PL1 ESTRUCTURA DE DATOS

- Rayan Bentaleb Zaki 48160981Q
- Óscar López Rojo 03492385L

ÍNDICE

1. Definición de los TAD's utilizados en el proyecto

TAD Nodo

- **Descripción:**
  Representa la **unidad básica** usada para construir estructuras enlazadas (como pilas y colas).
  Cada nodo almacena:
  - Un **dato** del tipo `Pedido`.
  - Un **puntero** al siguiente nodo (`siguiente`).
- **Función principal:**
  Permite **enlazar elementos entre sí** para formar listas, pilas o colas dinámicas.

TAD Pila

- **Descripción:**
  Estructura de datos **LIFO (Last In, First Out)**, donde el último elemento en entrar es el primero en salir.
- **Componentes y operaciones:**
  - `cima`: puntero al nodo superior de la pila.
  - `apilar(Pedido p)`: agrega un elemento en la cima.
  - `desapilar()`: elimina y devuelve el elemento en la cima.
  - `esVacia()`: comprueba si la pila está vacía.
  - `mostrar()`: imprime el contenido de la pila desde la cima.
  - `getNumeroElementos()`: devuelve cuántos elementos hay en la pila.
- **Uso principal:**
  Almacenar los libros dentro de las cajas, para poder operar con ellas y mostrarlas siguiendo un orden FIFO.

TAD Cola

- **Descripción:**
  Estructura de datos **FIFO (First In, First Out)**, donde el primer elemento en entrar es el primero en salir.
- **Componentes y operaciones:**

  - `primero`: puntero al primer nodo (frente de la cola).
  - `ultimo`: puntero al último nodo (final de la cola).
  - `encolar(Pedido p)`: agrega un elemento al final.
  - `desencolar()`: elimina y devuelve el elemento del frente.
  - `esVacia()`: verifica si la cola está vacía.
  - `mostrar()` y `mostrarConFormatoDeTabla()`: muestran el contenido de la cola.

- **Uso principal:**
  Almacenar los pedidos para operar con ellos y posteriormente mostrarlos de manera cómoda siguiendo un orden FIFO.

2. Explicación del funcionamiento del programa y de los métodos/funciones implementadas.

El programa simula un sistema de gestión de pedidos para una editorial. La lógica principal está encapsulada dentro de la clase `Editorial`, que gestiona el flujo de pedidos a través de diferentes fases (colas) y su empaquetado final (pilas). El archivo `main.cpp` actúa como el controlador, presentando un menú al usuario y llamando a los métodos correspondientes de la clase `Editorial`.

La implementación se basa en Tipos Abstractos de Datos (TAD) fundamentales, como la Pila y la Cola, implementados dinámicamente mediante nodos enlazados.

TADs y Estructuras de Datos Base (editorial.h y editorial.cpp)

La base del sistema se construye sobre las estructuras `Pedido`, `Libro`, y las clases `Nodo`, `Pila` y `Cola`.

Estructuras de Datos

- **`struct Pedido`**: Almacena toda la información de un único pedido (ID de librería, ID de pedido, código de libro, materia, unidades y estado). Es la unidad de información que se mueve a través del sistema.

- **`struct Libro`**: Representa un título en el catálogo de la editorial. Almacena su código, materia y el `stock` disponible. Se utiliza un array de esta estructura (`catalogo[MAX_TITULOS]`) para el control de stock.

class Nodo

Es el componente básico de las listas enlazadas. Cada `Nodo` almacena un `Pedido` (`dato`) y un puntero al siguiente nodo (`siguiente`). Se reutiliza tanto para la Pila como para la Cola.

class Pila

Implementa una estructura LIFO (Last-In, First-Out) utilizada para modelar las `cajas` de envío.

- **Miembros**: `Nodo *cima` (puntero al nodo superior).
- **`Pila()` (Constructor)**: Inicializa la pila como vacía (`cima = nullptr`).
- **`~Pila()` (Destructor)**: Libera toda la memoria dinámica, vaciando la pila mediante llamadas sucesivas a `desapilar()`.
- **`apilar(Pedido p)`**: Crea un nuevo `Nodo` con el pedido `p` y lo añade en la `cima` de la pila.
- **`desapilar()`**: Elimina el nodo de la `cima`, libera su memoria y devuelve el `Pedido` que contenía.
- **`esVacia()`**: Devuelve `true` si `cima` es `nullptr`.
- **`getNumeroElementos()`**: Recorre la pila y devuelve el número total de nodos. Es fundamental para saber cuándo una caja ha alcanzado su capacidad (`CAP_CAJA`).
- **`mostrar()`**: Recorre la pila desde la `cima` hasta el fondo, imprimiendo cada pedido en formato de tabla.

class Cola

Implementa una estructura FIFO (First-In, First-Out) utilizada para modelar cada una de las fases de procesamiento: `Iniciado`, `Almacén`, `Imprenta` y `Listo`.

- **Miembros**: `Nodo *primero` y `Nodo *ultimo` (punteros al inicio y fin de la cola).
- **`Cola()` (Constructor)**: Inicializa la cola como vacía (`primero` y `ultimo` a `nullptr`).
- **`~Cola()` (Destructor)**: Libera la memoria llamando a `desencolar()` hasta que la cola esté vacía.

- **`encolar(Pedido p)`**: Crea un nuevo `Nodo` y lo añade al final de la cola, actualizando el puntero `ultimo`. Maneja correctamente el caso de encolar en una cola vacía.
- **`desencolar()`**: Elimina el `Nodo` del `primero` de la cola, libera su memoria y devuelve el `Pedido`. Maneja el caso de desencolar el último elemento.
- **`esVacia()`**: Devuelve `true` si `primero` es `nullptr`.
- **`mostrarConFormatoDeTabla()`**: Recorre la cola desde `primero` hasta `ultimo`, imprimiendo cada pedido en formato de tabla.

Clase Principal (Editorial)

Esta clase encapsula toda la lógica de la simulación y gestiona el estado del sistema.

Miembros Privados

- **Colas**: `colaIniciado`, `colaAlmacen`, `colaImprenta`, `colaListo`. Cada una almacena los pedidos que están en esa fase específica del proceso.
- **Pilas**: `Pila cajas[LIBRERIAS]`. Un array de pilas donde cada índice `i` representa la caja de envío para la librería con `id_editorial = i`.
- **Stock**: `Libro catalogo[MAX_TITULOS]`. Un array que funciona como el sistema de control de stock.
- **`ultimoIdPedido`**: Un contador entero para generar IDs de pedido únicos y secuenciales (ej. "P21508", "P21509"...).

Métodos Privados (Auxiliares)

- **`inicializarCatalogo()`**: Se llama desde el constructor. Rellena el array `catalogo` con `MAX_TITULOS` libros, asignándoles una materia, un `cod_libro` aleatorio (usando `generarCodigoLibroAleatorio`) y un stock inicial aleatorio.
- **`generarCodigoLibroAleatorio()`**: Genera un string con el formato alfanumérico especificado (ej. "963K76").
- **`buscarLibro(string cod_libro)`**: Busca un libro en el `catálogo` por su código. Devuelve el *índice* del libro en el array si lo encuentra, o `-1` si no existe.
- **`procesarCaja(int id_libreria)`**: Simula el envío de una caja llena. Vacía completamente la pila `cajas[id_libreria]` llamando a `desapilar()` repetidamente e informa al usuario.

Métodos Públicos (Menú)

- **`Editorial()` (Constructor)**: Inicializa la semilla aleatoria (`srand`), establece el `ultimoIdPedido` inicial y llama a `inicializarCatalogo()` para preparar el stock.
- **`generarPedidos(int n)` (Opción 1)**: Crea n nuevos pedidos. Para cada pedido, genera datos aleatorios (ID de librería, libro del catálogo, unidades) y un ID de pedido único. Establece su estado a "Iniciado" y lo añade a `colaIniciado` usando `encolar()`.
- **`mostrarPedidosGenerados()`**: Método de apoyo a la Opción 1. Llama a `colaIniciado.mostrarConFormatoDeTabla()` para que el usuario vea los pedidos que acaba de crear.
- **`ejecutarPasoSimulacion()` (Opción 2)**: Es el corazón de la simulación. Procesa un número limitado de pedidos (`N_PEDIDOS_PASO`) de cada fase. El procesamiento se ejecuta en **orden inverso** (de la última fase a la primera) para evitar que un mismo pedido avance múltiples fases en un solo paso.
    - **Fase 1 (Listo -> Caja)**: Desencola de `colaListo`. Apila el pedido en la `caja[p.id_editorial]` correspondiente. Si la pila (caja) alcanza `CAP_CAJA`, llama a `procesarCaja()`.
    - **Fase 2 (Imprenta -> Listo)**: Desencola de `colaImprenta`. Incrementa el stock del libro (`TAM_LOTE`) y encola el pedido en `colaListo`.
    - **Fase 3 (Almacén -> ?)**: Desencola de `colaAlmacen`. Comprueba el stock (`buscarLibro`).
        - Si hay stock suficiente: Reduce el stock, actualiza el estado a "Listo" y encola en `colaListo`.
        - Si no hay stock: Actualiza el estado a "Imprenta" y encola en `colaImprenta`.
    - **Fase 4 (Iniciado -> Almacén)**: Desencola de `colaIniciado`, actualiza el estado a "Almacén" y encola en `colaAlmacen`.
- **`mostrarEstadoSistema()` (Opción 3)**: Proporciona una vista completa del sistema.
    - Muestra el contenido de las cuatro colas de procesamiento (`Iniciado`, `Almacén`, `Imprenta`, `Listo`).
    - Muestra el estado actual del `catálogo` (stock de cada libro).
    - Muestra el contenido de todas las `cajas` (pilas) que no están vacías.
- **`verContenidoCaja(int id_libreria)` (Opción 4)**: Muestra el contenido detallado de una caja específica. Valida el ID de la librería y luego llama a `cajas[id_libreria].mostrar()`.

Programa Principal (main.cpp)

- **`main()`**: Punto de entrada del programa.
    - Crea una única instancia de la clase: `Editorial miEditorial;`.
    - Inicia un bucle `do-while` que se ejecuta mientras la `opcion` del usuario no sea 0 (Salir).
    - Dentro del bucle, llama a `mostrar_menu()` y espera la entrada del usuario.
    - Implementa un **manejo robusto de errores de entrada**: si el usuario introduce algo que no es un número, limpia el *stream* de `cin` para evitar un bucle infinito.
    - Utiliza una estructura `switch` para dirigir la `opcion` del usuario al método público correspondiente del objeto `miEditorial`.
- **`mostrar_menu()`**: Función global simple que imprime por pantalla las opciones disponibles para el usuario.

3. Problemas encontrados durante el desarrollo de la práctica y solución adoptada

Durante la implementación de la práctica, nos enfrentamos a varios desafíos de diseño y lógica que requirieron una reevaluación de nuestro enfoque inicial.

1. Ineficiencia en el diseño del Control de Stock

- **Problema:** Inicialmente, consideramos la idea de modelar el catálogo de libros (control de stock) utilizando una estructura de datos dinámica, como una cola o una lista, para ser coherentes con el resto de la práctica. Sin embargo, nos percatamos de que esta decisión presentaba un problema de eficiencia significativo. En la **Fase 3 (Almacén -> Listo/Imprenta)**, es necesario consultar el stock disponible para *cada* pedido que se procesa. Si el stock estuviera en una cola, cada consulta requeriría recorrer la estructura (O(n)) o un proceso destructivo de desencolar y volver a encolar todos los elementos.
- **Solución Adoptada:** Decidimos implementar el control de stock mediante un **array de tamaño fijo** (`Libro catalogo[MAX_TITULOS]`). Aunque la especificación indicaba que "se deja a elección del alumno", esta solución es mucho más eficiente para este caso de uso. La búsqueda de un libro por su `cod_libro` (`buscarLibro`) sigue siendo O(n), pero "n" es una constante pequeña (`MAX_TITULOS = 12`). La ventaja principal es el acceso directo para la

lectura y actualización del stock, que se convierte en una simple asignación en un índice del array.

2. Falta de Encapsulación y Organización del Código

- **Problema:** Comenzamos el desarrollo implementando la lógica de forma procedural. A medida que avanzamos y teníamos funcionales las opciones 0 (Salir), 1 (Generar Pedidos) y 3 (Mostrar Estado), el código se volvió difícil de gestionar. Todas las estructuras de datos (las 4 colas de fases, el array de pilas para las cajas y el catálogo) tenían que ser accesibles globalmente o pasarse como parámetros a múltiples funciones, generando un alto acoplamiento.
- **Solución Adoptada:** A mitad del desarrollo, decidimos **refactorizar el código** para adoptar un enfoque orientado a objetos. Creamos la clase `Editorial`. Todas las estructuras de datos y variables de estado (colas, pilas, catálogo, `ultimoIdPedido`) se convirtieron en **miembros privados**. Toda la lógica de la simulación (opciones 1, 2, 3 y 4) se implementó como **métodos públicos** (`generarPedidos`, `ejecutarPasoSimulacion`, etc.). Esta solución mejoró drásticamente la organización y la legibilidad, simplificando `main.cpp` a un simple gestor del menú.

3. Lógica del Paso de Simulación y el "Efecto Cascada"

- **Problema:** Al diseñar la función `ejecutarPasoSimulacion`, el primer impulso fue procesar las fases en orden lógico (Fase 4 -> Fase 3 -> ...). Esto creaba un "efecto cascada" no deseado: un pedido recién creado podía moverse de `Iniciado` a `Almacén`, ser procesado de `Almacén` a `Listo` y de `Listo` a `Caja` en *una única ejecución* de la Opción 2. Esto no reflejaba un "paso" de simulación realista, donde los pedidos deben esperar en cada fase.
- **Solución Adoptada:** Implementamos el procesamiento de fases en **orden inverso** (de la última a la primera). El método `ejecutarPasoSimulacion` primero procesa `colaListo` (Fase 1), luego `colaImprenta` (Fase 2), luego `colaAlmacen` (Fase 3) y finalmente `colaIniciado` (Fase 4). De esta forma, un pedido que avanza (p.ej., de `Almacén` a `Listo`) no será procesado por la fase siguiente (`Listo` a `Caja`) hasta la *siguiente* vez que el usuario ejecute el paso 2.

4. Gestión de Memoria Dinámica y Fugas de Memoria

- **Problema:** Dado que las clases `Pila` y `Cola` se basan en memoria dinámica (uso de `new Nodo(...)`), existía un riesgo claro de fugas de memoria (memory leaks). Si no se gestionaba correctamente, cada `desapilar()` o `desencolar()`

dejaría un Nodo huérfano en memoria. Además, al terminar el programa, todos los pedidos restantes en las colas y pilas no liberarían su memoria asociada.

- **Solución Adoptada:** Se implementó un manejo cuidadoso de la memoria en tres puntos:
    - **Métodos de extracción:** En `desapilar()` y `desencolar()`, nos aseguramos de guardar una referencia temporal al nodo (`Nodo *aux = ...`) antes de re-enlazar los punteros de la estructura, y finalmente usar `delete aux;` para liberar explícitamente la memoria de ese nodo.
    - **Destructores:** Se implementaron destructores (`~Pila()` y `~Cola()`) que vacían la estructura llamando a `desapilar`/`desencolar` en bucle hasta que `esVacia()` sea `true`.
    - **Encapsulación:** Dado que la `Editorial` gestiona las colas y pilas, sus propios destructores se encargarán de llamar a los destructores de sus miembros (`Pila` y `Cola`), asegurando que toda la memoria se libere al salir del programa.

5. Generación de Identificadores de Pedido Únicos

- **Problema:** La especificación requiere que `id_pedido` sea "único". Nuestra primera idea fue generarlo aleatoriamente, de forma similar al `cod_libro`. Sin embargo, la generación aleatoria pura puede producir colisiones (dos pedidos con el mismo ID), lo que violaría el requisito de unicidad y es un error común.
- **Solución Adoptada:** Descartamos la aleatoriedad para el `id_pedido` y optamos por un **generador de secuencia simple**. Se añadió una variable miembro privada a la clase `Editorial`: `int ultimoIdPedido`. Al generar pedidos, este contador se utiliza para crear el ID (ej. "P" + `to_string(ultimoIdPedido)`) y luego se incrementa (`ultimoIdPedido++`). Esto garantiza que cada ID de pedido sea único de forma sencilla y robusta.