Práctica 2:
Estructuras de Datos

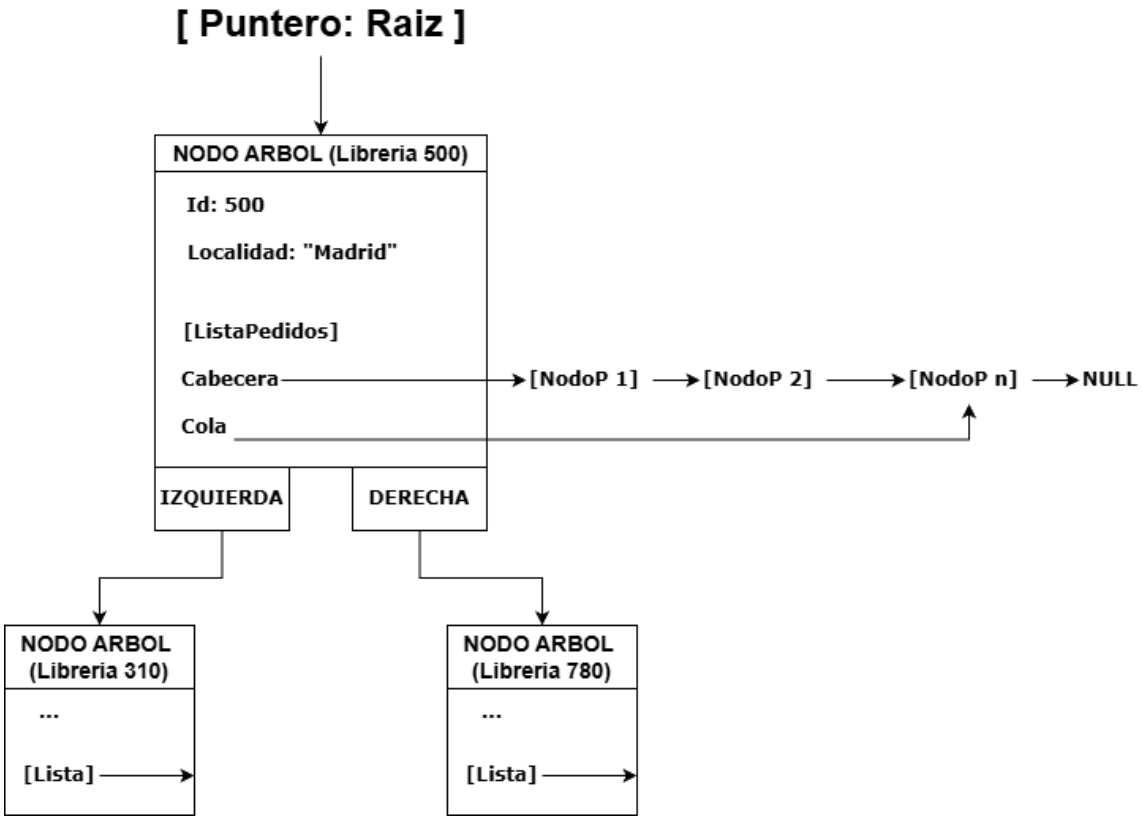
Simulación de Centro de Control de Editorial.

Rayan Bentaleb Zaki **DNI: 48160981Q**

Óscar López Rojo **DNI:03492385L**

ÍNDICE

1. Dibujo de los TADs y Estructuras de Datos



1.1. Árbol Binario de Búsqueda (Clase ABBLibrerias)

Estructura principal utilizada para almacenar y gestionar las **Librerías**. Se ha elegido un ABB para optimizar las búsquedas por `id_libreria` (complejidad logarítmica O(log n) en caso promedio), lo cual es superior a una lista lineal dado el volumen potencial de librerías.

1.2. Lista Enlazada Simple (Clase ListaPedidos)

Cada nodo del árbol encapsula una lista enlazada para almacenar los **Pedidos**. Se ha optado por una estructura dinámica con punteros a `cabecera` y `cola` para permitir una inserción eficiente O(1) al final de la lista y un crecimiento dinámico según la demanda de cada librería.

2. Explicación del Funcionamiento y Métodos Implementados

El sistema ha sido diseñado como una aplicación de consola en C++ que gestiona estructuras de datos dinámicas anidadas para simular el flujo de trabajo de una editorial. El funcionamiento se divide en una fase de inicialización automática y un bucle de gestión interactiva.

2.1. Inicialización y Flujo Principal (main.cpp)

Al iniciar el programa, se ejecuta una rutina de generación automática para poblar las estructuras de datos, garantizando que el sistema no arranque vacío:

1. **Generación de Librerías:** Se instancian `N_LIBRERIAS` (10) objetos de tipo `Libreria` con datos aleatorios. Para garantizar la consistencia, se utiliza un array auxiliar `idsValidos[]` que almacena los IDs generados; esto permite verificar duplicados antes de la inserción y servirá después para asignar pedidos correctamente.
2. **Inserción en el Árbol:** Cada librería generada se inserta en el objeto `arbolReal` (instancia de `ABBLibrerias`), ordenándose automáticamente por su ID.
3. **Generación y Distribución de Pedidos:** Se generan `N_PEDIDOS` (30). A cada pedido se le asigna un `id_libreria` seleccionado aleatoriamente del array `idsValidos`, asegurando que cada pedido tenga un destinatario existente. Posteriormente, el método `arbolReal.distribuirPedido()` enruta cada pedido a su librería correspondiente.
4. **Bucle Interactivo:** El programa entra en un ciclo `do-while` que presenta un menú de opciones y procesa la entrada del usuario mediante una estructura `switch`. Se ha implementado limpieza de buffer (`cin.ignore`) y pausas en pantalla para mejorar la usabilidad.

2.2. Implementación de la Clase ListaPedidos

Esta clase gestiona una **Lista Enlazada Simple** de pedidos. Se ha diseñado para ser eficiente en la inserción y flexible en el borrado.

- **Atributos:**
    - `cabecera`: Puntero al primer nodo de la lista.
    - `cola`: Puntero al último nodo. Su inclusión permite realizar inserciones al final con coste computacional O(1), evitando recorrer la lista completa cada vez.

- **Métodos Principales:**
    - **`insertar(Pedido p)`:** Crea un nuevo `NodoPedido` en el *heap* (memoria dinámica). Si la lista está vacía, actualiza `cabecera` y `cola`. Si no, enlaza el nuevo nodo al `sig` de la `cola` actual y actualiza el puntero `cola`.
    - **`borrar(const char* id)`:** Recorre la lista buscando el ID especificado. Implementa la lógica de "puenteado" de punteros para desconectar el nodo a eliminar de la cadena antes de liberar su memoria con `delete`. Gestiona casos especiales como el borrado del primer elemento (`cabecera`) o del último (actualización de `cola`).
    - **`extraer(const char* id)`:** Esencial para la funcionalidad de "Mover Pedido". Busca y desconecta un nodo de la lista, pero **no libera su memoria**. Devuelve el puntero al nodo desconectado para que pueda ser insertado en otra lista diferente.
    - **`mostrar()`:** Recorre la lista secuencialmente. Utiliza la librería `<iomanip>` con los manipuladores `setw`, `left` y `right` para imprimir los datos en formato de tabla alineada.

2.3. Implementación de la Clase ABBLibrerias

Esta clase gestiona un **Árbol Binario de Búsqueda** donde cada nodo contiene una librería y, por ende, su propia lista de pedidos.

- **Lógica Recursiva:** La mayoría de las operaciones (`insertar`, `borrar`, `buscar`, `mostrar`) se implementan mediante una función pública (interfaz) que llama a una función privada recursiva que recibe el nodo actual como parámetro.

- **Métodos Principales:**
    - **`insertar(Libreria lib):`** Recorre el árbol comparando el ID de la nueva librería con el del nodo actual. Si es menor, desciende por la rama izquierda (`izq`); si es mayor, por la derecha (`der`). Si encuentra una posición nula, crea el nuevo `NodoLibreria`.
    - **`borrar(int id):`** Implementa los tres casos teóricos de eliminación en ABB:
        - **Nodo Hoja:** Se elimina directamente y el puntero del padre se establece a `NULL`.
        - **Un solo hijo:** El puntero del padre se actualiza para apuntar al hijo del nodo eliminado (el abuelo "adopta" al nieto).
        - **Dos hijos:** Se busca el **sucesor in-orden** (el menor del subárbol derecho). Debido a que los nodos contienen una estructura compleja (`ListaPedidos`), no se puede realizar una copia superficial. Se implementó una **copia profunda manual**: se vacía la lista del nodo destino y se copian uno a uno los pedidos de la lista del sucesor. Finalmente, se llama recursivamente a `borrar` para eliminar el nodo sucesor original.
    - **`distribuirPedido(Pedido p):`** Busca la librería destinataria en el árbol (búsqueda binaria). Una vez encontrado el nodo, delega la inserción a la lista interna: `nodo->datos.pedidos.insertar(p)`.
    - **`mostrarEstadisticas():`** Realiza un recorrido completo del árbol. Utiliza estructuras auxiliares (`RegistroConteo`) para agregar los datos de todas las librerías y calcular métricas globales como el libro más vendido o la materia con más unidades solicitadas.

2.4. Validaciones y Robustez

Se han implementado controles específicos para asegurar la estabilidad del programa:

- **Gestión de Memoria:** Los destructores de `ABBLibrerias` y `ListaPedidos` aseguran la liberación en cascada de toda la memoria dinámica al cerrar el programa.
- **Validación de Entrada (Opción 1):** Se utiliza un bucle `do-while` para garantizar que el ID introducido manualmente sea numérico y esté en el rango de 3 cifras (100-999).
- **Consistencia en Generación (Opción 8):** Al generar nuevos pedidos tras un posible borrado de librerías, el sistema verifica mediante `buscar()` si el ID de destino sigue existiendo en el árbol. Si la librería fue borrada, busca otro candidato válido para evitar la pérdida de pedidos.

3. Problemas Encontrados y Soluciones Adoptadas

Durante el ciclo de desarrollo de la práctica, enfrentamos varios problemas técnicos, principalmente derivados de la gestión manual de memoria dinámica y la interacción entre estructuras de datos anidadas. Para ello, implementamos las soluciones de la siguiente manera:

3.1. Eliminación de Nodos con dos hijos en el ABB (Gestión de Memoria)

- **Problema:** El desafío técnico más complejo surgió al implementar la función borrar() en el árbol para el caso de un nodo con dos hijos. El algoritmo estándar que coincide con lo dicho en clase dicta sustituir el contenido del nodo a eliminar por el de su sucesor in-orden y posteriormente eliminar el nodo sucesor.

Sin embargo, dado que cada nodo del árbol (NodoLibreria) encapsula un objeto dinámico (ListaPedidos), una asignación directa de la estructura (nodo->datos = sucesor->datos) provocaba una copia superficial. Esto hacía que ambos nodos

apuntaran a la misma lista en memoria. Al invocar borrar(sucesor), el destructor de la lista liberaba la memoria, dejando al nodo original con punteros colgantes, lo que derivaba en errores de segmentación (Segmentation Fault) al intentar acceder posteriormente a esa librería.

- **Solución:** Se implementó una lógica de **copia profunda manual** dentro del método de borrado. Antes de eliminar el sucesor físico, el algoritmo realiza los siguientes pasos:
    - Vacía explícitamente la lista de pedidos del nodo destino para evitar fugas de memoria.
    - Recorre la lista de pedidos del nodo sucesor.
    - Inserta copias nuevas de cada pedido en la lista del nodo destino.

De esta forma, se garantiza que la integridad de los datos se mantiene tras la eliminación del nodo sucesor.

3.2. Inconsistencia de IDs en la Generación de Pedidos (Opción 8)

- **Problema:** El sistema utiliza un array auxiliar `idsValidos` para asignar pedidos aleatorios a librerías existentes. Se detectó un fallo lógico cuando el usuario eliminaba una librería (Opción 2) y posteriormente solicitaba generar nuevos pedidos (Opción 8). El generador seguía utilizando el ID de la librería eliminada (presente en el array, pero no en el árbol), lo que provocaba que los pedidos asignados a dicha librería se perdieran silenciosamente al no encontrar destinatario en el árbol.
- **Solución:** Se robusteció la lógica en el `main.cpp`. Antes de asignar un pedido a un ID del array, el programa verifica la existencia real de la librería en el árbol mediante el método `buscar()`. Si la librería ha sido eliminada, el sistema descarta ese ID y busca otro candidato válido mediante un bucle de reintentos, asegurando que el 100% de los pedidos generados se distribuyan correctamente.

3.3. Validación de Entrada de Datos y Tipos

- **Problema:** Durante las pruebas del código, se observó que el programa entraba en un bucle infinito si el usuario introducía caracteres no numéricos cuando se solicitaba un ID (entero). Esto ocurría porque el flujo de entrada `cin` entraba en estado de error y no se limpiaba el buffer.

- **Solución:** Se protegieron todas las lecturas de datos críticos con bloques de validación condicional. Se verifica el estado de `cin.fail()`; en caso de error, se limpia el flag de error (`cin.clear()`) y se vacía el buffer de entrada (`cin.ignore()`), solicitando al usuario que introduzca el dato nuevamente. Además, en la inserción manual (Opción 1), se añadió un bucle `do-while` para forzar que el ID esté estrictamente en el rango de 3 cifras (100-999).

3.4. Formateo y Visualización de Datos

- **Problema:** Al mostrar los listados de pedidos, la salida por consola resultaba desalineada y difícil de leer debido a la longitud variable de los campos (IDs, nombres de materias, fechas).

- **Solución:** Se integró la librería `<iomanip>` para formatear la salida. Se utilizaron manipuladores como `setw()` para fijar el ancho de columna y `left/right` para la alineación del texto. Esto permitió generar tablas visualmente ordenadas independientemente de la longitud de los datos contenidos.

4. Reparto de Tareas

Para la realización de esta práctica, el trabajo se ha distribuido de manera equitativa, asignando responsabilidades específicas sobre las estructuras de datos y funcionalidades, aunque la fase de diseño inicial y depuración final se realizó de manera conjunta.

Alumno 1: [Óscar López Rojo]

- **Diseño de Datos Base:** Definición de las estructuras `struct Pedido`, `struct Libreria` y los nodos correspondientes en el archivo de cabecera `ccontrol.h`.

- **Implementación de `ListaPedidos`:** Desarrollo de la clase gestora de la lista enlazada, incluyendo la lógica de punteros para la inserción al final (`cola`) y el borrado de nodos intermedios ("puenteado" de punteros).
- **Generación de Datos:** Implementación de las funciones auxiliares para la generación aleatoria de IDs, fechas y localidades.
- **Interfaz de Usuario (`main.cpp`):** Estructuración del menú principal, implementación del bucle de control y desarrollo de las rutinas de validación de entrada de datos (control de tipos numéricos y rangos de IDs).

Alumno 2: [Rayan Bentaleb Zaki]

- **Implementación de `ABBLibrerias`:** Desarrollo de la clase gestora del Árbol Binario de Búsqueda y de todos sus métodos recursivos privados (`insertar`, `buscar`, `borrar`).
- **Gestión Avanzada de Memoria:** Solución al problema del borrado de nodos con dos hijos, implementando la lógica de **copia profunda** de la lista interna para evitar errores de segmentación.
- **Algoritmos de Análisis:** Desarrollo de la funcionalidad de estadísticas (Opción 7) y búsqueda global de pedidos.
- **Visualización:** Integración de la librería `<iomanip>` para el formateo de tablas y alineación de datos en la salida por consola.

Tareas Comunes:

- Diseño del diagrama de TADs y jerarquía de clases.
- Pruebas de codigo para detectar casos límite (borrado de raíz, listas vacías).
- Redacción de la memoria y documentación del código