elementos muy desordenados se muevan rápidamente a través de la lista.
quickSort	O(n \log n)	O(n^2)	**Funcionamiento:** Algoritmo de **"divide y vencerás"**. Selecciona un elemento **pivote**, y particiona la lista para que todos los elementos menores al pivote queden a su izquierda y los mayores a su derecha. Luego se llama recursivamente a las sublistas. **Justificación:** Es el algoritmo de **comparación más rápido** en el caso promedio. Su complejidad O(n \log n) lo hace ideal para ordenar grandes volúmenes de libros en la prueba de rendimiento.